



THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

Department of Electronic and Information Engineering

Final Year Project Final Report (2019/20)

Model Compression for On-Device Deep Learning

Student Name: Wong Tsz Ho

Student ID: 15070475d

Programme Code: 42470

Supervisor(s): Dr Zhang J.

Submission Date: May 12, 2020

The Hong Kong Polytechnic University

Department of Electronic and Information Engineering

EIE4430 / EIE4433 Honours Project

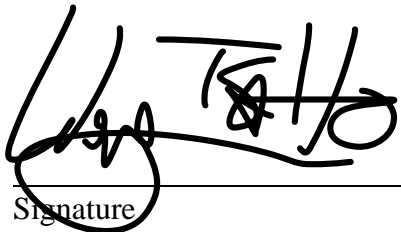
1. Student Name: _____Wong Tsz Ho_____ (Student No.: 1507045d)
2. Programme Code: 42470
3. Project Title: Model Compression for On-Device Deep Learning
4. Supervisor Name: Dr Zhang J
5. Project summary
 1. Design and Performing On-device deep learning with Nvidia Jetson Nano using custom target data 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”

DECLARATION OF ORIGINALITY

Except where reference is made in the text of this report, I declare that this report contains no material published elsewhere or extracted in whole or in part from any works or assignments presented by me or any other parties for another subject. In addition, it has not been submitted for the award of any other degree or diploma in any other tertiary institution.

No other person's work has been used without due acknowledgement in the main text of the Report.

I fully understand that any discrepancy from the above statements will constitute a case of plagiarism and be subject to the associated.



Signature

Abstract

This final year report is to discuss an application which utilizes the model compression for on-device deep learning. The use case is a "Deep Learning aided Visual System." The system will use the model compression method, deep learning knowledge, and finally put the network on the device, in this project, Nvidia Jetson Nano. The deep learning framework used is PyTorch, one of the popular deep learning frameworks in the academic field. The neural network architecture used will be ResNet-18. The objective of this project is to evaluate the performance of the model-compression for on-device deep learning and discuss the feasibility of developing an application on it. The technique used here is transfer learning, which can transfer the domain knowledge of a pre-trained model into some custom datasets. In this project, it will demonstrate how to perform transfer learning on Jetson Nano, and how we can utilize the result to build other application on top of it. Next, it will demonstrate how the model compression performs with machine learning algorithm K-Mean clustering and evaluate the performance with each of the clustered models.

This report shows the ability of Jetson Nano, able to run a deep learning model ResNet-18 and Transfer Learning. Followed by the results and test is my recommendation of model compression for on-device deep learning.

Acknowledgments

I would like to express my thankfulness to my supervisor, Dr. Jun Zhang, for guidance. Throughout the development of the idea of the thesis, he has tried to provide all forms of possible and feasible assistance. His answers bring aspiration to me during the discussion about the problems I faced. He deserved the accreditations of the projects.

Content Page

Abstract	4
Acknowledgments	5
List of Tables and Figures	7
A. Introduction	8
A1. Background and the context of the problem	8
A2. Objective	10
A3. Summary and organization of the chapter	10
B. Literature Review	11
C. Methodology	13
C1. Overview	13
C2. Connecting the camera to Jetson Nano	14
C3. JupyterLab	17
C4. Transfer Learning	20
C5. Regression Model	25
C6. Handling the model	26
C7. Pruning and Sharing the weights using KMeans	27
C8. Using Model for application development	33
C9. Audio system	34
C10. Hardware	35
D. Findings/Results	37
D1. Classification	37
D2. Regression	39
D3. Pruning and Sharing the weights using KMeans	40
D4. Pruning with quantization	43
D5. Pruning with close-to-zero values	44
E. Discussion	46
F. Recommendations and Conclusion	47
G. Reference	49
H. Appendix	51

List of Tables and Figures

List of Figure

Figure C1a	Block Diagram of the system	13
Figure C1b	A diagram illustrates the application	13
Figure C2a	Login page of Jetson Nano image	14
Figure C2b	Code block showing image capture by JetCam	15
Figure C2c	Code block showing image widget	17
Figure C3a	Code showing ioypad widget code	18
Figure C3b	Code block showing complicated layout	19
Figure C4a	The block diagram of the function of object recognition	23
Figure C5a	The fully connected layer of the regression model	25
Figure C7a	Code block showing stem plot	27
Figure C7b	Stem plot of a single node	28
Figure C7c	Code block showing plotting stem plot with center	29
Figure C7d	Code block showing plotting stem plot with center	30
Figure C7e	Code block showing running a model	31
Figure C7f	a diagram showing a conversion of different list and array	32
Figure C8a	Code block showing how to use the model	33
Figure C9b	Showing the USB Adaptor	34
Figure C10a	Picture of Nano	35
Figure C10b	Picture of connection diagram	36
Figure C10c	Jupyter Lab development environment	36
Figure D2a	The performance of tracking finger with few sample	39
Figure D3a	Code block showing tensor state dict	40

List of Table

Table 1	Stem plot of different KMeans for first and second node	40
Table 2	Performance of 5 different KMean clusters	42
Table 3	Performance of 5 different KMean clusters with close-to-zero values	43

A. Introduction

A1. Background and the context of the problem

This project is to propose an application to run model compression for on-device deep learning and thus to evaluate the efficiency of model compression for on-device deep learning. The application was proposed to be a "Deep Learning aided Visual System." Current application on the deep learning aided system required a lot of internet or cloud service. With the development of the hardware technology, chips are running faster and faster, and some ASIC are specified to calculate tensor operation. Those technologies enable on-device deep learning. "Deep Learning aided Visual System" needed the system response faster instantaneously and to be more privacy-sensitive. On-device deep learning is the best solution for this application. Since the resources on the device are limited, a special technique is needed, like model compression technique or hardware acceleration.

Around the global, artificial intelligence (AI) development had seen a boom. However, those AI development is heavily relying on computational resources. Cloud AI services like Microsoft Azure, IBM Watson, and Google Cloud become very common nowadays. A lot of industry is being "transform" by AI technology. Those solutions, using centralized networks, cloud service, may have few problems for the customer. First, the AI service cannot be work without the internet, and second, the business confidential may leak to the tech giant. Cloud service may lead to more problems, like wasting a lot of traffic to send raw data. Thus, the tech giants are changing their strategy from Cloud AI to Edge AI, which improves the service response speed and alleviate the traffic. We can see the trend of AI service is moving toward the end-user. Therefore, on-device AI will be the destination. Chipsets designed from Giant like

Google, Nvidia, Canaan are focusing on computing tensor operation. China, one of the biggest nations in the world, is putting a lot of resources into the AI industry. The State Council of China [1] released the "New Generation Artificial Intelligence Development Plan" in 2017. The new business model is encouraged and enhanced, "AI+." This model can improve the efficiency of the traditional industry, which can also enhance performance and profit. It is the new trend of the industrial revolution.

On-device deep learning is not an unreachable dream for the researcher to be achieved. Google, a leading tech giant on artificial intelligence, has released its first smartphone using on-device deep learning, Google Pixel 4. "Pixel Neural Core™" is the key where it can perform on-device deep learning for Google Assistance. For other smartphones, Google or Apple has provided a different level of API abstraction, like Apple Core ML, Android NN, CMSIS-NN, and ARM NN, for the developer to develop their artificial intelligence application [2]. Different frameworks are now adapting more lite network. TensorFlow, one of the most popular frameworks, must launch TensorFlow Lite for Android, iOS, or ARM64 while it is actively developing TensorFlow Lite Micro. Nvidia, a company develops GPU, has developed a smaller, on-device deep learning series called Jetson for the developer to develop an application on it. Those examples show that the whole industry is targeting the on-device application. Thus, this project is also targeting the related skills, model compression for on-device deep learning.

A2. Objective

This project aims to run a deep learning model on a device by using the technique of model compression and thus evaluate the efficiency of model compression for on-device deep learning. By applying the model compression technique for on-device deep learning, an application using on-device Deep Learning is designed and shows the result for deep learning. The application will be “Deep Learning aided Visual System”. Users, wearing the camera next to glasses, can point to any object, and the system will tell you what object is by the on-device deep learning model.

A3. Summary and organization of the chapter

This report is going to present why model compression is important to on-device deep learning. It comes with literature review which to discuss the machine learning running on device and how other device and chipset create their form of on-device deep learning framework. Next it will be studying the ResNet-18 and how transfer learning can be used to perform custom target data classification and regression. Finally, it will visit some of the model compression technique for saving the model size and computation power. This report will demonstrate running classification model and regression model on Jetson Nano, and perform transfer learning from pretrained model, ResNet-18 to a custom data. By studying the model weight, this report will prune the final layer of the model to perform the model compression and evaluate the performance of the compression technique. The weight are group with serval cluster, and this report are using machine learning technique to cluster those weight and grouping and sharing weight in the final layer of the model. It will then evaluate which one is the best alternative. This report will end with some recommendation and conclusion that for next one who need a deeper study on this topic.

B. Literature Review

Stated by Computer Vision Machine Learning Team in Apple Inc, there are three benefits to perform on-device deep learning [3]. First, it saves the traffic and the computation power of the cloud computing because on-device deep learning can complete the task locally without using the computation power of the cloud service, which in turn saving the cost of the cloud computing. Second, user's privacy can be protected since the device is entirely offline to the internet. Third, the response of the AI can be a boost in which creating a low latency environment for demanding application. That three-point shows that it is crucial to study the way to run deep learning on an end-user device than on the cloud side.

For the deep learning architecture, in the early stage of the project, I was considering using the MobileNets[4] since the model is relatively efficient for embedded devices and mobile vision applications. ResNet[5], awarding champion in ILSVRC (ImageNet Large Scale Visual Recognition Challenge), is easier to be optimized and higher accuracy gains. The layers are range from 18-layers up to 152-layers. In this project, we will use the smallest network, that is ResNet-18. The reason why using the ResNet-18 is that we need to balance the power consumption and the accuracy of the use case.

