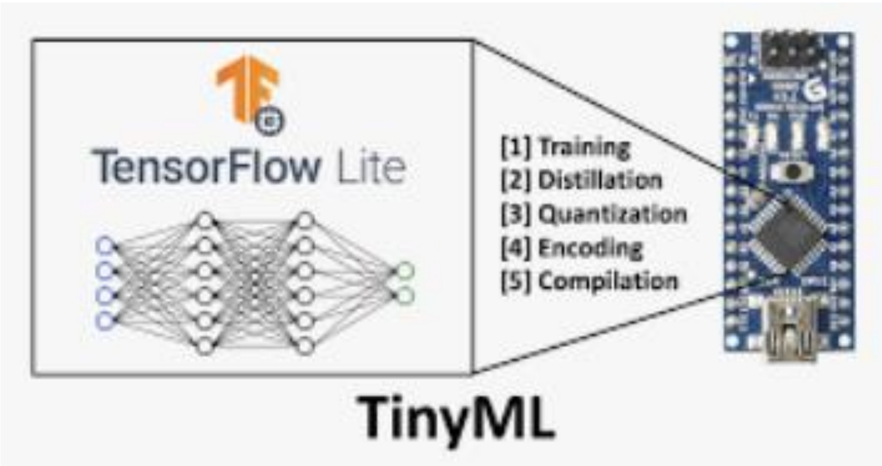
Introduction to Computer Networks
Semester 4 Project Report

Gesture Recognition Using TinyML



Arrun Sivasubramanian (CB.EN.U4.AIE19013)
Gannu Avinash Dora (CB.EN.U4.AIE19028)
P S Sai Ganesh (CB.EN.U4.AIE19044)
Prashanth VR (CB.EN.U4.AIE19047)

Table of Contents

What is an Embedded Device?

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. Embedded systems have extensive applications in consumer, commercial, automotive, industrial and healthcare markets.
Embedded artificial intelligence (AI) is the application of machine and deep learning in software at the device level. Software can be programmed to provide both predictive and reactive intelligence, based on the data that is collected and analyzed.

What is TinyML?

TinyML is a cutting-edge field that brings the transformative power of machine learning (ML) to the performance and power-constrained domain of tiny devices and embedded systems. Successful deployment in this field requires intimate knowledge of applications, algorithms, hardware, and software.

Tiny Machine Learning (TinyML) is the latest embedded software technology is about making computing at the edge cheaper, less expensive and more predictable. Python programming language using TensorFlow (Lite/Micro) is used to power these devices as well as important topics in the responsible design of Artificial Intelligence systems.

Why Traditional DL fails?

Over the past decade, we have witnessed the size of machine learning algorithms grow exponentially due to improvements in processor speeds and the advent of big data. Initially, models were small enough to run on local machines using one or more cores within the CPU.

More recently, we have seen the development of specialized application-specific integrated circuits (ASICs) and tensor processing units (TPUs), which can pack the power of ~8 GPUs. This came to a head recently with the release of the GPT-3 algorithm.

What is TensorFlow Lite?

TensorFlow Lite is a production ready, cross-platform framework for deploying ML on mobile devices and embedded systems.

Some Key features of TensorFlow Lite are as follows:

- Optimized for on-device machine learning, by addressing 5 key constraints: latency (there's no round-trip to a server), privacy (no personal data leaves the device), connectivity (internet connectivity is not required), size (reduced model and binary size) and power consumption (efficient inference and a lack of network connections).

- Multiple platform support, covering Android and iOS devices, embedded Linux, and microcontrollers.
- Diverse language support, which includes Java, Swift, Objective-C, C++, and Python.
- High performance, with hardware acceleration and model optimization.

End-to-end examples, for common machine learning tasks such as image classification, object detection, pose estimation, question answering, text classification, etc. on multiple platforms.

About the Dataset:

The dataset that was used for this project was collected using accelerometer and gyroscope sensors. The total number of gestures recognized is 8.

They are: down_to_up, forward_clockwise, left_fall, up_clockwise, up_anticlockwise, left_to_right, right_to_left, forward_fall.

