PROJECT REPORT

OBJECT CLASSIFICATION ON RASPBERRYPI PI

DHARMSINH DESAI UNIVERSITY

FACULTY OF TECHNOLOGY

DEPARTMENT OF ELECTRONICS AND COMMUNICATION

PREPARED BY:	**GUIDED BY:**
Dharmesh Kanzariya (EC-028)	**YKM SIR**
Harshal Kaswala (EC-030)	

ACKNOWLEDGEMENT

We hereby, completing this project, would like to express our very great appreciation to Prof. Yogesh K. Meghrajani for providing required guidance and his valuable and constructive suggestions during our project work. His willingness to give his time so generously has been very much appreciated.

We would also like to thank our friends and batch mates for their timely help comment and suggestions. Their suggestions and comments had really given me new direction to think on project.

We also thank the head of department Prof. Dr. NIKHIL J. KOTHARI for giving us the opportunity to carry out a thesis of our choice.

Lastly, we would like to extend our gratitude to authors of the papers and information sources without which this project would not have been possible.

The Faculty of technology of Dharmsinh Desai University- Nadiad has provided us with the knowledge and assistance that constructed the foundation required in us to initiate and follow through a project such as this, and for that we are grateful to all the teachers, officers, and staff of the EC Department

ABSTRACT

In this project, we apply supervised learning techniques on state-of-the-art deep convolutional neural networks (CNN) to classify Human at run time. Neural Network comes at the cost of high computational complexity. We implement CNN in low power device like pi using tflite. We train a model for human classification Using transfer learning. We intend that our results benefit further work applying CNNs in surveillance and security-related tasks.

Table of Contents

1. INTRODUCTION

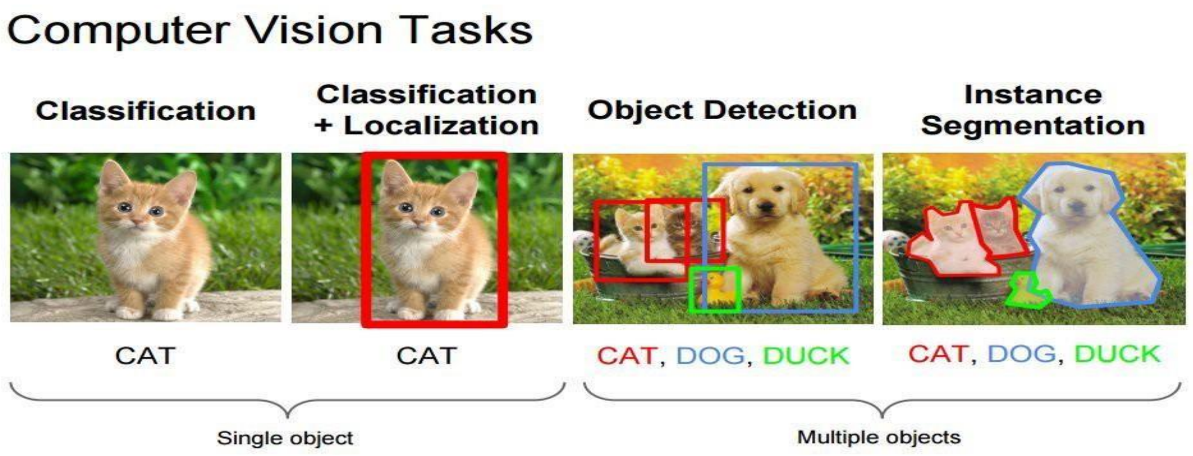
Machine learning techniques have been widely applied in a variety of areas such as pattern recognition, natural language processing and computational learning. With machine learning techniques, computers are endowed with the capability of acting without being explicitly programmed, constructing algorithms that can learn from data, and making data-driven decisions or predictions. With rapid development of computation techniques, a powerful framework has been provided by Airtificial neural networks(ANNs) with deep architectures for supervised learning. Generally speaking, the deep learning algorithm consists of a hierarchical architecture with many layers each of which constitutes a non-linear information processing unit. Deep neural networks (DNNs), which employ deep architectures in NNs, can represent functions with higher complexity if the numbers of layers and units in a single layer are increased. Given enough labeled training datasets and suitable models, deep learning approaches can help humans establish mapping functions for operation convenience.

Convolutional neural networks (CNN) have yielded state-of-the-art performance in image classification and other computer vision tasks include character recognition, classification of medical scans and object detection for autonomous vehicles. Due to these developments, automated systems can do the work that a human would typically do, with the effect of potentially reducing human error in some cases, or significantly decreasing the number of required hours of work. There is, however, one area in which computer vision is lacking: object and activity detection in surveillance video. Monitoring surveillance video is a task that would greatly benefit from computer vision; it is a physically taxing job where long hours can lead to errors, and it is costly to employ a person to continuously oversee security footage. It would be much more efficient to utilize a system that could alert a human if anomalous behavior or a specific object was detected.

The difficulty of this application is introduced by various qualities of surveillance footage. The camera angles are usually different in each scenario, the video resolution depends entirely on the camera model and can vary greatly, and the background of the scenes and the activities that are considered normal can be wildly different. Furthermore, an extremely low false negative rate is desirable so that the system does not pose security threats, and the false positive rate must be low enough for the system to add value.

We propose a solution that uses CNNs to help make surveillance analysis easier and more efficient. The particular problem we are addressing is one of being able to classify scenes captured in video frames from surveillance footage of a particular location. For this project, we focus on identifying whether an image has a Human in it on not.

ConvNets have been applied with great success to the detection, segmentation and recognition of objects and regions in images. These were all tasks in which labelled data was relatively abundant, such as traffic sign recognition, the segmentation of biological images particularly for the detection of faces, text, pedestrians and human bodies in natural images. A major recent practical success of ConvNets is face recognition.



