UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Reconocimiento del estado de cocción de la carne vacuna a través de técnicas de procesamiento de imágenes por computadora

Autores: Harguindeguy Juan Ignacio
Director: Waldo Hasperué
Codirector: Ronchetti Franco
Carrera: Licenciatura en Sistemas

Resumen

Existen diversas investigaciones relacionadas a los alimentos, pero ninguna relacionada con la detección de estados de cocción en carne vacuna a través del procesamiento de imágenes. Por este motivo se estudiaron las Máquinas de Soporte Vectoriales (SVM) y su utilización con diversos descriptores tales como Haralick, Local Binary Pattern y Histogram of Oriented Gradients. Además, se ahondó sobre la técnica de Deep Learning, utilizando diversos modelos, algunos modelos avanzados y otros modelos simples a medida, de esta técnica para poder compararlos con las SVM.

Se desarrolló un conjunto de datos con el cual poder entrenar los distintos modelos. A este conjunto de datos se le aplicó un preprocesamiento y una clasificación manual de las imágenes para lograr las diversas clases que se deseaban detectar, "Crudo", "Jugoso", "A Punto", "Cocido" y "Bien Cocido". Por último, se implementó una plataforma para probar estos modelos de regresión haciéndolos accesibles a través de Internet. Para lograr esto se utilizó una plataforma Platform as a Service (también conocido como PaaS), que permitió el acceso de la aplicación backend que consume el modelo creado para la "Puesta en Marcha" de la plataforma. Este servicio, luego, es consumido por una aplicación móvil desarrollada con una tecnología híbrida para poder utilizarla en tanto en sistemas Operativos Android como iOS.

Palabras Claves

Inteligencia Artificial, Machine Learning, Deep Learning, Mobilenet, Support Vector Machine, descriptores, Histogram of Oriented Gradients, Local Binary Pattern, Haralick, PaaS.

Conclusiones

Se demostró que es posible realizar la detección de estados de cocción en cortes de carne vacuna mediante técnicas de Machine Learning. Los modelos obtenidos con la técnica de Deep Learning presentan mejores resultados, en comparación a las SVM, a costa de la necesidad de una mayor potencia de cómputo y el tiempo que consume el entrenamiento de estos modelos. A su vez, se logró tener una versión de "producción" del mejor modelo obtenido a través del desarrollo de la plataforma basada en tecnología Cloud, el uso de frameworks para desarrollo web y de desarrollo móvil híbrido.

Trabajos Realizados

Se analizaron técnicas de procesamiento de imágenes dentro del campo de Machine Learning, logrando un modelo capaz de clasificar 5 estados de cocción con un error absoluto medio de 0.31. Además, se desarrollaron aplicaciones que dan soporte para el consumo de este modelo clasificador y se demostró que se puede realizar una versión de producción de esta prueba de concepto a través de internet, mediante uso de plataformas en la nube. Por último, se generó una base de datos de imágenes para futuras investigaciones que sean relativas al análisis de cortes de carne vacuna.

Trabajos Futuros

Analizar el funcionamiento de la red InceptionV3 de Google para analizar otra red avanzada en el campo de Deep Learning. Además, se podría aumentar y optimizar el dataset generado para analizar el comportamiento de los modelos utilizados con esta mejora del conjunto de datos. También, se podría aumentar la funcionalidad de las aplicaciones que dan soporte al modelo clasificador.

Fecha de la presentación: Mayo 2020

Agradecimientos

Quiero expresar en este espacio, el agradecimiento que le tengo a todas aquellas personas que me ayudaron de un modo u otro, durante toda esta etapa donde uno termina formándose tanto profesionalmente como personalmente.

Especiales agradecimientos a Waldo y Franco, por haber dedicado su tiempo a ayudarme a llevar a cabo este trabajo, por tener la paciencia y darme la motivación necesaria para poder concluirlo.

Muchas gracias a la Facultad de Informática y a todo el organismo de la UNLP por ser la Institución que son, formando personas de la mejor manera posible, siempre buscando la innovación y la mejora continua.

Por último, quería agradecer particularmente a toda mi familia y amigos, por darme el apoyo necesario en los momentos en que uno está dubitativo, donde quiere tirar la toalla, prestándome sus orejas, motivándome y lo mejor de todo, pasando muy bellos momentos entre todos. Muchas muchas gracias.

PD: *Eskerrik asko aitona, izan zinen bezalakoa izateagatik, beti gogoratuko zaitut irribarre handi batekin aurpegian, unerik ederrenekin eta baso bat ardorekin. Poztuko nintzatekeen zu nire etxean egon izan bazina, baina orain lasai eta pozik zaude zauden lekuan. Amonatxo maite zaitut.*

Eternamente agradecido, Juani.

Índice

Índice 3

Índice de Ilustraciones 5

Capítulo 1: Introducción 8
- 1.1 Objetivos 8
- 1.2 Motivación 8
- 1.3 Desarrollos Obtenidos 10
- 1.4 Estructura del Trabajo 11

Capítulo 2: Marco Teórico 13
- 2.1 Introducción 13
- 2.2 Conceptos 13
  - 2.2.1 Inteligencia Artificial (Artificial Intelligence) 13
  - 2.2.2 Aprendizaje Automático (Machine Learning) 14
    - 2.2.2.1 Descriptores 19
      - 2.2.2.1.1 Patrón Local Binario (Local Binary Pattern) 20
      - 2.2.2.1.2 Haralick 23
      - 2.2.2.1.3 Histograma de Gradientes Orientados (HOG) 25
    - 2.2.2.2 Aprendizaje Supervisado 36
      - 2.2.2.2.1 Máquinas de Soporte Vectorial (Support Vector Machines) 37
      - 2.2.2.2.2 Redes Neuronales (Neural Networks) 39
    - 2.2.2.3 Aprendizaje Transferido (Transfer Learning) 53
  - 2.2.3 Aprendizaje Profundo (Deep Learning) 55
- 2.3 Frameworks y Librerías 62
  - 2.3.1 OpenCV 62
  - 2.3.2 TensorFlow 62
  - 2.3.3 Keras 63
  - 2.3.4 Ionic 63
  - 2.3.5 Django 64
- 2.4 Estado del Arte 64
- 2.5 Revisión del objetivo 66

Capítulo 3: Trabajos Realizados 67
- 3.1 El Dataset 67
- 3.2 La aplicación Django, el backend 69
- 3.3 La aplicación Ionic, el frontend 71
- 3.4 El modelo de clasificación 73

3.4.1 Support Vector Regression 74
3.4.1.1 Local Binary Pattern (LBP) 74
3.4.1.2 Haralick 77
3.4.1.3 Histogram of Oriented Gradients (HOG) 80
3.4.2 Deep Learning 83
3.4.2.1 Modelo Simple 87
3.4.2.2 Modelo Two Lanes 91
3.4.2.3 Modelo MobileNet 93
3.5 Puesta en marcha 99

Capítulo 4: Conclusiones 105

Capítulo 5: Trabajos Futuros 108

Capítulo 6: Referencias Bibliográficas 110

Apéndice A 115
Instalación de Frameworks y Librerías 115
Ionic 115
Aplicación Móvil 115
Instalación 115
Testing 116
Django 117
Aplicación Backend 120
Instalación 120
Testing 121
TensorFlow 122
Keras 123
Instalación de controladores de gráficos NVIDIA 123
CUDA 124
cuDNN 124
Instalación vía TAR 124
Instalación vía DEB 124
Verificación 125

Índice de Ilustraciones

Figura 1.1 Esquema de Inteligencia Artificial y sus subcampos. Aprendizaje Automático, Redes Neuronales y Aprendizaje Profundo 14
Figura 1.2 Aprendizaje Automático: un nuevo paradigma 15
Figura 1.3 Datos de muestra 17
Figura 1.4 Cambio de representaciones 17
Figura 1.5 Operación de LBP con una matriz de adyacencia de 3x3 20
Figura 1.6 Procesamiento de la matriz LBP y cálculo de su valor decimal 21
Figura 1.7 Almacenamiento del resultado computado en la matriz LBP resultante 21
Figura 1.8 Imágenes en escala de grises para utilizarlas con el descriptor LBP 22
Figura 1.9 Imagen resultante luego de computar la representación *LBP* 22
Figura 2.0 Representación gráfica de la extensión del algoritmo LBP 23
Figura 2.1 Matriz de Coexistencia de Nivel de Grises (GLCM) 24
Figura 2.2 Primer paso del algoritmo HOG, redimensionamiento 26
Figura 2.3 Datos del método shape de opencv, retorna número de filas, columnas y canales 27
Figura 2.4 Kernel horizontal y vertical respectivamente para filtrar la imagen 27
Figura 2.5 Cálculo de los gradientes de la imagen redimensionada 27
Figura 2.6 Diversas representaciones de los gradientes 28
Figura 2.7 Cálculo de gradientes de un vector numpy de punto flotante 32 29
Figura 2.8 Muestra de los vectores de magnitudes y direcciones de los gradientes 30
Figura 2.9 Representación visual de los vectores de magnitudes y direcciones 30
Figura 3.0 Representación visual de skimage HOG 31
Figura 3.1 División del vector en celdas de 8x8 y 16x16 respectivamente 32
Figura 3.2 Representación de la celda con los vectores de magnitudes y direcciones 33
Figura 3.3 Cálculo del histograma por cada celda 34
Figura 3.4 Representación de la normalización del histograma para mejorar el acierto 35
Figura 3.5 Interpretación gráfica del clasificador SVC con distintos kernels 37
Figura 3.6 Hiperplano óptimo entre 2 clases 38
Figura 3.7 Visualización de *kernel trick* para solucionar conjuntos inseparables 39
Figura 3.8 Representación de la estructura de una neurona 40
Figura 3.9 Elementos de una neurona artificial 42
Figura 4.0 Red *feedforward* de 1 capa 43
Figura 4.1 Funciones de error 45
Figura 4.2 Superficie del error cuadrático para una neurona lineal 46

Figura 4.3 Visualización de los errores cometidos en cada iteración de *GD* 47
Figura 4.4 Variación en tasas de aprendizaje 48
Figura 4.5 Movimiento Zig Zag 49
Figura 4.6 Estructura del perceptrón 50
Figura 4.7 Representación gráfica de la convolución 52
Figura 4.8 Cálculo de la convolución con y sin *Padding* 52
Figura 4.9 Representación del funcionamiento en capas de las neuronas 55
Figura 5.0 Gráficas de las funciones ReLu, *tanh, sigmoid* respectivamente 57
Figura 5.1 Representación de cómo se utiliza la retroalimentación para ajustar los pesos 58
Figura 5.2 Ejemplo de un slice de profundidad 3 en entrada *RGB* 60
Figura 5.3 Funcionamiento de una capa *Max Pooling* con stride 2 61
Figura 5.4 Ejemplo de Dropout con una capa fully-connected 62
Figura 5.5 Esquema de funcionamiento de Ionic 64
Figura 5.6 Arquitectura general del sistema planteado 67
Figura 5.7 Categorías del dataset 68
Figura 5.8 Diccionario de clases para el clasificador 69
Figura 5.9 Modelo que representa una imagen con su resultado de clasificación 70
Figura 6.0 Serializador de datos 70
Figura 6.1 Definición de un *endpoint* particular usando viewsets 71
Figura 6.2 Inicio de la aplicación móvil y muestra de la funcionalidad de la cámara 72
Figura 6.3 Muestra de los diálogos y pantallas que muestra la aplicación 73
Figura 6.4 Métricas obtenidas del modelo *SVR* con el descriptor *LBP* aplicando la técnica de *fine tuning* 75
Figura 6.5 Histogramas del set de datos utilizado para entrenar el modelo 77
Figura 6.6 Histogramas obtenidos del descriptor Haralick 78
Figura 6.7 Métricas obtenidas del modelo *SVR* con el descriptor Haralick 79
Figura 6.8 Métricas obtenidas del modelo *SVR* con el descriptor *HOG* aplicando la técnica de *fine tuning* 82
Figura 6.9 Visualización de los gradientes calculados por cada categoría 82
Figura 7.0 Arquitectura de la red convolucional MobileNet 85
Figura 7.1 Arquitectura del segundo modelo utilizado denominado "*Two Lanes*" 85
Figura 7.2 Arquitectura del modelo simple 86
Figura 7.3 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo simple con parámetros por defecto 88
Figura 7.4 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo simple con tasa de aprendizaje de 0,001 89
Figura 7.5 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo simple con tasa de aprendizaje de 0,001 90
Figura 7.6 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo *Two Lanes* con tasa de aprendizaje de 0,001 92

Figura 7.7 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo *MobileNet* con tasa de aprendizaje de 0,01 con *TL* 94
Figura 7.8 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo *MobileNet* con tasa de aprendizaje de 0,001 con *TL* 95
Figura 7.9 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación en 50 épocas del modelo *MobileNet* con tasa de aprendizaje de 0,001 sin *TL* 96
Figura 8.0 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación en 100 épocas del modelo *MobileNet* con tasa de aprendizaje de 0,001 sin *TL* 97
Figura 8.1 Esquema de funcionamiento de la aplicación en ambiente de producción 100
Figura 8.2 Logs de la aplicación cuando se solicita evaluar una imagen a la aplicación alojada en Heroku 101
Figura 8.3 Procesamiento de una imagen de un corte en estado "Crudo" 102
Figura 8.4 Procesamiento de una imagen con un corte en estado "A Punto" 103
Figura 8.5 Procesamiento de una imagen con un corte en estado "Cocido" 104
Figura 8.6 Inicio de sesión autogenerado 118
Figura 8.7 Sitio de administración autogenerado 118
Figura 8.8 Aplicaciones instaladas en el archivo settings.py 119
Figura 8.9 Archivo serializers.py 119
Figura 9.0 Archivo views.py con la vista genérica de django-rest-framework 119
Figura 9.1 Archivo urls.py con las vistas generadas en el paso anterior 120
Figura 9.2 Interfaz de Postman 121
Figura 9.3 Respuesta del servidor en caso de no enviar una foto 121
Figura 9.4 Petición y respuesta a la API *backend* 122

Capítulo 1: Introducción

1.1 Objetivos

El objetivo general de esta tesina es crear una aplicación móvil que pueda identificar el estado de cocción de una pieza de carne mediante el análisis de una fotografía. Para realizar este análisis se estudiarán e investigarán técnicas de inteligencia artificial para conseguir un modelo que sea capaz de clasificar imágenes de carne vacuna en distintos estados de cocción (crudo, jugoso, a punto, cocido y bien cocido).
Se proponen los siguientes objetivos específicos:

- Generar un *dataset* propio con imágenes de cortes vacunos en distintos estados de cocción en diversos ambientes, con variedad lumínica y diversidad en el modo de cocción.
- Investigar las técnicas actuales utilizadas en aprendizaje automático (específicamente, *deep learning*) y técnicas de procesamiento de imágenes para detección de texturas.
- Comparar las técnicas analizadas para determinar cuál se adecúa mejor a la problemática.
- Desarrollar una aplicación móvil para poder utilizar el sistema de clasificación desarrollado.
- Validar el modelo generado con fotos tomadas desde la aplicación.

1.2 Motivación

Comer es una necesidad básica para todas las personas. Desde los comienzos de la humanidad, el ser humano siempre tuvo que alimentarse para poder vivir. Con el pasar del tiempo se fueron refinando los métodos de cocción y los paladares de las personas; el comer pasó de ser una necesidad a un regocijo, un placer, un arte para el ser humano. A la hora de cocinar carne vacuna, por ejemplo, es muy importante detectar su estado de cocción, ya que esto nos permite asegurarnos diferentes factores, tales como:

- La eliminación de bacterias y platelmintos parásitos que puede tener el corte en cuestión, tales como la *Escherichia coli*, la *Taenia solium* y la *Taenia saginata*, respectivamente, o la eliminación de la *Salmonella*.
- La jugosidad, que es una característica apreciable para ciertos paladares.

- La palatabilidad de la carne, permitiendo adecuarnos al gusto de cada persona.
- La fácil digestión, previniendo algún tipo de patología en el sistema digestivo.
- La eliminación de los antinutrientes, como por ejemplo el ácido úrico.

Por otra parte, como se menciona en [1], “De acuerdo con las estimaciones de la OMS, en 2010, 285 millones de personas del mundo entero sufrían discapacidad visual y 39 millones de ellas eran ciegas”. Según la OMS (Organización Mundial de la Salud), actualmente 188,5 millones de personas tienen una deficiencia visual moderada, 217 millones tienen una deficiencia visual de moderada a grave y 36 millones son ciegas [16]. Además, según [17], el 8% de la población masculina y el 1% de la femenina padecen de algún tipo de daltonismo.

El sector de la población, previamente mencionado, posee dificultades a la hora de cocinar carne. El no poder distinguir los distintos estados de cocción les imposibilita reconocer los factores de interés expuestos anteriormente, además de que no les permite ahondar en el arte culinario por sí mismos.

En relación con el arte culinario se extrae de [2] una frase respaldada por la UNESCO, “*Sus hallazgos confirman la validez y la importancia de revitalizar el patrimonio cultural en todas sus formas, el cual debe ser preservado, valorizado y transmitido a las futuras generaciones como testimonio de la experiencia y de las aspiraciones humanas, merecedoras según sus derechos a educación y formación de calidad que respete plenamente su identidad cultural*”. El arte culinario ha crecido mucho y tiene tanto significado para las personas que se la considera un factor identificatorio de la sociedad; y por ello, forma parte de la identidad y cultura de la población. Por esto, quizás, las personas con disminución visual u alguna afección de la misma, no se sientan completamente identificadas con sus raíces a causa de sus limitaciones subyacentes.

Es por ello que resulta muy interesante contar con un sistema que permita de manera automática identificar el estado de cocción de los alimentos.

El área de la identificación automática del estado de los alimentos aún no se ha investigado en demasía, como se menciona en [3]. En esa investigación, se analiza la forma física del alimento en cuestión (ya sea cubeteado, en julianas, enharinado, pelado, cremoso, etc).

Otro campo, como la clasificación de la calidad de un corte de carne (en base al nivel y tipo de grasa que tiene, en base al color, la suavidad, etc. [4] [5] [6]) sí ha sido investigado ya que su beneficio es muy importante para la cadena industrial, para lograr mejores niveles de calidad en sus productos, lo que reduce costos sin inutilizar mercadería. En todos los artículos previamente mencionados, se utilizan distintas técnicas de inteligencia artificial para la clasificación de fotografías tomada de distintos métodos: ya sea mediante rayos-x, fotos RGB, espectroscópicas, entre otras, lo que deja lugar a nuevas investigaciones en distintos aspectos sobre los estados de los alimentos, como por ejemplo el estado de cocción.

A lo largo de la historia se han ido desarrollando y probando distintas formas de clasificar imágenes utilizando diversos descriptores (como por ejemplo *SIFT*, *LSD*, *ASM*, *GLCM*, *LDA*, etc) y con varias técnicas de redes neuronales artificiales (*Kohonen ANN*, *Counter-Propagation ANN*, *Back-Propagation ANN*, entre otras). En el trabajo de J. Shmidhuber [59] se hace una revisión de estas técnicas, arribando al último avance tecnológico del campo conocido como redes de aprendizaje profundo

(*Deep Learning Neural Networks*). Estas redes en los últimos años han tenido un crecimiento inmenso dado los resultados que se han logrado en diversos campos de investigación que requieren análisis de imágenes. *Deep Learning* es un subcampo específico de *Machine Learning*, que propone un aprendizaje incremental entre cada capa de la red neuronal, estipulando el incremento de capas totales de la red; lo que conlleva a un mecanismo de múltiples etapas para aprender distintas representaciones de datos. Dentro de las redes profundas, podemos encontrar tres grupos de aprendizaje, estos son: supervisado, no supervisado y aprendizaje reforzado. Los más comunes de estos modos de aprendizaje son el supervisado y el no supervisado.

1.3 Desarrollos Obtenidos

Ya se ha realizado la creación de un *dataset* de imágenes relativas a la problemática. Este contiene fotos de carne vacuna con diversas calidades, en distintos sitios (dentro de una cocina, en una parrilla de exterior, etc.) y con distintas intensidades lumínicas (luz artificial, luz natural, etc.) lo que produjo como resultado un *dataset* con diversidad de entornos y distintas metodologías de cocción de la carne vacuna, como por ejemplo, al horno, a la parrilla, a la plancha, a la cacerola, etc.

Las imágenes del *dataset* se dividieron en cinco categorías que identifican diversos estados de cocción: Crudo, Jugoso, A punto, Cocido y Bien Cocido.

Si bien el grado de cocción de una pieza de carne es subjetivo y no es posible definir con exactitud los límites entre cada categoría, se utilizará este *dataset* para el desarrollo de esta tesina. A futuro se podría utilizar otro *dataset*, o reetiquetar las imágenes de esta base de datos para conseguir nuevos modelos de clasificación.

Este *dataset* contiene 650 imágenes de 224 x 224 píxeles en formato *RGB*, de las cuales se divide el conjunto en 2 conjuntos, uno de entrenamiento y otro de validación, 80% del total para la primera categoría y 20% para la etapa de validación. Esta división nos brinda un conjunto de 150 imágenes por cada una de las categorías (130 imágenes para entrenar y 20 imágenes para validar nuestro clasificador). Además, en la etapa de utilizar los modelos de *Deep Learning* se empleó la técnica de aumentación de datos para generar un volumen mayor con el cual entrenar nuestro modelo.

Se estudiaron e investigaron los modelos de *deep learning* existentes para conseguir uno que se adecúe a las necesidades y así conseguir uno capaz de clasificar de manera aceptable los estados de cocción de la carne mediante el análisis de una imagen digital. Para el estudio y los ensayos se utilizó como *backend* de *machine learning* el *framework* Tensorflow, utilizando como capa de abstracción, para una implementación más intuitiva del modelo, la librería Keras. Por esto, se eligió el lenguaje Python 3.x como lenguaje base para el desarrollo del *backend*.

Se utilizaron distintos algoritmos de la librería OpenCV-Python y Scikit-Learn para la identificación de texturas con el objetivo de detectar el estado de cocción de la carne mediante el análisis de las características obtenidas con dichos algoritmos.

Con los resultados obtenidos con el modelo de *deep learning* y los conseguidos mediante el análisis de las texturas, se realizó un análisis entre ambos para determinar cuál de las 2 técnicas se comportaba de mejor manera con la problemática planteada.

Con los modelos conseguidos se desarrolló una aplicación móvil simple, utilizando el *framework Ionic* para poder portarla a diversos sistemas operativos. Esta aplicación permite tomar una fotografía y utilizando el modelo previamente entrenado, le informará al usuario a qué grupo de los cinco previamente estipulados corresponde la imagen tomada.

De esta manera, se ofrece a la población con disminución visual o con alguna afección de la vista una aplicación que permita identificar el grado de cocción (aproximado) de una pieza de carne mientras se la cocina.

1.4 Estructura del Trabajo

Capítulo 1: En el primer capítulo se busca brindar una idea general de la tesina propuesta. Para finalizar el capítulo 1, se pretende presentar la estructura de esta tesina junto con una guia sobre su contenido y el orden de los capítulos venideros.

Capítulo 2: En el segundo capítulo, se busca presentar el marco general del trabajo para poder brindar un mejor contexto y entendimiento de la investigación. Se definirán términos y conceptos y se proporcionará un poco de la historia de cómo y por qué se ha llegado a las tecnologías que se utilizarán en la investigación del trabajo.

Capítulo 3: En el tercer capítulo, se verá el desarrollo de los procedimientos llevados a cabo en la investigación: su definición, los aspectos del diseño de la arquitectura elegida para llevar a cabo la aplicación deseada y los resultados obtenidos de cada una de las técnicas aplicadas.

Capítulo 4: En el cuarto capítulo, se presentan las conclusiones obtenidas sobre los distintos resultados.

Capítulo 5: En el quinto capítulo, se detallan los trabajos futuros que se podrían realizar a partir de esta investigación en base a los resultados obtenidos en este trabajo.

Capítulo 6: En el sexto capítulo, se brinda un apartado de referencias bibliográficas que se han utilizado durante la investigación de esta tesina.

Apéndice A: En el apéndice, se brinda una guía de la instalación de cada una de las tecnologías utilizadas y de las aplicaciones desarrolladas durante la investigación de esta tesina. Incluye la instalación de los *frameworks* y librerías que dan soporte a los modelos de clasificación, como también, la consola de aumentación, la aplicación móvil y la *API* de *backend*.

Capítulo 2: Marco Teórico

2.1 Introducción

El objetivo de este capítulo es el de brindar un contexto con el cual el lector de este trabajo pueda comprender más fácilmente su desarrollo. Para esto se brindará un contexto histórico de los conceptos más relevantes con los cuales el lector va a lograr una mayor y mejor comprensión de las áreas que involucra el trabajo y sus implicancias. Para lograr esto, debemos dar un marco general sobre la IA (Inteligencia Artificial): cómo nace, el objetivo del área, casos de uso, algoritmos y los avances que se fueron logrando hasta el desarrollo del *Deep Learning*.

Además, se brindará una breve introducción de los *frameworks* utilizados, lo que hará que el lector tenga una mayor comprensión de las herramientas utilizadas, tanto para la aplicación móvil como para la aplicación de procesamiento.

2.2 Conceptos

2.2.1 Inteligencia Artificial (*Artificial Intelligence)*

La Inteligencia Artificial (*AI* por sus siglas en inglés, *Artificial Intelligence*), nace a mediados del siglo XX con el objetivo de intentar recrear la capacidad de pensamiento de los seres humanos. La IA es un campo que abarca tanto al Aprendizaje Automático (*Machine Learning* o su sigla en inglés*, ML)* como al *Deep Learning.* En la figura siguiente, podemos ver una clasificación posible del campo de la Inteligencia Artificial y sus subcampos.

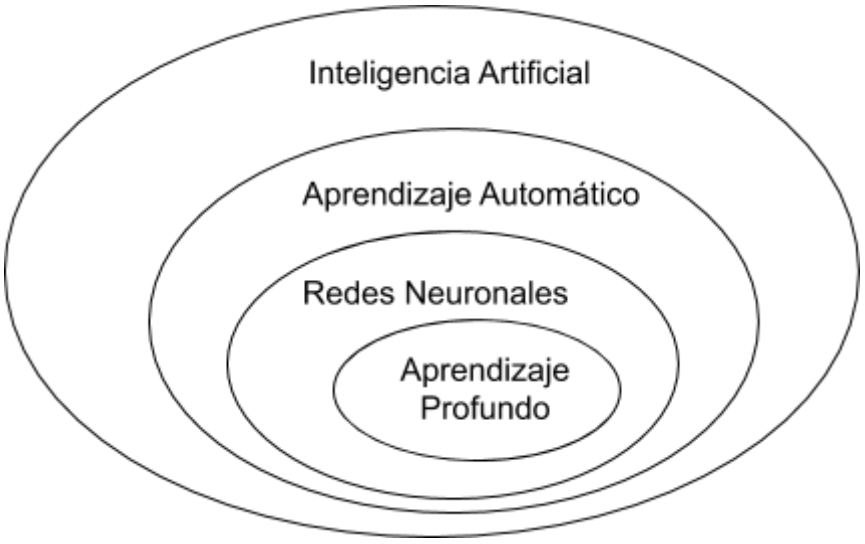

Figura 1.1 Esquema de Inteligencia Artificial y sus subcampos Aprendizaje Automático, Redes Neuronales y Aprendizaje Profundo

Además de los subcampos mencionados, la IA, abarca otro tipo de problemas que no necesariamente supone el aprendizaje automático en las computadoras; por ejemplo, los sistemas expertos (*ES*, *Expert System*) se utilizaron de manera masiva en la década de los 80s. Los *ES* no brindaban ningún tipo específico de aprendizaje automático, se basaban principalmente en un conjunto bien definido de reglas sobre un tópico específico que podría responder de manera eficiente y correcta las consultas de un usuario final que no necesariamente debía ser idóneo en la temática tratada. Este tipo de sistemas, es conocido como *symbolic AI*, y tuvo su auge entre los años 1950 a 1980, culminando en los *ES*; si bien el enfoque que se utilizaba en los *symbolic AI* era adecuado para la resolución de problemas bien definidos y problemas lógicos, no lo era para problemáticas más complejas. En aquellos problemas donde se debían definir reglas con un mayor grado de complejidad, como por ejemplo en lo referente a clasificación de imágenes, reconocimiento de voz o traducción de lenguajes, era casi imposible de resolver con el enfoque utilizado por los *symbolic AI*. Esto dió lugar al nacimiento del *Machine Learning,* un nuevo paradigma que no requería hacer una especificación detallada de las reglas a seguir.

2.2.2 Aprendizaje Automático (*Machine Learning*)

Como mencionamos en la sección anterior, el Aprendizaje Automático fue la respuesta a un cuestionamiento acerca de la posibilidad de que las computadoras pudieran aprender automáticamente, sin tener que escribir a mano las reglas que se usaban para clasificar. Esto abre un nuevo paradigma de la IA que pretende romper

con la programación clásica, en donde el hombre tiene que ingresar las reglas y los datos para ser procesados de acuerdo con las reglas previamente incorporadas. El nuevo paradigma busca que las computadoras puedan generar sus propias reglas de clasificación, catalogando la información según los datos ingresados y las respuestas que el usuario espera obtener en una primera instancia. Luego, las reglas que va generando el sistema autónomo pueden ser utilizadas para clasificar cualquier nuevo conjunto de datos produciendo un nuevo conjunto de respuestas. A continuación podemos ver cómo cambian los tipos de entrada y salida que consumen los programas cotidianos y los programas de Aprendizaje Automático.

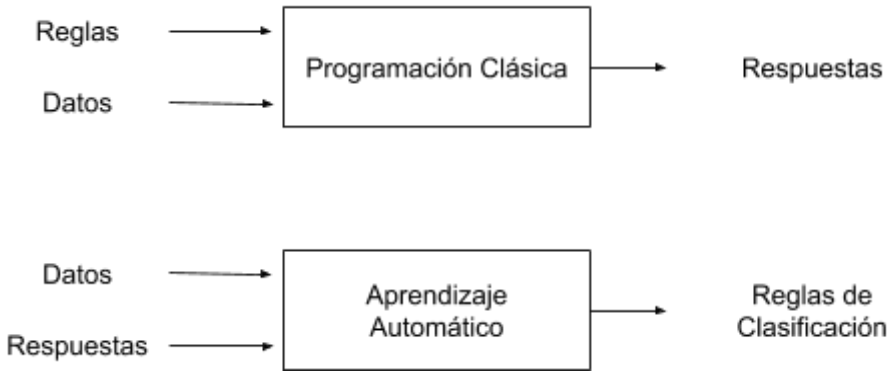

Figura 1.2 Aprendizaje Automático: un nuevo paradigma

Por lo mencionado anteriormente, se dice que los sistemas de aprendizaje automático son entrenados y no programados explícitamente. Estos sistemas logran aprender del conjunto de muestras que el usuario le brinda y logran armar una estructura estadística en base a estos datos que les permite formular reglas para la automatización de la tarea.

Para ejemplificar el funcionamiento de estos sistemas, piense por ejemplo que se desea clasificar fotos de autos en base al color. El usuario del sistema le brinda un conjunto de datos con una etiqueta que representa la categoría de esa imagen al sistema. El sistema aprende que a cierta imagen le corresponde determinada etiqueta, y así sucesivamente hasta lograr formular el conjunto de reglas estadísticas que utilizará posteriormente. Luego de la fase de entrenamiento, el sistema va a ser capaz de poder etiquetar nuevas imágenes en base a las reglas formuladas en la fase de entrenamiento, sin necesitar la ayuda del usuario para que le brinde una etiqueta. Como es de esperar, al ser un conjunto de reglas estadísticas las usadas para clasificar, los resultados de la clasificación de imágenes serán probabilidades.

Como hemos visto, *Machine Learning* está altamente relacionado con la estadística matemática, pero tiene ciertas diferencias. Por ejemplo, en estadística por lo general se utilizan conjuntos de datos no muy grandes, en contrapartida, para entrenar un modelo de *ML*, se requieren conjuntos de datos más robustos, ya que mientras mejor sea la calidad de la información, el sistema, en la etapa de

entrenamiento, va a lograr una mayor exactitud a la hora de clasificar, lo que producirá mejores resultados. Además, el área de *ML* y específicamente el de *Deep Learning* son más bien campos empíricos, que se basan en la prueba y el error, y modifican los modelos que hacen aprender al sistema, mientras que en la estadística tradicional, todo suele ser desde un punto de vista más teórico.

Para poder definir *Deep Learning* y diferenciarlo de otros enfoques de *ML*, debemos explicar el algoritmo general que utiliza un sistema de *ML* para lograr aprender. Como mencionamos en el apartado anterior, estos sistemas aprenden a partir de conjuntos de datos que ingresa el usuario con su respectiva clasificación, por esto podemos decir que para que funcione un sistema de *Machine Learning*, necesitamos:

- Datos de entrada.
- Ejemplos de los resultados esperados.
- Una manera de medir si el algoritmo utilizado está haciendo bien su trabajo o no.

Con este conjunto de herramientas, podemos lograr que un sistema de *Machine Learning* haga las transformaciones pertinentes a un conjunto de entrada de datos y "aprenda" las modificaciones realizadas dentro del procesamiento para poder clasificar de manera correcta el resultado de una nueva clasificación. Es por esto que una parte crucial del *Machine Learning* y del *Deep Learning* consiste en lograr que las transformaciones que haga sobre los datos de entrada sean significativas y útiles. Con esto se quiere decir que las representaciones que hace el sistema, en base a las transformaciones de los datos, sean útiles. Pero, ¿qué es una representación?

Una representación es otra forma de ver la información de entrada que disponemos, otra forma de codificar los datos. Por ejemplo, una imagen tiene diversas representaciones: una de ellas podría ser mediante el sistema *RGB* (*Red-Green-Blue)*, que descompone la imagen como una matriz con las representaciones de colores; por otro lado se podría usar el enfoque *HSV* (*Hue-Saturation-Value)*, que nos permite tener la misma imagen pero en escala de grises. Justamente esto, es la parte fundamental de los modelos de *ML*: deben buscar maneras apropiadas de representar la información brindada por el usuario para que sean de mayor utilidad para la tarea encomendada. En ciertos casos será más adecuado utilizar la imagen en su formato *RGB* y en otros casos donde se quiera analizar saturaciones será mejor utilizar la imagen en *HSV*. Por este motivo se hace hincapié en que debe aprender de representaciones útiles para el problema en cuestión.

Para clarificar lo expuesto previamente, veamos un ejemplo. Nos piden clasificar el conjunto de datos de la Figura 1.3, para poder separar protones de electrones.

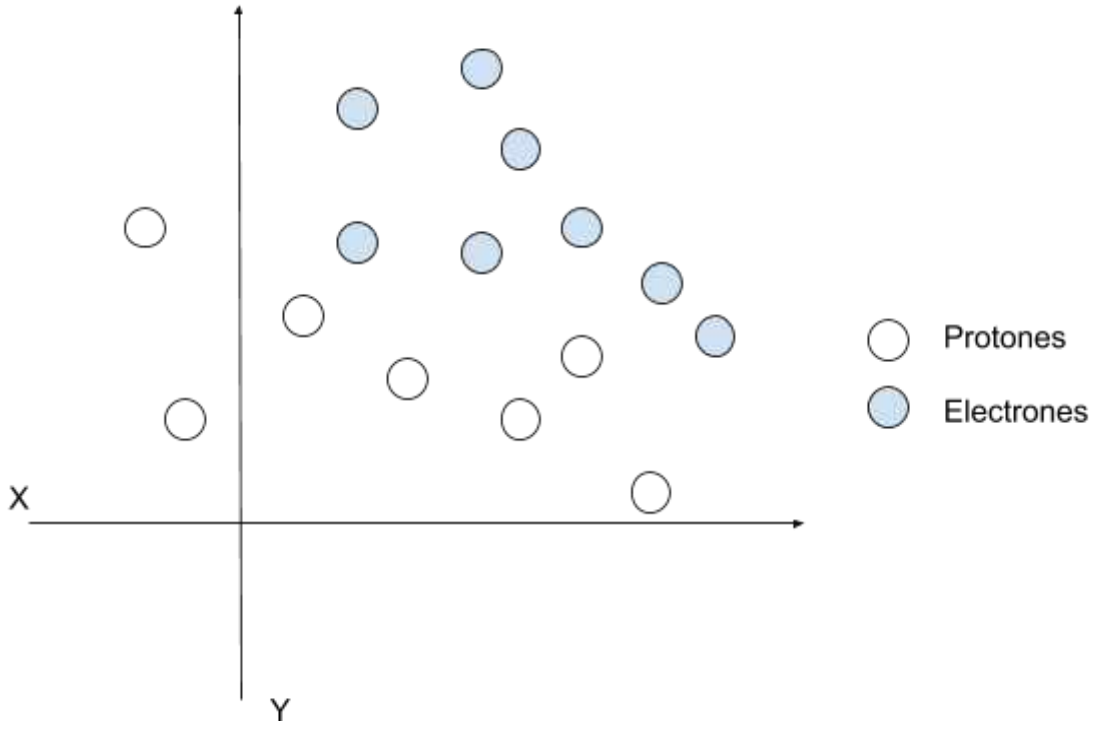

Figura 1.3 Datos de muestra

Para lograr esto, podríamos utilizar diversas funciones que se vayan adecuando a la distribución de los distintos tipos de datos (electrones y protones) y ver cuál de todas las funciones representa mejor este conjunto de datos. Posibles representaciones de los datos podrían ser las que se muestran en la Figura 1.4.

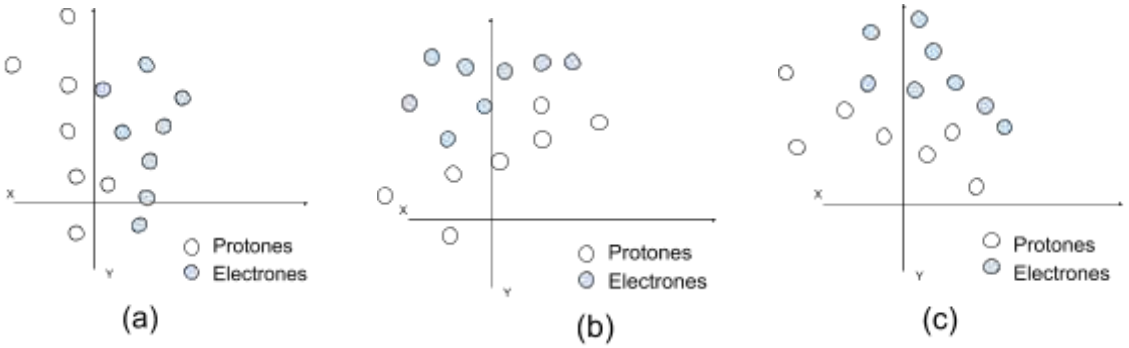

Figura 1.4 Cambio de representaciones

Se puede ver en los gráficos de la Figura 1.4 que la representación más intuitiva (para el ojo humano) para generar una clasificación más sencilla en este caso sería la del primer gráfico (Figura 1.4-a), dado que logra separar casi completamente los protones de los electrones usando el eje *y* como divisor. Aquellas que estén con coordenadas $y < 0$ corresponderán a protones y aquellas que sean superiores a $y > 0$ serán electrones.

En este caso, nosotros hemos brindado distintas representaciones de los datos, pero si, en lugar de que la búsqueda sea manual, se aprendiera del porcentaje de celdas bien clasificadas con cada representación, se podría decir que el sistema sería un sistema de *Machine Learning*.

El aprendizaje, dentro de *ML*, se entiende como un proceso de búsqueda automático de las mejores representaciones posibles. Generalmente, el conjunto de funciones que se aplican para buscar estas mejores representaciones ya vienen predefinidas en el sistema de *ML*; a este conjunto se las suele denominar como *hypothesis space* o en español, espacio de hipótesis.

Generalmente, en la bibliografía relativa a la *IA*, se suele categorizar al *Machine Learning* con 3 distintos enfoques de aprendizaje. Estos enfoques se los conoce como Aprendizaje Supervisado, Aprendizaje No Supervisado y Aprendizaje Reforzado.

Otra forma de catalogar los problemas de *ML* es según el valor que se busca como resultado. Si nuestro problema requiere categorizar cierto dato dentro de valores discretos (como es el típico caso de la problemática donde se busca distinguir los 9 dígitos numéricos a través de imágenes de los números en manuscrita) estamos en presencia de un problema de clasificación. Por el contrario, si nuestro problema requiere de un valor real como resultado (como es el caso de la problemática planteada en esta tesina, ya que queremos distinguir qué tan lejos categoriza nuestro modelo del valor real, y dado que no hay una categoría bien distinguida entre 2 estados posibles, más bien un rango entre estas 2 categorías), se lo suele denominar como problema de regresión.

En esta investigación, haremos uso del enfoque de Aprendizaje Supervisado, ya que nuestro modelo deberá distinguir entre las 5 clases planteadas y estaremos interactuando con un problema de regresión, ya que buscamos valores reales como resultado del procesamiento, para saber si la categorización está más cerca de una categoría o la otra, pero siempre analizando un rango de valores continuos. Es por este motivo, que no se ahondará en el resto de los enfoques de aprendizaje ni en problemas de clasificación.

Ahora bien, una pregunta que puede resonar en la mente del lector de este trabajo es: ¿de qué información aprende el algoritmo de *ML*? Los algoritmos de *ML*, como dijimos previamente, aprenden sobre información procesada, con esto entendemos que la información que se le da como entrada al sistema de *ML* ya fue analizada y se le aplicaron distintas técnicas (cómo aplicarle distintos algoritmos para obtener diversos representaciones o un redimensionamiento del tamaño, etc) para obtener representaciones de posible interés para nuestro modelo. Esto está altamente relacionado con las formas de representación de la información que vimos unos párrafos atrás. Existen algoritmos que nos permiten realizar cierto tipo de operaciones sobre un dato de entrada que nos devuelve una representación que nosotros creemos que será de relevancia para el modelo para que pueda aprender correctamente qué es cada dato. Estos algoritmos, se los suele conocer con el

nombre de Descriptores. En la siguiente sección, se procederá a describir su funcionamiento.

2.2.2.1 Descriptores

Los descriptores retornan una estructura de datos (generalmente representada como histograma) que contiene la información calculada sobre un conjunto de datos en particular. En esta investigación se utilizan tres descriptores distintos. Cada uno de estos calcula cierto tipo de característica de un conjunto de datos: dos de ellos se especializan en detectar texturas en los datos, estos son Haralick y *Local Binary Pattern* (*LBP*). El descriptor restante es un algoritmo ampliamente utilizado para la detección de objetos en imágenes conocido como *Histogram of Oriented Gradients* (*HOG*) .

Previo a ver el funcionamiento de los descriptores, haremos una breve definición de qué es un descriptor de características. Los descriptores capturan información relevante de una imagen o de una porción de ella y desechan la información que no nos aporta nada para el aprendizaje de las características específicas de la imagen. Generalmente, los descriptores de características convierten las imágenes de entrada de un *ancho x largo x 3* (canales) en un vector de características de tamaño n.

Previamente decíamos que los descriptores toman información relevante. Un interrogante que nos puede surgir es: ¿para qué o quién es relevante? Típicamente, estos vectores que se calculan son posteriormente utilizados para entrenar modelos de clasificación como por ejemplo los *Support Vector Regressor* (*SVR*). Estos clasificadores serán definidos en el apartado "2.2.2.2 Aprendizaje Supervisado", ya que fueron utilizados en la investigación.

Para ejemplificar cuándo una característica sería útil, pensemos en el siguiente ejemplo. Si nosotros quisiéramos detectar la presencia de una pelota de fútbol en una imagen, nos sería más relevante obtener características de la forma en lugar del color, por ejemplo, ya que una pelota puede ser de diversos colores, pero siempre va a tener forma esférica. Sumado a esto, nuestro descriptor debería obtener características que logren distinguir además de objetos con distintos contornos, objetos con contornos similares pero que no son lo mismo, ya que si tenemos dos objetos circulares, nuestro clasificador podría interpretar a las dos figuras como pelotas de fútbol cuando la otra quizás haya sido otro objeto, como por ejemplo, una naranja.

Teniendo una noción más clara de qué es un descriptor, comencemos a ver cómo funciona cada uno de los algoritmos utilizados en este trabajo.

2.2.2.1.1 Patrón Local Binario (*Local Binary Pattern*)

Empecemos por ver el descriptor *Local Binary Pattern* o *LBP* por sus siglas en inglés. Este descriptor es un algoritmo capaz de diferenciar texturas en una imagen. *LBP*, a diferencia de otros algoritmos, computa representaciones locales de texturas. Estas representaciones se generan comparando cada píxel de la imagen con una matriz de píxeles adyacentes. El primer paso para usar este descriptor, como lo mencionamos previamente, es convertir nuestra imagen a analizar en una escala de grises. Una vez transformada nuestra imagen a escala de grises, por cada uno de los píxeles en la imagen en escala de grises, seleccionaremos una matriz de adyacencia de tamaño *r* con el píxel a evaluar en el centro de la matriz. Después de esto, se computa el valor *LBP* y se lo guarda en un arreglo de 2 dimensiones con la misma altura y ancho que la imagen inicial. Para ejemplificar esto, veamos la siguiente imagen que representa una operación en una matriz de adyacencia de 3x3.