To compress the deep learning model so that the model can run faster. According to the paper by Cheng and Wang [6], deep learning model compression can be done by following,

1. Pruning and Sharing
2. Quantization and binarization,

3. Designing Structural Matrix.

Those three techniques are widely used to perform the model compression. First, for the Pruning and Sharing, some similar weight can be group together, and then it can save some memory. For example, if one weight is around 2.1 and 1.9, we can share two nodes to save one memory unit. Also, we can do the quantization and binarization to save more memory further. We can fix the weight into a specific value and estimate it with the nearest digit and put those positions into a table matrix. By freeing the memory of the weight, the resource-hungry device can have more room to run the interface, giving the optimized result.

Transfer learning is a technique for adapting the pretrained model and replaced it with a custom layer and retrained it as a new model. Transfer learning is needed if we do not have a lot of sample gathered for the model. The source data from the pretrained model may not be the same as our target data, however, with the help of transfer learning, it can feed target data into the pretrained model and fine-tuning the model. Transfer learning can be described as self-taught learning [7]. Compared to a typical model training from scratch, transfer learning saves times and computation resource and do not required a huge amount of labelled data, which makes transfer learning widely applicable to many practical learning problems

C. Methodology

"Model Compression for On-Device Deep Learning" is about demonstrating the feasible solution for on-device deep learning and evaluate model compression method efficiency. The following will show you the basic structure of the solution for "Deep Learning aided Visual System." It can basically be divided into three parts, object recognition, finger tracking, and text to speech part. This section will end with the hardware connection of the application.

C1. Overview

This application is designed since there is someone out there who is now suffering Presbyopia, which cannot see the thing clearly. "Deep Learning aided Visual System" .is designed for them to points an object, and the AI system will tell the user what the object is. This application requires very low latency and offline environment to the user. Since this is a personal gadget, the user may also take piracy into account. Figure C1a shows the overall logic flow of the system.

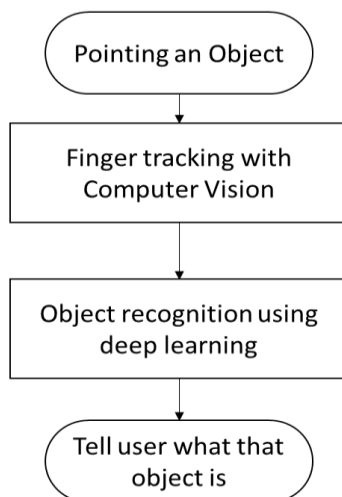


Figure C1a:

Block diagram of the system



Figure C1b

A diagram illustrates the application

The device is designed to wear on the glass, and it has a camera pointing outward. It has a small speaker which is put near the ear. Figure C1b shows the concept of this product. The system can be implemented not only on the Jetson Nano, but this project will also only indicates that the idea of the application is work, and the model can transform and translate onto a mobile APP, which cost is lower than a wearable gadget. But in the developing stage, we will use a separate device to build a fast solution.

C2. Connecting the camera to Jetson Nano

First of the step to deal with Nvidia Jetson Nano is to flash the correct image to the device. Nvidia has developed a lot of different distros for us to download [8]. For the beginner of the developer, we use the image (FigureC2a) "dlinano_v1-0-0_image_20GB.zip," which is pre-install JupyterLab service on it. The program I used to flash the image on the SD card is "Etcher-Portable-1.4.6-x64.exe". After flashing the SD card, insert it on the Nvidia Jetson Nano and grab a suitable power supply to turn on the device.

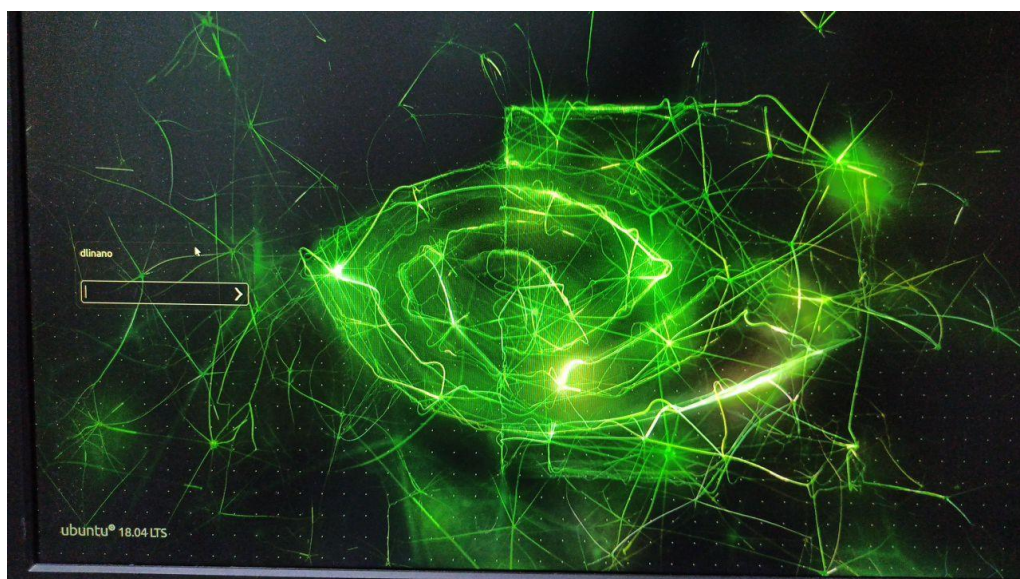


Figure C2a: FigureC2a: Login page of Jetson Nano Image

The Jetson Nano is quite an energy hunger, which the ordinary power bank of cell phone charger cannot power it up. Therefore, I buy a 5V4A adaptor separately. Since then, the Jetson Nano can run normally. The camera I used is a C270i IPTV camera from Logitech. It is a typical IP camera. To check whether Jetson nano can detect the camera, we can run the following command on the terminal.

```
>> ls -ltrh /dev/video *
```

The meaning of this command is to let the Linux list out as a 'long-format' with 'time sorting' and 'reverse' order in a 'better' format. Those flags allow us to check the device much faster. We should do the checking every time we use the Jetson Nano. The reason is the poor connection between the metal and the Jetson Nano. A runtime error was caused as the USB camera cannot be connected. To access the camera from the Nvidia Jetson Nano, we need a python module named 'jetcam.usb_camera'. This Python module (JetCam) is developed by Nvidia to provide an easy-to-use Python camera interface for Nvidia Jetson. The source code of JetCam is listed here: <https://github.com/NVIDIA-AI-IOT/jetcam>.

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

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

```
>> image = camera.read()
```

Firstly, we set up an object call camera and define that the image of the camera will be 640 and 480. Next, we can get the image from the camera with the method 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]]

```

Figure C2b: Code block showing image capture by JetCam

Figure C2b shows that the dimension of the image size is (224,224,3), which is in RGB format. Knowing the dimension of the camera is essential, especially for using CNN, since we need to design the details of the CNN Kernel and the size of the input shape. It is important to note that the image is in the format of BGR8, which is not in JPEG format. To view the image, we can utilize the module from 'jetcom.tils' (Figure C2c), `bgr8_to_jpeg(image)` to translate the image from BGR8 format into JPEG format, which we can open with. To view the image on a .ipy file, we can utilize the ipywidgets module.


```
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)
```



Figure C2c: Code block showing image widget


C3. JupyterLab

JupyterLab can be expressed as the next generation of Jupyter Notebook, which has fewer functions comparatively. JupyterLab integrates Jupyter Notebook and other utility into a web-based user interface app. We can open the terminal, python shell on JupyterLab. The interface for controlling Jetson Nano is also via JupyterLab using terminal and python script. JupyterLab allows us to access the file system on the left, which is very convenient to work with. It also has a different kind of widget [9] to use. We can directly type the Markdown language to write some comment on that code. This environment is ideal for developers to work with.

JupyterLab inside the Jetson Nano has a lot of widgets for us to build out user interface. We can then build out a user interface to display the captured image on the JupyterLab. The usage of the ipywidgets will be first, creating the object of the widget, next, setting the variable and parameter of the widget we use. And finally display it on ipython notebook. Figure C3a shows that the way to develop a visualizing code and interactive code from python using ipywidget is clean and simple. It links the front end to the backend, developer no need to write JavaScript or HTML to develop the UI for people to use. It has different widgets to interact with, we can run `.key()` method to check the attribute of the ipywidgets. The following program is built using ipywidgets.

```
[2]: import ipywidgets

[9]: Slider = ipywidgets.IntSlider()
    display(Slider)
```



```
[12]: Slider.value # getting the value of the slider

[12]: 42

[14]: Slider.value = 100 # setting the value of the slider

[15]: Slider.keys

[15]: ['_dom_classes',
      '_model_module',
      '_model_module_version',
      '_model_name']
```

Figure C3a: Code block showing ipywidgets code

The module of IPyWidgets provides tools to construct comprehensive user interfaces with Python. The widget is an object showing front-end information and linking back-end. The UI layout can be constructed using VBox, HBox function. By arranging the VBox widget and HBox widget, we can arrange a complicated layout format like Figure C3b

