ECE479ICC LAB 2: Raspberry Pi, TF-Keras, and Face Detection

PART I: Raspberry Pi Setup and Basics

Raspberry Pi Setup

From this lab on, you will apply what you have learned in the lectures to train and deploy a neural network on an edge device, Raspberry Pi 4.

Raspberry Pi 4 is an inexpensive but powerful IoT device. You may review this report (https://www.techrepublic.com/article/inside-the-raspberry-pi-the-story-of-the-35-computer-that-changed-the-world/) to learn more about its story. To start, you need to install the proper operating system, you can refer to this tutorial (https://www.tomshardware.com/reviews/raspberry-pi-headless-setup-how-to,6028.html). You can use the official imager (https://www.raspberrypi.com/software/) to prepare your OS, you should use this image_file (https://github.com/google-coral/aiy-maker-kit-tools/releases/download/v20220518/aiy-maker-kit-2022-05-18.img.xz). If you are at home without a monitor, you can set up the OS with the headless option.

Please notice that we are using **Raspberry Pi 4**, with **custom OS(provided)**.

Once your RPi can boot into the OS normally, you can enable the VNC and access the desktop environment using your laptop/desktop. You can follow the "Enabling and Connecting over VNC" section of the tutorial (https://www.tomshardware.com/reviews/raspberry-pi-headless-setup-how-to,6028.html) for this.

Now that you have the desktop environment, follow this guide (https://picamera.readthedocs.io/en/latest/) to set up your pi-camera. You should be able to take pictures and videos with your camera.

In this part of the lab, you need to complete the following tasks:

- Install Raspbian OS in your Raspberry Pi, set up your network connection, and take a selfie on the pi-camera.
- **Print** the CPU specs with `cat /proc/cpuinfo`
- **Print** network interface configuration with `ifconfig`

TensorFlow Setup on Raspberry Pi

Once you can log into the Raspberry Pi desktop, please verify to make sure tf_lite is correctly installed in the raspberry pi. By running

```
import tflite_runtime.interpreter as tflite
```

in a python3 environment. If there is no error, the installation is done.

If error occurs, you can follow the this_guide (https://www.tensorflow.org/lite/guide/python) and this_guide (https://coral.ai/docs/accelerator/get-started/) to install **TensorFlow Lite** on your Raspberry Pi and work with the accelerator.

TensorFlow Setup on Your PC

Please follow the following commands in your conda environment to install **TensorFlow 2.10** on your own PC.

```
conda install -c conda-forge tensorflow=2.10
```

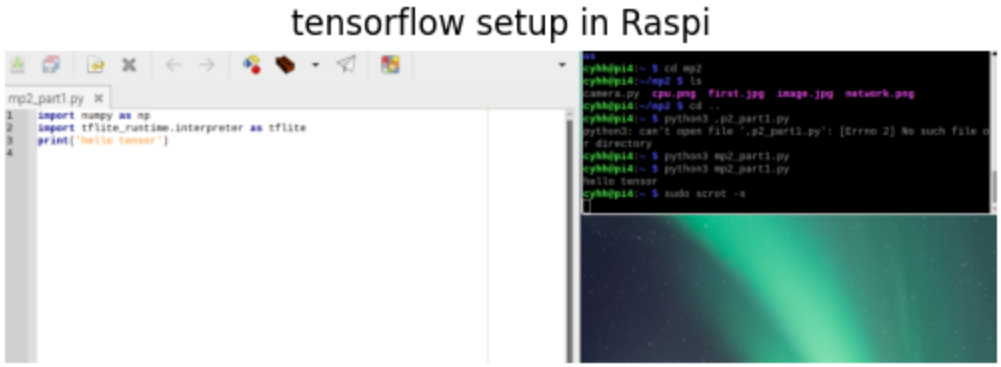
If you have a Apple Silicon Mac, please follow this guide (https://towardsdatascience.com/installing-tensorflow-on-the-m1-mac-410bb36b776). It is a little more complicated.

After the installation. Please verify that the installation is properly done. By running

```
import tensorflow as tf
tf.__version__
```

in a python environment, notice the version number for your Tensorflow

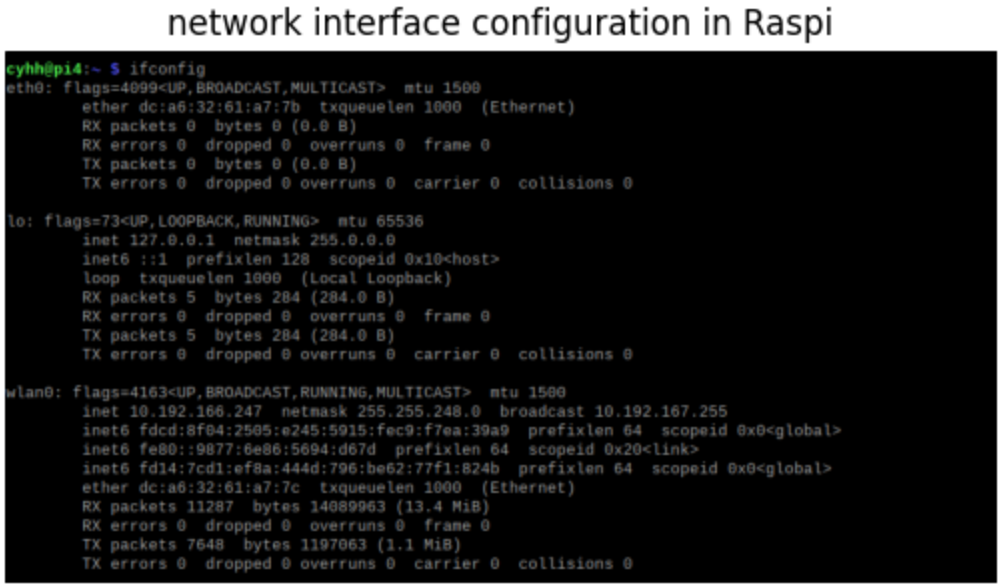
In [4]:
```
import os
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Directory containing the images
filepath = 'img_raspi/tensorlite.png'
# Load the image using Matplotlib
img = mpimg.imread(filepath)
# Display the image
plt.imshow(img)
plt.title("tensorflow setup in Raspi")  # Set the title of the plot to the filename
plt.axis('off')  # Turn off axis
plt.show()
```



In [39]:
```
# Directory containing the images
filepath = 'img_raspi/cpu.png'
# Load the image using Matplotlib
img = mpimg.imread(filepath)
# Display the image
plt.imshow(img)
plt.title("cpu_spec in Raspi")  # Set the title of the plot to the filename
plt.axis('off')  # Turn off axis
plt.show()
print("code:cat /proc/cpuinfo")
```

code:cat /proc/cpuinfo

In [40]:
```
# Directory containing the images
filepath = 'img_raspi/network.png'
# Load the image using Matplotlib
img = mpimg.imread(filepath)
# Display the image
plt.imshow(img)
plt.title("network interface configuration in Raspi")  # Set the title of the plot to
plt.axis('off')  # Turn off axis
plt.show()
print("code:ifconfig")
```




```
code:ifconfig
```

PART II: Building and Training with TF-Keras

In part II, you will build a Convolutional Neural Network (CNN) based image classifier. In our lecture, we have introduced CNN, a type of Deep Neural Network (DNN) that features convolutional layers. This document (http://cs231n.github.io/convolutional-networks/) by Stanford CS231n is also a good resource to learn more about CNN. You will build and train a simple CNN with TensorFlow on your own computers.

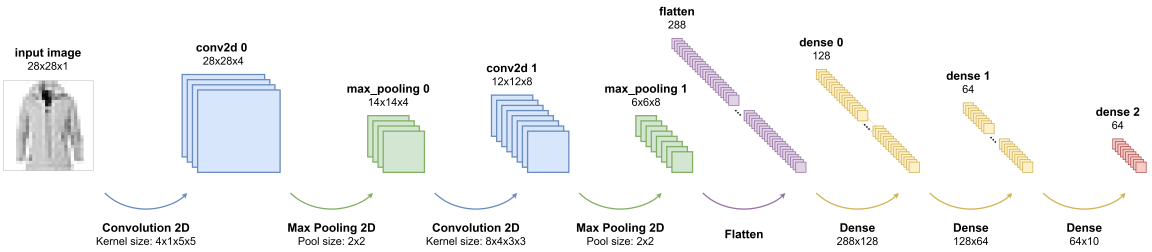
There were two ways to construct and train a neural network model in TF1.x: Keras and low-level APIs. As Tensorflow evolves to 2.x, it leans more toward Keras, a higher-level framework than the previous version's low-level APIs. Keras APIs provide straightforward construction and training of neural networks, which hide away some tedious details of the network.

Some may argue that low-level APIs provide more precise control over some aspects of the network. While users can still find a way to tune the details by looking deeper into the APIs, the framework encourages them to focus on the big picture and develop better and more robust network models rather than fixating on trivialities. As a beginner, it is helpful for us to start with Keras and establish a basic intuition on how to build and train a simple network.

In this part of the lab, you need to follow this tutorial (https://www.tensorflow.org/tutorials/keras/classification) to build a neural network to classify the

FashionNet

You will construct a neural network by yourselves. Let's call it the FashionNet. Here is an overview of the proposed Convolutional Neurual Network (CNN):



This CNN contains two convolutional layers, two max pooling layers, and three fully connected layers (dense layers). The configuration details of each layer are shown in the following table (in sequential order):

Layers	Configuration	Activation	Output Dimensions
convolution	input size = (28, 28, 1), kernel size = (5, 5), stride = 1, padding = 'same'	ReLU	(28, 28, 4)
max pooling	pool_size = (2, 2), stride = 2	-	(14, 14, 4)
convolution	kernel size = (3, 3), stride = 1, padding = 'valid'	ReLU	(12, 12, 8)
max pooling	pool_size = (2, 2), stride = 2	-	(6, 6, 8)
dense		ReLU	(128)
dense		ReLU	(64)
dense		Softmax	(10)

