On-Device Augmented Image Training and Inference utilizing the TinyEngine Library on a STM32F746G-DISCO Microcontroller

Andy Zhao
Operations Research Center
MIT
`andyzhao@mit.edu`

Javi Ocampo
EECS
MIT
`javio@mit.edu`

Abstract

This paper presents Tiny Engine, an optimization-driven inference library tailored for microcontrollers, which facilitates on-device training and inference in resource-constrained environments. We demonstrate the effectiveness of Tiny Engine through a series of benchmarks, highlighting significant improvements in computational efficiency and cost-effectiveness compared to traditional cloud-based and mobile AI platforms by training and fine-tuning an augmented image data set live on a microcontroller with a camera. Our approach leverages a combination of in-place depth-wise convolution, patch-based inference, and advanced data augmentation techniques, which collectively enable the execution of sophisticated neural network models on edge devices. We validate our model using a dataset augmented through transformations, showcasing the potential for broader application in real-world scenarios.

1 Introduction

The contemporary landscape of artificial intelligence is witnessing a transformative shift towards edge computing, a movement driven by the increasing ubiquity of Internet of Things (IoT) devices and the pressing need for real-time, low-latency processing. Edge computing brings data analysis and knowledge generation to the location where it is needed, on the device itself. This shift is particularly relevant in the domain of computer vision, where the ability to locally process visual data promises enhancements in speed, reliability, and privacy. Autonomous vehicles, for instance, benefit from instantaneous decision-making capabilities, unencumbered by the latency inherent in cloud-based computing. Similarly, in-home personal assistants that utilize computer vision can process sensitive data on the device, alleviating privacy concerns by minimizing data transmission to remote servers.

The requirement for on-device training is underscored not only by privacy and efficiency but also by the practical constraints of connectivity. Devices operating in remote or highly secure environments may not have the luxury of cloud access, necessitating local data processing capabilities. This is where Tiny Engine's role becomes indispensable. Tiny Engine facilitates the deployment of machine learning models on devices with stringent memory and compute limitations by leveraging several optimization techniques, each contributing to its ability to perform complex tasks in constrained environments.

The deployment of computer vision models conventionally necessitates substantial computational resources, often entailing high costs related to powerful GPUs, expansive storage, and substantial energy consumption for both training and inference processes. Such requirements render the technology less accessible for small-scale applications and present challenges in terms of scalability and sustainability. Furthermore, the reliance on centralized data processing incurs additional costs for data transmission and cloud computing services, which can be prohibitive for continuous real-time analysis.

In contrast, Tiny Engine's approach represents a paradigm shift towards cost-effective and localized machine learning solutions. By enabling the execution of vision models on edge devices, which are significantly less costly than their cloud-based counterparts, Tiny Engine reduces the financial barriers to entry. This democratization of AI allows for a broader range of applications, from consumer electronics to remote sensing devices, all while minimizing energy demands and operational costs. The local processing eliminates the need for continuous cloud connectivity, reducing bandwidth usage and associated costs.

Additionally, Tiny Engine's optimization strategies significantly lower the memory and compute overheads typically associated with deep learning models. This not only alleviates the need for expensive hardware but also opens up the potential for battery-operated devices to perform complex vision tasks, a novelty in the realm of ubiquitous computing. The resultant cost savings, coupled with the increased privacy and reduced latency of on-device processing, make Tiny Engine an attractive proposition for contexts where rapid, on-site decision-making is paramount, and resources are constrained. Use cases such as in-field agricultural monitoring, privacy-sensitive surveillance, and consumer-grade smart cameras exemplify scenarios where the balance of performance and cost offered by Tiny Engine is not just beneficial but perhaps essential.

The efficiencies achieved by Tiny Engine are made possible through a series of advanced optimization techniques. These strategies serve as the cornerstone of Tiny Engine's ability to execute deep learning models on edge devices efficiently. By rethinking conventional computational methods, Tiny Engine not only brings machine learning models to the edge but does so with remarkable cost-efficiency and performance. As we pivot to examine the individual optimizations, we shall see how each method contributes to reducing the computational footprint, allowing sophisticated vision tasks to be performed directly on-device, even in the most resource-constrained environments.

