

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

A. N. Mazumder and T. Mohsenin, "Reg-TuneV2: Hardware-Aware and Multi-Objective Regression-Based Fine-Tuning Approach for DNNs on Embedded Platforms," in IEEE Micro, doi: 10.1109/MM.2023.3316433.

<https://doi.org/10.1109/MM.2023.3316433>

Access to this work was provided by the University of Maryland, Baltimore County (UMBC) ScholarWorks@UMBC digital repository on the Maryland Shared Open Access (MD-SOAR) platform.

Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it's important to you. Thank you.

THEME ARTICLE: tinyML

Reg-TuneV2: Hardware-Aware and Multi-Objective Regression-Based Fine-Tuning Approach for DNNs on Embedded Platforms

Arnab Neelim Mazumder, *University of Maryland Baltimore County, MD, 21250, USA*

Tinoosh Mohsenin, *John Hopkins University, MD, 21218, USA*

Abstract—*Fine-tuning Deep Neural Networks (DNNs) for deployment has traditionally relied on computationally intensive methods such as grid search and neural architecture search (NAS), which may not consider hardware-aware metrics. Moreover, it is essential to consider multiple objectives to develop a range of solutions for tinyML hardware deployment with real-time latency and low power constraints. To address these problems, we propose Reg-TuneV2, a systematic approach to fine-tune DNNs for hardware deployment by considering multiple objectives, including accuracy, power, and latency contours. In addition, this approach uses metric learning to achieve smaller and better-suited configurations for deployment, achieving 90.5% accuracy with only 340 KB of memory for keyword spotting on FPGA. When compared to baselines for keyword spotting and image classification on the Nvidia Jetson Nano 4 GB SDK, the proposed method achieves a 14.5 \times and 101.8 \times reduction in model size coupled with 2.5 \times and 5.9 \times better inference efficiency, respectively.*

In recent years, extensive research has demonstrated that an increase in the performance of Deep Neural Networks (DNNs) is heavily reliant on the size of the network. However, this performance improvement is accompanied by a substantial increase in energy consumption. As a result, finding a solution is required to retain the network's standard performance while considerably reducing its size. For any deployment, regardless of the target platform, the designer must adjust the network to adhere to device limitations, known as fine-tuning. The fine-tuning process can be either homogeneous or heterogeneous. Traditional heterogeneous techniques involve search algorithms like grid search, random search, and Neural Architecture Search (NAS) [1], [2] to optimize DNNs at a granular level, demanding substantial computational time. In contrast, homogeneous methods involve intuitive fine-tuning of DNN layers through a single parameter, achieving near-optimal configurations with fewer computations. This study employs a homogeneous fine-tuning approach using reasoned polynomials to opti-

mize for power and latency costs. Additionally, there is limited research that considers hardware parameters and metrics in the context of NAS [2], [3], [4] or fine-tuning approaches [5], [6] to form a hardware-in-the-loop tuning problem. The fine-tuning process proposed in this study draws inspiration from NAS [1], [2], [4] techniques, but with different objectives. Rather than aiming for an optimal solution, the purpose is to obtain a near-optimal configuration that meets multiple deployment objectives. Unlike NAS techniques, this problem does not require a brute-force search mechanism. Instead, it can be solved using simple polynomial regression. We propose a proxy exploration approach in Reg-TuneV2 that allows users to obtain near-optimal solutions while exploring a reduced design space. This approach involves using regression to train models for different configurations of variables to profile accuracy, power, or latency. The resulting contours can help users select near-optimal solutions or choose a range of variables for more focused search methods to pinpoint the exact optimal configuration.

In a previous study [3], a heuristic fine-tuning approach for accelerating DNNs was introduced. This approach generated resource-efficient hardware representations under FPGA (Field Programmable Gate

