InSight: Enabling NLOS Classification, Error Correction, and Anchor Selection on Resource-Constrained UWB Devices

Markus Gallacher*‡, Michael Stocker*‡, Michael Baddeley§, Kay Römer‡, and Carlo Alberto Boano‡

*Both authors contributed equally to this research
‡Institute of Technical Informatics, Graz University of Technology, AT §Technology Innovation Institute, AE

{markus.gallacher@tugraz.at; michael.stocker; roemer; cboano}@tugraz.at
michael.baddeley@tii.ae

Abstract

The accuracy of ultra-wideband (UWB) ranging is severely affected when the direct path between devices is partly or fully occluded, i.e., in non-line-of-sight (NLOS) conditions. To detect and correct erroneous ranging measurements, many solutions based on machine learning models have been proposed, but they are usually deployed on edge devices rather than on the UWB device itself. In fact, existing works often focus on maximizing the NLOS classification accuracy and error correction performance, which results in large and computationally-complex models that cannot be run on UWB tags with limited processing power and memory. Whilst convenient, off-loading NLOS classification and error correction tasks to an edge device severely affects, among others, the scalability, privacy, and responsiveness of UWB-based localization systems, as tags need to actively exchange data with a third party and wait for its response, which may be delayed due to heavy load or unreliable communication.

In this paper, we present `InSight`: a framework that enables the deployment of NLOS classification and error correction models *directly* on resource-constrained UWB devices. `InSight` allows to train and generate such models according to specific requirements (e.g., on memory usage and on runtime), and to shed light on how to reduce the model size and runtime without degrading the classification accuracy and error correction performance. The selected models are then seamlessly integrated into a *NLOS engine* running on the device alongside existing applications and supporting any localization service. With `InSight`, we can perform NLOS classification and error correction directly on an UWB tag in 0.6 ms, and with as little as 8 B of RAM and 19 kB of flash memory – while retaining a classification accuracy of up to 86% and reducing the 90th-percentile ranging error by more than 1 m. We further show how a localization service can leverage `InSight` to select only anchors in direct line-of-sight and to correct erroneous NLOS ranging measurements, which improves the 90th-percentile localization error by up to 1.6 m on our 120 m^2 testbed.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Data Analytics; C.4 [**Performance of Systems**]

General Terms

Design, Experimentation, Performance

Keywords

DW1000, Localization, Neural Network, Machine Learning, NLOS, Obstacles, Testbed, UWB, Anchor selection

1 Introduction

Ultra-wideband (UWB) has recently become the technology of choice to create centimetre-accurate ranging and indoor positioning applications [20]. Its market is growing at a fast pace and is expected to hit 2.7 billion USD by 2025 [22], fueled by the introduction of UWB radios into high-end smartphones [10] and modern vehicles [33], as well as by the increasing adoption of the technology in asset tracking [17], robot navigation [18], and assisted-living [5] applications.

UWB achieves its high ranging accuracy by using a high bandwidth (≥500 MHz), which allows for the transmission of very short signal pulses (≈2 ns long). This enables UWB receivers to differentiate between the first pulse (corresponding to the direct path between two devices) and its reflections (originating from walls and other objects), enabling the precise calculation of the time-of-arrival (ToA) of a signal and, consequently, of the distance between two devices.

Under optimal conditions, i.e., when no obstacles block the direct line-of-sight (LOS) between two devices, UWB radios typically achieve centimetre-level accuracy [20]. However, in non-line-of-sight (NLOS) conditions, the direct path between two UWB devices is either attenuated by partially-blocking obstacles, causing ranging errors of a few decimetres, or entirely blocked, causing ranging errors up to a few metres [21, 29]. NLOS conditions hence strongly affect the performance of UWB ranging in real-world settings.

To address this issue, a large body of research has investigated how to *classify* NLOS conditions and consequently how to *correct* NLOS-induced ranging errors. Existing solutions range from simple methods analyzing the distribution of ranging measurements (where a higher variance is likely to indicate NLOS conditions) [3, 26], to more complex methods analyzing the radio's channel impulse response (CIR) and extracting parameters that can be used for threshold-based NLOS classification [26] or for performing likelihood-ratio tests from which to derive NLOS conditions [14].

Maranò et al. [21] go a step further and propose to use *machine learning* (ML) models trained on features extracted from the CIR for NLOS classification as well as error correction. Later works [1, 4, 25, 29, 34] build upon this seminal work and evaluate the performance of several ML methods, such as *multilayer perceptrons (MPLs)* [1, 25, 4], *support vector machines (SVMs)* [1, 4, 21, 34], and *ensemble methods* based on decision trees [19, 24, 25]. Recently, there has also been an increasing interest in using *convolutional neural networks (CNNs)* [1, 4, 32], as they do not rely on manually-extracted channel statistics as features, but instead autonomously extract features from the channel estimate.

Limitations of existing work. Unfortunately, most of the existing works focus on achieving the highest NLOS classification and error correction performance, and do not provide details about the memory requirements or the runtime of the proposed solution. Many of the proposed ML models, especially the CNNs, are computationally expensive: as a consequence, several authors propose to run them on dedicated *edge devices* such as Raspberry Pis with powerful CPUs/GPUs and hundreds of MBs of memory available [1, 4, 11].

Disadvantages of edge-based solutions. Running NLOS classification and error correction on edge devices, however, has negative implications on the scalability, reliability, timeliness, as well as privacy of UWB ranging and localization systems. In fact, a tag needs to reveal itself and actively transmit packets to the edge device for processing, which would be deleterious for privacy-preserving localization systems such as those based on time-difference-of-arrival [8, 13]. Moreover, communication to/from the edge device may be unreliable (e.g., due to the presence of radio interference causing re-transmissions [6]) and the edge device itself may be overloaded with requests from multiple tags, resulting in non-deterministic delays affecting the frequency at which a tag can update its position. For these reasons, NLOS classification and error correction should ideally be performed *directly* on the UWB device itself.

Limited support for on-device solutions. Unfortunately, UWB tags often have limited processing power and memory, and little research has focused on the creation of lightweight models as well as on their seamless integration alongside existing applications. Angarano et al. [1] have proposed a CNN whose size varies between 615 kB and 32 kB depending on the applied optimizations, but left the integration of the deep learning architecture on an embedded UWB device as future work. Jonas et al. [16] have been the first to run NLOS classification and error correction on an embedded UWB device (Qorvo's `DWM1001`), but their solution is not generic and there are no details about its memory footprint or runtime. Hence, to date, there is no work tackling the flexible integration of NLOS classification and error correction techniques on embedded UWB devices, while shedding light on how to reduce the model size and runtime to satisfy user-specific requirements without degrading performance.

Contributions. We fill this gap and present `InSight`, a framework enabling the deployment of state-of-the-art ML-based NLOS classification and error correction models on resource-constrained UWB devices. `InSight` allows to create and train models that satisfy specific requirements on runtime and memory usage, and to seamlessly integrate these models into a *NLOS engine* running on the UWB device alongside existing applications. `InSight` also allows to study the interplay between a model size/runtime and its performance: this enables us to carry out NLOS classification and error correction on an UWB tag in 0.7 ms and with as little as 8 B / 19 kB of RAM and flash memory – while retaining a classification accuracy of up to 88% and reducing the 90th-percentile ranging error by more than 1.4 m. We study `InSight`'s performance experimentally, showing also how a localization service can select only LOS anchors and correct erroneous NLOS ranging measurements, improving the 90th-percentile localization error by up to 1.6 m.

With `InSight`, we make the following contributions:

- We present an open-source framework that allows to generate ML models based on user-specific requirements, and to deploy them into a NLOS engine running on resource-constrained UWB devices (§ 3).
- We show the performance of state-of-the-art ML-based NLOS classification and error correction methods as a function of the model size/runtime, highlighting how one can reduce the input and model size by up to 80% and 98% while retaining a high performance (§ 4).
- We seamlessly integrate `InSight`'s NLOS engine in Contiki-NG, such that it runs alongside existing applications (§ 5) and evaluate its performance experimentally on off-the-shelf `Qorvo MDEK1001` UWB tags (§ 6).
- We carry out testbed experiments to show the benefits of on-device NLOS classification and error correction compared to the use of an edge device (§ 6), and further show how a localization service can easily leverage `InSight` to select only LOS anchors and correct erroneous NLOS ranging measurements (§ 7).

2 Related Work

Since the availability of the first IEEE 802.15.4-compliant UWB transceivers, a large body of work has explored various techniques for UWB NLOS classification and error correction. Existing work can be coarsely classified into *position-based*, *ranging-based*, and *channel-based* solutions.

Position-based solutions such as [15] estimate the presence of LOS/NLOS conditions between two nodes based on floor maps and by leveraging a ray tracing algorithm. While this approach can yield excellent classification results, it is limited to static environments and is unable to cope with any substantial change in the environment.

