TinyNS: Platform-Aware Neurosymbolic Auto Tiny Machine Learning

SWAPNIL SAYAN SAHA, University of California - Los Angeles, USA
SANDEEP SINGH SANDHA, Abacus.AI, USA
MOHIT AGGARWAL, BrightNight, USA
BRIAN WANG, University of California - Los Angeles, USA
LIYING HAN, University of California - Los Angeles, USA
JULIAN DE GORTARI BRISENO, University of California - Los Angeles, USA
MANI SRIVASTAVA, University of California - Los Angeles, USA

Machine learning at the extreme edge has enabled a plethora of intelligent, time-critical, and remote applications. However, deploying interpretable artificial intelligence systems that can perform high-level symbolic reasoning and satisfy the underlying system rules and physics within the tight platform resource constraints is challenging. In this paper, we introduce TinyNS, the first platform-aware neurosymbolic architecture search framework for joint optimization of symbolic and neural operators. TinyNS provides recipes and parsers to automatically write microcontroller code for five types of neurosymbolic models, combining the context awareness and integrity of symbolic techniques with the robustness and performance of machine learning models. TinyNS uses a fast, gradient-free, black-box Bayesian optimizer over discontinuous, conditional, numeric, and categorical search spaces to find the best synergy of symbolic code and neural networks within the hardware resource budget. To guarantee deployability, TinyNS talks to the target hardware during the optimization process. We showcase the utility of TinyNS by deploying microcontroller-class neurosymbolic models through several case studies. In all use cases, TinyNS outperforms purely neural or purely symbolic approaches while guaranteeing execution on real hardware.

CCS Concepts: • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: neurosymbolic, neural architecture search, TinyML, AutoML, Bayesian, platform-aware

1 INTRODUCTION

Tiny machine learning (TinyML) refers to hardware and software suites that enable always-on, ultra-low-power (≤ 1 mW), and on-device sensor data analytics on low-end (≤ 1-2 MB of SRAM and eFlash) Internet of Things (IoT) platforms [51, 126, 136, 148]. TinyML holds the key to making on-board intelligent inferences from unstructured data for time-critical and remote applications, such as aerial robotics [127], underwater navigation [134], picosatellite machine inference [45], and wildlife monitoring [47]. 2.5 billion TinyML platforms are expected to ship in 2030 [4].

An integral component in the TinyML workflow is neural architecture search (NAS) or AutoML, which automatically constructs the most performant neural network (NN) from a set of lightweight ML blocks [79, 81,

Authors' addresses: Swapnil Sayan Saha, swapnilsayan@g.ucla.edu, University of California - Los Angeles, Los Angeles, CA, USA; Sandeep Singh Sandha, sandeep@abacus.ai, Abacus.AI, Seattle, WA, USA; Mohit Aggarwal, mohit@brightnightpower.com, BrightNight, Austin, TX, USA; Brian Wang, wangbri1@g.ucla.edu, University of California - Los Angeles, Los Angeles, CA, USA; Liying Han, liying98@ucla.edu, University of California - Los Angeles, Los Angeles, CA, USA; Julian de Gortari Briseno, julian700@g.ucla.edu, University of California - Los Angeles, Los Angeles, CA, USA; Mani Srivastava, mbs@ucla.edu, University of California - Los Angeles, Los Angeles, CA, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1539-9087/2023/5-ART $15.00
https://doi.org/10.1145/3603171

Just Accepted

94, 162, 167, 173] and connection rules given target platform SRAM, eFlash, energy, and latency constraints [14, 60, 101, 103, 136, 139]. The NAS-generated model is compiled to the target device using TinyML compiler suites [30, 43, 65, 67, 95, 103], which perform operator and inference engine optimizations [27, 42, 103, 177], model compression [76], and code generation [136, 171]. After deployment, periodic fine-tuning of the model accounts for feature distribution shifts using on-device training [24, 100, 129] and federated learning [89, 110]. AutoML is preceded by data acquisition and analytics [131], and feature projection for dimensionality reduction [164] in the TinyML workflow [136].

The first generation efforts in TinyML focused on the exploration (lightweight model blocks), optimization (NAS, AutoML), and integration (compiler suites) of standalone NNs within the device platform constraints [136]. However, IoT applications in the wild need to obey specific rules, physics, and heuristics for provably correct operation, context awareness, and explainability [106, 136, 142, 174]. Examples include:

- A localization ML model regressing position from motion sensor data should not output displacements when rotational artifacts dominate translational movements [134].
- An aerial vehicle should not exceed a certain bank angle to remain stable [44].
- In nurse care settings, certain atomic events (e.g., washing hands) must precede other events (e.g., administering medicine to a patient) to comply with sanitary protocols [174] and not vice-versa.
- Certain spectral features (e.g., peak frequency) in the embedding manifold improve the accuracy and interpretability of wearable human activity recognition models [8].

While ML models have achieved superior performance on unstructured, multimodal, and noisy sensor inputs over human-engineered symbolic techniques, three issues plague the deployment of standalone ML models for context-aware sensor data analytics. *Firstly*, even with large datasets, ML models cannot guarantee the learned feature representations obey all the rules, symmetries, and physics of the underlying system [37, 85, 134, 152]. *Secondly*, the contextual field of ML models (even transformers) is limited to a few minutes, making them unsuitable for high-level reasoning on atomic events that can span several hours (if not days) with spatial and temporal constraints [7, 114, 128, 166, 174]. *Thirdly*, ML models lack transparency and interpretability, with the decision trace (e.g., causation versus correlation) and learned features difficult to understand [63, 109, 114, 121, 147, 176].

Neurosymbolic artificial intelligence (AI) is a potential bridge to connect the interpretability, verifiability, data efficiency, and context awareness of symbolic techniques with the scalability, flexibility, robustness, and performance of NNs [64, 70, 106, 108, 109, 118, 142, 146, 149, 174, 175]. Neurosymbolic AI integrates NNs with expert principles expressed as probabilistic reasoning modules, logical reasoning modules, knowledge graphs, question/answering engines, and constraint satisfaction functions [64, 142]. Concatenation of neural and symbolic reasoning has been successful in a broad spectrum of challenging problems. These include complex event recognition [7, 128, 166, 169, 174], commonsense reasoning [20, 141], visual question answering [109, 176], oceanographic forecasting [35, 58], autonomous driving [71, 150, 157], business management [19, 36], and bioinformatics [5, 92]. Thereby, neurosymbolic AI can enable rich, complex, and intelligent inferences at the extreme edge beyond the perception of atomic events [128, 136, 165]. However, real-time adoption of neurosymbolic frameworks on extremely resource-constrained platforms such as microcontrollers is challenging, as discussed next.

Just Accepted

1.1 Challenges

Given the ultra-resource constraints of TinyML platforms, manually finding the optimal synergy between the hyperparameters of the NN and the symbolic program is arduous and challenging [128]. Deployment of hybrid programs requires AutoML platforms that can perform neurosymbolic optimization.

- **Absence of Platform-Aware AutoML Tools for Neurosymbolic Optimization**: While AutoML and NAS frameworks have been proposed for optimizing NNs for TinyML platforms [14, 60, 83, 101, 103, 123,

124, 139], existing AutoML tools are not designed to perform platform-aware joint optimization of neural and symbolic components [136]. Platform-aware neurosymbolic optimization is necessary to not only fit the highest-performing program within the platform resource constraints but also discover previously unknown high-utility symbolic subroutines as seen in AlphaTensor [57].

- **Fitting Neural and Symbolic Components Within Platform Constraints**: TinyML hardware platforms have tight memory, power, and compute budget [103]. A typical ARM Cortex-M4 microcontroller has only 128 kB of SRAM and 1 MB of eFlash, while a smartphone or cloud server can have RAM and storage in the order of tens of gigabytes and terabytes, respectively [14, 136]. While standalone NNs and standalone symbolic logic are capable of running on TinyML platforms [136], directly porting existing neurosymbolic frameworks on microcontrollers, in-sensor processors [31], and field-programmable gate arrays [83] is not computationally tractable.

1.2 Contributions

We introduce TinyNS, a platform-in-the-loop framework for automatic optimization and deployment of neurosymbolic programs on commodity microcontrollers. Given a search space containing the hyperparameters, logical association rules, and constraints of symbolic and ML (neural or non-neural) model operators, TinyNS automatically finds the best combination of symbolic and ML operators and hyperparameters within the target device memory, latency, and energy constraints. The ML models may be feedforward, residual, or recurrent. The framework provides recipes to map neurosymbolic program atoms from a prototyping language (e.g., Python) to a deployment language (e.g., C). To guarantee program deployability, TinyNS communicates with the target hardware during the optimization process to receive hardware and program runtime metrics instead of relying on proxies. The framework builds on top of a state-of-the-art, gradient-free, black-box Bayesian optimizer [138, 139] designed to optimize non-gradient-friendly and expensive objective functions within a few iterations. Using TinyNS, we showcase several previously unseen applications on microcontrollers. These include physics-aware inertial navigation [134], yielding adversarially robust TinyML models, picking the best model from a zoo of neural and non-neural models [135], and co-optimizing features, Kalman filters and NNs [50]. Our contributions are summarized as follows:

- **Fast, Gradient-Free, and Black-Box Bayesian Optimizer**: We present a fast, parallel, gradient-free, and application-agnostic Bayesian optimizer that can handle non-gradient friendly objectives, categorical and conditional search spaces, and expensive objective functions, all while converging to near-global optima within few iterations [138, 139]. The optimizer forms the basis for our search algorithm.
- **Platform-in-the-Loop Neurosymbolic Architecture Search**: To the best of our knowledge, we are the first to showcase a platform-in-the-loop neurosymbolic architecture search framework for microcontrollers. Our framework automatically synthesizes the most performant neurosymbolic program from a symbolic and ML operator search space within the target platform constraints.
- **Recipes for Deploying Neurosymbolic Programs on Microcontrollers**: Using case studies, we showcase recipes for defining the neurosymbolic program synthesis search space for all five neurosymbolic program categories [142]. Our framework includes parsers that automatically write microcontroller code according to these recipes.
- **Pushing the Boundaries of Handcrafted Neurosymbolic Programs**: We showcase several unseen TinyML applications made possible by joint optimization of neural and symbolic components.

TinyNS is available open-source at: https://github.com/nesl/neurosymbolic-tinyml.

Just Accepted

1.3 Organization

The rest of the paper is organized as follows: Section 2 presents related work and background on porting ML models onto microcontrollers and neurosymbolic AI. Section 3 describes the Bayesian optimization algorithm. Section 4 details the platform-in-the-loop neurosymbolic architecture search space formulation and the recipes for deploying neurosymbolic programs. Afterward, Section 5 presents extensive experimental evaluations of our framework through six case studies. Finally, Section 6 provides concluding remarks and future directions.

2 BACKGROUND AND RELATED WORK

In this section, we first discuss the workflow for porting ML models onto microcontrollers [136], which we modify to realize neurosymbolic TinyML (Section 2.1). Next, we discuss the features of existing NAS frameworks and their shortcomings in performing joint optimization of ML and symbolic operators (Section 2.2). Afterward, we provide a brief overview of the taxonomy, languages, and recent trends in neurosymbolic AI (Section 2.3). Finally, we provide a brief overview of existing Python to microcontroller code parsers (Section 2.4).

2.1 Machine Learning on Microcontrollers

Fig. 1 illustrates the typical workflow for porting ML models to commodity microcontrollers [136]. First, in the *model development phase*, **data engineering** frameworks collect, analyze, clean, label, and store raw sensor data to produce an **application-specific dataset** suitable for training ML models [131]. These frameworks also include tools for targetted augmentation, outlier identification, unit tests, class balancing, and heuristic-assisted automated labeling. The additional tools ensure the trained models are free from bias and shortcuts while generalizing well on edge cases and unseen scenarios [111, 136]. Afterward, optional **feature projection** applies linear methods, non-linear methods, or domain-specific feature extraction for dimensionality reduction while preserving data variance [56]. Linear methods include matrix factorization [46, 99] and principal component analysis (PCA) [12, 34]. Non-linear methods are suitable for minimizing the distance between non-linear high-dimensional input space and the prototype manifold. Common non-linear methods include autoencoders [132], t-distributed stochastic neighbor embedding [163], and kernel PCA [144]. Domain-specific feature extraction applies signal processing, statistical, and time-series functions to the input data depending on the application area [75]. Next, a model backbone is picked from a **zoo of lightweight models** geared towards embedded deployment, based on application and platform specifications. Examples include decision trees and k-nearest neighbor blocks with sparse projection matrices [74, 93], lightweight spatial convolution (e.g., squeeze and excitation modules [81] and depthwise-separable convolution [79]), low rank, stabilized, and quantized recurrent networks [94, 158, 167], temporal convolutional networks [97, 162], and attention condensers [173]. The hyperparameters of the backbone are optimized using **neural architecture search** given a cost function and the hyperparameter search space based on target device constraints [11, 130, 180]. Search space representation includes layer-wise, cell-wise, and hierarchical [130]. Search strategies include reinforcement learning (RL), differentiable NAS, evolutionary algorithms

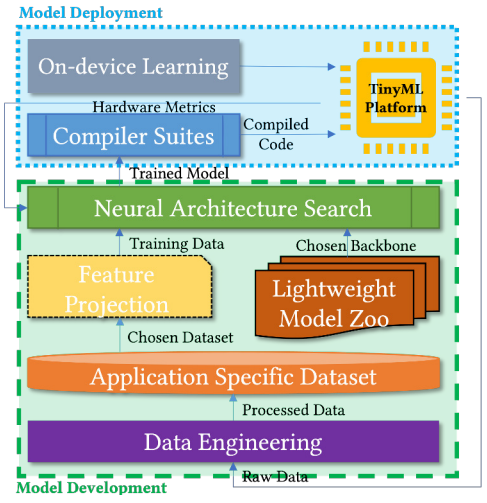

Fig. 1. Closed loop workflow for deploying neural networks on microcontrollers [136].

Just Accepted

Table 1. Qualitative comparison of existing NAS frameworks for microcontrollers versus TinyNS

Method	Search Strategy	Profiler	Search Space	Cost Function Parameters	Inference Engine	Compression Awareness	Open Source
SpArSe [59]	Gradient-driven Bayesian	Analytical	Conv2D (regular, depthwise, downsampled)	Error, SRAM, Flash	uTensor	Pruning (structured, unstructured)	No
MCUNet [102, 103]	Evolutionary (with weight sharing)	Lookup tables, prediction models	Conv2D (elastic)	Error, SRAM, Flash, Latency	TinyEngine [103]	None	No
MicroNets [14]	One-shot DNAS	Analytical	Conv2D (MbNetv2, DS-CNN)	Error, SRAM, Flash, Latency	TFLite Micro [43], CMix-NN [26]	Quantization (sub-byte)	No
μNAS [101]	Evolutionary (no weight sharing)	Analytical	Conv2D (regular, depthwise)	Error, SRAM, Flash, Latency	TFLite Micro [43]	Structured Pruning	Yes
iNAS [112]$^{\wedge}$	Reinforcement Learning	Lookup tables, analytical	Conv2D, tile size, loop order, preservation batch size	Error, Flash, Latency*, Volatile Buffer, Power-Cycle Energy$^{@}$	Accelerated intermittent	Quantization (2 bytes)	Yes
UDC [60]	DNAS with exploration and exploitation	Analytical	Conv2D, sparsity, bitwidth	Error, Flash	Vela NPU	Unstructured pruning, quantization (sub-byte)	No
TinyNS	Gradient-free Bayesian with exploration and exploitation	Real measurements, analytical	Any supported ML operator and symbolic program atoms	Any scalar term	TFLite Micro [43]	Quantization (1 byte)	Yes

$^{\wedge}$ intermittent-aware NAS
* sum of progress preservation, progress recovery, battery recharge, and compute cost
$^{@}$ sum of progress preservation, progress recovery, and compute cost

Just Accepted

(with or without weight sharing), or Bayesian optimization [54]. The hardware metrics can come from real measurements (slowest), lookup tables, prediction models, or analytical proxies (fastest) [54, 130].

The *model deployment phase* begins by generating embedded code to run the best-performing candidate model from the NAS algorithm on the device. This is done by **compiler suites**, some of which provide inference engines for resource management and model graph realization during execution [43, 103]. Compiler suites also perform operator fusion [30, 103], loop transformations [27, 42], data reuse [95], and model compression (pruning, quantization and encoding) [76] to improve memory usage and runtime latency [136]. Afterward, the model file system is flashed onto the microcontroller and occasionally fine-tuned to account for data distribution shifts using on-device training (e.g., transfer learning, incremental training, or continual learning) [24, 100, 129] or federated learning techniques [110].

Variations of the closed loop workflow have been applied to varying applications domains, including image recognition, audio keyword spotting, visual wake words, anomaly detection, navigation, gesture recognition, mHealth, and face recognition [13, 136]. However, these applications assume decisions being made by a standalone ML model, with no symbolic programs (apart from optional feature projection) present on the microcontroller for high-level reasoning [136]. TinyNS modifies the workflow to incorporate symbolic atoms from which programs can be constructed and optimized jointly with the model backbones.

2.2 Neural Architecture Search for Microcontrollers

Table 1 compares prominent NAS frameworks for microcontrollers against TinyNS. In particular, TinyNS adopts a black-box, Bayesian, gradient-free, and platform-in-the-loop search strategy to balance training infrastructure cost, NAS convergence time, guaranteed execution, application support, and neurosymbolic search space characteristics. iNAS [112] uses RL to formulate the NAS multi-objective optimization process as a Markov decision process, with the ability to support complex and discontinuous search spaces with thousands of dimensions [136]. However, RL has a long convergence time (e.g., 5 GPU years) with additional fine-tuning costs [23, 136]. MCUNet [102, 103] and μNAS [101] use evolutionary search on RL search spaces to achieve faster convergence. In particular, MCUNet uses weight-sharing to decouple training from search, mutating, and crossing Pareto-optimal sub-network populations from a "once-for-all" supernetwork [23]. This allows networks for several target hardware to be optimized together. Nevertheless, evolutionary NAS with weight sharing requires GPU infrastructure capable of supernetwork training, suffers from fine-tuning costs, and has a convergence time of 3-8 GPU weeks [23, 136]. MicroNets [14]

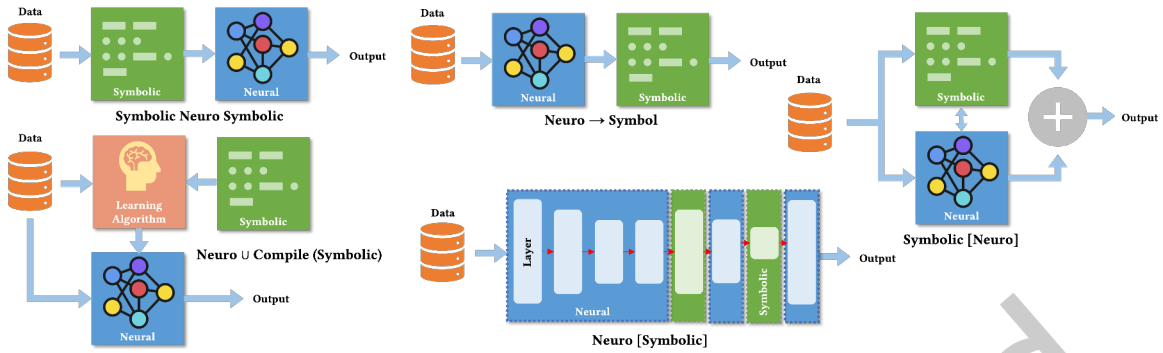

Fig. 2. The five categories of neurosymbolic artificial intelligence [86, 142].

and UDC [60] use differentiable NAS (DNAS), which performs continuous gradient descent relaxation of weights and architectural encodings jointly with approximate gradients via path binarization [25, 104]. This reduces the convergence time to 1-3 GPU weeks [23]. However, DNAS cannot directly model loss contour discontinuities (e.g., categorical or conditional hyperparameters) and have high GPU memory usage owing to the over-parametrized network formulation [112, 136]. Bayesian optimization can handle discontinuous search spaces and cost functions while being executable on commodity GPU workstations [134, 135], further reducing the convergence time to 1-10 GPU days [59]. However, vanilla Bayesian optimization struggles in search spaces beyond a dozen hyperparameters and assumes dense distribution of performant models in the search space [41, 60]. Since neurosymbolic search space dimensions can be orders of magnitude higher than NN search spaces, TinyNS uses Monte Carlo sampling with Upper Confidence Bound (UCB) as the acquisition function instead of the gradient-based approach of SpArSe [59] to perform exploration and exploitation similar to UDC [60]. This prevents TinyNS from being stuck to local optima or evaluating invalid configurations [134, 138] even in complex RL search spaces. Moreover, TinyNS adopts a black-box approach similar to RL or evolutionary NAS. The black-box approach allows optimization of any scalar term beyond model performance and hardware metrics in the cost function and eventually permits the inclusion of both symbolic and any Tensorflow Lite Micro supported ML operators in the search space beyond convolutional operators. Further, TinyNS talks to the target hardware during the NAS process to get resource metrics instead of relying on proxies. Platform-in-the-loop not only guarantees the deployability of the neurosymbolic code, but also allows TinyNS to ignore neurosymbolic programs that induce faults, runtime errors, compilation errors, or flash overflow, saving on convergence time. In fact, TinyNS automatically writes the C code of the neurosymbolic program from Python constructs using proposed neurosymbolic recipes without user intervention.

2.3 Neurosymbolic Artificial Intelligence

Over the past decade, deep learning (DL) has been extensively used to make complex inferences from unstructured, noisy, and high-dimensional data, such as in computer vision, LIDAR point clouds, speech processing, drug discovery, time-series processing and genetics [98]. However, traditional DL is data-hungry even for simple tasks, lacks interpretability and explainability, does not guarantee to follow rules, physics, and constraints, fails on feature distribution shifts, and struggles to learn long-range temporal patterns [37, 63, 64, 121, 147]. The flipside is symbolic AI, which was once the dominant trend of AI research several decades ago before the prevalence of DL [116, 153]. Symbolic programs are data efficient, interpretable, and good at reasoning over the long-term, but suffer when solving NP-hard problems and dealing with spatial and temporal uncertainties in the input data [142].

Just Accepted

Neurosymbolic AI couples DL with symbolic methods to have fast computation time, deal with unstructured data and uncertainty effortlessly, maintain explainable models, and capture complex relations [64, 70, 86, 108, 118, 142]. Neurosymbolic learning is analogous to the two types of human reasoning [84]: type 1 reasoning is fast and intuitive, corresponding to pattern recognition in DL, and type 2 is slower and logical, corresponding to symbolic algorithms and logical reasoning.