XXXX-XXX © 2023 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

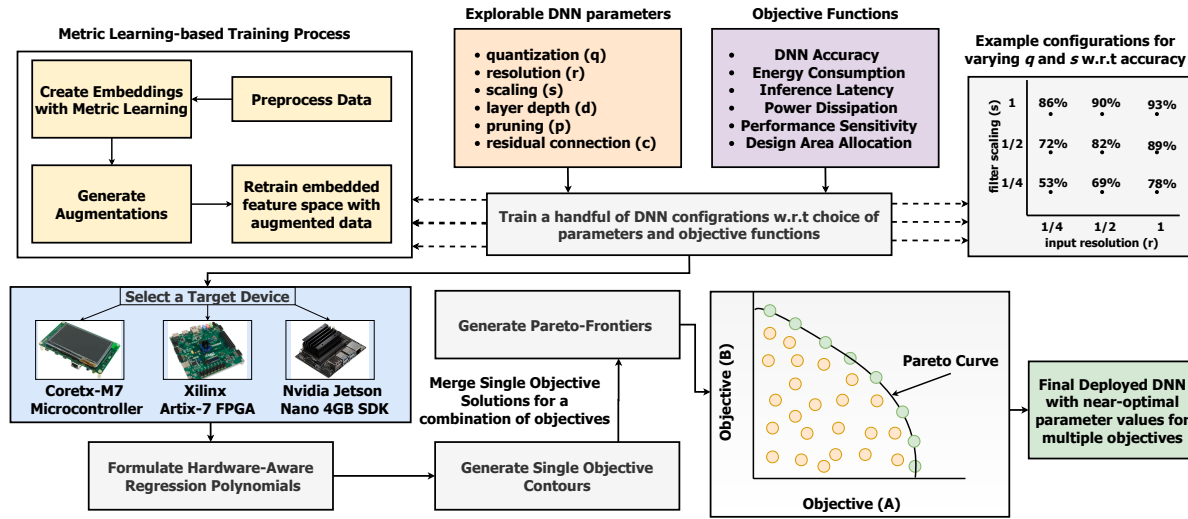


FIGURE 1. Overview of Reg-TuneV2 for multiple-objectives with a metric-learning-based training approach. By combining the solutions achieved for individual objectives (shown in the pink block), we can generate Pareto frontiers that highlight the feasible configurations, providing the best trade-off for a combination of objectives.

Array) on-chip memory constraints but was specific to FPGA profiling. However, fitting medium to large-sized models onto low-power FPGAs without using off-chip memory sources can be challenging. To overcome this bottleneck, accelerating DNNs on mobile edge platforms with GPUs (Graphics Processing Units), such as the Nvidia Jetson series, can be a viable solution for real-time inference in various applications. Our previous works [5], [6] have demonstrated the effectiveness of the fine-tuning technique for low power consumption on FPGAs and various applications. Building upon these earlier developments, we introduce the profiling process for Nvidia Jetson Nano 4 GB SDK (Software Development Kit) and discuss the modifications necessary to accommodate fine-tuning for multiple objectives. Additionally, to further improve our methodology's efficacy, we explore using metric learning to extract better features from the DNN training process without changing the DNN parameters. The significant contributions include:

- Introduction of Reg-TuneV2, a systematic approach for selecting near-optimal hardware configurations based on accuracy, power, and latency contours through polynomial regression.
- Exhaustive breakdown of the polynomial formulation required for profiling different metrics on different embedded platforms.
- An analytical process to create Pareto frontiers for multiple objectives leading to near-optimal solutions for hardware deployment.

- Inclusion of metric learning into the training process to create more general models, validated through two case studies: keyword spotting (KWS) with Google Speech Commands (GSC) [7], and image classification with CIFAR-10 [8].
- The metric learning-based accuracy profile coupled with multi-objective solutions complement the fine-tuning approach to obtain configurations for hardware deployment, resulting in $14.5\times$ and $101.8\times$ compression in model size and $2.5\times$ and $5.9\times$ inference efficiency improvement compared to baselines for GSC and CIFAR-10, respectively, on the Nvidia Jetson Nano 4 GB SDK. Similarly, we achieve a better accuracy of 90.5% with Reg-TuneV2 for the same configuration chosen in [6] in terms of FPGA deployment.

Problem Formulation and Reg-TuneV2 Framework

We consider a two-dimensional (2D) convolutional neural network (CNN) block that takes an input with a resolution factor of r . To account for the impact of different filter sizes on the accuracy, we introduce a scaling variable (s) to our 2D CNN formulation, resulting in $\mathcal{NN}(r, s)$. This scaling variable modifies the number of filters in each convolutional layer by a factor of s . In summary, we define the problem as follows:

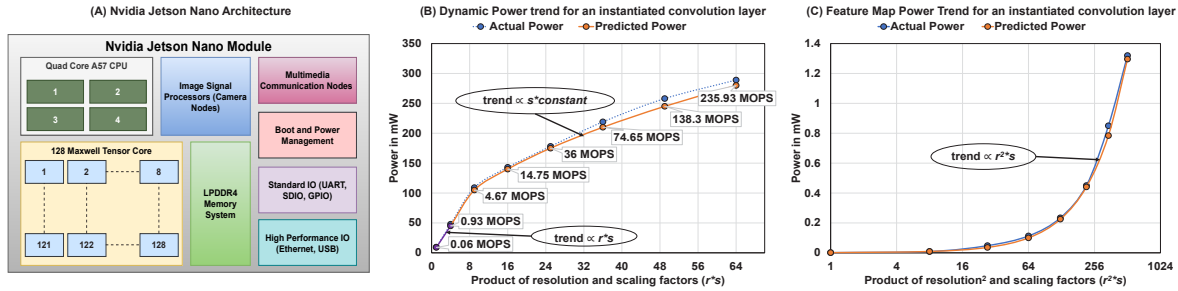


FIGURE 2. (A) depicts an overview of the Nvidia Jetson Nano module with CPU and GPU core blocks, memory system, sensor nodes, and general purpose I/Os. (B) and (C) illustrate the power trend on Nvidia Jetson Nano 4 GB SDK GPUs for execution and feature maps with varying r and s . The results are for a convolution layer with 16×16 input and 5 filters as baseline ($r = 1, s = 1$). r and s are increased as a factor of 16 r and 5 s .

$$\begin{aligned} & \text{minimize } \mathcal{NN}(\text{Power}) \text{ for } \text{config} \in \{r, s\} \\ & \text{minimize } \mathcal{NN}(\text{Latency}) \text{ for } \text{config} \in \{r, s\} \\ & \text{s.t. Accuracy}(\mathcal{NN}) \geq \text{target_accuracy} \end{aligned} \quad (1)$$

