UNIVERSITAT POLITÈCNICA DE VALÈNCIA

APLICACIONES DE GPUs EN VISIÓN E INTELIGENCIA ARTIFICIAL PARA EL RECONOCIMIENTO DE FORMAS

Autor: Javier Giménez Campos

Tutor: Antonio José Cebrián Ferriols

Cotutor: Roberto Agustín Vivó Hernando

Tutor de empresa: Christian Sena Pardo

Trabajo Fin de Máster presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València para la obtención del Título de Máster Universitario en Ingeniería de Sistemas Electrónicos

Curso 2020-21

Valencia, septiembre de 2021

RESUMEN

El presente trabajo toma como referencia el actual campo en plena expansión y futuro inmediato de la aplicación de equipos y sistemas de procesamiento electrónico aplicado al campo de la Visión e Inteligencia Artificial. Para ello se pretende realizar la implementación de varios esquemas de programación y con respecto al uso de uno de los dispositivos más potentes existentes en la actualidad a tal efecto, los actualmente conocidos como GPU siendo su nombre específico, Unidad de Procesamiento Gráfico (Graphics Processing Unit).

En la actualidad numerosos proyectos por parte de importantes empresas tecnológicas se encuentran en la realización de destacados proyectos con importantes implementaciones en todo tipo de disciplinas profesionales como, la automoción, la medicina, servicio público y privado, alimentación, agricultura y un sinfín de campos donde gracias a estos procesadores es posible tal realización y de la cual, en este trabajo se pretende dar una idea técnica de la implementación desde su base más sencilla a este campo de actual y próximo futuro.

Su realización se lleva a cabo conjunto con la empresa de mi actual empleo como Ingeniero de Sistemas electrónicos para varias empresas donde en la actualidad, como se ha indicado, tienen en proyecto la implementación de sistemas como los que se desarrollan en este trabajo y para su servicio. La empresa es, Amara Sonido, Imagen y Comunicación.

El objetivo principal se centra en la implementación de algoritmos básicos y como primera aproximación a esta apasionante tecnología, donde se reúnen primeros lenguajes potentes de programación actuales como puedan ser: C, C++ y Python, junto con el principal dispositivo funcional para poder llevar a cabo con éxito estas implementaciones, basadas en el dispositivo electrónico GPU. Se establecerá una comparativa de estos con su equipo homólogo en computación, CPU. Donde principalmente destacaremos las grandes ventajas del uso de las GPUs frente a las CPUs.

RESUM

El present treballe presa com a referència l'actual camp en plena expansió i futur immediat de l'aplicació d'equips i sistemes de processament electrònic aplicat al camp de la Visió i Intel·ligència Artificial. Per a això es pretén realitzar la implementació de diversos esquemes de programació i respecte a l'ús d'un dels dispositius més potents existents en l'actualitat a aquest efecte, els actualment coneguts com GPU sent el seu nom específic, Unitat de Processament Gràfic (Graphics Processing Unit).

En l'actualitat nombrosos projectes per part d'importants empreses tecnològiques es troben en la realització de destacats projectes amb importants implementacions en tota mena de disciplines professionals com, l'automoció, la medicina, servei públic i privat, alimentació, agricultura i una infinitat de camps on gràcies a aquests processadors és possible tal realització i de la qual, en aquest treball es pretén donar una idea tècnica de la implementació des de la seua base més senzilla a aquest camp d'actual i pròxim futur.

La seua realització es du a terme conjunt amb l'empresa de la meua actual ocupació com a Enginyer de Sistemes electrònics per a diverses empreses on en l'actualitat, com s'ha indicat, tenen en projecte la implementació de sistemes com els que es desenvolupen en aquest treball i per al seu servei. L'empresa és, Estimara So, Imatge i Comunicació.

L'objectiu principal se centra en la implementació d'algorismes bàsics i com primera aproximació a aquesta apassionant tecnologia, on es reuneixen primers llenguatges potents de programació actuals com puguen ser: C, C++ i Python, juntament amb el principal dispositiu funcional per a poder dur a terme amb èxit aquestes implementacions, basades en el dispositiu electrònic GPU. S'establirà una comparativa d'aquests amb el seu equip homòleg en computació, CPU. On principalment destacarem els grans avantatges de l'ús de les GPUs enfront de les CPUs.

ABSTRACT

This work takes as a reference the current field in full expansion and the immediate future of the application of electronic processing equipment and systems applied to the field of Vision and Artificial Intelligence. For this, it is intended to implement several programming schemes and with respect to the use of one of the most powerful devices currently existing for this purpose, those currently known as GPU, its specific name being the Graphics Processing Unit.

Currently, numerous projects by important technology companies are in the realization of outstanding projects with important implementations in all kinds of professional disciplines such as automotive, medicine, public and private service, food, agriculture and endless fields where thanks to these processors such a realization is possible and of which, in this work it is intended to give a technical idea of the implementation from its simplest base to this field of current and near future.

Its realization is carried out jointly with the company of my current job as Electronic Systems Engineer for several companies where currently, as indicated, they have in project the implementation of systems such as those developed in this work and for their service. The company is, Amara Sound, Image and Communication.

The main objective is focused on the implementation of basic algorithms and as a first approach to this exciting technology, where the first powerful current programming languages such as: C, C ++ and Python meet, together with the main functional device to be able to carry out successfully these implementations, based on the electronic device GPU. A comparison of these will be established with their counterpart in computing, CPU. Where we will mainly highlight the great advantages of the use of GPUs over CPUs.

Contenido

MOTIVO, OBJETIVO Y SUMARIO DEL TRABAJO DE FIN DE MÁSTER 7
Motivo 7
Objetivo 7
Sumario 8
Parte I: Conceptos y fundamentos teóricos. 9
1.- Introducción 9
2.- Unidad Procesadora de Gráficos, GPU 13
3.- Aplicaciones de las GPU: Deep Learning, Inteligencia Artificial y Visión Artificial. ... 17
4.- Inteligencia artificial. 25
4.1.- Introducción 25
4.2.- Concepto de neurona. 27
4.3. Perceptrón simple. 29
4.4.- Ejemplo: Evaluación del Perceptrón, función XOR. 31
5.- Aplicación TensorFlow y ejemplo de evaluación del dataset MINST. 36
6.- Evaluación de imágenes en Deep Learning. Aplicación Keras. 39
6.1.- Capas básicas en Keras: 41
6.2.- El dataset CIFAR-10 43
7.- Visión Artificial. Open CV. 50
7.1.- Leer una imagen 50
7.2.- Mostrar una imagen 50
7.3.- Guardar una imagen 51
7.4.- Captura de vídeo desde la cámara 51
7.5.- Reproducción de vídeo desde un archivo 52
7.6.- Guardando un vídeo 53
7.7.- Operaciones Básicas en Imágenes en Open CV con Python 53
7.7.1.- Accediendo y Modificando los valores de píxeles 53
7.7.3.- Accediendo a las Propiedades de Imagen 54
7.7.4.- Dividiendo y Combinando Canales de Imagen 55
7.7.5.- Realización de bordes para la Imagen (*Padding*) 55
7.8.- Operaciones Aritméticas en Imágenes Open CV con Python 56
7.8.1.- Suma de imágenes 56
7.8.2.- Fusión de imágenes 57
7.8.3.- Operaciones Bitwise 57
7.9.- Procesamiento de imágenes. 59

7.9.1.- Cambiando el Espacio de Color Open CV con Python 59
7.9.2.- Transformaciones geométricas de imágenes con Open CV 59
7.9.3.- Redimensionalización 59
7.9.4.- Traslación 60
7.9.5.- Rotación 60
7.9.6.- Transformación Afín 61
7.9.7.- Transformación de Perspectiva 62
7.9.8.- Umbralización en Open CV con Python 63
7.9.8.1.- Umbralización Simple 63
7.9.8.2.- Umbralización Adaptativa 64
7.9.9.- La Binarización de Otsu 66
7.10.- Suavizando Imágenes con Open CV 67
7.10.1.- Convolución 2D (Filtrado de imágenes) 67
7.10.2.- Difuminando imágenes (alisando o suavizando imágenes) 68
7.10.2.1.- Promedio 68
7.10.2.2.- Filtro Gaussiano 69
7.10.2.3.- Filtro de Mediana 69
7.11.- Transformaciones morfológicas 70
7.11.1.- Erosión 70
7.11.2.- Dilatación 71
7.11.3.- Apertura 71
7.11.4.- Cierre 71
7.11.5.- Gradiente Morfológico 72
7.12.- Gradiente de Imágenes 72
7.12.1.- Derivadas Sobel y Scharr 72
7.12.2.- Derivadas Laplacianas 72
7.13.- Algoritmo de Canny 75
7.13.1.- Reducción de ruido 75
7.13.2.- Encontrando el gradiente de intensidad de la imagen 75
7.13.3.- Supresión de falsos máximos 75
7.13.4.- Umbral de histéresis 76
7.13.5.- Canny algoritmo en Open CV 76
7.14.- Contornos 77
7.14.1.- ¿Qué es un contorno? 77
7.14.2.- ¿Cómo dibujar contornos? 78
7.14.3.- Método de aproximación de contornos 78

7.14.4.- Momentos 79
7.14.5.- Área de contorno 79
7.14.6.- Perímetro de contorno 79
7.14.7.- Aproximación de contorno 79
7.14.8.- Envoltura convexa 80
7.14.8.1.- Revisando convexidad 81
7.14.9.- Rectángulo delimitador 81
7.14.9.1- Rectángulo recto 81
7.14.9.2.- Rectángulo rotado 81
7.14.10- Círculo mínimo de inclusión 82
7.14.11.- Ajustando a una elipse 82
7.14.12.- Ajustando a una línea 83
7.14.13.- Propiedades de los contornos 83
7.14.13.1.- Relación de aspecto 83
7.14.13.2.- Extensión 83
7.14.13.3.- Solidez 83
7.14.14.- Diámetro equivalente 84
7.14.15.- Orientación 84
7.14.16.- Máscara y número de píxeles 84
7.14.17.-Valores mínimo y máximo y sus respectivas coordenadas 84
7.14.18.- Color medio o Intensidad media 84
7.14.19.- Puntos extremos 84
7.14.20.- Defectos de convexidad 85
7.14.21.- Prueba de polígono de puntos 86
7.14.22.- Haciendo coincidir formas 87
7.14.23- Jerarquía de contornos 87
7.14.23.1.- Representación de Jerarquías en Open CV 88
7.15.- Histogramas de Open CV 92
7.15.1.- Generar, graficar y analizar histogramas 92
7.15.2.- Generar un histograma 93
7.15.2.1.- Cálculo del histograma en OpenCV 94
7.15.3.- Cálculo del histograma con Numpy 94
7.15.4.- Cómo graficar histogramas 95
7.15.4.1.- Utilizando Matplotlib 95
7.15.4.2.- Utilizando Open CV 96
7.15.5.- Cómo aplicar una máscara 96

7.15.6.- Ecualización de histogramas ... 97
7.15.6.1.- Ecualización de histogramas en Open CV ... 99
7.15.7.- Contraste de ecualización adaptable del histograma (CLAHE por sus siglas en inglés) ... 100
7.15.8.- Histogramas 2D ... 101
7.15.8.1.- Histogramas 2D en Open CV ... 102
7.15.8.2.- Histogramas 2D en Numpy ... 102
7.15.8.3.- Graficando histogramas 2D ... 102
7.15.9.- Retroproyección ... 103
7.15.9.1.- Algoritmo en Numpy ... 104
7.15.9.2.- Retroproyección en Open CV ... 105
7.15.10.- Transformada de Fourier ... 106
7.15.10.1.- Transformada de Fourier en Numpy ... 107
7.15.10.2.- Transformada de Fourier en Open CV ... 109
7.15.10.3.- Optimización del rendimiento de DFT ... 110
7.15.11.- Emparejamiento de plantillas ... 112
7.15.11.1.- Emparejamiento de plantillas en Open CV ... 113
7.15.11.2.- Emparejamiento de plantillas con múltiples objetos ... 115
7.15.12.- Transformada de línea de Hough ... 116
7.15.12.1.- Tranformada de Hough en OpenCV ... 120
7.15.13.- Transformada Probabilística de Hough ... 120
7.15.14.- Transformada de círculo de Hough ... 122
7.15.15.- Segmentación de imágenes con el algoritmo Watershed ... 123
7.15.16.- Extracción interactiva del fondo usando el algoritmo GrabCut ... 127
7.15.16.1.- GrabCut en Open CV ... 129
7.16.- Detección y descripción de características ... 131
7.16.1.- Entendiendo las características ... 131
7.16.2.- Entender las características ... 132
7.16.3.- Características ... 132
7.16.4.- Detección de esquinas Harris ... 132
7.16.4.1.- Detector de esquinas Harris en Open CV ... 133
7.16.4.2.- Esquina con precisión de subpíxeles ... 134
7.16.5.- Detector de Esquina Shi-Tomasi ... 134
7.16.6.- Algoritmo FAST para la detección de esquinas ... 136
7.16.6.1.- Detección de características con FAST ... 136
7.16.6.2.- Detector de características FAST en Open CV ... 136

7.16.7.- ORB (Oriented Fast y Rotativo BRIEF) 137
7.16.7.1.- ORB en Open CV 138
7.16.8.- Feature Matching / Comparación de funciones 139
7.16.8.1.- Conceptos básicos de Brute-Force Matcher 139
7.16.8.2.- Combinación de fuerza bruta con descriptores ORB 140
7.16.8.3.- Combinación de fuerza bruta con descriptores ORB código python: 140
7.17.- Vídeo análisis 141
7.17.1.- Algoritmos Meanshift y Camshift. 141
7.17.2.- Camshift 143
7.17.2.1.- Código Camshift en Open CV 143
7.18.- Flujo óptico 144
7.18.1.- Método Lucas-Kanade 145
7.18.8.1.- Ejemplo Flujo óptico Lucas-Kanade en Open CV 146
7.18.8.2.- Flujo óptico denso Gunner Farneback en Open CV 147
7.18.2.- Substracción de fondo de un vídeo 148
7.18.2.1.- BackgroundSubtractorMOG2 149
7.18.2.2.- BackgroundSubtractorKNN 150
7.19.- Calibración de la cámara y reconstrucción 3D 150
7.19.1.- Calibración de la cámara 150
7.19.2.- Código de Calibración de la cámara 152
7.19.3.- Configuración de Calibración de la cámara 152
7.19.4.- Calibración 154
7.19.5.- No deformación 154
7.19.6.- Error de re-proyección 155
7.19.7.- Pose Estimación 156
7.19.8.- Geometría Epipolar 158
7.19.8.1.- Matriz esencial 159
7.19.8.2.- Código de Geometría Epipolar 159
7.20.- Fotografía computacional y Detección de Facial y Caras 161
7.20.1.- Denoising de imagen 161
7.20.1.1.- Denoising de imágenes en Open CV 162
7.20.2.- Detección de rostros, caras y ojos con Haar Cascades 163
7.20.2.1.- Detección de rostros o caras de Haar-cascade en Open CV 165
Parte II: Realización práctica. Implementación de programación. 166
8.- Ejemplos prácticos. 166
8.1.- Ejemplo práctico 1. Reconocimiento de formas en imagen estática. 167

8.1.1.- Caso de imágenes del juego de Pokemons: 167
8.1.2- Caso de imágenes de animales, clasificación de razas de perros: 182
8.3.- Evaluación de resultados. 195
8.4.- Ejemplo práctico 2. Reconocimiento de formas en imagen dinámica, video. 198
9.- Conclusión y línea futura. 202
Bibliografía: 205

MOTIVO, OBJETIVO Y SUMARIO DEL TRABAJO DE FIN DE MÁSTER

Motivo

El motivo de este trabajo se centra en la necesidad de cubrir y atender la gran demanda tecnológica que actualmente acontece en este campo de la Visión e Inteligencia Artificial y para lo cual se necesita de la realización de proyectos precisos y altamente ajustados a las necesidades de cada demanda. Por nuestra parte, los principales equipos influyentes son los actuales potentes dispositivos conocidos como GPUs y aplicando programación de actuales lenguajes de programación preparados y adaptados a tal fin, lenguajes como C, C++ y Python principalmente.

En conjunto se requiere de dispositivos ordenadores distintos a los actuales implementados mediante CPUs y que no tienen capacidad para la realización de estos proyectos. Sobre todo teniendo en cuenta la principal relación difícil de mantener como es capacidad versus coste económico como principal factor a tener en cuenta por parte de las empresas de consumo de estos sistemas.

Factores como eficacia, eficiencia, precisión, velocidad y adaptabilidad junto con otros sistemas informáticos juegan un rol y papel principal en la realización de estos proyectos frente al factor principal a tener en cuenta como costes, rentabilidad, producción y en general ahorros tanto técnicos como económicos que afecta a la toma de decisión de las principales empresas a la hora de adquirir estos sistemas.

Conjunto a esto anterior indicado, se pretende conseguir una muestra de las ventajas competitivas técnicas de las que favorecen el uso de la GPU frente al uso de CPU. Para ello, la principal referencia de muestra será la adquisición y tratamiento de imágenes como uno de los principales usos y retos de estos sistemas actualmente. Siendo este trabajo un ejemplo principal referente del uso de estos sistemas y que sirva de referencia para otros proyectos de mayor capacidad y complejidad, tanto a nivel de computo técnico como su rentabilidad en coste de tiempo y economía.

Objetivo

Demostrar el uso de la potencia de las Unidades de Procesamiento Gráfico (GPU) conjunto con eficientes algoritmos realizados en los actuales lenguajes de programación indicados y donde sirva de base para la realización de proyectos basados en el futuro campo de la Visión e Inteligencia Artificial.

Para ello, se recurre partiendo de esquemas y algoritmos básicos que ponen en marcha estos dispositivos y a través de la realización de una comparativa de sus equipos competentes, principalmente CPUs, tomar una referencia base y guía para la realización de proyectos de mayor envergadura partiendo de mayor capacidad, complejidad y por lo tanto envergadura.

Se realiza la programación básica, tomando como lenguaje principal de referencia, el lenguaje Python y se programan algoritmos básicos de Machine Learning como uno de los principales componentes constituyentes de la Inteligencia Artificial a través de medios de entorno preparado para tales pruebas como Google Colab, donde se establece el montaje de programación y con fondo el uso que nos reporta Colab respecto a la posibilidad de utilizar tanto CPU como GPU, siendo aquí en principal centro motor de este trabajo. Gracias a Google

Colab y sus aplicaciones dispuestas para la programación, podemos disponer de un entorno ideal de prueba, ensayo, error, corrección y funcionamiento óptimo de nuestro sistema.

Se tomará como principal referencia conjuntos de imágenes, actualmente conocidos basados en conjuntos de datos (datasets), en este caso imágenes y como referencia motora para la evaluación del funcionamiento de una GPU.

Se tiene como principal factor de referencia, el tiempo de ejecución de evaluación y entrega del resultado. Considerando este como uno de los factores clave a la hora de implementar estos sistemas. Esperando obtener grandes ventajas de este factor como principal ventaja frente a la clásica y conocida CPU.

El resultado será la demostración de eficacia y eficiencia de ejecución de nuestro sistema programado. Basado en la identificación de forma respecto de la imagen de referencia. Siendo este considerado por parte de este trabajo como uno de los de mayor uso en la actualidad, como por ejemplo, la identificación facial y otros similares basados en la misma referencia de algoritmo.

Sumario

Este trabajo se compone de dos partes:

Parte I: Conceptos y fundamentos teóricos.

- Introducción a la Unidad de Procesamiento Gráfico (GPU). Aplicaciones.

En esta se describen los dispositivos GPU, su composición, función y aplicaciones. Donde podemos tomar una primera referencia de funcionamiento de este trabajo en las siguientes partes componentes. Uno de los principales fabricantes de GPUs como en Nvidia y su unidad CUDA.

- Introducción a la Inteligencia Artificial. Partes principales componentes.

Conceptos como que es una neurona electrónica y una de sus componentes principales como es el perceptrón simple y como unidad básica de constitución de los módulos funcionales del Deep Learning y la Inteligencia Artificial.

- Evaluación de imágenes en Deep Learning.

Como referencia de aplicación principal en este trabajo se contempla el estudio y análisis de imágenes, tanto estáticas como dinámicas de vídeo como uno de los principales campos de aplicación.

Como aplicaciones componentes de módulos de utilidad y librerías funcionales se trata en módulo Tensorflow y librería de componentes y métodos Keras. A continuación se disponen un par de ejemplos de dos de los datasets de ejemplo más populares: MINST y CIFAR – 10.

  - Módulo Tensorflow y librería Keras.
  - Datasets MINST y CIFAR – 10.

Parte II: Realización práctica. Implementación de programación.

- Identificación de formas.
    - Identificación de formas en imagen.
    - Identificación de formas en cámara. Reconocimiento facial.

Parte I: Conceptos y fundamentos teóricos.

1.- Introducción

En los últimos años se han sucedido grandes avances en tecnología de computadores progresando sobre todo en lo que más nos acontece que es hacia una mayor capacidad de computación respecto del tiempo de ejecución, lo que comúnmente nos referimos como velocidad y respecto capacidad, ambas muy aumentadas hoy en día. A esto, es aquí donde nos destacamos principalmente este trabajo que nos ocupa.

En primera referencia principal hacer referencia a la dependencia hardware – software, en la que nos centraremos principalmente. Partiendo de esquemas de computadores donde operan conjuntamente CPU y GPU junto con otros operativos como grandes y veloces medios de almacenamiento como memorias y discos si bien duros o de estado sólido SSD, [1] en concreto veremos características principales de operación conjunta en el que se destaca en este caso a las GPUs respecto de características como:

- Las GPUs liberan a la CPU de realizar tareas concretas de procesamiento gráfico de manera repetitiva.
- Las GPUs actuales también son procesadores multinúcleo con procesamiento gráfico inherentemente paralelo. Teniendo como paralelismo como forma de computación en la que pueden realizare varios cálculos simultáneamente.

La necesidad de ejecutar múltiples operaciones para procesar cada imagen se satisface mediante muchos threads capaces de ejecutarse en paralelo.

Como podemos observar a nivel descriptivo podemos decir que esta combinación hardware-software se basa en el principio de dividir problemas grandes en varios problemas pequeños, que son posteriormente solucionados en paralelo (divide y vencerás).

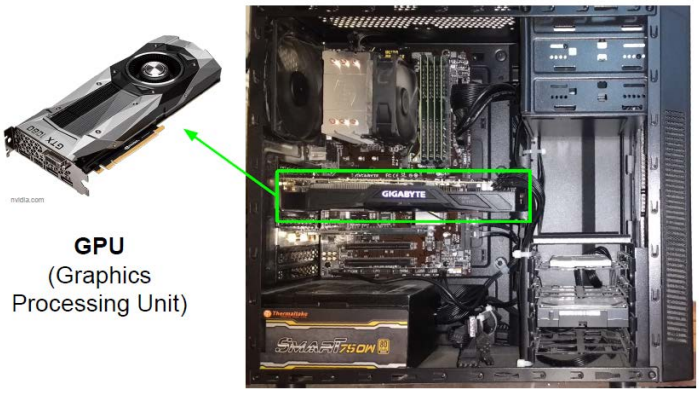


Fig.1: GPU en la actualidad en un ordenador convencional.

Fig.2: Vista gráfica de un procesador GPU.

Fig. 3: Organización de módulos operativos de CPU vs. GPU.

CPU:

- Menos densidad de unidades de cómputo.
- Lógica de control más compleja.
- Optimizado para tareas secuenciales.
- Número de núcleos: de 4 a 12 (Intel y AMD).
- Clock más rápido.

GPU:

- Alta densidad de unidades de cómputo.
- Más cálculos por cada acceso a memoria.
- Diseñado para operaciones en paralelo.
- Número de núcleos: 8 a 512 y actualmente mayores (NVidia).
- Clock más lento.

Una primera comparativa principal lo resumiría la siguiente tabla:

	Núcleos	**Velocidad de reloj**	**Memoria**	**Precio**	**Velocidad**
CPU (Intel Core i7-7700K)	4 (8-hilos con hyperthreading)	4.2 GHz	RAM	339$	~540 GFlops
GPU (Envidia GTX 1080i)	3584	1.6 GHz	GDDR5 11GB	699$	~11.4 TFlops
TPU (Envidia TITAN V)	5120 CUDA, 640 Tensor	1.5 GHz	HBM 2 12 GB	2999 $	~14 TFlops ~112 TFlops
TPU (Google Cloud)	?	?	HBM 64 GB	6.50 $ Hora	~ 180 TFlops

Tabla 1: Comparativa operacional de distintos procesadores.

A continuación una gráfica de evolución y comparativa de coste de GFlops/Dólar.

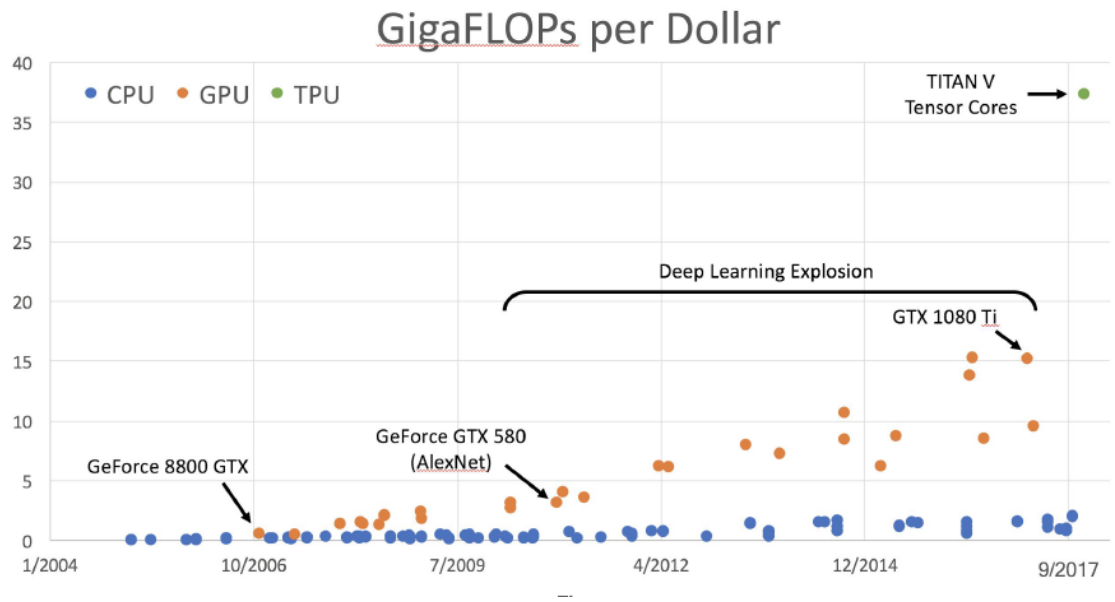


Fig.4: Tabla de evolución de operativa GFlops vs. coste por dólar. Fuente: Curso de Data-Flow, https://mlelarge.github.io/dataflowslides/Slides

Acerca de NVIDIA

NVIDIA es una de las compañías líderes del sector de tecnologías de visualización digital y la inventora de la GPU, un procesador de altas prestaciones que genera gráficos interactivos de gran impacto visual en estaciones de trabajo, ordenadores personales, videoconsolas y dispositivos móviles. NVIDIA ofrece soluciones a numerosos sectores del mercado, lo que incluye el mercado de consumo y del entretenimiento con sus productos GeForce®, los profesionales del diseño gráfico y la visualización de imágenes a través de sus productos NVIDIA Quadro® y el mercado de sistemas computacionales de alto rendimiento mediante sus

productos Tesla™. La compañía tiene su sede en Santa Clara, California, y cuenta con oficinas en Asia, Europa y América.

NVIDIA desarrolla su plataforma de GPUs sobre una plataforma de computación de programación llamada CUDA (Compute Unified Device Architecture), haciendo referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA.

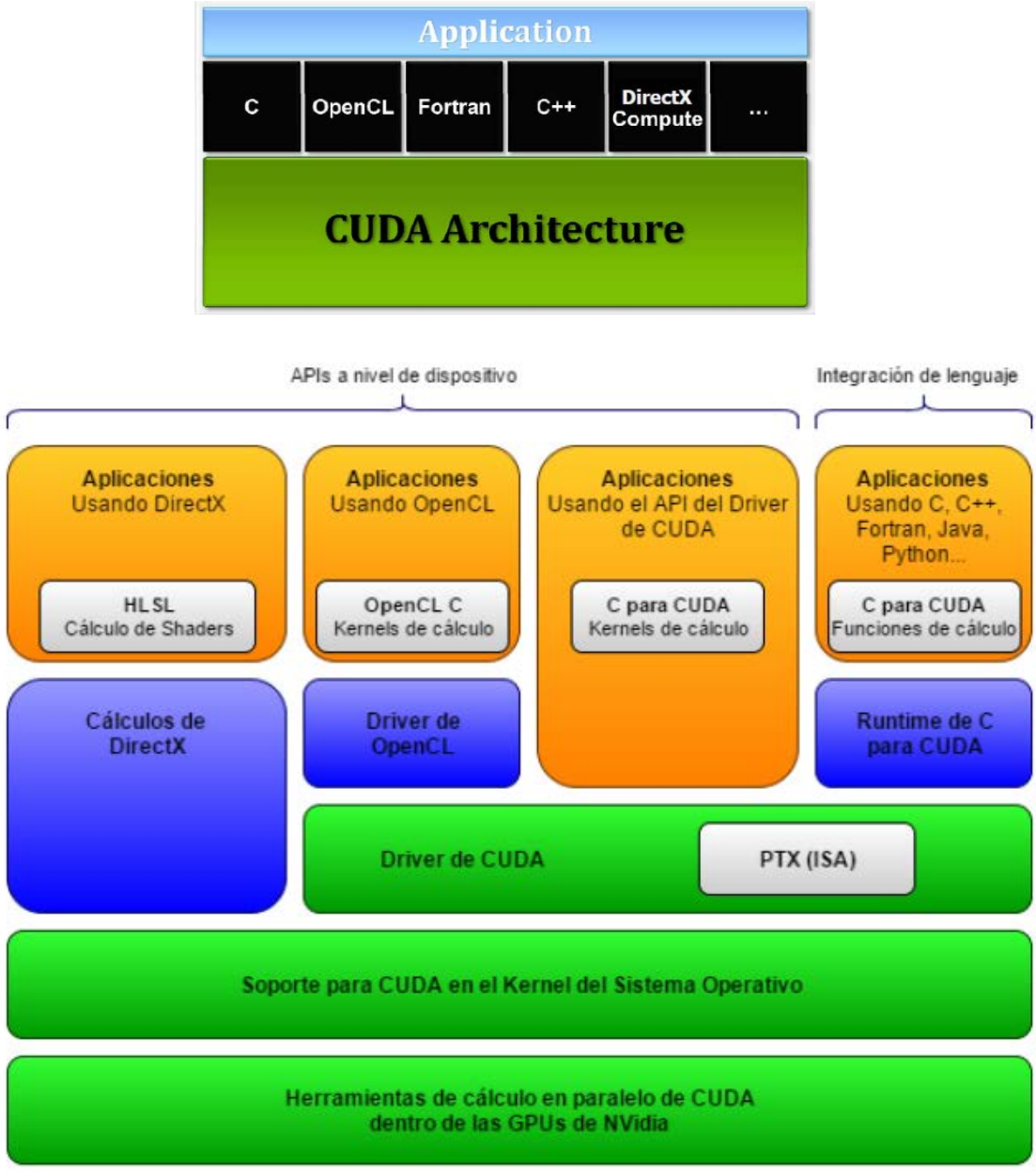


Fig.5: [4][5] Esquemas de Arquitectura CUDA.

Las GPU proporcionan 20 veces más capacidad de cálculo y 10 veces más rendimiento por dólar para la reconstrucción de imágenes y la simulación del flujo sanguíneo,

Algunos referentes comparativos de CPU vs. GPU según Guillaume Alain del Instituto de Algoritmos para el Aprendizaje de Montreal en el que se compara la evaluación de entrenamiento y resultado de analizar 1000 imágenes (a razón media de 100/segundo) es una tabla como la siguiente:

	Entrenamiento	Evaluación de 1000 imágenes.
CPU	2216 horas (92 días)	80 segundos (aprox. 1 minuto)
GPU	28 horas	10 segundos

Tabla 2: [4] Comparativa operacional CPU vs. GPU.

Actualmente el principal sistema procesador lo tenemos en las GPUs CUDA.

- Sólo por Nvidia.
- Es una variante de C que corre directo en la GPU.
- Existen librerías de alto nivel: cuBLAS (Álgebra lineal), cuFFT (Transformada de Fourier), cuDNN (Red Neuronal Profunda).

En el caso concreto que nos ocupa, las GPUs aceleran el entrenamiento de una red neuronal profunda en 60x o más respecto a CPU (dependiendo de la arquitectura/aplicación).

Preguntas a realizar a la hora de elegir una GPU:

- Número de núcleos.
- Memoria de la GPU.
- Ancho de banda de la memoria.
- GFlops / TFlops.

[1] Algunos datos referentes a capacidades de cómputo son:

- Multiplicación de matrices densas: speedup de 9.3x.
- Procesamiento de vídeo (H.264): speedup de 12.23x
- Cálculo de potencial eléctrico: speedup de 64x.
- Resolución de ecuaciones polinomiales: speedup de 205x.

Actualmente NVIDIA ha tenido éxito en la computación GPGPU en la que supercomputadores top utilizan clústers de GPUs para aumentar todavía más su capacidad computacional.

2.- Unidad Procesadora de Gráficos, GPU

Antes de adentrarnos en el proceso operativo de estas unidades procesadoras y en nuestro ejemplo de aplicación al Deep Learning o Inteligencia Artificial, destacar una sección donde vemos la estructura y arquitectura principal de una GPU.

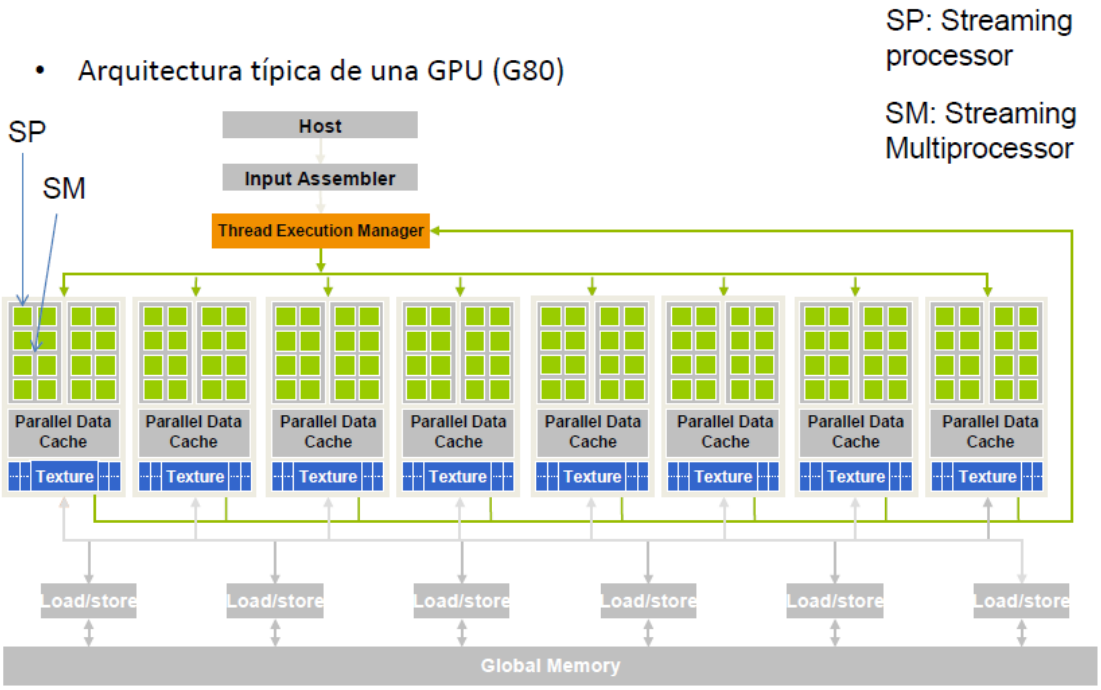
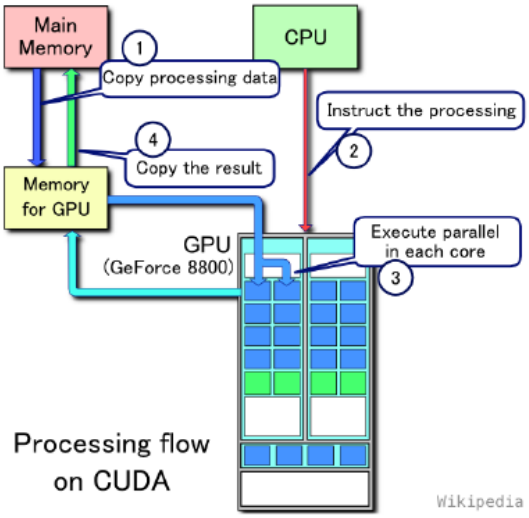


Fig.6: Esquema hardware de una GPU.

Un referente del ciclo operativo como flujo de ejecución de una GPU lo tenemos en la siguiente gráfica:



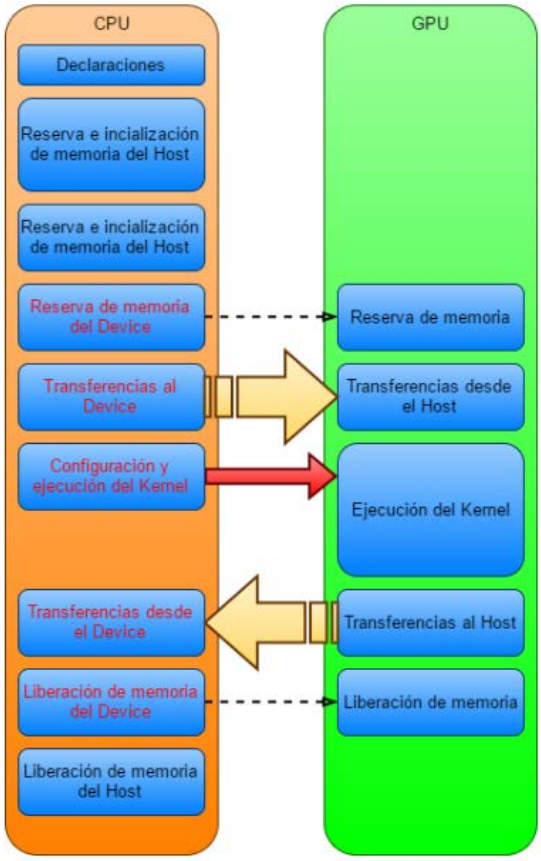


Fig.7: [4][5] Referencias de operativa de GPU con CPU.

Calificadores de funciones

__global__ : función llamada por la CPU para ser ejecutada en la GPU (**kernels**)

__device__: función llamada por la GPU para ser ejecutada en la GPU

__host__ : función llamada por la CPU para ser ejecutada en la CPU

(**__host__**y **__device__**pueden combinarse)

Los pasos principales son:

1.- Inicialización de la GPU.

2.- Reserva de memoria en la parte host y device.

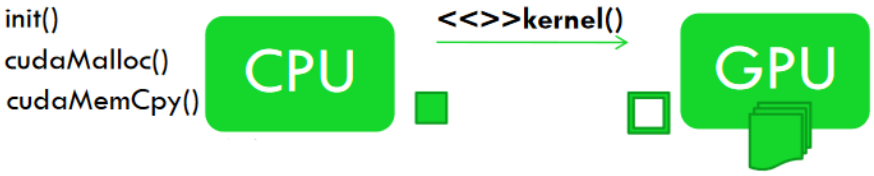
3.- Copia de datos desde el host haca la memoria device.

4.- Lanza la ejecución de múltiples copias del kernel.

5.- Copia los datos desde la memoria device al host.

6.- Se repiten los pasos 3-5 tantas veces como sea necesario.

7.- Se libera la memoria y finaliza la ejecución proceso maestro.



Una GPU puede gestionar miles o incluso millones de *threads* de forma simultánea, pero para facilitarle el trabajo se pide al programador que agrupe los *threads* que va a lanzar en bloques y a su vez los bloques en un *grid* o cuadrícula.

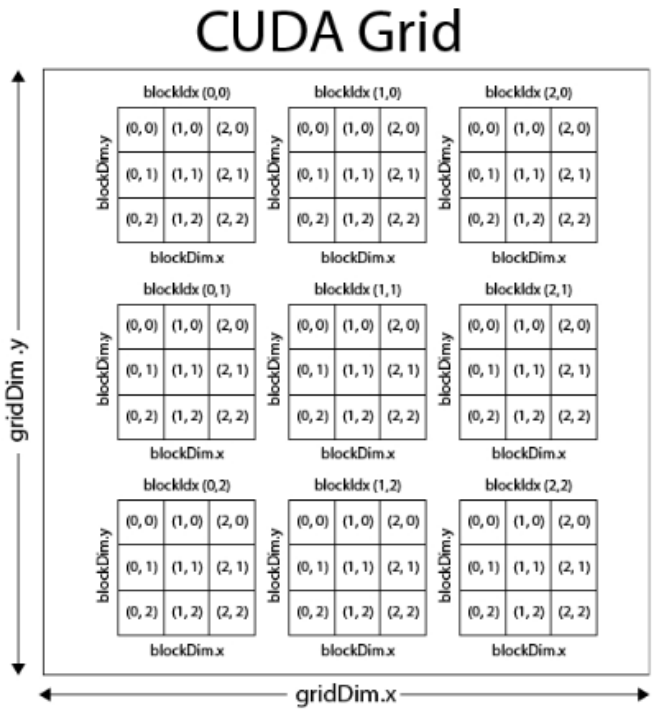


Fig.8: División funcional de estructura de Bloques e Hilos como unidades operativas principales de un GPU.

Con CUDA, dentro del programa se arranca la ejecución de los kernels paralelos mediante la siguiente sintaxis extendida de llamada a función:

kernel <<<dimGrid, dimBlock>>> (...parameters...);

donde dimGrid y dimBlock son parámetros especializados que especifican, respectivamente, la dimensión (en bloques) de la rejilla de procesamiento paralelo y la dimensión (en hilos) de los bloques.

Durante la ejecución del kernel, los hilos tienen acceso a seis tipos de memoria dentro del dispositivo GPU, según la siguiente jerarquía (o niveles de acceso) predefinidos por NVIDIA

Memoria global: Es una memoria de lectura/escritura y se localiza en la tarjeta del GPU.

Memoria para constantes: Es una memoria rápida (cache) de lectura y se localiza en la tarjeta de laGPU.

Memoria para texturas: Es una memoria rápida (cache) de lectura y se localiza en la tarjeta de la GPU.

Memoria local: Es una memoria de lectura/escritura para los hilos y se localiza en la tarjeta del GPU.

Memoria compartida: Es una memoria de lectura/ escritura para los bloques y se localiza dentro del circuito integrado de la GPU.

Memoria de registros: Es la memoria más rápida, de lectura/escritura para los hilos y se localiza dentro del circuito integrado del GPU.

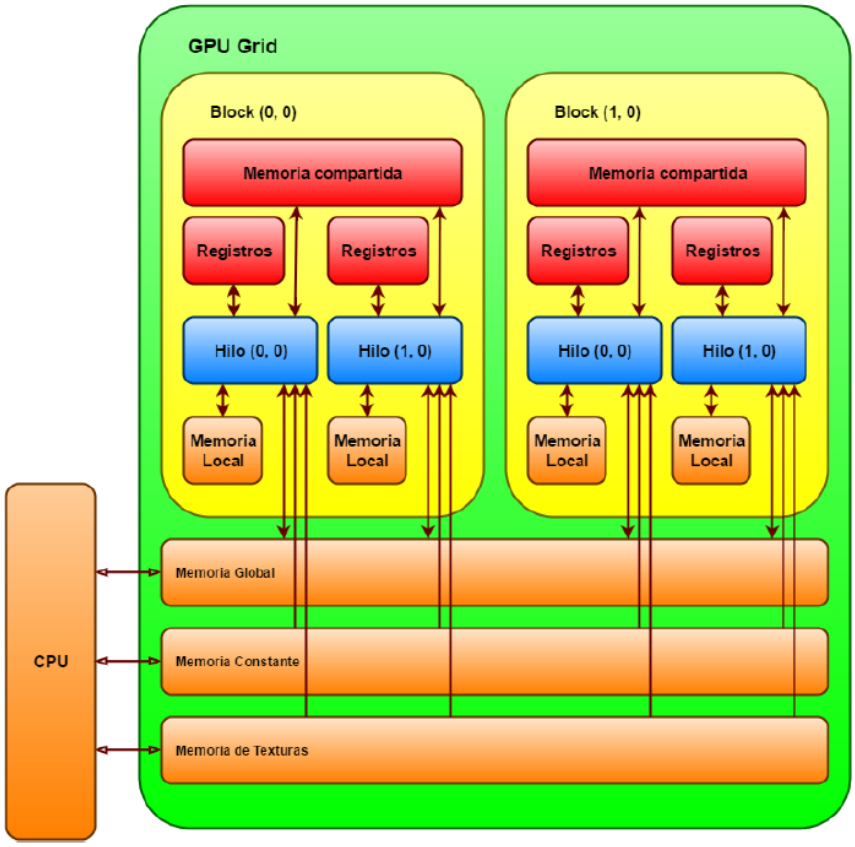


Fig. 9: [5] Esquema del modelo de memoria para cada tipo de memoria utilizado en CUDA.

3.- Aplicaciones de las GPU: Deep Learning, Inteligencia Artificial y Visión Artificial.

Entramos ya en la parte donde exponemos uno de los campos de aplicación más importantes en la actualidad de la operación con GPUs. Una de las aplicaciones actuales de las GPU es aparte de la Inteligencia Artificial, otros campos asociados como la Visión Artificial.

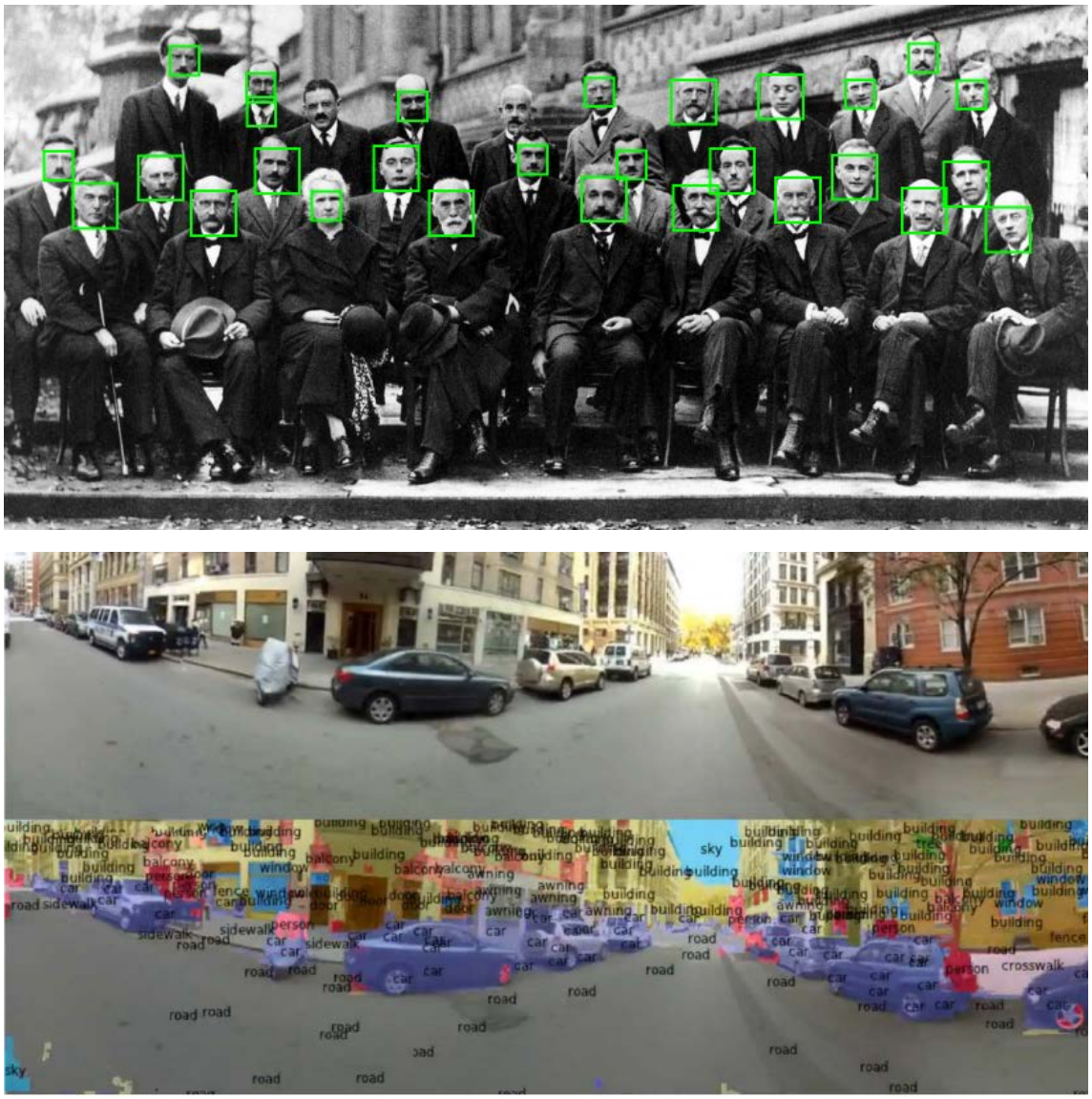


Fig.10: Ejemplos de Visión Artificial.

[5] También se definen valores primeros asociados de aplicación de las GPU como el Aprendizaje Automático como una rama de la Inteligencia Artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender (Wikipedia).

También el Reconocimiento de imágenes, donde tenemos gran variabilidad intra-clase, deformaciones, iluminación, etc, en el que en ocasiones es imposible escribir reglas que describan objetos de forma precisa. Por ejemplo, como vamos a tratar en este trabajo en el que se practica una clasificación de imágenes. Para ello se aplican técnicas de aprendizaje profundo mediante técnicas asociadas de redes convolucionales con red compuesta de capas en cascada:

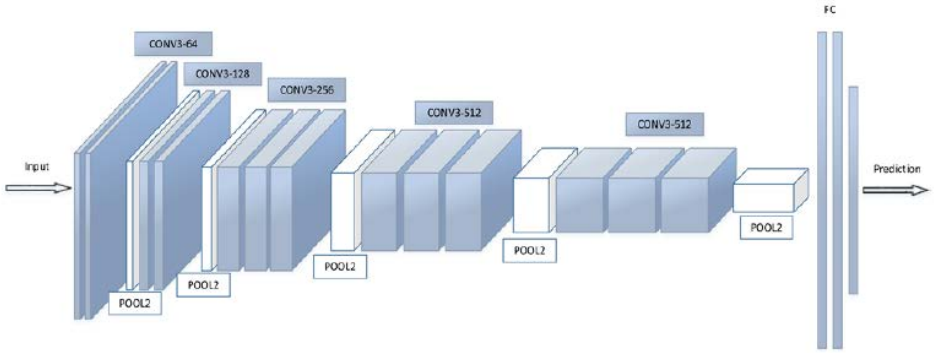


Fig.11: Muestra de Red Convolucional.

En estas redes, en cada capa se extraen y transforman las variables. Cada capa usa la salida de cada capa anterior como su entrada, transformando la representación anterior en otra de mayor nivel de abstracción. Múltiples niveles de representación se corresponden con diferentes niveles de abstracción (jerárquia). Con la ayuda de la alta capacidad de cálculo de las GPU adquirimos representaciones desde una bajo nivel con características más comunes a una de alto nivel con mayor generalidad e invarianza.

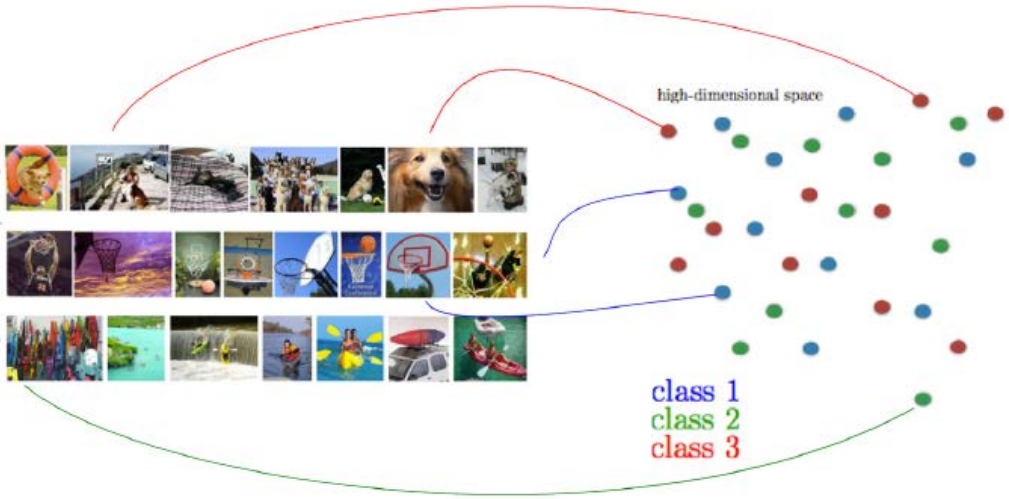


Fig.12: Aplicación de clasificación de imágenes.

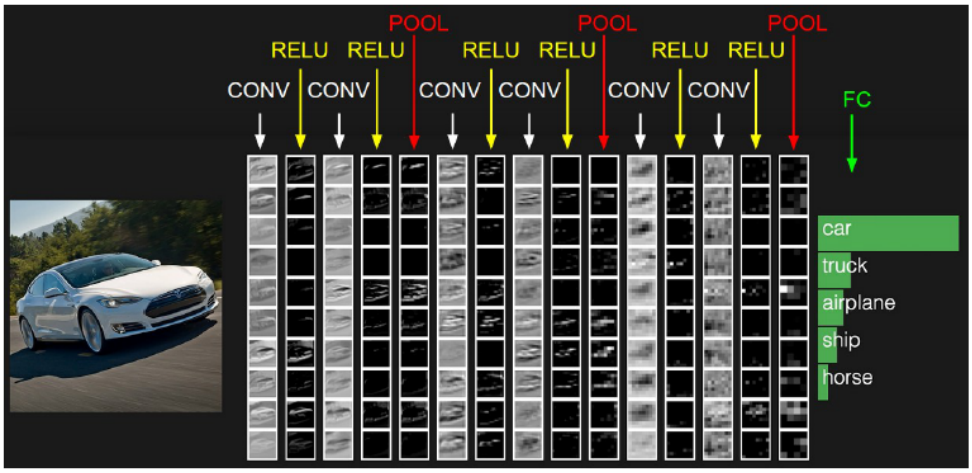


Fig.13: Funciones de activación y clasificación aplicadas a la predicción.

Como veremos, funciones de convolución, pool y de activación como Sigmoide y Relu determinan valores de decisión respecto de la predicción en evaluación.

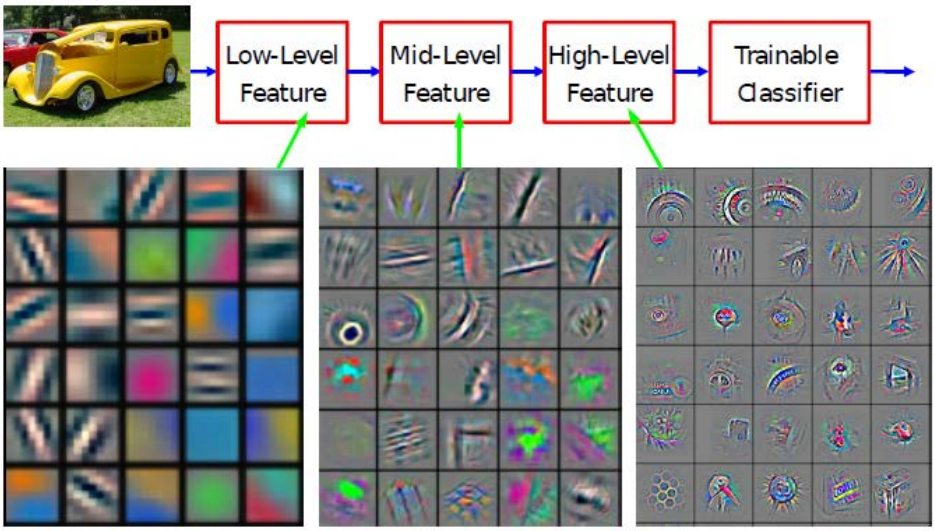


Fig.14: Etapas de evaluación y entrenamiento. Abstracción de características.

Veremos ejemplos en los que realizaremos abstracciones como las siguientes:

- Texto:

Historias → frases → grupos de palabras → palabras → caracteres.

- Imágenes:

Objetos → partes → bordes → pixel.

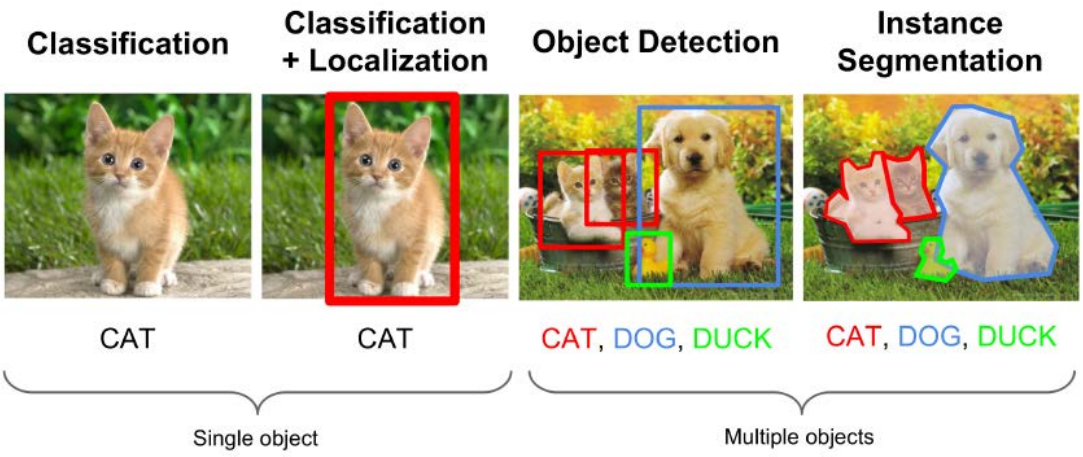


Fig.15: Distintas aplicaciones realizables por Deep Learning gracias a las GPU.

Para el caso concreto que nos ocupa, el tratamiento de imágenes y como función objetivo de la aplicación operativa de las GPU, a continuación exponemos la parte operativa resultado de las operaciones realizadas. El objetivo de aprendizaje automático es la evaluación computacional del resultado de la comparación computacional que se va a procesar en orden numérico y en el que trabajaremos con dos valores principales que son, respecto del resultado de salida, la

obtención y su evaluación de la diferencia entre un valor objeto a llamar (y) y un valor estimado llamado ŷ. Para ello se realizan pruebas con diferentes modelos y parámetros de test. El objetivo es que, llamando e interpretando la diferencia de estos valores como error (respecto de diferencia de valores, llamemos por ejemplo E_{train}) tendremos que:

E_{train} sea muy pequeño (ideal).

E_{train} respecto de E_{test} sean similares.

- Si E_{train} es grande: el modelo subajusta (underfitting), modelo elegido no suficientemente complejo para representar los datos (poca capacidad).
- Si $E_{train} – E_{test}$ es grande: el modelo sobreajusta (overfitting) el conjunto de entrenamiento, (demasiada capacidad).

Indicamos aquí con esto que, tenemos una relación entre los modelos propuesto como veremos (básicos como convolucional y otros compuestos como Keras) donde habremos de realizar ajustes principales como:

- Tasa de entrenamiento (η).
- Número de épocas (epoch).
- Profundidad en capas de la red.

Donde sin un ajuste correcto si bien conforme de estos parámetros podemos realizar una evaluación sub-dimensionada (baja capacidad) y por lo tanto inefectiva o no válida y sobre-dimensionada (capacidad excesiva). Dando lugar a situaciones de no convergencia (under o overfitting) según casos y no obtener un resultado objetivo.

Para ayudarnos de esta referencia realizaremos la obtención de gráficas indicativas como la siguiente donde observaremos nuestra eficacia programada respecto de tener principalmente los anteriores parámetros indicados correctamente regularizados y donde obtengamos una respuesta óptima donde obtengamos una capacidad apropiada con una tasa de error mínima y nuestra línea de error objetivo (verdadero valor) con respecto al entrenado sea el mínimo y se obtengan respuestas paralelas (similares) como se muestra en la siguiente figura.

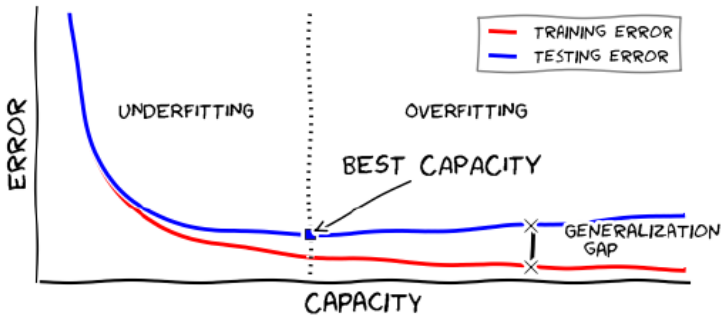


Fig.16: Evaluación de respuesta de una Red Neuronal entrenada.

Será a través de la correcta regularización donde reduzcamos el error de generalización (no el de entrenamiento) y por lo tanto un resultado objetivo correcto, propio y conforme. A continuación un ejemplo de un referente respecto de la elección de la Tasa de aprendizaje y a fin de no caer en una operación de computación apropiada.

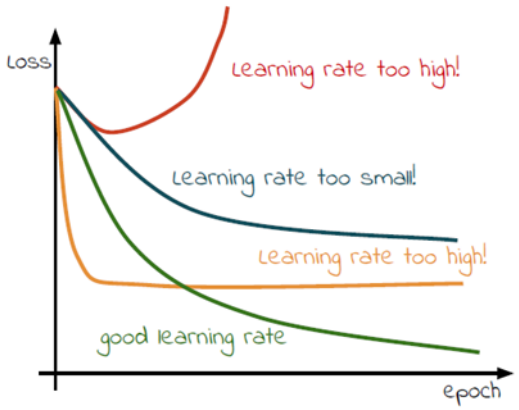


Fig.17: Referencia de valores de Tasa de aprendizaje.

Adelantamos en este punto otros tipos de regularización y que ayudan a una evaluación óptima como:

- Data augmentation: aumento de datos específicos a fin de precisión objetiva.
- Dropout: desconexión de conexiones totales de algunas capas que no aporten información a la prueba o sean irrelevantes.

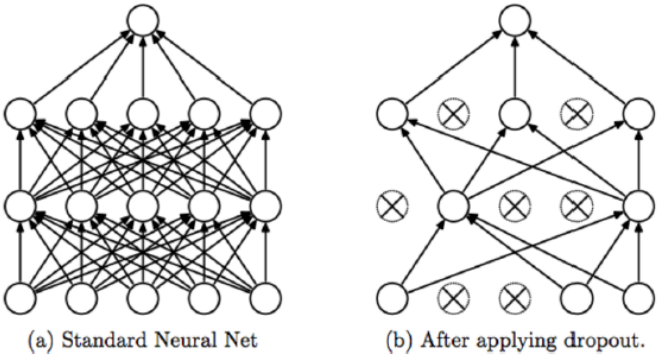


Fig.18: Aplicación de técnica Dropout.

También indicar aquí que para el caso que nos ocupa, haremos uso de apoyo del framework proporcionado por Google TensorFlow, un framework desarrollado por Google y que permite la ejecución de operaciones matemáticas, mediante lo que en este proyecto nos ocupa, diagramas de flujos de datos de forma optimizada en una GPU proporcionada por Google TensorFlow.

Fig.19: Identificativo asociado TensorFlow.

TensorFlow utiliza tensores (contenedores de datos) para realizar las operaciones. En TensorFlow primero se definen las operaciones a realizar (construimos el grafo) y luego se

ejecutan (se ejecuta el grafo). Permite ejecutar el código implementado paralelamente en una o varias GPUs, a elección del usuario.

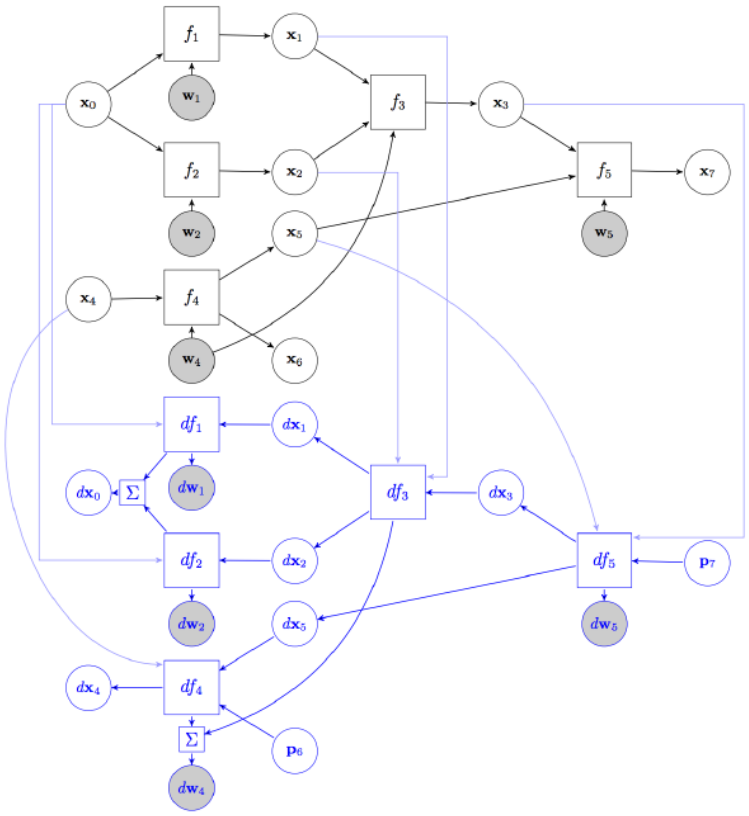


Fig.20: Grafos computacionales.

En el ámbito de la Inteligencia Artificial, los tensores se pueden entender simplemente como contenedores de números, una referencia sería la siguiente:

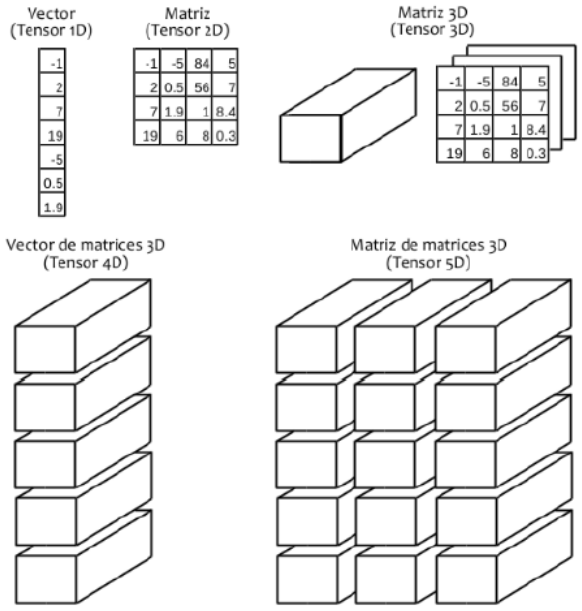


Fig.21: Diversos contenedores de datos en TensorFlow.

- Tensores 3D: utilizados en series temporales.
- Tensores 4D: utilizados con imágenes.
- Tensores 5D: utilizados con video.

Por ejemplo, so queremos almacenar 64 imágenes RGB de 224x224 píxels, necesitaríamos un vector de matrices 3D, o lo que es lo mismo, un tensor 4D. ¿Dimensiones del tensor?: (64,224,224,3).

Aunque no vamos a entrar en mucho detalle en esta parte respecto del trabajo que nos ocupa, a continuación reportamos una muestra de nuestra llamada al framework y a como nos devuelve la aportación de 2 CPUs y 2 GPUs que nos complementarán en nuestros operativos siguientes.

```
[ ]
1 from tensorflow.python.client import device_lib
2
3 def get_available_devices():
4   local_device_protos = device_lib.list_local_devices()
5   return [x.name for x in local_device_protos]
6
7 print(get_available_devices())
```

```
['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
```

En este caso, como vemos Google nos provee además de nuestra CPU, una CPU más y 2 GUPs. XLA (Accelerated Linear Algebra) hace referencia a un compilador optimizado para acelerar operaciones y sobre todo con datos como imágenes.

Podemos ya a continuación pasar a nuestra aplicación principal que aquí nos abarca que es la aplicación de la GPU en el campo concreto del Deep Learning o Inteligencia Artificial. Para el caso que nos ocupa, el lenguaje de programación empleado asociado a poder realizar funciones de Deep Learning es el lenguaje de programación Python.

4.- Inteligencia artificial.

4.1.- Introducción

Definiremos de forma general la Inteligencia Artificial como el conjunto de algoritmos y técnicas que pueden se pueden usar para resolver problemas que los humanos realizamos intuitivamente, pero que son realmente difíciles para un ordenador. Este tipo de procesos van desde el aprendizaje al razonamiento e incluso la autocorrección.

Actualmente se recogen tres técnicas principales dentro de este campo, el Deep Learning, el Machine Learning y la Inteligencia Artificial. Un gráfico resumen de la estructura que lo conforman lo tenemos en la siguiente figura:

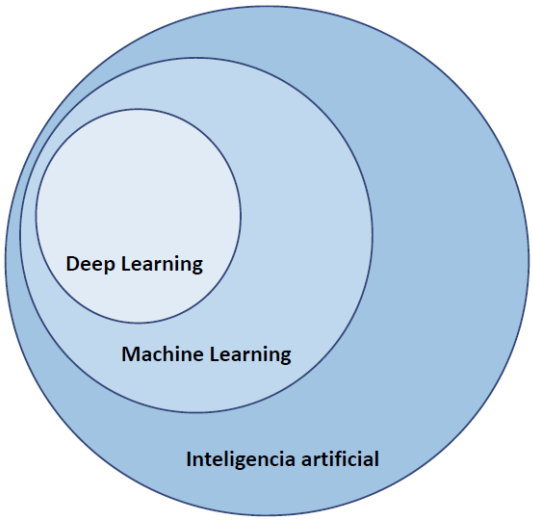


Fig.22: Estructura organizativa del campo de la Inteligencia Artificial.