2.3.1 Taxonomy of Neurosymbolic AI. Neurosymbolic AI systems are categorized into five groups [86, 142], as illustrated in Fig. 2:

- **Symbolic Neuro Symbolic** or **Neural-after-Symbolic**: This is the most common paradigm [86]. The inputs are symbolic, while the processing is purely neural. The neural component either learns the relations between the symbols or learns to focus on some specific symbols based on needs. Examples include inference over human-engineered features [87] and graph NN inference with pre-processed graph nodes [143]. While this technique allows applying human-engineered functions on the inputs, the synergy between neural and symbolic components is weak, with no high-level reasoning possible over the outputs.
- **Neuro→Symbol** or **Symbolic-after-Neural**: In this approach, NNs process raw inputs and output structured data, which are fed to symbolic programs for further reasoning. Examples include DUA [114] and DeepProbLog [108]. In DUA, a symbolic meta-policy learning module with common sense background knowledge combines primitive actions from a deep RL agent. In DeepProbLog, NNs are trained to output probabilistic predicates, which are fed to a logic program to evaluate user-defined logic rules. The technique allows the flow of gradients from the symbolic output through the network but suffers from the high compute cost of the reasoning module.
- **Neuro ∪ Compile (Symbolic)** or **Symbolically-constrained Neural**: This technique adds a symbolic component to the learning process of a neural model to follow constraints, norms, or rules, which are compiled away during training [96]. An example includes Pylon [3], where user-defined constraints on the output are converted to an additional loss added to the traditional error cost. While constraints are simple to express using this method, the network is not guaranteed to satisfy hard thresholds.
- **Symbolic[Neuro]** or **Neurosymbolic Aggregation**: In this method, a neural model and symbolic program aggregate their results to achieve more robust inference. The neural component models errors resulting from uncertainties of the symbolic program, or the symbolic program forces the NN to follow some constraints or rules. In STLnet [106], a neural student model learns to predict succeeding output sequences by learning temporal logic relations, while a symbolic teacher model generates an output sequence most similar to that prediction within the given relational constraints.
- **Neuro[Symbolic]** or **Neurally-accelerated Symbolic** or **Symbolically-structured Neural**: This is the preferred neurosymbolic paradigm [86], where the NN architecture is generated using (or has layers embedded with) symbolic reasoning. A neural model replaces slow or non-differentiable symbolic programs while keeping the latter's functionality. Examples include logic Tensor Networks [146], which generates a first-order logic language into TensorFlow computational graphs. Pix2rule [33] embeds a differentiable linear layer in a deep NN, which is biased to capture the semantics of AND and OR to extract spatial symbolic rules. Neuroplex [174] adopts a knowledge distillation approach to train a neural model that can replace the logic reasoner for complex event pattern detection. While allowing pure type 2 reasoning, this method may include special ML operators unsupported on TinyML hardware.

2.3.2 Neurosymbolic Language Tools. Neurosymbolic language tools synthesize programs from user-defined rules. DeepProbLog [108] is a probabilistic logic programming language where users can define logical rules and network architectures. The symbolic reasoning module is differentiable, allowing backpropagation of target labels at the output of the logic program through the NN. Pylon [3] is a PyTorch framework that learns deep

Just Accepted

NNs with constraints. It automatically converts constraints defined by users into a constraint loss, and the NN is trained using the summation of this constraint loss and a regular loss function. Gen [40] is a probabilistic programming language designed for general-purpose neurosymbolic program synthesis. It can build generative models to represent data-generating processes, supports flexible DL and differentiable programming, and can make probabilistic inferences.

2.3.3 Recent Trends in Neurosymbolic Artificial Intelligence. Recent research in neurosymbolic AI focuses on handling domain shifts, performing error correction, increasing data efficiency, and improving the interpretability of ML systems [64, 142]. Symbolic background knowledge allows extrapolation when dealing with input distribution different from training data [105]. Error correction designs robust ML systems enabling streamlined recovery from wrong outputs without retraining on new data [18]. Symbolic reasoning allows NNs to be trainable with less data [142]. Improving the interpretability of ML systems makes NN decisions more transparent and explainable [115]. Unfortunately, the deployment of neurosymbolic programs on IoT platforms or for real-time inference has received little attention. μCEP [128] is the only framework that allows complex event processing on neural outputs using logical rules on commodity microcontrollers. However, μCEP is hard-coded for a single application (complex activity detection), few network architectures (fully-connected and convolutional), and a specific neurosymbolic AI category (Neuro→Symbol), with no notion of co-optimization of neural and symbolic components or platform-awareness. In contrast, our framework allows platform-aware automatic co-design of ML (neural or non-neural) and symbolic components regardless of application, choosing the best synergy of ML operators and symbolic hyperparameters within the tight resource bounds of TinyML platforms.

2.4 Python to Microcontroller Code Parsers

Parsers automate the porting of code written in a high-level language (e.g., Python) to a deployment-time language (e.g., C). There are two kinds of parsers relevant to this work.

2.4.1 TinyML Compiler Suites. These software suites take an ML model trained in a high-level ML framework to generate embedded code and perform operator optimizations, model compression, and inference engine optimizations. The embedded file system is then flashed onto the microcontroller for inference. Some of these frameworks provide memory planners, intermittent computing, runtime interpreters, and operator resolver functionalities in the form of inference engines [136]. The frameworks use a template file system to map tensor manipulation operations, logging, and input/output handling from the high-level model schema to objects. TensorFlow Lite Micro (TFLM) [43], uTVM [30], Microsoft EdgeML [68, 69, 74, 93, 94, 133], CMSIS-NN [95], and EON compiler [80] are popular frameworks that automatically parse TensorFlow [1] and PyTorch [120] neural networks to C code mainly for deploying on ARM Cortex-M processors. STM32Cube.AI [1], Eloquent ML [2], and Sklearn Porter [3] parse support vector machines, decision trees, naive Bayes, k-nearest neighbors, random forest, XGBoost, and regressors from Scikit-Learn [122] to C [136]. For model parsing, we adopt and modify TFLM for parsing neural networks to C. *Firstly,* we add scripts to check for use of unsupported ML operators and detect compilation and memory overflow faults during neurosymbolic program optimization by talking to the target hardware. *Secondly,* our parser can automatically modify the TFLM file system to invoke only the necessary operators, take care of quantization and dequantization, assign appropriate arena and buffer sizes, and place `.c` and `.h` files in the appropriate directories. *Thirdly*, our parser invokes the embedded C compiler directly from Python and flashes the compiled program on the target hardware.

[1] https://www.st.com/en/embedded-software/x-cube-ai.html
[2] https://eloquentarduino.com/
[3] https://github.com/nok/sklearn-porter

Just Accepted

2.4.2 General Purpose Parsers. These parsers convert general-purpose Python code to C. Shed Skin [4], Nuitka [5], Pyrex [6], Cython [16], SWIG [15], and BoostPython [90] are popular Python-to-C source-to-source translators. Most of these frameworks convert implicitly statically typed Python programs to C/C++, write boilerplate code using interface files through a shared library, perform compiler optimizations, and transmute data structures and types. However, these parsers lack support for runtime-interpreted program aspects and functions, cross-compilation, standard library, and unrestricted function definitions. Recently, large conversational language models such as ChatGPT [7] are being used as code translation assistants [172]. The generated code is not error-free most of the time but helps save manual code conversion time for programmers. MicroPython [161] and Zerynth [8] are software implementations of Python written in C for 32-bit microcontrollers. MicroPython supports features in the most popular Python modules, allows code portability due to the use of the hardware abstraction layer, offers modular programming, provides access to low-level hardware, and immediately executes commands. Similar to TFLM, MicroPython includes a runtime interpreter to interpret the bytecode. Unfortunately, MicroPython is $10^1 - 10^2$ orders of magnitude slower than pure C/C++ [82], preventing its adoption in time-critical systems. In contrast, instead of providing direct source-to-source translation, TinyNS provides recipes to map the symbolic component for 4 of the 5 neurosymbolic paradigms from Python to pure C/C++. We assume the user has implemented the symbolic code in C either manually or using an existing source-to-source translator, and instead focuses on activating and passing arguments to the C objects from Python. For *symbolic neuro symbolic*, we use the concept of an array of over-parametrized function pointers selected using a binary mask. For *neuro* $\rightarrow$ *symbol*, we use ANTLR to port program trees from Python to C. For *neuro* $\cup$ *compile (symbolic)*, a physics extraction function is activated. For *symbolic [neuro]*, the function arguments are sent to a Kalman update step. The recipes call for the use of CMSIS libraries for mathematical, tensor, and signal processing operations.

3 MANGO: FAST, PARALLEL AND GRADIENT-FREE BAYESIAN OPTIMIZER

TinyNS adopts *Mango* [138, 139], which is an efficient realization of Bayesian optimization. Bayesian optimization provides a state-of-the-art approach to optimize expensive objective functions in a few iterations, approximated by a surrogate model.

3.1 Surrogate Model

Typical surrogate models used in Bayesian optimization libraries are Gaussian processes (GP), tree-structured Parzen estimators, and random forests. Among the available surrogate models, *Mango* uses the GP surrogate ($\mathcal{GP}$) over the search space (Ω) due to its ability to provide a tractable assessment of prediction uncertainty incorporating the effect of data scarcity [154]. The GP is a non-parametric machine learning model specified using a mean (μ) and a kernel function (k).

$$\hat{f}(\Omega) \sim \mathcal{GP}(\mu(\Omega), k(\Omega, \Omega')) \tag{1}$$

Vanilla GP models work well on continuous search spaces but struggle to deal with the discontinuity in the search spaces induced by categorical, mixed, and hierarchical search spaces. Naive rounding or one-hot encoding causes the GP to get stuck to the same candidate model. Thereby, Mango adopts the solution proposed by Garrido-Merchan *et al.* [66], which modifies the GP covariance function to account for regions in the search space where the objective function becomes constant due to one-hot encoding or rounding inside the objective function evaluator wrapper. The constant behavior cannot be modeled by GP. We use a transformation of the

[4] https://shedskin.github.io/
[5] https://www.nuitka.net/
[6] https://www.csse.canterbury.ac.nz/greg.ewing/python/Pyrex/
[7] https://openai.com/blog/chatgpt
[8] https://zerynth.com/blog/python-and-c-hybrid-programming-on-a-microcontroller-with-zerynth/

Just Accepted

input variables that rounds real-valued hyperparameters and performs one-hot encoding of categorical variables, causing the Cartesian distance between the sample points with the same configuration becoming 0. This allows the GP to indirectly model the expected constant behavior, as the transformation enforces maximum correlation between the function evaluations at the sample points with the same configuration under the GP.

3.2 Acquisition Function

The exploration-exploitation is handled using the UCB [155, 156] as the acquisition function. In UCB the next sample (Ω_t) at iteration t is sampled from the search space (Ω) using the predicted mean (μ_{t-1}) and the corresponding variance (σ^2_{t-1}) at iteration $t-1$. The exploration factor (β) balances the contributions of the mean and variance.

$$\Omega_t = \arg\max_{\Omega}(\mu_{t-1}(\Omega) + \beta^{0.5}\sigma_{t-1}(\Omega)) \tag{2}$$

The first term (mean) in the acquisition function refers to the goodness of the current sampled point (exploitation), while the second term refers to the uncertainty of the sampled point (exploration). Mango adopts UCB because of four reasons. *Firstly*, UCB is robust to uncertainty and noise in the function evaluations without pre-processing. . *Secondly,* UCB allows efficient sampling for cases where picking a suboptimal point may cause a time-consuming and expensive function evaluation. *Thirdly*, UCB balances exploration and exploitation by sampling points that are not just likely to improve the final score (exploitation), but also sampling points that have high uncertainty (exploration). This not only prevents the optimizer from getting stuck in a local optimum but also provides both a coarse and a fine-grained view of the objective plane, allowing the score to achieve theoretical optimal values at the boundary of violating deployability constraints. *Lastly*, UCB uses of an adaptive β with theoretical convergence guarantees within 90% of the optimal value [48, 155, 156]. β is heuristically decided based on the complexity of the search space (domain size) $|\Omega|$, the current iteration count t, and the variance (uncertainty) $\sigma^2_{t-1}(\Omega)$ at iteration $t-1$.

$$\beta = \alpha \cdot \exp(2 \cdot C), \quad \alpha = \sqrt{2\log(0.6 \cdot |\Omega| \cdot t^2 \cdot \pi^2)}, \quad C = \frac{8}{\log(1 + \frac{1}{\delta + \sigma_{t-1}(\Omega)})}, \quad \delta = 1e^{-6} \tag{3}$$

Firstly, if the search space is bigger, α will increase logarithmically, leading to a bigger β. This will cause the acquisition function to be dominated by exploration. *Secondly,* as the search progresses, α increases logarithmically. This impels the acquisition function to be exploration dominant in the later iterations. *Thirdly*, sample points near already explored regions will return a lower value of $\sigma^2_{t-1}(\Omega)$, leading to a lower value of β. *Lastly*, if a region is invalid or bad, then $\mu_{t-1}(\Omega)$ will be higher, causing the acquisition function to be dominated by exploration. If a region is valid or good or near the theoretical optimal boundary, then $\mu_{t-1}(\Omega)$ will be lower, causing the acquisition function to be dominated by exploitation. The four factors cause Mango to perform what is known as sampling to find the boundaries in the objective plane. t ensures that exploration never stops in case Mango has not found a "hidden" region where global optima may reside. However, exploration dependent on t is logarithmic, leading to only a small increase in the β with each passing iteration. $\sigma^2_{t-1}(\Omega)$ ensures that as more regions of the objective plane are explored, Mango moves from primarily exploration-driven to exploitation-driven sampling, which allows Mango to perform fine-grained sampling at later iterations. $\mu_{t-1}(\Omega)$ ensures that this fine-grained sampling is being performed at the boundaries close to the theoretical optimal value with 90% probability. The entire formulation makes Mango explore all unexplored boundaries (coarse-grained sampling), and then find the points close to the theoretical optimal value (fine-grained sampling).

3.3 Handling Mixed Search Spaces

Traditionally, gradient-driven optimizers (e.g., GpyOpt [9] and Skopt [10]) are used to find the next promising sample, such as in SpArSe [59]. Sandha *et al.* [138, 139] showed that gradient-driven optimization in complex search

Just Accepted

spaces having discrete or categorical values can provide sub-optimal solutions by evaluating gradients at invalid configurations of the search space. *Mango* realizes a gradient-free optimizer for handling non-gradient-friendly values. Mango directly supports discrete integer values and continuous values and converts pure categorical to the one-hot encoding. However, this comes with the challenge that the decision boundary of the acquisition function becomes discontinuous due to the discrete values. Further, one-hot encoding of categorical variables increases the dimensionality of the search. To handle the discontinuous decision boundary, Mango adopts a gradient-free optimizer that doesn't assume the continuity of gradient in the acquisition function search space. This is based on the Monte Carlo optimization of the acquisition function. Since the evaluation of the acquisition function is very cheap, this approach is scalable to search decision boundaries extensively to parallelly select the next optimal points. The acquisition function is evaluated at thousands of valid samples in the search space; thus, there is no mismatch between the proposed and actual evaluations. This approach also works directly for the one-hot encoded spaces by doing evaluations only at the valid regions of the one-hot encoding without sampling the intermediate regions between 1 and 0 where no valid real sample exists. It is to be noted that in a gradient driven approach, the optimal point is finally converted to the correct sample either by rounding-off that can degrade the search results, which is not the case in Mango. This sampling-based approach also reduces the computational complexity [139] of the optimizer compared to the gradient-based methods used in other Bayesian optimization libraries [9, 10, 59].

To reduce the search space complexity even further, TinyNS proposes the use of slider matrices, enumerated trees, and ordinal masks. Instead of exposing Mango directly to the heterogeneous variables, for high-dimensional search spaces, TinyNS exposes Mango to the normalized slider matrix, inspired by the wrapper-based approach proposed in Garrido-Merchan*et al.* [66]. The slider matrix is a continuous formulation of the mixed parameter space normalized between 0 and 1. The one-hot encoding or rounding is performed inside the objective function evaluator wrapper as proposed in [66] via a mapping that maps the terms in the slider matrix to the mixed parameter space. For even more complicated search spaces, TinyNS uses tree enumeration algorithms to generate program tree candidates and exposes TinyNS to an ordinal mask that selects one of the trees.

3.4 Parallelization

Another challenge in solving Eq. 2 is parallelizing the sequential search process, selecting a batch of values to ensure exploration or diversity in the batch. The straightforward approach of ranking the search choices according to the acquisition function and then selecting the top picks is sub-optimal due to limited exploration [48]. To enable parallel search, *Mango* provides a *clustering search* algorithm on the samples drawn from the acquisition function. The clustering search selects promising domain samples from different clusters based on their distance in the search space. The different clusters are far from each other in the hyperparameter space to enable exploration or diversity. The number of clusters is equal to the batch size and is flexible.

3.5 Addition to Mango

TinyNS expands the state-of-the-art Bayesian optimizer to perform neurosymbolic architecture search in three ways. *Firstly,* while Mango internally handles categorical and continuous variables, the optimizer alone cannot deal with complex neurosymbolic search spaces on its own. We provide recipes to show how Mango can deal with neurosymbolic search spaces through the intelligent use of slider matrices, Boolean masks, and enumerated trees. This significantly increases the types of problems Mango can handle. *Secondly,* to prevent wasting valuable GPU hours and improve convergence time, we use a guided optimization strategy. Specifically, we do not train programs that violate deployability constraints or induce faults. We penalize Mango by a constant number when it makes wrong choices. Yet, we design the optimization function in such a way that Mango is still able to find the boundaries in the objective plane even in complex search spaces and achieve near-optimal results. *Thirdly*, we

Just Accepted

make Mango platform-aware by allowing it to talk to the target hardware during deployment time. This allows guaranteed program deployment and accurate profiling. We discuss these additions in more detail in Section 4.

3.6 Evaluation: Parallel Search in Mango

We visualize the parallel search enabled by *Mango* in Fig. 3 (Left). Four iterations of the *clustering search* algorithm are shown for a 1-D function having multiple optimal points. The ground-truth function is represented by *objective*. The *samples* are the points that have been evaluated, and hence the true objective function values are known. A batch size of 3 is used, representing the parallel evaluation of 3 samples in each iteration. The *Surrogate function* shows the internal approximation of the ground-truth objective based on the evaluated samples. The *acquisition function* is based on the UCB. The three *clusters* created in different regions of the *acquisition function* are shown. The *next sampling locations* represent the points selected from each cluster for evaluation in the next iteration. We observe that the ground-truth max optimal is found by Mango in the fourth iteration, which occurs at -1.0 and has a value of 4.72.

3.7 Evaluation: Comparison Against Other Bayesian Optimizers

We compare *Mango* for hyperparameter tuning with existing state-of-the-art Bayesian optimization libraries using the multiple criteria methodology proposed by Dewancker *et al.* [49]. Specifically, we measure the performance of an optimizer by considering the solution's proximity to the optimal point (accuracy) and the number of iterations required to reach the optima (speed). We compared the performance for hyperparameter tuning of three ML classifiers: Xgboost, K-Nearest Neighbor (KNN), Support Vector Machines (SVM) to maximize the 3-way cross-validation accuracy for the iris plants dataset, wine recognition dataset, and breast cancer Wisconsin (diagnostic) dataset taken from Scikit-learn [122], i.e., a total of 9 tuning tasks (three classifiers trained using

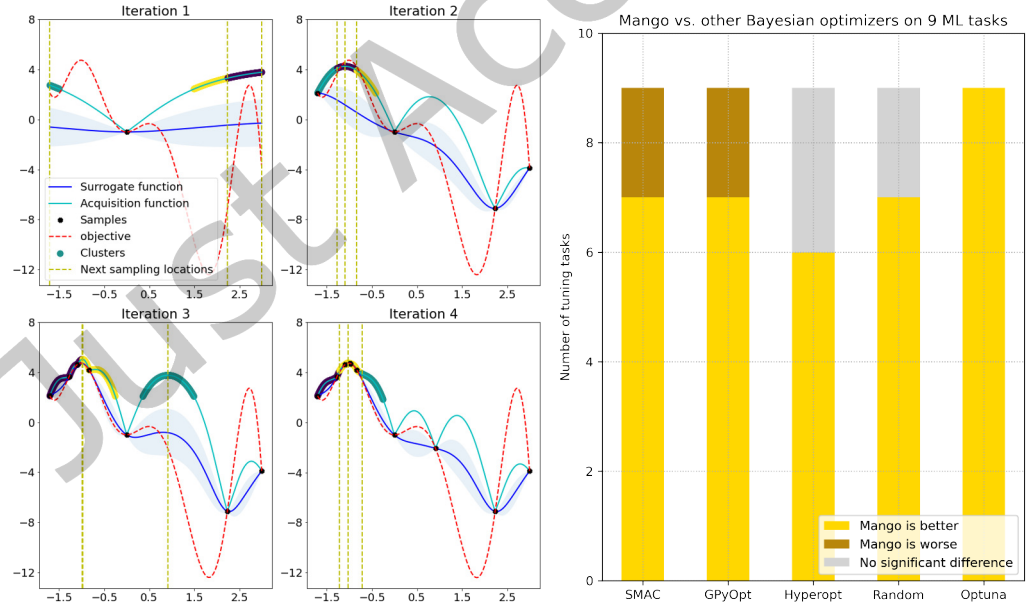

Fig. 3. (Left) Visualizing parallel optimization in Mango. (Right) Sequential optimization performance of Mango on 9 ML classification tasks versus 5 other state-of-the-art Bayesian optimizers.

Just Accepted

three datasets). The search space includes continuous, integer, and categorical hyperparameters with the exact definitions available [137]. We tune each classifier for 80 iterations and repeat each tuning experiment 30 times. Results are shown in Fig. 3 (Right). *Mango* performs better than all other libraries in 6 or more tasks out of 9 in hyperparameter tuning for classifiers with mixed hyperparameters (continuous, integer, and categorical) spaces. Specifically, Mango outperforms HyperOpt (TPE surrogate), SMAC (random forest surrogate), Optuna (TPE surrogate), and GPyOpT (vanilla GP surrogate). Overall, Mango offers state-of-the-art optimization capabilities for handling complex search spaces.