This minimizes the power and latency generated for deploying these models, provided that the target accuracy is maintained for the deployed configuration. To enhance the performance of DNNs, we employ metric learning in addition to the fine-tuning process. The fundamental objective of metric learning is to learn feature vectors that reduce the distances between similar examples and increase the distances between dissimilar examples in the embedded space. Once the space is constructed, a softmax layer can be added to classify the data. One example of metric learning is the triplet loss, which is formulated in a way that aims to minimize such distances as described by [9].

$$\text{Triplet Loss} = \text{sum}(\max(d(A, P) - d(A, N) + \alpha, 0)) \quad (2)$$

Here, the model learns three embeddings, i.e., anchor, positive, and negative. First, the anchors are used as a reference to create batches of similar (positive) or dissimilar (negative) samples. Next, the distance between anchor-positive ($d(A, P)$) and anchor-negative ($d(A, N)$) embeddings is used as the loss. As a result, we can generate samples belonging to different classes clustered together in feature space, as shown in Figure 4. And, α , in this case, acts as a margin and ensures that the difference does not become negative. In the context of this work, the triplet loss model consists of all Convolutional Neural Network (CNN) layers followed by the first fully connected (FC) layer to create the one-dimensional feature space for the UMAP plots. The embedding is further fine-tuned for 10 epochs with a couple more fully connected layers, as shown in Equation 8. Finally, Figure 1 depicts the major steps of

our approach. The first step is to preprocess the input data as required before feeding it to the DNNs. Next, we train several different model configurations, each corresponding to a different set of DNN parameters, using metric learning. The trained configurations act as data points for polynomial regression, approximating the relationship between the DNN parameters and various metrics of interest, such as accuracy, power, and latency. This approximation has some margin of error but allows us to predict the performance of unknown configurations. Using interactive contours, we can then shuffle between different profiles of objectives to identify the configuration that works best for a single objective. By merging the solutions obtained for multiple single objectives, we create Pareto-frontiers, which identify configurations that are suitable for at least two objectives while satisfying the given constraints.

Forming the Polynomials

We must create regression polynomials that are specific to the target platform to analyze various metrics. These polynomials will help us forecast the accuracy, power consumption, and latency of a particular DNN running on the device.

Nvidia Jetson Nano

To create an **accuracy** polynomial, we used the equation developed in [6] that effectively represents the accuracy trend. Our goal is to approximate the accuracy level of a specific DNN with a certain degree of confidence by incorporating a custom equation that utilizes the least-square error method. This process involves using polynomial regression to capture the relationship between two independent variables on the XY plane, as represented in Equation (3). Furthermore, we aim to structure Equation 3 to adopt a rational

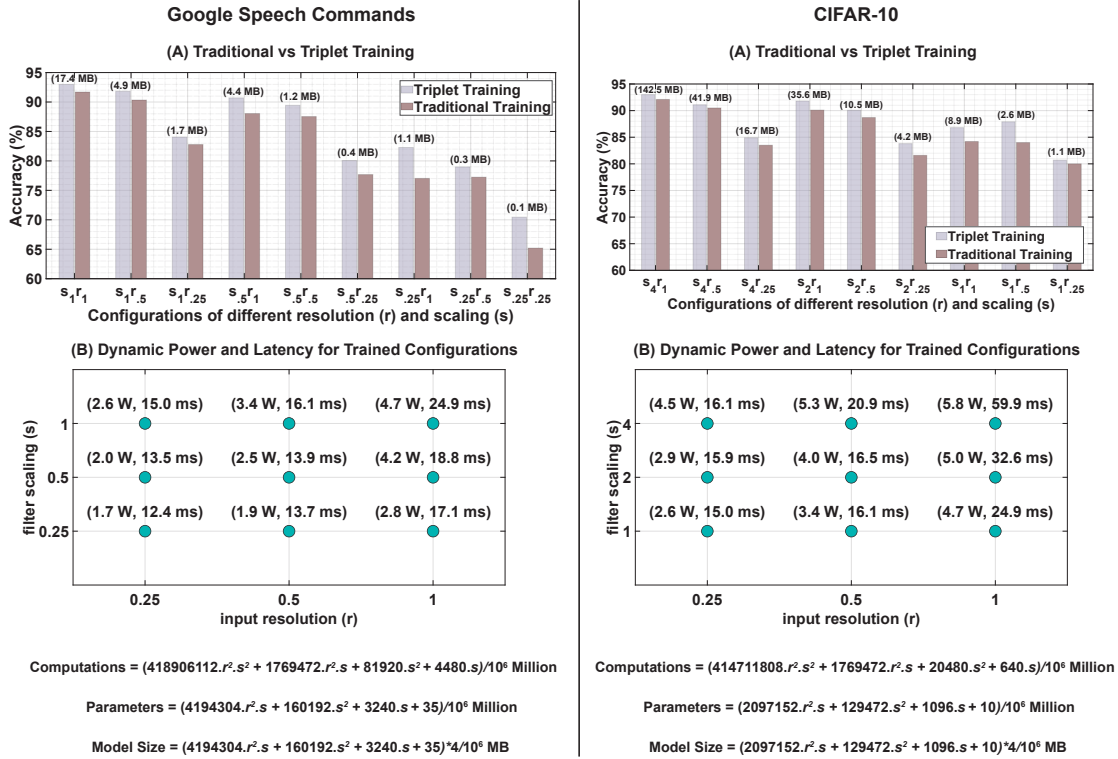


FIGURE 3. (A) represents accuracy and model size for different configurations with respect to both metric and traditional learning for KWS on GSC and image classification on CIFAR-10. (B) depicts the dynamic power and latency of running inference of the same configurations on Nvidia Jetson Nano 4 GB SDK.

