Model Compression for On-Device Deep Learning

Name: WONG Tsz Ho

Supervisor Name: Zhang J

Content

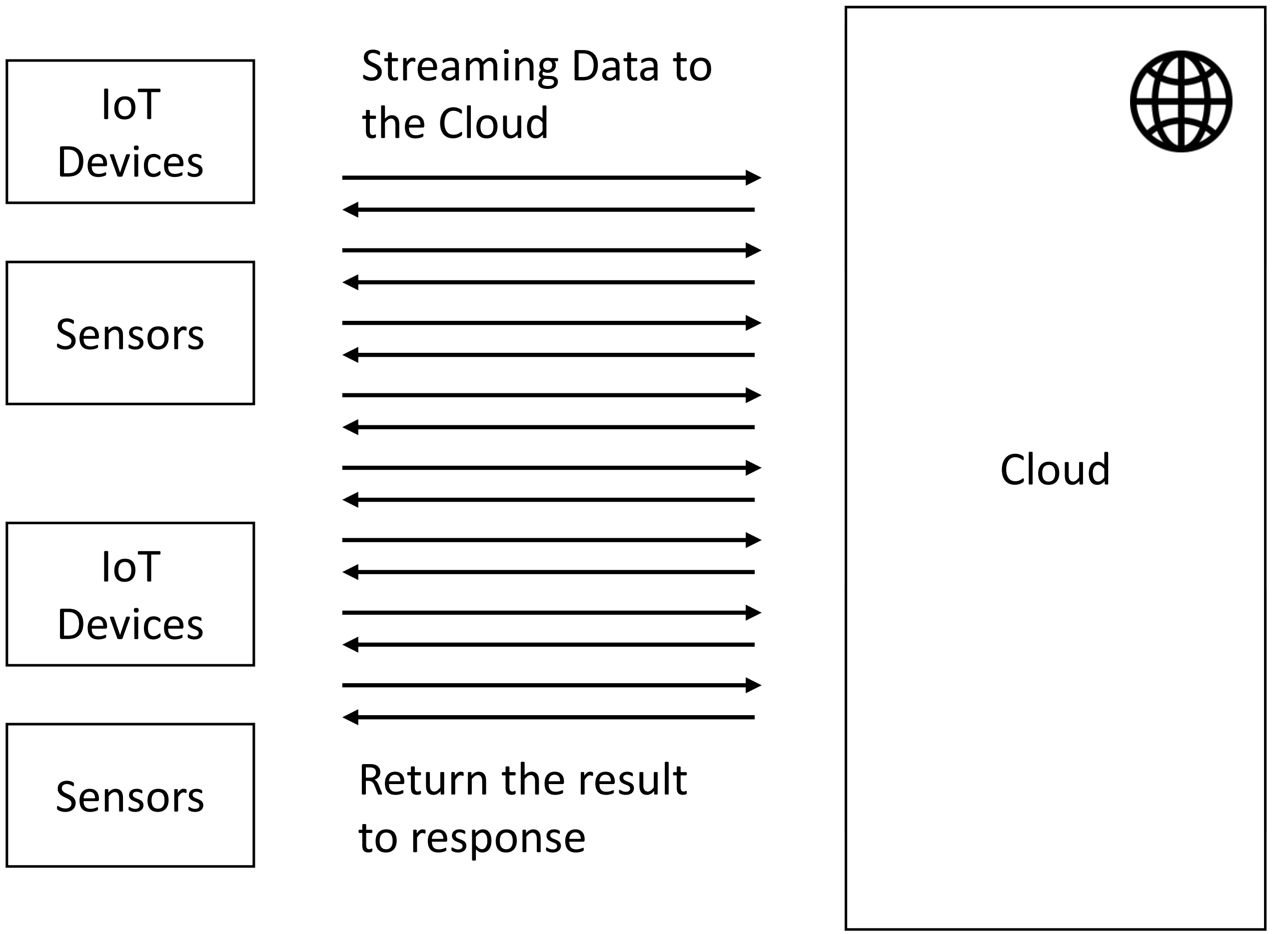
- Background
- Introduction
- Methodology
- Result
- Discussion and Recommendation

Background

Why On-Device Deep Learning?

Cloud AI

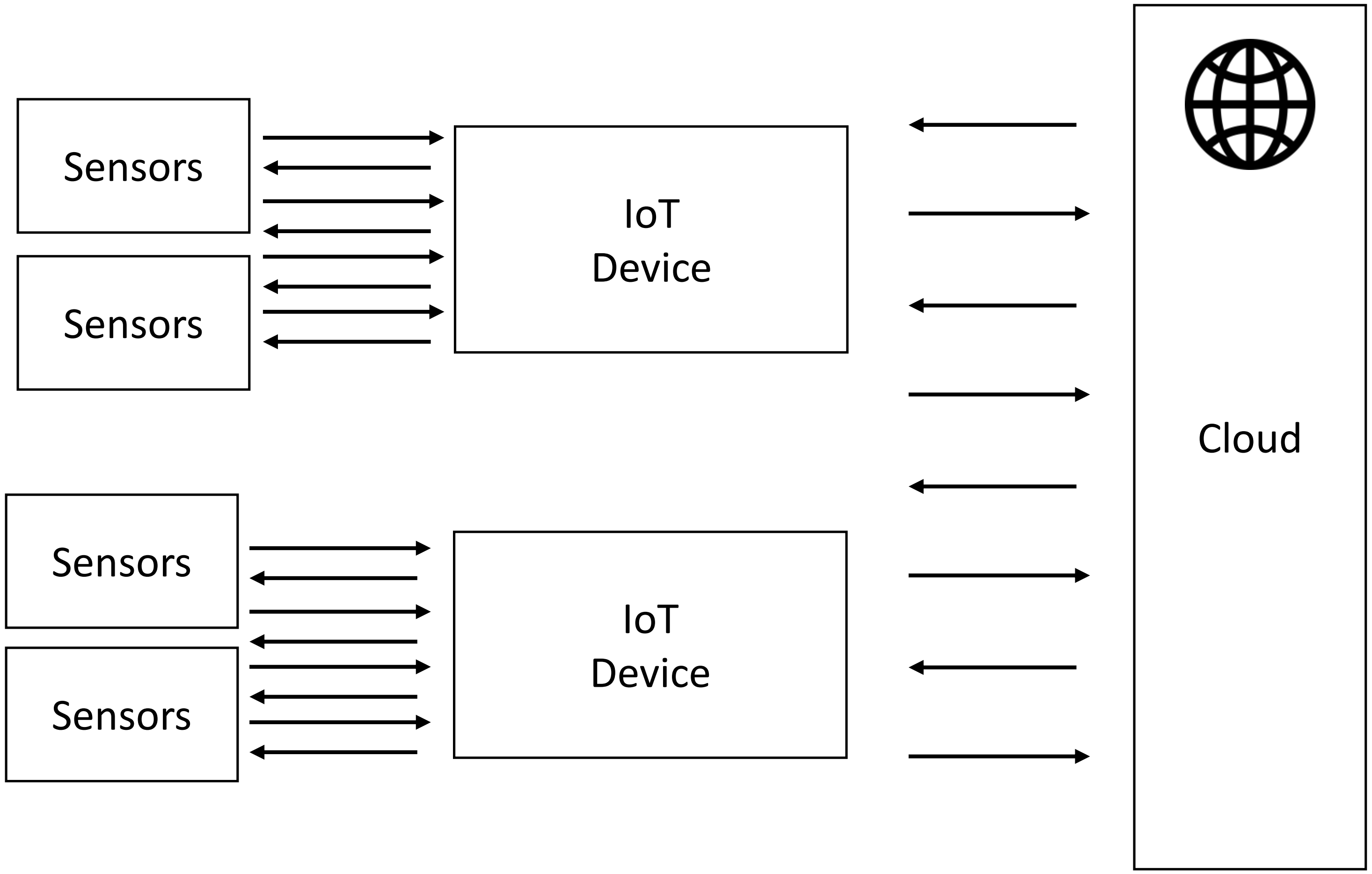
- Cost Saving
- Reliability
- Manageability



- Downtime
- Security
- Bandwidth
- Privacy
- Network

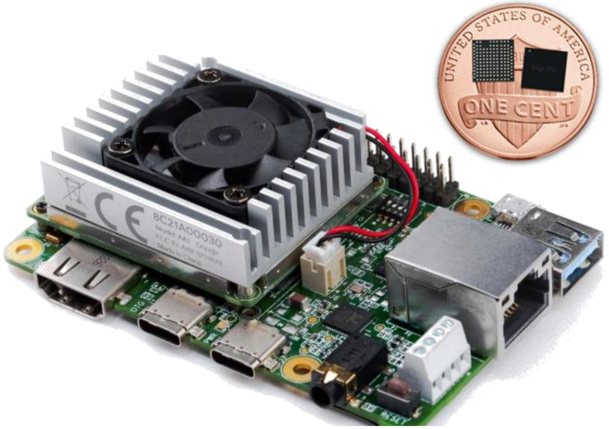
On-Device AI

- Latency
- Bandwidth
- Privacy



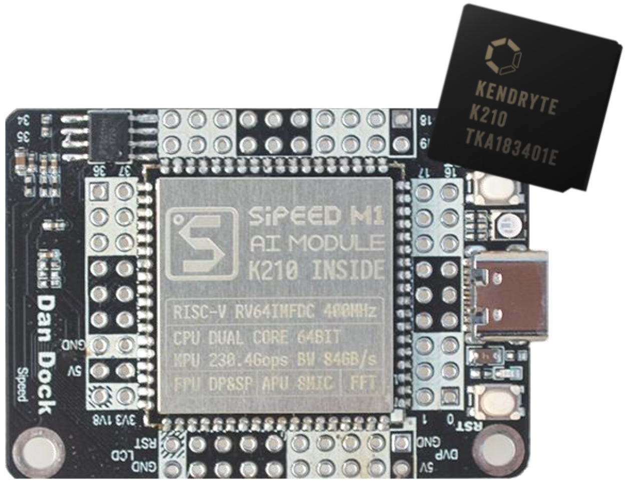
- More hardware

Possible Hardware solution



Google

TPU - ASIC



嘉楠科技
Canaan Inc.

