# INSTITUTO TECNOLÓGICO DE CHIHUAHUA

**DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN**

# *"EFICIENTIZACIÓN DE REDES NEURONALES PARA INTERNET DE LAS COSAS SOBRE CÓMPUTO EN LA FRONTERA Y EN LA NIEBLA"*

## *TESIS*

**QUE PARA OBTENER EL GRADO DE**

## *MAESTRO EN INGENIERÍA MECATRÓNICA*

**PRESENTA:**


## *ING. EVER ALEJANDRO FLORES ÁVILA*


**DIRECTOR DE TESIS:**
**M.C. ALBERTO PACHECO GONZÁLEZ**

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

**CHIHUAHUA, CHIH., MEXICO.**
**AGOSTO 2019.**

Chihuahua, Chih., a 30 de agosto de 2019.

Dr. María Elena Álvarez-Buylla.
Director de CONACYT.

**At´n. Luis Gil Cisneros**
Dirección de Formación de Científicos y Tecnólogos

P r e s e n t e.

Por este conducto aprovecho la ocasión para saludarlo e informarle que a la fecha he obtenido el Grado de Maestría en Ingeniería Mecatrónica en la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Chihuahua. Motivo por el cual agradezco todo el apoyo brindado por esta Institución que Usted representa, el otorgamiento de esta beca-crédito permitió dedicarme de tiempo completo a la realización de mis estudios de Posgrado y de esta manera lograr el cumplimiento del objetivo principal del convenio establecido.

Sin otro particular por el momento, me es grato quedar de Usted como su seguro servidor, no sin antes reiterar mi agradecimiento. ¡Muchas Gracias!

A t e n t a m e n t e

_____

Ever Alejandro Flores Ávila
Exbecario CONACYT No. 844461

c.c.p M.F. Luis Cardona Chacón.
Jefe de la División de Posgrado e Investigación

*“Si hiciéramos lo que debemos, haríamos milagros”*
*J.M. Silva*

*“Decisiones Escobar, decisiones…”*
*Ricardo Sáenz*

*“Estén atentos; firmes en la fe; sean valientes y esfuércense”*
*1 Co 16:13*

# AGRADECIMIENTOS

A Dios, por amarme primero.

A mis padres, hermanos y amigos a quienes con mucho cariño considero como mi familia, por su apoyo siempre incondicional y por ser el gran motivante para alcanzar este objetivo.

A Alberto Pacheco, por su extraordinario compromiso y dedicación, no sólo al desarrollo de esta tesis, sino también a mi desarrollo profesional e incluso personal; por siempre buscar y encontrar oportunidades que elevaran la calidad de este trabajo; y por su confianza y apoyo estos dos años.

A Mariano Rivera, por invitarme al CIMAT, por todo el tiempo que dedicó y todo el conocimiento que incesantemente transmitió durante mi estadía en Guanajuato para el desarrollo de CNN-PSL.

A Pedro Márquez, Carmen García y Claudia Prieto por sus consejos y seguimiento, desde la propuesta de tesis hasta los últimos cambios en el título. A Rogelio Baray, Isidro Robledo, Salvador Almanza y al resto de maestros que contribuyeron al desarrollo de este trabajo, así como a mi formación académica y profesional.

A Rául, por acompañarme y apoyarme cuando no entendíamos nuestro tema de tesis, así como por sus aportaciones en el desarrollo de las primeras aplicaciones.

A Pablo, por toda su ayuda en el desarrollo de este trabajo, dedicando incluso semanas para resolver los problemas de compatibilidad e implementación que tuvimos sobretodo al final.

A quienes además de ser mis compañeros de posgrado durante estos años, considero también mis amigos. Mariana, Andrés, Daniel, David, Gabriel, Marco y por supuesto Pablo y Rául, por su compañía, sus pláticas, su cariño y por hacer de ese edificio un segundo hogar.

**¡GRACIAS!**

# RESUMEN

EFICIENTIZACIÓN DE REDES NEURONALES PARA INTERNET DE LAS COSAS SOBRE CÓMPUTO EN LA FRONTERA Y EN LA NIEBLA

Ever Alejandro Flores Ávila
Maestro en Ingeniería Mecatrónica
División de Estudios de Posgrado e Investigación del
Instituto Tecnológico de Chihuahua
Chihuahua, Chih. Agosto 2019.
Director de Tesis: M.C. Alberto Pacheco González

En nuestros días, el 90% de los datos del internet de las cosas se procesan en la nube. Para 2025 se espera que el 75% sea procesado usando otras arquitecturas. Sin embargo, un problema abierto es cómo ejecutar modelos de aprendizaje profundo a partir de los limitados recursos de cómputo y almacenamiento en arquitecturas recientes como el cómputo en la frontera. El presente trabajo aborda este reto eficientizando redes neuronales para que sean ejecutadas en dispositivos IoT de recursos limitados, mediante la reducción de clases, parámetros y tiempo de ejecución en un modelo de red neuronal para detectar personas, aplicando técnicas de adaptación de poda (*pruning*), transferencia de aprendizaje (*transfer learning*) y cuantización. Se propone además una técnica de poda junto con su planteamiento matemático inédito, denominada CNN-PSL basada en un enfoque dependiente de datos empleando regularización con norma L1 para podar filtros. Esta técnica, junto con las demás, fueron aplicadas al modelo de detección de objetos TinyYOLOv2 con el fin de derivar un modelo más eficiente de detección de personas para desarrollar un sistema de control de iluminación de un aula inteligente con cómputo en la frontera y en la niebla. Solucionando problemas de interoperabilidad, tanto del modelo original como del modelo adaptado fueron probados en nueve configuraciones distintas (CPU vs NPU), en diferentes plataformas de cómputo basadas en dispositivos iPad, Intel NUC, Raspberry Pi 3 y aceleradores neuronales Intel Movidius. Se validó la eficientización del modelo adaptado respecto al original en cada una de las configuraciones de prueba, obteniendo como resultados una compresión de hasta el 54.13% y una aceleración promedio del 45%, con una pérdida de precisión del 3.51%. Adicional a esos resultados, para la plataforma iOS se realizó una cuantización adicional logrando una compresión total del 88% con respecto al modelo original.

# ÍNDICE

## LISTA DE FIGURAS

## LISTA DE TABLAS

# I. INTRODUCCIÓN Y PLANTEAMIENTO DEL PROBLEMA

## 1.1. Importancia, Tendencias e Impacto de las Áreas de Investigación.

En los últimos años se ha acentuado el crecimiento del uso del Internet de las Cosas (*Internet of Things*, IoT). Según la organización GSMA Intelligence (2018), se estima que para 2025 estén conectados a Internet a nivel mundial alrededor de 25 mil millones de dispositivos IoT, y para entonces resulte para las personas como la industria y las organizaciones algo normal y rutinario utilizar directa o indirectamente dispositivos IoT (Andri et al., 2018).

Por otro lado, de forma análoga, las aplicaciones de la Inteligencia Artificial (*Artificial Intelligence*, AI) se extienden también a un gran ritmo, con expectativas de adopción en un 35% de las compañías a nivel global para 2030 (Bughin, Seong, Manyika, Chui, & Joshi, 2018). Siendo una de las ramas de AI, el Aprendizaje Automático (*Machine Learning*, ML) tiene actualmente una industria global con una tasa compuesta anual de crecimiento (CAGR) del 42%. Destacando de las demás técnicas de aprendizaje automático, Aprendizaje Profundo (*Deep Learning*, DL) posee un mercado de aplicaciones en EE.UU. con proyección de crecimiento de $100 millones de dólares en 2018 a $935 millones para 2025 (Fakhruddin, 2018). Además, existe un creciente interés académico, los artículos de AI publicados en arXiv en 2018 fueron tres veces mayores a los de 2015 (Hao, 2019) y en Scopus, el porcentaje de trabajos de AI que abordan aprendizaje automático creció de un 28% en 2010 a 56% en 2017 (Shoham et al., 2018).

Tanto IoT como AI pueden ser tecnologías de soporte para la denominada Industria 4.0 (Geissbauer, Lübben, Schrauf, & Pillsbury, 2018), cuyo mercado creció a un CAGR de 14.9% en 2018 (Zion Market Research, 2018) ofreciendo oportunidades para crear productos, servicios, empleos y modelos de negocios basados en tecnologías digitales inteligentes (Deloitte, 2018; PwC, 2016). Otro campo de aplicación son las Ciudades Inteligentes

(GlobeNewswire, 2018), incorporando dispositivos inteligentes a los sistemas urbanos y logrando mejorar indicadores de calidad de vida hasta en un 30% (Woetzel et al., 2018).

En general, por cuestión pragmática, la primera generación de aplicaciones IoT han sido implementadas a través del Cómputo en la Nube (*Cloud Computing*) (Belgaum et al., 2017), pero en realidad, las aplicaciones de IoT tienen requerimientos donde las capacidades de la nube pueden resultar insuficientes, como es el caso del procesamiento en tiempo real y las restricciones de conectividad. Es por ello que las tecnologías derivadas del Cómputo en la Frontera (*Edge Computing*) y el Cómputo en la Niebla (*Fog Computing*) han tomado gran participación recientemente (Naha et al., 2018). Para el cómputo en la frontera según PRNewswire (2019), se tiene proyectado un CAGR de 34% para el período de 2019-2024, además de ser el término más buscado en Google Scholar en comparación con otras arquitecturas de cómputo similares, seguido de cerca por el cómputo en la niebla, cuyas búsquedas han aumentado casi tres veces de 2010 a 2017 (Naha et al., 2018).

En este trabajo es de particular interés la convergencia de todas las tendencias anteriormente mencionadas, que serán referidas bajo el término compuesto IoT+DL (PRNewswire, 2019), incorporadas a las arquitecturas de cómputo en la frontera y la niebla. La tendencia conjunta IoT+DL en áreas como Industria 4.0 y ciudades inteligentes según un estudio de Pemberton (2017), estima que para 2022, el 80% de aplicaciones IoT tengan al menos un componente de AI. Resulta entonces natural que un sistema IoT produzca grandes volúmenes de datos, mismos que pueden analizar los algoritmos de aprendizaje automático de AI, por lo que no resulta casual que IoT+DL esté entre las tres mejores tendencias tecnológicas estratégicas de los últimos años (Mohammadi et al., 2018).

## 1.2. Problema a Resolver.

Como se mencionó, actualmente han comenzado a converger diversas tendencias para dotar en los próximos años a un mayor número de dispositivos IoT con ciertas capacidades de procesamiento en la frontera para el análisis de grandes cantidades de datos de IoT, perfilándose el aprendizaje automático y el aprendizaje profundo como una de las alternativas para abordar un número creciente de aplicaciones tales como la automatización en la industria 4.0, y atender los retos en la compleja operación de las ciudades y edificios inteligentes, entre otras áreas y dominios de aplicación emergentes para IoT + DL.

Sin embargo, de acuerdo con diversos autores y expertos del área (Cheng, Wang, Zhou, & Zhang, 2017; Wu et al. 2016; Yao et al., 2018), uno de los retos actuales consiste en encontrar cómo ejecutar modelos de aprendizaje profundo en dispositivos de IoT y/o cómputo en la frontera, por lo que existe un campo muy reciente de investigación (Kim et al., 2015) para encontrar nuevas formas de simplificar o reducir la gran complejidad de dichos modelos DL que son procesados actualmente, por dicha razón, dentro de las enormes infraestructuras de cómputo en la nube (Saiyeda & Mir, 2017). Por diversas razones, que se expondrán más adelante, el cómputo en la nube no siempre es la mejor solución para los requerimientos de muchas posibles aplicaciones de IoT (Dey & Mukherjee, 2018).

Esta tesis se abocó a explorar una forma de adaptar un modelo DL pre-entrenado para detectar objetos, con el objetivo de realizar la etapa de inferencia en dispositivos con limitada capacidad de procesamiento y almacenamiento, que son propias del cómputo en la niebla y el cómputo en la frontera. Para validar la solución se desarrolló un control de iluminación para un aula inteligente, procesando directamente el video de una cámara de video de bajo costo ubicada en el aula con el objetivo de detectar la ubicación de personas dentro de varias regiones de iluminación, para activar o no de forma directa e inmediata la iluminación en cada una de dichas regiones.

## 1.3. Hipótesis de la Investigación.

En la actualidad, el cómputo en la nube es el estándar por defecto (*de facto*) para entrenar y ejecutar modelos de aprendizaje profundo. Esta opción puede ser válida pero impráctica, sobretodo por los tiempos de respuesta para implementar el sistema de iluminación inteligente utilizando modelos de aprendizaje profundo para detectar personas a partir de flujos de video tomados directamente de una videocámara, motivo de esta tesis. Como se mencionó anteriormente, el problema y objetivo del presente trabajo consiste en investigar y aplicar técnicas para reducir el consumo de memoria, incrementar la velocidad de procesamiento y disminuir el consumo de energía, de tal forma que sea posible realizar la etapa de inferencia del modelo de aprendizaje profundo en dispositivos con limitados recursos de cómputo y con una precisión adecuada para efectuar un control de iluminación en tiempo real.

En la siguiente sección se detallan los requerimientos del sistema y a continuación se establece la hipótesis de la presente tesis:

"Modificando una red neuronal convolucional pre-entrenada para detectar objetos es posible lograr que sea ejecutada en un dispositivo con recursos de cómputo limitados, mejorando su rendimiento o consumo de recursos computacionales en la etapa de inferencia para detectar personas en comparación con el rendimiento de la red neuronal original".

Es importante, aclarar que cuando se menciona un “dispositivo con recursos de cómputo limitados”, se refiere a una arquitectura de cómputo en la frontera/niebla, como se especificará más adelante con mayor detalle en los requerimientos del mismo.

## 1.4. Requerimientos del Sistema.

En esta sección se presentan los requerimientos del sistema establecidos para lograr alcanzar el objetivo de la presente tesis. Estos mismos van desde el planteamiento del escenario de validación hasta los detalles técnicos mínimos a satisfacer, explicando en qué consisten y por qué se definieron así.

### 1.4.1. Un aula inteligente como escenario de prueba.

Para el escenario de prueba y la aplicación desarrollada debe ser factible incorporar distintos dispositivos móviles, de IoT, *Edge* y *Fog* capaces de analizar los mismos datos en cada una de las distintas plataformas, ejecutando el mismo modelo DL. Se selecciona un aula como laboratorio de las pruebas debido a la disponibilidad, grado de innovación y costo de implementación dentro del mismo laboratorio donde se llevó a cabo este trabajo y poder escalarse a futuro a varias aulas, o incluso hasta un edificio inteligente.

### 1.4.2. Control de iluminación del aula inteligente.

De las posibles aplicaciones a implementar para el aula inteligente, se selecciona el control de iluminación basado en una videocámara fija por ser actualmente un área en sí misma de desarrollo e innovación, ya que puede proporcionar ahorros importantes en el consumo de energía eléctrica y mejorar el grado de confort para las personas. Otros factores a considerar fueron la viabilidad práctica que ofrece la implementación del prototipo, ya que al realizarse pruebas adaptando luces led comerciales con control de intensidad inalambricamente por WiFi o ZigBee, se pudo verificar el funcionamiento de los modelos DL de forma inmediata y práctica, pues los cambios de iluminación proporcionan una retroalimentación visual rápida y palpable para determinar si el sistema está respondiendo como debería, en vez de trabajar con dispositivos más lentos como un termostato o un proyector. Tampoco se consideraron elementos de seguridad, acceso y encriptado dentro del sistema, como por ejemplo la implementación del

control de una chapa eléctrica para control de acceso en la puerta del laboratorio, debido a que el presente es un prototipo orientado a validar exclusivamente la ejecución, portabilidad, y adaptación de modelos DL, cuyos requerimientos quedan por debajo de los necesarios para una aplicación de seguridad. Aún así, la cobertura de este requerimiento abre la oportunidad para posteriormente incorporar dichas funcionalidades y dispositivos.

### 1.4.3. Detección de personas para el control de iluminación.

El control de iluminación se limita a la detección de personas analizando secuencias de video en tiempo real aplicando arquitecturas y modelos DL pre-entrenados para detectar objetos con un rendimiento mínimo de 2 FPS (Nikouei et al., 2018). En la toma de decisiones del algoritmo de control del sistema de iluminación se limita a determinar si una persona detectada vía modelos DL se ubica o no dentro de una de cuatro regiones de iluminación dentro del campo de visión de una cámara de video fija instalada en el aula (pared lateral con pizarrones de 2m x 5.5m. Ver Figura 1.1). Este prototipo servirá de base para posteriormente, en versiones más avanzadas del prototipo, poder controlar intensidad y color de la iluminación, así como contabilizar el número de personas ubicadas en cada región y datos históricos del sistema, como número de personas detectadas, consumo de energía, etc.

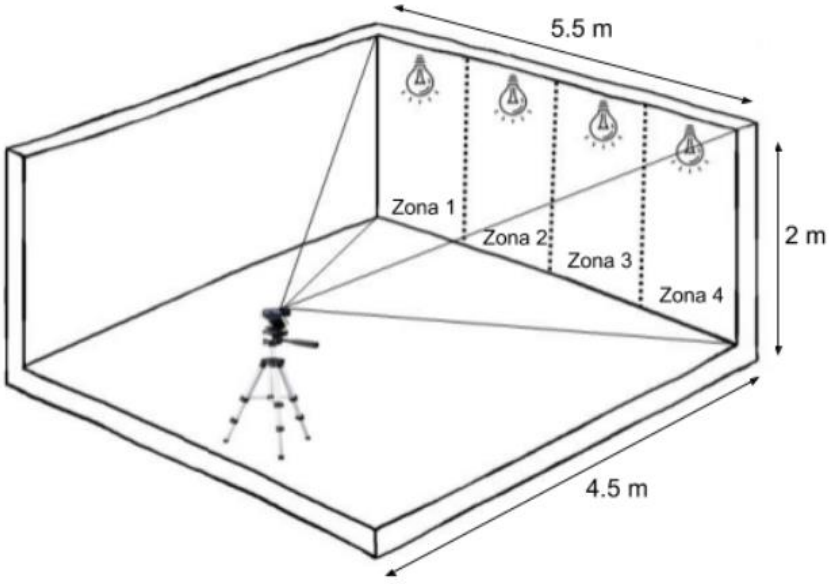


Figura 1.1. Escenario de validación para el control de iluminación.

### 1.4.4. Selección del Modelo CNN para la detección de personas.

El proceso de selección realizó una revisión de la literatura más reciente para encontrar algoritmos y modelos DL de detección de objetos que contaran con la capacidad de detectar personas y de identificar el área dentro de la imagen donde se detectó a cada persona. Además, dichos modelos DL deberían ser compatibles con las plataformas de los distintos *frameworks* y dispositivos de prueba tanto a nivel de cómputo en la frontera, como cómputo en la niebla (es opcional si también era posible ejecutarlos en la nube para efectos de comparación de tiempos de inferencia).

### 1.4.5. Ejecución del Modelo CNN usando Cómputo en la Niebla y en la Frontera.

Para validar la portabilidad y comparar el rendimiento del modelo DL en distintas plataformas es indispensable implementar el sistema de iluminación en al menos un nodo basado en una arquitectura de cómputo en la niebla y un nodo de cómputo en la frontera con o sin unidades de aceleración neuronal y, posteriormente, con o sin la aplicación de al menos una técnica de adaptación, cuidando que todas las pruebas sean realizadas usando el mismo modelo DL o la variante más similar posible. La finalidad es evaluar y comparar la velocidad de ejecución de un mismo modelo DL en cada arquitectura incluyendo determinar las modalidades de ejecución, si están basadas en CPU, GPU y/o acelerador neuronal o una mezcla de varios y mostrar cuál se ajusta mejor al objetivo de la tesis y la aplicación misma (escenario de prueba). Las plataformas disponibles en el laboratorio son: Como dispositivo de cómputo en la frontera se empleó una tableta Apple iPad 6th-Gen 9.7” con procesador de cuatro núcleos A10 Fusion ARMv8 64-bit CPU @ 2.34GHz, seis núcleos externos de GPU y 2GB RAM. Para los nodos de cómputo en la niebla se cuenta con una micro-computadora Raspberry Pi 3 (RPi 3) con procesador con cuatro núcleos ARMv8 64-bit CPU @ 1.4 GHz y 1 GB RAM, y una computadora PC Intel NUC7-i5BNHXF con procesador de dos núcleos Pentium i5-7260U 64-bit CPU @ 2.2 GHz y 16 GB RAM. Como aceleradores neuronales se cuenta con dos Movidius con NPU Myriad 2 y una con Myriad X.

### 1.4.6. Adaptación del Modelo DL a Dispositivos Limitados.

Debido a las limitantes de capacidad de procesamiento y almacenamiento de los dispositivos de cómputo en la frontera y en la niebla, los modelos DL, que se caracterizan por su elevada complejidad computacional y requerimientos de memoria, pueden ejecutarse eficientemente en una sofisticada infraestructura de cómputo en la nube, pero puede resultar inviable ejecutarlos en los dispositivos tan limitados que serán utilizados para la validación. De ahí el reto del presente trabajo de encontrar alternativas para acelerar, compactar o adaptar un modelo DL usando las tecnologías o técnicas apropiadas para incidir en uno o más de los siguientes factores: reducir hasta un 50% el consumo de memoria (expresado en MB), reducir de 1,000 categorías a una sola (clase persona) en la capa de salida del modelo DL, sin disminuir significativamente la precisión del modelo DL (tolerancia de 5% mAP) y/o mejorar en un 50% el rendimiento (FPS) del modelo DL original para realizar las tareas de control de iluminación en tiempo real, misma que requiere un mínimo de 2 FPS (Nikouei et al., 2018), considerando el tiempo de ejecución del algoritmo de control y accionamiento de los dispositivos de iluminación.

## 1.5. Metodología.

Para el desarrollo del trabajo y alcanzar el objetivo, validar la hipótesis y cubrir los requerimientos planteados se propuso un conjunto de actividades organizadas en una secuencia de etapas que son descritas brevemente a continuación:

1.5.1. Investigación documental. Se realizó una revisión sistemática de literatura con el fin de encontrar trabajos relacionados, establecer una descripción general de las áreas de investigación involucradas, establecer el marco teórico del trabajo y realizar una revisión de las técnicas de adaptación pertinentes para mejorar la etapa de inferencia del modelo CNN usando dispositivos con recursos limitados de hardware.

1.5.2. Primera prueba de la hipótesis. Se realizó una primera etapa de experimentación para validar el proceso de conversión y las herramientas de desarrollo (CoreML, UIKit, Vision) para ejecutar inferencias de modelos CNN clasificadores de objetos sobre las imágenes tomadas directamente desde la cámara de un dispositivo iOS para confirmar si era viable ejecutar modelos CNN en dispositivos con recursos limitados.

1.5.3. Seleccionar modelo CNN para detectar objetos. Se investigaron modelos CNN para elegir aquel que cumpliera con los requerimientos 1.4.4, abordando las problemáticas de interoperabilidad entre plataformas, compatibilidad entre *frameworks* y entre tipos de capas del modelo CNN con respecto a cada uno de los *frameworks* a utilizar.

1.5.4. Conversiones y portabilidad del modelo CNN. Se validó la compatibilidad entre plataformas, *frameworks* y capas mediante conversiones y adaptaciones de modelos CNN, además de asegurar que las versiones de sistemas operativos, lenguajes de programación y bibliotecas sean compatibles. Se esperó al final documentar una guía con las posibles rutas de conversión para cada uno de los distintos *frameworks* utilizados.

1.5.5. Habilitar nodos de cómputo en la frontera y en la niebla. Como base de pruebas se buscó implementar un sistema de iluminación inteligente basado en distintas plataformas (iOS, Raspbian y Ubuntu) para cada uno de los dispositivos y arquitecturas de cómputo en la frontera y en la niebla evaluados.

1.5.6. Segunda validación de la hipótesis (modelo CNN original). Pruebas de rendimiento en los nodos de cada arquitectura de cómputo con y sin acelerador neuronal (NPU), con o sin escenario de prueba del sistema de iluminación. Para cada uno de los nodos sólo se evaluó la métrica de FPS, debido a que las otras dos métricas de interés, MB y mAP, no varían con respecto al dispositivo de implementación.

1.5.7. Tercera validación de la hipótesis (modelo CNN adaptado). Se implementaron y validaron las técnicas de adaptación de cuantización, transferencia de aprendizaje y poda de redes

(*pruning*). Por último, se evaluaron las métricas del modelo CNN adaptado de acuerdo a los requerimientos 1.4.6 y se compararon con el modelo CNN original.

## 1.6. Justificación.

Actualmente, las arquitecturas de IoT y DL ofrecen grandes oportunidades de innovación y desarrollo en componentes para sistemas con aplicación en Ciudades, Edificios y Casas Inteligentes (CECI). Para 2025, se estima un impacto económico anual de IoT en el rango de los $2,700 a $6,200 millones de dólares, siendo salud, industria y energía, las principales áreas de ese mercado (Mohammadi et al., 2018). Además, con la expectativa de que para 2020 se genere un volumen de datos de alrededor de 600 ZB por año (Linthicum, 2017).

Por otro lado, considerando las arquitecturas de cómputo que sostendrán este volumen de datos; en 2017, un 45% de las cargas de trabajo de tecnologías de la información fueron ejecutadas con servicios de la nube, la tendencia al finalizar 2019 es alcanzar un 60% (Buckley, 2017) y un 83% para 2020 (Columbus, 2018). A pesar de esa tendencia, llevar a cabo el cómputo en la nube no es siempre la mejor solución por varias razones. Una de ellas es debido a que los datos del usuario se envían a través de internet hacia la nube, lo cual implica una alta latencia de red y riesgos de pérdida de privacidad, aspectos clave para aplicaciones en IoT y edificios inteligentes. Por otro lado, entre más datos se envíen a través de una red inalámbrica mayor es el consumo de energía debido a sobrecarga de comunicación, además del problema de descargar los resultados del cómputo en la nube en ubicaciones con recepción de señal débil, de ahí la importancia de habilitar estas aplicaciones sin el requisito de tener que conectarse siempre a esta infraestructura en la nube (Andri et al., 2018; Maji & Mullins, 2017; Motamedi, Fong, & Ghiasi, 2016). El cómputo en la niebla y el cómputo en la frontera son dos enfoques viables para abordar estos problemas, posibilitando el análisis inmediato de los datos desde locaciones lo más cercanas posibles a los sensores, transmitiendo a la nube de forma más esporádica y breve datos pre-procesados (Andri et al., 2018). Ante la tendencia de incorporar alternativas al cómputo en la nube, es que se proyecta que para 2025, un 75% de los datos generados por las empresas sean procesados fuera de la nube, muy por encima del 10% actual (Meulen, 2018).

Pero esto también implica problemas, tal es el caso de las aplicaciones móviles donde normalmente se asume que el entrenamiento se realiza en el servidor y la inferencia se ejecuta en el dispositivo. Uno de los problemas más críticos en las aplicaciones IoT y móviles que implementan DL es que estos dispositivos tienen restricciones estrictas en términos de poder de cómputo, batería y capacidad de almacenamiento. Por lo tanto, es imprescindible adaptar modelos DL a los recursos disponibles en los dispositivos IoT (Kim et al., 2015; Yao et al., 2018). Hasta el momento no se ha logrado resolver este problema, lo cual ha impedido la utilización generalizada de modelos DL en los sistemas embebidos y dispositivos IoT con recursos limitados (Cheng et al., 2017; Maji & Mullins, 2017; Motamedi, Fong, & Ghiasi, 2016, 2017; Wu et al., 2016).

Según Maji & Mullins (2017), los sensores especializados para el reconocimiento de actividades humanas y de salud, pueden consumir entre 10-100 mW, mientras otras clases de sensores como cámaras, micrófonos y relojes inteligentes, suelen consumir alrededor de 1 W. Por otro lado, los *gateways* IoT consumen de 1 a 5 W, cuya función es resolver la heterogeneidad de protocolos entre los nodos de una red e internet, además de unir y fortalecer la gestión de estos nodos (Kang & Choo, 2018). Existen además *gateways* M2M (*machine-to-machine*) a escala industrial que pueden tener un consumo de 5 a 20 W. En contraste, ejecutar una red neuronal de mil millones de conexiones a 20 Hz, requiere alrededor de 13 W sólo para acceder a la DRAM (Han, Pool, Tran, & Dally, 2015), por ello es crucial poder simplificar o adaptar estos modelos DNN sin disminuir significativamente su rendimiento (Cheng et al., 2017; Wu et al., 2016).

Por ejemplo, el modelo ResNet-50 con 50 capas convolucionales necesita 95 MB para almacenamiento y una cantidad elevada de FLOPs para analizar cada imagen. Después de adaptar el modelo descartando los pesos redundantes (*pruning*), la red sigue funcionando como antes, pero ahorrando más del 75% de memoria para parámetros y 50% del tiempo de inferencia. Por eso, para dispositivos móviles, FPGA’s y, en general, dispositivos de recursos limitados, es importante poder compactar y acelerar los modelos DL que se usan en ellos (Cheng et al., 2017).

Desde la contribución en aplicaciones IoT hasta la necesidad de adaptar modelos DL a tales aplicaciones, en la Figura 1.2 se muestra un resumen por etapas de la problemática abordada en esta tesis.

AI se presenta como el enfoque primario, específicamente las aplicaciones móviles y IoT que se combinan con DL para soluciones como es el caso de edificios inteligentes, en esta ocasión a través del control de iluminación de un aula inteligente por medio de la detección de personas con modelos CNN.

El procesamiento se realiza en arquitecturas de cómputo que permiten mantener el control con mínima latencia como es el caso del cómputo en la frontera y el cómputo en la niebla. Para un eficiente procesamiento en estas arquitecturas de cómputo se adapta el modelo de detección de objetos a través de la aplicación de técnicas de adaptación como poda (*pruning*), transferencia de aprendizaje (*transfer learning*) y cuantización (*quantization*), con las cuales obtener un modelo de detección de personas, reduciendo las clases del detector de objetos a solo la de personas, además de acelerar la inferencia y comprimir el modelo CNN.

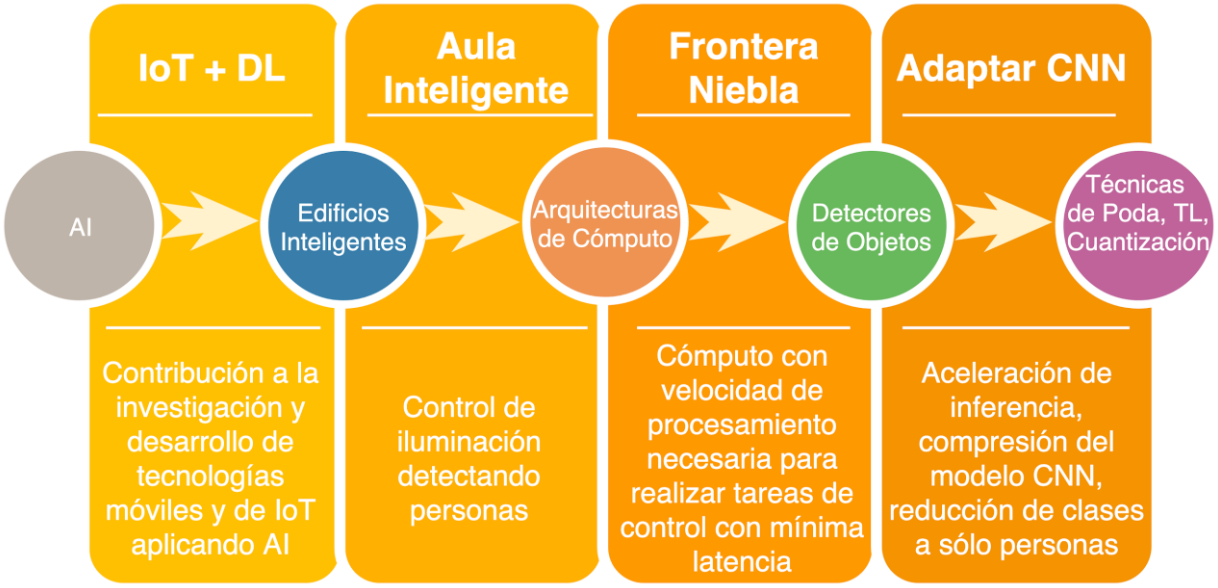


Figura 1.2. Etapas de la problemática abordada.

# II. CONTEXTO, ÁREA DE INVESTIGACIÓN Y TRABAJOS RELACIONADOS

Según el reporte "*Towards an AI strategy in Mexico*" (Martinho-Truswell et al., 2018), la automatización tiene el potencial de acrecentar la economía mundial hasta $15.7 billones de dólares para 2030. Esto es equivalente a un aumento del 14% en el PIB mundial. Además, solamente considerando tecnologías con base en aprendizaje profundo, se estima una generación anual para los próximos años de entre $3.5 y $5.8 billones de dólares (Chui et al., 2018). La distribución de estos beneficios económicos entre los países dependerá de factores como la velocidad de adopción dentro del sector privado, la definición de estrategias de automatización en las economías sectoriales, además de las políticas gubernamentales para apoyar la innovación, investigación y desarrollo tecnológico (Martinho-Truswell et al., 2018).

Lee (2018) predice que AI reemplazará entre un 40% y 50% de los empleos en EE.UU. dentro de 15 años. Mientras que, para México, según Martinho-Truswell et al. (2018), se estima que un 19% de los empleos se vean afectados por la automatización en las próximas dos décadas, de los cuales un 51% sólo verán ciertas mejoras, 33% serán transformados y 16% se verán completamente reemplazados por sistemas automatizados. La manufactura, la construcción, la minería, los servicios de alojamiento y alimentación, la industria del automóvil, la sanidad y los servicios financieros, entre otros, son particularmente susceptibles de una automatización generalizada, con la subsecuente pérdida de utilidad económica del trabajo humano en dichos sectores junto con sus posibles implicaciones sociales (Harari, 2015). Para México, será de impacto puesto que la manufactura es un sector grande de la economía, área que de acuerdo con el reporte de Martinho-Truswell et al. (2018), en 2016 representó el 20% de su PIB. Además, según este mismo reporte se estima que el 64% de las horas de trabajo en manufactura son automatizables, dada esta situación la economía de México es vulnerable a pérdidas de empleos, con crecientes tendencias de adopción de AI en la automatización a nivel global.

En el desarrollo del presente trabajo se involucraron distintas áreas de investigación tanto para el dominio de aplicación como para la incorporación de diferentes métodos, técnicas y arquitecturas, mismas que se presentan de forma breve y lo más acorde posible al problema resuelto con el propósito de establecer un antecedente conceptual en el marco teórico, además de presentar los trabajos relacionados obtenidos metodológicamente mediante una revisión sistemática de la literatura (método SLR, ANEXO 1) para con todos estos elementos establecer formalmente el contexto del presente trabajo de tesis.

## 2.1. Aprendizaje Automático, Redes Neuronales Profundas y Convolucionales.

Desde la perspectiva de las Ciencias de la Computación, AI se puede entender como una disciplina que busca aproximar sus resultados a los del razonamiento humano a través de la organización y manipulación de datos (Singh, 2014). Dentro de la disciplina de AI se encuentra el Aprendizaje Automático (*Machine Learning*, ML), área de investigación interdisciplinaria que combina ideas de varias ramas de las ciencias, tales como Estadística, Teoría de la Información, Matemáticas e incluso Biología y Psicología; convirtiéndose en la década de los 90s en un tema independiente de la Estadística (Athmaja, Hanumanthappa, & Kavitha, 2017; Burch, 2001). El Aprendizaje Automático es considerado como un paradigma cuyos algoritmos pueden aprender de la experiencia para mejorar su desempeño futuro de manera automática, sin intervención humana (Das, Behera, & Tech, 2017). Algunas aplicaciones típicas provienen de áreas tales como Bio-informática, Mercadotecnia, Visión por Computadora, Robótica y Automatización Industrial (Angra & Ahuja, 2017; Charrington, 2017).

En los inicios de ML, en una investigación enfocada a entender cómo funciona un cerebro biológico para el diseño de AI, Warren McCulloch y Walter Pitts publican en 1943 el primer concepto de una célula cerebral simplificada, llamada neurona McCullock-Pitts (MCP) (McCulloch & Pitts, 1943). Para 1957, Rosenblatt llevaría a cabo la primera implementación de este concepto, el perceptrón (Rosenblatt, 1957), siendo la base de las Redes Neuronales

Artificiales (ANN) (Jain & Mao, 1996). Años más tarde, a finales del siglo XX, debido a la mejora en las técnicas de entrenamiento como *backpropagation* y al incremento en las capacidades de cómputo con la llegada de los GPU (Foote, 2017), se populariza el concepto de algoritmos de Aprendizaje Profundo (*Deep Learning*, DL), originado a partir del estudio sobre ANN (Liu et al., 2017). Las Redes Neuronales Profundas (*Deep Neural Networks*, DNN), han proporcionado mejores soluciones que las técnicas analíticas tradicionales para un 69% de las aplicaciones de AI reportadas por el estudio de Chui et al. (2018), incluyendo áreas como Visión Artificial, Reconocimiento de Voz y Robótica. Sin embargo, si bien las DNN ofrecen una precisión mayor en esos casos, sus modelos se caracterizan por poseer una alta complejidad computacional (Sze et al., 2017).

Una arquitectura DNN diseñada específicamente para trabajar con imágenes (Altenberger & Lenz, 2018) y que tiene el mayor potencial de aplicación entre las arquitecturas DNN (Chui et al., 2018), es la de Redes Neuronales Convolucionales (*Convolutional Neural Networks*, CNN), las cuales son un tipo de red neuronal que utiliza al menos una capa de tipo convolución (Hao, Zhang, & Ma, 2016), la cual es particularmente útil para tareas de reconocimiento y clasificación de objetos, ganando poder con el increíble crecimiento de las bases de datos (*datasets*) para entrenamiento más recientes (Fan, 2015). Además, en los últimos años ha habido un creciente interés por realizar análisis más complejos de imágenes y videos para clasificar los objetos que aparecen en cada imagen, y además determinar con precisión también las ubicaciones de dichos objetos (Agarwal, Terrail, & Jurie, 2018). Esta tarea se conoce como detección de objetos, con amplia aplicación como tarea básica para asistir, por ejemplo, en la detección de rostros y de peatones en las calles (Agarwal et al., 2018; Zhao, Zheng, Xu, & Wu, 2019).

Un área de investigación reciente dentro de las arquitecturas DNN se especializa en atender el problema del alto consumo de recursos de cómputo y memoria de almacenamiento de los actuales modelos DNN, lo que los hace inadecuados para su uso en dispositivos IoT del orden de mW (Andri, Cavigelli, Rossi, & Benini, 2018), siendo un gran reto ejecutar modelos DNN en tales dispositivos. Es por ello que ha habido un creciente interés por adaptar estos

modelos DNN a dispositivos de recursos limitados (también llamados dispositivos en la frontera), logrando con ello llevar a cabo las inferencias eficientemente a través del cómputo en la niebla o en la frontera. La revisión de literatura muestra dos ramas principales que abordan este problema, el enfoque a nivel implementación y el enfoque de creación de modelos eficientes, los cuales son llamados técnicas de adaptación y explicados con detalle en el capítulo III.

En la Figura 2.1 se resume la relación que existe entre las áreas, arquitecturas y algoritmos de Inteligencia Artificial que se mencionaron anteriormente, desde el aspecto más general hasta el más específico; hasta delimitar el área de interés específica del presente trabajo de tesis.

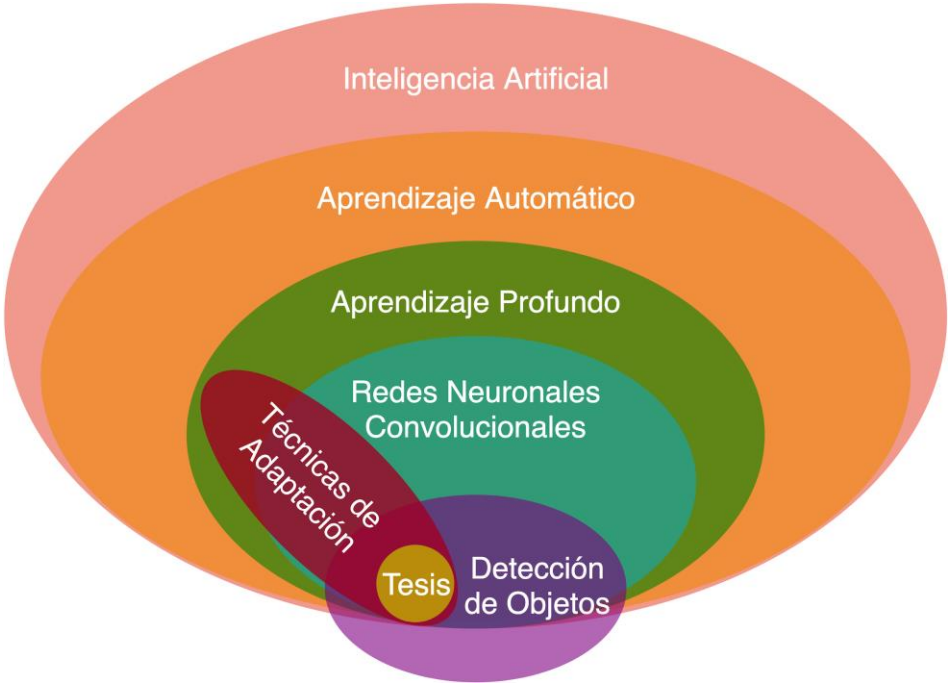


Figura 2.1. Relación entre AI, ML, DL, CNN, Detección de Objetos y Técnicas de Adaptación.