1.2 FUNDAMENTAL THEORY

1.2.1 Supervised Learning

The most common form of machine learning, deep or not, is supervised learning. Imagine that we want to build a system that can classify images as containing, say, a house, a car, a person or a pet. We first collect a large data set of images of houses, cars, people and pets, each labelled with its category. During training, the machine is shown an image and produces an output in the form of a vector of scores, one for each category. We want the desired category to have the highest score of all categories, but this is unlikely to happen before training. We compute an objective function that measures the error between the output scores and the desired pattern of scores. The machine then modifies its internal adjustable parameters to reduce this error. These adjustable parameters, often called weights, are real numbers that define the input–output function of the machine. In a typical deep-learning system, there may be hundreds of millions of these adjustable weights, and hundreds of millions of labelled examples with which to train the machine.

To properly adjust the weight vector, the learning algorithm computes a gradient vector that, for each weight, indicates by what amount the error would increase or decrease if the weight were increased by a tiny amount. The weight vector is then adjusted in the opposite direction to the gradient vector.

The objective function, averaged over all the training examples, can be seen as a kind of hilly landscape in the high-dimensional space of weight values. The negative gradient vector indicates the direction of steepest descent in this landscape, taking it closer to a minimum, where the output error is low on average. In practice, most practitioners use a procedure called gradient descent. This consists of showing the input vector for a few examples, computing the outputs and the errors, computing the average gradient for those examples, and adjusting the weights accordingly. The process is repeated for many small sets of examples from the training set until the average of the objective function stops decreasing. This simple procedure usually finds a good set of weights. After training, the performance of the system is measured on a different set of examples called a test set. This serves to test the generalization ability of the machine — its ability to produce sensible answers on new inputs that it has never seen during training.

Many of the current practical applications of machine learning use linear classifiers on top of hand-engineered features. A two-class linear classifier computes a weighted sum of the feature vector components. If the weighted sum is above a threshold, the input is classified as belonging to a particular category.

Since the 1960s we have known that linear classifiers can only carve their input space into very simple regions, namely half-spaces separated by a hyperplane. But problems such as image and speech recognition require the input–output function to be insensitive to irrelevant variations of the input, such as variations in position, orientation of an object, or variations in the pitch or accent of speech, while being very sensitive to particular minute variations. At the pixel level, images Human in different poses and in different environments may be very different from each other, whereas images of a man and women in the same position and on similar backgrounds may be very similar to each other. To make classifiers more powerful, if good features can be learned automatically using a general-purpose learning procedure. This is the key advantage of deep learning.

A deep-learning architecture is a multilayer stack of simple modules, all (or most) of which are subject to learning, and many of which compute non-linear input–output mappings. Each module in the stack transforms its input to increase both the selectivity and the invariance of the representation. With multiple non-linear layers, say a depth of 5 to 20, a system can implement extremely intricate functions of its inputs that are simultaneously sensitive to small detail distinguishing human from other objects and insensitive to large irrelevant variations such as the background, pose, lighting and surrounding objects.

1.2.2 Convolutional Neural Networks

ConvNets are designed to process data that come in the form of multiple arrays, for example a color image composed of three 2D arrays containing pixel intensities in the three color channels. Many data modalities are in the form of multiple arrays: 1D for signals and sequences, including language; 2D for images or audio spectrograms; and 3D for video.

The architecture of a typical ConvNet is structured as a series of stages. The first few stages are composed of two types of layers: convolutional layers and pooling layers. Units in a convolutional layer are organized in feature maps, within which each unit is connected to local patches in the feature maps of the previous layer through a set of weights called a filter. The result of this local weighted sum is then passed through a non-linearity such as a ReLU.

Mathematically, the filter convolve with image is a discrete convolution, hence the name. Although the role of the convolutional layer is to detect local conjunctions of features from the previous layer, the role of the pooling layer is to merge semantically similar features into one. Two or three stages of convolution, non-linearity and pooling are stacked, followed by more convolutional and fully-connected layers. Backpropagating gradients through a ConvNet is as simple as through a regular deep network, allowing all the weights in all the filter banks to be trained.

2. DESIGN APPROCH

2.1 Multilayer Neural Networks And Backpropagation

a. A multilayer neural network can distort the input space to make the classes of data linearly separable. the networks used for object recognition or natural language processing contain tens or hundreds of thousands of neurons.

b. The chain rule of derivatives tells us how two small effects (that of a small change of x on y, and that of y on z) are composed. A small change Δx in x gets transformed first into a small change Δy in y by getting multiplied by ∂y/∂x (partial derivative). Similarly, the change Δy creates a change Δz in z. Substituting one equation into the other gives the chain rule of derivatives — how Δx gets turned into Δz through multiplication by the product of ∂y/∂x and ∂z/∂x.

c. The equations used for computing the forward pass in a neural net with hidden layers and output layer, each constituting a module through which one can backpropagate gradients. At each layer, we first compute the total input z to each unit, which is a weighted sum of the outputs of the units in the layer below. Then a non-linear function f(.) is applied to z to get the output of the unit. The non-linear functions used in neural networks include the rectified linear unit (ReLU) f(z) = max(0 , z), commonly used in recent years, as well as the more conventional sigmoid and hyperbolic tangent.

d. The equations used for computing the backward pass. At each hidden layer we compute the error derivative with respect to the output of each unit, which is a weighted sum of the error derivatives with respect to the total inputs to the units in the layer above. We then convert the error derivative with respect to the output into the error derivative with respect to the input by multiplying it by the gradient of f(z). At the output layer, the error derivative with respect to the output of a unit is computed by differentiating the cost function.