```
# 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)
```

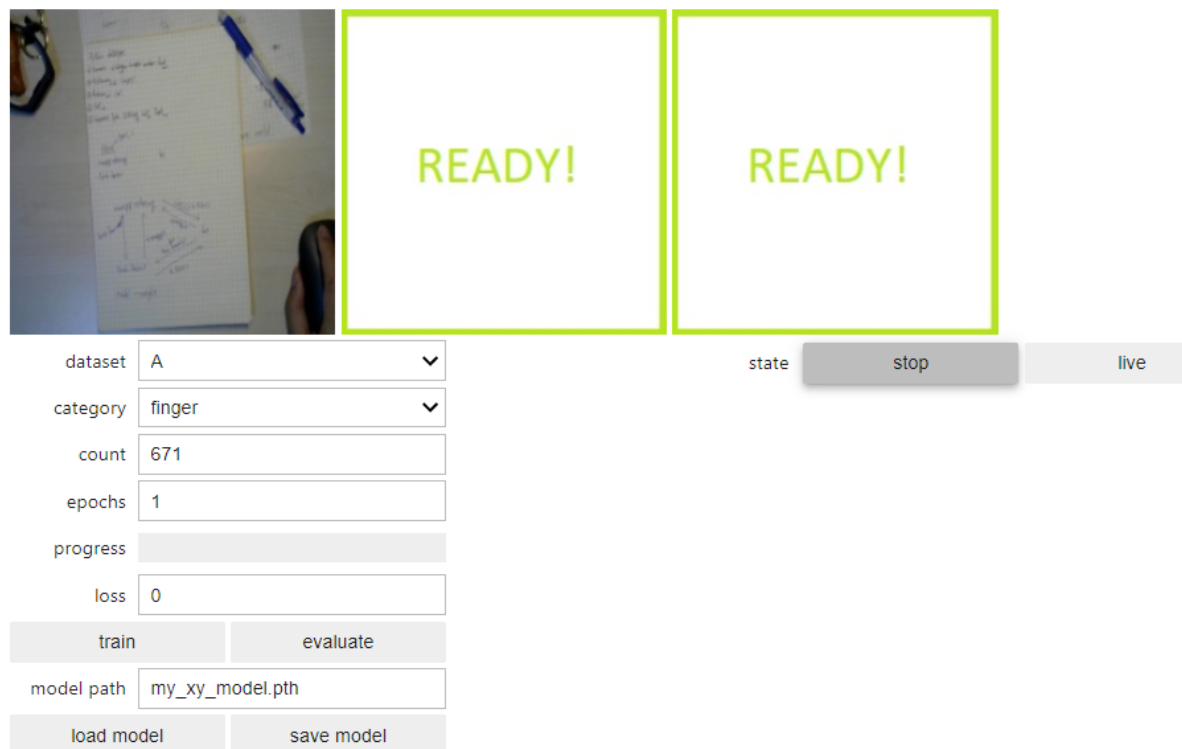


Figure C3a: Code block showing complicated layout

We will use this as the building block of the program.

C4. Transfer Learning

Learned in the EIE4100, I understand that the basic knowledge of Artificial Neural Networks. However, when it comes to the neural network of image. Traditional Artificial Neural Networks may lose the spatial information of the image. Therefore, a new Neural Network architecture is used for image processing. Convolutional Neural Networks are used. Since the operation of the pixel is spatial and in matrix form, performing convolution will preserve the spatial information of the image. After a lot of operations like convolution and max-pooling of numerous layers, we can flatten the vector and connect it with fully connected layers (some calls FC). The convolution and pooling layers can be designed differently. In this project and the following task, I will be using ResNet-18. In the fully connected layers, it is crucial for doing transfer learning.

Transfer Learning is used in this project. Transfer learning means that to transfer a trained model into a customized output by replacing the last layer into the self-designed neural network. The pre-trained network used within this project will be ResNet-18. It is the smallest size of the network of ResNet – Residual Network. The residual network solves the problem that the old model is not good at where the network can perform. The pre-trained ResNet-18 in PyTorch [10] is an excellent solution to this question. The model can be loaded with the following code.

```
>> import torch
```

```
>> import torchvision
```

```
>> model = torchvision.models.resnet18(pretrained = True)
```

```
>> model.fc = torch.nn.Linear(512, len(dataset.categories))
```

The code uses the PyTorch module, and thus, in the beginning, we need to import the module to python code. The first line of code is to load the pre-trained ResNet-18 model with pre-trained weight. Note that we need to set the input parameter 'pre-trained' to be 'true'.

The summary of the Pretrained ResNet 18 is as following:

```
>> # RESNET 18
>> model = torchvision.models.resnet18(pretrained=True)
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)
    )
  )
)
```

That weight is loaded to the "model" class. Initially, the model has a total of 1000 output with 512 input, dimension of (512, 1000). For the transfer learning technique, we replace another fully connected (fc) layer with the original layer and re-trained it with our custom datasets. The second line of code states that the fully connected layer is re-configured to a linear layer with 512 input with customized output labels. In the following test case, I have tested it with a simple thumbs up and thumbs down the task. It consists of two datasets one thumbs_up and one thumbs_down, and the result is shown in figure 4a

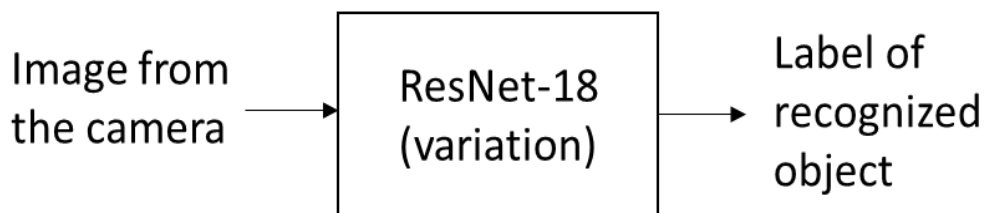


Figure 4a shows the block diagram of the function of object recognition.

Jetson Nano can transfer and transform the existing ResNet-18 network into a customized one to do object recognition. The most time-consuming work is to prepare a set of good quality of data. Once the data is ready, we can train the neural network with ten epochs. After training the network, we can test it. If the accuracy of the result is not high enough, we can add more photos sample to a different background. Just like the photo shown in Figure 4b. Using different backgrounds with the same object can tell the neural network that the background is irrelevant to the object. It can improve the accuracy of the network.

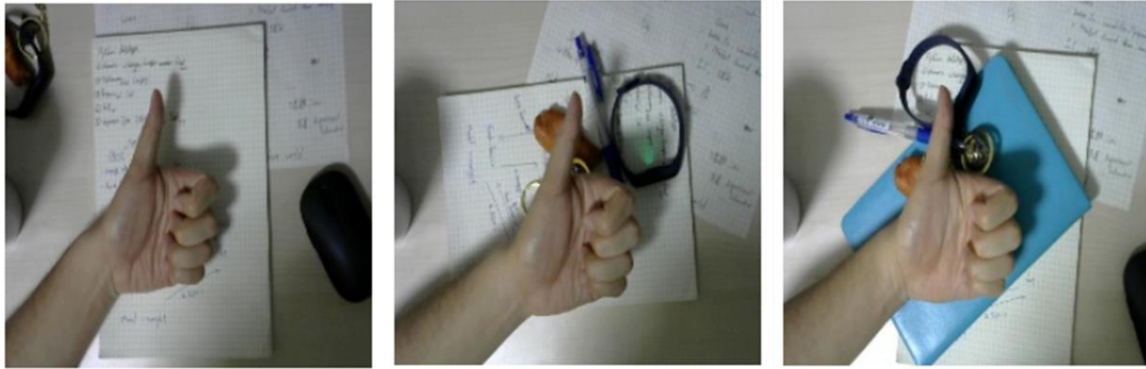


Figure 4b shows the object with different background

The softmax function used in the classification problem is to build a probability distribution of the output values.

```
>> import torch.nn.functional as Func
```

```
>> output = model(preprocessed)
```

```
>> output = Func.softmax(output,dim = 1).detach().cpu().numpy().flatten()
```

```
>> ocategory_index = output.argmax()
```

```
>> prediction_widget.value = dataset.categories[category_index]
```

The softmax function is also known as normalized exponential function. Its job is to normalize the output to a range (0,1), and the sum is 1 as well. It turns the non-normalized output into probabilities distribution. The greater the value is, the higher the probabilities it has to be classified into the class. The formula of Softmax

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

C5. Regression Model

Image Classification is dealing with some object which has discrete output or different class. The regression model is needed to handle with continuous output, such as (x,y) coordinates. Using also with the pre-trained model with ResNet-18, the output layer of the regression model is twice the amount of the classification since this has to deal with x, y coordinate of an object. As we would like to track the finger of the image, the classification model is not suitable because the result we want to obtain is the coordinate, which is continuous outputs. Classification applications can run only due to the problem with discrete outputs. Therefore, to track the x, y coordinate of the finger, we need to use the regression model. Figure C5a shows that the fully connected network has two output, one for x coordinate, and the other one is for y coordinate. The regression model does not require a Softmax layer as well. Jetson Nano is capable of performing the finger tracking problem. With the coordinate of your finger obtained on the image, we can achieve the object recognition to let the AI assist us in seeing a thing.

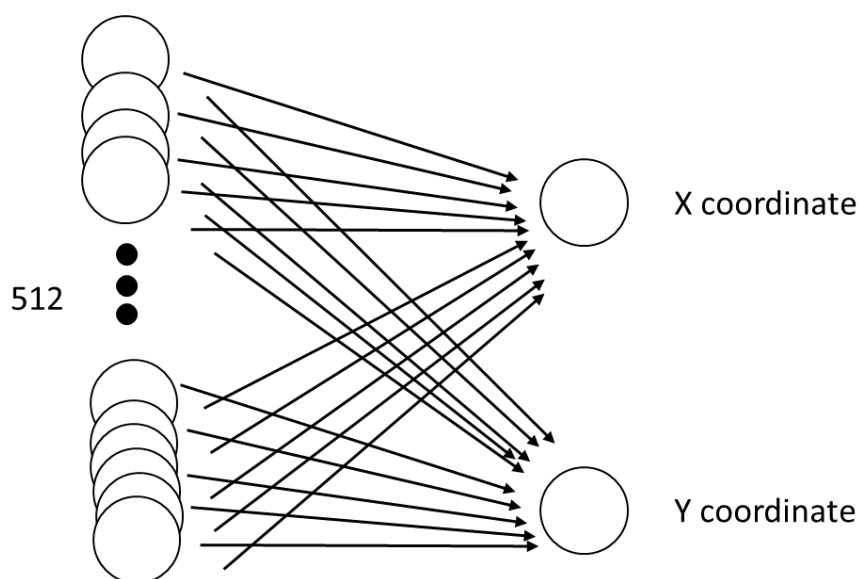


Figure C5a. Showing the fully connected layer of the regression model

C6. Handling the model

A model in a neural network consists of learnable parameters, we can check the value of the node using the following code.