format where the numerator and denominator have similar representations of the XY plane. Additionally, we observed that the accuracy tends to increase as the values for the variables (r, s) increase. The R^2 , adjusted R^2 , and RMSE values using the accuracy polynomial turn out to be 0.999, 0.998, and 0.341, respectively, indicating a good fit for the predicted data in relation to the actual data.

$$Accuracy(\mathcal{NN}(q, s)) \approx \frac{\hat{A}_6.q.s + \hat{A}_5.s + \hat{A}_4.q + \hat{A}_3}{q.s + \hat{A}_2.s + \hat{A}_1.q + \hat{A}_0} \quad (3)$$

$$Dynamic Power_{\mathcal{H}\mathcal{V}} \approx Power_{Memory} + Power_{Execution} \quad (4)$$

The dynamic **power** polynomial formulation focuses on identifying the power trend for execution and memory, as represented in Equation 4 on Nvidia Jetson Nano 4 GB SDK (referred to as the 'target device' from here on). In the context of our work, we aim to generate a power trend for the target device, where execution power is mainly derived from computations (as shown in Figure 2(B)), feature maps (as shown in Figure 2(C)), and static power. To generate the

trend for power due to computations, we conduct 100 iterations of the inference pipeline and take the average of the rounds as our power and latency numbers using the Tegrastats utility tool. We also increase the resolution by a factor of r , where $r = 2$ corresponds to an input of 32×32 . We scale the filters by a factor of s , where $s = 2$ corresponds to a filter size of 10. We also operate the device in the maximum performance mode, which uses a scheduling process to optimize CPU (Central Processing Unit) and GPU resources for maximum performance. The target device consists of a quad-core A57 CPU and a 128 Maxwell Tensor Core GPU, as shown in Figure 2(A). The static power for the device when peripherals such as a keyboard, mouse, and monitor are connected is around 2.2 W. Our experiments indicate that the dynamic power trend due to computations follows $r.s$ up to 1 million operations, after which it becomes proportional to s multiplied by a constant. The power due to feature maps, however, follows the trend of $r^2.s$. Compared to the actual experimental data, the predicted data have an R^2 -value and adjusted R^2 -value of 0.995 and 0.994, respectively, for power due to computations. In the case of power

coming from the feature maps, the fit described R^2 -value and adjusted R^2 -value turn out to be 0.997 and 0.996, respectively. Since R^2 and adjusted R^2 -values are a statistical measure of the variance between the actual and predicted values, any number around 0.99 suggests a very good fit with 99% of the variance in the data being captured by the model. The same formulations hold true for FC layers. Based on these observations, we formulate the power polynomial as follows:

$$\text{Dynamic Power}(\mathcal{HW}|\mathcal{NN}(r, s)) \approx \hat{B}_2.r^2.s + \hat{B}_1.s + \hat{B}_0 \quad (5)$$

Here, \hat{B}_0 , \hat{B}_1 , and \hat{B}_2 are learnable parameters. \hat{B}_0 represents the dynamic power due to factors unrelated to r and s .

Latency polynomials conform to the combination of latency approximators for different layers included in the DNN. For example, traditional layers such as CNNs, RNNs (Recurrent Neural Networks), and FCs follow different latency trends during execution. To that extent, we use the time complexity of the layers involved in our implementations on the edge device to generate the latency polynomial. In regular convolution layers, execution time changes in proportion to $r^2.s^2$, whereas for FC and maxpool layers, this is defined in proportion to s and r , respectively. The final latency polynomial is as follows:

$$\text{Latency} \approx \hat{C}_3.r^2.s^2 + \hat{C}_2.s + \hat{C}_1.r + \hat{C}_0 \quad (6)$$

Here, \hat{C}_0 , \hat{C}_1 , \hat{C}_2 , and \hat{C}_3 are learnable parameters. \hat{C}_0 represents the delay due to factors unrelated to r and s . Furthermore, we focus on performing on-device inference, and as a result, the communication power and latency in our case are negligible. However, this may not be the case for situations where the device communicates with different sensors for either execution or action. As a result, it will require an alternate formulation to encapsulate the communication trend.

FPGA

The polynomial for FPGA power is derived from the accelerator developed in our previous work presented in [6], which considers the power consumption of multiplication and addition operations, feature maps, and static power. The polynomial generated with respect to the parameters of quantization (q) and filter scaling (s) is depicted as follows:

$$\text{Power}(\mathcal{HW}|\mathcal{NN}(q, s)) \approx \hat{D}_3.q^2.s^2 + \hat{D}_2.q.s^2 + \hat{D}_1.q.s + \hat{D}_0 \quad (7)$$

Power in Equation 7 includes terms proportional to $q.s$ for memory, $q^2.s^2$ for multipliers, $q.s^2$ for adders,

with \hat{D}_0 for static power and power coming from factors unrelated to q and s . To profile FPGA latency, we use a fixed configuration of 72 processing engines and 8 multipliers, as in [6], and represent the latency of the accelerator as a function of the time complexity of the layers using Equation 6.

