CARLOS GARCÍA SILVA
UNIVERSIDAD DE HUELVA

GRADO INGENIERÍA INFORMÁTICA
COMPUTACIÓN

SEGUIMIENTO DE ROSTRO

NVIDIA JETSON-NANO

Índice

Introducción

En este proyecto consiste en el desarrollo de un modelo capaz de reconocer un rostro humano y seguirle para que la imagen esté siempre centrada en él.

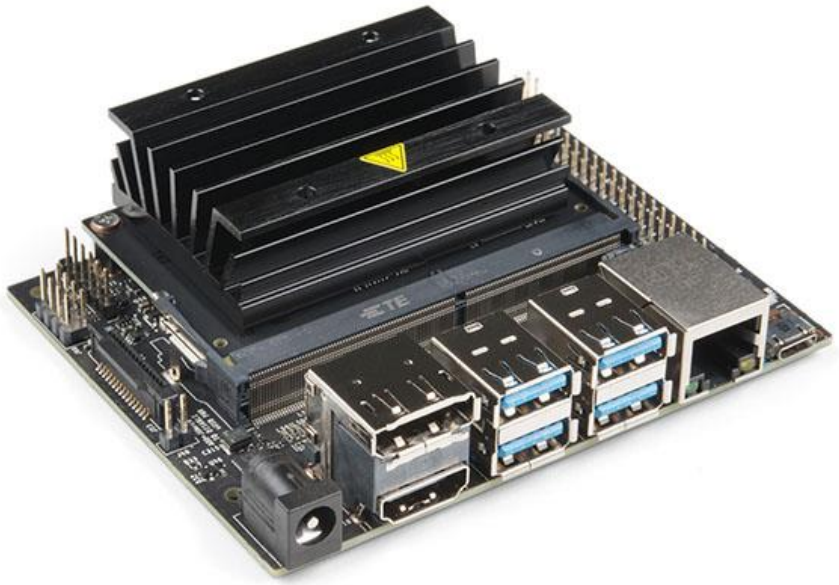
Para conseguir esto, se nos ha facilitado una placa Nvidia Jetson Nano, junto con una cámara y un servo.

Esta tarjeta cuenta con un sistema operativo basado en Ubuntu, por lo que la fase de implementación ha resultado similar al uso de un equipo sobremesa o portátil tradicional. Ha sido necesario la instalación de varias librerías y repositorios que iremos comentando a lo largo de la memoria.

Cabe destacar que la fase de instalación del sistema operativo no fue realizada por nosotros por lo que no aparecerá explicado, pero para cualquier duda sobre ello en el propio foro de nvidia existen tutoriales que sirven como guía.

El dispositivo

Como hemos mencionado antes, la Jetson cuenta con un sistema operativo basado en Ubuntu, para el desarrollo de la práctica decidimos conectar el equipo a diferentes periféricos para usarlo directamente sobre la propia máquina y no tener que conectarnos remotamente para manejarla. Esto más tarde nos trajo muchas facilidades a la hora de visualizar la cámara y hacer de forma más rápidas diferentes pruebas y ajustes.

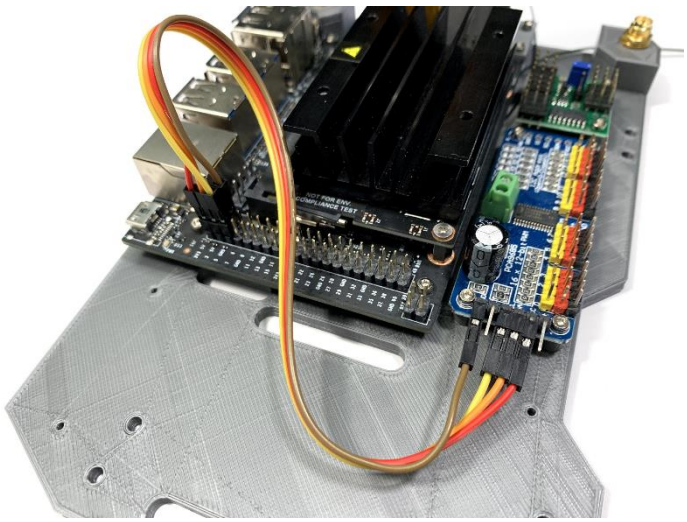


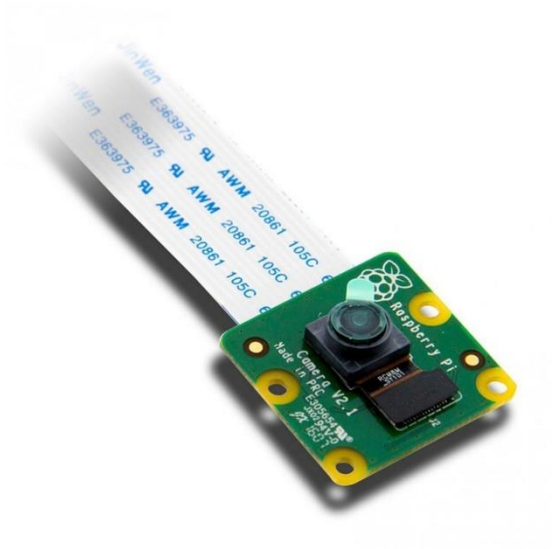
Como podemos observar tiene varias entradas, de las cuales nosotros usaremos:

- HDMI: Para poder visualizar en una pantalla externa la interfaz gráfica.
- Dos puertos USB tipo A: Para teclado y ratón.
- Ethernet: Para dotar de conexión a internet. Más tarde explicaremos como hacerlo.
- Alimentación por entrada “Barril”.

Teóricamente se puede alimentar la placa con un cable micro usb, pero nos ha dado problemas y se ha optado por usar la otra entrada mencionada anteriormente.

Esto sería el kit básico de la placa, pero como hemos mencionado, necesitamos unas extensiones, aquí es donde entra en juego una segunda placa para controlar el servo y la cámara. En las imágenes podemos ver un ejemplo de cada una de ellas.





En cuanto a este apartado no vamos a aportar más información, ya que hay muchísimas posibilidades de configuración y se pueden utilizar componentes diferentes a los que hemos usado. Se los han facilitado las placas ya previamente armadas y conectadas por lo que nos vamos a centrar en la fase de desarrollo, más que en la de montaje y preparación de Hardware.

Preparación del Entorno de trabajo

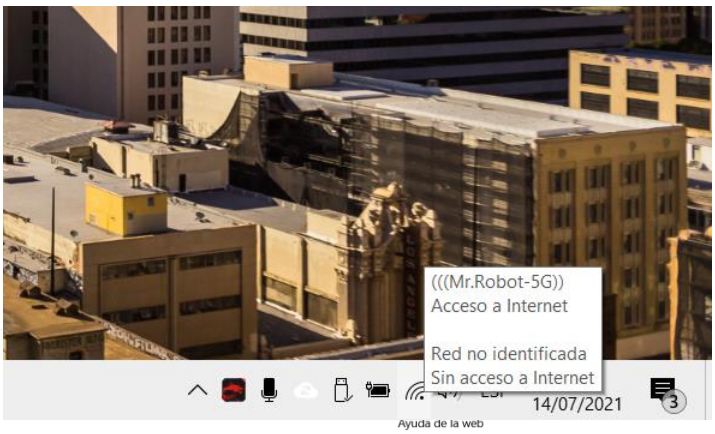
Como ya hemos mencionado, este equipo trabajo con el sistema operativo Ubuntu. Recomendamos conectar todos los periféricos antes de dotar de alimentación a la placa.

Lo primero que veremos al inicio será la típica pantalla de usuario, donde deberemos iniciar con las credenciales facilitados o si hemos realizado nosotros la instalación, el usuario y contraseña que creamos en su momento.

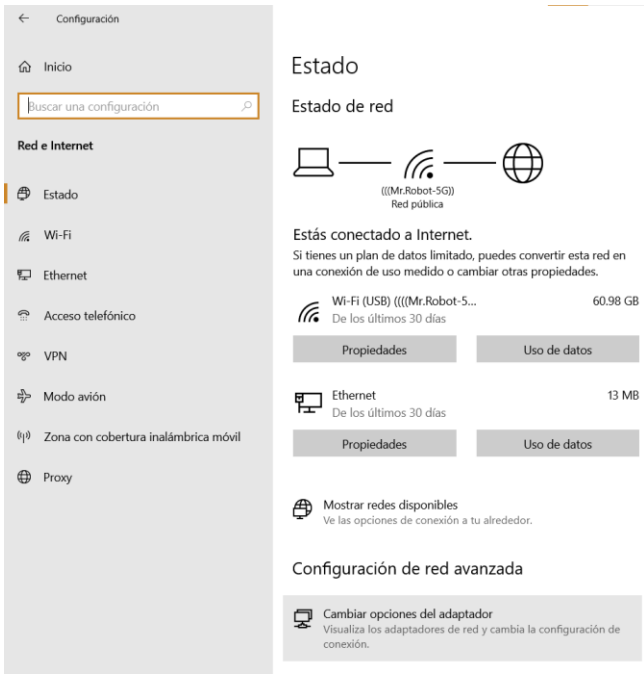
Como lo recomendado antes de empezar a trabajar es asegurarse que el equipo está actualizado, necesitaremos conexión a internet. Si no disponemos de la posibilidad de tener el router cerca de nosotros, podemos usar otro equipo con conexión wifi para hacer de puente. Este ha sido nuestro caso y explicaremos como se puede solucionar desde un equipo Windows.

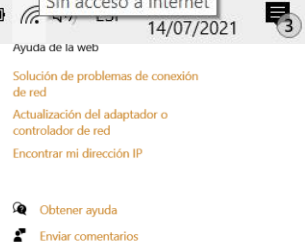
Conexión a internet mediante puente

Una vez en el equipo que hará de puente lo primero es asegurarnos que tiene conexión por wifi. Dando click derecho en el icono de conexión a internet de la barra de tareas, nos dirigimos a la configuración de red e internet.

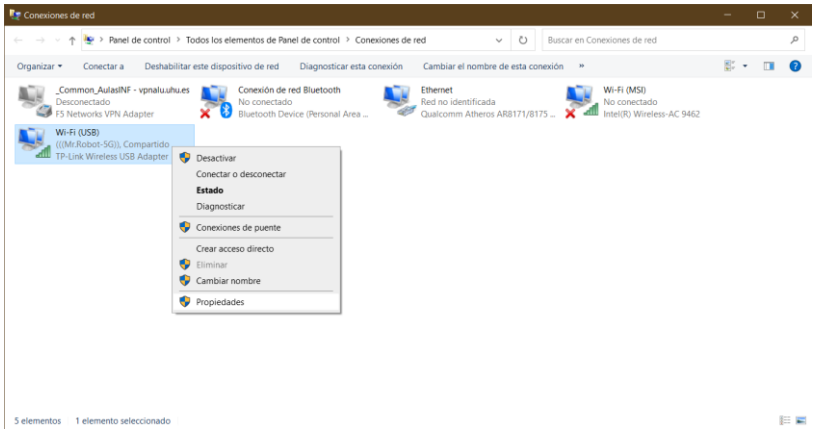


En ella nos dirigimos al apartado de cambiar opciones de adaptador.

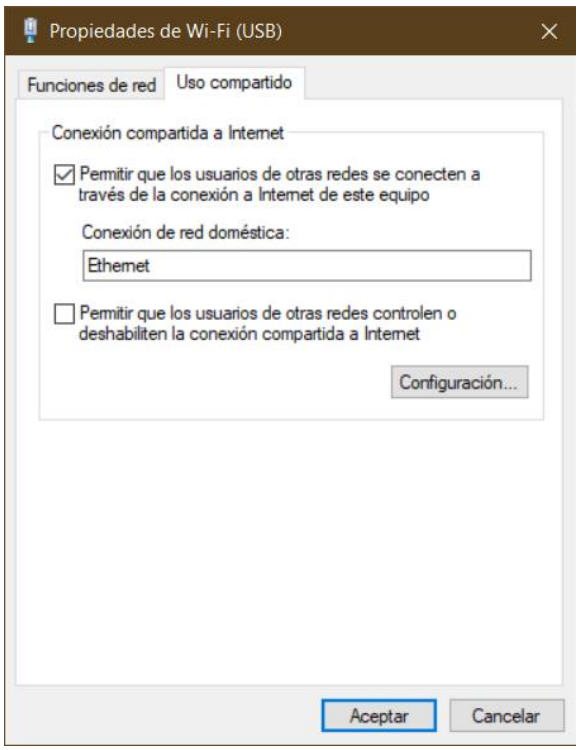




Nos aparecerá una ventana con todos los adaptadores disponibles, seleccionamos en deseado y damos botón derecho sobre él y nos vamos a propiedades.



