ACE: Automated Optimization Towards Iterative Classification in Edge Health Monitors

Yuxuan Wang, Lara Orlandic, Simone Machetti, Giovanni Ansaloni, *Member, IEEE*, and David Atienza, *Fellow, IEEE*

***Abstract*—Wearable devices for health monitoring are essential for tracking individuals' health status and facilitating early detection of diseases. However, processing biomedical signals online for real-time monitoring is challenging due to limited computational resources on edge devices. To address this challenge, we propose an application-agnostic methodology called ACE (Automated optimization towards classification on the Edge). ACE converts a health monitoring algorithm with feature extraction and classification into an iterative detection process, incorporating algorithms of varying complexities and minimizing re-computation of shared data. First, ACE decomposes a monolithic model, employing a single feature set and classifier, into multiple algorithms with different computational complexities. Then, our automatic analysis tool integrates buffering logic into these algorithms to prevent re-computation of shared computational-intensive data. The optimized algorithm is then converted into a low-level language in C for deployment. During runtime, the system initiates monitoring with the lowest complexity algorithm and iteratively involves algorithms with higher complexity without recomputing the existing data. The iteration process continues until a pre-defined confidence threshold is met. We demonstrate the effectiveness of ACE on two biomedical applications: seizure detection and emotional state classification. ACE achieves at least 28.9% and 18.9% runtime savings without any accuracy loss on a Cortex-A9 edge platform for the two benchmarks, respectively. We discuss and demonstrate how ACE can be used by designers of such biomedical algorithms to automatically optimize and deploy their applications on the edge.**

***Index Terms*—Health monitoring, edge computing, runtime optimization, embedded deployment, iterative classification.**

I. Introduction

CONTINUOUS and real-time health monitoring is essential to minimize the risk of disease progression and enable early prevention of further health complications. Medical surveillance using wearable devices has garnered significant attention in managing individuals' health conditions at low cost [1], [2]. Its applications span a wide spectrum of biomedical fields, such as seizure detection [3], [4], early prediction of Alzheimer's disease [5], and other monitoring of chronic disorders or neurocognitive conditions [6].

Deploying healthcare monitoring applications on edge devices is challenging due to the computational limitations of the platforms, which are often constrained in terms of computational resources and energy consumption [7], [8]. These factors hinder edge devices from performing the complex computations required for bio-signal processing. To run healthcare monitoring workloads on wearable devices, algorithms must be adapted to meet the constraints of embedded systems.

Biomedical Artificial Intelligence (AI) algorithms typically involve three key steps: preprocessing, which includes signal detection and denoising [9], [10], feature extraction, which derives a domain-specific set of features within processing windows [11], [12], and a machine learning (ML) classification [13], which employs classifiers to infer the subject's condition based on the extracted features.

Optimizing each step for execution on resource-constrained wearable devices requires careful considerations in runtime and energy reduction, embedded memory management, and converting high-level algorithms to a low-level programming language. Thus, the effort required to manually optimize algorithms prevents a wide range of health applications from gaining edge execution benefits. To bridge this gap, we propose Automated optimization towards Classification on the Edge (ACE): an automatic methodology that transforms a high-level biomedical AI algorithm into an edge-friendly adaptation and converts the optimized algorithm into a low-level implementation in C. Such an approach could be used by biomedical AI application developers to automatically optimize and run their code on wearable devices.

Our methodology is based on the observation that, in health monitoring applications, using all available features at every inference window may not be required, causing an unnecessary computational burden for biosignal windows with little noise and/or clear patterns. Thus, we consider multiple feature sets and corresponding classifiers operating at varying complexities at runtime. We further enhance this iterative process with automatic buffering of reused variable values. Specifically, the main contributions of our works are as follows:

Received 19 April 2024; revised 8 June 2024 and 16 July 2024; accepted 17 July 2024. Date of publication 25 September 2024; date of current version 5 February 2025. This work was supported in part by the Swiss NSF under Grant 10.002.812: Edge-Companions: Hardware/Software Co-Optimization Toward Energy-Minimal Health Monitoring at the Edge, in part by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project, and in part by the EU's Horizon 2020 grant agreement no. 101017915 (DIGIPREDICT). This paper was recommended by Associate Editor A. Cossettini. *(Corresponding author: Yuxuan Wang.)*

The authors are with the Embedded System Laboratory, École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland (e-mail: yuxuan.wang@epfl.ch; lara.orlandic@epfl.ch; simone.machetti@epfl.ch; giovanni.ansaloni@epfl.ch; david.atienza@epfl.ch).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TBCAS.2024.3468160.

Digital Object Identifier 10.1109/TBCAS.2024.3468160

- We develop a toolchain to generate the feature subsets and classifiers operating at different computation complexities for a broad range of applications.
- We present an automatic analysis tool designed to identify shared computation-intensive data in feature extraction and adapt the algorithm to buffer this data[1].
- We build a framework that converts the optimized algorithm to low-level languages for deployment.
- We apply our automated optimization methodology to two existing state-of-the-art (SoA) applications: seizure detection and emotional state classification. Our results show runtime savings of at least 28.9% and 18.9% across platforms respectively with no loss in F-1 score.

The rest of the paper is organized as follows: Section II presents related works in the context of edge health monitoring. Section III describes the proposed methodology operating in a general healthcare application. Then, Section IV and Section V describe the experimental setup and show the results achieved by our methodology. Lastly, Section VI concludes our work.

II. Related Work

Two main approaches are commonly employed to alleviate the computation burden on edge devices: reducing the inputs to be processed and reducing the algorithm complexity. Belonging to the first category, the work in [14] adjusts the number of sensors involved to reduce the processing data based on the activity of subjects such as sitting, driving, and walking outdoors. Similarly, the authors of [15] aim to improve the energy efficiency of the sensor nodes by adjusting their sampling frequency based on the quality of the input signal. By processing fewer data samples under pre-defined circumstances, the storage and computational burdens of the devices are alleviated. However, this approach employs a single model for feature extraction and classification across all inputs, disregarding the adaptation of subsequent data processing algorithms to accommodate the down-sampled data.

As for the second approach, that of reducing complexity algorithmically, most proposals aim to achieve a trade-off between classification cost and algorithm performance by developing multiple classifiers with varying computational complexities [16], [17], [18]. However, in addition to the computational cost of classification, pre-processing and feature extraction also contribute to the runtime, energy expenditure, and memory consumption of the embedded system [19], [20].

Most proposals aim to identify optimal subsets of features extracted from the corresponding captured signals to mitigate the computational cost of feature extraction [21], [22], [23]. The works [24], [25], [26] consider multiple feature sets with different complexities and select one predefined feature set for each window based on the latest classification result. The one-time prediction methods are designed offline to achieve energy efficiency but fail to address potential accuracy degradation without more complex algorithms involved during runtime.

To maintain control of ML performance during runtime, the framework presented in [27] adopts, as we also do in our work, an iterative approach to ensure the confidence of the output. Such an incremental approach dynamically increases the algorithm complexity at application runtime by expanding the feature set based on the output confidence of the classifier. Nevertheless, as opposed to us, their strategy lacks automated data dependency analysis and buffering, so that feature sets must be recomputed anew from the raw captured signal each time, leading to highly redundant computations.

