Embedded AI Labs with Jetson Nano

The following **EAI Labs** are based on Jetson Nano board and a number of prepared packages and libraries.

The Hardware:
NVIDIA® Jetson Nano™ Developer Kit is a small, powerful computer that lets you run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts.
It's simpler than ever to get started! Just insert a microSD card with the system image, boot the developer kit, and begin using the same NVIDIA JetPack SDK used across the entire NVIDIA Jetson™ family of products. **JetPack** is compatible with NVIDIA's world-leading AI platform for training and deploying AI software, reducing complexity and effort for developers.

The Software:
Our EAI Labs concern Inference, Facial Recognition and Audio. The OS-JetPack and the required packages are prepared on a **64 GB SD** card.
You just start power the board and wait for the **login** invite.
use user : **`jetson`** and password: **`jetson`** .

The following is the prontout of the main directory:

```
jetson@jetson-desktop:~$ ls
Desktop             examples.desktop  jetson-inference  Public
dlib-19.17          face_recognition  jetson-voice      sound_class
dlib-19.17.tar.bz2  Images            Modèles           Téléchargements
Documents           installSwapfile   Musique           Vidéos
jetson@jetson-desktop:~$
```

Lab1: To activate **`jetson-inference docker`**:

```
jetson@jetson-desktop:cd  jetson-inference
jetson@jetson-desktop:~/jetson-inference$ docker/run.sh
ARCH:  aarch64
reading L4T version from /etc/nv_tegra_release
L4T BSP Version:  L4T R32.7.1
[sudo] password for jetson:
size of data/networks:  623468514 bytes
CONTAINER:     dustynv/jetson-inference:r32.7.1
DATA_VOLUME:   --volume /home/jetson/jetson-inference/data:/jetson-inference/data --volume
/home/jetson/jetson-inference/python/training/classification/data:/jetson-inference/python/
training/classification/data --volume
/home/jetson/jetson-inference/python/training/classification/models:/jetson-inference/python/
training/classification/models --volume
/home/jetson/jetson-inference/python/training/detection/ssd/data:/jetson-inference/python/training/
detection/ssd/data --volume /home/jetson/jetson-inference/python/training/detection/ssd/models:/
jetson-inference/python/training/detection/ssd/models
USER_VOLUME:
USER_COMMAND:
V4L2_DEVICES:    --device /dev/video0
localuser:root being added to access control list
xhost:  must be on local machine to add or remove hosts.
root@jetson-desktop:/jetson-inference#
```

Lab2: To use **`face_recognition`** examples:

```
jetson@jetson-desktop:~$ cd face_recognition/examples/
jetson@jetson-desktop:~/face_recognition/examples$ ls
alex-lacamoire.png                          find_facial_features_in_picture.py
bako.jpg                                    hamilton_clip.mp4
benchmark.py                                identify_and_draw_boxes_on_faces.py
biden.jpg                                   ipynb_examples
blink_detection.py                          knn_examples
blur_faces_on_webcam.py                     lin-manuel-miranda.png
digital_makeup.py                           obama-1080p.jpg
face_distance.py                            obama-240p.jpg
facerec_from_video_file.py                  obama2.jpg
facerec_from_webcam_faster.py               obama-480p.jpg
facerec_from_webcam_multiprocessing.py      obama-720p.jpg
facerec_from_webcam.py                      obama.jpg
```

```
facerec_ipcamera_knn.py                          obama_small.jpg
face_recognition_knn.py                          project1.py
face_recognition_svm.py                          recognize_faces_in_pictures.py
facerec_on_raspberry_pi.py                       short_hamilton_clip.mp4
facerec_on_raspberry_pi_Simplified_Chinese.py    two_people.jpg
find_faces_in_batches.py                         web_service_example.py
find_faces_in_picture_cnn.py                     web_service_example_Simplified_Chinese.py
find_faces_in_picture.py
```

Lab3: To activate **`jetson-voice docker`**:

```
jetson@jetson-desktop:~$ cd jetson-voice/
jetson@jetson-desktop:~/jetson-voice$ docker/run.sh
ARCH:  aarch64
reading L4T version from /etc/nv_tegra_release
L4T BSP Version:  L4T R32.7.1
CONTAINER:     dustynv/jetson-voice:r32.7.1
DEV_VOLUME:
DATA_VOLUME:   --volume /home/jetson/jetson-voice/data:/jetson-voice/data
USER_VOLUME:
USER_COMMAND:
root@jetson-desktop:/jetson-voice#
```

Table of Contents

Lab 1

Jetson Nano – Inference

1.0 Introduction

In this lab we will analyze and test a number of applications with **NN** models prepared for inference.
All applications are developed with the **Python** language.
We start with **image classification** with **`ImageNet`** and **object localization** with **`DetectNet`**.

1.0.1 Image Classification with `ImageNet`

There are several types of deep learning networks available, including recognition, detection/localization, and semantic segmentation. The first deep learning capability we highlight in this lab is **image recognition** using **classification networks** that have been trained to identify scenes and objects.
The **`ImageNet`** object accepts an input image and generates the probability for each class. After being trained on the **`ImageNet`** ILSVRC dataset of 1000 objects, the **`GoogleNet`** and **`ResNet-18`** models were automatically downloaded during the build step.

Other classification models can also be downloaded and used.

1.0.2 Object Localization with `DetectNet`

The preceding image recognition examples have **output class probabilities** representing the entire input image. The second deep learning capability we highlight in this lab is **detecting objects** and finding their **location** in the video (i.e. extracting their **bounding boxes**). This is done using a **`detectNet`** - or object detection/location network.

The **`detectNet`** object accepts the 2D image as input and generates a **list of coordinates** of the detected bounding boxes. For object detection, a pre-trained model (like **`ssd-mobilenet-v2`**) is used by default with boundary coordinate labels included in the training dataset in addition to source images.

1.0.3 Pose Estimation with `PoseNet`

Pose estimation involves **locating various body parts** (aka **keypoints**) that form a skeletal topology (aka **links**). Pose estimation has a variety of applications, including **gestures**, AR/VR, HMI (human/machine interface), and posture/gait correction. Pre-trained models are provided for pose estimation of human **body** and **hand** which are able to detect multiple people per image. By default the application uses the **`resnet18-body`** model.

1.0.4 Monocular Depth Sensing with `DepthNet`

Depth sensing is useful for tasks such as **mapping**, **navigation**, and **obstacle detection**, but historically required a **stereo camera** or an RGB-D camera.
There are now **DNNs** that can infer relative depth from a single monocular image (aka **mono depth**). Such an approach can be accomplished using **fully convolutional networks** (**FCNs**). The pre-trained (default) model used in this application is **`fcn-mobilenet`**.

1.0.5 Semantic Segmentation with `SegNet`

The third deep learning capability that we can highlight is **semantic segmentation**. Semantic segmentation is based on **image recognition**, except that **classifications occur at the pixel level** as opposed to whole-image classification as with image recognition.
This is accomplished by "convolutionalizing" a pre-trained image recognition model (like **`Alexnet`**), which transforms it into a fully convolutional segmentation model capable of **per-pixel labeling**. By default the application uses the **`fcn-resnet18-voc`** template.

Useful for environment sensing and **collision avoidance**, segmentation produces pixel-dense classification of many different potential objects per scene, including scene foregrounds and backgrounds.

1.1 Image Classification with `ImageNet`

There are several types of deep learning networks available, including recognition, detection/localization, and semantic segmentation. The first deep learning capability we highlight in this lab is **image recognition** using **classification networks** that have been trained to identify scenes and objects.

The **`ImageNet`** object accepts an input image and generates the **probability for each class**. After being trained on the **`ImageNet`** ILSVRC dataset of **1000 objects**, the **`GoogleNet`** and **`ResNet-18`** models were automatically downloaded during the build step.
Other classification models can also be downloaded and used.

For our code examples with **`imageNet`**, we use Python:

- **`imagenet-console.py`**

Later in this section we will also discuss versions of a recognizer from a **camera** or **WebCam** in **live mode**:

- **`imagenet-camera.py`**

1.1.1 Using the `console` program on Jetson

First, let's try using the **`imagenet-console.py`** program to test **`imageNet`** recognition on some sample images. It loads an image, uses **TensorRT** and the **`imageNet`** class to perform the inference, then overlays the classification result and saves the output image.

The repository comes with some sample images you can use.

First we initialize a **docker session** by:

```
$ cd jetson-inference
docker/run.sh
```

```
reading L4T version from /etc/nv_tegra_release
L4T BSP Version:  L4T R32.6.1
[sudo] password for jetson:
size of data/networks:  1269108476 bytes
CONTAINER:     dustynv/jetson-inference:r32.6.1
DATA_VOLUME:   --volume /home/jetson/jetson-inference/data:/jetson-inference/data --volume
/home/jetson/jetson-inference/python/training/classification/data:/jetson-inference/python/
training/classification/data --volume
/home/jetson/jetson-inference/python/training/classification/models:/jetson-inference/python/
training/classification/models --volume
/home/jetson/jetson-inference/python/training/detection/ssd/data:/jetson-inference/python/training/
detection/ssd/data --volume /home/jetson/jetson-inference/python/training/detection/ssd/models:/
jetson-inference/python/training/detection/ssd/models
USER_VOLUME:
USER_COMMAND:
V4L2_DEVICES:
localuser:root being added to access control list
xhost:  must be on local machine to add or remove hosts.
root@jetson-desktop:/jetson-inference#
```

After building the repository, make sure your terminal is in the **directory**:

```
build/aarch64/bin :
```

```
root@jetson-desktop:/jetson-inference#cd build/aarch64/bin
```

Vous vérifiez la présence des programmes (applications) dan le répertoire **`bin :`**

You check for programs (applications) in the **bin** directory:

```
root@jetson-desktop:/jetson-inference/build/aarch64/bin# ls
```

```
camera-capture      detectnet             imagenet-console     segnet-camera
camera-viewer       detectnet-camera      imagenet-console.py  segnet-camera.py
```

```
camera-viewer.py     detectnet-camera.py    imagenet.py          segnet-console
cuda-examples.py     detectnet-console      images               segnet-console.py
cuda-from-cv.py      detectnet-console.py   my-detection.py      segnet.py
cuda-from-numpy.py   detectnet.py           my-recognition.py    segnet_utils.py
cuda-to-cv.py        gl-display-test        networks             video-viewer
cuda-to-numpy.py     gl-display-test.py     posenet              video-viewer.py
depthnet             imagenet               posenet.py
depthnet.py          imagenet-camera        segnet
depthnet_utils.py    imagenet-camera.py     segnet-batch.sh
```

Next, let's classify an image with the **`imagenet-console`** program:

`imagenet-console` accepts 3 command line arguments:

- the path (**`path`**) to an input image (jpg, png, tga, bmp)
- optional path to output image (jpg, png, tga, bmp)
- optional **`flag`** --network which modifies the classification model used (the default network is GoogleNet).

Note that there are additional command line parameters available for loading custom templates. Launch the application with the **`--help`** flag to receive more information on their use.
Here are some examples of running the program in **Python**:

```
$ ./imagenet-console.py --network=googlenet images/cat_0.jpg images/out_0.jpg
```

Remarks:
The **`--network`** flag is optional. The first time you run the program, **TensorRT** may take up to **a few minutes** to optimize the network.
This network-optimized file **is cached to disk** after the first run, so future runs will load faster.

1.1.2 Download other classification models

By default, the repository is configured to download **`GoogleNet`** and **`ResNet-18`** networks during the build step.
There are other pre-trained models that you can also use, if you choose to download them:

Network	CLI argument	NetworkType enum
AlexNet	alexnet	ALEXNET
GoogleNet	googlenet	GOOGLENET
GoogleNet-12	googlenet-12	GOOGLENET_12
ResNet-18	resnet-18	RESNET_18
ResNet-50	resnet-50	RESNET_50
ResNet-101	resnet-101	RESNET_101
ResNet-152	resnet-152	RESNET_152
VGG-16	vgg-16	VGG-16
VGG-19	vgg-19	VGG-19
Inception-v4	inception-v4	INCEPTION_V4

Tab 1. Models for the classification of objects.

Remarque : pour télécharger des réseaux supplémentaires, exécutez l'outil **Model Downloader** :

Note: To download additional networks, run the **Model Downloader** tool:

```
$ cd jetson-inference/tools
$ ./download-models.sh
```

In general, more complex networks can have **higher classification accuracy**, with increased run time.

Caution: Loading some very large templates may cause the system to crash.

1.1.3 Use of different classification models

You can specify which model to load by setting the **`--network`** flag on the command line to one of the corresponding CLI arguments from the table above.
By default, **`GoogleNet`** is loaded if the optional **`--network`** flag is not specified.