Una vez en propiedades, nos dirigimos a la pestaña de uso compartido y en el campo de selección ponemos nuestro adaptador ethernet que deberá tener conectado el cable desde nuestro equipo a la jetson. Con esto dotaremos de conexión internet a nuestra placa y podremos navegar por internet y descargar las librerías necesarias.



Actualización de la placa

Para actualizar la placa deberemos ejecutar dos comandos.

```
alumno2@jetson-2:~$ sudo apt-get upgrade
```

```
alumno2@jetson-2:~$ sudo apt-get update
```

Una vez actualizado podemos comenzar a instalar el entorno de trabajo. Para ello vamos a explicar como instalar un editor de código bastante cómodo, se trata del Visual Studio Code, solo que lo haremos bajo otro nombre, pero en definitiva es dicho programa.

Instalación de editor de código

Lo primero que deberemos hacer es abrir una terminar y dirigirnos a la carpeta de descargas para ejecutar el siguiente comando:

```
alumno2@jetson-2:~/Descargas$ sudo apt-get install curl
```

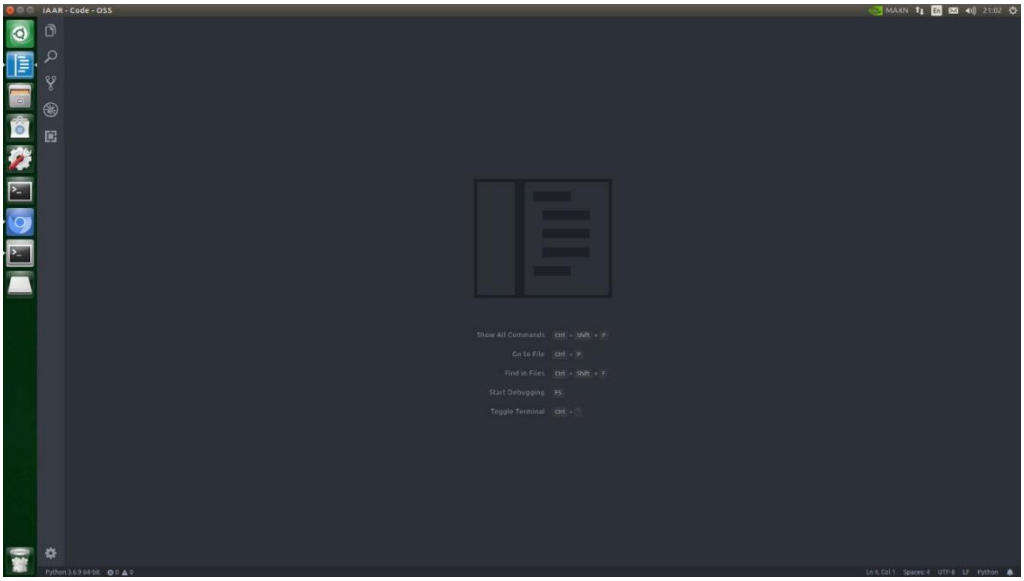
Utilizaremos Curl para descargar el siguiente repositorio, que es el que contiene el editor:

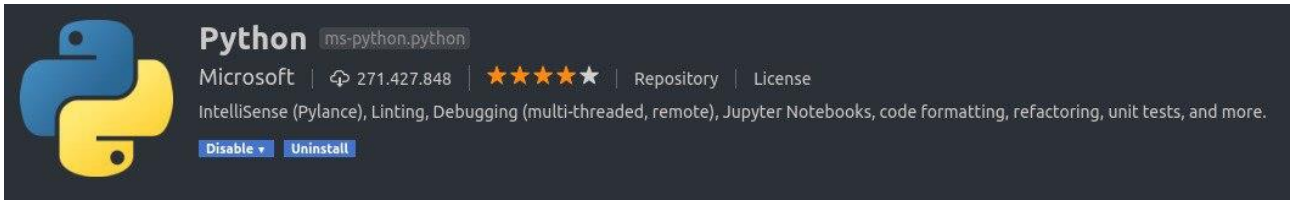
```
alumno2@jetson-2:~/Descargas$ curl -L
https://github.com/toolboc/vscode/releases/download/1.32.3/code-oss_1.32.3-
arm64.deb -o code-oss_1.32.3-arm64.deb
```

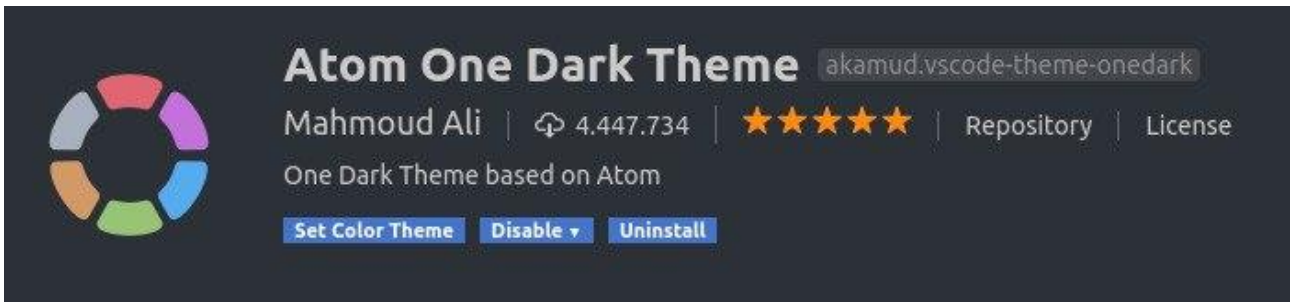
Una vez descargado solo hace falta instalarlo con el siguiente comando, una vez termine podremos disfrutar del editor:

```
alumno2@jetson-2:~/Descargas$ sudo dpkg -i code-oss_1.32.3-arm64.deb
```

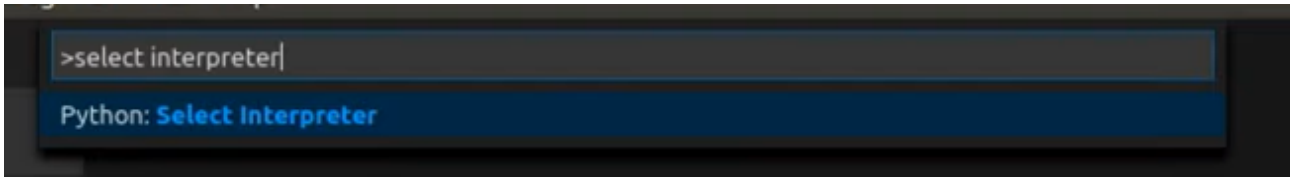
Ahora podemos abrirlo y se vería como podemos observar en la imagen, pero nos hacen faltan una serie de extensiones para poder editar y lanzar nuestro código desde el propio VSCode.



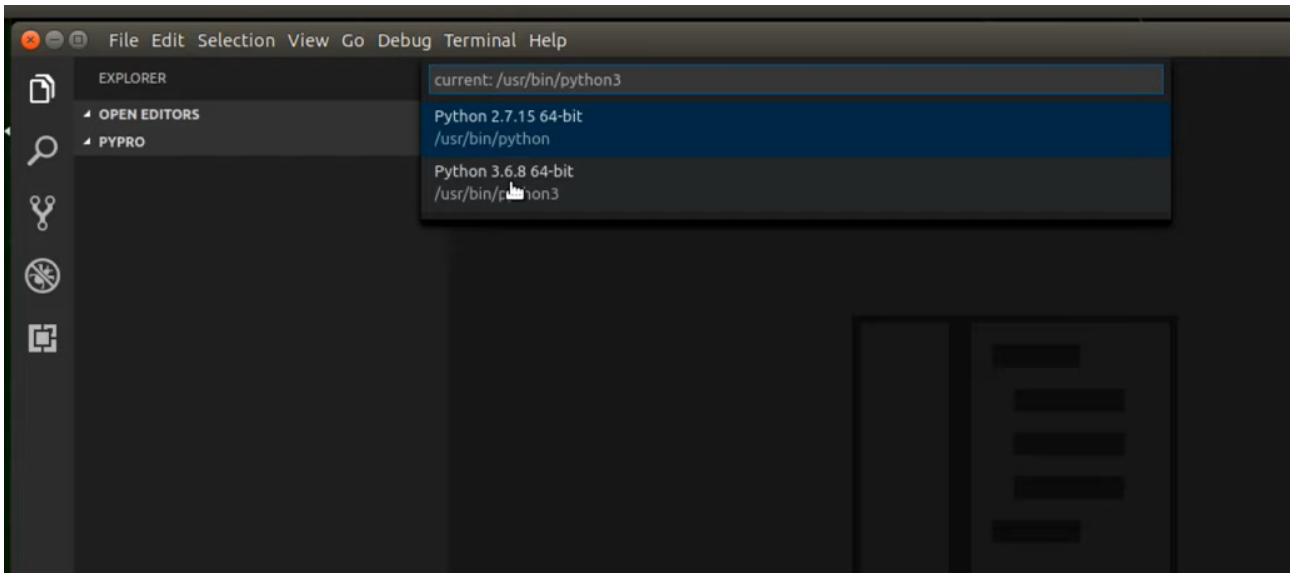




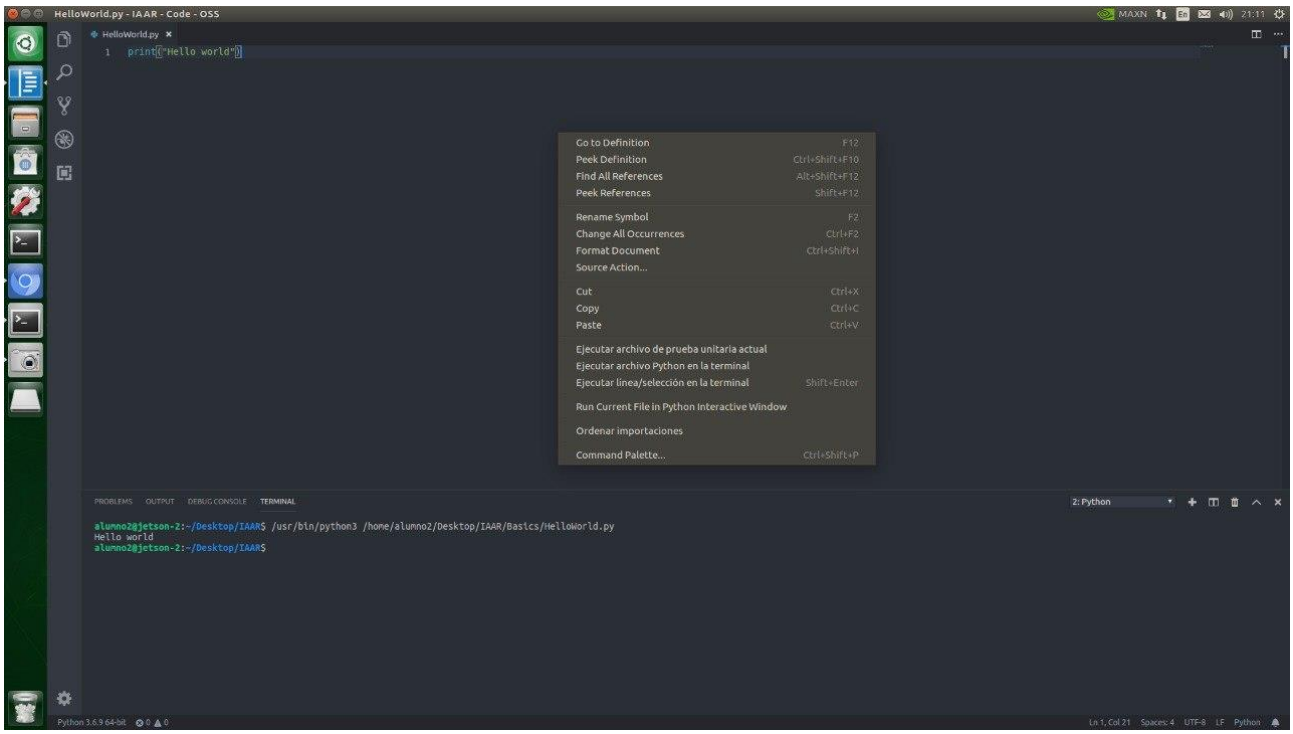
Para poder comenzar a usar la extensión de Python deberemos configurar la versión que vamos a utilizar para ello abrimos una línea de comando y escribimos:



Más tarde deberemos elegir la versión que deseamos, en nuestro caso la segunda.