Table I summarizes the figures of merits (FoM) of the aforementioned SoA methods for specific biomedical tasks evaluated by us. The algorithms proposed in [21], [23] perform feature extraction and classifier developed offline without considering online savings. The remaining works [27], [28] consider the online runtime savings achieved while maintaining a certain ML loss tolerance. Specifically, [28] reports 16.80% runtime saving with 1.84% gmean loss and [27] achieves 6.1% speedup with 1% F-1 score loss. Additionally, most proposals are evaluated on computers instead of on edge devices. Our methodology is not limited to a specific biomedical application or machine learning classifier and considers edge performance, which most SoA methods do not investigate. To highlight this generality, we explore its applicability in two very diverse applications (seizure detection and emotional state classification), showcasing 35.7% and 49.0% runtime savings within 1% F-1 score degradation on edge devices, respectively.

III. Methodology

Fig. 1 plots the workflow of transforming a health monitoring algorithm into an efficient embedded implementation with multiple feature sets and classifiers using ACE. The workflow operates on a general health monitoring algorithm. It produces an efficiently optimized version which performs algorithms of varying computational complexity levels sequentially, while buffering data to prevent re-computation during the iteration process. The input health monitoring algorithm utilizes a monolithic model, which is a model that extracts all available features and performs one classification based on them. In the offline phase, the conversion process consists of two parts: (i) developing an incremental classification strategy, which involves extracting subsets of features from the monolithic feature set and training ML classifiers on these subsets, and (ii) identifying the variables that need to be buffered to prevent re-computation during the iterative process.

The design phase produces submodels composed of feature subsets and the corresponding classifiers for each subset. During the ACE runtime, the first submodel, associated with the initial and smallest feature set, classifies one data window and reports the confidence level of its outcome. If the confidence meets the predefined threshold, it outputs the result. Otherwise, the process proceeds with the next submodel until either the confidence threshold is satisfied or the last submodel is reached. During the iterative process when the application utilizes more complicated feature sets, ACE buffers reusable variables to minimize re-computation to mitigate unnecessary runtime and energy consumption. In the following sections, we detail the submodel generation and data buffering strategies in the iterative process used by ACE.

[1]The tool is open-sourced at https://github.com/yuxwang99/ACE-adapt.git.

TABLE I
COMPARISON WITH STATE-OF-THE-ART METHODOLOGIES FOR BALANCING RUNTIME SAVINGS AND ML PERFORMANCE

Ref	Apps	Data	Win. len	Edge. proc	Feature/ML selection	Selection level	FoM	
							Runtime saving	ML loss
[21]	Seizure detection	EEG	1.17s	No	Offline	-	-	
[28]	Seizure detection	ECG	100 RR intervals	No	Online	2	16.80%	1.85%(gmean)
[27]	Seizure detection	ECG, SpO2	120 s	No	Online	Any	6.1%	1%(F-1 score)
ACE	Seizure detection↑	ECG, SpO2	120 s	Yes	Online	Any	35.7%	1%(F-1 score)
[23]	CW - ESC	SKT, RSP ECG, PPG	60 s	No	Offline	-	-	
ACE	CW - ESC↑	SKT, RSP ECG, PPG	60 s	Yes	Online	Any	49%	1%(F-1 score)

CW: Cognitive Workload; ESC: emotional state classification; EEG: electroencephalogram; SKT: skin temperature; RSP: respiration; ECG: electrocardiogram; PPG: photoplethysmogram.
↑ ACE takes algorithms from the previous row as inputs, and applies runtime optimization to them.

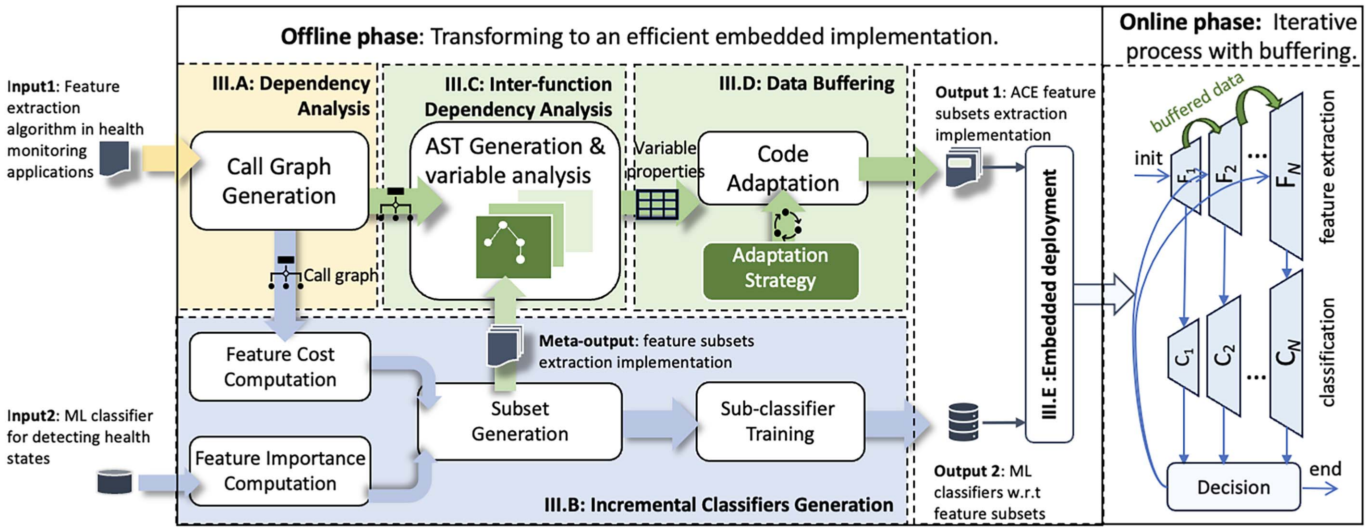

Offline phase: Transforming to an efficient embedded implementation.
Online phase: Iterative process with buffering.
Input1: Feature extraction algorithm in health monitoring applications
III.A: Dependency Analysis
Call Graph Generation
Call graph
III.C: Inter-function Dependency Analysis
AST Generation & variable analysis
III.D: Data Buffering
Variable properties
Code Adaptation
Adaptation Strategy
Output 1: ACE feature subsets extraction implementation
III.E :Embedded deployment
Meta-output: feature subsets extraction implementation
Feature Cost Computation
Subset Generation
Sub-classifier Training
Feature Importance Computation
III.B: Incremental Classifiers Generation
Input2: ML classifier for detecting health states
Output 2: ML classifiers w.r.t feature subsets
buffered data
init
F1
F2
FN
feature extraction
C1
C2
CN
classification
Decision
end

Fig. 1. Workflow for ACE transformation of monolithic health monitoring applications into multi-feature-multi-classifier models with buffers and runtime deployment.

A. Dependency Analysis

The function dependency of feature extraction is depicted through its call graph, which is a directed acyclic graph (DAG) that comprises vertices and edges, denoted as an aggregate (V, E). Each vertex represents a function, and the direction of the edges from one vertex to another represents the invocation relationships between callers and callees.

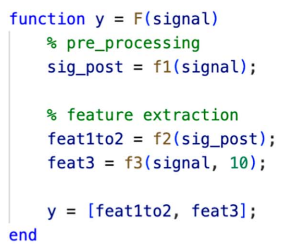
In the first stage of the conversion workflow shown in Fig. 1, the call graph is constructed from top to bottom to perform dependency analysis. A code examiner identifies functions invocations from the root function and employs a depth-first traversal to construct the call graph.