## 2.2. De Ciudades Inteligentes a Aulas Inteligentes.

Como se describió en la sección anterior, AI ha tomado fuerza en los últimos años y la tendencia favorece a un crecimiento aún mayor. Por otro lado, una de las áreas que también ha tomado fuerza y se ha impulsado por el crecimiento mismo de AI, es la de Ciudades, Edificios y Casas Inteligentes (CECI) (Pacheco et al., 2018).

### 2.2.1. Ciudades Inteligentes.

Una ciudad inteligente se define como una ciudad que monitorea e integra las condiciones de sus infraestructuras críticas, incluyendo carreteras, ferrocarriles, aeropuertos, puertos marítimos, comunicaciones, energía e incluso edificios importantes. Con sistemas avanzados de monitoreo y sensores inteligentes integrados, los datos se pueden recopilar y evaluar en tiempo real, mejorando la toma de decisiones de la administración de la ciudad (Vermesan y Friess, 2014). Este concepto prevé una mejora en la utilización de los recursos disponibles en una ciudad de manera sostenible, al tiempo que mejora la calidad de vida de su población (Cavalcante, Cacho, Lopes y Batista, 2017).

Transporte, energía, salud, seguridad, gobierno, edificios y redes de suministro de electricidad (Arasteh et al., 2016; Cavalcante et al., 2017; Hui, Sherratt, & Sánchez, 2016; Kogan & Lee, 2014) se presentan como las principales áreas en las que se están haciendo aportaciones dentro del esquema de ciudades inteligentes. En cada caso se enfrentan ciertos requerimientos esenciales, como lo son la interoperabilidad de objetos, sustentabilidad, monitoreo en tiempo real, una eficiente capacidad de almacenamiento de datos históricos, tecnologías móviles, disponibilidad, privacidad, procesamiento y sensado distribuido, capacidad de integración entre sistemas, flexibilidad, extensibilidad e incluso aspectos sociales (Da Silva et al., 2013; Song et al., 2017).

### 2.2.2. Edificios y Casas Inteligentes.

Según Manic, Amarasinghe, Rodriguez-Andina, & Rieger (2016), el concepto de edificio inteligente puede abarcar tanto edificios residenciales, comerciales como industriales. En estos casos existen componentes en común que definen un edificio inteligente, como pueden ser las técnicas de medición del consumo de energía y sensado del contexto que representan una gran oportunidad para aplicar como soporte las técnicas de AI (De Paola, Ortolani, Lo Re, Anastasi, & Das, 2014).

En el caso de los edificios residenciales, a principios del siglo XX apareció el concepto de casa inteligente (*smart home*), que alcanzó la factibilidad técnica hasta principios del siglo XXI, con la difusión de los recientes desarrollos en tecnologías de información y comunicación relacionadas con las redes de computadoras, sistemas embebidos y AI (Badica, Brezovan, & Badica, 2013), con los objetivos de mejorar el confort, ahorrar energía, tiempo y dinero, además de garantizar la seguridad y protección para los habitantes (Wilson, Hargreaves, & Hauxwell-Baldwin, 2017). Actualmente, estos beneficios se buscan a través de la implementación de dispositivos inteligentes tales como termostatos, iluminación, enchufes, interruptores o electrodomésticos que puedan ser habilitados de manera remota, con horarios programados e incluso que cuenten con algoritmos que aprendan hábitos de uso de estos dispositivos y con ello administrar mejor los recursos (Ford, Pritoni, Sanguinetti, & Karlin, 2017).

### 2.2.3. Aulas Inteligentes.

El concepto de aula inteligente tiene diversas interpretaciones, enfoques y hay varios trabajos relacionados en este sentido, tales como:

- Aulas que cuentan con pizarrones inteligentes, interactivos, con pantalla táctil y capaces de conectarse a un proyector o a una computadora (SmartBoards, 2018).

- Inclusión de software para facilitar el aprendizaje y el flujo de información (SmartTech, 2011).
- Instalación de equipos como lo son podios inteligentes con conexión a laptop y PC, cámaras de documentos, proyectores, microscopios digitales y calculadoras para graficar (Alderton, 2016; SmartWay, 2015).
- Capacidad de grabar audio y video de un pizarrón, donde el sistema se activa y configura dependiendo de la actividad que el instructor va a realizar, sin necesidad de un control manual (Winer & Cooperstock, 2002).
- Aula con cámaras y micrófonos instalados para determinar qué es lo que el presentador está tratando de hacer, y tomar acciones en consecuencia, interactuando con el sistema por medio de comandos de voz y/o gestos (Franklin, Flachsbart, & Hammond, 1999).
- Otra perspectiva más compleja, que además de considerar el equipo mencionado en los casos anteriores (así como sus capacidades de captura de audio y video, y su capacidad de adaptar automáticamente el entorno físico de acuerdo al contexto de uso), agrega mejoras tales como digitalizar las anotaciones del maestro, facilitar el intercambio de información entre estudiantes e incluso proveer mentoría a distancia, como es en el caso de Antona et al. (2011), donde el aula inteligente incorpora el uso de la computación ubicua y móvil, redes de sensores, robótica, cómputo multimedia, sistemas de agentes y AI.

En el campo de la Educación, los algoritmos de AI están comenzando a realizar aportaciones de distintas maneras, por ejemplo, ayudando a maestros a detectar áreas de oportunidad en sus enseñanzas y señalando dónde los estudiantes están teniendo problemas con la materia, o detectando si la reacción de un alumno a un concepto sigue un patrón de falta de entendimiento, dando la oportunidad al maestro de obtener una retroalimentación en tiempo real (Dickson, 2017). Otro ejemplo, es la incorporación de AI en tutores inteligentes capaces de detectar las necesidades de aprendizaje y guiar al alumno con base en eso. También se ha utilizado ML para mejorar el contenido de los cursos, para sistemas de calificación avanzados y hasta para vincular a maestros y escuelas (Ark, 2015).

En los casos mencionados, se busca aplicar tecnologías para proveer un entorno físico "extendido" más enfocado en asistir al desarrollo de una clase. Maheshwari (2016) enuncia las características que considera debe tener un aula inteligente, como un aprendizaje colaborativo entre alumnos, dispositivos de cómputo o equipo multimedia; sus objetivos, como ayudar a los estudiantes a desarrollar sus habilidades, poder instruirlos de manera remota o facilitar recursos digitales; principios, de adaptabilidad, conectividad, confort o seguridad; y componentes, tales como pizarrones inteligentes, pantallas, proyectores o computadoras; además de una serie de ventajas y desventajas, perspectiva que coincide también con un enfoque basado en mejorar la experiencia enseñanza-aprendizaje de alumnos y maestros.

En el presente trabajo de tesis, al hablar de aula inteligente se hará referencia a un enfoque que corresponde a las definiciones presentadas anteriormente de CECI, es decir, monitoreando las condiciones de su infraestructura, enfocándose en las economías generadas en un uso eficiente de los recursos de infraestructura, tales como la iluminación, temperatura, control de acceso o de otros dispositivos con los cuales alumnos y maestros interactúan dentro del aula. Todo esto con la intención de que tal entorno, a través de la aplicación de algoritmos de AI, se convierta en un ambiente más autónomo, predictivo y/o adaptativo. Con base en estos últimos elementos y en el resto de ellos presentados en esta sección, en la Tabla 2.1 se señalan las características de CECI con las cuales se formula la definición de trabajo de "aula inteligente".

Tabla 2.1. Características de la definición de trabajo de Aula Inteligente para la tesis.

| Aplicación | Áreas donde se realizan principales aportes | Componentes clave | Requerimientos | Objetivos | Prototipo |
|---|---|---|---|---|---|
| Aula Inteligente | Aplicación de modelos DL adaptados | Detección de Personas | Monitoreo en tiempo real. Nodos IoT interoperables | Ahorrar energía, y mejorar confort | Sistema de Iluminación Inteligente |

## 2.3. IoT+DL aplicado a Sistemas CECI.

La visión de IoT+DL consiste en transformar los tradicionales objetos, en objetos inteligentes, como puede ser a través del uso de modelos DL para realizar el análisis de grandes conjuntos de datos (Mohammadi et al., 2018).

En el IoT, uno de los principales problemas abiertos es el de cómo extraer de forma confiable e instantánea datos del mundo real, considerando además que los entornos reales son ruidosos y complejos, y suelen evadir las técnicas convencionales de análisis de datos. Ante esta problemática, DL ha sido considerado como uno de los enfoques más promisorios (Li, Ota, & Dong, 2018), siendo capaz de analizar grandes cantidades de datos en un tiempo razonable, reduciendo además la intervención humana en los flujos de trabajo (Dey & Mukherjee, 2018). De hecho, para Mohammadi et al. (2018), el principal elemento en la mayoría de las aplicaciones IoT puede ser el proporcionar un mecanismo de aprendizaje inteligente, capaz de hacer predicciones, minería de datos, reconocimiento de patrones o análisis de datos en general, por ello es que IoT+DL está entre las tres mejores tendencias tecnológicas estratégicas de los últimos años, de acuerdo con el estudio de dicho autor.

IoT se caracteriza entonces por usar una enorme cantidad de dispositivos o nodos finales en la frontera del internet. Sin embargo, estos nodos son dispositivos compactos que normalmente no tienen grandes capacidades de procesamiento (Zhao et al., 2019), lo que lleva al problema de dónde y cómo procesar los datos del IoT.

## 2.4. Arquitecturas de Cómputo para IoT+DL.

Para resolver el problema de dónde y cómo procesar los datos IoT y cubrir los requerimientos característicos de los sistemas CECI como la interoperabilidad, la capacidad de sensado, procesamiento y almacenamiento de datos, el monitoreo en tiempo real e incluso la privacidad, es necesario revisar arquitecturas de cómputo sobre las cuales pueden ser ejecutados

estos sistemas, considerando además, como se mencionó en la sección anterior (§2.3), los limitados recursos de procesamiento de los dispositivos IoT contra la gran capacidad de procesamiento requerida para trabajar con modelos DL.

### 2.4.1. Cómputo en la Nube.

El cómputo en la nube se introdujo por primera vez en 1996 como un paradigma de cómputo basado en redes (Zhao et al., 2019), pero, según Satyanarayanan (2017), ha sido apenas en la última década que la nube ha dominado el curso de las tecnologías de la información, esto debido a que la centralización reduce el costo marginal de la administración y las operaciones del sistema, y debido también a que las organizaciones pueden evitarse la inversión de crear un centro de datos, consumiendo recursos informáticos a través de internet, por medio de un proveedor de servicios que permite el acceso a aplicaciones, documentos y servicios desde cualquier parte del mundo a través de internet usando equipos de cómputo (Bose & Singh, 2014). Para el caso de DL, la nube ha sido una buena opción, ya que ésta provee sus recursos de cómputo bajo demanda, lo cual es importante ya que el entrenamiento de DL requiere una costosa infraestructura de hardware, misma que constantemente evoluciona y se vuelve obsoleta, siendo por tanto necesario invertir en actualizar el equipo de manera frecuente (Saiyeda & Mir, 2017). Estos elementos ayudan a sostener la expectativa que se tiene sobre el mercado global del cómputo en la nube, la cual ascenderá a 1 billón de dólares para 2024 (Zhao et al., 2019), y donde los principales participantes son servicios como Google Cloud Platform, Amazon Web Services, Microsoft Azure, Cirrascale, Nervana Cloud e IBM Rescale (Saiyeda & Mir, 2017). En contraparte, el principal factor disuasivo para proporcionar servicios de DL basados en la nube son las interrupciones en la señal de comunicaciones que esta sufre, además de una latencia relativamente alta. Los sistemas basados en la nube para el análisis de datos provenientes de sensores a menudo se ven rebasados por el requisito de baja latencia que necesitan las aplicaciones para ser operables, otro factor que puede afectar es el ancho de banda y la intermitencia en la conectividad de red cuando existe una alta movilidad en los dispositivos (Dey & Mukherjee, 2018).

### 2.4.2. Cómputo en la Niebla.

Propuesto por primera vez en 2012 (Zhao et al., 2019), el cómputo en la niebla ofrece una plataforma que proporciona servicios de cómputo, almacenamiento y redes entre dispositivos finales y servidores en la nube tradicionales. Una definición más formal lo establece como un escenario donde dispositivos heterogéneos, ubicuos y descentralizados se comunican y potencialmente cooperan entre ellos y la red para realizar tareas de almacenamiento y procesamiento sin la intervención de terceros (Yi, Li, & Li, 2015). Algunas de las principales ventajas sobre el cómputo en la nube son: un menor consumo de energía, menor latencia en la red, mayor movilidad del nodo (capacidad de mover el hardware de lugar), mejor tiempo de respuesta, menor inversión para el sistema de enfriamiento y menor espacio requerido para su implementación física. Otra ventaja de este paradigma es que permite disminuir la cantidad de datos que se envían a la nube (Prabhu, 2019). Sin embargo, las principales desventajas son una menor capacidad de cómputo y de memoria con respecto a la nube, lo que posiblemente sea una seria limitante para entrenar modelos DL, mas no sucede así, para el caso de realizar inferencias en dichos modelos (Mujthaba, Abdalla, & Manjur, 2018; Naha et al., 2018).

### 2.4.3. Cómputo en la Frontera.

Hoy en día, las tecnologías y aplicaciones emergentes para la computación móvil y el IoT están impulsando el cómputo en la frontera. De acuerdo con Satyanarayanan (2017), además del cómputo en la nube y en la niebla, el cómputo en la frontera es una importante alternativa para la inferencia de modelos DL, ya que en este paradigma se ubican recursos de cómputo y almacenamiento en la frontera de la red, muy cerca de los dispositivos móviles y/o sensores. Por ejemplo, el entrenamiento de un modelo de redes neuronales artificiales puede tomar horas o hasta días en una computadora convencional de escritorio, requiriendo costoso equipo de cómputo, o bien, llevarse a cabo en la nube. Pero la inferencia, por otro lado, puede ocurrir ya sea en la nube o en la frontera (por ejemplo, dispositivos IoT o móviles). De acuerdo con Carroll & Chandramouli (2019) alrededor de un 55% de los datos generados por IoT estarán pronto

siendo procesados en la frontera. Para el caso de las inferencias es deseable procesarlas cerca del sensor debido a cuestiones como la privacidad, falta de conectividad en la zonas de implementación o la necesidad de obtener respuestas del procesamiento con el menor tiempo de espera. Por ejemplo, en aplicaciones de visión artificial (tales como medir los tiempos de espera en tiendas o predecir patrones de tráfico), el hecho de poder extraer información del video directamente en la cámara de video (*smart cameras*) en lugar de enviar el video a la nube ayuda a reducir el tiempo y costo de la comunicación. Para aplicaciones como vehículos autónomos, navegación con drones y robótica, es una ventaja poder realizar un procesamiento local ya que la latencia y los riesgos de seguridad son más manejables (Sze, Chen, Yang, & Emer, 2017).

En la Tabla 2.2 se muestra una comparativa entre las arquitecturas de cómputo en la nube, la niebla y la frontera con base en los principales requerimientos a tener en cuenta en aplicaciones para CECI que tienen el potencial de incorporar modelos DL. Como se puede observar, una de las principales desventajas y retos en las áreas del cómputo de la frontera y la niebla consisten en adecuar los requerimientos de procesamiento propios de los modelos DL a las limitadas capacidades de cómputo y de memoria que poseen los nodos y dispositivos IoT que operan bajo estas arquitecturas.

Tabla 2.2. Comparación de requerimientos entre el Cómputo en la Nube, Niebla y Frontera.

| Requerimiento | Nube | Niebla | Frontera |
|---|---|---|---|
| Consumo de energía | Alto | Bajo | Muy bajo |
| Latencia de red | Alta | Baja | Nula |
| Movilidad del equipo | Baja | Alta | Muy alta |
| Tiempo de respuesta | Alto | Bajo | Bajo |
| Costo de sistema de enfriamiento | Alta | Baja | Baja |
| Espacio físico | Alto | Bajo | Muy bajo |
| Seguridad | Baja | Alta | Muy alta |
| Capacidad de procesamiento | Alta | Baja | Muy baja |
| Capacidad de almacenamiento | Alta | Baja | Muy baja |

## 2.5. Trabajos Relacionados.

Una parte importante de cada trabajo de investigación es la revisión del estado del arte, con el objetivo de encontrar y conocer trabajos similares al propio. Para lograr este objetivo se aplicó la metodología SLR para revisión sistemática de literatura, descrita en el Anexo 1. A continuación, se presentan los principales trabajos relacionados y se explica en qué son similares, así como otras perspectivas y posibilidades sobre cómo abordar el problema e hipótesis planteada en el presente trabajo.

Nikouei et al. (2018) presentan una aplicación CECI de vigilancia, destacando su observación sobre el problema de trabajar con video, debido a que representa una carga intensiva para las comunicaciones vía red, por lo que resulta importante y recomendable realizar al menos parte del procesamiento del video antes de transferirlo a la nube. Utilizan un modelo DL para detección de objetos llamado *Single Shot Multi-Box Detector* (SSD), con la arquitectura MobileNet como clasificador base. Para reducir el consumo de recursos de este sistema se apoyan en la técnica de *depthwise separable convolution* y aunque no mencionan el método que siguieron, también reducen el espacio de búsqueda del clasificador a sólo una clase, la de personas. Los autores mencionan que los sistemas de vigilancia inteligentes se adaptan de manera efectiva con una arquitectura jerárquica de frontera-niebla-nube, de modo que su prototipo realiza el cómputo en la frontera usando un nodo implementado en una RPi 3, con la cual alcanzan un promedio de 1.79 FPS y un máximo de 2.06 FPS. Además, afirman que para esta aplicación de vigilancia, 2 FPS resultan ser suficientes para un desarrollar un sistema funcional.

De manera similar, Wei Yang & Yen Su (2018) presentan un sistema de vigilancia inteligente, bajo el argumento de que buscan no sólo capturar y almacenar video, sino analizarlo también para poder prevenir, por ejemplo, crímenes, terrorismo o accidentes. También utilizan SSD como sistema base para detectar personas y realizan pruebas con TinyYOLO (no especifican exactamente cuál versión), pero los implementan sin modificación alguna, es decir, los usan con todas las clases que ya vienen definidas y tampoco se implementa algún tipo de

técnica de adaptación para reducir el consumo de recursos del modelo CNN. Para realizar el cómputo de la inferencia ellos usan un acelerador Intel Movidius NCS (*Neural Compute Stick*), las microcomputadoras RPi 3 y Rock64: Los autores no colocan su trabajo dentro del contexto de los conceptos de cómputo en la frontera o en la niebla. Los resultados que presentan son los tiempos de ejecución con cada dispositivo con y sin la ayuda del acelerador Intel NCS.

Jensen, Gade, & Moeslund (2018) aplican la detección de personas para realizar un análisis espacio-temporal de la ocupación de albercas y con ello facilitar la administración de los costosos recursos que consumen estas instalaciones y facilitar que los clientes puedan agendar sus actividades en horas que no sean pico. En este caso tampoco implementan técnicas de adaptación de modelos DL, ni manejan arquitecturas de cómputo de frontera/niebla, aunque sí hacen énfasis en las diferencias entre usar GPU's de alto rendimiento y usar una RPi 3 con un acelerador NCS. Lo que hacen en ese trabajo es entrenar dos redes CNN, una con base en los pesos de YOLOv2 y otra con los pesos de TinyYOLO, con imágenes, etiquetas y cuadros delimitadores generados por ellos mismos para darles la información si hay personas en la alberca o no, cuántas personas hay y en qué carril están nadando.

O'Keeffe & Villing (2018) hacen énfasis en que la detección de objetos es un factor clave para desarrollos como los vehículos autónomos, detección de peatones o robots domésticos. En este trabajo evalúan la eficacia de realizar una poda de un modelo TinyYOLO, además de luego aplicar *fine-tuning* para mejorar los resultados, alcanzando una reducción de FLOPS de 4.5x y una reducción de parámetros de 7x sin pérdidas en la precisión. Realizan reducción de clases para solamente tomar en cuenta cuatro escenarios aplicando la técnica de transferencia de aprendizaje y posteriormente la técnica de poda. No consideran tampoco las arquitecturas de cómputo en la nube y la frontera.

Chen, Ruan, & Liao (2018), consideran importante conocer el grado de ocupación de un aula para poder administrar de manera eficiente sus recursos de energía. En lugar de iluminación trabajan con el control del HVAC, y detectan a través de una cámara la cantidad de personas aplicando un modelo YOLOv3, sin modificar las clases y sin adaptar de alguna manera el

modelo y tampoco considerando las arquitecturas de cómputo de frontera/niebla. En su aplicación logran una precisión del 60% con una sóla cámara, concluyendo que posiblemente con dos cámaras sea más que suficiente para lograr un sistema funcional.

Los trabajos de Jensen, Gade, & Moeslund (2018) y Ni, Chen, Sang, Gao, & Liu (2018) entrenan una red CNN YOLOv2, pero en el caso de este último, es para detectar 10 tipos de gestos con las manos. A pesar de que no toman en cuenta las arquitecturas de cómputo en la frontera o en la niebla y que se limitan a realizar sus pruebas en una GeForce GTX TITAN X, si contemplan la adaptación del modelo CNN para mejorar la velocidad de ejecución y el tamaño de almacenamiento requerido, esto lo hacen implementando también una poda de la red CNN, mejorando una precisión de 96.8% hasta 98.06%, además pasando de una velocidad de 40 FPS a 125 FPS y de un tamaño de 250MB a 4MB.

En el trabajo de Zhao, Barijough, & Gerstlauer (2018), implementan un modelo DNN usando cómputo en la frontera basada en una RPi 3 y destacan que es una tendencia que está mejorando temas como la escalabilidad y la privacidad mediante el procesamiento local de las grandes cantidades de datos que utilizan los modelos DNN. Conscientes también de que el cómputo en la frontera tiene el problema de que sus dispositivos son de recursos limitados, consideran las técnicas de adaptación como una opción para facilitar las aplicaciones en la frontera. Tomando como base YOLOv2, aplican una serie de técnicas como el particionamiento de capas convolucionales, distribución dinámica de carga de trabajo y un proceso de programación de tareas, logrando con ello mejoras en el tiempo de inferencia que van de 1.7 a 2.2 veces más veloces que el modelo CNN original.

Szemenyei & Estivill-Castro (2019) desarrollan un sistema de detección de objetos para el robot Nao v6 llamado ROBO. Este detector de objetos está habilitado para la clasificación de cuatro clases relevantes para que el robot se desenvuelva en un ambiente de fútbol para robots (balón, líneas delimitadoras del campo, postes de portería y robots). Esto lo llevan a cabo con transferencia de aprendizaje con TinyYOLOv3 como modelo base. Durante la etapa de *fine-tuning* de la transferencia de aprendizaje, aplican una regularización por norma L1 para reducir

las matrices de pesos de la red y después podar tales pesos. Logran podar aproximadamente el 90% de los parámetros del modelo ROBO, prácticamente sin pérdidas de precisión y reduciendo el tiempo de ejecución en un 70%. Proponen como trabajo a futuro podar filtros completos en lugar de sólo pesos individuales, con lo cual lograr mejores resultados.

Singh, Verma, Rai, & Namboodiri (2019) presentan una biblioteca para poda de filtros de CNN's llamada *Play and Prune* (PP). Esta técnica lleva a cabo la poda en conjunto con un proceso de *fine-tuning* para reducir y ajustar los pesos de la red a través de una tasa de poda adaptativa en cada iteración, manteniendo el desempeño de la red. Esto lo llevan a cabo con dos módulos, uno para la poda de filtros y otro para controlar la tasa de poda para maximizar la precisión durante el proceso. A diferencia de la mayoría de los métodos previos, este permite especificar y podar con base en una tolerancia de error deseada, en lugar de hacerlo con base en el nivel de poda. Con este enfoque logran reducir los parámetros de la red VGG en un factor de 17.5, reducir el número de FLOPs en un factor de 6.43 y sin pérdidas de precisión.

Según Lee, Ajanthan, & Torr, 2019, actualmente la mayoría de los métodos de poda llevan a cabo un procedimiento de optimización iterativo, ya sea con esquemas de poda diseñados heurísticamente o con hiper-parámetros adicionales, mientras que ellos presentan un enfoque que poda la red en una sola iteración antes del entrenamiento. Para ello aplican un criterio de prominencia (*saliency criterion*) basado en la sensitividad de las conexiones, identificando las conexiones más importantes en la red para cierta tarea. Después de hacer la poda realizan el entrenamiento de manera normal. Esta técnica logra una compresión del modelo VGG del 95%, con pérdidas de precisión menores al 0.5%.

La selección reportada de trabajos relacionados, mismos que se enlistan y caracterizan en la Tabla 2.3, si bien abordan temas de investigación similares y recientes, adolecen sin embargo, como ha sido evidenciado, de que ninguno de ellos aborda de manera integral todos los temas centrales del presente trabajo, es decir, que refieran una aplicación de un sistema CECI aplicando arquitecturas de cómputo de la frontera/niebla para IoT ejecutando la etapa de inferencia de modelos CNN para detección de personas en dispositivos con recursos de

procesamiento limitado y aplicando técnicas de adaptación en modelos CNN originales para acelerar y/o aligerar dichos modelos.

Tabla 2.3. Resumen y calificación de trabajos relacionados.

| Título | Autores | Año | N | S | C | P | Y | A | Calif. |
|---|---|---|---|---|---|---|---|---|---|
| Evaluating pruned object detection networks for real-time robot vision | O'Keeffe & Villing | 2018 | 4 | 1 | 0 | 4 | 4 | 4 | 17 |
| ROBO: Robust, Fully Neural Object Detection for Robot Soccer | Szemenyei & Estivill-Castro | 2019 | 4 | 1 | 0 | 4 | 3 | 4 | 16 |
| Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN | Nikouei et al. | 2018 | 4 | 3 | 3 | 4 | 0 | 0 | 14 |
| Low-cost CNN Design for Intelligent Surveillance System | Wei Yang & Yen Su | 2018 | 4 | 3 | 0 | 4 | 3 | 0 | 14 |
| Swimming Pool Occupancy Analysis Using Deep Learning on Low Quality Video | Jensen, Gade, & Moeslund | 2018 | 4 | 3 | 0 | 4 | 3 | 0 | 14 |
| Light YOLO for High-Speed Gesture Recognition | Ni, Chen, Sang, Gao, & Liu | 2018 | 4 | 0 | 0 | 4 | 2 | 4 | 14 |
| DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters | Zhao, Barijough, & Gerstlauer | 2018 | 4 | 1 | 3 | 4 | 2 | 0 | 14 |
| iOccupancy: An Investigation of Online Occupancy-driven HVAC Control in Campus Classrooms | Chen, Ruan, & Liao | 2018 | 4 | 3 | 0 | 4 | 2 | 0 | 13 |
| Play and Prune: Adaptive Filter Pruning for Deep Model Compression | Singh, Verma, Rai, & Namboodiri | 2019 | 4 | 0 | 0 | 4 | 0 | 4 | 12 |
| SNIP: Single-Shot Network Pruning based on Connection Sensitivity | Lee, Ajanthan, & Torr | 2019 | 4 | 0 | 0 | 3 | 0 | 4 | 11 |

Criterio de calificación (ver detalles en Figura 2.2):

N: Tipo de Modelo de AI

S: Área de implementación

C: Arquitectura de cómputo

P: Tipo de algoritmo de detección de personas

Y: Versión de TinyYOLO

A: Técnicas de adaptación

Para esta revisión de trabajos relacionados se implementó la metodología SLR, la cual es explicada con más detalle en el Anexo I. La búsqueda de artículos se realizó en las bases de datos de IEEE, ACM y ResearchGate. Se obtuvieron 160 trabajos, con combinaciones de palabras clave como “cnn”, “pruning”, “edge”, “fog” o “yolo”; de los cuales se presentaron los diez trabajos más relacionados a la presente tesis, esa relación se determinó a través de un proceso de calificación basado en el criterio descrito en la Figura 2.2.

Los números dentro del hexágono representan la secuencia en que dicha área fue considerada para la formulación de esta tesis, las letras se utilizan para codificar los trabajos relacionados y los puntajes junto a cada subárea se usaron para calificar. La primera consideración fue obtener trabajos que emplean la misma subárea de AI (ML), además de usar la misma técnica (DL) y la misma arquitectura de red neuronal (CNN). El segundo criterio fue buscar trabajos con aplicación en aulas o edificios inteligentes, considerando también trabajos que pudieran tener un enfoque más amplio como el de ciudades inteligentes o al menos aplicaciones IoT en general. En arquitecturas de cómputo la calificación más alta se asigna a trabajos que consideran el cómputo en la frontera y en la niebla, de lo contrario puede ser sólo uno de los dos, con prioridad en la frontera debido a que se busca la mayor proximidad con el sensor; o bien, si el trabajo implementa cómputo en la nube también se considera por motivos de comparación. Con el criterio de detección de personas se buscan trabajos que realicen esa tarea aplicando detectores de objetos, considerando que existen otras soluciones que sólo apliquen CNN, que resuelvan el problema sin DL o incluso que no sean de visión. La quinta consideración es la arquitectura de YOLO, donde se buscan trabajos que usen TinyYOLOv2, pero tomando en cuenta también las otras versiones que son muy similares. El último criterio son las técnicas de adaptación, donde los trabajos más relacionados implican el uso de poda de redes y transferencia de aprendizaje, considerándose también si sólo usan una de ellas o si implementan cuantización, la cual también se tomó en cuenta en este trabajo, pero con menor ponderación debido a su menor relación con la solución del problema con respecto a las otras dos.

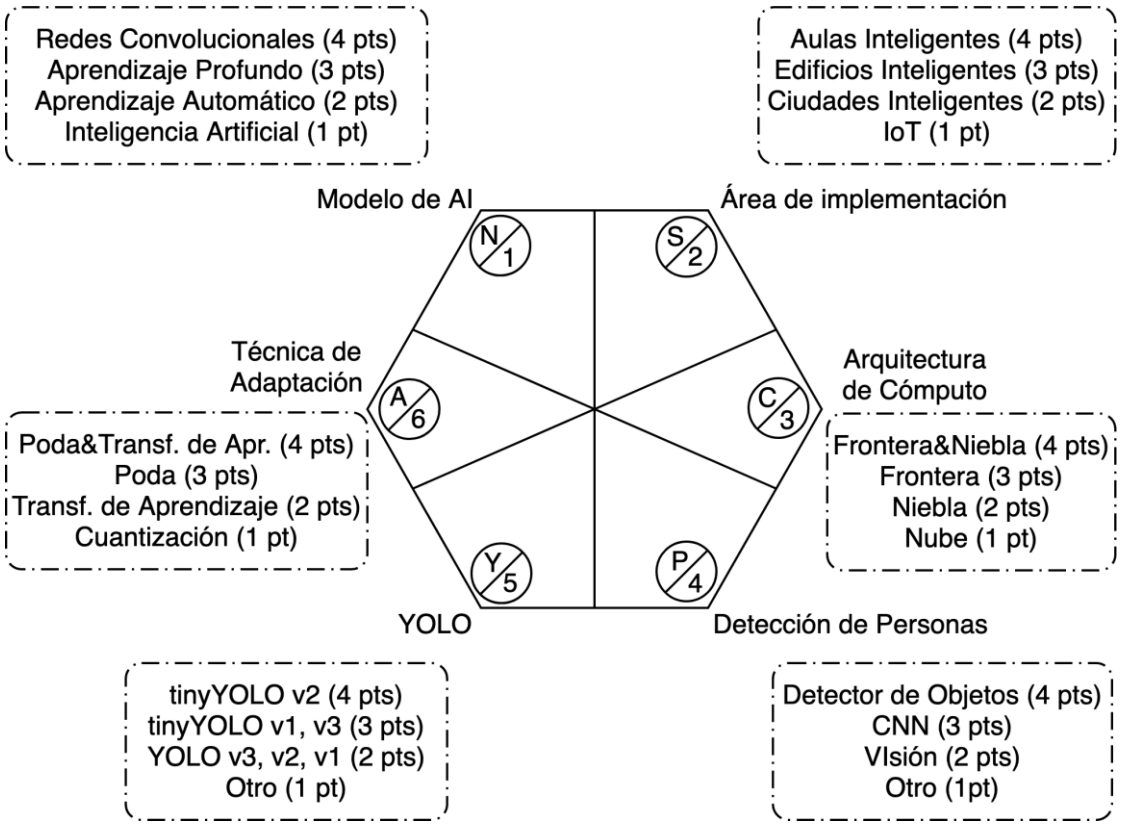


Figura 2.2. Criterio de selección y calificación de Trabajos Relacionados.

# III. MARCO TEÓRICO

El presente trabajo de tesis involucra diversas áreas de conocimiento, motivo por el cual tanto la revisión de la literatura como la selección de los temas y conceptos resulta compleja, por lo que se propone una clasificación de los mismos organizada en tres niveles, partiendo en la periferia de los temas más generales hasta llegar a la parte central donde se localizan los conceptos, métodos y técnicas más específicas que fueron aplicadas en el presente trabajo. En la Figura 3.1 se representan dichas áreas y niveles de conocimiento. En el capítulo anterior se describieron las tendencias actuales de investigación y desarrollo en las áreas generales y algunos temas de la capa intermedia. A continuación, se describen de manera formal los conceptos centrales para establecer el marco teórico del trabajo.

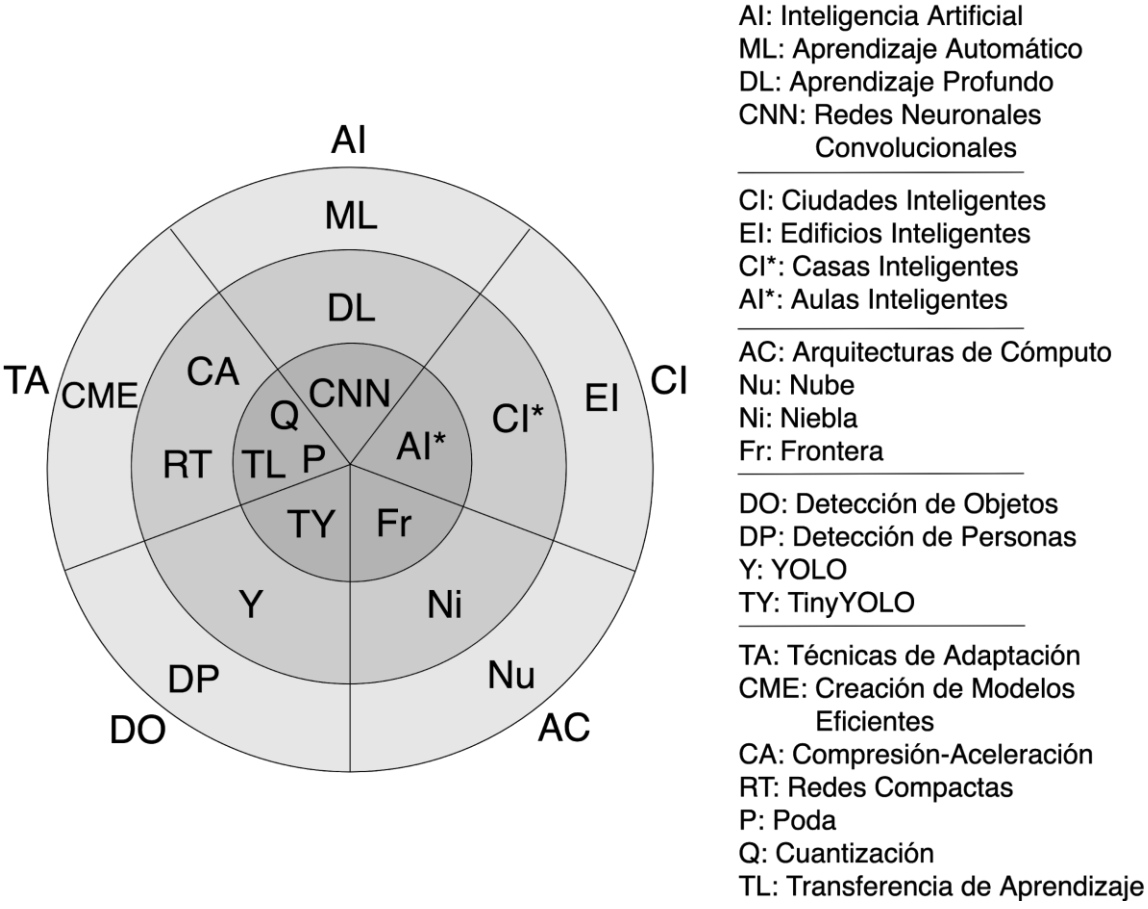


Figura 3.1. Organización de las áreas teóricas involucradas en el presente trabajo de tesis.

## 3.1. Modelos de Aprendizaje Profundo para la Detección de Objetos.

El sistema de detección de objetos se limita a la aplicación específica de modelos CNN que pertenecen a DL, que a su vez son parte de ML, una de las principales ramas de AI. ML puede verse como una herramienta usada para procesar datos complejos a gran escala, desempeñándose mejor cuando puede usar grandes cantidades de datos, mejorando así su poder predictivo y analítico debido a su capacidad de aprender a descubrir patrones e irregularidades escondidas en dichos datos (Zocca, Spacagna, Slater, & Roelants, 2017). Por otro lado, DL es comúnmente definido como una clase de algoritmos de ML que emplean redes neuronales con más de una capa de procesamiento de información no lineal para la extracción y transformación de características en modelos supervisados o no supervisados, para el análisis y la clasificación de patrones (Deng & Yu, 2014). En la literatura también es común encontrar como equivalente el concepto de red neuronal profunda (*Deep Neural Network*, DNN), aunque existe diferencia entre ellos. Una DNN se define como una red neuronal con más de una capa oculta, mientras que a los algoritmos desarrollados especialmente para entrenar de manera más eficiente tales redes, se les conoce precisamente como DL (Raschka & Mirjalili, 2017). El sistema por el cual estas redes aprenden es llamado entrenamiento (*training*), en el cual la red extrae características de los datos que se le presentan para aprender, y con ello posteriormente poder realizar lo que llaman inferencia (*inference*), que básicamente consiste en inferir una respuesta ante información que no conocía previamente con base en lo aprendido (Copeland, 2016).

Existe una extensa variedad de arquitecturas de redes neuronales profundas. Por ejemplo, las redes CNN poseen una de las arquitecturas más utilizadas para el reconocimiento de imágenes, y las redes RNN (*Recurrent Neural Networks*) son en cambio más aplicables a tareas secuenciales, como la escritura a mano o el reconocimiento de voz (Hao et al., 2016). En general, los modelos DL tienen un alto rendimiento, dando en un 69% de los casos (Chui et al., 2018) soluciones más eficientes en áreas como el habla, el lenguaje o la visión. Además, en DL no hay necesidad de aplicar técnicas manuales de ingeniería de características (*feature engineering*), que es una de las partes que más tiempo consume en la práctica en otros algoritmos de ML, ya que en este caso la extracción de características la hace la misma red DL. En contraparte, a

diferencia de otros clasificadores como árboles de decisión o regresión logística, lo que aprende un modelo DL no resulta nada fácil de interpretar, comprender o explicar (Zeiler & Fergus, 2014). Además, la determinación de la estructura de la red DL, los tipos de capas, el método de entrenamiento y el ajuste de hiper-parámetros son tareas complicadas y de tipo heurístico ya que no cuentan con una guía, método determinista o teoría de soporte para dicho proceso, siendo todo un arte llevar a cabo esas configuraciones al momento de entrenar un modelo DL (Bengio, Courville, & Vincent, 2012).

A continuación, se presentan los elementos básicos de una red neuronal artificial, una red neuronal convolucional y una breve introducción al proceso de entrenamiento y los métodos de regularización, *dropout* y aumentación de datos (*data augmentation*), tomando como referencia los trabajos de Altenberger & Lenz (2018), Fan (2015) y Sze et al. (2017).

Desde un punto de vista general, las ANN pueden estratificarse en capas, distinguiéndose tres principales tipos: la capa de entrada, la capa de salida y opcionalmente capas ocultas, que son simplemente todas las demás capas que estén en medio, las cuales desempeñan cómputos complementarios, siendo representaciones intermedias de la entrada (Ver Figura 3.2). En las primeras capas ocultas (las más cercanas a la capa de entrada) es donde se reconocen características de bajo nivel como líneas o bordes. Mientras que en las capas más profundas se pueden encontrar patrones de alto nivel como ojos, nariz o boca.

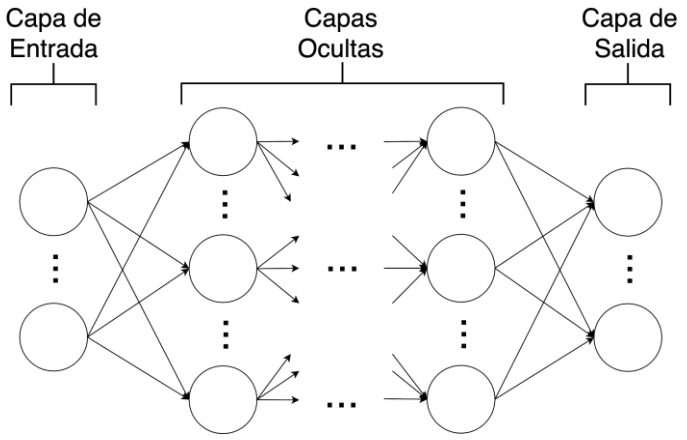


Figura 3.2. Estructura de ANN: Capa de entrada, capas ocultas y capa de salida.

