



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Trabajo Terminal II

“Prototipo de aplicación para la detección de deficiencia
de nutrientes en cultivos de hidroponía”

2017-A054



Presenta:

Edgar Rodrigo Arredondo Basurto

Directores:

Ing. Eduardo Gutiérrez Aldana

Dr. José Félix Serrano Talamantes



Junio de 2018



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
SUBDIRECCIÓN ACADÉMICA



No de TT:2017-A054

1 de junio de 2018

Documento técnico

“Prototipo de aplicación para la detección de deficiencia de nutrientes en cultivos de hidroponía”

2017-A054

Presenta:

Edgar Rodrigo Arredondo Basurto¹

Directores:

Ing. Eduardo Gutiérrez Aldana

Dr. José Félix Serrano Talamantes

RESUMEN

El objetivo de este proyecto es desarrollar un prototipo de aplicación de visión por computadora que tome como entrada imágenes de hojas de cultivos con una enfermedad visible y como resultado se obtiene un probable diagnóstico de la enfermedad de la planta, bajo ciertas restricciones de plantas y enfermedades. Es así que en este documento se describen el análisis, diseño y construcción del sistema que cumple con dicho objetivo.

Palabras clave: Clasificador, descriptor, aprendizaje automático, aprendizaje profundo, red neuronal artificial.

¹edgar_arredondo@outlook.com

I. Resumen

El objetivo de este proyecto es desarrollar un prototipo de aplicación de visión por computadora que analice imágenes de hojas de tomate que presenten síntomas visibles de alguna enfermedad. Como resultado se obtiene un probable diagnóstico de cuál es la enfermedad que presenta la planta, sobre un subconjunto definido de posibles enfermedades.

Es así como en este documento se definen y diseñan las diferentes etapas que requiere este prototipo para su implementación, así como detalles y evidencias del desarrollo y puesta a punto de este.

En el capítulo 1 se presenta una breve introducción en la que se explica el contexto de este proyecto y la motivación que dio origen a su desarrollo. Se definen el objetivo general y los objetivos particulares de este proyecto y finalmente se presenta la metodología de desarrollo propuesta, que consiste en una metodología incremental.

En el capítulo 2 se exponen los fundamentos teóricos sobre los que se desarrolla la propuesta solución planteada, que consiste en una Red Neuronal Convolutacional (CNN por sus siglas en inglés). Se incluye una breve introducción a los conceptos del aprendizaje profundo (*deep learning*) y redes neuronales, haciendo mención especial de las redes CNN.

En los capítulos 3, 4 y 5 se aborda el análisis, diseño y construcción, respectivamente, del primer incremento propuesto en la metodología, que consiste en el desarrollo del clasificador. Dicho clasificador será entrenado usando el framework Caffe, especializado en aprendizaje profundo. Una vez entrenado el clasificador podrán realizarse predicciones con él usando el módulo de aprendizaje profundo (dnn) de OpenCV.

En los capítulos 6, 7, 8 y 9 se describen el análisis, diseño, implementación e implantación del sistema Web. Este sistema permitirá a los usuarios identificar la enfermedad que presenta la hoja de su imagen a través de una interfaz Web.

¹edgar_arredondo@outlook.com

En el capítulo 10 se describen las pruebas realizadas en el sistema y se discuten los resultados obtenidos tomando como referencia los objetivos de este proyecto.

En el Anexo A se presenta el código fuente de los principales componentes del sistema Web. En el Anexo B se especifica el manual de usuario del sistema Web, en el que se describe la funcionalidad a disposición del usuario, las características ideales de la imagen que debe suministrarse y la interpretación de resultados. Finalmente, en el Anexo C se describen los síntomas de las nueve enfermedades del tomate que el sistema es capaz de identificar.

II. Abstract

The objective of this project is the developing of a computer vision application that analyze images of tomato plant leaves with visible symptoms of a disease. Then, the application gives a probably diagnostic of the plant disease, on a defined subset of possible diseases.

In this document are defined and designed the different phases that this prototype needs for its implementation, as well as details and evidences of its development and implementation.

In the chapter 1 is presented a brief introduction where is explained the context of this project and the motivation behind its development. The general objective and the particular objectives of this project are defined and finally, the software development methodology is presented, an incremental methodology.

In the chapter 2 is discussed the theoretical fundamentals behind the proposed solution, consisting of a Convolutional Neural Network (CNN). It includes a brief introduction to deep learning concepts, with special mention of CNN networks.

In chapters 3, 4 and 5 are discussed the analysis, design and construction of the first increment proposed in the methodology, respectively. This first increment is the development of the image classifier. The classifier is trained with Caffe framework, a framework specialized in deep learning. Once the classifier is trained, predictions can be made with using the OpenCV deep learning module (dnn).

In chapters 6, 7, 8 and 9 are described the analysis, design, implementation and implantation of the Web system. This system allows users to identify the disease of the plant on their images, through a Web interface.

In chapter 10 are described the tests to the classifier and the Web system. The results are compared against the objectives of this project.

In the Appendix A is located the source code of the main software components of the Web system. In the Appendix B is specified the user manual for the Web

system, which describes the functionality available to the user, the ideal characteristics of the supplied image and the interpretation of results. Finally, in Appendix C, are described the nine tomato diseases that the system is able to identify.

III. Índice general

I. Resumen.....	3
II. Abstract.....	5
III. Índice general	7
Capítulo 1. Introducción	10
1.1 Planteamiento del problema.....	10
1.2 Objetivos.....	12
1.2.1 Objetivo general.....	12
1.2.2 Objetivos particulares.....	12
1.3 Justificación.....	12
1.4 Metodología.....	13
Capítulo 2. Marco teórico	15
2.1 Comparación de clasificación de imágenes usando modelos de aprendizaje automático y aprendizaje profundo.....	15
2.2 Redes Neuronales Artificiales (ANN).....	17
2.2.1 Redes neuronales unidireccionales.....	18
2.2.2 Funciones de activación.....	18
2.2.3 Entrenamiento de una red neuronal artificial.....	19
2.3 Redes Neuronales Convolucionales (CNN).....	19
2.3.1 Capa convolucional.....	20
2.3.2 Capa de agrupamiento.....	21
2.4 Arquitectura de las redes neuronales convolucionales.....	21
Capítulo 3. Análisis del clasificador.....	23
3.1 Requisitos del sistema.....	23
3.1.1 Requisitos de software.....	23
3.2 Modelo de casos de uso.....	25
3.3 Interfaz gráfica.....	25

Capítulo 4. Diseño del clasificador.....	26
4.1 Calibración fina.....	26
4.2 Caffe. Framework de aprendizaje profundo.....	27
Capítulo 5. Construcción del clasificador.....	29
5.1 Instalación de Caffe.....	29
5.1.1 Instalación de OpenCV.....	29
5.1.2 Instalación de Caffe.....	31
5.2 Calibración fina de las redes GoogLeNet y CaffeNet con Caffe.....	33
5.3 Realización de predicciones con la CaffeNet, Python y OpenCV.....	49
Capítulo 6. Análisis del sistema Web.....	55
6.1 Requisitos del sistema.....	55
6.1.1 Requisitos de software.....	55
6.2 Modelo de casos de uso.....	56
6.2.1 CU1. Seleccionar imagen.....	57
6.2.1 CU2. Obtener diagnóstico.....	58
6.3 Interfaz gráfica.....	59
6.3.1 IU1.0 Pantalla de inicio.....	59
Capítulo 7. Diseño del sistema Web.....	61
7.1 Arquitectura del sistema Web.....	61
7.2 Diagrama de clases.....	62
7.3 Tecnologías de desarrollo.....	63
Capítulo 8. Implementación del sistema Web.....	64
8.1 Integración de OpenCV con Java.....	64
8.2 Integración de OpenCV, Maven y Spring boot en NetBeans.....	65
8.3 Integración con JavaServer Faces y PrimeFaces.....	68
8.4 Desarrollo de la aplicación.....	71
Capítulo 9. Implantación del sistema Web.....	73
9.1 Implantación en AWS.....	73
Capítulo 10. Pruebas del sistema Web.....	75

10.1 Plan de pruebas.....	75
10.2 Resultados.....	76
Anexo A. Código fuente.....	77
Anexo B. Manual de usuario.....	89
1. Características generales y funcionalidades	89
2. Selección de imágenes.....	90
3. Clasificación e interpretación de resultados.....	92
Anexo C. Enfermedades del tomate identificables	94
1. Virus del rizado amarillo del tomate	94
2. Virus del mosaico del tomate.....	95
3. Corynespora cassiicola. (Mancha en forma de blanco).....	96
4. Araña roja.....	97
5. Septoriosis.....	97
6. Passalora fulva (Mojo en la hoja).....	98
7. Tizón tardío.....	99
8. Tizón temprano.....	100
9. Mancha bacteriana.....	101
Referencias.....	102

Capítulo 1.

Introducción.

En las siguientes secciones se proporciona información general acerca del contexto de este proyecto; la situación que dio origen a su desarrollo, el alcance y objetivos de este, la metodología de desarrollo y el marco teórico relacionado con la propuesta solución planteada para el problema de la clasificación de enfermedades de plantas con base en el análisis de imágenes.

1.1 Planteamiento del problema.

La hidroponía es una rama de la agricultura. Es una técnica de producción intensiva de plantas, que se caracteriza por abastecer el agua y los nutrientes de manera controlada y de proporcionar a las plantas los elementos nutritivos en las concentraciones y proporciones más adecuadas, a través de una solución de nutrientes minerales. Para su aplicación se utilizan sustratos inertes diferentes al suelo agrícola a los que se les adiciona en forma constante una solución nutritiva, preparada a partir de fertilizantes comerciales; con esto se logra un medio que proporciona las condiciones físicas, químicas y sanitarias más adecuadas para el desarrollo de los cultivos.

La hidroponía ha sido utilizada en forma comercial desde hace 50 años y se ha adaptado a diferentes situaciones, tanto con cultivos al aire libre como bajo condiciones de invernadero. Este sistema de producción es importante porque permite cultivar especies para el consumo humano en regiones donde no existen tierras de cultivo o donde el clima no es favorable para el cultivo tradicional de ciertas especies [1].

Las ventajas anteriores han favorecido el mercado global de la hidroponía, el cual fue valuado en 411.88 millones de dólares (USD) en el año 2017 y se espera que

crezca a una tasa anual de crecimiento compuesta (CARG) de 12.81% para alcanzar un mercado de 752.57 millones de dólares en el año 2022 [2].

Sin embargo, en México el 60% de los invernaderos de hidroponía que se han instalado han fracasado ante el desconocimiento de productores, la falta de capacitación de técnicos y de mercado [3]. Parte de este problema es el mantener saludables las plantas, ya que la deficiencia de algún nutriente en la solución puede causar que los frutos no sean aptos para consumo humano, o en casos extremos, que la planta muera. Además, existen otro tipo de enfermedades derivadas de bacterias, virus o insectos que una planta puede padecer. Identificar la enfermedad tomando como referencia los síntomas visibles que presenta la planta no es una tarea sencilla, ya que requiere de conocimientos y experiencia para realizar un diagnóstico preciso.

El prototipo propuesto tiene como finalidad ayudar al agricultor a diagnosticar posibles enfermedades de plantas con base en los síntomas visibles que se presenten. Como se verá más adelante, la solución propuesta permite clasificar diferentes tipos de enfermedades y plantas, siempre y cuando se tenga una cantidad suficiente de imágenes y una computadora con los recursos suficientes para entrenar el algoritmo clasificador. En dicha solución se considera un conjunto de datos de 16,419 imágenes distribuidas en diez clases, nueve clases para las enfermedades del tomate y una más para las plantas sanas. La cantidad de imágenes por cada enfermedad se muestra en la tabla 1.1. En el Anexo C se presentan los síntomas característicos de cada una de las enfermedades mencionadas.

Enfermedad	Número de imágenes
Virus del rizado amarillo del tomate (<i>Tomato yellow leaf curl virus</i>)	4032
Virus del mosaico del tomate (<i>Tomato mosaic virus</i>)	325
Corynespora cassiicola. Mancha en forma de blanco (Target spot)	1,356
Araña roja (<i>Spider mites</i>)	1,628
Septoriosis (<i>Septoria spot</i>)	1,723
Passalora fulva. Moho en la hoja (<i>Leaf mold</i>)	904
Tizón tardío (<i>Lateblight</i>)	1,781
Tizón temprano (<i>Earlyblight</i>)	952
Mancha bacteriana (<i>Bacterial spot</i>)	2,127
Total	14,828

Tabla 1.1 Conjunto de datos del proyecto.

1.2 Objetivos.

Los objetivos de este proyecto están clasificados en un objetivo general y en diversos objetivos para metas particulares. Dichos objetivos se describen a continuación.

1.2.1 Objetivo general.

Diseñar y desarrollar el prototipo de una aplicación de visión por computadora que analiza imágenes de hojas de tomate con una anomalía visible y realiza un diagnóstico de una posible enfermedad, bajo un subconjunto predefinido de enfermedades del tomate.

1.2.2 Objetivos particulares.

- Entrenar el modelo de clasificación de imágenes con el conjunto de datos predefinido.
- Realizar pruebas de eficiencia del clasificador, obteniendo un resultado superior al 90%.
- Implementar un sistema web en el que se aloje el clasificador y permita realizar identificaciones de enfermedades a los usuarios.

1.3 Justificación.

La arquitectura general de sistemas de clasificación de enfermedades de plantas a través del análisis imágenes consiste de tres etapas: pre-procesamiento, extracción de descriptores y clasificación [4].

En el pre-procesamiento las imágenes son preparadas usando algunas operaciones, tales como conversión de espacio de color de RGB a algún otro. También suele incluirse la eliminación el fondo de la imagen tratando de concentrar el análisis en el objeto de interés. Desafortunadamente, este tipo de operación es complicada y

algunas veces requiere de la intervención del usuario, lo cual decrece la automatización del sistema.

Los descriptores propuestos por expertos son extraídos de la imagen para formar un vector descriptor. Ejemplos de descriptores son los momentos de color o de textura obtenidos a partir de la matriz de co-ocurrencia. El objetivo de un descriptor es, a partir de su valor, poder clasificar un objeto entre distintas clases.

La última etapa determina a que clase pertenece el objeto no identificado a través de un modelo o algoritmo de clasificación. Este modelo debe ser entrenado usando algoritmos de aprendizaje y muestras de imágenes pre-clasificadas (muestras etiquetadas). Ejemplos de este tipo de algoritmos son la Máquina de Vectores de Soporte (SVM), los k-vecinos más cercanos (KNN) y la Red Neuronal Artificial (ANN).

Las etapas de pre-procesamiento y extracción de descriptores implican operaciones que suelen ser complejas y consumir tiempo considerable. Además, debido a esto es común solicitar asistencia del usuario lo que hace que el sistema no esté completamente automatizado.

En este proyecto se plantea el uso de un modelo de aprendizaje profundo, específicamente de una Red Neuronal Convolutacional (CNN) como una alternativa a la clasificación de enfermedades en plantas. Este modelo permite que los descriptores sean definidos y obtenidos de una forma completamente automática. Además, permite realizar la etapa de entrenamiento y clasificación directamente sobre las imágenes, sin incluir operaciones de pre-procesamiento. Las características anteriores permitirán desarrollar un sistema completamente automatizado en el que la única acción que el usuario tendrá que realizar es proporcionar la imagen de la hoja enferma.

1.4 Metodología.

El objetivo de este proyecto y los métodos para alcanzarlo están bien definidos, tal como se describió en las secciones anteriores. Teniendo esto en cuenta y considerando que hay dos etapas fácilmente identificables en el desarrollo del proyecto

(el clasificador y el desarrollo del sistema web) se plante un modelo incremental de desarrollo de software.

El primer incremento consiste en el análisis, diseño y construcción del clasificador haciendo uso de un algoritmo CNN. El segundo incremento consiste en el análisis, diseño y construcción del sistema web en el que será integrado el clasificador para que esté disponible a los usuarios. Lo anterior se encuentra representado en la figura 1.1.

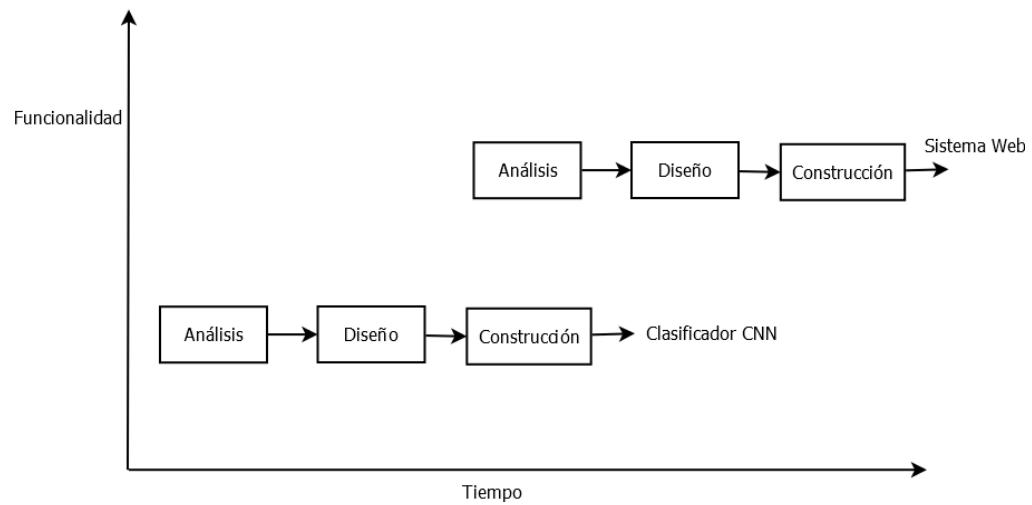


Figura 1.1 Metodología de desarrollo propuesta.

En los capítulos siguientes se desarrolla cada una de las etapas del modelo incremental propuesto.

Capítulo 2.

Marco teórico.

El aprendizaje profundo es una tendencia reciente en el aprendizaje automático (*machine learning*), el cual ha tenido éxito en áreas como la visión por computadora, el reconocimiento automático de voz y el procesamiento de lenguaje natural.

2.1 Comparación de clasificación de imágenes usando modelos de aprendizaje automático y aprendizaje profundo.

La clasificación de imágenes usando algoritmos de aprendizaje automático está compuesta de dos fases:

- Fase de entrenamiento. En esta fase se entrena el algoritmo usando un conjunto de datos pre-clasificados (muestras etiquetadas).
- Fase de predicción. En esta fase se utiliza el algoritmo entrenado para predecir la etiqueta de imágenes fuera del conjunto de entrenamiento.

La fase de entrenamiento para el problema de clasificación de imágenes tiene dos etapas principales:

1. Extracción de descriptores. En esta etapa se hace uso del conocimiento general en el área para seleccionar y extraer descriptores que serán usados por el algoritmo de aprendizaje automático, de acuerdo con el tipo de imágenes a clasificar. HOG y SIFT (Histograma de Gradientes Orientados y Transformación de Características Invariante a la Escala, respectivamente) son ejemplos de descriptores usados en la clasificación de imágenes.

2. Entrenamiento del modelo. En esta etapa se hace uso de un conjunto de entrenamiento compuesto por descriptores de imágenes y sus etiquetas correspondientes para entrenar el modelo de aprendizaje automático.

En la fase de predicción, se aplica el mismo proceso de extracción de descriptores a imágenes nuevas y los descriptores obtenidos se pasan al algoritmo de aprendizaje automático para predecir la etiqueta o clase a la que pertenece. Ambas etapas, entrenamiento y predicción se muestran de forma esquemática en la figura 2.1.