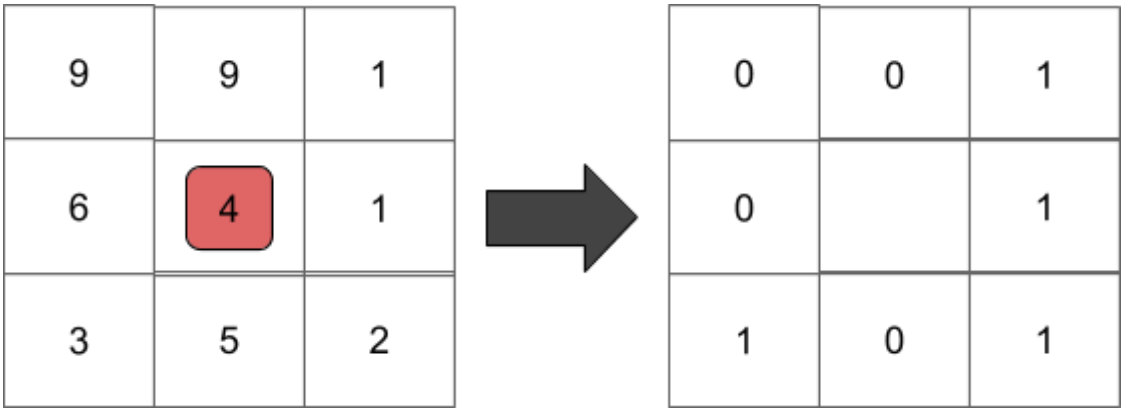

Figura 1.5 Operación de LBP con una matriz de adyacencia de 3x3

Como vemos, en el procesamiento de *LBP* se toma el píxel central de la matriz y se compara con los 8 restantes. La comparación que se evalúa en base al central es que si la intensidad del píxel central es mayor o igual a la de su vecino, en el arreglo resultante se coloca un 1, en caso de que sea menor se coloca un 0. Notar que al haber 8 píxeles circundantes, tenemos un total de 2^8 = 256 combinaciones posibles de códigos *LBP*.

A partir de ahí, debemos calcular el valor *LBP* para el píxel central. Podemos comenzar desde cualquier píxel vecino y avanzar en el sentido de las agujas del reloj o en el sentido contrario, pero nuestro orden debe mantenerse constante para todos los píxeles de nuestra imagen y todas las imágenes en nuestro conjunto de datos. Dado un vecindario de 3 x 3, tenemos 8 vecinos con los que debemos realizar una prueba binaria. Los resultados de esta prueba binaria se almacenan en una matriz de 8 bits, que luego convertimos a decimal, como podemos ver en la siguiente figura:

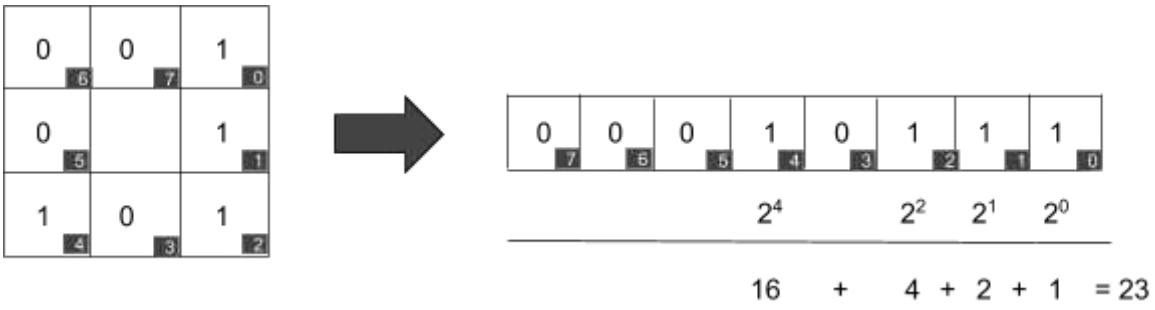

Figura 1.6 Procesamiento de la matriz LBP y cálculo de su valor decimal

Como podemos notar, en este ejemplo decidimos seleccionar como posición inicial el lado superior derecho de la matriz en sentido horario. Este sentido e inicio de la cadena lo debemos mantener durante todo el procesamiento. Una vez completados los valores en nuestro vector, podemos hacer la conversión a decimal clásica. Luego de calcular nuestro valor, en este ejemplo el total es igual a 23, lo almacenamos en la matriz de 2 dimensiones de igual tamaño que nuestra imagen inicial, como lo podemos ver reflejado en la siguiente figura:

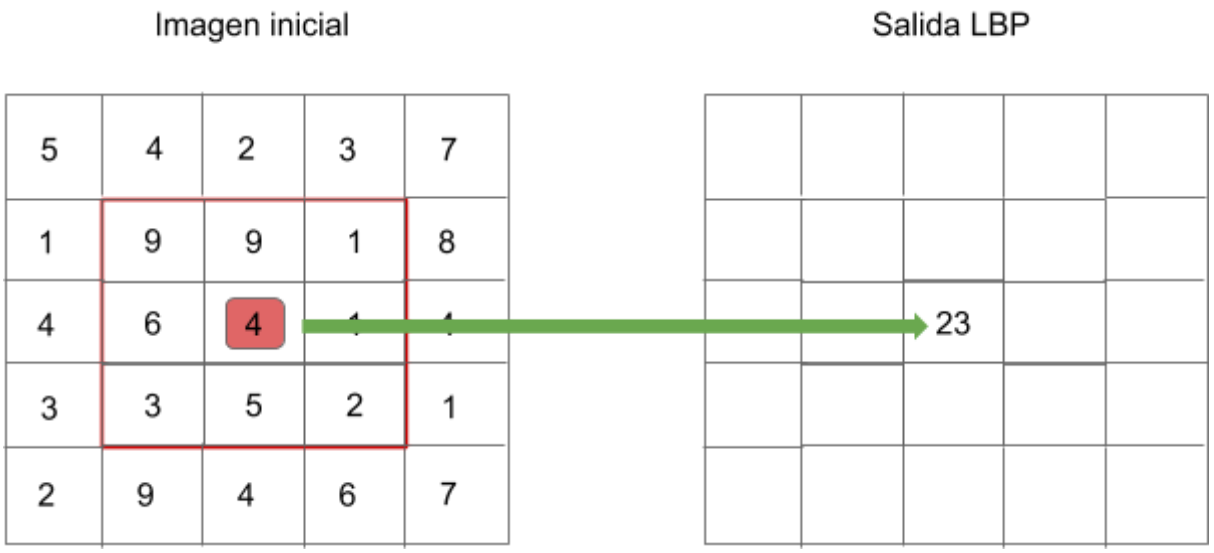

Figura 1.7 Almacenamiento del resultado computado en la matriz LBP resultante

Este mecanismo de crear umbrales, acumular sus valores binarios y luego guardarlos en la matriz de salida *LBP* hay que repetirlo por cada uno de los píxeles de la imagen fuente. En la Figura 1.8, podemos ver las imágenes en su estado en escala de grises, la cual se utiliza en el primer paso del cálculo de la matriz *LBP*.

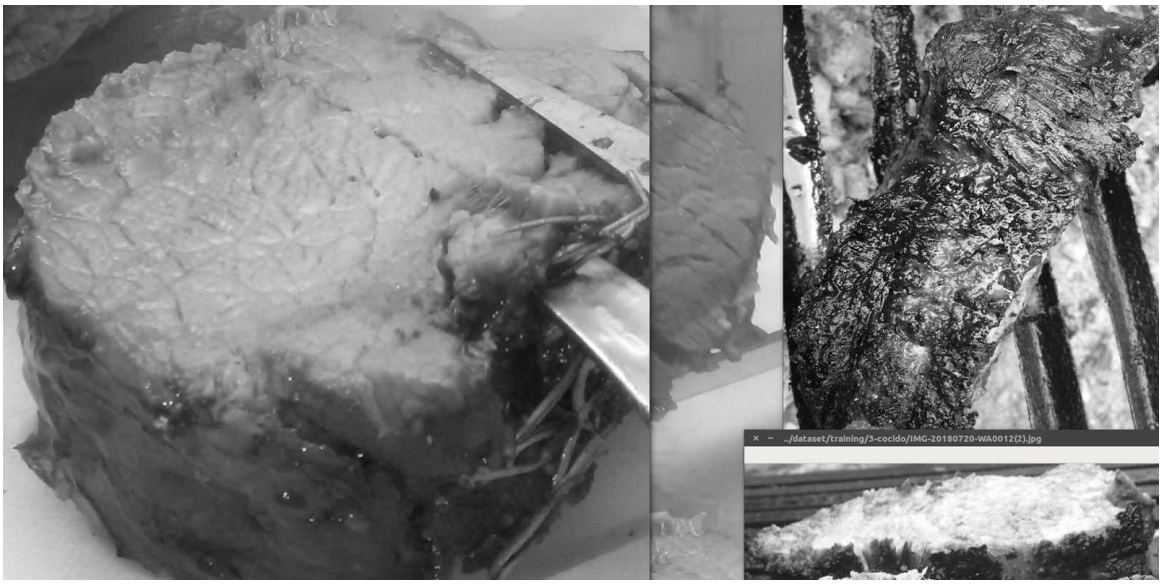

Figura 1.8 Imágenes en escala de grises para utilizarlas con el descriptor LBP

El paso siguiente de nuestro algoritmo para obtener las características es realizar las operaciones pertinentes para lograr obtener nuestra matriz 2D *LBP*. Utilizando la librería scikit-learn, utilizamos del módulo *feature* el método *local_binary_pattern*, el cual computa la matriz *LBP*. Para tener una representación visual del cómputo realizado por el descriptor y así lograr un mejor entendimiento del funcionamiento del algoritmo, podemos utilizar la matriz computada y visualizarla como se muestra en la imagen siguiente.

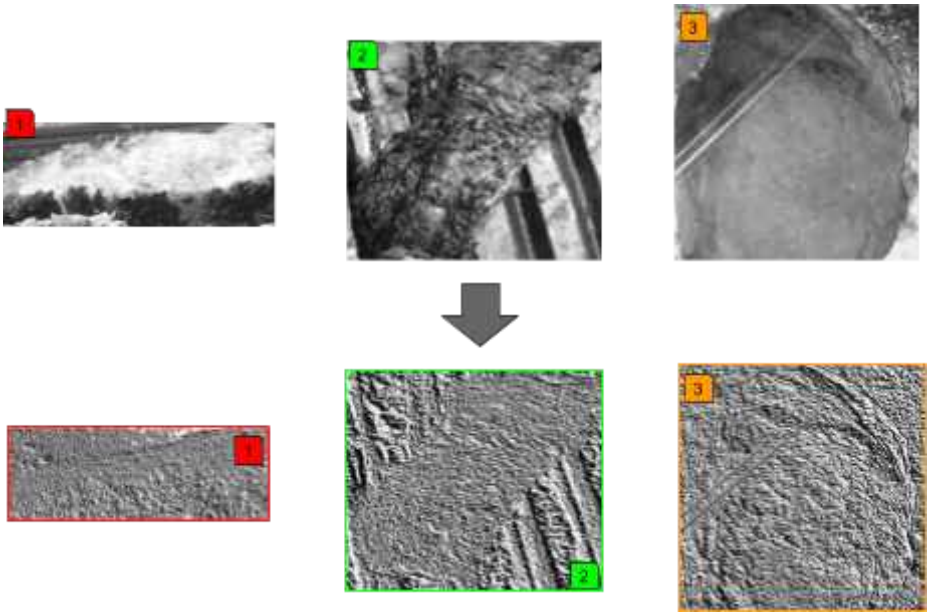

Figura 1.9 Imagen resultante luego de computar la representación *LBP*

Podemos intuir que este tipo de descriptores es bueno para detectar detalles muy finos ya que analiza las submatrices 3x3 de la imagen. Este es uno de los beneficios principales del algoritmo. Sin embargo, ser capaz de capturar detalles a una escala tan pequeña también es uno de los inconvenientes más destacables del

algoritmo ya que no nos permite capturar detalles en escalas mayores. Por este motivo, se generó una extensión del algoritmo que permite analizar un área mayor de la imagen por cada iteración. Para lograr esto, se introdujeron dos parámetros a la función: la cantidad de puntos y el radio del círculo. Básicamente, se planteó analizar un área denotada por una circunferencia que estaba generada por la cantidad de puntos y el radio que se pasan como parámetro. Esta extensión del algoritmo se la conoce también como PLB Circular ó *Circular LBP*. Esta idea la podemos clarificar con un esquema como el siguiente:

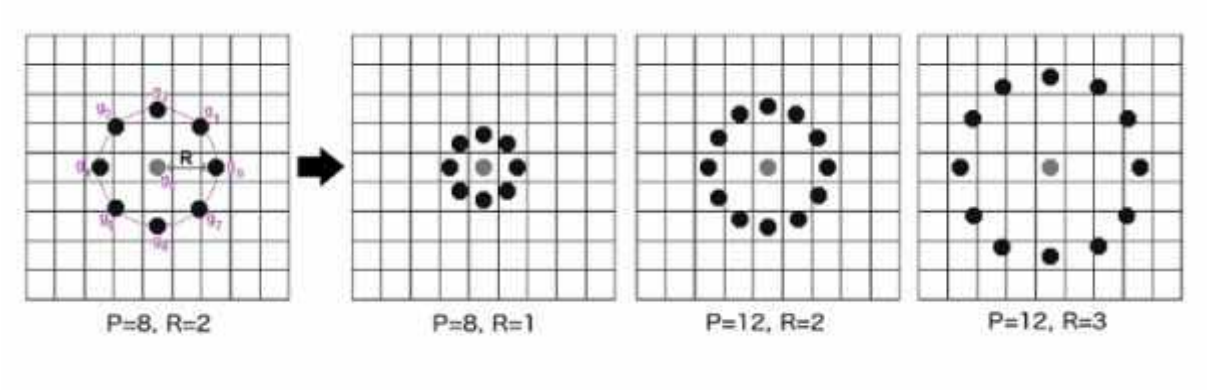

Figura 2.0 Representación gráfica de la extensión del algoritmo LBP

Esta versión del algoritmo es la que se utiliza (*Circular LBP*) en esta investigación, con imágenes en escala de grises y de tamaño fijo de 224 x 224 píxeles. Con esto finalizamos la descripción del algoritmo *LBP*, continuaremos viendo el funcionamiento del descriptor Haralick.

2.2.2.1.2 Haralick

Otro descriptor que se utilizó para ver su funcionamiento en la problemática planteada fue el algoritmo Halarick. Este descriptor está basado en el concepto de *Gray Level Co-occurrence Matrix*. La matriz de coexistencia de nivel de grises (*GLCM* por sus siglas en inglés) utiliza el concepto de adyacencia en las imágenes. La idea básica es que busca pares de valores de píxeles adyacentes que se producen en una imagen y continúa guardandolos en toda la imagen. Este concepto lo podemos comprender mejor con un gráfico como el siguiente:

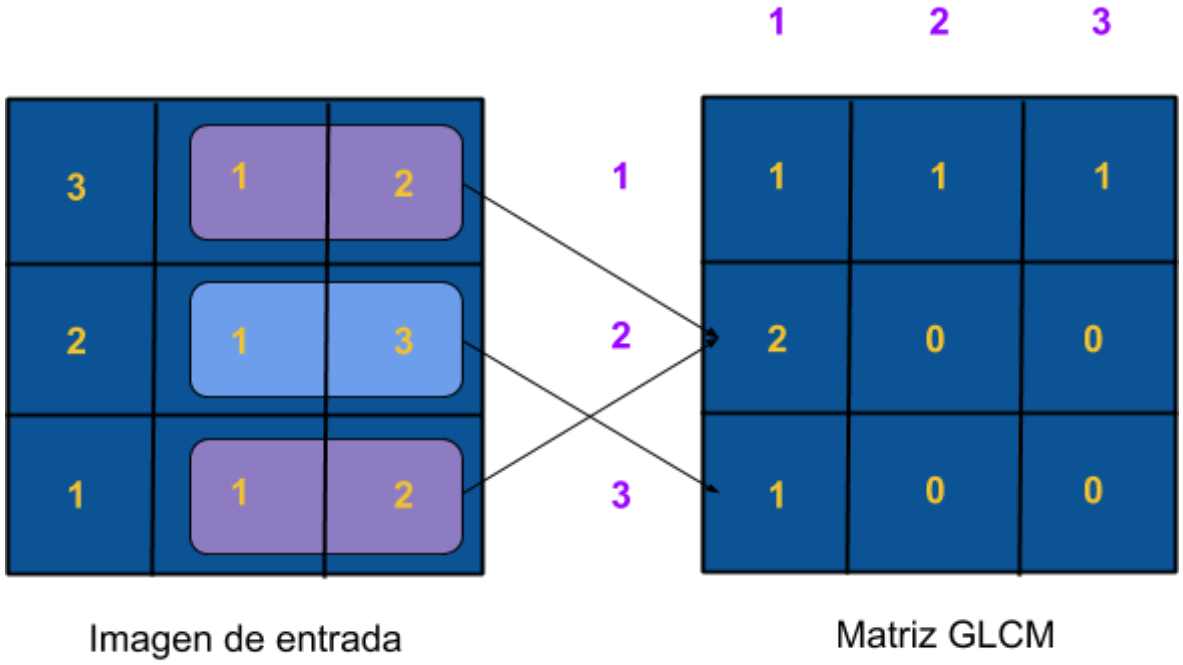

Figura 2.1 Matriz de Coexistencia de Nivel de Grises (GLCM)

Como se puede ver en la imagen anterior, el valor de los píxeles de nivel de gris 1 y 2 aparece dos veces en la imagen, por lo tanto, *GLCM* lo registra como dos. Además, vemos que el valor de píxel 1 y 3 ocurre solo una vez en la imagen, por lo tanto, *GLCM* lo guarda como una única ocurrencia. Vale destacar que en esta descripción del funcionamiento del algoritmo solo estamos teniendo en cuenta un sentido de procesamiento, de izquierda a derecha, pero en realidad, el algoritmo calcula 4 matrices en 4 sentidos distintos de procesamiento. Los cuatro tipos de adyacencia que se mencionan son:

- Izquierda a Derecha
- Arriba hacia Abajo
- Arriba a la izquierda hacia abajo a la derecha
- Arriba a la derecha hacia abajo a la izquierda

De estas 4 matrices *GLCM* que se generan, se calculan 14 características texturales que se basan en teorías estadísticas descritas en "*Textural Features for Image Classification*" [32]. Para dar una idea general del funcionamiento de este algoritmo, podemos basarnos en la siguiente cita de la investigación:

"These features are calculated in the spatial domain, and the statistical nature of texture is taken into account in our procedure, which is based on the assumption that the texture information in an image I is contained in the overall or "average" spatial relationship which the gray tones in the image have to one another. We compute a

set of graytone spatial-dependence probability-distribution matrices for a given image block and suggest a set of 14 textural features which can be extracted from each of these matrices. These features contain information about such image textural characteristics as homogeneity, gray-tone linear dependencies (linear structure), contrast, number and nature of boundaries present, and the complexity of the image. It is important to note that the number of operations required to compute any one of these features is proportional to the number of resolution cells in the image block. It is for this reason that we call these features quickly computable.".

Como podemos observar, el funcionamiento del algoritmo Haralick está basado en la relación que tienen las imágenes en el supuesto de que la información de textura en una imagen *I* está contenida en la relación espacial general o "promedio" que los tonos de gris en la imagen tienen entre sí. Además, en la investigación, se menciona que es un algoritmo de uso general y que el tiempo de cómputo del mismo está íntimamente relacionado con la resolución de nuestra imagen a analizar.

2.2.2.1.3 Histograma de Gradientes Orientados (*HOG*)

Por último, se decidió utilizar el descriptor *Histogram of Oriented Gradients* para ver cómo es su comportamiento con la clasificación de estados de cocción en carne vacuna. Este descriptor, a diferencia de los anteriormente vistos, es un descriptor que se basa en el cálculo de los gradientes de una imagen.

En el trabajo que dio a conocer este algoritmo "*Histograms of Oriented Gradients for Human Detection*" [33], presentado por Dalal y Triggs, se buscaba poder mejorar algoritmos previos, los cuales se centraban en la detección de figuras humanas en imágenes digitales. Si bien el objetivo principal de este método es la detección de objetos en las imágenes [53], en esta investigación se busca ver cómo se comporta a la hora de clasificar diversos estados de cocción en carne vacuna. Si bien no se utiliza para su propósito original, se busca tener una primer nocion de como funciona este descriptor de gradientes en esta temática.

Comenzaremos por ver el funcionamiento general del algoritmo para tener una mayor noción de los parámetros que podríamos llegar a ajustar y saber qué podríamos esperar de la detección de gradientes en los histogramas, entre otras cosas.

Para presentar el algoritmo *HOG*, las imágenes de entrada serán de 64 x 128 x 3 dándonos un vector resultante de 3780 de longitud. Debe tenerse en cuenta que este es el tamaño de las imágenes que se utilizaron en el trabajo donde se presentó el método *HOG*, por lo que en un comienzo se seguirá esta premisa para ejemplificar el funcionamiento descriptor, pero en el trabajo utilizaremos el mismo

conjunto de datos previamente mencionado, con tamaños de imágenes de 224 x 224 píxeles.

En particular, en el descriptor *HOG*, la distribución (histogramas) de las direcciones de los gradientes (gradientes orientados) se utiliza como características. Los gradientes (derivados *x* e *y*) de una imagen son útiles porque la magnitud de los gradientes es grande alrededor de los bordes y esquinas (regiones de cambios bruscos de intensidad) y sabemos que los bordes y las esquinas contienen mucha más información sobre la forma del objeto que las regiones planas.

Habiendo hecho un breve repaso sobre los descriptores y una breve descripción de las características que recolecta el algoritmo *HOG*, continuemos con el funcionamiento del algoritmo. Primero que nada, como mencionamos hace unos párrafos atrás, comenzaremos a ver el funcionamiento del algoritmo *HOG* con imágenes de tamaño iguales a las originales del paper donde se presentó el mismo.

Para esto, la primer fase del algoritmo es la de reducir nuestra imagen a una escala 1:2 con un ancho de 64 por 128 píxeles de alto. Para ilustrar este punto, podemos ver el siguiente gráfico, donde se muestra la imagen en su tamaño original y luego se la achica en una escala 1:2. También, podríamos generar un recorte en la imagen, para focalizar un punto en concreto a analizar; dado que nuestro *dataset* tiene una versión de imágenes cortadas que focalizan el corte de carne, este paso no se realiza y directamente proseguimos con el reajuste del tamaño.

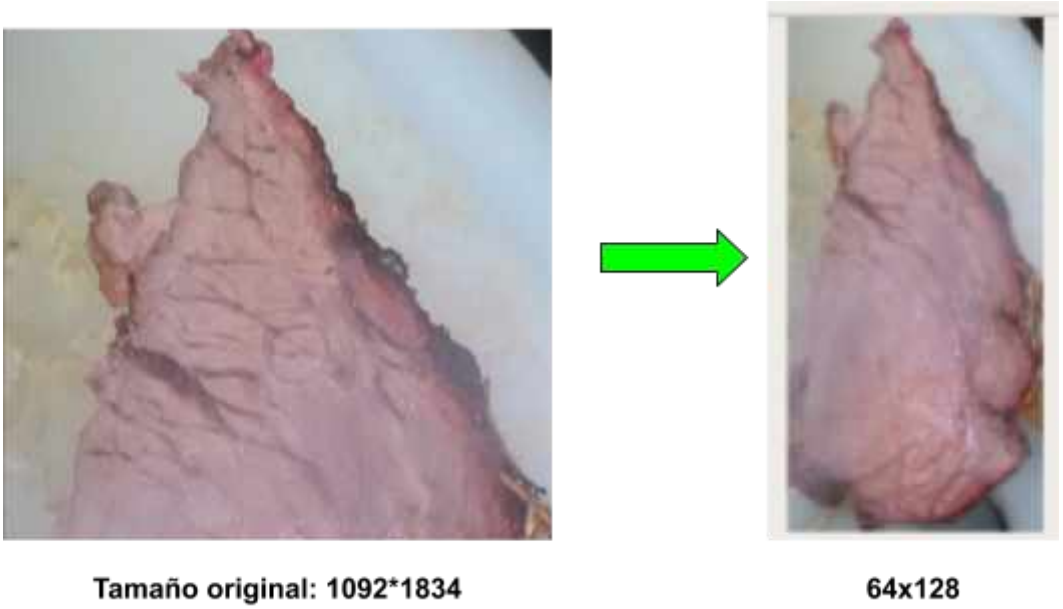

Figura 2.2 Primer paso del algoritmo HOG, redimensionamiento

Además de ver el redimensionamiento en la figura anterior, podemos obtener más información de la imagen a través de la utilización del método *shape* de la imagen cargada. Este nos devuelve la siguiente información:

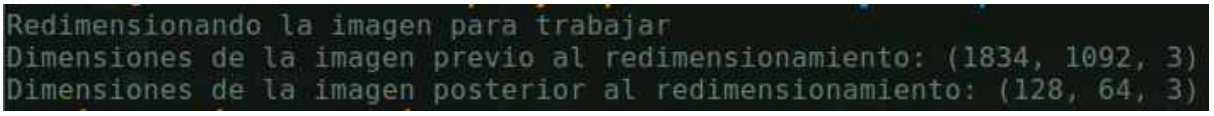

Figura 2.3 Datos del método shape de opencv, retorna número de filas, columnas y canales

Como podemos ver, la imagen destino tiene las dimensiones con las que se trabaja en el paper original del algoritmo *HOG*. Más adelante haremos la prueba de utilizar imágenes sin redimensionarlas para ver las variaciones que se obtienen. Por último, en el trabajo de Dalal y Triggs, se menciona la corrección gamma en la etapa de preprocesamiento, pero según ese trabajo, la ganancia de aplicar esta corrección es mínima, por lo que no se utilizará en este desarrollo.

El siguiente paso en nuestro descriptor es calcular los gradientes de nuestra imagen redimensionada. Primero tenemos que calcular los gradientes horizontales y verticales. Para lograr esto, podemos filtrar nuestra imagen utilizando los siguientes kernels:

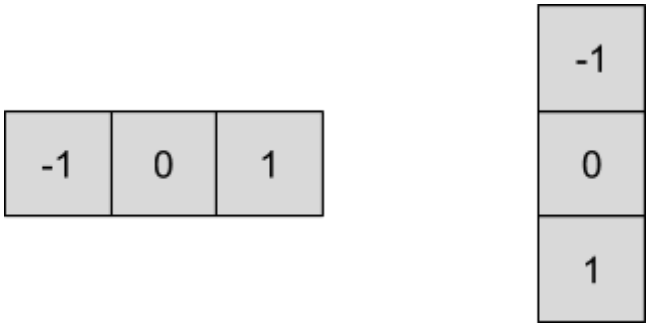

Figura 2.4 Kernel horizontal y vertical respectivamente para filtrar la imagen

Esto mismo lo podemos lograr también con la función Sobel brindada por la librería OpenCV con un tamaño de kernel igual a 1.

Figura 2.5 Cálculo de los gradientes de la imagen redimensionada

A continuación se puede apreciar distintas representaciones de los gradientes calculados:

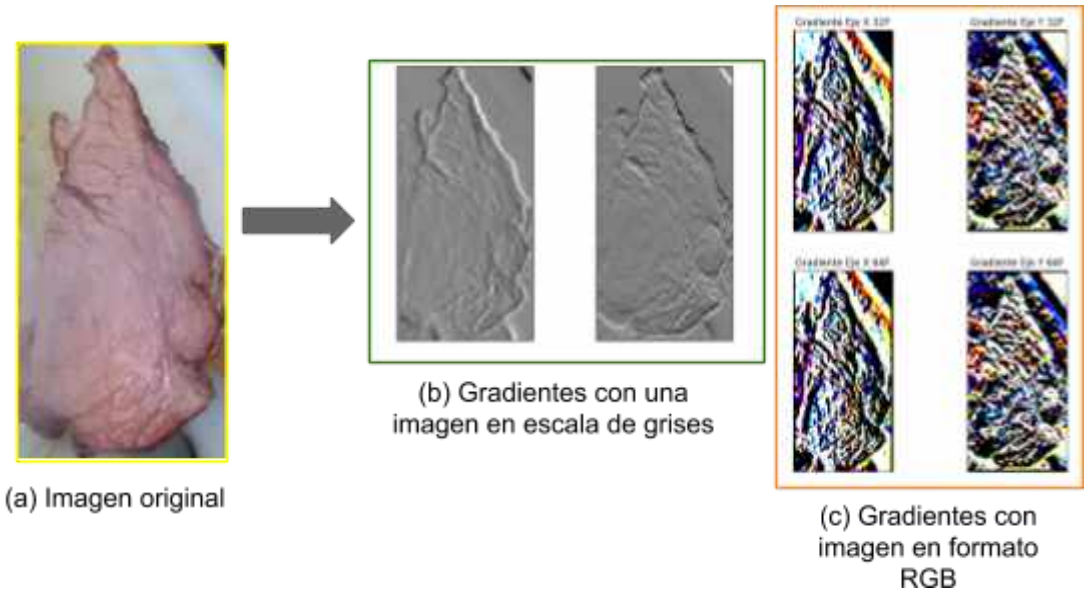

Figura 2.6 Diversas representaciones de los gradientes

Como podemos observar, se brindan diversas representaciones de los gradientes calculados sobre la misma imagen. En la imagen de la izquierda, se calculan los gradientes en el eje *x* e *y* sobre una imagen en escala de grises, con un tamaño de kernel de 1 y un vector de salida de 32F. Por otro lado, tenemos una representación en formato RGB donde se calculan los vectores de salida para punto flotante 32 y 64 sobre cada uno de los ejes con el mismo tamaño de kernel que la imagen en escala de grises. También se hizo la prueba de generar un arreglo con la librería numpy para convertir la imagen en un vector de punto flotante. Con esta representación, cuando calculamos los gradientes de cada eje, obtenemos una representación como la siguiente:

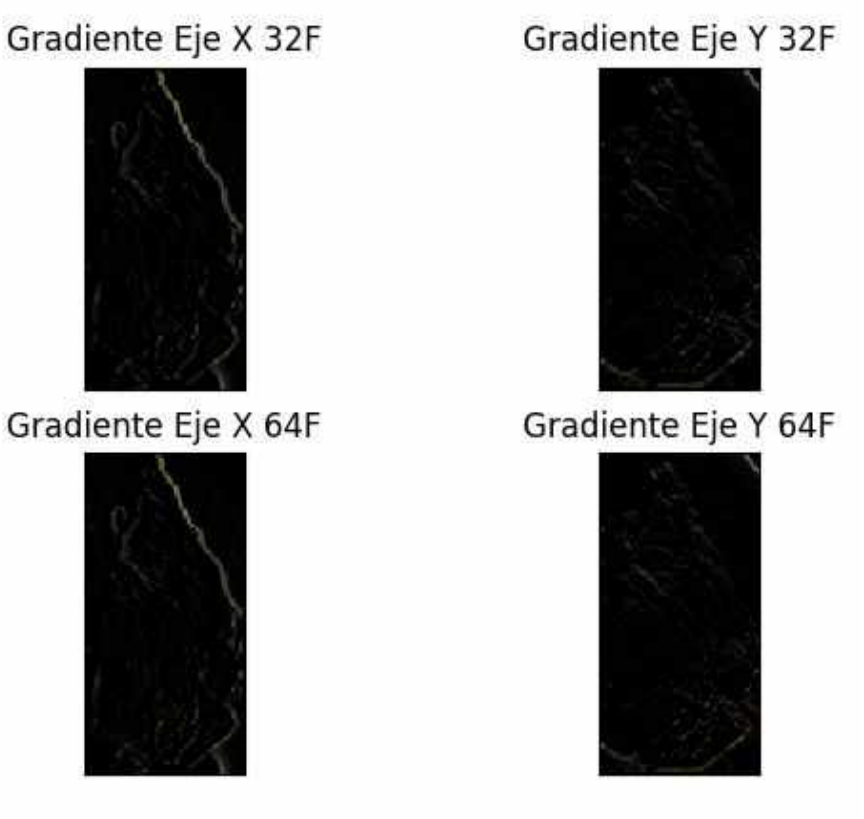

Figura 2.7 Cálculo de gradientes de un vector numpy de punto flotante 32

Una vez calculados los gradientes de ambas direcciones de nuestra imagen, podemos proseguir para encontrar la magnitud y la dirección de los mismos. Para calcular estas dos propiedades, podemos utilizar las siguientes fórmulas:

$$g = \sqrt{g_x^2 + g_y^2}$$
$$\theta = arctan\ \frac{g_y}{g_x}$$

Con la librería OpenCV, esto lo podemos llevar a cabo con la función *cartToPolar*. Esta función calcula la magnitud y ángulo de un vector de 2 dimensiones. A continuación podemos ver la salida del cálculo de la magnitud y dirección de la imagen previamente redimensionada.

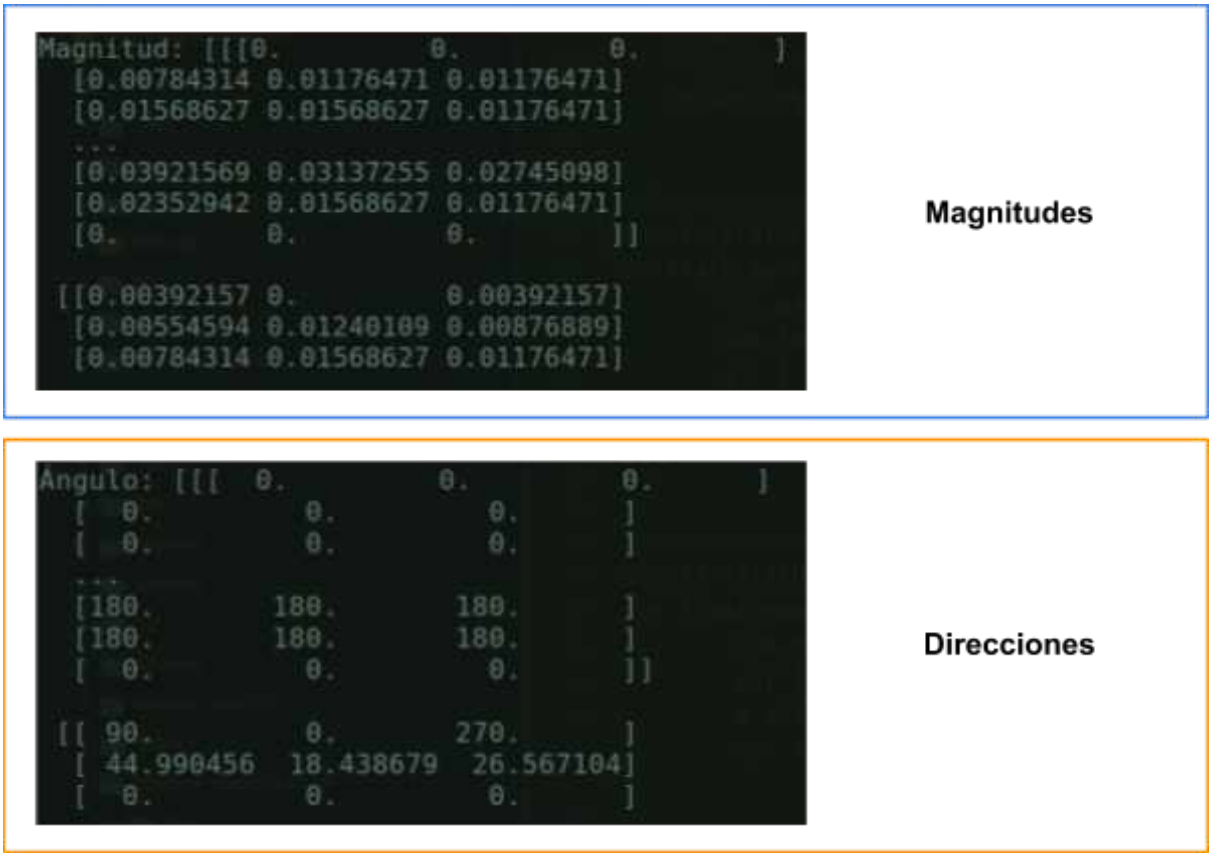

Figura 2.8 Muestra de los vectores de magnitudes y direcciones de los gradientes

Para clarificar qué es lo que representan estos vectores, podemos utilizarlos y verlos como imágenes para ver su representación visual. En la figura siguiente, mostramos la representación gráfica del vector de magnitudes y de la distribución de las direcciones respectivamente:

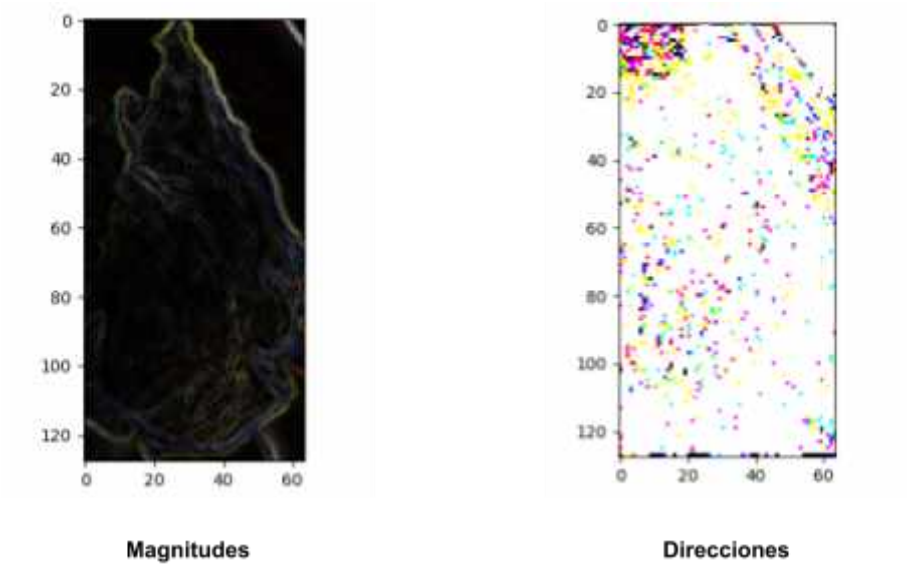

Figura 2.9 Representación visual de los vectores de magnitudes y direcciones

Otra forma de calcular *HOG* es utilizar el método *hog* del módulo *feature* de la librería scikit-image. Utilizando este método, obtenemos los siguientes vectores:

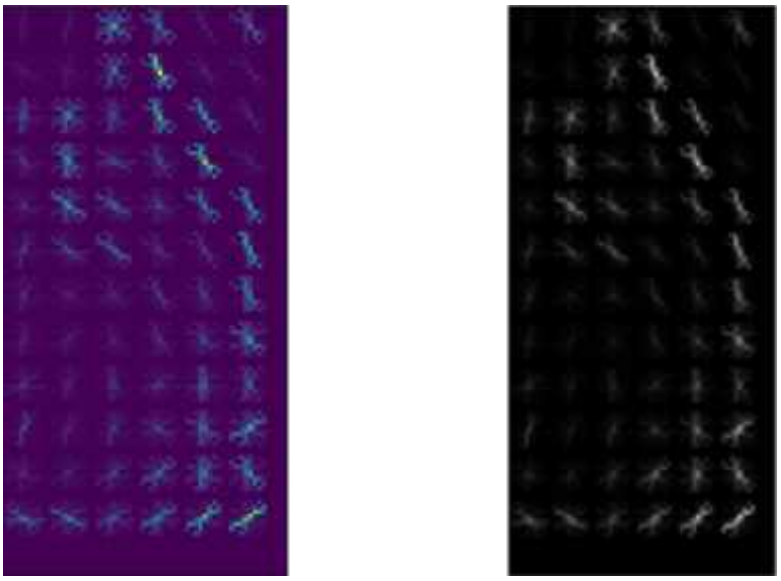

Figura 3.0 Representación visual de skimage HOG

En estas representaciones podemos ver la direccionalidad y la magnitud de los vectores calculados. Es una forma más intuitiva de ver qué es lo que está caracterizando el descriptor *HOG*. Vale notar que la utilización del método *hog* de scikit-image hace todo el procesamiento que venimos describiendo sobre cómo funciona el descriptor *HOG*. Este método sería equivalente a usar la clase *HOGDescriptor* de la librería OpenCV-Python y calcular todo el descriptor de la imagen con el método *compute*, el cual realiza todo el cómputo automáticamente.

Prosiguiendo con el funcionamiento del algoritmo *HOG*, una vez calculados los vectores de magnitud y direccionalidad, debemos analizar por cada píxel de la imagen los 3 canales de colores que posee la misma, en caso de que sea una imagen RGB (cómo es el caso de esta problemática). La magnitud del gradiente en un píxel en particular es el máximo de la magnitud de los gradientes de los tres canales, y el ángulo es el ángulo correspondiente al gradiente máximo. El siguiente paso a realizar, una vez calculado los vectores de magnitud y de direcciones, es dividir la imagen en **celdas**. Una **celda** es un rectángulo definido por la cantidad de pixeles que pertenecen a ella. Por ejemplo, si tenemos una imagen de 128 x 128 y definimos el tamaño de celda de 4 x 4, pasariamos a tener una imagen dividida en 32 x 32 celdas, lo que genera un total de 1024 celdas en toda la imagen. Notemos que el valor del tamaño de la celda puede variar, no tiene un valor fijo. Esto lo podemos ver reflejado en las siguientes figuras, donde vemos cómo dividimos nuestra imagen en celdas de 8 x 8 y 16 x 16 respectivamente. Definir celdas de mayor o menor tamaño nos brindará distintos resultados, lo que mejorará o empeorará el funcionamiento del algoritmo para la problemática en cuestión. Por esto, seleccionar un tamaño de celda está sujeto a cada problemática en particular;

es una elección de diseño basada en la escala de características que estemos buscando.

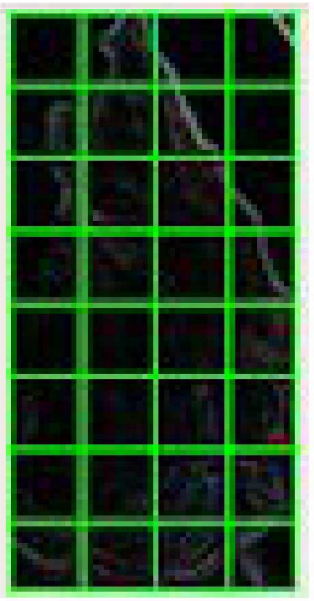

Figura 3.1 División del vector en celdas de 8x8 y 16x16 respectivamente

Una de las principales razones, para hacer esta división en celdas en el uso de este descriptor, es que las celdas nos brindan una representación compacta de la imagen. Por ejemplo, en la división de la imagen previamente planteada, teníamos una celda de 8 x 8 que contenía 8 x 8 x 3 = 192 píxeles, ya que la imagen es una representación en RGB, por lo que tenemos 3 canales de 8 x 8. Ahora bien, el gradiente de esta celda contiene 2 valores por píxel, la magnitud y la dirección del mismo, dándonos un total de 128 números. Estos 128 valores los podemos representar como un vector de 9 posiciones, por lo que no solo la representación de la imagen es más compacta sino que calcular histogramas sobre estas celdas nos brinda, además, una representación menos sensible al ruido que pueda estar presente en una imagen.

Una vez dividida nuestra imagen por la cantidad de celdas deseadas, debemos proseguir a calcular los histogramas de gradientes orientados con las magnitudes y las direcciones previamente obtenidas. El histograma será un vector de 9 posiciones que representan los ángulos posibles de cada píxel. Para dilucidar esta idea, veamos un ejemplo utilizando nuestra imagen previamente dividida en celdas de 8 x 8.

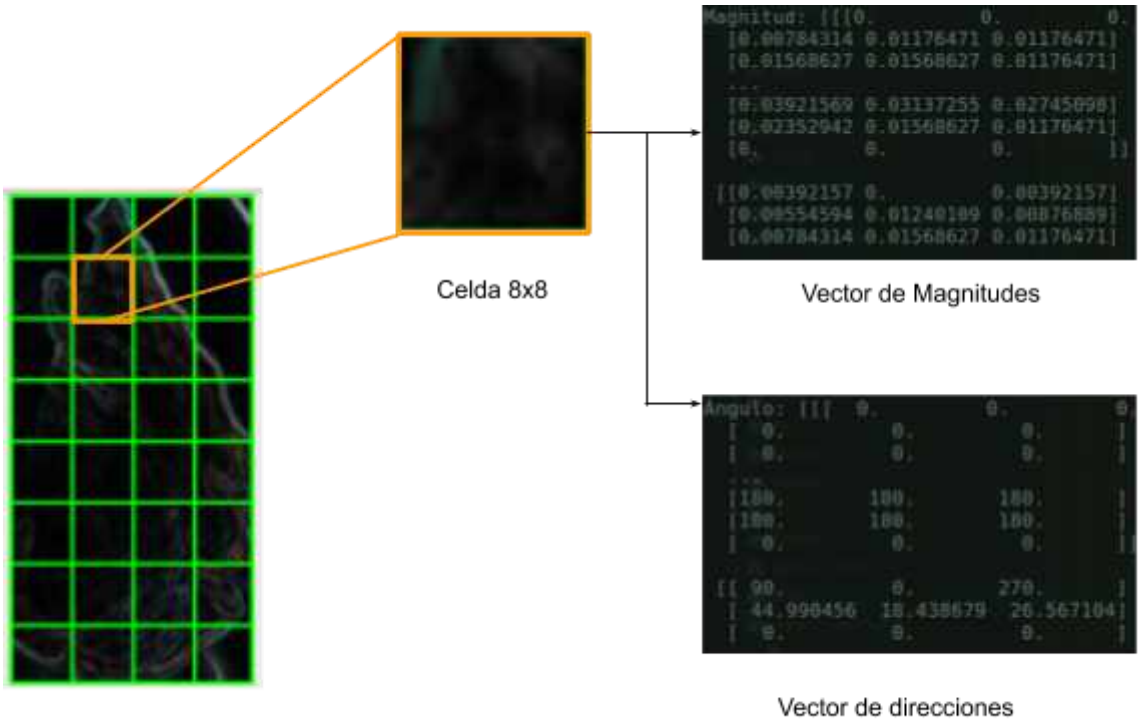

Figura 3.2 Representación de la celda con los vectores de magnitudes y direcciones

A la derecha, vemos los números sin procesar que representan los gradientes en las celdas 8 × 8. Existen dos tipos de ángulos que se pueden calcular, los "sin signo" o "con signo", también conocidos como *unsigned* y *signed* en inglés. Los ángulos "sin signo", están entre 0 y 180 grados en lugar de 0 a 360 grados. Estos se llaman gradientes "sin signo" porque un gradiente y su negativo están representados por los mismos números. En otras palabras, una dirección del gradiente y la de 180 grados opuesta a ella se consideran iguales. Algunas implementaciones de *HOG* permiten especificar si se desea usar gradientes con o sin signo. En la representación anterior, podemos ver que se trata de un vector de direcciones con signo, ya que hay ángulos que superan los 180 grados.

