TensorFlow Lite Inference on Arduino Micro-Controllers

Maxime Alos, Venetis Pallikaras, Akshay Sridharan
General Electric Research
Georgia Institute of Technology M.S. in Analytics Practicum Project

Introduction and problem statement:
The use of machine learning in industrial settings is an immensely useful and well-researched topic with astounding capabilities for operational improvements. In recent years, the field of TinyML, which utilizes lightweight, small-imprint edge devices in resource constrained contexts has received much attention for its potential in bringing the power and accuracy of machine learning analytics to a small form factor. Working alongside General Electric Research, our team set out to document and identify the feasibility of such methods for use in the corporation's various industrial applications. In order to do this, we designed and evaluated proof-of-concept models to identify the resource consumption and effectiveness of Arduino-based TinyML pipelines for a set of relevant inference problems. We provided particular emphasis on the exploration of various conversion and compression methods to further expound on the capacities of the presently available technologies.

In industrial settings, it is imperative to maintain close watch of machinery and their functionality to minimize downtime, maintain safe and efficient operation, and, ultimately, maximize profits. Presently, machine learning pipelines for prognostic health management (PHM) of industrial machinery utilize a cloud-based pipeline with offline data. This introduces significant issues with respect to both the latency of obtaining, processing, and producing inference on data and the network connectivity necessary to do so in the first place. Whether it be detecting anomalies in wind turbines or monitoring power generator loads remotely, small amounts of latency in detection and course correction can lead to a number of consequences.

Hardware and Software Description:
For the extent of our project we developed using the Adafruit EdgeBadge and Arduino Nano 33 BLE. The Adafruit EdgeBadge is equipped with 512KB of Flash, 192KB of RAM, and 2MB of QSPI flash for file storage. The Arduino Nano is equipped with 1MB of Flash Memory and 256KB of SRAM. The programs run on these microcontrollers to conduct inference were written in C++ through the Arduino IDE. Deep learning models were developed using Python with the TensorFlow library. Within the library, the TensorFlow Lite mobile library was used to create models usable for inference on microcontrollers. A TensorFlow Lite uses a compression method that converts a TensorFlow model from the default Protocol Buffer format to the FlatBuffer format, which both composes a smaller storage footprint by representing hierarchical data in a flat binary buffer while simultaneously decreasing inference time. The FlatBuffer format decreases inference time as data is directly accessed without an extra parsing/unpacking step that is found in the Protocol Buffer format. Structurally, the TensorFlow Lite model is stored in the combination of a non-data buffer, total-data buffer, and zero-value buffer. The non-data buffer is tasked with holding data attributed to the operators and subgraph structure of the model, while the total data buffer stores the non-zero parameters necessary for model inference. The zero-value buffer holds all zeros in the model. From this FlatBuffer format, it was necessary to convert, and thereby partially decompress, the file into a C++ Array to be compiled in the Arduino script with inference executed on-device. This array was the only file structure presently compatible for inference on Arduino devices.

Metrics to be explored:
In order to identify the capacity of the TinyML pipeline to decrease resource consumption, it was necessary to identify the primary metrics of interest. As the most notable constraint imposed on modeling from the use of a microcontroller is the device's memory, the compression factor between the initial Keras file and the final usable C++ Array was the primary metric considered. Next, in order to develop a holistic comparative understanding of the various models considered as well as the conversion and compression methods under review, the accuracy of each model configuration was under consideration.

Configurations considered and overview of model compression/conversion methods:
In order to properly catalog the capabilities of these pipelines, varied model compression and conversion methods were tested and documented. The model compression methods explored included post-training floating point full quantization, post-training integer full quantization, quantization-aware training with post-training full integer quantization, weight clustering, and pruning. The methods tested for converting between the TensorFlow model format and the TensorFlow Lite FlatBuffer format included the "from keras model" method and the "from saved model" method using a concrete function and input signature. The "from keras model" method (fkm method) utilizes a high-level API while the "from saved model" method makes use of a low-level API, thereby allowing for some improved performance by omitting unnecessary redundancies. The concrete function (which is a wrapper around a tf.Graph) and input signature mentioned in the from saved model method simply creates a callable version of the TensorFlow graph and fixes the input tensor size and datatype for the converted model.

