Jetson Xavier AGX: Guia de desarrollo del proyecto

Proceso: Instalación y arranque

En este documento se describen los diferentes procesos para realizar el proyecto sobre las `Jetson Xavier AGX`. El objetivo es conocer y entender cómo podemos implementar en primer lugar procesos de inferencia de redes neuronales y en segundo lugar cómo utilizarlos en aplicaciones específicas. Las aplicaciones que utilizaremos y desarrollaremos estarán centradas principalmente en el uso y tratamiento de imágenes.

Antes de empezar con el proyecto debemos conectarnos a la máquina asignada y descargarnos el proyecto. El usuario para acceder es `jxav` y la contraseña `jxav`. Una vez hemos entrado abrimos un terminal para descargarnos y configurar el proyecto:

```
$ cd ~/proyecto1
$ git clone --recursive --depth=1 https://github.com/dusty-nv/jetson-inference
```

Ten en cuenta que se te habrá asignado un directorio de trabajo. En el código anterior antes de clonar el proyecto hemos entrado en el directorio `proyecto1`. En tu caso, solicita qué directorio se te asigna y utilizalo.

Ahora creamos un directorio de trabajo donde almacenaremos todos nuestros ficheros.

```
$ mkdir misdatos
```

Seguidamente lanzamos el contenedor con todo el proyecto. La primera vez que lo lanzamos se descargará el contenedor. Fíjate que estamos enlazando el directorio que hemos creado con el contenedor. Por tanto, ese directorio será accesible tanto desde el contenedor como desde fuera del contenedor.

```
$ cd jetson-inference
$ docker/run.sh --volume ~/proyecto1/misdatos:/misdatos
```

Una vez descargado ya estamos dentro del contenedor y podemos empezar el proyecto. Cada vez que entremos en la máquina deberemos entrar en el contenedor/proyecto de nuevo con el código anterior, lo tienes a continuación:

```
$ cd ~/proyecto1/jetson-inference
$ docker/run.sh --volume ~/proyecto1/misdatos:/misdatos
```

Acuérdate que en tu caso puede ser otro directorio a `proyecto1`.

En todo el proyecto vamos a utilizar objetos que nos ayudarán a realizar la inferencia de diferentes modelos. Toda la documentación de estos modelos está disponible en:

https://github.com/dusty-nv/jetson-inference

https://rawgit.com/dusty-nv/jetson-inference/master/docs/html/python/jetson.inference.html#poseNet

Proceso: Clasificación de imágenes

Subproceso: Clasificación de imágenes con ImageNet

El objeto (aplicación) `imageNet` acepta una imagen en la entrada y nos devuelve la probabilidad de cada clase que identifica en la imagen. Esta aplicación acepta diferentes modelos entrenados con la base de datos `ImageNet ILSVRC` que detecta 1000 tipos de objetos diferente. Vamos a utilizar la aplicación (ya sea en su versión en C++ o en su versión en Python).

Todos los programas están en el directorio `/jetson-inference/build/aarch64/bin`. Ten presente que debes ejecutar todo el código desde dentro del contenedor, es decir, desde el terminal en el que has lanzado el contenedor. Si abres otro terminal este no estará ejecutándose dentro del contenedor.

```
$ cd /jeton-inference/build/aarch64/bin
```

Para ejecutar en C++:

```
$ ./imagenet images/orange_0.jpg /misdatos/output_0.jpg
```

Para ejecutar en Python:

```
./imagenet.py images/orange_0.jpg /misdatos/output_0.jpg
```

En ambos casos se realizará la inferencia de la imagen de entrada y se generará la imagen de salida sobreponiendo la predicción que ha realizado el modelp. Ten en cuenta que la primera vez que ejecutamos un modelo este se optimiza para `TensorRT`, tardando un par de minutos. Una vez optimizado ya se ejecutará mucho más rápido.

Ten en cuenta que el resultado de la clasificación se almacena en el directorio `/misdatos`, el cual puedes acceder desde el *host* (fuera del contenedor) para ver el resultado. Se recomienda abrir la carpeta en el host para poder visualizar todos los resultados generados.

Mira también la salida por consola al realizar una inferencia. Nos indica tiempos de inferencia.

En el directorio `images` tienes varias imágenes de prueba. Puedes probarlas y ver el resultado.

La aplicación permite utilizar otros modelos de redes neuronales. Puedes ver todos los modelos disponibles ejecutando:

```
./imagenet --help
```

Antes has utilizado el modelo `GoogleNet` por defecto. Con la opción `--network` prueba ahora otros modelos y mira el impacto en la precisión.

Podemos lanzar también la inferencia sobre un video. Para ello, nos bajamos el video y ejecutamos la aplicación:

```
$ wget https://nvidia.box.com/shared/static/tlswont1jnyu3ix2tbf7utaekpzcx4rc.mkv \
       -O /misdatos/jellyfish.mkv
```

```
# C++
$ ./imagenet --network=resnet-18 /misdatos/jellyfish.mkv /misdatos/jellyfish_resnet18.mkv

# Python
$ ./imagenet.py --network=resnet-18 /misdatos/jellyfish.mkv \
/misdatos/jellyfish_resnet18.mkv
```

Para ver un video podemos utilizar la aplicación `video-viewer`. Ten en cuenta que esta aplicación está disponible dentro del docker (no fuera).

```
video-viewer /misdatos/jellyfish_resnet18.mkv
```

Subproceso: Codificar un programa de reconocimiento de imágenes en Python

Creamos y editamos un fichero fuente de nuestro programa (en el directorio `/misdatos`):

```
$ gedit my-recognition.py
```

En la primera línea del fichero indicamos el `bash` que se utilizará para ejecutarlo:

```
#!/usr/bin/python3
```

Después importamos los módulos necesarios:

```
import jetson_inference
import jetson_utils
import argparse
```

A continuación, gestionamos los parámetros de entrada de nuestra aplicación:

```
# parse the command line
parser = argparse.ArgumentParser()
parser.add_argument("filename", type=str, help="filename of the image to process")
parser.add_argument("--network", type=str, default="googlenet", help="model to use, can be:
googlenet, resnet-18, etc. (see --help for others)")
args = parser.parse_args()
```