Dentro de una misma capa pueden existir una o más neuronas, las cuales se conectan con las neuronas de las capas adyacentes. Las conexiones entre una neurona y otra son llamadas sinapsis, cada una contiene un peso, el cual determina cuán importante es el resultado de la neurona de bajo nivel para el resultado de la neurona de alto nivel. Además, cada neurona contiene un *bias* que representa que tan probable es, en general, que un patrón correspondiente se encuentre presente. Estos dos valores, pesos y *bias*, son los parámetros de la red, los cuales serán aprendidos durante el proceso de entrenamiento. Existen otros atributos, que son los del procedimiento de entrenamiento, mismo que son seleccionados de manera manual y son llamados hiper-parámetros.

Una de las partes más importantes en el diseño de una red neuronal es poder evaluar la precisión de la red, es decir, conocer cuál es el error del sistema. Para una red neuronal de clasificación o de regresión es posible definir una función de pérdida $l$, dada la salida de la red neuronal y el *ground truth* de los datos de entrada $x$. Con $l$como función de pérdida de un solo dato de entrenamiento, se puede definir $L$ como función de pérdida sobre el conjunto de datos $x$ como:

$$\boldsymbol{L(x, W)} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{l(x_i, W)} \tag{3.1}$$

La suma de los cuadrados y *softmax* son funciones de pérdida ($l$) empleadas de manera frecuente.

La meta en estos sistemas es lograr minimizar la función de pérdida $L$ con respecto a los parámetros de cada capa. Probablemente, la forma más conocida de lograr ésto es con el método de aprendizaje por gradiente, basado en que la ecuación (3.1) puede minimizarse estimando los efectos de pequeñas variaciones en los valores de los parámetros. El descenso de gradiente (*gradient descent*) y el descenso de gradiente estocástico (*stochastic gradient descent*) son ejemplos comunes de este método. La actualización de pesos basada en descenso de gradiente consiste en actualizar los pesos con base en el gradiente negativo de la función de pérdida, en cambio con descenso de gradiente estocástico, la actualización hace con base en una combinación lineal del gradiente negativo de la función de pérdida y el valor de la actualización

de pesos anterior, siendo con esto más robusto al ruido en los datos de entrenamiento. La parte clave para poder actualizar los parámetros de una red usando estos métodos es obtener el gradiente de la función de pérdida con respecto a los parámetros, y una manera eficiente de llevar a cabo eso es a través del algoritmo de retro-propagación (*back-propagation*). Básicamente, consiste en propagar el cómputo de los gradientes desde la capa de salida hacia la capa de entrada, es decir, realizar el cómputo de las derivadas de la función de pérdida $L$ con respecto a los parámetros de la red desde las últimas capas hasta las primeras mediante la regla de la cadena.

Las capas son los bloques fundamentales de las redes neuronales, para las CNN existen algunas capas representativas. Una de las más sencillas y frecuentemente presente es la capa de activación, la cual puede estar presente como una capa o embebida en otra en forma de función de activación a la salida. El propósito principal de la función de activación es introducir una no-linealidad a la red neuronal para derivar o "aprender" transformaciones más complejas entre las entradas y las salidas, debe ser una función diferenciable para los cálculos del descenso de gradiente y puede además modular cuándo se dispara una neurona artificial (*activation*) al momento de superar cierto umbral (*threshold*), e.g. función ReLU. Sin estas funciones de activación la red sería una simple transformación lineal (un modelo de regresión lineal) desde la entrada a la salida. En la Figura 3.3 se muestran algunas de las funciones de activación más usadas, que son la ReLU, Leaky ReLU, Sigmoid, y TanH. La función ReLU y Leaky ReLU son ampliamente usadas en modelos CNN, demostrando empíricamente mejores resultados que otras funciones (Xu, Wang, Chen, & Li, 2015). ReLU es computacionalmente menos costosa que Leaky ReLU, pero tiene el riesgo de eliminar una gran cantidad de neuronas en la retro-propagación si en la inicialización prevalecen ceros en las salidas. Una solución a ese problema es la LeakyReLU, ya que no convierte en cero los valores de salida negativos, sólo los disminuye en gran proporción y con eso se evita la eliminación de neuronas (Böhm, 2018; Chevalier, 2019). Otro tipo de capa son las *fully connected*, las cuales reciben entradas de todas las neuronas de la capa anterior. En una CNN típica, los mapas de características (*feature maps*) (cada capa genera sucesivamente una abstracción de nivel superior de los datos de entrada, denominada mapa de características, esta salida conserva información esencial y única de cada entrada) de la última

capa convolucional se vectorizan y se conectan completamente (*fully connected)* con las unidades de salida, que normalmente se siguen por una capa de función de pérdida del tipo *softmax*.

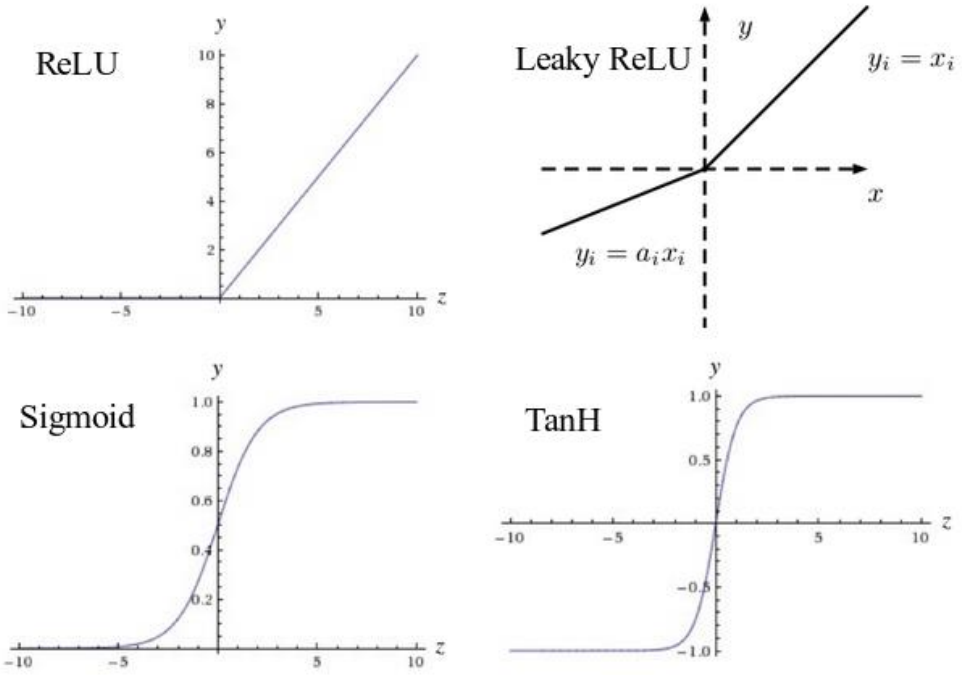


Figura 3.3. Funciones de activación. ReLU, Leaky ReLU, Sigmoid y TanH. Adaptado de Buduma & Locascio (2017) y Xu et al. (2015).

## 3.2. Redes Neuronales Convolucionales (CNN).

Las redes CNN, son modelos DNN de tipo *feed-forward*, esto se refiere al hecho de que cada capa sirve como entrada a la siguiente capa, sin ciclos, a diferencia de las RNN (Raschka & Mirjalili, 2017). Las CNN se distinguen por utilizar la capa tipo convolución en al menos una de sus capas, es decir, combinan ANN y convolución discreta para el procesamiento de imágenes que se puede usar para extraer características automáticamente. Por lo tanto, están especialmente diseñadas para reconocer datos bidimensionales, como imágenes y videos. Las imágenes se pueden usar directamente como entrada de la red, lo que evita la extracción de características complejas y el proceso de reconstrucción de datos en los algoritmos tradicionales

de reconocimiento de imágenes (Hao et al., 2016). Cabe señalar que las CNN son la primera arquitectura de DL en alcanzar resultados destacables en términos de precisión, esto debido al entrenamiento exitoso de sus capas, además de que la topología CNN aprovecha las relaciones espaciales para reducir el número de parámetros en la red (Liu et al., 2017).

La convolución (*convolution*) es una operación entre dos funciones, en el caso de redes neuronales suelen ser una entrada (por ejemplo, de dos dimensiones para el caso de imágenes y vídeo) y una función que representa las ponderaciones o pesos, llamada *kernel*. Con el resultado de esta operación se logra resaltar e identificar ciertas características de la entrada y enviar esa información a la siguiente capa. Esa salida es el llamado mapa de características (*feature map*). A diferencia de las capas *fully connected* que son las que tienen mayor impacto en la cantidad de parámetros de una red, las capas convolucionales tienen mayor impacto en la complejidad computacional del modelo (Ge, 2018).

Para una capa convolucional $l$, se lleva a cabo una convolución entre los mapas de características de las capas previas y los *kernels* para formar el mapa de características de esta capa. Sea $i$ un mapa de características de la capa $l$, se denotará como $x_i^l$, y sea $j$ uno de los mapas de características de la capa $l-1$, denotado como $x_j^{l-1}$. $x_i^l$ se obtiene mediante:

$$\boldsymbol{x_i^l = \sum_{j \in M_i} k_{ij}^l * x_j^{l-1} + b_i^l} \tag{3.2}$$

siendo $i = 1,2,\ldots,|M|$. $b_i^l$ es el *bias*, compartido por todas las conexiones del mapa de características $i$. $|M|$ es la cantidad de mapas de características en la capa $l$. $M_i$ es el subconjunto (o conjunto completo) de mapas de características de la capa $l-1$ que se conectan a las unidades $i$ de la capa $l$. En la Figura 3.4 se muestra esta operación, donde la cuadrícula azul es el mapa de características de la capa $l-1$, la cuadrícula verde indica el mapa de características de la capa $l$ y la zona en azul oscuro el *kernel* de 3x3.

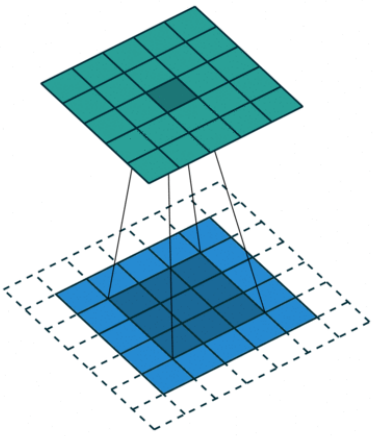

Figura 3.4. Representación gráfica de la operación de Convolución.

Además del tamaño del *kernel*, otro de los hiper-parámetros importantes para la convolución es llamado *stride*, con el cual se establece que tan grande es el intervalo entre el escaneo de dos regiones, en la Figura 3.5 se muestra un ejemplo donde un *kernel* de 1x1 representado por los cuadros en azul oscuro recorre un mapa de características con un *stride* de 2. Otro hiper-parámetro es el *padding*, con el cual es posible agregar a la imagen de entrada una columna y fila de cada lado con ceros, con ello, por ejemplo, es posible conservar el tamaño de la entrada. En la Figura 3.4 también puede visualizar este hiper-parámetro, si se utiliza el *padding*, el *kernel* estaría haciendo el recorrido sobre los cuadros en blanco también, de lo contrario sólo sobre la zona azul.

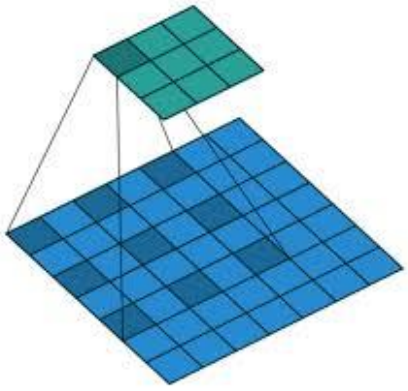

Figura 3.5. Representación gráfica del hiper-parámetro *stride*.

Casi en cualquier arquitectura de una CNN, después de una capa convolucional (o de un bloque de ellas) se coloca una capa de *pooling*, también llamada capa de submuestreo. Es usada principalmente para reducir las dimensiones del mapa de características, y de esa manera reducir el costo computacional al descartar valores locales que no son máximos. También provee resistencia ante ruido y distorsiones en los mapas de características. Los tipos más comunes de *pooling* son dos, uno donde se conserva el valor máximo (*max pooling*) y otro donde se conserva el promedio (*average pooling*). A diferencia de las capas convolucionales, las de *pooling* no tienen parámetros, es decir, no tienen pesos o *bias* para ser entrenados.

Durante el entrenamiento de las capas que si tienen parámetros entrenables, uno de los efectos que se busca evitar a toda costa es el del sobre-entrenamiento (*overfitting*). Este detecta cuando el error de entrenamiento decrece de manera consistente durante el entrenamiento, mientras que el error en la validación comienza a aumentar en cierto punto. Para explicar este efecto suele decirse que es como si la red hubiera aprendido de memoria que es cada elemento que se usó para el entrenamiento sin realmente haber aprendido a detectar las características que lo definen, por ello al observar datos distintos (los de validación) el error es mayor al de los datos de entrenamiento. Entre menos datos de entrenamiento se usen, mayor es el riesgo de sobreentrenar la red, riesgo que tambíen se eleva entre más parámetros tenga un modelo. Afortunadamente, existen diversas técnicas que sirven para ayudar a disminuir la aparición de este efecto indeseado durante el entrenamiento de una red neuronal. La técnica de regularización (*regularization*), agrega un término de regularización durante la medición del error con el fin de controlar la complejidad del modelo CNN. La técnica *dropout* también es útil para reducir la posibilidad de llegar al sobre-entrenamiento, esta técnica consiste en omitir de manera aleatoria un porcentaje de neuronas tanto para la propagación hacia adelante de la entrada como la propagación hacia atrás del error. Esta omisión sólo se hace para el entrenamiento, para la evaluación si se usan todas las neuronas. Esta técnica obliga a romper las posibles coincidencias de ciertas características y hace que la red aprenda características más distintivas y sólidas. La última técnica para evitar el sobre-entrenamiento de una red neuronal es la aumentación de datos (*data augmentation*), con ella es posible aumentar la cantidad de datos de entrenamiento

alterando el conjunto de dato para, en el caso de redes CNN, generar nuevas imágenes a partir de las imágenes de entrenamiento disponibles, usando diversos métodos como voltear la imagen, cortarla, modificar su contraste, color e incluso agregando ruido.

A través de la combinación de estas distintas capas y funciones especiales que se han mencionado, es como se logra armar una red CNN para clasificar o detectar características en imágenes, aunque normalmente, para obtener modelos DL que logren resolver un problema de manera eficiente, es necesario agregar más de una capa, lo que conlleva un aumento en la cantidad de parámetros y, por lo tanto, una alta complejidad computacional.

## 3.3. Detección de Personas con CNN's.

De acuerdo con la evidencia encontrada durante la revisión del estado del arte y ciertos trabajos relacionados analizados, se encontró que para abordar el requerimiento de detección de personas (§1.4.3) es posible aplicar modelos CNN para detección de objetos. Según Agarwal et al., (2018) la detección de objetos es una extensión natural del problema de clasificación con el reto agregado de detectar y localizar con precisión la presencia de uno o varios objetos en una imagen, encerrándolos en un cuadro delimitador y evitando detecciones falsas debido al fondo de la imagen o a detecciones múltiples del mismo objeto. Entendiendo una clase como una categoría de clasificación, este problema de detección de objetos en imágenes se aborda como un problema de clasificación multiclase. En un problema multiclase, cada objeto etiquetado (clase) en cada imagen de entrenamiento pertenece a sólo una de *N* diferentes clases, siendo el objetivo la construcción de una función que, dado un objeto nuevo dentro de una imagen, pueda predecir la clase más probable a la que ese objeto pertenece (Rifkin, 2008). Para los modelos de detección de objetos, la precisión de dicha predicción suele evaluarse mediante la métrica mAP (*mean Average Precision*), calculada con base en la IOU (*Intersection Over Union*) (§3.4) para la localización, y AP (*Average Precision*) para la clasificación, la cual se calcula por medio de la relación de detecciones de objetos verdaderas para el número total de objetos que predijo el clasificador y el número total de objetos en la base de datos de entrenamiento (Arlen, 2018).

A continuación, se describirán diferentes modelos CNN que pueden ser aplicados para la detección de objetos más conocidos con buenos resultados. Posteriormente, en la siguiente sección se explicará con más detalle el modelo CNN elegido para realizar la implementación del sistema de iluminación inteligente.

**3.3.1. SSD.** Uno de los sistemas de detección de objetos más conocidos es SSD (*Single Shot Multi-Box Detector*) (Liu et al., 2016). Este sistema discretiza los valores de ubicación de los cuadros delimitadores de la salida en un conjunto de cuadros predeterminados sobre diferentes relaciones de aspecto y escalas por ubicación de mapa de características. En la predicción, la red genera puntajes para la presencia de cada categoría de objeto dentro de cada uno de los cuadros predeterminados, generando ajustes para que tales cuadros rodeen de la mejor manera posible a los objetos detectados por medio de la aplicación de pequeños filtros convolucionales aplicados a los mapas de características (*feature mapping*). Además, para detectar objetos de diferentes tamaños, la red combina predicciones de múltiples mapas de características utilizando distintas resoluciones. A diferencia de los métodos que tienen etapas para pre-localizar objetos, SSD evita el procesamiento extra que eso conlleva y efectúa la predicción con una sola red neuronal. La red base para este sistema es VGG16 (Simonyan & Zisserman, 2015), al cual le hicieron cambios en capas y filtros, además de un proceso de *fine-tuning.*

**3.3.2. DSSD.** Al año siguiente de la propuesta de SSD, a mediados de 2017, se publica DSSD (*Deconvolutional Single Shot Detector*) (Fu, Liu, Ranga, Tyagi, & Berg, 2017) como una versión aumentada de SSD, cuya mejora principal consiste en introducir un contexto adicional a la detección de objetos. Para alcanzar ésto, el sistema SSD anterior se combinó con el clasificador *Residual-101* (reemplazando al modelo anterior VGG-19) para después agregar capas deconvolucionales que permiten introducir un contexto adicional y mejorar la precisión, especialmente para objetos pequeños.

**3.3.3. CSSD.** Otro trabajo similar, también orientado a la adición de contexto, basado en SSD y VGG, es CSSD (*Context-aware SSD*) (Xiang, Zhang, Yu, & Athitsos, 2018), el cual

además de implementar capas deconvolucionales también usa capas convolucionales dilatadas, logrando también mejorar la detección de objetos pequeños.

**3.3.4. SSDLite.** Posterior a la propuesta de SSD, surge SSDLite (Sandler, Howard, Zhu, Zhmoginov, & Chen, 2018), que tiene como base el clasificador MobileNetV2, donde tanto el clasificador como el sistema de detección de objetos son publicados en el mismo documento por los autores, quienes se dieron a la tarea de mejorar el modelo MobileNet, cuya característica principal es el uso de capas del tipo *depth-wise separable convolution*, adicionándolo al sistema SSD, logrando mejorar los resultados de los sistemas anteriores en términos de precisión (22.1% mAP) y complejidad computacional (0.8 BMAdd), equiparando a YOLOv2 en precisión pero alrededor de 20 veces más eficiente y 10 veces más compacto.

**3.3.5. Faster R-CNN.** Este detector utiliza una red neuronal que genera propuestas de regiones donde es probable encontrar algún objeto, transfiriendo luego los resultados a una red neuronal que se encarga de la detección (Ren, He, Girshick, & Sun, 2016). La red que propone las regiones es completamente convolucional, simultáneamente predice los límites de localización del objeto y puntuaciones con un nivel de certeza de que exista un objeto dentro de una posición. El modelo CNN que se usa como base es VGG-16, el sistema se evalúa en términos de cuadros por segundo (FPS) y precisión con respecto a las bases de datos PASCAL VOC y MS COCO.

**3.3.6. R-FCN.** Similar a Faster-CNN, R-FCN (*Region-based Fully Convolutional Networks*), sigue la estrategia de proponer regiones y después llevar a cabo la clasificación en aquellas con mayor probabilidad de contener un objeto (Dai, Li, He, & Sun, 2016). Tanto la red que propone las regiones como la de clasificación son completamente convolucionales. Al final de la red colocan una capa de *pooling* selectiva basándose en las probabilidades que tiene cada región de contener un objeto, esto guía a la última capa convolucional a aprender características más específicas con base en la posición. El modelo base que utilizan es una de las llamadas *residual networks,* específicamente ResNet-101.

**3.3.7. Pelee.** Dentro de los trabajos más recientes, se encuentra Pelee (Wang, Bohn, & Ling, 2018), que es una combinación de SSD y PeleeNet. Este último es un modelo de red

neuronal propuesto por los mismos autores, quienes remarcan que esta red está diseñada para su uso en dispositivos móviles, ya que a diferencia de redes como MobileNet, esta no hace uso de capas del tipo *depth-wise separable convolution*, la cual no es compatible con *frameworks* como OpenVino y CoreML dificultando su implementación en dispositivos móviles y de IoT. Pelee sólo requiere capas convolucionales normales, alcanzando mejor precisión, mayor rapidez y menor tamaño que MobileNet. La combinación de esta red y SSD alcanzó buenos resultados en comparación con sistemas como TinyYOLO, SSD+MobileNet y SSDLite+MobilNetv2, demostrando que no solamente con *depth-wise separable convolution* se pueden construir modelos eficientes.

**3.3.8. Tiny-DSOD.** Por último, conscientes de que los sistemas de detección de objetos requieren una alta capacidad de cómputo y puede resultar complicado lograr su adecuada implementación en dispositivos de recursos limitados, Li et al. (2018) propone Tiny-DSOD, basándose en su trabajo anterior DSOD (*Deeply Supervised Object Detector*) (Shen et al., 2017) y pensando específicamente en un uso eficiente para dispositivos de recursos limitados. Tiny-DSOD está basado principalmente en dos principios novedosos y eficientes, *depth-wise dense block* y *depth-wise feature-pyramid-network.* Los autores destacan que en comparación con sistemas como Tiny-YOLO, SSD+MobileNet (v1 y v2) y Pelee, Tiny-DSOD obtiene mejores resultados en términos de parámetros, operaciones de punto flotante y precisión, alcanzando 72.1% mAP, 0.95M de parámetros y 1.06B FLOPs.

## 3.4. Detección de Objetos usando Arquitectura YOLO.

En el 2016 surge YOLO, que por sus siglas en inglés significa *You Only Look Once*, como un nuevo enfoque para la detección de objetos (Redmon, Divvala, Girshick, & Farhadi, 2016). Sus autores enmarcan la detección de objetos como un problema de regresión para los cuadros delimitadores (*bounding boxes*) y sus respectivas probabilidades de clase asociadas. A diferencia de métodos como *sliding window* (Nakahara, Yonekawa, & Sato, 2017), donde el algoritmo recorre una ventana a través de la imagen completa buscando espacio por espacio

alguna clase que el clasificador reconozca, o como el algoritmo *region proposal* (Dong & Wang, 2016), donde primero con un algoritmo más simple se proponen una cierta cantidad de cuadros delimitadores dentro de la imagen y después se ejecuta el clasificador de mayor desempeño solamente en los cuadros propuestos, en YOLO es diferente, una sola red neuronal, basada en el *framework* Darknet, predice estos cuadros y probabilidades directamente de imágenes completas mediante una sola evaluación. El diseño de esta red permite un entrenamiento de tipo *end-to-end* (Henderson & Ferrari, 2016), el cual consiste en poder entrenar la red completa, es decir, sin necesidad de efectuar un proceso de modularización durante su entrenamiento, lo cual se traduce en ventajas en términos de mayor velocidad de ejecución manteniendo un alto promedio de precisión. Para evaluar este algoritmo con la base de datos PASCAL VOC, una de las más populares bases de datos para la detección de objetos, el sistema divide la imagen de entrada en una cuadrícula de 7 x 7, prediciendo en cada celda dos cuadros delimitadores y sus respectivas puntuaciones de confianza. Estas puntuaciones se calculan con base en la seguridad que tiene el modelo de que en ese cuadro hay un objeto y a la precisión con la cual ese cuadro ocupe el espacio donde está el objeto, esta precisión se obtiene calculando la función IOU (*Intersection Over Union*), es decir, dividiendo el área de intersección (entre el cuadro de la predicción y el real provisto previamente por el conjunto de datos de evaluación) entre el área de unión (de los mismos cuadros). Cada uno de los cuadros delimitadores consiste de cinco predicciones: las coordenadas *(x, y)* del centro del cuadro con relación a los límites de la celda de la cuadrícula, (*w, y*) como ancho y altura del cuadro con respecto a la imagen completa, y por último la probabilidad de la predicción representada por la función IOU. Además, para cada celda el sistema predice 20 probabilidades, una para cada clase de la base de datos. Por lo tanto, en la predicción final se obtiene un tensor (Kolecki, 2002) de 7 x 7 x 30.

Este modelo se implementó como una red CNN, pre-entrenando las primeras 20 capas convolucionales con la base de datos ImageNet que contiene 1,000 clases, y agregándose para el entrenamiento con PASCAL VOC cuatro capas convolucionales y dos capas *fully connected*. Además, generaron otra versión, actualmente identificada como TinyYOLO, la cual cuenta con sólo 9 capas convolucionales, más rápida, pero de menor precisión (esta versión compacta también la continuaron generando para las siguientes versiones). Los resultados obtenidos de

aplicar YOLO para detección de objetos, resultaron ser de los más rápidos disponibles en su momento. Sin embargo, con baja precisión en la detección, siendo la localización de objetos el punto más débil. Atendiendo este punto, dichas limitantes fueron atendidas dando lugar a una nueva versión YOLOv2.

En YOLOv2 (Redmon & Farhadi, 2016), una de las mejoras consistió en agregar a cada capa convolucional, un proceso de *batch normalization* durante el entrenamiento, mejorando así la convergencia de la red e incrementando ligeramente su precisión (2%). Otra mejora consistió en utilizar un clasificador de mayor resolución (448 x 448), con el cual la precisión también mejoró (4%). Además, en lugar de predecir las coordenadas de los cuadros delimitadores, se predicen compensaciones con respecto a cuadros delimitadores anclados, siendo más fácil de aprender para la red. Estos cuadros anclados, a diferencia de algoritmos donde son seleccionados a mano, son seleccionados mediante el algoritmo *k-means clustering* para encontrar automáticamente las mejores opciones. Aún así, el sistema presentó inestabilidad con respecto al uso de estos cuadros delimitadores anclados, por lo que optaron por implementar el método del sistema anterior (YOLO), calculando las compensaciones con respecto a la cuadrícula de división de la imagen. Otra característica de YOLOv2, es lograr buenas predicciones con diferentes tamaños de entrada, esto lo logran cambiando aleatoriamente el tamaño de la entrada durante el entrenamiento. Con esto la red aprende a hacer las predicciones aún con estos cambios, lo cual ofrece ventajas para intercambiar velocidad y precisión con base en el tamaño de la imagen de entrada. Darknet-19 es el nombre que los autores le dieron a la red neuronal, la cual cuenta con 19 capas convolucionales y 5 capas *maxpooling*. Junto con YOLOv2, los autores presentan YOLO9000, basado en YOLOv2 y entrenado con las bases de datos de COCO e ImageNet, logrando detectar más de 9,000 clases a pesar de que los datos de clasificación no cuentan con información de cuadros delimitadores.

Hasta el momento, la versión más reciente de este sistema, publicada en 2018, YOLOv3 (Redmon & Farhadi, 2018) utiliza regresión logística para calcular un puntaje de objetividad de cada cuadro delimitador, ayudándose del cuadro delimitador anterior para asegurar el resultado de la predicción. La red neuronal base para esta versión es Darknet-53, la cual cuenta con 53 capas convolucionales, con mayor precisión que Darknet-19 y naturalmente menor velocidad,

pero aún así mayor a modelos como ResNet. Además, esta versión calcula los cuadros delimitadores en tres escalas (multi-escala), que ayuda a obtener mayor información semántica e información más detallada al combinar los resultados de las distintas escalas. Para esta última versión de YOLO el sistema aún tiene problemas para localizar de manera precisa objetos, los autores siguen investigando posibles mejoras, pero aún así el sistema se presenta como una opción rápida y eficiente para la detección de objetos.

Un resumen y comparación entre los detectores de objetos que se han mencionado se presenta la Tabla 3.1, basada en el trabajo de Li et al. (2018), en donde se llevaron a cabo los experimentos con la base de datos PASCAL VOC 2007 haciendo uso de una GPU Nvidia Titan X.

Tabla 3.1. Comparación entre detectores de objetos de Li et al. (2018).

| Método | Tamaño de entrada | Backbone | FPS | #Parámetros | FLOPs | mAP(%) |
|---|---|---|---|---|---|---|
| Faster-RCNN | 600 × 1000 | VGGNet | 7 | 134.70M | 181.12B | 73.2 |
| R-FCN | 600 × 1000 | ResNet-50 | 11 | 31.90M | - | 77.4 |
| SSD | 300 × 300 | VGGNet | 46 | 26.30M | 31.75B | 77.2 |
| YOLO | 448 × 448 | - | 45 | 188.25M | 40.19B | 63.4 |
| YOLOv2 | 416 × 416 | Darknet-19 | 67 | 48.20M | 34.90B | 76.8 |
| DSOD | 300 × 300 | DS/64-192-48-1 | 17.4 | 14.80M | 15.07B | 77.7 |
| Tiny-YOLO | 416 × 416 | - | 207 | 15.12M | 6.97B | 57.1 |
| MobileNet-SSD | 300 × 300 | MobileNet | 59.3 | 5.50M | 1.14B | 68 |
| Pelee | 300 × 300 | PeleeNet | - | 5.98M | 1.21B | 70.9 |
| Tiny-DSOD | 300 × 300 | G/32-48-64-80 | 105 | 0.95M | 1.06B | 72.1 |

## 3.5. Técnicas de Adaptación para modelos CNN en la Frontera.

Existen una diversidad de trabajos y esfuerzos de investigación muy recientes que se llevan a cabo con el fin de lograr ejecutar CNN's en dispositivos de recursos limitados de procesamiento y almacenamiento. En este trabajo se agruparon dichos trabajos bajo el nombre de técnicas de adaptación. Producto de la revisión sistemática de la literatura sobre este tema, siguiendo la metodología descrita en el ANEXO 1, se han recopilado y seleccionado los trabajos más representativos y pertinentes. A continuación, se describen algunos de los distintos enfoques que han surgido para el desarrollo y aplicación de algunas técnicas de adaptación de modelos DNN, tanto a nivel de su implementación, como para la creación de modelos más eficientes.

Cuando se habla de soluciones a nivel implementación se trabaja con los modelos DNN originales, es decir, sin hacerse algún cambio sobre ellos y solamente trabajando con la manera en que se implementa dicho modelo. Ejemplos de estas técnicas son la paralelización de inferencias utilizando los GPU's de los dispositivos móviles (Motamedi, Fong, & Ghiasi, 2016, 2017), la granularidad de hilos concurrentes (Motamedi et al., 2017), el método FFT (Kim et al., 2015), la optimización del código nativo del CPU (Kim et al., 2015), la aceleración basada en unidades de coprocesamiento ASIC, FPGA y SoC (Motamedi et al., 2017), la optimización de lazo (*loop optimization*) (Ma, Cao, Vrudhula, & Seo, 2018), entre otros. Por el otro lado, y siendo la rama de estudio central del presente trabajo, se encuentran los métodos con enfoque en la creación de modelos eficientes, donde las técnicas más usadas se pueden clasificar en dos tipos, las de compresión y aceleración de redes pre-entrenadas o las de entrenamiento de redes compactas directamente (Howard et al., 2017). Entre los métodos más comunes de aceleración y compresión están SVD (*Singular Value Decomposition*) (Iandola et al., 2016; Kim et al., 2015), podar parámetros de redes DNN (Cheng et al., 2017; Howard et al., 2017; Iandola et al., 2016; Kim et al., 2015), la factorización de bajo rango o descomposición del tensor (Cheng et al., 2017; Kim et al., 2015; Wu et al., 2016), la cuantización (Andri et al., 2018; Cheng et al., 2017; Kim et al., 2015; Wu et al., 2016), y desde luego, la combinación de diversas técnicas para construir métodos híbridos. Por último, con el enfoque del entrenamiento directo de redes

compactas, los métodos que más se destacan en la literatura son la transferencia de aprendizaje o *transfer learning* (Li & Hoiem, 2016; Pan & Yang, 2010) y la destilación (Cheng et al., 2017; Howard et al., 2017; Li & Hoiem, 2016). Un resumen y relación de estas técnicas se presenta en la Figura 3.6, además en la siguiente sección se explican las técnicas de adaptación que se consideraron para este trabajo de tesis.

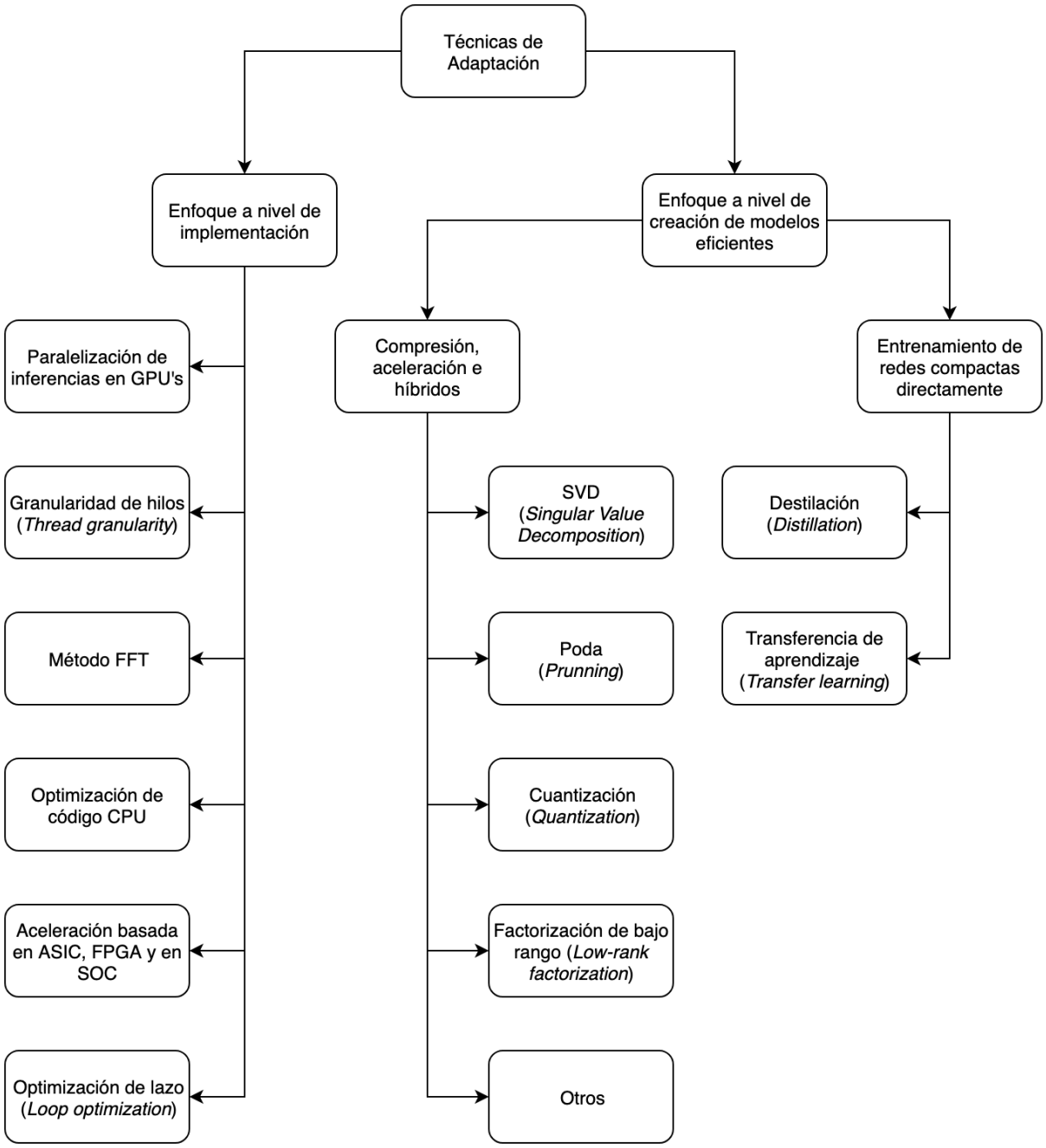


Figura 3.6. Taxonomía de Técnicas de Adaptación.

### 3.5.1. Poda.

Muchos investigadores han encontrado que los modelos DNN sufren de una alta sobre-parametrización, y aunque esto parece ser necesario durante el entrenamiento, es posible analizar sistemáticamente para reducir el tamaño del modelo después del entrenamiento (Luo, Wu, & Lin, 2017). El concepto de poda de redes (*network pruning*) fue usado por primera vez por Le Cun, Denker, & Solla (1989), estos métodos de poda de parámetros exploran la redundancia en los parámetros de un modelo e intentan remover aquellos que no son críticos, es decir, los que no son sensitivos para el desempeño (Cheng et al., 2017). Por ello es que la poda de redes se considera un esquema de compresión, colocando en cero los pesos que tienen la menor influencia en la precisión de la red y siendo necesario en muchos casos reentrenar la red para recuperar pérdidas de precisión (Pilipovic, Bulic, & Risojevic, 2018).

El proceso de poda se puede efectuar sobre los canales (son las entradas del filtro, que corresponden a las salidas de la capa anterior), removiendo de cada filtro los *kernels* que sean menos importantes (Zhuang et al., 2018) o sobre el filtro completo, dejando a la respectiva capa con menos unidades, haciendo la red más estrecha (Luo et al., 2017). De acuerdo con la revisión de literatura (Han, Mao, & Dally, 2016; Luo et al., 2017; O'Keeffe & Villing, 2018; Pilipovic et al., 2018), la metodología de poda se puede resumir de la siguiente manera:

1. Evaluar importancia de *kernels* o filtros basándose en alguna métrica.
2. Remover los *kernels* o filtros menos significativos.
3. Reentrenar la red o aplicar *fine-tuning* para recuperar precisión.
4. Si lo anterior sólo se aplicó a una capa, iterar para las demás.

En el proceso de poda de la red, el problema más difícil es determinar cuáles pesos deben eliminarse y cuáles no (Pilipovic et al., 2018). Para resolverlo, se reconocen dos principales enfoques, el de independencia de datos y el de dependencia de datos (O'Keeffe & Villing, 2018), es decir, si se requiere llevar a cabo o no un proceso adicional de entrenamiento para evaluar la importancia de los pesos.

**Independencia de Datos**. En este caso, probablemente el método más simple es el basado en la magnitud de los pesos, midiendo la importancia de cada filtro a través de la suma del valor absoluto de los pesos (Li, Kadav, Durdanovic, Samet, & Graf, 2017). Otra técnica consiste en tener en cuenta las activaciones después de la función de activación, tomando el cálculo del porcentaje promedio de ceros (APoZ) de cada filtro como métrica de selección (Hu, Peng, Tai, & Tang, 2016). También existen los métodos basados en reconstrucción, eliminando pesos y comparando el error respecto al error del modelo original para saber si los pesos eliminados eran importantes (Luo et al., 2017; Pilipovic et al., 2018), la reconstrucción se aborda como un problema de optimización que se puede resolver con técnicas como algoritmos *greedy* o una regresión LASSO (Zhuang et al., 2018).

**Dependencia de Datos**. También llamados métodos basados en regularización (Wang, Zhang, Wang, Lu, & Hu, 2019) o basados en entrenamiento (Alvarez & Salzmann, 2016), consisten en agregar regularización a los pesos que se desee evaluar su importancia para que en el entrenamiento la función de costo determine cuáles pesos podar (Cheng et al., 2017; Zhuang et al., 2018), los pesos que tengan menor impacto en la precisión serán más cercanos a cero después del entrenamiento y los que sí son importantes sólo habrán tenido un ligero decremento en su valor (Pilipovic et al., 2018). Los criterios más comunes para la regularización son la norma L1, la norma L2, APoZ y expansiones de Taylor (Cheng et al., 2017; Molchanov, Tyree, Karras, Aila, & Kautz, 2017; Pilipovic et al., 2018; Wang et al., 2019).

Es relevante mencionar que para algunos de los métodos independientes de datos, la convergencia del algoritmo de selección es lenta en comparación con los métodos dependientes de datos (Pilipovic, Bulic, & Risojevic, 2018), aunque por otro lado, en estos métodos dependientes de datos el proceso es computacionalmente costoso, sobretodo para redes muy profundas y/o para redes que se entrenen con grandes bases de datos (Cheng et al., 2017; Zhuang et al., 2018), pero ofrecen la ventaja de alcanzar mejores resultados que los métodos independientes de datos (O'Keeffe & Villing, 2018).

### 3.5.2. Transferencia de Aprendizaje.

Tradicionalmente, los algoritmos de aprendizaje para redes DNN están diseñados para abordar tareas o problemas de manera aislada, es decir, para cada tarea que se desee aprender es necesario un proceso de aprendizaje diferente en cada caso. La técnica de transferencia de aprendizaje (*transfer learning*, TL) busca ir más allá de ese concepto y utilizar el conocimiento adquirido en una tarea para solucionar otras tareas relacionadas. Por ejemplo, en clasificación de imágenes, ciertas características de bajo nivel como bordes, formas o iluminación pueden ser compartidas entre una tarea y otra, siendo así útil poder transferir ese conocimiento entre ellas (Sarkar, Bali, & Ghosh, 2018). Además, hay casos donde se desea llevar a cabo una tarea en cierto dominio de interés, pero sólo hay suficientes datos para entrenamiento en otro dominio, incluso pueden estar en un espacio de características distinto o tener una distribución de datos diferente. En esos casos, implementar TL de manera efectiva ayudaría enormemente a mejorar el desempeño del proceso de aprendizaje, evitando costosos esfuerzos para etiquetar información del dominio deseado (Pan & Yang, 2010), además de lograr un mejor desempeño base, un mejor desempeño final y reducir el tiempo necesario para el aprendizaje del modelo DNN (Sarkar et al., 2018).

