

Universidad Nacional de La Plata
Facultad de Informática



**Detección y Clasificación de Enfermedades en el Tomate
Mediante Deep Learning y Computer Vision**

Tesis presentada para obtener el grado de
Magister en Ingeniería de Software

Autor: Sergio Hernán Valenzuela Cámara

Director de Tesis: PhD. Alejandro Fernández

Co-Director de Tesis: MSc. Diego Alberto Aracena Pizarro

La Plata – Argentina
Septiembre de 2021

... A mis hijos Favio Andrés, Mauricio Elías y Naomi Andrea, a mi esposa, compañera y cómplice de la vida Indira Elva, y a toda mi familia por su permanente apoyo, cariño, comprensión e infinita paciencia que tuvieron para conmigo, en todo el tiempo que estuve trabajando en esta tesis.

Resumen

La detección y control de enfermedades es uno de los mayores desafíos en el ámbito de la agricultura. La precisión y detección temprana de enfermedades en las plantas de tomate, pueden ayudar a desarrollar una técnica para un tratamiento oportuno que ayude a reducir las pérdidas económicas.

Con el advenimiento de las redes neuronales, ha permitido mejorar drásticamente la precisión, tanto de la detección de objetos como de la clasificación de imágenes.

En el presente trabajo se utilizan dos redes neuronales pre-entrenadas (transfer learning), para hacer la detección de la hoja de tomate (primera: Faster Mask R-CNN) y luego a partir de la detección, realizar la clasificación de la enfermedad (segunda: red neuronal convolucional), tomando como entrada el área de la imagen donde se encuentra la hoja, luego de realizar la clasificación el sistema desarrollado proporciona información de los síntomas asociados a la enfermedad, como también como proceder con la prevención y el tratamiento a seguir en tiempo real.

Palabras clave: enfermedades en la planta de tomate, redes neuronales convolucionales, detección de objetos, clasificación de imágenes.

Agradecimientos

A mi director de tesis PhD. Alejandro Fernández, por haber creído en mi proyecto, después de ver que casi todas las puertas se cerraban para llevar a cabo mi “locura”, cuyos consejos y asistencia allanaron el camino para llegar a buen puerto en el presente trabajo.

A mi co-director de tesis MSc. Diego Alberto Aracena Pizarro de la Universidad de Tarapacá (Arica-Chile) de quien siempre recibí su apoyo, colaboración y tutoría para alcanzar la meta propuesta, plasmada en la culminación del presente trabajo.

Quiero también extender mi agradecimiento a PhD. Claudio Delrieux, quien con sus consejos y recomendaciones, aportaron de manera significativa en el desarrollo de la tesis.

A los docentes (gestión 2012-2014) del programa maestría en Ingeniería de Software de la Universidad Nacional de La Plata, Facultad de Informática, por brindarme sus conocimientos en mi formación.

... A todos mi más sincero y profundo agradecimiento.

Índice de contenido

RESUMEN	2
AGRADECIMIENTOS.....	3
CAPÍTULO 1: INTRODUCCIÓN.....	9
1.1 INTRODUCCIÓN	9
1.2 JUSTIFICACIÓN	9
1.3 FUNDAMENTACIÓN DEL TEMA ELEGIDO	10
1.4 MOTIVACIÓN	11
1.5 OBJETIVOS	12
1.5.1 <i>Objetivos Generales</i>	12
1.5.2 <i>Objetivos Específicos</i>	12
1.6 METODOLOGÍA GENERAL DE INVESTIGACIÓN Y DESARROLLO DE LA TESIS	12
1.7 ORGANIZACIÓN DE LA TESIS.....	13
CAPÍTULO 2: FUNDAMENTACIÓN TEÓRICA.....	14
2.1 COMPUTER VISION.....	14
2.1.1 <i>Procesamiento de imágenes</i>	17
2.1.1.1 <i>Actividades del procesamiento de imágenes</i>	18
2.1.2 <i>Detección de objetos</i>	19
2.1.3 <i>Segmentación</i>	20
2.1.4 <i>Extracción de características</i>	20
2.2 INTELIGENCIA ARTIFICIAL	21
2.2.1 <i>Machine Learning</i>	22
2.2.2 <i>Deep Learning</i>	23
2.2.3 <i>Deep Learning en la clasificación de imágenes</i>	25
2.2.4 <i>Machine Learning Clásico versus Deep Learning</i>	25
2.2.5 <i>CNNs (Redes Neuronales Convolucionales)</i>	28
2.2.6 <i>DETECCIÓN DE OBJETOS BASADO EN DEEP LEARNING</i>	30
CAPÍTULO 3: DEEP LEARNING EN EL MARCO DE LOS PROCESOS DE INGENIERÍA DE SOFTWARE	34
3.1 MACHINE LEARNING Y SOFTWARE ENGINEERING.....	35
3.2 DEUDA TÉCNICA.....	36
3.3 DESAFÍOS DE DESARROLLO	37
3.3.1 <i>Gestión de experimentos</i>	38
3.3.2 <i>Transparencia limitada</i>	39
3.3.3 <i>Solución de problemas</i>	39
3.3.4 <i>Limitaciones de recursos</i>	40
3.3.5 <i>Pruebas (Testing)</i>	40
3.4 DESAFÍOS DE PRODUCCIÓN	41
3.4.1 <i>Gestión de dependencias</i>	41
3.4.2 <i>Monitoreo y registro</i>	42
3.4.3 <i>Bucles de retroalimentación involuntaria</i>	42
3.4.4 <i>Código de pegamento y sistemas de soporte</i>	42
3.5 DESAFÍOS ORGANIZACIONALES	43
3.5.1 <i>Estimación de esfuerzo</i>	43
3.5.2 <i>Privacidad y seguridad de los datos</i>	43
3.5.3 <i>Diferencias culturales</i>	44
3.6 CONCLUSIONES	44
CAPÍTULO 4: DESCRIPCIÓN DE LA SOLUCIÓN	46

4.1 ADQUISICIÓN DE IMÁGENES.....	46
4.1.1 Enfoque 1: Buscar o recopilar un conjunto de datos.....	46
4.1.2 Enfoque 2: Extensión del navegador Fatkun Batch descargar imagen	47
4.2 PRE PROCESAMIENTO.....	49
4.2.1 Creación del conjunto de datos de entrenamiento para detección de objetos.....	50
4.3 DETECCIÓN DE OBJETOS Y CLASIFICACIÓN	52
4.3.1 Metodología para detección de objetos en la imagen.....	53
4.3.1.1 Redes Neuronales Convolucionales Basada en la Región (R-CNNs).....	53
4.3.1.2 Faster R-CNN	54
4.3.1.2.1 Red Base Para Extraer Características	55
4.3.1.2.2 Region Proposal Network (RPN).....	56
4.3.2 Metodología para la clasificación de imágenes	57
4.4 RECURSOS NECESARIOS	62
4.4.1 Hardware	62
4.4.2 Software.....	62
CAPÍTULO 5: IMPLEMENTACIÓN	65
5.1 ARQUITECTURA	65
5.2 PIPELINE (TUBERÍA DE PROCESAMIENTO).....	66
5.2.1 La tubería de procesamiento de ML en proyectos de IA.....	66
5.3 ALGORITMO DE DETECCIÓN DE OBJETOS (HOJA DE TOMATE)	67
5.3.1 Preparación de la data.....	67
5.3.2 Construcción y entrenamiento del modelo (Build & Train Models).....	69
5.3.2.1 Elección del modelo	69
5.3.2.2 Configuración de archivos para entrenar el modelo	69
5.3.2.2.1 Configuración del modelo.....	71
5.3.2.2.2 Etiquetas de entrenamiento	72
5.3.2.3 Entrenar el modelo	72
5.3.2.4 Guardar modelo entrenado para ejecutarlo posteriormente	74
5.3.3 Predicción y despliegue.....	75
5.4 ALGORITMO DE CLASIFICACIÓN DE ENFERMEDADES.....	77
5.4.1 Preparación de la data.....	77
5.4.2 Construcción y entrenamiento del modelo (Build & Train Models).....	79
5.4.2.1 Elección del modelo	79
5.4.2.2 Entrenar el modelo	81
5.4.2.3 Guardar modelo entrenado para ejecutarlo posteriormente	83
5.4.3 Predicción y despliegue.....	83
CAPÍTULO 6: RESULTADOS.....	85
6.1 ANÁLISIS DE RESULTADOS	85
6.2 RENDIMIENTOS OBTENIDOS.....	92
6.3 ESCENARIO DE APLICACIÓN: PROTOTIPO DOCTOR TOMATTO	95
6.4.1 Estructura del directorio del prototipo.....	96
6.4.2 Códigos principal y complementarios	97
CAPÍTULO 7: TRABAJOS RELACIONADOS	98
7.1 PAPERS CONSULTADOS	98
7.2 APLICACIONES RELACIONADAS.....	99
CAPÍTULO 8: CONCLUSIONES.....	102
8.1 CONCLUSIONES	102
8.2 DESARROLLOS FUTUROS	103
BIBLIOGRAFÍA.....	104

ANEXO A – TUTORIAL DOCTOR TOMATTO.....	106
ANEXO B – CÓDIGO PARA LA DETECCIÓN DE HOJA.....	113

Índice de figuras

FIGURA 2.1: RELACIÓN DE COMPUTER VISION CON OTRAS ÁREAS DE ESTUDIO.	14
FIGURA 2.2: EL COMPUTADOR VE UNA IMAGEN COMO UNA MATRIZ DE VALORES DE PÍXELES.	15
FIGURA 2.3: FILTROS DE IMÁGENES POR CONVOLUCIÓN DE MATRICES.	16
FIGURA 2.5: ACTIVIDADES DEL PROCESAMIENTO DE IMÁGENES.	19
FIGURA 2.6: IMAGEN CON DETECCIÓN DE OBJETOS EN RECUADRO.	19
FIGURA 2.7: IMAGEN ENTRADA (LADO IZQUIERDO), IMAGEN SEGMENTADA (LADO DERECHO).	20
FIGURA 2.8: INTELIGENCIA ARTIFICIAL, MACHINE LEARNING Y DEEP LEARNING.	21
FIGURA 2.9: MACHINE LEARNING EL NUEVO PARADIGMA EN LA PROGRAMACIÓN.	23
FIGURA 2.10: LAS SIMILITUDES ENTRE NEURONAS BIOLÓGICAS Y NEURONAS ARTIFICIALES.	24
FIGURA 2.11: RED NEURONAL ARTIFICIAL.	24
FIGURA 2.12: MACHINE LEARNING EN EL PROCESAMIENTO PARA CLASIFICAR IMÁGENES.	26
FIGURA 2.13: DEEP LEARNING EN EL PROCESAMIENTO PARA CLASIFICAR IMÁGENES.	27
FIGURA 2.14: RENDIMIENTO DEEP LEARNING FRENTE A OTROS ALGORITMOS DE APRENDIZAJE.	27
FIGURA 2.15: PROCESAMIENTO DE IMÁGENES CON REDES NEURONALES CONVOLUCIONALES.	28
FIGURA 2.16: APLICACIÓN DE UNA OPERACIÓN DE CONVOLUCIÓN.	29
FIGURA 2.17: FASTER-RCNN DESCRIPCIÓN GENERAL.....	31
FIGURA 2.18: RPN RED DE REGIÓN PROPUESTA.	32
FIGURA 2.19: DETECCIÓN DE OBJETOS ASOCIADA CON SU PROBABILIDAD.	33
FIGURA 3.1: DEMANDA DE APRENDIZAJE PROFUNDO A LO LARGO DEL TIEMPO RESPALDADA POR LA COMPUTACIÓN.	35
FIGURA 3.2: MACHINE LEARNING = DATOS + CÓDIGO.	36
FIGURA 4.1: RESULTADO DE LA BÚSQUEDA EN CHROME PARA “TOMATE HOJAS”	48
FIGURA 4.2: SELECCIONANDO IMÁGENES A TRAVÉS DE LA EXTENSIÓN <i>FATKUN</i>	48
FIGURA 4.3: DETECCIÓN DE HOJAS CON ENFERMEDAD.....	49
FIGURA 4.4: INTERFAZ DE <i>LABELIMG</i>	50
FIGURA 4.5: ETIQUETANDO UNA IMAGEN Y COLOCANDO CUADRO DELIMITADOR CON <i>LABELIMG</i>	51
FIGURA 4.6: ARCHIVO XML DE UNA IMAGEN ETIQUETADA Y DELIMITADA CON RECUADRO.....	51
FIGURA 4.7: MÉTODOS DE DEEP LEARNING PARA EL PROCESAMIENTO DE IMÁGENES.....	52
FIGURA 4.8: ARQUITECTURA RED NEURONAL CONVOLUCIONAL (CNN).....	53
FIGURA 4.9: ARQUITECTURA FASTER R-CNN TIENE DOS COMPONENTES: 1) UNA RED DE REGIÓN PROPUESTA (RPN), QUE IDENTIFICA REGIONES QUE PUEDEN CONTENER OBJETOS DE INTERÉS Y SU UBICACIÓN APROXIMADA; Y 2) UNA RED FASTER R-CNN, QUE CLASIFICA OBJETOS Y REFINA SU UBICACIÓN, DEFINIDA MEDIANTE CUADROS DELIMITADORES.	55
FIGURA 4.10: IMPLEMENTACIÓN CONVOLUCIONAL DE UNA ARQUITECTURA RPN, DONDE K ES EL NÚMERO DE ANCLAS (ANCHORS).....	56
FIGURA 4.11: EL CLASIFICADOR RPN PREDICE LA PUNTUACIÓN DE OBJETIVIDAD, QUE ES LA PROBABILIDAD DE QUE UNA IMAGEN CONTenga UN OBJETO (PRIMER PLANO) O UN FONDO.	57
FIGURA 4.12: CLASIFICADOR DE IMÁGENES, HOJA TOMATE Y No HOJA TOMATE CON SU PROBABILIDAD.	58
FIGURA 4.13: EN EL APRENDIZAJE AUTOMÁTICO TRADICIONAL, LOS INGENIEROS DEBEN DESARROLLAR ALGORITMOS PARA EXTRAER CARACTERÍSTICAS QUE SE PUEDEN ALIMENTAR AL MODELO. UN MODELO DL NO NECESITA ESTE COMPLEJO PASO PRELIMINAR.	58
FIGURA 4.14: UNA SOLA NEURONA ARTIFICIAL RECIBE DOS NÚMEROS (A Y B), REALIZA OPERACIONES MATEMÁTICAS SIMPLES COMO SUMAR, MULTIPLICAR Y EMITE OTRO NÚMERO Y.	59
FIGURA 4.15: LAS COSAS COMIENZAN A PONERSE INTERESANTES CUANDO SE CONECTAN VARIAS NEURONAS JUNTAS A MEDIDA QUE SE VUELVEN CAPACES DE EXPRESAR OPERACIONES MATEMÁTICAS MÁS COMPLEJAS.	59
FIGURA 4.16: LAS REDES NEURONALES PROFUNDAS PROCESAN LAS IMÁGENES DE ENTRADA (A LA IZQUIERDA) Y PRODUCEN UNA CLASIFICACIÓN QUE SEA CONSISTENTE CON LAS ETIQUETAS DE ENTRENAMIENTO (A LA DERECHA).....	60
FIGURA 4.17: REPRESENTACIONES INTERNAS DESARROLLADAS POR UNA RED NEURONAL PROFUNDA ENTRENADA PARA RECONOCER HOJAS.	61

FIGURA 5.1: ARQUITECTURA, PARTE INTERIOR IZQUIERDA ES DETECCIÓN Y LA DERECHA ES CLASIFICACIÓN.....	65
FIGURA 5.2: TUBERÍA DE PROCESAMIENTO (PIPELINE) DE MACHINE LEARNING.	66
FIGURA 5.3: ESTRUCTURA DE ARCHIVOS PARA LA DETECCIÓN DE HOJA DE TOMATE.....	67
FIGURA 5.4: DIRECTORIO <i>IMG_ENTRENAMIENTO</i> CON LOS ARCHIVOS DE IMÁGENES Y XMLS DE ANOTACIONES.	68
FIGURA 5.5: ARCHIVO XML DE UNA IMAGEN CON LAS ANOTACIONES DE LAS COORDENADAS DEL RECTÁNGULO DIBUJADO.	68
FIGURA 5.6: EJECUCIÓN DE COMANDOS PARA CARGAR EL DATASET DESDE EL REPOSITORIO DE GOOGLE DRIVE Y FRAMEWORK MASKRCNN PARA DETECCIÓN DE OBJETOS.....	70
FIGURA 5.7: ESTRUCTURA DE ARCHIVOS PARA REALIZAR LA DETECCIÓN.	70
FIGURA 5.8: CONFIGURACIÓN DE LA LIBRERÍA MASK RCNN PARA REALIZAR DETECCIÓN.	71
FIGURA 5.9: ACTUALIZACIÓN DE ALGUNOS PARÁMETROS DE LA CONFIGURACIÓN DEL MODELO A ENTRENARSE.....	71
FIGURA 5.10: ASIGNACIÓN DE ETIQUETAS.	72
FIGURA 5.11: PARTE DEL CÓDIGO (RECUADRO) QUE EJECUTA EL ENTRENAMIENTO DEL MODELO.	72
FIGURA 5.12: INICIO DE LA EJECUCIÓN DEL ENTRENAMIENTO DEL MODELO.	73
FIGURA 5.13: SE TERMINA LA EJECUCIÓN DEL ENTRENAMIENTO CUANDO SE CUMPLE EL NÚMERO DE <i>EPOCHS</i> Y SE CONSIGUE UN <i>LOSS</i> CON UN VALOR MÁS BAJO POSIBLE (INFERIOR A 0.9).	73
FIGURA 5.14: CÓDIGO (RECUADRO) QUE EJECUTA EN QUE CARPETA (LOGS) SE GUARDA EL MODELO CREADO.	74
FIGURA 5.15: CARPETA LOGS CON PARTE DE LOS MODELOS CREADOS POR CADA EPOCH.	74
FIGURA 5.16: CÓDIGO PARA HACER PREDICCIONES Y DESPLIEGUE.	75
FIGURA 5.17: IMÁGENES DE INGRESO (COLUMNA IZQUIERDA), IMÁGENES DE SALIDA (COLUMNA DERECHA).	76
FIGURA 5.18: ESTRUCTURA DE ARCHIVOS PARA LA CLASIFICACIÓN DE ENFERMEDADES EN LA HOJA DE TOMATE.	78
FIGURA 5.19: ARCHIVOS DE IMÁGENES EN EL DIRECTORIO: <i>CLASIFICACIONENFERMEDADES/ENTRENAMIENTO/TOMATEIZONTEMPRANO</i>	78
FIGURA 5.20: CARGANDO REPOSITORIO DE GOOGLE DRIVE DEL DATASET E IMPORTANDO LIBRERÍAS.	79
FIGURA 5.21: CONFIGURANDO ALGUNOS HIPER PARÁMETROS.....	79
FIGURA 5.22: DIRECCIONANDO CARPETAS CON LOS ARCHIVOS DE IMÁGENES.	80
FIGURA 5.23: NORMALIZANDO LAS IMÁGENES.	80
FIGURA 5.24: CÓDIGO DE ENTRENAMIENTO DEL MODELO.	81
FIGURA 5.25: INICIO DE LA EJECUCIÓN DEL ENTRENAMIENTO DEL MODELO.	82
FIGURA 5.26: SE TERMINA LA EJECUCIÓN DEL ENTRENAMIENTO CUANDO SE ALCANZA EL NÚMERO DE <i>EPOCHS</i> DEFINIDO Y SE OBTIENE UN <i>LOSS</i> TIENE UN VALOR INFERIOR A 0.9.....	82
FIGURA 5.27: GRABAR EL MODELO EN DISCO.	83
FIGURA 5.28: IMPORTANDO LIBRERÍAS Y CARGANDO EL MODELO YA ENTRENADO.	83
FIGURA 5.29: RESULTADO DE LA EJECUCIÓN DEL PROGRAMA DE PREDICCIÓN DE CLASIFICACIÓN.	84
FIGURA 6.1: PORCENTAJES DESTINADOS A ENTRENAMIENTO Y TEST DEL DATASET PARA DETECCIÓN.....	85
FIGURA 6.2: PORCENTAJES DESTINADOS A ENTRENAMIENTO Y TEST DEL DATASET PARA CLASIFICACIÓN.	86
FIGURA 6.3: PORCENTAJES DESTINADOS A ENTRENAMIENTO Y TEST POR CATEGORÍAS DEL DATASET DE DETECCIÓN.	86
FIGURA 6.4: PORCENTAJES DESTINADOS A ENTRENAMIENTO Y TEST POR CATEGORÍAS DEL DATASET DE CLASIFICACIÓN.	87
FIGURA 6.5: IZQUIERDA - PROCESO TRADICIONAL DE TOMAR UN CONJUNTO DE IMÁGENES DE ENTRADA, APLICANDO DISEÑOS ALGORITMOS DE EXTRACCIÓN DE CARACTERÍSTICAS, SEGUIDOS DEL ENTRENAMIENTO DE UN CLASIFICADOR DE APRENDIZAJE AUTOMÁTICO EN LAS CARACTERÍSTICAS. DERECHA - ENFOQUE DE APRENDIZAJE PROFUNDO DE APILAR CAPAS UNA ENCIMA DE LA OTRA QUE APRENDEN AUTOMÁTICAMENTE CARACTERÍSTICAS MÁS COMPLEJAS, ABSTRACTAS Y DISCRIMINATORIAS [21].....	88
FIGURA 6.6: A MEDIDA QUE AUMENTA LA CANTIDAD DE DATOS DISPONIBLES PARA LOS ALGORITMOS DE APRENDIZAJE PROFUNDO, LA PRECISIÓN MEJORA NOTABLEMENTE, SUPERANDO SUSTANCIALMENTE LOS ENFOQUES TRADICIONALES DE EXTRACCIÓN DE CARACTERÍSTICAS + APRENDIZAJE AUTOMÁTICO.	88
FIGURA 6.7: VISUALIZACIÓN GRÁFICA DE LAS PÉRDIDAS (LOSS) DE ENTRENAMIENTO Y VALIDACIÓN.	89
FIGURA 6.8: VISUALIZACIÓN GRÁFICA DE LAS ACURACIAS DE ENTRENAMIENTO Y VALIDACIÓN.	90
FIGURA 6.9: VISUALIZACIÓN GRÁFICA DE LAS PÉRDIDAS (LOSS) DE ENTRENAMIENTO Y VALIDACIÓN.	90
FIGURA 6.10: CUADRO COMPARATIVO DE LOS ALGORITMOS DE DETECCIÓN.	91
FIGURA 6.11: RENDIMIENTO ALGORITMO DETECCIÓN POR CATEGORÍAS.	92
FIGURA 6.12: RENDIMIENTO ALGORITMO CLASIFICACIÓN POR CATEGORÍAS.....	93
FIGURA 6.13: MÉTRICAS DE RENDIMIENTO OBTENIDAS PARA LA DETECCIÓN.....	94
FIGURA 6.14: MÉTRICAS DE RENDIMIENTO OBTENIDAS PARA LA CLASIFICACIÓN.	95
FIGURA 6.15: DIAGRAMA DE LAS TECNOLOGÍAS EMPLEADAS.	96

FIGURA 1: PÁGINA INICIAL DE LA APLICACIÓN.	106
FIGURA 2: INTERFAZ QUE PERMITE SELECCIONAR LA IMAGEN.	107
FIGURA 3: INTERFAZ DONDE SE SELECCIONA LA FOTO A PROCESAR.....	107
FIGURA 4: FOTO SELECCIONADA Y ACTIVADO EL BOTÓN <i>SUBIR FOTO</i>	108
FIGURA 5: RESULTADO DESPUÉS DE LA DETECCIÓN Y CLASIFICACIÓN.	108
FIGURA 6: ZOOM DE IMAGEN <i>DETECCIÓN DE LA HOJA</i> SELECCIONADA DEL CARRUSEL EN LADO DERECHO.	109
FIGURA 7: DESCRIPCIÓN DE LOS SÍNTOMAS DE LA ENFERMEDAD DETECTADA.....	109
FIGURA 8: DESCRIPCIÓN DE CÓMO HACER LA PREVENCIÓN DE LA ENFERMEDAD DETECTADA.	110
FIGURA 9: DESCRIPCIÓN DE CÓMO HACER EL TRATAMIENTO DE LA ENFERMEDAD DETECTADA.....	110
FIGURA 10: IMAGEN SELECCIONADA Y SUBIDA, QUE NO CORRESPONDE A UNA HOJA DE TOMATE.....	111
FIGURA 11: MENSAJE DE ERROR CUANDO LA IMAGEN SUBIDA NO ES DE UNA HOJA DE TOMATE.	111
FIGURA 12: INFORMACIÓN REFERENTE A LA OPCIÓN DE <i>CRÉDITO</i> MENÚ PRINCIPAL.	112

Índice de tablas

TABLA 4.1: ESPECIES DE CULTIVO QUE CONTEMPLA EL DATASET PLANTVILLAGE.....	46
TABLA 4.2: ENFERMEDADES Y PLAGAS QUE CONTEMPLA EL DATASET PLANTVILLAGE.	46
TABLA 6.1: CIFRAS NUMÉRICAS DE LOS DATASET EMPLEADOS.	85
TABLA 6.2: DATOS NUMÉRICOS DE LAS CATEGORÍAS DEL DATASET DE DETECCIÓN.	86
TABLA 6.3: DATOS NUMÉRICOS DE LAS CATEGORÍAS DEL DATASET DE CLASIFICACIÓN.....	87
TABLA 6.4: MÉTRICAS CUANTITATIVAS DE LA FIGURA 6.11.	91
TABLA 6.5: RENDIMIENTO ALGORITMO DE DETECCIÓN.	92
TABLA 6.6: RENDIMIENTO ALGORITMO DE CLASIFICACIÓN.	92
TABLA 7.1: COMPARACIÓN DE LAS FUNCIONALIDADES QUE PRESTAN PLANTNET Y PLANTSAP VERSUS DOCTOR TOMATTO.	100
TABLA 7.2: COMPARACIÓN DE LAS FUNCIONALIDADES QUE PRESTAN LEAF DOCTOR Y PLANTIX VERSUS DOCTOR TOMATTO.....	101

Capítulo 1: Introducción

1.1 Introducción

El tomate es uno de los cultivos más producidos en todo el mundo. De acuerdo con las estadísticas obtenidas de la Organización de las Naciones Unidas para la Alimentación y la Agricultura (FAO), aproximadamente 170.750 kilotonnes de tomate producidos en el año 2014 en total en todo el mundo. Según el Instituto de Estadística de Turquía, Turquía ha producido 12.600 kilotonnes de tomate en el año 2016. Estas cantidades de producción se ven afectadas por las plagas y enfermedades que se producen en las plantas de tomate. Para prevenir estas enfermedades y plagas, se utilizan métodos costosos y diversos pesticidas. El uso generalizado de estos métodos químicos perjudica la salud de las plantas y la salud humana, así como afecta al medio ambiente de manera negativa. También estos métodos, aumentan los costos de producción [1] [2].

El diagnóstico de enfermedades de plantas mediante observación óptica de los síntomas sobre las hojas de ellas, incorpora un grado de complejidad significativamente alto. Debido a esta complejidad y al gran número de plantas cultivadas y sus problemas fitopatológicos existentes, incluso agrónomos experimentados y los patólogos de plantas a menudo no logran diagnosticar con éxito específicas enfermedades, en consecuencia se tiende a conclusiones y tratamientos erróneos. La existencia de un sistema computacional automatizado para la detección y diagnóstico de enfermedades de plantas, ofrecería una valiosa asistencia al agrónomo a quien se le solicita que realice dichos diagnósticos a través de la observación óptica de hojas de plantas infectadas [4].

1.2 Justificación

En las tareas y actividades agrícolas, específicamente en horticultura, la detección y clasificación de enfermedades a simple observación en algunas hortalizas, particularmente en el tomate resulta ser una labor no trivial, tanto para el agricultor como también para el mismo ingeniero agrónomo ante la presencia de ciertas enfermedades [4].

La identificación de enfermedades de las plantas es una de las actividades más básicas e importantes en la agricultura. En la mayoría de los casos, la identificación se realiza manualmente, ya sea visualmente o por microscopía. El problema con la evaluación visual es que, siendo una tarea subjetiva, es propensa a los fenómenos psicológicos y cognitivos que puede llevar a sesgos, ilusiones ópticas y, en última instancia a error. Por otro lado, los análisis de laboratorio como el molecular, enfoques inmunológicos o basados en el cultivo de patógenos, a menudo consumen mucho tiempo y no proporcionan respuestas de una manera oportuna. En este contexto, es obligatorio desarrollar métodos automáticos capaces de identificar enfermedades de forma rápida y fiable [3].

En base a lo expresado en párrafos anteriores se establece que los sistemas de asistencia automática para la identificación de enfermedades ayudan al agricultor a simplificar la labor y contribuyen a reducir pérdidas frente a los enfoques existentes que presentan dificultades para detectar e identificar las enfermedades a tiempo para su consiguiente tratamiento.

La propuesta de valor del presente trabajo tiene como aporte el mejorar la precisión de la detección y clasificación de enfermedades en términos de tiempo, dinero (minimizar pérdidas) y esfuerzo usando Deep Learning y Computer Vision en comparación a sistemas existentes con menor precisión [5][6].

Con el desarrollo de sistemas computacionales en los últimos años, y en particular unidades gráficas de procesamiento (GPU), las aplicaciones de Inteligencia Artificial relacionadas con el Aprendizaje Automático (Machine Learning) han logrado un crecimiento exponencial, conduciendo al desarrollo de nuevas metodologías y modelos, que ahora forman una nueva categoría, la del Aprendizaje Profundo (Deep Learning). Este aprendizaje se refiere al uso de arquitecturas de redes neuronales artificiales que contienen un número bastante grande de capas de procesamiento, en contraposición a las arquitecturas "shallower" (superficiales) de más metodologías de redes neuronales tradicionales el cual actualmente es computacionalmente factible. Los modelos de aprendizaje profundo han revolucionado sectores como el reconocimiento de patrones en imágenes dando un gran impulso a las aplicaciones que utilizan estos procesos, en especial en la agricultura [4].

1.3 Fundamentación del tema elegido

Aunque el aprendizaje profundo (DL) es un subcampo bastante antiguo de aprendizaje automático (ML), solo comenzó a destacar a principios de la década de 2010. En los pocos años transcurridos desde entonces, ha logrado nada menos que una revolución en el campo, con resultados notables en problemas de percepción como ver y escuchar, problemas que involucran habilidades que parecen naturales e intuitivas para los humanos, pero durante mucho tiempo han sido esquivos para las máquinas.

En particular, el aprendizaje profundo ha logrado los siguientes avances, todos en áreas históricamente difíciles del aprendizaje automático:

- Clasificación de imágenes a nivel casi humano
- Reconocimiento de voz a nivel casi humano
- Transcripción de escritura a mano a nivel humano
- Traducción automática mejorada
- Conversión de texto a voz mejorada
- Asistentes digitales como Google Now y Amazon Alexa
- Conducción autónoma a nivel casi humano
- Target (función objetivo) mejor orientación de los anuncios, como la utilizan Google, Baidu y Bing
- Resultados de búsqueda mejorados en la web

- Habilidad para responder preguntas de lenguaje natural

Todavía se está explorando todo lo que puede hacer el aprendizaje profundo. Comenzando a aplicarlo a una amplia variedad de problemas más allá de la percepción de la máquina y del lenguaje natural [7].

¿Por qué el aprendizaje profundo? ¿por qué ahora? las dos ideas clave del aprendizaje profundo para la visión por computador (redes neuronales convolucionales y propagación hacia atrás) ya se entendieron bien en 1989. El algoritmo de memoria de corto plazo largo (LSTM), que es fundamental para el aprendizaje profundo para serie temporal, fue desarrollado en 1997 y apenas ha cambiado desde entonces, ¿por qué el aprendizaje profundo solo comenzó a despegar después de 2012? ¿qué cambió en estas dos décadas?. En general, tres fuerzas técnicas están impulsando avances en el aprendizaje automático:

- Hardware
- Conjuntos de datos y puntos de referencia
- Avances algorítmicos

Debido a que el campo se guía por hallazgos experimentales en lugar de teóricos, los avances algorítmicos solo son posibles cuando se dispone de datos y hardware apropiados para probar nuevas ideas (o ampliar las ideas antiguas, como suele ser el caso). El aprendizaje automático no es solo matemáticas o física, donde se pueden hacer grandes avances con una pluma y un pedazo de papel es una ciencia de la ingeniería. Los verdaderos cuellos de botella a lo largo de los años 90 y 2000 fueron datos y hardware, esto es lo que sucedió durante ese tiempo: Internet despegó, y el alto rendimiento de los chips gráficos fueron desarrollados para las necesidades del mercado de juegos [7].

1.4 Motivación

El aprendizaje profundo es una técnica emergente de inteligencia artificial (IA) que utiliza estructuras de análisis sofisticadas llamadas redes neuronales para realizar asociaciones precisas dentro de un conjunto de datos. En particular, los sistemas de aprendizaje profundo pueden aprender procesando datos en bruto sin reglas codificadas por humanos o conocimiento de dominio. Estos sistemas son particularmente expertos en el lenguaje y la clasificación de imágenes, donde un patrón puede representar una idea abstracta como sentimiento, intención o incluso el concepto general de cómo se ve un gato o un perro. Estos sistemas también son excelentes para hacer predicciones, como el comportamiento de los clientes o las previsiones meteorológicas a largo plazo. ¡También hay un increíble potencial para el análisis de imágenes de frutas y hortalizas, para diagnosticar enfermedades y plagas, facilitando y agilizando la detección y diagnóstico de plagas y/o enfermedades presente en las plantas para luego realizar el tratamiento respectivo.

1.5 Objetivos

1.5.1 Objetivos Generales

El objetivo general de este trabajo es realizar investigación que se constituya en un aporte creativo a nivel nacional al **aplicar el aprendizaje profundo a problemas de visión de computadora tales como detección y clasificación específicamente en las enfermedades del tomate mediante el procesamiento de imágenes digitales.**

1.5.2 Objetivos Específicos

1. Construir un dataset con imágenes etiquetadas para entrenar y probar los algoritmos de detección y clasificación automática de enfermedades del tomate.
2. Describir las características relevantes de los atributos y de las etiquetas que conformarán el modelo.
3. Determinar el rendimiento del modelo entrenado.
4. Incorporar métricas para definir el modelo que tiene mejor representación.
5. Adquirir criterios para determinar que algoritmos se deben aplicar dentro del marco de la construcción del modelo.
6. Describir los desafíos del Deep Learning en el marco de los procesos de ingeniería de software.

1.6 Metodología General de Investigación y Desarrollo de la Tesis

Comprende una revisión bibliográfica extensiva del estado del arte, además del desarrollo de la solución, que involucra:

- Adquisición de Imágenes
- Pre Procesamiento
- Detección de Objetos y Clasificación
- Recursos Necesarios

Los que se detallan de manera más profusamente en el Capítulo 4: Descripción de la Solución.

1.7 Organización de la Tesis

Esta tesis está estructurada en 8 capítulos y dos anexos, según se detalla a continuación:

Capítulo 1 – Introducción: se expone una introducción global donde se describe el escenario para este trabajo, el objetivo principal y los objetivos específicos del mismo.

Capítulo 2 – Fundamentación Teórica: se presenta la definición de conceptos de computer visión, inteligencia artificial, machine learning y deep learning, la diferencia entre machine learning y deep learning, se detallan los usos y campos de aplicación como también los desafíos del Deep Learning en el marco de los procesos de ingeniería de software.

Capítulo 3 – Deep Learning en el marco de los procesos de Ingeniería de Software: se hace una descripción de la falta e herramientas que funcionen bien, como también la ausencia de mejores prácticas para construir sistemas de Deep Learning en las tres áreas de desarrollo, producción y desafíos.

Capítulo 4 – Descripción de la Solución: se describe de forma clara la solución, teniendo como contenidos del capítulo: Descripción general, Método de Trabajo y Recursos Necesarios.

Capítulo 5 – Implementación: se presenta la aplicación de la metodología previamente definida y la descripción de las actividades llevadas a cabo para configurar e implementar el algoritmo de detección de la hoja del tomate mediante TensorFlow, como también el algoritmo para la clasificación de las enfermedades que se presentan en la hoja de tomate mediante Keras y el Prototipo Doctor Tomatto como prueba de concepto.

Capítulo 6 – Resultados: se estructura en base a los objetivos específicos definidos en el primer capítulo. Considerando dichos objetivos, cada subsección de este capítulo cuenta con la siguiente estructura: Análisis de Resultados, Rendimientos Obtenidos y Escenario de Aplicación.

Capítulo 7 – Trabajos Relacionados: se describen artículos publicados en revistas y trabajos presentados en reuniones científicas como también aplicaciones desarrolladas que están relacionadas a este trabajo de tesis.

Capítulo 8 – Conclusiones: se detallan las conclusiones del trabajo y se comenta posibles nuevas líneas de investigación a desarrollar en el futuro cercano.

Anexo A – Tutorial Doctor Tomatto.

Anexo B – Código para detección de la hoja.

Capítulo 2: Fundamentación Teórica

En este capítulo se presentan los principales conceptos relacionados con Computer Vision, Inteligencia Artificial, Machine Learning y Deep Learning y concluyendo Deep Learning en el marco de los procesos de Ingeniería de Software, con el objetivo de contextualizar el proyecto. También se describen las tecnologías que se utilizan hoy en día en torno a la detección y clasificación aplicado al procesamiento de imágenes digitales. Finalmente se analizan desafíos del Deep Learning en el marco de los procesos de ingeniería de software.

2.1 Computer Vision

La visión computacional es un área de la informática, matemáticas e ingeniería eléctrica. Incluye formas de adquirir, procesar, analizar y comprender imágenes y videos del mundo real para imitar la visión humana. Además, a diferencia de la visión humana, la visión artificial también se puede utilizar para analizar y procesar la profundidad de imágenes infrarrojas.

