CHS

TinyMl

Akash Singh – 2019UCO1528

Introduction

Tiny Machine Learning (TinyML) is a field of machine learning that focuses on the development and deployment of ML models on low power, low-footprint microcontroller devices like an Arduino. TinyML can be defined as a subfield of ML which pursues enabling ML applications on devices that are cheap, as well as a resource and power-constrained Tiny machine learning is broadly defined as a fast-growing field of machine learning technologies and applications including hardware, algorithms and software capable of performing on-device sensor data analytics at extremely low power, typically in the mW range and below, and hence enabling a variety of always-on use-cases and targeting battery-operated devices.

Example: We can make image recognition on Both ML as well as TinyML.

Framework

TensorFlow model is essentially a set of instructions that tell an *interpreter* how to transform data to produce an output. When we want to use our model, we just load it into memory and execute it using the TensorFlow interpreter. However, TensorFlow's interpreter is designed to run models on powerful desktop computers and servers. Since we will be running our models on tiny microcontrollers, we need a different interpreter that's designed for our use case. Fortunately, TensorFlow provides an interpreter and accompanying tools to run models on small, low-powered devices. This set of tools is called TensorFlow Lite. Before TensorFlow Lite can run a model, it first must be converted into the TensorFlow Lite format and then saved to disk as a file. We do this using a tool named the *TensorFlow Lite Converter*. The converter can also apply special optimizations aimed at reducing the size of the model and helping it run faster, often without sacrificing performance.

TensorFlow is a set of tools for building, training, evaluating, and deploying machine

learning models. Originally developed at Google, TensorFlow is now an open-source project built and maintained by thousands of contributors across the world. It is the most popular and widely used framework for machine learning. Most developers interact with TensorFlow via its Python library. TensorFlow does many different things. Keras is most often used, it is TensorFlow's high-level API that makes it easy to build and train deep learning networks. We'll also use TensorFlow Lite, a set of tools for deploying TensorFlow models to mobile and embedded devices, to run our model on-device.

Steps involved

1. Obtain a simple dataset.

2. Train a deep learning model.

3. Evaluate the model’s performance.

4. Convert the model to run on-device.

5. Write code to perform on-device inference.

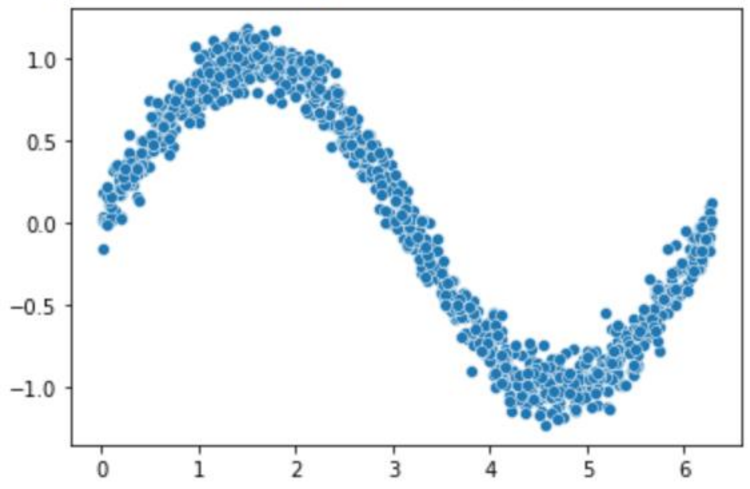
6. Build the code into a binary.

