On-Device Training

Zeyuan Hu
Pierce Lai
Karl Velazquez
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Abstract

The current machine learning paradigm focuses on massive models existing on expensive enterprise hardware. This establishes a high barrier of entry to artificial intelligence development, and also forces users to expose their sensitive data to the organizations operating these models. These concerns have increased the demand for transferring models to run on edge devices; however, their limited resources compared to enterprise machines present many challenges to this. One of the most difficult aspects is model training, as it requires additional computation to perform. Substantial progress has already been made to enable effective on-device training. Using the Tiny Training method proposed by MIT Han Lab, this paper examines the performances of an object detection model trained on a microcontroller over different datasets. We find that the microcontroller can be easily trained for object detection tasks, and that for best results the object to be detected take up a substantial proportion of the image. We additionally find that the optimal number of training steps for the model on our datasets is around 50 to 150, and that accuracy deteriorates after this point. Our video demos for the on-device training can be found at: `https://drive.google.com/drive/folders/1EUDs4pxN-WTBzATFC-VijM9JKKAB0wPd?usp=sharing`.

1 Introduction

The overwhelming majority of large scale machine learning currently centers around grand cloud computing datacenters. However, this arrangement presents several issues. These large scale models are expensive to operate in both training and inference contexts. Users are required to send and expose their data to unknown and therefore untrusted agents. Sending data over the wire also stacks latencies to the operation times of these models.

These issues naturally lead to deploying machine learning on edge devices instead of the cloud. Having models stored on devices like phones or microcontrollers no longer requires data to be transmitted anywhere, as everything stays within the device. This protects user privacy and network latencies no longer contribute to model operating times. However, this transfer to edge devices comes with the challenge of scaling down these models to work in limited resource environments. Most models are far too big to fit in phones or microcontrollers, requiring hundreds of megabytes or even gigabytes of memory. Additionally, training in particular poses its own additional set of problems, as it requires storing and computing many gradients across layers and updating the parameters of weight matrices. The issue then becomes how to transform traditional large scale machine learning models to train and infer on those devices, some of which may have only a megabyte of memory available.

This leads to our project, which investigates the efficacy of current methods to perform model training on device. We utilize the Tiny Training Engine framework proposed by Han Lab [3], and explore its on-device object detection capability against different data and training steps.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

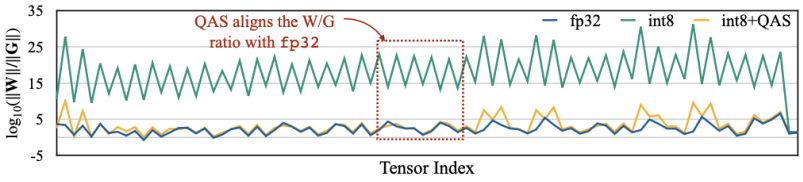

Figure 1: Quantization drastically increases the weight-to-gradient ratio compared to the original model, making it difficult to train. Quantization-aware scaling returns the weight-to-gradient ratio to its original state.

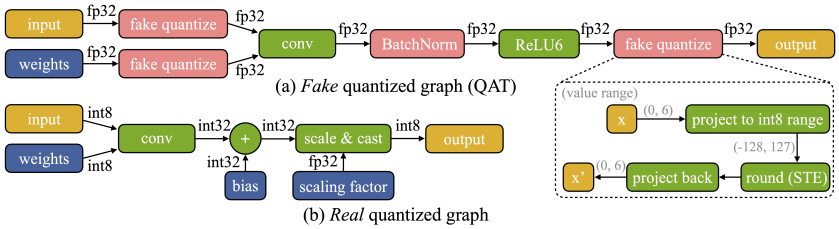

Figure 2: Real vs fake quantization. Source: [4]

2 Methods

2.1 Tiny Training Engine

MIT Han Lab has developed a framework — the Tiny Training Engine — to achieve on device training [4, 2]. In order to adapt model training to machines with severe resource constraints, they leverage two main innovations: Quantization Aware Scaling and Sparse Updating.

