UNIVERSITY OF CALIFORNIA

Los Angeles

Physics-Aware Tiny Machine Learning

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical and Computer Engineering

by

Swapnil Sayan Saha

2023

ABSTRACT OF THE DISSERTATION

Physics-Aware Tiny Machine Learning

by

Swapnil Sayan Saha
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2023
Professor Mani B. Srivastava, Chair

Tiny machine learning has enabled Internet of Things platforms to make intelligent inferences for time-critical and remote applications from unstructured data. However, realizing edge artificial intelligence systems that can perform long-term high-level reasoning and obey the underlying system physics, rules, and constraints within the tight platform resource budget is challenging. This dissertation explores how rich, robust, and intelligent inferences can be made on extremely resource-constrained platforms in a platform-aware and automated fashion. *Firstly*, we introduce a robust training pipeline that handles sampling rate variability, missing data, and misaligned data timestamps through intelligent data augmentation techniques during training time. We use a controlled jitter in window length and add artificial misalignments in data timestamps between sensors, along with masking representations of missing data. *Secondly*, we introduce TinyNS, a platform-aware *neurosymbolic* architecture search framework for the automatic co-optimization and deployment of neural operators and physics-based process models. TinyNS exploits fast, gradient-free, and black-box Bayesian optimization to automatically construct the most performant learning-enabled, physics, and context-aware edge artificial intelligence program from a search space containing neural and symbolic operators within the platform resource constraints. To guarantee deployability, TinyNS receives hardware metrics directly from the target hardware during the optimization process. *Thirdly*, we introduce the concept of neurosymbolic tiny machine learning, where we showcase recipes for defining the physics-aware tiny machine learning program synthesis search space from five neurosymbolic program categories. Neurosymbolic artificial intelligence combines the context awareness and integrity of symbolic techniques with the robustness and performance of machine learning models. We develop parsers to automatically write microcontroller code for neurosymbolic programs and showcase several previously unseen TinyML applications. These include onboard physics-aware neural-inertial navigation, on-device human activity recognition, on-chip fall detection, neural-Kalman filtering, and co-optimization of neural and symbolic processes. *Finally*, we showcase techniques to personalize and adapt tiny machine learning systems to the target domain and application. We illustrate the use of transfer learning, resource-efficient unsupervised template creation and matching, and foundation models as pathways to realize generalizable, domain-aware, and data-efficient edge artificial intelligence systems.

The dissertation of Swapnil Sayan Saha is approved.

Yang Zhang

Mohammad Khalid Jawed

Puneet Gupta

Emre Ertin

Mani B. Srivastava, Committee Chair

University of California, Los Angeles

2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

"It's better to burn out than to fade away..."

My rousing, often poignant, doctoral journey was the key to sustaining the theatrics of a fading star dragging down the memory lane, scared to touch the sky. The doctoral path was a contrived path with a heart and respect.

I express my utmost gratitude to Prof. Mani B. Srivastava for advising me for the past 3.5 years and transforming me into who I am today professionally and academically. He might come off as stern, austere, and detailed-oriented, but you will be rewarded with unrivaled critical thinking, system design skills, storytelling, and salesmanship. I want to thank my doctoral committee members, Prof. Puneet Gupta, Prof. Mohammad Khalid Jawed, Prof. Yang Zhang, and Prof. Emre Ertin for providing their valuable and insightful feedback that helped me shape my research directions.

I am specifically grateful to Sandeep Singh Sandha (Abacus.AI) for mentoring me throughout my doctoral journey. Sandeep wholeheartedly and selflessly developed my soft skills, spent sleepless nights working on my experiments and papers, guided my job search, and gave me a new perspective about the professional world. I am also thankful to my core research and industry collaborators, namely Prof. Luis A. Garcia (USC), Yayun Du (NorthwesternU), Siyou Pei (UCLA), Vivek Jain (XCOM Labs), Ziqi Wang (UCLA), Mohit Aggarwal (BrightNight), Caden Davis (UCLA), Jason Wu (UCLA), Golara A. Azar (UCLA), Prof. Nader Sehatbakhsh (UCLA), Mahesh Chowdhary (STMicroelectronics), Denis Ciocca (STMicroelectronics, Humane), and S. Siddharth (STMicroelectronics) for being pilots and lighthouses during my journey.

I would like to thank the wonderful board members at UCLA GSA, UCLA EGSA, and UCLA ECEGAPS for giving me a chance to develop my leadership skills, keeping me sane, and allowing me to contribute to the UCLA community. I am grateful to Noor Nakhaei (GSA, EGSA), Chris Chien (GSA), Tanvi Shinkre (GSA), Nathan Chen (GSA), Emma

Landry (GSA), Jamie Leonard (EGSA), Vera Smirnova Koutnik (EGSA, BCG), Rajashree Bhattacharya (EGSA), Jean Marques (EGSA), Andrew Liu (EGSA, ECEGAPS, RiceU), Jeya Vikranth Jeyakumar (EGSA, Nvidia), Wojciech Romaszkan (ECEGAPS, Amazon), Steven Moran (ECEGAPS, Apple), Alexander Johnson (ECEGAPS), Akash Deep Singh (WHRA, Ownwell), and Muratkhan Abdirash (ASUCLA) for believing in my capabilities to host events for the entire graduate student body.

Music has always been a public canvas to paint my desires and thoughts on for the last 13 years. I am grateful to my fellow bandmates from UCLA, namely Julian de Gortari Briseno, Brian Wang, Jacob Lee, and Asher Cohen for working on my debut album ”Memory Lane”, which has been one of my life’s bucket-list items. I am indebted to the UCLA Music Club for reviving my musicianship. I am especially thankful to Anindita Mashsharat (DU, NSI) for being my music mentor for the last 8 years.

Whenever I got overwhelmed by work and campus politics, I went to eat out, go-kart, go on long drives, hike, and play recreational sports. I am thankful to Debarnab Mitra (Qualcomm), Chinmay Talegaonkar (UCSD), Pradyumna Chari (UCLA), Vivian Dao (UCLA), Ankur Sarker (Apple), and everyone else who introduced me to some of the most beautiful and exciting spots in the USA.

As someone who rarely opens up to the external world, I have been fortunate to have had a select few who actually care for my inner well-being rather than my materialistic accolades. I am grateful to my long-time friend Sameia Zaman (MIT), who has kept my mental health in check, listened to my rants and blues, and made me believe in myself. I am grateful to Sayem Lincoln (UConn), Mozammel Hossain Rifat (GMG), Shafayat Osman (NSU), Bias Binte Kamal (UCSD), and Rian Binte Kamal (UIdaho) for valuing my heart.

Last but not least, I would thank my past collaborators, mentors, peers, and professors from DU and high school, without whom I would not be at UCLA. I would like to mention Prof. Atiqur Rahman Ahad (UEastLondon), Miftahul Jannat Rasna (PrincetonU), Prof. Subrata Kumar Aditya (DU, SHU), Prof. Mosabber Uddin Ahmed (DU), Prof. Z.

M. Parvez Sazzad (DU, BSMRU), Prof. Mainul Hossain (DU), Prof. Md. Shafiul Alam (DU), Shaheen Khan (MLIS), Abdur Raquib (MLIS), Shafizur Rahman Seeam (RIT), Prof. Bahauddin Omar (UColorado), Walid Bin Habib (FreiburgU), Farhan Bin Tarik (ClarkSC), Syed Ahmed Al Muyeed (Google), Anindya Das Antar (UMich), Suyog Vyawahare (Quectel), Md. Sadman Siraj (UNM), Taposh Ghosh (YorkU), Prodipta Guha (UniMelb), Amor Debnath (NewsCred), Md. Istiak Uddin (DU), Sujay Saha (DU), Borun Saha (DU), Ariful Islam Tusher (DU), Rakib Hasan Bhasha (USU), and Sarmistha Sarna Gomasta (UMassA) for being role models and encouraging me to pursue what I am good at.

The research reported in this dissertation 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 King Abdullah University of Science and Technology (KAUST) through its Sensor Innovation research program; the NIH mHealth Center for Discovery, Optimization and Translation of Temporally-Precise Interventions (mDOT) under award 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, KAUST, 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.

VITA

2015-2019	B.Sc., Electrical and Electronic Engineering, *University of Dhaka (DU).* Vice Chair, *IEEE Student Branch DU*; Research Assistant, *FAB Lab DU*; Embedded Intern, *Suyog Technologies*; Instructor, *Maple Leaf Intl' School.*
2019-2021	M.S., Electrical and Computer Engineering, *UCLA*.
2021-2023	Sensor Software Solutions Intern, *STMicroelectronics*; Director of Graduate Events, *UCLA GSA*; Vice President, *UCLA EGSA* and *ECEGAPS.*

PUBLICATIONS

Saha, S. S., Davis, C., Sandha, S. S., Park, J., Geronimo, J., Garcia, L. A., and Srivastava, M. (2023), "LocoMote: AI-driven Sensor Tags for Fine-Grained Undersea Localization and Sensing." in *IEEE Internet of Things Journal.* (under review)

Saha, S. S., Sandha, S. S., Aggarwal, M., Wang, B., Han, L., de Gortari Briseno, J. and Srivastava, M. (2023) "TinyNS: Platform-Aware Neurosymbolic Auto Tiny Machine Learning." in *ACM Transactions on Embedded Computing Systems.* (under review after revision)

Chowdhary, M., and Saha, S. S., (2023) "On-Sensor Online Learning and Classification Under 8 KB Memory." in *2023 26th International Conference on Information Fusion*, IEEE.

Saha, S. S., Du, Y., Sandha, S. S., Garcia, L. A., Jawed, M. K., and Srivastava, M. (2023) "Inertial Navigation on Extremely Resource-Constrained Platforms: Methods, Opportunities and Challenges." in *2023 IEEE/ION Position, Location and Navigation Symposium*, IEEE.

Du, Y., Saha, S. S., Sandha, S. S., Lovekin, A., Wu, J., Siddharth, S., Chowdhary, M., Jawed, M. K., and Srivastava, M. (2023), "Neural-Kalman GNSS/INS Navigation for Precision Agriculture." in *2023 International Conference on Robotics and Automation (ICRA)*, IEEE.

Kaidarova, A., Geraldi, N.R., Wilson, R.P., Kosel, J., Meekan, M. G., Eguíluz, V. M., Hussain, M. M., Shamim, A., Liao, H., Srivastava, M., Saha, S. S. *et al* (2023), "Wearables for an Internet of Marine Life." in *Nature Biotechnology.*

Saha, S. S., Sandha, S. S., and Srivastava, M. (2022). "Machine learning for microcontroller-class hardware-a review." *IEEE Sensors Journal*, vol. 22, no. 22, pp. 21362-21390.

Saha, S. S., Sandha, S. S., Garcia, L. A., and Srivastava, M. (2022). "Tinyodom: Hardware-aware efficient neural inertial navigation." *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(2), 1-32.

Saha, S. S., Sandha, S. S., Pei, S., Jain, V., Wang, Z., Li, Y., Sarker, A. and Srivastava, M. (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), 1-34.

Sandha, S. S., Aggarwal, M., Saha, S. S., and Srivastava, M. (2021). "Enabling hyperparameter tuning of machine learning classifiers in production." in *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)* (pp. 262-271). IEEE.

Saha, S. S., Sandha, S.S., Srivastava, M. (2020). "Deep Convolutional Bidirectional LSTM for Complex Activity Recognition with Missing Data." in *Human Activity Recognition Challenge*, Smart Innovation, Systems and Technologies, vol 199. Springer, Singapore.

CHAPTER 1

Introduction

Over the past decade, deep learning (DL) has been extensively used to make robust 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 [LBH15]. Deep neural networks (DNN) have been shown to provide rich and complex inferences over the first-principle approaches for sensor data analytics [SSS22a]. Thereby, it is desirable to port DNN pipelines onto low-end Internet-of-Things (IoT) nodes adopted in resource-limited applications, turning them from simple data harvesters to learning-enabled inference generators [SSA23]. Applications such as fitness activity detection, sleep monitoring, underwater biologging, anomaly detection, batteryless imaging, and keyword spotting require on-board intelligence, made possible through tiny machine learning (TinyML) [SSS22a]

TinyML refers to hardware and software suites that enable always-on, ultra-low-power ($\leq$ 1 mW), and on-device sensor data analytics on low-end ($\leq$ 1-2 MB of SRAM and eFlash) IoT platforms [DB21, RKA22, STR21, Ray21, SSS22a, SSA23]. TinyML holds the key to making on-board intelligent inferences from unstructured data for time-critical and remote applications, such as aerial robotics [ROF21], underwater navigation [SSG22], industrial machinery debugging [BRT21], picosatellite machine inference [DL19], and wildlife monitoring [DNB22]. 2.5 billion TinyML platforms are expected to ship in 2030 [AI22].

Fig. 1.1 illustrates the typical workflow for porting ML models to commodity IoT platforms. First, in the *model development phase*, **data engineering** (1) performs acquisition, analytics, and storage of raw sensor streams [MCB21, SSS22a, RHW19, LKR15]. Next, op-

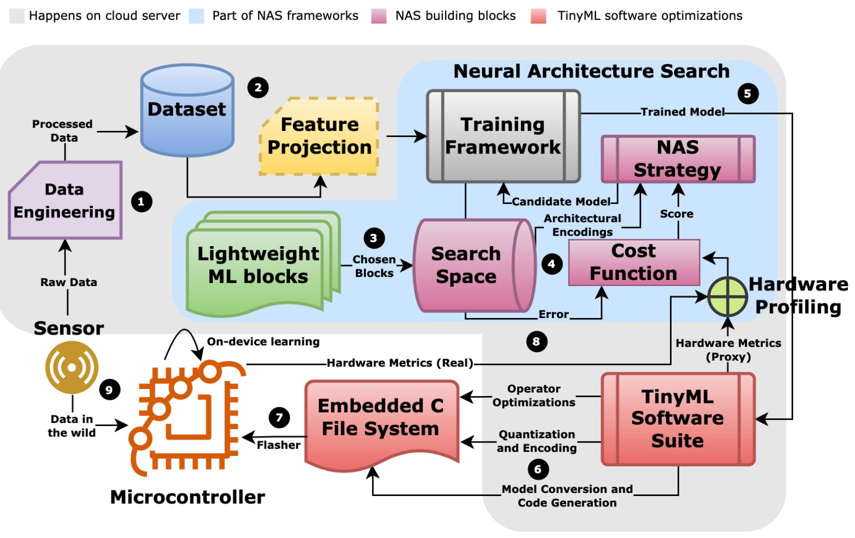


Figure 1.1: Closed-loop workflow of porting machine learning models onto microcontrollers. Step (3) to Step (8) are repeated until the desired performance is achieved [SSS22a].

tional **feature projection** (2) applies linear methods [DZB14, LS99, Com94, BG98], non-linear methods [RHW85, MH08, SSM97], or domain-specific feature extraction [GGN08] for dimensionality reduction while preserving data variance [EMK19]. Afterward, models are chosen from a **lightweight model zoo** (3) geared towards embedded deployment based on the application and hardware specifications [KGV17, GSG17, IHM16, HZC17, KSB18, VKE19, TFZ21, ODZ16, LVR16, WFS20]. The parameters of the backbone are optimized automatically using **neural architecture search** (NAS) given a *cost function* and the parameter *search space* based on the target device constraints (4) [BZF21, FMT22, SAS21, LDL21, LCL20]. In the *model deployment phase*, the trained candidate model is ported to a **TinyML compiler suite** (5) [DDJ21, LCL20, CMJ18, GRC20, LSC18, GLB19], which performs operator and inference engine optimizations [YLD19, LCL20, CMT94, DKA19], deep compression [HMD16], and code generation [SSS22a, WS19] (6). The generated **embedded C file system** is flashed onto the microcontroller via a command line interface (7). **On-device training** [RAR21b, LN20, CGZ20] or federated learning are used occasionally to account for shifts in incoming data distribution (9) [MBG21, KLB22].

1.1 Challenges of Deploying TinyML Sensing Systems

The first generation efforts in TinyML focused on the exploration (lightweight model blocks), optimization (NAS, AutoML), and integration (compiler suites) of standalone DNN within the device platform constraints [SSS22a, SSA23]. However, several issues plague the deployment of standalone ML models for robust, context-aware, and platform-aware sensor data analytics [SSS22a, SDS23b].

1.1.1 Addressing Spatial and Timing Uncertainty in Sensor Streams

Sensor data in the wild suffer from missing data, cross-channel timestamp misalignment, and window jitter [SSS20, SAN19, SNA20]. These uncertainties may stem from scheduling and timing stack delays, system clock imperfections, sensor malfunction, memory overflow, or power constraints [SBB15, HAR20]. Sensing uncertainty can reduce the performance of TinyML models, particularly for complex event processing [SSS20].

1.1.2 Injecting Platform-Awareness in AutoML

TinyML hardware platforms have tight memory, power, and compute budgets [LCL20]. 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 [SSS22a, BZF21]. However, directly porting ML models designed for high-end edge devices such as mobile phones or single-board computers are not suitable for microcontrollers. While AutoML and NAS frameworks have been proposed for optimizing lightweight NN backbones for TinyML platforms [SAS21, BZF21, LCL20, LDL21, JZS19, PCA20, PCA22, FMT22], existing AutoML tools lack platform-awareness, unable to guarantee error-free deployment of ML models [SSS22a, DSS23, SSG22, SSP22, SAS21]. These frameworks optimize a single model backbone at the architectural level and do not go down to the execution (compiler) level to include operator optimization, run-time dynamics, or inference optimizations [SSS22a, SSA23]. Thus, existing NAS frameworks are not suitable to perform platform-aware optimization of IoT programs that go beyond a standalone ML model [SSS22a, SSA23], such as a NN and a Kalman filter operating jointly [DSS23].

1.1.3 Obeying Underlying Physics, Context, and Rules

Real-world IoT applications must obey specific rules, physics, and heuristics for provably correct operation, context awareness, and explainability [SSS22a, MGF20, SZE21, XGV20]. For example, a UAV cannot exceed a certain bank angle without compromising stability [DRT09]. In complex event processing, specific granular action primitives (e.g., cooking a dish) must always precede other primitives (e.g., chopping vegetables) [RAA12]. Neural networks cannot assure that the learned distributions and decision trace obey all the rules, symmetries, and physics of the underlying system [MGF20, SSG22, SFM20, CGH20, KKL21]. Furthermore, the limited contextual field (few minutes) [RAR21a, XGV20, APS21, VXT21, MTC22] and lack of interpretability of ML models makes them unsuitable for high-level reasoning on atomic events [MGK18, MTC22, GGL19, YWG18, Pea19, SSS22b]. Given the ultra-resource constraints of TinyML platforms, manually crafting a context-aware, learning-enabled, and physics-aware sensing system is arduous and challenging [SSA23, RAR21a].

1.1.4 Personalizing to Target Domain and Application

Underparametrized models are less robust across domains and applications compared to vanilla models [SSP22]. Thus, TinyML models in the wild need to be fine-tuned periodically to ensure robustness across domain shifts in incoming data distribution [LN20]. The inference requirements may deviate from the training time requirements during deployment [CS23]. However, software-centric resource constraints, constrained learning theories, and static resource budgets prevent on-device learning from being a viable alternative to cloud-based training for microcontrollers [DGL21]. Moreover, collecting labeled data in the target domain for fine-tuning pre-trained models via transfer learning by non-experts is challenging [DSS23, SDS23b].

1.2 Research Objective

This dissertation explores "*physics, platform, domain, and uncertainty-aware auto tiny machine learning*" through the lens of inertial sensors.

1.3 Contributions and Dissertation Outline

The contributions of this dissertation directly tackle the four challenges identified in the previous section.

1.3.1 Uncertainty-Aware Robust Deep Learning

Chapter 2 introduces a robust training pipeline for handling sensing and timing uncertainties in DL frameworks during training time. The pipeline uses controlled jitter in window length and adds artificial misalignments in data timestamps between sensors, along with masking and window-aligned representations of missing data [SSS20].

1.3.2 Platform-Aware AutoML

Chapter 3 introduces TinyNS, a platform-in-the-loop framework for automatic optimization and deployment of *neurosymbolic* programs on commodity microcontrollers. A neurosymbolic program contains both ML and symbolic components, allowing the realization of physics and context-aware TinyML. Given a search space containing the parameters, 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 parameters within the target device memory, latency, and energy constraints. To guarantee program deployability, the framework communicates with the target hardware during the optimization process to receive hardware and program runtime metrics instead of relying on hardware proxies. The framework builds on top of a state-of-the-art, gradient-free, black-box Bayesian optimizer [SAF20, SAS21] designed to optimize non-gradient-friendly and expensive objective functions within a few epochs [SAS21, SSA23].

1.3.3 Neurosymbolic Tiny Machine Learning

Building on TinyNS, Chapter 4 introduces recipes and search space definition to map neurosymbolic program atoms from a prototyping language (e.g., Python) to a TinyML deployment language (e.g., C). Using case studies, TinyNS showcases recipes for defining the neurosymbolic program synthesis search space for five neurosymbolic program categories [SZE21]. The framework includes parsers that automatically write microcontroller code according to these recipes. Chapter 4 also showcases several unseen TinyML applications made possible by joint optimization of neural and symbolic components [SSA23]. These applications include:

- Hardware-aware neural inertial navigation [SSG22, SDS23b].
- Onboard human activity recognition and fall detection for earables [SSP22].
- Co-optimizing features and multiple model backbones for on-device human activity recognition [SSA23].
- Neural-Kalman filtering [DSS23].
- Co-optimizing neural object detector and symbolic object tracker [SSA23].

1.3.4 Fine-Tuning, Online Learning, and Foundation Models

Chapter 5 *firstly* showcases the utility of transfer learning for personalizing pre-trained models in the target domain with as little as 1 minute of labeled data [SSG22, SDS23b]. The chapter also introduces an automated pipeline to generate labeled inertial sensor data in the target domain for fine-tuning [DSS23, SDS23b]. *Secondly*, the chapter briefly outlines an ultra-lightweight, application-agnostic, and on-device learning and inference algorithm for on-sensor motion recognition [CS23]. *Lastly*, the chapter introduces ongoing work on foundation inertial-language models as a path towards realizing generalizable motion sensing applications.

Finally, Chapter 6 provides concluding remarks and future directions.

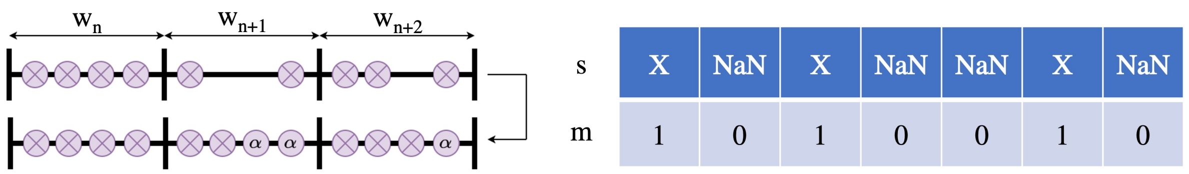
CHAPTER 2

Handling Spatial and Temporal Uncertainties

Sensing systems in the wild have to deal with data from multiple noisy sensors with variable sampling rates [SBB15], misaligned time stamps [SNA20], and missing data [HI19]. Traditional model training approaches are unable to deal with abnormal data on their own, requiring handcrafted features and domain knowledge [HI19][JLX18].

2.1 Contributions

We describe intelligent data augmentation and informative missingness injection techniques to make learning-enabled sensing systems robust to runtime sensing uncertainties during training time without feature engineering. Specifically, we use a controlled jitter in window length and add artificial misalignments in data timestamps between sensors, along with masking representations of missing data during training time. We evaluate our pipeline on the *Cooking Activity Dataset with Macro and Micro Activities* [ALT21], benchmarking the performance of a deep neural network complex activity detection classifier. In our evaluations, our framework achieves test accuracies of 88% and 72% respectively for macro and micro-activity classification, bringing in an 11% and 24% improvement in macro and micro-activity classification over uncertainty-unaware classifiers. Compared to competitors in the Cooking Activity Recognition Challenge, our framework provides 16% improvement over uncertainty-unaware machine-learning pipelines. The code is available at `https://github.com/nesl/Robust-Deep-Learning-Pipeline`.



s	X	NaN	X	NaN	NaN	X	NaN
m	1	0	1	0	0	1	0

Figure 2.1: Handling missing data in temporal streams. (Left) Window alignment with contained samples popped ahead. (Right) Mask metadata channel to characterize missing data location.

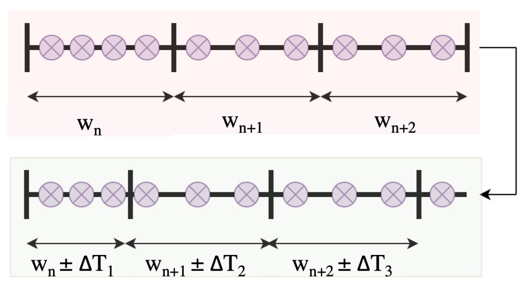
2.2 Handling Missing Data

Consider a multimodal sensor stream $\mathbf{s}$ with d channels (inputs) feeding data to a NN $f(\cdot)$ in chunks of fixed size n (called windows w). Adversarial elements such as communication outages, sensor malfunction, power outages, limited memory, timing errors, and sampling rate jitter can cause samples to be dropped from the channels, leading to missing data in the input stream [HI19][HGA18][HAR20]. It has been shown that missing data can hurt vanilla DL-based time-series processors, degrading medical data imputation accuracy by 18-65%, medical data classification by 2-5%, and complex activity recognition by 11-24% [CPC18][YZS18][SSS20]. To handle missing data, we use a combination of independent mask metadata channels to characterize missing samples and window alignment with contained samples popped ahead at the start of the windows during training [CPC18][SSS20]:

$$\mathbf{m}_t^d = \begin{cases} 1, \mathbf{s}_t^d \in \emptyset \\ 0, \text{otherwise} \end{cases} \tag{2.1}$$

$$\begin{cases} \mathbf{s}_{t:t+n}^d \to \mathbf{s}_{t:t+n-\gamma}^d \quad \forall |\mathbf{s}_{t:t+n}^d| = n-\gamma, \;\; \gamma > 0 \\ \mathbf{s}_{t+n-\gamma+1:t+n}^d = \alpha, \;\; \alpha \notin \mathbb{E}(\mathbf{s}^d) \end{cases} \tag{2.2}$$

The mask vector $\mathbf{m}_t^d$ consists of zeros at timestamps where data in channel d is missing and one at all other points. However, the masking approach may fail when the timestamps across the channels are misaligned or the actual window size is jittery. Thus, instead of aligning



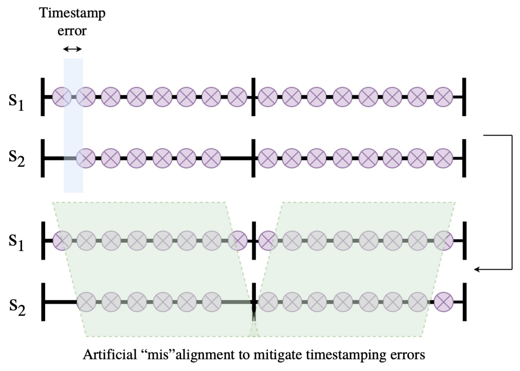


Figure 2.2: (Left) Handling sampling rate jitter. (Right) Time shift data augmentation.

the sample points, the individual overlapping windows are aligned based on the initial and terminal timestamps of each window, appending α for missing samples while choosing an average sampling rate for each window. The drawback of this approach is that α might be interpreted as part of the actual sensor stream. As a result, α must be chosen as a constant outside the expected value of input data. Both approaches are shown in Fig. 2.1.

2.3 Handling Window Jitter and Timestamp Misalignment

We apply intelligent data augmentation techniques when creating sensor windows during the training phase, shown in Fig. 2.2. Our augmentations exploit the observation that timing characteristics are variable and unstable for the devices capturing data during training and deployment settings. For example, the sampling rate of smartphone peripherals can vary wildly, with the inertial sensor sampling rate varying between 40 and 100 Hz for a 100 Hz accelerometer and the microphone sampling rate oscillating between 189 kHz and 195 kHz for 192 kHz/24-bit recording [SBB15][SAN19]. Several factors, including delay in the operating system time stamping and variable instantaneous I/O load, are responsible for this variation. For fixed sampling rates, the general approach is to use a fixed duration for window creation. Aware of the sampling rate instability, we introduce a controlled timing jitter δ in the window length when creating windows from the training files. We hypothesize that the introduced jitter in the window length, sampled from a uniform distribution $\mathbb{U}$, explicitly exposes $f(\cdot)$ to the variable sampling rate, allowing them to generalize on the test data [SAN19]:

$$|\mathbf{s}^d_{t:t+n}| \rightarrow |\mathbf{s}^d_{t:t+n}| \pm \delta, \;\; \delta \sim \mathbb{U}\left(0.5(a_1+b_1), \sqrt{\frac{(b_1-a_1)^2}{12}}\right) \tag{2.3}$$

In multimodal sensing, Sandha et al. [SNA20] have shown that data timestamps across sensors can have significant timing errors that can misalign the modalities. For example, the timing stack on Android smartphones can have as much as 5 seconds of error. A 1000 mS timestamp misalignment can drop multimodal fusion classification accuracy by 6% [SNA20]. Causes of misaligned modalities include poor management of the timing stack by the operating system and the choice of time synchronization techniques used by edge devices. This can result in overfitting of $f(\cdot)$ on the timing characteristics of data, with poor generalization in the deployment scenarios [SBB15]. To avoid this situation, we use the proposed time-shift data augmentation approach [SNA20] of adding artificial misalignments across the device's data timestamps when creating windows. This artificial misalignment helps $f(\cdot)$ maintain accuracy in the presence of variable timing characteristics in the wild:

$$\mathbf{s}^1_{t:t+n}, \mathbf{s}^2_{t:t+n} \rightarrow \mathbf{s}^1_{t\pm\epsilon:t\pm\epsilon+n}, \mathbf{s}^2_{t\mp\epsilon:t\mp\epsilon+n}, \;\; \epsilon \sim \mathbb{U}\left(0.5(a_2+b_2), \sqrt{\frac{(b_2-a_2)^2}{12}}\right) \tag{2.4}$$

2.4 Experimental Setup

In this section, we discuss the dataset used to evaluate our uncertainty-aware training pipeline (Section 2.4.1), as well as chosen models and baselines (Section 2.4.2 and Section 2.4.3).