Como mencionamos previamente, debemos generar nuestro histograma en función de los valores calculados para nuestras celdas. El histograma contiene 9 posiciones correspondientes a los ángulos 0, 20, 40, …, 160. Ahora bien, para calcular nuestro histograma en base a los gradientes calculados y colocar el resultado en una de las 9 posiciones tenemos que realizar el siguiente procedimiento. Comenzamos seleccionando una ubicación según la dirección calculada, y el valor que se guarda en dicha posición se selecciona según la magnitud. Para ejemplificar el funcionamiento de este procedimiento veamos el siguiente ejemplo:

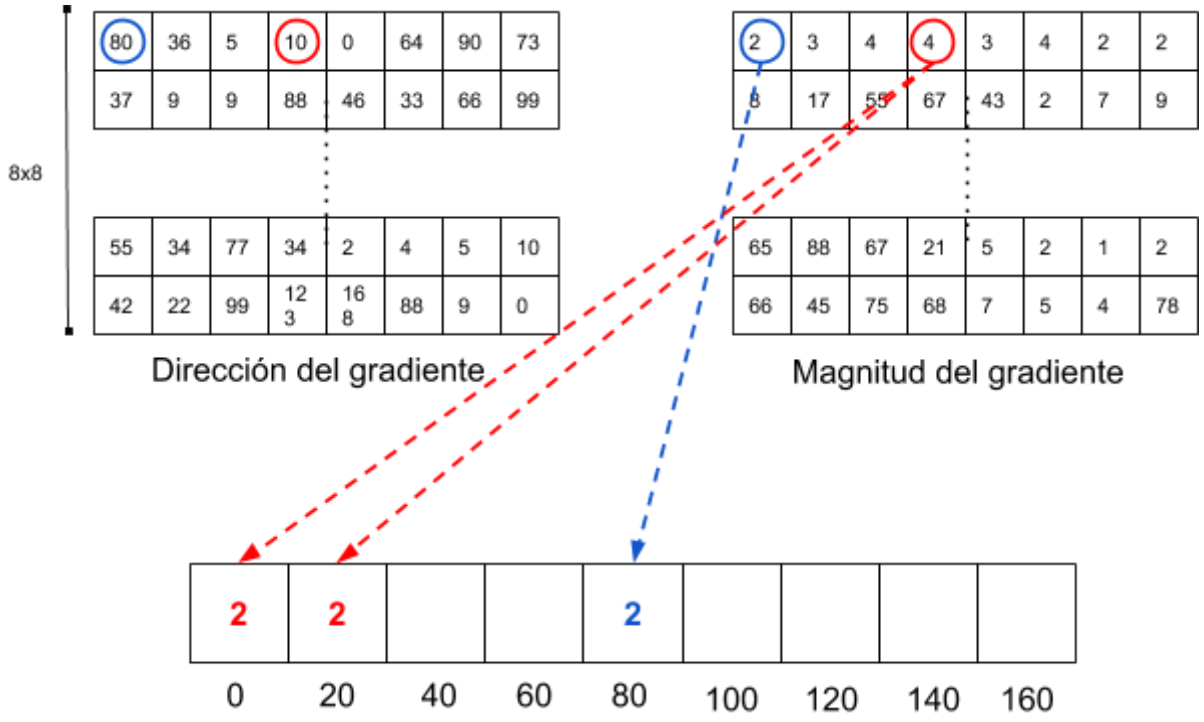

Figura 3.3 Cálculo del histograma por cada celda

Primero, centrémonos en el píxel rodeado de azul: en el vector de direcciones, tiene un ángulo (dirección) de 80 grados y una magnitud de 2. Por lo tanto, agrega 2 a la quinta posición. El gradiente en el píxel rodeado con rojo, en el vector de direcciones, tiene un ángulo de 10 grados y una magnitud de 4. Dado que 10 grados se encuentra entre 0 y 20 grados, el valor del píxel se divide uniformemente en los dos contenedores. En caso de que el valor de la magnitud sea impar y la división no de como resultado un número entero, se adiciona la parte decimal a cada posición del vector. Otra situación que vale la pena marcar es cuando el ángulo calculado es superior a 160 grados. En este caso, sí sabemos que los ángulos están contenidos entre 0 y 180, el valor de la magnitud va a contribuir a ambas celdas, la primera y la última. Este proceso debemos realizarlo por cada celda de la imagen y cada celda va a tener un histograma asociado, el cual será utilizado para llevar a cabo la última parte del algoritmo.

El último paso del algoritmo consta de la concatenación de todos estos histogramas calculados para generar un único histograma con todas las características obtenidas. Si bien con este último paso ya lograríamos tener nuestro descriptor funcionando, se recomienda un paso adicional para normalizar nuestras celdas en bloques. Se recomienda la normalización de nuestro histograma ya que los gradientes de una imagen son sensibles a la iluminación general de la misma. Si se oscurece la imagen al dividir todos los valores de píxeles por 2, la magnitud del gradiente cambiará a la mitad y, por lo tanto, los valores del histograma cambiarán a la mitad. Idealmente, queremos que nuestro descriptor sea independiente de las variaciones de iluminación, para que los histogramas no se vean afectados por las variaciones de iluminación. Para comprender mejor la normalización que se está

planteando realizar, veamos un ejemplo. Supongamos que tenemos el vector RGB [128, 64, 32]. El largo de este vector es igual a $\sqrt{128^2 + 64^2 + 32^2} = 146.64$. Este procedimiento también es conocido como normalización L2. Ahora bien, si dividimos nuestro vector original por el valor de la longitud obtenido (146.64) nos retorna un vector normalizado igual a [0.87, 0.43, 0.22]. Consideremos, a continuación, otro vector en el cual los valores son el doble de nuestro vector anterior [256, 128, 64]. Si llevamos a cabo la normalización L2 en este vector, llegaremos a que nos da el mismo resultado [0.87, 0.43, 0.22]; con esto notamos que al normalizar un vector, estamos dejando de lado la escala del mismo. Vale notar que en el procedimiento de normalización anterior, estamos llevando a cabo la normalización L2, lo cual no implica que no se puedan utilizar otros tipos de normalizaciones dentro del cálculo del descriptor *HOG*, pero se decidió utilizar L2 ya que es ampliamente utilizada [54].

Ahora bien, una vez comprendido el proceso de normalización de vectores, podemos aplicarlo al histograma calculado para nuestra imagen. Para realizar esta tarea, podemos agrupar nuestras celdas en bloques. Es común utilizar una misma celda en distintos bloques.

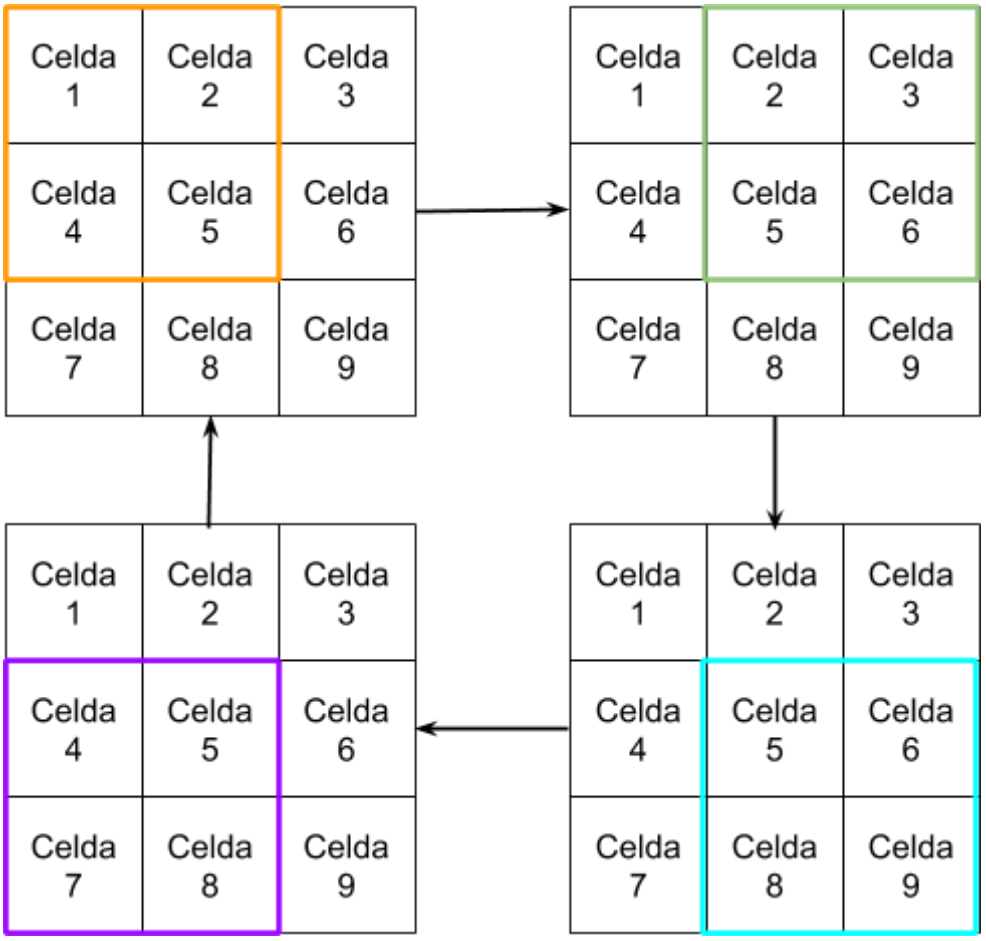

Figura 3.4 Representación de la normalización del histograma para mejorar el acierto

Para cada una de las celdas del bloque actual, concatenamos sus histogramas de gradiente correspondientes, seguidos de L1 o L2 que normalizan todo el vector de características concatenadas. De nuevo, realizar este tipo de normalización implica que cada una de las celdas se representará en el vector de características final varias veces, pero se normalizará con un valor diferente. Si bien esta representación múltiple es redundante y desperdicia espacio, en realidad aumenta el rendimiento del descriptor.

Finalmente, una vez que todos los bloques se normalizan, tomamos los histogramas resultantes, los concatenamos y los tratamos como nuestro vector de características resultante.

Con esto finalizamos la revisión de los descriptores utilizados en este trabajo. A continuación, seguiremos viendo algoritmos de aprendizaje supervisado del área de *Machine Learning*.

2.2.2.2 Aprendizaje Supervisado

Los algoritmos de Aprendizaje Supervisado, son una clase de algoritmos de *ML* que usan datos previamente procesados para aprender las características necesarias para clasificar la nueva información que se le brinde. Un ejemplo típico de este tipo de aprendizaje es el de la clasificación de correos electrónicos. Un usuario podría catalogar los correos que tiene en su casilla como correos "spam" o "no spam", si utilizamos este conjunto de datos ya categorizados, podemos entrenar un modelo para que luego pueda categorizar de manera automática nuevos correos que lleguen. Esta categorización se va a llevar a cabo comparando la información que tiene como base de conocimiento con los nuevos correos. Si un correo tiene más semejanzas con la categoría "spam" lo clasificará como tal; por el contrario, si tiene más similitudes a correo "no spam", lo catalogará con dicha clase. En este ejemplo, podemos notar que la categorización se produce con 2 clases distintas, pero este comportamiento se lo podría llevar a un conjunto de *n* clases, el cual nos permita realizar comparaciones multiclase.

Algunos de los algoritmos más conocidos de aprendizaje supervisado son, por ejemplo:

- *Linear and logistic regression* o Regresor lineal y logístico
- *Support Vector Machines* o Máquinas de Soporte Vectorial
- *Decision Trees* o Árboles de Decisión
- *Naive Bayes* o Bayesiano Ingenuo
- *Neural Networks* o Redes Neuronales

En esta investigación haremos uso de los algoritmos *SVM* y de las Redes Neuronales, ya que son algoritmos muy utilizados para este tipo de problemas.

Comenzaremos viendo los *SVM* y luego, en el apartado "2.2.2.3 Redes Neuronales", veremos en detalle las *Neural Networks*.

Específicamente usaremos una versión de *SVM* que nos permite aplicar un *SVM* para problemas de regresión, conocido como algoritmo *SVR, Support Vector Regression*, ya que como menciona Saed Sayad en [37], los principios básicos de los SVM son aplicables para problemas de regresión, salvo que hay que tener en cuenta un valor *epsilon*, que nos va a servir de margen de tolerancia para subsanar el hecho de que no son valores exactos, sino que son valores continuos, reales. El resto de los tipos de aprendizaje supervisado se mencionan para dejar la noción de que existen y que forman parte del aprendizaje supervisado, pero no serán descriptos en este trabajo.

2.2.2.2.1 Máquinas de Soporte Vectorial (*Support Vector Machines*)

Salvando las distancias previamente mencionadas entre las *SVM* y las *SVR*, comenzaremos a describir el funcionamiento general de las *SVM*. El objetivo de este tipo de clasificadores es el de encuadrar los datos que se le brinda dentro de las posibles "categorías" que dispone, retornando un hiperplano de "mejor concordancia", que divide los datos de entrada dentro los grupos posibles. Para aclarar el funcionamiento de este clasificador, podemos ayudarnos con un gráfico como el siguiente.

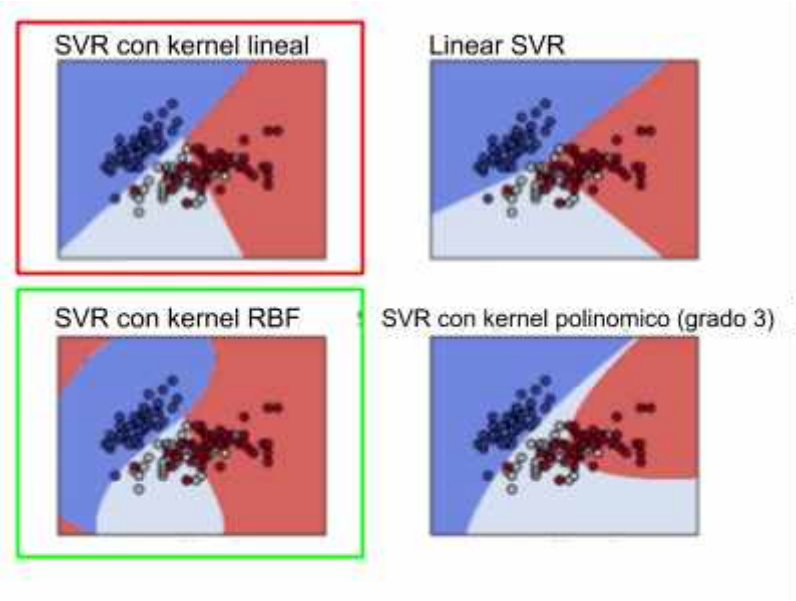

Figura 3.5 Interpretación gráfica de las *SVR* con distintos *kernels*

El clasificador *SVM* puede ser usado con distintos *kernels*, en este trabajo se utiliza el *SVR* con un *kernel* lineal y RBF; de las representaciones del gráfico anterior, es el marcado con el recuadro rojo y verde, respectivamente. Los distintos *kernels* que se pueden utilizar en este tipo de clasificador nos brindan la posibilidad

de tener una mejor manera de separar nuestros datos para la clasificación, cómo se puede ver en el trabajo [48] que utilizan un *SVM* con kernel polinómico para lograr una separación no lineal de los datos.

Ahora bien, ¿qué significa que los algoritmos *SVM* buscan un hiperplano de "mejor concordancia"? Un hiperplano es un plano en una dimensión mayor del espacio, es decir, si nuestro espacio tiene una dimensión de tamaño *n*, nuestro hiperplano va a tener una dimensión de *n - 1*. Por ejemplo, si nuestro espacio es de 1 dimensión, nuestro hiperplano va a ser sencillamente un punto dentro de ese espacio; asimismo si nuestro espacio es de 2 dimensiones, nuestro hiperplano será una recta dentro de los ejes *x* e *y*. Este último ejemplo, lo podemos ver en la figura anterior, donde tenemos un plano con eje *x* e *y*, con un conjunto de datos distribuidos dentro del espacio. Nuestro hiperplano dentro de estos conjuntos están denotados por el fondo de distintos colores. Vale aclarar, que lo que produce distintas líneas en cada una de las imágenes es el *kernel* que mencionamos previamente. Esta función nos permitirá dividir nuestro espacio de diversas maneras para poder hacer una mejor segmentación del espacio, y así, lograr una mejor segmentación de nuestros datos. Los algoritmos *SVM*, buscan el hiperplano óptimo, esto significa que los puntos tengan la máxima distancia hacia el mismo. Esto se puede ver más claramente en un gráfico como el siguiente.

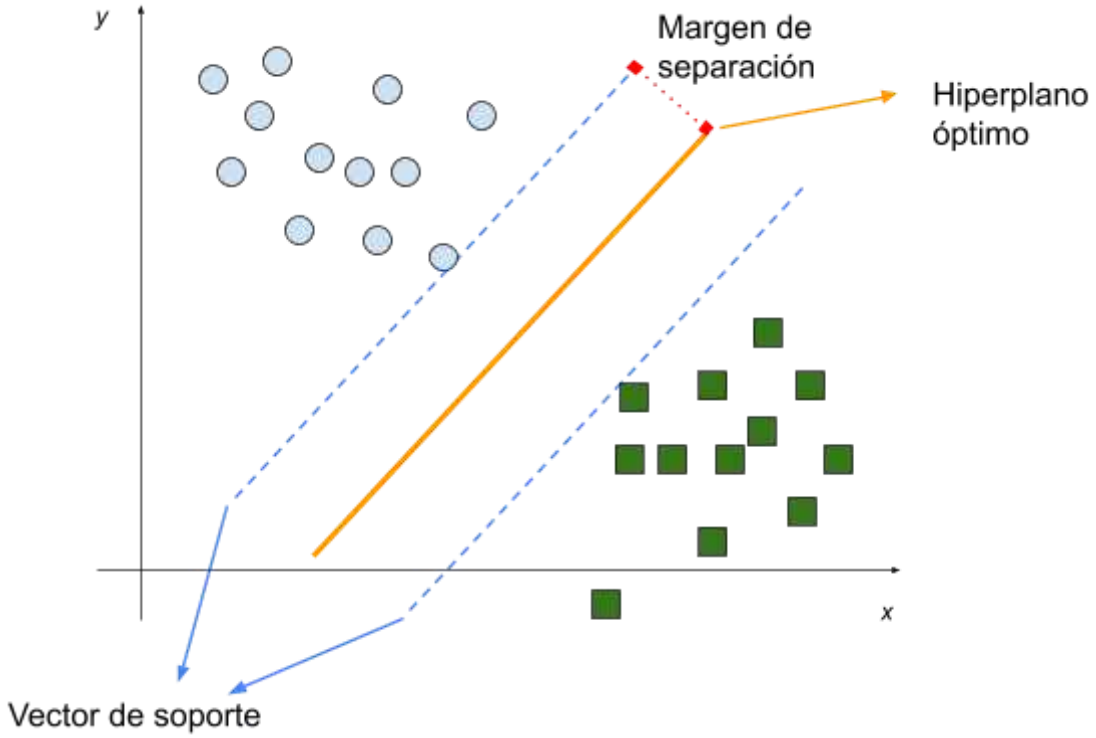

Figura 3.6 Hiperplano óptimo entre 2 clases

Los clasificadores *SVM*, también son útiles cuando los datos a clasificar son linealmente inseparables. Para solucionar estos casos hay 2 enfoques que se pueden emplear, *soft margins* o *kernel trick*.

La técnica de *soft margins* permite clasificar erróneamente algunos elementos en pos de mantener el mayor porcentaje de clasificación posible. Por otro

lado, la técnica de *kernel trick*, soluciona esta problemática con otro enfoque. Supongamos que tenemos un espacio bidimensional, pero las clases son linealmente inseparables, *kernel trick* utiliza una función núcleo que agrega una dimensión a nuestro espacio original, en nuestro ejemplo de 2 dimensiones nos deja con un espacio de 3 dimensiones en total. Las clases que eran inseparables en nuestro conjunto origen de 2 dimensiones, al agregarles una nueva dimensión se convertirán en clases linealmente separables en 3 dimensiones, ya que nos permite generar un "corte" en nuestro espacio tridimensional que logra separar las 2 clases de manera óptima. La lógica de este método la podemos apreciar mejor mediante un gráfico como el siguiente:

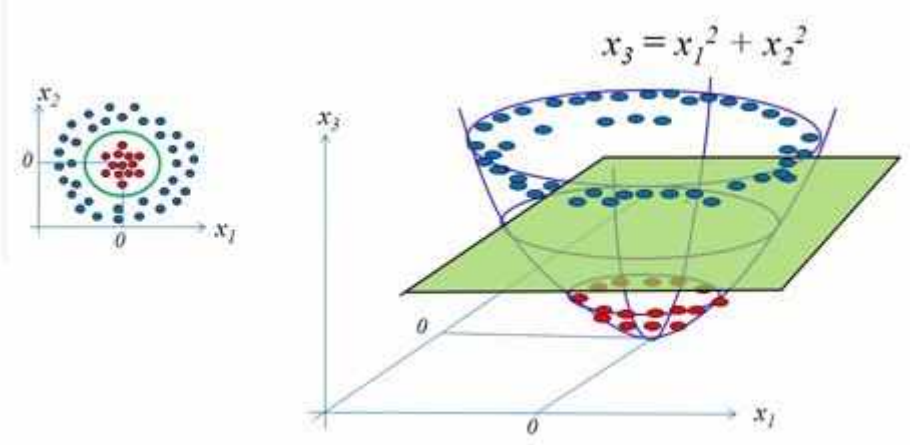

Figura 3.7 Visualización de *kernel trick* para solucionar conjuntos inseparables

Ahora que entendemos mejor cómo es el proceso de "aprender" en los sistemas de *ML*, y conocemos más sobre cómo funcionan los clasificadores *SVM* y de qué información aprenden, podemos definir el siguiente campo que da soporte al campo de *Deep Learning*, conocido como *Neural Networks* (*NN*) o Redes Neuronales.

2.2.2.2.2 Redes Neuronales (*Neural Networks*)

Las redes neuronales surgen de la idea de utilizar los conocimientos que tenemos sobre cómo funciona nuestro cerebro [57]. Veamos cómo está compuesto el mismo para lograr entender mejor cuando decimos que las redes neuronales surgen de esta premisa. La unidad mínima de procesamiento del cerebro se denomina neurona, está compuesta por axones, dendritas, un cuerpo celular y sus terminales presinápticos. El axón contiene electricidad y su función principal es la de transmitir la información procesada a las demás neuronas; se extiende desde el cuerpo celular hasta el objetivo (por ejemplo, otra neurona, o algún tejido muscular). Las dendritas, por otra parte, se encargan de recibir la información que va a ser

procesada y, posteriormente, transmitida por los axones a las células conectadas a esta neurona. Cada una de las conexiones de entrada, son valoradas según la asiduidad con que se use esta dendrita en particular; mientras más se use cierto terminal, mayor valoración va a tener la información captada. Por último, tenemos el cuerpo celular, que se encarga de procesar la información obtenida por las dendritas, y sumar todas las entradas previo al cálculo de la fuerza de sus respectivas conexiones. Después de todo el cálculo de la suma de las entradas, el cuerpo celular se lo pasa a los axones para que le transmita el resultado calculado a sus conexiones. Podemos ver en la siguiente figura, un esquema de la estructura neuronal descrita:

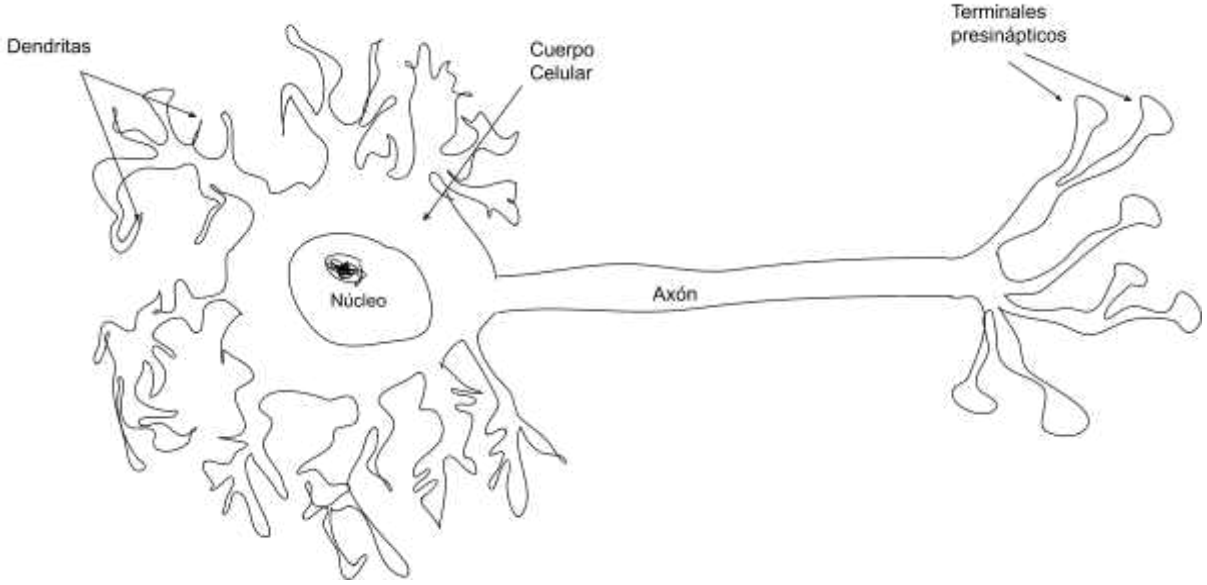

Figura 3.8 Representación de la estructura de una neurona

Como es de esperar, una sola neurona no es capaz de generar conocimiento tal que nos permita, por ejemplo, distinguir un objeto, reconocer algún color, etc. Según Mountcastle, Vernon B. en [19], el cerebro está organizado en capas, y se dice que su córtex (es la estructura responsable de gran parte de la inteligencia humana), está compuesta por 6 capas, donde la salida de cada capa es la entrada de la siguiente capa y así sucesivamente, hasta lograr transformar los datos sensoriales de entrada en la comprensión conceptual. Esta topología en capas, que en cada capa hay un conjunto de neuronas que van adquiriendo características relevantes del objeto a analizar, se lo pasan al siguiente nivel y así continúa el sistema hasta que en la capa final se logra interpretar el objeto a analizar y nos permite decidir si lo que estamos viendo, oliendo o probando, es un auto, una pizza, si está condimentado con pimienta o no, si huele rico o no, etc.

Más técnicamente, podríamos definir a las redes neuronales como un modelo matemático de procesamiento de información. La diferencia con cualquier otro enfoque de *ML*, está en que las redes neuronales son un modelo, por lo que no es

un algoritmo estático, fijo, sino que es un sistema que procesa información. Algunas características de las redes neuronales son las siguientes:

- El procesamiento de la información ocurre en su forma más sencilla posible, a través de elementos conocidos como neuronas o *neurons* en inglés.
- Las neuronas están conectadas e intercambian información entre ellas a través de enlaces de conexión.
- Los enlaces de conexión entre las neuronas pueden ser más fuertes o más débiles, lo que determina cómo se interpreta la información leída por ese enlace.
- Cada neurona tiene un estado interno que está determinado por todas las conexiones de entrada de las otras neuronas.
- Cada neurona tiene una función de activación, que es calculada en su estado y determina la señal de salida.

Otra definición con la que se podría describir a las *NN*, sería como un grafo computacional de operaciones matemáticas. Se pueden identificar 2 características principales dentro de las redes neuronales, estas son:

- La **arquitectura** de la red neuronal: describe la forma de conexionado entre las neuronas (*feedforward, recurrent, multi-layered, single-layered*), el número de capas y la cantidad de neuronas por cada capa.
- El proceso de **aprendizaje** de la red neuronal: describe cómo aprende nuestro modelo. Una de las formas más comunes de entrenamiento, pero no la única, es utilizar la técnica de *gradient descent* y *backpropagation* (descritos más adelante).

Ahora bien, ¿qué es una neurona en el contexto de redes neuronales? Una neurona, es una función matemática que toma uno o más valores de entrada y devuelve un único valor numérico. A continuación, podemos observar un esquema de una neurona artificial:

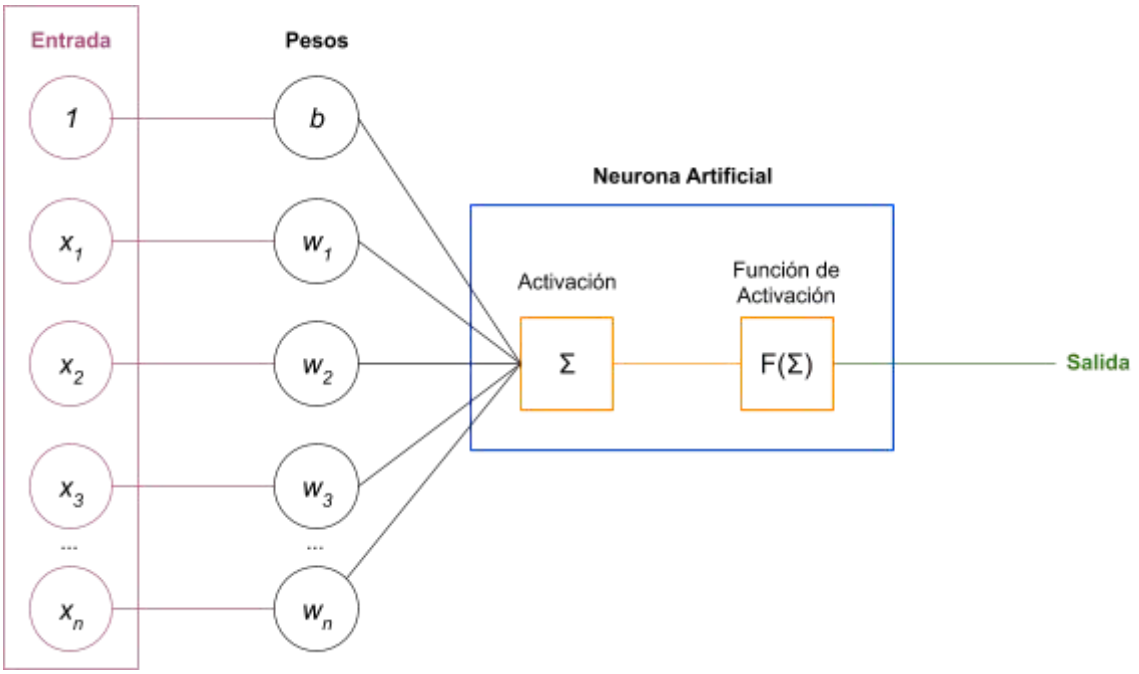

Figura 3.9 Elementos de una neurona artificial

La neurona, también puede ser definida con la siguiente función matemática:

$$y = f(\sum_i x_i w_i + b)$$

Como podemos apreciar, primero computamos la sumatoria $\sum_i x_i w_i$ de las entradas x_i y los pesos w_i, también conocidos como valores de activación. Los valores de x_i, pueden ser tanto entradas de datos como salidas de otras neuronas, en caso de que sea parte de una red neuronal. Los pesos w_i son valores numéricos que representan tanto la fuerza de la entrada o la fuerza de la conexión entre las neuronas. Por último, el valor *b* se lo suele conocer como **sesgo** o *bias* en inglés y su valor es siempre 1. Una vez calculada la suma ponderada, usamos este resultado como una entrada a la función *f*, también conocida como **función de transferencia** o **función de activación**. Hay varios tipos de funciones de activación, pero el principal requerimiento que deben cumplir es que sean no lineales.

Habiendo visto la unidad de cómputo de las redes neuronales, otra parte importante de destacar son las capas que se forman entre cada unas de las neuronas. La cantidad de capas de una *NN* va a depender del interconexionado de las neuronas que esta posee. Por ejemplo, podríamos tener una red neuronal de 1 capa que sea capaz de clasificar los dígitos del 0 al 9 del conjunto de datos de *MNIST* (*Modified National Institute of Standards and Technology*). Esto lo podemos ver más claro en la siguiente figura.

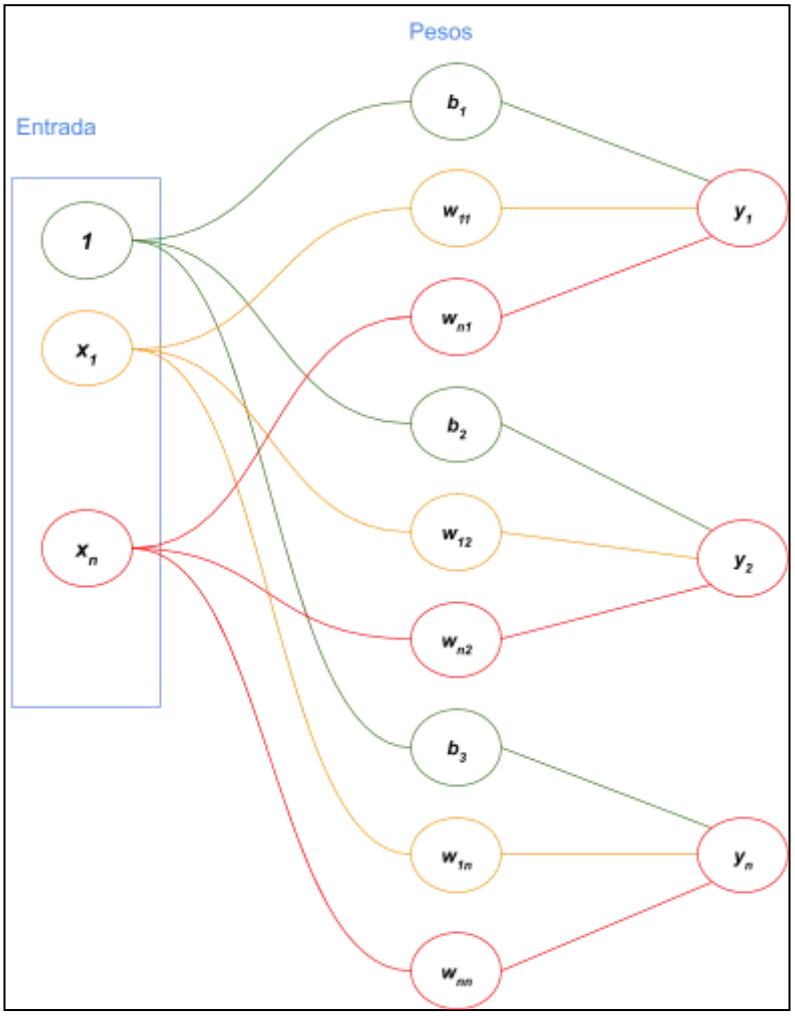

Figura 4.0 Red *feedforward* de 1 capa

En la figura anterior vemos una red neuronal de 1 capa, donde las neuronas de la izquierda representan las entradas con el sesgo *b*, la columna central representa los pesos de cada conexión, y las neuronas de la derecha representan las salidas dados los pesos *w*. Las neuronas, pueden estar conectadas con más de una neurona de otra capa, pero, estas no pueden conectarse con neuronas de su misma capa. Esta organización de las neuronas hace que las redes neuronales sean redes multicapa. Este tipo de arquitectura son las utilizadas en el campo de *Deep Learning*, por lo que en el apartado siguiente, "Aprendizaje Profundo (*Deep Learning*)" se hará una revisión más profunda de este tipo de redes neuronales.

Otro aspecto interesante de mencionar es que la forma en que las redes neuronales, también se suelen diferenciar por el tipo de conexión que tiene sus neuronas. Por ejemplo, si tenemos una red donde todas las neuronas de la capa *n* se conectan a todas las neuronas de la siguiente capa *n+1*, se la suele conocer como red *fully connected,* o completamente conectada, un ejemplo de uso de estas redes lo podemos ver en [60]. Otro tipo de interconexión pueden ser las recurrentes,

que son un caso particular de grafos cíclicos dirigidos, pero que no describiremos en este trabajo ya que no son utilizadas.

Por último, vale la pena notar de la arquitectura de las redes neuronales, que existen diversas funciones de activación que las neuronas pueden utilizar. Las redes neuronales multicapa se pueden utilizar para clasificar clases linealmente inseparables, pero para poder realizarlo deben satisfacer una condición clave, que las funciones de activación no sean lineales. Pero, ¿por qué las redes neuronales deben utilizar funciones no lineales? Si las neuronas no tuviesen funciones de activación, la salida de la red sería la suma ponderada de las entradas $\Sigma i\, w_i x_i$, que sería una función lineal. Es por esto, que la red neuronal entera, se convierte en una composición de funciones lineales, que es en sí mismo, una función lineal. Esto significa que aunque agregaramos capas intermedias, la red sería equivalente a un modelo de regresión lineal. Por este motivo es que se requiere que cada neurona utilice funciones de activación no lineales. Vale notar, que en general, las neuronas de una capa, utilizan la misma función de activación, pero diferentes capas, pueden tener diferentes funciones de activación. Las funciones de activación más comunes, son:

- $f(a) = a$, conocida como función identidad ó ***identity function***.
- $f(a) = \{1\ si\ a \geq 0;\ 0\ si\ a < 0$, conocida como función de umbral de actividad ó ***threshold activity function***.
- $f(a) = \frac{1}{1+exp(-a)}$, conocida como función logística o sigmoide logístico, en inglés ***logistic function***.
- $f(a) = \frac{2}{1+exp(-a)} - 1 = \frac{1-exp(-a)}{1+exp(-a)}$, conocida como ***bipolar sigmoid***.
- $f(a) = \frac{exp(a)-exp(-a)}{exp(a)+exp(-a)} = \frac{1-exp(-2a)}{1+exp(-2a)}$, conocida como ***hyperbolic tangent*** ó *tanh*.
- $f(a) = \left\{\frac{a\ si\ a \geq 0}{0\ si\ a < 0}\right\}$, conocida como ***ReLu*** (*Rectified Linear Unit*). Esta función posee diversas variantes como *Noisy ReLu, Leaky ReLu, ELU*, entre otras.

La función identidad y la función de umbral de actividad fueron muy utilizadas durante el nacimiento de las redes neuronales, con implementaciones como el perceptrón o Adaline (*adaptative linear neuron*), pero con el tiempo, comenzaron a perder popularidad en favor de *logistic sigmoid*, *hyperbolic tangent* ó *ReLu* y sus variantes [55] [58]. En el siguiente apartado, veremos aquellas funciones de activación que son mayormente utilizadas en el campo del *Deep Learning*.

Una vez definida la arquitectura de nuestro modelo neuronal, esto incluye el tipo de interconexión entre las neuronas, la cantidad de *hidden layers*, el número de neuronas por capa y la función de activación, nos resta especificarle al modelo los pesos, que nos determinarán los valores internos de cada neurona de nuestra red

neuronal. Para llevar a cabo este paso, hay varios algoritmos de optimización que podemos utilizar, algunos de estos son:

- *Stochastic Gradient Descent* o Gradiente Descendente.
- *Stochastic Gradient Descent with Momentum*
- *AdaGrad Optimization* o Optimización AdaGrad
- *RMSProp Optimization* o Optimización RMSProp
- *Adam*

Antes de hacer una revisión de estos algoritmos de optimización, deberíamos tener en claro qué queremos optimizar. Como se explicó previamente, durante el entrenamiento, nuestras redes neuronales aplican funciones matemáticas a ciertos valores para poder llegar a un valor de resultado esperado, nuestro objetivo. Este cálculo se debe realizar de forma iterativa ya que el valor calculado por la red neuronal en un comienzo va a diferir del valor objetivo. Esta diferencia en el valor esperado y el valor obtenido se la conoce como el error cometido y puede ser calculada con funciones de pérdida, en inglés conocidas como ***loss functions***. Las *loss functions* se las pueden clasificar principalmente dentro de 2 categorías, Pérdidas de Regresión (*Regression Losses*) y Pérdidas de Clasificación (*Classification Losses*), que están relacionadas al tipo de problemas que aplaca cada una. En la siguiente tabla podemos ver algunas funciones de cada grupo.

Classification Losses	Error de Hinge / Error SVM Multiclase (*SVM Loss*)	$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$
	Error de entropía cruzada	$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$
Regression Losses	Error Cuadrático Medio (*MSE*)	$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}$
	Error Absoluto Medio (*MAE*)	$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$
	Error de Sesgo Medio (*MBE*)	$MBE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)}{n}$

Figura 4.1 Funciones de error

El cálculo del error se lleva a cabo para permitirle a la red neuronal aprender del error que cometió en una iteración previa, y así poder ir corrigiendo el valor de

cada neurona para poder direccionar nuestro aprendizaje hacia el valor objetivo. En este trabajo se hace uso de *MAE* y *MSE* [61]. Habiendo visto qué son las funciones de error, podemos continuar con las funciones de optimización, que buscan optimizar de manera automática el error cometido (calculado con las funciones de error definidas previamente). En esta investigación se utilizaron 2 optimizadores principalmente, el *SGD* y *RMSProp*. Comencemos viendo la función de optimización *SGD*, que es de las funciones más simples utilizadas para buscar los parámetros que minimicen nuestro error cometido. Para comprender mejor *SGD*, primero debemos definir qué es el Gradiente Descendente. Para comprender este concepto veamos el siguiente ejemplo, supongamos que tenemos una neurona lineal que tiene sólamente 2 entradas (pesos), w_1 y w_2. Podemos imaginar un espacio tridimensional donde la dimensión horizontal corresponden a los pesos, la dimensión vertical corresponde a diferentes configuraciones de los pesos y la tercer dimensión se corresponde al error cometido para cada uno de los puntos. Si consideramos los errores cometidos para todas las posibilidades de pesos, obtendríamos un cono cuadrático en nuestro espacio tridimensional, como se muestra en la siguiente figura:

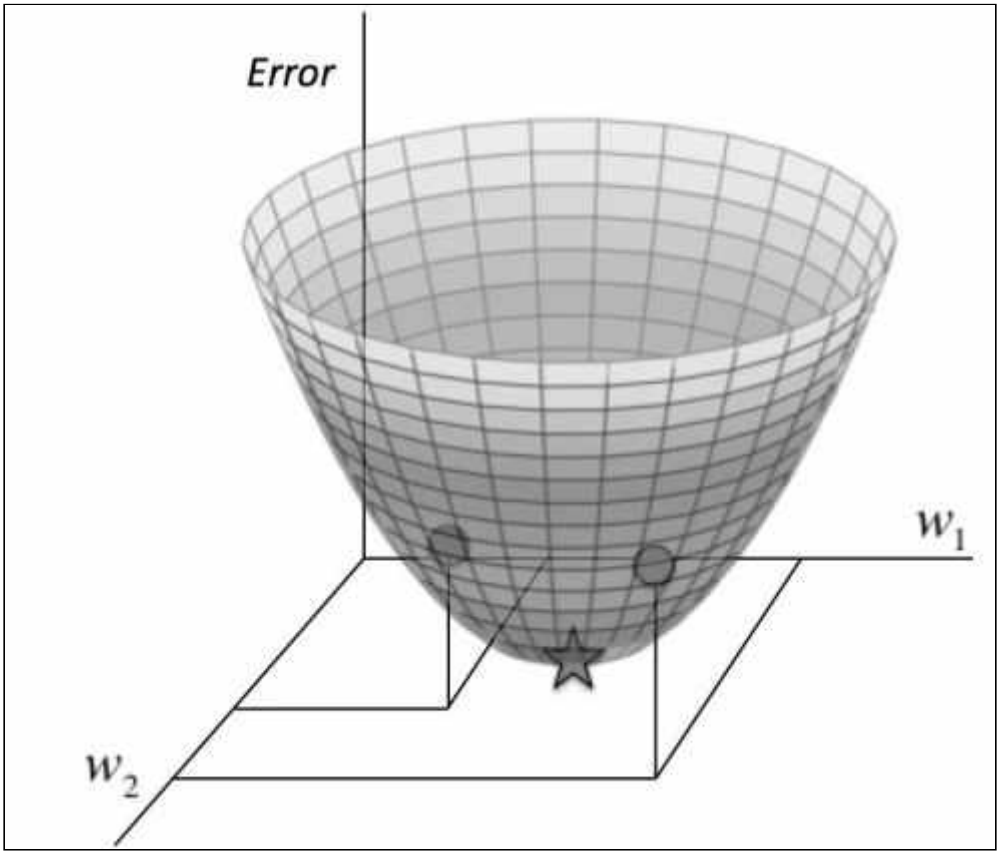

Figura 4.2 Superficie del error cuadrático para una neurona lineal

Esta superficie, la podemos visualizar también como un conjunto de contornos elípticos (vistos desde arriba), donde el error mínimo se encuentra en el

centro de la figura, marcada con una estrella. Con esta configuración estaríamos trabajando en un espacio bidimensional donde las dimensiones se corresponden a los dos pesos. Los contornos de nuestras elipses son las distintas posibles configuraciones de w_1 y w_2 que evalúan el mismo valor E. Mientras más cercanos estén los contornos, más inclinada es la pendiente, de hecho, resulta que la dirección del descenso más pronunciado siempre es perpendicular a los contornos. Esta dirección de salto se expresa como un vector conocido como el gradiente. Supongamos que inicializamos aleatoriamente los pesos en nuestra red neuronal, esto nos dejaría en algún lugar en particular dentro de nuestro espacio bidimensional. Con el cálculo del gradiente en la posición actual, podemos averiguar la dirección que debemos seguir para minimizar nuestro error. Siguiendo este procedimiento, en cada nueva iteración, nos iremos encontrando cada vez más cerca del error mínimo, por lo que iremos achicando nuestro error cometido hasta llegar a un punto donde el error sea el mínimo posible. En la siguiente figura, podemos visualizar el funcionamiento del gradiente descendente.