Each of the sensors returned values along the x, y and z axes respectively. Each gesture had 600 data points, which means the accelerometer and the gyroscope was used 100 times for one gesture.

The final text file contains n rows, in this case 477 rows and 601 columns, 600 columns for the data points and the final column is the gesture name or the class label in this case.

	1	2	3	4	600	Gesture
0	0.003022	0.093937	0.011679	-0.000629	-0.001888	forward_fall
1	0.000798	0.096556	0.020467	-0.000109	-0.000641	right_to_left
2	0.002546	0.090785	0.027262	0.000192	-0.000430	down_to_up
3	0.017293	0.095955	0.019115	-0.000139	-0.000643	left_to_right
4	-0.000055	0.095312	0.020812	-0.000233	-0.000150	down_to_up
...	...	...	...	...	...	...
472	0.012845	0.098365	0.013678	-0.000165	-0.000531	left_to_right
473	-0.002079	0.080596	0.006062	0.000606	0.001311	left_fall
474	0.011863	0.095148	0.028367	-0.000533	-0.003628	right_to_left
475	0.007047	0.093727	0.008762	0.000426	-0.000190	forward_fall
476	0.000861	0.093104	0.010468	-0.000235	-0.004736	forward_clockwise

Snapshot of dataset with 477 rows and 601 columns

The Code for a bigger and smaller model:

Importing necessary Libraries

```
[1] import pandas as pd
    import csv
    import os
    import tensorflow as tf
    import keras
    from sklearn import preprocessing
    from sklearn.preprocessing import normalize, OneHotEncoder
    import numpy as np
    import matplotlib.pyplot as plt
    import urllib
```

Data Pre-Processing

```
[2] gestures = ['down_to_up', 'forward_clockwise', 'left_fall', 'up_clockwise', 'up_anticlockwise', 'left_to_right', 'right_to_left', 'forward_fall']
```

```
[3] df = pd.read_csv('/content/sample_data/gesture60_8_.csv')
```

```
[4] df.describe()
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
count	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.000000	477.
mean	0.118386	9.566627	1.698821	0.011603	0.009892	-0.002908	0.099369	9.565779	1.689824	0.008068	0.008117	-0.003709	0.085160	9.552621	1.
std	0.703841	0.279369	0.831210	0.051517	0.053500	0.046253	0.720297	0.293941	0.845079	0.051266	0.052146	0.044873	0.733245	0.310650	0.
min	-1.933954	8.013336	-0.763738	-0.128932	-0.153798	-0.231234	-2.139846	8.194091	-0.822394	-0.243105	-0.181956	-0.231234	-2.179348	8.182120	-0.
25%	-0.295193	9.464160	1.139570	-0.017051	-0.019408	-0.023593	-0.313149	9.456978	1.138373	-0.020370	-0.019495	-0.023698	-0.345469	9.436628	1.
50%	0.132154	9.620974	1.685425	0.004855	0.003759	-0.004221	0.108213	9.619776	1.671060	0.002703	0.002585	-0.004574	0.108213	9.624564	1.
75%	0.505633	9.749058	2.198959	0.029015	0.031472	0.016665	0.511618	9.763422	2.165442	0.026513	0.029775	0.016894	0.519998	9.764620	2.
max	2.157561	10.012409	5.086242	0.319155	0.253930	0.286086	2.163547	10.017197	5.106592	0.319155	0.253930	0.286086	2.161152	10.072262	5.

8 rows × 600 columns

Normalizing and Shuffling the Data