Ranging-based solutions such as [3] and [35] measure the variability between several consecutive ranging estimates for NLOS classification. The rationale is that the ranging estimates change frequently in NLOS conditions, due to the high variability of multi-path components. Whilst effective, this approach requires repeated distance estimations to the same anchor, which is undesirable due to the resulting message overhead, higher delays, and energy expenditure.

Channel-based approaches, which are the most common, extract various parameters from the CIR estimate, as the latter follows specific patterns under LOS and NLOS conditions. For example, Schröder et al. [26] use the amplitude and delay statistics of the CIR for threshold-based classification. Vu et al. [23] evaluate features provided directly by the radio transceiver to classify occlusions caused by the human body. Güvenç et al. [14] utilize a likelihood-ratio test using three parameters extracted from the CIR, namely the root mean square delay spread, the mean excess delay, and Kurtosis.

ML-based channel-based solutions. However, as pointed out by Maranò et al. [21], deriving sufficient statistical models can be an error-prone process: for this reason, the authors propose instead the use of *non-parametric ML* with manually-extracted features for NLOS classification and error correction. Stocker et al. [29] follow this line of research and evaluate the approach from Maranò et al. [21] on off-the-shelf devices, showing that the NLOS error correction is not perfect and negatively affects LOS measurements. Ramadan et al. [24] train a random forest algorithm, based on multiple decision trees, on manually extracted features and show its viability in classifying NLOS conditions. More recently, the focus of related work has shifted to various ML approaches using *self-learned* instead of manually-extracted features [4, 16, 1, 27, 28]. This development is driven by the assumption that self-learned features are able to capture complex patterns and relationships that are overlooked by humans. Bregar et al. [4] are among the first to use MLPs and CNNs to extract features directly from the CIR to perform NLOS classification and error correction on edge devices. Angarano et al. [1] propose a CNN for error correction, using state-of-the-art neural network architectures such as feature attention mechanism at its core. Stahlke et al. [27] propose and test several neural network architectures for NLOS classification in an industrial environment. The authors also use variational autoencoders for estimating the reliability of UWB range estimates, and further use this information in a Kalman filter [28]. Sung et al. [31] propose a deep neural network to estimate range measurement uncertainties and update the weights of a Kalman filter accordingly.

Addressing the ML gap. Most of the aforementioned works concentrate on achieving a high performance, and do not provide details about the memory requirements or the runtime of the solution. As a consequence, many authors propose to run the ML models on edge devices [4, 1], and there has been little to no focus on the integration of NLOS classification and error correction capabilities *into an embedded device* [4, 21, 28, 29], as well as on how to decrease the model size and runtime without sacrificing performance. Angarano et al. [1] did investigate how to reduce the model size and runtime of the proposed neural network model using various mode complexity reduction techniques (such as integer quantization), resulting in a model requiring only 32 kB of flash memory and RAM, but left the integration on an embedded device as future work. Jonas et al. [16] showed the feasibility of running ML models on a UWB device, but with a very specific implementation, and did not provide an analysis of the model size or runtime. In order to enable the deployment of ML methods on embedded UWB devices, it is needed to (i) have a clear understanding of the interplay between accuracy and size/runtime for different ML models; (ii) find configurations that satisfy different requirements (e.g., very short runtime to support fast update rates or tiny memory footprint to support devices whose limited memory is already largely used by the application or operating system), ideally in an automated way; (iii) work on the seamless integration alongside the operating system and application to support a variety of ranging and localization services. These three points represent a still open research gap, which we address with the design of the `InSight` framework.

Table 1: Examples of user-specific requirements.

Configuration	**Memory footprint** max (RAM)	max (Flash)	**Runtime** max
On-device `Small`	77 kB	154 kB	10 ms
On-device `Tiny`	10 kB	77 kB	0.5 ms

3 `InSight`: Overview

Fig. 1 shows a high-level overview of the `InSight` framework's architecture and its inputs. Two key modules are at the core of `InSight`: an offline *model generator* that allows to easily train and analyse ML models on different datasets based on user-specific memory and runtime requirements (detailed in § 4), and an *embedded NLOS engine* running on the UWB device that provides on-device NLOS classification and error correction to any ranging/localization service providing a CIR as input (detailed in § 5). We keep the design of `InSight` modular to easily adapt the NLOS detection and error correction models to new requirements, and we make it available open-source[1] to make NLOS-aware UWB ranging/localization widely available.

Framework inputs. `InSight` is designed to be generic, and can be configured by specifying different *user-specific requirements*, *ML methods*, and *datasets* as input.

User-specific requirements. `InSight` allows tailoring of the models to the requirements of the platform/application at hand. To this end, the user can specify the maximum memory (RAM and flash) and runtime (in ms) that the selected classification/error correction model should satisfy. Tab. 1 lists exemplary requirements for a `Small` UWB device (built with an `nRF52840-DK` with 256 kB of RAM and 512 kB of flash memory), as well as a `Tiny` UWB device such as the `MDEK1001` (with 32 kB of RAM and 256 kB of flash memory). We set the max memory footprint per model to 1/3 of the available RAM and flash memory on the device in order to leave sufficient space for the operating system (OS) and

[1] `http://iti.tugraz.at/uwb-nlos`

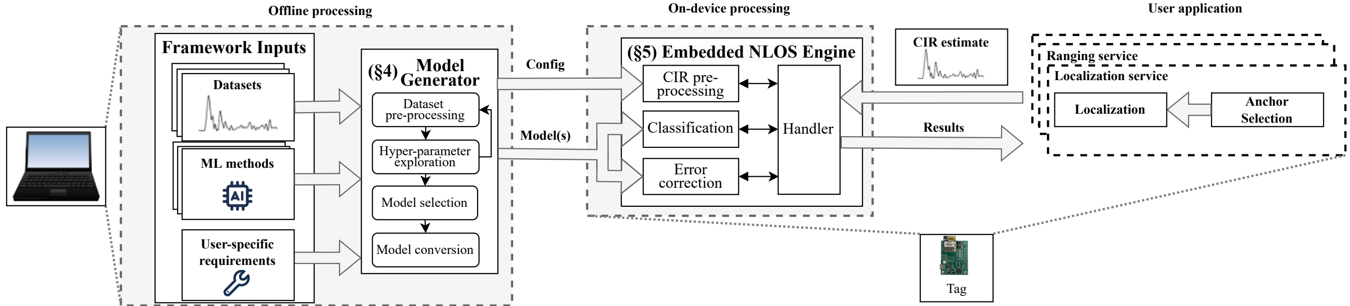

Figure 1: `InSight` module overview, including the offline *model generator* (§ 4), which outputs the best suitable model according to the *framework inputs*, and the on-device *embedded NLOS engine* (§ 5), which performs classification and/or error correction based on the CIR supplied by a localization or ranging service/application.

user application. We further limit the `Tiny` device to a max runtime of 0.5 ms per model so to allow for a fast update rate, whilst the `Small` device allows for a longer runtime (10 ms) to emulate a system that prioritizes classification accuracy and error correction performance over a high update rate.

ML methods. In principle, any ML method can be provided to the *Model Generator* if a compatible interface is present. Currently, the *Model Generator* provides implementations for SVMs, XGBoost trees, and neural networks. These represent non-parametric ML models with manually-extracted features from the CIR (SVMs), non-parametric ML models with automatic feature extraction by directly inputting the CIR (XGBoost trees), and parametric ML models with automatic feature extraction by directly inputting the CIR (CNNs). The SVM is based on the library `libsvm`[2] by Chang et al. [7]. The XGBoost models are based on the *dmlc/xgboost*[3] library. Both the `libsvm` and `dmlc` libraries can also be used through the popular `scikit-learn` library. The REMNet, a CNN for error correction based on the structure proposed by Angarano et al. [1], is implemented in TensorFlow. For REMNet-based classification, we modify the proposed structure to use a sigmoid activation function as output layer instead of a linear activation function. We also combine both error correction and classification models into a single one, calling it REMNet multi-output (REMNet MO) model [12], where we use two output neurons, one with a sigmoid activation function for classification and one linear activation function for error correction.

Datasets. `InSight` is designed to take single or multiple datasets as training and/or testing sets. This allows the easy comparison of ML methods in diverse settings (from self-recorded to publicly-available traces). Multiple datasets can also be combined to generate larger training/test sets. When no test dataset is provided, a *k-fold* cross validation is performed, i.e., we divide data into `k` chunks and average the test results from each chunk, while training on the other chunks.

Model generator. Using the aforementioned user requirements, ML methods and datasets, the *model generator* module entails the execution of four components, namely the *dataset pre-processing*, the *hyper-parameter exploration*, the *model selection*, and the *model converter*. The *dataset pre-processing* component takes care of bringing several datasets into a common format, as well as performing feature extraction and scaling. The pre-processed datasets are then supplied to the *parameter exploration* component to find optimal hyper-parameter combinations. Essentially, this component tries various combinations of input sizes and model hyper-parameters within a pre-defined parameter search space. The *model selection* component evaluates the results of the parameter exploration module and selects the best fitting models based on the user-specific requirements, such as those shown in Tab. 1. The *model converter* converts the models to be usable by the target platform and generates configuration files needed to pre-process the CIR. The models and configuration files are the input of the embedded NLOS engine.

