Tiny Neural Networks for Environmental Predictions: an integrated approach with Miosix

Francesco Alongi, Nicolò Ghielmetti
DEIB, Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
alongi.francesco96@gmail.com
nicolo.ghielmetti@gmail.com

Danilo Pau, FIEEE
System Research and Applications
STMicroelectronics, Italy
danilo.pau@st.com

Federico Terraneo, William Fornaciari
DEIB, Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
federico.terraneo@polimi.it
william.fornaciari@polimi.it

***Abstract*—Collecting vast amount of data and performing complex calculations to feed modern Numerical Weather Prediction (NWP) algorithms require to centralize intelligence into some of the most powerful energy and resource hungry supercomputers in the world. This is due to the chaotic complex nature of the atmosphere which interpretation require virtually unlimited computing and storage resources. With Machine Learning (ML) techniques, a statistical approach can be designed in order to perform weather forecasting activity. Moreover, the recently growing interest in Edge Computing Tiny Intelligent architectures is proposing a shift towards the deployment of ML algorithms on Tiny Embedded Systems (ES). This paper describes how Deep but Tiny Neural Networks (DTNN) can be designed to be parsimonious and can be automatically converted into a STM32 microcontroller-optimized C-library through X-CUBE-AI toolchain; we propose the integration of the obtained library with Miosix, a Real Time Operating System (RTOS) tailored for resource constrained and tiny processors, which is an enabling factor for system scalability and multi tasking. With our experiments we demonstrate that it is possible to deploy a DTNN, with a FLASH and RAM occupation of 45,5 KByte and 480 Byte respectively, for atmospheric pressure forecasting in an affordable cost effective system. We deployed the system in a real context, obtaining the same prediction quality as the same DNN model deployed on the cloud but with the advantage of processing all the necessary data to perform the prediction close to environmental sensors, avoiding raw data traffic to the cloud.**

***Index Terms*—Tiny Machine Learning, Neural Networks, Embedded Systems, Real Time Operating Systems, Edge Computing**

I. Introduction

Although predicting weather is an activity with a long history and the methods used have improved over time, it still remains a challenge for meteorologists to finely perform local predictions based on NWP, which predicts weather conditions using mathematical models of the atmosphere and the oceans. Indeed, these models need a very high computational effort, since they are composed by the complex partial differential equations which capture how the phenomena affects the atmosphere. Moreover, in order to increase the prediction resolution, both in space and time, NWP centers need an even greater computational effort, that leads to a rapid increase in power consumption and in the number of computing resources needed to run such calculations. For this reason, vertically centralized high-performance computing technologies will likely reach a bottleneck to scale computational tasks (such as NWP), thus mandating decentralized approaches in which scalability will be achieved by distributing the computational load across multiple distributed heterogeneous computer architectures [1] running different families of methods, such as ML algorithms. Since ML does not necessarily require a complete understanding of the underlying physical processes [2], it enables alternative solutions to an analytical approach or physical modelling which are too time consuming, costly or sometimes impossible to develop [3]. With our experiment we adopted a distributed architecture (e.g., Edge Computing) embedding Tiny ML, trying to take advantages from both of them. In order to ease and better integrate the Tiny ML algorithm at the edge of the system we selected Miosix, a configurable, tiny, open and real time operating system described in [4]: this choice enabled the design of a solution that exploits multi-threading and process multi-sensors data patterns. The result of this implementation is a scalable and power-efficient system.
Our project aims to illustrate the architectural solution just introduced by means of a proof-of-concept demonstrator: in particular, in order to provide a coarse-grain estimation of the weather, we considered atmospheric pressure as a parameter on which to elaborate predictions. The fields of application and the target markets of this work, with a sufficient resolution granularity and the appropriate extensions, could be: renewable energy, smart agriculture, insurance and aviation. For instance, a useful application of this project could be the prevention of hydro-geological instability in risk areas by deploying a swarm of DTNN sensing units on the field.
This paper describes the design and the implementation of a DTNN, intended for predicting the atmospheric pressure. It describes how to build the DTNN using a popular deep learning framework, then automatically convert these networks into an efficient C library with X-CUBE-AI software-tool and integrating the library with Miosix running on a tiny STM32 microcontroller. This work also demonstrates that the performances estimated during the design and training phases of the DTNN are coherent with respect to the ones measured