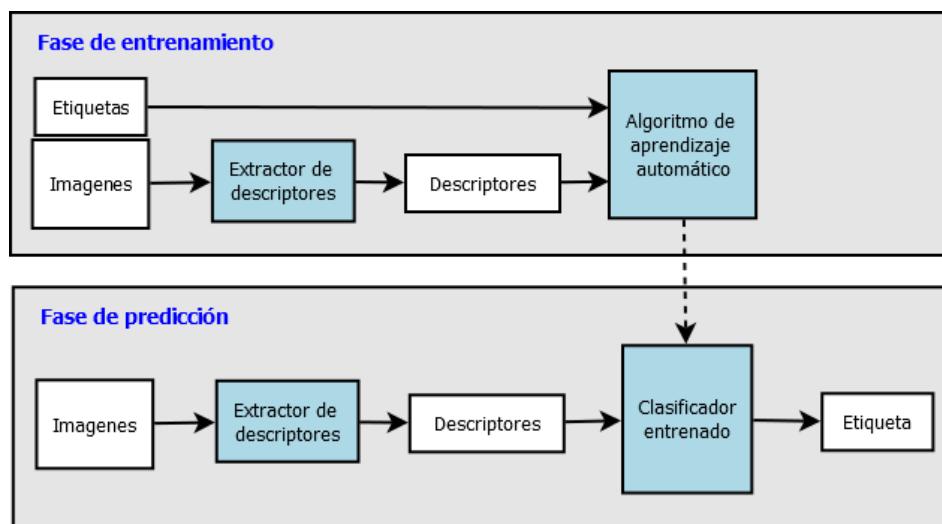
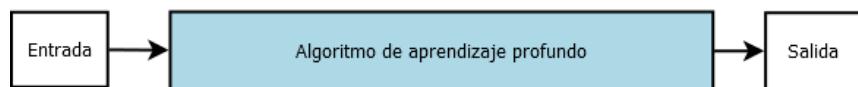


Figura 2.1 Clasificación con algoritmos de aprendizaje automático.

Ahora, la principal diferencia de los algoritmos tradicionales de aprendizaje automático y los algoritmos de aprendizaje profundo radica en la ingeniería de descriptores. En los algoritmos tradicionales de aprendizaje automático, este proceso es elaborado manualmente, es decir, la selección y extracción debe ser diseñada e implementada por el programador, con base en el conocimiento y recomendaciones de expertos. En cambio, en los algoritmos de aprendizaje profundo, la ingeniería de descriptores es realizada automáticamente por el algoritmo. La ingeniería de descriptores es costosa, consume tiempo importante y requiere de cierta experiencia. Un descriptor mal seleccionado condena al fracaso el resto del clasificador. Es así como los algoritmos de aprendizaje profundo prometen resultados más precisos comparados con los algoritmos tradicionales de aprendizaje automático, con menos o incluso sin ingeniería de descriptores. Esta diferencia se muestra en la figura 2.2.



Flujo de un modelo tradicional de aprendizaje automático



Flujo de un modelo de aprendizaje profundo

Figura 2.2 Diferencias entre la clasificación con modelos de aprendizaje automático y aprendizaje profundo.

2.2 Redes Neuronales Artificiales (ANN).

Aprendizaje profundo se refiere a una clase de redes neuronales artificiales compuestas de muchas capas de procesamiento. Las redes neuronales tienen décadas de existencia pero el reciente éxito y popularidad del aprendizaje profundo puede rastrearse a la publicación “*ImageNet Classification with Deep Convolutional Networks*” realizada por Krizhevsky, Sutskever, y Hinton en el año 2012. En dicha publicación demostraron por primera vez como una CNN puede superar en desempeño, de forma contundente, a otros métodos tradicionales de clasificación de imágenes. Además, factores como el incremento del poder computacional de las máquinas, el uso de GPUs y la disponibilidad de grandes conjuntos de datos han favorecido el desarrollo de los modelos de aprendizaje profundo.

Las neuronas artificiales están inspiradas en las neuronas biológicas. Una neurona artificial tiene un número finito de entradas con pesos asociadas a ellas, y una función de activación. La salida de la neurona es el resultado de la función de activación aplicada a la suma de las entradas con los pesos correspondientes (figura 2.3). Las neuronas artificiales se conectan entre sí para formar redes neuronales artificiales.

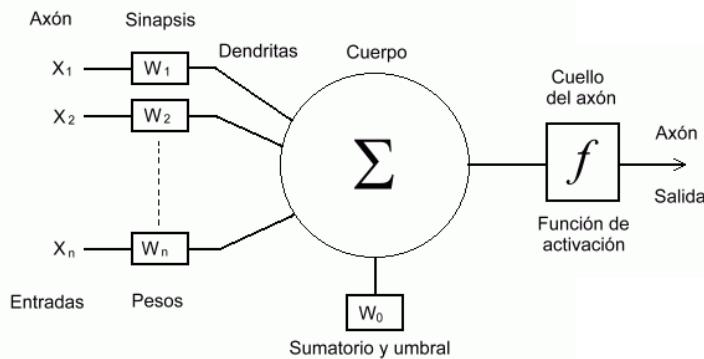


Figura 2.3 Estructura de una neurona artificial.

2.2.1 Redes neuronales unidireccionales.

Las redes neuronales unidireccionales (*feedforward*) son las más sencillas de las redes neuronales artificiales. Estas redes tienen tres tipos de capas: capa de entrada, capa oculta y capa de salida. Los datos se mueven de la capa de entrada hacia las neuronas ocultas y desde estas hacia las neuronas de salida. En la figura 2.4 se muestra un ejemplo de una red unidireccional en la que cada nodo o neurona está conectada a todas las de la siguiente capa. Este tipo de conexión es llamado “Completamente conectado” (*fully-connected*). El número de capas ocultas y su tamaño es variable. Mientras más grandes y de mayor profundidad sean las capas ocultas, pueden modelarse patrones más complejos.

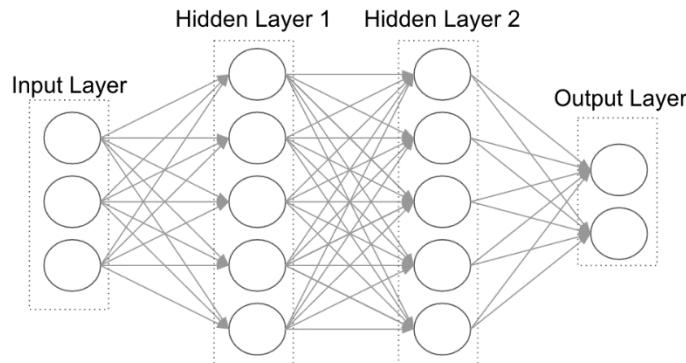


Figura 2.4 Red neuronal unidireccional con dos capas ocultas.

2.2.2 Funciones de activación.

Las funciones de activación transforman la suma de las entradas con los pesos aplicados. Estas funciones deben ser no lineales para permitir codificar patrones complejos de datos. Las funciones de activación más comunes son las funciones

Sigmoidal, tangente hiperbólica y ReLU (figura 2.5). En las redes neuronales profundas (con muchas capas ocultas) la función más usada es la ReLU.

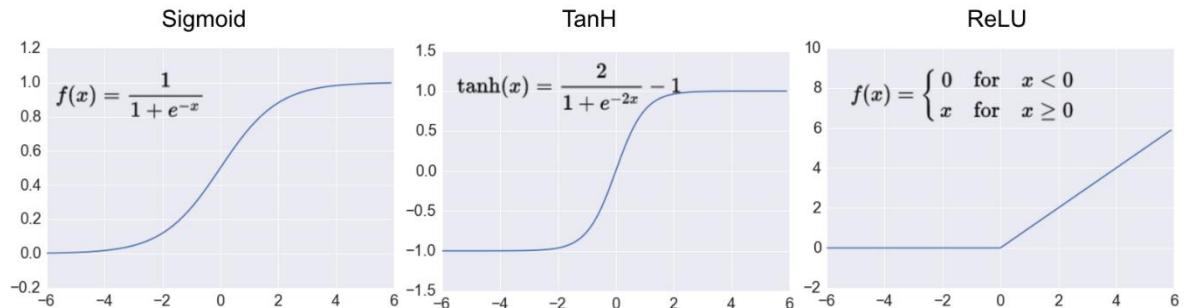


Figura 2.5 Funciones de activación Sigmoidal, tanh y ReLU.

2.2.3 Entrenamiento de una red neuronal artificial.

El objetivo de la fase de entrenamiento es determinar los valores de los pesos de la red neuronal. Para alcanzar este objetivo son necesarios dos elementos:

- El conjunto de datos de entrenamiento. En el caso de clasificación de imágenes el conjunto de datos está compuesto de imágenes y sus correspondientes etiquetas.
- Una función de pérdida. Esta función mide la imprecisión de las predicciones.

Con estos dos elementos se entrena la red neuronal con un algoritmo llamado “propagación hacia atrás” (*backpropagation*) [5].

2.3 Redes Neuronales Convolucionales (CNN).

Las redes neuronales convolucionales son un tipo especial de redes unidireccionales. Estos modelos están diseñados para emular el comportamiento de la corteza visual, por lo que asumen que la entrada de la red son imágenes. Las CNNs tienen un muy buen desempeño en tareas de reconocimiento visual. La arquitectura de una CNN está compuesta por tres capas: Capa convolucional (CONV), capa de agrupamiento (*pooling*) y capa conectada completamente (*fully-connected*). Estas

capas permiten a la red codificar determinadas propiedades de la imagen. En la figura 2.6 se muestra la estructura de la CNN LeNet.

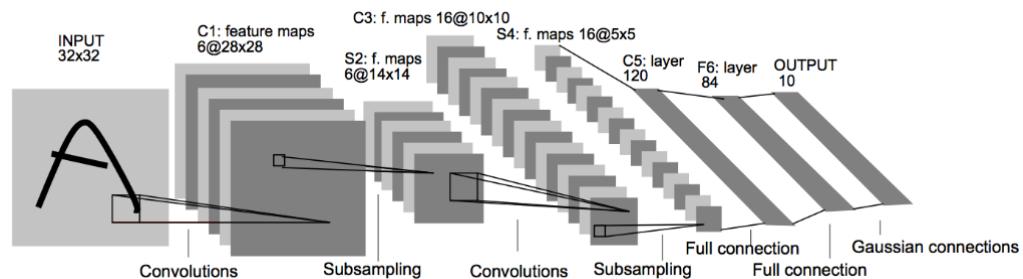


Figura 2.6 CNN LeNet.

2.3.1 Capa convolucional.

La capa convolucional es el bloque fundamental con que se construye una red convolucional y donde se realiza la mayor parte del trabajo computacional de la red. Los parámetros de la red convolucional consisten en un conjunto de filtros aprendibles. Cada filtro es pequeño espacialmente, pero es desplazado por toda la imagen calculando la convolución entre el filtro y la imagen de entrada. Conforme el filtro se desplaza por el alto y ancho de la imagen, se producen mapas de activación de dos dimensiones que indican la respuesta de dicho filtro en cada posición espacial. De forma intuitiva la red aprenderá los filtros que se activan cuando se encuentran algún tipo de característica visual, tales como bordes, orientaciones o regiones de algún color. Lo anterior en las primeras capas de la red. En capas superiores pueden detectarse características más complejas como ruedas, señales de tránsito, etc.

Cada neurona en la capa convolucional está conectada a una región local espacial de la imagen de entrada, pero considera todos los canales de color. En la figura 2.7 se tienen cinco neuronas de la capa convolucional, todas ellas conectadas a la misma región de la imagen de entrada.

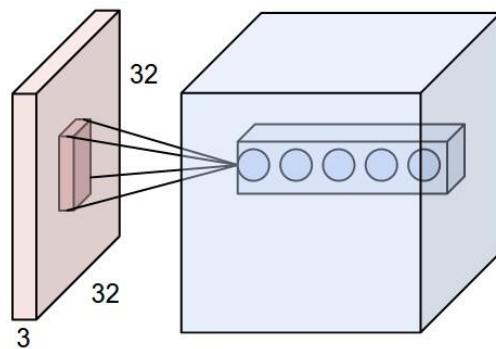


Figura 2.7 Conectividad local de las neuronas de la capa convolucional.

2.3.2 Capa de agrupamiento.

Es común en una arquitectura CNN insertar periódicamente capas de agrupamiento entre capas convolucionales sucesivas, con la intención de progresivamente reducir el tamaño espacial de la representación para reducir la cantidad de parámetros y el cómputo en la red. De esta forma se controla el sobreajuste, para evitar entrenar de más la red.

La capa de agrupamiento ajusta el tamaño espacial empleando la operación MAX. La forma más común es tomando un filtro de 2x2 aplicado a toda la entrada espacialmente. De esta forma se reduce en un 75% las activaciones (figura 2.8).

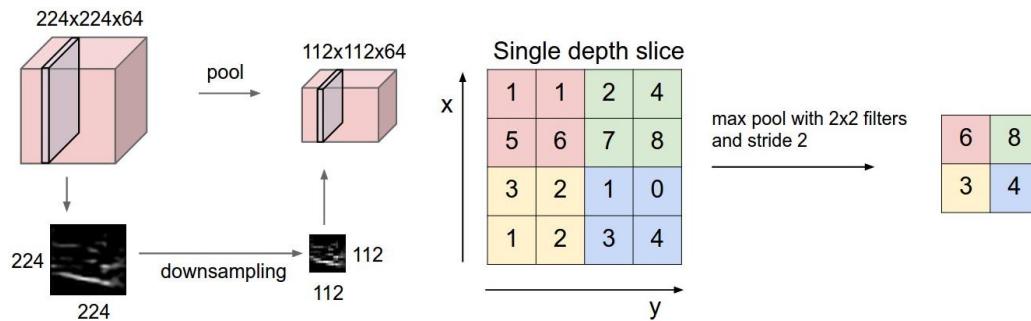


Figura 2.8 Reducción del volumen en la capa de agrupamiento.

2.4 Arquitectura de las redes neuronales convolucionales.

La arquitectura más sencilla de una red neuronal convolucional comienza con una capa de entrada (imágenes) seguida de una secuencia de capas convolucionales

y de agrupamiento, terminando con capas completamente conectadas. Las capas convolucionales usualmente están seguidas de una capa de funciones de activación ReLU (figura 2.9).

Las capas convolucionales, de agrupamiento y ReLU actúan como extractores de descriptores aprendibles, mientras que las capas completamente conectadas actúan como un clasificador de aprendizaje automático. Además, las primeras capas de la red codifican patrones genéricos de las imágenes, mientras que las capas posteriores codifican patrones detallados de las imágenes.

También debe mencionarse que solo las capas convolucionales y completamente conectadas tienen pesos, los cuales son aprendidos en la etapa de entrenamiento [6].

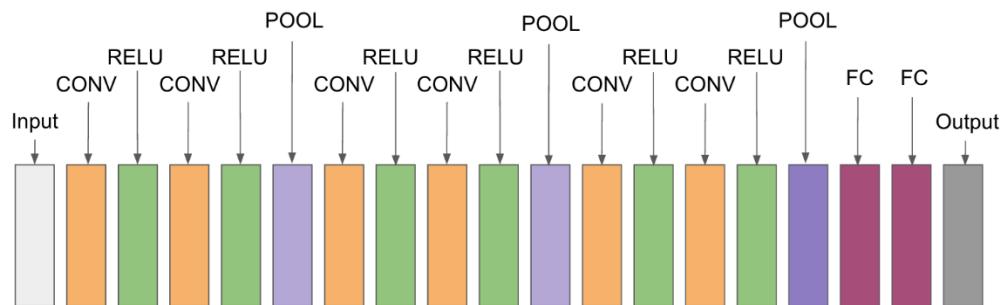


Figura 2.9 Ejemplo de arquitectura CNN.

Existen distintas arquitecturas con un nombre asignado. Las más comunes son las siguientes:

- LeNet. Desarrollada por Yann LeCun en la década de 1990. Con esta arquitectura se desarrollaron las primeras aplicaciones exitosas de redes convolucionales.
- AlexNet. Es el primer trabajo que popularizó las redes convolucionales en la visión por computadora, desarrollado por Alex Krizhevsky, Ilya Sutskever y Geoff Hinton. La AlexNet fue inscrita en el reto *ImageNet Large Scale Visual Recognition Competition (ILSVRC)* en 2012 y superó de forma significativa al segundo clasificado.
- ZF Net. Una mejora de la AlexNet que resultó ganadora del ILSVRC 2013, desarrollada por Matthew Zeiler y Rob Fergus.
- GoogLeNet. La ganadora del ILSVRC 2014, desarrollada por Szegedy et al [7] de Google.

Capítulo 3.

Análisis del clasificador.

En esta sección se describe el análisis del clasificador, correspondiente al primer incremento propuesto en la metodología de la sección 1.4.

Esta sección se realizó teniendo en consideración el conjunto de datos que se logró recopilar, descrito en la sección 1.1. Este conjunto de datos es un subconjunto del repositorio de acceso abierto publicado en www.PlantVillage.org. En el momento del desarrollo de este proyecto, dicho repositorio se encontraba temporalmente inhabilitado, por lo que las imágenes usadas en este proyecto fueron facilitadas por Brahimí M. [8].

3.1 Requisitos del sistema.

Este proyecto surgió de la necesidad de poder identificar de forma automática enfermedades en plantas, a partir de síntomas visuales. Y además, teniendo en consideración los objetivos y el alcance de este proyecto, los requisitos de software de este incremento de la metodología, se describen a continuación.

3.1.1 Requisitos de software.

Requisitos funcionales.

RF01. Clasificación.

Nivel de madurez: Alta.

Prioridad: Alta.

Descripción: El sistema clasificará la imagen en alguna de las nueve enfermedades descritas en la tabla 1.1 o en hoja sana, indicando la probabilidad de la predicción.

Requisitos no funcionales.

RNF01. La eficiencia de clasificación deberá ser superior al 90%.

RNF02. El clasificador identificará la clase de la imagen de entrada en un tiempo no mayor a cinco segundos.

3.1.2 Requisitos de hardware.

El mayor trabajo computacional se realiza durante la etapa de entrenamiento del clasificador, donde dependiendo del cpu, gpu y de la cantidad de memoria, el proceso puede extenderse por horas, o incluso días. Sin embargo, una vez entrenado el clasificador, predecir la clase de una imagen es una tarea muy sencilla que se lleva a cabo en cuestión de algunos segundos. Por lo anterior, el dispositivo en el que se ejecuten las predicciones, ya con el clasificador entrenado, no se ven restringido en este sentido.

La etapa de entrenamiento será realizada con el framework Caffe. Caffe es un framework para aprendizaje profundo desarrollado en C++ por Berkeley AI Research (BAIR). Los requisitos del sistema para este framework realmente dependen de las acciones que se vayan a realizar. Para este proyecto se realizará una calibración fina de una red pre-entrenada (fine-tuning), lo que reduce el cómputo. Esto se detalla en el capítulo de diseño de este incremento. Por la razón anterior, una computadora con los requisitos siguientes es suficiente:

- CPU: Intel core i3.
- RAM: 4 GB.
- GPU*: Se requiere CUDA para el modo GPU, versión 6 o superior.

La GPU es opcional pero su uso puede reducir el tiempo de cómputo a unas pocas horas. De forma similar, un mejor CPU o más RAM ayudan en el mismo sentido.

3.2 Modelo de casos de uso.

El clasificador no tiene interacción directa con el usuario, por lo que en este incremento un modelo de casos de uso no resulta aplicable. El clasificador se encontrará en un entorno en el que a través de una interfaz gráfica el usuario pueda clasificar sus imágenes. Esto se aborda en el análisis del segundo incremento.

3.3 Interfaz gráfica.

Por la misma razón expuesta en la sección 3.2, una interfaz gráfica no resulta aplicable en este incremento.

Capítulo 4.

Diseño del clasificador.

Como ya se ha mencionado anteriormente, el algoritmo clasificador será un algoritmo de aprendizaje profundo, en particular una red neuronal convolucional (CNN). Lo anterior debido a las ventajas que presenta al ser comparado con los algoritmos tradicionales de aprendizaje automático, especialmente en la etapa de la ingeniería de descriptores.

Algunas de las arquitecturas de CNNs fueron mencionadas en la sección 2.4. Entre ellas la AlexNet y GoogLeNet son las más populares debido a la calidad de los resultados obtenidos en distintas aplicaciones, tales como el *ImageNet Large Scale Visual Recognition Competition* (ILSVRC). Estas dos arquitecturas será usadas inicialmente, y al final será seleccionada la que presente una mayor eficiencia.

