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



*Mayo de 2018*

INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

SUBDIRECCIÓN ACADÉMICA

No de TT:2017-A054 de mayo de 2017

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, machine learning, deep learning, red neuronal artificial.

I. Resumen

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

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 del mismo.

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 de una Red Neuronal Convolucional (CNN). Se incluye una breve introducción a los conceptos del 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 de deep learning Caffe. Una vez entrenado el clasificador podrán realizarse predicciones con él usando Python y en particular el módulo dnn de OpenCV.

II. Abstract

The purpose of this project is the developing of a computer vision application that analyze plant leaves images with a visible disease. As the result, the application gives a probably diagnostic of the plant disease.

In this document are defined and designed the different phases that this prototype needs for its implementation.

In the chapter 1 is presented a brief introduction where is explained the context of this project and the motivation behind its development. Also, the objectives for the design phase are presented.

In the chapter 2 is found the theoretical framework the object detection topic is discussed. The object detection is required in the segmentation step, where the leaves are separated from the background of the image, because the leaves contains the useful information for the next steps of the application.

III. Índice general

[I. Resumen 3](#_Toc508765863)

[II. Abstract 4](#_Toc508765864)

[III. Índice general 5](#_Toc508765865)

[Capítulo 1. Introducción 7](#_Toc508765866)

[1.1 Planteamiento del problema. 7](#_Toc508765867)

[1.2 Objetivos. 8](#_Toc508765868)

[1.2.1 Objetivo general. 8](#_Toc508765869)

[1.2.2 Objetivos particulares. 9](#_Toc508765870)

[1.3 Justificación. 9](#_Toc508765871)

[1.4 Metodología. 10](#_Toc508765872)

[Capítulo 2. Marco teórico 12](#_Toc508765873)

[2.1 Comparación de clasificación de imágenes usando modelos tradicionales de Machine Leaning y Deep Learning. 12](#_Toc508765874)

[2.2 Redes Neuronales Artificiales (ANN). 14](#_Toc508765875)

[2.2.1 Redes neuronales unidireccionales. 15](#_Toc508765876)

[2.2.2 Funciones de activación. 15](#_Toc508765877)

[2.2.3 Entrenamiento de una red neuronal artificial. 16](#_Toc508765878)

[2.3 Redes Neuronales Convolucionales (CNN). 16](#_Toc508765879)

[2.3.1 Capa convolucional. 17](#_Toc508765880)

[2.3.2 Capa de agrupamiento. 18](#_Toc508765881)

[2.4 Arquitectura de las redes neuronales convolucionales. 18](#_Toc508765882)

[Capítulo 3. Análisis del clasificador. 20](#_Toc508765883)

[3.1 Requisitos del sistema. 21](#_Toc508765884)

[3.1.1 Requisitos de software. 21](#_Toc508765885)

[3.2 Modelo de casos de uso. 22](#_Toc508765886)

[3.3 Interfaz gráfica. 22](#_Toc508765887)

[Capítulo 4. Diseño del clasificador. 23](#_Toc508765888)

[4.1 Fine-Tuning. 23](#_Toc508765889)

[4.2 Caffe. Deep Learning Framework. 24](#_Toc508765890)

[Capítulo 5. Construcción del clasificador. 26](#_Toc508765891)

[5.1 Instalación de Caffe. 26](#_Toc508765892)

[5.1.1 Instalación de OpenCV. 26](#_Toc508765893)

[5.1.2 Instalación de Caffe. 28](#_Toc508765894)

[5.2 Fine-tuning de las redes GoogLeNet y CaffeNet con Caffe. 30](#_Toc508765895)

[5.3 Realización de predicciones con la CaffeNet, Python y OpenCV. 45](#_Toc508765896)

[Capítulo 6. Análisis del sistema Web. 52](#_Toc508765897)

[6.1 Requisitos del sistema. 52](#_Toc508765898)

[6.1.1 Requisitos de software. 52](#_Toc508765899)

[6.2 Modelo de casos de uso. 53](#_Toc508765900)

[6.2.1 CU1. Seleccionar imagen. 54](#_Toc508765901)

[6.2.1 CU2. Obtener diagnóstico. 55](#_Toc508765902)

[6.3 Interfaz gráfica. 56](#_Toc508765903)

[6.3.1 IU1.0 Pantalla de inicio. 56](#_Toc508765904)

[Referencias. 58](#_Toc508765905)

Capítulo 1. Introducción

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 del mismo, 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. 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 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 procesarlas. En dicha solución se considera un conjunto de datos de 14,828 imágenes clasificadas en nueve enfermedades que presenta el tomate.

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 anormalidad 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 [3].

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 (en adelante llamado deep learning), específicamente de una Red Neuronal Convolucional (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

Marco teórico.

El deep learning es una tendencia reciente en al 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 tradicionales de Machine Leaning y Deep Learning.

La clasificación de imágenes usando algoritmos de machine learning 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 machine learning, 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 machine learning.

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 machine learning para predecir su etiqueta o clase a la que pertenece. Ambas etapas, entrenamiento y predicción se muestran de forma esquemática en la figura 2.1.

