Training up to 50 Class ML Models on 3 $ IoT Hardware via Optimizing One-vs-One Algorithm

Bharath Sudharsan[1,2], **Paul Dolan**[1], **Istvan Maczko**[1], **James Fraser**[1], **Eoghan Kennedy**[1], **Esdras Neto**[1], **Duc-Duy Nguyen**[2], **Piyush Yadav**[2], **John G. Breslin**[2]
[1]ARM ML Infrastructure, Galway, Ireland
[2]Data Science Institute, NUI Galway, Ireland
{bharath.sudharsan, paul.dolan, istvan.maczko, james.fraser, eoghan.kennedy, esdras.neto}@arm.com
{bharath.sudharsan, ducduy.nguyen, piyush.yadav, john.breslin}@nuigalway.ie

Abstract

Multi-class classifier training using traditional meta-algorithms such as the popular One-vs-One (OvO) method may not always work well under cost-sensitive setups. Also, during inference, OVO becomes computationally challenging for higher class counts K as $O\left(K^2\right)$ is its time complexity. In this paper, we present *Opt-OVO*[1], which is an optimized (resource-friendly) version of One-vs-One algorithm which enables high-performance multi-class ML classifier training and inference directly on microcontroller units (MCUs). *Opt-OVO* enables billions of resource-constrained IoT devices to self learn/train (offline) after their deployment, using live data from a wide range of IoT use-cases. We demonstrate *Opt-OVO* by performing live ML model training on 4 popular MCU boards using datasets of varying class counts, sizes and feature dimensions. The most exciting finding was, on the 3 $ ESP32, *Opt-OVO* trained a multi-class ML classifier using a dataset of class count 50 and performed unit inference in super real-time of 6.2 ms.

1 Introduction

The majority of IoT devices such as smart plugs, thermostats, fitness bands, etc. have MCUs as their brain [13]. Ultra-low-power machine learning (TinyML) is a fast-growing research area committed to democratizing ML for commodity MCU-based devices [3]. Currently, MCUs are not capable to train a full ML model due to their resource constraints such as: limited memory, low operations per second, parallel processing inability, etc. The state-of-the-art TinyML studies are rapidly advancing [5, 9] only towards the efficient ML inference on MCUs [14, 7], where the model is first trained on a data center GPU using a historic dataset, then C-code for the trained model is generated and flashed on MCUs [8]. This process impedes the flexibility of billions of deployed ML-powered IoT devices as they cannot learn unseen data patterns (static intelligence) and are impossible to adapt to dynamic scenarios. In this demonstration, we present *Opt-OVO*, a contribution to the TinyML domain by enabling high-performance training and inference on MCUs.

2 Optimized One-vs-One (Opt-OVO) Design

Currently, trainable algorithms are attached to an existing model deployed on MCUs [11, 6, 12] to perform online/continuous learning. The training of a full multi-class ML classifier on commodity MCUs, using any existing algorithms is currently not feasible. When analyzing the OVO method [10], we discovered that the OVO's $k(k-1)/2$ base learners/classifiers, for a few datasets, contain

[1]C++ implementation, performance report, MCU specifications, demo video, etc. of Opt-OVO released at: `https://github.com/bharathsudharsan/Optimized-One-vs-One-Algorithm`

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

classifiers that lack significant contributions to the overall multi-class classification result - this occurs when a classifier is already within a big interdependent group. Hence in *Opt-OVO*, we propose to identify and remove the less important base classifiers to improve the resource-friendliness of OVO.

2.1 Background

Here, we outline the state-of-the-art OVA and OVO methods before providing *Opt-OVO*, which we propose as an optimized extension of the OVO method to enable multi-class classifier training on MCUs.

To provide an MCU-executable multi-classifier training method, we initially considered employing $k-1$ classifiers where each classifier separates points in a particular class C_k from points that do not belong to that class (solves a two-class problem). This approach is the existing *OVA* classifier where the algorithm finds $k-1$ classifiers, i.e., $f_1, f_2, \ldots, f_{k-1}$. To explain in detail, the binary classifier f_1 classifies 1 from $\{2, 3, \ldots, k\}$, the next f_2 classifies 2 from $\{1, 3, \ldots, k\}$, finally the f_{k-1} classifies $k-1$ from $\{1, 2, \ldots, k-2\}$. The input values that are not classified to any classes belong to the k class. There are multiple examples in the literature showing this heuristic method ambiguously classifies regions of input space. This means some input values get classified as they belong to multiple classes. Another drawback is, it creates one model for each class, so $k-1$ models have to be stored in MCU's limited memory, restraining training using large datasets with multiple classes.

