The University of Hong Kong

ELEC6604 Neural Networks,
Fuzzy Systems and Genetic Algorithms

Report on Google Vision Kit

Lecturer: Dr. G. Pang

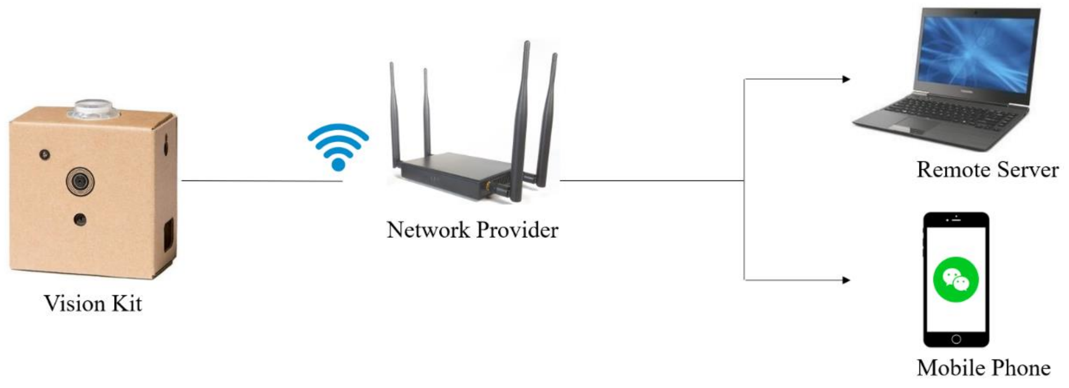
Students: *ZHAO Yunqing* *CHENG Ming*

29/03/2019 Hong Kong

Contents

Part 1. Communication Mechanism

1.1 Connection method of Kit



Graph 1 connection procedure of devices

The steps of the communication are as follows:

1. Firstly, the Google Vision Kit must be connected to a Wi-Fi network, so we can use SSH proxy to login it remotely using our PC.
2. Secondly, because the Vision Kit has no screen or other peripheral devices, it is difficult to see the calculating/prediction result of ANN model. To address this problem, we developed a platform that the Vision Kit could communicate with mobile phone by WeChat.

1.2 WeChat Platform Construction

The step **2**: WeChat Platform Construction:

(1) In the first step, once the Vision Kit starts working, we changed the **source document** inside the system, making it run the python script automatically, which can let us login in a WeChat account of a Web version. In this procedure, we transform the above-mentioned python script as a specified service when plug in the power of Google Vision Kit. Hence, the goal can be achieved.

(2) After login in the Web WeChat by using a famous python library called "itchat" (**https://pypi.org/project/itchat/**), we used and revised some related functions of itchat to let the WeChat Account (dedicatedly used for Vision Kit) continuously listen to our group Chatting-Room, then analyze and return the feedback of the received messages.

We set a series of standards for parsing the received messages. For example, once the message with a prefix of "**cmd**", then the other part of the message will be regarded as a Linux command by Kit, and our program could call OS module to execute this command.

(3) On the other hand, the self-started program has multiple threads and it can execute several parallel tasks. One of the threads is to shoot a photo if the button is pressed, then save the picture in a pointed file path. Another thread is to scan this file path continuously, and send the new picture in it to WeChat, to tell us what it has shoot and saved.

By this communication mechanism and the construction of the WeChat platform, we can easily control the Vision Kit and get different types of responses from it. It is helpful to our next tasks.

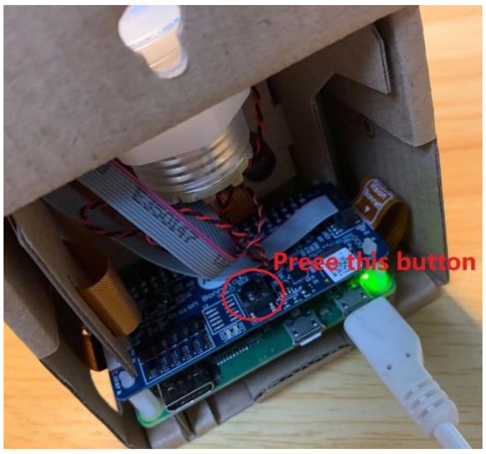
1.3 Detailed Communication Process Step by Step

1. Download the AIY Project app

As has been mentioned in the official website of Google Vision Kit, the AIY Project app can only be downloaded by android smart phones. Go to the **Google Play Store(Only available on Android Phones)** and download the `AIY Projects app`**.** This app will allow you to connect your Vision Kit to a Wi-Fi network, and display an IP address which you'll use to communicate with your Vision Kit wirelessly via a separate computer and SSH.