```
>> import torch

>> model_state_dict = torch.load("./myModelPath.pth")

>> model_state_dict['fc.weight']

>> model_state_dict['fc.weight'].size()

>> model = load_state_dict(model_state_dict)

>> model.eval()

>> torch.save(model.state_dict(), "./myModelPath.pth")
```

Since the model we saved is using the state_dict mode. It means that it saves in the form of Python Dictionary. By loading the weight to an object named model_state_dict using torch.load('./xxxx.pth'), we can access a specific layer by passing in the variable 'fc.weight' to the list model_state_dict. We can also check the size of the layer using the method called .size(). The size of the fully-connected layer with tracking finger is torch.Size([2, 512]). 2 is the output layer, and 512 is the input layer. The Python dictionary is then loaded to become a model. After that, make sure we evaluate the model with model.eval().

The output of the >> 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')
```

It is important to note that in the object `torch.Tensor()` has an element called `device`, and the device is assigned to `'cuda:0'`. It is a special optimization for Nvidia products. CUDA means Compute Unified Device Architecture, and it was developed by Nvidia. It helps speeding up the calculation of floating-point numbers and tensor calculations. It is recommended to enable CUDA when using Jetson nano. This line of code is used to convert PyTorch tensor into a CUDA tensor.

```
>> torch.Tensor(np_PTH_Quantization).cuda()
```

C7. Pruning and Sharing the weights using KMeans

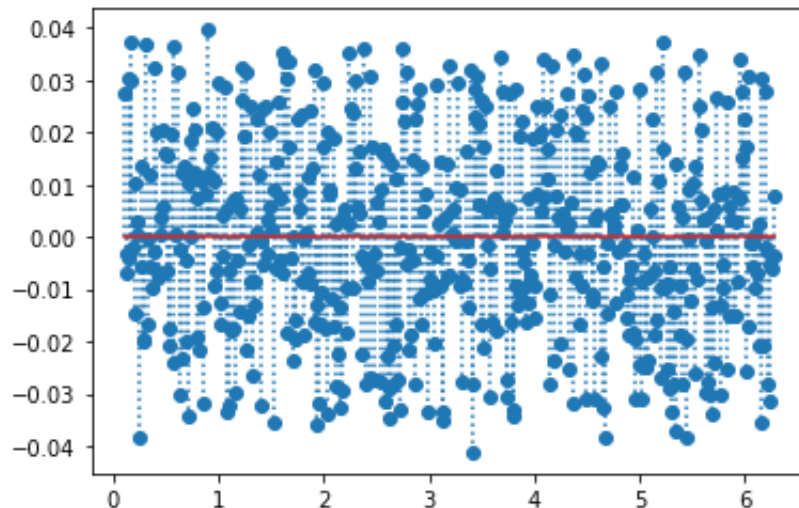
A model weights can be optimized by sharing weight of nodes. The `state_dict` is in the form of the Python Dictionary. First, we plot the value of all the nodes using the python module `matplotlib`. I am going to plot this graph with a STEM plot to show the position pattern of the weight of those nodes.

As we need to do the STEM plot with the weight, we need two python modules, `matplotlib.pyplot`, and `NumPy`. First, as a usual plotting procedure, we define the `x` as line space with 512 precision since the total number of weight for each node is 512.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0.1, 2 * np.pi, 512)
for i in range(len(PTH)):
    plt.stem(x, PTH[i], use_line_collection=True, linefmt=':')
    plt.show()
```

FigureC7a: Code block showing stem plot



FigureC7b: Stem plot of a single node

This plot shows the value of 512 weights. We can observe that some of the values are closed to each other. We can find some mean value to represent those weight and prune those close-value weight and sharing with a mean value. Machine learning techniques can help.

The method used is called K-Means Clustering.

```
>> from sklearn.cluster import KMeans

>> km = KMeans()

>> km.fit(np.array(PTH[0]).reshape(-1,1))
```

The code shows the way to do KMeans Clustering. The Python Module we use is scikit-learn, and it is a typical machine learning python package. It is built on top of NumPy and Scipy. So the input parameters are mostly with NumPy array but not python list. After importing the Python module KMeans, we construct an object called km using the method called KMeans(). Then, we use a method fit to start the

clustering. Noted that the input array is needed to be reshaped as range -1 to 1. Two results are impressive. One is the clustering result, and the second is the mean value of each of the clustering results.

```
>> km.labels
```

```
>> km.cluster_centers_
```

```
[array([2, 2, 2..., 2, 1, 0], dtype=int32),
 array([2, 1, 2..., 2, 2, 0], dtype=int32)]
[array([-0.02606429], [-0.00019427], [ 0.02478429]]),
 array([-0.00015555], [-0.02869738], [ 0.02799711]])]
```

Km.labels are in array form labeling each of the weight to the corresponding cluster, in the code block above, the KMean clustering is set to be three, which has cluster 0, cluster 1 and cluster 2. The km.cluster_centers shows the corresponding center point of the weight, in the case of grouping network weight, the mean value can be use represent the related value of the weight.

After acquiring the value from KMean Clustering, they will plot together with the original stem plot for observation.

```
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()
```

Figure C7c: Code block showing plotting stem plot with center

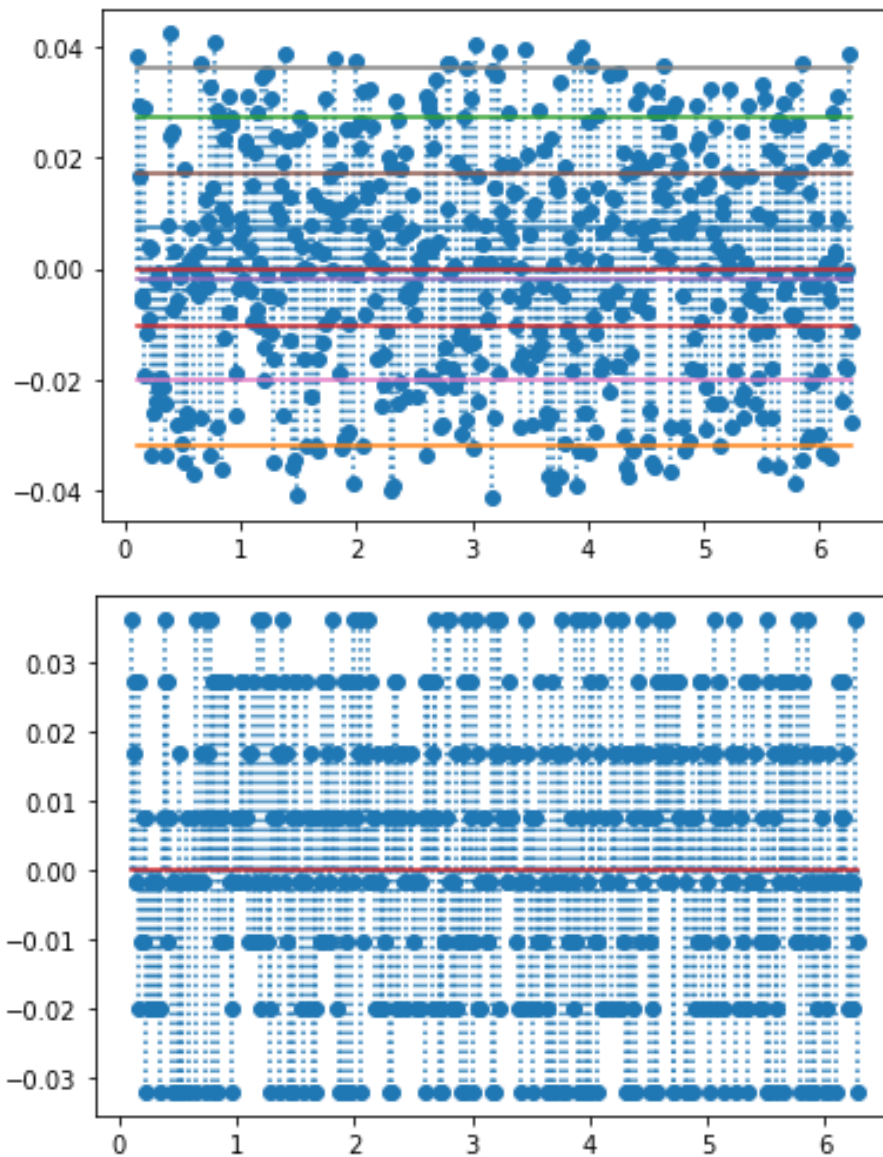


Figure C7d: showing two plotting stem plot with center

It helps us understand the quantification process. Next, we need to reassign those values to its clustering center. The following code block is to assign the value to its labeled cluster center value.