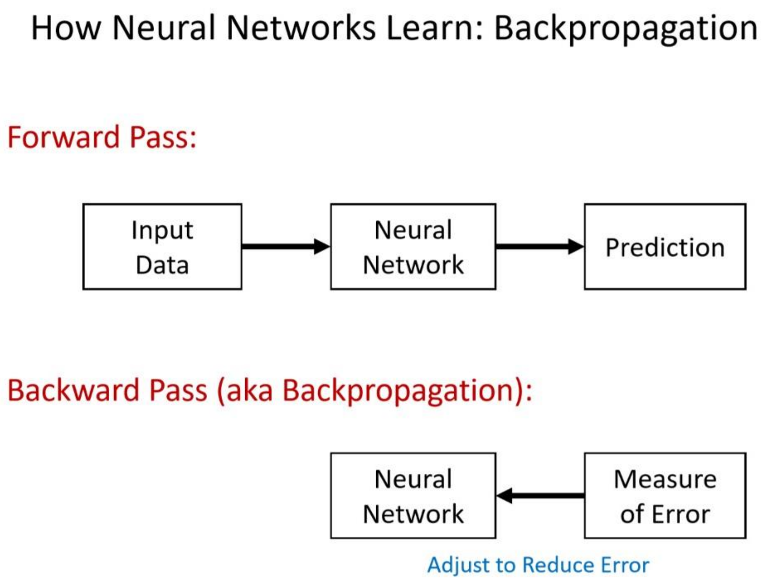
2.2 Backpropagation (To Train Multilayer Architectures)

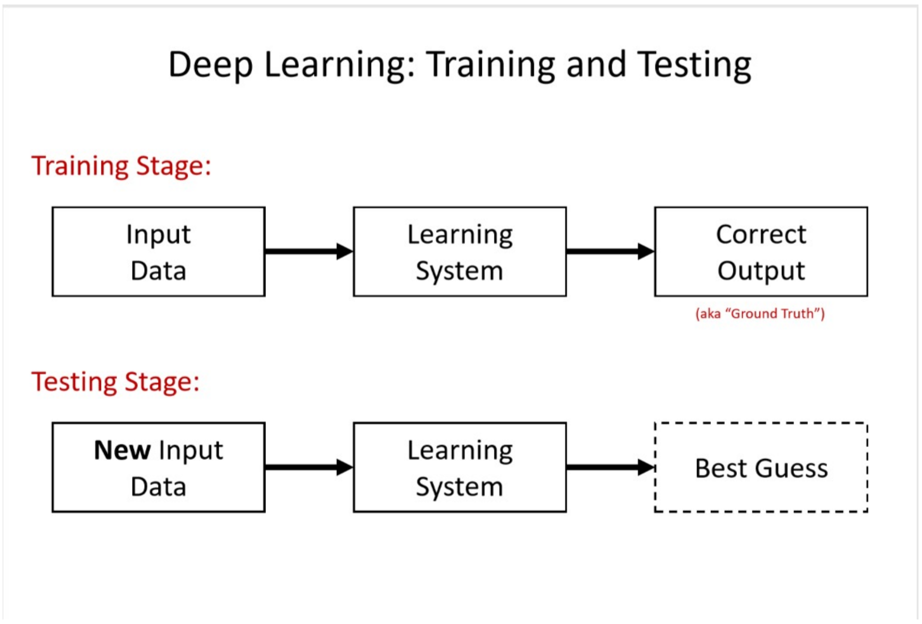
Multilayer architectures can be trained by gradient descent. As long as the modules are relatively smooth functions of their inputs and of their internal weights, one can compute gradients using the backpropagation procedure. The backpropagation procedure to compute the gradient of an objective function with respect to the weights of a multilayer stack of modules is nothing more than a practical application of the chain rule for derivatives. The key insight is that the derivative (or gradient) of the objective with respect to the input of a module can be computed by working backwards from the gradient with respect to the output of that module. The backpropagation equation can be applied repeatedly to propagate gradients through all modules, starting from the output all the way to the input. Once these gradients have been computed, it is straightforward to compute the gradients with respect to the weights of each module.

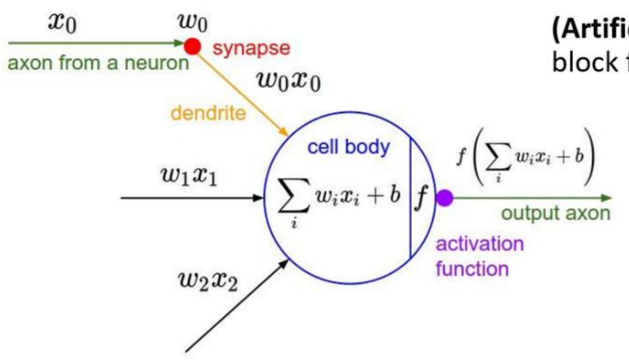
Many applications of deep learning use feedforward neural network architectures, which learn to map a fixed-size input (for example, an image) to a fixed-size output (for example, a probability for each of several categories). To go from one layer to the next, a set of units compute a weighted sum of their inputs from the previous layer and pass the result through a non-linear function. At present, the most popular non-linear function is the rectified linear unit (ReLU), which is simply the half-wave rectifier f(z) = max(z, 0). Units that are not in the input or output layer are conventionally called hidden units. The hidden layers can be seen as distorting the input in a non-linear way so that categories become linearly separable by the last layer.

In particular, it was commonly thought that simple gradient descent would get trapped in poor local minima weight configurations for which no small change would reduce the average error. In practice, poor local minima are rarely a problem with large networks. Regardless of the initial conditions, the system nearly always reaches solutions of very similar quality. Recent theoretical and empirical results strongly suggest that local minima are not a serious issue in general.

There was, however, one particular type of deep feedforward network that was much easier to train and generalized much better than networks with full connectivity between adjacent layers. This was the convolutional neural network (ConvNet). it has been widely adopted by the computer vision community.







(Artificial) Neuron: computational building block for the "neural network"