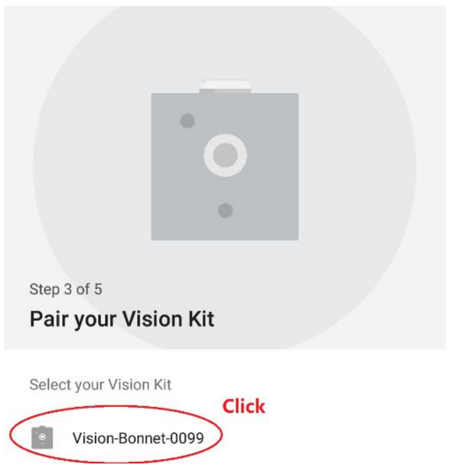
2. Start the Vision Kit and get its IP address

(1) Press the button on Vision Bonnet and Open the App on an Android cell phone.



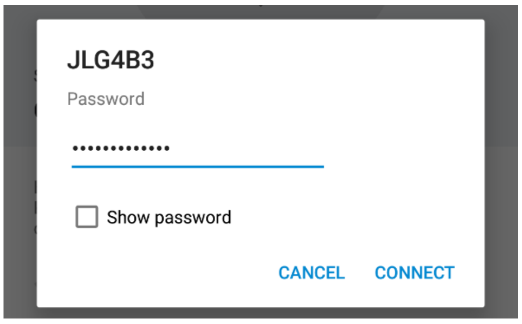
Graph 2 Button for Bluetooth Connection

Search AIY devices using blue-tooth, and pair the Vision Kit with your smart phone.



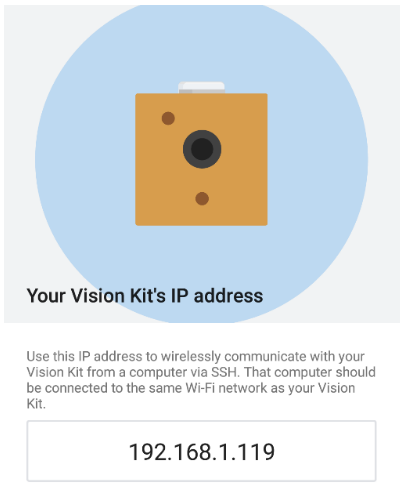
Graph 3 Find the ID of Vision Kit

Using the app to help Vision Kit to connect to a Wi-Fi network.



Graph 4 Input the password of local network

(2) Then, you will get the current IP address of the Vision Kit.


Your Vision Kit's IP address

Use this IP address to wirelessly communicate with your Vision Kit from a computer via SSH. That computer should be connected to the same Wi-Fi network as your Vision Kit.

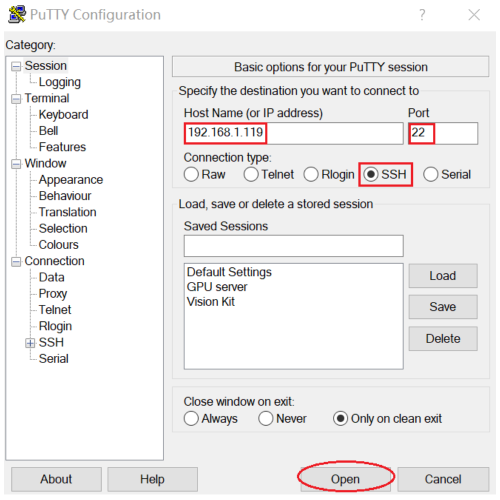
192.168.1.119

Graph 5 get the IP address of Vision Kit

Connect to the Vision Kit using SSH proxy

(1) Download the WinSCP from https://winscp.net/eng/download.php, **upload your files** simply. Then, download the SSH client from https://www.putty.org/, which can be used to **remotely connect to a Linux computer(Kit here)** using Wi-Fi network, command the Kit to run programs.

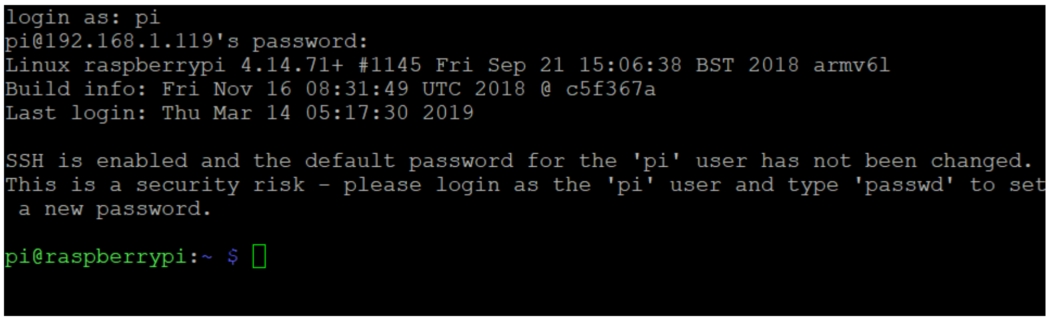
(2) Input the IP address of Vision Kit and choose 22 as the port number because of the requirement of SSH proxy. Finally, click the button of "Open".