4 PLATFORM-AWARE NEUROSYMBOLIC OPTIMIZATION

TinyNS treats neurosymbolic architecture search as nonlinear programming [17] over the search space $\mathbf{\Omega}$:

$$\min \mathbf{f}(\Omega), \quad \text{s.t.} \quad \mathbf{f}(\Omega) \leq \mathbf{b} \tag{4}$$

where

$$\mathbf{f}(\cdot) = \lambda_k \sum_n g_k(\Omega), \quad \Omega = \{\{V, E\}, [\theta_m, m, w], [\theta_s, s, u]\}, \quad \sum_n \lambda_k = 1, \quad k \in [1, n] \tag{5}$$

Ω contains both ML components and symbolic components. The ML components include the ML hyperparameters θ_m, trainable ML parameters w (e.g., NN weights and biases), and ML operators m (e.g., convolution, pooling, support vector kernel, fully connected, etc.). The ML operators may be feedforward, residual, or recurrent. The symbolic components include the symbolic hyperparameters θ_s, numerical parameters to be optimized u (e.g., Kalman filter gain), and symbolic program atoms s (e.g., predicates, terms, features, etc.). Candidate neurosymbolic programs constructed from $\mathbf{\Omega}$ can be thought of as directed acyclic graphs $q^{\Omega}(\mathbf{X})$ with edges E, vertices V and input tensor $\mathbf{X}$. The goal is to find a neurosymbolic program that satisfies the aggregate constraint $\mathbf{f}(\Omega) \leq \mathbf{b}$. In other words, the objective function seeks a Pareto-frontier configuration Ω^* under competing objectives [59] such that:

$$\mathbf{f}_k(\Omega^*) <= \mathbf{f}_k(\Omega) \;\; \forall k, \Omega \;\; \wedge \exists j : \mathbf{f}_j(\Omega^*) < \mathbf{f}_j(\Omega) \;\; \forall \Omega \neq \Omega^* \tag{6}$$

The aggregate constraint function $\mathbf{f}(\cdot)$ is a linear combination of individual objectives $g(\cdot)$ weighted by random scalarizers λ. Let $\mathcal{A}$ be a complete Boolean algebra, ω_ω be the ordinal set, and $\mathbb{A}$ be a fixed set of names. Then, $g(\cdot)$ and $\mathbf{\Omega}$ have the following properties:

- $d \vee \neg d, \quad \underbrace{d = (\exists g_k(\cdot) \wedge \exists c \in \Omega) \Rightarrow \left(\nexists \lim_{x \to c} g_k(x) \vee \nexists g(c) \vee \lim_{x \to c} g_k(x) \neq g(c)\right)}_{\text{discontinuity condition}}$
- $\exists z \in \mathbf{\Omega} \Rightarrow \Big[\underbrace{z \in \mathbb{R}}_{\substack{\text{continuous,}\\ \text{numeric}}} \vee \underbrace{[z \in \mathbf{B}, \mathbf{B} \subseteq \mathbb{R}, f : \mathbf{B} \to \mathbb{N}]}_{\text{discrete, numeric}} \vee \underbrace{[((\forall q \in \bar{q}) \pi q = q) \Rightarrow \pi \cdot z = z, \pi \in \text{Perm } \mathbb{A}]}_{\text{categorical, nominal}} \vee \underbrace{z \in \omega_\omega}_{\substack{\text{categorical,}\\ \text{ordinal}}} \Big]$
- $\underbrace{\exists x | a \in \mathbf{X}, x \in \mathbf{\Omega}, a, b \in \mathcal{A} \Rightarrow \big[(a = b \Rightarrow x|a = y|b) \wedge (x|b = y|b \Rightarrow x|a = y|a) \wedge \big(\forall (a_i)_{i \in I} \in \mathcal{A}, \forall (x_i)_{i \in I} \in \boldsymbol{X}, \forall i \in I \Rightarrow \exists! x (x|a_i = x_i|a_i)\big)\big]}_{\text{conditional inclusion}}$

The base formulation of Eq. 4 and Eq. 5 is given as:

$$\min f_{\text{opt}}, \quad f_{\text{opt}} = \lambda_1 f_{\text{error}}(\mathbf{\Omega}) + \lambda_2 f_{\text{flash}}(\mathbf{\Omega}) + \lambda_3 f_{\text{SRAM}}(\mathbf{\Omega}) + \lambda_4 f_{\text{latency}}(\mathbf{\Omega}) \tag{7}$$

Just Accepted

where,

$$f_{\text{flash}}(\Omega) = \begin{cases} \gamma_f \Leftrightarrow \left(|\gamma_f| < 1 \wedge \underbrace{\epsilon_{\text{flag}} = 0}_{\text{fault flag}} \right), \quad \gamma_f = \left(\underbrace{-\frac{||h_{\text{FB}}(w, \{V, E\})||_0}{\text{flash}_{\text{max}}}}_{\text{model proxy}} + \underbrace{\xi_f}_{\substack{\text{slack for} \\ \text{symbolic}}} \vee \underbrace{-\frac{\text{Compiler-reported flash}}{\text{flash}_{\text{max}}}}_{\text{real measurement}} \right) \\ \alpha_f, \quad \alpha_f \gg \text{flash}_{\text{max}} \end{cases} \tag{8}$$

$$f_{\text{SRAM}}(\Omega) = \begin{cases} \gamma_s \Leftrightarrow \left(|\gamma_s| < 1 \wedge \underbrace{\epsilon_{\text{flag}} = 0}_{\text{fault flag}} \right), \quad \gamma_s = \left(\underbrace{-\frac{\max_{l \in [1,L]} \{||x_l||_0 + ||a_l||_0\}}{\text{SRAM}_{\text{max}}}}_{\text{model proxy}} + \underbrace{\xi_s}_{\substack{\text{slack for} \\ \text{symbolic}}} \vee \underbrace{-\frac{\text{Compiler-reported SRAM}}{\text{SRAM}_{\text{max}}}}_{\text{real measurement}} \right) \\ \alpha_s, \alpha_s \gg \text{SRAM}_{\text{max}} \end{cases} \tag{9}$$

$$f_{\text{latency}}(\Omega) = \begin{cases} \underbrace{\frac{\text{FLOPS}}{\text{FLOPS}_{\text{target}}}}_{\text{model proxy}} \vee \underbrace{\frac{\text{RTOS-reported latency}}{\text{latency}_{\text{target}}}}_{\text{real measurement}} \Leftrightarrow \underbrace{\epsilon_{\text{flag}} = 0}_{\text{fault flag}} \\ \alpha_l, \alpha_l \gg \text{FLOPS}_{\text{target}} \vee \text{latency}_{\text{target}} \end{cases} \tag{10}$$

The goal of the base formulation is to find a Pareto-optimal neurosymbolic program with the lowest possible runtime latency but maximizes the device's full SRAM and flash capacity without inducing overflow or faults. The performance of a candidate neurosymbolic program on the validation dataset at each iteration in the search provides $f_{\text{error}}(\Omega)$. When the target hardware is connected to the training server, the compiler provides the program SRAM consumption $f_{\text{SRAM}}(\Omega)$ and flash consumption $f_{\text{flash}}(\Omega)$, while the onboard real-time operating system (RTOS) reports the program runtime latency $f_{\text{latency}}(\Omega)$. The measurements are conditioned on the absence of faults, indicated by ϵ_{flag}. Based on prior work [134, 135], we set λ_1 to 1.0, λ_2 to 0.01, λ_3 to 0.01, and λ_4 to 0.05. TinyNS has the following *fault detection* capabilities:

- Flash, SRAM, or model arena buffer overflow (the program is too big to fit).
- Use of unsupported ML operators.
- Compilation errors.
- Runtime RTOS faults.

If $\epsilon_{\text{flag}} = 0$, the hardware metrics are normalized by the device SRAM and flash capacities (SRAM_{max}, $\text{flash}_{\text{max}}$), and target latency ($\text{latency}_{\text{target}}$) to a common scale. If $\epsilon_{\text{flag}} \neq 0$, the hardware metrics are set to a value much larger than the device capacity or target latency. We set $\alpha_f = 125$, $\alpha_s = 125$, $\alpha_l = 50$, resulting in f_{opt} being 5.0 whenever deployability constraints are violated. This policy, called *hard thresholding*, achieves *full device capability exploitation*. Since violating deployability constraints always returns an f_{opt} of 5, after sufficient iterations, TinyNS can observe and exploit the small but valid linear region of SRAM and flash usage between -1 and 0 (γ_f and γ_s are valid between -1 and 0), striving to move γ_f and γ_s towards -1. Yet, TinyNS is aware that certain choices of ML operators and symbolic atoms would make γ_f and γ_s more negative (hence the objective should ideally be minimized even further) but are invalid. In other words, the optimizer is penalized by a large constant number when it picks candidate models that do not fit within the device or induce faults and instead encourages the acquisition function to not pick too many points in the regime where the violation may occur. After sampling sufficient points in the small but valid linear region and the invalid regions, the surrogate function smooths out sufficiently to match the linear region in the objective plane where the accuracy improvement is proportional to memory usage without inducing faults. Hard thresholding is possible thanks to the adoption of parallel version [48] of GP-UCB [155, 156]. During exploitation, GP-UCB picks candidate models which are likely to minimize f_{opt}. The sample points in this phase will be close to one or more of the "successful" points in the

Just Accepted

linear/valid region found during previous iterations. Exploitation, thereby, provides a finer-grained view of the objective plane. During exploration, GP-UCB will either pick points in the valid or invalid region to make sure the optimizer is not stuck in local optima. Exploration, thereby, provides a coarse-grained view of the objective plane. With sufficient iterations, the acquisition function moves from being exploration driven to exploitation driven, converging near theoretical optimal value at the boundary of violating deployability constraints. The parallel implementation allows the optimizer to have access to more “batches of sample points" at each iteration. The policy of hard thresholding is not possible to implement with gradient-based optimizers due to discontinuous penalization. For those optimizers, one would have to train the model to get the accuracy even if GPU hours are wasted, calculate the memory usage, and penalize in a continuous fashion proportional to the memory usage (referred to as coupling of deployability and performances). Since we do not train a candidate model once deployability constraints have been violated, hard thresholding (combined with fault detection) also prevents TinyNS from training a candidate model that does not satisfy all the constraints, saving valuable GPU hours by as much as 50% over gradient-based optimizers.

Note that SpArSe [59] treats λ as a *super-hyperparameter* bring drawn from a random distribution at each iteration. However, realizing λ as a *super-hyperparameter* in complex neurosymbolic search spaces with a gradient-free and black-box optimizer is challenging as compared to the gradient-based optimizer in SpArSe. For the same program candidate, different values of λ will yield different values of f_{opt} at each iteration, resulting in a large number of iterations needed to achieve acceptable performance. We are aware that our choices of λ and α may not provide the most optimal neurosymbolic program for each application, but, as we will showcase, are able to guarantee high-utility and deployable neurosymbolic programs that significantly outperform the state-of-the-art.

When the target device is absent, TinyNS relies on well-known analytical proxies to provide device resource usage estimates. $f_{\text{flash}}(\boldsymbol{\Omega})$ is given by the size of the flatbuffer model schema $h_{\text{FB}}(\cdot)$ [43]. $f_{\text{SRAM}}(\boldsymbol{\Omega})$ is given by the standard NN SRAM usage model, with intermediate layer-wise activation maps and tensors stored in the SRAM [59]. $f_{\text{latency}}(\boldsymbol{\Omega})$ is provided by the FLOPS count [14]. Assuming the ML component dominates resource usage over symbolic components, a static slack constant ξ is added to the SRAM and flash proxies to account for SRAM and flash usage by the symbolic program. There are, however, several issues with this profiling approach:

- Proxies are inaccurate and do not work for a wide variety of ML operators (e.g., well-known proxies were developed only for convolutional models) [134, 135]. Proxies do not even exist for symbolic programs.
- Model proxies tend to overestimate device capabilities without considering overhead from symbolic programs, runtime inference engines, RTOS, or data stacks [134, 135].
- Proxies cannot capture all the faults that the platform-in-the-loop approach can. Hence, the correctness of the neurosymbolic program is not guaranteed.
- Proxies cannot take into account compiler suite optimizations at the execution level, often yielding sub-optimal models compared to the platform-in-the-loop approach.

For each candidate neurosymbolic program, TinyNS automatically writes embedded C code for microcontrollers from Python constructs using *parsers*. The recipes used by the parsers are discussed next.

4.1 Symbolic Neuro Symbolic

Problem Formulation (Symbolic). Consider a vector of independent domain-engineered functions $\mathbf{z}(\cdot)$ constructed from s in $\boldsymbol{\Omega}$ that operate on $\mathbf{X}$. During the search process, each function in $\mathbf{z}(\cdot)$ can be accessed through a binary mask c, signifying the activation and deactivation of a collection of elements of $\mathbf{z}(\cdot)$.

$$\mathbf{X}_i^{\text{feat}} = z_i^{\mathbf{U}_i}(\mathbf{X}) \Leftrightarrow c_i = 1, \quad i \in [1, n], \quad c_i \in 0 \vee 1 \tag{11}$$

$\mathbf{U}$ is a 2D hyperparameter data structure for $\mathbf{z}(\cdot)$. i^{th} row of $\mathbf{U}$ correspond to the hyperparameters for z_i. The number of columns of $\mathbf{U}$ is the number of optimization hyperparameters for that z_i which takes the maximum

Just Accepted

number of hyperparameter arguments, e. Each element in $\mathbf{U}$ corresponds to the range of possible floating point numbers in the search space for the $(i, j)^{\text{th}}$ hyperparameters, expressed as a list. Boolean hyperparameters are converted to (0.0, 1.0), and nominal variables are converted to ordinal choices (e.g, 1.0, 2.0, 3.0, 4.0, 5.0). The length of each element in $\mathbf{U}$ varies.

$$\mathbf{U} = \begin{bmatrix} [\alpha_1^{1,1}, \alpha_2^{1,1}, ..., \alpha_{\gamma_1^1}^{1,1}] & [\alpha_1^{1,2}, \alpha_2^{1,2}, ..., \alpha_{\gamma_2^1}^{1,2}] & ... & [\alpha_1^{1,e}, \alpha_2^{1,e}, ..., \alpha_{\gamma_e^1}^{1,e}] \\ [\alpha_1^{2,1}, \alpha_2^{2,1}, ..., \alpha_{\gamma_1^2}^{2,1}] & [\alpha_1^{2,2}, \alpha_2^{2,2}, ..., \alpha_{\gamma_2^2}^{2,2}] & ... & [\alpha_1^{2,e}, \alpha_2^{2,e}, ..., \alpha_{\gamma_e^2}^{2,e}] \\ . & . & ... & . \\ . & . & ... & . \\ . & . & ... & . \\ [\alpha_1^{n,1}, \alpha_2^{n,1}, ..., \alpha_{\gamma_1^n}^{n,1}] & [\alpha_1^{n,2}, \alpha_2^{n,2}, ..., \alpha_{\gamma_2^n}^{n,2}] & ... & [\alpha_1^{n,e}, \alpha_2^{n,e}, ..., \alpha_{\gamma_e^n}^{n,e}] \end{bmatrix} \tag{12}$$

An example of $\mathbf{U}$ is shown below. There are 3 feature functions in $\mathbf{z}$. The first feature takes 4 hyperparameter arguments, the second feature takes 1 hyperparameter argument, and the third feature takes 2 hyperparameter arguments. All the functions are programmed to accept 4 arguments, but each function may not use all 4 arguments. The arguments are internally processed by each function to the correct form.

$$\mathbf{U}_{\text{sample}} = \begin{bmatrix} [0.0, 1.0] & \text{range}(3.0, 64.0) & \text{uniform}(-5.0, 10.0) & [1.2, 5.2] \\ [0.2, 0.5, 0.8, 1.5, 2.3] & [0.0] & [0.0] & [0.0] \\ [1.0, 2.0, 3.0, 4.0] & \text{linspace}(-22.0, 22.0, 100) & [0.0] & [0.0] \end{bmatrix} \tag{13}$$

To normalize each element in $\mathbf{U}$ to the same scale and make the search tractable, TinyNS uses a *slider matrix* $\mathbf{U}_{\text{slider}}$ during the search process instead of being directly exposed to $\mathbf{U}$.

$$\mathbf{U}_{\text{slider}} = \begin{bmatrix} \zeta_{1,1} & \zeta_{1,1} & ... & \zeta_{1,e} \\ \zeta_{2,1} & \zeta_{2,2} & ... & \zeta_{2,e} \\ . & . & ... & . \\ . & . & ... & . \\ . & . & ... & . \\ \zeta_{n,1} & \zeta_{n,2} & ... & \zeta_{n,e} \end{bmatrix}, \quad \zeta_{i,j} = \begin{cases} \text{linspace}(0, 1, \delta) \Leftrightarrow \left| [\alpha_1^{i,j}, \alpha_2^{i,j}, ..., \alpha_{\gamma_j^i}^{i,j}] \right| \neq 1 \\ 0 \end{cases} \tag{14}$$

δ represents the *granularity factor*, which controls how finely each element in $\mathbf{U}$ can be chosen. Ideally, δ should be equal to the length of the largest array in $\mathbf{U}$. Let $\eta_{i,j}$ be a value in an array element in $\mathbf{U}$. The mapping between $\zeta_{i,j}$ and $\eta_{i,j}$ is:

$$\eta_{i,j} = \alpha_\kappa^{i,j}, \quad \kappa = \text{round}\left(\zeta_{i,j} \cdot \left| [\alpha_1^{i,j}, \alpha_2^{i,j}, ..., \alpha_{\gamma_j^i}^{i,j}] \right| \right), \quad \mu_{i,j} \in [0, 1] \tag{15}$$

The search space for the symbolic components, thereby, is composed of the binary mask c and $\mathbf{U}_{\text{slider}}$.

Problem Formulation (Neural). Consider a collection of k model backbones ϕ constructed from m in Ω. During each iteration in the search process, only one of the models is considered via an ordinal mask d.

$$\text{model}_{\text{iteration}_t} = \phi_i, \quad i \in d, \quad d = [1, 2, ..., k] \tag{16}$$

Each model will have its own optimization hyperparameters (e.g., number of convolutional layers, kernel size, support vector kernel type, etc.). We modify the concept of hyperparameter data structure and slider matrix from the symbolic search space to account for ordinal model choice. Let $\mathbf{V}$ be the 2D hyperparameter data structure for ϕ. The structure of $\mathbf{V}$ remains the same as that of $\mathbf{U}$, now with k rows of hyperparameters. The number of columns of $\mathbf{V}$ is equal to the number of optimization hyperparameters for that ϕ_i which takes the maximum

Just Accepted

number of arguments f.

$$\mathbf{V} = \begin{bmatrix} [\beta_1^{1,1}, \beta_2^{1,1}, ..., \beta_{\gamma_1^1}^{1,1}] & [\beta_1^{1,2}, \beta_2^{1,2}, ..., \beta_{\gamma_2^1}^{1,2}] & \cdots & [\beta_1^{1,f}, \beta_2^{1,f}, ..., \beta_{\gamma_f^1}^{1,f}] \\ [\beta_1^{2,1}, \beta_2^{2,1}, ..., \beta_{\gamma_1^2}^{2,1}] & [\beta_1^{2,2}, \beta_2^{2,2}, ..., \beta_{\gamma_2^2}^{2,2}] & \cdots & [\beta_1^{2,f}, \beta_2^{2,f}, ..., \beta_{\gamma_f^2}^{2,f}] \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ [\beta_1^{k,1}, \beta_2^{k,1}, ..., \beta_{\gamma_1^k}^{k,1}] & [\beta_1^{k,2}, \beta_2^{k,2}, ..., \beta_{\gamma_2^k}^{k,2}] & \cdots & [\beta_1^{k,f}, \beta_2^{k,f}, ..., \beta_{\gamma_f^k}^{k,f}] \end{bmatrix} \tag{17}$$

An example of $\mathbf{V}$ is shown below. The first row corresponds to the hyperparameters for a temporal convolutional network (TCN) [162], and the second row corresponds to the hyperparameters for Bonsai [93].

$$\mathbf{V}_{\text{sample}} = \begin{bmatrix} \underbrace{\text{range}(2, 64)}_{\text{kernel size}} & \underbrace{[1.0, 2.0, 5.0]}_{\text{stack count}} & \underbrace{[[1, 2, 4], [1, 2, 4, 8], [1, 4, 8, 32]]}_{\text{dilation factors}} & \underbrace{\text{uniform}(0.0, 1.0)}_{\text{dropout}} \\ \underbrace{\text{range}(40, 60)}_{\text{prototype count}} & \underbrace{\text{range}(1, 4)}_{\text{sigmoid parameter}} & \underbrace{\text{range}(1, 6)}_{\text{depth}} & [0.0] \end{bmatrix} \tag{18}$$

Since d is ordinal, $\mathbf{V}_{\text{slider}}$ takes a vector form:

$$\mathbf{V}_{\text{slider}} = \begin{bmatrix} \chi_{1,1} & \chi_{1,2} & \cdots & \chi_{1,f} \end{bmatrix}, \quad \chi_{i,j} = \begin{cases} \text{linspace}(0, 1, \delta) \Leftrightarrow \left| [\beta_1^{i,j}, \beta_2^{i,j}, ..., \beta_{\gamma_j^i}^{i,j}] \right| \neq 1 \\ 0 \end{cases} \tag{19}$$

The search space for the neural components, thereby, is composed of the ordinal mask d and $\mathbf{V}_{\text{slider}}$. Note that when $k = 1$, the elements in $\mathbf{V}$ are directly fed to the search algorithm.

Parsing (Symbolic). The python constructs for each function in $\mathbf{z}(\cdot)$ have equivalent C constructs, declared in a `.h` file and defined in a `.cc` file. The `.cc` file also includes an `extract_symbolic(raw_data[], output_feat[], mask[], params[])` function, which takes the windowed and raw sensor data as input (`raw_data[]`), picks functions according to a binary mask array (`mask[]`), applies the corresponding hyperparameters to the chosen functions (`params[]`), and outputs the processed data (`output_feat[]`). TinyNS writes the Pareto-optimal mask c^* as `mask[]`, the Pareto-optimal values in the 2D hyperparameter data structure $\mathbf{U}^*$ as flattened array `params[]`, and the maximum number of arguments each function can take `MAX_PARAM_COUNT` to the `.cc` file. Algorithm 1 provides example implementation for the `extract_symbolic()` function. All of the functions are programmed to take a hyperparameter array of length `MAX_PARAM_COUNT`, internally processing the arguments to the correct form like in Python. An array of function pointers of type `f` allows flexible addition, removal, and access to functions, retaining the same order of functions from Python and allowing sequential application of each function to the raw input data. The output channel count for each function is variable and defined in `func_output_size[]`.

Parsing (Neural). TinyNS uses the TensorFlow Lite Micro (TFLM) [43] Mbed RTOS C file system for real-time model inference on microcontrollers. Algorithm 2 shows the `main.cc` file of the file system. We choose TFLM as the runtime inference engine due to its widespread public use, portable design philosophy, heterogenous hardware support, memory efficient paradigms, static memory allocation, and pathways for easy model replacement [43, 136]. First, the model backbone in Python is constructed using Keras [72] or Keras/TensorFlow wrappers for Scikit-learn [122] with TensorFlow backend [1]. Next, the Keras model is converted to a `.tflite` model, with appropriate quantization schemes applied during conversion (e.g., no quantization or full integer quantization using a

Just Accepted

Algorithm 1 Example of `extract_symbolic()` for Symbolic Neuro Symbolic parsing

```
#include "_____.h"
.
.
#define MAX_PARAM_COUNT 3 //written by parser
#define MAX_NUMBER_OF_FUNC 4
const int func_output_size[MAX_NUMBER_OF_FUNC] = {1,1,1,4};
int mask_array[MAX_NUMBER_OF_FUNC] = {1,1,0,1}; //written by parser
float params_array[MAX_NUMBER_OF_FUNC*MAX_PARAM_COUNT] =
{2.2,39,-23,1.2,0.0,0.0,23.5,2.2,0.0,-5.1,0.95,0.0}; //written by parser

void func_1(float* input_ar, float* output_ar, float* param_ar){
}
void func_2(float* input_ar, float* output_ar, float* param_ar){
}
void func_3(float* input_ar, float* output_ar, float* param_ar){
}
void func_4(float* input_ar, float* output_ar, float* param_ar){
}

void extract_symbolic(float *raw_data, float *output_feat,
    int *mask, float* params){
    typedef void (*f)(float[], float[], float[]);
    int j = 0;
    float param_ar[MAX_PARAM_COUNT] = {0.0};
    f func[MAX_NUMBER_OF_FUNC] = {&func_1, &func_2, &func_3, &func_4};

    for(int i = 0; i < MAX_NUMBER_OF_FUNC; i++){
        for (int k = 0; k<MAX_PARAM_COUNT; k++){
            param_ar[k] = params[i*MAX_PARAM_COUNT + k];
        }
        if (mask[i] == 1){
                float temp_buff[func_output_size[i]];
                func[i](raw_data, temp_buff, param_ar);
                for (int k = 0; k < func_output_size[i]; k++){
                    output_feat[j] = temp_buff[k];
                    k = k+1;
                    j = j+1;
                }
        }
    }
}
```

Algorithm 2 Example of `main.cc` for Symbolic Neuro Symbolic parsing

```
#include "_____.h"
.
.
Timer t;
constexpr int kTensorArenaSize = 500 * 1024; //written by parser
alignas(16) uint8_t tensor_arena[kTensorArenaSize];
tflite::MicroModelRunner<float, float, 13> *runner; //written by parser
float raw_data[kInputSize]; //written by parser
float input_model[kModelInputSize]; //written by parser

int main() {
   static tflite::MicroMutableOpResolver<13> resolver; //written by parser
    resolver.AddShape(); //written by parser
    resolver.AddStridedSlice(); //written by parser
    .
    .

    static tflite::MicroModelRunner<float, float, 13>model_runner(
    g_featnn_model_data, resolver, tensor_arena,
    kTensorArenaSize); //written by parser
    runner = &model_runner;

    get_sensor_data(raw_data);
    extract_symbolic(raw_data, input_model, mask_array, params_array);

    t.start();
    runner->SetInput(input_model);
    runner->Invoke();
    t.stop();

    for (size_t i = 0; i < kCategoryCount; i++) {
        float converted = runner->GetOutput()[i]; //written by parser
        printf("%0.3f", converted);
        if (i < (kCategoryCount - 1)) {
            printf(",");
        }
    }
    printf("\n");
    printf("timer output: %f\n", t.read());
    t.reset();
}
```

representative dataset). The parser now needs to check if the operators in the `.tflite` file are present in the TFLM operator resolver list. The steps are:

- Read the `.tflite` file as a flatbuffer byte array.
- Decode the value at the start of the flatbuffer using packer type `flatbuffers.packer.uoffset` to create a model object.
- Unpack the model object into a graph of flatbuffer objects.
- Convert the hierarchy of flatbuffer objects to a nested opcode dictionary.
- Match the opcode keys in the model to the opcode names in the `BUILTIN_OPCODE2NAME` dictionary provided with the TFLite API.
- Check if the resulting set of names is present in the `AVAILABLE_TFLM_OPS` list.