Ahora ya podemos crear código y ejecutarlo en la misma ventana. Para poder ejecutarlo basta con hacer click derecho sobre el código y elegir la opción Ejecutar archivo Python en la terminal. Esto hará que en la parte inferior se nos habrá una nueva terminal que ejecuta el código, como podemos ver en el ejemplo de la imagen:



Desarrollo del modelo

Para hacer de forma más intuitiva la explicación de cómo hemos llegado al resultado final, hemos optado por dividir el desarrollo en pequeñas explicaciones de los diferentes conceptos necesarios, por separado, para poder centrarnos en una característica que más tarde combinaremos entre sí para conseguir el proyecto final. Comenzaremos por el movimiento del Servo.

Movimiento del servo

Para el movimiento del servo tenemos pensado que éste barra la zona cuando no encuentre un rostro humano. Lo que necesitamos por ahora es conocer cómo funciona el posicionamiento del servo.

Instalaciones previas

Para poder hacer uso del servo, deberemos realizar una serie de instalaciones previas, como por ejemplo pip de Python3, o descargar una librería para controlar el posicionamiento.

Lo primero será instalar python3-pip si no lo tenemos ya instalado:

```
alumno2@jetson-2:~$ sudo apt-get install python3-pip
```

Una vez instalado pip, deberemos instalar la librería para poder operar con las piezas del servo y por tanto manejar su posicionamiento:

```
alumno2@jetson-2:~$ sudo pip3 install adafruit-circuitpython-servokit
```

También vamos a necesitar instalar la librería OpenCV para python3:

```
alumno2@jetson-2:~$ sudo apt-get install python3-opencv
```

A continuación, deberemos de hacer una serie de configuraciones en carpetas internas de Nvidia y otras más que no entraremos en detalle de su funcionamiento. Hemos seguido un tutorial donde se explica más a fondo, lo dejaremos en la bibliografía:

```
alumno2@jetson-2:~$ sudo usermod -aG i2C pjm
alumno2@jetson-2:~$ sudo groupadd -f  -r gpio
alumno2@jetson-2:~$ sudo usermod -a -G gpio pjm
alumno2@jetson-2:~$ sudo cp /opt/nvidia/jetson-gpio/etc/99-gpio.rules
/etc/udev/rules.d
alumno2@jetson-2:~$ sudo udevadm control –reload-rules &&  sudo udevadm
trigger
alumno2@jetson-2:~$ sudo reboot now
```

Script para movimiento del servo

Para empezar deberemos importar las siguiente librerías (si no contamos con alguna de ellas en la bibliografía aparecerán enlaces a tutoriales para su instalación):

```
import cv2
import time

from adafruit_servokit import ServoKit
```

Nos crearemos una variable para el servo, éste será nuestro objeto con el cual realizaremos las llamadas para cambiar el posicionamiento. Como vemos la función necesita como parámetro el número de canales, en nuestro caso siempre es 16 y es con el que hemos tenido mejor funcionamiento.

```
servo = ServoKit(channels=16)
```

Para seleccionar la posición debemos manejar dos parámetros que corresponden al ángulo horizontal y al ángulo vertical. En la imagen podemos ver como servo[0] corresponde con el horizontal y servo[1] con el vertical. En este caso hemos seleccionado como posición inicial el ángulo 0º horizontal y 60º vertical.

```
def resetServo():
    servo.servo[0].angle = 0
    servo.servo[1].angle = 60
```

Para hacer el efecto barrido solo es necesario hacer un bucle en el sentido horizontal desde el ángulo mínimo hasta el máximo y si se desea, pues retroceder invirtiendo el bucle inicial:

```
def sondea():
    for i in range(0, 180, 1):
        servo.servo[0].angle = i
        time.sleep(.05)
    for i in range(180, 0, -1):
        servo.servo[0].angle = i
        time.sleep(.05)
```

Si ejecutamos en script **IAAR-MovimientoServo.py** podremos ver todo esto en acción.

Seguimiento de un objeto

En este apartado vamos a explicar cómo podemos hacer que el servo haga un tracking de un objeto. Siguiendo un tutorial sobre el tema, se nos explica cómo poder hacer tracking de un objeto según su color, en otras palabras, si nuestro objeto es azul, hacer un tracking de la zona que detecta la cámara con dicho color. Una vez encontrada dicha zona nos quedamos con el rectángulo de menor área que abarca toda la zona, es decir, su bounding box. A partir de este bbox encontramos su punto central y ese será el lugar a donde tiene que apuntar nuestro servo. Por lo que el siguiente paso sería calcular los ángulos necesarios para apuntar a dicha zona ya mover nuestro servo con lo que ya hemos aprendido.

Instalaciones previas

Dos de las tres instalaciones necesarias ya la hemos realizado en los anteriores pasos, se tratan de las librerías OpenCV y Adafruit para ServoKit. La tercera librería que vamos a usar es Numpy.

```
alumno2@jetson-2:~$ sudo apt-get install git cmake libpython3-dev python3-
numpy
```

Script Seguimiento de color

Comenzamos a ver todo lo explicado sobre en código. Como siempre primero las importaciones necesarias.

```
import cv2
print(cv2.__version__)
import numpy as np
from adafruit_servokit import ServoKit
```

Cargamos nuestro servo como hicimos en el otro punto. Y en este caso como queremos seleccionar el color, en el tutorial que seguimos nos enseñaron cómo crear una ventana con barras deslizadoras para escoger los parámetros. Nosotros no vamos a explicarlo, vendrá en el script y dejaremos un enlace a dicho video para mejor comprensión.

Deberemos configurar los siguientes parámetros: ancho y alto de la ventana y el modo de rotación. Esto último puede varias dependiendo de cómo se haya instalado la cámara, en nuestro caso es 0.

```
dispW=640
dispH=480
flip=0
```

A continuación, tenemos que crear nuestra fuente de video, de donde sacaremos los frames que vamos a tratar. En este caso lo estamos haciendo desde cv2 pero más adelante veremos una forma alternativa para usarlo desde jetson.utils, que a nuestro parecer, nos ha resultado más sencillo.

```
camSet='nvarguscamerasrc !  video/x-raw(memory:NVMM), width=3264, height=2464, format=NV12, framerate=21/1 ! nvvidconv flip-method='
+str(flip)+' ! video/x-raw, width='+str(dispW)+', height='+str(dispH)+', format=BGRx ! videoconvert ! video/x-raw, format=BGR ! appsink'

cam= cv2.VideoCapture(camSet)
```

Como hemos mencionado, tenemos que obtener un frame para tratarlo y así generar las observaciones sobre él.

```
    ret, frame = cam.read()
```

Mediante CV2 vamos a encontrar los contornos en la imagen que cumplen nuestra condición, la de que sea el color que mediante los parámetros, hemos ajustado. Como puede que encontremos más de uno, vamos a ordenarlo con el criterio de primero los de mayor área, entendiendo que nuestro objeto va a ser la zona donde mayor número de pixeles detectemos como nuestro color y así evitamos pequeñas agrupaciones de pixeles de color similar o incluso pixeles de ruido.

```
    contours, hierarchy = cv2.findContours(FGmaskComp, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours=sorted(contours,key=lambda x:cv2.contourArea(x),reverse=True)
```

Ahora comenzamos un bucle que para cada uno de los contornos detectados, calcularemos su área y su bbox. La función usada para obtener el bounding box nos devuelve su punto de la esquina superior izquierda y su ancho y largo, por lo que nos será fácil calcular la “caja”.

```
    for cnt in contours:
        area=cv2.contourArea(cnt)
        (x,y,w,h)=cv2.boundingRect(cnt)
```

Vamos a dibujar el borde de esta caja sobre el frame en cuestión. Para poder aplicar el criterio que mencionamos antes, vamos a hacer un filtrado de las detecciones con área menos que valor 50. Ahora sí, podemos mostrar el rectángulo del bbox, para ello haciendo uso de funciones de cv2, quedaría tal que así:

```
if area>=50:
    #cv2.drawContours(frame,[cnt],0,(255,0,0),3)
    cv2.rectangle(frame,(x,y),(x+w,y+h),(255,0,0),3)
```

Ahora calcularemos los ángulos necesarios para colocar el servo y poder hacer el tracking. Debemos calcular el centro del bbox, para ello sabiendo su esquina superior izquierda y su alto y ancho, podemos averiguar sus coordenadas X e Y. Según el tutorial, realizar el cálculo de esta forma puede llevar un pequeño error consigo que compensaremos calculando el error de paneo y tilt, es decir los ángulos horizontal y vertical respectivamente.

```
objX=x+w/2
objY=y+h/2
errorPan=objX-width/2
errorTilt=objY-height/2
if abs(errorPan)>15:
    pan=pan-errorPan/75
if abs(errorTilt)>15:
    #tilt=tilt-errorTilt/75
    tilt=tilt+errorTilt/75
```

En dichos cálculos podemos pasarnos de rango y obtener valores mayores que los límites, para evitar eso deberemos comprobar el resultado de cada ángulo y si lo sobre pasa, asignarle el valor máximo para evitar errores a la hora de ejecutar.

```
if pan>180:
    pan=180
    print("Pan Out of  Range")
if pan<0:
    pan=0
    print("Pan Out of  Range")
if tilt>180:
    tilt=180
    print("Tilt Out of  Range")
if tilt<0:
    tilt=0
    print("Tilt Out of  Range")
```

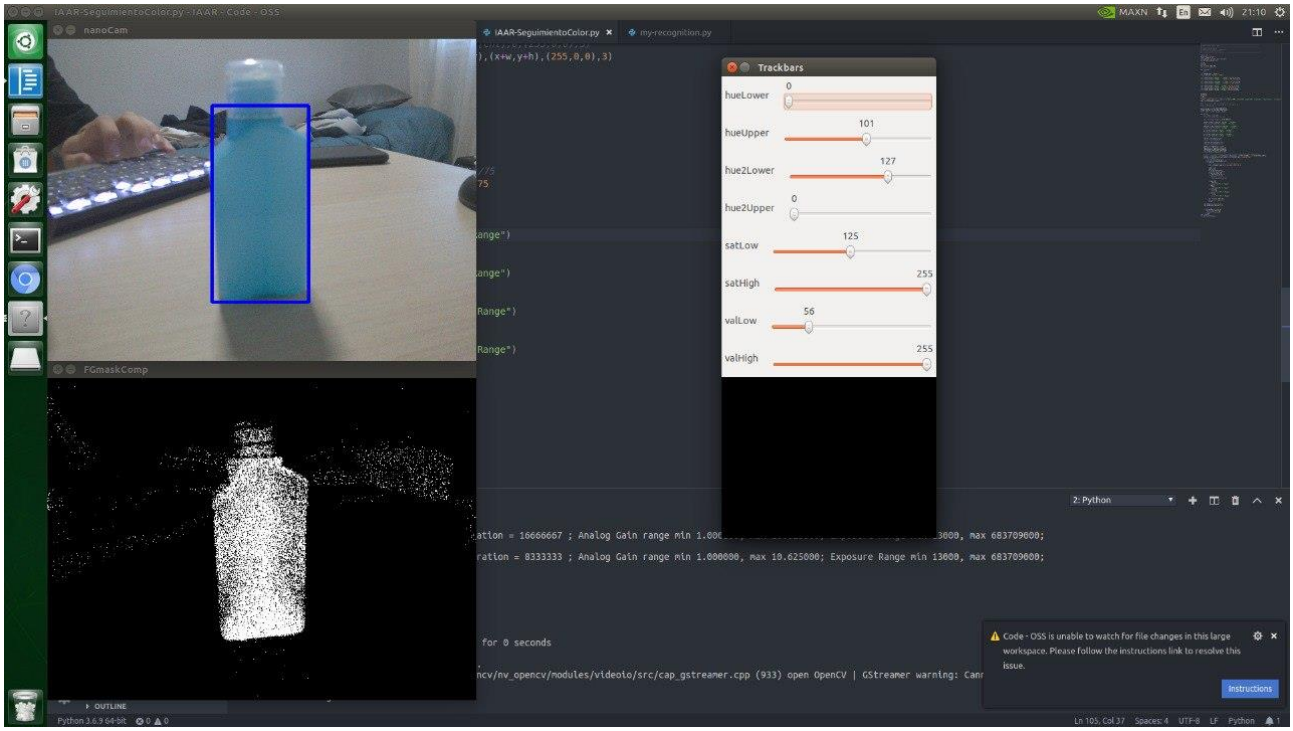
Asignamos la posición como aprendimos anteriormente y mostramos el frame modificado en la ventana correspondiente:

```
cv2.imshow('nanoCam',frame)
cv2.moveWindow('nanoCam',0,0)
```