2.4.1 Benchmark Dataset

The 2020 Cooking Activity Recognition Challenge [ALT21] embraces the hurdles of uncertainty and abnormality in sensor-based complex activity detection in the wild, intending to build a classifier to classify 3 distinct macro and 10 distinct micro-activities as follows:

- Making a sandwich - cut, wash, take, put, other
- Preparing fruit-salad - cut, take, peel, add, mix, put, other
- Preparing cereal - cut, take, pour, peel, put, open, other

The training dataset consists of 30-second windows from 3 subjects in 288 data frames/files. Sensors include 4 triaxial accelerometers placed at the left wrist, right wrist, right arm, and left hip and 29 triaxial motion capture (mo-cap) markers placed at random locations of the body, totaling 99 sensor channels. Each frame corresponds to a single macro-activity, with one-to-multiple possible micro-activities. Mo-cap samples are captured at approximately 100 Hz, while the accelerometers' sampling rates vary between 50 Hz - 100 Hz. The starting and terminal timestamps for each micro-activity are absent. The test dataset contains unlabeled data from a 4th subject.

For our experiments, we chose to omit the 3 left-wrist channels due to the high frequency of wrongly annotated timestamps and missing data in all files. Furthermore, mo-cap supplies absolute position rather than motion signatures unique to each action primitive. In our experiments, the motion capture data did not improve the validation accuracy. Thus, we trained the final models using 9 accelerometer channels (right wrist, right arm, and left hip). For classification, the files in the training set were split into 60:20:20 (train: validation: test) ratio randomly for both macro and micro-activity classification. The dataset was split into files (sessions/trials).

2.4.2 Model Implementation Specifics

We use an ensemble of 10 deep-convolutional bidirectional LSTM (DCBL) classifiers with majority voting decision fusion to classify macro-activities and conditionally select some of 20 micro-activity DCBL binary classifiers (one-vs-all, ensemble of 2) based on the macro-activity label. A sliding overlapping window of length 10 seconds (other candidates included 3 and 6 seconds) was used, with a stride of 1 second. No feature extraction or post-processing

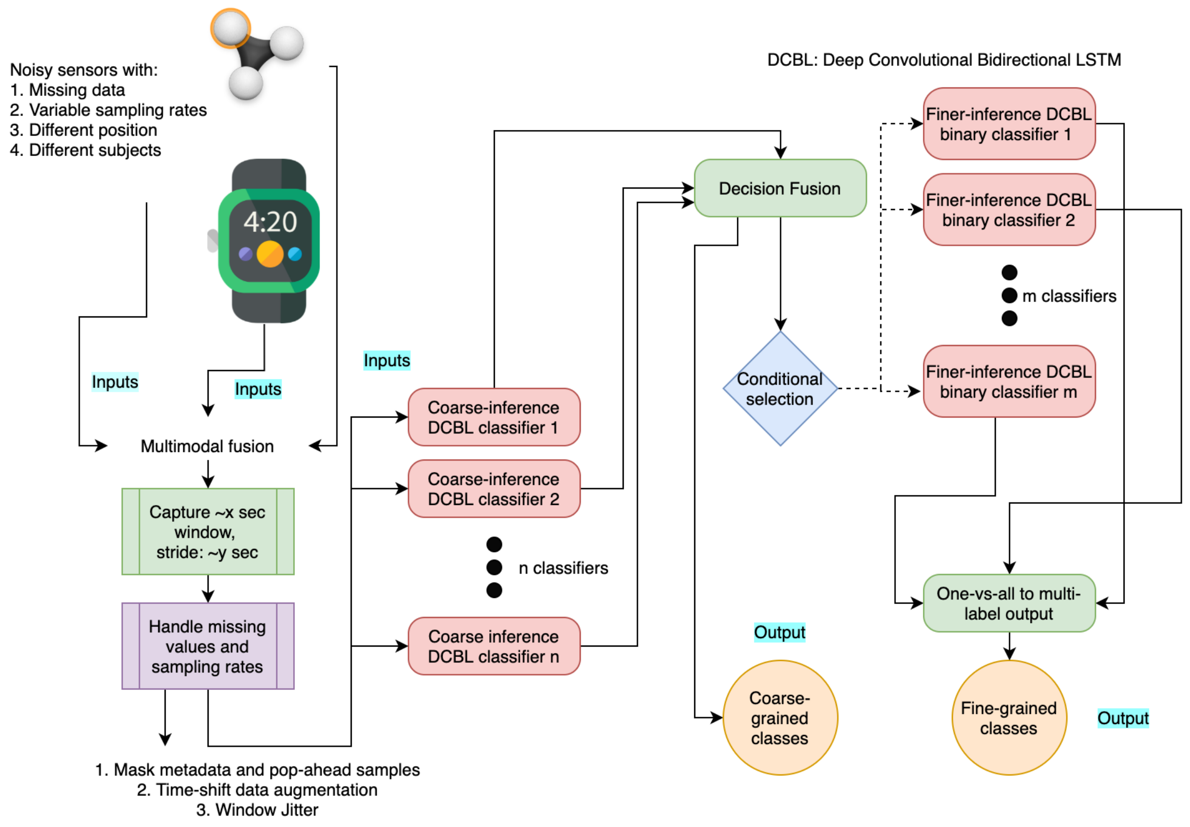

Noisy sensors with:
1. Missing data
2. Variable sampling rates
3. Different position
4. Different subjects
4:20
Inputs
Inputs
Inputs
Multimodal fusion
Capture ~x sec window, stride: ~y sec
Handle missing values and sampling rates
1. Mask metadata and pop-ahead samples
2. Time-shift data augmentation
3. Window Jitter
Coarse-inference DCBL classifier 1
Coarse-inference DCBL classifier 2
n classifiers
Coarse inference DCBL classifier n
DCBL: Deep Convolutional Bidirectional LSTM
Decision Fusion
Conditional selection
Output
Coarse-grained classes
Finer-inference DCBL binary classifier 1
Finer-inference DCBL binary classifier 2
m classifiers
Finer-inference DCBL binary classifier m
One-vs-all to multi-label output
Fine-grained classes
Output

Figure 2.3: Overview of the model architecture for uncertainty-aware complex event processing. The uncertainty injection is done during training time.

techniques were applied to the final models. Fig. 2.3 shows the architecture of the proposed model. The models were implemented in Jupyter notebook (Python), using Keras and Scikit-learn via a Tensorflow backend, with MATLAB being used for minor errands such as generating labels and splitting the training set. All models were trained on a GPU machine with 128 GB RAM, 2x 12 GB Nvidia GeForce GTX 1080 Ti, and 3.4GHz AMD Ryzen Threadripper 1950X 16-core CPU.

2.4.3 Baselines and Variations

For uncertainty-aware complex activity detection, we evaluate the following baselines:

- *Vanilla deep neural network (DNN)*: We tested the performance of the DCBL without any uncertainty injection.

- *DNN with normalization and interpolation*: Vanilla DNN with classical uncertainty pre-processing techniques such as bandpass filtering (using 20 Hz Butterworth low-pass filter), normalization (z-normalization, uni-variance, and min-max scaling), and vanilla interpolation (linear, spline, autoregression).

- *DNN with mo-cap data*: Multi-label classifiers trained on motion capture data instead of accelerometer signals.

- *DNN with time-shift augmentation and window jitter*: We add controlled artificial shifts and sampling rate jitter to the training data. To handle missing data, we follow the push ahead contained samples (no masking). We use the hierarchical DNN architecture discussed in Section 2.4.2.

- *DNN with time-shift augmentation, window jitter, and masking*: Same as previous, but with mask metadata channel included.

- *Top 10 competitors*: We also compare the performance of our approach with the top 10 (out of 78) competitors in the Cooking Activity Recognition Challenge. Models include convolutional neural network (CNN), CNN-LSTM, deep convolutional gated recurrent unit (DC-GRU), light gradient boosting machines (LightGBM), naive-Bayes (NB), Graph CNN, k-nearest neighbors (kNN), hidden Markov models (HMM), and multi-sampling classifiers [ALT21].

2.5 Evaluation

We provide results of locally run benchmarks of our pipeline (Section 2.5.1), as well as performance in the 2020 Cooking Activity Recognition Challenge in this section (Section 2.5.2).

Table 2.1: Summary of local benchmarks for our uncertainty-aware training pipeline.

Method	**Test Accuracy**
Vanilla DCBL	Macro: 0.78, Micro: 0.48
DCBL with normalization and interpolation	Macro: 0.30-0.77
DCBL with mo-cap data	Macro: 0.37-0.44
DCBL with time-shift augmentation and window jitter	Macro: 0.83, Micro: 0.72
DNN with time-shift augmentation, window jitter, and missing data handling	**Macro: 0.88, Micro: 0.72**

2.5.1 Local Benchmarks

Table 2.1 summarizes the results of our local benchmarks, containing all baselines except the competition baselines. From our evaluation, we see that the performance of vanilla classifiers improves significantly when we incorporate our proposed training pipeline in the channel. Classical approaches such as interpolation, filtering, normalization, and feature extraction (on mo-cap data) do not yield significant (in some cases lessen accuracy) performance improvement over vanilla classifiers. We hypothesize that classical approaches fail to converge due to the unnatural way the missing data was emulated in the dataset, consisting of long blocks of missing and abnormally aligned samples not normally encountered in the real world, coupled with the innate trade-off and time-evolving issues of classical methods. The uncommon nature of the dataset hurts micro-activity classification performance gains when we incorporate masking in the pipeline along with jitter and augmentation, yielding negligible performance improvement. This occurs because masking assumes perfect sample alignment among adjacent sensors. Thus, the gains obtained from ”space-aware” neural architectures are nullified by the natural misalignments of sensor samples in the dataset. However, our proposed pipeline component, namely augmentation, and jitter clusters the non-missing samples within a window at the beginning, ultimately yielding 24% performance improvement for complex activity recognition over vanilla approaches and thus exhibiting promising performance characteristics in worst-case multimodal sensing scenarios.

Table 2.2: Summary of the top teams in the 2020 Cooking Activity Recognition Challenge [ALT21].

Rank	**Method**	**Used Modalities**	**Competition Accuracy**	**Relabeling**	**Uncertainty-aware**
1	HMM	MoCap	92.08%	Yes	No
2	kNN	MoCap, Accelerometer	61.05%	Yes	No
3	**Ours (Robust DL)**	**Accelerometer**	**59.11%**	**No**	**Yes**
4	Graph CNN	MoCap, Accelerometer	56.63%	Yes	No
5	LightGBM, NB	MoCap, Accelerometer	55.00%	Yes	No
6	CNN-LSTM	Accelerometer	52.79%	No	No
7	Multi-sampling	MoCap, Accelerometer	43.39%	Yes	No
8	DC-GRU	Accelerometer	42.16%	No	No
9	CNN	Accelerometer	32.75%	No	No

2.5.2 Performance in the Cooking Activity Recognition Challenge

For the test dataset (without labels) provided for the competition, we used an ensemble of the 10 macro-activity classifiers (5 with masking and 5 without masking), while for macro-activity classification, we used 20 micro-activity binary classifiers, with 2 classifiers (masked and non-masked) for each of 10 micro-activities. Table 2.2 illustrates the performance of our model on the unlabelled test set versus other contestants. We ranked 3rd in the competition. Out of all the teams using DNN, our framework scored the highest and acknowledged the issues of missing data, sampling rate jitter, and timestamp misalignment in the dataset. The top two teams required manual relabeling of the training and test set using reference videos generated from the Mo-cap data. Our pipeline, on the other hand, is completely autonomous and agnostic of the test data characteristics. Among the best-performing teams using only the inertial sensor data, Team 6 performed $\sim$ 6% worse than our framework, while on average, our framework provides 16% improvement in classification accuracy over other teams using only the inertial sensor data.

2.6 Discussion

We introduced a robust deep-learning pipeline to handle data from multiple sensors in the presence of missing data, misaligned samples, and variable sampling rates. We evaluated the applicability of the time-shift data augmentation, controlled window jitter, and masks to handle the hurdles mentioned above, benchmarking our pipeline using state-of-the-art convolutional-LSTM architecture. Our evaluations yield 11% and 24% performance improvement for macro and micro-activity recognition over vanilla architectures. The proposed changes to the training pipeline can be incorporated into any DL framework to yield models robust to runtime uncertainties beyond simple activity detection applications.

There are several scopes of improvement to our work. While we have focused on making models robust to missing data during training, an alternative approach is to impute the missing samples synthetically using generative adversarial models (GAN) [YJS18][LJM18][ACS17], followed by training. Synthetic data generation is useful when analyzing microscopic granularities in temporal streams as well as data augmentation. In addition, the proposed training pipeline needs to be benchmarked on scenarios where multiple inertial sensors may be available intermittently, requiring spatially independent approaches [JLS19][XSB18].

CHAPTER 3

Platform-Aware Neurosymbolic Architecture Search

TinyML compiler suites enable the transfer of trained machine learning models generated by well-known libraries to microcontrollers [SSS22a]. These libraries provide comprehensive sets of optimized ML operators, algorithms, and tools, perform pruning, quantization (fixed and mixed precision), and model compression [HMD16], and convert models to deployable C code [DDJ21, LSC18, GRC20]. However, the suites assume that the trained model can fit within the device resource constraints. To satisfy the tighter hardware constraints of low-end IoT devices, neural architecture search (NAS) optimizes a given model backbone regularized by the target hardware specifications to strike a balance between accuracy and efficiency [FAM19, LCL20]. NAS is the automated process of finding the most optimal neural network within a neural network search space given target architecture and network architecture constraints, achieving a balance between accuracy, latency, and energy usage [RXC21][ZL17][BGN17]. While plentiful AutoML and NAS frameworks have been proposed for optimizing NN for TinyML platforms [SAS21, BZF21, LCL20, LDL21, JZS19, PCA20, PCA22, FMT22], these frameworks are not designed to perform *platform-aware joint optimization of neural and symbolic components* [SSS22a]. 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 [RAR21a], necessitating the need for AutoML platforms that can perform neurosymbolic optimization [SSA23] for deploying physics-aware TinyML programs.

3.1 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. 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. We use 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 a few iterations [SAF20, SAS21]. The optimizer forms the basis for our search algorithm. Our framework automatically synthesizes the most performant neurosymbolic program from a symbolic and ML operator search space within the target platform constraints. The Bayesian optimizer is available at: `https://github.com/ARM-software/mango`. TinyNS is available at: `https://github.com/nesl/neurosymbolic-tinyml`.

3.2 Mango: Fast, Parallel and Gradient-free Bayesian Optimizer

TinyNS adopts *Mango* [SAF20, SAS21], 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.

3.2.1 Surrogate Model

Typical surrogate models used in Bayesian optimization libraries are Gaussian processes, tree-structured Parzen estimators, and random forests. Among the available surrogate models, *Mango* uses the Gaussian process surrogate ($\mathcal{GP}$) over the search space ($\mathbf{\Omega}$) due to its ability to provide a tractable assessment of prediction uncertainty incorporating the effect of data scarcity [SLA12]. The Gaussian process is a non-parametric machine learning model specified using a mean (μ) and a kernel function (k).

$$\hat{f}(\mathbf{\Omega}) \sim \mathcal{GP}(\mu(\mathbf{\Omega}), k(\mathbf{\Omega}, \mathbf{\Omega}')) \tag{3.1}$$

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.* [GH20], 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 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.

3.2.2 Acquisition Function

The exploration-exploitation is handled using the upper confidence bound as the acquisition function. In upper confidence bound the next sample ($\mathbf{\Omega}_t$) at iteration t is sampled from the search space ($\mathbf{\Omega}$) 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.

$$\mathbf{\Omega}_t = \arg\max_{\mathbf{\Omega}}(\mu_{t-1}(\mathbf{\Omega}) + \beta^{0.5}\sigma_{t-1}(\mathbf{\Omega})) \tag{3.2}$$

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 [SKK10, SKK12, DKB14]. β is heuristically decided based on the complexity of the search space (domain size) $|\mathbf{\Omega}|$, the current iteration count t, and the variance (uncertainty) $\sigma^2_{t-1}(\mathbf{\Omega})$ at iteration $t-1$.

$$\beta = \alpha \cdot \exp(2 \cdot C), \quad \alpha = \sqrt{2\log(0.6 \cdot |\mathbf{\Omega}| \cdot t^2 \cdot \pi^2)}, \quad C = \frac{8}{\log(1 + \frac{1}{\delta + \sigma_{t-1}(\mathbf{\Omega})})}, \quad \delta = 1e^{-6} \tag{3.3}$$

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 $\sigma^2_{t-1}(\mathbf{\Omega})$, leading to a lower value of β. *Lastly*, if a region is invalid or bad, then $\mu_{t-1}(\mathbf{\Omega})$ 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 $\mu_{t-1}(\mathbf{\Omega})$ 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. $\sigma^2_{t-1}(\mathbf{\Omega})$ 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. $\mu_{t-1}(\mathbf{\Omega})$ 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).

3.2.3 Handling Mixed Search Spaces

Traditionally, gradient-driven optimizers (e.g., GpyOpt [aut16a] and Skopt [aut16b]) are used to find the next promising sample, such as in SpArSe [FAM19]. Sandha *et al.* [SAF20, SAS21] showed that gradient-driven optimization in complex search 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 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 [SAS21] of the optimizer compared to the gradient-based methods used in other Bayesian optimization libraries [aut16a, aut16b, FAM19].

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.* [GH20]. 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 [GH20] 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.

3.2.4 Parallelization

Another challenge in solving Eq. 3.2 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 [DKB14]. 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 hyperparameters space to enable exploration or diversity. The number of clusters is equal to the batch size and is flexible.

3.2.5 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 make Mango platform-aware by allowing it to talk to the target hardware during deployment time. This allows guaranteed program deployment and accurate profiling.

3.2.6 Evaluation: Parallel Search in Mango

We visualize the parallel search enabled by *Mango* in Fig. 3.1 (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 upper confidence bound. 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.

3.2.7 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.* [DMC16]. 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 [PVG11], i.e., a total of 9 tuning tasks (three classifiers trained using three datasets). The search space includes continuous, integer, and categorical hyperparameters with the exact definitions available [San21]. We tune each classifier for 80 iterations and repeat each tuning experiment 30 times. Results are shown in Fig. 3.1 (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. Overall, Mango offers state-of-the-art optimization capabilities handling complex search spaces.

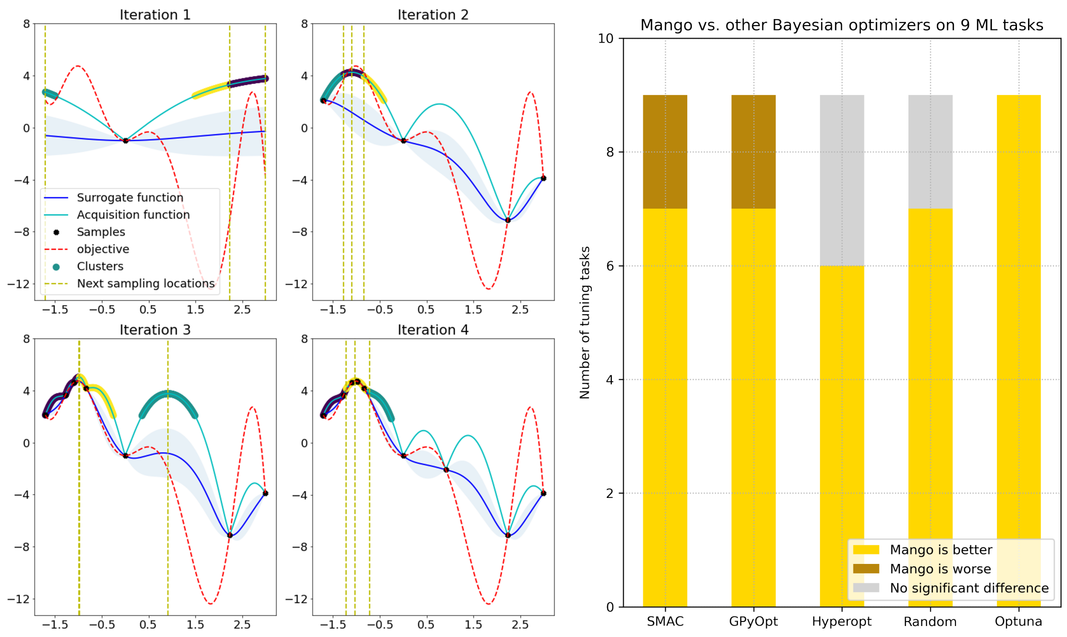


Figure 3.1: (Left) Visualizing parallel optimization in Mango. (Right) Sequential optimization performance of Mango on 9 ML classification tasks versus 5 other state-of-the-art Bayesian optimizers.

3.3 Platform-Aware Neurosymbolic Optimization

TinyNS treats neurosymbolic architecture search as nonlinear programming [Ber16] over the search space $\mathbf{\Omega}$:

$$\min \mathbf{f}(\mathbf{\Omega}), \quad \text{s.t. } \mathbf{f}(\mathbf{\Omega}) \leq \mathbf{b} \tag{3.4}$$

where

$$\mathbf{f}(\cdot) = \lambda_k \sum_n g_k(\mathbf{\Omega}), \;\; \mathbf{\Omega} = \{\{V, E\}, [\theta_m, m, w], [\theta_s, s, u]\}, \;\; \sum_n \lambda_k = 1, \;\; k \in [1, n] \tag{3.5}$$

$\mathbf{\Omega}$ 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 symbolic components include the fixed 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 $q^{\mathbf{\Omega}}(\mathbf{X})$ with edges E, vertices V and input tensor $\mathbf{X}$. The goal is to find a neurosymbolic program that satisfies the aggregate constraint $\mathbf{f}(\mathbf{\Omega}) \leq \mathbf{b}$. In other words, the objective function seeks a Pareto-frontier configuration Ω^* under competing objectives [FAM19] such that:

$$\mathbf{f}_k(\mathbf{\Omega}^*) <= \mathbf{f}_k(\mathbf{\Omega}) \;\; \forall k, \mathbf{\Omega} \;\; \wedge \exists j : \mathbf{f}_j(\mathbf{\Omega}^*) < \mathbf{f}_j(\mathbf{\Omega}) \;\; \forall \mathbf{\Omega} \neq \mathbf{\Omega}^* \tag{3.6}$$

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:

- $d \vee \neg d, \;\; \underbrace{d = (\exists g_k(\cdot) \wedge \exists c \in \mathbf{\Omega}) \Rightarrow \left(\nexists \lim_{x \to c} g_k(x) \vee \nexists g(c) \vee \lim_{x \to c} g_k(x) \neq g(c)\right)}_{\text{discontinuity condition}}$

- $$\exists z \in \mathbf{\Omega} \Rightarrow \Big[\underbrace{z \in \mathbb{R}}_{\substack{\text{continuous,}\\\text{numeric}}} \vee \underbrace{[z \in \mathbf{B}, \mathbf{B} \subseteq \mathbb{R}, f : \mathbf{B} \to \mathbb{N}]}_{\text{discrete, numeric}} \vee \underbrace{[((\forall q \in \bar{q})\pi q = q) \Rightarrow \pi \cdot z = z, \pi \in \text{Perm } \mathbb{A}]}_{\text{categorical, nominal}} \vee \underbrace{z \in \omega_\omega}_{\substack{\text{categorical,}\\\text{ordinal}}} \Big]$$
- $$\underbrace{\begin{aligned}\exists x|a \in \mathbf{X}, x \in \mathbf{\Omega}, a, b \in \mathcal{A} \Rightarrow \Big[(a = b \Rightarrow x|a = y|b) \wedge (x|b = y|b \Rightarrow x|a = y|a) \wedge \\ \big(\forall (a_i)_{i\in I} \in \mathcal{A}, \forall (x_i)_{i \in I} \in \boldsymbol{X}, \forall i \in I \Rightarrow \exists! x(x|a_i = x_i|a_i)\big)\Big]\end{aligned}}_{\text{conditional inclusion}}$$

The base formulation of Eq. 3.4 and Eq. 3.5 is given as:

$$\min f_{\text{opt}}, \quad f_{\text{opt}} = \lambda_1 f_{\text{error}}(\mathbf{\Omega}) + \lambda_2 f_{\text{flash}}(\mathbf{\Omega}) + \lambda_3 f_{\text{SRAM}}(\mathbf{\Omega}) + \lambda_4 f_{\text{latency}}(\mathbf{\Omega}) \tag{3.7}$$

where,

$$f_{\text{flash}}(\mathbf{\Omega}) = \begin{cases} \gamma_f \Leftrightarrow \Bigg(|\gamma_f| < 1 \wedge \underbrace{\epsilon_{\text{flag}} = 0}_{\text{fault flag}}\Bigg), \;\; \gamma_f = \Bigg(\underbrace{-\frac{||h_{\text{FB}}(w, \{V, E\})||_0}{\text{flash}_{\max}}}_{\text{model proxy}} + \underbrace{\xi_f}_{\substack{\text{slack for}\\\text{symbolic}}} \vee \underbrace{-\frac{\text{Compiler-reported flash}}{\text{flash}_{\max}}}_{\text{real measurement}}\Bigg) \\ \alpha_f, \;\; \alpha_f \gg \text{flash}_{\max} \end{cases} \tag{3.8}$$

$$f_{\text{SRAM}}(\mathbf{\Omega}) = \begin{cases} \gamma_s \Leftrightarrow \Bigg(|\gamma_s| < 1 \wedge \underbrace{\epsilon_{\text{flag}} = 0}_{\text{fault flag}}\Bigg), \;\; \gamma_s = \Bigg(\underbrace{-\frac{\max_{l \in [1,L]}\{||x_l||_0 + ||a_l||_0\}}{\text{SRAM}_{\max}}}_{\text{model proxy}} + \underbrace{\xi_s}_{\substack{\text{slack for}\\\text{symbolic}}} \vee \underbrace{-\frac{\text{Compiler-reported SRAM}}{\text{SRAM}_{\max}}}_{\text{real measurement}}\Bigg) \\ \alpha_s, \alpha_s \gg \text{SRAM}_{\max} \end{cases} \tag{3.9}$$

$$f_{\text{latency}}(\mathbf{\Omega}) = \begin{cases} \underbrace{\frac{\text{FLOPS}}{\text{FLOPS}_{\text{target}}}}_{\text{model proxy}} \vee \underbrace{\frac{\text{RTOS-reported latency}}{\text{latency}_{\text{target}}}}_{\text{real measurement}} \Leftrightarrow \underbrace{\epsilon_{\text{flag}} = 0}_{\text{fault flag}} \\ \alpha_l, \alpha_l \gg \text{FLOPS}_{\text{target}} \vee \text{latency}_{\text{target}} \end{cases} \tag{3.10}$$

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 $f_{\text{error}}(\mathbf{\Omega})$. When the target hardware is connected to the training server, the compiler provides the program SRAM consumption $f_{\text{SRAM}}(\mathbf{\Omega})$ and flash consumption$f_{\text{flash}}(\mathbf{\Omega})$, while the onboard real-time operating system (RTOS) reports the program runtime latency $f_{\text{latency}}(\mathbf{\Omega})$. The measurements are conditioned on the absence of faults, indicated by ϵ_{flag}. 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 [DKB14] of GP-UCB [SKK10, SKK12]. 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 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 [FAM19] 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. $f_{\text{flash}}(\mathbf{\Omega})$ is given by the size of the flatbuffer model schema $h_{\text{FB}}(\cdot)$ [DDJ21]. $f_{\text{SRAM}}(\boldsymbol{\Omega})$ is given by the standard NN SRAM usage model, with intermediate layer-wise activation maps and tensors stored in the SRAM [FAM19]. $f_{\text{latency}}(\boldsymbol{\Omega})$ is provided by the FLOPS count [BZF21]. 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) [SSG22, SSP22]. 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 [SSG22, SSP22].
- 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 in Chapter 4.

3.4 TinyNS Evaluation for Neural Architecture Search

We provide a qualitative comparison of TinyNS against state-of-the-art NAS frameworks designed for microcontrollers in Section 3.4.1. Afterward, we validate the viability of TinyNS for generating performant microcontroller-class models on the industry-standard MLPerf Tiny v0.5 Inference Benchmark [BRT21] in Section 3.4.2.

3.4.1 Neural Architecture Search for Microcontrollers

Table 3.1 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 [MKH21] 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 [SSS22a]. However, RL has a long convergence time (e.g., 5 GPU years) with additional fine-tuning costs [SSS22a, CGW19]. MCUNet [LCL20, LCC21] and μNAS [LDL21] 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 [CGW19]. 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 [SSS22a, CGW19]. MicroNets [BZF21] and UDC [FMT22] use differentiable NAS (DNAS), which performs continuous gradient descent relaxation of weights and architectural encodings jointly with approximate gradients via path binarization [LSY18, CZH18]. This reduces the convergence time to 1-3 GPU weeks [CGW19]. 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 [MKH21, SSS22a]. Bayesian optimization can handle discontinuous search spaces and cost functions while being executable on commodity GPU workstations [SSG22, SSP22], further reducing the convergence time to 1-10 GPU days [FAM19]. However, vanilla Bayesian optimization struggles in search spaces beyond a dozen hyperparameters and assumes dense distribution of performant models in the search space [FMT22, DEB22]. 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 [FAM19] to perform exploration and exploitation similar to UDC [FMT22]. This prevents TinyNS from being stuck to local optima or evaluating invalid configurations [SSG22, SAF20] 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 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.

3.4.2 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 [BRT21, SSS22a]. The tasks include image classification (CIFAR10 dataset [Kri09]), unsupervised anomaly detection (ToyADMOS dataset [KSU19]), keyword spotting (Google Speech Commands dataset [War18]), and visual wake words detection (Visual Wake Words dataset [CWS19]). We benchmark TinyNS on the first three tasks.

3.4.2.1 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

Table 3.1: Qualitative comparison of existing microcontroller NAS versus TinyNS

Method	**Search Strategy**	**Profiler**	**Search Space**	**Cost Function Hyperparameters**	**Inference Engine**	**Compression Awareness**	**Open Source**
SpArSe [FAM19]	Gradient-driven Bayesian	Analytical	Conv2D (regular, depthwise, downsampled)	Error, SRAM, Flash	uTensor	Pruning (structured, unstructured)	No
MCUNet [LCL20, LCC21]	Evolutionary (with weight sharing)	Lookup tables, prediction models	Conv2D (elastic)	Error, SRAM, Flash, Latency	TinyEngine [LCL20]	None	No
MicroNets [BZF21]	One-shot DNAS	Analytical	Conv2D (MbNetv2, DS-CNN)	Error, SRAM, Flash, Latency	TFLite Micro [DDJ21], CMix-NN [CRF20]	Quantization (sub-byte)	No
μNAS [LDL21]	Evolutionary (no weight sharing)	Analytical	Conv2D (regular, depthwise)	Error, SRAM, Flash, Latency	TFLite Micro [DDJ21]	Structured Pruning	Yes
iNAS [MKH21]^	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 [FMT22]	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 [DDJ21]	Quantization (1 byte)	Yes

^ intermittent-aware NAS

* sum of progress preservation, progress recovery, battery recharge, and compute cost

@ sum of progress preservation, progress recovery, and compute cost 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 $49\times10\times1$ tensor.