Regular expressions (RE) are utilized to scrutinize function invocation while processing source code statements. The RE identifies whether a sequence of characters specifies a match pattern in the text that follows the function invocation syntax of the high-level programming languages. Once a successful match is identified, the examiner creates a new vertex in the call graph, and a connection is established between the parent and the child function. The call graph is updated as follows:

$$V := \{u \cup V\}$$
$$E := \{v \rightarrow u \cup E\}$$

where v is the caller (the current node) and u is the callee.

Following the depth-first rules, the examiner further steps into the callee to determine its dependencies on other functions before returning to the current node. This recursive process automatically constructs the call graph, which provides information to analyze the feature computation pattern within the codebase. Fig. 2 illustrates the call graph generation results for a sample feature extraction codebase. The code snippet in Fig. 2(a) is the implementation of the root function F that extracts 3 features with a processing function and other callees to compute the subsets of the monolithic feature sets. Fig. 2(b)

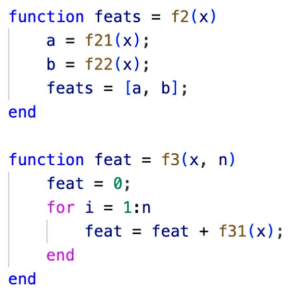


```
function y = F(signal)
    % pre_processing
    sig_post = f1(signal);

    % feature extraction
    feat1to2 = f2(sig_post);
    feat3 = f3(signal, 10);

    y = [feat1to2, feat3];
end
```

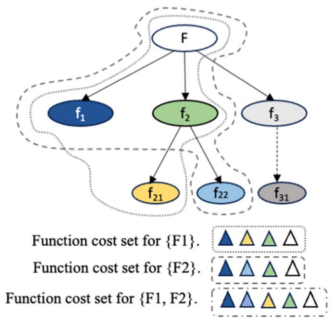
(a) Root function for extracting 3 features



```
function feats = f2(x)
    a = f21(x);
    b = f22(x);
    feats = [a, b];
end

function feat = f3(x, n)
    feat = 0;
    for i = 1:n
        feat = feat + f31(x);
    end
end
```

(b) Callee functions of the root for computing specific features


F
f1
f2
f3
f21
f22
f31
Function cost set for {F1}.
Function cost set for {F2}.
Function cost set for {F1, F2}.

Leaf node	Dependent features
f_1	[1,1,0]
f_{21}	[1,0,0]
f_{22}	[0,1,0]
f_{31}	[0,0,1]

(c) left: Call graph of the feature extraction function, with the table showing the dependencies between the leaf nodes and the extracted features; right: the aggregate sum for computing the cost of specific feature sets

Fig. 2. Dependency analysis on feature extraction code.

shows the callees of F at depth one that utilizes functions to compute specific features, recursively connecting to their respective callees, completing the call graph as shown in Fig. 2(c). As shown in in Fig. 1, we discuss the key role of the call graph for submodel generation and inter-function dependency analysis in the following sections respectively.

B. Incremental Classifier Generation

In the submodel generation phase, feature subsets of the monolithic model are selected and classifiers are trained based on these subsets. In this section, we introduce the generation of submodels in increasing complexity based on the analysis of the ML significance and computation cost of the submodel. In addition, we present how to evaluate these metrics by using the call graph generated during dependency analysis.

1) Feature Set Cost: The feature set cost refers to the execution time required when computing all the features within the sets. The execution time of the algorithm depends on both the language used for implementation and the deployment platform. The high-level monolithic model is first converted to C for profiling on a personal computer (PC) to prevent compilation variations like just-in-time compilation and pre-compilation used in high-level languages. These profiling differences varying in languages and platforms are analyzed in Section V. The feature cost computation block automatically manages the profiling process, calculating the execution time for each function. The runtime metrics are then recorded and normalized to the full monolithic model's runtime, mitigating the discrepancies across platform performance.

Furthermore, the profiling results are automatically mapped to the call graph to analyze the cost of a function invocation, where the execution time of each node excludes its descendants but includes its own arithmetic computing and control logic. As illustrated in the table of Fig. 2(c), a binary vector annotates the correlation between each leaf node and monolithic feature sets indicating whether the leaf node is involved in producing any of the features. For the example in Fig. 2(c), f_1 is marked with vector $\{1,1,0\}$ indicating it is required for the computation of the first two features.

With the annotation on the leaf nodes, the cost of a feature set is computed by its dependent nodes from bottom to top using the obtained call graph. As depicted in Fig. 2(c), the cost of the set that only includes feature 1, {F1}, is determined by adding the costs of all the nodes on which the function depends. This computation starts from the leaf node explicitly marked for {F1}, e.g., f_1 and f_{21}, and backtracks to its ancestors until it reaches the root node. The dotted lines in the call graph shown in Fig. 2(c) represent the aggregate set of functions required to execute for computing {F1}. Similarly, the cost of computing {F2} is the sum of the execution time of functions in the dashed blocks that include f_1, F, f_2, and f_{22}. Moreover, the cost of any combination of feature sets can be computed by adding the costs of the nodes in a subgraph built by backtracking the dependent leaf nodes in the call graph.

2) Feature Set Importance: The feature importance is a measurement of the contribution of the input features to the estimation outcome. There are various metrics to evaluate the importance score [29], [30], whose choice is orthogonal with respect to the ACE methodology. Statically, filter methods [31] use proxy measures to capture the usefulness of the feature set. Common model-agnostic measures include mutual information, Pearson product-moment correlation coefficient, the scores of significance tests for each feature combination, and Shapley additive explanations [29].

In this work, we apply ACE on two benchmarks employing bagged Decision Trees (DT) as ML classifiers, hence we utilize the commonly-used Gini impurity [32] to measure the feature contribution on the DT splits. The importance of one feature set is equal to the sum of feature importances among all its constituent feature elements. To showcase the generality of choice of the classifiers and the feature importance metric, we also consider correlation coefficients, which can be obtained regardless of the choice of ML classifier, in Section V-E.

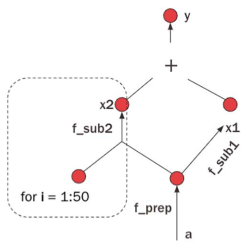
3) Generation of Feature Subsets and Classifiers: For an ML classification task with N input features, the number of possible feature sets is $\mathbf{O}(2^N)$. To constrain this exponential design space, the submodel candidates are generated according to their costs. Specifically, we include a new feature in the incremental order of computation cost to confine the exploration space to $\mathbf{O}(n)$ complexity. The feature sets are clustered based on their importance and computation cost into n groups. Within each cluster, the sets are ranked based on their importance, and only the set with the highest importance score is retained.

```
function y = f(a)
    a = f_prep(a);
    x1 = f_sub1(a);

    for i = 1:50
        x2 = f_sub2(a,i);
    end

    y = x1 + x2;
end
```

(a) Code snippet in Matlab



(b) AST representation

```
function y = f(a)
    global x1 ctrl_vec;
    a = f_prep(a);
    if ctrl_vec(get_var_index('x1'))
        x1 = f_sub1(a);
        ctrl_vec(get_var_index('x1')) = 0;
    end
    for i = 1:50
        x2 = f_sub2(a,i);
    end
    y = x1 + x2;
end
```

(c) Automatically generated code snippet with data buffering

Fig. 3. An example code snippet and the corresponding generated AST for variable buffering analysis.

