Training on the Fly: On-device Self-supervised Learning aboard Nano-drones within 20 mW

Elia Cereda, *Student Member, IEEE,* Alessandro Giusti, *Member, IEEE,* and Daniele Palossi

Abstract—Miniaturized cyber-physical systems (CPSes) powered by tiny machine learning (TinyML), such as nano-drones, are becoming an increasingly attractive technology. Their small form factor (i.e., , ∼10 cm diameter) ensures vast applicability, ranging from the exploration of narrow disaster scenarios to safe human-robot interaction. Simple electronics make these CPSes inexpensive, but strongly limit the computational, memory, and sensing resources available on board. In real-world applications, these limitations are further exacerbated by domain shift. This fundamental machine learning problem implies that model perception performance drops when moving from the training domain to a different deployment one. To cope with and mitigate this general problem, we present a novel on-device fine-tuning approach that relies only on the limited ultra-low power resources available aboard nano-drones. Then, to overcome the lack of ground-truth training labels aboard our CPS, we also employ a self-supervised method based on ego-motion consistency. Albeit our work builds on top of a specific real-world vision-based human pose estimation task, it is widely applicable for many embedded TinyML use cases. Our 512-image on-device training procedure is fully deployed aboard an ultra-low power GWT GAP9 System-on-Chip and requires only 1 MB of memory while consuming as low as 19 mW or running in just 510 ms (at 38 mW). Finally, we demonstrate the benefits of our on-device learning approach by field-testing our closed-loop CPS, showing a reduction in horizontal position error of up to 26% vs. a non-fine-tuned state-of-the-art baseline. In the most challenging never-seen-before environment, our on-device learning procedure makes the difference between succeeding or failing the mission.

Index Terms—On-device Learning, Self-supervised Learning, Embedded ML, TinyML, Resource-constrained CPS.

SUPPLEMENTARY VIDEO MATERIAL

Supplementary video material of the in-field experiments at https://youtu.be/3yNbMwszpSY

I. INTRODUCTION

MINIATURIZED unmanned aerial vehicles (UAVs) as small as the palm of one hand, also known as nano-UAVs, are appealing cyber-physical systems (CPS), which, leveraging tiny machine learning (TinyML) algorithms deployed aboard, have reached an unprecedented level of autonomy. Thanks to their small form factor, i.e., 10 cm in diameter and sub-50 g in weight, nano-UAVs can fly in narrow and constrained spaces [1] or safely in the human proximity [2]

embodying the ultimate dynamic IoT smart sensors, capable of analyzing their surrounding and flying where their presence is most needed. To cope with extremely limited onboard sensory, memory, and computational resources, i.e., , low-resolution cameras, a few MB off-chip memories, and sub-100 mW power envelope for the computational units, recent works make extensive use of optimized TinyML workloads, such as convolutional neural networks (CNNs) [2], [3], [4].

Apart from the well-known challenges in deploying complex deep learning (DL) models on ultra-low-power microcontroller units (MCUs), TinyML algorithms are vulnerable to the fundamental problem of *domain shift* [5]. Domain shift occurs when a DL model, such as a classifier or regressor, trained on data acquired in a given context, i.e., *domain*, is deployed in a different one. Then, the predictive performance of the system (i.e., the accuracy or estimation error) often decreases because the training data is not representative of the deployment domain. This problem is particularly present in real-world applications, such as the robotic task we address in this work. **This paper explores on-device learning as a viable and effective solution to the domain shift problem.**

We consider the case in which a CNN, previously trained on a given task using large datasets, is *fine-tuned* on-device, using a small amount of data collected after deployment. Fine-tuning consists of updating the parameters of the pre-trained models by executing a limited number of additional training steps. With on-device self-supervised learning, data can be collected by the robot precisely in the domain in which the model will operate, counteracting the domain shift. Despite on-device learning being conceptually simple and attractive, pulling it off in a real-world application is extremely challenging. Fine-tuning typically requires significant memory and computational resources, including hardware support for floating-point arithmetic (not always available on MCUs) and data availability with the corresponding ground-truth labels.

In this work, we study this problem in the context of a real-world robotic application, covering the full pipeline of design, deployment, and testing of a vertically integrated on-device learning system and evaluating the practical implications of our design choices. In particular, we strive to answer three key research questions: *i)* What is the best fine-tuning strategy to exploit the limited computing power? *iii)* Can the limited onboard memory fit enough data for fine-tuning to be effective? *ii)* How to deal with the lack of ground-truth data in the field? While analysed in a specific context, these questions yield valuable insights for real-world applications of on-device learning also on other tasks.

We start from the State-of-the-Art (SotA) PULP-Frontnet CNN [2] for the human pose estimation task, whose outputs

TABLE I
EMBEDDED ON-DEVICE LEARNING LITERATURE REVIEW. WE REPORT THE LOWEST POWER COMPUTATIONAL DEVICE CONSIDERED IN EACH WORK.

Work	Task	Arch.	Data	Training	Supervision	Compute	Memory [MB]	Power [W]	Field-tested
Mudrakarta *et al.* [6]	object class. + object detect.	CNN	benchmark datasets	fine-tuning	supervised	GPU (unspecified)	1000s	100s	✗
TinyTL [7]	object classification	CNN	benchmark datasets	fine-tuning	supervised	GPU (unspecified)	1000s	100s	✗
PULP-TrainLib [8]	object class. + keyword spotting	CNN	benchmark datasets	from scratch	supervised	GWT GAP9	0.4	0.1	✗
RLtools [9]	robot control	fully connected	external simulator	from scratch	reinforcement	i.MX RT1060	16	0.3	✗
TinyTrain [10]	object classification	CNN	benchmark datasets	fine-tuning	supervised	Raspberry Pi Zero 2	512	10	✗
PockEngine [11]	object classification	CNN + Transformer	benchmark datasets	fine-tuning	supervised	STM32F7	0.3	0.2	✗
LifeLearner [12]	object classification	CNN	benchmark datasets	continual learning	supervised	STM32H7	1	1	✗
Ours	**human pose estimation**	**CNN**	**real-world robot**	**fine-tuning**	**self-supervised**	**GWT GAP9**	**1**	**0.1**	✓

control the nano-UAV and allow to keep it in front of the user at a predefined distance. We tackle the first computational challenges by adopting an ultra-low-power GWT GAP8 System-on-Chip (SoC) and its next generation, the GAP9. Then, we investigate four fine-tuning strategies, spanning from the most memory-hungry fine-tuning of all layers of the CNN down to minimal fine-tuning of only the final fully connected layer. Further, we propose a self-supervised state-consistency loss term [13] to handle the data availability problem, relieving our fine-tuning process from ground-truth labels. Our **main contributions** are the following:

- an in-depth analysis of four fine-tuning strategies, which explores the trade-off between computational and memory requirements and their prediction performance in a real-world scenario. Our results show a regression performance improvement up to 30% of our self-supervised method against a non-fine-tuned baseline, which grows to 56% when using ground-truth labels.
- On-device implementation and profiling of the resulting best fine-tuning approach, requiring only 6.6 s@102 mW and 511 ms@38 mW for fine-tuning on 512 images (5 epochs) on the GAP8 and GAP9 SoC, respectively.
- An in-field evaluation in a very challenging deployment field where the SotA PULP-Frontnet baseline fails in following the user, while all self-supervised fine-tuned models can complete between 92 and 100% of the expected path (over 3 runs for each model).
- A quantitative evaluation of the nano-UAV position error resulting from an autonomous closed-loop controller that uses the fine-tuned model for perception. Compared to a pre-trained perception model, it reduces by 26% the horizontal position error.

Our work marks the first real-world demonstration of on-device learning aboard a nano-UAV, addressing the critical domain shift problem and paving the way for more general advancements in the scientific community.

II. RELATED WORK

Domain shift [5] is a significant challenge for machine learning approaches in every field: models that perform well on their training domain often underperform when deployed because real-world conditions differ or even change over time. Robotics further exacerbates the issue due to a scarcity of training data, which compounds on hardware-constrained platforms such as nano-UAVs, with the tiny DNN architectures affordable aboard. Past efforts address this issue by better-taking advantage of the limited real-world training data [14], generating vast training datasets in simulation [15], and taking advantage of additional sensors that are more robust to sim-to-real domain shift (e.g., depth sensors) [16].