4.1 Calibración fina.

Para implementar una red neuronal convolucional se tiene básicamente dos opciones: entrenar toda la red neuronal desde cero (con valores iniciales aleatorios), o usar una red pre-entrenada con un conjunto muy grande de datos y adaptarla a las necesidades particulares.

En la práctica, muy pocas personas entran completamente una red convolucional desde cero, porque es muy raro disponer de un conjunto de datos del tamaño suficiente. En cambio, es muy común pre-entrenar una red convolucional sobre un conjunto de datos muy grande (por ejemplo ImageNet, que contiene 1.2 millones de imágenes en mil categorías distintas), y entonces usar la red convolucional, ya sea como inicialización o como un extracto de descriptores fijo para la tarea de interés modificando solo las últimas capas. Los tres principales escenarios de transferencia de aprendizaje son los siguientes:

- **CNN como un extractor de descriptores fijo.** Se toma una red convolucional pre-entrenada sobre ImageNet y se remueve la última capa completamente conectada (las salidas de esta red son las mil clases distintas de ImageNet). Entonces el resto de la red convolucional se trata como un extractor de descriptores fijo para un nuevo conjunto de datos. Este extractor de descriptores puede usarse para entrenar un clasificador lineal (SVM por ejemplo) para el nuevo conjunto de datos.
- **Calibración fina de la red convolucional.** Esta estrategia consiste en reemplazar y reentrenar el clasificador que se encuentra al final de la red convolucional, sobre el nuevo conjunto de datos. También considera volver a ajustar los pesos de la red pre-entrenada continuando con la propagación hacia atrás. Es posible volver a ajustar todas las capas de la red o también conservar algunas de las primeras capas intactas y las demás ajustarlas. Esto último considerando la observación de que las primeras capas de la red contienen descriptores genéricos (bordes o colores por ejemplo) que deberían ser útiles en muchas aplicaciones, pero en las capas siguientes la red convolucional se vuelve progresivamente más específica de acuerdo a los detalles contenidos en las imágenes del conjunto de datos original.
- **Modelos pre-entrenados.** Debido a que el proceso de entrenamiento sobre ImageNet de las redes convolucionales modernas toma entre dos y tres semanas, considerando el uso de múltiples GPUs, es muy común que las personas compartan sus modelos entrenados para el beneficio de otros, quienes pueden usar esos modelos para realizar calibración fina.

Ahora, teniendo en consideración que el conjunto de datos de este proyecto es relativamente grande (16,419 imágenes) es factible la realización de una calibración sobre una red pre-entrenada en ImageNet [9].

4.2 Caffe. Framework de aprendizaje profundo.

Caffe es un framework para aprendizaje profundo desarrollado en C++ por Berkeley AI Research (BAIR). Algunas de los beneficios de este framework son los siguientes [10]:

- Los modelos y la optimización son definidos mediante archivos de configuración sin la necesidad de editar código fuente directamente. Puede seleccionarse el modo CPU + GPU o únicamente CPU modificando una sola bandera.
- Código extensible. Durante su primer año, el repositorio de caffe fue bifurcado (*forked*) por más de mil desarrolladores que contribuyeron a cambios significativos mediante su retroalimentación. Gracias a estos colaboradores el framework se mantiene alineado con el estado del arte del aprendizaje profundo.
- Caffe puede procesar alrededor de sesenta millones de imágenes por día con una sola GPU NVIDIA K40 (1 ms/imagen).
- Caffe cuenta con una comunidad formada por académicos, investigadores, ingenieros, etcétera. Incluso tiene un “*Model Zoo*” en el que los usuarios comparten sus modelos de Caffe para el beneficio mutuo de la comunidad.

Para realizar el entrenamiento de la red se tomará modelos de caffe pre-entrenados sobre ImageNet, en particular de la GoogLeNet y de una adaptación de la AlexNet para caffe llamada BVLC reference caffenet [11]. Para la etapa de entrenamiento se realizará un proceso de calibración fina sobre el conjunto de datos de este proyecto.

Para lo anterior se instalará Caffe en una computadora con las siguientes características.

- Sistema operativo: Ubuntu 16.04
- CPU: Intel Core i3
- RAM: 4 GB

Dicho equipo no cuenta con una GPU (CUDA) compatible con Caffe, por lo que la instalación se realizara en modo solo CPU. Esto genera el inconveniente de un mayor tiempo de cómputo, sin embargo, para fines de este prototipo, no representa un problema mayor.

Capítulo 5.

Construcción del clasificador.

En este capítulo se describen los diversos procedimientos y acciones realizadas para llevar a cabo la construcción del clasificador, el cual consiste en el entrenamiento de la red neuronal convolucional sobre el conjunto de datos de este proyecto.

5.1 Instalación de Caffe.

Las instrucciones siguientes [12] indican los pasos realizados para la instalación de Caffe en un sistema operativo Ubuntu 16.04 en modo solo CPU, ya que, como se mencionó en el capítulo anterior, la computadora usada no cuenta con una GPU compatible.

5.1.1 Instalación de OpenCV.

Las siguientes constituyen las instrucciones seguidas para instalar OpenCV 3.3 y las bibliotecas para Python. [13].

Actualizar los paquetes y bibliotecas pre-instaladas.

```
sudo apt-get update  
sudo apt-get upgrade
```

Instalar las siguientes herramientas de desarrollo.

```
sudo apt-get install build-essential cmake pkg-config
```

Instalar las bibliotecas para leer imágenes desde disco.

```
sudo apt-get install libjpeg8-dev libtiff5-dev libjasper-dev  
libpng12-dev
```

Instalar las bibliotecas para procesamiento de video.

```
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev  
libv4l-dev  
sudo apt-get install libxvidcore-dev libx264-dev
```

Instalar la biblioteca GTK y bibliotecas para la optimización de ciertas funcionalidades de OpenCV.

```
sudo apt-get install libgtk-3-dev  
sudo apt-get install libatlas-base-dev gfortran
```

Instalar los archivos de desarrollo de Python 2.7 y Python 3.5.

```
sudo apt-get install python2.7-dev python3.5-dev  
cd ~  
wget https://bootstrap.pypa.io/get-pip.py  
sudo python get-pip.py  
pip install numpy
```

Descargar el código fuente de OpenCV.

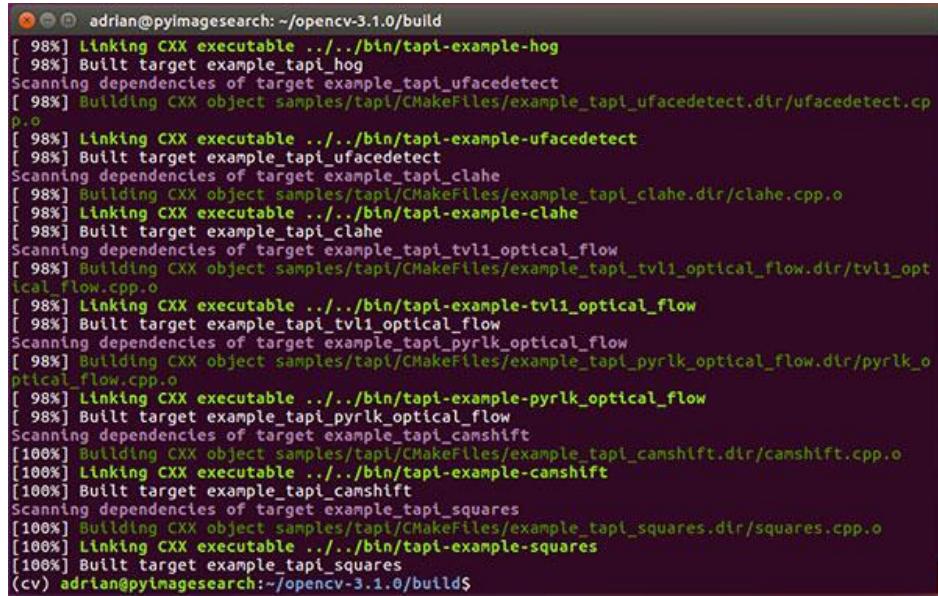
```
cd ~  
wget -O opencv.zip https://github.com/Itseez/opencv/archive/3.3.0.zip  
unzip opencv.zip  
wget -O opencv_contrib.zip https://github.com/Itseez/opencv_contrib/archive/3.3.0.zip  
unzip opencv_contrib.zip
```

Configurar OpenCV con CMake.

```
cd ~/opencv-3.1.0/  
mkdir build  
cd build  
cmake -D CMAKE_BUILD_TYPE=RELEASE \  
      -D CMAKE_INSTALL_PREFIX=/usr/local \  
      -D INSTALL_PYTHON_EXAMPLES=ON \  
      -D INSTALL_C_EXAMPLES=OFF \  
      -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-3.3.0/modules \  
      -D PYTHON_EXECUTABLE=/bin/python \  
      -D BUILD_EXAMPLES=ON  
      -D BUILD_SHARED_LIBS=OFF ..
```

Compilar OpenCV. La figura 5.1 muestra la captura de una compilación exitosa de OpenCV y python en Ubuntu 16.04.

```
make
```



A terminal window showing the output of a 'make' command. The output is a series of build logs for various OpenCV samples. It includes messages like '[98%] Linking CXX executable .../bin/tapi-example-hog', '[98%] Built target example_tapi_hog', and '[100%] Built target example_tapi_squares'. The terminal window has a dark background with light-colored text. The cursor is visible at the bottom left. The title bar of the terminal window shows the path: 'adrian@pyimagesearch:~/opencv-3.1.0/build'.

```
[ 98%] Linking CXX executable .../bin/tapi-example-hog
[ 98%] Built target example_tapi_hog
Scanning dependencies of target example_tapi_ufacedetect
[ 98%] Building CXX object samples/tapi/CMakeFiles/example_tapi_ufacedetect.dir/ufacedetect.cpp.o
[ 98%] Linking CXX executable .../bin/tapi-example-ufacedetect
[ 98%] Built target example_tapi_ufacedetect
Scanning dependencies of target example_tapi_clahe
[ 98%] Building CXX object samples/tapi/CMakeFiles/example_tapi_clahe.dir/clahe.cpp.o
[ 98%] Linking CXX executable .../bin/tapi-example-clahe
[ 98%] Built target example_tapi_clahe
Scanning dependencies of target example_tapi_tvli_optical_flow
[ 98%] Building CXX object samples/tapi/CMakeFiles/example_tapi_tvli_optical_flow.dir/tvli_optical_flow.cpp.o
[ 98%] Linking CXX executable .../bin/tapi-example-tvli_optical_flow
[ 98%] Built target example_tapi_tvli_optical_flow
Scanning dependencies of target example_tapi_pyrLK_optical_flow
[ 98%] Building CXX object samples/tapi/CMakeFiles/example_tapi_pyrLK_optical_flow.dir/pyrLK_optical_flow.cpp.o
[ 98%] Linking CXX executable .../bin/tapi-example-pyrLK_optical_flow
[ 98%] Built target example_tapi_pyrLK_optical_flow
Scanning dependencies of target example_tapi_camshift
[100%] Building CXX object samples/tapi/CMakeFiles/example_tapi_camshift.dir/camshift.cpp.o
[100%] Linking CXX executable .../bin/tapi-example-camshift
[100%] Built target example_tapi_camshift
Scanning dependencies of target example_tapi_squares
[100%] Building CXX object samples/tapi/CMakeFiles/example_tapi_squares.dir/squares.cpp.o
[100%] Linking CXX executable .../bin/tapi-example-squares
[100%] Built target example_tapi_squares
(cv) adrian@pyimagesearch:~/opencv-3.1.0/build$
```

Figura 5.1 Compilación exitosa de OpenCV.

Instalar OpenCV.

```
sudo make install
sudo ldconfig
```

5.1.2 Instalación de Caffe.

Ejecutar los siguientes comandos.

```
sudo apt-get install -y build-essential cmake git pkg-config

sudo apt-get install -y libprotobuf-dev liblibleveldb-dev libsnapy-dev libhdf5-serial-dev protobuf-compiler

sudo apt-get install -y libatlas-base-dev

sudo apt-get install -y --no-install-recommends libboost-all-dev

sudo apt-get install -y libgflags-dev libgoogle-glog-dev liblmdb-dev
```

Instalar los archivos de desarrollo de Python 3.5.

```
sudo apt-get install -y python3-dev  
sudo apt-get install -y python3-numpy python3-scipy
```

Descargar la versión 1.0RC5 de caffe desde <https://github.com/BVLC/caffe/archive/rc5.zip> y descomprimirla.

Crear una copia del archivo Makefile.config.example.

```
cp Makefile.config.example Makefile.config
```

Modificar el archivo creado de la siguiente forma. Descomentar la línea:

```
CPU_ONLY := 1
```

Ubicar las variables PYTHON_INCLUDE, INCLUDE_DIRS y LIBRARY_DIRS y modificarlas de la forma siguiente:

```
PYTHON_INCLUDE := /usr/include/python3.5m \  
                  /usr/lib/python3.5/site-packages/numpy/core/include  
WITH_PYTHON_LAYER := 1  
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/serial  
LIBRARY_DIRS := $(PYTHON_LIB) /usr/local/lib /usr/lib  
                /usr/lib/x86_64-linux-gnu /usr/lib/x86_64-linux-gnu/hdf5/serial
```

Ahora, dentro del directorio raíz de Caffe, ejecutar lo siguiente:

```
cd python  
for req in $(cat requirements.txt); do sudo pip install $req; done
```

Editar el archivo Makefile con lo siguiente. Identificar esta línea:

```
NVCCFLAGS += -ccbin=$(CXX) -Xcompiler -fPIC $(COMMON_FLAGS)
```

Y reemplazarla con lo siguiente:

```
NVCCFLAGS += -D_FORCE_INLINES -ccbin=$(CXX) -Xcompiler -fPIC $(COMMON_FLAGS)
```

También agregar en el mismo archivo:

```
LIBRARIES += glog gflags protobuf leveldb snappy \
    lmdb boost_system boost_filesystem hdf5_serial_hl hdf5_serial m \
    opencv_core opencv_highgui opencv_imgproc opencv_imgcodecs
opencv_videoio
```

Editar el archivo CMakeLists.txt y agregar lo siguiente:

```
# ---[ Includes
set(${CMAKE_CXX_FLAGS} "-D_FORCE_INLINES ${CMAKE_CXX_FLAGS}")
```

Ahora, compilar, instalar y probar la instalación de Caffe con los siguientes comandos:

```
make all
make test
make runtest
make pycaffe
make distribute
```

Los archivos Makefile y Makefile.config se encuentran en el disco adjunto con este documento.

5.2 Calibración fina de las redes GoogLeNet y CaffeNet con Caffe.

Como se ha mencionado anteriormente, Caffe cuenta con un repositorio que cuenta con modelos ya entrenados sobre ImageNet de diferentes arquitecturas de redes convolucionales. Dichos modelos se encuentran en <https://github.com/BVLC/caffe/tree/master/models/>. En dicha ubicación se encuentran los modelos de la GoogLeNet y la CaffeNet. Los archivos incluidos con cada modelo y su finalidad se describen a continuación.

*.caffemodel: El modelo Caffe entrenado de la arquitectura correspondiente, es decir, un archivo que incluye los pesos de la red.

train_val.prototxt: La definición del modelo donde se especifican los datos de entrada, el número de salidas y la especificación de cada capa de la red convolucional.

solver.prototxt: La definición del solucionador, el cual es responsable de la optimización del modelo. En este archivo se definen los parámetros del solucionador.

Para el proceso de fine tuning existen ciertos parámetros que deben modificarse en ambos archivos prototxt, como el coeficiente de aprendizaje y la especificación del clasificador. Dichos cambios se analizarán en las secciones 5.2.2 y 5.2.3.

5.2.1 Preparación de las imágenes.

El proceso de entrenamiento de la red se encuentra acompañado de etapas periódicas de prueba que sirven para evaluar el desempeño de la red. Es así como el conjunto de imágenes de entrada debe separarse en dos subconjuntos, uno exclusivamente para entrenamiento y otro para prueba de la red.

Para especificar estos dos subconjuntos se generan dos archivos de texto que contienen una lista de las imágenes con su correspondiente etiqueta, de la forma siguiente:

/ubicación/de/la/imagen.jpg clase

Por ejemplo:

```
/yellowLeaf/img01.jpg 0  
/yellowLeaf/img02.jpg 0  
/healty/img01.jpg 1
```

La clase es un número entero, iniciando desde cero, que sirve para identificar cada clase distinta de imágenes. En este caso se tienen diez clases distintas de imágenes, nueve para enfermedades y una para plantas sanas, por lo que la clase será un número entre cero y nueve.

Los registros dentro de los archivos deben estar mezclados para que el proceso de entrenamiento sea efectivo. No se deben colocar en la primera parte del archivo todas las imágenes de la clase cero, seguidas de las imágenes de la clase uno, por ejemplo. Deben estar mezcladas.

El archivo con los registros de las imágenes de entrenamiento es llamado train.txt y el correspondiente a las imágenes de prueba test.txt. Se eligió un 80% del total de imágenes para usarse en el entrenamiento y el restante para las pruebas. Dichos archivos se generaron con un script de Python, el cual se muestra a continuación.

```
import os

TRAIN_FILE = "./data/train.txt"
TEST_FILE = "./data/test.txt"

TRAIN_PERCENT = 0.8

label = 0

trf = open(TRAIN_FILE, 'w')
tef = open(TEST_FILE, 'w')
for root, directories, filenames in os.walk('/home/edgar/git/PlantDiseaseDetection/data/color/'):
    for directory in directories:
        theDir = os.path.join(root, directory)
        images = [f for f in os.listdir(theDir)]

        print('Writing from dir', theDir, '...')
        print('Writing train images', '...')
        for image in images[:int(TRAIN_PERCENT * len(images))]:
            trf.write(os.path.join(theDir, image) + ' ' + str(label) +
'\n')

        print('Writing test images', '...')
        for image in images[int(TRAIN_PERCENT * len(images)) : ]:
            tef.write(os.path.join(theDir, image) + ' ' + str(label) +
'\n')

    label = label + 1

trf.close()
tef.close()
```

El script anterior no genera el contenido de los archivos de forma aleatoria, sino de forma ordenada de acuerdo con la clase. Para mezclar el contenido de ambos archivos se usó el siguiente script.

```
import random
with open('train.txt', 'r') as source:
    data = [(random.random(), line) for line in source]
data.sort()
```

```

with open('train.txt','w') as target:
    for _, line in data:
        target.write( line )

with open('test.txt','r') as source:
    data = [ (random.random(), line) for line in source ]
data.sort()
with open('test.txt','w') as target:
    for _, line in data:
        target.write( line )

```

5.2.1 Calibración fina de la CaffeNet.

El contenido del archivo solver.prototxt original de la CaffeNet se muestra a continuación:

```

1 net: "models/bvlc_reference_caffenet/train_val.prototxt"
2 test_iter: 1000
3 test_interval: 1000
4 base_lr: 0.01
5 lr_policy: "step"
6 gamma: 0.1
7 stepsize: 100000
8 display: 20
9 max_iter: 450000
10 momentum: 0.9
11 weight_decay: 0.0005
12 snapshot: 10000
13 snapshot_prefix: "models/bvlc_reference_caffenet/caffenet_train"
14 solver_mode: GPU

```

La línea número 1 indica la ubicación del archivo train_val.prototxt por lo que debe actualizarse con la ubicación del archivo modificado para el proceso de fine tuning.

Las líneas 2 y 3 indican parámetros para hacer pruebas de la red cada cierto número de iteraciones de entrenamiento. En este caso indican que cada 1000 iteraciones de entrenamiento se realizarán 1000 iteraciones de prueba.

La línea 4 indica el coeficiente de aprendizaje (*Learning Rate*). En el caso del proceso de calibración fina, dicho coeficiente debería ser menor, debido a que la red ya fue pre-entrenada en un proceso previo y los pesos no deberían modificarse

demasiado, a diferencia de un entrenamiento desde cero, donde los pesos iniciales son aleatorios.

Conforme el entrenamiento avanza, es una práctica común disminuir el coeficiente de aprendizaje. La forma en que esta disminución ocurre se indica en las líneas 5, 6 y 7. El parámetro lr_policy indica el modo en que ocurre la disminución. En el caso “step” se disminuye el coeficiente de aprendizaje cada cierto número de iteraciones (stepsize) por un factor de gamma. Por lo tanto, en un proceso de calibración fina, el intervalo de disminución también debería ser menor, ya que se está más cerca de completar el proceso de entrenamiento comparado con un entrenamiento desde cero.