Full quantization methods aim to enable size reductions through the truncation of the default 32-bit parameters to 16-bit floating point values or 8-bit integers. In doing so, the storage required for weights and activations decrease by a factor of 2 or 4, respectively. However, there are drawbacks with these methods. Namely, this modification to the model's parameters leads to a potential decrease in accuracy. Additionally, this change only leads to compression in the total data buffer size of the TFLite model, and in fact leads to an increase in the size of the TFLite model's non-data buffer size as it requires the quantization of compressed data used to describe the operators and subgraphs of the model. In models with a relatively low number of trainable parameters (small weight and activation tensors) this leads to a net increase in the file size. There exists options to utilize hybrid quantization, in which only some subset of parameters are quantized, however the Arduino implementation of TensorFlow Lite does not presently support the execution of hybridized graphs.

To overcome the issues posed by underfitting from full post-training quantization, a variation known as quantization-aware training may be utilized. The method functions by retraining a model of the same structure in which the parameters are made robust to quantization by emulating inference-time quantization. This model is thereby "quantization-aware" and is subsequently quantized using full post-training quantization. This process allows for a theoretical improvement in the inference performance of a model while still providing the compression benefits of quantization.

Weight clustering is a method for reducing a neural network's size that involves performing a clustering algorithm on its weights in order to introduce redundancy in the weights data. This is achieved by performing K-means clustering on the weights of each layer of the architecture. For each of those layers, the weights are divided into K clusters and the centroid value of each cluster is found. Each of the weights are then replaced by their centroid value. This introduces a large amount of redundancy in the data that can be taken advantage of to compress the model size.

Weight Pruning for machine learning is setting some of the model's weights to zero in order to omit them and reduce the size of the model, thus compressing it.. In some situations, it may be desirable to prune nodes along with weights. The difficulty most usually encountered while performing weight pruning is understanding which weights to prune and which not to. The final goal in removing weights is to remove as many as possible of the less important parameters while conserving the more important ones. That definition being ambiguous, a number of methods to perform weight pruning exist. One of the simplest solutions would be to cut off all weights that are below a predetermined threshold value. If working on nodes instead of simply working on weights, the intuition to use is that neurons that rarely output high values should be pruned. This could mean either pruning neurons outputting low values or those with low activation values.

Diabetes Classification:

The first problem that we explored was performing classification for the detection of diabetes in patients based on medical data that is typically indicative of the presence of pregnancy diabetes in women.

The data used was extracted from online databases and contains information concerning the number of pregnancies, glucose levels, blood pressure levels, skin thickness, body mass index, a diabetes indicator called diabetes pedigree function, and age. For each patient it is also indicated whether they actually have diabetes.

To solve this problem an ANN architecture was implemented. The architecture consisted of 3 dense layers with ReLu activation of sizes 128, 64 and 32, as well as one last dense layer with sigmoid activation. We used binary cross entropy as the loss function, 50 epochs, a batch size of 10, and a learning rate of 0.001. The optimizer was Adam. To perform tests and measure accuracy the data was split into 60% training data and 40% test data. This model achieved an accuracy of 73.38%.

Once that model was established as a baseline, we went on to test model compression methods to try and optimize the size of the model before deployment on an Arduino board. We tried implementing the following methods: full integer quantization, quantization aware training, weight pruning and weight clustering, and combinations of those. Once those compression methods are applied we obtain a tflite model that can be converted into a header file that is ready for deployment on the Arduino. This means the header file is the file whose size we ultimately want to reduce, and which will be compared to the original keras model file size.

Integer quantization is the simple conversion of model weights into integers. It achieves a compression ratio of about 2.88 with a size of 91,200 bytes while entirely conserving the accuracy of the model as it achieves 73.37% accuracy.

Quantization aware training is a second round of training applied to the model after the classical round of training, and is used to fine tune the weights to be optimally converted into integers with integer quantization. Interestingly, while the tflite model obtained that way is slightly smaller than the one obtained purely with quantization without quantization aware training, that advantage is lost in the conversion into a header file. However, the final compression remains very similar to the one obtained with quantization only as the final array size is 97,170 bytes. The accuracy is slightly lower than previously at 69.16%.