On-device learning is an emerging field that proposes a radically different approach [17], [18], [19]: abandon the traditional *train-once, deploy-everywhere* paradigm by enabling devices to adapt a model to their domain directly in the field. Table I gives an overview of the most significant on-device learning literature in chronological order. However, applying on-device learning to nano-UAVs is still hurdled by the limitations of the onboard sensors and computationally-constrained processors. Compared to larger-scale drones [20], nano-UAVs can only afford MCU devices with 1/1000th the memory and computational power of their bigger counterparts. Embodiments of the Parallel Ultra Low Power (PULP) platform [21], in particular, have enabled a number of breakthrough applications on nano-UAVs [3], [4]. Regarding on-device learning on PULP devices, *PULP-TrainLib* [8] was recently proposed to accelerate neural network back-propagation through parallelization, software optimizations, and reduced 16-bit floating-point precision.

A number of approaches have been also demonstrated on other embedded platforms. For example, gradient rescaling, sparse updates, and compile-time graph optimization allow *PockEngine* [11] to achieve 8-bit approximated backward passes on STM32 MCUs. *RLtools* [9] introduce a highly
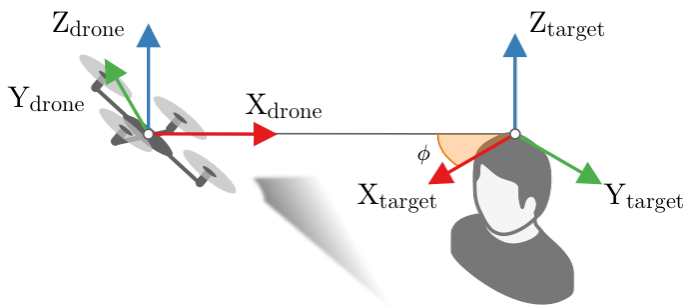
Fig. 1. Human pose estimation task and reference frames [14].

Fig. 2. Robot platform: Crazyflie 2.1 with AI-deck and Flow-deck boards.

portable on-device reinforcement learning implementation and test it, among others, on an i.MX RT1060 MCU. Despite being tailored for robotics applications, RLtools focus on fully connected models and control tasks, while our focus is on more computationally intensive perception tasks with CNN-based methods.

Various approaches propose to reduce the re-training workload through *spare updates*, i.e., by fine-tuning only a subset of the model parameters [6], [7], [10], [12]. Mudrakarta *et al.* [6] limits updates to just the batch normalization layers, which TinyTL [7] further reduces to only the bias parameters. The latter translates to significant memory savings (up to $6.5\times$) by not storing activation maps during the forward pass since they are needed only for back-propagation to weight parameters, not biases. TinyTrain [10] introduces a dynamic strategy based on task-adaptive sparse updates, which achieves a 5.0% accuracy gain on image classification benchmarks and reduces the backward-pass memory and computation by up to $2.3\times$ and $7.7\times$, respectively. LifeLearner [12] applies lossy compression to the stored activation maps to reduce their size in memory by at least $11.4\times$. Crucially, these methods are tested only on image classification tasks and assume that a training dataset with ground-truth labels is provided externally. Further, not all methods are deployed and evaluated on the target embedded devices. On the contrary, our approach extends these techniques [6], [7] to a regression task, and deploys them on-device and in the real world, where ground-truth labels are not readily available. Addressing these fundamental problems has a far-reaching impact on many real-life perception problems.

Exploiting self-collected data and deriving supervisory information exclusively from onboard sensors (as opposed to external infrastructure, manual annotation, or user intervention) is the core idea of self-supervised learning. Three categories of approaches have been recently applied in autonomous robotics. Most approaches derive noisy approximations to the ground-truth labels using task-specific methods, such as exploring an environment with a drone until it crashes [22] or while continuously measuring its distance from the surrounding environment [23]. Others learn a secondary task, for which ground-truth labels are known to the robot, as a pretext for learning the task of interest, for which no or little data is available. Namely, predicting a quadrotor's sound from camera images is a strong proxy for estimating its location [24]. Additionally, a masked autoencoder for image reconstruction can be adapted
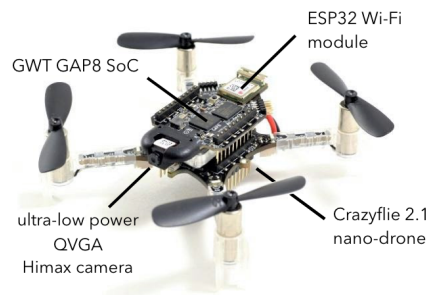
to perform a range of robot manipulation tasks [25]. Finally, audio, optical flow, and depth estimation are useful pretexts for visual odometry [26]. The third group imposes geometric consistency constraints to improve model predictions: an object pose estimation model must be consistent over time with the robot's ego motion [27], while visual odometry between pairs of camera images must satisfy the transitive property [28]. The photometric reprojection error is another common constraint used to train self-supervised monocular depth [29], optical flow [30], and visual odometry [31], [32] models.

Our work proposes a self-supervised fine-tuning process based on an ego-motion consistency loss and approximated labels derived through task-specific collaboration with the user. However, in contrast to the above approaches, which assume abundant self-supervised data, we are tightly constrained by the amount of fine-tuning data we can store on our embedded system's 8MB DRAM memory. Additionally, previous consistency approaches assume self-supervised labels that are noisy but fully measurable (e.g., learning object pose estimation in the presence of odometry error while the object remains still [27]). By comparison, we also deal with inherently unknown data, such as the movements of human subjects.

III. BACKGROUND

A. Regression task

We demonstrate our approach aboard an autonomous nano-UAV that performs the human pose estimation task [2]. A perception CNN takes gray-scale 160×96 px camera frames and estimates the subject's 4DOF pose $[x, y, z, \phi]$, relative to the drone's horizontal frame as shown in Figure 1. Therefore, the model output poses do not depend on the pitch and roll orientations of the drone and human subjects. The poses are processed with a Kalman filter and fed to a closed-loop velocity controller that autonomously flies the drone, keeping a fixed distance Δ in front of the human subject and centering them in the image.

B. Robotic platform

Our target robot is the Bitcraze Crazyflie 2.1, a commercial off-the-shelf (COTS) nano-UAV, extended by the pluggable AI-deck and Flow-deck companion boards as depicted in Figure 2. Table II compares the computational and memory resources available on this drone. The Crazyflie relies on

TABLE II
COMPARISON OF SoCs ABOARD CRAZYFLIE NANO-UAVs

SoC (ISA)	CPU cores	L1 [kB]	L2 [MB]	L3 [MB]	SIMD instr.	FPUs	Max freq. [MHz]	Power [mW]
STM32 (ARM)	1	64	0.1	–	–	1	168	116
ESP32 (Xtensa)	2	32	0.5	–	–	2	240	300
GAP8 (RISC-V)	8 (+1)	64	0.5	8	✓	–	175	96
GAP9 (RISC-V)	9 (+1)	**128**	**1.5**	**32**	✓	**4**	**370**	**66**

an STM32 single-core microcontroller unit (MCU) for low-level flight control and state estimation. It can reach up to 7 min flight time on a single 380 mA h battery. The AI-deck extends onboard sensing and processing capabilities with a GreenWaves Technologies (GWT) GAP8 SoC and a Himax HM01B0 gray-scale QVGA camera, while the Flow-deck provides a time-of-flight laser-based altitude sensor and an optical flow sensor to improve the drone's state estimation for autonomous flight.

The GAP8 SoC is an embodiment of the parallel ultra-low power platform (PULP) [21], shown in Figure 3. It features two power domains: a computationally-capable 8-core cluster (CL) and a single-core fabric controller (FC), in charge of data orchestration for the CL's execution. All cores are based on the RISC-V instruction set architecture; the FC can reach up to 250 MHz, while the CL peaks at 175 MHz. The on-chip memories are organized in a fast 64 kB scratchpad L1 and a slower 512 kB L2 memory. Additionally, the AI-deck features also 8 MB off-chip DRAM memory and 64 MB Flash. Finally, the GAP8 does not provide any hardware support for floating-point calculations and requires either costly soft-float emulation (10× overhead) or fixed-point arithmetic through quantization.

The next-generation SoC GAP9 marks significant improvements compared to GAP8, as shown in Table II. Most significantly, GAP9's CL includes four shared floating point units (FPUs), which execute floating-point instructions in a single clock cycle. In the context of on-device training, floating-point hardware support is extremely valuable for back-propagation, for which the basic primitives are implemented in the *PULP-TrainLib* [8] software library. For convolutional layers, PULP-TrainLib running on the GAP9 achieves a peak performance efficiency of 5.3 and 4.6 multiply-accumulate operations (MAC) per clock cycle for, respectively, the forward and backward passes.