Experimental Setup

The experiments were conducted on two datasets, GSC and CIFAR-10. GSC consists of 35 classes of audio keywords used for wake-word detection and keyword spotting, while CIFAR-10 comprises 50K training and 10K testing RGB images of animals and vehicles, each with dimensions of 32×32 pixels and divided into 10 classes. For GSC, a VGG-like DNN was adapted (Equation 8), denoted by $\mathcal{NN}(r, s)$, where r and s are variables representing input resolution and filter scaling. Here, $C_{3 \times 3}$, $pool_{2 \times 2}$ and FC denote convolution, maxpool, and FC layers, respectively. The input resolution was varied from 0.25 to 1, while the filter scaling was homogeneous across all layers. Similarly, for CIFAR-10, an adapted VGG-like DNN (Equation 8) was used, where the input resolution was varied from 1 to 4 and the filter scaling from 0.25 to 1. The computations, parameters, and model size equations for both case studies are shown in Figure 3. The only difference between the two architectures is the last few dense layers. For GSC, these layer corresponds to the configuration of $256s - 128s - 64s - out$, whereas for CIFAR-10, this corresponds to $128s - 64s - 32s - out$.

$$\begin{aligned} & (64r \times 64r) - 2 \times (8s C_{3 \times 3}) - 2 \times (16s C_{3 \times 3}) - \\ & 3 \times (32s C_{3 \times 3}) - (32s pool_{2 \times 2}) - 3 \times (64s C_{3 \times 3}) - \\ & (64s pool_{2 \times 2}) - (256s \text{ or } 128s FC) - (128s \text{ or } 64s FC) - \\ & (64s \text{ or } 32s FC) - (out) \end{aligned} \quad (8)$$

We trained all models using metric learning to create clustered embeddings for 100 epochs, followed by retraining using a categorical loss function. During this phase, we froze the layers responsible for creating the embedding and added a few fully connected layers for classification. We retrained for 10 epochs for GSC, while for CIFAR-10, we required 100 epochs to converge. We used the Adam optimizer with an initial learning rate of 0.001, which decays by 0.1 every 40 epochs. Additionally, we used general data augmentation techniques for CIFAR-10 and created spectrograms from the raw audio for GSC. We evaluated the VGG-like architectures' accuracy for nine different configurations with model sizes ranging from 0.11 MB to 17.43 MB for GSC and 1.05 MB to 142.54 MB for CIFAR-10. We deployed these configurations on the

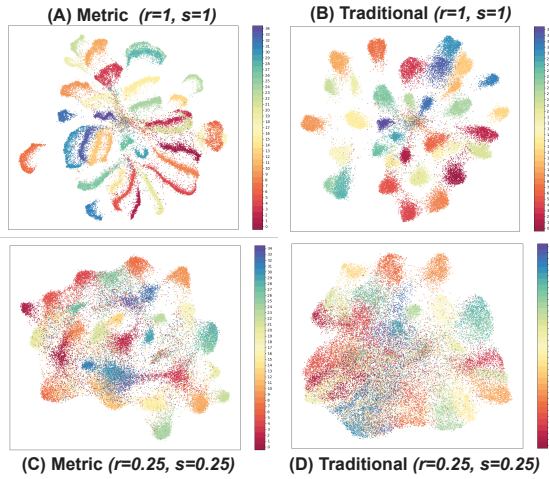


FIGURE 4. GSC UMAP representation of the embedded feature space for $\mathcal{NN} \langle r=1, s=1 \rangle$ and $\mathcal{NN} \langle r=0.25, s=0.25 \rangle$ with metric learning (A), (C) and traditional learning (B), (D) respectively. Here, r and s correspond to input resolution and filter scaling, respectively.

target device to measure dynamic power and latency, as shown in Figures 3(B) for GSC and CIFAR-10. Figures 3(A) demonstrate that metric learning-based training improves accuracy by at least 1.5-2% compared to traditional methods for both GSC and CIFAR-10. We also validated these results using Figure 4, which displays the embeddings of the largest and smallest configurations for GSC. Figure 4(A) shows clearly separated clusters with minimal overlap compared to Figure 4(B). The feature space is more dispersed for the smallest configuration, as seen in Figures 4(C) and (D). However, even in this case, metric learning-based embeddings are more structured than their traditional counterparts. This suggests that with metric learning-based training, we can achieve the same level of accuracy as traditional methods with reduced DNN scale and input resolution. Consequently, deployment overhead will significantly reduce without changing parameters.