Por último, tenemos la opción de cerrar todas las ventanas y finalizar el programa cerrando la cámara:

```
cam.release()
cv2.destroyAllWindows()
```

Todo el código se encuentra en el script **IAAR-SeguimientoColor.py**, a continuación, vamos a ver una captura de cómo se ve el programa en ejecución:



Ejecutar un modelo

En este apartado vamos ya a entrar en la base de conocimiento de este proyecto. Vamos a ejecutar un modelo de reconocimiento de objetos, pero en el repositorio que hemos seguido se encuentran muchas funcionalidades muy interesantes, como modelos de reconocimiento de imagen, segmentación, etc

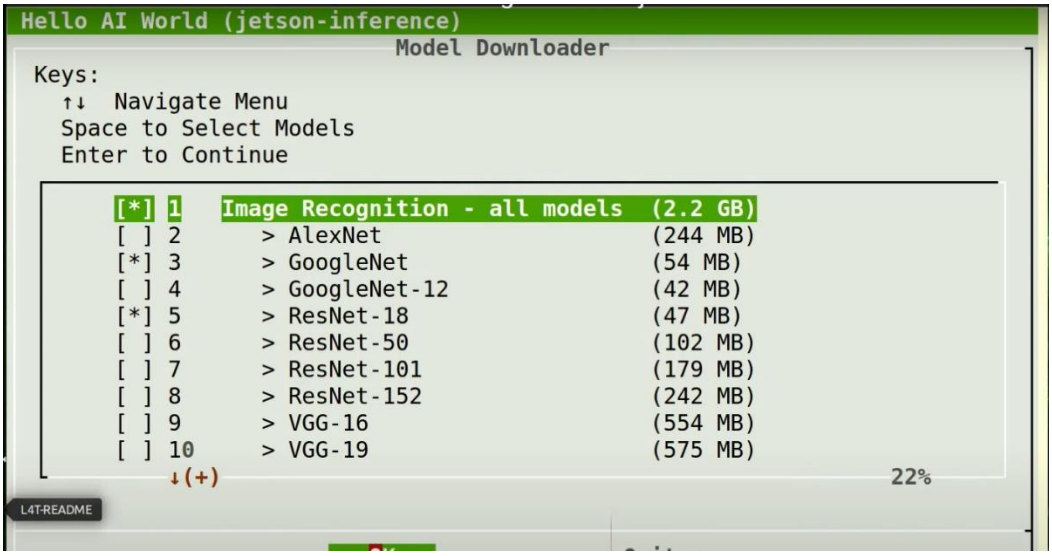
Deberemos descargar el repositorio jetson-inference en la carpeta que deseemos:

```
alumno2@jetson-2:~/Descargas$ git clone --recursive https://github.com/dusty-nv/jetson-inference
```

Después de descargar el repositorio, deberemos instalar varios modelos ya preeentrenados para poder hacer pruebas con ellos, serán de varios tipos de campos de aplicación. Nosotros hemos descargado los indicados en el tutorial que hemos dejado en el bibliografía, pero realmente hemos usado únicamente los del tipo ssd-mobilenet, que más tarde explicaremos su estructura.

```
alumno2@jetson-2:~/Descargas/jetson-inference/build$ cmake ../
```

Al ejecutar este comando, comenzará a instalar la librería necesaria y abrirá una ventana de instalación de los modelos como la que mostramos:



Después de instalar, nos aparecerá otra ventana de instalación de Pytorch. Si hemos seguido todos los tutoriales que hemos facilitado, además de esta memoria, deberemos tener especial cuidado en escoger la versión de Python 3.6 que es en la que hemos desarrollado, si se está usando otra, asegurarse de que es la misma que se tiene instalada en el equipo.

Deberemos ejecutar algunos comandos más para la configuración de jetson-inference:

```
alumno2@jetson-2:~/Descargas/jetson-inference/build$ make -j$(nproc)

alumno2@jetson-2:~/Descargas/jetson-inference/build$ sudo make install

alumno2@jetson-2:~/Descargas/jetson-inference/build$ sudo ldconfig

alumno2@jetson-2:~/Descargas/jetson-inference/build$ sudo apt-get install
v4l-utils
```

Script Ejecuta modelo

Vamos a ver el código implementado para poder ejecutar un modelo de reconocimiento de objetos preentrenado y que se encuentre en el repositorio jetson-inference o si poseemos otro modelo con una extensión admitida, podremos utilizarlo. Las compatibilidades y modelos admitidos las explicaremos más adelante en otro apartado.

Las librerías que necesitaremos serán:

```
import jetson.inference
import jetson.utils

import argparse
import sys
```

Lo siguiente es crear un parser de Python para poder cargar los parámetros a la hora de hacer la llamada a guión. En nuestro caso lo hemos configurado para que cargue un modelo que hemos entrenado nosotros mismos, que será uno de los siguientes apartados. En jetson-inference existe un script a modo de ejemplo llamado detectnet.py donde se podrá ver mejor esta parte si lo que se quiere es cargar uno de los modelos preentrenados.

```
parser = argparse.ArgumentParser(description = "Locate objects in a live camera stream using an object detection DNN.",
            formatter_class = argparse.RawTextHelpFormatter, epilog=jetson.inference.detectNet.Usage()
                + jetson.utils.videoSource.Usage()
                + jetson.utils.videoOutput.Usage()
                + jetson.utils.logUsage())

parser.add_argument("input_URI", type = str, default = "", nargs = '?', help = "URI of the input stream")
parser.add_argument("output_URI", type = str, default = "", nargs = '?', help = "URI of the output stream")
parser.add_argument("--network", type = str, default = "ssd-mobilenet-v2", help = "pre-trained model to load (see below for options)")
parser.add_argument("--overlay", type = str, default = "box,labels,conf", help = "detection overlay flags (e.g. --overlay=box,labels,conf)\nvalid combinations are:
parser.add_argument("--threshold", type = float, default = 0.5, help = "minimum detection threshold to use")
```

Los modelos preentrenados son variados y detectan infinidad de objetos, pero hay algunos específicos para rostros como el facenet:

Object Detection

Network	CLI argument	NetworkType enum	Object classes
SSD-Mobilenet-v1	`ssd-mobilenet-v1`	`SSD_MOBILENET_V1`	91 (COCO classes)
SSD-Mobilenet-v2	`ssd-mobilenet-v2`	`SSD_MOBILENET_V2`	91 (COCO classes)
SSD-Inception-v2	`ssd-inception-v2`	`SSD_INCEPTION_V2`	91 (COCO classes)
DetectNet-COCO-Dog	`coco-dog`	`COCO_DOG`	dogs
DetectNet-COCO-Bottle	`coco-bottle`	`COCO_BOTTLE`	bottles
DetectNet-COCO-Chair	`coco-chair`	`COCO_CHAIR`	chairs
DetectNet-COCO-Airplane	`coco-airplane`	`COCO_AIRPLANE`	airplanes
ped-100	`pednet`	`PEDNET`	pedestrians
multiped-500	`multiped`	`PEDNET_MULTI`	pedestrians, luggage
facenet-120	`facenet`	`FACENET`	faces

Para cargar el modelo el modelo tenemos que crear un objeto de la clase detectNet, es una de las clases de jetson-inference implementada en Python y C++, en nuestro caso estamos haciendo uso de la versión en Python, la definición es la siguiente:

```
class detectNet(tensorNet)
    Object Detection DNN - locates objects in an image

    Examples (jetson-inference/python/examples)
         detectnet-console.py
         detectnet-camera.py

    __init__(...)
         Loads an object detection model.

         Parameters:
           network (string) -- name of a built-in network to use
                               see below for available options.

           argv (strings) -- command line arguments passed to detectNet,
                             see below for available options.

           threshold (float) -- minimum detection threshold.
                                default value is 0.5

    detectNet arguments:
      --network=NETWORK     pre-trained model to load, one of the following:
                                * ssd-mobilenet-v1
                                * ssd-mobilenet-v2 (default)
                                * ssd-inception-v2
                                * pednet
                                * multiped
                                * facenet
                                * coco-airplane
                                * coco-bottle
                                * coco-chair
                                * coco-dog
      --model=MODEL         path to custom model to load (caffemodel, uff, or onnx)
      --prototxt=PROTOTXT   path to custom prototxt to load (for .caffemodel only)
      --labels=LABELS       path to text file containing the labels for each class
      --input-blob=INPUT    name of the input layer (default is 'data')
      --output-cvg=COVERAGE name of the coverge output layer (default is 'coverage')
      --output-bbox=BOXES   name of the bounding output layer (default is 'bboxes')
      --mean-pixel=PIXEL    mean pixel value to subtract from input (default is 0.0)
      --batch-size=BATCH    maximum batch size (default is 1)
      --threshold=THRESHOLD minimum threshold for detection (default is 0.5)
      --alpha=ALPHA         overlay alpha blending value, range 0-255 (default: 120)
      --overlay=OVERLAY     detection overlay flags (e.g. --overlay=box,labels,conf)
                            valid combinations are:  'box', 'labels', 'conf', 'none'
      --profile             enable layer profiling in TensorRT
```

Continuamos creando dos fuentes, una de video que será la que capture los frames de nuestra cámara y otra para generar la salida de video. Para la entrada debemos indicar la cámara y también configuramos la rotación. En puntos anteriores enseñamos cómo hacerlo con la librería OpenCV, esta vez lo haremos con jetson-utils:

```
input = jetson.utils.videoSource("csi://0", argv=['--input-flip=rotate-180'])
output = jetson.utils.videoOutput(opt.output_URI, argv=sys.argv+is_headless)
```

Finalmente creamos un bucle para que todos los frames se estén mostrando por pantalla. Cargaremos un frame, buscaremos las detecciones con el método detect de nuestra red, que tiene la siguiente definición:

```
Detect(...)
    Detect objects in an RGBA image and return a list of detections.

    Parameters:
      image   (capsule) -- CUDA memory capsule
      width   (int)  -- width of the image (in pixels)
      height  (int)  -- height of the image (in pixels)
      overlay (str)  -- combination of box,labels,none flags (default is 'box')

    Returns:
      [Detections] -- list containing the detected objects (see detectNet.Detection)
```