during the deployment in a real context. The paper is structured as follows: in Section II it reviews some existing approaches. In Section III it discusses the system architecture of the experimental setup. In Section IV it describes the process which lead to the final topology of the DTNN. In Section V are described all the activities required to generate the C library for the STM32 microcontroller. In Section VI it presents the experimental results and close with the conclusions in Section VII.

II. Related works

Weather prediction and forecasting is an activity with a long history, predating that of computers [5]. However, the availability of computers made for the first time feasible to perform on-line simulations based on atmospheric models [6]. Among the approaches to weather prediction, the numerical approach models the atmospheric dynamics. Due to the large amount of input data and the even greater computational complexity, high-performance computer architectures and supercomputers are often employed [7], with increasing processing capability used to improve resolution. Other approaches to weather prediction include the integration with big data [8], fuzzy logic [9] as well as machine learning [10]. Weather prediction has also been implemented in embedded systems, such as [11], but the work is focused on weather monitoring purposes and not on weather forecasting. An existing work that implements in an ES a weather forecasting system is [12], which adopts ML techniques such as Decision Trees and ARIMA models for computing the prediction, whilst the experimentation shown in [13] demonstrates the possibility to perform weather forecasting using a LSTM, unfortunately it does not deploy the ML model into a MCU (MicroController Unit).

III. System Architecture

The system architecture we have designed is sketched in Figure 1.

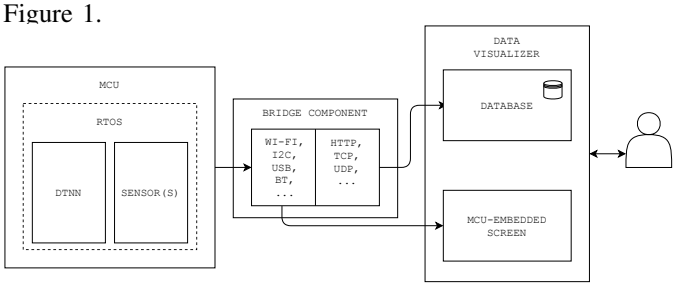

Fig. 1. System architecture diagram.

The system consists of 3 main components: a STM32 MCU, a data visualizer and a bridge component used to forward data from the MCU to the data visualizer (Figure 1). The MCU is the component that acquires the measurements of interest so that a prediction can be computed. We selected the MCU family produced by STMicroelectronics, widely used by the mass market community. Moreover, this family is supported by a tool that automatically translates the pre-trained DNN model into an STM32 microcontroller-optimized C-library that also exploits at best the available FLASH, RAM and clock cycles. In general, it is required the availability of a peripheral to export the computed outputs, to connect the sensors and, optionally, the easy compatibility with a RTOS so that the scalability of the system can be achieved.
In order to get data, the MCU uses one or more sensors capable of transducing a physical dimension into an electronic one that will be subsequently processed and fed to a DTNN.
The bridge component is an interface with the aim of making possible the communication between the MCU and the data visualizer. While the latter is intended to display the data sent by the MCU in a human readable format, the bridge component can be stand-alone or it can be embedded inside the MCU itself; if the MCU has any integrated mean to display the data, the interface in this case is identified by the protocol that is needed to enable the communication between the MCU with the data visualizer (e.g., LCD screen). Otherwise the bridge component is identified as the interface for remote communication (e.g., WiFi, Ethernet, Bluetooth).