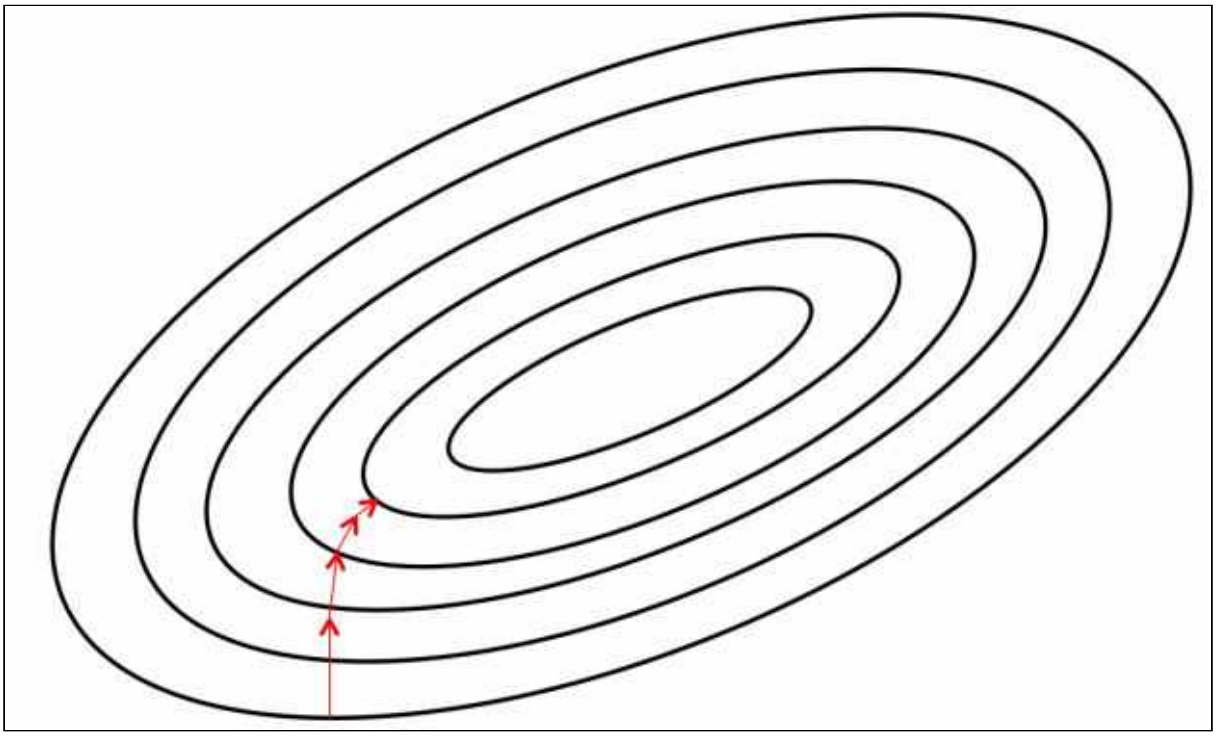

Figura 4.3 Visualización de los errores cometidos en cada iteración de *GD*

Ahora bien, para poder hacer el cálculo de manera satisfactoria, algunos otros parámetros son necesarios. Uno de estos parámetros es conocido como tasa de aprendizaje, o *learning rate* en inglés. En cada etapa, donde debemos realizar el salto perpendicular al contorno de nuestra elipse, debemos determinar qué tan lejos vamos a llegar con nuestros saltos antes de calcular nuestro próximo salto. Alguien se podría preguntar, ¿por qué me interesaría saber cuán lejos voy a realizar el salto? La respuesta reside en que cuando nosotros nos vamos acercando al error

mínimo, queremos reducir nuestra longitud de salto, ya que de lo contrario nos llevaría a un punto muy alejado de éste. Además, nos interesa la variación en el salto, ya que si siempre realizamos el mismo salto, éste siendo un valor pequeño, nos tomaría mucho tiempo llegar a nuestro objetivo. Lo mismo sucede con un valor grande en la tasa de aprendizaje, ya que realizaríamos saltos constantemente pero nunca lograríamos acercarnos lo suficiente a nuestro error mínimo, por lo que se suele tener un valor de salto inicial y mientras que nos vamos acercando al error mínimo, vamos reduciendo nuestro valor de salto. Esto último, lo podemos ver mejor gráficamente cómo se muestra en la siguiente figura:

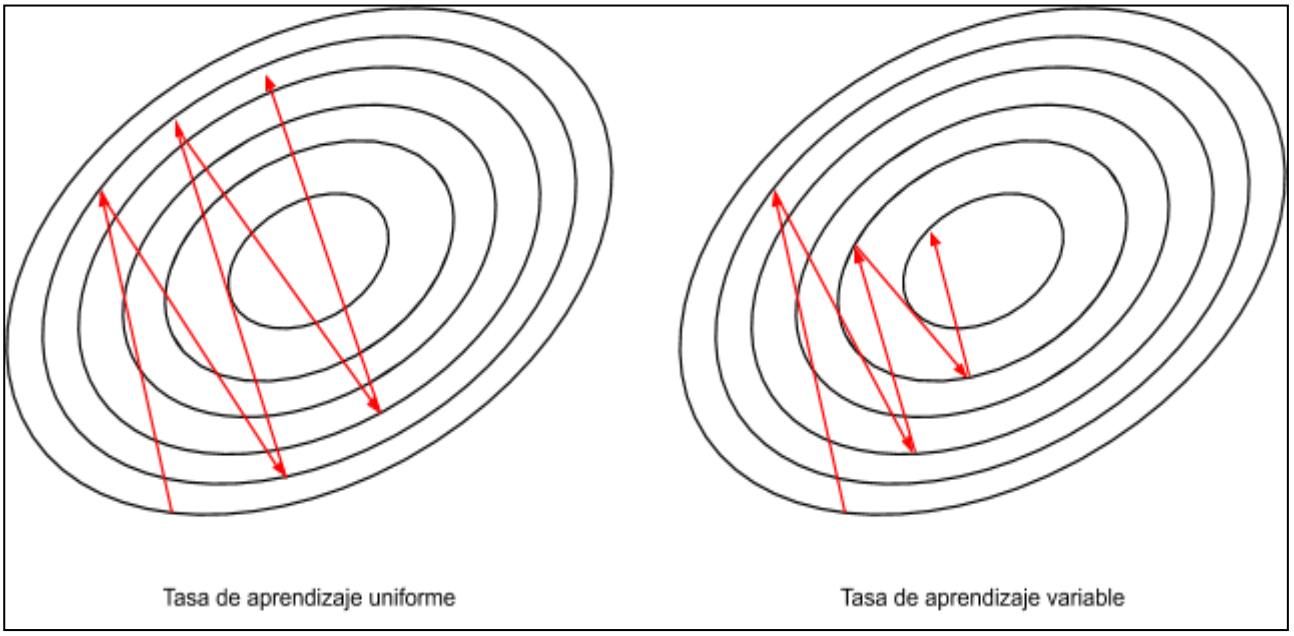

Figura 4.4 Variación en tasas de aprendizaje

Ahora bien, el cálculo del gradiente en cada uno de los puntos, no es más que la derivada parcial de la función de error con respecto a cada uno de los pesos de nuestra red neuronal. Matemáticamente, este cálculo, lo podemos expresar de la siguiente manera:

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}$$

$$= -\epsilon \frac{\partial}{\partial w_k}\left(\frac{1}{2}\Sigma_i \left(t^{(i)} - y^{(i)}\right)^2\right)$$

$$= \Sigma_i \epsilon \left(t^{(i)} - y^{(i)}\right)\frac{\partial y_i}{\partial w_k}$$

$$= \Sigma_i \epsilon x_k^{(i)}\left(t^{(i)} - y^{(i)}\right)$$

Habiendo visto cómo funcionan los gradientes descendentes veamos el algoritmo de optimización *SGD*. El algoritmo comienza definiendo un valor aleatorio inicial para los parámetros, como lo habíamos mencionado cuando explicamos *GD*. El funcionamiento del algoritmo es como el que describimos en *GD*; va a calcular las derivadas para cada parámetro relacionado a la función de error. Las derivadas calculadas, nos van a dar el ajuste numérico que tenemos que hacer con cada parámetro para minimizar el error. Este proceso se va a repetir hasta encontrar un mínimo local o global. Este algoritmo, si bien es uno de los más sencillos de aplicar, tiene algunas contras. La primera que podemos mencionar es cuando el algoritmo detecta un mínimo local (el gradiente va a ser cero en todas las direcciones) pero en realidad no es el mínimo global. Esto nos genera que el error cometido no se haya minimizado totalmente, por lo que la precisión de la clasificación no será óptima. Otra problemática del optimizador es la conocida como Movimiento de Zig Zag (*Zig Zag Motion*). Esta problemática está dada cuando uno de los parámetros es más "sensible" a los cambios que los demás. Esto conlleva a que el tiempo necesario para llegar a nuestro error mínimo aumente significativamente. El comportamiento éste lo podemos ver graficado de la siguiente manera.

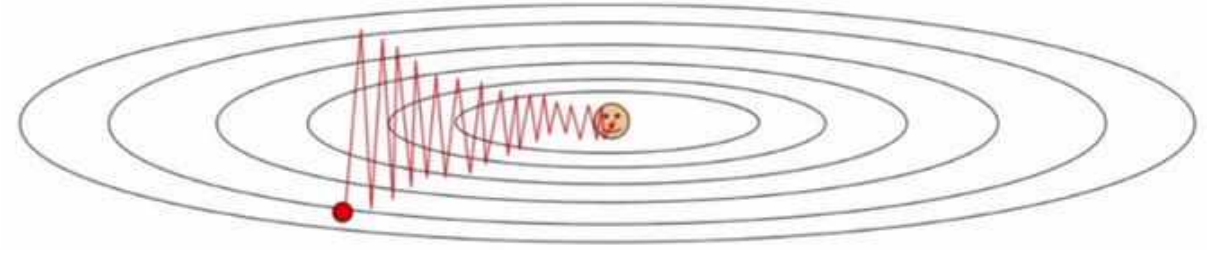

Figura 4.5 Movimiento Zig Zag

El otro algoritmo de optimización utilizado en esta investigación es el *RMSProp*. Este optimizador es una variación del algoritmo *AdaGrad*, el cual guarda la suma de los cuadrados de todos los gradientes calculados por cada uno de los parámetros de la red neuronal. Esta suma, es utilizada posteriormente para escalar

la tasa de aprendizaje del algoritmo. La diferencia, es que el *RMSProp* en lugar de dejar que esa estimación al cuadrado se acumule durante el entrenamiento, se deja que la estimación decaiga gradualmente.

Hasta ahora, hemos visto cómo actualizar los pesos de una única capa con algoritmos de optimización, pero, si nosotros dispusiéramos de una red neuronal con múltiples capas, sólo podríamos aplicar esta optimización a los pesos que conectan la última *hidden layer* con la capa de salida. Esto se debe a que no tenemos valores objetivos para las salidas de las capas ocultas. Para subsanar esta problemática, lo que haremos es calcular el error en la última capa oculta y estimaremos cuál sería el error cometido en la capa previa. Luego, propagaremos hacia atrás este error, desde la capa final a la capa inicial, logrando que haya una retroalimentación de los errores cometidos. A esta técnica se la conoce con el nombre de ***Backpropagation***. Habiendo visto el funcionamiento general de las redes neuronales, continuaremos viendo algunas implementaciones de estas.

Uno de los primeros ejemplos de redes neuronales fue desarrollado por Frank Rosenblatt en el año 1957 [56], conocido como perceptrón o *perceptron*. El perceptrón, es un algoritmo de clasificación muy similar al algoritmo regresión logística, que utiliza características de la entrada de datos para predecir un valor resultante (entre 0:1), como por ejemplo, el valor de una casa dado un conjunto de características como el tamaño, la edad de la casa, la cantidad de baños, cantidad de pisos, cantidad de habitaciones o ubicación, entre otros. El perceptrón es un ejemplo de una red neuronal *feedforward* con una única capa. A continuación tenemos una representación gráfica de la estructura:

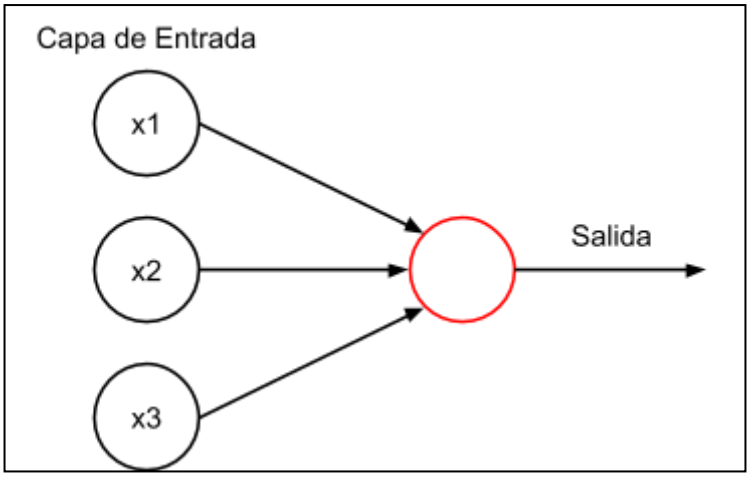

Figura 4.6 Estructura del perceptrón

La salida del perceptrón es una función igual a la sumatoria del producto de la entrada por el peso, $f(\sum_i w_i.x_i)$. La única diferencia con *logistic regression* es que *f* es una función simple, por lo que:

- El resultado es 1 si, $f(x.w) > 0$
- El resultado es 0 si, $f(x.w) <= 0$

En sus comienzos el perceptrón fue muy prometedor, pero al poco tiempo se descubrió que tenía varias limitaciones, por lo que sólo servían para problemas de clasificación linealmente separables.

Otro tipo de redes neuronales que son de gran importancia, ya que revolucionaron la forma de procesar imágenes con computadoras, son las Redes Neuronales Convolucionales, o *Convolutional Neural Networks* (*CNN*) en inglés. Hasta el momento, si quisiéramos entrenar una red neuronal con una imagen, teníamos que redimensionar la imagen de un arreglo de 2 dimensiones a 1 dimensión para que nuestra red neuronal sea capaz de procesarla. Éste fue el propósito principal por el cual las *CNN* fueron creadas, para poder hacer que la información de una neurona más cercana sea más relevante que la información brindada por otras neuronas que estén más alejadas. En problemas de análisis de imágenes, esto se traduce a hacer que las neuronas procesen pixeles que están más cercanos a sí mismas. Las redes neuronales convolucionales, nos permiten alimentarlas con entradas de datos de 1, 2 o 3 dimensiones, produciendo una salida de la misma dimensionalidad que la entrada. Ahora bien, las *CNN* aplican filtros convolucionales a nuestra entrada de datos, pero, ¿qué es una convolución? Las convoluciones consisten en tomar un grupo de píxeles cercanos, por ejemplo una matriz de 3x3 o 5x5, e ir realizando operaciones matemáticas (producto escalar) contra esta matriz de *nxn*. A esta matriz, se la suele conocer como *kernel* (no necesariamente tiene que haber uno solo), dicho *kernel* es aplicado a toda la imagen de entrada, de izquierda a derecha, de arriba hacia abajo, generando una nueva matriz resultante de salida. Este procedimiento lo podemos comprender mejor con la ayuda del siguiente gráfico:

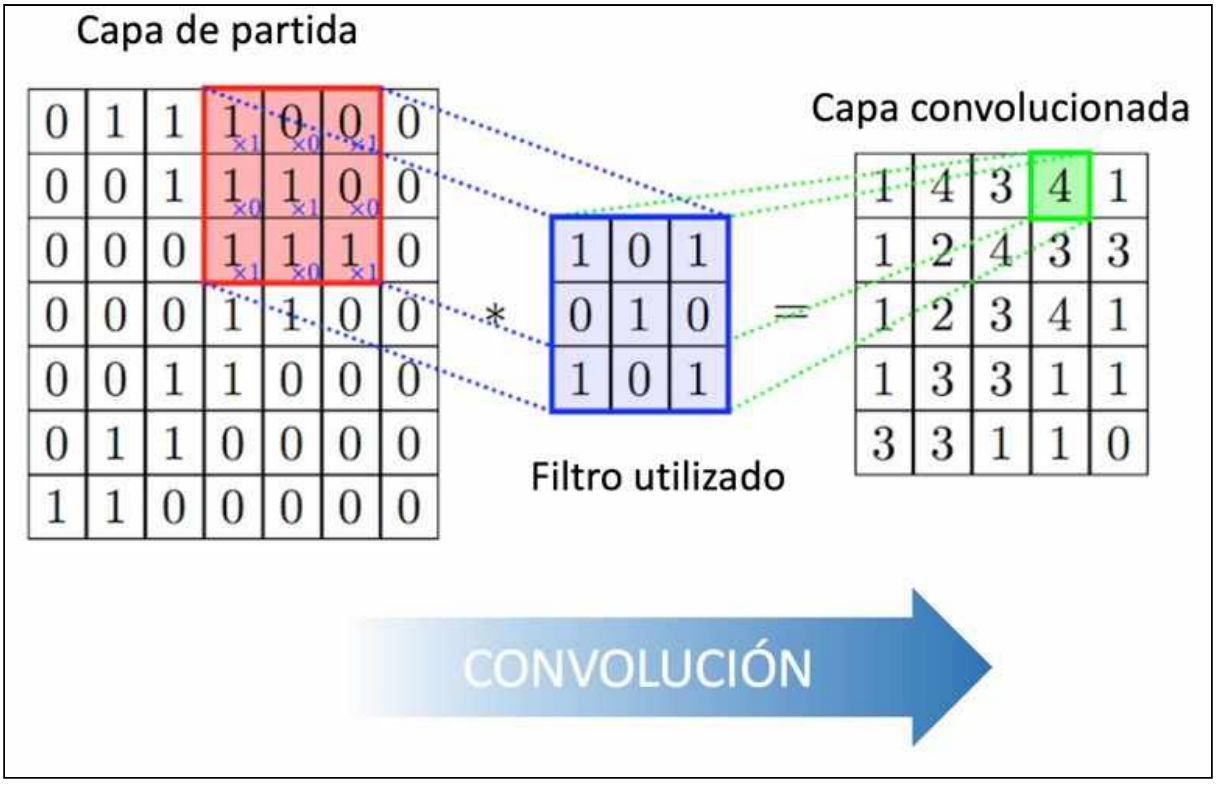

Figura 4.7 Representación gráfica de la convolución

Vale la pena destacar, que si nuestra imagen se trata de una imagen en color, este procedimiento se ejecuta para cada una de las matrices RGB y luego se hace un promedio de las 3 matrices resultantes.

La matriz resultante puede tener 2 tipos de resultados con respecto a las dimensiones de la matriz de entrada, una es que la matriz resultante sea menor y la otra es que se mantenga igual o aumente la dimensión. Esto depende de si se utiliza la técnica de *Valid Padding* o *Same Padding*. En el primer caso donde la dimensión se reduce se utiliza *Valid Padding* y cuando las dimensiones se mantienen o aumentan es porque se utiliza *Same Padding*. Esto lo podemos ver gráficamente en la siguiente imágen:

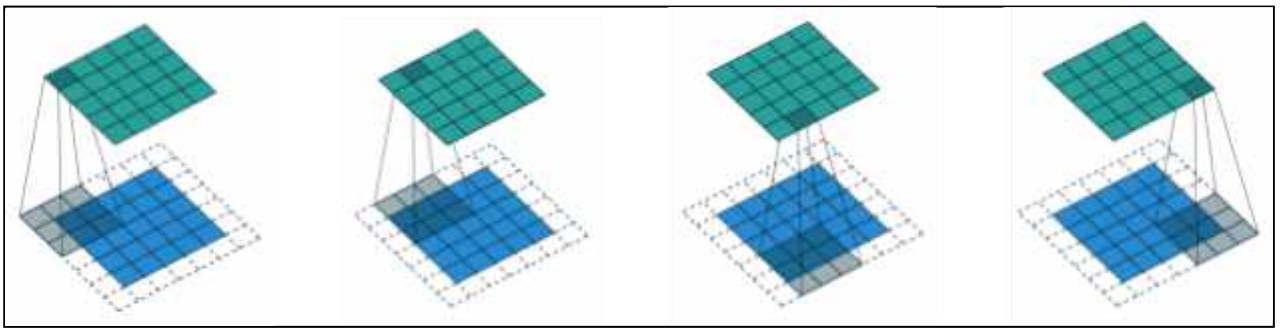

Figura 4.8 Cálculo de la convolución con y sin *Padding*

Como podemos observar, en el caso en que nuestro *kernel* utilice las posiciones que estén fuera de la imagen original, estamos en presencia de la técnica de *Same Padding*, ya que busca mantener las dimensiones de la imagen original. Por el contrario, si sólo aplicáramos nuestro *kernel* a la imagen concreta (denotada por la cuadrícula en azul) nuestra matriz resultante se va a ver reducida a la dimensión del *kernel*. Luego de obtener la matriz resultante, se le aplican a estas las funciones de activación que vimos anteriormente. Por lo general, en las *CNN* se utilizan las distintas variantes de la función *ReLu*.

En los últimos años, se estuvo investigando el hecho de que las *CNN*, no son capaces de asociar una posición a un objeto, por lo que si nuestra red neuronal detectara por ejemplo, una nariz, un par de ojos y una boca, seguramente detecte que en la imagen analizada hay un rostro, cuando en realidad, en la imagen podría haber un conjunto de estos objetos dispersos en toda la imagen sin relación alguna, y aún así, nuestra red neuronal detectaría un rostro en la imagen. Para solucionar esta problemática, es que se desarrolló un nuevo tipo de red neuronal, denominadas *capsule networks*. Este tipo de redes neuronales excede el alcance de esta investigación, pero vale la pena mencionarlas ya que son un tipo de redes neuronales que tienen mucho potencial y van adquiriendo cada vez más popularidad.

En el siguiente apartado veremos el concepto de *Transfer Learning*, por qué usarlo, distintas categorías y un ejemplo básico para entender en qué nos puede ayudar y ver la forma de utilizarlo.

2.2.2.3 Aprendizaje Transferido (*Transfer Learning*)

La técnica de *Transfer Learning* surge de la idea de que el conocimiento adquirido de un modelo para una tarea *T1* específica puede ser de utilidad para entrenar otro modelo para una tarea *T2* diferente pero con alguna relación a la primer tarea. El Aprendizaje Transferido, según Pan y Yang en "*A Survey on Transfer Learning*" [38], es una técnica del campo del *Machine Learning* que viene siendo investigada desde hace varios años que tiene distintas maneras de ser aplicada para distintos escenarios posibles, ya que no siempre se busca transferir los mismos datos aprendidos. Según Pan y Yang, la técnica de *TL* puede ser categorizada en base a los algoritmos de *ML* involucrados, principalmente las categorías planteadas son:

- *Inductive Transfer Learning*: en este escenario, los dominios de origen y destino son los mismos, pero las tareas de origen y destino son diferentes entre sí. Los algoritmos intentan utilizar los sesgos inductivos del dominio fuente para ayudar a mejorar la tarea objetivo. Dependiendo de si el dominio de origen contiene datos etiquetados o no, esto se puede dividir en dos subcategorías, similar al aprendizaje multitarea y el aprendizaje autodidacta, respectivamente.
- *Unsupervised Transfer Learning*: esta configuración es similar a la transferencia inductiva, pero focalizada en tareas no supervisadas en el dominio de destino. Los dominios de origen y destino son similares, pero las tareas son diferentes. En este escenario, los datos etiquetados no están disponibles en ninguno de los dominios.
- *Transductive Transfer Learning*: en este escenario, hay similitudes entre las tareas de origen y de destino, pero los dominios correspondientes son diferentes. En esta configuración, el dominio de origen tiene muchos datos etiquetados, mientras que el dominio de destino no tiene ninguno. Esto, a su vez, se puede dividir haciendo referencia a configuraciones donde los espacios de características son diferentes o las probabilidades marginales.

Particularmente, el *TL* ha tomado una importancia considerable en el campo del *DL* (que es donde aplicaremos esta técnica) ya que en general, se requiere de conjuntos de datos bastante grandes para lograr buenos resultados. Es por esto que se utiliza, por lo general, el conocimiento adquirido de modelos previamente entrenados (InceptionV3, MobileNet, VGG, ResNet, Xception, etc) sobre conjuntos de datos globalmente conocidos como el ImageNet [41] [42] [43] (actualmente contiene 14 millones de imágenes etiquetadas jerárquicamente de distintas categorías), entre muchos otros [44] [45] [46] [47]. En base a este conocimiento previo se realizan las nuevas extracciones de características sobre el conjunto de datos de

la nueva problemática planteada. Hay diversas formas de aplicar esta técnica ya que se puede utilizar el conocimiento de cierto número *n* de capas y el resto entrenarlas nuevamente con el nuevo conjunto de datos específico de la problemática en cuestión, esto lo podemos ver bien ejemplificado en el artículo de Dipanjan Sarkar "A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning" [39].

Veamos un ejemplo sencillo para esclarecer esto último; si el conocimiento adquirido en nuestra tarea *T1* era para detectar distintos tipos de comida según su forma y la tarea *T2* planteada es saber el estado de distintos alimentos (rebanados, rallado, harina, etc), a nosotros nos interesaría poder transferir más las etapas iniciales del modelo *T1* ya que se tratan representaciones y características más generales en vez de características que nos puedan ayudar a determinar si es un tomate o una zanahoria dependiendo su geometría. De este modo, es que podemos utilizar la técnica de *TL* de diversos modos, ya que podríamos utilizar todo el conocimiento obtenido (si nuestras tareas origen y fin son bastante similares) o podríamos utilizar solo las representaciones generales aprendidas al comienzo y entrenar las capas que obtienen características más específicas de nuestros objetos a analizar.

En el siguiente apartado veremos el concepto de *Deep Learning* y varios tipos de capas utilizadas en los modelos utilizados en esta investigación, como por ejemplo, *convolution layers*, *pooling layers*, *dropout layer*, entre otras más.

2.2.3 Aprendizaje Profundo (*Deep Learning*)

El campo del *Deep Learning*, se sustenta de los conocimientos que tenemos sobre cómo funciona nuestro cerebro (haciendo foco principalmente en la forma en que nuestro cerebro aprende y no en la unidad principal, ya que esta última fue lo que impulsó el campo de las redes neuronales como vimos anteriormente) y busca aplicar este funcionamiento para procesar modelos de *Machine Learning* donde los problemas se tornan mucho más complejos. Problemas clásicos que se pueden resolver de manera satisfactoria con este nuevo enfoque, son, por ejemplo, problemas de reconocimiento de imágenes o de análisis de textos, donde nuestros datos se tornan extremadamente dimensionales, y las relaciones que queremos encontrar se transforman en relaciones altamente no lineales, dificultando la clasificación de nuestro conjunto de datos.

La semejanza que hay entre el aprendizaje del cerebro humano y el *Deep Learning* radica en que el aprendizaje se logra debido a la gran cantidad de capas que extraen características de relevancia que el modelo va a utilizar para poder distinguir el objeto en cuestión. Esta red con múltiples capas intermedias que logran capturar características de interés, la podríamos graficar de la siguiente manera:

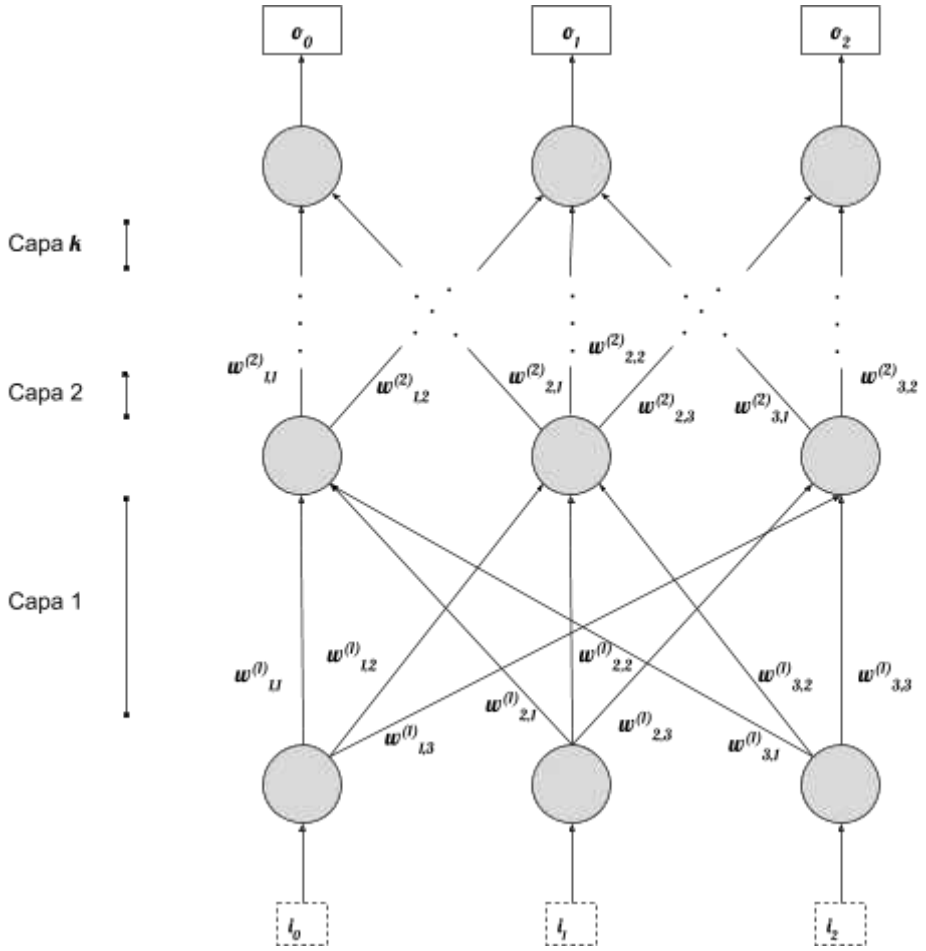

Figura 4.9 Representación del funcionamiento en capas de las neuronas

Como se muestra en la figura anterior, vemos que las entradas a la red se las suele denotar como i_x, los pesos de cada enlace como $w^{(k)}_{i,j}$, que representa el peso de la conexión entre la neurona **i** en la capa **k** con la neurona **j** de la capa **k+1**, y los resultados como o_x. Estos pesos constituyen el vector paramétrico θ, y como vimos previamente, la resolución de problemas de *Machine Learning* se basa en la búsqueda del parámetro óptimo. Las redes neuronales, generalmente, están compuestas por 3 grupos de neuronas, las iniciales, las intermedias y las terminales. Las neuronas de la capa inicial son las que captan la información de entrada. Las neuronas de la capa intermedia, también conocida como *hidden layer,* se encarga de hacer todo el aprendizaje de las características relevantes de nuestro objeto. Por último, la capa final, se encarga de computar todas las características obtenidas y brindarnos un resultado. Como podemos apreciar en la Figura 4.8, las neuronas de esta red sólo se comunican con las neuronas del siguiente nivel, a este tipo de red se las conoce como redes *feed forward*, y son las mayormente utilizadas en *Deep Learning*. Las comunicaciones de las neuronas no siempre tienen que suceder como en las redes *feed forward*, podría suceder que una capa de nivel superior envíe información a capas de nivel inferior, obteniendo conexiones más sofisticadas. Vale la pena aclarar que:

- Cada capa, no necesariamente debe tener la misma cantidad de neuronas.
- No es obligatorio que cada neurona de la red esté conectada con todas las neuronas del siguiente nivel, de hecho, se dice que el diseño de conexión entre neuronas es un arte que se gana con la experiencia.
- Las entradas y salidas son representaciones vectorizadas.

Ahora que tenemos un mayor conocimiento de cómo operan las redes, las podemos reformular matemáticamente como una serie de operaciones de vectores y matrices. Consideremos a la entrada de datos ***i-ésima*** cómo un vector **x =** [x_1, x_2, …, x_n]. Nuestro objetivo es obtener el vector **y =** [y_1, y_2, …, y_n] producido por la propagación de las entradas por las neuronas. Para lograr esto, podemos expresarlo como una multiplicación de matrices si definimos **W**, de tamaño ***n x m,*** y un vector de tamaño ***m*** que representa los sesgos de cada uno. Cada una de las columnas de la matriz ***W*** corresponden a una neurona, donde el ***j*-ésimo** elemento de la columna corresponde al peso de la conexión del ***j*-ésimo** elemento de la entrada. En otras palabras, podríamos representar esta operatoria como una función **y =** $f(W^Tx + b)$, donde la función de transformación se aplica al vector de forma elemental.

Este tipo de funciones, se las llaman funciones lineales y tienen grandes limitaciones a la hora de obtener características relevantes de nuestros objetos. Además, si nuestra red *feed-forward* está compuesta sólo por funciones lineales, se puede demostrar que se puede expresar como una red sin *hidden layer*, lo que nos llevaría a no obtener características de gran importancia para la clasificación de nuestro objeto. Para mejorar la obtención de estas características, generalmente, se opta por tener neuronas no-lineales, esto es, neuronas que utilizan una función con algún tipo de no linealidad. Como vimos en el apartado anterior, algunas de estas funciones que se utilizan como funciones no-lineales dentro del campo de *deep learning*, pueden ser: *sigmoid*, *tanh y ReLu*.

La neurona sigmoide utiliza la función $f = 1 / (1+ e^{-z})$. Esta función es la inversa de la función *logit*, que calcula el logaritmo de las probabilidades. La función *sigmoid* se la utiliza, generalmente, para mapear valores reales a probabilidades; es muy frecuente, al finalizar una clasificación, querer asignar una probabilidad para saber qué tan probable es que suceda ese evento. El comportamiento de la función *tanh*, es muy similar a la de la función *sigmoid*, salvo que esta mapea valores en el rango de [-1,1]. Se podría querer utilizar está función, ya que es una función que está centrada en el 0. Por último, la función *ReLu*, utiliza la función $f = \max(0,z)$, esta función, nos permite tener una gráfica de comportamiento 0 hasta x=0. Las gráficas de estas funciones son las siguientes:

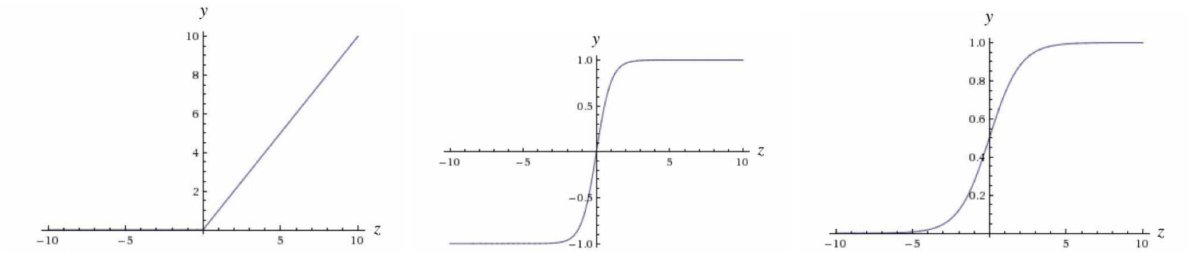

Figura 5.0 Gráficas de las funciones ReLu, *tanh, sigmoid* respectivamente

Para dejar en claro, lo que hace al *Deep Learning* un subcampo particular de *Machine Learning*, es que *Deep Learning* hace uso específico de un aprendizaje en gran cantidad de capas, basadas en redes neuronales, especialmente las convolucionales. En cada capa de su modelo, pone énfasis en ganar capas sucesivas de representaciones cada vez más significativas. Por esto, cuando se lo nombró "Aprendizaje Profundo" al subcampo, se buscaba que sea notoria la particular forma de aprender que tienen sus modelos, que pueden llegar a contener decenas, incluso cientos de capas sucesivas de representaciones, en contrapartida de otras ramas del *Machine Learning* que puede incluir una o dos capas de representaciones de datos. Para expresarlo de otra forma, el Aprendizaje Profundo se lo puede definir técnicamente como una forma de etapas múltiples para aprender representaciones de datos.

Este subcampo, también hace uso de la función denominada *loss function* que vimos en la sección anterior, que se utiliza luego de computar el resultado de la clasificación al compararlo con el resultado esperado que nosotros habíamos brindado en un comienzo. Esta función lo que hace es retroalimentar a la red, con ayuda de la técnica de *backpropagation*, con un valor extra sobre la relación entre el resultado obtenido y el esperado, denominado *loss score*, que con la ayuda de un optimizador, se logra corregir los pesos para lograr direccionar los resultados de la clasificación a algo más cercano al resultado esperado. Este método se logra gracias a la iteración del proceso de obtención de características, dado que cuando se entrena una red de estas características se suele hacerlo durante varias iteraciones, denominadas *epochs* o épocas en castellano. Podemos ver esta idea en el gráfico siguiente:

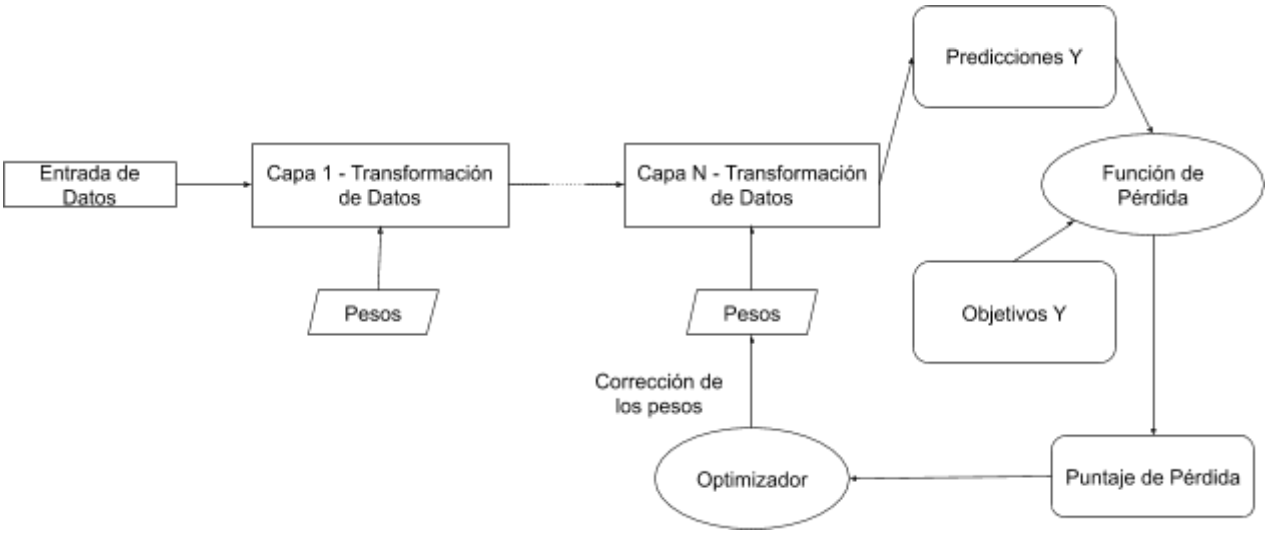

Figura 5.1 Representación de cómo se utiliza la retroalimentación para ajustar los pesos

Además de lo dicho anteriormente, otra característica que lo hace resaltar dentro del campo de *ML*, es que el *Deep Learning* hace uso de una gran cantidad de datos, dado que precisa ir adquiriendo características de diversas fuentes, con la mayor cantidad posible de variaciones, para poder lograr una mejor generalización del modelo. Una técnica muy utilizada, en el campo del *Deep Learning* que sirve para esto es la técnica de aumentación de datos o ***data augmentation*** en inglés. Esta técnica implica generar nuevas representaciones de las imágenes, aplicando transformaciones (rotando, traducir y/o escalar, agregando algo de ruido), a partir del conjunto de datos inicial.

A continuación, veremos las capas utilizadas en los modelos de *Deep Learning* que fueron utilizados durante el desarrollo de esta tesina, que se verán en el "Capítulo 3: Trabajos Realizados", estas son:

- *Input Layer*
- *Convolution Layer*
- *Batch Normalization Layer*
- *Pooling Layer*
- *Lambda*
- *Concatenate*
- *Flatten*
- *Dense*
- *Dropout*

Comenzaremos por ver la capa de ***Input***. Este tipo de capa es utilizado en Keras (en la siguiente sección definiremos las librerías utilizadas) como punto de partida para inicializar tensores. Un tensor de Keras es un objeto tensor del *backend* base utilizado (en nuestro caso TensorFlow). Keras, aumenta la funcionalidad del tensor del *backend*, agregándole ciertos atributos que nos permiten construir un

modelo de Keras simplemente sabiendo las entradas y salidas del modelo. Estos atributos agregados son: *_keras_shape* y *_keras_history*.

Luego de la capa *Input*, la siguiente capa utilizada es ***Convolution Layer***, esta es de las capas más utilizadas dado que es la que aplica filtros convolucionales (vistos en la sección anterior) en nuestra red neuronal. Esta capa consiste en un conjunto de filtros (también conocidos como *kernels* ó *feature detectors*), donde se aplica cada filtro a través de todo el área de los datos de entrada. La entrada de datos, es una imagen, con lo que nuestro vector de entrada tendría el formato [224, 224, 3] (estas dimensiones fueron las utilizadas durante la investigación), donde se especifica el alto y ancho de la imagen y los 3 canales de colores RGB. A estos datos se le aplican los filtros previamente mencionados, que da como salida la suma ponderada de sus entradas aplicando dicho filtro convolucional que es una matriz de igual dimensiones, ya sea 2x2, 4x4, 5x5, *n*x*n*. El propósito es el de resaltar una característica particular de la entrada, como por ejemplo un lado, una línea, etc. El grupo de neuronas cercanas, que participan en la entrada, se denomina **campo receptivo**. En el contexto de la red, la salida del filtro representa el valor de activación de una neurona en la siguiente capa. Este proceso de aplicar la matriz de dimensión *n*x*n* y calcular la suma ponderada de esta porción de datos se lo repite para toda la imagen a analizar, por esto decimos que se desplaza la matriz y se recomputan los valores para la neurona en cuestión. El conjunto de salida de las sumas ponderadas va a ser la entrada de la siguiente capa dentro de nuestra red neuronal convolucional. A continuación podemos ver una representación gráfica de dicho proceso.

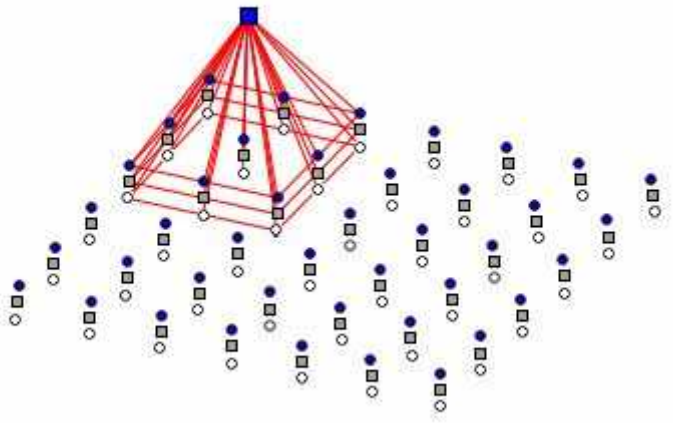

Figura 5.2 Ejemplo de un slice de profundidad 3 en entrada *RGB*

Dicho ***slice***, se irá desplazando por el ***stride*** que tenga configurado la capa convolucional, por toda la entrada de datos. El *stride* es la cantidad de unidades que se va a desplazar la matriz una vez calculado el peso para la porción de datos actual.

La siguiente capa que encontramos, se trata de una ***BatchNormalization***, este tipo de capas normaliza las activaciones calculadas por la capa anterior por cada *batch* y aplica una transformación que mantiene las activaciones media cerca de 0 y la desviación estándar cerca de 1. Según la investigación de Sergey Ioffe y Christian Szegedy [35], las redes que utilizan la capa de normalización por lotes, demoran menos tiempo en el entrenamiento y pueden utilizar mayores tasas de aprendizaje.

A continuación de la capa de normalización, veremos la capa de ***Pooling***, este tipo de capa se utiliza para aumentar el campo de recepción de las neuronas de salida. Con esto queremos decir, que el objetivo de este tipo de capas es el de disminuir la potencia computacional requerida para procesar los datos a través de la reducción de dimensionalidad, extrayendo las características dominantes que son invariantes posicionales y de rotación, manteniendo así el proceso de entrenamiento efectivo del modelo. Existen 2 tipos de capas de agrupamiento, las *MaxPooling* (retornan sólo el valor máximo de la submatriz de agrupamiento) y las *AveragePooling* (retornan el promedio de los valores de la submatriz de agrupamiento). Estas capas dividen el segmento de entrada en una cuadrícula, donde cada celda representa un campo receptivo de un número de neuronas. Luego de esto, una operación de agrupación es aplicada por cada celda de la grilla. Estas capas no cambian la profundidad del volumen, porque cada operación es llevada a cabo independientemente en cada segmento. En particular, en este trabajo, se utiliza la capa *Max Pooling*, esta capa, como se dijo previamente, toma el valor de activación más grande en cada una de las grillas y propaga sólo ese valor.