```

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]

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])

```

The code used three nested for loop to assign the value one by one. After assigning the values, we can visualize the result of weight with a stem plot again.

After quantifying the model, we can then feed the quantifying weight back to the model and observe the result.

Python has a lot of different packages and its object, it is not compatible across different python modules, so to better organize the relationship between different python modules(python list, NumPy array, and PyTorch tensor relationship), I have created a diagram to show it.

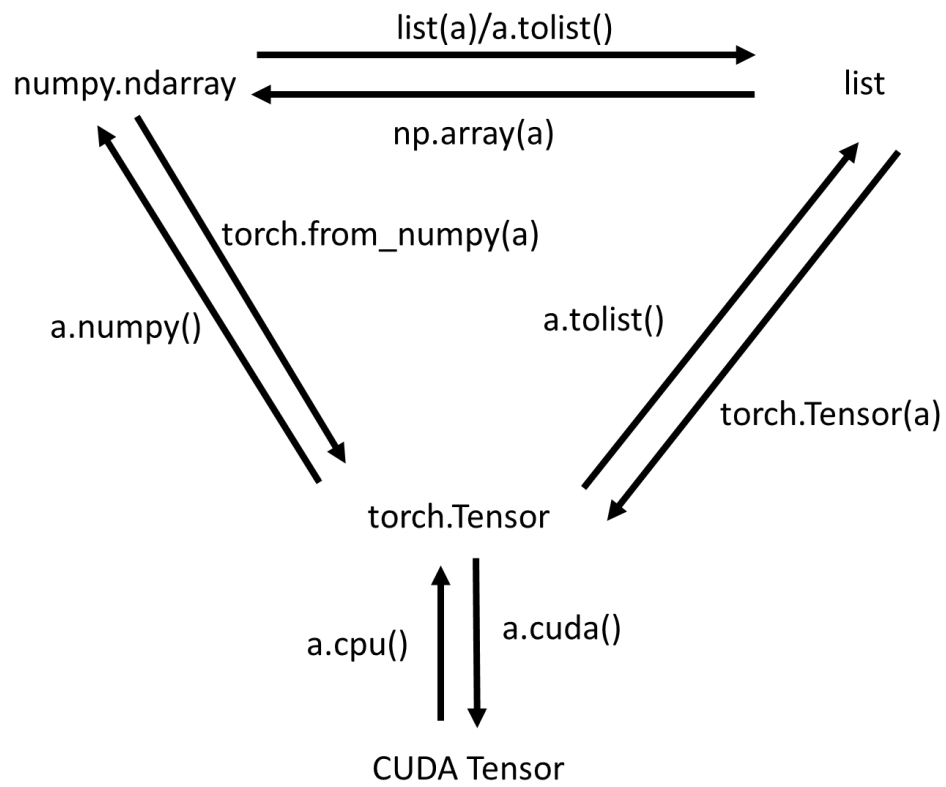
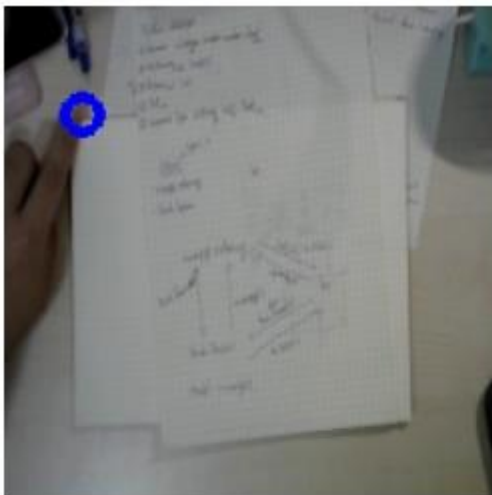


Figure C7f: a diagram showing a conversion of different list and array

C8. Using Model for application development

The PyTorch model trained in the python program can be called using the following code, and we can think the model as a function to call it. As the program code shows, the image is obtained using 'camera.value', and then it fed into a preprocessing state. After that, it e into to model, and after the computation of the neural network, we detach the output and transfer to CPU and back to NumPy format. And finally, we flatten it and assign it back to an object called output. The x and y coordinates are in the object output.



```
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

Figure8a: Code block showing how to use the model

C9. Audio system

The final step of the AI system will be the voice system. Once Jetson Nano knows the position of your finger and also the object you are pointing at, Jetson Nano will read out the label they obtain. The text-to-speech system will adapt to the existing gTTS library to do that job. The text to speech python module used is TTS. It adopts the text-to-speech service of Google, which code is simple and as the following.

```
from gtts import gTTS
import os

mytext = ''
language = 'en'

myTTS = gTTS(text=mytext, lang=language, slow=False)
myTTS.save("object.mp3")
os.system("mpg321 object.mp3")
```

In the program, import the python module gTTS and called construction function with text and specifying the language. After that, save as mp3 file and play the file with `os.system("mpg321 object.mp3")`. Jetson Nano does not have a headphone jack, and hence a USB sound card is added to the board.



Figure9a: Showing the USB adaptor

C10. Hardware

Nvidia Jetson Nano (Figure C9a) is selected in this project because there are plenty of support documents on the internet. The size is not too large, just 70x45mm. It is the smallest Jetson device in Nvidia. It acquires 128-core Maxwell GPU and Quad-core ARM A57 @ 1.43 GHz CPU, and thus, it has enough computation power to act as the interface of deep learning. Using a Linux-powered device can speed up the development process. Image from the Nvidia also include some useful deep learning package for example pytorch, matplotlib etc. Nvidia also provide python module JetCam, which provide useful API for developer to connect with the camera via simple code. The connectivity is plenty that it has one HDMI type A port, one DP port, four USB 3.0 and speaker can then hook up with using the USB interface for testing. It got expansion header for developer to hook up with some low-level communication such as IIC/SPI communication. It is important for electronic developer since some of the develop board only support low-level communication.



Figure C10a: Picture of Jetson Nano

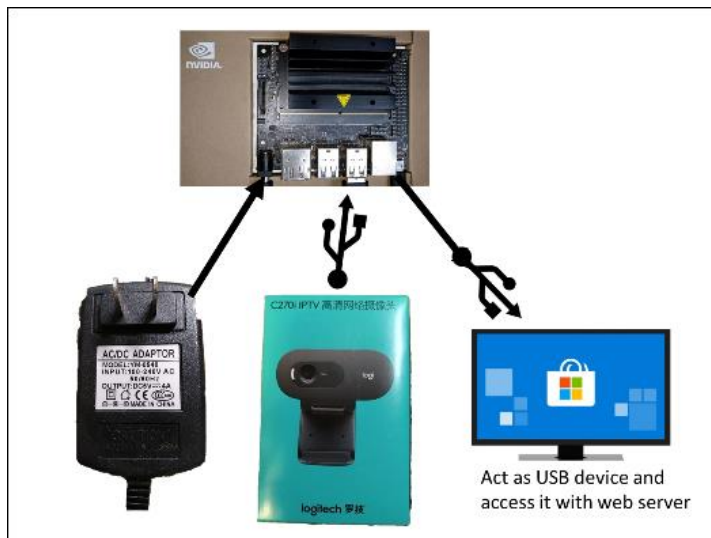


Figure C10b: Picture of connection diagram

The development environment on Nvidia Jetson is smart and convenient. First, by flashing the OS with Jupyter Lab webserver to the Jetson Nano, the only thing the developer does is connect the Jetson Nano as a USB device and access it with a web browser. This method saves time for switching between screen monitor or keyboard mouse. For a python application, the developer can focus on developing the code and algorithms rather than the environment.

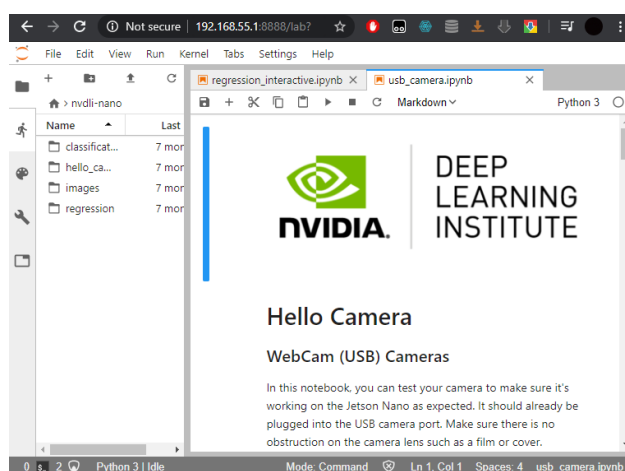
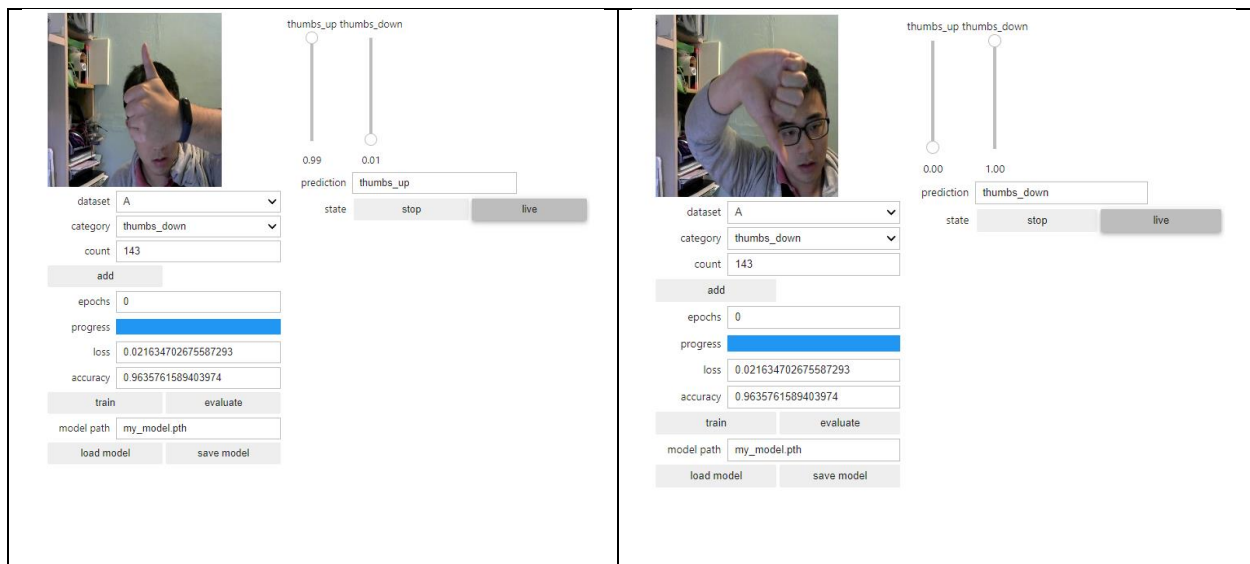