Dataset

In this lab, you will use the Fashion MNIST dataset. You can load the dataset in the following way, which is also shown in the tutorial:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.loa
d_data()
```

As discussed in the lecture, you should only use `train_images` and `train_labels` to train your model. You may also want to validate your model with a training/validation split. We will discuss the details in the next section.

Build, Train, and Evaluate

Generally, there are three stages in the process of developing a neural network: build, train, and evaluate. In Keras, there are corresponding functions for each stage:

1. **Build** First, you define the model architecture. In this lab, you can use a simple `keras.Sequential` function to define a model, as shown in this example (https://www.tensorflow.org/guide/keras/sequential_model#when_to_use_a_sequential_mod There are other ways to define a model, but we will not discuss those in this lab. After you have defined your model, you need to tell Keras what you want to do with it by compiling it. In the `model.compile` function, you specify the optimizer, loss function, metrics, and other configurations to prepare for training.
2. **Train** You use `model.fit` function to train your model. In this function, you can specify the training parameters, including batch size, epochs, etc. You should specify your validation split in this function too. Please refer to the document and see how to do that.
3. **Evaluate** After you have trained your model, you need to test the final accuracy using the testing data. The `model.evaluate` function gives you the percentage of the correct results.

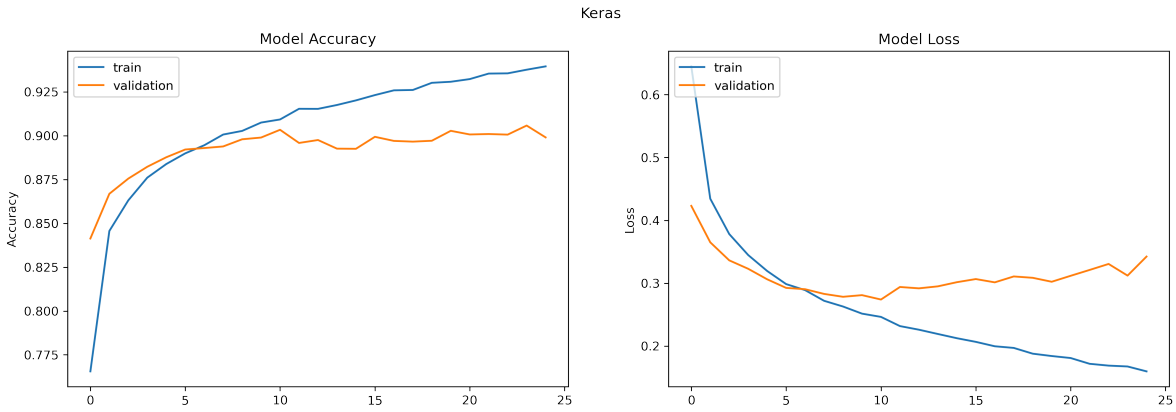
You can find the document for the above functions here (https://keras.io/api/models/model_training_apis/). You may also find that in FashionNet, you use 2-D convolution in the first couple of layers. You may find this document (https://www.tensorflow.org/api_docs/python/tf/keras/layers) helpful since the `layers.Conv2D` and `layers.MaxPool2D` functions are not covered in the tutorial.

The training of this CNN should not take a very long time on a laptop/desktop CPU. The final testing accuracy should be higher than 85%, and the training accuracy should be higher than 90%.

Plot the Training Progress

The `model.fit` function outputs a history of the training metrics (accuracy and loss), including validation metrics. This infomation can be very helpful for diagnosing the model.

Use `matplotlib` to graph the model accuracy and loss on the training and validation dataset, and include them in your report. The graph should look something like the following, but yours can be different if you have different configurations of the training process.



Saving Your Trained Model

Finally, you will need to save your trained models to model files. Saving a trained model allows you to deploy it to other devices or restore previous training progress at a later time.

For the purpose of this lab, we ask you to save it using **the TensorFlow SavedModel format**, which will contain the information of the model as well as the weights.

```
# Calling `save('my_model')` creates a SavedModel folder `my_model`.
your_trained_model.save("my_model")
```

Please read this document

In [2]:
```
# TensorFlow and tf.keras
import os
import tensorflow as tf
print(tf.__version__)
from tensorflow import keras
from tensorflow.keras import layers
# # Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

```
2.1.0
Num GPUs Available:  0
```

In [11]:
```
# TODO:
#  Load Fashion MNIST dataset
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
train_images=train_images.reshape((60000,28,28,1))
# train_labels=train_labels.reshape((60000,1))

# reshape the input images to satisfy required format
test_images=test_images.reshape((10000,28,28,1))
# test_labels=test_labels.reshape((10000,1))

# normalize the images
train_images= train_images /255.0
test_images= test_images /255.0

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

In [9]:
```
# 2. Build, Train, and Evaluate

## Build
# Define Sequential model with 7 layers
model = keras.Sequential(
    [
#         keras.Input(shape=(28,28,1)),
        layers.Conv2D(4,(5,5),strides=(1, 1),padding='same',activation="relu",input_s
        layers.MaxPooling2D(pool_size=(2, 2),strides=(2,2),padding='valid'),
        layers.Conv2D(8,(3,3),strides=(1, 1),padding='valid',activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2),strides=(2,2),padding='valid'),
        layers.Flatten(),
        layers.Dense(128, activation="relu"),
        layers.Dense(64, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)


# Define the optimizer with custom learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)  # Specify your desired lea

## Train
model.compile(optimizer='adam',loss=tf.keras.losses.SparseCategoricalCrossentropy(fro
model.summary()
# Train the model
history = model.fit(train_images, train_labels, epochs=20, batch_size=64, validation_
# history = model.fit(train_images,train_labels, epochs=10)


# Evaluate the model
test_loss, test_accuracy = model.evaluate(test_images, test_labels)

# Print test accuracy
print('Test accuracy:', test_accuracy)
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 28, 28, 4)         104
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 14, 14, 4)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 12, 12, 8)         296
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 6, 6, 8)           0
_________________________________________________________________
flatten_3 (Flatten)          (None, 288)               0
_________________________________________________________________
dense_9 (Dense)              (None, 128)               36992
_________________________________________________________________
dense_10 (Dense)             (None, 64)                8256
_________________________________________________________________
dense_11 (Dense)             (None, 10)                650
=================================================================
Total params: 46,298
Trainable params: 46,298
Non-trainable params: 0
_________________________________________________________________
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [==============================] - 11s 224us/sample - loss: 1.7936 - ac
curacy: 0.6726 - val_loss: 1.7192 - val_accuracy: 0.7425
Epoch 2/20
48000/48000 [==============================] - 10s 202us/sample - loss: 1.6957 - ac
curacy: 0.7675 - val_loss: 1.6881 - val_accuracy: 0.7722
Epoch 3/20
48000/48000 [==============================] - 10s 211us/sample - loss: 1.6777 - ac
curacy: 0.7847 - val_loss: 1.6826 - val_accuracy: 0.7793
Epoch 4/20
48000/48000 [==============================] - 10s 204us/sample - loss: 1.6692 - ac
curacy: 0.7923 - val_loss: 1.6794 - val_accuracy: 0.7813
Epoch 5/20
48000/48000 [==============================] - 10s 208us/sample - loss: 1.6620 - ac
curacy: 0.7993 - val_loss: 1.6652 - val_accuracy: 0.7958
Epoch 6/20
48000/48000 [==============================] - 10s 200us/sample - loss: 1.6557 - ac
curacy: 0.8057 - val_loss: 1.6273 - val_accuracy: 0.8352
Epoch 7/20
48000/48000 [==============================] - 9s 197us/sample - loss: 1.6029 - acc
uracy: 0.8594 - val_loss: 1.6060 - val_accuracy: 0.8559
Epoch 8/20
48000/48000 [==============================] - 9s 198us/sample - loss: 1.5885 - acc
uracy: 0.8730 - val_loss: 1.5911 - val_accuracy: 0.8707
Epoch 9/20
48000/48000 [==============================] - 10s 200us/sample - loss: 1.5848 - ac
curacy: 0.8771 - val_loss: 1.5899 - val_accuracy: 0.8708
Epoch 10/20
48000/48000 [==============================] - 10s 201us/sample - loss: 1.5801 - ac
curacy: 0.8814 - val_loss: 1.5973 - val_accuracy: 0.8637
Epoch 11/20
48000/48000 [==============================] - 10s 200us/sample - loss: 1.5768 - ac
```

```
curacy: 0.8846 - val_loss: 1.5821 - val_accuracy: 0.8791
Epoch 12/20
48000/48000 [==============================] - 10s 201us/sample - loss: 1.5727 - ac
curacy: 0.8886 - val_loss: 1.5855 - val_accuracy: 0.8751
Epoch 13/20
48000/48000 [==============================] - 10s 202us/sample - loss: 1.5681 - ac
curacy: 0.8938 - val_loss: 1.5786 - val_accuracy: 0.8825
Epoch 14/20
48000/48000 [==============================] - 10s 200us/sample - loss: 1.5689 - ac
curacy: 0.8925 - val_loss: 1.5817 - val_accuracy: 0.8796
Epoch 15/20
48000/48000 [==============================] - 11s 237us/sample - loss: 1.5651 - ac
curacy: 0.8964 - val_loss: 1.5830 - val_accuracy: 0.8775
Epoch 16/20
48000/48000 [==============================] - 11s 219us/sample - loss: 1.5650 - ac
curacy: 0.8965 - val_loss: 1.5788 - val_accuracy: 0.8819
Epoch 17/20
48000/48000 [==============================] - 10s 198us/sample - loss: 1.5626 - ac
curacy: 0.8989 - val_loss: 1.5788 - val_accuracy: 0.8816
Epoch 18/20
48000/48000 [==============================] - 11s 223us/sample - loss: 1.5593 - ac
curacy: 0.9022 - val_loss: 1.5838 - val_accuracy: 0.8774
Epoch 19/20
48000/48000 [==============================] - 10s 205us/sample - loss: 1.5582 - ac
curacy: 0.9032 - val_loss: 1.5765 - val_accuracy: 0.8843
Epoch 20/20
48000/48000 [==============================] - 9s 195us/sample - loss: 1.5562 - acc
uracy: 0.9051 - val_loss: 1.5733 - val_accuracy: 0.8874
10000/10000 [==============================] - 1s 101us/sample - loss: 1.5733 - acc
uracy: 0.8883
Test accuracy: 0.8883
```

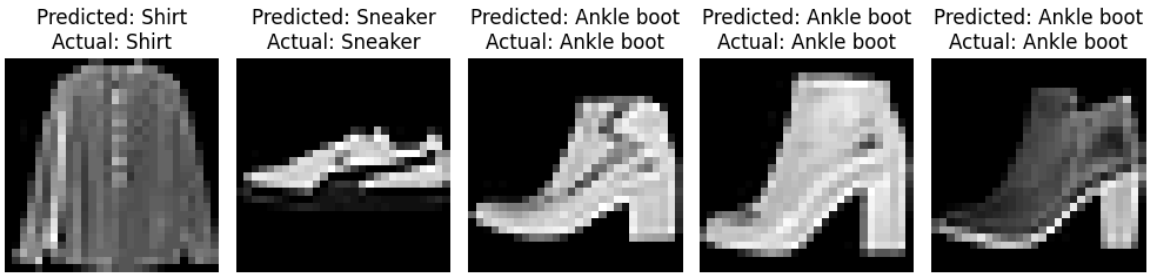
Training Accuracy: 90.51%

Testing Accuracy: 88.83%

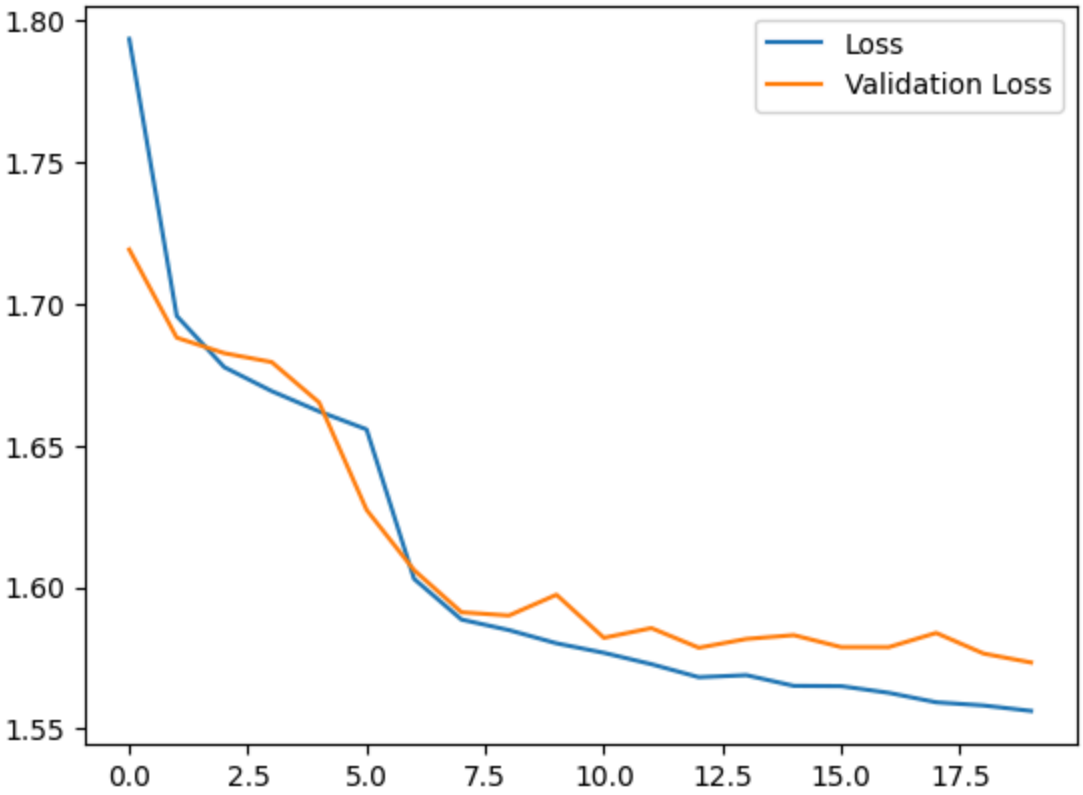
In [13]:
```
## user evaluation of the quality of the model

probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)
# predictions[0]
print(np.argmax(predictions[0]))
print(test_labels[0])
print(class_names[test_labels[0]])
```

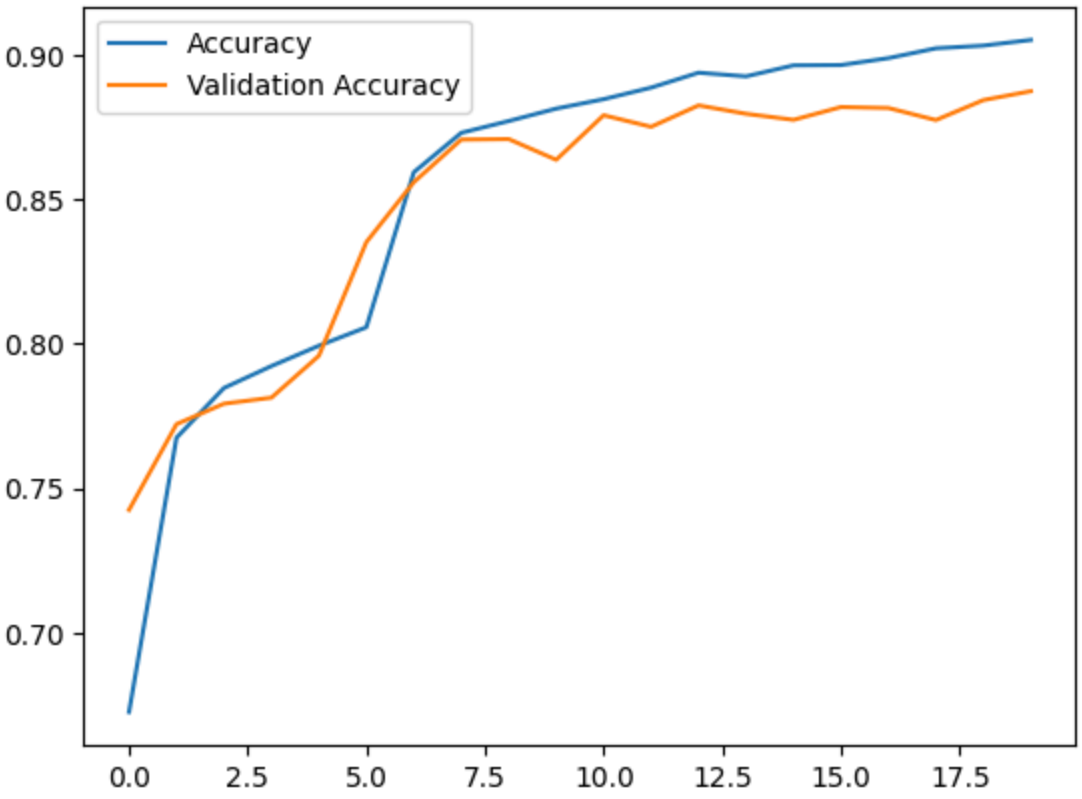
```
9
9
Ankle boot
```

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt

# Assuming model_predictions is an array of predicted labels for test_images
# and test_labels are the actual labels

# Plot some random images along with their predicted and actual labels
num_images_to_plot = 5
random_indices = np.random.choice(len(test_images), size=num_images_to_plot, replace=

plt.figure(figsize=(10, 5))

for i, idx in enumerate(random_indices):
    plt.subplot(1, num_images_to_plot, i+1)
    plt.imshow(test_images[idx], cmap='gray')
    predicted_label = np.argmax(predictions[idx])
    actual_label = test_labels[idx]
    plt.title(f'Predicted: {class_names[predicted_label]}\nActual: {class_names[actual
    plt.axis('off')

plt.tight_layout()
plt.show()
```



In [15]:
```
## plot the training loss rate and validation loss rate with respect to epoches
loss = np.array(history.history['loss'])
acc = np.array(history.history['accuracy'])
val_loss = np.array(history.history['val_loss'])
val_acc = np.array(history.history['val_accuracy'])
plt.figure()
plt.plot(loss, label="Loss")
plt.plot(val_loss, label="Validation Loss")
plt.legend()
plt.show()
plt.plot(acc, label="Accuracy")
plt.plot(val_acc, label="Validation Accuracy")
plt.legend()
plt.show()
```





In [14]:
```
# 3. Save your model
model.save("my_model")
```

```
WARNING:tensorflow:From C:\Users\14435\anaconda3\envs\tensorflow\lib\site-packages
\tensorflow_core\python\ops\resource_variable_ops.py:1786: calling BaseResourceVari
able.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is
deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
INFO:tensorflow:Assets written to: my_model\assets
```

Convert Your Trained Model to TFLite model

Before you start the migration process of your trained model onto our edge devices, Raspberry Pi 4, please make sure you have followed the previous steps carefully and have a properly trained TF model with reasonable accuracy on the validation dataset.

Now, locate the directory where your trained model files are saved and create a TFLite converter ready for the next step.

Please refer to [this document](https://www.tensorflow.org/lite/api_docs/python/tf/lite/TFLiteConverter#from_saved_model) for more information on the TFLite model.

In [18]:
```
# TODO: Convert to TFLite
# Converting a SavedModel to a TensorFlow Lite model.
converter = tf.lite.TFLiteConverter.from_saved_model("my_model")
```

Optimize your TFLite model with post-trainning quantization

With the converter, we can directly transform our model to TFLite model; however, edge and IoT devices are usually low-powered and relatively lack computational power compared to PC. Therefore, to further enhance the performance, you can apply the post-training quantization technique before deploying the model onto the Raspberry Pi.

Notice that, although the post-training quantization is similar to the quantization and retrain technique that we will introduce in Lecture 12, since the model will not be retrained after the quantization, the impact on the accuracy of the model is usually larger.

In this part, you will need to apply two quantization methods, namely, **Dynamic range quantization** and **Full integer quantization**, listed in this document (https://www.tensorflow.org/lite/performance/post_training_quantization) to your pre-trained models and compare the accuracy and performance.

1. For dynamic range quantization, quantization parameters are automatically detected and generated by TensorFlow. Please write a piece of code to convert your original trained model into a TFLite model using dynamic range quantization.
2. For integer quantization, TensorFlow needs to sample the dataset you used for training the models to estimate the **MIN** and **MAX** values. This process can be described as "representative data generation". An example representative data generator is

```
def representative_data_gen():
  for input_value in tf.data.Dataset.from_tensor_slices(trainning_dataset).
batch(1).take(100):
    yield [input_value]
```

You will need to define your own representative data generator function and properly select the following parameters as below:

```
# dataset used for training
trainning_set
# batch number x
.batch(x)
# input y
.take(y)
```

Please refer to this official document (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices) for explanations on the **from_tensor_slices()** API. On the same page, you will also learn about how the TensorFlow ***tf.Data.Dataset*** format works.

In [20]:
```
# TODO:
# 1. Dynamic range quantization
converter = tf.lite.TFLiteConverter.from_saved_model("my_model")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_dy_range_quant_model = converter.convert()
# Save the model.
with open('Dynamic_range_Quant_model.tflite', 'wb') as f:
    f.write(tflite_dy_range_quant_model)
```

In [21]:
```
# 2. Full integer quantization
def representative_dataset():
    for data in tf.data.Dataset.from_tensor_slices((train_images)).batch(1).take(100):
        yield [tf.dtypes.cast(data, tf.float32)]

converter = tf.lite.TFLiteConverter.from_saved_model("my_model")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8  # or tf.uint8
converter.inference_output_type = tf.int8  # or tf.uint8
tflite_full_int_quant_model = converter.convert()
# Save the model.
with open('full_int_Quant_model.tflite', 'wb') as f:
    f.write(tflite_full_int_quant_model)
# with open('full_int_Quant_model_1.tflite', 'wb') as f:
#     f.write(tflite_full_int_quant_model)
```

Save and deploy your TFLite model

Save your converted TFLite models as binary files. Example code:

```
# Save the model.
with open('model.tflite', 'wb') as f:
  f.write(tflite_model)
```

To run execute the TFLite model on the RPi, you can refer to [this example (https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_python)](https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_python) for the details. In general, the following code is the standard process.