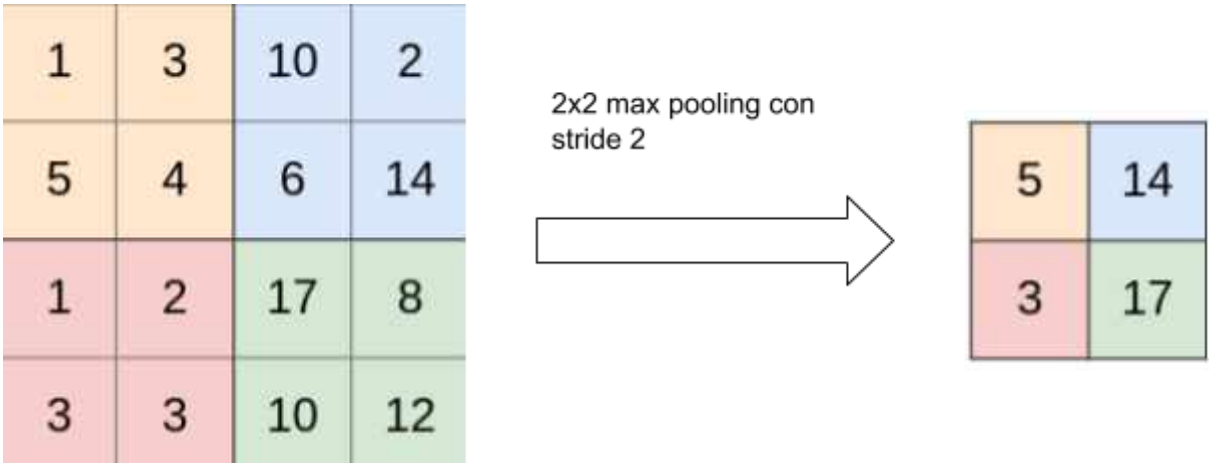

Figura 5.3 Funcionamiento de una capa *Max Pooling* con stride 2

La siguiente capa que veremos, es una capa de tipo ***Lambda***, la misma nos permite ejecutar una función arbitraria como si fuese una capa del modelo. En uno de los modelos que veremos en el capítulo 3, utilizamos esta capa para poder dividir el mapa de características en 2, y así poder generar los distintos flujos de procesamiento.

Otro tipo de capa que se utiliza es la capa ***Concatenate***. Esta capa se la utiliza para unificar los tensores de características de cada flujo de procesamiento de nuestro modelo. El funcionamiento general de esta capa es el de recibir como entrada una lista de tensores de igual forma (excepto el eje de concatenación) y retorna un tensor único con la concatenación de todos los tensores de entrada.

Por último, una vez que concatenamos todos los tensores de características obtenidos, se utilizan 3 tipos de capa finales, la capa *Flatten, Dense* y *Dropout*.

La capa ***Flatten***, genera que nuestros tensores de cierta forma pasen a ser unidimensionales, por lo que un tensor de la forma (64, 32, 32) se transformaría en uno de la forma (None, 65536), modificando la dimensionalidad del mismo.

La capa ***Dense***, implementa la operación *salida = activation(dot(input, kernel) + bias)*, donde la activación es la función de activación del elemento que se pasa como el argumento de activación, el kernel es una matriz de pesos creada por la capa y el sesgo (*bias*) es un vector de sesgo creado por la capa.

El último tipo de capa que veremos, es la capa **Dropout**. Esta capa se la utiliza generalmente como capa de optimización luego de capas convolucionales [49] [50]. La técnica consiste en eliminar las salidas de ciertas neuronas aleatoriamente. Cada neurona tiene la misma probabilidad *p* de ser eliminada, esto se hace para asegurar que ninguna neurona termine confiando demasiado en otras neuronas y aprenda algo útil para la red en su lugar. Esta técnica es ampliamente utilizada para combatir el *overfitting* durante la etapa de entrenamiento. En la siguiente figura, podemos ver gráficamente el funcionamiento de esta técnica.

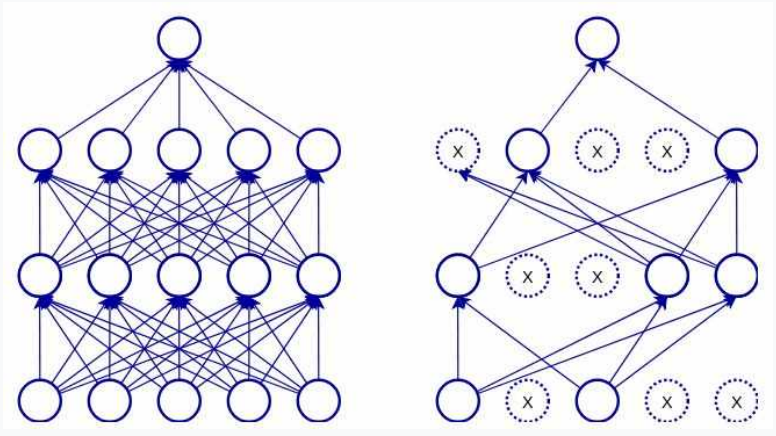

Figura 5.4 Ejemplo de Dropout con una capa fully-connected

Con lo visto hasta el momento, podemos tener una idea más clara de cómo funcionan las redes profundas, lo cual nos permitirá una mejor comprensión del desarrollo de la tesina cuando expliquemos las decisiones y los experimentos realizados en la parte que corresponde a *Deep Learning*, en el capítulo 3.

2.3 Frameworks y Librerías

2.3.1 OpenCV

Open-cv es una librería que contiene una gran cantidad de algoritmos relacionados con *Computer Visión* y *Machine Learning*. El proyecto se comenzó en 1999 en los laboratorios de Intel a cargo de Gary Bradsky. Su primer lanzamiento fue en el año 2000.

En este caso particular, utilizaremos la librería OpenCV-Python, que es una API de python para la librería OpenCV, para utilizar algoritmos de clasificación de texturas y luego poder comparar el comportamiento de este con el enfoque de *Deep Learning*. Elegimos utilizar esta librería dado que el lenguaje Python nos brinda varias librerías que son de utilidad para el desarrollo de la tesina.

2.3.2 TensorFlow

Como se expresó en el capítulo 1, se eligió para el desarrollo de esta investigación la librería TensorFlow como soporte de procesamiento de *Machine Learning*.

TensorFlow es una biblioteca de software libre que se utiliza para realizar cálculos numéricos mediante diagramas de flujo de datos. Los nodos de los diagramas representan operaciones matemáticas y las aristas reflejan las matrices de datos multidimensionales (tensores) comunicadas entre ellas.

Fue producto del trabajo de investigadores e ingenieros de Google, que formaban parte de la organización de investigación del aprendizaje automático. Su objetivo era realizar investigaciones en el campo del aprendizaje automático y las redes neuronales profundas. A pesar de que este era su propósito inicial, se trata de un sistema lo bastante general como para poder aplicarse en muchos otros campos.

2.3.3 Keras

Keras es una API de alto nivel escrita en python que corre sobre Theano, CNTK o TensorFlow. En esta tesina se eligió TensorFlow como backend ya que es la librería recomendada por la documentación de Keras. Esta librería surge con la idea de permitir un ágil prototipado de los modelos para no perder tanto tiempo en el desarrollo del mismo y así poder invertir este tiempo en cuestiones de diseño más relevantes. Para tener funcional esta librería se requiere tener instalado el *backend* que se desee utilizar (en este caso TensorFlow) y cuDNN para poder acelerar el procesamiento del entrenamiento con una GPU, si es que se dispone de una.

CUDA es una plataforma de procesamiento paralelo y un modelo de programación desarrollado por NVIDIA. Permite un aumento dramático en el

rendimiento de las computadoras al aprovechar el poder de cómputo de las unidades de procesamiento gráfico (GPU *Graphics Processing Unit*).

En el "Apéndice" de este trabajo, disponemos de una sección de cómo instalar esta librería y sus dependencias.

2.3.4 Ionic

Ionic es un framework que nos permite crear aplicaciones móviles aprovechando las tecnologías de desarrollo web. Es una librería de código abierto que nos permite crear aplicaciones web que se ven en nuestros celulares como si fuesen nativas. Al basarse en tecnología web, nos permite portar una misma aplicación a diferentes sistemas operativos móviles, lo cual nos simplifica el desarrollo de la aplicación, ya que, de otro modo, deberíamos tener al menos 2 aplicaciones que hagan lo mismo pero escritas nativamente en los lenguajes que entiende cada sistema operativo. Ionic, está compuesto por código HTML, CSS y Javascript, y se sustenta de Apache Cordova, que nos permite acceder a diversos periféricos del *smartphone*.

Figura 5.5 Esquema de funcionamiento de Ionic

2.3.5 Django

Django es un *framework* web que permite y fomenta un ágil desarrollo de aplicaciones. Posee una gran cantidad de librerías que facilitan el desarrollo de las tareas más comunes a la hora de crear aplicaciones web. Es un *framework* de código abierto lo que permite una gran flexibilidad a la hora de personalizar la

aplicación. Además, brinda un conjunto de herramientas por defecto que lo hacen especial para proyectos con tiempos ajustados, por ejemplo, el micro *framework* de autenticación y permisos que posee nos brinda un gran abanico de posibilidades, además de una página de administración por defecto, que nos permite interactuar con los objetos de nuestro modelo. En el Apéndice de este trabajo, hay una sección relativa a Django que nos muestra todo lo necesario para poder utilizar este y así poder replicar el trabajo realizado en esta investigación.

2.4 Estado del Arte

Hoy en día, los estudios relacionados con técnicas de procesamiento de imágenes por computadora en relación a la gastronomía, se centra, mayoritariamente, en la clasificación de la calidad de la misma o en la identificación de la misma, ya sea en diferentes estados, comidas, platos. Uno de los estudios que se encontró que se relaciona con tipos de comida y clasificación de la misma fue, "*Food Classification with Deep Learning in Keras / Tensorflow*" [21], en el cual el autor, genera un sistema capaz de distinguir comidas, basándose en un *dataset* ya armado, que contiene distintas clases ya clasificadas. El dataset utilizado es procesado con un modelo desarrollado con la librería Keras, utilizando TensorFlow como backend de procesamiento de *Machine Learning*, para dicha librería.

Otro trabajo que utiliza la comida como base de investigación, es "*Real-time Mobile Recipe Recommendation System Using Food Ingredient Recognition*" [22], en este trabajo, el objetivo es lograr un sistema recomendador de recetas a partir de la detección de ingredientes que capta el dispositivo del usuario. El trabajo, arriba a un 83,93% de aciertos dentro del top 6 de recetas candidatas. Utiliza el algoritmo SURF de la librería OpenCV, para obtener el vector de características y luego usa un clasificador *SVM* lineal para lograr la clasificación de las características obtenidas. El reconocimiento de objetos se lleva a cabo a través de imágenes captadas por el dispositivo del usuario del sistema.

Una idea similar a la del trabajo [22], fue la del estudio que se llevó a cabo en Japón, por los investigadores Yushiyuki Kawano y Keiji Yanai en "Real-time Mobile Food Recognition System" [23], que proponían un sistema para dispositivos móviles Android que pudiera reconocer alimentos para un posterior cálculo y registro de las calorías y nutrientes consumidos por el usuario, permitiéndole al usuario ver sus hábitos alimenticios durante el día.

Otras investigaciones más relacionadas con el estudio de la carne, se centran en la clasificación de la grasa o de la ternura de la carne para poder determinar la calidad de la misma. Estos estudios están incentivados del hecho que hoy en día es complicado poder determinar la calidad de un corte de carne determinado sin tener que hacer técnicas que consumen tiempo y son destructivas de los cortes que se

utilizan para llevar a cabo la misma. Algunos ejemplos de este campo de investigación son, "*Development of a flexible Computer Vision System for marbling classification*" [11], "*Automatic assessment of meat marbling and tenderness*" [24], "*Application of Artificial Neural Networks in Meat Production and Technology*" [8], "*Beef Quality Identification Using Thresholding Method and Decision Tree Classification Based on Android Smartphone*" [9]. Todos estos trabajos, buscan parámetros con los cuales se pueda llegar a determinar la calidad de la carne, algunos a través del análisis de la grasa, otros del color, otros por la textura del corte, utilizando técnicas de procesamiento de imágenes, lo que permitiría una rápida clasificación, eficiente, poco costosa y no destructiva.

Como podemos ver, todas las investigaciones de hoy en día que están orientadas a la carne, buscan poder determinar la calidad, pero ninguna se centra en la cocción de la misma. Un trabajo que se focaliza en el cambio de estado de los alimentos es "*Identifying Object States in Cooking-Related Images*" [3], pero la investigación tampoco está centrada para la cocción de carne, es un trabajo que analiza a los alimentos en general y logra clasificar y detectar diversos estados de un producto. Por esto concluyo, que hoy en día, según mi conocimiento, no se ha realizado ningún trabajo específico en el área de identificación del estado de cocción de la carne, que es el propósito de este trabajo, lo cual va a ser un punto de partida para cualquiera que quiera seguir investigando la detección de los cambios de estado de cocción en cortes de carne.

2.5 Revisión del objetivo

Habiendo enunciado y explicado aquellos tópicos que son necesarios para lograr una mayor comprensión sobre el desarrollo del trabajo, volveremos a revisar el propósito de esta investigación, para refrescar el objetivo.

Como vimos en este capítulo, los algoritmos del campo de Inteligencia Artificial, son muy buenos para detectar y analizar imágenes. Sumado a esto, el surgimiento del *Deep Learning* ha sido una revolución para el área, logrando crear aplicaciones con muy buenos porcentajes de acierto, que resuelven problemas que las técnicas anteriores no lo lograban con tanta facilidad.

Cada una de estas técnicas tienen sus propósitos específicos, y no siempre se comportan de la misma manera, por esto hay que hacer una selección de aquellos algoritmos y modelos que se adecúen mejor al propósito.

Como vimos en el capítulo 1, el objetivo es realizar un sistema capaz de clasificar los estados de cocción de carne vacuna, durante su cocción. Para lograr esto, se investigará qué técnica aplica mejor a esta problemática. Por esto, luego de obtener los resultados con cada enfoque, hay que hacer una comparativa para ver cuál se adapta mejor.

Para los modelos que se van a investigar en el área de *Deep Learning*, se va a requerir una gran cantidad de imágenes para lograr entrenar el modelo. Por lo que a mi respecta, no existe, hoy en día, un *dataset* que sea sobre carne vacuna, por lo tanto se desarrolló uno desde cero.

Por otro lado, para lograr una aplicación prototipo que utilice nuestro modelo, debemos generar una arquitectura de aplicaciones que nos permita acceder a nuestro clasificador desde internet, luego de haber seleccionado el que se adecúe mejor. Esto permite que el clasificador sea accesible desde distintos tipos de dispositivos, si así se desea, lo único que habría que hacer es desarrollar la aplicación para la plataforma en cuestión que consuma el *backend* implementado. Para este trabajo, se optó por una aplicación híbrida para dispositivos móviles, ya que la idea es brindar el servicio a tantos usuarios como sea posible, dado que el objetivo es brindar una herramienta útil para aquellos que lo necesiten.

Capítulo 3: Trabajos Realizados

Una vez vistos todos los conceptos introductorios del capítulo 2, para lograr una mejor comprensión del desarrollo del trabajo, comenzaremos a describir la investigación llevada a cabo. Primero vamos a focalizarnos en todas las aplicaciones que dan soporte al uso del clasificador, estas son: la aplicación que actúa de *backend*, la aplicación móvil híbrida y la generación del *dataset* utilizado para el entrenamiento. A continuación podemos ver un diagrama de la arquitectura planteada para el sistema:

Figura 5.6 Arquitectura general del sistema planteado

3.1 El *Dataset*

Como vimos en los capítulos anteriores, una parte esencial de este trabajo, se basa en las imágenes que necesitamos para poder entrenar nuestro modelo. Como se mencionó previamente, no existe un *dataset* de características que se adecúen a las necesidades de la investigación en curso (al menos ninguno que conozca al día de hoy). Por este motivo, se comenzó a desarrollar un *dataset* propio que al momento consta de 650 imágenes en crudo.

Del conjunto total de datos, se generaron 5 categorías diversas que representan los diversos estados de cocción que se pueden detectar en la carne vacuna. Si bien esta división es muy subjetiva, dado que no hay un límite bien estipulado que defina un estado del otro y para cada persona puede que existan diversos estados y que cada uno tenga ciertas características, se generó este etiquetado ya que así parecía una cantidad prudente de categorías, que no era ni muy limitada ni tampoco tan específica. Las imágenes se fueron tomando con distintos dispositivos, con diversas cualidades cada uno; algunos de estos, fueron un celular BLU HD 5.0 que consta con una cámara de 13 Megapíxeles (su sigla

MP), otro fue un Iphone 5 que posee una cámara de 8 MP y un Samsung J1 con cámara de 5 MP, entre otros. Todos estos celulares poseen flash, lo cual permitía obtener una mayor variación de características en las imágenes tomadas, además de la calidad de la imágen ya que cada dispositivo constaba con una cámara con distintas características.

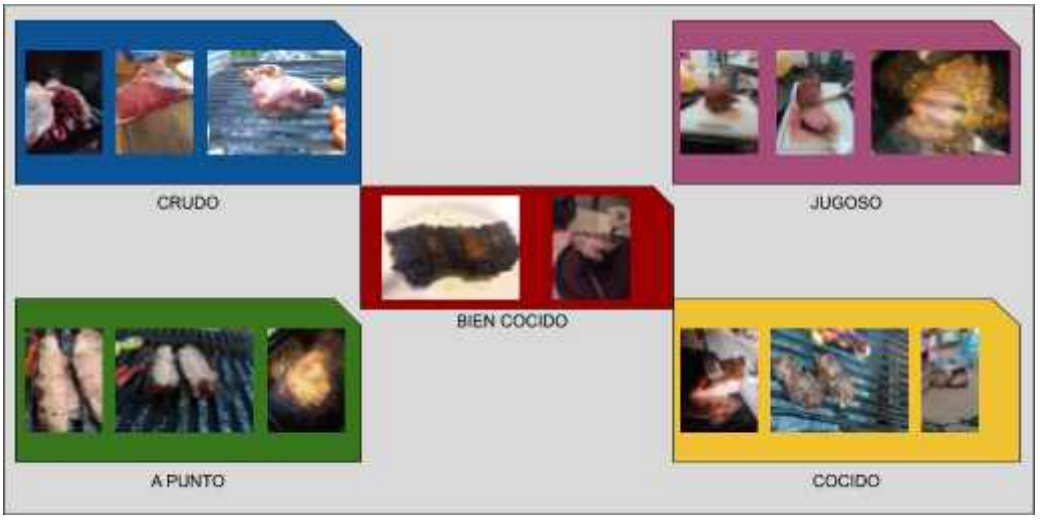

Figura 5.7 Categorías del dataset

Sumado al hecho de la diversidad tecnológica de cada dispositivo, se logró obtener una gran variación en los escenarios donde se obtenían las imágenes. Esta variación de los escenarios, quiere decir que hay fotos en exteriores e interiores, con diversos métodos de cocción, como por ejemplo, horno, parrilla, plancha o cacerola. Esto nos permite tener un conjunto de datos mucho más heterogéneo, lo cual ayuda a que el modelo pueda extraer mayor cantidad de características, que nos permitan detectar las características relevantes del objeto en análisis. Las imágenes que podemos apreciar en la figura anterior, nos muestran la variación en todos estos parámetros que se mencionan.

Posteriormente, cada una de estas imágenes se las clasificó manualmente etiquetándolas bajo una de las categorías que se decidió para este *dataset*, estas son: crudo, jugoso, a punto, cocido y bien cocido. Otro aspecto a tener en cuenta a la hora de generar el *dataset*, es la cantidad de imágenes que tenemos por cada categoría, ya que tener un conjunto de datos heterogéneo en la cantidad de fotografías de cada una de las etiquetas llevaría a que nuestro modelo tuviese cierta tendencia a clasificar una categoría sobre otras, ya que no aprendería de manera homogénea las cualidades de cada categoría. Esto último, se decidió hacerlo así, ya que al utilizar el *dataset* con distintas cantidades de imágenes en cada una de las categorías en los primeros ensayos, tendía a categorizar las imágenes del conjunto de datos de validación a una categoría que tenía más cantidad de imágenes que otras.

Luego de los primeros ensayos, se optó por modificar el *dataset* con las imágenes en crudo, editando cada una de las imágenes para que se enfocara en primer plano el corte de carne vacuna. Se llevó a cabo este paso dado que el modelo podría estar obteniendo características del entorno que no serían relevantes a la hora de realizar la evaluación de cocción sobre el corte de carne. Por este motivo, se brinda el *dataset* en 2 formatos, la versión *raw* y aquella en la cual se centra la imagen en el objeto. Este conjunto de datos fue utilizado para todos los modelos de clasificación propuestos con un tamaño de imagen de 224 x 224 en su formato *RGB*, si el algoritmo lo permitía.

3.2 La aplicación Django, el *backend*

Continuando con el desarrollo del trabajo, procederemos a hacer una revisión de la aplicación *backend* que captura las peticiones de nuestra aplicación móvil y utiliza el procesamiento de nuestro modelo de clasificación, que describiremos en la sección "3.4 El modelo de clasificación".

La aplicación *backend*, es una aplicación desarrollada en python con el *framework* Django versión 2.1. Es una aplicación sencilla ya que no se requiere demasiada lógica en la misma. Se generó una *API* muy reducida que brinda un *endpoint* para poder clasificar una imagen en particular. Esta *API* se la generó utilizando la librería Django Rest Framework. Además, la *API* nos permite subir fotos y almacenarlas en nuestra base de datos si así lo quisiéramos. Esto, se lo diagramó así para poder escalar la aplicación en caso de ser necesario, ya que nos permitiría generar un esquema donde guardemos usuarios que tengan imágenes relacionadas y podríamos guardar los resultados de la clasificación en nuestra base de datos, permitiendo que el usuario pueda consultar sus clasificaciones en cualquier momento y nosotros pudiéramos consumir esta *API* desde cualquier aplicación *frontend*.

Ahora, veremos un poco el código de la aplicación para poder entender el funcionamiento de la misma.

```
CLASSES = {
    0: 'CRUDO',
    1: 'JUGOSO',
    2: 'A PUNTO',
    3: 'COCIDO',
    4: 'BIEN COCIDO'
}
```

Figura 5.8 Diccionario de clases para el clasificador

Como podemos ver en la figura anterior, se utiliza un diccionario con el propósito de poder devolver a la aplicación *frontend*, un resultado más significativo, y no solo a que categoría corresponde la imágen en cuestión, ya que no podrían determinar cuál es cada una de ellas; sumado a esto, ya que nuestro modelo es de regresión, se debe utilizar una función escalón que lleve nuestro valor real a un valor natural, por ejemplo, si nuestra clasificación devolvió el resultado 2,77, se hace un redondeo al valor natural más cercano, en este caso 3, por lo que la *API* devolvería el resultado 'JUGOSO' a la aplicación *frontend*.

Por otra parte, tenemos un modelo minimalista, que permite almacenar la imagen subida, como se había mencionado previamente. Esto lo podemos ver en la siguiente imagen, donde se declara una clase "*Picture*", que contiene 2 campos, la imagen en sí y el resultado de la clasificación.

```
class Picture(models.Model):
    classification_result = models.CharField(max_length=50, null=True, blank=True)
    picture = models.ImageField(upload_to="pic_folder/")
```

Figura 5.9 Modelo que representa una imagen con su resultado de clasificación

Como se mencionó en capítulos anteriores, la aplicación requiere un objeto serializador que permite transformar objetos a un formato entendible para la *API*, que generalmente estas peticiones utilizan el formato *JSON*. Esto se hace para lograr que el *backend* trabaje con objetos que es lo que procesa normalmente ya que se hace uso de una base de datos relacional y el *ORM*. El serializador funciona tanto para la recepción de la petición como para la respuesta de la misma. Este serializador, está creado desde el módulo de DRF (Django Rest Framework), que nos simplifica la generación de estos objetos. En este caso se está haciendo uso del serializador *HyperlinkedModelSerializer*. A continuación podemos ver el serializador descripto.

```
class ImageSerializer(serializers.HyperlinkedModelSerializer):
    # picture = Base64ImageField()
    class Meta:
        model = Picture
        fields = ('picture', 'classification_result')
```

Figura 6.0 Serializador de datos

Resta ver el *endpoint* que se brinda para hacer la clasificación de la imagen en cuestión. Para esto utilizamos una herramienta que nos brinda DRF, los *viewsets*, estos objetos nos brindan la generación de URL's, normalmente denominados *endpoints* en el desarrollo web, para cada método que requiere

nuestro objeto; generalmente es un ABM (Alta, Baja y Modificación), pero nos permite, además, una forma de generar *endpoints* específicos. Este es el caso del *endpoint* que se generó para la clasificación, el cual es "http://{mi-dominio}/pictures/upload_docs/". Ya que las pruebas se las hace en un servidor local, la URL sería "http://localhost:8000/pictures/ upload_docs".

```
@method_decorator(csrf_exempt, name='dispatch')
class PictureViewSet(viewsets.ModelViewSet):
    queryset = Picture.objects.all()
    serializer_class = ImageSerializer

    @action(detail=False, methods=['post', 'put'])
    def upload_docs(self, request):
```

Figura 6.1 Definición de un *endpoint* particular usando viewsets

En la figura anterior, podemos ver el encabezado de nuestra vista, en la misma se declara el conjunto de datos con el que se va a trabajar y el serializador que se va a utilizar para convertir las peticiones a objetos y visceversa. Por último, podemos ver cómo con el decorador @action(), logramos definir un *endpoint* personalizado, especificando cuales son los métodos permitidos para las peticiones.

Para poder instalar y saber cómo realizar el testing de la aplicación Django, podemos utilizar la guía brindada en el "Apéndice A", en la sección relativa a "Django". Con esto concluimos con la revisión de la aplicación de *backend*. Para finalizar con las aplicaciones que dan soporte a nuestro modelo de procesamiento, continuaremos viendo el consumidor de la *API*, el *frontend*. Esta aplicación es la que permite al usuario poder utilizar lo previamente visto, es la aplicación móvil que nos deja tomar la foto y visualizar el resultado.

3.3 La aplicación Ionic, el *frontend*

La aplicación *frontend*, es una aplicación sencilla, desarrollada con el *framework* Ionic versión 3. Esta aplicación, nos permite tomar una foto, enviarla a nuestro *backend* de procesamiento y luego de que el *backend* responde, ver el resultado de la clasificación.

Como vimos previamente, en la sección "2.3.4 Ionic", Ionic es un *framework* que nos permite crear aplicaciones híbridas para dispositivos móviles aprovechando que todos los dispositivos son capaces de ejecutar aplicaciones web y sus lenguajes. En este caso, se desarrolló una aplicación muy sencilla que nos brinda la posibilidad de sacar imágenes con la cámara del dispositivo, enviarlas a nuestra aplicación de procesamiento mediante la *API* vista anteriormente y poder mostrar el resultado que se obtuvo. La aplicación que se desarrolló es sencilla ya que el objetivo de la misma no es hacer una versión comercial que deba tener mayor

cantidad de validaciones y que brinde un óptimo funcionamiento, más bien era poder dar una demostración del fundamento de la tesina, que es hacer una comparación entre técnicas de clasificación de imágenes para la problemática propuesta.

La aplicación consta de 2 páginas, la primera, es el punto de inicio de la aplicación que nos brinda un botón para poder sacar la imagen para que podamos enviarla a clasificar.

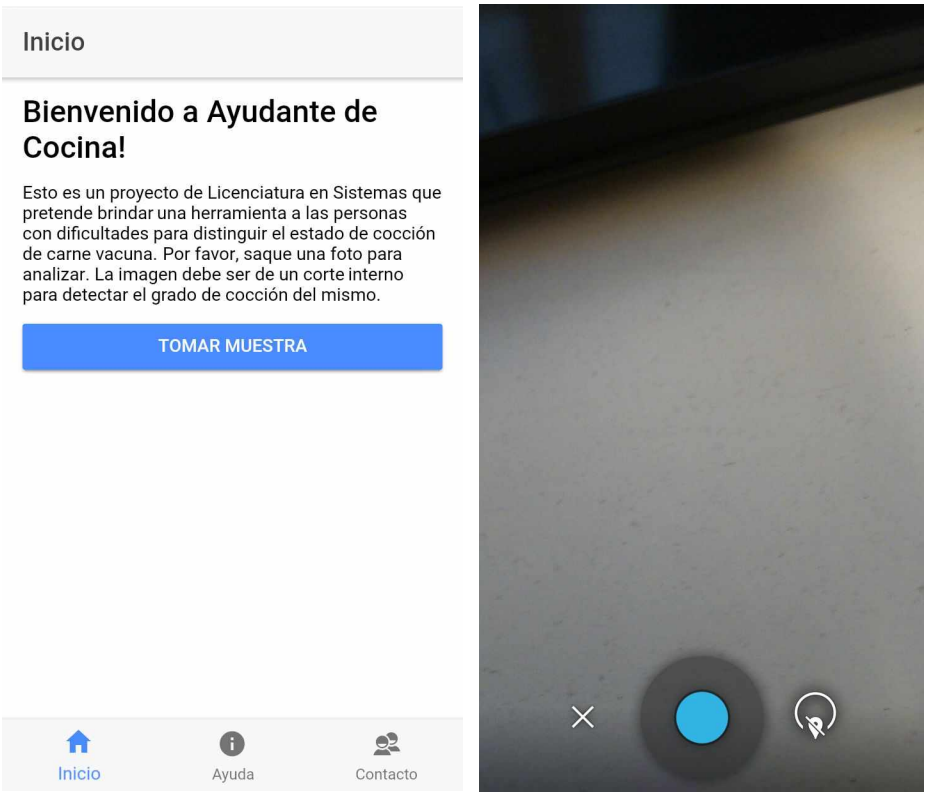

Figura 6.2 Inicio de la aplicación móvil y muestra de la funcionalidad de la cámara

En la figura anterior, vemos lo previamente mencionado, una vez que hayamos tomado la fotografía del corte de carne vacuno a analizar, nos aparecerá un tilde para enviar. Una vez enviada la imágen la aplicación muestra un diálogo de que se está procesando la imagen, luego de eso nos muestra un cartel de que se subió correctamente el archivo o que falló algo mientras se procesaba y por último nos muestra el resultado con la imagen tomada y algunos datos de la imagen.

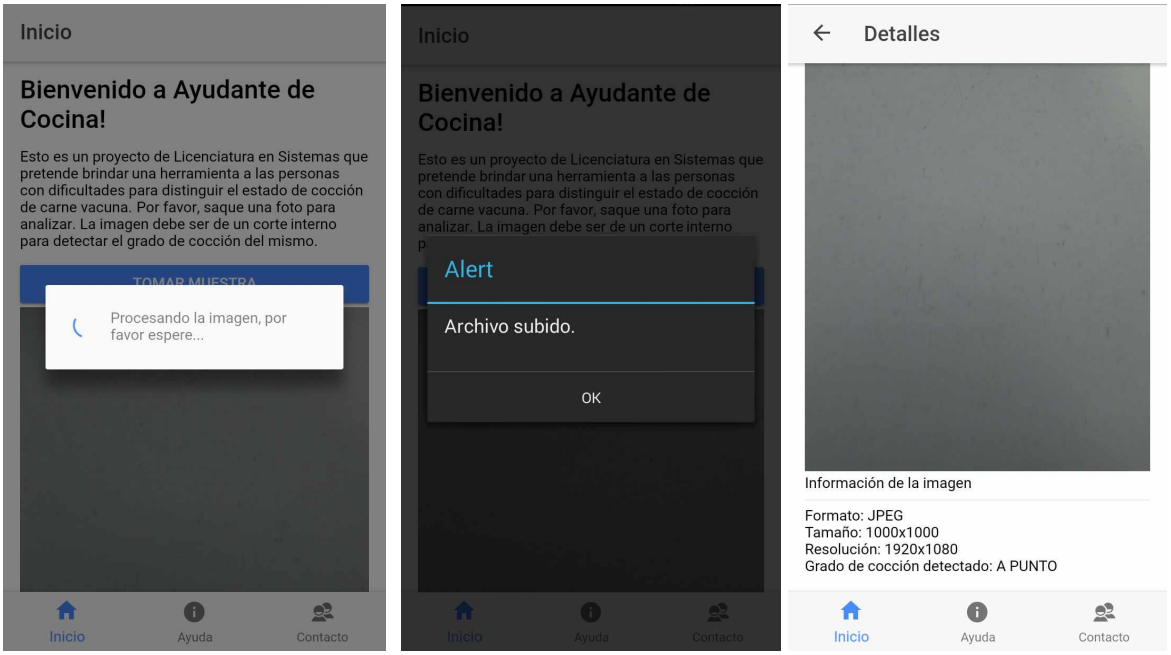

Figura 6.3 Muestra de los diálogos y pantallas que muestra la aplicación

Siguiendo con la revisión, en la vista de "Ayuda", hay una breve guía de cómo utilizar la aplicación, que explica sintéticamente el comportamiento de la misma.

En el "Apéndice A", disponemos de una sección con los pasos a seguir para instalar la aplicación en un teléfono celular con sistema operativo Android y el procedimiento para probar la misma.

Con lo visto hasta el momento, tenemos una noción de la arquitectura general que se desarrolló, habiendo revisado cada una de las partes que dan soporte a nuestro modelo de clasificación. Lo que resta analizar es nuestro modelo de procesamiento en si, el encargado de distinguir si una imagen corresponde a una categoría u otra. Por esto, en los apartados venideros, se procederá a desarrollar toda la investigación relativa al tópico de clasificación de imágenes.

3.4 El modelo de clasificación

En este apartado, se describirán todos los procedimientos, pruebas y conclusiones que se llevaron a cabo y obtuvieron al realizar las pruebas con las distintas técnicas de *Machine Learning* y *Deep Learning*. Además, se hará una comparativa entre los distintos acercamientos para poder distinguir ventajas y desventajas de cada una de las mismas y poder determinar cuál se adecúa mejor a la problemática propuesta.

Por otro lado, para tener una noción de los resultados obtenidos en cada uno de los entrenamientos y post evaluación del conjunto de datos de validación, se utilizaron métricas para problemas de regresión; las métricas utilizadas fueron:

- Error Absoluto Medio (*Mean Absolute Error*)
- Error Cuadrático Medio (*Mean Square Error*)

Estas métricas fueron utilizadas en todos los experimentos realizados en esta investigación para poder comparar el desempeño de cada modelo entrenado.

3.4.1 *Support Vector Regression*

En este apartado veremos el desempeño del modelo *SVR*, con los distintos descriptores seleccionados. Comenzaremos por analizar aquellos algoritmos que nos permiten obtener características texturales en las imágenes, particularmente se busca analizar cómo se comporta el modelo *SVR* con los descriptores *LBP* y Haralick. Luego analizaremos el comportamiento del modelo con el descriptor *Histogram of Oriented Gradients.* Vale destacar que, en cada una de las pruebas realizadas con el modelo *SVR* se hizo uso de 2 kernels distintos, uno lineal y el otro conocido como *RBF* (*Radial Basis Function*). Además, para cada uno de los descriptores seleccionados, se realizaron entrenamientos de los modelos *SVR* con distintos parámetros en los descriptores para ver su funcionamiento mediante la utilización de la técnica de *fine tuning*, siempre y cuando el descriptor lo permitiera.

Los valores de las métricas presentadas en este apartado, son todos calculados post entrenamiento del *SVR* sobre los datos obtenidos del conjunto de validación descrito en 3.1 El Dataset.

3.4.1.1 *Local Binary Pattern (LBP)*

Luego de ver el funcionamiento del algoritmo *LBP*, en el capítulo 2 sección 2.2.2.1.1, veremos cómo se comportó nuestro modelo *SVR* con las características obtenidas por este descriptor. Los valores utilizados para el *fine tuning* en *LBP*, fueron los siguientes:

- Radio: el radio de la circunferencia utilizada por el algoritmo *LBP*.
- Puntos: la cantidad de puntos que conforman el contorno de la circunferencia.

En la tabla siguiente, podemos observar las métricas obtenidas post entrenamiento y validación con los respectivos conjuntos de datos.

Tabla 1 - *SVR* - *LBP*
Métricas obtenidas en las distintas pruebas realizadas con el modelo *SVR*

utilizando el descriptor *LBP* como generador de características.			
	Fine tuning	***MAE***	***MSE***
Kernel *RBF*	- Puntos: 64 - Radio: 5	1.210744859 (± 0.741)	1.991906195 (± 1.963)
	- Puntos: 24 - Radio: 8	**1.199963604 (± 0.706)**	**1.96946065 (± 1.802)**
	- Puntos: 24 - Radio: 3	1.195237832 (± 0.753)	1.960906769 (± 2.024)
	- Puntos: 64 - Radio: 3	1.209648875 (± 0.765)	1.991297787 (± 2.061)
Kernel Lineal	- Puntos: 64 - Radio: 5	**1.150305714 (± 0.712)**	**1.830457383 (± 1.717)**
	- Puntos: 24 - Radio: 8	1.151534776 (± 0.696)	1.811149639 (± 1.762)
	- Puntos: 24 - Radio: 3	1.150831381 (± 0.717)	1.838638056 (± 1.612)
	- Puntos: 64 - Radio: 3	1.154586058 (± 0.721)	1.854217723 (± 1.774)

Para tener una comparativa en cada una de las pruebas realizadas, se brinda, a continuación, un gráfico de barras con cada una de las métricas calculadas por cada entrenamiento realizado con sus respectivas desviaciones estándar.

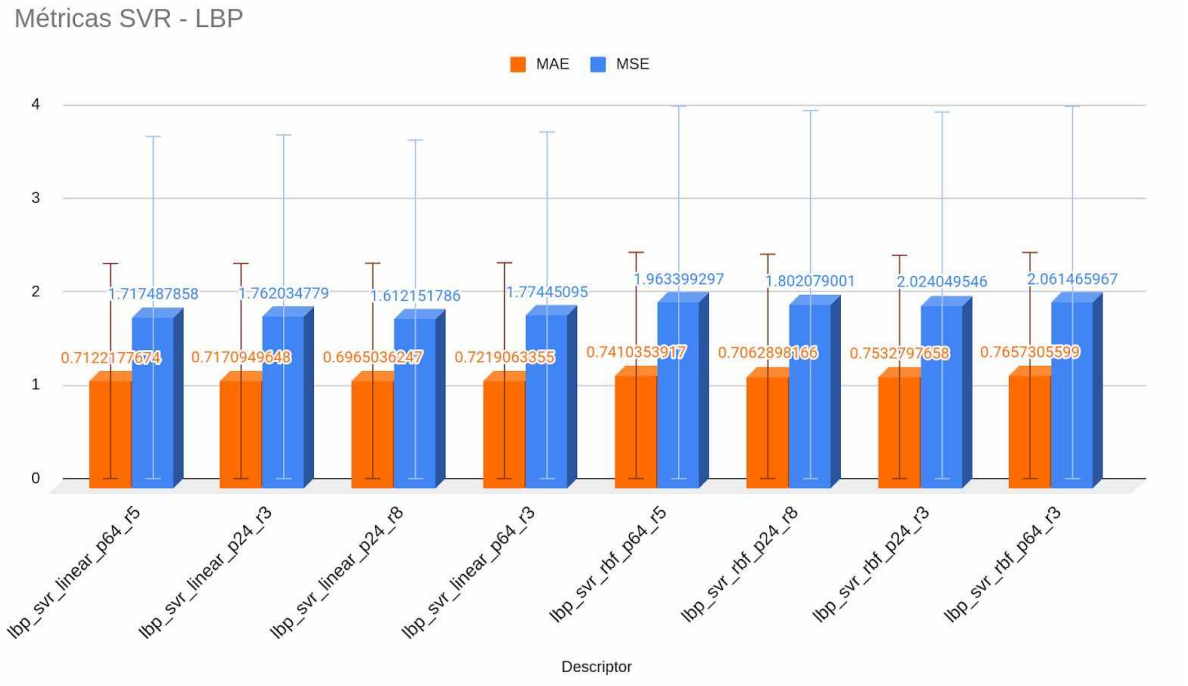

Figura 6.4 Métricas obtenidas del modelo *SVR* con el descriptor *LBP* aplicando la técnica de *fine tuning*

Podemos observar de la tabla de resultados y del gráfico brindado, que las filas resaltadas fueron las que obtuvieron el "mejor" conjunto de valores en las métricas. Esta evaluación se trata del *SVR* con *kernel* lineal y con el descriptor *LBP* de 24 puntos y radio 8 y del modelo *SVR* con kernel *RBF* y con el descriptor *LBP* con parámetros de 64 puntos y radio 5. La elección de los valores durante el *fine tuning*, son producto de varias iteraciones de pruebas que se realizaron y se decidió plasmar estos para dar una noción de cómo iban variando los resultados con la modificación de los mismos.

Ahora bien, el lector se podría preguntar, ¿por qué son los "mejores" resultados obtenidos? Según los datos provistos, se puede ver que estas ejecuciones son las que obtuvieron los menores valores en las métricas *MAE* y *MSE*. Analicemos el caso del modelo *SVR* con *kernel RBF* para entender mejor los valores obtenidos. El valor de *MAE* obtenido para este entrenamiento, fue de 1,19, esto significa que, en promedio, el error entre el valor esperado y el obtenido es de 1,19 por lo que si a nuestra imagen de entrada se le había asignado un valor esperado de 2 (a punto, dentro de nuestra categorización), el modelo, en promedio, tendría un error de a lo sumo +/- 1,19, por lo que el valor resultante, en general, estaría entre 0,81 y 3,19, lo que no representa un escenario óptimo pero a lo sumo difiere entre las clases linderas a la esperada. Además, vale notar que la desviación estándar de *MAE*, tiene un valor de 0.70, esto quiere decir que si tenemos una distribución normal, los resultados de la clasificación van a tender, en su mayoría, a ubicarse a una distancia de 0,70 con respecto a la media de la distribución, por lo que denota una dispersión pequeña para la problemática planteada, ya que en su mayoría, los resultados van a estar ubicados a una distancia de 0,70 de la media poblacional, por lo que habría clases que no se llegarían a identificar del todo bien. Este mismo análisis aplica para la versión del *kernel* lineal, que obtuvo mejores resultados en comparación al *kernel RBF*, ya que como podemos observar en la tabla, la σ de este entrenamiento fue un poco mayor a la versión *RBF* y además, se logró reducir los valores de las métricas de *MAE* y *MSE* a 1,15 y 1,83 respectivamente. Esto nos da la pauta de que la versión del modelo *SVR* funciona mejor con un *kernel* lineal que con uno *RBF*, según los experimentos realizados.

Para clarificar un poco más estos valores analizados, podemos hacer uso de algunos de los histogramas obtenidos con las características del descriptor en cuestión. A continuación, podemos observar algunos de los histogramas utilizados durante el entrenamiento del modelo *SVR*:

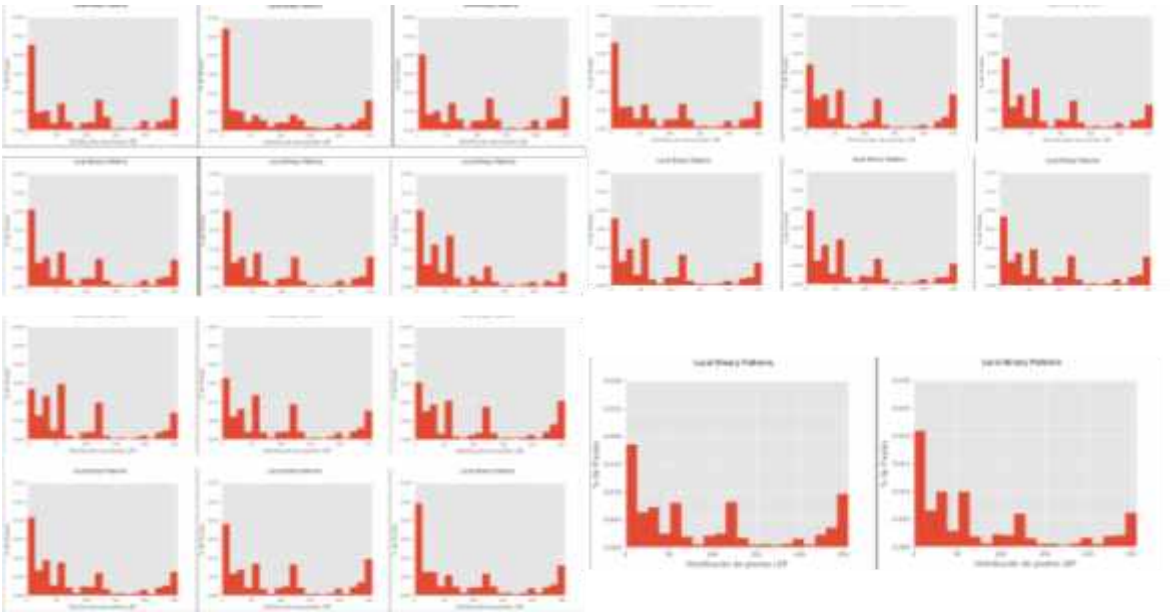

Figura 6.5 Histogramas del set de datos utilizado para entrenar el modelo

Como podemos observar, los histogramas de las imágenes analizadas son muy similares, por lo que contribuye a la idea de que el clasificador no logrará realizar las clasificaciones correctamente, debido a la falta de variabilidad en los histogramas vistos.