Here is an example of using the **`ResNet-18`** model:

```
$ ./imagenet-console.py --network=resnet-18 jellyfish.jpg output_jellyfish.jpg
```

1.1.4 Classification with a live camera/WebCam

We have a **real-time image recognition** with camera demo available for Python:

`imagenet-camera.py`

Similar to the previous **`console-imagenet`** example, camera applications are created in the **`/aarch64/bin`** directory. They operate on a live camera feed with **OpenGL** rendering and accept 4 optional command line arguments:

- --**`network`** flag defining the classification model (the default value is **`GoogleNet`**)
  See Downloading other classification models for available networks to use.

- --**`camera`** flag defining the camera to use
  - MIPI CSI cameras are used by specifying the sensor index (0 or 1, ect.)
  - V4L2 USB cameras are used by specifying their /dev/video node (**`/dev/video0`**, **`/dev/video1`**, ect.)
  - Default is MIPI CSI sensor 0 (--camera=0)

- --**`width`** and --**`height`** flags defining the resolution of the camera (default is **1280x720**)
  - The resolution must be set to a format supported by the camera.

You can combine the use of these indicators according to your needs. Additional command line parameters are available for loading custom templates.

Launch the application with the **`--help`** flag to receive more information.

Here are some typical program launch scenarios:

```
$ ./imagenet-camera.py      # using GoogleNet, default MIPI CSI camera (1280x720)
$ ./imagenet-camera.py --network=resnet-18  # using ResNet-18, default MIPI CSI camera (1280x720)
$ ./imagenet-camera.py --camera=/dev/video0 # using GoogleNet, V4L2 camera /dev/video0 (1280x720)
$ ./imagenet-camera.py --width=640 --height=480 # using GoogleNet, default MIPI CSI camera (640x480)
```

For the **live camera feed**, the classified object name and classified object trust, as well as the network frame rate, are displayed in the **OpenGL** window. On Jetson Nano, you should see up to around 75 FPS for **`GoogleNet`** and **`ResNet-18`**.

The application can recognize up to 1000 different object types because the classification models are trained on the ILSVRC **`ImageNet`** dataset which contains 1000 object classes.

The **name mapping** for the 1000 object types you can find in the repo under

`data/networks/ilsvrc12_synset_words.txt`

1.2 Object detection/location from command line/console

To process the test images with **detectNet** and **TensorRT** on the Jetson, we can use the **detectnet-console** program.

detectnet-console accepts command line arguments representing the input image path and the output image path (with rendered **bounding box overlays**).

1.2.1 Execution with a pre-trained model

Alternatively, to load one of the pre-trained snapshots provided with the repository, you can specify the optional --**network** flag which changes the detection model used (the default network is **SSD-Mobilenet-v2**).

Here is an example of **locating humans** in an image with the default **SSD-Mobilenet-v2** model:

```
$ python3 ./detectnet-console.py images/peds_1.jpg images/out_1.jpg
```

You can view available and generated images in:

```
~/jetson-inference/data/images
```

1.2.2 Detection/localization with pre-trained models available

Below is a table of pre-trained object detection networks available for download, and the --**network** argument to **detectnet-console** used to load the pre-trained models:

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)
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

Tab 2. Models for object detection.

1.2.3 Running different detection/localization models

You can specify which model to load by setting the --**network** flag on the command line to one of the corresponding **CLI** arguments from the table above.
By default, **PedNet** is loaded (**pedestrian detection**) if the --**network** option is **not specified**.

Let's try running some of the other COCO models:

```
$ ./detectnet-console.py --network=coco-dog dog_1.jpg output_1.jpg
```

1.2.4 Running the Live Detection/Localization Demo

We have a real-time object detection camera/webcam demo available for Python:

detectnet-camera.py

Similar to the previous **detectnet-console** example, these camera apps use detection networks, except they process a live video stream from a camera. **detectnet-camera** accepts **3 optional command line parameters**:

- --**network** flag defining the classification model (the default value is **PedNet**)
  - See Available pre-trained detection models for available networks to use.
- --**camera** flag defining the camera to use
  - **MIPI CSI** cameras are used by specifying the sensor index (0 or 1, ..)
  - **V4L2 USB** cameras are used by specifying their **/dev/video** node (**/dev/video0**, **/dev/video1**, ..)
  - Default is to use MIPI CSI 0 sensor (--**camera=0**)
- Flags --**width** and --**height** defining the resolution of the camera (the default value is **1280x720**)
  - The resolution must be set to a format supported by the camera.

You can combine the use of these flags as needed, and additional command line parameters are available for loading custom models. Run the application with the --**help** flag to receive more information, or see the sample **readme** file.

Here are some typical program launch scenarios:

```
$ ./detectnet-camera.py                          # using PedNet,  default MIPI CSI camera (1280x720)
$ ./detectnet-camera.py --network=facenet        # using FaceNet, default MIPI CSI camera (1280x720)
$ ./detectnet-camera.py --camera=/dev/video0     # using PedNet,  V4L2 camera /dev/video0 (1280x720)
$ ./detectnet-camera.py --width=640 --height=480 # using PedNet,  default MIPI CSI camera (640x480)
```

4.2.4.1 Visualization

In the **OpenGL** window the live camera feed is displayed superimposed with the bounding boxes of the detected objects. Note that **SSD** (Single-Shot Detector) **based models** currently have the highest performance. Here is one using the **coco-dog** pattern:

```
$ ./detectnet-camera.py -network=coco-dog
```

1.3 Pose estimation with `posenet.py` and `resnet18-body`

Pose estimation involves locating various body parts (aka **keypoints**) that form a skeletal topology (aka **links**). By default the application uses the **resnet18-body** model.

Here is the complete **posenet.py** code.

```
import jetson.inference
import jetson.utils
import argparse
import sys
# parse the command line
parser = argparse.ArgumentParser(description="Run pose estimation DNN on a video/image stream.",
                                 formatter_class=argparse.RawTextHelpFormatter,
epilog=jetson.inference.poseNet.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="resnet18-body", help="pre-trained model to load
(see below for options)")
parser.add_argument("--overlay", type=str, default="links,keypoints", help="pose overlay flags (e.g.
--overlay=links,keypoints)\nvalid combinations are:  'links', 'keypoints', 'boxes', 'none'")
parser.add_argument("--threshold", type=float, default=0.15, help="minimum detection threshold to
use")
try:
        opt = parser.parse_known_args()[0]
except:
        print("")
        parser.print_help()
        sys.exit(0)

# load the pose estimation model
net = jetson.inference.poseNet(opt.network, sys.argv, opt.threshold)
# create video sources & outputs
input = jetson.utils.videoSource(opt.input_URI, argv=sys.argv)
output = jetson.utils.videoOutput(opt.output_URI, argv=sys.argv)
```

```
# process frames until the user exits
while True:
    # capture the next image
    img = input.Capture()
    # perform pose estimation (with overlay)
    poses = net.Process(img, overlay=opt.overlay)
    # print the pose results
    print("detected {:d} objects in image".format(len(poses)))
    for pose in poses:
        print(pose)
        print(pose.Keypoints)
        print('Links', pose.Links)
    # render the image
    output.Render(img)
    # update the title bar
    output.SetStatus("{:s} | Network {:.0f} FPS".format(opt.network, net.GetNetworkFPS()))
    # print out performance info
    net.PrintProfilerTimes()
    # exit on input/output EOS
    if not input.IsStreaming() or not output.IsStreaming():
        break
```

To observe your position launch the program with:

```
./posenet.py -camera=/dev/video0
```

Note that the parameters - **positions** of your body are displayed on the terminal with **x,y** coordinates.

1.4 Monocular Depth with `DepthNet`

Depth sensing is useful for tasks such as **mapping**, **navigation**, and **obstacle detection**, but historically required a stereo camera or an RGB-D camera.

There are now **DNNs** that can infer **relative depth from a single monocular image** (aka **mono depth**). Such an approach can be accomplished using **fully convolutional** networks (**FCNs**).

The pre-trained (default) model used in this application is **`fcn-mobilenet`**.

```
import jetson.inference
import jetson.utils
import argparse
import sys
from depthnet_utils import depthBuffers
# parse the command line
parser = argparse.ArgumentParser(description="Mono depth estimation on a video/image stream using
depthNet DNN.",
                                 formatter_class=argparse.RawTextHelpFormatter,
epilog=jetson.inference.depthNet.Usage() +
                                 jetson.utils.videoSource.Usage() + jetson.utils.videoOutput.Usage()
+ jetson.utils.logUsage())

parser.add_argument("input_URI", type=str, default="", nargs=&apos;?&apos;, help="URI of the input
stream")
parser.add_argument("output_URI", type=str, default="", nargs=&apos;?&apos;, help="URI of the output
stream")
parser.add_argument("--network", type=str, default="fcn-mobilenet", help="pre-trained model to load,
see below for options")
parser.add_argument("--visualize", type=str, default="input,depth", help="visualization options (can
be &apos;input&apos; &apos;depth&apos; &apos;input,depth&apos;")
parser.add_argument("--depth-size", type=float, default=1.0, help="scales the size of the depth map
visualization, as a percentage of th$
parser.add_argument("--filter-mode", type=str, default="linear", choices=["point", "linear"],
help="filtering mode used during visualiza$
parser.add_argument("--colormap", type=str, default="viridis-inverted", help="colormap to use for
visualization (default is &apos;viridis-inv$
                                 choices=["inferno", "inferno-inverted", "magma", "magma-inverted",
"parula", "parula-inverted",
                                          "plasma", "plasma-inverted", "turbo", "turbo-inverted",
"viridis", "viridis-inverted"])

try:
        opt = parser.parse_known_args()[0]
except:
        print("")
        parser.print_help()
        sys.exit(0)

# load the segmentation network
net = jetson.inference.depthNet(opt.network, sys.argv)
# create buffer manager
buffers = depthBuffers(opt)
# create video sources & outputs
input = jetson.utils.videoSource(opt.input_URI, argv=sys.argv)
output = jetson.utils.videoOutput(opt.output_URI, argv=sys.argv)
# process frames until user exits
while True:
    # capture the next image
    img_input = input.Capture()
    # allocate buffers for this size image
    buffers.Alloc(img_input.shape, img_input.format)
    # process the mono depth and visualize
    net.Process(img_input, buffers.depth, opt.colormap, opt.filter_mode)
    # composite the images
    if buffers.use_input:
        jetson.utils.cudaOverlay(img_input, buffers.composite, 0, 0)
       if buffers.use_depth:
        jetson.utils.cudaOverlay(buffers.depth, buffers.composite, img_input.width if
buffers.use_input else 0, 0)

    # render the output image
    output.Render(buffers.composite)
```

```
    # update the title bar
    output.SetStatus("{:s} | {:s} | Network {:.0f} FPS".format(opt.network, net.GetNetworkName(),
net.GetNetworkFPS()))
    # print out performance info
    jetson.utils.cudaDeviceSynchronize()
    net.PrintProfilerTimes()
    # exit on input/output EOS
    if not input.IsStreaming() or not output.IsStreaming():
        break
```

You can launch this program as follows:

```
./depthnet.py  --camera=/dev/video0  --height=480  --width=640
```

1.5 Semantic segmentation with `SegNet`

The third deep learning capability that we can highlight is **semantic segmentation**.

Semantic segmentation is based on **image recognition**, except that classifications occur **at the pixel level** as opposed to whole-image classification as with image recognition.

This is accomplished by "convolutionalizing" a pre-trained image recognition model (like **`Alexnet`**), which transforms it into a **fully convolutional segmentation** model capable of per-pixel labeling. By default the application uses the fcn-resnet18-voc template.