Figure C10c: Jupyter Lab development environment

D. Findings/Results

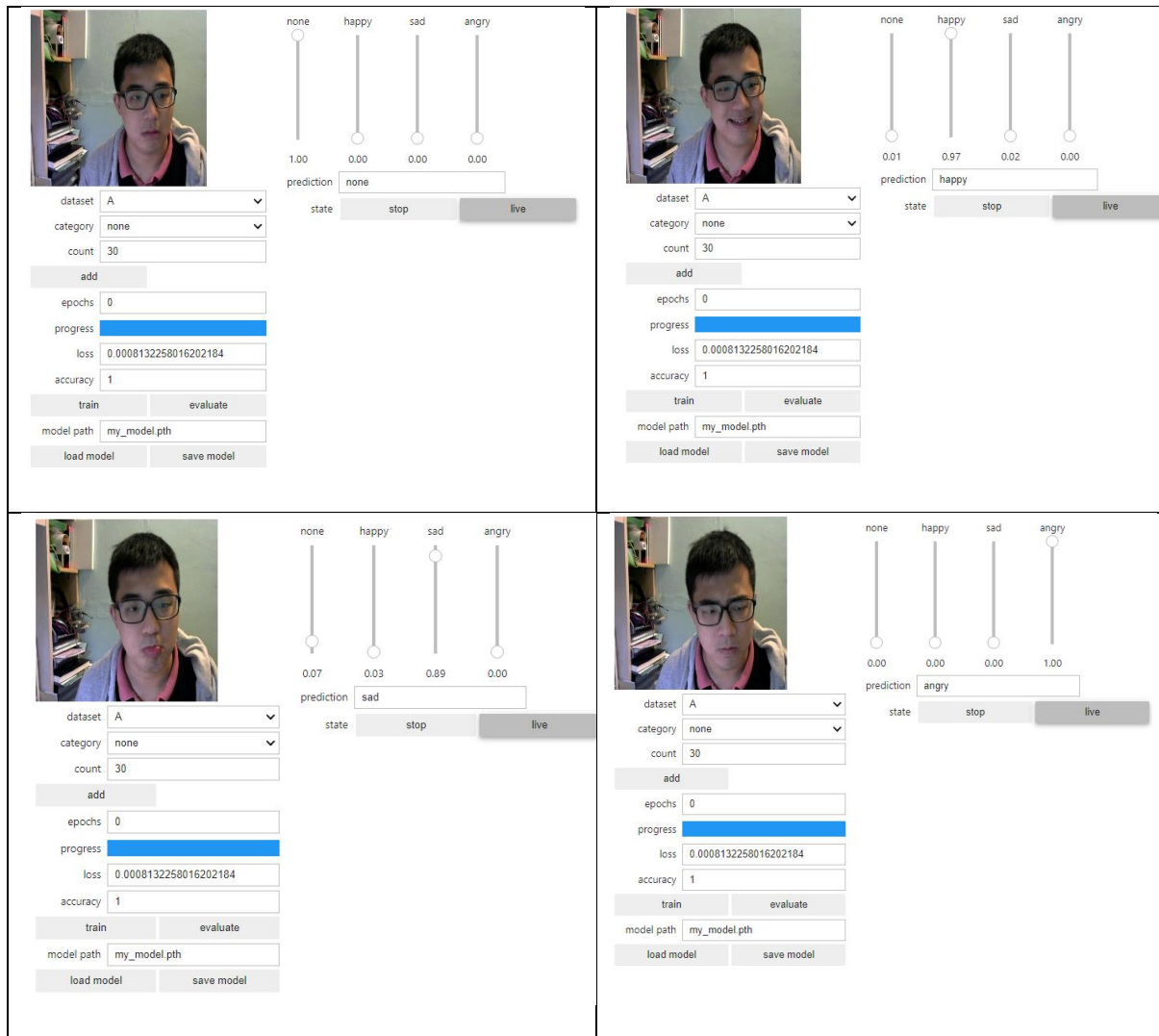
D1. Classification

The following is the result and the finding for Jetson Nano running a classification application.



The SoftMax layer converts the final output into probability showing that how possibly the object be. The probability is shown in the slide on the right top side. This testing shows that with the help of transfer learning, we can perform a classification problem with few samples. The model can perform well in classifying whether my thumb is going up or going down.

To test more and to be more familiar with the development environment, I have done one more tiny task to check out that if the network can recognize my facial expression. As ResNet-18 is powerful in processing images, it is surprising that the pre-trained model with transfer learning can determine the facial expression.



D2. Regression

While for the regression problem like tracking the finger, the actual values are needed to be kept to get the actual X and Y coordinate values. Therefore, the output dimension is twice the amount of the object. In this case, we need to track the fingertips, so, 2 output node is required. I complete the task with few data gathered. And the result is not quite accurate. The blue circle may still deviate from my actual fingertips with few data samples gathered.

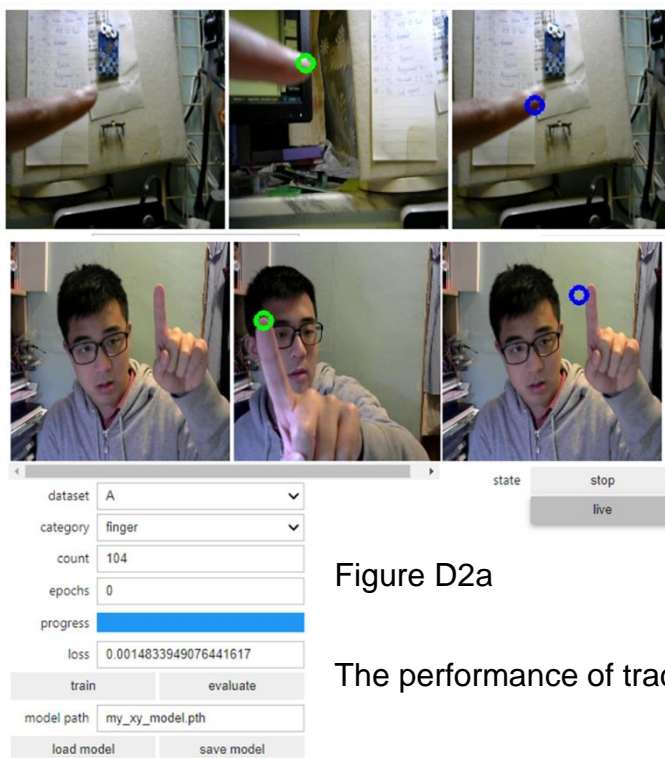


Figure D2a

The performance of tracking finger with few sample

The task above shows the possibility for the Jetson Nano to perform the finger tracking problem. With the coordinate of your finger obtained on the image, we can achieve the object recognition to let the AI assist us in seeing a thing.

D3. Pruning and Sharing the weights using KMeans

The model we use to test is name TF_800_2.pth, with a size of 46.6MB. It is already a large file for some of the microcontroller. When the model is loaded into Jetson Nano, it required a while

A preview of the trained network looks like:

```
model_state_dict = torch.load("./TF_800_2.pth")
# model_state_dict

model_state_dict['fc.weight'].size()

torch.Size([2, 512])

fc_wright_model_state_dict = model_state_dict['fc.weight']
fc_wright_model_state_dict

tensor([[[-0.0250, -0.0053,  0.0027, ..., -0.0002, -0.0132,  0.0163],
        [-0.0054,  0.0138,  0.0174, ..., -0.0001, -0.0036,  0.0241]],
        device='cuda:0'])

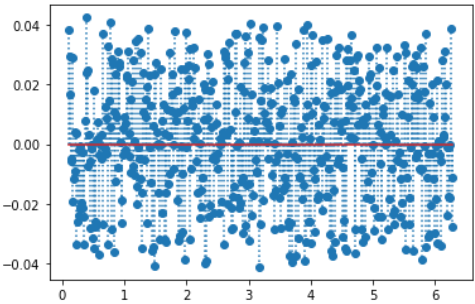
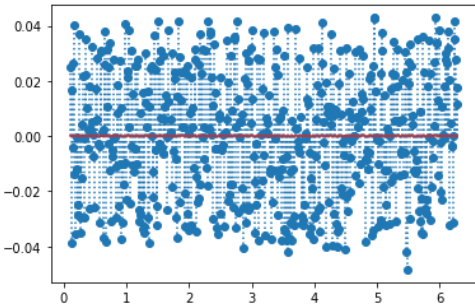
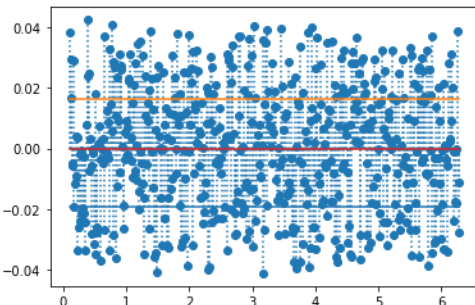
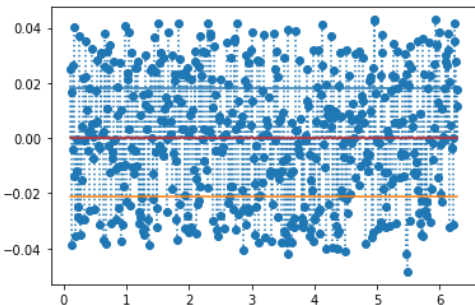
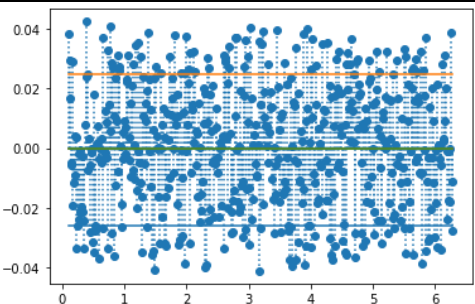
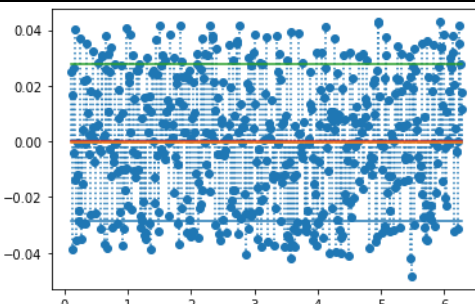
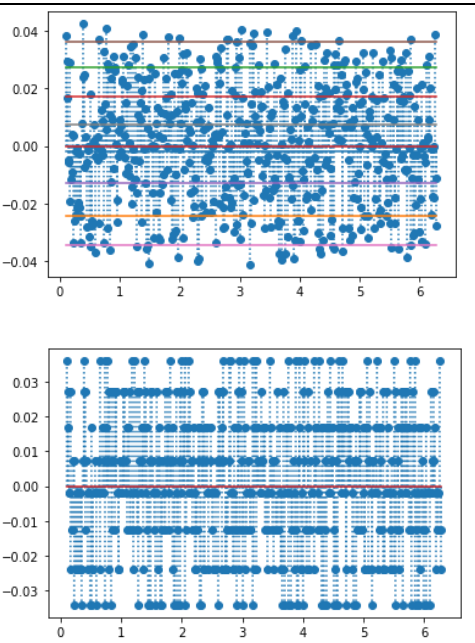
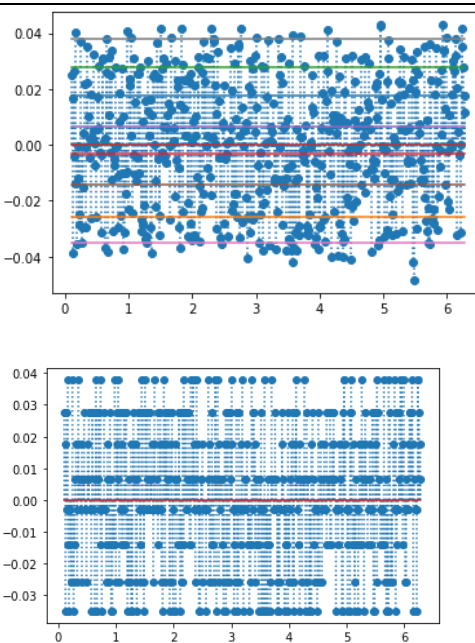
model_state_dict['fc.bias']

tensor([ 0.0221, -0.0295], device='cuda:0')
```