FPU – RISC-V

NVIDIA

GPU

Nvidia Jetson Nano

GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI and DP
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I2C, I2S, SPI, UART
Mechanical	100 mm x 80 mm x 29 mm

Jetson Nano delivers 472 GFLOPs for running modern AI algorithms fast

Network API abstraction

Apple
Core ML

Android
NNAPI

ARM NN

CMSIS-NN

High level Neural Network Framework

- Easy to write your own layer types and run on GPU

- TensorFlow Lite for Mobile & IoT

- TensorFlow.js for JavaScript

- Multi-Language support

Introduction

Project

NVIDIA

PyTorch

Technology

Project Goal

1. **Design and Performing On-device deep learning** with Nvidia Jetson Nano **using custom target data** on classification and regression problem
2. Perform and develop program to **perform model compression with Machine Learning**
3. **Design application** for on-device deep learning with “Deep Learning aided Visual System”

Project Application

"Deep Learning aided Visual System"

Figure C1b

A diagram illustrates the application

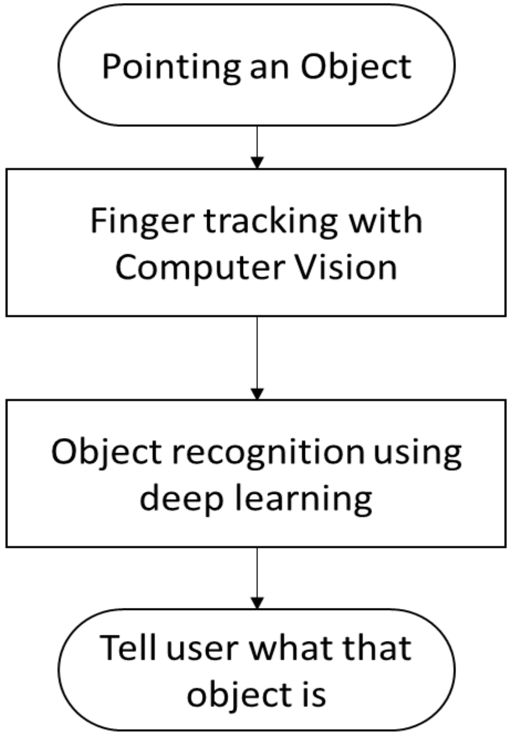
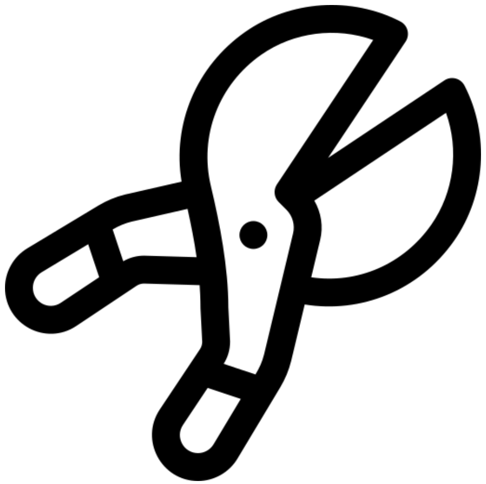


Figure C1a:

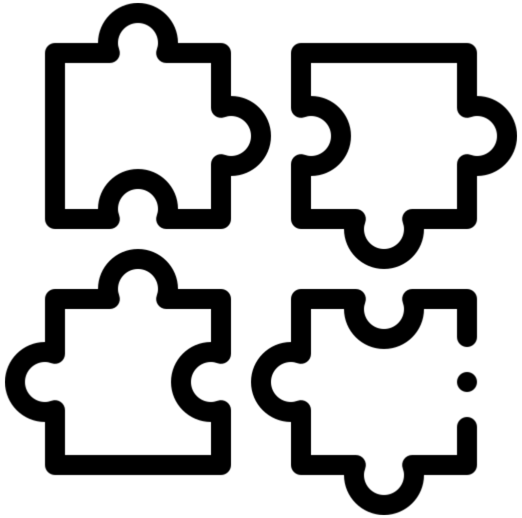
Block diagram of the system

Literature Review – Model Compression

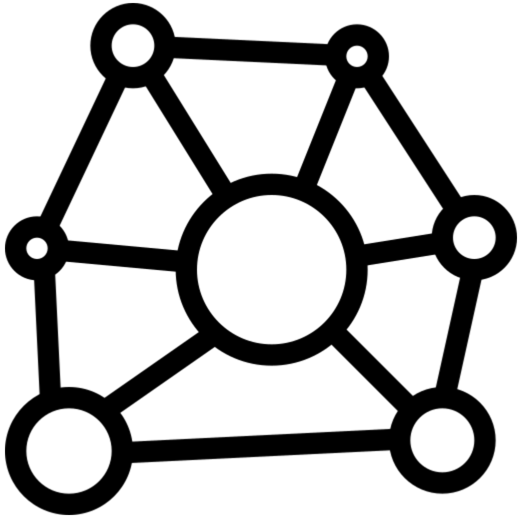
arXiv preprint arXiv:1710.09282



1. Pruning and Sharing



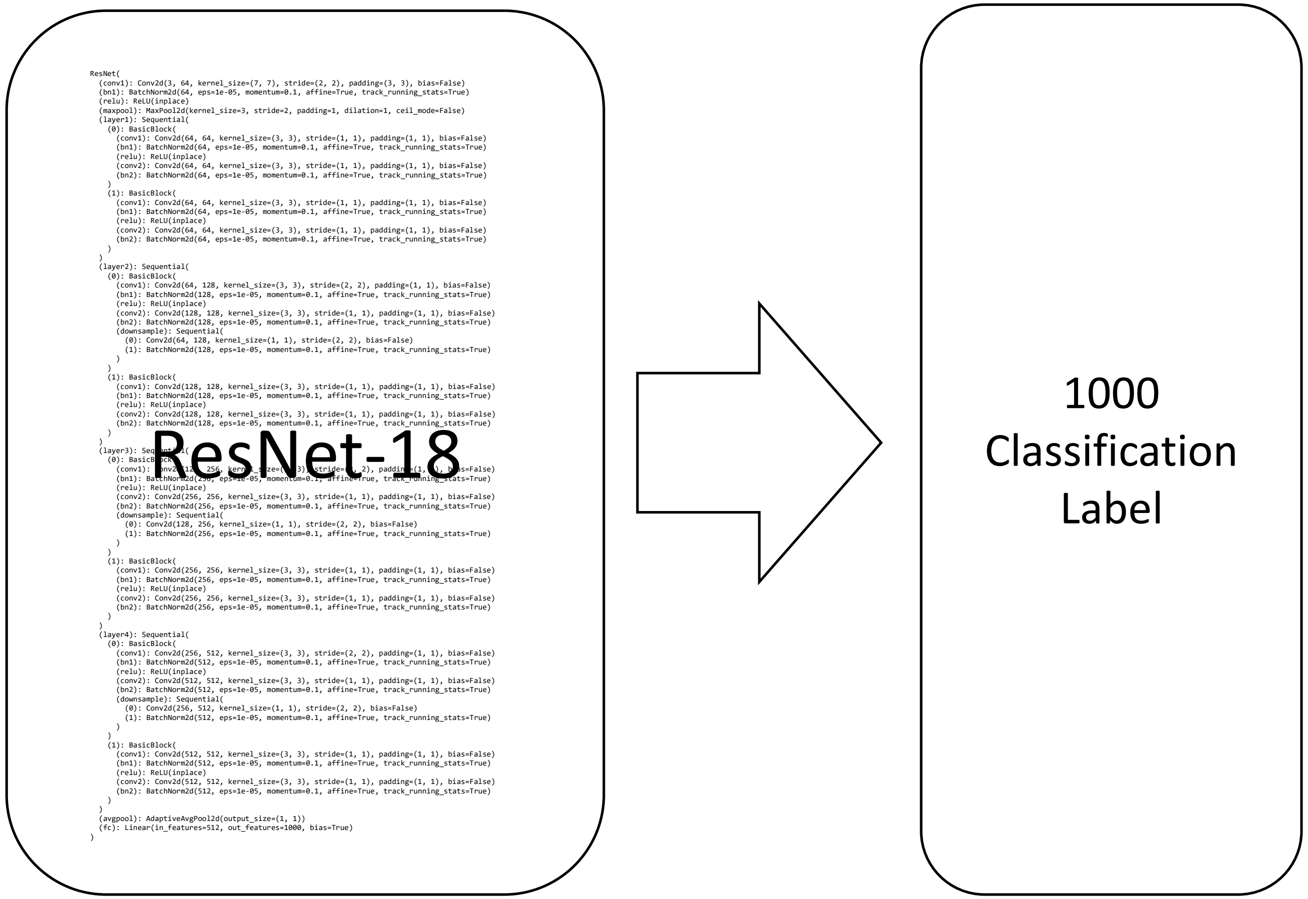
2. Quantization and binarization



3. Designing Structural Matrix.

Transfer learning

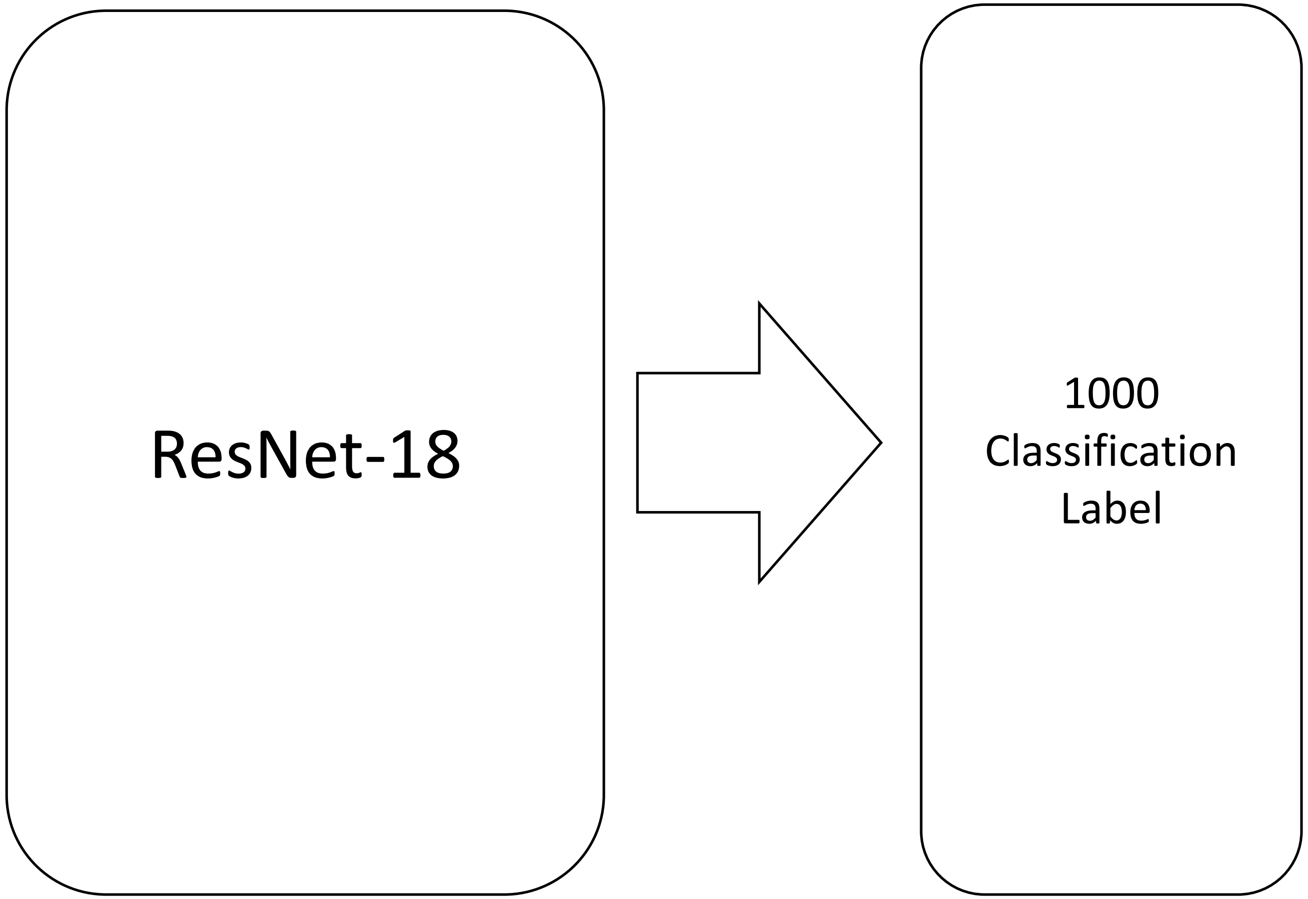
the Pretrained ResNet 18



```
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pa
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, t
    (relu): ReLU(inplace)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), pa
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, t
  )
 )
 (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
 (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

Transfer learning

the Pretrained ResNet 18



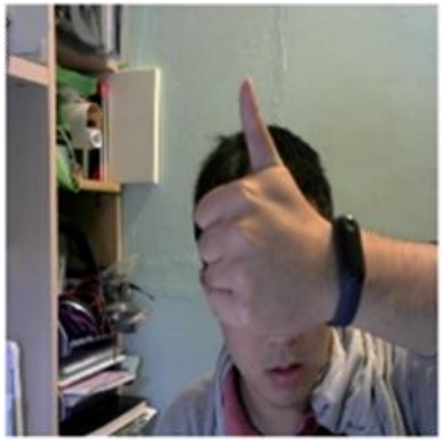
Transfer learning

the Pretrained ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

ResNet-18

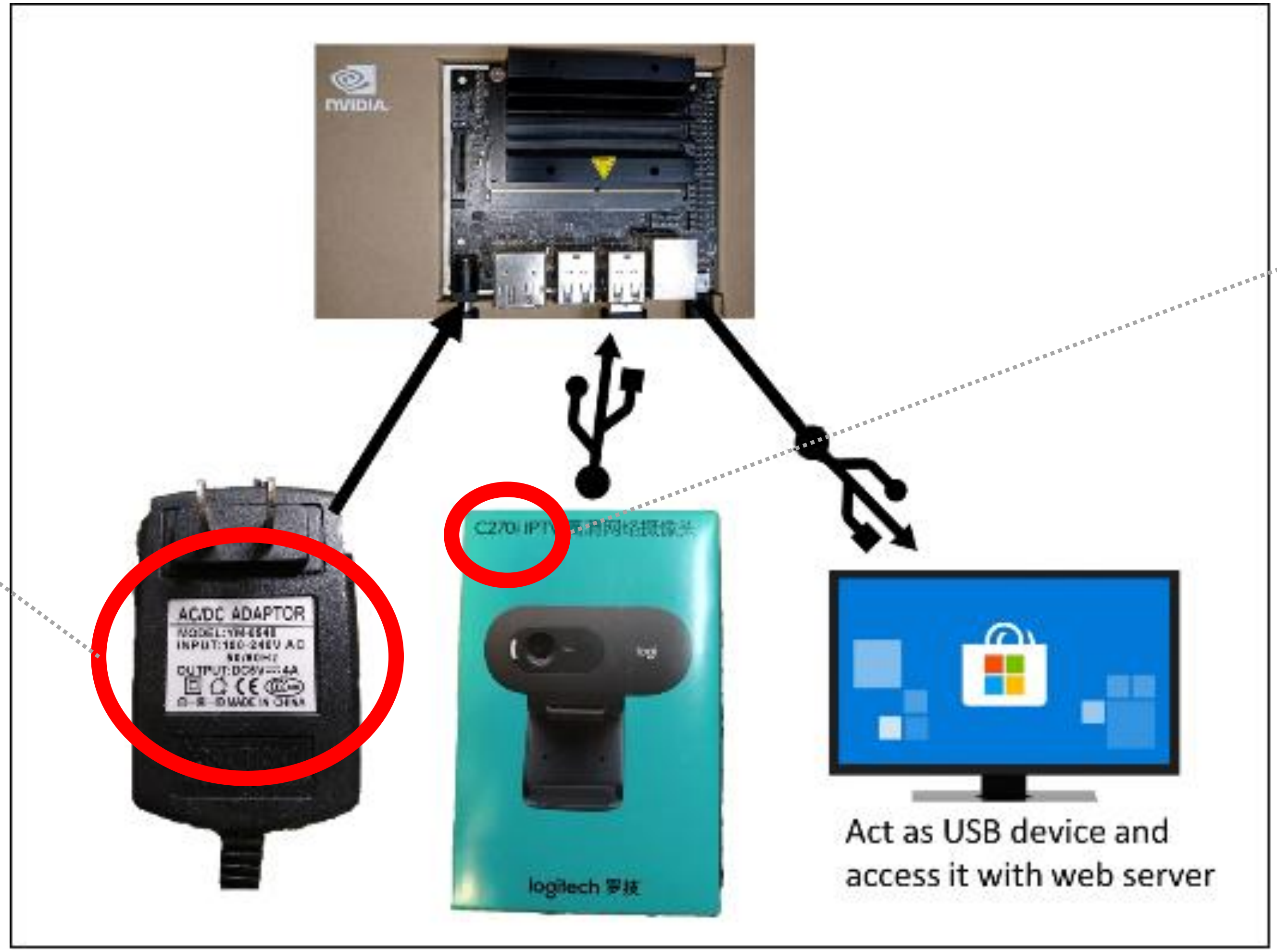
Custom
Classification
Label

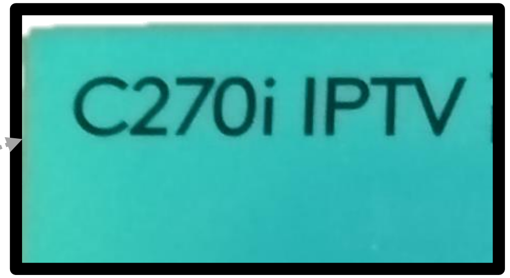


Methodology

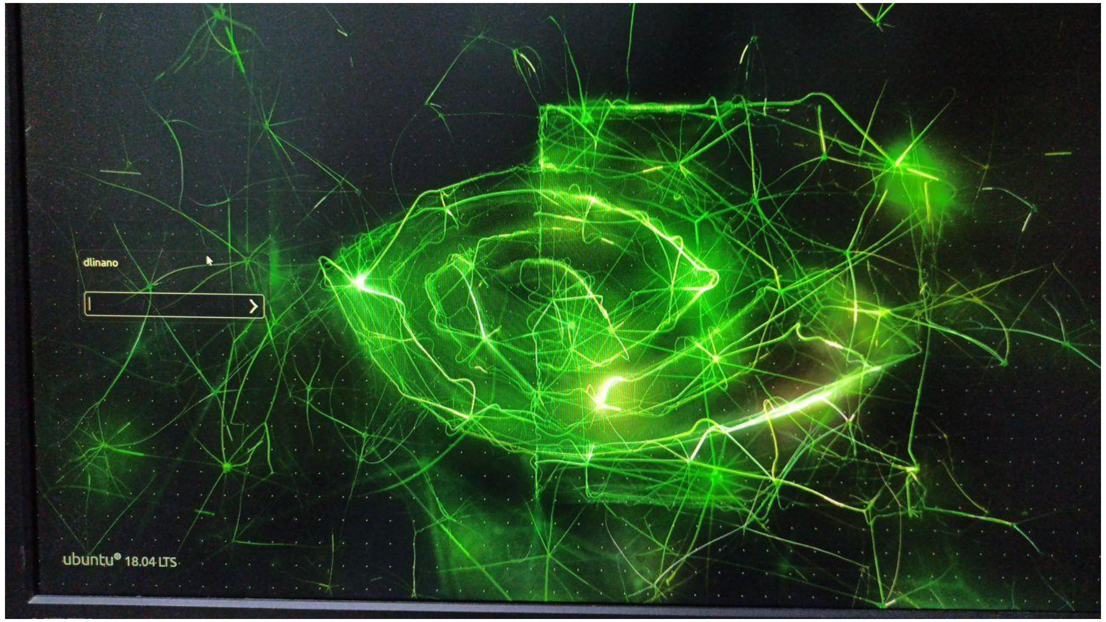
Connection


AC/DC ADAPTOR
MODEL:YM-0540
INPUT:100-240V AC
50/60Hz
OUTPUT:DC5V⎓4A
MADE IN CHINA

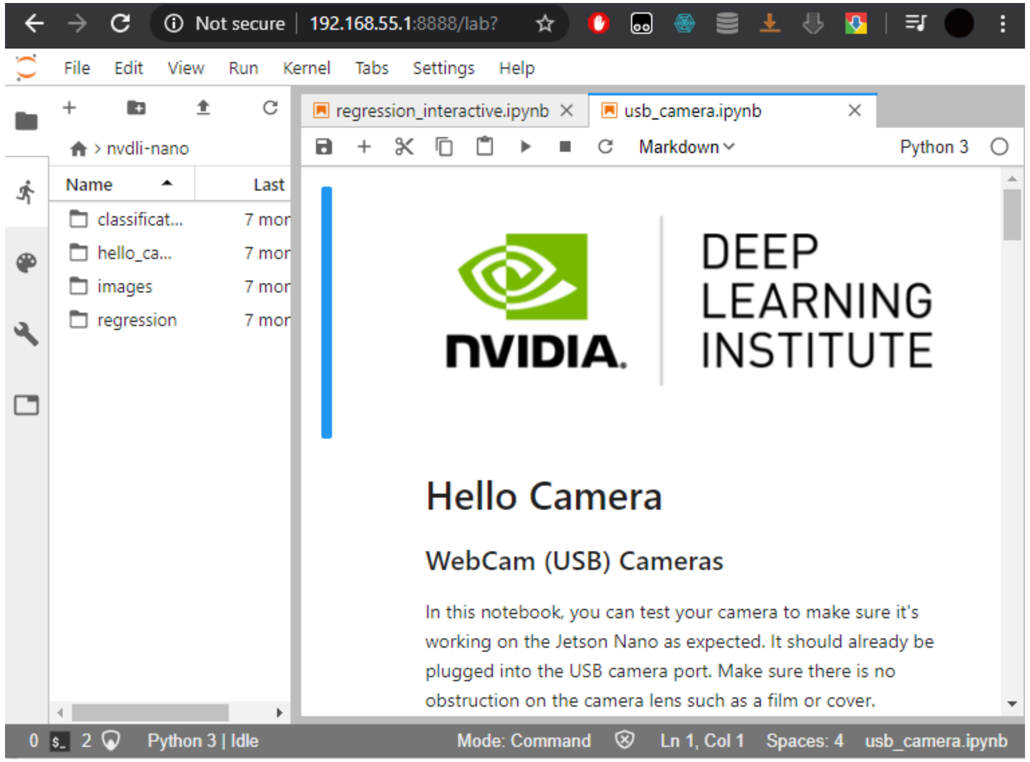

Act as USB device and
access it with web server


C270i IPTV

Turning on Nvidia Jetson



Login page of Jetson Nano Image



Jupyter Lab development environment

Setting Up the environment

```
>> from jetcam.usb_camera import USBCamera¶

>> camera = USBCamera(width = 224, height = 224, capture_width = 640, capture_height = 480, capture_device = 0)

>> image = camera.read()¶
```

```
[4]: print(image.shape)
(224, 224, 3)

[5]: print(camera.value.shape)
(224, 224, 3)

[6]: print(image)
[[[ 41  61  76]
  [ 54  76  86]
  [ 61  78  88]
  ...
  [109 112 112]
  [106 112 112]
  [103 112 111]]