Useful for **environment sensing** and **collision avoidance**, segmentation produces pixel-dense classification of many different potential objects per scene, including scene foregrounds and backgrounds.

```
import jetson.inference
import jetson.utils
import argparse
import sys
from segnet_utils import *
# parse the command line
parser = argparse.ArgumentParser(description="Segment a live camera stream using an semantic
segmentation DNN.",
                                 formatter_class=argparse.RawTextHelpFormatter,
epilog=jetson.inference.segNet.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="fcn-resnet18-voc", help="pre-trained model to
load, see below for options")
parser.add_argument("--filter-mode", type=str, default="linear", choices=["point", "linear"],
help="filtering mode used during visualiza$
parser.add_argument("--visualize", type=str, default="overlay,mask", help="Visualization options
(can be 'overlay' 'mask' 'overlay,mask'$
parser.add_argument("--ignore-class", type=str, default="void", help="optional name of class to
ignore in the visualization results (def$
parser.add_argument("--alpha", type=float, default=150.0, help="alpha blending value to use during
overlay, between 0.0 and 255.0 (defau$
parser.add_argument("--stats", action="store_true", help="compute statistics about segmentation mask
class output")

is_headless = ["--headless"] if sys.argv[0].find('console.py') != -1 else [""]

try:
        opt = parser.parse_known_args()[0]
except:
        print("")
        parser.print_help()
        sys.exit(0)

# load the segmentation network
net = jetson.inference.segNet(opt.network, sys.argv)
# set the alpha blending value
net.SetOverlayAlpha(opt.alpha)
# create buffer manager
buffers = segmentationBuffers(net, opt)
# create video sources & outputs
input = jetson.utils.videoSource(opt.input_URI, argv=sys.argv)
output = jetson.utils.videoOutput(opt.output_URI, argv=sys.argv+is_headless)
# process frames until user exits
while True:
        # capture the next image
        img_input = input.Capture()
        # allocate buffers for this size image
        buffers.Alloc(img_input.shape, img_input.format)
        # process the segmentation network
        net.Process(img_input, ignore_class=opt.ignore_class)

        # generate the overlay
        if buffers.overlay:
                net.Overlay(buffers.overlay, filter_mode=opt.filter_mode)
        # generate the mask
```

```
        if buffers.mask:
                net.Mask(buffers.mask, filter_mode=opt.filter_mode)
        # composite the images
        if buffers.composite:
                jetson.utils.cudaOverlay(buffers.overlay, buffers.composite, 0, 0)
                jetson.utils.cudaOverlay(buffers.mask, buffers.composite, buffers.overlay.width, 0)
        # render the output image
        output.Render(buffers.output)
        # update the title bar
        output.SetStatus("{:s} | Network {:.0f} FPS".format(opt.network, net.GetNetworkFPS()))
       # print out performance info
        jetson.utils.cudaDeviceSynchronize()
        net.PrintProfilerTimes()
    # compute segmentation class stats
        if opt.stats:
                buffers.ComputeStats()
        # exit on input/output EOS
        if not input.IsStreaming() or not output.IsStreaming():
                break
```

You can launch the execution of this program statically – a photo (**segnet-console.py**), or dynamically with the capture of the video by a camera – WebCam (**segnet-camera.py**).

For instance :

```
./segnet-camera.py -camera=/dev/video0  --width=640  --height=480
```

1.6 Mini project 1 - `my-recognition.py`

Coding your own image recognition program (Python).
In the previous step, we ran an application provided with the **`jetson-inference`** repository.
Now let's walk through creating a new program from scratch in Python for the image recognition called **`my-recognition.py`**.

This script will load an arbitrary image from disk and classify it using the **`imageNet`** object.

For your convenience and reference, the full source is available in the python file **`/examples/my-recognition.py`** from the repository, but the guide below will act as if it resides in the user's home directory or an arbitrary directory of your choice.

1.6.1 Implementation of the project

You can store the **`my-recognition.py`** example that we are going to create wherever you want on your Jetson.
For simplicity, this guide will create it with some test images in a directory under the directory personal user located at **`~/my-recognition-python`**.

Run these commands from a terminal to create the required directory and files:

```
$ cd ~/$ mkdir my-recognition-python$ cd my-recognition-python
$ touch my-recognition.py
$ chmod +x my-recognition.py
$ 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
```

Some test images are also downloaded to the folder with the **`wget`** commands above. Next, we are going to add the Python code to the empty source file we've created here.

1.6.2 Source code

Open **`my-recognition.py`** in your editor of choice (e.g. **`gedit my-recognition.py`**).

1.6.2.1 Importing modules

Add **`import`** statements to import the **`jetson.inference`** and **`jetson.utils`** modules used for recognizing images and loading images.
We will also load the standard **`argparse`** package to parse the command line.

```
import jetson.inference
import jetson.utils
import argparse
```

1.6.2.2 Command line analysis (`parser`)

Next, add code to parse the image filename and the --**`network`** parameter:

```
# 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, ect. (see --help for others)")
opt = parser.parse_args()
```

This example loads and classifies a user-specified image. It should be executed like this:

```
$ ./my-recognition.py my_image.jpg
```

The image file name to load should be changed to **`my_image.jpg`**. You can also specify the --**`network`** parameter to change the classification network used (the default is **`GoogleNet`**):

```
$ ./my-recognition.py --network = resnet-18 my_image.jpg
```

1.6.2.3 Loading image from disk

You can load images into GPU memory using the **`loadImageRGBA()`** function. The Supported formats are: **JPG**, **PNG**, **TGA** and **BMP**.

Add this line to load the image with the specified filename from the command line:

```
img, width, height = jetson.utils.loadImageRGBA (opt.filename)
```

The loaded image will be stored in **shared memory** which is **mapped to both CPU and GPU**. Since the CPU and Jetson's integrated GPU share the same physical memory, memory copies (**`cudaMemcpy()`**) between devices are not necessary.

Note that the image is loaded in **`float4`** **RGBA** format, with pixel values between **0.0 and 255.0**.

1.6.2.4 Loading image recognition network/model

Using the **`imageNet`** object, the following code will load the desired **classification model** with **TensorRT**. Unless you specified a different network using the --**`network`** flag, by default it will load **`GoogleNet`**, which was already downloaded when you originally created the **`jetson-inference`** directory (the **`ResNet-18`** model was also uploaded).

All available classification models are pre-trained on the **`ImageNet`** dataset ILSVRC, which can recognize up to **1000 different classes of objects**, such as different types of fruits and vegetables, many different animal species, as well as everyday objects like vehicles, office furniture, sports equipment, etc.

```
# load the reconnaissance network
net = jetson.inference.imageNet(opt.network)
```

1.6.2.5 Image Classification

Next, we will classify the image with the recognition network using the function:

`imageNet.Classify()`:

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

`imageNet.Classify()` accepts the image and its dimensions and performs inference with **TensorRT**. It returns a tuple containing the entire **index of the object class** under which the image was recognized, and the **confidence value** (in floating point) of the result.

1.6.2.6 Results interpretation

In the last step, we retrieve the class description and print the results of the classification:

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

`ImageNet.Classify()` returns the **index** of the recognized class of objects (between 0 and 999 for those models which were trained on ILSVRC).
Given the class index, the **`imageNet.GetClassDesc()`** function returns the string containing the textual description of this class. These descriptions are automatically loaded from **`ilsvrc12_synset_words.txt`** file.

And there you go ! This is all of the Python code we need for image classification.

1.6.2.7 Source code

For completeness, here is the **full source of the Python script** we just created.

```
import jetson.inference
import jetson.utils
import argparse
# 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, ect. (see --help for others)")
opt = parser.parse_args()
# load an image (into shared CPU/GPU memory)
img, width, height = jetson.utils.loadImageRGBA(opt.filename)
# load the recognition network
net = jetson.inference.imageNet(opt.network)
# classify the image
class_idx, confidence = net.Classify(img, width, height)
# 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))
```

The image file name to load should be changed to **`my_image.jpg`**. You can also (optionally) specify the --network parameter to change the classification network used (the default is **`GoogleNet`**):

```
$ python3 ./my-recognition.py --network=resnet-18 images/cat_0.jpg
..
image is recognized as 'tabby, tabby cat' (class #281) with 8.544922% confidencejetson.utils --
freeing CUDA mapped memoryPyTensorNet_Dealloc()

$ python3 ./my-recognition.py --network=GoogleNet images/cat_0.jpg
..
image is recognized as 'tiger cat' (class #282) with 17.700195% confidencejetson.utils -- freeing
CUDA mapped memoryPyTensorNet_Dealloc()
```

Figure 1 Image **`cat_0.jpg`**

1.7 Mini project 2 - `my-detection.py`

In this part of the lab, we will go through the creation of an application for **real-time object detection** on a camera (or Webcam) stream in just 10 lines of Python code.
The program will load the detection network with the **`detectNet`** object, capture video images and process them, then display the names (labels) of the detected objects on the screen.

1.7.1 Source code

First, open your text editor of choice and create a new file. We'll assume that you save it in your user's home directory as: **`my-detection.py`**.

1.7.1.1 Importing modules

At the top of the source file, we'll import the Python modules that we're going to use in the script.
Add **`import`** statements to load the **`jetson.inference`** and **`jetson.utils`** modules used for object detection and camera capture.

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

1.7.1.2 Loading the detection model

Then use the following line to create a **`detectNet`** object instance that loads the **`SSD-model-Mobilenet-v2`** with **91 classes**:

```
# load object detection model
net = jetson.inference.detectNet("ssd-mobilenet-v2", threshold=0.5)
```

Note that you can override the pattern string to load a different detection pattern. We also set the **default detection threshold** to **0.5** for illustration purposes - you can change it later if necessary.

1.7.1.3 Open Camera Stream

To connect to the video capture device for streaming, we will create an instance of the object **`gstCamera`**:

```
camera = jetson.utils.gstCamera(1280,720,"/dev/video0") # en utilisant V4L2
```

Its constructor accepts 3 parameters - the **`width`**, the **`height`** and the **`video device`** to use.
Replace the following snippet depending on whether you are using a **MIPI CSI** camera or a **V4L2 USB** camera, along with the preferred resolution:

MIPI CSI cameras are used by specifying the sensor index ("0" or "1", etc.)

```
camera = jetson.utils.gstCamera(1280,720,"0")
```

V4L2 USB Webcams are used by specifying their node: ("**`/dev/video0`**","**`/dev/video1`**",etc.)

```
camera = jetson.utils.gstCamera(1280,720,"/dev/video0")
```

If needed, modify **1280** and **720** above to desired **width**/**height**

1.7.1.4 Display Loop

Next, we'll create an **OpenGL** display with the **`glDisplay`** object and create a main loop that will run until the user exits:

```
display = jetson.utils.glDisplay()
while display.IsOpen():
# main loop will go here
```

Note that the rest of the code below should be indented below this **`while`** loop.

1.7.1.5 Capture de la vidéo

The first thing that happens in the main loop is capturing the next video-frame. **`camera.CaptureRGBA()`** - will wait until the next frame has been sent from the camera, and once acquired by the Jetson, it will convert it to the **RGBA floating-point** format residing in the GPU's memory.

```
img,width,height = camera.CaptureRGBA()
```

The returned result is a **tuple** containing a **reference** to the image data on the GPU, along with its **dimensions**.

1.7.1.6 Object detection

Then the detection network processes the image with the **`net.Detect()`** function. It takes the image, the **`width`** and the **`height`** provided by **`camera.CaptureRGBA()`** and returns a **list of detections**:

```
detections = net.Detect(img, width, height)
```

This feature automatically displays detection results above the input image.
If you want, you can add **coordinates**, **trust** and **class** information a print to the terminal for each detection result.
Also see the **`detectNet`** documentation for information on the different members of the **`Detection`** structure that are returned to access them directly in a custom application.

1.7.1.7 Rendering

Finally, we'll visualize the results with **OpenGL** and update the window title to show the current performance:

```
display.RenderOnce(img,width,hight)
display.SetTitle ("Detection | Network {:.0f} FPS".format(net.GetNetworkFPS()))
```

The **`RenderOnce()`** function will automatically return the **backbuffer** which is used when we only have **one frame to render**.

1.7.1.8 Full source code

For completeness, here is the full source of the Python script we just created:

```
import jetson.inference
import jetson.utils
net = jetson.inference.detectNet("ssd-mobilenet-v2", threshold=0.5)
camera = jetson.utils.gstCamera(1280, 720, "/dev/video0") # using V4L2
display = jetson.utils.glDisplay()
while display.IsOpen():
  img, width, height = camera.CaptureRGBA()
  detections = net.Detect(img, width, height)
  display.RenderOnce(img, width, height)
  display.SetTitle("Object Detection | Network {:.0f}  FPS".format(net.GetNetworkFPS()))
```

Note that this release assumes you are using a **V4L2 USB Webcam**. Refer to section **Open the camera feed** above for information on modifying it to use a MIPI camera or support different resolutions.

1.7.1.9 Program execution

To run the application we just coded, simply launch it from a terminal with the Python interpreter:

```
$ python3 my-detection.py
```

To change the results, you can try changing the model loaded with the **detection threshold**.

Lab 2

Facial recognition with prebuilt Python libraries

2.1 Introduction

Facial recognition is actually a series of several related problems:

1. First, look at an image and **find all the faces** in it
2. Second, focus on each face and be able to understand that even if a **face is turned** in a strange direction or in bad lighting, it is still the same person.
3. Third, being able to **choose unique facial features** that you can use to distinguish him from other people, such as eye size, face length, etc.
4. Finally, **compare the unique features** of this face to all the people you already know to determine the person's name.

The human brain is programmed to do all of this automatically.
Computers aren't capable of this kind of high-level generalization (at least not yet…), so we have to teach them to do each step of this process separately.

We need to create a pipeline where we solve each face recognition step separately and pass the result of the current step to the next step. In other words, we have to **chain several machine learning algorithms.**

2.1 Facial recognition — step by step

Let's tackle this problem one step at a time. For each step, we will discover a different machine learning algorithm. We won't fully explain each algorithm to avoid it turning into a book, but you will learn the main ideas behind each and how you can create your own facial recognition system in Python using `OpenFace` and `dlib`.