Dado que el proceso de entrenamiento es largo, caffe ofrece la posibilidad de guardar el estado de la red cada cierto número de iteraciones en un *snapshot*, o también si el proceso es interrumpido presionando la combinación de teclas Ctrl + C. El intervalo de iteraciones se indica en la línea 12 y la ubicación donde se almacena el *snapshot* en la línea 13.

La única línea indica el modo de entrenamiento, CPU más GPU o solo CPU. Como se mencionó antes, el entrenamiento será realizado en modo solo CPU, por lo que dicha línea se omitirá.

Teniendo en cuenta lo anterior, el nuevo archivo solver.prototxt para el proceso de calibración fina queda como se muestra a continuación.

```
net: "/home/edgar/git/PlantDiseaseDetection/deepLearning/models/fine-tune_tomato_diseases/train_val.prototxt"
test_iter: 100
test_interval: 1000
base_lr: 0.001
lr_policy: "step"
gamma: 0.1
stepsize: 20000
display: 20
max_iter: 100000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "/home/edgar/git/PlantDiseaseDetection/deepLearning/models/finetune_tomato_diseases/finetune_tomato_diseases"
```

Los cambios realizados son la ubicación del archivo train_val.txt, el número de iteraciones de prueba, el coeficiente de aprendizaje, el intervalo de reducción del coeficiente y la ubicación de los *snapshots*.

Ahora, el archivo train_val.prototxt contiene la especificación de las capas de la red; la capa de entrada, las capas ocultas y la capa de salida. Los cambios por realizarse en este archivo son en las capas de entrada y de salida.

En la capa de entrada debe especificarse el conjunto de datos de este proyecto, consistente en las imágenes de las enfermedades del tomate. En la capa de salida, originalmente se consideran mil salidas, que son las clases distintas de imágenes de ImageNet. En el caso de las enfermedades de tomate se tienen solo nueve clases distintas, más una clase correspondiente a plantas de tomate sanas, es decir, en total diez clases. Este cambio debe especificarse en la capa de salida.

La capa de entrada original se muestra a continuación:

```

1 name: "CaffeNet"
2 layer {
3   name: "data"
4   type: "Data"
5   top: "data"
6   top: "label"
7   include {
8     phase: TRAIN
9   }
10  transform_param {
11    mirror: true
12    crop_size: 227
13    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
14  }
15  data_param {
16    source: "examples/imagenet/ilsvrc12_train_lmdb"
17    batch_size: 256
18    backend: LMDB
19  }
20 }
21 layer {
22   name: "data"
23   type: "Data"
24   top: "data"
25   top: "label"
26   include {
27     phase: TEST
28   }

```

```

29   transform_param {
30     mirror: false
31     crop_size: 227
32     mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
33   }
34   data_param {
35     source: "examples/imagenet/ilsvrc12_val_lmdb"
36     batch_size: 50
37     backend: LMDB
38   }
39 }
```

En realidad, se especifican dos capas de entrada, una para la etapa de entrenamiento y otra para las etapas de prueba que se realizan periódicamente. Los puntos que resaltar aquí son los siguientes.

En la línea 13 se especifica la imagen promedio del conjunto de datos de entrenamiento usado para entrenar la red, es decir la imagen promedio de ImageNet. Dicha imagen es usada para normalizar el conjunto de datos de entrada para minimizar que variaciones, por ejemplo, de luminosidad o de contraste afecten el proceso de entrenamiento y prueba.

En las líneas 15 a 19 se especifican los datos de entrada. Para el entrenamiento original de la CaffeNet sobre ImageNet, las imágenes se encontraban en una base de datos LMDB (Lightning Memory-Mapped Database).

La única diferencia entre la capa de entrada de entrenamiento y de prueba es el conjunto de datos de entrada (líneas 15-19 y 34-38).

La nueva especificación de dichas capas para el proceso de calibración fina se muestra a continuación.

```

1 name: "TomatoDiseasesCaffeNet"
2 layer {
3   name: "data"
4   type: "ImageData"
5   top: "data"
6   top: "label"
7   include {
8     phase: TRAIN
9   }
10  transform_param {
11    mirror: true
```

```

12   crop_size: 227
13   mean_file: "/home/edgar/git/PlantDiseaseDetection/deepLearn-
14 ing/data/ilsvrc12/imagenet_mean.binaryproto"
15 }
16 image_data_param {
17   source: "/home/edgar/git/PlantDiseaseDetection/data/train.txt"
18   batch_size: 100
19   new_height: 256
20   new_width: 256
21 }
22 }
23 layer {
24   name: "data"
25   type: "ImageData"
26   top: "data"
27   top: "label"
28   include {
29     phase: TEST
30   }
31   transform_param {
32     mirror: false
33     crop_size: 227
34     mean_file: "/home/edgar/git/PlantDiseaseDetection/deepLearn-
35 ing/data/ilsvrc12/imagenet_mean.binaryproto"
36   }
37   image_data_param {
38     source: "/home/edgar/git/PlantDiseaseDetection/data/test.txt"
39     batch_size: 100
40     new_height: 256
41     new_width: 256
42   }
43 }
```

En la capa de salida de la red es donde se encuentra el clasificador. La especificación original, que se encuentra en el archivo train_val.prototxt se muestra a continuación.

```

1 layer {
2   name: "fc8"
3   type: "InnerProduct"
4   bottom: "fc7"
5   top: "fc8"
6   param {
7     lr_mult: 1
8     decay_mult: 1
9   }
10  param {
11    lr_mult: 2
12    decay_mult: 0
```

```

13 }
14 inner_product_param {
15   num_output: 1000
16   weight_filler {
17     type: "gaussian"
18     std: 0.01
19   }
20   bias_filler {
21     type: "constant"
22     value: 0
23   }
24 }
25 }
```

De esta capa debe modificarse el nombre (línea 2), para que al iniciar el proceso de calibración fina, esta capa sea ignorada y se comience su entrenamiento desde cero. En consecuencia, también deben realizarse las modificaciones correspondientes en otras capas que hagan referencia a esta, que son las capas posteriores que miden la precisión y calculan la función de pérdida.

En las líneas 7 y 11 se indica un coeficiente por el que se multiplica el coeficiente de aprendizaje para esta capa en particular. Dado que el entrenamiento de esta capa inicia desde cero, entonces es conveniente que el coeficiente de aprendizaje sea más grande que el especificado en el archivo solvert.prototxt.

Por último, en la línea 15 se indica el número de salidas que esta capa tiene. Originalmente se especifican mil salidas, ya que son el número de clases en que se dividen las clases de ImageNet. En este proyecto el número de clases, y por tanto el número de salidas de esta capa, debe ser diez.

Es así como la nueva especificación de esta capa, con los cambios correspondientes, se muestra a continuación. También se muestran los cambios en las capas que miden la precisión y la pérdida.

```

1 layer {
2   name: "fc8_tomatoDisease"
3   type: "InnerProduct"
4   bottom: "fc7"
5   top: "fc8_tomatoDisease"
6   param {
7     lr_mult: 10
8     decay_mult: 1
9   }
```

```

10 param {
11   lr_mult: 20
12   decay_mult: 0
13 }
14 inner_product_param {
15   num_output: 10
16   weight_filler {
17     type: "gaussian"
18     std: 0.01
19   }
20   bias_filler {
21     type: "constant"
22     value: 0
23   }
24 }
25 }
26 layer {
27   name: "accuracy"
28   type: "Accuracy"
29   bottom: "fc8_tomatoDisease"
30   bottom: "label"
31   top: "accuracy"
32   include {
33     phase: TEST
34   }
35 }
36 layer {
37   name: "loss"
38   type: "SoftmaxWithLoss"
39   bottom: "fc8_tomatoDisease"
40   bottom: "label"
41   top: "loss"
42 }
43

```

Ahora ya se cuenta con los elementos necesarios para comenzar con el proceso de entrenamiento de la CaffeNet. El comando para entrenar, teniendo una consola direccionada en el directorio raíz de Caffe, es el siguiente.

```
./build/tools/caffe train -solver /ruta/al/archivo/solver.prototxt
-weights /ruta/al/archivo/bvlc_reference_caffenet.caffemodel
```

En 3614 iteraciones, la red alcanzó una precisión de alrededor de 98% durante la etapa de pruebas. Esto tardo cerca de 60 horas de cómputo, en una computadora con las prestaciones descritas en la sección de diseño. La red se entrenó durante 72 horas más alcanzando una precisión mayor al 99%, en 8097 iteraciones. En ese punto

se detuvo el proceso de entrenamiento ya que, analizando los resultados de la precisión para pruebas anteriores, está dejó de mostrar incrementos significativos.

5.2.2 Calibración fina de la GoogLeNet.

El contenido del archivo solver.prototxt original de la GoogLeNet se muestra a continuación:

```
1 net: "models/bvlc_googlenet/train_val.prototxt"
2 test_iter: 1000
3 test_interval: 4000
4 test_INITIALIZATION: false
5 display: 40
6 average_loss: 40
7 base_lr: 0.01
8 lr_policy: "step"
9 stepsize: 320000
10 gamma: 0.96
11 max_iter: 10000000
12 momentum: 0.9
13 weight_decay: 0.0002
14 snapshot: 40000
15 snapshot_prefix: "models/bvlc_googlenet/bvlc_googlenet"
16 solver_mode: GPU
```

Analizando el valor de ciertos parámetros, como el número máximo de iteraciones, el intervalo de generación del *snapshot* o el *stepsize*; puede concluirse que esta red requiere de un mayor número de iteraciones para ser entrenada, en comparación con la CaffeNet.

El contenido del archivo con las modificaciones para el proceso de calibración fina se muestra a continuación.

```
net: "/home/edgar/git/PlantDiseaseDetection/deepLearning/models/fine-
tune_tomato_diseases/train_val.prototxt"
test_iter: 100
test_interval: 1000
test_INITIALIZATION: false
display: 40
average_loss: 40
base_lr: 0.001
lr_policy: "step"
stepsize: 64000
```

```

gamma: 0.96
max_iter: 10000000
momentum: 0.9
weight_decay: 0.0002
snapshot: 40000
snapshot_prefix: "/home/edgar/git/PlantDiseaseDetection/deepLearning/models/finetune_tomato_diseases/caffeModel/finetune_tomato_diseases"

```

Los cambios realizados son de acuerdo a lo expuesto en la sección 5.2.1.

Al igual que en la CaffeNet, la capa de entrada especificada en el archivo train_val.prototxt debe modificarse para indicar la ubicación de los datos con que se van a realizarse las etapas de entrenamiento y pruebas. Una diferencia de la GoogLeNet con la CaffeNet en esta capa es que en lugar de usar una imagen promedio para normalizar las imágenes, se ocupa el pixel promedio. Esto se observa a continuación, en la especificación original de la capa de entrada.

```

1 name: "TomatoDiseasesGoogleNet"
2 layer {
3   name: "data"
4   type: "ImageData"
5   top: "data"
6   top: "label"
7   include {
8     phase: TRAIN
9   }
10  transform_param {
11    mirror: true
12    crop_size: 224
13    mean_value: 104
14    mean_value: 117
15    mean_value: 123
16  }
17  image_data_param {
18    source: "/home/edgar/git/PlantDiseaseDetection/data/train.txt"
19    batch_size: 50
20    new_height: 256
21    new_width: 256
22  }
23 }
24 layer {
25   name: "data"
26   type: "ImageData"
27   top: "data"
28   top: "label"
29   include {
30     phase: TEST

```

```

31    }
32    transform_param {
33      mirror: false
34      crop_size: 224
35      mean_value: 104
36      mean_value: 117
37      mean_value: 123
38    }
39    image_data_param {
40      source: "/home/edgar/git/PlantDiseaseDetection/data/test.txt"
41      batch_size: 50
42      new_height: 256
43      new_width: 256
44    }
45 }

```

En las líneas 13-15 y 35-37 es donde se especifica el valor del pixel promedio de todo el conjunto de datos de ImageNet.

También, a diferencia de la CaffeNet, la GoogLeNet tiene tres capas de clasificación en lugar de solo una. Entonces, la especificación de estas tres capas debe modificarse en el archivo train_val.prototxt, en el mismo sentido en que se modificó la capa de clasificación de la CaffeNet. La especificación original de estas capas se muestra a continuación.

```

1 layer {
2   name: "loss1/classifier"
3   type: "InnerProduct"
4   bottom: "loss1/fc"
5   top: "loss1/classifier"
6   param {
7     lr_mult: 1
8     decay_mult: 1
9   }
10  param {
11    lr_mult: 2
12    decay_mult: 0
13  }
14  inner_product_param {
15    num_output: 1000
16    weight_filler {
17      type: "xavier"
18    }
19    bias_filler {
20      type: "constant"
21      value: 0
22    }

```

```
23  }
24 }
25 ...
26 layer {
27   name: "loss2/classifier"
28   type: "InnerProduct"
29   bottom: "loss2/fc"
30   top: "loss2/classifier"
31   param {
32     lr_mult: 1
33     decay_mult: 1
34   }
35   param {
36     lr_mult: 2
37     decay_mult: 0
38   }
39   inner_product_param {
40     num_output: 1000
41     weight_filler {
42       type: "xavier"
43     }
44     bias_filler {
45       type: "constant"
46       value: 0
47     }
48   }
49 }
50 ...
51 layer {
52   name: "loss3/classifier"
53   type: "InnerProduct"
54   bottom: "pool5/7x7_s1"
55   top: "loss3/classifier"
56   param {
57     lr_mult: 1
58     decay_mult: 1
59   }
60   param {
61     lr_mult: 2
62     decay_mult: 0
63   }
64   inner_product_param {
65     num_output: 1000
66     weight_filler {
67       type: "xavier"
68     }
69     bias_filler {
70       type: "constant"
71       value: 0
72     }
73 }
```

74 }

La nueva especificación de esta capa, con los cambios correspondientes, se muestra a continuación. También se modificaron las referencias a estas capas en las capas de precisión y pérdida.

```

1 layer {
2   name: "loss1/classifier_tomatoDiseases"
3   type: "InnerProduct"
4   bottom: "loss1/fc"
5   top: "loss1/classifier_tomatoDiseases"
6   param {
7     lr_mult: 10
8     decay_mult: 1
9   }
10  param {
11    lr_mult: 20
12    decay_mult: 0
13  }
14  inner_product_param {
15    num_output: 10
16    weight_filler {
17      type: "xavier"
18    }
19    bias_filler {
20      type: "constant"
21      value: 0
22    }
23  }
24 }
25 ...
26 layer {
27   name: "loss2/classifier_tomatoDiseases"
28   type: "InnerProduct"
29   bottom: "loss2/fc"
30   top: "loss2/classifier_tomatoDiseases"
31   param {
32     lr_mult: 10
33     decay_mult: 1
34   }
35   param {
36     lr_mult: 20
37     decay_mult: 0
38   }
39   inner_product_param {
40     num_output: 10
41     weight_filler {
42       type: "xavier"

```

```

43      }
44      bias_filler {
45          type: "constant"
46          value: 0
47      }
48  }
49 }
50 ...
51 layer {
52     name: "loss3/classifier_tomatoDiseases"
53     type: "InnerProduct"
54     bottom: "pool5/7x7_s1"
55     top: "loss3/classifier_tomatoDiseases"
56     param {
57         lr_mult: 10
58         decay_mult: 1
59     }
60     param {
61         lr_mult: 20
62         decay_mult: 0
63     }
64     inner_product_param {
65         num_output: 10
66         weight_filler {
67             type: "xavier"
68         }
69         bias_filler {
70             type: "constant"
71             value: 0
72         }
73     }
74 }
```

El comando para entrenar la red es el mismo que en la CaffeNet, modificando el direccionamiento al archivo prototxt y caffemodel correspondiente:

```
./build/tools/caffe train -solver /ruta/al/archivo/solver.prototxt
-weights /ruta/al/archivo/bvlc_googlenet.caffemodel
```

Tal como se imaginó en un principio, esta red requiere de un tiempo considerablemente mayor para ser entrenada. En una semana supero apenas las mil iteraciones alcanzando un 96% de precisión. Se entrenó la red por una semana más y se obtuvo una precisión del 97%. Debido al tiempo excesivo que estaba tomando el proceso de entrenamiento y considerando que la CaffeNet tuvo una eficiencia superior al 98%, se decidió interrumpir el proceso y seleccionar la CaffeNet como el clasificador por usar en las siguientes etapas del proyecto.

5.3 Realización de predicciones con la CaffeNet, Python y OpenCV.

En la instalación de caffe se incluyen módulos para su uso con Matlab y Python. En el caso de este último, es llamado PyCaffe.

OpenCV 3 también incluye soporte para deep learning con su módulo “dnn”, sin embargo, en las versiones 3.1 y 3.2 dicho módulo se encontraba limitado ya que no era compatible con los frameworks de deep learning más usados, como Caffe, TensorFlow y Torch/PyTorch.

A partir de la versión 3.3 de OpenCV dicho módulo sufrió cambios significativos, siendo ahora compatible con los frameworks mencionados.

Usando el módulo “dnn” es posible cargar el modelo caffe (*.caffemodel) y realizar predicciones de imágenes. Lo anterior fue lo que se realizó con imágenes del subconjunto de pruebas, para verificar la eficiencia del clasificador, que en teoría debe ser la misma que se calculó durante la etapa de entrenamiento.

El script para realizar la predicción de una imagen, usando el módulo dnn de OpenCV y Python es el siguiente [14].

```
1 # USAGE
2 # python deep_learning_with_opencv.py --image images/Bact.Sp_01.JPG --
3 prototxt deploy.prototxt --model finetune_tomato_disease_iter_8097.caffemodel --labels synset_words.txt
4
5
6 # import the necessary packages
7 import numpy as np
8 import argparse
9 import time
10 import cv2
11
12 # construct the argument parse and parse the arguments
13 ap = argparse.ArgumentParser()
14 ap.add_argument("-i", "--image", required=True,
15                 help="path to input image")
16 ap.add_argument("-p", "--prototxt", required=True,
17                 help="path to Caffe 'deploy' prototxt file")
18 ap.add_argument("-m", "--model", required=True,
```

```

19         help="path to Caffe pre-trained model")
20 ap.add_argument("-l", "--labels", required=True,
21                 help="path to ImageNet labels (i.e., syn-sets)")
22 args = vars(ap.parse_args())
23
24 # load the input image from disk
25 image = cv2.imread(args["image"])
26
27 # load the class labels from disk
28 rows = open(args["labels"]).read().strip().split("\n")
29 classes = [r[r.find(" ") + 1:][0] for r in rows]
30
31 # our CNN requires fixed spatial dimensions for our input image(s)
32 # so we need to ensure it is resized to 224x224 pixels while
33 # performing mean subtraction (104, 117, 123) to normalize the input;
34 # after executing this command our "blob" now has the shape:
35 # (1, 3, 224, 224)
36 blob = cv2.dnn.blobFromImage(image, 1, (227, 227), (104, 117, 123) ,
37 False)
38
39 # load our serialized model from disk
40 print("[INFO] loading model...")
41 net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
42
43 # set the blob as input to the network and perform a forward-pass to
44 # obtain our output classification
45 net.setInput(blob)
46 start = time.time()
47 preds = net.forward()
48 end = time.time()
49 print("[INFO] classification took {:.5} seconds".format(end - start))
50
51 # sort the indexes of the probabilities in descending order (higher
52 # probability first) and grab the top-5 predictions
53 idxs = np.argsort(preds[0])[:-1][:-5]
54
55 # loop over the top-5 predictions and display them
56 for (i, idx) in enumerate(idxs):
57     # draw the top prediction on the input image
58     if i == 0:
59         text = "Label: {}, {:.2f}%".format(classes[idx],
60                                         preds[0][idx] * 100)
61         cv2.putText(image, text, (5, 25), cv2.FONT_HERSHEY_SIMPLEX,
62                     0.7, (0, 0, 255), 2)
63
64     # display the predicted label + associated probability to the
65     # console
66     print("[INFO] {} . label: {}, probability: {:.5}".format(i + 1,
67                                                               classes[idx], preds[0][idx]))

```

```
# display the output image
cv2.imshow("Image", image)
cv2.waitKey(0)
```

El programa anterior requiere de cuatro datos de entrada; la imagen a clasificar, la definición del modelo (*.prototxt), el modelo (*.caffemodel) y un archivo con los nombres de cada clase (synset_words.txt).