Para llevar a cabo esta técnica de TL, existen dos principales enfoques, explicados por Chollet (2017) y Li & Hoiem (2016) de la siguiente manera:

- *Feature extraction*. Este enfoque consiste en utilizar las representaciones tal cual fueron aprendidas por una red DNN anterior para extraer características a partir de nuevas muestras. Estas características se ejecutan a través de un nuevo clasificador, que se entrena desde cero. Para este proceso normalmente se usan las bases (son todas las capas anteriores a la capa de clasificación, encargadas de la extracción de caracteriśticas) de redes consolidadas como VGG o InceptionV3, estas bases se “congelan”, es decir, se configuran las capas como “no entrenables” para que los pesos en ellas no sean modificados y solamente se modifiquen los pesos del clasificador.

- *Fine-tuning*. Consiste en descongelar una o dos de las capas superiores de una base de modelo DNN congelada utilizada para *feature extraction*, y entrenar conjuntamente la parte recién agregada del modelo DNN (en este caso el clasificador) y estas capas superiores. Esto se denomina *fine-tuning* porque ajusta levemente las representaciones más abstractas del modelo que se reutiliza, haciéndolas más relevantes para el problema en cuestión. Normalmente se usa una taza de aprendizaje baja para prevenir grandes cambios en los pesos de las capas descongeladas.

También es importante mencionar que cuando el conjunto de datos del dominio de destino es pequeño con respecto a la capacidad de la red, el proceso de *fine-tuning* puede terminar en sobre-entrenamiento. Además, ha sido demostrado que las últimas capas de una red son específicas de la tarea, mientras que las primeras son específicas de la modalidad (Castrejon, Aytar, Vondrick, Pirsiavash, & Torralba, 2016). Es por ello que incluso la decisión de cuántas capas transferir puede ser complicada, ya que esto depende tanto de la proximidad entre las dos tareas involucradas como de la proximidad entre las correspondientes modalidades de imagen (Christodoulidis, Anthimopoulos, Ebner, Christe, & Mougiakakou, 2017).

TL es una técnica que ha sido aplicada y estudiada en el contexto del aprendizaje inductivo, como es el caso de las redes neuronales, redes bayesianas y aprendizaje reforzado, por lo que este concepto puede aplicarse en todas ellas, es decir, no está limitado su uso exclusivo para modelos DL (Sarkar et al., 2018).

### 3.5.3. Cuantización.

Cuantización de redes DNN consiste en comprimir la red original por medio de reducir el número de bits requeridos para representar cada peso (Cheng et al., 2017). Debido a que cada peso utiliza menos bits, el efecto se puede entender como una agrupación de pesos que compartirán los mismos valores con la cuantización (Han et al., 2016), para determinar tales agrupaciones se emplean métodos como una función *hash* (Chen, Wilson, Tyree, Weinberger,

& Chen, 2015) o *k-means* (Han et al., 2016), con variantes como primero concatenar los parámetros de la red DNN en subvectores y luego aplicar el algoritmo de agrupación (Cheng, Wu, Leng, Wang, & Hu, 2018).

Una vez obtenidas las agrupaciones, es necesario determinar el valor que representa al conjunto, para ello se coloca un centroide, cuya localización impacta en la precisión del modelo DNN. Siendo este el mayor problema a resolver para esta técnica, la revisión de literatura realizada por (Han et al., 2016; Hubara, Courbariaux, Soudry, El-Yaniv, & Bengio, 2018; Pilipovic et al., 2018) muestra dos principales enfoques para ubicar los centroides: la cuantización lineal y la no lineal.

**Cuantización Lineal**. Los algoritmos basados en este enfoque asumen que los pesos del modelo DNN están distribuidos de manera uniforme (Pilipovic et a., 2018), de esa forma una distribución lineal de los pesos es bastante adecuada, colocando el centroide entre el valor mínimo y el máximo de cada agrupación (Han et al., 2016). En este enfoque también se considera la conversión de representaciones de pesos de punto flotante a punto fijo (Pilipovic et a., 2018). Un caso extremo de cuantización lineal es el de pesos binarios, donde la representación se hace sólo con un bit de precisión, técnica que ha tenido amplio interés de investigación para dispositivos muy limitados (Cheng et al., 2017; Pilipovic et al., 2018).

**Cuantización No lineal**. Si se determina que los valores de los pesos de una red neuronal no cuentan con una distribución uniforme (Pilipovic et al., 2018), es posible aplicar algoritmos de cuantización no lineales para obtener mejores resultados, al tener menores pérdidas en la precisión (Hubara et al., 2018), siendo la representación logarítmica el algoritmo con los mejores resultados (Hubara et al., 2018; Miyashita, Lee, & Murmann, 2016; Pilipovic et al., 2018).

Sea lineal o no lineal, la representación de pesos con menos bits resulta en un menor consumo de memoria y de energía, pero también resulta en pérdidas de precisión en la inferencia, de ahí la búsqueda por encontrar la mejor combinación de precisión y cantidad de bits para representar los pesos (Pilipovic et al., 2018). Por ejemplo, Li et al. (2019) aplica una

técnica de cuantización que convierte parámetros de 32 bits de una red CNN hasta una resolución de 4 bits, es decir reduciendo 8 veces el consumo de memoria, presentando tan sólo una pequeña pérdida de precisión de solamente 2% mAP.

# IV. IMPLEMENTACIÓN Y PRUEBAS DEL MODELO CNN ORIGINAL

En este capítulo se describe la investigación sobre las primeras etapas de implementación y pruebas de modelos CNN para la detección de objetos para el escenario del sistema de iluminación inteligente, tanto a nivel de imágenes estáticas como para flujos continuos de video (*video streaming*), sin llevar a cabo ninguna alteración que vaya más allá de las conversión del formato de los archivos para el modelo CNN pre-entrenado entre los distintos *frameworks* y plataformas requeridas para la validación del mismo, tanto para las arquitecturas de cómputo en la frontera como para cómputo en la niebla. En este apartado se cubren la mayoría de los requerimientos (§1.4) que contribuyen además a proporcionar los resultados que corresponden a la primera de dos partes que son fundamentales para validar la hipótesis del trabajo, es decir, las métricas de rendimiento para la etapa de inferencia en dispositivos de hardware limitado ejecutando modelos CNN sin adaptaciones o modificaciones en la arquitectura o componentes internos del modelo CNN original, para posteriormente aplicar diversas técnicas de adaptación y realizar las mismas mediciones correspondientes bajo los mismos escenarios de prueba (capítulo 5 y 6) y finalmente, analizar y comparar los resultados obtenidos (capítulo 7). En resumen, la sección 4.1 describe las primeras pruebas de los modelos CNN usando dispositivos móviles (requerimiento 1.4.5); la sección 4.2 explica la solución al requerimiento de selección del modelo CNN para detección de objetos (requerimiento 1.4.4); en la sección 4.3 se discute la problemática presentada debido a las conversiones y aspectos de compatibilidad a considerar para lograr ejecutar el modelo CNN seleccionado en las distintas plataformas de implementación (requerimientos 1.4.3 - 1.4.5); en la sección 4.4 se describen las consideraciones y problemáticas de implementación propias de cada plataforma de procesamiento de cómputo en la frontera y la niebla (requerimiento 1.4.5); finalmente, la sección 4.6 describe la aplicación y escenario de prueba implementado para el sistema de iluminación inteligente basado en el modelo CNN elegido y probado para la detección de personas, así como la medición y obtención de resultados que servirán para cubrir posteriormente el requerimiento

1.4.6. En la Figura 4.1 se ilustra dicho procedimiento de experimentación representado como un diagrama a bloques.

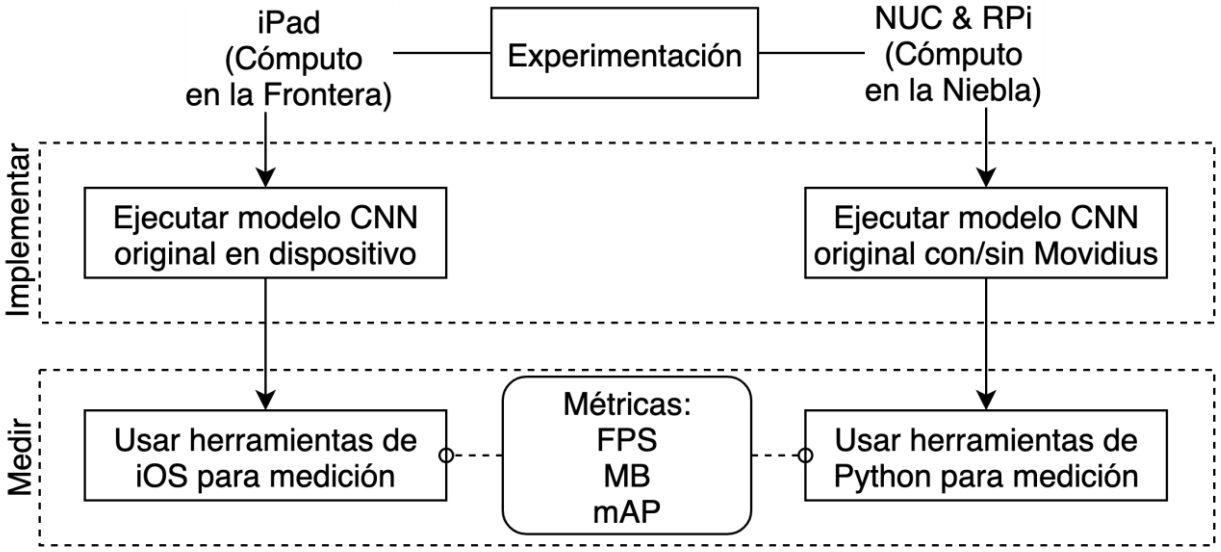


Figura 4.1. Experimentación con modelo CNN original para Cómputo en la Frontera y en la Niebla.

## 4.1. Pruebas Iniciales: Etapa de Inferencia CNN en Cómputo en la Frontera.

El objetivo de las pruebas iniciales consistió en comprender el proceso y manejo de diversas herramientas involucradas en la entonces inusual implementación de aplicaciones móviles capaces de ejecutar modelos CNN pre-entrenados no provistos por el propio fabricante del dispositivo móvil (finales de 2017 y principios de 2018). Antes de implementar el sistema completo de iluminación inteligente, se procedió primero a verificar si resultaba o no viable dicha capacidad de ejecución en los dispositivos. Por tanto, se realizó y publicó un estudio comparativo del desempeño de diferentes modelos CNN para clasificar objetos analizando imágenes tomadas directamente desde la cámara de un dispositivo móvil, para verificar si la etapa de inferencia de la red neuronal, que se caracteriza por su gran exigencia de poder de cómputo y almacenamiento, puede ser realizada en dispositivos que forman parte de la arquitectura de cómputo en la frontera.

Para la aplicación de estas pruebas se desarrolló un prototipo de aplicación móvil, cuyo desarrollo y resultados fueron publicados en Pacheco, Flores, Sanchez, & Almanza (2018). El prototipo se implementó para dispositivos móviles capaces de correr la plataforma iOS 12. La aplicación móvil es capaz de detectar diversos objetos usando modelos CNN pre-entrenados que fueron convertidos al formato `.mlmodel` usando herramientas provistas por Apple para efectuar dicha conversión.

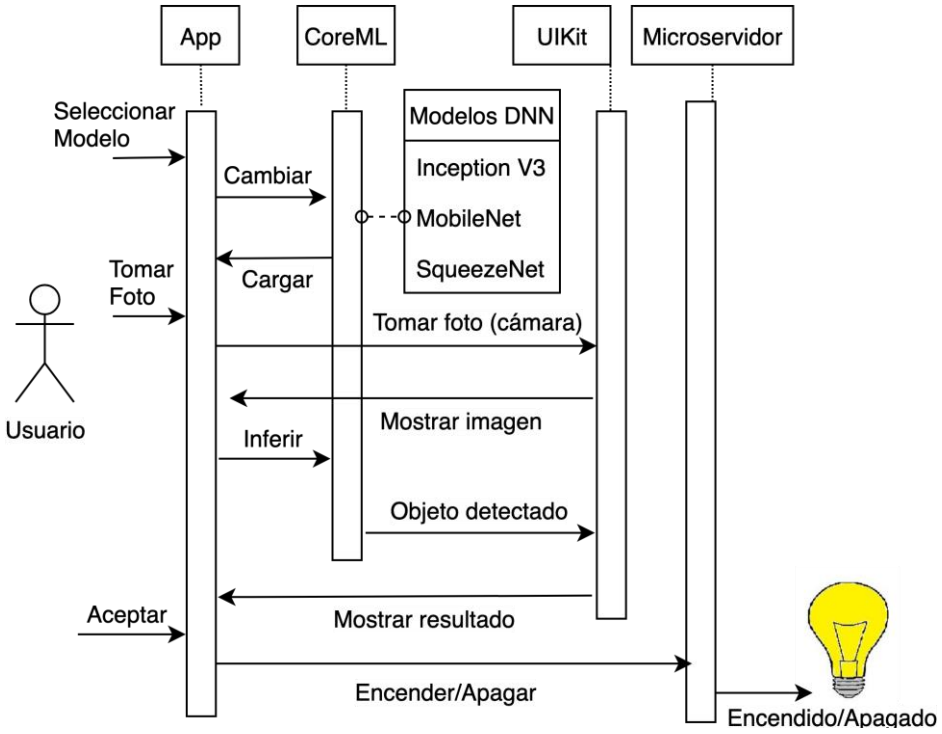


Figura 4.2. Diagrama de secuencia UML de aplicación iOS para pruebas iniciales. Adaptado de Pacheco et al. (2018).

La Figura 4.2 muestra un diagrama de secuencia UML que describe cómo se efectúa cronológicamente cada tarea para cada "rol" o componente clave de la aplicación móvil de iOS desarrollada. La Tabla 4.1 presenta los resultados obtenidos para los tres modelos CNN que corrieron con éxito, la cuarta columna corresponde a la precisión oficial (Top1) utilizando el dataset de entrenamiento ImageNet.

Tabla 4.1. Resultados aplicación iOS ejecutando modelos CNN. Adaptado de (Pacheco et al., 2018).

| Modelo CNN (1,000 clases) | Total Capas | Tamaño (MB) | Precisión (%) | Velocidad Prom. (FPS) |
|---|---|---|---|---|
| InceptionV3 | 48 | 94.7 | 77.0 | 1.5 |
| MobileNet | 28 | 17.1 | 70.6 | 9.9 |
| SqueezeNet | 18 | 4.8 | 57.5 | 15.5 |

Al iniciar la aplicación móvil de iOS, primero se elige uno de los tres modelos CNN a probar, luego se toma una foto con la cámara del teléfono usando el API del *framework* UIKit. La aplicación móvil ejecuta la red neuronal elegida con la imagen captada usando el API del *framework* CoreML (Apple Inc., 2019a), de donde se obtiene y despliega una etiqueta (texto) y un porcentaje (número) que especifican: 1) la categoría de clasificación con mayor probabilidad para el objeto encontrado en la imagen captada, y 2) la precisión de la inferencia, respectivamente. En la Figura 4.3 se muestra la interfaz gráfica de usuario de la aplicación móvil mostrando como ejemplo un objeto clasificado usando el modelo CNN InceptionV3 para una imagen que contiene una foto del proyector multimedia del laboratorio.


Figura 4.3. Interfaz gráfica de aplicación iOS.

Este prototipo se implementó en Swift 4 para iOS 12 usando los *frameworks* de UIKit, Vision y CoreML de Xcode 10.1 y fueron evaluados en un teléfono inteligente iPhone 6. Entre los resultados más relevantes se observó que al ejecutar la aplicación móvil existe un extenso y tardado proceso de carga e inicialización de los modelos CNN, además para la primera inferencia de cada modelo CNN existe un mayor retardo (inicialización), misma que fue omitida para calcular el tiempo promedio de 200 inferencias. Las velocidades promedio de inferencia. En cuanto a velocidad de procesamiento de inferencias en imágenes analizadas por segundo (*frames per second*, FPS), aparecen reportadas en la última columna de la Tabla 4.1.

A pesar del tardado proceso de inicialización de los modelos, con estos resultados, especialmente los de velocidad de procesamiento para el caso de MobileNet y SqueezeNet, se logró demostrar la viabilidad de ejecución de inferencias de modelos CNN con cómputo en la frontera, ya que incluso los 9.9 FPS alcanzados por MobileNet satisfacen el requerimiento de velocidad de procesamiento establecido en 1.4.6. Sin embargo, existe también la posibilidad de implementación de modelos CNN como InceptionV3 que son más eficientes, a costa de un mayor consumo de memoria y demanda en la capacidad de procesamiento que puede resultar inadecuado para un dispositivo de baja capacidad. La aplicación móvil desarrollada en iOS para clasificar objetos captando imágenes desde la cámara integrada del dispositivo sirvió para hacer el estudio comparativo entre modelos CNN y con ello comprender el proceso requerido para implementar este tipo de modelos CNN en aplicaciones móviles, así como las herramientas necesarias. Con base en esta validación que atiende a la sección 1.5.2 de la metodología, se logró realizar una primer prueba de la hipótesis (sin considerar técnicas de adaptación de modelos CNN), sirviendo además de base para realizar la búsqueda y selección de nuevos modelos CNN (§1.5.3), que a diferencia de los clasificadores evaluados en esta sección, si cumplieran con el requerimiento de proporcionar información sobre la localización del objeto detectado (§1.4.4) y de procesar imágenes directamente de un flujo continuo de video requerido por el sistema de iluminación inteligente (§1.4.2).

## 4.2. Selección de Modelo CNN para Detección de Personas.

Se procedió a localizar modelos CNN exclusivamente pre-entrenados para detectar y localizar personas, pero no se encontraron modelos de código abierto y que fueran compatibles con las plataformas de prueba. Sin embargo, dado que los detectores de objetos CNN, dentro de sus múltiples clases, consideran la categoría de personas, se inició una segunda fase de experimentación para explorar y seleccionar el modelo CNN más adecuado para cubrir los requerimientos planteados para el presente trabajo (§1.4). Los resultados de la investigación realizada sobre el estado del arte de algoritmos CNN para detección de objetos han sido reportados en los apartados 3.2 - 3.3. La arquitectura YOLO como se describió en 3.3, puede efectuar la detección de forma continua tanto en imágenes como en *video-streaming* y además enmarca en una región rectangular (*bounding box*), la región en la imagen donde fue identificado el objeto.

El tipo de algoritmo de detección de objetos seleccionado fue TinyYOLO, para elegir la versión más adecuada se realizaron diversas pruebas (§4.3.3). A continuación, se enlistan los criterios considerados para seleccionar un modelo CNN para detectar personas:

a) Desempeño de la etapa de inferencias del modelo CNN. El principal criterio de selección fue usar el detector de objetos que procese la mayor cantidad de cuadros por segundo (FPS), debido a que la implementación se lleva a cabo en dispositivos de recursos limitados y una métrica de FPS alta implica una menor complejidad computacional, con lo cual se facilita la implementación del modelo CNN en tales dispositivos. Actualmente, TinyYOLO sigue siendo el mejor sistema con esa métrica (Li et al., 2018), seleccionado para pruebas en otros trabajos recientes por la relación precisión/fps que ofrece (Mittal, 2019).

b) Documentación disponible acerca del modelo CNN. En la investigación realizada, se encontró que la arquitectura YOLO dispone de una mayor documentación, publicaciones y trabajos de investigación.

c) Compatibilidad del modelo CNN para distintas plataformas y *frameworks* de desarrollo. La arquitectura YOLO es compatible con las plataformas de prueba y con los distintos *frameworks* de desarrollo para CNN disponibles, permitiendo de esta forma ejecutar dicho modelo en distintos dispositivos con hardware muy diverso e incompatible entre sí: x86 vs ARM, CPU vs GPU, GPU vs NPU, Linux vs iOS.

## 4.3. Validación de Interoperabilidad y Compatibilidad de TinyYOLO.

El proceso de validación del detector de objetos para verificar si es una opción apropiada y compatible para ser ejecutada en las distintas plataformas de prueba fue un proceso complicado por varias razones:

1) Las diferencias e incompatibilidades entre sistemas operativos y sus respectivas plataformas de hardware. Por un lado, iOS para nodos de cómputo *Edge* y por otro Raspbian o Ubuntu para nodos de cómputo *Fog*. También la incompatibilidad entre los *frameworks* de desarrollo para manipular modelos CNN, esto con el propósito de implementar un mismo modelo CNN al hardware específico de cada una de las plataformas de prueba, proceso que puede implicar más de una conversión y uso de herramientas, *frameworks* ó APIs;

2) La incompatibilidad a nivel de diferencias en tipos de capas soportadas por cada framework para ejecutar modelos DL en cada plataforma de hardware, ya que al momento de ser convertido y transferido al formato nativo para cada plataforma, lenguaje, SDK, API y *framework* de DL en cada nodo de procesamiento *Fog/Edge*, puede resultar que en alguno de ellos no sea posible ejecutar dicho modelo CNN, incluso cuando aparentemente existe dicho soporte en un *framework* determinado, dado que pueden presentarse incompatibilidades debido a la versión del sistema operativo, la biblioteca o el lenguaje de programación.

A continuación, se detallan los experimentos realizados en cada uno de los apartados mencionados para cada uno de los dispositivos, plataformas y escenarios de prueba.

### 4.3.1. Interoperabilidad entre Plataformas de Hardware y *Frameworks* de Software.

Con el fin de representar la compatibilidad y restricciones que existen en el proceso de implementación de CNN's en distintos dispositivos de la frontera y de la niebla, además de que a su vez sea una guía para futuras implementaciones, la Figura 4.4. recopila una buena parte de las posibles rutas de conversión, preparación e implementación (*deployment workflow*) de un modelo CNN específico para ser ejecutado en plataformas *Edge* ARM iOS/Android y plataformas *Fog* en x86 Linux y ARM Raspbian (sección 3 y 4), utilizando para ello algunos de los principales *frameworks* de desarrollo para DL (sección 1.1 y 1.2) para generar archivos del modelo CNN en distintos formatos (sección 1.3 y 1.4), para ser optimizado para una de unidad de procesamiento específica, ya sea CPU, GPU o acelerador neuronal (sección 2) donde se realizará la etapa de inferencia del modelo CNN dentro del dispositivo de prueba.

Al usar modelos CNN pre-entrenados, como es el caso de TinyYOLO, se presenta el problema de que para realizar con éxito el proceso de implementación surge la limitante del nivel de soporte que brinda un *framework* DL, y por lo tanto formato, para un modelo CNN específico. Generalmente, un modelo CNN de código abierto (*open source*) es implementado en un framework DL y una plataforma en particular, cuya versión puede estar descontinuada si es un modelo que ya tiene tiempo (uno o dos años de ser publicado), además, como es parte del objetivo del presente trabajo, la situación puede complicarse aún más si se desea implementar en más de una plataforma, ya que prácticamente cada compañía fabricante del dispositivo (hardware) maneja su propio *framework* DL para implementar modelos DL, eso nos lleva a tratar de usar aquellos que sean más compatibles tanto con otros *frameworks* como con la mayor cantidad de dispositivos, y así facilitar la interoperabilidad de la aplicación misma.

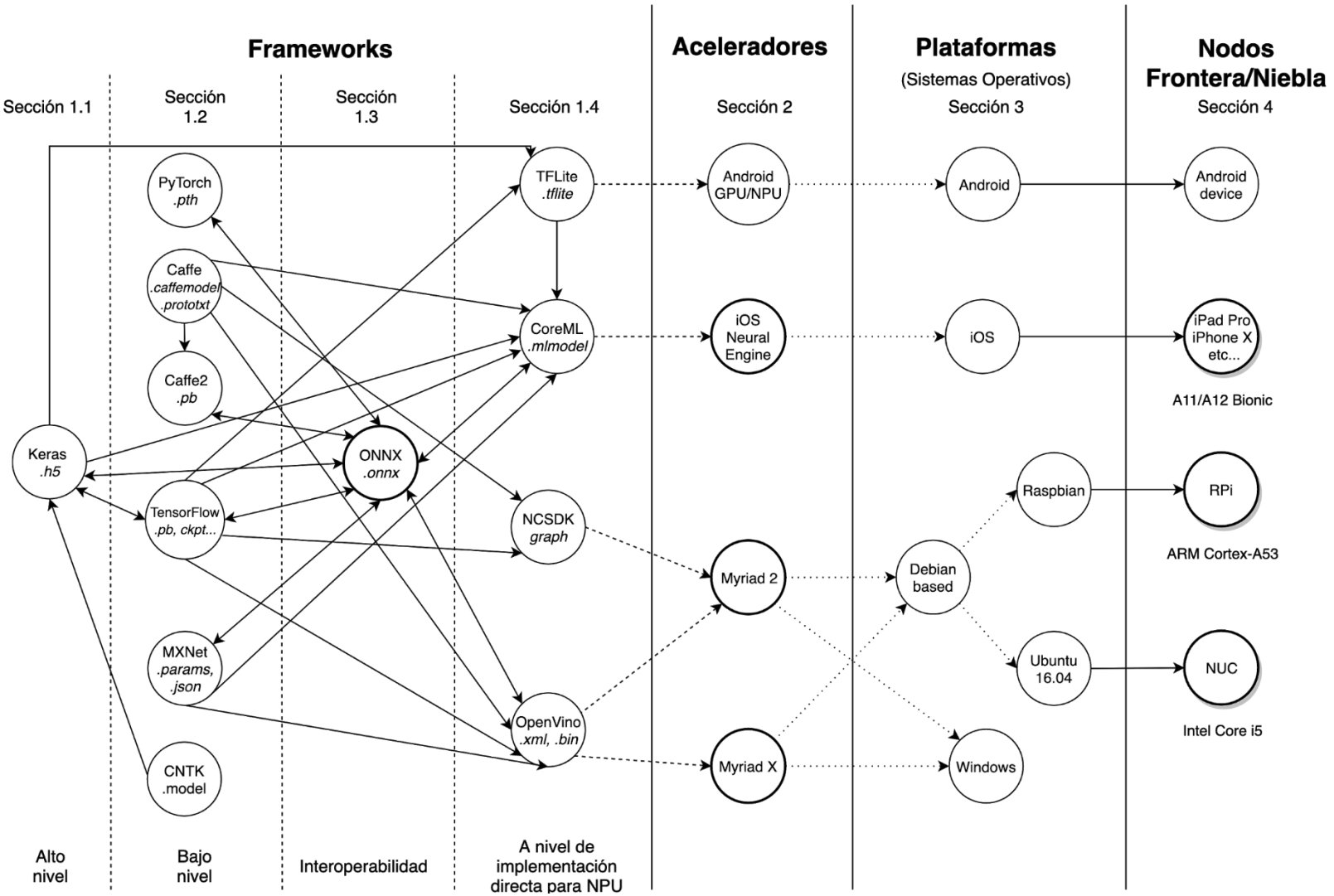


Figura 4.4. Diagrama de flujo de trabajo entre *frameworks*, aceleradores neuronales, sistemas operativos y nodos Edge/Fog.

En la Figura 4.4, el sentido de las líneas que interconectan indican la posibilidad de convertir un modelo de un *framework* a otro. Para portar un modelo CNN a una plataforma específica se recomienda realizar la menor cantidad de conversiones posible usando el menor número de *frameworks* posible, considerando además, que aunque exista soporte para la conversión, hay que tomar en cuenta la compatibilidad de capas y las operaciones disponibles de cada *framework* (§4.3.2). A continuación, se describe de manera más detallada cada sección del diagrama de compatibilidad de la Figura 4.4:

**Sección 1.1. Frameworks de alto nivel.** El entorno Keras permite importar e integrar facilidades de las demás herramientas de bajo nivel, es un framework muy accesible y útil ya que presenta muchas facilidades tanto para la creación de nuevos modelos CNN y también manipular modelos CNN pre-entrenados.

**Sección 1.2. Frameworks de bajo nivel.** Los *frameworks* DL como TensorFlow, CNTK y Theano son considerados de bajo nivel, se usan principalmente para la definición (capa por capa), entrenamiento y evaluación de modelos CNN.

**Sección 1.3. Formato ONNX.** Recientemente ONNX se perfila como una excelente opción para realizar conversiones entre distintos *frameworks* DL, básicamente si se requiere cambiar de algún framework a otro puede pensarse en ONNX, aunque como muestra el diagrama, muchas veces no es necesario ya que existen conversiones directas, y como se mostrará más adelante, entre menos conversiones se efectúen, mayor será la probabilidad de éxito.

**Sección 1.4. Conversiones a nivel de unidad de procesamiento.** Para ejecutar un modelo CNN es necesario especificar la unidad de procesamiento a utilizar (CPU, GPU, NPU) y realizar un proceso de adecuación del mismo que resulta obligado para poder realizar la etapa de inferencia en dicha unidad de hardware que deberá ser configurada para correr un determinado modelo CNN.

**Sección 2. Aceleradores neuronales.** Al buscar llevar a cabo la inferencia con cómputo en la frontera y en la niebla, los aceleradores neuronales se presentan como una gran opción ante la presencia de unidades de procesamiento (CPU/GPU) muy limitadas, poco accesibles para uso exclusivo e intensivo dado que están muy comprometidas en la operación básica del dispositivo y las otras aplicaciones del sistema y del usuario. Por ello, la sección 2 puede indicar una posible ruta de conversión para que el modelo CNN sea habilitado para ser ejecutado dentro de un acelerador neuronal compatible con la plataforma en cuestión.

**Sección 3. Sistemas operativos.** De acuerdo con la plataforma del dispositivo será la ruta de conversión final que genere el archivo final que será embebido en la aplicación móvil de frontera o del nodo de cómputo en la niebla para soportar todas las configuraciones y optimizaciones, y ejecutar de manera eficiente y acelerar por hardware la ejecución del modelo CNN elegido.

**Sección 4. Dispositivos/nodos de prueba.** Para realizar las pruebas de rendimiento del modelo CNN elegido, al final de la ruta de conversión es posible ubicar alguno de los dispositivos y nodos de cómputo de la frontera/niebla específicos para este trabajo (iPad con iOS, NUC con Linux, RPi con Raspian), con excepción de algunas pruebas realizadas en dispositivos móviles tipo Android debido a que no fue posible contar con dispositivos en esta plataforma que contaran con aceleradores neuronales, que por cierto, en los últimos meses han comenzado a surgir, como es el caso del destacado modelo del acelerador neural del chip Kirin 970/980 presente en los teléfonos inteligentes Huawei.

### 4.3.2. Compatibilidad entre capas.

Se llevó a cabo una investigación documental entre los principales fabricantes de dispositivos y de frameworks DL, cuya recopilación aparece documentada en la Tabla 4.2 que reporta información sobre aquellos tipos de capas que serán soportados por cada framework, de dónde puede deducirse cuáles son compatibles con otros *frameworks* DL, ya que a pesar de herramientas como ONNX y el soporte para conversiones entre *frameworks*, la idea es buscar diseñar modelos CNN de manera que la migración entre plataformas sea posible. A nivel de diseño con la información provista por la Tabla 4.2 es posible consultar el nivel de impacto en la compatibilidad que puede tener el hecho de incorporar en la arquitectura del modelo CNN un tipo de capa específico, si por ejemplo, los tipos de capas son soportados por todas las herramientas y frameworks DL, entonces, en teoría el modelo CNN en cuestión tendrá un alto grado de compatibilidad y portabilidad.

Para esta tabla se consideraron siete importantes *frameworks* DL, que fueron con los que más se interactuó durante el desarrollo de las pruebas aquí presentadas. La información se obtuvo de la página web oficial de cada uno. Es importante mencionar que la tabla marca los *frameworks* que cuentan con esas funciones como capas, en algunas ocasiones (sobre todo en el caso de TensorFlow) no están marcadas ciertas capas, pero no significa que esa función no esté soportada, sino que en lugar de estar presente como capa está presente como operación. Para simplificar la tabla y evitar complejos detalles como cuáles son operaciones en lugar de capas y qué implicaciones hay sobre ello, sólo se consideraron las funciones que sí aparecen como capas, y que por lo tanto tendrán mayor facilidad de conversión.

Tabla 4.2. Compatibilidad de capas entre *frameworks*.

| | Tensorflow | TFLite | Keras | OpenVino - TF | OpenVino - Caffe | OpenVino - MXNet | OpenVino - ONNX | NCSDK | Caffe | CoreML |
|---|---|---|---|---|---|---|---|---|---|---|
| AvgPooling | X | X | X | X | X | X | X | X | X | X |
| Batch Normalization | X | X | X | X | X | X | X | X | X | X |
| Convolution | X | X | X | X | X | X | X | X | X | X |
| ConvTranspose | X | X | X | X | X | X | X | X | X | X |
| Dense | X | X | X | X | X | X | X | X | X | X |
| MaxPooling2D | X | X | X | X | X | X | X | X | X | X |
| Eltwise (mul, add,add_n,sub) | | X | X | X | X | X | X | X | X | X |
| Reshape | | X | X | X | X | X | X | X | X | X |
| Slice (split, cropping) | | X | X | X | X | X | X | X | X | X |
| Softmax | | X | X | X | X | X | X | X | X | X |
| Concatenation | | X | X | X | X | X | X | | X | X |
| Flatten | X | | X | | X | X | X | X | X | X |
| LocalResponse Normalization | | X | | X | X | X | X | X | X | X |
| Relu | | X | X | X | X | X | X | X | X | |
| Pow (square, sqrt, rsqrt, neg) | | X | | X | X | X | X | X | X | |
| Permute | | X | X | X | X | X | X | | | X |
| Dropout | X | | X | | X | X | X | | X | |

### 4.3.3. Validación de TinyYOLO.

Para validar la selección del modelo CNN para detección de objetos/personas, se llevaron a cabo conversiones y pruebas de ejecución de diferentes versiones de TinyYOLO por medio de los nodos *Edge/Fog*, partiendo de proyectos base de código abierto publicados por la comunidad de desarrolladores para cada una de las plataformas de prueba. Las versiones evaluadas y sus respectivos resultados son los siguientes:

- **TinyYOLO Legacy**. A pesar de poder convertir este modelo CNN al formato `.mlmodel` para CoreML de iOS para el nodo *Edge*, no se lograron realizar las inferencias debido a diferencias en el formato de salida entre el modelo convertido por CoreMLTools y la configuración de salida esperada por el proyecto base de Xcode.
- **TinyYOLO v1**. No fue posible convertir al formato CoreML de iOS en el nodo *Edge* debido a un problema de incompatibilidad de CoreMLTools con la capa tipo *connected*.
- **TinyYOLO v2**. Esta versión presenta incompatibilidad para conversión a *frameworks DL* fuera de DarkNet debido a una capa tipo *region*, que establece las regiones, dentro de cada celda de la cuadrícula (§3.4), que es donde el algoritmo buscará algún objeto. Afortunadamente, para este caso se logró encontrar una adaptación de esa capa fuera de DarkNet (Duangenquan, 2017), con ello fue posible ejecutar este modelo CNN en cada plataforma de prueba.

De esta manera, por motivos de compatibilidad e interoperabilidad la versión seleccionada fue TinyYOLOv2, la cual está entrenada con la base de datos PASCAL VOC que cuenta con 20 categorías de clasificación, entre ellas la de personas. En la Figura 4.5 se muestran las distintas conversiones y *frameworks* DL que fueron utilizados para ejecutar con éxito el modelo CNN TinyYOLOv2, tanto para el cómputo en la frontera con iOS, como para el cómputo en la niebla con Ubuntu 16.04 y Raspbian Stretch, en cuyos casos se hicieron pruebas con y sin aceleradores neuronales, siendo necesario utilizar *frameworks* diferentes para cada caso. Las respectivas implementaciones se definen con mayor detalle en las secciones 4.4 y 4.5.

Cabe mencionar, que en la Figura 4.5 también se muestran las etapas en las que tendrán lugar las técnicas de adaptación que se presentarán en los capítulos V y VI.

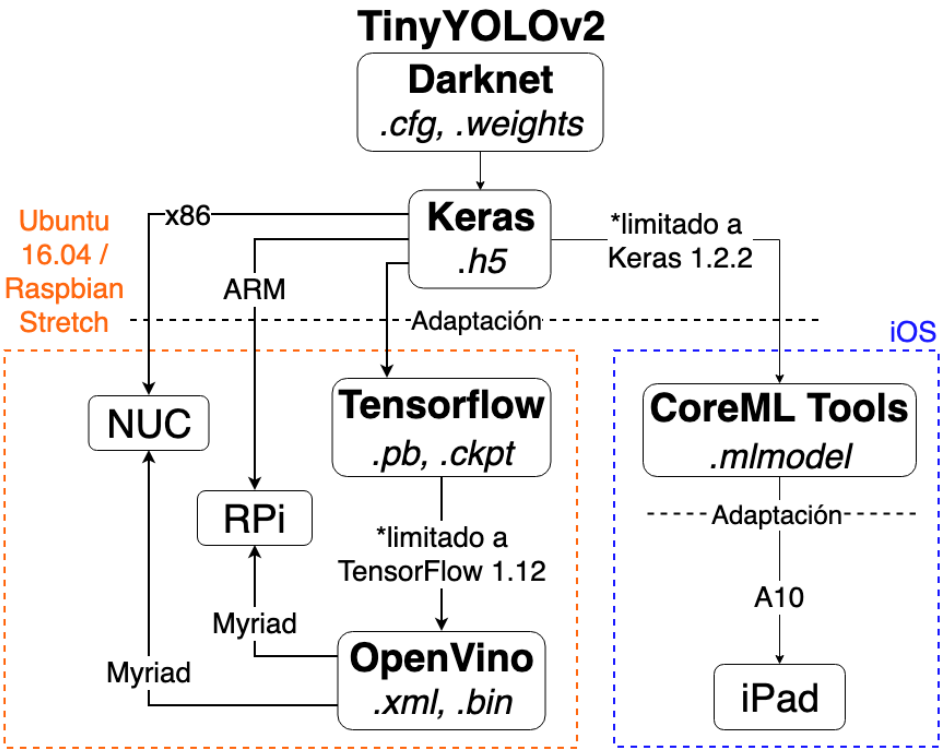


Figura 4.5. Conversiones de modelo TinyYOLOv2 para ejecución en iPAD, RPi y NUC.

## 4.4. Implementación de TinyYOLO con Cómputo en la Frontera.

Dentro de la segunda validación de la hipótesis (§1.5.6), la primera parte de la experimentación corresponde al nodo *Edge*, basado en la tableta iPad, en la cual se implementó una aplicación móvil en iOS capaz de ejecutar modelos CNN, para lo que fue necesario usar las versiones más recientes de CPU (A10+), iOS (ver. 12+), CoreML (ver. 2), Xcode (ver. 10.1) y macOS (Mojave ver. 10.14). Parte del presente desarrollo fue publicado en Pacheco, Cano, Flores, Trujillo, & Márquez (2018).

Partiendo de la implementación del proyecto base (Hollance, 2017) se lleva a cabo la conversión del modelo TinyYOLOv2 desde su *framework* original (DarkNet) hacia Keras, lo cual se realiza a través de la herramienta YAD2K (Allanzelener, 2016). Una vez en Keras se utilizan las librerías de CoreMLTools para convertir el modelo de formato `.h5` (formato de

Keras) a `.mlmodel` (formato de CoreML), luego el modelo CNN se incrusta y carga en la aplicación móvil de iOS, interactuando con dicho modelo a través del API CoreML. Una impresión de pantalla del iPad durante la ejecución de TinyYOLO se muestra en la Figura 4.6, donde el modelo CNN está detectando a una persona, una silla y dos monitores, colocando un cuadro delimitador sobre ellos, así como una etiqueta con el nombre del objeto detectado y un número que representa la probabilidad con la que reconoció a cada objeto.

Dentro del proyecto base (Hollance, 2017) se proporciona la medición de la métrica FPS, debido a que al inicio la inferencia de modelos CNN es más lenta (como se mencionó en la sección 4.1), para esta métrica de rendimiento se tomó el promedio de los primeros 200 cuadros evaluados. Por cuestiones comparativas, además de TinyYOLOv2 también se implementó YOLOv2 (Shen, 2017) y YOLOv3 (Ma, 2017). En la Tabla 4.3 se muestran los resultados de las mediciones de rendimiento expresadas en FPS, donde aparece el valor mínimo (sólo en las primeras inferencias), el máximo y el promedio de los 200 cuadros. Además, se muestra el tamaño (MB) y la precisión (mAP) (§3.3) de cada uno con respecto a la base de datos COCO, considerando IOU = 0.5 (§3.4). También se muestra el tamaño de la imagen de entrada para cada modelo CNN, ya que de eso también depende el desempeño del mismo.