In-Place Depth-Wise Convolution: Depth-wise convolution is an efficient variant of the traditional convolution operation, where filters are applied separately to each input channel, reducing computational complexity. Tiny Engine takes this a step further with in-place depth-wise convolution, where the output of a filter operation overwrites the input data once it is no longer needed. This approach not only conserves memory but also reduces the need for data movement between different memory areas, which is both time-consuming and power-intensive. Such in-situ processing is crucial for edge devices, where memory is not a plentiful resource.

Patch-Based Inference: Traditional convolutional operations process entire images or feature maps, which can be memory-intensive. Tiny Engine employs patch-based inference, where an image is divided into smaller, manageable patches. Each patch is processed sequentially, significantly reducing the peak memory requirement. This localized processing allows edge devices to handle large models like MobileNetV2, which would typically exceed their memory capabilities.

Operator Fusion: Operator fusion is a technique that combines multiple operations into a single, compound operation. By fusing operations, Tiny Engine minimizes the intermediate memory footprint and the overhead of loading and storing temporary data. This is particularly beneficial when executing sequences of operations that involve small, reusable data, as it allows for a reduction in the necessary memory traffic—a common bottleneck in performance.

SIMD Programming: Single Instruction, Multiple Data (SIMD) programming is a parallel processing strategy that allows the same operation to be performed on multiple data points simultaneously. Tiny Engine's use of SIMD exploits the data-level parallelism of edge devices' CPUs, accelerating the execution of the same operation across different elements of a data set, such as pixels in an image or neurons in a network layer.

HWC to CHW Weight Format Transformation: Convolutional neural networks commonly use tensors in Height x Width x Channels (HWC) format. However, for optimized memory access patterns, particularly on CPUs that favor contiguous data access, Tiny Engine transforms these tensors into Channel x Height x Width (CHW) format. This reordering aligns the data to be more cache-friendly, improving cache hit rates and reducing costly memory access delays.

Im2Col Convolution: The Im2Col (Image to Column) technique reshapes the input data into columns before performing matrix multiplication, a fundamental operation in convolutional layers. This restructuring allows for the use of highly optimized General Matrix Multiply (GEMM) routines, which are well-tuned for performance on various architectures, thereby accelerating convolution operations.

Loop Optimizations (Reordering, Unrolling, Tiling): Loop optimizations are critical in maximizing performance. Loop reordering adjusts the order of loop iterations to optimize memory access patterns, loop unrolling increases the loop's granularity, reducing the loop overhead and enabling further parallelism, and loop tiling breaks down loops into smaller blocks, allowing for better cache utilization. Together, these optimizations mitigate the impact of slow memory operations on computation speed.

Each of these techniques, holistically integrated into Tiny Engine, represents a significant stride towards realizing the full potential of on-device training and inference. They collectively enable a new class of applications where edge devices can independently learn from their environment, becoming smarter and more context-aware without the shackles of cloud dependence.

2 Methodology

2.1 Set Up

The practical implementation of Tiny Engine for on-device training and inference is contingent upon the assembly of hardware components. Within our setup is the microcontroller, an STM32F746G-DISCO board. This board interfaces with an Arducam Shield Mini 2MP Plus, which serves as the vision sensor, capturing real-time visual data for model training and inference. The connectivity between these components is facilitated by a set of male-to-female jumper wires, ensuring the flow of data and power across the system.

2.2 Data Methods

The methodology employed in this study involves a process of dataset compilation, image generation, and pre-processing to prepare a robust dataset for binary classification.

2.3 Data Compilation

Professor Han's Dataset: The images representing one class in the binary classification task were specifically sourced from Google Images by searching for "Professor Song Han" affiliated with notable institutions like MIT, Stanford, and Tsinghua. These images ensure the inclusion of a diverse range of poses and environments, representing a realistic scenario in facial recognition tasks. LFW Funneled Dataset: The other class of images was sourced from the LFW (Labeled Faces in the Wild) Funneled Dataset. This dataset is an extension of the original LFW dataset, where the images have been aligned using commercial face alignment software. This alignment process standardizes the images, making them more suitable for consistent facial recognition analysis.

2.4 Image Generation

Given the limited number of images available for Professor Han, data augmentation was essential to enhance the dataset's size and variability.

To augment the dataset, TensorFlow's `ImageDataGenerator` was employed. This tool significantly enriches the dataset by applying various transformations to the existing images, thus simulating different conditions and angles. The specified parameters for augmentation included:

- `rotation_range=40`: Rotates the images by up to 40 degrees, introducing rotational variability.
- `width_shift_range=0.2` and `height_shift_range=0.2`: Translates the images up to 20% in both horizontal and vertical directions, simulating off-center positioning of faces.
- `shear_range=0.2`: Applies shear transformations, mimicking the effect of different camera angles.