Furthermore, the new DTs are trained using the incremental feature sets for the health monitoring application.

4) Meta Output: The incremental classifier generation block outputs a meta-output as shown in Fig. 1. The meta-output extracts feature subsets based on the iteration times, which mirrors the structure of the monolithic feature extraction code. The execution of feature extraction corresponds to a sub-graph of the produced call graph. As explained in Section III-B3, the generated feature sets exhibit a hierarchical structure, where smaller sets are subsets of larger ones, thereby the larger ones always encompass the subgraphs of smaller sets executed. This hierarchy arises as new input features are incrementally incorporated. During the iteration of computing the next feature subsets, recomputing the same data in the extraction process leads to unnecessary consumption of computational resources. However, the call graph cannot identify the data to buffer, as the invocation relies on input data that is not depicted within the call graph. For example, $feat$ in $f3$ cannot be directly buffered as it is updated in the loop. Moreover, buffering only the leaf nodes is not efficient. For example, if feature extraction for {F1, F2} is needed at the end of extraction of {F1}, $f1$ would be invoked again with the same input data to compute *sig_post* as only {F1} is buffered. Therefore, to determine what needs to be buffered, an analysis of how variables are computed across function calls is necessary.

C. Inter-Function Dependency Analysis

The inter-function dependency analysis block processes the meta-output in a level-order traversal of its call graph which is the same as the monolithic model. The parsing algorithm automatically analyzes each variable production from its generation statement in the meta-output, providing information to decide whether buffering the variable would enhance runtime efficiency. If the variable produced by a callee has been buffered, the execution time is reduced by avoiding the function invocation in the call graph.

Each function implementation can be represented as an abstract syntax tree (AST), which captures the syntactic structure of the source code [33]. The nodes in AST denote a construction from a statement in the source code. Descendants of a node abstract its computation, while ancestors illustrate how it has been utilized. Each code statement serves either a control purpose, declaring its effective scope, or a computation purpose, assigning a value to a variable. To construct the AST, the parser employs RE to process each statement that either converts a statement into control scopes or creates nodes to denote newly generated variables, connecting them to their respective producer nodes. Fig. 3 illustrates an example of constructing an AST by parsing a Matlab code snippet. In the representation, the nodes symbolize variables and the connections denote how they are computed. The scope of the control clause, such as the function declaration, is pushed onto a stack when parsed. Each time a new node is constructed, its effective scope is marked by the top scope in the aforementioned stack.

The effective scope within the control clause is identified by keywords such as **for**, **while**, and **if**. Each abstract block corresponds to a control clause based on the syntax of the high-level language, and this abstraction is attached to the variables to annotate its effective scope. The scope can be nested, with the resulting variable annotated only with its current scope. Nested scopes can be recursively tracked along with their operators. The production on the right-hand side of the expression describes how the value is assigned to the left-hand side. In the AST, descendant nodes represent operators that generate the node. The marked generation ways indicate how it is computed, such as through arithmetic computation or by returning a value from a child function. In the code parsing process to construct the AST, we use the index of variables as a reference, delineating the problem of namesake, where the identifiers share the same name.

In the next step of code analysis, the child function *f_prep* and its child function *f_sub1* are parsed accordingly. The parsing results then provide the information on analyzing variable selection for buffering.

D. Data Buffering

In this section, we propose an automatic buffering strategy and generate a new implementation with variable buffering to circumvent the need for re-computation. The meta-output code is automatically adapted to a new code base without affecting the original functionality but achieving higher efficiency. We first define our strategy of selecting variables to buffer based

TABLE II
VARIABLE ATTRIBUTES FOR BUFFERED VARIABLE SELECTION

Notation	Identifiers	Usage	Block	Generation
#0	a	#1	top func	(input)
#1	a	#2, #4	top func	f_prep
#2	x_1	#4	top func	f_sub1
#3	i	#4	for loop	iterator
#4	x_2	#5	for loop	f_sub2
#5	y	/	top func	(output)

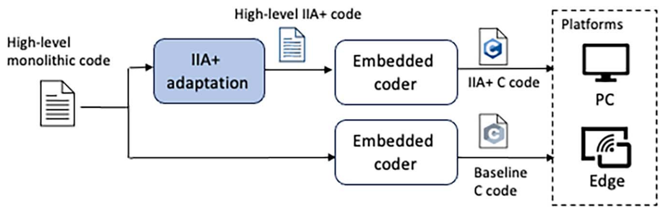


Fig. 4. Deployment of the monolithic and ACE models.

on the generated ASTs for each function and attributes obtained from the parser. The attributes of the variables in Fig. 3, outlined in the columns of Table II, include the identifiers of the variables, their use by the consumer functions, their effective block, and the generation method. All the attributes serve as analytical criteria for buffered variable selection. The selection rules for buffering are listed as follows:

1) The variable's name is not shared, to avoid ambiguity.
2) The variable and its descendants do not belong to a loop block where their values are updated in the loop.
3) The variable is consumed by other variables, which are listed in the "usage" column.
4) The variable is generated by user-defined functions to avoid abuse of access to edit built-in functions.
5) The variable is neither an input nor an output of the function. It is inefficient to buffer such parameters as they can be easily accessed directly from the parent function.

Table II lists the variable attributes corresponding to Fig. 3. Following the buffering strategy, we exclude variables #0 and #1 due to their shared name. Similarly, variables #3 and #4 are excluded as they are redefined within the loop. Additionally, variable #5 is excluded since it represents the output and is not consumed by other variables within the function scope. The variable that satisfies the rules and would be buffered is #2. For each function, we create a storage list that manages the variables to be buffered. This list examines each variable according to the buffering strategy and includes the variable if it meets the defined criteria.

The production statements for the variables in the storage list are automatically rewritten, and an access vector takes control of the computation of each variable. If the computation has not been performed yet, the control vector allows the computation to take place and then disables the write access to avoid re-computation. When the write access is disabled, the function can directly access the existing variable and utilize the computed value. At the end of code adaptation when all new scripts are generated, an additional function is generated to control the read and write permission of all buffered variables. Fig. 3(c) illustrates the generated new code that performs the same functionality as the original code snippet in Fig. 3(a). This function returns the index of the buffered variable based on the input string, which is formed by concatenating the function name and the unique identifiers of the variables.

After the computation, the control bit corresponding to the variable is set to 0, indicating that the value is buffered through the global declaration and no further computation is needed for the same input. Upon completion of the iteration process, the system resets the access control vector to its default value of one, computing the feature sets for newly captured data.

E. Embedded Device Deployment

To deploy the high-level health monitoring algorithm on embedded devices, we use the MATLAB coder to convert the proposed algorithm from MATLAB to C. We generate a standalone C codebase from the original code that is not dependent on specific libraries, software, and platforms. The automatically generated codebase only requires a C compiler for building the project, and the resulting executable program can run on both PCs and general embedded devices, provided that the memory requirements of the target application are met. To speed up the execution on edge, we further adapt the floating point representation from 64-bit (FP64) to 32-bit (FP32) without any Quality of Service (QoS) degradation in terms of F-1 score, achieving a 25% reduction in run-time.