```
[5] df[df.columns[:-1]] = normalize(df[df.columns[:-1]])
    df = df.sample(frac=1, random_state=13).reset_index(drop = True)
    df
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1
0	0.003022	0.093937	0.011679	-0.000629	0.000282	-0.000210	0.002838	0.093238	0.010267	-0.000629	0.000282	-0.000210	0.004020	0.093685	0.009006	-0.000629	0.00028
1	0.000798	0.096556	0.020467	-0.000109	-0.000561	-0.001020	0.000401	0.095487	0.020298	-0.000109	-0.000561	-0.001020	-0.000404	0.094645	0.020262	-0.000109	-0.00056
2	0.002546	0.090785	0.027262	0.000192	0.000018	-0.000177	0.004073	0.091596	0.029648	0.000192	0.000018	-0.000177	0.003083	0.092061	0.031008	0.000192	0.00001
3	0.017293	0.095955	0.019115	-0.000139	0.000187	-0.000324	0.017862	0.096802	0.019986	-0.000139	0.000187	-0.000324	0.018225	0.097274	0.019998	-0.000139	0.00018
4	-0.000055	0.095312	0.020812	-0.000233	0.000047	0.000046	-0.000883	0.095468	0.022240	-0.000233	0.000047	0.000046	-0.000811	0.095396	0.020680	-0.000233	0.00004
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	.
472	0.012845	0.098365	0.013678	-0.000165	-0.000197	0.000305	0.012250	0.099106	0.013836	-0.000165	-0.000197	0.000305	0.011703	0.099191	0.013338	-0.000165	-0.00019
473	-0.002079	0.080596	0.006062	0.000606	0.000420	-0.000667	-0.002528	0.080146	0.005661	0.000606	0.000420	-0.000667	-0.003105	0.079559	0.006062	0.000606	0.00042
474	0.011863	0.095148	0.028367	-0.000533	-0.000350	-0.000925	0.011118	0.095004	0.028283	-0.000533	-0.000350	-0.000925	0.009076	0.094259	0.028018	-0.000533	-0.00035
475	0.007047	0.093727	0.008762	0.000426	0.000409	-0.000170	0.006576	0.094048	0.008969	0.000426	0.000409	-0.000170	0.005854	0.094335	0.008659	0.000426	0.00040
476	0.000861	0.093104	0.010468	-0.000235	-0.000126	0.000363	0.001809	0.093196	0.009577	-0.000235	-0.000126	0.000363	0.002491	0.092815	0.010317	-0.000235	-0.00012

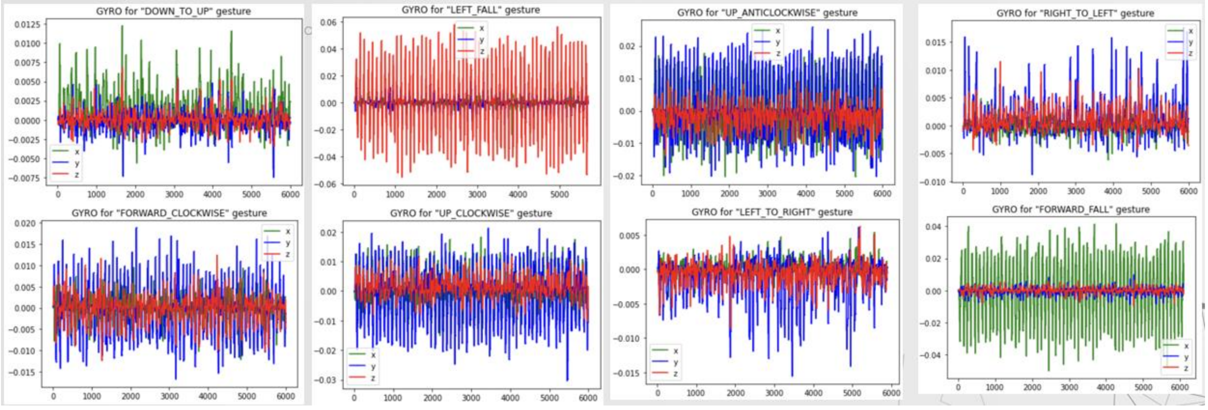
477 rows × 601 columns

```
df.to_csv('gesture60_8_normalized.csv', index = False)
```