3.4.2.2 Model Backbones, Training Details, and Search Space Definition

For image recognition, we optimize the ResNet [HZR16] backbone provided in the benchmark suite. Following the settings in the MLPerf Tiny v0.5 Benchmark [BRT21] and state-of-the-art NAS frameworks for microcontrollers [LCL20, BZF21, PCA22, LDL21, EMH19b, FMT22], 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 [SDS20] 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 [HZR16], MobileNets [HZC17], or SqueezeNet [IHM16]) or compact (e.g., FastGRNN [KSB18], Bonsai [KGV17], ProtoNN [GSG17] or temporal CNN [LVR16]), allowing the use of known and fixed training epochs or a small number of training epochs to achieve acceptable performance [SSS22a]. 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]

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.* [TKB20]. The encoder is a TCN [ODZ16, LVR16] without dilated kernels, followed by a 1D convolutional layer (linear activation) with a quarter and one-third of the number of filters and 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)
- 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 parameter count [ODZ16, LVR16]. 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]

3.4.2.3 Overall Performance

Fig. 3.2 (Left) and Fig. 3.3 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 [FAM19] and μNAS [LDL21] by 4.5%-17.5% while taking 1.7×-7.7× lower convergence time (shown in Fig. 3.2 (Right)). Compared to LEMONADE [EMH19a], 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

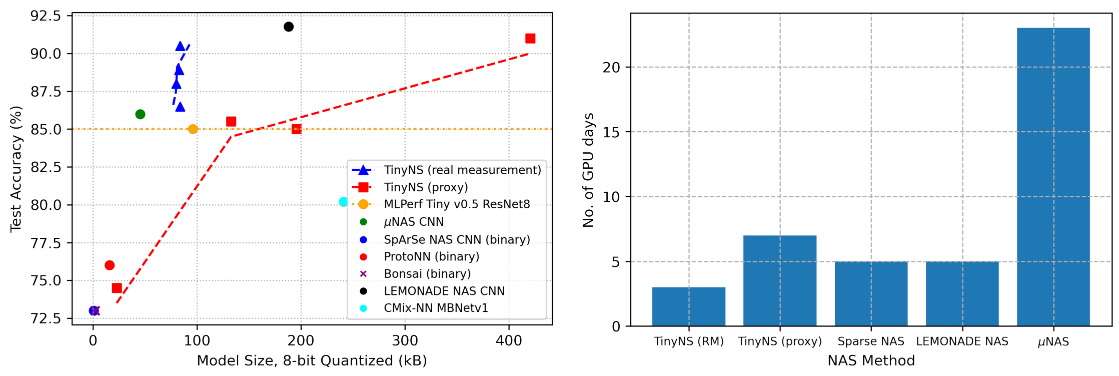


Figure 3.2: (Left) Test accuracy versus model size of CIFAR10 ResNet models found by TinyNS versus competing CIFAR10 models designed for microcontrollers. (Right) NAS convergence time for TinyNS and competing microcontroller NAS frameworks on the CIFAR10 dataset.

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 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 steps.

For anomaly detection, TinyNS outperforms attention-based OutlierNets [AFS21] by 6.3% and guarantees deployability over MobileNetv2 [SHZ18], but underperforms over MicroNets [BZF21] models. We hypothesize that flattening the log MFCC in the 1D-CNN backbone loses spatial correlation across the feature coefficients. This phenomenon also gen-

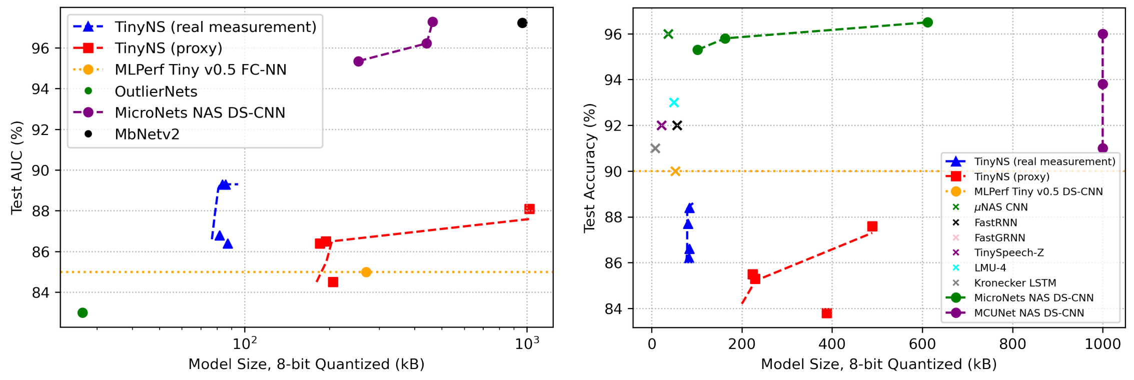


Figure 3.3: (Left) Test AUC versus the model size of anomaly detection models (1D-CNN) found by TinyNS versus competing anomaly detection models designed for microcontrollers on the ToyADMOS dataset. (Right) Test accuracy versus the model size of keyword spotting models (TCN) found by TinyNS versus competing keyword spotting models designed for microcontrollers on the Google Speech Commands dataset.

erates sub-optimal TinyNS models for keyword spotting, failing to cross the benchmark accuracy of 90% as shown in Fig. 3.3 (Right). This showcases the importance of performing NAS not just over a single model backbone, but over multiple model backbones. In Chapter 4, 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.

3.4.2.4 Architectural Adaptation Based on Resource Availability

Table 3.2, Table 3.3, and Table 3.4 show the hyperparameters of the model backbones for the three tasks generated by TinyNS for four different STM32 microcontrollers 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

Table 3.2: 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	Latency (s), FLOPS	Filters	Kernel size	Stacks	Batch Normalization	Activations
F446RE (128, 512)	Real	107 kB	0.58 (L)	10	5	4	True	True
	Proxy	95.8 kB	12.9M (F)	4	7	4	True	True
L476RG (128, 1024)	Real	87.8 kB	3.13 (L)	24	5	2	True	True
	Proxy	56.5 kB	3.82M (F)	6	3	3	True	True
F746ZG (320, 1024)	Real	308 kB	1.39 (L)	22	7	2	True	True
	Proxy	286 kB	55.9M (F)	24	3	3	True	True
L4R5ZI_P (640, 2048)	Real	608 kB	1.13 (L)	20	3	4	True	True
	Proxy	309 kB	40.9M (F)	18	3	4	False	True

Table 3.3: 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	Latency (s), FLOPS	Filters	Kernel size	Layers per stack	Stacks	Skip connection
F446RE (128, 512)	Real	87.8 kB	0.01 (L)	50	3	5	1	True
	Proxy	81.3 kB	0.32M (F)	16	10	4	1	True
L476RG (128, 1024)	Real	88.2 kB	0.06 (L)	38	10	6	1	True
	Proxy	62.0 kB	0.24M (F)	26	3	5	1	True
F746ZG (320, 1024)	Real	288 kB	0.01 (L)	42	4	4	3	True
	Proxy	78.1 kB	0.31M (F)	30	4	3	1	True
L4R5ZI_P (640, 2048)	Real	608 kB	0.03 (L)	63	3	5	1	True
	Proxy	444 kB	1.77M (F)	57	6	4	2	True

Table 3.4: 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	Latency (s), FLOPS	Filters	Kernel size	Dilations, layers per stack	Stacks	Skip conn.
F446RE (128, 512)	Real	106 kB	0.31 (L)	51	9	[1,8,64,128], 4	2	True
	Proxy	77.8 kB	21.6M (F)	27	9	[1,2,16,32,64,128], 6	2	True
L476RG (128, 1024)	Real	95.4 kB	0.65 (L)	44	7	[1,2,4,8,16,128], 6	2	True
	Proxy	79.4 kB	22.0M (F)	30	9	[1,2,8,16,128], 5	2	True
F746ZG (320, 1024)	Real	286 kB	0.04 (L)	45	4	[1,4,16,64,128], 5	1	True
	Proxy	147 kB	32.4M (F)	56	4	[1,4,8,64], 4	3	True
L4R5ZI_P (640, 2048)	Real	606 kB	1.66 (L)	63	8	[1,4,8,16,32,64,128,256], 8	3	True
	Proxy	210 kB	68.2M (F)	55	8	[1,16,128], 3	3	True

number of kernels, size of kernels, and the number of convolutional stacks with increasing device capabilities to maximize accuracy. Fig. 3.4 and Fig. 3.5 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 3.2, Table 3.3, and Table 3.4 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.

3.4.2.5 Convergence Time of Proxyless versus Proxied TinyNS

Fig. 3.6 shows the number of steps needed to reach the best optimization score for proxy less and proxied TinyNS for all three tasks. For both profiling techniques, tighter hardware constraints (lower SRAM and flash capacities) equate to more steps required for convergence. However, proxy less TinyNS converges 3.2×-12.6× 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

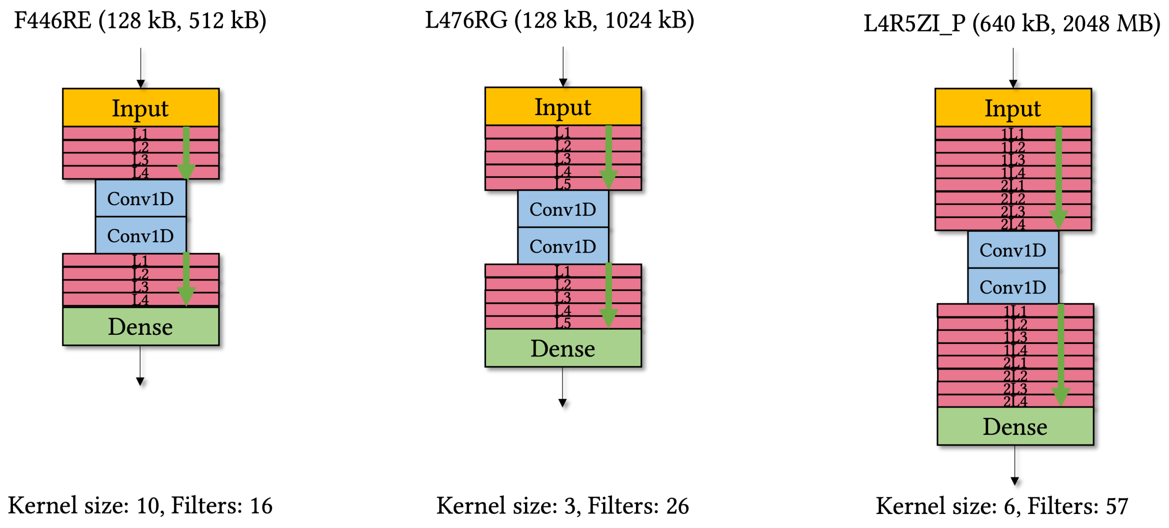

F446RE (128 kB, 512 kB)
L476RG (128 kB, 1024 kB)
L4R5ZI_P (640 kB, 2048 MB)
Input
Conv1D
Conv1D
Dense
Kernel size: 10, Filters: 16
Kernel size: 3, Filters: 26
Kernel size: 6, Filters: 57

Figure 3.4: 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.

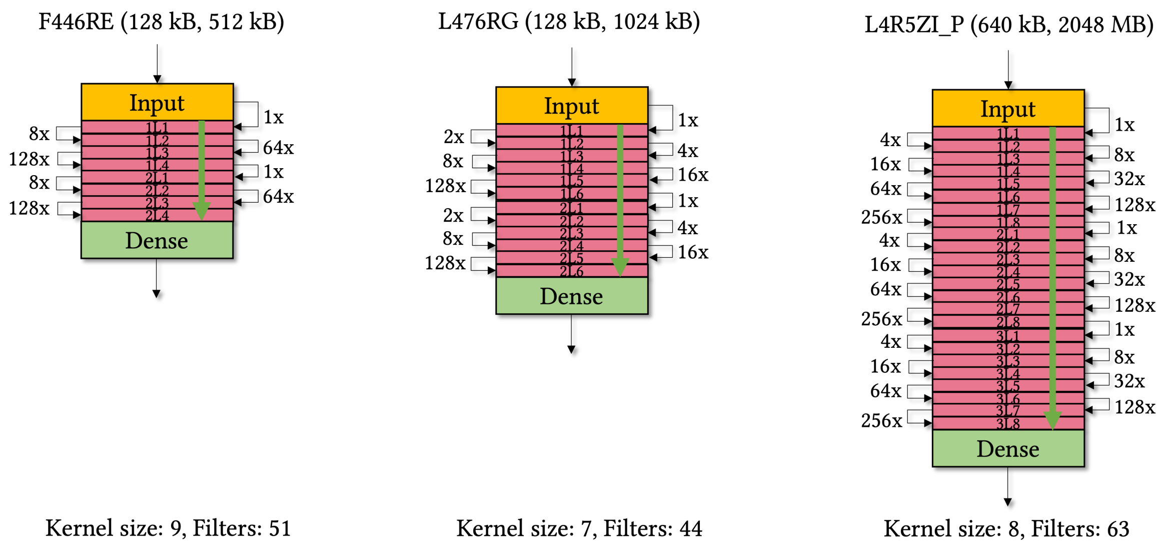

F446RE (128 kB, 512 kB)
L476RG (128 kB, 1024 kB)
L4R5ZI_P (640 kB, 2048 MB)
Input
Dense
Kernel size: 9, Filters: 51
Kernel size: 7, Filters: 44
Kernel size: 8, Filters: 63

Figure 3.5: 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.

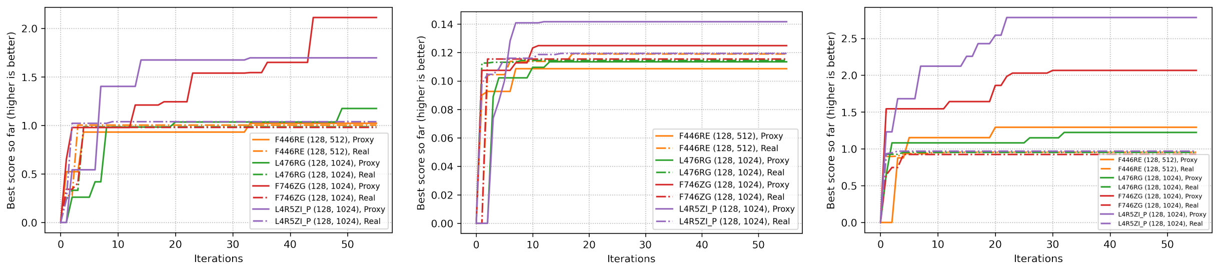


Figure 3.6: Convergence steps required for proxy less and proxied TinyNS. (Left) CIFAR10, (Center) ToyADMOS, (Right) Google Speech Commands. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash). Note that a higher score for proxied TinyNS does not necessarily guarantee deployability, while the highest score for proxy less TinyNS guarantees deployability on the target microcontroller.

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.

3.5 Discussion

TinyNS provides a stepping stone in automating the deployment of neurosymbolic frameworks onto ultra-resource-constrained IoT devices like microcontrollers. 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.

Our framework only supports TensorFlow Lite Micro (TFLM) [DDJ21] so far for model parsing. However, there are other inference engines for which support must be added. Moreover, while our Bayesian optimizer provides a user-friendly and problem-agnostic way to inexpensively optimize complex search spaces, the approach is likely sub-optimal to RL or evolutionary approaches. A detailed benchmark needs to be performed to study the viability of RL or evolutionary techniques as the search algorithm.

CHAPTER 4

Neurosymbolic Tiny Machine Learning

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 [SSG22, SFM20, CGH20, KKL21]. *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 [RAR21a, XGV20, APS21, VXT21, MTC22]. *Thirdly*, ML models lack transparency and interpretability, with the decision trace (e.g., causation versus correlation) and learned features difficult to understand [MGK18, MTC22, GGL19, YWG18, Pea19, SSS22b].

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 [MGF20, XGV20, MGK18, MDK18, PMS17, GBB22, SG16, GDY19, YPJ19, SZS20, SZE21]. Neurosymbolic AI integrates NNs with expert principles expressed as probabilistic reasoning modules, logical reasoning modules, knowledge graphs, question/answering engines, and constraint satisfaction functions [SZE21, GBB22]. Concatenation of neural and symbolic reasoning has been successful in a broad spectrum of challenging problems. These include complex event recognition [XGV20, VXT21, APS21, RAR21a, WSW22], commonsense reasoning [SLA19, BRS19], visual question answering [MGK18, YWG18], oceanographic forecasting [CA98, FC03], autonomous driving [SSH21, SSS17, GTC20], business management [CBP04, BBC11], and bioinformatics [AKM17, KKH18]. Thereby, neurosymbolic AI can enable rich, complex, and intelligent inferences at the extreme edge beyond the perception of atomic events [SSS22a, RAR21a, VVM21]. However, real-time adoption of neurosymbolic frameworks on extremely resource-constrained platforms such as microcontrollers. Directly porting existing neurosymbolic frameworks on microcontrollers, in-sensor processors [CD18], and field-programmable gate arrays [JZS19] is not computationally tractable.

4.1 Contributions

Building upon the neurosymbolic architecture search framework described in Chapter 3, we showcase the neurosymbolic program formulation in TinyNS. Using case studies, we showcase recipes for defining the neurosymbolic program synthesis search space for all five neurosymbolic program categories [SZE21]. Our framework includes parsers that automatically write microcontroller code according to these recipes. TinyNS provides recipes to map neurosymbolic program atoms from a prototyping language (e.g., Python) to a deployment language (e.g., C). We showcase several unseen TinyML applications made possible by the joint optimization of neural and symbolic components [SSA23, SDS23b, DSS23, SDS23a, SSP22].

4.2 What is 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 [LBH15]. However, traditional DL is data-hungry even for simple tasks, lacks interpretability and explainability, does not guarantee to follow rules, physics, and con-

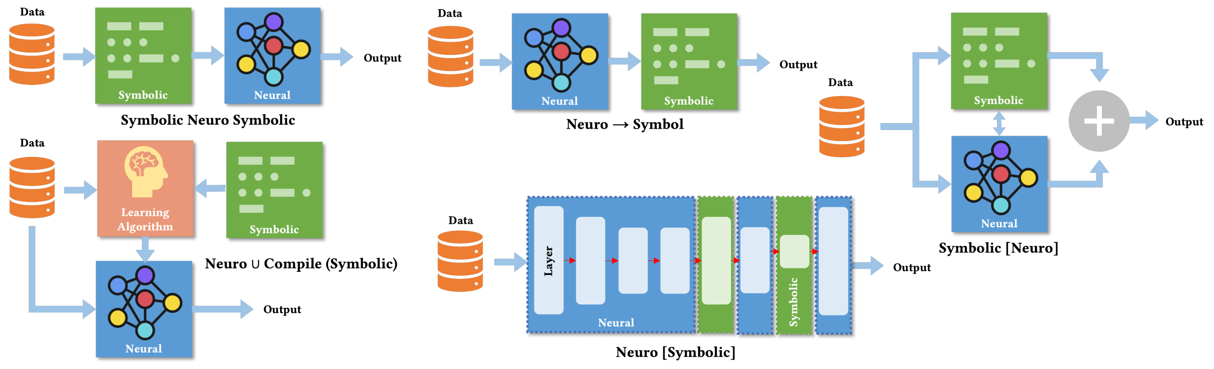


Figure 4.1: The five categories of neurosymbolic artificial intelligence [Kau22, SZE21].

straints, fails on feature distribution shifts, and struggles to learn long-range temporal patterns [CGH20, GGL19, SSS22b, Pea19, GBB22]. The flipside is symbolic AI, which was once the dominant trend of AI research several decades ago before the prevalence of DL [Smo87, New80]. 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 [SZE21]. 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 [GBB22, GDY19, Kau22, MDK18, PMS17, SZE21]. Neurosymbolic learning is analogous to the two types of human reasoning [Kah11]: 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.

4.2.1 Taxonomy of Neurosymbolic AI

Neurosymbolic AI systems are categorized into five groups [SZE21, Kau22], as illustrated in Fig. 4.1:

- **Symbolic Neuro Symbolic** or **Neural-after-Symbolic**: This is the most common paradigm [Kau22]. 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 [KKN14] and graph NN inference with pre-processed graph nodes [SGT08]. 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 [MTC22] and DeepProbLog [MDK18]. 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 [LC19]. An example includes Pylon [ALT22], 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 [MGF20], 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 [Kau22], 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 [SG16], which generates a first-order logic language into TensorFlow computational graphs. Pix2rule [CR22] 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 [XGV20] 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.

4.2.2 Neurosymbolic Language Tools

Neurosymbolic language tools synthesize programs from user-defined rules. DeepProbLog [MDK18] 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 [ALT22] is a PyTorch framework that learns deep 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 [CSL19] 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.

4.2.3 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 [SZE21, GBB22]. Symbolic background knowledge allows extrapolation when dealing with input distribution different from training data [MFL19]. Error correction designs robust ML systems enabling streamlined recovery from wrong outputs without retraining on new data [BKS18]. Symbolic reasoning allows NNs to be trainable with less data [SZE21]. Improving the interpretability of ML systems makes NN decisions more transparent and explainable [MA20]. Unfortunately, the deployment of neurosymbolic programs on IoT platforms or for real-time inference has received little attention. μCEP [RAR21a] 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$\rightarrow$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.

4.3 TinyML Neurosymbolic Problem Formulation

The recipes used by the parsers in TinyNS are discussed in this section.

4.3.1 Symbolic Neuro Symbolic

Problem Formulation (Symbolic). Consider a vector of independent domain-engineered functions $\mathbf{z}(\cdot)$ constructed from s in $\mathbf{\Omega}$ 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 $\mathbf{z}(\cdot)$.

$$\mathbf{X}_i^{\text{feat}} = z_i^{\mathbf{U}_i}(\mathbf{X}) \Leftrightarrow c_i = 1,\;\; i \in [1, n],\;\; c_i \in 0 \vee 1 \tag{4.1}$$

$\mathbf{U}$ 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 parameters for that z_i which takes the maximum 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 $(i, j)^{\text{th}}$ hyperparameter, 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.

$$\mathbf{U} = \begin{bmatrix} [\alpha_1^{1,1}, \alpha_2^{1,1}, ..., \alpha_{\gamma_1^1}^{1,1}] & [\alpha_1^{1,2}, \alpha_2^{1,2}, ..., \alpha_{\gamma_2^1}^{1,2}] & ... & [\alpha_1^{1,e}, \alpha_2^{1,e}, ..., \alpha_{\gamma_e^1}^{1,e}] \\ [\alpha_1^{2,1}, \alpha_2^{2,1}, ..., \alpha_{\gamma_1^2}^{2,1}] & [\alpha_1^{2,2}, \alpha_2^{2,2}, ..., \alpha_{\gamma_2^2}^{2,2}] & ... & [\alpha_1^{2,e}, \alpha_2^{2,e}, ..., \alpha_{\gamma_e^2}^{2,e}] \\ . & . & ... & . \\ . & . & ... & . \\ . & . & ... & . \\ [\alpha_1^{n,1}, \alpha_2^{n,1}, ..., \alpha_{\gamma_1^n}^{n,1}] & [\alpha_1^{n,2}, \alpha_2^{n,2}, ..., \alpha_{\gamma_2^n}^{n,2}] & ... & [\alpha_1^{n,e}, \alpha_2^{n,e}, ..., \alpha_{\gamma_e^n}^{n,e}] \end{bmatrix} \tag{4.2}$$

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.

$$\mathbf{U}_{\text{sample}} = \begin{bmatrix} [0.0, 1.0] & \text{range}(3.0, 64.0) & \text{uniform}(-5.0, 10.0) & [1.2, 5.2] \\ [0.2, 0.5, 0.8, 1.5, 2.3] & [0.0] & [0.0] & [0.0] \\ [1.0, 2.0, 3.0, 4.0] & \text{linspace}(-22.0, 22.0, 100) & [0.0] & [0.0] \end{bmatrix} \tag{4.3}$$

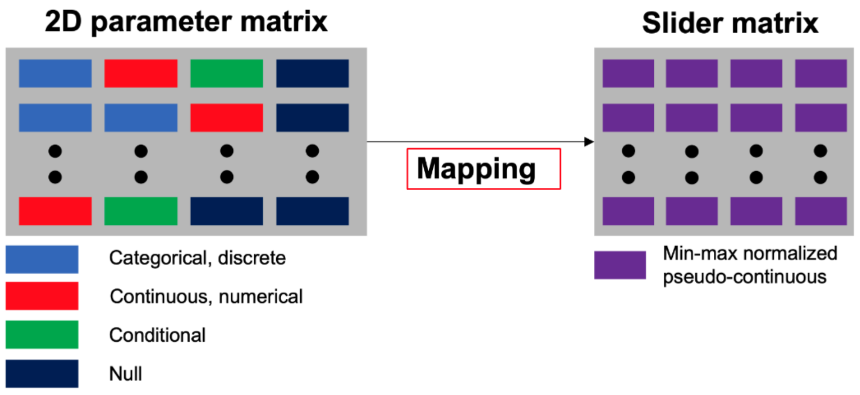


Figure 4.2: Mapping hyperparameter data structure to slider matrix.

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 $\mathbf{U}$.

$$\mathbf{U}_{\text{slider}} = \begin{bmatrix} \zeta_{1,1} & \zeta_{1,1} & \cdots & \zeta_{1,e} \\ \zeta_{2,1} & \zeta_{2,2} & \cdots & \zeta_{2,e} \\ . & . & \cdots & . \\ . & . & \cdots & . \\ . & . & \cdots & . \\ \zeta_{n,1} & \zeta_{n,2} & \cdots & \zeta_{n,e} \end{bmatrix}, \quad \zeta_{i,j} = \begin{cases} \text{linspace}(0,1,\delta) \Leftrightarrow \left|[\alpha_1^{i,j}, \alpha_2^{i,j}, ..., \alpha_{\gamma_j^i}^{i,j}]\right| \neq 1 \\ 0 \end{cases} \tag{4.4}$$

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:

$$\eta_{i,j} = \alpha_\kappa^{i,j}, \quad \kappa = \text{round}\left(\zeta_{i,j} \cdot \left|[\alpha_1^{i,j}, \alpha_2^{i,j}, ..., \alpha_{\gamma_j^i}^{i,j}]\right|\right), \quad \mu_{i,j} \in [0,1] \tag{4.5}$$

The search space for the symbolic components, thereby, is composed of the binary mask c and $\mathbf{U}_{\text{slider}}$. Fig. 4.2 illustrates the mapping.