[26] El proceso referente que siguen las siguientes evaluaciones que vamos a tratar en este trabajo se resumen en la siguiente figura. En ella, partimos de unos datos muestras, a continuación procedemos a la extracción de características y seguidamente pasamos a la clasificación de los mismos.

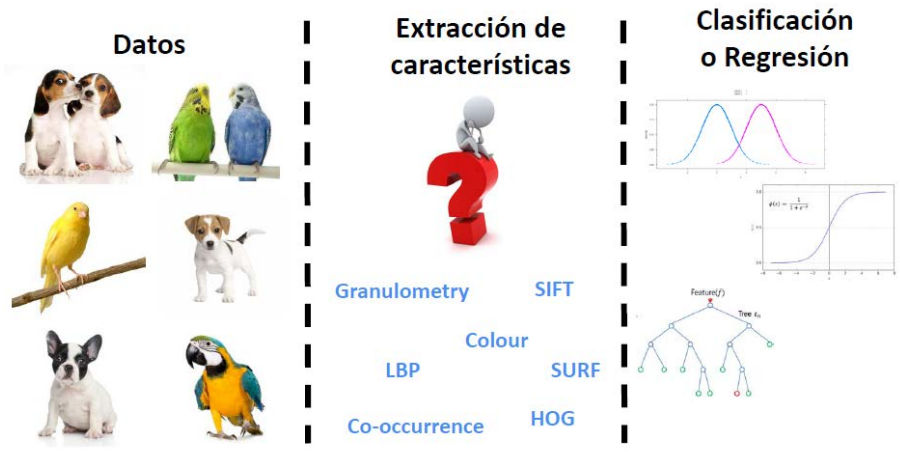


Fig.23: Esquema de proceso de evaluación de un análisis de datos.

En concreto nos basaremos en la principal unidad operativa que nos ocupa y que será la que se lanzará hacia nuestra GPU a computación, la Red Neuronal, partiremos de una red poco profunda, siendo esta la unidad de una única capa hasta varias profundidades (varias capas),

obviamente de menor a mayor carga computacional, a cambio de obtener resultados más óptimos en función de datos más sencillos a más complejos.

En primer lugar, veamos el esquema funcional operacional que compone una red neuronal. Describiendo en primer lugar una **Red Neuronal como un conjunto de capas compuestas por neuronas**. En la figura siguiente vemos esta composición, en la que trataremos principalmente con capas totalmente conectadas o fully connected.

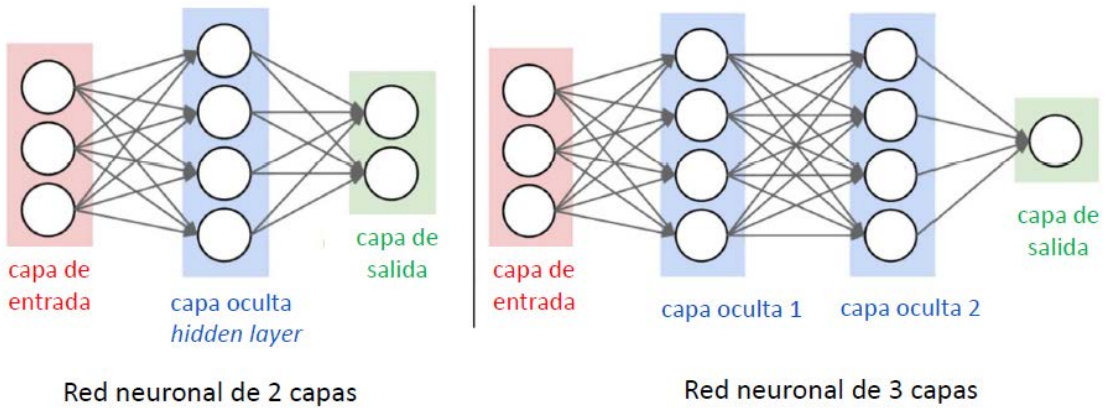


Fig.24: Estructura de Red Neuronal.

A continuación, estructura de una red neuronal profunda.

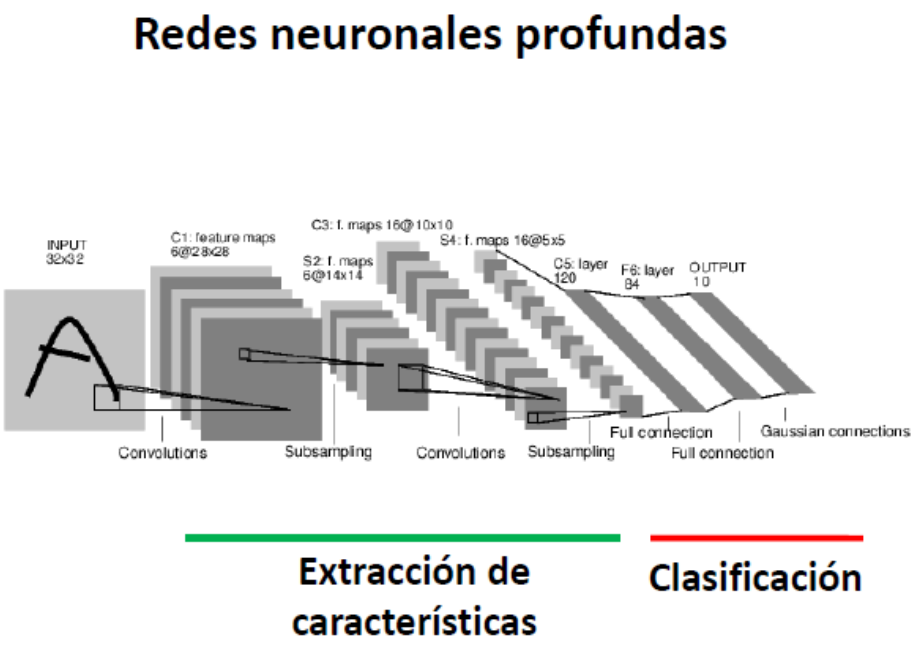


Fig.25: Estructura de una Red neuronal con varias profundidades y unidades operativas.

A medida que avancemos en este trabajo iremos describiendo y evaluando cada componente por separado. En principio y en primer lugar trataremos con la red de una capa y unidad básica operativa, el perceptrón.

4.2.- Concepto de neurona.

Una vez hemos definido la estructura general que compone la estructura inicial base de evaluación de objetos por el proceso de Inteligencia Artificial o por referente Deep Learning, vamos a ir viendo cada uno de los componentes por separado y su inter relación con el resto de componentes que compondrán el operativo de computación que se le encargará a la GPU.

Partimos en primer lugar de la unidad básica operativa: La neurona:

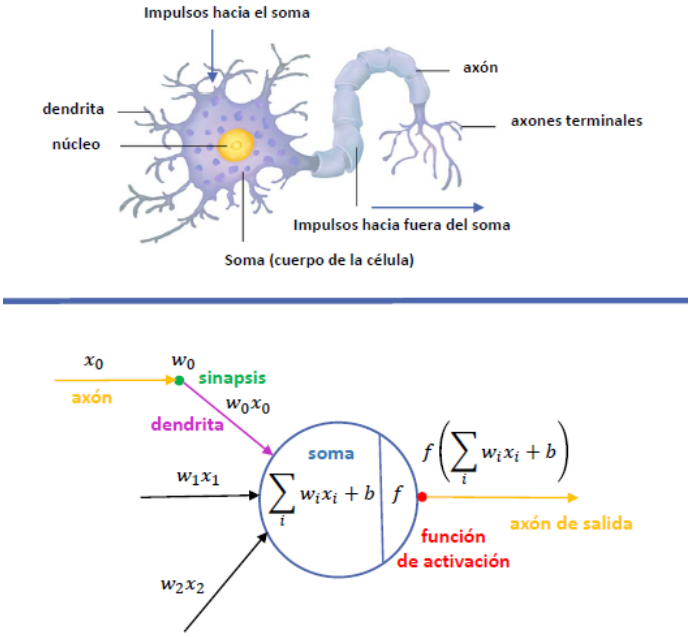


Fig.26: Unidad operativa básica: Neurona.

Aunque de forma general no existe una correspondencia completa del concepto de neurona orgánica con neurona artificial, si que podemos corresponder con su concepto operativo como vemos en la figura anterior. Para ello vamos a describir su parte operativa para nuestra mejor comprensión y con vistas a poner en funcionamiento la unidad operativa básica, perceptrón y del que en siguiente apartado describiremos. Antes, destacaremos aquí en primer lugar la llamada Clasificación Lineal, en ella vemos el proceso llevado a cabo para la extracción de lo que sería la "toma de decisión" del objeto dato a clasificar a través del proceso llamado: de entrenamiento.

La función de clasificación lineal principal es la siguiente:

$$f(W, x_i, b) = Wx_i + b$$

siendo:

f: función objetivo.

x_i: muestras del conjunto de entrenamiento.

W: valor de pesos a contribuir en el ajuste operativo.

b: valor de vías, referente de ajuste.

El proceso funcional se describe en el siguiente esquema:

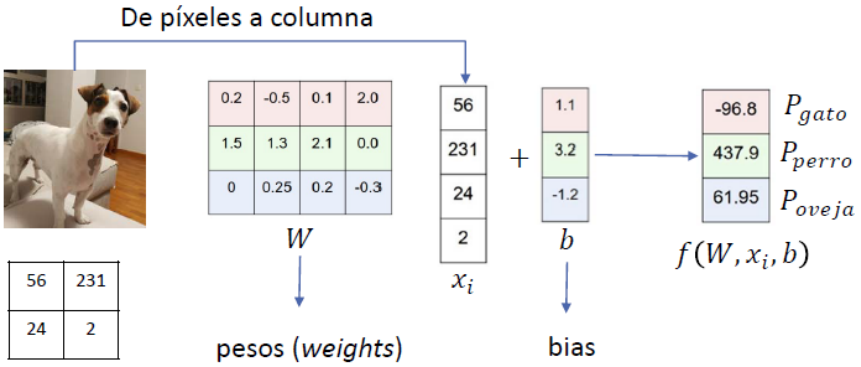


Fig.27: Esquema operativo del Clasificador Lineal.

En este ejemplo, se trata de realizar la evaluación de identificación de la imagen de un perro a partir de la imagen de varios animales (gato y oveja). En este, vemos como la predicción de perro alcanza su mayor peso porcentaje de probabilidad de acierto a través de operar con la función objetivo y con sus valores descritos de x_i, W y b principalmente.

Un esquema ahora más completo los vemos en la siguiente figura, en la que vemos el clasificador asociado con el componente posterior que componen la red neuronal en la que a través de un proceso de realimentación (a continuación en el perceptrón se describe el algoritmo de forward – back propagation) y en el que se procede a través de la realización de una realimentación hacia atrás (back propagation) optimizar los valores de W y b en función la llamada Función de pérdidas (J) en la que se evalúa la el objeto dato real o ticket (y) y el estimado ŷ.

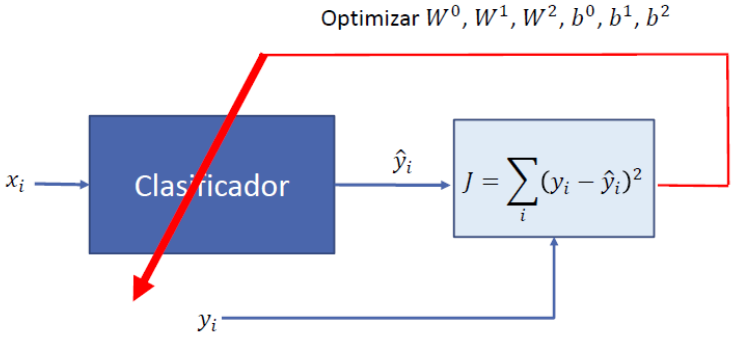


Fig.28: Conjunto operativo Clasificador Lineal – Estimador.

Una manera de ver la carga computacional respecto de este operativo se puede ahora aportar un punto operativo importante, el componente Descenso por Gradiente, en el que a través de un proceso iterativo, se realizan varias operaciones por realimentación a fin de obtener el valor óptimo que nos reportaría idealmente y como es de esperar, un valor de la Función de pérdidas lo más mínima posible si bien el ideal sería función de pérdidas igual a cero en el que el objeto estimado sea igual al objeto de referencia (ticket). A continuación mostramos una de las partes computacionales centrales, el cálculo del Descenso por Gradiente:

$$w_0(t+1) = w_0(t) - \eta \frac{\partial J}{\partial w_0}$$
$$w_1(t+1) = w_1(t) - \eta \frac{\partial J}{\partial w_1}$$

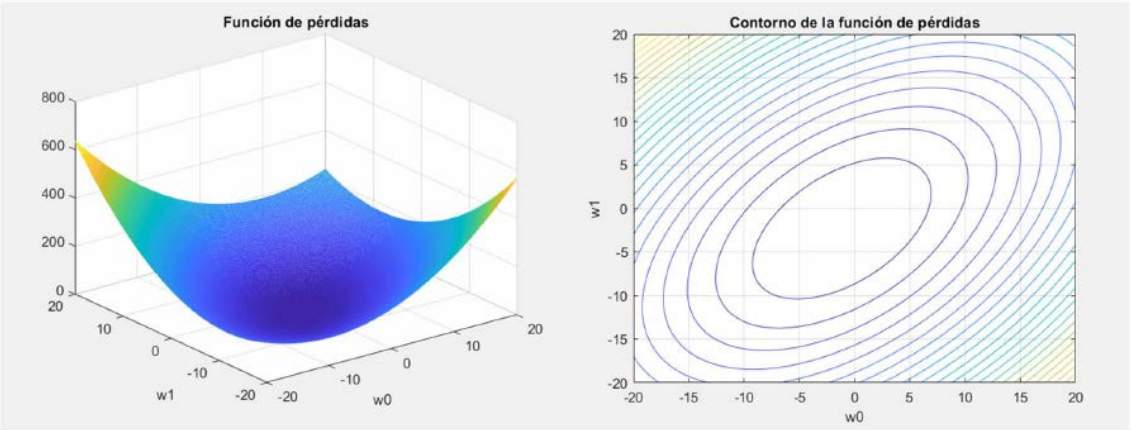


Fig.29: Algoritmo computacional: Descenso por gradiente.

En este, a través del parámetro tasa de aprendizaje η (learning rate), contribuirá a la optimización del cálculo realizado en contribución a la optimización de los valores W y b con los que a través de la función de clasificación obtendremos nuestro cálculo objetivo esperado. En este caso, la predicción de identificación de una imagen respecto de identificar un perro entre varios animales.

Llegados a este punto, pasamos a la siguiente estructura operativa y en la que ya vamos a poder contar con un motor operativo computacional principal, el perceptrón.

4.3. Perceptrón simple.

Definidos los principales componentes que intervienen en este operativo computacional, pasamos a verlo desde su unidad operativa principal, el perceptrón simple.

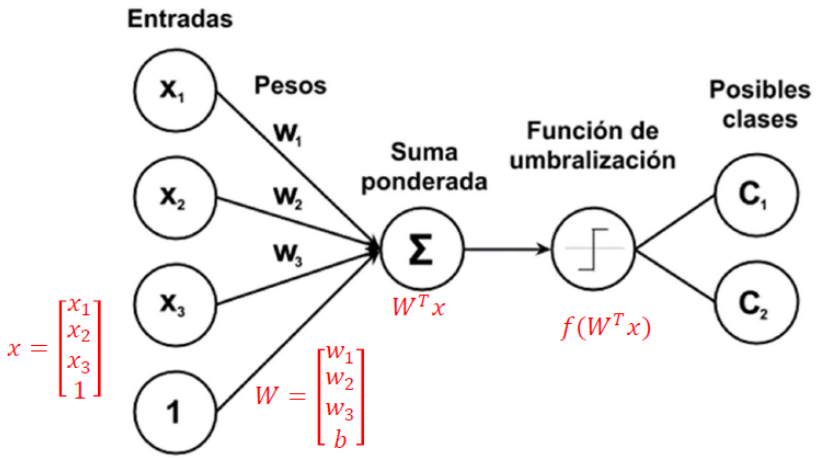


Fig.30: Esquema operativo del Perceptrón simple.

Vemos en esta figura anterior los componentes principales del mismo y que van a constituir la unidad básica de carga computacional de la GPU y en la que posteriormente veremos un ejemplo de programación a través del lenguaje de programación Python.

A modo de resumen, se puede describir el proceso de ejecución del perceptrón a través de los siguientes pasos:

1.- Inicializar el vector de pesos W con números aleatorios pequeños (distintos de cero).

2.- Para cada dato (para cada imagen en este caso):

Cálculo de la salida

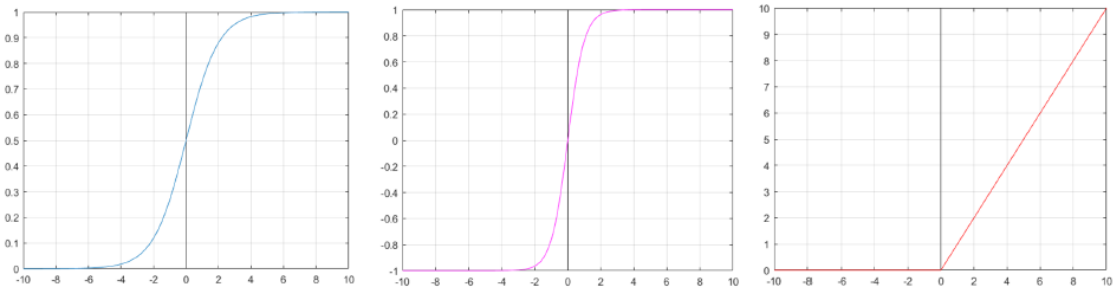
- $y = f(W^T x)$ *# f es umbralización* $W^T x \geq 0 \;\; y = 1$ *si no* $y = 0$

Actualización de los pesos

- $W(t+1) = W(t) + \eta * (d - y) * x$

Cuando todos los datos del conjunto entrenamiento pasa una vez por el algoritmo se dice que ha pasado una **Época**.

Entra también en funcionamiento aquí la llamada Función de activación, en la que a través de una función matemática apropiada, asociamos la decisión del valor estimado ŷ con valores umbrales característicos, siendo estas funciones las más conocidas, la función Sigmoide, Tangente Hiperbólica y Relu.



Sigmoide

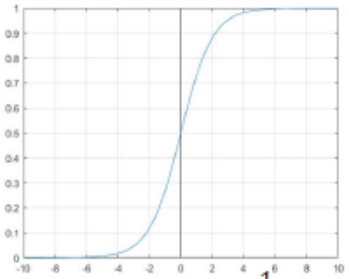
$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Tangente hiperbólica

$$f(x) = tanh(x) = 2\sigma(2x) - 1$$

Relu

$$f(x) = \max(0, x)$$



Sigmoide

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Softmax

$$f(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Fig.31: Funciones de activación.

Posteriormente, en el ejemplo programado se describe a mejor detalle cada una de estas funciones.

Ahora ya podemos tener una vista más completa de nuestro sistema computacional en el que va a ser la GPU la única unidad operativa.

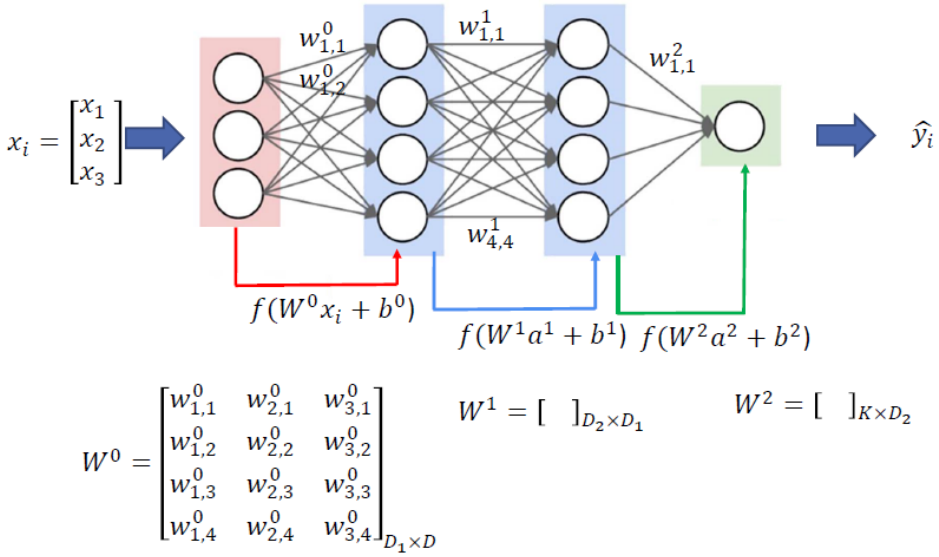


Fig.32: Estructura de la unidad operativa Perceptrón.

Antes de comenzar con nuestro ejemplo programado, indicar en este punto que, nuestra unidad operativa, funcionará a través de la evaluación llamada en Inteligencia Artificial, Algoritmo Forward & Backpropagation, en el que como su nombre nos intuye, a través varios procesos, Forward (hacia adelante) y Backward (hacia atrás) alcanzaremos nuestra realización de estimación objetiva de nuestro dato de evaluación objetiva.

A continuación y a fin de ir realizando este aprendizaje por nuestra parte y en orden de complejidad, pasamos en primer lugar a comentar de forma resumida el operativo a través de un sencillo ejemplo del Perceptrón, donde posteriormente, pasaremos a ver un par de ejemplos más completos y constituyentes de este trabajo aquí realizado: aplicaciones del uso de la GPU en Inteligencia Artificial, donde por varios autores referentes principales califican como mejor hardware el uso de GPU.

4.4.- Ejemplo: Evaluación del Perceptrón, función XOR.

Pasamos ya en este apartado a ver un ejemplo de como se realiza la programación del perceptrón y a esto, a ver su funcionamiento a través de un primer ejemplo en este caso sencillo y en orden de complejidad y donde finalmente pasaremos a ver un ejemplo más completo de evaluación de una imagen como objetivo de nuestro interés que aquí nos ocupa.

Esta simple red conecta nuestro vector de entrada (x_j) con el nodo de salida por medio de ciertas conexiones caracterizadas por unos pesos en un instante dado (W(t)) junto con el término bias (b). La predicción de salida por lo tanto se puede obtener como $p_j=w(t)*x_j$. El objetivo de la fase de entrenamiento de una red neuronal es optimizar los valores de los pesos W con el objetivo de minimizar el error a la salida entre la predicción (p_j) y el verdadero valor (ground truth) (d_j). Para ello, cuando una muestra de entrenamiento pase por nuestra red, se obtendrá el error (en función de una precisión, accuracy) como $e = d_j - p_j$ y se actualizarán los pesos según la ecuación $W_i(t+1) = w_i(t) - \eta(d_j-p_j)x_{j,i}$ para todas las características $0 \leq i \leq n$.

A continuación el código de la clase preparada en Python a la que llamaremos **Perceptrón.** Clase compuesta de los siguientes métodos:

__init__: constructor de la clase. Método con el que instanciaremos la clase. Recibirá como parámetros den entrada el número de entradas al perceptrón (N) y la tasa de aprendizaje η.

Tomaremos como valor por defecto 0.1. Este método también inicializa aleatoriamente los pesos W siguiendo una distribución normal (Gaussiana) de media cero y varianza unidad.

theresholding: método que aplicará un umbral de predicciones para convertirlas a un valor binario. Si $p_j(x) > 0$ entonces el valor predecido $\hat{y}=1$, en caso contrario $\hat{y}=0$.

fit: método que será el encargado de "ajustar" los datos al modelo, es decir, se encargará de la fase de entrenamiento. Para ello, se recorre un número de épocas dado y para cada época (por defecto epoch=15) se calcula el producto de la entrada por los pesos (para todas las conexiones de la red) y se aplica la función umbral. Posteriormente se calcula el error y se actualizan los pesos.

predict: método que se encarga de realizar la predicción de nuevas muestras. El método devuelve la multipliación de la entrada por los pesos (obtenidos en la fase en entrenamiento) habiendo aplicado a dicha operación la función umbral.

```
import numpy as np #importamos paquete cientifico para operaciones matematicas.

class Perceptron:

  def __init__(self, N, eta=0.1):
    # Inicializar la matriz de pesos y almacenar la tasa de aprendizaje
    self.W = np.random.randn(N + 1) / np.sqrt(N) #(X)
    self.eta = eta #(X)

  def thresholding(self, x):
    # Aplicar a función umbral
    return 1 if x>0 else 0 #(X)

  def fit(self, X, y, epochs=15):
    # Añadir bias en la última columna de la matriz
    X = np.c_[X, np.ones((X.shape[0]))]
    # Recorremos el número de épocas pre-establecido
    for epoch in np.arange(0, epochs):
      # Para cada uno de los puntos del dataset
      for (x, target) in zip(X, y):
        # Producto de la entrada por los pesos de cada una de las conexiones
        # (producto matricial, i.e. dot en numpy) y aplicar la función umbral
        p = self.thresholding(np.dot(x, self.W)) #(X)
        # Actualizar pesos en caso de que la predicción y el ground truth sean distintos
        if p != target:
          # Calculo del error
          error = p - target #(X)
          # Actualización de pesos
          self.W += -self.eta * error * x #(X)

  def predict(self, X, addBias=True):
    # Aseguramos que la entrada es una matriz
    X = np.atleast_2d(X)
    # Añadimos el bias a las muestras de test si es necesario
    if addBias:
      X = np.c_[X, np.ones((X.shape[0]))]
    # Devolevmos la etiqueta con la predicción
    return self.thresholding(np.dot(X, self.W)) #(X)
```

A continuación procedemos a su entrenamiento:

```
import numpy as np
# Construimos el dataset XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) #(X)
y = np.array([[0], [1], [1], [0]]) #(X)

# Definimos nuestro perceptrón y lo entrenamos
print("[INFO]: Training perceptron with the XOR dataset...")
p = Perceptron(X.shape[1], eta=0.1) #(X)
p.fit(X, y, epochs=20) #(X)

# Evaluemos nuestro preceptrón
print("[INFO]: Testing perceptron with the XOR dataset...")
for (x, target) in zip(X, y):
  pred = p.predict(x) #(X)
  print("[INFO]: data={}, ground-truth={}, pred={}".format(x, target[0], pred))
```

El resultado:

```
[INFO]: Training perceptron with the XOR dataset...
[INFO]: Testing perceptron with the XOR dataset...
[INFO]: data=[0 0], ground-truth=0, pred=1
[INFO]: data=[0 1], ground-truth=1, pred=1
[INFO]: data=[1 0], ground-truth=1, pred=0
[INFO]: data=[1 1], ground-truth=0, pred=0
```

A mejor función lo reportamos desde una evaluación de red neuronal multicapa y con la implementación del algoritmo de propagación hacia delante (forward) ya retropropagación (backpropagation). Se genera una nueva clase NeuralNetwork y se añaden nuevos métodos como:

El método **fit_partial** se encarga del proceso de entrenamiento. En este método, para este caso de red multicapa, se distinguen tres fases:

1.- Propagación hacia delante (Forward propagation): producto de la entrada de los pesos y función de activación para cada una de las capas de la arquitectura (almacenada en una lista).

2.- Propagación del error haca detrás (Backpropagation): cálculo del error a la salida y obtención del error en cada capa aplicando la regla de la cadena. Se debe recorrer las capas de atrás hacia delante e ir calculando el error asociado a cada capa.

3.- Actualización de los pesos (weights update): actualizar los pesos de cada capa.

sigmoid: contiene la ecuación de la función de activación sigmiode.

sigmoid_deriv: contiene la derivada de la función de activación empleada en la parte del algoritmo backpropagation.

calculate_loss: cálculo del error cuadrático medio a partir de las etiquetas y las predicciones.

predict: realizará la predicción de nuevas muestras. El método devuelve la multiplicación de la entrada por los pesos de cada capa (obtenidos en la fase de entrenamiento) y aplica la función de activación.

```python
# Importamos la única librería con la que desarrollaremos nuestra primera NN
import numpy as np
# Una librería más solo para propósitos de visualización
import matplotlib.pyplot as plt

class NeuralNetwork:
  def __init__(self, layers, eta=0.1):
    # Inicialicemos una lista de pesos y almacenemos la arquitectura de red y lr
    self.W = []
    self.layers = layers # Lista de enteros, ejem [2,2,1] significa que tenemos dos entradas, una capa oculta de dos nodos y una neur
    self.eta = eta # Tasa de aprendizaje
    # Inicializando pesos de las capas
    for i in np.arange(0, len(layers) - 2):
      w = np.random.randn(layers[i] + 1, layers[i+1] + 1) # Matriz M x N (salida capa i/entrada capa i+1) de pesos siguiendo distr. g
      self.W.append(w / np.sqrt(layers[i])) # Almacenamos pesos escalados
    w = np.random.randn(layers[-2] + 1, layers[-1]) # Inicialización última capa #(X)
    self.W.append(w / np.sqrt(layers[-2]))

  def __repr__(self):
    # Devuelve un string con la arquitectura de la red
    return "NeuralNetwork: {}".format("-".join(str(l) for l in self.layers))

  def sigmoid(self, x):
    return 1.0 / (1 + np.exp(-x)) #(X)

  def sigmoid_deriv(self, x):
    # La derivada de la función sigmoide f(x) es f'(x) = f(x) * (1 - f(x))
    # Supondremos que aquí ya entra f(x), es decir, que nos entra la salida de la sigmoide
    return x * (1 - x) #(X)
```

```
def fit(self, X, y, epochs=1000, displayUpdate=100):
  # Bias trick: Concatenamos los bias para que sean parámetros entrenables de la red
  X = np.c_[X, np.ones((X.shape[0]))]
  # Lista para almacenar pérdidas para hacer un plot Loss vs epochs
  my_losses = []
  # Recorremos épocas
  for ep in np.arange(0, epochs):
    # Recorremos datos de entrada y entrenamos red
    for (x, target) in zip(X, y):
      self.fit_partial(x, target)
    # Calculamos pérdidas de todos los datos de entrenamiento
    loss = self.calculate_loss(X,y)
    my_losses.append(loss)
    # Muestro una de cada displayUpdate muestras
    if ep == 0 or (ep + 1) % displayUpdate == 0:
      print("[INFO]: epoch={}, loss={:.7f}".format(ep + 1, loss))
  # Visualización de la curva Loss vs Epochs
  plt.plot(np.arange(0, epochs), my_losses, 'r')
  plt.ylabel('Loss')
  plt.xlabel('Epoch #')
  plt.title('Pérdidas en la fase de entrenamiento en XOR')
  plt.show()

def fit_partial(self, x, y):
  # Lista de activaciones para cada capa conforme el dato pasa por la red
  # La primera activación es un caso especial, el vector de características en si mismo
  A = [np.atleast_2d(x)]
```

```
  # 1. PROPAGACIÓN HACIA DELANTE (FEEDFORWARD)
  for layer in np.arange(0, len(self.W)):
    net = A[layer].dot(self.W[layer]) # Multiplicación de activación por pesos actual (X)
    out = self.sigmoid(net) # Función de activación (X)
    A.append(out) # Añadimos la activación en cuestión a nuestra lista de activaciones (X)

  # 2. RETROPROPAGACIÓN (BACKPROPAGATION)
  error = A[-1] - y # Cáculo del error total(X)
  D = [error * self.sigmoid_deriv(A[-1])] #dE/do
  # Recorremos las capas para ir calculando las derivadas parciales
  # La última capa ya la hemos tenido en cuenta en el cáculo de la primera delta
  for layer in np.arange(len(A) - 2, 0, -1):
    # La nueva delta es la anterior multiplicada matricialmente por los pesos de la capa actual (traspuestos)
    # seguido del producto entre la delta y la derivada de la función de activación
    delta = D[-1].dot(self.W[layer].T) # (X)
    delta = delta * self.sigmoid_deriv(A[layer]) # (X)
    D.append(delta)
  # Invertimos la matriz D para tener ordenadas nuestras deltas según la red
  D = D[::-1]

  # 3. FASE DE ACTUALIZACIÓN DE PESOS (Aquí es dónde el aprendizaje se lleva a cabo)
  # Los nuevos pesos serán los antiguos pesos menos (dirección gradiente) el producto
  # de las activaciones de la capa en cuestión por el producto matricial de las deltas de dicha capa
  for layer in np.arange(0, len(self.W)):
    self.W[layer] += -self.eta*(A[layer].T.dot(D[layer])) # (X)

def predict(self, X, addBias=True):
  # Inicializamos la salida de la predicción con los valores de entrada
  p = np.atleast_2d(X)
  # Comprobar si hay que añadir el termino del bias
  if addBias:
    p = np.c_[p, np.ones((p.shape[0]))]
  # Vamos recorriendo todas las capas y computamos la activación en cuestión
  for layers in np.arange(0, len(self.W)):
    p = self.sigmoid(np.dot(p, self.W[layers])) # (X)
  return p

def calculate_loss(self, X, targets):
  # Predecimos las entradas y calculamos pérdidas
  targets = np.atleast_2d(targets)
  predictions = self.predict(X, addBias=False)
  loss = 0.5 * np.sum((predictions-targets)**2) # (X)
  return loss
```

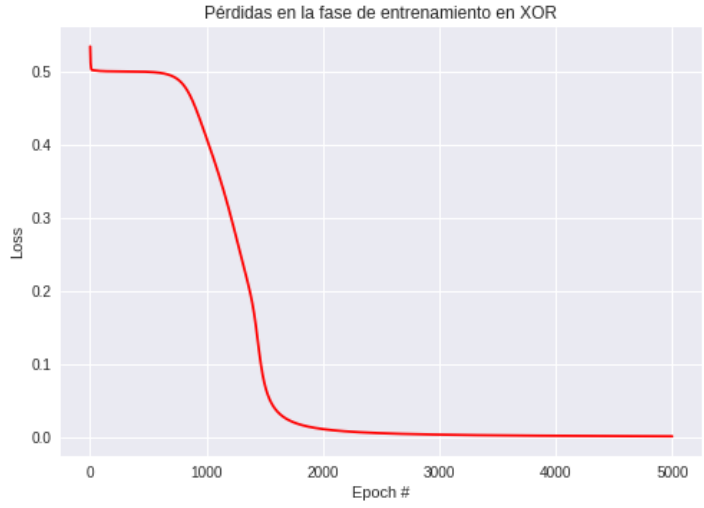
Pasamos a continuación al entrenamiento de la red con valores estimados tras varias pruebas de tasa de aprendizaje η=0.5 y del número de Épocas (epochs=5000).

```
import numpy as np
# Construimos el dataset XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Definición de arquitectura haciendo uso de la clase anterior
nn = NeuralNetwork([2, 2, 1], eta=0.5) # (X)

# Entrenamiento de la misma
nn.fit(X, y, epochs=5000)

# Fase de predicción
for x, target in zip (X,y):
  pred = nn.predict(x)[0][0]
  label = 1 if pred > 0.5 else 0
  print("[INFO] data={}, ground-truth={}, pred={:.4f}, step={}".format(x, target[0], pred, label))
```

```
[INFO]: epoch=1, loss=0.5338364
[INFO]: epoch=100, loss=0.5002617
[INFO]: epoch=200, loss=0.4998482
[INFO]: epoch=300, loss=0.4997261
[INFO]: epoch=400, loss=0.4995754
[INFO]: epoch=500, loss=0.4991514
[INFO]: epoch=600, loss=0.4978302
[INFO]: epoch=700, loss=0.4938036
[INFO]: epoch=800, loss=0.4817765
[INFO]: epoch=900, loss=0.4523759
[INFO]: epoch=1000, loss=0.4090519
```

……..

```
[INFO]: epoch=4000, loss=0.0022020
[INFO]: epoch=4100, loss=0.0021117
[INFO]: epoch=4200, loss=0.0020284
[INFO]: epoch=4300, loss=0.0019511
[INFO]: epoch=4400, loss=0.0018794
[INFO]: epoch=4500, loss=0.0018126
[INFO]: epoch=4600, loss=0.0017502
[INFO]: epoch=4700, loss=0.0016918
[INFO]: epoch=4800, loss=0.0016371
[INFO]: epoch=4900, loss=0.0015858
[INFO]: epoch=5000, loss=0.0015374
```




```
[INFO] data=[0 0], ground-truth=0, pred=0.0234, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.9750, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.9674, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.0289, step=0
```

Fig.33: Resultado de puesta en práctica del perceptrón para estudio de la función lógica XOR.

Se resume la vista de valores del cálculo de los 5000 recorridos (épocas) y en el que vemos la función de minimización del error según pasan/avanza el número de épocas y la disminución del valor de pérdidas y a esto la predicción. Vemos que aproximadamente al alcance de las 1800 épocas aproximadamente para este simple caso ya alcanza un valor de pérdidas prácticamente cero y el valor de salida pred de predicción alcanza el 96-97% para el valor 1 y el 0% para el valor 0.

Un ejemplo de referencia sencillo de la carga operacional de la GPU, operaciones matemáticas en la parte de Backpropagation y donde se hace latente en uso de GPU:

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}} \qquad \sigma(x) = \frac{1}{1+e^{-x}}$$ sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

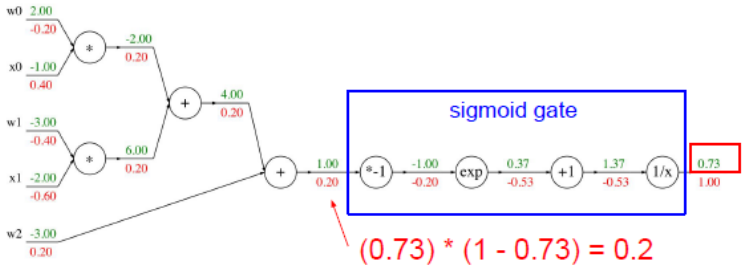


Fig.34: Carga computacional del algoritmo Backpropagation.

Estas operaciones llegan a realizarse por millones de datos según muestras de objetos a analizar como veremos más adelante.

5.- Aplicación TensorFlow y ejemplo de evaluación del dataset MINST.

A continuación de familiarizarnos con la tecnología de las GPU y una introducción a su aplicación en Deep Learning. es el momento aportar un ejemplo más detallado de su aplicación conjunta. El objetivo es mostrar un ejemplo de identificar dígitos del 0 al 9 escritos de forma manual. El set de datos MNIST es un conjunto de 70000 imágenes de 28x28 píxels que contienen números manuscritos junto con la etiqueta solución del número codificado (i.e. nuestro ground truth).

Fig. 35: Dataset MINST.

Se pretende en este ejemplo de aplicación y antes de pasar al siguiente ejemplo más completo donde realizaremos la evaluación que más nos destaca en nuestro estudio, la evaluación de imágenes, aportar una referencia a cómo podemos preparar una parte de programación con los cálculos correspondientes propios de Deep Learning a fin de aportarle los cálculos a efectuar la GPU y devolvernos los resultados esperados y en un tiempo eficiente y efectivo. Observar aplicativo de Tensorflow a las variables y funciones (ver tf.xxxxxx…) a fin de aportar asociación del programa con la GPU.

A continuación se reporta sumario del programa y operativa:

En este, se realiza la programación tomando como referencia los principales valores a tener en cuenta en la realización del entrenamiento de una red neuronal, preparando valores principales a fin de obtener una buena estimación de:

- Profundidad de la red: 128 neuronas.
- Tasa de aprendizaje: 0.005.
- Funciones de activación a predicción: softmax y sigmoid.
- Número de épocas: 1000.

```
def train_shallow_net(learning_rate, batch_size, num_epochs):
  # Creamos placeholders para ir almacenando los datos de entrada y los labels
  X = tf.placeholder(tf.float32, [None, 784]) #(X) # Imágenes del mnist: 28*28 = 784
  Y_true = tf.placeholder(tf.float32, [None, 10]) #(X) # Número indicando la clase 0-9 => 10 clases

  # Creamos e inicializamos las variables W y b con valores aleatorios que sigan una distribución normal
  W1 = tf.Variable(tf.truncated_normal([784, 200], stddev=0.1)) #(X)
  B1 = tf.Variable(tf.zeros([200])) #(X)
  W2 = tf.Variable(tf.truncated_normal([200, 10], stddev=0.1)) #(X)
  B2 = tf.Variable(tf.zeros([10])) #(X)

  # Calculamos las predicciones
  Y1 = tf.nn.sigmoid(tf.matmul(X,W1) + B1) #(X)
  Y_pred = tf.nn.softmax(tf.matmul(Y1,W2) + B2) #(X)

  # Definimos función de pérdidas (Cross_entropy)
  loss = tf.reduce_mean(-tf.reduce_sum(Y_true * tf.log(Y_pred), reduction_indices=1)) #(X)
  #loss = tf.reduce_mean(-tf.reduce_sum(Y_true*tf.log(Y_pred), reduction_indices=1))

  # Optimizador SGD
  train = tf.train.GradientDescentOptimizer(lr).minimize(loss) #(X)

  # % de predicciones correctas en un determinado batch (i.e. accuracy)
  is_correct = tf.equal(tf.argmax(Y_pred,1), tf.argmax(Y_true,1))
  accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))

  # Inicializamos variables
  init = tf.initializers.global_variables() #(X)
```

```
  # Abrimos la sesión
  with tf.Session() as sess:
    sess.run(init) #(X)
    # Entrenamiento de nuestra red
    acc_epoch_tr = []
    acc_epoch_val = []
    loss_epoch_tr = []
    loss_epoch_val = []
    for epoch in range(n_epochs):
      avg_acc = 0.
      avg_loss = 0.
      steps = int(mnist.train.num_examples/batch_size) #(X) Calcular número de batches
      for i in range(steps):
        batch_X, batch_Y = mnist.train.next_batch(batch_size) #(X) # Pedir un nuevo batch del set de datos (emplear función next_batch)
        sess.run(train, feed_dict={X: batch_X, Y_true: batch_Y}) #(X) # Entrenamos
        a,l = sess.run([accuracy, loss], feed_dict={X: batch_X, Y_true: batch_Y}) #(X) # Calculamos accuracy y cross_entropy del batch
        avg_acc += a / steps #(X) Calcular el accuracy medio de los diferentes batches
        avg_loss += l / steps #(X) Calcular las pérdidas medias de los diferentes batches
      # Almacenamos el accuracy y las losses medios para cada época
      acc_epoch_tr.append(avg_acc) #(X)
      loss_epoch_tr.append(avg_loss) #(X)
      # Calculamos accuracy y losses en validation
      a_val, l_val = sess.run([accuracy, loss], feed_dict={X: mnist.validation.images, Y_true: mnist.validation.labels}) #(X)
      acc_epoch_val.append(a_val) #(X)
      loss_epoch_val.append(l_val) #(X)
      # Sacamos información por pantalla
      print("[INFO]: Época {} ---> Acc_train = {} - Loss_train = {} - Acc_val = {} - Loss_val = {}".format(epoch, avg_acc, avg_loss, a_val, l_val)) #(X)

    # Cálculo de accuracy y losses en el conjunto de test
    a_test, l_test = sess.run([accuracy, loss], feed_dict={X: mnist.test.images, Y_true: mnist.test.labels}) #(X)
    print("[INFO]: Accuracy en test = {} - Losses en test = {}".format(a_test, l_test)) #(X)

    # Gráficar losses por época
    plt.plot(np.arange(0, n_epochs), loss_epoch_tr) #(X)
    plt.plot(np.arange(0, n_epochs), loss_epoch_val)
    plt.legend(['train', 'val'], loc='upper left')
    plt.title('Training Loss') #(X)
    plt.xlabel('Epoch #') #(X)
    plt.ylabel('Loss') #(X)

# Lanzamos entrenamiento
lr = 0.005
b_size = 128
n_epochs = 1000
train_shallow_net(lr, b_size, n_epochs)
```

A continuación se resume reporte de cálculo operativo donde se aprecia el reporte operativo de valores como el número de época, la precisión obtenida (Acc_train con respecto a la objetiva Acc_val) y la disminución de perdidas por optimización de los pesos y a esto optimización según aumenta en este caso el número de épocas y según la profundidad de la red.

```
[INFO]: Época 980 ---> Acc_train = 0.9703889860139862 - Loss_train = 0.1079792779390559 - Acc_val = 0.9692000150680542 - Loss_val = 0.11726
[INFO]: Época 981 ---> Acc_train = 0.9716091200466205 - Loss_train = 0.10575051805425645 - Acc_val = 0.968999981880188 - Loss_val = 0.11718
[INFO]: Época 982 ---> Acc_train = 0.9704800407925411 - Loss_train = 0.10639106357656856 - Acc_val = 0.9688000082969666 - Loss_val = 0.1171
[INFO]: Época 983 ---> Acc_train = 0.9702797202797208 - Loss_train = 0.1082424585878987 - Acc_val = 0.9693999886512756 - Loss_val = 0.11712
[INFO]: Época 984 ---> Acc_train = 0.9714634324009315 - Loss_train = 0.10556996419226926 - Acc_val = 0.9692000150680542 - Loss_val = 0.1169
[INFO]: Época 985 ---> Acc_train = 0.9706439393939393 - Loss_train = 0.10689124753286228 - Acc_val = 0.968999981880188 - Loss_val = 0.11695
[INFO]: Época 986 ---> Acc_train = 0.9698608682983693 - Loss_train = 0.10866232223739293 - Acc_val = 0.968999981880188 - Loss_val = 0.11694
[INFO]: Época 987 ---> Acc_train = 0.9704800407925409 - Loss_train = 0.10581644772093898 - Acc_val = 0.968999981880188 - Loss_val = 0.11683
[INFO]: Época 988 ---> Acc_train = 0.9713177447552448 - Loss_train = 0.10663967294779138 - Acc_val = 0.9692000150680542 - Loss_val = 0.1167
[INFO]: Época 989 ---> Acc_train = 0.9712813228438241 - Loss_train = 0.10452528745109667 - Acc_val = 0.968999981880188 - Loss_val = 0.11670
[INFO]: Época 990 ---> Acc_train = 0.9706803613053616 - Loss_train = 0.1058512522398314 - Acc_val = 0.9697999954223633 - Loss_val = 0.11661
[INFO]: Época 991 ---> Acc_train = 0.9711174242424241 - Loss_train = 0.10659093308177864 - Acc_val = 0.9693999886512756 - Loss_val = 0.1165
[INFO]: Época 992 ---> Acc_train = 0.9709535256410262 - Loss_train = 0.10558094039107814 - Acc_val = 0.9697999954223633 - Loss_val = 0.1165
[INFO]: Época 993 ---> Acc_train = 0.970953525641026 - Loss_train = 0.10643288380268832 - Acc_val = 0.9693999886512756 - Loss_val = 0.11642
[INFO]: Época 994 ---> Acc_train = 0.970698572261073 - Loss_train = 0.10530056329422344 - Acc_val = 0.9696000218391418 - Loss_val = 0.11640
[INFO]: Época 995 ---> Acc_train = 0.9714452214452216 - Loss_train = 0.10494673970678617 - Acc_val = 0.9693999886512756 - Loss_val = 0.1164
[INFO]: Época 996 ---> Acc_train = 0.9705346736596728 - Loss_train = 0.10637011201856848 - Acc_val = 0.9696000218391418 - Loss_val = 0.1161
[INFO]: Época 997 ---> Acc_train = 0.9715726981351984 - Loss_train = 0.1054518478803145 - Acc_val = 0.9685999751091003 - Loss_val = 0.11624
[INFO]: Época 998 ---> Acc_train = 0.9701522435897441 - Loss_train = 0.10702457731819791 - Acc_val = 0.968999981880188 - Loss_val = 0.11617
[INFO]: Época 999 ---> Acc_train = 0.9718640734265737 - Loss_train = 0.10350357612599306 - Acc_val = 0.9697999954223633 - Loss_val = 0.1160
[INFO]: Accuracy en test = 0.964900016784668 - Losses en test = 0.11837846785783768
```

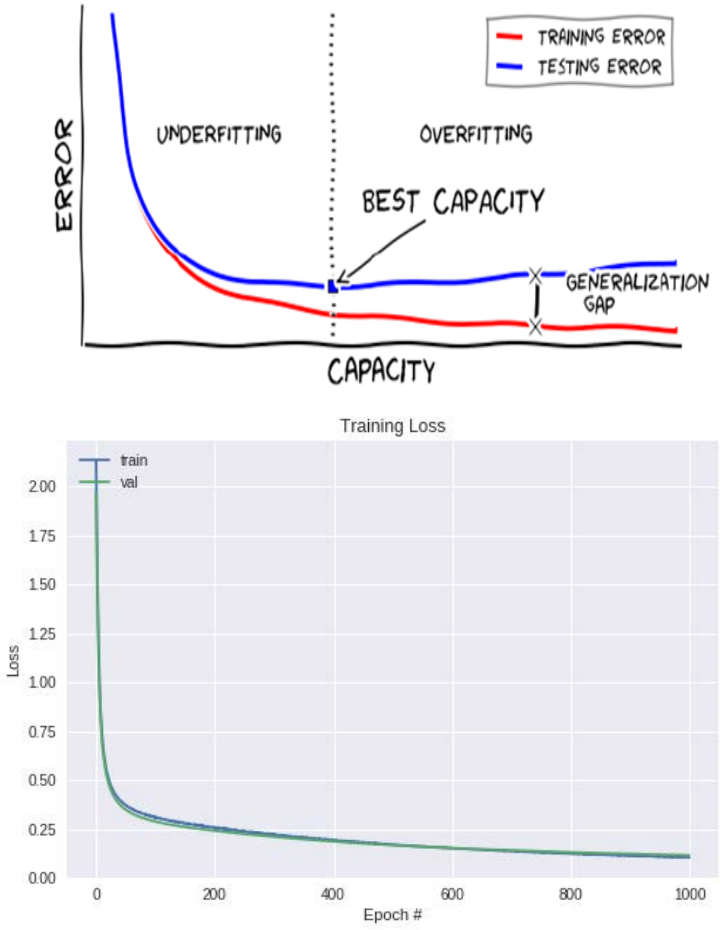


Fig.36: Obtención de resultados óptimos tras entrenamiento del MINST.

Resultados: para esta prueba se requirió en un ordenador de referencia convencional, un tiempo aproximado de 20 minutos. Por motivos de exclusión del objetivo de este trabajo, resumimos aquí el reporte de una prueba objetiva conseguida como es este ejemplo. Quedaría la realización

de diferentes evaluaciones de observación vs. respuesta según diferentes valores de relación de parámetros como Tasa de aprendizaje en relación a profundidad de la red y número de épocas, valores que constituyen los principales motivos de varias evaluaciones y que gracias a las GPU podemos tener posibilidad de realizar estas evaluaciones y a esto, poder realizar estudios competitivos en esta materia aquí tratada.

En el siguiente ejemplo principal aplicado a la evaluación de imágenes, reportamos más detalles de este referente a fin de una mejor comprensión de estudio de relación GPU con estas aplicaciones.

6.- Evaluación de imágenes en Deep Learning. Aplicación Keras.

A continuación y una vez puesto en referencia todo el proceso operativo con el que nos disponemos a realizar trabajos de alta capacidad de computo en una GPU, pasamos a continuación a uno de los ejemplos más importantes que nos destaca en este proyecto, la evaluación de imágenes.

En el siguiente proceso se reporta un sumario de la preparación del programa con el que le pasamos a la GPU la "carga" operativa. Para ello en primer lugar pasamos a reportar el esquema operativo y de programación con el que operaremos en la GPU. También destacar como anteriormente por la sencillez por muestra para con respecto a un ejemplo sencillo, no se ha contemplado la posibilidad de implementar parámetros adicionales que si que reportaremos ahora.

En primer lugar mostramos el proceso de preparación del programa en el que comenzaremos con un resumen de los pasos a realizar:

- En primer lugar y como hemos reportado en los ejemplos anteriores de Perceptrón simple donde realizamos un entrenamiento a una prueba simple de evaluación de la función XOR respecto de predicción de sus valores y en el que a continuación hemos ampliado detalle a ver el ejemplo de evaluación del dataset MINST también de carga computacional simple como ejemplos principales sencillos. A continuación mostramos como para la evaluación de imágenes se requiere de la preparación de esquemas de red neuronal más completos y a esto más complejos. Entran a formar parte aquí las redes convolucionales y si bien procesos adicionales como Pooling y Dropout y Data aumentation y parámetros adicionales como padding (ajuste tamaño de la imagen con respecto del filtro de convolución) y pooling (diezmado de las dimensiones espaciales a fin de reducir la cantidad de parámetros y el coste computacional de la red y como control del overfitting: convergencia de los valores predicción respecto optimización de la red) con el fin de observar en nuestro caso la capacidad de computo añadido a la GPU.

También se utilizan funciones adicionales como la división del training set en subsecciones, llamadas batches, validación y test:

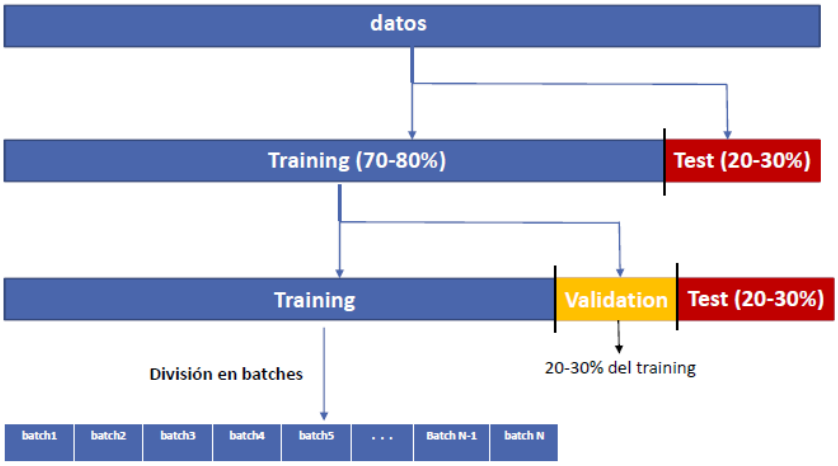


Fig.37: División del training set en subsecciones: training, validation y test.

Una forma de optimizar el conjunto operativo es la división de los datos (training set) en subdivisiones (training, validation y test), el training subdividido en lotes (batches) donde de esta manera optimizamos y aplicamos eficiencia al conjunto operativo y respecto a su contenido, pasando tan solo a la GPU las partes principales.

Bloques convolucionales: la salida (mapa de activación) de la primera capa convolucional es la entrada de la segunda capa y así sucesivamente.

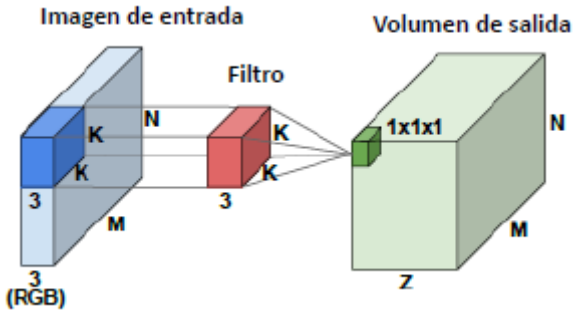


Fig.38: Estructura de capa convolucional. Aplicación de filtros.

$$y[m,n] = x[m,n] * h[m,n] = \sum_k \sum_l x[k,l]h[m-k,n-l]$$

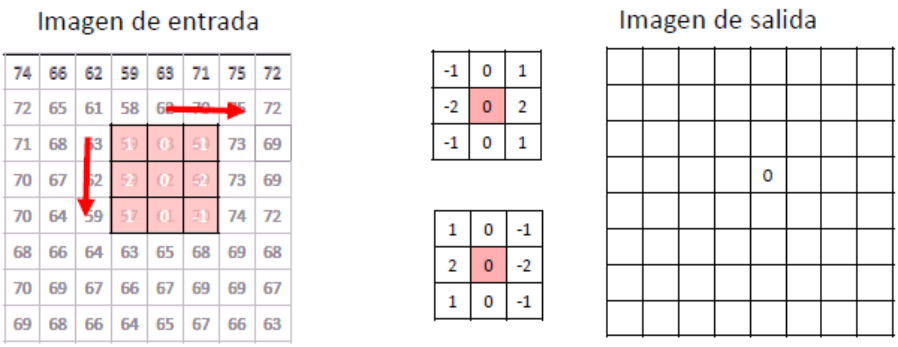


Fig.39: Esquema de operación de convolución.

Diversas capas dependientes, interconectadas e interactuando como:

- Capa convolucional.
- Capa de activación.
- Capa de pooling.
- Capa fully-connected.

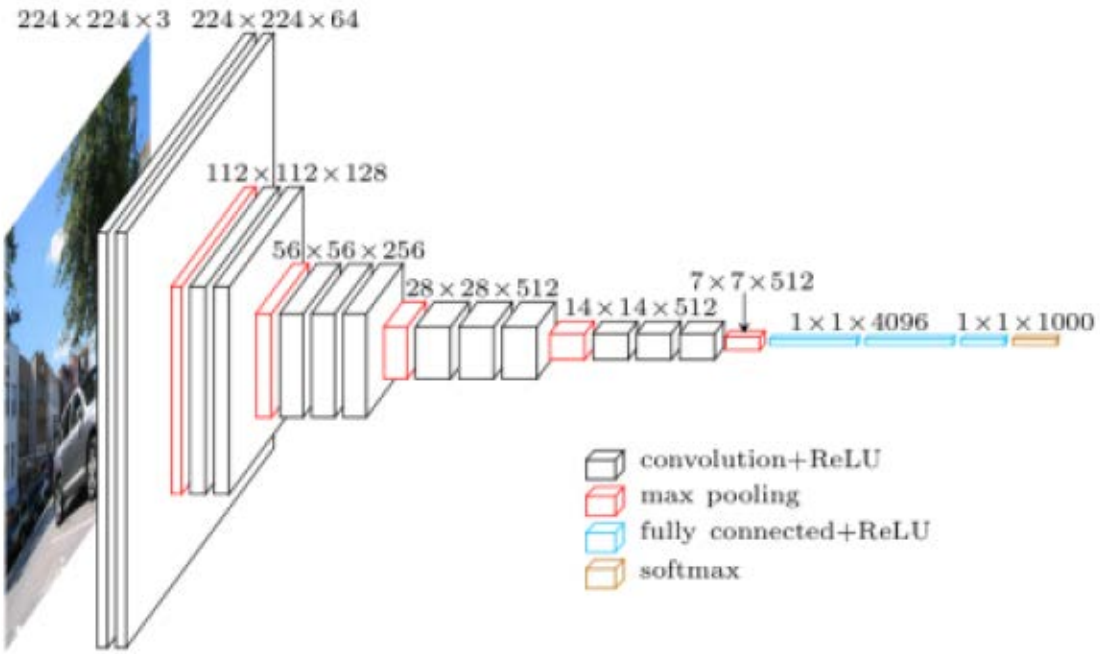


Fig.40: Arquitectura de una Red Convolucional.

Una vez visto uno de los esquemas a más completitud más importantes implementados en los métodos operativos de evaluación de una red Deep Learning. Pasamos a continuación a completar con la parte de programación asociada y en la que en este caso ponemos de ejemplo uno de los frameworks más utilizados, Keras.

Keras: es un framework de alto nivel para el entrenamiento de redes neuronales. Esta librería fue desarrollada por François Chollet en 2015 con el objetivo de simplificar la programación de algoritmos basados en aprendizaje profundo.

El entrenamiento se realiza utilizando GPU, teniendo en cuenta que es la única forma de entrenar una red neuronal en un intervalo de tiempo permisible.

El porque de Keras se debe a ser un código optimizado y ampliamente validado. Dispone de librerías de alto nivel específicas para este propósito. También permite disponer de librerías de bajo nivel para interactuar con otras como las de TensorFlow, Microsoft Cognitive Toolkit o Theano.

6.1.- Capas básicas en Keras:

keras.layers.Dense(units, activation=**None**, use_bias=**True**,
kernel_initializer='glorot_uniform', bias_initializer='zeros')

keras.layers.Activation(activation)

keras.layers.Dropout(rate, noise_shape=**None**, seed=**None**)

keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid')

keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)

keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=**True**, scale=**True**)

keras.layers.Flatten(data_format=None)

Definiendo una arquitectura:

En primer lugar creamos la API en la que en este caso diseñaremos un modelo de Keras y que corresponde a la referencia más simple, referencia llamada Modo Secuencial, en la que se crea una pila de capas lineal.

#Imports necesarios

from keras.models **import** Sequential

from keras.layers **import** Dense,Activation

#Creamos objeto modelo secuencial

model=**Sequential**()

#Definimos la arquitectura añadiendo capas al modelo

model.add(**Dense**(64,input_dim=784))

model.add(**Activation**('relu'))

model.add(**Dense**(64,activation='relu'))

model.add(**Dense**(10,activation='softmax'))

Antes de entrenar el modelo Keras es necesario realizar una compilación del mismo configurando el proceso de entrenamiento. Este proceso se lleva a cabo mediante el comando **model.compile**, a continuación varios ejemplos según la referencia de clasificación:

 # Ejemplo para un problema de clasificación multi clase

model.**compile**(optimizer='sgd',

loss='categorical_crossentropy',

metrics=['accuracy'])

 # Ejemplo para un problema de clasificación binario

model.**compile**(optimizer=**SGD**(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True),

loss='binary_crossentropy',

metrics=['accuracy'])

 # Ejemplo para un problema de regresión

model.**compile**(optimizer='rmsprop',

loss='mse')

Una vez compilado, ya es posible lanzar el proceso de entrenamiento. Keras entrena modelos a partir de datos y etiquetas almacenadas en Numpy arrays. La función de entrenamiento es el método **model.fit**

 # Generación de datos

import numpy **as** np

data = np.random.random((1000, 100))

labels= np.random.randint(2, size=(1000, 1))

 # Ejemplo con etiquetas en one-hot encoding

model.**compile**(optimizer='rmsprop',loss='categorical_crossentropy',

metrics=['accuracy'])

one_hot_labels= keras.utils.to_categorical(labels, num_classes=10)

model.**fit**(data, one_hot_labels, epochs=10, batch_size=32)

 # Ejemplo etiquetas categóricas

model.**compile**(optimizer='rmsprop', loss='sparse_categorical_crossentropy',

metrics=['accuracy'])

model.**fit**(data, labels, epochs=10, validation_data=(data_v, labels_v), batch_size=32)

 # Ejemplo con model.evaluate

model.**evaluate**(x=X_test, y=y_test, batch_size=None)

% Devuelve los valores de pérdidas y métricas empleadas enentrenamiento para los datos de test

 # Ejemplo con model.predicty classification_report

fromsklearn.metrics**import**classification_report

model.**predict**(x=X_test, batch_size=None)

print(**classification_report**(y_labels, predictions.argmax(axis=1))

% Devuelve más métricas

6.2.- El dataset CIFAR-10

Se trata de un conjunto de datos compuesto de 60000 imágenes RGB de dimensiones 32x32pixels pertenecientes a 10 clases distintas (6000 imágenes por clase). CIFAR10 se separa en dos subconjuntos de datos: 50000 imágenes para entrenamiento y las 10000 restantes se emplean como set de test.

A continuación las clases que componen el dataset, así como las 10 imágenes aleatorias que componen cada clase.

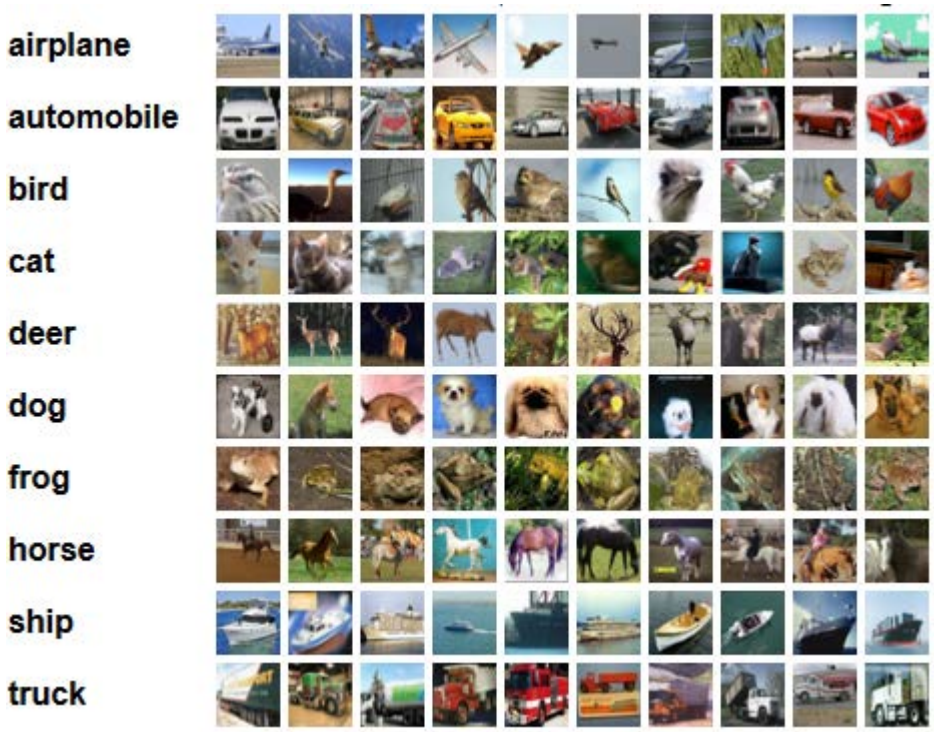


Fig.41: Conjunto de clases e imágenes del dataset CIFAR-10

Debido al entorno en el que estamos obligados a desarrollar para disponer de una GPU de manera gratuita (**Google Colab**) se hace indispensable la comunicación con **Google Drive** para la **lectura, escritura de datos**.

from google.colab **import** drive #Paquete para la comunicación

drive.**mount**('/content/drive') **#**Montamos la unidad de Drive

Como hemos indicado, contamos con 60000 imágenes RGB de dimensiones 32x32 píxels pertenecientes a 10 clases distintas (6000 imágenes por clase). Se realiza la separación en dos subconjuntos de datos: 50000 imágenes para entrenamiento y las 10000 restantes se emplean como set de test.

Así pues, lo primero a realizar es Importar el set de datos:

```
# Importando el set de datos CIFAR10
from keras.datasets import cifar10
from sklearn.preprocessing import LabelBinarizer
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0
labelNames = ["Avión", "Automóvil", "Pájaro", "Gato", "Ciervo", "Perro", "Rana", "Caballo", "Barco", "Camión"]
# Por si es necesario convertir a one-hot encoding
#lb = LabelBinarizer()
#trainY = lb.fit_transform(trainY)
#testY = lb.transform(testY)
```

```
Using TensorFlow backend.
[INFO] loading CIFAR-10 data...
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 40s 0us/step
```

Una vez en memoria el set de datos CIFAR-10, mostramos unas cuantas imágenes para visualizar la variedad existente:

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(14,10))
for n in range(1, 29):
    fig.add_subplot(4, 7, n)
    img = trainX[n]
    plt.imshow(img)
    plt.title(labelNames[trainY[n][0]])
    plt.axis('off')
```

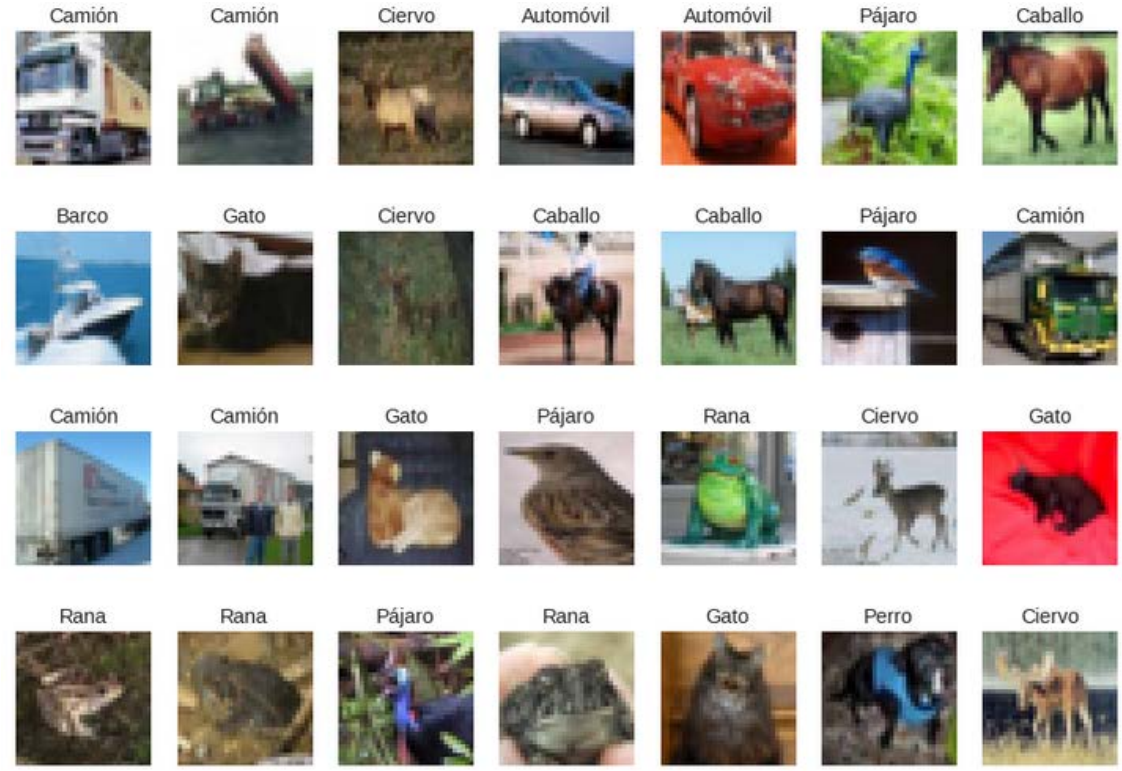
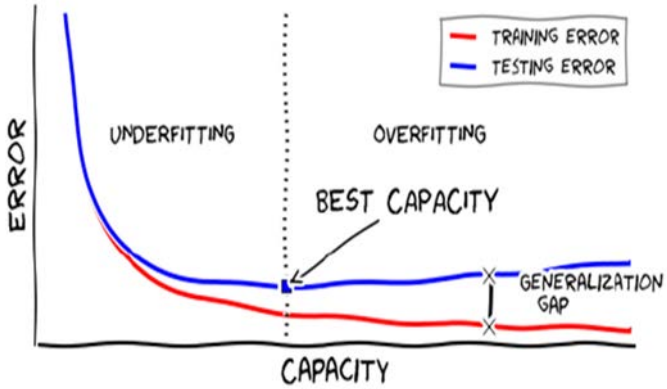


Fig. 42: Conjunto de datos para entrenamiento de CIFAR-10 cargado.

Programamos nuestro set de entrenamiento con varios valores de neuronas (entre 1024 y 512), funciones de activación ReLU y SGD como optimizadores y valores de tasa de aprendizaje de 0.01, Nº de épocas =50 y tamaño de bath=32. Posteriormente aplicamos Dropout (desconexión de algunas neuronas).