Embedded NLOS engine. This module performs on-device classification and/or error correction and is easy to integrate into a variety of applications (e.g., from simple ranging to complex localization systems). We have a prototypical implementation for Contiki-NG, where the *embedded NLOS engine* is added seamlessly, i.e., it does not affect any existing functionality and has a simple API that can be called by any ranging/localization service. The embedded NLOS engine module comprises the *handler*, *CIR pre-processing*, *classification*, and *error correction* components. The *handler* is the common interface between a user application and the embedded NLOS engine, and handles CIR classification as well as error correction requests. Similar to the *dataset pre-processing* component in the model generator, the *CIR pre-processing* component of the embedded NLOS engine is responsible for CIR pre-processing, feature extraction, and feature scaling. The *classification* and *error correction* components are the models that have been generated and configured via the *model generator*. The embedded NLOS engine and its components are OS-agnostic.

4 Model Generator

This section provides details on the various *model generator's* components. We first outline the dataset pre-processing steps (§ 4.1). We then discuss the hyper-parameter exploration and show how different hyper-parameter combinations affect the model size and performance (§ 4.2). We finally show how a suitable model is matched to the user requirements (§ 4.3), and explain how models are converted to be suitable as an input for the embedded NLOS engine (§ 4.4).

[2] `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`, version 3.23.0.4

[3] `https://xgboost.readthedocs.io/en/stable/`, version 1.6.1.

This section ultimately answers the following questions:

- To what extent can the model size (complexity) be reduced without resulting in a significant loss in classification accuracy and error correction performance? Which hyper-parameters are crucial in doing so? (§ 4.2)
- Can the CIR input size be reduced? If so, where is the most valuable info within the CIR? Is there a difference for NLOS classification and error correction? (§ 4.2)
- Can we automatically select models that satisfy user requirements, e.g., those listed in Tab. 1 demanding as little as 0.5 ms runtime and 10 kB of RAM? (§ 4.3)

4.1 Dataset Pre-processing

This component ensures that training and testing data follow a standard structure and are well prepared for the individual models. Dataset pre-processing can be coarsely grouped into three steps: *loading*, *filtering*, and *transforming*, each of which is represented by re-configurable pipelines.

The *loading pipeline* is specific to each dataset and takes care of loading various datasets and bringing them into a standard format. This encapsulates simple steps such as providing a common naming scheme across datasets, as well as correctly *scaling* the CIR and *aligning the first path* to a unified position in the CIR. Firstly, scaling is necessary since CIRs are often recorded with different numbers of preamble symbols and the number of preamble symbols affects the amplitudes in the CIR. Furthermore, alignment of the first path (FP) is necessary, as UWB radios do not always place the detected FP at the same index within the CIR, and the FP index may hence be dissimilar across datasets [30].

The *filtering pipeline* is common to all loaded datasets and removes extreme outliers that can occur due to random hardware- or environment-related effects. This is achieved by calculating the median range value of measurements taken in the same placement and removing range measurements that exceed 2.5 times the standard deviation. With this step, the natural fluctuation of NLOS measurements is preserved while avoiding incorporating extreme outliers that can negatively impact the scaling of features.

The *transformer pipeline* is specific to each model and prepares the data to be fed to the model. This pipeline firstly selects a CIR window, i.e., a subset of the CIR samples that is configurable via two parameters: CIR start index and CIR end index. For feature-based methods such as the SVM, channel statistics are extracted from the selected CIR samples and scaled with a min-max scaler. For raw CIR methods such as XGBoost and REMNet, the selected CIR samples are scaled and forwarded to the ML model. Note that the transformer pipeline is executed every time the CIR window is altered before running the hyper-parameter exploration.

4.2 Hyper-Parameter Exploration

The *hyper-parameter exploration* component creates an extensive mapping between the user-requirements (model size and runtime), the model performance (classification accuracy and R^2-score for error correction), as well as different sets of hyper-parameters and CIR input sizes. A simple grid search approach is used to show the impact of various hyper-parameters on the performance, as well as to gain insights on the model complexity and CIR input size reduction[4].

Table 2: Overview of hyper-parameters which have significant impact on memory footprint, latency, and performance.

ML method	Name	Values
Common to all methods	CIR Start Index CIR End Index	[0,5,10,15,18] [172,132,72,38,25]
REMNet	#Filters (F) #Residual blocks (N)	[16, 8, 4] [3, 2, 1]
SVM	Regularization term (C) Penalization tolerance (ε)	[1, 100, 1000] [0.1, 0.2, 0.3]
XGBoost	Maximum tree depth (D) Maximum number of Trees (T)	[10, 6, 3, 1] [100, 60, 30, 10]

Dataset. To exemplify the hyper-parameter exploration, we use the publicly-available dataset by Stocker et al. [29], containing 37450 measurements recorded in a hallway and several office/lecture rooms. In this dataset, it is distinguished between LOS, weak-LOS (i.e., small obstacles such as monitors), and NLOS (i.e., large obstacles such as walls). However, since most related work treats the NLOS problem as a binary classification, we combine weak-LOS and NLOS under a unified NLOS label. We refer to this dataset as `ST`.

Hyper-parameters. Each of the investigated ML methods has a multitude of hyper-parameters to tune, resulting in an explosion of possible combinations. Thus, we limit our analysis to the hyper-parameters presented in Tab. 2, as we expect these parameters to impact the model size and performance the most. Common to all methods is the CIR window, which is defined by the index of the first and last sample of the CIR[5]. For the *REMNet* [1], we identify the number of filters (`F`) used in each convolution step and the number of residual reduction modules (`N`), i.e., the number of feature attention and layer size reduction blocks, to have great effects on size and accuracy. For the SVM, the regularization parameters `C` and ε determine how lenient the model is to misclassification and wrong error correction values during training. Depending on the data distribution, both parameters can affect the model complexity and size. The size and performance of the *XGBoost* tree largely depend on the maximum depth (`D`) of a single tree estimator, i.e., how many nodes a single tree can have, and on the maximum number of tree estimators (`T`), i.e., how many individual decision trees are allowed.

Reduction of model complexity. Figs. 2 to 5 show the results of the hyper-parameter exploration for different ML models and shed light on to which extent the model size can be reduced and on which hyper-parameter has the biggest impact. Each point in the plots represents one hyper-parameter combination selected from the values in Tab. 2. To find the best hyper-parameter combinations, we search for Pareto-optimal

[4] Note that the grid search approach is computationally expensive. For example, finding a suitable model for XGBoost on an AMD Ryzen 7 PRO 5850U CPU with the hyper-parameters listed in Tab. 2 takes around one day. However, one can easily replace the grid search with other hyper-parameter optimization approaches, such as halving grid search, randomized search, and FLAML (a fast library for automated ML and tuning), which can deliver similar optimal hyper-parameters in much shorter time. For the purpose of this paper, we stick to the grid search, as faster approaches do not show the impact and boundaries of individual hyper-parameters and their combinations with the same granularity as the grid search approach.

[5] The *data-loading* pipeline moves the detected FP of each measurement to index 20.

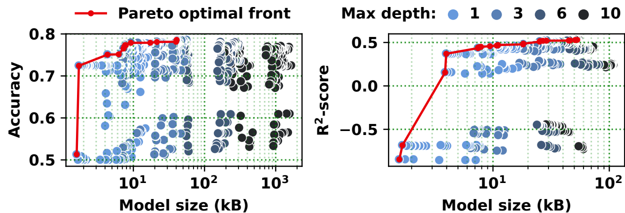

Figure 2: Performance of XGBoost (in terms of classification accuracy and R^2 score for error correction) as a function of the model size (flash memory). The impact of the max depth of each tree `D` is depicted as shaded data points.

points, which assert that increasing one objective function's performance is impossible without reducing the performance of another objective function. The red dots represent the individual Pareto-optimal points, which are connected with a red line to form a Pareto front. The grid search exploration has shown that we can extract one hyper-parameter for each ML method that contributes vastly to the model size, but does not lead to a drop in the classification and error correction performance. Fig. 2 shows the XGBoost performance as a function of the model size (flash memory). Reducing the `max depth` of each individual tree can decrease the model size by one or two orders of magnitude, while still achieving a high accuracy and R^2-score. We focus on flash memory usage, as the XGBoost model requires only 4 bytes of RAM (it uses a single floating point variable), whilst the rest of the code is stored in flash. Fig. 3 shows the REMNet performance as a function of the model size (RAM). We can see that the number of residual blocks (`N`) reduces the model size by a third, but keeps a high accuracy and error correction performance. The same is true for the multi-output layer REMNet, whose performance is shown in Fig. 4, which has the advantage of providing a single model for classification and error correction (i.e., half the model size and runtime than the two individual REMNets). Note that for the REMNet models the constraining factor is the RAM usage, which is why we depict the model size in RAM. Fig. 5 shows the performance of the SVM: we can notice a decrease in model size with higher `C` values for classification, and with higher ε for error correction. However, the hyper-parameter ε defines which error is penalized by the SVM. In our context, this corresponds to the ranging error, and should hence be kept to a value of 0.1 (i.e., a 10 cm error) if we do not want to degrade the system's performance. Note that we depict the model size in RAM also for the SVM, as this is the limiting factor in the SVM implementation.