La definición del modelo es un archivo derivado del archivo train_val.prototxt pero con ciertas modificaciones para el uso de la red en un ambiente de producción. A este archivo se le suele llamar deploy.prototxt. Para generarlo, debe hacerse las siguientes modificaciones en el archivo train_val.prototxt [15].

1. Eliminar las capas de entrada (entrenamiento y pruebas) usadas para el proceso de entrenamiento.
2. Eliminar cualquier capa que requiera de datos etiquetados (en este caso las capas que miden la precisión y la pérdida).
3. Ajustar la red para aceptar datos. Al inicio del archivo, después del nombre de la red escribir:

```
input: "data"
input_dim: 1
input_dim: 3
input_dim: 224
input_dim: 224
```

4. Al final del archivo especificar una capa que mida la probabilidad de que la predicción sea correcta.

```
layer {
    name: "prob"
    type: "Softmax"
    bottom: "fc8_tomatoDisease"
    top: "prob"
}
```

Ahora, el archivo con los nombres de las clases tiene el objetivo de proporcionar un resultado más comprensible para el usuario, para que en lugar de que el resultado sea “La imagen pertenece a la clase 4”, se mencione el nombre de la clase: “La imagen pertenece a la clase Hoja sana”. El contenido del archivo synset_words.txt es el siguiente.

```
0 Spider_mites
1 Septoria_Leaf_Spot
2 Bacterial_Spot
3 Yellow_Leaf_Curl_Virus
4 Healty
5 Mosaic_virus
6 Target_Spot
7 Early_Blight
8 Late_Blight
9 Leaf_Mold
```

Proporcionando los cuatro datos de entrada, el programa carga el modelo, la imagen, las etiquetas y realiza la predicción. Como resultado se muestra la imagen en una ventana, y sobre la imagen aparece escrito la clase a la que pertenece, que es aquella con la mayor probabilidad de clasificación. Además, en consola se muestran las cinco clases con mayor probabilidad de clasificación. En las figuras 5.2, 5.3 y 5.4 se muestran los resultados de clasificación.

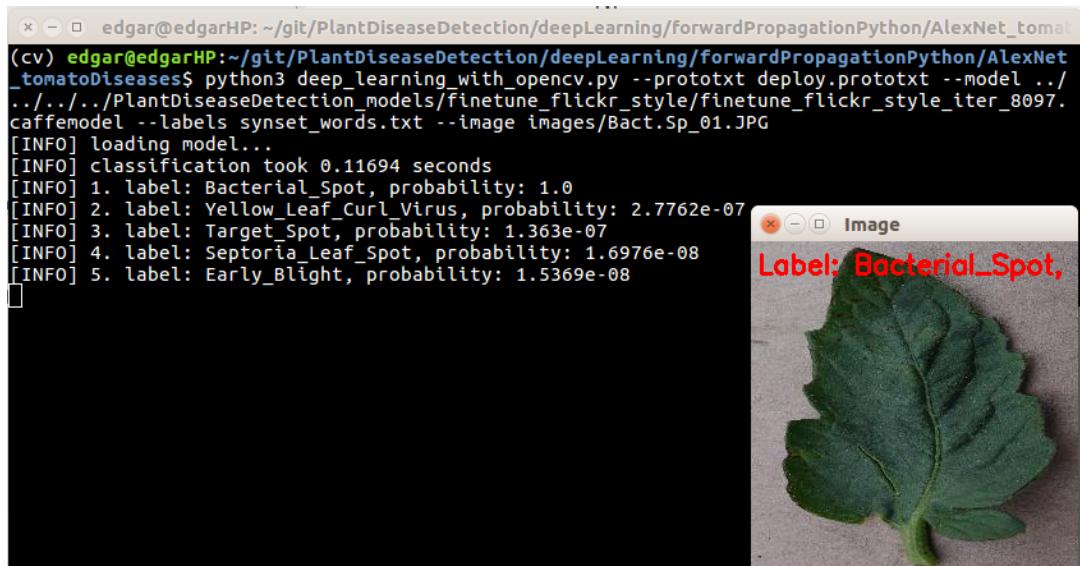


Figura 5.2 Predicción de una hoja sana.

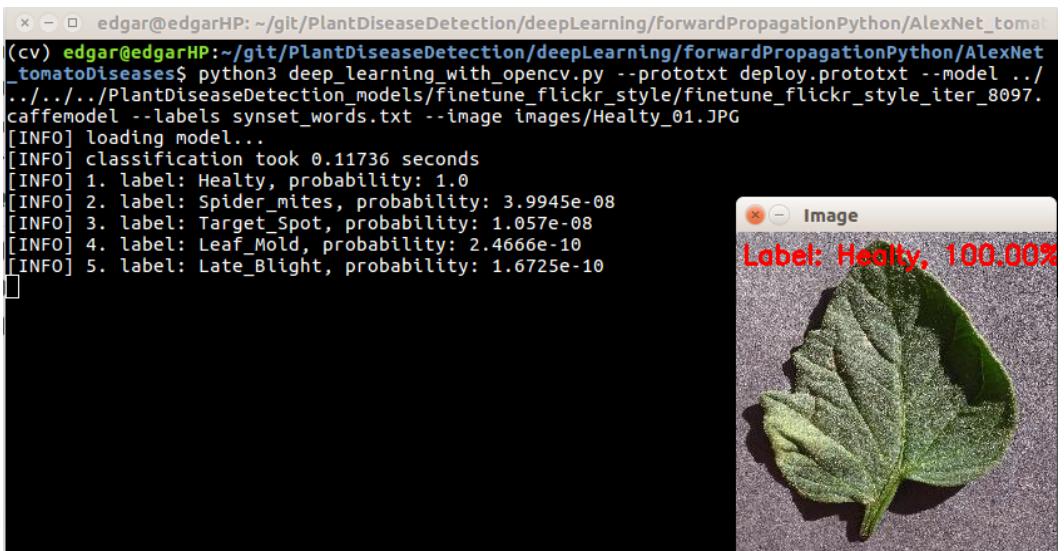


Figura 5.3 Predicción de una hoja sana.

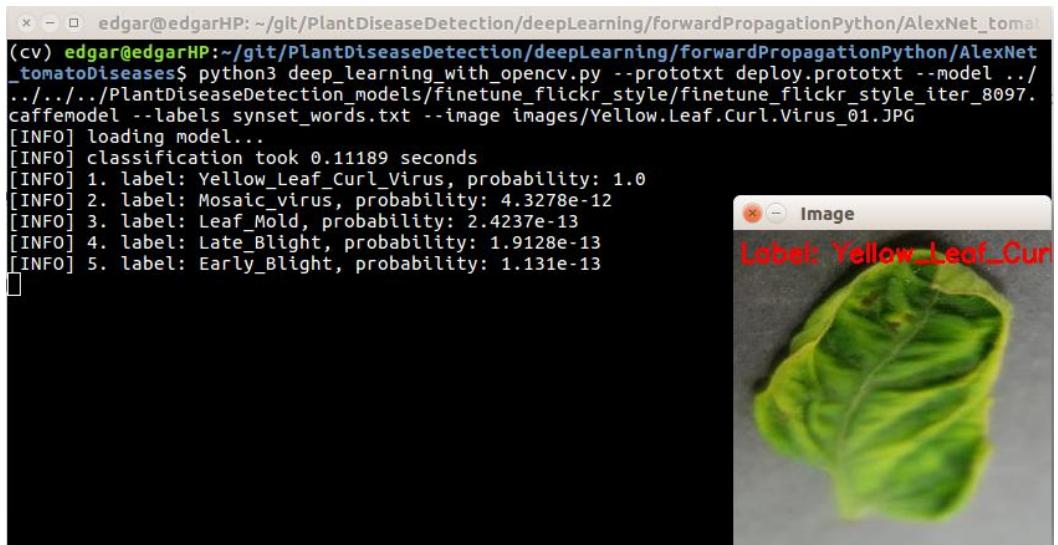


Figura 5.4 Predicción de una hoja con el virus del rizado amarillo del tomate.

El script anterior fue modificado para poder medir la eficiencia de clasificación sobre todo el subconjunto de imágenes de prueba (3637 imágenes). Como resultado se obtuvo una eficiencia de clasificación del 98.82% (Ver figura 5.5).

```
(cv) edgar@edgarHP:~/git/PlantDiseaseDetection$ python3 predictions.py --prototxt deeplearning/models/finetune_flickr_style/deploy.prototxt --model deepLearning/models/finetune_flickr_style/caffeModel/finetune_flickr_style_iter_8097.caffemodel --labels deepLearning/models/finetune_flickr_style/synset_words.txt --images data/test.txt
[INFO] loading model...
Progress: 100.00%
[INFO] Eficiency: 98.82%
(cv) edgar@edgarHP:~/git/PlantDiseaseDetection$
```

Figura 5.5 Eficiencia del clasificador

De esta forma ya se cuenta con el clasificador listo para realizar predicciones usando OpenCV y, en este caso particular, Python, sin embargo, el proceso de predicción puede realizarse también con alguno de los otros lenguajes de programación compatibles con OpenCV, como Java y C++. Con esto se concluye el primer incremento de la metodología propuesta. De esta forma, en los capítulos siguientes se describe el análisis, diseño e implementación del segundo incremento del proyecto, que consiste en el sistema final con el que el usuario interactuará para obtener el diagnóstico de acuerdo a la imagen que proporcione.

Capítulo 6.

Análisis del sistema Web.

En esta sección se describen los requisitos de hardware y software, se presenta un modelo de caso de usos y la especificación de la interfaz gráfica, correspondientes al segundo incremento de la metodología propuesta en la sección 1.4.

6.1 Requisitos del sistema.

A continuación, se presentan los requisitos funcionales y no funcionales de este incremento. Considerando que de cierta forma la funcionalidad de este incremento incluye al del primer incremento, aquí se incluyen los requisitos enunciados en la sección 3.1 más los propios de este incremento.

6.1.1 Requisitos de software.

Requisitos funcionales.

RF01. Selección de imagen.

Nivel de madurez: Alta.

Prioridad: Media.

Descripción: El sistema permitirá seleccionar una imagen del sistema de archivos local del dispositivo del usuario.

RF02. Clasificación.

Nivel de madurez: Alta.

Prioridad: Alta.

Descripción: El sistema clasificará la imagen en alguna de las nueve enfermedades descritas en la tabla 1.1 o en la clase “Hoja sana”, indicando la probabilidad de que esa sea la enfermedad correcta.

Requisitos no funcionales.

RNF01. La eficiencia de clasificación deberá ser superior al 90%.

RNF02. El clasificador identificará la clase de la imagen de entrada en un tiempo no mayor a cinco segundos.

RNF05. El sistema será desarrollado para un ambiente Web.

RNF06. El sistema podrá usarse a través del navegador Chrome v.64.

6.1.2 Requisitos de hardware.

En este incremento ya se tiene el clasificador entrenado. Con el modelo entrenado solo se requiere realizar una propagación hacia adelante en la red para realizar las predicciones, tarea sencilla que se ejecuta en menos de un segundo, de acuerdo con las evidencias presentadas en la sección 5.3. Por lo que se tienen los siguientes requisitos de hardware, tomando como referencia las especificaciones de las instancias gratuitas de Amazon Web Services, en las que se instalará el prototipo.

- 1 GB de memoria RAM.
- 1 CPU Intel Xeon de alta frecuencia.
- 30 GB de espacio en disco.

6.2 Modelo de casos de uso.

Desde el punto de vista del usuario, la aplicación le permitirá hacer las acciones siguientes: Seleccionar una imagen desde su dispositivo y obtener el diagnóstico de enfermedad de la planta contenida en la imagen. Esto se ilustra en el diagrama de casos de uso de la figura 6.1.

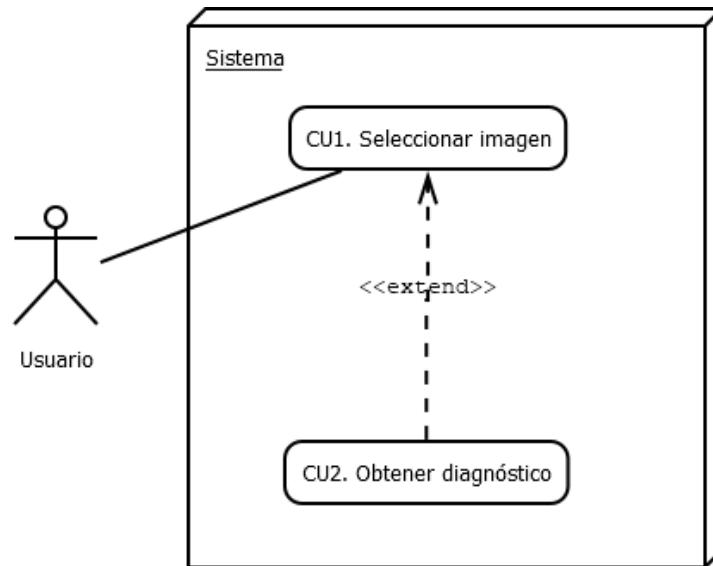


Figura 6.1 Diagrama de casos de uso del prototipo.

6.2.1 CU1. Seleccionar imagen.

Resumen.

El usuario accederá a la interfaz del sistema y seleccionará la imagen de la hoja con la enfermedad que quiere identificar.

Descripción.

Caso de uso:	CU1. Seleccionar imagen
Versión	1.0
Actor	Usuario.
Propósito	Seleccionar la imagen con la hoja enferma.
Entradas	Ninguna.
Origen	IU1.0 Pantalla de inicio.
Salidas	Imagen seleccionada
Destino	IU1.0 Pantalla de inicio.
Precondiciones	Dar click en el botón “Buscar”.
Postcondiciones	Imagen seleccionada y lista para enviar.
Errores	Extensión incorrecta del archivo seleccionado.
Reglas de negocio	Solo son admitidas las extensiones jpg, png.
Tipo	Caso de uso primario.

Observaciones	Solo se selecciona la imagen, aún no se envía al servidor para clasificación.
Autor	Edgar Rodrigo Arredondo Basurto.
Estatus	Versión 1.0 Revisada.

Trayectorias del caso de uso.

Trayectoria principal.

- 1  Da click en el botón “Buscar”.
 - 2  Selecciona un archivo en la ventana de dialogo. [Trayectoria A].
- *Fin del caso de uso.*

Trayectoria alternativa A.

Condición: El usuario selecciono un archivo con una extensión distinta a jpg o png.

- 1  Muestra el mensaje “Seleccionar un archivo válido (*.jpg, *.png)”.
 - 2  Termina el caso de uso.
- *Fin de la trayectoria.*

6.2.1 CU2. Obtener diagnóstico.

Resumen.

El usuario enviará la imagen seleccionada y obtendrá como resultado la clasificación de esta en alguna de las nueve categorías mostradas en la tabla 1.1 o en la categoría hoja sana.

Descripción.

Caso de uso:	CU2. Obtener diagnostico
Versión	1.0
Actor	Usuario.
Propósito	Obtener la clasificación de la imagen con la hoja enferma.
Entradas	La imagen por clasificar.
Origen	IU1.0 Pantalla de inicio.

Salidas	Los dos diagnósticos más probables y su probabilidad.
Destino	IU1.0 Pantalla de inicio.
Precondiciones	Dar click en el botón “Enviar”.
Postcondiciones	Ninguna.
Errores	Ninguno.
Reglas de negocio	Solo se puede clasificar en alguna de las nueve categorías listadas en la tabla 3.1 más la categoría “Hoja sana”
Tipo	Caso de uso primario.
Observaciones	Ninguna.
Autor	Edgar Rodrigo Arredondo Basurto.
Estatus	Versión 1.0 Revisada.

Trayectorias del caso de uso.

Trayectoria principal.

- 1  Da click en el botón “Enviar”.
- 2  Muestra como resultado de la clasificación las dos clases más probables y su probabilidad.

- - - - - *Fin del caso de uso.*

6.3 Interfaz gráfica.

En el sistema solo es necesaria una pantalla, desde la cual el usuario selecciona la imagen la envía y recibe el diagnóstico.

6.3.1 IU1.0 Pantalla de inicio.

Objetivo.

Desde esta pantalla el usuario puede seleccionar la imagen desde su dispositivo y enviarla para obtener el diagnóstico. El resultado le será informado con un mensaje en esta misma pantalla.

Diseño.

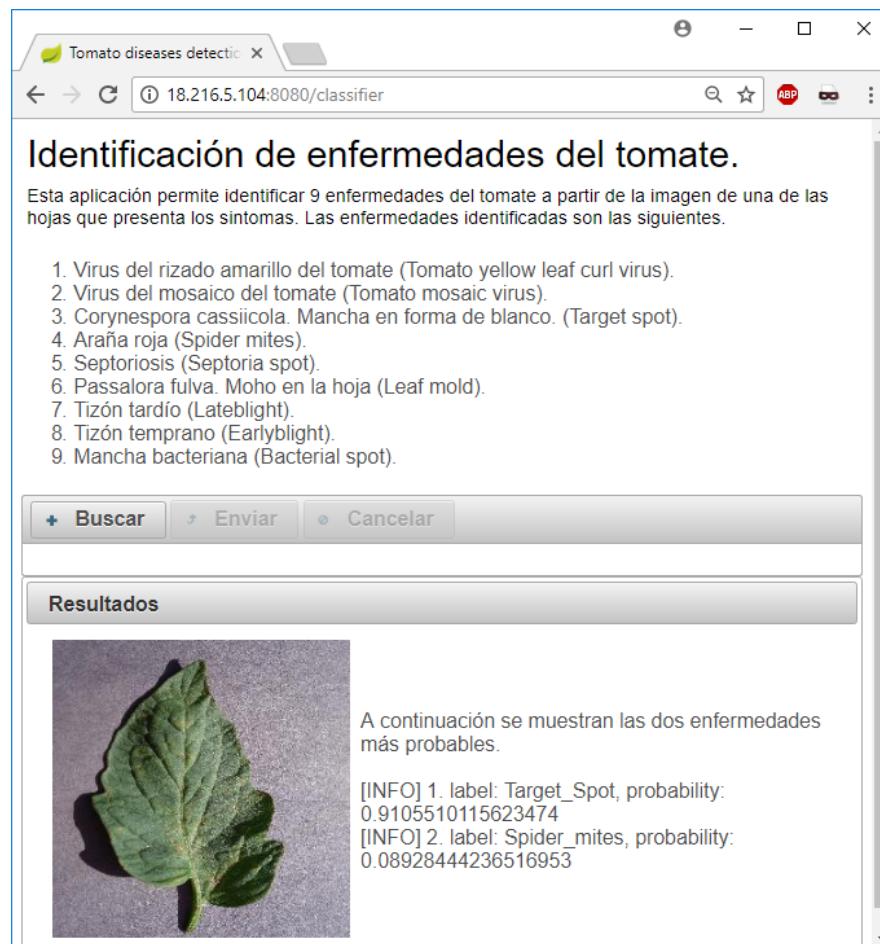


Figura 6.2 IU1.0 Pantalla de inicio.

Capítulo 7.

Diseño del sistema Web.

En esta sección se realiza el diseño del sistema Web, correspondiente al segundo incremento de la metodología propuesta en la sección 1.4.

7.1 Arquitectura del sistema Web.

Un sistema Web es un sistema que puede ser accedido por usuarios a través de un navegador Web. El navegador genera peticiones HTTP a URL's específicas que mapean en recursos ubicados en un servidor Web. El servidor genera y devuelve páginas HTML hacia el cliente, las cuales el navegador puede mostrar. La parte central de un sistema Web es la lógica en el lado del servidor, la cual suele estar diseñada mediante una arquitectura en capas. La arquitectura más común es una arquitectura de tres capas: presentación, negocio y datos. [16].