 [[ 30  46  62]
  [ 57  75  85]
  [ 61  78  89]
  ...
  [112 113 113]
  [110 112 112]
  [106 112 111]]
```

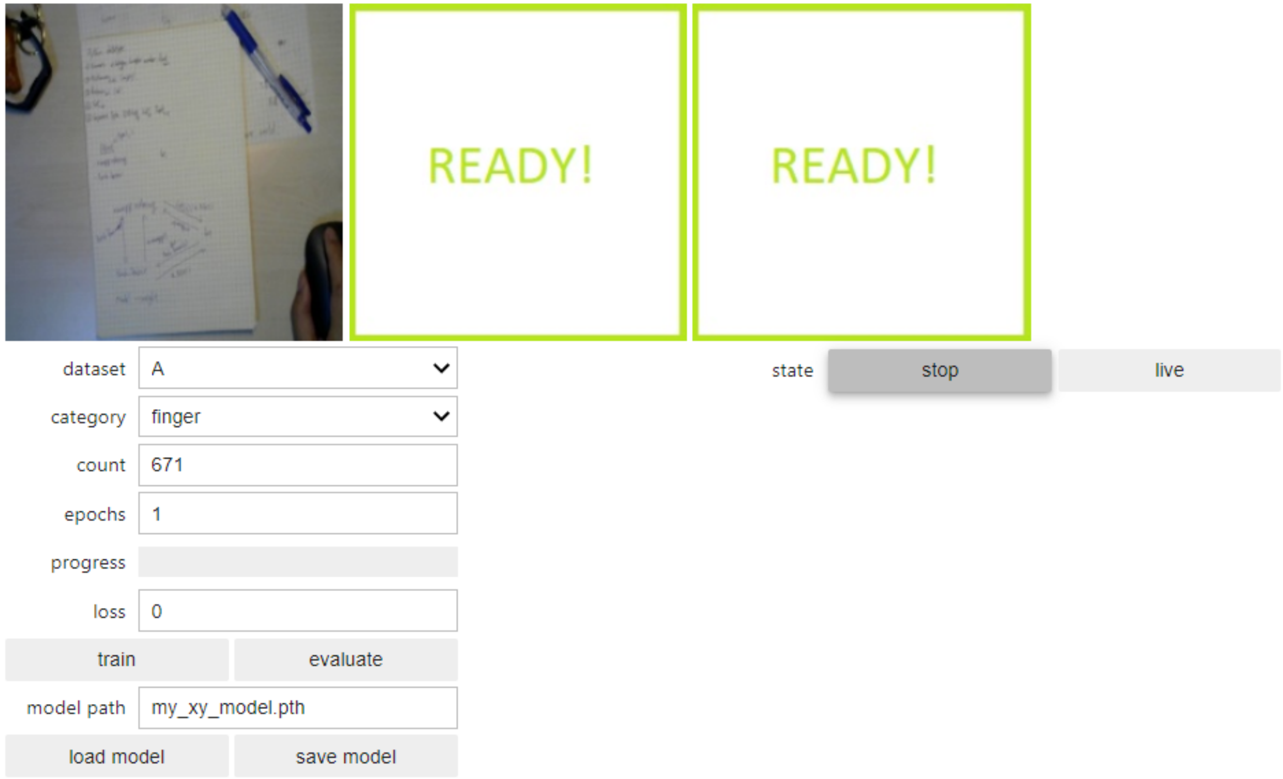
```
import ipywidgets
from IPython.display import display
from jetcam.utils import bgr8_to_jpeg

image_widget = ipywidgets.Image(format='jpeg')

image_widget.value = bgr8_to_jpeg(image)

display(image_widget)
```

Python Module used

- Ipywidgets
- IPython
- jetcam

Interface for Transfer Learning

```
# Combine all the widgets into one display
all_widget = ipywidgets.VBox([
    ipywidgets.HBox([data_collection_widget, live_execution_widget]),
    train_eval_widget,
    model_widget
])

display(all_widget)
```



ResNet-18

Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2 $\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

PyTorch Model prepare

$\gg$ *import torch*¶

$\gg$ *import torchvision*¶

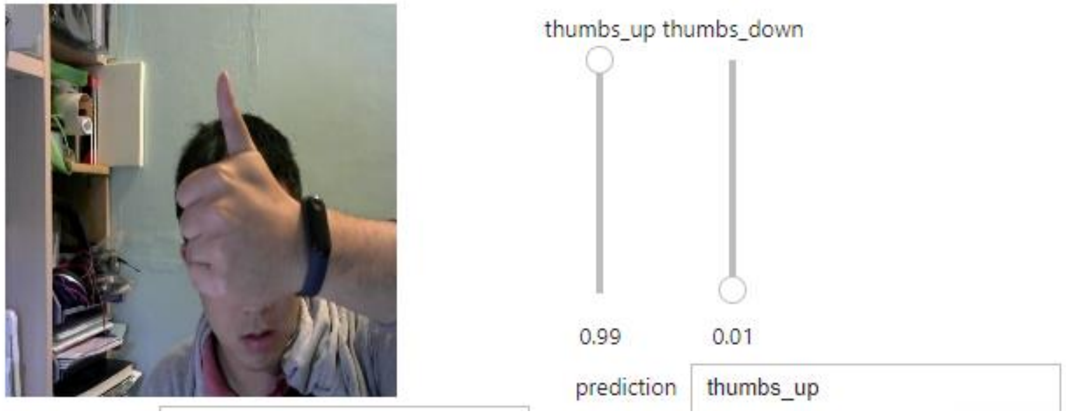
$\gg \ model \ = \ torchvision.models.resnet18(pretrained = \boxed{True})$¶

$\gg \ model.fc \ = \ torch.nn.Linear(512, len(dataset.categories))$¶

Replace the final fully connected layer for Transfer Learning

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

On-device deep learning for Classification



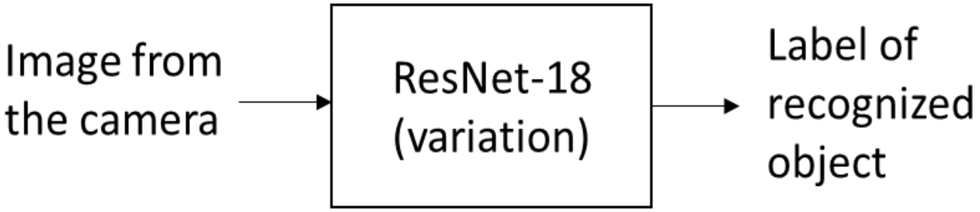
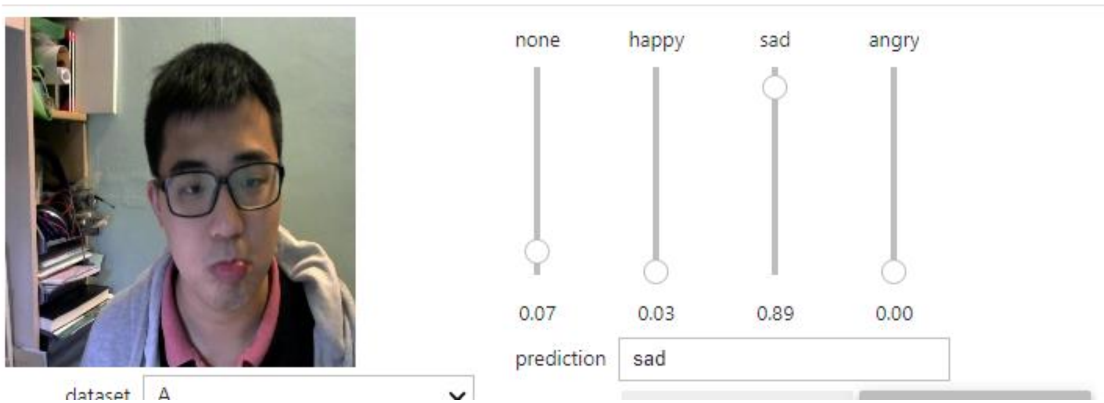
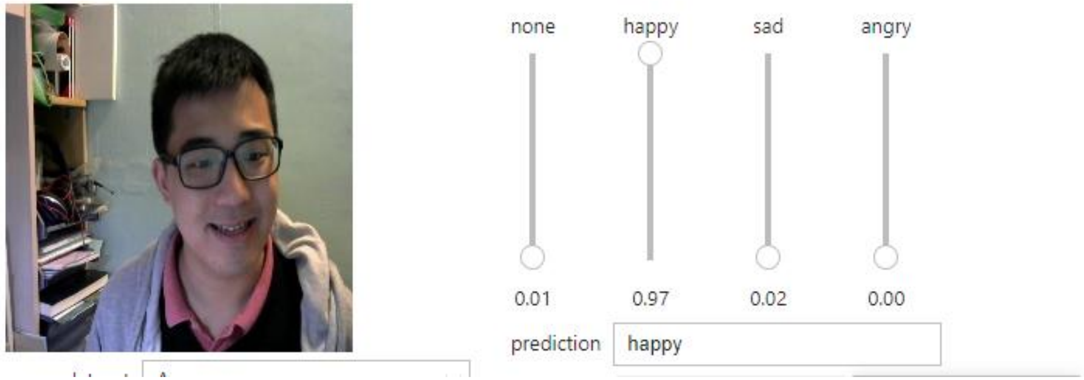
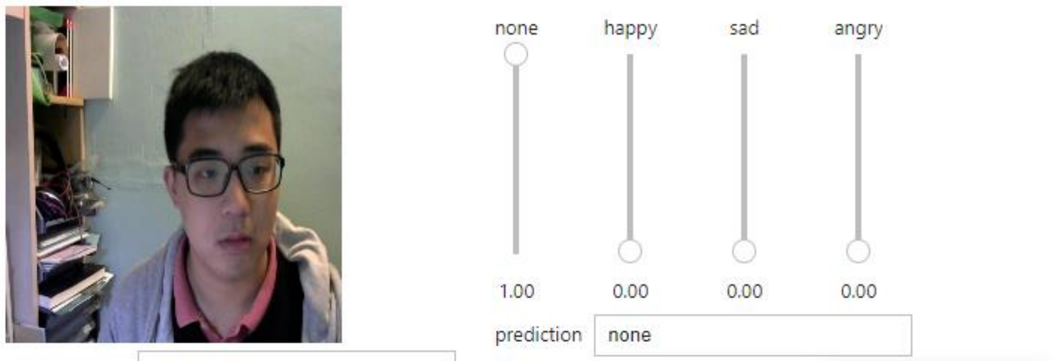


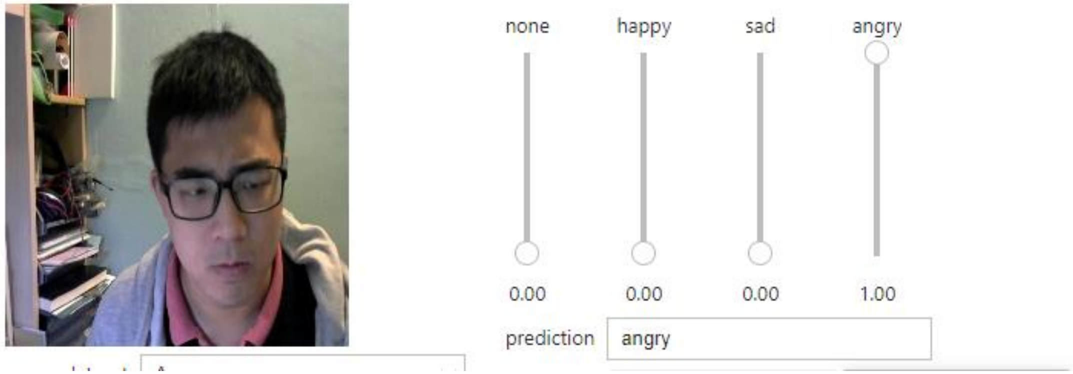
Figure shows the block diagram of the function of object recognition

On-device deep learning for Classification









On-device deep learning for Classification (Con’t)

$$\sigma(\boldsymbol{z}) = \frac{e^{\boldsymbol{z}}}{\sum_{\boldsymbol{z}} e^{\boldsymbol{z}}}$$

$\gg import\ torch.nn.functional\ as\ Func$

$\gg output\ =\ model(preprocessed)$

$\gg\ output\ =\ Func.softmax(output, dim = 1).detach().cpu().numpy().flatten()$

$\gg\ ocategory\_index\ =\ output.argmax()$

normalize the output to a range (0,1), and the sum is 1

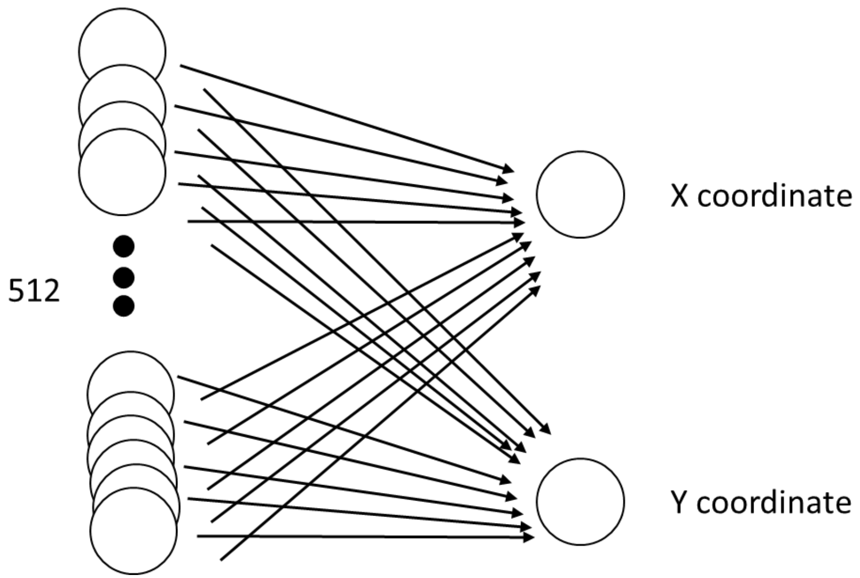
It turns the non-normalized output into probabilities distribution

$\gg prediction\_widget.value\ =\ dataset.categories[category\_index]$

*Using different background may help for the performance

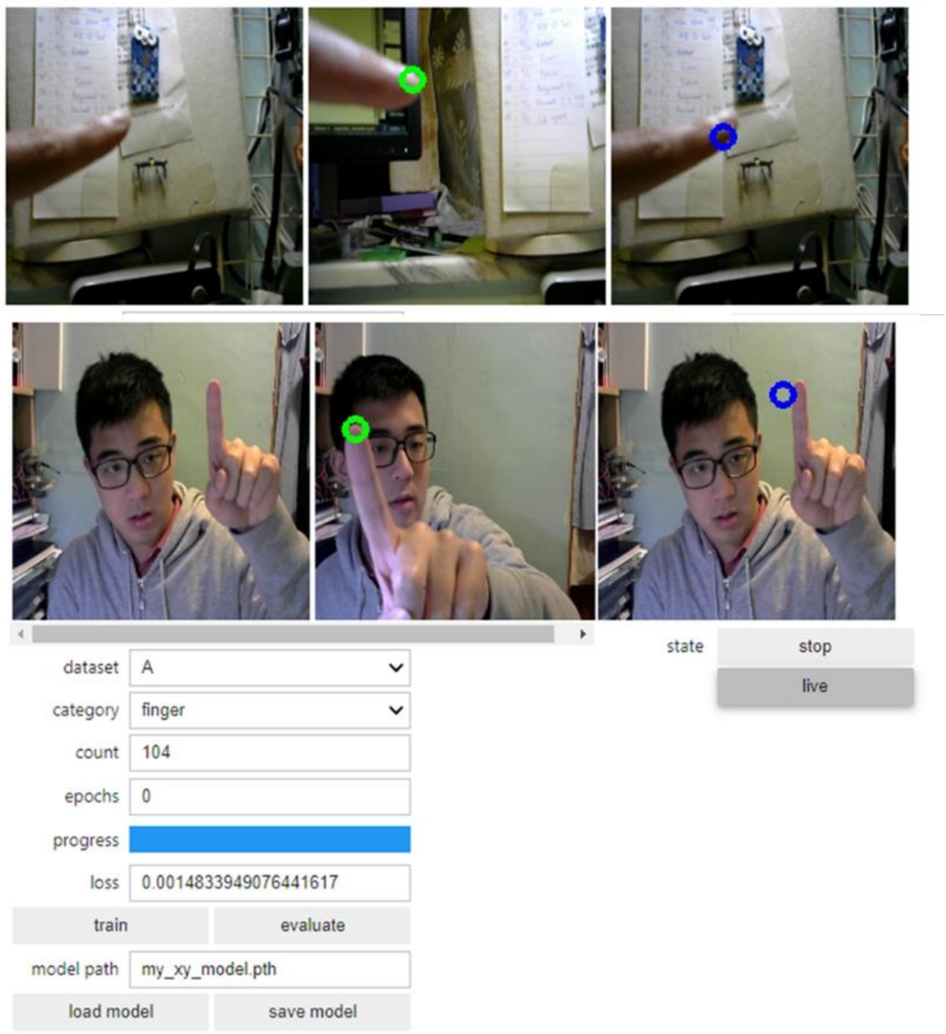
On-device deep learning for Regression

Goal: Tracking Finger



*The regression model does not require a Softmax layer

On-device deep learning for Regression



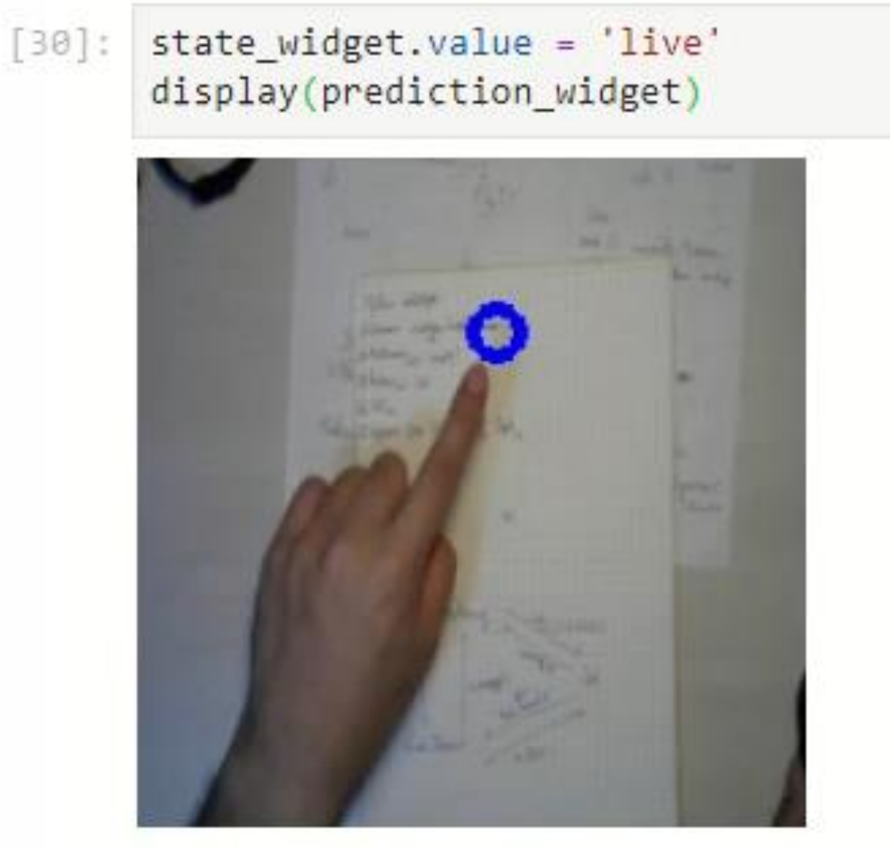


Figure D2a

The performance of tracking finger with few sample

Looking into the Model

$\gg$ *import torch*

$\gg$ $model\_state\_dict = torch.load("./myModelPath.pth")$

$\gg$ $model\_state\_dict['fc.weight']$

$\gg$ $model\_state\_dict['fc.weight'].size()$

$\gg$ $model = load\_state\_dict(model\_state\_dict)$

$\gg$ $model.eval()$

$\gg$ $torch.save(model.state\_dict(), "./myModelPath.pth")$

We will obtain the weight of fully connected layer and process it to perform model compression

The·ouput·of·the·$\gg$ $model\_state\_dict['fc.weight']$¶