We conclude that, for XGBoost and REMNet, we can reduce the model size for UWB datasets by 66-98%, while only suffering up to 5% performance penalty. The SVM size depends on the number of support vectors and is also affected by the dataset distribution, which makes it harder to shrink.

CIR input size reduction. Using the Pareto-optimal hyper-parameter combinations, we study where the most important information lies within a CIR for our models, and by how much the CIR length can be reduced. A reduced CIR window is beneficial to our embedded NLOS engine, as we can read and process the shortened CIR faster. Furthermore, the

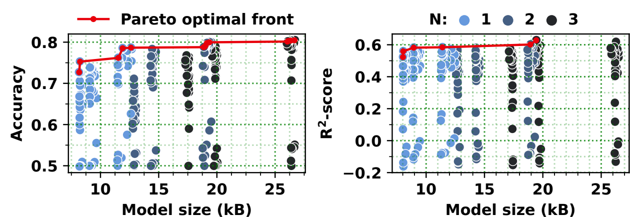

Figure 3: Performance of REMNet (in terms of classification accuracy and R^2 score for error correction) as a function of the model size (RAM). The impact of the number of residual reduction modules `N` is depicted as shaded data points.

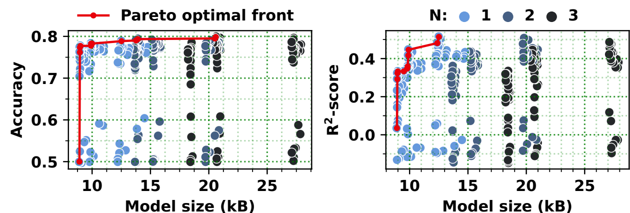

Figure 4: Performance of the multi-output REMNet (in terms of classification accuracy and R^2 score for error correction) as a function of the model size (RAM). The impact of `N` is depicted as shaded data points.

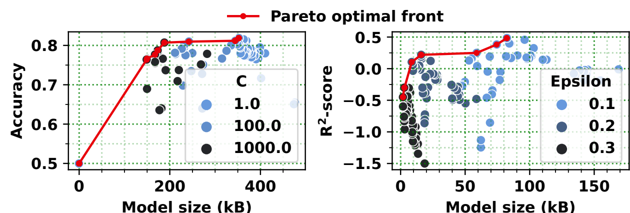

Figure 5: Performance of the SVM (in terms of classification accuracy and R^2 score for error correction) as a function of the model size (RAM). The impact of `C` and ε is depicted as shaded data points.

CIR window size determines the input size of the REMNet and directly affects the time needed for the feature extraction of the SVM input. Fig. 6 shows the impact of the CIR end index on the Pareto-optimal hyper-parameter combinations (i.e., the Pareto-optimal points are ordered based on the CIR end index). It can be seen that for XGBoost and REMNet we can reduce the number of CIR samples to 5 or 10 samples after the first path, before noticing a performance degradation, while the SVM requires a larger CIR window.

We conclude that for the XGBoost and REMNet models, all the necessary information required for classification and error correction lies within 20 samples before the FP and 10 samples after the FP. The SVM, using manually extracted features, requires a larger CIR window. As the size of the SVM models are very large (see Fig. 5) and do not meet the target memory requirements in Tab. 1, we focus only on the REMNet and XGBoost models in the rest of this paper.

4.3 Model Selection

Using the results from the *hyper-parameter exploration* component, the *model selection* component selects the best-performing model out of all ML methods that satisfy the

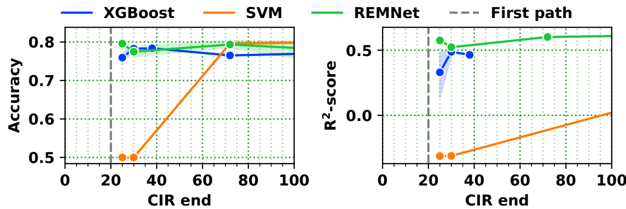

Figure 6: Impact of the CIR end index based on the Pareto-optimal points. The performance of XGBoost and REMNet does not decrease even when reducing the CIR end index to 10 samples after the FP. The SVM, instead, needs more samples for the feature extraction to perform well.

user's memory and runtime requirements. To exemplify this, we use the requirements from Tab. 1 and follow three essential steps. First, we confine our hyper-parameter exploration results to all combinations that satisfy our user requirements, i.e., feasible domain. Second, we calculate the Pareto-optimal points of the feasible domain based on our objective functions (i.e., accuracy, R^2-score, or both for the multi-output REMNet). Third, with a single objective function, the Pareto-optimal point is always the hyper-parameter combination that maximizes the objective function and is our best suitable model that satisfies all user requirements. With two or more objective functions, it is not guaranteed to find a single Pareto-Optimal point: therefore, a trade-off between the objective functions is made. In this case, we use an achievement scalarization function from the `pymoo` library [2], i.e., a decomposition method to reduce the multi-objective problem into a single-objective problem, and find the best Pareto-optimal point.

Size and run-time estimation. To satisfy the user requirements, we estimate the RAM size of the REMNet, as well as the flash memory size of the XGBoost models, as these are the limiting memory constraints. The RAM of the REMNet model depends on the memory needed for inputs, outputs and intermediate results, i.e., the tensor arena size. This can be estimated, for example, using a divide and conquer algorithm[6]. The XGBoost model resides entirely in program memory, and is represented by a sequence of if/else comparisons. The flash memory size is calculated using the byte size of the code file. To estimate the runtime of each model, we first calculate the number of expected floating point operations (FLOPs) per model and hyper-parameter configuration. We then approximate the relationship between FLOPs and runtime by measuring the runtime of several reference models on the target device. Estimating the FLOPs for the REMNet is already supported by the TensorFlow library. For XGBoost, the worst case number of FLOPs is `(D+1)·T`, where `D` is the maximum tree depth, and `T` is the number of trees. This is a worst-case estimate, as the individual trees are subject to pruning and must not be of the full depth `D`.

Meeting the user-specific requirements. Tab. 3 shows the four best suitable models for the user requirements listed in Tab. 1. The REMNet provides the best suitable model for the on-device `Small` requirements. Both the classification

[6]https://github.com/edgeimpulse/tflite-find-arena-size

Table 3: Selected classification and error correction models that meet the user-specific requirements in Tab. 1.

Requirement	ML method	Memory footprint RAM	Memory footprint Flash	Classification Accuracy	Error correction R^2-score	Runtime Estimate
On-device `Small`	REMNet	11.9 kB	11.6 kB	0.76	-	5.3 ms
		11.4 kB	11.1 kB	-	0.58	7.4 ms
On-device `Tiny`	XGBoost	4 B	21.4 kB	0.78	-	0.29 ms
		4 B	18.2 kB	-	0.49	0.29 ms

and error correction models use `N=1` and `F=16`. The classification model uses a CIR start/end index of 15 and 30, whereas the error correction model uses a CIR start/end index of 0 and 25, respectively. Note that as the user requirements are assessed on a per-model basis, the size advantages of multi-output models are not considered. XGBoost provides the best suitable models for the on-device `Tiny` user requirements. Both classification and error correction models use a max depth `D=3` and `T=30` tree estimators. The CIR start/end index are 0/30 for classification and 5/25 for error correction, respectively.

4.4 Model Conversion

The *model conversion* component takes care of converting the trained models into a suitable representation to be used in the embedded NLOS engine, and of providing a configuration file. The individual conversion steps are custom to each model and involve the extraction of the model parameters from the trained model and the conversion of these parameters to a configuration file used at compile time. The REMNet-based models are implemented using the TensorFlow library, and we use the conversion functionality to export a full integer quantized version of the trained models. The model parameters are then written into a *C* file with a single `uint8_t` array containing all model parameters. The TensorFlow Lite Micro library then uses the latter and is loaded at initialization time. The model converter for XGBoost follows a different approach: instead of exporting only the model parameters, the whole tree-like structure of the XGBoost model is replicated in *C* with if/else blocks. This allows for the code to be stored in flash memory, while only requiring a single floating point variable in RAM.

5 Embedded NLOS Engine

We detail next the embedded NLOS engine internals (§ 5.1) and then discuss its integration within *Contiki-NG* (§ 5.2).