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

$$\text{model}_{\text{iteration}_t} = \phi_i, \;\; i \in d, \;\; d = [1, 2, ..., k] \tag{4.6}$$

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 hyperparameter. The number of columns of $\mathbf{V}$ is equal to the number of optimization hyperparameters for that ϕ_i which takes the maximum number of arguments f.

$$\mathbf{V} = \begin{bmatrix} [\beta_1^{1,1}, \beta_2^{1,1}, ..., \beta_{\gamma_1^1}^{1,1}] & [\beta_1^{1,2}, \beta_2^{1,2}, ..., \beta_{\gamma_2^1}^{1,2}] & ... & [\beta_1^{1,f}, \beta_2^{1,f}, ..., \beta_{\gamma_f^1}^{1,f}] \\ [\beta_1^{2,1}, \beta_2^{2,1}, ..., \beta_{\gamma_1^2}^{2,1}] & [\beta_1^{2,2}, \beta_2^{2,2}, ..., \beta_{\gamma_2^2}^{2,2}] & ... & [\beta_1^{2,f}, \beta_2^{2,f}, ..., \beta_{\gamma_f^2}^{2,f}] \\ . & . & ... & . \\ . & . & ... & . \\ . & . & ... & . \\ [\beta_1^{k,1}, \beta_2^{k,1}, ..., \beta_{\gamma_1^k}^{k,1}] & [\beta_1^{k,2}, \beta_2^{k,2}, ..., \beta_{\gamma_2^k}^{k,2}] & ... & [\beta_1^{k,f}, \beta_2^{k,f}, ..., \beta_{\gamma_f^k}^{k,f}] \end{bmatrix} \tag{4.7}$$

An example of $\mathbf{V}$ is shown below. The first row corresponds to the hyperparameters for a temporal convolutional network (TCN) [ODZ16], and the second row corresponds to the hyperparameters for Bonsai [KGV17].

$$\mathbf{V}_{\text{sample}} = \begin{bmatrix} \underbrace{\text{range}(2,64)}_{\text{kernel size}} & \underbrace{[1.0, 2.0, 5.0]}_{\text{stack count}} & \underbrace{[[1,2,4],[1,2,4,8],[1,4,8,32]]}_{\text{dilation factors}} & \underbrace{\text{uniform}(0.0,1.0)}_{\text{dropout}} \\ \underbrace{\text{range}(40,60)}_{\text{prototype count}} & \underbrace{\text{range}(1,4)}_{\text{sigmoid parameter}} & \underbrace{\text{range}(1,6)}_{\text{depth}} & [0.0] \end{bmatrix} \tag{4.8}$$

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

$$\mathbf{V}_{\text{slider}} = \begin{bmatrix} \chi_{1,1} & \chi_{1,2} & ... & \chi_{1,f} \end{bmatrix}, \;\; \chi_{i,j} = \begin{cases} \text{linspace}(0,1,\delta) \Leftrightarrow \left| [\beta_1^{i,j}, \beta_2^{i,j}, ..., \beta_{\gamma_j^i}^{i,j}] \right| \neq 1 \\ 0 \end{cases} \tag{4.9}$$

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();
}
```

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) [DDJ21] 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 [DDJ21, SSS22a]. First, the model backbone in Python is constructed using Keras [GP17] or Keras/TensorFlow wrappers for Scikit-learn [PVG11] with TensorFlow backend [ABC16]. 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 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.
- 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.

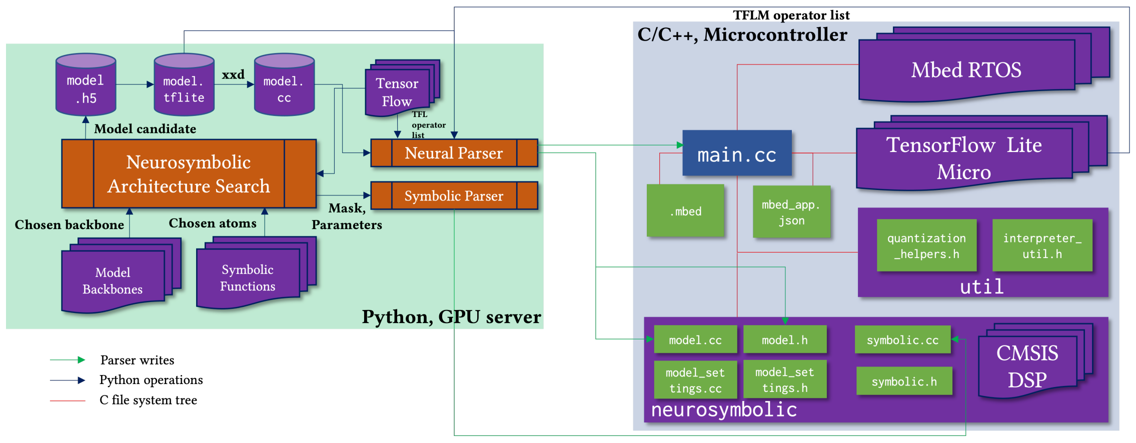
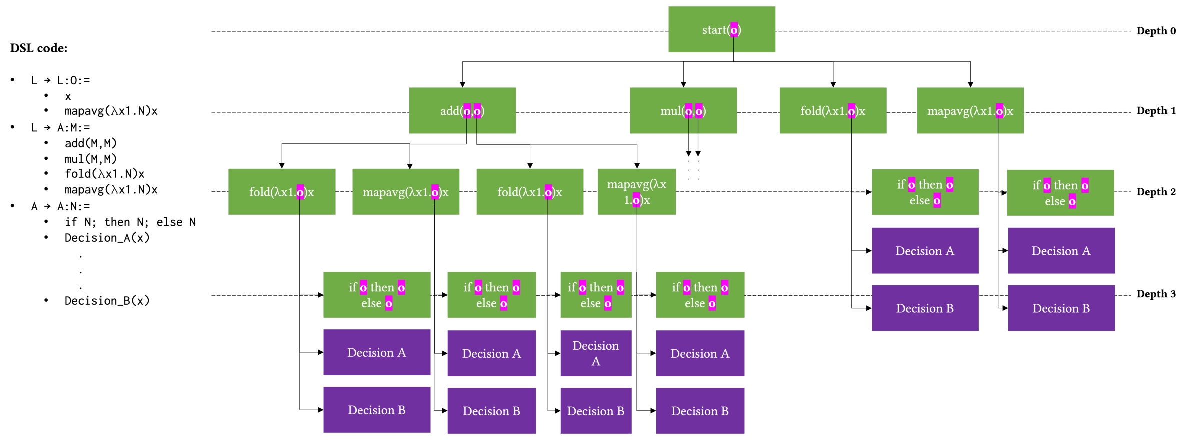


Figure 4.3: Architecture of automated neurosymbolic parsing for Symbolic Neuro Symbolic.

- 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. 4.3 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 4.4.1 and Section 4.4.2. 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


DSL code:

```
•  L → L:O:=
     •  x
     •  mapavg(λx1.N)x
•  L → A:M:=
     •  add(M,M)
     •  mul(M,M)
     •  fold(λx1.N)x
     •  mapavg(λx1.N)x
•  A → A:N:=
     •  if N; then N; else N
     •  Decision_A(x)
          .
          .
          .
     •  Decision_B(x)
```

Figure 4.4: Sample program supergraph generated from the DSL operator space for Neuro → Symbol [PMS17, SZS20, TSK21]. Green nodes represent non-terminal nodes and purple nodes represent goal nodes.

hyperparameters that can detect falls within a tight memory budget.

4.3.2 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 $\mathbf{u}$.

$$\mathbf{u} = \Big[[\alpha_1^{1,1}, \alpha_2^{1,1}, ..., \alpha_{\gamma_1^1}^{1,1}] \quad [\alpha_1^{1,2}, \alpha_2^{1,2}, ..., \alpha_{\gamma_2^1}^{1,2}] \quad ... \quad [\alpha_1^{1,e}, \alpha_2^{1,e}, ..., \alpha_{\gamma_e^1}^{1,e}] \Big] \tag{4.10}$$

$\mathbf{u}$ is similar in form as $\mathbf{U}$ from Section 4.3.1, but only corresponds to the optimization hyperparameter space for a single function. The neural search space is the same as that shown in Section 4.3.1.

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 $\Re$, arithmetic (e.g., add, multiply) operators Ξ, and conditional (e.g., if else then) operators Υ, expressed in a *Domain-Specific Language* (DSL) [PMS17]. 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.

$$\mathbf{G} = \text{GenerateProgramTree}(\{\Lambda, \Re, \Xi, \Upsilon\}, \wp, N) \tag{4.11}$$

Fig. 4.4 shows an example program supergraph generated from the DSL operator space, from which candidate trees can be extracted. The GenerateProgramTree() is an enumeration algorithm [PMS17, TSK21] that generates all possible combinations of program graphs $\mathbf{G}$ 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. 4.4, 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.

$$\text{program}_{\text{iteration}_t} = G_i, \;\; G_i \in \mathbf{G}, \;\; i \in j, \;\; j = [1, 2, ..., N] \tag{4.12}$$

The neural search space is the same as that shown in Section 4.3.1.

Parsing. Neuro→Symbol follows the same model parsing strategy discussed in Section 4.3.1, Algorithm 2 and Fig. 4.3. 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 [Par13] 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.

- 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 [ZKK20]. We showcase this example in Section 4.4.3. 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 $c \in \{0, 1, ..., C-1\}$.

$$\mathcal{R}_q(\{\mathbf{p}_0, \mathbf{p}_1, ...\}) = \max_i \exp\left(\frac{(\mathbf{p}_i - \mathbf{q})^2}{2\sigma_i^2}\right), \mathbf{q} \in \mathbb{R}^2, \mathbf{p} \in \mathbb{R}^2 \tag{4.13}$$

$\mathbf{q}$ 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

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();
}
```

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 $\mathbf{s}$), and the local location regression $\mathcal{L}_{\text{off}}$ (based on ground truth object positions $\mathbf{p}_i$).

$$\mathcal{L}_k = \frac{1}{N}\sum_{xyc}\begin{cases}(1-\hat{Y}_{xyc})^2\log(\hat{Y}_{xyc}) \Leftrightarrow Y_{xyc}=1\\(1-Y_{xyc})^4(\hat{Y}_{xyc})^2\log(1-\hat{Y}_{xyc})\end{cases} \tag{4.14}$$

$$\mathcal{L}_{\text{size}} = \frac{1}{N}\sum_{i=1}^{N} |\hat{S}_{\mathbf{p}_i} - \mathbf{s}_i| \tag{4.15}$$

$$\mathcal{L}_{\text{off}} = \frac{1}{N}\sum_{i=1}^{N} \left|\hat{D}_{\mathbf{p}_i^t} - (\mathbf{p}_i^{t-1} - \mathbf{p}_i^t)\right| \tag{4.16}$$

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) [ZKK20].

4.3.3 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. 3.7:

$$\min f_{\text{opt}}, \;\; f_{\text{opt}} = \lambda_1 f_{\text{error}}(\mathbf{\Omega}')+\lambda_2 f_{\text{flash}}(\mathbf{\Omega}')+\lambda_3 f_{\text{SRAM}}(\mathbf{\Omega}')+\lambda_4 f_{\text{latency}}(\mathbf{\Omega}')+\lambda_5 f_{\text{rule 1}}(\mathbf{\Omega}')+\lambda_6 f_{\text{rule 2}}(\mathbf{\Omega}')+\ldots \tag{4.17}$$

$\mathbf{\Omega}'$ contains only the ML components (i.e., $\mathbf{\Omega}' = \{\{V, E\}, \theta_m, m, w\}$), 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 $\mathbf{\Omega}'$. *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:

$$\min f_{\text{opt}}, \;\; f_{\text{opt}} = \lambda_1 f_{\text{error}}(\mathbf{\Omega}') + \lambda_2 f_{\text{flash}}(\mathbf{\Omega}') + \lambda_3 f_{\text{SRAM}}(\mathbf{\Omega}') + \lambda_4 f_{\text{latency}}(\mathbf{\Omega}') \tag{4.18}$$

where,

$$f_{\text{error}}(\mathbf{\Omega}') = \mathcal{L}_{\text{validation}}(\mathbf{Y}', \mathbf{Y}), \;\; \mathbf{Y}' = q^{\mathbf{\Omega}'}(\mathbf{X}, \mathbf{x}_{\text{physics metadata channel}}) \tag{4.19}$$

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 4.3.1, Algorithm 2 and Fig. 4.3. 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.

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 $f_{\text{rule 1}}(\boldsymbol{\Omega}')$ denotes the white-box adversarial robustness score from RobustBench [CAS21] or AutoAttack [CH20] 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.

$$f_{\text{rule 1}}(\boldsymbol{\Omega}') = 1 - \frac{1}{N}\sum_{i=0}^{N} q_i, \quad q_i = \begin{cases} 1 \Leftrightarrow y'^{x_i} = y'^{x_i,\text{perturbed}} \\ 0 \end{cases} \tag{4.20}$$

where,

$$x_{i,\text{perturbed}} = \underbrace{\left[x_i + \varepsilon \cdot \text{sign}\left(\nabla_{x_i}\mathcal{L}_{\text{validation}}(q^{\boldsymbol{\Omega}'}(x_i), y^i)\right)\right]}_{\text{FGSM}} \vee \underbrace{\left[\text{clip}_\varepsilon\left(x_i^t + \alpha \cdot \text{sign}\left(\nabla_{x_i}\mathcal{L}_{\text{validation}}(q^{\boldsymbol{\Omega}'}(x_i)^t, y^i)\right)\right)\right]}_{\text{PGD}} \tag{4.21}$$

α and ε are attack strength hyperparameters in Eq. 4.21. 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 [SSG22].

$$\mathbf{x}_{\text{physics metadata channel}} = c({}^I\hat{\mathbf{a}}), \;\; c_j({}^I\hat{\mathbf{a}}) = \underbrace{\begin{cases} 1 \Leftrightarrow \hat{\mathbf{a}}^I_{L,\Delta t} > \zeta \cdot \sqrt{\frac{\sum_{k\in\Delta t}\left(\hat{\mathbf{a}}^I_{L,k} - \overline{\hat{\mathbf{a}}^I_{L,\Delta t}}\right)^2}{n}} \\ 0 \end{cases}}_{\text{step detector}} \vee \underbrace{\overline{|\text{FFT}(|\hat{\mathbf{a}}^I_{\Delta t}|)|}}_{\text{Fourier transform}} \tag{4.22}$$

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.

$$\mathbb{E}(y'_j) \to 0 \mid x_{j,\text{physics metadata channel}} < \tau \tag{4.23}$$

We showcase the examples in Section 4.4.4 and Section 4.4.5.

4.3.4 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 iteration $k+1$. The neural system evolution is given as follows:

$$\hat{\mathbf{x}}_{k+1|k} = g_v(\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}, \mathbf{w}_{k+1}) \tag{4.24}$$

$\mathbf{w}_{k+1}$ 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.

$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + \mathbf{K}_{k+1} \left(\underbrace{\mathbf{z}_{k+1} - h_u(\hat{\mathbf{x}}_{k+1|k}, \mathbf{v}_k)}_{\text{measurement residual}} \right) \tag{4.25}$$

$\mathbf{v}_k$ 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 [DSS23].

$$\mathbf{P}_{k+1|k} = \mathbf{A}\mathbf{P}_k\mathbf{A}^T + \mathbf{B}_{k+1}\mathbf{U}_k\mathbf{B}_{k+1}^T + \mathbf{Q}_k, \quad \mathbf{A}_{k+1} = \left.\frac{\partial g}{\partial x}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}, \mathbf{w}_{k+1}}, \quad \mathbf{B}_{k+1} = \left.\frac{\partial g}{\partial u}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}_{k+1}, \mathbf{w}_{k+1}} \tag{4.26}$$

$$\mathbf{P}_{k+1|k+1} = (\mathbf{I} - \mathbf{K}_{k+1}\mathbf{H}_{k+1})\,\mathbf{P}_{k+1|k}, \quad \mathbf{H}_{k+1} = \left.\frac{\partial h}{\partial x}\right|_{\hat{\mathbf{x}}_{k+1|k}, \mathbf{v}_k} \tag{4.27}$$

where,

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^T \left(\underbrace{\mathbf{H}_{k+1}\mathbf{P}_{k+1|k}\mathbf{H}_{k+1}^T + \mathbf{R}_{k+1}}_{\text{innovation covariance}} \right)^{-1} \tag{4.28}$$

$\mathbf{A}_{k+1}$ 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 [SSF04]. 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:

$$\mathbf{u} = \Big[[\alpha_1^{1,1}, \alpha_2^{1,1}, ..., \alpha_{\gamma_1^1}^{1,1}] \;\; [\alpha_1^{1,2}, \alpha_2^{1,2}, ..., \alpha_{\gamma_2^1}^{1,2}] \;\; ... \;\; [\alpha_1^{1,e}, \alpha_2^{1,e}, ..., \alpha_{\gamma_e^1}^{1,e}]\Big] \tag{4.29}$$

$$\mathbf{v} = \Big[[\beta_1^{1,1}, \beta_2^{1,1}, ..., \beta_{\gamma_1^1}^{1,1}] \;\; [\beta_1^{1,2}, \beta_2^{1,2}, ..., \beta_{\gamma_2^1}^{1,2}] \;\; ... \;\; [\beta_1^{1,f}, \beta_2^{1,f}, ..., \beta_{\gamma_f^1}^{1,f}]\Big] \tag{4.30}$$

Parsing. The model parsing follows the same recipe shown in Section 4.3.1, Algorithm 2, and Fig. 4.3. 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 [LSC18] 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 (`GetJacobian()`).

Examples. We showcase a Neural-Kalman filter that fuses GPS measurements with a neural inertial odometry model to regress an object's position [DSS23]. The example is shown in Section 4.4.6. The neural network regresses the 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:

$$(v_{x,k}, v_{y,k}) = g\big(\mathbf{v}^I(0), \mathbf{g}_0^I, \mathbf{N}_0^I, \hat{\mathbf{a}}_{q:q+n}^I, \hat{\mathbf{w}}_{q:q+n}^I, \hat{\mathbf{m}}_{q:q+n}^I, c_k(^I\hat{\mathbf{a}})\big), \;\; c_k(^I\hat{\mathbf{a}}) = \left|\overline{|\mathrm{FFT}(|\hat{\mathbf{a}}_{q:q+n}^I|)|}\right|. \tag{4.31}$$

The system propagation is given as follows:

$$\hat{\mathbf{x}}_{k+1|k} = \mathbf{A}\hat{\mathbf{x}}_k + f(\mathbf{u}_{k+1}) \tag{4.32}$$