C. Baseline models

We demonstrate our approach on the PULP-Frontnet CNN [2], a field-proven model for human pose estimation aboard nano-UAVs. The architecture, in Figure 4 is composed of eight convolutional layers, based on the pattern *conv, batch norm, ReLU*, and one fully connected, for a total of 304 k parameters. The total computational load for inference is 14.3 MMAC (million MAC) per frame, which leads to a throughput of 48 Hz while consuming only 96 mW when
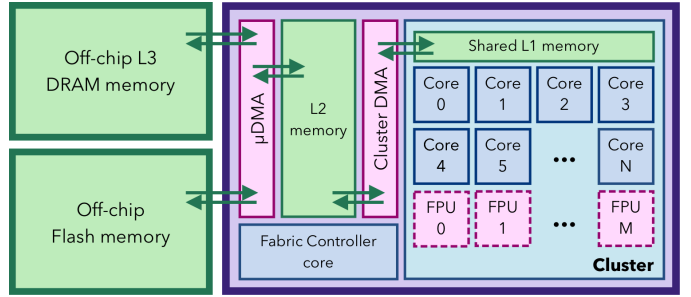

Fig. 3. The parallel ultra-low power System-on-Chip architecture (PULP). Cluster cores perform parallel computationally intensive workloads, while a fabric controller orchestrates data transfers through two direct memory access (DMA) units. Optional floating-point units (FPU) are shared in the cluster.

deployed with 8-bit integer quantization on the GAP8. By comparison, computing one update step (forward + backward) for the entire architecture requires 3.7× as many operations (53.1 MMAC).

Initial training: we employ two datasets for human pose estimation in the initial training, from which we obtain two baseline models. The first [2] is acquired in the real world and contains 2.6 k training samples with mm-precise mocap-based labels. The second [16] provides 75 k images and ground-truth labels from the Webots simulator. In line with the state of the art [2], we train the models with the Adam optimizer at a learning rate of 10^{-3} for 100 epochs, choosing the best model during validation.

IV. METHOD

A. On-device fine-tuning

For our fine-tuning procedure, we implement the full back-propagation algorithm on an ultra-low power embedded device. As in [8], we consider a feed-forward neural network of N layers. Each layer computes a non-linear function f parameterized by a weight tensor W_i:

$$Y_i = f_{W_i}(X_i) \qquad i \in 0, \ldots, N-1 \qquad (1)$$

where X_i and Y_i are input and output tensors. The network is a composition of the layers, where each layer's input $X_i = Y_{i-1}$ is the previous layer's output, up to the model's input X_0.

We train the network on a loss function $\mathcal{L}$ defined on the model outputs. An optimization procedure, e.g., stochastic gradient descent (SGD), iteratively updates the weights W_i to minimize $\mathcal{L}$ according to the update rule:

$$W_i \leftarrow W_i + \eta dW_i \qquad (2)$$

based on the gradients of the loss function w.r.t. each weight, $dW_i = \delta\mathcal{L}/\delta W_i$, and a learning rate η.

Back-propagation enables efficient computation of the gradients through the gradient chain rule. It performs a forward pass (FW), which computes the model's output Y_{N-1} and the loss function, followed by a backward pass (BW). For each layer starting from the last, the BW pass computes the input gradient dX_i, used to propagate the errors to the previous layers (BW-IG), and the weight gradient dW_i, used to update the layer's own parameters (BW-WG).
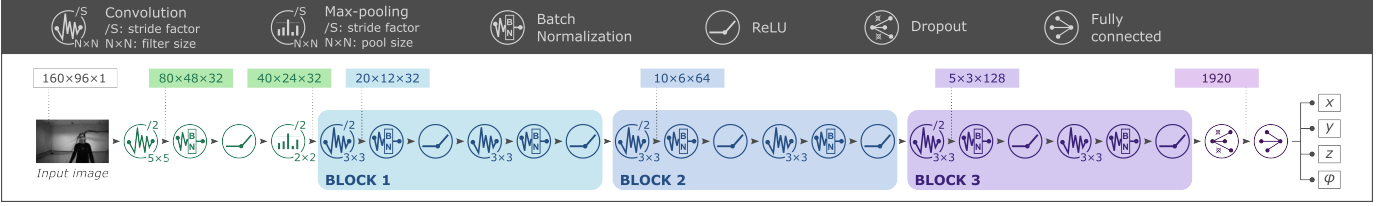
Fig. 4. PULP-Frontnet [2], our target CNN architecture with 9 layers and 304 k parameters. Inference requires 14.3 MMAC operations per frame.

TABLE III
MEMORY AND COMPUTATIONAL REQUIREMENTS OF FINE-TUNING METHODS

Fine-tuning strategy	Params [k]		Memory [kB/frame]				Computation [MMAC/frame/step]			
			Input	Activations	Weight grads	Tot.	FW	BW-IG	BW-WG	Tot.
all	304.4	(100%)	15.0 (u8)	870.0 (f32)	1188.9 (f32)	2073.9	14.3	24.5	14.3	53.1
bn	1.0	(0.33%)	15.0 (u8)	585.0 (f32)	3.8 (f32)	603.8	14.3	24.4	0.10	38.8
bias	**0.5**	(0.15%)	15.0 (u8)	**0.8** (u1)	**1.9** (f32)	**17.7**	14.3	24.4	–	38.7
fc	7.7	(2.5%)	**1.9** (u8)	–	30.0 (f32)	31.9	**0.01**	–	0.01	**0.02**

The trainable parameters for affine layers, such as convolution, fully connected, and batch normalization, are weight W_i and bias B_i tensors. Thus, their BW-IG and BW-WG steps correspond, respectively, to the products:

$$dX_i = dY_i \cdot \delta Y_i / \delta X_i = dY_i \cdot W_i^T \quad (3)$$

$$dW_i = \delta Y_i / \delta W_i \cdot dY_i = X_i \cdot dY_i \quad (4)$$

$$dB_i = \delta Y_i / \delta B_i \cdot dY_i = 1 \cdot dY_i \quad (5)$$

Crucially, BW-IG and BW-WG make neural network training $\sim 3\times$ as computationally expensive as inference (which executes only the FW phase). In addition, computing dW_i requires storing intermediate outputs X_i from the FW to the BW pass, quickly making training unfeasible on memory-constrained embedded devices. Finally, weights and weight gradients typically have order-of-magnitude differences in scale, which complicates resorting to reduced precision arithmetic (16-bit float or quantized int). Thus, we focus our implementation to 32-bit float arithmetic.

Fine-tuning strategies: Our hardware limitations motivate us to investigate strategies to reduce the cost of the fine-tuning workload. We select four fine-tuning strategies from the family of sparse updates, adhering to two design constraints: *i)* each strategy updates a fixed subset of the model parameters, *ii)* each subset contains a uniform type of parameters (e.g., batch normalization, fully connected, or biases). Under these constraints, the strategies follow from optimizing different regression performance/memory/computation trade-offs due to the characteristics of each layer's forward and backward phases. In Table III, we analyze the workload of one optimization step (FW + BW) with our architecture on a single frame, depending on which subset of model parameters is updated.

(a) Entire model (all): our baseline setting is to update the weight and biases of every layer in the model, the most expensive method at 53 MMAC per frame. In addition, it requires storing all intermediate activations and weight gradients, which in 32-bit floats amount to 2 MB for a single frame, 25% of

the AI-deck's 8 MB DRAM, which is also needed to store our fine-tuning dataset.

(b) Only batch-norm layers (bn): batch normalization layers contain a mere 0.33% parameters of the entire model while significantly impacting model performance [6]. This method requires 70% less memory and 26% less computation due to the smaller weight gradients.

(c) Only biases (bias): as BW-WG of the biases is simply the BW-IG of the following layer, full intermediate activations don't need to be stored [7]. The 1-bit signs of the intermediate activations are still needed for BW-IG of ReLU non-linearities, 0.8 kB per frame. Total memory decreases to just 17.7 kB, while computation remains comparable to *bn*.

(d) Only the fully-connected layer (fc): by far, the greatest reduction comes from fine-tuning only the final layer of the network, although this also has the least expected improvement among the four methods [7]. The forward pass of all layers up to the final fully connected one can be pre-computed just once at the beginning of fine-tuning, a 99.9% reduction in computation. In our implementation, this pre-computed forward pass is further quantized to 8-bit integers and runs in real-time at 48 Hz during fine-tuning set acquisition. The fine-tuning dataset then only stores the resulting feature vectors (1920 elements) instead of full input images (160×96 px), reducing memory storage by 98.1%.

Hyper-parameters: compared to the initial training, we substitute Adam with stochastic gradient descent (SGD) due to $3\times$ lower computation cost. With hyper-parameter search, we set the learning rate to 10^{-2} to retain comparable regression performances to Adam. Additionally, we select a fine-tuning set size of 512 samples, such that the entire dataset fits the AI-deck's 8 MB DRAM memory even for methods that require storing the full input images (7.8 MB), and set five epochs as fixed fine-tuning length. We always select the final model at the end of fine-tuning, thus avoiding storing multiple model checkpoints.