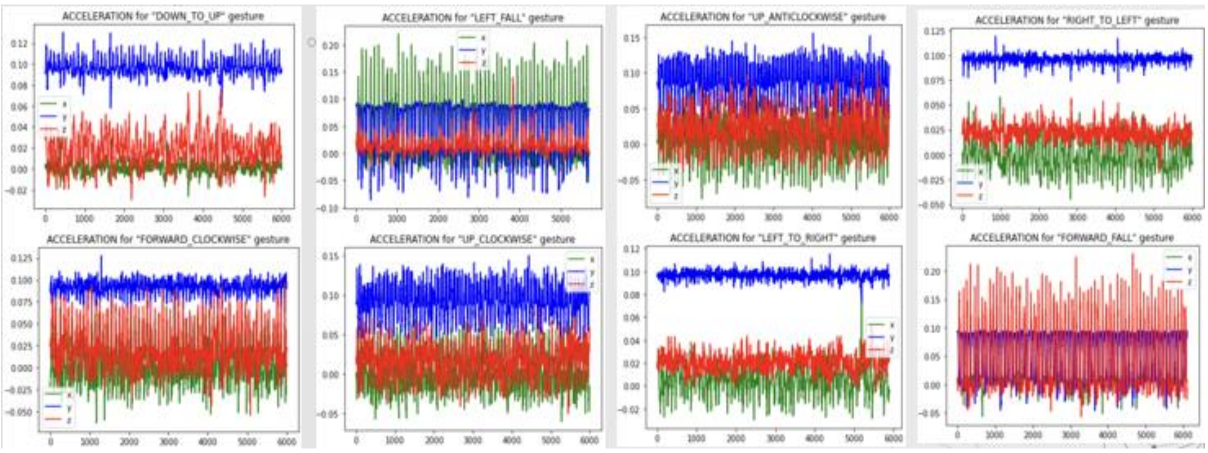
Visualization of the Data

```
[10] def visualize(type, gesture):
         for i in gestures:
             title = type.upper()+' for \"'+i.upper()+'\" gesture'
             plt.title(title)
             if type == 'acceleration':
                 index = range(1, len(data_dict[i]['ax']) + 1)
                 plt.plot(index, data_dict[i]['ax'], 'g.', label='x', linestyle='solid', marker=',')
                 plt.plot(index, data_dict[i]['ay'], 'b.', label='y', linestyle='solid', marker=',')
                 plt.plot(index, data_dict[i]['az'], 'r.', label='z', linestyle='solid', marker=',')

             if type == 'gyro':
                 index = range(1, len(data_dict[i]['ax']) + 1)
                 plt.plot(index, data_dict[i]['gx'], 'g.', label='x', linestyle='solid', marker=',')
                 plt.plot(index, data_dict[i]['gy'], 'b.', label='y', linestyle='solid', marker=',')
                 plt.plot(index, data_dict[i]['gz'], 'r.', label='z', linestyle='solid', marker=',')

             plt.legend()
             plt.show()
```

```
[11] visualize('gyro', gestures)
```




```
[12] visualize('acceleration', gestures)
```



One Hot Encoding the Class Labels

```
[14] y_df = pd.get_dummies(df.Gesture)[gestures]
     print(y_df)
     print(np.array(y_df)[0].shape)
```

```
         down_to_up  forward_clockwise  ...  right_to_left  forward_fall
    0             0                  0  ...              0             1
    1             0                  0  ...              1             0
    2             1                  0  ...              0             0
    3             0                  0  ...              0             0
    4             1                  0  ...              0             0
    ..          ...                ...  ...            ...           ...
    472           0                  0  ...              0             0
    473           0                  0  ...              0             0
    474           0                  0  ...              1             0
    475           0                  0  ...              0             1
    476           0                  1  ...              0             0

    [477 rows x 8 columns]
    (8,)
```

```
[15] df = pd.concat([df, y_df], axis=1)
     df.head()
```

	ax	ay	az	gx	gy	gz	ax	ay	az	gx	gy	gz	ax	ay	az	gx	gy
0	0.003022	0.093937	0.011679	-0.000629	0.000282	-0.000210	0.002838	0.093238	0.010267	-0.000629	0.000282	-0.000210	0.004020	0.093685	0.009006	-0.000629	0.000282
1	0.000798	0.096556	0.020467	-0.000109	-0.000561	-0.001020	0.000401	0.095487	0.020298	-0.000109	-0.000561	-0.001020	-0.000404	0.094645	0.020262	-0.000109	-0.000561
2	0.002546	0.090785	0.027262	0.000192	0.000018	-0.000177	0.004073	0.091596	0.029648	0.000192	0.000018	-0.000177	0.003083	0.092061	0.031008	0.000192	0.000018
3	0.017293	0.095955	0.019115	-0.000139	0.000187	-0.000324	0.017862	0.096802	0.019986	-0.000139	0.000187	-0.000324	0.018225	0.097274	0.019998	-0.000139	0.000187
4	-0.000055	0.095312	0.020812	-0.000233	0.000047	0.000046	-0.000883	0.095468	0.022240	-0.000233	0.000047	0.000046	-0.000811	0.095396	0.020680	-0.000233	0.000047