Tabla 4.3. Resultados de YOLOv3, YOLOv2 y TinyYOLOv2 en iOS.

| Modelo CNN | Imagen de entrada | mAP | Tamaño (MB) | Rendimiento (FPS) |
|---|---|---|---|---|
| YOLOv3 | 416 x 416 | 55.3 | 84 | 0.88-2.64 (2.46 avg.) |
| YOLOv2 | 608 x 608 | 48.1 | 204 | 0.32-0.90 (0.80 avg.) |
| TinyYOLOv2 | 416 x 416 | 23.7 | 63.6 | 1.14-6.31 (5.90 avg.) |

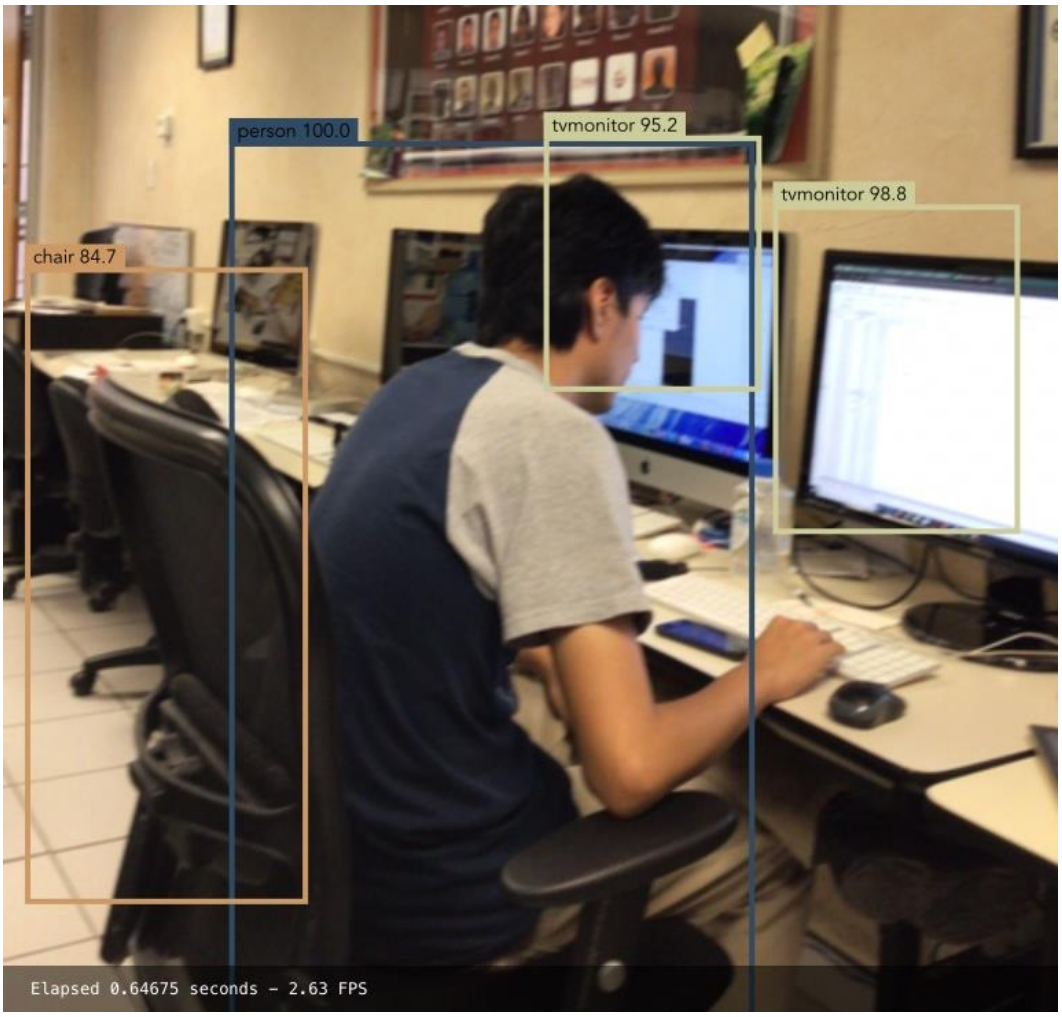


Figura 4.6. Impresión de pantalla del sistema de detección de objetos TinyYOLO en iOS.

Como lo demuestran los resultados de rendimiento obtenidos (Tabla 4.3), la versión YOLOv2 con un tamaño 3.2 veces superior a TinyYOLOv2 tiene un desempeño muy pobre (no alcanza a superar el requerimiento 1.4.3), por lo que resulta impráctico para el cómputo en la frontera y el escenario de prueba (basada en aplicación móvil procesando *video-streaming*). YOLOv3 tiene menos del doble de cantidad de parámetros que YOLOv2, con lo cual alcanza una mejor velocidad, además de contar con una mejor precisión debido a su moderna arquitectura. Por otro lado, TinyYOLOv2 con respecto a YOLOv3 maneja el mismo tamaño de entrada, tiene menos parámetros y una velocidad de procesamiento de más del doble pero también una precisión de menos de la mitad.

A pesar de que la precisión y fluidez de operación de TinyYOLO en esta aplicación móvil de iOS satisface adecuadamente la detección de personas para el control de iluminación, resulta impráctico utilizar la cámara del dispositivo iPad para monitorear la presencia de personas dentro del aula. Este problema se aborda y resuelve con la implementación del modelo CNN para cómputo en la niebla, mismo que se presenta en la siguiente sección.

## 4.5. Implementación de TinyYOLO con Cómputo en la Niebla.

La segunda validación de la hipótesis (§1.5.6) concluye con el nodo *Fog* basado en Intel NUC y RPi 3 asistidos por los dispositivos Intel Movidius Myriad 2 (NCS) y Movidius Myriad X (NCS 2) (Intel Corp., 2019d), estos aceleradores neuronales denominados NPU (*Neural Processing Unit*) (Chatterjee, 2018; WikiChip, 2019), se muestran en la Figura 4.7. Inicialmente se utilizaron los *frameworks* de Caffe (BVLC, 2017) y NCSDK (Intel Corp., 2019b) para las implementaciones con NPU (Duangenquan, 2017); y TensorFlow (Google Brain, 2019), para las implementaciones sin NPU (Simonelli, 2017), pero debido a la falta de soporte de NSCDK para usar el NPU Myriad X (Movidius 2) se optó por utilizar OpenVino (Intel Corp., 2019c) según se indicó en la Figura 4.5. La restricción más importante para lograr que la NUC fuera capaz de ejecutar un modelo DL en el hardware de aceleración neuronal (Intel Movidius) fue el hecho de requerir de manera obligada el uso del sistema Ubuntu 16.04 LTS y la versión de Python 2.7. Se hicieron evaluaciones también con la versión LTS más reciente (Ubuntu 18.04) y con Python 3, pero no fueron compatibles. Por otro lado, la Raspberry Pi 3, dado que ofrece servicios de control en la red local para dispositivos *Edge* puede considerarse un micro-servidor *Fog* muy primitivo, con el cual también fue posible hacer uso de los aceleradores NPU usando la versión de Raspbian Stretch.

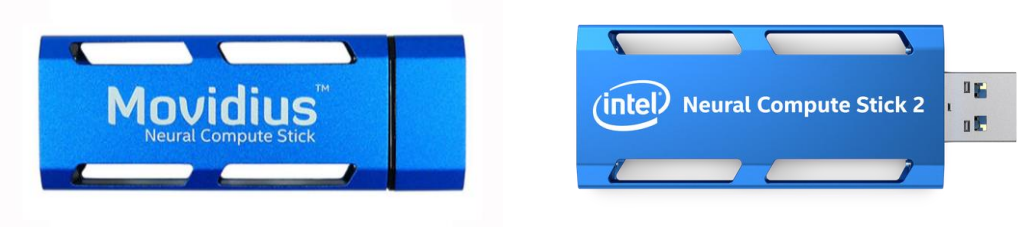


Figura 4.7. Aceleradores Neuronales (NPU) Intel Movidius NCS (Myriad 2) y NCS 2 (Myriad X).

Las implementaciones sin acelerador NPU se llevaron a cabo de manera directa con Keras, mientras que para las pruebas con NPU fue necesario seguir el procedimiento de conversión de Keras a TensorFlow (Abdi, 2017) y después de TensorFlow a OpenVino (Intel Corp., 2019a). El *framework* OpenVino soporta modelos de Caffe, TensorFlow, MXNet y ONNX. Para las pruebas realizadas se tomaron como base los archivos de TensorFlow: `.pb` (donde se encuentra guardada la arquitectura del modelo) y `.ckpt` (donde están guardados los pesos de la red CNN). Una vez generados los archivos `.xml` (arquitectura) y `.bin` (pesos) con las bibliotecas de OpenVino, se implementó el modelo TinyYOLOv2 con base en el proyecto de Hyodo (2018).

En la Tabla 4.4 se muestran los resultados obtenidos de cada prueba llevada a cabo con la versión TinyYOLOv2, tanto con cómputo en la niebla como en la frontera, donde se resaltan las siguientes observaciones:

- En el caso de la RPi 3 sin NPU, es decir, usando la combinación de unidades internas de CPU/GPU, su rendimiento (FPS) resulta insuficiente para pensar en una aplicación de control de iluminación eficiente (requerimiento 1.4.3). Sin embargo, al momento de incorporar un acelerador NPU M2, el nodo basado en RPi 3 justo logra satisfacer el requerimiento de rendimiento (§1.4.3). Por otro lado, los resultados con un NPU MX son mejores que los del M2, pero inferiores a los de dos NPU M2.

- Para el nodo basado en NUC, los resultados sin NPU fueron incluso superiores al uso de un acelerador NPU, que resultó como se esperaba ya que el procesador Core i5 tiene mayor capacidad de procesamiento que los procesadores ARM de la RPi 3. Al usar dos unidades NPU externas, se obtiene la mejor marca de rendimiento, superando ampliamente el rendimiento del procesador Core i5 de la NUC (quedando completamente dedicado dicho CPU para realizar las tareas de control y la red).

- A pesar de ser un dispositivo móvil, el iPad con su NPU interno obtuvo mejores resultados que la NUC con 1 NPU y con CPU, demostrando gran capacidad para realizar

cómputo en la frontera basados en modelos CNN. Sin embargo, resulta impráctico para satisfacer el requerimiento de automatización del sistema de iluminación (§1.4.2).

Tabla 4.4. Resultados de velocidad de procesamiento(FPS) aplicando TinyYOLO en RPi, NUC y iPad.

| Tiny YOLO v2 - FPS | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Raspberry Pi 3 | | | | Intel NUC Core i5 | | | | iPad A10 NPU interno |
| CPU | 1 NPU M2 | 2 NPU M2 | 1 NPU MX | CPU | 1 NPU M2 | 2 NPU M2 | 1 NPU MX | |
| 0.5 | 2.5 | 5.8 | 3.2 | 5.5 | 4.7 | 12.4 | 10.2 | 5.9 |

M2 = Myriad 2 (Movidius 1), MX = Myriad X (Movidius 2)

## 4.6. Prototipo de Control de Iluminación con TinyYOLO en la Niebla.

De acuerdo con el requerimiento 1.4.2, una vez en ejecución las pruebas en la niebla, se implementó sobre cada plataforma un prototipo usando focos LED de color, atenuables e inalámbricos modelo Philips Hue Starter Kit E26, conectados a un hub Zigbee/WiFi. Se implementó un micro-servicio en los nodos Fog que controla inalámbricamente la intensidad y color de cada foco LED. Se ubicó una videocámara fija de bajo costo Logitech C120 para cubrir la sección del aula (campo de visión horizontal de 5.5 metros y distancia entre pared y cámara de 4.7 metros) correspondiente a los pizarrones con el objetivo de iluminar cada zona según la posición de las personas ubicadas frente al pizarrón. Al ejecutar el modelo TinyYOLOv2, con el área de detección establecida, fue posible determinar la posición aproximada de la persona ubicada enfrente del pizarrón y de esta forma controlar de manera apropiada la activación de cada uno de los focos LED, logrando así automatizar y reducir el consumo de energía por concepto de iluminación, mejorando además la sensación de confort al adaptar la iluminación en las zonas de operación y al momento de terminar la sesión apagando las luces de forma automática. Los detalles de este prototipo se publicaron en Pacheco, Flores, Cano, & Tena (2018). En la Figura 4.8 se muestra una impresión de pantalla donde se observa la detección de

una persona y la activación del foco correspondiente a esa zona, además de las divisiones de cada zona indicadas por las líneas verticales en color rojo, las cuales ocupan un espacio horizontal de 1,38 metros cada una.


Figura 4.8. Impresión de pantalla del sistema de detección de objetos TinyYOLO en Ubuntu.

# V. APLICACIÓN DE CUANTIZACIÓN Y TRANSFERENCIA DE APRENDIZAJE AL MODELO CNN

Para la implementación de las primeras técnicas de adaptación basadas en cuantización y transferencia del conocimiento (requerimiento 1.4.6) y como tercera validación de la hipótesis (§1.5.7), la experimentación se dividió en dos partes: adaptación con dispositivos iOS y adaptación con Keras. Ampliando el diagrama de la Figura 4.1, a continuación se presenta el proceso de adaptación del modelo CNN en la Figura 5.1. En este capítulo se abordan las técnicas de adaptación provistas por la plataforma de iOS, que son cuantización y transferencia de aprendizaje (*transfer learning*, TL). Luego se presenta la experimentación implementada basada en Keras, tanto para las pruebas de cuantización como la implementación de TL para crear un nuevo modelo DL con sólo la clase para personas. La última etapa de adaptación basada en poda (*pruning*), será presentada en el capítulo VI.

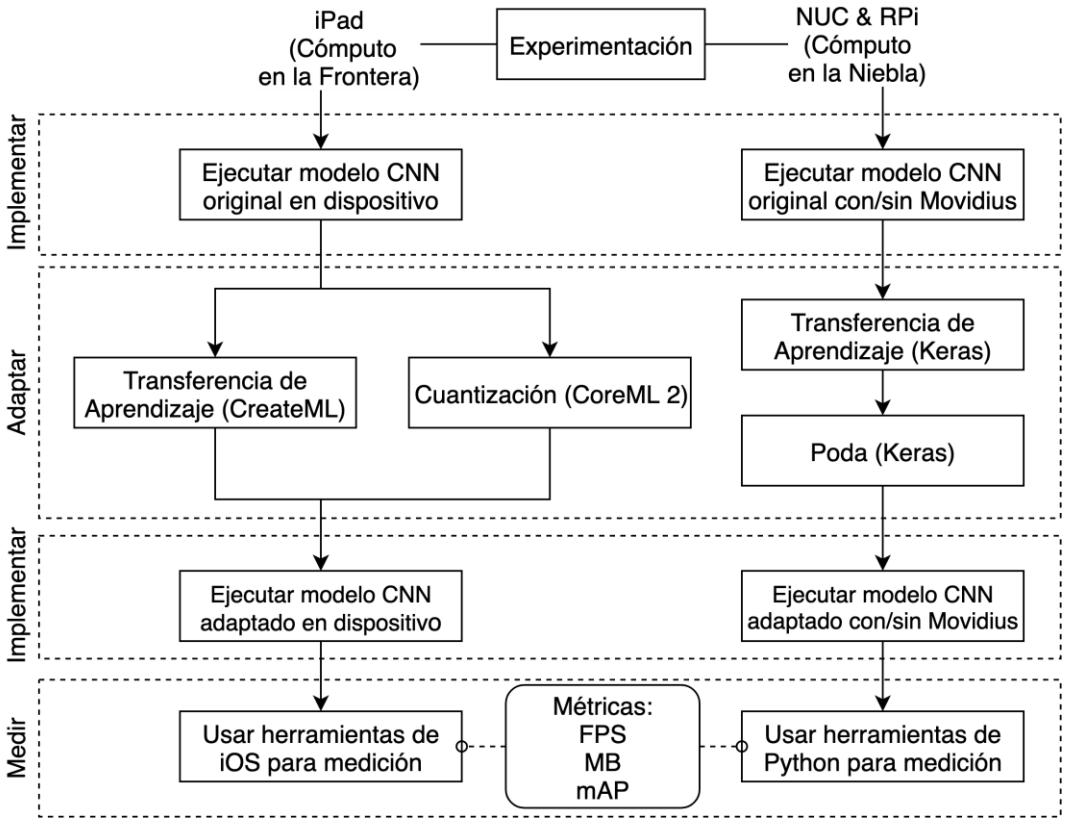


Figura 5.1. Experimentación con modelo CNN adaptado para Cómputo en la Frontera y en la Niebla.

## 5.1. Adaptación CNN usando CoreML de iOS.

En esta sección se presenta la implementación de la técnica de cuantización al modelo CNN de detección de personas TinyYOLOv2, así como la experimentación llevada a cabo basada en TL partiendo de una base de datos propia y la base de datos de INRIA. Estas técnicas fueron aplicadas haciendo uso de las herramientas más recientes de CoreMLTools (Apple Inc., 2019b) y CreateML (Apple Inc., 2019c) provistas por el compilador Xcode 10 para desarrollar aplicaciones de iOS.

### 5.1.1. Cuantización de TinyYOLO con CoreMLTools.

La biblioteca de CoreMLTools para Python permite llevar a cabo cuantización de DNN de manera muy efectiva y simple. Sin embargo, las opciones de configuración que presenta la versión actual son muy limitadas. A partir del modelo TinyYOLOv2 en formato `.mlmodel`, la cuantización del modelo se realizó ejecutando el script de Python 2.7 desarrollado para la conversión desde macOS Mojave 10.14.2, mismo que requirió solamente cinco líneas de código:

```
import coremltools
from coremltools.models.neural_network.quantization_utils import *
model = coremltools.models.MLModel('TinyYOLO.mlmodel')
lin_quant_model = quantize_weights(model, 8, "linear")
lin_quant_model.save('QuantizedTinyYOLO.mlmodel')
```

Este script de Python lee el archivo del modelo `TinyYOLO.mlmodel` para posteriormente aplicar la técnica de cuantización lineal sobre todos los pesos del modelo CNN (§3.5.3), reduciendo así la resolución en bits de la representación de los parámetros del modelo CNN de 32 bits a 8 bits y finalmente, se graba el archivo del modelo CNN cuantizado como `QuantizedTinyYOLO.mlmodel`. Este modelo CNN se carga posteriormente en la aplicación móvil de iOS (versión mejorada del prototipo presentado en la sección 4.4).

Tabla 5.1. Resultados de Cuantización de TinyYOLOv2.

| Modelo CNN | Original (32 bits) | | Cuantizado (8 bits) | |
|---|---|---|---|---|
| | Tamaño (MB) | FPS | Tamaño (MB) | FPS |
| TinyYolov2 | 63 | 5.9 | 16 | 5.9 |

Los resultados obtenidos se muestran en la Tabla 5.1, que establece la comparación entre el modelo CNN original de 32 bits y el nuevo modelo cuantizado a 8 bits. Como se puede observar, después de la cuantización el modelo CNN se redujo a un 25% de su tamaño original, reducción importante considerando que es una aplicación móvil y el dispositivo tiene recursos de almacenamiento muy limitados. Por otro lado, a pesar de la reducción de tamaño del modelo CNN, el rendimiento expresado en FPS se mantiene igual que con el modelo CNN sin cuantizar.

Después de considerar el efecto de la cuantización en términos del tamaño del modelo CNN y su rendimiento (FPS), hace falta considerar el impacto de esta técnica en la precisión del modelo. Para ésto es necesario evaluar el modelo CNN con la base de datos PASCAL VOC, ya que fue la base de entrenamiento de este modelo y la información de precisión está con referencia a ella. Desafortunadamente, no se encontraron proyectos de iOS o herramientas con CoreML para ello debido a la complejidad de evaluación que presentan este tipo de modelos CNN a causa de los *bounding boxes*. Aún así, dentro de la aplicación móvil de iOS la detección de objetos se percibe con la misma funcionalidad, sin un deterioro perceptible en la precisión de la detección.

Otra opción para evaluar la precisión pudo ser convertir el modelo CNN de CoreML a otro *framework* DL en el que si hubiera herramientas para hacer la evaluación, pero al momento de las pruebas no se encontraron herramientas que permitieran hacer dicha conversión. Una opción que a la fecha de redacción se puede explorar es ONNX, como se mostró en la Figura 4.4, ya que existe la posibilidad de convertir un modelo `.mlmodel` a `.onnx` y de ahí pasar a algún otro *framework* DL para hacer la evaluación de la precisión basada en PASCAL VOC.

### 5.1.2. Transferencia del Aprendizaje basada en CreateML.

La herramienta para transferencia de aprendizaje de CreateML de Xcode es tanto o más práctica de emplear que la de cuantización con CoreMLTools, la desventaja es que es aún más limitada. Su capacidad de transferencia de aprendizaje se limita a tareas de clasificación, reconocimiento de texto y regresión, siendo imposible transferir aprendizaje para el caso de detectores de objetos, además de no permitir la selección de la arquitectura CNN base, que es desde donde se transfiere el aprendizaje (§3.5.2). El uso del API CreateML se realiza a través de *playgrounds* de Swift para macOS. En la Figura 5.2 se muestran las tres líneas de código de Swift necesarias en Xcode para ejecutar la interfaz gráfica con la cual se efectúa la transferencia de aprendizaje.

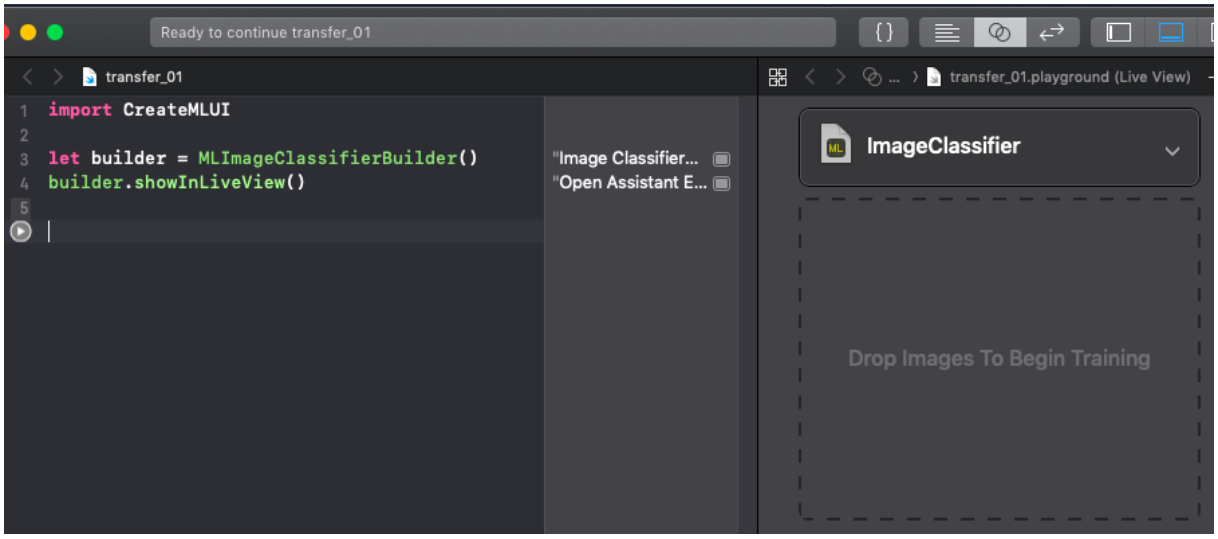


Figura 5.2. *Playground* en Swift y Xcode para TL con CreateML.

Al ejecutar ese código aparecen los recuadros de la derecha de la Fig. 5.2. Las imágenes para el entrenamiento del nuevo modelo CNN se arrastran hacia el recuadro con líneas punteadas, después de cargarse en el entorno de Xcode y realizar el entrenamiento aparece otro recuadro similar para arrastrar las imágenes de evaluación. Un ejemplo de los resultados que arroja se muestra en la Figura 5.3, donde aparecen la cantidad de imágenes empleadas en la evaluación, la cantidad de clases y la precisión de la red, además de detalles como la matriz de

confusión (*confusion matrix*), la cual muestra el número de predicciones correctas e incorrectas con un desglose por clase, dando información de los tipos de errores que se están cometiendo, es decir, información sobre las categorías entre las cuales existe mayor confusión para el clasificador (Sharma, 2017).

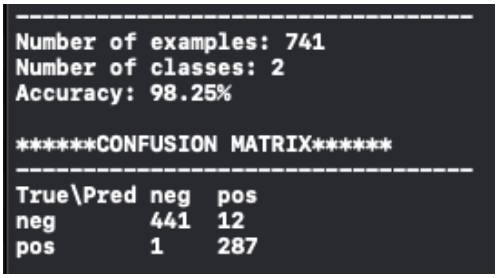


Figura 5.3. Ejemplo de resultados de TL con CreateML.

Para evaluar esta técnica de adaptación se hicieron pruebas con dos bases de datos, la de INRIA (Dalal & Triggs, 2005) y una base de datos propia con imágenes tomadas dentro del laboratorio (BDLab). En ambos casos las bases de datos se dividen en dos, contando con imágenes donde aparecen al menos una persona e imágenes sin personas. La información sobre la cantidad de imágenes de entrenamiento y de evaluación para cada caso, así como los resultados de precisión obtenidos con TL en CreateML se presentan en la Tabla 5.2.

Tabla 5.2. Resultados de TL con CreateML basado en INRIA y BDLab.

| Datos (Imágenes) | | Entrenamiento | |
|---|---|---|---|
| | | INRIA (1832) | BDLab (90) |
| Evaluación | INRIA (741) | 96% | 44% |
| | BDLab (21) | 54% | 100% |

La experimentación se hizo con las cuatro combinaciones mostradas. En el caso de INRIA, los resultados cuando se usaron sus datos tanto para entrenamiento como para evaluación fueron los mejores (debido a la mayor cantidad de datos), esto a pesar del 100% que aparece con el entrenamiento en BDLab. Aunque ese fue el resultado para las imágenes de

evaluación de la misma BDLab, al evaluarse con INRIA la precisión cayó hasta un 44%, esto debido a que BDLab contiene sólo imágenes de personas dentro del laboratorio, por ello es natural que al evaluarse en imágenes con distintos fondos la respuesta de la red sea deficiente. Efecto similar ocurrió con el entrenamiento en INRIA, aunque sólo cayó hasta 54% con BDLab, pero al ser muchas más imágenes de evaluación en INRIA, el 96% de precisión que alcanza es más confiable.

Debido a que este modelo CNN es sólo de clasificación y no de detección, no arroja resultados sobre la posición de las personas detectadas, en video sólo clasificaría cada *frame* como "persona" o "no persona", quedando fuera del requerimiento 1.4.4, por ello no se realizó la evaluación con video para medir su rendimiento (FPS). Aún así, es importante destacar que este modelo CNN adaptado con TL en CreateML tiene un reducido tamaño de 17 KB. La documentación de Apple es escasa al respecto, en general la naturaleza de los modelos resultantes de estas técnicas de adaptación es en gran parte cerrada, en un supuesto, este valor tan reducido se puede deber a que internamente el modelo sólo guarda la última capa (el clasificador) y el resto (la base) se carga aparte por medio de CreateML.

## 5.2. Adaptación CNN con Keras.

Usando el API CoreML de Xcode para iOS se logró implementar TinyYOLO en dispositivos iOS, además de aplicar cuantización para disminuir la cantidad de parámetros, pero sin lograr la reducción de clases ya que la herramienta de CreateML que proporciona para TL está muy limitada y no permite realizar una reducción de clases. Además, la portabilidad de sus modelos CNN hacia otros *frameworks* DL es también muy limitada, eso impide aprovechar estas técnicas de adaptación para reducir modelos CNN y que éstos se implementen en otras plataformas de dispositivos distintas a iOS. Por ello se decidió buscar herramientas de otros *frameworks* DL. Considerando el flujo de trabajo mostrado en la Figura 4.4 y la compatibilidad de capas de la Tabla 4.2, se seleccionó Keras (basado en Python) (Chollet, 2019) con TensorFlow como *backend*, por la posibilidad y facilidad para trasladar modelos CNN entre los

distintos *frameworks* DL de implementación que se emplearon en la experimentación de esta tesis, constatando así el nivel de compatibilidad para las capas internas del modelo CNN TinyYOLO.

Keras es conocido como el *framework* de DL más adecuado para experimentación y de sencilla interacción en comparación con los demás *frameworks* DL, mayormente debido a su modularidad que favorece el desarrollo de prototipos rápido, por ello se le considera un *framework* de alto nivel, funcionando "arriba" de otros *frameworks* llamados *backends*, los cuales son TensorFlow, Theano y CNTK, de ellos utiliza las funciones de bajo nivel con las cuales construye sus módulos. Por otro lado, la desventaja de Keras reside en la eficiencia de los resultados, ya que debido a esa misma modularidad es complicado modificar parámetros a bajo nivel. Debido a eso, es importante mencionar que los resultados y métricas obtenidos en esta experimentación son sólo una referencia, es decir, el mismo método de adaptación que se desarrolló en Keras puede ser implementado en otros *frameworks* DL que permitan trabajar a más bajo nivel, en ese caso se podrían explotar más configuraciones y seguramente se obtendrían mejores resultados. Recordando que el alcance de este desarrollo no busca optimizar los modelos CNN, busca adaptarlos lo suficiente para lograr los requerimientos de funcionalidad planteados (§1.4.6).

### 5.2.1. Cuantización basada en Keras.

Con el objetivo de aprender a manejar la herramienta y delinear el proceso requerido para aplicar la técnica de cuantización y TL, la primera experimentación de cuantización con Keras consistió en cambiar los pesos de un clasificador de MNIST de float32 a float16 y cargarlos al modelo CNN. Al ejecutar el modelo el tiempo de inferencia y la precisión se conservan, pero al consultar el tipo de dato de los pesos nuevamente aparecen como float32. Revisando la literatura, algunos comentan que cuantizar los pesos de una red pre-entrenada no es posible actualmente con Keras 2.2.2, ya que los pesos se conservan en el formato en el que fueron guardados cuando se creó el modelo CNN original (Gupta, 2019; Price, 2018).

Una segunda alternativa fue cambiar el tipo de dato a float16 desde antes del entrenamiento (aunque esto no se ajusta a los requerimientos de la tesis ya que se busca trabajar con modelos CNN pre-entrenados, sería viable si se considera como complemento a TL). En este caso la red efectivamente reduce su tamaño a la mitad (de 3.3MB a 1.6MB) pero la precisión cae drásticamente (de 97% a 9%). Se intentó reentrenar la red, pero no fue posible de realizar en un procesador estándar (Intel Core i5-3210M CPU @ 2.50GHz), para ello se hizo uso de una tarjeta gráfica GPU NVIDIA GeForce RTX 2080 provista por el laboratorio del Dr. Mariano Rivera del Centro de Investigación en Matemáticas A.C. (CIMAT). Al terminar el reentrenamiento se guardó el modelo CNN, pero al cargarlo de nuevo aparecen errores de incompatibilidad debido al tipo de dato float16, sin permitir el uso del modelo.

Como tercera alternativa, se evaluó la cuantización a través de la herramienta de TensorFlow Lite (TFLite). Para ello fue necesario convertir el modelo de Keras a TFLite, debido a que Keras trabaja con *backend* de TensorFlow existen herramientas del propio *framework* para realizar esta conversión. Existieron dificultades con respecto a la versión de TensorFlow (Murthy, 2018; Sikr, 2018), ya que recientemente publicaron la versión 1.13 y trae algunos cambios que tienen que ver con la herramienta de cuantización de TFLite. Fue necesario emplear la versión anterior (1.12), con la cual se realizó con éxito la cuantización. Se buscaron herramientas para convertir de TFLite a Keras, pero no se encontraron, de manera que esta opción de cuantización sólo sería un camino viable cuando se buscan aplicaciones para dispositivos móviles con Android o un acelerador neuronal como Google Coral, que son los dispositivos compatibles con TFLite. Debido a que tales dispositivos quedan fuera de los requerimientos (§1.4.5), no se llevó a cabo la evaluación de precisión o de rendimiento para tal modelo, pero si se considera como una opción complementaria a las técnicas aquí presentadas para el caso de aplicaciones móviles para Android.

### 5.2.2. Transferencia de aprendizaje con reducción de clases usando Keras.

Con las pruebas de TL en iOS (§5.1.2) se observó su potencial de resolver el problema de reducción de clases y con base en los principios explicados anteriormente (§3.5.2) se aplicó TL tomando como base las primeras ocho capas convolucionales de TinyYOLOv2, dejando fuera la capa convolucional del clasificador y agregando una nueva para obtener un modelo CNN que sólo detecta personas llamado TinyYOLOvP (versión Personas). En la Tabla 5.4 se muestra la arquitectura de TinyYOLOv2 y características de cada capa, consultadas mediante el comando `model.summary()` desde el *framework* de Keras.

A diferencia de modelos CNN para clasificación entrenados en ImageNet que cuenta con 1,000 categorías, o modelos CNN para detección de objetos como YOLO entrenados con la base de datos de COCO que cuenta con 80 categorías, la base de datos que se usó en esta ocasión fue PASCAL VOC 2012, esto debido a que el modelo de TinyYOLOv2 está entrenado con esta base de datos y debido a que es de menor tamaño con respecto a COCO, facilitando la experimentación. PASCAL VOC 2012 cuenta con 20 clases y 17,125 imágenes, de las cuales sólo se incluyeron aquellas donde hay personas, quedando con la distribución que muestra la Tabla 5.3.

Tabla 5.3. Base de datos PASCAL VOC 2012, clase Person.

| Clase | Imágenes de Entrenamiento (80% del total) | Imágenes de Validación (20% del total) | # de Objetos dentro del total de imágenes |
|---|---|---|---|
| Persona | 7,666 | 1,917 | 17,401 |

Tabla 5.4. Arquitectura del modelo CNN TinyYOLOv2 (original).

| # | Tipo de capa | Filtros | Tamaño/Stride | Salida | Parámetros |
|---|---|---|---|---|---|
| 1 | **Conv1** | 16 | 3 x 3 | 416 x 416 | 432 |
| 2 | BatchNorm | - | - | - | 64 |
| 3 | LeakyRelu | - | - | - | 0 |
| 4 | MaxPool | - | 2 x 2 / 2 | 208 x 208 | 0 |
| 5 | **Conv2** | 32 | 3 x 3 | 208 x 208 | 4,608 |
| 6 | BatchNorm | - | - | - | 128 |
| 7 | LeakyRelu | - | - | - | 0 |
| 8 | MaxPool | - | 2 x 2 / 2 | 104 x 104 | 0 |
| 9 | **Conv3** | 64 | 3 x 3 | 104 x 104 | 18,432 |
| 10 | BatchNorm | - | - | - | 256 |
| 11 | LeakyRelu | - | - | - | 0 |
| 12 | MaxPool | - | 2 x 2 / 2 | 52 x 52 | 0 |
| 13 | **Conv4** | 128 | 3 x 3 | 52 x 52 | 73,728 |
| 14 | BatchNorm | - | - | - | 512 |
| 15 | LeakyRelu | - | - | - | 0 |
| 16 | MaxPool | - | 2 x 2 / 2 | 26 x 26 | 0 |
| 17 | **Conv5** | 256 | 3 x 3 | 26 x 26 | 294,912 |
| 18 | BatchNorm | - | - | - | 1,024 |
| 19 | LeakyRelu | - | - | - | 0 |
| 20 | MaxPool | - | 2 x 2 / 2 | 13 x 13 | 0 |
| 21 | **Conv6** | 512 | 3 x 3 | 13 x 13 | 1,179,648 |
| 22 | BatchNorm | - | - | - | 2,048 |
| 23 | LeakyRelu | - | - | - | 0 |
| 24 | MaxPool | - | 2 x 2 | 13 x 13 | 0 |
| 25 | **Conv7** | 1,024 | 3 x 3 | 13 x 13 | 4,718,592 |
| 26 | BatchNorm | - | - | - | 4,096 |
| 27 | LeakyRelu | - | - | - | 0 |
| 28 | **Conv8** | 1,024 | 3 x 3 | 13 x 13 | **9,437,184** |
| 29 | BatchNorm | - | - | - | 4,096 |
| 30 | LeakyRelu | - | - | - | 0 |
| 31 | Classifier | 125 | 3 x 3 | 13 x 13 | 128,125 |
| | | | | **Parámetros totales** | **15,867,885** |

Antes de iniciar el procedimiento de TL, y como para cualquier caso de entrenamiento CNN, se requiere hacer una etapa de pre-procesamiento de los datos de entrada de la red, tanto de las imágenes como de las anotaciones (el archivo donde viene la información de los objetos detectados en una imagen, que son la clase y las coordenadas de los cuadros delimitadores de cada objeto), sobre todo éstas últimas debido a que se cargan desde archivos `.xml`. Este pre-procesamiento, así como la función de pérdida a utilizar para el entrenamiento, sus parámetros e hiper-parámetros y la evaluación del modelo CNN, son bloques funcionales donde en cada uno recae una responsabilidad específica sobre el desempeño final de la red neuronal. Sin embargo, cada uno por su cuenta es todo un tema de investigación. En este trabajo, debido al enfoque en la aplicación de técnicas de adaptación se decidió buscar y tomar dichos bloques de la comunidad de desarrolladores de Keras, encontrándose un proyecto (Experiencor, 2017), referido a partir de ese momento como proyecto base o PB, que implementa TL para siete arquitecturas, entre ellas TinyYOLOv2. En esencia este proyecto resuelve el problema de TL, pero fue necesario realizar algunos cambios.

A continuación, con base en la Figura 5.4 se explica el procedimiento llevado a cabo junto con esos cambios mencionados. El procedimiento se efectúa con Keras 2.2.2 y Python 3.6.8.

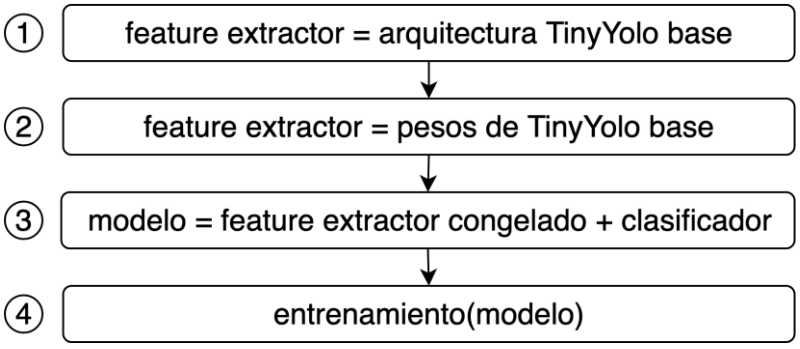


Figura 5.4. Procedimiento de TL.

- **Paso 1.** La variable *feature extractor* representa el modelo base, al cual se le asignan las ocho capas convolucionales mostradas en la Tabla 5.4, así como sus respectivas capas de *BatchNormalization, LeakyRelu* y *MaxPooling*.

- **Paso 2.** Después de asignar las capas al modelo CNN, es posible cargar los pesos de TinyYOLOv2, exceptuando por supuesto los pesos de la última capa.

- **Paso 3.** Se define un nuevo modelo CNN, el cual incluye la base anterior, pero con todas las capas congeladas (§3.5.2) agregando otra capa convolucional como clasificador. El cálculo de la cantidad de filtros de esa capa se obtiene con la ecuación 5.1.

$$B * 5 + C \tag{5.1}$$

  Donde *B* es la cantidad de cuadros delimitadores propuestos para cada celda (§3.4), que en este caso fueron cinco (parte de la configuración provista por el PB, más cuadros implican mayor precisión de localización, pero también mayor procesamiento). Sus valores representan la probabilidad de que exista un objeto en ese cuadro, y al mismo tiempo para cada uno de ellos existen 5 valores de predicciones: *x, y, w, h* y *confidence*. Por último, la *C* indica el número de clases a entrenar, en este caso 1. De esta manera la cantidad de filtros para esta última capa son 30, que son las salidas de la red que aportan la información de cuántos y qué objetos se detectaron, además de las coordenadas de los correspondientes cuadros delimitadores.

- **Paso 4.** Para el entrenamiento el tamaño de las imágenes de entrada fue de 416x416, con 63 épocas de entrenamiento de acuerdo a los parámetros establecidos en el PB. Este procesamiento se efectuó en 17 horas con una GPU NVIDIA GeForce RTX 2080.

Con respecto a la implementación del PB se hicieron principalmente dos cambios. El primero se debe a que en dicho proyecto no se realiza TL congelando las capas base, paso importante ya que cuando se agrega el clasificador, este cuenta con pesos inicializados aleatoriamente. Eso implica que al iniciar el entrenamiento los gradientes de la optimización tiendan a ser grandes y si las capas de la base están descongeladas, los pesos sufren severos cambios que afectan a la precisión de la red neuronal, la cual se podría recuperar, pero tomaría muchas más épocas de entrenamiento, desaprovechando el aprendizaje que tenía la red al principio. Se congelaron las capas base para 10 épocas y en las subsecuentes se intentó realizar

*fine-tuning* (§3.5.2), descongelando la octava capa convolucional, pero el valor de la función de pérdida mejoraba muy lentamente. Por ello, se decidió entrenar el resto de épocas con el modelo completamente descongelado, los pesos de la última capa ya no eran aleatorios y el gradiente ya no sería tan grande como en un principio. El segundo cambio consistió en diseñar la arquitectura como un sólo bloque, que el proyecto PB la define en dos partes, anidando la base y el clasificador. Eso consiste en convertir la base en un modelo CNN y luego considerarla como capa, agregar el clasificador y guardar un nuevo modelo. El problema surge cuando se trabaja con capas congeladas, actualmente en Keras existe un *bug* que impide trabajar con modelos anidados que contienen ciertas capas congeladas (Ciurana, 2019). Por ese motivo, se hicieron los cambios necesarios en código para adaptar el proyecto a una arquitectura CNN de un sólo bloque que contiene sólo capas, no un bloque de capas más el clasificador.