- `zoom_range=0.2`: Allows for random zooming, representing different distances from the camera.
- `horizontal_flip=True`: Adds horizontal flips, useful for simulating different orientations.
- `fill_mode='nearest'`: Ensures that any new pixels created as a result of augmentation are filled in a way that maintains the image's integrity.

2.5 Dataset Balancing and Test Set Preparation

Balancing the Dataset: To address potential class imbalances, 25 images were randomly selected from the LFW Funneled Dataset and combined with 25 augmented images of Professor Han. This resulted in a balanced training dataset comprising 50 images. Test Set Creation: For testing purposes, 5 random images from the LFW Funneled Dataset were chosen, supplemented with 5 specific images of Professor Han. This led to a total of 10 images in the test set, ensuring both diversity and relevance in the evaluation phase.

2.6 Image Pre-Processing Mechanism

Image Capture: The process begins with capturing raw image data, typically executed through direct interaction with camera hardware in an embedded system environment. Decoding and Processing: The captured raw data, usually in a compressed format like JPEG, is decoded into an array of pixel values, converting it into a usable format for further processing. Normalization: An essential step where pixel values are scaled to a range suitable for the neural network, aiding in faster convergence and better performance during training. Resizing: To ensure uniformity, each image is resized to match the neural network's required input dimensions. Format Conversion: Finally, the images are converted into a format compatible with the neural network. This may include flattening the image data or adjusting color channel orders.

3 Neural Network in Embedded Systems

3.1 Neural Network Training Mechanism in Embedded Deep Learning Systems for Image Classification

The training mechanism of a neural network within the context of embedded systems, particularly for image classification applications, involves a series of specialized steps. These steps are tailored to deal with the unique challenges and limitations of embedded environments, such as constrained computational resources and memory limitations.

Forward Pass: The initial stage of neural network training commences with a forward pass, processing the input data, such as images, in a sequential manner. This involves the application of various network layers, including convolutional layers, activation functions like ReLU, and pooling layers that collectively contribute to the reduction of data dimensionality. The culmination of this forward pass is the generation of a prediction by the network, which represents an estimate of the image's label based on the current model weights.

Loss Calculation: The essence of neural network training hinges on evaluating the accuracy of the network's predictions, a process implemented through the use of a loss function, like cross-entropy loss, which measures the disparity between the network's prediction and the actual label. For effectiveness in embedded systems, where processing power is often limited, the selection of a loss function is crucial, necessitating a choice that balances computational efficiency with precise performance assessment.

Backpropagation: Backpropagation serves a vital function in neural network training, where it is used to ascertain the contribution of each weight to the total error. This process entails calculating the gradient of the loss function concerning each weight, effectively applying the chain rule of calculus in reverse. For its effective adaptation in embedded systems, which often face memory constraints, optimizations such as reduced precision arithmetic are crucial to ensure that the backpropagation algorithm operates within the limited resources available.

Weight Update: Following backpropagation, the neural network undergoes a critical phase where its weights are updated. This is typically achieved through optimization algorithms like Stochastic Gradient Descent (SGD) or the Adam optimizer, which incrementally adjust the weights to minimize the loss. In the context of embedded systems, where computational capacity is limited, it's imperative that this weight update process is resource-efficient, ensuring that the adjustments are made within the constraints of the available hardware resources.

Iteration and Convergence: The neural network training process involves repeating multiple iterations, or epochs, across the entire dataset, a repetitive nature that is crucial for fine-tuning the network. Each iteration aims to minimize error, thereby enhancing the accuracy of the network's predictions. In the context of embedded systems, where processing capabilities are inherently limited, the number of epochs is carefully balanced to optimize the training process within these constraints, ensuring effective learning without overtaxing the system's resources.

3.2 Inference Mechanism in Embedded Deep Learning Applications

In the realm of embedded deep learning applications, inference plays a crucial role following the completion of the neural network training phase. This stage is pivotal for making predictions on new, unseen data, leveraging the learned patterns and relationships embedded within the trained model.