```
tensor([[-0.0332,··0.0090,·-0.0124,·...,··0.0529,·-0.0117,·-0.0261],¶
········[-0.0091,··0.0018,··0.0331,·...,·-0.0393,·-0.0008,··0.0010]],¶
······device='cuda:0')¤
```

Model Compression using KMeans

```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
Clusters = []
Labels = []
for i in range(len(PTH)):
  np_PTH = np.array(PTH[i])

  km = KMeans(n_clusters=16)
  km.fit(np_PTH.reshape(-1,1))  # -1 will be calculated to be 13876 here

  # print(km.labels_)
  Labels.append(km.labels_)
  Clusters.append(km.cluster_centers_)
```

Import Python Model, Scikit-Learn, a common Machine Learning python package

Setting the number of clustering

```
Labels = []
for i in range(len(PTH)):
```

Model Compression using KMeans

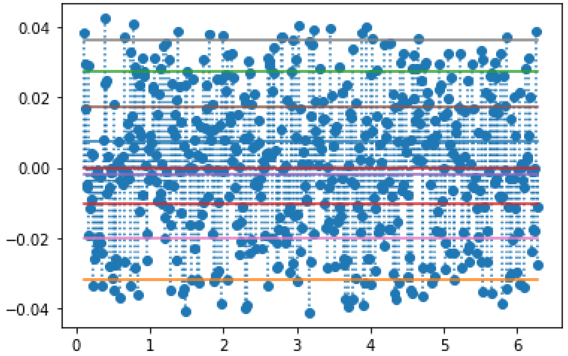
```
  # print(km.labels_)
  Labels.append(km.labels_)
  Clusters.append(km.cluster_centers_)
```

```
[0 3 3 4 6 0 5 5 3 7 1 4 3 5 4 1 7 6 3 4 5 1 3 5 7 5 6 4 1 6 4 7 2 0 7 4 0
 0 5 3 7 1 1 3 6 2 0 6 4 7 2 6 5 3 0 2 6 3 1 4 4 2 1 7 1 5 1 0 3 7 3 5 5 3
 4 4 1 5 7 6 7 1 6 5 5 3 3 2 0 5 7 4 3 3 6 2 7 3 0 3 6 0 1 0 5 7 2 5 4 5 2
 4 3 0 1 1 0 2 7 5 3 0 1 0 1 0 5 5 0 3 5 4 4 3 3 4 4 5 2 3 0 6 4 1 5 0 1 4
 0 7 7 6 0 1 3 5 3 0 0 5 1 0 0 5 3 1 7 5 1 3 2 3 3 7 7 1 2 0 1 5 1 0 4 6 7
 4 4 4 4 4 7 0 3 1 3 4 3 2 0 4 3 4 7 5 5 5 3 4 4 7 0 7 4 2 5 3 1 0 6 1 6 7
 3 5 4 5 4 5 0 6 1 2 7 6 4 5 0 6 3 6 4 1 0 6 5 3 1 7 5 6 2 4 1 6 1 1 0 3 3
 1 3 6 3 7 6 7 3 5 3 3 4 1 6 4 0 2 0 3 1 4 6 0 2 5 5 4 0 2 0 4 4 6 3 5 0 1
 1 4 7 4 4 1 7 7 3 3 0 6 1 5 5 4 7 0 4 4 3 2 7 6 5 4 6 7 5 6 3 2 4 6 0 1 7
 1 0 4 7 1 4 5 3 0 6 0 6 0 3 0 2 5 5 4 2 7 4 2 3 1 7 5 0 0 1 0 3 3 4 4 4 0
 5 1 0 4 2 4 0 6 4 4 5 7 6 7 7 4 6 2 6 6 3 1 0 0 7 5 1 3 6 2 2 6 5 7 3 3 5
 7 1 0 3 2 4 6 4 7 1 5 5 2 7 2 7 6 4 4 3 2 0 5 1 7 4 4 7 6 1 0 6 1 0 0 7 4
 5 2 4 6 1 6 1 4 7 1 3 5 3 7 4 4 1 1 1 3 3 2 2 4 1 2 0 0 2 3 1 7 0 1 0 6 1
 4 5 6 5 0 6 4 2 1 2 5 7 0 3 1 6 0 4 0 1 2 1 4 3 5 5 1 0 1 4 2]
array([[ 0.03393101],
       [-0.00904315],
       [-0.028715  ],
       [ 0.01348972],
       [ 0.00219211],
       [ 0.02392563],
       [-0.03764875],
       [-0.02073361]])
```