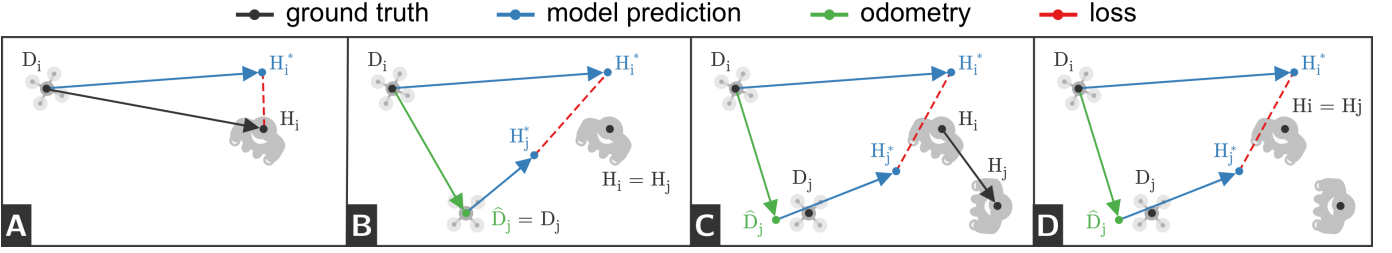**Data augmentation:** we employ standard photometric data

Fig. 5. Loss functions: A) task loss, B) original state-consistency loss [27], and C-D) our state-consistency loss with uncertain drone odometry and moving subject. Subject movements are either C) known or D) unknown (subject assumed still). The depicted reference frames represent drone D and subject H ground-truth poses, at times i and j, and the corresponding relative poses estimated by, respectively, drone odometry $\hat{D}$ and model predictions H^*.

augmentations, including exposure and contrast adjustment, Gaussian noise, box blurring, and vignetting. These transformations operate mostly pixel-wise, benefiting from parallelization and high spatial locality, allowing efficient implementations on embedded devices such as GAP SoCs. Furthermore, we randomly flip images and ground-truth labels horizontally to guarantee the fine-tuning set follows a symmetric distribution along the y axis and yaw orientation. Similarly, we randomly apply time reversal when fine-tuning with state consistency to impose that relative poses $\mathbf{T}_{D_i}^{D_j}$ and $\mathbf{T}_{H_i}^{H_j}$ also respect a zero-mean symmetric distribution (i.e., centered on the identity).

B. Self-supervised loss

We perform self-supervised learning with the state-consistency loss introduced by Nava *et al.* [27]. We consider a dataset of fine-tuning samples acquired at timesteps $i \in \mathcal{T}$, where each sample comprises an image from the drone's camera and the drone's and subject's poses. We define $\mathbf{T}_A^B$ in SE(3) as the relative pose of reference frame B w.r.t. A. The fine-tuning process minimizes the loss function

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \lambda_{\text{sc}} \mathcal{L}_{\text{sc}}, \qquad (6)$$

composed of a task loss term $\mathcal{L}_{\text{task}}$ and a state-consistency loss term $\mathcal{L}_{\text{sc}}$. In accordance with [27], we set $\lambda_{\text{sc}} = 1$. Figure 5 depicts the two loss terms and their effects on model predictions. The **task loss** is defined on individual timesteps i, taken from the (possibly empty) subset $\mathcal{T}_t \subseteq \mathcal{T}$ for which target relative poses $\mathbf{T}_{D_i}^{H_i}$ are known:

$$\mathcal{L}_{\text{task}} = \frac{1}{|\mathcal{T}_t|} \sum_{i \in \mathcal{T}_t} \Delta(\mathbf{T}_{D_i}^{H_i^*}, \mathbf{T}_{D_i}^{H_i}), \qquad (7)$$

where $\mathbf{T}_{D_i}^{H_i^*}$ represents the model's estimate of the subject w.r.t. the drone relative pose at time i and $\Delta(\mathbf{T_1}, \mathbf{T_2})$ is a distance function between relative poses, which we define as the L1 loss between 4DOF pose vectors (x, y, z, ϕ)[1]. When $\mathbf{T}_{D_i}^{H_i}$ are ground-truth relative poses (e.g., acquired by a motion capture system), this loss is equivalent to ordinary supervised learning. Noisy estimates can also be used as $\mathbf{T}_{D_i}^{H_i}$.

---

[1] Angles are expressed in radians and angle differences are computed on the circle group, to account for discontinuities at $\pm\pi$. This weighs equally position errors of 1 m and rotation errors of 1 rad, which is a reasonable heuristic in our setting.

Our experiments explore the case when the relative pose is known at a time i, and the subject subsequently remains still, which allows us to define the relative pose at a later time j as $\mathbf{T}_{\hat{D}_j}^{H_j} = \mathbf{T}_{\hat{D}_j}^{D_i} \mathbf{T}_{D_i}^{H_i}$. The relative pose $\mathbf{T}_{D_i}^{\hat{D}_j}$ indicates the (possibly noisy) odometry estimate of the drone's pose at time j w.r.t time i.

The **state-consistency loss** is defined instead on pairs of timesteps i and $j = i + dt$ at a fixed time delta (a hyperparameter) sampled from the subset $\mathcal{T}_{\text{sc}} \subseteq \mathcal{T}$:

$$\mathcal{L}_{\text{sc}} = \frac{1}{|\mathcal{T}_{\text{sc}}|} \sum_{i \in \mathcal{T}_{\text{sc}}} \Delta(\mathbf{T}_{H_i^*}^{D_i} \mathbf{T}_{D_i}^{\hat{D}_j} \mathbf{T}_{D_j}^{H_j^*}, \ \mathbf{T}_{H_j}^{H_i}), \qquad (8)$$

where $\mathbf{T}_{H_j}^{H_i}$ is the subject's relative pose at time j w.r.t time i. Compared to [27], we reformulate the state-consistency loss to model drone and subject movements separately. Noisy estimates for the former are generally available through drone odometry, while the latter are not known by the drone. In the experiments, we evaluate the regression performance impact of these sources of uncertainty individually as depicted in Figure 5-BCD. Subject movements $\mathbf{T}_{H_j}^{H_i}$ will be replaced by the identity matrix $\mathbf{I}$, *i.e.*, we assume the subject stands still.

C. Embedded implementation

We implement the proposed fine-tuning process in optimized C code for the PULP architecture. Figure 6 shows the algorithm pipeline, depicting dependencies and concurrent operations. We compute the forward, loss function, and backward passes using hand-written 32-bit float kernels that operate entirely in L1 memory. The 512-sample fine-tuning set, composed of input images and the corresponding 4-element label outputs, totals 7.8 MB stored as 8-bit quantized integers in the off-chip L3 memory. The *fc* method, which only ever needs to perform the forward pass before the fully-connected layer once (see Section IV-A), stores just the fully-connected layer's 1920-element input features and label outputs ($\sim$1 MB).

During fine-tuning, the dataset is processed in 16 batches of 32 samples each. Each batch is transferred from L3 to L2 memory, while samples are transferred to L1 memory one by one. Both transfers exploit the SoCs' DMA peripherals. Weight gradients are accumulated in L1 memory and applied to the weights once per batch. The updated weights are written out to L2 at the end of each epoch.

Optimizations: In our kernel implementations, we apply four software optimizations to exploit the target platform's
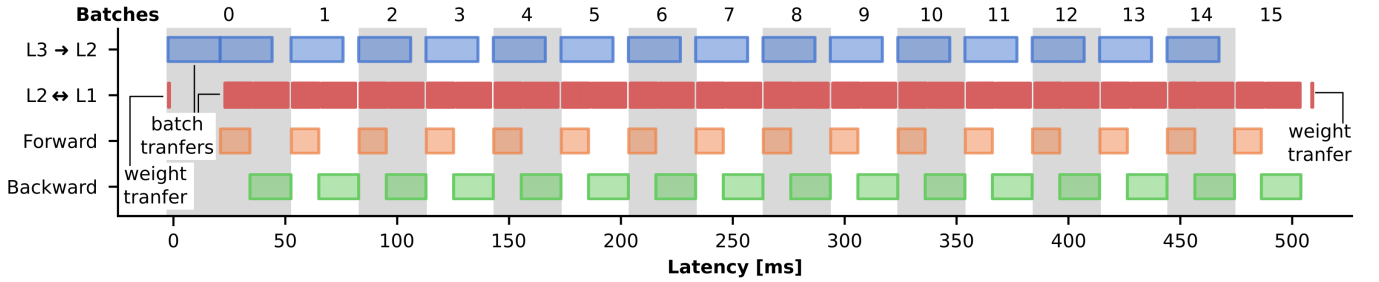
Fig. 6. One epoch of the fine-tuning pipeline on GAP9 with all optimizations (8 cores, L3-L2 double buffering, loop unrolling, and L1 memory).