IV. EXPERIMENTAL SETUP

Fig. 4 plots the experimental setup for the model deployment pipeline. We first convert the monolithic model with ACE adaptation in MATLAB automatically using the methodology described in Section III. Then, the ACE adaptation and the original algorithm of the health monitoring application are converted to C using the MATLAB embedded coder.

A. Benchmark Applications

We employ two complex health monitoring applications as listed in Table I. More details of the algorithm implementation and dataset are described in the following:

Seizure detection: The seizure detection (SD) [27] algorithm takes 256 Hz ECG and SpO2 signals as inputs. The baseline [27] application includes a pre-processing phase with a band-pass filter that removes the noise components, then the feature extraction phase extracts 17 features. The monolithic feature set consists of 15 features extracted from the ECG signals and 2 features are instead derived from the SpO2 signal. A binary DT classifier is trained to distinguish between seizure and non-seizure states based on the feature set.

The binary classification dataset contains 90 seizure windows and 90 non-seizure windows, each 120 seconds long. The dataset is split into 80% for training, 10% for validation, and 10% for testing. More details can be found in [27].

Emotional state classification: The second benchmark performs emotional state classification (ESC) [23] using a

60-second window of four captured biosignals: ECG and respiration, sampled at 256Hz, and skin temperature and electrodermal activity, sampled at 4Hz. The optimal monolithic feature set uses 45 features extracted from the time and frequency domains. A multi-class DT classifier is applied for three-types classification.

The ESC dataset captures data are collected from 18 volunteers. Each recording session spans 20 minutes and includes subjects' cognitive, emotional, and neutral states, as reported in [24]. The datasets are split into training and testing sets using an 80% / 20% ratio.

B. Evaluation Support

Platforms: To test the generalization of the low-level implementation, the algorithms are deployed on two platforms that support FP computation: 1) Apple M1 chip (A-M1) on a PC operating at 3.2 GHz, with more powerful computation capabilities compared to edge devices, 2) ARM Cortex-A9 (C-A9) processor, running at 667 MHz, a widely adopted edge platform. We demonstrate that profiling can be extended from a PC to a embedded device to achieve a comparable speedup, despite a significant disparity in computational capabilities.

Metrics: We evaluate the runtime across various platforms to test the performance of the proposed methodology. The benchmark runtime is averaged among all inputs in the dataset to evaluate the overall speedup. Moreover, the runtime is normalized to that of the monolithic model for comparison across different platforms.

We use a weighted F-1 score to evaluate the classification performance of the ML models for binary and ternary classification tasks. The weighted F-1 is computed using precision and recall as follows:

$$\text{precision} = \frac{TP}{TP + FP} \tag{1}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{2}$$

$$\text{F-1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{3}$$

$$\text{F-1\_weighted} = \sum_{i=1}^{K} w_i \cdot \text{F-1}_i \tag{4}$$

where TP represents the sum of true positive samples, FP and FN denote incorrect positive or negative classifications, K denotes the number of classes, and w_i represents the proportion of samples belonging to the class i.

Baselines: We use the metrics evaluated by monolithic models as a baseline, showcasing the overall enhancement achieved through the optimization methodology.

Moreover, we compare the final feature extraction output to the meta-output for subset generation, as shown in Fig. 1. The comparison results are used to analyze the efficiency of the buffering, as the meta-output extracts feature subsets without buffering computation-intensive data. The call graphs of the meta-output are plotted in Fig. 5. Corresponding details about the vertices and edges, as well as the code complexities, are listed in Table III. These two benchmarks are illustrative examples to represent the code complexity of a general health monitoring application using our automated tools.

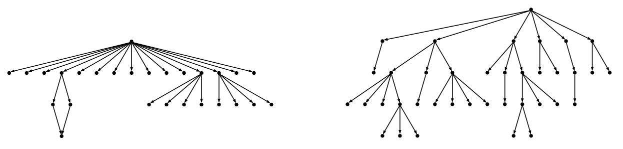

(a) Call graph for SD (b) Call Graph for ESC

Fig. 5. Benchmarks call graphs.

TABLE III
BENCHMARKS SIZE AND COMPLEXITY

Benchmarks	Code Base		Call Graph	
	Text Size (Bytes)	Lines	Vertices	Edges
SD	32514	1013	26	27
ESC	56727	1866	40	37

Configurations: We evaluate our methodology under different configurations that can affect overall system efficiency and ML performance. The configurations vary in two factors: the number of submodels, which determines the maximum number of iterative steps, and the confidence threshold, which controls the termination criteria for the iterative process.

V. Results

We describe the results obtained by applying ACE on the two benchmarks in this section. We further analyze the performance of the runtime efficiency and ML accuracy under various configurations across platforms.

A. Submodel Runtime

We generate the different numbers of submodels for iterative monitoring. Fig. 6 plots the submodel runtime normalized to the runtime of the monolithic model. Each column represents a selection of different submodels in the configurations, with submodels indexed from bottom to top for iterative monitoring. The generation algorithm ensures the final model is the monolithic model regardless of the submodel numbers. The model generation is based on the feature costs that are profiled with gprof [34] on the PC. The difference in normalized runtime of submodels across the various platforms is within 3% for SD, and less than 2% for ESC. However, we find that the runtime variance can reach up to 30% utilizing profiling metrics from MATLAB. This validates our proposed methodology, indicating that profiling should be performed in a low-level language to avoid performance variation of computing capabilities across platforms. Based on the generation results of the PC metrics, the relative submodel costs can be replicated when deploying the benchmarks on embedded devices.

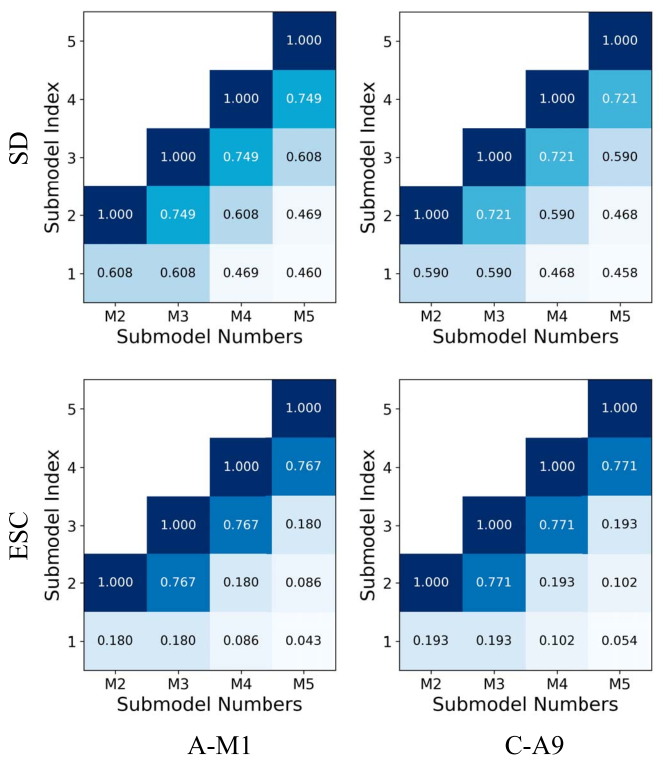


Fig. 6. Normalized submodel runtime in configurations with different numbers of submodels (M2-M5) on various platforms and benchmarks.