Step 1: load the model and allocate the tensors

We first need to instantiate the model with the TFLite model file, and then allocate all the tensors.

```
# Load the TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()
```

Step 2: Check input and output info

To get the input and output tensor's names and their corresponding sizes, we will first need do the following steps to extract these information.

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

Step 3: Set input tensor and run the model

Once we got the input tensor's name and shape, we can then set the input tensor and run the model.

```
# Test the model on random input data.
input_shape = input_details[0]['shape']
interpreter.set_tensor(input_details[0]['index'], input_data)

# Run the model
interpreter.invoke()
```

Step 4: Extract the output tensor

We can finally run the `get_tensor(...)` function again to extract the output.

```
# The function `get_tensor()` returns a copy of the tensor data.
# Use `tensor()` in order to get a pointer to the tensor.
output_data = interpreter.get_tensor(output_details[0]['index'])
print(output_data)
```

Once you confirm that the model is deployable on your RPi, you are ready to use the Coral Edge TPU to accelerate your computing. Please follow this tutorial (https://coral.ai/docs/accelerator/get-started/#1-install-the-edge-tpu-runtime) to install required Edge TPU runtime. Follow tutorial (https://coral.ai/docs/accelerator/get-started/#3-run-a-model-on-the-edge-tpu) to run your converted TFLite model on your Raspberry Pi with the Edge TPU.

Please convert your model with these two different methods and report their corresponding accuracy and inference time with the TFLite model on the RPi with and without the Edge TPU. Compare and comment on what you observed in the report.

```
Dynamic Quatization without TPU:
accuracy:88.76
inference time/ms:5428.4954071044921875000000000000000

Dynamic Quantization with TPU:
accuracy:88.76
inference time/ms:5332.8530788421630859375000000000000

Full-integer Quantization without TPU:
accuracy:88.68
inference time/ms:2449.8648643493652343750000000000000

Full-integer Quantization with TPU:
accuracy:88.68
```

```
inference time/ms:2208.0540657043457031250000000000000
```

1. Observation:

   (1)The accuracy of Dynamic Quatization is a little bit higher than Full-integer Quantization; But the accuracy with TPU has no difference from that without TPU.

   (2)The inference time of Dynamic Quatization is slower than Full-integer Quantization; And the inference time with TPU is faster than that without TPU.
2. Explanation:

   (1)Dynamic Quantization typically involves less aggressive quantization compared to Full-integer Quantization. Therefore, it might preserve more information during the quantization process, leading to slightly higher accuracy.

   (2)However, the use of a TPU may not significantly impact accuracy because TPUs primarily accelerate computation but do not directly affect the quantization process or model architecture. Hence, there may be no noticeable difference in accuracy when using a TPU compared to without using one.

   (3)Dynamic Quantization involves quantizing the model dynamically during runtime, which can introduce overhead and potentially increase inference time compared to Full-integer Quantization, where the quantization is done offline.

   (4)On the other hand, TPUs are specifically designed to accelerate computation, including inference tasks. Therefore, using a TPU can lead to faster inference times compared to running inference on a CPU or GPU alone.Consequently, the observation that Dynamic Quantization has slower inference time than Full-integer Quantization aligns with expectations, as does the observation that inference time is faster with TPU compared to without TPU.

On average, their inference time is quite similar.Because there is no significant improvemnt if we use edge TPU. It might because we connect edge TPU to raspberry pi which may result in time loss during data transfeering.

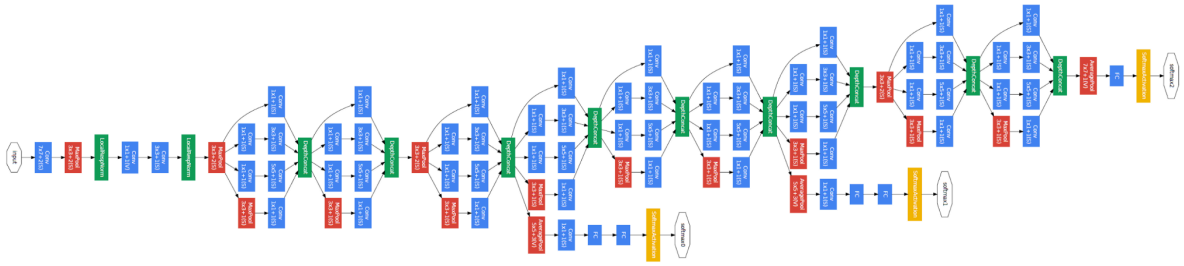
PART III: Face Detection and Recognition

In part III, you will have a chance to utilize your Raspberry Pi 4 as a powerful edge platform for human face detection and recognition applications. You will need to capture valid images using your PiCamera and port them into the recognition neural network with proper processing.

Prepare a Pre-trained Convolutional Neural Network

Introduction

In this part, you will use the award-winning "very deep neural network" for face recognition. It is the ILSVRC 2014 winner, GoogLeNet, and it also has the name ***Inception ResNet v1***. For more information, please refer to this review (https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7), and the original paper could be found here (https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf). The goal of this part is to reconstruct the Inception ResNet v1 with Keras APIs, load the given pre-trained weights, and convert the whole model into a TFLite model. You will need the knowledge from the previous parts of the lab.



The original GooLeNet (Inception ResNet v1) is huge and complex. It has 22 major layers and more than 10 milion trainable parameters. Thus, it will be painful to reconstruct it using the same way you learned in **Part II** of this lab, i.e., listing out all layers in order by using `keras.Sequential` function.

There is a better way to define a model architecture. You can use the pre-built Keras APIs to reconstruct the Inception Resnet v1. You can find all the layers here (https://www.tensorflow.org/api_docs/python/tf/keras/layers). It may be helpful to have this page opened on aside while working on this part.

For example, to apply two `tf.keras.layers.Conv2D` layers to the existing model `x`, you just need to

```
from tensorflow.keras.layers import Conv2D
x = Conv2D(...)(x)
x = Conv2D(...)(x)
```

The output of model `x` on the left side will be connected by a new `Conv2D` layer and the resulting new model will be assigned to `x` on the right side. By doing this repeatedly in proper order, you achieve the same goal as using `keras.Sequential`.

The given files structure is listed as,

```
.
├── lab2_part3.py
├── inception_resnet.py
├── resnet_block.py
├── conv2d_bn.py
├── modules.py
└── weights
    └── inception_keras_weights.h5
```

We have provided the `modules.py` and `resnet_block.py` for you, and we expect you NOT to modify this file. You will need to implement code blocks within `conv2d_bn.py` and `inception_resnet.py` according to the instructions and parameters given to you.

Because of the dependencies between functions, it's highly recommended that you implement the codes following the order of steps below. **Notice: while applying a new Keras layer, please make sure you include the `name=` argument.** This will not affect the functionality of your model, but it's helpful for you to locate problematic layers. You can follow the convention of the example blocks/layers to name your layers/blocks with the `name_fmt` and

Step 1: Implement `conv2d_bn(*)` function

`conv2d_bn(*)` is defined in the `conv2d_bn.py`. We have imported all the Keras layer APIs for you, and thus you can directly use them by calling their names, such as `Conv2D` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) and `Activation` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Activation).

This function is relatively simple and helps you get familiar with the Keras format. Look for `## TO DO` in the function.

`## TO DO Step1`: Apply a `Conv2D` layer to model `x`, with all useful parameters listed as arguments in the function signature of `conv2d_bn(*)`. Assign the new model back to `x`. When calling `Conv2D`, make sure all the available input arguments for the `conv2d_bn()` function are used.

`## TO DO Step2`: Apply a Batch Normalization layer to model `x`, with

Arguments	Values
axis	bn_axis
momentum	0.995
epsilon	0.001
scale	False
name	bn_name

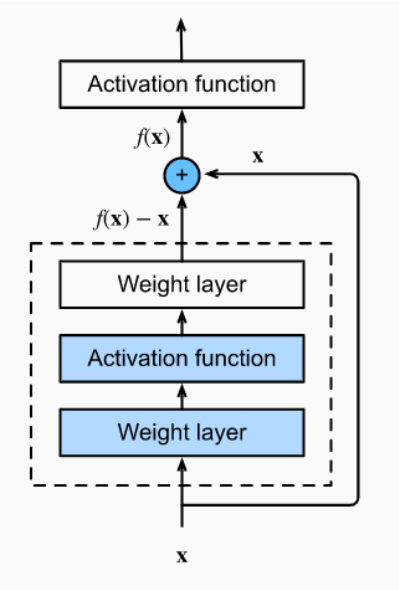
`## TO DO Step3`: Apply an Activation layer to model `x`, with the argument `activation` passed into the `conv2d_bn(*)`. Use `ac_name` as the name for the layer.

Step 2: Familiarize with `inception_resnet_block(*)` function

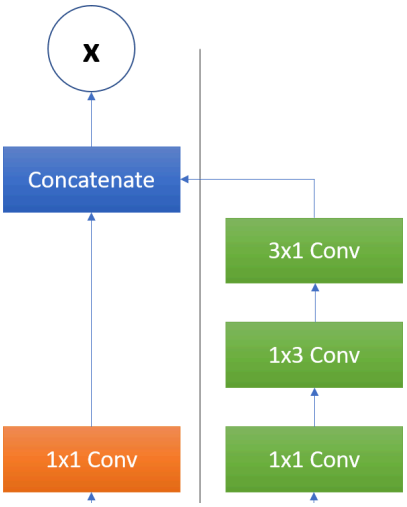
`inception_resnet_block(*)` is defined in the `resnet_block.py`. The function defines three different types of Inception ResNet blocks `Inception_block_a`, `Inception_block_b` and `Inception_block_c`.

The function implements reusable building blocks for inception blocks in the Inception ResNet v1. You can imagine this function as the pre-built bricks in LEGO, and what you need to do is to put them in the proper places when you build the entire network.

ResNet (https://d2l.ai/chapter_convolutional-modern/resnet.html) or Residual Netoworks has a very special stucture called **shortcut**.