Host Name (or IP address): 192.168.1.119
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

Graph 6 Putty Setting Interface

(3) Input the user account and pin number of Vision Kit, then you will login the Linux system successfully.(account name : ***pi***, default password: ***raspberry)***



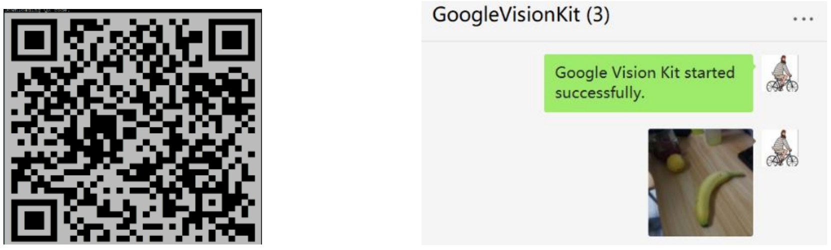
```
login as: pi
pi@192.168.1.119's password:
Linux raspberrypi 4.14.71+ #1145 Fri Sep 21 15:06:38 BST 2018 armv6l
Build info: Fri Nov 16 08:31:49 UTC 2018 @ c5f367a
Last login: Thu Mar 14 05:17:30 2019

SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set
 a new password.

pi@raspberrypi:~ $
```

Graph 7 Login into the Linux system

3. Run the programme

(1) Using the rules of Linux command line, type in ***"sudo python3 demo.py"*** to run the file named "demo.py".(under the ***home/pi***)

(2) Scan the QR code using your smart phone with WeChat, to verify the login. Then the communication by WeChat will be established.



GoogleVisionKit (3)

Google Vision Kit started successfully.

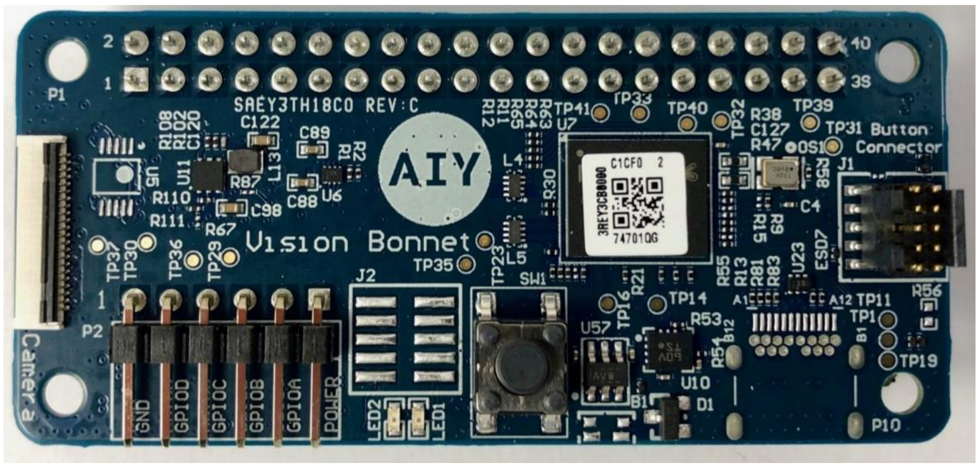
Graph 8 Establishment of the WeChat Platform

(3) Now, the built-in functions could use WeChat to send images and messages to us, which is a good way to display information(since no screen of Kit.)

Part 2. Processing of Recognition Platform

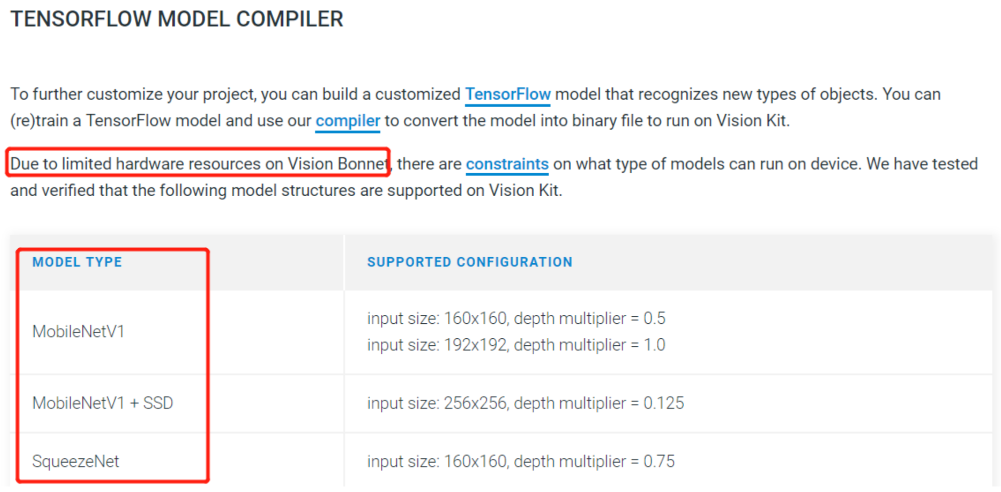
2.1 Official Guide from Google (*Unsuccessful*)

Since we want to build our own customized machine learning model, firstly we refer to the official guideline from the google, because we want to use the Bonnet (a part of hardware of Vision Kit) to accelerate the Machine Learning inference process by its TPU module.