The submodel generation effectively partitions the computation of feature sets. The first submodel of the 5-model configuration comprises 45.98% of the monolithic runtime in SD, while only consuming 4.31% in ESC. This discrepancy can be attributed to the prominence of frequency domain features dominating the feature extraction phase of ESC; submodels 1 through 3 in M5 exclude these features. The generated submodel runtime is hence application-dependent, as it correlates with the complexities of the features and their grouping into sets. Nonetheless, results show that ACE can, in practice, effectively partition computation across submodels.

B. Trade-Off: Runtime vs. F-1

To explore the effect of the configuration on the system performance, we tested various generated configurations with confidence thresholds ranging from 0.5 to 0.8. Fig. 7 plots both the normalized runtime and F-1 score of each combination of model configuration and confidence threshold. For example, a configuration of 3 submodels in the system using a confidence threshold of 0.6 is denoted as M3-0.6. The runtime values are normalized relative to the average runtime of monolithic models on the targeted platforms across all test inputs. The solid bar represents the normalized runtime of ACE, while the total bar height represents the normalized runtime with the deployment of the meta-output. The transparent portion of the bar illustrates the runtime speedup achieved through the ACE buffering methodology.

The monolithic model achieves an F-1 score of 66.67% on SD, and for ESC, the baseline F-1 score is 70.54%. For SD, F-1 scores using M5-0.7 and M5-0.8 are 72.71%, which is higher than the baseline. This is because F-1$_{M1|conf>0.8}$ > F-1$_{M5|conf>0.8}$, where under the condition of confidence higher than 0.8, the F-1 of the first submodel is higher than the monolithic model. In this case, the seizure detection process terminates after the first iteration, causing the obtained F-1 score higher than the monolithic baseline. The ML performance does not vary across platforms as long as the processors support FP32 computation.

As expected, a high F-1 score is obtained by using high confidence thresholds. Nonetheless, in most cases a confidence threshold of 0.6 suffice to achieve comparable F-1 scores with respect to monolithic models. A higher confidence threshold results in a stricter condition to terminate the iterative process, increasing the possibility that a more complex submodel is involved which prolongs the runtime. ACE runtime is very close to the baseline using 0.8 confidence. However, with a confidence of 0.6, it achieves an average speedup of 30% to 40%. Confidence 0.6 is a critical point for most cases, where under the configurations of confidence lower than 0.6, the ML performance degrades by 29.4% for SD and 28.7% for ESC. The complexities of the submodels are low, enabling high speedup, but a low confidence level can compromise the system's QoS (e.g. F1 score).

Without buffering, increasing the threshold from 0.6 to 0.7 significantly increases the runtime beyond that of the baseline monolithic model. On the other hand, a few configurations lead to a shorter runtime below the baseline. Therefore, the ACE buffering strategy significantly reduces the runtime in the high confidence threshold cases, where almost all configurations run faster than the monolithic model.

C. Runtime Composition Analysis

Fig. 8 showcases the runtime decomposition of M-5 tested on C-A9, which illustrates the composition of the iterative detection process using all five models. Feature extraction using the first submodel is executed for all test samples which accounts for 48.3% of the runtime with confidence 0.8 and 76.7% of the runtime with confidence 0.5. Compared to the feature extraction time, the classification time plotted in orange takes on a small ratio in the iteration process. This explains our motivation to buffer variables in the feature extraction process for speedup as feature extraction dominates the runtime. Our experiments show that classification accounts for 1.77% at most in the detection for all SD submodels, and 1.1% for ESC.

As the confidence threshold increases, the feature extraction time does not significantly increase even as more complex computation is involved in the latter submodels. This is due to the early exiting of the detection when confidence threshold is met and reused data is buffered.

D. Comparison Between PC and Edge Device

Table IV summarizes the deployment analysis across platforms, including physical runtime and data buffering size which

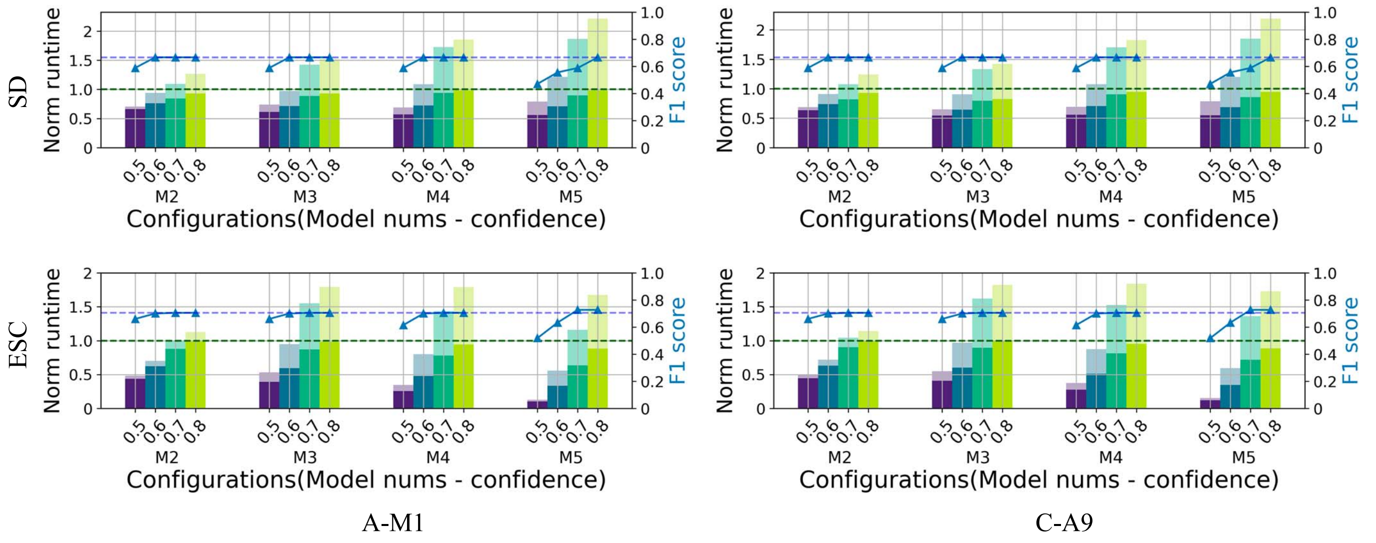


Fig. 7. Normalized runtime vs F-1 score under different configurations tested on various platforms. The results for each combination of model configuration and confidence threshold are shown.

TABLE IV
DEPLOYMENT ANALYSIS WITH MAXIMUM SPEEDUP

Applications	Platforms	Buffering size (kB)	Monolithic runtime (s)	Max speedup - meta-output		Max speedup - ACE	
				0% loss	1% loss	0% loss	1% loss
SD	A-M1 C-A9	122.89	0.077 2.195	6.1% 9.2%	6.1% 9.2%	**28.9%** **35.7%**	**28.9%** **35.7%**
ESC	A-M1 C-A9	11.95	0.078 2.711	-7.5% -0.1%	29.7% 28.3%	**22.0%** **18.9%**	**52.3%** **49.0%**

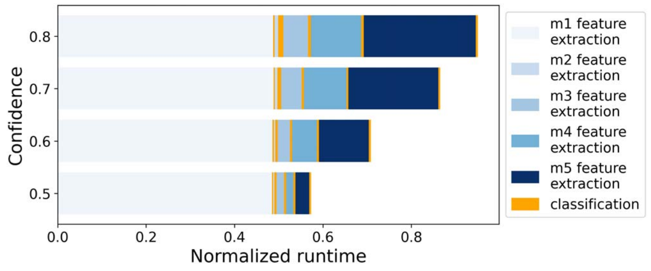


Fig. 8. Runtime decomposition of SD on C-A9, where the orange bars indicate the classification time and blue shows feature extraction time using different feature subsets.

is the sum of the data size for variables that are marked as global. ACE buffers 122.89 kB data on SD and only 11.95 kB on ESC. It highlights the highest speedup achieved without and with a limited F-1 loss (1%) among all tested configurations utilizing ACE, both with and without buffering.