La visión por ordenador también se ocupa de la teoría de la extracción de información de imágenes y videos. Un sistema de visión por ordenador puede aceptar diferentes formas de datos como una entrada que incluye, entre otras, imágenes, secuencias de imágenes y videos que pueden ser transmitidos desde múltiples fuentes para procesar y extraer información útil para la toma de decisiones [8].

La inteligencia artificial y la visión por ordenador comparten muchos temas, como el procesamiento de imágenes, reconocimiento de patrones y técnicas de aprendizaje automático, como se muestra en el siguiente diagrama:

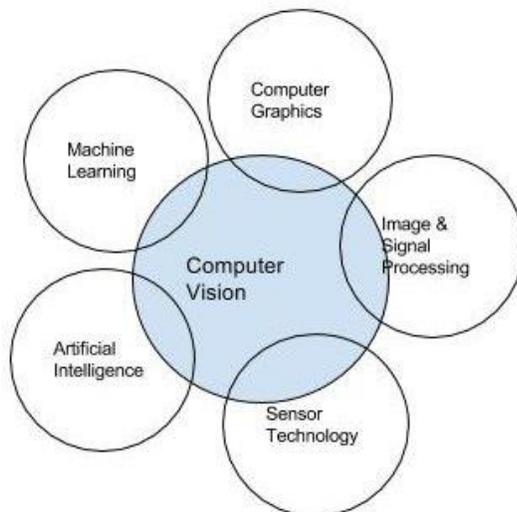


Figura 2.1: Relación de Computer Vision con otras áreas de estudio.

Las tareas típicas de la visión computacional son las siguientes:

- Reconocimiento y clasificación de objetos
- Detección y análisis de movimiento
- Reconstrucción de imágenes y escenas

El análisis de imágenes y la visión por computadora siempre han sido importantes en aplicaciones industriales y científicas. Con la popularización de los teléfonos celulares con cámaras y conexiones de internet potentes, las imágenes ahora son generadas cada vez más por los consumidores. Por lo tanto, hay oportunidades para hacer uso de la visión computacional para proporcionar una mejor experiencia de usuario en nuevos contextos [9].

Antes de sumergirnos directamente en el procesamiento de imágenes, primero se debe entender las imágenes. Una imagen, como la ven los humanos, es una cuadrícula bidimensional con cada celda de la cuadrícula rellena con un valor de color, también denominado valor de píxel. Cada celda de la cuadrícula se denomina formalmente un elemento de imagen (comúnmente abreviado como píxel). Una computadora también ve la imagen de la misma manera. Una imagen en una computadora es una matriz bidimensional de números con cada celda en la matriz que almacena los valores de píxeles correspondientes en la imagen. La siguiente figura es un ejemplo de una matriz de imágenes. La matriz de una parte de la imagen recortada (recuadro verde) se muestra a la derecha:

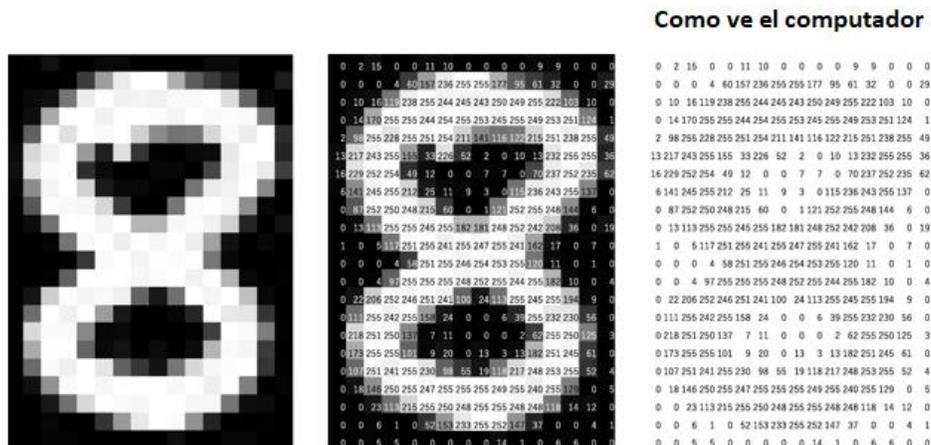


Figura 2.2: El computador ve una imagen como una matriz de valores de píxeles.

Una imagen en su forma más simple (un solo canal; por ejemplo, imágenes binarias o monocromas, en escala de grises o en blanco y negro) es una función bidimensional $f(x, y)$ que asigna un par de coordenadas a un valor entero / real, que está relacionado con la intensidad / color del punto.

Una imagen también puede tener múltiples canales (por ejemplo, imágenes en color RGB, donde un color se puede representar utilizando tres canales: rojo, verde y azul). Para una imagen RGB de color, cada píxel en la coordenada (x, y) puede representarse por una tupla de tres (r_x, y, g_x, y, b_x, y) . Para

poder procesarlo en una computadora, una imagen $f(x, y)$ necesita digitalizarse espacialmente y en amplitud.

La digitalización de las coordenadas espaciales (x, y) se denomina muestreo de imagen. La digitalización de amplitud se llama cuantización de nivel de gris. En una computadora, un valor de píxel correspondiente a un canal generalmente se representa como un valor entero entre $(0-255)$ o un valor de punto flotante entre $(0-1)$.

Una imagen se almacena como un archivo y puede haber muchos tipos (formatos) de archivos diferentes. En general, cada archivo tiene algunos metadatos y algunos datos que se pueden extraer como matrices multidimensionales (por ejemplo, matrices 2D para imágenes binarias o de nivel de gris y matrices 3D para imágenes en color RGB y YUV). La siguiente figura muestra cómo los datos de imagen se almacenan como matrices para diferentes tipos de imagen. Como se muestra, para una imagen en escala de grises, una matriz (matriz 2-D) de ancho x alto es suficiente para almacenar la imagen, mientras que una imagen RGB requiere una matriz 3D de una dimensión de ancho x alto x 3.

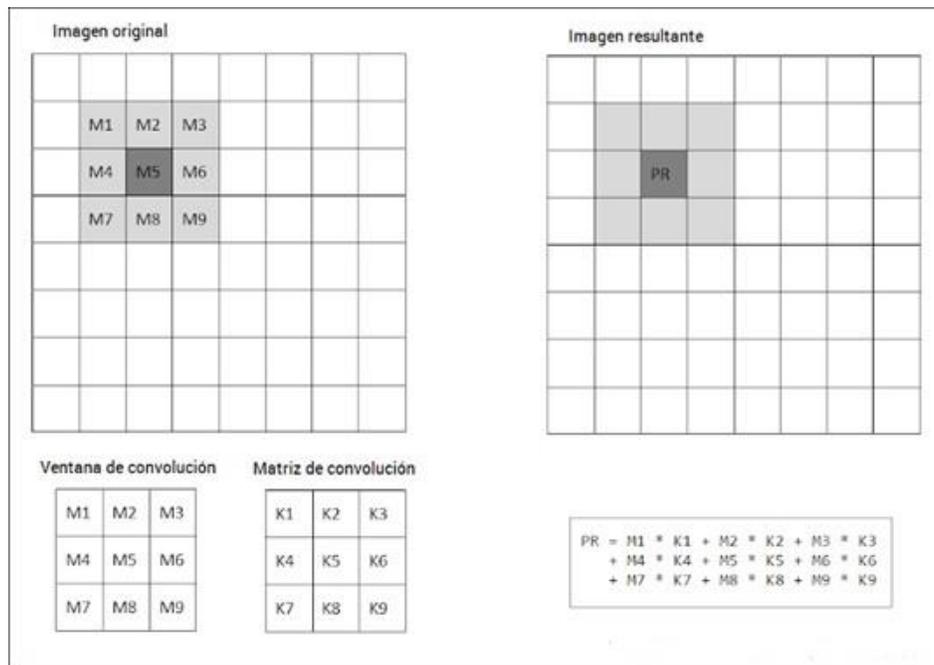


Figura 2.3: Filtros de imágenes por convolución de matrices.

La figura 2.4 muestra un ejemplo de imagen binaria, escala de grises y RGB:



Figura 2.4: Imágenes binaria, escala de grises y RGB.

2.1.1 Procesamiento de imágenes

El procesamiento de imágenes se refiere al procesamiento automático, la manipulación, el análisis y la interpretación de imágenes mediante algoritmos y códigos en una computadora. Tiene aplicaciones en muchas disciplinas y campos de la ciencia y la tecnología como la televisión, la fotografía, robótica, teledetección, diagnóstico médico e inspección industrial. Redes sociales como Facebook e Instagram, a los que nos hemos acostumbrado en nuestra vida diaria y donde cargamos toneladas de imágenes cada día, son ejemplos típicos de las industrias que necesitan para usar / innovar muchos algoritmos de procesamiento de imágenes para procesar las imágenes que subimos [10].

El procesamiento de imágenes como se mencionó anteriormente es el campo de estudio y análisis de imágenes. Hay una gran cantidad de información oculta en una imagen que procesamos inconscientemente. Por ejemplo, ¿cuáles son los diferentes objetos en la imagen?, ¿hay un automóvil en la imagen? ¿Cuáles son las similitudes entre dos imágenes? Las respuestas a estas preguntas pueden ser simples para nosotros, los humanos, pero para una computadora, responder estas preguntas es extremadamente difícil. En el transcurso de esta investigación, nuestro objetivo es implementar algunos de los algoritmos que pueden ayudarnos a responder algunas de estas preguntas. La esencia del procesamiento de imágenes es utilizar las diferentes propiedades de una imagen, como el color, las correlaciones entre diferentes píxeles, objetos. Ubicaciones y otros detalles finos para extraer información significativa, como bordes, objetos y contornos, que se denominan formalmente características de imagen. Estas características se pueden usar en diferentes aplicaciones, como medicamentos, seguridad, servicios de redes sociales, autos que se manejan solos, detección y clasificación de enfermedades del tomate [11].

2.1.1.1 Actividades del procesamiento de imágenes

Los siguientes pasos describen las actividades básicas en el proceso de procesamiento de imágenes:

1. **Adquisición y almacenamiento:** Diversos tipos de sensores son utilizados para la adquisición de imágenes (por ejemplo, cámara digital, aparato de rayos X, aparatos de ultra sonido, escáner, etc.) La imagen debe ser capturada y almacenada en algún dispositivo (como un disco duro), como un archivo (por ejemplo, un archivo JPEG) [10] [12].

En cuanto al almacenamiento, la imagen debe leerse del disco duro a la memoria y almacenarse utilizando alguna estructura de datos (por ejemplo, numpy ndarray), y la estructura de datos debe ser serializada en un archivo de imagen más adelante, posiblemente después de ejecutar algunos algoritmos en la imagen [10].

2. **Pre-procesamiento:** Para la manipulación, mejora y restauración es necesario ejecutar algunos algoritmos de pre-procesamiento para hacer lo siguiente [10]:
 - Ejecutar algunas transformaciones en la imagen (muestreo y manipulación; por ejemplo, conversión a escala de grises).
 - Mejorar la calidad de la imagen (filtrado, por ejemplo, desenfoque).
 - Restaurar la imagen a partir de la degradación del ruido.
 - Modificar tamaño de la imagen a un estándar.

Es esencial que el concepto de *región* u *objeto de interés* sea asimilado: Son las regiones o elementos presentes en la imagen del que se quiere identificar y obtener informaciones [12].

3. **Segmentación:** La imagen necesita ser segmentada en una o más imágenes con el propósito de facilitar la extracción de características de los objetos que son de interés [10] [12].
4. **Extracción de características:** Información extracción/representación de la imagen necesita ser representada en alguna forma alternativa, por ejemplo, una de las siguientes [10]:
 - Se pueden calcular algunos descriptores de características hechos manualmente que pueden ser computados desde la imagen (por ejemplo, descriptores HOG, con procesamiento clásico de imágenes).
 - Algunas características pueden aprenderse automáticamente desde la imagen (por ejemplo, los valores de pesos y sesgos aprendidos en las capas ocultas de una red neuronal con aprendizaje profundo).
 - La imagen será representada usando esa representación alternativa.
5. **Reconocimiento de patrones:** Comprensión/interpretación de la imagen, esta representación será utilizada para comprender mejor la imagen con lo siguiente [10]:
 - Clasificación de imágenes (por ejemplo, si una imagen contiene un objeto humano o no).

- Reconocimiento de objetos (por ejemplo, encontrar la ubicación de los objetos del automóvil con un recuadro remarcado).

6. **Resultado:** En esta última etapa ocurre el procesamiento de alto nivel, con el objetivo de reconocer el objeto segmentado a través de sus características y definirlo en una determinada clase [10].

El siguiente diagrama describe las diferentes actividades en el procesamiento de imágenes:

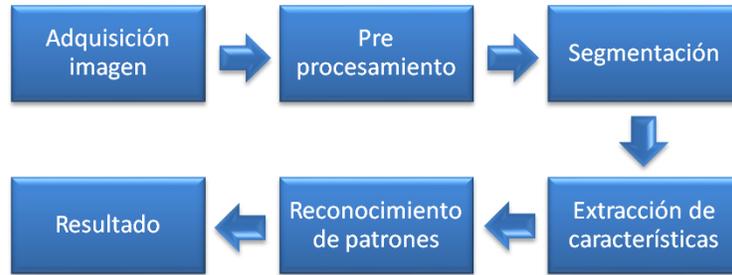


Figura 2.5: Actividades del Procesamiento de Imágenes.

2.1.2 Detección de objetos

En la imagen a continuación, se muestran recuadros de contorno rectangulares alrededor de algunos objetos, no se está categorizando qué objeto está en el cuadro. El siguiente paso sería decir que la caja contiene una hoja, un tomate o una manzana. Esta observación combinada de detección y categorización de la caja a menudo se refiere como detección de objetos.

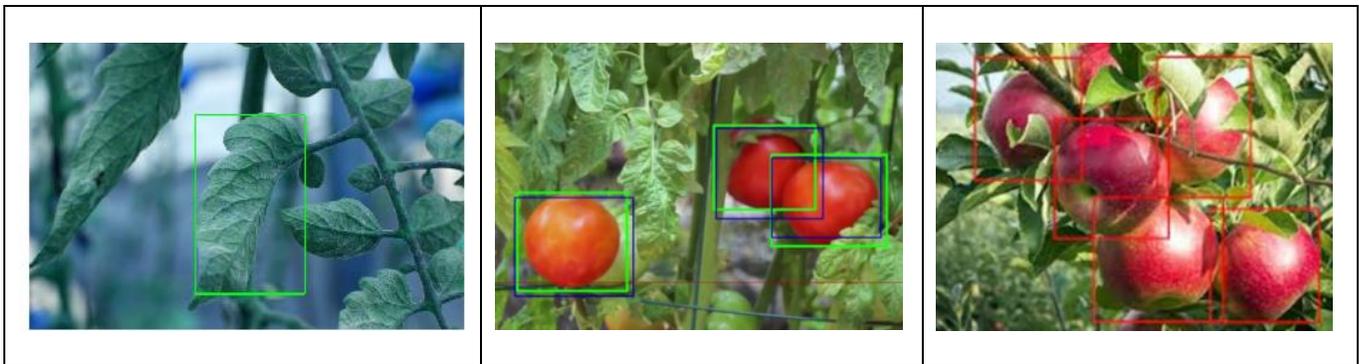


Figura 2.6: Imagen con detección de objetos en recuadro.

La detección de objetos consiste en dibujar rectángulos (llamados "cuadros delimitadores") alrededor de objetos de interés en una imagen y asociar cada rectángulo con una clase.

2.1.3 Segmentación

La segmentación de imágenes es la creación de regiones de clúster en imágenes, de manera que el clúster (agrupación) tiene propiedades similares. El enfoque habitual es agrupar los píxeles de la imagen perteneciente al mismo objeto. Las aplicaciones recientes han crecido en autos autónomos (sin conductor) y análisis de imágenes médicas en el campo de la salud (rayos X, tomografía, etc.) [13].

Un ejemplo es el que se muestra en lo siguiente (figura 2.7), donde se ve que las entradas están a la izquierda y los resultados de la segmentación están a la derecha. Los colores de un objeto están de acuerdo con categorías de objetos predefinidas.

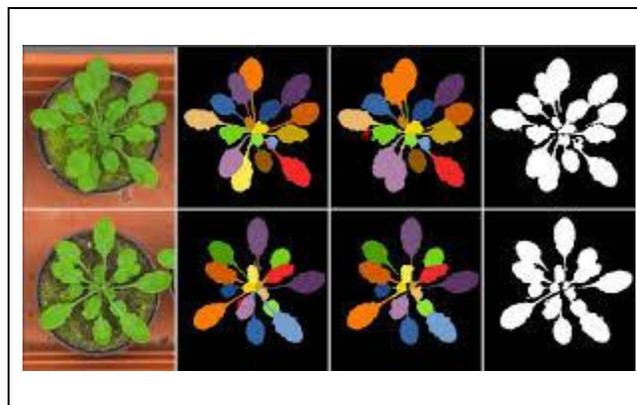


Figura 2.7: Imagen entrada (lado izquierdo), imagen segmentada (lado derecho).

2.1.4 Extracción de características

La extracción de características de un objeto en una imagen es un procedimiento fundamental de cualquier sistema de Visión Computacional. Ella consiste en obtener informaciones que hacen posible clasificar o identificar un objeto.

Las características de un objeto pueden ser clasificadas en categorías, siendo las principales: características de aspecto, dimensionales, inerciales y topológicas [12].

Un sistema de visión por computador necesita aprender varias características que describen un objeto, tal que sea bastante fácil distinguirlo de otros objetos.

Cuando se diseña software para hacer coincidir imágenes o detectar objetos en imágenes, lo básico es la tubería de procesamiento (pipeline) para la detección, que se formula desde una perspectiva de aprendizaje automático. Esto significa que se considera un conjunto de imágenes, se extrae rasgos de información importante, el modelo aprende y usa lo aprendido para predecir sobre nuevas imágenes para detectar objetos similares [13].

En la visión por computador, un rasgo característico (feature) es una pieza de información (a menudo representado matemáticamente como un vector de una o dos dimensiones) que es extraído de datos que son relevantes para la tarea en cuestión. Las características pueden ser algunos puntos clave en las imágenes, bordes específicos y pedazos discriminativos. Deben ser fáciles de obtener de nuevas imágenes y contener la información necesaria para un mayor reconocimiento [14].

2.2 Inteligencia Artificial

Primeramente se necesita definir claramente de que se está hablando cuando se menciona el término Inteligencia Artificial (IA). ¿Qué es Inteligencia Artificial, Machine Learning y Deep Learning ? ¿cómo se relacionan entre sí?

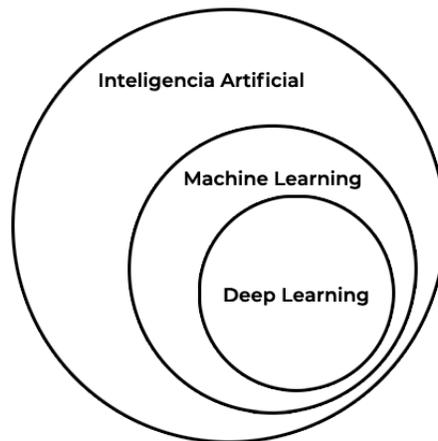


Figura 2.8: Inteligencia Artificial, Machine Learning y Deep Learning.

La inteligencia artificial nació en la década de 1950, cuando un puñado de pioneros del incipiente campo de la informática comenzó a preguntar si las computadoras podrían hacerse para "pensar": una pregunta cuyas ramificaciones todavía estamos explorando hoy. Una definición concisa del campo sería la siguiente: ***el esfuerzo por automatizar las tareas intelectuales que normalmente realizan los humanos***. Como tal, la IA es un campo general que abarca el machine learning (aprendizaje automático) y deep learning (aprendizaje profundo), pero eso también incluye muchos más enfoques que no involucran aprendizaje. Los primeros programas de ajedrez, por ejemplo, solo involucraban reglas codificadas elaboradas por programadores, y no calificaron como aprendizaje automático.

Durante bastante tiempo, muchos expertos creían que la inteligencia artificial a nivel humano podría lograrse al tener los programadores que elaboran un conjunto suficientemente grande de reglas explícitas para manipular conocimiento. Este enfoque se conoce como IA simbólica, y fue el paradigma

dominante en IA desde los años cincuenta hasta finales de los ochenta. Alcanzó su máxima popularidad durante el boom de los sistemas expertos en los años ochenta.

Aunque la IA simbólica demostró ser adecuada para resolver problemas lógicos bien definidos, como jugar al ajedrez, resultó ser intratable descubrir reglas explícitas para resolver problemas más complejos y difusos, como clasificación de imágenes, reconocimiento de voz y traducción de idiomas. Surgió un nuevo enfoque para tomar el lugar simbólico de la IA: el machine learning (aprendizaje automático) [7].

2.2.1 Machine Learning

En la Inglaterra victoriana, Lady Ada Lovelace era amiga y colaboradora de Charles Babbage, el inventor del motor analítico: el primer computador mecánico de propósito general conocido. Aunque visionario y muy adelantado a su tiempo, el motor analítico no era una computadora de uso general cuando se diseñó en el 1830 y 1840, porque el concepto de cómputo de propósito general aún no se había inventado. Simplemente se entiende como una forma de utilizar operaciones mecánicas para automatizar ciertos cálculos desde el campo del análisis matemático, de ahí el nombre de motor analítico. En 1843, Ada Lovelace comentó sobre la invención, "El motor analítico no tiene pretensiones de originar nada. Puede hacer lo que sepamos, ordenar que actúe ... Su función es ayudar a poner a disposición con lo que ya estamos familiarizados".

Este comentario fue citado posteriormente por el pionero de la inteligencia artificial Alan Turing como "la objeción de Lady Lovelace" en su histórico artículo de 1950 "Maquinaria de cómputo e inteligencia"¹ que presentó la prueba de Turing, así como los conceptos clave que darían forma a la IA. Turing citaba a Ada Lovelace mientras reflexionaba si las computadoras de uso general podrían ser capaces de aprender y originalmente llegó a la conclusión de que sí podían.

El machine learning (aprendizaje automático) surge de esta pregunta: ¿podría una computadora ir más allá de "lo que nosotros sabemos cómo ordenar que se realice" y aprender por sí misma cómo realizar una tarea específica?

¿Podría una computadora sorprendernos? En lugar de que los programadores elaboren el procesamiento de datos reglas a mano, ¿podría una computadora aprender estas reglas automáticamente al observar los datos?

Esta pregunta abre la puerta a un nuevo paradigma de programación. En la programación clásica, el paradigma de la IA simbólica, los humanos introducen reglas (un programa) y datos para procesarse de acuerdo con estas reglas y obtener respuestas (ver figura 2.9). Con el aprendizaje automático, los humanos ingresan datos, así como las respuestas que se esperan de los datos y tiene como salida las reglas. Estas reglas se pueden aplicar a los datos nuevos para producir respuestas originales.

¹ A. M. Turing, "Computing Machinery and Intelligence," *Mind* 59, no. 236 (1950): 433-460.

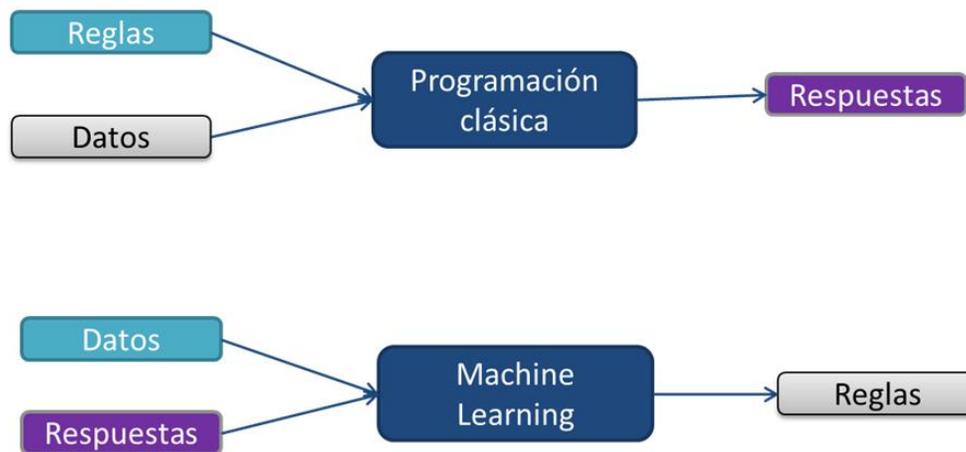


Figura 2.9: Machine Learning el nuevo paradigma en la programación.

Un sistema de aprendizaje automático (aprendizaje supervisado) es *entrenado* en lugar de programarse explícitamente. Se presenta con muchos ejemplos relevantes para una tarea, y encuentra una estructura estadística en estos ejemplos que eventualmente permite que el sistema presente reglas para automatizar la tarea.

Por ejemplo, se desea automatizar la tarea de etiquetar sus fotos de vacaciones, usted podría presentar un sistema de aprendizaje automático con muchos ejemplos de imágenes ya etiquetadas por humanos y el sistema aprendería reglas estadísticas para asociar imágenes a etiquetas específicas.

Aunque el aprendizaje automático solo comenzó a florecer en la década de 1990, rápidamente pasó a convertirse en el sub campo más popular y más exitoso de IA, una tendencia impulsada por la disponibilidad de hardware más rápido y conjuntos de datos más grandes. El aprendizaje automático está estrechamente relacionado a las estadísticas matemáticas, pero difiere de las estadísticas en varias formas importantes.

A diferencia de las estadísticas, el aprendizaje automático tiende a tratar con conjuntos de datos grandes y complejos (como un conjunto de datos de millones de imágenes, cada una compuesta por decenas de miles de píxeles) para el que el análisis estadístico clásico, como el análisis bayesiano, no sería práctico. Como resultado, el aprendizaje automático, y especialmente el aprendizaje profundo, exhibe relativamente poca teoría matemática — quizás muy poco — y está orientado a la ingeniería. Es una práctica disciplinaria en la que las ideas se prueban empíricamente más a menudo que teóricamente [7].

2.2.2 Deep Learning

El comportamiento de aprendizaje de las neuronas biológicas inspiró a los científicos a crear una red de neuronas que están conectadas entre sí. Imitando cómo se procesa la información en el cerebro humano, cada neurona artificial envía una señal a todas las neuronas a las que está conectada cuando se

activan suficientes señales de entrada. Por tanto, las neuronas tienen un mecanismo simple a nivel individual, pero cuando se tienen millones de estas neuronas apiladas en capas y conectadas entre sí, cada neurona está conectada a miles de otras neuronas, lo que produce un comportamiento de aprendizaje. La construcción de una red neuronal multicapa se denomina aprendizaje profundo (figura 2.11) [19].

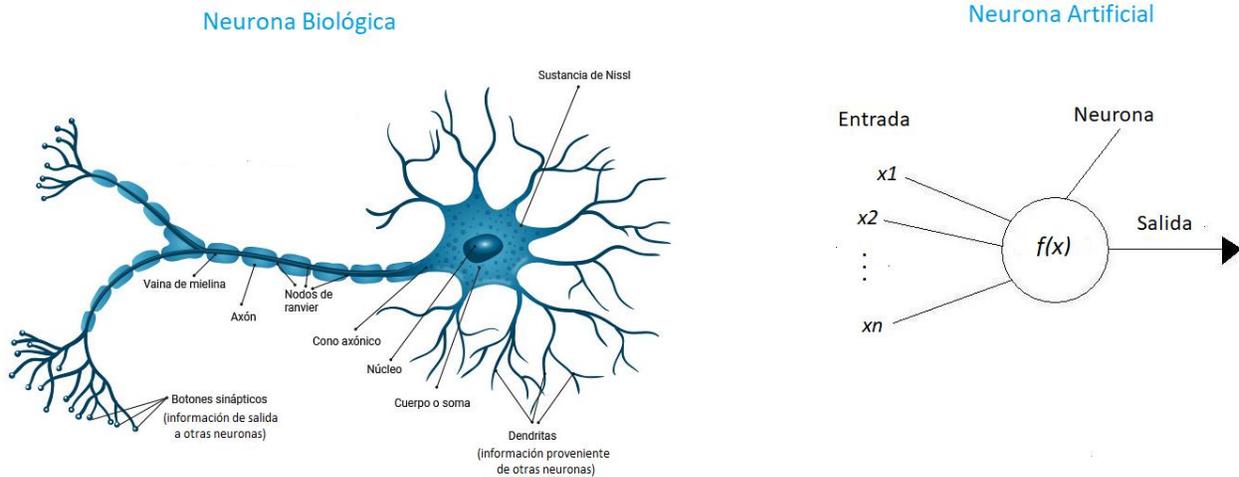


Figura 2.10: Las similitudes entre neuronas biológicas y neuronas artificiales.

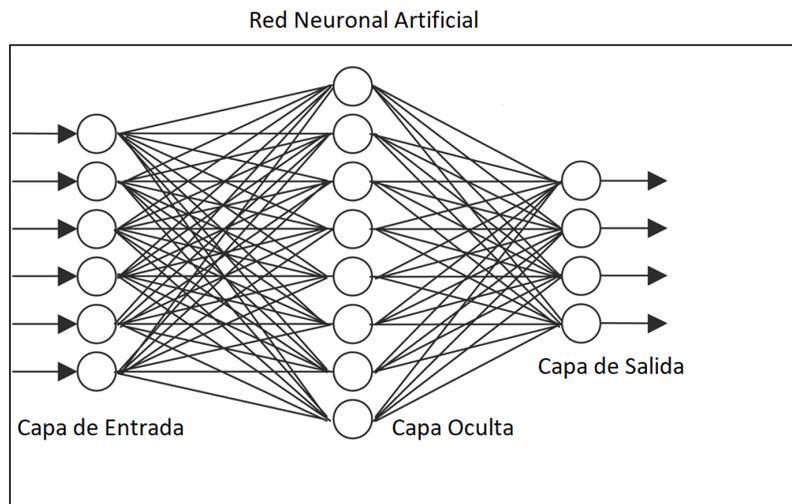


Figura 2.11: Red neuronal artificial.

El deep learning (aprendizaje profundo) es un sub campo específico del machine learning: una nueva toma de representaciones de aprendizaje a partir de datos que pone énfasis en el aprendizaje de capas sucesivas de cada vez más representaciones significativas. La profundidad del aprendizaje profundo no es una referencia a ningún tipo de comprensión más profunda lograda por el enfoque; más bien, representa esta idea de capas sucesivas de representaciones.

¿Cuántas capas contribuyen a un modelo de datos (lo cual es llamado la profundidad del modelo)? Otros nombres apropiados para el campo podrían haber sido aprendizaje de representaciones en capas y aprendizaje de representaciones jerárquicas. El moderno deep learning a menudo implica decenas o incluso cientos de capas sucesivas de representaciones: y todos aprenden automáticamente de la exposición a los datos de entrenamiento. Mientras tanto, otros enfoques del aprendizaje automático tienden a centrarse en aprender solo una o dos capas de representaciones de los datos; por lo tanto, a veces se les llama *aprendizaje superficial* [7].

2.2.3 Deep Learning en la clasificación de imágenes

En un problema de clasificación de imágenes, un modelo de aprendizaje profundo aprende las clases de imagen de manera incremental utilizando su arquitectura de capa oculta.

Primero, se extrae automáticamente las características de bajo nivel, como identificar regiones claras u oscuras y luego extrae características de alto nivel como bordes. Más tarde, extrae el nivel más alto de características, como formas, para que puedan clasificarse.

Cada nodo o neurona representa un pequeño aspecto de toda la imagen. Cuando se juntan representan toda la imagen. Son capaces de representar la imagen completamente. Además a cada nodo y cada neurona de la red se le asignan pesos. Estos pesos representan el peso real de la neurona con respecto a la fuerza de su relación con la salida. Estos pesos se pueden ajustar mientras se desarrolla el aprendizaje en los modelos.

2.2.4 Machine Learning Clásico versus Deep Learning

Extracción de características manualmente versus automatizadas: para resolver problemas de procesamiento de imágenes con las técnicas tradicionales de ML, las más importantes son en el paso de pre procesamiento es la función de extracción de características artesanal (por ejemplo, HOG y SIFT) para reducir la complejidad de una imagen y hacer patrones más visibles para que los algoritmos de aprendizaje funcionen. La mayor ventaja del aprendizaje profundo es que los algoritmos intentan aprender características de bajo y alto nivel del entrenamiento imágenes de manera incremental. Esto elimina la necesidad de extracción manual o ingeniería de las características.

Por partes versus solución de extremo a extremo: las técnicas tradicionales de ML resuelven el planteamiento del problema desglosando el problema, resolviendo diferentes partes primero y luego agregando los resultados para dar salida, mientras que en el aprendizaje profundo las técnicas resuelven el problema utilizando un enfoque de extremo a extremo. Por ejemplo, en un problema de detección de objetos, los algoritmos clásicos de ML como SVM (Máquina de Vectores de Soporte) requieren un algoritmo de detección de objetos del cuadro delimitador que primero identificará todos los posibles objetos que necesitarán tener HOG como entrada al algoritmo ML para reconocer los objetos correctos. Pero en un método de aprendizaje profundo, como YOLO la red, toma la imagen

como entrada y proporciona la ubicación y el nombre del objeto como salida. Claramente de principio a fin.

Tiempo de entrenamiento y hardware avanzado: a diferencia de los algoritmos tradicionales de ML, los algoritmos de aprendizaje profundo tardan mucho en entrenarse debido a la gran cantidad de parámetros y conjuntos de datos relativamente grandes. Por lo tanto, siempre debemos entrenar a un modelo de aprendizaje profundo en hardware de gama alta como por ejemplo una GPU y recordando entrenar para un tiempo razonable, ya que el tiempo es un aspecto muy importante en el entrenamiento eficaz de los modelos.

Adaptable y transferible: las técnicas clásicas de ML son bastante restrictivas considerando que las técnicas de aprendizaje profundo se pueden aplicar a una amplia gama de aplicaciones y varios dominios. Una gran parte se destina al aprendizaje de transferencia que nos permite usar redes profundas pre-entrenadas para diferentes aplicaciones dentro del mismo dominio. Por ejemplo en el procesamiento de imágenes, se utilizan redes previamente entrenadas para la clasificación de imágenes generalmente como un frontend para la extracción de características para detectar objetos y redes de segmentación.

Veamos ahora las diferencias entre un modelo de aprendizaje automático y un modelo de aprendizaje profundo cuando se usa en la clasificación de imagen (por ejemplo, imágenes de gato versus perro).

El machine learning tradicional tendrá la extracción de características y un clasificador para dar una solución a cualquier problema [10]:

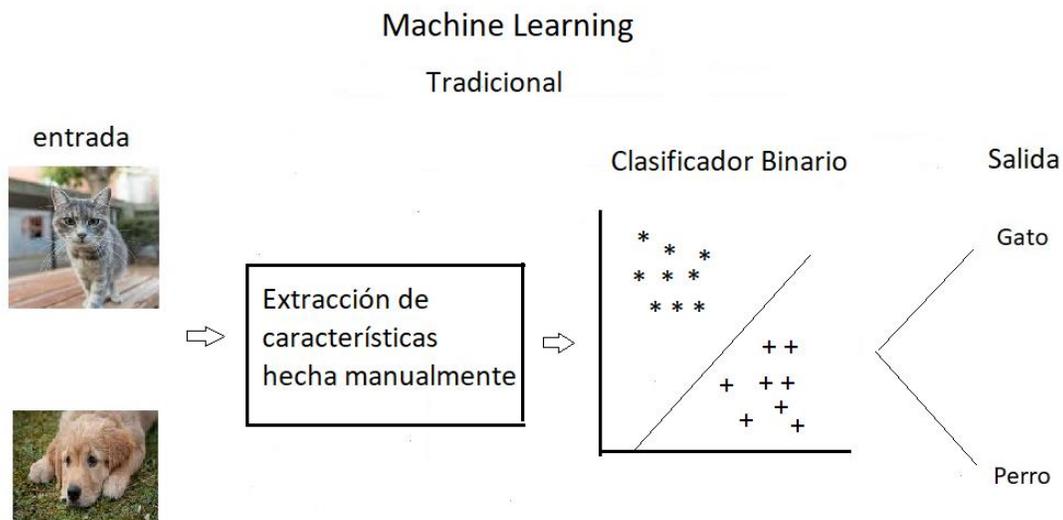


Figura 2.12: Machine Learning en el procesamiento para clasificar imágenes.

Con el aprendizaje profundo se puede ver las capas ocultas en la acción al hacer la decisión:

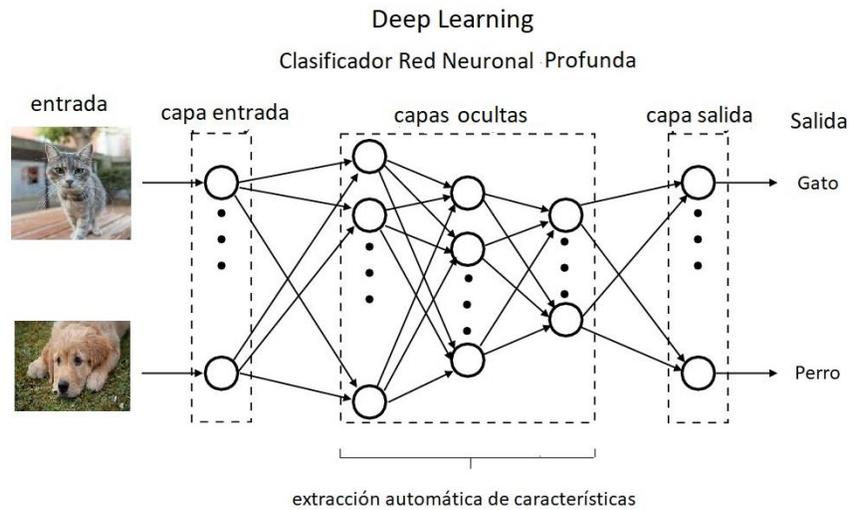


Figura 2.13: Deep Learning en el procesamiento para clasificar imágenes.