FigureD3a: code block showing tensor state dict

The tensor has an attribute device, and its value is cuda:0, it means that the tensor is already optimized for running on CUDA. When we would like to handle the element of the tensor, we need to first load back to the CPU with method `>> .cpu()`

First, we read and list out the weight of each node. We can plot a stem plot to visualize the value pattern of the weight

	First Node	Second Node
Untouch		
KMeans=2		
KMeans=3		
KMeans=8		

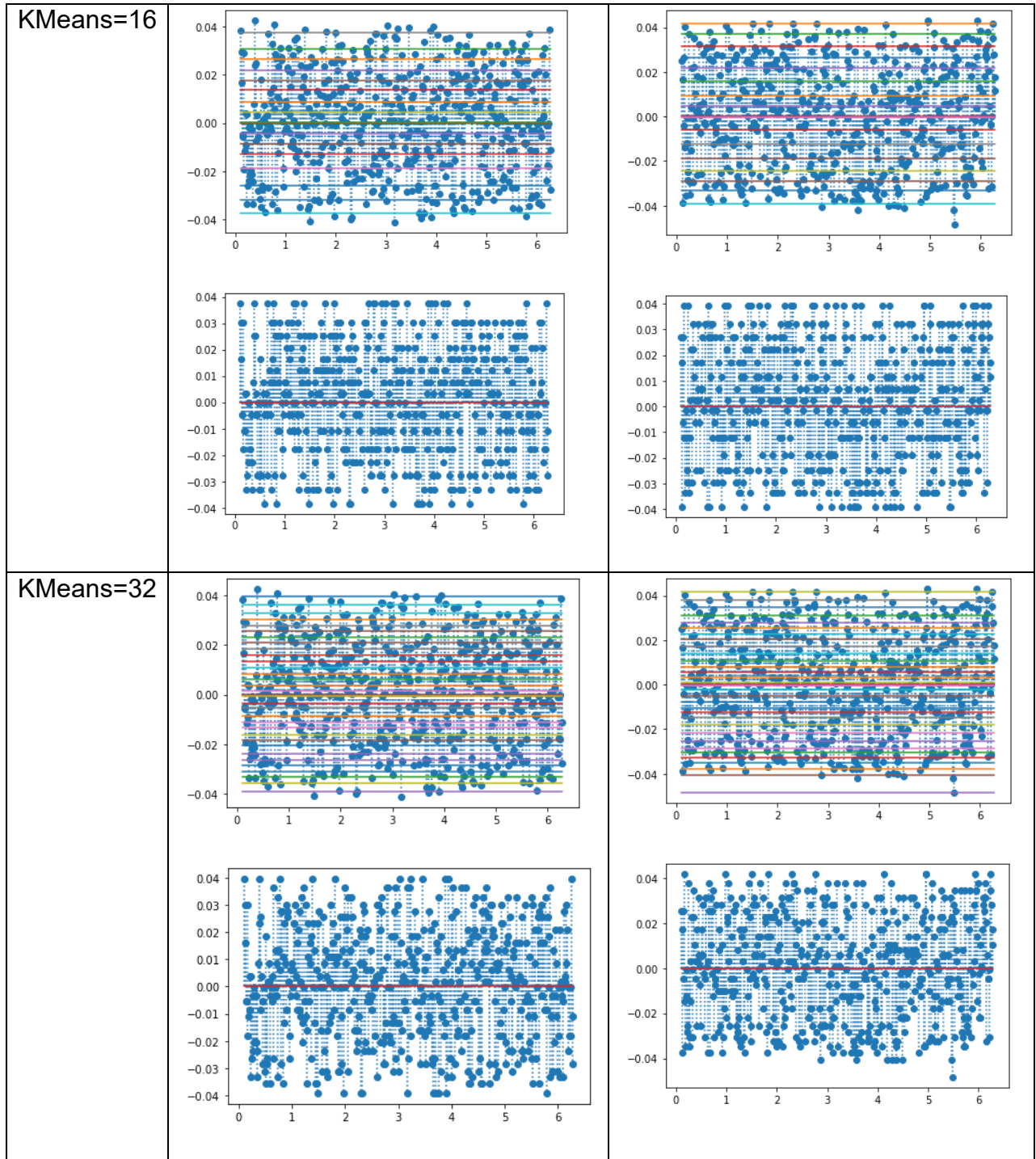



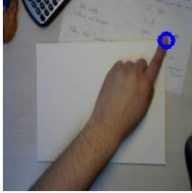


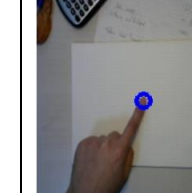


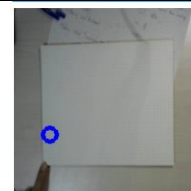
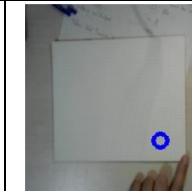
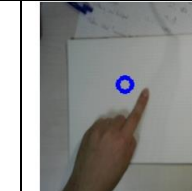



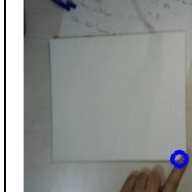
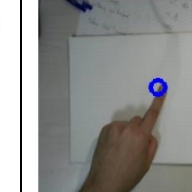
Table 1: Stem plot of different KMeans for first and second node

As attached in the table, with the comparison with the untouched version, it is clearly seen that when generally grouping weight into two groups as increasing the KMean clustering group, the quantization version of the stem plot looks similar with the untouched model. With each of the pruned models, I will reload it back to model and

run that model to perform finger tracking problems also and see how the pruning affects the performance. The weight in the final layer is used to calculate the position of the x,y coordinate.

D4. Pruning with quantization

I have performed the KMean clustering with 5 different cases, I completed one with untouched model weight value for comparison, and I am going to group the weight value from the fully connected layer to 2,3,8,16 group and observe the behavior it has.

Untouch					
KMean =2					
<pre>model_state_dict['fc.weight']</pre> <pre>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')</pre>					
KMean =3					
<pre>model_state_dict['fc.weight']</pre> <pre>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')</pre>					

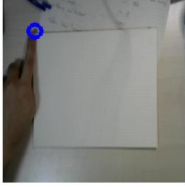




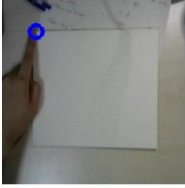

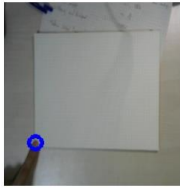

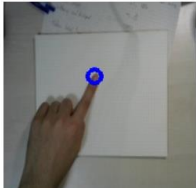
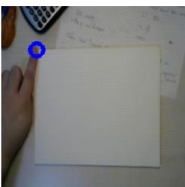
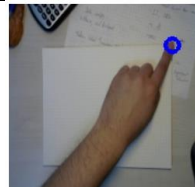

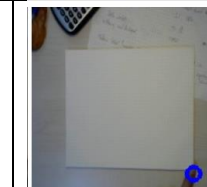
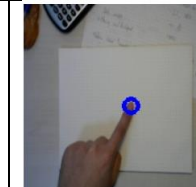


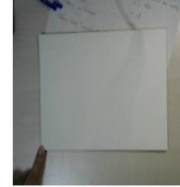
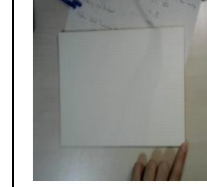
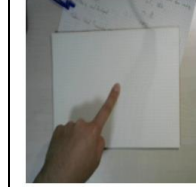
KMean =8					
KMean =16					

Table 2: Performance of 5 different KMean clusters with quantization

As we can see that the fewer the quantization is, the poor the performance it is, the tracking point cannot meet and track my fingertips. With increasing the cluster and the quantization level, the performance is getting better and better. In this case, setting KMean clustering into 16 clusters perform the best in this experiment.

D5. Pruning with close-to-zero values

Next, I would like to prune model with some small significant effect weight. From all the clustering centers, some of the neurons have a value that very close to zero. I will try to prune the network by removing the neuron, which has a very small effect on the overall performance.