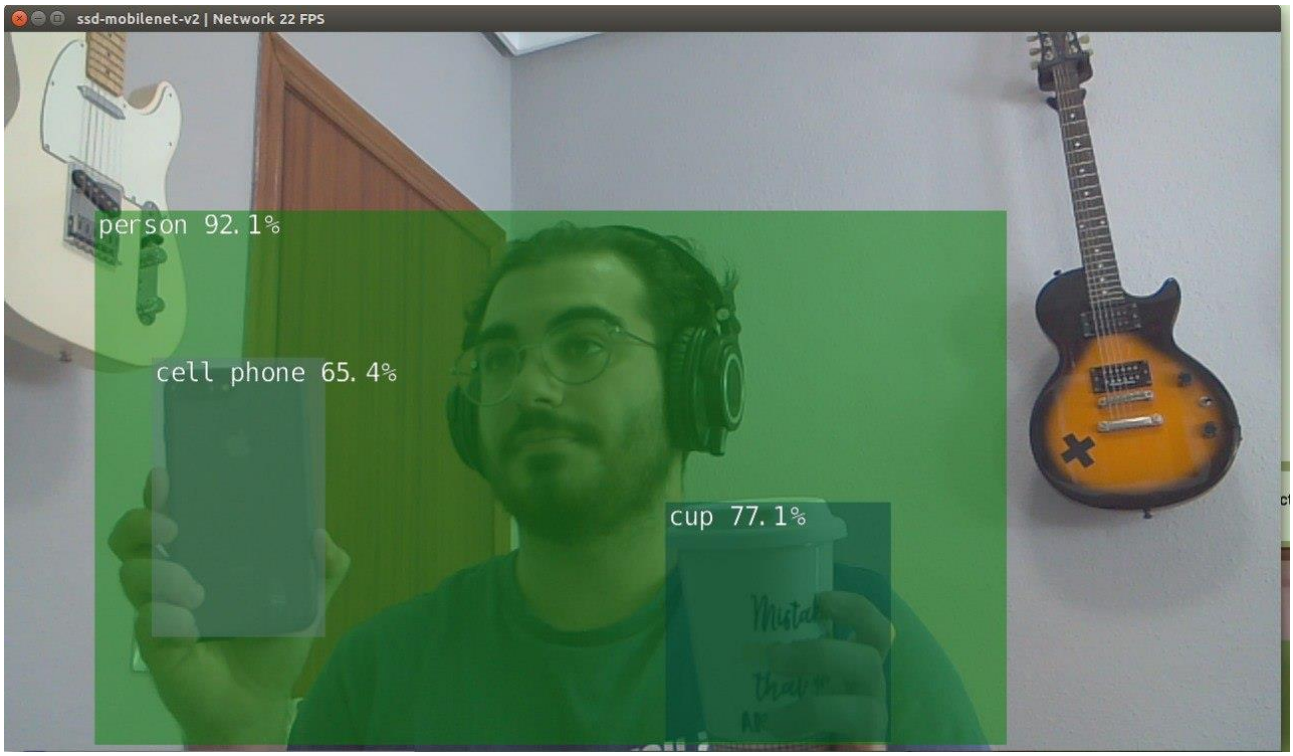
Como podemos ver, nos devuelve una lista de detecciones. Éstas tienen como atributos los siguientes(los cuales nos han resultado de gran utilidad en la implementación posterior):

```
Detection = <type 'jetson.inference.detectNet.Detection'>
    Object Detection Result

    ----------------------------------------------------------------------
    Data descriptors defined here:

    Area
        Area of bounding box

    Bottom
        Bottom bounding box coordinate

    Center
        Center (x,y) coordinate of bounding box

    ClassID
        Class index of the detected object

    Confidence
        Confidence value of the detected object

    Height
        Height of bounding box

    Instance
        Instance index of the detected object

    Left
        Left bounding box coordinate

    Right
        Right bounding box coordinate

    Top
        Top bounding box coordinate

    Width
         Width of bounding box
```

La propia función detect se encarga, mediante el tipo de overlay, de representar la información en al frame. Si por ejemplo detecta un plátano en la escena, por defecto dibujará un rectángulo alrededor de la fruta, con el nombre de la clase a la que pertenece la detección y el porcentaje que tiene de pertenecer a dicha clase. Esto en código quedaría de la siguiente manera:

```
while True:
    # capture the next image
    img = input.Capture()

    # detect objects in the image (with overlay)
    detections = net.Detect(img, overlay=opt.overlay)

    # print the detections
    print("detected {:d} objects in image".format(len(detections)))

    for detection in detections:
        print(detection)

    # render the image
    output.Render(img)

    # update the title bar
    output.SetStatus("{:s} | Network {:.0f} FPS".format("Modelo para personas", net.GetNetworkFPS()))

    # print out performance info
    net.PrintProfilerTimes()

    # exit on input/output EOS
    if not input.IsStreaming() or not output.IsStreaming():
        break
```

Si queremos ejecutar el código, se encuentra implementado en el script **IAAR-EjecutaModelo.py**, debajo mostramos un ejemplo de cómo se vería la ventana de previsualización generada ejecutando un modelo preentrenado, por defecto se trata del ssd-mobilenet-v2:



Seguimiento de detección

En este apartado vamos a explicar el concepto final, cómo vamos a mezclar todos los conocimientos previos y los vamos a poner en común en el mismo script para crear nuestro programa, un script que, a partir de las percepciones de un modelo, haga un tracking de un rostro humano detectado.

Lo necesario para este punto ya lo hemos explicado, salvo como entrenar a nuestro propio modelo. Realmente esto no es necesario del todo, puesto que podemos usar modelos ya preentrenados que detectan caras o incluso usar un modelo que detecte varios objetos y clasificar la ID de la clase de las caras.

Script Seguimiento de modelo

Comenzamos con las importaciones de las librerías que vamos a necesitar, en este caso son:

```
import jetson.inference
import jetson.utils
import time

from adafruit_servokit import ServoKit
```

Para mejor comprensión, hemos dividido el main en varios pasos, donde en cada paso hemos intentado integrar el código en una función, para una mejor claridad y mayor facilidad a la hora de explicarlo.

Como primer paso nos creamos nuestro objeto servo que ya hemos aprendido a usar en anteriores puntos:

```
# 1.-Cargamos objeto para manejar Servo
servo = ServoKit(channels=16)
```

Después cargamos la red:

```
# 2.-Cargamos la red neuronal de deteccion de objetos
red = cargaRed()
```

```
# FUNCION: cargaRed()
#
# PROPOSITO: Carga una red neuronal ya entrenada con los parametros necesarios para la
# ejecucion de la practica.
# --model: Modelo de red que ejecutara, en nuestro caso la red entrenada en google colab
# --labels: Fichero de texto con las clases que reconoce nuestra red
# --input-blob: Para nuestra practica input_0
# --output-cvg: Para nuestra practica scores. Muestra los porcentajes.
# --output-bbox: Para nuestra practica boxes. Muestra los bounding boxes.
#
# DEVUELVE: Un objeto de tipo detectNet.
def cargaRed():
    argv=['--model=/home/alumno2/Desktop/IAAR/Deteccion-objetos/jetson-inference/python/training/detection/ssd/models/colab/ssd-mobilenet.onnx',
    '--labels=/home/alumno2/Desktop/IAAR/Deteccion-objetos/jetson-inference/python/training/detection/ssd/models/colab/labels.txt',
    '--input-blob=input_0',
    '--output-cvg=scores',
    '--output-bbox=boxes']

    return jetson.inference.detectNet("ssd-mobilenet", argv, threshold=0.5)
```

Para poder obtener los frames necesitamos una fuente de entrada, en este caso nuestra cámara, aunque se podría utilizar un video ya grabado. También necesitaremos una fuente de salida, en este caso una ventana en el propio SO, también existen proyecto donde la salida es enviada a otro dispositivo, por ejemplo: retransmisiones en streaming o guardar el video en un servidor, …

```
# 3.-Creamos las fuentes de video y la ventana
(camara, ventana, ancho, alto) = cargaFuentes()
```

```
# FUNCION: cargaFuentes()
#
# PROPOSITO: Crea las fuentes de entrada y salida de video. En este caso la camara
# y la ventana donde se visualizara el video y las detecciones sobre la imagen.
#
# DEVUELVE: Fuente de video, fuente de salida y dimensiones de alto y ancho.
def cargaFuentes():
    camera = jetson.utils.videoSource("csi://0", argv=['--input-flip=rotate-180'])

    preview = jetson.utils.videoOutput()

    image = camera.Capture()

    width = image.width
    height = image.height

    return (camera, preview, width, height)
```

Vamos a comenzar ya con las detecciones y el movimiento del servo, es por ello que hemos optado por colocar el servo en una posición inicial desde la cual comenzará a barrer la sala donde se encuentre la cámara:

```
# 4.-Colocamos el Servo en la posicion inicial que hemos escogido
(servo_horizontal, servo_vertical, sentido) = resetServo()
```

```
# FUNCION: resetServo
#
# PROPOSITO: Coloca el servo en la posivion de partida. En este caso selecciona como
# angulo horizontal 0grados y como vertical 60 grados.
#
# DEVUELVE: El angulo horizontal, vertical en el que ha quedado colocado el servo y
# el sentido con el que comenzara a sondear en el caso de no encontrar detecciones
def resetServo():
    pan = 0
    servo.servo[0].angle = pan
    servo.servo[1].angle = 60
    sentido = "-->"

    return (pan,60, sentido)
```

Ahora sí, comenzamos a tratar las detecciones, usando el objeto cámara, capturamos un frame y mediante detect, obtenemos las detecciones que encuentra nuestro modelo en dicho frame. Dentro el método ya creado, tenemos la opción de que mediante el overlay, nos modifique el frame de tal manera de que pinte el bounding box de las detecciones, el nombre de la clase (que en este caso solo será Human fase) y el porcentaje con el que dicha detección pertenece a nuestra clase, todo esto se puede cambiar con las opciones "box", "label", "conf"y "none".

```
# 5.-Reconocimiento de objetos y visualizacion por pantalla
while True:

    # 5.1-Detectamos las caras humanas
    (frame, detecciones, numDetecciones) = detecta(camara)
```

```
def detecta(camara):
    imagen = camara.Capture()
    detecciones = red.Detect(imagen)
    num = len(detecciones)

    return (imagen, detecciones, num)
```

Ahora llegamos a un punto que en función de lo obtenido antes, realizaremos una cosa u otra. Si hemos detectado alguna cara, vamos a buscar la detección con mayor área, esto lo hacemos porque en caso de haber dos personas, vamos a trackear a la más cercana, que suele ser la que mayor área de bbox tenga.

```
def seleccionaDeteccion(detecciones):
    area_max = 0
    pos = 0
    for i in detecciones:
        if i.Area > area_max:
            area_max = i.Area
            pos = i.Instance

    (x, y) = detecciones[pos].Center
    w = detecciones[pos].Width
    h = detecciones[pos].Height

    return (x, y, w, h)
```

Una vez tengamos escogida la detección, podemos usar varios atributos con los que cuenta esta clase, de todos ellos nosotros vamos a utilizar el centro del bbox, el ancho y el alto y su área para crear un criterio de selección de región de interés.

Aunque existen más atributos que hemos puesto para que puedan ser utilizados para crear criterios alternativos al nuestro:

```
Detection = <type 'jetson.inference.detectNet.Detection'>
    Object Detection Result

    ----------------------------------------------------------------------
    Data descriptors defined here:

    Area
        Area of bounding box

    Bottom
        Bottom bounding box coordinate

    Center
        Center (x,y) coordinate of bounding box

    ClassID
        Class index of the detected object

    Confidence
        Confidence value of the detected object

    Height
        Height of bounding box

    Instance
        Instance index of the detected object

    Left
        Left bounding box coordinate

    Right
        Right bounding box coordinate

    Top
        Top bounding box coordinate

    Width
         Width of bounding box
```

Una vez escogido todo, solo tenemos que mover el servo, para ello calculamos su posición, como ya vimos en anteriores puntos de la memoria:

```
def calculaPosicion(x, y, w, h, width, height, pan, tilt):
    obX = x
    obY = y
    errorPan = obX-width/2
    errorTilt = obY-height/2

    if abs(errorPan)>15:
        pan = pan-errorPan/75
    if abs(errorTilt)>15:
        tilt = tilt+errorTilt/75

    if pan>180:
        pan = 180
    if pan<0:
        pan = 0

    if tilt>180:
        tilt = 180
    if tilt<0:
        tilt = 0

    return (pan, tilt)
```

La única diferencia que encontramos en esta función con el método que hemos mencionado, es que anteriormente partíamos de la esquina superior izquierda del bbox de los contornos, pero esta vez la clase Detect ya nos da el centro de ese bounding box, por lo que la dos primeras líneas son solo para renombrar las variables para poder reciclar código. Tras esa pequeña aclaración solo basta utilizar la función para colocar el servo, que es muy simple: se pasa como parámetros los ángulos horizontal y vertical y se asignan al servo. Como ya hemos comprobado en la función calculaPosición, que los ángulos no sobrepasen los límites, no tenemos que preocuparnos por asignar un ángulo mayor que 180 o menos que 0.