5 rows × 609 columns

Splitting the Dataset into Testing and Training Datasets

```
[16] train_count = int(0.8*len(df))
     test_count = len(df) - train_count
     print(train_count, test_count)

     x_train, y_train = np.array(df)[:train_count, :600].astype('float32'), np.array(df)[:train_count, -(len(gestures)):].astype('int64')
     x_test, y_test = np.array(df)[train_count:, :600].astype('float32'), np.array(df)[train_count:, -(len(gestures)):].astype('int64')

     print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)

     381 96
     (381, 600) (381, 8) (96, 600) (96, 8)
```

Building the NN Model

```
[18] bigmodel = tf.keras.models.Sequential()
     bigmodel.add(tf.keras.layers.Dense(512, activation="relu", input_shape = (600,)))
     bigmodel.add(tf.keras.layers.Dense(128, activation="relu"))
     bigmodel.add(tf.keras.layers.Dense(64, activation="relu"))
     bigmodel.add(tf.keras.layers.Dense(len(gestures), activation="softmax"))
```

```
[19] bigmodel.summary()

     Model: "sequential"
     _________________________________________________________________
     Layer (type)                 Output Shape              Param #
     =================================================================
     dense (Dense)                (None, 512)               307712
     _________________________________________________________________
     dense_1 (Dense)              (None, 128)               65664
     _________________________________________________________________
     dense_2 (Dense)              (None, 64)                8256
     _________________________________________________________________
     dense_3 (Dense)              (None, 8)                 520
     =================================================================
     Total params: 382,152
     Trainable params: 382,152
     Non-trainable params: 0
     _________________________________________________________________
```

```
[20] bigmodel.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

Training the NN Model

```
[21] bigmodel.fit(x_train, y_train, epochs = 16, batch_size = 8)

     Epoch 1/16
     48/48 [==============================] - 1s 4ms/step - loss: 1.4897 - accuracy: 0.6037
     Epoch 2/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.3828 - accuracy: 0.9318
     Epoch 3/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.1289 - accuracy: 0.9764
     Epoch 4/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0720 - accuracy: 0.9869
     Epoch 5/16
     48/48 [==============================] - 0s 3ms/step - loss: 0.0514 - accuracy: 0.9895
     Epoch 6/16
     48/48 [==============================] - 0s 3ms/step - loss: 0.0540 - accuracy: 0.9869
     Epoch 7/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0372 - accuracy: 0.9895
     Epoch 8/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0182 - accuracy: 0.9974
     Epoch 9/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0096 - accuracy: 1.0000
     Epoch 10/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0071 - accuracy: 1.0000
     Epoch 11/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0072 - accuracy: 1.0000
     Epoch 12/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0038 - accuracy: 1.0000
     Epoch 13/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0031 - accuracy: 1.0000
     Epoch 14/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0026 - accuracy: 1.0000
     Epoch 15/16
     48/48 [==============================] - 0s 3ms/step - loss: 0.0019 - accuracy: 1.0000
     Epoch 16/16
     48/48 [==============================] - 0s 4ms/step - loss: 0.0018 - accuracy: 1.0000
```

Evaluating the NN Model

```
[22] bigmodel.evaluate(x_test, y_test)

     3/3 [==============================] - 0s 5ms/step - loss: 0.3039 - accuracy: 0.9583
     [0.30394694209098816, 0.9583333134651184]
```