Ahora cargamos la imagen que se nos ha pasado por argumento:

```
img = jetson_utils.loadImage(args.filename)
```

La imagen será un objeto `jetson_utils.cudaImage` con los siguientes atributos:

```
<jetson.utils.cudaImage>
  .ptr      # memory address (not typically used)
  .size     # size in bytes
  .shape    # (height,width,channels) tuple
  .width    # width in pixels
  .height   # height in pixels
  .channels # number of color channels
  .format   # format string
  .mapped   # true if ZeroCopy
```

Ahora cargamos el modelo con el objeto `imageNet`. La siguiente función cargará el modelo de clasificación con `TensorRT`. Todos los modelos de clasificación disponibles están pre-entrenados con el dataset `ImageNet ILSVRC`, que reconoce 1000 clases diferentes de objetos.

```
# load the recognition network
net = jetson_inference.imageNet(args.network)
```

Una vez cargada la imagen y el modelo, solamente nos queda realizar la inferencia, para ello utilizamos la función `imageNet.Classify()`.

```
class_idx, confidence = net.Classify(img)
```

La función anterior realiza la inferencia utilizando `TensorRT`. Devuelve una tupla, siendo el primer elemento el índice del objeto (de 0 a 999) y el segundo el valor de confianza del resultado (de 0 a 1).

Solamente nos queda mostrar el resultado. Para ello, obtenemos la descripción de la clase y lo mostramos por pantalla:

```
# find the object description
class_desc = net.GetClassDesc(class_idx)

# print out the result
print("image is recognized as '{:s}' (class #{:d}) with {:f}%
confidence".format(class_desc, class_idx, confidence * 100))
```

Ya hemos terminado, solo nos queda guardar el fichero y realizar pruebas de inferencia:

Primero cargamos tres imágenes:

```
$ cd /misdatos
$ wget https://github.com/dusty-nv/jetson-inference/raw/master/data/images/black_bear.jpg
$ wget https://github.com/dusty-nv/jetson-inference/raw/master/data/images/brown_bear.jpg
$ wget https://github.com/dusty-nv/jetson-inference/raw/master/data/images/polar_bear.jpg
```

Y ahora lanzamos nuestra aplicación:

```
$ chmod +x /misdatos/my-recognition.py
$ /misdatos/my-recognition.py polar_bear.jpg
$ /misdatos/my-recognition.py brown_bear.jpg
$ /misdatos/my-recognition.py black_bear.jpg
```

Prueba otros modelos de red con el parámetro `--network`

Subproceso: Codificar un programa de reconocimiento de imágenes en C++

En esta actividad, igual que hemos hecho anteriormente, vamos a crear un programa en C++ que realizará la misma inferencia. Editamos el nuevo fichero para nuestro programa (lo ubicamos en `/misdatos`):

```
$ gedit my-recognition.cpp
```

En primer lugar, incluimos los ficheros de cabecera necesarios.

```
// include imageNet header for image recognition
#include <jetson-inference/imageNet.h>

// include loadImage header for loading images
#include <jetson-utils/loadImage.h>
```

Ahora vamos a codificar parte de la función `main`, encargada de gestionar los argumentos que pasaremos a nuestra aplicación:

```
// main entry point
int main( int argc, char** argv )
{
        // a command line argument containing the image filename is expected,
        // so make sure we have at least 2 args (the first arg is the program)
        if( argc < 2 )
        {
                printf("my-recognition:  expected image filename as argument\n");
                printf("example usage:   ./my-recognition my_image.jpg\n");
                return 0;
        }

        // retrieve the image filename from the array of command line args
        const char* imgFilename = argv[1];
```

A continuación, cargamos la imagen a partir del argumento que se nos habrá pasado por línea de comandos:

```
        // these variables will store the image data pointer and dimensions
        uchar3* imgPtr = NULL;   // shared CPU/GPU pointer to image
        int imgWidth   = 0;      // width of the image (in pixels)
        int imgHeight  = 0;      // height of the image (in pixels)

        // load the image from disk as uchar3 RGB (24 bits per pixel)
        if( !loadImage(imgFilename, &imgPtr, &imgWidth, &imgHeight) )
        {
                printf("failed to load image '%s'\n", imgFilename);
                return 0;
        }
```

Ahora utilizamos la clase `imageNet` para cargar el modelo de red neuronal. En este caso se utiliza el modelo `GoogleNet`. Como ves, puedes modificar el modelo en el propio código.

```
        // load the GoogleNet image recognition network with TensorRT
        // you can use imageNet::RESNET_18 to load ResNet-18 instead
        imageNet* net = imageNet::Create("googlenet");

        // check to make sure that the network model loaded properly
        if( !net )
        {
```

```
                printf("failed to load image recognition network\n");
                return 0;
        }
```

Ya solo nos queda realizar la inferencia y obtener el resultado:

```
        // this variable will store the confidence of the classification (between 0 and 1)
        float confidence = 0.0;

        // classify the image, return the object class index (or -1 on error)
        const int classIndex = net->Classify(imgPtr, imgWidth, imgHeight, &confidence);
```

El resultado lo mostramos por pantalla:

```
        // make sure a valid classification result was returned
        if( classIndex >= 0 )
        {
                // retrieve the name/description of the object class index
                const char* classDescription = net->GetClassDesc(classIndex);

                // print out the classification results
                printf("image is recognized as '%s' (class #%i) with %f%% confidence\n",
                           classDescription, classIndex, confidence * 100.0f);
        }
        else
        {
                // if Classify() returned < 0, an error occurred
                printf("failed to classify image\n");
        }
```

Queda cerrar la función `main()`:

```
        // free the network's resources before shutting down
        delete net;

        // this is the end of the example!
        return 0;
}
```

Vamos a crear ahora un fichero para compilar la aplicación. Edita el fichero `CMakeLists.txt` (ubícalo en `/misdatos`) e introduce el siguiente texto:

```
# require CMake 2.8 or greater
cmake_minimum_required(VERSION 2.8)

# declare my-recognition project
project(my-recognition)

# import jetson-inference and jetson-utils packages.
# note that if you didn't do "sudo make install"
# while building jetson-inference, this will error.
find_package(jetson-utils)
find_package(jetson-inference)

# find VPI package (optional)
find_package(VPI 2.0)

# CUDA is required
find_package(CUDA)

# add directory for libnvbuf-utils to program
link_directories(/usr/lib/aarch64-linux-gnu/tegra)
```

```
# compile the my-recognition program
cuda_add_executable(my-recognition my-recognition.cpp)

# link my-recognition to jetson-inference library
target_link_libraries(my-recognition jetson-inference)
```