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

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////////////////////////
}
```

where,

$$\hat{\mathbf{x}} = \begin{bmatrix} \hat{L}_x \\ \hat{L}_y \\ v_x \\ v_y \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \mathbf{a}^I_{q:q+n} \\ \mathbf{w}^I_{q:q+n} \\ \mathbf{m}^I_{q:q+n} \\ c(\mathbf{a}^I_{q:q+n}) \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} \mathbf{I}_{2\times2} & \mathbf{0}_{2\times2} \\ \mathbf{0}_{2\times2} & \mathbf{0}_{2\times2} \end{bmatrix}, \quad \mathbf{B}_{k+1} = \begin{bmatrix} \frac{\Delta t \partial g_v(\cdot)_x}{\partial \mathbf{a}^I_{q:q+n}} & \frac{\Delta t \partial g_v(\cdot)_x}{\partial \mathbf{w}^I_{q:q+n}} & \frac{\Delta t \partial g_v(\cdot)_x}{\partial \mathbf{m}^I_{q:q+n}} & \frac{\Delta t \partial g_v(\cdot)_x}{\partial c(\mathbf{a}^I_{q:q+n})} \\ \frac{\Delta t \partial g_v(\cdot)_y}{\partial \mathbf{a}^I_{q:q+n}} & \frac{\Delta t \partial g_v(\cdot)_y}{\partial \mathbf{w}^I_{q:q+n}} & \frac{\Delta t \partial g_v(\cdot)_y}{\partial \mathbf{m}^I_{q:q+n}} & \frac{\Delta t \partial g_v(\cdot)_y}{\partial c(\mathbf{a}^I_{q:q+n})} \\ \frac{\partial g_v(\cdot)_x}{\partial \mathbf{a}^I_{q:q+n}} & \frac{\partial g_v(\cdot)_x}{\partial \mathbf{w}^I_{q:q+n}} & \frac{\partial g_v(\cdot)_x}{\partial \mathbf{m}^I_{q:q+n}} & \frac{\partial g_v(\cdot)_x}{\partial c(\mathbf{a}^I_{q:q+n})} \\ \frac{\partial g_v(\cdot)_y}{\partial \mathbf{a}^I_{q:q+n}} & \frac{\partial g_v(\cdot)_y}{\partial \mathbf{w}^I_{q:q+n}} & \frac{\partial g_v(\cdot)_y}{\partial \mathbf{m}^I_{q:q+n}} & \frac{\partial g_v(\cdot)_y}{\partial c(\mathbf{a}^I_{q:q+n})} \end{bmatrix} \tag{4.33}$$

$$f(\cdot) = \begin{bmatrix} \Delta t \cdot \mathbf{I}_{2\times2} \\ \mathbf{I}_{2\times2} \end{bmatrix} \cdot g_v(\cdot), \quad \Delta t = \frac{s}{n-s}, \quad s = \text{stride}, n = \text{window size} \tag{4.34}$$

U consists of Allan variance parameters [EHN07] 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.

4.3.5 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 4.3.1, Algorithm 2, and Fig. 4.3, 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`.

4.4 Evaluation

In this section, we evaluate the performance of TinyNS on six different case studies resembling four neurosymbolic architecture search recipes (Section 4.4.1 to Section 4.4.6)

4.4.1 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.

4.4.1.1 Dataset and Task Description

We use the UCI-HAR dataset [AGO13] 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.

4.4.1.2 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]

Table 4.1: 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 Freq.	Max. Power Spec.
F446RE (128, 512)	🟦 Mean	IQR	Maximum	Median	🟦 Variance	MAD	🟦 Abs. Energy	🟦 Entropy	Peak-to-Peak	🟦 FFT Mean Coeff.	Fundamental Freq.	Max. Power Spec.
L476RG (128, 1024)	🟦 Mean	IQR	Maximum	Median	🟦 Variance	🟦 MAD	🟦 Abs. Energy	🟦 Entropy	🟦 Peak-to-Peak	🟦 FFT Mean Coeff.	Fundamental Freq.	Max. Power Spec.
F746ZG (320, 1024)	🟦 Mean	IQR	🟦 Maximum	🟦 Median	🟦 Variance	MAD	Abs. Energy	🟦 Entropy	Peak-to-Peak	🟦 FFT Mean Coeff.	Fundamental Freq.	Max. Power Spec.
L4R5ZI_P (640, 2048)	🟦 Mean	🟦 IQR	Maximum	🟦 Median	🟦 Variance	🟦 MAD	Abs. Energy	Entropy	🟦 Peak-to-Peak	🟦 FFT Mean Coeff.	🟦 Fundamental Freq.	Max. Power Spec.

- 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 4.1. 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 4.3.1.

4.4.1.3 Target Hardware

We perform neurosymbolic optimization for the same four microcontrollers from Section 3.4.2. 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

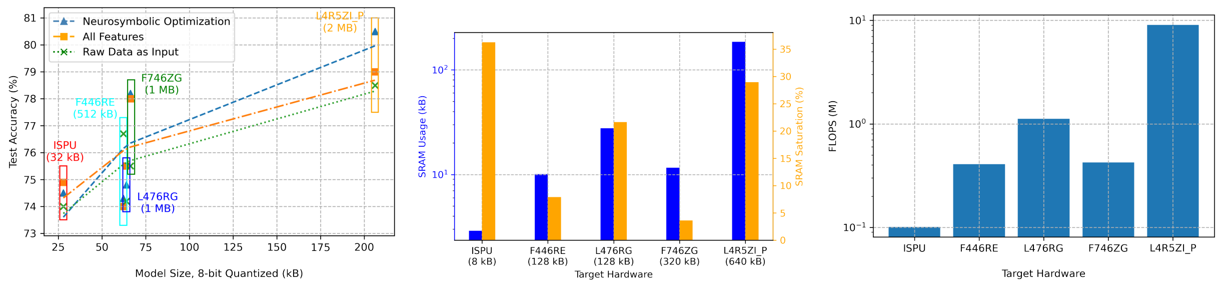


Figure 4.5: (Left) Flash usage of models found via neurosymbolic optimization of features and model hyperparameters. The accuracy of the said models operating on all features and directly on the raw data is also shown. Flash limits of the target hardware are shown in parentheses. (Center) SRAM usage 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.

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 [MRS22, RSZ22].

4.4.1.4 Overall Performance

Fig. 4.5 (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 4.1 and Table 4.2 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

Table 4.2: 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

capacities. These features are well-known to have the highest effect on classifier performance in human activity recognition literature [AMD15, WCY19]. 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 3.4.2, 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. 4.5 (Center) and Fig. 4.5 (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 model hyperparameters makes it possible to run models on extremely resource-constrained platforms beyond microcontrollers like the ISPU.

4.4.2 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.

4.4.2.1 Dataset and Task Description

We use the Auritus dataset [SSP22] 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.

4.4.2.2 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 and shown in Fig. 4.6.

- **Temporal Convolution**: Without explosion of parameter, memory footprint, layer count, or overfitting, TCN kernels allow the network to discover the global context in long inertial sequences while maintaining input resolution and coverage. In TCN, the convolution operation has three desirable properties:
  - *Causality*: The output of the operator at the current timestep t depends only on the current and past inputs but not future inputs. This ensures temporal ordering of the input sequences without requiring recurrent connections. The ordering is maintained via weight sharing among the input chunks.

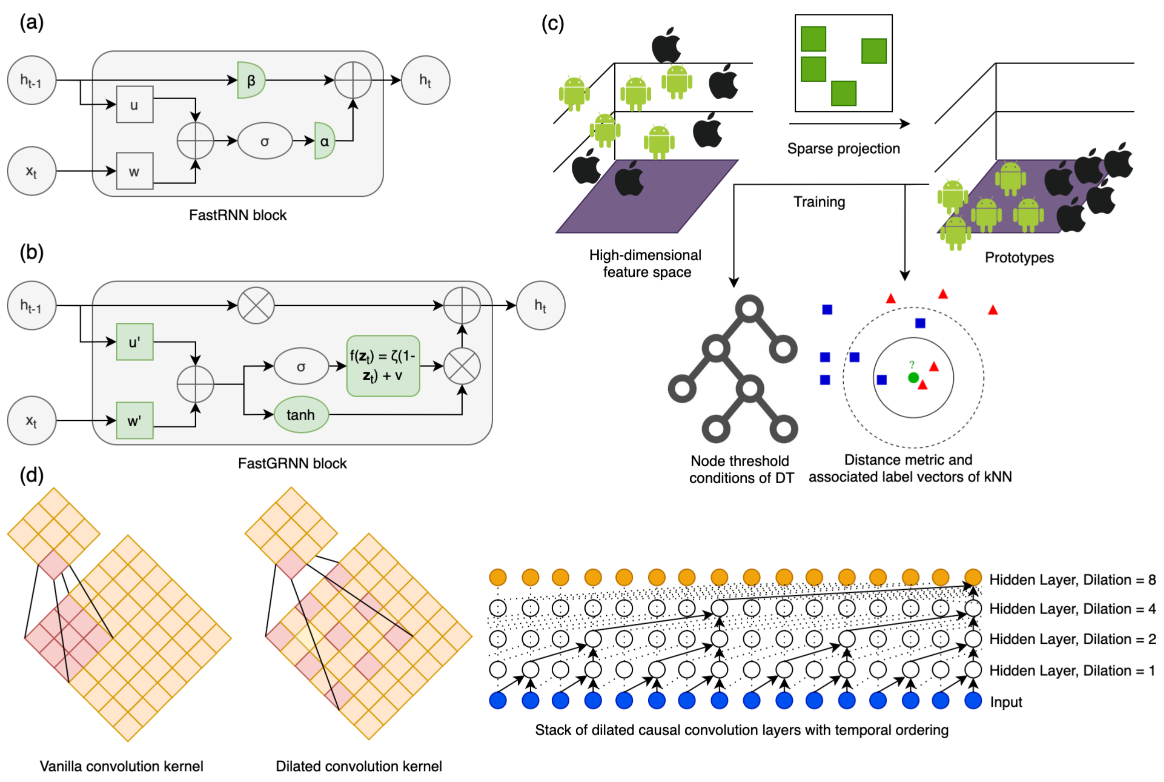

(a)
h_{t-1}
x_t
u
w
σ
α
β
h_t
FastRNN block
(b)
h_{t-1}
x_t
u'
w'
σ
tanh
f(z_t) = ζ(1-z_t) + ν
h_t
FastGRNN block
(c)
High-dimensional feature space
Sparse projection
Training
Prototypes
Node threshold conditions of DT
Distance metric and associated label vectors of kNN
(d)
Vanilla convolution kernel
Dilated convolution kernel
Hidden Layer, Dilation = 8
Hidden Layer, Dilation = 4
Hidden Layer, Dilation = 2
Hidden Layer, Dilation = 1
Input
Stack of dilated causal convolution layers with temporal ordering

Figure 4.6: Illustration of lightweight model architectures geared towards TinyML devices. (a) The addition of a residual connection with two scalars (α, β) stabilizes vanilla RNN training while taking advantage of the relative lightweightness of vanilla RNN against gated RNN. (b) Converting the residual connection to a gate while enforcing $\mathbf{U}$ and $\mathbf{W}$ to be LSQ yields lightweight yet accurate gated RNN. (c) Sparsely projecting input features to a low-dimensional space allows DT and kNN to be computationally efficient. (d) Enforcing causal convolution and dilated kernels allows spatial and temporal feature extraction in long time-series sequences without requiring recurrent connections or significant compute.

- *Dilated Convolution*: The receptive field F_i of each unit in the ith layer in a TCN dilated causal kernel of size $k \times k$ with dilation factor l is given by:

$$F_{i,\text{TCN}} = F_{i-1} + (k_l - 1) \times l, F_0 = 1 \tag{4.35}$$

  $F_{i,\text{TCN}}$ is larger than $F_{i,\text{CNN}}$, which is $i \times (k-1)+k$. When dilated CNN are stacked on top of each other, the dilation factor increases exponentially, increasing model capacity and receptive field size with fewer layers and parameter count over vanilla CNN or RNN

- *Residual Blocks:* Two stacks of dilated causal convolution layers, f and g, are fused through gated residual blocks $\mathbf{z}$ for expressive yet bounded non-linearity, complex interactions, and temporal correlation modeling in the input sequence:

$$\mathbf{z} = \tanh(\mathbf{W}_{f,k} * \mathbf{x}) \odot \sigma(\mathbf{W}_{g,k} * \mathbf{x}) \tag{4.36}$$

  where $\mathbf{W}$ are the weights in each layer, σ is the sigmoid function and $\mathbf{x}$ is the input.

- **Stabilized RNN with LSQ Matrices**: Vanilla RNN, albeit lightweight, suffers from exploding and vanishing gradient problem (EVGP) for long temporal sequences. Existing solutions to EVGP (e.g., gated RNN (long short-term memory (LSTM) and unitary RNN) come at the cost of either accuracy loss or increased memory and latency overhead. Fast RNN [KSB18] solves EVGP by adding a weighted residual connection with two scalars (α, β) to generate well-conditioned gradients:

$$\tilde{\mathbf{h}}_t = \sigma(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b}), \quad \mathbf{h}_t = \alpha\tilde{\mathbf{h}}_t + \beta\mathbf{h}_{t-1} \tag{4.37}$$

  where $0 \leq \alpha \ll 1, \beta \approx 1-\alpha, \beta \leq 1$, σ is a non-linear activation function, $\mathbf{W}$ and $\mathbf{U}$ are RNN matrices, $\mathbf{b}$ is bias vector, $\mathbf{h}$ is the hidden state and $\mathbf{x}$ is the input. By varying α and β, we can control the update extent of $\mathbf{h}_t$ based on $\mathbf{x}_t$. Fast GRNN [KSB18] then converts this residual connection to a gate while enforcing $\mathbf{W}$ and $\mathbf{U}$ to be low-rank, sparse, and quantized (LSQ):

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}'\mathbf{x}_t + \mathbf{U}'\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{4.38}$$

$$\mathbf{h}_t = (\zeta(\mathbf{1} - \mathbf{z}_t) + v) \odot \tilde{\mathbf{h}}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1}, \quad \mathbf{z}_t = \sigma(\mathbf{W}'\mathbf{x}_t + \mathbf{U}'\mathbf{h}_{t-1} + \mathbf{b}_z) \tag{4.39}$$

$$\mathbf{W}' = \mathbf{W}^1(\mathbf{W}^2)^\top, \ \mathbf{U}' = \mathbf{U}^1(\mathbf{U}^2)^\top \tag{4.40}$$

where, $\zeta \geq 0, v \leq 1$. Fast GRNN, thus, is able to provide the capabilities of gated RNN without the associated compute overhead.

- **Sparse Low-Dimensional Projection**: Bonsai [KGV17] is a shallow and sparse DT with non-linear activations, making inferences on data projected in low-dimensional space called prototypes. Similarly, ProtoNN [GSG17] is a lightweight k-nearest neighbor (kNN) classifier designed to operate on prototypes. The sparse projection matrix is learned using stochastic gradient descent and iterative hard thresholding. Sparsely projecting high-dimensional feature space onto a low-dimensional linear manifold reduces parameter count for Bonsai and ProtoNN, allowing them to be computationally efficient.

The search space is as follows:

- TCN (neural) [ODZ16, LVR16] - 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) [KSB18] - number of hidden units: range (20, 60).
- FastRNN (neural) [KSB18] - number of hidden units: range (20, 60).
- Bonsai (non-neural) [KGV17] - projection dimension: range (10, 70); sigmoid parameter: uniform (1.0, 4.0); depth: range(1, 6).

- ProtoNN (non-neural) [GSG17] - 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.

4.4.2.3 Overall Results

Fig. 4.7 summarizes the accuracy and model size for the highest performing models for each of the 5 backbones against competing models, while Table 4.3 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

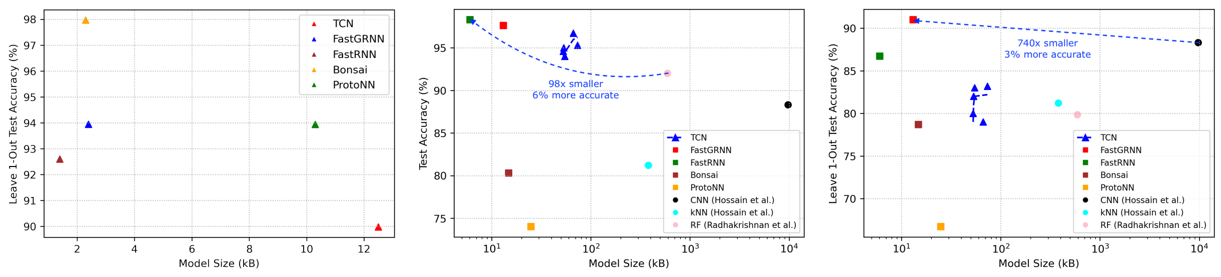


Figure 4.7: (Left) Highest performing models found by TinyNS for earable fall detection under 2 kB memory constraint when optimizing several model backbones. (Center and Right) Test accuracy and leave 1-out test accuracy of highest performing models found by TinyNS versus state-of-the-art earable activity detection classifiers when optimizing several model backbones. The TCN backbone is optimized for 5 different target hardware (eSense earable, F446RE, L476RG, F407VET6, and F746ZG).

may have more parameters). Even if one architecture performs poorly, the search algorithm would have other architectures to choose from. Thereby, exploring various architectures is important for squeezing highly performant models beyond microcontrollers, such as ISPU.

4.4.3 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.

4.4.3.1 Dataset and Task Description

We use the MOT17 dataset [MLR16] 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 [ZSJ22].

Table 4.3: 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
	None (hardware-agnostic)	**Hidden Units**			
FastGRNN		50			
FastRNN		32			
		Projection Dimension		**Sigmoid Parameter**	**Depth**
Bonsai		22		1.0	3
		Projection Dimension		**Prototypes**	γ
ProtoNN		70		70	0.004

4.4.3.2 Model Backbones and Search Space Definition

We use the ByteTrack library [ZSJ22] to implement the CenterNet algorithm [ZKK20], which was discussed in Section 4.3.2. 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)

Table 4.4: Chosen object detector and tracking filter hyperparameters for CenterNet algorithm under different size limits.

Constraint	Flash Usage (MB)	Performance		Model hyperparameters				Filter hyperparameters	
		MOTA	IDF1	Kernel size	Stack count	Head convolution	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

4.4.3.3 Overall Results

Table 4.4 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.

4.4.4 Improving Adversarial Robustness of TinyML Models (Neuro $\cup$ Compile (Symbolic))

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

4.4.4.1 Dataset and Task Description

We use the Auritus dataset in this case study (the same dataset used in Section 4.4.2). The goal and the dataset splits are the same as that in Section 4.4.2, except that now we

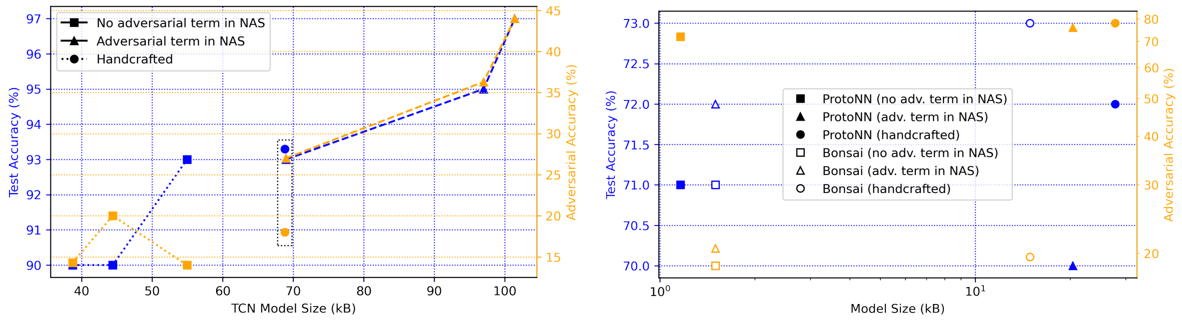


Figure 4.8: (Left) Test accuracy, adversarial accuracy, and model size of TCN backbones for three different target hardware (F446RE, L476RG, and F746ZG). (Right) Test accuracy, adversarial accuracy, and model size for ProtoNN and Bonsai backbones. For all three model backbones, the results are shown for NAS with adversarial robustness term, NAS without adversarial robustness term, and handcrafted models.

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 4.3.3).

4.4.4.2 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 4.4.2. 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 4.4.2.

4.4.4.3 Overall Results

Fig. 4.8 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 [PXJ22].

4.4.5 Physics-Aware Neural Inertial Localization (Neuro $\cup$ Compile (Symbolic))

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

4.4.5.1 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 [CZL20] and RoNIN [HYF20], AQUALOC [FCM19] unmanned underwater vehicle (UUV) tracking, EuRoC MAV [BNG16] undermanned aerial vehicle (UAV) tracking, and the GunDog [GHS21] animal tracking. The split information for all the datasets is shown in Table 4.5. The goal is to train a model

Table 4.5: 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.

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).

4.4.5.2 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 4.4.2.

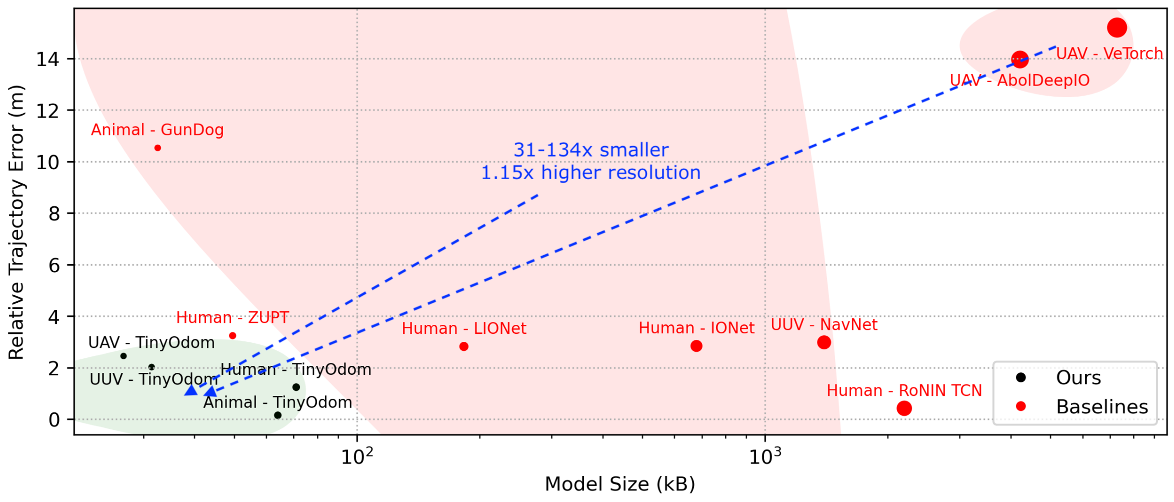


Figure 4.9: Odometric resolution of physics-aware neurosymbolic-inertial odometry models (TinyOdom) found via neurosymbolic architecture search, versus state-of-the-art handcrafted neural and symbolic models for tracking humans, animals, unmanned underwater vehicles (UUV), and unmanned aerial vehicles (UAV).

4.4.5.3 Overall Results

Fig. 4.9 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

Table 4.6: 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

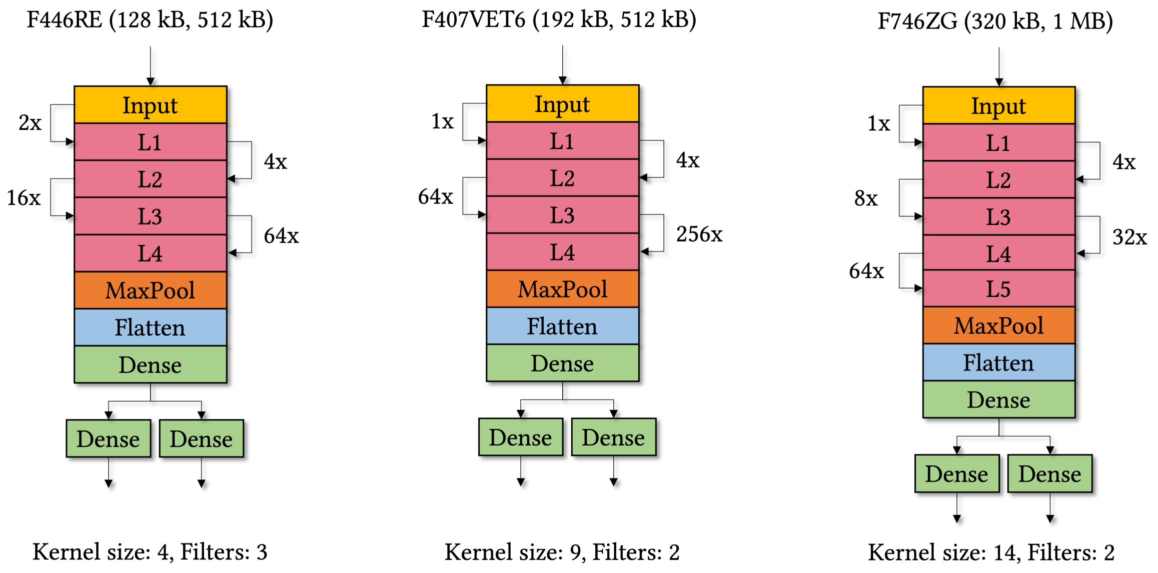


Figure 4.10: Architectural adaptation and device capability exploitation by TinyNS on the AQUALOC dataset. The SRAM and flash limits of the hardware are given in parenthesis in kB in the form (SRAM, Flash). L_i refers to i^{th} layer of the TCN.

microcontrollers. The superior performance is possible partly due to the inclusion of the physics channel, which improves the resolution by $1.1\times$ on average, as showcased in Table 4.6. 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. 4.10 visualizes the architectural adaptation 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.

4.4.6 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.

4.4.6.1 Dataset and Task Description

We use the AgroBot dataset [DSS23] 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.

4.4.6.2 Model Backbones, Training Details, Target Hardware, and Search Space Definition

We used the same model backbone and search space outlined in Section 4.4.5. In addition, we optimize noise parameters in the Kalman filter Allan variance matrix:

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

4.4.6.3 Overall Results

Table 4.7 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

Table 4.7: 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 [CLM18]	1.71	5.58 — 10.1	0.92 — 0.57
	L-IONet [CZL20]	0.55	8.11 — 18.6	0.91 — 1.40
	AbolDeepIO [EWW19]	12.5	7.24 — 20.5	0.96 — 0.93
	VeTorch [GXZ21]	29.6	2.86 — 15.6	0.44 — 0.84
Symbolic	UKF-M INS+GPS [BBC17]	0.192	5.50	0.49
	EKF INS+GPS [QM02]	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

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.

4.5 Discussion

Neurosymbolic AI provides a pathway for making context-aware, physics-aware, robust, interpretable, and performant AI systems. Through TinyNS, we have showcased state-of-the-art performance in various unseen applications. 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.

CHAPTER 5

Fine-Tuning, Online Learning, and Foundation Models

DL is innately data-hungry and struggles to personalize to deployment-time data distribution. For robust inference, onboard TinyML models need to be adaptively personalized using unlabeled data streams with minimal user input to account for domain shifts [SSS22a, SDS23b].

5.1 Contributions

We showcase *three* techniques to personalize and adapt TinyML models to target domains and applications. We start by showcasing the viability of transfer learning to adapt pre-trained models to deployment time feature characteristics using as little as 1 minute of labeled data. For collecting the labeled data, we introduce a user-friendly video-processing toolbox to generate high-resolution data for fine-tuning pre-trained models in the field for inertial odometry [SSG22, DSS23, SDS23b]. Next, we propose a method for personalized and online on-device learning for on-chip classification, inference, and information fusion applications for motion sensing. The framework requires no human-engineered parameters and allows for the personalization and addition of new motion artifacts [CS23]. Finally, we outline ongoing work on large language-inertial models (LLIM) to extract generalizable physics with language context from unlabeled data, which can be fine-tuned for domain-specific applications through few-shot learning.

Table 5.1: RTE (m) of neural-inertial models across different datasets (left) and applications (right) without fine-tuning. The training dataset or application is shown in parentheses.

Method	OxIOD	RoNIN
IONet (OxIOD)	2.84	4.7
IONet (RoNIN)	7.65	7.63
LIONet (OxIOD)	2.82	4.7
LIONet (RoNIN)	8.35	14.84
RoNIN TCN (OxIOD)	0.42	13.4
RoNIN TCN (RoNIN)	10.3	1.21
TinyOdom (OxIOD)	1.26	97.2
TinyOdom (RoNIN)	3.16	6.74

Method	PDR	UUV
IONet (PDR)	2.84	5.15
LIONet (PDR)	2.82	3.94
RoNIN TCN (PDR)	0.42	14.96
TinyOdom (PDR)	1.26	7.83
NavNet (UUV)	93	2.96
TinyOdom (UUV)	5.82	2.45

5.2 Fine-Tuning Pre-trained Models

Autoregressive applications such as neural-inertial navigation need large amounts of high-resolution training data in the target domain for providing acceptable resolution [SSG22, DSS23, SDS23a]. Table 5.1 shows an example of resolution degradation of neural-inertial models across domains. In general, these models perform well within the learned data distribution but fail to adapt to different motion artifacts and IMU characteristics in a new domain due to differences in learned physical embeddings across domains. Moreover, under-parameterized TinyML models overfit on the dataset-specific spatiotemporal features due to a lack of redundant weights, poorly generalizing across domains.

5.2.1 Data-Efficient Transfer-Learning

Instead of training new models from scratch using large amounts of training data in the new domain, we propose using transfer learning [PY10] to fine-tune pre-trained models using as little as 1 minute of labeled data in the target domain. We freeze some of the lower layers of the pre-trained NN and make the higher layers trainable.

Table 5.2 and Table 5.3 showcase the data efficiency and resolution improvement brought on by transfer learning. For the first case, fine-tuning reduces RTE by 1.6 - 13.6× in the target domain, while increasing data efficiency by >20×. In other words, 1 minute of fine-tuning exceeds training from scratch on 20 minutes of labeled data by 1.3×. For the second

Table 5.2: Fine-tuning pre-trained models across different phases of the AgroBot dataset

Training Dataset	**RTE (m) on Inference Dataset (Unseen Trajectory)**					
	P1	**P1 (FT)**	**P2**	**P2 (FT)**	**P3**	**P3 (FT)**
P1	1.10		14.5	1.45	15.0	4.96
P2	2.71	1.09	0.97		4.25	2.63
P3	1.85	0.76	1.93	1.15	2.58	

Method	**RTE (m) with T minutes of data in the new domain***		
	$T = 1$	$T = 5$	$T = 20$
Train from scratch	26.8	3.29	2.55
Fine-tune*	1.92	1.62	1.45

P1: Phase 1, P2: Phase 2, P3: Phase 3

FT: Fine-tuning with 20 minutes of data in the new domain for 100 epochs

* The pre-trained model was trained on Phase 1 data; target dataset: Phase 2

Table 5.3: Fine-tuning pre-trained OxIOD *TinyOdom* model on the AgroBot dataset

Method	**RTE (m) with T minutes of data in the new domain***							
	T= 1	T= 3	T= 5	T= 10	T= 20	T= 40	T= 75	T= 100
Train from scratch	3.26	1.39	1.09	0.94	0.86	0.83	0.81	0.80
Fine-tine	1.09	0.85	0.81	0.80	0.80	0.80	0.80	0.80

case, 1 minute of fine-tuning reduces the RTE by 8× for a pre-trained model with no fine-tuning, while 5 minutes of fine-tuning equals training from scratch on 100 minutes of labeled data. Pre-trained models already have some notion of inertial dynamics in a different domain, which models trained from scratch must learn, resulting in fine-tuning being data-efficient.

5.2.2 Collecting Labeled Data in New Domain

Fine-tuning still requires some high-resolution labeled inertial odometry data. Specialized motion capture systems suffer from limited coverage, high cost, use of specialized software, high computational requirements, and ambient lighting conditions [SLC20]. Vanilla GPS is noisy, with a maximum resolution of around 2 m [SDS23a, DSS23] (unless differential GPS is used, which can achieve centimeter-level accuracy at the cost of limited coverage, complexity, and time delay [MJ95]). We develop an automated data extraction framework operating on overhead quadrotor video feeds [DSS23] that mitigates the aforementioned limitations.

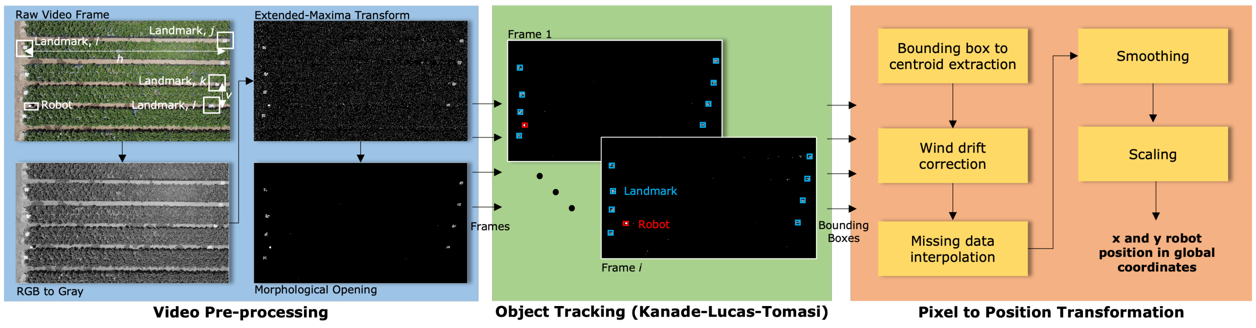


Figure 5.1: Automated pipeline to extract labeled inertial odometry data from monocular quadrotor video feeds.

Fig. 5.1 illustrates the automated inertial odometry data extraction pipeline. The user places several printed checkerboard patterns as reference landmarks at the boundaries of the quadrotor camera's field of view (FOV). The horizontal distance, h between landmark i and j, and the vertical distance, v between landmarks k and l are measured and noted. The object to be tracked is then moved within the FOV of the quadrotor camera. The IMU data is logged onboard the object, while the quadrotor camera records the object moving. The camera frames are synchronized with the IMU data using "static-rotate-static" motion patterns. The pipeline has three steps:

Video Pre-Processing: After converting the RGB video frames to grayscale, extended maxima transform [Soi99] and morphological opening [VV92] are applied to leave only the landmark and the object to be tracked in the frame.

Object Tracking: The user marks the bounding boxes for the object and the landmarks in the first frame. The Kanade-Lucas-Tomasi tracker [TK91], coupled with a minimum eigenvalue feature extractor [ST94], tracks the bounding boxes across subsequent frames.

Pixel to Position Transformation: The centroids of the bounding boxes are derived from the corner points, which are corrected for quadrotor wind drift by observing the movement of static landmarks. The user provides warm starts when the tracker loses the object or the landmarks due to light intensity changes. Linear interpolation fills the gaps between warm-starts and the last known object location. Median filtering [Jus81] cancels high-frequency tracking noise. Finally, the derived pixel positions are scaled with the scale factor s_x, s_y to convert to global coordinates as follows:

$$s_x = \frac{|C^i_{x,1} - C^j_{x,1}|}{h}, \quad s_y = \frac{|C^k_{y,1} - C^l_{y,1}|}{v}, \quad s_x \approx s_y. \tag{5.1}$$

$C^c_{a,b}$ is the centroid of landmark c at frame b for axis a. The pipeline, executable on commodity computers, provides ground truth locations at a resolution of ±5.0 cm.

5.3 Online Motion Recognition

Motion classification algorithms deployed in the wild require the ability to handle domain shifts in incoming data distribution, fit customer inference classes, and adapt to varying application scenarios [SSS22a, SSP22, CMI20, BBL19]. The same motion primitive can have multiple feature representation distributions across different users [SSP22, CZY21, GKS19, FMM20]. On-device fine-tuning and personalization are necessary for achieving robust algorithmic performance [SSS22a], preventing the adoption of static human-engineered heuristics. Moreover, each customer may have specific inference requirements depending on the application. For example, customer A may want to perform motion recognition for full-body fitness monitoring. Customer B may want a solution for anomaly detection of fan blades. Customer C may want to detect hand gestures. The diversity and variability in required motion primitives among users prevent the adoption of static ML classifiers, which can only detect the classes present in the training dataset. During fine-tuning, the incoming data stream must be properly segmented to store only the portion of data concerning the event of interest. The customer should not be burdened with the task of selecting the start and end times of the event of interest during training. In addition, the segmentation algorithm should fit within the compute and resource budget of the low-end IoT processor [SHS17, WSJ16, KB15].

Table 5.4: Existing TinyML On-Device Learning Techniques Versus Proposed Algorithm

Method	**Framework**	**Supported Hardware***	**Application**	**Personalizing Output Classes**	**Automatic Learning**
Transfer learning	LITW [LN20]	TI MSP430 (66 kB)	Image recognition	No (static CNN)	Per-output feature distribution divergence
	TinyOL [RAR21b]	ARM Cortex-M (256 kB)	Inertial anomaly detection	No (static autoencoder)	Running mean and variance of streaming input
	TinyTL [CGZ20]	ARM Cortex-A (32-66 MB)	Image recognition	No (static MBNet)	None
Incremental training	Train++ [SYB21]	ARM Cortex-M, ARM Cortex-A, Espressif ESP32, Xtensa LX (6 kB - 860 kB)	Image recognition, mHealth	No (static binary classifiers)	Confidence score of prediction
Optimized backpropagation	ML-MCU [Sud21]	ARM Cortex-M, Espressif ESP32 (6kB - 241 kB)	Image recognition, mHealth	No (static binary classifiers)	None
	TTE [LZC22]	ARM Cortex-M (256 kB)	Image recognition	No (static MBNet, MCUNet)	None
Continual learning	QLR-CL [RRN21]	PULP (64 MB)	Image recognition	No (static MBNetV1)	None
Template matching (ours)	This work	Any MCU, ISPU (8 kB)	Inertial motion recognition	Yes (template modification)	Running mean and variance of streaming input

* Parenthesis shows working memory (SRAM) usage

5.3.1 On-Sensor Learning and Classification

We designed an on-device inertial motion learning and classification framework for the execution on-sensor and on low-end microcontrollers. During the *learning* phase, the user performs the target event of interest with the IMU mounted on the target device. Our pipeline uses an inexpensive event segmentation method that can automatically identify the start and end points of the target motion primitive during the training phase without user intervention. The segmented data stream is then converted to an image template. The template and axes variance information is stored on the sensor. During *inference*, image templates are created in real-time from IMU data windows. These templates are matched against stored templates using image similarity and axes variance metrics to provide the detected class label. Customers can replace or remove stored classes with new motion primitives on-the-fly, allowing personalization and tuning for a broad application spectrum. The pipeline is fully automated and requires no user-supplied parameters. The whole algorithm fits within the tight mem-

Table 5.5: Existing IoT Inertial Motion Recognition Frameworks Versus Proposed Algorithm

Framework	**SRAM Usage**	**Online Learning**	**Segmentation**	**Personalized Output Classes**	**Application**	**Rep Count**	**No Manual Tuning**
Machine Learning-Based Approaches							
MiLift [SHS17]	$<$ 512 MB on Moto 360	SVM confidence scores	Hierarchical ML pipeline	No (static CRF, DT + HMM, and SVM)	Workout tracking	Revisit-Based	No
GesturePod [PDP19]	∼2 kB on Cortex-M0+	None	None	No (static ProtoNN)	Gesture recognition	None	No
Auritus [SSP22]	∼2 kB on Cortex MCU	None	None	No (static FastGRNN, Bonsai, ProtoNN)	Activity recognition	None	No
FastGRNN [KSB18]	∼2 kB on AVR RISC	None	None	No (static FastGRNN)	Activity recognition	None	No
TinyOL [RAR21b]	$<$ 256 kB on Cortex-M4	Transfer Learning	Running mean and variance	No (static autoencoder)	Anomaly detection	None	No
Coelho *et al.* DT [CSF21]	$<$ 128 kB on Cortex-M4	None	None	No (static DT)	Activity recognition	None	No
Active Learning [SHK17]	$<$ 512 MB on Gear Live	Active learning (querying)	None	No (static RF, DT, NB, SVM)	Activity recognition	None	No
Elsts *et al.* CNN [EM21]	$<$ 96 kB on Cortex-M3	None	None	No (static CNN)	Activity recognition	None	No
T'Jonck *et al.* CNN [TKV21]	$<$ 256 kB on Cortex-M4	None	None	No (static CNN)	Bed activity, anomaly	None	No
Finite State Machines (FSM) and Functional Programs							
FSM Sequence [KG15]	$<$ 3 GB on Core 2 Duo	Sequence matching	Smoothing	Yes (Fast learning FSM)	Gesture recognition	None	Yes
FnSM [WR15]	$<$ 512 MB on Sony Watch 3	None	None	No (static functional FSM)	Gesture recognition	None	No
Bosch GDL [WGD19]	$<$ 44 kB on integrated processor	GDL evaluation and regex	(Supported)	Yes (GDL + Non-deterministic FSM)	Gesture recognition	Yes	Yes
Template Matching							
Ours	<8 kB on ISPU	Template-based	Running mean and variance	Yes (template modification)	All of the above	Adaptive	Yes

ory bounds of an intelligent sensor processing unit (ISPU) from STMicroelectronics [RSZ22], consuming less than 8 kB of memory, achieving 2000× memory savings over competing techniques designed for microcontrollers. We qualitatively compare our framework against other on-device learning and motion inference frameworks (Section 5.3.2), discuss the automatic segmentation and learning process (Section 5.3.3), and the inference and personalization pipeline (Section 5.3.4).

5.3.2 On-device Learning and Motion Inference

Table 5.4 summarizes online learning techniques developed for low-end IoT platforms. Existing on-device learning paradigms (**how to learn**) for ML models can be classified into four classes. *Transfer learning* fine-tunes the last few layers of a frozen network graph [PY10]. Efficient transfer learning algorithms include reusing feedforward propagation representation maps for backpropagation [LN20], sample-wise stochastic gradient descent [RAR21b], and performing bias updates via lite residual learning [CGZ20]. Transfer learning suffers from catastrophic forgetting and limited capacity of unfrozen layers [CGZ20]. *Incremental training* performs sample-wise and gradient-free weight updates using constrained optimization [SYB21], suffering from limited supported model types. *Optimized backpropagation* algorithms reduce training memory usage through sparse updates of the most important network layers [LZC22], compile-time gradient calculation [LZC22], and combining the stability of gradient descent with the efficiency of stochastic gradient descent [Sud21]. On-device auto differentiation has a higher learning capacity than transfer learning, but limits online learning to either limited model types or processors with at least 256 kB of memory. *Continual learning* uses slow learning and a latent replay layer to store representation maps from previous training samples, preventing catastrophic forgetting [RRN21]. However, continual learning has high resource usage. To detect feature distribution shift in the streaming data and start the training process automatically (**when to learn**), the frameworks monitor the moments in the input data window [RAR21b], the model prediction confidence scores [SYB21], or the divergence in the covariate distribution of principal components in the input data window [LN20].