```
[23] y_pred = bigmodel.predict(x_test)
     for i in range(30):
         m = max(y_pred[i])
         j = np.where(y_pred[i] == m)[0][0]
         k = np.where(y_test[i] == 1)[0][0]
         if(j!=k):
             print("------- Mismatch -------")
         print("Actual: " + gestures[k] + "\nPredicted: " + gestures[j] + "\nConfidence: " + str(m) + "\n")
```

```
Actual: up_clockwise
Predicted: up_clockwise
Confidence: 0.9054782

Actual: up_anticlockwise
Predicted: up_anticlockwise
Confidence: 0.99999666

Actual: forward_fall
Predicted: forward_fall
Confidence: 0.9999274

Actual: forward_fall
Predicted: forward_fall
Confidence: 0.9999685

Actual: left_to_right
Predicted: left_to_right
Confidence: 0.99223

Actual: down_to_up
Predicted: down_to_up
Confidence: 0.9908875

------- Mismatch -------
Actual: up_clockwise
Predicted: down_to_up
Confidence: 0.99820125

Actual: up_clockwise
Predicted: up_clockwise
Confidence: 0.9999536

Actual: left_fall
Predicted: left_fall
Confidence: 0.9999958

Actual: forward_clockwise
Predicted: forward_clockwise
Confidence: 0.9998758

Actual: down_to_up
Predicted: down_to_up
Confidence: 0.99974364

Actual: left_fall
Predicted: left_fall
```

Doing the same thing for a small architecture

```
smallmodel = tf.keras.models.Sequential()
smallmodel.add(tf.keras.layers.Dense(16, activation="relu", input_shape = (600,)))
smallmodel.add(tf.keras.layers.Dense(8, activation="relu"))
smallmodel.add(tf.keras.layers.Dense(4, activation="relu"))
smallmodel.add(tf.keras.layers.Dense(len(gestures), activation="softmax"))

smallmodel.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])

smallmodel.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 16)                9616
_________________________________________________________________
dense_5 (Dense)              (None, 8)                 136
_________________________________________________________________
dense_6 (Dense)              (None, 4)                 36
_________________________________________________________________
dense_7 (Dense)              (None, 8)                 40
=================================================================
Total params: 9,828
Trainable params: 9,828
Non-trainable params: 0
_________________________________________________________________
```

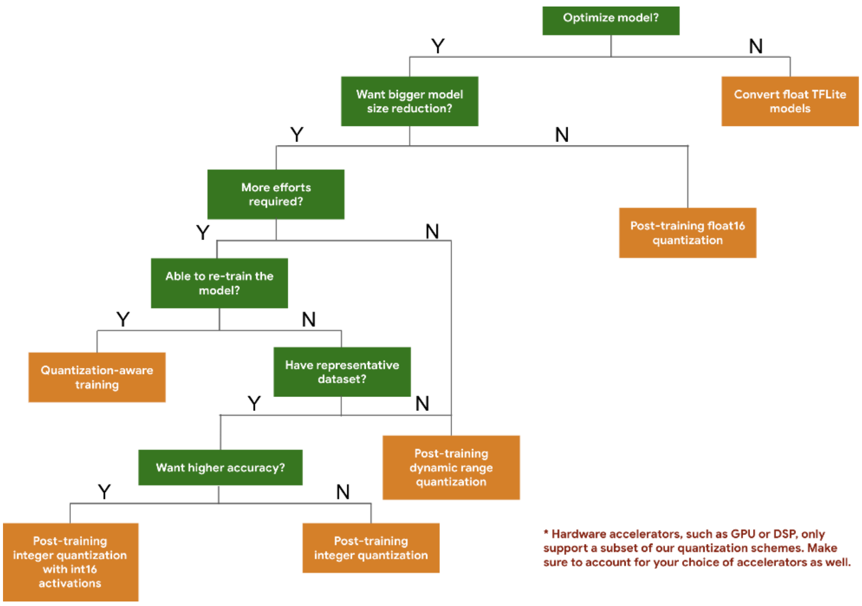
```
smallmodel.fit(x_train, y_train, epochs = 32, batch_size = 16)
smallmodel.evaluate(x_test, y_test)
```