Image Capture and Pre-processing: In the neural network inference process, the initial step involves decoding the raw image data, which is often in compressed formats, into a usable array of pixel values. This is followed by normalization, where pixel values are scaled to a range compatible with the neural network's requirements, a critical step for ensuring consistent data input and significantly impacting the network's performance. The image is then resized to match the input size expected by the neural network, ensuring uniformity in data shape and size. Finally, the image undergoes format conversion, adapting it into a format suitable for the network, which may include flattening the image into a one-dimensional array or reorganizing the color channels.

Loading the Trained Model: In the inference stage, the application leverages the neural network model that has been trained on relevant data. This step involves loading the model, complete with its learned weights, into the device's memory. Given the often-limited resources of embedded systems, particularly in terms of memory, this process must be meticulously optimized for efficiency.

Forward Pass (Prediction): During the inference phase of a neural network, the input image is processed through various layers of the network, such as convolutional, activation, and pooling layers, using the weights that were refined during the training process. This forward pass through the network's layers ultimately results in a prediction. The nature of this prediction varies depending on the specific application; it could be a specific class label in classification tasks or continuous outputs in regression models.

Post-processing of Output: The output from the neural network, often raw in its initial form, may require post-processing to become interpretable and useful. This might involve selecting the highest probability class as the final prediction in classification tasks, or tailoring the output to user-friendly formats or integrating it with other application-specific datasets.

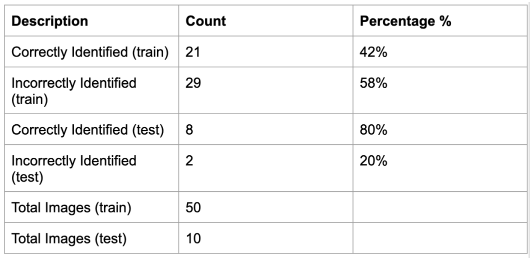
Feedback and Display: The derived prediction is typically rendered on an output interface, such as an LCD screen, or utilized within the application's decision-making workflow. Interactive applications might also include user feedback mechanisms based on these predictions.

Continuous Operation in Real-Time Applications: In real-time scenarios, the inference process is often in a state of continual operation, processing new images as they are captured and offering updated predictions based on the latest data. This ongoing operation is vital in applications where timely decision-making and responsiveness are key.

4 Report on Image Classification Model Performance and Future Improvements

4.1 Training and Testing Accuracy Analysis

The observed disparity between testing and training accuracy in our image classification model presents a notable case. Initially, the model's learning process is slow, marked by a high incidence of



Description	Count	Percentage %
Correctly Identified (train)	21	42%
Incorrectly Identified (train)	29	58%
Correctly Identified (test)	8	80%
Incorrectly Identified (test)	2	20%
Total Images (train)	50	
Total Images (test)	10	

Figure 1: Results

mistakes that diminish training accuracy. This is a natural part of the model's adjustment to the data, where it learns to generalize and refine its parameters. However, these early errors are crucial for learning, despite temporarily lowering accuracy.

The testing phase, conducted with only 10 data points, predominantly showed false negatives. This indicates the model's proficiency in identifying non-professors while struggling to accurately recognize professors. The small size of the test dataset likely contributes to an inflated accuracy due to the high variance from limited data. To enhance the reliability of testing outcomes, expanding the test dataset is essential. A larger and more diverse dataset would offer a truer reflection of the model's real-world performance and reduce variance-induced inaccuracies.

4.2 Model Learning and Bias Mitigation

Throughout the training phase, there was a notable decrease in false negatives over time, suggesting an increasing proficiency in identifying specific subjects, such as Professor Han. This improvement could stem from the model's gradual learning of distinct features and fine-tuning of its recognition algorithms. Further reduction in false negatives could be achieved by integrating more diverse data and refining model parameters.

However, a significant number of images were misclassified when subject to large rotational angles, indicating that the augmented dataset introduces a bias in classification. To counter this, implementing more sophisticated data augmentation strategies, encompassing varied angles, lighting, and backgrounds, will be crucial. This approach aims to bolster the model's generalization capabilities and reduce sensitivity to specific conditions.

4.3 Background Standardization and Real-World Testing

Background variations play a critical role in image classification accuracy. Standardizing the background in preprocessing could significantly enhance the model's focus on the subject and improve classification accuracy. Employing background subtraction techniques or subject-focused deep learning methods would further this goal.