In the case of SD, the maximum speedup for 0% loss is performed under the configuration M3-0.6. ACE methodology achieves 28.9% and 35.7% speedups using M1 chip and C-A9 respectively, running over 20% faster with buffering. For ESC, configuration M4-0.7 achieves 22.0% speedup on M1 chip and 18.9% on C-A9 without any F-1 degradation. For a 1% F-1 degradation tolerance, configuration M4-0.6 enhances ESC performance by up to 52.3% across platforms, achieving an F-1 score of 70.04%. which is marginally lower than the baseline of 70.54%.

Table IV records absolute monolithic runtime on different platforms. Though the absolute runtime of M1 is over 30 times faster than the C-A9, the normalized acceleration results exhibit similar trends as illustrated in Fig. 7. This suggests that the deployment of low-level implementation generated by ACE does not experience significant variations in runtime efficiency on embedded devices.

E. Generality of ACE Adaptation

While the benchmark applications described in Section IV-A chose tree-based classifiers to maximize ML performance for their specific tasks, ACE is not restricted to specific biomedical applications or categories of classifiers. ACE supports submodel generation as long as a measure of feature importance score can be provided, as explained in Section III-B2. To explore the generality of the ACE, we conduct the same evaluation experiment introduced in Section V-B for ESC on C-A9 using correlation coefficients as importance scores. We further train a SVM classifier for the monolithic model and submodels to explore the effect of classifier and importance score selection.

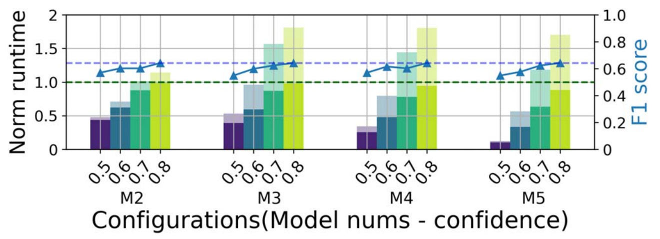


Fig. 9. Normalized runtime vs F-1 score of ESC on C-A9 using correlation coefficients as the chosen importance score for a SVM classifier.

Fig. 9 shows the trade-off between runtime savings and confidence scores under such settings.

We can observe similar trends in both the runtime savings and F-1 score increase for different model configurations as in Fig. 7. The runtime savings under this SVM-based configuration do not exhibit large discrepancies from the results in Section V-B, as the costs of submodels generated by the correlation coefficients exhibit a 1% deviation from those presented in Section V-A. The baseline F-1 score of the monolithic model is 64.2% as opposed to 70.54% using the DT classifier. Maintaining a confidence level greater than 0.8 ensures no loss in the F-1 score, which is a stricter threshold compared to DTs, because tree-based models are better suited for the particular classification task.

The model weights of the submodels for the iterative process are embedded in the generated C code, adding an additional memory burden. With 5 models in ESC, the first four submodels account for 14.21% of the total code size, whereas four SVM models only contribute 0.22%, making it lighter for edge deployment.

VI. Conclusion

In this paper, we presented the ACE methodology for automatically transforming any health monitoring application into an efficient iterative implementation, achieved by buffering computation-intensive data and deploying the optimized version on edge devices. During the offline phase, the system generates submodels operating at various complexities and identifies shared variables among these submodels. In the online phase, the optimized application starts with the simplest submodel and involves more complex ones until the confidence meets the pre-defined threshold. This process avoids recomputation of computation-intensive data with buffering.

To demonstrate the effectiveness of the proposed methodology, we compared the optimized model with the original monolithic model using two biomedical benchmarks and tested the runtime on various platforms. The experimental results showed that without any F-1 score loss, ACE achieves 35.7% and 18.9% speedup using an ARM Cortex-A9 processor for seizure detection and emotion classification, respectively.

In conclusion, our methodology effectively decomposes the complexity of a monolithic algorithm, resulting in significant speedup for general health monitoring applications on an edge device. The adaptation improves the usability of general state-of-the-art algorithms operating with feature extraction and ML classification, thus enabling developers of biomedical algorithms to seamlessly and efficiently deploy them on edge.

References

[1] S. M. Iqbal, I. Mahgoub, E. Du, M. A. Leavitt, and W. Asghar, “Advances in healthcare wearable devices,” *NPJ Flexible Electron.*, vol. 5, no. 1, 2021, Art. no. 9.

[2] D. Sopic, A. Aminifar, and D. Atienza, “e-Glass: A wearable system for real-time detection of epileptic seizures,” in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2018, pp. 1–5.

[3] A. T. Tzallas, M. G. Tsipouras, and D. I. Fotiadis, “Epileptic seizure detection in EEGs using time–frequency analysis,” *IEEE Trans. Inf. Technol. Biomed.*, vol. 13, no. 5, pp. 703–710, Sep. 2009.

[4] M. K. Siddiqui, R. Morales-Menendez, X. Huang, and N. Hussain, “A review of epileptic seizure detection using machine learning classifiers,” *Brain Inform.*, vol. 7, no. 1, 2020, Art. no. 5.

[5] R. Varatharajan, G. Manogaran, M. K. Priyan, and R. Sundarasekar, “Wearable sensor devices for early detection of alzheimer disease using dynamic time warping algorithm,” *Cluster Comput.*, vol. 21, pp. 681–690, Mar. 2018.

[6] L. Orlandic, J. Thevenot, T. Teijeiro, and D. Atienza, “A multimodal dataset for automatic edge-AI cough detection,” in *Proc. 45th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (EMBC)*, Sydney, Australia. Piscataway, NJ, USA: IEEE Press, Jul. 2023, pp. 1–7.

[7] P. Mach and Z. Becvar, “Mobile edge computing: A survey on architecture and computation offloading,” *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart. 2017.

[8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[9] Z. Liu, J. Wang, and B. Liu, “ECG signal denoising based on morphological filtering,” in *Proc. 5th Int. Conf. Bioinf. Biomed. Eng.*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 1–4.

[10] A. Kawala-Sterniuk et al., “Comparison of smoothing filters in analysis of EEG data for the medical diagnostics purposes,” *Sensors*, vol. 20, no. 3, 2020, Art. no. 807.

[11] P. Melillo, M. Bracale, and L. Pecchia, “Nonlinear heart rate variability features for real-life stress detection. Case study: Students under stress due to university examination,” *Biomed. Eng. Online*, vol. 10, Nov. 2011, Art. no. 96.