Como conclusión de los resultados vistos con este descriptor, podemos decir que no se haría una clasificación correcta utilizando este descriptor con un modelo *SVR* con este conjunto de datos de entrenamiento y de validación en estas circunstancias. Vale notar, que, los valores obtenidos para *MAE* y *MSE*, no son del todo desalentadores, ya que un error de aproximadamente 1, no es malo, pero sí habría que mejorar los valores obtenidos en la desviación estándar, para lograr que el modelo sea capaz de clasificar los 5 tipos de categorías por igual.

3.4.1.2 Haralick

Ahora veamos cómo se comporta el algoritmo Haralick con la detección de características para la problemática planteada en este trabajo. Para la investigación del algoritmo Haralick, se utilizó la implementación del descriptor brindada por la librería mahotas.

Como se expresó al inicio del apartado, este descriptor también fue utilizado con un modelo *SVR* con 2 *kernels* distintos, uno lineal y el otro *RBF*. Se intentó aplicar la técnica de *fine tuning*, como se llevó a cabo con el descriptor *LBP*, pero luego de realizar un análisis de la librería, no se encontró ningún parámetro que nos permitiera aplicar está técnica. A continuación podemos ver algunos histogramas utilizados para los entrenamientos del modelo *SVR,* con el descriptor Haralick:

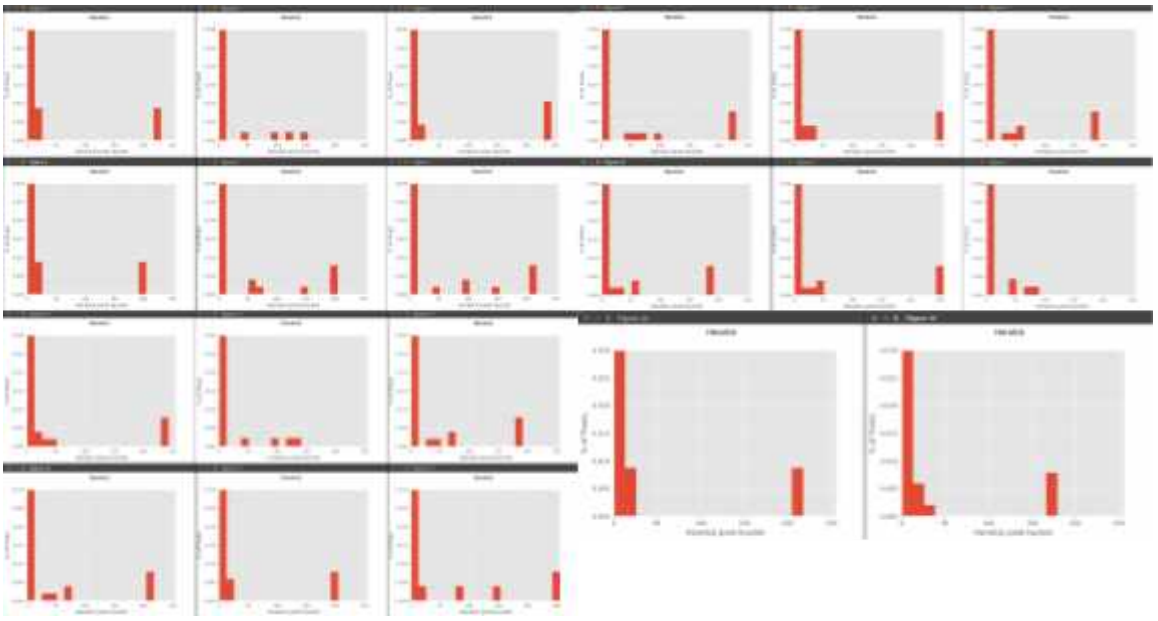

Figura 6.6 Histogramas obtenidos del descriptor Haralick

Vemos en los histogramas calculados, que no se detectan grandes diferencias entre los distintos estados de cocción, por lo que podríamos pensar que el modelo no va a ser capaz de distinguir correctamente las categorías planteadas. Para evaluar mejor el desempeño del *SVR* con el descriptor Haralick, analicemos los valores obtenidos en las métricas sobre el conjunto de validación. En la tabla siguiente podemos apreciar los valores obtenidos en la validación de los modelos *SVR*:

Tabla 2 - *SVR - Haralick*		
Métricas obtenidas en las distintas pruebas realizadas con el modelo *SVR* utilizando el descriptor Haralick como generador de características.		
	MAE	***MSE***
Kernel Lineal	5.248228925 (± 8.017)	50.22492341 (± 87.817)
Kernel *RBF*	**1.33525974** (± **0.711**)	**2.557380557** (± 3.078)

Figura x.x Métricas obtenidas de la validación del modelo SVR con el descriptor Haralick

Al igual que con el descriptor *LBP*, se brinda un gráfico de barras de los valores expuestos en la tabla anterior:

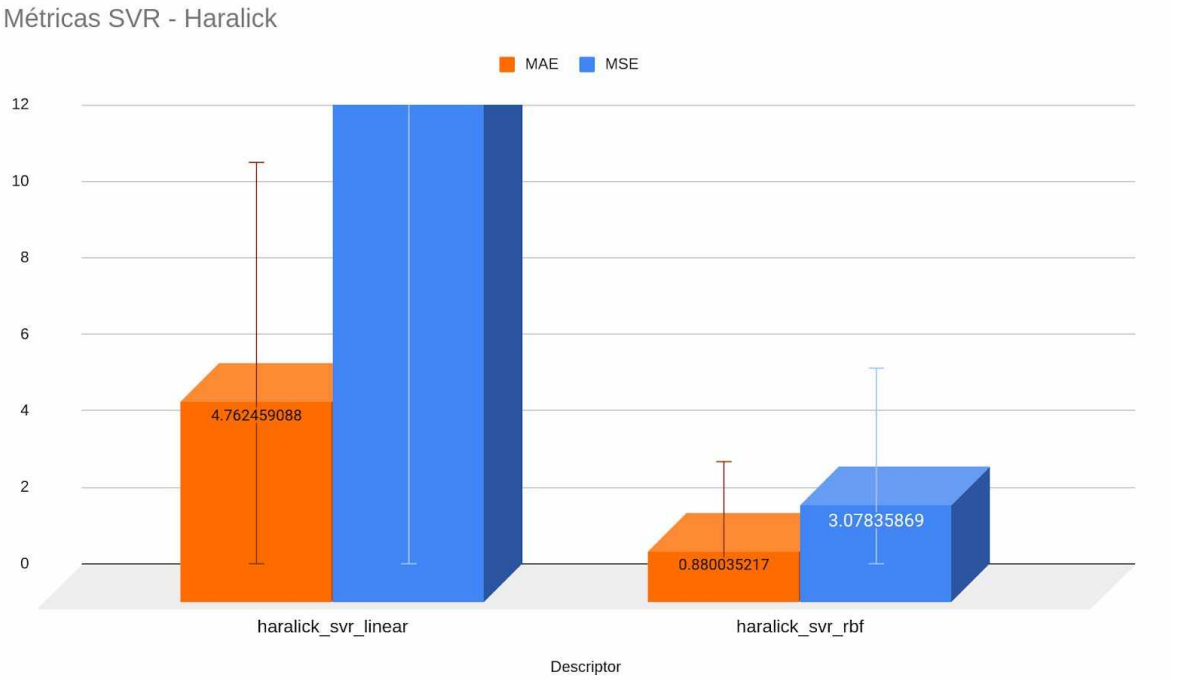

Figura 6.7 Métricas obtenidas del modelo *SVR* con el descriptor Haralick

Vale destacar que en la gráfica del entrenamiento se aplicó una cota superior al eje *y* para poder tener una mejor apreciación de los valores, ya que de otro modo se verían muy pequeños todos, excepto el *MSE* y la σ del mismo del modelo *SVR* con *kernel* lineal que tenía un valor muy grande en comparación a los demás resultados.

De los valores expuestos, podemos dilucidar que el modelo con *kernel* lineal no se comporta para nada bien, es muy errático ya que tiene un *MAE* de 5.24 y un *MSE* de 50.22, con una desviación estándar de 4,76 para *MAE* y 87.81 para *MSE*. Si bien el valor obtenido en la desviación estándar nos permitiría clasificar todas las clases disponibles, el error obtenido es demasiado como para poder considerar este modelo. Por este motivo se lo descarta totalmente al modelo *SVR* con *kernel* lineal para el descriptor Haralick.

Por otro lado, tenemos las métricas del modelo con *kernel RBF* que no son tan desalentadoras como las del *kernel* lineal. Con el *kernel RBF*, las métricas son más similares a las obtenidas en los modelos entrenados con el descriptor *LBP*. Obtuvimos un *MAE* de 1,33 y un *MSE* de 2,55, que son bastante similares a los obtenidos con el descriptor *LBP* con *kernel RBF*, 1,15 y 1,83 respectivamente. A su vez, los valores obtenidos para la desviación estándar de *MAE* y *MSE* son también bastante similares a los del *LBP*. Las desviaciones obtenidas fueron 0.880 para *MAE* y 3.078 para *MSE* con el descriptor Haralick y 0.712 y 1.717 para *LBP* respectivamente. Por lo tanto, todo indicaría que el modelo con Haralick se comportaría de manera muy similar al modelo con *LBP*.

Estos últimos datos sumado a los histogramas mostrados previamente, en la figura 8.2, nos llevaría a pensar que las clasificaciones entre distintas imágenes de diversas clases iban a ser muy parecidas.

3.4.1.3 *Histogram of Oriented Gradients* (*HOG*)

Ahora, analizaremos cómo se comporta el modelo *SVR* con las características obtenidas por el descriptor *HOG* con la categorización de estados de cocción de carne vacuna.

Como mencionamos previamente, para poder generar un modelo capaz de categorizar, haremos uso del modelo *SVR*, el cual nos permite realizar una regresión con las distintas clases en base a las características extraídas. Una vez entrenado nuestro modelo con las características obtenidas, procedemos a generar la evaluación del conjunto de datos de validación, el cual es distinto del conjunto de entrenamiento.

Como con los descriptores anteriormente vistos, se hicieron entrenamientos para el modelo *SVR* con *kernel* lineal y *RBF*, y además, como el descriptor *HOG* permite realizar *fine tuning*, también se aplicó esta técnica para ver la variación en las métricas subyacentes.

Para llevar a cabo el *fine tuning* del descriptor *HOG*, se hizo una prueba a mano de los parámetros modificables del descriptor para seleccionar los más relevantes para ver cómo se comportaba el modelo con las distintas características obtenidas. A continuación se puede ver los parámetros utilizados durante el proceso de selección de los parámetros:

- *orientations*
- *pixels per cell*
- *cells per block*
- *transform_sqrt*
- *block_norm*

A medida que se fueron probando diversas combinaciones con estos parámetros, se fue notando cierta variación cuando se modificaban algunos de ellos más que otros, por lo que se decidió utilizar aquellos parámetros que lograban mayor variación en los resultados calculados. El parámetro seleccionado fue *pixels per cell*, dejando el resto de los parámetros fijos en su valor por defecto.

En la siguiente tabla podemos ver las métricas obtenidas de los distintos entrenamientos previamente mencionados:

Tabla 3 - *SVR - HOG*

Métricas obtenidas en las distintas pruebas realizadas con el modelo *SVR* utilizando el descriptor *HOG* como generador de características.			
	***Fine tuning* (Puntos Por Pixel)**	**MAE**	**MSE**
***Kernel* Lineal**	PPP: 16x16	1.374404126 (± 0.919)	2.734124269 (± 3.023)
	PPP: 8x8	**1.145060349 (± 0.726)**	**1.838883835 (± 2.019)**
	PPP: 5x5	1.140788004 (± 0.682)	1.767495576 (± 1.744)
	PPP: 3x3	1.175023838 (± 0.696)	1.865417908 (± 1.649)
Kernel RBF	**PPP: 16x16**	**1.205932601 (± 0.705)**	**1.985026491 (± 1.774)**
	PPP: 8x8	1.209467286 (± 0.680)	1.99029577 (± 1.664)
	PPP: 5x5	1.213429226 (± 0.682)	1.998829906 (± 1.583)
	PPP: 3x3	1.217393581 (± 0.717)	2.008046026 (± 1.619)

Como en las tablas presentadas anteriormente, se resaltan las entradas con mejores resultados para cada uno de los *kernels* utilizados. En el caso del modelo con *kernel* lineal, vemos que de las evaluaciones realizadas, la que mejor se comportó en base a las métricas resultantes de la evaluación del conjunto de validación, fue la que tenía como parámetro de PPP (Puntos Por Pixel) una matriz de 8x8. La métrica *MAE* obtenida con esta configuración es el segundo más bajo, pero se considera que es mejor que la matriz de 5x5 porque las desviaciones que muestra la configuración de 8x8 es mejor que la obtenida en el otro modelo, ya que la desviación estándar para *MAE* de PPP = 8x8 es de 0.726 contra 0.682. Haciendo un análisis similar al de los apartados anteriores, se ve que dentro del *kernel* lineal el entrenamiento con PPP de 8x8 es el modelo que mejor se va a comportar ya que la desviación estándar es de las más amplias, permitiéndonos clasificar un mayor rango de valores, sin elevar demasiado nuestro error absoluto medio, por lo que a lo sumo el modelo catalogará una imagen con un error de +/- 1,14.

Por otro lado, tenemos las evaluaciones realizadas con el *kernel RBF*, que muestran una leve desmejora con respecto al *kernel* lineal. Esta asunción se fundamenta del hecho que las métricas de los experimentos realizados, en general todos arrojaron valores mayores que los obtenidos con los entrenamientos del *kernel* lineal. A pesar de estas mínimas variaciones en las métricas, podemos observar en el gráfico siguiente, que casi todos los experimentos retornan modelos *SVR* que se comportan de manera muy similar.

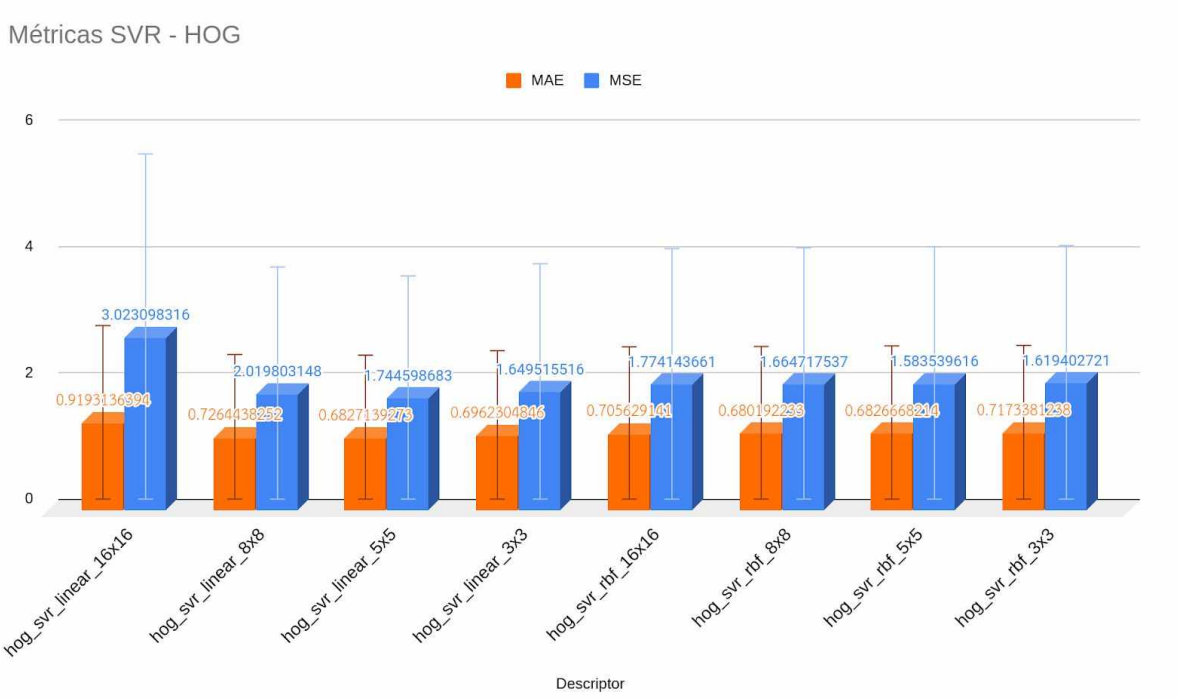

Figura 6.8 Métricas obtenidas del modelo *SVR* con el descriptor *HOG* aplicando la técnica de *fine tuning*

Para comprender los datos de entrenamiento que se le brindaron al modelo, podemos visualizar las características obtenidas de las imágenes brindadas. A continuación, se pueden apreciar algunos de los gradientes por cada una de las clases que se le brindaron al modelo en la etapa de entrenamiento.

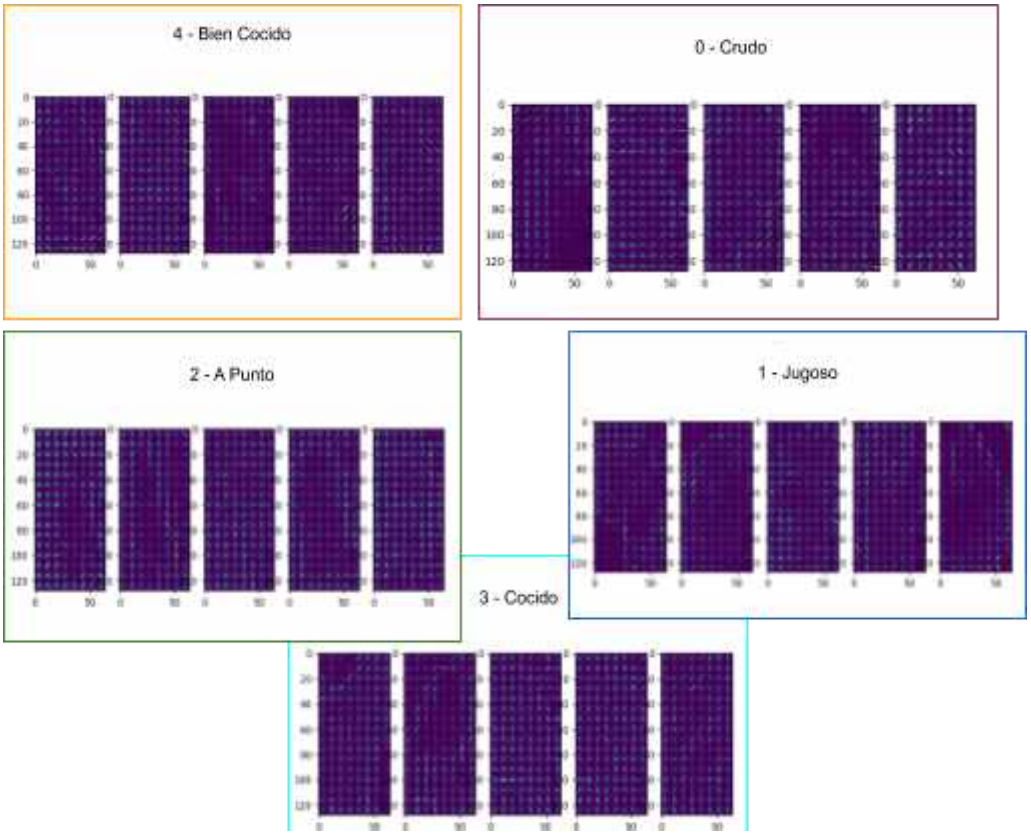

Figura 6.9 Visualización de los gradientes calculados por cada categoría

Como podemos notar, no hay grandes variaciones entre los gradientes obtenidos en las distintas categorías. Quizás, las 2 clases que se nota que los gradientes tienen más peso (por el color que se denota en cada una de las figuras) y por lo tanto se podrían distinguir más del resto de las clases, son la categoría "A Punto" y "Cocido", pero aún así no se ven grandes diferencias, por lo que puede llevar a convalidar que el modelo no tenga una desviación tan grande como para distinguir características distintivas de cada una de las clases.

A continuación, seguiremos con la última técnica que analizaremos para la problemática de clasificar estados de cocción en cortes de carne vacuna, esta técnica se la conoce como *Deep Learning*.

3.4.2 *Deep Learning*

En esta sección se describirán los pasos llevados a cabo durante las experimentaciones con modelos de *Deep Learning*. Se hará una breve descripción de los modelos utilizados, los resultados obtenidos y las conclusiones que se lograron recabar durante el desarrollo del mismo. Como se explicó en capítulos anteriores, el funcionamiento de los algoritmos de *Deep Learning* es bastante diferente a los algoritmos previamente utilizados, donde primero se obtenían las características de relevancia y luego se utilizaban estos tensores de características para entrenar un modelo de *Machine Learning* capaz de evaluar, posteriormente, las imágenes de validación que se le fueran brindando. Los modelos de *Deep Learning*, por el contrario, son capaces de ir aprendiendo características de nuestro conjunto de datos automáticamente, sin intervención del usuario.

Para todas las pruebas realizadas con modelos de *Deep Learning* se siguió siempre el mismo procedimiento de entrenamiento y validación. Para cada uno de los 3 modelos elegidos, se realizaron diversos entrenamientos en cuanto a la cantidad de épocas de entrenamiento y distintos valores en los parámetros de aprendizaje, pero siempre manteniendo el mismo conjuntos de datos de entrenamiento y de validación. Los 3 modelos utilizados durante la investigación son:

- Modelo MobileNet
- Modelo simple (desarrollado a medida)
- Modelo "*Two Lanes*" (utilizado en [40])

Los diversos parámetros de entrenamientos los podemos ver plasmados en la siguiente tabla:

Tabla 4 - Parámetros utilizados en los entrenamientos			
	Optimizador *SGD* - Tasa de aprendizaje	**Épocas**	**Métricas**
Modelo Simple	- 0,1 - 0,01 - 0,001	- 500 - 300	- *MAE* (±) - *MSE* (±)
Modelo MobileNet	- 0,1 - 0,01 - 0,001	- 300 (Con *Transfer Learning*) - 100 - 50	- *MAE* (±) - *MSE* (±)
Modelo *Two Lanes*	- 0,01 - 0,001	- 300	- *MAE* (±) - *MSE* (±)

En todos los entrenamientos realizados se utilizó la técnica de aumentación de datos para lograr un mayor volúmen de información en nuestro conjunto de entrenamiento. Los parámetros utilizados en el aumentador de datos fueron los siguientes:

- Reescalar = 1. / 255
- Rango de recorte = 0.2
- Rango de enfoque = 0.3
- Rango de brillo = [0.2, 1.0]
- Rotación horizontal = True

Vale aclarar que no se utilizó esta técnica con el conjunto de validación, ya que se busca validar con imágenes reales y no alteraciones de las mismas.

Las primeras pruebas que se realizaron con algoritmos de este tipo se basaron en investigaciones previas sobre clasificación de alimentos, como [21] que analizaba qué comida era, ya sea sushi, sopa, etc. En este estudio se utilizaba una red convolucional pre-entrenada conocida como InceptionV3, desarrollada por Google; en este caso haremos uso de la red MobileNet, también desarrollada por Google, con y sin *Transfer Learning* para ver el comportamiento de una red compleja con la temática. Se decidió utilizar la red MobileNet ya que es una red más liviana que la InceptionV3 que ha tenido gran éxito en las últimas competencias de *Deep Learning*. Esta red tiene un rendimiento muy similar a la InceptionV3, pero al ser más liviana tiene un menor costo de procesamiento [51]. A continuación podemos ver la arquitectura de la red MobileNet:

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figura 7.0 Arquitectura de la red convolucional MobileNet

Todos los entrenamientos se realizaron en una *laptop* con procesador Intel I7 4700MQ y 16GB de memoria RAM.

El segundo modelo utilizado (que lo denominaremos como modelo *Two Lanes*), se utilizó en otra investigación [40] relativa a la temática de detección de colores en imágenes. Este modelo divide el procesamiento en 2 columnas que luego se concatenan los resultados calculados de ambas columnas para unificar los conocimientos adquiridos. A continuación, podemos ver una representación de la arquitectura planteada.

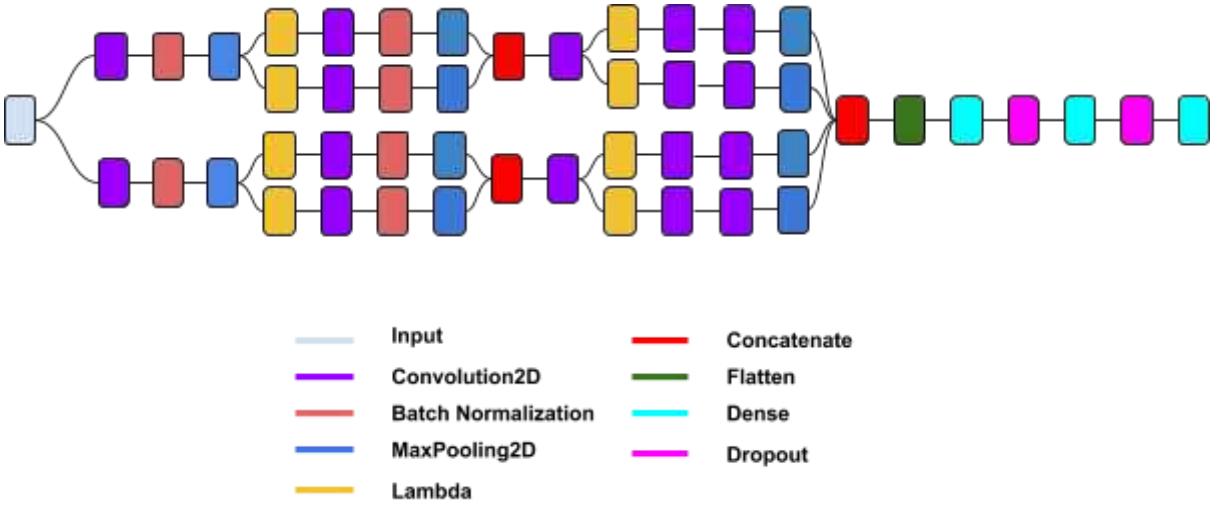

Figura 7.1 Arquitectura del segundo modelo utilizado denominado "*Two Lanes*"

Como podemos observar, el modelo contiene 9 capas diferentes, cada una de ellas tiene su propósito particular que fueron descriptos en el capítulo 2, en la sección "Redes Neuronales".

Este modelo también fue entrenado con la misma configuración de *hardware* del modelo anterior. Utilizando el mismo conjunto de entrenamiento el tiempo estimado de entrenamiento era de unas 6-8 horas, dependiendo de la configuración de cada capa utilizada.

Por último, el tercer modelo utilizado es un modelo simple desarrollado de cero para analizar cómo se comporta una red sencilla y poder contrastar los resultados obtenidos de redes más complejas y más simples. La arquitectura del modelo la podemos ver plasmada en la siguiente figura:

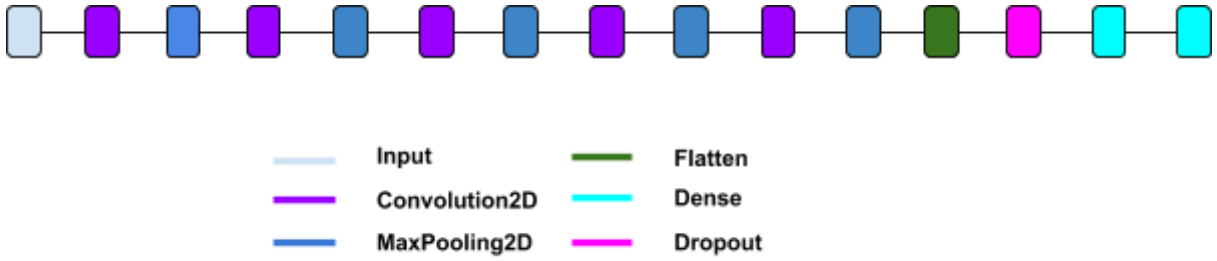

Figura 7.2 Arquitectura del modelo simple

Habiendo descrito las arquitecturas de cada una de las redes utilizadas, veremos los resultados de cada uno de los entrenamientos en cada modelo e iremos analizando los resultados obtenidos en las métricas post validación del modelo con el conjunto de validación visto en la sección 3.1.

En la siguiente tabla, podemos ver un resumen de las métricas obtenidas para cada uno de los entrenamientos, luego iremos analizando las gráficas de los entrenamientos en cada modelo y explicaremos las decisiones que se fueron tomando para llegar a los entrenamientos realizados basados en dichas gráficas y resultados.

Tabla 5 - Métricas obtenidas para los modelos utilizados con *Deep Learning*

Deep Learning	**MAE**		**MSE**	
Modelo Simple	**entrenamiento**	**validación**	**entrenamiento**	**validación**
sgd_default_300_epocas	**0.5213**	**0.6121 (± 1.067)**	**0.4356**	**0.6098 (± 4.160)**

sgd_lr=0,001_300_epocas	0.8018	0.7238 (± 1.040)	1.0294	0.8718 (± 4.033)
sgd_default_500_epocas	0.4385	0.6467 (± 1.046)	0.3056	0.7091 (± 3.924)
Modelo MobileNet				
sgd_default_300_epocas_con_TF	**0.6339**	**0.6422 (± 0.877)**	**0.6296**	**0.6831 (± 2.909)**
sgd_lr=0,001_300_epocas_con_TF	0.4403	0.6890 (± 0.964)	0.3172	0.7389 (± 3.414)
sgd_lr=0,001_50_epocas_sin_TF	0.6153	0.7108 (± 1.159)	0.5833	0.8436 (± 5.110)
sgd_lr=0,001_100_epocas_sin_TF	0.5099	0.6685 (± 1.133)	0.4296	0.7462 (± 4.836)
Modelo Two Lanes				
sgd_lr=0,001_300_epocas	**0.3737**	**0.5313 (± 1.058)**	**0.2368**	**0.5492 (± 4.045)**

A continuación iremos analizando cada uno de los entrenamientos llevados a cabo para cada uno de los modelos enunciados, se irá describiendo las decisiones que se fueron tomando y el por qué de las mismas. Se comenzará analizando los valores obtenidos para el modelo simple, que nos dará una primera noción de cómo se puede comportar un modelo de *Deep Learning* con la problemática de detectar estados de cocción en carne vacuna.

3.4.2.1 Modelo Simple

El primer entrenamiento que se realizó sobre este modelo fue de 300 épocas de entrenamiento con el optimizador *SGD* con sus parámetros por defecto (*learning_rate*=0.01, *momentum*=0.0, *nesterov*=False). A continuación podemos observar las curvas en cada una de las épocas para las métricas *MAE* y *MSE*.

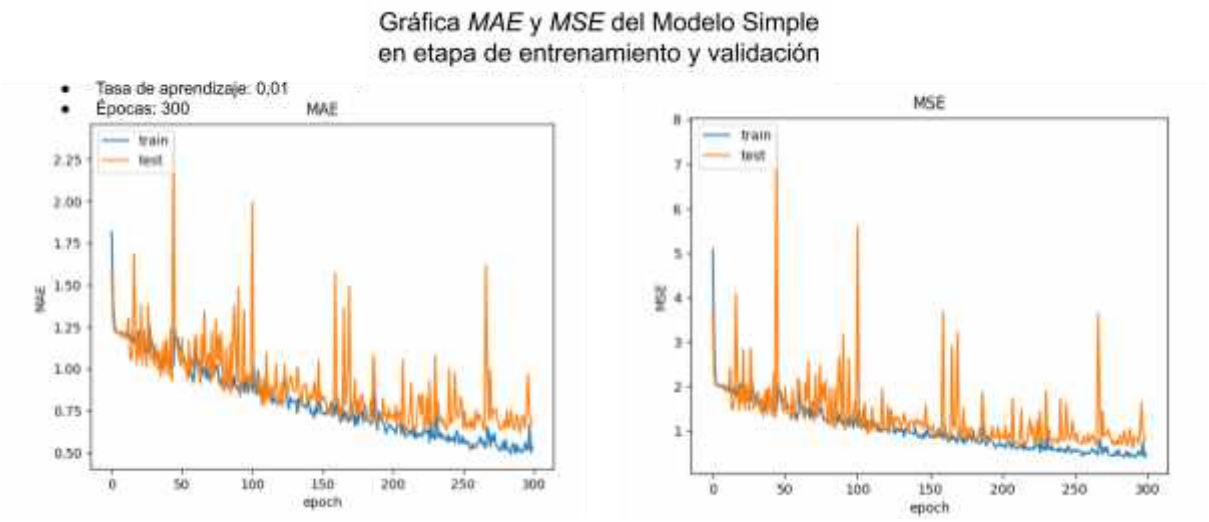

Figura 7.3 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo simple con parámetros por defecto

De la figura anterior podemos notar que el modelo no está padeciendo de *overfitting* hasta la época 250 aproximadamente, donde se comienza a ver que las curvas de errores para ambas etapas comienzan a divergir. Las oscilaciones que se ven en las curvas de validación pueden estar dadas a que el conjunto de datos de validación tenga un número reducido de imágenes, por lo que nos daría la idea de que haciendo un aumento en el volúmen de información lograríamos curvas menos irregulares. Luego de haber visto las gráficas y poder conjeturar que el modelo no padece de *overfitting* hasta la época ≈250, observemos los resultados obtenidos en las demás métricas.

- *MAE* (entrenamiento / validación): 0.5213 / 0.6121 (± 1.067)
- *MSE* (entrenamiento / validación): 0.4356 / 0.6098 (± 4.160)

Los resultados obtenidos, son bastante alentadores, ya que según las métricas obtenidas el modelo no se estaría comportando tan errático. Decimos esto ya que el error absoluto medio que se obtuvo para la etapa de validación fue de 0,61 por lo que nuestro modelo estaría fallando en una medida de +/- 0,61 en promedio a la clase real de la evaluación puntual. Esto, sumado a que la desviación estándar obtenida es de 1.06, supone que el abanico de rangos posibles que el modelo alcanza es bastante amplio, por lo que se podría a llegar a etiquetar de manera correcta un rango mayor de evaluaciones, en comparación a los datos obtenidos con los *SVR*. Para clarificar esta última aseveración sobre la desviación, si tenemos una distribución normal, lo que muestra la desviación en este caso, si nuestros valores van desde 0 a 4 y la media es 2, tener una desviación de 1,06 significa que los valores resultantes de las evaluaciones pueden estar, en promedio entre 0,94 y 3,06, por lo que la amplitud lograda es bastante satisfactoria para el rango de valores que se está buscando evaluar.

La siguiente prueba que se realizó con este modelo, fue modificar la tasa de aprendizaje del optimizador *SGD*, para analizar si se podía corregir las alteraciones en las curvas de *MAE* y *MSE* obtenidas con el primer entrenamiento. Por este motivo se utilizó una tasa de aprendizaje de 0,001, que es la décima parte del parámetro por defecto en el optimizador *SGD*. A este modelo también se lo entrenó durante 300 épocas. Ahora bien, veamos las gráficas obtenidas en las métricas previamente mencionadas.

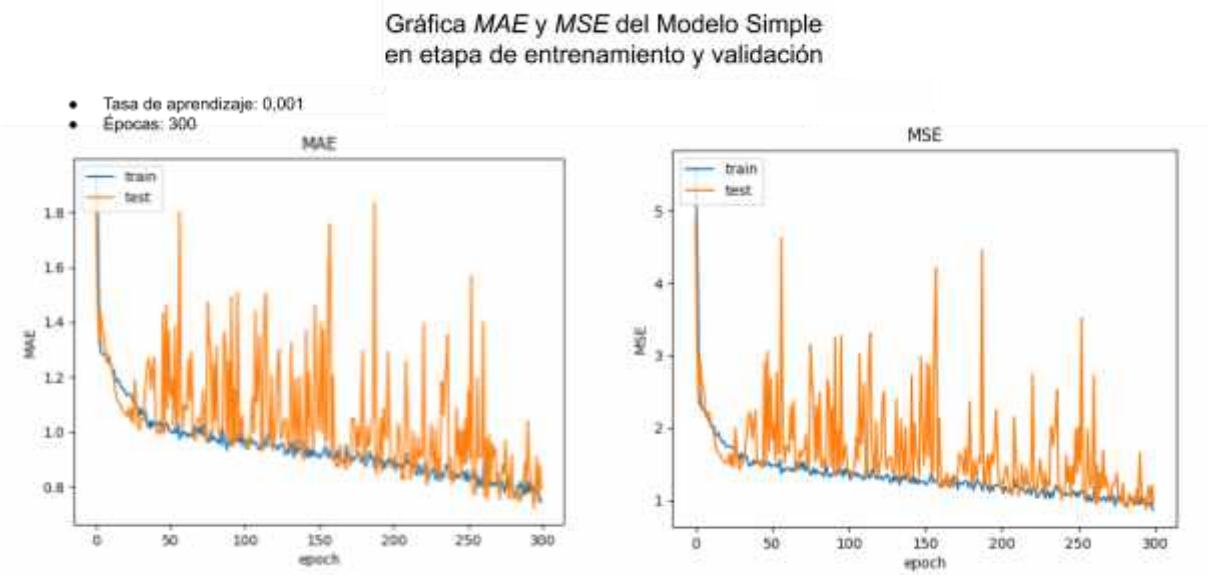

Figura 7.4 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo simple con tasa de aprendizaje de 0,001

Analizando la figura anterior notamos que reduciendo la tasa de aprendizaje, el modelo se comporta peor en cada etapa de validación, haciéndose notorio a lo largo de todas las épocas de entrenamiento. Otro aspecto que podemos resaltar de estos gráficos, es que, al contrario del entrenamiento anterior, no se detectaría un punto de quiebre donde las curvas comiencen a divergir, por lo que esto nos da una noción de que la tasa de aprendizaje más reducida hace que el modelo logre un mejor entrenamiento. Que la curva de validación sea más irregular que la de entrenamiento es debido a que la cantidad de imágenes que contiene dicho conjunto es más reducida, por lo que para lograr que acompañe la evolución de la de entrenamiento se debería aumentar el número de imágenes en el conjunto de validación. Ahora bien, veamos los resultados obtenidos en las métricas:

- *MAE* (entrenamiento / validación): 0.8018 / 0.7238 (± 1.040)
- *MSE* (entrenamiento / validación): 1.0294 / 0.8718 (± 4.033)

En líneas generales notamos que los valores obtenidos en las métricas fueron un poco peor en comparación al entrenamiento anterior. El error medio absoluto es de 0,72 y el cuadrático de 0,87, esto nos da la pauta de que el modelo

va a cometer más errores en comparación al entrenamiento anterior. Notamos que no hubo una gran variación con respecto al entrenamiento anterior en cuanto a la desviación obtenida, por lo que esto es un punto positivo para este entrenamiento, ya que, en promedio, tiene un espectro de etiquetado bastante amplio, permitiendo alcanzar casi todas las clases planteadas.

Luego de este entrenamiento, se hizo una prueba más con este modelo con 300 épocas de entrenamiento, utilizando el optimizador con una tasa de aprendizaje mayor, de 0,1, pero durante el entrenamiento el modelo divergía rápidamente y se dejaba de poder representar las métricas obteniendo valores nan ya que no se podían representar los números obtenidos, por lo que se decidió no incluir ese entrenamiento entre los resultados. Luego de eso, se decidió hacer un entrenamiento de 500 épocas con el optimizador *SGD* con parámetros por defecto para analizar el *overfitting* que detectado previamente, en pos de descartar que haya sido un mínimo local. A continuación, podemos ver las gráficas de las métricas de *MAE* y *MSE* para ver cómo afecta la cantidad de iteraciones en el error obtenido para descartar que sea un mínimo local.

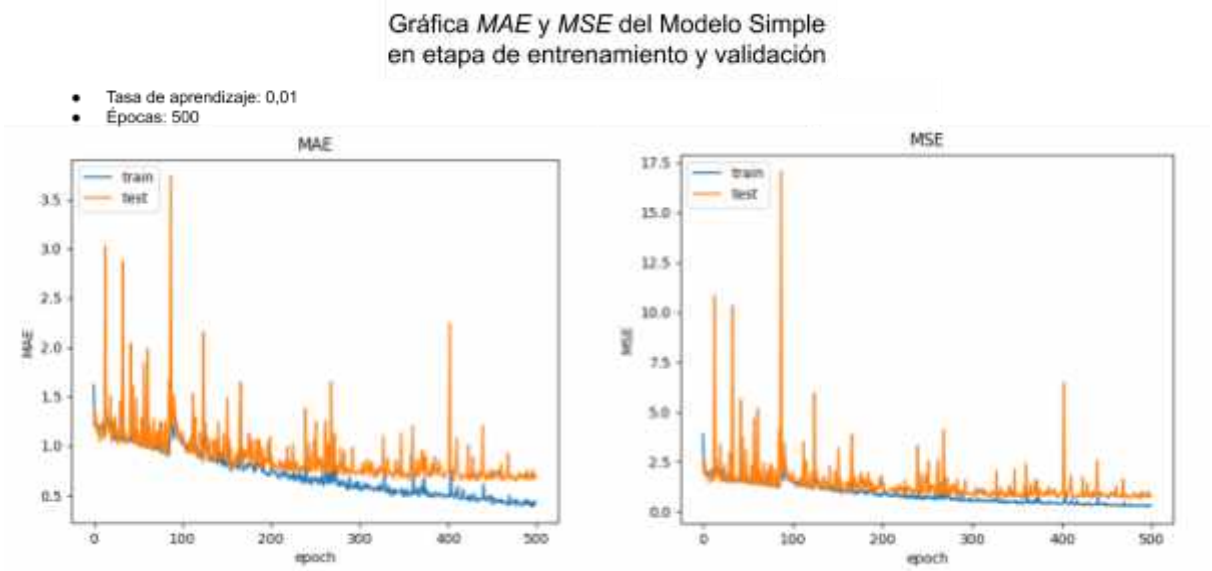

Figura 7.5 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo simple con tasa de aprendizaje de 0,001

Esta gráfica nos reafirma que el modelo comienza a padecer overfitting después de las 280 épocas aproximadamente. Por lo que con el conjunto de datos que se dispone, bastaría con entrenar el modelo aproximadamente durante 300 iteraciones para obtener el punto de mejor rendimiento del mismo. Como en los casos anteriores, veamos el resto de las métricas calculadas para analizar además del fenómeno del overfitting, el comportamiento del modelo con 500 épocas de entrenamiento.

- *MAE* (entrenamiento / validación): 0.4385 / 0.6467 (± 1.046)

- *MSE* (entrenamiento / validación): 0.3056 / 0.7091 (± 3.924)

Como vemos, es evidente el *overfitting* al ver los resultados de *MAE* y *MSE* obtenidos para el entrenamiento y la validación. Vemos que la diferencia entre cada uno de los valores es más del doble en *MSE* y del 50% aproximadamente en *MAE*. Luego, si analizamos la desviación obtenida, el modelo tendría un rango bastante amplio como para clasificar las imágenes brindadas, pero dado el hecho de que no se ven mejoras contra la primera prueba con este modelo, se decide seleccionar como mejor prueba del modelo simple a la primer prueba realizada con 300 épocas de entrenamiento y el optimizador *SGD* con parámetros por defecto.

Con esto concluimos los estudios realizados con el modelo simple para la temática planteada, seguiremos con el análisis del modelo denominado en este trabajo como modelo *Two Lanes*.

3.4.2.2 Modelo *Two Lanes*

Para este modelo se realizaron 2 pruebas de 300 épocas cada una. Este modelo en el trabajo de reconocimiento de colores en el que se lo encontró, tenía ciertos parámetros por defecto que lo que se buscó era ver cómo variaba si modificabamos la tasa de aprendizaje manteniendo el resto de los parámetros iguales. Los parámetros del optimizador *SGD* con los que venía configurado eran:

- Tasa de aprendizaje: 1e-3
- Decaimiento: 1e-6
- Momentum: 0.9
- Momento Nesterov: *True*

Además del entrenamiento con esos parámetros se hizo una prueba más con la tasa de aprendizaje modificada a 0,01. Vale destacar la optimización en los parámetros e inicialización en este modelo, que según [52] aporta una gran mejora en el desempeño de las *DNN* y *RNN*. Comencemos el análisis de este modelo observando los resultados obtenidos con la configuración con la que se utilizó el modelo en la investigación original. A continuación, podemos observar las gráficas del *MAE* y *MSE* obtenidos para esta experimentación:

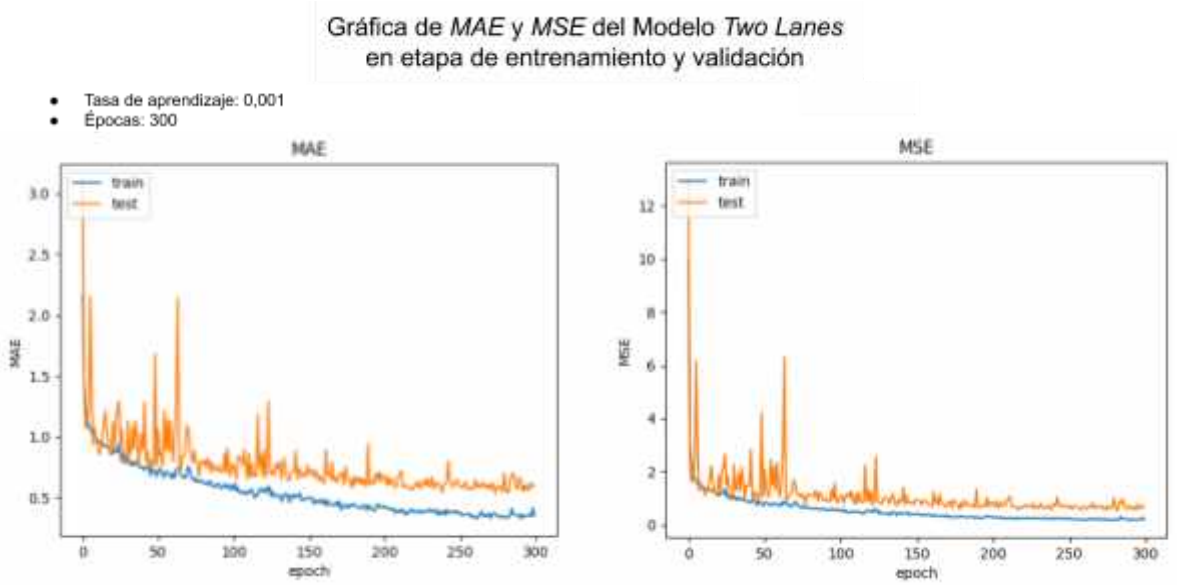

Figura 7.6 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo *Two Lanes* con tasa de aprendizaje de 0,001

Como podemos apreciar de la gráfica anterior, las curvas de error durante el aprendizaje y la validación tienen un comportamiento muy similar. Si bien las gráficas comienzan a divergir a partir de la época 125 aproximadamente, vemos que se sigue manteniendo una distancia pareja para el resto de las épocas venideras, por lo que no se puede afirmar que el modelo esté padeciendo de *overfitting*. Más bien, se podría intuir que la diferencia entre las 2 curvas de error, está dada por la diferencia en el volúmen de información que tiene cada conjunto de datos. Los mejores resultados de *MAE* y *MSE* obtenidos fueron 0.53 0.55 respectivamente, por lo que es un error bastante reducido ya que la categorización actual va a diferir entre +/- 0,53 con respecto al valor real, en promedio. Ahora bien, veamos los resultados obtenidos para las métricas en entrenamiento y validación con su desviación estándar:

- *MAE* (entrenamiento / validación): 0.3737 / 0.5313 (± 1.058)
- *MSE* (entrenamiento / validación): 0.2368 / 0.5492 (± 4.045)

Analizando la desviación de este entrenamiento, vemos que el valor obtenido es de 1,05; como se mencionó en los análisis de los entrenamientos previos, este valor en la desviación nos estaría permitiendo un abanico bastante amplio de posibilidades, ya que, en promedio se podría llegar a todas las clases estipuladas en el trabajo, dado que si nuestra media es 2 en una distribución normal, con este valor en la varianza llegaríamos de 0,88 hasta 3,12.