5.1 Engine Internals

To make the embedded NLOS engine model-agnostic, we define four interfaces resembling the components, as shown in Fig. 1: the *handler*, *CIR pre-processing*, *classification*, and *error correction* interfaces. In addition, there is also a model-specific *initialization* interface, which takes care, for example, of loading the REMNet model parameters into RAM. Except for the *handler*, all other interfaces are custom to the ML model and need to be implemented when adding a new ML method to `InSight`. In the case of XGBoost, the NLOS engine adds only 348 B of program memory and 992 bytes of data segment due to the pre-processing component, on top of which the actual model has to be added.

The *handler* component is the entry point for new classification or error correction requests to the NLOS engine. It

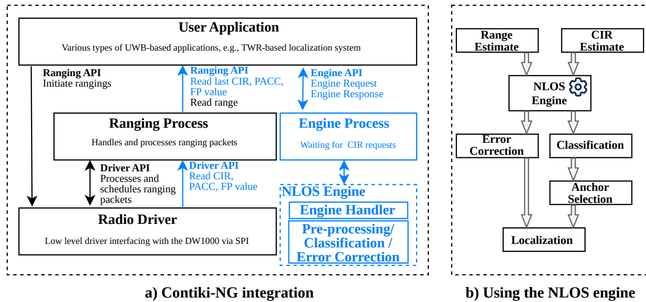

Figure 7: Integration of `InSight`'s NLOS engine into Contiki-NG (a) and exemplary use of the engine to improve the performance of a localization application (b). A user application initiates a ranging estimate via the ranging API; the *ranging process* handles any subsequent TWR message exchange. Minor modifications (highlighted in blue) to the *ranging process* allow the application to retrieve the CIR estimate. The application then issues a request to the *engine process* to perform classification and/or error correction.

can be operated on one of four *operational modes*: *classification only*, *error correction only*, *classification and error correction*, as well as *error correction if NLOS*. The latter provides an error correction only if NLOS is detected, and hence saves inference time. The *handler* component takes as input the CIR, the preamble accumulation counter (PACC) required for CIR normalization, the estimated range, and the operational mode. After execution, the *handler* returns a *result* object containing the classification and error correction.

The *CIR pre-processing* component is executed by the *handler* and is model specific. Its purpose is to prepare the supplied CIR to be fed into to the deployed ML model. This involves the correct selection of the CIR window, as well as the extraction of features and their scaling (if required). The individual steps and parameters are configured at compile time and mimic the steps performed in the data pre-processing block of the model generator.

Classification and error correction. The *classification* and *error correction* components are executed by the *handler*. These components start the inference of the underlying ML models, and convert/scale the output if necessary. For example, our ML models for error correction are trained to output a value between [0,1], and must be scaled to values in metres.

5.2 Integration into Contiki-NG

We seamlessly integrate `InSight`'s embedded NLOS engine into the popular Contiki-NG operating system, showing that it allows user applications to request both NLOS classification and error correction functionality with only minor modifications to existing code. Fig. 7(a) gives an overview of the engine's integration into Contiki-NG, showing in blue the additions compared to Contiki's baseline implementation[7]. These additions result in an increase of only ≈ 200 B in program memory, 20 B in RAM, and 228 B in the data segment.

The *radio driver* interfaces directly with the transceiver and offers a rich API for tuning the message reception and transmission. We create a preliminary implementation for Qorvo's `MDEK1001` platform, which embeds the DW1000 radio; hence, the radio driver interfaces with the DW1000 via SPI. The *radio driver* also provides access to radio diagnostic values, including the CIR estimate, the position of the estimated FP, as well as the PACC value. When reading the CIR, we ensure that only samples of interest are read, according to the CIR window specified by the model generator.

The *ranging process* is part of Contiki's original ranging stack and handles the exchange of UWB messages during the two-way ranging (TWR) process. Two to four messages are exchanged (depending on whether single-sided or double-sided TWR is executed), and their transmission and reception timestamps are recorded to calculate the ToF and distance between two UWB radios. Due to imperfections of crystal quartz clocks, which tend to accumulate errors over time, the TWR process should be completed as quickly as possible to minimize errors. Therefore, any time-consuming operation, such as reading the CIR via the SPI interface, should be avoided. To this end, while integrating `InSight` into Contiki-NG, we ensure that the user application only retrieves the CIR of the last message in the TWR sequence. The last CIR is hence returned together with the PACC counter to the *user application*, where it can be forwarded to the *engine process* in order to leverage `InSight`'s classification and error correction functionality.

The *engine process* waits for requests from the user application and passes them on to the NLOS engine. Once the NLOS classification or error correction results are available, they are returned to the application through the same process.

Fig. 7(b) shows how an exemplary *user application* can use the NLOS engine to improve its localization accuracy. While estimating the range to several anchor nodes, the application provides each individual ranging measurement along with its CIR estimate to the NLOS engine. The engine's results can be used to discard or correct ranging measurements taken in NLOS conditions. In § 7, we will show experimentally how this indeed allows an improved anchor selection and an increased localization accuracy by up to 1.6 m.

6 Evaluation

We evaluate next the performance of `InSight` experimentally. After describing our testbed setup and the datasets employed to train the models (§ 6.1), we investigate the classification accuracy, error correction performance, and runtime of the lightweight models produced by `InSight`, providing also a comparison with the performance of state-of-the-art solutions (§ 6.2). We further show the benefits of performing *on-device* NLOS classification and error correction by comparing the delay with that of an edge-based solution (§ 6.3).

6.1 Testbed and Datasets

We test `InSight`'s performance experimentally on an UWB testbed installed at our University, which consists of 36 Qorvo `MDEK1001` devices mounted on walls across an area of 120 m^2 encompassing an office, an entrance, and a hallway.

We let `InSight` generate models matching the user-specific requirements listed in Tab. 1. We train the classification models using two publicly-available datasets: the `ST` dataset de-

[7] We have used the Contiki UWB stack available at `https://github.com/d3s-trento/contiki-uwb` as a starting point, ported it to Contiki-NG, and enriched it with the embedded NLOS engine.

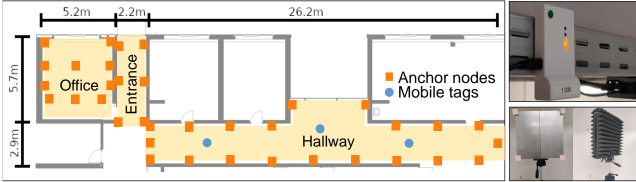

Figure 8: Overview of our testbed setup. Left: testbed topology, with 36 anchor nodes (orange squares) and 3 mobile tags (blue circles). Top-right: wall-mounted anchor nodes. Bottom-right: obstacles used to create NLOS conditions.

scribed in § 4.2, and the dataset published by Bregar et al. [4], which we refer to as the `BR` dataset. For training the error correction models, we use the `ST` dataset supplemented with 20606 measurements gathered from our testbed, which we refer to as the anchor-to-anchor (`ATA`) dataset. To create the `ATA` dataset, we let pairs of nodes mounted on walls (i.e., the anchor nodes marked as orange squares in Fig. 8) perform double-sided TWR (DS-TWR) among each other[8] in both LOS and NLOS conditions. To introduce additional NLOS conditions than those caused by the building's geometry, we artificially place a metal sheet and a foam absorber at various locations throughout the testbed[9]. We further create an additional tag-to-anchor (`TTA`) dataset with 54175 measurements that we use exclusively for testing. The `TTA` dataset is created by placing a mobile tag at three locations within the testbed (marked as blue circles in Fig. 8), and by letting them range to nearby anchors[10]. All measurements in the two datasets are taken using channel 5, 128 preamble symbol repetitions, and a pulse repetition frequency of 64 MHz.

6.2 Models' Correctness and Performance

We study the correctness as well as the classification and error correction performance of the models generated by `InSight`.

Models' correctness. When providing `InSight`'s model generator with the user requirements listed in Tab. 1 and with the training sets detailed in § 6.1, we obtain two sets of models. The best suitable classification and error correction models for the on-device `Small` requirements are based on a REMNet, and have one residual reduction model (`N`=1). The classification model uses a CIR start/end index of 5/25 and `F`=8, whereas the error correction model uses a CIR start/end index of 0/25 and `F`=16, respectively. Tab. 4 shows the memory footprint of the two REMNet models, i.e., 8.4/9.3 kB of RAM and 8.8/11.1 kB of flash memory for the classification/error correction models, respectively.

The best suitable classification and error correction models for the on-device `Tiny` requirements are based on XGBoost. The classification/error correction models use a max depth `D=3/1`, a max number of tree estimators `T`=10/100, a CIR start index of 18/10, and a CIR end index of 38/38, respectively. Tab. 4 shows the memory footprint of the two XGBoost models, i.e., 4/4 B of RAM and 6.7/11.7 kB of flash for the classification/error correction models, respectively. As expected, all models returned by `InSight`'s model generator satisfy the memory requirements listed in Tab. 1.

Table 4: Memory footprint, runtime, and performance of the on-device models used by `InSight` to satisfy the user requirements listed in Tab. 1, and of a baseline model [1].