[12] M. Omidvar, A. Zahedi, and H. Bakhshi, “EEG signal processing for epilepsy seizure detection using 5-level Db4 discrete wavelet transform, GA-based feature selection and ANN/SVM classifiers,” *J. Ambient Intell. Humanized Comput.*, vol. 12, pp. 1–9, Nov. 2021.

[13] M. Habib, Z. Wang, S. Qiu, H. Zhao, and A. S. Murthy, “Machine learning based healthcare system for investigating the association between depression and quality of life,” *IEEE J. Biomed. Health Inform.*, vol. 26, no. 5, pp. 2008–2019, 2022.

[14] H. Hoffmann, A. Jantsch, and N. D. Dutt, “Embodied self-aware computing systems,” *Proc. IEEE*, vol. 108, no. 7, pp. 1027–1046, 2020.

[15] E. H. Hafshejani et al., “Self-aware data processing for power saving in resource-constrained IoT cyber-physical systems,” *IEEE Sensors J.*, vol. 22, no. 4, pp. 3648–3659, Feb. 2022.

[16] T. K. Ho, J. Hull, and S. Srihari, “Decision combination in multiple classifier systems,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 16, no. 1, pp. 66–75, Jan. 1994.

[17] H. A. Kholerdi, N. TaheriNejad, and A. Jantsch, “Enhancement of classification of small data sets using self-awareness—An Iris flower case-study,” in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2018, pp. 1–5.

[18] K. Neshatpour, F. Behnia, H. Homayoun, and A. Sasan, “ICNN: An iterative implementation of convolutional neural networks to enable energy and computational complexity aware dynamic approximation,” in *Proc. Des., Automat. Test Europe Conf. Exhib. (DATE)*, 2018, pp. 551–556.

[19] S. Benatti et al., “A versatile embedded platform for EMG acquisition and gesture recognition,” *IEEE Trans. Biomed. Circuits Syst.*, vol. 9, no. 5, pp. 620–630, Oct. 2015.

[20] Y.-Y. Hsieh, Y.-C. Lin, and C.-H. Yang, “A 96.2nJ/class neural signal processor with adaptable intelligence for seizure prediction,” in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, vol. 65, 2022, pp. 1–3.

[21] V. Pandya, U. P. Shukla, and A. M. Joshi, “Novel features extraction from EEG signals for epilepsy detection using machine learning model,” *IEEE Sensors Lett.*, vol. 7, no. 10, pp. 1–4, Oct. 2023.

[22] Y. Ye, R. Zhang, W. Zheng, S. Liu, and F. Zhou, "RIFS: A randomly restarted incremental feature selection algorithm," *Sci. Rep.*, vol. 7, no. 1, 2017, Art. no. 13013.

[23] L. Orlandic, A. A. Valdes, and D. Atienza, "Wearable and continuous prediction of passage of time perception for monitoring mental health," in *Proc. IEEE 34th Int. Symp. Comput.-Based Med. Syst. (CBMS)*, 2021, pp. 444–449.

[24] F. Dell'Agnola, U. Pale, R. Marino, A. Arza, and D. Atienza, "Mbiotracker: Multimodal self-aware bio-monitoring wearable system for online workload detection," *IEEE Trans. Biomed. Circuits Syst.*, vol. 15, no. 5, pp. 994–1007, Oct. 2021.

[25] F. Forooghifar et al., "Self-aware anomaly-detection for epilepsy monitoring on low-power wearable electrocardiographic devices," in *Proc. IEEE 3rd Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, 2021, pp. 1–4.

[26] A. Anzanpour et al., "Self-awareness in remote health monitoring systems using wearable electronics," in *Proc. Des., Automat. Test Eur. Conf. Exhib. (DATE)*, 2017, pp. 1056–1061.

[27] L. Ferretti et al., "INCLASS: Incremental classification strategy for self-aware epileptic seizure detection," in *Proc. Des., Automat. Test Eur. Conf. Exhib. (DATE)*, 2022, pp. 1449–1454.

[28] F. Forooghifar, A. Aminifar, and D. Atienza Alonso, "Self-aware wearable systems in epileptic seizure detection," in *Proc. 21st Euromicro Conf. Digit. Syst. Des. (DSD)*, 2018, pp. 426–432.

[29] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 4768–4777.

[30] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu, "Definitions, methods, and applications in interpretable machine learning," *Proc. Nat. Acad. Sci.*, vol. 116, no. 44, pp. 22071–22080, 2019.

[31] N. Sánchez-Maroño, A. Alonso-Betanzos, and M. Tombilla-Sanromán, "Filter methods for feature selection—A comparative study," in *Proc. 8th Int. Conf. Intell. Data Eng. Automated Learn. (IDEAL)*, Birmingham, UK, 2007. Berlin, Heidelberg: Springer-Verlag, 2023, pp. 178–187

[32] Y. Qi, "Random forest for bioinformatics," 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:1117445

[33] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2006.

[34] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proc. SIGPLAN Symp. Compiler Construction (SIGPLAN '82)*, New York, NY, USA: ACM, 1982, pp. 120–126.

Yuxuan Wang received the bachelor's degree from the University of Electronic Science and Technology of China (UESTC) and the master's degree from the École Polytechnique Fédérale (EPFL), Switzerland, both in electrical engineering. She is currently working toward the Ph.D. degree and serving as a Doctoral Assistant with the Embedded Systems Laboratory (ESL), EPFL. Her research interests include the intersection of energy-efficient edge computing systems and applied machine learning.

Lara Orlandic received the bachelor's degree from Georgia Institute of Technology, USA, and the master's degree from the École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, both in electrical engineering. She is currently working toward the Ph.D. degree with the Embedded Systems Laboratory, EPFL, Switzerland. Her research interests include software-hardware co-design for biomedical signal processing and edge artificial intelligence.

Simone Machetti received the master's (*Summa Cum Laude*) degree in computer engineering, with a major in embedded systems, from the Politecnico di Torino, Italy. He is currently working toward the Ph.D. degree with the Embedded Systems Laboratory (ESL), EPFL, Switzerland. He worked as a Firmware Engineer with SPEA, a world-leading company in the design and manufacture of automatic test equipment (ATE). His research interest includes the design of new ultra-low-power architectures for machine learning-based edge applications.

Giovanni Ansaloni (Member, IEEE) received the Ph.D. degree in informatics from USI, in 2011. He is a Researcher with the Embedded Systems Laboratory of EPFL, Lausanne, CH. He previously worked as a Postdoc with the University of Lugano, USI, CH) from 2015 to 2020, and with the EPFL from 2011 to 2015. His research interests include domain-specific and ultra-low-power architectures and algorithms for edge computing systems, including hardware and software optimization techniques.

David Atienza (Fellow, IEEE) received the Ph.D. degree in computer science and engineering from UCM, Spain, and IMEC, Belgium, in 2005. He is a Full Professor of electrical and computer engineering, and the Director of the Embedded Systems Laboratory (ESL) with the Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland. He was a co-author of more than 300 papers in peer-reviewed international journals and conferences, several book chapters, and seven patents. His research interests include system level design methodologies for high-performance multi-processor system-on-chip (MPSoC) and low-power Internet-of-Things (IoT) systems, including new 2D/3D thermal-aware design for MPSoCs and many-core servers, ultra-low power edge AI architectures for wireless body sensor nodes and smart consumer devices. He is an ACM Fellow.