Vamos ahora a compilar la aplicación:

```
$ cd /misdatos
$ cmake .
$ make
```

Por último probamos diferentes imágenes:

```
$ ./my-recognition polar_bear.jpg
$ ./my-recognition brown_bear.jpg
$ ./my-recognition black_bear.jpg
```

Subproceso: Reconocimiento de imágenes asociado a la cámara

Podemos conectar el proceso de inferencia con `imagenet` (tanto en C++ como en Python) a la cámara. Acuérdate que imagenet está ubicado en `/jetson-inference/build/aarch64/bin`

En C++:

```
$ ./imagenet /dev/video0                # V4L2 camera
$ ./imagenet /dev/video0 output.mp4     # save to video file
```

En Python:

```
$ ./imagenet.py /dev/video0             # V4L2 camera
$ ./imagenet.py /dev/video0 output.mp4  # save to video file
```

Podemos ver en la ventana de la cámara (en la barra superior) los frames por segundo obtenidos por la aplicación. Acuérdate que puedes probar diferentes modelos de red neuronal, cada uno de ellos obtendrá una precisión y velocidad (imágenes por segundo) distintas. Prueba a ver.

Actividad sugerida: Utilizando el objeto imagenet, desarrolla una aplicación, ya sea en Python o en C++, que monitorice la cámara y active una alarma (un mensaje por pantalla) ante la presencia de una persona en el campo de visión de la cámara.

Subproceso: Clasificación de múltiples objetos

Existen modelos de clasificación múltiple, capaces de reconocer diferentes objetos simultáneamente. La diferencia con los modelos de clasificación de una única clase radica en la utilización de una capa de activación al final de la red de tipo `sigmoide`, a diferencia de la capa `softmax`. Tenemos disponible el modelo `resnet18-taggin-voc` de clasificación múltiple entrenado con el datasaset `Pascal VOC`. Vamos a probarlo.

```
# C++
$ ./imagenet --model=resnet18-tagging-voc --topK=0 --threshold=0.25 \
           "images/object_*.jpg" /misdatos/tagging_%i.jpg

# Python
$ ./imagenet.py --model=resnet18-tagging-voc --topK=0 --threshold=0.25 \
            "images/object_*.jpg" /misdatos/tagging_%i.jpg
```

Fíjate cómo utilizamos `--topK=0` indicando que se muestren todas las clases que excedan el umbral indicado con `--threshold`.

Los cambios en el código radican en el uso de la función `Classify` tanto en Python como en C++:

C++:

```
imageNet::Classifications classifications; // std::vector<std::pair<uint32_t, float>>
(classID, confidence)

if( net->Classify(image, input->GetWidth(), input->GetHeight(), classifications, topK) < 0
)
        continue;

for( uint32_t n=0; n < classifications.size(); n++ )
{
        const uint32_t classID = classifications[n].first;
        const char* classLabel = net->GetClassLabel(classID);
        const float confidence = classifications[n].second * 100.0f;

        printf("imagenet:  %2.5f%% class #%i (%s)\n", confidence, classID, classLabel);

}
```

Python:

```
predictions = net.Classify(img, topK=args.topK)

for n, (classID, confidence) in enumerate(predictions):
   classLabel = net.GetClassLabel(classID)
   confidence *= 100.0
   print(f"imagenet:  {confidence:05.2f}% class #{classID} ({classLabel})")
```

Proceso: Detección de objetos

Subproceso: Detección de objetos con DetectNet

A diferencia de los modelos anteriores donde se obtenían probabilidades representando toda la imagen de entrada, ahora vamos a centrarnos en la detección de objetos, ubicándolos espacialmente en la imagen. Por tanto, vamos a realizar la inferencia de qué objeto es y en qué coordenadas (*bounding box*) se encuentra el objeto dentro de la imagen. Los modelos de detección de objetos son capaces de encontrar diferentes objetos por cada imagen.

El objeto (aplicación) `detectNet` recibe una imagen como entrada y devuelve una lista de coordenadas de los marcos donde encuentra objetos, indicando para cada uno de ellos su clase y su nivel de confianza. Tenemos disponible tanto la versión en Python como la versión en C++. Por defecto se utiliza el modelo `SSD-Mobilenet-v2`, entrenado con el dataset `MS COCO`, el cual realiza inferencia en tiempo real en la `Jetson` utilizando `TensorRT`.

Vamos en primer lugar a detectar objetos a partir de imágenes:

```
# C++
$ ./detectnet --network=ssd-mobilenet-v2 images/peds_0.jpg /misdatos/output.jpg

# Python
$ ./detectnet.py --network=ssd-mobilenet-v2 images/peds_0.jpg /misdatos/output.jpg
```

Acuérdate que la primera vez que utilizamos un modelo, este se compila y optimiza con `TensorRT`, pudiendo tardar un par de minutos.

Podemos procesar también un directorio de imágenes:

```
# C++
./detectnet "images/peds_*.jpg" images/test/peds_output_%i.jpg

# Python
./detectnet.py "images/peds_*.jpg" images/test/peds_output_%i.jpg
```

También podemos procesar videos:

```
# Download test video
wget https://nvidia.box.com/shared/static/veuuimq6pwvd62p9fresqhrrmfqz0e2f.mp4 \
     -O /misdatos/pedestrians.mp4

# C++
./detectnet /misdatos/pedestrians.mp4 /misdatos/pedestrians_ssd.mp4

# Python
./detectnet.py /misdatos/pedestrians.mp4 /misdatos/pedestrians_ssd.mp4
```

O incluso asociar la inferencia a la cámara:

```
# en C++
$ ./detectnet /dev/video0                # V4L2 camera
$ ./detectnet /dev/video0 output.mp4     # save to video file
```

```
$ En Python
$ ./detectnet.py /dev/video0             # V4L2 camera
$ ./detectnet.py /dev/video0 output.mp4  # save to video file
```

Podemos utilizar diferentes modelos, algunos de ellos especializados en subconjuntos de objetos:

Model	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)
TAO PeopleNet	`peoplenet`	`PEOPLENET`	person, bag, face
TAO PeopleNet (pruned)	`peoplenet-pruned`	`PEOPLENET_PRUNED`	person, bag, face
TAO DashCamNet	`dashcamnet`	`DASHCAMNET`	person, car, bike, sign
TAO TrafficCamNet	`trafficcamnet`	`TRAFFICCAMNET`	person, car, bike, sign
TAO FaceDetect	`facedetect`	`FACEDETECT`	face

Pruébalos utilizando el argumento `--network`. Fíjate en la precisión obtenida.