To address these limitations, the existing OVO classifier, introduces $k(k-1)/2$ binary classifiers (one for every possible pair of classes), where it finds $k(k-1)/2$ classifiers, i.e., $f_{(1,2)}, f_{(1,3)}, \ldots, f_{(k-1,k)}$. To explain in detail, the binary classifier $f_{(1,2)}$ classifies 1 from 2, the $f_{(1,3)}$ classifies 1 from 3, finally the $f_{(k-1,k)}$ classifies $k-1$ from k. Here, the classification result for each multi-class input is based on the majority vote amongst the employed classifiers, or in other words, the final result is the class with the highest votes.

2.2 Setup

We use a set of base classifiers, B to produce classification results for a multi-class input $x^{(n)}$. We store outputs of the entire base classifiers in R_B. In other words, each classifier $b_i \in B$ is a base learner that produces a result $\in$ {-1, +1}. Therefore, R_B contains outcomes for all the $k(k-1)/2$ binary classifiers over the entire training chunk D_{tr}. We assign l as the class indicator (label) for our problem with $k(k-1)/2$ classes. To understand our setup better, R_B contains the required information to find out which class a given multi-class input $x^{(n)}$ belongs to and also used to compute $P(l_i \mid R_B)$ (Probablity). For example, $x^{(n)}$ and a set B with three base learners, it's R_B is of the form $R_B(x^{(n)}) = \langle -1, +1, -1 \rangle$. Using Bayes theorem,

$$P(y^{(n)} = l_i \mid R_B) = \frac{P(R_B \mid y^{(n)} = l_i) P(y^{(n)} = l_i)}{P(R_B)} \propto P(R_B \mid y^{(n)} = l_i) P(y^{(n)} = l_i), \quad (1)$$

Here, since $P(R_B)$ is common for all classes, it can be suppressed. Hence considering this independence between all the outcomes of base learners in B for a sample input $x^{(n)}$, Eqn. 1 becomes

$$P(y^{(n)} = l_i \mid R_B) \propto \prod_{b_i \in B} P(R_B^{b_i} \mid y^{(n)} = l_i) P(y^{(n)} = l_i), \quad (2)$$

Here, $R_B^{b_i}$ is the classification result of a base learner $b_i \in B$. As shown, using this independence concept simplifies the model, but it might not suit all cases. Hence we relax it and do not use Eqn. 2 in our algorithm design.

When the results of two base classifiers overlap within a tight area for the same input data, we group such correlated classifiers to reduce the number of base classifiers. The *Opt-OVO* algorithm finds groups of correlated base classifiers, and creates a Probability Table (PT) for each group. Hence, the multi-class classification problem is modeled to be conditioned to groups of correlated base classifiers $Corr_{class}$. Hence the model in Eqn. 1 becomes,

$$P(y^{(n)} = l_i \mid R_B, Corr_{class}) \propto P(y^{(n)} = l_i) P(R_B, Corr_{class} \mid y^{(n)} = l_i). \quad (3)$$

We assume independence only among the groups of highly correlated base learners $b_{corr} \subset Corr_{class}$. Therefore, to find the class of an input $x^{(n)}$ we use,

$$class(x^{(n)}) = \arg\max_{j} \prod_{b_{corr} \subset Corr_{class}} P(y^{(n)} = l_j) P(R_B^{b_{corr}}, b_{corr} \mid y^{(n)} = l_j), \tag{4}$$

Here $R_B^{b_{corr}}$ is the outcome of all base learners that belong to the group of highly correlated base classifiers $b_{corr} \subset Corr_{class}$, for training data D_{tr}. To find the groups of correlated base classifiers $Corr_{class}$, we create a correlation matrix C_m to measure the level of correlation between two base classifiers when classifying for a train chunk D_{tr}.

2.3 Core Algorithm

Here, we present the *Opt-OVO*, which we propose as an optimized extension of the state-of-the-art OVO method. From our analysis, we discovered that when using existing OVO, the $k(k-1)/2$ base learners/classifiers, for a few datasets, contain classifiers that lack significant contributions to the overall multi-class classification result. This occurs when a classifier is already within a big interdependent group. Hence, we provide a method, which is a part of the *Opt-OVO* that identifies then removes the less important base classifiers, thus improving the overall resource-friendliness when executing on MCUs.

Algorithm 1 *Opt-OVO* with *SGD* base learners to train multi-class classifiers on MCUs.