Model name	ML method	Memory footprint RAM	Memory footprint Flash	Classification Accuracy	Error correction R^2-score	Runtime Measured
`Baseline`	REMNet	42.1 kB	37.5 kB	0.81	-	1.03 ms
		41.9 kB	37.3 kB	-	0.25	0.99 ms
On-device `Small`	REMNet	8.4 kB	8.8 kB	0.90	-	2.8 ms
		9.3 kB	11.1 kB	-	0.58	8.7 ms
On-device `Tiny`	XGBoost	4 B	6.7 kB	0.86	-	0.08 ms
		4 B	11.7 kB	-	0.35	0.46 ms

Models' performance. We evaluate next the classification accuracy and R^2 score of the aforementioned models by running them on an `MDEK1001` UWB device. We also compute the accuracy and R^2 score obtained by the offline-generated models (i.e., those returned by the model generator before the conversion step discussed in § 4.4). We use the `TTA` dataset (see § 6.1) as a test set for all models. The REMNet models (`Small`) achieve an R^2-score of 0.58/0.61 and an accuracy of 0.91/0.9 for the on-device/offline models, respectively. The XGBoost models (`Tiny`) achieve an R^2-score of 0.35/0.37 and an accuracy of 0.86/0.84 for the on-device/offline models, respectively. The slight mismatch among the on-device/offline models is due to full integer quantization (when converting the REMNet model) and to reduced floating point decimals (for the XGBoost models).

Comparison with other models. Tab. 4 summarizes the classification accuracy and error correction performance (R^2-score) obtained by the on-device `Small` and `Tiny` models. It further lists the performance obtained by the original REMNet by Angarano et al. [1] with the configuration recommended by the authors run without quantization on a laptop with an AMD Ryzen 7 PRO 5850U CPU (`Baseline`). Looking at Tab. 4, we can observe that `Baseline` is outperformed by the on-device `Small` and `Tiny` models in terms of both classification accuracy and error correction. [11] Moreover, we can observe that the on-device `Small` REMNet models have a 5% higher classification rate and a higher R^2-score (+ 0.32) than the on-device `Tiny` XGBoost model. However, the `Small`'s runtime is 19/34 times longer than that of `Tiny` for error correction and classification, respectively.

6.3 On-Device vs. Edge Performance

We now show the benefits of performing *on-device* NLOS classification and error correction by comparing the delay in running the ML models on the embedded device, and in offloading the computation to an edge device.

On-device delays. We use Contiki-NG's `rtimer` to measure the processing time of the embedded NLOS engine's *pre-processing*, *classification*, and *error correction* components.

[8] We perform DS-TWR between pairs of anchors that are up to 10 m apart.

[9] Metal sheet and foam absorber have a size of 50x50 cm and a thickness of 2 mm and 19 cm, respectively. Obstacles are at 0.4 m from the anchors.

[10] The tag performs DS-TWR to 13 surrounding anchors (8–10 of which in LOS, and 3–5 in NLOS) at different orientations (which we vary at 45° steps). Obstacles (i.e., the aforementioned metal sheet and absorber) are placed at 0.2, 0.5, and 0.8 m from the tag.

[11] Throughout our study, we have observed that larger models tend to overfit, resulting in worse classification and error correction performance in new environments. Smaller models, e.g., on-device `Small` and `Tiny`, seem more robust. When using `TTA` as a test set, we have indeed observed using `InSight`'s model generator that the returned on-device `Small` model is the best performing regardless of any memory/runtime requirements. In future work, we will investigate if this observation holds true in the general case.

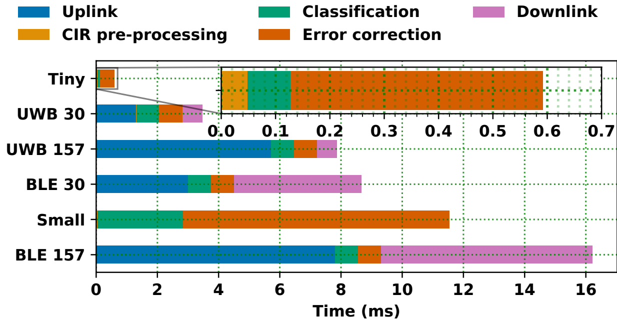

Figure 9: Delays in running the classification and error correction task on the embedded device, and in off-loading a similar tasks to an edge device with a given CIR length.

We exemplify this by running the Tiny / Small models derived in § 6.2. Fig. 9 illustrates the results: about 8% / 0.4% of the total processing time is due to CIR scaling (≈ 48 µs for Tiny and Small). The NLOS classification and error correction take 80 µs / 2.8 ms and 464 µs / 8.7 ms, respectively. These values remain rather constant: the standard deviation is below the resolution of the rtimer of ≈ 16µs.

Edge delays. We use a laptop with an AMD Ryzen 7 PRO 5850U CPU acting as an edge device, and measure the execution time of the same *pre-processing*, *classification*, and *error correction* components implemented in Python. To exemplify, we run the Small REMNet models derived in § 6.2 before the conversion step discussed in § 4.4 in Python. The *pre-processing* takes only 14 µs, executing the classifier takes 760 µs, whereas running the error correction model takes 740 µs. As off-loading the NLOS classification and error correction task to an edge device encompasses the transmission of the CIR (uplink) as well as the reception of the results (downlink), we also measure the time it takes an MDEK1001 device to send 30 and 157 CIR samples and to wait for the edge device's response. We do so using a Bluetooth Low Energy (BLE) connection with the fastest settings available (i.e., 2 Mbps PHY, MTU size of 244 bytes, and connection interval set to 7.5 ms), as well as using the DW1000 UWB module for communication (with a data rate of 6.8 Mbps and a preamble length of 128). Fig. 9 illustrates the results: sending 30 CIR samples using BLE (BLE 30) takes 8.66 ms, out of which 3 ms uplink and 4.15 ms downlink (note that the use of a connection interval of 7.5 ms introduces some delays and unpredictability in the BLE transmissions). When sending 157 CIR samples using BLE (BLE 157), which would be the typical amount used before introducing the optimizations shown in § 4.2, the total time increases to 16.21 ms, with 7.8 ms for the uplink and 6.9 ms for the downlink transmissions. When using the UWB radio for communication, we have observed that the uplink takes 1.31 ms and 5.71 ms when sending 30 and 157 CIR samples, respectively. While faster than the use of BLE connection, off-loading the NLOS classification and error correction tasks using the UWB module still takes 3.47 ms (UWB 30) and 7.87 ms (UWB 157). This is ≈ 6 times longer than the processing time we measured on the device when running the Tiny XGBoost models. Note that the on-device Small REMNet models have a runtime of 11.5 ms for pre-processing, classification, and error correction, which places them in between BLE 30 and BLE 157.

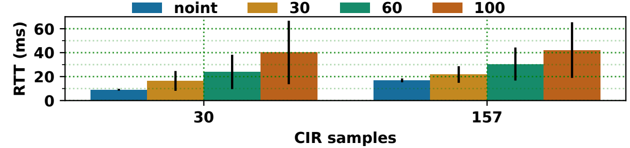

Figure 10: Radio interference can affect the RTT in sending the CIR to the edge device and receiving a response.

Determinism of the on-device delays. The aforementioned delays in off-loading the computations to the edge device have been computed in ideal conditions (i.e., without packet loss). In real-world environments, however, the communication to/from an edge device may be unreliable (e.g., due to the presence of radio interference causing re-transmissions), which results in non-deterministic delays affecting the frequency at which a tag can update its position. Fig. 10 shows the round-trip time (RTT), i.e., the delay between sending the CIR to the edge device and receiving a response, using BLE in the presence of radio interference. In absence of interference (noint), the response time remains stable at around 16 ms and 9 ms when sending 157 and 30 CIR symbols. When adding three nearby Raspberry Pi 3 devices generating traffic on Wi-Fi channel 1, 6, and 11 with a channel occupancy of roughly 30, 60, and 100%, the delay increases by up to 442%, i.e., from 9 ms to 40 ms for a CIR of 30 symbols[12]. More importantly, the standard deviation of this delay increases from 0.78 ms (no interference) to 26.5 ms (strong interference). In contrast, the on-device processing delay is rather deterministic (with a standard deviation of only ≈ 4 µs for XGBoost and ≈ 0.5 ms for REMNet), which makes them suitable for location-aware applications that need to update the position of a device within given time bounds.

7 InSight in Action

We finally evaluate the performance of InSight by quantifying the NLOS classification and error correction performance while running the embedded NLOS engine on mobile tags moving across our testbed area (§ 7.1). We then demonstrate how InSight can be leveraged by a localization application to reduce the errors in the position estimates taken in NLOS conditions by means of an improved anchor selection and ranging error correction strategy (§ 7.2).