If all the operators in the model are supported by TFLM, then, the `.tflite` file is converted to a flatbuffer model schema using Linux hex dump, generating `.cc` file of the model. The parser opens the `main.cc` file and makes the following changes:

- Declare the TFLM arena size depending on target hardware constraints. The arena is a stack in the SRAM used for initialization and runtime variable storage.

Just Accepted

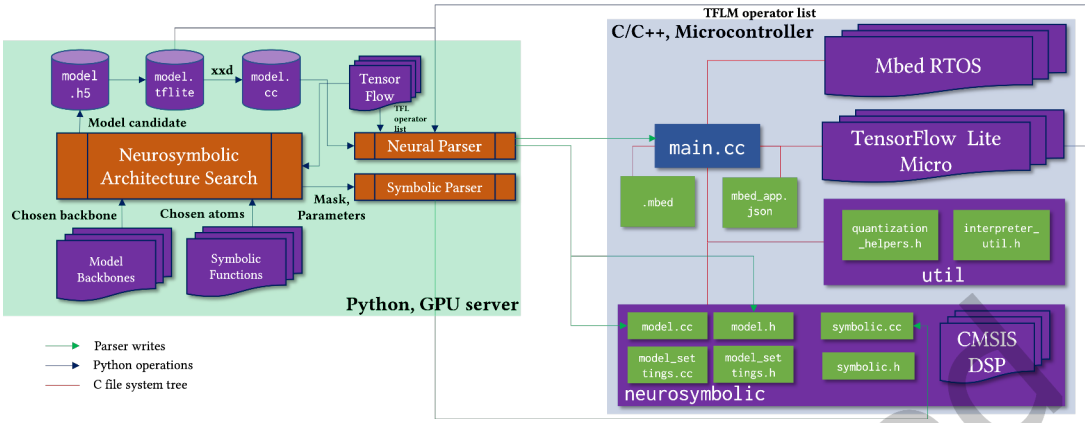

Fig. 4. Architecture of automated neurosymbolic parsing for Symbolic Neuro Symbolic.

- Declare the arrays for storing raw data and processed output from `extract_symbolic`, which is also the input to the model. The arrays can be float or int depending on model quantization. In TFLM, flattened input arrays are internally reshaped to match the input tensor shape of the model.
- Declare a TFLM interpreter instance (`MicroModelRunner`), which resolves the model graph during runtime. The data types should be the input and output data types of the model, and the last number indicates the number of unique ML operators that need to be called by the operator resolver.
- Declare the TFLM operator resolver instance (`MicroMutableOpResolver`), which links only the essential ML operators to the model graph.
- Add the operators necessary to resolve the graph from the intersection of the set of model opcode names and the `AVAILABLE_TFLM_OPS` list.
- Pass the flatbuffer model schema, the operator resolver, and the arena to the interpreter.
- Dequantize the outputs if the model output is quantized.

Fig. 4 summarizes the parser operation between the Python file system and the TFLM Mbed RTOS C file system.

Examples. An example includes finding the best set of features for on-device wearable human activity recognition. Another example includes finding the best model among a set of models for on-device wearable fall detection under 2 kB of memory. We showcase the examples in Section 5.2 and Section 5.3. In the first example, the search algorithm is given a model backbone and several temporal, statistical, and spectral features that can operate on the raw, windowed data. The goal is to find the best model hyperparameters and features that work well to give maximal activity detection accuracy within the hardware constraints. In the second example, the goal is to find the best model and its corresponding hyperparameters that can detect falls within a tight memory budget.

4.2 Neuro→Symbol

Problem Formulation. There are two ways to realize this paradigm. *Firstly*, if a static domain-engineered function $z(\cdot)$ with hyperparameter data vector $\mathbf{u}$ operates on the output of the model to produce high-level reasoning, then the symbolic search space only contains $\mathbf{u}$.

$$\mathbf{u} = \left[[\alpha_1^{1,1}, \alpha_2^{1,1}, ..., \alpha_{\gamma_1^1}^{1,1}] \quad [\alpha_1^{1,2}, \alpha_2^{1,2}, ..., \alpha_{\gamma_2^1}^{1,2}] \quad ... \quad [\alpha_1^{1,e}, \alpha_2^{1,e}, ..., \alpha_{\gamma_e^1}^{1,e}] \right] \tag{20}$$

Just Accepted

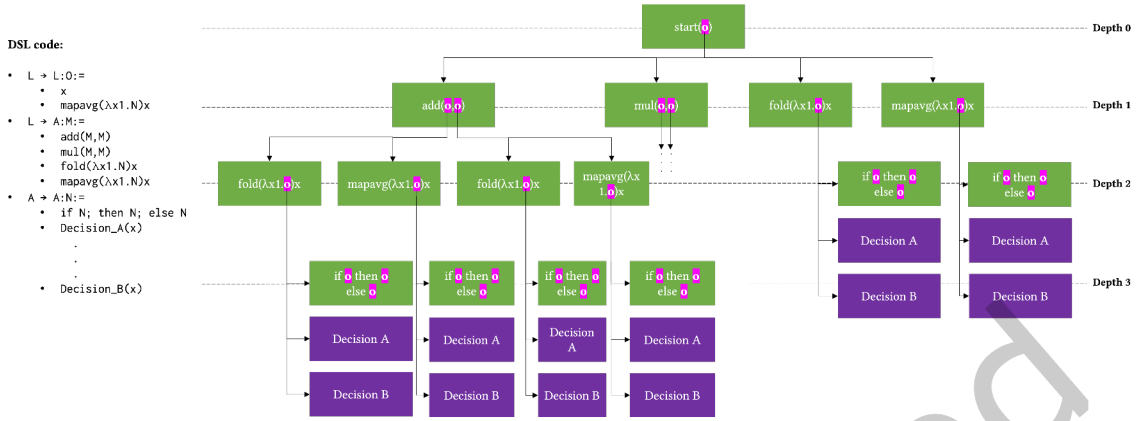

Fig. 5. Sample program supergraph generated from the DSL operator space for Neuro → Symbol [118, 149, 160]. Green nodes represent non-terminal nodes and purple nodes represent goal nodes.

u is similar in form as **U** from Section 4.1, but only corresponds to the optimization hyperparameter space for a single function. The neural search space is the same as that shown in Section 4.1.

Secondly, consider a collection of logical (e.g., AND, OR, NOT) operators Λ, relational (e.g., equivalence, less than or equal to, greater than or equal to) operators $\mathfrak{R}$, arithmetic (e.g. add, multiply) operators Ξ, and conditional (e.g., if else then) operators Υ, expressed in a *Domain-Specific Language* (DSL) [118]. Given maximum tree depth $\wp$ and a finite number of trees N, the symbolic atoms can be combined to synthesize candidate program graphs (or program decision trees) that can perform high-level reasoning over several neural output timesteps.

$$\mathbf{G} = \text{GenerateProgramTree}(\{\Lambda, \mathfrak{R}, \Xi, \Upsilon\}, \wp, N) \tag{21}$$

Fig. 5 shows an example program supergraph generated from the DSL operator space, from which candidate trees can be extracted. The GenerateProgramTree() is an enumeration algorithm [118, 160] that generates all possible combinations of program graphs **G** given $\wp$ and N using *context-free grammar*. The rules of connection are fixed by the DSL. Ideally, the path cost of the program graph should be low for interpretability and resource savings, yet have high accuracy. In other words, in Fig. 5, the goal is to find the top-performing shortest path to Decision A and Decision B. The symbolic search space is an ordinal mask j that represents one of N program subgraphs extracted from the program supergraph.

$$\text{program}_{\text{iteration}_t} = G_i, \quad G_i \in \mathbf{G}, \quad i \in j, \quad j = [1, 2, ..., N] \tag{22}$$

The neural search space is the same as that shown in Section 4.1.

Parsing. Neuro→Symbol follows the same model parsing strategy discussed in Section 4.1, Algorithm 2 and Fig. 4. For symbolic parsing, in the first case, the symbolic parser passes the Pareto-optimal $\mathbf{u}^*$ as `hyperparameter_vector[]` to the `main.cc` file, where the function $z(\cdot)$ is defined as `symbolic_function()`. This function operates on the output of the model. An example of this case is shown in Algorithm 3. In the second case, the program decision tree along with the grammar and the parser runtime are ported as header files. The steps to port a program tree generated using ANTLR [119] are:

- Port the graph as a `.txt` or `.h` file, expressed in DSL.
- Define the lexer rules in a `.g4` files. The lexer rules are necessary to tokenize the DSL program tree.

Just Accepted

Algorithm 3 Example of `main.cc` for the first case of Neuro→Symbol

```
SAME AS ALGORITHM 2
.
.

float hyperparameter_vector[3] = [-2.4, 1.1, 2.0]; //written by parser
void symbolic_function(float* inp, float* out, float* params){
}

float raw_data[kInputSize]; //written by parser
float model_output[kOutputSize]; //written by parser
float symbolic_output[kSymbolicSize]; //written by parser

int main() {
    SAME AS ALGORITHM 2
    .
    .
    runner = &model_runner;

    get_sensor_data(raw_data);
    t.start();
    runner->SetInput(raw_data);
    runner->Invoke()
    for(size_t i = 0; i< kCategoryCount; i++){
        model_output[i] = runner->GetOutput()[i]; //written by parser
    }
   symbolic_function(model_output, symbolic_output, hyperparameter_vector);
    t.stop();
    for(size_t i = 0; i< kSymbolicSize; i++){
        printf("%0.3f\n", symbolic_output[i]);
    }
    printf("\n");
    printf("timer output: %f\n", t.read());
    t.reset();
}
```

Algorithm 4 Example of `main.cc` for the second case of Neuro→Symbol

```
SAME AS ALGORITHM 2
.
.
INCLUDE TREE, PARSER RUNTIME AND GRAMMAR HEADER FILES HERE

float raw_data[kInputSize]; //written by parser
float model_output[kOutputSize]; //written by parser
float symbolic_output[kSymbolicSize]; //written by parser

int main() {
    SAME AS ALGORITHM 2
    .
    .
    runner = &model_runner;

    get_sensor_data(raw_data);

    t.start();
    runner->SetInput(raw_data);
    runner->Invoke()

    for(size_t i = 0; i< kCategoryCount; i++){
        model_output[i] = runner->GetOutput()[i]; //written by parser
    }
  program_graph_runtime(model_output, symbolic_output); //lexer->parser->visitor
    t.stop();

    for(size_t i = 0; i< kSymbolicSize; i++){
        printf("%0.3f\n", symbolic_output[i]);
    }
    printf("\n");
    printf("timer output: %f\n", t.read());
    t.reset();
}
```

Just Accepted

- Run the ANTLR runtime engine with the `lexer.g4` file in the target language (Python or C) to create the necessary lexer files.
- Define the grammar in another `.g4` file. The grammar defines the relations between the class of tokens, assigning labels using the DSL operator space.
- Run the ANTLR runtime engine again, but with the `grammar.g4` file to create the parser files, which processes the program graph to create a hierarchical abstract syntax tree. Specify the -visitor flag when running the engine to have control over the query traversal.
- Create a visitor, which will traverse the tree according to the parser grammar.
- Pass the DSL graph from the `.txt` or `.h` file to the lexer as a string argument. The tokenized tree is passed to the parser to generate the syntax tree, which is finally passed to the visitor for traversal.

Examples. An example of the first approach includes joint optimization of a symbolic object tracker with a neural object detector using the CenterNet algorithm [179]. We showcase this example in Section 5.4. The object detector backbone is a ResNet-34 + Deformable Convolutional Network, with the optimization hyperparameters being the number of convolutional stacks, the kernel size, whether to use layer-wise activations or not and the head convolutional value. Given an input image $I^t \in \mathbb{R}^{W \times H \times 3}$, the model outputs the center points $\hat{D}_{\mathbf{p}_i}$ and bounding box dimensions $\hat{S}_{\mathbf{p}_i}$ of the detected objects, as well as a heatmap of the centroid of the objects $\hat{Y}_{xyc}, \hat{Y} \in [0,1]^{\frac{W}{R} \times \frac{H}{R} \times C}$) based on the rendering function $\mathcal{R}$ with Gaussian Kernel σ_i for each class $c \in \{0, 1, ..., C-1\}$.

$$\mathcal{R}_q(\{\mathbf{p}_0, \mathbf{p}_1, ...\}) = \max_i \exp\left(\frac{(\mathbf{p}_i - \mathbf{q})^2}{2\sigma_i^2}\right), \mathbf{q} \in \mathbb{R}^2, \mathbf{p} \in \mathbb{R}^2 \tag{23}$$

q is a position on the image. To track and associate objects across frames, the network is also fed the previous frame I^{t-1} and prior detection heatmaps $\mathcal{R}(\mathbf{p}^{t-1})$. The network then outputs the 2D offset of the object $\mathbf{d}^t$, with associations performed using greedy matching. Thus, the network is trained via a weighted sum of the focal loss $\mathcal{L}_k$ (based on ground truth heatmap Y_{xyc}, $Y \in [0,1]^{\frac{W}{R} \times \frac{H}{R} \times C}$), the size $\mathcal{L}_{\text{size}}$ (based on ground truth bounding box dimensions **s**), and the local location regression $\mathcal{L}_{\text{off}}$ (based on ground truth object positions $\mathbf{p}_i$).

$$\mathcal{L}_k = \frac{1}{N} \sum_{xyc} \begin{cases} (1-\hat{Y}_{xyc})^2 \log(\hat{Y}_{xyc}) \Leftrightarrow Y_{xyc} = 1 \\ (1-Y_{xyc})^4 (\hat{Y}_{xyc})^2 \log(1-\hat{Y}_{xyc}) \end{cases} \tag{24}$$

$$\mathcal{L}_{\text{size}} = \frac{1}{N} \sum_{i=1}^{N} |\hat{S}_{\mathbf{p}_i} - \mathbf{s}_i| \tag{25}$$

$$\mathcal{L}_{\text{off}} = \frac{1}{N} \sum_{i=1}^{N} \left| \hat{D}_{\mathbf{p}_i^t} - (\mathbf{p}_i^{t-1} - \mathbf{p}_i^t) \right| \tag{26}$$

A filter is used to discard heatmaps below a certain rendering threshold τ or objects whose detection confidence scores $w, w \in [0,1]$ are below a certain threshold θ. These thresholds form the optimization hyperparameters for the symbolic component (the filter). The error metric is the sum of the multi-object tracking accuracy (MOTA) and the minimal cost change from the predicted identification of objects to the correct identification (IDF1) [179].

4.3 Neuro ∪ Compile (Symbolic)

Problem Formulation. There are two ways to realize this paradigm. *Firstly*, if the rules are non-differentiable, the rules are characteristic of certain architectural encodings post-training, or the rules cannot be explicitly expressed in the model learning algorithm, then the constraints can be expressed as regularizer terms in Eq. 7:

$$\min f_{\text{opt}}, \quad f_{\text{opt}} = \lambda_1 f_{\text{error}}(\Omega') + \lambda_2 f_{\text{flash}}(\Omega') + \lambda_3 f_{\text{SRAM}}(\Omega') + \lambda_4 f_{\text{latency}}(\Omega') + \lambda_5 f_{\text{rule 1}}(\Omega') + \lambda_6 f_{\text{rule 2}}(\Omega') + \ldots \tag{27}$$

Ω' contains only the ML components (i.e., $\Omega' = \{\{V, E\}, \theta_m, m, w\}$), reducing the neurosymbolic architecture search to a NAS problem, regularized by additional scalar rules. The rules can form *soft constraints* that do not form piecewise penalization functions, or *hard constraints* like SRAM and flash consumption to strongly penalize the search algorithm beyond a small, valid region of Ω'. *Secondly*, if the rules are differentiable, or the rules can be compiled away during training as input-output pairs, then the constraints can be included as physics metadata channels in the learning algorithm as inputs to the model graph q:

$$\min f_{\text{opt}}, \quad f_{\text{opt}} = \lambda_1 f_{\text{error}}(\Omega') + \lambda_2 f_{\text{flash}}(\Omega') + \lambda_3 f_{\text{SRAM}}(\Omega') + \lambda_4 f_{\text{latency}}(\Omega') \tag{28}$$

where,

$$f_{\text{error}}(\Omega') = \mathcal{L}_{\text{validation}}(\mathbf{Y}', \mathbf{Y}), \quad \mathbf{Y}' = q^{\Omega'}(\mathbf{X}, \mathbf{x}_{\text{physics metadata channel}}) \tag{29}$$

Parsing. In the first case, the parsers only need to map the model from Python to C, following the recipe of model parsing in Section 4.1, Algorithm 2 and Fig. 4. In the second case, since the rules and hyperparameters are static and operate on the input data, there is no concept of symbolic optimization or symbolic parsing. Rather, there exists a function called `extract_physics()` in `main.cc` that operates on the raw data to generate the physics metadata channel, shown in Algorithm 5. The channel is appended to the end of the raw data, which is then fed to the model as an input tensor.

Just Accepted

Algorithm 5 Example of `main.cc` for the second case of Neuro ∪ Compile (Symbolic)

```
SAME AS ALGORITHM 2
.
.
float raw_data[kInputSize]; //written by parser
float physics_channel[kPhysicsSize];
float input_model[kInputSize + kPhysicsSize]; //written by parser

int main() {
    SAME AS ALGORITHM 2
    .
    .
    runner = &model_runner;

    get_sensor_data(raw_data);
    extract_physics(raw_data, physics_channel);
    for (int i = 0; i < kInputSize; i++){
        input_model[i] = raw_data[i];
    }
    int j = 0;
    for (int i = kInputSize; i < kInputSize + kPhysicsSize; i++){
        input_model[i] = physics_channel[j];
        j = j+1;
    }
    t.start();
    .
    .
    SAME AS ALGORITHM 2
}
```

Examples. An example of the first technique includes finding adversarially robust TinyML models, where $f_{\text{rule 1}}(\Omega')$ denotes the white-box adversarial robustness score from RobustBench [38] or AutoAttack [39] benchmarks on a perturbed validation set (e.g., perturbed using fast gradient sign method (FGSM) or projected gradient descent (PGD)) versus the clean validation set.

$$f_{\text{rule 1}}(\Omega') = 1 - \frac{1}{N}\sum_{i=0}^{N} q_i, \quad q_i = \begin{cases} 1 \Leftrightarrow y'^{x_i} = y'^{x_i,\text{perturbed}} \\ 0 \end{cases} \tag{30}$$

where,

$$x_{i,\text{perturbed}} = \underbrace{\left[x_i + \varepsilon \cdot \text{sign}\left(\nabla_{x_i}\mathcal{L}_{\text{validation}}(q^{\Omega'}(x_i), y^i)\right)\right]}_{\text{FGSM}} \vee \underbrace{\left[\text{clip}_{\varepsilon}\left(x_i^t + \alpha \cdot \text{sign}\left(\nabla_{x_i}\mathcal{L}_{\text{validation}}(q^{\Omega'}(x_i)^t, y^i)\right)\right)\right]}_{\text{PGD}} \tag{31}$$

α and ε are attack strength hyperparameters in Eq. 31. An example of the second technique includes supplying a neural inertial navigation model with local-variance step detector binary mask or mean Fourier transform coefficients of accelerometer readings ${}^I\hat{\mathbf{a}}$, signifying transportation modes. The goal is to prevent the network from outputting invalid displacements when the object is static [134].

$$\mathbf{x}_{\text{physics metadata channel}} = c({}^I\hat{\mathbf{a}}), \quad c_j({}^I\hat{\mathbf{a}}) = \underbrace{\begin{cases} 1 \Leftrightarrow \hat{\mathbf{a}}^I_{L,\Delta t} > \zeta \cdot \sqrt{\frac{\sum_{k\in\Delta t}\left(\hat{\mathbf{a}}^I_{L,k} - \overline{\hat{\mathbf{a}}^I_{L,\Delta t}}\right)^2}{n}} \\ 0 \end{cases}}_{\text{step detector}} \vee \underbrace{\overline{|\text{FFT}(|\hat{\mathbf{a}}^I_{\Delta t}|)|}}_{\text{Fourier transform}} \tag{32}$$

where, j is the measurement epoch, Δt is the length of current time window, $\hat{\mathbf{a}}^I_{L,\Delta t} = G_{5,f_c}(|\hat{\mathbf{a}}^I_{\Delta t}|) - G_{5,f_c}(\overline{|\hat{\mathbf{a}}^I_{\Delta t}|})$, ζ is a tunable parameter and $G_{5,f_c}(\cdot)$ represents a 5th order low-pass filter with cutoff f_c. The model is expected to output zero displacements when the physics metadata channel value drops below a threshold τ.

$$\mathbb{E}(y'_j) \rightarrow 0 \mid x_{j,\text{physics metadata channel}} < \tau \tag{33}$$

Just Accepted

We showcase the examples in Section 5.5 and Section 5.6.

4.4 Symbolic[Neuro]

Problem Formulation. Consider a dynamical system such that $g : \hat{\mathbf{x}}_{k+1|k} \rightarrow \mathbf{u}_{k+1}, \hat{\mathbf{x}}_k \mid g$ is non-linear. $\hat{\mathbf{x}}_{k+1|k}$ represents the state at epoch $k+1$, $\hat{\mathbf{x}}_k$ represents the state at epoch k, $g(\cdot)$ is a neural network backbone, and $\mathbf{u}_{k+1}$ represents the control input (sensor measurements) at epoch $k+1$. The neural system evolution is given as follows:

$$\hat{\mathbf{x}}_{k+1|k} = g_v(\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}, \mathbf{w}_{k+1}) \tag{34}$$

$\mathbf{w}_{k+1}$ is the additive White Gaussian process noise with covariance $\mathbf{Q}$. Now, consider measurement updates $\mathbf{z}_{k+1}$ coming from a symbolic observation model $h(\cdot)$ via complementary sensor measurements.