A. Experimental setup

The objective of this project is to provide atmospheric pressure forecasting in a short-range setting with an affordable low cost system. The predictions are performed over the following convenient time divisions of the day:

- Time band 1 (pressure measurements acquired from 00:00 to 07:59);
- Time band 2 (pressure measurements acquired from 08:00 to 15:59);
- Time band 3 (pressure measurements acquired from 16:00 to 23:59).

The system acquires pressure measurements and aggregate them in order to have only one mean value for each time band. Moreover, the system forecasts the average pressure value of a time band given the average values of the previous three time bands.
Considering our use-case, the components mapped with respect to the Figure 1 are as follows:

- Sensor(s): since our project aims to perform weather forecasts based on atmospheric pressure, the sensor used is the LPS22HB, which is an absolute pressure sensor. This sensor samples at a given frequency both the atmospheric pressure and the environment temperature at the same time, providing a single 40-bit output: the first 24 bit are reserved to the pressure, while the other 16 bit are for the temperature. These 40-bit outputs are pushed back into a FIFO with 32 slots embedded in the sensor as soon as they are sampled. In our project, both of these values are useful: the pressure measurements are used to perform predictions and the temperature values are used to compute the sea-level atmospheric pressure starting from the absolute one. Both the pressure and the temperature are casted to 32 bit floating point (FP32). The sampling frequency of the sensor has been set to 1Hz, which is the lowest one available. Being the sensor mounted on an expansion board containing multiple sensors driven by

an I2C module, we developed an I2C driver to retrieve data from the considered sensor using C++ and STL data structures;

DTNN: atmospheric pressure variations are basically univariate time series. For this reason, we decided to implement a Tiny Deep Neural Network using LSTM cells, which are commonly used for time series data processing. We used a single neural network with 3 FP32 inputs, one for each average pressure value associated with a timeband;

MCU: for our project we chose a NUCLEO-F401RE based on the MCU STM32F401RET6, which is an ARM cortex M4 with 512 KB of FLASH memory and 96KB of SRAM. Through a multi threading mechanism, enabled by Miosix [14], and the use of interrupts, we have ensured that the DTNN and the sensor data acquisition are synchronized and power-efficient during the execution;

Bridge component: since the MCU chosen for our project does not have natively any integrated data display device, we sent the measured data and predictions made by the DTNN to a Rapsberry Pi via USB, which in turn, using a Python script, forwards them to the data visualizer. This approach was used in order to log the gathered data and validate the performance of the prototype; in a real deployment the data could be consumed locally in the microcontroller;

Data visualizer: in order to graphically display and perform some analysis on the performance of our system we used an open source time series database: InfluxDB 2.0 Cloud. In this way it has been possible to create intuitive dashboards through Grafana and run queries to get the stored data.

The diagram in Figure 2 summarizes how the threads interact in the program that was installed in the NUCLEO-F401RE FLASH memory for demonstrator development.

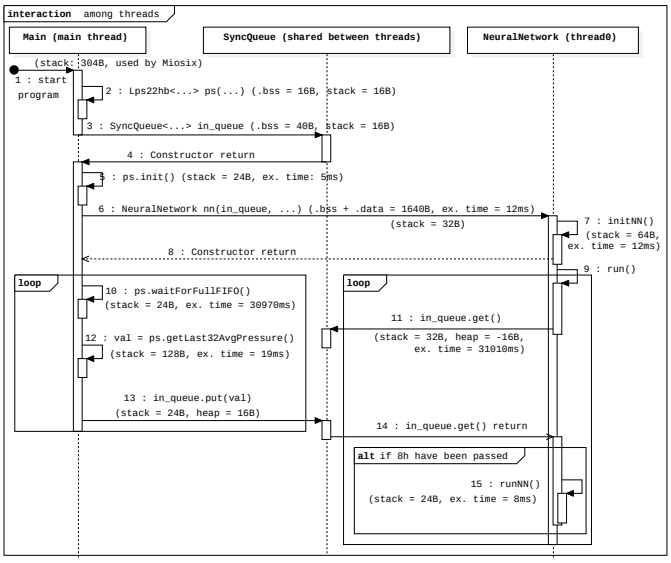

Fig. 2. Sequence diagram of the principal OS actors

In the sequence diagram shown in Figure 2 are reported the memory usage and the execution time of each major function. Cases where metrics are not specified have to be considered less than 1Byte and 1ms respectively.