Como conclusión, con base en la evaluación de mAP del proyecto PB original, la red neuronal adaptada alcanzó una precisión de 49.67%, aceptable en comparación con el 57.1% de TinyYOLO, considerando que este entrenamiento sólo tuvo 63 épocas (vs 340 del original) y no se implementaron estrategias como aumentación de datos (§3.1) o multi-escala (§3.4) debido a que el tiempo de entrenamiento habría sido mayor.

Gracias a este procedimiento implementado en Keras, fue posible resolver el problema de reducción de clases para el detector de objetos elegido (TinyYOLOv2), logrando construir un modelo CNN de detección de personas (TinyYOLOvP), pero faltando por resolver el problema de aceleración y compresión del modelo CNN, mismo que se describe en el siguiente capítulo.

# VI. APLICACIÓN DE PODA CNN-PSL AL DETECTOR DE PERSONAS

Al aplicar la técnica de transferencia de aprendizaje (TL) al modelo TinyYOLOv2, el modelo CNN resultante (TinyYOLOvP) se redujo de 20 categorías a sólo una (§5.2.2). Sin embargo, en su estructura interna, las ocho capas convolucionales (base) tienen la misma cantidad de filtros y parámetros que el modelo original de TinyYOLOv2. En un modelo CNN, después del entrenamiento comúnmente se presenta una anomalía conocida como redundancia de parámetros (*parameter redundancy*) (§3.5.1), lo cual significa que existen pesos o incluso filtros completos que no están aportando algo significativo al resultado o que aportan mínimamente. Después de un proceso de TL es fácil que este efecto se intensifique debido a que los pesos de la base están configurados para detectar características de clases que ya no están presentes. A pesar de que durante la aplicación de la técnica TL (§5.2.2) se descongelaron y modificaron todos los pesos, la hipótesis es que aún existe un gran porcentaje de redundancia, en cuyo caso una de las recomendaciones que se encontró en la literatura (§3.5) es aplicar la técnica de poda de redes neuronales (*network pruning*).

En este trabajo se propuso un nuevo enfoque para la implementación de tal técnica, llamado CNN-PSL (*CNN Pruning by Switch Layer*), con un modelo teórico y pruebas experimentales correspondientes al trabajo en desarrollo de Rivera, Flores, & Pacheco (2019). Esta técnica está inspirada en los trabajos con enfoque en la poda dependiente de datos explicados en la sección 3.5.1, esto debido a que los datos de entrenamiento son reducidos y eso permite que el enfoque de dependencia de datos sea viable y así aspirar a mejores resultados con respecto a aplicar un enfoque no dependiente de datos.

La estrategia de esta técnica consiste en podar la red neuronal incorporando una capa *switch*, la cual se implementa como una capa convolucional y es la encargada de detectar y eliminar los filtros de la capa convolucional anterior que menos aportan al resultado, removiendo además los canales correspondientes en la capa posterior a la capa *switch*. Al reducir la cantidad de parámetros, la red neuronal se comprime, con lo cual se ahorra memoria y

sobretodo, dependiendo de la cantidad de filtros convolucionales que se eliminen, también es posible obtener un significativo impacto en la reducción de complejidad computacional y tiempo de procesamiento (§3.2).

A continuación, se describen con detalle las características de esta capa *switch*, así como el proceso de implementación que consta de dos etapas: la adición de la capa *switch* al modelo TinyYOLOvP y la poda de los filtros convolucionales seleccionados. Al final se presentan los resultados de la aplicación de esta técnica de adaptación CNN-PSL al detector de personas.

## 6.1. Descripción de la técnica propuesta CNN-PSL.

De manera muy general, la salida de una capa en una red neuronal se puede describir según la ecuación 6.1, donde $y_i$ es la salida de la capa $i$, $\phi_i$ su función de activación, $W_i$ la matriz de pesos, $b_i$ su respectivo bias y $y_{i-1}$ la salida de la capa anterior.

$$y_i = \phi_i(W_i y_{i-1} + b_i) \tag{6.1}$$

Por otro lado, la salida de una capa convolucional se puede visualizar como se muestra en la Figura 6.1, donde M son las dimensiones del mapa de características de $y_i$ y F el número de filtros de $y_i$, representación análoga a la de un filtro, donde M sería la dimensión del *kernel* y F la cantidad de canales, pero en ese caso la poda sería de canales, y lo que se busca con esta técnica es podar filtros para ahorrar la mayor cantidad de parámetros de una manera práctica.

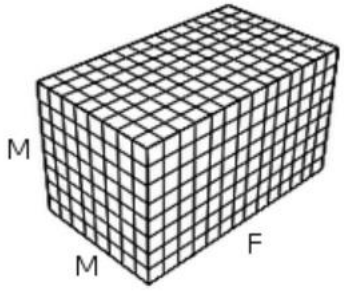


Figura 6.1. Visualización de la salida de una capa convolucional.

La propuesta consiste en obtener un conjunto de valores ($S$) que representen la importancia de cada filtro de la capa convolucional que se desea podar ($y_i$) y que mediante una multiplicación elemento a elemento o producto de Hadamard (Million, 2007) se descarten los filtros menos importantes. Este producto podría visualizarse como muestra la Figura 6.2.

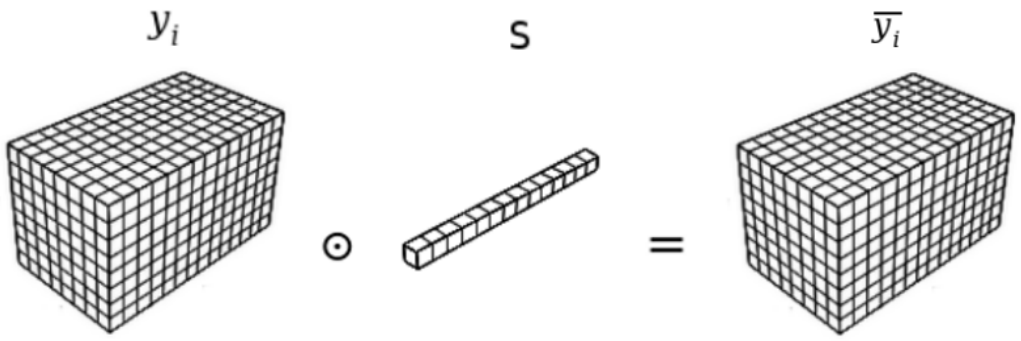


Figura 6.2. Visualización simbólica del producto de Hadamard.

Con respecto a la ecuación 6.1 este producto también se puede representar con las ecuaciones 6.2, 6.3 y 6.4.

$$\overline{y_i} = y_i \odot S \tag{6.2}$$

$$y_{i+1} = \phi_{i+1}(W_{i+1}\,\overline{y_i} + b_{i+1}) \tag{6.3}$$

equivalente a

$$y_{i+1} = \phi_{i+1}(W_{i+1}\,(S \odot \phi_i(W_i\,y_{i-1} + b_i) + b_{i+1}) \tag{6.4}$$

El cual se presenta como un problema de optimización, buscando los valores de $W's, b's$ y $S's$ que minimicen la función de costo, lo cual es resuelto a través del proceso de entrenamiento de la red neuronal. Esto se expresa en las ecuaciones 6.5 y 6.6, donde $y_{i+1}$ son los valores o etiquetas estimados por la red y $\hat{y}$ son los valores verdaderos. Esta función de costo se calcula a través de la norma L2, que es la mayormente implementada para el entrenamiento.

$$\left|\left|\, \hat{y} - y_{i+1} \,\right|\right|_2 \tag{6.5}$$

$$\left|\left|\, \hat{y} - \phi_{i+1}(W_{i+1}\,(S \odot \phi_i(W_i\, y_{i-1} + b_i) + b_{i+1}) \,\right|\right|_2 \tag{6.6}$$

En este caso, la solución para $S$ es trivial con $S = 1$, es decir, no usar *switches*, esto debido a que no hay incentivo para apagar $S's$. Por ello se decide agregar una penalización por usar valores altos de $S$ a través de un regularizador, mostrándose en la ecuación 6.7, donde $\lambda$ es el parámetro de regularización para $S$ a través de la norma L1. Esta norma se ajusta mejor al problema debido a que se busca que los valores de $S$ lleguen a cero, y así poder eliminar más filtros.

$$\left|\left|\, \hat{y} - y_{i+1} \,\right|\right|_2 + \lambda\left|\left|\, S \,\right|\right|_1 \tag{6.7}$$

En este punto aún no es posible obtener los valores de $S$ que se buscan, ya que incluso con valores de $S$ muy pequeños, las activaciones de $y_i$ pueden hacerse muy grandes. Un ejemplo se presenta en la ecuación 6.8, donde dividir $S$ entre $k$ (siendo $k$ un valor muy grande que representa una penalización para $S$, esta división se hace sólo para explicar el efecto que tendría la penalización) ejemplifica un caso donde $S$ es muy pequeña, pero por el otro lado en la activación, también aparece $k$, ejemplificando que tanto como pueda reducirse $S$, así puede crecer la activación, compensando automáticamente la penalización de $S$.

$$\left(\frac{S}{k}\right) \odot \phi_i((W_i\, y_{i-1} + b_i) * k) \tag{6.8}$$

Penalizar $S$ por sí solo no ayuda si no se acota $\phi_i$, y para ello se agregaron restricciones de cota superior a $\phi_i$. En este punto, $S$ ya puede obtener los valores deseados para seleccionar cuáles filtros eliminar, el siguiente problema es cómo implementar esto a través del algoritmo de retro-propagación y las bibliotecas de Keras.

Idealmente, se buscaría aplicar el producto de Hadamard como una operación o capa propia del *framework* y con ello fácilmente resolver la ecuación 6.6, pero Keras no cuenta con esa función. La primera alternativa para lograr esta implementación fue crear una capa personalizada con Keras, con parámetros entrenables y que desempeñara la operación de dicho

producto, pero al ser este un *framework* de alto nivel, muchas configuraciones son de difícil acceso y eso provocó la necesidad de buscar otra alternativa. Como se mencionó al principio de esta sección, el *switch* finalmente se implementa mediante una capa convolucional adicionada inmediatamente después de la capa convolucional que se desea podar, con kernel de 1x1, sin bias y con la misma cantidad de filtros de la capa anterior. Al ser una capa convolucional, esta tiene parámetros entrenables que son los que se ajustan según la regularización y los que determinan cuáles filtros son menos importantes para la etapa de inferencia. Estos parámetros se almacenan en un tensor de la forma:

(K, K, C, F)

Donde K son las dimensiones del kernel (en el caso del *switch* K es igual a 1), C es el número de canales y F el número de filtros. Dado que los dos primeros ejes son 1, básicamente la información de cuáles filtros podar se encuentra en los parámetros de (C, F). Para lograr simular la multiplicación elemento a elemento, se requiere que para cada filtro a evaluar su importancia corresponda sólo a un valor del *switch* ($S$) y para ello se requiere que esa matriz (C, F) sea diagonal, es decir, que todos los valores sean nulos menos los de la diagonal de esa matriz, y dicha diagonal representa a $S$ en la Figura 6.2.

Como ejemplo, considérese la ecuación 6.9, donde $A$ es el vector diagonal $S$, $W$ es la matriz de pesos de $y_i$, $f$ su cantidad de filtros y $c$ el número de kernels que contiene cada filtro (o bien, número de canales).

$$A \cdot W = \begin{bmatrix} a_1 w_{11} & a_1 w_{12} & a_1 w_{13} & \cdots & a_1 w_{1c} \\ a_2 w_{21} & a_2 w_{22} & a_2 w_{23} & \cdots & a_2 w_{2c} \\ \vdots & \vdots & \vdots & & \vdots \\ a_f w_{f1} & a_f w_{f2} & a_f w_{f3} & \cdots & a_f w_{fc} \end{bmatrix} \tag{6.9}$$

Para este caso, si $A$ contiene a un elemento $a_j$ con valor de cero o cercano a cero, estará eliminando toda una fila, correspondiente a un filtro completo. De esta manera el objetivo es obtener el tensor de pesos de la capa *switch* como una matriz diagonal con restricción de sus valores a sólo números positivos.

Una vez conocidos los filtros menos importantes, se crea una nueva red CNN con arquitectura reducida, es decir, con las mismas capas, pero con menos filtros en la convolucional que se trabajó, en donde se transfieren los pesos de los filtros útiles. En esta nueva red ya no se coloca la capa *switch*, eso provoca que falte la multiplicación por los pesos que no eran cero para conservar los pesos que se alcanzaron en el entrenamiento. De quedarse así la precisión disminuye drásticamente, por ello se debe compensar la extracción de la capa *switch* multiplicando directamente los pesos de la capa podada por los valores de *S* que no eran cero. Hecho esto, la aplicación de la técnica es iterable para el resto de capas convolucionales.

Partiendo de la columna uno (`num-capa`) y la columna dos (`tipo-capa`) de la Tabla 5.4, se definieron identificadores de variables (incluidas en scripts de Python y figuras) para cada una de las capas del modelo TinyYOLO aplicando la siguiente gramática EBNF (Extended Backus-Naur Form), donde `tecnica` especifica el símbolo terminal `_s` para la capa *switch* y `_p` se reservó para designar las capas donde se aplicó la técnica de poda:

```
1:   <id-capa> ::= <num-capa><tipo-capa><tecnica>*
2:  <num-capa> ::= c<digito>+
3: <tipo-capa> ::= conv<digito>+ | norm | relu | clasif
4:   <tecnica> ::= _s | _p
```

La Figura 6.3 muestra cómo se derivan los modelos CNN durante el proceso de poda, indicando las capas involucradas durante la aplicación de la técnica propuesta (recuadros sombreados). En el extremo izquierdo aparece la sección final de la arquitectura TinyYOLOvP (la cual tiene la misma estructura que TinyYOLOv2 salvo por la cantidad de filtros de la última capa). Posteriormente se agrega la capa *switch* para derivar el modelo TYvP, y al final se obtiene el modelo TinyYOLOvPP (versión Personas Podada) seleccionando los filtros más importantes que definen la capa `c28conv8_p`, una reducción de canales en la capa `c29norm_p` y una sola clase en la capa `c31clasif_p`.

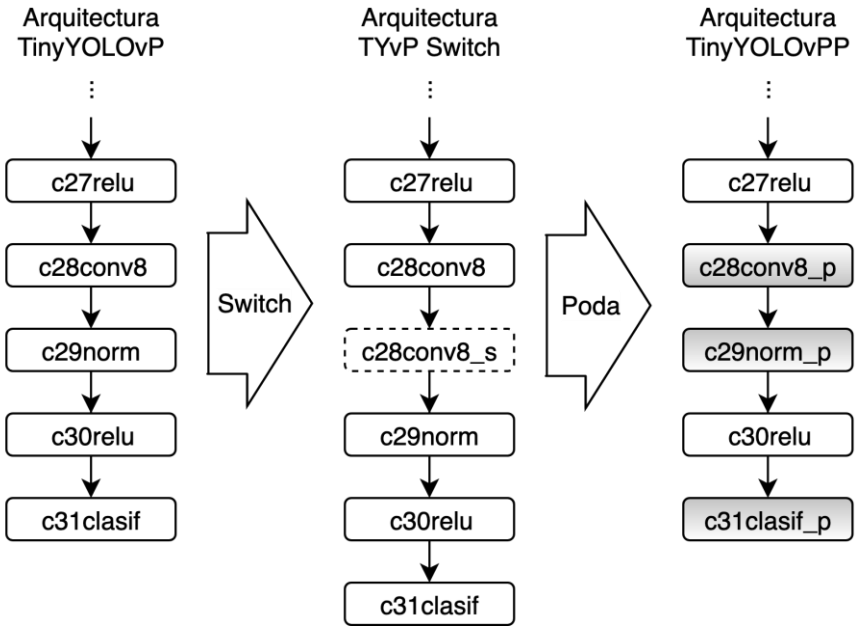


Figura 6.3. Cambios en la arquitectura TinyYOLOvP involucrados en el proceso de poda propuesto.

## 6.2. Adición de la capa *Switch*.

La adición de la capa *switch* a la red TinyYOLOvP requiere de los tres pasos mostrados en la Figura 6.4, mismos que fueron aplicados para podar la octava capa convolucional (`c28conv8`) ya que ésta contiene el 60% de los parámetros totales de la red debido a la gran cantidad de filtros y canales que posee (Tabla 5.4), por ello esta capa es la primera opción para realizar la poda, con la cual se eliminarán los filtros donde se detecte la mayor redundancia.

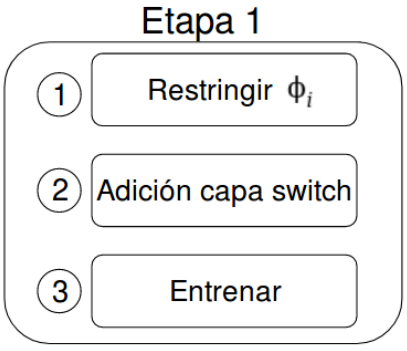


Figura 6.4. Procedimiento de adición de la capa *switch* a la arquitectura TinyYOLOvP (etapa 1).

- **Paso 1.** Para restringir los valores de $\phi_i$ y que no crezcan de manera proporcional a la reducción de $S$, se aplicó la función de Keras `MaxNorm` para limitar la norma de los valores de los pesos $W$ de una capa convolucional, en este caso a un valor máximo de 1 (`max_value` en línea 1). Dicho criterio para restringir los valores de $\phi_i$ se aplicó al momento de definir la octava capa convolucional `c28conv8` (línea 2). Como se indica en la Tabla 5.4 y la Figura 6.3, dicha capa `c28conv8` está ubicada justo después de la capa de activación LeakyReLU (`c27relu`), razón por la cual se recibe como parámetro (capa de entrada) para la función de retorno del constructor `Conv2D()` usando la modalidad funcional del API de Keras (línea 2).

```
1: criterio = keras.constraints.MaxNorm(max_value=1, axis=0)
2: c28conv8 = Conv2D(1024, (3,3),
      strides = (1,1), padding = 'same',
      name = 'conv_8', use_bias = False,
      kernel_constraint = criterio) (c27relu)
```

- **Paso 2.** La adición de la capa *switch* (`c28conv8_s`) se define en la línea 3 que va justo después de la definición de la capa `c28conv8`, seguida posteriormente por las capas `BatchNormalization (c29norm)` y `LeakyRelu (c30relu).` Agregando las dos condiciones que se habían mencionado, la penalización de los pesos aplicando un regularizador con norma L1, cuyo parámetro de regularización es el que se encarga de ajustar la penalización, si es muy grande se eliminarían muchos filtros y la precisión caería drásticamente; para esta experimentación el valor con el que se obtienen los mejores resultados fue $\lambda = 1e - 5$ (§6.4). La otra condición es la de convertir $S$ en una matriz diagonal y limitarla a sólo valores positivos, esto se realiza a través de la implementación de una restricción personalizada llamada `DiagonalNonNegNorm`, además esta restricción también se configuró para que los valores menores a 0.01 se eliminen (quedando como ceros) y con esto facilitar que al momento de seleccionar los filtros a eliminar se escojan aquellos iguales a cero.

```
3: c28conv8_s = Conv2D(1024, (1,1),
      name = 'switch_8', use_bias = False,
      kernel_constraint = DiagonalNonNegNorm(),
      kernel_regularizer = regularizers.l1(1e-5)) (c28conv8)
```

- **Paso 3.** El reentrenamiento de la red TYvP Switch se lleva a cabo congelando todas las capas (código mostrado en las líneas 4-8) excepto `c28conv8_s`, `c28conv8`, `c29norm` y `c31clasif`, estas se dejan descongeladas debido a que se requieren para que la capa *switch* pueda encontrar los valores de importancia de cada filtro. El resto se congelan para evitar mayores pérdidas de precisión, ya que este es el único paso en donde es posible perder precisión. Se realizan 3 épocas de entrenamiento, con los mismos datos y configuraciones que se usaron para TL. En caso de que al finalizar las 3 épocas la precisión haya caído demasiado, es posible hacer más épocas para mejorar la precisión. Si aún así la precisión no mejora, significa que la penalización debe reducirse ya que la cantidad de filtros que se están eliminando es tan alta que la función de pérdida ya no converge, los filtros no son suficientes para transmitir la información de la extracción de características de las capas anteriores.

```
4: for layer in model.layers:
5:         if layer.name.find('8') != -1:
6:               layer.trainable = True
7:         else:
8:               layer.trainable = False
```

Al terminar este entrenamiento la red alcanza una precisión de 46.16% mAP (obtenida con el promedio de cinco evaluaciones), similar a la del modelo TinyYOLOvP (49.67% mAP). En esta instancia la capa *switch* ya contiene la información de cuáles filtros podar y cuáles conservar, en la siguiente etapa se explica la manera de extraer esa información y cómo usarla para realizar la poda.

## 6.3. Poda.

Una vez almacenada en la capa *switch* la información de cuántos y cuáles filtros podar, se lleva a cabo la etapa dos, descrita en la Figura 6.5 y explicada a continuación.

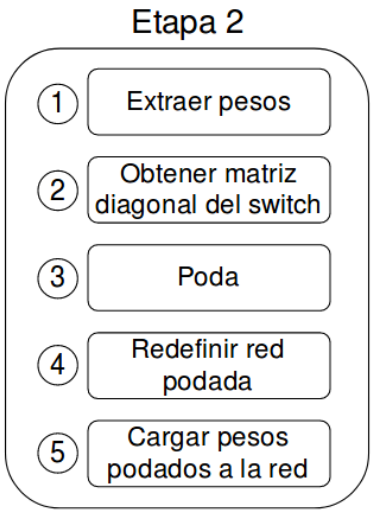


Figura 6.5. Procedimiento de poda de la red CNN TinyYOLOvP (etapa 2).

- **Paso 1.** El primer paso es la extracción de los pesos de la arquitectura TYvP Switch y guardarlos en la variable `W` (línea 9).

```
9: W = model.get_weights()
```

- **Paso 2.** Extraídos los pesos, lo primero que se desea es obtener la matriz diagonal con los valores de $S$ (línea 10), obteniéndose con base en el tensor de pesos de la capa *switch*, cuyo arreglo tiene las siguientes dimensiones o forma (atributo `shape` para arreglos en NumPy):

```
(1, 1, 1024, 1024)
```

  Siendo (1,1) la dimensión del *kernel* y (1024, 1024) los canales y filtros de $S$ representados en la Figura 6.3. Posteriormente, se extrae la diagonal de esa matriz, que posee los valores que determinan la importancia de los filtros, guardándose en la variable `W2_diag` (línea 11).

```
10: pesos = W[36][0, 0, :, :]
11: W2_diag = numpy.diag(pesos)
```

- **Paso 3**. En la arquitectura de TYvP Switch, después de la capa *switch* (`c28conv8_s`) y antes de la siguiente capa convolucional (`c31clasif`) se encuentra (al igual que en la arquitectura TinyYOLOvP) una capa `BatchNormalization` (`c29norm`) y una capa `LeakyRelu` (`c30relu`), donde la primera es una capa que también contiene parámetros entrenables, es decir, también tiene pesos. La cantidad de filtros de `c28conv8` representa la cantidad de canales de la capa `c29norm` y de `c31clasif`, al modificarse los filtros de `c28conv8` se deben modificar los canales de esas capas subsecuentes, de ahí que finalmente se podan los filtros de `c28conv8`, los canales de `c29norm` y los canales de `c31clasif`. La poda de estos filtros y canales mencionados se lleva cabo con un barrido de la variable `W2_diag` (líneas 20-28), donde posición a posición se evalúa si se conservan o se eliminan los pesos correspondientes a esas posiciones, pero de las capas a podar (octava convolucional, subsecuente `BatchNormalization` y subsecuente convolucional). Al final del barrido un nuevo conjunto de variables (definidas en las líneas 12-18) almacenan únicamente los pesos de los filtros y canales asociados más importantes. Por último, y debido a que la capa *switch* con sus valores de *S* se removerá, se hace la compensación multiplicando los valores de *S* que no eran cero por los nuevos pesos de `c28conv8_p` y de `c29norm_p`. Debido a que los canales de `c31clasif_p` están después de la capa de activación no es necesario compensarlos.

```
12: new_filters,new_channels,new_switch = [],[],[]
13: new_batch1,new_batch2,new_batch3,new_batch4 = [],[],[],[]
14: n_s = W2_diag.shape[0]
15: for i in range(n_s):
16:     if W2_diag[i] > 0:
17:             new_filters.append(W[35][:, :, :, i])
18:             new_batch1.append(W[37][i])
```

```
19:             new_batch2.append(W[38][i])
20:             new_batch3.append(W[39][i])
21:             new_batch4.append(W[40][i])
22:             new_channels.append(W[41][:, :, i, :])
23:             new_switch.append(W2_diag[i])
24: new_filters  = np.rollaxis(np.array(new_filters), 0, 4)
25: new_channels = np.rollaxis(np.array(new_channels), 0, 3)
26: new_filters *= new_switch
27: new_batch1  *= new_switch
28: new_batch2  *= new_switch
29: new_batch3  *= new_switch
30: new_batch4  *= new_switch
```

- **Paso 4.** Se define la arquitectura de la red TinyYOLOvPP (líneas 33-54), removiendo la capa *switch* y cambiando el número de filtros de la octava convolucional, en lugar de los 1,024 filtros que tenía dicha capa del modelo TinyYOLOvP, se reduce a la cantidad de valores de $S$ que no eran cero, para el caso de esta experimentación con $\lambda = 10^{-5}$ los filtros activados fueron 101, valor obtenido mediante las líneas 31 y 32, donde en la línea 31 gracias a las herramientas de NumPy es posible evaluar cuáles valores del arreglo `pesos` son mayores a 0 y asignarles sólo a ellos el valor de 1 en una simple línea de código, para luego en la línea 32 sumar esos valores y obtener la cantidad de datos que fueron mayores que cero, lo cual representa el número de filtros que no fueron afectados por la penalización y que se conservarán en la capa `c28conv8_p` de la arquitectura TinyYOLOvPP (Figura 6.3).

```
31: pesos[pesos>0] = 1
32: num_filters = int(np.sum(pesos))
```

```
# Arquitectura TinyYOLOvPP
input_image = Input(shape=(416, 416, 3))
c1conv1 = Conv2D(16, (3,3), strides = (1,1), padding =
  'same', use_bias = False) (input_image)
c2norm = BatchNormalization(name = 'norm_1')(c1conv1)
c3relu = LeakyReLU(alpha = 0.1)(c2norm)
x = MaxPooling2D(pool_size = (2, 2))(c3relu)
for i in range(0,4):
    x = Conv2D(32*(2**i), (3,3), strides = (1,1),
         padding = 'same', use_bias = False)(x)
    x = BatchNormalization(name = 'norm_' + str(i+2))(x)
    x = LeakyReLU(alpha = 0.1)(x)
    x = MaxPooling2D(pool_size = (2, 2))(x)
c21conv6 = Conv2D(512, (3,3), strides = (1,1), padding =
  'same', use_bias = False)(x)
c22norm = BatchNormalization(name = 'norm_6')(c21conv6)
c23relu = LeakyReLU(alpha = 0.1)(c22norm)
c24pool = MaxPooling2D(pool_size = (2, 2),
  strides = (1,1), padding = 'same')(c23relu)
c25conv7 = Conv2D(1024, (3,3), strides = (1,1), padding =
  'same', use_bias = False)(c24pool)
c26norm = BatchNormalization(name = 'norm_7')(c25conv7)
c27relu = LeakyReLU(alpha = 0.1)(c26norm)
c28conv8_p = Conv2D(num_filters, (3,3), strides = (1,1),
  padding = 'same', use_bias = False)(c27relu)
c29norm_p = BatchNormalization(name = 'norm_8')(c28conv8_p)
c30relu = LeakyReLU(alpha = 0.1)(c29norm_p)
c31clasif_p = Conv2D(30, (1,1), strides = (1,1),
  padding = 'same')(c30relu)
model = Model(input_image, c31clasif_p)
```

- **Paso 5.** Por último, se cargan capa por capa los pesos para obtener la red modificada TinyYOLOvPP (líneas 55-71) transfiriendo los datos almacenados en las listas generadas en el paso anterior, de la siguiente forma: a) los filtros podados para la capa `c28conv8_p` (`model.layers[28]`) se actualizan a partir de la lista `new_filters`; b) los pesos podados de la capa `c29norm_p` (`model.layers[29]`) se obtienen de las listas `new_batch1, new_batch2, new_batch3` y `new_batch4`; c) los canales podados para la capa `c31clasif_p` (`model.layers[31]`) se transfieren a partir de la lista `new_channels`.

```
55: model.layers[28].set_weights([new_filters])
56: model.layers[29].set_weights([new_batch1, new_batch2,
        new_batch3, new_batch4])
57: model.layers[31].set_weights([new_channels, W[42]])
58: model.layers[1].set_weights([W[0]])
59: model.layers[2].set_weights([W[1], W[2], W[3], W[4]])
60: model.layers[5].set_weights([W[5]])
61: model.layers[6].set_weights([W[6], W[7], W[8], W[9]])
62: model.layers[9].set_weights([W[10]])
63: model.layers[10].set_weights([W[11], W[12], W[13], W[14]])
64: model.layers[13].set_weights([W[15]])
65: model.layers[14].set_weights([W[16], W[17], W[18], W[19]])
66: model.layers[17].set_weights([W[20]])
67: model.layers[18].set_weights([W[21], W[22], W[23], W[24]])
68: model.layers[21].set_weights([W[25]])
69: model.layers[22].set_weights([W[26], W[27], W[28], W[29]])
70: model.layers[25].set_weights([W[30]])
71: model.layers[26].set_weights([W[31], W[32], W[33], W[34]])
```

## 6.4. Resultados obtenidos durante el proceso de Poda de Filtros.

En esta sección se presentan los resultados de la aplicación de la técnica de poda de filtros CNN-PSL propuesta, misma que fue aplicada sobre la octava capa convolucional del modelo CNN detector de personas (TinyYOLOvP) basado en TinyYOLOv2. En la Figura 6.6 se muestra una gráfica de la relación de los resultados obtenidos ante la variación del valor de penalización ($\lambda$) para la capa *switch*. Cada punto (triángulos) en la figura representa un valor de $\lambda$ distinto con respecto a la Tabla 6.1.

Tabla 6.1. Valores de $\lambda$ para la penalización de la capa *switch*.

| $\lambda_1$ | $\lambda_2$ | $\lambda_3$ | $\lambda_4$ | $\lambda_5$ | $\lambda_6$ |
|---|---|---|---|---|---|
| $10^{-6}$ | $10^{-5}$ | $2 \cdot 10^{-5}$ | $3 \cdot 10^{-5}$ | $5 \cdot 10^{-5}$ | $10^{-4}$ |

Para cada valor de $\lambda$ se obtienen resultados de la precisión (mAP) de la red y de la cantidad de filtros activados en la octava capa convolucional, es decir, la cantidad de filtros que la capa *switch* detectó que aportan más al resultado de la red después de aplicar las respectivas penalizaciones.

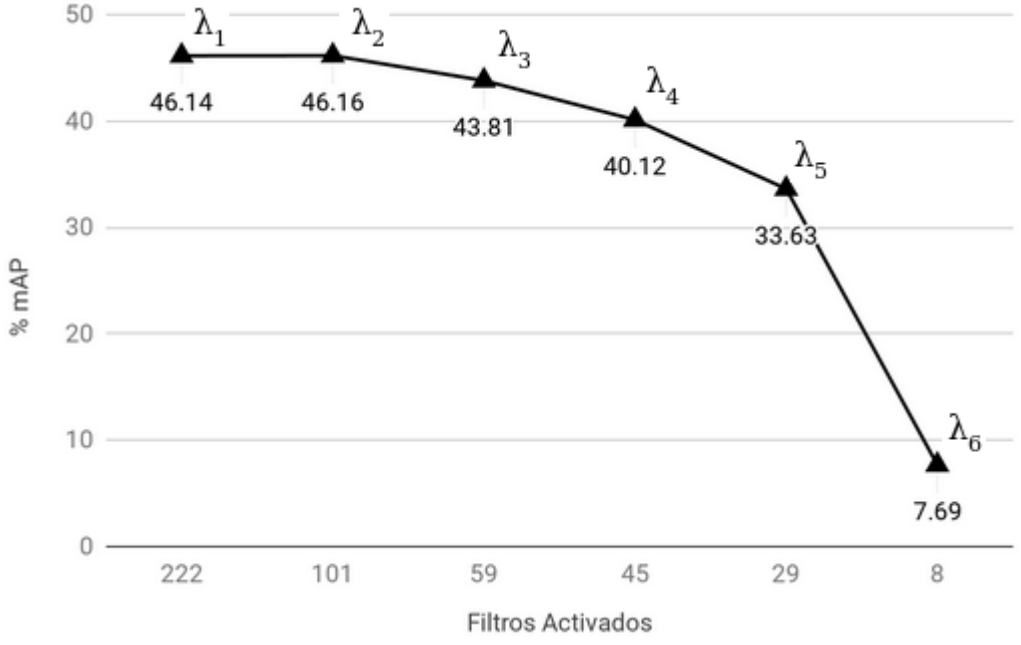


Figura 6.6. Relación de filtros activados y precisión (mAP) para distintos valores de penalización ($\lambda$).

Con base en el límite de 5% de margen de pérdida de precisión establecido en el requerimiento 1.4.6, a partir de la experimentación realizada (Figura 6.6) se seleccionó el modelo CNN con 101 filtros activados en la octava capa convolucional y una precisión de 46.16% mAP, esto debido a que con la siguiente prueba la precisiń cayó de tal manera (43.81% mAP) que superó el 5% de margen de pérdida de precisión (49.67 - 43.81 = 5.86), siendo entonces el modelo CNN obtenido bajo el criterio de valor de $\lambda_2$ ($10^{-5}$) que detectó 101 filtros activados el que aportó la mayor cantidad de filtros a podar sin quedar fuera de la precisión del requerimiento 1.4.6.

Ahora, en lugar de que la octava capa convolucional cuente con 1,024 (`c28conv8`) filtros, serán sólo 101 (`c28conv8_p`). Esta reducción impacta en el número de parámetros de la red, lo cual es directamente proporcional al tamaño del modelo (MB), además reduce su complejidad computacional, con lo cual el tiempo de inferencia disminuye. En la Tabla 6.2 se hace la comparación de los modelos TinyYOLOv2, TinyYOLOvP (§5.2.2) y del modelo CNN resultante de la aplicación de la técnica propuesta de poda de filtros TinyYOLOvPP, siendo la reducción de filtros (y respectivos canales de capas subsecuentes) la única diferencia con respecto a TinyYOLOvP. Esta comparación se realiza con las métricas de cantidad de categorías de clasificación, precisión (mAP), cantidad de parámetros y tamaño del modelo (MB).

Tabla 6.2. Resultados TinyYOLOvPP.

| Modelo | Clases | Precisión (%mAP) | Parámetros | Tamaño (MB) |
|---|---|---|---|---|
| TinyYOLOv2 | 20 | 57.10 | 15,867,885 | 63.6 |
| TinyYOLOvP | 1 | 49.67 | 15,770,510 | 63.2 |
| TinyYOLOvPP | 1 | 46.16 | 7,232,760 | 29.0 |

En estos resultados se destaca el grado de compresión que la red podada alcanzó con la reducción de filtros mencionada, así como la pérdida de precisión. Revisando las métricas con base en los requerimientos (§1.4.6) se tuvieron los siguientes resultados:

Pérdida de precisión debido a la técnica de poda = 3.51 %

Compresión = 54.41 %

La pérdida de precisión se considera con respecto al modelo TinyYOLOvP debido a que, como se mencionó en 5.2.2, la precisión después de transferir el aprendizaje de TinyYOLOv2 es posible mejorarla a través de ciertas técnicas ahí mencionadas, y por lo tanto las pérdidas de precisión que se busca controlar son las de la técnica de poda. Así, se observa una pérdida de precisión de un 3.51%, que cumple con el requerimiento de una pérdida máxima del 5%, y una compresión del 54.41% que también cubre el requerimiento de una compresión del 50% (§1.4.6).

Finalmente, la última métrica a comparar es el rendimiento (FPS), para lo cual se evaluaron los modelos TinyYOLOvP y TinyYOLOvPP con base en las experimentaciones hechas en la sección 4.5. En la Tabla 6.3 se muestra una comparación de estos dos modelos junto con el de TinyYOLOv2, presentando los resultados del rendimiento promedio en CPU para una muestra de 200 *frames* para cada uno de los dispositivos RPi 3, NUC y iPad. Posteriormente, se repitieron dichas mediciones considerando la adición de aceleradores neuronales para los nodos de cómputo en la niebla usando una o dos unidades NPU externas Intel Movidius M2 (Myriad 2) y la NPU más avanzada y reciente Movidius MX (Myriad X).

Tabla 6.3. Resultados de velocidad de procesamiento (FPS promedio) por plataforma.

| Modelo CNN | Rend. Raspberry Pi 3 (FPS) | | | | Rend. Intel NUC Core i5 (FPS) | | | | iPad A10 |
|---|---|---|---|---|---|---|---|---|---|
| | CPU | 1 NPU M2 | 2 NPU M2 | 1 NPU MX | CPU | 1 NPU M2 | 2 NPU M2 | 1 NPU MX | |
| Tiny YOLO v2 | 0.5 | 2.5 | 5.8 | 3.2 | 5.5 | 4.7 | 12.4 | 10.2 | 5.9 |
| Tiny YOLO vP | 0.5 | 2.5 | 5.8 | 3.2 | 5.7 | 5.0 | 13.1 | 10.5 | 6.2 |
| Tiny YOLO vPP | 0.7 | 3.2 | 6.0 | 4.8 | 7.6 | 6.9 | 18.0 | 17.1 | 11.3 |

M2 = Myriad 2 (Movidius 1), MX = Myriad X (Movidius 2)

En la Tabla 6.3 es posible observar la similitud de los resultados entre el modelo original (TinyYOLOv2) y el modelo modificado con TL (TinyYOLOvP), es decir, no se obtuvo una mejora de rendimiento significativa, debido a que son prácticamente modelos CNN con la misma cantidad de parámetros y la única diferencia en su arquitectura es la cantidad de filtros presentes en la capa final del clasificador (125 vs 30 filtros), debido a la reducción de categorías de clasificación producto de la técnica TL aplicada para el segundo modelo CNN (TinyYOLOvP). En cambio, para el modelo CNN podado (TinyYOLOvPP) se obtuvo un incremento en la velocidad de procesamiento en todos los dispositivos (39% promedio para CPU y 40% promedio para NPU externo). Aplicando la técnica de poda se logró satisfacer el requerimiento de incrementar un 50% la velocidad de procesamiento (§1.4.6), lograda para el caso del dispositivo *Edge* iPad (91%), y para los dispositivos *Fog*, NUC + 1 NPU MX (67%) y RPi + 1 NPU MX (50%). Esta técnica en particular presentó un incremento en velocidad de procesamiento destacado para el caso de los dispositivos con acelerador neuronal externo (NPU Intel Movidius) en comparación con la aceleración lograda con CPU, esto a pesar de que la literatura menciona que en los aceleradores NPU externos se presentan latencias significativas al momento de transferir datos por puertos USB 3 (Allan, 2019; Vincent, 2017), lo cual se convierte en un cuello de botella importante durante la fase de procesamiento del modelo CNN, efecto que puede degradar incluso el rendimiento de un modelo CNN modificado mediante la técnica de poda.

Para colocar los resultados obtenidos en contexto con otras técnicas de poda, se presenta la Tabla 6.4, adaptada del trabajo de Singh, Verma, Rai, & Namboodiri (2019). En esta tabla se muestra la aplicación de seis técnicas de poda al modelo VGG-16, con sus respectivas pérdidas de precisión, porcentaje de reducción de parámetros y porcentaje de reducción de FLOPs. Se puede observar como la mayoría de estas técnicas tienen un porcentaje de reducción de parámetros alrededor del 93%, una reducción de FLOPs promedio de 82% (para los cuatro casos más altos) y con pérdidas de precisión de menos del 1%.

Por otro lado, Lee, Ajanthan, & Torr (2019) también presenta una aplicación de poda a VGG, logrando un 95% de compresión de parámetros y con pérdidas de precisión de menos del

0.5%. En un trabajo más relacionado al de la presente tesis, Szemenyei & Estivill-Castro (2019) aplican una técnica de poda sobre un modelo de detección de objetos basado en TinyYOLOv3, alcanzando una reducción de parámetros del 90% y aceleración de ejecución del 70%.

Tabla 6.4. Comparación de poda de VGG-16 entrenado en CIFAR-10. Adaptada de Singh et al. (2019).

| Técnica | Pérdida de Precisión (%) | Parámetros Podados (%) | FLOPs Podados (%) |
|---|---|---|---|
| Li-pruned | 0.09 | 64.0 | 34.20 |
| SBP | 0.99 | 75.1* | 56.52 |
| AFP-E | 0.55 | 93.3 | 79.69 |
| AFP-F | 0.62 | 93.5 | 81.39 |
| PP-1 | 0.03 | 92.5 | 82.8 |
| PP-2 | 0.14 | 94.3 | 84.5 |

*Los autores de SBP (Neklyudov, Molchanov, Ashukha, & Vetrov, 2017) no proporcionan información de parámetros podados, sólo de filtros podados los cuales fueron un 75.1%.