```
24/24 [==============================] - 0s 1ms/step - loss: 0.7740 - accuracy: 0.6783
Epoch 27/32
24/24 [==============================] - 0s 1ms/step - loss: 0.7157 - accuracy: 0.6964
Epoch 28/32
24/24 [==============================] - 0s 1ms/step - loss: 0.6992 - accuracy: 0.7079
Epoch 29/32
24/24 [==============================] - 0s 1ms/step - loss: 0.6661 - accuracy: 0.7010
Epoch 30/32
24/24 [==============================] - 0s 1ms/step - loss: 0.6579 - accuracy: 0.6723
Epoch 31/32
24/24 [==============================] - 0s 1ms/step - loss: 0.6231 - accuracy: 0.7073
Epoch 32/32
24/24 [==============================] - 0s 1ms/step - loss: 0.6071 - accuracy: 0.7269
```

Quantization:

Quantization is a compression technique to reduce model size. This will help us to run the models faster. Currently, quantization can be used to reduce latency by simplifying the calculations that occur during inference, potentially at the expense of some accuracy.

This can be divided further into two broad classes: Post Training Quantization and Training Aware quantization. The default type of quantization is the Dynamic Range Quantization. Other Quantization Techniques adhere to the table given below.



Model Quantization Decision Tree

The default Strategies from the Decision tree are summarized below:

Technique	Benefits	Hardware
Weight quantization	4x smaller, 2-3x speedup, accuracy	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, etc.
Float16 quantization	2x smaller, potential GPU acceleration	CPU/GPU

The TF lite conversion using this quantization generates a TFLite Buffer that makes memory efficient operations. Another added advantage is that the converted object is a cross platform tool with no dependencies.

Inference:

The process of executing TF lite model to make predictions based on input data. To perform this, we use an interpreter in code.

Code for TF lite Quantization:

Converting the NN Model to TensorFlow Lite format

a) Without Quantization

```
converter = tf.lite.TFLiteConverter.from_keras_model(bigmodel)
tflite_model = converter.convert()
open("gesture_model.tflite", "wb").write(tflite_model)

import os
basic_model_size = os.path.getsize("gesture_model.tflite")
print("Model is %d bytes" % basic_model_size)
```

```
INFO:tensorflow:Assets written to: /tmp/tmprnjbildu/assets
INFO:tensorflow:Assets written to: /tmp/tmprnjbildu/assets
Model is 1530780 bytes
```

Saving Model to be loaded as a Arduino header file

```
bigmodel.save('gestureTinyModel.h5')
```

```
model = tf.keras.models.load_model('gestureTinyModel.h5')
```

```
model.evaluate(x_test, y_test)
```

```
3/3 [==============================] - 0s 2ms/step - loss: 0.2603 - accuracy: 0.9479
[0.26029565930366516, 0.9479166865348816]
```

<u>b)</u> With Integer 8 Bit Quantization

To do this, we first need to create a representative data gen using tensor_slices tool

```
def representative_data_gen():
    for input_value in tf.data.Dataset.from_tensor_slices(x_train):
        yield [input_value]
```

```
converter = tf.lite.TFLiteConverter.from_keras_model(bigmodel)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8

tflite_model_quant = converter.convert()
open("gesture_model_quant.tflite", "wb").write(tflite_model_quant)

basic_model_quant_size = os.path.getsize("gesture_model_quant.tflite")
print("Model is %d bytes" % basic_model_quant_size)
```

```
INFO:tensorflow:Assets written to: /tmp/tmp0uytjqlh/assets
INFO:tensorflow:Assets written to: /tmp/tmp0uytjqlh/assets
WARNING:absl:For model inputs containing unsupported operations which cannot be quantized, the `inference_input_type` attribute will default to the original
Model is 387280 bytes
```

```
interpreter = tf.lite.Interpreter(model_content = tflite_model_quant)
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'>
```

Conclusion:

As discussed earlier, we managed to get a model with a size 4 times smaller than the original model with an accuracy just slightly lesser than the big models. Since Edge devices often have limited memory or computational power, various optimizations can be applied to models so that they can be run within these constraints.

TensorFlow Lite and the TensorFlow Model Optimization Toolkit provide tools to minimize the complexity of optimizing inference. Thus, it's recommended that we always consider model optimization during our application development process.