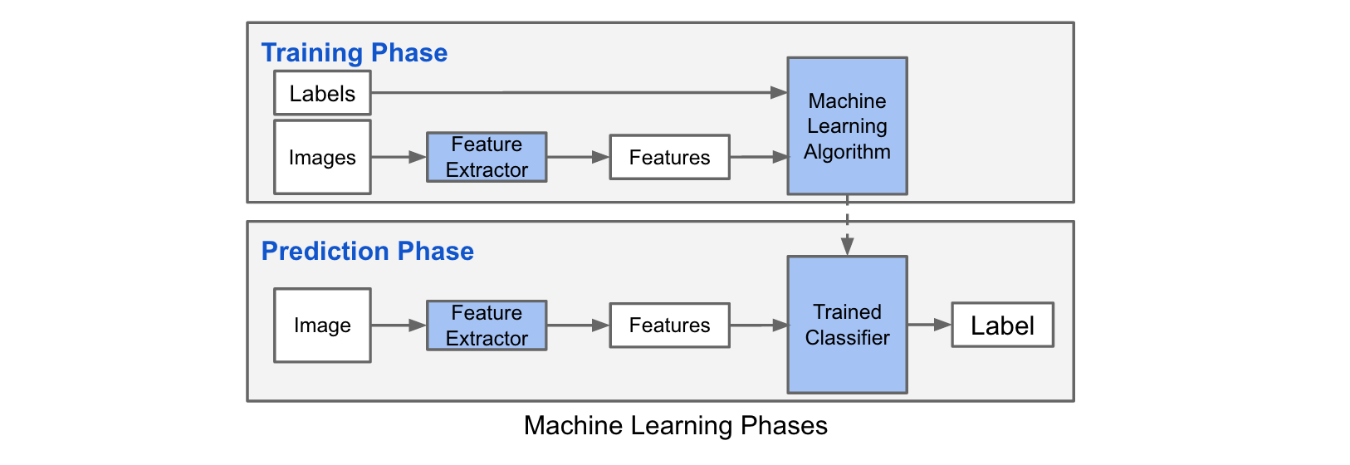


Figura 2.1 Clasificación con algoritmos de machine learning.

Ahora, la principal diferencia de los algoritmos tradicionales de machine learning y los algoritmos de deep learning radica en la ingeniería de descriptores. En los algoritmos tradicionales de machine learning este proceso es elaborado manualmente, es decir, la selección y extracción debe ser diseñada e implementada por el programador. En cambio, en los algoritmos de deep learining 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 machine learning prometen resultados más precisos comparados con los algoritmos tradicionales de machine learning, con menos o incluso sin ingeniería de descriptores. Esta diferencia se muestra en la figura 2.2.

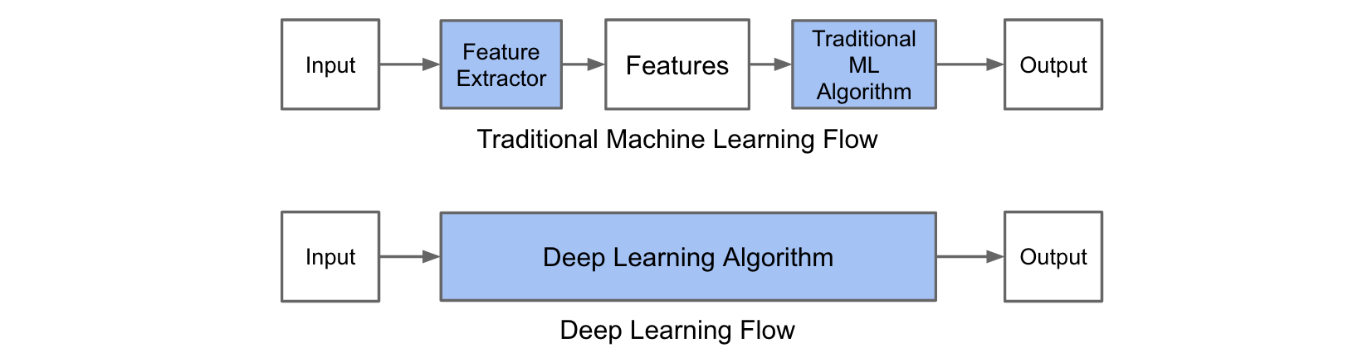


Figura 2.2 Diferencias entre la clasificación con modelos de machine learning y deep learning.

2.2 Redes Neuronales Artificiales (ANN).

Deep learning 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 deep learning puede rastrearse a la publicación “ImageNet Classification with Deep Convolutional Networks” realizada por Krizhevsky, Sutskever, and 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 computadoras, el uso de GPUs y la disponibilidad de grandes conjuntos de datos han favorecido el desarrollo de los modelos de deep learning.

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.

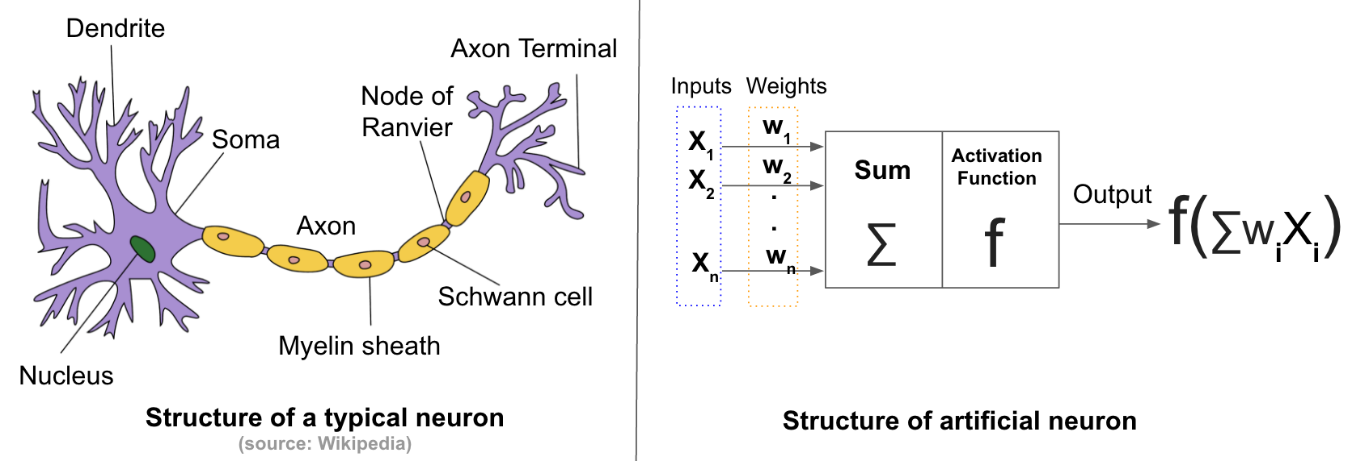


Figura 2.3 Estructura de una neurona artificial.