```
# Imports necesarios
import numpy as np
from sklearn.metrics import classification_report
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD
import matplotlib.pyplot as plt

# Pasamos los datos a vector con la función reshape
trainX = trainX.reshape((trainX.shape[0], 3072)) #(X)
testX = testX.reshape((testX.shape[0], 3072)) #(X)

# Arquitectura de red
# Definimos el modo API Sequential
model = Sequential() #(X)
# Primera capa oculta
model.add(Dense(2048, input_shape=(3072,), activation="relu")) #(X)
#model.add(Dropout(0.5))
# Segunda capa oculta
model.add(Dense(1024, activation="relu")) #(X)
#model.add(Dropout(0.5))
# Tercera capa oculta
model.add(Dense(512, activation="relu")) #(X)
#model.add(Dropout(0.5))
# Cuarta capa oculta
model.add(Dense(128, activation="relu")) #(X)
#model.add(Dropout(0.5))
# Quinta capa oculta
model.add(Dense(32, activation="relu")) #(X)
# Capa de salida
model.add(Dense(10, activation="softmax")) #(X)
```

```
# Compilamos el modelo y entrenamos
print("[INFO]: Entrenando red neuronal...")
# Compilamos el modelo
model.compile(loss="sparse_categorical_crossentropy", optimizer=SGD(0.01), metrics=["accuracy"]) # Etiquetas en decimal #(X)
# model.compile(loss="sparse_categorical_crossentropy", optimizer=SGD(0.01), metrics=["accuracy"]) # Etiquetas binarias #(X)
# Entrenamos el perceptrón multicapa
H = model.fit(trainX, trainY, validation_data=(testX, testY), epochs=50, batch_size=32) #(X)

# Evaluamos con las muestras de test
print("[INFO]: Evaluando modelo...")
# Efectuamos predicciones
predictions = model.predict(testX, batch_size=32) #(X)
# Obtenemos el report
print(classification_report(testY, predictions.argmax(axis=1), target_names=labelNames)) # Etiquetas en decimal #(X)
# print(classification_report(testY.argmax(axis=1), predictions.argmax(axis=1), target_names=labelNames)) # Etiquetas binarias

# Mostramos gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 50), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 50), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 50), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, 50), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```



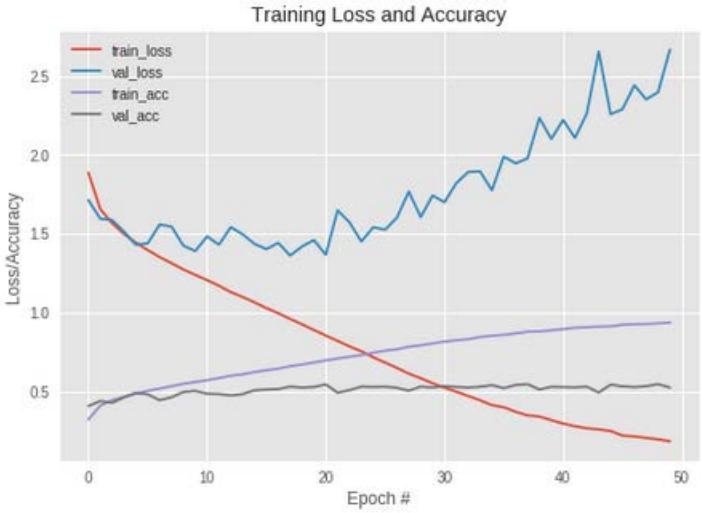


Fig. 43: Resultados del entrenamiento respecto a distintos valores de variables.

Como resultado, reportamos aquí una muestra de imprecisión donde con los valores iniciales pasados, no obtenemos un resultado conforme (comparar gráficas anteriores: valor esperado vs. valor obtenido) donde vemos como los valores no corresponden al esperado. Con esto pretendemos ver la carga operacional a la que podemos llegar a ocupar una GPU en función de los distintos parámetros y valores con los que participamos. Por esto, requerimos de una estructura más compleja donde preparamos un training ser más completo a fin de obtener el

resultado óptimo esperado, observándose overfitting (no convergencia al valor óptimo esperado).

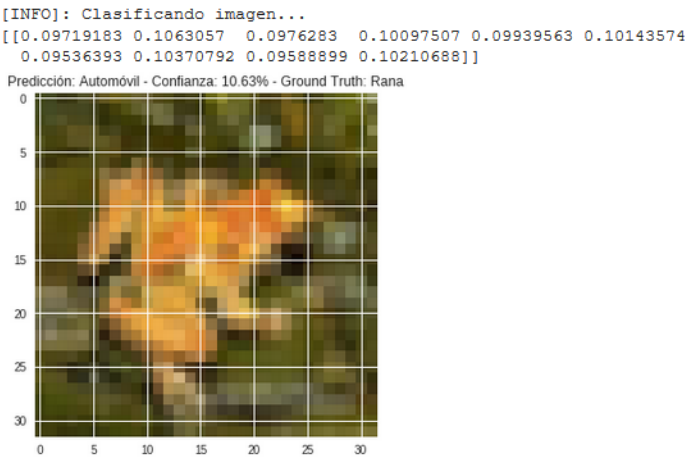


Fig.44: Resultado no optimo tras un ajuste no apropiado de training set.

Vemos aquí como falla la predicción (el valor real de la imagen (Ground truth: rana) y el predecido ha sido un automóvil. Así pues, hemos de configurar nuestra estructura de training set con complementos adicionales como (pooling y dropout) a fin de obtener una mejor precisión y resultado conforme.

Así pues, a la vista del resultado, debemos de realizar operaciones de ajustes y modificación de parámetros a nuestro esquema de red neuronal en la que ahora pasamos a implementar esquemas de red más completos como el siguiente:

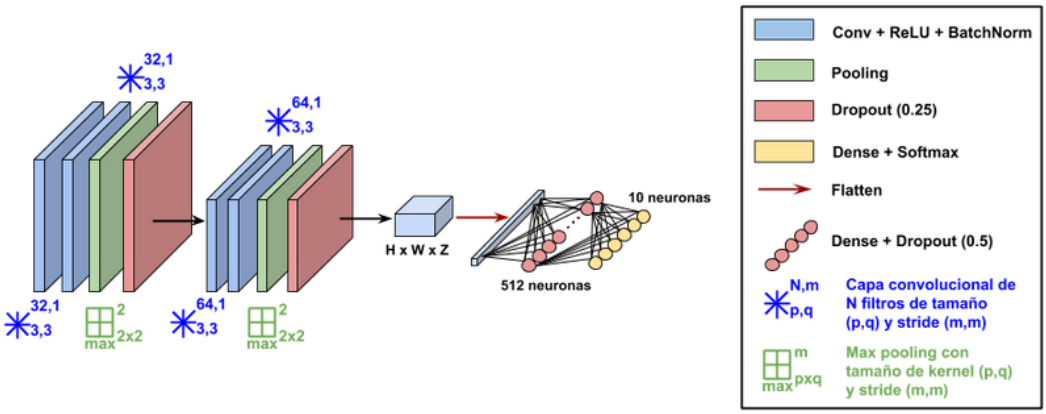


Fig. 45: Training set más constituido a fin de precisar un resultado óptimo.

Con esta estructura de programa, vamos a ver a continuación como ahora si que encontramos una arquitectura que discrimina con buena precisión entre las 10 clases de CIFAR-10. En el que, aplicando técnicas adicionales como batchNormalization se reduce el overfitting (no convergencia como hemos visto en el resultado anterior).

A continuación procedemos a modificar el programa donde ahora definimos una función: def Deep_CNN

```python
# import the necessary packages
import numpy as np
from keras import backend as K
from keras.layers.convolutional import Conv2D
from keras.layers import Input
from keras.models import Model
from keras.layers.core import Activation, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import MaxPooling2D
from keras.models import Sequential
from keras.optimizers import SGD
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from google.colab import drive

def deep_CNN(width, height, depth, classes, batchNorm):

  # Definimos entradas en modo "channels last"
  inputs = Input(shape=(height, width, depth)) #(X)

  # Definimos la arquitectura
  # Primer set de capas CONV => RELU => CONV => RELU => POOL
  x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(inputs) #(X)
  if batchNorm:
    x1 = BatchNormalization()(x1) #(X)
  x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(x1) #(X)
  if batchNorm:
    x1 = BatchNormalization()(x1) #(X)
  x1 = MaxPooling2D(pool_size=(2, 2))(x1) #(X)
  x1 = Dropout(0.25)(x1) #(X)

  # Segundo set de capas CONV => RELU => CONV => RELU => POOL
  x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x1) #(X)
  if batchNorm: #(X)
    x2 = BatchNormalization()(x2) #(X)
  x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x2) #(X)
  if batchNorm:
    x2 = BatchNormalization()(x2) #(X)
  x2 = MaxPooling2D(pool_size=(2, 2))(x2) #(X)
  x2 = Dropout(0.25)(x2) #(X)

  # Primer (y único) set de capas FC => RELU
  xfc = Flatten()(x2) #(X)
  xfc = Dense(512, activation="relu")(xfc) #(X)
  if batchNorm:
    xfc = BatchNormalization()(xfc) #(X)
  xfc = Dropout(0.5)(xfc) #(X)
  # Clasificador softmax
  predictions = Dense(classes, activation="softmax")(xfc) #(X)
```

```
[INFO]: Clasificando imagen...
[[1.3345449e-04 1.2835833e-04 1.0331073e-02 3.7939978e-01 2.3606065e-01
  8.3651528e-02 2.8488237e-01 9.0961327e-04 2.2878039e-03 2.2153726e-03]]
```

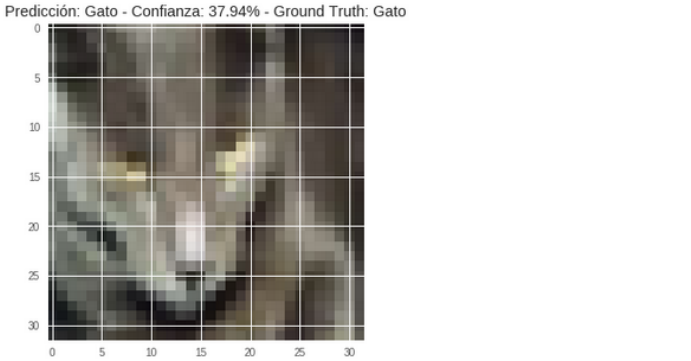


Fig.46: Tras ajustes en la red: vista de un resultado más optimizado

Vemos como añadiendo modificaciones disponibles a la red neuronal podemos obtener mejores resultados. Se hace aquí visible la obviedad de relación mayor carga computacional mayor orden de operación en la GPU.

Finalmente, tras otros varios ajustes que escapan a este trabajo se reporta a continuación un resultado finalmente óptimo en ejecución:

```
Epoch 45/50
55000/55000 [==============================] - 2s 38us/step - loss: 0.0674 - acc: 0.9808 - val_loss: 0.0907 - val_acc: 0.9734
Epoch 46/50
55000/55000 [==============================] - 2s 39us/step - loss: 0.0647 - acc: 0.9814 - val_loss: 0.0902 - val_acc: 0.9732
Epoch 47/50
55000/55000 [==============================] - 2s 39us/step - loss: 0.0634 - acc: 0.9823 - val_loss: 0.0907 - val_acc: 0.9726
Epoch 48/50
55000/55000 [==============================] - 2s 39us/step - loss: 0.0614 - acc: 0.9827 - val_loss: 0.0896 - val_acc: 0.9722
Epoch 49/50
55000/55000 [==============================] - 2s 39us/step - loss: 0.0600 - acc: 0.9835 - val_loss: 0.0920 - val_acc: 0.9734
Epoch 50/50
55000/55000 [==============================] - 2s 39us/step - loss: 0.0584 - acc: 0.9836 - val_loss: 0.0888 - val_acc: 0.9734
[INFO]: Evaluando red neuronal...

Using TensorFlow backend.
[INFO]: Cargando VGG16...
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (
Instructions for updating:
Colocations handled automatically by placer.
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_ke
553467904/553467096 [==============================] - 7s 0us/step
[INFO]: Clasificando imagen con el modelo VGG16
Downloading data from https://s3.amazonaws.com/deep-learning-models/image-models/imagenet_class_index.json
40960/35363 [==============================] - 0s 2us/step
1. soccer_ball: 93.43%
2. rugby_ball: 6.06%
3. golf_ball: 0.20%
4. volleyball: 0.17%
5. tennis_ball: 0.05%
(-0.5, 569.5, 378.5, -0.5)
```

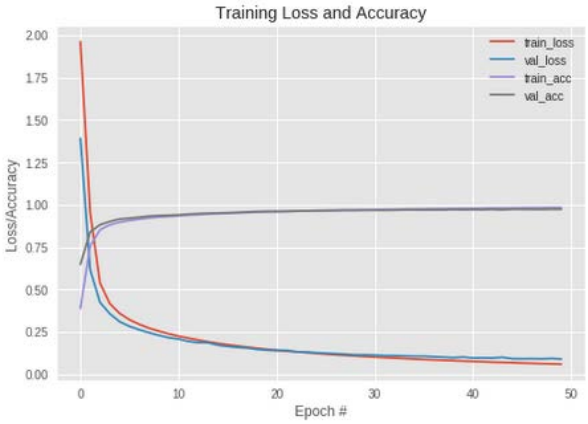


Fig.47: Resultados de training set con la configuración anterior. Valor óptimo esperado.

Finalmente, a modo de resumen, mostramos este ejemplo anterior en el que tras diversos ajustes, vemos como podemos conseguir programas con los que poner en aplicación a las GPUs y obtener resultados operativos esperados (observar gráfica de respuesta Nº de épocas / Precisión y Pérdida: bajo número de épocas y buenos valores de precisión (mayores al 90%) y

bajas pérdidas (menores al 1%)) con ello poder realizar trabajos que actualmente se están aplicando en la actualidad gracias a su potencia de cálculo.

En este ejemplo anterior, se ha conseguido finalmente acertar de entre un conjunto de imágenes aleatorias, acertar con una precisión del 93.43 % un objeto (pelota de fútbol en este caso).

7.- Visión Artificial. Open CV.

En esta sección se aborda el concepto de visión artificial con aplicación en el uso de GPU. Para ello y siguiendo en la línea de lo anterior, a continuación veremos una de las principales librerías utilizadas en la actualidad, Open CV, con multitud de funciones además de características avanzadas para el procesamiento de imágenes y objetos tanto en imagen única como en procesamiento de vídeo. Posteriormente se aborda un ejemplo de primera actualidad como es el reconocimiento facial.

Fig.48: Librería de visión artificial Open CV.

7.1.- Leer una imagen

Disponemos la función **cv2.imread()** para leer una imagen. La imagen debe estar en el directorio de trabajo o se ha de señalar una ruta absoluta a la imagen.

El segundo argumento es un indicador (o bandera) que especifica la forma en que se debe leer la imagen.

- cv2.IMREAD_COLOR: carga una imagen de color. Cualquier transparencia de la imagen será ignorada. Es el indicador (o bandera) predeterminado.
- cv2.IMREAD_GRAYSCALE: carga la imagen en modo de escala de grises
- cv2.IMREAD_UNCHANGED: carga la imagen como sin alteraciones incluyendo el canal alfa

Nota: en lugar de estos tres indicadores, simplemente podemos pasar números enteros, en concreto 1, 0, o -1.

```
import cv2
# Ejemplo de cargar una imagen en escala de grises
img = cv2.imread('imagen.jpg',0)
cv2.imshow('image',img)
```

7.2.- Mostrar una imagen

Disponemos de la función **cv2.imshow()** para mostrar una imagen en una ventana. La ventana se ajusta automáticamente al tamaño de la imagen.

El primer argumento corresponde a un un nombre de ventana el cual es una cadena (tipo de dato string). El segundo argumento es nuestra imagen. Podemos crear tantas ventanas como desees, pero con nombres diferentes de ventana.

```
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

cv2.waitKey () es una función de enlace con el teclado. Su argumento es tiempo en milisegundos. La función espera durante milisegundos especificados que ocurra cualquier evento de teclado. Si presionamos cualquier tecla en ese momento, el programa continúa. Si le pasamos el valor 0, la espera del evento es indefinida hasta que se presione una tecla. También se puede configurar para detectar pulsaciones de teclas específicas, por ejemplo, si se presiona la tecla *a* tecla, etc,

cv2.destroyAllWindows() Esta función destruye todas las ventanas creadas. Si deseamos destruir una ventana específica, utilice la función de cv2.destroyWindow () donde se pasa el nombre de la ventana a eliminar como argumento.

Hay un caso especial en que podemos crear una ventana y cargar la imagen posteriormente. En ese caso, podemos especificar si la ventana es redimensionable o no. Esto se realiza con la función **cv2.namedWindow()**. Por defecto, el indicador es cv2.WINDOW_AUTOSIZE. Pero si se especifica la que el indicador sea cv2.WINDOW_NORMAL, podemos cambiar el tamaño de la ventana. Esto será útil cuando las dimensiones de la imagen sean muy grandes y se añada una barra de seguimiento (o un scroll).

```
cv2.namedWindow('image', cv2.WINDOW_NORMAL)
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

7.3.- Guardar una imagen

Disponemos de la función **cv2.imwrite ()** para guardar una imagen.

El primer argumento es el nombre del archivo y el segundo argumento es la imagen que deseamos guardar.

```
cv2.imwrite('deepgris.png',img)
```

Esto guardará la imagen en formato PNG en el directorio de trabajo.

7.4.- Captura de vídeo desde la cámara

Para adquirir emisiones en vivo con la cámara. Open CV proporciona un interfaz muy simple para esto. Podemos capturar un vídeo desde la cámara (utilizando una webcam incorporada al pc), convirtiendo en vídeo a escala de grises y mostrándolo.

Para capturar un vídeo, es necesario crear un objeto **VideoCapture**. Su argumento puede ser el índice del dispositivo o el nombre de un archivo de vídeo. El índice del dispositivo es el número para especificar qué cámara se utilizará. Normalmente se conectará una cámara. Así que simplemente pasamos 0 (o -1). Podemos seleccionar una segunda cámara pasando 1 y así sucesivamente. Después de esto, podemos capturar el vídeo. Al final, no olvidar liberar la captura.

```
import numpy as np
import cv2
cap = cv2.VideoCapture(0)
```

```
while(True):
  # Captura video cuadro a cuadro
  ret, frame = cap.read()
  # Nuestras operaciones sobre los cuadros se hacen aqui
  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  # Muestra el cuadro resultante
  cv2.imshow('frame',gray)
  if cv2.waitKey(1) & 0xFF == ord('q'):
      break

# Cuando todo está listo, se libera la captura
cap.release()
cv2.destroyAllWindows()
```

cap.read() devuelve un bool (verdadero / falso). Si el cuadro se lee correctamente, será verdadero (true). Por lo tanto, se puede comprobar el final del vídeo consultando este valor de retorno.

A veces, cap puede no haber inicializado la captura. En ese caso, este código mostrara error. Puede comprobar si se inicia o no con el método **cap.isOpened()**. Si es Verdad (true), excelente. De lo contrario abrirlo con **cap.open()**.

También podemos acceder a algunas de las características de este vídeo usando el método **cap.get(PropID)**, donde PropID es un número del 0 al 18. Cada número indica una propiedad del video (si es aplicable a ese vídeo) y se puede detallar completamente aquí: **Property Identifier** . Algunos de estos valores se pueden modificar mediante **cap.set (propId, value)**. Valor es el nuevo valor que se desea.

Como ejemplo, podemos comprobar la anchura del marco y la altura por cap.get(3) and cap.get(4). Da como resultado por defecto 640×480. Pero si quieres modificarlo a 320×240.Sólo tienes que utilizar ret = cap.set(3,320) y ret = cap.set(4,240).

7.5.- Reproducción de vídeo desde un archivo

Es igual que capturar desde la Cámara, solo cambiar el índice de la cámara con el nombre del archivo de vídeo. También, mientras que se visualice el cuadro, utiliza el tiempo apropiado para cv2.waitKey(). Si es demasiado corto, el video será muy rápido y si es demasiado alto, el video será lento (bueno, esta es una forma de mostrar vídeos en cámara lenta). 25 milisegundos estarán bien en casos normales.

```
import numpy as np
import cv2
cap = cv2.VideoCapture('vtest.avi')
while(cap.isOpened()):
  ret, frame = cap.read()
  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  cv2.imshow('frame',gray)
  if cv2.waitKey(1) & 0xFF == ord('q'):
      break
cap.release()
cv2.destroyAllWindows()
```

Nota: asegurarse de que las versiones adecuadas de ffmpeg o gstreamer estén instaladas. A veces, es un dolor de cabeza para trabajar con Capturas de Video, principalmente debido a una instalación incorrecta de ffmpeg / gstreamer.

7.6.- Guardando un vídeo

Para guardar un vídeo que hemos capturado previamente y procesado cuadro por cuadro. Para el caso de las imágenes, sólo necesitamos usar **cv2.imwrite()**. En el caso de los vídeos, sin embargo, es necesario un poco más de trabajo.

Para esto crearemos un objeto **VideoWriter**. Primero, especificamos el nombre que queremos dar al fichero (ej: *output.avi*). Luego tenemos que especificar el código **FourCC** (ver detalles más abajo), el número de cuadros por segundo (*fps*) y el tamaño de cuadro. El último argumento que debemos pasarle es la bandera **isColor**. Si es verdadero, el codificador espera cuadros a color, de lo contrario trabaja con escala de grises.

FourCC es un código de 4 bytes usado para especificar el códec del vídeo. Podemos encontrar una lista de los diferentes códigos disponibles en fourcc.org. Estos códigos dependen de la plataforma que se utilice, con lo cual, dependiendo del sistema operativo que estemos usando puede que algunos funcionen y otros no. En el ejemplo que mostramos más abajo, que ha sido corrido sobre Windows 10, se ha utilizado DIVX.

```
import cv2

cap = cv2.VideoCapture(0)

# Define el codec y crea el objeto VideoWriter
fourcc = cv2.VideoWriter_fourcc(*'DIVX')
out = cv2.VideoWriter('output.avi',fourcc, 20.0, (640,480))

while(cap.isOpened()):
    ret, frame = cap.read()
    if ret==True:
       frame = cv2.flip(frame,0) #invierte el cuadro

       # escribe el cuadro invertido
       out.write(frame)
       cv2.imshow('frame',frame)
       if cv2.waitKey(1) & 0xFF == ord('q'):
           break
    else:
        break

#Libera todo si la tarea ha terminado
cap.release()
out.release()
cv2.destroyAllWindows()
```

7.7.- Operaciones Básicas en Imágenes en Open CV con Python

Casi todas las operaciones están principalmente relacionadas con Numpy más que a Open CV. Conocer al detalle Numpy es bueno para escribir un código mejor optimizado con Open CV.

7.7.1.- Accediendo y Modificando los valores de píxeles

Primero carguemos una imagen a color:

```
import cv2
import numpy as np
img = cv2.imread('trump.jpg')
```

Podemos acceder al valor de un pixel por medio de las coordenadas de su fila y su columna. Para imágenes RGB, regresan una gama de valores entre Azul, Verde y Rojo. Para las imágenes en escala de grises, sólo la intensidad correspondiente es regresada.

```
px = img[100,100]
print px
# acceso solo al pixel azul
blue = img[100,100,0]
print blue
```

Podemos modificar los valores de pixel de la misma forma.

```
img[100,100] = [255,255,255]
print  img[100,100]
```

Nota: el método mencionado anteriormente se usa normalmente para seleccionar una región, por ejemplo, las primeras 5 filas y las últimas 3 columnas de este modo. Para acceder a los píxeles de forma individual, los métodos de Numpy array.item() y array.itemset() se consideran mejores. Pero siempre devuelve un escalar. Así que si deseamos acceder a los valores R,G,B necesitamos llamar array.item() de forma separada para todos.

Un mejor método para acceder al pixel y editarlo:

```
# accessing RED value
img.item(10,10,2)
# modifying RED value
img.itemset((10,10,2),100)
img.item(10,10,2)
```

7.7.3.- Accediendo a las Propiedades de Imagen

Las propiedades de imagen incluyen número de filas, columnas y canales, tipo de dato de imagen, número de píxeles, etc.

Accedemos a la forma de la imagen por medio de img.shape. Este retorna una tupla de números de filas, columnas y canales (si la imagen es a color):

```
print img.shape
```

Nota: si la imagen corresponde a escala de grises, la tupla devuelta contiene sólamente el número de filas y columnas. Por lo tanto, es un buen método para verificar si la imagen cargada está a escala de grises o es una imagen a color.

Accedemos al número total de píxeles por medio de img.size:

```
print img.size
```

El tipo de datos (*datatype*) de la imagen se obtiene por medio de img.dtype:

```
print img.dtype
```

img.dtype es muy importante al momento de la depuración porque un gran número de errores en el código OpenCV-Python son causados por tipos de datos inválidos.

7.7.4.- Dividiendo y Combinando Canales de Imagen

Cuando se necesario los canales R,G,B de una imagen se pueden dividir en sus planos individuales. Luego, podemos combinar nuevamente dichos canales para nuevamente formar una imagen:

```
b,g,r = cv2.split(img)
img = cv2.merge((b,g,r))
```

O

```
b = img[:,:,0]
```

Supongamos que queremos hacer que todos los píxeles rojos valgan cero, no necesitamos dividirlos todos de esta forma y colocarlos igual a cero. Podemos usar indexación Numpy en tanto que es más rápida.

```
img[:,:,2] = 0
```

cv2.split() es una operación costosa (en términos de tiempo), así que sólo utilízala si es necesario. La indexación Numpy es mucho más eficiente y debería usarse siempre que sea posible.

7.7.5.- Realización de bordes para la Imagen (*Padding*)

En caso de que deseemos crear un borde alrededor de una imagen, algo como un marco, se puede usar la función **cv2.copyMakeBorder()**. Pero tiene más aplicaciones para la operación de circonvolución, *zero padding*, etc. Esta función toma los siguientes argumentos:

- **src** – introducir imagen
- **top**, **bottom**, **left**, **right** – ancho de borde en número de píxeles en correspondencia con las direcciones.
- **borderType – Marca que define el tipo de borde a añadir. Puede ser de los siguientes tipos:**
    - **BORDER_CONSTANT** – Añade un borde de color constante. El valor debería ser provisto como el siguiente argumento.
    - **BORDER_REFLECT** – El border será el reflejo de sus elementos, como este: fedcba|abcdefgh|hgfedcb
    - **BORDER_REFLECT_101** or **cv2.BORDER_DEFAULT** – Igual que arriba, pero con un pequeño cambio, como este: gfedcb|abcdefgh|gfedcba
    - **BORDER_REPLICATE** – El último elemento es replicado alrededor, así: aaaaaa|abcdefgh|hhhhhhh
    - **BORDER_WRAP** – No puedo explicarlo, luciría así: cdefgh|abcdefgh|abcdefg

valor – El color del borde si el tipo de borde es cv2.BORDER_CONSTANT

Más abajo hay una muestra del código demostrando todos estos tipos de borde para su mejor entendimiento:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
BLUE = [255,0,0]
img1 = cv2.imread('opencv_logo.png')
replicate = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REPLICATE)
```

```
reflect = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT)
reflect101 = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_REFLECT_101)
wrap = cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_WRAP)
constant= cv2.copyMakeBorder(img1,10,10,10,10,cv2.BORDER_CONSTANT,value=BLUE)
plt.subplot(231),plt.imshow(img1,'gray'),plt.title('ORIGINAL')
plt.subplot(232),plt.imshow(replicate,'gray'),plt.title('REPLICATE')
plt.subplot(233),plt.imshow(reflect,'gray'),plt.title('REFLECT')
plt.subplot(234),plt.imshow(reflect101,'gray'),plt.title('REFLECT_101')
plt.subplot(235),plt.imshow(wrap,'gray'),plt.title('WRAP')
plt.subplot(236),plt.imshow(constant,'gray'),plt.title('CONSTANT')
plt.show()
```

Veamos en el resultado de abajo. (La imagen se expone usando *matplotlib*. Así que los planos del ROJO y el AZUL están intercambiados):

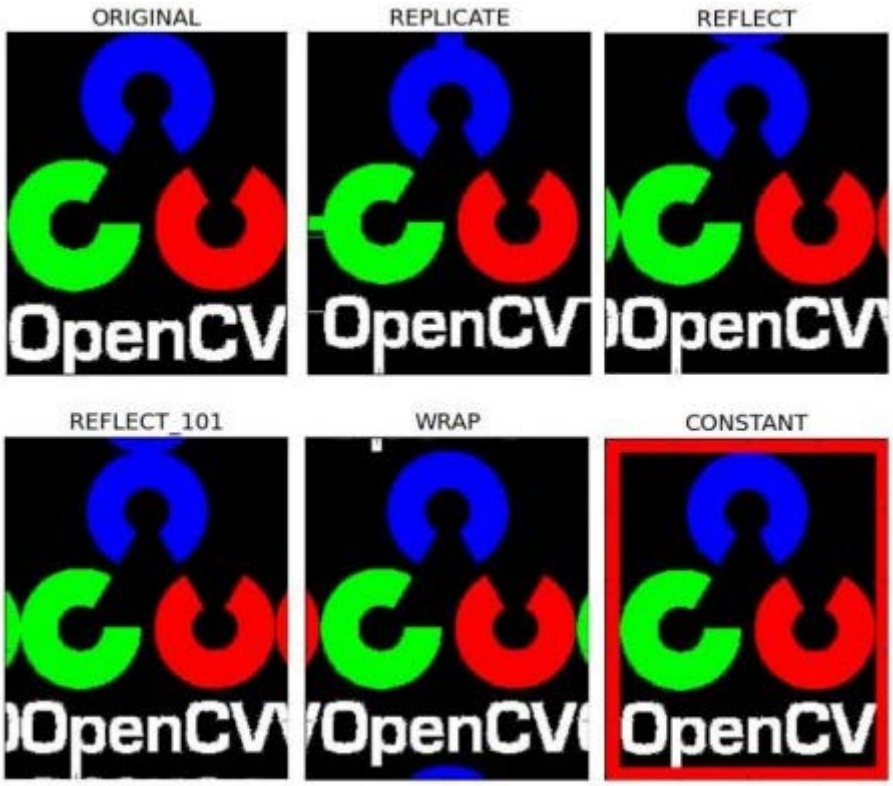


Fig. 49: Ejemplo de realización de bordes.

7.8.- Operaciones Aritméticas en Imágenes Open CV con Python

7.8.1.- Suma de imágenes

Puedes sumar dos imágenes usando la función de Open CV, **cv2.add()** o simplemente por medio de una operación *numpy*, res = img1 + img2. Ambas imágenes deberían tener la misma profundidad y tipo, o la segunda puede ser un valor escalar.

Existe una diferencia entre la suma *Open CV* y la suma *Numpy*. La suma *Open CV* es una operación saturada mientras que la suma *Numpy* es una operación modular.

Por ejemplo, considera este ejemplo:

```
x = np.uint8([250])
y = np.uint8([10])
```

```
print cv2.add(x,y) # 250+10 = 260 => 255
print x+y # 250+10 = 260 % 256 =4
```

7.8.2.- Fusión de imágenes

También es suma de imágenes, pero con diferentes pesos lo cual da una sensación de mezcla o transparencia. Las imágenes se suman mediante esta ecuación:

$$g(x) = (1 - \alpha) f_0(x) + \alpha f_1(x)$$

Variando alfa de 0 a infinito, puedes efectuar una transición suave entre una imagen y la otra.

Aquí tomamos dos imágenes para fusionarlas. A la primera le doy un peso de 0.7 y a la segunda uno de 0.3. cv2.addWeighted() aplico la siguiente ecuación sobre la imagen.

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

Aquí gamma se toma como cero.

```
import cv2

img1 = cv2.imread('python-log.png')
img2 = cv2.imread('pirata.png')
img3=img1[1:568,1:1001,:]

dst = cv2.addWeighted(img3,0.7,img2,0.3,0)

cv2.imshow('dst',dst)
cv2.imwrite('dst.png',dst)
cv2.waitKey(0)

cv2.destroyAllWindows()
```

Fig.50: Muestra de fusión de imágenes.

7.8.3.- Operaciones Bitwise

Esto incluye *bitwise* Y, O, NO y operaciones *XOR*. Son útiles para extraer cualquier parte de la imagen, definir y trabajar con un *ROI* no rectangular, etc. Abajo veremos un ejemplo de cómo cambiar una particular *ROI*.

Supongamos que tratamos de colocar el logo de *OpenCV* sobre una imagen. Si agregamos dos imágenes, cambiará de color. Si la mezclamos, obtenemos un efecto de transparencia. Pero queremos que sea opaco. Si fuera una región rectangular podríamos usar *ROI*. Pero el logo de *aprednderPython* no es una forma rectangular. Así que podemos hacerlo con operaciones *bitwise* como se muestra a continuación:

```
import cv2
# cargamos 2 imagenes
img1 = cv2.imread('dst.png')
img2 = cv2.imread('aprenderpython.png')
# Se quiera poner el logo en el corner izquierdo y por eso creo un ROI
rows,cols,channels = img2.shape
roi = img1[0:rows, 0:cols ]
# Ahora creo una máscara de logotipo y hago su máscara inversa también
img2gray = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)
ret, mask = cv2.threshold(img2gray, 250, 255, cv2.THRESH_BINARY)
mask_inv = cv2.bitwise_not(mask)
# Ahora ponemos oscura el área de logotipo en ROI
img1_bg = cv2.bitwise_and(roi,roi,mask = mask)
# Tomamos solamente la región del logotipo de la imagen del logotipo
img2_fg = cv2.bitwise_and(img2,img2,mask = mask)
# Ponemos el logotipo en ROI y modificamos la imagen principal
dst = cv2.add(img1_bg,img2_fg)
img1[0:rows, 0:cols ] = dst
cv2.imshow('res',img1)
cv2.waitKey(0)
cv2.destroyAllWindows()
cv2.imshow('res',roi )
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Fig.51: Muestra de operación bitwise.

7.9.- Procesamiento de imágenes.

7.9.1.- Cambiando el Espacio de Color Open CV con Python

Contamos con más de 150 métodos de conversión de espacio de color en Open CV. Los mayormente usados, RGB <-> Gris, RGB <-> HSV.

Para conversión de color, usamos la función

cv2.cvtColor(input_image, flag) donde flag determina el tipo de conversión.

Para la conversión de RGB -> Gris usamos los indicadores cv2.COLOR_BGR2GRAY. De forma similar para RGB -> HSV. Usamos el indicador cv2.COLOR_BGR2HSV. Para obtener otros indicadores, sólo ejecuta los siguientes comandos en tu terminal *Python*:

```
import cv2

flags = [i for i in dir(cv2) if i.startswith('COLOR_')]

print flags
```

Para HSV, el rango de tonos es [0,179], el rango de saturación es [0,255] y el rango de valor es [0,255]. Los distintos programas usan distintas escalas. Por lo tanto, si estamos comparando los valores de *Open CV* con ellos, necesitamos normalizar estos rangos.

7.9.2.- Transformaciones geométricas de imágenes con Open CV

Una transformación geométrica de una imagen es una transformación en su sistema de coordenadas. Open CV proporciona dos funciones de transformación, **cv2.warpAffine** y **cv2.warpPerspective**, con las que se pueden realizar todo tipo de transformaciones.

- *Cv2.warpAffine* toma una matriz de transformación 2×3 mientras
- *Cv2.warpPerspective* toma una matriz de transformación 3×3 como entrada.

7.9.3.- Redimensionalización

Para cambiar el tamaño de una imagen, Open CV viene con la función **cv2.resize()**. El tamaño deseado de la imagen final se puede especificar manualmente, o se puede indicar especificando un factor de escala. La función usa diferentes métodos de interpolación, siendo los más usados: **cv2.INTER_AREA**, para contraer la imagen y **cv2.INTER_CUBIC** (suave) & **cv2.INTER_LINEAR**, para acercar la imagen. El método de interpolación utilizado por defecto es **cv2.INTER_LINEAR**, que sirve para cualquier cambio de tamaño que se desee realizar. Cualquiera de los métodos mostrados a continuación, se pueden utilizar para cambiar el tamaño de una imagen:

```
import numpy as np
import cv2

img = cv2.imread('MotherofDragons.jpg')

#Indicando el factor de escala
res = cv2.resize(img,None,fx=2, fy=2, interpolation = cv2.INTER_CUBIC)

# Indicando manualmente el nuevo tamaño deseado de la iamgen
height, width = img.shape[:2]
res = cv2.resize(img,(2*width, 2*height), interpolation = cv2.INTER_CUBIC)
```

7.9.4.- Traslación

Una traslación es el desplazamiento de la posición de un objeto. Si se conoce la magnitud del desplazamiento (t_x, t_y) en las direcciones *x* e *y*, respectivamente, se puede escribir la matriz de transformación ***M*** como:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

La matriz ***M*** se puede crear utilizando la función **np.float32** de la librería Numpy. Esta matriz luego se pasa como argumento a la función **cv2.warpAffine().** Observar el siguiente ejemplo para un desplazamiento de (210, 20).

Nota: el tercer argumento de la función **cv2.warpAffine ()** corresponde al tamaño de la imagen de salida, que debe estar en forma de (anchura, altura). Recordar, ***width*** = número de columnas y ***height*** = número de filas.

Fig.52: Traslación de imagen.

7.9.5.- Rotación

La rotación de una imagen, en un cierto ángulo θ, se logra aplicando la siguiente matriz de transformación:

$$M = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

Sin embargo, Open CV permite además personalizar más la rotación multiplicando por un factor de escala. Por otro lado, también permite cambiar el centro de rotación. La matriz de transformación modificada, con estas dos nuevas opciones, tiene la forma:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha)\cdot center.y \end{bmatrix}$$

donde

$$\alpha = scale \cdot \cos\theta, \\ \beta = scale \cdot \sin\theta$$

Para encontrar esta matriz de transformación, Open CV proporciona la función **cv2.getRotationMatrix2D**. Obsevemos a continuación un ejemplo en el cual se gira la imagen 45 grados con respecto al centro sin aplicar ningún factor de escala.

```
img = cv2.imread('MotherofDragons.jpg',0)
rows,cols = img.shape
```

```
M = cv2.getRotationMatrix2D((cols/2,rows/2),45,1)
dst = cv2.warpAffine(img,M,(cols,rows))
```

Fig.53: Rotación de imagen.

7.9.6.- Transformación Afín

En la transformación afín todas las líneas paralelas en la imagen original seguirán siendo paralelas en la imagen de salida. Para encontrar la matriz de transformación, necesitamos tres puntos de la imagen de entrada y sus ubicaciones correspondientes en la imagen de salida. Luego **cv2.getAffineTransform** creará una matriz 2×3 que se pasará a **cv2.warpAffine**.

Comprobemos a continuación el ejemplo, y observemos los puntos que he seleccionado (marcados en color verde):

```
import numpy as np
import matplotlib.pyplot as plt #carga la librería para graficar
import cv2

img = cv2.imread('cuadricula.png')
rows,cols,ch = img.shape

pts1 = np.float32([[100,400],[400,100],[100,100]])
pts2 = np.float32([[50,300],[400,200],[80,150]])


M = cv2.getAffineTransform(pts1,pts2)
dst = cv2.warpAffine(img,M,(cols,rows))

plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

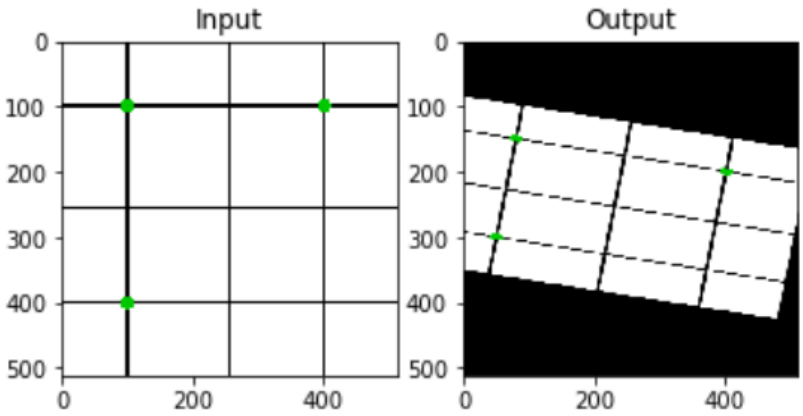


Fig.54: Transformación afín de imagen.

7.9.7.- Transformación de Perspectiva

Para realizar una transformación de perspectiva es necesario especificar una matriz de 3×3. A continuación de aplicar este tipo de transformación, las líneas rectas permanecerán rectas. Para generar la matriz de 3×3 se necesita indicar cuatro puntos sobre la imagen de inicial y los correspondientes puntos sobre la imagen resultante. Tres de los cuatro puntos, tienen que ser no-colineales. De esta forma la matriz de transformación se puede generar utilizando la función **cv2.getPerspectiveTransform**. Luego, para aplicar la transformación, se utiliza **cv2.warpPerspective** teniendo en cuenta la matriz de 3×3 generada con la función anterior.

A continuación, un ejemplo:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2

img = cv2.imread('sudoku.png')
rows,cols,ch = img.shape

pts1 = np.float32([[56,65],[368,52],[28,387],[389,390]])
pts2 = np.float32([[0,0],[300,0],[0,300],[300,300]])

M = cv2.getPerspectiveTransform(pts1,pts2)

dst = cv2.warpPerspective(img,M,(300,300))

plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
plt.show()
```

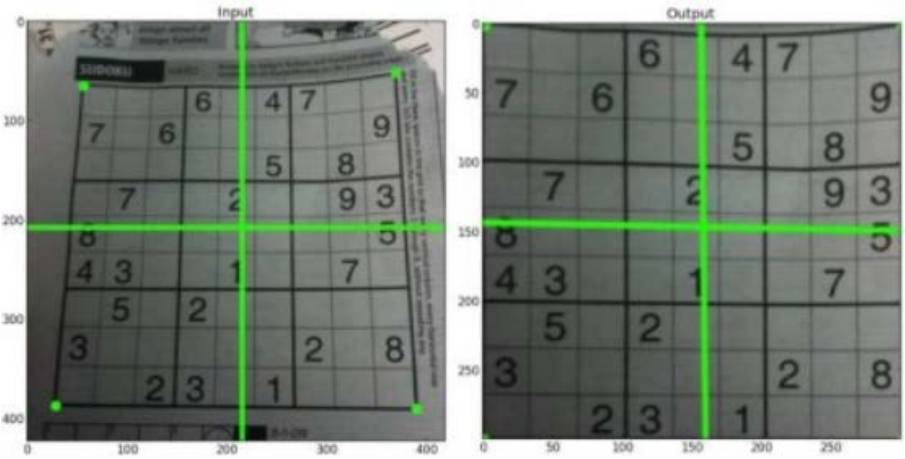


Fig.55: Transformación de perspectiva.

7.9.8.- Umbralización en Open CV con Python

7.9.8.1.- Umbralización Simple

Consistente en que, si el valor del pixel es mayor al valor del umbral, se le asigna un valor (puede ser blanco), de otro modo se le asigna otro valor (puede ser negro). La función es **cv2.threshold**. El primer argumento es la imagen fuente, que en lo posible, debería encontrarse en escala de grises. El segundo argumento es el valor del umbral que se usa para calificar los valores de pixeles. El tercer argumento es el **maxVal** el cual representa el valor dado si el valor del pixel es mayor que (a veces menor que) el valor del umbral. *Open CV* provee diferentes estilos de umbralización y se decide por medio del cuarto parámetro de la función. Los distintos tipos son:

- THRESH_BINARY
- THRESH_BINARY_INV
- THRESH_TRUNC
- THRESH_TOZERO
- THRESH_TOZERO_INV

La documentación explica claramente para qué funciona cada uno.

```
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
```

En este ejemplo se indican parámetros:

- El primer parámetro (img) es la imagen en escala de grises.
- El segundo parámetro (127) es el valor de umbral.
- El tercer parámetro (255) es el valor máximo si el valor del pixel es mayor.
- El cuarto parámetro es el tipo de umbralización.

Se obtienen dos salidas. La primera es un **retval**, que veremos más adelante. La segunda es nuestra **imagen umbralizada**.

Código:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('musica.png',0)
```

```
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)

titles                                  =                                  ['Original
Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
miArray = np.arange(6)
for i in miArray:
  plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
  plt.title(titles[i])
  plt.xticks([]),plt.yticks([])

plt.show()
```

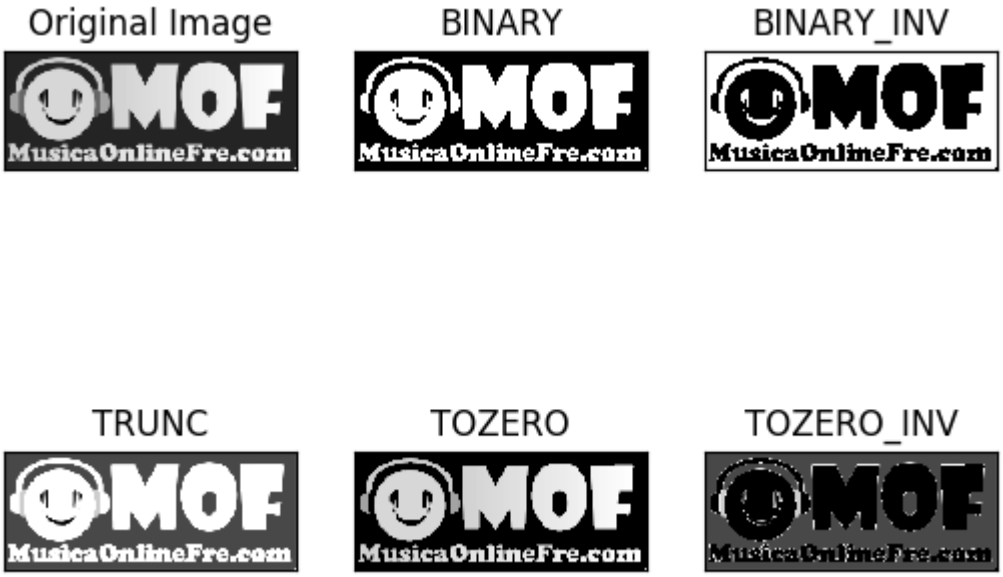


Fig.56: Umbralización y sus diferentes tipos.

7.9.8.2.- Umbralización Adaptativa

En la sección previa, usamos un valor global como valor umbral. Pero puede no ser bueno en todos los casos donde las imágenes difieren en cuanto a condiciones de luz en distintas áreas. En ese caso, utilizamos la umbralización adaptativa. En esta, el algoritmo calcula el umbral para una pequeña región de la imagen. Así que obtenemos diferentes umbrales para distintas regiones de la misma imagen. Y nos da mejores resultados para imágenes con iluminación variante.

Posee tres parámetros “especiales” de entrada y sólo un argumento de salida.

Método Adaptativo – Decide cómo el valor de umbralización es calculado.

- ADAPTIVE_THRESH_MEAN_C : el valor umbral es equivalente al valor del área vecina.
- ADAPTIVE_THRESH_GAUSSIAN_C : en este caso el valor umbral es la suma de los pesos de los valores vecinos donde dichos valores correspondían a pesos de una ventana gaussiana.

Block Size – Decide el tamaño del área vecina.

C – Es sólo una constante que se sustrae del cálculo del medio o el peso del medio calculado.

El fragmento de código expresado abajo compara la umbralización global con la adaptativa para una imagen de iluminación variante:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('anna-min.jpg',0)
img = cv2.medianBlur(img,5)

ret,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,\
 cv2.THRESH_BINARY,11,2)
th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
 cv2.THRESH_BINARY,11,2)

titles = ['Original Image', 'Global Thresholding (v = 127)',
 'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]
miArray = np.arange(4)
for i in miArray:
 plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

Imagen original a color

Resultado

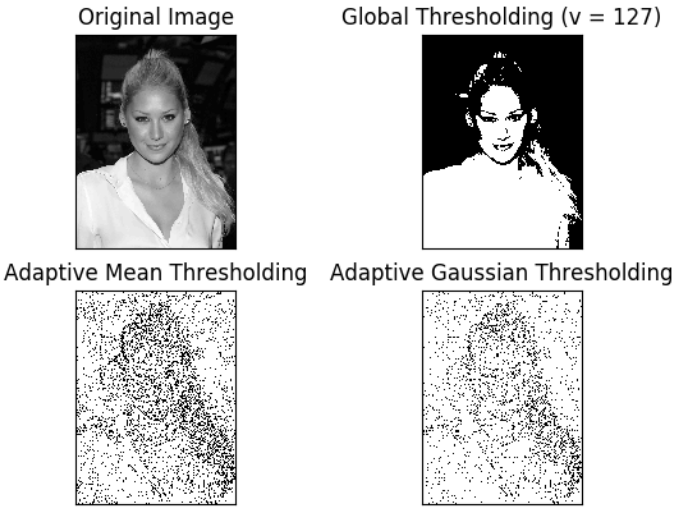


Fig.57: Ejemplo de umbralización adaptativa.

7.9.9.- La Binarización de Otsu

En la sección anterior, se ha visto que había un segundo parámetro denominado **retVal**. Su uso toca cuando usamos la **Binarización de Otsu**. Así que, ¿qué es? En la umbralización global, utilizamos un valor arbitrario como umbral, ¿correcto? Así, ¿cómo podemos saber si el valor que hemos escogido es bueno o no? La respuesta es, mediante el método de prueba y error. Pero considera una **imagen bimodal** (en pocas palabras, una imagen bimodal es una imagen cuyo histograma posee dos picos). Para esa imagen, podemos tomar un valor aproximado entre esos dos picos como el valor umbral, ¿correcto? Eso es lo que hace la **binarización de Otsu**. En pocas palabras, se calcula de forma automática un valor de umbral desde el histograma de la imagen bimodal. (Para imágenes que no son bimodales, la binarización no será precisa).

Para esto, usamos la función cv2.threshold(), pero con un indicador adicional, cv2.THRESH_OTSU. Para el valor umbral, sólo usamos cero**.** Luego el algoritmo encuentra el valor umbral óptimo y lo regresa como la segunda salida, retVal. Si la umbralización de Otsu no se usa, retVal es igual al valor de umbral que usamos.

Veamos a continuación el ejemplo. La imagen de entrada contiene mucho ruido. En el primer caso, aplicamos la umbralización global para un valor de 127. En el segundo caso, aplicamos la umbralización de Otsu de forma directa. En el tercer caso, filtramos la imagen con *kernel* gaussiano 5×5 para eliminar el ruido, luego aplicamos la umbralización de Otsu. Observemos como el filtro que elimina el ruido mejorando los resultados.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('anna-min.jpg',0)

# global thresholding
ret1,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)

# Otsu's thresholding
ret2,th2 = cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# Otsu's thresholding after Gaussian filtering
blur = cv2.GaussianBlur(img,(5,5),0)
ret3,th3 = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# plot all the images and their histograms
images = [img, 0, th1, img, 0, th2, blur, 0, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',
'Original Noisy Image','Histogram',"Otsu's Thresholding",
'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]
miArray = np.arange(3)
for i in miArray:
  plt.subplot(3,3,i*3+1),plt.imshow(images[i*3],'gray')
  plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
  plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
  plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
  plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2],'gray')
  plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()
```

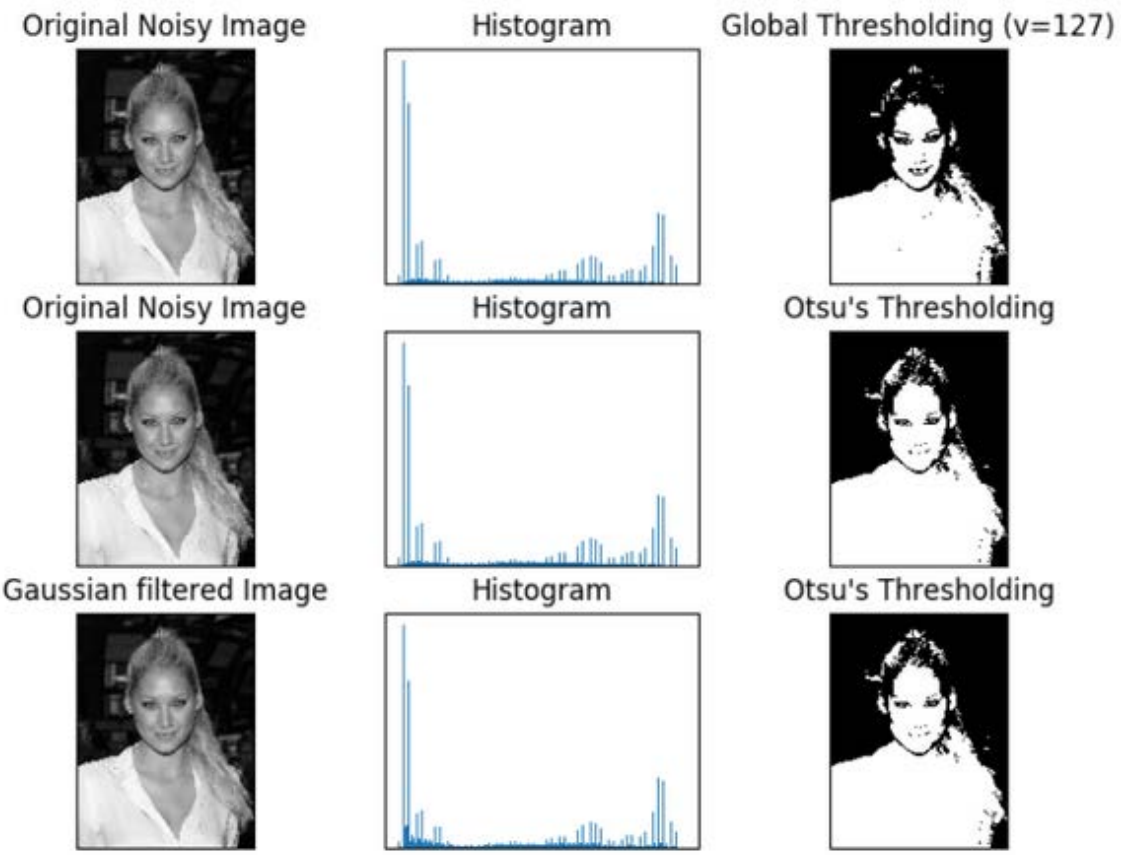


Fig.58: Ejemplo de binarización Otsu.

7.10.- Suavizando Imágenes con Open CV

En el proceso de suavizado (smoothing en inglés) las imágenes acaban difuminadas. Una imagen se ve más nítida o más detallada si somos capaces de percibir todos los objetos y sus formas, correctamente, en ella. Mientras menos definidos son los bordes de las formas individuales dentro de la imagen, más difícil es distinguir una forma de otra. Esto es precisamente lo que hace el suavuzado, es decir, reducir el contenido de los bordes de las formas en la imagen, suavizando así la transición entre los distintos colores. Para esto se utilizan filtros de imágenes.

7.10.1.- Convolución 2D (Filtrado de imágenes)

Al igual que las señales unidimensionales, las imágenes también se pueden ser filtrar con varios tipos de filtros, como por ejemplo, filtros pasa bajo (FPB), filtros pasa alto (FPA), filtros pasa banda, etc. Mientras que un FPB ayuda a eliminar el ruido en la imagen o a difuminar la imagen, un FPA ayuda a encontrar los bordes en una imagen. La función **cv2.filter2D(),** disponible en Open CV, nos permite aplicar una convolución entre un kernel dado y una imagen. Un ejemplo de un kernel es un filtro para promediar, como el FPB de 5×5 que se muestra a continuación:

$$K = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

El filtrado de una imagen dada con el kernel anterior funciona de la siguiente forma: sobre cada pixel de la imagen se centra una ventana de 5×5. Los píxeles contenidos en esta ventana se suman y se dividen por 25, y el valor resultante es asignado al pixel. Esto equivale a calcular el promedio del valor de los píxeles que caen en la ventana de 5×5. La operación se repite sobre todos los píxeles de la imagen, dando lugar a la imagen filtrada. El siguiente código genera el kernel ***K*** y lo aplica a una imagen:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('TajMahal.jpg')

#Crea el kernel
kernel = np.ones((5,5),np.float32)/25

#Filtra la imagen utilizando el kernel anterior
dst = cv2.filter2D(img,-1,kernel)

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dst),plt.title('Promediada')
plt.xticks([]), plt.yticks([])
plt.show()
```

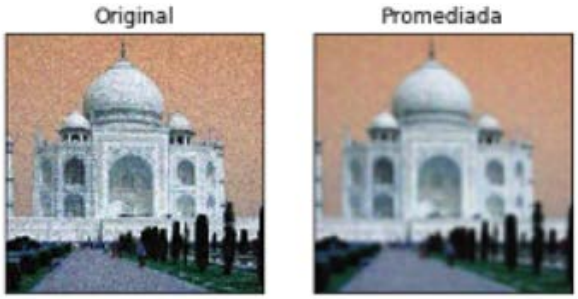


Fig.59: Ejemplo de filtro de convolución 2D.

7.10.2.- Difuminando imágenes (alisando o suavizando imágenes)

Como ya se hemos comentado, el difuminado de la imagen se consigue convolucionando la imagen con un kernel de filtro pasa bajo (FPB). Los FPB eliminan el contenido de alta frecuencia (ej.: ruido y bordes) de la imagen, lo que resulta, en general, en imágenes con bordes más borrosos. No obstante, también hay filtros que eliminan el ruido con poco efecto sobre los bordes. Tres de las técnicas de difuminado más utilizadas, que vienen implementados en Open CV son: el promediado, los filtros gaussianos y los filtros de mediana. A continuación, se describen estos tres tipos de FPB.

7.10.2.1.- Promedio

Este es justo el caso explicado más arriba, en el caso de los filtros personalizados. El promedio se realiza convolucionando la imagen con un filtro de caja normalizado. De este modo, este filtro toma el promedio de todos los píxeles bajo el área del kernel y reemplaza al elemento central por este promedio. Una manera alternativa de hacer esto es mediante las funciones **cv2.blur()** o **cv2.boxFilter()**. Al utilizar estas funciones tenemos que especificar el ancho y la altura del kernel. Un filtro de caja normalizado 3×3 se vería así:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Si no deseamos utilizar un filtro de caja normalizado, utilice **cv2.boxFilter()** y poner el argumento **normalize = False** en la función.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('TajMahal.jpg')

blur = cv2.blur(img,(3,3))

plt.subplot(121),plt.imshow(img),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(blur),plt.title('Difuminada')
plt.xticks([]), plt.yticks([])
plt.show()
```

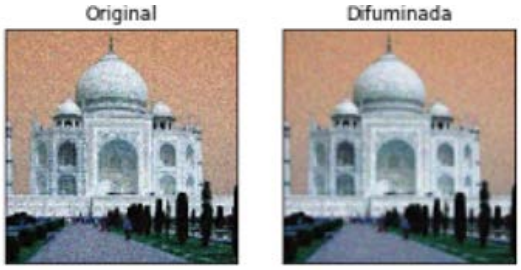


Fig.60: Ejemplo de filtro de promedio.

7.10.2.2.- Filtro Gaussiano

En este enfoque, en lugar de un filtro de caja que consta de coeficientes iguales, se utiliza un núcleo gaussiano. Esto se hace con la función, **cv2.GaussianBlur()**. Como parámetros de entrada se tienen que pasar el ancho y la altura del kernel, que debe ser positivo e impar. También hay que especificar la desviación estándar en las direcciones X e Y, sigmaX y sigmaY, respectivamente. Este filtrado es muy eficaz para eliminar el ruido gaussiano de la imagen.

Nota: Si sólo se especifica sigmaX, sigmaY se toma como igual a sigmaX. Si ambas se pasan como ceros, se calculan a partir del tamaño del núcleo.

El código anterior puede ser modificado para aplicar el filtro Gaussiano, sustituyendo *blur* por:

```
blur = cv2.GaussianBlur(img,(5,5),0)
```

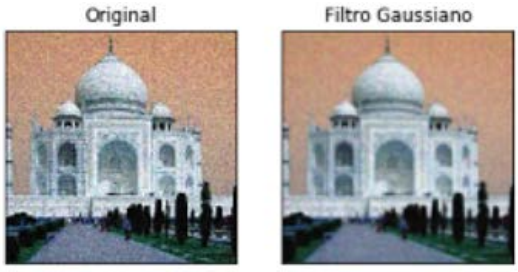


Fig.61: Ejemplo de filtro de Gaussiano.

7.10.2.3.- Filtro de Mediana

En este filtro se calcula la mediana de todos los píxeles bajo la ventana del kernel y el píxel central se sustituye con este valor mediano. Esto es muy efectivo para eliminar el ruido conocido como ruido de sal y pimienta. Open CV dispone de la función **cv2.medianBlur()** para aplicar este tipo de filtro a una imagen. Al igual que en el filtro Gaussiano, el tamaño del kernel en el filtro de mediana tiene que ser un número entero impar positivo.

Un ejemplo del desempeño de este tipo de filtro se muestra a continuación. Sólo debemos sustituir en el código anterior, *blur* por:

```
median = cv2.medianBlur(img,5)
```

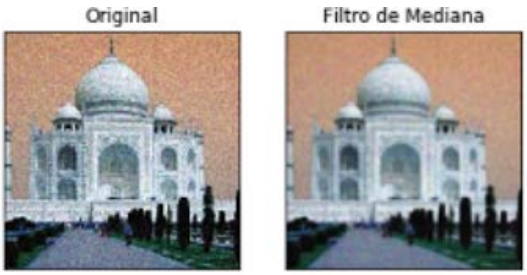


Fig.62: Ejemplo de filtro de mediana.

7.11.- Transformaciones morfológicas

Las transformaciones morfológicas comprenden operaciones simples basadas en la forma de la imagen, aplicado normalmente a imágenes binarias. Necesita dos entradas, una es nuestra imagen original, la segunda se llama elemento estructurante o núcleo (kernel) que decide la naturaleza de la operación. Dos operadores morfológicos básicos son Erosión y Dilatación. Posee, formas variantes como Apertura, Cierre, Gradiente, etc, también entran en juego. A continuación, algunas de estas transformaciones, apoyándonos en la siguiente imagen binaria:

Fig.63: Imagen de ejemplo para transformaciones morfológicas.

7.11.1.- Erosión

Parecido a la convolución 2D, en el proceso de erosión se recorre un kernel a través de la imagen. Un píxel de la imagen original (1 ó 0) sólo se considerará 1 si todos los píxeles que caen dentro de la ventana del kernel son 1, de lo contrario se erosiona (se hace a cero). Por tanto, todos los píxeles cerca de los bordes de los objetos en la imagen se descartarán en función del tamaño del kernel. Como consecuencia, el grueso o el tamaño de los objetos en primer plano disminuye o, en otras palabras, la región blanca disminuye en la imagen. Esta fución es útil para eliminar pequeños ruidos blancos, separar dos objetos conectados, etc. Un ejemplo donde se utiliza un kernel de 7×7 formado por unos:

```
import cv2
import numpy as np

img = cv2.imread('A.png',0)
kernel = np.ones((7,7),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 1)
```

Fig.64: Imagen de ejemplo del proceso de Erosión.

7.11.2.- Dilatación

Contrario a la erosión. En este, un elemento de píxel es '1' si al menos un píxel de la imagen de los que caen dentro de la ventana del kernel es '1'. Por lo tanto, la dilatación aumenta el tamaño de los objetos de primer plano, es decir, la región blanca. Normalmente, en casos como la eliminación del ruido, a la dilatación le sigue la erosión. El motivo es que aunque la erosión elimina los ruidos blancos también encoge los objetos. Por tanto, para recuperar el tamaño inicial, este se dilata. La transformación de dilatación también es útil para unir partes rotas de un objeto. A continuación un ejemplo de cómo funciona la dilatación:

```
dilatacion = cv2.dilate(img,kernel,iterations = 1)
```

donde el kernel es el mismo que se ha utilizado en el ejemplo de Erosión.

Fig.65: Imagen de ejemplo del proceso de Dilatación.

7.11.3.- Apertura

La apertura es simplemente otro nombre para erosión seguida de dilatación. Como se ha comentado anteriormente, es útil para eliminar el ruido. En este caso se utiliza la función, **cv2.morphologyEx()**. Ver el ejemplo a continuación:

```
apertura = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
```

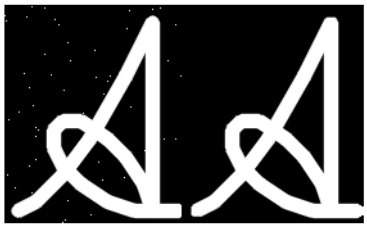


Fig.66: Imagen de ejemplo del proceso de Apertura.

7.11.4.- Cierre

El Cierre es el contrario de Apertura, es decir, dilatación seguida de erosión. Útil para cerrar pequeños agujeros dentro de los objetos de primer plano, o pequeños puntos negros en el objeto. Ver un ejemplo a continuación:

```
cierre = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```

7.11.5.- Gradiente Morfológico

Diferencia entre la dilatación y la erosión de una imagen. El resultado se verá como el contorno del objeto.

```
gradiente = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
```

Fig.67: Imagen de ejemplo del proceso de Gradiente.

En los ejemplos se han creado manualmente elementos estructurantes (kernels) de forma rectangular (utilizando np.ones()). Sin embargo, en algunos casos, es necesario crear núcleos elípticos / circulares. Para este propósito, Open CV tiene la función cv2.getStructuringElement().

7.12.- Gradiente de Imágenes

El gradiente de una imagen mide cómo cambia la imagen con referencia al color o intensidad. La magnitud del gradiente nos indica la rapidez con la que la imagen cambia, mientras que la dirección del gradiente nos indica la dirección en la que la imagen está cambiando más rápidamente. Matemáticamente, el gradiente se define por las derivadas parciales de una función dada (intensidad en el caso imágenes) a lo largo de las direcciones *X* e *Y*.

Los puntos donde la derivada es máxima (o mayor que cierto umbral) corresponden a cambios de intensidad grandes, normalmente asociados a los bordes de los objetos en la imagen. Por tanto, este operador resulta útil para encontrar los bordes de las formas dentro de una imagen.

En Open CV existen tres tipos de filtros de gradiente (o filtros pasa altos), estos son: Sobel, Scharr y Laplaciano.

7.12.1.- Derivadas Sobel y Scharr

A la hora de calcular el gradiente de una imagen, los operadores Sobel y Scharr no son más que aproximaciones, más o menos precisas. Ambos se definen a través de kernels cuadrados como los mostrados en los filtros pasa bajos.

Sobel: el operador Sobel aplica un alisamiento Gausssiano común y estima las derivadas parciales a lo largo de *X* e *Y*. Utilizando los argumentos *yorder* y *xorder*, se puede especificar la dirección de las derivadas a tomar, vertical u horizontal, respectivamente. También se puede especificar el tamaño del kernel utilizando el argumento *ksize*. Cuando *ksize* = -1, se utiliza el kernel Scharr 3×3 que da mejores resultados que el kernel Sobel 3×3.

7.12.2.- Derivadas Laplacianas

Este operador calcula el Laplaciano de la imagen, dado por la relación:

$$\Delta src = \frac{\partial^2 src}{\partial x^2} + \frac{\partial^2 src}{\partial y^2}$$

donde cada derivada se determina utilizando el operador Sobel. Por ejemplo, si *ksize* = 1, el kernel que se utiliza para filtrar es:

$$kernel = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

El código muestra todos los operadores en un solo diagrama. Todos los kernels son de tamaño 3×3.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('cebra.jpg',0)

laplacian = cv2.Laplacian(img,cv2.CV_64F)
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=3)
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=3)

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplaciano'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel X'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])

plt.show()
```

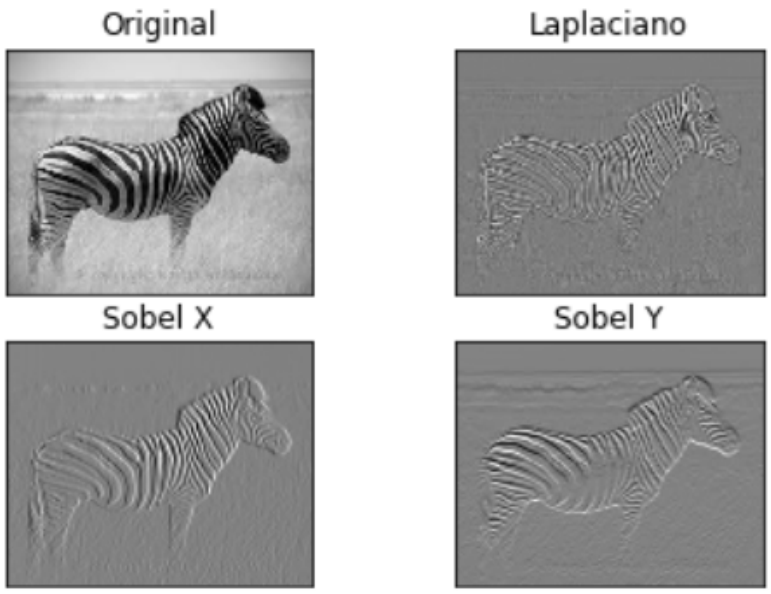


Fig.68: Imágenes de ejemplo del proceso de derivadas de gradiente.

Notar que mientras el filtro Sobel no logra detectar las rayas horizontales a lo largo de la dirección *X*, el filtro Sobel *Y* no puede detectar las rayas verticales. Por otro lado, las rayas diagonales son visibles utilizando cualquiera de los dos filtros dado que estas tienen las dos componentes, vertical y horizontal.