In our implementation, two threads were used: one with the role of "producer", i.e. the one dedicated to the querying of the I2C module to read atmospheric pressure data from the sensor, and the other with the role of "consumer", i.e. the one dedicated to the processing of "produced" data, then used to make the prediction through the DTNN. The implemented threads communicate in a synchronized way thanks to the SyncQueue object, that is a shared queue implemented using mutexes and condition variables from the Miosix API. In a first step, the main thread instantiates and initializes the objects associated with the atmospheric pressure sensor (i.e. LPS22HB), the synchronized queue and the neural network. As shown in Figure 2, since they are all global, they produce an increment over the .bss or .data sections or both (so they affect the object file produced by the compiler) but they do not impact over the heap or stack. After this phase, the program proceeds through two parallel and independent loops: the one inside the main thread exploits the LPS22HB hardware interrupts mechanism which "awakens" the main thread notifying it of the data availability. This condition occurs when the FIFO embedded in the sensor contains 32 values (i.e. the FIFO is full).

As specified before, the sampling rate of the sensor is set to 1Hz, so the main thread has to wait for approximately 32s until the FIFO full condition is satisfied. As shown in timing profile information, which can be found in Figure 2, the actual sampling frequency is faster than the one expected: in fact the FIFO fills up completely in 30,97s. Once the FIFO full condition is satisfied, the main thread empties the FIFO by reading the contained data and makes the arithmetic mean over the 32 values with a FP32 precision. At this point, the main thread compute the sea-level pressure starting from the arithmetic mean of the absolute one (i.e. the one acquired by the LPS22HB sensor and therefore contained in the FIFO). The Equation 1 shows how to find the sea-level pressure P_{sl} from the absolute pressure P_a (both in hPa), where h is the sensor altitude in meters and T is the measured temperature in $°C$.

$$P_{sl} = P_a \left(1 \quad \frac{0{:}0065 \quad h}{T + 0{:}0065 \quad h + 273{:}15}\right)^{5{:}257} \tag{1}$$

The derivation of the Equation 1 can be found in [15].

Once the sea-level pressure has been computed, it is inserted by the main thread into the syncronized queue. The loop within the neural network thread waits for data to be inserted into the synchronized queue using the get() blocking function (i.e. if there is no data inside the queue, the thread which invoked the function, in this case the neural network one, is put on hold). Also for this function we can see a high value for the execution time; this is because the neural network thread has to wait for the sea-level pressure to be inserted in the synchronized queue.

Once the main thread inserted data into it, the thread synchronization mechanism implemented within the queue "awakens" the neural network thread that was waiting for them.
When the necessary data are obtained, the pressure incremental mean is updated and if 8 hours have been passed since the last prediction, it runs the runNN() function to produce a new prediction.
The Figure 3 summarizes the flow of the data.