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 de 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. Está función mide la imprecisión de las predicciones.

Con estos dos elementos se entrena la red neuronal con un algoritmo llamado backpropagation [4].

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 CNNs está compuesta por capas tres capas: Capa convolucional (CONV layer), capa de agrupamiento (pooling layer) y capa conectada completamente (Fully-connected layer). 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 de 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 machine learning. 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 [5].



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 popularizo 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 ILSVRC challenge 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 [6] de Google.

Capítulo 3. Análisis del clasificador.

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. Este conjunto de datos es un subconjunto del repositorio de acceso abierto publicado en [www.PlantVillage.org](http://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 Brahimi M. [7].

El conjunto de datos usados en este sistema está formado por 14,828 imágenes divididas en nueve enfermedades del tomate, tal como se describe en la tabla 3.1.

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

Tabla 3.1 Conjunto de datos del proyecto.

3.1 Requisitos del sistema.

Este proyecto nació 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 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 3.1, 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, clasificar 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 deep learning desarrollado en C++ por Berkeley Al Research (BAIR). Los requisitos del sistema para este framework realmente dependen de las acciones que se vayan a realizar. Para este proyecto se realizará un fine-tuning (red pre-entrenada), 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 require 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.

Diseño del clasificador.

Como ya se ha mencionado anteriormente, el algoritmo clasificador será un algoritmo de deep leaning, en particular una red neuronal convolucional (CNN). Lo anterior debido a las ventajas que presenta al ser comparado con los algoritmos tradicionales de machine learning, 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 presenta una mayor eficiencia.

4.1 Fine-Tuning.

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 entrenan 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 conjuntos 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 extractor 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 (Fine-tuning).** 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 el backpropagation. 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 fine-tuning.

Ahora, teniendo en consideración que el conjunto de datos de este proyecto es relativamente grande (14,828 imágenes) es factible la realización de un fine-tuning sobre una red pre-entrenada en ImageNet [8].

4.2 Caffe. Deep Learning Framework.

Caffe es un framework para deep learning desarrollado en C++ por Berkeley Al Research (BAIR). Algunas de los beneficios de este framework son los siguientes [9]:

* Los modelos y la optimización son definidos mediante archivos de configuración sin la necesidad de hard-coding. 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 deep learning.
* 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, etc. 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 [10]. Para la etapa de entrenamiento se realizará un proceso de fine-tuning 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 (CPU only). 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.

Construcción del clasificador.

Es 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 [11] 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 conexiones con Python. [11].

Actualizar los paquetes y bibliotecas pre-instaladas.

sudo apt-get update

sudo apt-get upgrade

Instalar las siguientes herramientas de desarrollo.

sudo at-get install build-essential cmake pkg-config

Instalar las bibliotecas para leer imagines 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



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 libleveldb-dev libsnappy-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 archive 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 archive 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 archive:

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 archive 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 Fine-tuning 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 separase 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 primer parte del archivo todas la 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 Fine tuning de la CaffeNet.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **net**: **"models/bvlc\_reference\_caffenet/train\_val.prototxt"**  **test\_iter**: **1000**  **test\_interval**: **1000**  **base\_lr**: **0.01**  **lr\_policy**: **"step"**  **gamma**: **0.1**  **stepsize**: **100000**  **display**: **20**  **max\_iter**: **450000**  **momentum**: **0.9**  **weight\_decay**: **0.0005**  **snapshot**: **10000**  **snapshot\_prefix**: **"models/bvlc\_reference\_caffenet/caffenet\_train"**  **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), que para el caso del proceso de fine tuning 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 decrementar el coeficiente de aprendizaje. La forma en que este decremento ocurre se indica en las líneas 5, 6 y 7. El parámetro lr\_policy indica la forma en que ocurre este decremento. En el caso “step” se decrementa el coeficiente de aprendizaje cada cierto número de iteraciones (stepsize) por un factor de gamma. Por lo tanto, en un proceso de fine tuning, el stepsize también debería ser mayor, 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 fine tuning queda como se muestra a continuación.

**net**: **"/home/edgar/git/PlantDiseaseDetection/deepLearning/models/finetune\_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 capad 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | **name:** "CaffeNet"  layer {  **name:** "data"  **type:** "Data"  **top:** "data"  **top:** "label"  include {  **phase:** TRAIN  }  transform\_param {  **mirror:** **true**  **crop\_size:** **227**  **mean\_file:** "data/ilsvrc12/imagenet\_mean.binaryproto"  }  data\_param {  **source:** "examples/imagenet/ilsvrc12\_train\_lmdb"  **batch\_size:** **256**  **backend:** LMDB  }  }  layer {  **name:** "data"  **type:** "Data"  **top:** "data"  **top:** "label"  include {  **phase:** TEST  }  transform\_param {  **mirror:** **false**  **crop\_size:** **227**  **mean\_file:** "data/ilsvrc12/imagenet\_mean.binaryproto"  }  data\_param {  **source:** "examples/imagenet/ilsvrc12\_val\_lmdb"  **batch\_size:** **50**  **backend:** LMDB  }  } |