Habiendo visto este primer entrenamiento, prosigamos con la siguiente prueba que se realizó con este modelo. Como mencionamos previamente, se intentó aumentar la tasa de aprendizaje de 0,001 a 0,01. El entrenamiento del modelo con este parámetro, rápidamente comenzaba a devolver valores *nan* en las

métricas estipuladas, esto se debe a que al intentar optimizar la función con un valor mayor, los valores resultantes crecían demasiado y por lo tanto el sistema dejó de interpretarlo. Por este motivo, se decidió no aumentar el valor y dado que con el primer entrenamiento se obtuvieron resultados bastante alentadores, se decidió dejar ese entrenamiento como el mejor de este modelo.

Con esto finalizamos el análisis del modelo "*Two Lanes*" y ahora continuaremos viendo el último modelo utilizado en los entrenamientos de *Deep Learning*, el modelo *MobileNet* de Google.

3.4.2.3 Modelo *MobileNet*

El modelo *MobileNet* es el más complejo de los modelos de *Deep Learning* utilizados durante esta investigación. Se decidió utilizar un modelo complejo para poder comparar el funcionamiento de un modelo avanzado contra modelos más simples y así poder comparar el desempeño de estos para la temática planteada.

Las primeras pruebas realizadas con este modelo fueron aplicadas con la técnica de *Transfer Learning*, que nos permite tener un modelo pre entrenado con pesos en sus neuronas que en general nos sirven para optimizar el aprendizaje del modelo. Se realizaron 2 entrenamientos con esta técnica, ambos de 300 épocas pero con distintas tasas de aprendizaje en el optimizador *SGD*, uno con 0,001 y el otro con 0,01.

Comencemos viendo el entrenamiento con el optimizador *SGD* con parámetros por defecto. A continuación, podemos observar los resultados obtenidos de este entrenamiento:

- *MAE* (entrenamiento / validación): 0.6339 / 0.6422 (± 0.877)
- *MSE* (entrenamiento / validación): 0.6296 / 0.6831 (± 2.909)

Como podemos notar, el *MAE* y *MSE* obtenidos son muy similares a todos los entrenamientos con los distintos modelos previamente realizados, por lo que son errores bastante alentadores, ya que no diverge mucho del valor real. Habiendo visto los mejores errores, veamos cómo se fueron comportando los mismos a lo largo de las 300 épocas de entrenamiento:

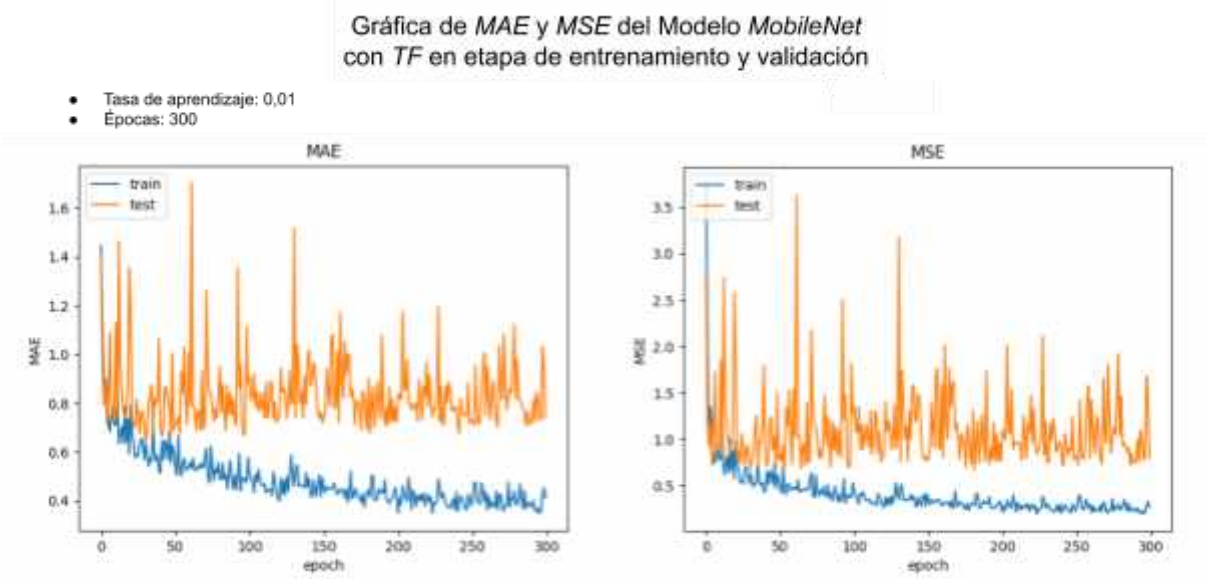

Figura 7.7 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo *MobileNet* con tasa de aprendizaje de 0,01 con *TL*

Como se puede apreciar, en este caso, el mejor valor obtenido en los errores es de una etapa de aprendizaje media que puede ser debido al aprendizaje pre obtenido mediante la técnica de *TL*; sumado a esto, notamos que a partir de la época 50 las curvas de los errores se empiezan a separar por lo que se podría asumir que a medida que más etapas pasan más será la diferencia entre el entrenamiento y la validación, por lo que el modelo estaría padeciendo de *overfitting*. Otra característica que podemos apreciar de las curvas de errores, es que la curva del error de validación es muy heterogénea, dado que tiene muchos saltos abruptos, por lo que se entiende que el conjunto de datos de validación no estaría sirviendo para validar las características aprendidas durante el entrenamiento. Esto puede estar radicado en 2 problemas mayormente; el primero es que el conjunto de validación disponga de poco volúmen de información o que la información que tiene no es representativa de la clase en cuestión. Esta última suposición, también podría ser aplicada para el conjunto de entrenamiento, ya que si el conjunto de entrenamiento no tuviera información suficientemente representativa y el conjunto de validación estuviera bien, esto haría que el sistema se comporte de este mismo modo. Pero dado la evidencia previamente obtenida con los otros modelos, se estima que el problema radica más en el conjunto de validación que en el de entrenamiento.

Continuando con los experimentos, ahora veamos el entrenamiento realizado con el cambio en la tasa de aprendizaje del optimizador *SGD* a 0,001. A continuación podemos ver las gráficas de los errores obtenidos para el entrenamiento y la validación:

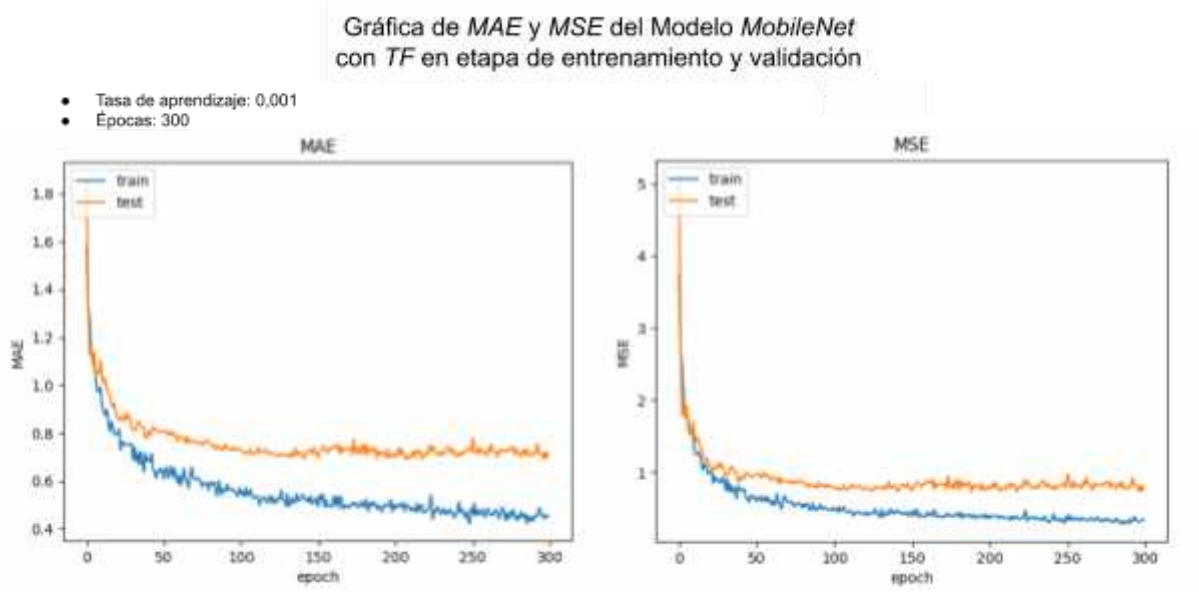

Figura 7.8 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación del modelo *MobileNet* con tasa de aprendizaje de 0,001 con *TL*

Se puede dilucidar de las curvas de entrenamiento, que al reducir la tasa de aprendizaje, el entrenamiento se va realizando mucho más homogéneo, haciendo que no haya tantos picos en las curvas de los errores. Como en el entrenamiento anterior, vemos que a partir de la época 40, aproximadamente, se nota que el modelo comienza a padecer de *overfitting* ya que las curvas de errores se van alejando cada vez más una de otra, donde la curva de entrenamiento se sigue optimizando hacia un error que tiende a 0 y el error de la validación queda oscilando cerca de 0,7/0,8. Ahora bien, veamos los valores obtenidos en las demás métricas, para poder analizar mejor el comportamiento de este experimento:

- *MAE* (entrenamiento / validación): 0.4403 / 0.6890 (± 0.964)
- *MSE* (entrenamiento / validación): 0.3172 / 0.7389 (± 3.414)

Podemos notar, que el aumento en la desviación en este caso, está vinculado a que el modelo se comporta un poco peor que el anterior, ya que al aumentar el error, nuestro modelo aumenta también la desviación, dado que va a poder alcanzar valores un poco más a los extremos de nuestro rango de valores reales. La situación ideal, sería que nuestra desviación aumente y que el error disminuya, logrando reconocer todas las categorías y garantizando que la diferencia entre el valor obtenido y el esperado es el mínimo posible. Si bien no es el mejor resultado el de este entrenamiento, dado que no logra reducir el error, si nos deja regularizar las curvas de errores, tanto en el entrenamiento como en la validación. Por este motivo se decidió dejar la tasa de aprendizaje del optimizador *SGD* en 0,001 para los siguientes experimentos.

La siguiente prueba que se hizo, se llevó a cabo sin aplicar la técnica de *Transfer Learning* con el optimizador previamente mencionado. Se llevaron a cabo 2

pruebas con estas configuraciones que variaron en la cantidad de épocas de entrenamiento. Comencemos analizando la prueba de 50 épocas de entrenamiento con el optimizador *SGD* con tasa de aprendizaje de 0,001. A continuación, podemos observar los resultados obtenidos en las métricas obtenidas:

- *MAE* (entrenamiento / validación): 0.6153 / 0.7108 (± 1.159)
- *MSE* (entrenamiento / validación): 0.5833 / 0.8436 (± 5.110)

Ahora bien, veamos las gráficas de los errores para poder realizar una mejor evaluación del escenario brindado por este experimento:

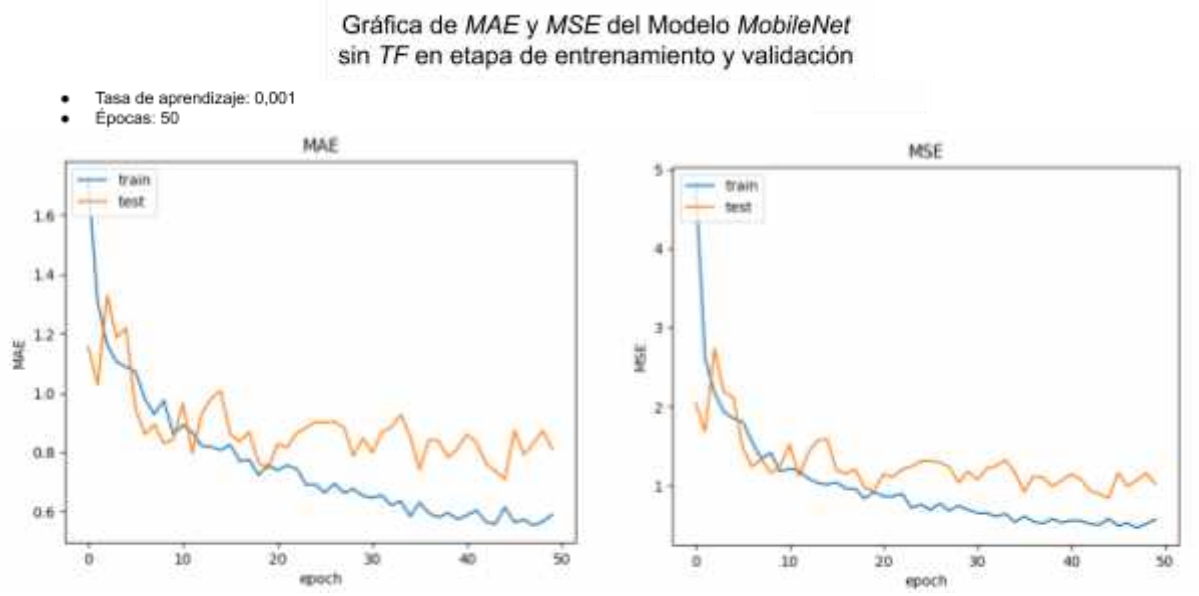

Figura 7.9 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación en 50 épocas del modelo *MobileNet* con tasa de aprendizaje de 0,001 sin *TL*

Notamos que el *MAE* y *MSE* obtenidos en este experimento es mayor al obtenido en los entrenamientos con la técnica de *TF*, pero a su vez, este entrenamiento logró mejorar ampliamente la desviación obtenida en la validación, llegando a 1.15 en contra de un 10% de aumento en los errores calculados aproximadamente. De la gráfica, podemos notar que no tiene un entrenamiento homogéneo en las primeras épocas para la validación, y luego comienza a tender entre 0.8 y 1, mientras que la curva de entrenamiento denota un aprendizaje mucho más gradual y parejo. Como mencionamos previamente, esta diferencia en las curvas de los errores de validación y entrenamiento, puede estar dada, principalmente, por la cantidad de datos que tiene el conjunto de validación en comparación al de entrenamiento. Además, podemos ver en la gráfica que las curvas de error comienzan a divergir a partir de la época 25 aproximadamente, pero al tener saltos bastante pronunciados, no se ve claramente si el modelo está padeciendo de un gran *overfitting*, ya que quizás se logre mejorar con un número

mayor de épocas de entrenamiento. Por este motivo, se decidió hacer una última prueba con el modelo *MobileNet* sin *TF*, para evaluar mejor si el modelo padece o no de *overfitting*. Este último entrenamiento se lo realizó con el mismo optimizador, sin *TF* durante 100 épocas. A continuación podemos ver la gráfica de los errores de este nuevo entrenamiento:

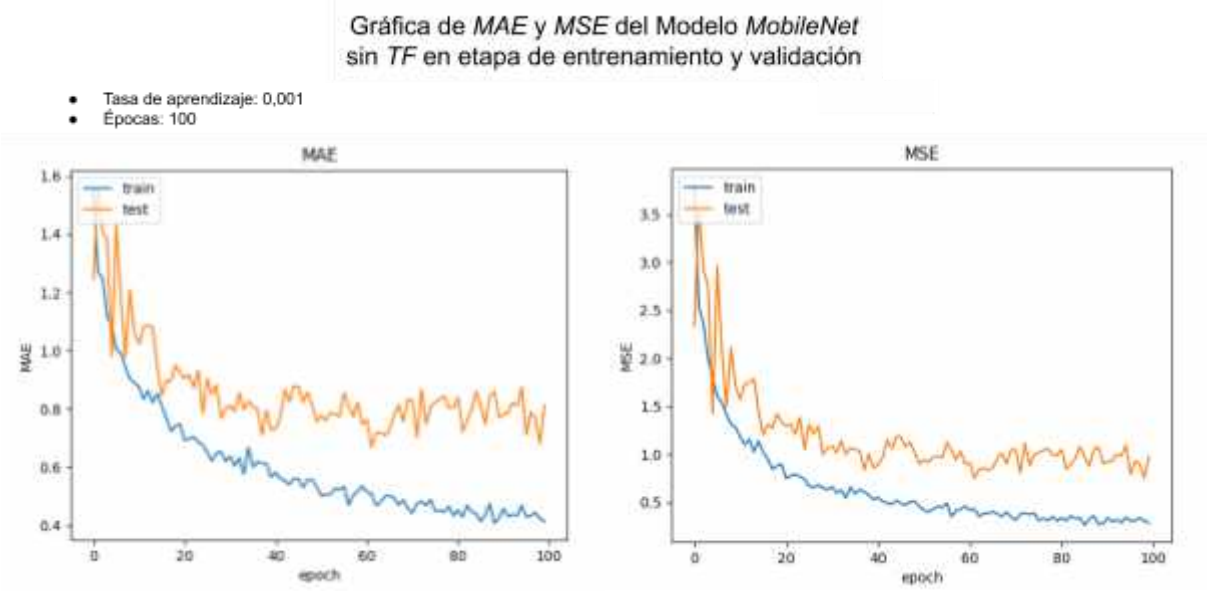

Figura 8.0 Curva del error absoluto medio y cuadrático durante el entrenamiento y la validación en 100 épocas del modelo *MobileNet* con tasa de aprendizaje de 0,001 sin *TL*

Viendo esta gráfica, podemos determinar que el modelo comienza a sufrir de *overfitting* cerca de la época 40, como habíamos intuido en el experimento anterior. Se ve como la curva de error en el entrenamiento va mejorando en cada época, mientras que la de validación queda oscilando cerca entre 0,8 y 1. Ahora bien, veamos la totalidad de las métricas calculadas para este experimento:

- *MAE* (entrenamiento / validación): 0.5099 / 0.6685 (± 1.133)
- *MSE* (entrenamiento / validación): 0.4296 / 0.7462 (± 4.836)

Podemos notar que la desviación estándar obtenida con esta experimentación se redujo un poco pero por el otro lado el *MAE* y *MSE* han mejorado. Esto está relacionado a haberle dado más épocas de entrenamiento, ya que de este modo se le permitió al modelo poder corregir un poco más lo aprendido, pero, no hay que olvidar que el modelo tiene *overfitting* a partir de la época 40 aproximadamente, por lo que la mejoría en *MAE* y *MSE* no podemos estar seguros de que en realidad el modelo se esté comportando mejor.

Con esto hemos finalizado de analizar todas las experimentaciones con los modelos de procesamiento de esta investigación.

De las últimas puntuaciones obtenidas podemos afirmar que mientras más volumen de información tengamos en nuestro conjunto de datos, el aprendizaje va a ser más regular y homogéneo.

Más allá de las distintas observaciones que se fueron realizando en cada uno de los entrenamientos, en todos ellos notamos que el comportamiento de los modelos de *DL* son mejores que los resultados obtenidos con los algoritmos de detección de características y el clasificador *SVR* utilizados previamente. A continuación, se brinda una tabla comparativa de los mejores resultados obtenidos con cada una de las técnicas utilizadas durante la investigación de esta tesina.

Tabla 6 - Comparativa de los mejores resultados obtenidos con cada una de los modelos entrenados para *SVR* y *Deep Learning*

SVR		
HOG	**MAE**	**MSE**
hog_svr_linear_8x8	**1.145060349 (± 0.726)**	**1.838883835 (± 2.019)**
hog_svr_rbf_16x16	1.205932601 (± 0.705)	1.985026491 (± 1.774)
LBP	**MAE**	**MSE**
lbp_svr_rbf_p24_r8	1.199963604 (± 0.706)	1.96946065 (± 1.802)
lbp_svr_linear_p64_r5	**1.150305714 (± 0.712)**	**1.830457383 (± 1.717)**
Haralick	**MAE**	**MSE**
haralick_svr_linear	5.248228925 (± 8.017)	50.22492341 (± 87.817)
haralick_svr_rbf	**1.33525974 (± 0.711)**	**2.557380557 (± 3.078)**

Deep Learning				
	MAE		**MSE**	
Modelo Simple	**training**	**validation**	**training**	**validation**
sgd_default_300_epocas	**0.5213**	**0.6121 (± 1.067)**	**0.4356**	**0.6098 (± 4.160)**
Modelo MobileNet	**training**	**validation**	**training**	**validation**
sgd_default_300_epocas_con_TF	**0.6339**	**0.6422 (± 0.877)**	**0.6296**	**0.6831 (± 2.909)**
Modelo Two Lanes	**training**	**validation**	**training**	**validation**
sgd_lr=0.001_300_epocas	**0.3737**	**0.5313 (± 1.058)**	**0.2368**	**0.5492 (± 4.045)**

Habiendo visto todas las pruebas realizadas con los modelos de clasificación y habiendo descrito la arquitectura y cada una de las aplicaciones, se procederá, en la siguiente sección, a dar una visualización holística de todas estas partes previamente descritas, para ver cómo sería la aplicación en funcionamiento en un ambiente utilizable.

3.5 Puesta en marcha

Luego de haber visto cada una de los componentes de la aplicación, ahora ahondaremos en la puesta en producción de nuestras aplicaciones para lograr que sea accesible desde internet. Para lograr esto, se hizo uso de un servidor *PaaS* (*Platform as a Service*) que nos permite abstraernos completamente de la administración del servidor ya que se encarga de realizar el *deploy* de la aplicación y dejarlo accesible a través de internet en un dominio público. En este caso se hizo uso de la plataforma Heroku, que nos facilita *tiers* gratuitos para poder desplegar aplicaciones sin un costo mensual con ciertas características, en caso de que la aplicación crezca y el flujo a la misma lo haga también, se debería cambiar a un *tier* que nos permita escalar nuestra plataforma.

Generalmente, luego de las pruebas con modelos de *Machine Learning* y *Deep Learning*, lo que se hace es seleccionar el modelo que satisface mejor las necesidades del problema planteado. Una vez elegido el modelo de todos los disponibles, se realiza un nuevo entrenamiento sobre el 100% de los datos que tenía el *dataset* original. Esta es una práctica normal dentro del campo del *Deep Learning* ya que se busca que el modelo tenga el mayor volúmen de información posible para entrenarlo, para lograr generalizar mejor y reducir su error de clasificación. Es por esto, que se decidió tomar el modelo *Two Lanes* con 300 épocas de entrenamiento con los parámetros con que se lo había encontrado para realizar este nuevo entrenamiento y así tener un modelo listo para la puesta en marcha. Las métricas del nuevo entrenamiento de este modelo fueron las siguientes:

- Error Absoluto Medio: 0.31852
- Error Cuadrático Medio: 0.169283

Vale destacar de los resultados de la etapa de entrenamiento obtenidos, que si traspolamos el comportamiento de los entrenamientos previos del modelo *Two Lanes* había una diferencia entre las métricas de entrenamiento y validación producto del *overfitting* detectado en la etapa de entrenamiento. Es por esto que podríamos intuir, que estos valores obtenidos en la etapa de entrenamiento también

padecen de un sobreajuste, por lo que el error real del modelo sería levemente mayor.

Una vez entrenado el modelo seleccionado, lo que generalmente se hace, es generar un nuevo conjunto de datos, normalmente llamado conjunto de prueba o *test* para validar cómo se comporta el modelo entrenado con el 100% del conjunto de datos inicial. El nuevo conjunto de prueba no debe tener el mismo volúmen de información que el de entrenamiento y de validación, ya que lo que se busca hacer es ver que tan bien se comporta el modelo con este nuevo entrenamiento, por lo que se suele generar un conjunto mucho más reducido para probar ya que se supone que es el mejor modelo encontrado. En este apartado haremos uso de un nuevo conjunto de imágenes para validar el funcionamiento del modelo a través de la aplicación desplegada en Heroku con la aplicación Ionic desarrollada en un *smartphone* con sistema operativo Android. A continuación podemos ver el flujo de ejecución y todas las partes que intervienen en la utilización de la plataforma.

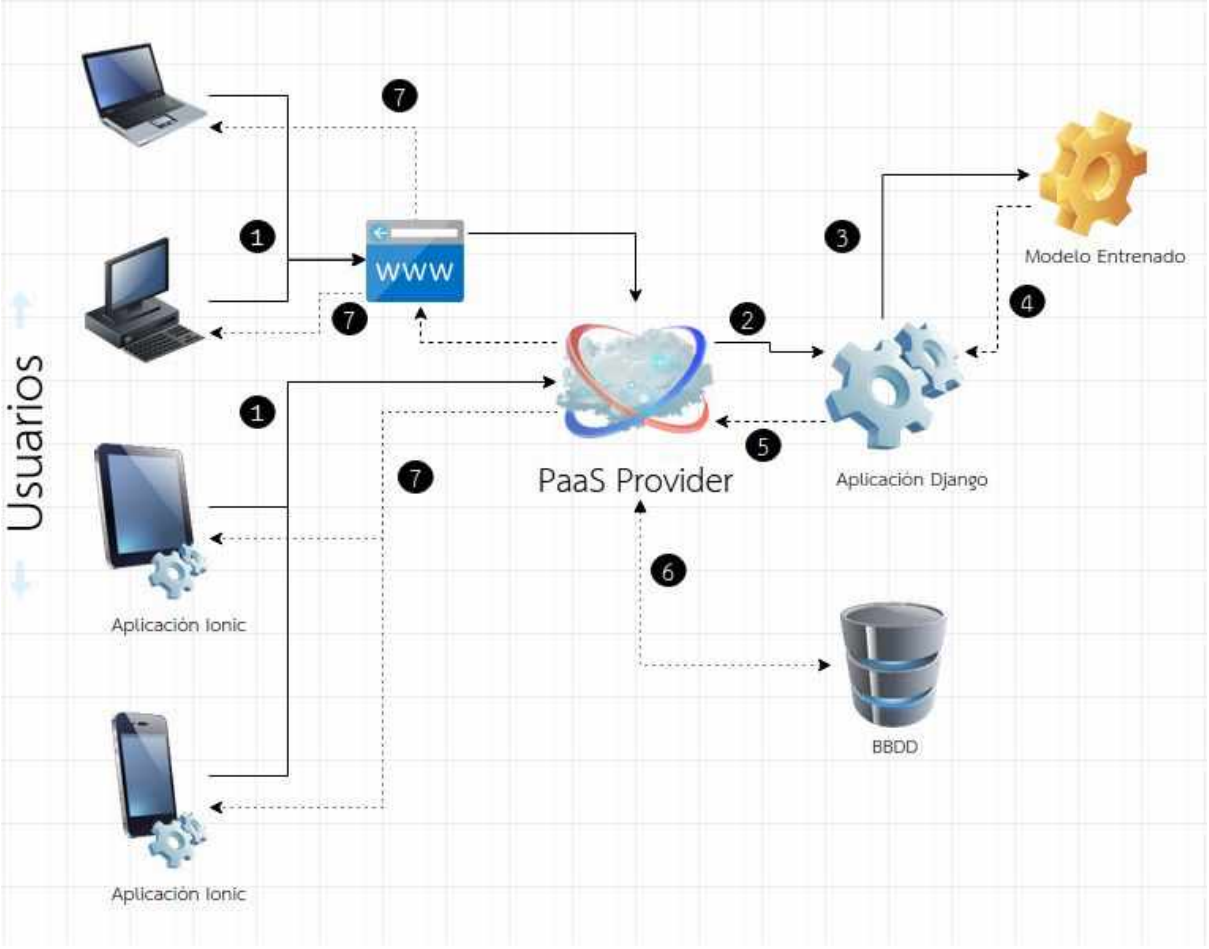

Figura 8.1 Esquema de funcionamiento de la aplicación en ambiente de producción

Como podemos observar, la aplicación con el modelo clasificador se la puede acceder de diversos dispositivos, ya sean computadoras de escritorio, *laptops* a través de un navegador o de un posible cliente pesado o por dispositivos móviles

utilizando la aplicación Ionic desarrollada en este trabajo. En este caso utilizamos esta última que mencionamos para poder interactuar y conseguir que nuestra imagen sea catalogada por nuestra aplicación *backend*.

Una vez hecho el *deploy* de la aplicación en nuestro proveedor *PaaS*, nuestra *API* queda expuesta bajo el dominio que el proveedor nos asigna o en caso de haber comprado algún dominio se podría configurar la plataforma para utilizar el mismo. En este caso, la aplicación Django desarrollada queda expuesta bajo el dominio https://ayudante-de-cocina.herokuapp.com/, con lo que si deseamos analizar una imagen en particular basta con direccionar la solicitud con la imagen en cuestión hasta el *endpoint* específicado, https://ayudante-de-cocina.herokuapp.com/pictures/upload_docs/. En la imagen siguiente podemos observar los logs de ejecución que se realizan en nuestro servidor a la hora de procesar una imagen enviada desde la aplicación Ionic.

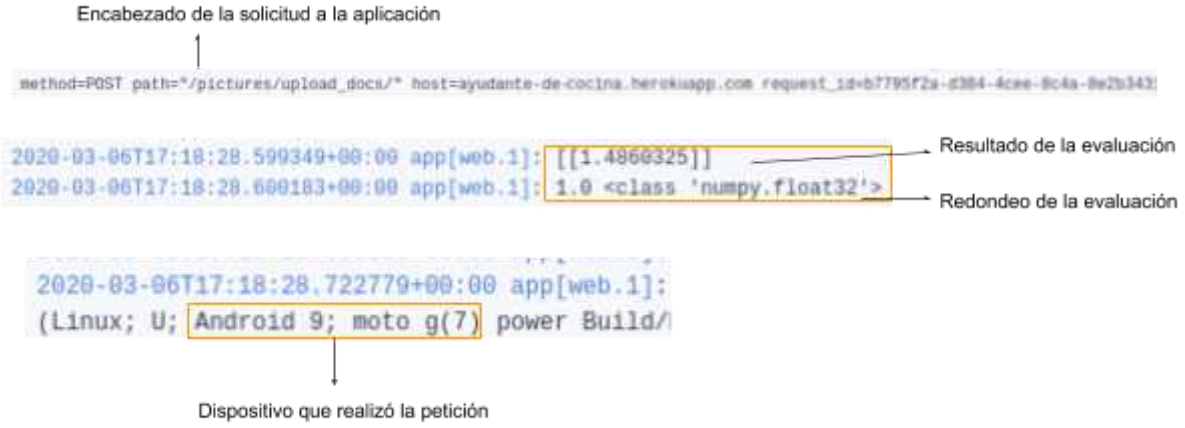

Figura 8.2 Logs de la aplicación cuando se solicita evaluar una imagen a la aplicación alojada en Heroku

Una vez clasificada la imagen, la aplicación guarda la imagen en la base de datos y devuelve el resultado de la clasificación a la aplicación que hizo esta petición, que en este caso se llevó a cabo desde la aplicación móvil desarrollada en Ionic. Vale destacar, que al resultado obtenido con el modelo, se le aplica una función escalón para aproximar el resultado al número natural más cercano. El motivo de esto es poder devolverle al usuario de la aplicación un estado de cocción en particular y no un número real que queda a interpretación del usuario; haciendo el uso de la aplicación lo más intuitiva y fácil posible. A continuación se brinda una serie de ejemplos realizados con el modelo candidato seleccionado, donde se deja en evidencia el funcionamiento de la plataforma integralmente. Se procede a visualizar las imágenes y los resultados de la clasificación utilizando la interfaz de la aplicación móvil desarrollada, donde se deja en evidencia cuando se envía la imagen para clasificar y sus respectivos resultados.

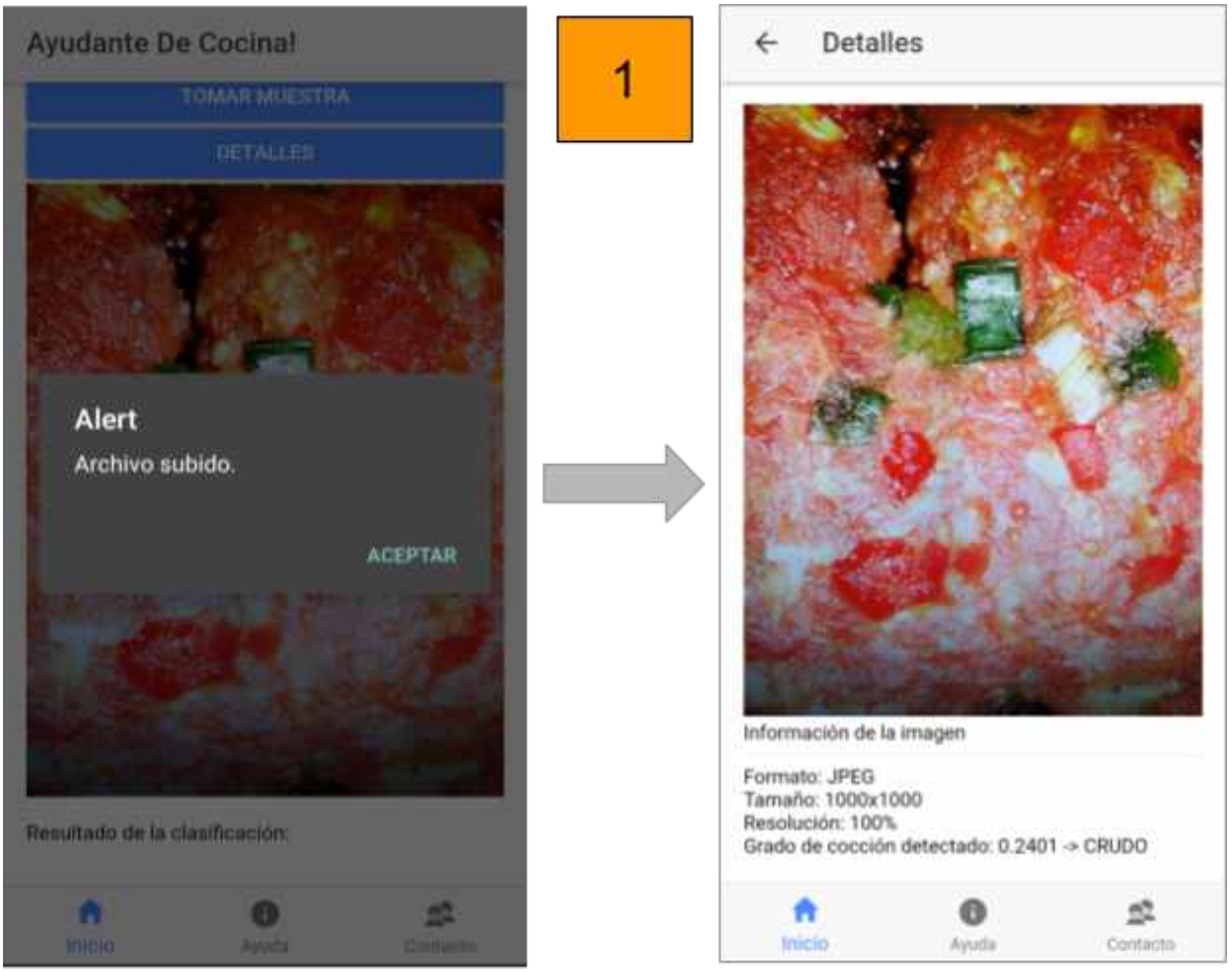

Figura 8.3 Procesamiento de una imagen de un corte en estado "Crudo"

La primer imagen que se analizó corresponde a un corte de carne en estado "Crudo". Como podemos apreciar en la figura anterior el modelo fue capaz de catalogarla correctamente, devolviendo el valor de *0.2401*. A este valor, luego, se le aplica una función escalón para poder hacer una aproximación a una categoría de las planteadas para poder devolverle un resultado "significativo" al usuario de la aplicación. El valor resultante de esta operación es la categoría 0 que corresponde a la etiqueta de "Crudo".

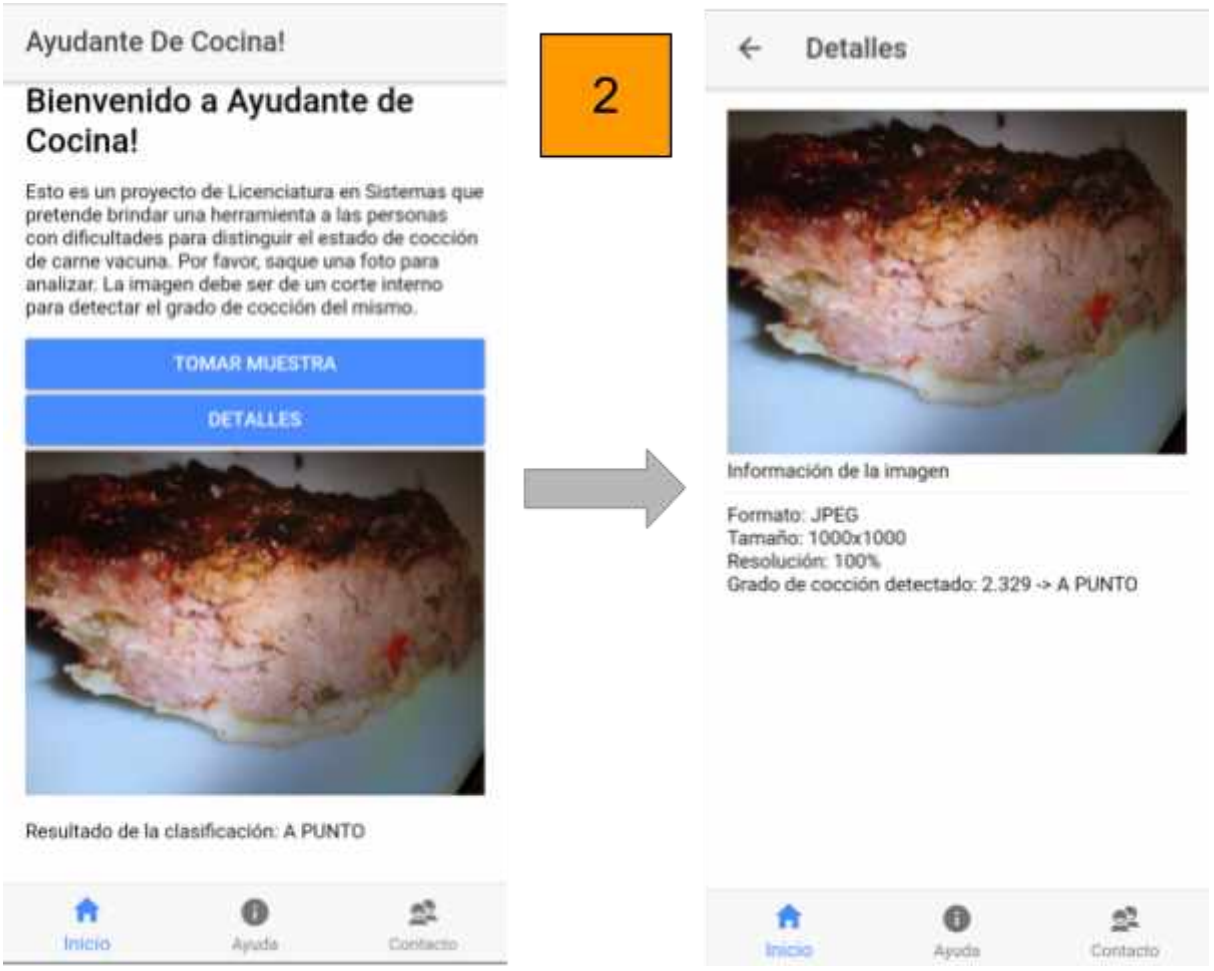

Figura 8.4 Procesamiento de una imagen con un corte en estado "A Punto"

La segunda imagen analizada, la cual la podemos apreciar en la figura anterior, se trata de un corte de carne en estado "A Punto", ya que se veían ciertos matices rosados en la captura. El valor resultante del modelo luego de procesar la imagen, como podemos apreciar, fue de *2.329*. Aplicando el mismo procedimiento descrito con la Figura 8.3, luego de aplicar la función escalón, la categoría obtenida es la número 2, que como se mencionó previamente, corresponde a la etiqueta "A Punto".

Por último se hizo una prueba con una imagen de un corte en estado "Cocido" ó "Bien Cocido", dependiendo del parecer de la persona, que la podemos ver en la figura siguiente.

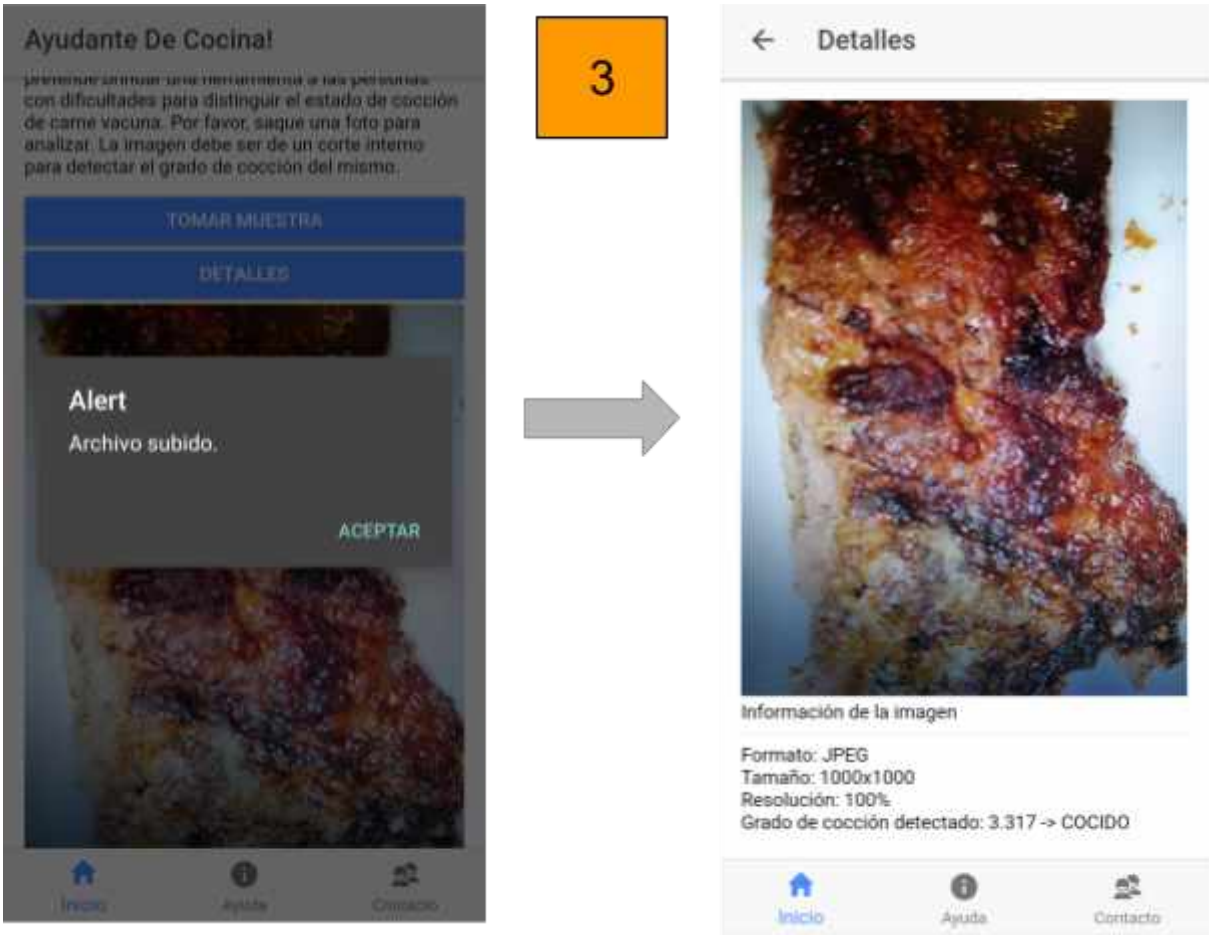

Figura 8.5 Procesamiento de una imagen con un corte en estado "Cocido"

El valor resultante del procesamiento de la última evaluación fue de *3.317*, como se puede observar en la Figura 8.5. Queda en evidencia, que al aplicarle la función escalón a este valor la etiqueta resultante es "Cocido". Por esto podemos decir que el modelo etiquetó dicha imagen dentro de los resultados esperados para la imagen en cuestión.