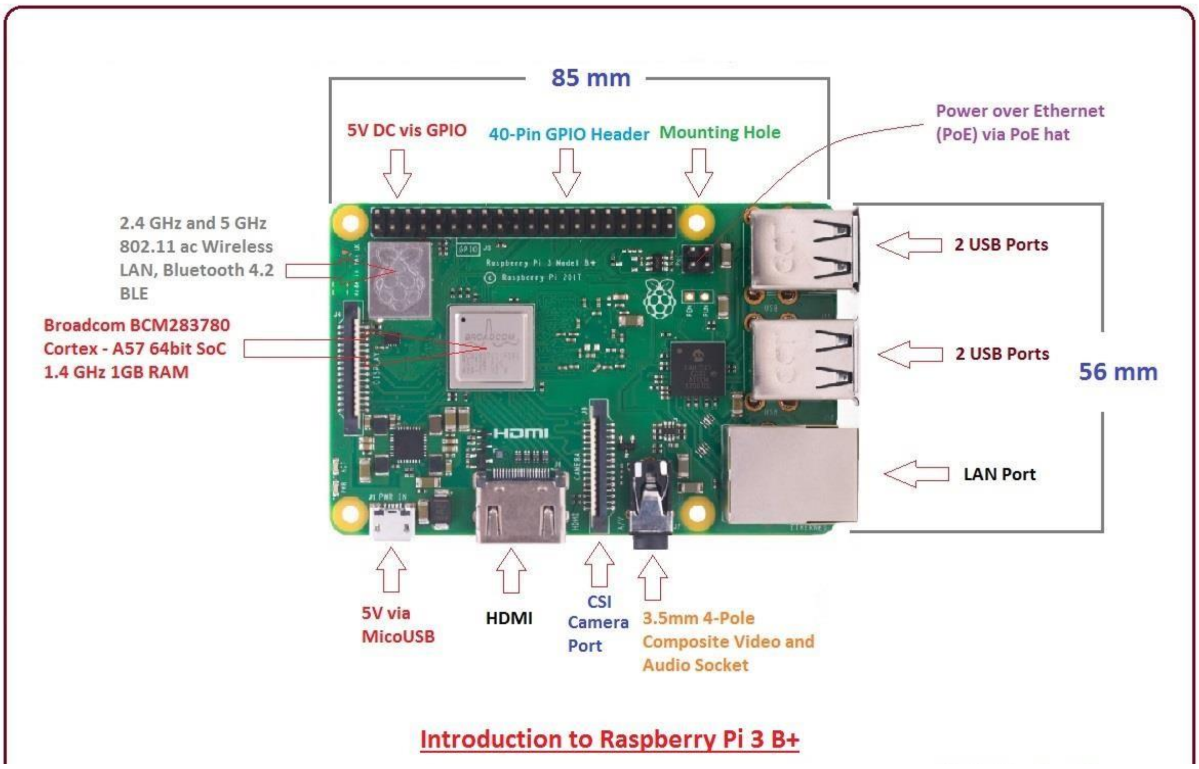
2.3 The Final Model

- We implement Digit recognition model using Deep neural network using Mnist Data Set.
- Implement Convolution and pooling layer from scratch.
- Implement cat vs dog model using CNN.
- Implement human Vs non-human model using transfer learning.
- Convert model into tflite model.
- Then tflite model run in to pi.

3. SYSTEM SPECIFICATIONS

3.1 COMPONENTS DESCRIPTION

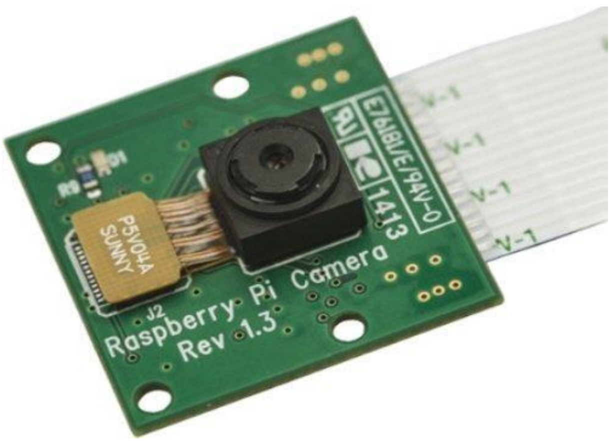
1.Raspberry Pi 3B+:


85 mm
5V DC vis GPIO
40-Pin GPIO Header
Mounting Hole
Power over Ethernet (PoE) via PoE hat
2.4 GHz and 5 GHz 802.11 ac Wireless LAN, Bluetooth 4.2 BLE
Broadcom BCM283780 Cortex - A57 64bit SoC 1.4 GHz 1GB RAM
2 USB Ports
2 USB Ports
56 mm
LAN Port
5V via MicoUSB
HDMI
CSI Camera Port
3.5mm 4-Pole Composite Video and Audio Socket
Introduction to Raspberry Pi 3 B+

The Raspberry Pi model 3 B+ Specification

1. **SOC**: Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC
2. **CPU**: 1.4GHz 64-bit quad-core ARM Cortex-A53 CPU
3. **RAM**: 1GB LPDDR2 SDRAM
4. **WIFI**: Dual-band 802.11ac wireless LAN (2.4GHz and 5GHz) and Bluetooth 4.2
5. **Ethernet**: Gigabit Ethernet over USB 2.0 (max 300 Mbps). Power-over-Ethernet support (with separate PoE HAT). Improved PXE network and USB mass-storage booting.
6. **Thermal management**: Yes
7. **Video**: Yes – VideoCore IV 3D. Full-size HDMI
8. **Audio**: Yes
9. **USB** 2.0: 4 ports
10. **GPIO**: 40-pin
11. **Power**: 5V/2.5A DC power input
12. **Operating system support**: Linux and Unix

2. Camera Module:



1. 5 megapixel native resolution sensor-capable of 2592 x 1944 pixel static images
2. Supports 1080p30, 720p60 and 640x480p60/90 video
3. Camera is supported in the latest version of Raspbian, Raspberry Pi's preferred operating system

3.2 SET UP IN PI:

1. Virtual environment , tensorflow opencv installation:

For virtual environment installation first install pip command using terminal.pip is also useful for installing any setup.

Installing pip:

For install pip command write below command in terminal:

```
Command: python3 -m pip install --user --upgrade pip
```

Afterwards, you should have the newest pip installed in your user site:

```
Command: python3 -m pip --version
pip 9.0.1 from $HOME/.local/lib/python3.6/site-packages (python 3.6)
```

Installing virtualenv:

virtualenv is used to manage Python packages for different projects. Using virtualenv allows you to avoid installing Python packages globally which could break system tools or other projects. You can install virtualenv using pip.

Command: python3 -m pip install --user virtualenv

Creating a virtual environment:

Virtual environment is allow you to manage separate package installations for different projects. They essentially allow you to create a virtual isolated Python installation and install packages into that virtual installation. When you switch projects, you can simply create a new virtual environment and not have to worry about breaking the packages installed in the other environments. It is always recommended to use a virtual environment while developing Python applications.

venv (for Python 3) and virtualenv (for Python 2)

Command: python3 -m venv env1(environment name)

Activating a virtual environment:

Before you can start installing or using packages in your virtual environment you'll need to *activate* it. Activating a virtual environment will put the virtual environment-specific python and pip executables into your shell's PATH.