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 fine tuning se muestra a continuación.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | **name:** "TomatoDiseasesCaffeNet"  layer {  **name:** "data"  **type:** "ImageData"  **top:** "data"  **top:** "label"  include {  **phase:** TRAIN  }  transform\_param {  **mirror:** **true**  **crop\_size:** **227**  **mean\_file:** "/home/edgar/git/PlantDiseaseDetection/deepLearning/data/ilsvrc12/imagenet\_mean.binaryproto"  }  image\_data\_param {  **source:** "/home/edgar/git/PlantDiseaseDetection/data/train.txt"  **batch\_size:** **100**  **new\_height:** **256**  **new\_width:** **256**  }  }  layer {  **name:** "data"  **type:** "ImageData"  **top:** "data"  **top:** "label"  include {  **phase:** TEST  }  transform\_param {  **mirror:** **false**  **crop\_size:** **227**  **mean\_file:** "/home/edgar/git/PlantDiseaseDetection/deepLearning/data/ilsvrc12/imagenet\_mean.binaryproto"  }  image\_data\_param {  **source:** "/home/edgar/git/PlantDiseaseDetection/data/test.txt"  **batch\_size:** **100**  **new\_height:** **256**  **new\_width:** **256**  }  } |

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | layer {  **name:** "fc8"  **type:** "InnerProduct"  **bottom:** "fc7"  **top:** "fc8"  param {  **lr\_mult:** **1**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **2**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **1000**  weight\_filler {  **type:** "gaussian"  **std:** **0.01**  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  } |

De esta capa debe modificarse el nombre (línea 2), para que al iniciar el proceso de fine tuning, 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | layer {  **name:** "fc8\_tomatoDisease"  **type:** "InnerProduct"  **bottom:** "fc7"  **top:** "fc8\_tomatoDisease"  param {  **lr\_mult:** **10**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **20**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **10**  weight\_filler {  **type:** "gaussian"  **std:** **0.01**  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  }  layer {  **name:** "accuracy"  **type:** "Accuracy"  **bottom:** "fc8\_tomatoDisease"  **bottom:** "label"  **top:** "accuracy"  include {  **phase:** TEST  }  }  layer {  **name:** "loss"  **type:** "SoftmaxWithLoss"  **bottom:** "fc8\_tomatoDisease"  **bottom:** "label"  **top:** "loss"  } |

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 dejo de incrementarse.

5.2.2 Fine tuning de la GoogLeNet.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **net**: **"models/bvlc\_googlenet/train\_val.prototxt"**  **test\_iter**: **1000**  **test\_interval**: **4000**  **test\_initialization**: **false**  **display**: **40**  **average\_loss**: **40**  **base\_lr**: **0.01**  **lr\_policy**: **"step"**  **stepsize**: **320000**  **gamma**: **0.96**  **max\_iter**: **10000000**  **momentum**: **0.9**  **weight\_decay**: **0.0002**  **snapshot**: **40000**  **snapshot\_prefix**: **"models/bvlc\_googlenet/bvlc\_googlenet"**  **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 fine tuning se muestra a continuación.

**net**: **"/home/edgar/git/PlantDiseaseDetection/deepLearning/models/finetune\_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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | **name:** "TomatoDiseasesGoogleNet"  layer {  **name:** "data"  **type:** "ImageData"  **top:** "data"  **top:** "label"  include {  **phase:** TRAIN  }  transform\_param {  **mirror:** **true**  **crop\_size:** **224**  **mean\_value:** **104**  **mean\_value:** **117**  **mean\_value:** **123**  }  image\_data\_param {  **source:** "/home/edgar/git/PlantDiseaseDetection/data/train.txt"  **batch\_size:** **50**  **new\_height:** **256**  **new\_width:** **256**  }  }  layer {  **name:** "data"  **type:** "ImageData"  **top:** "data"  **top:** "label"  include {  **phase:** TEST  }  transform\_param {  **mirror:** **false**  **crop\_size:** **224**  **mean\_value:** **104**  **mean\_value:** **117**  **mean\_value:** **123**  }  image\_data\_param {  **source:** "/home/edgar/git/PlantDiseaseDetection/data/test.txt"  **batch\_size:** **50**  **new\_height:** **256**  **new\_width:** **256**  }  } |

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | layer {  **name:** "loss1/classifier"  **type:** "InnerProduct"  **bottom:** "loss1/fc"  **top:** "loss1/classifier"  param {  **lr\_mult:** **1**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **2**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **1000**  weight\_filler {  **type:** "xavier"  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  }  ...  layer {  **name:** "loss2/classifier"  **type:** "InnerProduct"  **bottom:** "loss2/fc"  **top:** "loss2/classifier"  param {  **lr\_mult:** **1**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **2**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **1000**  weight\_filler {  **type:** "xavier"  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  }  ...  layer {  **name:** "loss3/classifier"  **type:** "InnerProduct"  **bottom:** "pool5/7x7\_s1"  **top:** "loss3/classifier"  param {  **lr\_mult:** **1**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **2**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **1000**  weight\_filler {  **type:** "xavier"  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  } |

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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | layer {  **name:** "loss1/classifier\_tomatoDiseases"  **type:** "InnerProduct"  **bottom:** "loss1/fc"  **top:** "loss1/classifier\_tomatoDiseases"  param {  **lr\_mult:** **10**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **20**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **10**  weight\_filler {  **type:** "xavier"  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  }  ...  layer {  **name:** "loss2/classifier\_tomatoDiseases"  **type:** "InnerProduct"  **bottom:** "loss2/fc"  **top:** "loss2/classifier\_tomatoDiseases"  param {  **lr\_mult:** **10**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **20**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **10**  weight\_filler {  **type:** "xavier"  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  }  ...  layer {  **name:** "loss3/classifier\_tomatoDiseases"  **type:** "InnerProduct"  **bottom:** "pool5/7x7\_s1"  **top:** "loss3/classifier\_tomatoDiseases"  param {  **lr\_mult:** **10**  **decay\_mult:** **1**  }  param {  **lr\_mult:** **20**  **decay\_mult:** **0**  }  inner\_product\_param {  **num\_output:** **10**  weight\_filler {  **type:** "xavier"  }  bias\_filler {  **type:** "constant"  **value:** **0**  }  }  } |

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, 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 caffe cálculo 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 [13].