Regression Results

To predict the values of accuracy, dynamic power, and latency with a degree of confidence, we have performed polynomial regression using Equations 3, 5, 6 on our 9 data points depicted in Figures 3(A) for both GSC and CIFAR-10. The regression approximator's coefficients were determined for both GSC and CIFAR-10 datasets, and the accuracy of the approximator was evaluated using the root mean square error (RMSE) values. For the GSC dataset, the RMSE values were

TABLE 1. Verification of regression fits with new data points of $\mathcal{NN} \langle r=0.55, s=0.85 \rangle$ and $\mathcal{NN} \langle r=0.55, s=1.6 \rangle$ for GSC and CIFAR-10 respectively

Metrics	GSC		CIFAR-10	
	Actual	Predicted	Actual	Predicted
Acc. (%)	91.3	91.6	90.0	90.1
Dyn. Pwr. (W)	3.5	3.3	3.9	3.7
Lat. (ms)	17.1	16.9	19.8	19.2

0.34% for accuracy, 0.29 W for dynamic power, and 0.64 ms for latency. Similarly, for CIFAR-10, the RMSE values were 1.07% for accuracy, 0.38 W for dynamic power, and 1.28 ms for latency. The contour plots representing the trends of the metrics with variables r and s are illustrated in Figures 5(A) through (C) for both case studies, where the vertical and horizontal lines signify the points chosen by Pareto-Frontiers, as explained in the following section. To test the regression approach's effectiveness, we have used actual accuracy values from new data points for both case studies with all metrics shown in Table 1. As evident, the predictions are not too far apart from the actual values.

Multi-Objective Solutions

We extend the scope of our single objective regression solutions for accuracy, dynamic power, and latency by merging them to form multi-objective solutions. Such multi-objective solutions, when illustrated as scatter plots, can represent Pareto-Frontiers. Pareto-Frontiers represent a set of non-dominated solutions that are superior to all other solutions in the search space. This means that no single solution is superior to all others with respect to all objectives, and changing variables in the Pareto set could not improve all objectives simultaneously. Solutions thus generated are usually located at the edge of the metric design space and form a line known as the Pareto curve. To create the Pareto-Frontiers, we first generated single objective regression polynomials for our metrics of interest. Then, we used these to predict the metrics for untrained configurations of r and s . We then merged the dynamic power and latency polynomials by substituting r as a function of s and *Accuracy* in Equation 5 and 6, creating two equations that allowed us to infer dynamic power and latency values for a range of r and s given an *Accuracy* constraint. Finally, we plotted these values using a scatter plotting tool, with dynamic power on the X-axis, latency on the Y-axis, and different accuracy ranges represented by different colors.

From the resulting Pareto-Frontiers shown in Figures 6(A) and (B), we aimed to identify the solutions in the plot's top right-hand corner, representing con-

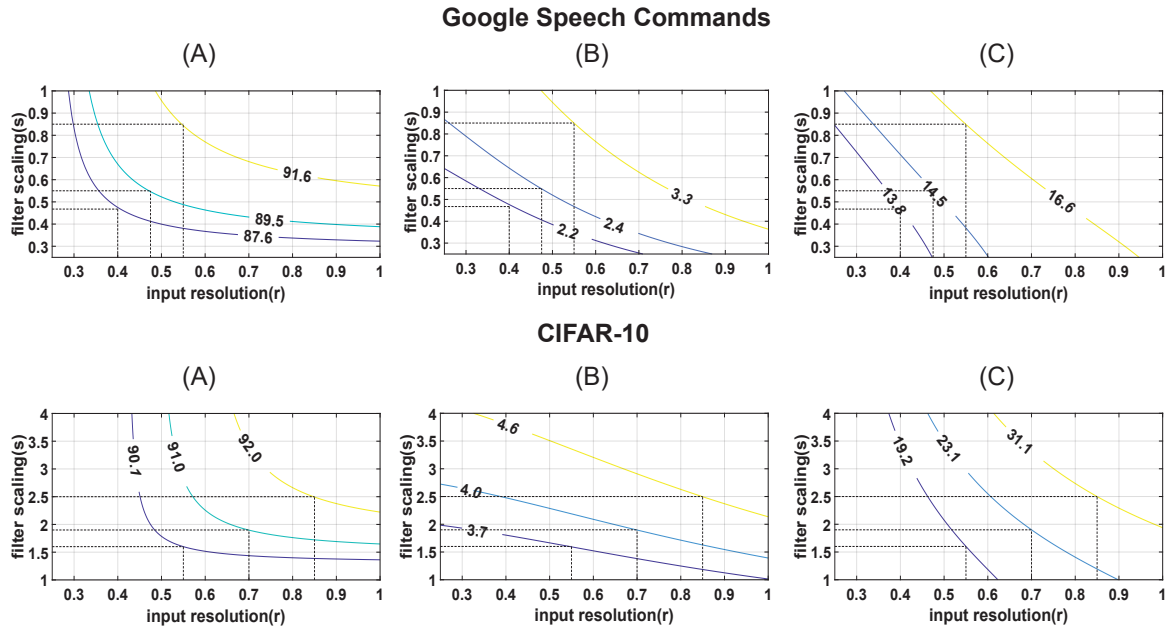


FIGURE 5. Contour profiles generated through regression for accuracy in % (A), dynamic power in W (B), and latency in ms (C) within the span of r and s for KWS on GSC and image classification on CIFAR-10.