El prototipo de este proyecto no incluye persistencia de datos, por lo que la capa de datos no será incluida. De esta forma, las capas de la arquitectura del servidor serán las siguientes:

- Capa de presentación. Capa donde se procesan las peticiones del cliente.
- Capa de servicios. Capa a la que la capa de presentación delega la ejecución de procedimientos.
- Capa de negocio. Capa con la lógica y entidades de negocio.

La arquitectura anterior se ilustra en la figura 7.1.

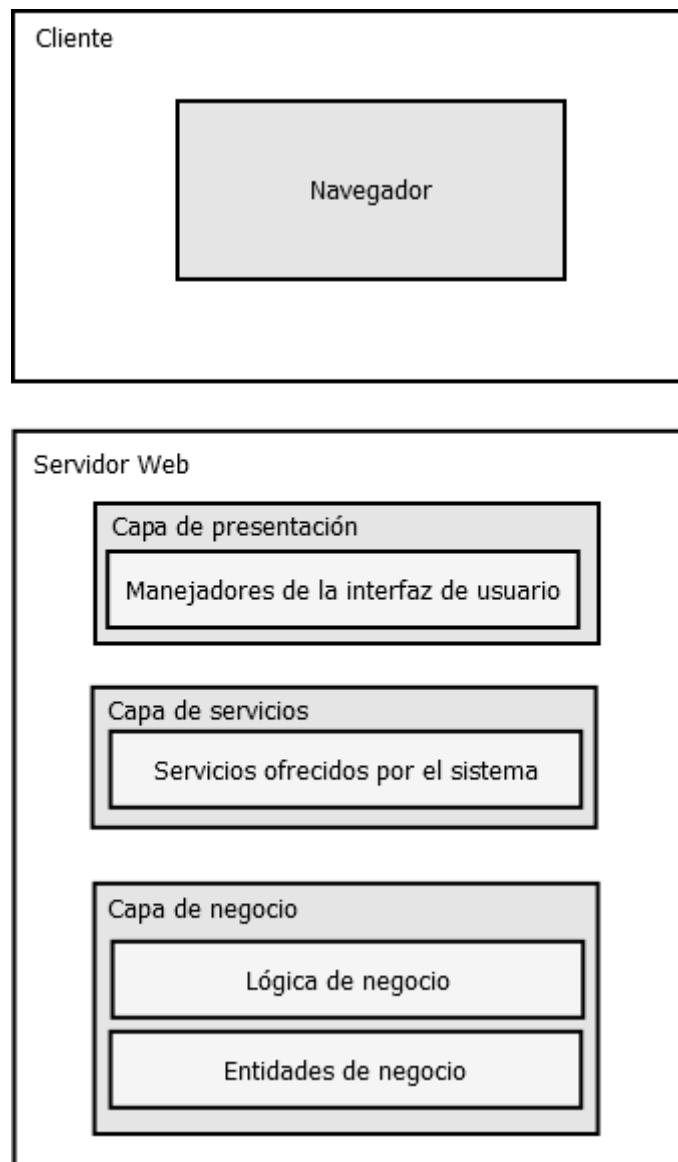


Figura 7.1 Arquitectura del sistema.

7.2 Diagrama de clases.

Con base en la arquitectura anterior, en la figura 7.2 se presenta el diagrama de clases del sistema.

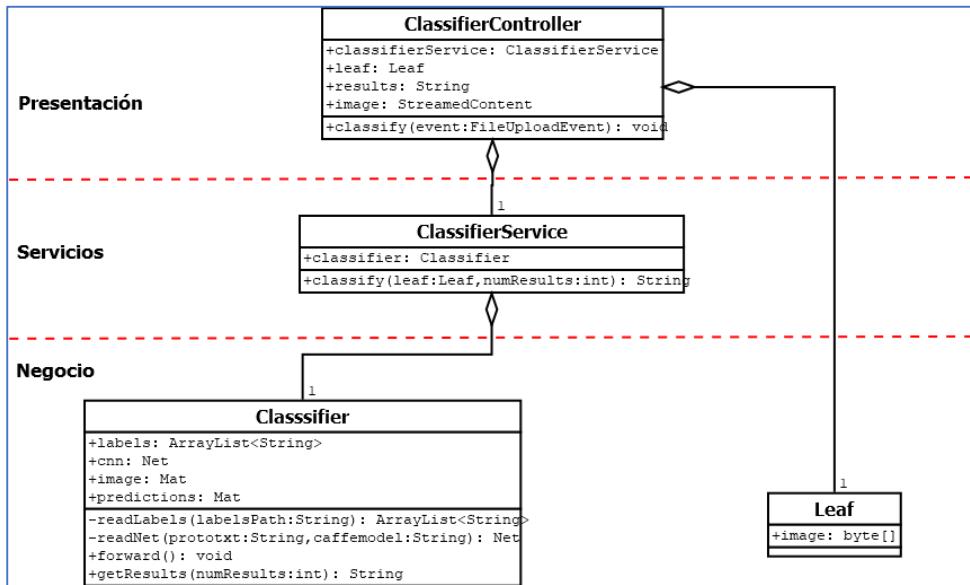


Figura 7.2 Diagrama de clases del sistema.

7.3 Tecnologías de desarrollo.

El sistema Web, en particular la lógica del servidor será implementado en lenguaje Java. Con esto, se pretende facilitar el desarrollo de una eventual aplicación móvil en Android en un trabajo a futuro. Además de java, se usarán las tecnologías siguientes:

- OpenCV. Para la etapa de clasificación.
- Maven y Spring boot. Para el desarrollo y administración de dependencias del proyecto.
- JavaServer Faces (JSF) y PrimeFaces. Para el desarrollo de las interfaces de usuario y la comunicación entre el cliente y la capa de presentación del servidor.

Con las tecnologías anteriores se generará un jar con un servidor web integrado y todas las dependencias necesarias, de forma tal que, para instalar la aplicación en un ambiente de producción, solo sea necesario ejecutar el jar y tener instalado OpenCV con las bibliotecas nativas necesarias.

Capítulo 8.

Implementación del sistema Web.

En esta sección se presentan evidencias de la implementación del sistema Web, de acuerdo con el análisis y diseño presentados en los capítulos 6 y 7.

8.1 Integración de OpenCV con Java.

La instalación de OpenCV descrita en la sección 5.1.1 incluye la generación de la biblioteca para java, llamada opencv-330.jar, ubicada en el directorio opencv-3.3.0/build/bin. Para lo anterior basta con agregar la bandera de cmake `-D BUILD_SHARED_LIBS=OFF`. Si el jar no se encuentra en dicho directorio, hay que asegurarse de que en los resultados de la ejecución de CMake, en la sección de java aparezcan los enlaces a las ubicaciones de Ant y JNI, tal como se muestra en la figura 8.1. En caso contrario verificar que Ant este correctamente instalado y de ser necesario instalarlo con el comando “`sudo apt-get install ant`”. En el caso del JNI verificar que la variable de entorno `JAVA_HOME` exista y este direccionada a la ubicación del jdk.

```
-- Java:  
--   ant:          /usr/bin/ant (ver 1.8.2)  
--   JNI:          /usr/lib/jvm/java-6-openjdk/include /usr/lib/jvm/java-6-  
--   Java tests:  YES
```

Figura 8.1 Configuración de OpenCV para la integración con Java.

También es requerida una biblioteca nativa (*.so en Linux, *.dll en Windows) llamada libopencv_java330.so, la cual se ubica en el directorio opencv-3.3.0/build/lib. Esta biblioteca no puede ser simplemente copiada al servidor ya que incluye dependencias a otras bibliotecas nativas del sistema operativo, que son generadas durante la instalación de OpenCV, por lo que será necesario repetir el proceso de instalación de OpenCV descrito en la sección 5.1.1 en el servidor de producción.

8.2 Integración de OpenCV, Maven y Spring boot en NetBeans.

Para generar el proyecto de Maven con Java y Spring boot se usó la herramienta de spring ubicada en <http://start.spring.io/> (figura 8.2) donde es posible indicar las dependencias iniciales, el grupo, artefacto, nombre y demás metadatos que identifican a un proyecto de Spring.

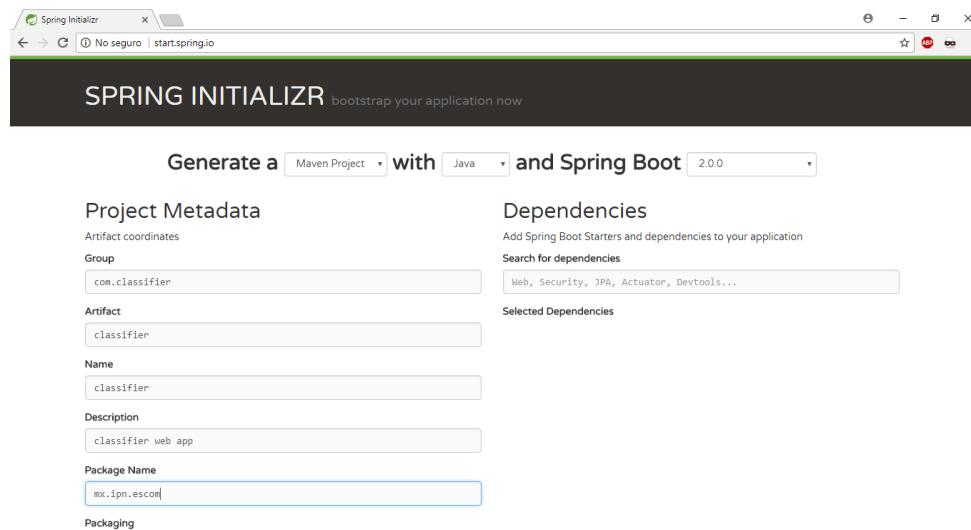


Figura 8.2 Generación del proyecto Maven.

La herramienta anterior genera un archivo comprimido (*.zip) que es posible importar desde el IDE de NetBeans. El archivo desde NetBeans se genera un proyecto con la estructura mostrada en la figura 8.3.

Las dependencias de un proyecto Maven son especificadas en el archivo pom.xml. Maven cuenta con un repositorio central en línea para todas aquellas bibliotecas libres. Para integrarlas en el proyecto solo basta con declararlas en el archivo mencionado. Sin embargo, la biblioteca de OpenCV versión 3.3.0 no se encuentra en el repositorio central. Por esta razón, debe instalarse en el repositorio local de Maven el jar generado durante la instalación de OpenCV. Para esto, desde una consola dirigida en el directorio raíz del proyecto, ejecutar el siguiente comando maven:

```
mvn install:install-file -Dfile=/path/a/opencv/opencv-
3.3.0/build/bin/opencv-330.jar -DgroupId=org.opencv -DartifactId=opencv -
Dversion=3.3.0 -Dpackaging=jar
```

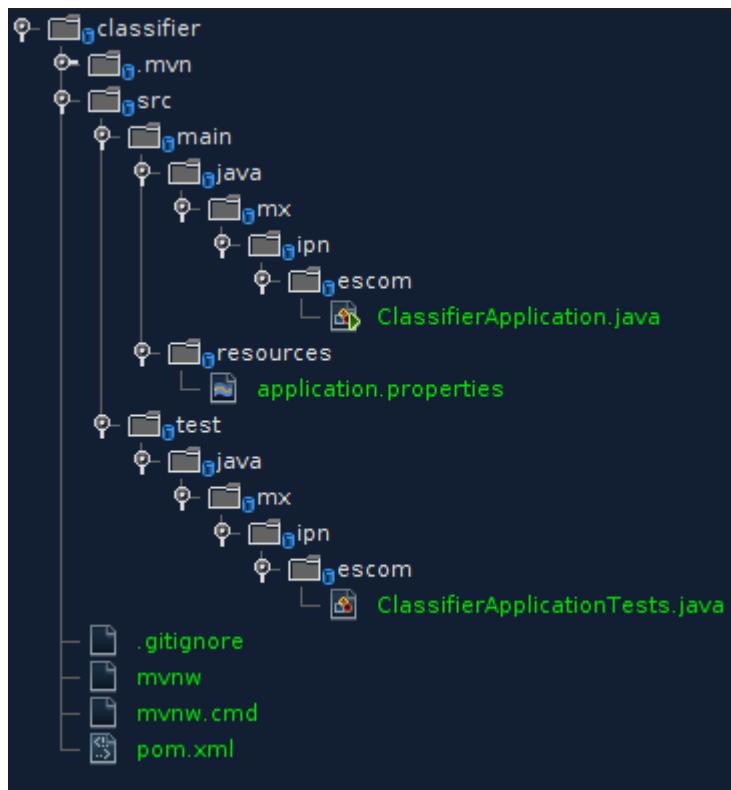


Figura 8.3 Proyecto de Spring Initializr importado en NetBeans.

Ahora ya es posible agregar la dependencia en el archivo pom.xml de la forma siguiente:

```
<dependency>
    <groupId>org.opencv</groupId>
    <artifactId>opencv</artifactId>
    <version>3.3.0</version>
</dependency>
```

Agregando esta biblioteca es posible escribir y compilar código que use la biblioteca de OpenCV. Sin embargo, en tiempo de ejecución también son necesarias referencias a bibliotecas nativas de OpenCV (*.so). Esto se logra agregando la bandera **-Djava.library.path** al comando java, indicando la ubicación de la mencionada biblioteca nativa. En NetBeans se agrega dicha bandera en las propiedades del proyecto, como se muestra en la figura 8.4.

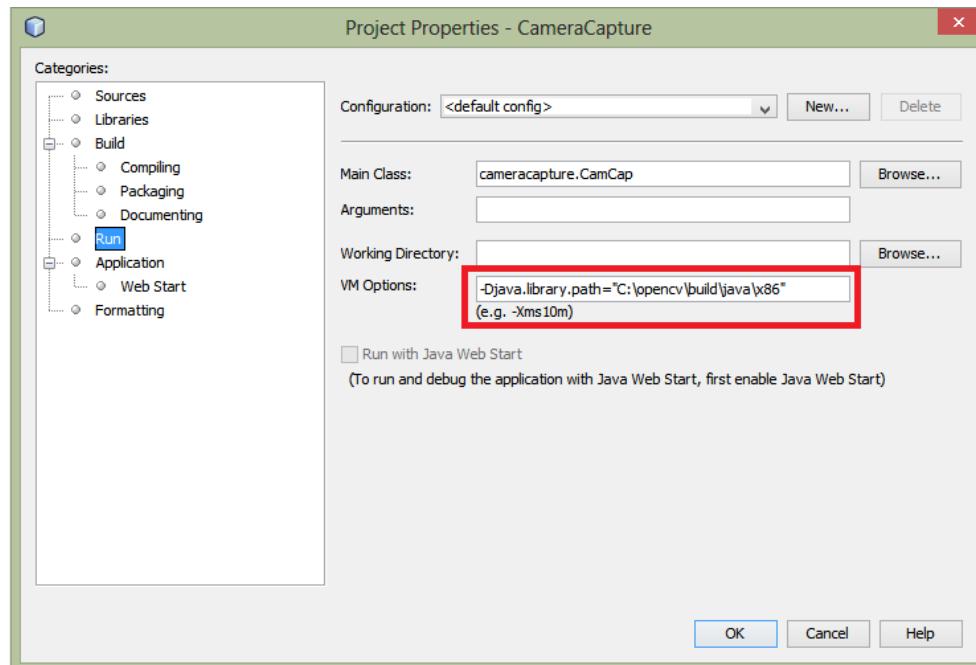


Figura 8.4 Ubicación de la biblioteca nativa de OpenCV.

Por último en la clase principal del proyecto, es decir en el método main de la clase “ClassifierApplication” debe de cargarse la biblioteca nativa de OpenCV. A continuación, se presenta un fragmento de dicha clase en donde se muestra la forma en que se carga la biblioteca (línea 10).

```

1 package mx.ipn.escom;
2
3 org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.opencv.core.Core;
5 /**
6 * @SpringBootApplication
7 public class ClassifierApplication {
8
9     public static void main(String[] args) {
10         System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
11         SpringApplication.run(ClassifierApplication.class, args);
12     }
13     /**
14 }
```

8.3 Integración con JavaServer Faces y PrimeFaces.

En las aplicaciones Web tradicionales las peticiones al servidor se hacen desde funciones JavaScript residentes en la página Web, en el lado del cliente. Estas funciones envían la petición y los datos al servidor y éste devuelve los datos en algún tipo de formato como texto plano, XML o JSON. Las funciones JavaScript procesan la respuesta del servidor y actualizan ciertas partes de la página Web. Este enfoque es llamado AJAX (Asynchronous JavaScript and XML).

Existen frameworks de JavaScript muy populares, como jQuery, que facilitan la tarea de definir la interfaz de usuario de la aplicación. Sin embargo, el mantenimiento y pruebas de este código es complicado, ya que hay muchos elementos implicados, como la comunicación con el servidor, las páginas HTML y el propio código JavaScript. Otra desventaja es que es complicado reutilizar y compartir el código JavaScript entre distintas páginas y desarrolladores.

Como alternativa a la codificación en JavaScript aparecen los frameworks con base en el servidor, en los que los desarrolladores no escriben directamente JavaScript, sino que utilizan componentes de alto nivel (paneles, botones, listas desplegables, etcétera) que el servidor inserta en las páginas resultantes. Este enfoque es que utiliza el framework JavaServer Faces (JSF).

En este enfoque el desarrollador define la interfaz de usuario en un lenguaje de componentes específico. Esta definición reside en el servidor en forma de texto. Cuando el servidor recibe una petición realiza un procesamiento de esta página de texto y genera otra que contiene los componentes de la interfaz en formato HTML y JavaScript y que se envía de vuelta al navegador.

En el caso de JSF 2, la definición de la interfaz se realiza en forma de páginas XHTML con distintos tipos de etiquetas que permiten definir componentes básicos de una interfaz Web como etiquetas, campos de texto, botones, listas, tablas, etcétera [17].

PrimeFaces es un framework de componentes para JSF integrando más de cien componentes. Los componentes de PrimeFaces extienden la funcionalidad de los componentes originales de JSF, incluyendo también componentes adicionales.

El funcionamiento de JSF es el siguiente. El navegador realiza una petición a una determinada URL en la que reside la página JSF que se quiere mostrar. En el servidor, un servlet recibe la petición y construye un árbol de componentes a partir de la página JSF que se solicita. Una vez construido el árbol de componentes se ejecuta el código Java en el servidor para llenar los elementos del árbol con los datos de la aplicación. Por último, a partir del árbol de componentes se genera la página HTML que se envía al navegador. El proceso anterior se muestra de forma esquemática en la figura 8.5.

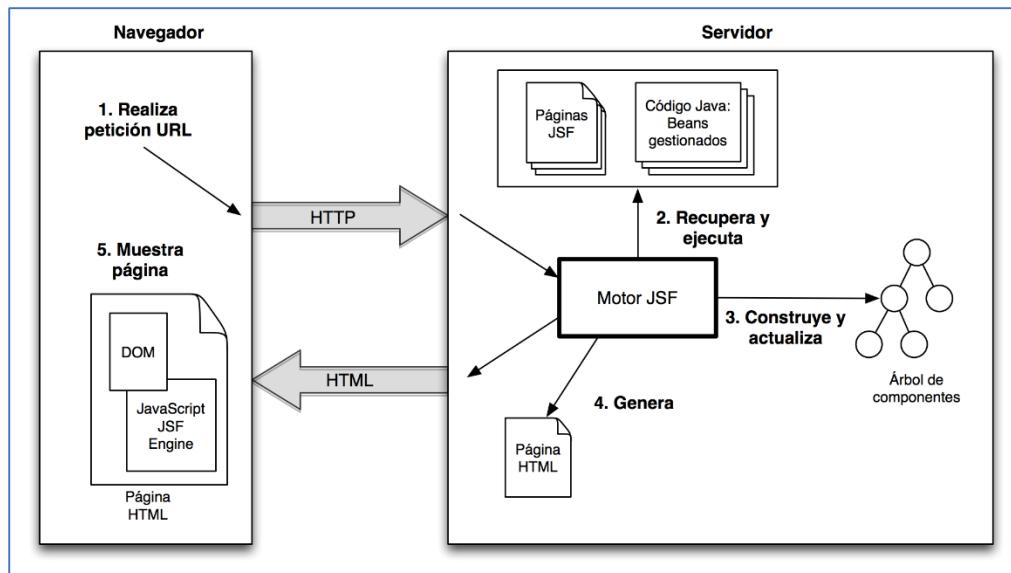


Figura 8.5 Funcionamiento de JSF.