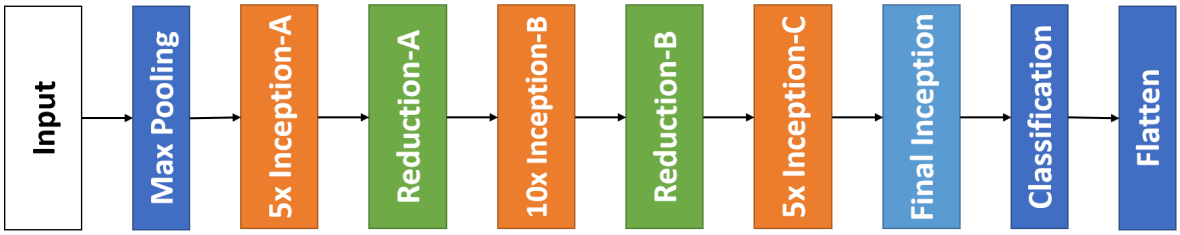
In a more general sense, it allows multiple parallel paths, or "branches", through the neural network. These "branches" are merged or aggregated by a specific layer named `Concatenate`. For example, in `Inception_block_c`, we have two branches: `branch_0` and `branch_1`. `branch_0` has one Conv 2D layer with 192 output channels and a kernel size of *1x1*. `branch_1` has three Conv 2D layers with 192 output channels and kernel sizes of *1x1*, *1x3*, and *3x1*.



Step 3: Finish the Inception ResNet Structure

Now, with the building blocks that you have prepared in previous steps, you have three powerful tools to build the actual Inception ResNet v1, a very deep convolutional neural network that has 448 layers and sublayers. (You do not even want to list out all of these layers.)

The overview of the Inception ResNet v1 is,



You need `tensorflow.keras.layers`, `conv2d_bn(*)`, and `resnet_block(*)` to build the network. Follow the instructions and hints below to finish the whole convolutional neural network.

`## TO DO Step1`: Finish the Maxpooling 2D preprocessing for model `x` with,

Layers	Configuration	Output Dimensions
conv2d_bn	input size = (79, 79, 32), kernel size = (3, 3), padding='valid'	(77, 77, 32)
conv2d_bn	connect to previous, kernel size = (3, 3)	(77, 77, 64)
MaxPooling2D	connect to previous, pool_size = (2, 2), stride = 2	(38, 38, 64)
conv2d_bn	connect to previous, kernel size = (1, 1), padding='valid'	(38, 38, 80)
conv2d_bn	connect to previous, kernel size = (3, 3), padding='valid'	(36, 36, 192)
conv2d_bn	connect to previous, kernel size = (3, 3), stride = 2, padding='valid'	(17, 17, 256)

`## TO DO Step2`: Instantiate **5 connected** Inception ResNet *block_type_a* using,

scale	block_idx	block_type
0.17	1 to 5	Inception_block_a

`## TO DO Step3`: Instantiate **10 connected** Inception ResNet *block_type_b* using,

scale	block_idx	block_type
0.1	1 to 10	Inception_block_b

`## TO DO Step4`: Instantiate **5 connected** Inception ResNet *block_type_c* using,

scale	block_idx	block_type
0.2	1 to 5	Inception_block_c

`## TO DO Step5`: Apply Global Average Pooling (`GloablAveragePooling2D(...)`) + Dropout (`Drop(...)`) layers to model `x`. You can find the documentation of these layers here (https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalAveragePooling2D) and here (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout). For the Dropout layer, use `dropout_keep_prob` to properly calculate the `rate` argument of the `Dropout` layer. (Hint: what's the relationship between "keep" and "dropout"?)

Step 4: Compile the model and load weights

The structure of Inception ResNet v1 is completed and ready to be compiled. Implement your code in `lab2_part3.py`. This file will be the place where your main logic should go.

To compile a model, simply do

```
model = InceptionResNetV1Norm();
```

**You may see some warnings due to TensorFlow version difference, but they do not usually matter.*

Now try to print your model's summary to find out the number of trainable parameters and the number of layers. You will need these numbers while writing your report.

To load the pre-trained weights, use `model.load_weights(*)`. We have prepared the weights for you located at `"weights/inception_keras_weights.h5"`.

At this point, if your model loads without any other errors, then your model implementation is

Step 5: Convert the model into TFLite model

If you reach here without any problem, your implementation for Inception ResNet v1 should be basically correct.

Unfortunately, there is no easy way to partially verify your implementation, so if you meet any errors that stall the process, you should go back to previous steps to fix any existing problem before proceeding.

Follow the same instructions as **Convert Your Trained Model to TFLite model** in **Part II**. You **do not** need to apply post-training quantization this time.

In [2]:
```
import os
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Directory containing the images
directory = "model_summary"

# List all files in the directory
files = os.listdir(directory)

print(" Output of Loading Weights for Our Model ")
# Iterate over the files in the directory
for file in files:
    # Check if the file is an image (you can adjust the condition based on your file e
    if file.endswith('.jpg') or file.endswith('.png'):
        # Construct the full path to the image
        filepath = os.path.join(directory, file)

        # Load the image using Matplotlib
        img = mpimg.imread(filepath)

        # Display the image
        plt.imshow(img)
        plt.title(file)  # Set the title of the plot to the filename
        plt.axis('off')  # Turn off axis
        plt.show()
```

Output of Loading Weights for Our Model

1.png

batch_normalization_96 (BatchNo	(None, 3, 3, 192)	576	Inception_block_c_3_Branch_1_Conv
activation_96 (Activation)	(None, 3, 3, 192)	0	batch_normalization_96[0][0]
Inception_block_c_3_Branch_1_Co	(None, 3, 3, 192)	110592	activation_96[0][0]
batch_normalization_97 (BatchNo	(None, 3, 3, 192)	576	Inception_block_c_3_Branch_1_Conv
activation_97 (Activation)	(None, 3, 3, 192)	0	batch_normalization_97[0][0]
Inception_block_c_3_Branch_0_Co	(None, 3, 3, 192)	344064	Inception_block_c_2_Activation[0]
Inception_block_c_3_Branch_1_Co	(None, 3, 3, 192)	110592	activation_97[0][0]
batch_normalization_95 (BatchNo	(None, 3, 3, 192)	576	Inception_block_c_3_Branch_0_Conv
batch_normalization_98 (BatchNo	(None, 3, 3, 192)	576	Inception_block_c_3_Branch_1_Conv
activation_95 (Activation)	(None, 3, 3, 192)	0	batch_normalization_95[0][0]
activation_98 (Activation)	(None, 3, 3, 192)	0	batch_normalization_98[0][0]
Inception_block_c_3_Concatenate	(None, 3, 3, 384)	0	activation_95[0][0] activation_98[0][0]
Inception_block_c_3_Conv2d_1x1	(None, 3, 3, 1792)	689920	Inception_block_c_3_Concatenate[0
lambda_17 (Lambda)	(None, 3, 3, 1792)	0	Inception_block_c_3_Conv2d_1x1[0]
add_17 (Add)	(None, 3, 3, 1792)	0	Inception_block_c_2_Activation[0] lambda_17[0][0]
Inception_block_c_3_Activation	(None, 3, 3, 1792)	0	add_17[0][0]
Inception_block_c_4_Branch_1_Co	(None, 3, 3, 192)	344064	Inception_block_c_3_Activation[0]
batch_normalization_100 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_4_Branch_1_Conv
activation_100 (Activation)	(None, 3, 3, 192)	0	batch_normalization_100[0][0]
Inception_block_c_4_Branch_1_Co	(None, 3, 3, 192)	110592	activation_100[0][0]
batch_normalization_101 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_4_Branch_1_Conv

2.png

batch_normalization_100 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_4_Branch_1_Conv
activation_100 (Activation)	(None, 3, 3, 192)	0	batch_normalization_100[0][0]
Inception_block_c_4_Branch_1_Co	(None, 3, 3, 192)	110592	activation_100[0][0]
batch_normalization_101 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_4_Branch_1_Conv
activation_101 (Activation)	(None, 3, 3, 192)	0	batch_normalization_101[0][0]
Inception_block_c_4_Branch_0_Co	(None, 3, 3, 192)	344064	Inception_block_c_3_Activation[0]
Inception_block_c_4_Branch_1_Co	(None, 3, 3, 192)	110592	activation_101[0][0]
batch_normalization_99 (BatchNo	(None, 3, 3, 192)	576	Inception_block_c_4_Branch_0_Conv
batch_normalization_102 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_4_Branch_1_Conv
activation_99 (Activation)	(None, 3, 3, 192)	0	batch_normalization_99[0][0]
activation_102 (Activation)	(None, 3, 3, 192)	0	batch_normalization_102[0][0]
Inception_block_c_4_Concatenate	(None, 3, 3, 384)	0	activation_99[0][0] activation_102[0][0]
Inception_block_c_4_Conv2d_1x1	(None, 3, 3, 1792)	689920	Inception_block_c_4_Concatenate[0
lambda_18 (Lambda)	(None, 3, 3, 1792)	0	Inception_block_c_4_Conv2d_1x1[0]
add_18 (Add)	(None, 3, 3, 1792)	0	Inception_block_c_3_Activation[0] lambda_18[0][0]
Inception_block_c_4_Activation	(None, 3, 3, 1792)	0	add_18[0][0]
Inception_block_c_5_Branch_1_Co	(None, 3, 3, 192)	344064	Inception_block_c_4_Activation[0]
batch_normalization_104 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_5_Branch_1_Conv
activation_104 (Activation)	(None, 3, 3, 192)	0	batch_normalization_104[0][0]
Inception_block_c_5_Branch_1_Co	(None, 3, 3, 192)	110592	activation_104[0][0]
batch_normalization_105 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_5_Branch_1_Conv

3.png

batch_normalization_105 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_5_Branch_1_Conv
activation_105 (Activation)	(None, 3, 3, 192)	0	batch_normalization_105[0][0]
Inception_block_c_5_Branch_0_Co	(None, 3, 3, 192)	344064	Inception_block_c_4_Activation[0]
Inception_block_c_5_Branch_1_Co	(None, 3, 3, 192)	110592	activation_105[0][0]
batch_normalization_103 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_5_Branch_0_Conv
batch_normalization_106 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_5_Branch_1_Conv
activation_103 (Activation)	(None, 3, 3, 192)	0	batch_normalization_103[0][0]
activation_106 (Activation)	(None, 3, 3, 192)	0	batch_normalization_106[0][0]
Inception_block_c_5_Concatenate	(None, 3, 3, 384)	0	activation_103[0][0] activation_106[0][0]
Inception_block_c_5_Conv2d_1x1	(None, 3, 3, 1792)	689920	Inception_block_c_5_Concatenate[0
lambda_19 (Lambda)	(None, 3, 3, 1792)	0	Inception_block_c_5_Conv2d_1x1[0]
add_19 (Add)	(None, 3, 3, 1792)	0	Inception_block_c_4_Activation[0] lambda_19[0][0]
Inception_block_c_5_Activation	(None, 3, 3, 1792)	0	add_19[0][0]
Inception_block_c_6_Branch_1_Co	(None, 3, 3, 192)	344064	Inception_block_c_5_Activation[0]
batch_normalization_108 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_6_Branch_1_Conv
activation_108 (Activation)	(None, 3, 3, 192)	0	batch_normalization_108[0][0]
Inception_block_c_6_Branch_1_Co	(None, 3, 3, 192)	110592	activation_108[0][0]
batch_normalization_109 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_6_Branch_1_Conv
activation_109 (Activation)	(None, 3, 3, 192)	0	batch_normalization_109[0][0]
Inception_block_c_6_Branch_0_Co	(None, 3, 3, 192)	344064	Inception_block_c_5_Activation[0]
Inception_block_c_6_Branch_1_Co	(None, 3, 3, 192)	110592	activation_109[0][0]