TABLE IV
AVERAGE REGRESSION PERFORMANCE

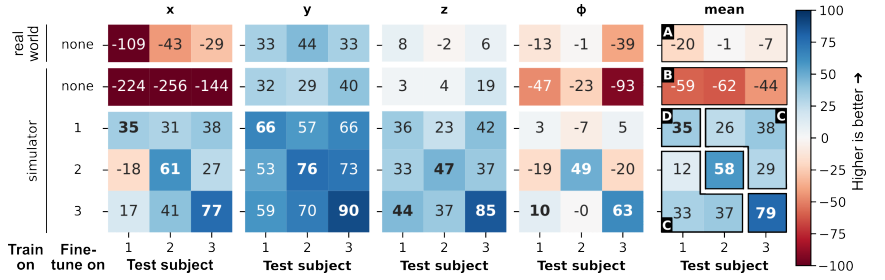| Train on	Fine-tune on	MAE	R^2 [%]	cfr. Fig. 7
real world	–	0.50	-9.3	(A), SotA [2]
	env.	0.39	23.5	–
	env. + subj.	0.35	41.5	–
sim.	–	0.61	-55.0	(B), SotA [16]
	env.	0.38	29.1	(C)
	env. + subj.	**0.27**	**57.4**	(D)

Fig. 7. R^2 scores [%] for each combination of fine-tuning and test subjects.

hardware architecture efficiently.

i) Double buffering: L3-L2 memory transfers are constrained by the limited off-chip memory bandwidth ($\sim$90 MB/s). As shown in Figure 6, we introduce double buffering to schedule DMA transfers concurrently with forward/backward computation, thereby fully hiding transfer latency (except for the first batch in each epoch).

ii) Loop unrolling: on GAP chips (strictly in-order architectures), memory instructions suffer from load stalls due to access latency. To hide these latencies, we manually unroll 4 iterations of our kernels along the output tensor dimension.

iii) L1 memory: We take advantage of the cluster L1 memory area (64 kB on GAP8, 128 kB on GAP9, single-cycle access time) for every buffer (i.e., input, outputs, labels, weights, and weight gradients) in the inner loops of our kernels. The cluster DMA handles L2-L1 transfers sample-by-sample.

iv) Parallelization: We take advantage of the GAP chips' multi-core cluster by executing the forward/backward computation on 8 cores, with a blocking strategy that distributes work among cores along the input tensor dimension. Section V-C measures the latency improvements of each optimization individually.

Quantization: At inference time, we run the int-8 quantized model obtained through Parameterized Clipping Activations (PACT) [33] and Quantization-Aware Training (QAT). The quantization process is first performed offline on the non-finetuned baselines through the same deployment pipeline adopted by the original works [2], [16]. These quantized initial model parameters are deployed to our SoC's Flash memory. At fine-tuning time, the quantized weights are loaded and dequantized to a 32-bit floating-point in the on-chip L2 memory. Fine-tuning is then performed entirely in floating-point arithmetic. Updated weights are quantized again at the end of the fine-tuning process to the same 8-bit integer representation used for inference. In particular, we found the overall scale of model parameters does not change significantly during fine-tuning.

As such, we quantize using the same quantization parameters (quantization ε and bit precision) as the non-finetuned model, without repeating QAT.

V. RESULTS

A. Fine-tuning strategy

We start our experiments by analyzing the proposed fine-tuning methods offline in the PyTorch framework on a set of 4.7 k real-world images from [14]. This dataset comprises 18 in-flight sequences from three subjects in challenging situations, e.g., different subject appearances, dynamic motions of both subjects and the drone. It is collected with the same robotic platform introduced in Section III (including the Himax camera) and is used both for fine-tuning the pre-trained models and testing them. This dataset stresses the domain shift problem, as it represents a completely novel domain for the baseline models, which are trained either entirely in simulation or in a different real-world environment.

For each subject, we select a random temporally-contiguous 128 s segment of the dataset (512 samples @ 4 Hz) as the fine-tuning set, while the rest is used as the test set. To provide unbiased measures of regression performance, we discard 100 contiguous samples (25 s) between fine-tuning and test segments, and we apply cross-validation, repeating 3 runs for each subject with different random fine-tuning segment (9 total experiment runs). At most, 75% of each subject's samples are used as the fine-tuning set.

Baseline performance: A lower bound on the test regression performance achieved by our models is obtained by not doing any fine-tuning. An upper bound corresponds to fine-tuning in the best-case scenario: we fine-tune all model parameters, i.e. method *all*, entirely on the task loss (*i.e.*,
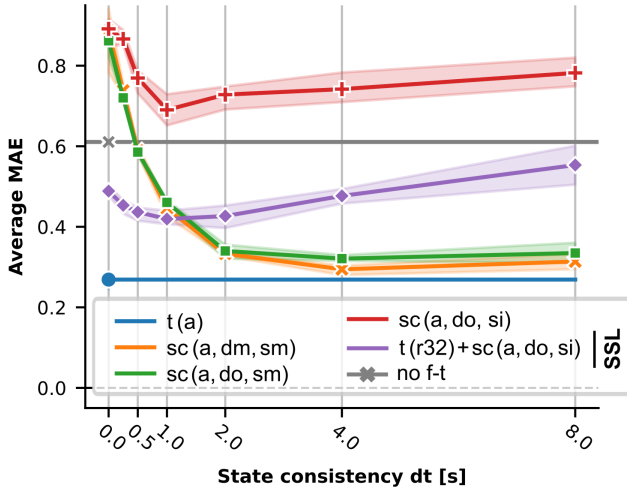
Fig. 8. Regression performance across loss functions. The self-supervised loss (SSL) is computed entirely on data from the drone's onboard sensors and reaches up to 50% of the MAE improvement of the supervised loss $t(a)$.

$\mathcal{T}_t = \mathcal{T}$ and $\mathcal{T}_{sc} = \varnothing$) and assuming perfect knowledge of the drone and subject poses.

Table IV reports the resulting mean absolute errors (MAE) and R^2 scores[2]. Figure 7 breaks down these results by the test subject, both on the four individual regression outputs and as an average. Model (*A*), trained on real-world data and not fine-tuned, represents the performance of the original PULP-Frontnet [2] on our test set. Model (*B*) is trained on simulated data, is not fine-tuned, and constitutes our lower bound.

For the fine-tuned models, we report several averages: (*D*) reports the average performance of a model when tested on the same subject and the same environment that it was fine-tuned on; (*C*) is the average performance of a model, tested on a different subject (but the same environment) that it was fine-tuned on. Table IV shows that, despite the higher initial performance, the models trained with real-world data achieve a poorer fine-tuned performance than those trained on simulation data. This confirms that training on a larger and more varied simulation train set translates to a model that better adapts to unseen subjects and environments when fine-tuned. Therefore, in the following experiments, we focus on scenario (*D*): models trained on simulated datasets and tested on the same subject and environment they were fine-tuned on.

Self-supervised learning: in Figure 8, we report our fine-tuning results obtained when ground-truth labels are unavailable. We consider three setups for the drone pose – *perfect absolute pose*, *perfect odometry* (dm), and *uncertain odometry* (do) – and three for the pose of the human subject – *perfect absolute pose*, *perfect odometry* (sm), and *unknown* (si).

When perfect absolute poses are known for both drone and subject, we have the ground-truth information to fine-tune using $\mathcal{L}_t$ (*i.e.*, regular supervised learning). The *supervised*

[2]The R^2 score is a normalized metric of regression performance that aids comparisons among different sets of data. A perfect model scores 100%, while a dummy model always predicts the test set mean scores zero.

TABLE V
REGRESSION PERFORMANCE COMPARED ACROSS FINE-TUNING METHODS

	Fine-tuning	none	*all*	*bn*	*bias*	*fc*
MAE	Supervised	0.61	**0.27**	0.36	0.39	0.45
	SSL	–	**0.43**	**0.43**	0.46	0.47

case where these are known for all samples, considered in all previous experiments, is named $t(a)$ and reaches MAE 0.27.

To reduce our reliance on privileged information, we assume only odometry is known, *i.e.* relative poses between two instants in time. When odometry is perfect for both drone and subject on all samples $sc(a, dm, sm)$, we can fine-tune using $\mathcal{L}_{sc}$, which achieves 93% of the ideal improvement. Uncertain drone odometry $sc(a, do, sm)$ also has a limited impact on performance and achieves 85% of the ideal. Performance improves at higher state-consistency time deltas dt (horizontal axis), as imposing state-consistency between samples farther in time carries a larger information content.