Para integrar JSF con spring boot, el primer paso es agregar las dependencias siguientes en el archivo pom.xml.

```

1 <dependency>
2     <groupId>org.apache.myfaces.core</groupId>
3     <artifactId>myfaces-impl</artifactId>
4     <version>2.2.12</version>
5 </dependency>
6 <dependency>
7     <groupId>org.apache.myfaces.core</groupId>
8     <artifactId>myfaces-api</artifactId>
9     <version>2.2.12</version>
10 </dependency>
11 <dependency>
12     <groupId>org.primefaces</groupId>
13     <artifactId>primefaces</artifactId>
14     <version>6.1</version>

```

15 </dependency>

La configuración de JSF se realiza en dos archivos: web.xml y faces-config.xml. Normalmente, en una aplicación de Spring Boot no es necesario el archivo web.xml, pero en este caso, dado que es necesario integrar JSF, hay que crear el archivo web.xml en el directorio src/main/webapp/WEB-INF/ con el siguiente contenido.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="3.1">
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <listener>
        <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
    </listener>
</web-app>
```

Los dos primeros elementos son los responsables de crear y configurar el FacesServlet. Dicho servlet es el que recibe y resuelve las peticiones a páginas JSF. El elemento servlet-mapping le indica al servlet el tipo de URLs que va a resolver. En este caso son aquellas con la terminación *.jsf. Los dos últimos elementos, son los responsables de integrar JSF dentro del contexto de Spring.

El contenido del archivo faces-config.xml se muestra a continuación.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
    version="2.2">
    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELRe-
solver</el-resolver>
```

```
</application>
</faces-config>
```

Lo que se realiza en el archivo faces-config.xml es registrar un Expression Language resolver (ELResolver) que delega a Spring la responsabilidad de resolver el nombre de los beans a los que se haga referencia en las páginas JSF. Es decir, de esta forma, pueden usarse los beans administrados por Spring en el contexto de JSF.

Por último, debe de actualizarse la clase ClassifierApplication.java con la definición de dos beans más.

```
@Bean
public ServletRegistrationBean servletRegistrationBean() {
    ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(new FacesServlet(), "*.jsf");
    servletRegistrationBean.setLoadOnStartup(1);
    return servletRegistrationBean;
}

@Bean
public FilterRegistrationBean rewriteFilter() {
    FilterRegistrationBean rwFilter = new FilterRegistrationBean(new RewriteFilter());
    rwFilter.setDispatcherTypes(EnumSet.of.DispatcherType.FORWARD,
DispatcherType.REQUEST, DispatcherType.ASYNC, DispatcherType.ERROR));
    rwFilter.addUrlPatterns("/*");
    return rwFilter;
}
```

8.4 Desarrollo de la aplicación.

En acuerdo con la arquitectura del sistema propuesta, la estructura del proyecto se muestra en la figura 8.6.

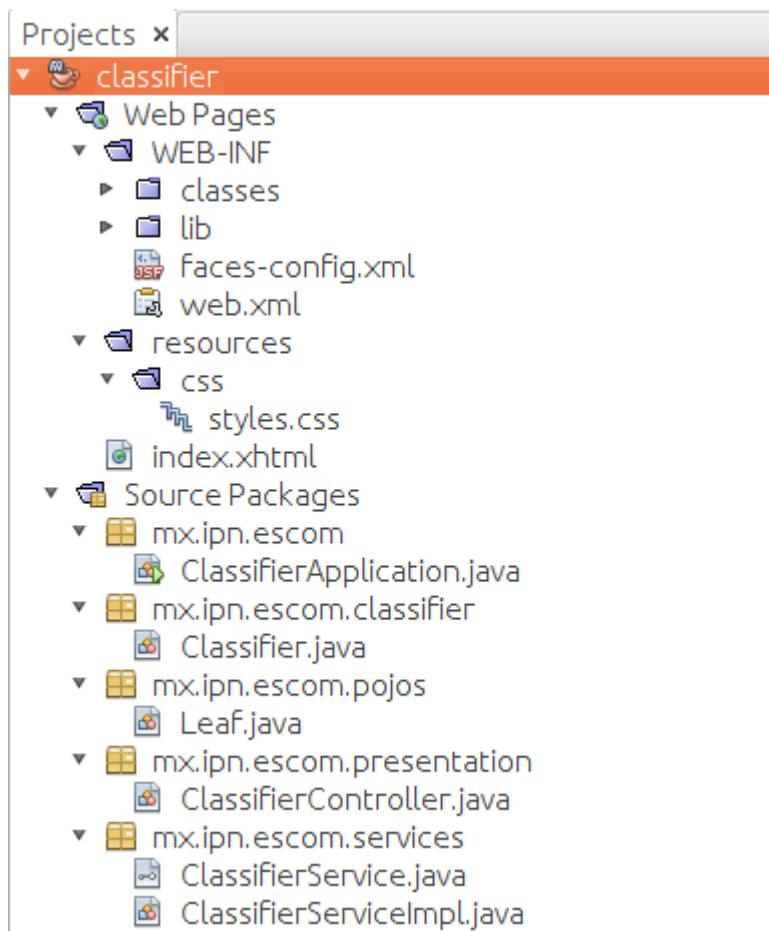


Figura 8.6 Estructura del proyecto.

En el paquete raíz (mx.ipn.escom) se ubica la clase principal ClassifierApplication.java. También se tienen los paquetes “presentation” y “services” que corresponden con la capa de presentación y servicios respectivamente. Finalmente, se tienen dos paquetes más, “classifier” y “pojos”. Ambos paquetes corresponden a la capa de negocio. La página JSF se ubica en el apartado Web pages, donde también se tienen los archivos xml de configuración de JSF y los recursos de la interfaz Web, que en este caso consisten solo de una hoja de estilos (CSS).

El código fuente de cada clase y del archivo pom.xml se presenta en el Anexo A.

Al compilar este proyecto se genera un *.jar, el cual contiene un servidor Tomcat embebido, de forma tal, que al ejecutar el jar se levanta el servidor en el puerto 8080 y la ruta para acceder a la interfaz web es /classifier.

Capítulo 9.

Implantación del sistema Web.

En esta sección se describe la implantación del sistema Web en un servidor de producción.

9.1 Implantación en AWS.

Amazon Web Services (AWS) ofrece una capa gratuita que puede usarse sin costo alguno durante el primer año de servicio. Entre los servicios que se ofrecen en esta capa gratuita se tiene el Amazon Elastic Compute Cloud (Amazon EC2), servicio que proporciona capacidad informática en la nube segura y de tamaño modificable. Está diseñado para facilitar a los desarrolladores el uso de la informática en la nube a escala de la Web [18]. La capa gratuita de AWS incluye 750 horas de instancias t2.micro de Windows y Linux al mes durante el primer año.

Se seleccionó una instancia Ubuntu Server 16.04 LTS (HVM), configurando un espacio en disco de 30 GB, siendo este el máximo gratuito. Esta instancia cuenta con 1GB de RAM. Para un mejor desempeño se creó un archivo SWAP de 4GB. Generalmente SWAP es una partición del disco que se usa como RAM bajo ciertas condiciones, sin embargo, también es posible crear un archivo que opere como SWAP en la misma partición del sistema operativo, siendo esto último lo que se realizó con los siguientes comandos.

```
sudo fallocate -l 4G /swapfile  
sudo chmod 600 /swapfile  
sudo mkswap /swapfile  
sudo swapon /swapfile
```

En esta instancia de Ubuntu Server fue instalado el jdk 1.8, Maven y OpenCV con las bibliotecas para Java. También fue transferido el proyecto para la posterior generación del jar. Otra alternativa es generar el jar y después transferirlo a la

instancia de Ubuntu Server. La instalación de OpenCV es necesaria, ya que además del jar que es agregado a las dependencias del proyecto, también son necesarias ciertas bibliotecas nativas generadas durante la instalación, y cuya ubicación se indica mediante la bandera **-Djava.library.path** del comando java.

También es necesario habilitar en la consola de AWS el tráfico con la instancia a través del puerto 8080. Posteriormente, una vez generado o transferido el jar y teniendo OpenCV instalado en Ubuntu Server, la aplicación se inicia con el comando siguiente.

```
java -Djava.library.path="/home/ubuntu/opencv-3.3.0/build/lib" -jar target/classifier-0.0.1-SNAPSHOT.jar
```

Ahora la aplicación es visible a través de un navegador mediante la URL siguiente: <http://ip-publica-ubuntu-server:8080/classifier>.

Capítulo 10.

Pruebas del sistema Web.

En esta sección se describen las pruebas realizadas al sistema Web y los resultados obtenidos.

10.1 Plan de pruebas.

Las pruebas a continuación planteadas tienen como objetivo verificar el cumplimiento de los requisitos de software descritos en la sección 6.1.1. En la tabla 10.1 se presentan el conjunto de pruebas a realizar.

Número de prueba	Requisito	Descripción de la prueba	Resultado extensión esperado
1	RF01	Seleccionar una imagen del sistema de archivos local del dispositivo.	Es posible seleccionar imágenes con *.jpg, *.png y *.bmp.
2	RF01	Seleccionar un archivo con extensión distinta a las permitidas.	No es posible seleccionar archivos con extensión distinta a las permitidas.
3	RF02	Seleccionar una imagen etiquetada y clasificarla.	El sistema clasifica la imagen en alguna de las nueve enfermedades o en hoja sana.
4	RF02	Seleccionar una imagen que no corresponda con ninguna de las diez clases válidas.	El sistema debe indicar que la imagen no corresponde a ninguna de las enfermedades identificables u a una hoja sana.
5	RNF01	Medir la eficiencia de clasificación con imágenes del conjunto de prueba.	La eficiencia de clasificación es superior al 90% para al menos 10% de imágenes del conjunto de prueba.

6	RF02	Medir el tiempo de respuesta del sistema Web.	El tiempo promedio de clasificación es menor a cinco segundos.
---	------	---	--

Tabla 10.1 Plan de pruebas de la aplicación.

10.2 Resultados.

A continuación, se muestran los resultados de la ejecución de las pruebas descritas en la tabla 10.1.

Número de prueba	Resultado	Observaciones
1	Se permitió la selección de archivos extensiones *.jpg, *.png y *.bmp.	
2	No se permitió la selección de archivos con extensión distinta a las permitidas.	
3	El sistema clasificó la imagen de forma correcta y mostró los dos resultados más probables.	
4	El sistema clasifica algunas de las imágenes en clases válidas.	Se debería agregar al clasificador una clase específica para aquellas imágenes no reconocidas.
5	La prueba de eficiencia (98.82%) fue presentada al final de la sección 5.3.	
6	El tiempo promedio de clasificación con imágenes del subconjunto de pruebas fue de 582 ms.	Solo el proceso de clasificación tarda en promedio 115 ms. El resto consiste en la transmisión de datos entre cliente y servidor.

Anexo A.

Código fuente.

En esta sección se presentan el contenido del archivo pom.xml y las clases pertenecientes al proyecto del sistema Web.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://ma-
ven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>mx.ipn.escom</groupId>
  <artifactId>classifier</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>classifier</name>
  <description>Classifier for tomato diseases webapp</description>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>1.5.8.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <properties>
    <project.build.sourceEncoding>UTF-8</pro-
ject.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.report-
ing.outputEncoding>
    <java.version>1.8</java.version>
  <maven.compiler.source>1.8</maven.compiler.source>
```

```
<maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
        <version>1.5.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>1.5.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <version>1.5.8.RELEASE</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
        <version>1.5.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.apache.myfaces.core</groupId>
        <artifactId>myfaces-impl</artifactId>
        <version>2.2.12</version>
    </dependency>
    <dependency>
        <groupId>org.apache.myfaces.core</groupId>
        <artifactId>myfaces-api</artifactId>
        <version>2.2.12</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
        <version>8.5.23</version>
    </dependency>
    <dependency>
        <groupId>org.ocpsoft.rewrite</groupId>
        <artifactId>rewrite-servlet</artifactId>
        <version>3.4.1.Final</version>
    </dependency>
    <dependency>
        <groupId>org.ocpsoft.rewrite</groupId>
        <artifactId>rewrite-integration-faces</artifactId>
        <version>3.4.1.Final</version>
    </dependency>

```

```

        </dependency>
        <dependency>
            <groupId>org.ocpsoft.rewrite</groupId>
            <artifactId>rewrite-config-prettyfaces</artifactId>
            <version>3.4.1.Final</version>
        </dependency>
        <dependency>
            <groupId>org.primefaces</groupId>
            <artifactId>primefaces</artifactId>
            <version>6.1</version>
        </dependency>
        <dependency>
            <groupId>org.opencv</groupId>
            <artifactId>opencv</artifactId>
            <version>3.3.0</version>
        </dependency>
    </dependencies>

    <build>
        <outputDirectory>src/main/webapp/WEB-INF/classes</outputDirectory>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>1.5.8.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

ClassifierApplication.java

```

package mx.ipn.escom;

import org.ocpsoft.rewrite.servlet.RewriteFilter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
```

```

import org.springframework.context.annotation.ComponentScan;
import javax.faces.webapp.FacesServlet;
import javax.servlet.DispatcherType;
import java.util.EnumSet;
import mx.ipn.escom.classifier.Classifier;
import org.opencv.core.Core;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration;
import org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;

@ComponentScan({"mx.ipn.escom.presentation", "mx.ipn.escom.services"})
@SpringBootApplication
public class ClassifierApplication {

    public static void main(String[] args) {
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
        SpringApplication.run(ClassifierApplication.class, args);
    }

    @Bean
    public ServletRegistrationBean servletRegistrationBean() {
        ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean(
                new FacesServlet(), "*.jsf");
        servletRegistrationBean.setLoadOnStartup(1);
        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean rewriteFilter() {
        FilterRegistrationBean rwFilter = new FilterRegistrationBean(new
RewriteFilter());
        rwFilter.setDispatcherTypes(EnumSet.of(DispatcherType.FORWARD,
DispatcherType.REQUEST,
DispatcherType.ASYNC, DispatcherType.ERROR));
        rwFilter.addUrlPatterns("/*");
        return rwFilter;
    }

    @Bean
    public Classifier getClassifier() {
        return new Classifier();
    }
}

```

ClassifierController.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package mx.ipn.escom.presentation;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import mx.ipn.escom.pojos.Leaf;
import mx.ipn.escom.services.ClassifierService;
import org.ocpsoft.rewrite.annotation.Join;
import org.ocpsoft.rewrite.el.ELBeanName;
import org.primefaces.event.FileUploadEvent;
import org.primefaces.model.DefaultStreamedContent;
import org.primefaces.model.StreamedContent;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

/**
 *
 * @author edgar
 */

@Scope (value = "session")
@Component (value = "classifierBean")
@ELBeanName(value = "classifierBean")
@Join(path = "/classifier", to = "/index.jsf")
public class ClassifierController {

    @Autowired(required = true)
    private ClassifierService classifierService;

    private Leaf leaf;
    private String results;
    private StreamedContent image;

    private List<String> diseases;

    private ClassifierController(){
        this.leaf = new Leaf();
        this.diseases = new ArrayList();
    }
}
```

```

        this.diseases.add("Virus del rizado amarillo del tomate (Tomato
yellow leaf curl virus).");
        this.diseases.add("Virus del mosaico del tomate (Tomato mosaic
virus).");
        this.diseases.add("Corynespora cassiicola. Mancha en forma de
blanco. (Target spot).");
        this.diseases.add("Araña roja (Spider mites).");
        this.diseases.add("Septoriosis (Septoria spot).");
        this.diseases.add("Passalora fulva. Moho en la hoja (Leaf
mold).");
        this.diseases.add("Tizón tardío (Lateblight).");
        this.diseases.add("Tizón temprano (Earlyblight).");
        this.diseases.add("Mancha bacteriana (Bacterial spot).");
    }

    public String foo() {
        System.out.println("foo");
        return "Foo: ";
    }

    public void subirImagen(FileUploadEvent event) {
        try {
            byte[] content = new byte[event.getFile().get-
InputStream().available()];
            event.getFile().getInputStream().read(content);

            leaf.setImage(content);
            this.results = "A continuación se muestran las dos enfermeda-
des más probables.\n\n" + classifierService.classify(leaf, 2);

            this.image = new DefaultStreamedContent(event.getFile().get-
InputStream(), "image/png");

            FacesMessage message = new FacesMessage("Identificación", "Se
realizó la clasificación correctamente.");
            FacesContext.getCurrentInstance().addMessage(null, message);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    /**
     * @return the leaf
     */
    public Leaf getLeaf() {
        return leaf;
    }

    /**
     * @param leaf the leaf to set
     */
    public void setLeaf(Leaf leaf) {

```

```
    this.leaf = leaf;
}

/**
 * @return the results
 */
public String getResults() {
    return results;
}

/**
 * @param results the results to set
 */
public void setResults(String results) {
    this.results = results;
}

/**
 * @return the diseases
 */
public List<String> getDiseases() {
    return diseases;
}

/**
 * @param diseases the diseases to set
 */
public void setDiseases(List<String> diseases) {
    this.diseases = diseases;
}

/**
 * @return the image
 */
public StreamedContent getImage() {
    return image;
}

/**
 * @param image the image to set
 */
public void setImage(StreamedContent image) {
    this.image = image;
}

}
```

ClassifierService.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package mx.ipn.escom.services;

import mx.ipn.escom.pojos.Leaf;

/**
 *
 * @author edgar
 */
public interface ClassifierService {
    public String classify(Leaf leaf, int numResults);
}
```

ClassifierServiceImpl.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package mx.ipn.escom.services;

import mx.ipn.escom.classifier.Classifier;
import mx.ipn.escom.pojos.Leaf;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 *
 * @author edgar
 */
@Service
public class ClassifierServiceImpl implements ClassifierService{

    @Autowired(required = true)
    private Classifier classifier;

    @Override
    public String classify(Leaf leaf, int numResults){
        classifier.setImage(leaf.getImage());
        classifier.forward();
        return classifier.getResults(numResults);
    }
}
```

```
    }  
}  
  
}
```

Leaf.java

```
/*  
 * To change this license header, choose License Headers in Project Properties.  
 * To change this template file, choose Tools | Templates  
 * and open the template in the editor.  
 */  
package mx.ipn.escom.pojos;  
  
import java.io.InputStream;  
  
/**  
 *  
 * @author edgar  
 */  
public class Leaf {  
  
    private byte[] image;  
  
    /**  
     * @return the image  
     */  
    public byte[] getImage() {  
        return image;  
    }  
  
    /**  
     * @param image the image to set  
     */  
    public void setImage(byte[] image) {  
        this.image = image;  
    }  
}
```

Classifier.java

```
/*  
 * To change this license header, choose License Headers in Project Properties.  
 * To change this template file, choose Tools | Templates  
 * and open the template in the editor.  
 */  
package mx.ipn.escom.classifier;
```

```

/**
 *
 * @author edgar
 */
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;
import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.MatOfByte;
import org.opencv.core.Scalar;
import org.opencv.core.Size;
import org.opencv.dnn.Dnn;
import org.opencv.dnn.Net;
import org.opencv.imgcodecs.Imgcodecs;

public class Classifier {

    private final List<String> classes;
    private final Net cnn;
    private Mat image;
    private Mat predictions;

    public Classifier(String labels, String prototxt, String caffeModel,
String image) {
        this.classes = readLabels(labels);
        this.cnn = readCaffeModel(prototxt, caffeModel);
        this.image = Imgcodecs.imread(image);;
    }

    public Classifier(String labels, String prototxt, String caffeModel) {
        this.classes = readLabels(labels);
        this.cnn = readCaffeModel(prototxt, caffeModel);
        this.image = null;
    }

    public Classifier() {
        this.classes = readLabels(getPath("synset_words.txt"));
        this.cnn = readCaffeModel(getPath("deploy.prototxt"),
getPath("classifier.caffemodel"));
    }

    private List<String> readLabels(String labelsPath) {
        List<String> labels = new ArrayList();
        try{
            Scanner sc = new Scanner(new File(labelsPath));

```