2.1.1 Step 1: Find all faces

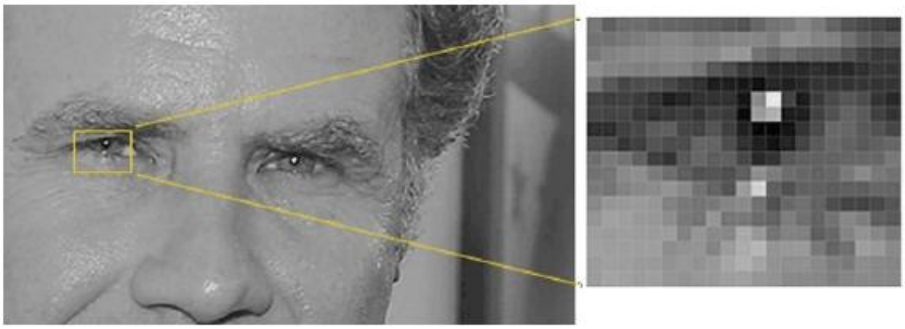
The first step in our pipeline is **face detection**. Obviously, we need to **locate faces** in a photo before we can try to tell them apart!
If you've used a camera in the past 10 years, you've probably seen face detection in action. Face detection is a great feature for cameras. When the camera can **automatically select faces**, it can ensure that all **faces are in focus** before taking the photo. But here we use it for a different purpose: to **find the areas** of the image that we want to pass to the **next stage** of our pipeline.

Face detection became mainstream in the early 2000s when Paul Viola and Michael Jones invented a way to detect faces fast enough to work on inexpensive cameras. However, much more reliable solutions now exist. We'll use a method invented in 2005 called **Histogram of Oriented Gradients** - or just **HOG** for short.

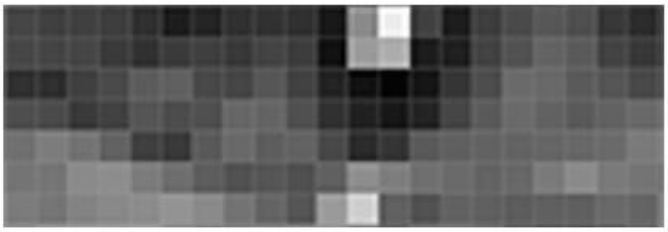
To find the we start by making our images **black and white**.

Next, we'll look at each pixel in our image.

For each pixel, we want to look at the pixels directly surrounding it:



Our goal is to determine how dark the current pixel is relative to the pixels directly surrounding it. Next, we want to draw an **arrow** indicating **in which direction the image darkens.**

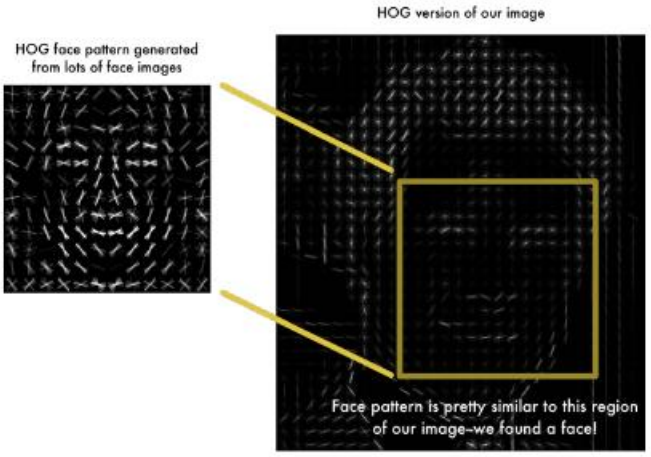


Looking at only this pixel and the pixels touching it, the image darkens towards the upper right corner.
If you repeat this process for each pixel in the image, you end up with each pixel replaced by an arrow. These arrows are called **gradients** and they show the **flow from light to dark** across the whole image:

It may seem random, but there's a really good reason to **replace pixels with gradients**. If we analyze pixels directly, really dark images and really bright images of the same person will have totally different pixel values. But considering only the direction in which the brightness changes, really dark images and really bright images will end up with the exact same representation. This makes the problem much easier to solve!
But saving the gradient for each pixel gives us way too much detail. It would be better if we could just see the **basic light/dark flow** at a higher level so we could see the **basic pattern of the image**.

The end result is that we transform the original image into a very simple representation that captures the basic structure of a face in a simple way.
To do this, we will divide the image into **small squares** of **16x16 pixels** each. In each square, we will count how many gradients point in each main direction (how many point up, point up right, point right, etc...).
Next, we'll replace that square in the image with the **directions of the arrows that were strongest**.

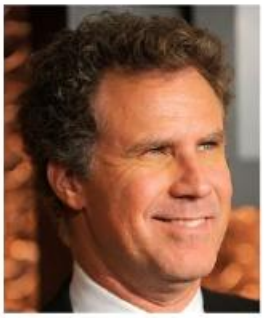
To find faces in this **HOG** image, all we have to do is find the part of our image that most closely resembles a **known HOG pattern** that was extracted from a bunch of other training faces:



If you want to try this step yourself using Python and `dlib`, here is the code showing how to generate and display **HOG** representations of images.

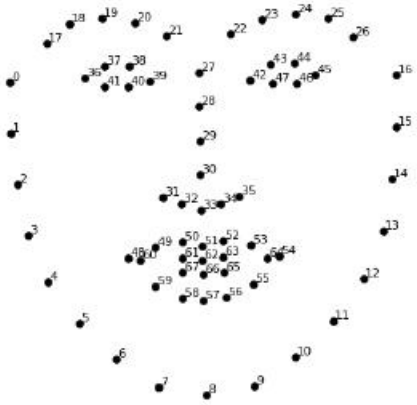
2.1.2 Step 2: Pose and projection of faces

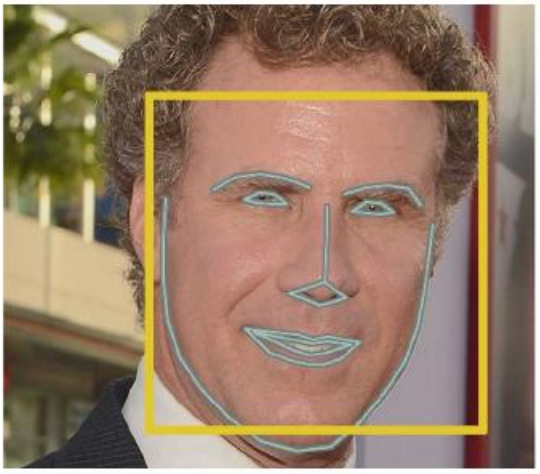
We isolated the faces in our image. But now we have to deal with the problem that faces **facing in different directions** look totally different to a computer:



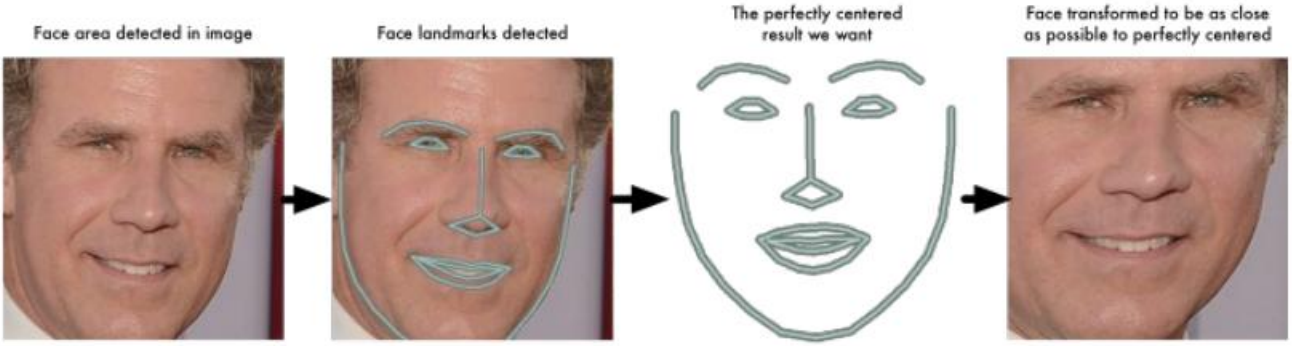
Humans can easily recognize that the two images are of Will Ferrell, but computers would see these images as two completely different people.
To account for this, we'll try to **warp each image** so that the eyes and lips are always in the same location in the image. This will make it much easier for us to compare faces in the next steps.

To do this, we'll use an algorithm called **face landmark estimation**. There are many ways to do this, but we'll use the approach invented in 2014 by Vahid Kazemi and Josephine Sullivan.
The basic idea is that we're going to come up with **68 specific points** (called landmarks) that exist on every face - the top of the chin, the outer edge of each eye, the inner edge of each eyebrow, etc. Next, we will train a machine learning algorithm to be able to find these 68 specific points on any face:





Now that we know where the eyes and mouth are, we'll simply rotate, scale, and shear the image so that the eyes and mouth are centered as best as possible. We will not do fancy 3D warping as this will introduce distortion into the image. We'll only use **basic image transformations** like **rotation** and **scale** that preserve parallel lines (called **affine transformations**)



Now, no matter which way the face is rotated, we are able to center the eyes and the mouth in roughly the same position in the image. This will make our next step much more precise.

2.1.3 Step 3: Encoding faces

We are now at the heart of the problem - **how to distinguish faces**. This is where things get really interesting!
The simplest approach to face recognition is to directly compare the unfamiliar face we found in step 2 with all the photos we have of people who have already been tagged.
When we find a previously tagged face that looks a lot like our unfamiliar face, it has to be the same person, right?
There is actually a huge problem with this approach. A site like Facebook with billions of users and a trillion photos cannot go through all the previously tagged faces to compare them to each newly uploaded image. It would take far too long. They need to be able to recognize faces in milliseconds, not hours.

What we need is a way to extract some basic measurements from each face. Then we could measure our unknown face in the same way and find the known face with the closest measurements. For example, we could measure the **size of each ear**, the **spacing between the eyes**, the **length of the nose**, etc.

2.1.3.1 The most reliable way to measure a face

What metrics should we collect from each face to build our known faces database? Ear size? Nose length? Eye colour? Something else?

It turns out that measurements that seem obvious to us humans (like eye color) don't really make sense to a computer looking at individual pixels in an image. The researchers found that the most accurate approach is to **let the computer calculate the measurements and collect them itself**.

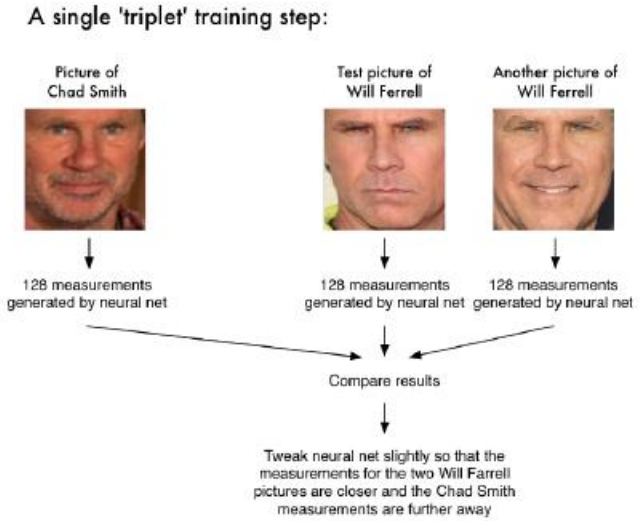
Deep learning does a better job than humans of determining which parts of a face are important to measure.

The solution is to train a **deep convolution neural network** (CNN). But instead of training the network to recognize image objects like we did last time, we're going to **train it to generate 128 measurements for each face**.

The training process works by looking at 3 face images at once:

1. Load a training **face image of a famous person**
2. Upload another photo of **the same famous person**
3. Upload a photo of a **totally different person**

Then the algorithm looks at the metrics it is currently generating for each of those 3 images.
It then **tweaks the neural network slightly** to ensure that the measurements it generates for images 1 and 2 are slightly closer together while ensuring that the measurements for 2 and 3 are slightly further apart:

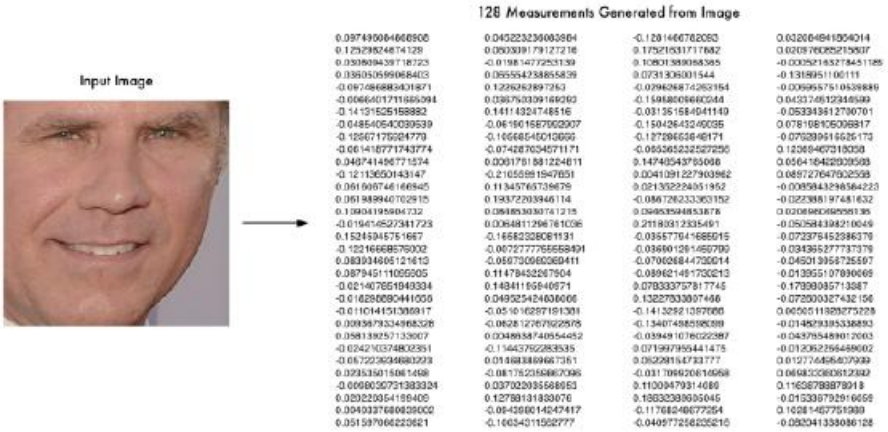


After repeating this step **millions of times for millions of images of thousands of different people**, the neural network learns to reliably generate 128 metrics for each person. Ten different images of the same person should give roughly the same measurements.
Machine-learning folks call the 128 bars on each side an embedding.
The idea of reducing complicated raw data like an image into a computer-generated list of numbers comes up a lot in machine learning.