Weight clustering and weight pruning work under the same principle of adding redundancy into the trained weights. Weight clustering achieves that by clustering weights together through a k-means algorithm, while weight pruning achieves it by pruning weights that are close enough to zero. Once the weights have been processed, a redundancy appears in the data that can be taken advantage of by zipping the resulting model file to further decrease its size. However, zipped files cannot be loaded into Arduino boards and used properly by the currently developed

libraries. This means that the size reduction of those two methods are not useful to the current project.

Anomaly Detection Exploration:
The next problem that was explored was the anomaly detection problem for multivariate time series data. The problem itself is tasked with identifying if an observation or sequence of observations occur during normal operation or are indicative of an anomaly in the underlying generative process (which would indicate potential machine failures).

To demonstrate the technologies present capabilities for this problem's methodology, a synthetic dataset was designed and created using the Adafruit EdgeBadge. Using the magic_wand Adafruit EdgeBadge Accelerometer example available through the Adafruit Arduino library, three-channel accelerometer data was written to the serial port at 25Hz for 90 seconds. The training data for the model was generated by shaking the EdgeBadge at a fixed frequency to simulate "normal turbulence" for one minute. The test data was generated by shaking device more erratically to simulate "anomalous turbulence" for the next 30 seconds. This method also introduced some unavoidable human error, allowing for more robust modeling and inference conclusions. The specific generation process was laying the device on a table and rhythmically tapping. The data was then loaded to Python from the serial port using the pySerial library. The 90 seconds of data consisted of a total of 2,250 observations. A 40%-60% train-test split was utilized to emulate low training data availability, and allow a mix of normal and anomalous observations in the test set.

Once the data had been generated and saved, the modeling phase was conducted. The methodology used throughout this phase was the autoencoder for anomaly detection. An autoencoder model functions by learning the identity function of the data (i.e. $f(x) = x$) while imposing structural constraints (smaller layers near the middle of the sequential model) to force mid-layer compression. This forces the model to be unable to perfectly recreate the training data, but instead emulates lossy compression thereby preventing overfitting.

When the model is subsequently evaluated on test data, large mean absolute residuals between the predicted and true 3-channel vectors are used to indicate the existence of an anomalous point. Specifically, an anomaly threshold is set from the distribution of the training loss to identify points in the test set with abnormally large losses as anomalies. For the extent of this exploration, the anomaly threshold was set to one standard deviation above the mean of the mean absolute error (MAE) training loss for each prediction. This method can be conducted pointwise (training to learn the identity function for singular 3-channel points) or sequentially (training to learn the identity function for a sequence of 3-channel points).

The method also can be coupled with a voting scheme with a training-defined voting threshold to collectively classify sequential batches as either all anomalous or all normal. The voting threshold utilized for the extent of this exploration was one standard deviation above the mean number of votes deeming points as anomalous in each disjoint, 15-sample batch within the training data. This voting scheme leads to improvements in prediction as the time distributions of accelerometer readings between normal and anomalous data are symmetric with similar mode, despite having very dissimilar tail behavior (the votes that would indicate an anomaly) as the accelerometer readings themselves are cyclic in nature.

The model architecture used to evaluate the variations of the above labeled options utilized four fully connected layers using ReLU (Rectified Linear Unit) activations. These sequential layers were of size 32, 8, 32, and the size of the required output. The input and output of the models varied based on whether the model was a pointwise or Seq2Seq (sequence-to-sequence) autoencoder. All models were trained using 100 epochs with batches of size 10. The validation split was 5%, the loss used was mean absolute error, and the optimizer used was Adam.

The model compression methods tested on this model architecture included quantization-aware training with full integer quantization, weight clustering, and pruning. Weight clustering was used with k-means++ centroid initialization and 32 clusters. Pruning was conducted with a polynomial decay pruning schedule with an initial sparsity of 50% and final sparsity of 75%. Both weight clustering and pruning were fine-tuned using 20 epochs, a batch size of 128, a learning rate of 10^{-5}, and a validation split of 5%.

The size and accuracy metrics for this exploration are summarized in the appendix, and are labeled appropriately with the model type, conversion method, and compression technique. One primary finding from the analysis was the superiority of the “from saved model” conversion method, which utilizes the low-level API detailed previously. This method led to improved compression across all models tested while showing little to no impact on test performance and inference capabilities. The analysis also showed the drawbacks of quantization compression methods on the TensorFlow Lite FlatBuffer format, particularly with respect to relatively small (low parameter count) models. As quantization requires the conversion of all portions of a model, including the data buffer used to store the operations and structure of the model, the method can lead to a net increase in the file size of the stored TensorFlow Lite model despite compression of the parameter data. Thus, the application of quantization is only useful for models with a significant enough parameter count.