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | # USAGE  # python deep\_learning\_with\_opencv.py --image images/Bact.Sp\_01.JPG --prototxt deploy.prototxt --model finetune\_tomato\_diseases\_iter\_8097.caffemodel --labels synset\_words.txt  # import the necessary packages  **import** **numpy** **as** **np**  **import** **argparse**  **import** **time**  **import** **cv2**  # construct the argument parse and parse the arguments  ap = argparse.ArgumentParser()  ap.add\_argument("-i", "--image", required=**True**,  help="path to input image")  ap.add\_argument("-p", "--prototxt", required=**True**,  help="path to Caffe 'deploy' prototxt file")  ap.add\_argument("-m", "--model", required=**True**,  help="path to Caffe pre-trained model")  ap.add\_argument("-l", "--labels", required=**True**,  help="path to ImageNet labels (i.e., syn-sets)")  args = vars(ap.parse\_args())  # load the input image from disk  image = cv2.imread(args["image"])  # load the class labels from disk  rows = open(args["labels"]).read().strip().split("**\n**")  classes = [r[r.find(" ") + **1**:].split(",")[**0**] **for** r **in** rows]  # our CNN requires fixed spatial dimensions for our input image(s)  # so we need to ensure it is resized to 224x224 pixels while  # performing mean subtraction (104, 117, 123) to normalize the input;  # after executing this command our "blob" now has the shape:  # (1, 3, 224, 224)  blob = cv2.dnn.blobFromImage(image, **1**, (**227**, **227**), (**104**, **117**, **123**) , **False**)  # load our serialized model from disk  print("[INFO] loading model...")  net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])  # set the blob as input to the network and perform a forward-pass to  # obtain our output classification  net.setInput(blob)  start = time.time()  preds = net.forward()  end = time.time()  print("[INFO] classification took {:.5} seconds".format(end - start))  # sort the indexes of the probabilities in descending order (higher  # probabilitiy first) and grab the top-5 predictions  idxs = np.argsort(preds[**0**])[::-**1**][:**5**]  # loop over the top-5 predictions and display them  **for** (i, idx) **in** enumerate(idxs):  # draw the top prediction on the input image  **if** i == **0**:  text = "Label: {}, {:.2f}%".format(classes[idx],  preds[**0**][idx] \* **100**)  cv2.putText(image, text, (**5**, **25**), cv2.FONT\_HERSHEY\_SIMPLEX,  **0.7**, (**0**, **0**, **255**), **2**)  # display the predicted label + associated probability to the  # console  print("[INFO] {}. label: {}, probability: {:.5}".format(i + **1**,  classes[idx], preds[**0**][idx]))  # display the output image  cv2.imshow("Image", image)  cv2.waitKey(**0**) |

El programa anterior require 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 [14].

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**

1. 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.

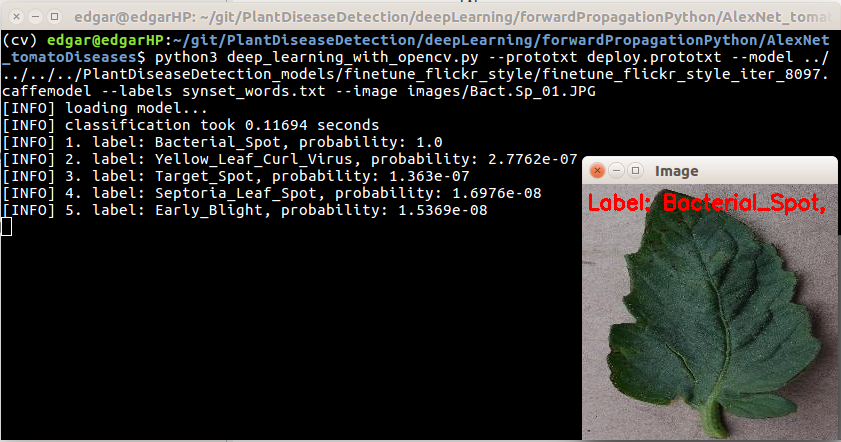


Figura 5.2 Predicción de una hoja sana.



Figura 5.3 Predicción de una hoja sana.

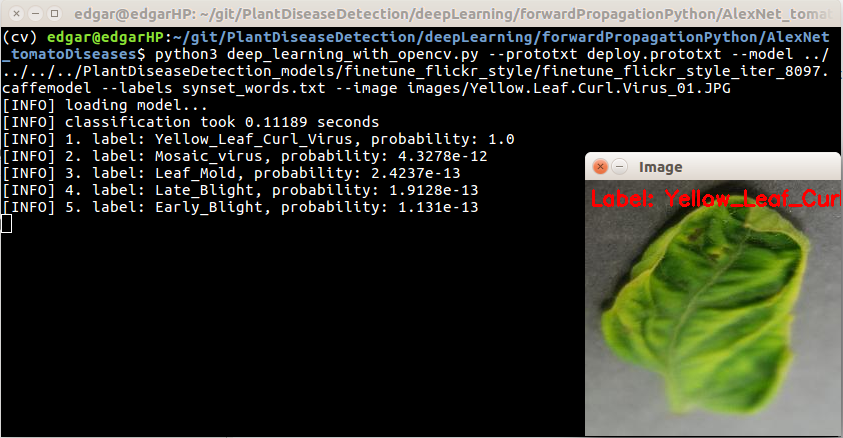


Figura 5.4 Predicción de una hoja con el virus del rizado amarillo del tomate.

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 99.66% (Ver figura 5.5).