TABLE 2. Comparison of \mathcal{NN} ($r=0.55, s=0.85$) and \mathcal{NN} ($r=0.55, s=1.6$) for GSC and CIFAR-10 with baseline and existing works

Dataset	GSC			CIFAR-10			
Work	[10]	Baseline	Reg-TuneV2	[11]	[12]	Baseline	Reg-TuneV2
Model	CNN	Adapted VGG		AlexNet	ResNet-50	Adapted VGG	
Precision	8-bits	32-bits		32-bits			
Accuracy (%)	87.6	93	91.3	82.6 (Top-5)	93.2	93	90.0
Device	Cortex-M7 Microcontroller	Nvidia Jetson Nano 4 GB SDK		Nvidia Jetson Nano 4 GB SDK			
Model Size (MB)	0.5	17.4	4.8	249.2	241.8	142.5	5.4
Dynamic Power (W)	-	4.7	3.5	-	-	5.8	3.9
Total Power (W)	-	6.9	5.7	-	-	8	6.1
Latency (ms)	100	24.9	17.1	31.3	37.0	60.0	19.8
Frame Rate (FPS)	10	40.2	58.5	31.9	27	16.7	50.5
FPS/Pwr (Inference/J)		5.8	10.3	-	-	2.1	8.3

figurations with low dynamic power and low latency. However, we also considered accuracy. If a data point belonging to a higher accuracy range was in the vicinity of a data point belonging to a lower accuracy range, the higher accuracy data point was considered a dominating solution and preferred over the other solutions. Using these criteria, we identified a Pareto Curve, which represented the set of non-dominated solutions along a highlighted black line in the Pareto-Frontiers. For example, for GSC, we identified the best solution

for achieving 87.5% accuracy as having a dynamic power of 2.17 W and a latency of 13.79 ms and used the same criteria to identify solutions for 89.5% and 91.6% accuracy. Similarly, for CIFAR-10, we identified a solution for achieving 90.1% accuracy with a dynamic power of 3.71 W and a latency of 19.24 ms.

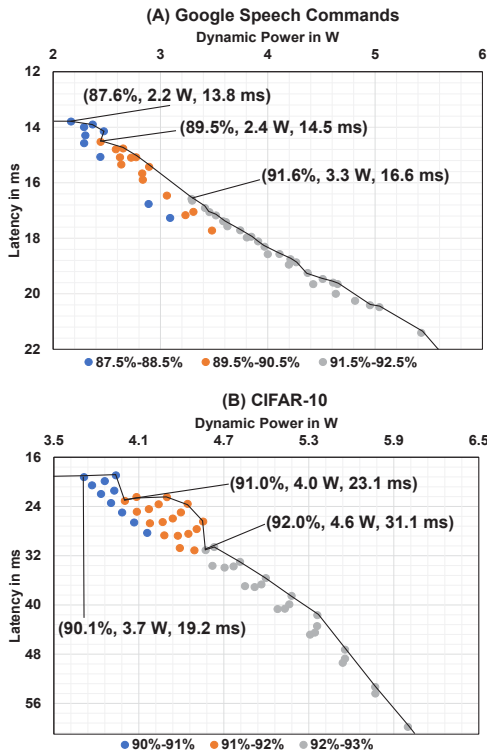


FIGURE 6. Pareto-frontiers created through merging single objective solutions of dynamic power and latency with the constraint of accuracy for KWS (A) and image classification (B). The points highlighted are the chosen configurations for deployment.

TABLE 3. Results for 8-bit quantization of $\mathcal{NN}_{(r=0.55, s=0.85)}$ and $\mathcal{NN}_{(r=0.55, s=1.6)}$ for GSC and CIFAR-10 respectively along with FPGA implementation using the accelerator from [6] when processed through the RegTuneV2 framework.

Dataset	GSC		CIFAR-10
Device	Artix-7 200t FPGA	Nvidia Jetson Nano 4~GB SDK	
Precision	4-bit	8-bit	
Accuracy (%)	90.5	91.0	89.6
Model Size (MB)	0.34	1.2	1.4
Total Power (W)	0.83	4.4	4.7
Latency (ms)	0.7	15.5	17.1
Frame Rate (FPS)	1429	64.5	58.5
FPS/Pwr (Inference/J)	1722	14.7	12.5

Comparison with Existing Works

Our experiments aim to accelerate large image classification and keyword spotting DNNs to tinyML platforms with limited memories and low-power envelopes. The configurations we deploy for GSC and CIFAR-10 are denoted by $\mathcal{NN}_{(r=0.55, s=0.85)}$ and $\mathcal{NN}_{(r=0.55, s=1.6)}$, respectively. These points were chosen from the Pareto frontier plots shown in Figures 6(A) and (B), with the actual values for these configurations depicted in Table 1. In comparison to the baseline implementation of $\mathcal{NN}_{(r=1, s=1)}$ for GSC, the chosen data point ($\mathcal{NN}_{(r=0.55, s=0.85)}$) achieved a $3.6\times$ reduction in model size with negligible accuracy difference and a $1.8\times$ improvement in inference efficiency (FPS/Pwr) through the proposed fine-tuning setup. Moreover, compared to the KWS task deployed on microcontrollers in [10], our approach achieved approximately $6\times$ better frame rate and a 3.7% accuracy boost. For CIFAR-10, the chosen data point ($\mathcal{NN}_{(r=0.55, s=1.6)}$) achieved $26.4\times$ compression in model size with 90% accuracy and almost $4\times$ better inference efficiency (FPS/Pwr) compared to the baseline implementation of $\mathcal{NN}_{(r=1, s=4)}$ through the proposed fine-tuning setup. The accuracy profile for the regression setup was created using metric learning. To achieve the same accuracy level of 90% with traditional training, we had to increase the resolution and scaling, resulting in a configuration of $\mathcal{NN}_{(r=1, s=2)}$ that was $6.6\times$ bigger (35.6 MB). Furthermore, compared to image classification tasks based on CIFAR-10 deployed on the target device as described in [11] and [12], our approach achieved approximately $1.6\times$ and $1.9\times$ better frame rate, respectively, with comparable (90% vs. 93% for [12]) or better accuracy (90% vs. 82.9% for [11]).