Fire Detection Classification and Fruit Detection Classification:

The next problems that were explored were the detection of fires in images(binary) and the detection of various fruits in images (multi-class) using the CNN architecture. Unfortunately transfer learning wasn't able to be utilized due to the large size of the pretrained models. Even "Mobilenet " which is considered the smallest pretrained model and is optimized for mobile deployment with alpha parameter set to minimum(proportion of filters used) was unable to fit in the memory of the arduino. Thus custom architectures were used, utilizing CNN, BatchNormalization, Dropout and Fully Connected(final softmax layer) Layers.

A few important notes that need to be mentioned. The sketch of the arduino code, the model and the arduino code constants are stored in Flash Memory. Thus our upper bound for model size must not exceed 1MB, in reality it should be even less given there are other parameters that need to fit in the memory. The dynamic arduino code variables, the tensors ("tensor arena") and sketch auto generated variables are stored in SRAM. This means our intermediate activations must not exceed 256KB of SRAM Memory. The model itself when deployed doesn't require dynamic memory contrary to input, output and intermediate activations, which reside in the "tensor arena".

After the models were trained using a custom architecture common for both, the models were converted from TensorFlow to TensorFlow Lite. Following this, four model size reduction approaches were used:

1. Full Integer Quantization: It converts all intermediate activations and weights into int8 from float32.
2. Quantization Aware Training + Full Integer Quantization: Quantization Aware Training is applied after our model is trained and works like fine tuning. It essentially trains our model for roughly 2 or 3 epochs(and small learning rate) and converts the weights in a format and range that would be ideal for integer quantization. Following this Full Integer Quantization is applied.
3. Weight Clustering + Full Integer Quantization: Weight Clustering is applied after our model is trained and works like fine tuning. It essentially prepares/trains our model for roughly 2 or 3 epochs(and small learning rate). Weights are clustered using the k-means++ algorithm. It first groups the weights in N clusters(user defined) and then replaces these weights with their cluster centroid value. Following this Full Integer Quantization is applied.
4. Weight Pruning + Full Integer Quantization:Weight Pruning is applied after our model is trained and works like fine tuning. It essentially prepares/trains our model for roughly 2 or 3 epochs(and small learning rate). Magnitude-Based weight pruning gradually zeroes out model weights during the fine-tuning process to achieve model sparsity. The percentage of sparsity is user defined. During inference these zeroes can be skipped resulting in latency improvements too.Following this Full Integer Quantization is applied.

Following these all models were converted to header files(.h) and were loaded into the arduino. Since a camera wasn't available, test images were directly loaded into the arduino in order to test the validity of the models. In both cases inference time was roughly 1.8 seconds.

It should be noted that only Full Integer Quantization could be used since Arduino doesn't support Hybrid Models. This means that all the internal components of our model such as activations and weights must share the same data type which could either be float-32 or int8. Essentially allowing us to only use int8. The input and output don't have to be int8, but they can only be converted to int8 not uint8 due to deprecated support. Although int8 input quantization didn't result in significant accuracy drop, the int8 output quantization rendered the model unusable thus it was skipped. Lastly, pruning and clustering lead to a significant size reduction when the models are compressed which isn't possible for arduino to be utilized since it doesn't have a file system nor a way to decompress it.

Primary Pitfalls:
Through our exploration we identified several gaps in the current technology that would prevent the production grade implementation of certain modeling approaches. These most notably arose with respect to the varieties of models that can be converted to the TensorFlow Lite FlatBuffer format and the model compression methods that can be feasibly utilized. Tensorflow models are formed through the combination of operations that are connected within subgraphs. The combination of some set of subgraphs composes a model in its entirety. One present issue with the current underlying implementation of TensorFlow Lite for Arduino is the lack of support for multi-subgraph models. This prevents the use of certain model structures, such as the combination of two or more LSTM (Long short-term memory) layers that require multiple subgraphs. Furthermore, there are also operations in TensorFlow that are not presently supported in TensorFlow Lite, and so some models are simply not convertible at this time.