En comparación, la técnica CNN-PSL aquí presentada logró una reducción de parámetros del 54.13% (considerando la reducción obtenida sólo por la poda), aceleración promedio de 45% (tomando en cuenta cada plataforma de experimentación) y una pérdida de precisión del 3.51%. Es importante resaltar, que estos resultados se alcanzaron haciendo la poda de solamente una capa, que representa el 60% de los parámetros totales de la red. Extender la técnica de poda al 40% restante distribuido en las demás capas puede acercar los resultados a los obtenidos en la literatura presentada. Actualmente, para la técnica de de CNN-PSL existe la limitación de perder más precisión en caso de continuar podando capas, lo cual se podría solucionar entrenando la red después de la poda para recuperar precisión, el problema está en que después de la aplicación de CNN-PSL, la red no es capaz de recuperar precisión con un reentrenamiento. La literatura sugiere que este efecto se debe a que la aplicación de esta técnica elimina un 54.13% de los parámetros totales de la red en una sola iteración, esto crea inestabilidad en la red y se traduce en una incapacidad de recuperar precisión (Frankle & Carbin, 2019; Li et al., 2017; O'Keeffe & Villing, 2018). Una alternativa es llevar a cabo la poda con

múltiples iteraciones, podando muchos menos filtros por iteración para evitar desestabilizar la red. Un ejemplo es el de O'Keeffe & Villing (2018), donde implementan una poda eliminando un filtro por iteración, al final tienen una pérdida de precisión del 5% pero después de reentrenar la precisión se restablece.

Por lo tanto, solucionando el problema de la pérdida de precisión, es posible aplicar CNN-PSL al resto de capas y regresar a comparar los resultados finales con el resto de trabajos de la literatura.

# VII. ANÁLISIS DE RESULTADOS, CONCLUSIONES Y TRABAJO A FUTURO

A continuación, se presentan los análisis, conclusiones y trabajo futuro de acuerdo a los resultados obtenidos para cada una de las etapas metodológicas (§1.5), así como el cumplimiento de cada requerimiento planteado respecto al problema a resolver (§1.4) y la validación de la hipótesis de la tesis (§1.3).

## 7.1. Detección de personas con TinyYOLO.

Se implementó un modelo CNN pre-entrenado para la detección de personas analizando secuencias de video en tiempo real en diversas unidades de cómputo en la frontera y en la niebla. Se seleccionó TinyYOLOv2 por su tamaño, su menor número de clases y por contar con la mayor velocidad de procesamiento con respecto a los otros modelos CNN investigados en la revisión del estado del arte. TinyYOLOv2 localiza y clasifica objetos entre 20 categorías (entre ellas personas), dibujando cuadros delimitadores alrededor del objeto detectado (*bounding boxes*). Se investigó y verificó su compatibilidad con cada *framework* y plataforma de prueba, como resultado de esa investigación se sintetizó la información en un diagrama de la Figura 4.4 y la Tabla 4.2. La Figura 4.4 atiende la problemática de compatibilidad entre *frameworks DL* y hardware como son los aceleradores neuronales o los dispositivos Edge/Fog y sus respectivos sistemas operativos. Esta figura resume las conversiones que es posible realizar entre *frameworks DL* para lograr la interoperabilidad en distintos dispositivos de un mismo modelo CNN con una arquitectura y tipos de capas específicos.

Gracias al conocimiento de dicho flujo de trabajo (Figura 4.4) fue posible verificar rápidamente la viabilidad de implementación en los dispositivos requeridos durante la selección de un modelo CNN entre diversos candidatos posibles. Una vez seleccionado el modelo CNN, se implementaron en Keras las técnicas de adaptación y posteriormente se implementó el modelo CNN adaptado para ser ejecutado usando CoreML y OpenVino para iOS y

Raspbian/Ubuntu respectivamente. En este último caso, se realizaron también las reconfiguraciones correspondientes para ejecutar el modelo CNN usando los aceleradores neuronales externos Intel Movidius (Myriad 2 y Myriad X).

Como complemento, la Tabla 4.2 aborda la problemática de compatibilidad al nivel de tipos de capas de una CNN, ya que solamente la compatibilidad entre *frameworks DL*, dispositivos y sistemas operativos no era suficiente garantía para la interoperabilidad. También existe el problema de que un modelo CNN a implementar posea capas particulares que no tengan soporte en ciertos *frameworks*. La Tabla 4.2 muestra las capas con mayor compatibilidad entre distintos *frameworks*, con ello se verificó la compatibilidad de TinyYOLOv2, pero además, esta tabla funciona como guía para el diseño de nuevos modelos, sugiriendo las capas a las que se debe limitar un modelo para garantizar una interoperabilidad entre diversos dispositivos, o bien, las capas a las que se debe limitar el modelo para su implementación en determinado dispositivo o sistema operativo con base en el *framework* correspondiente.

La compatibilidad del modelo se alcanzó para cada plataforma, aún así el proceso de conversiones para lograrlo es complicado, tanto por la cantidad de conversiones requeridas para ejecutar el modelo en cada plataforma como por el cuidado que se debe tener con las versiones de cada *framework*, biblioteca o lenguaje de programación. Por ejemplo, el caso de convertir un modelo de Keras a CoreML requiere que se usen exclusivamente las versiones de Python 2.7 y Keras 1.2.2, o el caso de convertir de Keras a TensorFlow que requiere la versión 1.12 de este último. Estos *frameworks* y bibliotecas sufren actualizaciones constantemente, en algunos casos facilitando y en otros complicando la compatibilidad, por ello es recomendable seguir de cerca esas actualizaciones, aprovechar las ventajas y prepararse para las desventajas que conllevan.

Como trabajo a futuro sería muy interesante implementar el uso de contenedores (*containers*) como es el caso de Docker, con los cuales encapsular estas aplicaciones y facilitar la compatibilidad e interoperabilidad entre plataformas, evitando tener que instalar *frameworks* y bibliotecas en cada plataforma de destino. También se recomienda la evaluación de otros modelos CNN de detección de objetos con características similares, ya que a pesar de registrar

menor velocidad de procesamiento que TinyYOLOv2, sería bueno explorar su comportamiento ante su implementación con los aceleradores neuronales o la aplicación de técnicas de adaptación. Incluso, saliendo del contexto de ANN, también se pueden evaluar y comparar los resultados obtenidos con otros modelos de detección de personas como *Harr Cascaded* o HOG+SVD (Nikouei, Xu, & Chen, 2019), con el fin de comparar y validar los modelos CNN como una solución para la detección de personas.

## 7.2. Ejecución de TinyYOLO con Cómputo en la Frontera y en la Niebla.

Como primera validación de la hipótesis se hicieron pruebas iniciales con modelos CNN de clasificación con cómputo en la frontera, comprobándose su viabilidad de implementación a través de la ejecución de InceptionV3, MobileNet y SqueezeNet, y aunque para el primero se alcanzaron sólo 1.5 FPS (una baja velocidad considerando el mínimo requerido de 2 FPS), para los otros dos se alcanzaron buenas velocidades de 9.9 y 15.5 FPS respectivamente, esto permitió validar el uso de modelos CNN en dispositivos *Edge* y continuar con la implementación de modelos CNN más complejos como los detectores de objetos. En la segunda validación de la hipótesis se logró ejecutar el modelo TinyYOLOv2 en cada dispositivo de prueba, atendiendo las conversiones necesarias para satisfacer en un 100% la portabilidad del modelo CNN para las distintas configuraciones de prueba. En 8 de 9 plataformas de experimentación se alcanzó el rendimiento mínimo de 2 FPS, siendo el caso de la RPi con ejecución del modelo en CPU la prueba más lenta con 0.5 FPS. Se implementó con éxito el sistema de iluminación con base en la detección de personas con TinyYOLOv2 para las plataformas RPi y NUC. Para una evaluación y comparación más rápida y precisa, las pruebas del modelo original y adaptado se hicieron con el sistema de detección de personas sin carga, es decir, sin la acción de control sobre la iluminación. El rendimiento del sistema de detección de personas decrece aproximadamente 0.1 FPS con la implementación del control de iluminación. Para este sistema, el uso de la cámara del iPad resulta impráctico debido a la misma movilidad del dispositivo, por ello el control de iluminación se limitó a RPi/NUC a pesar del buen rendimiento de 5.9 FPS obtenido por el modelo TinyYOLOv2 en el dispositivo iPad.

Tanto para la RPi como para la NUC, se obtuvo una aceleración destacada con los NPU, logrando mejores resultados con dos Movidius M2 que con una Movidius MX. Para la NUC, los resultados con CPU y con un NPU M2 son similares, pero superando al CPU en un 125% y 85% con dos NPU M2 y un NPU MX respectivamente. Aunque alcanzando menor velocidad, en la RPi el porcentaje de aceleración con respecto a la ejecución en CPU fue mucho mayor, mejorando en un 400%, 1,060% y 540% con ejecuciones en un NPU M2, dos NPU M2 y un NPU MX respectivamente, demostrando que la aceleración de modelos CNN con este tipo de dispositivos destaca por encima de la aceleración con las técnicas de adaptación presentadas en esta tesis. Estos resultados son importantes ya que con un sólo NPU M2 se alcanzan los 2 FPS del requerimiento, considerando que de los dispositivos de pruebas es el de las capacidades más limitadas. Por otro lado, si bien las NPU Intel Movidius tienen excelentes resultados en aceleración de inferencias, no aporta alguna solución al problema de compresión para reducir el consumo de memoria. Por ejemplo, Nikouei et al. (2019) mencionan el resultado de la experimentación del modelo VGG con ejecución en una RPi 3 modelo B. Cuando cargan esta red a la RPi, obtienen un error debido a que el espacio de RAM disponible es bajo y la ejecución se interrumpe. De ahí la importancia de compactar estos modelos para su uso en dispositivos con recursos limitados.

En próximos experimentos se recomienda evaluar otras plataformas de hardware como la RPi 4, RockPro, Google Coral DevBoard o Jetson Nano, algunos de estos dispositivos cuentan con puertos USB 3.0 con los que se puede aprovechar de mejor manera el rendimiento de las NPU Intel Movidius, en cuyo caso se debe investigar su funcionamiento interno a profundidad, así como características y limitaciones clave para obtener el mejor desempeño posible. Otra opción es evaluar otros aceleradores neuronales como el mismo Google Coral TPU o el A12 Bionic de Apple, validando para cada caso la compatibilidad del modelo CNN y así poder comparar resultados. Incluso extendiendo esa experimentación, se pueden llevar a cabo análisis comparativos entre resultados con pruebas en CPU, GPU y NPU, este trabajo se limitó a CPU y NPU (para la etapa de inferencia), pero para las CNN, las GPU son también una opción viable con altas velocidades de procesamiento, pero también a un alto costo tanto de energía como económico, compensaciones dignas de evaluarse y compararse con CPU’s y NPU’s.

## 7.3. Adaptación de TinyYOLO a Dispositivos Limitados mediante las Técnicas de CNN-PSL, TL y Cuantización.

En la tercera validación de la hipótesis, las primeras aproximaciones se hicieron para la adaptación de modelos CNN usando las herramientas CoreMLTools y CreateML de iOS para aplicar las técnicas de cuantización y transferencia del aprendizaje (TL) respectivamente. Ambas implementaciones presentaron resultados positivos, en el caso de cuantización se logró reducir la representación de los pesos de 32 a 8 bits, decreciendo con ello cuatro veces el tamaño del modelo, manteniendo a la vez sin cambios notables la velocidad de procesamiento en la etapa de inferencia del modelo CNN. Aunque no se logró evaluar la precisión de forma metódica y comparativa, la aplicación conservó la misma funcionalidad que el modelo CNN original, si bien existiera alguna pérdida de precisión, al observar los resultados de detección mostrados en la pantalla del dispositivo esta no es notoria.

Cuando se aplicó la técnica TL con Create ML en iOS, se hicieron pruebas con modelos CNN de clasificación de objetos. Utilizando la base de datos INRIA (741 imágenes) para su entrenamiento y evaluación se alcanzó una precisión del 96% en la etapa de inferencia del modelo CNN. Sin embargo, los resultados con la base de datos propia BDLab (44% de precisión al evaluarse en INRIA) demuestran que la red CNN no logra generalizar de manera adecuada, esto debido mayormente al reducido tamaño de dicha base de datos de entrenamiento (21 imágenes). Los modelos CNN obtenidos con la herramienta CreateML fueron de tamaño sumamente reducido (17 KB), con la limitante de que sólo fue posible aplicar la técnica de transferencia de aprendizaje para modelos CNN de clasificación de objetos, sin opciones de aplicación para modelos de detección de objetos.

Para las implementaciones de modelos CNN en iOS se recomienda la aplicación de la técnica de cuantización como último paso antes de cargar la aplicación móvil al dispositivo. También se recomienda el uso de TL con CreateML para aplicaciones sencillas que consistan en clasificación de imágenes. Como trabajo a futuro se propone explorar a fondo las futuras actualizaciones de los *frameworks* de Xcode versión 11 (disponibles a partir de septiembre de

2019) específicamente las herramientas CreateML, CoreMLTools y CoreML 3, para las cuales se han anunciado la incorporación de novedosas funcionalidades como la posibilidad de realizar entrenamiento de modelos CNN *on-device* (Apple Inc., 2019d).

Respecto a los dispositivos en la frontera y en la niebla basados en Linux, mediante el uso de Python y el *framework* Keras, la implementación de la técnica de cuantización no fue posible debido a errores (§5.2.1) que existen actualmente en la versión 2.2.2 de Keras, aún así se presentó la posibilidad de cuantizar convirtiendo el modelo de Keras a TFLite, conversión nativa debido al uso de TensorFlow como *backend* y que tiene implementación para dispositivos Android.

Al aplicar la técnica TL se derivó un modelo CNN exclusivo para detección de personas, eliminando el resto de clases de TinyYOLO, lo cual amplió las posibilidades de comprimir el modelo y acelerar la inferencia a través de técnicas de adaptación basadas en la redundancia de parámetros. El modelo TinyYOLOvP resultante de aplicar TL alcanzó una precisión de 49.67% mAP, que está por debajo del 57.1% del modelo TinyYOLO original. Es importante resaltar que para esta etapa de la experimentación no fue una prioridad alcanzar altos niveles de precisión, sobretodo porque ello habría implicado mucho más tiempo de entrenamiento y porque el enfoque ha sido la aplicación de técnicas de adaptación. Por supuesto que como trabajo a futuro se recomienda llevar a cabo experimentaciones con más épocas de entrenamiento y aplicando técnicas que ayuden a mejorar la precisión, tales como aumentación de datos y entrenamiento multi-escala.

Para superar el requerimiento 1.4.6, se propuso, implementó y validó una nueva técnica de adaptación por poda de redes llamada CNN-PSL (*CNN Pruning by Switch Layer*), partiendo de la observación relativa a la existencia de un alto porcentaje de redundancia en la red CNN que se presenta al momento de aplicar la técnica TL para derivar un modelo CNN con sólo una clase. Esta técnica pertenece a la categoría de poda de redes dependiente de datos, ya que la selección de pesos a podar la realiza a través de un proceso de entrenamiento, pero a diferencia de otros métodos que luego de entrenar y hacer la poda necesitan hacer *fine-tuning* o reentrenar

para recuperar precisión, esta técnica sólo requiere el entrenamiento para la selección de pesos y después ya no es necesario entrenar. Además, es una técnica de poda de filtros completos (ya que existen otras que podan sólo canales o kernels de manera parcial), esto se logra incorporando a la red una capa convolucional extra llamada *switch* después de la capa que se desea podar, capa que se configura para que durante el entrenamiento la función de costo calcule los valores en la capa *switch* con lo cuales determinar qué filtros podar. La técnica se aplicó sobre la octava capa convolucional debido a que es la que cuenta con más parámetros. Se hicieron pruebas con 6 valores distintos de penalización del *switch* ($\lambda$) para seleccionar el caso que tuviera la mejor relación mAP/Parámetros. La mejor combinación se alcanzó con $\lambda=10^{-5}$, obteniendo como resultado la activación de 101 filtros de los 1024 que tenía originalmente (compresión en esa capa del 90% y compresión total del modelo del 54.13%), alcanzando el requerimiento del 50% de compresión sólo con poda en la octava capa convolucional, ya que el modelo se redujo de 63.2 MB a 29 MB. Además, se obtuvo una precisión del 46.16% mAP, que representa una pérdida del 3.51%, la cual está dentro del rango del 5% de pérdida establecido en los requerimientos. Con respecto a la métrica de velocidad de procesamiento (FPS), la aceleración de mínimo 50% se alcanzó para los casos del iPad (91%), NPU MX en NUC (67%) y NPU MX en RPi (50%). Los resultados de menor aceleración fueron en la RPi, aunque sin mucha diferencia con respecto a la NUC, con excepción de la prueba con dos NPU M2 donde hubo una aceleración del 3%, requiriendo mayor investigación para encontrar las causas ya que, aunque fue el caso con menor aceleración, fue la prueba en RPi que alcanzó mayor velocidad (6 FPS).

Como se mencionó al final de la sección 6.4, solucionando el problema de la pérdida de precisión es posible aplicar CNN-PSL al resto de capas y aspirar a obtener resultados similares al resto de trabajos de la literatura. Por ello, dentro del trabajo a futuro se plantea la aplicación de esta técnica con múltiples iteraciones, así como su implementación al resto de capas de la red. Por otro lado, como se había mencionado, Keras es un *framework* para el desarrollo rápido de prototipos, lo que convierte a la aplicación de esta técnica en una opción práctica y de relativamente fácil implementación, pero también con limitaciones propias del *framework*. Implementar CNN-PSL y TL con *frameworks* como TensorFlow o PyTorch es una opción con la cual seguramente se obtendrán mejores resultados al poder usar herramientas de bajo nivel

que eficienticen las técnicas. Incluso la técnica de poda se puede extender, una vez podados los filtros también se pueden hacer pruebas para complementar la poda eliminando canales y hasta pesos individuales.

Como trabajo a futuro también se recomienda evaluar las técnicas de poda independiente de datos, el trabajo mencionado de Lee et al. (2019) es un ejemplo de ello. Mientras que las técnicas dependientes de datos requieren de varias ejecuciones de entrenamiento para ajustar los hiper-parámetros de la regularización (como fue el caso de $\lambda$ en esta tesis), las técnicas independientes de datos suelen sólo entrenar una vez, pero ejecutar en varias ocasiones su algoritmo de selección de filtros a podar (*saliency criterion*), el cual es computacionalmente menos costoso que el entrenamiento. Estas opciones de poda, así como mejorar el entrenamiento para TL e investigar alternativas para cuantización, son trabajo a futuro relacionado a las tres técnicas aplicadas en este trabajo de tesis, las cuales demostraron gran compatibilidad para complementarse. De hecho, el modelo podado TinyYOLOvPP fue posible de cuantizar con las herramientas de iOS para su implementación en el dispositivo iPad, reduciendo su tamaño de 29 MB a 7.25 MB para una compresión total con respecto a TinyYOLOv2 del 88%.

Finalmente, se recomienda evaluar otras técnicas de adaptación, ya sea con enfoque a nivel de creación de modelo eficientes, o bien, con enfoque a nivel de implementación (§3.5), ya que como se pudo observar, la aceleración relacionada al hardware tiene gran potencial.

## 7.4. Control de Iluminación del Aula Inteligente.

Los requerimientos referentes al desarrollo de una aplicación capaz de detectar personas para controlar la iluminación de un aula inteligente, mismos que establecen un escenario para la validación y comparación del modelo CNN (original y adaptado) fueron cubiertos y detallados en la implementación del prototipo presentado en la sección 4.6. Dicha aplicación localiza a las personas en un flujo continuo de video provisto por una cámara web de bajo costo, y con base en eso toma la decisión de cuál zona del aula debe ser iluminada activando los focos LED Philips Hue (dispositivos IoT). Esto se efectúa sin contar con el apoyo de ningún servicio de cómputo

en la nube, realizando todo el procesamiento con cómputo en la niebla a través de la RPi o la NUC.

La aplicación tiene ciertas limitaciones que se pueden abordar en trabajos futuros, como el control de intensidad de iluminación con base en la ubicación de las personas, lo cual ayudaría a obtener cambios graduales en la activación e intensidad de iluminación de los focos. Actualmente el prototipo presenta parpadeos cuando una persona se ubica lentamente entre dos zonas, esto debido a que en las múltiples inferencias que lleva a cabo el modelo CNN, en ocasiones detecta a la persona en una zona y con una pequeña variación en la siguiente inferencia puede detectarlo en la otra zona, activando y desactivando los focos, lo que provoca el efecto de parpadeo en la iluminación. Otra limitación es la incapacidad del sistema actual de detectar profundidad, es decir, si una persona pasa enfrente de la cámara activará la zona correspondiente, aunque realmente no haya alguien en la pared/pizarrón, no identifica a qué distancia se está detectando a la persona para poder decidir si activar las luces o no. También se puede trabajar en la habilitación del control de otros dispositivos como podría ser un termostato o una chapa eléctrica, aplicaciones que se benefician de conocer la presencia, localización y cantidad de personas dentro del aula. Por otro lado, esta implementación puede servir de referencia o escalarse incorporando más cámaras para atender varias aulas o para un escenario más grande como edificios inteligentes y algunas aplicaciones para ciudades inteligentes como control de tráfico o medición de densidad de peatones en determinadas zonas de interés.

El desarrollo de esta aplicación de detección de personas para el control de iluminación a través de dispositivos de recursos limitados representa los primeros escalones para alcanzar la visión de un aula o edificio inteligente con capacidades no sólo de toma de decisiones con base en las condiciones de presencia de personas, sino también siendo un sistema con capacidades adaptativas con base en una percepción más completa del entorno. A partir de estos primeros escalones, los siguientes pasos se pueden dar en dirección de las aplicaciones multimodales (Liu, Li, Xu, & Natarajan, 2018; Ngiam et al., 2011) que pueden combinar distintos modelos DL de texto, imágenes o audio para un mejor análisis del entorno. A su vez, la investigación también puede continuar con la adición de técnicas de *tracking* (Özer, Gürkan, & Günsel, 2018) para incorporar información espacio-temporal y habilitar aplicaciones de reconocimiento de

actividades (Radu et al., 2018), incorporando algoritmos de aprendizaje cada vez más sofisticados para detectar y predecir la ejecución de tareas y actividades acorde a los hábitos de uso de los usuarios (*context awareness*).

## 7.5. Conclusiones Finales.

De acuerdo con los resultados obtenidos en las diferentes pruebas realizadas, fue posible validar en múltiples ocasiones el planteamiento de la hipótesis del trabajo, logrando demostrar primero que resulta viable portar un modelo CNN pre-entrenado para detectar objetos, en este caso TinyYOLOv2, en diferentes dispositivos con recursos de cómputo limitados que se basan en muy distintas plataformas. Posteriormente se demostró que es posible modificar y eficientizar dicho modelo CNN para detectar personas aplicando técnicas de cuantización, transferencia de aprendizaje y poda. Con ello se mejoró su rendimiento en comparación con el rendimiento del modelo original (TinyYOLOv2).

Los requerimientos de reducción de clases a sólo una clase para personas, la compresión del modelo CNN en un 50% de espacio de memoria y tolerancia de 5% en la pérdida de precisión fueron cubiertos en su totalidad. El requerimiento de aceleración del 50% con técnicas de adaptación fue cubierto en tres de las nueve plataformas de prueba (iPad, NPU MX con RPi y NPU MX con NUC).

La técnica de transferencia de aprendizaje (TL) solucionó el problema de la reducción de clases satisfactoriamente, mientras que sus aportes en reducción de parámetros y aceleración son mínimos, pero cumple con su propósito de transferir aprendizaje para sólo detectar personas. La cuantización es la técnica que ofreció mayor tasa de compresión, con la capacidad de reducir parámetros en un 75%, prácticamente sin pérdidas de precisión y sin afectar el tiempo de inferencia. La aceleración por hardware con los dispositivos Intel Movidius demostró la mayor tasa de aceleración, no sólo con respecto a las pruebas realizadas en esta tesis, sino también con respecto a los resultados obtenidos con técnicas de adaptación en los trabajos de la revisión de

literatura, aumentando la velocidad de procesamiento en un 345% en promedio, sin afectar a la precisión y sin efectos de compresión sobre el modelo CNN. La técnica de poda propuesta CNN-PSL, logró una compresión del modelo del 54.13% y una aceleración del 45% en promedio. Estos resultados se alcanzaron aplicando dicha técnica solamente a una capa de la red, que representa el 60% de los parámetros, lográndose una reducción del 90% de los parámetros de dicha capa (reducción del 56% del espacio de memoria del modelo CNN completo), mayor incluso a la reducción lograda con la cuantización. Esta técnica presenta potencial para extender esa tasa de compresión al resto de la red a través de las mejoras propuestas en la sección 7.3, y por consiguiente, una mejora en el porcentaje de aceleración.

Con la aplicación de estas técnicas de adaptación, así como con el uso de aceleradores NPU Intel Movidius y el procesador A10 del iPad, fue posible llevar a cabo la ejecución del modelo CNN de detección de personas con cómputo en la frontera y en la niebla, habilitando con ello el procesamiento de imágenes de video en tiempo real para el control de iluminación en un aula inteligente. Evitando con ello los problemas de privacidad, latencia y conectividad que conlleva el uso de servicios de cómputo en la nube, el cual según pruebas realizadas en el laboratorio ejecutando el mismo modelo CNN original, sólo alcanzó a obtener un rendimiento promedio de 0.28 FPS, entre 10 y 20 veces inferior al logrado en las implementaciones desarrolladas mediante cómputo en la frontera y en la niebla.

En esta tesis se logró abordar el campo de las soluciones IoT+DL, tomando parte en el desarrollo de aplicaciones tecnológicas de gran tendencia como es el caso de CECI y considerando las recientes arquitecturas de cómputo en la niebla y en la frontera ofreciendo mayores ventajas que la nube para implementar aplicaciones IoT. Sin embargo, las limitadas capacidades de estas arquitecturas de cómputo presentaron la necesidad de adoptar estrategias para eficientizar la ejecución de modelos DL utilizando opciones de aceleración por hardware (NPU) y la adaptación del modelo CNN mediante las técnicas de poda (*pruning*), transferencia de aprendizaje (*transfer learning*) y cuantización (*quantization*).

# ANEXO 1

## METODOLOGÍA DE REVISIONES SISTEMÁTICAS DE LA LITERATURA

Llamados estudios secundarios, generalmente se usan para agregar estudios primarios y evidencias de la literatura para responder preguntas de investigación. Las revisiones de mapeos sistemáticos (SMR, *Systematic Mapping Review*) son estudios secundarios que tienen como objetivo ofrecer una visión general sobre el estado de la práctica o la investigación en algún tema específico. Por otro lado, las revisiones de literatura sistemática (SLR, *Systematic Literature Review*) se utilizan para investigar profundamente un tema, agregando típicamente evidencias recopiladas por otros estudios para responder preguntas de investigación más elaboradas. A continuación, se presenta el protocolo empleado para el desarrollo de estas revisiones sistemáticas.

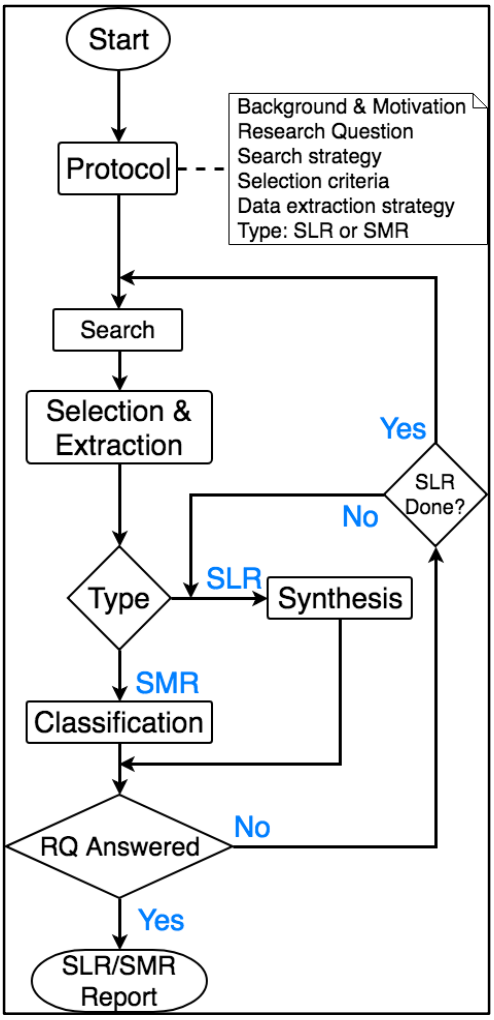


Figura A.1. Diagrama de revisiones sistemáticas.

**Protocolo.**

Un protocolo de revisión define los métodos que se utilizarán para llevar a cabo una revisión sistemática específica. Es necesario un protocolo predefinido para reducir la posibilidad de sesgo, sin un protocolo, es posible que la selección de estudios individuales o el análisis puedan ser impulsados por las expectativas de los investigadores.

- *Base. Fundamento o motivación para el estudio.*
    - Obtención de trabajos relacionados al tema de tesis.

- *Preguntas de investigación que la revisión pretende responder.*
    - ¿Existen trabajos similares al propuesto en esta tesis?

- *Estrategia utilizada para buscar estudios primarios, incluidos los términos de búsqueda y los recursos donde buscar. Los recursos incluyen bibliotecas digitales, revistas específicas y actas de congresos. Un estudio de mapeo inicial puede ayudar a determinar una estrategia apropiada.*
    - Términos de búsqueda: *IoT, Smart Cities, Person Detection, YOLO, CNN, Compression, Acceleration, Edge Computing y Fog Computing.*
    - Recursos: IEEE, ACM y Elsevier.
    - Estudio de mapeo inicial.

- *Criterios de selección de estudios. Éstos se utilizan para determinar qué estudios se incluyen o excluyen de una revisión sistemática. Por lo general, es útil realizar una prueba piloto de los criterios de selección en un subconjunto de estudios primarios.*
    - Trabajos que cubran al menos 4 de 6 áreas que sostienen la tesis.
    - Trabajos que sean firme referencia para alguna etapa de la tesis.

- *Estrategia de extracción de datos. Esto define cómo se obtendrá la información requerida de cada estudio primario. Si los datos requieren manipulación o suposiciones e inferencias, el protocolo debe especificar un proceso de validación apropiado.*
    - Extracción de similitudes con tesis.
    - Análisis y conclusiones propias del trabajo.

- *Calendario del proyecto. Esto debería definir el calendario de revisión.*
    - Del 1 de octubre al 30 de noviembre del 2018.

El resto de la metodología, tanto para la revisión como para el mapeo sistemático se puede visualizar en los diagramas de las Figuras A.2. y A.3.

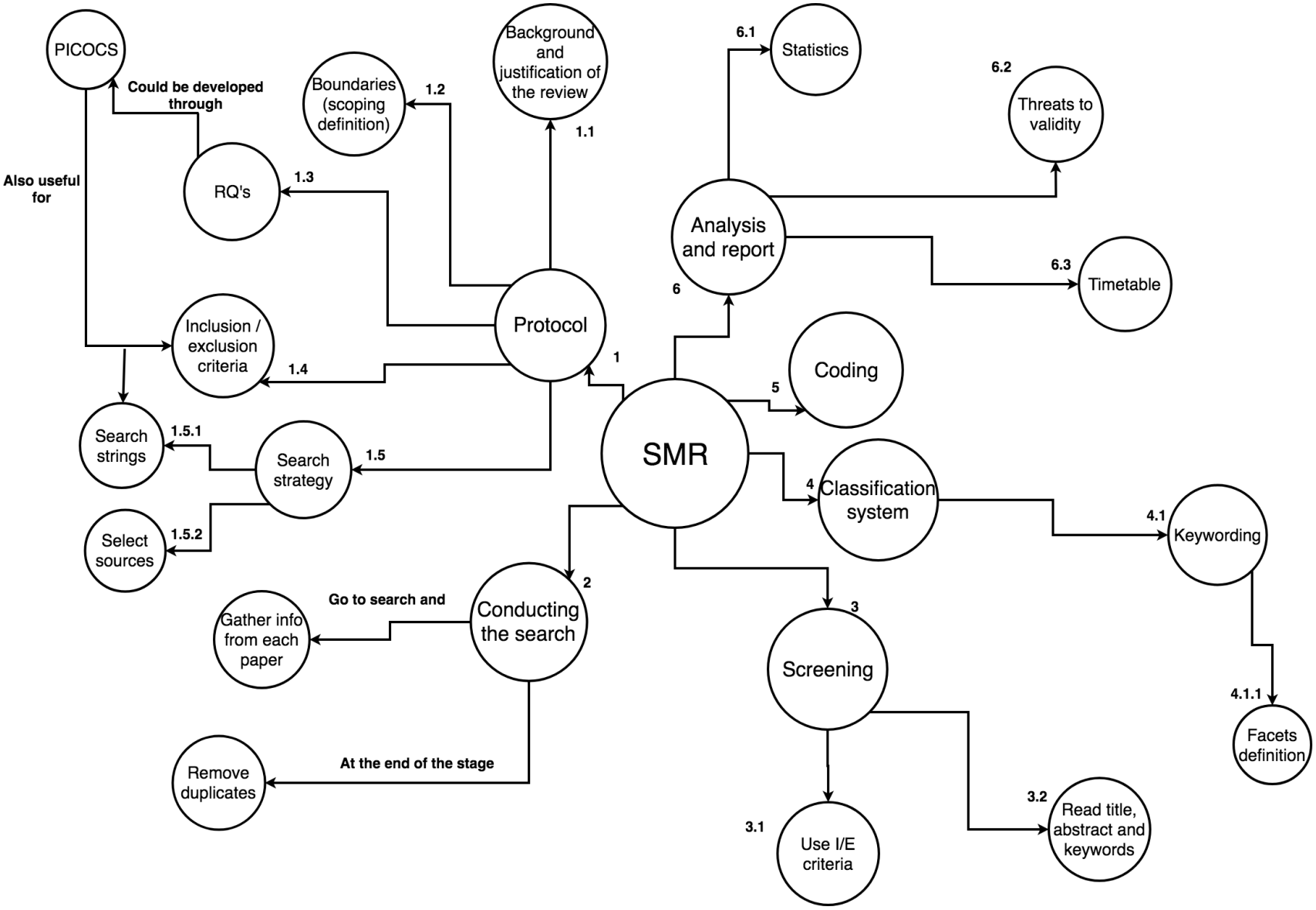


Figura A.2. Diagrama SMR.

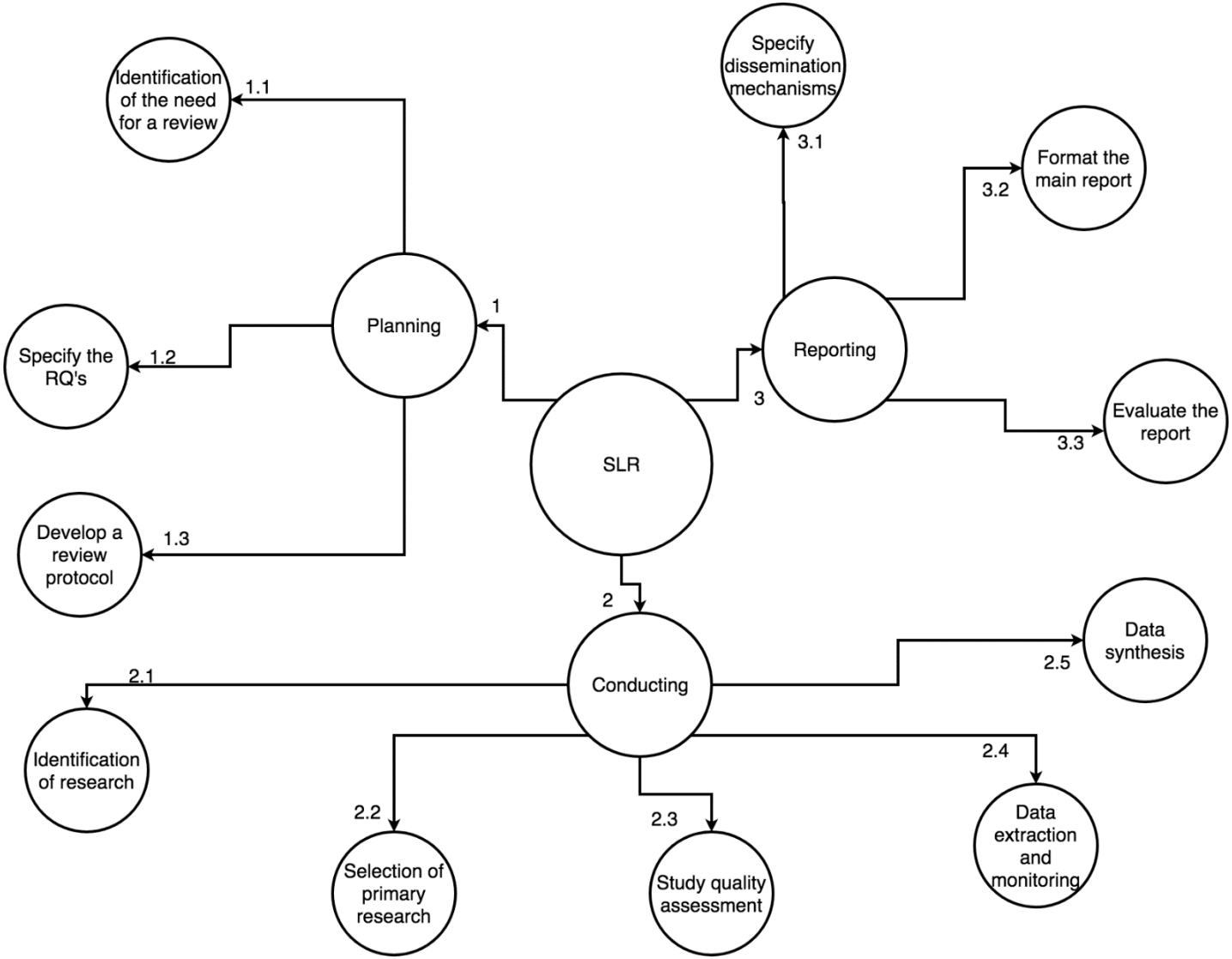


Figura A.3. Diagrama SLR.