Command: source env1(environment name)/bin/activate

Leaving the virtual environment:

If you want to switch projects or otherwise leave your virtual environment, simply run:

Command: deactivate

If you want to re-enter the virtual environment just follow the same instructions above about activating a virtual environment. There's no need to re-create the virtual environment.

Installing tensorflow:

for installing tensorflow following command write:

Command: pip3 install tensorflow

Tensorflow also needs the libatlas package. Install it following command:

Command :Sudo apt-get install libatlas-base-dev

Installing OpenCV:

OpenCV is is use for display images. If you comfortable with matplotlib then you can use matplotlib, but I prefer to use OpenCV because it’s easier to work with and less error prone.

To get OpenCV working on the Raspberry Pi, there’s quite a few dependencies that need to be installed through apt-get.

Commands:

```
sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install qt4-dev-tools libatlas-base-dev
```

Now that we’ve got all those installed, we can install OpenCV.

Command: `sudo pip3 install opencv-python`

Converting Model To Tflite Model:

if we have low power device then we use tflite model. this model is compressed and complexity of computation is reduced. This type model use in raspberry pi.

The TensorFlow Lite converter is a tool available as a Python API that converts trained TensorFlow models into the TensorFlow Lite format. It can also introduce optimizations, Optimize your model using Quantization.