Another significant issue was the fact the libraries and documentation were severely outdated. Also, memory restrictions prohibited us from using complex architectures and pretrained models. Furthermore, arduino restrictions in how the model should be structured/converted stopped us from exploring various quantization approaches, resulting in us being able to only use full integer quantization with and without input quantization. Furthermore, weight clustering and pruning were not originally designed to improve quantization performance but improve the compression. In practice though the sparsity and common values lead to not only a good compression but small reduction in accuracy. On the other hand, these models cannot be compressed since neither does arduino have a file system(model can be only saved as header file - .h) nor does it have a way to decompress it. Another significant problem was the fact that some models and operations aren't able to be used in arduino "local tensorflow" which again as a result limits the "tools" that can be used. Lastly, arduino as mentioned before lacks a file system. This limits not only the overall usability of arduino but also the format of our model (header file) which restricts it from being properly compressed in a way tensorflow would compress the models for computer usage.

These issues present significant drawbacks as it limits modeling capabilities, especially with particularly complex models.

Conclusions:
We would propose the usage of a microcontroller which not only will have low cost but increased performance in both processing and memory. Microcontrollers can also use TensorFlow Lite models directly without having to convert them to header files and having to interact with the Arduino TensorFlow library which is very restricting.

Appendix: SWAP-C Exploration

Below are the results for the various models used. These results contain information regarding the memory requirements, accuracy and the approach used.

Accelerometer Anomaly Detection:

Pointwise Anomaly Detection – File Size Metrics:

Compression Technique	Conversion Method	TFLite Model Size (Bytes)	TFLite Model Non-Data Buffer Size (Bytes)	TFLite Model Total Data Buffer Size (Bytes)	Final C Array File Size (Bytes)	Keras baseline gzipped .h5 File Size (Bytes)	Keras Clustered/ Pruned gzipped .h5 File Size (Bytes)	TFLite Clustered/ Pruned gzipped .tflite File Size (Bytes)
N/A	Concrete Function Input Signature from Saved Model	5,304	2,088	3,216	32,834	4,837	N/A	N/A
N/A	From Keras Model	12,128	8,884	3,244	74,940	4,837	N/A	N/A
Quantization-Aware Training with Integer Quantization (No Input Quantization)	Concrete Function Input Signature from Saved Model	5,632	4,500	1,132	34,876	4,837	N/A	N/A
Quantization-Aware Training with Integer Quantization (No Input Quantization)	From Keras Model	11,960	11,045	915	73,924	4,837	N/A	N/A

Weight Clustering	Concrete Function Input Signature from Saved Model	5,368	2,152	3,216	33,258	4,837	3,206	2,563
Weight Clustering	From Keras Model	12,360	9,116	3,244	76,402	4,837	3,206	4,499
Pruning	Concrete Function Input Signature from Saved Model	5,376	2,160	3,216	33,308	4,837	4,844	3,961
Pruning	From Keras Model	12,380	9,136	3,244	76,524	4,837	4,844	5,947

Pointwise Anomaly Detection – File Compression Metrics:

Compression Technique	Conversion Method	Compression ratio between .tflite and Original Keras	Compression ratio between C Array and Original Keras	Compression ratio between Keras Baseline gzipped .h5 and TFLite Clustered/Pruned gzipped .tflite	Compression ratio between Original Keras and TFLite Clustered/Pruned gzipped .tflite
N/A	Concrete Function Input Signature from Saved Model	2.71%	16.80%	N/A	N/A
N/A	From Keras Model	6.21%	38.35%	N/A	N/A
Quantization-Aware Training with Integer Quantization (No Input Quantization)	Concrete Function Input Signature from Saved Model	2.88%	17.85%	N/A	N/A

Quantization-Aware Training with Integer Quantization (No Input Quantization)	From Keras Model	6.12%	37.83%	N/A	N/A
Weight Clustering	Concrete Function Input Signature from Saved Model	2.75%	17.02%	52.99%	1.31%
Weight Clustering	From Keras Model	6.32%	39.10%	93.01%	2.30%
Pruning	Concrete Function Input Signature from Saved Model	2.75%	17.04%	81.89%	2.03%
Pruning	From Keras Model	6.34%	39.16%	122.95%	3.04%

Pointwise Anomaly Detection without Voting – Accuracy Metrics:

Compression Technique	Conversion Method	MAE Training Loss (Reconstruction Error)	Accuracy (Anomaly Detection Classification)	Precision (Anomaly Detection Classification)	Recall (Anomaly Detection Classification)	F1-Score (Anomaly Detection Classification)
N/A	Concrete Function Input Signature from Saved Model	0.0015703	0.5533333	0.7357724	0.2517385	0.3751295
N/A	From Keras Model	0.0015703	0.5533333	0.7357724	0.2517385	0.3751295
Quantization-Aware Training with Integer Quantization (No Input Quantization)	Concrete Function Input Signature from Saved Model	0.0101515	0.5111111	0.7657658	0.1182197	0.2048193
Quantization-Aware Training with Integer Quantization	From Keras Model	0.0107187	0.5111111	0.7706422	0.1168289	0.2028986

(No Input Quantization)						
Weight Clustering	Concrete Function Input Signature from Saved Model	0.0110993	0.5703704	0.7278689	0.3087622	0.4335938
Weight Clustering	From Keras Model	0.0110993	0.5703704	0.7278689	0.3087622	0.4335938
Pruning	Concrete Function Input Signature from Saved Model	0.0020026	0.5155556	0.6199262	0.2336579	0.3393939
Pruning	From Keras Model	0.0020026	0.5155556	0.6199262	0.2336579	0.3393939

Pointwise Anomaly Detection with Voting – Accuracy Metrics:

Compression Technique	Conversion Method	Accuracy (Anomaly Detection Classification)	Precision (Anomaly Detection Classification)	Recall (Anomaly Detection Classification)	F1-Score (Anomaly Detection Classification)
N/A	Concrete Function Input Signature from Saved Model	0.7437037	0.7762963	0.7287900	0.7517934
N/A	From Keras Model	0.7437037	0.7762963	0.7287900	0.7517934
Quantization-Aware Training with Integer Quantization (No Input Quantization)	Concrete Function Input Signature from Saved Model	0.6451852	0.6451852	0.6451852	0.6451852
Quantization-Aware Training with Integer Quantization	From Keras Model	0.6451852	0.6451852	0.6451852	0.6451852

(No Input Quantization)					
Weight Clustering	Concrete Function Input Signature from Saved Model	0.7103704	0.7733333	0.6453408	0.7035633
Weight Clustering	From Keras Model	0.7103704	0.7733333	0.6453408	0.7035633
Pruning	Concrete Function Input Signature from Saved Model	0.5118519	0.6000000	0.2503477	0.3532875
Pruning	From Keras Model	0.5118519	0.6000000	0.2503477	0.3532875

Sequence-to-Sequence Anomaly Detection – File Size Metrics:

Compression Technique	Conversion Method	TFLite Model Size (Bytes)	TFLite Model Non-Data Buffer Size (Bytes)	TFLite Model Total Data Buffer Size (Bytes)	Final C Array File Size (Bytes)	Keras baseline gzipped .h5 File Size (Bytes)	Keras Clustered/Pruned gzipped .h5 File Size (Bytes)	TFLite Clustered/Pruned gzipped .tflite File Size (Bytes)
N/A	Concrete Function Input Signature from Saved Model	5,304	2,088	3,216	32,834	4,841	N/A	N/A
N/A	From Keras Model	12,128	8,884	3,244	74,940	4,841	N/A	N/A
Quantization-Aware Training with Integer Quantization (No Input	Concrete Function Input Signature from Saved Model	5,632	4,500	1,132	34,876	4,841	N/A	N/A

Quantization)								
Quantization-Aware Training with Integer Quantization (No Input Quantization)	From Keras Model	11,960	11,045	915	73,924	4,841	N/A	N/A
Weight Clustering	Concrete Function Input Signature from Saved Model	5,352	2,136	3,216	33,160	4,841	3,149	2,512
Weight Clustering	From Keras Model	12,308	9,064	3,244	76,080	4,841	3,149	4,439
Pruning	Concrete Function Input Signature from Saved Model	5,352	2,136	3,216	33,160	4,841	4,797	3,915
Pruning	From Keras Model	12,308	9,064	3,244	76,080	4,841	4,797	5,887

Sequence-to-Sequence Anomaly Detection – File Compression Metrics:

Compression Technique	Conversion Method	Compression ratio between .tflite and Original Keras	Compression ratio between C Array and Original Keras	Compression ratio between Keras Baseline gzipped .h5 and TFLite Clustered/Pruned gzipped .tflite	Compression ratio between Original Keras and TFLite Clustered/Pruned gzipped .tflite
N/A	Concrete Function Input Signature from Saved Model	2.72%	16.82%	N/A	N/A
N/A	From Keras Model	6.21%	38.38%	N/A	N/A

Quantization-Aware Training with Integer Quantization (No Input Quantization)	Concrete Function Input Signature from Saved Model	2.88%	17.86%	N/A	N/A
Quantization-Aware Training with Integer Quantization (No Input Quantization)	From Keras Model	6.13%	37.86%	N/A	N/A
Weight Clustering	Concrete Function Input Signature from Saved Model	2.74%	16.98%	51.89%	1.29%
Weight Clustering	From Keras Model	6.30%	38.96%	91.70%	2.27%
Pruning	Concrete Function Input Signature from Saved Model	2.74%	16.98%	80.87%	2.01%
Pruning	From Keras Model	6.30%	38.96%	121.61%	3.01%

Sequence-to-Sequence Anomaly Detection without Voting – Accuracy Metrics:

Compression Technique	Conversion Method	MAE Training Loss (Reconstruction Error)	Accuracy (Anomaly Detection Classification)	Precision (Anomaly Detection Classification)	Recall (Anomaly Detection Classification)	F1-Score (Anomaly Detection Classification)
N/A	Concrete Function Input Signature from Saved Model	0.0029247	0.5600000	0.7802691	0.2420028	0.3694268
N/A	From Keras Model	0.0029247	0.5600000	0.7802691	0.2420028	0.3694268
Quantization-Aware Training with Integer Quantization (No Input	Concrete Function Input Signature from Saved Model	0.0125257	0.5059259	0.6181818	0.1891516	0.2896699

Quantization)						
Quantization-Aware Training with Integer Quantization (No Input Quantization)	From Keras Model	0.0127438	0.5125926	0.6331878	0.2016690	0.3059072
Weight Clustering	Concrete Function Input Signature from Saved Model	0.0196016	0.5200000	0.6059701	0.2823366	0.3851992
Weight Clustering	From Keras Model	0.0196016	0.5200000	0.6059701	0.2823366	0.3851992
Pruning	Concrete Function Input Signature from Saved Model	0.0140305	0.5800000	0.7303030	0.3351878	0.4594852
Pruning	From Keras Model	0.0140305	0.5800000	0.7303030	0.3351878	0.4594852

Sequence-to-Sequence Anomaly Detection with Voting – Accuracy Metrics:

Compression Technique	Conversion Method	Accuracy (Anomaly Detection Classification)	Precision (Anomaly Detection Classification)	Recall (Anomaly Detection Classification)	F1-Score (Anomaly Detection Classification)
N/A	Concrete Function Input Signature from Saved Model	0.8103704	0.8764228	0.7496523	0.8080960
N/A	From Keras Model	0.8103704	0.8764228	0.7496523	0.8080960
Quantization-Aware Training with Integer Quantization (No Input	Concrete Function Input Signature from Saved Model	0.4896296	0.5384615	0.2920723	0.3787196

Quantization)					
Quantization-Aware Training with Integer Quantization (No Input Quantization)	From Keras Model	0.6437037	0.7203704	0.5410292	0.6179508
Weight Clustering	Concrete Function Input Signature from Saved Model	0.7992593	0.8733333	0.7287900	0.7945413
Weight Clustering	From Keras Model	0.7992593	0.8733333	0.7287900	0.7945413
Pruning	Concrete Function Input Signature from Saved Model	0.8214815	0.8064103	0.8748261	0.8392262
Pruning	From Keras Model	0.8214815	0.8064103	0.8748261	0.8392262

Fire Detection - Binary Classification (CNN):