Un punto importante a tener en cuenta es la profundidad de la imagen de salida. En el ejemplo anterior se ha utilizado *cv2.CV_64F* que muestra la imagen en la escala de grises. Si se quisiere visualizar la imagen en blanco y negro entonces se debe sustituir, en el código anterior, *cv2.CV_64F* por *cv2.CV_8U*. Con este cambio, el resultado será el siguiente:

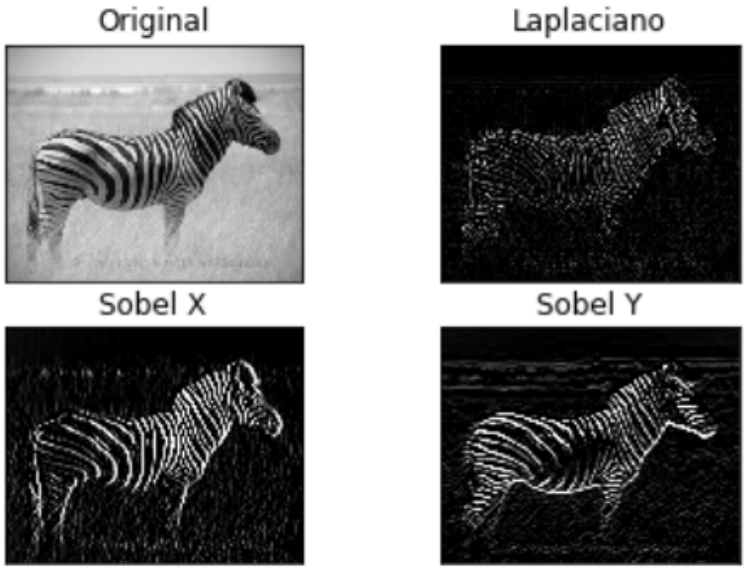


Fig.69: Imágenes de ejemplo del proceso de derivadas de gradiente vs profundidad de bits.

Si se comparamos con detalle estos resultados con los de más arriba se podremos ver que, en el caso de las imágenes en blanco y negro, algunos bordes han desaparecido. Esto es debido a que el gradiente de la transición de blanco a negro tiene valor positivo, mientras que la transición de negro a blanco tiene valor negativo. Luego, al utilizar *cv2.CV_8U* todas las transiciones negativas se hacen cero y por lo tanto no son detectadas por los filtros. Como resultado, algunos bordes de la imagen no se detectan.

Para evitar este problema y detectar ambos bordes, manteniendo la salida final en blanco y negro, la mejor opción es mantener el tipo de datos de salida en algunas formas superiores, como *cv2.CV_16S*, *cv2.CV_64F* etc. tomar su valor absoluto y luego convertirlo de nuevo a *cv2.CV_8U*. El código a continuación muestra este procedimiento para un filtro Sobel horizontal y las diferencias en los resultados.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('whitebox.png',0)

# Utilizando cv2.CV_8U
sobelx8u = cv2.Sobel(img,cv2.CV_8U,1,0,ksize=5)

#Utilizando cv2.CV_64F. Luego toma el valor absoluto y hace la conversión a
cv2.CV_8U
sobelx64f = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)
abs_sobel64f = np.absolute(sobelx64f)
sobel_8u = np.uint8(abs_sobel64f)

plt.subplot(1,3,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(1,3,2),plt.imshow(sobelx8u,cmap = 'gray')
plt.title('Sobel CV_8U'), plt.xticks([]), plt.yticks([])
plt.subplot(1,3,3),plt.imshow(sobel_8u,cmap = 'gray')
plt.title('Sobel abs(CV_64F)'), plt.xticks([]), plt.yticks([])
```

```
plt.show()
```

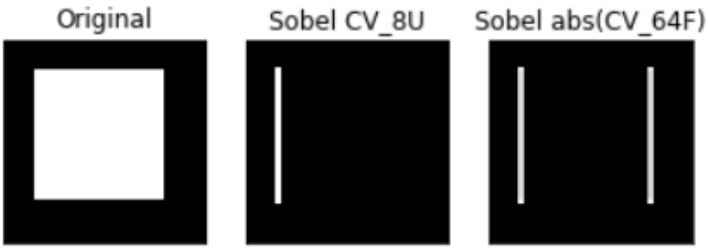


Fig.70: Imágenes de ejemplo del proceso de derivadas de gradiente vs. tipo de datos.

7.13.- Algoritmo de Canny

El algoritmo de Canny es un método de detección de bordes muy utilizado, desarrollado por John F. Canny en 1986. El algoritmo consta de varias etapas, cada una de las cuales se explica a continuación.

7.13.1.- Reducción de ruido

Debido a que la detección de bordes es un proceso que depende del ruido en la imagen, el primer paso es eliminarlo o reducirlo lo más que se pueda. Para esto, el algoritmo utiliza un filtro Gaussiano 5×5, como los vistos anteriormente.

7.13.2.- Encontrando el gradiente de intensidad de la imagen

Una vez que la imagen ha sido suavizada con el filtro Gaussiano, se calcula el gradiente de la misma. Para esto filtramos la imagen nuevamente, esta vez utilizando un kernel Sobel en la dirección horizontal (Gx) y la dirección vertical (Gy). A partir de estas dos imágenes, como resultado de aplicar los dos kernels Gx y Gy, se pueden encontrar los bordes y la dirección del gradiente en cada píxel de la imagen original:

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

donde *Edge_Gradient* es el valor del gradiente en los bordes de los objetos de la imagen y *Angle* indica la dirección perpendicular a los bordes. El valor de este ángulo se redondea a uno de los cuatro ángulos que representan la dirección vertical, la horizontal y las dos diagonales (0°, 45°, 90° y 135°).

7.13.3.- Supresión de falsos máximos

Esta técnica se utiliza para afinar los bordes encontrados en el paso anterior. Consiste en escanear la imagen para eliminar los píxeles que no formen parte de los bordes. Para esto se compara el valor de cada píxel con sus vecinos cercanos en la dirección del gradiente (perpendicular al borde). Si el valor del píxel en cuestión es mayor que sus píxeles vecinos, entonces este se considera un máximo local y el algoritmo lo acepta. De lo contrario, si el píxel resulta no ser un máximo local, entonces es eliminado. El resultado final será una imagen con bordes muy finos.

7.13.4.- Umbral de histéresis

El procedimiento anterior logra determinar los píxeles que conforman los bordes con bastante precisión. Sin embargo, aún pueden quedar algunos píxeles provenientes del ruido o de variaciones en los colores de la imagen. En esta cuarta etapa se decide cuáles píxeles pertenecen realmente a bordes y cuáles no. Para ello, se deben fijar dos valores de umbral, *minVal* y *maxVal*. Los píxeles con gradientes de intensidad mayores que *maxVal* se aceptarán como pertenecientes a los bordes, mientras que los menores que *minVal* se descartarán. Los píxeles correspondientes a bordes con valores de gradientes que se encuentren entre estos dos umbrales se etiquetan como píxeles débiles. Estos últimos se aceptarán o no, dependiendo de su conectividad. Si están conectados a píxeles "fuertes", se consideran parte de los bordes; de lo contrario, se descartan también. Para entender mejor este procedimiento, veamos el siguiente ejemplo:

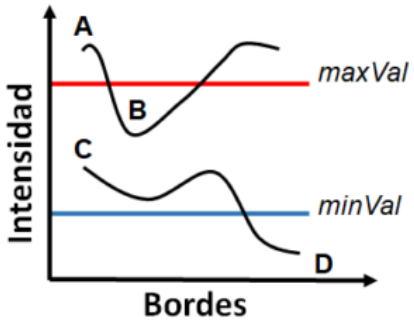


Fig.71: Ejemplo de umbral de histéresis.

El gráfico muestra el valor de la intensidad de los píxeles que conforman los bordes. En este caso, el píxel A se aceptará como parte del borde dado que su valor supera el umbral *maxVal*, mientras que el píxel D se descartará por tener un valor inferior a *minVal*. Por otro lado, los píxeles B y C son considerados débiles por encontrarse entre los dos valores umbrales. Sin embargo, B se aceptará como parte de un borde, mientras que C no. La razón de esto, es que B está conectado a A, que es un píxel fuerte, pero C sólo está conectado a píxeles débiles o descartados.

7.13.5.- Canny algoritmo en Open CV

Todos los pasos del algoritmo de Canny, están contenidos en la función en Open CV: **cv2.Canny()**. El primer argumento es la imagen de entrada, mientras que el segundo y tercer argumento son *minVal* y *maxVal*, respectivamente. El cuarto argumento es *aperture_size*, que no es más que el tamaño del kernel Sobel utilizado para buscar gradientes de imagen (por defecto es 3). El último argumento es *L2gradient* que especifica la ecuación para encontrar la magnitud del gradiente. Si es verdadero, utiliza la ecuación mencionada anteriormente que es más exacta, de lo contrario utiliza la siguiente función (que viene desactivada por defecto):

$$Edge\_Gradient\ (G) = |G_x| + |G_y|$$

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi.jpg',0)
bordes = cv2.Canny(img,180,260)

plt.subplot(121),plt.imshow(img,cmap = 'gray')
```

```
plt.title('Imagen original'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(bordes,cmap = 'gray')
plt.title('Bordes de la Imagen'), plt.xticks([]), plt.yticks([])

plt.show()
```

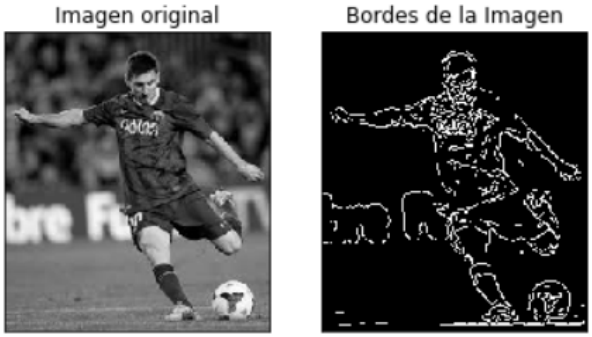


Fig.72: Ejemplo del algoritmo de Canny.

7.14.- Contornos

7.14.1.- ¿Qué es un contorno?

Un contorno es una curva que une todos los puntos continuos en una imagen (a lo largo de los bordes), con el mismo color o intensidad. Los contornos son útiles para el análisis de formas y para la detección y reconocimiento de objetos. Algunas consideraciones generales a tener en cuenta:

- Para una mayor precisión lo mejor es utilizar imágenes binarias. Así que antes de encontrar los contornos, es recomendable aplicar cierto umbral o utilizar el algoritmo de Canny para la detección de bordes.
- La función **findContours** modifica la imagen de origen. Por lo tanto, si desea conservar la imagen original incluso después de encontrar contornos, esta se debe almacenar en una variable distinta.
- En OpenCV, encontrar contornos es como encontrar objetos blancos de fondo negro. Así que recuerde, el objeto a ser encontrado debe ser blanco y el fondo debe ser negro.

Ejemplo de encontrar contornos de una imagen binaria:

```
import numpy as np
import cv2

im = cv2.imread('test.jpg')
imgray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
ret,thresh = cv2.threshold(imgray,127,255,0)
imagen,                contornos,                jerarquia                =
cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

Ver que hay tres argumentos en la función **cv2.findContours()**, el primero es la imagen fuente, el segundo es el modo de recuperación de contorno, y el tercero es el método de aproximación de contorno. La función posee tres variables de salida: imagen, contornos y jerarquía. Contornos es una lista de Python de todos los contornos de la imagen. Cada contorno individual es una matriz Numpy de coordenadas (x, y) de los puntos de los bordes del objeto.

Nota: más adelante se verá en detalle, los argumentos segundo y tercero, y sobre la jerarquía. Hasta el momento, los valores dados a ellos en el ejemplo del código funcionarán bien para todas las imágenes.

7.14.2.- ¿Cómo dibujar contornos?

Para dibujar los contornos tenemos la función **cv2.drawContours**. Esta función también es utilizable para dibujar cualquier forma siempre que se conozcamos sus contornos. El primer argumento de la función es la imagen fuente, el segundo argumento son los contornos, que deben ser pasados como una lista de Python; el tercer argumento es el índice de los contornos (útil para dibujar contornos individuales; para dibujar todos los contornos fijar este parámetro en -1), y los restantes argumentos son color, grosor, etc.

Ejemplo para dibujar todos los contornos en una imagen:

```
img = cv2.drawContours(img, contornos, -1, (0,255,0), 3)
```

Para dibujar, supongamos, el cuarto contorno:

En general, la mayoría de las veces el siguiente método resulta mucho más útil.

```
cnt = contornos[4]
img = cv2.drawContours(img, [cnt], 0, (0,255,0), 3)
```

Nota: aunque los dos métodos anteriores conducen al mismo resultado, más adelante veremos que el segundo resulta mucho más útil.

7.14.3.- Método de aproximación de contornos

Corresponde con el tercer argumento en la función **cv2.findContours**. Veamos su significado.

Más arriba se ha comentado que los contornos son los límites de una forma con la misma intensidad. La variable contornos almacena las coordenadas (x, y) de los bordes de una forma. Pero, ¿almacena todas las coordenadas? Esto se precisamente lo que se especifica mediante este método de aproximación de contorno.

Si se pasa **cv2.CHAIN_APPROX_NONE**, todos los puntos de los bordes se almacenan. Pero ¿realmente necesitamos todos los puntos?. Por ejemplo, supongamos que encontramos el contorno de una línea recta. ¿Necesitamos todos los puntos de la recta para representar esa línea? No, sólo necesitamos dos puntos a los extremos de esa línea. Esto es lo que hace **cv2.CHAIN_APPROX_SIMPLE**. Es decir, elimina todos los puntos redundantes y comprime el contorno, con el consiguiente ahorro de memoria.

A continuación se muestra una imagen de un rectángulo que ilustra esta técnica. Basta con dibujar un círculo en todas las coordenadas de la matriz de contorno (en color azul). La primera imagen muestra los puntos que se obtienen con **cv2.CHAIN_APPROX_NONE** (734 puntos) y la segunda imagen muestra los obtenidos con **cv2.CHAIN_APPROX_SIMPLE** (sólo 4 puntos). ¡Observar, cuánta memoria nos ahorramos!.

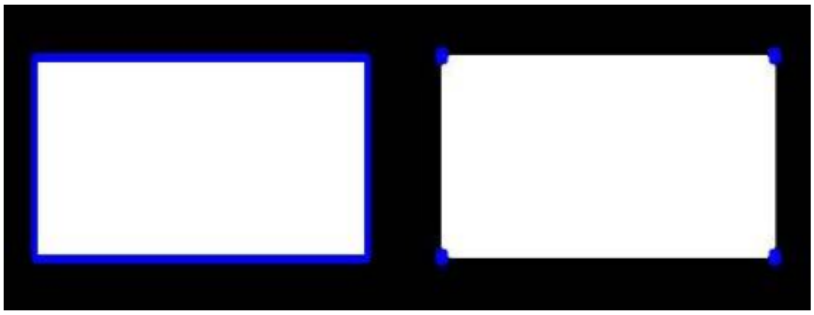


Fig.73: Ejemplo localización de contornos.

7.14.4.- Momentos

Los momentos de la imagen nos permiten calcular algunas de sus características, como el centro de masa del objeto, el área del objeto, etc.

La función **cv2.moments()** devuelve todos los momentos de la imagen. Sigue un ejemplo de como utilizar esta función:

```
import cv2

img = cv2.imread('test.jpg',0)
ret,thresh = cv2.threshold(img,127,255,0)
image,contours, hierarchy = cv2.findContours(thresh,1,2)

cnt = contours[0]
M = cv2.moments(cnt)
print(M)
```

M contiene la información de todos los momentos. Los distintos momentos son útiles para calcular diferentes parámetros de la imagen, como por ejemplo el centroide, que se determina a partir:

```
cx = int(M['m10']/M['m00'])
cy = int(M['m01']/M['m00'])
```

7.14.5.- Área de contorno

El área de contorno viene dada por la función **cv2.contourArea()** o por el momento M [‘m00’].

```
area = cv2.contourArea(cnt)
```

7.14.6.- Perímetro de contorno

También llamado longitud de arco, se puede encontrar utilizando la función **cv2.arcLength()**. El segundo argumento especifica si la forma es un contorno cerrado (si se pasa como *True*) o simplemente una curva.

```
perimetro = cv2.arcLength(cnt,True)
```

7.14.7.- Aproximación de contorno

Para aproximar una forma de contorno a otra forma con menos número de vértices, dependiendo de la precisión que se especifique. Corresponde a una implementación del algoritmo de Douglas-Peucker.

Para entender esto, supongamos que estamos tratando de encontrar un cuadrado en una imagen, pero debido a algunos problemas en la imagen, no obtuvo un cuadrado perfecto, sino una "mala forma" (como se muestra en la primera imagen siguiente). La función **cv2.approxPolyDP** permite aproximar esta forma a un cuadrado. El segundo argumento de esta función se denomina épsilon, que es la distancia máxima entre el contorno y el contorno aproximado. Es un parámetro de precisión y por lo tanto, es necesaria una correcta selección de épsilon para obtener la salida correcta.

```
epsilon = 0.1*cv2.arcLength(cnt,True)
approx = cv2.approxPolyDP(cnt,epsilon,True)
```

Abajo, en la segunda imagen, la línea verde muestra la curva aproximada para *epsilon = 10%*. La tercera imagen muestra lo mismo para *epsilon = 1%*. El tercer argumento de **cv2.approxPolyDP()** especifica si la curva es cerrada o no.

Fig.74: Ejemplo aproximación de contornos.

7.14.8.- Envoltura convexa

La envoltura convexa aporta un resultado similar a la aproximación de contorno. No obstante, aunque ambas pueden proporcionar los mismos resultados en algunos casos, en general, son dos técnicas distintas. La función **cv2.convexHull()** comprueba una curva de los defectos de convexidad y la corrige. En general, las curvas convexas son las curvas que siempre están abultadas, o al menos planas. Si la curva está abombada en el interior, se dice que tiene defectos de convexidad. Por ejemplo, veamos la imagen de la mano a continuación. La línea roja muestra la envoltura convexa de la mano. Las marcas de las flechas de doble cara muestran los defectos de convexidad, que son las desviaciones máximas locales de la envoltura de los contornos.

Fig.75: Ejemplo de envoltura convexa.

La sintaxis para obtener la envoltura de contornos es la siguiente:

```
envoltura = cv2.convexHull(points[, hull[, clockwise[, returnPoints]]
```

Donde los argumentos de la función tiene los siguientes significados:

- **points**: son las coordenadas de los contornos que pasamos.
- **hull**: es la salida, que normalmente se evita.
- **clockwise**: Indicador de orientación. Si es verdadero, la envoltura convexa de salida estará orientada en sentido horario. De lo contrario, estará orientada en sentido contrario a las agujas del reloj.
- **ReturnPoints**: Por defecto está fijado en ***True***. Devuelve las coordenadas de los puntos de la envoltura. Si es ***Falso***, devuelve los índices de los puntos de contorno correspondientes a los puntos de la envoltura.

De manera que para obtener una imagen como la anterior es suficiente con escribir:

```
envoltura = cv2.convexHull(cnt)
```

Nota: Si queremos encontrar los defectos de convexidad, necesitamos pasar **returnPoints** = ***False***.

7.14.8.1.- Revisando convexidad

Hay una función para comprobar si una curva es convexa o no, **cv2.isContourConvex()**. Sólo devuelve si es *Verdadero* o *Falso*.

```
k = cv2.isContourConvex(cnt)
```

7.14.9.- Rectángulo delimitador

Existen dos tipos de rectángulos delimitadores, rotado y sin rotar.

7.14.9.1- Rectángulo recto

Es un rectángulo recto, no considera la rotación del objeto. Así que el área del rectángulo delimitador no será mínima. Se encuentra por la función **cv2.boundingRect()**.

Sea (*x*, *y*) la coordenada superior izquierda del rectángulo y (*w*, *h*) su anchura y altura.

```
x,y,w,h = cv2.boundingRect(cnt)
img = cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
```

7.14.9.2.- Rectángulo rotado

En este caso el rectángulo delimitador se dibuja con el área mínima, por lo que también considera la rotación. La función utilizada es **cv2.minAreaRect()**. Devuelve una estructura Box2D que contiene los siguientes detalles: (esquina superior izquierda (x, y), (anchura, altura), ángulo de rotación). Pero para dibujar este rectángulo, necesitamos 4 esquinas del rectángulo, que se obtienen mediante la función **cv2.boxPoints()**.

```
rect = cv2.minAreaRect(cnt)
box = cv2.boxPoints(rect)
box = np.int0(box)
im = cv2.drawContours(im,[box],0,(0,0,255),2)
```

A continuación podemos ver ambos rectángulos (rotado y sin rotar) sobre una imagen segmentada de una célula migratoria. Este tipo de análisis es muy útil para investigar las características del movimiento de estas células.

Fig.76: Ejemplo de rectángulo rotado.

7.14.10- Círculo mínimo de inclusión

La función **cv2.minEnclosingCircle()** permite encontrar el círculo que cubre completamente el objeto con un área mínima.

```
(x,y),radius = cv2.minEnclosingCircle(cnt)
center = (int(x),int(y))
radius = int(radius)
img = cv2.circle(img,center,radius,(0,255,0),2)
```

Fig.77: Ejemplo circulo de mínima inclusión.

7.14.11.- Ajustando a una elipse

La función **cv2.fitEllipse()** ajusta una elipse a un objeto. Devuelve la elipse inscrita en el rectángulo rotado.

```
ellipse = cv2.fitEllipse(cnt)
im = cv2.ellipse(im,ellipse,(0,255,0),2)
```

Fig.78: Ejemplo de ajuste a elipse.

7.14.12.- Ajustando a una línea

Del mismo modo se puede ajustar una línea a un conjunto de puntos. A continuación un ejemplo de cómo trabaja la función **cv2.fitLine()**:

```
rows,cols = img.shape[:2]
[vx,vy,x,y] = cv2.fitLine(cnt, cv2.DIST_L2,0,0.01,0.01)
lefty = int((-x*vy/vx) + y)
righty = int(((cols-x)*vy/vx)+y)
img = cv2.line(img,(cols-1,righty),(0,lefty),(0,255,0),2)
```

Fig.79: Ejemplo de ajuste a línea.

7.14.13.- Propiedades de los contornos

7.14.13.1.- Relación de aspecto

Es la razón entre el ancho y la altura del contorno del objeto.

$$RazondeAspecto = \frac{Ancho}{Altura}$$

```
x,y,w,h = cv2.boundingRect(cnt)
aspect_ratio = float(w)/h
```

7.14.13.2.- Extensión

La extensión es la razón entre el área del contorno y el área del rectángulo delimitador.

$$Extension = \frac{Areadel Objeto}{Areadel RectanguloDelimitador}$$

```
area = cv2.contourArea(cnt)
x,y,w,h = cv2.boundingRect(cnt)
rect_area = w*h
extension = float(area)/rect_area
```

7.14.13.3.- Solidez

La solidez es la razón entre el área del contorno y el área de su envoltura convexa.

$$Solidez = \frac{Areadel Contorno}{AreadeEnvolturaConvexa}$$

```
area = cv2.contourArea(cnt)
hull = cv2.convexHull(cnt)
hull_area = cv2.contourArea(hull)
```

```
solidez = float(area)/hull_area
```

7.14.14.- Diámetro equivalente

Es el diámetro del círculo cuya área es igual que el área del contorno.

$$DiametroEquivalente = \sqrt{\frac{4 \times AreadelContorno}{\pi}}$$

```
area = cv2.contourArea(cnt)
equi_diametro = np.sqrt(4*area/np.pi)
```

7.14.15.- Orientación

La orientación es el ángulo que forma el eje mayor de la elipse circunscrita al objeto, con la dirección horizontal. El siguiente método también da las longitudes del Eje Mayor y del Eje Menor de dicha elipse.

```
(x,y),(MA,ma),angle = cv2.fitEllipse(cnt)
```

7.14.16.- Máscara y número de píxeles

En algunos casos es muy útil crear una máscara del objeto de interés. Adicionalmente, puede resultar útil conocer los puntos que conforman el objeto. Ambas operaciones se pueden hacer de la siguiente manera:

```
mask = np.zeros(imgray.shape,np.uint8)
cv2.drawContours(mask,[cnt],0,255,-1)
pixelpoints = np.transpose(np.nonzero(mask))
#pixelpoints = cv2.findNonZero(mask)
```

Ambos métodos mostrados conducen al mismo resultado. El primero utiliza funciones Numpy, mientras que el segundo utiliza la función Open CV (última línea comentada). Aunque los resultados de los dos métodos son iguales, existe una ligera diferencia entre ellos. Numpy da las coordenadas en formato (*fila, columna*), mientras que Open CV da las coordenadas en formato (*x, y*). Así que básicamente las respuestas serán intercambiadas. Hemos de tener en cuenta que, *row* = *x* y *column* = *y*.

7.14.17.-Valores mínimo y máximo y sus respectivas coordenadas

Estos valores pueden encontrarse utilizando una máscara de la imagen:

```
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(imgray,mask = mask)
```

7.14.18.- Color medio o Intensidad media

Aquí podemos encontrar el color medio de un objeto. O puede ser la intensidad media del objeto en modo de escala de grises. De nuevo usamos la misma máscara para hacerlo.

```
mean_val = cv2.mean(im,mask = mask)
```

7.14.19.- Puntos extremos

Los puntos extremos son cuatro y se corresponden con: el punto superior, el inferior, el derecho y el izquierdo de la imagen.

```
izquierdo = tuple(cnt[cnt[:,:,0].argmin()][0])
derecho = tuple(cnt[cnt[:,:,0].argmax()][0])
```

```
superior = tuple(cnt[cnt[:,:,1].argmin()][0])
inferior = tuple(cnt[cnt[:,:,1].argmax()][0])
```

Por ejemplo, veamos el código anterior a un mapa de sudamérica, obtenemos el resultado siguiente:

Fig.80: Ejemplo de punto extremo.

7.14.20.- Defectos de convexidad

En entradas anteriores hemos visto el concepto de envoltura convexa. Cualquier desviación del objeto de esta envoltura puede considerarse como defecto de convexidad.

Open CV viene con una función ya hecha para encontrar esto, **cv2.convexityDefects()**. A continuación un ejemplo de cómo llamar a esta función:

```
#Carga la imagen
img = cv2.imread('test.jpg',0)

#Determina los contornos
ret,thresh = cv2.threshold(img,120,255,0)
image,contours, hierarchy = cv2.findContours(thresh,1,2)
cnt = contours[0]

#Halla los defectos de convexidad
envoltura = cv2.convexHull(cnt,returnPoints = False)
defectos = cv2.convexityDefects(cnt,envoltura)
```

Nota: Recordar que debe pasar **returnPoints** = ***False***, cuando determinemos la envoltura convexa, a fin de encontrar defectos de convexidad.

La función **cv2.convexityDefects()** devuelve una matriz con los siguientes valores en sus filas: las coordenadas *inicial* y *final* de los puntos contiguos de mayor convexidad, el punto más alejado ("*el defecto*") de lo que debería ser el contorno convexo sin defectos y, la *distancia* hasta este punto. En el siguiente ejemplo se ilustra el significado de cada uno de estos puntos. Para ello dibujamos una línea (amarilla) que una los puntos de mayor convexidad, luego dibujamos un círculo (rojo) en el punto más lejano.

```
import cv2

#Carga la imagen
img = cv2.imread('estrella.png')

#Convierte la imagen a escala de grises
imgray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

```
ret,thresh = cv2.threshold(imgray,127,255,0)

#Determina los contornos
image,contours, hierarchy = cv2.findContours(thresh,1,2)
cnt = contours[0]

#Determina los defectos de convexidad
envoltura = cv2.convexHull(cnt,returnPoints = False)
defectos = cv2.convexityDefects(cnt,envoltura)

#Dibuja la envoltura convexa y los defectos de convexidad
for k in range(defectos.shape[0]):
    i,f,l,d = defectos[k,0]
    inicio= tuple(cnt[i][0])
    fin= tuple(cnt[f][0])
    lejos = tuple(cnt[l][0])
    cv2.line(img,inicio,fin,[0,255,255],2)
    cv2.circle(img,lejos,5,[0,0,255],-1)

#Muestra la imagen final
cv2.imshow('img',img)
cv2.waitKey(0)
cv2.destroyAllWindows()

#Guarda la imagen
cv2.imwrite('def_convexidad.png',img)
```

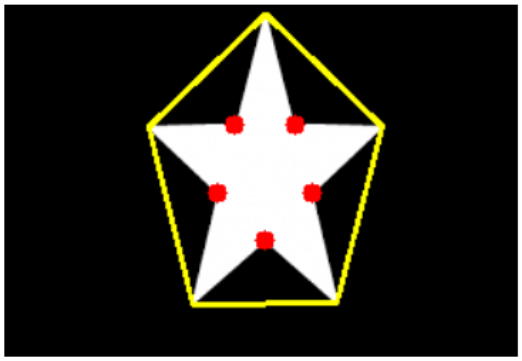


Fig.81: Ejemplo de defecto de convexidad.

7.14.21.- Prueba de polígono de puntos

Esta función encuentra la distancia más corta entre un punto de la imagen y su contorno. Esta distancia es negativa si el punto está fuera del contorno, positiva si está dentro y cero si está sobre el contorno.

Por ejemplo, podemos comprobar el punto (30,30) de la siguiente manera:

```
dist = cv2.pointPolygonTest(cnt,(30,30),True)
```

Aquí, el tercer argumento de la función es *measureDist*. Si es ***True***, encuentra la distancia mínima explicada anteriormente. Si es ***Falso***, entonces la función devuelve +1, -1 ó 0, indicando que el punto está dentro, fuera o sobre el contorno, respectivamente.

Nota: si no deseamos encontrar la distancia, debemos asegurarnos de que el tercer argumento es ***False***, porque este es un proceso que consume mucho tiempo. De hecho, fijando el tercer argumento como ***False***, la función corre de dos a tres veces más rápido.

7.14.22.- Haciendo coincidir formas

Otra interesante función de Open CV es **cv2.matchShapes()** que nos permite comparar dos formas, o dos contornos y devuelve una métrica que muestra la similitud. Cuanto menor sea el resultado, más similares serán las imágenes comparadas. Se calcula sobre la base de los valores de los momentos invariantes de Hu.

```
import cv2
#Carga ambas imágenes que se desean comparar
img1 = cv2.imread('estrella.png',0)
img2 = cv2.imread('estrella2.png',0)

#Determina los contornos de ambas imágenes
ret, thresh = cv2.threshold(img1, 127, 255,0)
images,contours,hierarchy = cv2.findContours(thresh,2,1)
cnt1 = contours[0]
ret, thresh2 = cv2.threshold(img2, 127, 255,0)
images,contours,hierarchy = cv2.findContours(thresh2,2,1)
cnt2 = contours[0]

#Calcula la similitud entre ambas imágenes y muestra el resultado
ret = cv2.matchShapes(cnt1,cnt2,1,0.0)
print(ret)
```

A continuación se muestran los resultados obtenidos al comparar las siguientes imágenes:

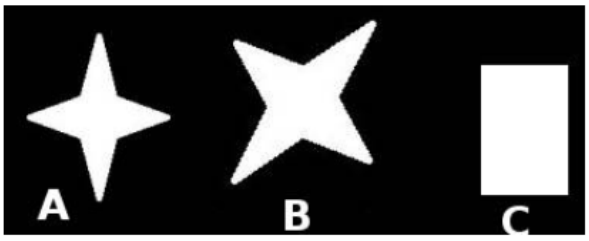


Fig.82: Ejemplo de similitud de formas.

Resultados de comparación: Imagen A con ella misma = 0.0 Imagen A con la imagen B = 0,001946 Imagen A con imagen C = 0.326911Note que incluso la rotación de la imagen no afecta mucho a esta comparación.

Nota: Los Momentos de Hu son siete momentos invariantes a la traslación, la rotación y la escala. Estos valores se pueden encontrar utilizando la función **cv2.HuMoments()**.

7.14.23- Jerarquía de contornos

Como se ha indicado anteriormente, el objetivo principal de la función **cv2.findContours()** es detectar objetos en una imagen. Aunque muchas veces los objetos están en lugares diferentes, en algunos casos, algunas formas están anidadas dentro de otras formas. En este último escenario, se llama ***padres (parents)*** a los contornos más externos e hijos ***(children)*** a los contornos contenidos en los anteriores. De esta forma, los contornos de una imagen tienen cierta relación entre sí. Utilizando esta definición se puede especificar cómo los contornos están conectados entre sí, por ejemplo, especificando si un contorno es hijo de otro, o por el contrario su padre. Es precisamente la representación de esta relación a lo que se llama ***Jerarquía***.

A continuación se muestra un ejemplo donde esta idea queda más clara:

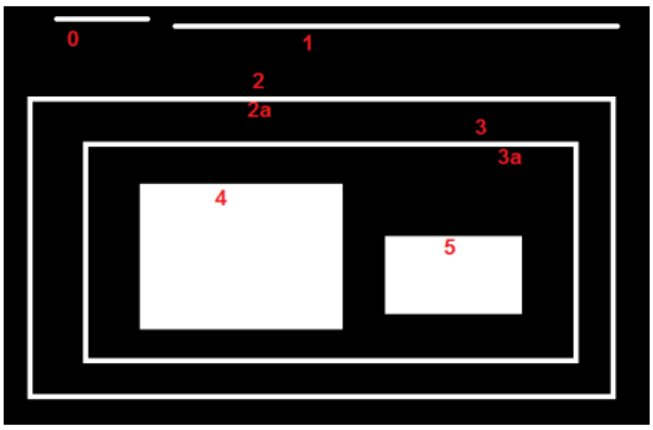


Fig.83: Ejemplo de jerarquías de contornos.

En esta imagen hay algunas formas, numeradas de 0 a 5. ***2*** y ***2a*** denotan los contornos externo e interno de la caja más externa, respectivamente.

Aquí, los contornos ***0,1,2*** son externos o ultraperiféricos. Podemos decir que están en ***jerarquía-0*** o simplemente que están en el mismo nivel de jerarquía.

Luego viene el contorno-***2a***, que puede considerarse como un hijo del contorno-***2*** (o en sentido opuesto, el contorno-2 es el padre del contorno-***2a***). Estos definen la ***jerarquía-1***.

Del mismo modo el contorno-***3*** es hijo de contorno-***2*** y pertenece a la siguiente ***jerarquía, 2***.

Finalmente, los contornos ***4*** y ***5*** son los hijos del contorno-***3a***, y forman el último nivel de ***jerarquía, 3***.

Debido a la manera en que están enumeradas las cajas, se podría estar tentado a pensar que el contorno-***4*** es el primer hijo del contorno-***3a***. Sin embargo, el contorno-***5*** también podría ser el primer hijo.

Con este ejemplo quedan claros conceptos tales como: *mismo nivel de jerarquía*, *contorno externo*, *contorno hijo*, *contorno padre*, *primer hijo*, etc. Ahora estamos listos para ver estos conceptos en Open CV.

7.14.23.1.- Representación de Jerarquías en Open CV

Como se ha visto, cada contorno tiene su propia información sobre a qué jerarquía pertenece, quién es su hijo, quién es su padre, etc. Open CV representa esta información como una matriz de cuatro valores: [***Next, Previous, First_Child, Parent***]. Esta matriz es la tercera salida de la función **cv2.findContours()**.

Next (siguiente)

Next indica el siguiente contorno en el mismo nivel jerárquico. Por ejemplo, consideremos el contorno-***0*** de la imagen anterior. ¿Cuál es el siguiente contorno en su mismo nivel? Es el

contorno-***1***. Por lo tanto *Next = 1*. De manera similar, para contorno-***1***, el siguiente es contorno-***2***. Por consiguiente *Next = 2*.

En el caso del contorno-***2***, como no hay siguiente contorno en el mismo nivel, *Next = -1*. Por otra parte, el contorno-***4***, está en el mismo nivel del contorno-***5***. Así que su siguiente contorno es contorno-***5***, luego *Next = 5*.

Previous (anterior)

Previous indica el contorno anterior en el mismo nivel jerárquico. Por ejemplo, el contorno anterior del contorno-***1*** es el contorno-***0*** en el mismo nivel. Del mismo modo para el contorno-***2***, es el contorno-***1***. Para el contorno-***0***, no hay anterior, así que *Previous=-1*.

First_Child (primer hijo)

First_Child denota el primer contorno hijo. Por ejemplo, para contorno-***2***, el hijo es contorno-***2a***. Así que First Child tendrá el valor de índice correspondiente del contorno-***2a***. Por otro lado, el contorno-***3a*** tiene dos hijos. Sin embargo, sólo se toma el primer hijo, que es el contorno-***4***. Así que *First_Child = 4* para el contorno-***3a***. Por último, si no hay hijos, como en el caso del contorno-***4*** o ***5***, *First_Child=-1***.**

Parent (padre)

Parent indica el índice del controno padre. Es justo lo opuesto de First_Child. Tanto para el contorno-***4*** como para el contorno-***5***, el contorno padre es el contorno-***3a***. Para contorno-***3a***, es contorno-***3*** y así sucesivamente. Si no hay contorno padre, como en el caso del contorno-***0***, *Parent=-1*.

Modos de recuperación de contorno:

Una vez sabemos sobre el estilo de jerarquía utilizado en Open CV, podemos entender los modos de recuperación de contorno en Open CV, con la ayuda de la misma imagen anteriormente utilizada. Veamos el significado de las banderas ***cv2.RETR_LIST, cv2.RETR_TREE, cv2.RETR_CCOMP*** y **cv2.RETR_EXTERNAL.**

RETR_LIST

Esta es la más simple de las cuatro banderas (desde el punto de vista de la explicación). Simplemente, recupera todos los contornos, pero no crea ninguna relación padre-hijo. Los padres y los hijos son iguales bajo esta regla, y son sólo contornos. Es decir, todos ellos pertenecen al mismo nivel jerárquico.

Así que el 3er y 4to término en la matriz de jerarquía, correspondientes a ***First_Child*** y ***Parent,*** respectivamente, serán siempre -1. Obviamente, los términos ***Next*** y ***Previous*** tendrán sus valores correspondientes.

A continuación se muestra el resultado obtenido al utilizar la bandera RETR_LIST. En cada fila están los detalles de la jerarquía del contorno correspondiente. Por ejemplo, la primera fila corresponde al contorno-***0***. El siguiente contorno es el contorno-***1***. Así que *Next = 1*. No hay contorno anterior, así que *Previous = -1*. Los dos términos restantes, como se explicó anteriormente, serán -1.

```
print(hierarchy)
```

```
([[[ 1, -1, -1, -1],
   [ 2,  0, -1, -1],
   [ 3,  1, -1, -1],
   [ 4,  2, -1, -1],
   [ 5,  3, -1, -1],
   [ 6,  4, -1, -1],
   [ 7,  5, -1, -1],
   [-1,  6, -1, -1]]])
```

Esta es una buena opción, si no se está usando ninguna característica de jerarquía.

RETR_EXTERNAL

Si utiliza este indicador, sólo se devuelven indicadores externos extremos. Los contornos hijos no se tendrán en cuenta. Podemos decir, bajo esta ley, que sólo el mayor en cada familia es conservado.

Por lo tanto, en nuestra imagen, sólo tendremos 3 contornos, corrspondientes al nivel de jerarquía 0. Es decir, sólo se tendrán en cuenta los contornos ***0,1,2***. A continuación se muestra el resultado de utilizar esta bandera a la imagen de prueba:

```
print(hierarchy)
```

```
([[[ 1, -1, -1, -1],
   [ 2,  0, -1, -1],
   [-1,  1, -1, -1]]])
```

Este indicador es útil si se desea extraer sólo los contornos exteriores.

RETR_CCOMP

Este indicador recupera todos los contornos y los organiza en una jerarquía de dos niveles. Es decir, los contornos externos del objeto (es decir, su frontera) se colocan en la jerarquía-1. Y los contornos de los agujeros dentro del objeto (si los hay) se colocan en la jerarquía-2.

Basta con considerar la imagen de un “gran cero blanco” sobre un fondo negro. El círculo exterior al cero pertenece a la primera jerarquía, y el círculo interno del cero pertenece a la segunda jerarquía.

Para que quede la idea más clara, se usará la siguiente imagen, donde se ha etiquetado el orden de los contornos en color rojo y la jerarquía a la que pertenece, en color verde (1 o 2). El orden es el mismo que el orden en el que Open CV detecta los contornos.

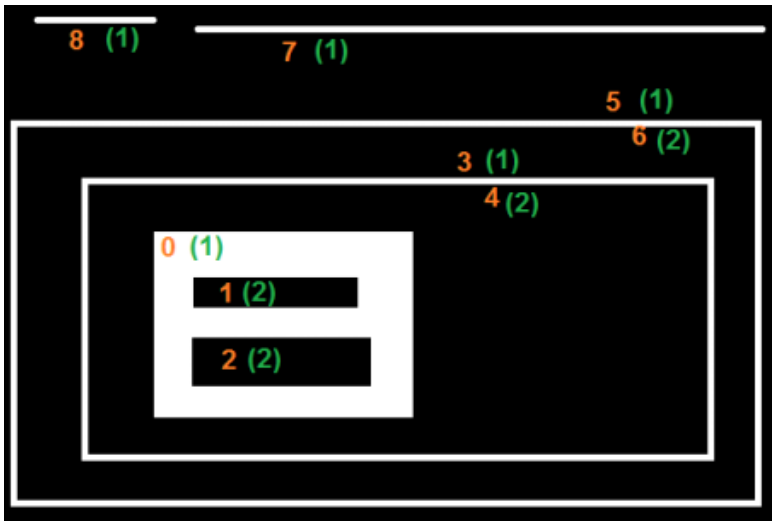


Fig.84: Ejemplo de organización de contornos.

Por ejemplo, en la imagen se puede ver que el contorno-***0*** *es de* ***jerarquía-1***, ya que tiene dos agujeros, los contornos ***1*** y ***2,*** que por lo tanto son de ***jerarquía-2***. Bajo esta estructura, el contorno siguiente al contorno-***0***, en el mismo nivel de jerarquía, es el contorno-***3***, por tanto *Next=3*. Por otro lado, *Previous=-1*, ya que no existe un contorno anterior. Finalmente, su primer hijo es contorno-***1*** en ***jerarquía-2***. No tiene padre, porque está en ***jerarquía-1***. Así que su matriz jerárquica sería [3, -1,1, -1].

El contorno-***1***, por otra parte, está en ***jerarquía-2***. El siguiente en la misma jerarquía (bajo la paternidad del contorno-1) es contorno-***2***. No tiene contorno anterior ni hijos, pero el padre es contorno-***0***. Así que la matriz de jerarquía será [2, -1, -1,0].

Del mismo modo para el contorno-***2*** se tendrá: ***jerarquía-2***, no contorno siguiente, contorno anterior el contorno-***1***, *n*ingún hijo y contorno padre el contorno-***0***. Por tanto la matriz será [-1,1, -1,0].

Contorno-***3***: Siguiente en ***jerarquía-1*** es contorno-***5***. Anterior es el contorno-***0***. El hijo es contorno-***4*** y ningún padre. Así que la matriz será [5,0,4, -1].

Contorno-**4**: Está en ***jerarquía-2*** bajo contorno-***3*** y no tiene hermano. Así que no hay siguiente, no hay anterior, no hay hijo y el padre es el contorno-***3***. Así que la matriz de jerarquía para este contorno será [-1, -1, -1,3].

Aplicando este indicador a la imagen de arriba, el resultado que se obtiene es el siguiente:

```
print(hierarchy)

([[[ 3, -1,  1, -1],
   [ 2, -1, -1,  0],
   [-1,  1, -1,  0],
   [ 5,  0,  4, -1],
   [-1, -1, -1,  3],
   [ 7,  3,  6, -1],
   [-1, -1, -1,  5],
   [ 8,  5, -1, -1],
   [-1,  7, -1, -1]]])
```

RETR_TREE

Este indicador recupera todos los contornos y crea una lista completa de jerarquías familiares. Incluso cuenta, quién es el abuelo, padre, hijo, nieto, etc.

Por ejemplo, utilicemos la misma imagen anterior pero ahora indicando la jerarquía que devuelve la bandera ***cv2.RETR_TREE***. De nuevo, las letras rojas dan el número de contorno y las letras verdes dan el orden de la jerarquía.

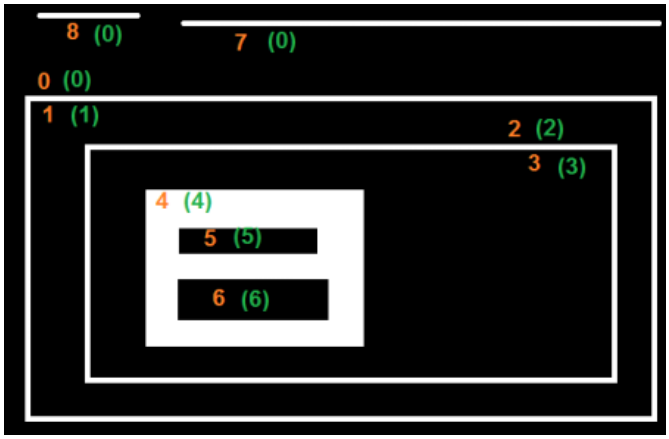


Fig.85: Ejemplo de organización de contornos (2).

Veamos algunos ejemplos:

contorno-***0***: Está en ***jerarquía-0***. El siguiente contorno en la misma jerarquía es el contorno-***7***. No hay contornos anteriores. El hijo es el contorno-***1*** y no hay padre. Entonces la matriz de jerarquía es [7, -1,1, -1].

contorno-***2***: Está en ***jerarquía-1***. No hay contornos en el mismo nivel. No anterior. El hijo es el contorno-***2*** y el padre es el contorno-***0***. Así que la matriz para este contorno es [-1, -1,2,0].

Intenta obtener los restantes por tu cuenta y compara tus resultados con la respuesta correcta, que se muestra a continuación:

```
print(hierarchy)
```

```
([[[ 7, -1, 1, -1],
  [-1, -1, 2, 0],
  [-1, -1, 3, 1],
  [-1, -1, 4, 2],
  [-1, -1, 5, 3],
  [ 6, -1, -1, 4],
  [-1, 5, -1, 4],
  [ 8, 0, -1, -1],
  [-1, 7, -1, -1]]])
```

7.15.- Histogramas de Open CV

7.15.1.- Generar, graficar y analizar histogramas

De forma general, el histograma es un gráfico que muestra la distribución de frecuencias de una variable dada. En el caso de las imágenes, el histograma da una idea general sobre la distribución de intensidades. Cuando hablamos de imágenes, un histograma es un gráfico con valores de píxeles (que, en general, oscilan entre 0 y 255 aunque no siempre) en el eje *X* y la cantidad correspondiente de píxeles en la imagen en el eje Y.

Es simplemente otra forma de entender la imagen. Al mirar el histograma de una imagen, se puede tener idea sobre el contraste, el brillo, la distribución de intensidad, etc. de esa imagen. Casi todas las herramientas de procesamiento de imágenes de hoy en día, proporcionan características en el histograma. A continuación, una imagen del sitio web de Cambridge in Color.

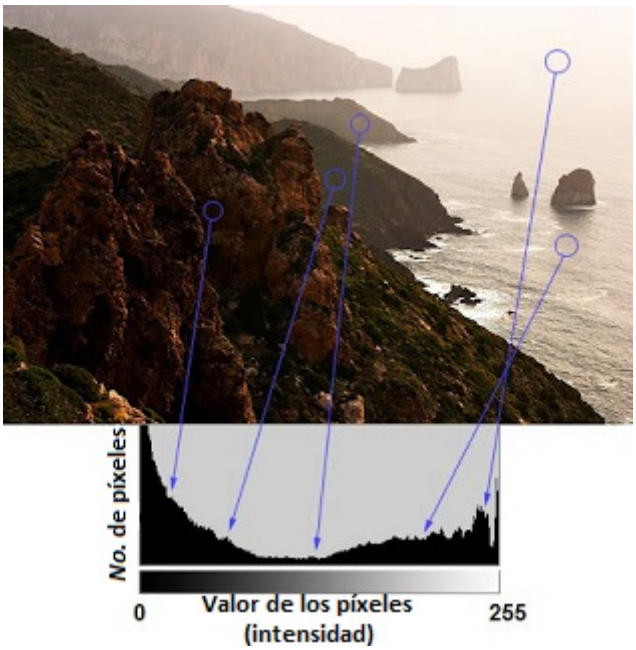


Fig.86: Ejemplo de un histograma.

En la figura se muestra la imagen y su histograma. (Recordar, este histograma se dibuja para la imagen en escala de grises, no para la imagen en color). La región izquierda del histograma muestra la cantidad de píxeles más oscuros en la imagen y la región derecha muestra la cantidad de píxeles más brillantes. Del histograma puede verse que las regiones oscuras en la imagen son mayores que las regiones brillantes, pues hay más píxeles con valores cerca de **0** que píxeles cerca de su valor máximo (**255**). Por otra parte, la cantidad de medios tonos (valores de píxel en el rango medio, digamos alrededor de **127**) son muy inferiores.

7.15.2.- Generar un histograma

Ahora que ya tenemos una idea de qué es un histograma, podemos ver cómo generarlo. Tanto ***Open CV*** como ***Numpy*** vienen con una función incorporada para esto. Pero antes de usar esas funciones, necesitamos comprender algunas terminologías relacionadas con los histogramas.

BINS: el histograma anterior muestra la cantidad de píxeles para cada valor de píxel, o sea, de 0 a 255. O sea que en el eje *X* hay 256 valores representados. Pero ¿qué pasa si no necesita encontrar la cantidad de píxeles para todos los valores de píxeles por separado, sino el número de píxeles en un intervalo de valores de píxeles? por ejemplo, necesita encontrar la cantidad de píxeles entre 0 y 15, luego 16 a 31, …, 240 a 255. Necesitará solo 16 valores para representar el histograma.

Entonces, lo que hace es simplemente dividir el histograma completo en 16 subintervalos y el valor de cada subintervalo es la suma de todos los recuentos de píxeles en él. Cada uno de estos subintervalos se denomina "*BIN*" (o columna en español). En el primer caso, el número de *BINS* era 256 (una por cada píxel), mientras que en el segundo caso, es solo 16. *BINS* está representado por el término *histSize* en Open CV.

DIMS: es la cantidad de parámetros para los que recopilamos los datos. En este caso, recopilamos datos con respecto solo a una cosa, valor de intensidad. Por lo tanto aquí será 1.

RANGO: es el rango de valores de intensidad que se desea medir. Normalmente, es [0,256], es decir, todos los valores de intensidad.

7.15.2.1.- Cálculo del histograma en OpenCV

La librería de Open CV viene provista de una función para calcular histogramas, esta es: ***cv2.calcHist()***. A continuación analizaremos esta función y sus parámetros:

cv2.calcHist (*imágenes, canales, máscara, histSize, rangos [, hist [, acumular]]****)***

Imágenes: es la imagen fuente del tipo *uint8* o *float32*. debe darse entre corchetes, es decir, "[img]".

c*anales*: también se debe poner entre corchetes. Es el índice de canal para el que calculamos el histograma. Por ejemplo, si la entrada es una imagen en escala de grises, su valor es [0]. Para la imagen en color, puede pasar [0], [1] o [2] para calcular el histograma del canal azul, verde o rojo, respectivamente.

máscara: imagen de máscara. Para encontrar el histograma de la imagen completa, se indica *"None"*. Pero si desea encontrar un histograma de una región particular de la imagen, debe crear una imagen de máscara para eso y darle una máscara. (Más adelante se mostrará un ejemplo).

histSize: esto representa nuestro conteo de *BINS*. Necesita ponerse también entre corchetes. Para la escala completa, pasamos [256].

gamas: Este es el rango de valores que puede tomar cada pixel, normalmente es [0,256].

Comencemos, pues, con una imagen de muestra. Simplemente, cargaremos la imagen y encontraremos su histograma.

```
img = cv2.imread('ejemplo.jpg',0)
hist = cv2.calcHist([img],[0],None,[256],[0,256])
```

hist es una matriz de 256×1, donde cada valor corresponde al número de píxeles en la imagen con valor 0, 1, 2…ó 255.

7.15.3.- Cálculo del histograma con Numpy

Numpy también posee una función para calcular el histograma: ***np.histogram()***. Prueba utilizar esta función en lugar de ***calcHist()***:

```
hist,bins = np.histogram(img.ravel(),256,[0,256])
```

En este caso, ***hist*** será exactamente igual lo devuelto por la función ***calcHist()***. Sin emabrgo, los contenedores tendrán 257 elementos, porque Numpy calcula los contenedores como 0-0.99, 1-1.99, 2-2.99, etc. Por lo tanto, el rango final sería 255-255.99. Para representar eso, también se agrega 256 al final de los contenedores. Pero no necesitamos ese 256. Hasta 255 es suficiente.

Nota: Numpy tiene otra función, *np.bincount()* que es mucho más rápida (alrededor de 10X) que *np.histogram()*. Por lo tanto para histogramas unidimensionales, puedes utilizar mejor esta

función. No olvides establecer *minlength* = 256 en *np.bincount*. Por ejemplo, **hist = np.bincount (img.ravel (), minlength = 256).**

La función de *Open CV* es más rápida (alrededor de 40X) que *np.histogram()*. Así que mejor utilizar la función de *Open CV*.

7.15.4.- Cómo graficar histogramas

Hay dos maneras de hacer esto:

1. Forma corta: utilizando las funciones de trazado ***Matplotlib***
2. Forma larga: utilizando las funciones de dibujo de ***OpenCV***

7.15.4.1.- Utilizando Matplotlib

Matplotlib viene con una función de trazado de histogramas: *matplotlib.pyplot.hist()*. Esta función encuentra directamente el histograma y lo traza. No es necesario uilizar primero la función ***calcHist()*** o ***np.histogram()*** para buscar el histograma. Vea el código a continuación:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('parlamentoBP.jpg',0)
plt.hist(img.ravel(),256,[0,256]); plt.show()
```

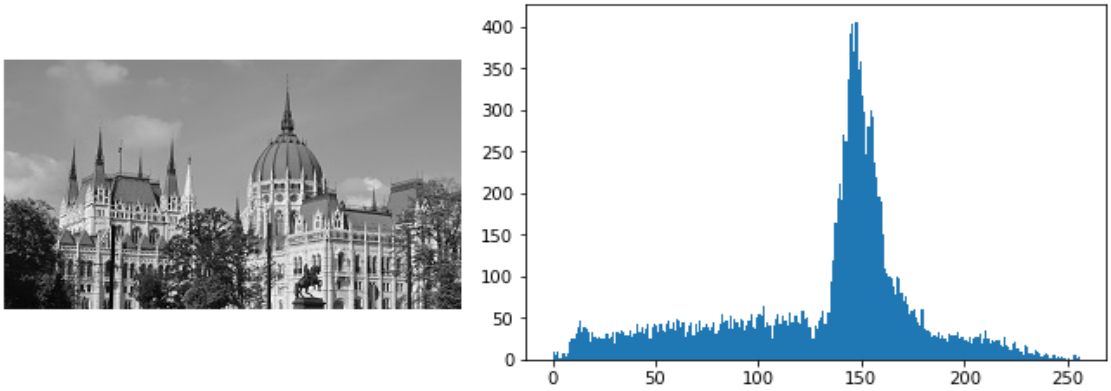


Fig.87: Ejemplo de histograma con Matplotlib.

O también puede usar la representación normal de ***Matplotlib***, lo cual sería bueno para el gráfico de RGB. Para esto, primero es necesario encontrar los datos del histograma. El código a continuación muestra un ejemplo de cómo hacer esto:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('parlamentoBP.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
    histr = cv2.calcHist([img],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

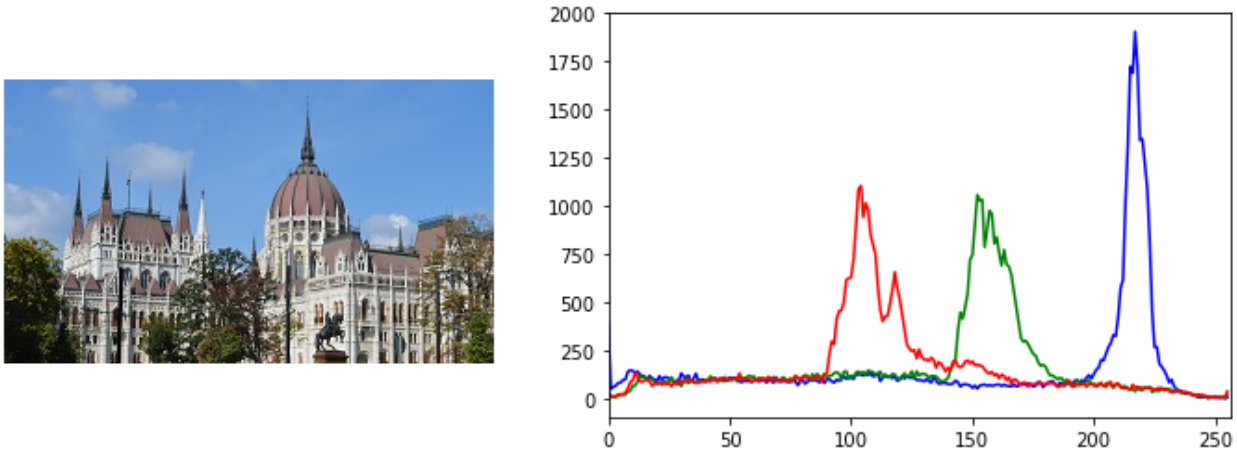


Fig.88: Ejemplo de histograma con Matplotlib y por colores RGB.

Del histograma anterior se puede deducir los colores rojos, verdes y azules están en áreas bien definidas de la imagen. Estos corresponden, obviamente, a los techos del Parlamento de Budapest (rojo), los árboles(verde) y el cielo (azul).

7.15.4.2.- Utilizando Open CV

Para dibujar un histograma como el mostrado anterior, utilizando la librería de Open CV, hay que utilizar una de las siguientes funciones: ***cv2.line()*** o ***cv2.polyline()***. Para esto es necesario pasar las coordenadas *X* e *Y*, correspondientes a los valores de *BINS* y los valores del histograma, respectivamente.

7.15.5.- Cómo aplicar una máscara

Hasta ahora hemos utilizado ***cv2.calcHist()*** para encontrar el histograma de la imagen completa. ¿Qué sucede si quisiéramos encontrar histogramas de algunas regiones de una imagen? Para esto, simplemente, debemos crear una imagen de máscara con un color blanco en la región en la que desea buscar el histograma y, el resto de la imagen en negro.

```
img = cv2.imread('parlamentoBP.jpg',0)

# crear máscara
mask = np.zeros(img.shape[:2], np.uint8)
mask[10:140, 100:200] = 255
masked_img = cv2.bitwise_and(img,img,mask = mask)

# Calcular el histohrama con máscara y sin máscara
# Fijar el tercer argumento como "mask"
hist_full = cv2.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv2.calcHist([img],[0],mask,[256],[0,256])

plt.subplot(221), plt.imshow(img, 'gray')
plt.subplot(222), plt.imshow(mask,'gray')
plt.subplot(223), plt.imshow(masked_img, 'gray')
plt.subplot(224), plt.plot(hist_full), plt.plot(hist_mask)
plt.xlim([0,256])

plt.show()
```

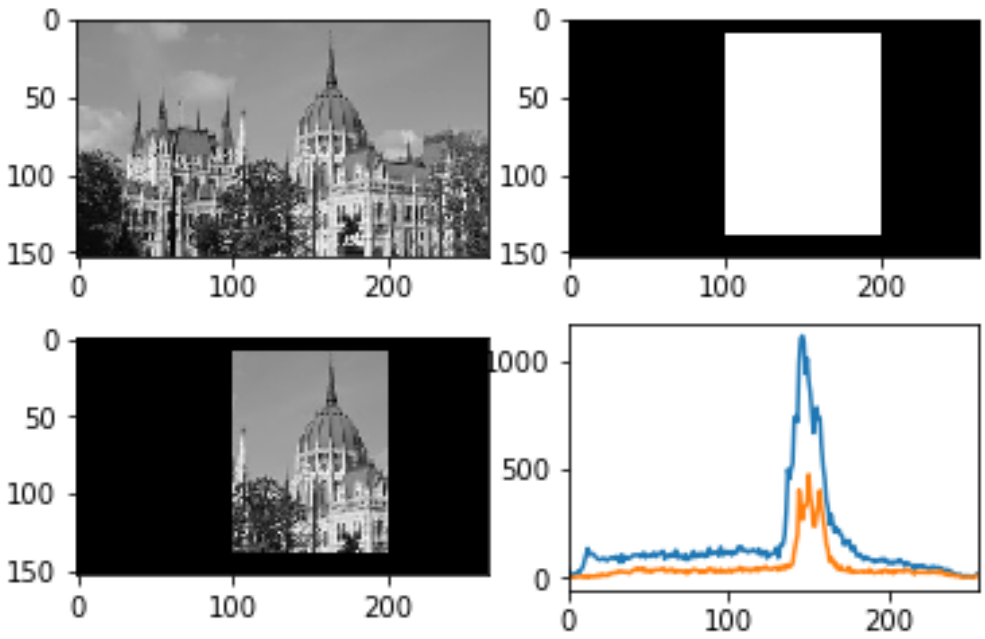


Fig.89: Ejemplo de histograma con máscara.

7.15.6.- Ecualización de histogramas

Consideremos una imagen cuyos valores de píxeles están limitados solo a un rango específico de valores. Por ejemplo, una imagen muy brillante tendrá todos los píxeles confinados en valores altos. Sin embargo, una buena imagen (con mayor contraste) tendrá diferentes valores de píxeles en todas las regiones de la imagen. Por lo tanto, una técnica usualmente empleada para aumentar el contraste de una imagen es estirar su histograma hacia cualquiera de los extremos (ver imagen siguiente). Esto es precisamente lo que realiza la ecualización de histogramas.

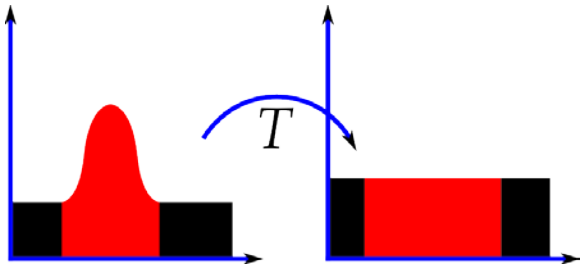


Fig.90: Ecualización de histograma.

En la práctica, tras aplicar la ecualización no se obtiene un histograma completamente plano como el mostrado en la anterior figura. Esto se debe a que los valores que pueden tomar los píxeles son discretos.

Aquí al momento vemos un ejemplo de su implementación utilizando las funciones de ***Numpy***. Más adelante veremos la función correspondiente en Open CV.

```
import numpy as np
from matplotlib import pyplot as plt

img1 = cv2.imread('wikimage.jpg',0)
```

```
#Genera el historama de la imagen
hist,bins = np.histogram(img1.flatten(),256,[0,256])

#Genera la función de distribución acumulada (cdf por sus siglas en inglés)
cdf = hist.cumsum()
cdf_normalized = cdf * hist.max()/ cdf.max()

#Genera los gráficos del histograma y de la función de distribución acumulada
plt.plot(cdf_normalized, color = 'b')
plt.hist(img1.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histograma'), loc = 'upper right')
plt.show()
```

Nota: el comando *img.flatten()* transforma el array bidimensional de la imagen en un vector unidimensional.

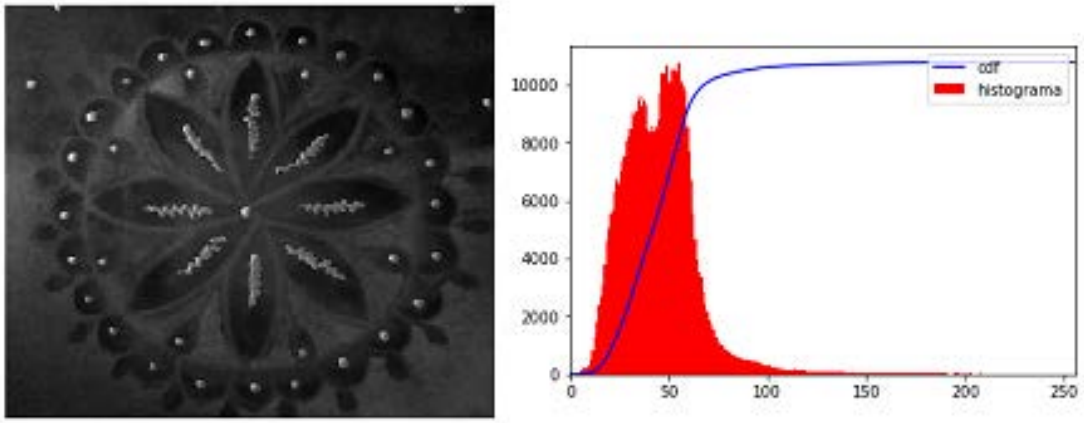


Fig.91: Ejemplo de histograma sin ecualizar.

En el histograma se observa que la mayoría de los píxeles tienen valores cercanos a cero (consistente con una imagen oscura). Para aumentar el contraste de esta imagen necesitamos ecualizar el histograma, o sea expandirlo en todo el rango de valores de 0 a 255.

Para esto encontramos los valores máximo y mínimo de la función de distribución (excluyendo los ceros) y aplicamos la ecuación de ecualización del histograma tal y como aparece en wikipedia. Para excluir los ceros utilizaremos la función ***np.ma.masked_equal()***, que enmascara (ignora) los valores iguales al número que se pase en el segundo argumento.

```
#Enmascara los valores iguales a cero
cdf_m = np.ma.masked_equal(cdf,0)

#Aplica la transformación de ecualización
cdf_m = (cdf_m - cdf_m.min())*255/(cdf_m.max()-cdf_m.min())

#Rellena los valores previamente enmascarados con ceros
cdf = np.ma.filled(cdf_m,0).astype('uint8')

#Aplica la ecualización a los píxeles de la imagen original
img2 = cdf[img1]

#Grafica la imagen resultante de aplicar la ecualización del histograma
cv2.imshow('image',img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Para graficar el histograma y la función de distribución repita el mismo código de más arriba cambiando img1 por img2. El resultado de la ecualización del histograma se muestra a continuación:

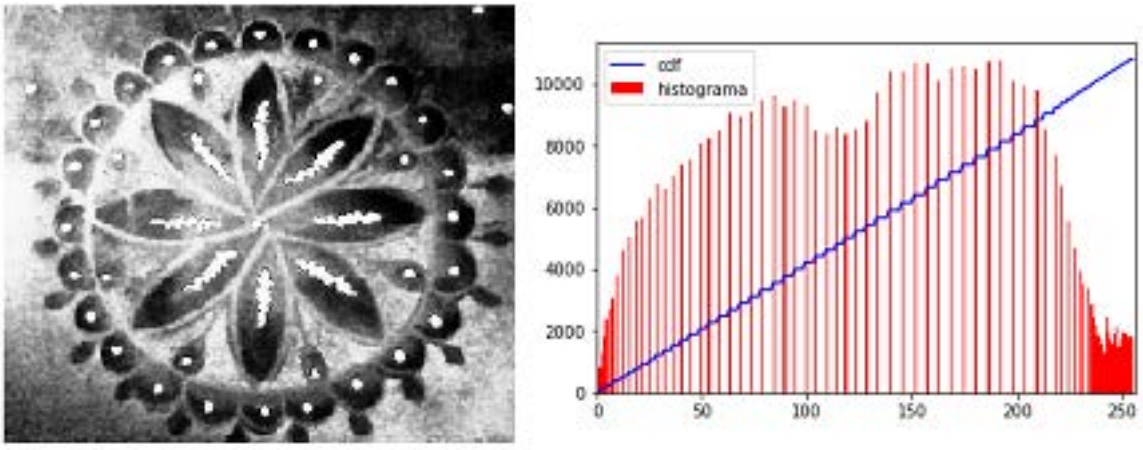


Fig.92: Ejemplo de realización de ecualización de histograma.

Otra característica importante es que, incluso si utilizaremos la misma imagen anterior pero más brillante (en lugar de tan oscura como la que hemos utilizado), después de la ecualización tendremos casi la misma imagen que obtuvimos tras ecualizar la oscura. Como resultado, esto se usa como una “herramienta de referencia” para igualar las condiciones de iluminación de imágenes diferentes. Esto es útil en muchos casos. Por ejemplo, en el reconocimiento facial, antes de entrenar los datos de la cara, las imágenes de las caras se ecualizan para hacer que todas tengan las mismas condiciones de iluminación.

7.15.6.1.- Ecualización de histogramas en Open CV

Open CV tiene una función para hacer esto, ***cv2.equalizeHist()***. Su entrada es solo una imagen en escala de grises y la salida es nuestra imagen luego de ecualizado su histograma.

A continuación se muestra un fragmento de código que muestra el uso de esta función para la misma imagen utilizada anteriormente.

```
img = cv2.imread('wikimage.jpg',0)
equ = cv2.equalizeHist(img)
res = np.hstack((img,equ)) #Agrupa las imágenes una al lado de la otra
cv2.imwrite('res.png',res)
```

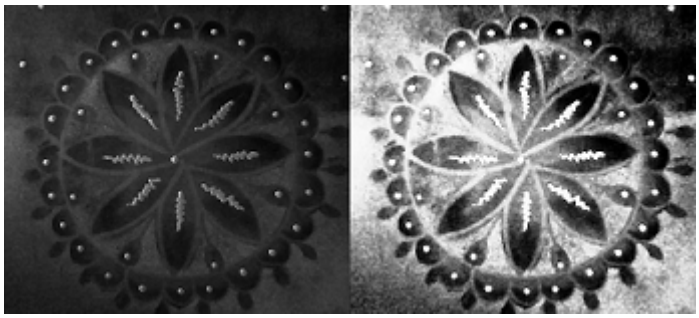


Fig.93: Ejemplo de realización de ecualización de histograma con Open CV.

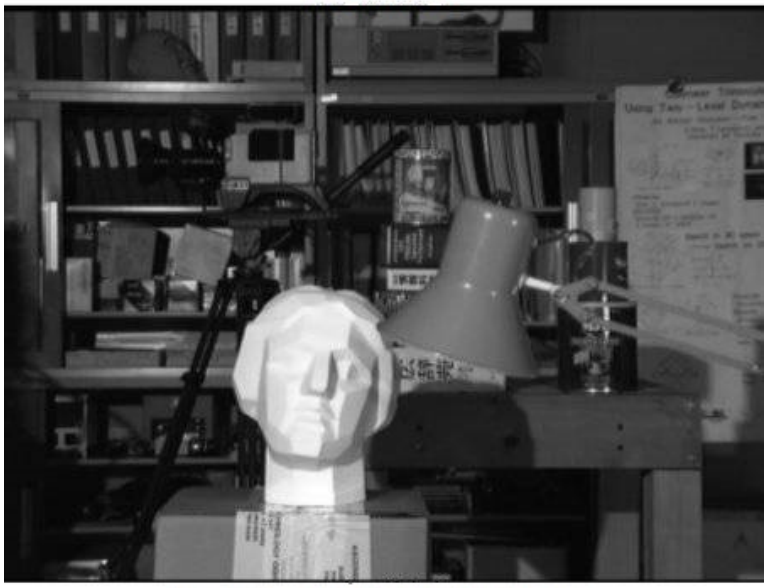
Ahora podemos utilizar diferentes imágenes con diferentes condiciones de luz, ecualizarlas y verificar los resultados.