On the other hand, an unknown subject odometry $sc(a, do, si)$ drastically reduces performance[3]. In this case, in order to compute $\mathcal{L}_{sc}$, we assume that $\mathbf{T}_{H_i}^{H_j} = \mathbf{I}$, *i.e.*, that the subject is always still. Although the time reversal augmentation ensures this holds on average, *i.e.*, the fine-tuning set has $\mathbb{E}[\mathbf{T}_{H_i}^{H_j}] = \mathbf{I}$ by design, the model degenerates to a dummy predictor that always outputs a constant value.

To address the above issue, we design a cooperative scenario in which the subject moves to a known pose w.r.t. the drone (e.g., 1 m in front of the drone, directly facing the camera and centered in the field of view). While the subject stands still, the drone randomly moves around to acquire a small number of samples. The procedure is repeated from multiple start locations in the environment to acquire highly diverse fine-tuning data. We test this scenario, called $t(r32) + sc(a, do, si)$, by selecting a random 32-sample subset of frames on which we optimize $\mathcal{L}_t$. It relies on realistic in-field infrastructure-free data acquisition and achieves a significant improvement, up to 39%. As such, we select it as the SSL loss function in the following experiment, in the best-performing $dt = 2\,s$ configuration.

Fine-tuning methods: in Table V, we explore the effectiveness of methods that reduce the fine-tuning workload by limiting the subset of model parameters to update. Full fine-tuning, named *all*, sets the lower bound at an MAE of 0.27 (−56 % compared to the non-finetuned baseline). Optimizing only the batch-norm layers, *bn*, is second best, followed closely by fine-tuning the biases, *bias*. Fine-tuning the final fully connected layer, *fc*, performs the worst, but notably still shows an MAE improvement of −26 %.

B. In-field experiments

In this experimental section, we assess the end-to-end performance of our system when deployed in the real world, i.e.,

[3]The case with unknown subject odometry but perfect drone odometry $sc(a, dm, si)$, omitted, exhibits similar behavior.

TABLE VI
IN-FIELD EXPERIMENT RESULTS (AVERAGE OVER 3 RUNS).
THE SYMBOL * MARKS PREMATURELY FAILED RUNS.

	Model	Completed path [%]	R^2 [%]			Control error	
			x	y	ϕ	e_{xy} [m]	e_θ [rad]
	mocap	100	100.0	100.0	100.0	0.18	0.21
SotA	real world [2]	0	-80.7	26.3	1.9	2.68*	0.32*
SotA	simulator [16]	91	9.0	42.1	12.7	1.05*	0.37*
Supervised	all	100	**64.5**	**75.8**	**24.6**	0.70	0.47
Supervised	bn	76	53.4	62.3	20.5	0.60*	0.30*
Supervised	bias	100	46.3	55.2	18.8	0.72	**0.46**
Supervised	fc	100	44.9	54.5	15.6	**0.67**	0.47
SSL	all	92	**36.2**	**55.4**	**21.1**	1.03*	0.45*
SSL	bn	100	33.9	54.1	14.6	1.00	0.59
SSL	bias	100	33.3	48.8	15.8	0.99	0.43
SSL	fc	100	32.9	48.0	15.8	**0.78**	**0.42**

TABLE VII
ON-DEVICE FINE-TUNING COST (5 EPOCHS)

	SoC and operating point (Vdd, FC and CL frequencies)	Latency [ms]	Avg. Power [mW]	Energy [mJ]
GAP8	most energy efficient (1.0 V, 25 MHz, 75 MHz)	18 899	24.6	464.7
GAP8	peak throughput (1.2 V, 250 MHz, 175 MHz)	6553	101.5	664.9
GAP9	most energy efficient (0.65 V, 240 MHz, 240 MHz)	785	**18.7**	**14.7**
GAP9	peak throughput (0.80 V, 370 MHz, 370 MHz)	**511**	38.3	19.5

in the field. We challenge our models in an environment and test subject combination that significantly differs from those seen during initial training. As described above, we collect a 512-sample fine-tuning dataset, then fine-tune eight models: four fine-tuning methods, *all*, *bn*, *bias*, and *fc*, multiplied by two loss functions, supervised and self-supervised. We record three flights for each of our two baselines and eight fine-tuned models ($3 \times 10 = 30$ flights), plus one additional flight based on the perfect mocap position of the subject, for a total of 31 test flights. As the GAP9 SoC is not currently available on a COTS module for the Crazyflie, for in-field experiments, we deploy all fine-tuned models on GAP8, applying 8-bit integer quantization to achieve a real-time inference throughput of 48 frame/s.

We reproduce the same testing procedure introduced in [2]: the human subject walks a predetermined path, with distinct phases designed to stress the different components of the model prediction, while the fully autonomous nano-UAVs must stay in front of the subject at a distance of 1.3 m. The results are summarized in Table VI, where we report the path completion (in percentage), the R^2 regression performance, and the control errors, all of them averaged over the three runs of each model[4].

We define path completion as the percentage of the expected path the drone can follow, and we conclude the test if the subject exits the camera's field of view or the drone leaves our 6×6m flight arena. The SotA real-world baseline [2] immediately fails the experiment: it moves away from the subject instead of approaching, thus completing 0% of the path. The SotA simulator baseline [16] manages to follow the subject two out of three runs for the entire path, resulting in a completion score of 91%. Fine-tuned models track the subject until the end of every experiment, except one run each of the supervised *bn* and self-supervised *bias* models. The supervised *bn* run is aborted because the drone comes close to the subject, while the drone in the self-supervised *all* run drifts out of the arena.

The second metric analyzed in Table VI is the R^2, which indicates the regression performance on a test set. For comparability between models, we build this test set from all 31 test flights, resulting in 60 104 images. The relative trends between models closely follow those in Section V-A, i.e., fine-tuned models outperform the simulator baseline by up to 55%, and self-supervised models also manage a 27% improvement.

Finally, the last metric we present in Table VI is the control performance, which is composed of the mean horizontal position error e_{xy} and the mean absolute angular error e_θ. The former is computed against the desired position of the drone (i.e., 1.5 m in front of the subject) while e_θ accounts for the drone orientation vs. the desired one (i.e., facing the subject). Compared to the two SotA baselines, the supervised models improve the in-field control performance by -36% on e_{xy}. Self-supervised models *all*, *bn*, and *fc* show only a minor reduction in control error compared to the SotA simulator baseline (i.e., -6% on e_{xy}), while *fc* marks a clear improvement w.r.t the same baseline, with e_{xy} of just 0.78 m.

This last result on the SSL *fc* remarkably shows only a minor degradation (only 16%) in control performance compared with the vastly costlier supervised *all*, i.e., $10\,000\times$ more operations and $200\times$ more memory than SSL *fc*. To further investigate this key finding, we show in Figure 9 the individual predictions of each model. From these, we notice the real-world baseline consistently under-estimates x, outside the controller convergence zone (in green). On the contrary, all fine-tuned models show improved performance compared to the baselines. Self-supervised models reduce prediction noise even more than supervised models but show stronger miscalibration w.r.t. the ground-truths, which should be explored in future work. The better control performance by SSL *fc* w.r.t. the other three SSL models is explained by the inability of the latter three to estimate phi.

C. On-device fine-tuning

As a final experiment, we deploy the proposed method on the GAP8 and GAP9 SoCs and profile its workload. Due to its computational and memory advantages, combined with the highest in-field self-supervised performance, we focus on *fc* for on-device deployment.