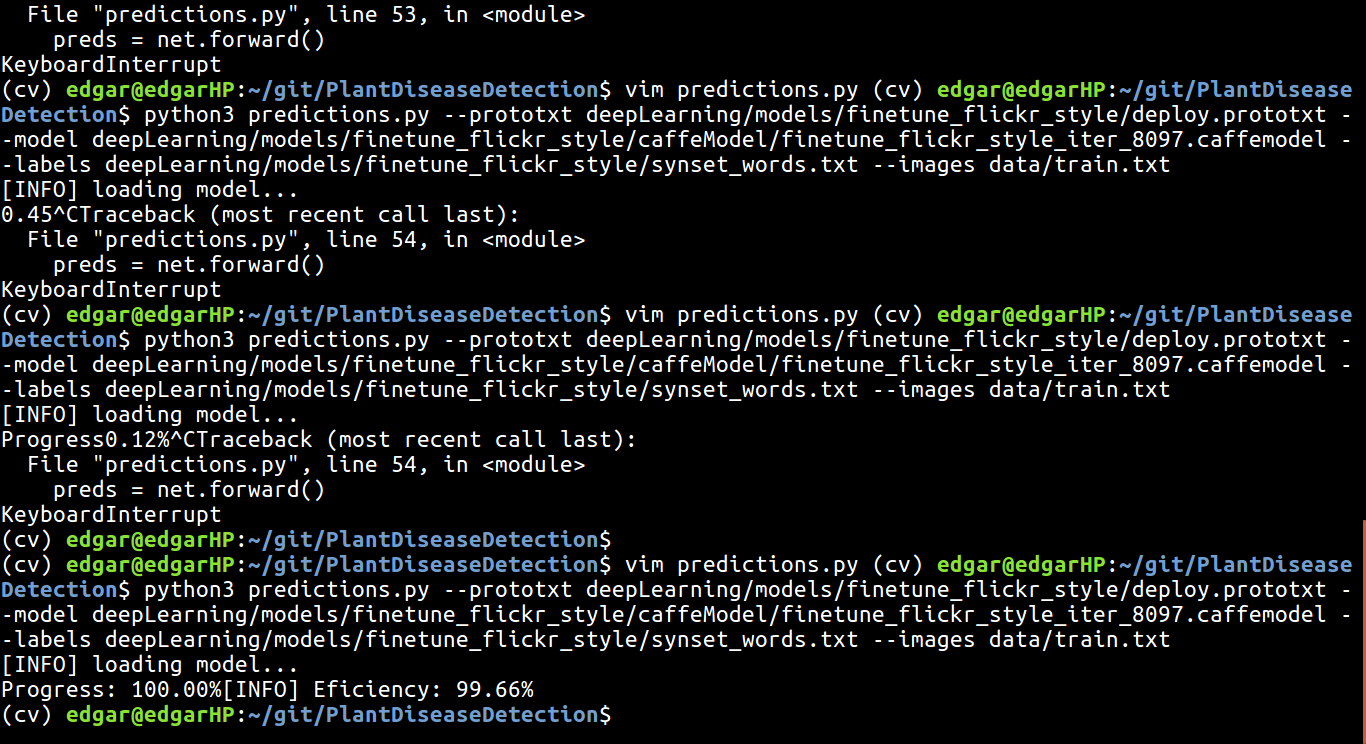


Figura 5.5 Eficiencia del clasificador

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.

Análisis del sistema Web.

En esta sección se describen los requisitos de hardware y software, se presenta un modelos 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 3.1, 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 forward propagation en la red para realizar las predicciones, tarea sencilla que se ejecuta en menos de un segundo, de acuerdo a 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.

6.2 Modelo de casos de uso.

Desde el punto de vista del usuario, la aplicación le permitirá hacer las siguientes acciones. 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 la misma en alguna de las nueve categorías mostradas en la tabla 3.1 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 a 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.

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 hacía 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. [15]. Esto se ilustra en la figura 7.2.

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 7.2.

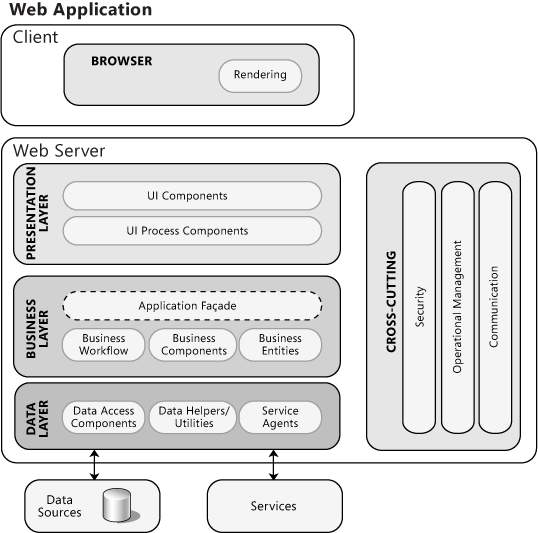


Figura 7.1 Arquitectura típica de un sistema Web.

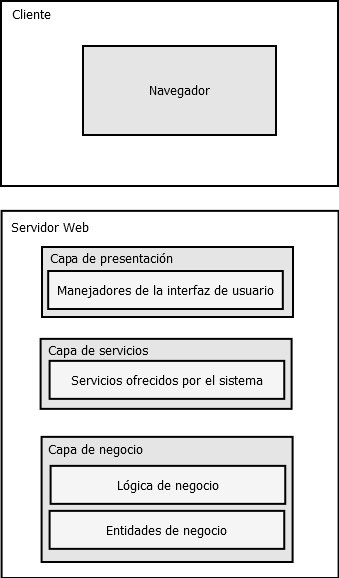


Figura 7.2 Arquitectura del sistema.

7.2 Diagrama de clases.

Con base en la arquitectura anterior, en la figura 7.3 se presenta el diagrama de clases del sistema.