The following example shows a TensorFlow SavedModel being converted into the TensorFlow Lite format:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_model)
```

4. RESULT AND DISCUSSION

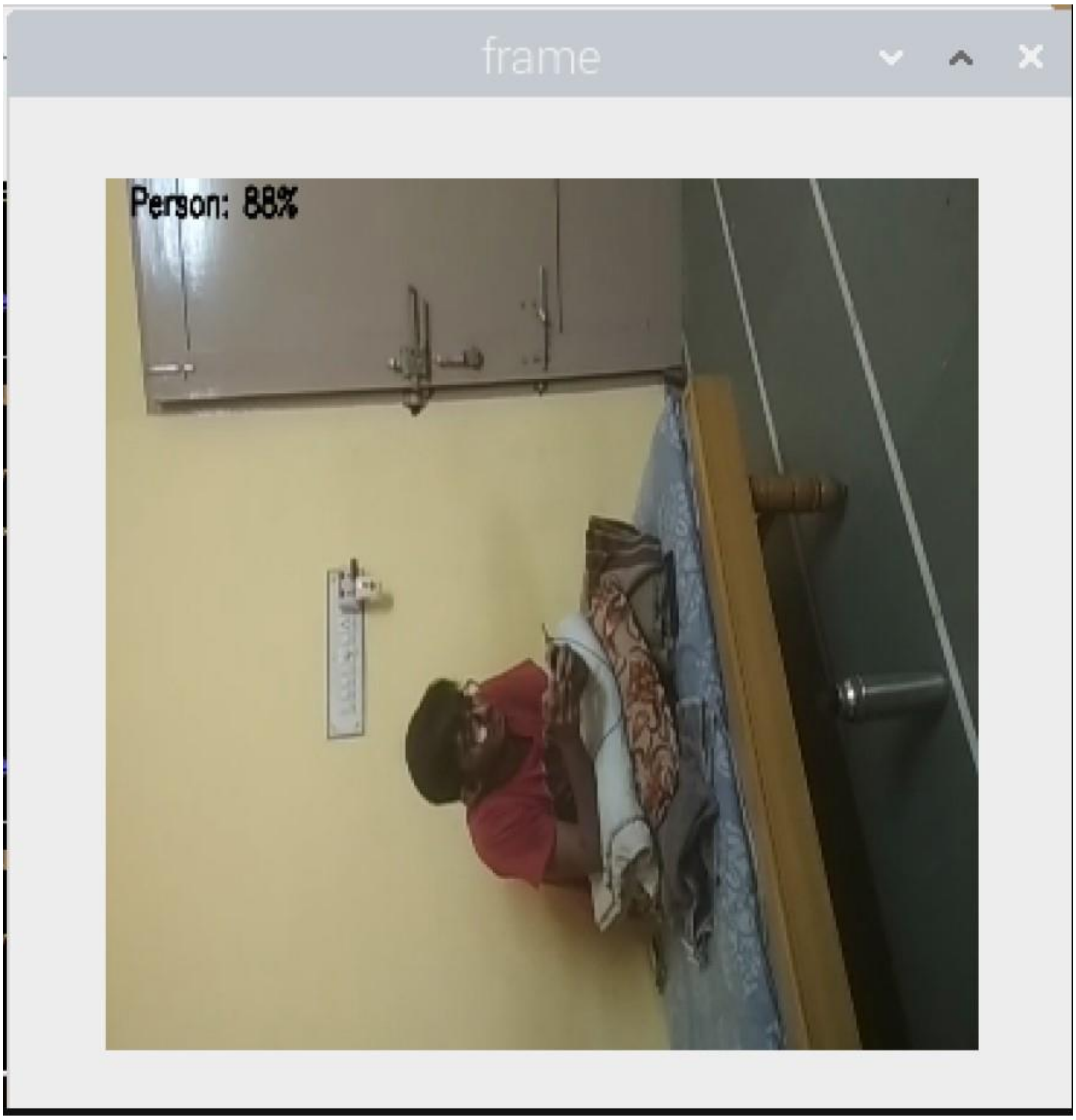


Fig 4.1 model run real time to detect human in Frame or Not

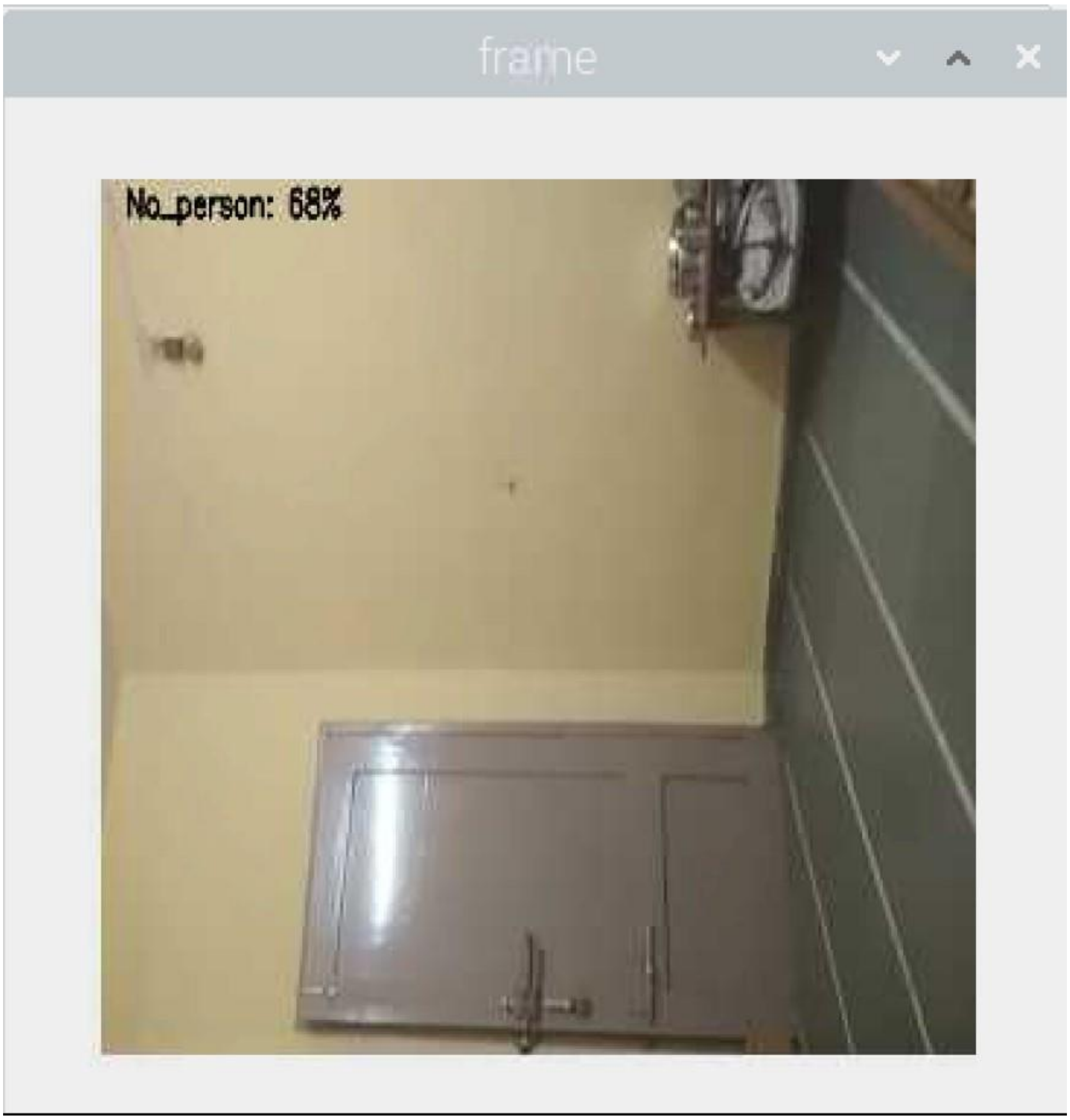
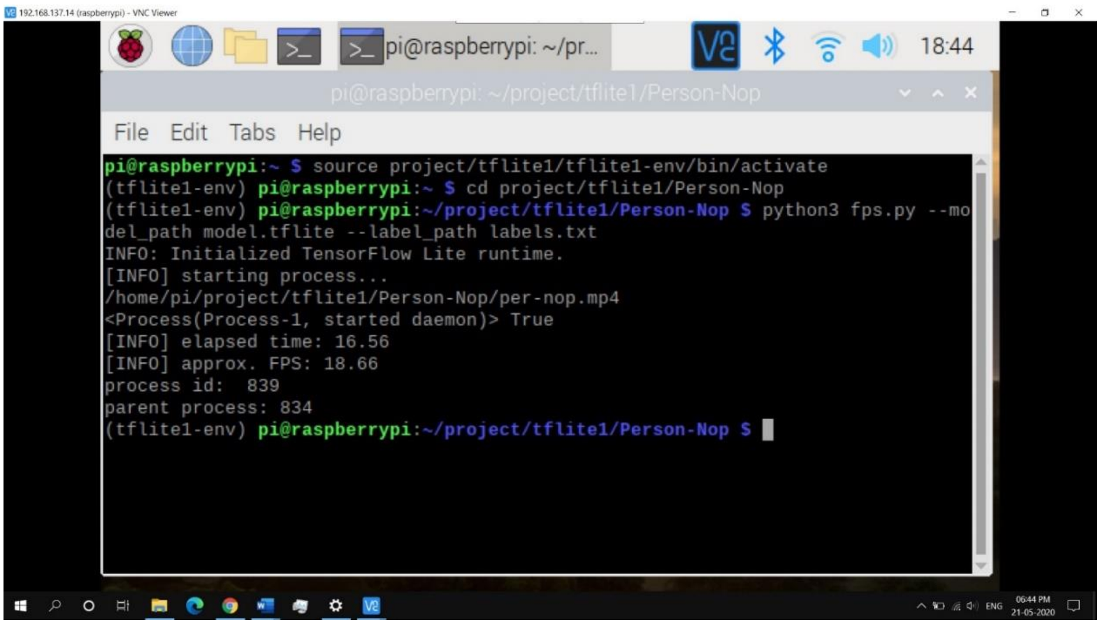


Fig 4.2 model tales not human with 68% accuracy

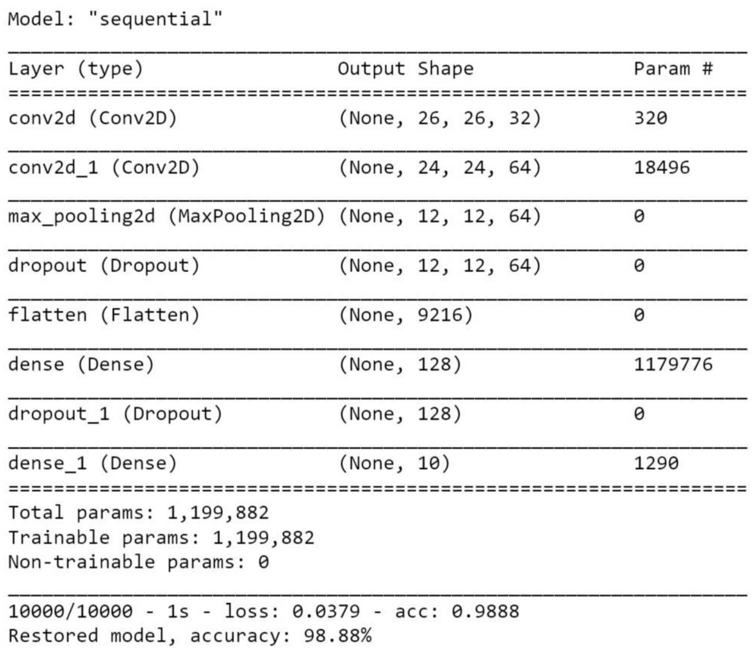

192.168.137.14 (raspberrypi) - VNC Viewer
pi@raspberrypi: ~/pr...
18:44
pi@raspberrypi: ~/project/tflite1/Person-Nop
File Edit Tabs Help
ENG
06:44 PM
21-05-2020