Quantization Aware Scaling: To reduce the computational and memory requirements of a model, the user can quantize the model. This means the weights and activations of the neural network are converted from high precision floating point numbers to a lower precision data type, reducing the memory footprint. However, during training, this introduces the problem of computed gradients differing from what they would have been if the parameters had stayed in their original higher precision data type. The quantization distorts the gradient by massively increasing the weight-to-gradient ratio, as seen in Figure 1.

An approach to fix this is to keep the parameters in the floating point type and compute from there (demonstrated by the fake graph in Figure 2); however, this does not actually save any memory and so is not possible on-device. The Tiny Training Engine approach instead introduces a scaling factor that aligns the magnitudes of the gradients with what would have been expected in a high precision data type model (demonstrated by the real graph in Figure 2). This approach stabilizes the training process by reducing the weight-to-gradient ratio, thereby maintaining the convergence properties of the optimization process.

Sparse Updating: Training a neural network involves a forward pass and then performing back propagation through all the layers. Back propagation is an especially expensive operation, as it involves storing many intermediate values and updating numerous parameters. With the limitations of the system, this overhead is impossible to support. This is why instead of tending to the entire network, Sparse Updating is a strategy where the model only updates the layers and subtensors that contribute the most to downstream accuracy. The strategy was inspired by traditional pruning techniques which involve removing less important weights. Sparse Updating also favors bias updates, which do not require storing intermediate activations and therefore make it more favorable memory wise.

For the experimental set-up, we use a STM32F746G-DISCO microcontroller, which has 1 MB of flash memory and 320 KB SRAM. The camera is an Arducam Shield Mini 2MP Plus Camera. The classification model, inference script, and training script are retrieved from [2].

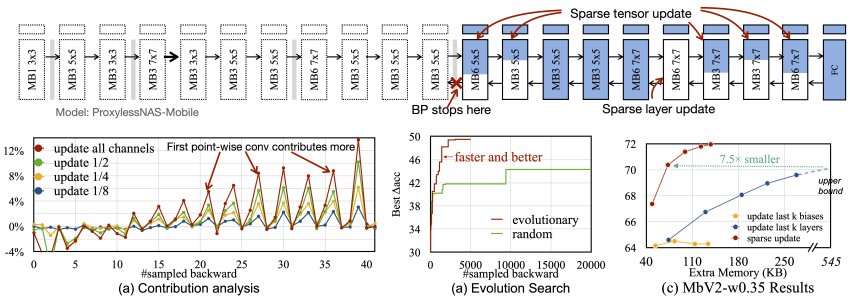

Figure 3: Visualization of sparse updating. Source: [4]

Table 1: Accuracy on the training set.

	Positive	Negative	Total	Accuracy
Cat	31	9	40	78%
No Cat	19	21	40	53%
Total	50	30	80	65%

3 Results

3.1 Cat Detection

We start with a very simple test, which is training the MCU to determine if an image depicts a cat or not. For the training set, we procure 80 images from the internet, 40 of which are pictures of cats and 40 of which do not have cats. The test set is likewise constructed, 15 images with cats and 15 without. Some examples of images are shown in figure 4. We train the model on the training set for one epoch, and then measure the accuracy on the training and test set.

We find an overall accuracy of about 60%, which is slightly better than random guessing, and an F1 score about 0.7. The model tends to confuse other animals with cats, including humans, dogs, and cows, so there are lots of false positives. The results are displayed in tables 1, 2, and 3.

3.2 Sensitivity of Inference Performance

In this section, we consider an real application scenario - person detection. We want to test the dependence of the training effectiveness on different data sets. Specifically, we want to explore two aspects of training data. One is the number of samples in the training data. The other is the data quality.