2.1.3.2 Encode our face image

This process of training a convolutional neural network to produce face integrations requires a lot of data and processing power. Even with an NVidia Tesla GPU card, it takes about 24 hours of continuous training to get good accuracy.
But once the network has been trained, it can generate measurements for any face, even ones it has never seen before! **This step therefore only needs to be done once**.
Fortunately for us, the the folks at **OpenFace** have already done this and they have published several trained networks that we can use directly.
So all we have to do ourselves is run our face images through their **pre-trained network** to get all 128 measurements for each face.
So what parts of the face do these 128 numbers measure exactly? Turns out **we have no idea**. It doesn't really matter to us.
All we care about is that the **network generates roughly the same numbers when looking at two different images of the same person**.

If you want to try this step yourself, **OpenFace** provides a script that will generate embeds of all images in a folder and write them to a `csv` file.



Here are the measurements for our test image.

2.1.4 Step 4: Find the person's name from the encoding

This last step is actually the easiest step in the whole process. All we have to do is find the person in our known people database who has the closest measurements to our test image.
You can do this using any basic machine learning classification algorithm. No fancy deep learning tricks are needed. We will use a simple linear SVM classifier, but many classification algorithms could work.

All we have to do is train a classifier that can take the measurements of a new test image and report which known person is the closest match. This classifier takes milliseconds to run. The **result** of the classifier is the **name of the person**!

So let's try our system. First, we trained a classifier with the embeds of about 20 images each from Will Ferrell, Chad Smith, and Jimmy Falon:

Next, we can run the classifier on each frame of the famous youtube video of Will Ferrell and Chad Smith pretending to be each other on the Jimmy Fallon show:

2.2 Facial recognition system with Nvidia Jetson Nano and Python

2.2.1 Installation of Linux and Python libraries required for facial recognition

Once you have completed the initial Linux/Ubuntu setup, you need to install several libraries that we will use in our facial recognition system.

From the Jetson Nano desktop, open a Terminal window and run the following commands. Each time it asks for your password, enter the same password you entered when creating your user account:

```
sudo apt-get update
sudo apt-get install python3-pip cmake libopenblas-dev liblapack-dev libjpeg-dev
```

First, we are updating **apt**, which is the standard Linux software installation tool that we'll use to install everything else. Next, we are installing some basic libraries with apt that we will need later to compile **numpy** and **dlib**.

Before we go any further, we need to create a **swapfile**. The Jetson Nano **only** has **4GB** of RAM which won't be enough to compile **dlib**. To work around this, we'll set up a **swapfile** which lets us use disk space as extra RAM. Luckily, there is an easy way to set up a swapfile on the Jetson Nano. Just run these two commands:

Install **swap file** (6GB) – your SD card should be a 64 GB one

```
git clone https://github.com/JetsonHacksNano/installSwapfile
./installSwapfile/installSwapfile.sh
```

At this point, you need to **reboot the system** to make sure the swapfile is running. If you skip this, the next step will fail. You can **reboot** from the menu at the top right of the desktop.

```
sudo reboot now
```

When you are logged back in, open up a fresh **Terminal** window and we can continue. First, let's install **numpy**, a Python library that is used for **matrix math calculations**:

```
pip3 install numpy
```

This command will take **15 minutes** since it has to compile **numpy** from scratch. Just wait until it finishes and don't get worried it seems to freeze for a while.

Now we are ready to install **dlib**, a deep learning library created by Davis King that does the heavy lifting for the **face_recognition** library.

However, there is currently a bug in Nvidia's own CUDA libraries for the Jetson Nano that keeps it from working correctly. To work around the bug, we'll have to download **dlib**, **edit a line of code**, and **re-compile it**. But don't worry, it's no big deal.

In Terminal, run these commands:

```
wget http://dlib.net/files/dlib-19.17.tar.bz2 tar jxvf dlib-19.17.tar.bz2cd dlib-19.17
```

That will download and uncompress the source code for dlib. Before we compile it, we need to comment out a line. Run this command:

```
gedit dlib/cuda/cudnn_dlibapi.cpp
```

This will open up the file that we need to edit in a text editor. Search the file for the following line of code (which should **around** line **854**):

```
forward_algo = forward_best_algo;
```

And **comment it out** by adding two slashes in front of it, so it looks like this:

```
//forward_algo = forward_best_algo;
```

Now save the file, close the editor, and go back to the Terminal window. Next, run these commands to compile and install **dlib**:

```
sudo python3 setup.py install
```

This will take around **30–60 minutes** to finish and your Jetson Nano might get hot, but just let it run.

Finally, we need to install the **face_recognition** Python library.
Do that with this command:

```
sudo pip3 install face_recognition
```

Now your Jetson Nano is ready to do face recognition with **full CUDA GPU acceleration**.

On to the fun part!

2.2.2 Testing `face_recognition` package

First, let us download the following git repo:

https://github.com/ageitgey/face_recognition.git

Then we find in **~/face_recognition/examples** directory the following examples and data pictures.

```
jetson@jetson-desktop:~/face_recognition/examples$ ls
alex-lacamoire.png                              find_facial_features_in_picture.py
benchmark.py                                    hamilton_clip.mp4
biden.jpg                                       identify_and_draw_boxes_on_faces.py
blink_detection.py                              ipynb_examples
blur_faces_on_webcam.py                         knn_examples
digital_makeup.py                               lin-manuel-miranda.png
face_distance.py                                obama-1080p.jpg
facerec_from_video_file.py                      obama-240p.jpg
facerec_from_webcam_faster.py                   obama2.jpg
facerec_from_webcam_multiprocessing.py          obama-480p.jpg
facerec_from_webcam.py                          obama-720p.jpg
facerec_ipcamera_knn.py                         obama.jpg
face_recognition_knn.py                         obama_small.jpg
face_recognition_svm.py                         recognize_faces_in_pictures.py
facerec_on_raspberry_pi.py                      short_hamilton_clip.mp4
facerec_on_raspberry_pi_Simplified_Chinese.py   two_people.jpg
find_faces_in_batches.py                        web_service_example.py
find_faces_in_picture_cnn.py                    web_service_example_Simplified_Chinese.py
find_faces_in_picture.py
jetson@jetson-desktop:~/face_recognition/examples$
```

2.2.2.1 First example: `find_faces_in_picture.py`

This code finds all the faces in the image using the **default HOG-based** model. This method is fairly accurate, but not as accurate as the CNN model and not GPU accelerated.

See also: **find_faces_in_picture_cnn.py**

```
from PIL import Image
import face_recognition

# Load the jpg file into a numpy array
image = face_recognition.load_image_file("biden.jpg")
face_locations = face_recognition.face_locations(image)

print("I found {} face(s) in this photograph.".format(len(face_locations)))
```

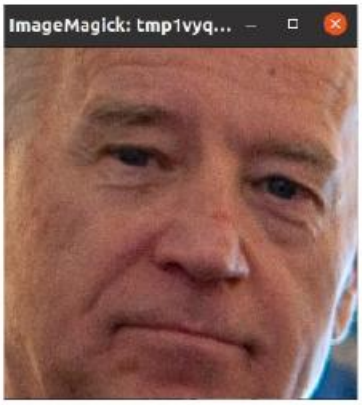
```
for face_location in face_locations:

    # Print the location of each face in this image
    top, right, bottom, left = face_location
    print("A face is located at pixel location Top: {}, Left: {}, Bottom: {}, Right: {}".format(top,
left, bottom, right))

    # You can access the actual face itself like this:
    face_image = image[top:bottom, left:right]
    pil_image = Image.fromarray(face_image)
    pil_image.show()
```

Result:

```
jetson@jetson-desktop:~/face_recognition/examples$ python3 find_faces_in_picture_cnn.py
I found 1 face(s) in this photograph.
A face is located at pixel location Top: 235, Left: 428, Bottom: 518, Right: 712
jetson@jetson-desktop:~/face_recognition/examples$
```



To do:

Put the following line:

```
face_locations = face_recognition.face_locations(image, number_of_times_to_upsample=0, model="cnn")
```

instead of:

```
face_locations = face_recognition.face_locations(image)
```

2.2.2.2 Second example: `find_faces_in_picture.py`

This example shows the facial features of two persons found in the picture.

```
from PIL import Image, ImageDraw
import face_recognition
# Load the jpg file into a numpy array
image = face_recognition.load_image_file("two_people.jpg")
# Find all facial features in all the faces in the image
face_landmarks_list = face_recognition.face_landmarks(image)
print("I found {} face(s) in this photograph.".format(len(face_landmarks_list)))
# Create a PIL imagedraw object so we can draw on the picture
pil_image = Image.fromarray(image)
d = ImageDraw.Draw(pil_image)

for face_landmarks in face_landmarks_list:

    # Print the location of each facial feature in this image
    for facial_feature in face_landmarks.keys():
        print("The {} in this face has the following points: {}".format(facial_feature,
face_landmarks[facial_feature]))

    # Let's trace out each facial feature in the image with a line!
    for facial_feature in face_landmarks.keys():
        d.line(face_landmarks[facial_feature], width=5)

# Show the picture
```

```
pil_image.show()
```

Result:

```
jetson@jetson-desktop:~/face_recognition/examples$ python3 find_facial_features_in_picture.py
I found 2 face(s) in this photograph.
The chin in this face has the following points: [(789, 129), (788, 150), (787, 171), (788, 191),
(795, 211), (808, 227), (824, 239), (842, 249), (862, 253), (881, 254), (897, 247), (911, 236),
(921, 222), (928, 206), (933, 188), (937, 171), (940, 153)]
The left_eyebrow in this face has the following points: [(821, 110), (831, 102), (845, 101), (860,
104), (872, 111)]
The right_eyebrow in this face has the following points: [(891, 117), (905, 116), (918, 118), (929,
124), (935, 135)]
The nose_bridge in this face has the following points: [(881, 128), (880, 141), (880, 154), (879,
167)]
The nose_tip in this face has the following points: [(857, 174), (865, 178), (874, 181), (881, 181),
(888, 180)]
The left_eye in this face has the following points: [(834, 124), (843, 122), (851, 124), (858, 129),
(850, 129), (841, 127)]
The right_eye in this face has the following points: [(896, 138), (906, 136), (914, 138), (920,
143), (913, 144), (904, 142)]
The top_lip in this face has the following points: [(838, 198), (852, 196), (863, 196), (870, 200),
(878, 199), (887, 203), (895, 209), (891, 207), (877, 203), (869, 202), (862, 200), (842, 198)]
The bottom_lip in this face has the following points: [(895, 209), (884, 210), (875, 208), (867,
207), (859, 205), (850, 202), (838, 198), (842, 198), (861, 199), (869, 201), (877, 202), (891,
207)]
The chin in this face has the following points: [(231, 119), (233, 138), (235, 156), (239, 175),
(247, 192), (260, 205), (277, 215), (296, 222), (317, 226), (337, 225), (352, 218), (365, 206),
(373, 192), (378, 177), (382, 161), (384, 144), (385, 129)]
The left_eyebrow in this face has the following points: [(259, 92), (270, 81), (285, 74), (301, 73),
(317, 78)]
The right_eyebrow in this face has the following points: [(339, 80), (352, 78), (366, 82), (377,
91), (382, 105)]
The nose_bridge in this face has the following points: [(328, 95), (328, 105), (329, 114), (329,
124)]
The nose_tip in this face has the following points: [(306, 138), (315, 140), (326, 143), (334, 142),
(343, 141)]
The left_eye in this face has the following points: [(275, 101), (284, 95), (293, 95), (302, 101),
(293, 102), (284, 102)]
The right_eye in this face has the following points: [(344, 106), (353, 102), (362, 104), (369,
111), (361, 110), (352, 108)]
The top_lip in this face has the following points: [(285, 170), (301, 160), (314, 156), (324, 158),
(332, 157), (342, 164), (351, 176), (346, 175), (332, 167), (323, 166), (313, 165), (291, 170)]
The bottom_lip in this face has the following points: [(351, 176), (340, 178), (330, 178), (321,
178), (311, 176), (299, 174), (285, 170), (291, 170), (313, 167), (322, 168), (331, 169), (346,
175)]
jetson@jetson-desktop:~/face_recognition/examples$
```

2.3 Projects

In this section we are going to develop two face-recognition based projects.
The first project uses face_recognition functions to detect new faces and to send the message(label of the face) to the corresponding topics on an MQTT server.