```
x = np.linspace(0.1, 2 * np.pi, 512)
for i in range(len(PTH)):
  np_PTH = np.array(PTH[i])
  plt.stem(x, np_PTH, use_line_collection=True, linefmt=':')
  for ii in range(len(Clusters[i])):
    plt.plot(x, np.zeros(x.size)+Clusters[i][ii])
  plt.show()

np_PTH_Quantization = []

for i in range(len(PTH)):
```

Model Compression using KMeans

```
x = np.linspace(0.1, 2 * np.pi, 512)
for i in range(len(PTH)):
  np_PTH = np.array(PTH[i])
  plt.stem(x, np_PTH, use_line_collection=True, linefmt=':')
  for ii in range(len(Clusters[i])):
    plt.plot(x, np.zeros(x.size)+Clusters[i][ii])
  plt.show()
```

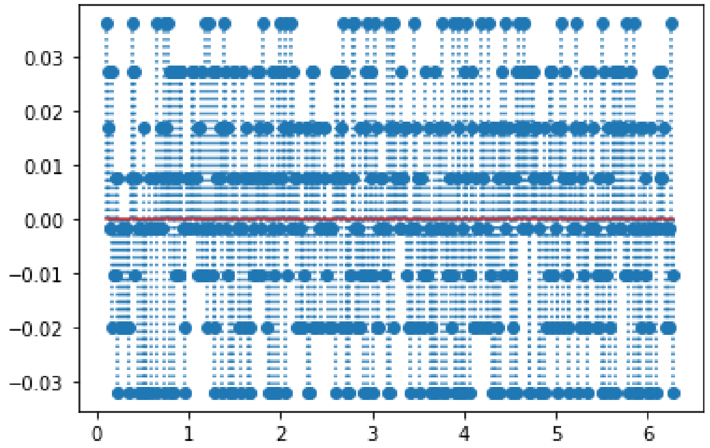



```
np_PTH_Quantization = []

for i in range(len(PTH)):
  np_PTH_Quantization.append(np.zeros(len(PTH[0])))
  for ii in range(len(PTH[0])):
    for iii in range(len(Clusters[i])):
      if Labels[i][ii] == iii:
        np_PTH_Quantization[i][ii] = Clusters[i][iii]
```

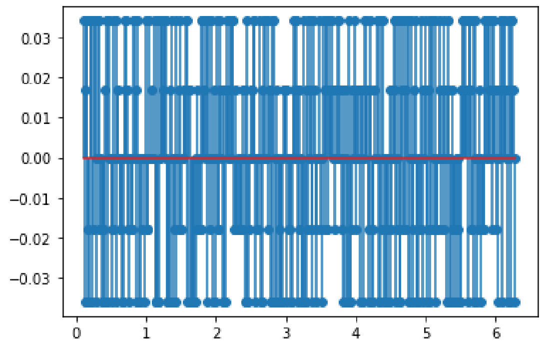
```
    for ii in range(len(Clusters[i])):
```

Model Compression using KMeans

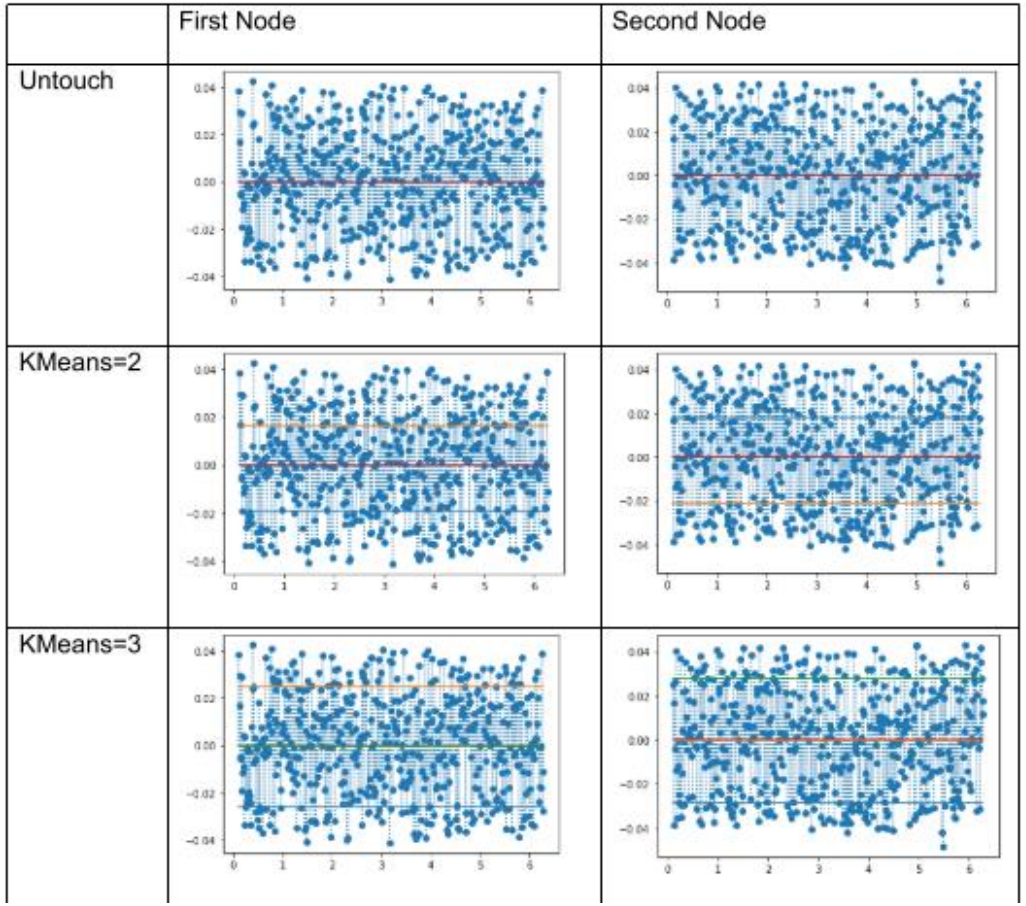
```
for i in range(len(PTH)):
    np_PTH_Quantization.append(np.zeros(len(PTH[0])))
    for ii in range(len(PTH[0])):
        for iii in range(len(Clusters[i])):
            if Labels[i][ii] == iii:
                np_PTH_Quantization[i][ii] = Clusters[i][iii]
np_PTH_Quantization_zero = []
```

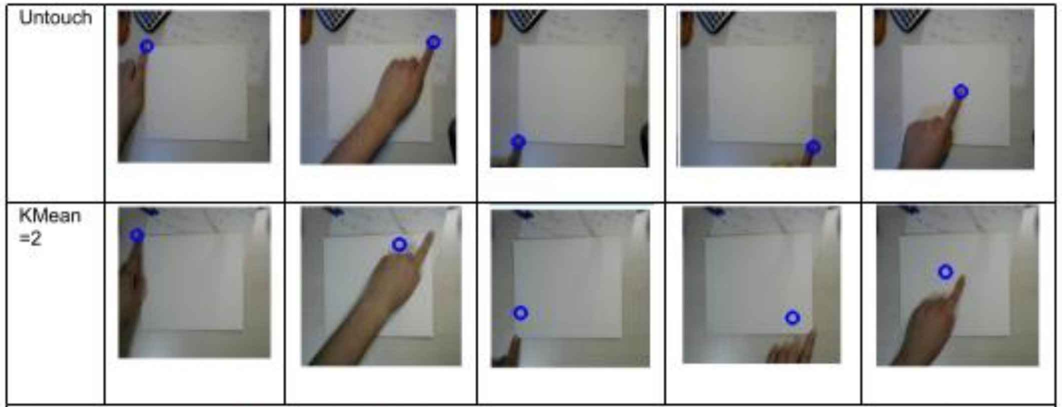



```
for i in range(len(PTH)):
    np_PTH_Quantization_zero.append(np.zeros(len(PTH[0])))
    miin = abs(Clusters[i]) == min(abs(Clusters[i]))
    for ii in range(len(PTH[0])):
        for iii in range(len(Clusters[i])):
            if Labels[i][ii] == iii:
                np_PTH_Quantization_zero[i][ii] = Clusters[i][iii]*(not miin[iii])
```

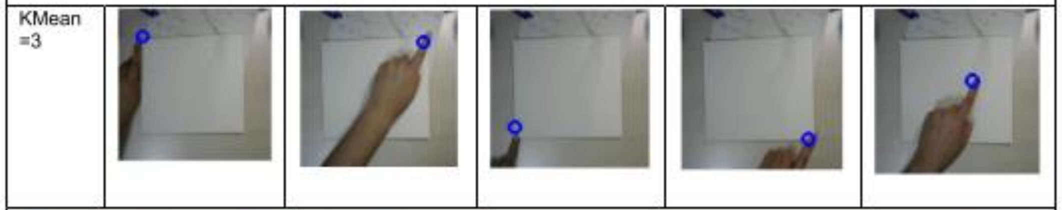


Result on Pruning with KMean (different k)



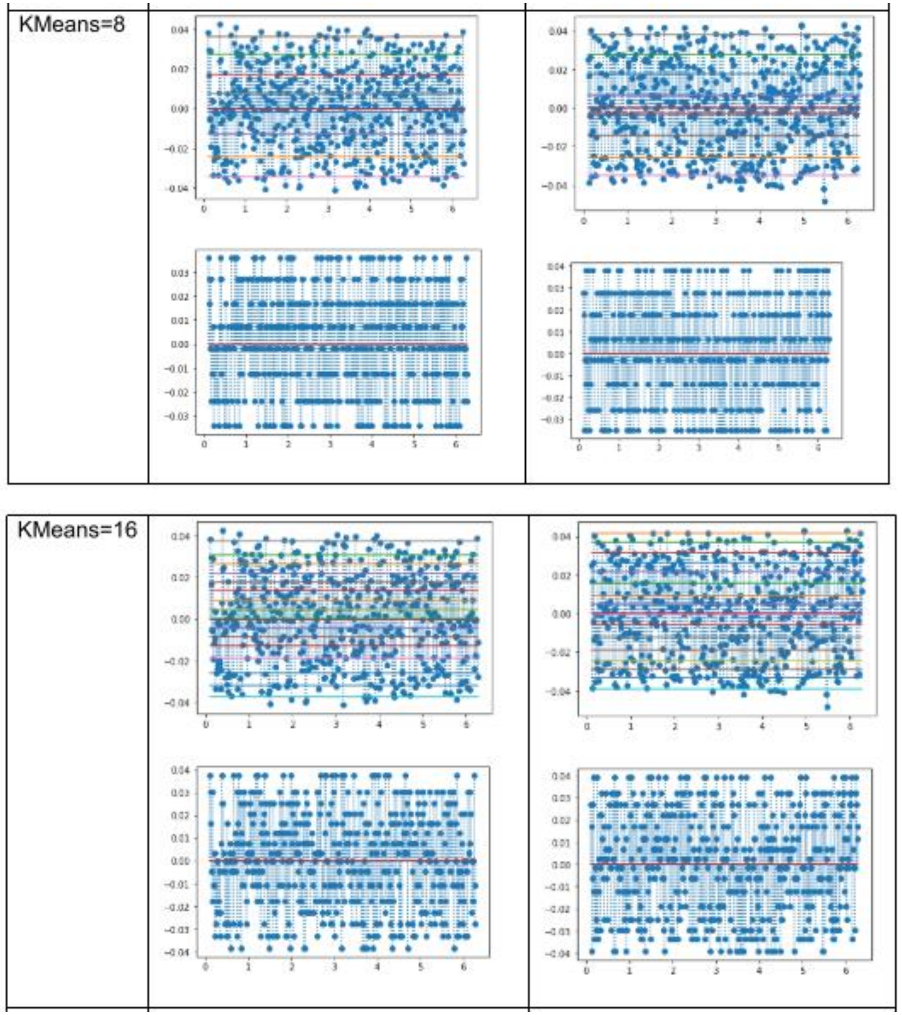