La ecualización del histograma es buena cuando el histograma de la imagen está confinado a una región en particular. Ojo: No funcionará bien en lugares donde hay grandes variaciones de intensidad donde el histograma cubre una región grande, es decir, que están presentes píxeles brillantes y oscuros.

7.15.7.- Contraste de ecualización adaptable del histograma (CLAHE por sus siglas en inglés)

La ecualización de histograma que hemos visto anteriormente considera el contraste global de la imagen. Sin embargo, en muchos casos, esto no es una buena idea. Por ejemplo, la imagen inferior muestra una imagen de entrada y su resultado después de la ecualización del histograma global.

Imagen Original



Después de ecualización global de histograma

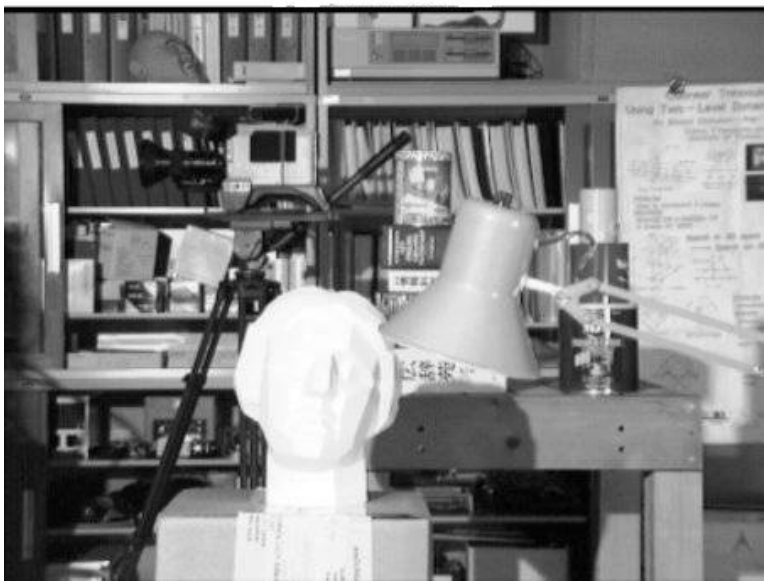


Fig.94: Imagen con referencia a utilizar ecualización de histograma adaptativo.

Es cierto que el contraste de fondo ha mejorado después de la ecualización del histograma. Sin embargo, si comparamos la cara de la estatua en ambas imágenes, notamos que hemos perdido la mayor parte de la información debido al exceso de brillo. Esto es debido a que su histograma no está confinado a una región en particular como vimos en casos anteriores (Tratar de trazar el histograma de la imagen de entrada, obtendrá más intuición).

Para resolver este problema, se utiliza la ecualización de histograma adaptativo. En este caso, la imagen se divide en pequeños bloques llamados “tiles” (*tileSize* es 8×8 por defecto en Open CV). A continuación, cada uno de estos bloques se ecualiza como siempre. Entonces, en un área pequeña, el histograma se limitaría a una región pequeña (a menos que haya ruido). Si hay ruido, se amplificará. Para evitar esto, se aplica la limitación de contraste. Si cualquier *BIN* del histograma está por encima del límite de contraste especificado (por defecto 40 en Open CV), esos píxeles se recortan y se distribuyen uniformemente a otros *BINS* antes de aplicar la ecualización de histograma. Después de la ecualización, para eliminar artefactos en los bordes de los mosaicos, se aplica la interpolación bilineal.

El siguiemte fragmento de código muestra cómo aplicar *CLAHE* en Open CV:

```
import numpy as np
import cv2

img = cv2.imread('ejemplo.png',0)

# Crea un objeto CLAHE (los argumentos son opcionales).
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
cl1 = clahe.apply(img)

cv2.imwrite('clahe.jpg',cl1)
```

Vemos el resultado a continuación y comparado con los resultados anteriores, especialmente la región de la estatua:

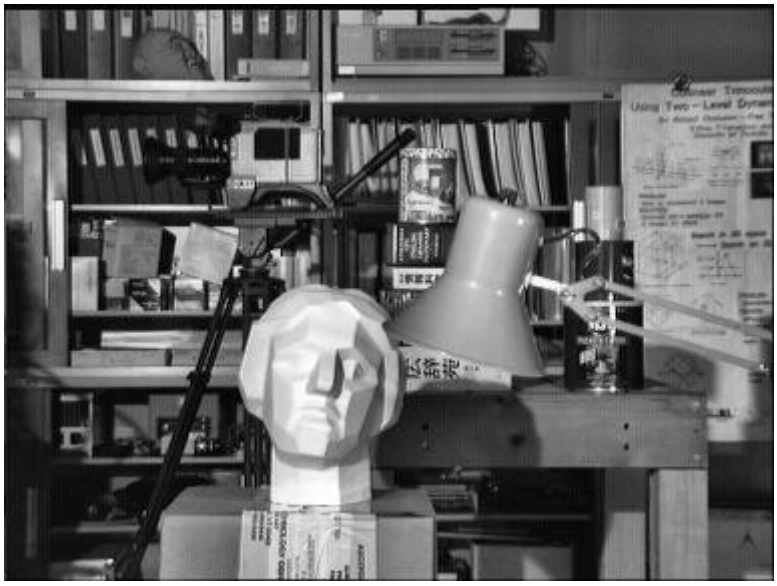


Fig.95: Ejemplo de ecualización adaptativa con Clahe.

7.15.8.- Histogramas 2D

En la anterior sección, hemos calculado y trazado el histograma unidimensional. Se llama unidimensional porque solo tenemos en cuenta una característica, es decir, el valor de intensidad de escala de grises del píxel. Pero, como vamos a ver, en los histogramas bidimensionales, se

consideran dos características. Normalmente se usa para encontrar histogramas de color donde las dos características de interés son los valores de Matiz (*H*) y Saturación (*S*) de cada píxel.

El matiz (o Hue en inglés) define el color mientras que la saturación indica cuanto gris hay en el color. Así, un valor de saturación de 0 indica un tono mayormente gris mientras que un valor de 255 indica un tono blanco.

En esta sección comprenderemos cómo crear dicho histograma de color, lo cual resultará útil para comprender otros temas, como los histogramas de retroproyección.

7.15.8.1.- Histogramas 2D en Open CV

Es bastante sencillo y se calcula utilizando la misma función, ***cv2.calcHist()***. Para los histogramas de color, necesitamos convertir la imagen de RGB a HSV (Recordar que para el histograma 1D, se ha convertido de RGB a escala de grises). Para los histogramas 2D, sus parámetros los indicaremos de la siguiente manera:

channels = [0,1] porque necesitamos procesar el plano H y S.
bin = [180,256] 180 para el plano H y 256 para el plano S.
range = [0,180,0,256] El valor del tono se encuentra entre 0 y 180 y la saturación se encuentra entre 0 y 256.

Probamos el siguiente código en una imagen:

```
import cv2
import numpy as np

img = cv2.imread('ejemplo.jpg')
hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV)

hist = cv2.calcHist([hsv], [0, 1], None, [180, 256], [0, 180, 0, 256])
```

7.15.8.2.- Histogramas 2D en Numpy

Numpy también posee una función específica para esto: ***np.histogram2d()***. (Recordar, para el histograma 1D utilizamos ***np.histogram()***).

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('ejemplo.jpg')
hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV)

hist,                    xbins,                    ybins                    =
np.histogram2d(h.ravel(),s.ravel(),[180,256],[[0,180],[0,256]])
```

El primer parámetro es el plano H, el segundo es el plano S, el tercero es el número de ***bins*** para cada uno y el cuarto es su ***rango***.

Ahora podemos verificar cómo trazar este histograma de color.

7.15.8.3.- Graficando histogramas 2D

Método 1: Utilizando cv2.imshow()

El resultado obtenido en ***hist*** es una matriz bidimensional de tamaño 180×256. Para mostrarlo podemos, sencillamente, utilizar la función ***cv2.imshow()***. La imagen resultante estará en escala de grises y no dará mucha idea de qué colores hay, a menos que conozca los valores de H correspondientes a cada color.

Método 2: Utilizando Matplotlib

A pode usar la función ***matplotlib.pyplot.imshow()*** para trazar un histograma 2D con diferentes mapas de colores. Este tipo de gráfico da una idea mucho mejor acerca de la diferencia en densidad de píxeles. No obstante, este método tampoco da idea de qué colores (a menos que conozcas qué valores de H corresponden a cada color) están presentes en la imagen. Aún así este método se prefiere sobre el anterior, ya que es más sencillo.

Nota: Recordar, cuando usemos esta función, el flag de interpolación debe fijarse como *nearest* para obtener mejores resultados.

Consideremoa el siguiente código:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('ejemplo.jpg')
hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV)
hist = cv2.calcHist( [hsv], [0, 1], None, [180, 256], [0, 180, 0, 256] )

plt.imshow(hist,interpolation = 'nearest')
plt.show()
```

A continuación se vemos la imagen de entrada y su gráfico de histograma de color. El eje *X* muestra valores S y el eje *Y* muestra la Tonalidad, H.

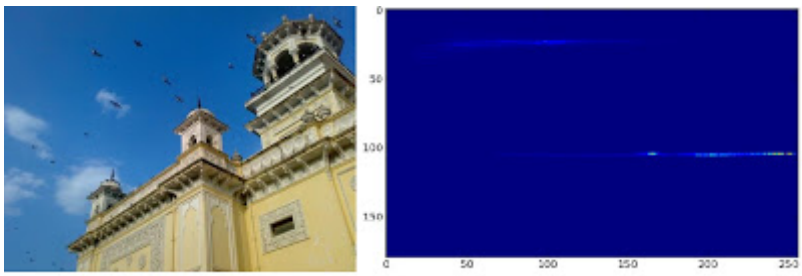


Fig.96: Ejemplo de histograma 2D.

En el histograma, podemos ver algunos valores altos cerca de H = 100 y S = 200, que corresponden al azul del cielo. De manera similar, se puede ver otro pico cerca de H = 25 y S = 100, correspondientes con el amarillo del palacio.

7.15.9.- Retroproyección

Propuesto por Michael J. Swain, Dana H. Ballard en su documento “Indexación a través de histogramas de color”.

Ahora bien, ¿qué es un histograma de retroproyección? Se utiliza para la segmentación de imágenes o para encontrar objetos de interés en una imagen. En otras palabras, este método crea

una imagen del mismo tamaño (pero solo en un canal) de la imagen de entrada, donde cada píxel corresponde a la probabilidad de ese píxel de pertenecer a nuestro objeto. En otras palabras, en la imagen de salida, nuestro objeto de interés lucirá más blanco que el resto de la imagen. Los histogramas de retroproyección se utilizan con algoritmos de cambio de cámara, etc.

¿Cómo funciona? Creamos un histograma de una imagen que contiene nuestro objeto de interés (en nuestro caso, el cielo azul; ver siguiente figura). El objeto debe llenar la imagen lo más posible para obtener mejores resultados. Es preferible utilizar un histograma de color antes que un histograma en escala de grises, porque el color del objeto es una mejor forma de definir el objeto que su intensidad de escala de grises. A continuación "retro-proyectamos" este histograma sobre nuestra imagen de prueba donde necesitamos encontrar el objeto, es decir, calculamos la probabilidad de cada píxel de pertenecer al cielo y lo vemos. La salida resultante en el umbral adecuado nos dará como resultado el cielo azul aislado del resto de objetos en la imagen.

7.15.9.1.- Algoritmo en Numpy

1. Primero debemos calcular el histograma de color tanto del objeto que necesitamos encontrar (llamémosle 'M') como de la imagen donde vamos a buscar (llamémosle 'I').

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

#roi es el objeto o región de la imagen que queremos encontrar
roi = cv2.imread('cielo_ROI.png')
hsv = cv2.cvtColor(roi,cv2.COLOR_BGR2HSV)

#&lt;em&gt;target&lt;/em&gt; es la imagen en la que buscamos
target = cv2.imread('parlamento.png')
hsvt = cv2.cvtColor(target,cv2.COLOR_BGR2HSV)

# Encuentra los histogramas usando calcHist. También pueden encontrarse con
np.histogram2d
M = cv2.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )
I = cv2.calcHist([hsvt],[0, 1], None, [180, 256], [0, 180, 0, 256] )
```

2. Encontrar la relación R = M/I. A continuación, volver a proyectar R, es decir, utilizar R como paleta y crear una nueva imagen con cada píxel como su correspondiente probabilidad de ser la imagen *target*, es decir, B (x, y) = R [h (x, y), s (x, y)] donde *h* es el Matiz y *s* es la saturación del píxel en (x, y). Después de eso aplica la condición B (x, y) = min [B (x, y), 1].

```
R = np.divide(M,I)
h,s,v = cv2.split(hsvt)
B = R[h.ravel(),s.ravel()]
B = np.minimum(B,1)
B = B.reshape(hsvt.shape[:2])
```

3. Ahora aplicamos una convolución con un disco circular, B = D*B, donde D es el kernel del disco.

```
disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(6,6))
cv2.filter2D(B,-1,disc,B)
B = np.uint8(B)
cv2.normalize(B,B,0,255,cv2.NORM_MINMAX)
```

4. La ubicación de intensidad máxima nos da la ubicación del objeto. Teniendo en cuenta la región de la imagen que nos interesa, y fijando un valor de umbral adecuado obtenemos un buen resultado.

```
ret,thresh = cv2.threshold(B,70,255,0)
```

7.15.9.2.- Retroproyección en Open CV

Open CV proporciona una función incorporada ***cv2.calcBackProject()***. Sus parámetros son casi los mismos que la función ***cv2.calcHist()***. Uno de sus parámetros es el histograma del objeto y tenemos que encontrarlo previamente. Además, el histograma del objeto debe normalizarse antes de pasarse a la función backproject. La función devuelve la probabilidad de la imagen. A continuación hacemos la convolución de la imagen con un kernel de disco y aplicamos el umbral. Veamos un ejemplo de aplicación de este método y su resultado:

```
import cv2
import numpy as np

roi = cv2.imread('cielo_ROI.png')
hsv = cv2.cvtColor(roi,cv2.COLOR_BGR2HSV)

target = cv2.imread('parlamento.png')
hsvt = cv2.cvtColor(target,cv2.COLOR_BGR2HSV)

# calcula el histograma del objeto
roihist = cv2.calcHist([hsv],[0, 1], None, [180, 256], [0, 180, 0, 256] )

# normaliza el histograma y aplica la retroproyección
cv2.normalize(roihist,roihist,0,255,cv2.NORM_MINMAX)
dst = cv2.calcBackProject([hsvt],[0,1],roihist,[0,180,0,256],1)

# Ahora aplica la covolución con un disco
disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(6,6))
cv2.filter2D(dst,-1,disc,dst)

# Aplica un umbral y convierte la imagen en blanco y negro
ret,thresh = cv2.threshold(dst,70,255,0)
thresh = cv2.merge((thresh,thresh,thresh))
res = cv2.bitwise_and(target,thresh)

res = np.vstack((target,thresh,res))
cv2.imwrite('res.jpg',res)
```

En este ejemplo utilizamos la región dentro del rectángulo rojo como objeto de muestra.

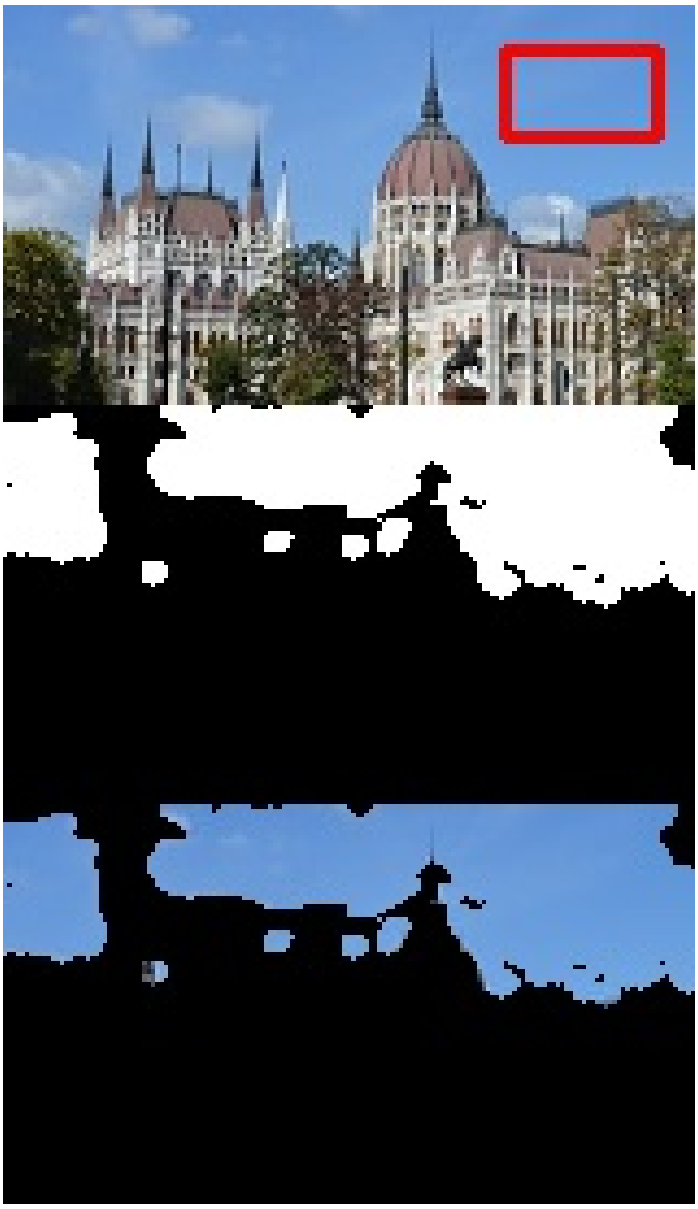


Fig.97: Ejemplo de retroproyección por Open CV.

7.15.10.- Transformada de Fourier

La transformada de Fourier se utiliza para analizar las características de frecuencia de varios filtros. Para las imágenes, la transformada discreta de Fourier 2D (DFT, por sus siglas en inglés) se utiliza para encontrar el dominio de frecuencia. Para el cálculo de la DFT se utiliza un algoritmo rápido llamado Transformada Rápida de Fourier (o Fast Fourier Transform en inglés, abreviado como FFT).

Para una señal sinusoidal de la forma:

$$x(t) = A\sin(2\pi ft)$$

donde ***f*** es la frecuencia de la señal, y si tomamos su dominio de frecuencia, podemos ver un pico en ***f***. Si muestreamos la señal para formar una señal discreta, obtenemos el mismo dominio de frecuencia, pero periódica en el rango [-pi, pi] o [0, 2pi] (o [0, *N*] para *N* puntos de la DFT).

Una imagen se puede considerar como una señal muestreada en dos direcciones. Así que al realizar la transformada de fourier en ambas direcciones *X* e *Y* da la representación de frecuencia de la imagen.

Más intuitivamente, para la señal sinusoidal, si la amplitud varía muy rápido en poco tiempo, podemos decir que es una señal de alta frecuencia. Si varía lentamente, es una señal de baja frecuencia. Podemos aplicar la misma idea a las imágenes. ¿Dónde varía destacadamente la amplitud en las imágenes? En los puntos extremos, o ruidos. Así podemos decir que los bordes y los ruidos son contenidos de alta frecuencia en una imagen. Si no hay muchos cambios en la amplitud, se trata de una componente de baja frecuencia.

Veamos pues cómo encontrar el Transformador de Fourier.

7.15.10.1.- Transformada de Fourier en Numpy

Primero veremos cómo encontrar Transformada de Fourier usando *Numpy*. *Numpy* tiene un paquete FFT para hacer esto. La función ***np. fft. fft2()*** nos proporciona la transformación de frecuencia, la cual será una matriz compleja. Su primer argumento es la imagen de entrada, que deberá estar en escala de grises. El segundo argumento es opcional y decide el tamaño de la matriz de salida. Si es mayor que el tamaño de la imagen de entrada, la imagen de entrada se rellena con ceros antes del cálculo de FFT. Si es inferior a la imagen de entrada, se recortará la imagen de entrada. Si no se pasa ningún argumento, el tamaño de la matriz de salida será igual al de la entrada.

Una vez obtenido el resultado, la componente de frecuencia cero (componente DC) estará en la esquina superior izquierda. Si quieres ponerlo en el centro, necesitas desplazar el resultado en N/2 en ambas direcciones. Esto se hace simplemente con la función ***np. fft. fftshift()*** (Es más fácil de analizar). Una vez que se encuentre la transformación de frecuencia, se puede encontrar el espectro de magnitudes.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
magnitudFFT = 20*np.log(np.abs(fshift))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Imagen de entrada'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitudFFT, cmap = 'gray')
plt.title('FFT'), plt.xticks([]), plt.yticks([])
plt.show()
```

Fig.98: Transformada de Fourier de una imagen.

En la imagen de la derecha podemos observar una región más blanca en el centro indicando que el contenido de baja frecuencia es mayor.

Ya encontramos la transformación de frecuencia Ahora podemos hacer algunas operaciones en el dominio de la frecuencia, como el filtrado paso alto y la reconstrucción de la imagen original, es decir, encontrar su DFT inverso. Para eso, debemos eliminar las bajas frecuencias utilizando una máscara formada por una ventana rectangular (en este caso utilizaremos una de tamaño 60×60). A continuación, se aplica el desplazamiento inverso utilizando ***np.fft.ifftshift()*** para que el componente de DC vuelva a aparecer en la esquina superior izquierda. Luego se encuentra la FFT inversa utilizando la función ***np.ifft2()***. El resultado, de nuevo, será un número complejo, de la que podemos tomar su valor absoluto.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)

rows, cols = img.shape
crow,ccol = int(rows/2) , int(cols/2)
fshift[crow-30:crow+30, ccol-30:ccol+30] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.abs(img_back)

plt.subplot(131),plt.imshow(img, cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(np.abs(fshift), cmap = 'gray')
plt.title('Filtro'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(img_back,cmap = 'gray')
plt.title('Transformada inversa'), plt.xticks([]), plt.yticks([])

plt.show()
```

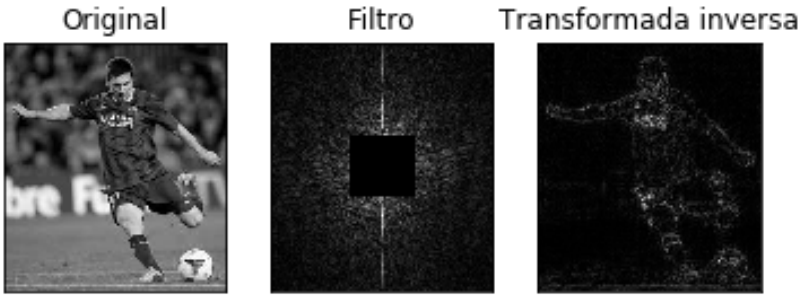


Fig.99: Transformada inversa de Fourier de una imagen.

En el resultado vemos que el Filtro Paso Altos es una operación de detección de bordes. Esto es lo que hemos visto en la sección de Gradientes de imagen. Esto también muestra que la mayoría de los datos de imagen están presentes en la región de baja frecuencia del espectro. Hasta aquí hemos visto cómo encontrar DFT, IDFT, etc. en Numpy. Ahora veamos cómo hacerlo en Open CV.

Aunque en este ejemplo no se aprecia mucho, este procedimiento puede causar artefactos en la imagen final, conocidos como *efectos de llamada* (*call effects* en inglés). Estos efectos son causados por la ventana rectangular usada para enmascarar. El problema es causado al convertir

esta máscara a la forma *seno.* Por lo tanto, no es aconsejable utilizar ventanas rectangulares para filtrar. En vez de esto, la mejor opción es utilizar ventanas Gaussianas.

7.15.10.2.- Transformada de Fourier en Open CV

Open CV dispone de las funciones ***cv2.dft()*** y ***cv2.idft()*** para esto. Estas funciones retornan el mismo resultado que las anteriores, pero con dos canales. El primer canal tendrá la parte real del resultado y el segundo canal tendrá la parte imaginaria del resultado. La imagen de entrada se debe convertir a *np.float32* primero. Veamos un ejemplo.

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('messi.jpg',0)

dft = cv2.dft(np.float32(img),flags = cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

magnitude_spectrum                                                    =
20*np.log(cv2.magnitude(dft_shift[:,:,0],dft_shift[:,:,1]))

plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Imagen de entrada'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('FFT'), plt.xticks([]), plt.yticks([])
plt.show()
```

El resultado de este código es exactamente el mismo que utilizando la función de *Numpy*.

Nota: también podemos utilizar ***cv2.cartToPolar()*** que devuelve tanto la magnitud como la fase al mismo tiempo.

Así pues, ahora hay que hallar la DFT inversa. En la sección anterior, creamos un Filtro Pasa Altos (FPA), esta vez veremos cómo eliminar los contenidos de alta frecuencia en la imagen, es decir, aplicamos un Filtro Pasa Bajos (FPB) a la imagen. El resultado de aplicar este tipo de filtro es una imagen desenfocada. Para esto, creamos una máscara primero con un valor alto (1) a bajas frecuencias, es decir, pasamos el contenido de baja frecuencia y 0 a la región altas frecuencia.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('messi.jpg',0)
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)
rows, cols = img.shape
crow,ccol = int(rows/2) , int(cols/2)

# Crea la máscara primero, el centro del cuadrado vale 1, el resto son ceros
mask = np.zeros((rows,cols,2),np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1

# Aplica la máscara y la DFT inversa
fshift = dft_shift*mask
f_ishift = np.fft.ifftshift(fshift)
img_back = cv2.idft(f_ishift)
img_back = cv2.magnitude(img_back[:,:,0],img_back[:,:,1])
```

```
plt.subplot(121),plt.imshow(img, cmap = 'gray')
plt.title('Imagen de entrada'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(img_back, cmap = 'gray')
plt.title('DFT'), plt.xticks([]), plt.yticks([])
plt.show()
```

Fig.100: Ejemplo de Transformada de Fourier por Open CV.

Nota: las funciones de *Open CV* ***cv2.dft()*** y ***cv2.idft()*** son más rápidas que las de Numpy, pero las funciones de Numpy son más fáciles de usar.

7.15.10.3.- Optimización del rendimiento de DFT

El rendimiento del cálculo DFT es mejor para algunos tamaños de matriz. Es más rápido cuando el tamaño de la matriz es potencia de dos. Las matrices cuyo tamaño es un producto de 2, 3 y 5 también se procesan de manera bastante eficiente. Por lo tanto, si nos preocupa el rendimiento de nuestro código, podemos modificar el tamaño de la matriz a cualquier tamaño óptimo (rellenando con ceros) antes de encontrar la DFT. Para Open CV, debemos agregar ceros manualmente. Pero para *Numpy*, sólo se necesita especificar el nuevo tamaño del cálculo de FFT, y automáticamente este rellenará los ceros que falten por nosotros.

Entonces, ¿cómo encontramos este tamaño óptimo? Open CV proporciona una función, ***cv2.getOptimalDFTSize()*** para esto. Es aplicable tanto a ***cv2.dft()*** como a ***np.fft.fft2()***. Comprobemos su rendimiento usando el comando mágico de Python: *% timeit*.

```
import cv2
import numpy as np

img = cv2.imread('messi.jpg',0)
rows,cols = img.shape
print(rows,cols)
<em>238 212</em>

nrows = cv2.getOptimalDFTSize(rows)
ncols = cv2.getOptimalDFTSize(cols)
print(nrows, ncols)
<em>240 216</em>
```

Notar que el tamaño (238, 212) se modifica a (240,216). Para aumentar el tamaño de la matriz rellenaremos con ceros los espacios nuevos. Esto puede hacerse creando una nueva matriz de ceros más grande (en este caso de dimensiones 240×216) y copiando los datos en ella, o utilizando ***cv2.copyMakeBorder()***.

```
nimg = np.zeros((nrows,ncols))
nimg[:rows,:cols] = img

o

right = ncols - cols
bottom = nrows - rows
bordertype = cv2.BORDER_CONSTANT #sólo para evitar la ruptura de línea en un archivo PDF
nimg = cv2.copyMakeBorder(img,0,bottom,0,right,bordertype, value = 0)
```

Ahora comparemos el rendimiento DFT de la función de *Numpy* con los dos tamaños de matriz:

```
%timeit fft1 = np.fft.fft2(img)
100 loops, best of 3: 3.64 ms per loop

%timeit fft2 = np.fft.fft2(img,[nrows,ncols])
100 loops, best of 3: 2.18 ms per loop
```

El resultado muestra una aceleración de ~1.5x. Ahora intentaremos lo mismo con las funciones de Open CV.

```
%timeit dft1= cv2.dft(np.float32(img),flags=cv2.DFT_COMPLEX_OUTPUT)
1000 loops, best of 3: 732 µs per loop

%timeit dft2= cv2.dft(np.float32(nimg),flags=cv2.DFT_COMPLEX_OUTPUT)
1000 loops, best of 3: 324 µs per loop
```

También se muestra una aceleración, en este caso de más de 2x. También podemos observar que las funciones de Open CV son alrededor de 5 veces más rápidas que las funciones de *Numpy*. Esto también se puede comprobar para la FFT inversa (se deja como ejercicio al lector).

¿Por qué Laplaciano es un filtro de paso alto?

Una de las respuestas a esta interrogante se encuentra en la Transformada de Fourier. Simplemente tome la transformada de Fourier del filtro Laplaciano para un tamaño mayor de FFT y analícelo:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# filtro promedio simple sin parámetro de escala
mean_filter = np.ones((3,3))

# crea un filtro Gaussiano
x = cv2.getGaussianKernel(8,2)
gaussian = x*x.T

# Diferentes filtros de detección de bordes
# scharr en la dirección X
scharr = np.array([[-3, 0, 3],
                   [-10,0,10],
                   [-3, 0, 3]])
# sobel en la dirección X
sobel_x= np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])
# sobel en la dirección Y
sobel_y= np.array([[-1,-2,-1],
                   [0, 0, 0],
```

```
                    [1, 2, 1]])
# laplaciano
laplacian=np.array([[0, 1, 0],
                    [1,-4, 1],
                    [0, 1, 0]])

filters = [mean_filter, gaussian, laplacian, sobel_x, sobel_y, scharr]
filter_name = ['filtro_media', 'gaussiano','laplaciano', 'sobel_x', \
                'sobel_y', 'scharr_x']
fft_filters = [np.fft.fft2(x,[200,200]) for x in filters]
fft_shift = [np.fft.fftshift(y) for y in fft_filters]
mag_spectrum = [np.log(np.abs(z)+1) for z in fft_shift]
#
for i in range(6):
    plt.subplot(2,3,i+1),plt.imshow(mag_spectrum [i],cmap = 'gray')
    plt.title(filter_name[i]), plt.xticks([]), plt.yticks([])

plt.show()
```

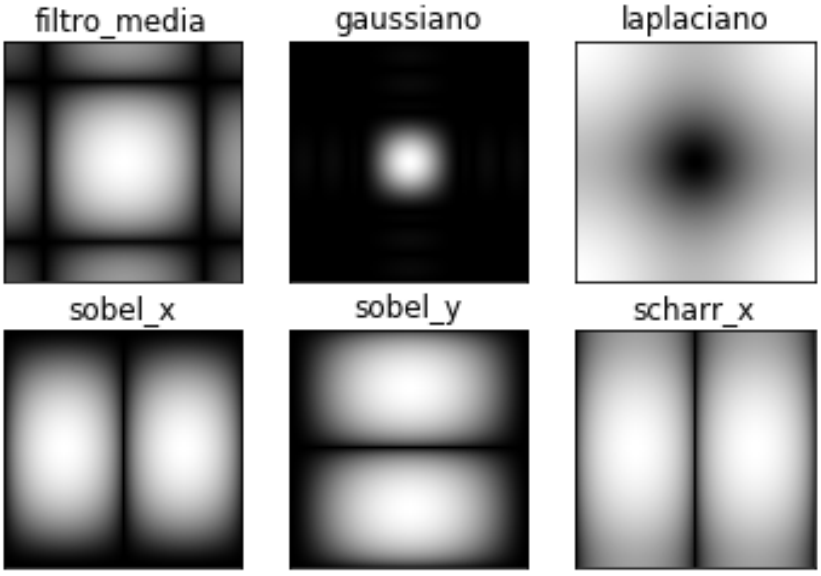


Fig.101: Ejemplo de filtro Laplaciano como filtro paso alto.

A partir de la imagen, podemos ver qué región de frecuencia bloquea cada núcleo y qué región pasa. A partir de esa información, podemos decir por qué cada núcleo es un FPB o un FPA.

7.15.11.- Emparejamiento de plantillas

El emparejamiento de plantillas (o template matching en inglés) es un método para buscar y encontrar la ubicación de una imagen de plantilla en una imagen más grande. Open CV dispone de la función ***cv2.matchTemplate()*** para este propósito. Esta función, desliza la imagen de la plantilla sobre la imagen de entrada (como en la convolución 2D) y en cada punto compara la plantilla con la porción correspondiente de la imagen de entrada. En Open CV están implementados varios métodos de comparación. La función devuelve una imagen en escala de grises, donde cada píxel indica cuánto coincide el entorno de ese píxel con la plantilla.

Si la imagen de entrada es de tamaño (*W*x*H*) y la imagen de la plantilla es de tamaño (*w*x*h*), la imagen de salida tendrá un tamaño de (*W*-*w* + 1, *H*-*h* + 1). Una vez obtengamos el resultado, podemos usar la función ***cv2.minMaxLoc()*** para encontrar dónde está el valor máximo / mínimo. El valor máximo/ mínimo corresponde a la esquina superior izquierda del rectángulo con ancho *w* y alto *h*. Ese rectángulo será la región de la imagen de entrada que mejor coincide con la plantilla.

Nota: Si utilizamos cv2.TM_SQDIFF como método de comparación, el valor mínimo dará la mejor coincidencia.

7.15.11.1.- Emparejamiento de plantillas en Open CV

A continuación compararemos el desempeño de diferentes métodos de emparejamiento de la función ***cv2.matchTemplate(),*** para encontrar la cara de un hombre entre los granos de café:

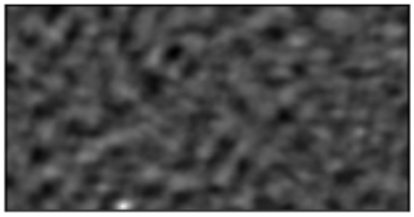
Fig.102: Imagen de referencia.

A continuación, el código que hace esto:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('imagen.png',0)
img2 = img.copy()
template = cv2.imread('plantilla.png',0)
w, h = template.shape[::-1]

# All the 6 methods for comparison in a list
methods = ['cv2.TM_CCOEFF', 'cv2.TM_CCOEFF_NORMED', 'cv2.TM_CCORR',
            'cv2.TM_CCORR_NORMED', 'cv2.TM_SQDIFF', 'cv2.TM_SQDIFF_NORMED']

for meth in methods:
    img = img2.copy()
    method = eval(meth)

    # Aplica el emparejamiento de plantillas
    res = cv2.matchTemplate(img,template,method)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)

    # Si el método es TM_SQDIFF o TM_SQDIFF_NORMED, tomar el mínimo
    if method in [cv2.TM_SQDIFF, cv2.TM_SQDIFF_NORMED]:
        top_left = min_loc
    else:
        top_left = max_loc
    bottom_right = (top_left[0] + w, top_left[1] + h)

    cv2.rectangle(img,top_left, bottom_right, 255, 10)

    plt.subplot(121),plt.imshow(res,cmap = 'gray')
    plt.title('Resultado del emparejamiento'), plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(img,cmap = 'gray')
    plt.title('Punto detectado'), plt.xticks([]), plt.yticks([])
    plt.suptitle(meth)

    plt.show()
```

cv2.TM_CCOEFF

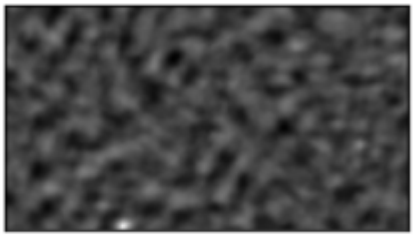
Resultado del emparejamiento



Punto detectado

cv2.TM_CCOEFF_NORMED

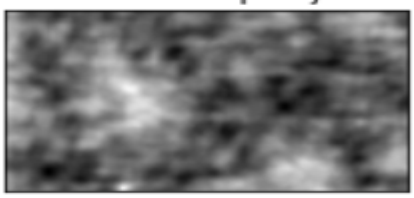
Resultado del emparejamiento



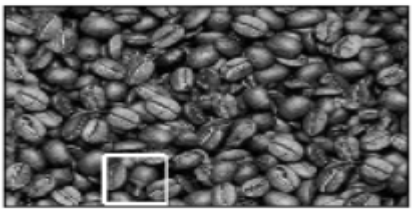
Punto detectado

cv2.TM_CCORR

Resultado del emparejamiento

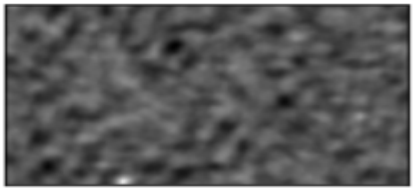


Punto detectado

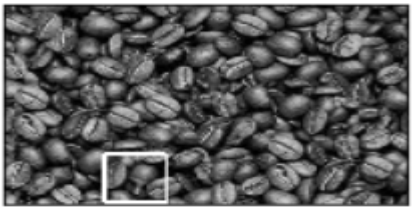


cv2.TM_CCORR_NORMED

Resultado del emparejamiento



Punto detectado



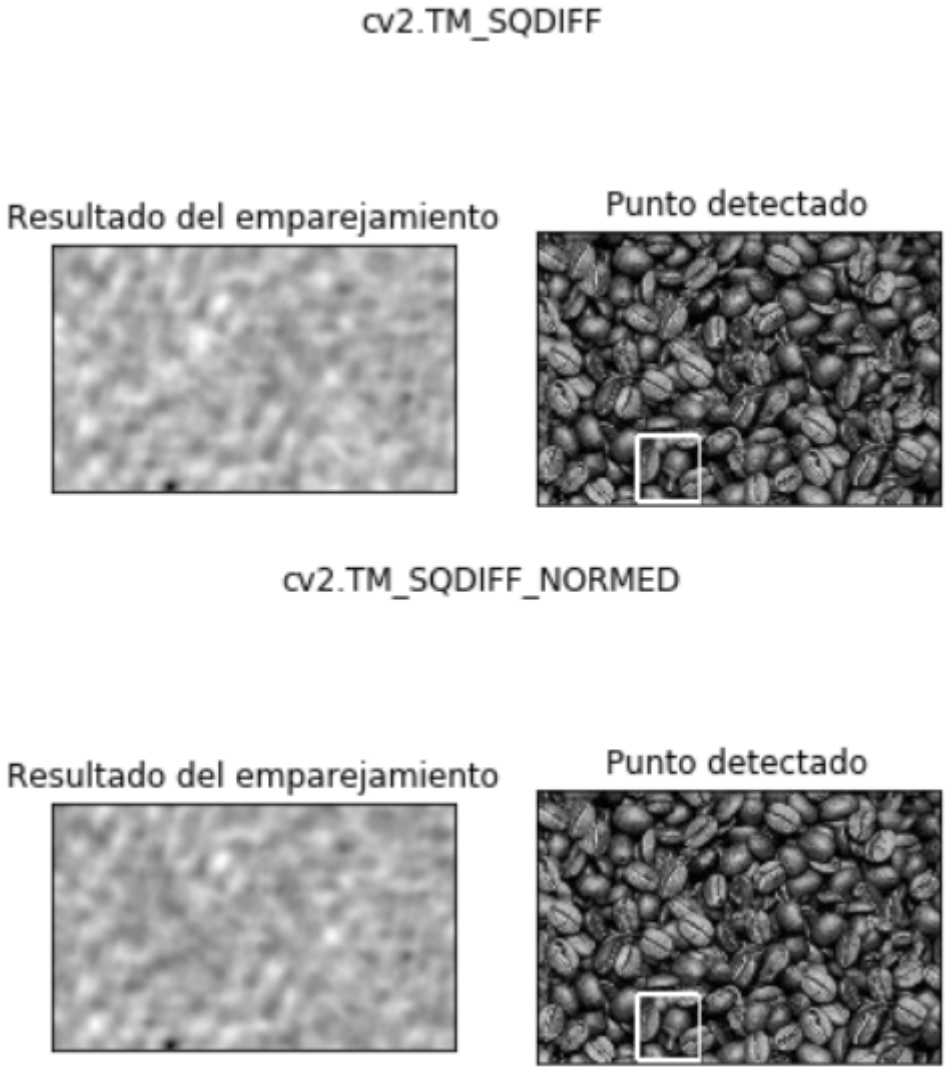


Fig.103: Ejemplo de emparejamiento de plantillas.

En este caso observamos que los seis métodos dan resultados similares. Sin embargo, esto puede variar dependiendo de la imagen y la plantilla en particular. Notar que en los cuatro primeros gráficos a la izquierda el punto de máxima coincidencia es blanco (correspondiente con un máximo) mientras que, con los últimos dos métodos el punto de máxima coincidencia es negro (correspondiente con un mínimo).

7.15.11.2.- Emparejamiento de plantillas con múltiples objetos

En la sección anterior, buscamos en la imagen la cara de un hombre, que aparece solo una vez en la imagen. Supongamos que está buscando un objeto que tiene múltiples ocurrencias, ***cv2.minMaxLoc()*** no nos dará todas las ubicaciones. En ese caso, fijaremos un valor umbral por encima (o por debajo, dependiendo del método que usemos) del cual se asumirá que el objeto en la plantilla coincide con el objeto en la imagen. A continuación un ejemplo, en el que se muestra una captura de pantalla del famoso juego *Mario.* Utilizaremos el método explicado para encontrar todas las monedas.

```
import cv2

import numpy as np
from matplotlib import pyplot as plt
```

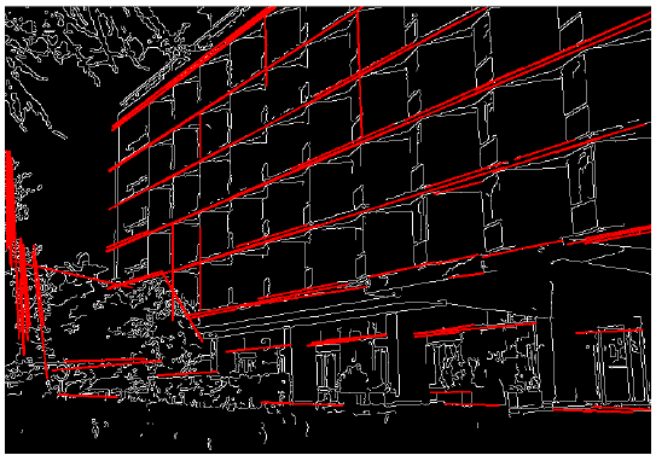
```
img_rgb = cv2.imread('mario.png')
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)
template = cv2.imread('moneda.png',0)
w, h = template.shape[::-1]

res = cv2.matchTemplate(img_gray,template,cv2.TM_CCOEFF_NORMED)
umbral = 0.8
loc = np.where( res &gt;= umbral)
for pt in zip(*loc[::-1]):
    cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)

cv2.imwrite('res.png',img_rgb)
```

Fig.104: Ejemplo de emparejamiento múltiple.

7.15.12.- Transformada de línea de Hough

La transformada de Hough es una técnica popular para detectar cualquier forma, siempre que se pueda representar esa forma matemáticamente. Este algoritmo puede detectar una forma determinada incluso si está rota o un poco distorsionada.

Función utilizada para detectar figuras en una imagen digital que pueden ser expresadas matemáticamente, tales como rectas, círculos o elipses.



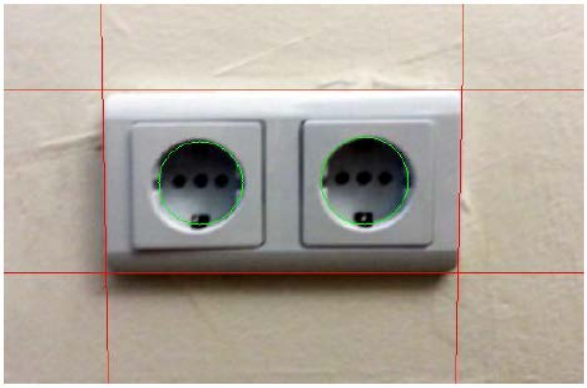


Fig.105: Ejemplos de Transformada de Hough.

Tener en cuenta que, debido a las imperfecciones, ya sea de la imagen captada o del detector de bordes, existen muchos puntos que pertenecen a la línea y que faltan en la imagen; también pueden existir separaciones espaciales entre la figura ideal, por ejemplo, una recta y los puntos ruidosos del borde detectado.

Le función de la transformada de Hough es resolver este problema, haciendo posible realizar agrupaciones de los puntos que pertenecen a los bordes de posibles figuras a través de un procedimiento de votación sobre un conjunto de figuras parametrizadas.

Será de nuestra atención establecer el umbral de votos para considerar que una línea existe en una imagen.

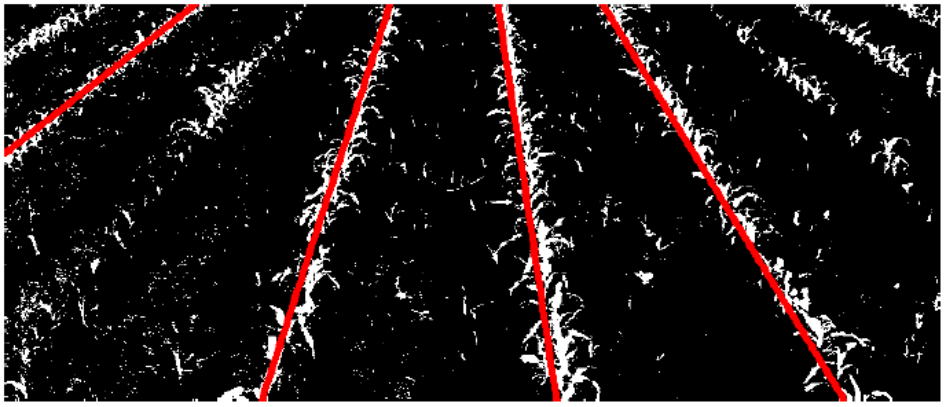


Fig.106: Ejemplo de Transformada de Hough sobre líneas rectas.

Recta:

Para poder representar todas las rectas posibles que puedan aparecer en la imagen, se puede utilizar la recta en coordenadas polares.

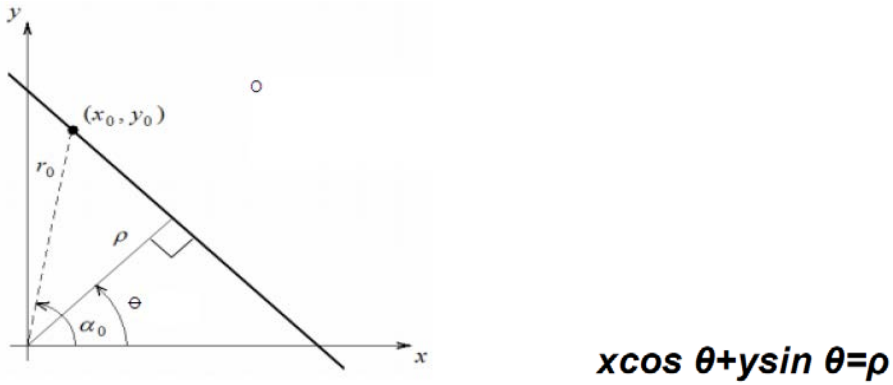


Fig.107: Operación sobre rectas en la Transformada de Hough.

Tendremos el rango natural de $\theta \in [0, 2\pi]$. A continuación, se transforma cada punto (x,y) de la imagen de origen, en los puntos (ρ_i, θ_i), el espacio definido por (ρ, θ) se denomina espacio de Hough para el conjunto de rectas en dos dimensiones.

Para un punto arbitrario en la imagen con coordenadas (x_0, y_0), las rectas que pasan por ese punto son las polares (ρ, θ) con $r=x*\cos\theta + y*\text{sen}\theta$ donde ρ (la distancia entre la línea y el origen) está determinado por θ. Esto corresponde a una curva senoidal en el espacio (ρ, θ) que es única para ese punto.

Si las curvas correspondientes a dos puntos se interceptan, el punto de intersección en el espacio de Hough corresponde a una línea en el espacio de la imagen que pasa por estos puntos.

Generalizando, un conjunto de puntos que forman una recta, producirán sinusiodes que se interceptan en los parámetros de esa línea.

El algoritmo de la transformada de Hough utiliza una matriz, llamada acumulador, cuya dimensión es igual al número de parámetros desconocidos del problema, en el caso de una recta la dimensión del acumulador será dos, correspondientes a los valores cuantificados para (ρ, θ).

Para construir el acumulador es necesario discretizar los parámetros que describen la figura. Cada celda del acumulador representa una figura cuyos parámetros se pueden obtener a partir de la posición de la celda.

Cada punto de la imagen vota por las posibles rectas a las que puede pertenecer ese punto. Esto se logra buscando todas las posibles combinaciones de valores para parámetros que describen la figura (los posibles valores se obtienen a partir del acumulador).

Si es así, se calculan los parámetros de esa figura, y después se busca la posición en el acumulador correspondiente a la figura definida, y se incrementa el valor que hay en esa posición.

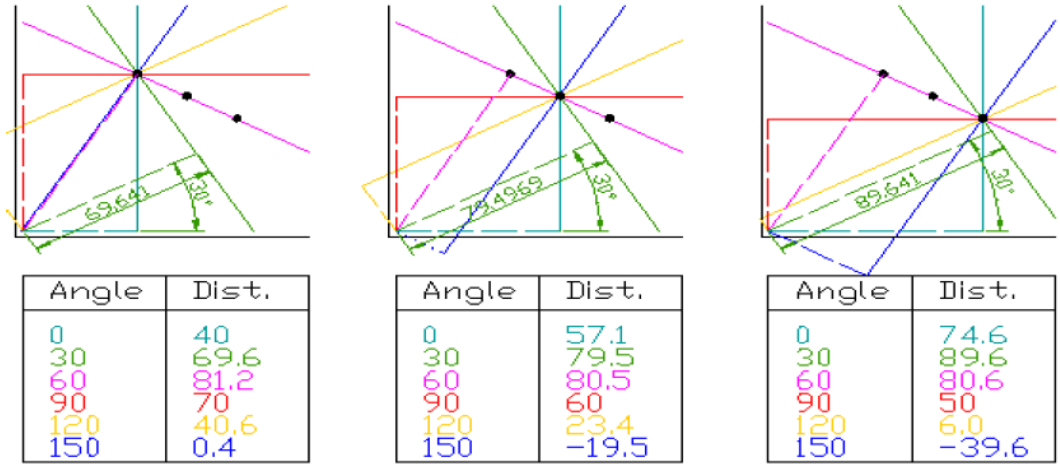
Las figuras se pueden detectar buscando las posiciones del acumulador con mayor valor (máximos locales en el espacio del acumulador). La forma más sencilla de encontrar estos picos es aplicando alguna forma de umbral.

RECTA: Algoritmo

1: cargar imagen
2: detectar los bordes en la imagen
3: por cada punto en la imagen:
4: si el punto (x,y) esta en un borde:
5: por todos los posibles ángulos θ :
6: calcular ρ para el punto (x,y) con un ángulo θ
7: incrementar la posición (ρ, θ) en el acumulador
8: buscar las posiciones con los mayores valores en el acumulador
9: devolver las rectas cuyos valores son los mayores en el acumulador.

Fig.108: Algoritmo de Hough para una recta.

Considerar tres puntos, mostrados en la siguiente figura como puntos negros.



Angle	Dist.
0	40
30	69.6
60	81.2
90	70
120	40.6
150	0.4

Angle	Dist.
0	57.1
30	79.5
60	80.5
90	60
120	23.4
150	−19.5

Angle	Dist.
0	74.6
30	89.6
60	80.6
90	50
120	6.0
150	−39.6

Fig.109: Ejemplo de detección de línea por el algoritmo de Hough.

Por cada punto se dibujan un número de líneas que pasan por los mismos, con distintos ángulos. Son las líneas continuas. Por cada línea se dibuja una recta perpendicular a esta que pasa por el origen de coordenadas. Son las líneas discontinuas.

Se calcula la longitud y el ángulo de cada línea discontinua. Los resultados se muestran en las tablas.

Se crea un grafo con las longitudes de las líneas por cada ángulo, conocido como grafo del espacio de Hough.

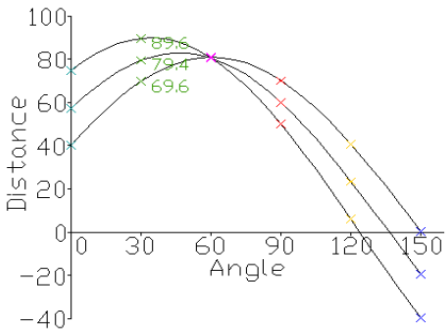


Fig.110: Grafo del espacio de Hough.

El punto donde se interceptan las curvas da la distancia y el ángulo. Esta distancia y este ángulo indican la recta que se intercepta con los puntos anteriores. El grafo muestra el punto rosado donde se interceptan las curvas, este punto corresponde a la recta rosada de la figura, que pasa por los tres puntos negros.

7.15.12.1.- Tranformada de Hough en OpenCV

Todo lo indicado anteriormente está recogido en la función de Open CV, ***cv2.HoughLines()***. Simplemente retorna una matriz de valores (ρ, θ). donde se mide en píxeles y se mide en radianes. El primer parámetro, la imagen de entrada, debe ser una imagen binaria, por lo tanto, aplique el umbral o use la detección de bordes astutos antes de aplicar la transformada de Hough. Los parámetros segundo y tercero son las precisiones en y respectivamente. El cuarto argumento es el umbral, es decir, el acumulado mínimo que debe obtener para que se le considere como una línea. Recordar, el número de votos (acumulados) dependerá del número de puntos en la línea. Por lo tanto, este número de votos representa la longitud mínima de línea a detectar.

```
import cv2
import numpy as np

img = cv2.imread('sudoku.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)

lines = cv2.HoughLines(edges,1,np.pi/180,200)
for rho,theta in lines[0]:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))

    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)

cv2.imwrite('houghlines1.jpg',img)
```

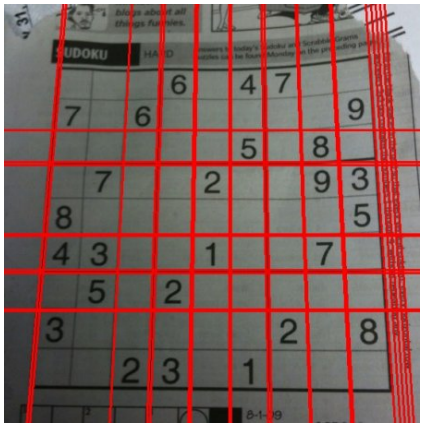


Fig.111: Ejemplo de Transformada de Hough por Open CV.

7.15.13.- Transformada Probabilística de Hough

En la transformación de Hough, podemos ver que incluso para una línea con dos argumentos, se necesita mucha computación. La transformada probabilística de Hough es una optimización de la Transformada de Hough que acabamos de ver. No toma todos los puntos en consideración, en

su lugar toma solo un subconjunto de puntos al azar y eso es suficiente para la detección de línea. Sólo se necesita disminuir el umbral. Veamos la imagen a continuación que compara la Transformada de Hough y la Transformada Probabilística de Hough en un espacio amplio.

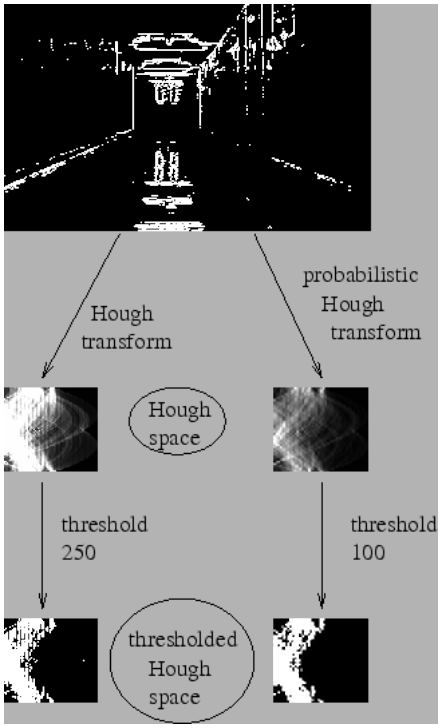


Fig.112: Transformada probabilística de Hough.

La implementación de Open CV se basa en la detección robusta de líneas usando la Transformación probabilística progresiva de Hough, de *Matas, J., Galambos, C.* y *Kittler, J.V* . La función utilizada es ***cv2.HoughLinesP()***, la cual posee dos nuevos argumentos:

- *MinLineLength*: longitud mínima de la línea. Los segmentos de línea más cortos que esto son rechazados.
- *maxLineGap*: espacio máximo permitido entre los segmentos de línea para tratarlos como una sola línea.

Lo mejor es que retorna directamente los dos puntos finales de las líneas. En el caso anterior, solo la función sólo retorna los parámetros de las líneas, y uno tiene que encontrar todos los puntos. En este caso, todo es mucho más directo y simple.

```
import cv2
import numpy as np

img = cv2.imread('sudoku.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray,50,150,apertureSize = 3)
minLineLength = 100
maxLineGap = 10
lines = cv2.HoughLinesP(edges,1,np.pi/180,100,minLineLength,maxLineGap)
for x1,y1,x2,y2 in lines[0]:
    cv2.line(img,(x1,y1),(x2,y2),(0,255,0),2)

cv2.imwrite('houghlines2.jpg',img)
```

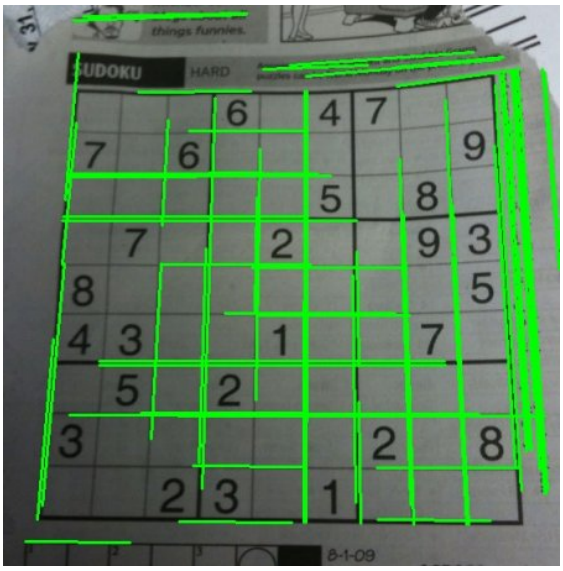


Fig.113: Ejemplo de Transformada probabilística de Hough.

7.15.14.- Transformada de círculo de Hough

Un círculo se representa matemáticamente como el centro del círculo, y r es el radio del círculo. A partir de la ecuación, podemos ver que tenemos tres parámetros, por lo que necesitamos un acumulador 3D para la transformada de Hough, que sería altamente ineficaz. Así que tenemos, el Método *Hough Gradient*, que utiliza la información de degradado de los bordes.

La función que aquí utilizada es ***cv2.HoughCircles()***. Tiene muchos argumentos que están bien explicados en la documentación. Veamos un ejemplo directamente:

```
import cv2
import numpy as np

img = cv2.imread('monedas.png',0)
img = cv2.medianBlur(img,5)
cimg = cv2.cvtColor(img,cv2.COLOR_GRAY2BGR)

circles = cv2.HoughCircles(img,cv2.HOUGH_GRADIENT,3,30,
                           param1=80,param2=20,minRadius=10,maxRadius=40)

circles = np.uint16(np.around(circles))
for i in circles[0,:]:
    # Dibuja la circusnferencia del círculo
    cv2.circle(cimg,(i[0],i[1]),i[2],(0,255,0),2)
    # dibuja el centro del círculo
    cv2.circle(cimg,(i[0],i[1]),2,(0,0,255),3)

cv2.imshow('círculos detectados',cimg)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

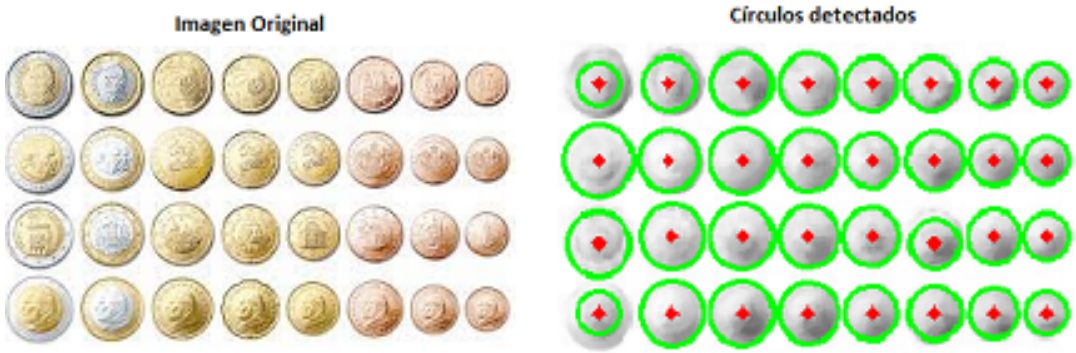


Fig.114: Identificación de formas por Transformada de círculo de Hough.

7.15.15.- Segmentación de imágenes con el algoritmo Watershed

Cualquier imagen en escala de grises la podemos ver como una superficie topográfica donde una intensidad alta indica picos y colinas, mientras que intensidades bajas indican valles. En este algoritmo se empieza por llenar cada valle aislado (mínimos locales) con agua de diferentes colores (etiquetas). A medida que el agua sube, dependiendo de los picos (pendientes) cercanos, el agua de diferentes valles, obviamente con diferentes colores, comenzará a fusionarse. Para evitar esto, se construyen barreras en los lugares donde se une el agua. A continuación, se continúa el trabajo de rellenar con agua y construir barreras hasta que todos los picos estén bajo el agua. Así, las barreras creadas en este proceso, no son más que la segmentación de la imagen. En esto se basa el algoritmo *Watershed* (En español: línea de división de aguas).

Sin embargo, este enfoque reporta un resultado sobresegmentado debido al ruido o a cualquier otra irregularidad en la imagen. Así que Open CV implementó un algoritmo de cuenca hidrográfica basado en marcadores en el que se especifican cuáles son todos los puntos del valle que se fusionarán y cuáles no. Corresponde a una segmentación de imagen interactiva. Lo que hacemos es dar diferentes etiquetas a nuestro objeto. De este modo, tendremos que etiquetar la región que estamos seguros de que es el primer plano o el objeto en sí con un color (o intensidad), y debemos etiquetar la región de la que estamos seguros que es el fondo y no el objeto, con otro color. Finalmente la región de la que no estamos seguros de nada la debemos etiquetar con 0. Ese es nuestro marcador. Sólo después de este etiquetado aplicamos el algoritmo *Watershed*. Entonces nuestro marcador se actualizará con las etiquetas que dimos, y los límites de los objetos tendrán un valor de -1.

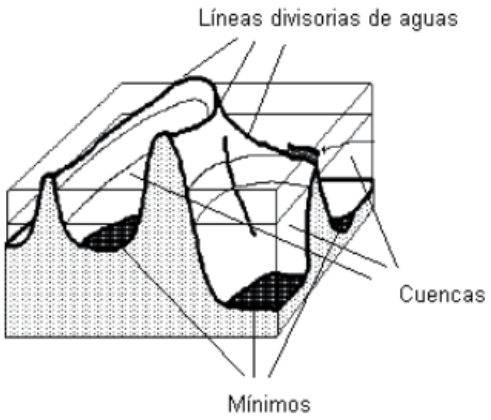


Fig.115: Esquema base del algoritmo de Watershed. Fuente: Implementación del algoritmo de Watershed para el análisis de imágenes médicas. Revista de Investigación de Sistemas e Informática. Facultad de Ingeniería de Sistemas e Informática. Universidad Nacional Mayor de San Marcos. Perú. 2011.

A continuación, un ejemplo sobre cómo usar la Transformación de distancia junto con el *Watershed* para segmentar objetos que se tocan mutuamente.

Consideremos la imagen de las monedas, las monedas se tocan entre sí. Incluso si lo limitamos, se tocarán entre sí.

Fig.116: Imagen base para aplicar en algoritmo de Watershed.

Comenzamos por encontrar una estimación aproximada de las monedas. Para eso, podemos usar la binarización de *Otsu*.

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('monedas.png')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

El resultado:

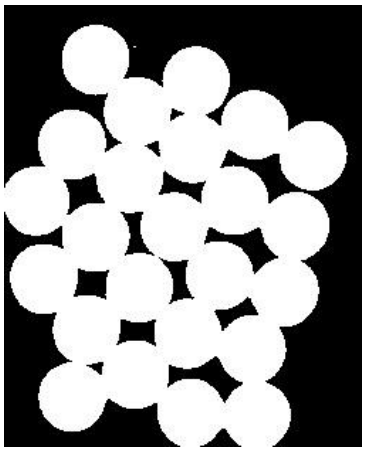


Fig.117: Estimación de las monedas en aplicación del algoritmo de Watershed.

Ahora se necesita eliminar cualquier pequeño ruido blanco en la imagen. Para eso podemos usar la apertura morfológica. Para eliminar cualquier agujero pequeño en el objeto, podemos usar el cierre morfológico. Por lo tanto, ahora sabemos con certeza que la región cercana al centro de los objetos está en primer plano y que la región más alejada del objeto es el fondo. Solo la región de la que no estamos seguros es la región límite de las monedas.

Entonces necesitamos extraer el área de la cual estamos seguros que son monedas. La erosión elimina los píxeles del límite. Entonces, lo que quede, podemos estar seguros de que es una moneda. Eso funcionaría si los objetos no se tocaran entre sí. Pero como se están tocando entre sí, otra buena opción sería encontrar la distancia de transformación y aplicar un umbral adecuado. Luego tenemos que encontrar el área que estamos seguros de que no son monedas. Para ello, aplicamos dilatación al resultado. La dilatación aumenta el límite del objeto al fondo. De esta manera, podemos asegurarnos de que cualquier región en el fondo en el resultado sea realmente un fondo, ya que la región límite se elimina. Veamos la imagen a continuación:

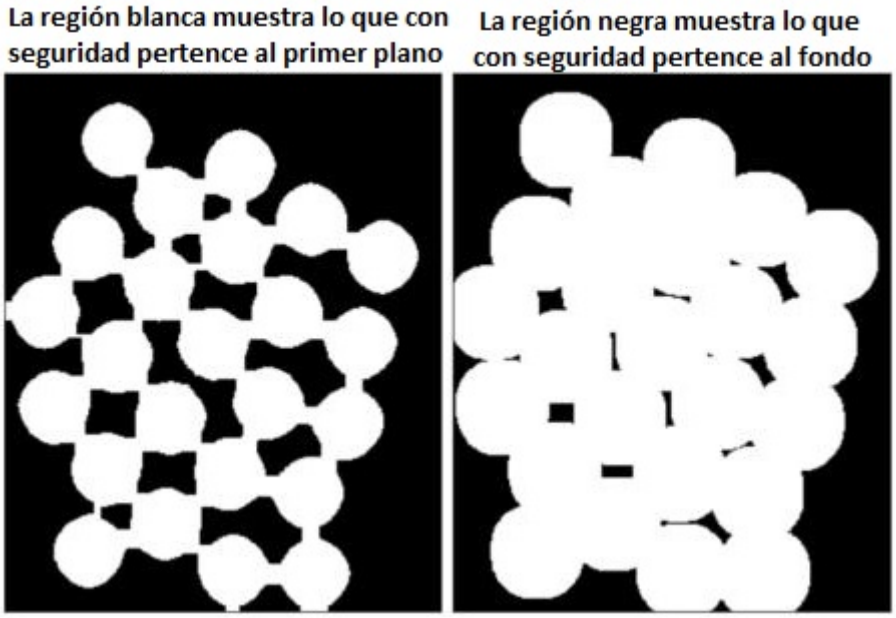


Fig.118: Procedimiento de segmentación por método de Watershed.

Las regiones restantes son aquellas de las que no tenemos conocimiento de si son monedas o fondo. El algoritmo de Watershed debería ser capaz de discriminar entre monedas y fondo en estas regiones conflictivas. Estas regiones corresponden al área alrededor de los límites de las monedas donde se cruzan el primer plano y el fondo (o incluso se encuentran dos monedas diferentes). Tales regiones delimitantes son las fronteras. Se puede obtener restando el área de la figura de la izquierda del área de la figura de la derecha.

```
# Eliminación del ruido
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)

# Encuentra el área del fondo
sure_bg = cv2.dilate(opening,kernel,iterations=3)

# Encuentra el área del primer
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
ret, sure_fg = cv2.threshold(dist_transform,0.7*dist_transform.max(),255,0)

# Encuentra la región desconocida (bordes)
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)
```

Veamos el resultado en la imagen a la que se ha aplicado un umbral; se obtienen algunas regiones de monedas de las cuales estamos seguros de las monedas y que ahora además están separadas.

En algunos casos, puede interesarnos la segmentación de solo el primer plano, y no en separar los objetos que se tocan mutuamente. En ese caso, no necesitamos usar la transformación de distancia, basta con la erosión. La erosión es solo otro método para extraer el área de primer plano.

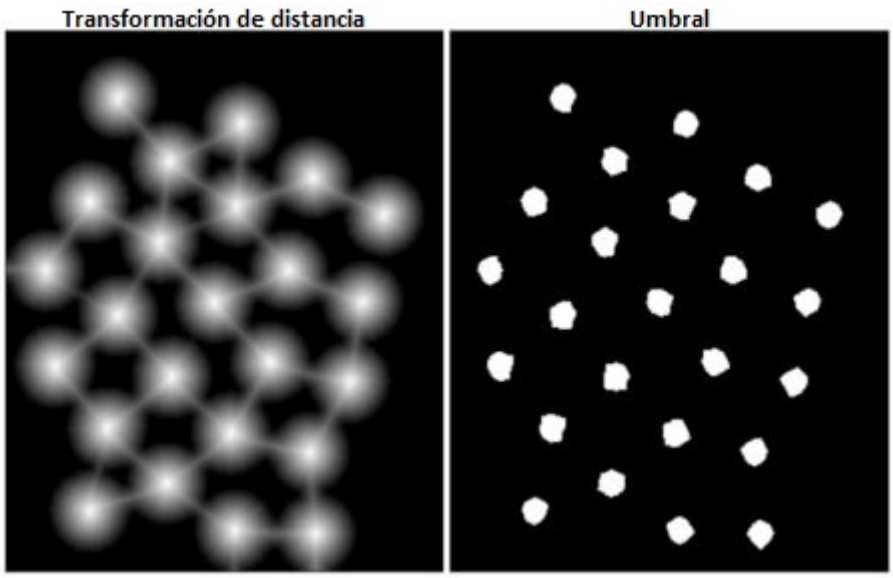


Fig.119: Procedimiento de segmentación por método de Watershed (Cont.).