Deploying existing online learning frameworks in the wild raises two issues. *Firstly*, these techniques operate on a static ML classifier pre-trained on an application-specific dataset. Thus, the customer cannot add, replace, or remove additional output classes from the model, lacking personalization capabilities. Moreover, if the deployment-time task and data distribution are significantly different from the original dataset and task, then these techniques fail to adapt. In contrast, our technique assumes no prior information about the customer application or data distribution, allowing the user to create motion templates in the field with only 10 seconds of IMU data per class. Training data segmentation is performed automatically by observing the input stream moments. *Secondly*, existing online learning techniques are not suitable for execution on on-sensor processors, usually requiring around 256 kB of SRAM. These frameworks also support limited processor architectures depending upon the tinyML compiler suite and vector accelerators. In comparison, our method can run on any commodity microcontroller and even on-sensor processors supporting C instructions, requiring less than 8 kB of SRAM.

Table 5.5 showcases inertial motion recognition frameworks designed to be run in real-time on IoT platforms. These frameworks can be divided into three categories.

Machine Learning-Based Approaches: These frameworks run static ultra-lightweight ML classifiers to make inferences for a specific application. Specially-designed classifiers with low-rank, sparse, shallow, and quantized parameters (e.g., Bonsai, ProtoNN, and Fast-GRNN) [KSB18, GSG17, KGV17] allow motion recognition under 2 kB of SRAM [SSP22, PDP19, KSB18] at the cost of domain generalizability [SSP22]. Vanilla convolutional neural networks (CNN), decision trees (DT), and autoencoders can either be hand-tuned or optimized using neural architecture search for making inferences under 256 kB SRAM [EM21, TKV21, CSF21, RAR21b]. CNN and autoencoders also exploit post-training quantization and weight pruning to lower latency and memory usage even further by 6-9$\times$ and 9-14$\times$ without significant accuracy loss [HMD16, SSS22a]. These large sparse models are more robust to domain shifts than small dense models [SSP22]. IoT devices with more relaxed memory constraints (e.g., smartwatches) feature conventional ML classifiers such as conditional random fields (CRF), random forests (RF), support vector machines (SVM), naive Bayes (NB), and DT [SHS17, SHK17]. These devices feature online learning in the form of binary classification (in-class or not) through the use of SVM confidence scores or ensemble voting [SHS17, SHK17]. During online learning, observing the running moments of the input stream or using additional ML classifiers can be used to detect the start and end of specific motion primitives [RAR21b, SHS17]. Unfortunately, all ML-based approaches are task-specific, lack the ability to add, remove or modify output classes, and require user supervision (data processing) during training.

State Machines and Functional Programs: Finite state machines (FSM) are computational models that can take one of a finite number of states at a time and simulate sequential logic, regex pattern matching, and directed graphs [Ric08]. A functional program is composed of a sequence of subroutines (expression trees) [Hud89]. These approaches construct motion expressions from an ordered sequence of atoms, which are repetitions of symbols (predicates over raw data), described using a context-free grammar [WGD19]. Regex matching detects new motion primitives by computing similarity scores with sample sequences. The onboard FSM is updated to add the new motion primitive using the optimal event sequence [KG15, WGD19]. Smoothing removes noise and unwanted motion artifacts, keeping only the desired training samples during segmentation [KG15]. While these techniques are allow modification of classes, are task-agnostic, and support online learning, the accuracy of these frameworks is around 75% [KG15, WR15], while ML-based approaches easily exceed 95% [SHS17, SSP22, KSB18, CSF21, TKV21, PDP19, MSG14]. The memory usage is at least 40 kB, higher than the ISPU data memory.

Template Matching: Template matching converts time-series data into image templates and stores them onboard. Images are created on-the-fly from the IMU data stream and matched against stored templates using a similarity metric. This approach enjoys the accuracy of ML-based approaches while allowing fully online on-device learning and personalization of output classes similar to state machines and functional programs. Training data segmentation is performed by detecting ”static-motion” primitive via running mean and variance. The technique consumes less than 8 kB of SRAM on the ISPU. While the analysis of time-series data using image representation and template matching is well-explored [BCC20, HGD18, YK09, HL10, JLG17, Fu11], the end-to-end integration of inertial template matching with on-device learning, automatic segmentation, rep counting, and personalizability on on-sensor processors is unexplored.

5.3.3 Unsupervised Segmentation and Online Learning

The training process follows data segmentation, template creation, and storage.

Automatic Segmentation: When the user starts the training phase, accelerometer samples a_t are stored in a buffer a of size N. The user is asked to remain static for n seconds, monitored using the sum of the rolling mean ${}_R\bar{a}_t$ and the rolling variance ${}_R\sigma^2_{a_t}$ for all three accelerometer axes. ${}_R\sigma^2_{a_t}$ is as follows:

$$ {}_R\sigma^2_{a_t} = (1-\alpha) \cdot (\sigma^2_{a_t} + \alpha \cdot d_t^2) \tag{5.2} $$

where,

$$ d_t = \sqrt{a^2_{x,t} + a^2_{y,t} + a^2_{z,t}} -_R \bar{a}_t \tag{5.3} $$

$$ {}_R\bar{a}_t =_R \bar{a}_t + (\alpha \cdot d_t),\ {}_R\bar{a}_0 = \sqrt{a^2_{x,0} + a^2_{y,0} + a^2_{z,0}} \tag{5.4} $$

α is a tunable parameter controlling the smoothness of the running moments. Static condition is detected when the sum of rolling mean and variances is less than a pre-defined threshold β. α and β were automatically selected using Bayesian optimization [SAS21], with the objective function minimizing the difference between start and end times predicted by the segmentation algorithm versus human-picked points for a task-specific dataset. If a stationary state of n seconds is detected, then the data collection for the motion primitive to be detected starts, otherwise, the training phase is canceled. The user performs the desired motion primitive for p seconds (until the buffer is full). Running moments are used to discard the small number of samples that elapsed between the time the user was asked to perform the motion primitive and when the user actually started the motion primitive. If the number of static samples during this phase exceeds $z\%$ of the buffer length, the training is canceled. This "static-motion" based segmentation algorithm is computationally efficient over autocorrelation, time-warping, revisit-based, or ML-based segmentation approaches [EA12, SHS17].

Template Creation: First, the roll ϕ and pitch θ are calculated from the stored buffer a:

$$\begin{cases} \phi_t = \arctan 2(a_{y,t}, a_{z,t}) \\ \theta_t = \arcsin \frac{a_{x,t}}{\sqrt{a_{x,t}^2 + a_{y,t}^2 + a_{z,t}^2}} \end{cases} \tag{5.5}$$

Next, the gravity vector swing g in each axis is calculated:

$$\begin{cases} g_{x,t} = \sin \theta_t \\ g_{y,t} = \cos \theta_t \sin \phi_t \\ g_{z,t} = \cos \theta_t \cos \phi_t \end{cases} \tag{5.6}$$

The variance of g is calculated for each axis:

$$\sigma_g^2 = \frac{1}{N} \sum_t (g_t - \bar{g})^2 \tag{5.7}$$

The two axes with the highest variance, g_a and g_b, are noted to retain maximal information (principal components). Afterward, an $m \times m$ byte grid is created. The numerical values in g_a and g_b are quantized into buckets as follows:

$$\begin{aligned} g_{u,t}^{\text{quantized}} &= 0.5\Big(\Delta \cdot i + \Delta \cdot (i-1)\Big) \\ \Leftrightarrow \Big(g_{u,t} &< \Delta \cdot i\Big) \wedge \Big(g_{u,t} > \Delta \cdot i - 1\Big) \end{aligned} \tag{5.8}$$

where, $i \in [1, m]$, $u = \{a, b\}$. Each element in the grid takes a value of $g_{u,t}^{\text{quantized}}$. Δ is the quantization resolution, which can be calculated from either of the two axes a or b:

$$\Delta = \frac{\max(g_v) - \min(g_v)}{m - 1}, \; v = \{a, b\} \tag{5.9}$$

The quantized values are used to fill up the elements of the byte grid, with the maximum value of a grid element equal to 255. The result is an $m \times m$ quantized image of g_a and g_b plotted against each other. Fig. 5.2 shows 20×20 templates created for three different exercise activities in real-time, namely bicep curl, lateral, and jack.

The motivation to convert accelerometer time series to 2D image templates are twofold. *Firstly,* the gravity vector creates a spatial view of the motion primitive by overlapping

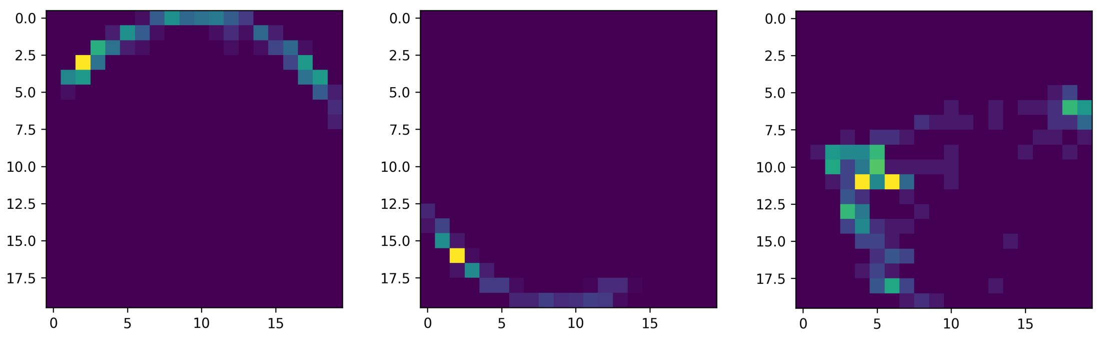


Figure 5.2: Example templates generated by our algorithm for (Left) bicep curl (Center) lateral (Right) jack exercises.

the accelerometer samples with the same gravity vector swing. It gets rid of the computational expense and algorithmic challenge to time align and then overlap consecutive motion primitive sequences for a certain motion primitive. *Secondly*, 2D images provide more information (variance) corresponding to real-world applications than 1D principal components. Most real-world motion sensing applications (e.g., fitness monitoring, activity recognition, and gesture detection) usually span two gravity vector axes.

Template Storage: The image and the name of the two axes with maximum variance are stored as an element in an array of structures in the data memory of the ISPU. Each element in the array of structures has 3 members: the quantized $m \times m$ byte template, a character specifying the axis with the highest variance, and a character specifying the axis with the second highest variance. Additionally, a counter maintains how many of the elements in the data structure have been used and is incremented by 1 when a new template is added.

5.3.4 Real-time Classification and Personalization

During inference, the buffer size is M, such that $M < N$. To promote variable reuse, we use the same buffer used during training. When the buffer has M elements, templates are created using the formulation described in Section 5.3.3. For each element in the array of structures, the two stored maximal axis variance names are matched with the two maximal axis variance names for the inference-time gravity vector swing. If a match is found, then the universal image quality index (UQI) [WB02] Q is calculated between the selected stored candidate image template e and the inference image f:

$$Q = \frac{4 \cdot \sigma_{ef} \cdot \bar{e} \cdot \bar{f}}{(\sigma_e^2 + \sigma_f^2) \cdot (\bar{e}^2 + \bar{f}^2)} \tag{5.10}$$

where

$$\bar{e} = \frac{1}{m^2} \sum_{i=1}^{m^2} e_i, \; \bar{f} = \frac{1}{m^2} \sum_{i=1}^{m^2} f_i \tag{5.11}$$

$$\sigma_e^2 = \frac{1}{m^2 - 1} \sum_{i=1}^{m^2} (e_i - \bar{e}), \; \sigma_f^2 = \frac{1}{m^2 - 1} \sum_{i=1}^{m^2} (f_i - \bar{f}) \tag{5.12}$$

$$\sigma_{e,f} = \frac{1}{m^2 - 1} \sum_{i=1}^{m^2} (e_i - \bar{e}) \cdot (f_i - \bar{f}) \tag{5.13}$$

Q varies between -1 and +1, with +1 signifying the two templates are identical, while -1 signifies the highest dissimilarity between the two templates. We perform the axis match first to save valuable computation time needed to calculate the UQI by eliminating stored image candidates whose axis does not align with the inference template. To handle domain shifts caused by noise, sensor placement offset, and inertial disturbances, we downsample e and f by applying uniform filtering:

$$w_{\text{blurred}} = w \, * \, J_k, \; w = \{e, f\} \tag{5.14}$$

$*$ is the convolution operator, performed only in the valid regions. J is the square unit matrix of length k, also known as the blur kernel. The idea is inspired by how a CNN successively downsamples an input image layer-by-layer by applying a convolution kernel to extract an abstract yet generalizable representation map [GBC16].

For personalization, the onboard software provides the customer the option to replace or erase an existing template. The user can choose to replace an existing template with a new one, append new templates at the end of the array of structures (provided the maximum

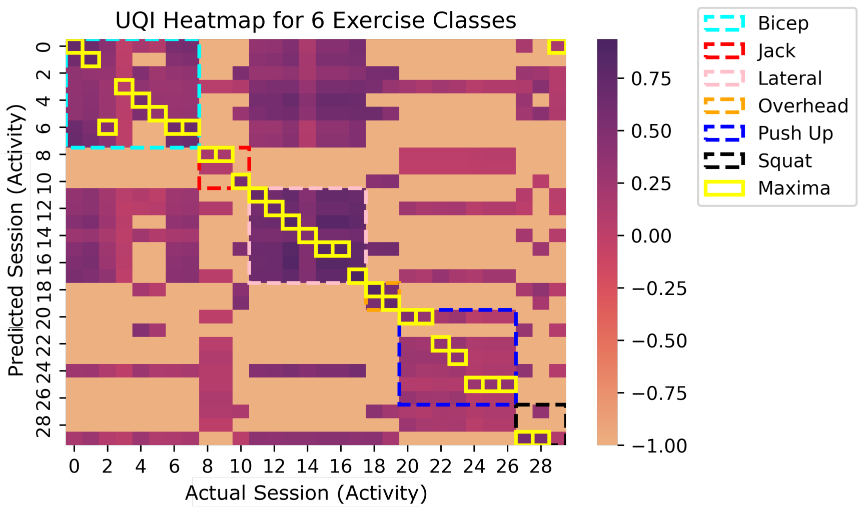


Figure 5.3: UQI heatmap and confusion matrix for workout tracking

number of allowable templates has not been reached), or erase the contents of the whole array of structures. The training process described in Section 5.3.3 is followed during this phase.

5.3.5 Evaluation

We evaluate the algorithm for workout activity recognition (Section 5.3.5.1), quantifying resource usage for commodity microcontrollers and the ISPU (Section 5.3.5.2). We use an internally collected IMU dataset containing 30 workout sessions from multiple volunteers spanning 6 classes, namely bicep curl, jack, lateral, overhead, push-up, and squat exercises. The participants wore the IMU on their wrist via a ST SensorTile.Box wristband in any orientation they liked. The data was stored on an onboard SD card. The test dataset was created by partitioning this dataset. The sensor core ODR was 26 Hz. n was 3, α was 0.01, z was 50, m was 20, M was 52 (2 seconds), N was 260 (10 seconds), and k was 8. The array of structures was limited to 6 templates.

5.3.5.1 Workout Tracking Accuracy

Fig. 5.3 shows a heatmap showing detected UQI values and the confusion matrix for workout tracking for the 30 sessions across the 6 classes. Note that the goal at inference time is to match the workout label among the 6 classes, and not necessarily the instance or session. For example, if the algorithm detects an activity as 'squat' but the match stems from a 'squat' activity from a different workout session, then the inference is still valid. Our algorithm achieves a 96.7% workout tracking accuracy, only misclassifying an instance of squat with bicep curls. In other cases, the predicted class lies either on the diagonal (matched instance) or the in-class bounds (matched workout label). For the sessions that correspond to the same workout class (e.g., 0-7 corresponds to bicep curl and 8-10 corresponds to jack) the highest UQI is always within the in-class bounds, signifying robustness to in-class domain shifts of our algorithm. Without the uniform filter, our algorithm still achieves 90% + accuracy. Other similarity metrics such as the mean squared error, structural similarity index [Wan04], spatial correlation coefficient [ZCS98], peak signal-to-noise ratio [Wan04], and spectral distortion index [AAB08] achieve 30-95% accuracy.

To ensure that the whole algorithm generalizes in real-world settings, we also performed a real-world study where the algorithm was ported to an ISPU within the ST SensorTile.Box. We asked two volunteers not present in the original dataset to test the training and inference pipelines using their own choice of wristband orientation and activities. The algorithm successfully segmented the region of interest during training and accurately detected the user-defined activities in real time.

5.3.5.2 Resource Usage

Table 5.6 outlines the resource usage and latency of various components of our algorithm on a general-purpose Cortex-M4 microcontroller and the ISPU. Template storage during training is the slowest. However, template storage is not related to inference and only happens

Table 5.6: Latency and Resource Usage of Our Algorithm

Algorithm Component	**Latency (mS)**	
	Cortex-M4 (84 MHz)	**ISPU (10 MHz)**
Template creation	4-6	32-48
Template storage	65	130
Template retrieval	0.19	1.55
Template matching	0.27	2.11

Memory Component	**Memory Usage (kB)**	
	Cortex-M4 (84 MHz)	**ISPU (10 MHz)**
Data	17.7	7.2
Program	45.0	24.0

during training. Template retrieval, matching, and rep counting have negligible latency on both devices. Template creation takes 40 mS, which is roughly within the ISPU processing cycle for 26 Hz ODR. The algorithm consumes less than 8 kB memory on the ISPU, which is 2.5× lower than the microcontroller implementation, and 2000× lower than existing online learning frameworks thanks to the optimized instruction set.

5.4 Foundation Models

Autoregressive large language models (LLMs) [DCL19, BMR20, CND22, MDL23] have revolutionized natural language processing (NLP) by showing unprecedented and high-bandwidth performance across a broad spectrum of tasks through *few-shot learning* [BMR20], *memorization* [TMZ22], and *compositionality* [ZSH22]. LLMs are neural networks that are trained using large quantities of unlabeled text through self-supervised learning and have billions of parameters [CND22]. LLMs generalize across a variety of tasks thanks to their parameter count and input dataset size and can *hallucinate* and exhibit *emergent abilities*.

Architecture: LLMs embrace the transformer [VSP17] architecture, which has the ability to process sequential input data, differentially weigh the importance and extract the context of parts of the input data, and parallelize computations across layers and sequence positions. Unlike RNNs, transformers do not suffer from the exploding and vanishing gradient problem or lack of parallelization, allowing more direct information flow and stable training [ZBI19]. Modern LLMs such as GPT-3 [BMR20], GPT-4 [Ope23], LaMBDA [TDH22], PaLM [CND22], and Megatron–Turing NLG [SPN22] use a *left-to-right*, *decoder-only* architecture. Compared to bidirectional transformers (BERT) [DCL19], modern LLM architectures can handle longer sequences due to their autoregressive nature, do not use *masked language modeling* (generalizes to downstream tasks without fine-tuning), and are more efficient at runtime.

Pre-training: LLMs are pre-trained using unsupervised learning on a large unlabeled text corpus to make the transformer learn the underlying semantics, relationship, and grammar [CND22]. The text is first tokenized (split into words or subwords). The LLM is then trained either using *masked language modeling* (MLM) or *autoregressive language modeling* (ALM). In MLM, the objective is to predict masked tokens based on the surrounding (past and future) context [DCL19]. This prevents memorization, forces the model to learn generalizable representations, and prevents the attention mechanism from seeing future information. In ALM, the objective is to predict the next token in the sequence given only the past tokens [CND22]. ALM is suitable for tasks that are generative, need longer tokens, and whose contexts are unidirectional in nature. However, MLM is less biased toward common tokens, can handle missing data, and is more flexible.

Adapting for downstream tasks: There are several techniques to apply a pre-trained LLM to a downstream task:

- **Fine-tuning**: The weights of the LLM are frozen to preserve learned knowledge from pre-training. New layers are introduced at the output of the LLM, whose weights are learned using supervised learning on the downstream dataset. This is the same as *transfer learning* [GDC21].
- **Prompting**: Few-shot prompting formulates the problem to be solved via a text prompt with a few or no (problem, solution) pairs [WWS22].
- **Tuning from feedback**: This is the continuous form of prompting. Reinforcement

learning from human feedback performs supervised fine-tuning on a human-generated (problem, solution) dataset, but a reward is provided to the LLM that was learned from the dataset [CLB17, OWJ22]. Self-instruct fine-tunes the model on a dataset generated by the LLM itself warm-started from a smaller human-generated (problem, solution) dataset [WKM22].

- **Knowledge distillation**: A smaller model is trained to mimic the logits, intermediate representations, or instance relations of the LLM using distillation loss. The smaller model is more suited for edge deployment [GYM21].

- **Multi-task learning**: Sequence-to-sequence learning combined with prompting during pre-training and downstream adaptation can yield multi-task LLM without requiring extra task-specific layers [WYM22].

Multimodal Foundation Models: Multimodal foundation models can deal with multiple modalities (e.g., text, image, audio, and video) simultaneously [GLL22, XYY23, KNC23, RKH21, DXS23, HWC22, MDL23, WYM22]. The embeddings can be generated by modality-specific frozen foundation models (e.g., BERT [DCL19] for text, and ViT for images [HWC22]), which are concatenated using late fusion via self-attention and cross-modal attention modules [XYY23, RKH21]. The self-attention module outputs embeddings that are aware of other embeddings (e.g., text-aware image embeddings). The fusion layers are fine-tuned via a linear combination of modality-specific loss functions. Another approach trains a single encoder-decoder architecture end-to-end to accept a vector containing the multimodal tokens for supposed joint understanding [KNC23, GLL22, DXS23]. This is called early fusion.

5.4.1 Large Language-Inertial Models (LLIM)

While pre-trained foundation models for text, audio, and vision modalities are plentiful [ZLL23], very little attention has been put forward developing pre-trained foundation models for inertial sensors [XZT21]. Given the broad domain spectrum of inertial sensors [CDK19], we

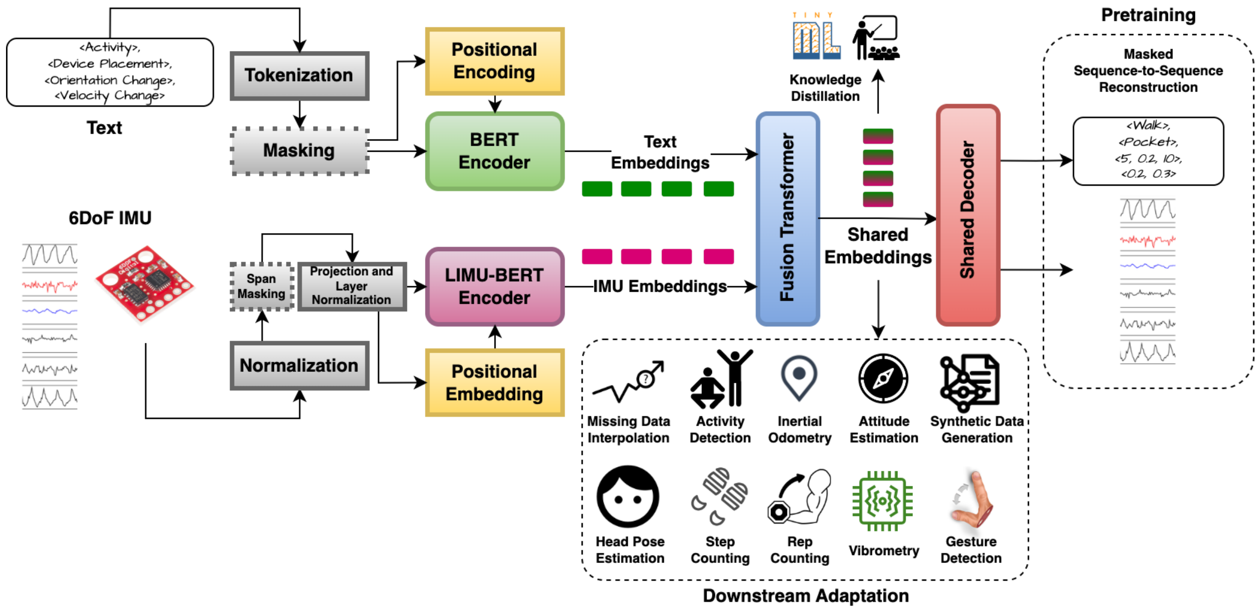


Figure 5.4: Overall architecture of LLIM showing pre-training and downstream adaptation.

present LLIM, a family of context-aware inertial foundation models, shown in Fig. 5.4. LLIM consists of two encoders. *Firstly*, BERT generates text embeddings for the context of the current inertial sensor window, such as activity primitive, sensor placement, change in orientation, and change in velocity. *Secondly*, a modified LIMU-BERT [XZT21] encoder generates embeddings for IMU data. LIMU-BERT is the only proposed foundation model architecture for IMU data in literature [XZT21]. The two embeddings are fused via attention mechanisms using a fusion transformer, which generates shared embeddings. We pre-train the modified LIMU-BERT architecture on large quantities of unlabeled IMU data using masked sequence modeling. Afterward, the BERT and the LIMU-BERT encoder are frozen. The fusion transformer and the shared decoder are then pre-trained on large quantities of existing labeled and unlabeled data using masked sequence-to-sequence reconstruction. The generated embeddings can be used for almost all of the downstream tasks discussed in this dissertation. Knowledge distillation can be used to hot-start and efficiently train gener-

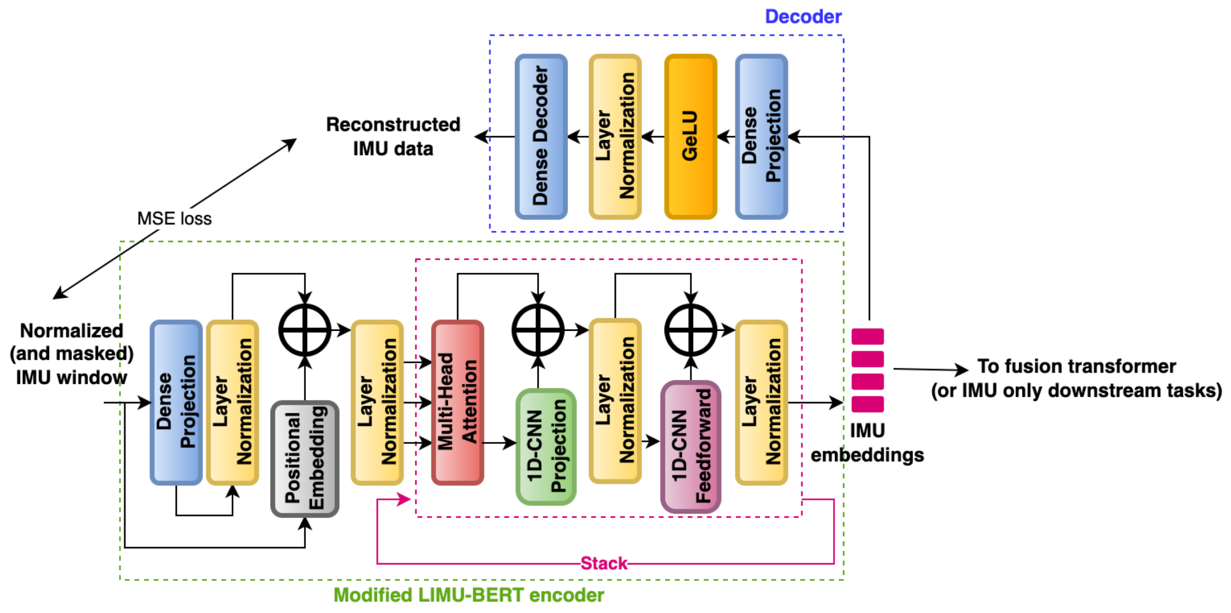


Figure 5.5: Architecture of modified LIMU-BERT.

alizable tinyML models suitable for edge deployment. We share the ongoing work in the upcoming subsections (Section 5.4.2 to Section 5.4.5).

5.4.2 Modified LIMU-BERT

The core component in LLIM is the IMU encoder, for which we start with the LIMU-BERT architecture [XZT21]. Fig. 5.5 shows the modified LIMU-BERT architecture. The sensor data sequence is first normalized. During pre-training, the span masking algorithm [JCL20] is used to probabilistically mask continuous regions of the input sequence. The masked sequence is projected to a high-dimensional space using a linear layer. The extracted implicit features are normalized via layer normalization [BKH16]. The normalized features are added with the output of the positional embedding function, and further normalized. The positional encoding adds recurrence (sense of time) to the transformer architecture [VSP17]. The output is then fed to the attention-centric block, which can be stacked one after the other with an increasing parameter count. There are three residual components with cross-layer parameter sharing in this block, namely the multi-attention block containing the self-attention layers, the projection block, and the feedforward block. The projection block contains a single neural network layer, while the feedforward block contains two neural network layers with Gaussian error linear unit (GeLU) activation. GeLU provides a smoother and more continuous shape than ReLU, making it suitable for learning complex patterns [HG16]. The output of the attention-centric block represents the generalized IMU embeddings, which can be fed to a decoder during LIMU-BERT pertaining, to the fusion transformer for pretraining LLIM, or to IMU-only downstream tasks by adding task-specific layers. Our contribution to the LIMU-BERT architecture is two-fold:

- We design the projection and feedforward blocks to be temporal convolutional. Convolutional layers can extract fine-grained local patterns from the IMU input sequence, allowing the transformer to extract both local and global dependencies [GQC20, WXC21].
- We allow stacking the attention-centric block several times to create a family of LIMU-BERT architectures with different parameter counts, allowing for transformers that are small and capable of running on edge devices, as well as large ones that have lower reconstruction loss.

Additionally, we pre-train the modified LIMU-BERT on a much larger dataset (discussed in Section 5.4.4. The hyperparameters to optimize include:

- Number of attention-centric blocks.
- Number of heads in the multi-head attention module.
- Number of hidden units in the first dense projection layer.
- Number of filters (or the number of hidden units) in the feedforward layer.

- Kernel size of the filters in the projection and feedforward layers (only if convolution is used)

In addition, the size of the input window is also a parameter to be chosen.

5.4.3 Fusion Transformer

The fusion transformer extracts cross-model semantics from the text and IMU embeddings in a joint embedding space. The fusion transformer contains a stack of transformer blocks with cross-attention, similar to ALBEF [LSG21], to yield multimodal text-inertial representations. The outputs of the transformer stack are fed to self-attention to extract dependencies within the shared representation. In the future, we would like to prepend a universal layers module [XYY23] to generate text-aware inertial and inertial-aware text embeddings for improved modality collaboration and modality disentanglement.

5.4.4 Pretraining

We use a pre-trained BERT encoder from Huggingface [1] for extracting text embeddings from the text. We pre-train the LIMU-BERT and the fusion transformer from scratch.

5.4.4.1 Pretraining LIMU-BERT

The original LIMU-BERT was trained on only small human activity recognition datasets individually containing 2000-11000 samples [XZT21]. In contrast, for our initial evaluation, we pre-train the modified LIMU-BERT on a joint human activity recognition dataset containing the Sussex-Huawei locomotion and transportation (SHL) dataset [GCW18], realistic sensor displacement (RealDisp) dataset [BDP12], RealWorld [SS16], and the PAMAP2 physical activity monitoring dataset [RS12]. The combined dataset has 2777 hours of IMU data

[1] https://huggingface.co/

from 44 volunteers performing 67 activity primitives across 21 different device placements. We down/upsampled the data at 50 Hz and created 2 million files containing 5 seconds (sequence length: 250) of IMU data. We divided the dataset into training (80%), validation (10%), and test splits (10%). We use the mean-squared error as the reconstruction loss for initial evaluation, training the decoder to interpolate the masked values and comparing them against the ground truth values at those positions. We plan to expand the dataset and experiment with other loss functions in the future.

5.4.4.2 Pretraining the Fusion Transformer

For an initial evaluation, we pre-train the fusion transformer with a shared decoder that reconstructs the IMU sequence and the activity label. The loss function should be a sum of the reconstruction loss for the IMU and the text. There are several loss functions we are experimenting with:

- Contrastive loss [WL21] for learning IMU-text pairs.
- Cross-entropy loss for IMU-captioning.
- Mean-squared error loss for IMU reconstruction [XZT21].
- Cross-entropy loss for text reconstruction [DCL19].

5.4.5 Downstream Adaptation

Currently, LLIM supports downstream adaptation either through fine-tuning or knowledge distillation. So far, we have evaluated downstream adaptation for IMU-only activity recognition tasks using our modified LIMU-BERT architecture. In contrast with the original LIMU-BERT, which pre-trains and tests downstream tasks primarily on the same dataset, we pre-train on the joint dataset described above and perform downstream adaptation via fine-tuning on an entirely different dataset.

Table 5.7: Human Activity Recognition Performance on Downstream Dataset (1% labeling rate) for modified LIMU-BERT vs. vanilla LIMU-BERT.

Method	**Pre-Training Dataset**	**Downstream Dataset**	**Test Accuracy**
LIMU-BERT	Shoaib [SBI14]	Shoaib	86.5
	UCI [ROS16]		80.0
	MotionSense [MCC19]		81.4
	HHAR [SBB15]		81.3
Modified LIMU-BERT (ours)	SHL, RealDisp, RealWorld, PAMAP2	Shoaib	**84.0**
LIMU-BERT	Shoaib [SBI14]	HHAR	56.2
	UCI [ROS16]		51.5
	MotionSense [MCC19]		56.4
	HHAR [SBB15]		66.9
Modified LIMU-BERT (ours)	SHL, RealDisp, RealWorld, PAMAP2	HHAR	**60.3**

Table 5.7 showcases the performance of modified LIMU-BERT and vanilla LIMU-BERT for downstream adaptation on datasets not present in the training set. Both models have the same number of parameters, but our model is trained on a larger dataset and has the modified architecture. In terms of generalization on a new dataset, our method outperforms the vanilla LIMU-BERT by 5%, showing the importance of pre-training foundation models on large quantities of unlabeled data. In the future, we would like to evaluate the modified LIMU-BERT for more downstream tasks, and ultimately move toward evaluating the overall LLIM architecture.

5.5 Discussion

In this chapter, we discussed three techniques to adapt TinyML programs to a new domain and application in the wild, namely *fine-tuning*, *on-device learning* (no neural component), and *foundation models*.

Firstly, we presented the utility of transfer learning in adapting TinyML programs in a new domain using a few minutes of labeled data. Transfer learning drastically reduces the data inefficiency of TinyML programs, allowing the usage of user-friendly video pipelines to collect labeled data and deployment of pre-trained models in a new domain. The approach, however, still requires labeled data. Moreover, transfer learning is limited in adapting to data distributions that are extremely divergent, and is not application agnostic.

Next, we described an on-sensor inertial on-device learning and classification framework that consumes less than 8 kB memory. The framework is application agnostic, allows personalization of output classes, supports online learning, and performs automatic segmentation all without any user supervision. The framework needs no labeled data. Future directions include quantifying in-field performance for other applications, automated hyperparameter tuning, and optimizing complex mathematical operations.

Lastly, we briefly discussed the value of foundation models in generalizing across a broad spectrum of tasks. We presented LLIM, a family of context-aware inertial models that excels across various downstream tasks and downstream datasets, pre-trained purely using unlabeled data. Future directions include experimenting with various loss functions, pre-training with more datasets, formalizing the fusion transformer architecture, evaluating LLIM on more downstream tasks, and evaluating knowledge distillation for generating domain-invariant TinyML models.

CHAPTER 6

Conclusion and Future Work

Tiny machine learning (TinyML) has opened a new opportunity to bring intelligence to Internet-of-Things platforms and embedded systems ubiquitous around us. In this dissertation, we explored techniques to inject uncertainty awareness, platform awareness, context awareness, physics awareness, and domain awareness into the existing TinyML workflow.