```
model_state_dict['fc.weight']

tensor([[ 0.0163,  0.0163,  0.0163,  ...,  0.0163, -0.0193, -0.0193],
        [ 0.0181, -0.0213,  0.0181,  ...,  0.0181,  0.0181,  0.0181]],
       device='cuda:0')
```




```
model_state_dict['fc.weight']

tensor([[ 0.0248,  0.0248,  0.0248,  ...,  0.0248, -0.0002, -0.0261],
        [ 0.0280, -0.0287,  0.0280,  ...,  0.0280,  0.0280, -0.0002]],
       device='cuda:0')
```

Result on Pruning with KMean (different k)



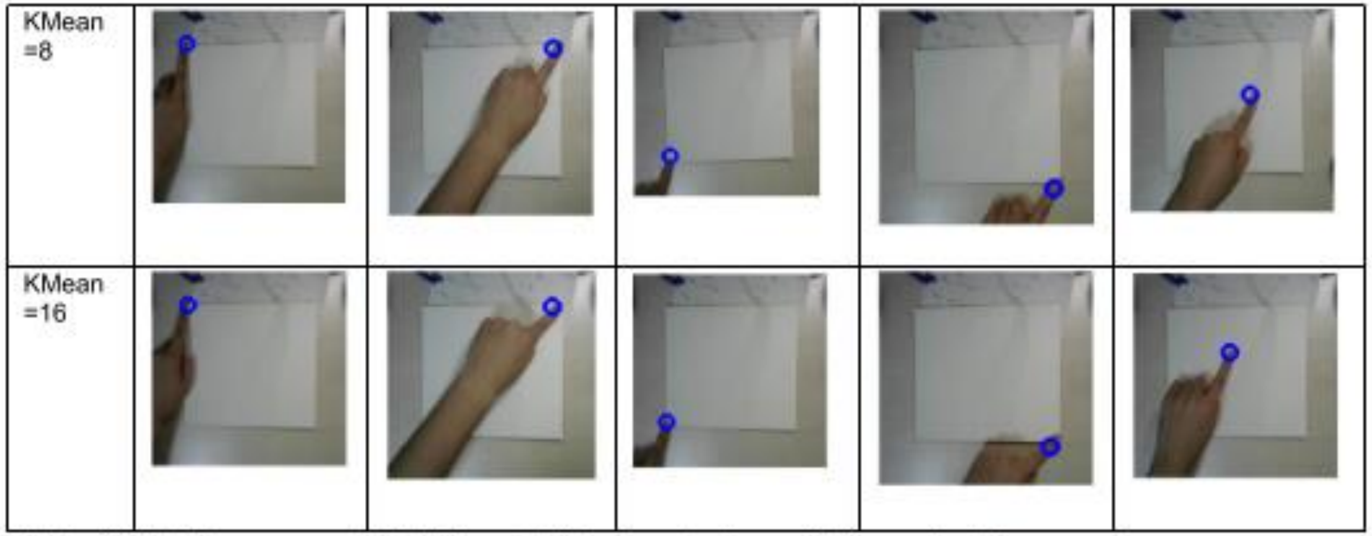


Table 2: Performance of 5 different KMean clusters with quantization

Comparing Result on Pruning

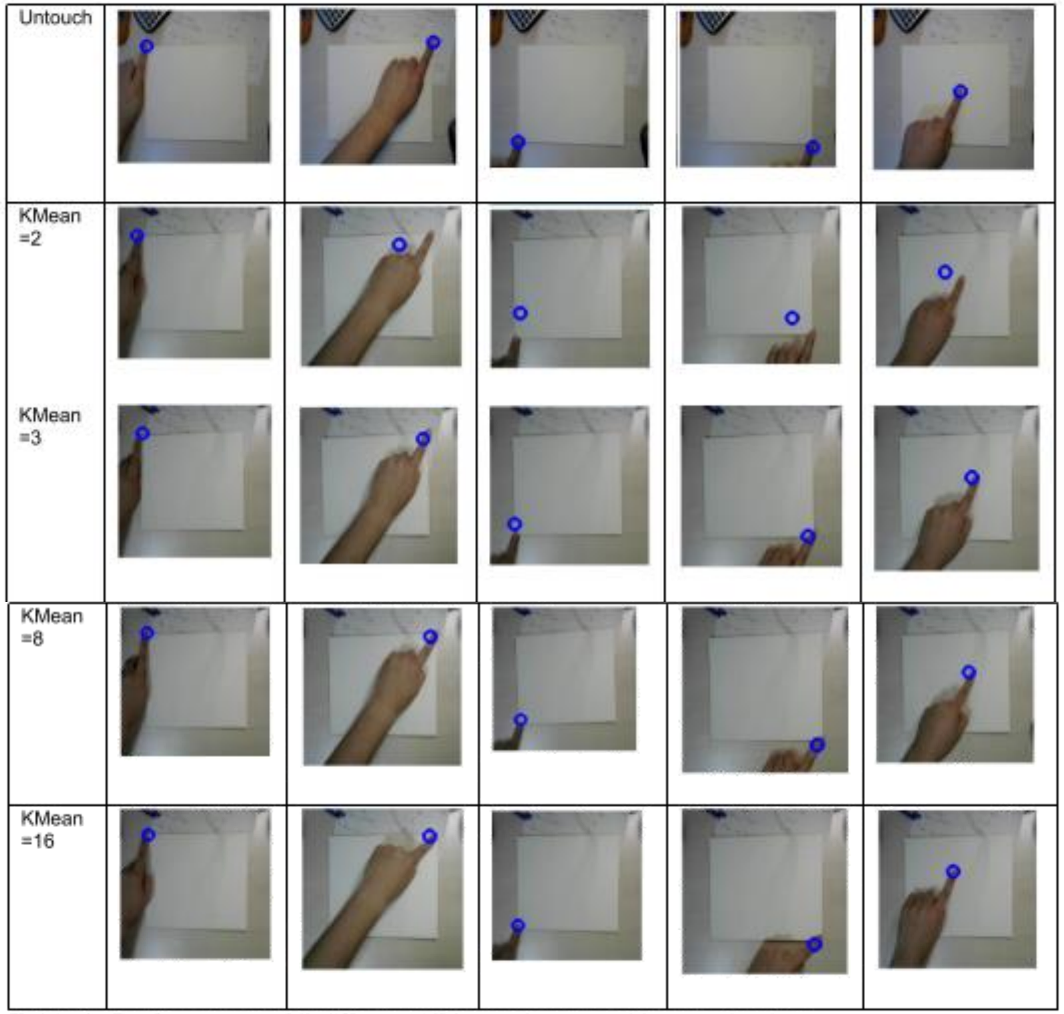


Table 2: Performance of 5 different KMean clusters with quantization

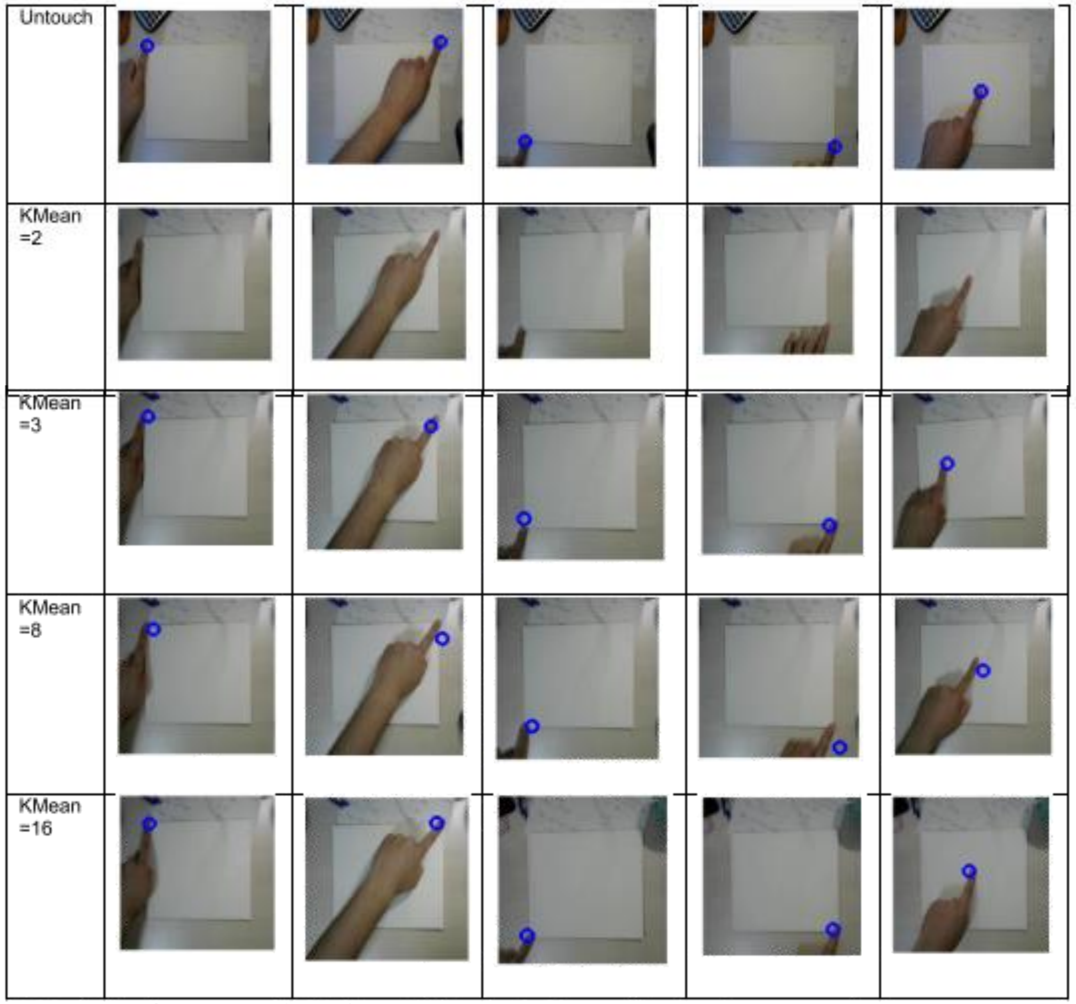
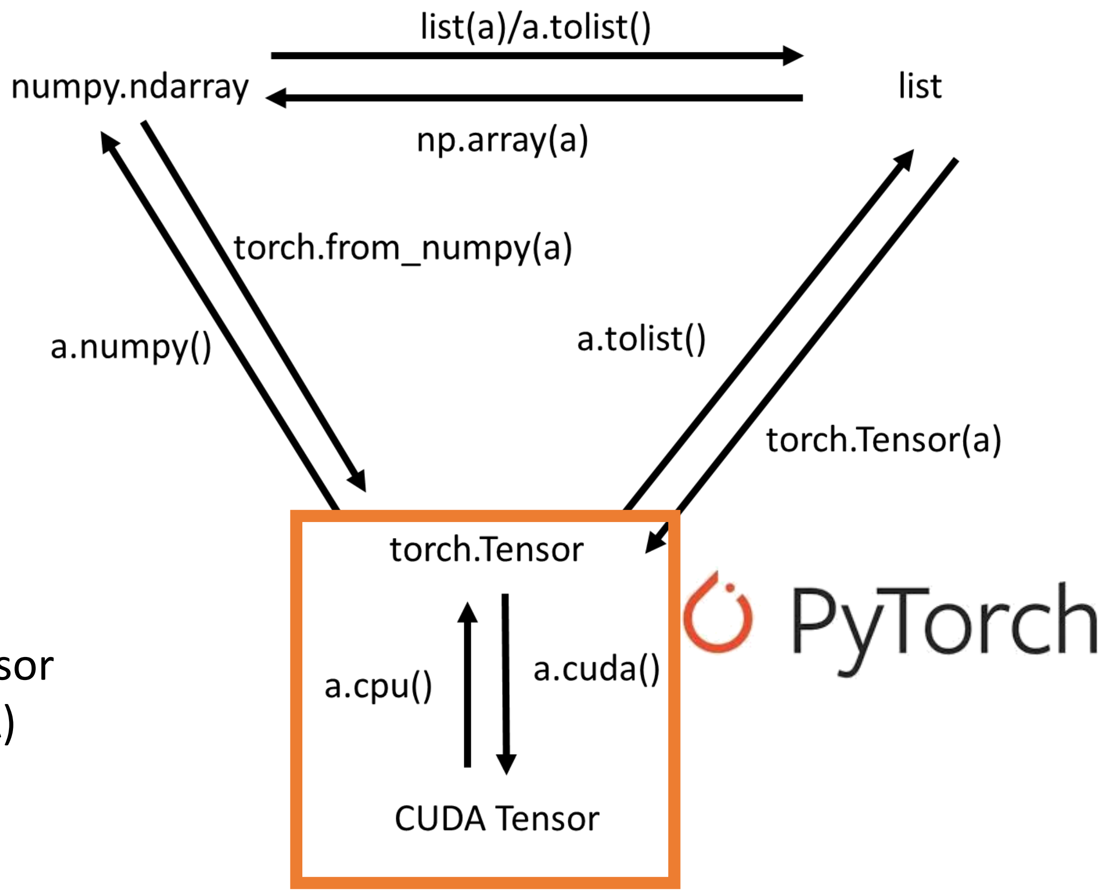


Table 3: Performance of 5 different KMean clusters with close-to-zero values

Datatype relation

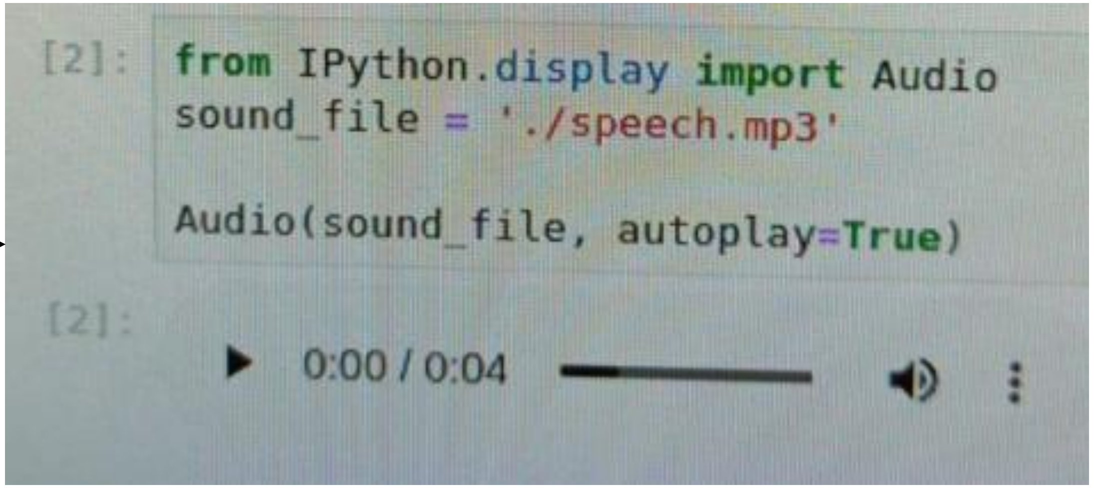


We need to load the Tensor
to the Nvidia GPU (CUDA)

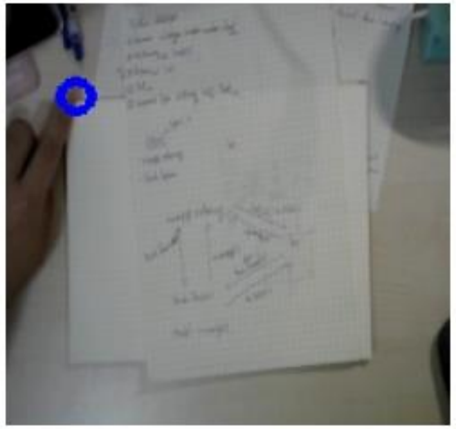
Audio