2.3.1 Sending (publishing) an MQTT message

To write the Python code using MQTT services we need first to install the python library paho-mqtt. You need '**pip3**' to install this module, so if you have not already done so, you will need to install **pip3**:

```
sudo apt-get install python3-pip
```

Now you can install **paho-mqtt**:

```
sudo pip3 install paho-mqtt
```

The following is a very simple script to **publish** a message:

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

import paho.mqtt.client as mqtt

# This is the Publisher

client = mqtt.Client()
client.connect("broker.emqx.io",1883,60)
client.publish("face_recognition/test", "John");
client.disconnect();
```

2.3.2 Face Recognition Program and publisher to MQTT broker

This is a demo of running face recognition on live video from your webcam. It's a little more complicated than the other example, but it includes some basic performance tweaks to make things run a lot faster:
1. Process each video frame at 1/4 resolution (though still display it at full resolution)
2. Only detect faces in every other frame of video.

Note: This example requires OpenCV (the `cv2` library) to be installed only to read from your webcam. OpenCV is *not* required to use the **`face_recognition`** library. It's only required if you want to run this specific demo.

```
import face_recognition
import cv2
import numpy as np
import paho.mqtt.client as mqtt
import time


# sending to MQTT server
def publish(mess):
        client = mqtt.Client()
        client.connect("broker.emqx.io",1883,60)
        client.publish("face_recognition/test", mess);
        client.disconnect();

# Get a reference to webcam #0 (the default one)
cap = cv2.VideoCapture(0)

def make_1080p():
    cap.set(3, 1920)
    cap.set(4, 1080)

def make_720p():
    cap.set(3, 1280)
    cap.set(4, 720)

def make_480p():
    cap.set(3, 640)
    cap.set(4, 480)

def change_res(width, height):
    cap.set(3, width)
    cap.set(4, height)

#make_720p()
#change_res(1280, 720)

make_480p()
change_res(640, 480)

# Load a sample picture and learn how to recognize it.
obama_image = face_recognition.load_image_file("obama.jpg")
obama_face_encoding = face_recognition.face_encodings(obama_image)[0]

# Load a second sample picture and learn how to recognize it.
biden_image = face_recognition.load_image_file("biden.jpg")
biden_face_encoding = face_recognition.face_encodings(biden_image)[0]

# Load a second sample picture and learn how to recognize it.
cathy_image = face_recognition.load_image_file("cathy.jpg")
cathy_face_encoding = face_recognition.face_encodings(cathy_image)[0]

# Load a second sample picture and learn how to recognize it.
bako_image = face_recognition.load_image_file("bako.jpg")
bako_face_encoding = face_recognition.face_encodings(bako_image)[0]

# Create arrays of known face encodings and their names
known_face_encodings = [
    obama_face_encoding,
    bako_face_encoding,
    biden_face_encoding
]
known_face_names = [
    "Barack Obama",
    "Bako",
    "Joe Biden"
]

# Initialize some variables
face_locations = []
face_encodings = []
face_names = []
process_this_frame = True

while True:
    # Grab a single frame of video
    ret, frame = cap.read()
    # Resize frame of video to 1/4 size for faster face recognition processing
    small_frame = cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)
    # Convert the image from BGR color (which OpenCV uses) to RGB color)
    rgb_small_frame = small_frame[:, :, ::-1]
    # Only process every other frame of video to save time
    if process_this_frame:
        # Find all the faces and face encodings in the current frame of video
```

```
        face_locations = face_recognition.face_locations(rgb_small_frame)
        face_encodings = face_recognition.face_encodings(rgb_small_frame, face_locations)

        face_names = []
        for face_encoding in face_encodings:
            # See if the face is a match for the known face(s)
            matches = face_recognition.compare_faces(known_face_encodings, face_encoding)
            name = "Unknown"
            # # If a match was found in known_face_encodings, just use the first one.
            # if True in matches:
            #     first_match_index = matches.index(True)
            #     name = known_face_names[first_match_index]

            # Or instead, use the known face with the smallest distance to the new face
            face_distances = face_recognition.face_distance(known_face_encodings, face_encoding)
            best_match_index = np.argmin(face_distances)
            if matches[best_match_index]:
                name = known_face_names[best_match_index]

            face_names.append(name)

    process_this_frame = not process_this_frame

    # Display the results
    for (top, right, bottom, left), name in zip(face_locations, face_names):
        # Scale back up face locations since the frame we detected in was scaled to 1/4 size
        top *= 4
        right *= 4
        bottom *= 4
        left *= 4

        # Draw a box around the face
        cv2.rectangle(frame, (left, top), (right, bottom), (0, 0, 255), 2)

        # Draw a label with a name below the face
        cv2.rectangle(frame, (left, bottom - 35), (right, bottom), (0, 0, 255), cv2.FILLED)
        font = cv2.FONT_HERSHEY_DUPLEX
        cv2.putText(frame, name, (left + 6, bottom - 6), font, 1.0, (255, 255, 255), 1)
        publish(name)

    # Display the resulting image
    cv2.imshow('Video', frame)

    # Hit 'q' on the keyboard to quit!
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release handle to the webcam
cap.release()
cv2.destroyAllWindows()
```

To do:

1. Code the above programs on your Jetson Nano.
2. Enter **your face images with the WEBCAM** using **`cheese`** program (the images remain on the SD card) in the directory where your Python program is located.
3. Modify the code to integrate the encoding of your images.
4. Launch My_MQTT application on your smartphone (**`broker.emqx.io,1883`**) to see the published messages.

2.3.3 Second Project - Doorbell Camera Demo App

2.3.3.1 Introduction and the complete code

Lets us detect faces, turn each detected face into a unique face encoding, and then compare those face encodings to see if it's likely the same person, all with just a few lines of code.
Using the **`dlib`** library, we may create a doorbell camera app that can recognize people walking towards the front door and track each time the person returns.

Here's what it looks like when you run it:

Here is the complete code of the program, the explanation of this code (**code walkthrough**) is given below.

```
import face_recognition
import cv2
from datetime import datetime, timedelta
import numpy as np
import platform
import pickle

# Our list of known face encodings and a matching list of metadata about each face.
known_face_encodings = []
known_face_metadata = []

def save_known_faces():
    with open("known_faces.dat", "wb") as face_data_file:
        face_data = [known_face_encodings, known_face_metadata]
        pickle.dump(face_data, face_data_file)
        print("Known faces backed up to disk.")

def load_known_faces():
    global known_face_encodings, known_face_metadata

    try:
        with open("known_faces.dat", "rb") as face_data_file:
            known_face_encodings, known_face_metadata = pickle.load(face_data_file)
            print("Known faces loaded from disk.")
    except IOError:
        print("No previous face data found - starting with a blank known face list.")
        pass


def running_on_jetson_nano():
    return platform.machine() == "aarch64"
```

```
def get_jetson_gstreamer_source(capture_width=1280, capture_height=720, display_width=1280,
display_height=720, framerate=60, flip_method=0):
    """
    Return an OpenCV-compatible video source
    """
    return ('nvarguscamerasrc ! video/x-raw(memory:NVMM),' +
            'width=(int){capture_width}, height=(int){capture_height}, ' +
            'format=(string)NV12, framerate=(fraction){framerate}/1 ! ' +
            'nvvidconv flip-method={flip_method} ! ' +
            'video/x-raw, width=(int){display_width}, height=(int){display_height},
format=(string)BGRx ! ' +
            'videoconvert ! video/x-raw, format=(string)BGR ! appsink'
            )


def register_new_face(face_encoding, face_image):
    """
    Add a new person to our list of known faces
    """
    # Add the face encoding to the list of known faces
    known_face_encodings.append(face_encoding)
    # Add a matching dictionary entry to our metadata list.
    # to keep track of how many times a person has visited, when we last saw them, etc.
    known_face_metadata.append({
        "first_seen": datetime.now(),
        "first_seen_this_interaction": datetime.now(),
        "last_seen": datetime.now(),
        "seen_count": 1,
        "seen_frames": 1,
        "face_image": face_image,
    })

def lookup_known_face(face_encoding):
    """
    See if this is a face we already have in our face list
    """
    metadata = None

    # If our known face list is empty, just return nothing since we can't possibly have seen this
face.
    if len(known_face_encodings) == 0:
        return metadata

    # Calculate the face distance between the unknown face and every face

    face_distances = face_recognition.face_distance(known_face_encodings, face_encoding)

    # Get the known face that had the lowest distance (i.e. most similar) from the unknown face.

    best_match_index = np.argmin(face_distances)

    # If the face with the lowest distance had a distance under 0.6, - face match
    if face_distances[best_match_index] < 0.65:
        # If we have a match, look up the metadata we've saved for it
        metadata = known_face_metadata[best_match_index]

        # Update the metadata for the face so we can keep track of how recently
        metadata["last_seen"] = datetime.now()
        metadata["seen_frames"] += 1

        # We'll also keep a total "seen count" that tracks how many times this person
        # if we have seen this person within the last 5 minutes is still the same
        # visit, not a new visit. But if they go away for awhile and come back, that is a new visit.
        if datetime.now() - metadata["first_seen_this_interaction"] > timedelta(minutes=5):
            metadata["first_seen_this_interaction"] = datetime.now()
            metadata["seen_count"] += 1

    return metadata

def main_loop():
    # Get access to the webcam.
    cap= cv2.VideoCapture(0)

    cap.set(3, 1280)
    cap.set(4, 720)
```

```
    # Track how long since we last saved a copy of our known faces to disk as a backup.
    number_of_faces_since_save = 0

    while True:
        # Grab a single frame of video
        ret, frame = video_capture.read()

        # Resize frame of video to 1/4 size for faster face recognition processing
        small_frame = cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)


        # Convert the image from BGR color (which OpenCV uses) to RGB color
        rgb_small_frame = small_frame[:, :, ::-1]

        # Find all the face locations and face encodings in the current frame of video
        face_locations = face_recognition.face_locations(rgb_small_frame)
        face_encodings = face_recognition.face_encodings(rgb_small_frame, face_locations)

        # Loop through each detected face and see if it is one we have seen before
        # If so, we'll give it a label that we'll draw on top of the video.
        face_labels = []
        for face_location, face_encoding in zip(face_locations, face_encodings):
            # See if this face is in our list of known faces.
            metadata = lookup_known_face(face_encoding)

            # If we found the face, label the face with some useful information.
            if metadata is not None:
                time_at_door = datetime.now() - metadata['first_seen_this_interaction']
                face_label = "At door " + str(int(time_at_door.total_seconds())) + "s"

            # If this is a brand new face, add it to our list of known faces
            else:
                face_label = "New visitor!"

                # Grab the image of the the face from the current frame of video
                top, right, bottom, left = face_location
                face_image = small_frame[top:bottom, left:right]
                face_image = cv2.resize(face_image, (150, 150))

                # Add the new face to our known face data
                register_new_face(face_encoding, face_image)

            face_labels.append(face_label)

        # Draw a box around each face and label each face
        for (top, right, bottom, left), face_label in zip(face_locations, face_labels):
            # Scale back up face locations since the frame we detected in was scaled to 1/4 size
            top *= 4
            right *= 4
            bottom *= 4
            left *= 4

            # Draw a box around the face
            cv2.rectangle(frame, (left, top), (right, bottom), (0, 0, 255), 2)

            # Draw a label with a name below the face
            cv2.rectangle(frame, (left, bottom - 35), (right, bottom), (0, 0, 255), cv2.FILLED)
            cv2.putText(frame,face_label,(left+6,bottom-6),cv2.FONT_HERSHEY_DUPLEX,0.8,(255,255,
255),1)

        # Display recent visitor images
        number_of_recent_visitors = 0
        for metadata in known_face_metadata:
            # If we have seen this person in the last minute, draw their image
            if datetime.now()-metadata["last_seen"]<timedelta(seconds=10) and
metadata["seen_frames"] > 5:
                # Draw the known face image
                x_position = number_of_recent_visitors * 150
                frame[30:180, x_position:x_position + 150] = metadata["face_image"]
                number_of_recent_visitors += 1

                # Label the image with how many times they have visited
                visits = metadata['seen_count']
                visit_label = f"{visits} visits"
                if visits == 1:
                    visit_label = "First visit"
```

```
                cv2.putText(frame,visit_label,(x_position+10,170),cv2.FONT_HERSHEY_DUPLEX,0.6,(255,
255, 255), 1)

        if number_of_recent_visitors > 0:
            cv2.putText(frame,"Visitors at Door",(5,18),cv2.FONT_HERSHEY_DUPLEX,0.8,(255,255,255),
1)



        # Display the final frame of video with boxes drawn around each detected fames
        cv2.imshow('Video', frame)

        # Hit 'q' on the keyboard to quit!
        if cv2.waitKey(1) & 0xFF == ord('q'):
            save_known_faces()
            break

        # We need to save our known faces back to disk every so often in case something crashes.
        if len(face_locations) > 0 and number_of_faces_since_save > 100:
            save_known_faces()
            number_of_faces_since_save = 0
        else:
            number_of_faces_since_save += 1

    # Release handle to the webcam
    video_capture.release()
    cv2.destroyAllWindows()


if __name__ == "__main__":
    load_known_faces()
    main_loop()
```

2.3.3.2 Step-by-step explanation of Python code

The code starts by importing the libraries that we are going to use. The most important are **OpenCV** (called **cv2** in Python), which we will use to read camera images, and **face_recognition**, which we will use to **detect** and **compare** faces.