1: **Inputs**: Train (D_{tr}) & test (D_{te}) chunks of local dataset D. *SGD* training method.
2: **Output**: Incrementally trained multi-class classifier.
3: **for** each $k(k-1)/2$ base classifiers $b_i \in B$ **do**
4: $\quad Model_i \leftarrow$ Train (D_{tr}, b_i, *SGD*). Train b_i with D_{tr} using *SGD* method.
5: $\quad R_i \leftarrow$ Evaluate (D_{te}, $Model_i$, *SGD*). Evaluate trained $Model_i$ with D_{te} using *SGD*.
6: **end for**
7: $R_B \leftarrow \bigcup R_i$. Store all outcomes R_i of $k(k-1)/2$ binary base classifier b_i in R_B.
8: **Create** a correlation matrix C_m for R_B.
9: **Find** groups of highly correlated base classifiers $Corr_{class}$ from C_m.
10: Using R_B **create** a PT for each highly correlated classifiers groups $b_{corr} \subset Corr_{class}$.
11: **Classifier:** Classify for any new $x^{(n)}$ by using thus obtained set of base classifiers B and $Corr_{class}$ in Eqn. 4.

We present the *Opt-OVO* in Algorithm 1. Here, in Line 3-6, all the $k(k-1)/2$ base classifiers b_i belonging to B are trained with the local data D_{tr} using the *SGD* Algorithm. In the function in Line 4, we provide the flexibility for users to replace our *SGD* method with the base learner of their choice, i.e., SVM, LDA, etc. But we use the *SGD* for its resource-friendly characteristics. In Line 5, we evaluate all the thus trained base classifiers. Here, each base classifiers b_i produce a binary output $\in \{-1, +1\}$ for each input vector $x^{(n)}$. In Line 7, for all data in D_{te}, we store outcomes of the $k(k-1)/2$ base learners R_i in R_B. Next, in Line 8, we create a correlation matrix C_m using the output of base classifiers stored in R_B. In Line 9, from C_m, we find $Corr_{class}$, which is the group of highly correlated base classifiers. In Line 10, from the groups of this found correlated base learners, we create a PT of each group to know the joint probability of the outcome R_B. These PTs provide the joint probabilities of the outcomes R_B and the groups of correlated classifiers $b_{corr} \subset Corr_{class}$ when evaluating using new/unseen data. In the final Line 11, we classify for any new multi-class input $x^{(n)}$ by using the algorithm produced set of base classifiers B and $Corr_{class}$ in Eqn 4.

Next, we explain our method to find the groups of highly correlated base classifiers $Corr_{class}$ from correlation matrix C_m. For a multi-class training chunk D_{tr}, we measure the level of correlation between two base classifiers by considering their binary classification result $\in \{-1, +1\}, R_i, R_j$. Where R_i & $R_j \in R_B$ are outcomes of base classifiers b_i & b_j and its correlation matrix C_m is given as,

$$C_{m(i,j)} = \frac{1}{N} \left| \sum_{\forall x^{(n)} \in D_{tr}} R_i(x^{(n)}) R_j(x^{(n)}) \right|, \tag{5}$$

Here, if both the base classifiers produce the same output for all the data points in D_{tr}, then the level of correlation between them is one, the highest, so, $R_i = R_j$. In cases when their outputs

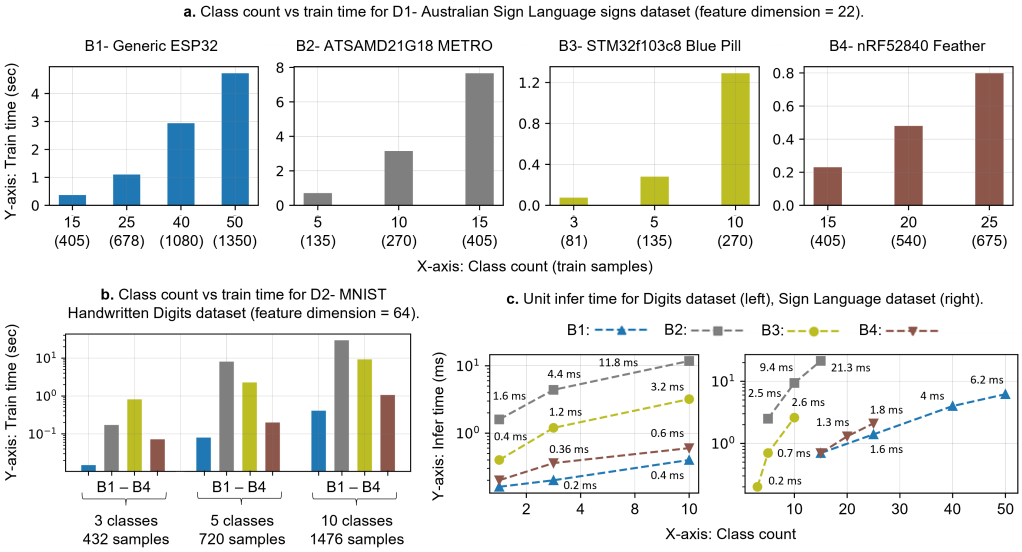

Figure 1: High performance ML model training and inference on MCU boards using *Opt-OVO*.