Method	Post Training Quantization	Test Accuracy	Model Size in KB	Fit Arduino Memory
Quantization Aware	Full Integer Without Input quantization	0.8	87.47	Y
Quantization Aware	Full Integer With Input quantization	0.8	87.18	Y
Quantization Aware	None	0.8	591	X
None	Full Integer Without Input quantization	0.83	86.68	Y
None	Full Integer With Input quantization	0.825	86.51	Y
None	None	0.83	291	N
Pruning	Full Integer Without Input quantization	0.82	87	Y
Pruning	Full Integer With Input quantization	0.82	86.5	Y
Pruning	None	0.84	X	X
Clustering	Full Integer Without Input quantization	0.86	86.6	Y
Clustering	Full Integer With Input quantization	0.855	86.4	Y
Clustering	None	0.86	X	X

Fruit Detection - Multi-Class Classification (CNN):

Method	Post Training Quantization	Test Accuracy	Model Size in KB	Fit Arduino Memory
Quantization Aware	Full Integer Without Input quantization	0.82	87.47	Y
Quantization Aware	Full Integer With Input quantization	0.82	87.18	Y
Quantization Aware	None	0.84	591	X
None	Full Integer Without Input quantization	0.83	86.68	Y
None	Full Integer With Input quantization	0.84	86.51	Y
None	None	0.83	291	N
Pruning	Full Integer Without Input quantization	0.82	87	Y
Pruning	Full Integer With Input quantization	0.82	86.5	Y
Pruning	None	0.84	X	X
Clustering	Full Integer Without Input quantization	0.86	86.6	Y
Clustering	Full Integer With Input quantization	0.84	86.4	Y
Clustering	None	0.88	X	X

Diabetes classification:

Compression Technique	TensorFlow Model Size (Bytes)	TFLite Model Size (Bytes)	TFLite Model Non-Data Buffer Size (Bytes)	TFLite Model Total Data Buffer Size (Bytes)	Final C Array File Size (Bytes)
Quantization + quantization aware training	262,999	15,736	3,524	12,212	97,170
Clustering + quantization aware training	262,999	15,736	3,524	12,212	97,170
Pruning + quantization aware training	262,999	15,736	3,524	12,212	97,170
Quantization	262,999	14,768	2,460	12,308	91,200
Clustering	262,999	14,736	2,428	12,308	91,024
Pruning	262,999	14,768	2,460	12,308	91,205

Compression Technique	Keras baseline gzipped .h5 File Size (Bytes)	Keras Clustered/Pruned gzipped .h5 File Size (Bytes)	TFLite Clustered/Pruned gzipped .tflite File Size (Bytes)	baseline accuracy (in %)	compressed algorithm accuracy (in %)
Quantization + quantization aware training	N/A	N/A	N/A	73.38	69.16
Clustering + quantization aware training	45,026	9,860	11,003	73.38	74.03
Pruning + quantization aware training	45,026	16,223	8,375	73.38	78.57
Quantization	N/A	N/A	N/A	73.38	73.37
Clustering	45,026	10,837	8,513	73.38	76.29

Pruning	45,026	16,165	5,883	73.38	69.81

Bibliography:

K. Rembor, lady ada, D. Halbert, and J. Park, “Adafruit Pybadge and PYBADGE LC,” *Adafruit Learning System*. [Online]. Available: https://learn.adafruit.com/adafruit-pybadge. [Accessed: 01-Aug-2022].

S. Hymel, “Edge Ai Anomaly Detection Part 3 - machine learning on Raspberry Pi,” *Digi-key*. [Online]. Available: https://www.digikey.com/en/maker/projects/edge-ai-anomaly-detection-part-3-machine-learning-on-raspberry-pi/af9dd958b23d4ea1b40bc3cc060ef8c9. [Accessed: 04-Aug-2022].

“Tensorflow Lite for microcontrollers,” *TensorFlow Lite for Microcontrollers*. [Online]. Available: https://www.tensorflow.org/lite/microcontrollers. [Accessed: 04-Aug-2022].

C. Kuo, “Anomaly detection with autoencoder,” *Medium*, 17-Nov-2021. [Online]. Available: https://towardsdatascience.com/anomaly-detection-with-autoencoder-b4cdce4866a6. [Accessed: 04-Aug-2022].

M. Ligade, “Tensorflow Lite deployment,” *Medium*, 28-Nov-2020. [Online]. Available: https://medium.com/techwasti/tensorflow-lite-deployment-523eec79c017. [Accessed: 04-Aug-2022].

“Convert Tensorflow Models: Tensorflow Lite,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/lite/models/convert/convert_models. [Accessed: 04-Aug-2022].