4.png

Inception_block_c_6_Branch_1_Co	(None, 3, 3, 192)	110592	activation_109[0][0]
batch_normalization_107 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_6_Branch_0_Conv
batch_normalization_110 (BatchN	(None, 3, 3, 192)	576	Inception_block_c_6_Branch_1_Conv
activation_107 (Activation)	(None, 3, 3, 192)	0	batch_normalization_107[0][0]
activation_110 (Activation)	(None, 3, 3, 192)	0	batch_normalization_110[0][0]
Inception_block_c_6_Concatenate	(None, 3, 3, 384)	0	activation_107[0][0] activation_110[0][0]
Inception_block_c_6_Conv2d_1x1	(None, 3, 3, 1792)	689920	Inception_block_c_6_Concatenate[0
lambda_20 (Lambda)	(None, 3, 3, 1792)	0	Inception_block_c_6_Conv2d_1x1[0]
add_20 (Add)	(None, 3, 3, 1792)	0	Inception_block_c_5_Activation[0] lambda_20[0][0]
globalaveragepooling2D (GlobalA	(None, 1792)	0	add_20[0][0]
dropout_layer (Dropout)	(None, 1792)	0	globalaveragepooling2D[0][0]
Bottleneck (Dense)	(None, 512)	917504	dropout_layer[0][0]
Bottleneck_BatchNorm (BatchNorm	(None, 512)	1536	Bottleneck[0][0]
normalize (Lambda)	(None, 512)	0	Bottleneck_BatchNorm[0][0]

Total params: 23,497,424
Trainable params: 23,467,824
Non-trainable params: 29,600

Face detection and recognition with MTCNN & FaceNet and integration with PiCamera

Now that you have converted our network to a TensorFlow Lite model, you can start integrating all the components that you have implemented so far towards a more realistic face detection and recognition system.

Step 1: Image capturing with PiCamera

The first step of the work is to implement a simple function that can capture an image using the PiCamera and convert it to the forms that the neural networks can use. More specifically, in this lab, you will use the very basic image capturing feature of the PiCamera and transform the captured image into an OpenCV object.

You can refer to the "Capturing to an OpenCV object" section of the [PiCamera package documentation (https://picamera.readthedocs.io/en/release-1.10/recipes1.html#capturing-to-an-opencv-object)](https://picamera.readthedocs.io/en/release-1.10/recipes1.html#capturing-to-an-opencv-object) for the details of capturing images with the PiCamera and fill in the following image function with the given signature. The returned object should be a 3-D tensor, with the three channels in RGB order.

import cv2 import picamera import numpy as np def capture_image(): # Instrctor note: this can be directly taken from the PiCamera documentation # Create the in-memory stream stream = io.BytesIO() with picamera.PiCamera() as camera: camera.capture(stream, format='jpeg')

```
# Construct a numpy array from the stream
data = np.frombuffer(stream.getvalue(), dtype=np.uint8)

# "Decode" the image from the array, preserving colour
image = cv2.imdecode(data, 1)

# OpenCV returns an array with data in BGR order.
# The following code invert the order of the last dimension.
```

Step 2: Face detection

In general, face detection can be seen as a special case of object detection, where there is only one object class, namely, "human face". The Multi-Task Cascaded Convolutional Networks (MTCNN) (https://kpzhang93.github.io/MTCNN_face_detection_alignment/) is one of the pioneering works in this area. It is purposedly built for detecting human faces and also identifying the keypoints (eyes, nose, and mouth). It is one of the most popular and successful networks in this area.

For this part of the lab, you will use an easy-to-use Python package for MTCNN, which provides a simple API for performing face detections, which suites our needs for this lab. However, you should notice that, while having a convenient python package, MTCNN is not dedicated to real-time detection on mobile devices. Therefore the inference latency is a few seconds on the Raspberry Pi.

For this step, you will first instantiate the MTCNN from the mtcnn package. Then, write a function that does the following operations. The function prototype is given to you:

1. Perform face detection with MTCNN and extract the bounding box of the first image.
2. Add a 20% margin to each dimension of the bounding box, such that the bounding box is 120% of the original size in each dimension. That is, the bounding box is expanded by 10% in each direction(up, down, left, right).
3. Perform image cropping according to the coordinate of the extended bounding box.
4. Return the cropped image and the coordinates (x, y, w, h) of the bounding box.
5. Finally, display the bounding box in the original image.

To help you implement the code, here are some useful tips and links:

- For simplicity, you can assume that there is only one face in the input image.
- The basic usage of MTCNN can be found in the MTCNN python package description here (https://pypi.org/project/mtcnn/). As you may find out, the interface is very compact and intuitive. For this lab, you only need the 'box' from the output, which specifies the coordinates of a corner and the dimensions of the bounding box.
- Performing the cropping on the OpenCV image is basically the same as working on a NumPy array, which you have already implemented in Lab 1. So you can borrow from your Lab 1 code if needed.
- To help you to draw the bounding box, we provide a function here that can do it for you.

In [19]:
```
# pip install mtcnn
# function provided for the students to draw the rectangle
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
def show_bounding_box(image, bounding_box):
    x1, y1, w, h = bounding_box
    fig, ax = plt.subplots(1,1)
    ax.imshow(image)
    ax.add_patch(Rectangle((x1, y1), w, h, linewidth=1, edgecolor='r', facecolor='non
    plt.axis('off')
    plt.show()
    return
```

In [26]:
```
pip install mtcnn
```

```
Collecting mtcnnNote: you may need to restart the kernel to use updated packages.
  Downloading mtcnn-0.1.1-py3-none-any.whl (2.3 MB)

     ---------------------------------------- 2.3/2.3 MB 2.0 MB/s eta 0:00:00
Requirement already satisfied: keras>=2.0.0 in c:\users\14435\anaconda3\envs\tensor
flow\lib\site-packages (from mtcnn) (2.11.0)
Requirement already satisfied: opencv-python>=4.1.0 in c:\users\14435\anaconda3\env
s\tensorflow\lib\site-packages (from mtcnn) (4.9.0.80)
Requirement already satisfied: numpy>=1.17.0 in c:\users\14435\anaconda3\envs\tenso
rflow\lib\site-packages (from opencv-python>=4.1.0->mtcnn) (1.21.5)
Installing collected packages: mtcnn
Successfully installed mtcnn-0.1.1
```

In [10]:
```
from mtcnn.mtcnn import MTCNN
def detect_and_crop(mtcnn, image):
    detection = mtcnn.detect_faces(image)[0]
    #TODO
    #get the bounding dimension
    box = detection["box"]
    start_x = box[0]
    start_y = box[1]
    width = box[2]
    height = box[3]

    start_x = int(start_x - 0.1 * width) if (start_x - 0.1 * width > 0) else 0
    start_y = int(start_y - 0.1 * height) if (start_y - 0.1 * height > 0) else 0

    # add 20% margin
    width = int(width * 1.2)
    height = int(height * 1.2)


    # according to the shape of center crop picture, we choose from the original array
    cropped_image = image[start_y: start_y + height, start_x: start_x + width]
    # show_bounding_box(image, bounding_box)
    box = (start_x, start_y, width, height)
    # print('shape of new image',image_newarray.shape)
    # plt.title('Center-Crop')
    # plt.axis('off')
    # plt.imshow(cropper_image)

    return cropped_image, box
```

#unused here mtcnn = MTCNN() image = capture_image() cropped_image = detect_and_crop(mtcnn, image)

Step 3: Face Recognition

Once the faces are detected in the picture, you can then use your FaceNet to perform recognition of the faces. This step is a continuation of Part 3.1, and you now need to pass the image to the face recognition network and get the output feature vector.

Before passing the input image to the model, you need to preprocess the image, which includes resizing the image to the expected size of the model and standardize values across the channels. For your convenience, the preprocessing function is provided.

The run_model function should perform the following tasks:

- Extract the input and output shape details from the model
- Set the input tensor
- Invoke the model
- Extract and return the output tensor

This is almost a standard process of running TensorFlow Lite models. You can refer to [this example (https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_python)](https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_python) for the details, you need very few changes in general.

Finally, write a short script that calls the functions and perform the following steps:

1. Load the TensorFlow Lite model
2. Allocate the tensors
3. Preprocess the cropped image

In [15]:
```
# preprocessing function provided to the students

def pre_process(face, required_size=(160, 160)):
    ret = cv2.resize(face, required_size)
    ret = ret.astype('float32')
    mean, std = ret.mean(), ret.std()
    ret = (ret - mean) / std
    return ret
```

In [16]:
```

def run_model(model, face):
# students will need to fill in the following function
    #TODO
     # Load the TFLite model and allocate tensors.


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

    # Test the model on random input data.
    input_shape = input_details[0]['shape']
    input_data = face.reshape(input_shape)
    model.set_tensor(input_details[0]['index'], input_data)

    model.invoke()

    # The function `get_tensor()` returns a copy of the tensor data.
    # Use `tensor()` in order to get a pointer to the tensor.
    output_data = model.get_tensor(output_details[0]['index'])

    return output_data

import tensorflow as tf
def read_image(file):
    img = cv2.imread(file)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
#     img = tf.convert_to_tensor(img, dtype=tf.float32)
    return img
    #return output_data
```

In [17]:
```
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2)**2)).astype(np.float32)
```

Step 4: Verification and Report

To verify the correct functionality of your face recognition system, you will need to calculate the euclidean distances between the feature vectors of pictures of yourself and other people. If the distance between your pictures is significantly smaller than the distance between the pictures of you and other people, then your network basically functions as expected.

The distance between the feature vectors can be evaluated with the euclidean distance as you have implemented in Lab 1. We provide a read_image() function here for you to read images from files easily.

To demonstrate the functionality of your network, in your final report include **Distances between pairs of pictures, picked from 2 pictures of yourself, and 2 pictures of other people (6 possible pairs in total). Also show all 4 pictures in your report.**

(You can choose the pictures of other people freely. They can be headshots of the TAs on the course website or other famous people if you want.)