```

        while(sc.hasNextLine()) {
            String line = sc.nextLine();
            labels.add( line.substring( line.indexOf(" ") + 1
).split(",") [0] );
        }
    } catch(FileNotFoundException fnfe) {
        fnfe.printStackTrace();
    }
    return labels;
}

private Net readCaffeModel(String prototxt, String caffeModel) {
    return Dnn.readNetFromCaffe(prototxt, caffeModel);
}

public void forward() {
    Mat inputBlob = Dnn.blobFromImage(this.image, 1.0, new Size(227,
227), new Scalar(104, 117, 123), false);
    this.cnn.setInput(inputBlob);
    this.predictions = cnn.forward();
}

public static List<Integer> argsort(final List<Double> a) {
    return argsort(a, true);
}

public static List<Integer> argsort(final List<Double> a, final boolean ascending) {
    List<Integer> indexes = new ArrayList();
    for (int i = 0; i < a.size(); i++) {
        indexes.add(i);
    }
    Collections.sort(indexes, (final Integer i1, final Integer i2) ->
(ascending ? 1 : -1) * Double.compare(a.get(i1), a.get(i2)));
    return indexes;
}

public String getResults(int numResults){
    String results = "";

    double len = this.predictions.size().width;
    List<Double> probs = new ArrayList();

    for(int i = 0; i < len; i++){
        probs.add(this.predictions.get(0, i)[0]);
    }
    List<Integer> idxs = Classifier.argsort(probs, false);

    for(int i = 0; i < numResults; i ++){
        results += "[INFO] " + (i + 1) + ". label: " + this.classes.get(idxs.get(i)) + ", probability: " + probs.get(idxs.get(i)) + "\n";
    }
}

```

```
    return results;
}

public void printResults(int numResults) {
    System.out.println(getResults(numResults));
}

public void setImage(byte[] image) {
    this.image = Imgcodecs.imread(new MatOfByte(image), Imgcodecs.CV_LOAD_IMAGE_UNCHANGED);
}

private String getPath(String file) {

    BufferedInputStream inputStream = null;
    try {
        inputStream = new BufferedInputStream(getClass().getClassLoader().getResourceAsStream(file));
        byte[] data = new byte[inputStream.available()];
        inputStream.read(data);
        inputStream.close();

        File outFile = new File(file);
        FileOutputStream os = new FileOutputStream(outFile);
        os.write(data);
        os.close();
        return outFile.getAbsolutePath();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
    return "";
}

}
```

Anexo B.

Manual de usuario.

En esta sección se presenta el manual de usuario, el cual describe la funcionalidad disponible para el usuario.

1. Características generales y funcionalidades.

El Sistema para la Identificación de Enfermedades del Tomate (en adelante referido como SIET) fue concebido y realizado como una herramienta de apoyo al agricultor en actividades de identificación de enfermedades en el tomate.

El usuario debe suministrar una imagen de una hoja que presente los síntomas, tal como se describe en la sección “Selección de imágenes”, y a continuación el SIET realizará la identificación de la enfermedad dentro de un conjunto de diez resultados posibles (nueve enfermedades y hoja sana).

Se muestran los dos resultados más probables y su probabilidad. Proporcionando una imagen adecuada, el SIET tiene una precisión del 98%, así que el resultado obtenido puede ser complementado con la propia experiencia del usuario para tener una mayor certidumbre de la enfermedad que presenta la planta.

La interfaz de usuario y sus componentes principales se muestran en la figura 1.1. Los componentes resaltados en dicha figura son los siguientes.

- Área de mensajes. En esta zona se muestran mensajes informativos de la aplicación.
- Enfermedades identificables. Lista de las nueve enfermedades del tomate que el SIET puede identificar.

- Selección de imagen. Área con opciones para seleccionar una imagen del almacenamiento local del dispositivo desde el que se está visualizando la interfaz de usuario.
- Resultados. Sección donde se presenta el resultado de la identificación. Se muestran los dos resultados más probables y su probabilidad.

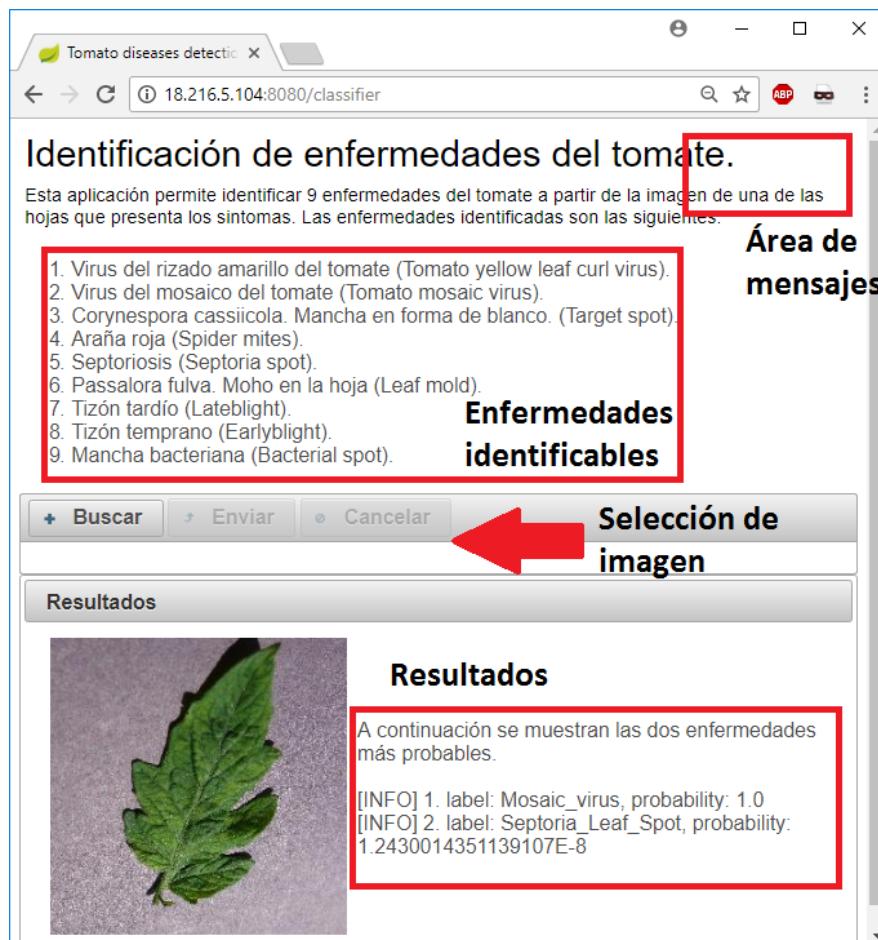


Figura 1.1 Interfaz de usuario.

2. Selección de imágenes.

La imagen que el usuario proporcione debe tener las características siguientes.

- Extensión jpg, png o bmp. Formatos soportados por el SIET.
- Resolución mínima de 256x256 pixeles. El SIET realiza un redimensionamiento de la imagen proporcionada a un tamaño de 256x256 pixeles sin importar la resolución original, por lo que está debe

ser la resolución mínima. Tampoco es recomendable que la imagen tenga una resolución alta ya que el tiempo de transmisión de esta aumentará.

- El contenido de la imagen debe ser una sola hoja con un fondo de color uniforme (figura 2.1). Cumplir esta restricción mejora la precisión de la predicción.



Figura 2.1 Imagen muestra.

Para seleccionar la imagen, el usuario debe presionar el botón **+ Buscar** en la sección de “Selección de imagen”. A continuación, se abre un cuadro de dialogo en donde el usuario puede seleccionar de su sistema de archivos local una imagen con extensión jpg, png o bmp (figura 2.2). En caso de seleccionar otro tipo de archivo se muestra un mensaje de error.

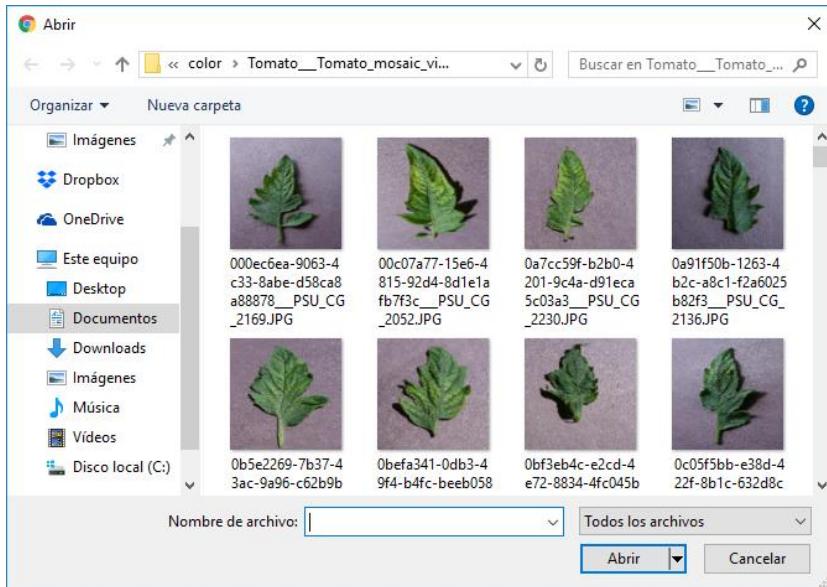


Figura 2.2 Selección de la imagen.

Una vez seleccionada la imagen se muestra una vista previa en miniatura (figura 2.3). Para realizar la identificación de la enfermedad usando dicha imagen, presionar el botón **Enviar**. Para descartar la imagen seleccionada presionar el botón **Cancelar**.



Figura 2.3 Vista previa de la imagen seleccionada.

3. Clasificación e interpretación de resultados.

Cuando el usuario envía la imagen el SIET comienza con la clasificación de esta. Per se, este proceso tarda poco más de 100 ms, sin embargo, por la transmisión de información entre cliente y servidor el tiempo de clasificación puede incrementar a algunos segundos, dependiendo principalmente de la resolución de la imagen, tal como se mencionó en la sección anterior.

El resultado de la clasificación se publica en la sección “Resultados”, mostrando las dos clases más probables entre las nueve enfermedades y hoja sana. También se presenta la probabilidad de la predicción para que el usuario pueda tener una mejor certidumbre acerca de la precisión del resultado (figura 3.1).



Figura 3.1 Resultados.

Anexo C.

Enfermedades del tomate identificables.

En esta sección se presenta los síntomas característicos de las enfermedades identificables por este sistema.

1. Virus del rizado amarillo del tomate.

El virus del rizado amarillo del tomate (TYLCV por sus siglas en inglés) es un virus que infectan al tomate transmitido por la mosca blanca *Bemisia tabaci*. Este virus pertenece al género de los *Begomovirus* de la familia *Geminiviridae*. El virus también puede transmitirse a partir de la germinación de semillas obtenidas de frutos de plantas infectadas. Este virus puede generar pérdidas críticas en la producción de tomate [19].

Plantas infectadas exhiben un enrollamiento de los bordes de la hoja hacia arriba y hacia adentro, las venas se vuelven amarillas y se detiene el crecimiento de la planta. El virus reduce considerablemente la producción de frutos, especialmente cuando la infección ocurre antes de la etapa de florecimiento. En los frutos no existen síntomas visibles derivados de la infección de la planta [20]. En la figura C.1 se presenta un ejemplo de una hoja infectada con este virus.



Figura C.1 Hoja de tomate infectada con el virus del rizado amarillo del tomate.

2. Virus del mosaico del tomate.

El virus del mosaico del tomate (ToMV por sus siglas en inglés) afecta principalmente a variedades tradicionales o autóctonas de tomate que no presentan genes de resistencia a *Tobamovirus*. También puede infectar a otras especies cultivadas y vegetación natural principalmente de la familia Solanaceae, pero también Aizoaceae, Amaranthaceae, Chenopodiaceae y Scrophulariaceae [21].

Los síntomas en tomate (figura C.2) pueden agruparse como:

- Mosaicos de color verde claro - verde oscuro con deformación de la hoja y a veces filiformismos. Las plantas son menos vigorosas y producen menos. La maduración del fruto no es uniforme.
- Mosaicos de color amarillo o blanquecinos similares a los de la planta aucuba.
- Necrosis de tallos pecíolos, hojas y frutos. Frutos con manchas externas pardo-oscuras y necrosis internas.



Figura C.2 Hoja de tomate infectada con el virus del mosaico.

3. Corynespora cassiicola. (Mancha en forma de blanco).

Corynespora cassiicola es una especie de hongo conocido como un patógeno de las plantas. Es un patógeno que sobrevive en los residuos de cultivo, las semillas y el suelo.

Genera lesiones foliares redondas e irregulares, marrón rojizas, variando desde pequeños puntos a manchas necróticas de 1.5 cm de diámetro. Es común observar un color amarillo alrededor de las lesiones (figura C.3) [22].



Figura C.3 Hoja de tomate enferma con *Corynespora cassiicola*.

4. Araña roja.

La araña roja del tomate es una especie de araña de color naranja pálido a rojo y se alimenta de la cara inferior de las hojas. Estas arañas son tan pequeñas que no pueden ser identificadas a simple vista, a menos de que se hayan multiplicado hasta formar colonias. Los daños aparecen como múltiples marcas brillantes de color amarillo. Eventualmente las hojas se tornan de color café y mueren o se desprenden. Un ataque más severo conduce a la formación de telaraña en la planta (figura C.4) [20].



Figura C.4 Hoja de tomate con daños generados por araña roja.

5. Septoriosis.

La septoriosis es causada por el hongo *Septoria lycopersici*. Es una de las enfermedades del tomate más destructivas y es particularmente severa en áreas donde el ambiente húmedo persiste por períodos extensos de tiempo.

La septoriosis generalmente se manifiesta en las hojas de la parte inferior de la planta como puntos circulares, de entre 1-6 mm de diámetro. Los puntos son de color café oscuro en los márgenes y gris en el centro. Generalmente existen muchos puntos en cada hoja (figura C.5). La enfermedad se propaga de las hojas viejas a las hojas jóvenes. Si las lesiones son numerosas, la hoja se torna ligeramente amarilla,

posteriormente café y finalmente se marchita. Los frutos raramente resultan infectados [23].



Figura C.5 Hoja de tomate con síntomas de septoriosis.

6. *Passalora fulva* (Mojo en la hoja).

Passalora fulva es una especie de hongo, el patógeno causante del moho en la hoja de tomate. Los primeros síntomas son una serie de puntos amarillos, de un margen indefinido, en la cara superior de la hoja (figura C.6). La esporulación, que se produce en la superficie inferior de la hoja, debajo de los puntos, es de un color gris claro, pero se vuelve de color café claro a café rojizo o verde oliva. La pérdida de follaje puede ocurrir si la infección es lo suficientemente severa [20].



Figura C.6 Hoja de tomate infectada con *passalora fulva*.

7. Tizón tardío.

El tizón tardío es la enfermedad más seria que afecta al tomate. Puede rápidamente (una semana) arruinar plantaciones enteras y generar fuentes de infección para otras plantas. El tizón tardío es distinto a otras enfermedades del tomate. Estás afectar generalmente solo las hojas o causan daños limitados al fruto, y aunque pueden disminuir la producción, generalmente no causan una pérdida total. Además, debido a que la mayoría de los patógenos no son dispersados por el viento, sus efectos se encuentran localizados. Por otro lado, el tizón tardío mata las plantas por completo y es altamente contagioso.

Los primeros síntomas son manchas de un color verde oscuro en las hojas jóvenes (figura C.7). Posteriormente la hoja se marchita y muere. Las manchas también pueden presentarse en los tallos y en los frutos y se vuelven de color café en condiciones de humedad. Las manchas se unen para formar regiones más grandes de un color café oscuro [20].



Figura C.7 Hoja de tomate infectada con tizón tardío.

8. Tizón temprano.

El tizón temprano es una enfermedad generada por un hongo. Esta enfermedad produce manchas irregulares de color café de entre 0.5 a 3 cm de diámetro. Alrededor de las manchas, la hoja se vuelve de color amarillo y puede llegar a secarse y desprenderse (figura C.8). La enfermedad comienza en las hojas viejas y se dispersa hacia las hojas jóvenes, en la parte superior de la planta [20].



Figura C.8 Hoja de tomate infectada con tizón temprano.

9. Mancha bacteriana.

La enfermedad mancha bacteriana también es conocida como costra bacteriana, cáncer de tallo o macha foliar. Produce manchas de color amarillo en la hoja que van adquiriendo un color café oscuro (figura C.9). Las flores afectadas se desprenden. Algunas veces las manchas están rodeadas por una zona de un color amarillo [20].



Figura C.9 Hoja de tomate con presencia de mancha bacteriana.

Referencias.

- [1] SAGARPA. Hidroponia rústica. [En línea] Disponible en: <http://www.sagarpa.gob.mx>.
- [2] Research and Markets. Global Hydroponics Market - Forecasts from 2017 to 2022. [En línea] Disponible en: <https://www.researchandmarkets.com>.
- [3] Sánchez F. Entrevista con Félice Sanchez del Castillo, investigador de la Universidad Autónoma Chapingo. Recuperado de <http://www.2000agro.com.mx>.
- [4] Mohanty S., Hughes D., Salathé M. Using Deep Learning for Image-Based Plant Disease Detection. 2016.
- [5] Moujahid A. A practical Introduction to Deep Learning with Caffe and Python. [En línea] Disponible en <http://adilmoujahid.com>
- [6] CS231n. Convolutional Neural Networks for Visual Recognition. [En linea] Disponible en <http://cs231n.github.io/convolutional-networks/>
- [7] Szegedy et al. Going Deeper with Convolutions. [En línea] Disponible en: <https://arxiv.org/abs/1409.4842>
- [8] Mohamed Brahimi. Computer Science Department. Mohamed El Bachir El Ibrahimi University.
- [9] CS231n. Transfer Learning. [En línea] Disponible en <http://cs231n.github.io/transfer-learning/>
- [10] Caffe. Deep Learning Framework. [En línea] Disponible en <http://caffe.berkeleyvision.org/>
- [11] BVLC reference caffenet. [En línea]. Disponible en https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet.
- [12] Berkeley Vision and Learning Center. Caffe, Ubuntu 16.04 Installation Guide. [En línea] Disponible en <https://github.com/BVLC/caffe/wiki/Ubuntu-16.04-or-15.10-Installation-Guide>.
- [13] Rosebrock A. Ubuntu 16.04: How to install OpenCV. [En línea] Disponible en <https://www.pyimagesearch.com>.
- [14] Rosebrock A. Deep Learning with OpenCV. [En línea] Disponible en <https://www.pyimagesearch.com>.

- [15] BVLC. Using a Trained Network: Deploy. [En línea] Disponible en <https://github.com/BVLC/caffe/wiki/Using-a-Trained-Network:-Deploy>.
- [16] Microsoft Developer Network. Designing Web Applications. [En línea] Disponible en <https://msdn.microsoft.com/>.
- [17] Universidad de Alicante. Introducción a JavaServer Faces. [En línea] Disponible en <http://www.jtech.ua.es/>
- [18] Amazon Web Services. Amazon EC2. [En línea] Disponible en <https://aws.amazon.com/es/ec2/>.
- [19] Nature research. Tomato yellow leaf curl virus (TYLCV-IL): a seed-transmissible geminivirus in tomatoes. [En línea] Disponible en <https://www.nature.com>.
- [20] CAB International .Plantwise Knoldge Bank. [En línea] Disponible en <https://www.plantwise.org/KnowledgeBank/>.
- [21] Ministerio de Agricultura y Pesca, Alimentación y Medio Ambiente (España). Virus del mosaico del tomate. [En línea] Disponible en <http://www.mapama.gob.es/>.
- [22] Sistema Nacional Argentino de Vigilancia y Monitoreo de plagas. Corynespora cassiicola. [En línea] Disponible en <http://www.sinavimo.gov.ar/plaga/corynespora-cassiicola>.
- [23] Missouri Botanical Garden. Septorial Leaf Spot of Tomato. [En línea] Disponible en <http://www.missouribotanicalgarden.org/>.