7.1 Real-World Performance

We conduct several experiments in the testbed facility described in § 6.1 to evaluate the performance of the proposed on-device NLOS classification and error mitigation in real-world settings. Specifically, we place a mobile tag within two different areas (ENTRANCE and HALLWAY, as displayed in Fig. 8). In both areas, we place the tag in 9 and 5 positions, and let it range to anchors in the ENTRANCE, HALLWAY, and OFFICE for 45 times on average in each position. For each ranging measurement, we record the actual and the estimated distance, as well as the NLOS classification and error correc-

[12] Note that Wi-Fi channels 1, 6, and 11 do not overlap entirely with the 37 BLE data channels; for this reason, even with a channel occupancy close to 100%, BLE transmissions eventually succeed after some retransmissions.

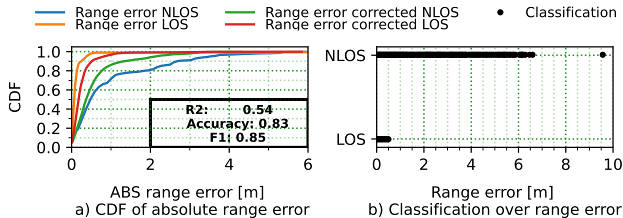

Figure 11: The cumulative absolute error of the range measurements in the the ENTRANCE and HALLWAY (a). All measurements classified as LOS have an error below 0.5 m (b).

tion returned by `InSight`'s NLOS engine[13].

Performance. Overall, an accuracy of 83 % is achieved for classification and an R^2-score of 0.54 for error correction. Fig. 11(a) shows the cumulative distribution function (CDF) of the absolute error for LOS and NLOS measurements. When measuring in LOS conditions, the median and 90th percentile error is 6.2 cm and 20.3 cm. Measurements taken under NLOS conditions are less accurate and have a median error of 53 cm and a 90th percentile error of 271.8 cm. When applying error correction, the median error for NLOS measurements is reduced to 39.7 cm and the 90th percentile to 137.6 cm. The error of LOS measurements slightly increases, resulting in a median and 90th percentile error of 19.30 cm and 53.5 cm. In § 7.2.3, we will show how these results translate in an overall increase of the localization accuracy when leveraging `InSight` in a practical application.

False negative errors. We further investigate if the NLOS classifier can reliably detect NLOS measurements in the presence of large errors. Fig. 11(b) shows the classifier's output for each sample as a function of the range error. We can observe that no measurement with an error above 0.5 m was wrongly classified as LOS, which means that false negatives have a small impact on localization systems.

7.2 Increasing the Localization Performance

We show next how the NLOS classification and error correction returned by `InSight` can be leveraged by an UWB-based localization system to reduce the errors in the position estimates. To this end, we introduce three strategies that leverage `InSight`'s NLOS classification (§ 7.2.1) and/or error correction results (§ 7.2.2), and perform a measurement campaign in our testbed to show the performance of each strategy in terms of localization accuracy (§ 7.2.3).

7.2.1 *Refining the Anchor Selection*

For 2D localization, a mobile tag measures the distance to at least three nearby anchors nodes. The position of the mobile tag is at the intersection of three (or more) circles, with the radius being the distance between the tag and the anchors and the circle's center being the anchor's position. Selecting a good set of anchors from all available options is crucial and impacts localization accuracy, as spatial distribution matters and also NLOS conditions affect the system's accuracy. We hence let a localization application use two strategies: a BASELINE approach that only considers the anchors' spatial distribution (i.e., without knowledge of whether they are in LOS/NLOS), and a NEAREST LOS approach that also leverages `InSight`'s NLOS classification results.

BASELINE. When using this strategy, the anchors are selected in two steps. First, based on an initial position estimate, each nearby anchor is assigned to one of four quadrants. Second, the nearest anchor within each quadrant list is selected and used to perform localization. This strategy is used in Qorvo's localization system and detailed in [9].

NEAREST LOS. This strategy extends BASELINE by removing anchors classified as NLOS from the quadrants' list, such that the nearest LOS anchor is chosen (if any). If no LOS anchor is available in one quadrant, the latter is not considered when performing localization. This is an iterative process: a tag first estimates its distance to the nearest anchor in each quadrant, and then performs NLOS classification on the performed ranging measurement. If NLOS is returned, the tag selects the next anchor from the same quadrant list.

7.2.2 *Correcting the Ranging Error*

The NEAREST LOS strategy fails when all anchors in at least two quadrants are not in LOS with the tag. We hence also let the application investigate two strategies leveraging `InSight`'s error correction functionality.

CORRECTION. This strategy uses the BASELINE anchor selection strategy and always applies error correction regardless of LOS or NLOS conditions.

DETECTION & CORRECTION. This strategy extends the aforementioned CORRECTION strategy by only applying error correction if the measurement is classified as NLOS.

7.2.3 *Results*

We evaluate the localization accuracy of the NEAREST LOS, CORRECTION, as well as DETECTION & CORRECTION strategy, and compare it to the BASELINE anchor selection in our testbed. To give a detailed analysis, we divided the testbed into six areas (Area I – Area VI) containing different experimental setups for LOS and NLOS, as shown in Fig. 12. To increase the number of NLOS conditions in some areas, we deactivate or place some obstacles in front of some anchors.

Localization in harsh NLOS conditions. Area I, III, and V show large localization errors (90th percentile error up to 1.77 m) in Fig. 12(b), (d), and (f), respectively, due to measurements through concrete walls and around corners[14]. To enforce a NLOS condition around the corner, we deactivate anchors 7 and 10 when measuring in Areas III and V. Consequently, the BASELINE strategy selects anchor 6 from Area III and anchor 11 from Area V, while selecting anchors 27-29 in the OFFICE from Area I. Compared to BASELINE, the NEAREST LOS, CORRECTION, and DETECTION & CORRECTION strategies significantly reduce the localization errors, i.e., by up to 1.6, 1.04, and 1.1 m, respectively.

We conclude that, under harsh NLOS conditions, all strategies are viable options to improve the localization accuracy.

Localization in mild NLOS conditions. Area II and IV show small localization errors (90th percentile error up to 0.26 m)

[13] For this evaluation, we opt to relax the runtime requirement of the error correction model and increase it to 600 μs, as we observe a 17% increase in R^2-score and hence a better LOS error correction performance.

[14] In fact, some of the BASELINE measurements (blue upside down triangles) taken in Area V appear in Area IV due to the large localization error.

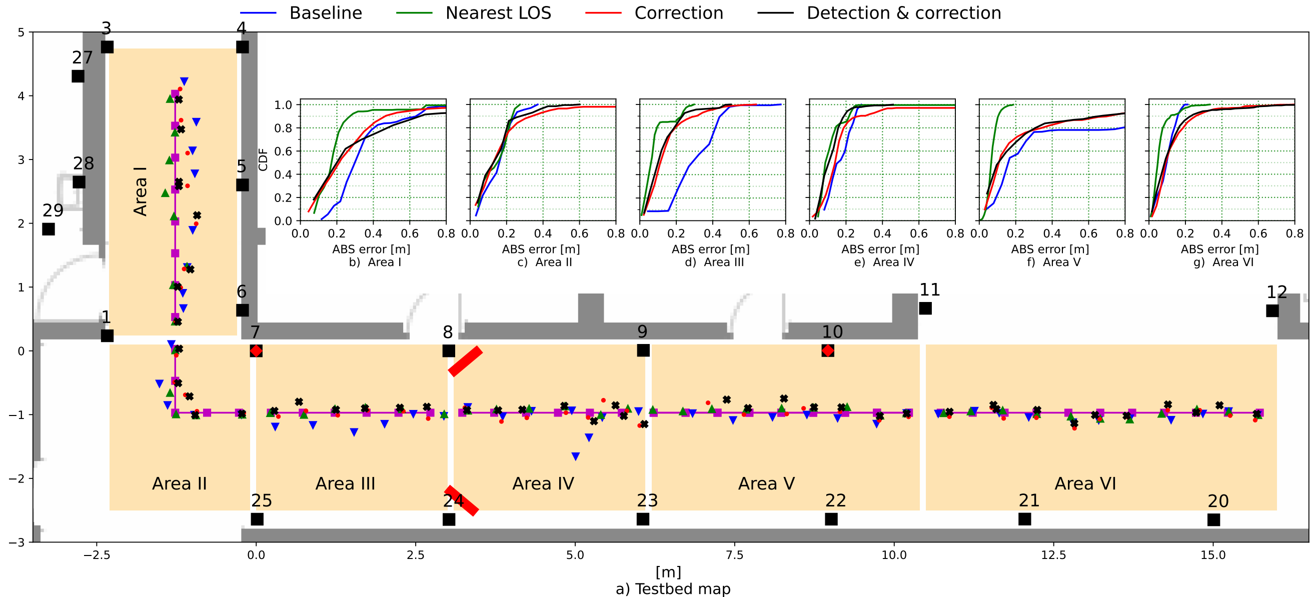

Figure 12: Black squares are the anchors, magenta rectangles mark the ground truth position, upside down blue triangles are the mean position of the BASELINE method, green triangles are the mean position of the NEAREST LOS strategy, red dots are the mean position of the CORRECTION strategy, and black *X* are the mean position of the DETECTION & CORRECTION strategy.

in Fig. 12(c) and (e), respectively, and are caused by obstacles. Specifically, in Area II, anchor 25 is partly blocked by a metal rail, and causes similar errors as the obstacles in Area IV. In Area IV, all strategies improve the localization error (up to the 80th percentile), before the CORRECTION strategy introduces errors. In Area II, all strategies improve or have a similar localization error (up to the 80th percentile), before the CORRECTION and the DETECTION & CORRECTION strategy introduce errors.