$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + \mathbf{K}_{k+1}\left(\underbrace{\mathbf{z}_{k+1} - h_u(\hat{\mathbf{x}}_{k+1|k}, \mathbf{v}_k)}_{\text{measurement residual}}\right) \tag{35}$$

$\mathbf{v}_k$ is the additive White Gaussian measurement noise with covariance $\mathbf{R}$ and $\mathbf{K}_{k+1}$ is a gain factor. The goal is to optimally fuse the neural system model and the symbolic measurement model. Assuming Markov property, modeling the uncertainty in $g(\cdot)$ and $h(\cdot)$ using Kalman filter theory allows optimal fusion [50].

$$\mathbf{P}_{k+1|k} = \mathbf{A}\mathbf{P}_k\mathbf{A}^T + \mathbf{B}_{k+1}\mathbf{U}_k\mathbf{B}_{k+1}^T + \mathbf{Q}_k, \quad \mathbf{A}_{k+1} = \left.\frac{\partial g}{\partial x}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}, \mathbf{w}_{k+1}}, \quad \mathbf{B}_{k+1} = \left.\frac{\partial g}{\partial u}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}, \mathbf{w}_{k+1}} \tag{36}$$

$$\mathbf{P}_{k+1|k+1} = (\mathbf{I} - \mathbf{K}_{k+1}\mathbf{H}_{k+1})\,\mathbf{P}_{k+1|k}, \quad \mathbf{H}_{k+1} = \left.\frac{\partial h}{\partial x}\right|_{\hat{\mathbf{x}}_{k+1|k}, \mathbf{v}_k} \tag{37}$$

where,

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^T\left(\underbrace{\mathbf{H}_{k+1}\mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^T + \mathbf{R}_{k+1}}_{\text{innovation covariance}}\right)^{-1} \tag{38}$$

$\mathbf{A}_{k+1}$ and $\mathbf{B}_{k+1}$ represents the linearized Jacobian of the neural network w.r.t. the past state and control inputs, while $\mathbf{H}_{k+1}$ represents the linearized partial derivative of the observation model w.r.t. the past state. The predicted process covariance $\hat{\mathbf{P}}$ is given by the Lyapunov equation and updated during measurements using algebraic Riccati recursion [151]. The goal of the search algorithm is to find the optimal hyperparameters of $g(\cdot)$ and $h(\cdot)$, given by hyperparameter vectors $\mathbf{v}$ and $\mathbf{u}$, respectively:

$$\mathbf{u} = \left[[\alpha_1^{1,1}, \alpha_2^{1,1}, ..., \alpha_{\gamma_1^1}^{1,1}] \quad [\alpha_1^{1,2}, \alpha_2^{1,2}, ..., \alpha_{\gamma_2^1}^{1,2}] \quad ... \quad [\alpha_1^{1,e}, \alpha_2^{1,e}, ..., \alpha_{\gamma_e^1}^{1,e}]\right] \tag{39}$$

$$\mathbf{v} = \left[[\beta_1^{1,1}, \beta_2^{1,1}, ..., \beta_{\gamma_1^1}^{1,1}] \quad [\beta_1^{1,2}, \beta_2^{1,2}, ..., \beta_{\gamma_2^1}^{1,2}] \quad ... \quad [\beta_1^{1,f}, \beta_2^{1,f}, ..., \beta_{\gamma_f^1}^{1,f}]\right] \tag{40}$$

Parsing. The model parsing follows the same recipe shown in Section 4.1, Algorithm 2, and Fig. 4. The symbolic parser sends the optimal $\mathbf{u}^*$ to `main.cc`. Algorithm 6 shows an example of the `main.cc`. The program extensively uses matrix operations (obtainable through CMSIS-NN library [95] available through TFLM) to compute the Kalman hyperparameters. CMSIS-NN matrix operation constructs are used in `reshape_jacobian()`, `lyapunov_eq()`, `measurement_update()`, `get_pd()`, `compute_kalman_gain()`, and `ricatti()` functions to accelerate matrix operations through vector processors found in some Cortex-M microcontrollers. However, a key challenge in realizing the Symbolic[Neuro] form is the lack of on-board Jacobian computation support

Just Accepted

(`GetJacobian()`).

Examples. We showcase a Neural-Kalman filter that fuses GPS measurements with a neural inertial odometry model to regress an object's position [50]. The example is shown in Section 5.7. The neural network regresses the

Algorithm 6 Example of `main.cc` for Symbolic[Neuro]

```
SAME AS ALGORITHM 2
.
INCLUDE CMSIS_NN HEADERS HERE
#define STATE_SIZE 3
float raw_data[kRawData];
float input_model[kRawData + STATE_SIZE];
float obs_model_params[4] = {-2.0, 1.0, 0.0, 37.5}; //written by parser
cur_state[3] = {0.0,0.0,0.0}
float jacobian[kJacobianSize] = {0.0};
float reshaped_jacobian[kA][kB];
float P[kC][kD];
float K[kE][kF];
float H[kG][kH];
float out[koutsize] = {0.0};

void reshape_jacobian(float* flattened_jacobian[], float* 2D_jacobian[][]){
}
void lyapunov_eq(float* covariance_mat[][], float* 2D_jacobian[][], float* sensor_data[]){
}
void measurement_update(float* state[], float* gain_matrix[][], float* sensor_data[]){
}
void get_pd(float* obs_model[][], float* output_obs_model[]){
}
void obs_model(float* out[], float* state[], float* params[]){
}
void compute_kalman_gain(float* gain_matrix[][], float* covariance_mat[][], float* out[]){
}
void ricatti(float* covariance_mat[][], float* gain_matrix[][], float* out[]){
}
int main() {
    SAME AS ALGORITHM 2
    .
    ////////////////////LOOP//////////////////////////
    get_sensor_data(raw_data);
    for(int i = 0; i < kRawData; i++){
        input_model[i] = raw_data[i]
    }
    for(int i = kRawData; i < kRawData + STATE_SIZE; i++){
        input_model[i] = cur_state[i];
    }
    t.start();
    runner->SetInput(input_model);
    runner->Invoke();
    for (size_t i = 0; i < STATE_SIZE; i++) {
       cur_state[i] = runner->GetOutput()[i]; //neural system model
    }
    for (size_t i = 0; i < STATE_SIZE; i++) {
       jacob[i] = runner->GetJacobian()[i];
    }
    reshape_jacobian(jacob,reshaped_jacobian); //reshape flattened Jacobian to 2D matrix
    lyapunov_eq(P, reshaped_jacobian, raw_data); //compute P for neural system model
    get_comp_sensor_data(raw_data);
    measurement_update(cur_state, K, raw_data); //update state
    for (size_t i = 0; i < STATE_SIZE; i++) {
       printf("%f", cur_state[i]);
    }
    obs_model(out, cur_state, obs_model_params); //get observations
    get_pd(H,out); //compute partial derivative
    compute_kalman_gain(K, P, H) //compute the gain matrix
    ricatti(P, K, H); //update P during measurement update
    t.stop();
    printf("\n");
    printf("timer output: %f\n", t.read());
    t.reset();
    ////////////////////LOOP//////////////////////////
}
```

Just Accepted

object's 2D velocity v_x, v_y from accelerometer $\hat{\mathbf{a}}^I$, gyroscope $\hat{\mathbf{w}}^I$ and magnetometer $\hat{\mathbf{m}}^I$ readings:

$$(v_{x,k}, v_{y,k}) = g\big(\mathbf{v}^I(0), \mathbf{g}_0^I, \mathbf{N}_0^I, \hat{\mathbf{a}}_{q:q+n}^I, \hat{\mathbf{w}}_{q:q+n}^I, \hat{\mathbf{m}}_{q:q+n}^I, c_k({}^I\hat{\mathbf{a}})\big), \quad c_k({}^I\hat{\mathbf{a}}) = \left|\overline{|\mathrm{FFT}(|\hat{\mathbf{a}}_{q:q+n}^I|)|}\right|. \tag{41}$$

The system propagation is given as follows:

$$\hat{\mathbf{x}}_{k+1|k} = \mathbf{A}\hat{\mathbf{x}}_k + f(\mathbf{u}_{k+1}) \tag{42}$$

$$\mathbf{P}_{k+1|k} = \mathbf{A}\mathbf{P}_k\mathbf{A}^T + \mathbf{B}_{k+1}\mathbf{U}_k\mathbf{B}_{k+1}^T, \quad \mathbf{B}_{k+1} = \left.\frac{\partial f}{\partial u}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}}$$

where,

$$\hat{\mathbf{x}} = \begin{bmatrix} \hat{L}_x \\ \hat{L}_y \\ v_x \\ v_y \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \mathbf{a}_{q:q+n}^I \\ \mathbf{w}_{q:q+n}^I \\ \mathbf{m}_{q:q+n}^I \\ c(\mathbf{a}_{q:q+n}^I) \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} \mathbf{I}_{2\times2} & \mathbf{0}_{2\times2} \\ \mathbf{0}_{2\times2} & \mathbf{0}_{2\times2} \end{bmatrix}, \quad \mathbf{B}_{k+1} = \begin{bmatrix} \frac{\Delta t \partial g_v(\cdot)_x}{\partial \mathbf{a}_{q:q+n}^I} & \frac{\Delta t \partial g_v(\cdot)_x}{\partial \mathbf{w}_{q:q+n}^I} & \frac{\Delta t \partial g_v(\cdot)_x}{\partial \mathbf{m}_{q:q+n}^I} & \frac{\Delta t \partial g_v(\cdot)_x}{\partial c(\mathbf{a}_{q:q+n}^I)} \\ \frac{\Delta t \partial g_v(\cdot)_y}{\partial \mathbf{a}_{q:q+n}^I} & \frac{\Delta t \partial g_v(\cdot)_y}{\partial \mathbf{w}_{q:q+n}^I} & \frac{\Delta t \partial g_v(\cdot)_y}{\partial \mathbf{m}_{q:q+n}^I} & \frac{\Delta t \partial g_v(\cdot)_y}{\partial c(\mathbf{a}_{q:q+n}^I)} \\ \frac{\partial g_v(\cdot)_x}{\partial \mathbf{a}_{q:q+n}^I} & \frac{\partial g_v(\cdot)_x}{\partial \mathbf{w}_{q:q+n}^I} & \frac{\partial g_v(\cdot)_x}{\partial \mathbf{m}_{q:q+n}^I} & \frac{\partial g_v(\cdot)_x}{\partial c(\mathbf{a}_{q:q+n}^I)} \\ \frac{\partial g_v(\cdot)_y}{\partial \mathbf{a}_{q:q+n}^I} & \frac{\partial g_v(\cdot)_y}{\partial \mathbf{w}_{q:q+n}^I} & \frac{\partial g_v(\cdot)_y}{\partial \mathbf{m}_{q:q+n}^I} & \frac{\partial g_v(\cdot)_y}{\partial c(\mathbf{a}_{q:q+n}^I)} \end{bmatrix} \tag{43}$$

$$f(\cdot) = \begin{bmatrix} \Delta t \cdot \mathbf{I}_{2\times2} \\ \mathbf{I}_{2\times2} \end{bmatrix} \cdot g_v(\cdot), \quad \Delta t = \frac{s}{n-s}, \quad s = \text{stride}, n = \text{window size} \tag{44}$$

U consists of Allan variance parameters [52] of the inertial measurement unit. The measurement updates **z** come from the GPS module. h denotes the inverse mapping from longitude-latitude to 2D Cartesian coordinates. The hyperparameters of the neural network and the Kalman filter are optimized jointly.

4.5 Neuro[Symbolic]

Problem Formulation and Parsing. This paradigm is equivalent to a model with special operators or layers. The search space, therefore, contains the hyperparameters of the model backbone to be optimized. The model parsing follows the same recipe shown in Section 4.1, Algorithm 2, and Fig. 4, with no symbolic parsing. However, the special layers must be added as custom operators first to TFLite, and then to TFLM. The steps are as follows:

- Create the custom operator in TensorFlow.
- Clone Tensorflow repository.
- Define the `init()`, `free()`, `prepare()`, and `eval()` functions for the operator in the `OPERATOR_NAME.cc` file in `tensorflow/lite/kernels/` directory.
- Register the operator in `tensorflow/lite/kernels/register.cc` and `register_ref.cc`. Add the registration under `namespace custom` and `BuiltinRefOpResolver::BuiltinRefOpResolver()`. In the `BUILD` file, under `cc_library( name = "builtin_op_kernels"`, add the operator `.cc` file names under `srcs`. Add the dependencies under `deps`.
- Configure, build, and install the modified TensorFlow. Load the model with the custom operator in the TFLite interpreter in Python to verify the correct operation.
- From `tensorflow/lite/core/api/flatbuffer_conversions.cc`, under `ParseOpDataTfLite`, extract the code for parsing the operator into a function.
- Extract the reference for the operator to a standalone header from `tensorflow/lite/kernels/internal/reference/`. Add the new header to `tensorflow/lite/kernels/internal/BUILD`.
- Copy the operator code from `tensorflow/lite/kernels/OPERATOR_NAME.cc` to `tensorflow/lite/micro/kernels/OPERATOR_NAME.cc`. Remove TFLite-specific code. Add the operator registrations in `micro_ops.h`, `micro_mutable_op_resolver.h`, and `all_op_resolver.cc`.

JUST Accepted

5 EVALUATION

In this section, we evaluate the performance of TinyNS on six different case studies resembling four neurosymbolic architecture search recipes (Section 5.2 to Section 5.7). We also validate the viability of TinyNS for generating performant microcontroller-class models on the industry-standard MLPerf Tiny v0.5 Inference Benchmark [13] in Section 5.1.

5.1 MLPerf Tiny v0.5 Inference Benchmark

The MLPerf Tiny v0.5 Benchmark Suite contains four classification tasks and quality target metrics representing a wide array of TinyML applications [13, 136]. The tasks include image classification (CIFAR10 dataset [91]), unsupervised anomaly detection (ToyADMOS dataset [88]), keyword spotting (Google Speech Commands dataset [170]), and visual wake words detection (Visual Wake Words dataset [32]). We benchmark TinyNS on the first three tasks.

5.1.1 Dataset Splits and Pre-processing. We use the standard dataset splits and pre-processing functions provided by the benchmark suite. For CIFAR10, 50000 32×32×3 images are used for training, and 10000 images are used for testing. The dataset has 10 output classes. For ToyADMOS, 3600 and 400 non-anomalous sound samples from 4 toy cars mixed with ambient noise are used for training and validation, respectively, and 2500 anomalous and non-anomalous sound samples from the same 4 toy cars are used for testing. The pre-processor extracts the Mel-scaled power spectrogram from the raw WAVE files using 128 Mel bands, 5 frames, an FFT window length of 1024, and a hop length of 512. The spectrogram is converted to log Mel energy, clipped to keep the central portion, and concatenated with other frames to generate features. Each input tensor is a vector of length 640. For Google Speech Commands, the 100503 1-second keywords from 2618 speakers are divided into 85511, 10102, and 4890 utterances for training, validation, and testing, respectively. The dataset has 12 output classes. The pre-processor extracts the log Mel-frequency cepstral coefficient (MFCC) fingerprints from the raw 16 KHz WAVE files after decoding, volume scaling, random time-shifting (100 mS), and adding background noise to the raw audio data. The window size is 30 mS and the stride is 20 mS. 10 MFCC coefficients are used, resulting in each model input being a 49×10×1 tensor.

5.1.2 Model Backbones, Training Details, and Search Space Definition. For image recognition, we optimize the ResNet [77] backbone provided in the benchmark suite. Following the settings in the MLPerf Tiny v0.5 Benchmark [13] and state-of-the-art NAS frameworks for microcontrollers [14, 54, 60, 101, 103, 124], we train each candidate model for a fixed number of epochs of 500. While green AI advocates for training epochs to be considered as a hyperparameter [145] to be optimized, the additional hyperparameter may lead to a longer NAS convergence time from more candidate models being trained to achieve acceptable accuracy, minimizing the reduction in the total number of training epochs. In addition, TinyML neural architectures are either well-known (e.g., ResNet [77], MobileNets [79], or SqueezeNet [81]) or compact (e.g., FastGRNN [94], Bonsai [93], ProtoNN [74] or temporal CNN [97]), allowing the use of known and fixed training epochs or a small number of training epochs to achieve acceptable performance [136]. We use the Adam optimizer with a learning rate scheduler having an initial learning rate of 0.001 and decaying by a factor of 0.99 with each passing epoch. The batch size is 32, the loss is categorical cross-entropy, and the NAS error metric is training accuracy. The optimization hyperparameters include:

- Number of convolutional stacks: range (1, 5)
- Kernel size: [1, 3, 5, 7]
- Number of filters (initial layer): [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
- Use batch normalization: [True, False]
- Use activations: [True, False]

Just Accepted

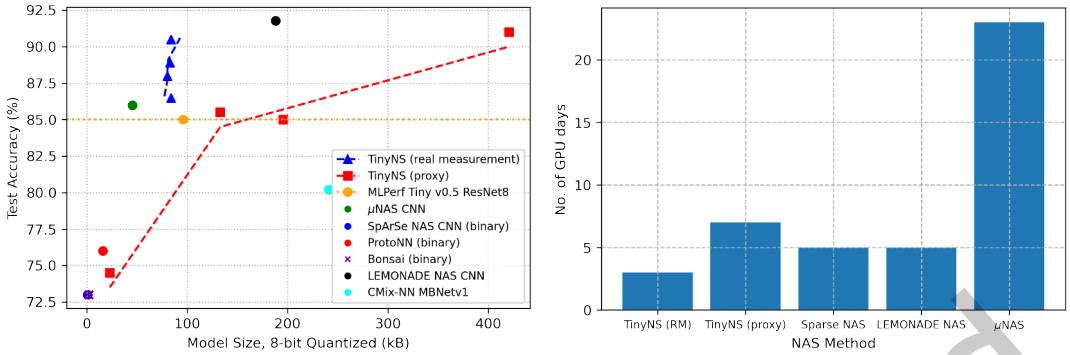

Fig. 6. (Left) Test accuracy versus model size of CIFAR10 ResNet models found by TinyNS versus competing CIFAR10 models designed for microcontrollers. (Right) NAS convergence time for TinyNS and competing microcontroller NAS frameworks on the CIFAR10 dataset.

For anomaly detection, we optimize a temporal convolutional autoencoder (denoted as 1D-CNN in the rest of the paper) backbone inspired by Thill *et al.* [159]. The encoder is a TCN [97, 162] without dilated kernels, followed by a 1D convolutional layer (linear activation) with a quarter and one-third of the number of filters and a kernel size of the TCN layer, respectively. The decoder includes the same layers but in reverse, followed by a fully-connected layer with 640 units and linear activation. Each candidate model is trained for 350 epochs, using the AMSGrad variant of the Adam optimizer with a learning rate of 0.001, β_1 of 0.9, β_2 of 0.999, and ϵ of 1e-8. The batch size is 1024, the loss is the mean squared error, and the NAS error metric is validation loss. The search space is as follows:

- Number of layers per stack: range (3, 8)
- Number of TCN stacks: [1, 2, 3]
- Number of filters in the TCN layers: range (3, 64)

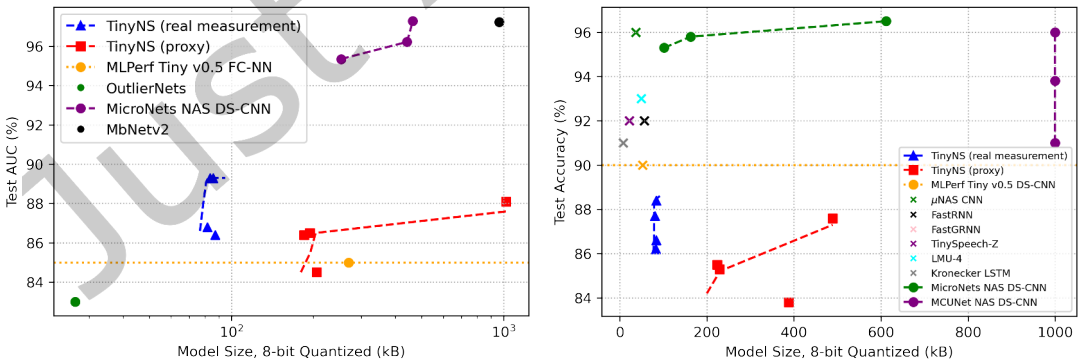

Fig. 7. (Left) Test AUC versus the model size of anomaly detection models (1D-CNN) found by TinyNS versus competing anomaly detection models designed for microcontrollers on the ToyADMOS dataset. (Right) Test accuracy versus the model size of keyword spotting models (TCN) found by TinyNS versus competing keyword spotting models designed for microcontrollers on the Google Speech Commands dataset.

Just Accepted

Table 2. Chosen ResNet model hyperparameters for each target hardware by TinyNS on the CIFAR10 dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash).

Device	Profiling	SRAM Usage (kB)	Latency (s) or FLOPS	Number of filters	Kernel size	Number of stacks	Batch normalization	Activations
F446RE (128, 512)	Real	107	0.58 (L)	10	5	4	True	True
	Proxy	95.8	12.9M (F)	4	7	4	True	True
L476RG (128, 1024)	Real	87.8	3.13 (L)	24	5	2	True	True
	Proxy	56.5	3.82M (F)	6	3	3	True	True
F746ZG (320, 1024)	Real	308	1.39 (L)	22	7	2	True	True
	Proxy	286	55.9M (F)	24	3	3	True	True
L4R5ZI_P (640, 2048)	Real	608	1.13 (L)	20	3	4	True	True
	Proxy	309	40.9M (F)	18	3	4	False	True

- Kernel size in the TCN layers: range (3, 16)
- Skip connections in TCN: [True, False]

For keyword spotting, we optimize a TCN, which can handle spatial and temporal features hierarchically without the explosion of hyperparameter count [97, 162]. The TCN layer is followed by a dense layer with 12 units and softmax activation. Each candidate model is trained for 60 epochs, using the Adam optimizer with a step function learning rate scheduler. The batch size is 1000, the loss is sparse categorical cross-entropy, and the NAS error metric is sparse categorical accuracy. The search space is as follows:

- Number of layers per stack: range (3, 8)
- Number of TCN stacks: [1, 2, 3]
- Number of filters in the TCN layers: range (2, 64)
- Kernel size in the TCN layers: range(2, 16)
- Skip connections in TCN: [True, False]
- Dilation factor choices: [1, 2, 4, 8, 16, 32, 64, 128, 256]

5.1.3 Overall Performance. Fig. 6 (Left) and Fig. 7 showcases the Pareto-optimal frontier generated by TinyNS versus competing frontiers and microcontroller models. TinyNS exceeds the benchmark accuracy by 4.3% and 5.5% for image recognition and anomaly detection, respectively, while consuming 1.14×-3.09× lower flash. For image recognition, TinyNS outperforms models generated SpArSe [59] and μNAS [101] by 4.5%-17.5% while taking 1.7×-7.7× lower convergence time (shown in Fig. 6 (Right)). Compared to LEMONADE [53], TinyNS provides 2.2× smaller models at the cost of 1.3% accuracy loss. TinyNS converges faster than gradient-based or evolutionary NAS due to two key properties. *Firstly*, TinyNS can eliminate infeasible candidate models in the search space without training, thanks to accurate hardware profiling using real microcontrollers during the search process. Proxies are unable to take into account the compiler runtime optimizations, and the dynamic overhead from RTOS, data stacks, and model interpreters. For all three tasks, the models generated by proxied TinyNS not only have sub-optimal accuracy (1.6%-5.5% lower) and flash usage (4.2× higher) compared to proxy less TinyNS but also have higher convergence time (2.3× higher). *Secondly*, the exploration-exploitation philosophy of the

Just Accepted

Table 3. Chosen 1D-CNN model hyperparameters for each target hardware by TinyNS on the ToyADMOS dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash).