Firstly, starting with data collection, real-world sensors are imperfect, suffering from spatial and temporal misalignments. Training time uncertainty augmentation can make TinyML models robust to deployment time uncertainties.

Secondly, the optimization of TinyML backbones needs to take into account the deployment time and execution-level dynamics of the platform the TinyML program is going to run on. A TinyML program may contain both neural and non-neural components, all of which must be jointly handled and optimized for guaranteed deployment. Platform-aware, black-box, Bayesian, and gradient-free neural architecture search can talk to the target platform during the optimization process to examine runtime faults, while also being able to efficiently handle mixed hyperparameter search spaces.

Thirdly, the TinyML program must obey certain bounds, rules, heuristics, and physics of the underlying domain for safe, reliable, and provably correct operation. Neurosymbolic tiny machine learning enables combining TinyML models with physics-based models within the ultra-resource constraints of TinyML platforms. The optimal synergy of neural and symbolic components opens up a broad spectrum of edge artificial intelligence applications such as inertial navigation, object detection and tracking, on-chip activity recognition, and neural-Kalman filtering.

Fourthly, TinyML programs must be personalizable across dataset distributions and applications during deployment time. Transfer learning, on-device learning, and foundation models pave the way for utilizing pre-trained models across a broad task spectrum.

There are several avenues of future work stemming from this dissertation.

6.0.1 Guiding the Training Towards Heuristic Goals

As models are shrunk, they become fragile in terms of generalizability, robustness, and backward compatibility. The *neuro* $\cup$ *compile [symbolic]* paradigm does not guarantee strict enforcement of user constraints, which are lost within the neural network embeddings. The *symbolic [neuro]* paradigm allows the fusion of neural and physics-based components through a Kalman filter, but the neural network is agnostic to the heuristic rules, physics, and bounds being managed by the Kalman filter. The preferred paradigm is *neuro [symbolic]*, where the neural architecture is embedded with special symbolic reasoning layers. The symbolic layer is biased to strictly enforce user-defined models and mimic a logical reasoning module [Kau22, SZE21]. Since the loss flows through the whole neural network, the neural network becomes aware of the intricacies of the symbolic layer. In other words, physics awareness, robustness and backward compatibility ingrained in the network weights may enable more trustworthy neurosymbolic systems. More work is required to implement such physics-aware reasoning layers [GDY19, CGH20, YPJ19, LCS21, BGS22] for deployment onto TinyML platforms.

6.0.2 Backward Compatibility

The changes in behavior when deploying an upstream model (e.g., a model on the cloud) to microcontrollers through the TinyML workflow cannot be measured in isolation using only the aggregate performance measures (such as accuracy) [SNK20] Even when a TinyML model (downstream model) and the upstream model have the same accuracy, they may not be functionally equivalent and may have sample-wise inconsistencies [YXK21] resulting in new failures impacting high stake domains such as healthcare. This notion of functional equivalence between an upstream and a downstream model is known as backward compatibility. When previously unseen errors are observed in the downstream model, the downstream model is said to be backward incompatible [BNK19] and has low fidelity [JCB20] and high perceived regression [YXK21] with respect to the upstream model. As a result, to have robust inference, the TinyML model must have both high accuracy and high fidelity with its upstream counterpart. Proposed solutions, such as positive congruent training [YXK21] and backward compatible learning [SXX20], are yet to be integrated and optimized for the TinyML workflow.

6.0.3 Long Term Planning and Reasoning

TinyML models mostly focus on short windows. However, dealing with long sequences is important, especially for complex event processing [SSS20]. While neurosymbolic AI combines the short-term perception power of neural networks with long-term context from symbolic models, we lose out on any performance gains possible by long-term context extracted from the data by a neural network. Thus, we need better spatial and temporal primitives for long-term reasoning at the neural level to counteract performance loss through purely symbolic reasoning methods. This is important for applications such as motion planning, simultaneous localization and mapping, and joint state control in the presence of noisy sensors and actuators [NPD22].

6.0.4 TinyML as Part of Bigger Systems

So far, this dissertation has focused on TinyML as standalone independent systems. However, TinyML systems can be part of bigger systems, each containing multiple individual TinyML components. The optimal way to distribute a system's computation paradigm across these independent TinyML platforms under dynamic and ever-changing resource constraints is an open research challenge [TIR20], especially in control problems.

Moreover, even with the advent of edge artificial intelligence, one cannot discount the value of the cloud, especially given the meteoric rise of foundation models [YZY22]. The TinyML program may occasionally tap a more accurate and compute-intensive foundation model on the cloud [RZH19]. Finding the optimal time to tap the cloud such that the communication latency and energy incurred in cloud inference are outweighed by performance gains under communication outages and a stochastic scheduling environment is challenging.

6.0.5 Security and Privacy in TinyML

While constraining private data within the TinyML platform node reduces the chance of privacy and security leaks associated with cloud-based inference, the attack surface on TinyML platforms is wide open. Compressed models are prone to adversarial attacks and false data injection with a higher success rate than larger models [LGH18][GWY19][YXL19]. At the sensing layer, microarchitectural and physical side channels can leak information from microcontroller chips through cache leaks, power analysis, and electromagnetic analysis [HCS19]. Direct attacks on TinyML devices include malware injection, model extraction, access control, man-in-the-middle, flooding, and routing [HCS19]. Therefore, the neural architecture search optimization function in the TinyML workflow should include adversarial robustness goals to provide not only the smallest models but also the models most robust to adversarial attacks [GWY19][YXL19][GYX20]. The workflow should also include attack surface analysis and tools to defend the inference pipeline against attacks.

6.0.6 Hardware/Software Co-Exploration

Much of the development in TinyML has been software-driven, with the hardware platform being static. While TinyML platforms hosting microcontrollers are shrinking due to Moore's Law, the workload and the complexity of neural networks have skyrocketed [STR21][XDH18]. Proposed hardware innovations include the use of a systolic array, stochastic computing, in-memory computing, near-data processing, spiking neural hardware, and non-von Neumann architectures [STR21][XDH18][LLW20]. However, such architectural innovations are largely disjoint from the TinyML software communities. Developments in TinyML software need to be performed hand-in-hand with attention-directed hardware design, with the platform and model being optimized jointly [JZS19][JYS20].

REFERENCES

[AAB08] Luciano Alparone, Bruno Aiazzi, Stefano Baronti, Andrea Garzelli, Filippo Nencini, and Massimo Selva. "Multispectral and panchromatic data fusion assessment without reference." *Photogrammetric eng. & Remote Sensing*, **74**(2), 2008.

[ABC16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "{TensorFlow}: a system for {Large-Scale} machine learning." In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[ACS17] Moustafa Alzantot, Supriyo Chakraborty, and Mani Srivastava. "Sensegen: A deep learning architecture for synthetic sensor data generation." In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 188–193. IEEE, 2017.

[AFS21] Saad Abbasi, Mahmoud Famouri, Mohammad Javad Shafiee, and Alexander Wong. "OutlierNets: highly compact deep autoencoder network architectures for on-device acoustic anomaly detection." *Sensors*, **21**(14):4805, 2021.

[AGO13] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra Perez, and Jorge Luis Reyes Ortiz. "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*, pp. 437–442, 2013.

[AI22] Norah N Alajlan and Dina M Ibrahim. "TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications." *Micromachines*, **13**(6):851, 2022.

[AKM17] Mona Alshahrani, Mohammad Asif Khan, Omar Maddouri, Akira R Kinjo, Núria Queralt-Rosinach, and Robert Hoehndorf. "Neuro-symbolic representation learning on biological knowledge graphs." *Bioinformatics*, **33**(17):2723–2730, 2017.

[ALT21] Sayeda Shamma Alia, Paula Lago, Shingo Takeda, Kohei Adachi, Brahim Benaissa, Md Atiqur Rahman Ahad, and Sozo Inoue. "Summary of the cooking activity recognition challenge." In *Human Activity Recognition Challenge*, pp. 1–13. Springer, 2021.

[ALT22] Kareem Ahmed, Tao Li, Thy Ton, Quan Guo, Kai-Wei Chang, Parisa Kordjamshidi, Vivek Srikumar, Guy Van den Broeck, and Sameer Singh. "PYLON: A PyTorch framework for learning with constraints." In *NeurIPS 2021 Competitions and Demonstrations Track*, pp. 319–324. PMLR, 2022.

[AMD15] Ferhat Attal, Samer Mohammed, Mariam Dedabrishvili, Faicel Chamroukhi, Latifa Oukhellou, and Yacine Amirat. "Physical human activity recognition using wearable sensors." *Sensors*, **15**(12):31314–31338, 2015.

[APS21] Gianluca Apriceno, Andrea Passerini, and Luciano Serafini. "A Neuro-Symbolic Approach to Structured Event Recognition." In *28th International Symposium on Temporal Representation and Reasoning (TIME 2021)*, 2021.

[aut16a] The GPyOpt authors. "GPyOpt: A Bayesian Optimization framework in python." `http://github.com/SheffieldML/GPyOpt`, 2016.

[aut16b] The Skopt authors. "Skopt: scikit-optimize." `https://scikit-optimize.github.io/`, 2016.

[BBC11] M Lourdes Borrajo, Bruno Baruque, Emilio Corchado, Javier Bajo, and Juan M Corchado. "Hybrid neural intelligent system to predict business failure in small-to-medium-size enterprises." *International journal of neural systems*, **21**(04):277–296, 2011.

[BBC17] Martin Brossard, Silvere Bonnabel, and Jean-Philippe Condomines. "Unscented Kalman filtering on Lie groups." In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2485–2491. IEEE, 2017.

[BBL19] Valentina Bianchi, Marco Bassoli, Gianfranco Lombardo, Paolo Fornacciari, Monica Mordonini, and Ilaria De Munari. "IoT wearable sensor and deep learning: An integrated approach for personalized human activity recognition in a smart home environment." *IEEE Internet of Things Journal*, **6**(5), 2019.

[BCC20] Silvio Barra, Salvatore Mario Carta, Andrea Corriga, Alessandro Sebastian Podda, and Diego Reforgiato Recupero. "Deep learning and time series-to-image encoding for financial forecasting." *IEEE/CAA Journal of Automatica Sinica*, **7**(3), 2020.

[BDP12] Oresti Baños, Miguel Damas, Héctor Pomares, Ignacio Rojas, Máté Attila Tóth, and Oliver Amft. "A benchmark dataset to evaluate sensor displacement in activity recognition." In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pp. 1026–1035, 2012.

[Ber16] Dimitri Bertsekas. *Nonlinear Programming*, volume 4. Athena Scientific, 2016.

[BG98] Suresh Balakrishnama and Aravind Ganapathiraju. "Linear discriminant analysis-a brief tutorial." *Institute for Signal and Information Processing*, **18**(1998):1–8, 1998.

[BGN17] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. "Designing neural network architectures using reinforcement learning." *International Conference on Learning Representations (ICLR)*, 2017.

[BGS22] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. "Logic tensor networks." *Artificial Intelligence*, **303**:103649, 2022.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization." *arXiv preprint arXiv:1607.06450*, 2016.

[BKS18] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. "Neuro-symbolic program corrector for introductory programming assignments." In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 60–70. IEEE, 2018.

[BMR20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language models are few-shot learners." *Advances in neural information processing systems*, **33**:1877–1901, 2020.

[BNG16] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. "The EuRoC micro aerial vehicle datasets." *The International Journal of Robotics Research*, **35**(10):1157–1163, 2016.

[BNK19] Gagan Bansal, Besmira Nushi, Ece Kamar, Daniel S Weld, Walter S Lasecki, and Eric Horvitz. "Updates in human-ai teams: Understanding and addressing the performance/compatibility tradeoff." In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 2429–2437, 2019.

[BRS19] Antoine Bosselut, Hannah Rashkin, Maarten Sap, Chaitanya Malaviya, Asli Celikyilmaz, and Yejin Choi. "COMET: Commonsense Transformers for Automatic Knowledge Graph Construction." In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4762–4779, 2019.

[BRT21] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. "MLPerf Tiny Benchmark." *Advances in Neural Information Processing Systems*, 2021.

[BZF21] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers." *Proceedings of Machine Learning and Systems*, **3**, 2021.

[CA98] JM Corchado and J Aiken. "Neuro-symbolic reasoning for real time oceanographic problems." In *Conference On Data Mining. IEE, Savoy Place, London*, 1998.

[CAS21] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Edoardo Debenedetti, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. "RobustBench: a standardized adversarial robustness benchmark." In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[CBP04] Juan M Corchado, M Lourdes Borrajo, María A Pellicer, and J Carlos Yáñez. "Neuro-symbolic system for business internal control." In *Industrial conference on data mining*, pp. 1–10. Springer, 2004.

[CD18] Mahesh Chowdhary and Sankalp Dayal. "Reconfigurable sensor unit for electronic device.", November 27 2018. US Patent 10,142,789.

[CDK19] Jussi Collin, Pavel Davidson, Martti Kirkko-Jaakkola, and Helena Leppäkoski. "Inertial sensors and their applications." *Handbook of Signal Processing Systems*, pp. 51–85, 2019.

[CGH20] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. "Lagrangian Neural Networks." In *ICLR 2020 Workshop on Integration of Deep Neural Models and Differential Equations*, 2020.

[CGW19] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. "Once-for-All: Train One Network and Specialize it for Efficient Deployment." In *International Conference on Learning Representations*, 2019.

[CGZ20] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. "Tinytl: Reduce memory, not parameters for efficient on-device learning." *Advances in Neural Information Processing Systems*, **33**:11285–11297, 2020.

[CH20] Francesco Croce and Matthias Hein. "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks." In *International conference on machine learning*, pp. 2206–2216. PMLR, 2020.

[CLB17] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. "Deep reinforcement learning from human preferences." *Advances in neural information processing systems*, **30**, 2017.

[CLM18] Changhao Chen, Xiaoxuan Lu, Andrew Markham, and Niki Trigoni. "Ionet: Learning to cure the curse of drift in inertial odometry." In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[CMI20] Youngjae Chang, Akhil Mathur, Anton Isopoussu, Junehwa Song, and Fahim Kawsar. "A systematic study of unsupervised domain adaptation for robust human-activity recognition." *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, **4**(1), 2020.

[CMJ18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "{TVM}: An automated {End-to-End} optimizing compiler for deep learning." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

[CMT94] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. "Compiler optimizations for improving data locality." *ACM SIGPLAN Notices*, **29**(11):252–262, 1994.

[CND22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. "Palm: Scaling language modeling with pathways." *arXiv preprint arXiv:2204.02311*, 2022.

[Com94] Pierre Comon. "Independent component analysis, a new concept?" *Signal Proceedings*, **36**(3):287–314, 1994.

[CPC18] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. "Recurrent neural networks for multivariate time series with missing values." *Scientific reports*, **8**(1):1–12, 2018.

[CR22] Nuri Cingillioglu and Alessandra Russo. "pix2rule: End-to-end Neuro-symbolic Rule Learning." *15th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)*, 2022.

[CRF20] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. "Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices." *IEEE Transactions on Circuits and Systems II: Express Briefs*, **67**(5):871–875, 2020.

[CS23] Mahesh Chowdhary and Swapnil Sayan Saha. "On-Sensor Unsupervised Learning and Classification Under 8 KB Memory." In *2023 26th International Conference on Information Fusion*. IEEE, 2023.

[CSF21] Yves Luduvico Coelho, Francisco de Assis Souza dos Santos, Anselmo Frizera-Neto, and Teodiano Freire Bastos-Filho. "A lightweight framework for human activity recognition on wearable devices." *IEEE Sensors Journal*, **21**(21), 2021.

[CSL19] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. "Gen: a general-purpose probabilistic programming system with programmable inference." In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pp. 221–236, 2019.

[CWS19] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. "Visual wake words dataset." *arXiv preprint arXiv:1906.05721*, 2019.

[CZH18] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware." In *International Conference on Learning Representations*, 2018.

[CZL20] Changhao Chen, Peijun Zhao, Chris Xiaoxuan Lu, Wei Wang, Andrew Markham, and Niki Trigoni. "Deep-learning-based pedestrian inertial navigation: Methods, data set, and on-device inference." *IEEE Internet of Things Journal*, **7**(5):4431–4441, 2020.

[CZY21] Kaixuan Chen, Dalin Zhang, Lina Yao, Bin Guo, Zhiwen Yu, and Yunhao Liu. "Deep learning for sensor-based human activity recognition: Overview, challenges, and opportunities." *ACM Computing Surveys (CSUR)*, **54**(4), 2021.

[DB21] Lachit Dutta and Swapna Bharali. "Tinyml meets iot: A comprehensive survey." *Internet of Things*, **16**:100461, 2021.

[DCL19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Lee Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pp. 4171–4186, 2019.

[DDJ21] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, et al. "Tensorflow lite micro: Embedded machine learning for tinyml systems." *Proceedings of Machine Learning and Systems*, **3**:800–811, 2021.

[DEB22] Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. "Multi-Objective Bayesian Optimization over High-Dimensional Search Spaces." In *The 38th Conference on Uncertainty in Artificial Intelligence*, 2022.

[DGL21] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. "A survey of on-device machine learning: An algorithms and learning theory perspective." *ACM Transactions on Internet of Things*, **2**(3):1–49, 2021.

[DKA19] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. "Dmazerunner: Executing perfectly nested loops on dataflow accelerators." *ACM Transactions on Embedded Computing Systems (TECS)*, **18**(5s):1–27, 2019.

[DKB14] Thomas Desautels, Andreas Krause, and Joel W Burdick. "Parallelizing exploration-exploitation tradeoffs in Gaussian process bandit optimization." *The Journal of Machine Learning Research*, **15**(1):3873–3923, 2014.

[DL19] Bradley Denby and Brandon Lucia. “Orbital edge computing: Machine inference in space.” *IEEE Computer Architecture Letters*, **18**(1):59–62, 2019.

[DMC16] Ian Dewancker, Michael McCourt, Scott Clark, Patrick Hayes, Alexandra Johnson, and George Ke. “A strategy for ranking optimization methods using multiple criteria.” In *Workshop on Automatic Machine Learning*, pp. 11–20. PMLR, 2016.

[DNB22] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. “Camaroptera: A Long-Range Image Sensor with Local Inference for Remote Sensing Applications.” *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[DRT09] Markus Deittert, Arthur Richards, Chris A Toomer, and Anthony Pipe. “Engineless unmanned aerial vehicle propulsion by dynamic soaring.” *Journal of guidance, control, and dynamics*, **32**(5):1446–1457, 2009.

[DSS23] Yayun Du, Swapnil Sayan Saha, Sandeep Singh Sandha, Arthur Lovekin, Jason Wu, S. Siddharth, Mahesh Chowdhary, Mohammad Khalid Jawed, and Mani Srivastava. “Neural-Kalman GNSS/INS Navigation for Precision Agriculture.” In *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023.

[DXS23] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. “Palm-e: An embodied multimodal language model.” *arXiv preprint arXiv:2303.03378*, 2023.

[DZB14] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. “Exploiting linear structure within convolutional networks for efficient evaluation.” In *Advances in Neural Information Processing Systems*, pp. 1269–1277, 2014.

[EA12] Philippe Esling and Carlos Agon. “Time-series data mining.” *ACM comp. Surveys (CSUR)*, **45**(1), 2012.

[EHN07] Naser El-Sheimy, Haiying Hou, and Xiaoji Niu. “Analysis and modeling of inertial sensors using Allan variance.” *IEEE Transactions on instrumentation and measurement*, **57**(1):140–149, 2007.

[EM21] Atis Elsts and Ryan McConville. “Are microcontrollers ready for deep learning-based human activity recognition?” *Electronics*, **10**(21), 2021.

[EMH19a] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. “Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution.” In *International Conference on Learning Representations*, 2019.

[EMH19b] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey." *Journal of Machine Learning Research*, **20**(1):1997–2017, 2019.

[EMK19] Mateus Espadoto, Rafael M Martins, Andreas Kerren, Nina ST Hirata, and Alexandru C Telea. "Toward a quantitative survey of dimension reduction techniques." *IEEE Transactions on visualization and Computer graphics*, **27**(3):2153–2173, 2019.

[EWW19] Mahdi Abolfazli Esfahani, Han Wang, Keyu Wu, and Shenghai Yuan. "AbolDeepIO: A novel deep inertial odometry network for autonomous vehicles." *IEEE Transactions on Intelligent Transportation Systems*, **21**(5):1941–1950, 2019.

[FAM19] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul N Whatmough. "SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers." *Advances in Neural Information Processing Systems*, **32**, 2019.

[FC03] F Fdez-Riverola and Juan M Corchado. "Forecasting red tides using an hybrid neuro-symbolic system." *AI Communications*, **16**(4):221–233, 2003.

[FCM19] Maxime Ferrera, Vincent Creuze, Julien Moras, and Pauline Trouvé-Peloux. "AQUALOC: An underwater dataset for visual–inertial–pressure localization." *The International Journal of Robotics Research*, **38**(14):1549–1559, 2019.

[FMM20] Anna Ferrari, Daniela Micucci, Marco Mobilio, and Paolo Napoletano. "On the personalization of classification models for human activity recognition." *IEEE Access*, **8**, 2020.

[FMT22] Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. "UDC: Unified DNAS for Compressible TinyML Models." *Advances in Neural Information Processing Systems*, **35**, 2022.

[Fu11] Tak-chung Fu. "A review on time series data mining." *Engineering Applications of Artificial Intelligence*, **24**(1), 2011.

[GBB22] Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Luis C Lamb, Leo de Penning, BV Illuminoo, Hoifung Poon, and Coppe Gerson Zaverucha. "Neural-symbolic learning and reasoning: A survey and interpretation." *Neuro-Symbolic Artificial Intelligence: The State of the Art*, **342**:1, 2022.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[GCW18] Hristijan Gjoreski, Mathias Ciliberto, Lin Wang, Francisco Javier Ordonez Morales, Sami Mekki, Stefan Valentin, and Daniel Roggen. "The university of sussex-huawei locomotion and transportation dataset for multimodal analytics with mobile devices." *IEEE Access*, **6**:42592–42604, 2018.

[GDC21] Beliz Gunel, Jingfei Du, Alexis Conneau, and Veselin Stoyanov. "Supervised Contrastive Learning for Pre-trained Language Model Fine-tuning." In *International Conference on Learning Representations*, 2021.

[GDY19] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. "Hamiltonian neural networks." *Advances in neural information processing systems*, **32**, 2019.

[GGL19] A Garcez, M Gori, LC Lamb, L Serafini, M Spranger, and SN Tran. "Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning." *Journal of Applied Logics*, **6**(4):611–632, 2019.

[GGN08] Isabelle Guyon, Steve Gunn, Masoud Nikravesh, and Lofti A Zadeh. *Feature extraction: foundations and applications*, volume 207. Springer, 2008.

[GH20] Eduardo C Garrido-Merchán and Daniel Hernández-Lobato. "Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes." *Neurocomputing*, **380**:20–35, 2020.

[GHS21] 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. "Dead-reckoning animal movements in R: a reappraisal using Gundog. Tracks." *Animal Biotelemetry*, **9**(1):1–37, 2021.

[GKS19] Taesik Gong, Yeonsu Kim, Jinwoo Shin, and Sung-Ju Lee. "Metasense: few-shot adaptation to untrained conditions in deep mobile sensing." In *Proceedings of the 17th Conference on embedded networked sensor systems*, 2019.

[GLB19] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. "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*, pp. 199–213, 2019.

[GLL22] Xinyang Geng, Hao Liu, Lisa Lee, Dale Schuurmans, Sergey Levine, and Pieter Abbeel. "Multimodal Masked Autoencoders Learn Transferable Representations." In *First Workshop on Pre-training: Perspectives, Pitfalls, and Paths Forward at ICML 2022*, 2022.

[GP17] Antonio Gulli and Sujit Pal. *Deep learning with Keras.* Packt Publishing Ltd, 2017.

[GQC20] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. "Conformer: Convolution-augmented Transformer for Speech Recognition." *Interspeech*, pp. 5036–5040, 2020.

[GRC20] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. "PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors." *Philosophical Transactions of the Royal Society A*, **378**(2164):20190155, 2020.

[GSG17] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. "Protonn: Compressed and accurate knn for resource-scarce devices." In *International Conference on Machine Learning*, pp. 1331–1340. PMLR, 2017.

[GTC20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. "A survey of deep learning techniques for autonomous driving." *Journal of Field Robotics*, **37**(3):362–386, 2020.

[GWY19] Shupeng Gui, Haotao N Wang, Haichuan Yang, Chen Yu, Zhangyang Wang, and Ji Liu. "Model compression with adversarial robustness: A unified optimization framework." *Advances in Neural Information Processing Systems*, **32**:1285–1296, 2019.

[GXZ21] Ruipeng Gao, Xuan Xiao, Shuli Zhu, Weiwei Xing, Chi Li, Lei Liu, Li Ma, and Hua Chai. "Glow in the Dark: Smartphone Inertial Odometry for Vehicle Tracking in GPS Blocked Environments." *IEEE Internet of Things Journal*, **8**(16):12955–12967, 2021.

[GYM21] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. "Knowledge distillation: A survey." *International Journal of Computer Vision*, **129**:1789–1819, 2021.

[GYX20] Minghao Guo, Yuzhe Yang, Rui Xu, Ziwei Liu, and Dahua Lin. "When nas meets robustness: In search of robust architectures against adversarial attacks." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 631–640, 2020.

[HAR20] Tahera Hossain, Md Ahad, Atiqur Rahman, and Sozo Inoue. "A method for sensor-based activity recognition in missing data scenario." *Sensors*, **20**(14):3811, 2020.

[HCS19] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. "A survey on IoT security: application areas, security threats, and solution architectures." *IEEE Access*, **7**:82721–82743, 2019.

[HG16] Dan Hendrycks and Kevin Gimpel. "Gaussian error linear units (gelus)." *arXiv preprint arXiv:1606.08415*, 2016.

[HGA18] Tahera Hossain, Hiroki Goto, Md Atiqur Rahman Ahad, and Sozo Inoue. "A study on sensor-based activity recognition having missing data." In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pp. 556–561. IEEE, 2018.

[HGD18] Nima Hatami, Yann Gavet, and Johan Debayle. "Classification of time-series images using deep convolutional neural networks." In *Tenth International Conference on machine vision*, volume 10696. SPIE, 2018.

[HI19] Tahera Hossain and Sozo Inoue. "A Comparative study on missing data handling using machine learning for human activity recognition." In *2019 Joint 8th International Conference on Informatics, Electronics & Vision (ICIEV) and 2019 3rd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pp. 124–129. IEEE, 2019.

[HL10] Bastian Hartmann and Norbert Link. "Gesture recognition with inertial sensors and optimized DTW prototypes." In *2010 IEEE International Conference on Systems, Man and Cybernetics*. IEEE, 2010.

[HMD16] Song Han, Huizi Mao, and William J Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." *International Conference on Learning Representations (ICLR)*, 2016.

[Hud89] Paul Hudak. "Conception, evolution, and application of functional programming languages." *ACM computing Surveys (CSUR)*, **21**(3), 1989.

[HWC22] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. "A survey on vision transformer." *IEEE transactions on pattern analysis and machine intelligence*, **45**(1):87–110, 2022.

[HYF20] Sachini Herath, Hang Yan, and Yasutaka Furukawa. "Ronin: Robust neural inertial navigation in the wild: Benchmark, evaluations, & new methods." In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3146–3152. IEEE, 2020.

[HZC17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861*, 2017.

[HZR16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[IHM16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size.” *arXiv preprint arXiv:1602.07360*, 2016.

[JCB20] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. “High accuracy and high fidelity extraction of neural networks.” In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp. 1345–1362, 2020.

[JCL20] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. “Spanbert: Improving pre-training by representing and predicting spans.” *Transactions of the Association for Computational Linguistics*, **8**:64–77, 2020.

[JLG17] Shuo Jiang, Bo Lv, Weichao Guo, Chao Zhang, Haitao Wang, Xinjun Sheng, and Peter B Shull. “Feasibility of wrist-worn, real-time hand, and surface gesture recognition via sEMG and IMU sensing.” *IEEE Transactions on Industrial Informatics*, **14**(8), 2017.

[JLS19] Jeya Vikranth Jeyakumar, Liangzhen Lai, Naveen Suda, and Mani Srivastava. “SenseHAR: a robust virtual activity sensor for smartphones and wearables.” In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pp. 15–28, 2019.

[JLX18] Jeya Vikranth Jeyakumar, Eun Sun Lee, Zhengxu Xia, Sandeep Singh Sandha, Nathan Tausik, and Mani Srivastava. “Deep convolutional bidirectional LSTM based transportation mode recognition.” In *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*, pp. 1606–1615, 2018.

[Jus81] BI Justusson. “Median filtering: Statistical properties.” *Two-Dimensional Digital Signal Processing II*, pp. 161–196, 1981.

[JYS20] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. “Hardware/software co-exploration of neural architectures.” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **39**(12):4805–4815, 2020.

[JZS19] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. “Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search.” In *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[Kah11] Daniel Kahneman. *Thinking, fast and slow.* Macmillan, 2011.

[Kau22] Henry Kautz. "The third AI summer: AAAI Robert S. Engelmore Memorial Lecture." *AI Magazine*, **43**(1):93–104, 2022.

[KB15] Sven Kratz and Maribeth Back. "Towards accurate automatic segmentation of imu-tracked motion gestures." In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in computing systems*, 2015.