Puedes ajustar el umbral de detección con el argumento `--threshold`

Subproceso: Codificar un programa de detección de objetos en Python

En este punto del proyecto vamos a desarrollar un programa para la detección de objetos en tiempo real, codificado en Python, y tomando la webcam como entrada de imágenes. Para ello, utilizamos el objeto `detectNet`.

Vamos a crear el fichero en el directorio `/misdatos`:

```
gedit my-detection.py
```

Lo primero que debemos hacer siempre en Python es importar los módulos que vamos a utilizar.

```
from jetson_inference import detectNet
from jetson_utils import videoSource, videoOutput
```

Como véis importamos el módulo de detección y dos módulos para gestionar la entrada de video y la salida. A continuación creamos una instancia del objeto `detectNet` indicándole el modelo a utilizar y el umbral.

```
net = detectNet("ssd-mobilenet-v2", threshold=0.5)
```

Podemos utilizar los modelos comentados anteriormente y mostrados en una tabla ("`ssd-mobilenet-v1`", "`ssd-mobilenet-v2`", ...). Podemos jugar con el umbral para ver la efectividad y nivel de seguridad del modelo.

A continuación, abrimos la cámara y creamos una salida de video.

```
camera = videoSource("/dev/video0")
display = videoOutput("display://0") # 'my_video.mp4' for file
```

Por último, creamos un bucle donde vamos a realizar el proceso iterativo.

```
while display.IsStreaming():
    img = camera.Capture()

    if img is None: # capture timeout
        continue

    detections = net.Detect(img)

    display.Render(img)
    display.SetStatus("Object Detection | Network {:.0f} FPS".format(net.GetNetworkFPS()))
```

En primer lugar capturamos una imagen (*frame*) de la cámara. Esta imagen se cargará en la GPU. Seguidamente, la imagen se procesa con el modelo, realizándose la inferencia. El modelo devuelve una lista de detecciones. El modelo también sobrepone a la imagen las detecciones. Por último, mostramos el resultado por la salida de video.

Podemos añadir en el bucle un *print* para que nos muestre las detecciones por consola:

```
print(detections)
```

Subproceso: Modelos TAO de detección

NVIDIA dispone de un toolkit, denominado TAO, que permite generar modelos de alta precisión. Incluye también modelos de detección de alta resolución y precisión, así como modelos cuantizados (a INT8) y ya reducidos por *pruning*. Estos modelos se basan en la arquitectura `DetectNet_v2`. Vamos, en este momento, a utilizarlos.

Los modelos disponibles que vamos a utilizar son:

- TAO PeopleNet, "`peoplenet`", detecta personas, bolsas y caras.
- TAO PeopleNet reducido, "`peoplenet-pruned`", detecta personas, bolsas y caras
- TAO DashCamNet, "`dashcamnet`", detecta personas, vehículos, bicicletas, señales de tráfico
- TAO TrafficCamNet, "`trafficcamnet`", detecta personas, vehículos, bicicletas, señales de tráfico
- TAO FaceDetect, "`facedetect`", detecta caras.

Échale un vistazo a los modelos TAO y el toolkit de TAO: https://developer.nvidia.com/tao-toolkit

Vamos con algunos de ellos.

`PeopleNet` es un modelo de alta resolución (950x544) con un 90% de precisión para detectar personas, bolsas y caras. Está basado en la arquitectura DNN `DetectNet_v2` y se basa en `ResNet-34`. Utiliza precisión INT8 en plataformas que lo soporten, en caso contrario utiliza FP16. Podemos utilizar también el modelo reducido (*pruned*). Para lanzarlo:

```
$ wget https://nvidia.box.com/shared/static/veuuimq6pwvd62p9fresqhrrmfqz0e2f.mp4 -O
/misdatos/pedestrians.mp4

# C++
$ detectnet --model=peoplenet /misdatos/pedestrians.mp4 /misdatos/pedestrians_peoplenet.mp4

# Python
$ detectnet.py --model=peoplenet /misdatos/pedestrians.mp4
/misdatos/pedestrians_peoplenet.mp4
```

Para lanzar el modelo reducido utilizamos `peoplenet-pruned` como modelo. Lanza los dos modelos y fíjate en las diferencias de prestaciones (latencia del modelo e imágenes por segundo).

`DashCamNet` es un detector de entrada 960x544 basado en `DetectNet_v2` y `ResNet-34`. Está especializado para detectar peatones, vehículos y señales de tráfico.

Por último, `FaceDetect` es un modelo TAO para detectar caras.

```
# C++
$ detectnet --model=facedetect "images/humans_*.jpg" /misdatos/facedetect_%i.jpg

# Python
$ detectnet.py --model=facedetect "images/humans_*.jpg" /misdatos/facedetect_%i.jpg
```

Recuerda que puedes utilizar la webcam como entrada para la inferencia.

Vamos cerrando el tema de detección y seguimiento de objetos. Un aspecto interesante es aplicar filtros temporales para suavizar la detección y hacerla más estable. Habrás notado que por ejemplo,

las personas son localizadas pero hay frames en los que dejan de ser detectados. Podemos utilizar la opción de `--tracking` del objeto `detectnet` para aplicar un filtro espacial. Básicamente con esta opción se realiza un análisis de las áreas detectadas entre frames, calculando la métrica `IOU` (intersección sobre la operación de unión). Utiliza la opción y mira de nuevo qué tal funciona el seguimiento de objetos.