```
import face_recognition
import cv2
from datetime import datetime, timedelta
import numpy as np
import platform
import pickle
```

We need to know how to access the camera - getting a picture from a Raspberry Pi camera module works differently than a **USB WEBCAM**. So just change this variable to True or False depending on your hardware:

```
# Set this depending on your camera type:
# - True = Raspberry Pi 2.x camera module
# - False = USB webcam or other USB video input (like an HDMI capture device)
USING_RPI_CAMERA_MODULE = False
```

Next, we'll create variables to store data about people walking past our camera. These variables will act as a simple database of known visitors.
This app is only a demo, so we store our known faces in a Python list. In a real-world application that processes more faces, you might want to use a real database instead.

```
known_face_encodings = []
known_face_metadata = []
```

Next we have a function to save and load known face data. Here is the **backup function**:

```
def save_known_faces():
    with open("known_faces.dat", "wb") as face_data_file:
        face_data = [known_face_encodings, known_face_metadata]
        pickle.dump(face_data, face_data_file)
        print("Known faces backed up to disk.")
```

This writes known faces to disk using Python's built-in **pickle.dump** feature. The data is reloaded in the same way.
Each time our program detects a new face, we'll call a function to add it to our known faces database:

```
def register_new_face(face_encoding, face_image):
    known_face_encodings.append(face_encoding)known_face_metadata.append({
        "first_seen": datetime.now(),
        "first_seen_this_interaction": datetime.now(),
        "last_seen": datetime.now(),
        "seen_count": 1,
        "seen_frames": 1,
        "face_image": face_image,
    })
```

First, we store the face encoding that represents the face in a list. Next, we store a dictionary corresponding to the face data in a second list.
We'll use it to track the time we first saw the person, how long they've been near the camera recently, how many times they've visited our home, and a small picture of their face.

We also need a **helper function** to check if an unknown face is already in our face database or not:

```
def lookup_known_face(face_encoding):
    metadata = None
    if len(known_face_encodings) == 0:
        return metadata
        face_distances = face_recognition.face_distance(known_face_encodings,face_encoding)
        best_match_index = np.argmin(face_distances)
        if face_distances[best_match_index] < 0.65:
          metadata = known_face_metadata[best_match_index]
          metadata["last_seen"] = datetime.now()
          metadata["seen_frames"] += 1
        if datetime.now() - metadata["first_seen_this_interaction"] > timedelta(minutes=5):
          metadata["first_seen_this_interaction"] = datetime.now()
          metadata["seen_count"] += 1
        return metadata
```

We do a few important things here:

1. Using the **`face_recognition`** library, we check how similar the unknown face is to all previous visitors. The **`face_distance()`** function gives us a numerical measure of how similar the unknown face is to all known faces — the smaller the number (distance), the more similar the faces.
2. If the face is very similar to one of our known visitors, we assume it is a regular visitor. In this case, we update their "**last seen**" time and increment the number of times we've seen them in a video frame.
3. Finally, if this person has been seen on camera within **the last five minutes**, we assume they are still there on the same visit. Otherwise, we'll assume it's a new visit to us, so we'll reset the tracking timestamp to their most recent visit.

The rest of the program is the main **`loop`** - **an endless loop** where we grab a video frame, search for faces in the frame, and process each face we see. This is the heart of the program. Let's look at this:

```
def main_loop():
    if USING_RPI_CAMERA_MODULE:
        video_capture =
            cv2.VideoCapture(
                get_jetson_gstreamer_source(),
                cv2.CAP_GSTREAMER
            )
    else:
        video_capture = cv2.VideoCapture(0)
```

The first step is to access the camera using the appropriate method for our computer hardware (RaspiCAM or WEBCAM USB).

Now let's start capturing the video footage:

```
while True:
    # Grab a single frame of video
    ret, frame = video_capture.read()    # Resize frame of video to 1/4 size
    small_frame = cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)    # Convert the image from BGR color
    rgb_small_frame = small_frame[:, :, ::-1]
```

Every time we fetch a frame from video, we will also **scale it down to 1/4**. This will speed up the facial recognition process at the expense of only detecting larger faces in the image. But since we're building a doorbell camera that only recognizes people near the camera, that shouldn't be a problem.

We also have to deal with the fact that **OpenCV** extracts the camera images with each pixel stored as a Blue-Green-Red (**BGR**) value instead of the standard Red-Green-Blue (**RGB**) order. Before we can run facial recognition on the image, we need to convert the format of the image.

We can now detect all faces in the image and convert each face to a face encoding. It only takes two lines of code:

```
face_locations = face_recognition.face_locations(rgb_small_frame)face_encodings =
face_recognition.face_encodings(
                    rgb_small_frame,
                    face_locations
                )
```

Next, we'll go through each detected face and decide if it's someone we've seen in the past or a brand new visitor:

```
for face_location, face_encoding in zip( face_locations, face_encodings):
  metadata = lookup_known_face(face_encoding)
  if metadata is not None:
    time_at_door = datetime.now() - metadata['first_seen_this_interaction']
    face_label = f"At door {int(time_at_door.total_seconds())}s"    else:
    face_label = "New visitor!"        # Grab the image of the face
    top, right, bottom, left = face_location
    face_image = small_frame[top:bottom, left:right]
    face_image = cv2.resize(face_image, (150, 150))  # Add the new face to our known face data
    register_new_face(face_encoding, face_image)
```

If we've seen the person before, we'll retrieve the metadata we've stored about their previous visits. Otherwise, we'll add them to our face database and grab their face photo from the video frame to add it to our database. Now that we've found all the people and determined their identities, we can loop over the detected faces again just to draw boxes around each face and add a label to each face:

```
for (top, right, bottom, left), face_label in
                  zip(face_locations, face_labels):
    # Scale back up face location
    # since the frame we detected in was 1/4 size
    top *= 4
    right *= 4
    bottom *= 4
    left *= 4    # Draw a box around the face
    cv2.rectangle(
        frame, (left, top), (right, bottom), (0, 0, 255), 2
    )    # Draw a label with a description below the face
    cv2.rectangle(
        frame, (left, bottom - 35), (right, bottom),
        (0, 0, 255), cv2.FILLED
    )
    cv2.putText(
        frame, face_label,
        (left + 6, bottom - 6),
        cv2.FONT_HERSHEY_DUPLEX, 0.8,
        (255, 255, 255), 1
  )
```

We also add a **running list** of recent visitors drawn at the top of the screen with the number of times they have visited your house.
To draw this, we need to cycle through all known faces and see which ones have been on camera recently.
For each recent visitor, we will draw their face image on the screen and establish a number of visits

```
number_of_recent_visitors = 0
for metadata in known_face_metadata:
    # If we have seen this person in the last minute
    if datetime.now() - metadata["last_seen"]  < timedelta(seconds=10):
      # Draw the known face image
        x_position = number_of_recent_visitors * 150
        frame[30:180, x_position:x_position + 150] =
              metadata["face_image"]number_of_recent_visitors += 1
      # Label the image with how many times they have visited
        visits = metadata['seen_count']
        visit_label = f"{visits} visits"if visits == 1:
            visit_label = "First visit"cv2.putText(
            frame, visit_label,
            (x_position + 10, 170),
            cv2.FONT_HERSHEY_DUPLEX, 0.6,
            (255, 255, 255), 1
        )
```

Finally, we can display the current frame of the video on screen with all our annotations drawn on it:

```
cv2.imshow('Video', frame)
```

And to make sure we don't lose data if the program crashes, we'll save our list of known faces to disk every 100 frames:

```
if len(face_locations) > 0 and number_of_frames_since_save > 100:
    save_known_faces()
    number_of_faces_since_save = 0
else:
    number_of_faces_since_save += 1
```

And that's all but a line or two of cleanup code to turn off the camera when the program exits.

The program start code is at the very bottom of the program:

```
if __name__ == "__main__":
    load_known_faces()
    main_loop()
```

All we do is load known faces (if any) and then start the main loop which continuously reads from the camera and displays the results on screen.
The entire program is only about 200 lines long, but it detects visitors, identifies them, and tracks each time they return to your door.

To do:

1. Prepare and execute the above program of your board.
2. Discuss the possible extensions , for example sending alert messages to the external server and/or your phone.

Lab 3

EAI audio with `jetson-voice`

`jetson-voice` is an ASR/NLP/TTS **deep learning inference library** for Jetson Nano, TX1/TX2, Xavier NX, and AGX Xavier. It supports Python and JetPack 4.4.1 or newer. The **DNN** models were trained with NeMo and deployed with TensorRT for optimized performance.
All computation is performed using the onboard **GPU**.

Currently the following capabilities are included:

- Automatic Speech Recognition (ASR)
  - Streaming ASR (QuartzNet)
  - Command/Keyword Recognition (MatchboxNet)
  - Voice Activity Detection (VAD Marblenet)
- Natural Language Processing (NLP)
  - Joint Intent/Slot Classification
  - Text Classification (Sentiment Analysis)
  - Token Classification (Named Entity Recognition)
  - Question/Answering (QA)
- Text-to-Speech (TTS)

The **NLP** models are using the DistilBERT transformer architecture for reduced memory usage and increased performance. For samples of the text-to-speech output, see the TTS Audio Samples section below.

3.1 Running the Container

jetson-voice is distributed as a Docker container due to the number of dependencies. There are pre-built containers images available on **DockerHub** for **JetPack 4.4.1** and newer:

```
dustynv/jetson-voice:r32.4.4    # JetPack 4.4.1 (L4T R32.4.4)
dustynv/jetson-voice:r32.5.0    # JetPack 4.5 (L4T R32.5.0) / JetPack 4.5.1 (L4T
R32.5.1)
dustynv/jetson-voice:r32.6.1    # JetPack 4.6 (L4T R32.6.1)
dustynv/jetson-voice:r32.7.1    # JetPack 4.6.1 (L4T R32.7.1)
```

To download and run the container, you can simply clone this repo and use the **`docker/run.sh`** script:

```
$ git clone --branch dev https://github.com/dusty-nv/jetson-voice
$ cd jetson-voice
$ docker/run.sh
```

note: if you want to use a USB microphone or speaker, plug it in *before* you start the container

There are some optional arguments to `docker/run.sh` that you can use:

- `-r` (`--run`) specifies a run command, otherwise the container will start in an interactive shell.
- `-v` (`--volume`) mount a directory from the host into the container (`/host/path:/container/path`)
- `--dev` starts the container in development mode, where all the source files are mounted for easy editing

The run script will automatically mount the `data/` directory into the container, which stores the models and other data files. If you save files from the container there, they will also show up under `data/` on the host.

3.2 Automatic Speech Recognition (ASR)

The speech recognition in **jetson-voice** is a streaming service, so it's intended to be used on live sources and transcribes the audio in 1-second chunks. It uses a QuartzNet-15x5 model followed by a CTC beamsearch decoder and language model, to further refine the raw output of the network. It detects breaks in the audio to determine the end of sentences. For information about using the ASR APIs, please refer to `jetson_voice/asr.py` and see `examples/asr.py`

After you start the container, first run a test audio file (`wav/ogg/flac`) through `examples/asr.py` to verify that the system is functional. Run this command (and all subsequent commands) inside the container:

```
$ examples/asr.py --wav data/audio/dusty.wav

hi
hi hi this is dust
hi hi this is dusty check
hi hi this is dusty check one two
hi hi this is dusty check one two three
hi hi this is dusty check one two three.

what's the weather or
what's the weather going to be tomorrow
what's the weather going to be tomorrow in pittsburgh
what's the weather going to be tomorrow in pittsburgh.

today is
today is wednesday
today is wednesday tomorrow is thursday
today is wednesday tomorrow is thursday.

i would like
i would like to order a large
i would like to order a large pepperoni pizza
i would like to order a large pepperoni pizza.

is it going to be
is it going to be cloudy tomorrow.
```

The first time you run each model, **TensorRT** will take **a few minutes** to optimize it.
This optimized model is then cached to disk, so the next time you run the model it will load faster.

3.2.1.1 Live Microphone

To test the ASR on a mic, first list the audio devices in your system to get the audio device ID's:

```
$ scripts/list_audio_devices.sh

----------------------------------------------------
 Audio Input Devices
----------------------------------------------------
Input Device ID 1 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,0)' (inputs=16)
(sample_rate=44100)
Input Device ID 2 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,1)' (inputs=16)
(sample_rate=44100)
Input Device ID 3 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,2)' (inputs=16)
(sample_rate=44100)
Input Device ID 4 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,3)' (inputs=16)
(sample_rate=44100)
Input Device ID 5 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,4)' (inputs=16)
(sample_rate=44100)
Input Device ID 6 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,5)' (inputs=16)
(sample_rate=44100)
Input Device ID 7 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,6)' (inputs=16)
(sample_rate=44100)
Input Device ID 8 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,7)' (inputs=16)
(sample_rate=44100)
Input Device ID 9 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,8)' (inputs=16)
(sample_rate=44100)
Input Device ID 10 - 'tegra-snd-t210ref-mobile-rt565x: - (hw:1,9)' (inputs=16)
(sample_rate=44100)
Input Device ID 11 - 'Logitech H570e Mono: USB Audio (hw:2,0)' (inputs=2)
(sample_rate=44100)
Input Device ID 12 - 'Samson Meteor Mic: USB Audio (hw:3,0)' (inputs=2)
(sample_rate=44100)
```

> If you don't see your audio device listed, exit and restart the container.
> USB devices should be attached ***before*** the container is started.

Then run the ASR example with the **`--mic <DEVICE>`** option, and specify either the device ID or name:

```
$ examples/asr.py --mic 11

hey
hey how are you guys
hey how are you guys.