Figure 10 shows the detailed power traces for one fine-tuning epoch at different operating points. Instead, Table VII

---

[4]Supplementary video material for the in-field experiments available online: https://youtu.be/3yNbMwszpSY
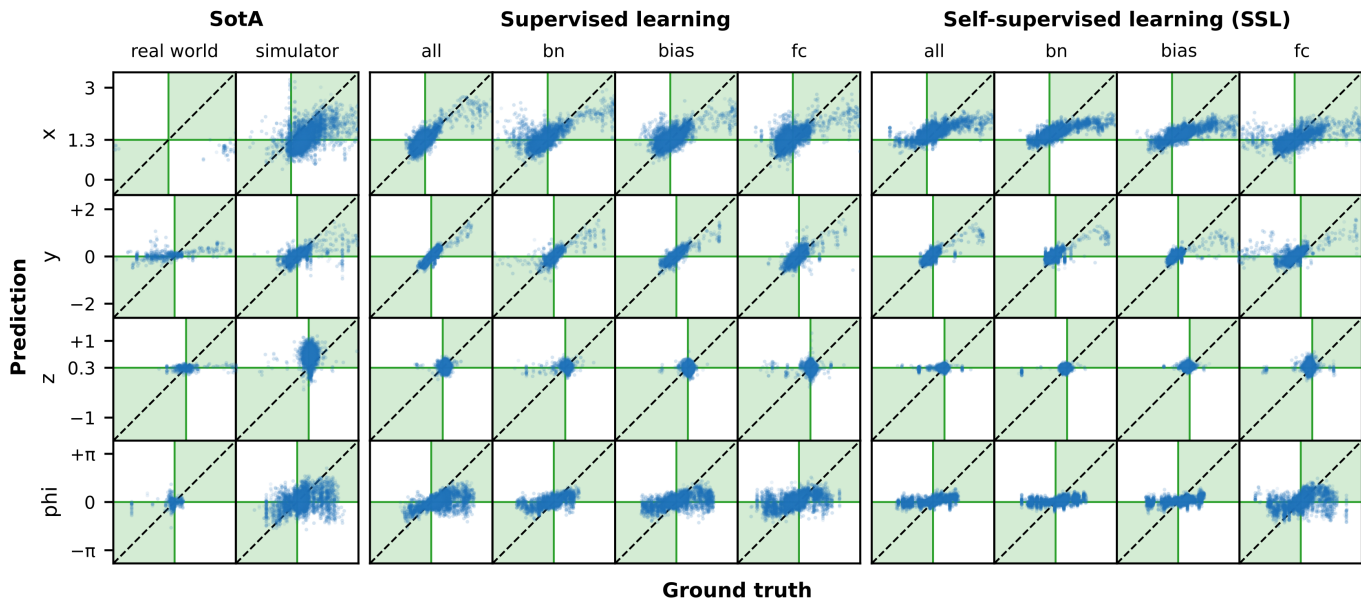
Fig. 9. Predictions (vertical) vs. ground-truths (horizontal) for each model during its three in-field experiment runs. Dashed diagonals correspond to a perfect predictor. Green lines mark the control setpoints: predictions in the green area contribute to drive the drone towards the desired pose.
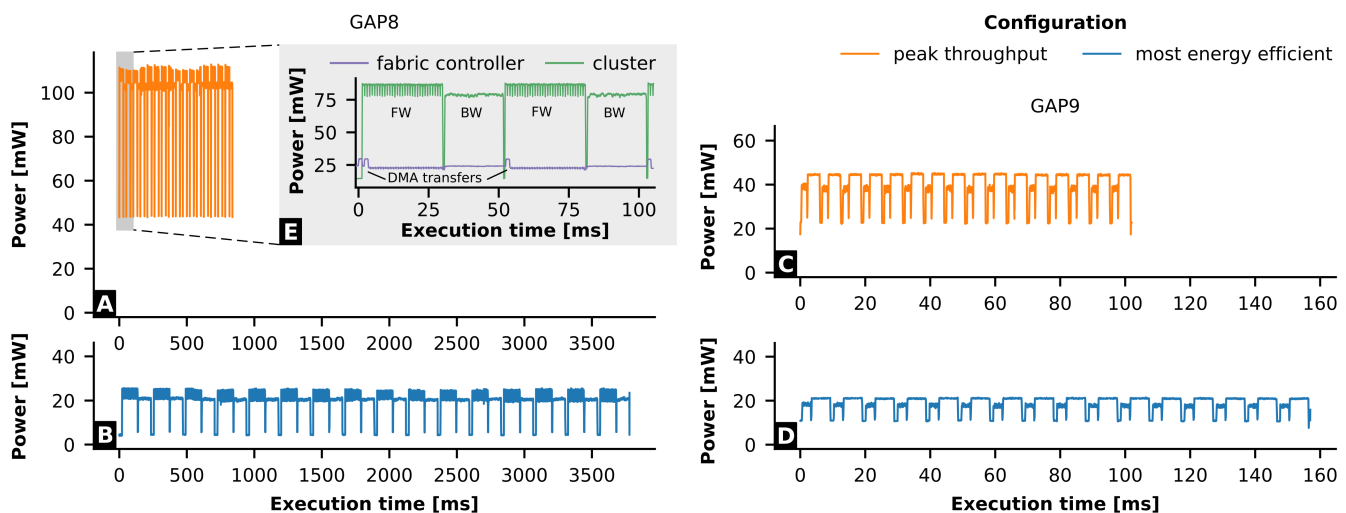

Fig. 10. Power traces for one epoch of the *fc* fine-tuning method, on A-B) GAP8 and C-D) GAP9. SoCs are configured in the A-C) *peak throughput* and B-D) *most energy efficient* operating points. E) Details of the first two batches on GAP8 at peak throughput, with power consumption broken down into fabric controller and cluster power domains.

provides aggregate metrics on the fine-tuning process composed of 5 epochs. The zoom-in on the first two batches in Figure 10-C highlights the sequence of forward (FW) and backward (BW) passes with their distinct power consumption patterns. The breakdown into fabric controller and cluster power consumption shows the double-buffered DMA transfers perfectly overlapped with computation to hide their latency. GAP8 and GAP9 exhibit similar behavior, albeit at significantly different time scales: soft-float emulation on GAP8, due to the lack of FPUs, introduces more than $10\times$ overhead. As a result, in the *peak throughput* configuration of GAP8, one epoch takes $1311\,\mathrm{ms}$ vs $102\,\mathrm{ms}$ on GAP9. GAP9 also more than halves the average power consumption from $102\,\mathrm{mW}$ to $38\,\mathrm{mW}$, which compounds to reduce the energy

required for fine-tuning by $34\times$. The *most energy efficient* configuration captures the case in which conserving energy is more important than the fine-tuning latency (e.g., when the drone lands to perform the fine-tuning process). In this operating point, GAP9 reduces energy consumption by -25% compared to peak throughput while increasing fine-tuning latency by 53%. These results confirm the feasibility of on-device learning aboard ultra-low power embedded devices in a real-world robotic application.

Figure 11 measures the impact of the software optimizations described in Section IV-C on our kernels' performance for the *fc* fine-tuning strategy. Combined, the four optimizations achieve an $80\times$ speed-up compared to a baseline platform-agnostic C implementation. Specifically, double buffering
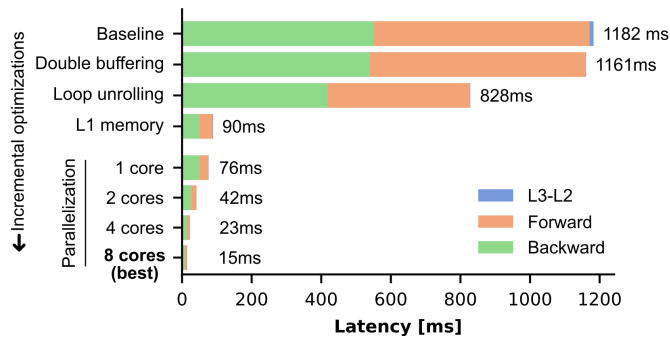
Fig. 11. Latency comparison between kernel implementations with different optimization levels. One batch on GAP9 at the "peak performance" operating point.

hides the latency of L3 DMA transfers by overlapping them with computation. Albeit this has limited impact on the baseline implementation, DMA transfers impose a fixed 20 ms cost that is not impacted by other optimizations, i.e., they would account for more than 50% of total latency of a fully optimized implementation. Loop unrolling, on the other hand, hides load stalls due to L2 accesses. Four-iteration unrolling on the output tensor dimension (i.e., a fully-unrolled kernel, as our model has four-element outputs for the *fc* strategy), yields a -30% improvement. The most impactful improvement, more than $9\times$, is achieved by exploiting L1 memory for all buffers in the kernels' inner loops, with L2-L1 transfers handled by DMA. Finally, parallelization yields a $5\times$ improvement by distributing work among our chip's 8 cores (i.e., a 60% scaling efficiency).

VI. CONCLUSION

We present on-device learning aboard nano-drones to mitigate the general TinyML problem of domain shift. Our fine-tuning approach requires only 19 mW, 1 MB of memory, and runs in just 510 ms (5 epochs) on the best-in-class GWT GAP9 SoC. We employ self-supervised learning to cope with the lack of ground-truth labels aboard our UAV. In-field results show an improvement in control performance up to 26% vs. a non-fine-tuned SotA baseline, making the difference between mission failure and success in a never-seen-before challenging environment.