In [20]:
```
import tensorflow as tf
import numpy as np
import cv2
mtcnn = MTCNN()
## initialize mtcnn network

# @tf.function(experimental_relax_shapes=True)
# def my_function(input_tensor):
#     # Your TensorFlow operations here
#     return output_tensor


# the first person first image
path = "c1.jpg"
c1 = read_image(path)
pre_c_1, box1 = detect_and_crop(mtcnn, c1)
c_1_f = pre_process(pre_c_1)
c_1_f = np.clip(c_1_f, 0, 1)

#first person second image
path = "c2.jpg"
c2 = read_image(path)
pre_c2_1, box2 = detect_and_crop(mtcnn, c2)
c_2_f = pre_process(pre_c2_1)
c_2_f = np.clip(c_2_f, 0, 1)

#second person first image
path = "b1.jpg"
b1 = read_image(path)
pre_b_1, box3 = detect_and_crop(mtcnn, b1)
b_1_f = pre_process(pre_b_1)
b_1_f = np.clip(b_1_f, 0, 1)

#second person second image
path = "b2.jpg"
b2 = read_image(path)
pre_b_2, box4 = detect_and_crop(mtcnn, b2)
b_2_f = pre_process(pre_b_2)
b_2_f = np.clip(b_2_f, 0, 1)

# cyhh1
path = "cyhh1.jpg"
cyhh1 = read_image(path)
pre_cyhh_1, box5 = detect_and_crop(mtcnn, cyhh1)
cyhh_1_f = pre_process(pre_cyhh_1)
cyhh_1_f = np.clip(cyhh_1_f, 0, 1)

#cyhh2
path = "cyhh2.jpg"
cyhh2 = read_image(path)
pre_cyhh_2, box6 = detect_and_crop(mtcnn, cyhh2)
cyhh_2_f = pre_process(pre_cyhh_2)
cyhh_2_f = np.clip(cyhh_2_f, 0, 1)

#haoyuh1
path = "haoyuh1.jpg"
haoyu1 = read_image(path)
```

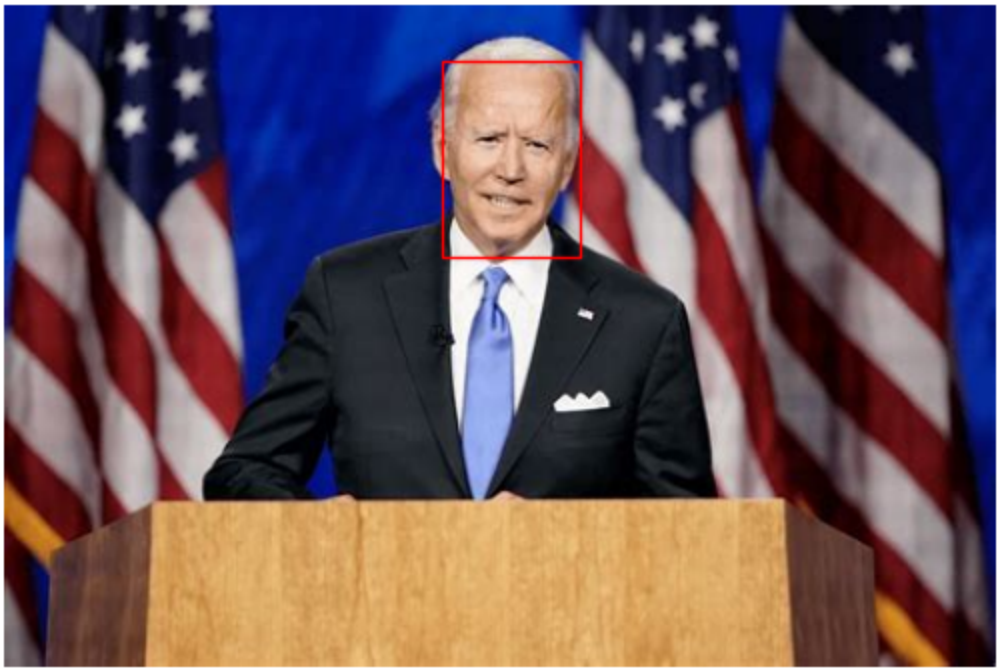
```
pre_haoyu_1, box7 = detect_and_crop(mtcnn, haoyu1)
haoyu_1_f = pre_process(pre_haoyu_1)
haoyu_1_f = np.clip(haoyu_1_f, 0, 1)

#haoyuh2
path = "haoyuh2.jpg"
haoyu2 = read_image(path)
pre_haoyu_2, box8 = detect_and_crop(mtcnn, haoyu2)
haoyu_2_f = pre_process(pre_haoyu_2)
haoyu_2_f = np.clip(haoyu_2_f, 0, 1)

plt.figure()

# display show bounding for each person we will locate the face of people
show_bounding_box(b1, box3)
show_bounding_box(b2, box4)
plt.imshow(b_1_f)
plt.axis('off')
plt.figure()
plt.imshow(b_2_f)
plt.axis('off')

show_bounding_box(c1, box1)
show_bounding_box(c2, box2)
plt.imshow(c_1_f)
plt.axis('off')
plt.figure()
plt.imshow(c_2_f)
plt.axis('off')


show_bounding_box(cyhh1, box5)
show_bounding_box(cyhh2, box6)
plt.imshow(cyhh_1_f)
plt.axis('off')
plt.figure()
plt.imshow(cyhh_2_f)
plt.axis('off')


show_bounding_box(haoyu1, box7)
show_bounding_box(haoyu2, box8)
plt.imshow(haoyu_1_f)
plt.axis('off')
plt.figure()
plt.imshow(haoyu_2_f)
plt.axis('off')

plt.show()

# load the model

tfl_file = "./code/GoogleNetV1Norm.tflite"
interpreter = tf.lite.Interpreter(model_path=tfl_file)
interpreter.allocate_tensors()

#run the model
```

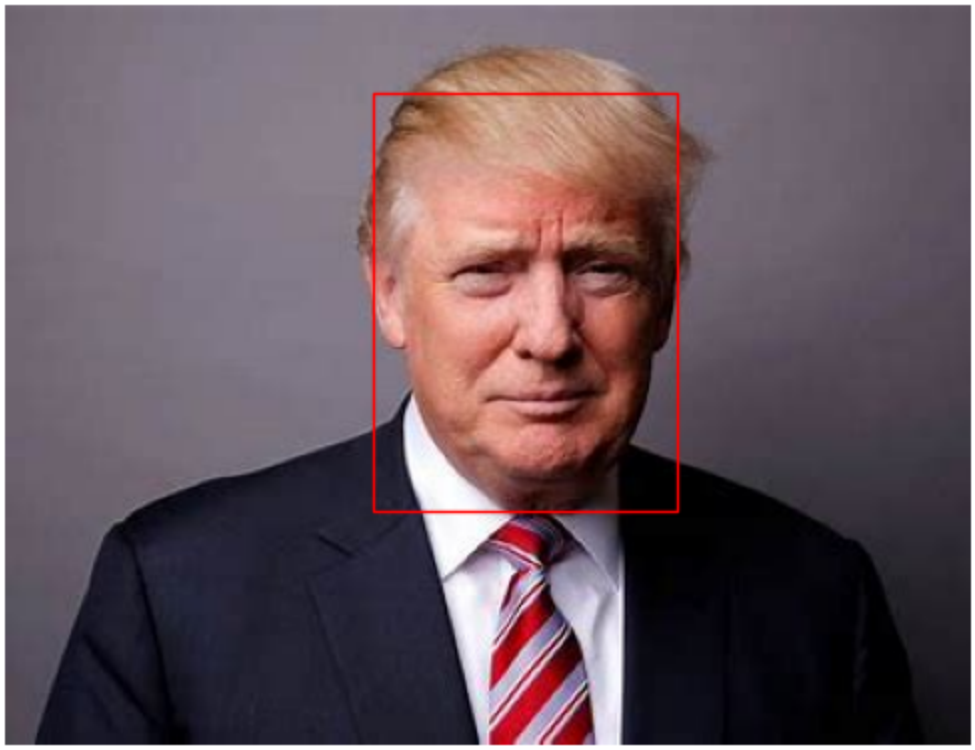
```
out1 = run_model(interpreter, c_1_f)
out2 = run_model(interpreter, c_2_f)
out3 = run_model(interpreter, b_1_f)
out4 = run_model(interpreter, b_2_f)
out5 = run_model(interpreter, cyhh_1_f)
out6 = run_model(interpreter, cyhh_2_f)
out7 = run_model(interpreter, haoyu_1_f)
out8 = run_model(interpreter, haoyu_1_f)
# # Call the function
# out1 = my_function(out1)
# out2 = my_function(out2)
# out3 = my_function(out3)
# out4 = my_function(out4)

print("dist between same person 1 = " + str(euclidean_distance(out1, out2)))
print("dist between same person 2 = " + str(euclidean_distance(out3, out4)))
print("dist between different images 1 = " + str(euclidean_distance(out1, out3)))
print("dist between different images 2 = " + str(euclidean_distance(out2, out4)))
print("dist between same person 3 = " + str(euclidean_distance(out5, out6)))
print("dist between same person 4 = " + str(euclidean_distance(out7, out8)))
print("dist between different images 3 = " + str(euclidean_distance(out5, out7)))
print("dist between different images  4= " + str(euclidean_distance(out6, out8)))
```

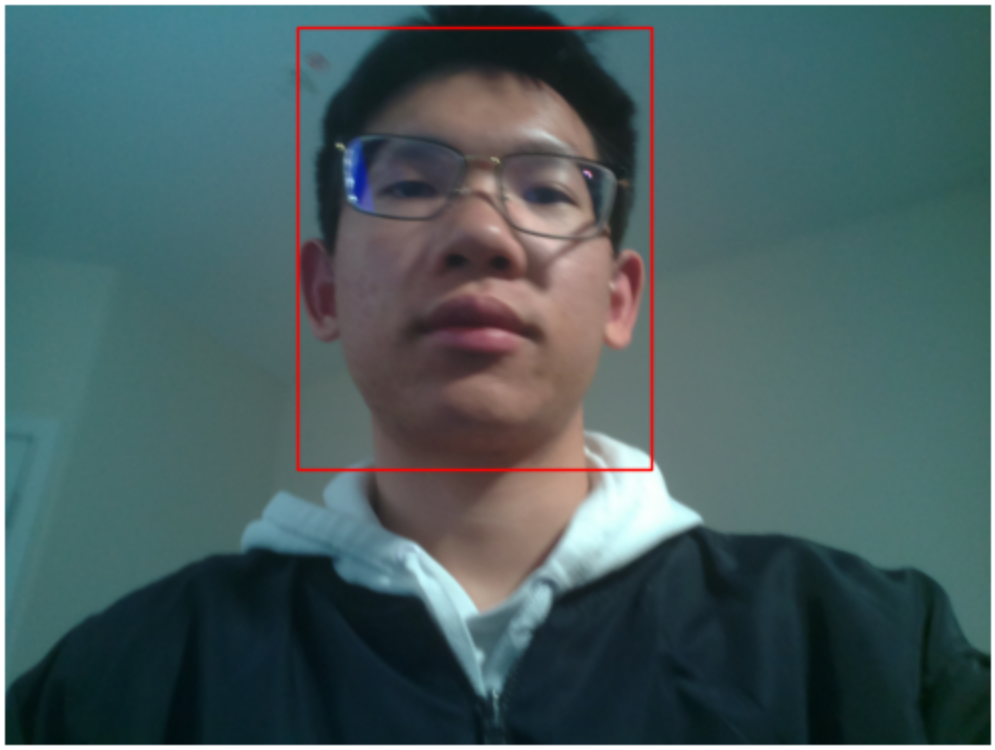
on_or_tensor_args (https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/python/tf/function) for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function _make_execution_function.<locals>.distributed_function at 0x000001D0468A00D8> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings is likely due to passing python objects instead of tensors. Also, tf.function has experimental_relax_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. Please refer to https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args (https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/python/tf/function) for more details.