# (Press Ctrl+C to exit)
```

3.3 ASR Classification

There are other ASR models included for command/keyword recognition (MatchboxNet) and voice activity detection (VAD MarbleNet). These models are smaller and faster, and classify chunks of audio as opposed to transcribing text.

3.3.1 Command/Keyword Recognition

The MatchboxNet model was trained on **12 keywords** from the Google Speech Commands dataset:

```
# MatchboxNet classes
"yes",
"no",
```

```
"up",
"down",
"left",
"right",
"on",
"off",
"stop",
"go",
"unknown",
"silence"
```

You can run it through the same ASR example as above by specifying the **`--model matchboxnet`** argument:

```
$ examples/asr.py --model matchboxnet --wav data/audio/commands.wav
```

```
class 'unknown' (0.384)
class 'yes' (1.000)
class 'no' (1.000)
class 'up' (1.000)
class 'down' (1.000)
class 'left' (1.000)
class 'left' (1.000)
class 'right' (1.000)
class 'on' (1.000)
class 'off' (1.000)
class 'stop' (1.000)
class 'go' (1.000)
class 'go' (1.000)
class 'silence' (0.639)
class 'silence' (0.576)
```

The numbers printed on the right are the classification probabilities between 0 and 1.

3.3.2 Voice Activity Detection (VAD)

The voice activity model (VAD MarbleNet) is a binary model that outputs `background` or `speech`:

```
$ examples/asr.py --model vad_marblenet --wav data/audio/commands.wav
```

```
class 'background' (0.969)
class 'background' (0.984)
class 'background' (0.987)
class 'speech' (0.997)
class 'speech' (1.000)
class 'speech' (1.000)
class 'speech' (0.998)
class 'background' (0.987)
class 'speech' (1.000)
class 'speech' (1.000)
class 'speech' (1.000)
class 'background' (0.988)
class 'background' (0.784)
```

3.4 Natural Language Processing (NLP)

There are two samples included for NLP:

- **`examples/nlp.py`** (intent/slot, text classification, token classification)
- **`examples/nlp_qa.py`** (question/answering)

These each use a **DistilBERT** model which has been fined-tuned for it's particular task. For information about using the NLP APIs, please refer to **`jetson_voice/nlp.py`** and see the samples above.

3.4.1 Joint Intent/Slot Classification

Joint Intent and Slot classification is a task of classifying an Intent and detecting all relevant Slots (Entities) for this Intent in a query. For example, in the query: **`What is the weather in Santa Clara tomorrow morning?`**, we would like to classify the query as a **`weather`** **Intent**, and detect **`Santa Clara`** as a **location slot** and **`tomorrow morning`** as a **date_time slot**.

Intents and Slots names are usually task specific and defined as labels in the training data. The included intent/slot model was trained on the NLU-Evaluation-Data dataset - you can find the various intent and slot classes that it supports here.

They are common things that you might ask a **virtual assistant**:

```
$ examples/nlp.py --model distilbert_intent

Enter intent_slot query, or Q to quit:

> What is the weather in Santa Clara tomorrow morning?

{'intent': 'weather_query',
 'score': 0.7165476,
 'slots': [{'score': 0.6280392, 'slot': 'place_name', 'text': 'Santa'},
           {'score': 0.61760694, 'slot': 'place_name', 'text': 'Clara'},
           {'score': 0.5439486, 'slot': 'date', 'text': 'tomorrow'},
           {'score': 0.4520608, 'slot': 'date', 'text': 'morning'}]}

> Set an alarm for 730am

{'intent': 'alarm_set',
 'score': 0.5713072,
 'slots': [{'score': 0.40017933, 'slot': 'time', 'text': '730am'}]}

> Turn up the volume

{'intent': 'audio_volume_up', 'score': 0.33523008, 'slots': []}

> What is my schedule for tomorrow?
```

```
{'intent': 'calendar_query',
 'score': 0.37434494,
 'slots': [{'score': 0.5732627, 'slot': 'date', 'text': 'tomorrow'}]}

> Order a pepperoni pizza from domino's

{'intent': 'takeaway_order',
 'score': 0.50629586,
 'slots': [{'score': 0.27558547, 'slot': 'food_type', 'text': 'pepperoni'},
           {'score': 0.2778827, 'slot': 'food_type', 'text': 'pizza'},
           {'score': 0.21785143, 'slot': 'business_name', 'text': 'dominos'}]}

> Where's the closest Starbucks?

{'intent': 'recommendation_locations',
 'score': 0.5438984,
 'slots': [{'score': 0.1604197, 'slot': 'place_name', 'text': 'Starbucks'}]}
```

3.4.2 Text Classification

In this text classification example, we'll use the included sentiment analysis model that was trained on the Standford Sentiment Treebank (SST-2) dataset. It will label queries as either positive or negative, along with their probability:

```
$ examples/nlp.py --model distilbert_sentiment

Enter text_classification query, or Q to quit:

> today was warm, sunny and beautiful out

{'class': 1, 'label': '1', 'score': 0.9985898}

> today was cold and rainy and not very nice

{'class': 0, 'label': '0', 'score': 0.99136007}
```

(class 0 is negative sentiment and class 1 is positive sentiment)

3.4.3 Token Classification

Whereas text classification classifies entire queries, token classification classifies individual tokens (or words). In this example, we'll be performing **Named Entity Recognition** (NER), which is the task of detecting and classifying key information (entities) in text. For example, in a sentence: `Mary lives in Santa Clara and works at NVIDIA`, we should detect that `Mary` is a **person**, `Santa Clara` is a **location** and `NVIDIA` is a **company**.

The included token classification model for NER was trained on the Groningen Meaning Bank (GMB) and supports the following annotations in IOB format (short for inside, outside, beginning)

- `LOC = Geographical Entity`
- `ORG = Organization`
- `PER = Person`
- `GPE = Geopolitical Entity`
- `TIME = Time indicator`
- `MISC = Artifact, Event, or Natural Phenomenon`

```
$ examples/nlp.py --model distilbert_ner

Enter token_classification query, or Q to quit:
> Mary lives in Santa Clara and works at NVIDIA

Mary[B-PER 0.989] lives in Santa[B-LOC 0.998] Clara[I-LOC 0.996] and works at NVIDIA[B-ORG 0.967]

> Lisa's favorite place to climb in the summer is El Capitan in Yosemite National Park in
California, U.S.

Lisa's[B-PER 0.995] favorite place to climb in the summer[B-TIME 0.996] is El[B-PER 0.577]
Capitan[I-PER 0.483]
```

```
in Yosemite[B-LOC 0.987] National[I-LOC 0.988] Park[I-LOC 0.98] in California[B-LOC 0.998], U.S[B-
LOC 0.997].
```

3.4.4 Question/Answering

Question/Answering (QA) works by supplying a context paragraph which the model then queries the best answer from. The **nlp_qa.py** example allows you to select from several built-in context paragraphs (or supply your own) and to ask questions about these topics.

The **QA model** is flexible and doesn't need re-trained on different topics, as it was trained on the SQuAD question/answering dataset which allows it to extract answers from a variety of contexts. It essentially learns to identify the information most relevant to your query from the context passage, as opposed to learning the content itself.

```
$ examples/nlp_qa.py

Context:
The Amazon rainforest is a moist broadleaf forest that covers most of the Amazon basin of South
America.
This basin encompasses 7,000,000 square kilometres (2,700,000 sq mi), of which 5,500,000 square
kilometres
(2,100,000 sq mi) are covered by the rainforest. The majority of the forest is contained within
Brazil,
with 60% of the rainforest, followed by Peru with 13%, and Colombia with 10%.

Enter a question, C to change context, P to print context, or Q to quit:

> How big is the Amazon?

Answer: 7,000,000 square kilometres
Score:  0.24993503093719482

> which country has the most?

Answer: Brazil
Score:  0.5964332222938538

To change the topic or create one of your own, enter C:
Enter a question, C to change context, P to print context, or Q to quit:
> C

Select from one of the following topics, or enter your own context paragraph:
   1. Amazon
   2. Geology
   3. Moon Landing
   4. Pi
   5. Super Bowl 55
> 3

Context:
The first manned Moon landing was Apollo 11 on July, 20 1969. The first human to step on the Moon
was
astronaut Neil Armstrong followed second by Buzz Aldrin. They landed in the Sea of Tranquility with
their
lunar module the Eagle. They were on the lunar surface for 2.25 hours and collected 50 pounds of
moon rocks.

Enter a question, C to change context, P to print context, or Q to quit:

> Who was the first man on the moon?

Answer: Neil Armstrong
Score:  0.39105066657066345
```

3.5 Text-to-Speech (TTS)

The **text-to-speech service** uses an ensemble of two models: **FastPitch** to generate MEL spectrograms from text, and HiFiGAN as the vocoder (female English voice). For information about using the TTS APIs, please refer to `jetson_voice/tts.py` and see `examples/tts.py`

The examples/tts.py app can output the audio to a speaker, wav file, or sequence of wav files. Run it with `--list-devices` to get a list of your audio devices.

```
$ examples/tts.py --output-device 11 --output-wav data/audio/tts_test

> The weather tomorrow is forecast to be warm and sunny with a high of 83 degrees.

Run 0 -- Time to first audio: 1.820s. Generated 5.36s of audio. RTFx=2.95.
Run 1 -- Time to first audio: 0.232s. Generated 5.36s of audio. RTFx=23.15.
Run 2 -- Time to first audio: 0.230s. Generated 5.36s of audio. RTFx=23.31.
Run 3 -- Time to first audio: 0.231s. Generated 5.36s of audio. RTFx=23.25.
Run 4 -- Time to first audio: 0.230s. Generated 5.36s of audio. RTFx=23.36.
Run 5 -- Time to first audio: 0.230s. Generated 5.36s of audio. RTFx=23.35.

Wrote audio to data/audio/tts_test/0.wav

Enter text, or Q to quit:
> Sally sells seashells by the seashore.

Run 0 -- Time to first audio: 0.316s. Generated 2.73s of audio. RTFx=8.63.
Run 1 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.61.
Run 2 -- Time to first audio: 0.127s. Generated 2.73s of audio. RTFx=21.51.
Run 3 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.68.
Run 4 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.68.
Run 5 -- Time to first audio: 0.126s. Generated 2.73s of audio. RTFx=21.61.

Wrote audio to data/audio/tts_test/1.wav
```

2.5.1.1 TTS Audio Samples

- **Weather forecast (wav)**
- **Sally sells seashells (wav)**

3.6 Tests

There is an automated test suite included that will verify all of the models are working properly. You can run it with the **`tests/run_tests.py`** script:

```
$ tests/run_tests.py

-----------------------------------------------------
 TEST SUMMARY
-----------------------------------------------------
test_asr.py (quartznet)                  PASSED
test_asr.py (quartznet_greedy)           PASSED
test_asr.py (matchboxnet)                PASSED
test_asr.py (vad_marblenet)              PASSED
test_nlp.py (distilbert_qa_128)          PASSED
test_nlp.py (distilbert_qa_384)          PASSED
test_nlp.py (distilbert_intent)          PASSED
test_nlp.py (distilbert_sentiment)       PASSED
test_nlp.py (distilbert_ner)             PASSED
test_tts.py (fastpitch_hifigan)          PASSED

passed 10 of 10 tests
saved logs to data/tests/logs/20210610_1512
```

The logs of the individual tests are printed to the screen and saved to a timestamped directory.

Table of Contents