Graph 9 Vision Bonnet

As the reference from official website of Google Vision Kit, (https://aiyprojects.withgoogle.com/vision/#makers-guide), we found that there are too many constraints, due to the **limited computational capability** of the Google Vision Kit, which means we can only use a very small number of open APIs released by google, on some specified network structures, e.g. MobileNet V1 and SqueezeNet, customized and provided by google. Meanwhile, we can only retrain the last fully connected network layer to make it recognize the things.



TENSORFLOW MODEL COMPILER

To further customize your project, you can build a customized TensorFlow model that recognizes new types of objects. You can (re)train a TensorFlow model and use our compiler to convert the model into binary file to run on Vision Kit.

Due to limited hardware resources on Vision Bonnet, there are constraints on what type of models can run on device. We have tested and verified that the following model structures are supported on Vision Kit.

MODEL TYPE	SUPPORTED CONFIGURATION
MobileNetV1	input size: 160x160, depth multiplier = 0.5 input size: 192x192, depth multiplier = 1.0
MobileNetV1 + SSD	input size: 256x256, depth multiplier = 0.125
SqueezeNet	input size: 160x160, depth multiplier = 0.75

Graph 10 The official guideline of constraint models

This method is of narrow limitation, in which we cannot achieve the goal by setting our own TensorFlow Model and change the structure to what we want. Also, there are so many syntax error and uncertainty of the command provided by google. For a conclusion, we **cannot** use the official module to achieve the digit recognition goal.

2.2 TensorFlow is not Available on Kit (*Unsuccessful*)

From the mentioned situation in 2.1, we cannot successfully use the official compiler provided by google, which should have transformed the TensorFlow model to binary file, originally. Thus, we try to perform our own model by TensorFlow or Keras on Raspberry Pi, which we already have one trained model.

However, after having a detailed search on the website, we found that, the Raspberry Pi equipped on the Google Vision Kit is Raspberry Pi zero, based on the ARM 6 architecture, cannot support TensorFlow. (Generally, we can run TensorFlow directly on Raspberry Pi 3B or above version.)That means, we **cannot successfully** install the TensorFlow module at all, the provided official demo like face detection of your smile/angry from Google is just a binary file compiled by google.

However, the compiler doesn't support to transform your own TensorFlow model or it is extremely hard(which make us waste several days to try to make it works). Thus, this method is aborted.

2.3 Forward Inference using NumPy (*Successful !*)

Since 2.1 and 2.2 demonstrated that the unsuccessful working with official guide and the TensorFlow installation on the Raspberry Pi Zero as well, we found a method to use NumPy to directly calculate the forward inference procedure and output the predicted result directly. Because the System of Raspberry Pi have some fundamental python libraries like **NumPy, PIL**(Python Image Library), so we can process the captured photos and make computation.

The detail steps for this process are as follows:

(1) Train the model for Digit Recognition(Or Fruits, we will talk about it later), by using Keras or TensorFlow, with dataset MNIST.

(2) Since it is not difficult to train a simple convolutional neural network, so in this place we omit the procedure to train a model. We just clarify how to achieve the forward process and get the right answer.

(3) Notice that we cannot directly use the TensorFlow to calculate the result by simply using the trained model by the code "model.predict" or something like this. Hence, to deal with the question, we wrote the mathematic calculation functions to make the forward process. The following is the concrete python codes.

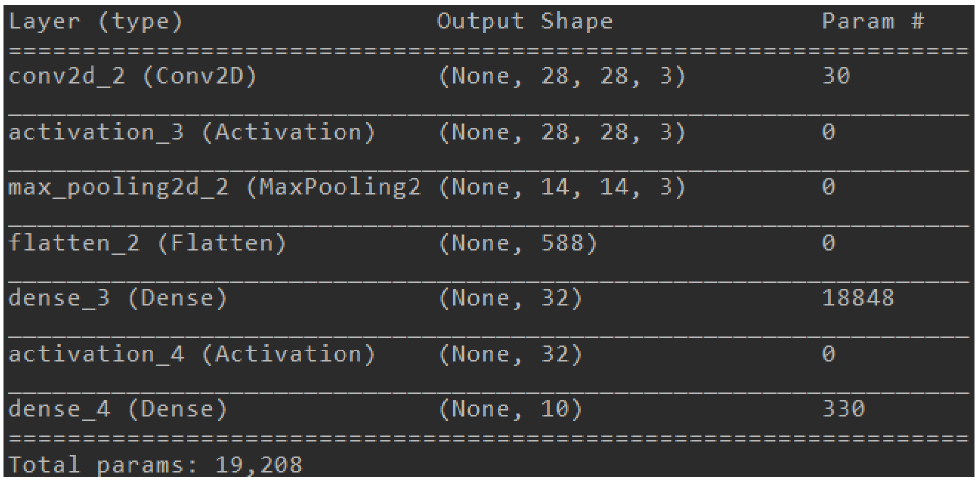
Suppose that we have got the trained model named "MNIST.h5". Now, we start building the inference functions step by step:
Firstly, we must import some dependencies functions:

```
import keras
import numpy as np
from keras.models import load_model
```

## This can help us to gain the weights from the ***MNIST.h5*** file.

```
model = load_model(r'C:\Users\Yunqing\Desktop\MNIST.h5')
print(model.summary())
```

By loading the ***MNIST.h5*** model file, and using "model.summary", we may get the following demonstration of our trained model.