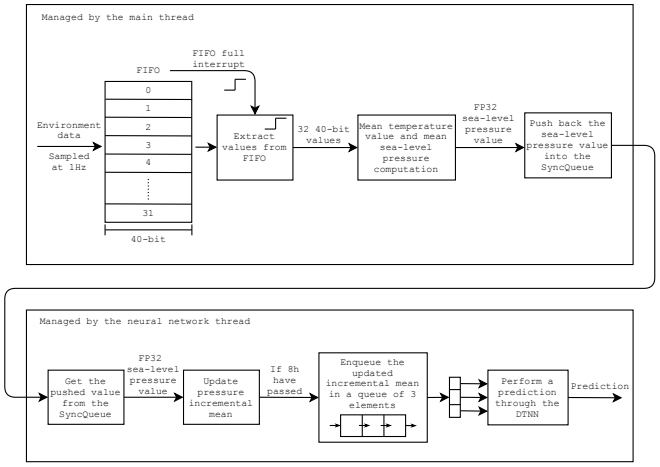

Fig. 3. Pipeline of the data: from the environment to the DTNN

IV. Deep Tiny Neural Network design

In the context of Tiny ML, models are intended to be deployed on systems with limited hardware resources (e.g., STM32). The workflow related to the design of such models follows a different path from the usual one for unconstrained neural networks. Indeed, in addition to the typical Data preparation - Training - Validation - Test - Deploy workflow, that assumes unconstrained computing resource availability, in this case it is useful to integrate a productive tool (e.g., X-CUBE-AI) that, starting from the pre-trained neural network model, creates a resource parsimonious version of the neural network. This result is achieved by considering MACC (Multiply-and-accumulate complexity i.e. a unity that indicates the complexity of a Deep Learning model from a processing standpoint), RAM and FLASH dimensions, which define the overall complexity of the model analyzed by X-CUBE-AI, both considering the chosen microcontroller target and the type of input network. In this way, the resulting workflow considers also the complexity as a variable upon which decide the model that best fit for the considered use-case. There are several open-source deep learning frameworks that can be used to design and implement neural network models. For instance, Keras [16] and TensorFlow [17] are widely used by the ML community, therefore we decided to adopt these frameworks to implement our DTNNs. Moreover, to optimize and fit it into STM32, X-CUBE-AI latest version 5.0.0 was used.
The flow above mentioned is depicted in Figure 4.

Starting from the top-left of the proposed workflow, we will describe in the next subsections all the stages of the pipeline,

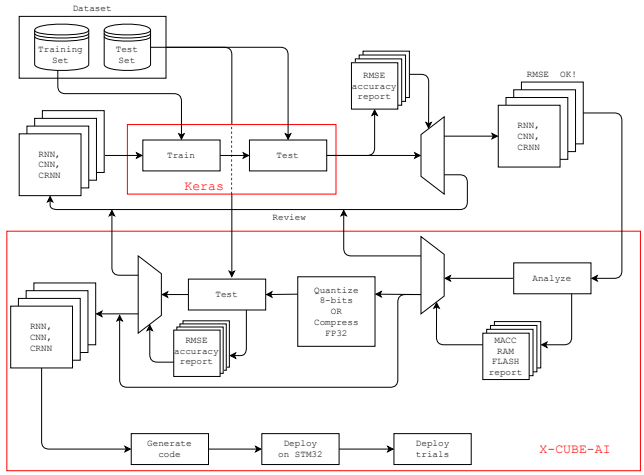

Fig. 4. Project flow

considering the particular case of our proof-of-concept system.

A. Data preparation

The dataset has been obtained from a certified weather station in a comma-separated values file. Since the original format of the dataset included more features than the ones used for this work, we had to go through a data preparation phase. Initially, the dataset contained most of the data needed for a professional weather forecasting service, including humidity, dew point, temperature, wind speed and direction other than timestamp and atmospheric pressure, which are the ones we kept for demonstration development. Once the features to use had been chosen, we aggregated the resulting dataset in timebands (as defined in Section III-A) using an arithmetic mean. Then, the aggregated dataset, containing 19939 samples, has been split into train, validation and test set with the ratios shown in Figure 5. However, since the chosen dataset is a time series, it is impossible to randomly pick samples from the starting one in order to create the validation and test set because, for time series, the temporal order of the data must be preserved. For this reason we decided to arrange the splitting in such a way to have the test set posterior to the validation set, and the latter posterior to the training set. Having defined the order of the three partitions and the splitting ratios, there is only one possible combination of the train/validation/test split. Finally, the three datasets have been normalized with a feature range of $[0; 1]$ via a min-max normalization, using for all the datasets the normalization parameters obtained from the train set.