To test training dependence on training sample, we use the Visual Wake Words (VWW) data set [1]. We sample 200 images from the VWW dataset as our training data and 50 images as our validation images. In both training and validation images, about half of the images contain a person while the rest do not contain any person. We train the model on the training data for 2 epochs and evaluate the model performance at multiple stages - at the beginning, after 50 images, 100 images, 150 images, 200 images, and 2 epochs.

Before the training, the neural net already shows some capability of detecting person when we face the camera to a real person. This is because the model is pretrained. However, the neural nets cannot always detect a person, especially when the face becomes small in the image or is partially covered

Table 2: Accuracy on the test set.

	Positive	Negative	Total	Accuracy
Cat	15	0	15	100%
No Cat	12	3	15	20%
Total	27	3	30	60%

Figure 4: Example of an image featuring with a cat (top) and an image without a cat (bottom).

Table 3: F1, recall, and precision results on the cat detection task.

	Training	Test
Precision	0.62	0.56
Recall	0.78	1.00
F1	0.69	0.71

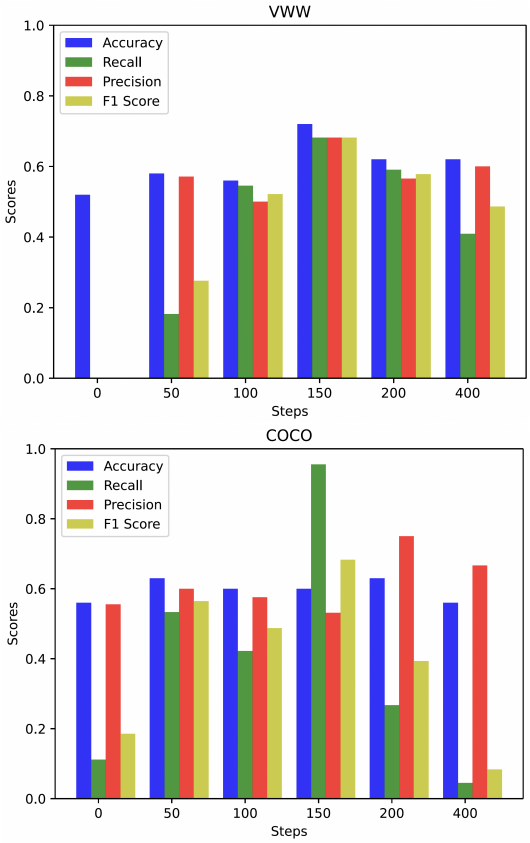

Figure 5: Evaluation performance on the VWW (top) and COCO (bottom) data set on four different metrics: accuracy (blue), recall (green), precision (red), F1 score (yellow). Step 0 to 200 are in the first epoch. Step 400 is at the end of 2 epochs.

by hands. When we evaluate the initial performance on the validation images, the performance is bad. It predicts almost all the images as non-person except for 3 false positives, with both recall and precision as 0 and total accuracy as 0.5.

Through the training, we see clear improvement of evaluation performance (Figure 5). Both the accuracy and F1 score peak at the end of step 150. The peak accuracy reaches at 0.72. However, after step 150, further training / one additional epoch does not improve the accuracy and F1 score. This might be related to a few things. One possibility is just random fluctuation of the performance due to the stochastic gradient descent. In that case, finetuning the learning rate, learning rate scheduler, and/or regularization might help. Another reason could be due to data quality.

We next test the influence of data quality on the training performance. One motivation is that, in situations with limited training data size on edge device, if there is a person in an image but occupying very small area, should we include such images labeled as person in the training data? To test it, we constructed another training data set from the COCO dataset [5]. COCO dataset is a super set of the VWW dataset. The difference is that VWW filters images by a threshold for the minimum percentage (0.5%) that the bounding box of a person must occupy in an image. COCO dataset thus could contain some images with a tiny person in the image. We use the COCO dataset to construct a training data of 200 images and a validation data of 50 images. The training and evaluation procedure is the same as using the VWW dataset, in which we perform evaluation at step 0, 50, 100, 200, and 400.