Actividad sugerida: Utilizando el objeto imagenet, desarrolla una aplicación, ya sea en Python o en C++, que monitorice la cámara y active una alarma (un mensaje por pantalla) cuando detecte personas, indicando cuántas hay en el campo de visión de la cámara.

Proceso: Segmentación semántica

La segmentación semántica se basa en el reconocimiento de objetos. Sin embargo, a diferencia de las técnicas anteriores que se basan en áreas de la imágen, en la segmentación semántica la clasificación se realiza a nivel de pixel. Por tanto, la salida de la red neuronal es una imagen donde cada píxel indica el objeto al que pertenece. En otras palabras, cada píxel se etiqueta. Estos modelos son útiles en sistemas de percepción del entorno.

Para realizar la segmentación semántica utilizamos el objeto (aplicación) `segNet`. Este objeto acepta imágenes 2D y obtiene una imagen del mismo tamaño, denominado máscara. Cada píxel de la máscara corresponde a una clase del objeto que se clasificó. Tenemos tanto una versión en python como una versión en C++ disponible. De la misma forma que en anteriores actividades, podemos tener como fuente de datos tanto imágenes en ficheros, videos y la propia webcam.

La clase `segNet` acepta los siguientes modelos:

Dataset	Resolución	Argumento CLI	Precisión	Jetson Nano	Jetson Xavier
Cityscapes	512x256	`fcv-resnet18-cityscapes-512x256`	83.3%	48 FPS	480 FPS
Cityscapes	1024x512	`fcn-resnet18-cityscapes-1024x512`	87.3%	12 FPS	175 FPS
Cityscapes	2048x1024	`fcn-resnet18-cityscapes-2048x1024`	89.6%	3 FPS	47 FPS
DeepScene	576x320	`fcn-resnet18-deepscene-576x320`	96.4%	26 FPS	360 FPS
DeepScene	864x480	`fcn-resnet18-deepscene-864x480`	96.9%	14 FPS	190 FPS
Multi-Human	512x320	`fcn-resnet18-mhp-512x320`	86.5%	34 FPS	370 FPS
Multi-Human	640x360	`fcn-resnet18-mhp-640x360`	87.1%	23 FPS	325 FPS
Pascal VOC	320x320	`fcn-resnet18-voc-320x320`	85.9%	45 FPS	508 FPS
Pascal VOC	512x320	`fcn-resnet18-voc-512x320`	88.5%	34 FPS	375 FPS
SUN RGB-D	512x400	`fcn-resnet18-sun-512x400`	64.3%	28 FPS	340 FPS
SUN RGB-D	640x512	`fcn-resnet18-sun-640x512`	65.1%	17 FPS	224 FPS

Vamos a lanzar algunas pruebas:

Cityscapes:

```
$ ./segnet –network=fcn-resnet18-cityscapes images/city_0.jpg /misdatos/city_0_semseg.jpg
```

DeepSceene (especializado en pistas y senderos de bosques y vegetación, para ayudar a robots en esos entornos):

```
$ ./segnet –network=fcn-resnet18-deepscene images/trail_0.jpg /misdatos/trail_0_semseg.jpg
$ ./segnet –visualize=mask –network=fcn-resnet18-deepscene images/trail_0.jpg
/misdatos/trail_0_semseg_mask.jpg
```

Multi-Human parsing (detecta partes del cuerpo como brazos, piernas, cabeza y diferentes prendas):

```
$ ./segnet –network=fcn-resnet18-mhp images/humans_0.jpg /misdatos/humans_0_semseg.jpg
```

Pascal VOC (especializado en personas, animales, vehículos y objetos de interior de casas):

```
$ ./segnet –network=fcn-resnet18-voc images/object_0.jpg /misdatos/object_0_semseg.jpg
```

Sun RGB-D (objetos de interior de casas y oficinas):

```
$ ./segnet –network=fcn-resnet18-sun images/room_0.jpg /misdatos/room_0_semseg.jpg
```

Habrás visto que puedes sacar solamente la máscara o la imagen. También tienes más imágenes disponibles de cada tipología.

Recuerda que puedes conectar la cámara al proceso de segmentación semántica:

```
$ ./segnet /dev/video0
```

Proceso: Estimación de la postura corporal (pose estimation)

La estimación de la postura corporal consiste en localizar diferentes partes del cuerpo humano, formando un esqueleto. La estimación de la postura corporal tiene aplicación en temas de identificación de gestos, el lenguaje corporal, la corrección de la postura y en interfaces hombre/máquina. Hay disponibles modelos pre-entrenados para detectar la estimación de la postura del cuerpo humano y de la mano. Puede detectar múltiples objetos (personas, manos) por frame.

Para trabajar con la estimación de la postura tenemos el objeto (aplicación) `poseNet`. Este objeto acepta una imagen como entrada y devuelve una lista de objetos. Cada objeto contiene una lista de puntos detectados con su ubicación y sus enlaces con otros puntos. Tenemos dos versiones, una en Python y la otra en C++. Podemos utilizar como entrada tanto imágenes a partir de ficheros, videos o la propia webcam.

Si ejecutamos la aplicación con `--help` podremos ver todas sus opciones:

```
$ # C++
$ posenet –help

$ # Python
$ posenet.py –help
```

Vamos ahora con algunas pruebas a partir de imágenes:

```
# C++
$ ./posenet "images/humans_*.jpg" /misdatos//pose_humans_%i.jpg

# Python
$ ./posenet.py "images/humans_*.jpg" /misdatos/pose_humans_%i.jpg
```

Ahora, a partir de la webcam:

```
# C++
$ ./posenet /dev/video0

# Python
$ ./posenet.py /dev/video0
```

En los ejemplos anteriores hemos utilizado el modelo `resnet18-body`, especializado en estimación de la postura corporal. Hay otro modelo para estimación de la postura corporal: `densenet121-body`. Pruébalo.

Vamos ahora a utilizar el modelo `resnet18-hand`, el cual está especializado en la posición de la mano:

```
# C++
$ ./posenet –network=resnet18-hand /dev/video0

# Python
$ ./posenet.py –network=resnet18-hand /dev/video0
```

Podemos trabajar con la información que obtenemos de la postura corporal para realizar cualquier tipo de aplicación que se nos ocurra. Por ejemplo, podemos realizar un código que analice si una

persona está acostada o de pie. Si tiene los brazos extendidos o no. También podemos analizar cuántos dedos está mostrando (extendidos). Para todo ello, necesitamos saber como obtener la información necesaria a partir de los objetos (puntos) que nos devuelve el modelo.