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 28, 28, 3)         30
_________________________________________________________________
activation_3 (Activation)    (None, 28, 28, 3)         0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 3)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 588)               0
_________________________________________________________________
dense_3 (Dense)              (None, 32)                18848
_________________________________________________________________
activation_4 (Activation)    (None, 32)                0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                330
=================================================================
Total params: 19,208
```

Graph 11 Trained Network Structure

The respective parameters of the model are saved in the corresponding layers, the zero place or "get_weights()[0]" represents the weights of the specified layer, and "get_weights()[1]" represents the bias of the specified layer (the following code shows how to extract the weights and bias from the layer 0, i.e. conv2d_2 (Conv2D)).

```
weight_origin_0 = model.layers[0].get_weights()[0]
weight_origin_01 = model.layers[0].get_weights()[1]
```

Actually, you can extract every weights and bias by changing the number of the layer. However, there is no need to extract the weights or bias from the layer that have no parameters (just see the last row of summary of the model by code "model.summary"), otherwise you will get the error. So, just use the weights that already existed.

After storing the parameters in the variable, we should output the weights and biases into .npy file by NumPy.

```
np.save('cnn_weights_1.npy', weight_origin_1)
np.save('cnn_bias_1.npy', weight_origin_11)
```

The above code shows how to save the weights and biases into a npy file. Next, we will show how to use the every npy file, to calculate the result by forward functions.

Now, since we get the npy file of our ***MNIST.h5*** model, we can specify how to calculate and connect these files (matrix) together, to get our final prediction value.

```
# using NumPy to achieve the calculation functions
# NumPy is embedded in the  Raspberry Pi, so no need to install
import numpy as np
# loading the np files, which can attach the parameters in each layer.
class Recognizer(): # Initialize with a class
   def __init__(self):
       self.conv_weights = np.load('model/weights/conv_weights.npy')
       self.conv_bias = np.load('model/weights/conv_bias.npy')
       self.fc1_weights = np.load('model/weights/fc1_weights.npy')
       self.fc1_bias = np.load('model/weights/fc1_bias.npy')
       self.fc2_weights = np.load('model/weights/fc2_weights.npy')
       self.fc2_bias = np.load('model/weights/fc2_bias.npy')
```

From now on, we have already loaded the every npy file we need to use later, these np matrixes can be connected by the following mathematic functions.

```
# Normalization the input data
   def Normalization(self, x):
       max_x = np.max(x)
       min_x = np.min(x)
       y = (x - min_x) / (max_x - min_x)
       return y
# Relu Function
   def Relu(self, x):
       x[x<0] = 0
       return x
# Resize input Tensor to (1,n) shape
   def Flatten(self, x):
       x = x.reshape(1,-1)
       return x
# Dense Layer
   def Dense(self, inputs, kernel, bias):
       # Initialize the dimension of input tensor and bias as (1,n)
       inputs = inputs.reshape(1,-1)
       bias = bias.reshape(1,-1)
       # vectorize the output tensor
       output = np.dot(inputs, kernel) + bias
       return output
# Convolutional network calculation
   def Conv2D(self, inputs, kernels, biases, strides=(1,1)):
       # input Tensor dimension as [heigh,width,channel]
       input_size = inputs.shape
       # kernel size dimension as [heigh,width,channel,k]
       kernel_size = kernels.shape
       # padding the input tensor according to the kernel size
       h_pad = int(kernel_size[0] / 2)
       w_pad = int(kernel_size[1] / 2)
       inputs = np.pad(inputs,
pad_width=((h_pad,h_pad),(w_pad,w_pad),(0,0)), mode='constant')
       # compute the dimension of output tensor
       w_new = int((input_size[0] - kernel_size[0] + 2*h_pad) /
strides[0] + 1)
       h_new = int((input_size[1] - kernel_size[1] + 2*w_pad) /
strides[1] + 1)
       # initialize the output tensor with calculated dimension
       outputs = np.zeros((w_new,h_new,kernels.shape[3]))
       # convolution with channel k
       for k in range(kernel_size[3]):
        # (i_new,j_new)represents the place this step in the output
tensor
        #(i,j)represents the window place in this step, by the center
of (i,j)
           for i_new,i in enumerate(range(h_pad, inputs.shape[0]-
h_pad, strides[0])):
               for j_new,j in enumerate(range(w_pad, inputs.shape[1]-
w_pad, strides[1])):
               # temp represents the computed value in the temporary
window.
                   temp = inputs[i-h_pad:i+h_pad+1, j-
w_pad:j+w_pad+1, :]
                   temp = self.Flatten(temp)
                   # kernel represents the k th kernel
                   kernel = self.Flatten(kernels[:,:,:,k])
               # compute the temporary value, then output to the
tensor
                   value = np.dot(temp, kernel.T) + biases[k]
                   outputs[i_new,j_new, k] = value
       return outputs
```

# Maxpooling Layer