Untouch					
KMean =2					



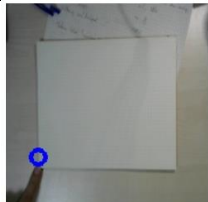

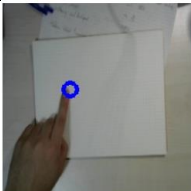

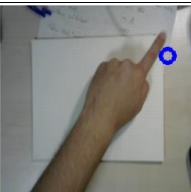


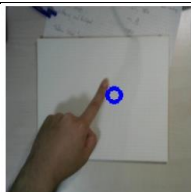


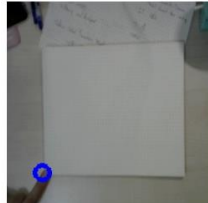


<pre>[25]: model_state_dict['fc.weight']</pre> <pre>[25]: tensor([[0.0000, 0.0000, 0.0000, ..., 0.0000, -0.0193, -0.0193], [0.0000, -0.0213, 0.0000, ..., 0.0000, 0.0000, 0.0000]], device='cuda:0')</pre>					
KMean =3					
KMean =8					
KMean =16					

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

When clustering into two groups (KMean=2), removing one with the smallest value means eliminating half of the node, the model corrupts and do not perform. Starting from KMean=3 and more, the effect is not significant when compared with the one with the smallest value. As mentioned, the last layer is used to calculate the (x,y) coordinate of the finger. Hence, the zero-closing value is not contributing to updating the position of the finger. Taking KMean=16 as an example, in the cases with the smallest value included in the network, the smallest value is $[-0.00041571]$ which is very close to zero and insignificant. Therefore, it is safe to eliminate it in the model.

E. Discussion

The computation power of Jetson Nano is enough for doing on-device deep learning. As shown in the Result session, Jetson Nano is capable of running a ResNet-18 model, and with the transfer learning technique, it can do the custom model object recognition and regression. Transfer learning provides developers a way to use the pre-trained model for a building block to develop with.

For an on-device deep learning machine, the memory and the resource on the device are not enough. Therefore, model compression is essential in this case. It has lots of ways to compress the model. One of the methods is to quantize the model weight into particular value and to store the labels of that model and assign it in the program. It can balance between the performance and storage resources. The way to perform the clustering is to adapt the KMean Clustering algorithm with python module scikit-learn. The result shows that using machine learning to find the clustering center and quantize the weight do not have a lot of effect on the existing model. Also, eliminating the smallest value of the model can help pruning the model into a smaller one.

This project shows the feasibility to run classification, regression model, and machine learning algorithm on an SBC, and evaluate the performance of the model-compression.

F. Recommendations and Conclusion

Jetson Nano is too bulky and having too many heat dissipations. If it is targeting the development of some small robot, it may be suitable for it to run some AI model on it. However, when it comes to wearable devices, Jetson Nano may not be the best solution. Upon the time of doing this project, some chip that is small but powerful to AI was available. It is more suitable for wearable devices. It uses an open-source architecture called RISC-V. The Chip called K210, which is developed by Kendryte. Sipeed has wrap K210 into a module such that it is easy to use [11]. The power of the Jetson Nano required 5V 4A, which is not easily accessible, and the heat dissipation metal is very hot after turning on the script for a while.

The advantage of using Jetson Nano is that it is a Nvidia product with a CUDA architecture, which can optimize the deep learning running on-device. It has an easy-to-use JupyterLab development environment. It is easy to remote the Jetson Nano by a USB cable.

The model compression I perform only consist of the final layer of the model and It is just a slice of the model. The compression technique should cover the whole model in order to maximize the model compression result. It is recommended to use some of the tools to assist in optimizing the model. Nvidia developed tools called TensorRT to optimize the deep learning model, which improves the efficiency of on-device deep learning.

Nvidia TensorRT is a Software development kit for a high-performance deep learning interface. It provides developer interface to do the quantization of model into int8 (Quantization and Binarization), fusing nodes within a kernel (Pruning and Sharing). That technique is needed to do the model compression.

To conclude, I have successfully developed an on-device object recognition system, an on-device regression system, and a model optimizer. It proves the feasibility of using on-device deep learning for developing a product. I have evaluated the importance of model compression for deep learning models.

G. 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," arXiv preprint arXiv:1710.09282., Feb. 2019.
- [7] Raina, R., Battle, A., Lee, H., Packer, B., & Ng, A. Y. (2007, June). Self-taught learning: transfer learning from unlabeled data. In Proceedings of the 24th international conference on Machine learning (pp. 759-766).
- [8] Plawrence,NVIDIA, "User Guide" JETSON NANO DEVELOPER KIT, May. 2016 [Revised May. 2018].
- [9] Jupyter Organization "ipywidgets Documentation", 2019
- [10] Eli Stevens, Luca Antiga "Deep Learning with PyTorch" , 2019

[end] Nair, D., Pakdaman, A., & Plöger, P. G. (2020). Performance Evaluation of Low-Cost Machine Vision Cameras for Image-Based Grasp Verification. arXiv preprint arXiv:2003.10167.

H. Appendix

Program code on Pruning and Sharing model

```

from sklearn.cluster import KMeans
import numpy as np

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_)

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]
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])

```

Program code on Transfer Learning

```
# Model
```

```
import torch
import torchvision

device = torch.device('cuda')
output_dim = 2 * len(dataset.categories) # x, y coordinate for each
category
```

```
# RESNET 18
```

```
model = torchvision.models.resnet18(pretrained=True)
model.fc = torch.nn.Linear(512, output_dim)
```

```
model = model.to(device)
```

```
model_save_button = ipywidgets.Button(description='save model')
model_load_button = ipywidgets.Button(description='load model')
model_path_widget = ipywidgets.Text(description='model path',
value='my_xy_model.pth')
```

```
def load_model(c):
    model.load_state_dict(torch.load(model_path_widget.value))
model_load_button.on_click(load_model)
```

```
def save_model(c):
    torch.save(model.state_dict(), model_path_widget.value)
model_save_button.on_click(save_model)
```

```
model_save_button.click()
```

```
model_widget = ipywidgets.VBox([
    model_path_widget,
    ipywidgets.HBox([model_load_button, model_save_button])
])
```

```
# display(model_widget)
print("model configured and model_widget created")
```

```

# Live Execution

import threading
import time
from utils import preprocess
import torch.nn.functional as F

state_widget = ipywidgets.ToggleButtons(options=['stop', 'live'],
description='state', value='stop')
with open("../images/ready_img.jpg", "rb") as file:
    default_image = file.read()
prediction_widget = ipywidgets.Image(format='jpeg',
width=camera.width, height=camera.height, value=default_image)

def live(state_widget, model, camera, prediction_widget):
    global dataset
    while state_widget.value == 'live':
        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))

        prediction = image.copy()
        prediction = cv2.circle(prediction, (x, y), 8, (255, 0, 0),
3)
        prediction_widget.value = bgr8_to_jpeg(prediction)

def start_live(change):
    if change['new'] == 'live':
        execute_thread = threading.Thread(target=live,
args=(state_widget, model, camera, prediction_widget))
        execute_thread.start()

state_widget.observe(start_live, names='value')

live_execution_widget = ipywidgets.VBox([
    prediction_widget,
    state_widget
])

# display(live_execution_widget)
print("live_execution_widget created")

```

```

# Training and Evaluation

BATCH_SIZE = 16
optimizer = torch.optim.Adam(model.parameters())
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-3,
momentum=0.9)
epochs_widget = ipywidgets.IntText(description='epochs', value=1)
eval_button = ipywidgets.Button(description='evaluate')
train_button = ipywidgets.Button(description='train')
loss_widget = ipywidgets.FloatText(description='loss')
progress_widget = ipywidgets.FloatProgress(min=0.0, max=1.0,
description='progress')

def train_eval(is_training):
    global BATCH_SIZE, LEARNING_RATE, MOMENTUM, model, dataset,
optimizer, eval_button, train_button, accuracy_widget, loss_widget,
progress_widget, state_widget

    try:
        train_loader = torch.utils.data.DataLoader(
            dataset,
            batch_size=BATCH_SIZE,
            shuffle=True
        )
        state_widget.value = 'stop'
        train_button.disabled = True
        eval_button.disabled = True
        time.sleep(1)

        if is_training:
            model = model.train()
        else:
            model = model.eval()

        while epochs_widget.value > 0:
            i = 0
            sum_loss = 0.0
            error_count = 0.0
            for images, category_idx, xy in iter(train_loader):
                # send data to device
                images = images.to(device)
                xy = xy.to(device)

                if is_training:
                    # zero gradients of parameters
                    optimizer.zero_grad()

                # execute model to get outputs
                outputs = model(images)

```

```

        # compute MSE loss over x, y coordinates for
associated categories
        loss = 0.0
        for batch_idx, cat_idx in
enumerate(list(category_idx.flatten())):
            loss += torch.mean((outputs[batch_idx][2 *
cat_idx:2 * cat_idx+2] - xy[batch_idx])**2)
            loss /= len(category_idx)

        if is_training:
            # run backpropogation to accumulate gradients
            loss.backward()

            # step optimizer to adjust parameters
            optimizer.step()

        # increment progress
        count = len(category_idx.flatten())
        i += count
        sum_loss += float(loss)
        progress_widget.value = i / len(dataset)
        loss_widget.value = sum_loss / i

    if is_training:
        epochs_widget.value = epochs_widget.value - 1
    else:
        break
except e:
    pass
model = model.eval()

train_button.disabled = False
eval_button.disabled = False
state_widget.value = 'live'

train_button.on_click(lambda c: train_eval(is_training=True))
eval_button.on_click(lambda c: train_eval(is_training=False))

train_eval_widget = ipywidgets.VBox([
    epochs_widget,
    progress_widget,
    loss_widget,
    ipywidgets.HBox([train_button, eval_button])
])

# display(train_eval_widget)
print("trainer configured and train_eval_widget created")

```