[KG15] Nurettin Çağrı Kılıboz and Uğur Güdükbay. "A hand gesture recognition technique for human–computer interaction." *Journal of Visual Communication and Image Representation*, **28**, 2015.

[KGV17] Ashish Kumar, Saurabh Goyal, and Manik Varma. "Resource-efficient machine learning in 2 kb ram for the internet of things." In *International Conference on Machine Learning*, pp. 1935–1944. PMLR, 2017.

[KKH18] Maxat Kulmanov, Mohammed Asif Khan, and Robert Hoehndorf. "DeepGO: predicting protein functions from sequence and interactions using a deep ontology-aware classifier." *Bioinformatics*, **34**(4):660–668, 2018.

[KKL21] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. "Physics-informed machine learning." *Nature Reviews Physics*, **3**(6):422–440, 2021.

[KKN14] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. "A survey of feature selection and feature extraction techniques in machine learning." In *2014 science and information conference*, pp. 372–378. IEEE, 2014.

[KLB22] Kavya Kopparapu, Eric Lin, John G Breslin, and Bharath Sudharsan. "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)*, pp. 79–81. IEEE, 2022.

[KNC23] Siddharth Karamcheti, Suraj Nair, Annie S Chen, Thomas Kollar, Chelsea Finn, Dorsa Sadigh, and Percy Liang. "Language-Driven Representation Learning for Robotics." *arXiv preprint arXiv:2302.12766*, 2023.

[Kri09] A Krizhevsky. "Learning Multiple Layers of Features from Tiny Images." *Master's thesis, University of Tront*, 2009.

[KSB18] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network." *Advances in neural information processing systems*, **31**, 2018.

[KSU19] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. "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)*, pp. 313–317. IEEE, 2019.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *Nature*, **521**(7553):436–444, 2015.

[LC19] Guillaume Lample and François Charton. "Deep Learning For Symbolic Mathematics." In *International Conference on Learning Representations*, 2019.

[LCC21] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. "Memory-efficient Patch-based Inference for Tiny Deep Learning." *Advances in Neural Information Processing Systems*, **34**:2346–2358, 2021.

[LCL20] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, Song Han, et al. "Mcunet: Tiny deep learning on iot devices." *Advances in Neural Information Processing Systems*, **33**:11711–11722, 2020.

[LCS21] Shuheng Li, Ranak Roy Chowdhury, Jingbo Shang, Rajesh K Gupta, and Dezhi Hong. "UniTS: Short-Time Fourier Inspired Neural Networks for Sensory Time Series Classification." In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pp. 234–247, 2021.

[LDL21] Edgar Liberis, Łukasz Dudziak, and Nicholas D Lane. "μNAS: Constrained Neural Architecture Search for Microcontrollers." In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pp. 70–79, 2021.

[LGH18] Ji Lin, Chuang Gan, and Song Han. "Defensive Quantization: When Efficiency Meets Robustness." In *International Conference on Learning Representations*, 2018.

[LJM18] Steven Cheng-Xian Li, Bo Jiang, and Benjamin Marlin. "MisGAN: Learning from Incomplete Data with Generative Adversarial Networks." In *International Conference on Learning Representations*, 2018.

[LKR15] Sara Landset, Taghi M Khoshgoftaar, Aaron N Richter, and Tawfiq Hasanin. "A survey of open source tools for machine learning with big data in the Hadoop ecosystem." *Journal of Big Data*, **2**(1):1–36, 2015.

[LLW20] Yidong Liu, Siting Liu, Yanzhi Wang, Fabrizio Lombardi, and Jie Han. "A survey of stochastic computing neural networks for machine learning applications." *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[LN20] Seulki Lee and Shahriar Nirjon. "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*, pp. 34–40, 2020.

[LS99] Daniel D Lee and H Sebastian Seung. "Learning the parts of objects by non-negative matrix factorization." *Nature*, **401**(6755):788–791, 1999.

[LSC18] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus." *arXiv preprint arXiv:1801.06601*, 2018.

[LSG21] Junnan Li, Ramprasaath Selvaraju, Akhilesh Gotmare, Shafiq Joty, Caiming Xiong, and Steven Chu Hong Hoi. "Align before fuse: Vision and language representation learning with momentum distillation." *Advances in neural information processing systems*, **34**:9694–9705, 2021.

[LSY18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search." In *International Conference on Learning Representations*, 2018.

[LVR16] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. "Temporal convolutional networks: A unified approach to action segmentation." In *European Conference on Computer Vision*, pp. 47–54. Springer, 2016.

[LZC22] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. "On-device training under 256kb memory." *Advances in Neural Information Processing Systems*, **35**, 2022.

[MA20] Jesse Mu and Jacob Andreas. "Compositional explanations of neurons." *Advances in Neural Information Processing Systems*, **33**:17153–17163, 2020.

[MBG21] 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. "On-device federated learning with flower." *On-Device Intelligence Workshop at MLSys*, 2021.

[MCB21] Mark Mazumder, Sharad Chitlangia, Colby Banbury, Yiping Kang, Juan Manuel Ciro, Keith Achorn, Daniel Galvez, Mark Sabini, Peter Mattson, David Kanter, et al. "Multilingual Spoken Words Corpus." In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[MCC19] Mohammad Malekzadeh, Richard G Clegg, Andrea Cavallaro, and Hamed Haddadi. "Mobile sensor data anonymization." In *Proceedings of the international conference on internet of things design and implementation*, pp. 49–58, 2019.

[MDK18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. "Deepproblog: Neural probabilistic logic programming." *Advances in Neural Information Processing Systems*, **31**, 2018.

[MDL23] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, et al. "Augmented language models: a survey." *arXiv preprint arXiv:2302.07842*, 2023.

[MFL19] Kaixin Ma, Jonathan Francis, Quanyang Lu, Eric Nyberg, and Alessandro Oltramari. "Towards Generalizable Neuro-Symbolic Systems for Commonsense Question Answering." In *Proceedings of the First Workshop on Commonsense Inference in Natural Language Processing*, pp. 22–32, 2019.

[MGF20] Meiyi Ma, Ji Gao, Lu Feng, and John Stankovic. "STLnet: Signal temporal logic enforced multivariate recurrent neural networks." *Advances in Neural Information Processing Systems*, **33**:14604–14614, 2020.

[MGK18] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. "The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision." In *International Conference on Learning Representations*, 2018.

[MH08] Laurens Van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." *Journal of Machine Learning Research*, **9**(11), 2008.

[MJ95] GJ Morgan-Owen and GT Johnston. "Differential GPS positioning." *Electronics & Communication Engineering Journal*, **7**(1):11–21, 1995.

[MKH21] Hashan Roshantha Mendis, Chih-Kai Kang, and Pi-cheng Hsiu. "Intermittent-Aware Neural Architecture Search." *ACM Transactions on Embedded Computing Systems (TECS)*, **20**(5s):1–27, 2021.

[MLR16] Anton Milan, Laura Leal-Taixé, Ian Reid, Stefan Roth, and Konrad Schindler. "MOT16: A benchmark for multi-object tracking." *arXiv preprint arXiv:1603.00831*, 2016.

[MRS22] Michele Magno, Andrea Ronco, and Lukas Schulthess. "On-Sensors AI with Novel ST Sensors: Performance and Evaluation in a Real Application Scenario." *TinyML Summit 2022*, 2022.

[MSG14] Dan Morris, T Scott Saponas, Andrew Guillory, and Ilya Kelner. "RecoFit: using a wearable sensor to find, recognize, and count repetitive exercises." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014.

[MTC22] Ludovico Mitchener, David Tuckey, Matthew Crosby, and Alessandra Russo. "Detect, Understand, Act: A Neuro-symbolic Hierarchical Reinforcement Learning Framework." *Machine Learning*, **111**(4):1523–1549, 2022.

[New80] Allen Newell. "Physical symbol systems." *Cognitive science*, **4**(2):135–183, 1980.

[NPD22] Sabrina M Neuman, Brian Plancher, Bardienus P Duisterhof, Srivatsan Krishnan, Colby Banbury, Mark Mazumder, Shvetank Prakash, Jason Jabbour, Aleksandra Faust, Guido CHE de Croon, et al. "Tiny robot learning: challenges and directions for machine learning in resource-constrained robots." In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 296–299. IEEE, 2022.

[ODZ16] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. "WaveNet: A Generative Model for Raw Audio." In *9th ISCA WKSH on Speech Synthesis WKSH (SSW 9)*, 2016.

[Ope23] OpenAI. "GPT-4 Technical Report.", 2023.

[OWJ22] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. "Training language models to follow instructions with human feedback." *Advances in Neural Information Processing Systems*, **35**:27730–27744, 2022.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[PCA20] Riccardo Perego, Antonio Candelieri, Francesco Archetti, and Danilo Pau. "Tuning deep neural network's hyperparameters constrained to deployability on tiny systems." In *International Conference on Artificial Neural Networks*, pp. 92–103. Springer, 2020.

[PCA22] Riccardo Perego, Antonio Candelieri, Francesco Archetti, and Danilo Pau. "AutoTinyML for microcontrollers: Dealing with black-box deployability." *Expert Systems with Applications*, **207**:117876, 2022.

[PDP19] Shishir G Patil, Don Kurian Dennis, Chirag Pabbaraju, Nadeem Shaheer, Harsha Vardhan Simhadri, Vivek Seshadri, Manik Varma, and Prateek Jain. "Gesturepod: Enabling on-device gesture-based interaction for white cane users." In *Proceedings of the 32nd Annual ACM symposium on User Interface Software and technology*, 2019.

[Pea19] Judea Pearl. "The seven tools of causal inference, with reflections on machine learning." *Communications of the ACM*, **62**(3):54–60, 2019.

[PMS17] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. "Neuro-Symbolic Program Synthesis." In *International Conference on Learning Representations*, 2017.

[PVG11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python." *the Journal of machine Learning research*, **12**:2825–2830, 2011.

[PXJ22] Ren Pang, Zhaohan Xi, Shouling Ji, Xiapu Luo, and Ting Wang. "On the Security Risks of {AutoML}." In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3953–3970, 2022.

[PY10] Sinno Jialin Pan and Qiang Yang. "A survey on transfer learning." *IEEE Transactions on knowledge and data engineering*, **22**(10):1345–1359, 2010.

[QM02] Honghui Qi and John B Moore. "Direct Kalman filtering approach for GPS/INS integration." *IEEE Transactions on Aerospace and Electronic Systems*, **38**(2):687–693, 2002.

[RAA12] Marcus Rohrbach, Sikandar Amin, Mykhaylo Andriluka, and Bernt Schiele. "A database for fine grained activity detection of cooking activities." In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1194–1201. IEEE, 2012.

[RAR21a] Haoyu Ren, Darko Anicic, and Thomas A Runkler. "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*, pp. 126–135, 2021.

[RAR21b] Haoyu Ren, Darko Anicic, and Thomas A Runkler. "Tinyol: Tinyml with online-learning on microcontrollers." In *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2021.

[Ray21] Partha Pratim Ray. "A review on TinyML: State-of-the-art and prospects." *Journal of King Saud University-Computer and Information Sciences*, 2021.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning internal representations by error propagation." Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[RHW19] Yuji Roh, Geon Heo, and Steven Euijong Whang. "A survey on data collection for machine learning: a big data-ai integration perspective." *IEEE Transactions on Knowledge and Data Engineering*, **33**(4):1328–1347, 2019.

[Ric08] Elaine Rich et al. *Automata, computability and complexity: theory and app.* Pearson Prentice Hall Upper Saddle River, 2008.

[RKA22] Visal Rajapakse, Ishan Karunanayake, and Nadeem Ahmed. "Intelligence at the extreme edge: a survey on reformable TinyML." *ACM Computing Surveys*, 2022.

[RKH21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. "Learning transferable visual models from natural language supervision." In *International conference on machine learning*, pp. 8748–8763. PMLR, 2021.

[ROF21] Wamiq Raza, Anas Osman, Francesco Ferrini, and Francesco De Natale. "Energy-Efficient Inference on the Edge Exploiting TinyML Capabilities for UAVs." *Drones*, **5**(4):127, 2021.

[ROS16] Jorge-L Reyes-Ortiz, Luca Oneto, Albert Samà, Xavier Parra, and Davide Anguita. "Transition-aware human activity recognition using smartphones." *Neurocomputing*, **171**:754–767, 2016.

[RRN21] Leonardo Ravaglia, Manuele Rusci, Davide Nadalini, Alessandro Capotondi, Francesco Conti, and Luca Benini. "A tinyml platform for on-device continual learning with quantized latent replays." *IEEE Journal on Emerging and Selected Topics in Circuits and systems*, **11**(4), 2021.

[RS12] Attila Reiss and Didier Stricker. "Creating and benchmarking a new dataset for physical activity monitoring." In *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments*, pp. 1–8, 2012.

[RSZ22] Andrea Ronco, Lukas Schulthess, David Zehnder, and Michele Magno. "Machine Learning In-Sensors: Computation-enabled Intelligent Sensors For Next Generation of IoT." In *2022 IEEE Sensors*. IEEE, 2022.

[RXC21] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. "A comprehensive survey of neural architecture search: Challenges and solutions." *ACM Computing Surveys (CSUR)*, **54**(4):1–34, 2021.

[RZH19] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet." *ACM Computing Surveys (CSUR)*, **52**(6):1–36, 2019.

[SAF20] Sandeep Singh Sandha, Mohit Aggarwal, Igor Fedorov, and Mani Srivastava. "Mango: A python library for parallel hyperparameter tuning." In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3987–3991. IEEE, 2020.

[SAN19] Sandeep Singh Sandha, Fatima M Anwar, Joseph Noor, and Mani Srivastava. "Exploiting smartphone peripherals for precise time synchronization." In *2019 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 1–6. IEEE, 2019.

[San21] Sandeep Singh Sandha. "Parameter search spaces use to evaluate Mango on classifiers." https://github.com/ARM-software/mango/blob/master/benchmarking/Parameter_Spaces_Evaluated.ipynb, 2021.

[SAS21] Sandeep Singh Sandha, Mohit Aggarwal, Swapnil Sayan Saha, and Mani Srivastava. "Enabling Hyperparameter Tuning of Machine Learning Classifiers in Production." In *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 262–271. IEEE, 2021.

[SBB15] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. "Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition." In *Proceedings of the 13th ACM Conference on embedded networked Sensor Systems*, pp. 127–140, 2015.

[SBI14] Muhammad Shoaib, Stephan Bosch, Ozlem Durmaz Incel, Hans Scholten, and Paul JM Havinga. "Fusion of smartphone motion sensors for physical activity recognition." *Sensors*, **14**(6):10146–10176, 2014.

[SDS20] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. "Green ai." *Communications of the ACM*, **63**(12):54–63, 2020.

[SDS23a] Swapnil Sayan Saha, Caden Davis, Sandeep Singh Sandha, Junha Park, Joshua Geronimo, Luis Antonio Garcia, and Mani Srivastava. "LocoMote: AI-driven Sensor Tags for Fine-Grained Undersea Localization and Sensing." *IEEE Internet of Things Journal*, 2023.

[SDS23b] Swapnil Sayan Saha, Yayun Du, Sandeep Singh Sandha, Luis Antonio Garcia, Mohammad Khalid Jawed, and Mani Srivastava. "Inertial Navigation on Extremely Resource-Constrained Platforms: Methods, Opportunities and Challenges." In *2023 IEEE/ION Position, Location and Navigation Symposium (PLANS)*. IEEE, 2023.

[SFM20] Aishwarya Sivaraman, Golnoosh Farnadi, Todd Millstein, and Guy Van den Broeck. "Counterexample-guided learning of monotonic neural networks." *Advances in Neural Information Processing Systems*, **33**:11936–11948, 2020.

[SG16] Luciano Serafini and Artur S d'Avila Garcez. "Learning and reasoning with logic tensor networks." In *Conference of the Italian Association for Artificial Intelligence*, pp. 334–348. Springer, 2016.

[SGT08] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The graph neural network model." *IEEE transactions on neural networks*, **20**(1):61–80, 2008.

[SHK17] Farhad Shahmohammadi, Anahita Hosseini, Christine E King, and Majid Sarrafzadeh. "Smartwatch based activity recognition using active learning." In *2017 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*. IEEE, 2017.

[SHS17] Chenguang Shen, Bo-Jhang Ho, and Mani Srivastava. "Milift: Efficient smartwatch-based workout tracking using automatic segmentation." *IEEE Transactions on Mobile Computing*, **17**(7), 2017.

[SHZ18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "Mobilenetv2: Inverted residuals and linear bottlenecks." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[SKK10] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. "Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design." In *Proceedings of the 27th International Conference on Machine Learning*, pp. 1015–1022, 2010.

[SKK12] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias W Seeger. "Information-theoretic regret bounds for gaussian process optimization in the bandit setting." *IEEE transactions on information theory*, **58**(5):3250–3265, 2012.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms." *Advances in neural information processing systems*, **25**:2951–2959, 2012.

[SLA19] Maarten Sap, Ronan Le Bras, Emily Allaway, Chandra Bhagavatula, Nicholas Lourie, Hannah Rashkin, Brendan Roof, Noah A Smith, and Yejin Choi. "Atomic: An atlas of machine commonsense for if-then reasoning." In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pp. 3027–3035, 2019.

[SLC20] Di Shao, Xiao Liu, Ben Cheng, Owen Wang, and Thuong Hoang. "Edge4Real: A cost-effective edge computing based human behaviour recognition system for human-centric software engineering." In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1287–1291, 2020.

[Smo87] Paul Smolensky. "Connectionist AI, symbolic AI, and the brain." *Artificial Intelligence Review*, **1**(2):95–109, 1987.

[SNA20] Sandeep Singh Sandha, Joseph Noor, Fatima M Anwar, and Mani Srivastava. "Time awareness in deep learning-based multimodal fusion across smartphone platforms." In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 149–156. IEEE, 2020.

[SNK20] Megha Srivastava, Besmira Nushi, Ece Kamar, Shital Shah, and Eric Horvitz. "An empirical analysis of backward compatibility in machine learning systems." In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3272–3280, 2020.

[Soi99] Pierre Soille et al. *Morphological image analysis: principles and applications*, volume 2. Springer, 1999.

[SPN22] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model." *arXiv preprint arXiv:2201.11990*, 2022.

[SS16] Timo Sztyler and Heiner Stuckenschmidt. "On-body localization of wearable devices: An investigation of position-aware activity recognition." In *2016 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 1–9. IEEE, 2016.

[SSA23] Swapnil Sayan Saha, Sandeep Singh Sandha, Mohit Aggarwal, Brian Wang, Liying Han, Julian de Gortari Briseno, and Mani Srivastava. "TinyNS: Platform-Aware Neurosymbolic Auto Tiny Machine Learning." *ACM Transactions on Embedded Computing Systems*, 2023.

[SSF04] Bruno Sinopoli, Luca Schenato, Massimo Franceschetti, Kameshwar Poolla, Michael I Jordan, and Shankar S Sastry. "Kalman filtering with intermittent observations." *IEEE transactions on Automatic Control*, **49**(9):1453–1464, 2004.

[SSG22] Swapnil Sayan Saha, Sandeep Singh Sandha, Luis Antonio Garcia, and Mani Srivastava. "Tinyodom: Hardware-aware efficient neural inertial navigation." *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, **6**(2):1–32, 2022.

[SSH21] Jiankai Sun, Hao Sun, Tian Han, and Bolei Zhou. "Neuro-Symbolic Program Search for Autonomous Driving Decision Module Design." In *Conference on Robot Learning*, pp. 21–30. PMLR, 2021.

[SSM97] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. "Kernel principal component analysis." In *International Conference on Artificial Neural Networks*, pp. 583–588. Springer, 1997.

[SSP22] Swapnil Sayan Saha, Sandeep Singh Sandha, Siyou Pei, Vivek Jain, Ziqi Wang, Yuchen Li, Ankur Sarker, and Mani Srivastava. "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):1–34, 2022.

[SSS17] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. "On a formal model of safe and scalable self-driving cars." *arXiv preprint arXiv:1708.06374*, 2017.

[SSS20] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. "Deep Convolutional Bidirectional LSTM for Complex Activity Recognition with Missing Data." In *Human Activity Recognition Challenge*, pp. 39–53. Springer, 2020.

[SSS22a] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. "Machine Learning for Microcontroller-Class Hardware: A Review." *IEEE Sensors Journal*, **22**(22):21362–21390, 2022.

[SSS22b] Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. "Toward verified artificial intelligence." *Communications of the ACM*, **65**(7):46–55, 2022.

[ST94] Jianbo Shi and Carlo Tomasi. "Good features to track." In *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, pp. 593–600. IEEE, 1994.

[STR21] Muhammad Shafique, Theocharis Theocharides, Vijay Janapa Reddy, and Boris Murmann. "TinyML: current progress, research challenges, and future roadmap." In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1303–1306. IEEE, 2021.

[Sud21] Bharath Sudharsan *et al.* "Ml-mcu: A framework to train ml classifiers on mcu-based iot edge devices." *IEEE Internet of Things Journal*, **9**(16), 2021.

[SXX20] Yantao Shen, Yuanjun Xiong, Wei Xia, and Stefano Soatto. "Towards backward-compatible representation learning." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6368–6377, 2020.

[SYB21] Bharath Sudharsan, Piyush Yadav, John G Breslin, and Muhammad Intizar Ali. "Train++: An incremental ml model training algorithm to create self-learning iot devices." In *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*. IEEE, 2021.

[SZE21] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart, and Pascal Hitzler. "Neurosymbolic artificial intelligence." *AI Communications*, pp. 1–13, 2021.

[SZS20] Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. "Learning differentiable programs with admissible neural heuristics." *Advances in neural information processing systems*, **33**:4940–4952, 2020.

[TDH22] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. "Lamda: Language models for dialog applications." *arXiv preprint arXiv:2201.08239*, 2022.

[TFZ21] Urmish Thakker, Igor Fedorov, Chu Zhou, Dibakar Gope, Matthew Mattina, Ganesh Dasika, and Jesse Beu. "Compressing RNNs to Kilobyte Budget for IoT Devices Using Kronecker Products." *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, **17**(4):1–18, 2021.

[TIR20] Shreshth Tuli, Shashikant Ilager, Kotagiri Ramamohanarao, and Rajkumar Buyya. "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks." *IEEE transactions on mobile computing*, **21**(3):940–954, 2020.

[TK91] Carlo Tomasi and Takeo Kanade. "Detection and tracking of point." *Int J Comput Vis*, **9**:137–154, 1991.

[TKB20] Markus Thill, Wolfgang Konen, and Thomas Bäck. "Time series encodings with temporal convolutional networks." In *International Conference on Bioinspired Methods and Their Applications*, pp. 161–173. Springer, 2020.

[TKV21] Kristof T'Jonck, Chandrakanth R Kancharla, Jens Vankeirsbilck, Hans Hallez, Jeroen Boydens, and Bozheng Pang. "Real-time activity tracking using TinyML to support elderly care." In *2021 XXX International Scientific Conference Electronics (ET)*. IEEE, 2021.

[TMZ22] Kushal Tirumala, Aram Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. "Memorization without overfitting: Analyzing the training dynamics of large language models." *Advances in Neural Information Processing Systems*, **35**:38274–38290, 2022.

[TSK21] Megan Tjandrasuwita, Jennifer J Sun, Ann Kennedy, and Yisong Yue. "Interpreting Expert Annotation Differences in Animal Behavior." In *CVPR 2021 Workshop on CV4Animation*, 2021.

[VKE19] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. "Legendre Memory Units: Continuous-Time Representation in Recurrent Neural Networks." *Advances in Neural Information Processing Systems*, **32**:15570–15579, 2019.

[VSP17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems*, **30**, 2017.

[VV92] Rein Van Den Boomgaard and Richard Van Balen. "Methods for fast morphological image transforms using bitmapped binary images." *CVGIP: Graphical Models and Image Processing*, **54**(3):252–258, 1992.

[VVM21] Dinesh C Verma, Archit Verma, and Utpal Mangla. "Addressing the Limitations of AI/ML in creating Cognitive Solutions." In *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 189–196. IEEE, 2021.

[VXT21] Marc Roig Vilamala, Tianwei Xing, Harrison Taylor, Luis Garcia, Mani Srivastava, Lance Kaplan, Alun Preece, Angelika Kimmig, and Federico Cerutti. "Using DeepProbLog to perform Complex Event Processing on an Audio Stream." In *Tenth International Workshop on Statistical Relational AI*, 2021.

[Wan04] Zhou Wang *et al.* "Image quality assessment: from error visibility to structural similarity." *IEEE Tran. on image proc.*, **13**(4), 2004.

[War18] Pete Warden. "Speech commands: A dataset for limited-vocabulary speech recognition." *arXiv preprint arXiv:1804.03209*, 2018.

[WB02] Zhou Wang and Alan C Bovik. "A universal image quality index." *IEEE signal proc. let.*, **9**(3), 2002.

[WCY19] Yan Wang, Shuang Cang, and Hongnian Yu. "A survey on wearable sensor modality centred human activity recognition in health care." *Expert Systems with Applications*, **137**:167–190, 2019.

[WFS20] Alexander Wong, Mahmoud Famouri, and Mohammad Javad Shafiee. "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.

[WGD19] Johann-P Wolff, Florian Gruetzmacher, Rainer Dorsch, Rolf Kaack, Lars Middendorf, and Christian Haubelt. "Towards Automated Prototyping of Gesture Recognition Systems for Wearable Devices using Inertial Sensors." In *Smart Systems Integration; 13th International Conference and Exhibition on Integration Issues of Miniaturized Systems*. VDE, 2019.

[WKM22] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. "Self-Instruct: Aligning Language Model with Self Generated Instructions." *arXiv preprint arXiv:2212.10560*, 2022.

[WL21] Feng Wang and Huaping Liu. "Understanding the behaviour of contrastive loss." In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2495–2504, 2021.

[WR15] Faisal Waris and Robert G Reynolds. "Using cultural algorithms to improve wearable device gesture recognition performance." In *2015 IEEE symposium series on computational intelligence*. IEEE, 2015.

[WS19] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[WSJ16] Jian Wu, Lu Sun, and Roozbeh Jafari. "A wearable system for recognizing American sign language in real-time using IMU and surface EMG sensors." *IEEE Journal of biomedical and health informatics*, **20**(5), 2016.

[WSW22] Ziqi Wang, Ankur Sarker, Jason Wu, Derek Hua, Gaofeng Dong, Akash Deep Singh, and Mani B Srivastava. "Capricorn: Towards Real-time Rich Scene Analysis Using RF-Vision Sensor Fusion." In *Proceedings of the 20th Conference on Embedded Networked Sensor Systems*, 2022.

[WWS22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H Chi, Quoc V Le, Denny Zhou, et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." In *Advances in Neural Information Processing Systems*, 2022.

[WXC21] Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang. "Cvt: Introducing convolutions to vision transformers." In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 22–31, 2021.

[WYM22] Peng Wang, An Yang, Rui Men, Junyang Lin, Shuai Bai, Zhikang Li, Jianxin Ma, Chang Zhou, Jingren Zhou, and Hongxia Yang. "Ofa: Unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework." In *International Conference on Machine Learning*, pp. 23318–23340. PMLR, 2022.

[XDH18] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. "Scaling for edge inference of deep neural networks." *Nature Electronics*, **1**(4):216–222, 2018.

[XGV20] Tianwei Xing, Luis Garcia, Marc Roig Vilamala, Federico Cerutti, Lance Kaplan, Alun Preece, and Mani Srivastava. "Neuroplex: learning to detect complex events in sensor networks through knowledge injection." In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pp. 489–502, 2020.

[XSB18] Tianwei Xing, Sandeep Singh Sandha, Bharathan Balaji, Supriyo Chakraborty, and Mani Srivastava. "Enabling edge devices that learn from each other: Cross modal training for activity recognition." In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, pp. 37–42, 2018.

[XYY23] Haiyang Xu, Qinghao Ye, Ming Yan, Yaya Shi, Jiabo Ye, Yuanhong Xu, Chenliang Li, Bin Bi, Qi Qian, Wei Wang, et al. "mPLUG-2: A Modularized Multi-modal Foundation Model Across Text, Image and Video." *arXiv preprint arXiv:2302.00402*, 2023.

[XZT21] Huatao Xu, Pengfei Zhou, Rui Tan, Mo Li, and Guobin Shen. "Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications." In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pp. 220–233, 2021.

[YJS18] Jinsung Yoon, James Jordon, and Mihaela Schaar. "Gain: Missing data imputation using generative adversarial nets." In *International conference on machine learning*, pp. 5689–5698. PMLR, 2018.

[YK09] Lexiang Ye and Eamonn Keogh. "Time series shapelets: a new primitive for data mining." In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge discovery and data mining*, 2009.

[YLD19] Jiecao Yu, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. "Tf-net: Deploying sub-byte deep neural networks on microcontrollers." *ACM Transactions on Embedded Computing Systems (TECS)*, **18**(5s):1–21, 2019.

[YPJ19] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, et al. "Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks." In *The World Wide Web Conference*, pp. 2192–2202, 2019.

[YWG18] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. "Neural-symbolic vqa: Disentangling reasoning from vision and language understanding." *Advances in neural information processing systems*, **31**, 2018.

[YXK21] Sijie Yan, Yuanjun Xiong, Kaustav Kundu, Shuo Yang, Siqi Deng, Meng Wang, Wei Xia, and Stefano Soatto. "Positive-congruent training: Towards regression-free model updates." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14299–14308, 2021.

[YXL19] Shaokai Ye, Kaidi Xu, Sijia Liu, Hao Cheng, Jan-Henrik Lambrechts, Huan Zhang, Aojun Zhou, Kaisheng Ma, Yanzhi Wang, and Xue Lin. "Adversarial robustness vs. model compression, or both?" In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 111–120, 2019.

[YZS18] Jinsung Yoon, William R Zame, and Mihaela van der Schaar. "Estimating missing data in temporal data streams using multi-directional recurrent neural networks." *IEEE Transactions on Biomedical Engineering*, **66**(5):1477–1490, 2018.

[YZY22] Jiangchao Yao, Shengyu Zhang, Yang Yao, Feng Wang, Jianxin Ma, Jianwei Zhang, Yunfei Chu, Luo Ji, Kunyang Jia, Tao Shen, et al. "Edge-cloud polarization and collaboration: A comprehensive survey for ai." *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[ZBI19] Albert Zeyer, Parnia Bahar, Kazuki Irie, Ralf Schlüter, and Hermann Ney. "A comparison of transformer and lstm encoder decoder models for asr." In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pp. 8–15. IEEE, 2019.

[ZCS98] Jie Zhou, Daniel L Civco, and John A Silander. "A wavelet transform method to merge Landsat TM and SPOT panchromatic data." *Intl. J. of remote sensing*, **19**(4), 1998.

[ZKK20] Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl. "Tracking objects as points." In *European Conference on Computer Vision*, pp. 474–490. Springer, 2020.

[ZL17] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning." *International Conference on Learning Representations (ICLR)*, 2017.

[ZLL23] Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, et al. "A comprehensive survey on pretrained foundation models: A history from bert to chatgpt." *arXiv preprint arXiv:2302.09419*, 2023.

[ZSH22] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. "Least-to-most prompting enables complex reasoning in large language models." *arXiv preprint arXiv:2205.10625*, 2022.

[ZSJ22] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. "Bytetrack: Multi-object tracking by associating every detection box." In *European Conference on Computer Vision*, pp. 1–21. Springer, 2022.