Como se mencionó anteriormente, si se tiene más datos, la mejor opción son redes profundas que rinden mucho mejor con bastantes datos (2.2.2 Deep Learning). Muchas veces, cuantos más datos se usan, más preciso es el resultado. El método clásico de aprendizaje automático necesita un conjunto complejo de algoritmos de ML y más datos solo van a dificultar su precisión. Los métodos complejos deben aplicarse para compensar la menor precisión. Además, incluso el aprendizaje se ve afectado: casi se detiene en algún momento cuando se agregan más datos de entrenamiento para entrenar el modelo [10].

Así es como se puede representar gráficamente:

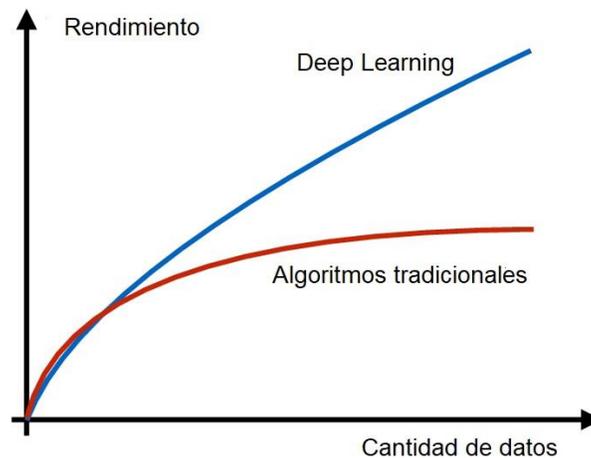


Figura 2.14: Rendimiento Deep Learning frente a otros algoritmos de aprendizaje.

2.2.5 CNNs (Redes Neuronales Convolucionales)

Las CNN son redes neuronales convolucionales profundas para las cuales la entrada utilizada principalmente son imágenes. Las CNNs aprenden de los filtros (características) que están diseñados a mano en algoritmos tradicionales. Esta independencia desde el conocimiento previo y el esfuerzo humano en el diseño de características es una gran ventaja. Los filtros también reducen el número de parámetros que se aprenderán con su arquitectura de pesos compartidos y poseen características de invariancia de traducción.

La figura 2.13 muestra la arquitectura típica de una CNN. Consiste en una o más capas convolucionales, seguida de una capa de activación ReLU (Rectified Linear Unit, Unidad Lineal Rectificada) no lineal, una capa de agrupación (Pool) y finalmente, una (o más) capas completamente conectada (FC), seguida de una capa FC softmax, por ejemplo, en el caso de una CNN diseñada para resolver un problema de clasificación de imágenes.

Puede haber múltiples capas de convolución ReLU agrupando secuencias de capas en la red, haciendo que la red neuronal sea más profunda y útil para resolver tareas complejas de procesamiento de imágenes, como se ve en el siguiente diagrama:

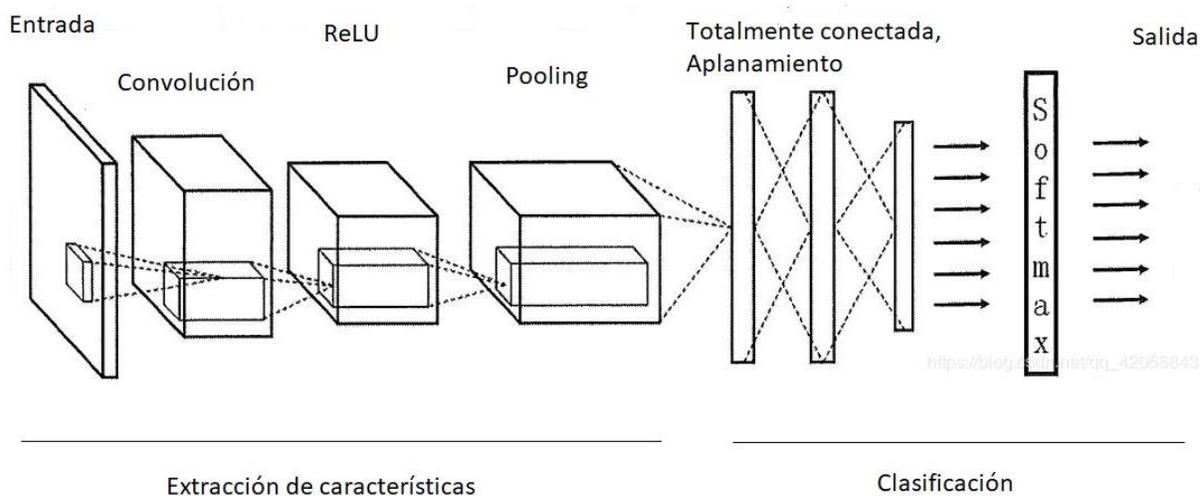


Figura 2.15: Procesamiento de imágenes con redes neuronales convolucionales.

Capa de convolución (convolution layer): El bloque de construcción principal de CNN es la capa convolucional. La capa convolucional consiste en una gran cantidad de filtros de convolución (núcleos). La convolución se aplica en la imagen de entrada utilizando un filtro de convolución para producir un mapa de características. En el lado izquierdo está la entrada a la capa convolucional; por ejemplo, la imagen de entrada. A la derecha está el filtro de convolución, también llamado núcleo (kernel). Como de costumbre, la operación de convolución se realiza deslizando este filtro sobre la entrada. En cada ubicación, la suma de la matriz de elementos de la multiplicación va al mapa de características. Una capa convolucional está representada por su ancho, altura (el tamaño de un filtro es

ancho x alto) y profundidad (número de filtros). Stride especifica cuánto se moverá el filtro de convolución en cada paso (el valor predeterminado es 1).

El relleno (padding) se refiere a las capas de ceros que rodean la entrada (generalmente se usa para mantener tamaño de imagen de entrada y salida igual). La figura 2.16 muestra cómo se aplican los filtros de convolución 3 x 3 en una imagen.

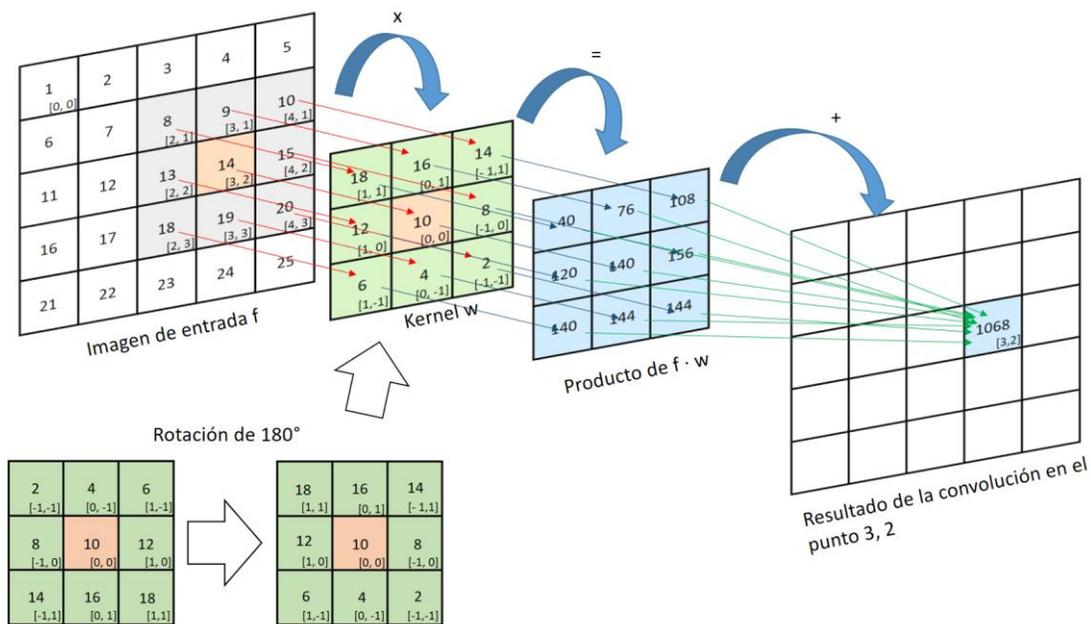


Figura 2.16: Aplicación de una operación de convolución.

Capa de agrupación (pooling layer): Después de una operación de convolución, generalmente se realiza una operación de agrupamiento para reducir la dimensionalidad y la cantidad de parámetros a aprender, lo que acorta el tiempo de capacitación, requiere menos datos para entrenar y combate el sobreajuste. Las capas de agrupación disminuyen cada uno de los mapas de características de forma independiente, reduciendo la altura y el ancho, pero manteniendo la profundidad intacta. El tipo más común de agrupación es la agrupación máxima (max pooling), que solo toma el máximo valor en la ventana de agrupación. Contrariamente a la operación de convolución, la agrupación no tiene parámetros. Esta desliza una ventana sobre su entrada y simplemente toma el valor máximo en la ventana. Similar a una convolución, se puede especificar el tamaño de la ventana y el paso (stride) para la agrupación.

No linealidad - capa ReLU (non linearity – ReLU layer): Para que cualquier tipo de red neuronal sea poderosa, debe tener no linealidad. El resultado de la operación de convolución, por lo tanto, se pasa a través de la función de activación no lineal. La activación de ReLU se usa en general para lograr la no linealidad (y para combatir el problema de la desaparición de gradiente con activación sigmoidea). La función ReLU transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran.

Capa FC (FC layer): Después de las capas convolucionales y de agrupación, generalmente se agregan un par de capas FC a concluir la arquitectura CNN. La salida de las capas convolucionales y de agrupación son volúmenes 3D, pero una capa FC espera un vector 1D de números. Entonces, la salida de la agrupación final la capa debe aplanarse a un vector, y eso se convierte en la entrada a la capa FC. Aplanar es simplemente organizar el volumen 3D de números en un vector 1D.

Dropout: Es la técnica de regularización más popular para redes neuronales profundas. El *dropout* es utilizado para evitar el sobreajuste, y generalmente se usa para aumentar el rendimiento (precisión) de la tarea de aprendizaje profundo en el conjunto de datos invisible. Durante el tiempo de entrenamiento, en cada iteración, una neurona se desactiva con cierta probabilidad, p . Esto significa que todas las entradas y salidas a esta neurona se desactivará en la iteración actual. Este hiperparámetro p se llama tasa de abandono, y generalmente es un número alrededor de 0.5, correspondiente al 50% de las neuronas abandonadas [10].

2.2.6 Detección de objetos basado en Deep Learning

Con los recientes avances de las redes neuronales convolucionales y su desempeño en la clasificación de imágenes, se ha vuelto intuitivo usar el modelo de estilo similar para la detección de objetos. Esto ha sido probado con buenos resultados (acápites 4.2.2.2.1 Uso de IA para la clasificación de imágenes: Aprendizaje Profundo). En los últimos años surgieron mejores detectores de objetos que aumentan la precisión general en puntos de referencia estándar. Algunos de los estilos de detectores ya están en uso en teléfonos inteligentes, automóviles autónomos robotizados, etc.

Una CNN genérica genera probabilidades de clase, como en el caso del reconocimiento de imágenes. Pero para detectar objetos, estos (objetos) deben modificarse para generar tanto la probabilidad de clase como el cuadro rectángulo delimitador coordenadas y forma. La detección temprana de objetos basada en CNN calcula posibles ventanas desde una imagen de entrada y luego calcula características usando una CNN modelo para cada ventana. Esta salida del extractor de características CNN nos dirá si la ventana elegida es el objeto de destino o no. Esto es lento debido a un gran cálculo de cada ventana a través del extractor de características CNN. Intuitivamente, nos gustaría extraer características de imágenes y usar esas funciones para la detección de objetos. Esto no solo mejora la velocidad para detección pero también filtra el ruido no deseado en la imagen [13].

Se han propuesto varios métodos para abordar estos problemas de velocidad y precisión en la detección de objetos. En general, se dividen en dos categorías principales:

Detectores de dos etapas: aquí, el proceso general se divide en dos pasos principales, de ahí el nombre de detectores de dos etapas. El más popular entre estos es RCNN más rápido. Más abajo se brindará una explicación detallada de este método.

Detectores de una etapa: mientras que los detectores de dos etapas aumentaron la precisión para la detección, aún eran difíciles de entrenar y eran más lentos para varias operaciones en tiempo real. Los detectores de una etapa rectificaron estos problemas haciendo una red en una sola arquitectura que

predice más rápido. Uno de los modelos populares de este estilo es detector multibox de disparo único (SSD).

En las siguientes líneas, veremos con más detalle el detector de dos etapas.

Detectores de dos etapas: A medida que las CNN muestran su desempeño en la clasificación general de imágenes, los investigadores usaron la misma CNN para hacer una mejor detección de objetos. Los enfoques iniciales que utilizan el aprendizaje profundo para objetos la detección se puede describir como detectores de dos etapas y uno de los más populares es Faster RCNN de Shaoqing Ren, Kaiming He, Ross Girshick y Jian Sun (2015) [24].

El método se divide en dos etapas:

En la primera etapa, las características se extraen de una imagen y región de interés propuesta (ROI). El ROI consiste en un posible cuadro donde un objeto podría estar en la imagen.

La segunda etapa utiliza características y ROI para calcular los cuadros delimitadores finales y probabilidades de clase para cada una de las cajas. Estos juntos constituyen la salida final.

Una descripción general de Faster-RCNN se muestra en la siguiente figura. Se utiliza una imagen de entrada para extraer características y propuestas de una región. Se utilizan estas características y propuestas extraídas juntos para calcular los cuadros delimitadores predichos y las probabilidades de clase para cada cuadro:

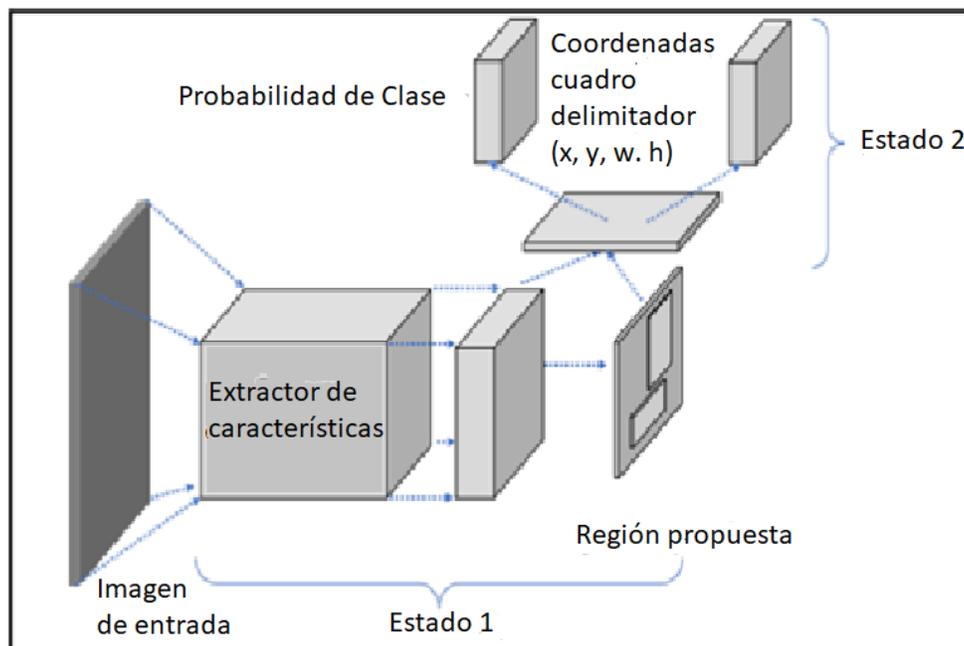


Figura 2.17: Faster-RCNN descripción general.

Como se muestra en la figura anterior, el método general se considera en dos etapas porque durante el entrenamiento el modelo primero aprenderá a producir ROI utilizando un sub modelo llamado Región Red de Propuestas (RPN). Luego aprenderá a producir probabilidades de clase correctas y ubicaciones de cuadro delimitador utilizando ROI y características. Una visión general de RPN es como se muestra en la figura 2.18. La capa RPN usa la capa de entidades ya que la entrada crea una propuesta para delimitar las cajas y probabilidades correspondientes:

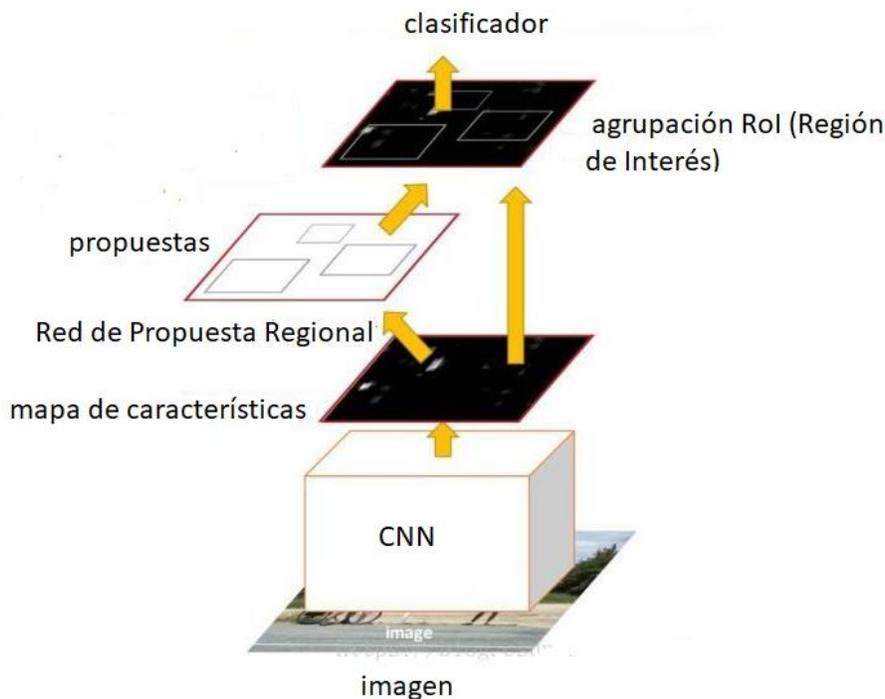


Figura 2.18: RPN Red de región propuesta.

Las ubicaciones del cuadro delimitador suelen ser valores normalizados para la coordenada superior izquierda del cuadro con valores de ancho y alto, aunque esto puede cambiar dependiendo de la forma en que el modelo es aprendido durante la predicción, el modelo genera un conjunto de probabilidades de clase, categorías de clase así como la ubicación del cuadro delimitador en formato (x, y, w, h). Este conjunto vuelve a pasar a través de un umbral para filtrar los cuadros delimitadores con puntajes de confianza inferiores al umbral.

La principal ventaja de usar este estilo de detector es que proporciona una mayor precisión que detectores de una etapa. Por lo general, estos logran una precisión de detección de vanguardia. Sin embargo sufren de velocidades más lentas durante las predicciones. Si para una predicción de una aplicación, el tiempo juega un papel crucial, entonces se aconseja proporcionar a estas redes un sistema de alto rendimiento o utilizar detectores de una etapa. Por otro lado, si el requisito es obtener la mejor precisión, se recomienda utilizar un método de este tipo para la detección de objetos [13][19].

Un ejemplo de salida de detección de objetos es como se muestra en la siguiente figura con el cuadro delimitador alrededor de los objetos detectados. Cada cuadro tiene una etiqueta que muestra el nombre de clase previsto y confianza para la caja:

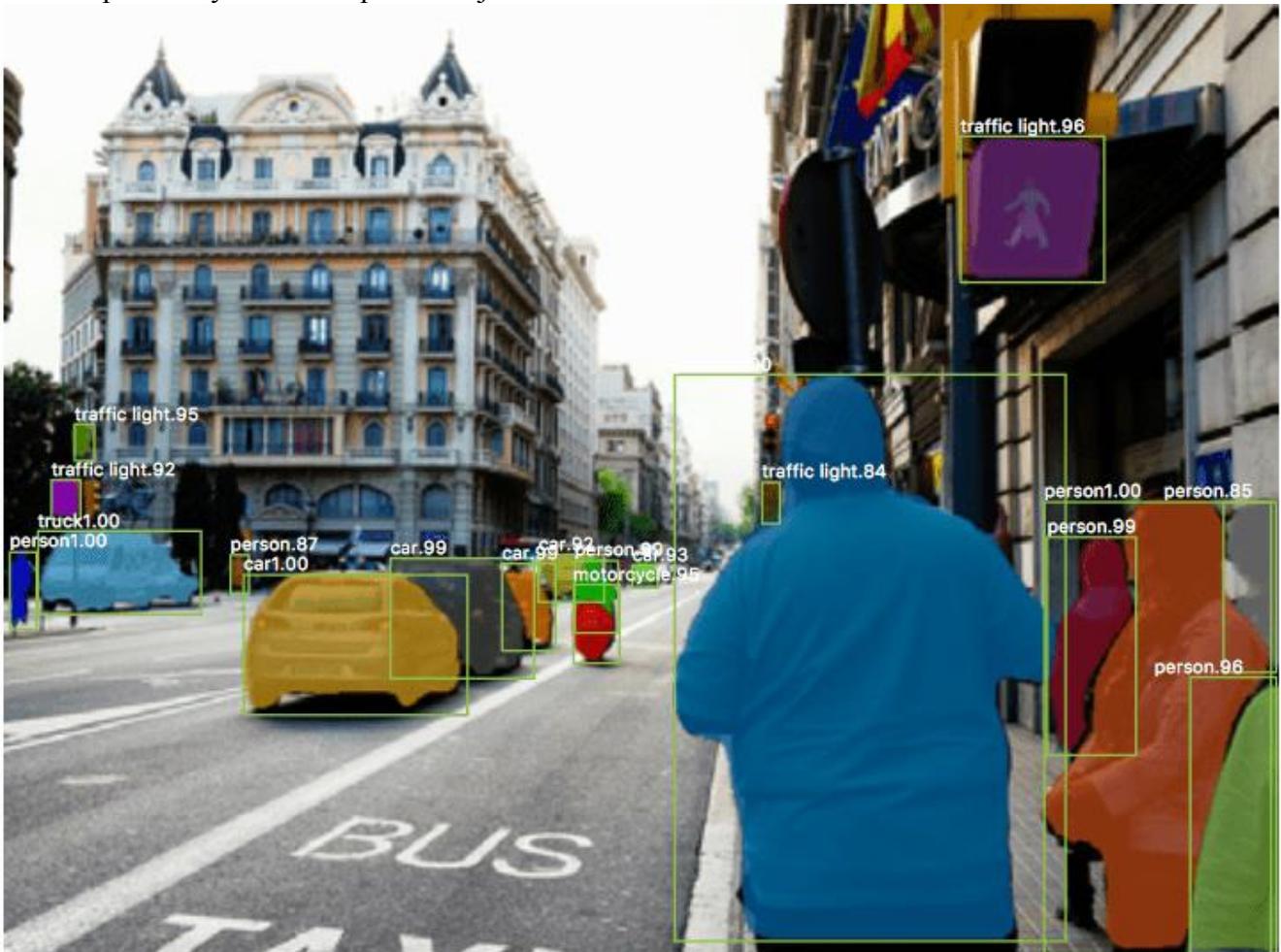


Figura 2.19: Detección de objetos asociada con su probabilidad.

La detección en la captura de pantalla anterior utiliza un modelo RCNN más rápido e incluso para pequeños objetos, como una persona en la parte inferior derecha, el modelo detecta con un buen puntaje de confianza. Los objetos detectados en general son autobuses, automóviles y personas. El modelo no detecta otros objetos, como árboles, postes, semáforos, etc. porque no ha sido entrenado para detectar esos objetos [13].

Capítulo 3: Deep Learning en el marco de los procesos de Ingeniería de Software

Existe una clara falta de herramientas que funcionen bien y mejores prácticas para construir sistemas DL en las tres áreas de desarrollo, producción, y desafíos organizacionales.

En comparación con otras áreas como la ingeniería de software o tecnologías de bases de datos, está claro que DL todavía es bastante inmadura y necesita más trabajo para facilitar el desarrollo de sistemas de calidad [15].

Deep Learning ha recibido considerable atención en los últimos años debido a su éxito en áreas como la visión de computadora (por ejemplo, reconocimiento de objetos, clasificación de imágenes, sistemas de recomendación, etc.) utilizando redes neuronales convolucionales, procesamiento de lenguaje natural. Una de las principales diferencias con los métodos tradicionales de aprendizaje automático (ML) es que DL aprende automáticamente cómo representar datos usando múltiples capas de abstracción, en ML tradicional, una cantidad significativa de trabajo tiene que ser empleado en "ingeniería de características" para construir esta representación manualmente, pero este proceso ahora puede automatizarse a un nivel superior. Tener un método automatizado y basado en datos para aprender a representar datos mejora tanto el rendimiento del modelo y reduce los requisitos para el trabajo de ingeniería de características [15].

Dados los recientes avances en ML, también se ha visto que la industria comienza a aprovechar cada vez más estas técnicas, especialmente en grandes empresas tecnológicas como Google, Apple y Facebook. Google aplica técnicas de DL a las cantidades masivas de datos recopilados en servicios como Google Traductor, reconocimiento de voz de Android, Street View de Google y su servicio de búsqueda. El asistente personal virtual de Apple Siri ofrece una variedad de servicios como informes meteorológicos, deportes, noticias y preguntas genéricas respondiendo utilizando técnicas como DL [15].

Esta adopción está impulsada por desarrollos en infraestructura, especialmente en términos de utilización de la potencia de cálculo. Ha desbloqueado el potencial del aprendizaje profundo y el aprendizaje automático. Se puede observar avances en el aprendizaje profundo correlacionados con desarrollos de computación [25].

En la figura 3.1 se puede observar los avances en el aprendizaje profundo correlacionados con el desarrollo de la computación (fuente: OpenAI, <https://openai.com/blog/ai-and-compute/>).

Dos eras distintas del uso de la computación en el entrenamiento de Sistemas de Inteligencia Artificial

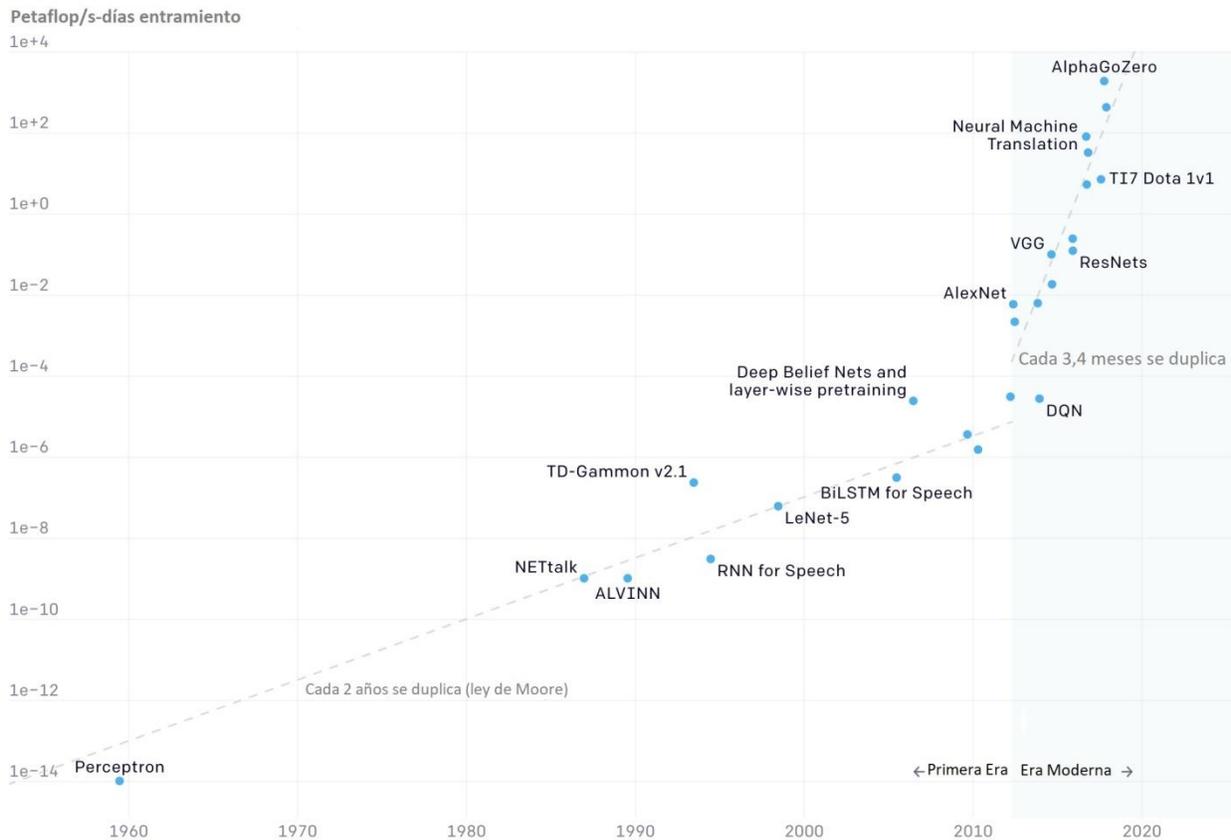


Figura 3.1: Demanda de aprendizaje profundo a lo largo del tiempo respaldada por la computación.

3.1 Machine Learning y Software Engineering

El aprendizaje automático, especialmente DL, difiere en parte de la tradicional ingeniería de software (SE) en que su comportamiento es fuertemente dependiente de datos del mundo externo. De hecho, es en estas situaciones donde ML se vuelve útil. Una diferencia clave entre sistemas ML y sistemas no ML es que los datos en parte reemplazan al código en un sistema ML y se utiliza un algoritmo de aprendizaje para identificar automáticamente patrones en los datos en lugar de escribir reglas codificadas. Esto sugiere que los datos deben ser probados tan minuciosamente como el código, pero actualmente hay una falta de mejores prácticas sobre cómo hacerlo [15].

La causa principal es que existe una diferencia fundamental entre el desarrollo de ML y desarrollo de software tradicional: el aprendizaje automático no es solo código; es código más datos. El modelo ML

se crea aplicando un algoritmo (mediante código) para ajustar los datos y dar como resultado un modelo ML [25].



Figura 3.2: Machine Learning = Datos + Código.

Se ha realizado una cantidad significativa de investigación sobre prueba de software y también para probar el rendimiento de ML. Sin embargo, la intersección de SE y ML no ha sido estudiada completamente. No es solo el modelo correcto que necesita pruebas, pero también la implementación de un sistema ML listo para producción [15].

3.2 Deuda técnica

Deep Learning también hace posible componer complejos modelos de un conjunto de sub modelos y potencialmente reutilizan parámetros pre entrenados con las llamadas técnicas de "transfer learning" (aprendizaje de transferencia). Esto no solo agrega dependencias adicionales en los datos, pero también en modelos externos que pueden entrenarse por separado y también puede cambiar en la configuración con el tiempo [15].

Los modelos externos agregan deuda de dependencia adicional y costo de propiedad para sistemas de ML. La deuda de dependencia se señala como una de las claves contribuyentes a la deuda técnica en SE [15].

No es raro que el código de soporte y la infraestructura incurran en una deuda técnica significativa, y se debe tener cuidado para planificar adecuadamente el tiempo necesario para desarrollar apoyo al código de tubería e infraestructura [15].

Un sistema maduro podría terminar con el 95% del código siendo código de pipelines y pegamento, que conecta principalmente diferentes bibliotecas de ML y paquetes. Un sistema de ML es generalmente dependiente de muchas tuberías diferentes que también pueden ser implementadas en diferentes lenguajes de programación, formatos y sistemas de infraestructura. No es raro que las tuberías de procesamiento cambien, agreguen y eliminen campos, o quedar en desuso. Mantener desplegado un sistema ML listo para producción actualizado con todos estos cambios requiere una

cantidad significativa de trabajo y requiere apoyo de sistemas de monitoreo y registro para poder detectar cuándo estos cambios ocurren [15].

Otra mala práctica común en los sistemas de ML es tener rutas de código experimental. Esto es similar a las death flags (señal de que no funciona) en el software tradicional. Con el tiempo, estas rutas de código acumuladas crean una deuda creciente que será difícil de mantener [15].

Un mal ejemplo de rutas de código experimentales fue Knight, el sistema de capital pierde \$ 465 millones en 45 minutos, aparentemente debido a un comportamiento inesperado de experimentos obsoletos de rutas de código [15].

Otro problema técnico de la deuda de los sistemas ML es la gestión de la configuración. La configuración de los sistemas ML a menudo no tiene el mismo nivel de control de calidad que el código del software, a pesar de que ambos pueden tener un mayor impacto en el rendimiento del modelo y también ser de gran tamaño [15].

Los cambios de configuración deben revisarse e incluso probarse de la misma manera como cambia el código [15].

La necesidad de ajustar las prácticas de ingeniería de software en la era reciente se ha discutido en el oculto contexto de la deuda técnica y resolución de problemas de IA integrativa. Se identifica varios aspectos de la arquitectura del sistema de ML y los requisitos que deben tenerse en cuenta durante diseño de sistemas. Algunos de estos aspectos incluyen bucles de comentarios ocultos, entrelazamiento de componentes y límites erosionados, propagación de errores no monotónicos, estados de calidad continuos y desajustes entre el mundo real y los conjuntos de evaluación. En los últimos años, se han realizado múltiples esfuerzos en la industria para automatizar este proceso. mediante la creación de marcos y entornos para respaldar el aprendizaje automático flujo de trabajo y su naturaleza experimental. Sin embargo, las investigaciones y las encuestas en curso muestran que los ingenieros todavía luchan por operacionalizar y estandarizar los procesos de trabajo [26].

3.3 Desafíos de desarrollo

Existen diferencias fundamentales entre desarrollar sistemas ML comparados con los sistemas SE tradicionales. Una de las principales diferencias es, que los datos se utilizan para programar el sistema "automáticamente" en lugar de escribir el código de software manualmente [15].

El rendimiento del sistema es desconocido hasta que se haya probado con los datos proporcionados, lo que dificulta la planificación del proyecto de manera estructurada [15].

Además, la falta de transparencia del modelo, la incapacidad de comprender modelos grandes y complicados, la dificultad en la depuración mediante bibliotecas con ejecución lenta hace que sea difícil estimar el esfuerzo necesario para completar el proyecto [15].

La necesidad y el deseo de más automatización e inteligencia han llevado a avances en el aprendizaje automático (ML) e inteligencia artificial (AI), sin embargo, todavía se experimenta fallas y deficiencias en los sistemas de software resultantes. La principal razón es el cambio en el paradigma de desarrollo inducido por ML e IA. Tradicionalmente, se construyen los sistemas de software deductivamente, escribiendo las reglas que gobiernan el comportamiento del sistema como código de programa. Sin embargo, con las técnicas de AI, estas reglas son inferidas de los datos de entrenamiento (de que los requisitos se generan de forma inductiva). Este paradigma hace que el cambio de razonar sobre el comportamiento de los sistemas de software sea difícil con los componentes de ML, lo que resulta en sistemas de software que son intrínsecamente difíciles de probar y verificar [27].

Dada la crítica y el creciente papel de los sistemas basados en ML y AI en nuestra sociedad, es imperativo por tanto para la ingeniería de software (SE) y comunidades de AI para investigar y desarrollar enfoques innovadores para hacer frente a estos desafíos. De hecho, el comportamiento aprendido basado en un sistema de ML puede ser incorrecto, incluso si el algoritmo de aprendizaje se implementa correctamente, una situación en que técnicas de prueba tradicionales son ineficaces. Un problema crítico es cómo desarrollar, probar y evolucionar tales sistemas, dado que ellos no tienen especificaciones (completas) ni el código fuente correspondiente a algunos de sus comportamientos críticos [27].

3.3.1 Gestión de experimentos

Durante el desarrollo de modelos de ML, generalmente se realizan una gran cantidad de experimentos para identificar el modelo óptimo. Cada experimento puede diferir de otros experimentos en varias formas y es importante para garantizar resultados reproducibles para estos experimentos.

Para tener resultados reproducibles, puede ser necesario conocer la versión exacta de componentes como:

1. Hardware (por ejemplo, GPU principalmente modelos)
2. Plataforma (por ejemplo, sistema operativo y paquetes instalados)
3. Código fuente (por ejemplo, modelo de entrenamiento y pre procesamiento)
4. Configuración (por ejemplo, configuración del modelo y ajustes de pre procesamiento)
5. Datos de entrenamiento (por ejemplo, señales de entrada y valores función objetivo)
6. Estado del modelo (por ejemplo, versiones de modelos entrenados).

El control de versiones para sistemas ML agrega una serie de desafíos en comparación con el desarrollo de software tradicional, especialmente dado el alto nivel de dependencia de datos en los sistemas ML.

Las diferentes versiones de datos producirán resultados diferentes y los datos de entrada son a menudo un conglomerado de datos de múltiples fuentes de datos heterogéneas. Se ha argumentado que uno de los componentes más difíciles de rastrear son los componentes de los datos, y el costo y el almacenamiento de datos versionados puede ser muy alto.

Además, se crean diferentes versiones del modelo durante el entrenamiento del modelo DL, cada una con diferentes parámetros y métricas que deben medirse y rastrearse adecuadamente.

Con la adición de dependencias de datos y un alto grado de parámetros de configuración, puede ser muy difícil mantener adecuadamente los sistemas de ML a largo plazo. Además, no es raro realizar ajustes de hiperparámetros de modelos, potencialmente haciendo uso de la optimización automatizada, métodos que generan cientos de versiones de los mismos datos y modelo pero con diferentes parámetros de configuración. El aprendizaje profundo también puede agregar el requisito de hardware específico.

3.3.2 Transparencia limitada

La ingeniería del software se basa sobre el principio de reducir sistemas complejos a pequeños bloques más simples siempre que sea posible, es deseable agrupar bloques en diferentes niveles de abstracción que tienen un significado conceptual similar. Aunque los sistemas DL, en principio hacen esto automáticamente es muy difícil saber exactamente cómo es realizado o predecir cómo se verán las capas de abstracción una vez que el modelo ha sido entrenado. Además, es difícil aislar un área funcional específica u obtener una comprensión semántica del modelo. Esto solo se puede realizar con métodos aproximados [15].

Los grandes avances que se han hecho en campos como la visión por computadora y el reconocimiento de voz se han logrado mediante la sustitución de una tubería de procesamiento modular con grandes redes neuronales que se entrenan de extremo a extremo. En esencia, la transparencia se intercambia por precisión. Esto es una realidad inevitable. Si el problema fue lo suficientemente simple como para ser explicado en lógica simbólica, entonces no habría necesidad de complejos sistemas para resolverlo. Sin embargo, si el problema es complejo y el modelo es inherentemente irreducible, una explicación precisa del modelo será tan complejo como el modelo mismo [15].