Como se pudo apreciar en las pruebas previamente vistas, si bien los puntos de cocción son bastante subjetivos, el modelo estaría etiquetando bastante acorde a las categorías propuestas en un inicio.

Habiendo visto todo el procedimiento realizado durante esta tesina y la puesta en producción de la plataforma, se procede a ver en el siguiente capítulo, las conclusiones que se obtuvieron en base a la investigación llevada a cabo.

Capítulo 4: Conclusiones

Luego de haber visto todo el estudio que se llevó a cabo durante la investigación de este trabajo, haremos una revisión de las partes más importantes de la misma, tanto relativas a la problemática del reconocimiento de imágenes como todas aquellas aplicaciones desarrolladas para dar soporte a nuestro clasificador de imágenes.

Comenzaremos por hacer una revisión de las aplicaciones de soporte, la aplicación Django y Ionic 3. Estas fueron aplicaciones sencillas que brindaron una base para poder desarrollar el *PoC* (*Proof of Concept*) de nuestro clasificador de imágenes. Como bien se menciona, son aplicaciones básicas, por lo que no se han tenido en cuenta muchos aspectos que una aplicación en producción debería contar, pero se logró demostrar que la arquitectura de la plataforma planteada es viable y nos permite el consumo de nuestro modelo de regresión capaz de distinguir entre las categorías planteadas en este trabajo.

Se pudo realizar una demo completa del funcionamiento total de las aplicaciones utilizando el mejor modelo obtenido, entrenado con el 100% del *dataset* para simular la puesta en producción del mismo, logrando brindar una base de prueba que puede ser utilizada y extendida fácilmente para continuar la investigación sobre la temática de este trabajo.

Otro punto importante para remarcar fue todo el proceso de generación del conjunto de datos para llevar a cabo la investigación. Este conjunto se tuvo que desarrollar desde 0, teniendo que indagar cuáles eran los puntos claves para generar buenos conjuntos de datos, aquellos puntos que lograran un conjunto robusto que no generara demasiado ruido para el entrenamiento de los modelos. Esto implicaba que se debían utilizar la mayor cantidad posible de dispositivos con los cuales se tomaban las imágenes, para generar variación en las calidades, ya sea por la resolución de la cámara, como también, por la conformación de la misma. Otro aspecto fundamental fue poder sacar muestras con distintos escenarios, distintas cantidades lumínicas, en distintos ambientes y con diversos métodos de cocción. Todo esto, generó que la realización del conjunto de datos tuviese varios parámetros con los que se debía lidiar para poder llegar a tener un buen conjunto de información para poder realizar la investigación de los clasificadores satisfactoriamente, culminando en una base de datos de imágenes de cortes de carne vacuna.

Además, se realizó todo un estudio sobre los campos de *Machine Learning* y *Deep Learning* para poder utilizar modelos de dichos campos para ver cómo se comportaban con la temática de esta investigación. En el área de *ML*, se investigaron diversos descriptores que nos permitieron obtener distintas características de un mismo conjunto de datos. Se investigaron diversas formas de visualizar estos resultados, para poder interpretar qué es lo que se estaba obteniendo con los mismos. Se utilizaron los resultados obtenidos por los descriptores para entrenar sistemas *SVR* y se analizó cómo respondían estos modelos con cada uno de los distintos descriptores, arribando a un modelo con un Error Absoluto Medio de 1,14 con desviación estándar de 0.726, utilizando el descriptor *HOG* como capturador de características.

A su vez, se investigaron las técnicas más utilizadas en la actualidad en el área del *DL*, para lograr un modelo y poder compararlo al "mejor" modelo *SVR* obtenido. El mejor modelo de *DL* obtenido fue el denominado en este trabajo como "*Two Lanes*", logrando reducir el *MAE* a 0.53 con desviación estándar de 1.05 en comparación a *SVR*.

Para la generación de los modelos de *DL* se buscaron las formas de optimizar los modelos, las formas de reducir el *overfitting* durante el entrenamiento de los mismos; se investigaron arquitecturas avanzadas de *Deep Learning* para ver cómo funcionaban, el tiempo que consumían, los recursos que se debían disponer para poder ejecutarlas y las diferentes problemáticas que podía resolver cada una de las mismas, llegando a la conclusión de que la técnica de *Deep Learning* logra modelos con mejores rendimientos a la hora de clasificar las categorías planteadas en este trabajo, pero con una contraparte de que requieren una mayor capacidad de cómputo, requiriendo hardware más costoso y más tiempo para lograr que el modelo converja, ya que el entrenamiento de las *SVR* rondaba por los 5 minutos (dependiendo el descriptor utilizado y sus parámetros) mientras que los tiempos de entrenamiento en *DL* rondaban entre 5 a 9 horas con el hardware previamente mencionado.

También se pudo observar que la variación en el volumen de información en modelos de *Deep Learning* hace variar mucho el aprendizaje de los mismos. Mientras más información se disponga, el modelo tiene curvas de aprendizaje mucho más graduales y certeras.

Se pudo comparar los resultados obtenidos por cada uno de los campos analizados, determinando, en base a los resultados obtenidos, que los modelos de *Deep Learning* se comportan de forma mucho más adaptativa y logran errores mucho menores a los enfoques utilizados con *SVR*, con los descriptores seleccionados para el desarrollo de esta tesina.

Este trabajo, como se mencionó anteriormente, es un inicio para el estudio de estados de cocción de diversos tipos de carne con técnicas de análisis de imágenes digitales. Brinda un estudio en el cual futuras investigaciones que intenten abordar la misma temática puedan basarse. Da un conjunto de aplicaciones y de bases de

conocimiento que permitirán un estudio más profundo de la temática, pudiendo utilizar este trabajo como guía para profundizar más la temática planteada.

Habiendo visto las conclusiones resultantes, continuaremos el trabajo con las distintas líneas de investigación que pueden surgir de esta investigación.

Capítulo 5: Trabajos Futuros

A lo largo del trabajo, vimos las distintas partes que conforman el todo de esta investigación, que brinda como resultado una prueba de concepto para la clasificación de estados de cocción en alimentos, específicamente en este caso, la carne vacuna. Es por esto, que podríamos utilizar el conocimiento generado por esta investigación como un punto de inicio para futuras investigaciones sobre análisis de estados de cocción en distintos productos.

Durante el desarrollo de la tesina se fueron dejando indicios de qué cosas se podrían mejorar o probar de hacer de otra forma para poder comparar los resultados. A continuación se brindarán líneas de investigación que pueden partir de esta investigación.

Para comenzar podemos mencionar la optimización de la aplicación móvil. La aplicación desarrollada hasta el momento, como se mencionó en el apartado donde se describió la misma, es una aplicación para demostrar la prueba de concepto del objetivo de esta tesina, por lo que una línea de investigación posible sería la de mejorar la aplicación Ionic para brindar mayor funcionalidad al usuario final del sistema. Sumado a la mejora en la funcionalidad de la misma, se podría investigar cómo entrenar modelos de Inteligencia Artificial nativamente en el dispositivo móvil, para no necesitar de un *backend* de procesamiento, el cual va a ser consumido por nuestra aplicación *frontend*.

Siguiendo con la línea de investigación de la aplicación móvil, podríamos mejorar nuestro modelo *backend*, nuevamente para brindar más funcionalidad y generar una plataforma más robusta para el usuario del sistema.

Otra línea de investigación que se puede seguir es la relativa a los modelos que se utilizaron como clasificadores de nuestro problema. Estos son *SVR*, en el campo de *Machine Learning* y los 3 modelos utilizados en *Deep Learning*, el MobileNet, el *Two Lanes* y el modelo simple. Con respecto a *SVM*, se podría seguir investigando el desempeño que puede brindar un *SVR* con distintos descriptores a los utilizados (*HOG*, *LBP* y Haralick). También, se podría seguir modificando el conjunto de entrenamiento de dicho modelo para ver cómo se comporta el modelo a una entrada de datos mayor, con distintos filtros y ver cómo se clasifican los histogramas obtenidos de dichos descriptores. Además, se podrían investigar otras técnicas dentro del *ML* para regresión y utilizar el conjunto de datos utilizado. Por otro lado, si se dispusiese de *hardware* específico para el entrenamiento de redes neuronales convolucionales, se podría llevar a cabo la prueba del entrenamiento de la red neuronal avanzada InceptionV3 para ver los resultados que se obtienen con la

problemática en cuestión con otra red avanzada con y sin la utilización de *Transfer Learning* y comparar los resultados a los expuestos en este trabajo.

También, se podría agrandar el conjunto de datos utilizado con el objetivo de generar mayor diversidad en las imágenes. Luego, con ese nuevo *dataset*, re entrenar los modelos utilizados y observar sus resultados para una posterior comparación con los resultados obtenidos en este trabajo.

También, se podría investigar más en profundidad la detección de la eliminación de bacterias durante la cocción de alimentos con técnicas de procesamiento de imágenes digitales. Esto sería un gran aporte para la sociedad ya que permitiría poder determinar si el consumo de una comida con carne vacuna es apta para el consumo humano, pudiendo así prevenir la ingesta de productos nocivos para el ser humano que puedan derivar en problemas de salud.

Por último, se podría hacer una evaluación más exhaustiva de cómo desplegar las aplicaciones de soporte, ya sea incluyendo el procesamiento de la imagen tomada directamente en la aplicación móvil en contra de tener el *backend* en un servidor y dejar la aplicación de *frontend* como consumidor de la *API* de procesamiento (como se implementó en esta tesina) en pos de mejorar los tiempos de respuesta del sistema global. Si bien se propone hacer esta investigación, no se ha encontrado pruebas suficientes que impulsen en demasía este tópico, ya que los tiempos de comunicación entre ambas aplicaciones no ha sido demasiado amplio como para desestimar la arquitectura planteada para este trabajo.

Si bien estas son posibles líneas de investigación, el lector de esta tesina se le podrían ocurrir otras, dado que como se dijo, es un campo con muchísima variabilidad y potencial, por lo que no está limitado a sólo este conjunto de posibilidades que se enunciaron.

Capítulo 6: Referencias Bibliográficas

[1] Organización Mundial de la Salud, Salud ocular universal: un plan de acción mundial para 2014-2019 [Internet], España: Catalogación por la Biblioteca de la OMS, 2013. Disponible en: http://www.who.int/blindness/AP2014_19_Spanish.pdf.

[2] Kauffmann, Maricela, Arte culinario tradicional: identidad y patrimonio de las culturas de la Costa Caribe de Nicaragua, 1era edición, Managua: CRAAN, 2012. Disponible en: http://unesdoc.unesco.org/images/0022/002283/228337S.pdf.

[3] Ahmad Babaeian Jelodar, Md Sirajus Salekin, and Yu sun, Identifying Object States in Cooking-Related Images [Internet], 2018. Disponible en: https://arxiv.org/pdf/1805.06956.pdf.

[4] Derico Setyabrata, Young L. Kim, Yuan H. Brad Kim, ANISOTROPY SCANNING: Novel Imaging Analysis for Beef Tenderness, Journal of Purdue Undergraduate Research, 6, 49–55. [Internet], 2016. Disponible en: https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1230&context=jpur.

[5] Pankaj B. Pathare, Anthony Paul Roskilly, Quality and Energy Evaluation in Meat Cooking, Food Engineering Reviews 8(4):435–447, 2016.

[6] Weiwei Cheng, Jun-Hu Cheng, Da-Wen Sun, and Hongbin Pu, Marbling Analysis for Evaluating Meat Quality: Methods and Techniques, Comprehensive Reviews in Food Science and Food Safety [Internet], 14(5):523-535, 2015. Disponible en: https://onlinelibrary.wiley.com/doi/full/10.1111/1541-4337.12149.

[7] François Chollet, Deep Learning with Python, Estados Unidos: Manning Publications Co, 2018.

[8] Maja Prevolnik, Dejan Škorjanc, Marjeta Čandek-Potokar and Marjana Novic, “Application of Artificial Neural Networks in Meat Production and Technology”, Artificial neural networks - Industrial and control engineering applications 223-240, 2011.

[9] Kusworo Adi, Sri Pujiyanto, Oky Dwi Nurhayati, and Adi Pamungkas, Beef Quality Identification Using Thresholding Method and Decision Tree Classification Based on Android Smartphone”, Journal of Food Quality 2017(9):1-10, 2017.

[10] Nek Valous y Da-Wen Sun, Image processing techniques for computer visión in the food and beverage industries, Woodhead Publishing Series in Food Science, Technology and Nutrition, (capítulo 4)97-129, 2012.

[11] Ana Paula Ayub da Costa Barbona, Sylvio Barbon Jr., Gabriel Fillipe Centini Campos, José Luis Seixas Jr., Louise Manha Peres, Saulo Martielo Mastelini, Nayara Andreo, Alessandro Ulrici, Ana Maria Bridi, Development of a flexible Computer Vision System for marbling classification, Computers and Electronics in Agriculture, 2017..

[12] Tensorflow, Disponible en: https://www.tensorflow.org/?hl=es

[13] Keras, Disponible en: https://keras.io/

[14] OpenCV-Python, Disponible en: https://opencv-python-tutroals.readthedocs.io/en/latest/
[15] Hokuto Kagaya, Kiyoharu Aizawa, Makoto Ogawa, Food Detection and Recognition Using Convolutional Neural Network, 1085-1088, 2014.
[16] Organización Mundial de la Salud (*WHO*), Ceguera y discapacidad visual [Internet], 2018. Disponible en: http://www.who.int/es/news-room/fact-sheets/detail/blindness-and-visual-impairment.
[17] Alegría Irene et al, Estudio de una Anomalía Genética: El Daltonismo [Internet], 2013. Disponible en: https://www.cac.es/cursomotivar/resources/document/2012/005.pdf.
[18] Nikhil Buduma and Nicholas Lacascio, Fundamentals of Deep Learning, Estados Unidos: O'Reilly Media Inc., 2017.
[19] Mountcastle, Vernon B. Modality and topographic properties of single neurons of cat's somatic sensory cortex, Journal of Neurophysiology 20.4: 408-434, 1957.
[20] NVIDIA, Deep Learning SDK Documentation (cuDNN). Disponible en: https://docs.nvidia.com/deeplearning/sdk/cudnn-install/ .
[21] Patrick Rodriguez, Food Classification with Deep Learning in Keras / Tensorflow, 2017. Disponible en: https://blog.stratospark.com/deep-learning-applied-food-classification-deep-learning-keras.html#deep-learning-applied-food-classification-deep-learning-keras.
[22] Takuma Maruyama, Yoshiyuki Kawano, Keiji Yanai, Real-time Mobile Recipe Recommendation System Using Food Ingredient Recognition, 27-34, 2012.
[23] Yushiyuki Kawano y Keiji Yanai, Real-time Mobile Food Recognition System, IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops:1-7, 2013.
[24] Laura Liu, Michael Ngadi, Automatic assessment of meat marbling and tenderness, CMSA News:27-32, 2014.
[25] PyInquirer, https://github.com/CITGuru/PyInquirer.
[26] PyFiglet, https://github.com/pwaller/pyfiglet.
[27] Postman, https://www.getpostman.com/.
[28] Gradle, https://gradle.org/install/.
[29] Java, https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[30] Android Studio, https://developer.android.com/studio/install.
[31] Kevin Salton Do Prado, Face Recognition: Understanding LBPH Algorithm, 2017. Disponible en: https://towardsdatascience.com/face-recognition-how-lbph-works-90ec258c3d6b.

[32] Robert M. Haralick, K. Shanmugam y Its'hak Dinstein , Textural Features for Image Classification, Studies in Media and Communication SMC-3(6):610-621, 1973.

[33] Navneet Dalal y Bill Triggs, Histograms of Oriented Gradients for Human Detection, IEEE Conference on Computer Vision and Pattern Recognition, 2005. Disponible en: http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf.

[34] Alexander Mordvintsev & Abid K., Image Gradients Open CV*,* 2013. Disponible en: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_gradients/py_gradients.html.

[35] Sergey Ioffe and Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate, 2015.

[36] learnopencv.com [Internet]. Satya Mallick; 2016. Histogram of Oriented Gradients. Disponible en: https://www.learnopencv.com/histogram-of-oriented-gradients/

[37] saedsayad.com [Internet]. Saed Sayad. Support Vector Machine - Regression (SVR). Disponible en: https://www.saedsayad.com/support_vector_machine_reg.htm.

[38] Sinno Jialin Pan and Qiang Yang, A Survey on Transfer Learning, IEEE Transactions on Knowledge and Data Engineering, Volúmen 22: 1345-1359, 2009. Disponible en: https://ieeexplore.ieee.org/document/5288526.

[39] Dipanjan Sarkar, A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning, 2018. Disponible en: https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a.

[40] Reza Fuad Rachmadi and I Ketut Eddy Purnama, Vehicle Color Recognition using Convolutional Neural Network, 2018. Disponible en: https://arxiv.org/pdf/1510.07391.pdf.

[41] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115(3):211–252, 2015.

[42] Deng, J., Berg, A. C., Li, K., and Fei-Fei, L. What does classifying more than 10,000 image categories tell us? In Daniilidis, K., Maragos, P., and Paragios, N., editors, Computer Vision – ECCV 2010, pages 71–84, Berlin, Heidelberg. Springer Berlin Heidelberg, 2010.

[43] Deng, J., Russakovsky, O., Krause, J., Bernstein, M., Berg, A. C., and Fei-Fei, L. Scalable multi-label annotation. In ACM Conference on Human Factors in Computing Systems (CHI), 2014.

[44] Kuznetsova, A., Rom, H., Alldrin, N., Uijlings, J., Krasin, I., Pont-Tuset, J., Kamali, S., Popov, S., Malloci, M., Duerig, T., and Ferrari, V. The open

images dataset v4: Unified image classification, object detection, and visual relationship detection at scale, 2018. Disponible en: https://arxiv.org/abs/1811.00982.

[45] Antol, S., Agrawal, A., Lu, J., Mitchell, M., Batra, D., Zitnick, C. L., and Parikh, D. VQA: Visual Question Answering. In International Conference on Computer Vision (ICCV), 2015.

[46] Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading digits in natural images with unsupervised feature learning, 2011.

[47] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[48] Boser, Bernhard & Guyon, Isabelle & Vapnik, Vladimir. A Training Algorithm for Optimal Margin Classifier. Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory. 5. 10.1145/130385.130401, 1996.

[49] Dahl, G. E., Sainath, T. N., and Hinton, G. E. Improving deep neural networks for lvcsr using rectified linear units and dropout. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pages 8609–8613, 2013.

[50] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15:1929–1958, 2014.

[51] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, 2017. Disponible en: https://arxiv.org/abs/1704.04861v1.

[52] Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton. On the importance of initialization and momentum in deep learning, Proceedings of the 30th International Conference on Machine Learning, PMLR 28(3):1139-1147, 2013.

[53] Navneet Dalal. Finding People in Images and Videos. Human-Computer Interaction [cs.HC]. Institut National Polytechnique de Grenoble - INPG, 2006. Disponible en: https://tel.archives-ouvertes.fr/tel-00390303/document.

[54] Lowe, David. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision. 60. 91-. 10.1023/B:VISI.0000029664.99615.94, 2004.

[55] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10). Omnipress, Madison, WI, USA, 807–814., 2010. Disponible en: https://www.cs.toronto.edu/~fritz/absps/reluICML.pdf.

[56] Rosenblatt, F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Cornell Aeronautical Laboratory,

Psychological Review, 65, 386-408, 1958. Disponible en: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf.

[57] Palm, G. Warren mcculloch and walter pitts: A logical calculus of the ideas immanent in nervous activity. In Palm, G. and Aertsen, A., editors, Brain Theory, pages 229–230, Berlin, Heidelberg. Springer Berlin Heidelberg, 1986. Disponible en: https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.

[58] Ren, S., He, K., Girshick, R. B., and Sun, J. Faster R-CNN: towards real-time object detection with region proposal networks. CoRR, abs/1506.01497, 2015. Disponible en: https://arxiv.org/abs/1506.01497.

[59] Schmidhuber, J. Deep learning in neural networks: An overview. Neural Networks, 61:85 – 117, 2015. Disponible en: https://arxiv.org/pdf/1404.7828.pdf.

[60] Shelhamer, Evan & Long, Jonathon & Darrell, Trevor. Fully Convolutional Networks for Semantic Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence. 39. 1-1. 10.1109/TPAMI.2016.2572683, 2016. Disponible en: https://arxiv.org/pdf/1605.06211.pdf.

[61] Chayan Kathuria. Regression — Why Mean Square Error?, 2019. Disponible en: https://towardsdatascience.com/https-medium-com-chayankathuria-regression-why-mean-square-error-a8cad2a1c96f.

Apéndice A

Instalación de *Frameworks* y Librerías

Ionic

Para lograr instalar Ionic en nuestro sistema, es necesario tener instalado Node.js. Una vez instalado Node, podemos instalar el framework desde la línea de comandos con:

```
$ npm install -g ionic
```

Con esto, finalizamos la instalación del framework que nos permite crear la aplicación móvil. Nos resta ver el framework utilizado para crear la aplicación *backend* que ejecuta el modelo entrenado y devuelve el resultado a la aplicación móvil.

Aplicación Móvil

La aplicación fue probada en un dispositivo con Android versión 4.4.2 con 1GB de RAM y un procesador Qualcomm MSM8926, conectado a la máquina que tiene el código fuente para poder instalarla por consola con los comandos brindados por Ionic.

Instalación

Para poder instalar la aplicación debemos tener conectado el dispositivo a la computadora que tiene el código fuente y disponer de las librerías necesarias para hacer el *deploy*. Para hacer un *deploy* en dispositivos con Android como sistema operativo, debemos contar con 3 dependencias instaladas:

- Java
- Gradle
- Android Studio

En primer lugar debemos contar con el JDK de Java ya que todas las aplicaciones nativas en Android están desarrolladas en ese lenguaje. Notar, que actualmente, Apache Cordova no es compatible con el JDK 11 de Java, por lo tanto

se recomienda utilizar el JDK 8 para un correcto funcionamiento, el mismo lo podemos encontrar en [29].

En segundo lugar, debemos tener el entorno Gradle instalado en nuestro sistema. Esta herramienta se utiliza para compilar aplicaciones Android. Para instalar Gradle podemos seguir la guía de instalación brindada en la página del sitio oficial [28].

Por último, Android Studio es el IDE utilizado para crear aplicaciones nativas de Android. Este, incluye el SDK de Android, que utilizaremos para lanzar las aplicaciones desde la línea de comandos. Otra opción para probar la aplicación sería generar un dispositivo virtual con las herramientas brindadas por el IDE Android Studio; este método no fue probado, por lo que queda a gusto del lector utilizar esta funcionalidad brindada. Para instalar Android Studio podemos seguir la guía de instalación de la página oficial [30].

Una vez instalados todos los entornos necesarios tenemos que exportar ciertas variables globales para que se pueda realizar el lanzamiento de la aplicación hacia el dispositivo. A continuación, podemos ver los comandos que debemos ejecutar para configurar las variables mencionadas.

```
$ export ANDROID_HOME=[ruta-hacia-Android-Sdk]/Android/Sdk
$ export PATH=${PATH}:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
$ export PATH=$PATH:[ruta-hacia-gradle]/opt/gradle/gradle-5.0/bin
```

Ya teniendo todo instalado y configurado, podemos lanzar la aplicación hacia nuestro dispositivo móvil con el comando:

```
$ ionic cordova run android --device
```

Cuando el comando termina de hacer el lanzamiento de la aplicación al dispositivo, en nuestro dispositivo deberíamos poder ver la aplicación corriendo. A continuación veremos los pasos necesarios para poder probar nuestra aplicación móvil con la aplicación de procesamiento.

Testing

Una vez instalada la aplicación en nuestro dispositivo móvil, debemos levantar nuestro servidor de *backend* para que pueda responder a la solicitud hecha por la aplicación cuando se envía una imagen desde la misma. Para lograr esto, se debe ejecutar la aplicación de procesamiento en una IP alcanzable desde la red local, en este caso, 192.168.0.11 en el puerto 8000. Notar, que el código fuente de la aplicación móvil, tiene esta URI como backend por defecto, por lo que deberíamos configurar nuestro servidor de procesamiento para que se ejecute en

esta IP, en caso contrario la aplicación móvil devolvería un error porque no se puede acceder a nuestra API de procesamiento. Para lograr lo anteriormente enunciado, podemos ejecutar desde la ubicación de nuestra aplicación *backend* el comando siguiente:

```
$ python3 manage.py runserver 192.168.0.11:8000
```

Con nuestra aplicación de procesamiento corriendo y nuestra aplicación móvil instalada en nuestro dispositivo, tenemos todo lo necesario para poder probar el funcionamiento del mismo. Podemos seguir los pasos descritos en el apartado 3.4 para realizar todas las pruebas deseadas o consultar la página de ayuda de la aplicación móvil.

Django

Para iniciar un proyecto Django, basta con generar un entorno virtual con virtualenv e instalar con la herramienta pip el framework Django, como se muestra a continuación:

```
$ virtualenv -p python3 mientorno
$ source mientorno/bin/activate
$ pip install Django
```

Con esta secuencia de comando obtendremos el archivo python manage.py que nos permite crear nuestro nuevo proyecto y la aplicación que usaremos como API para punto de entrada de nuestro modelo de procesamiento. Con los siguientes comando lograremos tener una aplicación sencilla corriendo.

```
$ python3 manage.py startproject mi-proyecto nuevo .
$ python3 manage.py startapp \
el-nombre-de-mi-nueva-aplicación
$ python3 manage.py migrate
$ python3 manage.py createsuperuser
$ python3 manage.py runserver
```

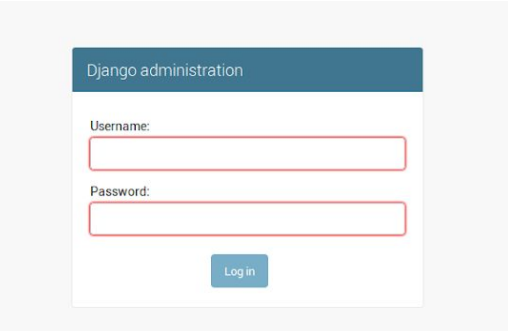

Figura 8.6 Inicio de sesión autogenerado

La primer línea crea el proyecto con nombre mi-proyecto. A continuación creamos una app que va a contener nuestro modelos, vistas y templates. Con el tercer comando generamos el esquema de la base de datos que nos brinda por default Django, ya permitiéndonos generar usuarios, con diversos permisos y grupos, entre otras cosas. Luego generamos un usuario admin para poder probar el sitio autogenerado de administración. Por último, ejecutamos runserver para lanzar el servidor local de prueba.

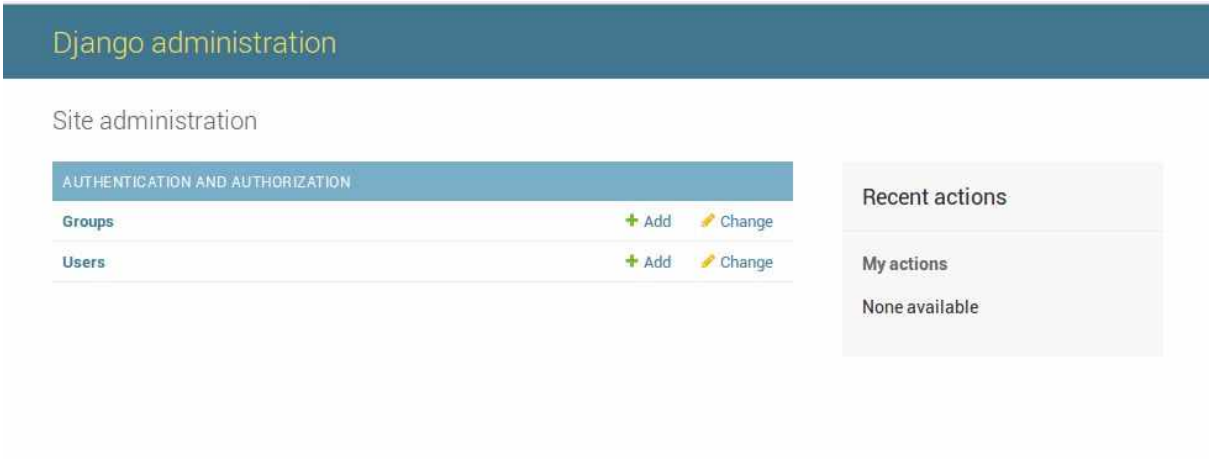

Figura 8.7 Sitio de administración autogenerado

Adicionalmente, utilizamos la librería django-rest-framework, que nos permite generar una API REST de nuestro modelo. Esto lo hacemos para poder recibir las imágenes de nuestra aplicación frontend realizada en Ionic. Para instalar esta librería debemos seguir los siguientes pasos:

```
$ pip install djangorestframework
$ pip install markdown
$ pip install django-filter
```

Para poder utilizar la librería debemos configurar ciertas cosas en nuestro proyecto Django, estas son:

- Agregar la librería djangorestframework a nuestras aplicaciones instaladas en settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'auth_app',
    'rest_framework',
    'rest_framework_simplejwt'
]
```

Figura 8.8 Aplicaciones instaladas en el archivo settings.py

- Debemos generar un archivo serializers.py en nuestra aplicación para poder traducir los objetos de nuestra base de datos a formato JSON.

```
from django.contrib.auth.models import User, Group
from rest_framework import serializers

from auth_app.models import Task


class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'groups')


class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Group
        fields = ('url', 'name',)


class TaskSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Task
        fields = ('owner', 'name', 'body', 'date',)
```

Figura 8.9 Archivo serializers.py

- Generar la vista para que utilice nuestro serializer y especificar el conjunto de datos que va a utilizar.

```
class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer


class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer
```

Figura 9.0 Archivo views.py con la vista genérica de django-rest-framework

- Agregar al archivo urls.py las urls para acceder a las vistas que generamos en el paso previo

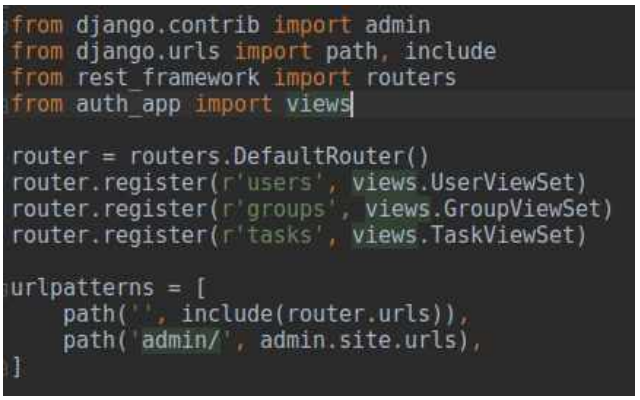

Figura 9.1 Archivo urls.py con las vistas generadas en el paso anterior

Con estos pasos hemos logrado tener una aplicación de muestra que tiene un sitio de administración y una API REST con un sistema de autenticación básico.

Aplicación *Backend*

Instalación

Para instalar nuestro *backend* de procesamiento, debemos seguir unos pasos bastante similares a los del apartado "Django" en el Apéndice del trabajo, salvo que en este caso, en vez de inicializar un proyecto desde cero, una vez descargado el proyecto del repositorio, tenemos que crear un entorno virtual, instalar los requerimientos del archivo "requirements.txt", entrar al entorno y ejecutar las migraciones pertinentes para que la aplicación logre funcionar sin ningún error. A continuación haremos una breve enumeración de los comandos a ejecutar para tener el entorno corriendo.

1. Primero debemos generar un entorno virtual
   `$ virtualenv [nombre-del-entorno]`
2. Segundo, tenemos que activar nuestro entorno virtual para poder instalar todas nuestras dependencias sin alterar la configuración global del sistema
   `$ source [nombre-del-entorno]/bin/activate`
3. Una vez activado el entorno, instalar los requerimientos desde el archivo 'requirements.txt'
   `$ pip install -r requirements.txt`
4. Luego, generar las migraciones necesarias para que la aplicación pueda generar la base de datos
   `$ python3 manage.py makemigrations`
5. Una vez generada las migraciones, migrar la base de datos
   `$ python3 manage.py migrate`

6. Por último debemos ejecutar el servidor local para poder testear la aplicación y corroborar que no haya fallas

```
$ python3 manage.py runserver
```

Testing

Para testear la aplicación Django, podemos utilizar una herramienta como "Postman". Este tipo de herramientas, nos permite generar peticiones para APIs, esto quiere decir que podemos generar una solicitud a una URL que nosotros queramos con los parámetros que necesites, especificando los headers y brindándonos todo lo necesario para configurar nuestra solicitud, como por ejemplo los encabezados de una petición. A continuación, se brinda una imagen ilustrativa de la aplicación Postman

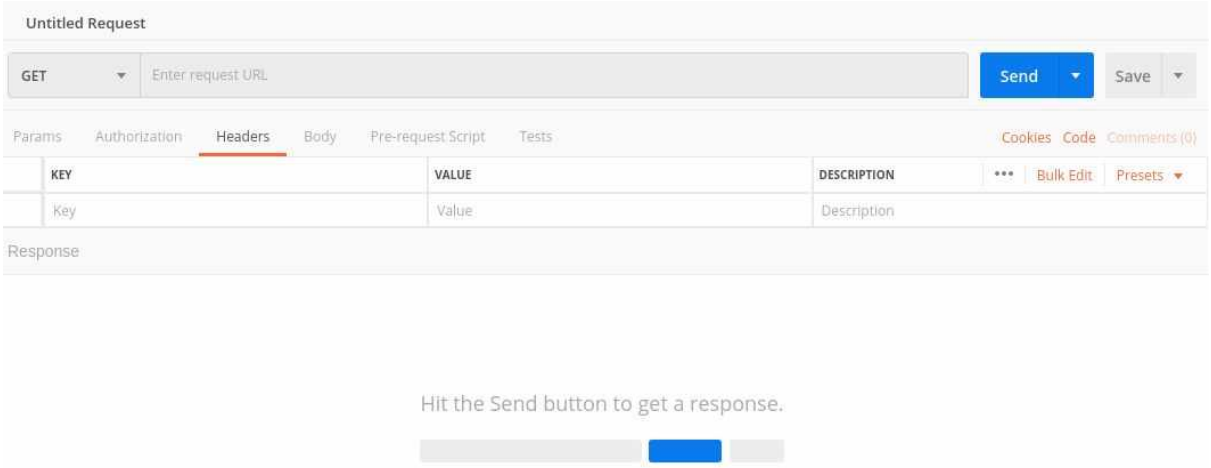

Figura 9.2 Interfaz de Postman

Una vez que tengamos la Postman disponible y el servidor local corriendo, podemos generar peticiones a los distintos puntos de acceso que hayamos creado. Para clasificar una imagen en particular, tenemos que usar una URL del estilo http://[ip-del-servidor]:[puerto]/[ruta-que-solicitamos]. En nuestro caso, el servidor local está ejecutando en una IP privada, para que pueda ser accesada desde cualquier dispositivo de la red, en el puerto 8000, por esto la petición sería *http://192.168.0.11:8000/pictures/upload_docs/*. Además, tenemos que configurar la petición para que envíe una imagen a clasificar y agregar en los encabezados necesarios. En caso de que no ingresáramos la imagen para clasificar obtendríamos una respuesta como la siguiente

```
{
    "detail": "Request has no resource file attached"
}
```

Figura 9.3 Respuesta del servidor en caso de no enviar una foto

Para agregar una imagen a nuestra petición tenemos que especificar el campo "file" dentro del body de la solicitud y en los encabezados, declarar el campo "Content-Type" igual a "application/x-www-form-urlencoded". Una vez configurado esto y hecho la solicitud obtendremos una respuesta que nos brindará a qué categoría corresponde la imagen enviada.

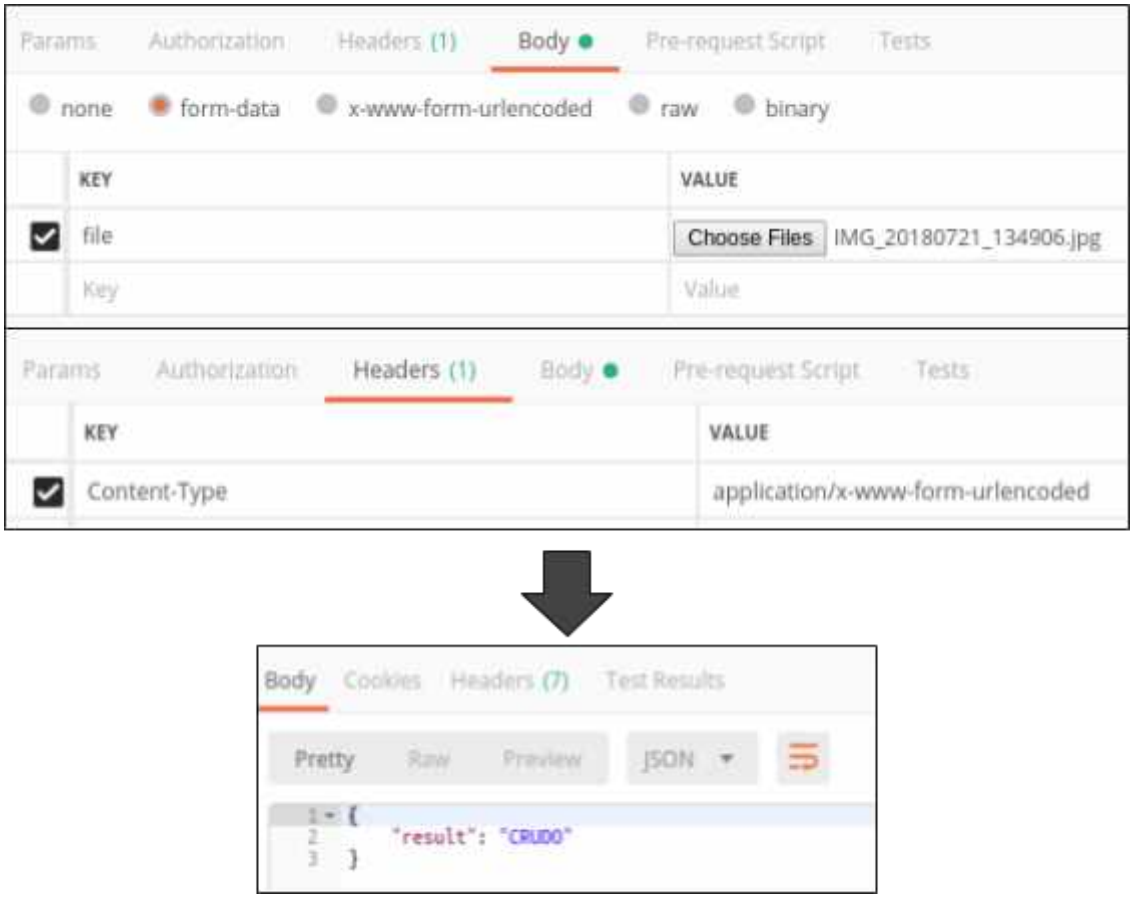

Figura 9.4 Petición y respuesta a la API *backend*

TensorFlow

Para lograr replicar lo hecho en la tesina, se debe tener instalado dicho framework para lograr entrenar los modelos desarrollados con la librería Keras. Para instalar esta librería se puede seguir la guía brindada por el sitio oficial de TensorFlow [12]. A continuación, se estipulan los pasos necesarios:

```
# Descargar python3 y pip si no estan presentes en
# el sistema
$ sudo apt update
```

```
$ sudo apt install python3-dev python3-pip
$ sudo pip3 install -U virtualenv
# Crear virtual environment
$ virtualenv --system-site-packages -p python3 ./venv
# Activar el entorno virtual
$ source ./venv/bin/activate
# Instalar tensorflow en el entorno virtual
$ pip install --upgrade tensorflow
# Verificar la instalación de tensorflow
$ python -c "import tensorflow as tf;\
tf.enable_eager_execution();\
print(tf.reduce_sum(tf.random_normal([1000, 1000])))"
```

Keras

Prerrequisitos de cuDNN

NVIDIA CUDA Deep Neural Network Library (cuDNN) es una librería de primitivas aceleradas por GPU para redes neuronales profundas.

En la documentación oficial de nvidia [20], encontraremos una guía de cómo instalar el sdk de Deep Learning. En este trabajo se siguió la guía de instalación para sistemas Linux.

Para lograr instalar la librería debemos verificar los requerimientos necesarios de cuDNN, estos son:

- Una GPU de capacidad de cómputo 3.0 o superior. Podemos ver si nuestra GPU está dentro de CUDA GPUs.
- Se precisa alguna de las siguientes combinaciones de arquitecturas - SO:
  - En x86_64 (vía archivos debian)- Ubuntu 14.04 or Ubuntu 16.04
  - En x86_64 (vía tgz)- Cualquier distribución de linux
  - En POWER8/POWER9 - RHEL7.4
- Alguno de las siguientes versiones de CUDA y drivers gráficos de NVIDIA:
  - Driver NVIDIA R410 o posterior para CUDA 10.0
  - Driver NVIDIA R390 o posterior para CUDA 9.2
  - Driver NVIDIA R384 o posterior para CUDA 9
  - Driver NVIDIA R375 o posterior para CUDA 8

Instalación de controladores de gráficos NVIDIA

Para instalar los controladores de gráficos NVIDIA en sistemas Linux debemos:

1. Ir a NVIDIA download drivers.

2. Descargar la versión compatible con nuestro sistema desde el sitio anterior.
3. Instalar el controlador descargado como indica en la página previamente mencionada.
4. Reiniciar el sistema para asegurarnos de toma efecto la instalación del controlador.

CUDA

Para instalar correctamente CUDA en nuestro sistema podemos seguir la guía de instalación del sitio oficial de NVIDIA, NVIDIA CUDA Guía de Instalación para Linux.

cuDNN

Una vez instalados los requerimientos de cuDNN, procedemos a descargar la versión que queremos de cuDNN desde el sitio oficial NVIDIA cuDNN. Para iniciar la descarga debemos completar un formulario y aceptar los términos y condiciones.

Para instalar cuDNN tenemos 2 opciones, una es instalarlo vía archivo Tar y el otro modo es vía archivos Debian, los procedimientos son similares y no deberían presentar mayores complicaciones.

En las siguientes secciones se supone que la instalación de CUDA se realizó en el directorio */usr/local/cuda/,* esta ubicación no necesariamente debe ser así, pudiendo modificar el directorio de instalación de CUDA.

Instalación vía TAR

1. Ir al directorio de descarga del cuDNN.
2. Descomprimir el paquete cuDNN.
   ```
   $ tar -xvzf cudnn-9.0-linux-x64-v7.tgz
   ```
3. Copiar los siguientes archivos al directorio del Toolkit de CUDA y cambiar los permisos.

```
$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn.h
/usr/local/cuda/lib64/libcudnn*
```

Instalación vía DEB

1. Ir al directorio de descarga del archivo .deb de cuDNN.
2. Instalar la librería.

```
$ sudo dpkg -i libcudnn7 7.0.3.11-1+cuda9.0 amd64.deb
```

3. Instalar la librería de desarrollador.

```
$ sudo dpkg -i libcudnn7-dev 7.0.3.11-1+cuda9.0 amd64.deb
```

4. Instalar los códigos de ejemplo y el manual de usuario de cuDNN(opcional).

```
$ sudo dpkg - libudnn7-doc 7.0.3.11-1+cuda9.0 amd64.deb
```

Verificación

Para verificar la correcta instalación y funcionamiento de cuDNN, es conveniente compilar el ejemplo mnistCUDNN ubicado en */usr/src/cudnn_samples_v7*.

1. Copiar el ejemplo cuDNN a un directorio que tenga permisos de escritura.

```
$ cp -r /usr/src/cudnn_samples_v7/ $HOME
```

2. Ir al directorio destino

```
$ cd $HOME/cudnn_samples_v7/mnistCUDNN
```

3. Compilar el ejemplo mnistCUDNN

```
$ make clean && make
```

4. Ejecutar el programa de ejemplo

```
$ ./mnistCUDNN
```

Si cuDNN fue instalado correctamente se debería visualizar un mensaje como el siguiente:

```
$ Test passed!
```

En caso de detectar algún error en la instalación se puede dirigir a la sección de "Solución de Problemas" brindada por NVIDIA NVIDIA Developer Forum.

Una vez instalado los requerimientos para poder ejecutar Keras, veremos cómo se instala este último. Podemos instalar la librería de diversas formas dependiendo como es nuestra forma de trabajo. Para instalar Keras globalmente en el sistema podemos utilizar la herramienta pip para localizar la librería e instalarla, como se muestra a continuación.

```
$ sudo pip install keras
```

En caso de que trabajemos con entornos virtuales, lo que es el caso de esta tesina debemos instalar Keras en dicho entorno. Previo al ingreso al entorno debemos ejecutar.

```
$ (venv) pip install keras
```

En caso de querer instalar keras desde el código fuente debemos hacer lo siguiente:

1. Bajar el código desde el repositorio
   $ git clone https://github.com/keras-team/keras.git
2. Ingresar al directorio descargado y ejecutar el comando de instalación.

```
$ cd keras
$ sudo python setup.py install
```