To further optimize the models for tinyML applications, we quantized them to their 8-bit precision counterparts using the TensorFlow LITE post-training quantization module, resulting in slight accuracy degradation of 91% for GSC and 89.6% for CIFAR-10. This allowed us to significantly reduce the model sizes, achieving a $14.5\times$ and $101.8\times$ reduction from the baseline implementations with $1.6\times$ and $3.5\times$ better frame rates for GSC and CIFAR-10, respectively. Also, the inference efficiency (FPS/Pwr) increases to $2.5\times$ and $5.9\times$ for the quantized models compared to the baseline implementations for GSC and CIFAR-10, respectively. In addition, the quantized models can now be easily accelerated with FPGA BRAMs or microcontroller SRAMs since they are both under 1.5 MB of memory. It is worth noting that Nvidia Jetson Nano 4 GB SDKs do not support GPU acceleration for INT8 precision, so the quantized models were

processed using the quad-core A57 CPUs. The final results from Table 3 show frame rates of 64.5 and 58.5 FPS for GSC and CIFAR-10, respectively, which are $6.5\times$ better than [10], and $1.8\times$ and $2.2\times$ better than [11] and [12], respectively. We also utilized the Reg-TuneV2 framework to profile the FPGA power and latency metrics of the DNN architecture previously introduced in [6]. This profiling process led to the selection of the configuration $\mathcal{NN}_{(q=4, s=4.5)}$, along with 340 KB of memory. This configuration further resulted in an inference efficiency of 1722 (FPS/Pwr) and an energy efficiency of 104.9 GOPJ (Giga Operations per Joule). In our earlier work, this configuration achieved an accuracy of 90.1%. However, by using accuracy profiles based on metric learning, we were able to improve the accuracy to 90.5%, as shown in Table 3 without making any changes to the parameters.

Conclusion

This work introduces Reg-TuneV2, a method to systematically configure DNN deployment on diverse embedded platforms while considering accuracy, power, and latency. Incorporating metric learning here enhances frame rates ($1.6\times$ and at least $3.5\times$) and reduces model sizes ($14.5\times$ and $101.8\times$) with minimal accuracy loss on Nvidia Jetson Nano for GSC and CIFAR-10 datasets. Notably, in FPGA deployment, Reg-TuneV2 achieves 90.5% accuracy, outperforming prior work, offering hardware-aware insights and integration potential with future NAS frameworks for optimized tinyML vision.

REFERENCES

1. M. Tan *et al.*, "MnasNet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Long Beach, CA, USA: IEEE, 2019, pp. 2820–2828.
2. M. S. Abdelfattah *et al.*, "Codesign-nas: Automatic fpga/cnn codesign using neural architecture search," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 315. [Online]. Available: <https://doi.org/10.1145/3373087.3375334>
3. H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE. Taipei, Taiwan: IEEE, 2016, pp. 1–12.
4. G. Li, S. K. Mandal, U. Y. Ogras, and R. Marculescu, "Flash: Fast neural architecture search with hardware optimization," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–26, 2021.
5. M. Hosseini *et al.*, "A fast method to fine-tune neural networks for the least energy consumption on fpgas," in *Proceedings of the Hardware Aware Efficient Training workshop of ICLR 2021*. Vienna, Austria: ICLR, 2021.
6. A. N. Mazumder and T. Mohsenin, "A fast network exploration strategy to profile low energy consumption for keyword spotting," *CoRR*, vol. abs/2202.02361, 2022. [Online]. Available: <https://arxiv.org/abs/2202.02361>
7. P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018.
8. A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.
9. F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.
10. Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.
11. X. Hu and H. Wen, "Research on model compression for embedded platform through quantization and pruning," in *Journal of Physics: Conference Series*, vol. 2078, no. 1. IOP Publishing, 2021, p. 012047.
12. Y. Fang, S. M. Shalmani, and R. Zheng, "Cachenet: A model caching framework for deep learning inference on the edge," *arXiv preprint arXiv:2007.01793*, 2020.

Arnab Neelim Mazumder is pursuing his Ph.D. degree in the Computer Science and Electrical Engineering Department at the University of Maryland Baltimore County, MD, USA. He received his B.S. degree from CUET, Bangladesh. His research interests focus on deep neural networks, hardware accelerators, explainable AI, and low-power embedded systems.

Tinoosh Mohsenin is an associate professor with the ECE Department at the Institute of Assured Autonomy, Johns Hopkins University, Baltimore, MD, 21218, USA, where she is also the director of the Energy Efficient High-Performance Computing Lab. Her research focuses on designing energy-efficient embedded processors for machine learning and signal processing, knowledge extraction techniques for autonomous systems, wearable smart health monitoring, and embedded Big Data computing.