3.3.3 Solución de problemas

Un gran desafío en el desarrollo de sistemas DL son las dificultades para estimar los resultados antes de que el sistema haya sido entrenado y probado. Además, la escasa comprensión del funcionamiento interno de las redes neuronales complejas hace que sea difícil depurarlos de forma tradicional. En una red neuronal la estructura combina las partes funcionales, la memoria y también se puede distribuir en múltiples máquinas [15].

Además, el uso de bibliotecas como TensorFlow potencialmente combinadas con frameworks de big data como Apache Spark hace que sea difícil solucionar y depurar problemas en el código. Como ambos tienen un gráfico de ejecución perezosa, donde el código no se ejecuta en orden imperativo, puede ser difícil solucionar los errores utilizando las herramientas SE tradicionales [15].

Otros Frameworks como PyTorch, no tienen el problema de la pereza del gráfico de ejecución, pero tiene otros problemas como una baja tasa de adopción en la comunidad de IA [15].

Incluso si fuera posible recorrer el código fuente y establecer puntos de interrupción, la evaluación manual es en la práctica, a menudo imposible ya que implicaría la inspección de millones de parámetros en comparación con el SE tradicional, un pequeño error en el código fuente no se puede detectar ni en tiempo de compilación ni en tiempo de ejecución. Durante el entrenamiento, la única información que el desarrollador o el científico de datos puede tener es una estimación del error global. Las redes neuronales son tan grandes que a veces tardan días o incluso semanas en entrenarse, no hay forma de garantizar que alguna vez se alcanzará cierto nivel de rendimiento. Por lo tanto, el esfuerzo requerido para construir redes neuronales puede llegar a ser muy grande [15].

3.3.4 Limitaciones de recursos

Trabajando con datos que requieren sistemas distribuidos agrega otra magnitud de complejidad en comparación con las soluciones de máquinas individuales. No es solo el volumen de los datos que pueden requerir una solución distribuida, también las necesidades computacionales para extraer, transformar datos, entrenar, evaluar el modelo, y colocarlo en producción. Para sistemas DL, también puede ser que la GPU tenga una memoria limitada que requiere técnicas especiales para dividir el modelo a través de múltiples GPU [15].

Trabajar con sistemas distribuidos, tanto para el procesamiento de datos con Apache Spark y entrenamiento DL con TensorFlow distribuido o TensorFlowOnSpark agrega complejidad en varias dimensiones. Esto no solo requiere un conocimiento adicional y tiempo para operarlos, sino también gestión adicional, costo de hardware y software asociado [15].

3.3.5 Pruebas (Testing)

Los sistemas de ML requieren pruebas del software utilizado para construir tuberías de datos, modelos de entrenamiento y servir en producción. Dada la alta dependencia de datos de los sistemas ML, los datos también necesitan ser probados. Sin embargo, actualmente solo existen unas pocas herramientas de prueba de datos en comparación con las pruebas de software. Un patrón frecuente es visto cuando se prueban los datos para hacer uso de una pequeña muestra del conjunto completo de datos. Es un desafío proporcionar una muestra que incluye todos los casos límite que pueden existir en el conjunto de datos completo. Además, como el mundo externo es dinámico y cambia con el tiempo, nuevos casos límite continuarán apareciendo más adelante en el tiempo.

Además la naturaleza no determinista de muchos algoritmos de entrenamiento hacen que la prueba de modelos sea aún más desafiante. Otro desafío puede ser el procesamiento y el modelo de datos en el servidor en modo de producción puede diferir en la implementación en comparación con el modo de

entrenamiento o prueba, lo que provoca un sesgo de entrenamiento en el rendimiento del modelo. Tener las pruebas (testing) adecuadas en el lugar se vuelven cruciales [15].

3.4 Desafíos de producción

Particularmente para DL, es importante aprovechar el último hardware. Esto genera un desafío significativo para mantener la frecuencia de las actualizaciones de software y las dependencias asociadas a una gestión de vanguardia para detectar con precisión los problemas introducidos por el cambio de comportamiento en las dependencias, incluidas las fuentes de datos que han sido modificadas, requiere una cuidadosa e inteligente supervisión [15].

Más interesante, ya que los sistemas ML a menudo se usan directamente por los usuarios finales, el modelo puede causar un cambio en la realidad que trata de entender. Por lo tanto, puede introducir una retroalimentación oculta en el ciclo entre los datos de producción y entrenamiento utilizados por el modelo [15].

3.4.1 Gestión de dependencias

La SE tradicionalmente se basa en el supuesto de que el hardware es en el mejor de los casos, un problema y en el peor una entidad estática que debe tenerse en consideración. Los sistemas DL están entrenados principalmente en GPUs dado que éstas proporcionan una aceleración de 40-100x sobre las CPUs clásicas. En los últimos 5 años, el rendimiento ha mejorado significativamente y las nuevas GPU se lanzan 1 o 2 veces al año [15].

Las plataformas de software DL se actualizan continuamente, a veces a diario y las actualizaciones suelen dar lugar a mejoras notables. Esto funciona bien para académicos, investigadores y para desarrollar pruebas de concepto pero puede causar problemas considerables para sistemas listos para producción. A diferencia de otros métodos de ML, DL a menudo escala directamente con el tamaño del modelo y cantidad de datos, en que los tiempos de entrenamiento pueden ser muy largos (generalmente desde unos pocos días hasta varias semanas) hay una motivación muy fuerte para maximizar el rendimiento utilizando el último software y hardware [15].

Cambiar el hardware y el software no solo puede causar problemas para poder mantener resultados reproducibles, también puede incurrir en costos de ingeniería significativos al mantener software y hardware al día.

3.4.2 Monitoreo y registro

Construyendo un ejemplo de juguete de un sistema ML o incluso un prototipo de investigación fuera de línea permite apreciar la cantidad de trabajo requerido para construir un sistema ML. En aplicaciones de ML del mundo real más allá del ejemplo de juguete puede ser difícil cubrir todos los casos límite, eso puede ocurrir una vez que un modelo se ha implementado en producción. También es común que las personas no reconozcan el esfuerzo necesario para mantener un sistema de ML desplegado a lo largo del tiempo [15].

Un sistema de ML puede volverse a entrenar con frecuencia y, por lo tanto, cambiar su comportamiento de forma autónoma. Como el comportamiento de los cambios del mundo externo, el comportamiento del sistema ML puede cambiar repentinamente sin ninguna acción humana en el "control" del sistema. En esta situación, las pruebas unitarias y las pruebas de integración son valiosas pero no es suficiente para validar el rendimiento del sistema. Los antiguos umbrales (valores críticos) que pueden haber sido asignados manualmente puede que ya no sean válidos dadas las variaciones en los datos del mundo externo [15].

La monitorización en vivo del rendimiento del sistema puede ayudar, pero elegir qué métricas monitorear puede ser un desafío [15].

3.4.3 Bucles de retroalimentación involuntaria

Un sistema ML es por definición siempre abierto ya que está impulsado por datos externos. No importa cuán cuidadosamente se diseñe y pruebe el modelo, su rendimiento final siempre dependerá mucho de los datos externos [15].

Además, especialmente en modelos implementados en el contexto Big Data (como los sistemas ML a menudo lo son), existe el riesgo de crear un ciclo de retroalimentación no intencionado donde los sistemas del mundo real se adaptan al modelo en lugar de ser al revés. Imaginemos tener un sistema de predicción de precios inmobiliarios muy utilizado. Cuando tal sistema se vuelve suficientemente popular, las predicciones pueden fácilmente convertirse en una profecía auto cumplida. Como se sabe por teoría de control, aparte de casos especiales controlados, los bucles de retroalimentación positiva son inherentemente inestables [15].

3.4.4 Código de pegamento y sistemas de soporte

Desafortunadamente la propiedad de los sistemas ML, y especialmente los sistemas DL, son solo una pequeña parte del sistema que se ocupa del modelo. en un sistema listo para producción, solo el 5% del código puede tratar el modelo y el resto es un "código de pegamento" que interactúa con los sistemas de soporte y pega bibliotecas y sistemas juntos [15].

El uso de servicios en la nube puede mejorar enormemente el tiempo de desarrollo y disminuir las necesidades de mantenimiento. Sin embargo, mantener el "código de pegamento" actualizado y mantenerse al día con los cambios externos en la nube de servicios, pueden introducir desafíos inesperados en sistemas listos para producción [15].

3.5 Desafíos organizacionales

Para poner en producción un modelo de ML generalmente se requiere la colaboración entre muchos equipos diferentes con diversas ideas, prioridades y valores culturales. Esto no solo introduce desafíos organizacionales desde un punto de vista cultural, pero también en poder estimar adecuadamente la cantidad de esfuerzo necesario por los diferentes tipos de equipos [15].

3.5.1 Estimación de esfuerzo

El diseño SE reduccionista modular de un proyecto que no es de ML hace que sea mucho más fácil estimar el tiempo y los recursos necesarios para completarlo. En un proyecto de ML, el objetivo también podría estar bien definido, pero no está claro hasta qué punto un modelo aprendido alcanzará ese objetivo y un número desconocido de varias iteraciones se necesitarán antes de alcanzar resultados con niveles aceptables. Esto está en la naturaleza de cualquier proyecto de investigación. También puede ser difícil disminuir el alcance y ejecutar el proyecto en una configuración basada en el tiempo, con una fecha de entrega predefinida, es difícil desde ya determinar cuándo un rendimiento aceptable se logrará [15].

Una complicación adicional es la falta de transparencia inherente en muchos modelos ML, especialmente en DL con potentes modelos pero muy complejos y poco entendidos. Desde entonces, en muchos casos, no hay una manera fácil de entender cómo funciona un modelo y es muy difícil de entender cómo se debe modificar para alcanzar mejores resultados [15].

3.5.2 Privacidad y seguridad de los datos

La falta de comprensión del funcionamiento interno de una gran red neuronal puede tener graves implicaciones para la privacidad y la seguridad de los datos. El conocimiento en la red neuronal se almacena de manera distribuida en todos los pesos de la red. Aunque se sabe que la especialización ocurre en regiones específicas de la red neuronal, su mecanismo exacto es poco entendido. Por lo tanto es muy difícil para los diseñadores controlar dónde y cómo se almacena la información. Eso tampoco es raro que las empresas tengan términos de acuerdos de servicios con sus usuarios finales que les impiden usar datos sin procesar como entrada directa a un modelo ML y en su lugar tienen que hacer uso de estadísticas y / o agregar datos de usuario anónimos. Esto no solo puede reducir el rendimiento

en el modelo, pero también puede requerir realizar tareas como la exploración de datos y solucionar problemas más difíciles. Nuevas regulaciones como el Reglamento General Europeo de Protección de Datos para hacer todo lo posible para mantener los datos seguros y proteger la privacidad. Sin embargo, si bien es importante mantener los datos seguros, también agrega desafíos importantes sobre cómo desarrollar y gestionar sistemas de ML [15].

Aunque la información en un modelo está oscurecida y hay no una forma trivial de transformarlo de nuevo en información legible para los humanos, no es imposible hacerlo. Ha habido algunos trabajos para preservar la seguridad y la privacidad de los datos, por ejemplo, privacidad diferencial, anonimato k y el cifrado de redes que utilizan cifrado homomórfico para mantener datos confidenciales seguros [15].

Sin embargo, se necesita más trabajo para preservar la privacidad y seguridad de conjuntos de datos confidenciales sin dejar de ser eficiente al realizar exploración de datos, desarrollar modelos y solucionar problemas [15].

3.5.3 Diferencias culturales

Construir un sistema ML listo para producción generalmente implica una colaboración entre personas con diferentes roles. Un científico de datos podría ser "pragmático" sobre su código siempre que logre los resultados deseados en un ambiente controlado, mientras que los miembros del equipo de ingeniería se preocupan mucho más por el mantenimiento y la estabilidad. Transformando un prototipo inicial en un sistema listo para producción que también interactúa con el backend y la interfaz de los sistemas existentes que generalmente requieren un esfuerzo significativamente mayor. Esto normalmente incluye la colaboración con los ingenieros del backend, los diseñadores de experiencia de usuario y los propietarios de productos [15].

No es raro que la cultura, las habilidades y las áreas de interés difieran entre estas personas. Por ejemplo, los científicos de datos pueden carecer de habilidades de SE y comprender las buenas prácticas de ingeniería. Los diseñadores de UX (User eXperience, experiencia de usuario, es aquello que una persona percibe al interactuar con un producto o servicio) que tienen una buena comprensión de cómo optimizar la experiencia del usuario también podrían tener diferentes culturas y formas de trabajo que pueden introducir desafíos en trabajar juntos para desarrollar un sistema ML listo para producción [15].

3.6 Conclusiones

Aunque la tecnología del DL ha logrado resultados muy prometedores, todavía hay una necesidad significativa de más investigación y desarrollo en cómo construir fácil y eficientemente sistemas DL listos para producción de alta calidad. SE tradicional tiene herramientas de alta calidad y prácticas para revisar, escribir pruebas y depurar código [15].

Sin embargo, rara vez son suficientes para construir sistemas listos para producción que contengan componentes DL. Si la comunidad SE, junto con la comunidad DL, pudiera hacer un esfuerzo en la búsqueda de soluciones a estos desafíos, el poder de la tecnología del DL no solo podría ponerse a disposición de los investigadores y grandes empresas tecnológicas, pero también a la gran mayoría de empresas de todo el mundo [15].

Capítulo 4: Descripción de la Solución

La solución propuesta comprende inicialmente el aplicar estrategias propias de la Visión por Computador en la adquisición y el pre-procesamiento de las imágenes, para luego emplear las estrategias de Aprendizaje Profundo a fin de obtener las tuberías de procesamiento (pipelines) adecuadas para generar los modelos tanto para la parte de detección de objetos como para la clasificación de imágenes.

4.1 Adquisición de imágenes

Para comenzar a resolver cualquier tarea de visión por computadora utilizando el aprendizaje profundo, primero se necesita tener un conjunto de datos de imágenes para entrenar. En este trabajo se utilizó dos enfoques diferentes para recopilar las imágenes de las categorías relevantes.

4.1.1 Enfoque 1: Buscar o recopilar un conjunto de datos

La forma más rápida de adquisición de imágenes es tener un conjunto de datos existente en mano. Hay toneladas de conjuntos de datos disponibles públicamente para los cuales una categoría o una subcategoría podría ser relevante para nuestra tarea. Por ejemplo, en este trabajo utilizamos un dataset de PlantVillage (<https://github.com/spMohanty/plantVillage-Dataset>) que contiene 54.303 imágenes de enfermedades del tomate.

Manzana	Ají (Pimienta)
Arándano	Papa
Cereza	Frambuesa
Maíz	Soya (Soja)
Uva	Zapallo (Calabaza)
Naranja	Frutilla (Fresa)
Durazno	Tomate

Tabla 4.1: Especies de cultivo que contempla el dataset PlantVillage.

Mancha Bacteriana	Arañuela Roja de dos Manchas
Tizón Temprano	Target Spot (Punto Objetivo)
Tizón Tardío	Yellow Leaf Curl Virus
Molde de Hoja	Virus del Mosaico
Mancha Foliar Septoria	Tomate Sano

Tabla 4.2: Enfermedades y Plagas que contempla el dataset PlantVillage.

El tipo de cultivo elegido del dataset PlantVillage es el que corresponde al tomate (Tabla 4.1) y las enfermedades consideradas en el presente trabajo, son aquellas que se dan en la hoja de tomate: Mancha Bacteriana, Tizón Temprano y Hoja Sana (Tabla 4.2).

Para descargar las imágenes (se utilizó un entorno Linux) con los siguientes comandos:

```
git clone https://github.com/spMohanty/PlantVillage-Dataset
cd PlantVillage-Dataset
```

En caso de que no se pueda encontrar un conjunto de datos, también es posible construir un conjunto de datos tomando fotos con un teléfono inteligente. Es esencial tomar imágenes representativas de cómo se usaría nuestra aplicación en el mundo real. Alternativamente, se puede recurrir a los amigos, familiares y compañeros de trabajo para generar un conjunto de datos diverso. Otro enfoque utilizado por las grandes empresas contratan a personas encargadas de recolectar imágenes. Por ejemplo, Google Allo lanzó una función para convertir selfies en pegatinas. Para construirlo, contrataron a un equipo de artistas para tomar una imagen y crear la etiqueta correspondiente para que puedan entrenar a un modelo en ella.

4.1.2 Enfoque 2: Extensión del navegador Fatkun Batch descargar imagen

Hay varias extensiones de navegador que nos permiten descargar por lotes múltiples imágenes de un sitio web. Un ejemplo de ello es *Fatkun Batch Download Image*, una extensión del navegador disponible en el navegador Chrome. Se puede tener todo el conjunto de datos listo con los siguientes pasos breves y rápidos.

1. Agregar la extensión a nuestro navegador.
2. Buscar la palabra clave en Google o Bing Image search.
3. Seleccionar el filtro apropiado para las licencias de imagen en el ajuste de la búsqueda.
4. Después de que la página se vuelva a cargar, desplácese hasta la parte inferior varias veces repetidamente para garantizar que se carguen más miniaturas en la página.
5. Abra la extensión y seleccione la opción "Esta pestaña", como se muestra en la figura 4.1.

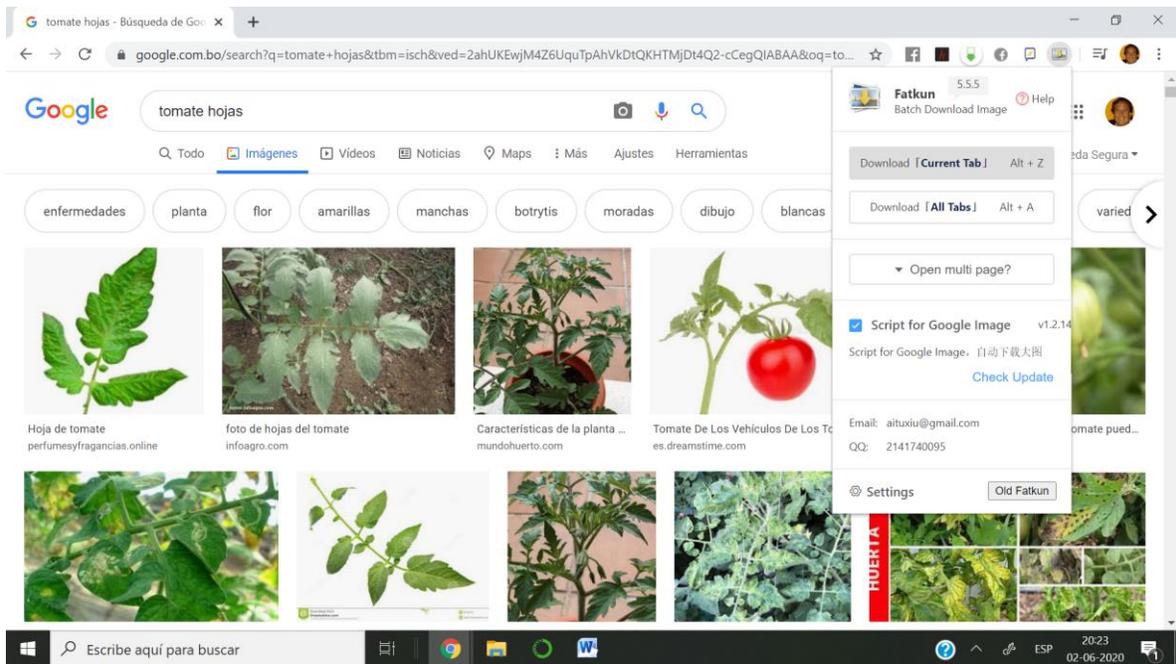


Figura 4.1: Resultado de la búsqueda en Chrome para “tomate hojas”.

Observar que todas las miniaturas están seleccionadas por defecto. En la cima de la pantalla, haga clic en el botón Alternar para anular la selección de todas las miniaturas y seleccione solo los que se necesitan. Se Puede establecer el ancho mínimo y la altura debe ser 224 (la mayoría de los modelos entrenados toman 224x224 como tamaño de entrada).

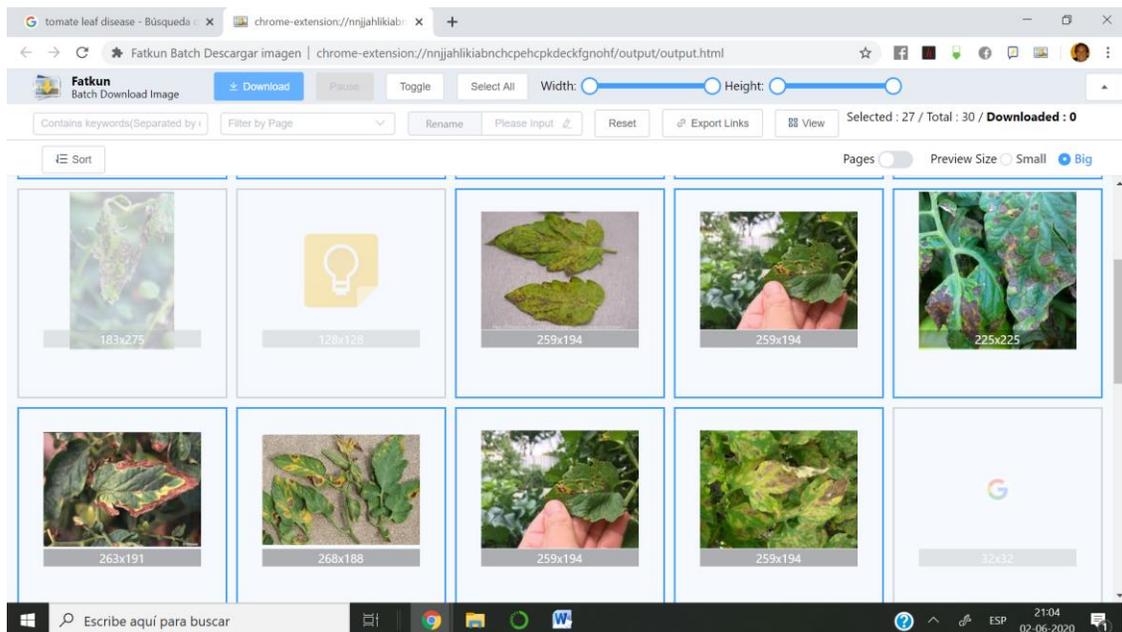


Figura 4.2: Seleccionando imágenes a través de la extensión *Fatkun*.

En la esquina superior derecha, hacer clic en *Download*, para descargar todas las imágenes seleccionadas a nuestra computadora.

Tener en cuenta que las imágenes que se muestran en las capturas de pantalla son imágenes icónicas (es decir, el principal objeto está en foco directo con un fondo limpio). Lo más probable es que usando las imágenes exclusivamente en nuestro modelo harán que no se generalice bien al mundo real, por ejemplo, en imágenes con un fondo blanco limpio (como en un sitio web de comercio electrónico), la red neuronal podría aprender incorrectamente que el fondo blanco es igual a hoja de tomate. Por lo tanto, al realizar la recopilación de datos, hay que asegurarse de que las imágenes de entrenamiento son representativas del mundo real [16].

4.2 Pre procesamiento

Tanto para la detección de objetos como la clasificación hay tareas comunes para ambas actividades, como la depuración del dataset a partir de las imágenes descargadas que no tengan una buena calidad y que no puedan ser mejoradas.

En cuanto a la detección de objetos otra actividad a realizar es la de ubicar el objeto (hoja) en una imagen. En este caso se tiene que detectar la hoja en la imagen, para posteriormente recortar el área de la hoja y someterla a clasificación para determinar si presenta alguna enfermedad o no.

Con el surgimiento de las redes neuronales convolucionales la detección de enfermedades puede ser más fácil con técnicas como el deep learning en comparación con los métodos manuales, a través del procesamiento de imágenes que permitan detectar en tallos, ramas, hojas y flores características propias de las diferentes enfermedades que pueden afectar a una planta de una manera más precisa. Estos sistemas incluyen no solo reconocimiento y clasificación de objetos en una imagen, sino que también pueden ubicar cada uno de ellos dibujando recuadros apropiados a su alrededor. Esto hace que detectar un objeto sea una tarea más difícil que su tradicional predecesora, la visión por computadora en la clasificación de imágenes [17].

Para comprender cómo se ve el resultado de la detección de objetos, la figura 4.3 muestra un ejemplo del problema a solucionar:

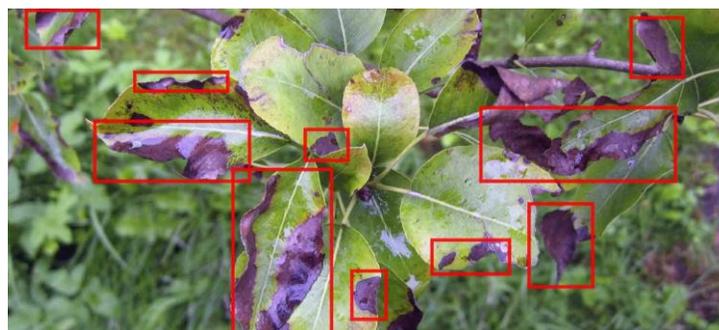


Figura 4.3: Detección de hojas con enfermedad.

4.2.1 Creación del conjunto de datos de entrenamiento para detección de objetos

La detección de objetos nos da como salida un cuadro delimitador que rodea el objeto de interés en una imagen. Para construir un algoritmo que detecte el límite cuadro que rodea el objeto en una imagen, se tendría que crear un mapeo de la entrada-salida, donde la entrada es la imagen y la salida son los cuadros delimitadores que rodean los objetos en la imagen dada.

Se tiene que tener en cuenta que cuando se detecta el cuadro delimitador, se identifica las ubicaciones de los píxeles de la esquina superior izquierda, como también el ancho correspondiente y altura del cuadro delimitador.

Para entrenar un modelo que proporciona el cuadro delimitador, se necesita la imagen y también las coordenadas correspondientes del cuadro delimitador de todos los objetos en una imagen.

Una de las formas de crear el conjunto de datos de entrenamiento con la imagen como entrada y los cuadros delimitadores correspondientes es mediante un archivo XML.

Para crear el conjunto de datos de entrenamiento que incluye imágenes con sus correspondientes cuadros delimitadores y las clases correspondientes alrededor de los objetos en la imagen, se utilizó el paquete *labelImg*². A continuación se describe el procedimiento de uso de dicho paquete.

Extraer y abrir la GUI *labelImg.exe*, que se muestra en la siguiente captura de pantalla:

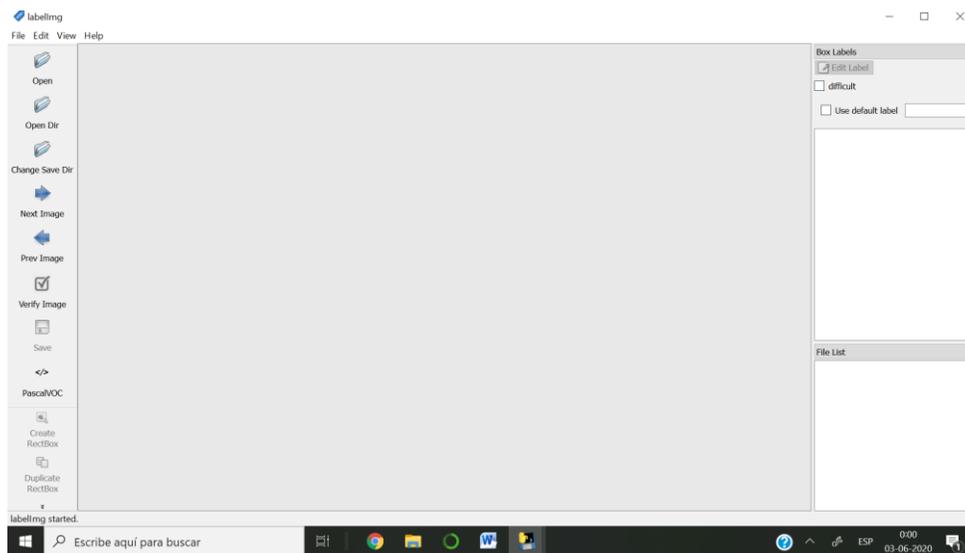


Figura 4.4: Interfaz de *LabelImg*.

² <https://github.com/tzutalin/labelImg/releases>

Cargar una imagen haciendo clic en Abrir en la GUI y anote la imagen haciendo clic en *Create RectBox*, que mostrará las clases que se seleccionarán de la siguiente manera:

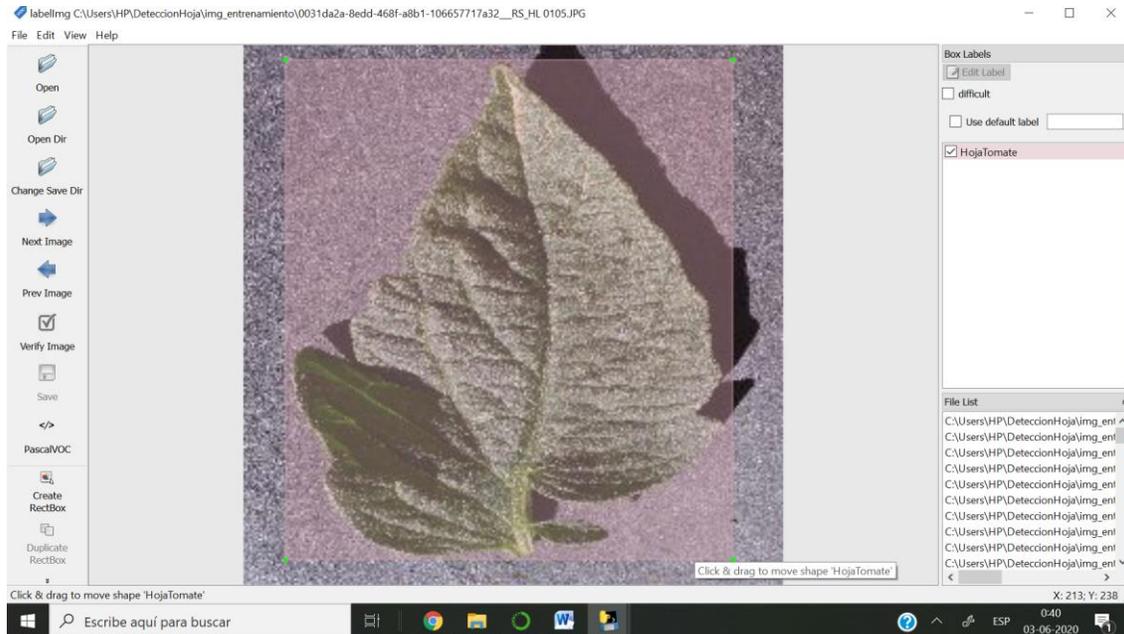


Figura 4.5: Etiquetando una imagen y colocando cuadro delimitador con *LabelImg*.

Hacer clic en Guardar y almacenar el archivo XML.

Inspeccionar el archivo XML. Una vista del archivo XML después de dibujar el rectangular el cuadro delimitador tiene el siguiente aspecto:

```
<annotation>
  <folder>ImagesTomateTizontemprano</folder>
  <filename>00c5c908-fc25-4710-a109-db143da23112__RS_Erly.B
7778.JPG</filename>
  <path>C:\Users\HP\Desktop\ImagesTomateTizontemprano\00c5c908
-fc25-4710-a109-db143da23112__RS_Erly.B 7778.JPG</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>256</width>
    <height>256</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>HojaTomate</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>17</xmin>
      <ymin>19</ymin>
      <xmax>225</xmax>
      <ymax>238</ymax>
    </bndbox>
  </object>
</annotation>
```

Figura 4.6: Archivo XML de una imagen etiquetada y delimitada con recuadro.

De la captura de pantalla anterior, se debe tener en cuenta que el *bndbox* contiene las coordenadas de los valores mínimos y máximos de las coordenadas x e y correspondientes al objeto de interés en la imagen. Además, también se debería estar en condiciones de extraer la clase correspondiente al objeto en la imagen [17].

4.3 Detección de objetos y clasificación

Los avances recientes en redes neuronales profundas (DNN), particularmente usando redes neuronales convolucionales, ha llevado a una revolución en el procesamiento de imágenes, logrando (o incluso superando) el rendimiento a nivel humano. Actualmente se tiene modelos artificiales que pueden rivalizar con cerebros humanos en actividades perceptivas complejas. Eberhardt y col., en "¿ Qué tan profundo es el análisis de características subyacente en una categorización visual rápida?" (<http://arxiv.org/abs/1606.01167>) comparado el desempeño de las CNN con humanos y se demostró que las CNN pueden lograr un rendimiento sobrehumano en un reconocimiento visual rápido [18].

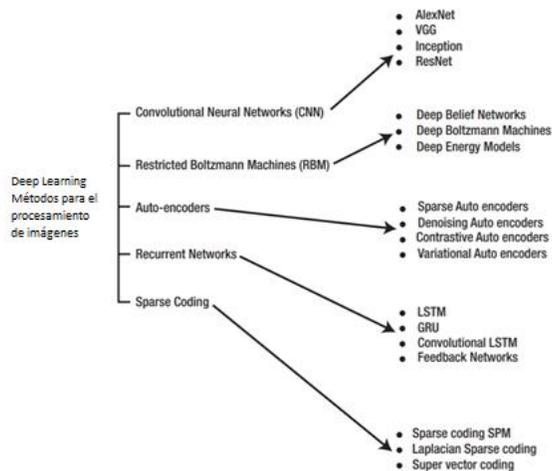


Figura 4.7: Métodos de Deep Learning para el procesamiento de imágenes.

De los Métodos para el Procesamiento de Imágenes utilizando Deep Learning en la figura 4.7, se eligió el de Redes Neuronales Convolucionales (CNN) tanto para la detección de objetos como para la clasificación de imágenes.

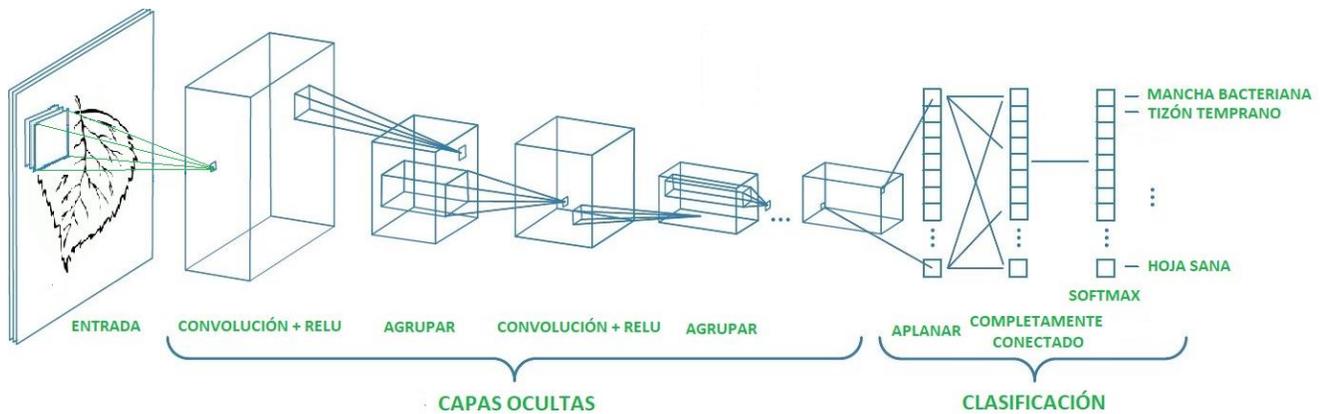


Figura 4.8: Arquitectura Red Neuronal Convolutiva (CNN).

4.3.1 Metodología para detección de objetos en la imagen

La detección de objetos tuvo una explosión tanto en aplicaciones como en investigaciones en años recientes. La detección de objetos es un problema de importancia en la visión por computadora. Similar a la tarea de clasificación de imágenes, las redes más profundas (DNN) han mostrado un mejor rendimiento en la detección. En la actualidad la precisión de estas técnicas son excelentes, por lo tanto se usa en muchas aplicaciones.

La clasificación de imagen etiqueta la imagen como un todo. Encontrar la posición del objeto además de etiquetar el objeto se llama *localización del objeto*. Por lo general, la posición del objeto se define mediante coordenadas rectangulares. Encontrar múltiples objetos en la imagen con las coordenadas rectangulares se llama detección.

Existen diferentes algoritmos para la detección de objetos, entre los que se puede mencionar:

- R-CNN
- SSD
- YOLO

En el presente trabajo se empleó R-CNN (Regions of the Convolutional Neural Network) que permite trabajar con imágenes, en tanto SSD (Single Shot Detection) y YOLO (You Only Look Once) se aplican a la captura de imágenes en videos.

4.3.1.1 Redes Neuronales Convolucionales Basada en la Región (R-CNNs)

La familia de técnicas de detección de objetos R-CNN generalmente se conoce como R-CNN, que es la abreviación para las redes neuronales convolucionales basadas en la región, desarrolladas por Ross Girshick y todos los demás en 2014 en su documento " Rich feature hierarchies for accurate object

detection and semantic segmentation". La familia R-CNN se ha expandido para incluir Fast-RCNN y Faster-RCNN que salió en 2015 y 2016. Evolución de la familia R-CNN de R-CNN a Fast R-CNN a Faster R-CNN.

R-CNN: Los cuadros delimitadores son propuestos por el algoritmo de "búsqueda selectiva", de cada uno que está deformado y las características se extraen a través de una red neuronal convolucional profunda, como AlexNet, antes de un conjunto final de clasificaciones de objetos y cuadro delimitador de predicciones se realizan con SVM lineales y regresores lineales.

Fast R-CNN: Un diseño simplificado con un solo modelo, una capa de agrupación de regiones de interés se usa después de CNN para consolidar regiones y el modelo predice ambas etiquetas de clase y regiones de interés directamente.