```
pi@raspberrypi:~ $ source project/tflite1/tflite1-env/bin/activate
(tflite1-env) pi@raspberrypi:~ $ cd project/tflite1/Person-Nop
(tflite1-env) pi@raspberrypi:~/project/tflite1/Person-Nop $ python3 fps.py --mo
del_path model.tflite --label_path labels.txt
INFO: Initialized TensorFlow Lite runtime.
[INFO] starting process...
/home/pi/project/tflite1/Person-Nop/per-nop.mp4
<Process(Process-1, started daemon)> True
[INFO] elapsed time: 16.56
[INFO] approx. FPS: 18.66
process id:  839
parent process: 834
(tflite1-env) pi@raspberrypi:~/project/tflite1/Person-Nop $ 
```

Fig 4.3 Compiled Model in Pi with fps and time.

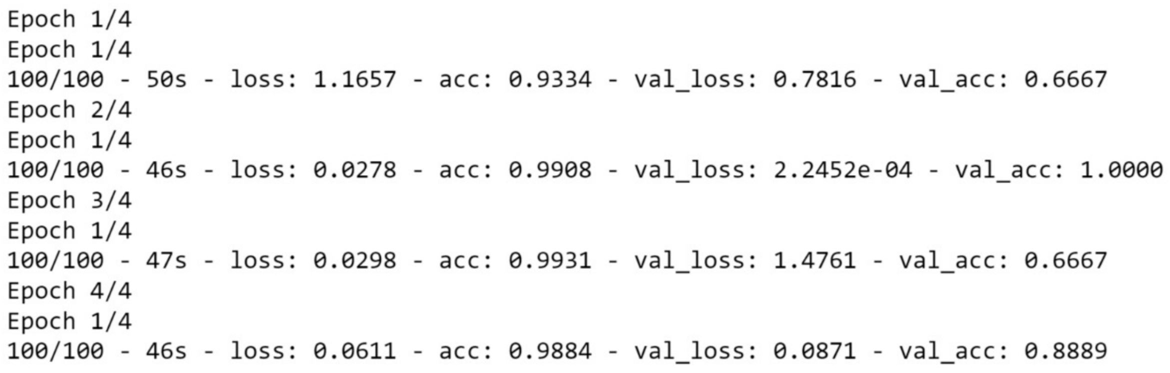
```
Epoch 10/15
Epoch 1/15
100/100 - 15s - loss: 0.6193 - acc: 0.6595 - val_loss: 0.6009 - val_acc: 0.6720
Epoch 11/15
Epoch 1/15
100/100 - 16s - loss: 0.6057 - acc: 0.6655 - val_loss: 0.5663 - val_acc: 0.7090
Epoch 12/15
Epoch 1/15
100/100 - 15s - loss: 0.5970 - acc: 0.6760 - val_loss: 0.6244 - val_acc: 0.6680
Epoch 13/15
Epoch 1/15
100/100 - 16s - loss: 0.6023 - acc: 0.6945 - val_loss: 0.5867 - val_acc: 0.6990
Epoch 14/15
Epoch 1/15
100/100 - 15s - loss: 0.5817 - acc: 0.6990 - val_loss: 0.6170 - val_acc: 0.6900
Epoch 15/15
Epoch 1/15
100/100 - 15s - loss: 0.5855 - acc: 0.6990 - val_loss: 0.5651 - val_acc: 0.6990
```

Fig 4.4 Cat – dog model gives accuracy 69 %



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320       
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496     
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0         
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 64)        0         
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0         
_________________________________________________________________
dense (Dense)                (None, 128)               1179776   
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290      
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
10000/10000 - 1s - loss: 0.0379 - acc: 0.9888
Restored model, accuracy: 98.88%
```

Fig 4.5 Models of MNIST Hand Written Digit Gives accuracy 98.88 %



```
Epoch 1/4
Epoch 1/4
100/100 - 50s - loss: 1.1657 - acc: 0.9334 - val_loss: 0.7816 - val_acc: 0.6667
Epoch 2/4
Epoch 1/4
100/100 - 46s - loss: 0.0278 - acc: 0.9908 - val_loss: 2.2452e-04 - val_acc: 1.0000
Epoch 3/4
Epoch 1/4
100/100 - 47s - loss: 0.0298 - acc: 0.9931 - val_loss: 1.4761 - val_acc: 0.6667
Epoch 4/4
Epoch 1/4
100/100 - 46s - loss: 0.0611 - acc: 0.9884 - val_loss: 0.0871 - val_acc: 0.8889
```

Fig 4.6 Model of Human vs. Nonhuman Gives 88.89% accuracy on validation set

5. FUTURE WORK

With the current iteration of the project, we can use it to flag the parts of the video with possible instances of a humans. To be more useful, it can be extended to tracking the man and also abandoned luggage through time to detect the persons responsible for it. A further extension would be to track the person through time to compile a list of their activities throughout the dataset. These could be achieved with a mix of segmentation and temporal models. Feeding the output of this model to a Faster R-CNN or YOLO based network could be extended to object localization that can be used to build scene graphs, count and description of the scene and human or abandoned luggage. Another interesting extension would be to use more datasets in varied locations and environments to develop a more generalized model which can in turn be used as a blanket model for further transfer learning.