Si no hemos detectado ningún rostro, el servo comienza a barrer la zona moviéndose horizontalmente, primero de izquierda a derecha y luego en sentido contrario.

```
# FUNCION: sondea
#
# PROPOSITO: Hace que el servo se mueva horizontalmente de un extremo a otro.
#
def sondea(sentido, pan, num):
    if num==0:
        if sentido=="-->":
            if pan>=180:
                sentido="<--"
                pan=pan-1
                servo.servo[0].angle = pan
                servo.servo[1].angle = 60
            else:
                pan=pan+1
                servo.servo[0].angle = pan
                servo.servo[1].angle = 60
        else:
            if pan<=0:
                sentido="-->"
                pan=pan+1
                servo.servo[0].angle = pan
                servo.servo[1].angle = 60
            else:
                pan=pan-1
                servo.servo[0].angle = pan
                servo.servo[1].angle = 60

    return (sentido, pan)
```

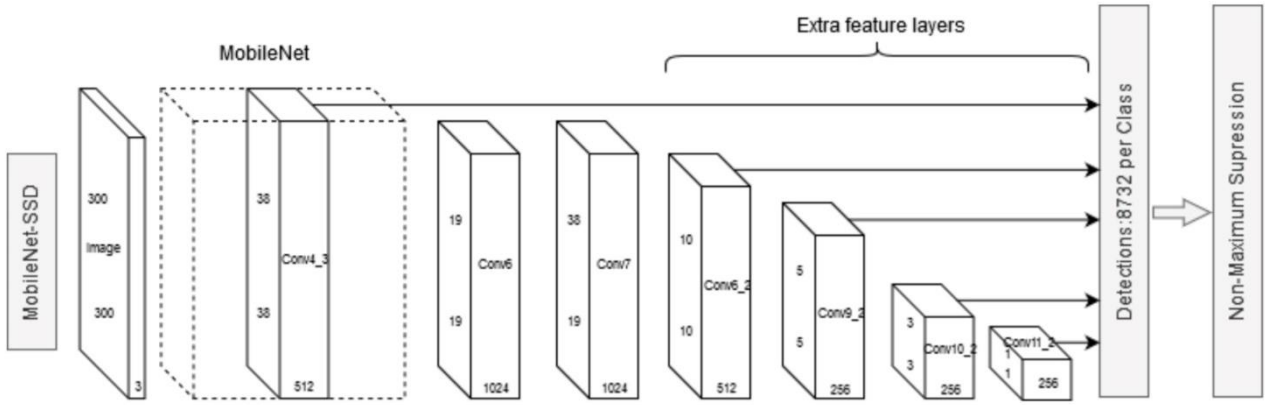
Esto sería todo el código final, para poder ejecutarlo deberemos usar el script **IAAR-SeguimientoModelo.py**. Como muestra de la ejecución podemos ver la siguiente imagen:

Re-entrenar modelo SSD

Como ya hemos nombrado antes, existen varios modelos ya establecidos en el repositorio, pero éste cuenta con un script para preentrenar modelos con las características que nosotros queramos, en este caso se hace uso del modelo **mobilenet-v1-ssd-mp-0_675.pth** que ya nos viene descargado en la carpeta Python/training/detection/ssd/models. Para ubicarnos un poco más en situación vamos a explicar qué es esto de modelo ssd.

Modelo SSD

Es un tipo de modelo que vamos a utilizar para entrenar el nuestro propio, concretamente se lo vamos a usar para detección de objetos. Vamos a utilizar Pytorch y el conjunto de datos de Open Images. SDD-MobileNet es una arquitectura de red que es bastante popular para la detección de objetos en tiempo real para dispositivos móviles y dispositivos integrados, combina el detector SSD-300 Single-Shot MultiBox con una red troncal de Mobilenet (que es una red neuronal convolucional para aplicaciones de visión móvil).



Configuración y entrenamiento

A continuación, vamos a configurar el entrenamiento y descargar todo lo necesario para conseguirlo, pero antes queremos informar de algunas cosas a tener en cuenta y que hemos querido compartir porque ha sido una de las claves de este proyecto.

La Jetson Nano posee unas claras limitaciones físicas a nivel de Hardware para poder realizar entrenamientos intensos y de buenos resultados, es por ello que hemos buscado una alternativa para realizar mejores entrenamientos y manejando cantidades de datos mayores. Vamos a explicar ambas alternativas para que el usuario decida bajo su propio criterio cuál se ajusta más a sus necesidades.

Entrenamiento en Jetson Nano

Esta es la opción más directa y sencilla, ya que contamos con todo lo necesario en nuestro dispositivo al descargar el repositorio, solo necesitaremos como extra cargar datos de imágenes para entrenamiento.

Como configuración previa, podríamos hacer uso de un Docker container que ya nos viene con el repositorio si no quisiéramos instalar en nuestro sistema los requerimientos, etc… Si este es vuestro caso, este paso os lo podéis saltar, esto es para aquellos que no quieran hacer uso del Docker. Deberemos ejecutar los siguientes comandos por consola

```
alumno2@jetson-2:~$ cd jetson-inference/python/training/gation/ssd

alumno2@jetson-2:~/jetson-inference/python/training/gation/ssd$ wget
https://nvidia.box.com/shared/static/djf5w54rjvpqocsiztzaandq1m3avr7c.pth -O
models / mobilenet-v1-ssd-mp-0_675.pth
alumno2@jetson-2:~/jetson-inference/python/training/gation/ssd$ pip3 install
-v -r requirements.txt
```

Después vamos a descargar las imágenes que queramos, nosotros vamos a usar una de las clases que posee Open Imágenes que tiene unas 600, las que nos interesa se llama Human fase y es la que pondremos en el comando de descarga. También podemos limitar la cantidad de imágenes, teniendo en cuenta las limitaciones de la Jetson Nano, recomendamos unas 1.000 fotos para ejecuciones asequibles en cuanto a tiempo de espera, hemos probado entrenamientos con datasets de 5.000 imágenes pero los tiempos de espera eran algo elevados y llegados a ese punto interesa utilizar la otra alternativa de entrenamiento de la que hablaremos.

```
alumno2@jetson-2:~$ python3 open_images_downloader.py --max-images=1000 --
class-names "Human face" --data=data/faces
```

Para entrenar el modelo tenemos que ejecutar el script que se encuentra en la carpeta Python/training/detection/ssd. Recomendamos descargar el dataset en esta carpeta también. Para entrenar deberemos indicar dónde se encuentra el dataset, qué modelo queremos cargar, un batch size, nuestra Jetson es la versión de 4Gb, por lo que no recomendamos poner 4, más aún si se está usando conectado a un monitor. Por último indicar el número de épocas que esto será el parámetro con más impacto en la duración del entrenamiento. Para la Jetson y una sesión que no se alargue excesivamente en el tiempo, no deberíamos colocar más de 5 épocas, a no ser que dejemos el dispositivo computando toda una noche, aunque no lo recomendamos ya que seguramente el dispositivo sufra de sobrecalentamiento y no es bueno. Una vez más si se desea realizar más, podremos usar la segunda alternativa.

```
alumno2@jetson-2:~$ python3 train_ssd.py –data=data/faces –model-
dir=models/faces5 –batch-size=2 –epochs=5
```

Tras acabar el entrenamiento, lo siguiente que debemos hacer es convertir nuestro modelo a una extensión compatible con detectnet:

```
alumno2@jetson-2:~$ python3 onnx_export.py --model-dir=models/faces5
```

Para procesar nuestro modelo solo bastaría con asegurarnos que en la función de cargar el modelo, dentro del script **IAAR-SeguimientoModelo.py**, se encuentra la ruta correcta al modelo que acabamos de entrenar y exportar. Con esto podremos disfrutar de nuestro modelo entrenado.

Entrenamiento Google Colab

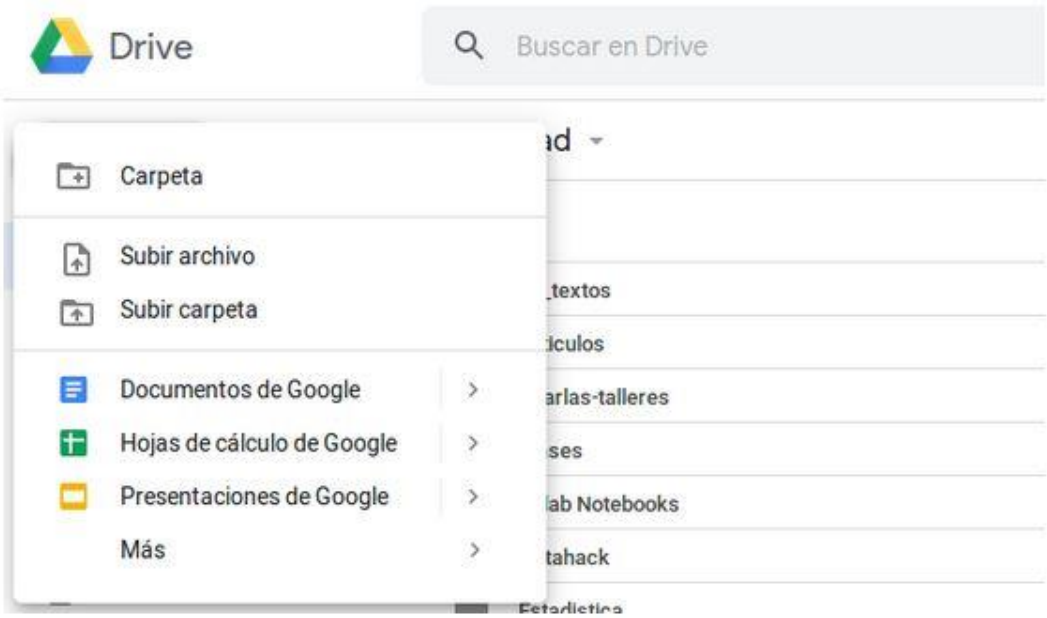
Como hemos mencionado la Jetson posee unas limitaciones, es por ello que decidimos buscar una alternativa para un mejor entrenamiento y con mayor potencia computacional, es por ello que usamos la plataforma Google Colab.

Colab es un servicio cloud, basado en los Notebooks de Jupyter, que permite el uso gratuito de los GPUs y TPUs de Google, como librerías como: Scikit-learn, Pytorch, TensorFlow, Keras y OpenCV. Todo ello con Python 2.7 y 3.6, que aún no está disponible para R y Scala.

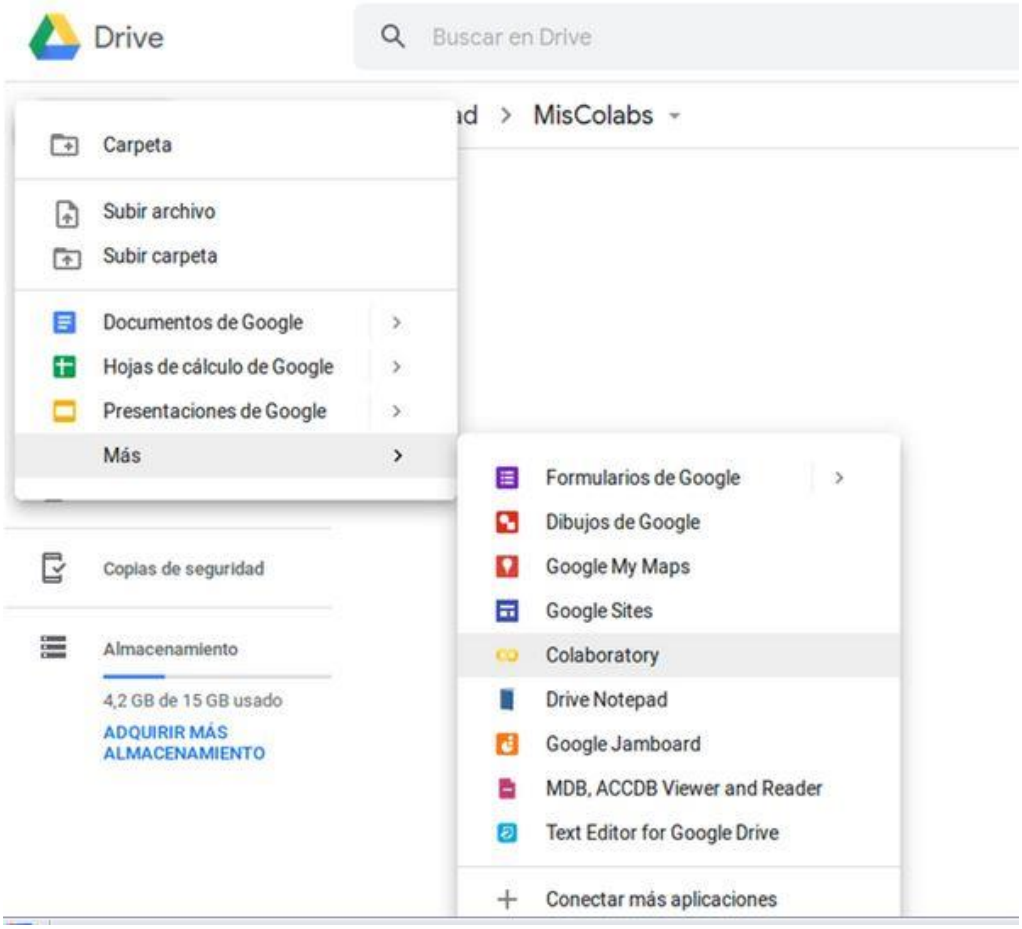
Aunque tiene algunas limitaciones, que pueden consultarse en su página de FAQ, es una herramienta ideal, no solo para practicar y mejorar nuestros conocimientos en técnicas y herramientas de Data Science, sino también para el desarrollo de aplicaciones (pilotos) de machine learning y Deep learning, sin tener que invertir en recursos hardware o del Cloud.