Ahora sabemos con certeza cuáles son las regiones de las monedas, que es parte del fondo y el resto. Ahora creamos marcador (es un array del mismo tamaño que el de la imagen original, pero con el tipo de datos int32) y etiquetamos las regiones dentro de él. Las regiones que sabemos con certeza (ya sea en primer plano o en segundo plano) están etiquetadas con números enteros positivos, pero enteros diferentes, y el área que no sabemos con certeza simplemente queda en cero. Para esto usamos ***cv2.connectedComponents()***. Con esta función etiquetamos el fondo de la imagen con 0, y el resto de los objetos quedan etiquetados con números enteros a partir de 1.

Pero sabemos que si el fondo está marcado con 0, el algoritmo de *Watershed* lo considerará como un área desconocida. Por tanto, queremos marcarlo con un número entero diferente. En cambio, marcaremos la región desconocida, definida como *unknown*, con 0.

```
# Etiquetado
ret, markers = cv2.connectedComponents(sure_fg)

# Adiciona 1 a todas las etiquetas para asegurra que el fondo sea 1 en lugar
de cero
markers = markers+1

# Ahora se marca la región desconocida con ceros
markers[unknown==255] = 0
```

Veamos el resultado que se muestra en el mapa de colores JET. La región azul oscura muestra la región desconocida. Las monedas están coloreadas con diferentes valores. El área restante, que es de fondo seguro, se muestra en azul claro en comparación con la región desconocida.

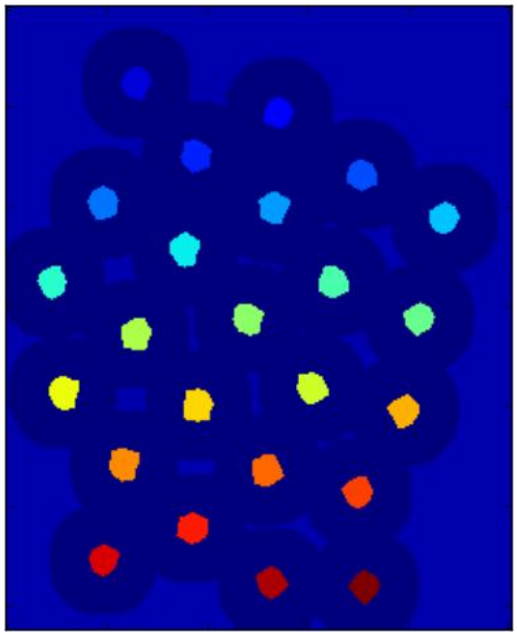


Fig.120: Procedimiento de segmentación por método de Watershed (Cont.).

Ahora nuestro marcador está listo. Es hora de dar el paso final, aplicar el algoritmo de Watershed. Al hacer esto la imagen del marcador será modificada y la región límite será etiquetada con -1.

```
markers = cv2.watershed(img,markers)
img[markers == -1] = [255,0,0]
```

Veamos el resultado a continuación. Aunque en general, el algoritmo encuentra muy bien las fronteras de las monedas, en algunos casos, en las regiones de contacto entre dos monedas, el algoritmo falla.

Fig.121: Procedimiento de segmentación por método de Watershed (Cont.).

7.15.16.- Extracción interactiva del fondo usando el algoritmo GrabCut

A raíz de necesitar un algoritmo para la extracción en primer plano con la mínima interacción por parte del usuario se generó este algoritmo. El algoritmo *GrabCut* fue diseñado por Carsten

Rother, Vladimir Kolmogorov y Andrew Blake de Microsoft Research Cambridge, Reino Unido. En su artículo, “GrabCut”: extracción interactiva en primer plano utilizando cortes iterativos de gráficos,

¿Cómo funciona desde el punto de vista del usuario? Inicialmente, el usuario dibuja un rectángulo alrededor de la región del primer plano (la región del primer plano debe estar completamente dentro del rectángulo). A continuación, el algoritmo lo segmenta de forma iterativa para obtener el mejor resultado. Sin embargo, en algunos casos, la segmentación puede que no este bien. Por ejemplo, puede haberse marcado alguna región de primer plano como fondo y viceversa. En ese caso, el usuario debe hacer retoques finos. Sólo debe dar algunos trazos en las imágenes donde hay algunos resultados defectuosos. Strokes básicamente dice “Oye, esta región debe estar en primer plano, la marcó en segundo plano, corrígela en la siguiente iteración” o su opuesto en segundo plano. A continuación, en la siguiente iteración, obtienes mejores resultados.

Veamos la imagen a continuación. El primer jugador y el fútbol están encerrados en un rectángulo azul. A continuación se realizan algunos retoques finales con trazos blancos (que denotan el primer plano) y trazos negros (que denotan el fondo). Y se obtiene un buen resultado.

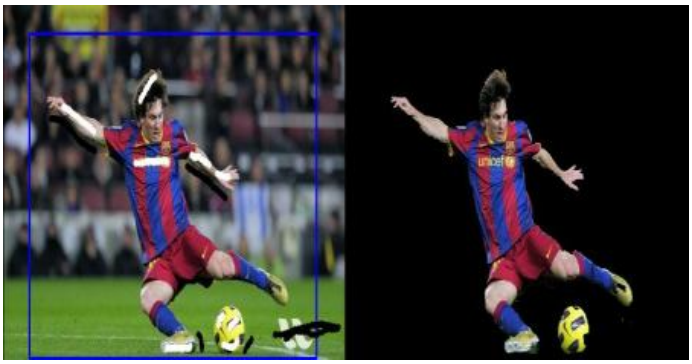


Fig.122: Principio de aplicación del Algoritmo GrabCut.

¿Pero, qué ha ocurrido por detrás?

- El usuario inserta el rectángulo. Todo lo que esté fuera de este rectángulo se tomará como fondo seguro (es por eso que se menciona antes que su rectángulo debe incluir todos los objetos). Todo dentro del rectángulo es desconocido. De forma similar, cualquier entrada de usuario que especifique el primer plano y el fondo se considera de etiqueta dura, lo que significa que no cambiará en el proceso.
- El algoritmo realiza un etiquetado inicial con los datos que proporcionamos. Etiqueta los píxeles de primer plano y de fondo (o etiquetas duras)
- Ahora se usa un Modelo de Mezcla Gaussiana (MGM) para modelar el primer plano y el fondo.
  Según los datos que proporcionamos, MGM aprende y crea una nueva distribución de píxeles. Es decir, los píxeles desconocidos están etiquetados como primer plano probable o fondo probable dependiendo de su relación con los otros píxeles con etiqueta dura en términos de estadísticas de color (es como la agrupación en clúster).
- Un gráfico se construye a partir de esta distribución de píxeles. Los nodos en los gráficos son píxeles. Se agregan dos nodos adicionales, nodo Fuente y nodo Sumidero. Cada píxel de primer plano está conectado al nodo Fuente y cada píxel de fondo está conectado al nodo Sumidero.
- Los pesos de los bordes que conectan los píxeles al nodo de origen / nodo final se definen por la probabilidad de que un píxel sea de primer plano / fondo. Los pesos entre

los píxeles están definidos por la información de borde o la similitud de píxeles. Si hay una gran diferencia en el color del píxel, el borde entre ellos obtendrá un peso bajo.

- Luego, se usa un algoritmo *mincut* para segmentar el gráfico. Corta el gráfico en dos nodos de origen separadores y receptores con una función de costo mínimo. La función de costo es la suma de todos los pesos de los bordes que se cortan. Después del corte, todos los píxeles conectados al nodo Fuente se convierten en primer plano y los conectados al nodo Sumidero se convierten en fondo.
- El proceso continúa hasta que la clasificación converge.

Este proceso se ilustra en la siguiente imagen:

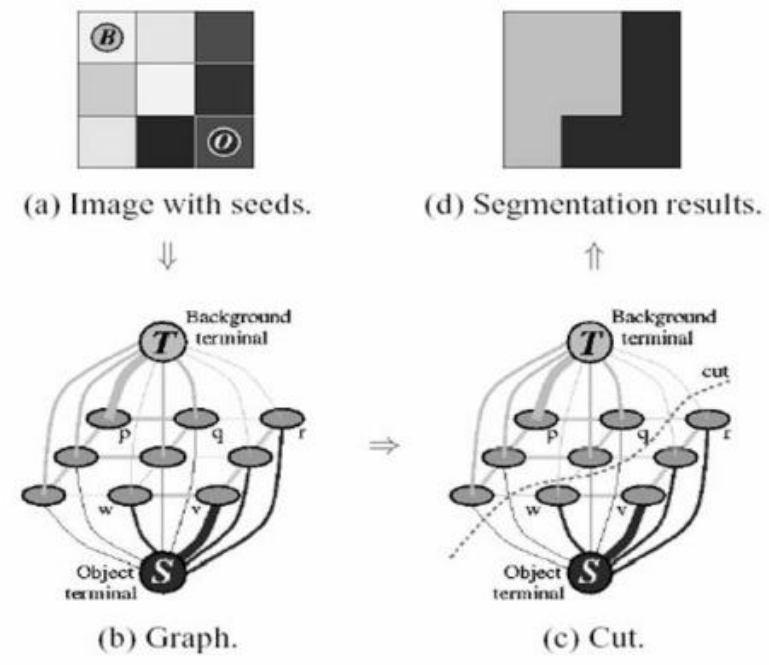


Fig.123: Estructura de aplicación del Algoritmo GrabCut.

7.15.16.1.- GrabCut en Open CV

Ahora vamos a por el algoritmo de *Grabcut* con Open CV. Disponemos de la función, *cv2.grabCut()* para esto. Primero veamos sus argumentos:

img : Imagen de entrada
mask: es una imagen de máscara donde especificamos qué áreas son fondo, primer plano o fondo probable / primer plano, etc. Se realiza mediante los siguientes indicadores, ***cv2.GC_BGD***, **c*v2.GC_FGD***, ***cv2.GC_PR_BGD***, ***cv2.GC_PR_FGD***, o simplemente pasando 0,1,2,3 a la imagen.
rect: Son las coordenadas de un rectángulo que incluye el objeto de primer plano en el formato (x, y, w, h)
bdgModel, *fgdModel* – Estas son matrices utilizadas internamente por el algoritmo.
iterCount: número de iteraciones que debe ejecutar el algoritmo.
mode: debe ser c***v2.GC_INIT_WITH_RECT*** o ***cv2.GC_INIT_WITH_MASK*** o combinado, que decide si estamos dibujando trazos de retoque rectangulares o finales.

Primero veamos con el modo rectangular. Cargamos la imagen, creamos una imagen de máscara similar. Creamos *fgdModel* y *bgdModel*. Le damos los parámetros rectangulares. Hasta ahora, todo sencillo. Dejamos que el algoritmo se ejecute durante 5 iteraciones. El modo debe ser

cv2.GC_INIT_WITH_RECT ya que estamos usando un rectángulo. A continuación, ejecutamos el *Grabcut*. Modificamos la imagen de la máscara. En la nueva imagen de máscara, los píxeles se marcarán con cuatro banderas que denotan fondo / primer plano como se especifica arriba. Así que modificamos la máscara de modo que todos los píxeles 0 y 2 se pongan a 0 (es decir, fondo) y todos los píxeles 1 y 3 se pongan a 1 (es decir, píxeles de primer plano). Ahora nuestra máscara final está preparada. Simplemente lo multiplicamos con la imagen de entrada para obtener la imagen segmentada.

```
import numpy as np

import cv2
from matplotlib import pyplot as plt

img = cv2.imread('messi5.jpg')
mask = np.zeros(img.shape[:2],np.uint8)

bgdModel = np.zeros((1,65),np.float64)
fgdModel = np.zeros((1,65),np.float64)

rect = (50,50,450,290)
cv2.grabCut(img,mask,rect,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_RECT)

mask2 = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask2[:,:,np.newaxis]

plt.imshow(img),plt.colorbar(),plt.show()
```

Veamos el resultado a continuación:

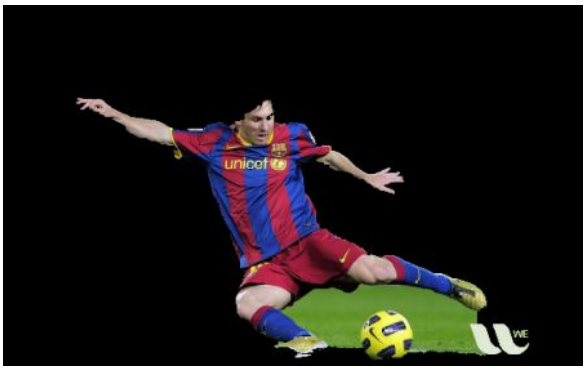


Fig.124: Resultado de aplicación del Algoritmo GrabCut.

Vaya!, el cabello de Messi ha desaparecido. ¿A quién le gusta Messi sin su cabello? Necesitamos retornarlo de vuelta. Entonces haremos un retoque fino con 1 píxel (primer plano seguro). Al mismo tiempo, parte de la tierra ha llegado a la imagen que no queremos, y también algún logotipo. Necesitamos suprimirlos. Ahí damos un retoque de 0 píxeles (fondo seguro). Por lo tanto, modificamos nuestra máscara resultante en el caso anterior tal como se propuso ahora.

Lo que realmente se hizo fue abrír la imagen de entrada en el Paint y agregué otra capa a la imagen. Usando la herramienta de pincel del Paint, marqué el primer plano perdido (cabello, zapatos, pelota, etc.) con fondo blanco y no deseado (como logotipo, suelo, etc.) con negro en esta nueva capa. Luego llené el fondo restante con gris. Después, cargué esa imagen de máscara en Open CV, y edité la imagen de máscara original que obtuvimos con los valores correspondientes en la imagen de máscara recién añadida. Verifica el código a continuación:

```
 # newmask es la máscara etiquetada manualmente

newmask = cv2.imread('newmask.png',0)

# donde sea que esté marcado en blanco (primer plano seguro), cambiar mask=1
# donde sea que esté marcado en negro (fondo seguro), cambiar mask=0
mask[newmask == 0] = 0
mask[newmask == 255] = 1

mask,                    bgdModel,                    fgdModel                    =
cv2.grabCut(img,mask,None,bgdModel,fgdModel,5,cv2.GC_INIT_WITH_MASK)

mask = np.where((mask==2)|(mask==0),0,1).astype('uint8')
img = img*mask[:,:,np.newaxis]
plt.imshow(img),plt.colorbar(),plt.show()
```

Veamos el resultado a continuación:

Fig.125: Resultado final de aplicación del Algoritmo GrabCut.

Aquí, en lugar de inicializar en modo *rect*, podemos pasar directamente al modo de máscara. Simplemente marcamos el área del rectángulo en la imagen de la máscara con 2-pixel o 3-pixel (fondo probable / primer plano). Luego marcamos nuestro *sure_foreground* con 1 píxel como lo hicimos en el segundo ejemplo. A continuación, aplicar directamente la función *GrabCut* con el modo de máscara.

7.16.- Detección y descripción de características

7.16.1.- Entendiendo las características

La única pregunta básica… ¿Cuáles son estas características? (La respuesta debe ser comprensible para una computadora también.)

Bueno, es difícil decir cómo los humanos encontramos estas características. Pero si miramos profundamente en algunas imágenes y buscamos diferentes patrones, encontraremos algo interesante o algo repetido.

7.16.2.- Entender las características

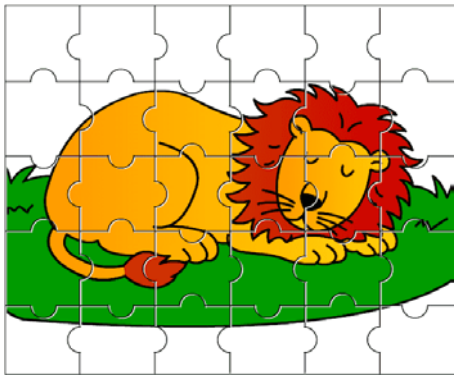


Fig.126: Imagen de referencia para el tratamiento de características.

La imagen es muy simple. En la parte superior de la imagen, se dan seis pequeños parches de imagen. La pregunta es encontrar la ubicación exacta de estos parches en la imagen original. ¿Cuántos resultados correctos podemos encontrar?

7.16.3.- Características

Tratando de responder a la pregunta:"¿Cuáles son estas características?". Pero la siguiente pregunta surge. ¿Cómo las encontramos? ¿O cómo encontramos las esquinas? Eso también respondimos de una manera intuitiva, es decir, buscamos las regiones en imágenes que tienen máxima variación cuando se mueven (por una pequeña cantidad) en todas las regiones a su alrededor. Esto se proyectaría en lenguaje informático en los próximos capítulos. Así que encontrar estas funciones de imagen se llama **Detección de características**.

Así que encontramos los rasgos en la imagen (Supongamos que lo hicimos). Una vez que lo encontramos, deberíamos encontrar lo mismo en las otras imágenes. ¿Qué hacemos?. Tomamos una región en torno a la característica. Básicamente, está describiendo la característica. De manera similar, la computadora también debe describir la región alrededor de la característica para que podamos encontrarla en otras imágenes. La llamada descripción se llama Descripción de la característica. Una vez que tenemos las características y su descripción, podemos encontrar las mismas características en todas las imágenes y alinearlas, ordenarlas o hacer lo que quieras.

Por lo tanto, en este módulo, buscamos diferentes algoritmos en Open CV para encontrar características, describirlas, igualarlas, etc.

7.16.4.- Detección de esquinas Harris

La detección de esquinas juega un papel importante en la visión artificial. En la anterior sección, vimos que las esquinas son regiones de la imagen con grandes variaciones de intensidad en todas las direcciones. Un primer intento de encontrar estas esquinas fue hecho por Chris Harris y Mike Stephens en su documento *A Combined Corner and Edge Detector en 1988*, motivo de llamarse **Harris Corner Detector**. Llevó esta simple idea a una forma matemática. Básicamente encuentra la diferencia de intensidad para un desplazamiento de (u, v) en todas las direcciones. Esto se expresa a continuación:

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} \underbrace{[I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

Por lo tanto, el resultado de **Harris Corner Detection** es una imagen en escala de grises. Lo haremos con una imagen sencilla.

7.16.4.1.- Detector de esquinas Harris en Open CV

OpenCV tiene la función **cv2.cornerHarris ()** para este propósito. Sus argumentos son:

- **img** – Imagen de entrada, debe ser en escala de grises y tipo float32.
- **blockSize** – Es el tamaño del vecindario considerado para la detección de esquinas.
- **ksize** – Parámetro de apertura del derivado Sobel utilizado.
- **k** – Parámetro libre del detector Harris en la ecuación.

Veamos el ejemplo a continuación:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('tablero.jpg',0)
img = np.float32(img)
dst = cv2.cornerHarris(img,2,3,0.04)

dst = cv2.dilate(dst,None)

plt.subplot(2,1,1), plt.imshow(dst )
plt.title('Harris Corner Detection'), plt.xticks([]), plt.yticks([])
plt.subplot(2,1,2),plt.imshow(img,cmap = 'gray')
```

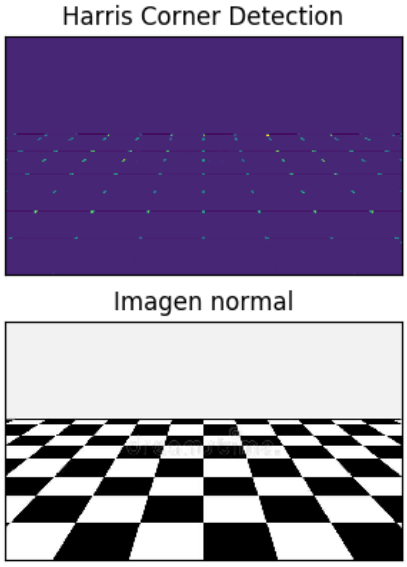


Fig.127: Ejemplo de aplicación del algoritmo de Harris.

7.16.4.2.- Esquina con precisión de subpíxeles

A veces, es posible que necesitemos encontrar las esquinas con la máxima precisión. Open CV dispone de la función **cv2.cornerSubPix ()** que refina aún más las esquinas detectadas con precisión de subpíxeles. A continuación, un ejemplo. Como siempre, tenemos que encontrar las esquinas de Harris primero. A continuación pasamos los centroides de estas esquinas (Puede haber un manojo de pixeles en una esquina, tomamos su centroide) para refinarlos. Las esquinas Harris están marcadas en píxeles rojos y las esquinas refinadas en píxeles verdes. Para esta función, tenemos que definir los criterios para detener la iteración. Lo paramos después de que se haya alcanzado un número específico de iteración o una cierta precisión, lo que ocurra primero. También tenemos que definir el tamaño del barrido en el que se buscarían las esquinas.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

filename = 'tablero.jpg'
img = cv2.imread(filename)
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# encuentra Harris corners
gray = np.float32(gray)
dst = cv2.cornerHarris(gray,2,3,0.04)
dst = cv2.dilate(dst,None)
ret, dst = cv2.threshold(dst,0.01*dst.max(),255,0)
dst = np.uint8(dst)

# encuentra centroides
ret, labels, stats, centroids = cv2.connectedComponentsWithStats(dst)

criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 0.1)
corners = cv2.cornerSubPix(gray,np.float32(centroids),(5,5),(-1,-1),criteria)

res = np.hstack((centroids,corners))
res = np.int0(res)
img[res[:,1],res[:,0]]=[0,0,255]
img[res[:,3],res[:,2]] = [0,255,0]

cv2.imwrite('subpixelharris.png',img)
```

Fig.128: Ejemplo de esquina con precisión de subpixels.

7.16.5.- Detector de Esquina Shi-Tomasi

Tras el detector de esquinas Harris. Más tarde en 1994, J. Shi y C. Tomasi hicieron una pequeña modificación con **Good Features to Track** que muestra mejores resultados en comparación con Harris Corner Detector. La función de puntuación en Harris Corner Detector fue dada por:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

En vez de esto, Shi-Tomasi propuso:

$$R = min(\lambda_1, \lambda_2)$$

Código detector de Esquina Shi-Tomasi:

Open CV dispone de una función, **cv2.goodFeaturesToTrack ()**. Encuentra N esquinas más fuertes en la imagen por el método Shi-Tomasi (o Detección de esquinas Harris, si lo especifica). Como siempre, la imagen debe ser en escala de grises. A continuación, especificaremos el número de esquinas que deseamos encontrar. A continuación, especificaremos el nivel de calidad, que es un valor entre 0-1, que indica la calidad mínima de esquina por debajo de la cual se rechaza a todo el mundo. A continuación, proporcionamos la distancia euclidiana mínima entre las esquinas detectadas.

Con todas estas informaciones, la función encuentra esquinas en la imagen. Se rechazan todas las esquinas por debajo del nivel de calidad. A continuación, clasifica las esquinas restantes según la calidad en orden descendente. Entonces la función toma la primera esquina más fuerte, tira todas las esquinas cercanas en el rango de distancia mínima y devuelve N esquinas más fuertes.

En el siguiente ejemplo, se trata de encontrar las 6 mejores esquinas:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('tablero.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray,6,0.01,10)
corners = np.int0(corners)

for i in corners:
  x,y = i.ravel()
  cv2.circle(img,(x,y),3,255,-1)

plt.imshow(img),plt.show()
```

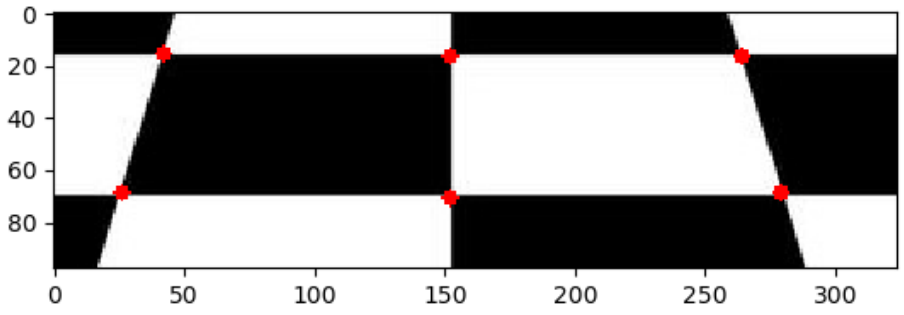


Fig.129: Ejemplo de detección de esquinas con método Shi-Tomasi.

7.16.6.- Algoritmo FAST para la detección de esquinas

Hemos podido ver varios detectores de características y muchos de ellos son realmente buenos. Pero viéndolos desde el punto de vista de la aplicación en tiempo real, no son lo suficientemente rápidos. Un mejor ejemplo sería SLAM (Simultaneous Localization and Mapping) robot móvil que tiene recursos computacionales limitados.

Como solución a esto, el algoritmo FAST (Features from Accelerated Segment Test) propuesto por Edward Rosten y Tom Drummond en su trabajo "Machine learning for speed corner detection" en 2006 (posteriormente revisado en 2010). Veamos un resumen básico del algoritmo.

7.16.6.1.- Detección de características con FAST

1. Seleccionar un píxel *p* en la imagen que debe ser identificado como un punto de interés o no. Que su intensidad sea *I_p*.
2. Seleccionar el valor umbral adecuado *t*.
3. Considerar un círculo de 16 píxeles alrededor del píxel bajo prueba.
4. Ahora el pixel *p* es una esquina si existe un conjunto de *n* píxeles contiguos en el círculo (de 16 píxeles) que son todos más brillantes que *I_p + t*, o todos más oscuros que *I_p – t*. *n* fue elegido para sea 12.
5. Se propuso una prueba de alta velocidad para excluir a un gran número de no esquinas. Esta prueba examina sólo los cuatro píxeles a 1,9,5 y 13 (Primeros 1 y 9 son probados si son demasiado brillantes u oscuros. Si es así, verifica 5 y 13). Si *p* es una esquina, entonces al menos tres de ellas deben ser más brillantes que *I_p + t* o más oscuras que *I_p – t*. Si ninguno de estos dos casos es el caso, entonces *p* no puede ser una esquina. El criterio de prueba de segmento completo se puede aplicar a los candidatos aprobados examinando todos los píxeles del círculo. Este detector en sí mismo muestra un alto rendimiento, pero hay varias debilidades:
    - No rechaza tantos candidatos para n < 12.
    - La elección de píxeles no es óptima porque su eficiencia depende del orden de las preguntas y la distribución de las apariencias de las esquinas.
    - Los resultados de las pruebas de alta velocidad se desechan.
    - Se detectan múltiples características adyacentes entre sí.

7.16.6.2.- Detector de características FAST en Open CV

Se denomina como cualquier otro detector de características en Open CV. Si lo deseamos, podemos especificar el umbral, si se debe aplicar o no la supresión no máxima, el vecindario a utilizar, etc.

Para el vecindario, se definen tres banderas,

cv2.FAST_FEATURE_DETECTOR_TYPE_5_8,
cv2.FAST_FEATURE_DETECTOR_TYPE_7_12 y
cv2.FAST_FEATURE_DETECTOR_TYPE_9_16.

A continuación, un código simple sobre cómo detectar y dibujar los puntos de la función FAST.

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
import itertools
```

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('tablero.png',0)

# Iniciar objeto FAST con valores propuestos
fast = cv2.FastFeatureDetector_create()

# encontrar y dibujar los puntos clave
kp = fast.detect(img,None)
img2 = cv2.drawKeypoints(img, kp,img)

cv2.imwrite('fast_tablero.png',img2)

# desactiva nonmaxSuppression
fast.setBool('nonmaxSuppression',0)
kp = fast.detect(img,None)
```

Fig.130: Ejemplo de aplicación del algoritmo FAST.

7.16.7.- ORB (Oriented Fast y Rotativo BRIEF)

Lo más destacado del ORB es que proviene de “Open CV Labs”. Este algoritmo fue criado por Ethan Rublee, Vincent Rabaud, Kurt Konolige y Gary R. Bradski en su trabajo ORB: Una alternativa eficiente a SIFT o SURF en 2011. Es una buena alternativa al SIFT y al SURF respecto a los costes de cálculo, igualando el rendimiento y principalmente las patentes. Sí, SIFT y SURF están patentados y debemos pagar por su uso. ¡ORB no lo es!!

ORB es una fusión del detector de punto clave FAST y el descriptor BRIEF con muchas modificaciones para mejorar el rendimiento. Primero usa FAST para encontrar los puntos clave, luego aplica la medida de la esquina de Harris para encontrar los mejores N puntos entre ellos. También utiliza la pirámide para producir rasgos multifuncionales. Pero un problema es que FAST no calcula la orientación. ¿Y qué pasa con la invariancia de rotación? A los autores se les ocurrió la siguiente modificación.

Calcula la intensidad ponderada del centroide del parche con la esquina localizada en el centro. La dirección del vector desde este punto de esquina al centroide da la orientación. Para mejorar la invariancia de rotación, los momentos se calculan con x y y que deben estar en una región circular de radio r, donde r es el tamaño del parche.

Ahora para descriptores, ORB usa descriptores BRIEF. Pero hemos visto que BRIEF funciona mal con la rotación. Por lo tanto, lo que hace ORB es “dirigir” según la orientación de los puntos clave. Para cualquier conjunto de características de n pruebas binarias en la ubicación (x_i, y_i), defina una matriz 2 \ multiplicado por n, S que contenga las coordenadas de estos píxeles. A continuación, usando la orientación del patch, \theta, se encuentra su matriz de rotación y gira la S para obtener la versión dirigida (rotada) S_\theta.

ORB discretiza el ángulo a incrementos de 2 \Npi /30 (12 grados), y construye una tabla de búsqueda de patrones BRIEF precalculados. Mientras la orientación de los puntos clave \theta sea consistente en todas las vistas, se utilizará el conjunto correcto de puntos S_\theta para calcular su descriptor.

BRIEF tiene una propiedad importante de que cada característica bit tiene una gran varianza y una media cercana a 0,5. Pero una vez que se orienta en la dirección de los puntos clave, pierde esta propiedad y se distribuye mejor. La alta varianza hace que una característica sea más discriminatoria, ya que responde de forma diferente a las variables de entrada. Otra propiedad deseable es tener las pruebas no correlacionadas, ya que cada prueba contribuirá al resultado. Para resolver todo esto, ORB realiza una búsqueda codiciosa entre todas las pruebas binarias posibles para encontrar las que tienen alta varianza y significa cerca de 0.5, además de no estar correlacionadas. El resultado se llama **rBRIEF**.

Para la correspondencia de descriptores, se utiliza LSH multi-sonda que mejora el LSH tradicional. El artículo dice que ORB es mucho más rápido que SURF y que SIFT y ORB descriptor funciona mejor que SURF. ORB es una buena opción en dispositivos de baja potencia para costuras panorámicas, etc.

7.16.7.1.- ORB en Open CV

Como venimos haciendo, tenemos que crear un objeto ORB con la función, **cv2.ORB ()** o utilizando el interfaz común feature2d. Tiene una serie de parámetros opcionales. Los más útiles son nFeatures que indican el número máximo de características a retener (por defecto 500), scoreType que indica si la puntuación de Harris o FAST para clasificar las características (por defecto, la puntuación de Harris) etc. Otro parámetro, WTA_K decide el número de puntos que producen cada elemento del descriptor BRIEF orientado. Por defecto son dos, es decir, selecciona dos puntos a la vez. En ese caso, para la comparación se utiliza la distancia NORM_HAMMING. Si WTA_K es 3 o 4, lo que toma 3 o 4 puntos para producir el descriptor BRIEF, entonces la distancia de coincidencia es definida por NORM_HAMMING2.

Un código que muestra el uso de ORB.

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('ojo.jpg',0)

# Initiate STAR detector
orb = cv2.ORB_create()

# find the keypoints with ORB
kp = orb.detect(img,None)

# compute the descriptors with ORB
kp, des = orb.compute(img, kp)

# draw only keypoints location,not size and orientation
img2 = cv2.drawKeypoints(img,kp,img, flags=0)
```

```
plt.imshow(img2),plt.show()
```

Fig.131: Ejemplo de aplicación del algoritmo ORB.

7.16.8.- Feature Matching / Comparación de funciones

7.16.8.1.- Conceptos básicos de Brute-Force Matcher

Brute-Force matcher es sencillo. Toma el descriptor de una característica en el primer set y se empareja con todas las otras características en el segundo set usando un cálculo de distancia. Y el más cercano se retorna. Se trata de hacer coincidir las características de una imagen con otras.

Para BF matcher, primero tenemos que crear el objeto BFMatcher usando **cv2.BFMatcher ()**. Se necesitan dos parametros opcionales. El primero es normType. Especifica la medición de distancia a utilizar. Por defecto, es cv2.NORM_L2. Es bueno para SIFT, SURF etc (cv2. NORM_L1 también está). Para descriptores binarios basados en cadenas de texto como ORB, BRIEF, BRISK etc. Se debe utilizar cv2.NORM_HAMMING, que utiliza la distancia de

Hamming como medida. Si ORB está usando VTA_K == 3 o 4, se debe usar cv2.NORM_HAMMING2.

El segundo parámetro es la variable booleana, crossCheck que es false por defecto. Si es cierto, Matcher retorna sólo aquellos partidos con valor (i, j) de tal manera que el descriptor i-ésimo del set A tenga el descriptor j-ésimo del set B como mejor partido y viceversa. Es decir, las dos características de ambos conjuntos deben coincidir entre sí. Proporciona un resultado consistente, y es una buena alternativa a la prueba de relación propuesta por D. Lowe en papel SIFT.

Una vez creado, tenemos dos métodos importantes, que son *BFMatcher. match ()* y *BFMatcher. knnMatcher ()*. El primero retorna el mejor partido. El segundo método devuelve k las mejores coincidencias donde k lo especifica el usuario. Puede ser útil cuando necesitemos hacer un trabajo adicional sobre eso.

Como usamos cv2.drawKeypoints () para dibujar puntos clave, **cv2.drawMatches ()** nos ayuda a dibujar los partidos. Apila dos imágenes horizontalmente y dibuja líneas de la primera imagen a la segunda, mostrando las mejores coincidencias. Hay también **cv2.drawMatchesKnn** que sortea todos los mejores partidos de k. Si k=2, dibujará dos líneas de combate para cada punto clave. Así que tenemos que pasar una máscara si queremos dibujarla selectivamente.

A continuación un ejemplo para cada uno de SURF y ORB (ambos usan diferentes medidas de distancia).

7.16.8.2.- Combinación de fuerza bruta con descriptores ORB

Aquí, veremos un ejemplo simple de cómo hacer coincidir las características entre dos imágenes. En este caso, tenemos una consulta Imagen y una imagen de un ojo. Intentaremos encontrar la consulta Imagen en trainImage usando la coincidencia de características.

Estamos utilizando descriptores SIFT para combinar las características. Así que vamos a empezar con la carga de imágenes, encontrar descriptores, etc.

A continuación creamos un objeto BFMatcher con medición de distancias cv2.NORM_HAMMING (ya que estamos usando ORB) y se activa el crosscheck para obtener mejores resultados. A continuación utilizamos el método Matcher.match () para obtener las mejores coincidencias en dos imágenes. Los clasificamos en orden ascendente de sus distancias para que los mejores partidos (con baja distancia) lleguen al frente. Luego sorteamos sólo las primeras 30 coincidencias (sólo por razones de visibilidad, podemos aumentarlo como queramos).

7.16.8.3.- Combinación de fuerza bruta con descriptores ORB código python:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img1 = cv2.imread('ojo.jpg',0) # queryImage
img2 = cv2.imread('ojo4.jpg',0) # trainImage

#Iniciar detector SIFT
orb = cv2.ORB_create()

# Encuentra los puntos clave y descriptores con SIFT
kp1, des1 = orb.detectAndCompute(img1,None)
kp2, des2 = orb.detectAndCompute(img2,None)
```

```
# crea BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors.
matches = bf.match(des1,des2)

# Clasifícalos en el orden de su distancia.
matches = sorted(matches, key = lambda x:x.distance)

# Dibuja las primeras 30 coincidencias.
img3 = cv2.drawMatches(img1,kp1,img2,kp2,matches[:30],None, flags=2)

plt.imshow(img3),plt.show()
```

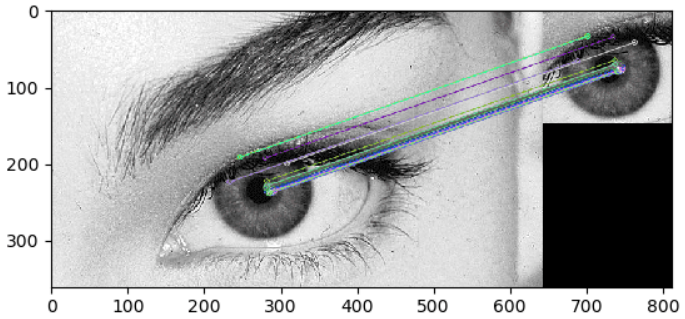


Fig.132: Imagen de ejemplo del algoritmo Feature Matching.

¿Qué es este objeto Matcher?

El resultado de matches=bf.match (des1, des2) es una lista de objetos DMatch. Este objeto DMatch tiene los atributos siguientes:

- DMatch.distance – Distancia entre descriptores. Cuanto más bajo, mejor.
- DMatch.trainIdx – Índice del descriptor en descriptores de trenes
- DMatch.queryIdx – Índice del descriptor en descriptores de consulta
- DMatch.imgIdx – Índice de la imagen del tren.

7.17.- Vídeo análisis

7.17.1.- Algoritmos Meanshift y Camshift.

La intuición que sigue al cambio de medias es sencilla. Consideremos que tenemos un conjunto de puntos. (Puede ser una distribución de píxeles como la retroproyección del histograma). Se nos da una pequeña ventana (puede ser un círculo) y tenemos que mover esa ventana al área de máxima densidad de píxeles (o número máximo de puntos). Se ilustra recorte de imagen simple:

Fig.133: Ejemplo de Meanshift

El vídeo muestra un cuadrado azul con el nombre. Su centro original se llama “C1_o”. Pero si encontramos el centroide de los puntos dentro de esa ventana, obtendremos el punto “C1_r” que es el verdadero centroide de la ventana. Seguramente no coinciden. Así que movamos nuestra ventana para que el círculo de la nueva ventana coincida con el centroide anterior. Volvemos a encontrar el nuevo centroide. Lo más probable es que no coincida. Así que lo movemos otra vez, y continuamos las iteraciones de tal manera que el centro de la ventana y su centroide caiga en la misma ubicación (o con un pequeño error deseado). Así que finalmente lo que se obtiene es una ventana con la máxima distribución de píxeles.

Así que normalmente pasamos la imagen del histograma retroproyectada y la ubicación inicial del objetivo. Cuando el objeto se mueve, obviamente el movimiento se refleja en la imagen retroproyectada del histograma. Como resultado, el algoritmo meanshift mueve la ventana a la nueva ubicación con la máxima densidad.

```
import numpy as np
import cv2

cap = cv2.VideoCapture('mundi2.mp4')

# toma el primer fotograma del video
ret,frame = cap.read()

# Configuración de la ubicación inicial de la ventana
r,h,c,w = 120,40,190,40 # simply hardcoded the values
track_window = (c,r,w,h)

# establece el ROI para rastrear
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask        =        cv2.inRange(hsv_roi,        np.array((0.,        60.,32.)),
np.array((180.,255.,255.)))
roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)

# Configure el criterio de terminación, ya sea 10 iteración o mover por lo
menos 1 pt.
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 50, 1 )

while(1):
  ret ,frame = cap.read()

  if ret == True:
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

```
    dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

    # Aplica meanshift para conseguir la nueva ubicación.
    ret, track_window = cv2.meanShift(dst, track_window, term_crit)

    # Dibújalo en la imagen
    x,y,w,h = track_window
    img2 = cv2.rectangle(frame, (x,y), (x+w,y+h), 255,2)
    cv2.imshow('img2',img2)

    k = cv2.waitKey(60) & 0xff
    if k == 27:
      break
    else:
      cv2.imwrite(chr(k)+".jpg",img2)

  else:
    break

cv2.destroyAllWindows()
cap.release()
```

7.17.2.- Camshift

¿Observamos la sección anterior el resultado? Hay un problema. La ventana siempre tiene el mismo tamaño cuando el jugador está más lejos o está muy cerca de la cámara. Eso no es bueno. Tenemos que adaptar el tamaño de la ventana con el tamaño y rotación del objetivo. Una vez más, la solución viene de Open CV y se llama CAMshift (Continuously Adaptive Meanshift) publicado por Gary Bradsky en su trabajo “Computer Vision Face Tracking for Use in a Perceptual User Interface” en 1988.

Aplica el cambio de significado primero. Una vez que mediashift converge, actualiza el tamaño de la ventana como, s = 2 x \(M_00 / 256)^1/2. También calcula la orientación de la elipse que mejor se adapte a ella. De nuevo se aplica la mediashift con la nueva ventana de búsqueda a escala y la ubicación de la ventana anterior. El proceso continúa hasta que se alcanza la precisión requerida.

7.17.2.1.- Código Camshift en Open CV

Es similar a meanshift, pero retorna un rectángulo girado (éste es nuestro resultado) y parámetros de caja (utilizados para pasar como ventana de búsqueda en la iteración siguiente). Veamos el código:

```
import numpy as np
import cv2

cap = cv2.VideoCapture('bola.flv')

# toma el 1º frame del video
ret,frame = cap.read()

# ajusta el tamño de la ventana
r,h,c,w = 250,90,400,125
track_window = (c,r,w,h)

# ajusta el ROI para el rastreo
roi = frame[r:r+h, c:c+w]
hsv_roi = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask        =        cv2.inRange(hsv_roi,        np.array((0.,        60.,32.)),
np.array((180.,255.,255.)))
roi_hist = cv2.calcHist([hsv_roi],[0],mask,[180],[0,180])
cv2.normalize(roi_hist,roi_hist,0,255,cv2.NORM_MINMAX)
```

```
# Configure el criterio de terminación, ya sea 10 iteración o mover por lo
menos 1 pt.
term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1 )

while(1):
  ret ,frame = cap.read()

  if ret == True:
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    dst = cv2.calcBackProject([hsv],[0],roi_hist,[0,180],1)

    #aplica meanshift para conseguir la nueva ubicacion
    ret, track_window = cv2.CamShift(dst, track_window, term_crit)

    # dibuja en la imagen
    pts = cv2.boxPoints(ret)
    pts = np.int0(pts)
    img2 = cv2.polylines(frame,[pts],True, 255,2)
    cv2.imshow('img2',img2)

    k = cv2.waitKey(60) & 0xff
    if k == 27:
      break
    else:
      cv2.imwrite(chr(k)+".jpg",img2)

  else:
    break

cv2.destroyAllWindows()
cap.release()
```

A continuación un ejemplo donde se puede ver la modificación de la ventana y a esto, el tamaño de la misma no es constante. El rastreo de objetos en video tiene muchas aplicaciones.

:

Fig.134: Ejemplo de CamShift.

7.18.- Flujo óptico

El flujo óptico es el patrón de movimiento aparente de los objetos de la imagen entre dos fotogramas consecutivos causado por el movimiento del objeto o la cámara. Es un campo vectorial 2D donde cada vector es un vector de desplazamiento que muestra el movimiento de los puntos del primer cuadro al segundo. Considerar la imagen siguiente.

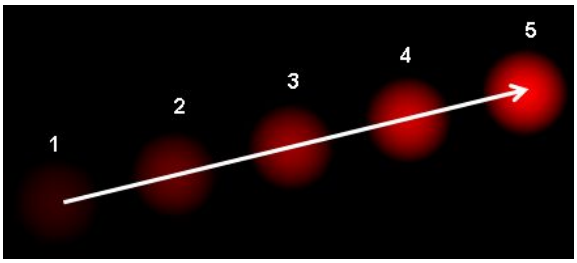


Fig.135: Ejemplo de referencia de flujo óptico.

Podemos ver una bola moviéndose en 5 cuadros consecutivos. La flecha muestra su vector de desplazamiento. El flujo óptico tiene muchas aplicaciones en áreas como:

- Estructura desde el movimiento
- Compresión de vídeo
- Estabilización de vídeo…

El flujo óptico funciona sobre varios supuestos:

1. Las intensidades de píxel de un objeto no cambian entre fotogramas consecutivos.
2. Los píxeles vecinos tienen movimiento similar.

Considere un píxel *I (x, y, t)* en el primer cuadro (Consultar la nueva dimensión, el tiempo, se agrega aquí. Antes sólo operábamos con imágenes, así que no necesitábamos tiempo). Se mueve por distancia *(dx, dy)* en el siguiente cuadro tomado después del tiempo *dt*. Así que como esos píxeles son los mismos y la intensidad no cambia, podemos decir,

I (x, y, t) = I (x+dx, y+dy, t+dt)

A continuación, realizar la aproximación de la serie Taylor del lado derecho, eliminar los términos comunes y dividir por dt para obtener la siguiente ecuación:

f_x u + f_y v + f_t = 0

donde:

$$f_x = \frac{\partial f}{\partial x} \; ; \; f_y = \frac{\partial f}{\partial x}$$
$$u = \frac{dx}{dt} \; ; \; v = \frac{dy}{dt}$$

La ecuación anterior se llama ecuación de flujo óptico. En ella podemos encontrar *f_x* y *f_y*, son gradientes de imagen. Del mismo modo *f_t* es el gradiente a lo largo del tiempo. Pero *(u, v)* es desconocido. No podemos resolver esta ecuación con dos variables desconocidas. Así que se proporcionan varios métodos para resolver este problema y uno de ellos es Lucas-Kanade.

7.18.1.- Método Lucas-Kanade

Hemos visto una suposición anterior de que todos los píxeles vecinos tendrán un movimiento similar. El método Lucas-Kanade toma un parche de 3×3 alrededor del punto. Así que todos los 9 puntos tienen la misma moción. Podemos encontrar (*f_x, f_y, f_t*) para estos 9 puntos. Así que ahora nuestro problema se convierte en resolver 9 ecuaciones con dos variables desconocidas

que están sobredeterminadas. Se obtiene una mejor solución con el método de ajuste cuadrado. Abajo está la solución final que son dos ecuaciones.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i {f_{x_i}}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i {f_{y_i}}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix}$$

(Comprobar la similitud de la matriz inversa con el detector de esquinas Harris. Denota que las curvas son mejores puntos a ser rastreados).

Así, desde el punto de vista del usuario, la idea es sencilla, damos algunos puntos a rastrear, recibimos los vectores de flujo óptico de esos puntos. Pero de nuevo hay algunos problemas. Hasta ahora, nos ocupábamos de pequeños movimientos. Así que falla cuando hay movimiento grande. Así que retornamos a la teoría de las pirámides. Cuando subimos en la pirámide, los movimientos pequeños son removidos y los movimientos grandes se convierten en movimientos pequeños. Aplicando Lucas-Kanade, tenemos flujo óptico junto con la escala.

7.18.8.1.- Ejemplo Flujo óptico Lucas-Kanade en Open CV

Open CV proporciona todo lo anterior en una sola función, **cv2.calcOpticalFlowPyrLK()**. Aquí creamos una aplicación sencilla que rastrea algunos puntos en un vídeo. Para decidir los puntos, utilizamos **cv2.goodFeaturesToTrack()**. Tomamos el primer fotograma, detectamos algunos puntos de esquina Shi-Tomasi en él, luego hacemos un seguimiento iterativo de esos puntos usando el flujo óptico Lucas-Kanade. Para la función **cv2.calcOpticalFlowPyrLK ()** pasamos el fotograma anterior, los puntos anteriores y el fotograma siguiente. Devuelve los siguientes puntos junto con algunos números de estado que tienen un valor de 1 si se encuentra el siguiente punto, o bien cero. Pasamos de manera iterativa estos próximos puntos como puntos anteriores en el próximo paso. Veamos el código siguiente:

```
import numpy as np
import cv2

cap = cv2.VideoCapture('toki.mp4')
# parametros para detección de esquinas ShiTomasi
feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

# Parámetros para el flujo óptico de Lucas Kanade
lk_params = dict( winSize = (15,15),
                  maxLevel = 2,
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
10, 0.03))

# Crea algunos colores aleatorios
color = np.random.randint(0,255,(100,3))
# Toma el primer cuadro y encuentra esquinas en él
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

# Crear una máscara de imagen para dibujar
mask = np.zeros_like(old_frame)

while(1):
  ret,frame = cap.read()
  frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

  # calcula optical flow
```

```
  p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)

  # Select good points
  good_new = p1[st==1]
  good_old = p0[st==1]

  # dibuja las lineas
  for i,(new,old) in enumerate(zip(good_new,good_old)):
    a,b = new.ravel()
    c,d = old.ravel()
    mask = cv2.line(mask, (a,b),(c,d), color[i].tolist(), 2)
    frame = cv2.circle(frame,(a,b),5,color[i].tolist(),-1)
  img = cv2.add(frame,mask)

  cv2.imshow('frame',img)
  k = cv2.waitKey(30) & 0xff
  if k == 27:
    break

  # Ahora actualiza el marco anterior y los puntos anteriores
  old_gray = frame_gray.copy()
  p0 = good_new.reshape(-1,1,2)

cv2.destroyAllWindows()
cap.release()
```

(Este código no verifica lo correctos que son los siguientes puntos clave. Por lo tanto, incluso si cualquier punto de característica desaparece en la imagen, existe la posibilidad de que el flujo óptico encuentre el siguiente punto que pueda mirarlo de cerca. Así, para un seguimiento robusto, los puntos de esquina deben se deben detectar en intervalos particulares. Las muestras de Open CV vienen con una muestra que encuentra los puntos de característica en cada 5 fotogramas. También ejecuta una comprobación hacia atrás de los puntos de flujo ópticos, que sólo tienen que seleccionar los buenos).

Veamos los resultados que tenemos:

Fig.136: Ejemplo de flujo óptico.

7.18.8.2.- Flujo óptico denso Gunner Farneback en Open CV

El método Lucas-Kanade calcula el flujo óptico para un conjunto de características dispersas (en nuestro ejemplo, las esquinas detectadas mediante el algoritmo Shi-Tomasi). Open CV proporciona otro algoritmo para encontrar el flujo óptico denso. Calcula el flujo óptico para todos los puntos del cuadro. Se basa en el algoritmo de Gunner Farneback, que se explica en "Two-Frame Motion Estimmation Based on Polynomial Expansion" de Gunner Farneback en 2003.

A continuación se muestra cómo encontrar el flujo óptico denso utilizando el algoritmo anterior. Obtenemos una matriz de 2 canales con vectores de flujo óptico, (u, v). Encontramos su magnitud y dirección. Coloreamos el resultado para una mejor visualización. La dirección corresponde al valor Hue de la imagen. La Magnitud corresponde al Plano de Valor. Veamos el código siguiente:

```
import cv2
import numpy as np
cap = cv2.VideoCapture("toki.mp4")

ret, frame1 = cap.read()
prvs = cv2.cvtColor(frame1,cv2.COLOR_BGR2GRAY)
hsv = np.zeros_like(frame1)
hsv[...,1] = 255

while(1):
  ret, frame2 = cap.read()
  next = cv2.cvtColor(frame2,cv2.COLOR_BGR2GRAY)

  flow = cv2.calcOpticalFlowFarneback(prvs,next, None, 0.5, 3, 15, 3, 5, 1.2,
0)

  mag, ang = cv2.cartToPolar(flow[...,0], flow[...,1])
  hsv[...,0] = ang*180/np.pi/2
  hsv[...,2] = cv2.normalize(mag,None,0,255,cv2.NORM_MINMAX)
  rgb = cv2.cvtColor(hsv,cv2.COLOR_HSV2BGR)

  cv2.imshow('frame2',rgb)
  k = cv2.waitKey(30) & 0xff
  if k == 27:
    break
  elif k == ord('s'):
    cv2.imwrite('opticalfb.png',frame2)
    cv2.imwrite('opticalhsv.png',rgb)
  prvs = next

cap.release()
cv2.destroyAllWindows()
```

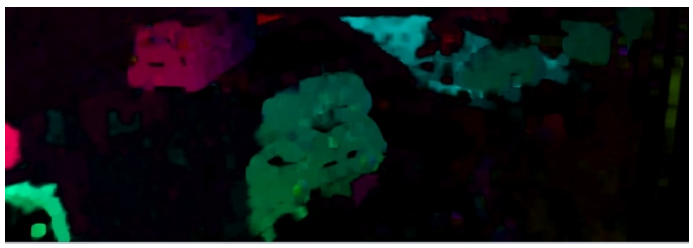


Fig.137: Ejemplo de flujo óptico denso.

7.18.2.- Substracción de fondo de un vídeo

La resta de fondo es uno de los pasos más importantes del preprocesamiento en muchas aplicaciones de la visión artificial. Por ejemplo, considerar los casos como el contador de visitantes donde una cámara estática toma el número de visitantes que entran o salen de la habitación, o una cámara de tráfico que extrae información sobre los vehículos, etc. En todos estos casos, primero hay que extraer a la persona o vehículos solos. Técnicamente, es necesario extraer el primer plano móvil del fondo estático.

Si tenemos una imagen de fondo solo, como la imagen de la habitación sin visitantes, la imagen de la carretera sin vehículos etc, es un trabajo fácil. Sólo resta la nueva imagen del fondo. Solo

nos quedamos con los objetos en primer plano. Pero en la mayoría de los casos, es posible que no tengamos una imagen de este tipo, por lo que necesitamos extraer el fondo de cualquier imagen que tengamos. Se vuelve más complicado cuando hay sombra de los vehículos. Puesto que la sombra también se está moviendo, una simple sustracción marcará eso también como primer plano y complica las cosas.

Para ello se introdujeron varios algoritmos. Open CV ha implementado tres algoritmos fáciles de usar.

Imagen original:

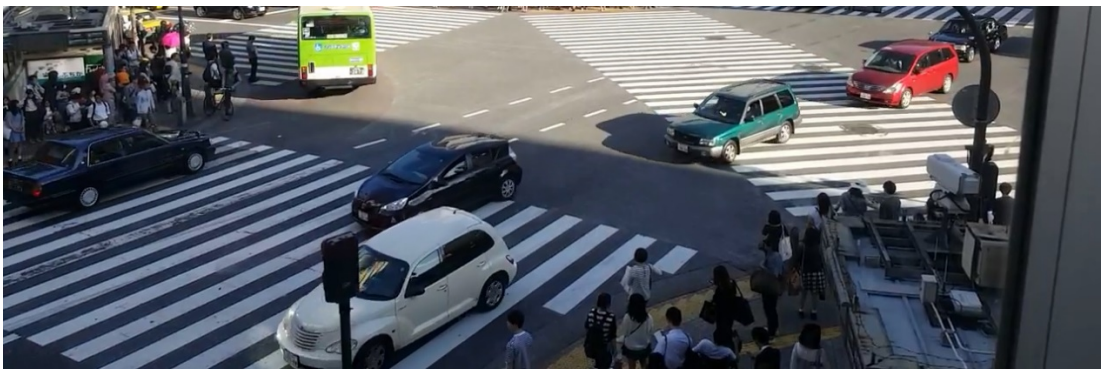


Fig.138: Imagen original de referencia para sustracción de fondo en vídeo.

7.18.2.1.- BackgroundSubtractorMOG2

Es un algoritmo de segmentación de fondo/primer plano basado en la mezcla gaussiana. Se basa en dos documentos de Z. Zivkovic,"Modelo de mezcla gausiana adaptativa mejorada para la sustracción de fondo" en 2004 y "Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Substraction" en 2006. Una característica importante de este algoritmo es que selecciona el número apropiado de distribución gaussiana para cada píxel. Proporciona una mejor adaptabilidad a la variación de escenas debido a cambios de iluminación, etc.

Tenemos que crear un objeto de substractor de fondo. Aquí tenemos la opción de seleccionar, según deseemos, que se detecte sombra o no. Si detectShadows = True (que es así por defecto), detecta y marca sombras, pero disminuye la velocidad. Las sombras se marcarán en color gris.

```
import numpy as np
import cv2

cap = cv2.VideoCapture('toki.mp4')

fgbg = cv2.createBackgroundSubtractorMOG2()

while(1):
  ret, frame = cap.read()

  fgmask = fgbg.apply(frame)

  cv2.imshow('frame',fgmask)
  k = cv2.waitKey(30) &amp;amp; 0xff
  if k == 27:
    break

cap.release()
cv2.destroyAllWindows()
```

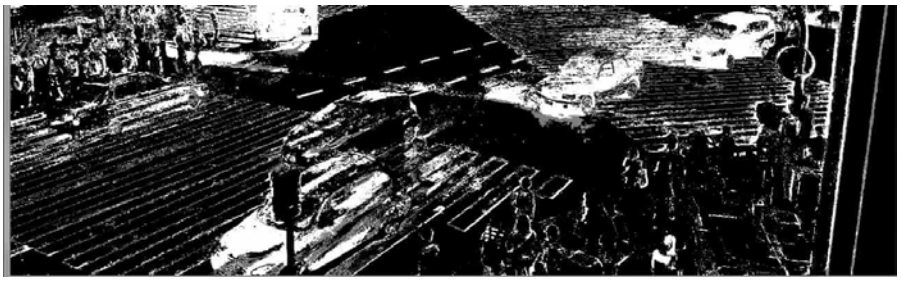


Fig.139: Imagen con fondo sustraído por método MOG2.

7.18.2.2.- BackgroundSubtractorKNN

```
import numpy as np
import cv2

cap = cv2.VideoCapture('toki.mp4')

fgbg = cv2.createBackgroundSubtractorKNN()

while(1):
  ret, frame = cap.read()

  fgmask = fgbg.apply(frame)

  cv2.imshow('frame',fgmask)
  k = cv2.waitKey(30) &amp;amp;amp; 0xff
  if k == 27:
    break

cap.release()
cv2.destroyAllWindows()
```

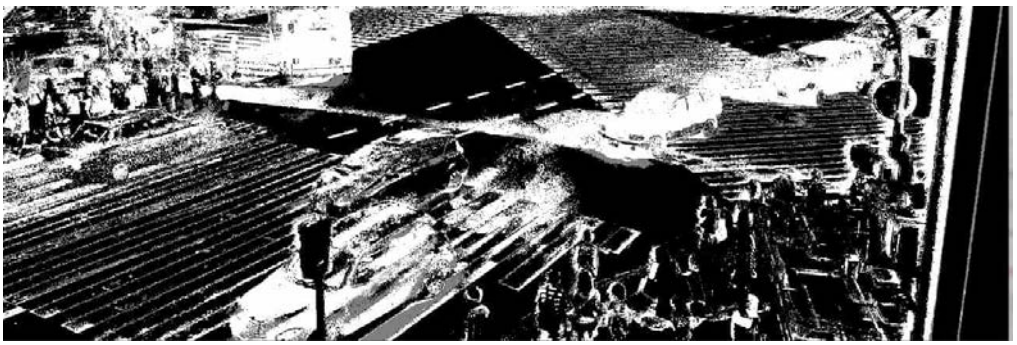


Fig.140: Imagen con fondo sustraído por método KNN.

7.19.- Calibración de la cámara y reconstrucción 3D

7.19.1.- Calibración de la cámara

Las cámaras baratas de hoy en día introducen mucha distorsión en las imágenes. Dos distorsiones principales son la distorsión radial y la distorsión tangencial.

Debido a la distorsión radial, las líneas rectas aparecerán curvadas. Su efecto es mayor a medida que nos alejamos del centro de la imagen. Por ejemplo, la imagen se muestra abajo. Pero podemos ver que el borde no es una línea recta y no coincide. Todas las líneas rectas esperadas están abultadas.

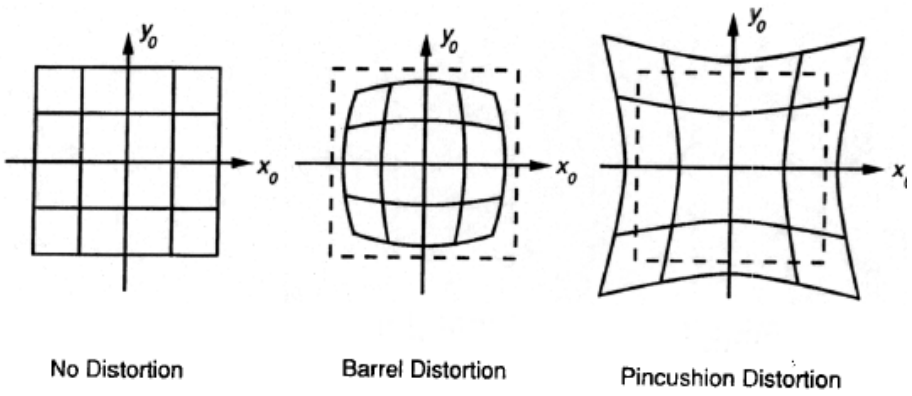


Fig.150: Distorsiones en cámaras: barril y de cojín.

Esta distorsión se resuelve de la siguiente manera:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Del mismo modo, otra distorsión es la distorsión tangencial que ocurre porque la toma de imágenes no está perfectamente alineada paralelamente al plano de imagen. Por lo tanto, algunas áreas en la imagen pueden verse más cercanas de lo esperado. Se resuelve como sigue:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

En forma de resumen, necesitamos encontrar cinco parámetros, conocidos como coeficientes de distorsión dados por:

$$Distortion\ coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Además, necesitamos encontrar más información, como los parámetros intrínsecos y extrínsecos de una cámara. Los parámetros intrínsecos son específicos de una cámara. Incluye información como distancia focal (*f_x, f_y*), centros ópticos *(c_x, c_y*), etc. También se llama matriz de cámara. Depende sólo de la cámara, por lo que una vez calculada, se puede almacenar para fines futuros. Se expresa como una matriz de 3×3:

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Los parámetros extrínsecos corresponden a vectores de rotación y traslación que traducen las coordenadas de un punto 3D a un sistema de coordenadas.

Para las aplicaciones estéreo, estas distorsiones necesitan corregirse primero. Para encontrar todos estos parámetros, lo que tenemos que hacer es proporcionar algunas imágenes de muestra de un patrón bien definido (p. ej., tablero de ajedrez). Encontramos algunos puntos específicos en él (esquinas cuadradas en tablero de ajedrez). Conocemos sus coordenadas en el espacio del mundo real y conocemos sus coordenadas en imagen. Con estos datos, algún problema matemático se resuelve en segundo plano para obtener los coeficientes de distorsión. Esto a

modo de resumen todo el proceso. Para obtener mejores resultados, necesitamos al menos 10 patrones de prueba.

7.19.2.- Código de Calibración de la cámara

Como se acaba de indicar, necesitamos al menos 10 patrones de prueba para la calibración de la cámara. Open CV viene con algunas imágenes del tablero de ajedrez, así que lo utilizaremos. Para entender, consideremos sólo una imagen de un tablero de ajedrez. Los datos de entrada importantes necesarios para la calibración de la cámara son un conjunto de puntos 3D del mundo real y sus puntos de imagen 2D correspondientes. Los puntos de imagen 2D están bien, que podemos encontrar fácilmente en la imagen. (Estos puntos de imagen son lugares donde dos cuadrados negros se tocan entre sí en tableros de ajedrez)

¿Qué hay de los puntos 3D del espacio del mundo real? Esas imágenes se toman de una cámara estática y los tableros de ajedrez se colocan en diferentes lugares y orientaciones. Así que necesitamos conocer los valores (X, Y, Z). Pero por simplicidad, podemos decir que el tablero de ajedrez se ha mantenido estacionario en el plano XY, (así que Z=0 siempre) y la cámara fue movida en consecuencia. Esta consideración nos ayuda a encontrar sólo los valores X, Y. Ahora para los valores X, Y, sencillamente podemos pasar los puntos como (0,0), (1,0), (2,0),… lo que denota la ubicación de los puntos. En este caso, los resultados que obtendremos serán en la escala de tamaño del tablero de ajedrez cuadrado. Pero si conocemos el tamaño cuadrado (digamos 30 mm), y podemos pasar los valores como (0,0), (30,0), (60,0),…, obtenemos los resultados en mm. (En este caso, no sabemos el tamaño del cuadrado ya que no tomamos esas imágenes, por lo que pasamos en términos de tamaño del cuadrado).

Los puntos 3D se llaman **puntos de objeto** y los puntos de imagen 2D se llaman **puntos de imagen**.

7.19.3.- Configuración de Calibración de la cámara

Así que para encontrar patrón en el tablero de ajedrez, utilizamos la función, **cv2.findChessboardCorners()**. También necesitamos pasar qué tipo de patrón estamos buscando, como la cuadrícula 8×8, la cuadrícula 5×5, etc. En este ejemplo, utilizamos una cuadrícula 7×6. (Normalmente un tablero de ajedrez tiene 8×8 cuadrados y 7×7 esquinas internas). Devuelve los puntos de las esquinas y el retval será True si se obtiene el patrón. Estas esquinas se colocarán en orden (de izquierda a derecha, de arriba a abajo).

Ver también:

Es posible que esta función no pueda encontrar el patrón requerido en todas las imágenes. Por lo tanto, una buena opción es escribir el código de tal manera que, se inicia la cámara y comprobar cada fotograma para el patrón requerido. Una vez obtenido el patrón, buscar las esquinas y guardarlo en una lista. También proporciona un cierto intervalo antes de leer el siguiente cuadro para que podamos ajustar nuestro tablero de ajedrez en diferente dirección. Continuar este proceso hasta que se obtenga el número requerido de buenos patrones. Incluso en el ejemplo que se ofrece aquí, no estamos seguros de las 14 imágenes dadas, cuántas son buenas. Así que leemos todas las imágenes y tomamos las buenas.

Ver también:

En vez de tablero de ajedrez, podemos usar alguna cuadrícula circular, pero luego usar la función **cv2.findCirclesGrid ()** para encontrar el patrón. Se dice que un número menor de imágenes es suficiente cuando se usa una cuadrícula circular.

Una vez que encontramos las esquinas, podemos aumentar su precisión usando **cv2.cornerSubPix ()**. También podemos dibujar el patrón usando **cv2.drawChessboardCorners ()**. Todos estos pasos están incluidos en el siguiente código:

```
import numpy as np
import cv2
import glob

# termination criterio
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# preparar puntos de objeto, como (0,0,0,0), (1,0,0,0), (2,0,0,0)...., (6,5,0)
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays para almacenar puntos de objeto y puntos de imagen de todas las
imágenes.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

images = glob.glob('*.jpg')

for fname in images:
  img = cv2.imread(fname)
  gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

  # Encuentra las esquinas del tablero de ajedrez
  ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

  # Si se encuentran, añada puntos de objeto, puntos de imagen (después de
refinarlos)
  if ret == True:
    objpoints.append(objp)

    corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
    imgpoints.append(corners2)

    # Dibuja y muestra las esquinas
    img = cv2.drawChessboardCorners(img, (7,6), corners2,ret)
    cv2.imshow('img',img)
    cv2.waitKey(500)

cv2.destroyAllWindows()
```

A continuación se muestra una imagen con el patrón dibujado en ella:

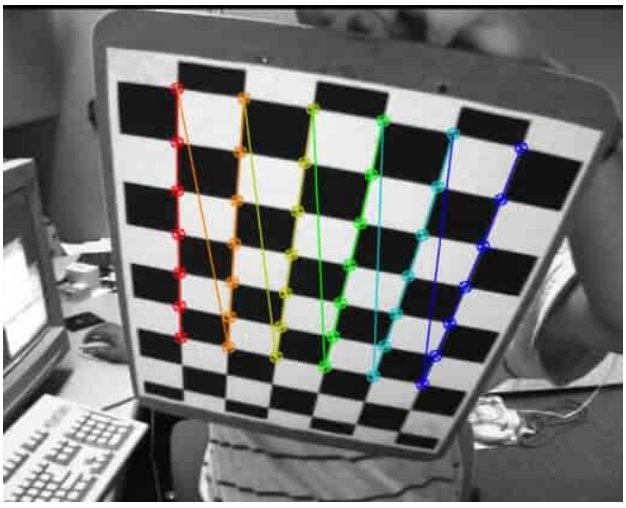


Fig.141: Imagen con patrón para calibración.

7.19.4.- Calibración

Ahora ya tenemos nuestros puntos de objeto y puntos de imagen listos para ser calibrados. Para ello utilizamos la función, **cv2.calibrateCamera ()**. Retorna la matriz de la cámara, coeficientes de distorsión, vectores de rotación y traslación, etc.

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1],None,None)
```

7.19.5.- No deformación

Tenemos lo que estábamos intentando. Ahora podemos tomar una imagen y no distorsionarla. Open CV viene con dos métodos, veremos ambos. Pero antes de eso, podemos refinar la matriz de la cámara basada en un parámetro de escalado libre usando **cv2.getOptimalNewCameraMatrix ()**. Si el parámetro de escala alpha=0, devuelve la imagen sin distorsión con píxeles no deseados mínimos. Así que podemos incluso eliminar algunos píxeles en las esquinas de la imagen. Si alfa=1, todos los píxeles se mantienen con algunas imágenes negras adicionales. También retorna un ROI de imagen que se puede utilizar para recortar el resultado.

Así que tomamos una nueva imagen (left02. jpg en este caso.)

1. Usando cv2. unistort ()

Este es el camino más corto. Simplemente llamamos a la función y utilizamos el ROI obtenido arriba para recortar el resultado.

```
# undistort
dst = cv2.undistort(img, mtx, dist, None, newcameramtx)

# crop the image
x,y,w,h = roi
dst = dst[y:y+h, x:x+w]
cv2.imwrite('calibresult.png',dst)
```

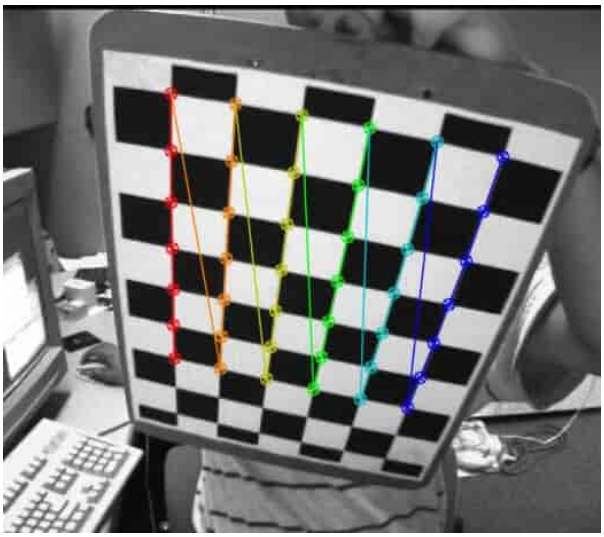
2. Uso del remapeo

Este es un camino curvo. Primero encuentra una función de mapeo desde una imagen distorsionada a una imagen no distorsionada. A continuación, utiliza la función remap.