Tenemos dos funciones que nos ayudan a discriminar puntos que corresponden a ciertas partes del cuerpo como son `FindKeypointID()` y `Keypoints()`. Con estas dos funciones podemos saber las coordenadas, por ejemplo, de muñeca izquierda y hombro izquierdo. A partir de esas dos coordenadas podemos saber hacia dónde apunta el brazo. El siguiente código muestra el código:

```
poses = net.Processing(img)

for pose in poses:
        # find the keypoint index from the list of detected keypoints
        # you can find these keypoint names in the model's JSON file,
        # or with net.GetKeypointName() / net.GetNumKeypoints()
        left_wrist_idx = pose.FindKeypointID('left_wrist')
        left_shoulder_idx = pose.FindKeypointID('left_shoulder')

        # if the keypoint index is < 0, it means it wasn't found in the image
        if left_wrist_idx < 0 or left_shoulder_idx < 0:
        continue

        left_wrist = pose.Keypoints[left_wrist_idx]
        left_shoulder = pose.Keypoints[left_shoulder_idx]

        point_x = left_shoulder.x - left_wrist.x
        point_y = left_shoulder.y - left_wrist.y

        print(f"person {pose.ID} is pointing towards ({point_x}, {point_y})")
```

Los diferentes nombres de puntos posibles son los siguientes.

ID	Nombre	Nombre	Nombre	Nombre
0	palm			
1,2,3,4	thumb_finger_1	thumb_finger_2	thumb_finger_3	thumb_finger_4
5,6,7,8	index_finger_1	index_finger_2	index_finger_3	index_finger_4
9,10,11,12	middle_finger_1	middle_finger_2	middle_finger_3	middle_finger_4
13,14,15,16	ring_finger_1	ring_finger_2	ring_finger_3	ring_finger_4
17,18,19,20	baby_finger_1	baby_finger_2	baby_finger_3	baby_finger_4

ID	Nombre	Nombre	Nombre	Nombre
0	nose			
1,2,3,4	left_eye	right_eye	left_ear	right_ear
5,6,7,8	left_shoulder	right_shoulder	left_elbow	right_elbow
9,10,11,12	left_wrist	right_wrist	left_hip	right_hip
13,14,15,16	left_knee	right_knee	left_ankle	right_ankle
17	neck			

Actividad sugerida: Utilizando el objeto posnet, desarrolla una aplicación, ya sea en Python o en C++, que monitorice la cámara y active una alarma (un mensaje por pantalla) cuando detecte una persona que esté en una posición (por ejemplo, levantando el brazo, sentada, acostada, saltando, …).

Proceso: Detección de acciones

Los modelos de reconocimiento de acciones clasifican una actividad, comportamiento o gesto cuando este ocurre en una secuencia de imágenes (*frames*) de video. Estos modelos utilizan redes neuronales de clasificación con una dimensión temporal añadida. Por ejemplo, modelos pre-entrenados basados en `ResNet18` utilizan una ventana de 16 imágenes.

El objeto (aplicación) `actionNet`, recibe como entrada un video, procesa imagen a imagen a imagen y devuelve la clase con mayor confianza. Tenemos tanto la implementación en Python como la implementación en C++ de la aplicación.

El modelo `actionNet` soporta dos modelos: `resnet18` (por defecto) y `resnet34`. Ambos clasifican 1040 clases. Podemos introducir como entrada un video o la webcam.

Ejemplo de uso:

```
# C++
$ ./actionnet /dev/video0

# Python
$ ./actionnet.py /dev/video0
```

Proceso: Eliminar el fondo (background)

Existen modelos también para eliminar el fondo de una imagen o video. Con esta técnica se genera una máscara que segmenta el objeto (foreground), y por tanto detecta el fondo (background). Podemos utilizarlo para reemplazar o difuminar el fondo (similar a las aplicaciones de videoconferencia), o simplemente como preprocesado para otros modelos de redes neuronales para un mejor seguimiento y detección de objetos.

Tenemos el objeto (aplicación) backgroundNet recibe como entrada una imágen o un stream ya sea un video o la propia webcam, y devuelve la máscara del objeto. Tenemos las dos versiones en Python y en C++.

```
# C++
$ ./backgroundnet images/bird_0.jpg /misdatos/bird_mask.png
$ ./backgroundnet --replace=images/snow.jpg images/bird_0.jpg /misdatos/bird_replace.jpg


# Python
$ ./backgroundnet.py images/bird_0.jpg /misdatos/bird_mask.png
$ ./backgroundnet.py --replace=images/snow.jpg images/bird_0.jpg /misdatos/bird_replace.jpg
```

Prueba con la webcam a ponerte un fondo a partir de cualquier imagen.

Proceso: Predicción de la profundidad con imágenes monoculares

La sensorización de la profundidad de los objetos en una imagen es vital en aplicaciones de ubicación, navegación y detección de obstáculos. Tradicionalmente se utilizan cámaras estéreo o cámaras RGB-D. Sin embargo, existen modelos de redes neuronales que son capaces de inferir la profundidad relativa a partir de una imagen monocular.

Disponemos del objeto (aplicación) `depthNet`, el cual acepta una imagen a color como entrada y obtiene el mapa de profundidad de la imagen. También acepta videos y la webcam. El mapa de profundidad nos aparece coloreado para efectos de visualización. También tenemos disponible el mapa de profundidad numéricamente. Disponemos de la versión en Python y en C++.

Probemos con imágenes de interior:

```
# C++
$ ./depthnet "images/room_*.jpg" /misdatos/depth_room_%i.jpg

# Python
$ ./depthnet.py "images/room_*.jpg" /misdatos/depth_room_%i.jpg
```

Y ahora con imágenes de exterior:

```
# C++
$ ./depthnet "images/trail_*.jpg" /misdatos/depth_trail_%i.jpg

# Python
$ ./depthnet.py "images/trail_*.jpg" /misdatos/depth_trail_%i.jpg
```

Prueba con la cámara.