4.3.1.2 Faster R-CNN

Faster R-CNN es la tercera generación de la familia R-CNN. Salió en 2016 por Ross Girshick y coautores en su artículo "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". Similar a Fast R-CNN, la imagen es la entrada a una red convolucional que proporciona un mapa de características convolucional. En lugar de usar un algoritmo de búsqueda selectiva en el mapa de características para identificar la región propuesta, se utiliza una red para predecir las regiones propuestas como parte del proceso de entrenamiento, denominado **Region Proposal Network** (RPN). Las regiones propuestas pronosticadas son reformadas utilizando una capa de agrupación de RoI (Regions of Interest) que luego se utiliza para clasificar la imagen dentro de la región propuesta y predecir los valores de desplazamiento para el recuadro. Estas mejoras reducen el número de regiones propuestas y aceleran la operación en tiempo de prueba del modelo casi en tiempo real con el rendimiento más avanzado [19].

La arquitectura de Faster R-CNN está compuesta de dos redes principales:

Region Proposal Network (RPN) - la búsqueda selectiva se reemplaza por una ConvNet para proponer regiones de interés (RoI) desde los últimos mapas de características del extractor de características a ser considerado para investigaciones. RPN tiene dos salidas; la "puntuación de objetividad" (objeto o sin objeto) y la ubicación de la caja.

Fast R-CNN - consiste de los típicos componentes de Fast R-CNN:

Red base para el extractor de características: una típica CNN pre entrenada para extraer características de la imagen de entrada.

Capa de agrupación de RoI: para extraer regiones de interés de tamaño fijo.

Capa de salida: contiene 2 capas completamente conectadas: 1) un clasificador softmax para proporcionar la probabilidad de la clase y 2) una regresión del cuadro delimitador CNN para la predicción del cuadro delimitador (recuadro).

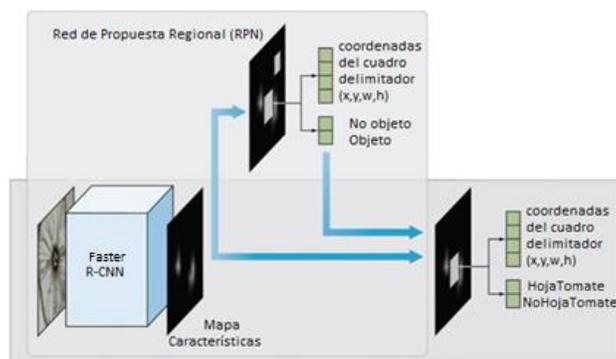


Figura 4.9: Arquitectura Faster R-CNN tiene dos componentes: 1) una red de región propuesta (RPN), que identifica regiones que pueden contener objetos de interés y su ubicación aproximada; y 2) una red Faster R-CNN, que clasifica objetos y refina su ubicación, definida mediante cuadros delimitadores.

Como se puede ver en el diagrama de arquitectura Faster R-CNN (Figura 4.9), la imagen de entrada es presentada a la red y sus características se extraen a través de una CNN previamente capacitada. Estas características en paralelo se envían a dos componentes diferentes de la arquitectura Faster R-CNN:

El RPN determina en qué parte de una imagen podría estar un objeto potencial. En este punto no se conoce qué es el objeto, solo que potencialmente hay un objeto en cierta ubicación en la imagen.

La agrupación de ROI (Regions of Interest) para extraer recuadros de características de tamaño fijo. La salida se pasa a dos capas completamente conectadas: una para el clasificador de objetos y otra para las predicciones de las coordenadas del cuadro delimitador para obtener las localizaciones finales.

Esta arquitectura logra un entrenamiento de extremo a extremo y la detección completa de la tubería objetos donde todos los componentes requeridos tienen lugar dentro de la red, incluidos [19]:

- Red base extractor de características
- Regiones de propuesta
- Agrupación de ROI
- Clasificación de objetos
- Regresor del cuadro delimitador

4.3.1.2.1 Red Base Para Extraer Características

Similar a Fast R-CNN, el primer paso es usar una CNN pre entrenada y cortar parte de su clasificación parte. La red base se utiliza para extraer características de la imagen de entrada. En este componente, se puede usar cualquiera de las arquitecturas populares de CNN basadas en el problema que se está tratando de resolver. El paper original de Faster R-CNN utilizaba redes pre entrenadas ZF y VGG en ImageNet, pero desde entonces ha habido muchas redes diferentes con un número variable de pesos.

Por ejemplo, MobileNet, es una arquitectura de red más pequeña y eficiente optimizada para la velocidad, tiene aproximadamente 3,3 millones de parámetros, mientras que ResNet-152 (152 capas), el estado del arte en la competencia de clasificación tiene a ImageNet alrededor de 60 millones. Más recientemente, nuevas arquitecturas como DenseNet están mejorando los resultados al tiempo que reducen el número de parámetros [19].

4.3.1.2.2 Region Proposal Network (RPN)

La red de propuesta de región (RPN) identifica regiones que podrían contener objetos de interés, basada en el último mapa de características de la red neuronal convolucional pre entrenada. RPN también es conocida como la "red de atención" porque guía la atención de la red a regiones interesantes en la imagen. Faster R-CNN utiliza la red de propuesta de región (RPN) para detectar la región propuesta directamente en la arquitectura R-CNN en lugar de ejecutar un algoritmo de búsqueda selectiva para extraer regiones de interés [19].

La arquitectura de RPN está compuesta de dos capas:

- Una capa totalmente convolucional 3x3 con 512 canales.
- Dos capas convolucionales paralelas 1x1: una es la capa de clasificación que se utiliza para predecir la clase, si la región contiene un objeto o no (la puntuación de fondo o primer plano), si la respuesta es sí, entonces la región se transmite para una mayor investigación mediante la agrupación de RoI (Región de Interés) y las capas finales de salida como se ve en la figura 4.11. La otra capa es para regresión o predicción del cuadro delimitador.



Figura 4.10: Implementación convolucional de una arquitectura RPN, donde k es el número de anclas (anchors).

La capa CONV 3x3 se aplica en el último mapa de características de la red base donde una ventana de deslizamiento de tamaño 3x3 se pasa sobre el mapa de características. La salida se pasa a dos capas CONV de 1x1, un clasificador y un regresor de cuadro delimitador. Tener en cuenta que el clasificador y el regresor del RPN no están tratando de predecir la clase del objeto y su cuadro delimitador. Esto vendrá más tarde después del RPN. El objetivo del RPN es determinar si la región tiene un objeto o no para ser investigado por las capas completamente conectadas posteriormente. En RPN se utiliza un clasificador binario para predecir la puntuación de objetividad de la región para determinar la probabilidad de que esta región sea un primer plano (que contiene un objeto) o un fondo (no contiene un objeto). Básicamente, mira la región y pregunta: "¿Esta región contiene un objeto?". Si la respuesta es sí, entonces la región es pasada para una mayor investigación por parte de RoI agrupación y las capas de salida finales. Ver figura 4.11.



Figura 4.11: El clasificador RPN predice la puntuación de objetividad, que es la probabilidad de que una imagen contenga un objeto (primer plano) o un fondo.

4.3.2 Metodología para la clasificación de imágenes

La clasificación de imágenes básicamente es la tarea de asignar una etiqueta a una imagen de un conjunto de categorías con una probabilidad asociada. Las aplicaciones incluyen la clasificación de género dada una imagen del rostro de una persona, identificar el tipo de mascota, etiquetar fotos, clasificar enfermedades, etc. Prácticamente, esto significa que la tarea es analizar una imagen de entrada y devolver una etiqueta que categoriza la imagen. La etiqueta es siempre de un conjunto predefinido de categorías posibles. El siguiente es un resultado de tal tarea de clasificación [20] [21]:

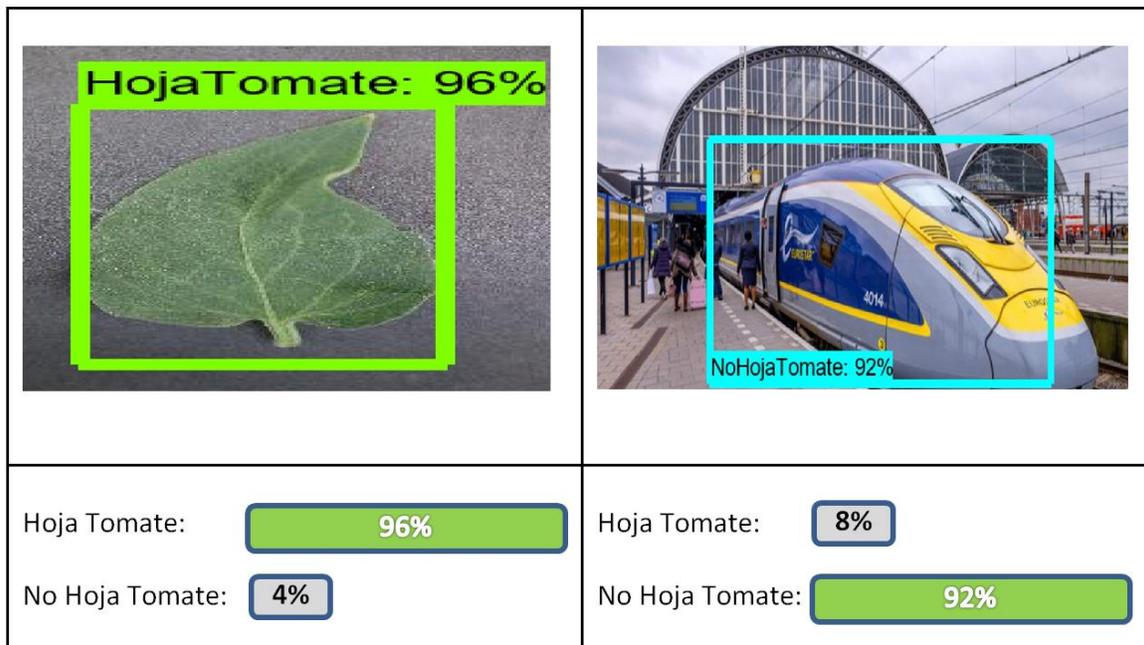


Figura 4.12: Clasificador de imágenes, Hoja Tomate y No Hoja Tomate con su probabilidad.

Un algoritmo DL en su núcleo sigue siendo un algoritmo de aprendizaje supervisado: aprende un mapeo entre una entrada y una salida de los datos de entrenamiento. La principal diferencia está en el tipo de datos de entrada con los que trabaja: mientras que los algoritmos ML tradicionales necesitan características refinadas, la extracción de características es manual (mitad superior de la figura 4.13), los algoritmos DL trabajan directamente con números sin procesar y pueden comprender características relevantes automáticamente, la extracción de características es automática (mitad inferior de la figura 4.13) [22].

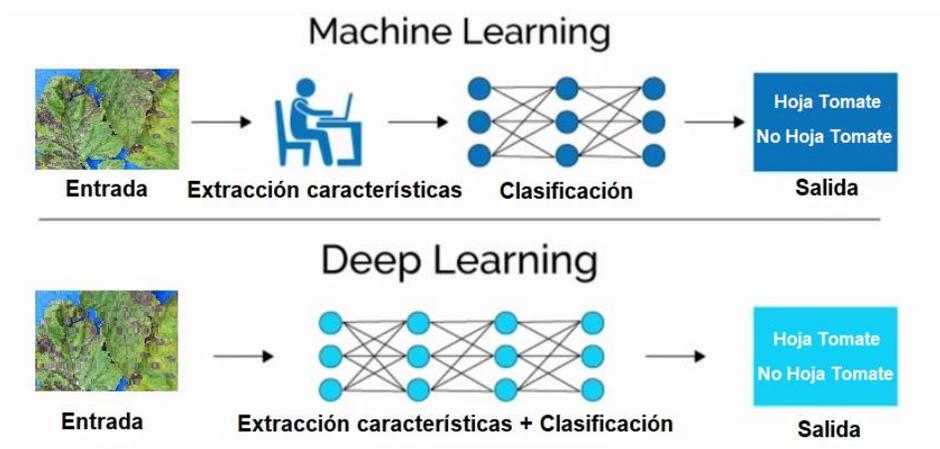


Figura 4.13: En el aprendizaje automático tradicional, los ingenieros deben desarrollar algoritmos para extraer características que se pueden alimentar al modelo. Un modelo DL no necesita este complejo paso preliminar.

El aprendizaje profundo se basa en una clase especial de algoritmos llamados redes neuronales profundas, compuestas por neuronas artificiales que no se parecen en nada a sus contrapartes biológicas. Ellos son extremadamente simples: solo pueden realizar multiplicaciones y sumas. Como se muestra en la figura 4.14, cada neurona recibe los números de entrada, realiza operaciones matemáticas simples con ellos y produce un número de salida [22].

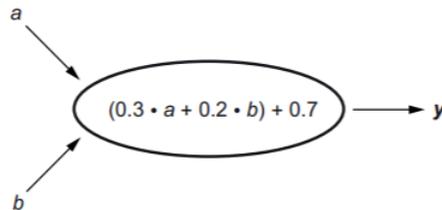


Figura 4.14: Una sola neurona artificial recibe dos números (a y b), realiza operaciones matemáticas simples como sumar, multiplicar y emite otro número y.

Una neurona por sí sola es simple y francamente inútil, sin embargo como conecta con muchas neuronas juntas para formar una red, se vuelven capaces de realizar cálculos sofisticados. Se muestra una red simple en la figura 4.15: la primera columna de neuronas recibe la entrada inicial y pasa la información procesada a la columna central de neuronas, y así. Finalmente, la neurona solitaria de la derecha la columna produce la salida final de la red.

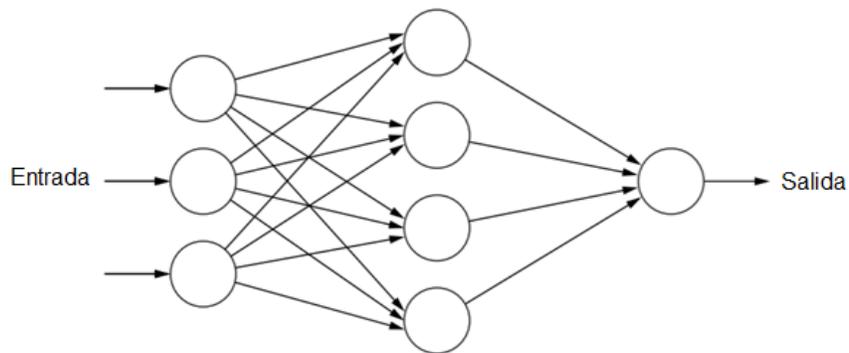


Figura 4.15: Las cosas comienzan a ponerse interesantes cuando se conectan varias neuronas juntas a medida que se vuelven capaces de expresar operaciones matemáticas más complejas.

En la clasificación de imágenes lo que se quiere hacer, es crear una red que recibe los píxeles de la imagen, realizar algunos cálculos internos y genera la clase (con suerte) correcta para la imagen de entrada. La figura 4.16 muestra cómo encaja todo esto.

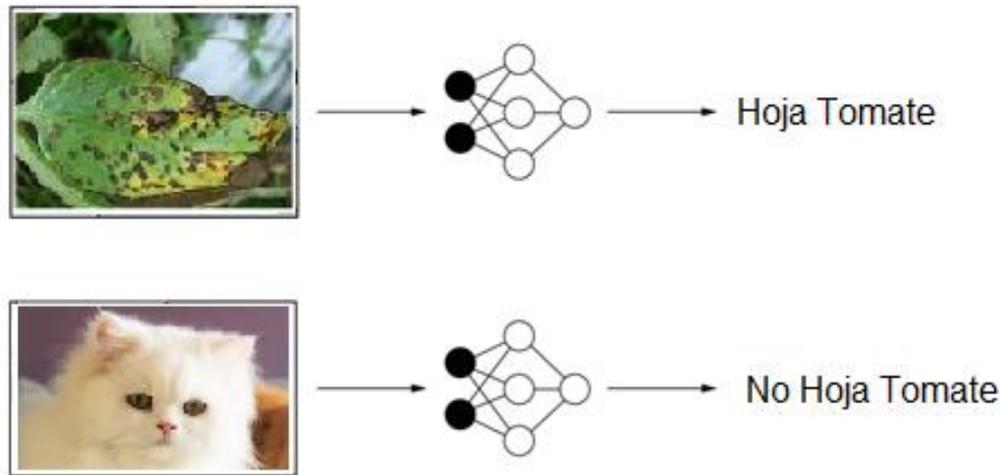


Figura 4.16: Las redes neuronales profundas procesan las imágenes de entrada (a la izquierda) y producen una clasificación que sea consistente con las etiquetas de entrenamiento (a la derecha).

En las redes neuronales profundas, las neuronas se organizan en columnas (o capas) y esta estructura es lo que las hace capaces de extraer características de las imágenes sin especificarlas manualmente. Sin embargo, ¿por qué y cómo?

Se puede imaginar cada capa como un paso en una línea de producción: las filas de números que componen la imagen ingresan a la primera capa, donde las neuronas las procesan aplicando operaciones matemáticas (multiplicación y suma). El resultado es que estos números sin procesar son modificados y transformados en otros números que luego se pasan a la segunda capa. La segunda capa aplica otras transformaciones y envía sus resultados a la tercera capa. Este proceso se repite para cada capa hasta la capa final, donde dirá, "La imagen contiene una hoja de tomate" o "La imagen contiene una no hoja de tomate".

La analogía con una línea de producción tiene sentido porque no existe una máquina mágica que puede convertir materias primas como el acero y el cuero en un automóvil nuevo y brillante en un solo paso. En una línea de producción se conoce que el proceso para construir un automóvil se divide en pequeños pasos: las materias primas se moldean primero con máquinas en pequeños componentes básicos que incluyen perillas, engranajes, etc. Estos componentes luego son procesados por otras máquinas que forman el cambio, el volante, la radio, etc. adelante. Sólo después de que se construyen estas piezas complejas se ensamblan finalmente para formar un automóvil; Sería imposible tener una sola máquina gigantesca que convierta materias primas en coches en un solo paso. Cada capa de una red neuronal profunda hace prácticamente la misma cosa, excepto que funciona con números y no con acero.

Estas transformaciones son la clave del poder mágico de la función de extracción de características de manera automática. Cuando una red neuronal comienza a entrenarse con una gran cantidad de imágenes, su precisión inicial es horrible, ya que las neuronas en las diversas capas aún no han descubierto cómo procesar estos números de manera significativa. A medida que la red ve más imágenes y su entrenamiento continúa, una técnica matemática especial llamada *backpropagation* (propagación hacia atrás) anima a cada neurona a ajustar las transformaciones que aplica para que la

red en general mejore su precisión de clasificación. Cuantas más imágenes tenga la red, sus neuronas aprenderán más transformaciones significativas.

Pero, ¿cuáles son estas transformaciones? Suena un tanto vago y es que en los primeros días de aprendizaje profundo, también era un poco misterioso para los investigadores que construían esta tecnología. Después de que las redes neuronales profundas lograran un rendimiento asombroso, los investigadores comenzaron a profundizar en su funcionamiento interno. Lo que encontraron fue extremadamente interesante: cuando las neuronas de las primeras capas optimizan su transformación, lo hacen de una manera que les permite reconocer líneas simples, bordes y esquinas de los píxeles sin procesar, de manera similar a los filtros hechos a mano que estaban probando los primeros ingenieros de visión por computadora. La siguiente capa recibirá esta información que es mucho más rica que la inicial. Números crudos: estos nuevos números representan la presencia de formas en diferentes partes de la imagen en lugar del color de un solo píxel. Esta segunda capa contendrá esta información y aplicará otra transformación que le permita reconocer formas más complejas como círculos y cuadrados, a partir de líneas, bordes y esquinas más simples. Posteriormente una capa eventualmente habrá reconocido las formas que nos importaban. En nuestro ejemplo (figura 4.16) hoja de tomate versus no hoja de tomate. La figura 4.17 muestra cómo se ven estas formas cada vez más complejas. Finalmente la última capa tendrá que tomar la decisión final sobre a qué clase pertenece la imagen, basado en los elementos extraídos por la capa anterior.

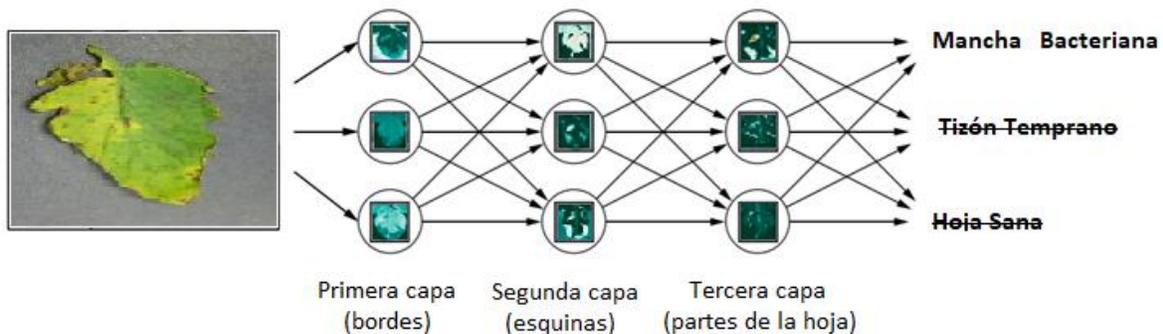


Figura 4.17: Representaciones internas desarrolladas por una red neuronal profunda entrenada para reconocer hojas.

La magia del aprendizaje profundo es que todas estas neuronas y capas aprendieron de forma autónoma a descomponer los millones de números que hacen una imagen de formas básicas a complejas y luego hacer una predicción basada en ellos. Básicamente la estructura modular de una red neuronal le permite procesar información en pasos y gracias a formulaciones matemáticas inteligentes, la red aprende a ajustar estos pasos para reconocer los rasgos más característicos de las imágenes de las que está aprendiendo: en este caso, hoja de tomate y no hoja de tomate.

Gracias al aprendizaje profundo son identificados automáticamente: nadie le ha dicho a la primera capa cómo reconocer líneas y esquinas, nadie le ha dicho a las siguientes cómo combinarlas para formar círculos y rectángulos, y nadie ha venido con una forma inteligente de enseñar a las últimas capas cómo reconocer elementos complejos como una alcachofa de la ducha. Todo lo que se hace es tomar una red

neuronal con varias capas, alimentarla con imágenes de hojas de tomate y no hojas de tomate (gatos, perros, tigres, etc.), y dejar por detrás que las matemáticas hagan el resto con *backpropagation*. El aprendizaje profundo requiere de una gran cantidad de potencia de cálculo y también de una considerable cantidad de datos [22].

4.4 Recursos Necesarios

A continuación se detallan los recursos tecnológicos tanto de hardware como de software empleados en el presente trabajo.

4.4.1 Hardware

El hardware utilizado es un computador portátil HP Pavilion x360 - 14-ba008la con las siguientes características:

- Procesador Intel® Core™ i7-7500U (2,7 GHz de frecuencia base, hasta 3,5 GHz con tecnología Intel® Turbo Boost, 4 MB de caché, 2 núcleos)
- Memoria 16 GB de SDRAM DDR4-2133 (2 x 8 GB)
- Disco duro SATA de 1 TB y 5400 rpm y SSD M.2 de 128 GB
- Pantalla multi táctil de vidrio borde a borde WLED FHD IPS de 35,6 cm (14") en diagonal (1920 x 1080)
- Tarjeta gráfica NVIDIA® GeForce® 940MX (4 GB de DDR3 dedicados)

4.4.2 Software

Se han empleado diversos tipos de software (basado en sus principales características) para las distintas tareas como: Visión por Computadora (descargar imágenes de manera automática de la web y pre procesamiento de imágenes); Aprendizaje Profundo (DL); Construcción de la Interfaz de Usuario. Que se requirieron el presente trabajo, a continuación se detalla en la siguiente lista:

- Sistema operativo: *Windows 10 home*
- Descargar imágenes y procesar tamaño: *I'm a Gentleman* es plugin que se instala en el navegador Chrome y se utiliza para descargar imágenes de manera automática; *Fatkun Batch* permite definir el tamaño y descargar imágenes de manera automática y se instala como una extensión del navegador Chrome.
- Pre procesamiento imágenes: *LabelImg* es una herramienta de anotación de imágenes gráficas; *PIL* (Python Imaging Library versión 7.1.2) es una librería gratuita que permite la edición de

imágenes directamente desde Python. Soporta una variedad de formatos, incluidos los más utilizados como GIF, JPEG y PNG. Una gran parte del código está escrito en C, por cuestiones de rendimiento.

- Framework: **AnacondaNavigator-Jupyter Notebook** (Anaconda Navigator versión 1.9.7; Jupyter Notebook versión 5.7.8) es un entorno pensado para satisfacer necesidades concretas y ajustarse al **flujo de trabajo de la ciencia de datos y la simulación numérica**. En una sola interfaz, los usuarios pueden escribir, documentar y ejecutar código, visualizar datos, realizar cálculos y ver los resultados.
- Framework en **TensorFlow y Keras para Detección y Segmentación de Objetos con Mask R-CNN**, de Matterport: https://github.com/matterport/Mask_RCNN (TensorFlow 1.3+). Se utiliza una versión actualizada a TensorFlow 2.x de Adam Kelly en el siguiente repositorio: https://github.com/akTwelve/Mask_RCNN.
- Lenguaje de programación: **Python** (versión 3.7.10)
- Librerías: **scikit-learn** (versión 0.22.2.post1) es un paquete de Python de código abierto para el aprendizaje automático.
- Librerías: **scikit-image** (versión 0.16.2) es un paquete de Python que está compuesta por una colección de algoritmos para el procesamiento de imágenes.
- Librerías: **opencv** (versión 4.1.2) es un paquete open-source de visión artificial y machine learning.
- Librerías: **NumPy** (versión 1.19.5) es un paquete de Python que significa “Numerical Python”, es la librería principal para la informática científica, proporciona potentes estructuras de datos, implementando matrices y matrices multidimensionales. Estas estructuras de datos garantizan cálculos eficientes con matrices.
- Librerías: **SciPy** (versión 1.4.1) es un paquete científico de Python (es decir, un módulo que tiene otros módulos) más completo, que incluye interfaces a librerías científicas muy conocidas como LAPACK, BLAS u ODR entre muchas otras.
- Librerías: **Cython** (versión 0.29.22) es un lenguaje, que es un super conjunto de Python que se compila en C, lo que produce aumentos de rendimiento que pueden ir desde un pequeño porcentaje hasta varios órdenes de magnitud, dependiendo de la tarea en cuestión.
- Librerías: **h5py** (versión 2.10.0) es una interfaz Pythonic para el formato de datos binarios HDF5. Permite almacenar grandes cantidades de datos numéricos y manipular fácilmente esos datos desde NumPy.
- Librerías: **Matplotlib** (versión 3.2.2) es una librería para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

- Deep Learning (Aprendizaje Profundo): **TensorFlow** (versión 2.4.1) es una biblioteca de código abierto para aprendizaje automático a través de un rango de tareas, y desarrollado por Google para satisfacer sus necesidades de sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos; **Keras** (versión 2.4.3) es una biblioteca de Redes Neuronales de Código Abierto escrita en [Python](#). Es capaz de ejecutarse sobre [TensorFlow](#).
- Construcción de la Interfaz de Usuario (aplicación web): **Flask** (versión 1.1.1) es un *microframework* de python para crear aplicaciones web; **Bootstrap** (versión 4.5.2) es un framework front-end utilizado para desarrollar aplicaciones web y sitios mobile first, con un layout que se adapta a la pantalla del dispositivo utilizado por el usuario.
- Google Colab es un servicio cloud, basado en los Notebooks de Jupyter, que permite el uso gratuito de las GPUs y TPUs de Google, con librerías como TensorFlow, Keras, OpenCV, etc.

Capítulo 5: Implementación

En este trabajo de investigación, se utilizarán algunos paquetes de Python para procesar una imagen. Inicialmente, se deberá usar diversas bibliotecas para realizar el procesamiento clásico de imágenes: desde la extracción de datos de imágenes, transformar los datos con algunos algoritmos utilizando funciones de biblioteca para pre-procesar, realzar, restaurar, representar (con descriptores), segmentar, clasificar y detectar y reconocer (objetos) para analizar, comprender e interpretar mejor los datos. Posteriormente, se utilizará otro grupo de bibliotecas para realizar el procesamiento de imágenes basado en el aprendizaje profundo, una tecnología que se ha vuelto muy popular en los últimos años [10].

Una vez hecha la descripción de la solución en el capítulo anterior, el presente trata de la implementación de los algoritmos para conseguir una eficiente detección del objeto (hoja de tomate) y clasificación de las enfermedades y estado sano a partir del procesamiento de imágenes de la hoja de tomate. Este capítulo contempla las siguientes secciones: Arquitectura, Pipeline (Tubería de Procesamiento), Detección de Objetos (hoja de tomate) y Clasificación de las Enfermedades.

5.1 Arquitectura

La arquitectura como se muestra en la Figura 5.1, define una imagen de entrada capturada por un dispositivo de cámara (Smartphone, Tablet o cámara digital) con diferentes resoluciones y escalas que alimenta al sistema, que después de procesar por la primera red profunda (extractor de características y clasificador) da como resultado la clase y localización del objeto (hoja de tomate) en la imagen. Luego se genera una nueva imagen con el área del objeto localizado que pasa a ser procesada por una segunda red neuronal (extractor de características y clasificación) proporcionando como resultado: la predicción de la enfermedad, síntomas y el tratamiento a seguir. Evitando así de manera temprana la expansión de la enfermedad a todo el cultivo y reducir el uso excesivo de soluciones químicas para su tratamiento.

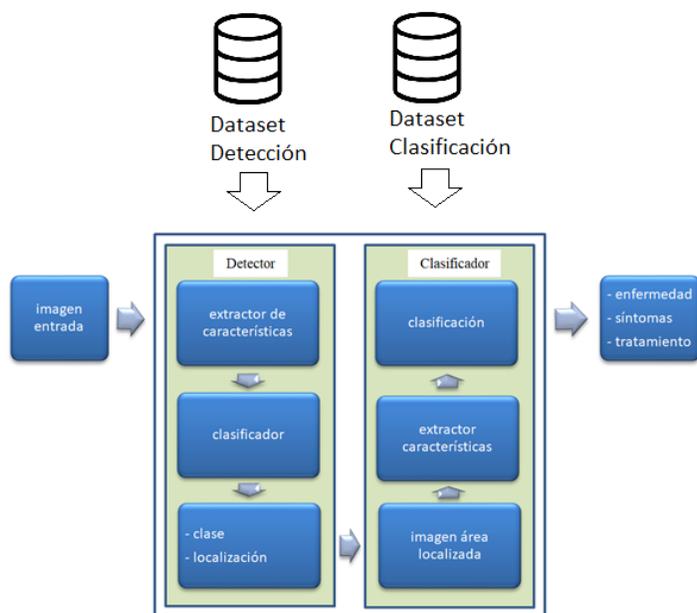


Figura 5.1: Arquitectura, parte interior izquierda es detección y la derecha es clasificación.

5.2 Pipeline (tubería de procesamiento)

El *pipeline* (tubería de procesamiento) tiene como propósito orientar el desarrollo de la arquitectura de software del proyecto de IA, reconociendo en qué aspectos el sistema de IA se comporta de manera diferente a otros sistemas de software. Al implementar eficazmente un proyecto de IA, es importante comprender los artefactos técnicos y la ciclo de vida de un proyecto de IA.

La tubería de procesamiento de ML describe cómo fluyen los datos a través del sistema, en qué alto nivel en él se realiza la transformación, qué algoritmos de ML e IA se aplican y cómo se obtienen los resultados presentados al usuario del sistema de IA [23].

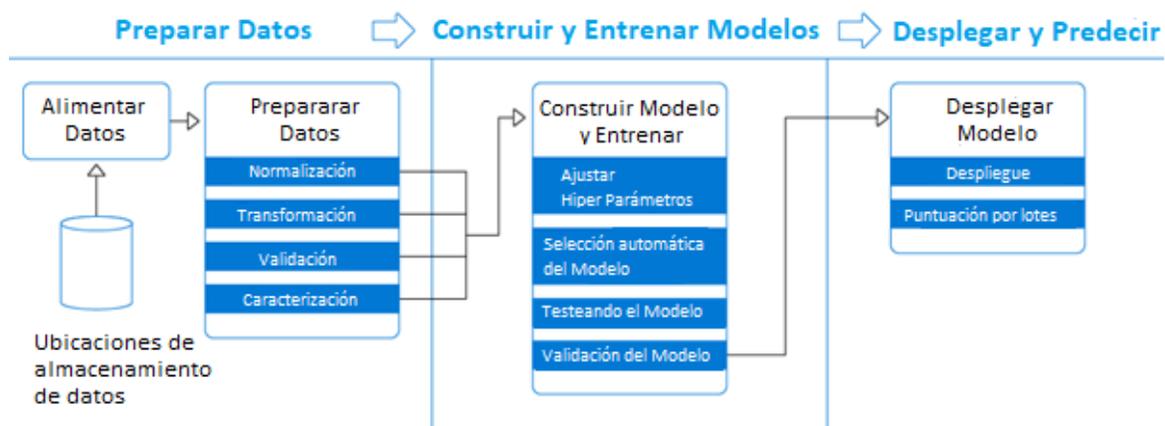


Figura 5.2: Tubería de procesamiento (pipeline) de Machine Learning.

5.2.1 La tubería de procesamiento de ML en proyectos de IA

El pipeline de ML en el sistema de software de IA no consta solo de algoritmos de IA, también se debe tener en cuenta ciertos aspectos [16][23]:

- Los algoritmos de IA operan sobre los datos. Esos datos deben almacenarse en algún lugar, lo que puede requerir una infraestructura técnica como un marco de big data.
- Los datos pueden estar sucios (lo que significa que podría haber varios errores e irregularidades), por lo que deben ser limpiados.
- Los datos relevantes para el sistema de IA a menudo residen en múltiples y diferentes fuentes de datos como un lago de datos o varias bases de datos, por lo que los datos de diversas fuentes deben ser reunidos y combinados.
- Cargar una imagen.
- Cambiar su tamaño a un tamaño predefinido, mínimo 224x224 y máximo 512x512 píxeles.

- Escalar los valores del píxel al rango [0,1], también conocido como normalización.
- Seleccionar un modelo previamente entrenado.
- Ejecutar el modelo previamente entrenado en la imagen para obtener una lista de predicciones de categorías y sus respectivas probabilidades.
- Mostrar algunas de las categorías de probabilidad más alta.

5.3 Algoritmo detección de objetos (hoja de tomate)

Para realizar la detección de la hoja de tomate (objeto) en la imagen, se empleó la arquitectura Faster R-CNN con un modelo pre entrenado, utilizando el framework Mask R-CNN de Matterport, que se corrió en Colab.

5.3.1 Preparación de la data

A partir del framework en Keras para detección de objetos (4.3.2 Software) se crean los siguientes directorios y archivos en Colab con la siguiente estructura:

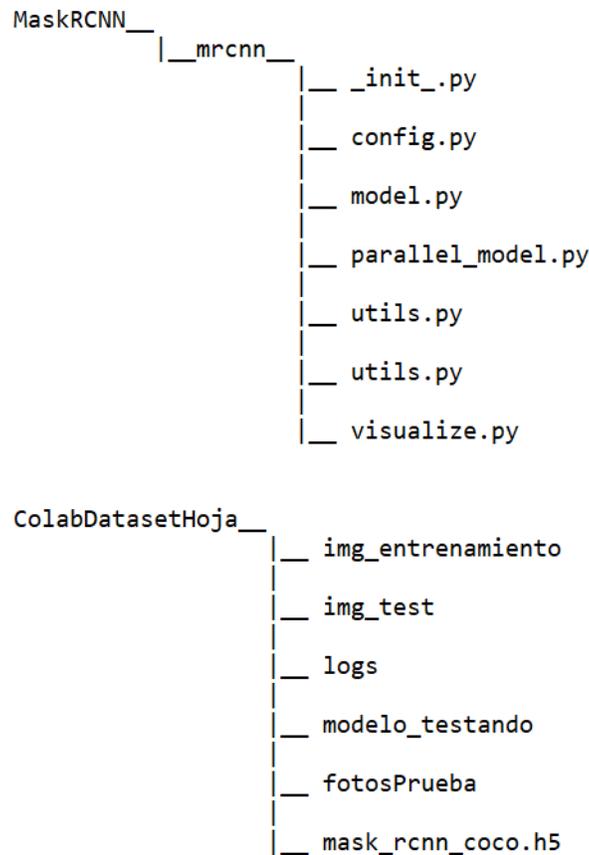


Figura 5.3: Estructura de archivos para la detección de hoja de tomate.

Siguiendo el flujo de la tubería de procesamiento (figura 5.2) la parte de Preparación de la Data (Prepare Data) se tiene la data de las imágenes depuradas y pre procesadas con la anotación (archivos XML que contienen las coordenadas del objeto que se quiere detectar, en este caso *HojaTomate* y *NoHojaTomate*) de las imágenes en las carpetas *img_entrenamiento* e *img_test* (con 3.133 imágenes y 3.133 anotaciones XML) y en la carpeta *img_test* (con 783 imágenes y 783 anotaciones XML) , que representan aproximadamente el 80% y 20% del total del dataset con 3.916 imágenes que son de hojas de tomate y también no hojas de tomate, ver figura 5.4.