Device	Profiling	SRAM Usage (kB)	Latency (s) or FLOPS	No. of filters	Kernel size	No. of layers per stack	No. of stacks	Skip connections
F446RE (128, 512)	Real	87.8	0.01 (L)	50	3	5	1	True
	Proxy	81.3	0.32M (F)	16	10	4	1	True
L476RG (128, 1024)	Real	88.2	0.06 (L)	38	10	6	1	True
	Proxy	62.0	0.24M (F)	26	3	5	1	True
F746ZG (320, 1024)	Real	288	0.01 (L)	42	4	4	3	True
	Proxy	78.1	0.31M (F)	30	4	3	1	True
L4R5ZI_P (640, 2048)	Real	608	0.03 (L)	63	3	5	1	True
	Proxy	444	1.77M (F)	57	6	4	2	True

Table 4. Chosen TCN model hyperparameters for each target hardware by TinyNS on the Google Speech Commands dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash).

Device	Profiling	SRAM Usage (kB)	Latency (s) or FLOPS	No. of filters	Kernel size	Dilations, no. of layers per stack	No. of stacks	Skip connections
F446RE (128, 512)	Real	106	0.31 (L)	51	9	[1,8,64,128], 4	2	True
	Proxy	77.8	21.6M (F)	27	9	[1,2,16,32,64,128], 6	2	True
L476RG (128, 1024)	Real	95.4	0.65 (L)	44	7	[1,2,4,8,16,128], 6	2	True
	Proxy	79.4	22.0M (F)	30	9	[1,2,8,16,128], 5	2	True
F746ZG (320, 1024)	Real	286	0.04 (L)	45	4	[1,4,16,64,128], 5	1	True
	Proxy	147	32.4M (F)	56	4	[1,4,8,64], 4	3	True
L4R5ZI_P (640, 2048)	Real	606	1.66 (L)	63	8	[1,4,8,16,32,64,128,256], 8	3	True
	Proxy	210	68.2M (F)	55	8	[1,16,128], 3	3	True

acquisition function, coupled with parallel search capabilities and the computationally-tractable sampling-based approach allows TinyNS to approach the global optimum without requiring evaluation of thousands of candidate architectures. Each model in the Pareto-frontier is generated within 10-50 iterations. For anomaly detection, TinyNS outperforms attention-based OutlierNets [2] by 6.3% and guarantees deployability over MobileNetv2 [140], but underperforms over MicroNets [14] models. We hypothesize that flattening the log MFCC in the 1D-CNN backbone loses spatial correlation across the feature coefficients. This phenomenon also generates sub-optimal TinyNS models for keyword spotting, failing to cross the benchmark accuracy of 90% as shown in Fig. 7 (Right). This showcases the importance of performing NAS not just over a single model backbone, but over multiple model backbones. In Section 5.3, we showcase how TinyNS operating on a search space with multiple models can generate models with the lowest flash usage and highest accuracy. Regardless, given an ideal model backbone, TinyNS can generate models with the highest accuracy and guaranteed deployability within a few evaluations without requiring expensive training infrastructure.

5.1.4 Architectural Adaptation Based on Resource Availability. Table 2, Table 3, and Table 4 show the hyperparameters of the model backbones for the three tasks generated by TinyNS for four different STM32 microcontrollers

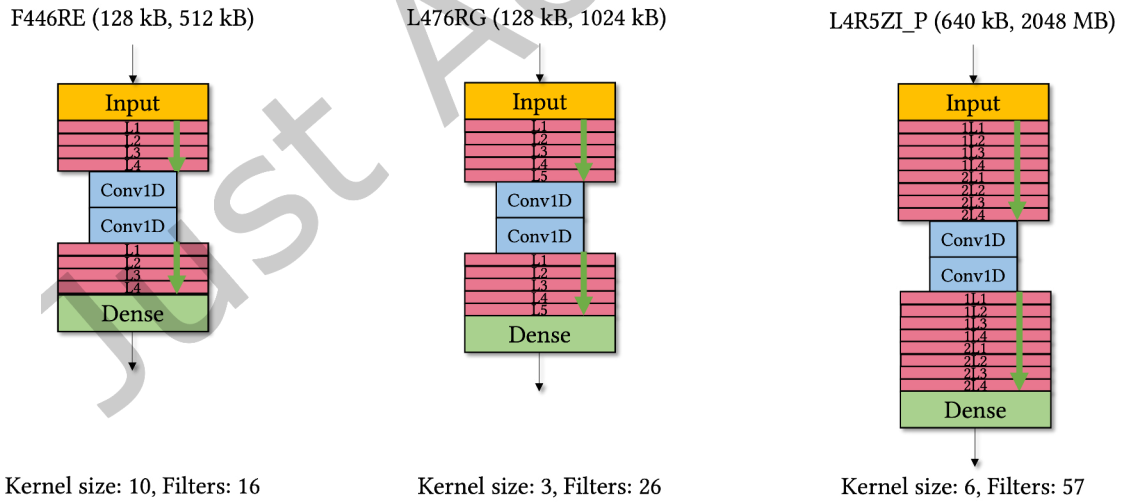

Fig. 8. Architectural adaptation and device capability exploitation by TinyNS on the ToyADMOS dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash). ${}_jL_i$ refers to i^{th} layer of the 1D-CNN in the jth stack.

Just Accepted

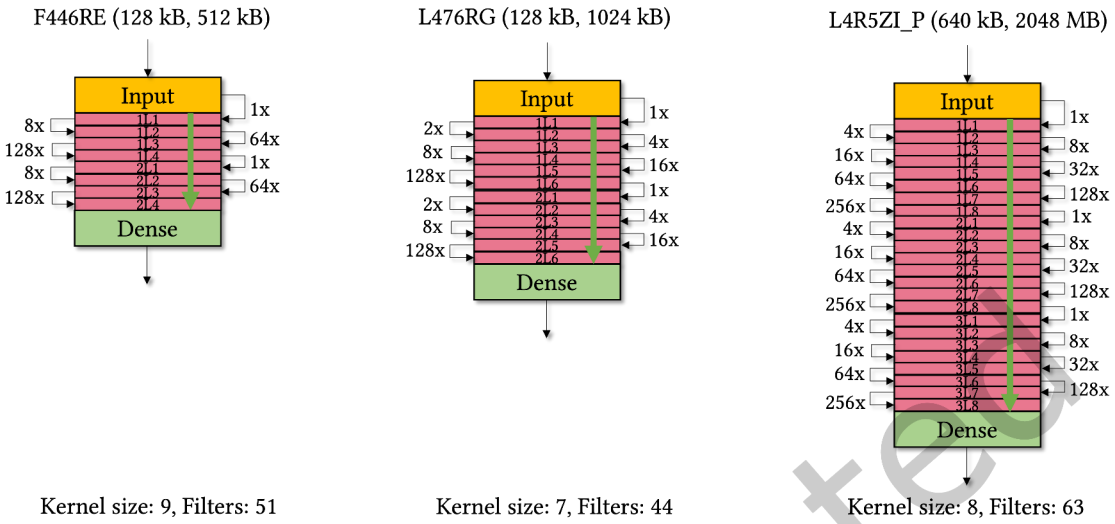

Fig. 9. Architectural adaptation and device capability exploitation by TinyNS on the Speech Commands dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash). ${}_jL_i$ refers to i^{th} layer of the TCN in the jth stack.

with varying SRAM and flash limits. In general, as the device capabilities increase, TinyNS generates models that have higher FLOPS, and higher SRAM and flash usage. Instead of providing the smallest model with the highest accuracy, TinyNS adapts hyperparameters such as the number of kernels, size of kernels, and the number of convolutional stacks with increasing device capabilities to maximize accuracy. Fig. 8 and Fig. 9 show visual examples of such architectural adaptation for three of the four microcontrollers. As the SRAM and flash capacity increases, TinyNS automatically adjusts the number of layers per stack, the number of stacks, the kernel size, and the number of filters depending on an increase in SRAM or flash. For example, a model with more parameters but a smaller kernel size and filter count are likely to benefit from an increase in flash but no change in SRAM. Likewise, when dilated convolutions are used, TinyNS assigns a small dilation factor to earlier layers and a large dilation factor in later layers when it cannot increase the number of layers due to resource limits. This allows a TCN with a limited layer count to have the same receptive field (albeit less fine-grained) as a TCN with more layer count, capturing both short-term local context and long-term global time-series inter-dependencies. Table 2, Table 3, and Table 4 further showcase the problem with proxies as opposed to real-hardware profiling. These models have a higher number of parameters but a lower number of filters and kernel size than proxy-less models. Since proxies are unable to take into account compiler optimizations, the generated models underestimate the available SRAM and overestimate the flash usage, yielding models with poor accuracy.

5.1.5 Convergence Time of Proxyless versus Proxied TinyNS. Fig. 10 shows the number of iterations needed to reach the best optimization score for proxy less and proxied TinyNS for all three tasks. Mango allows both random initialization and an initial set of evaluation points to warm up the optimizer. The user can either customize the initial evaluation points to guide the optimization process or choose random sampling to mitigate randomness effects [138]. We showcase the results for an average from 3 independent runs for each algorithm to account for the effect of randomness. For both profiling techniques, tighter hardware constraints (lower SRAM and flash capacities) equate to more iterations required for convergence. However, proxy less TinyNS converges 3.2×-12.6×

Just Accepted

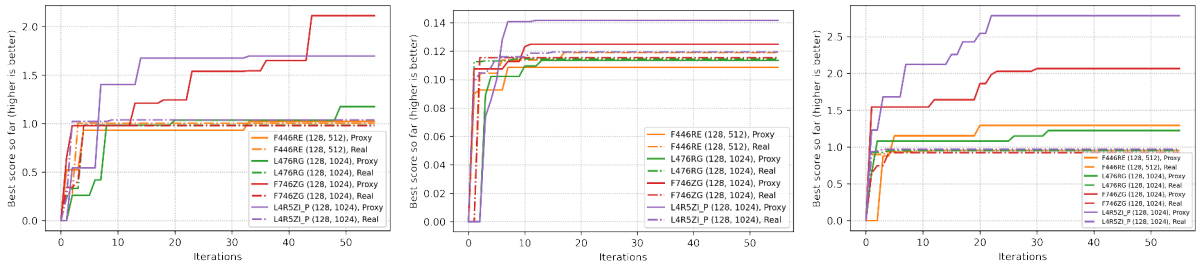

Fig. 10. Convergence iterations required for proxy less and proxied TinyNS. (Left) CIFAR10, (Center) ToyADMOS, (Right) Google Speech Commands. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash). Note that a higher score for proxied TinyNS does not necessarily guarantee deployability, while the highest score for proxy less TinyNS guarantees deployability on the target microcontroller.

faster to the highest performing model compared to proxied TinyNS. Intuitively, platform-in-the-loop should be slow while analytical proxies should be fast, as real measurements have compilation time and profiling time overhead and are not immediate. However, since proxies are inaccurate and do not reflect the execution level dynamics, more infeasible model candidates are trained rather than discarded, wasting valuable computing time and increasing the search completion time. In our evaluation, we found the platform-in-the-loop approach to be 50% faster than using proxies for hardware profiling. Even though proxied TinyNS achieves a higher score than proxy less TinyNS, the deployability of models generated by proxied TinyNS is not guaranteed due to high flash consumption. Further, we have seen earlier that these models do not fully exploit the SRAM capabilities and have lower accuracy than proxy-less models. The increased score achieved by proxied TinyNS is contributed by model candidates with a high flash footprint.

5.2 Optimization of Features and Neural Weights (Symbolic Neuro Symbolic)

In this case study, we showcase how TinyNS provides the best combination of features and neural network hyperparameters for various target hardware.

5.2.1 Dataset and Task Description. We use the UCI-HAR dataset [6] for this case study. The task is to classify 6 human activities (walking, walking upstairs, walking downstairs, sitting, laying, and standing) from a single waist-mounted x-axis accelerometer data sampled at 50 Hz from 30 volunteers. The dataset is split with leave-7 out, i.e., data from 21 volunteers are in the training set, and data from the rest 7 volunteers are in the test set. As suggested by the dataset authors, we use a window size of 128 (2.56 s) with a stride of 64. 10% of the training data is used for validation.

Table 5. Chosen features (shaded) for each target hardware for neurosymbolic optimization of input feature choices and model backbone. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash).

Device	Features											
ISPU (8, 32)	Mean	IQR	Maximum	Median	Variance	MAD	Abs. Energy	Entropy	Peak-to-Peak	FFT Mean Coeff.	Fundamental Frequency	Max. Power Spectrum
F446RE (128, 512)	Mean	IQR	Maximum	Median	Variance	MAD	Abs. Energy	Entropy	Peak-to-Peak	FFT Mean Coeff.	Fundamental Frequency	Max. Power Spectrum
L476RG (128, 1024)	Mean	IQR	Maximum	Median	Variance	MAD	Abs. Energy	Entropy	Peak-to-Peak	FFT Mean Coeff.	Fundamental Frequency	Max. Power Spectrum
F746ZG (320, 1024)	Mean	IQR	Maximum	Median	Variance	MAD	Abs. Energy	Entropy	Peak-to-Peak	FFT Mean Coeff.	Fundamental Frequency	Max. Power Spectrum
L4R5ZI_P (640, 2048)	Mean	IQR	Maximum	Median	Variance	MAD	Abs. Energy	Entropy	Peak-to-Peak	FFT Mean Coeff.	Fundamental Frequency	Max. Power Spectrum

Just Accepted

Table 6. Chosen model hyperparameters for each target hardware for neurosymbolic optimization of input feature choices and model backbone. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash).

Device	Number of filters	Kernel size	Number of stacks	Dilations, number of layers per stack	Skip connections
ISPU (8, 32)	3	5	1	[1,2,4,32,64,128], 6	False
F446RE(128, 512)	5	3	3	[1,2,16,32,128], 5	False
L476RG (128, 1024)	7	7	2	[1,2,4,32,128], 5	False
F746ZG (320, 1024)	3	10	3	[1,2,8,16,32], 5	True
L4R5ZI_P (640, 2048)	29	6	1	[1,4,16,64,128], 5	True

5.2.2 Model Backbones, Training Details, and Search Space Definition. The model backbone consists of a TCN. The TCN layer is followed by a dense layer with 6 units and softmax activation. Each candidate model is trained for 150 epochs, using the Adam optimizer with default parameters. The loss is categorical cross-entropy, and the NAS error metric is validation accuracy. The search space for the model is as follows:

- Number of layers per stack: range (3, 8)
- Number of TCN stacks: [1, 2, 3]
- Number of filters in the TCN layers: range (3, 64)
- Kernel size in the TCN layers: range(3, 16)
- Skip connections in TCN: [True, False]
- Dilation factor choices: [1, 2, 4, 8, 16, 32, 64, 128]

The feature space consists of 12 features listed in Table 5. There are 6 statistical features, 3 temporal features, and 3 spectral features to choose from. The search space for the features is defined using the binary mask technique shown in Section 4.1.

5.2.3 Target Hardware. We perform neurosymbolic optimization for the same four microcontrollers from Section 5.1. In addition, we also perform optimization for an integrated sensor processing unit (ISPU) from STMicroelectronics. The ISPU is an ultra-low-power 10 MHz 32-bit RISC processor (architecture: STRED) embedded within the LSM6DSOIS and ISM330IS 6DoF MEMS inertial sensor. The processor uses a proprietary version of TFLM (called q2c) to run on-chip neural networks without needing a power-hungry microcontroller in the loop and uses the STRED/ISPU toolchain to compile C++ programs. The processor has 8kB SRAM and 32kB flash [107].

5.2.4 Overall Performance. Fig. 11 (Left) shows the Pareto-frontier generated by TinyNS versus using all the features and directly operating on the raw accelerometer data. On average, TinyNS provides up to 2% improvement in accuracy over the same model operating on raw data or operating on all the features. Extracting all the features is computationally intensive (especially for the ISPU) while operating on raw data without a gyroscope or magnetometer or other axes of the accelerometer results in performance degradation. Table 5 and Table 6 show the chosen features and model hyperparameters for each target hardware. Surprisingly, TinyNS learns to pick only the most important features (e.g., peak-to-peak, FFT mean coefficients, entropy, and variance) for the ISPU and the microcontrollers with the lowest SRAM and flash capacities. These features are well-known to have the highest effect on classifier performance in human activity recognition literature [8, 168]. As the device capabilities increase, TinyNS selects other features in the feature set. TinyNS also performs architectural adaptation and device capability exploitation seen in Section 5.1, increasing the number of filters, the kernel size, and the number of stacks of the model candidates. To prevent exploding and vanishing gradient problem, TinyNS learns to add skip connections to deeper TCN models. The SRAM usage and FLOPS count of the models steadily increase with increasing device capabilities as shown in Fig. 11 (Center) and Fig. 11 (Right). The median SRAM saturation is around 20%, with the saturation being higher for devices with higher flash availability, showing full resource exploitation by TinyNS for each target hardware. Overall, choosing the best synergy of features and

Just Accepted

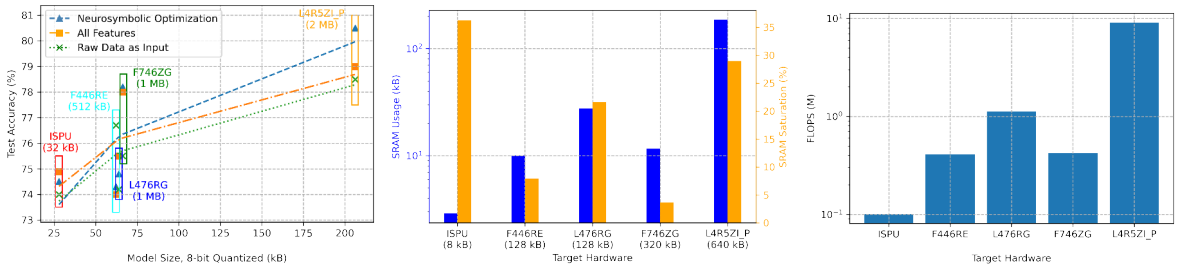

Fig. 11. (Left) Flash usage of models found via neurosymbolic optimization of features and model hyperparameters. The accuracy of the said models operating on all features and directly on the raw data is also shown. Flash limits of the target hardware are shown in parentheses. (Center) SRAM usage of models found via neurosymbolic optimization of features and model hyperparameters. SRAM limits of the target hardware are shown in parentheses. (Right) FLOPS count of models found via neurosymbolic optimization of features and model hyperparameters.

model hyperparameters makes it possible to run models on extremely resource-constrained platforms beyond microcontrollers like the ISPU.

5.3 Fall Detection under 2 kB and Activity Recognition (Symbolic Neuro Symbolic)

In this case study, we showcase how TinyNS picks the best model backbone (neural or non-neural) and its hyperparameters out of a zoo of TinyML model backbones.

5.3.1 Dataset and Task Description. We use the Auritus dataset [135] for this case study. There are two tasks. The first task is to distinguish between fall and non-fall activities under a 2 kB memory constraint (suitable for ISPU) using an ear-mounted 6DoF inertial measurement unit called earable. The second task is to classify 9 human activities (walking, jogging, standing, sitting, laying, turning left, turning right, jumping, and falling). The dataset is sampled at 100 Hz from 45 volunteers. We split the dataset in two ways: split with no unseen participants and split with leave-1 out. In the first splitting technique, we use 80% of the data for training, 10% for validation, and 10% for testing. In the second splitting technique, we perform 10-way cross-validation by leaving a random participant out of the training set. The data from the chosen 44 participants are split 90:10 for training: validation. The stride was set to 0.5 seconds and the window size was optimized as a hyperparameter.

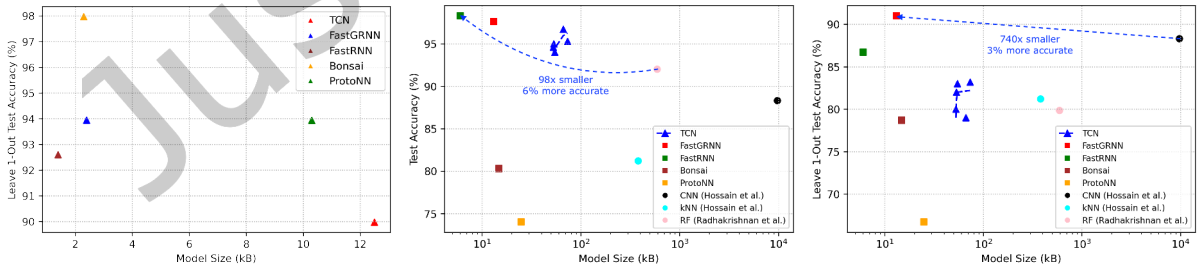

Fig. 12. (Left) Highest performing models found by TinyNS for earable fall detection under 2 kB memory constraint when optimizing several model backbones. (Center and Right) Test accuracy and leave 1-out test accuracy of highest performing models found by TinyNS versus state-of-the-art earable activity detection classifiers when optimizing several model backbones. The TCN backbone is optimized for 5 different target hardware (eSense earable, F446RE, L476RG, F407VET6, and F746ZG).

Just Accepted

Table 7. Chosen model hyperparameters for each backbone found by TinyNS when optimizing several model backbones for earable activity detection. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash).

Model Backbone	Device	Hyperparameters			
		Number of filters	**Kernel size**	**Dilations, number of layers per stack**	**Skip connections**
TCN	F446RE (128, 512)	18	2	[2, 4, 8, 16, 32, 64, 128, 256], 8	Yes
	L476RG (128, 1024)	13	7	[1, 4, 16, 32], 4	No
	eSense earable (128, 16000)	15	2	[1, 2, 4, 8, 32, 128, 256], 7	Yes
	F407VET6 (192, 512)	17	3	[2, 4, 32, 128, 256], 5	No
	F746ZG (320, 1024)	21	2	[2, 8, 16, 64, 128, 256], 6	Yes
FastGRNN	None (hardware-agnostic)	**Hidden Units**			
		50			
FastRNN		32			
Bonsai		**Projection Dimension**		**Sigmoid Parameter**	**Depth**
		22		1.0	3
ProtoNN		**Projection Dimension**		**Prototypes**	γ
		70		70	0.004

5.3.2 Model Backbones, Training Details, Target Hardware, and Search Space Definition. We set 5 different model backbones (3 neural, 2 non-neural) in the search space, each with its own set of optimization hyperparameters:

- TCN (neural) [97, 162] - number of filters in the TCN layers: range (2, 64); kernel size in the TCN layers: range (2, 16); skip connections in TCN: [True, False]; the number of layers per stack: range (3,8); dilation factor choices: [1,2,4,8,16,32,64,128,256].
- FastGRNN (neural) [94] - number of hidden units: range (20, 60).
- FastRNN (neural) [94] - number of hidden units: range (20, 60).
- Bonsai (non-neural) [93] - projection dimension: range (10, 70); sigmoid parameter: uniform (1.0, 4.0); depth: range(1, 6).
- ProtoNN (non-neural) [74] - projection dimension: range (10, 70); γ: uniform (0.0015, 0.05); the number of prototypes: range (10, 70).