We conclude that, under mild NLOS conditions, the NEAREST LOS always improves or maintains the localization accuracy, and that CORRECTION as well as DETECTION & CORRECTION can introduce errors above the 80th percentile (but otherwise also improve or keep the localization accuracy).

Localization in LOS conditions. Area VI shows minor localization errors (90th percentile error up to 0.16 m) in Fig. 12(g) and is recorded in LOS without any obstacles or walls. Surprisingly, rangings to anchor 12 introduce a small error and lead to a better performance of the NEAREST LOS strategy, which avoids this anchor, compared to the BASELINE strategy (4.3 cm at 90th percentile). The CORRECTION and DETECTION & CORRECTION strategies worsen the localization error by up to 9.5 cm at the 90th percentile.

We conclude that, under LOS, the CORRECTION and DETECTION & CORRECTION strategies decrease performance, whereas the NEAREST LOS strategy can improve the localization error even in erroneous LOS conditions.

Takeaway. Our experimental results summarized in Fig. 12 show that, in average, the accuracy of the location estimates across all areas is improved by 59% (i.e., from 0.26 cm to 0.11 cm), with improvements of the 90th-percentile localization error by up to 1.6 m – confirming the effectiveness of `InSight` in tackling real-world NLOS conditions.

8 Conclusions and Future Work

We have presented `InSight`, a framework that enables the deployment of NLOS classification and error correction models *directly* on resource-constrained UWB devices, showing its effectiveness through comprehensive testbed experiments. We integrated `InSight` into Contiki-NG on top of the popular Qorvo `MDEK1001` platform and made it available open-source, hoping that this will foster the creation of location-aware UWB applications sustaining a high performance despite harsh environmental conditions. We plan to leverage `InSight` to study and benchmark the performance of different anchor selection techniques, and to evaluate the performance of the developed lightweight ML models in different environments and with different UWB platforms.

Acknowledgements

The authors would like to thank the authors of [1] for sending their source code. This work has been partially supported by the FFG, Contract No. 881844, “Pro²Future”, through the ENHANCE-UWB project (“Benchmarking and Advancing Localization and Communication Performance of UWB Systems in Harsh Environments” – MFP 4.1.3). This work was also partially executed within the SPiDR project (“Secure, Performant, Dependable, and Resilient Wireless Mesh Networks”) financed by the Technology Innovation Institute. This work was also supported by the TRANSACT project. TRANSACT (`https://transact-ecsel.eu/`) has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement no. 101007260. This joint undertaking receives support from the European Union’s Horizon 2020 research and innovation programme and Austria, Belgium, Denmark, Finland, Germany, Poland, Netherlands, Norway, and Spain. TRANSACT is also funded by the Austrian Federal Ministry of Transport, Innovation and Technology under the program “ICT of the Future” (`https://iktderzukunft.at/en/`).

9 References

[1] S. Angarano et al. Robust UWB Range Error Mitigation with Deep Learning at the Edge. *Engineering Applications of AI*, 102, 2021.

[2] J. Blank et al. pymoo: Multi-Objective Optimization in Python. *IEEE Access*, 8, 2020.

[3] J. Borràs et al. Decision Theoretic Framework for NLOS Identification. In *Proc. of the 48th VTC Conf.*, 1998.

[4] K. Bregar et al. Improving Indoor Localization Using Convolutional Neural Networks on Computationally Restricted Devices. *IEEE Access*, 6, 2018.

[5] K. Bregar et al. UWB Radio-Based Motion Detection System for Assisted Living. *Sensors*, 21(11), 2021.

[6] H. Brunner et al. Understanding and Mitigating the Impact of Wi-Fi 6E Interference on Ultra-Wideband Communications and Ranging. In *Proc. of the 21st IPSN Conf.* IEEE, 2022.

[7] C.-C. Chang et al. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3), 2011.

[8] P. Corbalán et al. Chorus: UWB Concurrent Transmissions for GPS-like Passive Localization of Countless Targets. In *Proc. of the 18th IPSN Conf.* ACM, 2019.

[9] Decawave. DWM1001 System Overview And Performance, 2018.

[10] EETimes. VW and NXP Show First Car Using UWB To Combat Relay Theft, 2019. [Online] https://tinyurl.com/2o1ybh88y5e – Last access: 2023-04-30.

[11] J. Fontaine et al. Edge Inference for UWB Ranging Error Correction Using Autoencoders. *IEEE Access*, 8, 2020.

[12] M. Gallacher et al. Towards NLOS Ranging Error Detection and Mitigation using Machine Learning on Embedded Ultra-Wideband Devices. In *Proc. of the 19th EWSN Conf., poster session*, 2022.

[13] B. Großwindhager et al. SnapLoc: An Ultra-Fast UWB-Based Indoor Localization System for an Unlimited Number of Tags. In *Proc. of the 18th IPSN Conf.*, 2019.

[14] İ. Güvenç et al. NLOS Identification and Weighted Least-Squares Localization for UWB Systems Using Multipath Channel Statistics. *EURASIP J. Adv. Signal Process.*, 2007.

[15] Y.-H. Jo et al. Accuracy Enhancement for UWB Indoor Positioning Using Ray Tracing. In *Proc. of the IEEE/ION PLAN Symposium*.

[16] B. Jones et al. Tiny but Mighty: Embedded Machine Learning for Indoor Wireless Localization. In *Proc. of the 20th CCNC Conf.*, 2023.

[17] Kinexon GmbH. Official Match Ball With Integrated Tracking Sensor. [Online] https://bit.ly/3VyQWDT – Last access: 2023-04-30.

[18] A. Ledergerber et al. A Robot Self-Localization System using One-Way UWB Communication. In *Proc. of the IROS Conf.*, 2015.

[19] Y. Liu et al. Measuring Distance using Ultra-wideband Radio Technology Enhanced by Extreme Gradient Boosting Decision Tree (XGBoost). *Automation in Construction*, 126, 2021.

[20] M. Malajner et al. UWB Ranging Accuracy. In *Proc. of the IWSSIP Conf.*, 2015.

[21] S. Maranò et al. NLOS Identification and Mitigation for Localization Based on UWB Experimental Data. *IEEE JSAC*, 28, 2010.

[22] Markets and Markets. Ultra-Wideband Market. [Online] https://bit.ly/41Qx7JV – Last access: 2023-04-30.

[23] V. A. Minh Le et al. Human Occlusion in UWB Ranging: What Can the Radio Do for You? In *Proc. of the 18th MSN Conf.*, 2022.

[24] M. Ramadan et al. NLOS Identification for Indoor Localization using Random Forest Algorithm. In *Proc. of the 22nd WSA Workshop*, 2018.

[25] C. L. Sang et al. Identification of NLOS and Multi-Path Conditions in UWB Localization Using Machine Learning Methods. *Applied Sciences*, 10(11), 2020.

[26] J. Schroeder et al. NLOS Detection Algorithms for Ultra-Wideband Localization. In *Proc. of the 4th WPNC Workshop*, 2007.

[27] M. Stahlke et al. NLOS Detection using UWB Channel Impulse Responses and Convolutional Neural Networks. In *Proc. of the ICL-GNSS Conf.*, 2020.

[28] M. Stahlke et al. Estimating TOA Reliability With Variational Autoencoders. *IEEE Sensors*, 22(6), 2022.

[29] M. Stocker et al. Performance of Support Vector Regression in Correcting UWB Ranging Measurements under LOS/NLOS Conditions. In *Proc. of the 4th CPS-IoTBench Workshop*, 2021.

[30] M. Stocker et al. Applying NLOS Classification and Error Correction Techniques to UWB Systems: Lessons Learned and Recommendations. In *Proc. of the 6th CPS-IoTBench Workshop*, 2023.

[31] S. Sung et al. Accurate Indoor Positioning for UWB-Based Personal Devices Using Deep Learning. *IEEE Access*, 11, 2023.

[32] V. Tran et al. DeepCIR: Insights into CIR-based Data-driven UWB Error Mitigation. In *Proc. of the IEEE/RSJ IROS Conf.*, 2022.

[33] Wired. The Biggest iPhone News Is a Tiny New Chip Inside It, 2019. [Online] https://bit.ly/41OymcV – Last access: 2023-04-30.

[34] H. Wymeersch et al. A Machine Learning Approach to Ranging Error Mitigation for UWB Localization. *IEEE TCOM*, 60, 2012.

[35] J. Xin et al. A Bayesian Filtering Approach for Error Mitigation in Ultra-Wideband Ranging. 19(3), 2019.