```
# undistort
mapx,mapy = cv2.initUndistortRectifyMap(mtx,dist,None,newcameramtx,(w,h),5)
dst = cv2.remap(img,mapx,mapy,cv2.INTER_LINEAR)

# crop the image
x,y,w,h = roi
dst = dst[y:y+h, x:x+w]
cv2.imwrite('calibresult.png',dst)
```

Ambos métodos dan el mismo resultado. Veamos el resultado a continuación:

Sin calibrar:



Resultado de calibración

Fig.142: Resultados de calibración.

Se puede ver en el resultado que todos los bordes son rectos.

Ahora podemos almacenar la matriz de la cámara y los coeficientes de distorsión usando las funciones de escritura en Numpy (np. savez, np. savetxt etc) para usos futuros.

7.19.6.- Error de re-proyección

El error de re-proyección da una buena estimación de cuán exactos son los parámetros encontrados. Esto debería estar lo más cerca posible de cero. Dadas las matrices intrínsecas, de distorsión, rotación y traslación, primero transformamos el punto objeto punto a punto de imagen usando **cv2.projectPoints ()**. A continuación calculamos la norma absoluta entre lo que

obtuvimos con nuestra transformación y el algoritmo de búsqueda de esquinas. Para encontrar el error medio se calcula la media aritmética de los errores calculados para todas las imágenes de calibración.

```
mean_error = 0
for i in xrange(len(objpoints)):
  imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i], tvecs[i], mtx,
dist)
  error = cv2.norm(imgpoints[i],imgpoints2, cv2.NORM_L2)/len(imgpoints2)
  tot_error += error
print "total error: ", mean_error/len(objpoints)
```

7.19.7.- Pose Estimación

Durante la anterior sección de calibración de la cámara, se ha encontrado la matriz de la cámara, los coeficientes de distorsión, etc. Dada una imagen patrón, podemos utilizar la información anterior para calcular su posición, o cómo el objeto está situado en el espacio, como por ejemplo cómo se rota, cómo se desplaza, etc. Para un objeto plano, podemos asumir Z=0, de manera que el problema ahora es cómo se coloca la cámara en el espacio para ver nuestra imagen patrón. Por lo tanto, si sabemos cómo se encuentra el objeto en el espacio, podemos dibujar algunos diagramas 2D en él para simular el efecto 3D. Veamos el procedimiento.

El problema es que queremos dibujar nuestro eje de coordenadas 3D (ejes X, Y, Z) en la primera esquina de nuestro tablero de ajedrez. Eje X en color azul, eje Y en color verde y eje Z en color rojo. Así que en efecto, el eje Z debería sentirse como si fuera perpendicular a nuestro plano de ajedrez.

Primero, vamos a cargar la matriz de la cámara y los coeficientes de distorsión del resultado de calibración anterior.

```
import cv2
import numpy as np
import glob

# Load previously saved data
with np.load('B.npz') as X:
  mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]
```

Ahora creamos una función, draw que toma las esquinas en el tablero de ajedrez (obtenido usando **cv2.findChessboardCorners ()** y los ejes apuntan a dibujar un eje 3D.

```
def draw(img, corners, imgpts):
 corner = tuple(corners[0].ravel())
 img = cv2.line(img, corner, tuple(imgpts[0].ravel()), (255,0,0), 5)
 img = cv2.line(img, corner, tuple(imgpts[1].ravel()), (0,255,0), 5)
 img = cv2.line(img, corner, tuple(imgpts[2].ravel()), (0,0,255), 5)
 return img
```

A continuación, creamos criterios de terminación, puntos de objeto (puntos 3D de esquinas en tablero de ajedrez) y puntos de eje. Los puntos de eje son puntos en el espacio 3D para dibujar el eje. Dibujamos el eje de longitud 3 (las unidades serán en términos de ajedrez de tamaño cuadrado ya que nos calibramos en base a ese tamaño). Así que nuestro eje X está dibujado de (0,0,0,0) a (3,0,0,0), así que para el eje Y. Para el eje Z, se toma de (0,0,0,0) a (0,0, -3). Negativo significa que se dibuja hacia la cámara.

```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)
```

```
axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape(-1,3)
```

Ahora, cargamos cada imagen. Busca la cuadrícula 7×6. Si lo encontramos, lo refinamos con subpíxel. A continuación, para calcular la rotación y la traslación, utilizamos la función **cv2.solutionPnPRansac ()**. Una vez esas matrices de transformación las usamos para proyectar nuestros ejes al plano de la imagen. En otras palabras, encontramos los puntos en el plano de la imagen correspondientes a cada uno de (3,0,0,0), (0,3,0,0), (0,0,3) en el espacio 3D. Una vez que los tenemos, dibujamos líneas desde la primera esquina hasta cada uno de estos puntos usando nuestra función draw ().

```
for fname in glob.glob('left*.jpg'):
  img = cv2.imread(fname)
  gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
  ret, corners = cv2.findChessboardCorners(gray, (7,6),None)

  if ret == True:

    corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

    # Find the rotation and translation vectors.
    ret,rvecs, tvecs, inliers = cv2.solvePnP(objp, corners2, mtx, dist)

    # project 3D points to image plane
    imgpts, jac = cv2.projectPoints(axis, rvecs, tvecs, mtx, dist)

    img = draw(img,corners2,imgpts)
    cv2.imshow('img',img)
    k = cv2.waitKey(0) & 0xFF
    if k == ord('s'):
      cv2.imwrite(fname[:6]+'.png', img)

cv2.destroyAllWindows()
```

Veamos algunos resultados a continuación. Observar que cada eje tiene 3 cuadrados de longitud:

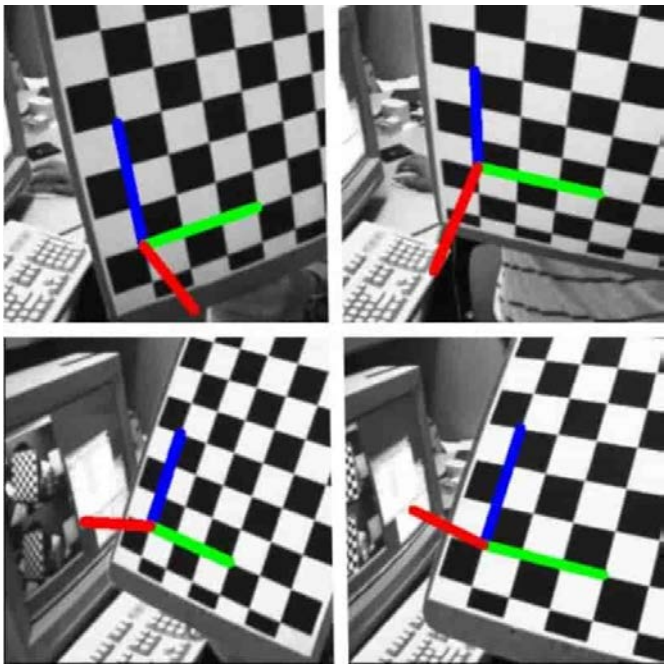


Fig.143: Pose estimación.

7.19.8.- Geometría Epipolar

Cuando adquirimos una imagen con la cámara, perdemos una información importante, es decir, la profundidad de la imagen. O qué tan lejos está cada punto de la imagen de la cámara porque es una conversión de 3D a 2D. Así que una pregunta importante es si podemos encontrar la información de profundidad usando estas cámaras. Y la respuesta es usar más de una cámara (Por ejemplo: Cámara dual). Nuestros ojos funcionan de la misma manera que cuando usamos dos cámaras (dos ojos) que se llama visión estéreo. Veamos lo que dispone Open CV en este cometido.

Antes de abordar las imágenes de profundidad, vamos a considerar primero algunos conceptos básicos en geometría multiview. En esta sección trataremos la geometría epipolar. Veamos la siguiente imagen que muestra una configuración básica con dos cámaras tomando la imagen de la misma escena.

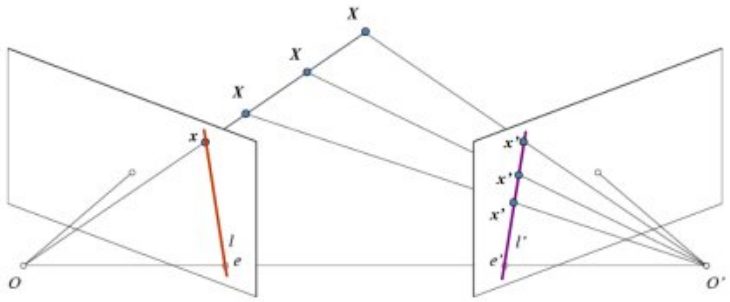


Fig.144: Referencia estructura de Geometría Epipolar. Cámara dual.

Si estamos usando sólo la cámara izquierda, no podemos encontrar el punto 3D correspondiente al punto x en la imagen porque cada punto de la línea OX se proyecta al mismo punto en el plano de la imagen. Pero de todas formas, consideremos también la imagen correcta. Ahora diferentes puntos de la línea OX se proyectan a diferentes puntos (x') en el plano derecho. Así que con estas dos imágenes podemos triangular el punto 3D correcto. Esta la clave de la idea.

La proyección de los diferentes puntos en OX forman una línea en el plano derecho (línea l'). Lo llamamos epilina correspondiente al punto x. Significa, para encontrar el punto x en la imagen derecha, buscar a lo largo de esta epilina. Debería estar en algún lugar de esta línea (Pensemos de esta forma, para encontrar el punto de coincidencia en otra imagen, no necesitas buscar la imagen completa, sólo buscar a lo largo de la epilina. Por lo tanto, proporciona un mejor rendimiento y precisión). Esto se llama Restricción Epipolar. Del mismo modo, todos los puntos tendrán sus correspondientes epilíneas en la otra imagen. El avión XOO' se llama Plano Epipolar.

O y O' son los centros de la cámara. A partir de la configuración realizada anteriormente, se puede ver que la proyección de la cámara derecha O' se ve en la imagen izquierda en el punto, e. Se llama epípolo. Epipole es el punto de intersección de la línea a través de los centros de la cámara y los planos de la imagen. Similarmente e' es el epípolo de la cámara izquierda. En algunos casos, no podremos localizar el epípolo en la imagen, puede que esté fuera de la imagen (lo que significa que una cámara no ve la otra).

Todas las epilinas pasan por su epipolo. Así que para encontrar la ubicación de epipole, podemos encontrar muchas epilinas y encontrar su punto de intersección.

Así que en esta sesión, nos enfocamos en encontrar líneas epipolares y epípoles. Pero para encontrarlos, necesitamos dos ingredientes más, la Matriz Fundamental (F) y la Matriz Esencial (E). La Matriz esencial contiene la información sobre traslación y rotación, que describe la ubicación de la segunda cámara en relación con la primera en coordenadas globales. Veamos la siguiente imagen:

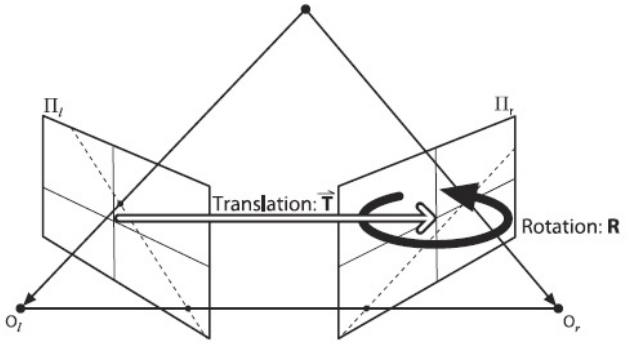


Fig.145: Planos en la Geometría Epipolar.

7.19.8.1.- Matriz esencial

Pero preferimos que las mediciones se hagan en coordenadas de píxeles, ¿no? la Matriz Fundamental contiene la misma información que la Esencial, además de la información sobre los intrínsecos de ambas cámaras para que podamos relacionar las dos cámaras en coordenadas de píxeles. (Si estamos utilizando imágenes rectificadas y normalizamos el punto dividiendo las distancias focales, F=E). En términos sencillos, la Matriz Fundamental F, asigna un punto en una imagen a una línea (epilínea) en la otra imagen. Esto se calcula a partir de los puntos de coincidencia de ambas imágenes. Se requieren un mínimo de 8 puntos de este tipo para encontrar la matriz fundamental (utilizando un algoritmo de 8 puntos). Se prefieren más puntos y utilizar RANSAC para obtener un resultado más sólido.

7.19.8.2.- Código de Geometría Epipolar

Primero se requiere encontrar la mayor cantidad posible de coincidencias entre dos imágenes para encontrar la matriz fundamental. Para ello, utilizamos descriptores SIFT con un matcher basado en FLANN y una prueba de ratio.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img1 = cv2.imread('myleft.jpg',0) #queryimage # left image
img2 = cv2.imread('myright.jpg',0) #trainimage # right image

sift = cv2.SIFT()

#Encuentra los puntos clave y descriptores con SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)

# FLANN parametros
FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks=50)
```

```
flann = cv2.FlannBasedMatcher(index_params,search_params)
matches = flann.knnMatch(des1,des2,k=2)

good = []
pts1 = []
pts2 = []

# prueba de proporción según el paper de Lowe' s
for i,(m,n) in enumerate(matches):
  if m.distance < 0.8*n.distance:
    good.append(m)
    pts2.append(kp2[m.trainIdx].pt)
    pts1.append(kp1[m.queryIdx].pt)
```

Ahora tenemos la lista de las mejores coincidencias de ambas imágenes. Busquemos la Matriz Fundamental.

```
pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
F, mask = cv2.findFundamentalMat(pts1,pts2,cv2.FM_LMEDS)

# We select only inlier points
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]
```

A continuación encontramos las epilinas. Las epilíneas correspondientes a los puntos de la primera imagen se dibujan en la segunda imagen. Así que aquí es importante mencionar las imágenes correctas. Tenemos un arsenal de líneas. Así que definimos una nueva función para dibujar estas líneas en las imágenes.

```
def drawlines(img1,img2,lines,pts1,pts2):
  ''' img1 - image on which we draw the epilines for the points in img2
      lines - corresponding epilines '''
  r,c = img1.shape
  img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
  img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
  for r,pt1,pt2 in zip(lines,pts1,pts2):
    color = tuple(np.random.randint(0,255,3).tolist())
    x0,y0 = map(int, [0, -r[2]/r[1] ])
    x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
    img1 = cv2.line(img1, (x0,y0), (x1,y1), color,1)
    img1 = cv2.circle(img1,tuple(pt1),5,color,-1)
    img2 = cv2.circle(img2,tuple(pt2),5,color,-1)
  return img1,img2
```

Ahora encontramos las epilinas en ambas imágenes y las dibujamos.

```
#Encuentra epilíneas correspondientes a puntos en la imagen derecha (segunda
imagen) y
# dibujando sus líneas en la imagen de la izquierda
lines1 = cv2.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)

# Encuentra epilíneas correspondientes a puntos en la imagen de la izquierda
(primera imagen) y
# dibujando sus líneas en la imagen derecha
lines2 = cv2.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
img3,img4 = drawlines(img2,img1,lines2,pts2,pts1)

plt.subplot(121),plt.imshow(img5)
plt.subplot(122),plt.imshow(img3)
```

```
plt.show()
```

A continuación, el resultado que se obtiene:

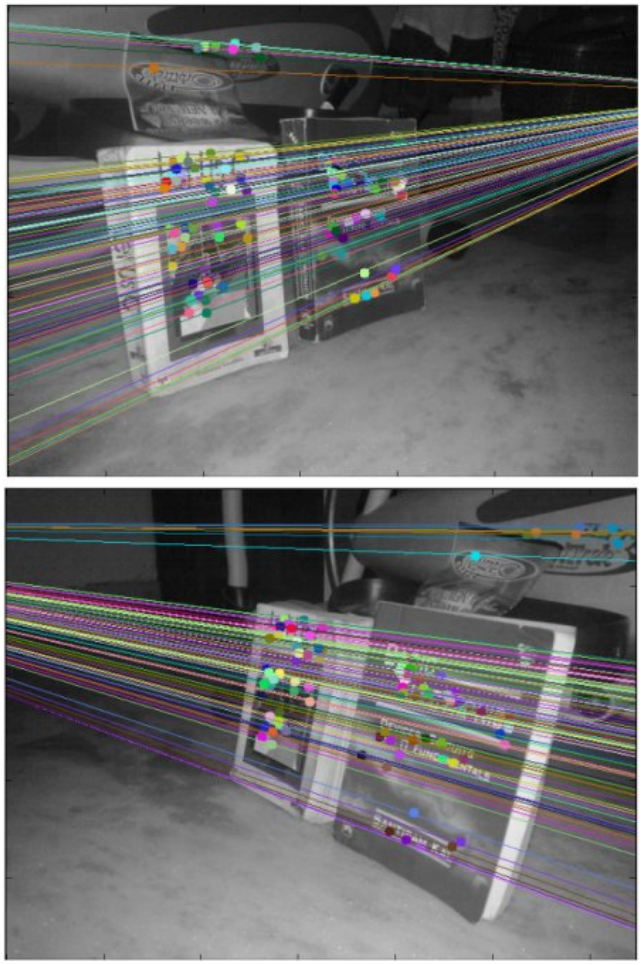


Fig.146: Imagen obtenido por aplicación de la Geometría Epipolar.

Podemos ver en la imagen de la izquierda que todas las epilíneas convergen en un punto fuera de la imagen del lado derecho. Ese punto de encuentro es el epípolo.

Para disponer de mejores resultados, debemos utilizar imágenes con buena resolución y muchos puntos no planares.

7.20.- Fotografía computacional y Detección de Facial y Caras

7.20.1.- Denoising de imagen

En secciones anteriores, hemos podido ver muchas técnicas de suavizado de imágenes como el desenfoque gaussiano, el desenfoque medio, etc. y han sido buenas en cierta medida para eliminar pequeñas cantidades de ruido. En esas técnicas, tomamos un pequeño vecindario alrededor de un píxel e hicimos algunas operaciones como el promedio ponderado gaussiano, mediana de los valores etc para reemplazar el elemento central. En resumen, la eliminación del ruido en un píxel era local a su vecindario.

Consideremos una propiedad principal del ruido. El ruido se considera generalmente como una variable aleatoria con una media cero. Consideremos un píxel con ruidos, $p = p\_0 + n$ donde

p_0 es el valor verdadero del píxel y *n* es el ruido en ese píxel. Podemos tomar un gran número de los mismos píxeles (digamos *N*) de diferentes imágenes y calcular su promedio. Idealmente, deberíamos obtener *p = p_0* en tanto que la media del ruido es cero.

Nosotros mismos podemos verificarlo mediante una sencilla configuración. Mantengamos una cámara estática en un lugar determinado durante un par de segundos. Esto nos aportará un montón de fotogramas, o muchas imágenes de la misma escena. A continuación, escribamos un código para encontrar el promedio de todos los fotogramas del vídeo. Comparar el resultado final y el primer cuadro. Se puede ver la reducción del ruido. Desafortunadamente, este sencillo método no es robusto a los movimientos de cámara y escena. También a menudo sólo hay una imagen ruidosa disponible.

Así que la idea es simple, necesitamos un conjunto de imágenes similares para promediar el ruido. Considere una ventana pequeña (digamos una ventana de 5×5) en la imagen. La probabilidad es grande de que el mismo parche pueda estar en otra parte de la imagen. A veces en un lugar pequeño a su alrededor. ¿Qué tal si usamos estos parches similares juntos y encontramos su promedio? Para esa ventana en particular, está bien.

La idea es que elegimos un pixel con ruido y tomamos una pequeña ventana alrededor de él, buscamos ventanas similares en la imagen, promediamos todas las ventanas y reemplazamos el píxel con el resultado que obtuvimos. Este método es la eliminación de medios no locales. Requiere más tiempo en comparación con las técnicas de desenfoque que vistas anteriormente, pero con resultado muy bueno.

En el caso de las imágenes en color, la imagen se convierte al espacio de color CIELAB y luego se denotan por separado los componentes L y AB.

7.20.1.1.- Denoising de imágenes en Open CV

Open CV proporciona cuatro variaciones de esta técnica.

1. **cv2.fastNlMeansDenoising ()** – funciona con una sola imagen en escala de grises
2. **cv2.fastNlMeansDenoisingColored ()** – funciona con una imagen en color.
3. **cv2.fastNlMeansDenoisingMulti ()** – funciona con la secuencia de imágenes capturadas en un corto periodo de tiempo (imágenes en escala de grises)
4. **cv2.fastNlSignificaDenoisingColoredMulti ()** – igual que arriba, pero para imágenes en color.

Los argumentos comunes son:

- h: parámetro que determina la fuerza del filtro. Un valor h más alto elimina mejor el ruido, pero también elimina los detalles de la imagen. (10 está bien)
- hForColorComponentes: igual que h, pero sólo para imágenes en color. (normalmente igual que h)
- templateWindowSize: debe ser impar. (recomendado 7)
- searchWindowSize: debe ser impar. (recomendado 21)

Demostraremos el punto 2. Dejamos el resto para que pruebe el lector un poco con estas funciones.

1. cv2. fastNlSignificaDenoisingColored ()

Como se indicó anteriormente, se utiliza para eliminar el ruido de las imágenes en color. (El ruido se espera que sea gaussiano). Veamos el ejemplo a continuación:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('fastNlMeansDenoising.jpg')

dst = cv2.fastNlMeansDenoisingColored(img,None,10,10,7,21)

plt.subplot(121),plt.imshow(img)
plt.subplot(122),plt.imshow(dst)
plt.show()
```

A la imagen de la derecha se le ha podido eliminar el ruido en un porcentaje bastante alto mejorando su calidad.

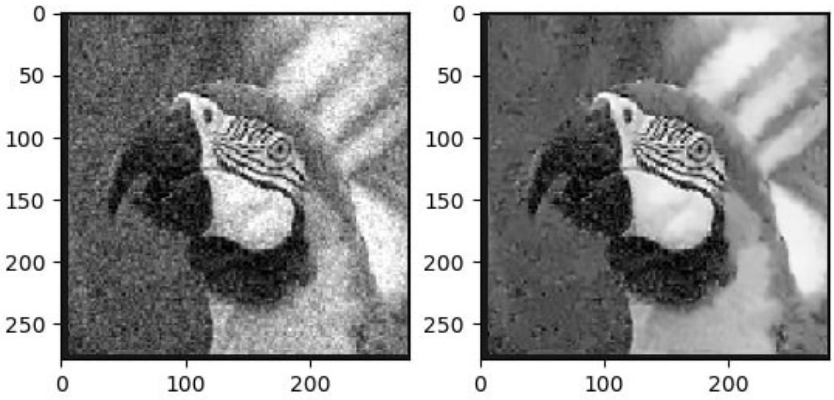


Fig.147: Supresión de ruido en una imagen mediante funciones propias de Open CV.

7.20.2.- Detección de rostros, caras y ojos con Haar Cascades

La detección de objetos mediante clasificadores en cascada basados en funciones de Haar es un método eficaz de detección de objetos propuesto por Paul Viola y Michael Jones en su documento,"*Rapid Object Detection using a Boosted Cascade of Simple Features"* en 2001. Se trata de un enfoque basado en el aprendizaje automático en el que la función en cascada se entrena a partir de muchas imágenes positivas y negativas. A continuación, se utiliza para detectar objetos en otras imágenes.

Aquí veremos con la **detección de rostros o caras**. Inicialmente, el algoritmo necesita muchas imágenes positivas (imágenes de caras) y negativas (imágenes sin caras) para entrenar el clasificador. Necesitamos extraer rasgos de él. Para ello se utilizan las características de Haar mostradas en la siguiente imagen. Son como un núcleo convolucional. Cada característica es un valor individual obtenido restando la suma de píxeles bajo rectángulo blanco de la suma de píxeles bajo rectángulo negro.

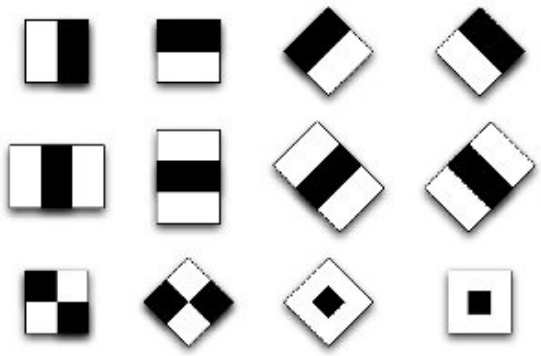


Fig.148: Clasificadores de Haar.

Ahora se utilizan todos los tamaños y ubicaciones posibles de cada núcleo para calcular un montón de características. (¿Sólo imaginemos cuánto cálculo requiere? Incluso una ventana de 24×24 resulta con más de 160000 características). Para cada cálculo de características, necesitamos encontrar la suma de píxeles bajo rectángulos blancos y negros. Para resolver esto, introdujeron las imágenes integrales. Simplifica el cálculo de la suma de píxeles, qué tan grande puede ser el número de píxeles, a una operación que involucra sólo cuatro píxeles. Curioso, ¿verdad?. Hace que las cosas sean súper rápidas.

Pero entre todas estas características calculamos que la mayoría de ellas son irrelevantes. Por ejemplo, considere la siguiente imagen. La fila superior muestra dos buenas características. El primer rasgo seleccionado parece centrarse en la propiedad de que la región de los ojos es a menudo más oscura que la región de la nariz y las mejillas. La segunda característica seleccionada se basa en la propiedad de que los ojos son más oscuros que el puente de la nariz. Pero las mismas ventanas que se aplican en las mejillas o cualquier otro lugar es irrelevante. Entonces, ¿cómo seleccionamos las mejores características de las más de 16000000 características? Lo consigue Adaboost.

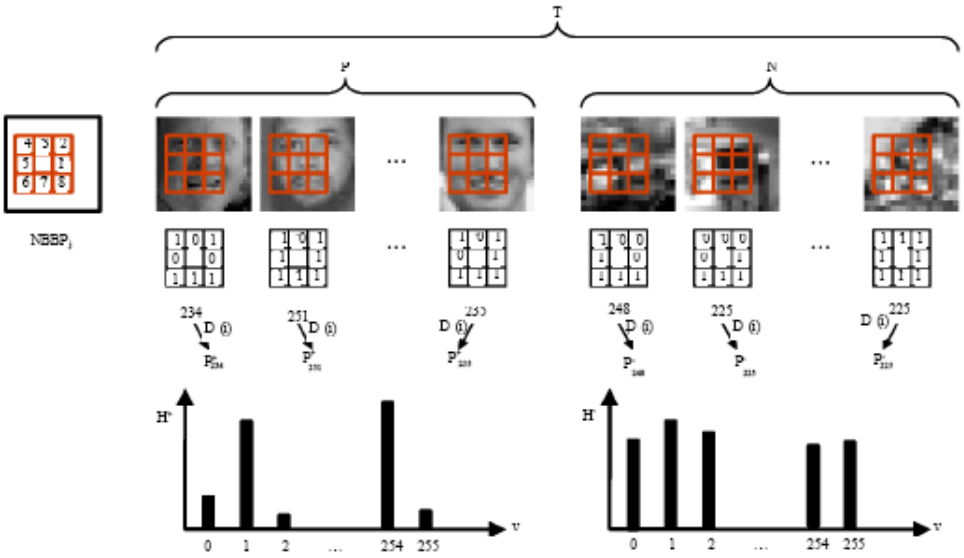


Fig.149: Proceso de clasificación del algoritmo Haars.

Para ello, aplicamos todas y cada una de las características en todas las imágenes de entrenamiento. Para cada característica, encuentra el mejor umbral que clasificará los rostros en positivos y negativos. Pero obviamente, habrá errores o clasificaciones erróneas. Seleccionamos

las características con una tasa de error mínima, lo que significa que son las características que mejor clasifican las imágenes faciales y no faciales. (El proceso no es tan simple como esto. Al principio, cada imagen tiene el mismo peso. Después de cada clasificación, aumenta el peso de las imágenes mal clasificadas. De nuevo se hace el mismo proceso. Se calculan nuevas tasas de error. También nuevos pesos. El proceso continúa hasta que se alcanza la precisión o la tasa de error requerida o se encuentra el número requerido de características).

El clasificador final es una suma ponderada de estos clasificadores débiles. Se llama débil porque solo no puede clasificar la imagen, pero junto con otros forma un fuerte clasificador. El papel dice que incluso 200 características proporcionan una detección con un 95% de precisión. Su configuración final tenía alrededor de 6000 características. (Imaginar una reducción de 16000000 características a 6000 características. Eso es una considerable ganancia).

Así que ahora tomamos una imagen. Tomar cada ventana de 24×24. Aplicamos 6000 características… ¿No es un poco ineficiente y lleva mucho tiempo? Sí, lo es. Los autores tienen una buena solución para eso.

En una imagen, la mayor parte de la región de la imagen es una región no facial. Por lo tanto, es mejor tener un método simple para comprobar si una ventana no es una región facial. Si no lo es, lo eliminamos de una sola atacada. No lo procesemos de nuevo. En su lugar, concentrémonos en la región donde puede haber un rostro. De esta forma, podemos encontrar más tiempo para comprobar una posible región facial.

Para ello se introdujo el concepto de **Cascada de clasificadores**. En lugar de aplicar todas las 6000 características en una ventana, agrupe las características en diferentes etapas de clasificadores y aplíquelas una por una. (Normalmente, las primeras etapas contienen un número de características muy inferior). Si una ventana falla en la primera etapa, eliminarla. No tenemos en cuenta las características restantes. Si pasa, aplicar la segunda etapa de las características y continuar el proceso. La ventana que pasa por todas las etapas es una zona de la cara.

El detector tenía más de 6000 características con 38 etapas con 1,10,25,25,25 y 50 características en las primeras cinco etapas. (Dos características de la imagen anterior se obtienen como las dos mejores características de Adaboost). Según los autores, en promedio, se evalúan 10 de las 6000 características por subventana.

Nota: Boosting es un método que pretende mejorar el desempeño de cualquier algoritmo de aprendizaje supervisado mediante la combinación de los resultados de varios clasificadores débiles o de base para obtener un clasificador final robusto. Una de las técnicas más populares de Boosting es el algoritmo Boosting Adaptativo (AdaBoost). La idea es, partiendo de clasificadores débiles (moderadamente precisos) se puede probar que es posible encontrar un clasificador más preciso combinando muchos "débiles". Pasamos de un clasificador débil a uno más fuerte. Utilizando técnicas como Bagging y Boosting. https://rephip.unr.edu.ar/handle/2133/14282

Así que esta es una explicación intuitiva de cómo funciona la detección de cara Viola-Jones.

7.20.2.1.- Detección de rostros o caras de Haar-cascade en Open CV

Open CV provee un entrenador y un detector. Si queremos entrenar nuestro propio clasificador para cualquier objeto como coches, aviones, etc. podemos usar este de Open CV.

Aquí veremos la detección. Open CV ya contiene muchos clasificadores pre-entrenados para cara, ojos, sonrisa, etc. Esos archivos XML se almacenan en opencv/data/haarcascades/ folder. Vamos a realizar un detector facial y ocular con Open CV.

Primero necesitamos cargar los clasificadores XML requeridos. A continuación, cargamos nuestra imagen de entrada (o vídeo) en modo escala de grises.

```
import numpy as np
import cv2

face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')

img = cv2.imread('camilla4.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Ahora encontramos las caras en la imagen. Si se encuentran caras, retorna las posiciones de las caras detectadas como Rect (x, y, w, h). Una vez que obtenemos estas ubicaciones, podemos crear una ROI para la cara y aplicar la detección de ojos en este ROI (partiendo de que los ojos están siempre en la cara).

```
faces = face_cascade.detectMultiScale(gray, 1.35, 1)
for (x,y,w,h) in faces:
  img = cv2.rectangle(img,(x,y),(x+w,y+h),(255,100,100),2)
  roi_gray = gray[y:y+h, x:x+w]
  roi_color = img[y:y+h, x:x+w]
  eyes = eye_cascade.detectMultiScale(roi_gray)
  for (ex,ey,ew,eh) in eyes:
    cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(40,55,200),2)

cv2.imshow('img',img)
cv2.imwrite('img.jpg',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

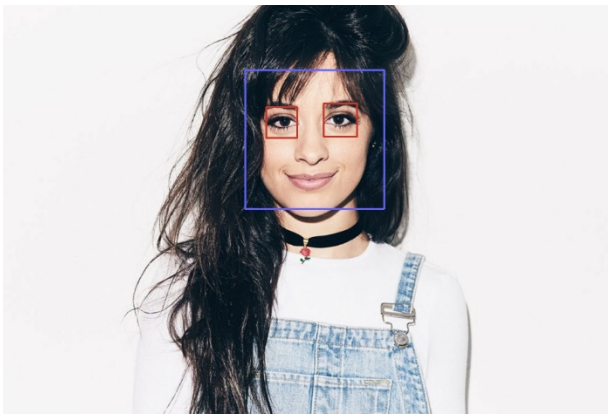


Fig.150: Ejemplo de detección de caras con clasificador de Haar.

Parte II: Realización práctica. Implementación de programación.

8.- Ejemplos prácticos.

A continuación, tras la introducción a los principales puntos teórico – práctico que componen los sistemas de visión e inteligencia artificial, pasamos a continuación a la realización de una programación práctica de los mismos y a evaluar los resultados. Para ello, se realiza un ejemplo

práctico número uno, donde se evalúa y analiza los resultados obtenidos objetivo de este trabajo respecto de comparativa en tiempos de ejecución CPU vs. GPU. En este primer ejemplo se contempla el análisis de reconocimiento de imagen estática, fotograma.

A continuación pasamos al ejemplo práctico número 2, donde realizamos la misma evaluación, pero esta vez sobre imagen en movimiento, video. Para ello se realiza la implementación de un programa de reconocimiento facial.

8.1.- Ejemplo práctico 1. Reconocimiento de formas en imagen estática.

Como ejemplo práctico se realiza un programa en Phyton donde evaluaremos a través del modelo de entrenamiento de redes neuronales Keras y en el que realizaremos nuestras pruebas con dos datasets de entrenamiento sencillos basado en imágenes: uno de los protagonistas de juego de Pokemon, con un conjunto de 1200 imágenes y otro de animales, 900 imágenes, clasificación de distintas razas de perros. Igualmente trasladable a cualquier otro conjunto de imágenes.

Para ambos dataset realizamos varias pruebas principales basadas en la evaluación de los distintos tiempos de ejecución comparando los empleados por una CPU que en el caso de las pruebas consiste en un Intel i5 y una GPU proporcionada a través del entorno de pruebas que nos reporta Google Colab donde podemos indicar la ejecución de nuestro programa Phyton en GPU.

Evaluaremos las distintas precisiones realizadas por la red neuronal con respecto a los distintos números de época, siendo los referentes considerados de 25, 50 y 75 épocas. También se ha contemplado un caso en el que modificamos el número de divisiones del batch size (división del set de datos en lotes (batches) que, teniendo como valor referente de 75, evaluamos para 100. Adelantamos en este apartado que no se obtuvieron mejores resultados. El motivo es porque hemos de indicar que estos dataset aquí evaluados son de evaluación y de un valor de consideración respecto a una evaluación óptima en nuestro caso considerado de pura prueba experimental y más bien de valor medio. Entendiendo por esto que para este valor obtener resultados óptimos se deberían de considerar otros dataset de mayor composición para llegar a observar resultados óptimos respecto de este parámetro.

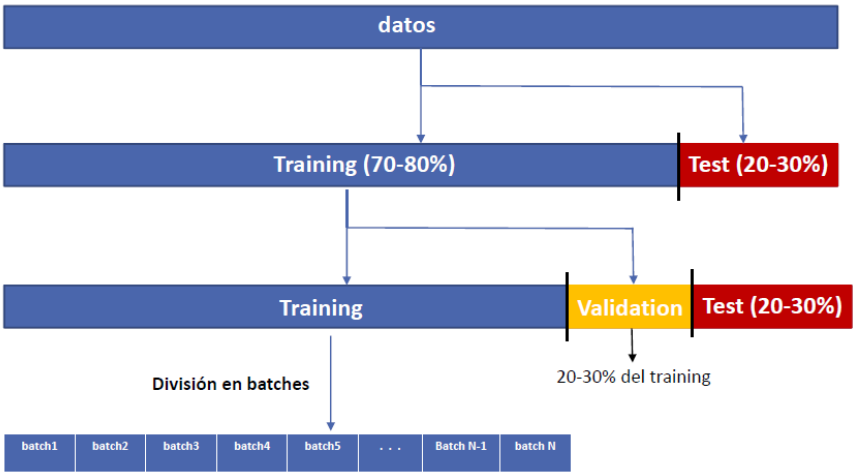


Fig. 151: División del set de datos de entrenamiento en lotes (batches).

8.1.1.- Caso de imágenes del juego de Pokemons:

En primer lugar descargamos el dataset a través de la plataforma Google Colab. Posteriormente descargamos las imágenes a un directorio nuestro de trabajo (ver In [1] DATA_DIR='Pokedex'), que corresponde realmente al directorio local c:\Pokedex.

Preparamos el programa con la importación de las librerías del modelo Keras respecto del tratamiento de imágenes. A continuación realizamos un array para cada una de las familias de imágenes del dataset Pokemon, consistente en unos subdirectorios con el contenido de cada familia de imágenes (Bulbasaur, Charmander, Pikachu, Newtwo y Squirtle).

A continuación pasamos el array por un etiquetador que será posteriormente evaluado en la etapa de entrenamiento por la función depp_CNN donde extraeremos el modelo que se compilará posteriormente y de la que evaluaremos la evolución y calidad de nuestro entrenador pasando como valores de entrenamiento los principales de nº de épocas = 50 y un batch_size=64. Obteniendo gráficas como la siguiente y comparada con la teórica (se adjunta muestras de alcance de valores de precisión y pérdida):

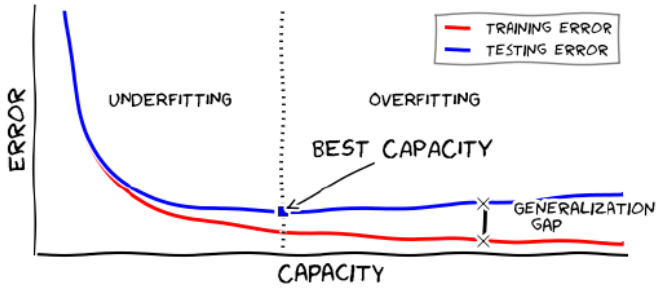


Fig.152: Respuesta de precisión y pérdidas del entrenador.

A continuación junto con la librería de gestión de imágenes Open CV, realizaremos la evaluación de nuestra imagen predicha con la real, obteniendo el siguiente resultado:

Código de programación:

A continuación reporto el programa donde se ha realizado la evaluación de la GPU.
Comentando los principales pasos.

1.- Preparamos los datos

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

```
!ls -la  '/content/drive/My Drive/dir_poke/dataset'
```

2.- Preparamos las variables de entrenamiento.

```
import os
import numpy as np
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
import cv2
import matplotlib.pyplot as plt

trainX = []
trainY = []
DATA_DIR = '/Pokedex'
train_dir = os.path.join(DATA_DIR)
```

3.- Preparamos la información respecto de las distintas imágenes que componen el dataset. Ejemplo de las imágenes de Bulbasaur y Charmander.

```
#Añadimos BULBASAUR

train_bulbasaur_dir = os.path.join(train_dir, 'bulbasaur')
train_bulbasaur_fnames = os.listdir(train_bulbasaur_dir)

for image in train_bulbasaur_fnames :
  bulbasaur = load_img(train_bulbasaur_dir + '/' + image, target_size=(32, 32))
  bulbasaur = np.resize(bulbasaur, (32, 32, 3))
  bulbasaur = img_to_array(bulbasaur)
  trainY.append('bulbasaur')
  trainX.append(bulbasaur)

#Añadimos CHARMANDER

train_charmander_dir = os.path.join(train_dir, 'charmander')
train_charmander_fnames = os.listdir(train_charmander_dir)

for image in train_charmander_fnames :
  charmander = load_img(train_charmander_dir + '/' + image, target_size=(32, 32))
  charmander = np.resize(charmander, (32, 32, 3))
  charmander = img_to_array(charmander)
  trainY.append('charmander')
  trainX.append(charmander)
```

4.- Entrenamos el modelo.

```
trainX = np.vstack(trainX)
trainY = np.vstack(trainY)
trainX = trainX.reshape([1168, 32, 32, 3])
labelNames = ['bulbasaur', 'charmander', 'mewtwo', 'pikachu', 'squirtle']

from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
trainY = le.fit_transform(trainY)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(trainX, trainY, test_size=0.2, random_state=42)
```

```
# importamos los paquetes necesarios
import numpy as np
from keras import backend as K
from keras.layers.convolutional import Conv2D
from keras.layers import Input
from keras.models import Model
from keras.layers.core import Activation, Flatten, Dense, Dropout
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import MaxPooling2D
from keras.models import Sequential
from keras.optimizers import SGD
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
```

5.- Definimos la función de la red neuronal.

```
def deep_CNN(width, height, depth, classes, batchNorm):

  # Definimos entradas en modo "channels last"
  inputs = Input(shape=(height, width, depth)) #(X)

  # Definimos la arquitectura
  # Primer set de capas CONV => RELU => CONV => RELU => POOL
  x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(inputs) #(X)
  if batchNorm:
    x1 = BatchNormalization()(x1) #(X)
  x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(x1) #(X)
  if batchNorm:
    x1 = BatchNormalization()(x1) #(X)
  x1 = MaxPooling2D(pool_size=(2, 2))(x1) #(X)
  x1 = Dropout(0.25)(x1) #(X)

  # Segundo set de capas CONV => RELU => CONV => RELU => POOL
  x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x1) #(X)
  if batchNorm: #(X)
    x2 = BatchNormalization()(x2) #(X)
  x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x2) #(X)
  if batchNorm:
    x2 = BatchNormalization()(x2) #(X)
  x2 = MaxPooling2D(pool_size=(2, 2))(x2) #(X)
  x2 = Dropout(0.25)(x2) #(X)

  # Primer (y único) set de capas FC => RELU
  xfc = Flatten()(x2) #(X)
  xfc = Dense(512, activation="relu")(xfc) #(X)
  if batchNorm:
    xfc = BatchNormalization()(xfc) #(X)
  xfc = Dropout(0.5)(xfc) #(X)
  # Clasificador softmax
  predictions = Dense(classes, activation="softmax")(xfc) #(X)

  # Unimos las entradas y el modelo mediante la función Model con parámetros inputs y ouputs
  model = Model(inputs=inputs, outputs=predictions) #(X)

  # La función debe devolver el modelo como salida
  return model
```

6.- Compilamos el modelo

```
# Compilar el modelo
print("[INFO]: Compilando el modelo...")
# Instanciamos el modelo ajustado al dataset CIFAR10
model = deep_CNN(32,32,3,5,True)
# Compilamos el modelo sacando como mérica el accuracy
model.compile(optimizer=SGD(lr=0.01, decay=0.01/50, momentum=0.9, nesterov=True), loss='sparse_categorical_cross

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
H = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs = 25, batch_size = 64, verbose=1)

# Almaceno el modelo en Drive
# Montamos la unidad de Drive
#drive.mount('/content/drive')
# Almacenamos el modelo empleando la función mdoel.save de Keras
model.save('/Pokedex/ModelMiniprojecte.h5')

# Evaluación del modelo
print("[INFO]: Evaluando el modelo...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
predictions = model.predict(x=X_test, batch_size=64)
# Sacamos el report para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=labelNames)) #(X)
```

7.- Visualizamos los datos

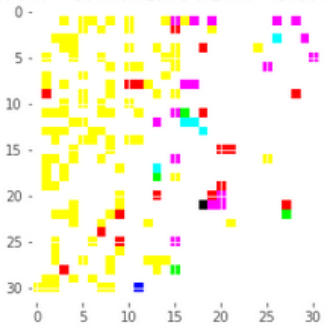
```
# Gráficas
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 25), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 25), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 25), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, 25), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

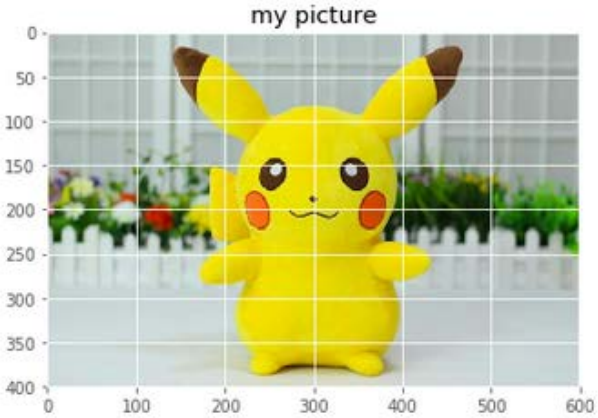
Pruebas en CPU

Para Épocas =25

```
[INFO]: Entrenando la red...
Train on 934 samples, validate on 234 samples
Epoch 1/25
934/934 [==============================] - 12s 12ms/step - loss: 1.3210 - acc: 0.5931 - val_loss: 3.4696 - v
al_acc: 0.5812
Epoch 2/25
934/934 [==============================] - 10s 11ms/step - loss: 0.6465 - acc: 0.8169 - val_loss: 1.7268 - v
al_acc: 0.6581
Epoch 3/25
934/934 [==============================] - 10s 11ms/step - loss: 0.4575 - acc: 0.8662 - val_loss: 1.3066 - v
al_acc: 0.6838
Epoch 4/25
934/934 [==============================] - 11s 11ms/step - loss: 0.3543 - acc: 0.8876 - val_loss: 2.0205 - v
al_acc: 0.6368
Epoch 5/25
934/934 [==============================] - 10s 11ms/step - loss: 0.3049 - acc: 0.9036 - val_loss: 1.4220 - v
 _
Epoch 21/25
934/934 [==============================] - 11s 11ms/step - loss: 0.0810 - acc: 0.9690 - val_loss: 0.6577 - v
al_acc: 0.8504
Epoch 22/25
934/934 [==============================] - 11s 11ms/step - loss: 0.1265 - acc: 0.9615 - val_loss: 0.5083 - v
al_acc: 0.8504
Epoch 23/25
934/934 [==============================] - 10s 11ms/step - loss: 0.0848 - acc: 0.9679 - val_loss: 0.4361 - v
al_acc: 0.9017
Epoch 24/25
934/934 [==============================] - 10s 11ms/step - loss: 0.0892 - acc: 0.9754 - val_loss: 0.4182 - v
al_acc: 0.8718
Epoch 25/25
934/934 [==============================] - 11s 12ms/step - loss: 0.0943 - acc: 0.9764 - val_loss: 0.3967 - v
al_acc: 0.8590
```

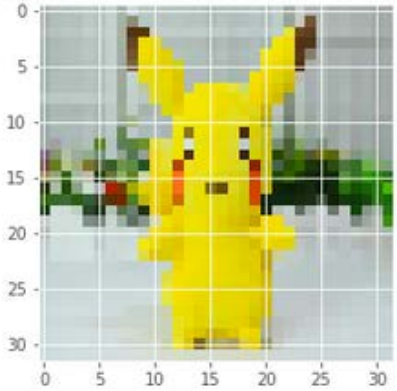
Predicción: pikachu - Confianza: 56.14% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[8.4728945e-06 2.7258971e-03 9.9710983e-01 9.0861147e-05 6.5026754e-05]]
```

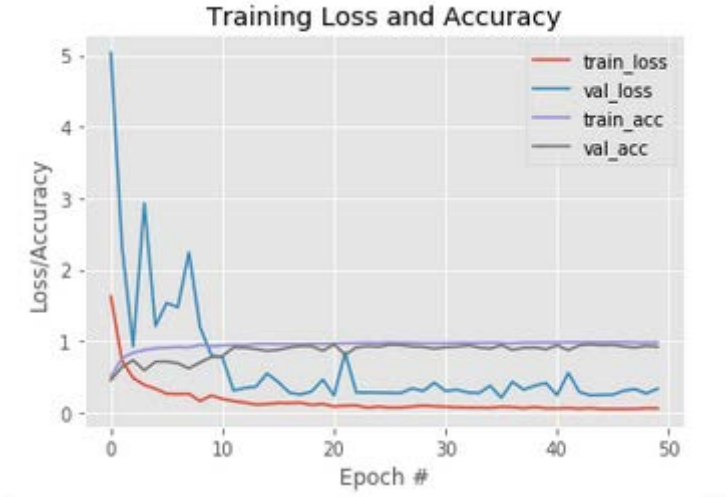
Predicción: mewtwo - Confianza: 99.71% - Ground Truth: pikachu



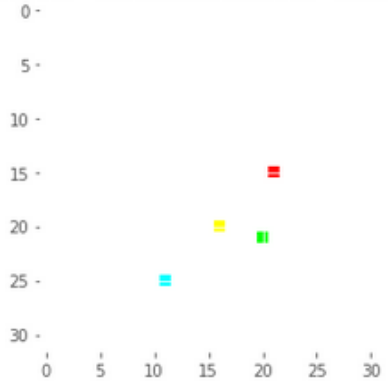
Para Épocas = 50

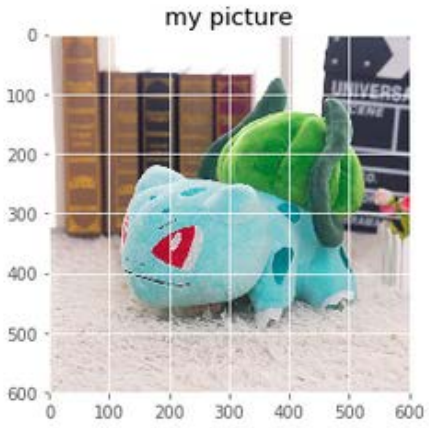
```
Train on 934 samples, validate on 234 samples
Epoch 1/50
934/934 [==============================] - 11s 12ms/step - loss: 1.6270 - acc: 0.4979 - val_loss: 5.0351 - v
al_acc: 0.4530
Epoch 2/50
934/934 [==============================] - 10s 11ms/step - loss: 0.7257 - acc: 0.7580 - val_loss: 2.3258 - v
al_acc: 0.6368
Epoch 3/50
934/934 [==============================] - 12s 12ms/step - loss: 0.4827 - acc: 0.8362 - val_loss: 0.9244 - v
al_acc: 0.7308
Epoch 4/50
934/934 [==============================] - 11s 11ms/step - loss: 0.3835 - acc: 0.8737 - val_loss: 2.9294 - v
al_acc: 0.5897
Epoch 5/50
934/934 [==============================] - 10s 11ms/step - loss: 0.3312 - acc: 0.8983 - val_loss: 1.2104 - v
al_acc: 0.7094
```

```
Epoch 46/50
934/934 [==============================] - 11s 11ms/step - loss: 0.0494 - acc: 0.9829 - val_loss: 0.2487 -
al_acc: 0.9444
Epoch 47/50
934/934 [==============================] - 10s 11ms/step - loss: 0.0485 - acc: 0.9839 - val_loss: 0.3042 -
al_acc: 0.9231
Epoch 48/50
934/934 [==============================] - 10s 11ms/step - loss: 0.0497 - acc: 0.9797 - val_loss: 0.3252 -
al_acc: 0.9103
Epoch 49/50
934/934 [==============================] - 10s 11ms/step - loss: 0.0589 - acc: 0.9775 - val_loss: 0.2661 -
al_acc: 0.9316
Epoch 50/50
934/934 [==============================] - 10s 11ms/step - loss: 0.0564 - acc: 0.9797 - val_loss: 0.3313 -
al_acc: 0.9188
```

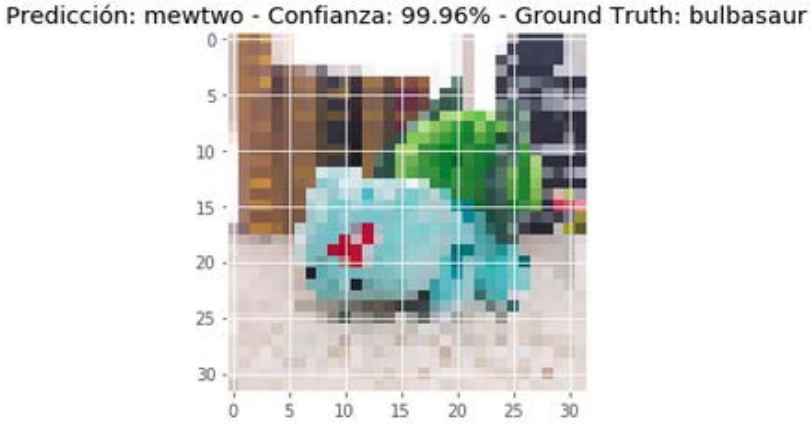


Predicción: bulbasaur - Confianza: 99.99% - Ground Truth: CIFAR10



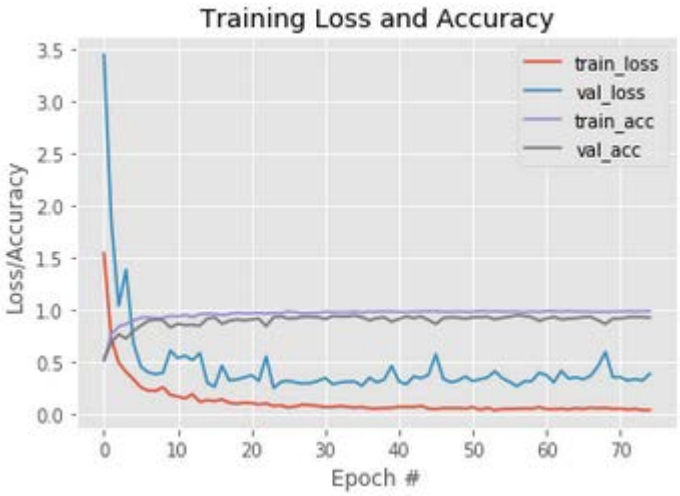


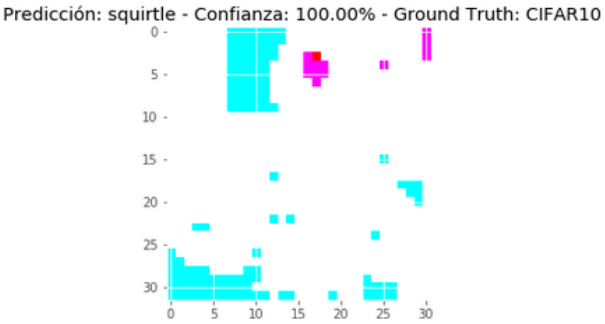

```
[INFO]: Clasificando imagen...
[[1.3120361e-07 4.3105494e-04 9.9955398e-01 4.2286192e-06 1.0576060e-05]]
```



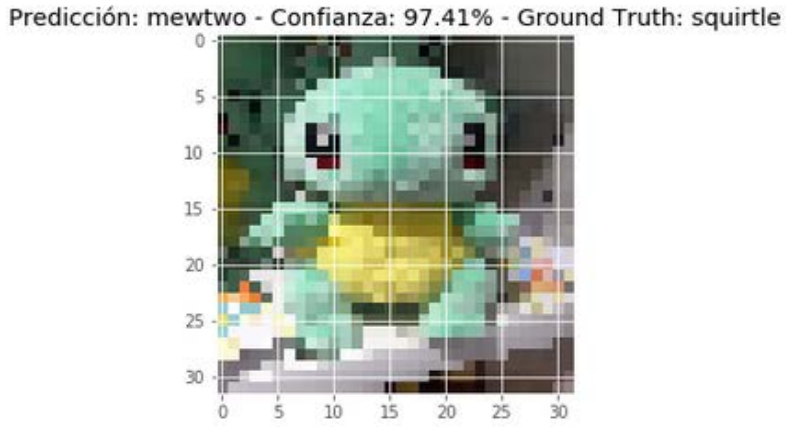
Para Épocas = 75

```
Train on 934 samples, validate on 234 samples
Epoch 1/75
934/934 [==============================] - 12s 13ms/step - loss: 1.5421 - acc: 0.5118 - val_loss: 3.4470 - v
al_acc: 0.5214
Epoch 2/75
934/934 [==============================] - 11s 12ms/step - loss: 0.7230 - acc: 0.7645 - val_loss: 1.8476 - v
al_acc: 0.6838
Epoch 3/75
934/934 [==============================] - 11s 12ms/step - loss: 0.4912 - acc: 0.8383 - val_loss: 1.0372 - v
al_acc: 0.7650
Epoch 4/75
934/934 [==============================] - 12s 12ms/step - loss: 0.4019 - acc: 0.8630 - val_loss: 1.3843 - v
al_acc: 0.7222
Epoch 5/75
934/934 [==============================] - 12s 13ms/step - loss: 0.3323 - acc: 0.8961 - val_loss: 0.6775 - v
al_acc: 0.7949
```


Training Loss and Accuracy
train_loss
val_loss
train_acc
val_acc
Loss/Accuracy
Epoch #
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
10
20
30
40
50
60
70


Predicción: squirtle - Confianza: 100.00% - Ground Truth: CIFAR10
0
5
10
15
20
25
30

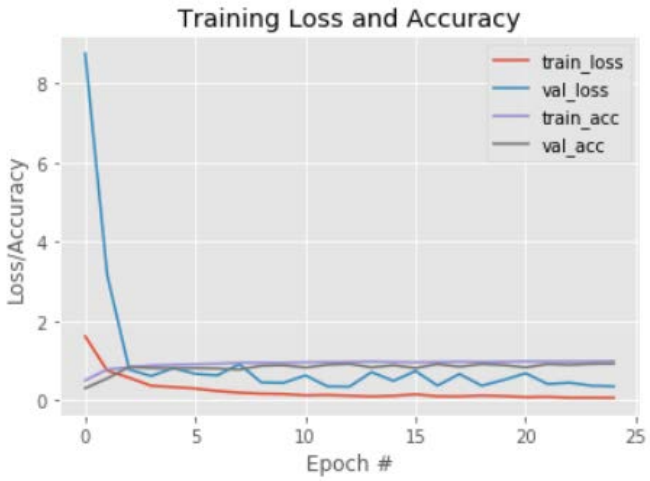

```
[INFO]: Clasificando imagen...
[[2.7327260e-07 1.9362275e-04 9.7409886e-01 1.4284908e-06 2.5705850e-02]]
```



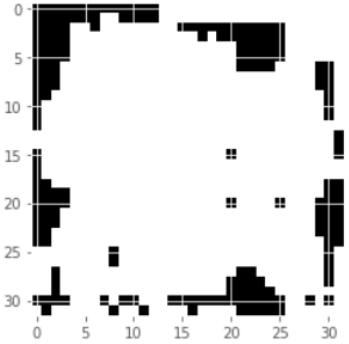
Para con GPU

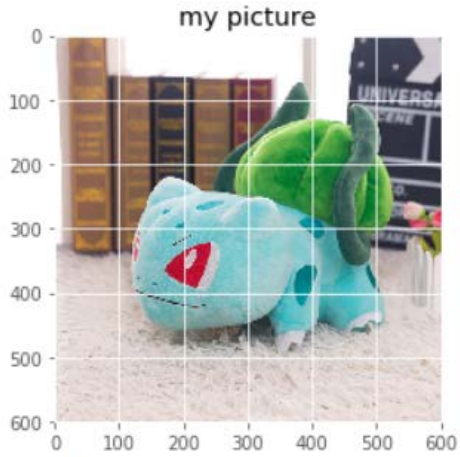
Para Épocas=25

```
1.13.1
['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Train on 934 samples, validate on 234 samples
Epoch 1/25
934/934 [==============================] - 3s 3ms/step - loss: 1.6100 - acc: 0.4968 - val_loss: 8.7481 -
Epoch 2/25
934/934 [==============================] - 0s 249us/step - loss: 0.7445 - acc: 0.7762 - val_loss: 3.1551
Epoch 3/25
934/934 [==============================] - 0s 222us/step - loss: 0.5635 - acc: 0.8233 - val_loss: 0.7700
Epoch 4/25
934/934 [==============================] - 0s 231us/step - loss: 0.3608 - acc: 0.8737 - val_loss: 0.6070
Epoch 5/25
934/934 [==============================] - 0s 220us/step - loss: 0.3274 - acc: 0.8929 - val_loss: 0.8068

Epoch 20/25
934/934 [==============================] - 0s 221us/step - loss: 0.1007 - acc: 0.9657 - val_loss: 0.5063 -
Epoch 21/25
934/934 [==============================] - 0s 205us/step - loss: 0.0732 - acc: 0.9754 - val_loss: 0.6777 -
Epoch 22/25
934/934 [==============================] - 0s 209us/step - loss: 0.0815 - acc: 0.9764 - val_loss: 0.4056 -
Epoch 23/25
934/934 [==============================] - 0s 207us/step - loss: 0.0600 - acc: 0.9764 - val_loss: 0.4385 -
Epoch 24/25
934/934 [==============================] - 0s 208us/step - loss: 0.0604 - acc: 0.9775 - val_loss: 0.3625 -
Epoch 25/25
934/934 [==============================] - 0s 223us/step - loss: 0.0570 - acc: 0.9850 - val_loss: 0.3435 -
```



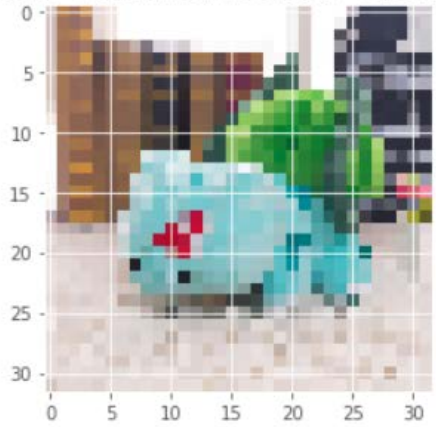
Predicción: bulbasaur - Confianza: 99.99% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[6.7740334e-06 3.1485030e-08 9.9999285e-01 3.7038851e-07 7.8900380e-09]]
```

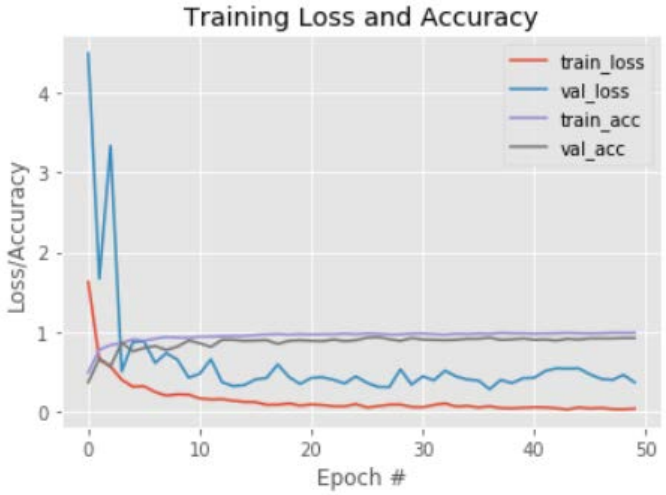
Predicción: mewtwo - Confianza: 100.00% - Ground Truth: bulbasaur



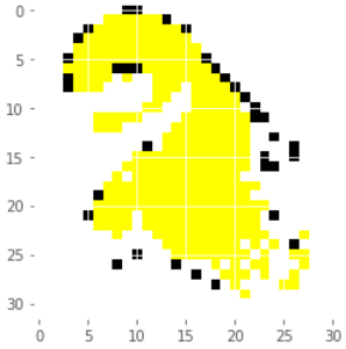
Para Épocas=50

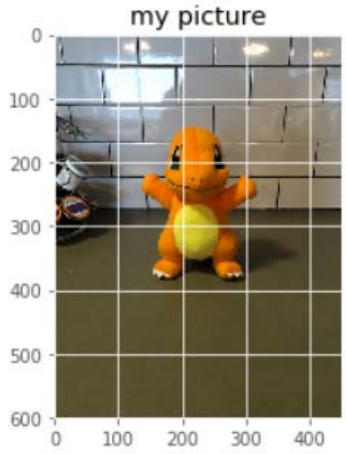
```
... 1.13.1
    ['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
    [INFO]: Compilando el modelo...
    [INFO]: Entrenando la red...
    WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py:3066: to_
    Instructions for updating:
    Use tf.cast instead.
    Train on 934 samples, validate on 234 samples
    Epoch 1/50
    934/934 [==============================] - 3s 4ms/step - loss: 1.6284 - acc: 0.5161 - val_loss: 5.6793 - v
    Epoch 2/50
    934/934 [==============================] - 0s 208us/step - loss: 0.6559 - acc: 0.7730 - val_loss: 1.9330 -
    Epoch 3/50
    934/934 [==============================] - 0s 205us/step - loss: 0.4934 - acc: 0.8308 - val_loss: 0.7510 -
    Epoch 4/50
    934/934 [==============================] - 0s 217us/step - loss: 0.3533 - acc: 0.8715 - val_loss: 1.1359 -
    Epoch 5/50
    934/934 [==============================] - 0s 214us/step - loss: 0.3234 - acc: 0.8876 - val_loss: 0.5248 -
    Epoch 6/50
    934/934 [==============================] - 0s 204us/step - loss: 0.2475 - acc: 0.9218 - val_loss: 0.3934 -
    Epoch 7/50
    934/934 [==============================] - 0s 207us/step - loss: 0.2426 - acc: 0.9240 - val_loss: 0.5355 -
    Epoch 8/50
    934/934 [==============================] - 0s 203us/step - loss: 0.1825 - acc: 0.9358 - val_loss: 1.1052 -
    Epoch 9/50
    934/934 [==============================] - 0s 219us/step - loss: 0.2123 - acc: 0.9283 - val_loss: 0.4806 -
```

```
Epoch 44/50
934/934 [==============================] - 0s 203us/step - loss: 0.0295 - acc: 0.9893 - val_loss: 0.5396
Epoch 45/50
934/934 [==============================] - 0s 200us/step - loss: 0.0552 - acc: 0.9839 - val_loss: 0.5433
Epoch 46/50
934/934 [==============================] - 0s 204us/step - loss: 0.0415 - acc: 0.9818 - val_loss: 0.4684
Epoch 47/50
934/934 [==============================] - 0s 214us/step - loss: 0.0491 - acc: 0.9829 - val_loss: 0.4127
Epoch 48/50
934/934 [==============================] - 0s 210us/step - loss: 0.0336 - acc: 0.9904 - val_loss: 0.3987
Epoch 49/50
934/934 [==============================] - 0s 202us/step - loss: 0.0320 - acc: 0.9882 - val_loss: 0.4629
Epoch 50/50
934/934 [==============================] - 0s 202us/step - loss: 0.0397 - acc: 0.9893 - val_loss: 0.3679
```



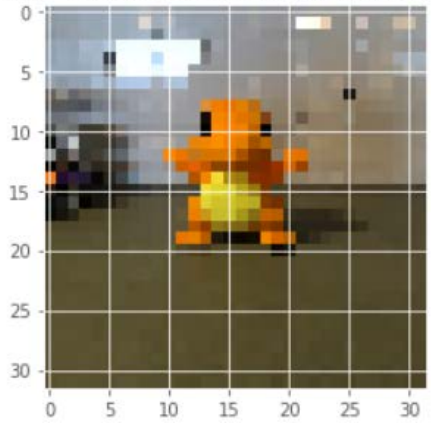
Predicción: charmander - Confianza: 100.00% - Ground Truth: CIFAR10





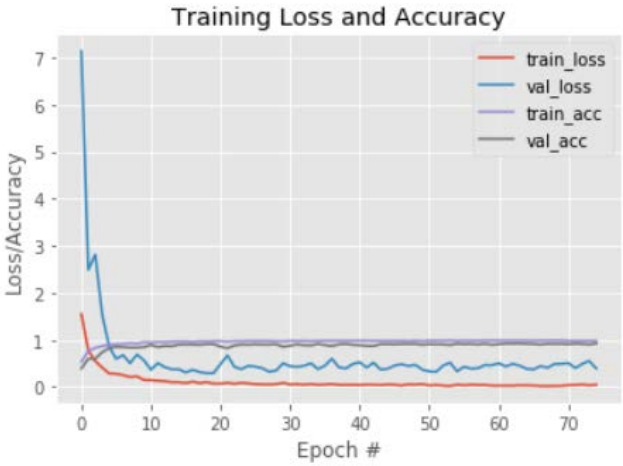

```
[INFO]: Clasificando imagen...
[[2.6342163e-06 1.7219727e-09 9.9999714e-01 1.3740986e-07 8.7149488e-08]]
```

Predicción: mewtwo - Confianza: 100.00% - Ground Truth: charmander

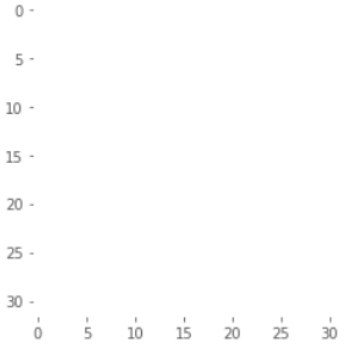


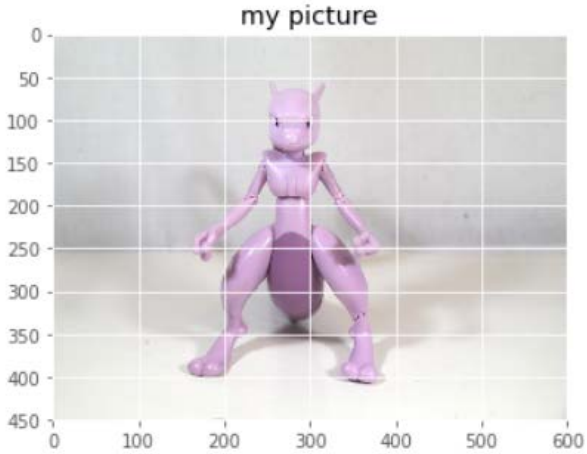
Para Épocas=75

```
1.13.1
['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Train on 934 samples, validate on 234 samples
Epoch 1/75
934/934 [==============================] - 2s 3ms/step - loss: 1.5247 - acc: 0.5257 - val_loss: 2.2989 - va
Epoch 2/75
934/934 [==============================] - 0s 245us/step - loss: 0.7141 - acc: 0.7859 - val_loss: 3.5667 -
Epoch 3/75
934/934 [==============================] - 0s 209us/step - loss: 0.5058 - acc: 0.8405 - val_loss: 3.3825 -
Epoch 4/75
934/934 [==============================] - 0s 206us/step - loss: 0.3875 - acc: 0.8758 - val_loss: 1.3444 -
Epoch 5/75
934/934 [==============================] - 0s 204us/step - loss: 0.3391 - acc: 0.8919 - val_loss: 0.6061 -
Epoch 6/75
934/934 [==============================] - 0s 206us/step - loss: 0.2514 - acc: 0.9111 - val_loss: 0.7131 -
Epoch 7/75
934/934 [==============================] - 0s 217us/step - loss: 0.2224 - acc: 0.9272 - val_loss: 0.7363 -
Epoch 8/75
934/934 [==============================] - 0s 212us/step - loss: 0.2370 - acc: 0.9154 - val_loss: 0.4817 -

Epoch 70/75
934/934 [==============================] - 0s 214us/step - loss: 0.0263 - acc: 0.9904 - val_loss: 0.4943 -
Epoch 71/75
934/934 [==============================] - 0s 212us/step - loss: 0.0417 - acc: 0.9797 - val_loss: 0.5089 -
Epoch 72/75
934/934 [==============================] - 0s 229us/step - loss: 0.0413 - acc: 0.9829 - val_loss: 0.4029 -
Epoch 73/75
934/934 [==============================] - 0s 208us/step - loss: 0.0545 - acc: 0.9807 - val_loss: 0.4945 -
Epoch 74/75
934/934 [==============================] - 0s 212us/step - loss: 0.0369 - acc: 0.9829 - val_loss: 0.5536 -
Epoch 75/75
934/934 [==============================] - 0s 208us/step - loss: 0.0499 - acc: 0.9807 - val_loss: 0.3930 -
```