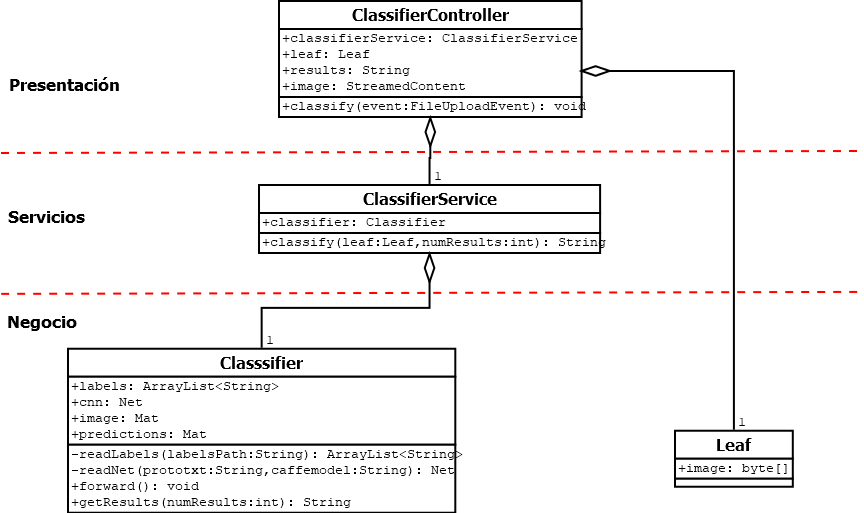


Figura 7.3 Diagrama de clases del sistema.

7.3 Tecnologías.

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.

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.

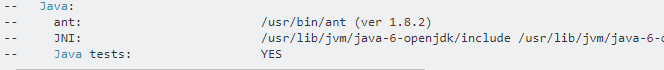


Figura 8.1 Configuración de OpenCV para la integración con Java.

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.

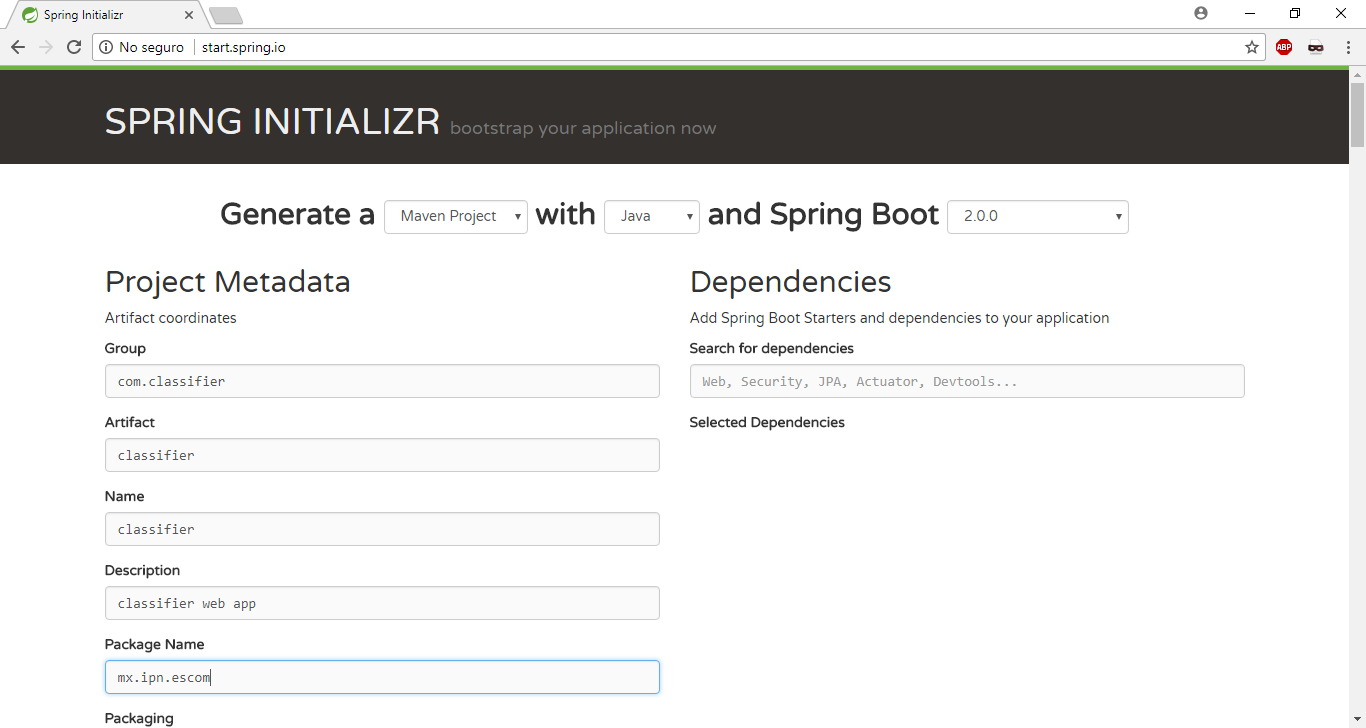


Figura 8.2 Generación del proyecto Maven.

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 direccionada 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

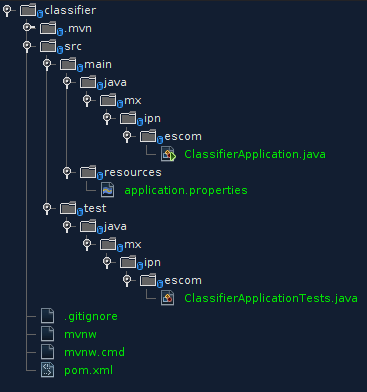


Figura 8.3 Proyecto de Spring Initializr importado en NetBeans.

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.