```
from gtts import gTTS
mytext = 'Hello Cenz'
language = 'en'

myTTS = gTTS(text=mytext, lang=language, slow=False)
myTTS.save("speech.mp3")
```

Figure Showing the USB adaptor



Application



```
image = camera.value
preprocessed = preprocess(image)
output = model(preprocessed).detach().cpu().numpy().flatten()
category_index = dataset.categories.index(category_widget.value)
x = output[2 * category_index]
y = output[2 * category_index + 1]

x = int(camera.width * (x / 2.0 + 0.5))
y = int(camera.height * (y / 2.0 + 0.5))

print(x)
print(y)
```

```
38
45
```

```
def live(state_widget, model, camera, prediction_widget, score_widget):
    global dataset
    while state_widget.value == 'live':
        image = camera.value
        preprocessed = preprocess(image)
        output = model(preprocessed)
        output = F.softmax(output, dim=1).detach().cpu().numpy().flatten()
        category_index = output.argmax()
        prediction_widget.value = dataset.categories[category_index]
        response()
        for i, score in enumerate(list(output)):
            score_widgets[i].value = score

############################################
from IPython.display import Audio

thumb_down_sound_file = './thumb_down.mp3'
thumb_up_sound_file = './thumb_up.mp3'
previous_prediction_value = ""

def response():
    global previous_prediction_value
    if prediction_widget.value == "thumb_up" and (previous_prediction_value != prediction_widget.value):
        display(Audio(thumb_up_sound_file, autoplay=True))

    elif prediction_widget.value == "thumb_down" and (previous_prediction_value != prediction_widget.value):
        print(prediction_widget.value)
        display(Audio(thumb_down_sound_file, autoplay=True))
    previous_prediction_value = prediction_widget.value

response()
###################################################
```

Discussion

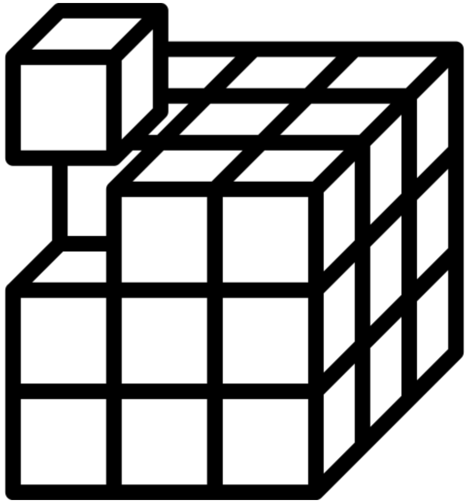
1. The computation power of Jetson Nano is **enough** for doing on-device deep learning

2. Model compression can **balance** between the **performance** and **storage** resources

3. The way to perform the clustering is to adapt the **KMean Clustering** algorithm with python module scikit-learn

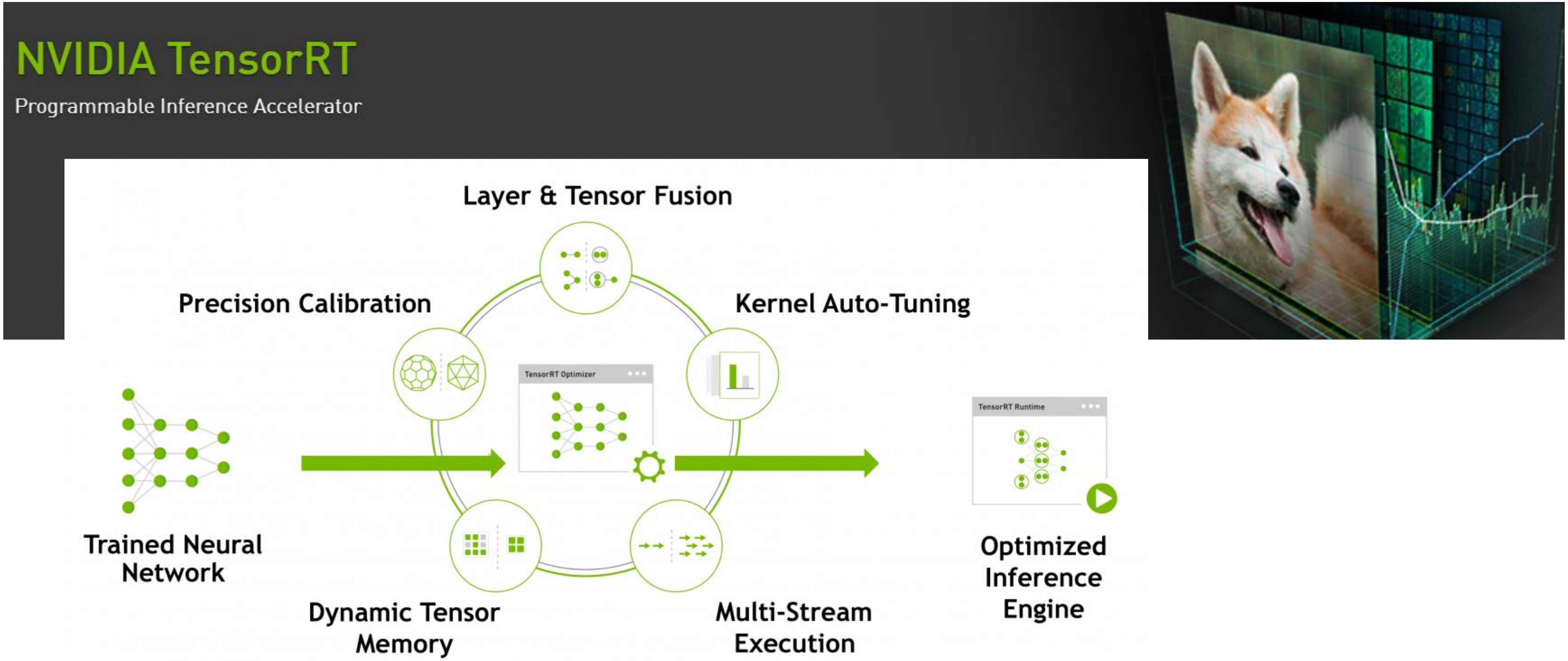
Recommendations and Conclusion

Heat dissipations



Bulky

Recommendations and Conclusion



Demo

Reference

[1] M. Bieri, “Artificial Intelligence: China’s High-Tech Ambitions,” CSS Analyses in Security Policy, Feb 2018
[2] Naveen Suda, Staff Engineer, “Machine Learning on Arm Cortex-M Microcontrollers,” White Paper
[3] Computer Vision Machine Learning Team, “An On-device Deep Neural Network for Face Detection,”
[4] Howard, A. G., Zhu, M., & Adam, H., “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” arXiv preprint arXiv:1704.04861., Apr. 2017.
[5] He, K., Zhang, X., Ren, S., & Sun, J. “Deep residual learning for image recognition”. In Proceedings of the IEEE conference on computer vision and pattern recognition, 2016
[6] Y. Cheng & D. Wang, “A Survey of Model Compression and Acceleration for Deep Neural Networks,” ., Feb. 2019.

Q&A