Using the COCO dataset, the peak evaluation accuracy drops to 0.63. The accuracy reaches the peak at step 50 and does not further improve. The F1 score peaks at step 150. Note that the recall shows large variation - close to 1 at step 150 and close to 0 at step 400, suggesting that the model is easy to switch from overconfident to overcautious. One observation through the training is that such overconfidence/overcautiousness tends to appear when the model sees a long sequence of only

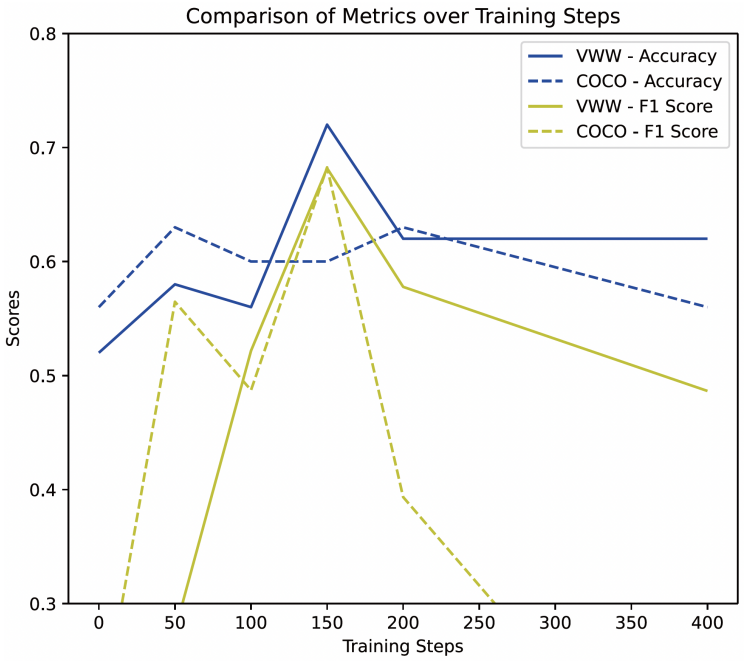

Figure 6: Comparison of evaluation metrics between using the VWW dataset (solid lines) and using the COCO dataset (dashed lines) as the training data against different sample sizes. The blue lines are for the accuracy. The yellow lines are for the F1 score.

positive training images or only negative training images. Figure 6 compares the accuracy and F1 score between using VWW and COCO data sets. Using VWW dataset leads to greater accuracy increase through training when compared to the initial performance. While this may not statistically prove that VWW dataset is indeed better than COCO dataset for the person detection task for edge device training, it raises a question for users whether they should input images with tiny people in it. Future work could involve more systematically examining the training performance by constructing a series of VWW datasets with different minimum thresholds for person bounding box.

4 Conclusion

On-device training is a technique that allows models to be stored, trained, and deployed completely on a user's device, reducing latency and protecting user data. Using Han Lab's Tiny Training Engine, we are able to train a microcontroller to perform a simple object detection task. We find accuracy peaks at around 150 training steps, and deteriorates with more training. We additionally conclude that the training data provided has a potential impact on the accuracy of the trained model; for best results, the object to detect might need to take up a substantial portion of the image.

Acknowledgments and Disclosure of Funding

We thank MIT Han Lab for providing the code and project idea and lending their equipment for this project.

References

[1] Aakanksha Chowdhery et al. *Visual Wake Words Dataset*. 2019. arXiv: 1906.05721 [cs.CV].

[2] MIT Han Lab. *tinyengine*. URL: https://github.com/mit-han-lab/tinyengine.

[3] Ji Lin et al. "On-Device Training Under 256KB Memory". In: (2022).

[4] Ji Lin et al. *On-Device Training Under 256KB Memory*. URL: https://tinyml.mit.edu/on-device-training/.

[5] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.