always don't match $R_i \neq R_j$, again their correlation is one. Whereas if the base classifiers have half outputs matching and rest not equal, then the correlation is zero. We use this method in the *Opt-OVO* Algorithm 1 to group similar output producing base binary classifiers.

3 Demonstration and Results

We show audience the live ML model training on MCU process with high performance and accurate results transparently from Serial port of MCUs. We select datasets D1 [1], D2 [2] using which *Opt-OVO* trains classifiers on 4 tiny MCU boards B1-B4. The training performance is presented in Fig 1. a-b. Here, even on the slowest B2, *Opt-OVO* trained using 10 classes D2 in 29.6 sec and 7.6 sec using the 15 classes data of D1. The 3 $ B1 trained in 0.4 sec for D2 and in 4.7 sec using the 50 classes data of D1. We present inference performance in Fig 1. c. Here, even for high 64 dimensional D2, *Opt-OVO* achieves real-time unit inference of 11.8 ms, even on the slowest B2. The cheapest 3 $ B1 was able to infer for a 50 class input in 6.2 ms. Overall, *Opt-OVO* performed unit inference for multi-class data in super real-time, within a second, across B1-B4. We also report that *Opt-OVO* trained models achieve similar accuracies as the *Python scikit-learn* models.

4 Conclusion and Future Work

We presented *Opt-OVO* algorithm, which achieves reduced computation than OVO by identifying and removing base classifiers that lack significant contributions to the overall multi-class classification result. As demonstrated, *Opt-OVO* enables high-performance ML model training and inference on MCUs. Researchers and engineers can leverage its open-sourced implementation to transform even the resource-constrained devices into intelligent systems that can self-learn by locally re-training themselves using the unseen/fresh data patterns it encounters after deployment. *Opt-OVO* can be used as a key component to enable practicing split-learning, distributed ensemble learning, federated learning, centralized learning by involving even the resource-constrained devices in complex ML-based learning tasks. In future work, similar to the TinyML benchmark [15], we plan to use more sophisticated datasets [4] and conduct extensive onboard training plus inference experiments involving the latest pocket-friendly FPGAs, SoCs and AIOT boards.

References

[1] Australian sign language signs (high quality) data set: `https://archive.ics.uci.edu/ml/datasets/Australian+Sign+Language+signs+(High+Quality)`.

[2] The mnist database of handwritten digits: `http://yann.lecun.com/exdb/mnist/`.

[3] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. In *Proceedings of Machine Learning and Systems (MLSys)*, 2021.

[4] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[5] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR)*, 2020.

[6] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. Tinytl: Reduce memory, not parameters for efficient on-device learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2020.

[7] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Shlomi Regev, et al. Tensorflow lite micro: Embedded machine learning on tinyml systems. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

[8] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. Compiling kb-sized machine learning models to tiny iot devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 79–95, 2019.

[9] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

[10] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE transactions on Neural Networks*, 13(2):415–425, 2002.

[11] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. In *Advances in Neural Information Processing Systems (NIPS)*, 2020.

[12] Haoyu Ren, Darko Anicic, and Thomas Runkler. Tinyol: Tinyml with online-learning on microcontrollers. *arXiv preprint arXiv:2103.08295*, 2021.

[13] Bharath Sudharsan, Pankesh Patel, John Breslin, Muhammad Intizar Ali, Karan Mitra, Schahram Dustdar, Omer Rana, Prem Prakash Jayaraman, and Rajiv Ranjan. Toward distributed, global, deep learning using iot devices. *IEEE Internet Computing*, 25(03):6–12, 2021.

[14] Bharath Sudharsan, Pankesh Patel, John G Breslin, and Muhammad Intizar Ali. Enabling machine learning on the edge using sram conserving efficient neural networks execution approach. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD)*, 2021.

[15] Bharath Sudharsan, Simone Salerno, Duc-Duy Nguyen, Muhammad Yahya, Abdul Wahid, Piyush Yadav, John G Breslin, and Muhammad Intizar Ali. Tinyml benchmark: Executing fully connected neural networks on commodity microcontrollers. In *IEEE 7th World Forum on Internet of Things (WF-IoT)*, 2021.