REFERENCES

[1] D. Palossi, A. Gomez, S. Draskovic, K. Keller, L. Benini, and L. Thiele, "Self-sustainability in nano unmanned aerial vehicles: A blimp case study," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 79–88.
[2] D. Palossi, N. Zimmerman, A. Burrello, F. Conti, H. Müller, L. M. Gambardella, L. Benini, A. Giusti, and J. Guzzi, "Fully onboard AI-powered human-drone pose estimation on ultralow-power autonomous flying nano-UAVs," *IEEE Internet of Things Journal*, vol. 9, no. 3, pp. 1913–1929, 2021.
[3] R. J. Bouwmeester, F. Paredes-Vallés, and G. C. De Croon, "NanoFlowNet: Real-time dense optical flow on a nano quadcopter," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 1996–2003.
[4] L. Lamberti, V. Niculescu, M. Barciś, L. Bellone, E. Natalizio, L. Benini, and D. Palossi, "Tiny-PULP-Dronets: Squeezing neural networks for faster and lighter inference on multi-tasking autonomous nano-drones," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2022, pp. 287–290.
[5] J. Wang, C. Lan, C. Liu, Y. Ouyang, T. Qin, W. Lu, Y. Chen, W. Zeng, and P. S. Yu, "Generalizing to unseen domains: A survey on domain generalization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 8, pp. 8052–8072, 2023.
[6] P. K. Mudrakarta, M. Sandler, A. Zhmoginov, and A. Howard, "K for the price of 1: Parameter efficient multi-task and transfer learning," in *International Conference on Learning Representations*, 2019.
[7] H. Cai, C. Gan, L. Zhu, and S. Han, "TinyTL: Reduce memory, not parameters for efficient on-device learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 285–11 297, 2020.
[8] D. Nadalini, M. Rusci, L. Benini, and F. Conti, "Reduced precision floating-point optimization for deep neural network on-device learning on microcontrollers," *Future Generation Computer Systems*, vol. 149, pp. 212–226, 2023.
[9] J. Eschmann, D. Albani, and G. Loianno, "RLtools: A fast, portable deep reinforcement learning library for continuous control," *arXiv preprint arXiv:2306.03530*, 2023.
[10] Y. D. Kwon, R. Li, S. I. Venieris, J. Chauhan, N. D. Lane, and C. Mascolo, "TinyTrain: Deep neural network training at the extreme edge," *arXiv preprint arXiv:2307.09988*, 2023.
[11] L. Zhu, L. Hu, J. Lin, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "PockEngine: Sparse and efficient fine-tuning in a pocket," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, p. 1381–1394.
[12] Y. D. Kwon, J. Chauhan, H. Jia, S. I. Venieris, and C. Mascolo, "LifeLearner: Hardware-aware meta continual learning system for embedded computing platforms," *arXiv preprint arXiv:2311.11420*, 2023.
[13] M. Nava, L. M. Gambardella, and A. Giusti, "State-consistency loss for learning spatial perception tasks from partial labels," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1112–1119, 2021.
[14] E. Cereda, D. Palossi, and A. Giusti, "Handling pitch variations for visual perception in MAVs: Synthetic augmentation and state fusion," in *13th International Micro Air Vehicle Conference (IMAV)*, G. de Croon and C. D. Wagter, Eds., Sep 2022, pp. 59–65.
[15] A. Moldagalieva and W. Hönig, "Virtual omnidirectional perception for downwash prediction within a team of nano multirotors flying in close proximity," in *2023 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, 2023, pp. 64–70.
[16] L. Crupi, E. Cereda, A. Giusti, and D. Palossi, "Sim-to-real vision-depth fusion CNNs for robust pose estimation aboard autonomous nano-quadcopters," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023, pp. 7711–7717.
[17] M. Pavan, E. Ostrovan, A. Caltabiano, and M. Roveri, "TyBox: an automatic design and code-generation toolbox for TinyML incremental on-device learning," *ACM Transactions on Embedded Computing Systems*, 2023.
[18] H. Ren, D. Anicic, and T. A. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.
[19] D. P. Pau, P. K. Ambrose, and F. M. Aymone, "A quantitative review of automated neural search and on-device learning for tiny devices," *Chips*, vol. 2, no. 2, p. 130–141, May 2023. [Online]. Available: http://dx.doi.org/10.3390/chips2020008
[20] C. D. Wagter, F. Paredes-Vallé, N. Sheth, and G. de Croon, "The sensing, state-estimation, and control behind the winning entry to the 2019 Artificial Intelligence Robotic Racing Competition," *Field Robotics*, vol. 2, no. 1, pp. 1263–1290, mar 2022.
[21] D. Rossi, F. Conti, M. Eggiman, A. Di Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen *et al.*, "Vega: A ten-core SoC for IoT endnodes with DNN acceleration and cognitive wake-up from MRAM-based state-retentive sleep mode," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, 2021.
[22] D. Gandhi, L. Pinto, and A. Gupta, "Learning to fly by crashing," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE Press, 2017, p. 3948–3955. [Online]. Available: https://doi.org/10.1109/IROS.2017.8206247
[23] A. Kouris and C.-S. Bouganis, "Learning to fly by myself: A self-supervised CNN-based approach for autonomous navigation," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE Press, 2018, p. 1–9. [Online]. Available: https://doi.org/10.1109/IROS.2018.8594204

[24] M. Nava, A. Paolillo, J. Guzzi, L. M. Gambardella, and A. Giusti, "Learning visual localization of a quadrotor using its noise as self-supervision," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 2218–2225, 2022.

[25] I. Radosavovic, T. Xiao, S. James, P. Abbeel, J. Malik, and T. Darrell, "Real-world robot learning with masked visual pre-training," in *Proceedings of The 6th Conference on Robot Learning (CoRL)*, vol. 205, 2023, pp. 416–426.

[26] L. Lai, Z. Shangguan, J. Zhang, and E. Ohn-Bar, "XVO: Generalized visual odometry via cross-modal self-training," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2023, pp. 10 094–10 105.

[27] M. Nava, A. Paolillo, J. Guzzi, L. M. Gambardella, and A. Giusti, "Uncertainty-aware self-supervised learning of spatial perception tasks," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6693–6700, 2021.

[28] G. Iyer, J. Krishna Murthy, G. Gupta, M. Krishna, and L. Paull, "Geometric consistency for self-supervised end-to-end visual odometry," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[29] C. Godard, O. Mac Aodha, M. Firman, and G. J. Brostow, "Digging into self-supervised monocular depth estimation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[30] P. Liu, M. Lyu, I. King, and J. Xu, "SelFlow: Self-supervised learning of optical flow," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[31] N. Yang, L. von Stumberg, R. Wang, and D. Cremers, "D3VO: Deep depth, deep pose and deep uncertainty for monocular visual odometry," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2020, pp. 1278–1289.

[32] J. Bian, Z. Li, N. Wang, H. Zhan, C. Shen, M.-M. Cheng, and I. Reid, "Unsupervised scale-consistent depth and ego-motion learning from monocular video," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[33] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv pre-print arXiv:1805.06085*, 2018.

Elia Cereda is a third-year Ph.D. student at the Dalle Molle Institute for Artificial Intelligence (IDSIA, USI-SUPSI) in Lugano, Switzerland. He received a double MSc degree from the University of Milano-Bicocca and USI, Lugano in 2021. His research focuses on developing AI-based autonomous algorithms for pocket-sized robotic platforms. He received Best Paper Awards at the IEEE ICCE'18 conference and ACM EWSN'23 SPICES workshop and was a member of the winning team of the first "Nanocopter AI Challenge" hosted at the IMAV'22 International Conference.

Alessandro Giusti is Professor of AI for Autonomous Robotics at the Dalle Molle Institute for Artificial Intelligence (IDSIA, USI-SUPSI) in Lugano, Switzerland. He leads the institute's research area on Autonomous Robotics. His work focuses on Self-Supervised Deep Learning applied to mobile and industrial robotics. He is the author of more than 80 peer-reviewed publications in top conferences and journals and the recipient of several awards, most of which for innovative applications of deep learning to various fields.
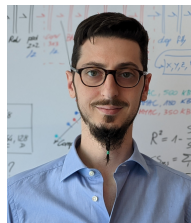
Daniele Palossi (he/his) received his Ph.D. in Information Technology and Electrical Engineering from ETH Zürich. He is currently a Senior Researcher at the Dalle Molle Institute for Artificial Intelligence (IDSIA), USI-SUPSI, Lugano, Switzerland, where he leads the nano-robotics research group, and at the Integrated Systems Laboratory (IIS), ETH Zürich, Zürich, Switzerland. His research stands at the intersection of artificial intelligence, ultra-low-power embedded systems, and miniaturized robotics. His work has resulted in 45+ peer-reviewed publications in international conferences and journals. Dr. Palossi was a recipient of the Swiss National Science Foundation (SNSF) Spark Grant, the 2nd prize at the Design Contest held at the ACM/IEEE ISLPED'19, several Best Paper Awards, and team leader of the winning team of the first "Nanocopter AI Challenge" hosted at the IMAV'22 International Conference.