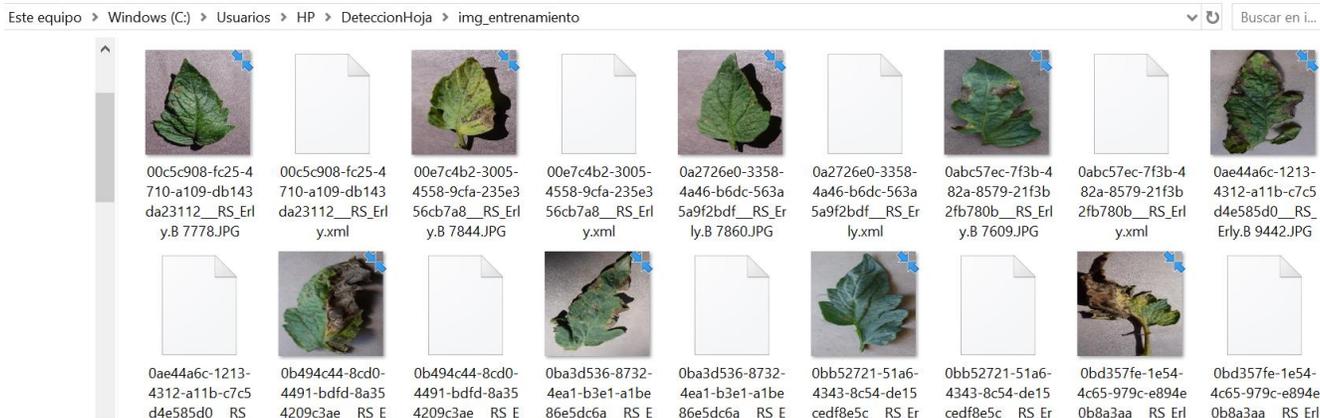


Figura 5.4: Directorio *img_entrenamiento* con los archivos de imágenes y XMLs de anotaciones.

```

00c5c908-fc25-4710-a109-db143da23112_RS_Erly.xml: Bloc de notas
Archivo Edición Formato Ver Ayuda
<annotation>
  <folder>ImagesTomateTizontemprano</folder>
  <filename>00c5c908-fc25-4710-a109-db143da23112_RS_Erly.B 7778.JPG</filename>
  <path>C:\Users\HP\Desktop\ImagesTomateTizontemprano\00c5c908-fc25-4710-a109-db143da23112_RS_Erly.B 7778.JPG</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>256</width>
    <height>256</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>HojaTomate</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>17</xmin>
      <ymin>19</ymin>
      <xmax>225</xmax>
      <ymax>238</ymax>
    </bndbox>
  </object>
</annotation>

```

Figura 5.5: Archivo XML de una imagen con las anotaciones de las coordenadas del rectángulo dibujado.

5.3.2 Construcción y entrenamiento del modelo (Build & Train Models)

Continuando con el flujo del pipeline (tubería de procesos, figura 5.2) lo que sigue ahora es la parte de Construcción y Entrenamiento del Modelo, para lo cual se siguen éstos pasos:

- Elección del modelo
- Configuración de archivos para entrenar el modelo
 - Configuración del modelo
 - Etiquetas de entrenamiento
 - Gráfica computacional del modelo
- Entrenar el modelo
- Guardar modelo entrenado para ejecutarlo posteriormente

5.3.2.1 Elección del modelo

Se utiliza el modelo *mask_rcnn_coco.h5*, que es pre entrenado (transfer learning) para realizar la detección de la hoja de tomate. Se ingresa al repositorio de matterport en github.com, para la detección de objetos en el siguiente enlace:

https://github.com/matterport/Mask_RCNN/releases/download/v2.0/mask_rcnn_coco.h5

Una vez descargado el modelo pre entrenado (*mask_rcnn_coco.h5*), se procede a guardarlo en el repositorio de Google Drive, en la carpeta *ColabDatasetHoja* con el propósito de utilizarlo en el cuaderno de Python *Detectando_hojas_Mask_R_CNN.ipynb* en Colab.

5.3.2.2 Configuración de archivos para entrenar el modelo

Una vez elegido el modelo para entrenar, se procede a ingresar a Colab (<https://colab.research.google.com/>) para crear, entrenar y guardar el modelo, como también hacer la predicción y despliegue.

```
# ...habilitar repositorio donde se encuentra el dataset ...
from google.colab import drive
drive.mount('/content/drive')

# ... cargar framework Mask RCNN para la detección de objetos ...
!git clone https://github.com/sergioFavio/MaskRCNN_TensorFlow2.x.git

Mounted at /content/drive
Cloning into 'MaskRCNN_TensorFlow2.x'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 16 (delta 1), reused 10 (delta 0), pack-reused 0
Unpacking objects: 100% (16/16), done.
```

Figura 5.6: Ejecución de comandos para cargar el dataset desde el repositorio de Google Drive y framework MaskRCNN para detección de objetos.

Una vez ejecutados los comandos para habilitar el dataset como también el framework para la detección de objetos, se tiene la siguiente estructura de archivos:

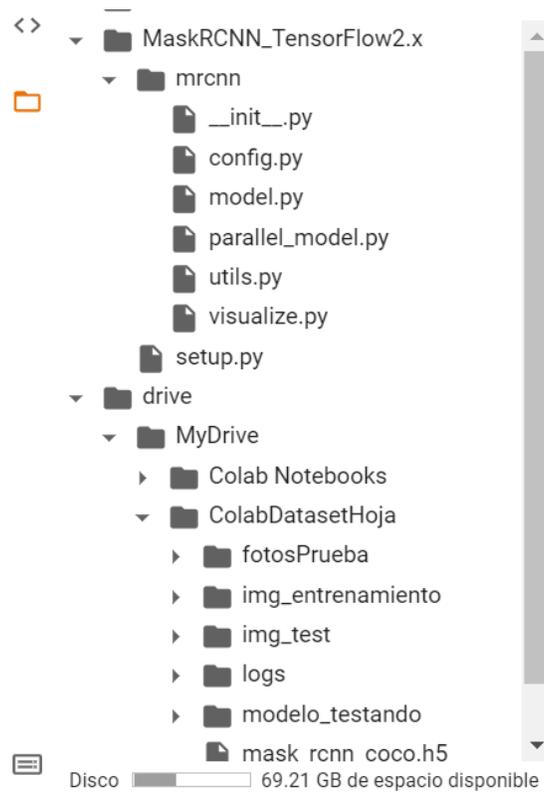


Figura 5.7: Estructura de archivos para realizar la detección.

Una vez ejecutados los comandos para cargar el dataset y la librería Mask RCNN, se procede a configurar la librería de Computer Vision para hacer detección de objetos.

```
▶ # corriendo el archivo setup para configurar la libreria de ... Mask_RCNN
# ... instalando setup.py
%cd /content/MaskRCNN_TensorFlow2.x
!ls
!pwd
!python setup.py install
```

Figura 5.8: Configuración de la librería Mask RCNN para realizar detección.

El código para realizar la detección de la hoja en la imagen se encuentra en el siguiente repositorio:

https://github.com/sergioFavio/Tesis-Magister-Ing.-Software-UNLP/blob/master/codigosDetectacClasifica/Detectando_hojas_Mask_R_CNN.ipynb

5.3.2.2.1 Configuración del modelo

La configuración del modelo se realiza utilizando el archivo **config.py** que se encuentra en la carpeta **mrcnn**, este archivo contiene información predefinida, por lo cual se tiene que especificar ciertos parámetros para ejecutar la detección de los objetos (hoja de tomate y no hoja de tomate), al invocar el modelo se tiene que pasar las configuraciones en la forma de una clase, actualizando los siguientes campos:

```
class ConfiguracionesHojas(Config):
    NAME = "configuraciones_hojas" # Dar a la configuración un nombre reconocible
    NUM_CLASSES = 1 + 2 # define o número de classes (background + HojaTomate + NoHojaTomate)

    # Todas muestras de imágenes de entrenamiento son 227x227 y 512x512
    IMAGE_MIN_DIM = 227
    IMAGE_MAX_DIM = 512

    STEPS_PER_EPOCH = 500 # número de pasos por época
    GPU_COUNT = 1 # cuantas GPUs son utilizadas
    IMAGES_PER_GPU = 2 # cuantas imagenes son pasadas para la GPU cada vez

    DETECTION_MAX_INSTANCES = 2 # número máximo de detección por imagen

    RPN_ANCHOR_SCALES = (8, 16, 32, 64, 128) # Longitud del lado del ancla cuadrada en píxeles

    # Número de ROI por imagen para alimentar a los cabezales de clasificador / enmascaramiento
    # El papel Mask RCNN usa 512 pero a menudo el RPN no genera
    # suficientes propuestas positivas para llenar esto y mantener un positivo: negativo
    # proporción de 1: 3. Puede aumentar el número de propuestas ajustando
    # el umbral de RPN NMS.
    TRAIN_ROIS_PER_IMAGE = 32
```

Figura 5.9: Actualización de algunos parámetros de la configuración del modelo a entrenarse.

La modificación de más o menos parámetros estará acorde al objetivo que se pretende en cada problema de detección de objetos.

5.3.2.2 Etiquetas de entrenamiento

El nombre que se asigne en las clases (etiquetas) *HojaTomate* y *NoHojaTomate*, debe ser el mismo que se empleó en la herramienta *labelImage* (incluyendo mayúsculas y espacio).

```
class DatasetHoja(Dataset):
    def cargar_dataset(self, direccion_dataset):
        self.add_class("dataset", 1, "HojaTomate")      # adicionar clase (etiqueta)
        self.add_class("dataset", 2, "NoHojaTomate")    # adicionar clase (etiqueta)
```

Figura 5.10: Asignación de etiquetas.

Por cada clase a detectar se asocia un número de secuencia creciente, como se aprecia en la figura 5.10, donde el primer argumento es el nombre del dataset (prefijo), el segundo argumento es el número de secuencia creciente (que debe comenzar en 1) dependiendo de la cantidad de clases a detectar, el tercer argumento está asociado con el nombre de la clase tal como se definió al hacer la anotación de las coordenadas del objeto en la imagen al utilizar la herramienta *labelImage*.

5.3.2.3 Entrenar el modelo

Llegando a esta etapa es crucial dentro del algoritmo de detección de objetos (HojaTomate, NoHojaTomate) y se procede a ejecutar los siguientes comandos para entrenar el modelo:

```
# Creando el modelo:
modelo = MaskRCNN(mode='training', model_dir='/content/drive/MyDrive/ColabDatasetHoja/logs/', config=config)
modelo.load_weights('/content/drive/MyDrive/ColabDatasetHoja/mask_rcnn_coco.h5', by_name=True, exclude=["mrcnn_class_logits", "mrcnn_bbox_fc", "mrcnn_bbox", "mrcnn_mask"])
modelo.train(dataset_img_entrenamiento, dataset_img_test, learning_rate=0.001, epochs=50, layers='heads')
```

Figura 5.11: Parte del código (recuadro) que ejecuta el entrenamiento del modelo.

Inmediatamente después de ejecutar el comando de entrenamiento, se aprecia lo siguiente:

```

mrcnn_bbox_fc          (TimeDistributed)
mrcnn_mask_deconv      (TimeDistributed)
mrcnn_class_logits     (TimeDistributed)
mrcnn_mask             (TimeDistributed)
Epoch 1/50
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:437: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("tr
"shape. This may consume a large amount of memory." % value)
500/500 [=====] - ETA: 0s - batch: 249.5000 - size: 2.0000 - loss: 0.4859 - rpn_class_loss: 0.0053 - rpn_bbox_loss: 0.2313 - mrcnn_class_loss:
warnings.warn('Model.state_updates' will be removed in a future version. '
500/500 [=====] - 685s 1s/step - batch: 249.5000 - size: 2.0000 - loss: 0.4859 - rpn_class_loss: 0.0053 - rpn_bbox_loss: 0.2313 - mrcnn_class_
Epoch 2/50
500/500 [=====] - 523s 1s/step - batch: 249.5000 - size: 2.0000 - loss: 0.2956 - rpn_class_loss: 0.0018 - rpn_bbox_loss: 0.1457 - mrcnn_class_

```

Figura 5.12: Inicio de la ejecución del entrenamiento del modelo.

El entrenamiento se ejecuta hasta alcanzar el número de *epochs* definido (Figura 5.11), obteniendo un *loss* (pérdida) lo más bajo posible (inferior a 0.9), ver siguiente figura:

```

Epoch 40/50
500/500 [=====] - 158s 315ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0901 - rpn_class_loss: 2.2352e-04 - rpn_bbox_loss: 0.0255 - mrcnn
Epoch 41/50
500/500 [=====] - 157s 315ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.1013 - rpn_class_loss: 3.8023e-04 - rpn_bbox_loss: 0.0359 - mrcnn
Epoch 42/50
500/500 [=====] - 158s 316ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.1019 - rpn_class_loss: 2.8244e-04 - rpn_bbox_loss: 0.0360 - mrcnn
Epoch 43/50
500/500 [=====] - 159s 318ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.1074 - rpn_class_loss: 3.5454e-04 - rpn_bbox_loss: 0.0424 - mrcnn
Epoch 44/50
500/500 [=====] - 158s 315ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0921 - rpn_class_loss: 3.4166e-04 - rpn_bbox_loss: 0.0321 - mrcnn
Epoch 45/50
500/500 [=====] - 158s 317ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0835 - rpn_class_loss: 2.8800e-04 - rpn_bbox_loss: 0.0217 - mrcnn
Epoch 46/50
500/500 [=====] - 159s 318ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0932 - rpn_class_loss: 3.4533e-04 - rpn_bbox_loss: 0.0290 - mrcnn
Epoch 47/50
500/500 [=====] - 157s 315ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0764 - rpn_class_loss: 2.8295e-04 - rpn_bbox_loss: 0.0194 - mrcnn
Epoch 48/50
500/500 [=====] - 159s 318ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0898 - rpn_class_loss: 3.3971e-04 - rpn_bbox_loss: 0.0263 - mrcnn
Epoch 49/50
500/500 [=====] - 158s 316ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0776 - rpn_class_loss: 2.3195e-04 - rpn_bbox_loss: 0.0207 - mrcnn
Epoch 50/50
500/500 [=====] - 159s 317ms/step - batch: 249.5000 - size: 2.0000 - loss: 0.0811 - rpn_class_loss: 1.8036e-04 - rpn_bbox_loss: 0.0211 - mrcnn

```

Figura 5.13: Se termina la ejecución del entrenamiento cuando se cumple el número de *epochs* y se consigue un *loss* con un valor más bajo posible (inferior a 0.9).

5.3.2.4 Guardar modelo entrenado para ejecutarlo posteriormente

Una vez finalizado el entrenamiento en la carpeta *logs* (figura 5.15) se guardan los archivos de los modelos creados (uno por cada epoch con la extensión .h5), a continuación parte del código que ejecuta la grabación del modelo.

```
# Creando el modelo:  
modelo = MaskRCNN(mode='training', model_dir='/content/drive/MyDrive/ColabDatasetHoja/logs/', config=config)  
modelo.load_weights('/content/drive/MyDrive/ColabDatasetHoja/mask_rcnn_coco.h5', by_name=True, exclude=['mrcnn_c  
modelo.train(dataset_img_entrenamiento, dataset_img_test, learning_rate=0.001, epochs=50, layers='heads')
```

Figura 5.14: Código (recuadro) que ejecuta en que carpeta (logs) se guarda el modelo creado.

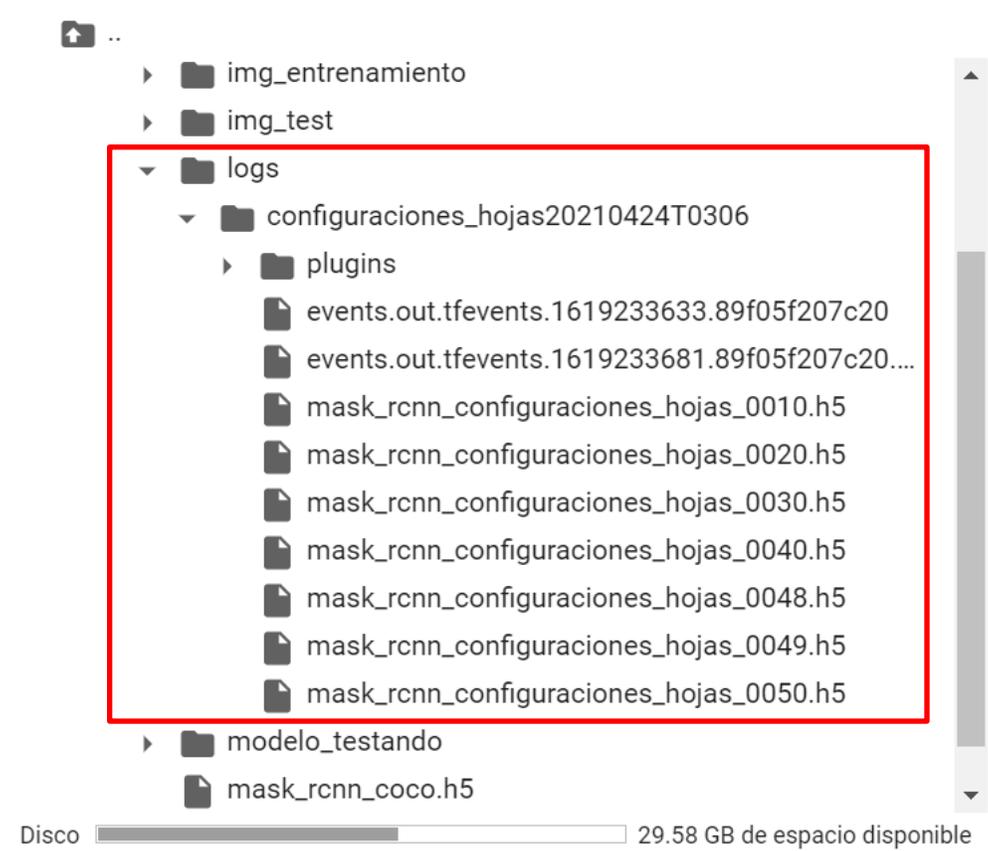


Figura 5.15: Carpeta logs con parte de los modelos creados por cada epoch.

5.3.3 Predicción y despliegue

En esta última fase de la detección de objetos (HojaTomate, NoHojaTomate) es el momento de probar el modelo para realizar predicciones, para ello guardamos imágenes de prueba en la carpeta *fotosPrueba* (figura 5.7) y corremos el siguiente código.

```
# haciendo predicciones y despliegue ...
import skimage
import os
import numpy as np
from mrcnn.utils import extract_bboxes
from mrcnn.visualize import display_instances
from mrcnn.config import Config
from mrcnn.model import MaskRCNN

# Creando una clase para las configuraciones de las predicciones:
class ConfiguracionesHojas(Config):
    NAME = "configuraciones_hojas"
    NUM_CLASSES = 1 + 2 # define el número de classes (background + HojaTomate + NoHojaTomate)
    IMAGE_MIN_DIM = 227
    IMAGE_MAX_DIM = 512
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    DETECTION_MAX_INSTANCES = 1
real_test_dir = '/content/drive/MyDrive/ColabDatasetHoja/fotosPrueba'
cfg = ConfiguracionesHojas()
modelo = MaskRCNN(mode='inference', model_dir='/content/drive/MyDrive/ColabDatasetHoja/modelo_testando/', config=cfg)
modelo.load_weights('/content/drive/MyDrive/ColabDatasetHoja/logs/configuraciones_hojas20210424T0306/mask_rcnn_configuraciones_hojas_0050.h5', by_name=True)
image_paths = []
for filename in os.listdir(real_test_dir):
    if os.path.splitext(filename)[1].lower() in ['.png', '.jpg', '.jpeg']:
        image_paths.append(os.path.join(real_test_dir, filename))
for image_path in image_paths:
    img = skimage.io.imread(image_path)
    img_arr = np.array(img)
    results = modelo.detect([img_arr], verbose=1)
    r = results[0]
    display_instances(img, r['rois'], r['masks'], r['class_ids'],
                    dataset_img_test.class_names, r['scores']*100, figsize=(15,10))
```

Figura 5.16: Código para hacer predicciones y despliegue.

El resultado se obtiene las siguientes imágenes de salida.

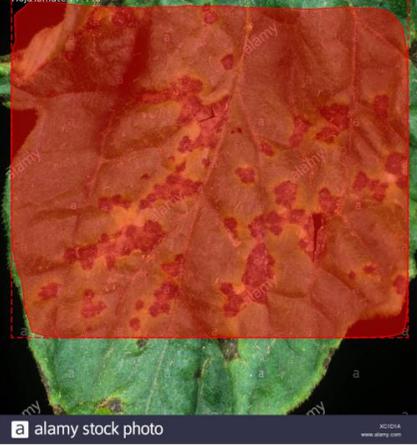
imagen entrada	Imagen salida
 <p>A photograph of a green leaf with several brown, necrotic spots. The leaf is set against a black background. A watermark 'alamy stock photo' is visible at the bottom.</p>	 <p>The same leaf as in the input image, but with a semi-transparent red overlay covering most of its surface. A red bounding box is visible around the leaf. A watermark 'alamy stock photo' is visible at the bottom.</p>
 <p>A photograph of a green leaf with several brown, necrotic spots, similar to the first image. The leaf is set against a dark background.</p>	 <p>The same leaf as in the input image, but with a semi-transparent red overlay covering most of its surface. A red bounding box is visible around the leaf. A watermark 'HojaTomate 72 116' is visible at the top left.</p>
 <p>A photograph of a tiger sitting in a grassy field, eating a large carrot. The background is a bamboo forest. A watermark '© 2007 Susan L. Pettitt Photography. All Rights Reserved.' is visible at the bottom.</p>	 <p>The same tiger as in the input image, but with a semi-transparent red overlay covering its body. A red bounding box is visible around the tiger. A watermark 'NoHojaTomate 70 640' is visible at the top left.</p>

Figura 5.17: Imágenes de ingreso (columna izquierda), imágenes de salida (columna derecha).

5.4 Algoritmo de clasificación de enfermedades

Para realizar la clasificación de enfermedades: tizón temprano, mancha bacteriana y tomate sano (no hay presencia de enfermedad) que se presentan en la hoja de tomate, se utiliza el modelo VGG16 pre entrenado (transfer learning), que fue desarrollado por Visual Graphics (VGG) en Oxford.

5.4.1 Preparación de la data

Se dividen los datos en dos partes: entrenamiento con 3.018 imágenes y validación con 754 imágenes. Con una distribución de 80% para entrenamiento y 20% para validación. Tener en cuenta que se hace la división de los datos al azar en estos dos conjuntos para garantizar la menor cantidad de sesgo que podría infiltrarse sin saberlo. La precisión final del modelo es determinada por la precisión en el conjunto de validación.

El modelo aprende de los datos de entrenamiento y usa el conjunto de validación para evaluar su desempeño. Los profesionales del aprendizaje automático toman este desempeño como retroalimentación para encontrar oportunidades para mejorar sus modelos de forma continua. Hay varios parámetros que podemos sintonizar para mejorar el rendimiento; por ejemplo, el número de capas para entrenar.

Las redes neuronales algunas veces son tan profundas y potentes que pueden memorizar datos de entrenamiento, lo que incluso da como resultado una precisión del 100% en los datos de entrenamiento. Sin embargo, su desempeño en el mundo real será bastante pobre. Es por eso que un conjunto de validación, no se usa para entrenar el modelo, proporciona una evaluación realista del rendimiento del modelo. Aunque se podría asignar un 20-25% de los datos como validación establecido, será de gran ayuda para guiarnos sobre qué tan bueno realmente es nuestro modelo.

Para el proceso de entrenamiento se necesita almacenar el conjunto de datos en una estructura de carpetas. Se dividen las imágenes en dos conjuntos: entrenamiento y validación. Para un archivo de imagen, Keras asignará automáticamente el nombre de la clase (categoría) según el nombre de la carpeta principal. La Figura 5.18 muestra la estructura ideal para recrear [16].

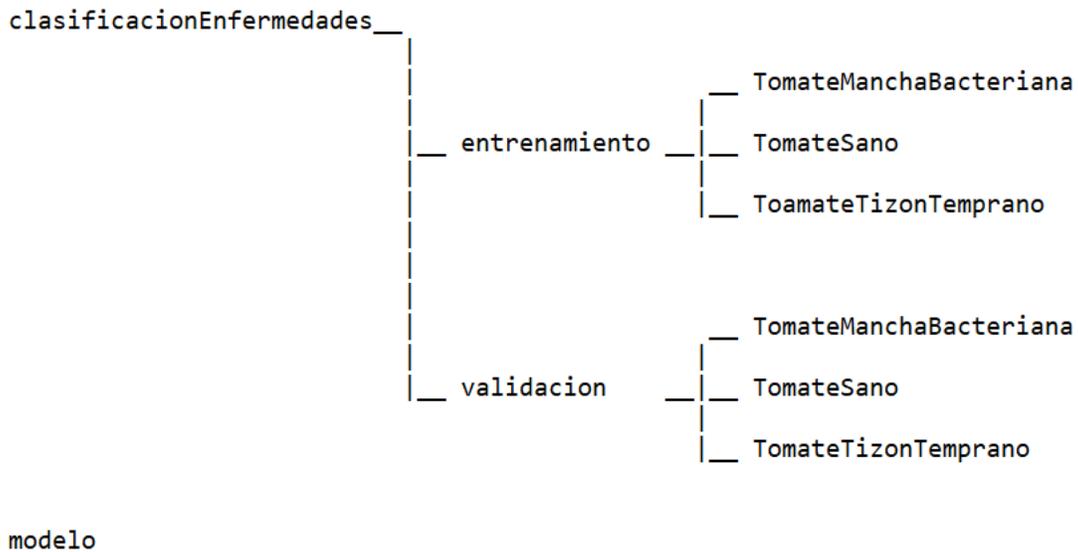


Figura 5.18: Estructura de archivos para la clasificación de enfermedades en la hoja de tomate.

Siguiendo el flujo de la tubería de procesamiento (figura 5.2) la parte de Preparación de la Data (Prepare Data) se tiene la data de las imágenes depuradas de las enfermedades en las carpetas *TomateManchaBacteriana*, *TomateSano* y *TomateTizonTemprano* con un total de 3.018 imágenes.

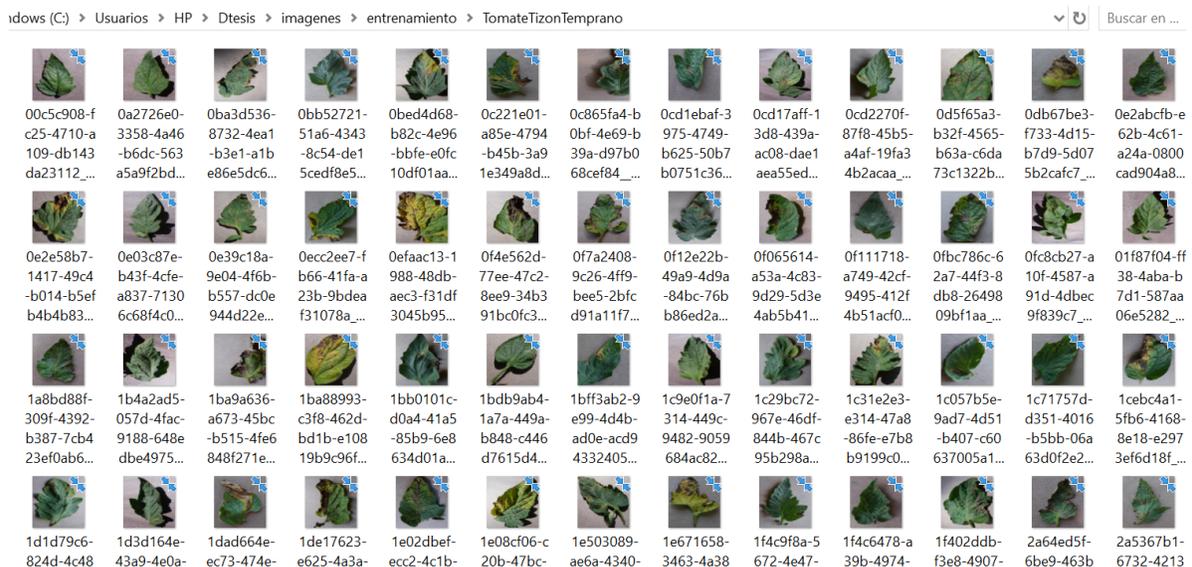


Figura 5.19: Archivos de imágenes en el directorio: clasificacionEnfermedades/entrenamiento/TomateTizonTemprano.

5.4.2 Construcción y entrenamiento del modelo (Build & Train Models)

Ahora que los datos ya están habilitados, llegamos al punto más crucial, la construcción del modelo.

5.4.2.1 Elección del modelo

En cuanto a la elección del modelo se decidió utilizar el modelo pre entrenado VGG16. En el código que sigue a continuación, contempla las siguientes partes:

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

from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPool2D, Dense, Flatten, Dropout, BatchNormalization, Input
from keras.optimizers import Adam
from keras.callbacks import TensorBoard, ModelCheckpoint
from keras.utils import np_utils
import os
import numpy as np
from keras.preprocessing import image
from keras.applications.imagenet_utils import preprocess_input, decode_predictions
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import ImageDataGenerator
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
```

Figura 5.20: Cargando repositorio de Google Drive del dataset e importando librerías.

Configuración de parámetros

```
[ ] width_shape = 224
    height_shape = 224
    num_classes = 3
    epochs = 100
    batch_size = 32
```

Figura 5.21: Configurando algunos hiper parámetros.

Path de dataset

```
[ ] train_data_dir = '../content/drive/MyDrive/clasificacionEnfermedades/entrenamiento' # 'dataset/train'  
validation_data_dir = '../content/drive/MyDrive/clasificacionEnfermedades/validacion' # 'dataset/valid'
```

Figura 5.22: Direccionando carpetas con los archivos de imágenes.

Generador de imágenes (entrenamiento y validación)

```
▶ train_datagen = ImageDataGenerator(  
    rotation_range=20,  
    zoom_range=0.2,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    horizontal_flip=True,  
    vertical_flip=False,  
    preprocessing_function=preprocess_input)  
  
valid_datagen = ImageDataGenerator(  
    rotation_range=20,  
    zoom_range=0.2,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    horizontal_flip=True,  
    vertical_flip=False,  
    preprocessing_function=preprocess_input)  
  
train_generator = train_datagen.flow_from_directory(  
    train_data_dir,  
    target_size=(width_shape, height_shape),  
    batch_size=batch_size,  
    #save_to_dir='',  
    class_mode='categorical')  
  
validation_generator = valid_datagen.flow_from_directory(  
    validation_data_dir,  
    target_size=(width_shape, height_shape),  
    batch_size=batch_size,
```

Figura 5.23: Normalizando las imágenes.

5.4.2.2 Entrenar el modelo

Esta etapa crucial comprende el siguiente código:

Entrenamiento de modelo VGG16

```
nb_train_samples = 3018 # 880 # 1490
nb_validation_samples = 754 # 220 # 50

image_input = Input(shape=(width_shape, height_shape, 3))

model = VGG16(input_tensor=image_input, include_top=True, weights='imagenet')

last_layer = model.get_layer('fc2').output
out = Dense(num_classes, activation='softmax', name='output')(last_layer)
custom_vgg_model = Model(image_input, out)

for layer in custom_vgg_model.layers[:-1]:
    layer.trainable = False

custom_vgg_model.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=['accuracy'])

custom_vgg_model.summary()

model_history = custom_vgg_model.fit_generator(
    train_generator,
    epochs=epochs,
    validation_data=validation_generator,
    steps_per_epoch=nb_train_samples//batch_size,
    validation_steps=nb_validation_samples//batch_size)
```

Figura 5.24: Código de entrenamiento del modelo.

El código se ejecuta hasta cumplir el número de *epochs* definido previamente y obtener un *loss* inferior a 0.9.

```

/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1844: UserWarning: `Model.fit_generator` is depreca
warnings.warn("`Model.fit_generator` is deprecated and '
Epoch 1/100
94/94 [=====] - 59s 529ms/step - loss: 1.2778 - accuracy: 0.4347 - val_loss: 1.2348 - val_accuracy: 0.4592
Epoch 2/100
94/94 [=====] - 47s 500ms/step - loss: 1.2318 - accuracy: 0.4471 - val_loss: 1.1609 - val_accuracy: 0.4796
Epoch 3/100
94/94 [=====] - 47s 501ms/step - loss: 1.1716 - accuracy: 0.4702 - val_loss: 1.1034 - val_accuracy: 0.5068
Epoch 4/100
94/94 [=====] - 47s 501ms/step - loss: 1.1015 - accuracy: 0.4975 - val_loss: 1.1010 - val_accuracy: 0.4973
Epoch 5/100
94/94 [=====] - 47s 502ms/step - loss: 1.0960 - accuracy: 0.4881 - val_loss: 1.0173 - val_accuracy: 0.5448
Epoch 6/100
94/94 [=====] - 47s 503ms/step - loss: 1.0665 - accuracy: 0.5069 - val_loss: 1.0318 - val_accuracy: 0.5435
Epoch 7/100
94/94 [=====] - 47s 501ms/step - loss: 1.0558 - accuracy: 0.5223 - val_loss: 1.0096 - val_accuracy: 0.5503
Epoch 8/100
94/94 [=====] - 47s 503ms/step - loss: 0.9984 - accuracy: 0.5443 - val_loss: 1.0344 - val_accuracy: 0.5285
Epoch 9/100
94/94 [=====] - 47s 500ms/step - loss: 0.9850 - accuracy: 0.5379 - val_loss: 0.9380 - val_accuracy: 0.5788
Epoch 10/100
94/94 [=====] - 47s 500ms/step - loss: 0.9508 - accuracy: 0.5453 - val_loss: 0.9532 - val_accuracy: 0.5761
Epoch 11/100
94/94 [=====] - 47s 501ms/step - loss: 0.9493 - accuracy: 0.5596 - val_loss: 0.8978 - val_accuracy: 0.5978
Epoch 12/100
94/94 [=====] - 47s 502ms/step - loss: 0.9371 - accuracy: 0.5841 - val_loss: 0.8948 - val_accuracy: 0.6005
Epoch 13/100
94/94 [=====] - 47s 504ms/step - loss: 0.9034 - accuracy: 0.5940 - val_loss: 0.8532 - val_accuracy: 0.6318
Epoch 14/100
94/94 [=====] - 47s 504ms/step - loss: 0.9079 - accuracy: 0.6040 - val_loss: 0.8550 - val_accuracy: 0.6250
Epoch 15/100
94/94 [=====] - 47s 501ms/step - loss: 0.9006 - accuracy: 0.6012 - val_loss: 0.8725 - val_accuracy: 0.6209

```

Figura 5.25: Inicio de la ejecución del entrenamiento del modelo.

```

Epoch 86/100
94/94 [=====] - 48s 506ms/step - loss: 0.4138 - accuracy: 0.8489 - val_loss: 0.4505 - val_accuracy: 0.8274
Epoch 87/100
94/94 [=====] - 48s 509ms/step - loss: 0.4364 - accuracy: 0.8529 - val_loss: 0.4480 - val_accuracy: 0.8193
Epoch 88/100
94/94 [=====] - 47s 504ms/step - loss: 0.4405 - accuracy: 0.8358 - val_loss: 0.4596 - val_accuracy: 0.8084
Epoch 89/100
94/94 [=====] - 47s 503ms/step - loss: 0.4407 - accuracy: 0.8351 - val_loss: 0.4261 - val_accuracy: 0.8438
Epoch 90/100
94/94 [=====] - 47s 504ms/step - loss: 0.4153 - accuracy: 0.8348 - val_loss: 0.4489 - val_accuracy: 0.8370
Epoch 91/100
94/94 [=====] - 47s 503ms/step - loss: 0.4302 - accuracy: 0.8390 - val_loss: 0.4349 - val_accuracy: 0.8315
Epoch 92/100
94/94 [=====] - 47s 502ms/step - loss: 0.4167 - accuracy: 0.8431 - val_loss: 0.4448 - val_accuracy: 0.8329
Epoch 93/100
94/94 [=====] - 47s 505ms/step - loss: 0.4075 - accuracy: 0.8524 - val_loss: 0.4419 - val_accuracy: 0.8234
Epoch 94/100
94/94 [=====] - 47s 505ms/step - loss: 0.4237 - accuracy: 0.8441 - val_loss: 0.4352 - val_accuracy: 0.8329
Epoch 95/100
94/94 [=====] - 47s 503ms/step - loss: 0.4117 - accuracy: 0.8589 - val_loss: 0.4323 - val_accuracy: 0.8451
Epoch 96/100
94/94 [=====] - 47s 503ms/step - loss: 0.3973 - accuracy: 0.8510 - val_loss: 0.4298 - val_accuracy: 0.8397
Epoch 97/100
94/94 [=====] - 48s 507ms/step - loss: 0.4041 - accuracy: 0.8485 - val_loss: 0.4304 - val_accuracy: 0.8302
Epoch 98/100
94/94 [=====] - 47s 504ms/step - loss: 0.4193 - accuracy: 0.8449 - val_loss: 0.4191 - val_accuracy: 0.8397
Epoch 99/100
94/94 [=====] - 48s 506ms/step - loss: 0.4050 - accuracy: 0.8473 - val_loss: 0.4191 - val_accuracy: 0.8342
Epoch 100/100
94/94 [=====] - 48s 507ms/step - loss: 0.4119 - accuracy: 0.8507 - val_loss: 0.4255 - val_accuracy: 0.8329

```

Figura 5.26: Se termina la ejecución del entrenamiento cuando se alcanza el número de *epochs* definido y se obtiene un *loss* tiene un valor inferior a 0.9.

5.4.2.3 Guardar modelo entrenado para ejecutarlo posteriormente

Una vez finalizado el entrenamiento en la carpeta *modelo* (figura 4.18) se guarda el archivo del modelo creado con la extensión *.h5*, a continuación parte del código que ejecuta la grabación del modelo.

▾ Grabar modelo en disco

```
[ ] custom_vgg_model.save("../content/drive/MyDrive/modelo/model_VGG16.h5")
```

Figura 5.27: Grabar el modelo en disco.

5.4.3 Predicción y despliegue

En esta última etapa de la clasificación de enfermedades es el momento de probar el modelo para realizar predicciones y desplegar los resultados, para ello se crea el siguiente script de predicción.

```
from keras.applications.imagenet_utils import preprocess_input, decode_predictions
from keras.models import load_model

names = ['TOMATE MANCHA BACTERIANA', 'TOMATE SANO', 'TOMATE TIZON TEMPRAMO']

modelt = load_model("../content/drive/MyDrive/modelo/model_VGG16.h5")

imaget_path = "tomateSano10.JPG"
imaget=cv2.resize(cv2.imread(imaget_path), (width_shape, height_shape), interpolation = cv2.INTER_AREA)
xt = np.asarray(imaget)
xt=preprocess_input(xt)
xt = np.expand_dims(xt,axis=0)
preds = modelt.predict(xt)

print('preds[0]:',preds[0]) # ... tiene las probabilidades por cada clase ...
print(names[np.argmax(preds)],preds[0][np.argmax(preds)]*100.0,'%')
plt.imshow(cv2.cvtColor(np.asarray(imaget),cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

Figura 5.28: Importando librerías y cargando el modelo ya entrenado.