Con colab se pueden crear notebooks o importar los que ya tengamos creados, además de compartirlos y exportarlos cuando queramos. Esta fluidez a la hora de manejar la información también es aplicable a las fuentes de datos que usemos en nuestros proyectos, de modo que podremos trabajar con información contenida en nuestro propio Google Drive, unidad de almacenamiento local, github e incluso otros sistemas de almacenamiento cloud, como S3 de Amazon.

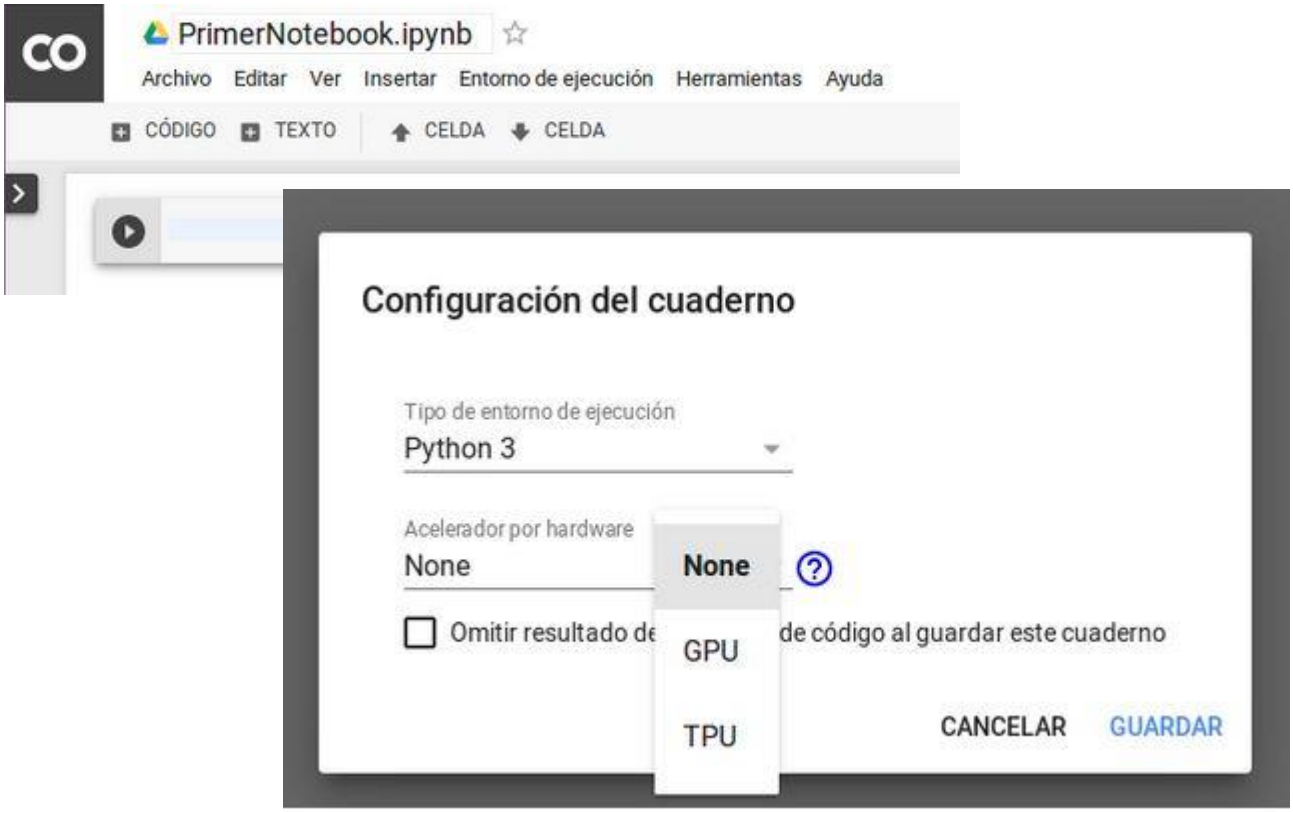
Para empezar a trabajar con colab, tendremos que tener una cuenta de Google y acceder al servicio Google Drive. Una vez dentro, le daremos a Nuevo > Carpeta, poniéndolo el nombre que queremos, por ejemplo: "IAAR-Colab".



Para crear nuestro primer Colab, entraremos dentro de la carpeta que hemos creado y daremos a Nuevo > Más > Colaboratory, a continuación, se abrirá un nuevo notebook:



Una vez hecho esto, hay que establecer el entorno de ejecución: pestaña Entorno de ejecución > Cambiar tipo de entorno de ejecución, tras lo que se abrirá la siguiente ventana:



En nuestro caso indicaremos Python 3 y GPU. Para cargar los datos usaremos la opción de Google drive, para ello, ejecutaremos el código que aparece en la imagen. Nos aparecerá una URL a la que debemos entrar para iniciar sesión con la cuenta que deseemos vincular, ya que puede ser una distinta a propietaria del Colab. Nos dará un código de verificación que deberemos pegar en el campo de texto y una vez realizado con éxito todos estos pasos se deberían de ver como en la siguiente imagen:

Ahora para poder realizar el entrenamiento de forma correcta solo deberemos subir a nuestro drive la carpeta ssd, del repositorio jetson-inference para poder trabajar con los scritps que se encuentran en dicha carpeta.

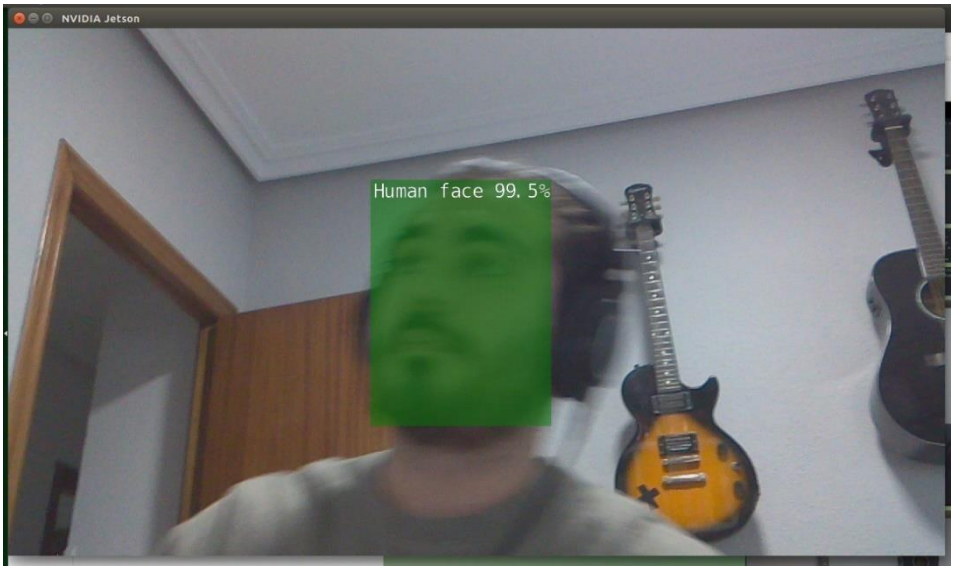
También deberemos descargar las imágenes de entrenamiento, pero subirlas al drive supone que el entrenamiento funcione de forma lenta, lo que recomendamos es descargarlas en la memoria interna de la máquina que "alquilamos" con Colab, aunque se borre al cerrar sesión, no supone un problema ya que tarda muy poco tiempo en descargar los datos.

Los comandos de consola Linux se ejecutan con "!" antes de cada línea. Teniendo esto en cuenta la forma de entrenar es exactamente la misma que en la jetson solo que cambiarán las rutas de los ficheros y directorios, por lo demás se ejecuta igual.

Una vez entrenado nuestro modelo, solo hay que descargarlo en la memoria local de la jetson y ponerlo en un lugar controlado, nosotros hemos creído oportuno colocarlo en la carpeta models dentro de ssd.

Resultado y conclusiones

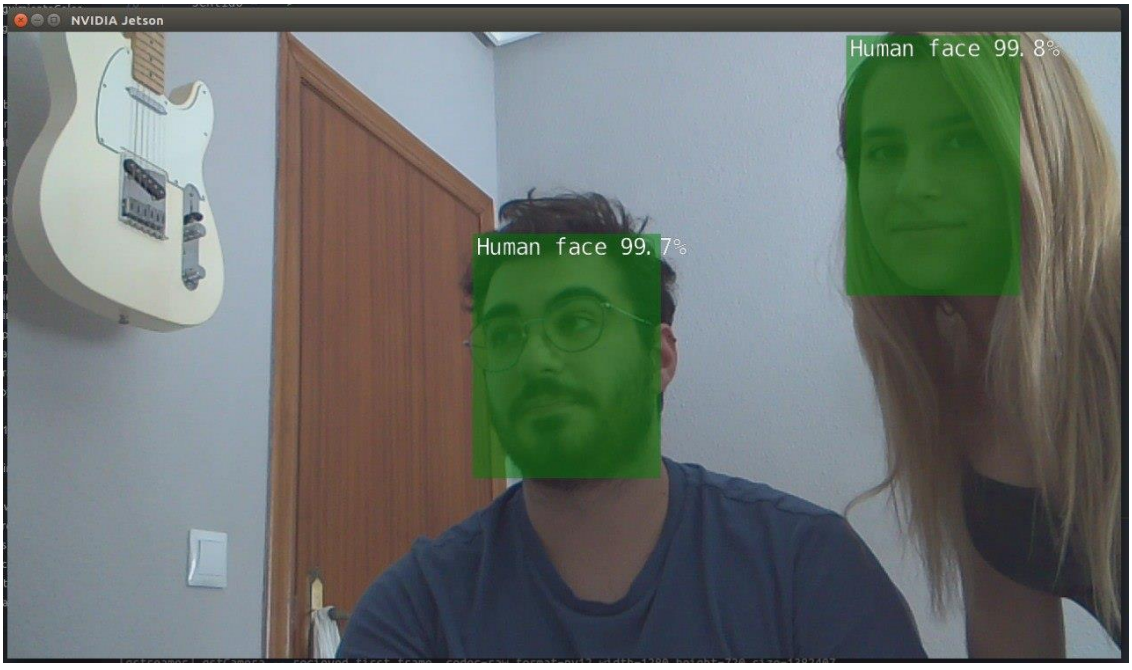
Uno de los problemas que encontramos en la fase de desarrollo era el cambio del modo sondeo a cambio de modo tracking. Debido a las vibraciones que generan los motores del servo, la imagen se distorsionaba y durante el cambio de posición encontraba otras detecciones, que cuando resultaban ser mayores que las de nuestra cara, hacían que el servo entrase en un bucle de cambio de posición, en otras palabras, se volvía loco. Esto se solucionó añadiendo un sleep al sondeo, así entre incremento de su posición horizontal y el siguiente, pasaba más tiempo y hacía un movimiento más suave, reduciendo las vibraciones, pero esto solo no fue suficiente, el cambio sustancial vino con un reentreno, es decir, cambiamos el modelo que estábamos usando.