```
   def Max_pooling(self, x, kernel_size=(2,2), strides=(2,2)):
       # compute the dimension of output tensor
       width, heigh, channel = x.shape
       # initialize the output tensor depending onkernel_size and
strides
       w_new = int((width - kernel_size[0]) / strides[0] + 1)
       h_new = int((heigh - kernel_size[1]) / strides[1] + 1)
       c_new = channel
       outputs = np.zeros((w_new,h_new, c_new))
```

```
        # pooling by layers, depends on the channel k
        for k in range(channel):
        # (i_new,j_new) as coordinates in output tensor
        # (i,j)as coordinates of input tensor
            for i_new,i in enumerate(range(0, width, strides[0])):
                for j_new,j in enumerate(range(0, heigh, strides[1])):
                    # (i,j)as upper left, extract the local largest
value
                    temp = x[i:i+kernel_size[0],j:j+kernel_size[1]]
                    outputs[i_new,j_new,k] = np.max(temp)
         return outputs
```

```
# To enhance the input image.
```

```
    def enhancement(self, x):
        mean = np.mean(x)
        x[x>=mean] = 1.0
        x[x<mean] = 0.0
        return x
```

By using these functions, we can imitate the calculation process which originally executed by TensorFlow or Keras Module.

By this way, we defined functions for computation of involved matrix for fully connected network , convolutional procedure for CNN, and Activation function like “RELU” and the output layer “SoftMax”, and the MaxPooling as well.

Then, we combine the functions with reference to the Model structure as Graph 11.

```
    def Inference(self,x):

        x = self.Conv2D(x, self.conv_weights, self.conv_bias,
strides=(1,1))     # Just follow the network structure
        x = self.Relu(x)
        x = self.Max_pooling(x, kernel_size=(2,2), strides=(2,2))
        x = self.Flatten(x)
        x = self.Dense(x, self.fc1_weights, self.fc1_bias)
        x = self.Relu(x)
        x = self.Dense(x, self.fc2_weights, self.fc2_bias)
        x = self.Normalization(x) # normalize the output predicted
vector
        return x
```

Finally, we output the prediction result.

```
    def predict(self, x, img_enhance=False):
       x = x.reshape((28,28,1))   # reshape the image
       x = x / 255.0              # normalize
       if img_enhance:
                x = self.enhancement(x)
```

```
        output = self.Inference(x)
        label = int(np.where(output==np.max(output))[1])
        # decide the max one as the prediction class
        # The Forward Inference procedure ended here.
        return  output,label
```

This means, we **no longer need TensorFlow** anymore, and we **no longer need OpenCV Module**(which was also of unsuccessful installation on Raspberry Pi for Vision Kit), we just analog the process of forward inference, and try to use PIL library (also embedded in Raspberry pi, use "import PIL" to call the library)to read the photos rather than OpenCV, use NumPy rather than TensorFlow to get the prediction result. Finally, we just need the communication circuit mentioned in Part 1 to send the prediction result to our WeChat chatting room.

The details of the inference procedure are as below:

Firstly, we need to introduce some dependencies as we mentioned above. The class Recognizer is from a python file named "Inference_Engine".

```
from Inference_Engine import Recognizer
import numpy as np
from PIL import Image   # read the image using embedded library
import os
                    # read related files
entire_files = []  # append the file path
for parent_path, _, files in os.walk('hand_written_data/data'):
    for file_name in files:
        path = os.path.join(parent_path, file_name)
        entire_files.append(path)  # append all the file path
```

Then, we thus can use these functions to test on the real pictures, by just input the following codes:

```
from matplotlib import pyplot as plt
img_path = entire_files[400]  # Randomly select one image as input
img = np.array(Image.open(img_path).convert('L'))
plt.imshow(img,'gray')

model = Recognizer()   # recognition class
model.predict(img)     # output the result (open in jupyter notebook
to directly show the result in code block, otherwise you can set a
variable to get the output by "output, label = model.predict(img)")
```

2.4 Real-time Detection on Written Number & Fruits

Now it is time to get start on combining all the modules together to recognize the real hand-written number & Fruits or anything you want. We should:

1. Building our own dataset

The ***MNIST*** dataset provide us the rich resource of hand-written number from different people, thus there is no need to get other data in this dataset, temporarily. However, we need to build our own dataset of different kinds of ***Fruits:***

Graph 12 Fruits images

2. Train your neural network model

In this step, you need use either Keras or TensorFlow or some other structure you like to get the well-trained Neural Network model. Suppose we have already used Keras to train the data by:

(1) Use ImageDataGenerator from Keras to generate the training data and valid data:

```
train_datagen = ImageDataGenerator(
       rescale=1./255,
       shear_range=0.2,
       zoom_range=0.2,
       horizontal_flip=False)
train_generator = 
train_datagen.flow_from_directory('C:\\Users\\Yunqing\\Desktop\\F\\Fr
uit_Data\\Fruit_data\\train',target_size=(160, 160),
                 batch_size=16,
                 shuffle=True,
                 class_mode='categorical')
```

(2) Train your model:

```
model.fit_generator(train_generator,
                      epochs=100,
                      validation_data=val_generator,
                      verbose=2) # training set
```

Then, by:

```
   model.save('model.h5')    # save in the same path
```

to gain the h5 file of neural network model.

3. Output the Weights of the network:

```
    import numpy as np
    from keras.models import load_model
    model = load_model(r'C:\Users\Yunqing\Desktop\Fruit.h5')#(You can
    use any model name as you want)
    # output the weights depending on your network structure
    weight_origin_1 = model.layers[1].get_weights()[0]
    weight_origin_11 = model.layers[1].get_weights()[1]
    # above is just an example
    # Save your weights as np file
    np.save('cnn_weights_1.npy', weight_origin_1)
    np.save('cnn_bias_1.npy', weight_origin_11) # get weights matrix
```

4. All the process of step 1 -3 are finished on your PC. Now, since we acquire all the files, weights, model files and datas as we want, we can get the inference result on the raspberry pi. The contents of following steps are the Linux command lines.

```
    Python3 demo.py
```

   You will see the functions are running, and you can use pictures to get the classification results.

5. Shoot the image

   (1) In demo.py, using the extended library named PiCamera in demo.py to shoot a picture and feed the picture into CNN model. (by just use "***python3 demo.py***" in Raspberry command line.)

   (2) After this, send the picture and its result of model prediction to WeChat. Then we can easily see the visualization about the result. And all the process can be interrupted by typing:

```
        Ctrl+C
```

   The task will be ended, and:

```
        sudo poweroff
```

   this will power off the Vision Kit.

   (3) Here are some problems with the Vision Kit **Camera**:
   The camera sometimes or randomly works in an abnormal way, by showing the changed color of the picture, without any extra setting or execution of the hardware or software, as the following picture shows:

Graph 13 Comparison between abnormal pictures(**left**) and normal pictures.(**right**)

   We both regard it as a hardware problem, maybe the next version of kit will fix this problem by upgrading the camera.

Part 3. Problems of *Real-Number* Recognition

3.1 Comparison between *MNIST* and Real-Number

The resource from the Internet are almost about "How to build the Machine Learning Mode for MNIST ". However, few people have curiosity about recognition of real handwritten number by us.

Since we have Google Vision Kit, we naturally want to make fully use of it, that is, once we shoot the **real number** we write on the paper/iPad, the Kit can automatically calculate the prediction result, as the process of Part 2.3, and output the class of number through the WeChat.

For this task, we invite some of our friends to write the real digits to test the model, the number of real numbers is 524.

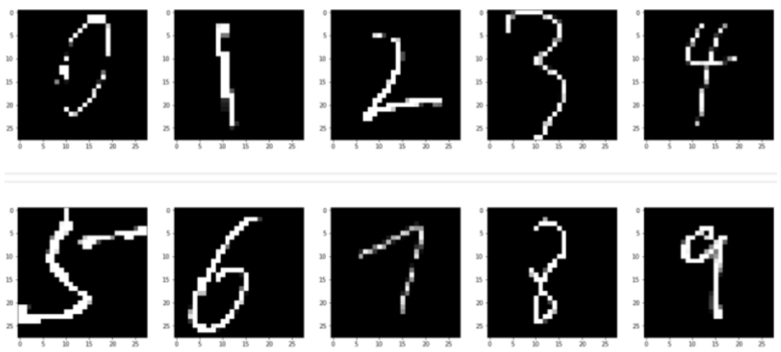
Not that easy, we soon found that the trained mature neural network model with highly accuracy on MNIST(99% or above) performed **poor** on the real numbers, as the Table 1 shows.

Datasets	**Loss**	**Accuracy**
MNIST	0.0687	0.9798
REAL NUMBER	**2.4393**	**0.41129**

Table 1 Result Comparison of Raw Model

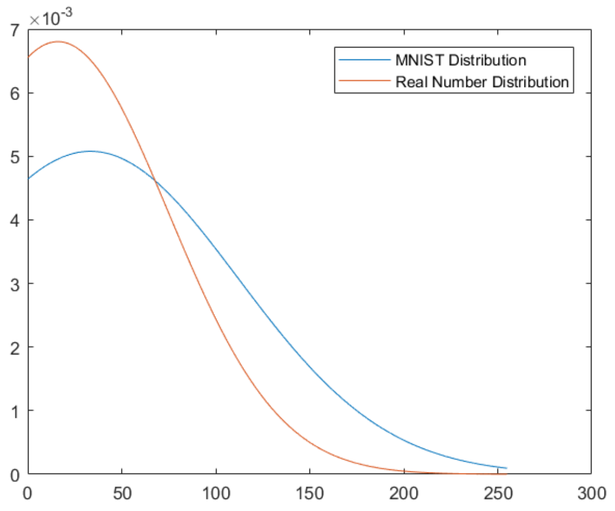
The following Graphs show the comparison between MNIST dataset and Real Hand-Written Numbers.

Graph 14 MNIST Number


Graph 15 Real Hand-Written Number

The problem is at this. After calculating the distribution of these numbers and the difference about the two datasets, we find that, the mean value of MNIST is 33.38596, the standard deviation is 78.567444. However, the mean value of Real Hand-Written

Number is 16.05849, and the standard deviation is 58.651166. Assuming that the distributions of these datasets are both Gaussian distributions, we plot the curves of them:



Graph 16 Probability Density Function of both Distribution

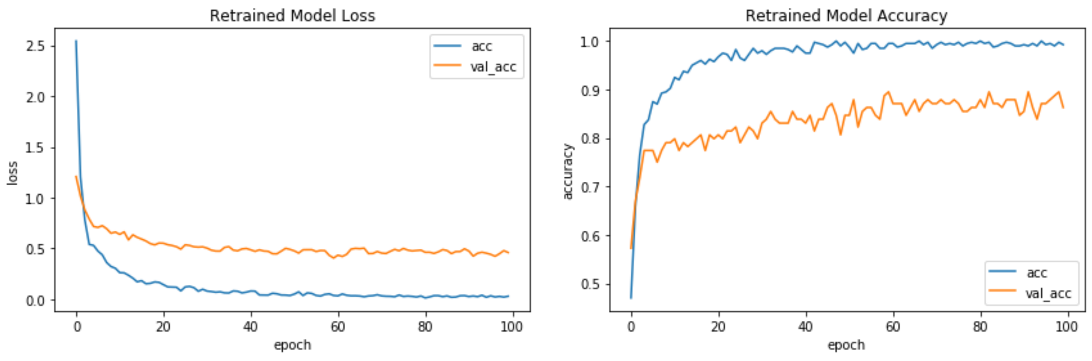
Hence, we clearly see that, the different distribution of respective dataset, that's why we cannot get the same performance as that of the MNIST images----in machine learning, we should keep the training set and testing set the IID condition(Independent and identically distributed).

3.2 Solution to the Problem: Transfer Learning!

Due to the big difference of their accuracy, we coped with the problem by transfer the raw model to retrain it on the Brand-New Dataset, named "Hand-Written Dataset", we will call it HWD from now on.

The HWD consists of 524 hand written numbers, written by 4 people. Three of the authors wrote the training set, and another one wrote the number for validation set, which means the validation set of HWD from a total new person, and the training set NOT consist of any number from the person that wrote the validation number. This is the background.

Then, we load the trained raw model of MNIST, deploy the HWD, and retrain the model. As has been mentioned before(Table 1), the raw model performed poor on the HWD with only 40% accuracy, however it jumped to 88% at the end of the retrain station, as Graph 17 shows, the decrease of the loss value and the increase of the accuracy demonstrate the stronger capability of generalization.



Graph 17 The training accuracy of retrained model

3.3 Realization of Real-Time Recognition

We have mentioned this in part 2.4. This is the final stage of this demo. We made the synthesis of the above all units. In this stage, we let the Google Vision Kit shoot the real-time digit Written Number and then output the result to the WeChat chatting room.

As the Graph 18 shows, once we shoot the photo by Kit, the Raspberry Pi will make inference calculation of the model forwardly and return the result in a short latency.

Based on the principle of considering all the details, we also upload the ***Video*** of our test scene by shooting and the inference procedure on the YouTube: https://youtu.be/ZyAYFuCDyrI. Feel free to reach it. The test result is relatively good, but we must point out some problems, which will be prohibited later in the fruit recognition:

(1) During the procedure of making HWD (Real digit numbers by hand), we scratch the square size of the picture, and then resize to 28*28. However, in the normal cases, we cannot get the so perfect size by simply shooting the photos in a casual way. So, maybe the training result is so good, but once we test the model as the YouTube Video like, we must fix the position of the Vision Kit and then take pictures. Once we take photos in a random angle or posture, the accuracy will drop down. So, for next task on fruit recognition, we will prohibit it by making datasets in a random angle and different size.

Graph 18 Test Scene

(2) During the procedure of making Real digit numbers by hand, even though from different people, all the digit numbers come from iPad, which means the dataset are all absolute binary image, however the image when shooting is of gray scale, and the image effect will also decay by effect of the

indoor lighting or sunshine or reflection of the light.

Hence, in order to achieve the better result, we must make sure the distribution of the image keep identity, i.e. take photos by Google Vision Kit to make training set, thus the similar pictures can be correctly recognized by the device.

Part 4. Conclusion

In this project, we design an integrated, device-based recognizer Vision Kit model. Firstly, we use the python library “itchat” and local Wi-Fi network to build a platform in order to make communication. Then we form a new database HWD and some fruits to retrain/build the network based on the raw one.

We also have met some problems, like communication platform construction, and the inference functions building. Please refer to the part 2.

Finally, the Vision Kit can recognize the real digit number and send the prediction result to the chatting room in a short latency, that is what we have achieved. And some technical problems will be resolved and dealt with in later task. Next, we may form a new goal and new objects, to have a total combination of the all modules, from dataset, algorithm, to the final result.