Una vez ejecutado el programa de predicción, se obtiene el siguiente resultado.

```
preds[0]: [0.06279553 0.86858475 0.0686197 ]  
TOMATE SANO 86.85847520828247 %
```



Figura 5.29: Resultado de la ejecución del programa de predicción de clasificación.

En base a la figura anterior se verifica que la clasificación ha sido realizada de manera exitosa, observando el valor arrojado contrastándolo con los valores definidos en el código del programa.

El código para realizar la clasificación de las enfermedades se encuentra en el siguiente repositorio:

<https://github.com/sergioFavio/Tesis-Magister-Ing.-Software-UNLP/blob/master/codigosDetectacClasifica/ClasificarImagenesVGG16.ipynb>

Capítulo 6: Resultados

En este capítulo se presentan los resultados alcanzados en la implementación (capítulo 5) a partir de los objetivos específicos definidos en el primer capítulo, contando con las siguientes subsecciones: Análisis de Resultados y Rendimientos Obtenidos.

6.1 Análisis de Resultados

Tomando en cuenta los objetivos específicos, se procede a la evaluación de los mismos.

Objetivo 1: Construcción de un dataset con imágenes etiquetadas para entrenar y probar los algoritmos de detección y clasificación automática de enfermedades del tomate.

Tal como se menciona en el capítulo 5, incisos 5.3.1 y 5.4.1 (Preparación de la data) para los datasets de imágenes de Detección y Clasificación.

Dataset	Total imágenes	Imágenes entrenamiento	Imágenes test
Detección	3.916	3.133	783
Clasificación	3.772	3.018	754

Tabla 6.1: Cifras numéricas de los dataset empleados.

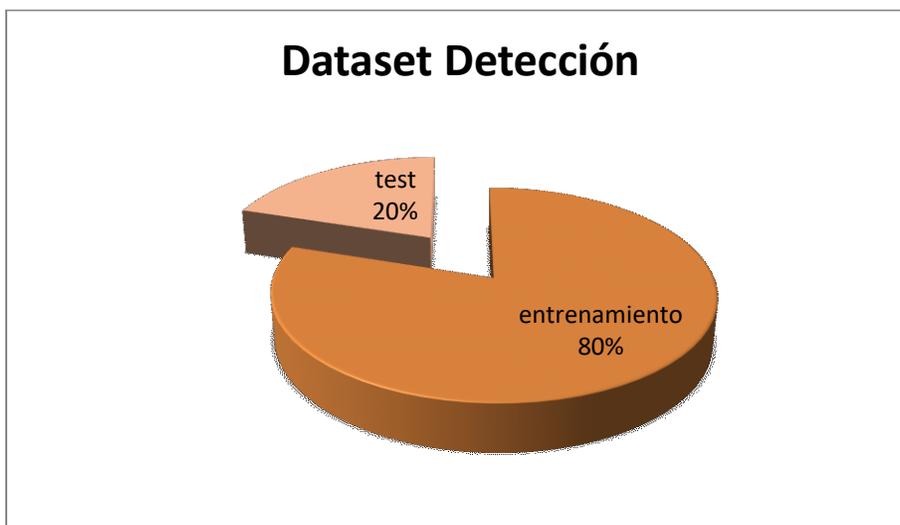


Figura 6.1: Porcentajes destinados a entrenamiento y test del dataset para Detección.

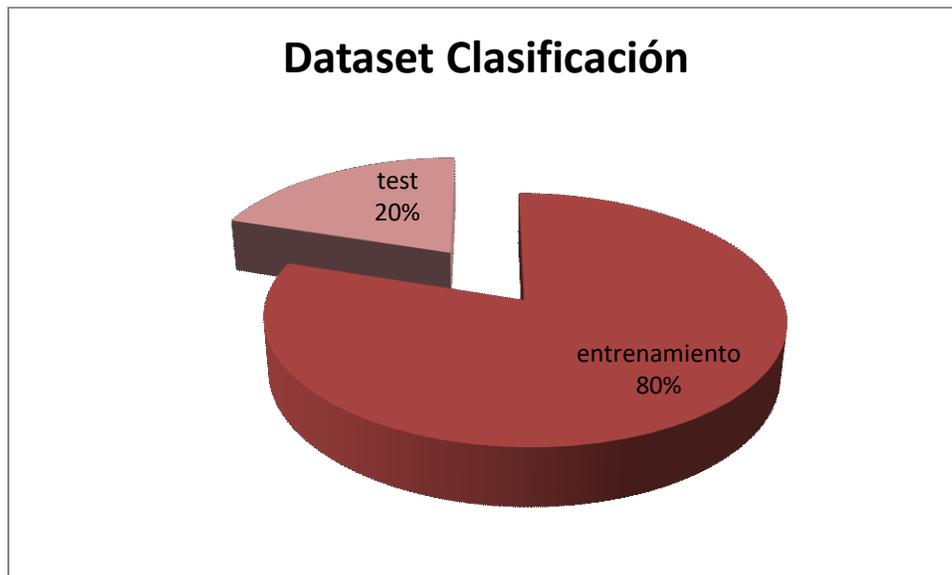


Figura 6.2: Porcentajes destinados a entrenamiento y test del dataset para Clasificación.

El dataset de Detección contempla dos categorías: HojaTomate y NoHojaTomate

categoria	Total imágenes	Imágenes entrenamiento	Imágenes test
HojaTomate	3.767	3.014	753
NoHojaTomate	149	119	30

Tabla 6.2: Datos numéricos de las categorías del dataset de Detección.

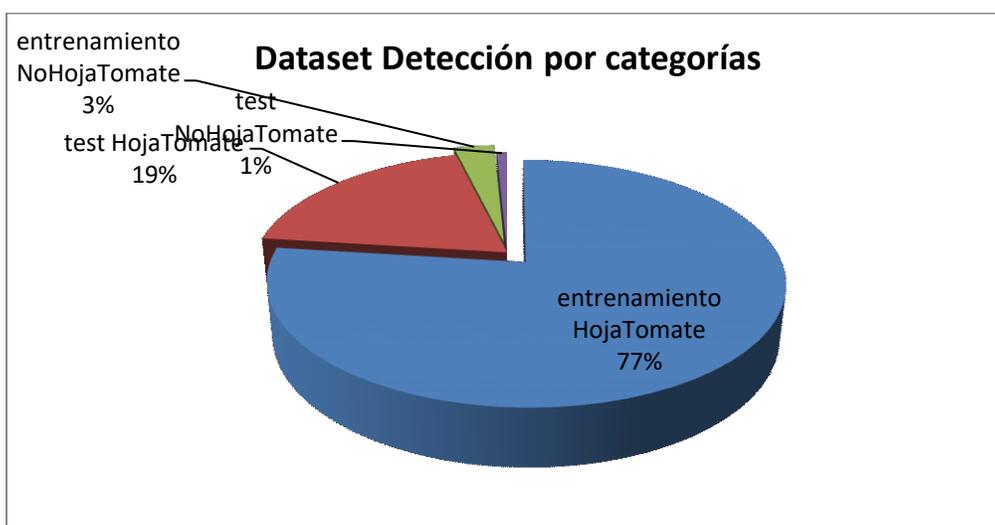


Figura 6.3: Porcentajes destinados a entrenamiento y test por categorías del dataset de Detección.

El dataset de Clasificación contempla tres categorías: Mancha Bacteriana, Tizón Temprano y Tomate Sano.

catgoría	Total imágenes	Imágenes entrenamiento	Imágenes test
ManchaBacteriana	1.700	1.361	339
TizonTemprano	800	640	160
TomateSano	1.272	1.017	255

Tabla 6.3: Datos numéricos de las categorías del dataset de Clasificación.

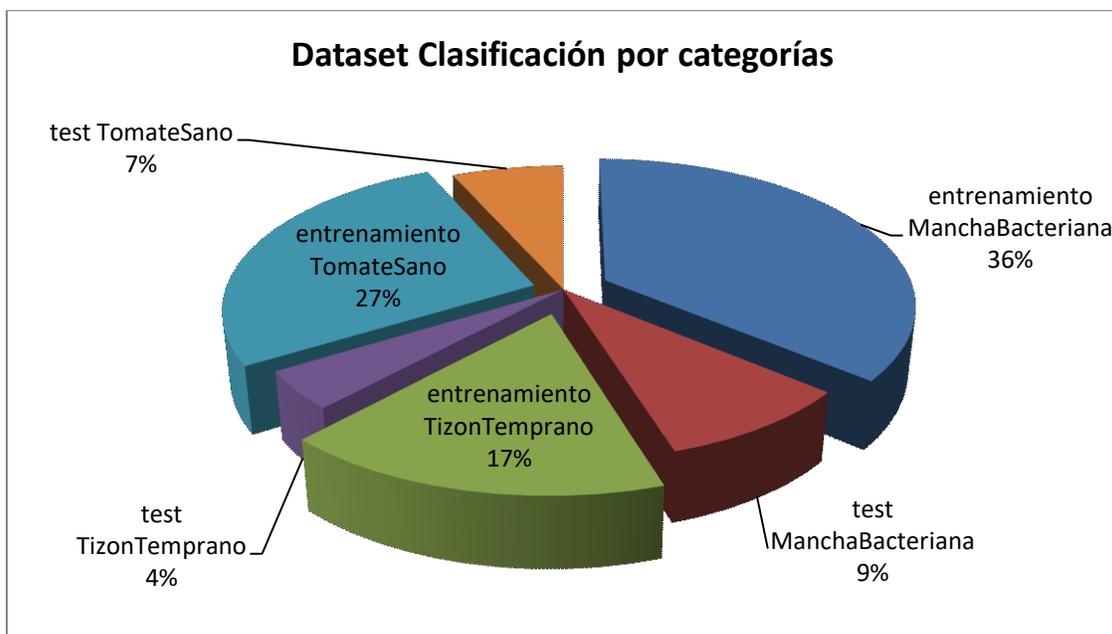


Figura 6.4: Porcentajes destinados a entrenamiento y test por categorías del dataset de Clasificación.

Objetivo 2: Describir las características relevantes de los atributos y de las etiquetas que conformarán el modelo.

Al emplear framework para Deep Learning (TensorFlow y Keras) tanto para las tareas de detección de hoja de tomate en las imágenes como de clasificación de enfermedades que se presentan en la hoja de tomate, la definición y extracción de características es de forma automática por parte del framework, no habiendo intervención humana, tal como se menciona en el capítulo cuatro inciso 4.3.2 (Metodología para la clasificación de imágenes).

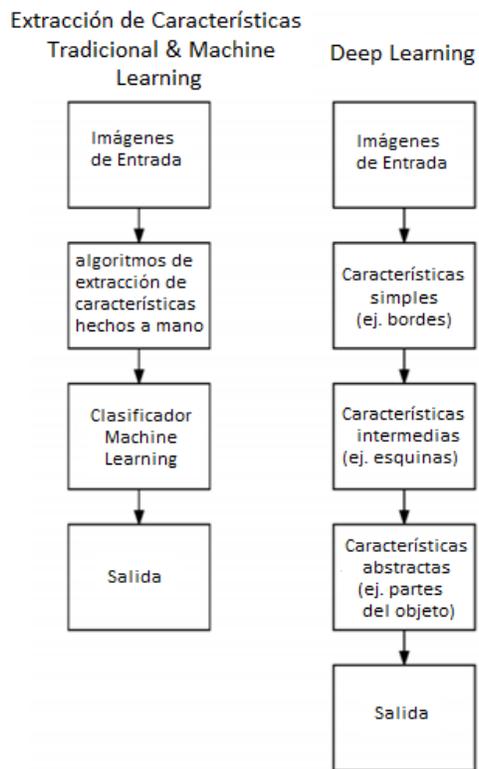


Figura 6.5: Izquierda - Proceso tradicional de tomar un conjunto de imágenes de entrada, aplicando diseños algoritmos de extracción de características, seguidos del entrenamiento de un clasificador de aprendizaje automático en las características. Derecha - Enfoque de aprendizaje profundo de apilar capas una encima de la otra que aprenden automáticamente características más complejas, abstractas y discriminatorias [21].

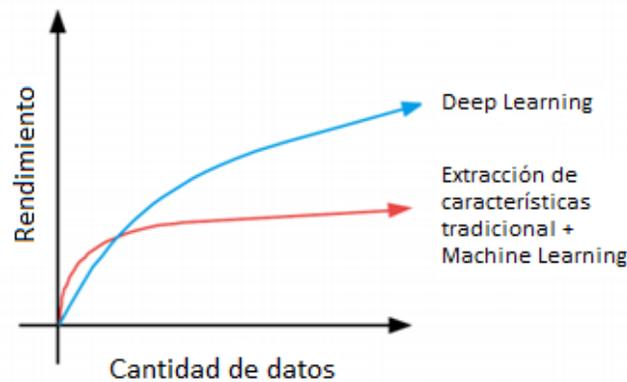


Figura 6.6: A medida que aumenta la cantidad de datos disponibles para los algoritmos de aprendizaje profundo, la precisión mejora notablemente, superando sustancialmente los enfoques tradicionales de extracción de características + aprendizaje automático.

Objetivo 3: Determinar el rendimiento del modelo entrenado.

El análisis de resultado de este objetivo se lo describe en un acápite aparte 6.2 Rendimientos Obtenidos, con más detalle.

Objetivo 4: Incorporar métricas para definir el modelo que tiene mejor representación.

Tanto en el modelo de detección como de clasificación se incorporan las siguientes métricas:

Modelo para Detección se utiliza *loss* (función costo o pérdida) como métrica para evaluar la confiabilidad del mismo, alcanzándose un valor de $loss=0.4119$ utilizando el framework de Tensorflow, tal como se aprecia en la figura 5.26 (capítulo 5, en la media que la curva se aproxime a cero (figura 6.7) aumenta la precisión del modelo entrenado.

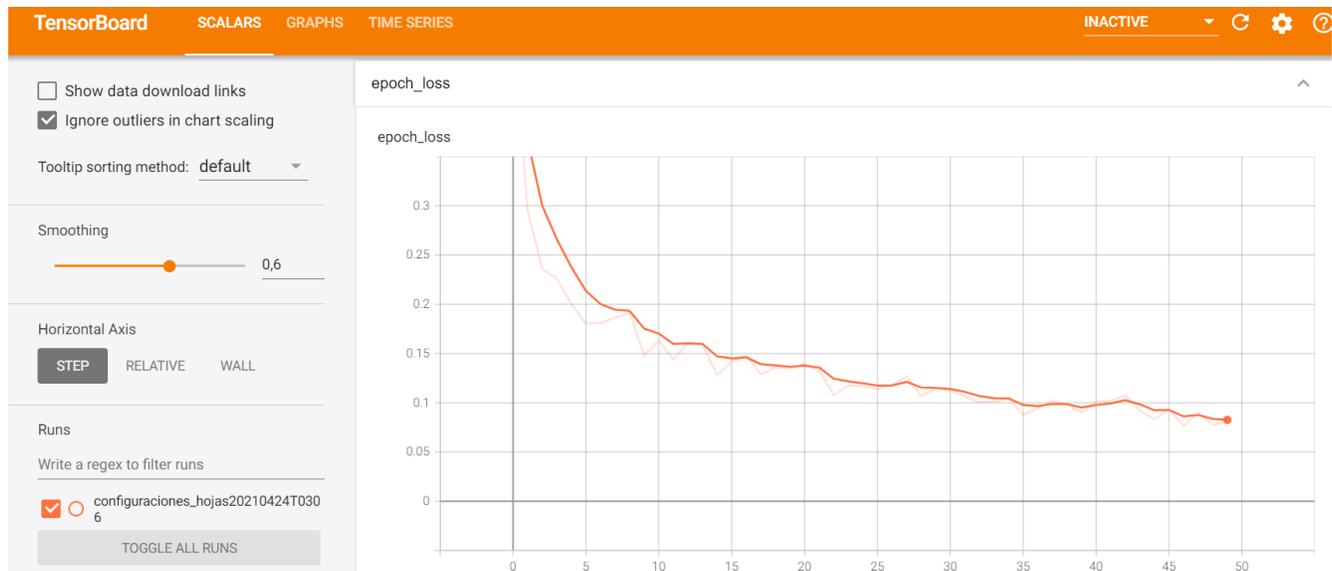


Figura 6.7: Visualización gráfica de las pérdidas (loss) de entrenamiento y validación.

Modelo para Clasificación se utilizan las métricas de *loss* (función de pérdida) y *acc* (accuracy: precisión) para evaluar la confiabilidad del modelo entrenado empleando el framework de Keras, arrojando los siguientes valores para $loss=0.4119$ y $acc=0.8507$ figura: 5.26 (capítulo 5).

Finalizado el proceso de entrenamiento del modelo inciso 5.4.2.2 en el capítulo cuatro se generan los siguientes gráficos:

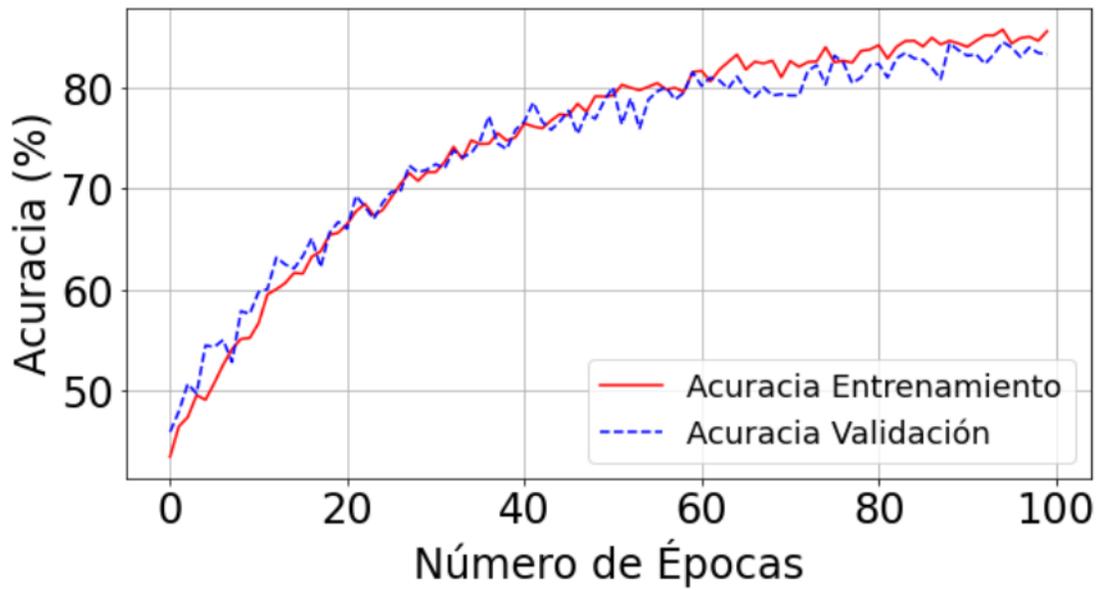


Figura 6.8: Visualización gráfica de las acuracias de entrenamiento y validación.

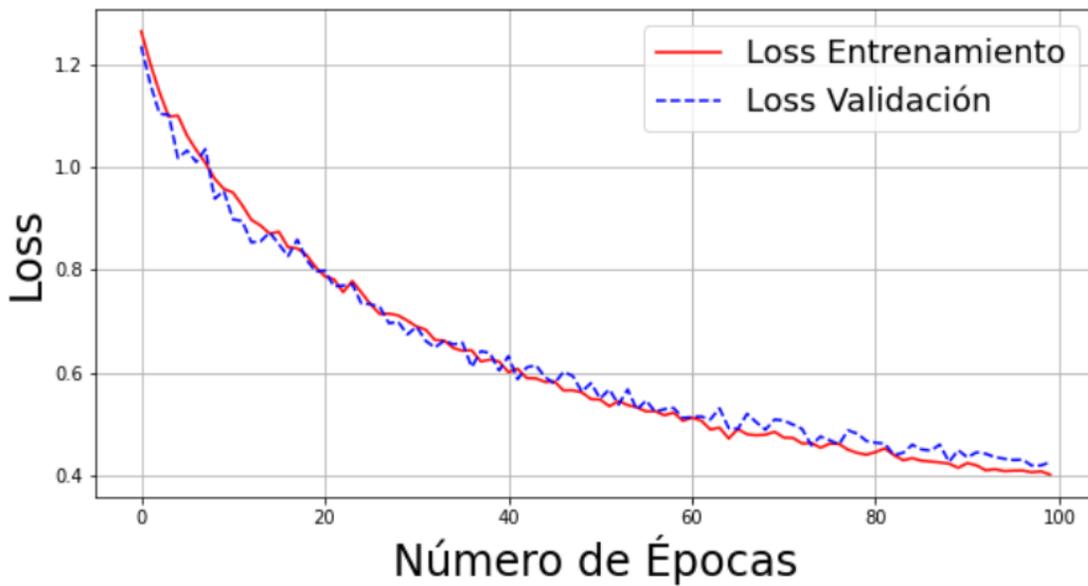


Figura 6.9: Visualización gráfica de las pérdidas (loss) de entrenamiento y validación.

Por lo tanto con los valores obtenidos tanto del modelo de detección como de clasificación dan aval sobre la confiabilidad de los mismos.

Objetivo 5: Adquirir criterios para determinar que algoritmos se deben aplicar dentro del marco de la construcción del modelo.

Examinando la literatura técnica, se recomienda como mejor algoritmo para la detección de objetos en imágenes Faster R-CNN (Transfer Learning) principalmente por el *Test time per image=0.2 seconds* y *Speed-up from R-CNN=250x*, en comparación a los otros algoritmos R-CNN Y Fast R-CNN, como se aprecia en la siguiente figura.

	R-CNN	Fast R-CNN	Faster R-CNN
mAP on PASCAL VOC2007	66.0%	66.9%	66.9%
Features	<ol style="list-style-type: none"> 1. Applies selective search to extract regions of interest (~2,000) from each image. 2. A ConvNet is used to extract features from each of the 2,000 regions extracted 3. Classification and bounding box predictions. 	<p>Each image is passed only once to the CNN and feature maps are extracted.</p> <ol style="list-style-type: none"> 1. A ConvNet is used to extract features maps from the input image. 2. Selective search is used on these maps to generate predictions. <p>This way we only run one ConvNet over the entire image instead of 2000 ConvNets over 2000 overlapping regions.</p>	<p>Replaces the selective search method with region proposal network (RPN) which makes the algorithm much faster.</p> <p>An end-to-end deep learning network.</p>
Limitations	High computation time as each region is passed to the CNN separately. Also, it uses three different models for making predictions.	Selective search is slow and hence computation time is still high.	Object proposal takes time and as there are different systems working one after the other, the performance of systems depends on how the previous system has performed.
Test time per image	50 seconds	2 seconds	0.2 seconds
Speed-up from R-CNN	1x	25x	250x

Figura 6.10: Cuadro comparativo de los algoritmos de detección.

	RCNN	Fast R-CNN	Faster R-CNN
mAP on Pascal VOC 2007	66.0%	66.9%	66.9%
Test time per image	50 seconds	2 seconds	0.2 seconds
Speed-up from R-CNN	1x	25x	250x

Tabla 6.4: Métricas cuantitativas de la figura 6.11.

En cuanto a la clasificación se decidió utilizar un algoritmo pre entrenado (transfer learning) por la razón de tener un mejor rendimiento versus un modelo entrenado desde cero (training from scratch).

Objetivo 6: Describir los desafíos del Deep Learning en el marco de los procesos de ingeniería de software. Son descritos en el capítulo tres.

6.2 Rendimientos Obtenidos

Se realizaron las siguientes pruebas tanto para detección de la hoja de tomate en la imagen como también para la clasificación de las enfermedades, los rendimientos de cada modelo entrenado se observan en los siguientes cuadros.

Detección

Clase	Imágenes	Aciertos	Desaciertos	Tasa de aciertos %
Hoja de tomate	50	41	9	82
No hoja de tomate	50	46	4	92

Tabla 6.5: Rendimiento algoritmo de Detección.

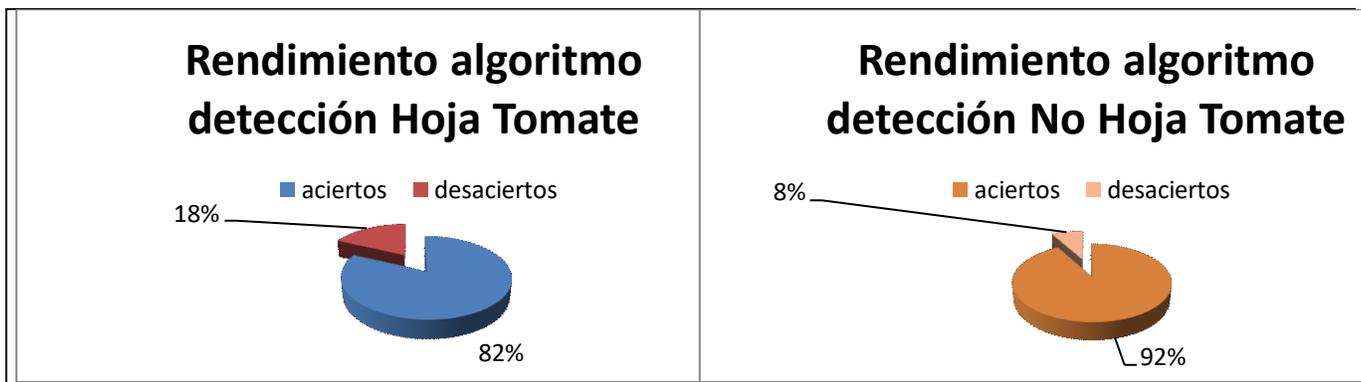


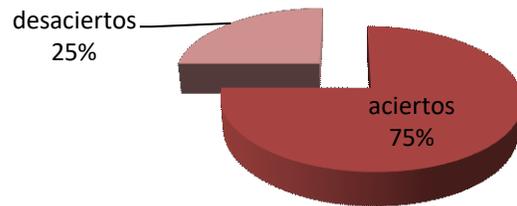
Figura 6.11: Rendimiento algoritmo detección por categorías.

Clasificación

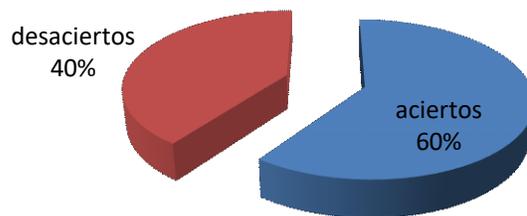
	Imágenes	Aciertos	Desaciertos	Tasa de aciertos %
Mancha Bacteriana	20	15	5	75
Tizón Temprano	20	12	8	60
Tomate Sano	20	16	4	80

Tabla 6.6: Rendimiento algoritmo de Clasificación.

Rendimiento algoritmo clasificación enfermedades: Mancha Bacteriana



Rendimiento algoritmo clasificación enfermedades: Tizón Temprano



Rendimiento algoritmo clasificación Tomate Sano

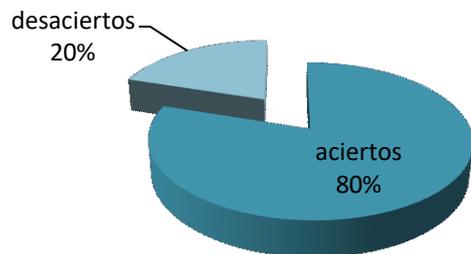


Figura 6.12: Rendimiento algoritmo clasificación por categorías.

Además de las métricas vistas anteriormente, también se utilizan otras métricas de desempeño de los sistemas de clasificación tanto en machine learning como deep learning en base a los TP (Verdaderos Positivos), FP (Falsos Positivos) y los FN (Falsos Negativos), como ser: precision, recall, y f1Score.

Precision Model: Mide la **calidad** del modelo de machine learning en tareas de clasificación.

$$Precision = \frac{TP}{TP + FP}$$

Recall (Exhaustividad): La métrica de exhaustividad nos informa sobre la **cantidad** que el modelo de machine learning es capaz de identificar.

$$Recall = \frac{TP}{TP + FN}$$

F1 Score: Se utiliza para combinar las medidas de precision y recall en un sólo valor. Compara el rendimiento combinado de la precisión y el recall entre varias soluciones.

$$F1 \text{ Score} = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

	Hoja de Tomate	No Hoja de Tomate	Ground Truth Totals
Hoja de Tomate	41	9	50
No Hoja de Tomate	4	46	50
Total	45	55	

	Precision	Recall
Hoja de Tomate	0.91	0.82
No Hoja de Tomate	0.83	0.92
Total	0.87	0.87

$$F1 \text{ Score} = \frac{2 \cdot 0.87 \cdot 0.87}{0.87 + 0.87} = 0.87 \Rightarrow 87\% \text{ Confiable}$$

Figura 6.13: Métricas de Rendimiento obtenidas para la Detección.

	Mancha Bacteriana	Tizón Temprano	Tomate Sano	Ground Truth Totals
Mancha Bacteriana	22	7	1	30
Tizón Temprano	9	20	1	30
Tomate Sano	5	0	25	30
Total	36	27	27	

	Precision	Recall
Mancha Bacteriana	0.73	0.61
Tizón Temprano	0.66	0.74
Tomate Sano	0.83	0.93
Total	0.74	0.76

$$F1 \text{ Score} = \frac{2 \cdot 0.74 \cdot 0.76}{0.74 + 0.76} = 0.75 \Rightarrow 75\% \text{ Confiable}$$

Figura 6.14: Métricas de Rendimiento obtenidas para la Clasificación.

En vista de los resultados obtenidos se concluye que se puede mejorar la precisión aumentando la cantidad de imágenes tanto para entrenamiento como para test, particularmente el caso de clasificación de Tizón Temprano, se disponía de una cantidad (800 imágenes) mucho menor en comparación con Mancha Bacteriana (1.700 imágenes) y Tomate Sano (1.272 imágenes), según se aprecia en la tabla 6.3.

6.3 Escenario de Aplicación: Prototipo Doctor Tomatto

Tomando como referencia la arquitectura propuesta en el inciso 5.1 (Arquitectura) y Figura 5.1, para realizar la detección y clasificación de enfermedades, se construye el siguiente prototipo *Doctor Tomatto* como prueba de concepto, utilizando básicamente las siguientes tecnologías:

- TensorFlow
- Keras
- Flask

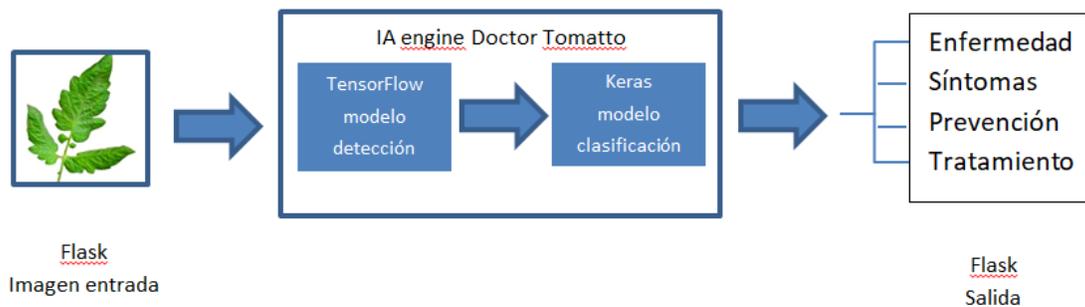


Figura 6.15: Diagrama de las tecnologías empleadas.

6.4.1 Estructura del directorio del prototipo

Los siguientes archivos y carpetas están presentes en la carpeta raíz *DoctorTomatto*:

- DoctorTomatto/:
 - modelo/:
 - modelo_clasificacion/:
 - model_VGG16.h5
 - modelo_deteccion/:
 - mask_rnn_Detecta.h5
 - static/:
 - css/: Contiene archivo de hoja de estilos
 - img/: Contiene distintas imágenes utilizadas en el portal
 - js/: Contiene archivos de java script
 - output/: Contiene imágenes de salida
 - upload/: Contiene la imagen subida como entrada
 - templates/:
 - credito.html
 - index.html
 - layout.html
 - mensaje.html
 - mensaje2.html
 - objetivo.html
 - prediccion.html
 - procesando.html
 - subirFoto.html
 - DrTomatto.ipynb: Contiene el código en python del programa principal.
 - etiquetaScore.txt: Este archivo contiene la etiqueta y el score del objeto detectado.
 - label_map_util.py: Archivo contiene funciones para manejar el mapa de etiquetas. Este código es parte del framework descargado para la detección de objetos
 - visualization_utils.py: Contiene un conjunto de funciones para la visualización de una imagen. Este código es parte del framework descargado para la detección de objetos.

6.4.2 Códigos principal y complementarios

Los Códigos principal y complementarios, se referencia al siguiente repositorio:

<https://github.com/sergioFavio/Tesis-Magister-Ing.-Software-UNLP>

Donde se encuentran los archivos en python, html, java script y css de los distintos programas empleados tanto para la detección de la hoja del tomate como para la clasificación de enfermedades del tomate que se presentan en la hoja.

Capítulo 7: Trabajos Relacionados

A continuación se describen algunos trabajos que se han desarrollado y que guardan relación con este trabajo de tesis.

7.1 Papers consultados

En [1], los autores describen como hacer la detección de plagas y enfermedades en la hoja de la planta de tomate utilizando una plataforma de robot (no implementada todavía) a partir de las imágenes tomadas con cámaras. En este caso, existe la necesidad de una computadora móvil y una cámara estándar RGB para la detección de enfermedades mediante un robot. La detección de la enfermedad se realiza en el robot, hace mención que la computadora móvil debe estar lo suficientemente potente como para realizar tareas computacionales, además de tareas adicionales como la navegación autónoma, conducción de motores, procesamiento de imágenes. Indicando que código de detección debe ser ligero en términos de coste de cálculo y tamaño de la RAM.

Utiliza el dataset de PlantVillage para entrenar el modelo con el framework Caffe de deep learning y en cuanto al hardware emplea la Nvidia Jetson Tx1, que tiene 256 núcleos CUDA, procesador ARM de cuatro núcleos, 4GB RAM, 16 GB eMMC y otros periféricos.

En [2], los autores explican el uso de un método que aplica la técnica de transformada de ondas de gabor para extraer características relevantes relacionadas con la imagen de la hoja de tomate utilizando Support Vector Machines (SVM) con un kernel de funciones alternas del núcleo para detectar e identificar el tipo de enfermedad que infecta la planta de tomate, empleando la técnica de características basadas en ondículas para identificar un subconjunto de características óptimas y finalmente para detectar y clasificar una máquina de vectores de soporte con diferentes funciones de clases del kernel, incluido el kernel de Cauchy, Invmult Kernel y Laplacian Kernel que se emplearon para evaluar la capacidad para detectar e identificar dónde la hoja de tomate está infectada con mildiú polvoroso o tizón temprano. El conjunto de datos consistió en 100 imágenes para cada tipo de enfermedades de tomate.

En [3], el autor describe el problema asociado con la identificación automática de enfermedades de las plantas usando rango visible las imágenes que han recibido una atención considerable en las últimas dos décadas, sin embargo, las técnicas propuestas hasta ahora suelen tener un alcance limitado y dependen de las condiciones ideales de captura para funcionar correctamente. La aparente falta de avances significativos puede explicarse parcialmente por algunos desafíos difíciles que plantea el tema: presencia de fondos complejos que no se pueden separar fácilmente de la región de interés (generalmente hoja y tallo), los límites de los síntomas a menudo no están bien definidos, las condiciones no controladas de captura pueden presentar características que dificultan el análisis de la imagen, ciertas enfermedades producen síntomas con una amplia gama de características, los síntomas producidos por diferentes enfermedades pueden ser muy similares y pueden estar presentes simultáneamente. Este artículo proporciona un análisis de cada uno de esos desafíos, enfatizando tanto los problemas que pueden causar como también pueden afectar potencialmente las técnicas propuestas

en el pasado. Algunas posibles soluciones capaces de superar en al menos se proponen algunos de esos desafíos.

En [4] el autor describe varios desarrollos de modelos de redes neuronales convolucionales para realizar la detección de enfermedades de las plantas y diagnóstico mediante imágenes de hojas simples de plantas sanas y enfermas, mediante metodologías de aprendizaje profundo. El entrenamiento de los modelos se realizó con el uso de una base de datos abierta de 87,848 imágenes, que contiene 25 plantas diferentes en un conjunto de 58 clases distintas de combinaciones [planta, enfermedad], incluidas plantas sanas. Se entrenaron varios modelos de arquitecturas con el mejor desempeño alcanzando una tasa de éxito del 99.53% en la identificación de combinación correspondiente [planta, enfermedad] (o planta sana). La tasa de éxito significativamente alta, hace que modelar una herramienta de advertencia o alerta temprana sea muy útil, y un enfoque que podría ampliarse aún más para apoyar un sistema integrado de identificación de enfermedades de las plantas para operar en condiciones reales de cultivo.

7.2 Aplicaciones relacionadas

En cuanto a las aplicaciones relacionadas con el presente trabajo podemos citar las siguientes:

- PlantNet – Identificación de plantas.
- PlantSnap – Identificación de plantas, flores, cactus y hongos.
- Doctor Leaf – Diagnostica la presencia de enfermedades en las plantas.
- Plantix – Diagnostica los cultivos infectados y ofrece tratamiento para cualquier plaga, enfermedad o deficiencia de nutrientes.

PlantNet.- Es una aplicación para la recopilación, anotación y recuperación de imágenes para ayudar en la identificación de plantas. Fue desarrollada por un consorcio formado por científicos de CIRAD, INRA, INRIA, IRD, y la red Tela Botánica en virtud de un proyecto financiado por la Fundación Agropolis.

Incluye un sistema de apoyo para la identificación automática de las plantas a partir de imágenes en comparación con las imágenes de una base de datos botánica. Los resultados se utilizan para obtener el nombre botánico de una planta, si esta es suficientemente ilustrada en la base. El nombre de especies procesadas y el nombre de imágenes utilizados evolucionan con las contribuciones al proyecto.