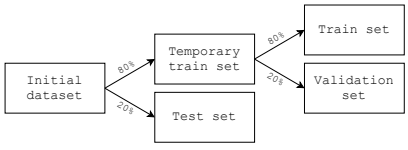

Fig. 5. Dataset split ratios

The three datasets have been fed to the neural network using a sliding window of size 3 for the input and of size 1 for the

target. For instance, in order to test the performance of the model, we predict the value at time $t+1$ feeding to our model the values at time t 2, t 1 and t, and the obtained prediction is evaluated with the truth value at time $t+1$. It shall be noted that $+1$ represents a prediction of 8 hours timeframe.

B. Tiny Deep Neural Network development

In the loop which involves the train and test stages, different neural network models have been explored, with the aim of finding the best performing model. The baseline from where we started the process was guided by the common knowledge of time series forecasting through neural networks. Indeed, we decided to start from a Recurrent Neural Networks (RNN) such as LSTM and GRU, which are often used for time series processing [18]. Moreover, we considered a mixed architecture involving both CNNs and recurrent neural networks, which have been successfully applied in several applications concerning time series analysis [19]. In this first loop we trained and evaluated four different families of model (i.e. LSTM, GRU, CNN-LSTM and CNN-GRU), considering the best performing model for each family. We performed the design space exploration on the 4 families of models chosen by varying the number of layers/filters and units, considering a total of 22 different models. The details of the chosen DTNN models per each family are shown in the Table I. The performances of the models have been evaluated in terms of Normalized Root Mean Square Error (NRMSE) and Normalized Mean Absolute Error (NMAE), which have been computed using Keras. The NRMSE and NMAE are defined as follows:

$$NRMSE = \frac{RMSE}{X_{max} \quad X_{min}} \; ; \quad NMAE = \frac{MAE}{X_{max} \quad X_{min}}$$

Where X_{max} and X_{min} are the normalization parameters obtained at training time.

In particular, the most performing model topology per each family, shown in Table I, has proved to be the following ones:

LSTM family: 2 LSTM cells with 70 units per cell, interleaved by Dropout layers of 20% drop rate;

GRU family: 2 GRU cells with 50 units per cell, interleaved by Dropout layers of 20% drop rate;

CNN-LSTM family: 1 Conv1D layer with 32 filters followed by 2 LSTM cells with 30 units;

CNN-GRU family: 1 Conv1D layer with 8 filters followed by 2 GRU cells with 30 units;

TABLE I
PERFORMANCE OF THE CHOSEN MODELS PER EACH FAMILY

	NRMSE	NMAE
LSTM	0.0255	**0.0193**
GRU	**0.0253**	**0.0193**
CNN-LSTM	0.0324	0.0255
CNN-GRU	0.0321	0.0240

V. IMPLEMENTATION ON STM32 MICROCONTROLLER

Once the performance of the models has been evaluated, we decided to proceed our study with the most performing families, which performances are highlighted in bold in the Table I: LSTM and GRU. Since the goal of this work was to create a DTNN that could achieve satisfying performance while keeping the complexity low, we tried to find a hyperparameter setting that could represent an even more convincing trade-off between performance and complexity to explore more opportunities for even tinier networks. The complexity of the models in terms of MACC, FLASH and RAM metrics have been computed using the "Analyze" functionality of X-CUBE-AI. In addition this tool enables other optimization that can be performed over the pre-trained model: Quantize to 8-bits and Compress FP32 (only applicable to CNN part of the topology). We scaled the LSTM and GRU families down starting from the hyperparameter setting considered in Table I, validating performance and complexity of all the configurations of the models taken into account. Our process of complexity reduction took place by equally decreasing the number of units in both cells, both for the LSTM and GRU family. The number of units decreased at each complexity reduction step has been set to 10. The Table II shows the metrics computed in the validated models of the LSTM family while the Table III shows the metrics of the GRU models taken into account. For readability reasons, in both tables are reported the models with a decreasing units step equal to 20. Given the metrics computation results shown in Table II and in Table III, we decided to integrate into our project the LSTM family with 2 cells and 30 units per cell. This decision was driven by the fact that we had prior knowledge of this architecture and additionally because this configuration was tested more than the others, so we felt more confident in deploying this architecture. Moreover, the size of this network, considered in terms of FLASH & RAM leaves plenty of available storage to Miosix and other application specific needs. In addition, on the deployed model, another representative measure of the network performance from a computational standpoint was computed: the cycles/MACC, resulting, in average for all layers of the DTNN, in 14.67. Once the model has been chosen, the STM32 microcontroller-optimized C-library has been generated using the "Generate code" functionality of X-CUBE-AI. In order to actually deploy our final Embedded System in the environment, we needed to integrate the code provided by X-CUBE-AI with Miosix so that we could implement the scalable and multi-threaded solution described in Section III.