In addition, for all the models, the search space for the window size is [1, 2, 3, 5] seconds. For TCN, we generate Pareto-frontier for 4 different STM32 microcontrollers (F446RE, L476RG, F407VET6, and F746ZG) and the Qualcomm CSR8670 microcontroller found inside the earable. We use proxies for profiling the CSR processor as it does not support firmware modification. For the STM32 microcontrollers, we use platform-in-the-loop profiling. For Bonsai and ProtoNN, we apply five features on the accelerometer and gyroscope vector sums: maxima, minima, range, variance, and standard deviation. The rest of the models operate directly on the raw data. The loss is categorical cross-entropy for all the models, except for Bonsai, which uses multi-class hinge loss. The NAS error metric is validation accuracy for TCN and training accuracy for the rest of the classifiers.

5.3.3 Overall Results. Fig. 12 summarizes the accuracy and model size for the highest performing models for each of the 5 backbones against competing models, while Table 7 shows the hyperparameters of the said models. TinyNS achieves state-of-the-art improvement in both accuracy and model size reduction, providing earable activity detection models that are 98×-740× smaller yet 3%-6% more accurate than competing models. The activity recognition models are as small as 6-13 kB. Further, TinyNS achieves 98% earable fall detection accuracy with a model as small as 2.3 kB. The case study illustrates the importance of optimizing several model backbones rather than a single backbone, particularly in unseen domains void of expert knowledge. Notably, models with more parameters do not necessarily provide higher accuracies. Appropriate architectural encodings make it possible to achieve the same or better accuracy with a lower parameter count (e.g., a CNN is likely to outperform a fully-connected neural network due to the ability to extract spatial relations, even though the latter may have more parameters). Even if one architecture performs poorly, the search algorithm would have other architectures

Just Accepted

Table 8. Chosen object detector and tracking filter hyperparameters for CenterNet algorithm under different size limits.

Constraint	Flash Usage (MB)	Performance		Model hyperparameters				Filter hyperparameters (thresholds)	
		MOTA	IDF1	Kernel size	Stack count	Head convolution value	Activations	Rendering	Confidence
Handcrafted (none)	238	36.5	55.0	1	1	128	True	0.4	0.5
250 MB limit	238	36.1	54.6	1	1	150	True	0.3	0.4
500 MB limit	270	38.0	57.2	9	1	100	False	0.7	0.5

to choose from. Thereby, exploring various architectures is important for squeezing highly performant models beyond microcontrollers, such as the ISPU.

5.4 Optimization of Neural Detector Weights and Symbolic Object Tracker (Neuro→Symbol)

In this case study, we show the ability of TinyNS to jointly optimize neural and symbolic modules, where the symbolic module makes high-level reasoning over the neural outputs.

5.4.1 Dataset and Task Description. We use the MOT17 dataset [113] for this case study. The goal is to develop multiple people tracking algorithms from a single camera feed under model size constraints. The dataset is pre-processed using the ByteTrack library [178].

5.4.2 Model Backbones and Search Space Definition. We use the ByteTrack library [178] to implement the CenterNet algorithm [179], which was discussed in Section 4.2. Each candidate model is trained for 70 epochs with a batch size of 16. The search space for the ResNet + Deformable Convolutional Network and the tracking filter are:

- Number of convolutional stacks: range (1, 5)
- Kernel size: [1, 3, 5, 7, 9,..., 23]
- Layer-wise activations: [True, False]
- Head convolutional value: [50, 100, 150,..., 300]
- Rendering threshold: linspace (0.1, 0.9, 9)
- Confidence threshold: linspace (0.1, 0.9, 9)

5.4.3 Overall Results. Table 8 shows the performance, resource usage, and hyperparameters of the CenterNet algorithm under hard memory constraints compared to the handcrafted algorithm with default hyperparameters. Note that the MOTA and IDF1 for all the models are low as no pre-training or fine-tuning on additional data is performed. The 250 MB model achieves MOTA and IDFf within 1% of the handcrafted model, while the 500 MB model exceeds the MOTA and IDF by 4.5%. The case study showcases that TinyNS can achieve the performance of neurosymbolic models hand-tuned using hundreds of human hours automatically, and even exceed the performance when device constraints relax. Compared to a human designer, TinyNS can find models whose hyperparameters may be counter-intuitive (e.g., reducing the head convolutional value from 150 to 100 and removing layer-wise activations for the 500 MB model) but provide superior performance.

5.5 Improving Adversarial Robustness of TinyML Models (Neuro ∪ Compile (Symbolic))

In this case study, we showcase how TinyNS can find model architectures that follow some coveted architecture-dependent constraints.

5.5.1 Dataset and Task Description. We use the Auritus dataset in this case study (the same dataset used in Section 5.3). The goal and the dataset splits are the same as that in Section 5.3, except that now we want TinyML models that not only have the highest accuracy within the device constraints but are also adversarially robust to white-box attacks (discussed in Section 4.3).

Just Accepted

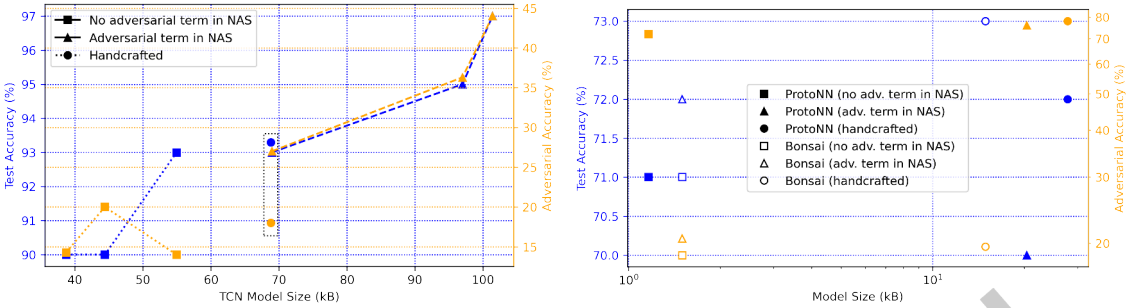

Fig. 13. (Left) Test accuracy, adversarial accuracy, and model size of TCN backbones for three different target hardware (F446RE, L476RG, and F746ZG). (Right) Test accuracy, adversarial accuracy, and model size for ProtoNN and Bonsai backbones. For all three model backbones, the results are shown for NAS with adversarial robustness term, NAS without adversarial robustness term, and handcrafted models.

5.5.2 Model Backbones, Training Details, Target Hardware, and Search Space Definition. We use the TCN, Bonsai, and ProtoNN backbones using the same model search space defined in Section 5.3. The window size is fixed to 5 seconds. For the TCN, we generate Pareto-frontier for F446RE, L476RG, and F746ZG. The rest of the training details are the same as Section 5.3.

5.5.3 Overall Results. Fig. 13 shows the test accuracy, adversarial accuracy, and the model size of TinyNS generated models with adversarial robustness optimization, versus handcrafted models and models generated by TinyNS with no adversarial robustness optimization. TinyNS generates models that are 1%-26% (9% on average more adversarially robust than competing models while maintaining or exceeding the accuracy on the main task. This comes at the cost of increased model size, albeit well within the flash constraints of the target hardware. This is because larger models have more parameters and are therefore more robust to small input perturbations. In addition, models generated by TinyNS without adversarial robustness optimization are more sensitive to small perturbations compared to handcrafted models. This is probably due to high loss smoothness and low gradient variance in the loss contour of NAS-generated models [117].

5.6 Physics-Aware Neural Inertial Localization (Neuro ∪ Compile (Symbolic))

In this case study, we showcase how TinyNS can force models to follow some coveted constraints via the inclusion of physics channels.

5.6.1 Dataset and Task Description. We use 5 inertial odometry datasets spanning 4 applications for this case study. These include two datasets for human tracking namely OxIOD [29] and RoNIN [78], AQUALOC [61] unmanned underwater vehicle (UUV) tracking, EuRoC MAV [22] undermanned aerial vehicle (UAV) tracking, and the GunDog [73] animal tracking. The split information for all the datasets is shown in Table 9. The goal is to

Table 9. Window size, stride, training-validation-test splits, and training epochs used in the inertial odometry datasets

Dataset	Sampling Rate (Hz)	Window Size	Stride	Splits (Tr, Val, Te) (%)	Model Epochs
OxIOD	100	200	10	85, 5, 10	900
RoNIN	200	400	20	70, 5, 25	900
AQUALOC	200	400	20	80, 5, 15	300
EuRoC MAV	200	50	5	80, 10, 10	300
GunDog	40	10	10	45*, 5*, 50	300

* Training trajectory split into 2 parts for train and validation splits.

Just Accepted

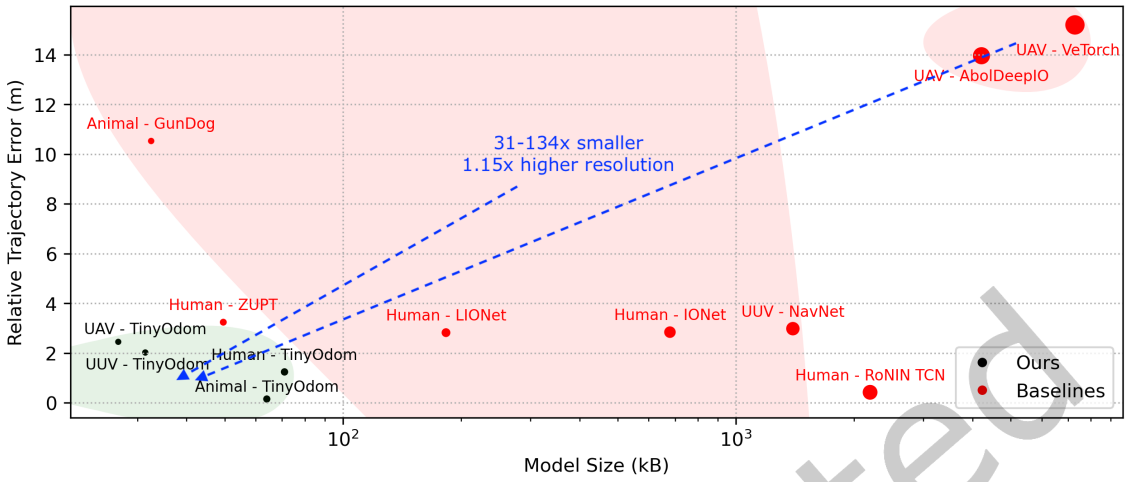

Fig. 14. Odometric resolution of physics-aware neurosymbolic-inertial odometry models (TinyOdom) found via neurosymbolic architecture search, versus state-of-the-art handcrafted neural and symbolic models for tracking humans, animals, unmanned underwater vehicles (UUV), and unmanned aerial vehicles (UAV).

train a model to predict the position of an object using inertial sensor data without GPS updates while mitigating position explosion error innate in inertial sensors due to bias and drift. The model must be able to detect when sufficient translational movement has not happened, thereby not updating the position (physics-aware).

5.6.2 Model Backbones, Training Details, Target Hardware, and Search Space Definition. We use a TCN backbone. The outputs of the TCN are reshaped, pooled, and flattened, and then fed to a 32-unit dense layer with linear activations. The loss is a mean-squared error, the optimizer is Adam with a learning rate of 0.001, and the NAS error metric is validation loss. The search space for the model is as follows:

- Number of layers per stack: range (3, 8)
- Dropout: uniform (0.0, 1.0)
- Normalization: [Weight, Layer, Batch]
- Number of filters in the TCN layers: range (2, 64)
- Kernel size in the TCN layers: range (2, 16)
- Skip connections in TCN: [True, False]
- Dilation factor choices: [1, 2, 4, 8, 16, 32, 64, 128, 256]

We generate the Pareto-frontier for the 4 STM32 microcontrollers outlined in Section 5.3.

5.6.3 Overall Results. Fig. 14 shows the odometric resolution of models found by TinyNS (called TinyOdom) versus handcrafted state-of-the-art neural and symbolic models. TinyNS models outperform purely neural and purely symbolic models on all four applications by 1.15× while being 31×-134× smaller. In other words, TinyNS not only exceeds the resolution of human-designed neural and symbolic models but also ensures the deployability of the models on microcontrollers. The superior performance is possible partly due to the inclusion of the physics channel, which improves the resolution by 1.1× on average, as showcased in Table 10. The physics channel ensures that lightweight and under-parameterized models such as those generated by TinyNS are able to follow the underlying system physics as well as over-parametrized baselines. Fig. 15 visualizes the architectural adaptation

Just Accepted

Table 10. Effect of removing the physics channel of proposed neural-inertial odometry models on 3 inertial odometry datasets.

Dataset	Absolute Trajectory Error (m)		Relative Trajectory Error (m)	
	With Physics	Without Physics	With Physics	Without Physics
OxIOD	3.35	3.86	0.90	1.24
AQUALOC	3.36	3.71	2.44	2.53
Agrobot (Phase 1)	7.85	9.13	1.10	1.33

and device capability exploitation by TinyNS when generating the Pareto-frontier. As observed in previous sections, TinyNS changes the appropriate hyperparameters to improve device resource usage and resolution.

5.7 Neural-Kalman Sensor Fusion (Symbolic[Neuro])

In this case study, we showcase how TinyNS can optimally combine a neural system model with a symbolic measurement model using Kalman filter theory.

5.7.1 Dataset and Task Description. We use the AgroBot dataset [50] in this case study. The goal is to perform precision localization of an agricultural robot using neural inertial localization, with intermittent GPS updates. The underlying system must fuse the smoothness and short-term resolution of neural inertial localization with the long-term precision of GPS. The dataset contains 6.5 hours and 4.5 km of inertial and GPS data. We used 80% of the dataset for training and 20% for testing.

5.7.2 Model Backbones, Training Details, Target Hardware, and Search Space Definition. We used the same model backbone and search space outlined in Section 5.6. In addition, we optimize noise hyperparameters in the Kalman filter Allan variance matrix:

- accelerometer noise variance: linspace (0, 1, 10000)
- gyroscope noise variance: linspace (0, 1, 10000)

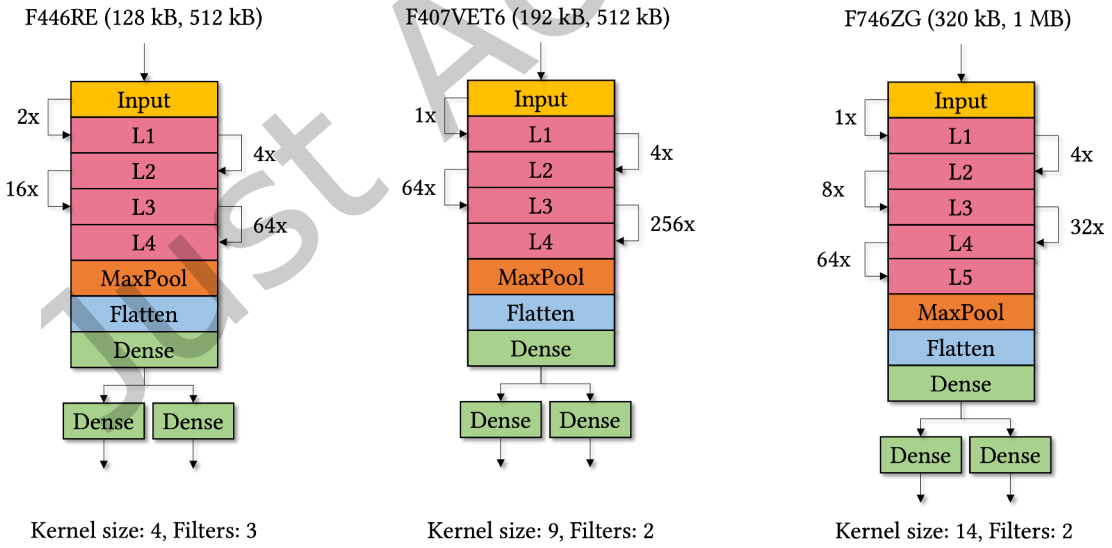

Fig. 15. Architectural adaptation and device capability exploitation by TinyNS on the AQUALOC dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash). L_i refers to i^{th} layer of the TCN.

Just Accepted

Table 11. Odometric resolution and flash usage of proposed neural-Kalman GPS-INS fusion for locating precision agricultural robots versus state-of-the-art neural and symbolic approaches.

Paradigm	Method	Code Size (MB)	Absolute Trajectory Error (m)	Relative Trajectory Error (m)
Neural	IONet [28]	1.71	5.58 | 10.1	0.92 | 0.57
	L-IONet [29]	0.55	8.11 | 18.6	0.91 | 1.40
	AbolDeepIO [55]	12.5	7.24 | 20.5	0.96 | 0.93
	VeTorch [62]	29.6	2.86 | 15.6	0.44 | 0.84
Symbolic	UKF-M INS+GPS [21]	0.192	5.50	0.49
	EKF INS+GPS [125]	0.077	3.31	0.58
	GPS only	-	1.89	0.42
Neurosymbolic	**Ours (no GPS, w physics)**	1.10	1.76 | 9.12	0.28 | 1.55
	Ours (w GPS, w physics)	1.12	1.02 | 1.81	0.28 | 0.64

first term in the error is on seen trajectory, second term is on unseen trajectory; single term is on unseen trajectory

- magnetometer noise variance: linspace (0, 1, 10000)

The batch size, optimizer, and training epochs were set to 256, Adam (learning rate: 0.001), and 3000, respectively. The NAS error metric is the absolute trajectory error during training. The model size constraint is set to 2 MB.

5.7.3 Overall Results. Table 11 outlines the performance of TinyNS generated neurosymbolic model versus human-engineered state-of-the-art neural and symbolic approaches of localization. Compared to competing neural models, TinyNS model without GPS lowers model size and absolute trajectory error by 1.5× - 27× and 1.4× - 5.8×, respectively. Compared to competing symbolic models, TinyNS model with GPS lowers absolute trajectory error and relative trajectory error by 1.2× - 11× and 1.1× - 3.8×. The neural-Kalman fusion exploited by TinyNS combines the long-term precision of symbolic models with the short-term robustness and resolution of neural networks within the 2 MB limit set forth in this case study.

6 CONCLUSION, LIMITATIONS, AND FUTURE WORK

Neurosymbolic AI provides a pathway for making context-aware, physics-aware, robust, interpretable, and performant AI systems. TinyNS provides a stepping stone in automating the deployment of neurosymbolic frameworks onto ultra resource-constrained IoT devices like microcontrollers and ISPUs. The Bayesian optimization formulation provides an inexpensive method to iterate over complex neurosymbolic search spaces, providing Pareto-optimal models depending upon resource availability. GP-UCB and hard thresholding policy allow fine-grained search space exploration and exploitation and improved convergence time. Through TinyNS, we have showcased state-of-the-art performance in various unseen applications. Several lessons, limitations, and directions for future work for our framework are as follows:

- There is an absence of general-purpose parsers, lexers, and visitors needed to realize symbolic program graphs on microcontrollers. We need tools that are similar to TFLM but for parsing program decision trees.
- The process of porting a custom symbolic layer from TF to TFLM is convoluted, with support for mostly the layers available in TFL. To run such custom layers, a user-friendly framework for the automatic porting of custom TF operators to TFLM is necessary.
- Our framework only supports TFLM so far for model parsing. However, there are other inference engines for which support must be added.

ACKNOWLEDGMENTS

The research reported in this paper was sponsored in part by the Air Force Office of Scientific Research (AFOSR) under Cooperative Agreement FA9550-22-1-0193; the IoBT REIGN Collaborative Research Alliance funded by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196; the NIH mHealth Center for Discovery, Optimization and Translation of Temporally-Precise Interventions (mDOT) under award

Just Accepted

1P41EB028242; the National Science Foundation (NSF) under awards # 1705135 and 1822935. and, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the AFOSR, ARL, DARPA, NIH, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Saad Abbasi, Mahmoud Famouri, Mohammad Javad Shafiee, and Alexander Wong. 2021. OutlierNets: highly compact deep autoencoder network architectures for on-device acoustic anomaly detection. *Sensors* 21, 14 (2021), 4805.

[3] Kareem Ahmed, Tao Li, Thy Ton, Quan Guo, Kai-Wei Chang, Parisa Kordjamshidi, Vivek Srikumar, Guy Van den Broeck, and Sameer Singh. 2022. PYLON: A PyTorch framework for learning with constraints. In *NeurIPS 2021 Competitions and Demonstrations Track*. PMLR, 319–324.

[4] Norah N Alajlan and Dina M Ibrahim. 2022. TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications. *Micromachines* 13, 6 (2022), 851.

[5] Mona Alshahrani, Mohammad Asif Khan, Omar Maddouri, Akira R Kinjo, Núria Queralt-Rosinach, and Robert Hoehndorf. 2017. Neuro-symbolic representation learning on biological knowledge graphs. *Bioinformatics* 33, 17 (2017), 2723–2730.

[6] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra Perez, and Jorge Luis Reyes Ortiz. 2013. A public domain dataset for human activity recognition using smartphones. In *Proceedings of the 21th international European symposium on artificial neural networks, computational intelligence and machine learning*. 437–442.

[7] Gianluca Apriceno, Andrea Passerini, and Luciano Serafini. 2021. A Neuro-Symbolic Approach to Structured Event Recognition. In *28th International Symposium on Temporal Representation and Reasoning (TIME 2021)*.

[8] Ferhat Attal, Samer Mohammed, Mariam Dedabrishvili, Faicel Chamroukhi, Latifa Oukhellou, and Yacine Amirat. 2015. Physical human activity recognition using wearable sensors. *Sensors* 15, 12 (2015), 31314–31338.

[9] The GPyOpt authors. 2016. GPyOpt: A Bayesian Optimization framework in python. http://github.com/SheffieldML/GPyOpt.

[10] The Skopt authors. 2016. Skopt: scikit-optimize. https://scikit-optimize.github.io/.

[11] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing neural network architectures using reinforcement learning. *International Conference on Learning Representations (ICLR)* (2017).

[12] Suresh Balakrishnama and Aravind Ganapathiraju. 1998. Linear discriminant analysis-a brief tutorial. *Institute for Signal and Information Processing* 18, 1998 (1998), 1–8.

[13] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. 2021. MLPerf Tiny Benchmark. *Advances in Neural Information Processing Systems* (2021).

[14] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems* 3 (2021), 517–532.

[15] David M Beazley. 1996. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996-Volume 4*. 15–15.

[16] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2010. Cython: The best of both worlds. *Computing in Science & Engineering* 13, 2 (2010), 31–39.

[17] Dimitri Bertsekas. 2016. *Nonlinear Programming*. Vol. 4. Athena Scientific.

[18] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 60–70.

[19] M Lourdes Borrajo, Bruno Baruque, Emilio Corchado, Javier Bajo, and Juan M Corchado. 2011. Hybrid neural intelligent system to predict business failure in small-to-medium-size enterprises. *International journal of neural systems* 21, 04 (2011), 277–296.

[20] Antoine Bosselut, Hannah Rashkin, Maarten Sap, Chaitanya Malaviya, Asli Celikyilmaz, and Yejin Choi. 2019. COMET: Commonsense Transformers for Automatic Knowledge Graph Construction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4762–4779.

[21] Martin Brossard, Silvere Bonnabel, and Jean-Philippe Condomines. 2017. Unscented Kalman filtering on Lie groups. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2485–2491.