7. Deploy the binary to a microcontroller.

Step 1

DataSet we plotted

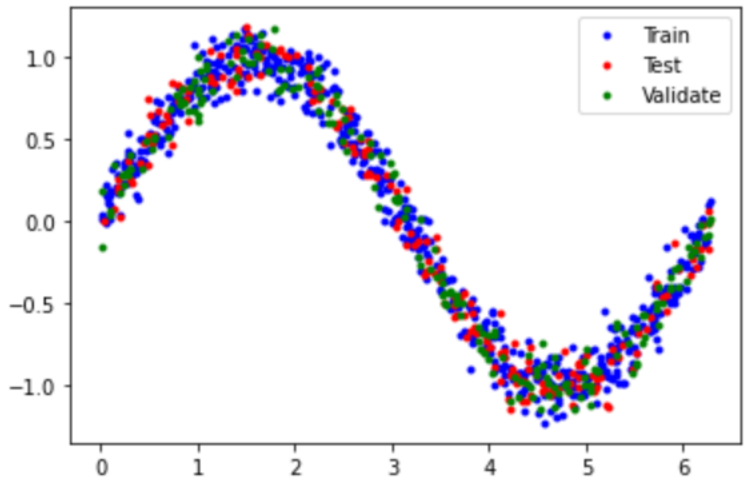
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc1a5a2d09
```



Step 2

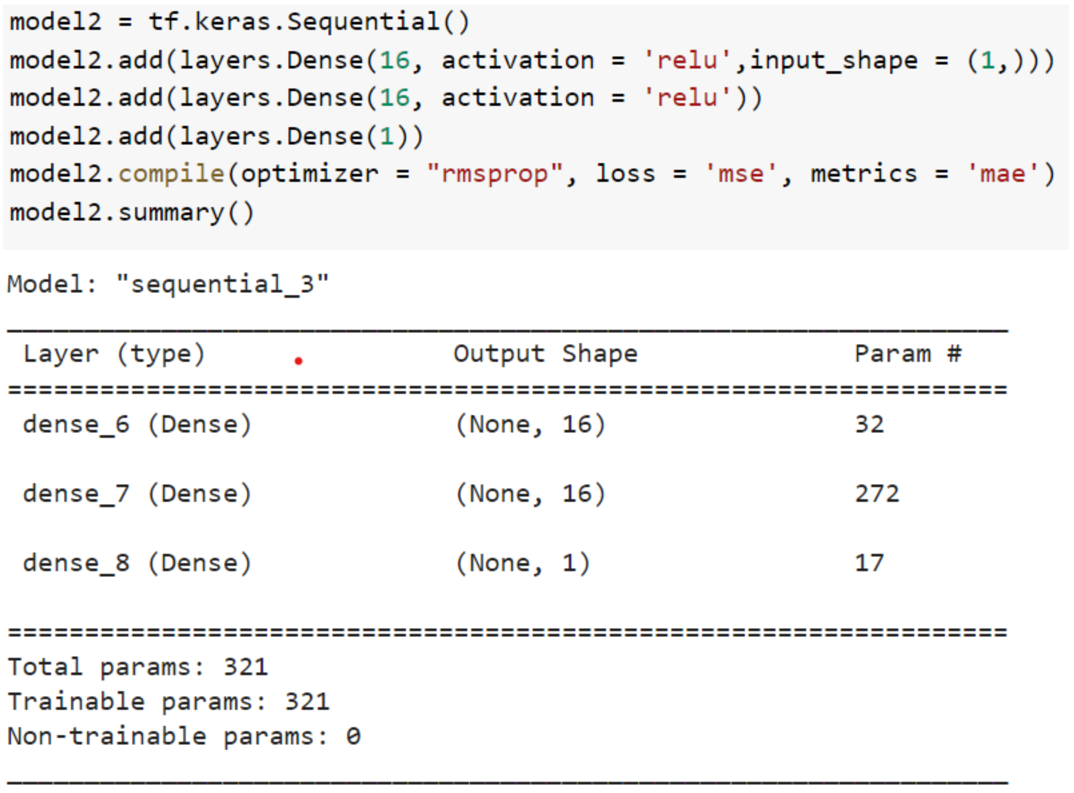
Training

```
plt.plot(x_train, y_train, 'b.', label="Train")
plt.plot(x_test, y_test, 'r.', label="Test")
plt.plot(x_validate, y_validate, 'g.', label = "Validate")
plt.legend()
plt.show()
```



Splliting the values and plotting them

Making a potential model



```
model2 = tf.keras.Sequential()
model2.add(layers.Dense(16, activation = 'relu',input_shape = (1,)))
model2.add(layers.Dense(16, activation = 'relu'))
model2.add(layers.Dense(1))
model2.compile(optimizer = "rmsprop", loss = 'mse', metrics = 'mae')
model2.summary()
```

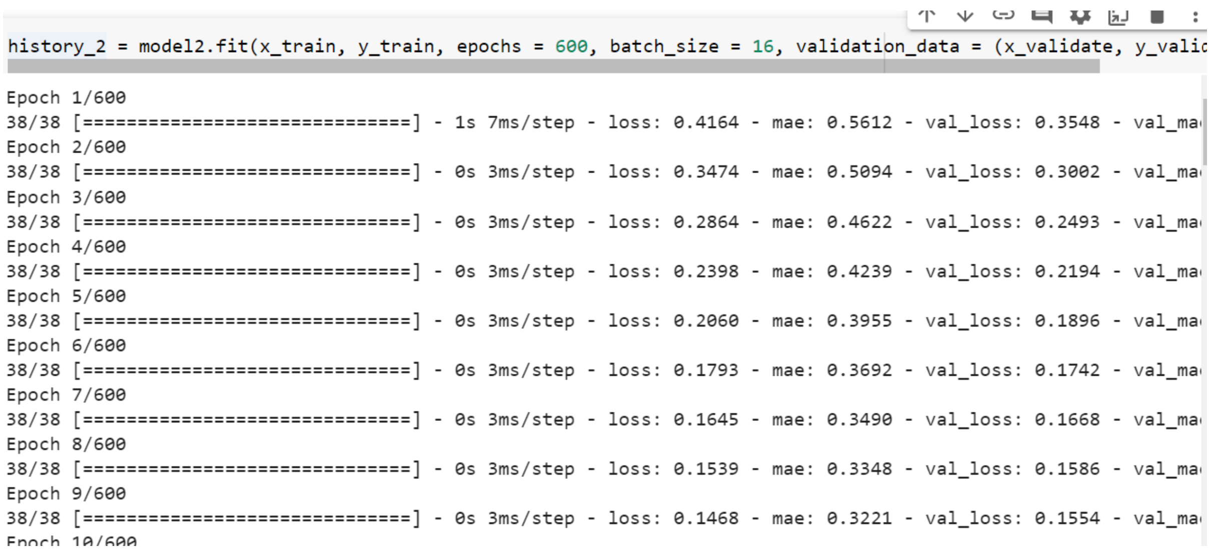
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 16)                32

 dense_7 (Dense)             (None, 16)                272

 dense_8 (Dense)             (None, 1)                 17

=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
_________________________________________________________________
```