Como hemos explicado, hemos usado Google Colab para realizar nuestros modelos, esto se debe a que primeramente entrenamos en la Jetson Nano, debido a las limitaciones técnicas hicimos una ejecución de un modelo con 5.000 imágenes y 2 generaciones, lo que no fue suficiente porque los ocurría, que en nuestra pared de fondo, tenemos colgada una guitarra y nos la detecta como cara humana:

Esto, además de significar que nuestro modelo no reconoce del todo bien a las personas, nos trajo un problema en el cual no caímos, qué debe hacer nuestro programa cuando detecte a dos personas y realmente sean dos personas.

El problema de qué hacer cuando hay dos personas tuvo fácil solución, comprobamos qué detección tiene mayor área y esa será a la que hagamos tracking. Esto lo pensamos porque damos por hecho que si se detecta una cara, y es un real positivo, la cara con mayor área, será la que se encuentre más cerca de la cámara, por lo que es la de mayor interés.



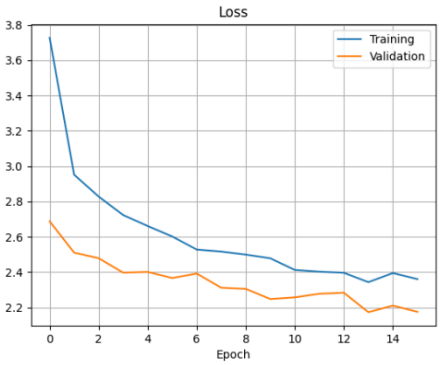
El problema de los falsos positivos no tuvo otra solución que hacer un mejor entrenamiento, esta vez desde Google Colab, con una cantidad mayor de datos y una cantidad mayor de épocas.

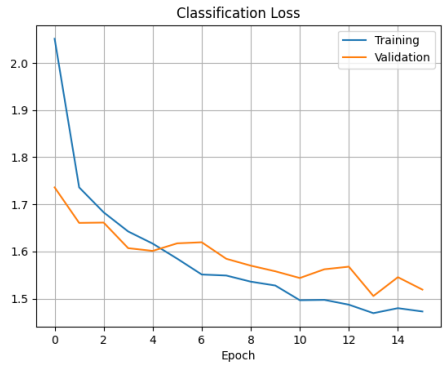
Realizamos varios entrenamientos, pero debido a un problema de tiempo de uso de Colab, no pudimos salvarlos todos, solo conseguimos ejecutar correctamente 3 de ellos. Colab tiene una limitación de tiempo de uso activo y a veces lo sobre pasamos.

Resumen de los entrenamientos

Modelo 15 épocas y 10.000 imágenes

El primer modelo que presentamos consta de: 10.000 imágenes de entrenamiento y 15 épocas. Hemos creado sus gráficas de evolución para poder entender cómo fue su proceso de entrenamiento:

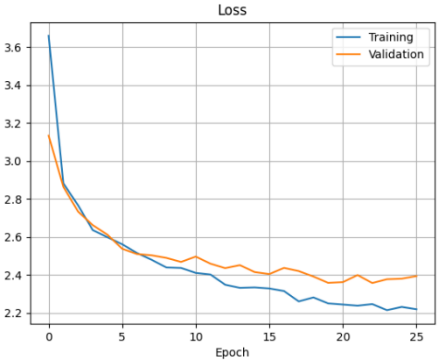


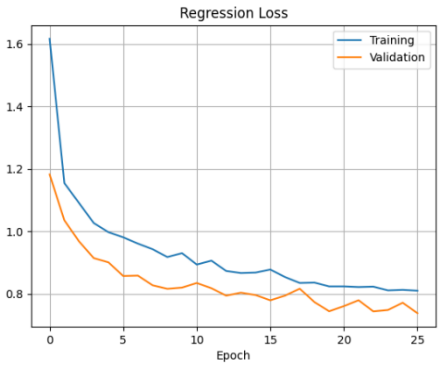


Este modelo ya nos aportaba unos resultado realmente mejorados, la detección de la guitarra como falso positivo solo ocurría en contadas veces, podría ser debido a que la guitarra tiene dos piezas circulares de una proximidad simular a la de los ojos, pusimos un pañuelo sobre ellos y efectivamente el error se corrigió, pero quisimos ser más técnicos y hacer un mejor modelo aún, por lo que decidimos aumentar el número de épocas.

Modelo 25 épocas y 10.000 imágenes

El segundo modelo consta de 25 épocas y 10.000 imágenes, en cuanto a la ejecución sí vimos una leve mejora en falsos positivos, pero en las gráficas observamos un "Overfitting" por lo que decidimos realizar otro entrenamiento con el objetivo de hacer uso de un modelo que nos dé buenos resultados y unas gráficas correctas.

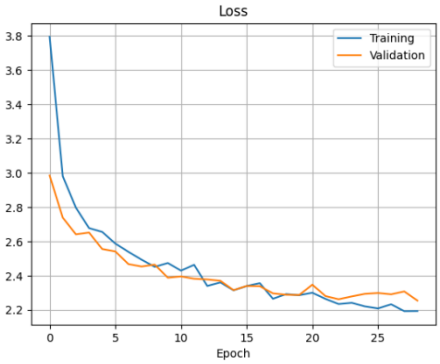


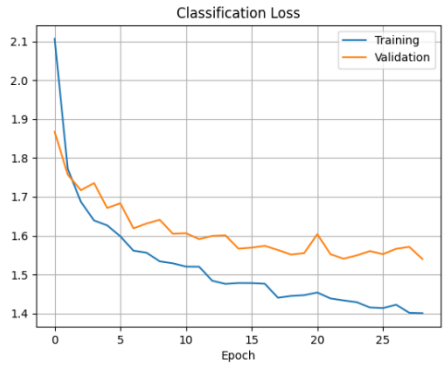


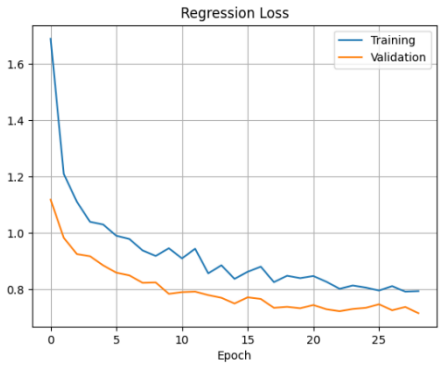
Modelo 30 épocas y 10.000 imágenes

Nuestro último modelo es el que hemos utilizado para mover el programa final. Se trata de un modelo con 30 épocas y 10.000 imágenes de entrenamiento. Nos solucionó por completo nuestro problema de falsos positivos, al menos en nuestra zona de desarrollo, no hemos podido comprobar los resultados en otros espacios porque no nos fue posible, aunque nos hubiera gustado poder recopilar datos y hacer una comparativo en distintas salas y generar gráficas comparativas de falsos positivos entre los distintos modelos.

Las gráficas de este modelo nos han parecido muy correctas y coherentes, por ello no quisimos realizar ningún otro entrenamiento con mayores números. Aunque podría haber sido interesante de cara a saber si realmente merece la pena invertir tanto tiempo en un entrenamiento y la mejoría en cuanto a resultados.







Conclusiones

Este proyecto nos ha resultado de lo más interesante, hemos abarcado varias técnicas estudiadas a lo largo del curso, conceptos estudiados en otras asignaturas como Aprendizaje Automático, Visión por computador, Sistemas de percepción, … Todo ello con la posibilidad de no tener que ser muy técnicos, pero sin dejar cerrada la posibilidad de que cada uno profundice hasta donde considere oportuno.

Al usar un gran número de imágenes y un alto número de épocas, nos hemos asegurado de que nuestro modelo aprenda con gran éxito a reconocer caras humanas, de hecho los porcentajes en las pruebas realizadas en nuestra sala, alcanzaban valores muy altos, cercanos al 100% en la mayoría de ocasiones y no detectábamos falsos positivos, o si los había era en momentos muy puntuales.

No hemos tenido en cuenta el uso de mascarillas, puesto que la descripción que nos dieron del proyecto era detección de caras y al usar la mascarilla prácticamente la mitad del rostro queda cubierto, además que de cogiendo como referencia modelos de desbloqueo facial como los que

puede haber en dispositivos móviles, han tenido serios problemas con el uso obligatorio de las mascarillas. No obstante, esto se podría haber solucionado con el uso de fotos de personas haciendo uso de estos elementos en sus caras y haciendo que el modelo aprenda también estos casos.

Relacionado con esto que acabamos de comentar, una mejora que se nos ocurrió fue añadir un criterio para seleccionar a una persona cuando se encontrase más de una, era la de buscar si había una de ellas sin mascarilla y hacer el tracking a esa persona. No fue posible desarrollarlo debido a la situación personal de poco tiempo disponible para dedicarlo a este proyecto (compaginar estudios con trabajo en una empresa). Pero hubiera sido una funcionalidad muy interesante.

A nivel personal, este trabajo me ha servido para tomármelo como primera toma de contacto para un futro TFG, ya que entra dentro del campo que me gustaría tratar y de hecho podría servir como base para poder comenzar.

Ubicaciones de archivos

En este apartado vamos a mostrar las ubicaciones de los diferentes ficheros y directorios mencionados a lo largo de la memoria.

Como directorio principal de trabajo creamos en el ESCRITORIO una carpeta llamada **IAAR**, donde se encuentra todo lo usado tanto para la entrega final como para pruebas.

Para ejecutar los scripts

Hemos creado una carpeta donde se encuentran todos los scripts finales para cada una de las funcionalidades implementadas y para el resultado final.

```
alumno2@jetson-2:~/Desktop/IAAR/IAAR-ProyectoFinal/Scripts$ ls
IAAR-ColocaServo.py    IAAR-EjecutaModelo.py  IAAR-MovimientoServo.py
IAAR-SeguimientoColor.py      IAAR-SeguimientoModelo.py    IAAR-VerCamara.py
IAAR-VerGrafica.py
```

El script que ejecuta el resultado final del proyecto es: **IAAR-SeguimientoModelo.py**

Fichero de jetson-inference

El directorio principal donde se descargan todas las cosas, donde se guardan los modelos por defecto y donde se encuentran los scripts para entrenar y descargar las imágenes, se encuentran en el directorio del repositorio de nvidia, que para una organización lo colocamos en un lugar concreto que más tarde no cambiamos para prevenir posibles fallos.

Se encuentra en **~/Desktop/IAAR/Detección-objetos/jetson-inference**

Dentro de esta carpeta podemos encontrar muchas más, pero la que nos interesa más es la ruta **/python/training/detection/ssd/** y el contenido que se encuentra en esta carpeta ya ha sido explicado en los diferentes apartados de este informe. En esta última carpeta se encuentran todos los modelos ya entrenados, pero existe un duplicado, que de hecho es el que se está usando en el script final, en la carpeta: **/Desktop/IAAR/IAAR-ProyectoFinal/models**.

Bibliografía

Primera configuración con JetPack: https://cutt.ly/0mVJhzv

Hello World configuración: https://cutt.ly/imVJbAi

Instalando entorno Python: https://cutt.ly/6mVJP5h

Instalación de Matplotlib y Numpy: https://cutt.ly/vmVJH8C

Instalación de OpenCV: https://cutt.ly/FmVJZbg

Prueba de Cámara Pi en Jetson con OpenCV: https://cutt.ly/8mVJMB0

Trackeando objeto mediante contornos: https://cutt.ly/9mVJ4kU

Montando Servo: https://cutt.ly/tmVKqwL

Controlando el servo: https://cutt.ly/1mVKtIg

Trackeando objeto con servo y OpenCV: https://cutt.ly/SmVKuTx

Instalando herramientas NVIDIA detecció objetos: https://cutt.ly/imVKg54

Introducción Deep learning Jetson Nano: https://cutt.ly/4mVKluD

Entrenando modelo de reconocimiento de objetos: https://cutt.ly/NmVKmLD

Repositorio de jetson-inference: https://cutt.ly/hmVKRGs

Google Colab: https://cutt.ly/UmVKPME

Curso, Aprende inteligencia artificial en Jetson Nano: https://cutt.ly/EmVKG1V


ETSI
ESCUELA TÉCNICA
SUPERIOR DE INGENIERÍA
uhu.es