La aplicación no permite la identificación de plantas ornamentales. Funciona mejor cuando las fotos presentadas se centran en un órgano o una parte precisa de la planta. Fotos de hojas de árboles con un fondo uniforme presentarán así los resultados más relevantes. Funciona es distintos tipos de dispositivos, smart phones, tablets y computadores, sitio web <http://www.plantnet.org>.

PlantSnap.- Es una aplicación que identifica plantas, flores, cactus y hongos a través del uso de un dispositivo móvil sea iOS o android, a partir de la captura de una imagen con el dispositivo móvil o cargar una imagen ya guardada a la aplicación, sitio web <http://www.plantsnap.com>.

Leaf Doctor.- Es una aplicación gratuita (iPhone o android), para patometría, realiza la evaluación cuantitativa de la intensidad de las enfermedades de las plantas. Esta aplicación interactiva permite a los usuarios recolectar o enviar fotografías de órganos de plantas sintomáticos (por ejemplo, hojas) y medir el porcentaje de área de tejido enferma. Los datos de evaluación y fotografías podrán ser enviados por correo electrónico a cualquier destinatario.

Plantix.- Es un aplicación que diagnostica los cultivos infectados y ofrece tratamiento para cualquier plaga, enfermedad o deficiencia de nutrientes en treinta tipos de cultivos de plantas a través del uso de un dispositivo móvil Android, a partir de la captura de una imagen con el dispositivo móvil o cargar una imagen ya guardada a la aplicación, sitio web <https://www.plantix.net>.

Aplicación	Detecta	Clasifica (Planta/Enfermedad: ManchaBacteriana (MB),Tizón Temprano(TT), Hoja Sana(HS))	Cultivo (Tomate/Otro)	Síntomas y Tratamiento	*AporteAcadémico (Reproducibilidad)
PlantNet	Objeto	planta	Tomate/Otro	No	No
PlantSnap	Objeto	planta	Otro	No	No
DoctorTomatto	Hoja	Enferm: MB,TT,HS	Tomate	**Sí	Sí

Tabla 7.1: Comparación de las funcionalidades que prestan PlantNet y PlantSnap versus Doctor Tomatto.

* Métodos que usa están documentados y disponibles que permite reproducir el Proceso y los resultados alcanzados.

** Con carácter ilustrativo, para mostrar potencialidad de la aplicación, ya que no es foco de la investigación de la tesis.

Aplicación	Detecta	Clasifica (Planta/Enfermedad: ManchaBacteriana (MB),Tizón Temprano(TT), Hoja Sana(HS))	Cultivo (Tomate/Otro)	Síntomas y Tratamiento	*AporteAcadémico (Reproducibilidad)
Leaf Doctor	Hoja (no muy bien)	Enferm: MB,TT,HS (no muy bien)	No especifica	No	No
Plantix	Hoja	Enferm: MB,TT,HS (no muy bien)	Tomate/Otro	Sí	No
DoctorTomatto	Hoja	Enferm: MB,TT,HS	Tomate	**Sí	Sí

Tabla 7.2: Comparación de las funcionalidades que prestan Leaf Doctor y Plantix versus Doctor Tomatto.

* Métodos que usa están documentados y disponibles que permite reproducir el Proceso y los resultados alcanzados.

** Con carácter ilustrativo, para mostrar potencialidad de la aplicación, ya que no es foco de la investigación de la tesis.

Concluyendo, se empleó como base, el uso de redes neuronales convolucionales [4] para desarrollar una solución más factible al problema planteado. En cuanto a las aplicaciones relacionadas (PlantNet y PlantSnap), sólo realizan clasificación de plantas y no así la detección y clasificación de enfermedades, cosa que caracteriza a Doctor Tomatto. Con respecto a las dos aplicaciones restantes (Leaf Doctor y Plantix), éstas si hacen detección y clasificación de enfermedades, aunque de una manera no muy exitosa con relación a Doctor Tomatto, para las enfermedades contempladas en el presente trabajo.

Por último, destacar que la implementación del prototipo de Doctor Tomatto, está documentada y disponible, de tal manera que se pueda reproducir el proceso y los resultados alcanzados en comparación a las anteriores aplicaciones mencionadas que no poseen esta cualidad.

Capítulo 8: Conclusiones

8.1 Conclusiones

Tanto la ingeniería de software y el aprendizaje profundo resuelven problemas comerciales. Ambos interactúan con bases de datos, analizan y codifican módulos y generar resultados que son utilizados por la empresa. La comprensión del dominio empresarial es imperativo para ambos campos y también lo es la usabilidad. Sobre estos parámetros, tanto la ingeniería de software como el aprendizaje automático son similares. Sin embargo, la diferencia clave radica en la ejecución y el enfoque utilizado para resolver el desafío de negocio.

La creación de software implica escribir código necesario que se puede ejecutar por el procesador, es decir, la computadora. Por otro lado ML recopila datos históricos y comprende las tendencias en los datos. Basado en las tendencias, el algoritmo ML predecirá la salida deseada.

El desarrollo de una solución de aprendizaje automático suele ser más iterativo que la ingeniería de software. Además, no es tan exacto como el desarrollo de software, ya que el uso de machine learning es más exploratorio en comparación con la ingeniería de software (IS) que no involucra este tipo de soluciones, ya que la IS utiliza una programación imperativa. Pero ML es seguro que es una buena solución generalizada. Es una solución válida para complejos problemas comerciales y a menudo, la única solución para problemas que los humanos somos incapaces de comprender. Aquí ML juega un papel fundamental. Su belleza radica en el hecho de que si los datos de entrenamiento cambian, no es necesario iniciar el proceso de desarrollo desde cero. El modelo puede ser reentrenado y estar listo.

Los aportes de la tesis están basados en dos pilares; la primera de ella en la contribución en el desarrollo de una metodología de trabajo para la detección y clasificación de enfermedades mediante el procesamiento de imágenes digitales que contempla: Adquisición de Imágenes; Pre Procesamiento; Detección de Objetos y Clasificación y Recursos Necesarios.

La segunda mediante la automatización en tiempo real de los procesos de detección, clasificación de enfermedades, advirtiendo sobre una posible amenaza para el cultivo y su posterior tratamiento mediante el procesamiento de imágenes digitales se logra reducir el esfuerzo requerido para el agricultor productor de tomates, ahorrándose el trabajo de desplazarse geográficamente y el tener que ubicar un profesional agrónomo para ese cometido, especialmente cuando el cultivo puede estar en un lugar remoto, además del diagnóstico de la enfermedad prescribir el uso de pesticidas apropiados a ser utilizados por los agricultores, limitando drásticamente la adquisición descontrolada plaguicidas que conducen a un uso excesivo e inadecuado con efectos consiguientemente negativos para el medio ambiente.

Respecto a los resultados obtenidos, se concluye que se puede mejorar la precisión aumentando la cantidad de imágenes tanto para entrenamiento como para test, particularmente el caso de clasificación de Tizón Temprano, se disponía de una cantidad (800 imágenes) mucho menor en comparación con Mancha Bacteriana (1.700 imágenes) y Tomate Sano (1.272 imágenes). Es importante indicar que al utilizar modelos pre entrenados (transfer learning) tanto para la detección como la clasificación versus

modelos desde cero (from scratch) la cantidad de parámetros que se utilizan en modelos pre entrenados es mucho menor en comparación a los modelos desde cero, en el caso de clasificación (<https://github.com/sergioFavio/Tesis-Magister-Ing.-Software-UNLP/blob/master/codigosDetectacClasifica/ClasificarImagenesVGG16.ipynb>).

Del total de 134.272.835 sólo se entrenaron 12.291 parámetros (134.260.544 parámetros que no se entrenaron), lo cual permite hacer un mejor uso (ahorro) en términos de recursos de cómputo como memoria y procesador, además de reflejar que la cantidad de tiempo empleado para entrenar el modelo es bastante inferior que si se hubiera realizado con un modelo entrenado desde cero.

8.2 Desarrollos Futuros

A partir del trabajo realizado se puede añadir otras enfermedades como también incorporar la detección y clasificación de plagas que se presentan tanto en la planta como en el fruto del tomate. También se puede escalar a otros tipos de hortalizas y frutos en base a la metodología de trabajo que se desarrolló para la detección y clasificación de enfermedades y plagas con su respectivo tratamiento.

Bibliografía

- [1] Durmuş, Halil, Ece Olcay Güneş, and Mürvet Kırıcı. “Disease Detection on the Leaves of the Tomato Plants by Using Deep Learning.” In 2017 6th International Conference on Agro-Geoinformatics, 1–5, 2017. <https://doi.org/10.1109/Agro-Geoinformatics.2017.8047016>.
- [2] Mokhtar, Usama, Mona A. S. Ali, Aboul Ella Hassenian, and Hesham Hefny. “Tomato Leaves Diseases Detection Approach Based on Support Vector Machines.” In 2015 11th International Computer Engineering Conference (ICENCO), 246–50, 2015. <https://doi.org/10.1109/ICENCO.2015.7416356>.
- [3] Barbedo, Jayme Garcia Arnal. “A Review on the Main Challenges in Automatic Plant Disease Identification Based on Visible Range Images.” *Biosystems Engineering* 144 (April 1, 2016): 52–60. <https://doi.org/10.1016/j.biosystemseng.2016.01.017>.
- [4] Ferentinos, Konstantinos P. “Deep Learning Models for Plant Disease Detection and Diagnosis.” *Computers and Electronics in Agriculture* 145 (February 1, 2018): 311–18. <https://doi.org/10.1016/j.compag.2018.01.009>.
- [5] De la Escalera Arturo, (2001) *Visión por Computador*. Prentice Hall. España.
- [6] Buduma Nikhil, (2017) *Fundamentals of Deep Learning DESIGNING NEXT-GENERATION MACHINE INTELLIGENCE ALGORITHMS*. O'REILLY. United State of America.
- [7] Chollet François, (2018) *Deep Learning with Python*. MANNING Publishing. United States.
- [8] Ashwin Pajankar, (2015) *Raspberry Pi Computer Vision Programming*. Packt Publishing. United Kingdom.
- [9] Coelho Luis Pedro, Richert Willi, (2015) *Building Machine Learning systems with Python*. Packt Publishing. United Kingdom.
- [10] Dey Sandipan, (2018) *Hands-On Image Processing with Python*. Packt Publishing. United Kingdom.
- [11] Kapur Saurabh, (2017) *Computer Vision with Python 3*. Packt Publishing. United Kingdom.
- [12] Barelli Felipe, (2018) *Introdução à Visão Computacional*. Casa do Código. Brasil.
- [13] Dadhich Abhinav, (2018) *Practical Computer Vision*. Packt Publishing. United Kingdom.
- [14] Planche Benjamin, Andres Elliot, (2019) *Hands-On Computer Vision with TensorFlow 2*. Packt Publishing. United Kingdom.

- [15] Arpteg, Anders, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. “Software Engineering Challenges of Deep Learning.” In 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 50–59, 2018. <https://doi.org/10.1109/SEAA.2018.00018>.
- [16] Anirudh Koul, Siddha Ganju, Meher Kasam, (2019) *Practical Deep Learning for Cloud, Mobile & Edge*. O’Reilly. United States of America.
- [17] V Kishore Ayyadevara, (2019) *Neural Networks with Keras Cookbook*. Packt Publishing. United Kingdom.
- [18] Armando Vieira, Bernardete Ribeiro , (2018) *Introduction to Deep Learning Business Applications for Developers*. Packt. New York, United States.
- [19] Mohamed Elgendy, (2020) *Deep Learning for Vision Systems*. MANNING Publishing. United States.
- [20] Rajalingappaa shanmugamani, (2018) *Deep Learning for Computer Vision*. Packt Publishing. United Kingdom.
- [21] Adrian Rosebrock, (2017) *Deep Learning for Computer Vision with Python*. PyImageSearch. United States.
- [22] Gianluca Mauro & Nicolo Valigi, (2020) *Zero to AI A nontechnical hype-free guide to prospering in the AI era*. MANNING Publishing. United States.
- [23] Veljko Kronic, (2019) *Succeeding with AI*. MANNING Publishing. United States.
- [24] Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.” In Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1, 91–99. NIPS’15. Cambridge, MA, USA: MIT Press, 2015.
- [25] Emamanuel Raj, (2021) *Engineering MLOps*. Packt Publishing. United Kingdom.
- [26] Amershi, Saleema, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. “Software Engineering for Machine Learning: A Case Study.” In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 291–300, 2019. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>.
- [27] Khomh, Foutse, Bram Adams, Jinghui Cheng, Marios Fokaefs, and Giuliano Antoniol. “Software Engineering for Machine-Learning Applications: The Road Ahead.” IEEE Software 35, no. 5 (September 2018): 81–84. <https://doi.org/10.1109/MS.2018.3571224>.

Anexo A – Tutorial Doctor Tomatto

Página principal, contempla las siguientes opciones en su menú:

- Inicio – Página principal.
- Objetivo – Describe el objetivo de la investigación de la tesis.
- Analizar – Es la función principal de la aplicación, que permite subir y analizar la imagen para hacer la detección, clasificación de la enfermedad y recomendaciones a seguir para su tratamiento.
- Crédito – Proporciona información del autor de la tesis.



Figura 1: Página inicial de la aplicación.

Luego se selecciona la opción de *Analizar imagen*, que nos permite seleccionar una imagen para que sea como se ve a continuación.

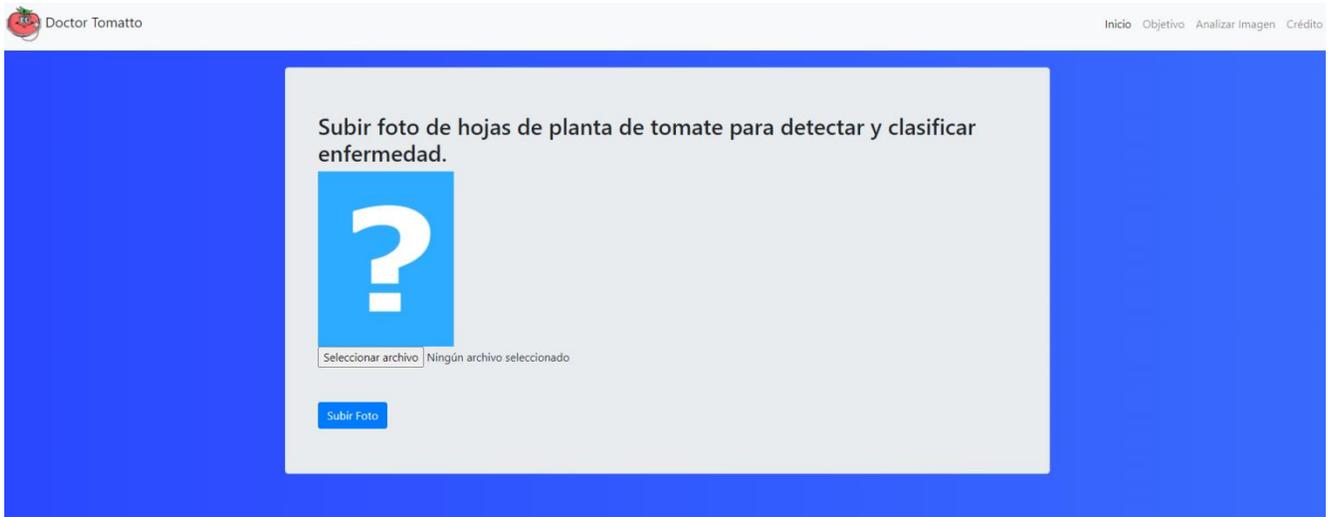


Figura 2: Interfaz que permite seleccionar la imagen.

A continuación se procede a seleccionar la imagen que será sometida a procesamiento activando el botón *Seleccionar archivo*.

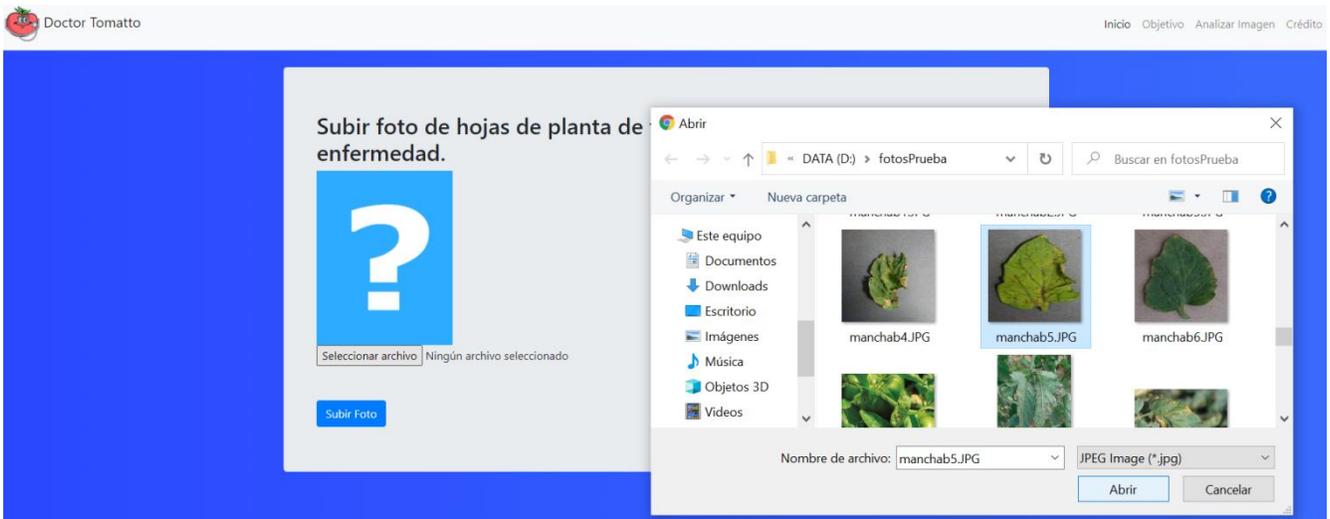


Figura 3: Interfaz donde se selecciona la foto a procesar.



Figura 4: Foto seleccionada y activado el botón *Subir foto*.

Una vez terminado el proceso de detección y clasificación, la aplicación arroja el siguiente resultado, donde se da el diagnóstico (predicción) seguido de un carrusel de tres imágenes (lado derecho) donde se muestran la imagen subida, detección de la hoja y área de la hoja detectada, en cada una de estas imágenes se puede hacer un zoom (ampliar la imagen) haciendo un click en el centro de la foto para tener una mejor apreciación. En el lado inferior izquierdo se presentan tres botones que se pueden activar, el primero de ellos describe los *síntomas* de la enfermedad, el segundo que es *prevención* muestra cómo se puede prevenir y el tercero indica el *tratamiento* a seguir si la enfermedad está totalmente manifestada en la plantación.

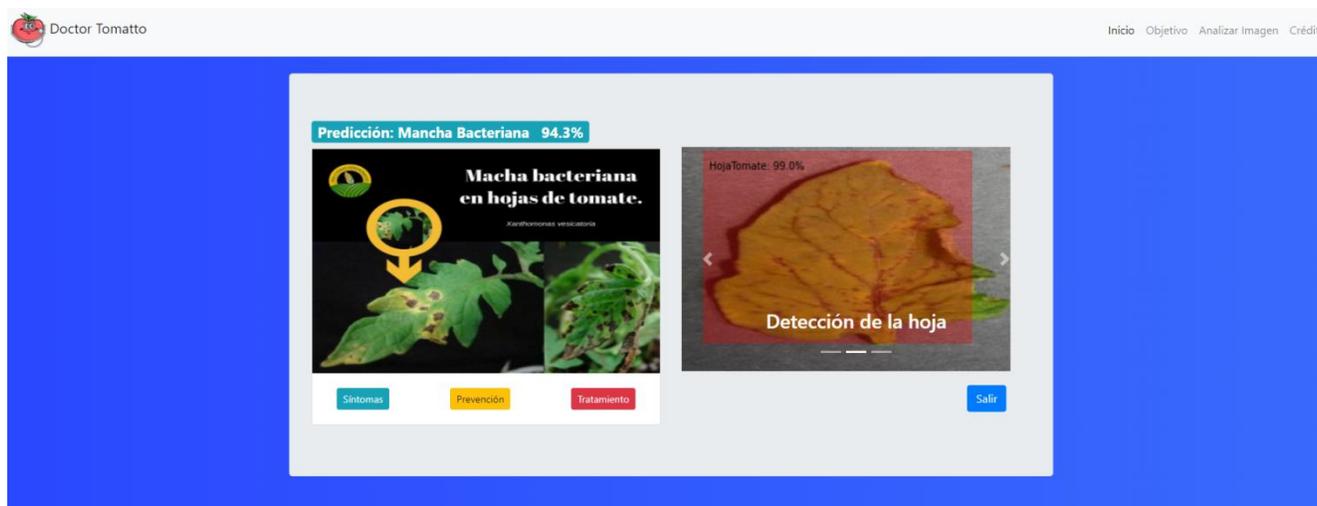


Figura 5: Resultado después de la detección y clasificación.



Figura 6: Zoom de imagen *Detección de la hoja* seleccionada del carrusel en lado derecho.

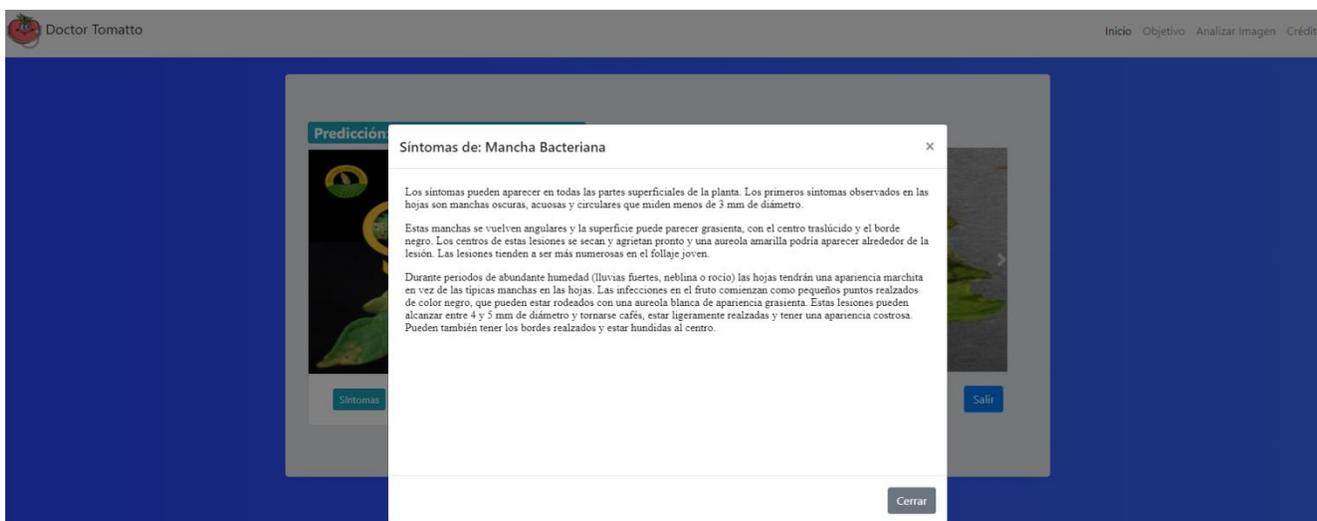


Figura 7: Descripción de los síntomas de la enfermedad detectada.

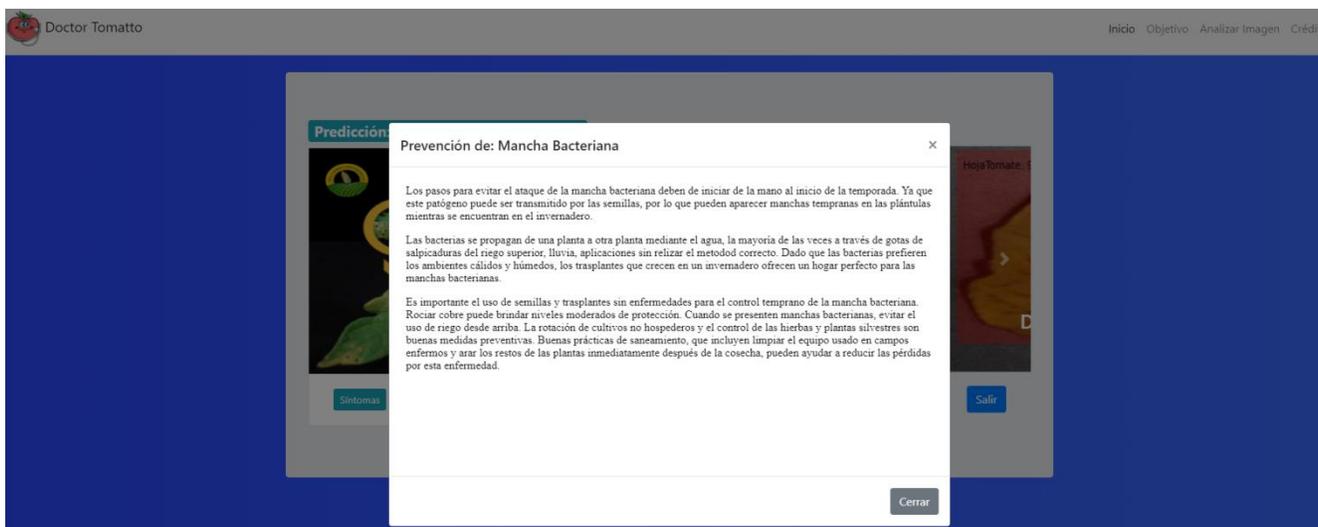


Figura 8: Descripción de cómo hacer la prevención de la enfermedad detectada.

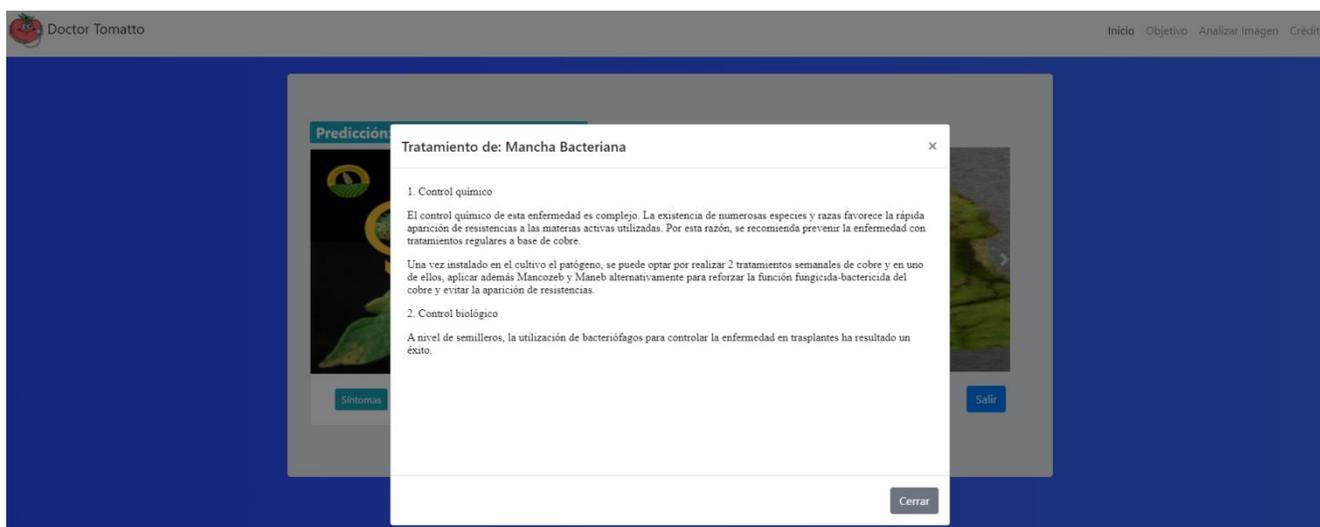


Figura 9: Descripción de cómo hacer el tratamiento de la enfermedad detectada.

La información proporcionada por el prototipo Doctor Tomatto en las figuras 7, 8 y 9 (Síntomas, Prevencción y Tratamiento), no son foco de estudio del presente trabajo, sino más bien mostrar la potencialidad del trabajo desarrollado como un complemento.

Cuando la imagen seleccionada y subida no es una hoja de tomate, como se ve a continuación.



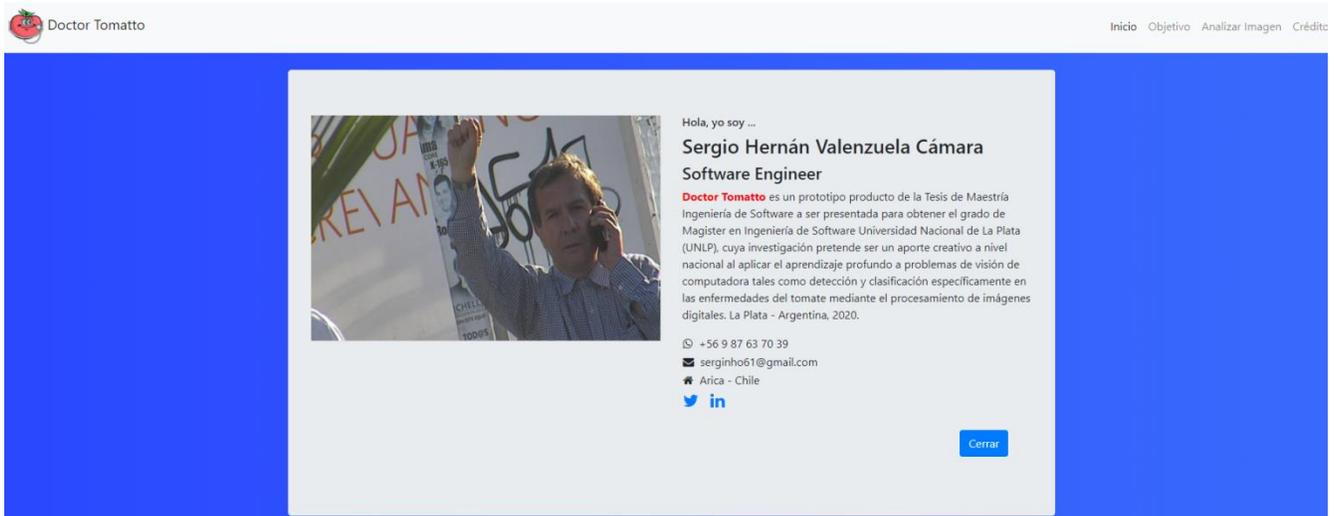
Figura 10: Imagen seleccionada y subida, que no corresponde a una hoja de tomate.

Luego de seleccionar y subir la imagen y terminado el proceso de detección, se muestra un mensaje de error indicando que la imagen no es de una hoja de tomate o tiene probabilidad inferior al umbral definido como hoja de tomate y se pide otra imagen.



Figura 11: Mensaje de error cuando la imagen subida no es de una hoja de tomate.

Por último la pestaña de *crédito* (menú principal) que describe la información asociada a ésta.



Doctor Tomatto

Inicio Objetivo Analizar Imagen Crédito

Hola, yo soy ...

Sergio Hernán Valenzuela Cámara
Software Engineer

Doctor Tomatto es un prototipo producto de la Tesis de Maestría Ingeniería de Software a ser presentada para obtener el grado de Magister en Ingeniería de Software Universidad Nacional de La Plata (UNLP), cuya investigación pretende ser un aporte creativo a nivel nacional al aplicar el aprendizaje profundo a problemas de visión de computadora tales como detección y clasificación específicamente en las enfermedades del tomate mediante el procesamiento de imágenes digitales. La Plata - Argentina, 2020.

+56 9 87 63 70 39
serginho61@gmail.com
Arica - Chile

[Twitter](#) [LinkedIn](#)

Cerrar

Figura 12: Información referente a la opción de *Crédito* menú principal.

Anexo B – Código para la detección de hoja

A continuación se lista el código para la realización de la detección:

```
# librerías utilizadas
import numpy
import scipy
import PIL
import cython
import matplotlib
import skimage
import tensorflow
import keras
import cv2
import h5py

from os import listdir
from xml.etree import ElementTree
from numpy import zeros
from numpy import asarray
from mrcnn.utils import Dataset
from mrcnn.config import Config
from mrcnn.model import MaskRCNN

class DatasetHoja(Dataset):
    def cargar_dataset(self, direccion_dataset):
        self.add_class("dataset", 1, "HojaTomate") # adicionar clase (etiqueta)
        self.add_class("dataset", 2, "NoHojaTomate") # adicionar clase (etiqueta)

        # directorios donde estan las imagenes e las anotaciones:
        direccion_imagenes = direccion_dataset
        direccion_anotaciones = direccion_dataset
        arch_aux='' # ... guarda temporalmente el nombre del archivo que esta en la lista listdir ...

        for nombre_imagen in listdir(direccion_imagenes):
            arch_aux=nombre_imagen
            if arch_aux.endswith('.xml'): # salta al siguiente loop si el archivo es.xml
                continue # cuando el if es verdadero, no continua, retornando para el próximo loop
```

```

        elif arch_aux.endswith('.jpg'): # salta al siguiente loop si el archiv
o es.xml
            imagen_id = nombre_imagen[:-4]
            directorio_completo_imagen = direccion_imagenes + nombre_imagen
            directorio_completo_annotaciones = direccion_annotaciones + imagen_id
            + '.xml'
            self.add_image('dataset', image_id=imagen_id, path=directorio_compl
eto_imagen, annotation=directorio_completo_annotaciones)

        else: # salta al siguiente loop si el archivo no es.xml ni .jpg
            continue # cuando el if es verdadero, no continua, retornando para
el próximo loop

def extraer_cajas(self, direccion_archivo):
    archivo = ElementTree.parse(direccion_archivo)
    raiz = archivo.getroot()
    cajas = []
    for caja in raiz.findall('.//bndbox'):
        xmin = int(caja.find('xmin').text)
        ymin = int(caja.find('ymin').text)
        xmax = int(caja.find('xmax').text)
        ymax = int(caja.find('ymax').text)
        coordenadas = [xmin, ymin, xmax, ymax]
        cajas.append(coordenadas)
    largura = int(raiz.find('.//size/width').text)
    altura = int(raiz.find('.//size/height').text)
    clase = str(raiz.find('.//object/name').text) # clase del objeto (HojaT
omate o NoHojaTomate)
    return cajas, largura, altura, clase

def load_mask(self, imagen_id):
    informaciones_imagen = self.image_info[imagen_id]
    directorio_annotacion = informaciones_imagen['annotation']
    cajas, l, a, clase = self.extraer_cajas(directorio_annotacion)
    mascararas = zeros([a, l, len(cajas)], dtype='uint8')
    classes_ids = []
    for i in range(len(cajas)):
        caja = cajas[i]
        x_inicio, x_final = caja[1], caja[3]
        y_inicio, y_final = caja[0], caja[2]
        mascararas[x_inicio:x_final, y_inicio:y_final, i] = 1
        if clase == 'HojaTomate':
            classes_ids.append(self.class_names.index('HojaTomate'))

```

```

        if clase == 'NoHojaTomate':
            classes_ids.append(self.class_names.index('NoHojaTomate'))

    return mascarar, asarray(classes_ids, dtype='int32')

# Cuando se invoca el modelo, tenemos que pasar las configuraciones en la forma de una clase:
class ConfiguracionesHojas(Config):
    NAME = "configuraciones_hojas" # Dar a la configuración un nombre reconocible
    NUM_CLASSES = 1 + 2 # define el número de clases (background + HojaTomate + NoHojaTomate)

    # Todas muestras de imágenes de entrenamiento son 227x227 y 512x512
    IMAGE_MIN_DIM = 227
    IMAGE_MAX_DIM = 512

    STEPS_PER_EPOCH = 500 # número de pasos por época
    GPU_COUNT = 1 # cuantas GPUs son utilizadas
    IMAGES_PER_GPU = 2 # cuantas imagenes son pasadas para la GPU cada vez

    DETECTION_MAX_INSTANCES = 2 # número máximo de detección por imagen

    RPN_ANCHOR_SCALES = (8, 16, 32, 64, 128) # Longitud del lado del ancla cuadrada en píxeles

    # Número de ROI por imagen para alimentar a los cabezales de clasificador / enmascaramiento
    # El papel Mask RCNN usa 512 pero a menudo el RPN no genera suficientes propuestas positivas para llenar esto y mantener un positivo: negativo
    # proporción de 1: 3. Puede aumentar el número de propuestas ajustando el umbral de RPN NMS.
    TRAIN_ROIS_PER_IMAGE = 32

    MAX_GT_INSTANCES = 50 # Número máximo de instancias de verdad del terreno para usar en una imagen

    # ROIs mantenidos después de la supresión no máxima (entrenamiento e inferencia)
    POST_NMS_ROIS_INFERENCE = 500
    POST_NMS_ROIS_TRAINING = 1000

# Creando dataset de entrenamiento:

```

```

dataset_img_entrenamiento = DatasetHoja()
dataset_img_entrenamiento.cargar_dataset('/content/drive/MyDrive/ColabDatasetHoja/img_entrenamiento/')
dataset_img_entrenamiento.prepare()
print('Tamaño img_entrenamiento: %d' % len(dataset_img_entrenamiento.image_ids))

# Creando dataset de test:
dataset_img_test = DatasetHoja()
dataset_img_test.cargar_dataset('/content/drive/MyDrive/ColabDatasetHoja/img_test/')
dataset_img_test.prepare()
print('Tamaño img_test: %d' % len(dataset_img_test.image_ids))

# Preparando las configuraciones:
config = ConfiguracionesHojas()

# Creando el modelo:
modelo = MaskRCNN(mode='training', model_dir='/content/drive/MyDrive/ColabDatasetHoja/logs/', config=config)
modelo.load_weights('/content/drive/MyDrive/ColabDatasetHoja/mask_rcnn_coco.h5', by_name=True, exclude=["mrcnn_class_logits", "mrcnn_bbox_fc", "mrcnn_bbox", "mrcnn_mask"])
modelo.train(dataset_img_entrenamiento, dataset_img_test, learning_rate=0.001, epochs=50, layers='heads')

```