Running the model with the spliced data set using backtracking to create a proper graph

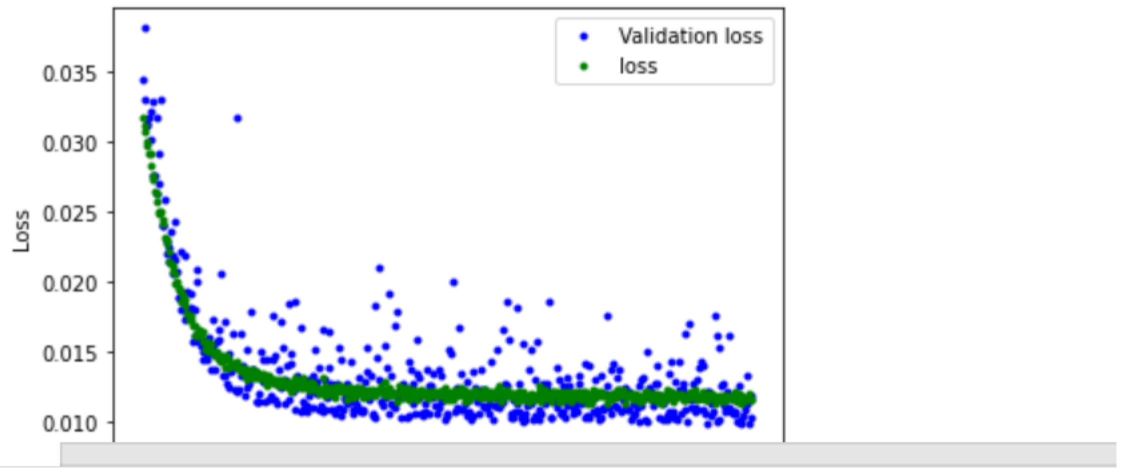


```
history_2 = model2.fit(x_train, y_train, epochs = 600, batch_size = 16, validation_data = (x_validate, y_vali
```

```
Epoch 1/600
38/38 [==============================] - 1s 7ms/step - loss: 0.4164 - mae: 0.5612 - val_loss: 0.3548 - val_ma
Epoch 2/600
38/38 [==============================] - 0s 3ms/step - loss: 0.3474 - mae: 0.5094 - val_loss: 0.3002 - val_ma
Epoch 3/600
38/38 [==============================] - 0s 3ms/step - loss: 0.2864 - mae: 0.4622 - val_loss: 0.2493 - val_ma
Epoch 4/600
38/38 [==============================] - 0s 3ms/step - loss: 0.2398 - mae: 0.4239 - val_loss: 0.2194 - val_ma
Epoch 5/600
38/38 [==============================] - 0s 3ms/step - loss: 0.2060 - mae: 0.3955 - val_loss: 0.1896 - val_ma
Epoch 6/600
38/38 [==============================] - 0s 3ms/step - loss: 0.1793 - mae: 0.3692 - val_loss: 0.1742 - val_ma
Epoch 7/600
38/38 [==============================] - 0s 3ms/step - loss: 0.1645 - mae: 0.3490 - val_loss: 0.1668 - val_ma
Epoch 8/600
38/38 [==============================] - 0s 3ms/step - loss: 0.1539 - mae: 0.3348 - val_loss: 0.1586 - val_ma
Epoch 9/600
38/38 [==============================] - 0s 3ms/step - loss: 0.1468 - mae: 0.3221 - val_loss: 0.1554 - val_ma
Epoch 10/600
```