Si queremos acceder al mapa de profundidad podemos utilizar la función `GetDepthField()` del objeto `depthNet`. Esta función nos devuelve una imagen de un solo canal de tamaño máximo 224x224. Aquí tenemos un ejemplo para calcular la profundidad menor y mayor de la imagen de entrada:

```
import jetson.inference
import jetson.utils

import numpy as np

# load mono depth network
net = jetson.inference.depthNet()

# depthNet re-uses the same memory for the depth field,
# so you only need to do this once (not every frame)
depth_field = net.GetDepthField()

# cudaToNumpy() will map the depth field cudaImage to numpy
# this mapping is persistent, so you only need to do it once
depth_numpy = jetson.utils.cudaToNumpy(depth_field)

print(f"depth field resolution is {depth_field.width}x{depth_field.height},
format={depth_field.format})

while True:
        img = input.Capture()    # assumes you have created an input videoSource stream
        net.Process(img)
```

```
        jetson.utils.cudaDeviceSynchronize() # wait for GPU to finish processing, so we can
use the results on CPU

        # find the min/max values with numpy
        min_depth = np.amin(depth_numpy)
        max_depth = np.amax(depth_numpy)
```

Proceso: Transfer Learning

Transfer learning es una técnica de re-entrenamiento de modelos de redes neuronales con un nuevo conjunto de datos. Este proceso requiere menor tiempo de entrenamiento comparado con un entrenamiento desde cero. Los pesos del modelo se ajustan para clasificar un conjunto de datos nuevo

En este proceso vamos a re-entrenar dos modelos: `ResNet-18` y `SSD-Mobilnet`. Ten en cuenta que el entrenamiento desde cero es más adecuado realizarlo en servidores con GPUs. Ahora bien, el reentrenamiento puede realizarse en las Jetson.

En este proceso vamos a utilizar PyTorch como plataforma de machine learning. Pytorch lo tenemos ya instalado en el contenedor.

Vamos a re-entrenar `ResNet-18` con dos bases de datos, una de perros y gatos, con 5000 imágenes y otra de plantas con 20 clases de plantas, con 10475 imágenes. En ambos casos realizaremos un proceso de clasificación de objetos. También vamos a re-entrenar `SSD-Mobilenet` con un conjunto de datos de frutas (8 clases, 6375 imágenes). En este último caso realizaremos un proceso de detección de objetos.

Vamos primero a obtener el conjunto de datos de perros/gatos. Ten en cuenta que ubicamos los datos en un directorio accesible tanto desde el contenedor como desde fuera del contenedor. Las órdenes siguientes las lanzamos en el contenedor.

```
$ cd jetson-inference/python/training/classification/data
$ wget https://nvidia.box.com/shared/static/o577zd8yp3lmxf5zhm38svrbrv45am3y.gz -O
cat_dog.tar.gz
$ tar xvzf cat_dog.tar.gz
```

Abre algunas imágenes para ver algunos ejemplos. Échale un vistazo también a la estructura de directorios del conjunto de datos.

Ahora, vamos a reentrenar. Debes lanzar el entrenamiento en el contenedor:

```
$ cd jetson-inference/python/training/classification
$ python3 train.py --model-dir=models/cat_dog data/cat_dog
```

Para parar el re-entrenamiento puedes pulsar `Ctrl+C`. Podemos reiniciar el entrenamiento con las opciones `--resume` y `--epoch-start`. Puedes obtener más información con la opción `--help`.

Dispones del modelo ya entrenado con todas las épocas en poliformaT. Debes sustituir el fichero model_best.pth.tar por el fichero disponible en poliformaT (ponle el mismo nombre).

Una vez entrenado (y con una precisión aceptable) vamos a optimizar el modelo para su ejecución en la Jetson. Para ello, debemos convertir el modelo al formato ONNX y de este modelo pasarlo a `TensorRT` para crear un engine.

```
$ python3 onnx_export.py --model-dir=models/cat_dog
```

```
NET=models/cat_dog
DATASET=data/cat_dog

# C++
imagenet --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt $DATASET/test/cat/01.jpg /misdatos/cat.jpg

# Python
imagenet.py --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt $DATASET/test/cat/01.jpg /misdatos/cat.jpg
```

Podemos utilizar la cámara como entrada.

```
# C++
imagenet --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt /dev/video0

# Python
imagenet.py --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt /dev/video0
```

Ahora vamos con el conjunto de datos de plantas. Primero descargamos el conjunto de imágenes:

```
$ cd jetson-inference/python/training/classification/data
$ wget https://nvidia.box.com/shared/static/vbsywpw5iqy7r38j78xs0ctalg7jrg79.gz -O
PlantCLEF_Subset.tar.gz
$ tar xvzf PlantCLEF_Subset.tar.gz
```

Y ahora entrenamos `ResNet-18`:

```
$ cd jetson-inference/python/training/classification
$ python3 train.py --model-dir=models/plants data/PlantCLEF_Subset
```

Dispones del modelo ya entrenado con todas las épocas en poliformaT. Debes sustituir el fichero model_best.pth.tar por el fichero disponible en poliformaT (ponle el mismo nombre).

Solo nos queda convertir el modelo a ONNX y compilarlo con TensorRT:

```
$ python3 onnx_export.py --model-dir=models/plants

NET=models/plants
DATASET=data/PlantCLEF_Subset

# C++
imagenet --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt $DATASET/test/cattail.jpg /misdatos/cattail.jpg

# Python
imagenet.py --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt $DATASET/test/cattail.jpg /misdatos/cattail.jpg
```

Proceso: Generar nuestro propio conjunto de datos para clasificación

Tenemos una aplicación preparada para recolectar y etiquetar un conjunto de datos nuevo. La aplicación nos generará tres directorios: `train`, `val`, y `test`. Dentro de cada directorio se generará un directorio por cada clase que definamos. En cada directorio se almacenarán las imágenes que vayamos generando para una determinada clase. Esta estructura de directorios es la utilizada por la aplicación anterior para entrenar modelos. Por tanto, si generamos un conjunto de datos suficientemente representativo (100 imágenes por clase) podremos generar una nueva aplicación que nos detecte nuevos objetos.