Just Accepted

[22] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. 2016. The EuRoC micro aerial vehicle datasets. *The International Journal of Robotics Research* 35, 10 (2016), 1157–1163.
[23] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations*.
[24] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. 2020. Tinytl: Reduce memory, not parameters for efficient on-device learning. *Advances in Neural Information Processing Systems* 33 (2020), 11285–11297.
[25] Han Cai, Ligeng Zhu, and Song Han. 2018. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations*.
[26] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. 2020. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 5 (2020), 871–875.
[27] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving data locality. *ACM SIGPLAN Notices* 29, 11 (1994), 252–262.
[28] Changhao Chen, Xiaoxuan Lu, Andrew Markham, and Niki Trigoni. 2018. Ionet: Learning to cure the curse of drift in inertial odometry. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
[29] Changhao Chen, Peijun Zhao, Chris Xiaoxuan Lu, Wei Wang, Andrew Markham, and Niki Trigoni. 2020. Deep-learning-based pedestrian inertial navigation: Methods, data set, and on-device inference. *IEEE Internet of Things Journal* 7, 5 (2020), 4431–4441.
[30] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[31] Mahesh Chowdhary and Sankalp Dayal. 2018. Reconfigurable sensor unit for electronic device. US Patent 10,142,789.
[32] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual wake words dataset. *arXiv preprint arXiv:1906.05721* (2019).
[33] Nuri Cingillioglu and Alessandra Russo. 2022. pix2rule: End-to-end Neuro-symbolic Rule Learning. *15th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)* (2022).
[34] Pierre Comon. 1994. Independent component analysis, a new concept? *Signal Proceedings* 36, 3 (1994), 287–314.
[35] JM Corchado and J Aiken. 1998. Neuro-symbolic reasoning for real time oceanographic problems. In *Conference On Data Mining. IEE, Savoy Place, London*.
[36] Juan M Corchado, M Lourdes Borrajo, María A Pellicer, and J Carlos Yáñez. 2004. Neuro-symbolic system for business internal control. In *Industrial conference on data mining*. Springer, 1–10.
[37] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. 2020. Lagrangian Neural Networks. In *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*.
[38] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Edoardo Debenedetti, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. 2021. RobustBench: a standardized adversarial robustness benchmark. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.
[39] Francesco Croce and Matthias Hein. 2020. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International conference on machine learning*. PMLR, 2206–2216.
[40] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*. 221–236.
[41] Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. 2022. Multi-Objective Bayesian Optimization over High-Dimensional Search Spaces. In *The 38th Conference on Uncertainty in Artificial Intelligence*.
[42] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–27.
[43] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. *Proceedings of Machine Learning and Systems* 3 (2021), 800–811.
[44] Markus Deittert, Arthur Richards, Chris A Toomer, and Anthony Pipe. 2009. Engineless unmanned aerial vehicle propulsion by dynamic soaring. *Journal of guidance, control, and dynamics* 32, 5 (2009), 1446–1457.
[45] Bradley Denby and Brandon Lucia. 2019. Orbital edge computing: Machine inference in space. *IEEE Computer Architecture Letters* 18, 1 (2019), 59–62.
[46] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*. 1269–1277.
[47] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. 2022. Camaroptera: A Long-Range Image Sensor with Local Inference for Remote Sensing Applications. *ACM Transactions on Embedded Computing Systems (TECS)* (2022).

Just Accepted

[48] Thomas Desautels, Andreas Krause, and Joel W Burdick. 2014. Parallelizing exploration-exploitation tradeoffs in Gaussian process bandit optimization. *The Journal of Machine Learning Research* 15, 1 (2014), 3873–3923.
[49] Ian Dewancker, Michael McCourt, Scott Clark, Patrick Hayes, Alexandra Johnson, and George Ke. 2016. A strategy for ranking optimization methods using multiple criteria. In *Workshop on Automatic Machine Learning*. PMLR, 11–20.
[50] Yayun Du, Swapnil Sayan Saha, Sandeep Singh Sandha, Arthur Lovekin, Jason Wu, S. Siddharth, Mahesh Chowdhary, Mohammad Khalid Jawed, and Mani Srivastava. 2023. Neural-Kalman GNSS/INS Navigation for Precision Agriculture. *International Conference on Robotics and Automation (ICRA)* (2023).
[51] Lachit Dutta and Swapna Bharali. 2021. Tinyml meets iot: A comprehensive survey. *Internet of Things* 16 (2021), 100461.
[52] Naser El-Sheimy, Haiying Hou, and Xiaoji Niu. 2007. Analysis and modeling of inertial sensors using Allan variance. *IEEE Transactions on instrumentation and measurement* 57, 1 (2007), 140–149.
[53] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution. In *International Conference on Learning Representations*.
[54] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.
[55] Mahdi Abolfazli Esfahani, Han Wang, Keyu Wu, and Shenghai Yuan. 2019. AbolDeepIO: A novel deep inertial odometry network for autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems* 21, 5 (2019), 1941–1950.
[56] Mateus Espadoto, Rafael M Martins, Andreas Kerren, Nina ST Hirata, and Alexandru C Telea. 2019. Toward a quantitative survey of dimension reduction techniques. *IEEE Transactions on visualization and Computer graphics* 27, 3 (2019), 2153–2173.
[57] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610, 7930 (2022), 47–53.
[58] F Fdez-Riverola and Juan M Corchado. 2003. Forecasting red tides using an hybrid neuro-symbolic system. *AI Communications* 16, 4 (2003), 221–233.
[59] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul N Whatmough. 2019. SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers. *Advances in Neural Information Processing Systems* 32 (2019).
[60] Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. 2022. UDC: Unified DNAS for Compressible TinyML Models. *Advances in Neural Information Processing Systems* 35 (2022).
[61] Maxime Ferrera, Vincent Creuze, Julien Moras, and Pauline Trouvé-Peloux. 2019. AQUALOC: An underwater dataset for visual–inertial–pressure localization. *The International Journal of Robotics Research* 38, 14 (2019), 1549–1559.
[62] Ruipeng Gao, Xuan Xiao, Shuli Zhu, Weiwei Xing, Chi Li, Lei Liu, Li Ma, and Hua Chai. 2021. Glow in the Dark: Smartphone Inertial Odometry for Vehicle Tracking in GPS Blocked Environments. *IEEE Internet of Things Journal* 8, 16 (2021), 12955–12967.
[63] A Garcez, M Gori, LC Lamb, L Serafini, M Spranger, and SN Tran. 2019. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *Journal of Applied Logics* 6, 4 (2019), 611–632.
[64] Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Luis C Lamb, Leo de Penning, BV Illuminoo, Hoifung Poon, and Coppe Gerson Zaverucha. 2022. Neural-symbolic learning and reasoning: A survey and interpretation. *Neuro-Symbolic Artificial Intelligence: The State of the Art* 342 (2022), 1.
[65] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. 2020. PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors. *Philosophical Transactions of the Royal Society A* 378, 2164 (2020), 20190155.
[66] Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. 2020. Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. *Neurocomputing* 380 (2020), 20–35.
[67] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–213.
[68] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-sized machine learning models to tiny IoT devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 79–95.
[69] Sachin Goyal, Aditi Raghunathan, Moksh Jain, Harsha Vardhan Simhadri, and Prateek Jain. 2020. DROCC: Deep robust one-class classification. In *International conference on machine learning*. PMLR, 3711–3721.
[70] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. 2019. Hamiltonian neural networks. *Advances in neural information processing systems* 32 (2019).
[71] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics* 37, 3 (2020), 362–386.
[72] Antonio Gulli and Sujit Pal. 2017. *Deep learning with Keras*. Packt Publishing Ltd.
[73] Richard M Gunner, Mark D Holton, Mike D Scantlebury, O Louis van Schalkwyk, Holly M English, Hannah J Williams, Phil Hopkins, Flavio Quintana, Agustina Gómez-Laich, Luca Börger, et al. 2021. Dead-reckoning animal movements in R: a reappraisal using Gundog. Tracks. *Animal Biotelemetry* 9, 1 (2021), 1–37.

Just Accepted

[74] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. Protonn: Compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*. PMLR, 1331–1340.
[75] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. 2008. *Feature extraction: foundations and applications*. Vol. 207. Springer.
[76] Song Han, Huizi Mao, and William J Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *International Conference on Learning Representations (ICLR)* (2016).
[77] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[78] Sachini Herath, Hang Yan, and Yasutaka Furukawa. 2020. Ronin: Robust neural inertial navigation in the wild: Benchmark, evaluations, & new methods. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3146–3152.
[79] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[80] Shawn Hymel, Colby Banbury, Daniel Situnayake, Alex Elium, Carl Ward, Mat Kelcey, Mathijs Baaijens, Mateusz Majchrzycki, Jenny Plunkett, David Tischler, et al. 2022. Edge Impulse: An MLOps Platform for Tiny Machine Learning. *arXiv preprint arXiv:2212.03332* (2022).
[81] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[82] Valeriu Manuel Ionescu and Florentina Magda Enescu. 2020. Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers. In *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*. IEEE, 234–237.
[83] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.
[84] Daniel Kahneman. 2011. *Thinking, fast and slow*. Macmillan.
[85] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. Physics-informed machine learning. *Nature Reviews Physics* 3, 6 (2021), 422–440.
[86] Henry Kautz. 2022. The third AI summer: AAAI Robert S. Engelmore Memorial Lecture. *AI Magazine* 43, 1 (2022), 93–104.
[87] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. 2014. A survey of feature selection and feature extraction techniques in machine learning. In *2014 science and information conference*. IEEE, 372–378.
[88] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. 2019. ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection. In *2019 IEEE WKSH on Applications of Signal Proceedings to Audio and Acoustics (WASPAA)*. IEEE, 313–317.
[89] Kavya Kopparapu, Eric Lin, John G Breslin, and Bharath Sudharsan. 2022. TinyFedTL: Federated Transfer Learning on Ubiquitous Tiny IoT Devices. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 79–81.
[90] Sandeep Koranne. 2011. Boost c++ libraries. *Handbook of open source tools* (2011), 127–143.
[91] A Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *Master's thesis, University of Tront* (2009).
[92] Maxat Kulmanov, Mohammed Asif Khan, and Robert Hoehndorf. 2018. DeepGO: predicting protein functions from sequence and interactions using a deep ontology-aware classifier. *Bioinformatics* 34, 4 (2018), 660–668.
[93] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient machine learning in 2 kb ram for the internet of things. In *International Conference on Machine Learning*. PMLR, 1935–1944.
[94] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. 2018. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. *Advances in neural information processing systems* 31 (2018).
[95] Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2018. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601* (2018).
[96] Guillaume Lample and François Charton. 2019. Deep Learning For Symbolic Mathematics. In *International Conference on Learning Representations*.
[97] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. 2016. Temporal convolutional networks: A unified approach to action segmentation. In *European Conference on Computer Vision*. Springer, 47–54.
[98] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[99] Daniel D Lee and H Sebastian Seung. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature* 401, 6755 (1999), 788–791.
[100] Seulki Lee and Shahriar Nirjon. 2020. Learning in the wild: When, how, and what to learn for on-device dataset adaptation. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*. 34–40.

Just Accepted

[101] Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. 2021. μNAS: Constrained Neural Architecture Search for Microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*. 70–79.
[102] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. 2021. Memory-efficient Patch-based Inference for Tiny Deep Learning. *Advances in Neural Information Processing Systems* 34 (2021), 2346–2358.
[103] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020), 11711–11722.
[104] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.
[105] Kaixin Ma, Jonathan Francis, Quanyang Lu, Eric Nyberg, and Alessandro Oltramari. 2019. Towards Generalizable Neuro-Symbolic Systems for Commonsense Question Answering. In *Proceedings of the First Workshop on Commonsense Inference in Natural Language Processing*. 22–32.
[106] Meiyi Ma, Ji Gao, Lu Feng, and John Stankovic. 2020. STLnet: Signal temporal logic enforced multivariate recurrent neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 14604–14614.
[107] Michele Magno, Andrea Ronco, and Lukas Schulthess. 2022. On-Sensors AI with Novel ST Sensors: Performance and Evaluation in a Real Application Scenario. *TinyML Summit 2022* (2022).
[108] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. 2018. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems* 31 (2018).
[109] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. 2018. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*.
[110] Akhil Mathur, Daniel J Beutel, Pedro Porto Buarque de Gusmao, Javier Fernandez-Marques, Taner Topal, Xinchi Qiu, Titouan Parcollet, Yan Gao, and Nicholas D Lane. 2021. On-device federated learning with flower. *On-Device Intelligence Workshop at MLSys* (2021).
[111] Mark Mazumder, Sharad Chitlangia, Colby Banbury, Yiping Kang, Juan Manuel Ciro, Keith Achorn, Daniel Galvez, Mark Sabini, Peter Mattson, David Kanter, et al. 2021. Multilingual Spoken Words Corpus. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.
[112] Hashan Roshantha Mendis, Chih-Kai Kang, and Pi-cheng Hsiu. 2021. Intermittent-Aware Neural Architecture Search. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–27.
[113] Anton Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. 2016. MOT16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831* (2016).
[114] Ludovico Mitchener, David Tuckey, Matthew Crosby, and Alessandra Russo. 2022. Detect, Understand, Act: A Neuro-symbolic Hierarchical Reinforcement Learning Framework. *Machine Learning* 111, 4 (2022), 1523–1549.
[115] Jesse Mu and Jacob Andreas. 2020. Compositional explanations of neurons. *Advances in Neural Information Processing Systems* 33 (2020), 17153–17163.
[116] Allen Newell. 1980. Physical symbol systems. *Cognitive science* 4, 2 (1980), 135–183.
[117] Ren Pang, Zhaohan Xi, Shouling Ji, Xiapu Luo, and Ting Wang. 2022. On the Security Risks of {AutoML}. In *31st USENIX Security Symposium (USENIX Security 22)*. 3953–3970.
[118] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *International Conference on Learning Representations*.
[119] Terence Parr. 2013. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.
[120] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[121] Judea Pearl. 2019. The seven tools of causal inference, with reflections on machine learning. *Commun. ACM* 62, 3 (2019), 54–60.
[122] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[123] Riccardo Perego, Antonio Candelieri, Francesco Archetti, and Danilo Pau. 2020. Tuning deep neural network's hyperparameters constrained to deployability on tiny systems. In *International Conference on Artificial Neural Networks*. Springer, 92–103.
[124] Riccardo Perego, Antonio Candelieri, Francesco Archetti, and Danilo Pau. 2022. AutoTinyML for microcontrollers: Dealing with black-box deployability. *Expert Systems with Applications* 207 (2022), 117876.
[125] Honghui Qi and John B Moore. 2002. Direct Kalman filtering approach for GPS/INS integration. *IEEE Trans. Aerospace Electron. Systems* 38, 2 (2002), 687–693.
[126] Partha Pratim Ray. 2021. A review on TinyML: State-of-the-art and prospects. *Journal of King Saud University-Computer and Information Sciences* (2021).
[127] Wamiq Raza, Anas Osman, Francesco Ferrini, and Francesco De Natale. 2021. Energy-Efficient Inference on the Edge Exploiting TinyML Capabilities for UAVs. *Drones* 5, 4 (2021), 127.

Just Accepted

[128] Haoyu Ren, Darko Anicic, and Thomas A Runkler. 2021. The synergy of complex event processing and tiny machine learning in industrial IoT. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 126–135.
[129] Haoyu Ren, Darko Anicic, and Thomas A Runkler. 2021. Tinyol: Tinyml with online-learning on microcontrollers. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
[130] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–34.
[131] Yuji Roh, Geon Heo, and Steven Euijong Whang. 2019. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering* 33, 4 (2019), 1328–1347.
[132] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation*. Technical Report. California Univ San Diego La Jolla Inst for Cognitive Science.
[133] Oindrila Saha, Aditya Kusupati, Harsha Vardhan Simhadri, Manik Varma, and Prateek Jain. 2020. RNNPool: Efficient non-linear pooling for RAM constrained inference. *Advances in Neural Information Processing Systems* 33 (2020), 20473–20484.
[134] Swapnil Sayan Saha, Sandeep Singh Sandha, Luis Antonio Garcia, and Mani Srivastava. 2022. Tinyodom: Hardware-aware efficient neural inertial navigation. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 2 (2022), 1–32.
[135] Swapnil Sayan Saha, Sandeep Singh Sandha, Siyou Pei, Vivek Jain, Ziqi Wang, Yuchen Li, Ankur Sarker, and Mani Srivastava. 2022. Auritus: An Open-Source Optimization Toolkit for Training and Development of Human Movement Models and Filters Using Earables. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 2 (2022), 1–34.
[136] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. 2022. Machine Learning for Microcontroller-Class Hardware - A Review. *IEEE Sensors Journal* (2022).
[137] Sandeep Singh Sandha. 2021. Parameter search spaces use to evaluate Mango on classifiers. https://github.com/ARM-software/mango/blob/master/benchmarking/Parameter_Spaces_Evaluated.ipynb.
[138] Sandeep Singh Sandha, Mohit Aggarwal, Igor Fedorov, and Mani Srivastava. 2020. Mango: A python library for parallel hyperparameter tuning. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 3987–3991.
[139] Sandeep Singh Sandha, Mohit Aggarwal, Swapnil Sayan Saha, and Mani Srivastava. 2021. Enabling Hyperparameter Tuning of Machine Learning Classifiers in Production. In *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*. IEEE, 262–271.
[140] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.
[141] Maarten Sap, Ronan Le Bras, Emily Allaway, Chandra Bhagavatula, Nicholas Lourie, Hannah Rashkin, Brendan Roof, Noah A Smith, and Yejin Choi. 2019. Atomic: An atlas of machine commonsense for if-then reasoning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 3027–3035.
[142] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart, and Pascal Hitzler. 2021. Neuro-symbolic artificial intelligence. *AI Communications* (2021), 1–13.
[143] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.
[144] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. 1997. Kernel principal component analysis. In *International Conference on Artificial Neural Networks*. Springer, 583–588.
[145] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. 2020. Green ai. *Commun. ACM* 63, 12 (2020), 54–63.
[146] Luciano Serafini and Artur S d'Avila Garcez. 2016. Learning and reasoning with logic tensor networks. In *Conference of the Italian Association for Artificial Intelligence*. Springer, 334–348.
[147] Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. 2022. Toward verified artificial intelligence. *Commun. ACM* 65, 7 (2022), 46–55.
[148] Muhammad Shafique, Theocharis Theocharides, Vijay Janapa Reddy, and Boris Murmann. 2021. TinyML: Current Progress, Research Challenges, and Future Roadmap. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1303–1306.
[149] Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning differentiable programs with admissible neural heuristics. *Advances in neural information processing systems* 33 (2020), 4940–4952.
[150] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2017. On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374* (2017).
[151] Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, Michael I Jordan, and Shankar S Sastry. 2004. Kalman filtering with intermittent observations. *IEEE transactions on Automatic Control* 49, 9 (2004), 1453–1464.
[152] Aishwarya Sivaraman, Golnoosh Farnadi, Todd Millstein, and Guy Van den Broeck. 2020. Counterexample-guided learning of monotonic neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 11936–11948.
[153] Paul Smolensky. 1987. Connectionist AI, symbolic AI, and the brain. *Artificial Intelligence Review* 1, 2 (1987), 95–109.
[154] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012), 2951–2959.
[155] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. 2010. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *Proceedings of the 27th International Conference on Machine Learning*. 1015–1022.

Under Accepted

[156] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias W Seeger. 2012. Information-theoretic regret bounds for gaussian process optimization in the bandit setting. *IEEE transactions on information theory* 58, 5 (2012), 3250–3265.
[157] Jiankai Sun, Hao Sun, Tian Han, and Bolei Zhou. 2021. Neuro-Symbolic Program Search for Autonomous Driving Decision Module Design. In *Conference on Robot Learning*. PMLR, 21–30.
[158] Urmish Thakker, Igor Fedorov, Chu Zhou, Dibakar Gope, Matthew Mattina, Ganesh Dasika, and Jesse Beu. 2021. Compressing RNNs to Kilobyte Budget for IoT Devices Using Kronecker Products. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 17, 4 (2021), 1–18.
[159] Markus Thill, Wolfgang Konen, and Thomas Bäck. 2020. Time series encodings with temporal convolutional networks. In *International Conference on Bioinspired Methods and Their Applications*. Springer, 161–173.
[160] Megan Tjandrasuwita, Jennifer J Sun, Ann Kennedy, and Yisong Yue. 2021. Interpreting Expert Annotation Differences in Animal Behavior. In *CVPR 2021 Workshop on CV4Animation*.
[161] Nicholas H Tollervey. 2017. *Programming with MicroPython: embedded programming with microcontrollers and Python.* " O'Reilly Media, Inc.".
[162] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. WaveNet: A Generative Model for Raw Audio. In *9th ISCA WKSH on Speech Synthesis WKSH (SSW 9)*.
[163] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, 11 (2008).
[164] Laurens Van Der Maaten, Eric Postma, Jaap Van den Herik, et al. 2009. Dimensionality reduction: a comparative. *J Mach Learn Res* 10, 66-71 (2009), 13.
[165] Dinesh C Verma, Archit Verma, and Utpal Mangla. 2021. Addressing the Limitations of AI/ML in creating Cognitive Solutions. In *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*. IEEE, 189–196.
[166] Marc Roig Vilamala, Tianwei Xing, Harrison Taylor, Luis Garcia, Mani Srivastava, Lance Kaplan, Alun Preece, Angelika Kimmig, and Federico Cerutti. 2021. Using DeepProbLog to perform Complex Event Processing on an Audio Stream. In *Tenth International Workshop on Statistical Relational AI*.
[167] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. 2019. Legendre Memory Units: Continuous-Time Representation in Recurrent Neural Networks. *Advances in Neural Information Processing Systems* 32 (2019), 15570–15579.
[168] Yan Wang, Shuang Cang, and Hongnian Yu. 2019. A survey on wearable sensor modality centred human activity recognition in health care. *Expert Systems with Applications* 137 (2019), 167–190.
[169] Ziqi Wang, Ankur Sarker, Jason Wu, Derek Hua, Gaofeng Dong, Akash Deep Singh, and Mani B Srivastava. 2022. Capricorn: Towards Real-time Rich Scene Analysis Using RF-Vision Sensor Fusion. In *Proceedings of the 20th Conference on Embedded Networked Sensor Systems*.
[170] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).
[171] Pete Warden and Daniel Situnayake. 2019. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media.
[172] Justin D Weisz, Michael Muller, Steven I Ross, Fernando Martinez, Stephanie Houde, Mayank Agarwal, Kartik Talamadupula, and John T Richards. 2022. Better together? an evaluation of ai-supported code translation. In *27th International Conference on Intelligent User Interfaces*. 369–391.
[173] Alexander Wong, Mahmoud Famouri, and Mohammad Javad Shafiee. 2020. AttendNets: Tiny Deep Image Recognition Neural Networks for the Edge via Visual Attention Condensers. *6th WKSH on Energy Efficient Machine Learning and Cognitive Computer (EMC2 2020)* (2020).
[174] Tianwei Xing, Luis Garcia, Marc Roig Vilamala, Federico Cerutti, Lance Kaplan, Alun Preece, and Mani Srivastava. 2020. Neuroplex: learning to detect complex events in sensor networks through knowledge injection. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 489–502.
[175] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, et al. 2019. Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks. In *The World Wide Web Conference*. 2192–2202.
[176] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. 2018. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Advances in neural information processing systems* 31 (2018).
[177] Jiecao Yu, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2019. Tf-net: Deploying sub-byte deep neural networks on microcontrollers. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–21.
[178] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. 2022. Bytetrack: Multi-object tracking by associating every detection box. In *European Conference on Computer Vision*. Springer, 1–21.
[179] Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl. 2020. Tracking objects as points. In *European Conference on Computer Vision*. Springer, 474–490.
[180] Barret Zoph and Quoc V Le. 2017. Neural architecture search with reinforcement learning. *International Conference on Learning Representations (ICLR)* (2017).

Just Accepted