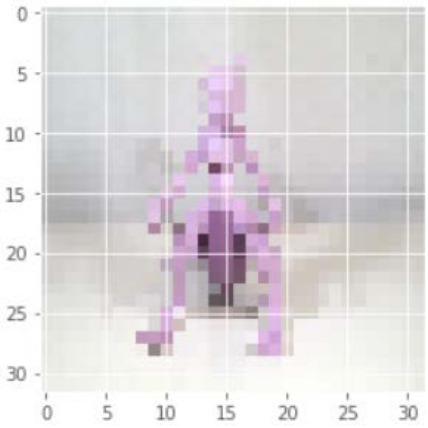
Predicción: mewtwo - Confianza: 100.00% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[1.0005103e-09 9.9278179e-05 9.9990070e-01 6.3379346e-09 2.1136623e-08]]
```

Predicción: mewtwo - Confianza: 99.99% - Ground Truth: mewtwo

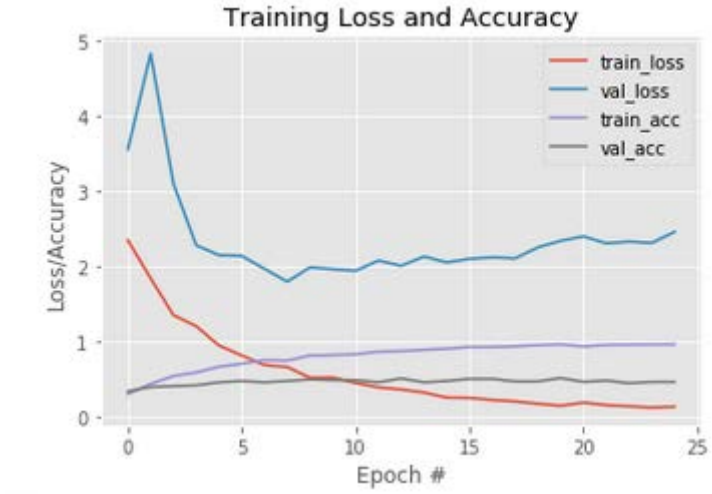


8.1.2- Caso de imágenes de animales, clasificación de razas de perros:

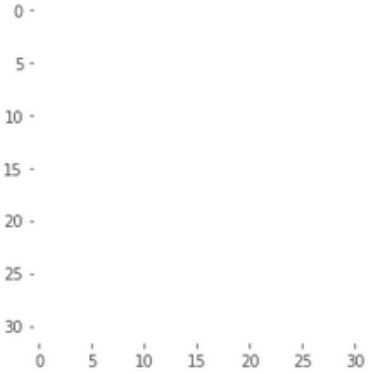
Para CPU

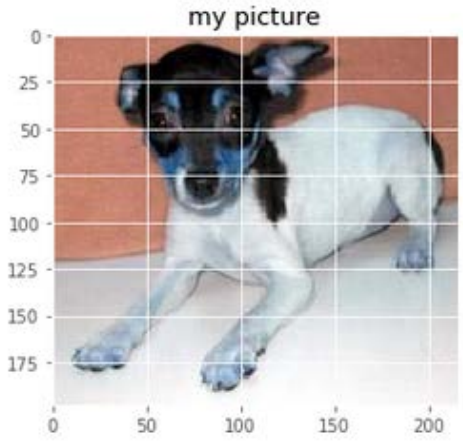
Para Épocas = 25

```
Epoch 1/25
700/700 [==============================] - 10s 14ms/step - loss: 2.3435 - acc: 0.2929 - val_loss: 3.5540 - v
al_acc: 0.3314
Epoch 2/25
700/700 [==============================] - 8s 11ms/step - loss: 1.8370 - acc: 0.4314 - val_loss: 4.8300 - va
l_acc: 0.3886
Epoch 3/25
700/700 [==============================] - 8s 11ms/step - loss: 1.3470 - acc: 0.5343 - val_loss: 3.0988 - va
l_acc: 0.4000
Epoch 4/25
700/700 [==============================] - 8s 11ms/step - loss: 1.1992 - acc: 0.5829 - val_loss: 2.2766 - va
_
Epoch 23/25
700/700 [==============================] - 8s 11ms/step - loss: 0.1318 - acc: 0.9529 - val_loss: 2.3252 - va
l_acc: 0.4400
Epoch 24/25
700/700 [==============================] - 8s 12ms/step - loss: 0.1160 - acc: 0.9557 - val_loss: 2.3073 - va
l_acc: 0.4571
Epoch 25/25
700/700 [==============================] - 8s 11ms/step - loss: 0.1265 - acc: 0.9571 - val_loss: 2.4546 - va
l_acc: 0.4571
```

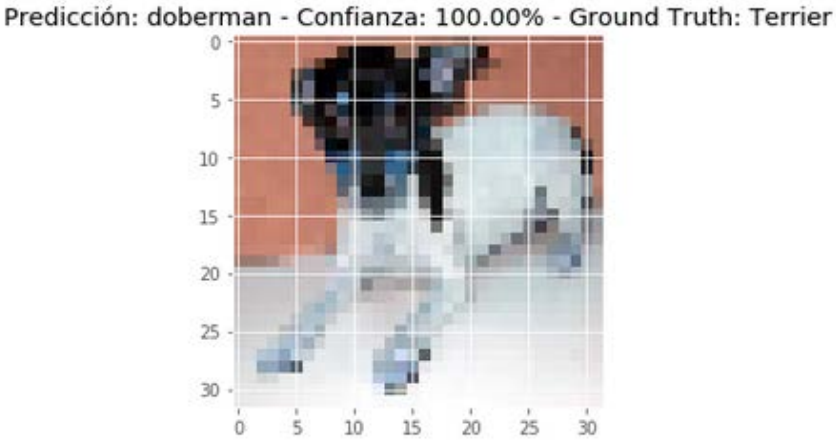


Predicción: terrier - Confianza: 62.35% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[1.1341415e-06 9.9999881e-01 5.7149109e-08 3.3261057e-11 3.6963249e-12]]
```

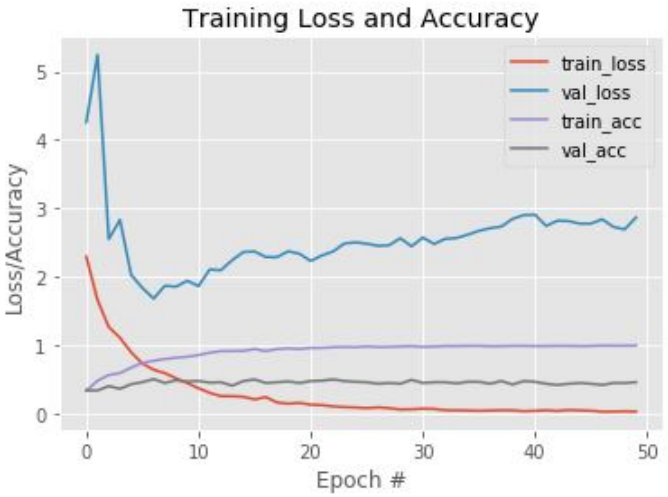


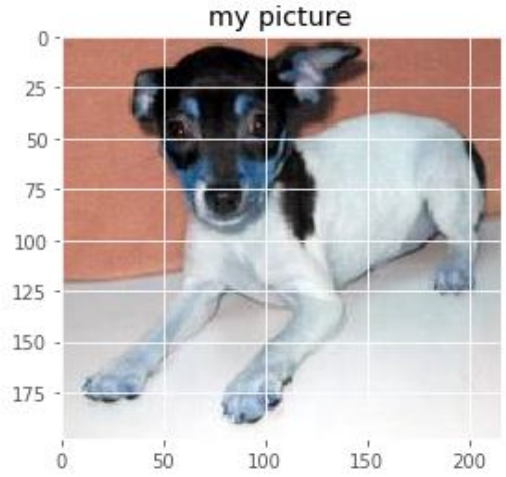
Para 50 épocas

```
Train on 700 samples, validate on 175 samples
Epoch 1/50
700/700 [==============================] - 8s 12ms/step - loss: 2.2924 - acc: 0.3329 - val_loss: 4.2593 - val_acc: 0.3371
Epoch 2/50
700/700 [==============================] - 8s 11ms/step - loss: 1.6594 - acc: 0.4786 - val_loss: 5.2461 - val_acc: 0.3371
Epoch 3/50
700/700 [==============================] - 8s 11ms/step - loss: 1.2632 - acc: 0.5614 - val_loss: 2.5503 - val_acc: 0.4000
Epoch 4/50
700/700 [==============================] - 7s 11ms/step - loss: 1.1066 - acc: 0.5900 - val_loss: 2.8314 - val_acc: 0.3600
Epoch 5/50
700/700 [==============================] - 8s 11ms/step - loss: 0.8946 - acc: 0.6700 - val_loss: 2.0262 - val_acc: 0.4286
Epoch 6/50
700/700 [==============================] - 8s 11ms/step - loss: 0.7285 - acc: 0.7400 - val_loss: 1.8361 - val_acc: 0.4571
Epoch 7/50
```

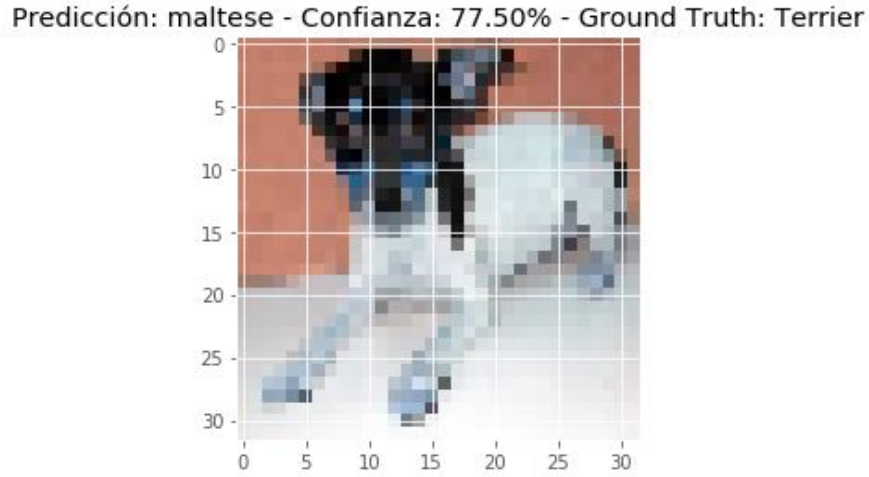
```
Epoch 43/50
700/700 [==============================] - 7s 10ms/step - loss: 0.0366 - acc: 0.9886 - val_loss: 2.8194 - val_acc: 0.4171
Epoch 44/50
700/700 [==============================] - 8s 11ms/step - loss: 0.0509 - acc: 0.9886 - val_loss: 2.8128 - val_acc: 0.4343
Epoch 45/50
700/700 [==============================] - 8s 11ms/step - loss: 0.0445 - acc: 0.9843 - val_loss: 2.7761 - val_acc: 0.4457
Epoch 46/50
700/700 [==============================] - 7s 11ms/step - loss: 0.0395 - acc: 0.9871 - val_loss: 2.7765 - val_acc: 0.4343
Epoch 47/50
700/700 [==============================] - 7s 11ms/step - loss: 0.0254 - acc: 0.9929 - val_loss: 2.8398 - val_acc: 0.4171
Epoch 48/50
700/700 [==============================] - 7s 11ms/step - loss: 0.0269 - acc: 0.9914 - val_loss: 2.7276 - val_acc: 0.4457
Epoch 49/50
700/700 [==============================] - 7s 11ms/step - loss: 0.0310 - acc: 0.9914 - val_loss: 2.6918 - val_acc: 0.4457
Epoch 50/50
700/700 [==============================] - 7s 11ms/step - loss: 0.0278 - acc: 0.9929 - val_loss: 2.8653 - val_acc: 0.4571
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

    chihuahua       0.34      0.29      0.32        34
     doberman       0.33      0.52      0.41        27
      maltese       0.58      0.61      0.60        51
     pekinese       0.50      0.31      0.38        26
      terrier       0.49      0.46      0.47        37

     accuracy                           0.46       175
    macro avg       0.45      0.44      0.43       175
 weighted avg       0.47      0.46      0.45       175
```

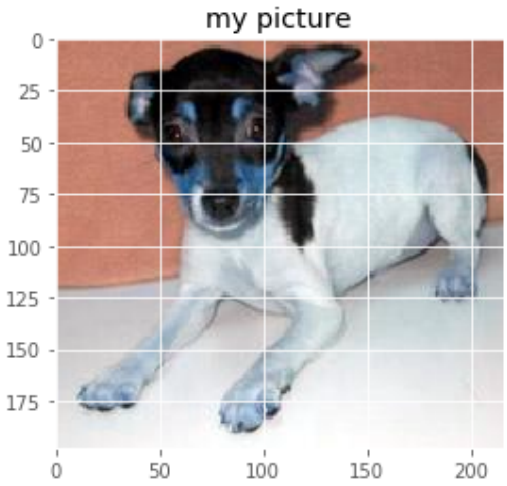




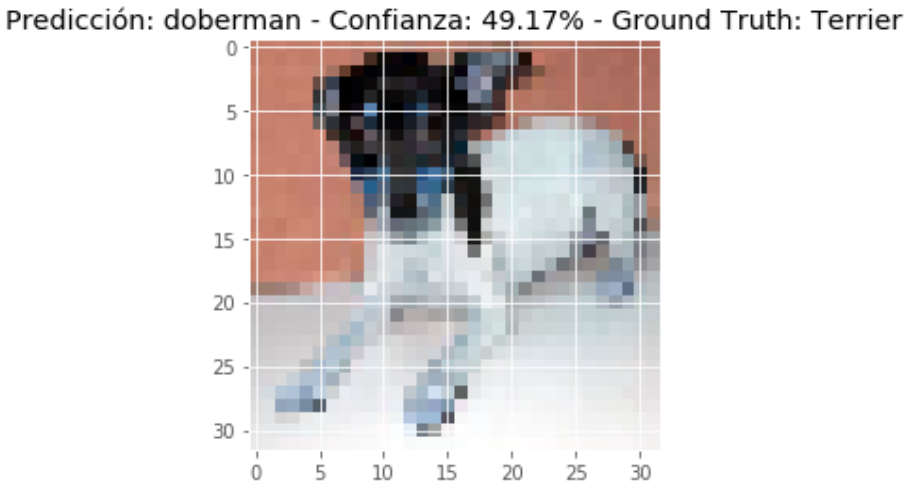

```
[INFO]: Clasificando imagen...
[[1.9608692e-06 2.2489896e-01 7.7503359e-01 6.0384082e-05 5.0424187e-06]]
```



Para Batch Size = 100.




```
[INFO]: Clasificando imagen...
[[6.1060724e-05 4.9168223e-01 4.3077058e-01 7.2517462e-02 4.9687019e-03]]
```



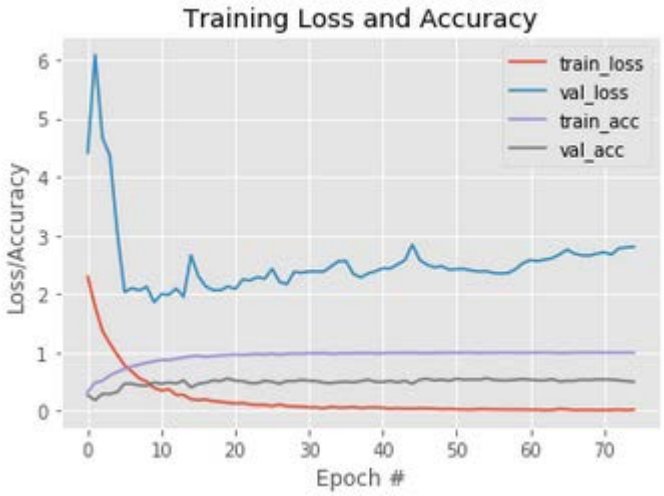
Aumenta el tiempo de computación y disminuye la confianza del 77.5% a menos del 50% para el mismo resultado predictivo.

Para Épocas=75

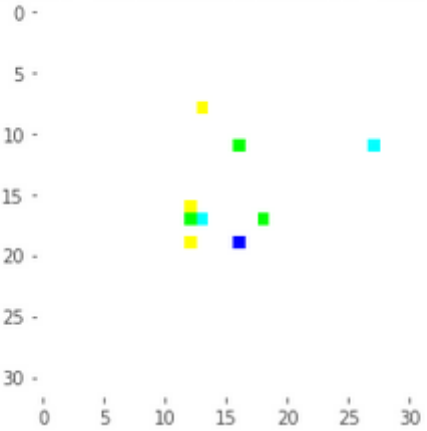
```
Train on 700 samples, validate on 175 samples
Epoch 1/75
700/700 [==============================] - 9s 14ms/step - loss: 2.2924 - acc: 0.3171 - val_loss: 4.4103 - val_acc: 0.2629
Epoch 2/75
700/700 [==============================] - 8s 12ms/step - loss: 1.7855 - acc: 0.4814 - val_loss: 6.0885 - val_acc: 0.1829
Epoch 3/75
700/700 [==============================] - 8s 12ms/step - loss: 1.3699 - acc: 0.5129 - val_loss: 4.6669 - val_acc: 0.2914
Epoch 4/75
700/700 [==============================] - 8s 11ms/step - loss: 1.1500 - acc: 0.6043 - val_loss: 4.3667 - val_acc: 0.2857
Epoch 5/75
700/700 [==============================] - 8s 12ms/step - loss: 0.9599 - acc: 0.6629 - val_loss: 3.0654 - val_acc: 0.3200
Epoch 6/75
700/700 [==============================] - 8s 11ms/step - loss: 0.7781 - acc: 0.7186 - val_loss: 2.0292 - val_acc: 0.4629
Epoch 7/75
700/700 [==============================] - 8s 11ms/step - loss: 0.6708 - acc: 0.7586 - val_loss: 2.0977 - val_acc: 0.4629
Epoch 8/75
700/700 [==============================] - 8s 11ms/step - loss: 0.5534 - acc: 0.7900 - val_loss: 2.0591 - val_acc: 0.4343
Epoch 9/75
700/700 [==============================] - 8s 11ms/step - loss: 0.5020 - acc: 0.8214 - val_loss: 2.1226 - val_acc: 0.4286
```

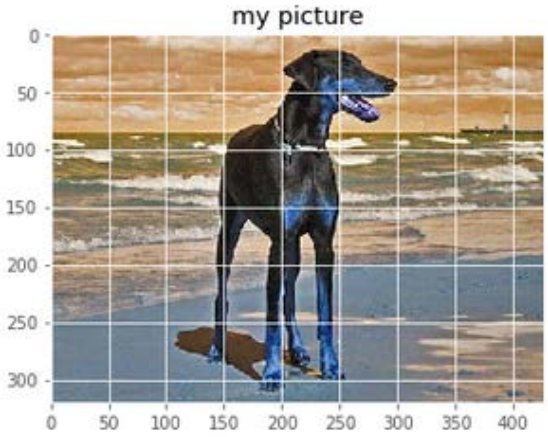
```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

   chihuahua       0.33      0.29      0.31        34
    doberman       0.48      0.44      0.46        27
     maltese       0.69      0.65      0.67        51
    pekinese       0.38      0.54      0.44        26
     terrier       0.63      0.59      0.61        37

    accuracy                           0.52       175
   macro avg       0.50      0.50      0.50       175
weighted avg       0.53      0.52      0.52       175
```

```
[INFO]: Evaluando el modelo...
              precision    recall  f1-score   support

   chihuahua       0.36      0.29      0.32        34
    doberman       0.37      0.56      0.44        27
     maltese       0.69      0.57      0.62        51
    pekinese       0.52      0.42      0.47        26
     terrier       0.51      0.59      0.55        37

    accuracy                           0.50       175
   macro avg       0.49      0.49      0.48       175
weighted avg       0.51      0.50      0.50       175
```

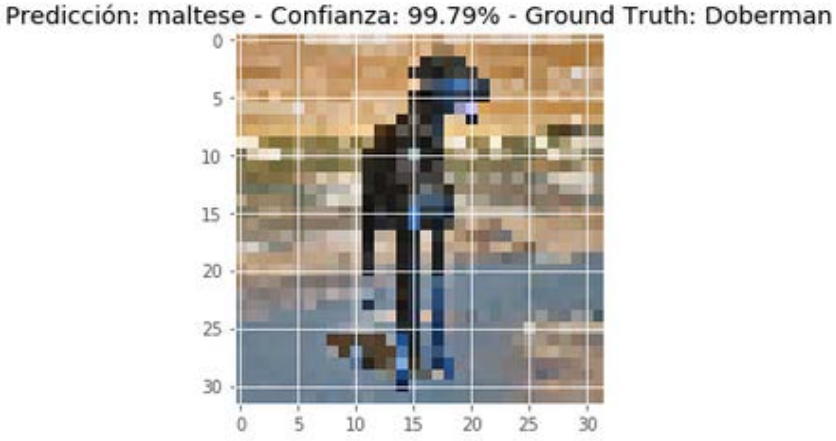


Predicción: doberman - Confianza: 100.00% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[6.0870481e-04 1.4599995e-03 9.9792224e-01 1.7645854e-06 7.2787038e-06]]
```

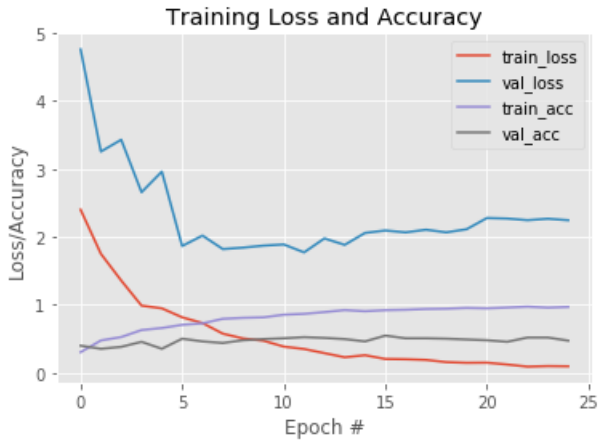


Para con GPU

Para Épocas = 25

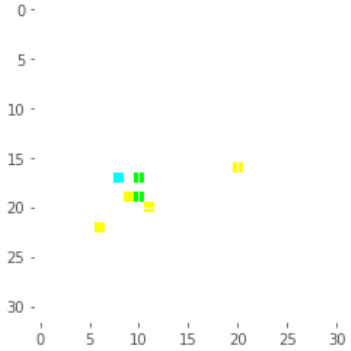
```
1.13.1
['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
[INFO]: Compilando el modelo...
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (from te
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py:3445: calling dropout (from tensorflo
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_prob`.
[INFO]: Entrenando la red...
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflow.python.
Instructions for updating:
Use tf.cast instead.
Train on 700 samples, validate on 175 samples
Epoch 1/25
700/700 [==============================] - 4s 6ms/step - loss: 2.3113 - acc: 0.3171 - val_loss: 6.3494 - val_acc: 0.2857
Epoch 2/25
700/700 [==============================] - 0s 217us/step - loss: 1.7252 - acc: 0.4714 - val_loss: 6.2243 - val_acc: 0.2800
Epoch 3/25
700/700 [==============================] - 0s 193us/step - loss: 1.3357 - acc: 0.5143 - val_loss: 3.2027 - val_acc: 0.3829
Epoch 4/25
700/700 [==============================] - 0s 196us/step - loss: 1.0515 - acc: 0.6200 - val_loss: 2.5523 - val_acc: 0.4171
Epoch 5/25
700/700 [==============================] - 0s 202us/step - loss: 0.9928 - acc: 0.6586 - val_loss: 2.1869 - val_acc: 0.4857
Epoch 6/25
```

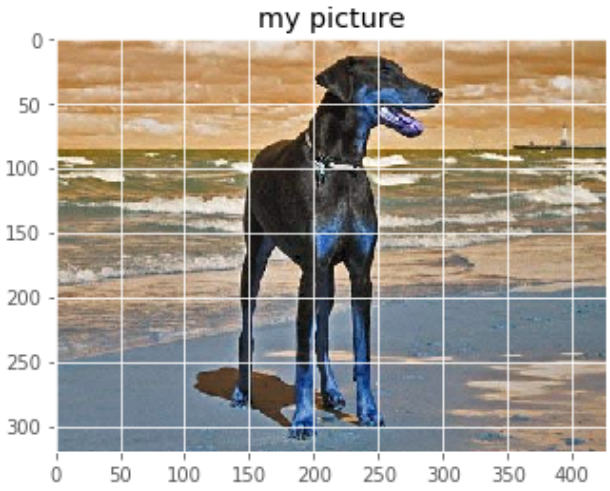
	precision	recall	f1-score	support
chihuahua	0.32	0.29	0.31	34
doberman	0.50	0.37	0.43	27
maltese	0.60	0.57	0.59	51
pekinese	0.44	0.31	0.36	26
terrier	0.45	0.70	0.55	37
accuracy			0.47	175
macro avg	0.46	0.45	0.45	175
weighted avg	0.48	0.47	0.47	175




```
[INFO]: Clasificando imagen...
[[0.07098605 0.6329108  0.01084844 0.00858925 0.27666548]]
```

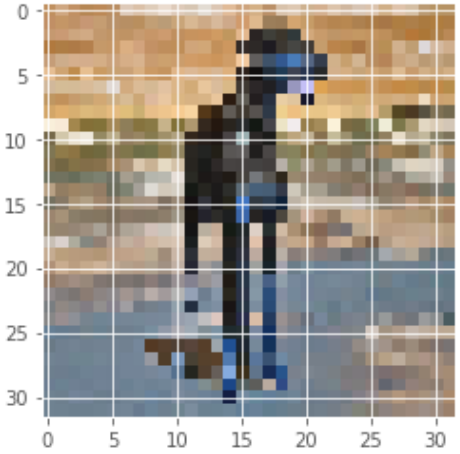
Predicción: doberman - Confianza: 63.29% - Ground Truth: CIFAR10





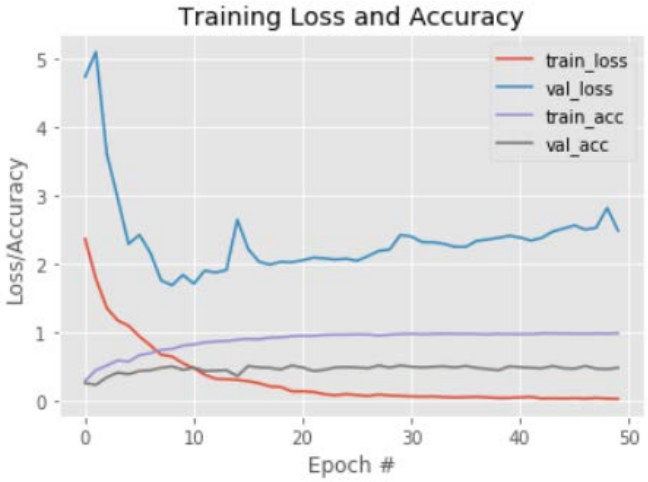

```
[INFO]: Clasificando imagen...
[[5.0971145e-04 9.9939239e-01 8.0785312e-06 7.5194803e-05 1.4607822e-05]]
```

Predicción: doberman - Confianza: 99.94% - Ground Truth: Doberman

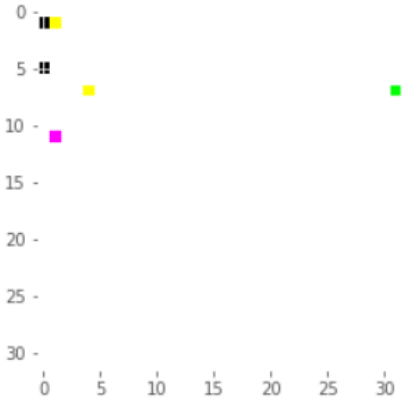


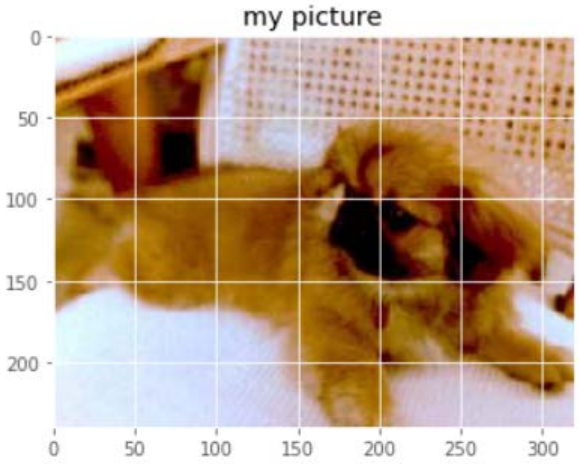
Para Épocas = 50

```
1.13.1
['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Train on 700 samples, validate on 175 samples
Epoch 1/50
700/700 [==============================] - 2s 3ms/step - loss: 2.3670 - acc: 0.2943 - val_loss: 4.7305 - val
Epoch 2/50
700/700 [==============================] - 0s 256us/step - loss: 1.7851 - acc: 0.4514 - val_loss: 5.0938 - v
Epoch 3/50
700/700 [==============================] - 0s 221us/step - loss: 1.3593 - acc: 0.5200 - val_loss: 3.6082 - v
Epoch 4/50
700/700 [==============================] - 0s 216us/step - loss: 1.1776 - acc: 0.5929 - val_loss: 2.9623 - v
Epoch 5/50
700/700 [==============================] - 0s 204us/step - loss: 1.1009 - acc: 0.5771 - val_loss: 2.2929 - v
Epoch 6/50
```



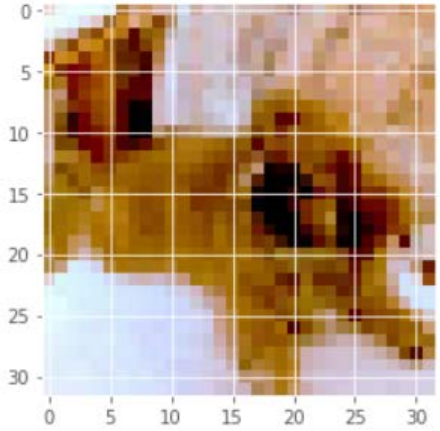
Predicción: pekinese - Confianza: 89.17% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[1.0594860e-02 9.8921984e-01 7.6934002e-06 1.9353565e-09 1.7757964e-04]]
```

Predicción: doberman - Confianza: 98.92% - Ground Truth: Pekinese

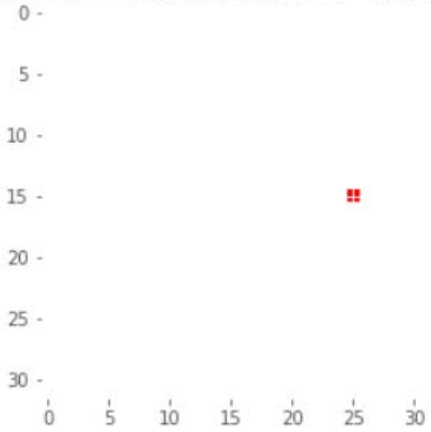


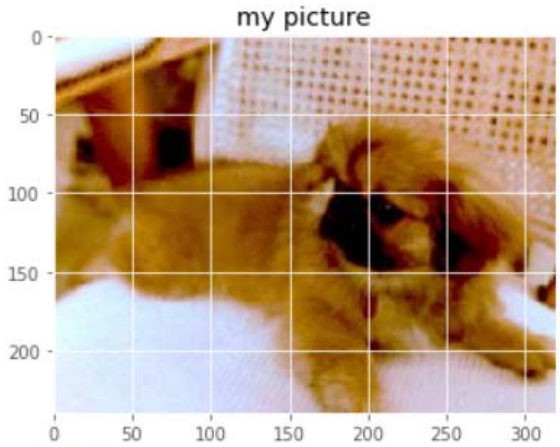
Para Épocas = 75

```
1.13.1
['/device:CPU:0', '/device:XLA_CPU:0', '/device:XLA_GPU:0', '/device:GPU:0']
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Train on 700 samples, validate on 175 samples
Epoch 1/75
700/700 [==============================] - 3s 4ms/step - loss: 2.1945 - acc: 0.3543 - val_loss: 3.5873 - va
Epoch 2/75
700/700 [==============================] - 0s 243us/step - loss: 1.8025 - acc: 0.4600 - val_loss: 3.8948 -
Epoch 3/75
700/700 [==============================] - 0s 229us/step - loss: 1.3842 - acc: 0.5314 - val_loss: 4.0699 -
Epoch 4/75
700/700 [==============================] - 0s 208us/step - loss: 1.1591 - acc: 0.5814 - val_loss: 2.8449 -
Epoch 5/75
700/700 [==============================] - 0s 220us/step - loss: 0.9950 - acc: 0.6114 - val_loss: 2.2688 -
Epoch 6/75
700/700 [==============================] - 0s 203us/step - loss: 0.9157 - acc: 0.6629 - val_loss: 2.2063 -
Epoch 7/75
700/700 [==============================] - 0s 206us/step - loss: 0.7439 - acc: 0.7343 - val_loss: 1.7556 -
Epoch 8/75
700/700 [==============================] - 0s 222us/step - loss: 0.7072 - acc: 0.7514 - val_loss: 2.1426 -
Epoch 9/75
700/700 [==============================] - 0s 203us/step - loss: 0.5693 - acc: 0.7829 - val_loss: 2.0929 -
```

```
Epoch 70/75
700/700 [==============================] - 0s 205us/step - loss: 0.0166 - acc: 0.9943 - val_loss: 2.7236 -
Epoch 71/75
700/700 [==============================] - 0s 203us/step - loss: 0.0286 - acc: 0.9886 - val_loss: 2.7641 -
Epoch 72/75
700/700 [==============================] - 0s 214us/step - loss: 0.0204 - acc: 0.9929 - val_loss: 2.6340 -
Epoch 73/75
700/700 [==============================] - 0s 213us/step - loss: 0.0154 - acc: 0.9943 - val_loss: 2.5457 -
Epoch 74/75
700/700 [==============================] - 0s 220us/step - loss: 0.0176 - acc: 0.9929 - val_loss: 2.6215 -
Epoch 75/75
700/700 [==============================] - 0s 205us/step - loss: 0.0209 - acc: 0.9929 - val_loss: 2.7154 -
```

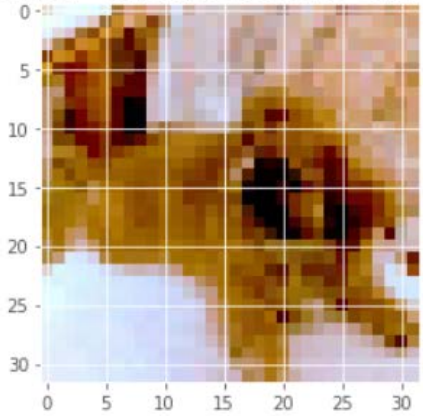
Predicción: pekinese - Confianza: 98.17% - Ground Truth: CIFAR10






```
[INFO]: Clasificando imagen...
[[5.1525305e-04 8.0009109e-01 2.3065525e-07 1.9188234e-01 7.5109974e-03]]
```

Predicción: doberman - Confianza: 80.01% - Ground Truth: Pekinese



8.3.- Evaluación de resultados.

A continuación ya podemos establecer un cuadro comparativo con la evaluación de los distintos resultados obtenidos:

Destacar el observar la diferencia de tiempos de ejecución (a observar en las tablas de ejecución de épocas adjuntas para cada una de las evaluaciones) de CPU respecto de GPU en el que, tanto para el caso de las imágenes de Pokemons como de animales, para el caso de CPU ha estado en el entorno de los 10 a 12 segundos por época aumentando en orden del número de épocas con respecto a los tiempos de GPU en el que salvo la primera época en el que ha tardado 3 segundos de media, el resto ha ocupado decenas de microsegundos (aprox. 200 microsegundos por época como máximo) para cada una de los distintos números de épocas. Viendo aquí los resultados prácticamente instantáneos obtenidos en el caso del uso de las GPU.

Valores referentes de tiempos los vemos en los siguientes cálculos:

Tiempo medio de ejecución de cada conjunto de épocas, tomando como referencia media 50 épocas:

- Para CPU: 50 épocas x 9 segundos/época = 450 segundos = 7.5 minutos.
- Para GPU: 3 segundos la primera época + 49 épocas x 0.00025 segundos = 3, 0125 segundos.

Aquí podemos apreciar la potencia de cálculo de las GPUs en el que difiere de una media de 7 minutos frente a los en cuanto apenas 3 segundos de las GPU.

Para el caso de imágenes Pokemons obtenemos la siguiente tabla:

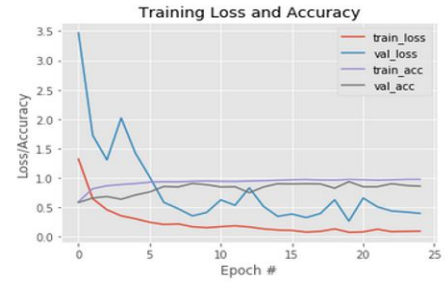
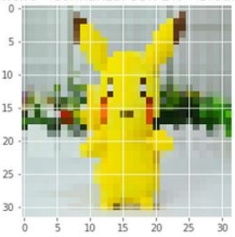
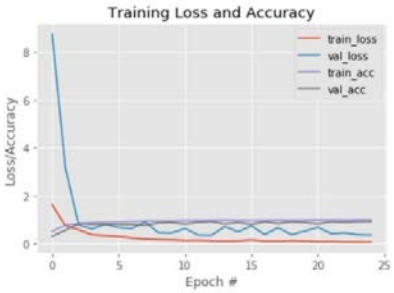
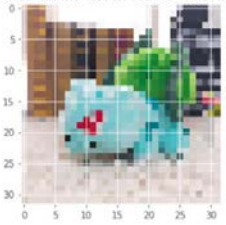
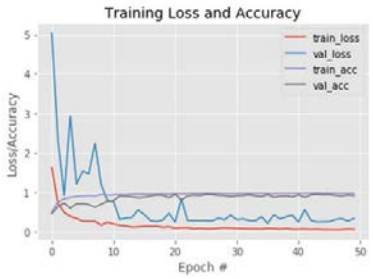
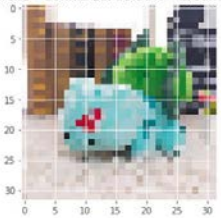
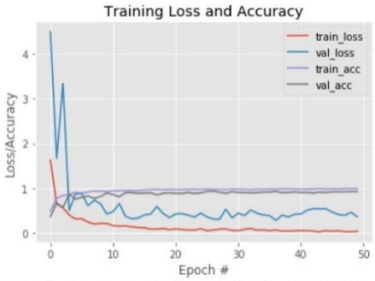
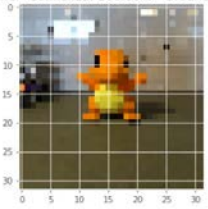
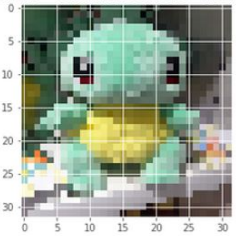
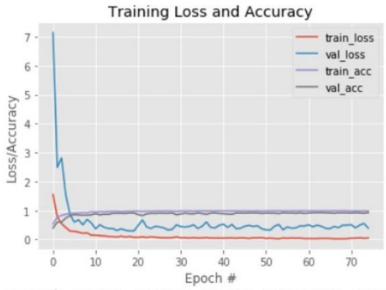
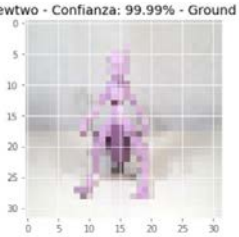
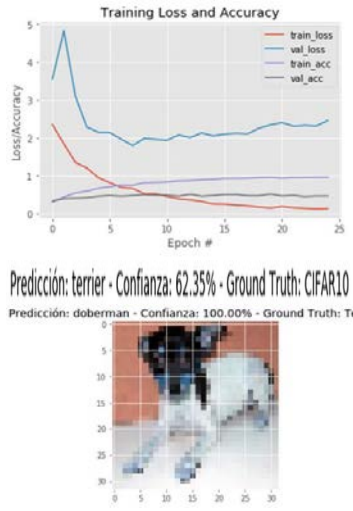
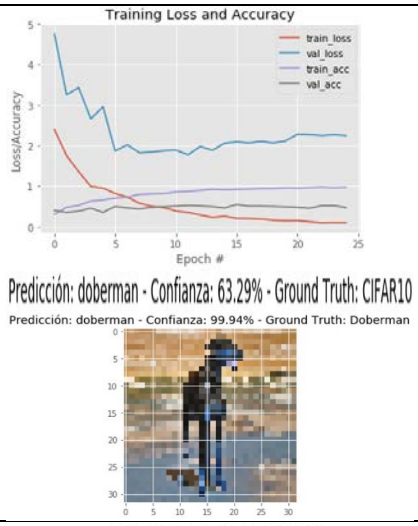
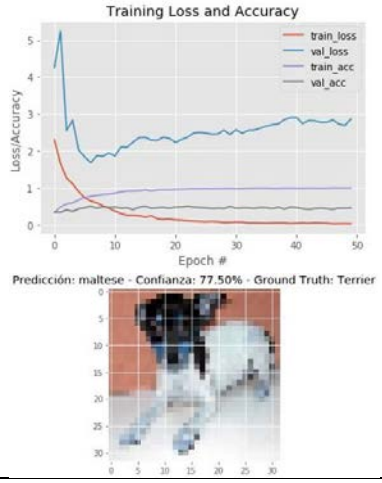
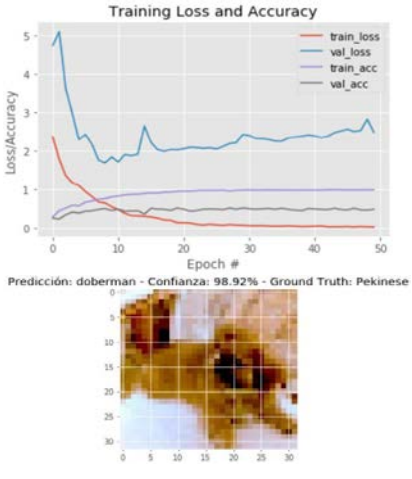
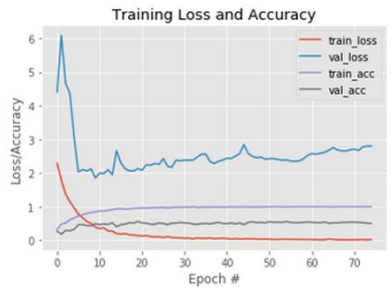
Nº de Épocas	CPU	GPU
25	 Predicción: pikachu - Confianza: 56.14% - Ground Truth: CIFAR10 Predicción: mewtwo - Confianza: 99.71% - Ground Truth: pikachu 	 Predicción: bulbasaur - Confianza: 99.99% - Ground Truth: CIFAR10 Predicción: mewtwo - Confianza: 100.00% - Ground Truth: bulbasaur 
50	 Predicción: bulbasaur - Confianza: 99.99% - Ground Truth: CIFAR10 Predicción: mewtwo - Confianza: 99.96% - Ground Truth: bulbasaur 	 Predicción: charmander - Confianza: 100.00% - Ground Truth: CIFAR10 Predicción: mewtwo - Confianza: 100.00% - Ground Truth: charmander 
75	 Predicción: squirtle - Confianza: 100.00% - Ground Truth: CIFAR10 Predicción: mewtwo - Confianza: 97.41% - Ground Truth: squirtle 	 Predicción: mewtwo - Confianza: 100.00% - Ground Truth: CIFAR10 Predicción: mewtwo - Confianza: 99.99% - Ground Truth: mewtwo 

Tabla 3: comparativa de imágenes estáticas. Pokemons.

En la tabla comparativa anterior observamos como los resultados han sido mucho mejores, más precisos y confiables, por ejemplo valores de gráficas con menos picos para el caso de las GPU llegando a confianzas del 100% para tan sólo con 25 épocas, pasando del 56% para el caso de CPU al 99.99% para el caso de la GPU.

En el caso de animales como veremos a continuación, no surte efecto el hecho de aumentar el bacth size de 64 a 100 en el que aumenta el tiempo de computación (sobre todo para el caso de CPU) y con resultado de disminución de la confianza resultando para nuestro caso de poco práctico si bien óptimo.

Para el caso de imágenes de clasificación de razas de animales obtenemos los siguientes resultados:

Nº de Épocas	**CPU**	**GPU**
25		
50		
75		

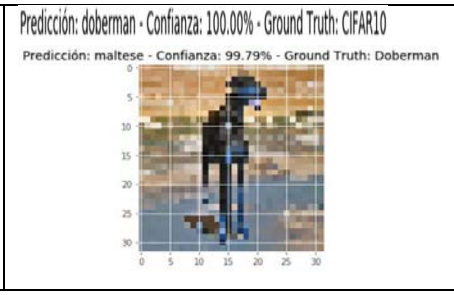
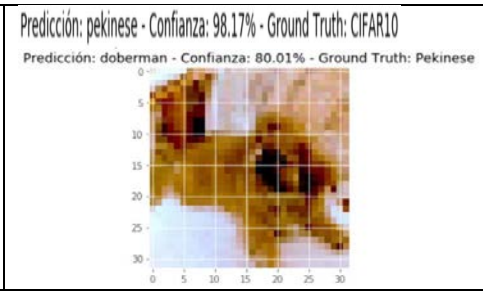
		

Tabla 4: comparativa de imágenes estáticas. Animales.

Hacer notar en este caso como a pesar de disponer de un número menor de imágenes en este caso, unas 900 frente a las 1200 de los Pokemons, se nota un descenso de la precisión, pero aún así, alcanzando prácticamente el 100% para un número de épocas igual o mayor a 50.

Tal como se he indicado en su sección, la prueba de aumentar en este caso el número de batch size de 64 a 100 para por ejemplo el caso de 50 épocas obtenemos un resultado no óptimo respecto a que aumenta el tiempo de computación y disminuye la confianza del 77.5% a menos del 50% para el mismo resultado predictivo.

Destacar que se han empleado datasets disponibles de evaluación media-baja a efectos de evaluación práctica, ya que para una evaluación precisa, tal cual requieren los sistemas profesionales, se disponen de datasets de más de 60000 imágenes y de distintas clasificaciones, en los que además del número de épocas se disponen de más parámetros a variar y observar como distintas capas y etapas convolucionales que contribuyen a una evaluación más precisa y óptima.

8.4.- Ejemplo práctico 2. Reconocimiento de formas en imagen dinámica, video.

En esta ocasión, con los conocimientos recogidos de [6], [7] y [8] en su parte de reconocimiento facial y de objetos, así como la implementación, como vamos a ver, del algoritmo Haars Cascade, se realiza la programación del algoritmo de reconocimiento facial, en el que se pretende en este caso analizar la misma comparativa anterior y en la que, junto con lo aprendido en el apartado 7 dedicado a la visión artificial y con el uso de la librería Haars Cascade ya comentada en el apartado 7.20 dedica a este objeto de reconocimiento concreto, podemos observar y analizar la respuesta de ambos dispositivos de forma común, CPU y GPU en la que en este caso, vemos que no se aprecia distinción si bien necesidad de uso concreto de GPU.

Como se observa en este caso, gracias a esta librería dedicada la identificación sobre todo de rostros humanos (cara, ojos, boca, etc.) donde ya recoge en un fichero pre-configurado las características necesarias para poder realizar la identificación de la correspondiente forma humana, podemos ver como es posible realizar grandes aplicaciones sin necesidad de uso de GPU.

En esta ocasión, consiste en la realización de establecimiento de un dataset de la imagen humana a identificar, en este caso concreto, la forma facial, en la que a través de la ejecución de una parte del programa, donde se realiza la toma de información y se constituye la carga de información, basada en tomas programables de la imagen facial de referencia, la mía en este caso y de la cual, al algoritmo realizara una tarea similar a la de un aprendizaje y extracción de características. Información que posteriormente utilizará en forma de test basado en la comparación de las características aprendidas con las encontradas a través de mi exposición en la cámara. La coincidencia de estas características, es la que producirá tal identificación y reconocimiento facial.

Códigos realizados:

Para la realización de esta parte de reconocimiento facial se requiere la utilización de tres programas distintos pero complementarios. Siendo estos, en orden de ejecución:

- Dataset.py
- Entrenamiento.py
- Detectar_e_identificar_rostros.py

Código de Dataset.py

En este código se realiza la toma de imágenes de referencia que van a componer el dataset donde el programa va a generar una colección de imágenes base donde poder, posteriormente, en el programa de entrenamiento.py, realizar la extracción de características. Características que van a servir de base para la propia identificación en este caso, facial.

En este programa, la combinación de la librería Open CV junto con la programación en Python, se realiza la toma de muestras (consistente en una base de muestras de la parte facial) programables en número de estas, siendo por ejemplo, 300. Dependiendo de como de precisa se requiere. En este caso, incluso con 200 imágenes ha sido más que suficiente.

Podemos observar también, aparte de la librería Open CV, el uso de librerías de Haar Cascade basadas en ficheros propios de información con referencia .xml.

```
import cv2

web_cam = cv2.VideoCapture(0)

cascPath = "C:/Users/Javier/Desktop/Rec_facial/Cascades/haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cascPath)

count = 0

while(True):
    _, imagen_marco = web_cam.read()

    grises = cv2.cvtColor(imagen_marco, cv2.COLOR_BGR2GRAY)

    rostro = faceCascade.detectMultiScale(grises, 1.5, 5)

    for(x,y,w,h) in rostro:
        cv2.rectangle(imagen_marco, (x,y), (x+w, y+h), (255,0,0), 4)
        count += 1

        cv2.imwrite("images/Javier/Javier_"+str(count)+".jpg", grises[y:y+h, x:x+w])
        cv2.imshow("Creando Dataset", imagen_marco)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

    elif count >= 300:
        break


# Cuando todo está hecho, liberamos la captura
web_cam.release()
cv2.destroyAllWindows()
```

Código de entrenamiento.py

A continuación del código anterior, le sigue la ejecución de este algoritmo donde el programa se dedica a recoger cada una de las imágenes anteriores (recogidas en el anterior programa dataset.py) y generará para cada imagen, una etiqueta de identificación y recogida de características donde generará el fichero (de características) llamado: entrenamiento.yml. Propio del funcionamiento del algoritmo Haars Cascade. Fichero, que utilizará el siguiente código: detección_e_identificar_rostros.py para realizar la auténtica comparación de la imagen capturada por la cámara y las características de nuestro rostro facial.

```
import cv2
import os
import numpy as np
from PIL import Image
import pickle

cascPath = "C:/Users/Javier/Desktop/Rec_facial/Cascades/haarcascade_frontalface_alt2.xml"
faceCascade = cv2.CascadeClassifier(cascPath)

#reconocimiento con opencv
reconocimiento = cv2.face.LBPHFaceRecognizer_create()

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
image_dir = os.path.join(BASE_DIR,"images")


current_id = 0
etiquetas_id = {}
y_etiquetas = []
x_entrenamiento = []

for root, dirs, archivos in os.walk(image_dir):
    for archivo in archivos:
        if archivo.endswith("png") or archivo.endswith("jpg"):
            pathImagen = os.path.join(root,archivo)
            etiqueta = os.path.basename(root).replace(" ", "-")#.lower()
            #print(etiqueta,pathImagen)

            #Creando las etiquetas
            if not etiqueta in etiquetas_id:
                etiquetas_id[etiqueta] = current_id
                current_id += 1
            id_ = etiquetas_id[etiqueta]
            #print(etiquetas_id)

            pil_image = Image.open(pathImagen).convert("L")
            tamanio = (550,550)
            imagenFinal = pil_image.resize(tamanio, Image.ANTIALIAS)
            image_array = np.array(pil_image,"uint8")
            #print(image_array)

            rostros = faceCascade.detectMultiScale(image_array, 1.5, 5)

            for (x,y,w,h) in rostros:
                roi = image_array[y:y+h, x:x+w]
                x_entrenamiento.append(roi)
                y_etiquetas.append(id_)


#print(y_etiquetas)
#print(x_entrenamiento)
with open("labels.pickle",'wb') as f:
    pickle.dump(etiquetas_id, f)

reconocimiento.train(x_entrenamiento, np.array(y_etiquetas))
reconocimiento.save("entrenamiento.yml")
```

Código detectar_e_identificar_rostros.py

En este código y como programa último del proceso de reconocimiento facial, se procede a conjuntar la parte de toma de características (ver inclusión del fichero: entrenamiento.yml) junto con otros de las librerías Haars con extensión .xml donde se pueden incluir varias características adicionales que componen un reconocimiento facial como características de ojos y sonrisa.

El programa consiste en puesta en funcionamiento de la cámara donde se captura la imagen facial de la persona a identificar y posteriormente se implementa el algoritmo de reconocimiento por parte del algoritmo de Haars Cascade.

```
import cv2
import pickle

cascPath = "C:/Users/Javier/Desktop/Rec_facial/Cascades/haarcascade_frontalface_alt2.xml"
faceCascade = cv2.CascadeClassifier(cascPath)

eyeCascade = cv2.CascadeClassifier("C:/Users/Javier/Desktop/Rec_facial/Cascades/haarcascade_eye.xml")
smileCascade = cv2.CascadeClassifier("C:/Users/Javier/Desktop/Rec_facial/Cascades/haarcascade_smile.xml")

reconocimiento = cv2.face.LBPHFaceRecognizer_create()
reconocimiento.read("C:/Users/Javier/Desktop/Rec_facial/entrenamiento.yml")

etiquetas = {"nombre_persona" : 1 }
with open("labels.pickle",'rb') as f:
    pre_etiquetas = pickle.load(f)
    etiquetas = { v:k for k,v in pre_etiquetas.items()}

web_cam = cv2.VideoCapture(0)

while True:
    # Capture el marco
    ret, marco = web_cam.read()
    grises = cv2.cvtColor(marco, cv2.COLOR_BGR2GRAY)
    rostros = faceCascade.detectMultiScale(grises, 1.5, 5)

    # Dibujar un rectángulo alrededor de las rostros
    for (x, y, w, h) in rostros:
        #print(x,y,w,h)
        roi_gray = grises[y:y+h, x:x+w]
        roi_color = marco[y:y+h, x:x+w]

        # reconocimiento
        id_, conf = reconocimiento.predict(roi_gray)
        if conf >= 4  and conf < 85:
            #print(id_)
            #print(etiquetas[id_])
            font = cv2.FONT_HERSHEY_SIMPLEX

            nombre = etiquetas[id_]

            if conf > 50:
                #print(conf)
                nombre = "Desconocido"

            color = (255,255,255)
            grosor = 2
            cv2.putText(marco, nombre, (x,y), font, 1, color, grosor, cv2.LINE_AA)

        img_item = "my-image.png"
        cv2.imwrite(img_item, roi_gray)

        cv2.rectangle(marco, (x, y), (x+w, y+h), (0, 255, 0), 2)

        rasgos = smileCascade.detectMultiScale(roi_gray)
        for(ex,ey,ew,eh) in rasgos:
            cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (0, 255, 0), 2)

    # Display resize del marco
    marco_display = cv2.resize(marco, (1200, 650), interpolation = cv2.INTER_CUBIC)
    cv2.imshow('Detectando Rostros', marco_display)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Cuando todo está hecho, liberamos la captura
web_cam.release()
cv2.destroyAllWindows()
```

En la siguiente figura podemos ver una muestra del dataset de mi imagen. En este caso, se tomaron como referencia 250 imágenes de mi rostro facial. Produciendo en todo caso un gran acierto de forma rápida y precisa, como se puede ver en la figura nº 154.

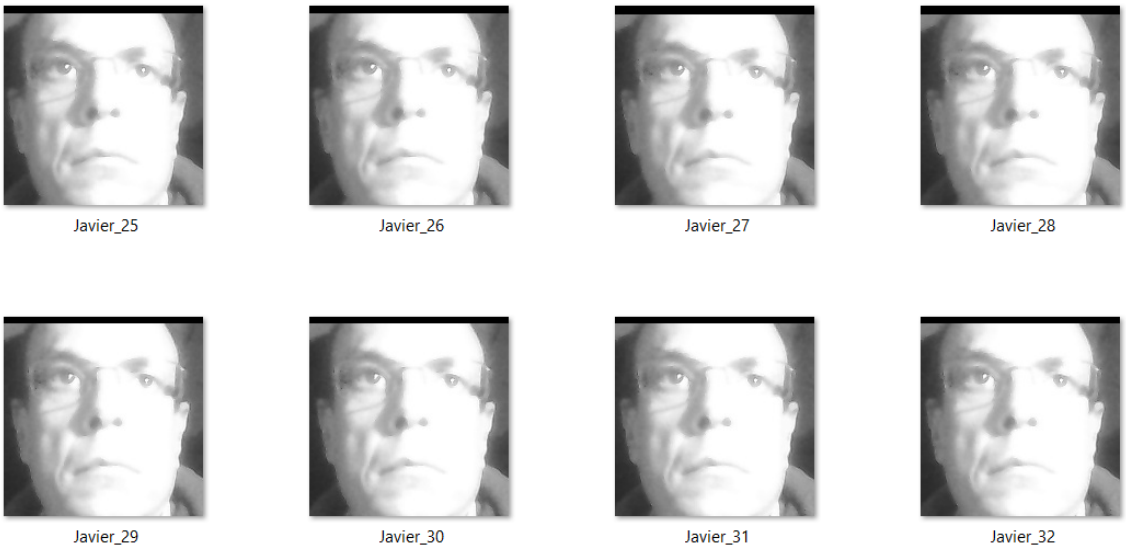


Fig.153: Dataset de imágenes para reconocimiento facial.

A continuación una muestra de la identificación por visión artificial de en este caso un rostro facial.

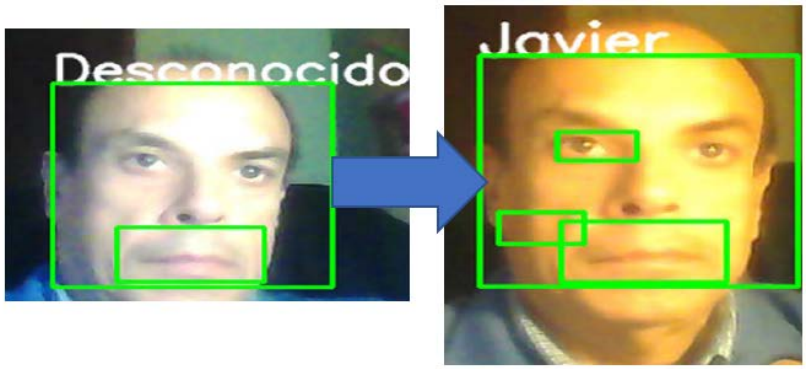


Fig.154: Muestra de ejecución del programa.

En esta ocasión, lo más importante ha sido la adquisición de forma coherente del dataset con hasta 250 imágenes mías. Pudiendo ser programable. Obviamente, dependerá de la aplicación final la necesidad de más o menos muestras. En esta ocasión, ha sido indiferente la necesidad de requerir una GPU. Con lo que vemos, es también factible poder realizar aplicaciones sin la necesidad de gran requerimiento de hardware.

9.- Conclusión y línea futura.

Finalmente realizar el aporte de conclusión a este trabajo documental donde reporto a la vista de principalmente lo que abarcamos, la aplicación de la GPU a los procesos de Deep Learning y Visión e Inteligencia Artificial en el que se ha visto la conjunción de operación conjunta hardware-software y como implementación de la últimas tecnologías actuales en funcionamiento en esta conjunción que cada día está más ligada y nos aporta un mayor y mejor futuro, sobre todo en este apasionante campo y sus aplicativos como Análisis de imágenes y

Visión Artificial principalmente, donde es de destacar la necesidad de empleo de estos procesadores GPU en los que sin ellos no podríamos realizar determinadas evaluaciones y por lo tanto avanzar y progresar en nuestro campo de aplicación tan importante actualmente como el aplicativo al campo de la salud y la medicina y realización de procesos que nosotros como humanos no podemos realizar.

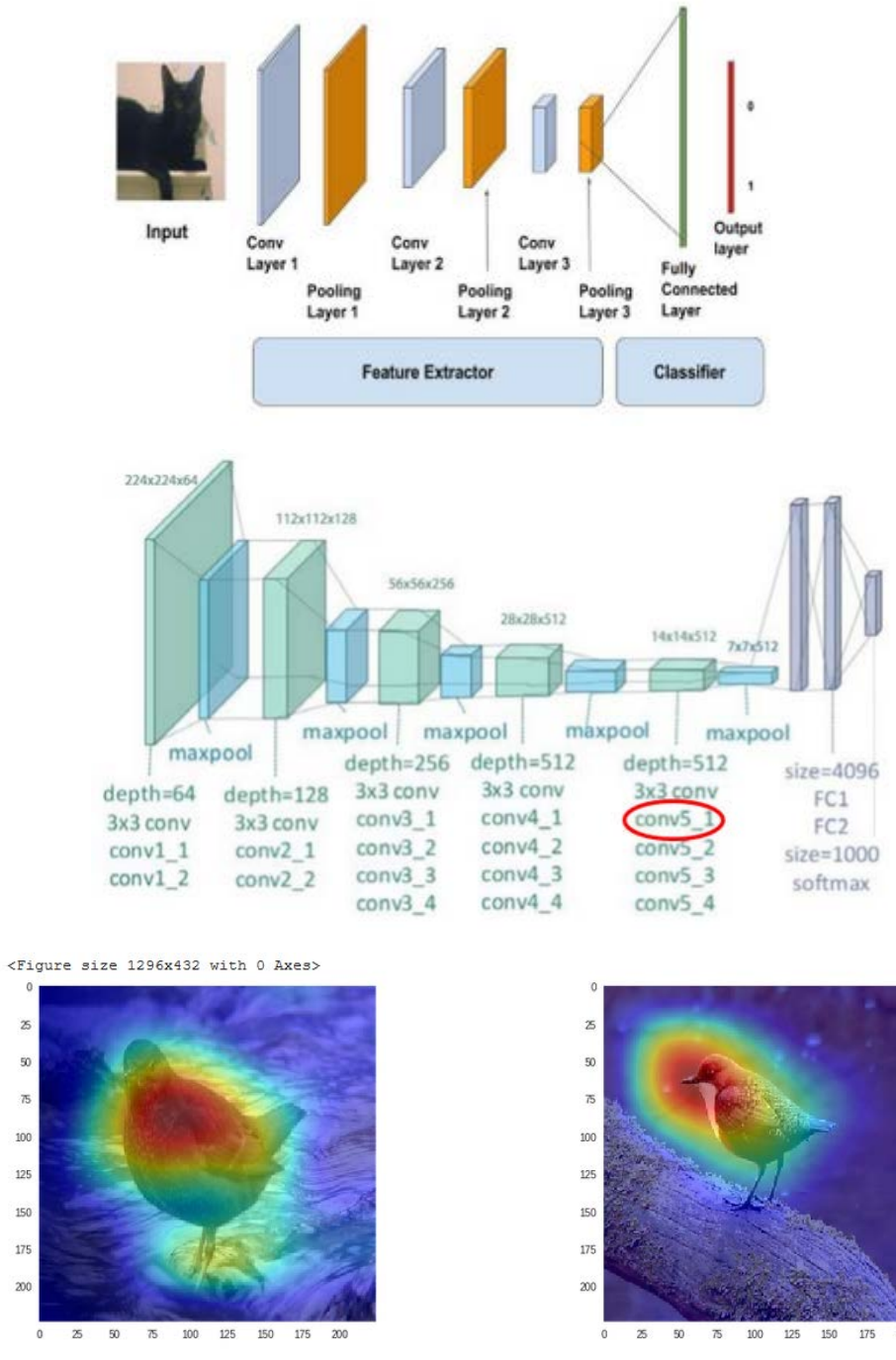


Fig.155: Esquemas operativos donde se requieren GPUs

Es a través de estos conjuntos computacionales como progresamos y avanzamos contando con su incondicional ayuda y prestación y en la que cada día somos más dependientes de estos sistemas.

Líneas actuales y futuras:

Como líneas actuales y futuras, a continuación se reporta uno de los proyectos actuales realizados en mi empresa de trabajo: Amara, donde de proyecta todo un sistema audiovisual de reconocimiento audio-visual. Donde una empresa nos encarga la implementación de un sistema de identificación de movimientos técnicos por parte de un operario con vistas a ser evaluado para su correcta implementación de su proceso de trabajo.

Una línea de implementación de este proyecto aquí realizado sería, tal como hemos visto, la inclusión de estos algoritmos de identificación de, si bien de objetos, si bien de formas, con respecto a poder disponer una mejor implementación y más adecuada a los procesos de análisis y evaluación de realización de trabajos. Tal como nos encarga esta empresa.

Actualmente hay distintos proveedores a nivel nacional, como por ejemplo la empresa valenciana Casmar (www.casmarglobal.com) como proveedora de equipos (por ejemplo, cámaras y procesadores GPU) donde poder incorporar e incluir en este realizado en Amara.

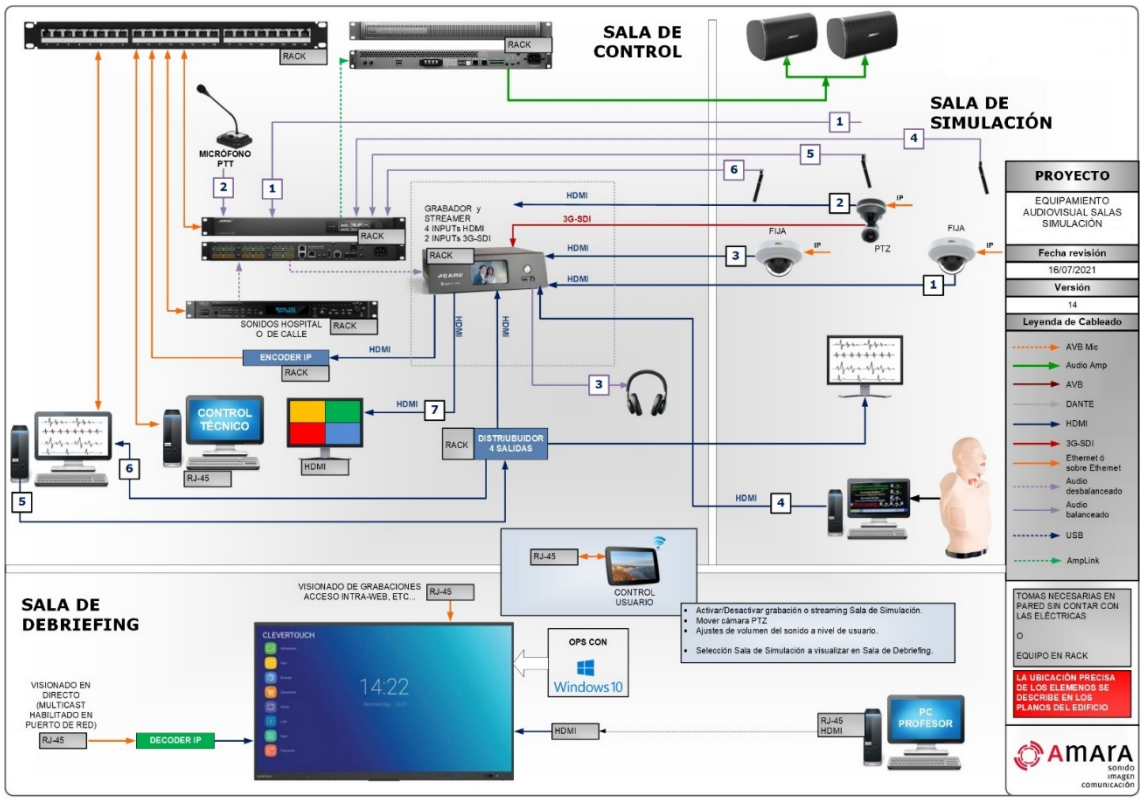


Fig.156: Esquema de proyecto audiovisual realizado por la empresa Amara.

Entendemos y cogemos este proyecto de referencia e inicio en nuestros siguientes proyectos a realizar en Amara y de los cuales consideramos de primera actualidad futura, donde el procesamiento de sistemas de Visión e Inteligencia Artificial, por nuestra parte, a diferencia de otras empresas basadas en procesos como Big Data, nos dedicamos a procesamiento de sistemas audiovisuales y donde en conjunto con otros procesos que acompañan a este, como Audio Artificial compondrán los siguientes procesos de realización de proyectos audiovisuales y de los cuales esperamos realizar e implementar en Amara.

Bibliografía:

[1] Estructura y tecnología de computadores. Alberto Ros. Universidad de Murcia.

[2] Ref.: cs231n (Stanford) - Fei-Fei Li & Justin Johnson & Serena Young

[3] https://www.nvidia.es

[4] Aprendizaje Profundo para Visión Artificial. Maurico Delbracio et. Al. Facultad de Ingeniería, Universidad de la República. 2018.

[5] Configuración de algoritmos de visión artificial en la tarjeta NVidia Jetson TK1 DevKit. TFG: Mario Luis Álvarez. Grado en Ingeniería y Automática Industrial. Universidad de Alcalá.

[6] Apuntes curso Deep Learning aplicado a tratamiento de señales e imágenes. Centro de Formación Post Grado. U. Politécnica de Valencia. 2019.

[7] Curso de programación Python para Visión Artificial con Open CV. https://unipython.com

[8] Realización de Reconocimiento facial y de objetos. https://www.youtube.com/watch?v=kUMjVo25kX0