Real-world testing opportunities have been limited, constraining our ability to assess the model's effectiveness in practical scenarios. Addressing this, synthetic data generation and controlled real-world experiments, possibly in collaboration with educational institutions, could provide more comprehensive validation and insights into practical performance.

4.4 Future Work and Learnings

4.4.1 Dataset Expansion and Advanced Augmentation

Our next steps involve substantially enlarging the dataset to improve model accuracy and learning depth. By embracing a wider array of data, the model can better recognize and generalize from varied patterns and features. Further exploration into advanced data augmentation techniques, including environmental simulations, will aim to enhance the model's resilience and adaptability to different imaging conditions.

4.4.2 Integration with Tiny Engine and Practical Applications

Integrating the Tiny Engine on microcontrollers with screens will open interactive avenues. Real-time feedback via screens could make the system more transparent and user-involved, particularly in retraining loops. Identifying specific use cases where on-device processing is most beneficial, such as immediate-action-required field devices or privacy-sensitive consumer devices, will be a focal point.

4.4.3 User Interface Enhancement

Improving the user interface on the microcontroller is pivotal. A sophisticated interface will not only provide performance insights but also build user trust and understanding of the model's decision-making process. This enhanced interface could also serve as a diagnostic tool, feeding back into model improvement strategies.

4.4.4 Enhancing Model Robustness and Generalization

To further improve the model's performance, we plan to focus on enhancing its robustness and ability to generalize across various scenarios. This involves not only expanding the diversity of the dataset but also refining the model's architecture and training methodologies. Techniques such as transfer learning, where the model is pre-trained on a large, diverse dataset and then fine-tuned with specific data, could be instrumental. Additionally, exploring different neural network architectures might yield better results in terms of accuracy and efficiency.

4.4.5 Ethical Considerations and Bias Minimization

An integral part of our future work will also involve addressing ethical considerations, particularly around bias minimization. Ensuring that the model does not inadvertently learn and perpetuate biases present in the training data is crucial. This will involve a thorough analysis of the data for potential biases and implementing strategies like balanced dataset creation and algorithmic fairness checks.

4.4.6 Continuous Monitoring and Feedback Loop

Establishing a continuous monitoring system for the model's performance in real-world scenarios will be key. This system would enable us to gather feedback on the model's effectiveness and areas for improvement. Integrating a feedback loop, where real-world performance data is used to continuously train and refine the model, will ensure that it remains relevant and accurate over time.

5 Conclusion

In conclusion, this study on image classification using Tiny Engine highlights key learnings and future directions. The disparity between training and testing accuracies underlines the importance of a robust dataset. Initially, the model struggled with accuracy, especially in recognizing specific subjects, which improved over time, indicating effective learning. However, the testing phase's limited dataset size points to the need for a larger, more diverse dataset to reduce variances and enhance real-world applicability. The model's performance under various conditions, particularly with images at large rotational angles, suggests a need for advanced data augmentation strategies. These strategies would enable the model to generalize better across different conditions, thereby reducing biases and improving accuracy. Standardizing backgrounds and employing focused deep learning methods could further enhance performance.

Future work involves expanding the dataset and exploring sophisticated augmentation techniques, including environmental simulations. Integrating Tiny Engine with microcontrollers and enhancing the user interface opens new avenues for interactive applications. Emphasis on model robustness, generalization, and ethical considerations, particularly bias minimization, is essential. Continuous monitoring and a feedback loop will ensure the model's relevance and effectiveness in real-world scenarios, balancing technical proficiency with practical utility. This holistic approach aims to not just advance the technical capabilities of the model but also address the ethical and practical challenges in deploying AI in diverse environments.

6 Acknowledgement

We express our profound appreciation to everyone who played a pivotal role in the successful completion of this project. Foremost, we extend our sincere thanks to our course instructor, Professor Song Han, and our teaching assistants, Han Cai, Ji Jin. Specical appreciation to Wei-Chen Wang for answering our questions. Their invaluable guidance, insightful feedback, and steadfast support have been instrumental throughout this project. Their expertise significantly influenced the direction and execution of our study. We also acknowledge the influential paper, "On-Device Training Under 256KB Memory" which served as a significant source of inspiration and guidance. Additionally, we utilized and adapted code from the GitHub repository mit-han-lab/tiny-training, which greatly facilitated our project's development. Furthermore, we are grateful for the assistance provided by ChatGPT in refining the wording and debugging.