Step 3

```
loss2 = history_2.history['loss']
val_loss2 = history_2.history['val_loss']
epochs = np.arange(start = 1, stop = len(loss2) + 1)
skip = 100
plt.plot(epochs[100:], val_loss2[100:], "b.", label = "Validation loss")
plt.plot(epochs[100:], loss2[100:], "g.", label = "loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Evaluating the performance. Seeing how the value of the loss and validation loss function is behaving with each.

Step 4

```
# Convert the model to the TensorFlow Lite format without quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model2)
tflite_model = converter.convert()

# Save the model to disk
open("sine_model.tflite", "wb").write(tflite_model)

# Convert the model to the TensorFlow Lite format with quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model2)

# Indicate that we want to perform the default optimizations,
# which include quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Define a generator function that provides our test data's x values as a representative dataset, and tell the
def representative_dataset_generator():
  for value in x_test:
# Each scalar value must be inside of a 2D array that is wrapped in a list
    yield [np.array(value, dtype=np.float32, ndmin=2)]

converter.representative_dataset = representative_dataset_generator
```

Converting the file to run on edge devices

```
import os
basic_model_size = os.path.getsize("sine_model.tflite")
print("Basic model is %d bytes" % basic_model_size)
quantized_model_size = os.path.getsize("sine_model_quantized.tflite")
print("Quantized model is %d bytes" % quantized_model_size)
difference = basic_model_size - quantized_model_size
print("Difference is %d bytes" % difference)
```

```
Basic model is 2932 bytes
Quantized model is 2752 bytes
Difference is 180 bytes
```

To convert the corresponding model we:

Instantiate an Interpreter object.

2. Call some methods that allocate memory for the model.

3. Write the input to the input tensor.

4. Invoke the model.

5. Read the output from the output tensor.

We use the TensorFlow Lite Converter's Python API to do this. It takes our Keras model and

writes it to disk in the form of a *FlatBuffer*, which is a special file format designed to be space-efficient because we're deploying to devices with limited memory. In addition to creating a FlatBuffer, the TensorFlow Lite Converter can also apply optimizations to the model. These optimizations generally reduce the size of the model, the time it takes to run, or both. This can come at the cost of a reduction in accuracy, but the reduction is often small enough.

One of the most useful optimizations is *quantization*. By default, the weights and biases in a model are stored as 32-bit floating-point numbers so that high-precision calculations can occur during training. Quantization allows you to reduce the precision of these numbers so that they fit into 8-bit integers—a four times reduction in size. it often results in minimal loss in accuracy.

After Appling quantization techniques we see the size of the model created. We can now using the corresponding files created in "tflite" format in the microcontroller to do as we please. Here I have created 2 files, one is with quantization technique and the other one is without quantization technique. Using these files we can compile it to different formats for different microcontrollers. However the microcontrollers themselves must be coded to use the files created for themselves.

TinyML Application

TinyML applications excel in embedded applications with size and power constraints. With microcontrollers as small as the tip of your thumb and consuming limited power despite the prolonged operation, TinyML can be easily applied in numerous scenarios to create **always-on smart applications that make predictions, process complex data, and administer solutions**!

1-Predictive Maintenance

By retrofitting industrial machinery with TinyML, we can monitor the performance characteristics of a machine and deliver maintenance ahead of failure. This minimizes equipment downtime and repair costs, which improves both business revenue and workplace safety.

2-On-Demand Healthcare

TinyML finally has an important role to play in providing persistent, on-demand healthcare. For example, motion-based fall detectors can recognize instances of falls in children or the elderly, while keyword detection can pick up cries for help. Furthermore, TinyML is now enabling affordable solutions in on-demand medical diagnostics, which can be used in rural areas to supplement limited healthcare facilities.

3-Smart Cities

TinyML will also play a critical role in developing smart cities with small, low-power devices that can be deployed virtually everywhere. These smart devices might be used to monitor and optimize traffic, analyze carbon footprints, or even develop vision-based security systems. For example, the addition of TinyML to traffic junctions could enable road traffic optimization in real-time. By further leveraging the interconnectivity of IoT, such benefits could be multiplied manifold through a network effect in these cities!

4-Smart Agriculture

TinyML serves to improve the performance of smart agriculture techniques by providing real-time analysis of environmental factors to improve crop growth in precision agriculture. With the rise of autonomous greenhouses, TinyML is also helping agricultural experts in making decisions that maximize crop yield by learning from past data. On the other hand, we can even use TinyML devices to analyze the conditions and behaviour of individual livestock to make intelligent conclusions about their state of health in real-time.