TABLE II
DETAILS OF THE LSTM MODEL COMPLEXITY REDUCTION

	MACC	FLASH(KB)	RAM(B)	NRMSE	NMAE
70 Units	179410	234.89	1116	0.0255	0.0193
50 Units	92150	120.90	800	0.0257	0.0197
30 Units	33690	44.42	480	0.0255	0.0194
10 Units	4030	5.43	160	0.0262	0.0198

TABLE III
DETAILS OF THE GRU MODEL COMPLEXITY REDUCTION

	MACC	FLASH(KB)	RAM(B)	NRMSE	NMAE
50 Units	68600	91.02	800	0.0253	0.0193
30 Units	24960	33.52	480	0.0256	0.0196
10 Units	2920	4.14	160	0.0264	0.0200

VI. Experimental results

In order to evaluate the performance and the mid-long term operability of the Embedded System in its integrity, it has been deployed in a real context for 30 days, from the 1st of March 2020 to the 31st of March 2020. The predictions performed by our system are shown together with the real atmospheric pressure values in Figure 6. In order to have also a numerical measurement of the performance of our system, the NRMSE and the NMAE have been computed from the data obtained during this live-testing session, resulting equal to 0.0328 and 0.0251 respectively. This errors are slightly greater from the one obtained during the validation phase (i.e., 0.0255 and 0.0194, Table II 30 Units) but still comparable. This is due to the fact that the atmospheric pressure variations are a local phenomenon and the system has been deployed in a location which is approximately 50 km from where the data used to train the DTNN have been acquired. The deployed system is shown in Figure 7.

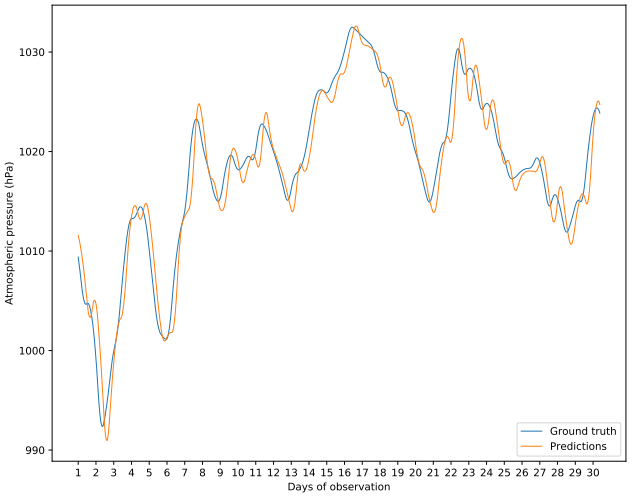

Fig. 6. Performance of the deployed system

Raspberry Pi

Fig. 7. Experimental setup

VII. Conclusion and future works

In the paper we introduced an embedded weather prediction system using a DTNN. The system achieved an adequate accuracy (RMSE = 2.10 hPa) considering the project's fields of application. In addition to presenting the solution, we also abstracted from our specific use-case, formalizing a general methodology based on X-CUBE-AI for projects aimed to develop DTNNs. The present research has highlighted some limitations that will be addressed as future works. Among these, the automatic identification problem of the model's hyperparameters remains an open problem. Other future developments include increasing the performance of the chosen model through the integration of other sensors. A further ambitious extension of this project could be to coordinate different units such as the one presented in this paper in order to produce a further analysis and/or prediction based on all the predictions made by the individual units.