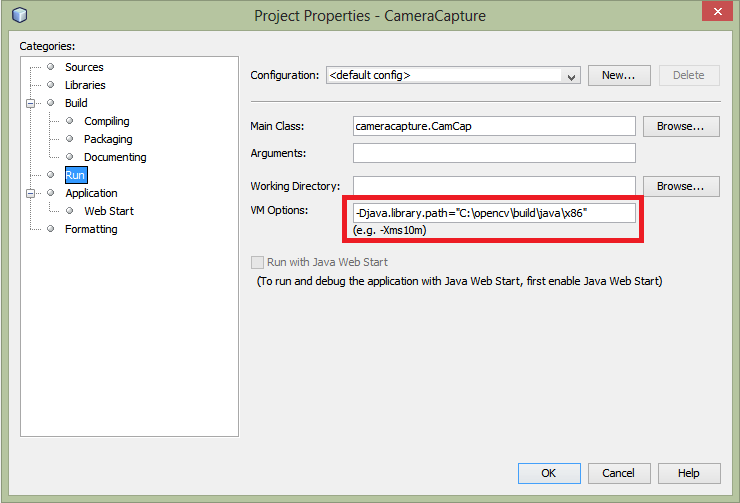


Figura 8.4 Ubicación de la biblioteca nativa de OpenCV.

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

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 [16].

PrimeFaces es un framework de componentes para JSF con 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 pagina JSF que se solicita. Una vez construido el árbol de componentes se ejecuta el código Java en el servidor para rellenar 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.

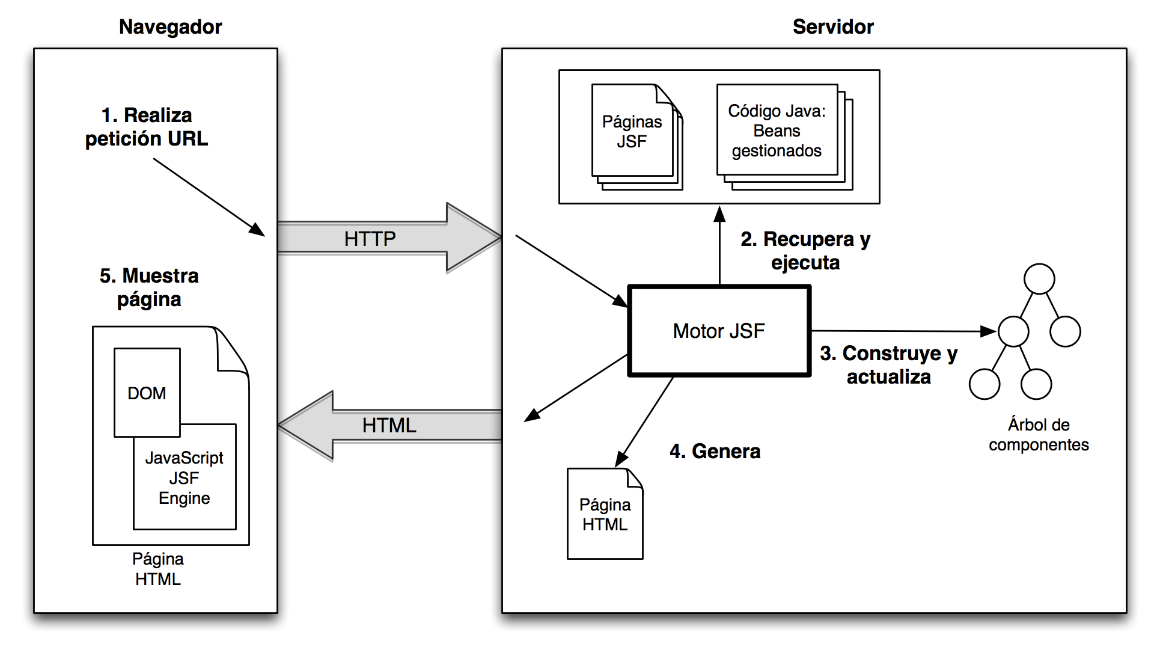


Figura 8.5 Funcionamiento de JSF.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <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.primefaces</groupId>  <artifactId>primefaces</artifactId>  <version>6.1</version>  </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.SpringBeanFacesELResolver</el-resolver>

</application>

</faces-config>

Lo que se realiza en el archivo faces-config.xml es registrar un Expresion Language resolver (ELResolver) que delega a Spring la responsabilidad de resolver el nombre de los beans a los que se haga referencia en las paginas 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.

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. Mohanty S., Hughes D., Salathé M. Using Deep Learning for Image-Based Plant Disease Detection. 2016.
4. Moujahid A. A practical Introduction to Deep Learning with Caffe and Python. [En línea] Disponible en http://adilmoujahid.com
5. CS231n. Convolutional Neural Networks for Visual Recognition. [En linea] Disponible en http://cs231n.github.io/convolutional-networks/
6. Szegedy et al. Going Deeper with Convolutions. [En línea] Disponible en: https://arxiv.org/abs/1409.4842
7. Mohamed Brahimi. Computer Science Department. Mohamed El Bachir El Ibrahimi University.
8. CS231n. Transfer Learning. [En línea] Disponible en http://cs231n.github.io/transfer-learning/
9. Caffe. Deep Learning Framework. [En línea] Disponible en http://caffe.berkeleyvision.org/
10. BVLC reference caffenet. [En línea]. Disponible en https://github.com/BVLC/caffe/tree/master/models/bvlc\_reference\_caffenet.
11. 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.
12. Rosebrock A. Ubuntu 16.04: How to install OpenCV. [En línea] Disponible en https://www.pyimagesearch.com.
13. Rosebrock A. Deep Learning with OpenCV. [En línea] Disponible en https://www.pyimagesearch.com.
14. BVLC. Using a Trained Network: Deploy. [En línea] Disponible en https://github.com/BVLC/caffe/wiki/Using-a-Trained-Network:-Deploy.
15. Microsoft Developer Network. Designing Web Applications. [En línea] Disponible en https://msdn.microsoft.com/.
16. Universidad de Alicante. Introducción a JavaServer Faces. [En línea] Disponible en http://www.jtech.ua.es/