6. CONCLUSION

In this project, we implemented a method to detect Human, cats-dogs and digits on a certain set of circumstances and environments. Through this, we have shown that it is possible to fine tune the models to particular environments and achieve very high accuracies. The models are not, necessarily, well-generalizable to other environments.

We use tflite to run computationally expensive model in to low power embedded device. This might be another factor which causes the model to be environment-specific. However, this might not necessarily be a large drawback.

In cases such as surveillance videos, most applications are specific to locations and types of events. Therefore, a specialized model would be quite useful especially, due to the ease of training and adaptation. As we only used a small number of data to create this model, we have shown that tuning the model to a different location would be are relatively quick and low resource intensive task.

Even with a relatively small dataset, we were able to achieve a good model by using data augmentation techniques which helped with two important aspects – generating more data for the model to train on and creating a better distribution of the input instances.

7. Code

https://github.com/Dharmesh1906/Embbeded-Deep-Learning

Final Code:

```
from tensorflow.lite.python.interpreter import Interpreter
import numpy as np
import argparse
import os
import cv2
from imutils.video import FPS
from multiprocessing import Process
from multiprocessing import Queue
import time

def classify_frame(net, inputQueue, outputQueue):
    # keep looping
    while True:
        # check to see if there is a frame in our input queue
        if not inputQueue.empty():
            # grab the frame from the input queue, resize it, and
            # construct a blob from it
            # Get input and output tensors.
            input_details = net.get_input_details()
            output_details = net.get_output_details()
            frame = inputQueue.get()
            video_resized = cv2.resize(frame, (224, 224))
            img = np.array(video_resized)
            input_data = np.expand_dims(img, axis=0)

            # Point the data to be used for testing and run the interpreter
            interpreter.set_tensor(input_details[0]['index'], input_data)
            interpreter.invoke()

            # Obtain results and map them to the classes
            predictions = interpreter.get_tensor(output_details[0]['index'])[0]

            # Get indices of the top k results
            top_k_indices = np.argsort(predictions)[::-1][:top_k_results]
            #print(top_k_indices)
            #print("##")
            #print(predictions)
            outputQueue.put(predictions)

parser = argparse.ArgumentParser(description='Image Classification')
parser.add_argument('--model_path', type=str, help='Specify the model path', required=True)
parser.add_argument('--label_path', type=str, help='Specify the label map', required=True)
parser.add_argument('--top_k', type=int, help='How many top results', default=2)
```

```
args = parser.parse_args()

model_path = args.model_path
label_path = args.label_path
top_k_results = args.top_k

with open(label_path, 'r') as f:
    labels = list(map(str.strip, f.readlines()))

# Load TFLite model and allocate tensors
interpreter = Interpreter(model_path=model_path)
interpreter.allocate_tensors()

# Get input and output tensors.
#input_details = interpreter.get_input_details()
#output_details = interpreter.get_output_details()

inputQueue = Queue(maxsize=1)
outputQueue = Queue(maxsize=1)
detections = None

# construct a child process *indepedent* from our main process of
# execution
print("[INFO] starting process...")
p = Process(target=classify_frame, args=(interpreter, inputQueue,
    outputQueue,))
p.daemon = True
p.start()

#IMAGE_NAME = 'miperson.mp4'
# IMAGE_NAME = 'person.mp4'
# IMAGE_NAME = 'per-nop.mp4'
#IMAGE_NAME = 'nop.mp4'
#IMAGE_NAME = 'dnop.mp4'
#IMAGE_NAME = 'dperson.mp4'

CWD_PATH = os.getcwd()
PATH_TO_IMAGE = os.path.join(CWD_PATH,IMAGE_NAME)
video = cv2.VideoCapture(0)
#time.sleep(2.0)
fps = FPS().start()
print(PATH_TO_IMAGE)
while(video.isOpened()):
    ret,frame = video.read()
    #frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    if ret == True:
        if inputQueue.empty():
            inputQueue.put(frame)
        # if the output queue *is not* empty, grab the detections
        if not outputQueue.empty():
            detections = outputQueue.get()

        predicted_label = np.argmax(detections)
        #print(detections)
        #print(predicted_label)
        #for i in range(top_k_results):
          #   print(labels[top_k_indices[i]], predictions[top_k_indices[i]] / 255.0)
        if detections is not None:
            label = '%s: %d%%' % (labels[predicted_label],
int((detections[predicted_label]/255.0)*100))
            cv2.putText(frame,label,(20,20),cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)
            frame=cv2.resize(frame,(500,500))
            cv2.imshow('frame',frame)
            if cv2.waitKey(1) == ord('q'):
                break
            # update the FPS counter
            fps.update()
    else:
        cv2.destroyAllWindows()
        break

# stop the timer and display FPS information
fps.stop()
time.sleep(1)
print(p,p.is_alive())
print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
print("process id: ",os.getpid())
print("parent process:",os.getppid())
#p.terminate()
p.kill()
#p.join()
video.release()
```

8. REFERENCE

- http://cs231n.stanford.edu/
- http://cs229.stanford.edu/
- Deep Learning
- Book by Aaron Courville, Ian Goodfellow, and Yoshua Bengio
- https://www.instructables.com/id/HOW-TO-INSTALL-RASPBIAN-OS-IN-YOUR-RASPBERRY-PI/
- https://www.tensorflow.org/lite/guide/inference
- https://docs.python.org/2/library/multiprocessing.html
- https://docs.opencv.org/2.4/modules/refman.html