References

[1] Bauer, Peter & Thorpe, Alan & Brunet, Gilbert. (2015). "The quiet revolution of numerical weather prediction." Nature. 525. 47-55. 10.1038/nature14956.

[2] Holmstrom, Mark, Dylan Liu, and Christopher Vo. "Machine Learning Applied to Weather Forecasting." Stanford University. 2016

[3] S. Akhtari, F. Pickhardt, D. Pau, A. D. Pietro and G. Tomarchio, "Intelligent Embedded Load Detection at the Edge on Industry 4.0 Powertrains Applications," 2019 IEEE 5th International forum on Research and Technology for Society and Industry (RTSI), Florence, Italy, 2019, pp. 427-430.

[4] Brandolese,C., Fornaciari,W. Rucco,L. and Terraneo, F. "Enabling ultralow power operation in high-end wireless sensor networks nodes." In Proc. of CODES+ISSS '12. ACM, New York, NY, USA, 433-442.2012.

[5] Wiston, M., and Mphale, K. M., Weather forecasting: From the early weather wizards to modern-day weather predictions. Journal of Climatology & Weather Forecasting 6(2):1–9, 2018.

[6] P. Lynch, "The origins of computer weather prediction and climate modeling", 2008 Journal of Computational Physics, pp. 3431 - 3444.

[7] U. Gartel, W. Joppich and A. Schuller, "Medium-range weather forecast on parallel systems," Proceedings of IEEE Scalable High Performance Computing Conference, Knoxville, TN, USA, 1994, pp. 388-391.

[8] P. C. Reddy and A. S. Babu, "Survey on weather prediction using big data analystics," 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, 2017, pp. 1-6.

[9] A. S. Aisjah and S. Arifin, "Maritime weather prediction using fuzzy logic in java sea," 2011 2nd International Conference on Instrumentation Control and Automation, Bandung, 2011, pp. 205-208.

[10] N. L. and M. H.S., "Atmospheric Weather Prediction Using various machine learning Techniques: A Survey," 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 2019, pp. 422-428.

[11] Lajara, Rafa & Alberola, Jorge & Pelegri-Sebastia, Jose & Sogorb, T. & Llario, J. Vicente. (2007). Ultra Low Power Wireless Weather Station. 469 - 474. 10.1109/SENSORCOMM.2007.4394965.

[12] Kailasanathan, Nallakaruppan & Kumaran, Senthil. (2019). IoT based machine learning techniques for climate predictive analysis. International Journal of Recent Technology and Engineering. 7. 171-175.

[13] D. N. Fente and D. Kumar Singh, "Weather Forecasting Using Artificial Neural Network," 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), Coimbatore, 2018, pp. 1757-1761, doi: 10.1109/ICICCT.2018.8473167.

[14] Martina Maggio, Federico Terraneo, Alberto Leva. (2014). Task Scheduling: A Control-Theoretical Viewpoint for a General and Flexible Solution. ACM Transactions on Embedded Computing Systems (TECS). 10.1145/2560015.

[15] World Meteorological Organization (2012), Other business: pressure reduction formula. CIMO/ET-Stand-1/Doc. 10 (20.XI.2012)

[16] F. Chollet and others, Keras io, 2015

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015.

[18] Petneházi, Gábor. (2018). Recurrent Neural Networks for Time Series Forecasting.

[19] N. Xue, I. Triguero, G. P. Figueredo and D. Landa-Silva, "Evolving Deep CNN-LSTMs for Inventory Time Series Prediction," 2019 IEEE Congress on Evolutionary Computation (CEC), Wellington, New Zealand, 2019, pp. 1517-1524.