# ANEXO 2

## IMPLEMENTACIONES Y PUBLICACIONES REALIZADAS

| **2.1 IMPLEMENTACIONES** | | |
|---|---|---|
| Autores | Prototipo de prueba | LINK |
| AF | Clasificador multiclase Inception, MobileNet y SqueezeNet en iOS | https://github.com/albertochiwas/TinyML |
| PC | Nodos Fog control iluminación en RPi3 y NUC + NPU Intel Movidius | |
| AF | Compilación de arquitectura TinyYOLOv2 en Keras 1.2.2 | |
| AF | Detector de objetos TinyYOLOv2 en iOS | |
| PC AF | TinyYOLOv2 en Ubuntu/Raspbian con Keras (CPU) | |
| PC AF | TinyYOLOv2 en Ubuntu/Raspbian con OpenVino (1 NPU) | |
| PC AF | TinyYOLOv2 en Ubuntu/Raspbian con OpenVino (2 NPU) | |
| AF | TinyYOLOv2 con transferencia aprendizaje y adición capa switch (Keras) | |
| AF MR | TinyYOLOv2 con poda de la red (Keras) | |
| TOTAL | 9 IMPLEMENTACIONES | |
| **2.2 PUBLICACIONES** | | |
| AP<br>AF<br>SA | A Smart Classroom Based On Deep Learning and Osmotic IoT Computing. IEEE CONIITI 2018 (Detroit, MI) | https://ieeexplore.ieee.org/document/8587095 |
| AP<br>AF<br>PC | Smart Classrooms Aided by Deep Neural Networks Inference On Mobile Devices. IEEE EIT 2018 (Bogotá, Colombia) | https://ieeexplore.ieee.org/document/8500260 |
| AP<br>AF<br>PC | Prototipo de un Aula Inteligente aplicando Internet de las Cosas y Modelos de Aprendizaje Profundo. ITESM ICM 2018 (Chihuahua, México) Distinción tercer lugar en sesión de demostración de prototipos. | http://www.chi.itesm.mx/icm/memorias2018/prototipos.pdf |
| MR<br>AF<br>AP | Convolutional Neural Networks Pruning by Switch Layer (trabajo en progreso) | |
| TOTAL | 3 PUBLICACIONES (4 CITAS), 1 TRABAJO EN PROGRESO | |

| **2.3 CONGRESOS Y TALLERES** | |
|---|---|
| Jalisco, México. 2018 North American Center for Collaborative Development Conference (NACCD) | |
| Michigan, EE.UU. 2018 International Conference on Electro/Information Technology (EIT) | |
| Bogotá, Colombia. 2018 Congreso Internacional de Innovación y Tendencias en Ingeniería (CONIITI) | |
| Chihuahua, México. 2018 Congreso Internacional de Investigación Científica Multidisciplinaria (ICM) | |
| Guanajuato, México. 2018 Taller-Escuela de Procesamiento de Imágenes (PI, CIMAT) | |
| Guanajuato, México. Estadía CIMAT del 4 de marzo al 2 de abril de 2019 | |
| Chihuahua, México. 2 Talleres Living Lab (2019) | |
| TOTAL | 4 CONGRESOS, 3 TALLERES, 1 ESTADÍA |
| **2.4 DEMOS (PROTOTIPOS)** | |
| INSTITUTO TECNOLÓGICO DE CHIHUAHUA | 8 DEMOS |
| LIVING LAB DEL CLUSTER DE TI DEL ESTADO | 8 DEMOS |
| EVENTOS, FOROS Y CONGRESOS | 4 DEMOS |
| TOTAL | 20 DEMOS |

Abreviaturas de autores de publicaciones (2.2):

AF: Alejandro Flores
PC: Pablo Cano
AP: Alberto Pacheco
SA: Salvador Almanza
MR: Mariano Rivera

## BIBLIOGRAFÍA


1. Abdi, A. (2017). *General code to convert a trained keras model into an inference tensorflow model: Amir-abdi/keras_to_tensorflow* [Python]. Recuperado de https://github.com/amir-abdi/keras_to_tensorflow (Original work published 2017)
2. Agarwal, S., Terrail, J. O. D., & Jurie, F. (2018). Recent Advances in Object Detection in the Age of Deep Convolutional Neural Networks. *ArXiv:1809.03193 [Cs]*. Recuperado de http://arxiv.org/abs/1809.03193
3. Alderton, M. (2016). Smart classrooms give tech boost to learning. Recuperado el 29 de enero de 2018, de USA TODAY website: https://www.usatoday.com/story/news/2016/07/30/smart-classrooms-give-tech-boost-learning/87699888/
4. Allan, A. (2019). Hands on with the Coral USB Accelerator. Recuperado de Medium website: https://medium.com/@aallan/hands-on-with-the-coral-usb-accelerator-a37fcb323553
5. Allanzelener. (2016). *YAD2K: Yet Another Darknet 2 Keras*. Recuperado de https://github.com/allanzelener/YAD2K
6. Altenberger, F., & Lenz, C. (2018). A Non-Technical Survey on Deep Convolutional Neural Network Architectures. *ArXiv:1803.02129 [Cs]*. Recuperado de http://arxiv.org/abs/1803.02129
7. Alvarez, J. M., & Salzmann, M. (2016). Learning the Number of Neurons in Deep Networks. *ArXiv:1611.06321 [Cs]*. Presentado en 30th Conference on Neural Information Processing Systems (NIPS), Barcelona, Spain. Recuperado de http://arxiv.org/abs/1611.06321
8. Andri, R., Cavigelli, L., Rossi, D., & Benini, L. (2018). Hyperdrive: A Systolically Scalable Binary-Weight CNN Inference Engine for mW IoT End-Nodes. *ArXiv:1804.00623 [Cs, Eess]*. Recuperado de http://arxiv.org/abs/1804.00623

9. Angra, S., & Ahuja, S. (2017). *Machine Learning and its Applications: A Review*. Presentado en 2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC), Chirala, India. https://doi.org/10.1109/ICBDACI.2017.8070809
10. Antona, M., Leonidis, A., Margetis, G., Korozi, M., Ntoa, S., & Stephanidis, C. (2011). A Student-Centric Intelligent Classroom. *Ambient Intelligence*, 248–252. https://doi.org/10.1007/978-3-642-25167-2_33
11. Apple Inc. (2019a). Core ML | Apple Developer Documentation. Recuperado el 25 de agosto de 2019, de Apple Developer website: https://developer.apple.com/documentation/coreml
12. Apple Inc. (2019b). Coremltools 2.0 documentation. Recuperado el 25 de agosto de 2019, de Apple GitHub website: https://apple.github.io/coremltools/
13. Apple Inc. (2019c). Create ML | Apple Developer Documentation. Recuperado el 25 de agosto de 2019, de Apple Developer website: https://developer.apple.com/documentation/createml
14. Apple Inc. (2019d). MLUpdateTask—Core ML | Apple Developer Documentation. Recuperado el 19 de agosto de 2019, de https://developer.apple.com/documentation/coreml/mlupdatetask?changes=latest_beta
15. Arasteh, H., Hosseinnezhad, V., Loia, V., Tommasetti, A., Troisi, O., Shafie-khah, M., & Siano, P. (2016, junio). *Iot-based smart cities: A survey*. 1–6. https://doi.org/10.1109/EEEIC.2016.7555867
16. Ark, T. V. (2015, noviembre 26). 8 Ways Machine Learning Will Improve Education. Recuperado el 29 de enero de 2018, de Getting Smart website: http://www.gettingsmart.com/2015/11/8-ways-machine-learning-will-improve-education/
17. Arlen, T. C. (2018). Understanding the mAP Evaluation Metric for Object Detection. Recuperado de Medium website: https://medium.com/@timothycarlen/understanding-the-map-evaluation-metric-for-object-detection-a07fe6962cf3
18. Athmaja, S., Hanumanthappa, M., & Kavitha, V. (2017). A survey of machine learning algorithms for big data analytics. *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, 1–4. https://doi.org/10.1109/ICIIECS.2017.8276028

19. Badica, C., Brezovan, M., & Badica, A. (2013). An overview of smart home environments: Architectures, technologies and applications. *CEUR Workshop Proceedings*, *1036*, 78–85.
20. Belgaum, M. R., Soomro, S., Alansari, Z., Musa, S., Alam, M., & Su'ud, M. M. (2017). Challenges: Bridge between cloud and IoT. *2017 4th IEEE International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, 1–5. https://doi.org/10.1109/ICETAS.2017.8277844
21. Bengio, Y., Courville, A., & Vincent, P. (2012). Representation Learning: A Review and New Perspectives. *ArXiv:1206.5538 [Cs]*. Recuperado de http://arxiv.org/abs/1206.5538
22. Böhm, T. (2018, agosto 28). A first Introduction to SELUs and why you should start using them as your Activation Functions. Recuperado el 21 de agosto de 2019, de Medium website: https://towardsdatascience.com/gentle-introduction-to-selus-b19943068cd9
23. Bose, S., & Singh, A. K. (2014). *A Survey on Cloud Computing*. Presentado en 2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE), Coimbatore, India. https://doi.org/10.1109/ICGCCEE.2014.6921423
24. Buckley, K. (2017). By 2019, 60% of IT workloads will run in the cloud. Recuperado de 451 Research website: https://451research.com/blog/1910-by-2019,-60-of-it-workloads-will-run-in-the-cloud
25. Buduma, N., & Locascio, N. (2017). *Fundamentals of Deep Learning* (1a ed.). Canada: O'Reilly Media Inc.
26. Bughin, J., Seong, J., Manyika, J., Chui, M., & Joshi, R. (2018). *Notes from the AI frontier Modeling the impact of AI on the world economy*. Recuperado de McKinsey Global Institute website: https://www.mckinsey.com/~/media/McKinsey/Featured%20Insights/Artificial%20Intelligence/Notes%20from%20the%20frontier%20Modeling%20the%20impact%20of%20AI%20on%20the%20world%20economy/MGI-Notes-from-the-AI-frontier-Modeling-the-impact-of-AI-on-the-world-economy-September-2018.ashx
27. Burch, C. (2001). *A survey of machine learning* (p. 42). Recuperado de https://pdfs.semanticscholar.org/31fa/0e7298912e7ccfa18b9d8eb81ca7f5344808.pdf
28. BVLC. (2017). Caffe | Deep Learning Framework. Recuperado el 25 de agosto de 2019, de Berkeley Vision and Learning Center website: https://caffe.berkeleyvision.org/

29. Carroll, K., & Chandramouli, M. (2019, junio). *Scaling IoT to meet enterprise needs. Balancing edge and cloud computing*. Recuperado de https://www2.deloitte.com/insights/us/en/focus/internet-of-things/enterprise-iot-solutions-edge-computing-cloud.html#endnote-5
30. Castrejon, L., Aytar, Y., Vondrick, C., Pirsiavash, H., & Torralba, A. (2016). Learning Aligned Cross-Modal Representations from Weakly Aligned Data. *ArXiv:1607.07295 [Cs]*. Recuperado de http://arxiv.org/abs/1607.07295
31. Cavalcante, E., Cacho, N., Lopes, F., & Batista, T. (2017). *Challenges to the Development of Smart City Systems: A System-of-Systems View*. 244–249. https://doi.org/10.1145/3131151.3131189
32. Charrington, S. (2017). *Artificial Intelligence for Industrial Applications* (p. 28). Recuperado de Cloud Pulse Strategies website: www.cloudpulsestrat.com/IndustrialAI
33. Chatterjee, A. (2018, diciembre 23). What is Neural processing unit (NPU)? Recuperado el 25 de agosto de 2019, de OpenGenus IQ website: https://iq.opengenus.org/neural-processing-unit-npu/
34. Chen, C., Ruan, Y., & Liao, Z. (2018). iOccupancy: An Investigation of Online Occupancy-driven HVAC Control in Campus Classrooms. *Proceedings of the 1st ACM International Workshop on Smart Cities and Fog Computing - CitiFog'18*, 25–28. https://doi.org/10.1145/3277893.3277900
35. Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., & Chen, Y. (2015). Compressing Neural Networks with the Hashing Trick. *ArXiv:1504.04788 [Cs]*. Recuperado de http://arxiv.org/abs/1504.04788
36. Cheng, J., Wu, J., Leng, C., Wang, Y., & Hu, Q. (2018). Quantized CNN: A Unified Approach to Accelerate and Compress Convolutional Networks. *IEEE Transactions on Neural Networks and Learning Systems*, *29*(10), 4730–4743. https://doi.org/10.1109/TNNLS.2017.2774288
37. Cheng, Y., Wang, D., Zhou, P., & Zhang, T. (2017). A Survey of Model Compression and Acceleration for Deep Neural Networks. *IEEE SIGNAL PROCESSING MAGAZINE, SPECIAL ISSUE ON DEEP LEARNING FOR IMAGE UNDERSTANDING*. Recuperado de http://arxiv.org/abs/1710.09282

38. Chevalier, G. (2019). What are the advantages of using Leaky Rectified Linear Units (Leaky ReLU) over normal ReLU in deep learning? Recuperado el 21 de agosto de 2019, de Quora website: https://www.quora.com/What-are-the-advantages-of-using-Leaky-Rectified-Linear-Units-Leaky-ReLU-over-normal-ReLU-in-deep-learning
39. Chollet, F. (2017). *Deep Learning with Python* (1a ed.). Manning Publications.
40. Chollet, F. (2019). Keras: The Python Deep Learning library. Recuperado el 25 de agosto de 2019, de Keras Documentation website: https://keras.io/
41. Christodoulidis, S., Anthimopoulos, M., Ebner, L., Christe, A., & Mougiakakou, S. (2017). Multisource Transfer Learning With Convolutional Neural Networks for Lung Pattern Analysis. *IEEE Journal of Biomedical and Health Informatics*, *21*(1), 76–84. https://doi.org/10.1109/JBHI.2016.2636929
42. Chui, M., Manyika, J., Miremadi, M., Henke, N., Chung, R., Nel, P., & Malhotra, S. (2018). *Notes from the AI frontier. Insights from hundreds of use cases*. Recuperado de McKinsey Global Institute website: http://www.mckinsey.com/mgi
43. Ciurana, A. (2019). Fix bug for save nested models with shared parameters. Recuperado de GitHub website: https://github.com/keras-team/keras/pull/11847
44. Columbus, L. (2018). 83% Of Enterprise Workloads Will Be In The Cloud By 2020. Recuperado de Forbes website: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#1a46c14d6261
45. Copeland, M. (2016). What's the Difference Between Deep Learning Training and Inference? Recuperado de NVIDIA website: https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/
46. Da Silva, W. M., Tomas, G. H. R. P., Dias, K. L., Alvaro, A., Afonso, R. A., & Garcia, V. C. (2013). *Smart cities software architectures: A survey*. 6.
47. Dai, J., Li, Y., He, K., & Sun, J. (2016). *R-FCN: Object Detection via Region-based Fully Convolutional Networks*. Recuperado de https://arxiv.org/abs/1605.06409v2
48. Dalal, N., & Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, *1*, 886–893. https://doi.org/10.1109/CVPR.2005.177

49. Das, K., Behera, R. N., & Tech, B. (2017). A Survey on Machine Learning: Concept, Algorithms and Applications. *International Journal of Innovative Research in Computer and Communication Engineering*, *5*(2), 9. https://doi.org/10.15680/IJIRCCE.2017. 0502001
50. De Paola, A., Ortolani, M., Lo Re, G., Anastasi, G., & Das, S. K. (2014). Intelligent Management Systems for Energy Efficiency in Buildings: A Survey. *ACM Computing Surveys*, *47*(1), 1–38. https://doi.org/10.1145/2611779
51. Deloitte. (2018). *Industry 4.0: Are you ready?* Recuperado de Deloitte website: https://www2.deloitte.com/content/dam/insights/us/collections/issue-22/DI_Deloitte-Review-22.pdf
52. Deng, L., & Yu, D. (2014). Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing*, *7*(3–4), 197–387. https://doi.org/10.1561/2000000039
53. Dey, S., & Mukherjee, A. (2018). Implementing Deep Learning and Inferencing on Fog and Edge Computing Systems. *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 818–823. https://doi.org/10.1109/PERCOMW.2018.8480168
54. Dickson, B. (2017, marzo 13). How Artificial Intelligence enhances education. Recuperado el 29 de enero de 2018, de The Next Web website: https://thenextweb.com/artificial-intelligence/2017/03/13/how-artificial-intelligence-enhances-education/
55. Dong, P., & Wang, W. (2016). Better region proposals for pedestrian detection with R-CNN. *2016 Visual Communications and Image Processing (VCIP)*, 1–4. https://doi.org/10.1109/VCIP.2016.7805452
56. Duangenquan. (2017). *YOLOv2 for Intel/Movidius Neural Compute Stick (NCS)* [GitHub]. Recuperado de https://github.com/duangenquan/YoloV2NCS
57. Experiencor. (2017). *YOLOv2 in Keras and Applications* [GitHub]. Recuperado de https://github.com/experiencor/keras-yolo2
58. Fakhruddin, H. (2018). Machine Learning in 2019: Tracing The Artificial Intelligence Growth Path. Recuperado de Teks mobile Teknowledge Software website: https://teks.co.in/site/blog/machine-learning-in-2019-tracing-the-artificial-intelligence-growth-path/

59. Fan, C. (2015). *Survey of Convolutional Neural Network* (p. 22). Recuperado de http://homes.sice.indiana.edu/fan6/docs/cnn_survey.pdf
60. Foote, K. D. (2017). *A Brief History of Deep Learning*. Recuperado de Dataversity website: https://www.dataversity.net/brief-history-deep-learning/
61. Ford, R., Pritoni, M., Sanguinetti, A., & Karlin, B. (2017). Categories and functionality of smart home technology for energy management. *Building and Environment*, *123*, 543–554. https://doi.org/10.1016/j.buildenv.2017.07.020
62. Frankle, J., & Carbin, M. (2019). The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. *ICLR*. Recuperado de http://arxiv.org/abs/1803.03635
63. Franklin, D., Flachsbart, J., & Hammond, K. (1999). The intelligent classroom. *IEEE Intelligent Systems and their Applications*, *14*(5), 2–5.
64. Fu, C.-Y., Liu, W., Ranga, A., Tyagi, A., & Berg, A. C. (2017). *DSSD : Deconvolutional Single Shot Detector* (p. 11). Recuperado de https://arxiv.org/abs/1701.06659
65. Ge, S. (2018). Efficient Deep Learning in Network Compression and Acceleration. En V. Asadpour (Ed.), *Digital Systems*. https://doi.org/10.5772/intechopen.79562
66. Geissbauer, R., Lübben, E., Schrauf, S., & Pillsbury, S. (2018). *How industry leaders build integrated operations ecosystems to deliver end-to-end customer solutions*. Recuperado de PricewaterhouseCoopers website: https://www.strategyand.pwc.com/media/file/Global-Digital-Operations-Study_Digital-Champions.pdf
67. GlobeNewswire. (2018). *Global Smart Cities Planning Analysis and Trends Report 2018*. Recuperado de GlobeNewswire website: https://www.globenewswire.com/news-release/2018/12/04/1661882/0/en/Global-Smart-Cities-Planning-Analysis-and-Trends-Report-2018.html
68. Google Brain. (2019). TensorFlow. Recuperado el 25 de agosto de 2019, de TensorFlow website: https://www.tensorflow.org/
69. GSMA Intelligence. (2018). *The Mobile Economy* (p. 60). Recuperado de https://www.gsma.com/mobileeconomy/wp-content/uploads/2018/02/The-Mobile-Economy-Global-2018.pdf

70. Gupta, R. (2019). Pre-trained models don't work in float16 mode · Issue #11989 · keras-team/keras. Recuperado el 14 de agosto de 2019, de GitHub website: https://github.com/keras-team/keras/issues/11989
71. Han, S., Mao, H., & Dally, W. J. (2016). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *ICLR*. Recuperado de http://arxiv.org/abs/1510.00149
72. Han, S., Pool, J., Tran, J., & Dally, W. J. (2015). Learning both Weights and Connections for Efficient Neural Networks. *ArXiv:1506.02626 [Cs]*. Recuperado de http://arxiv.org/abs/1506.02626
73. Hao, K. (2019). *We analyzed 16,625 papers to figure out where AI is headed next*. Recuperado de MIT Technology Review website: https://www.technologyreview.com/s/612768/we-analyzed-16625-papers-to-figure-out-where-ai-is-headed-next/
74. Hao, X., Zhang, G., & Ma, S. (2016). Deep Learning. *International Journal of Semantic Computing*, *10*(03), 417–439. https://doi.org/10.1142/S1793351X16500045
75. Harari, Y. N. (2014). *Homo Deus. Breve historia del mañana* (1a ed.). Penguin Random House Grupo Editorial.
76. Henderson, P., & Ferrari, V. (2016). End-to-end training of object class detectors for mean average precision. *ArXiv:1607.03476 [Cs]*. Recuperado de http://arxiv.org/abs/1607.03476
77. Hollance. (2017). *Tiny YOLO for iOS implemented using CoreML but also using the new MPS graph API.* [GitHub]. Recuperado de https://github.com/hollance/YOLO-CoreML-MPSNNGraph
78. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv:1704.04861 [Cs]*. Recuperado de http://arxiv.org/abs/1704.04861
79. Hu, H., Peng, R., Tai, Y.-W., & Tang, C.-K. (2016). Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. *ArXiv:1607.03250 [Cs]*. Recuperado de http://arxiv.org/abs/1607.03250

80. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2018). Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research*, 1–30.
81. Hui, T. K. L., Sherratt, R. S., & Sánchez, D. D. (2016). Major requirements for building Smart Homes in Smart Cities based on Internet of Things technologies. *Future Generation Computer Systems*, *76*, 358–369. https://doi.org/10.1016/j.future.2016.10.026
82. Hyodo, K. (2018). OpenVINO-YoloV3: YoloV3/tiny-YoloV3+RaspberryPi3/Ubuntu LaptopPC+NCS/NCS2+USB Camera+Python+OpenVINO. Recuperado el 21 de agosto de 2019, de GitHub website: https://github.com/PINTO0309/OpenVINO-YoloV3
83. Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *ICLR*. Recuperado de http://arxiv.org/abs/1602.07360
84. Intel Corp. (2019a). Converting a TensorFlow* Model—OpenVINO Toolkit. Recuperado el 21 de agosto de 2019, de OpenVinoToolKit website: https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html
85. Intel Corp. (2019b). Intel® Distribution of OpenVINO™ Toolkit. Recuperado el 25 de agosto de 2019, de Intel website: https://software.intel.com/en-us/openvino-toolkit
86. Intel Corp. (2019c). Intel® Neural Compute Stick 2. Recuperado el 25 de agosto de 2019, de https://software.intel.com/en-us/neural-compute-stick
87. Intel Corp. (2019d). Intel Movidius Neural Compute SDK. Recuperado el 25 de agosto de 2019, de Movidius GitHub website: https://movidius.github.io/ncsdk/
88. Jain, A. K., & Mao, J. (1996). *Artificial Neural Networks: A Tutorial*. Recuperado de https://csc.lsu.edu/~jianhua/nn.pdf
89. Jensen, M. B., Gade, R., & Moeslund, T. B. (2018). Swimming Pool Occupancy Analysis using Deep Learning on Low Quality Video. *Proceedings of the 1st International Workshop on Multimedia Content Analysis in Sports - MMSports'18*, 67–73. https://doi.org/10.1145/3265845.3265846
90. Kang, B., & Choo, H. (2018). An experimental study of a reliable IoT gateway. *ICT Express*, *4*(3), 130–133. https://doi.org/10.1016/j.icte.2017.04.002

91. Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., & Shin, D. (2015). Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. *ICLR*. Recuperado de http://arxiv.org/abs/1511.06530
92. Kogan, N., & Lee, K. J. (2014). Exploratory Research on the Success Factors and Challenges of Smart City Projects. *Asia Pacific Journal of Information Systems*, *24*(2), 141–189. https://doi.org/10.14329/apjis.2014.24.2.141
93. Kolecki, J. C. (2002). An Introduction to Tensors for Students of Physics and Engineering. *National Aeronautics and Space Administration*, 29.
94. Le Cun, Y., Denker, J. S., & Solla, S. A. (1989). Optimal Brain Damage. *NIPS*, *2*, 598–605.
95. Lee, K.-F. (2018). *AI Superpowers: China, Silicon Valley, and the New World Order* (1a ed.). Houghton Mifflin Harcourt.
96. Lee, N., Ajanthan, T., & Torr, P. H. S. (2019). SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. *ICLR*, 15.
97. Li, Hao, Kadav, A., Durdanovic, I., Samet, H., & Graf, H. P. (2017). Pruning Filters for Efficient ConvNets. *ICLR*. Recuperado de http://arxiv.org/abs/1608.08710
98. Li, He, Ota, K., & Dong, M. (2018). Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Network*, *32*(1), 96–101. https://doi.org/10.1109/MNET.2018.1700202
99. Li, R., Wang, Y., Liang, F., Qin, H., Yan, J., & Fan, R. (2019). Fully Quantized Network for Object Detection. *CVPR*, 10.
100. Li, Y., Li, J., Lin, W., & Li, J. (2018). *Tiny-DSOD: Lightweight Object Detection for Resource-Restricted Usages*. 12.
101. Li, Z., & Hoiem, D. (2016). Learning without Forgetting. *ArXiv:1606.09282 [Cs, Stat]*. Recuperado de http://arxiv.org/abs/1606.09282
102. Linthicum, D. S. (2017). Connecting Fog and Cloud Computing. *IEEE Cloud Computing*.
103. Liu, K., Li, Y., Xu, N., & Natarajan, P. (2018). Learn to Combine Modalities in Multimodal Deep Learning. *ArXiv:1805.11730 [Cs, Stat]*. Recuperado de http://arxiv.org/abs/1805.11730

104. Liu, Wei, Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single Shot MultiBox Detector. En B. Leibe, J. Matas, N. Sebe, & M. Welling (Eds.), *Computer Vision – ECCV 2016* (Vol. 9905, pp. 21–37). https://doi.org/10.1007/978-3-319-46448-0_2

105. Liu, Weibo, Wang, Z., Liu, X., Zeng, N., Liu, Y., & Alsaadi, F. E. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing*, *234*, 11–26. https://doi.org/10.1016/j.neucom.2016.12.038

106. Luo, J.-H., Wu, J., & Lin, W. (2017, julio 19). *ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression*. Presentado en 2017 IEEE International Conference on Computer Vision (ICCV), Venice, Italy. https://doi.org/10.1109/ICCV.2017.541

107. Ma, D. (2017). *YOLOv3 with Core ML* [GitHub]. Recuperado de https://github.com/Ma-Dan/YOLOv3-CoreML

108. Ma, Y., Cao, Y., Vrudhula, S., & Seo, J. (2018). Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *26*(7), 1354–1367. https://doi.org/10.1109/TVLSI.2018.2815603

109. Maheshwari, V. K. (2016). THE CONCEPT OF SMART CLASSROOM. Recuperado el 29 de enero de 2018, de http://www.vkmaheshwari.com/WP/?p=2352

110. Maji, P., & Mullins, R. D. (2017). *ADaPT: Optimizing CNN inference on IoT and mobile devices using approximately separable 1-D kernels*. 11.

111. Manic, M., Amarasinghe, K., Rodriguez-Andina, J. J., & Rieger, C. (2016). Intelligent Buildings of the Future: Cyberaware, Deep Learning Powered, and Human Interacting. *IEEE Industrial Electronics Magazine*, *10*(4), 32–49. https://doi.org/10.1109/MIE.2016.2615575

112. Martinho-Truswell, E., Miller, H., Nti Asare, I., Petheram, A., Stirling, R., Gómez Mont, C., & Martinez, C. (2018). *Towards an AI strategy in Mexico*. Recuperado de Oxford Insights & C Minds website: http://go.wizeline.com/rs/571-SRN-279/images/Towards-an-AI-strategy-in-Mexico.pdf

113. McCulloch, W. S., & Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133.

114. Merrit, R. (2019). TinyML Sees Big Hopes for Small AI. *EE|Times*. Recuperado de https://www.eetimes.com/document.asp?doc_id=1334484#

115. Meulen, R. (2018). What Edge Computing Means for Infrastructure and Operations Leaders. Recuperado el 24 de agosto de 2019, de Gartner website: //www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders/

116. Million, E. (2007). *The Hadamard Product*. Recuperado de http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf

117. Mittal, S. (2019). A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *Journal of Systems Architecture*, *97*, 428–442. https://doi.org/10.1016/j.sysarc.2019.01.011

118. Miyashita, D., Lee, E. H., & Murmann, B. (2016). Convolutional Neural Networks using Logarithmic Data Representation. *ArXiv:1603.01025 [Cs]*. Recuperado de http://arxiv.org/abs/1603.01025

119. Mohammadi, M., Al-Fuqaha, A., Sorour, S., & Guizani, M. (2018). Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Communications Surveys & Tutorials*. Recuperado de http://arxiv.org/abs/1712.04301

120. Molchanov, P., Tyree, S., Karras, T., Aila, T., & Kautz, J. (2017). Pruning Convolutional Neural Networks for Resource Efficient Inference. *ICLR*. Recuperado de http://arxiv.org/abs/1611.06440

121. Motamedi, M., Fong, D., & Ghiasi, S. (2016). Fast and Energy-Efficient CNN Inference on IoT Devices. *ArXiv:1611.07151 [Cs]*. Recuperado de http://arxiv.org/abs/1611.07151

122. Motamedi, M., Fong, D., & Ghiasi, S. (2017). Machine Intelligence on Resource-Constrained IoT Devices: The Case of Thread Granularity Optimization for CNN Inference. *ACM Transactions on Embedded Computing Systems*, *16*(5s), 1–19. https://doi.org/10.1145/3126555

123. Mujthaba, G. M., Abdalla, A., & Manjur, K. (2018). Cloud servers and fog or edge computing with their limitations & challenges. *Journal of Computer Hardware Engineering*, *1*. https://doi.org/10.63019/jche.v1i2.577

124. Murthy. (2018). Problem converting Keras model to Tensorflow-Lite using TocoConverter.from_keras_model_file. Recuperado el 15 de agosto de 2019, de Stack Overflow website: https://stackoverflow.com/questions/52735895/problem-converting-keras-model-to-tensorflow-lite-using-tococonverter-from-keras

125. Naha, R. K., Garg, S., Georgakopoulos, D., Jayaraman, P. P., Gao, L., Xiang, Y., & Ranjan, R. (2018). Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions. *ArXiv:1807.00976 [Cs]*. Recuperado de http://arxiv.org/abs/1807.00976

126. Nakahara, H., Yonekawa, H., & Sato, S. (2017). An object detector based on multiscale sliding window search using a fully pipelined binarized CNN on an FPGA. *2017 International Conference on Field Programmable Technology (ICFPT)*, 168–175. https://doi.org/10.1109/FPT.2017.8280135

127. Neklyudov, K., Molchanov, D., Ashukha, A., & Vetrov, D. (2017). Structured Bayesian Pruning via Log-Normal Multiplicative Noise. *NIPS*. Presentado en Long Beach, CA, USA. Recuperado de http://arxiv.org/abs/1705.07283

128. Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., & Ng, A. Y. (2011). Multimodal Deep Learning. *ICML*, 8.

129. Ni, Z., Chen, J., Sang, N., Gao, C., & Liu, L. (2018). *Light YOLO for High-Speed Gesture Recognition*. 5. https://doi.org/10.1109/ICIP.2018.8451766

130. Nikouei, S. Y., Chen, Y., Song, S., Xu, R., Choi, B.-Y., & Faughnan, T. R. (2018). Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN. *2018 IEEE International Conference on Edge Computing (EDGE)*, 125–129. https://doi.org/10.1109/EDGE.2018.00025

131. Nikouei, S. Y., Xu, R., & Chen, Y. (2019). *Fog and Edge Computing: Principles and Paradigms* (1a ed.). Wiley.

132. O'Keeffe, S., & Villing, R. (2018). *Evaluating Pruned Object Detection Networks for Real-Time Robot Vision.pdf*. Presentado en Torres Vedras, Portugal. https://doi.org/10.1109/ICARSC.2018.8374166

133. Özer, C., Gürkan, F., & Günsel, B. (2018). Object tracking by deep object detectors and particle filtering. *2018 26th Signal Processing and Communications Applications Conference (SIU)*, 1–4. https://doi.org/10.1109/SIU.2018.8404622

134. Pacheco, A., Cano, P., Flores, E., Trujillo, E., & Marquez, P. (2018). A Smart Classroom Based on Deep Learning and Osmotic IoT Computing. *2018 Congreso Internacional de Innovación y Tendencias En Ingeniería (CONIITI)*, 1–5. https://doi.org/10.1109/CONIITI.2018.8587095

135. Pacheco, A., Flores, E., & Cano, P. (2018). *Aula Inteligente con Interfaz Natural basada en Realidad Aumentada y Aprendizaje Automático*. Reporte técnico de proyecto financiado por el Tecnológico Nacional de México (TecNM), IT Chihuahua.

136. Pacheco, A., Flores, E., Cano, P., & Tena, A. (2018). *Prototipo de un Aula Inteligente aplicando Internet de las Cosas y Modelos de Aprendizaje Profundo*. Presentado en 10° Congreso Internacional Investigación Científica Multidisciplinaria, Chihuahua, México.

137. Pacheco, A., Flores, E., Sanchez, R., & Almanza-Garcia, S. (2018). Smart Classrooms Aided by Deep Neural Networks Inference on Mobile Devices. *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 0605–0609. https://doi.org/10.1109/EIT.2018.8500260

138. Pan, S. J., & Yang, Q. (2010). A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*(10), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

139. Pemberton, C. (2017, octubre). 3 AI Trends for Enterprise Computing. Recuperado de https://www.gartner.com/smarterwithgartner/3-ai-trends-for-enterprise-computing/

140. Pilipovic, R., Bulic, P., & Risojevic, V. (2018). Compression of convolutional neural networks: A short survey. *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 1–6. https://doi.org/10.1109/INFOTEH.2018.8345545

141. Prabhu, C. S. R. (2019). *Fog Computing, Deep Learning and Big Data Analytics-Research Directions*. Recuperado de https://www.springer.com/gp/book/9789811332081#aboutAuthors

142. Price, M. (2018). How to convert all layers of a pretrained Keras model to a different dtype (from float32 to float16)? Recuperado el 14 de agosto de 2019, de Stack Overflow website: https://stackoverflow.com/questions/47895494/how-to-convert-all-layers-of-a-pretrained-keras-model-to-a-different-dtype-from

143. PRNewswire. (2019). *Edge Computing Market—Growth, Trends, and Forecast (2019-2024)*. Recuperado de PRNewswire website: https://www.prnewswire.com/news-releases/edge-computing-market---growth-trends-and-forecast-2019---2024-300872771.html

144. PwC. (2016). *Industry 4.0: Building the digital enterprise*. Recuperado de PricewaterhouseCoopers website: https://www.pwc.com/gx/en/industries/industries-4.0/landing-page/industry-4.0-building-your-digital-enterprise-april-2016.pdf

145. Radu, V., Tong, C., Bhattacharya, S., Lane, N. D., Mascolo, C., Marina, M. K., & Kawsar, F. (2018). Multimodal Deep Learning for Activity and Context Recognition. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, *1*(4), 1–27. https://doi.org/10.1145/3161174

146. Raschka, S., & Mirjalili, V. (2017). *Python Machine Learning, Second Edition* (Edición: 2nd). Birmingham Mumbai: Packt Publishing.

147. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *ArXiv:1506.02640 [Cs]*. Recuperado de http://arxiv.org/abs/1506.02640

148. Redmon, J., & Farhadi, A. (2016). YOLO9000: Better, Faster, Stronger. *ArXiv:1612.08242 [Cs]*. Recuperado de http://arxiv.org/abs/1612.08242

149. Redmon, J., & Farhadi, A. (2018). *YOLOv3: An Incremental Improvement*. Recuperado de https://pjreddie.com/media/files/papers/YOLOv3.pdf

150. Ren, S., He, K., Girshick, R., & Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(6), 1137–1149. https://doi.org/10.1109/TPAMI.2016.2577031

151. Rifkin, R. (2008). *Multiclass Classification*. Recuperado de http://www.mit.edu/~9.520/spring09/Classes/multiclass.pdf

152. Rivera, M., Flores, E., & Pacheco, A. (2019). Convolutional Neural Networks Pruning by Switch Layer. *Trabajo en progreso*.

153. Rosenblatt, F. (1957). *The Perceptron: A Perceiving and Recognizing Automation*. Recuperado de https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf

154. Saiyeda, A., & Mir, M. A. (2017). Cloud computing for deep learning analytics:A survey of current trends and challenges. *International Journal of Advanced Research in Computer Science*, 5.

155. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. *ArXiv:1801.04381 [Cs]*. Recuperado de http://arxiv.org/abs/1801.04381

156. Sarkar, D., Bali, R., & Ghosh, T. (2018). *Transfer Learning with Python*. Packt Publishing.

157. Satyanarayanan, M. (2017). The Emergence of Edge Computing. *Computer*, *50*(1), 30–39. https://doi.org/10.1109/MC.2017.9

158. Sharma, A. (2017, octubre 15). Confusion Matrix in Machine Learning. Recuperado el 14 de agosto de 2019, de GeeksforGeeks website: https://www.geeksforgeeks.org/confusion-matrix-machine-learning/

159. Shen, S. (2017). *YOLO with Core ML* [GitHub]. Recuperado de https://github.com/syshen/YOLO-CoreML

160. Shen, Z., Liu, Z., Li, J., Jiang, Y.-G., Chen, Y., & Xue, X. (2017). DSOD: Learning Deeply Supervised Object Detectors from Scratch. *ArXiv:1708.01241 [Cs]*. Recuperado de http://arxiv.org/abs/1708.01241

161. Shoham, Y., Perrault, R., Brynjolfsson, E., Clark, J., Manyika, J., Niebles, J. C., … Bauer, Z. (2018). *The AI Index 2018 Annual Report*. AI Index Steering Committee, Human-Centered AI Initiative, Stanford University, Stanford, CA.

162. Sikr. (2018). Quantize a Keras neural network model. Recuperado el 15 de agosto de 2019, de Stack Overflow website: https://stackoverflow.com/questions/52259343/quantize-a-keras-neural-network-model

163. Simonelli, A. (2017). *Tiny YOLOv2 in Tensorflow made simple*. Recuperado de https://github.com/simo23/tinyYOLOv2

164. Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv:1409.1556 [Cs]*. Recuperado de http://arxiv.org/abs/1409.1556

165. Singh, D. (2014). A Critical Conceptual Analysis of Definitions of Artificial Intelligence as Applicable to Computer Engineering. *IOSR Journal of Computer Engineering*, *16*(2), 09–13. https://doi.org/10.9790/0661-16210913

166. Singh, P., Verma, V. K., Rai, P., & Namboodiri, V. P. (2019). Play and Prune: Adaptive Filter Pruning for Deep Model Compression. *ArXiv:1905.04446 [Cs]*. Recuperado de http://arxiv.org/abs/1905.04446

167. SmartBoards. (2018). Smart Boards | Are you sure which smart board is right for you? Recuperado el 29 de enero de 2018, de http://www.smart-boards.com/?gclid=Cj0KCQiA4bzSBRDOARIsAHJ1UO4hLzjiBASE4Uo7WV5DXQaWtO5dJBy8sNS2bzmcazlIFzR2PPpsdqcaAmq4EALw_wcB

168. SmartTech. (2011). SMART Classroom Suite. Recuperado el 29 de enero de 2018, de SMART Technologies website: https://support.smarttech.com/software/other-software/smart-classroom-suite

169. SmartWay. (2015, febrero 18). Smart Classroom. Recuperado el 29 de enero de 2018, de SmartWay website: http://smartway-me.com/services/smart-classroom

170. Song, H., Srinvasan, R., Sookoor, T., & Jeschke, S. (2017). *Smart cities: Foundations, principles, and applications* (1a ed.). John wiley & sons inc.

171. Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, *105*(12), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

172. Szemenyei, M., & Estivill-Castro, V. (2019). *ROBO: Robust, Fully Neural Object Detection for Robot Soccer*. 14.

173. Vermesan, O., & Friess, P. (2014). *Internet of things: Converging technologies for smart environments and integrated ecosystems*. River Publishers.

174. Vincent, J. (2017). This little USB stick is designed to make AI plug-and-play. Recuperado de The Verge website: https://www.theverge.com/2017/7/20/16002682/movidius-ai-neural-compute-stick-intel

175. Wang, H., Zhang, Q., Wang, Y., Lu, Y., & Hu, H. (2019). Structured Pruning for Efficient ConvNets via Incremental Regularization. *ArXiv:1804.09461 [Cs, Stat]*. Recuperado de http://arxiv.org/abs/1804.09461

176. Wang, J., Bohn, T., & Ling, C. (2018). Pelee: A Real-Time Object Detection System on Mobile Devices. *NIPS*, 10. Montreal, Canada.

177. Wei Yang, L., & Yen Su, C. (2018). Low-Cost CNN Design for Intelligent Surveillance System. *2018 International Conference on System Science and Engineering (ICSSE)*, 1–4. https://doi.org/10.1109/ICSSE.2018.8520133

178. WikiChip. (2019). Neural Processor. Recuperado el 25 de agosto de 2019, de WikiChip website: https://en.wikichip.org/wiki/neural_processor

179. Wilson, C., Hargreaves, T., & Hauxwell-Baldwin, R. (2017). Benefits and risks of smart home technologies. *Energy Policy*, *103*, 72–83. https://doi.org/10.1016/j.enpol.2016.12.047

180. Winer, L. R., & Cooperstock, J. (2002). The "intelligent classroom": Changing teaching and learning with an evolving technological environment. *Computers & Education*, *38*(1–3), 253–266.

181. Woetzel, J., Remes, J., Boland, B., Lv, K., Sinha, S., Strube, G., … von der Tann, V. (2018). *SMART CITIES: DIGITAL SOLUTIONS FOR A MORE LIVABLE FUTURE*. McKinsey Global Institute.

182. Wu, J., Leng, C., Wang, Y., Hu, Q., & Cheng, J. (2016). Quantized Convolutional Neural Networks for Mobile Devices. *CVPR*, 4820–4828. https://doi.org/10.1109/CVPR.2016.521

183. Xiang, W., Zhang, D.-Q., Yu, H., & Athitsos, V. (2018). Context-Aware Single-Shot Detector. *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 1784–1793. https://doi.org/10.1109/WACV.2018.00198

184. Xu, B., Wang, N., Chen, T., & Li, M. (2015). Empirical Evaluation of Rectified Activations in Convolutional Network. *ArXiv:1505.00853 [Cs, Stat]*. Recuperado de http://arxiv.org/abs/1505.00853

185. Yao, S., Zhao, Y., Zhang, A., Hu, S., Shao, H., Zhang, C., … Abdelzaher, T. (2018). Deep Learning for the Internet of Things. *Computer*, *51*(5), 32–41. https://doi.org/10.1109/MC.2018.2381131

186. Yi, S., Li, C., & Li, Q. (2015). A Survey of Fog Computing: Concepts, Applications and Issues. *Proceedings of the 2015 Workshop on Mobile Big Data - Mobidata '15*, 37–42. https://doi.org/10.1145/2757384.2757397

187. Zeiler, M. D., & Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. En D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *Computer Vision – ECCV 2014* (Vol. 8689, pp. 818–833). https://doi.org/10.1007/978-3-319-10590-1_53

188. Zhao, Y., Wang, W., Li, Y., Meixner, C. C., Tornatore, M., & Zhang, J. (2019). Edge Computing and Networking: A Survey on Infrastructures and Applications. *IEEE Access*, 1–1. https://doi.org/10.1109/ACCESS.2019.2927538

189. Zhao, Z., Barijough, K. M., & Gerstlauer, A. (2018). DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(11), 2348–2359. https://doi.org/10.1109/TCAD.2018.2858384

190. Zhao, Z.-Q., Zheng, P., Xu, S.-T., & Wu, X. (2019). Object Detection With Deep Learning: A Review. *IEEE Transactions on Neural Networks and Learning Systems*, 1–21. https://doi.org/10.1109/TNNLS.2018.2876865

191. Zhuang, Z., Tan, M., Zhuang, B., Liu, J., Guo, Y., Wu, Q., … Zhu, J. (2018). Discrimination-aware Channel Pruning for Deep Neural Networks. *NIPS*, 12.

192. Zion Market Research. (2018). *Industry 4.0 Market by Technology and by Vertical: Global Industry Perspective, Comprehensive Analysis, and Forecast, 2017-2024*. Recuperado de Zion Market Research website: https://www.globenewswire.com/news-release/2018/10/17/1622652/0/en/Global-Industry-4-0-Market-Will-Reach-USD-155-30-Billion-By-2024-Zion-Market-Research.html

193. Zocca, V., Spacagna, G., Slater, D., & Roelants, P. (2017). *Python Deep Learning*. Packt Publishing Ltd.