En primer lugar, para generar el conjunto de datos debemos editar un fichero con todas las clases que vamos a utilizar. Este fichero se denominará `labels.txt` y contendrá las clases **en orden alfabético**. Por ejemplo:

```
llaves
mando
telefono
```

Una vez tenemos el fichero de etiquetas vamos a lanzar la aplicación de captura, asociando su entrada a la cámara. Pero antes creamos un directorio para nuestro dataset y editamos el fichero `labels.txt`:

```
$ cd data
$ mkdir prueba
$ cd prueba
$ pico labels.txt
$ cd ..
$ cd ..
```

Ahora ya podemos arrancar la aplicación:

```
$ camera-capture /dev/video0
```

Nos aparecerá un diálogo de captura y la imágen de la cámara. Debemos indicar la ruta del conjunto de datos y el fichero de etiquetas. Una vez indicados se creará toda la estructura de directorios. Ahora, ubicamos en frente de la cámara un objeto y damos a la barra espaciadora para capturar la imagen y etiquetarla con la clase seleccionada y ubicarla en el conjunto de datos seleccionado (`train`, `val`, o `test`).

Se recomienda capturar al menos 100 imágenes por clase antes de entrenar. También es aconsejable que el conjunto de validación (y de test) represente un 10%-20% del conjunto de entrenamiento. También se recomienda realizar capturas de objetos desde diferentes ángulos, condiciones de luz y fondos diferentes para crear un modelo que sea robusto a ruido y cambios del entorno.

Para entrenar el modelo:

```
$ cd jetson-inference/python/training/classification
$ python3 train.py --model-dir=models/<YOUR-MODEL> data/<YOUR-DATASET>
```

Ahora lo exportamos a ONNX y lo compilamos con TensorRT:

```
$ python3 onnx_export.py --model-dir=models/<YOUR-MODEL>

NET=models/<YOUR-MODEL>
DATASET=data/<YOUR-DATASET>

# C++ (MIPI CSI)
imagenet --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt /dev/video0

# Python (MIPI CSI)
imagenet.py --model=$NET/resnet18.onnx --input_blob=input_0 --output_blob=output_0
--labels=$DATASET/labels.txt /dev/video0
```

Proceso: Re-entrenar Modelo de detección

En este punto vamos ahora a re-entrenar un modelo de detección de objetos, en concreto `SSD-Mobilenet`. Este modelo se utiliza en detección de objetos en tiempo real en dispositivos embebidos. Vamos a entrenar un modelo que localice 8 variedades de fruta diferentes.

Primero bajamos el conjunto de datos. Este conjunto consta de 600 clases de objetos. Podemos utilizar un script que nos bajará las clases que le indiquemos. Ahora bien, debemos ver cuanta información nos vamos a bajar. Esto lo conseguimos con la opción `--stats-only`.

$ python3 open_images_downloader.py --stats-only --class-names \
"Apple,Orange,Banana,Strawberry,Grape,Pear,Pineapple,Watermelon" --data=data/fruit

Sin la opción anterior nos bajamos el conjunto de datos:

```
$ cd /jetson-inference/python/training/detection/ssd
$ python3 open_images_downloader.py --class-names
"Apple,Orange,Banana,Strawberry,Grape,Pear,Pineapple,Watermelon" --data=data/fruit
```

También podemos acotar el número máximo de imágenes por clase con la opción –max-images y la opción –max-annotations-per-class nos permite limitar el número de detecciones por clase.

Vamos ya a entrenar el conjunto de datos:

```
python3 train_ssd.py --data=data/fruit --model-dir=models/fruit --batch-size=4
--epochs=30
```

Lo convertimos a ONNX:

```
python3 onnx_export.py --model-dir=models/fruit
```

Y lo procesamos con TensorRT:

```
IMAGES=<path-to-your-jetson-inference>/data/images   # substitute your
jetson-inference path here

detectnet --model=models/fruit/ssd-mobilenet.onnx --labels=models/fruit/labels.txt \
      --input-blob=input_0 --output-cvg=scores --output-bbox=boxes \
            "$IMAGES/fruit_*.jpg" $IMAGES/test/fruit_%i.jpg
```

¡Prueba con la cámara!.

Proceso: Generar nuestro propio conjunto de datos para detección

Con la aplicación `camera-capture` podemos generar un conjunto de datos de detección de objetos a partir de la cámara. Al seleccionar *detección* la aplicación generará un conjunto de datos en el formato `Pascal VOC` (soportado por la aplicación de entrenamiento).

Primero creamos un directorio:

$ cd /jetson-inference/python/training/detection/ssd/data
$ mkdir prueba
$ cd prueba
$ pico labels.txt
$ cd ..
$ cd ..

Lanzamos ahora la aplicación de captura:

$ camera-capture /dev/video0

En Dataset Type seleccionamos “detection”. Después seleccionamos la ruta al conjunto de datos y al fichero de clases.

Para capturar los objetos de una imagen debemos primero poner los objetos en frente de la cámara y pulsar el botón Freeze/Edit (o pulsamos la barra espaciadora). En ese momento se congelará la imagen y podremos seleccionar los objetos dibujando rectángulos (bounding boxes) sobre los objetos. Para cada bounding box seleccionamos la clase correspondiente. Cuando hemos terminado con la imagen volvemos a pulsar la barra espaciadora (o el botón Freeze/Edit).

Recuerda que debes obtener imágenes con diferentes orientaciones, puntos de cámara, condiciones de luz y diferentes fondos para crear un modelo robusto al ruido y al entorno. Si tu modelo final no funciona correctamente deberás añadir mas datos para el entrenamiento.

Solo nos queda ya entrenar el modelo, convertirlo a ONNX y finalmente optimizarlo con TensorRT.

```
$ cd jetson-inference/python/training/detection/ssd
$ python3 train_ssd.py --dataset-type=voc --data=data/prueba \
 --model-dir=models/prueba

$ python3 onnx_export.py --model-dir=models/prueba

$ NET=models/prueba

$ detectnet --model=$NET/ssd-mobilenet.onnx --labels=$NET/labels.txt \
          --input-blob=input_0 --output-cvg=scores --output-bbox=boxes \
            /dev/video0
```