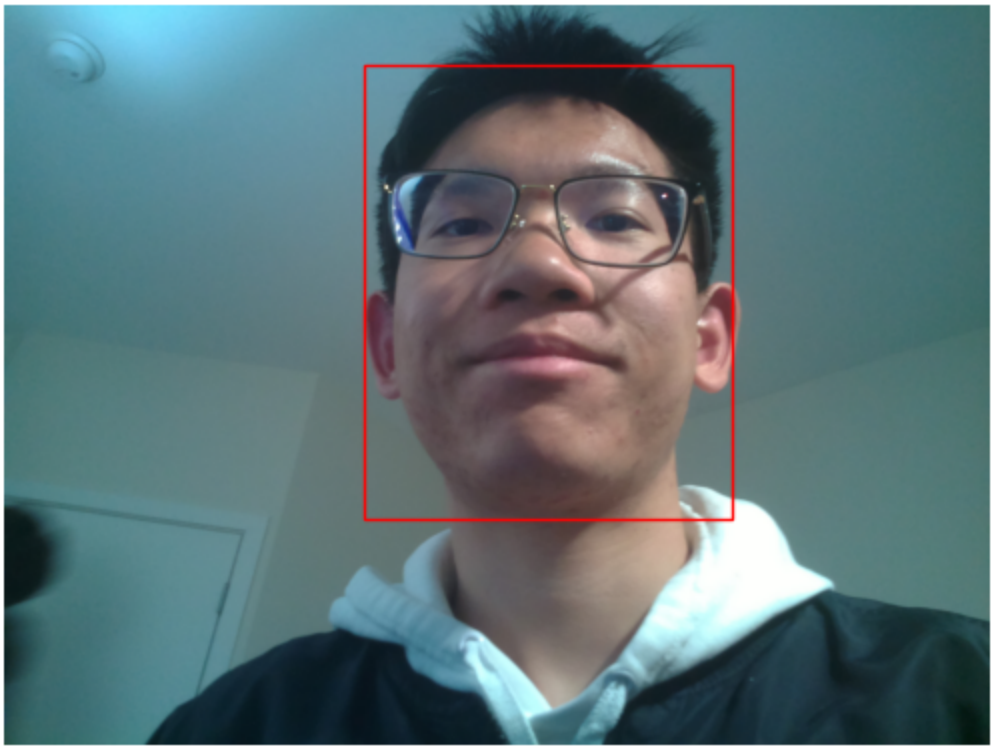
WARNING:tensorflow:11 out of the last 11 calls to <function _make_execution_function.<locals>.distributed_function at 0x000001D0468A00D8> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings is likely due to passing python objects instead of tensors. Also, tf.function has experimental_relax_shapes=True option that relaxes argument shapes that can avoid unnecessary retracing. Please refer to https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args (https://www.tensorflow.org/tutorials/customization/performance#python_or_tensor_args) and https://www.tensorflow.org/api_docs/python/tf/function (https://www.tensorflow.org/api_docs/python/tf/function) for more details.

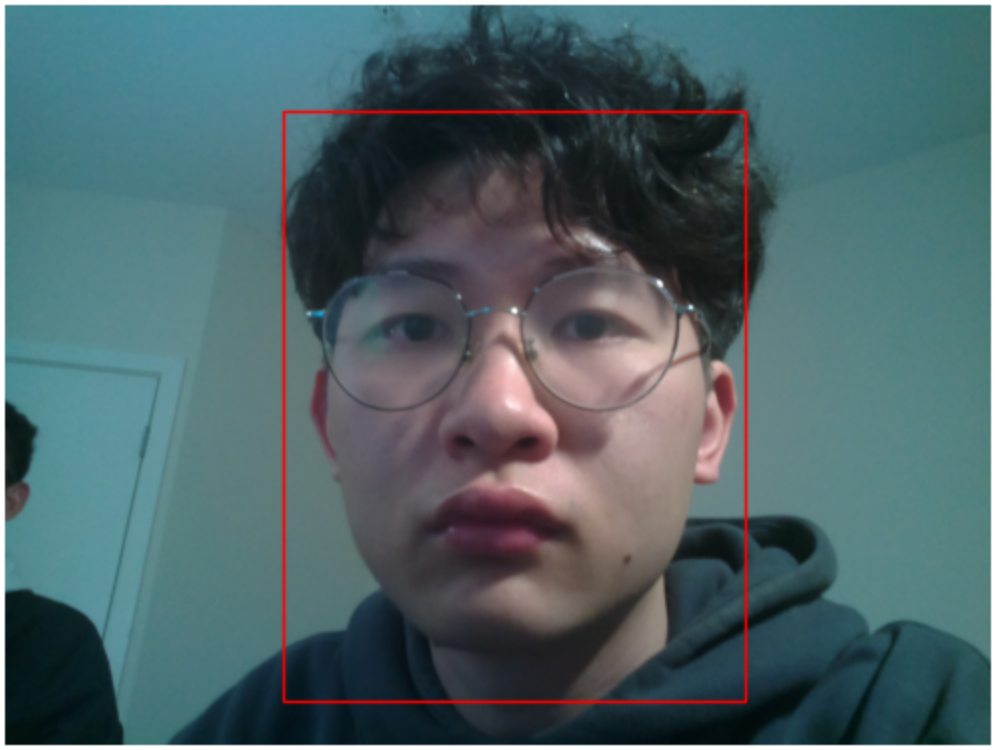
<Figure size 640x480 with 0 Axes>











```
dist between same person 1 = 0.9495734
dist between same person 2 = 0.6818495
dist between different images 1 = 1.1140394
dist between different images 2 = 1.1105659
dist between same person 3 = 0.5042385
dist between same person 4 = 0.0
dist between different images 3 = 0.94435203
dist between different images  4= 0.9026426
```

Report Requirements & Grading (10 pts)

You need to turn in a report and the code. You may turn in a separately-written report, but **in general, we recommend that you directly use the provided Jupyter Notebook and turn it in as your report.** For the first half of Part III, you don't need to implement all the code in the notebook, but you can run the load model step in the notebooks to demonstrate the result.

Part 1: (2 pts)

- Show the command/code and output for getting the following (0.5 points each):
  1. The CPU spec
  2. The network configuration
  3. The TensorFlow checks (version, GPU, etc.)
- You can make screenshots of both the command/code and the resulting output or paste them directly into the report or the notebook.

Part 2: (4 pts)

- For each of the steps, briefly explain your approach and include the key outputs.

- Show the training and testing accuracy of your model. For full credit, the testing accuracy should be over 85% and training accuracy should be over 90%. (2 Points)
- Show the plots of train and validation loss and accuracy, similar to the ones shown before. (1 Point)
- Show the accuracy and inference time of the TFLite models quantized with the two methods with and without the USB accelerator. Compare and comment on what you observed in the report. There might be no significant improvements in terms of performance, but you should try to understand and explain why in your report. (1 Point)

Part 3: (4 pts)

- For each of the steps, briefly explain your approach and include the key outputs.
- Show the output of loading weights for your model to confirm that your model is correctly implemented. You can get partial credit if you cannot load the model correctly. (2 Points)
- Show the bounding box of the face detected by MTCNN in the captured image, as described in the 'Face Detection' section of Part III. (1 Point)
- Report the distances between pairs of pictures, picked from 4 pictures: 2 of yourself and 2 of other people (6 pairs in total). Also, show all 4 pictures in your report. (1 Point)

Notes:

- Besides the above requirements, your report should also include the following details:
  - Your full name and your NetID.
  - The difficulties/bugs you encountered and how you solved them
  - What you learned from this lab
- Your report should cover all the required information, but please keep your report clean and concise.
- Please add references to your report if you referred to any resources when you work on your lab.
- Please submit your **report** to Canvas

Full name and NetID:

1. Yanghonghui Chen / yc47

2. Haoyu Huang / haoyuh3

Haoyu Huang:

1. Bug1 : Lab2_part2. When we deploy the network on raspberry pi, we have no idea how to test its accuracy and time. I solve this problem by save the test image as npy file and transfer to raspberry pi. We have poor performance with TPU at first which result in same output as the device without TPU. We solve this problem by changing the input shape of data. Because TPU is capable of running multiple images simultaneously in a batch.

2. Bug2 : Lab2_part3. I get zero output at first. I solve this by checking the neural network and forget to add activation='relu' in conv2d.

Yanghonghui Chen:

1. Difficulty 1: In part II, when I tried to test the accuracy of the model, it reported the accuracy of only 10%-30%. Then, I tried to normalize the input images, tune the batch size from 200 to 64,and tune the number of epoches from 10 to 20, which largely improved the accuracy.
2. Difficulty 2: In part III, when we wanted to verify the function of the Edge TPU, the inference time did'nt change much. Later, we changed the inference batch size from 1 to 20, the inference time with TPU became faster than that without TPU.

Conclusion:

Yanghonghui Chen: We learned how to build a complicated neural network like GoogleNet from scratch not only via layer by layer but also block by block. We learned how to deploy the large NN model to the edge devices like Raspberry Pi. This involves process like saveing model, model quantization and local inference on edge devices.

Haoyu Huang: In this lab, we have learned how to deploy neural network on raspberry pi. In part1, we have learn how to setup tensorflow on raspberry pi. In part2 we design our network and try to train it with different parameter. More importantly, we converted the model to TFLite model which is crucial for edge devices. Besides, we combine edge TPU with raspberry pi and improve its performance. In part3, we learn a different way to construct our network. We are capable of deploying mtcnn and face recognition on the raspberry pi. It is a good practice to combine picamera and neural network.

In [ ]: