INTRODUCTION TO COMPUTER NETWORKS

19AIE211

श्रद्धावान् लभते ज्ञानम्

AMRITA
VISHWA VIDYAPEETHAM
UNIVERSITY

SEMESTER-4 PROJECT

GESTURE RECOGNITION

(UNDER THE GUIDANCE OF MR.PREMJI)

By:

A.SRINIVAS	**CB.EN.U4AIE19010**
D.KARTHIK	**CB.EN.U4AIE19020**
KAUSHIK M	**CB.EN.U4AIE19036**
VIJAI SIMMON	**CB.EN.U4AIE19068**

ACKNOWLEDGEMENT

I would like to extend my deepest gratitude to our professor Mr. Premjith gave us the golden opportunity to work on this project on the topic of Gesture Recognition(Using Tensorflow Lite and CNN), and who have helped me in completing this project, and it also helped us in doing a lot of research and gaining immense knowledge about the subject. The completion of this project gave us much pleasure.

ABSTRACT

We explore the benefit that users in several application areas can experience from the “Gesture Recognition” function. We will see the theory behind Hand Gesture Recognition and we will consider the collection of near-infrared images of ten distinct hand gestures from The Hand Gesture Recognition Database is a. FInally we will see how we use end-to-end deep learning to build a classifier for these images.

Table of Contents

INTRODUCTION

Gesture recognition is a type of perceptual computing user interface that allows computers to capture and interpret human gestures as commands. The general definition of gesture recognition is the ability of a computer to understand gestures and execute commands based on those gestures.

In order to understand how gesture recognition works, it is important to understand how the word "gesture" is defined. In its most general sense, the word gesture can refer to any non-verbal communication that is intended to communicate a specific message. In the world of gesture recognition, a gesture is defined as any physical movement, large or small, that can be interpreted by a motion sensor. It may include anything from the pointing of a finger to a roundhouse kick or a nod of the head to a pinch or wave of the hand. Gestures can be broad and sweeping or small and contained. In some cases, the definition of "gesture" may also include voice or verbal commands.

Working of Gesture Recognition

Gesture recognition is an alternative user interface for providing real-time data to a computer. Instead of typing with keys or tapping on a touch screen, a motion sensor perceives and interprets movements as the primary source of data input. This is what happens between the time a gesture is made and the computer reacts.

- A camera feeds image data into a sensing device that is connected to a computer. The sensing device typically uses an infrared sensor or projector for the purpose of calculating depth,

- Specially designed software identifies meaningful gestures from a predetermined gesture library where each gesture is matched to a computer command.

- The software then correlates each registered real-time gesture, interprets the gesture and uses the library to identify meaningful gestures that match the library.

- Once the gesture has been interpreted, the computer executes the command correlated to that specific gesture.

For instance, Kinect looks at a range of human characteristics to provide the best command recognition based on natural human inputs. It provides both skeletal and facial tracking in addition to gesture recognition, voice recognition and in some cases the depth and color of the background scene. Kinect reconstructs all of this data into printable three-dimensional (3D) models. The latest Kinect developments include an adaptive user interface that can detect a user's height.

Hand Gesture Recognition

Hand gestures are the most common forms of communication and have great importance in our world. They can help in building safe and comfortable user interfaces for a multitude of applications. Various computer vision algorithms have employed color and depth cameras for hand gesture recognition, but robust classification of gestures from different subjects is still challenging. We did an algorithm for real-time hand gesture recognition using convolutional neural networks (CNNs). The proposed CNN achieves an average accuracy of 98.76% on our respective dataset.

Gestures Considered In Our Dataset

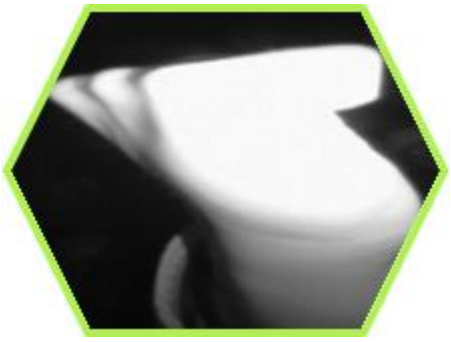
Our dataset consists of 10 hand gestures and 200 images for each gesture comprising a total of 2000 infrared images.

1. LETTER I:

2. PALM:

3. FIST:

4. FIST MOVED:

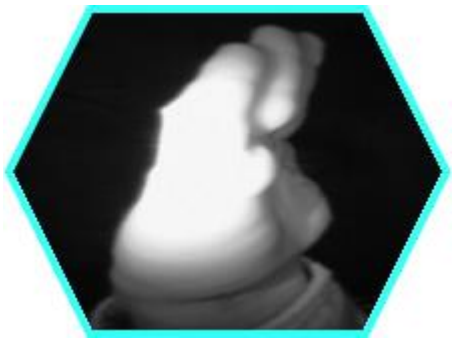


5. THUMB:

6. INDEX:

7. OK:

8. PALM MOVED :

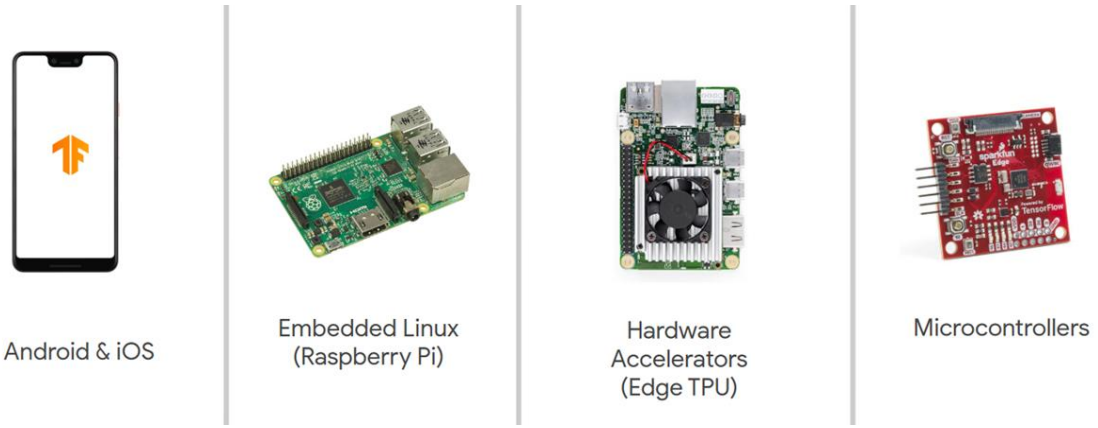


9. LETTER C:

10. HAND DOWNSHIFT:

TensorFlow lite

TensorFlow Lite is an open-source, product ready, cross-platform deep learning framework that converts a pre-trained model in TensorFlow to a special format that can be optimized for speed or storage.



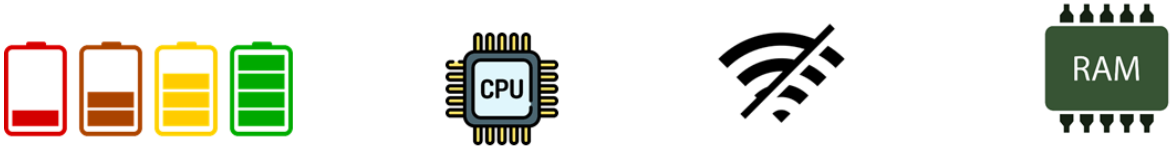
Embedded devices

An embedded device is an object that contains a special-purpose computing system. The system, which is completely enclosed by the object, may or may not be able to connect to the Internet.

Devices that can connect to the Internet are called smart or intelligent. If an embedded device can not connect to the Internet, it is called dumb.

Limitations of embedded devices

Embedded systems have limited computing resources and strict power requirements, writing software for embedded devices is a very specialized field that requires knowledge of both hardware components and programming.



Main Objectives for building a ML model

- Model size

Need a small model size to reduce possible download time and RAM usage

- Latency, Battery and heat

Need to reduce amount of computations needed for inference

- Other constraints

No UI or user interaction

Slow computational power

Steps to use a model on the embedded device

- Train and save a model (development machine)
- Convert the model (development machine)
- Copy the converted model on the device
- Run inference with the TF Lite interpreter

Main reason to convert a model

- Make it smaller (smaller memory footprint)
- Make inference more efficient
- Making Neural Nets Work With Low Precision
- Require less memory access
- Use less energy in inference

To achieve those goals the main component is what is called **QUANTIZATION**

QUANTIZATION

Two types of quantization

- **Post-training quantization**

More simple and easier to implement and in most cases extremely efficient

- **Training-aware quantization**

More complex that require the re-writing of the computational graph

Post-training quantization

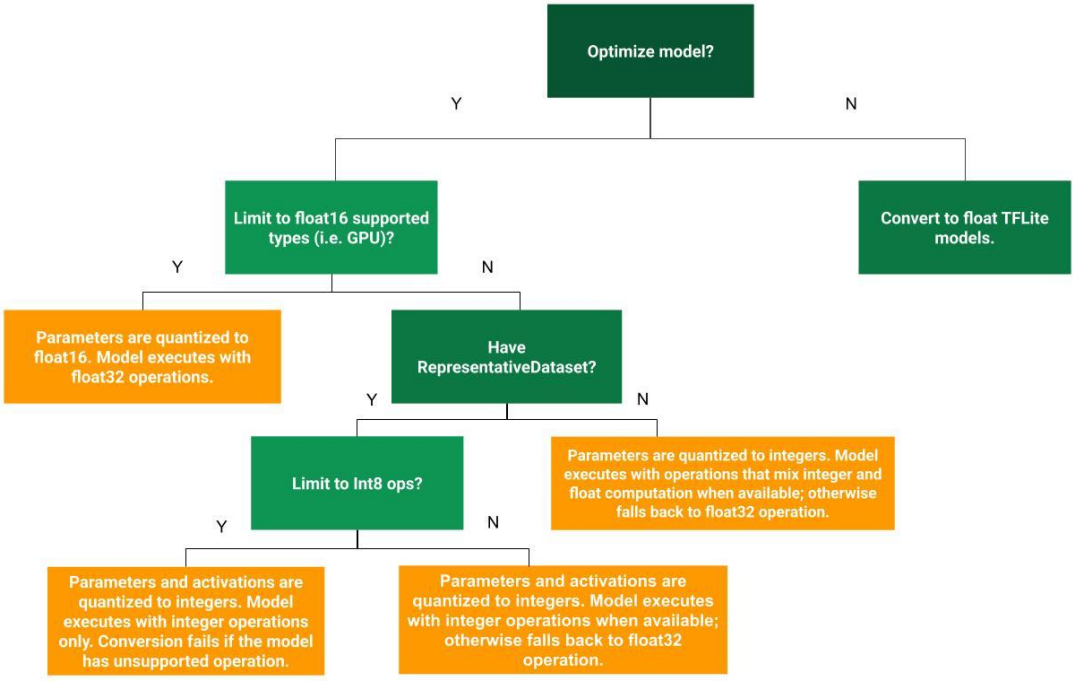
Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. You can quantize an already-trained float TensorFlow model when you convert it to TensorFlow Lite format using the TensorFlow Lite Converter.

Optimization Methods

There are several post-training quantization options to choose from. Here is a summary table of the choices and the benefits they provide:

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

The following decision tree can help determine which post-training quantization method is best for your use case:



1. Dynamic range quantization

The simplest form of post-training quantization statically quantizes only the weights from floating point to integer, which has 8-bits of precision

At inference, weights are converted from 8-bits of precision to floating point and computed using floating-point kernels. This conversion is done once and cached to reduce latency.

To further improve latency, "dynamic-range" operators dynamically quantize activations based on their range to 8-bits and perform computations with 8-bit weights and activations. This optimization provides latencies close to fully fixed-point inference. However, the outputs are still stored using floating point so that the speedup with dynamic-range ops is less than a full fixed-point computation.

2. Full integer quantization

Integer quantization is a technique for converting 4-byte or 8-byte floating-point numbers into single-byte integers. These integers not only require less memory: arithmetic with integers can run quickly on any general-purpose CPU. ... This is done by running several examples of input data through the floating-point model.

3. Float 16 quantization

Post-training float16 quantization reduces TensorFlow Lite model sizes (up to 50%), while sacrificing very little accuracy. It quantizes model constants (like weights and bias values) from full precision floating point (32-bit) to a reduced precision floating point data type

Implementation In Python

In the implementation we have two parts – 1st part deals with converting our image dataset to .npy format. The code for the same is given below.

```
# Importing the required packages
import glob
import os
from PIL import Image
import numpy as np
```

```
# Function used to scale image data ndarray between 0 and 1
def normalize(x):
    return np.array((x - np.min(x)) / (np.max(x) - np.min(x)))
```

```
# Function used to load the images from the directory
def load_image_from_directory(path,size,img_tot):
    #img_array will store all images corresponding to images in a particular folder
    img_array=[]
    path=os.path.join(path,"*.*")
    paths=glob.glob(path)
    for i in range(0,img_tot):
        img=Image.open(paths[i])
        img=img.resize(size)
        img=np.asarray(img,dtype=np.float16)
        img=normalize(img)
        img=np.expand_dims(img, axis=0)
        if len(img_array)==0:
            img_array=img
        else:
            img_array=np.concatenate((img_array,img),axis=0)

    return img_array
```

```
# Total images that we consider for each category
total_img_per_category = 200
```

```
path = 'leapGestRecog/01_palm'
size = (320, 120)
Palm_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Palm_data.npy', Palm_data)
```

```
path = 'leapGestRecog/02_l'
size = (320, 120)
Lshape_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Lshape_data.npy', Lshape_data)
```

```
path = 'leapGestRecog/03_fist'
size = (320, 120)
Fist_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Fist_data.npy', Fist_data)
```

```
path = 'leapGestRecog/04_fist_moved'
size = (320, 120)
Fistm_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Fistm_data.npy', Fistm_data)
```

```
path = 'leapGestRecog/05_thumb'
size = (320, 120)
Thumb_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Thumb_data.npy', Thumb_data)
```

```
path = 'leapGestRecog/06_index'
size = (320, 120)
Index_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Index_data.npy', Index_data)
```

```
path = 'leapGestRecog/07_ok'
size = (320, 120)
Ok_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Ok_data.npy', Ok_data)
```

```
path = 'leapGestRecog/08_palm_moved'
size = (320, 120)
Palm_m_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Palm_m_data.npy', Palm_m_data)
```

```
path = 'leapGestRecog/09_c'
size = (320, 120)
C_data = load_image_from_directory(path, size, total_img_per_category)
np.save('C_data.npy', C_data)
```

```
path = 'leapGestRecog/10_down'
size = (320, 120)
Down_data = load_image_from_directory(path, size, total_img_per_category)
np.save('Down_data.npy', Down_data)
```

The 2nd part is the main one where we implement our gesture recognition model with CNN. The pre-requisites are that, we have to upload the obtained .npy files to our google drive in order to give the input dataset to train and test the model. The code for the CNN model and the TensorFlow lite conversion based on quantization is given below.

ICN Project - Creating a TensorflowLite Model for Gesture Recognition using CNN

```
# Mounting google drive to the kernel
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[2] # Importing the required packages
import numpy as np
import os
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import keras
import tensorflow
from tensorflow.keras.utils import to_categorical
from keras import layers
from keras import models
from sklearn.model_selection import train_test_split
from keras.callbacks import ModelCheckpoint
from keras.callbacks import EarlyStopping
import tensorflow as tf
import pathlib
```

```
[3] # Defining the path to access the files
    path = '/content/drive/MyDrive/ICN-Dataset'
    # Defining all the categorical names
    categories = ["C_data", "Down_data", "Fist_data", "Fistm_data", "Index_data", "Lshape_data", "Ok_data", "Palm_m_data", "Palm_data", "Thumb_data"]
```

```
[4] # Total no. of categories
    total_categories = len(categories)
    # The dataset for each categories accessed
    x1 = np.load(path+'/'+categories[0]+'.npy')
    x2 = np.load(path+'/'+categories[1]+'.npy')
    x3 = np.load(path+'/'+categories[2]+'.npy')
    x4 = np.load(path+'/'+categories[3]+'.npy')
    x5 = np.load(path+'/'+categories[4]+'.npy')
    x6 = np.load(path+'/'+categories[5]+'.npy')
    x7 = np.load(path+'/'+categories[6]+'.npy')
    x8 = np.load(path+'/'+categories[7]+'.npy')
    x9 = np.load(path+'/'+categories[8]+'.npy')
    x10 = np.load(path+'/'+categories[9]+'.npy')
    # Concatenating all the dataset accessed for each category
    x_data = np.concatenate((x1,x2,x3,x4,x5,x6,x7,x8,x9,x10),axis=0)
```

```
    # Total no. of images
    len(x_data)
```

```
    2000
```

```
[6] # Reshaping the data
    x_data = x_data.reshape((len(x_data), 120, 320, 1))
```

```
[7] # Total images per category
    data_each_type = int(len(x_data)/total_categories)
    # Defining or assigning values to the categorical data
    y_data = np.zeros((1,data_each_type))
    for i in range(0,total_categories-1):
      a = np.ones((1,data_each_type))*(i+1)
      y_data = np.append(y_data,a)
    # len(y_data)
```

```
    # Creating the sequential CNN model
    model=models.Sequential()
    model.add(layers.Conv2D(32, (5, 5), strides=(2, 2), activation='relu', input_shape=(120, 320,1)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))
```

```
[9] # Obtaining the dataset for training and testing using train_test_split function
    x_train,x_test,y_train,y_test = train_test_split(x_data,y_data,test_size = 0.4, random_state = 0)
```

```
[10] # Early stopping used in order to find out the best epoch value to obtain the best model
     es = EarlyStopping(monitor = 'val_loss', patience = 20, verbose = 1)
     mc =  ModelCheckpoint('/content/drive/MyDrive/ICN-Dataset/best_model.h5', monitor='val_loss', verbose=1, save_best_only=True)
     # Categorical cross entropy used as loss function for the model
     model.compile(optimizer='rmsprop',
                   loss='sparse_categorical_crossentropy',
                   metrics=['accuracy'])
     # Fitting the dataset to model with 50 epochs and batch size of 50
     model.fit(x_train, y_train, validation_data = (x_test, y_test),
               epochs = 5, batch_size = 50, callbacks = [es, mc])
```

```
Epoch 1/5
24/24 [==============================] - 33s 789ms/step - loss: 1.6830 - accuracy: 0.4319 - val_loss: 0.0547 - val_accuracy: 0.9875

Epoch 00001: val_loss improved from inf to 0.05472, saving model to /content/drive/MyDrive/ICN-Dataset/best_model.h5
Epoch 2/5
24/24 [==============================] - 18s 767ms/step - loss: 0.0958 - accuracy: 0.9756 - val_loss: 0.0073 - val_accuracy: 0.9987

Epoch 00002: val_loss improved from 0.05472 to 0.00729, saving model to /content/drive/MyDrive/ICN-Dataset/best_model.h5
Epoch 3/5
24/24 [==============================] - 18s 767ms/step - loss: 0.0288 - accuracy: 0.9959 - val_loss: 0.0070 - val_accuracy: 0.9987

Epoch 00003: val_loss improved from 0.00729 to 0.00700, saving model to /content/drive/MyDrive/ICN-Dataset/best_model.h5
Epoch 4/5
24/24 [==============================] - 18s 758ms/step - loss: 0.0076 - accuracy: 0.9984 - val_loss: 0.0011 - val_accuracy: 1.0000

Epoch 00004: val_loss improved from 0.00700 to 0.00110, saving model to /content/drive/MyDrive/ICN-Dataset/best_model.h5
Epoch 5/5
24/24 [==============================] - 18s 758ms/step - loss: 0.0075 - accuracy: 0.9994 - val_loss: 0.0016 - val_accuracy: 0.9987

Epoch 00005: val_loss did not improve from 0.00110
<keras.callbacks.History at 0x7f22376a5650>
```

```
[11] #model.compile(optimizer='rmsprop',
                 # loss='categorical_crossentropy',
                  #metrics=['accuracy'])
     #model.fit(x_data,y_data, epochs=20)
```

```
[12] # We load out the best model that has been stored in our directory
     model = tf.keras.models.load_model('/content/drive/MyDrive/ICN-Dataset/best_model.h5')
```

```
     # Evaluating the accuracy and loss values for the best model
     [loss, acc] = model.evaluate(x_test,y_test,verbose=1)
     print("Accuracy:" + str(acc))
```

```
     25/25 [==============================] - 3s 104ms/step - loss: 0.0011 - accuracy: 1.0000
     Accuracy:1.0
```

```
[14] # Converted model with no quantization
     converter = tf.lite.TFLiteConverter.from_keras_model(model)
     tflite_model = converter.convert()
```

```
     INFO:tensorflow:Assets written to: /tmp/tmpk7uxyo8_/assets
```

```
[15] # Convert using dynamic range quantization
     converter = tf.lite.TFLiteConverter.from_keras_model(model)
     converter.optimizations = [tf.lite.Optimize.DEFAULT]

     tflite_model_quant1 = converter.convert()
```

```
     INFO:tensorflow:Assets written to: /tmp/tmpf3i7liok/assets
     INFO:tensorflow:Assets written to: /tmp/tmpf3i7liok/assets
```

```
     # Convert using float fallback quantization
     def representative_data_gen():
       for input_value in tf.data.Dataset.from_tensor_slices(x_data.astype('float32')).batch(1).take(100):
         # Model has only one input so each data point has one element.
         yield [input_value]

     # Converting the model to tensorflowlite model
     converter = tf.lite.TFLiteConverter.from_keras_model(model)
     converter.optimizations = [tf.lite.Optimize.DEFAULT]
     converter.representative_dataset = representative_data_gen

     tflite_model_quant2 = converter.convert()
```

```
     INFO:tensorflow:Assets written to: /tmp/tmpb9sg4duj/assets
     INFO:tensorflow:Assets written to: /tmp/tmpb9sg4duj/assets
```

```
[18] # Printing the parameters of the quantized model
     interpreter = tf.lite.Interpreter(model_content=tflite_model_quant2)
     input_type = interpreter.get_input_details()[0]['dtype']
     print('input: ', input_type)
     output_type = interpreter.get_output_details()[0]['dtype']
     print('output: ', output_type)
```

```
     input:  <class 'numpy.float32'>
     output:  <class 'numpy.float32'>
```

```
[19] # Convert using integer-only quantization
     def representative_data_gen():
       for input_value in tf.data.Dataset.from_tensor_slices(x_data.astype('float32')).batch(1).take(100):
         yield [input_value]

     converter = tf.lite.TFLiteConverter.from_keras_model(model)
     converter.optimizations = [tf.lite.Optimize.DEFAULT]
     converter.representative_dataset = representative_data_gen
     # Ensure that if any ops can't be quantized, the converter throws an error
     converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
     # Set the input and output tensors to uint8
     converter.inference_input_type = tf.uint8
     converter.inference_output_type = tf.uint8

     tflite_model_quant3 = converter.convert()
```

```
     INFO:tensorflow:Assets written to: /tmp/tmpa7140o0r/assets
     INFO:tensorflow:Assets written to: /tmp/tmpa7140o0r/assets
     WARNING:absl:For model inputs containing unsupported operations which cannot be quantized, the `inference_input_type` attribute will default to the original type.
```

```
[20] interpreter = tf.lite.Interpreter(model_content=tflite_model_quant3)
     input_type = interpreter.get_input_details()[0]['dtype']
     print('input: ', input_type)
     output_type = interpreter.get_output_details()[0]['dtype']
     print('output: ', output_type)

     input:  <class 'numpy.uint8'>
     output:  <class 'numpy.uint8'>
```

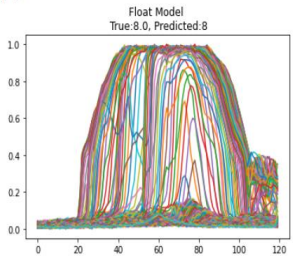
```
[21] tflite_models_dir = pathlib.Path("/content/drive/MyDrive/ICN-Dataset")
     tflite_models_dir.mkdir(exist_ok=True, parents=True)

     # Save the unquantized/float model:
     tflite_model_file = tflite_models_dir/"gesture_model.tflite"
     tflite_model_file.write_bytes(tflite_model)
     # Save the quantized model:
     tflite_model_quant_file1 = tflite_models_dir/"gesture_model_quant_dyn.tflite"
     tflite_model_quant_file1.write_bytes(tflite_model_quant1)
     tflite_model_quant_file2 = tflite_models_dir/"gesture_model_quant_float.tflite"
     tflite_model_quant_file2.write_bytes(tflite_model_quant2)
     tflite_model_quant_file3 = tflite_models_dir/"gesture_model_quant_int.tflite"
     tflite_model_quant_file3.write_bytes(tflite_model_quant3)

     804664
```

```
[21] # run the TensorFlow Lite models
     # Helper function to run inference on a TFLite model
     def run_tflite_model(tflite_file, test_image_indices):
       global x_test

       # Initialize the interpreter
       interpreter = tf.lite.Interpreter(model_path=str(tflite_file))
       interpreter.allocate_tensors()

       input_details = interpreter.get_input_details()[0]
       output_details = interpreter.get_output_details()[0]

       predictions = np.zeros((len(test_image_indices),), dtype=int)
       for i, test_image_index in enumerate(test_image_indices):
         test_image = x_test[test_image_index]
         test_label = y_test[test_image_index]

         # Check if the input type is quantized, then rescale input data to uint8
         if input_details['dtype'] == np.uint8:
           input_scale, input_zero_point = input_details["quantization"]
           test_image = test_image / input_scale
         test_image = np.expand_dims(test_image, axis=0).astype(input_details["dtype"])
         interpreter.set_tensor(input_details["index"], test_image)
         interpreter.invoke()
         output = interpreter.get_tensor(output_details["index"])[0]
         predictions[i] = output.argmax()

       return predictions
```

```
[23] # test the models on one image
     # Change this to test a different image
     test_image_index = 600

     ## Helper function to test the models on one image
     def test_model(tflite_file, test_image_index, model_type):
       global y_test
       predictions = run_tflite_model(tflite_file, [test_image_index])
       plt.plot(x_test[test_image_index].reshape((x_test[test_image_index].shape[0],x_test[test_image_index].shape[1])))
       template = model_type + " Model \n True:{true}, Predicted:{predict}"
       _ = plt.title(template.format(true= str(y_test[test_image_index]), predict=str(predictions[0])))
       plt.grid(False)
       print(predictions)
```

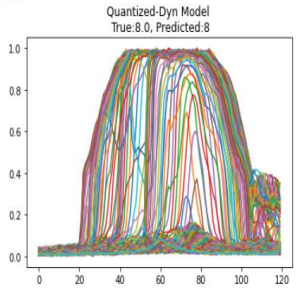
```
# test the float model
test_model(tflite_model_file, test_image_index, model_type="Float")
```

```
[8]
```



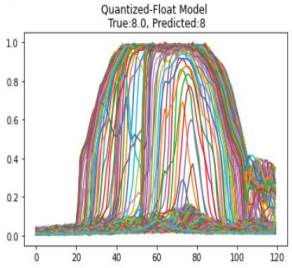
```
[25] # test the quantized model
     test_model(tflite_model_quant_file1, test_image_index, model_type="Quantized-Dyn")
```

[8]



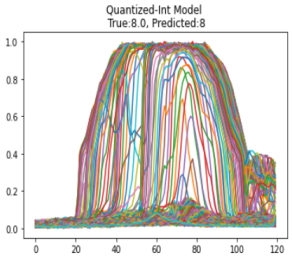

```
[26] # test the quantized model
     test_model(tflite_model_quant_file2, test_image_index, model_type="Quantized-Float")
```

[8]




```
[27] # test the quantized model
     test_model(tflite_model_quant_file3, test_image_index, model_type="Quantized-Int")
```

[8]




```
[28] # evaluate the models on all images
     def evaluate_model(tflite_file, model_type):
       global x_test
       global y_test

       test_image_indices = range(x_test.shape[0])
       predictions = run_tflite_model(tflite_file, test_image_indices)

       accuracy = (np.sum(y_test== predictions) * 100) / len(x_test)

       print('%s model accuracy is %.4f%% (Number of test samples=%d)' % (
           model_type, accuracy, len(x_test)))
```

```
[29] # evaluate the float model
     evaluate_model(tflite_model_file, model_type="Float")

     Float model accuracy is 100.0000% (Number of test samples=800)

[30] # evaluate the Dynamic quantized model
     evaluate_model(tflite_model_quant_file1, model_type="Quantized-Dyn")

     Quantized-Dyn model accuracy is 100.0000% (Number of test samples=800)

[31] # evaluate the Float quantized model
     evaluate_model(tflite_model_quant_file2, model_type="Quantized-Float")

     Quantized-Float model accuracy is 100.0000% (Number of test samples=800)

[32] # evaluate the Integer quantized model
     evaluate_model(tflite_model_quant_file3, model_type="Quantized-Int")

     Quantized-Int model accuracy is 100.0000% (Number of test samples=800)
```

Gesture Recognition Makers

Microsoft is leading the charge with Kinect, a gesture recognition platform that allows humans to communicate with computers entirely through speaking and gesturing. Kinect gives computers, “eyes, ears, and a brain.” There are a few other players in the space such as Soft Kinect, Gesture Tek, Point Grab, eyesight and PrimeSense, an Israeli company recently acquired by Apple. Emerging technologies from companies such as eyesight go far beyond gaming to allow for a new level of small motor precision and depth perception.

Applications of Gesture Recognition

Gesture recognition has huge potential in creating interactive, engaging live experiences. Here are a few gesture recognition examples that illustrate the potential of gesture recognition to educate, simplify user experiences and delight consumers.

1. In-store retail engagement

Gesture recognition has the power to deliver an exciting, seamless in-store experience. This example uses Kinect to create an engaging retail experience by immersing the shopper in relevant content, helping her to try on products and offering a game that allows the shopper to earn a discount incentive.

2. Changing how we interact with traditional computers

A company named Leap Motion last year introduced the Leap Motion Controller, a gesture-based computer interaction system for PC and Mac. A USB device and roughly the size of a Swiss army knife, the controller allows users to interact with traditional computers with gesture control. It's very easy to see the live experience applications of this technology.

3. The operating room

Companies such as Microsoft and Siemens are working together to redefine the way that everyone from motorists to surgeons accomplish highly sensitive tasks. These companies have been focused on refining gesture recognition technology to focus on fine motor manipulation of images and enable a surgeon to virtually grasp and move an object on a monitor.

4. Windshield wipers

Google and Ford are also reportedly working on a system that allows drivers to control features such as air conditioning, windows and windshield wipers with gesture controls. The Cadillac CUE system recognizes some gestures such as tap, flick, swipe and spread to scroll lists and zoom in on maps.

5. Mobile payments

Seeper, a London-based startup, has created a technology called Seemove that has gone beyond image and gesture recognition to object recognition. Ultimately, Seeper believes that their system could allow people to manage personal media, such as photos or files, and even initiate online payments using gestures.

6. Sign language interpreter

There are several examples of using gesture recognition to bridge the gap between the deaf and non-deaf who may not know sign language. This example showing how Kinect can understand and translate sign language from Dani Martinez Capilla explores the notion of breaking down communication barriers using gesture recognition.

7. Gaming Industry

The gaming industry is one of the major contributors to the global market. The gaming industry can be categorized into console gaming and PC gaming. PC gaming does not make much use of motion

control gaming, when compared to consoles. The three major console developers, Microsoft, Sony, and Nintendo, bundle and sell gesture recognition devices along with the console.

Technological advancements have led to the development of virtual reality, augmented reality, and mixed reality headsets, which rely heavily on gesture controls. The number of motion control games is poised to increase further, with the availability of VR and MR headsets.

Microsoft and Sony are two key companies which introduced their gesture recognition products, called the Kinect for Xbox One and PlayStation Move, through which motion games (such as Kinect Sports Rivals) can be played. Owing to this technology, many organizations, game developers, and third-party app developers have started developing games or apps that can support the technology, indirectly boosting the growth of this market.