CRIME: Input-Dependent Collaborative Inference for Recurrent Neural Networks

CRIME: Input-Dependent Collaborative Inference for Recurrent Neural Networks

Daniele Jahier Pagliari, *Member, IEEE,* Roberta Chiaro, *Member, IEEE,* Enrico Macii, *Fellow, IEEE,* and Massimo Poncino, *Fellow, IEEE*

Abstract—The excellent accuracy of Recurrent Neural Networks (RNNs) for time-series and natural language processing comes at the cost of computational complexity. Therefore, the choice between edge and cloud computing for RNN inference, with the goal of minimizing response time or energy consumption, is not trivial. An edge approach must deal with the aforementioned complexity, while a cloud solution pays large time and energy costs for data transmission. Collaborative inference is a technique that tries to obtain the best of both worlds, by splitting the inference task among a network of collaborating devices. While already investigated for other types of neural networks, collaborative inference for RNNs poses completely new challenges, such as the strong influence of *input length* on processing time and energy, and is greatly unexplored.
In this paper, we introduce a *Collaborative RNN Inference Mapping Engine* (CRIME), which automatically selects the best inference device for each input. CRIME is flexible with respect to the connection topology among collaborating devices, and adapts to changes in the connections statuses and in the devices loads. With experiments on several RNNs and datasets, we show that CRIME can reduce the execution time (or end-node energy) by more than 25% compared to any single-device approach.

Index Terms—Edge computing; recurrent neural networks; collaborative inference; deep learning; intelligent offloading; long short-term memory

1 Introduction

In recent years, deep learning (DL) has gained enormous success in a variety of application domains, consistently outperforming traditional machine learning methods [1]. The spread of DL has been partially fueled by the proliferation of IoT sensors and smart devices generating massive amounts of data [2], [3]. Indeed, many IoT and mobile applications can be enhanced thanks to DL-based algorithms. In particular, deep Recurrent Neural Networks (RNNs) such as those based on the Long-Short Term Memory (LSTM) architecture are commonly used for natural language and audio processing in smart personal devices [4], [5], for elaborating inertial [6] and biometric [7] sensor signals, as well as for processing time-series in smart cities, smart energy and smart manufacturing tasks [8], [9].

Unfortunately, the previously unimaginable accuracy reached by DL-based solutions comes at the cost of computational and memory complexity, both in the training and inference phases. To meet such computational requirements, a common approach is to offload these expensive tasks to high-performance cloud infrastructures equipped with GPUs and multi-core CPUs. However, pure cloud-computing raises several issues [10]. First, sending raw data to the cloud over a network link may yield long response latency in case of slow or intermittent connections. Second, this centralized approach is not scalable, as it imposes a lot of stress on the network infrastructure. Third, most smart and IoT devices are connected wirelessly, meaning that raw data offloading to the cloud is also expensive in terms of energy consumption. Lastly, processing in the cloud may arise concerns with user data privacy.

In view of these issues, edge computing turns out to be a convenient alternative, especially for the *inference* phase, and could lead to several benefits in terms of responsiveness, energy efficiency and security [2], [10], [11]. Research has been ongoing to deploy optimizations that allow to sustain the computational burden of deep learning inference on resource-constrained edge devices.

Many studies have focused on hardware accelerator designs, which exploit the parallelism of the matrix multiplication kernels that dominate Deep Neural Networks (DNN) inference, and leverage techniques such as weight quantization and pruning to reduce the memory bottleneck [12]. After initially targeting mostly Convolutional Neural Networks (CNNs), recent research is trying to extend DNN acceleration also to RNNs [13], [14], [15], [16], [17].

While accelerators are extremely time- and energy-efficient, most low-budget mobile and IoT systems cannot afford a dedicated inference hardware, and must rely on general purpose embedded CPUs. For these systems, despite the excellent latency and energy reduction results achieved by model compression techniques such as quantization [18], [19] and pruning [20], [21], performing inference entirely at the edge might still be sub-optimal. Indeed, it has been ascertained that, for general purpose devices, even better results in terms of latency and energy consumption can often be achieved by *splitting* the inference computation among edge and cloud devices [22], [23]. Such splitting can be applied in principle on top of any model, including both "vanilla" and compressed ones [23]. This approach, some-

- *D. Jahier Pagliari, R. Chiaro and M. Poncino are with the Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy, 10129. E-mail: {name.first_surname}@polito.it*
- *E. Macii is with the Interuniversity Department of Regional and Urban Studies and Planning, Politecnico di Torino, Turin, Italy, 10129. E-mail: enrico.macii@polito.it*

Manuscript received January XX, XXXX; revised January XX, XXXX.

times called *collaborative inference*, allows greater degrees of freedom in tuning the trade-off between the latency and energy overheads of transmitting data to the cloud and those deriving from a local computation. Different approaches to collaborative inference have been proposed [22], [23], [24], [25], [26], [27] but only focusing on feed-forward DNNs and CNNs. In contrast, collaborative inference for RNNs is an almost uncharted territory.

Nonetheless, recurrent networks pose several novel problems. For instance, as explained in Section 3, there is no benefit in *partitioning* these networks (e.g. at the level of layers) between multiple devices. Instead, the best approach is to execute the entire network in one of the collaborating devices. Moreover, since RNN inference complexity depends on input data size, taking input-driven offloading decisions is fundamental.

In this work, which extends [28], we present a *Collaborative RNN Inference Mapping Engine (CRIME)*, able to select the most suitable device for RNN inference among those available in a collaborative network, with the aim of minimizing the total inference execution time (or energy consumption). The mapping is driven by multiple factors: the input size, the current connection status (latency, bandwidth) and the computational resources of candidate devices, affected by their current workload. Results based on different RNNs and datasets show that CRIME can significantly reduce the total execution time or energy consumption by more than 25% with respect to any single-device inference approach.

With respect to the preliminary work presented in [28], this paper introduce the following main novel contributions:

- We extend our proposed collaborative inference framework for RNNs to support an arbitrary number of devices and interconnections (e.g. end-nodes, intermediate edge gateways and cloud servers).
- We propose a method to propagate variations in the network connection speed or in the computational load of individual devices at runtime.
- We describe in detail the distributed optimization performed by each node of a CRIME network and evaluate its effectiveness on an extended set of scenarios and applications.

The rest of the paper is organized as follows. Section 2 provides the needed theoretical background and reviews the state-of-the-art overview of collaborative inference techniques. Section 3 provides the motivation for our method, while Section 4 presents the details of CRIME. Experimental results are presented in Section 5, while Section 6 draws conclusions and suggests possible future work.

2 Background and Related Works

2.1 RNN Inference

Recurrent neural networks (RNNs) are DNNs specifically designed to solve sequential problems, whose distinctive feature is the presence of temporal relations between sequence elements. RNN architectures are therefore characterized by feedback loops, that allow information from prior steps to persist while processing sequences of data. The basic layers of RNNs are the so-called *cells* or *units*. There exist several fundamental types of cell, including the "vanilla" RNN cell, the Gated Recurrent Unit (GRUs) and the Long-Short Term Memory (LSTM). Both GRUs and LSTMs are designed to overcome the limitations of standard RNNs in handling long term dependencies [1].

Here and in the rest of this paper, we focus our discussion on LSTMs due to their widespread usage in many application domains. However, all fundamental considerations and hence our mapping strategy, can also be applied to GRUs and other variants. The functionality of a LSTM cell is governed by the following equations:

$$\begin{aligned} i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\ g_t &= \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{1}$$

In this equation x_t is the input vector at time t, while h_t and c_t are called *hidden state* and *cell state* respectively, and are fed-back to the cell in the next time-step, as shown in Figure 1a. The details of LSTM functionality are out of the scope of this work, and readers can refer to [1].

From a computational standpoint, (1) shows that each LSTM inference step is characterized by the same set of operations. In particular, the dominant kernels are large matrix-vector multiplications involving weight matrices W, followed by non-linear operations (hyperbolic tangent $\tanh$ and sigmoid σ). In practice, when performing inference on an input sequence of length n, the LSTM cell is unrolled (i.e. replicated) n times, as shown in Figure 1b for $n = 4$. Each replica shares the same weight matrices, learned during training, and computes the same operations. The outputs of the last cell of the chain (h_4 and c_4 in the figure) provide an encoded representation of the input sequence and are typically fed to a classifier, such as a fully-connected NN.

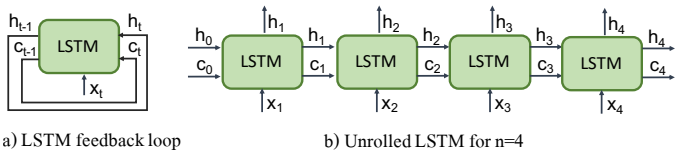
Given (1), it is evident that RNN/LSTM computational complexity grows linearly with the input length n. Moreover, although the cell performs a highly-parallel kernel, parallelism among different time-steps is limited: each replica cannot start the elaboration until it receives the outputs from the previous step [14], [15]. As a consequence, inference execution time in RNNs can also be expected to grow linearly with input length.

A similar reasoning applies to energy consumption: since each LSTM cell execution involves the same operations, the power consumption of the hardware performing inference can be assumed constant throughout the process, especially for a single-task system (e.g. a smart sensor). Consequently, energy should also grow linearly with n.

Notice that *batching*, i.e. parallel processing of multiple independent input sequences, while fundamental during training, is not normally used for latency-sensitive inference tasks, where response time is critical. The latter requires a *streaming* approach, where individual sequences are processed as soon as they are available.

2.2 Related Works

Optimizations to perform DNN inference totally or partially at the edge are very actively researched. Excellent recent



a) LSTM feedback loop b) Unrolled LSTM for n=4

Fig. 1. A LSTM cell with feedback loops (a) and its unrolled version for $n = 4$ (b).

surveys can be found in [2], [3], [10], [29].

One of the most popular approaches to implement RNN inference fully at the edge is through custom hardware accelerators, implemented on ASIC or FPGA technology. These designs optimize the most energy-intensive operations involved in the inference process (multiply-and-accumulate loops), generally using SIMD or systolic architectures [13], [14], [15]. Among others, [16], [17] propose respectively a LSTM-based and a GRU-based accelerator for FPGA, while [30] introduces ESE, a framework to accelerate compressed sparse LSTM models obtained by pruning parameters. Alternatively, offloading to mobile-GPUs can also be leveraged to accelerate edge inference [31]. In this perspective, [32] proposes GRNN, a GPU-based library, for serving RNN models with low latency, high scalability and efficient resource utilization, while BatchMaker [33] enables cell-level batching for RNNs with variable length inputs.

However, most mobile and IoT systems cannot afford dedicated inference hardware and must rely on general purpose embedded CPUs, much more limited in terms of performance and energy efficiency. These limitations can be partially mitigated by means of techniques that trade-off accuracy and computational demand. The most significant example is quantization, i.e. replacement of floating point data and operations with low-precision integers [18], [19], which simultaneously impacts the DNN memory footprint and access costs, as well as its arithmetic complexity. Being relatively straight-forward to apply on general purpose hardware, quantization (down to 8-bit precision) is currently supported by many inference libraries for edge CPUs [34]. Unfortunately, quantization of RNNs/LSTMs is less effective than for feed-forward networks [35]. *Pruning* [20], [21] is another popular technique (potentially combinable with quantization [36], [37]) which allows decreasing the model size and, depending on the underlying hardware capabilities, its latency and energy consumption, by removing redundant weights from DNNs. Finally, approximate computing techniques [38] can also help in reducing the complexity of DNNs with limited impact on accuracy. The downside of these techniques it that they are not easy to apply on general purpose CPUs, where they might not assure a sufficient speed-up and energy efficiency. For instance, pruning often negatively affects hardware utilization in Single-Instruction Multiple-Data (SIMD) CPUs, and complicates the memory access patterns for inference [39].

Because of these observations, researchers have started to devise hardware-independent accuracy vs complexity optimizations, working at the DNN model level. Most notably, [19], [40], [41], [42], [43], [44], [45] propose *adaptive* approaches, in which a different DNN model (or version thereof) is selected depending on the complexity of the processed input. Models can be completely separate, they can share part of the network architecture (as in the early-exit mechanism of [46]) or they can be even built by changing the parameters of a common architectural blueprint. In particular, [43], [44], [45] target RNNs/LSTMs. The former two tune the beam search width used in encoder/decoder networks at runtime, based on a measure of input difficulty, while the latter skips some LSTM cells entirely based on input similarity.

Another promising set of techniques to optimize inference on edge devices is edge caching. This method involves caching DNN inference results, locally or in the edge server, and leveraging the reusability of the previous outputs based on the similarity of the input. [47], [48], [49]

While hardware compression techniques and model optimizations can reduce the energy consumption and latency of DNN processing by several orders of magnitude [20], performing inference fully on-edge might still not be the most efficient choice. *Collaborative inference* was developed as an orthogonal research branch, aiming to find the optimal combination of local processing and cloud offloading. Collaborative and distributed execution of generic tasks has been a long studied discipline [50], [51]. However, DL-based inference tasks have peculiar characteristics from a computational standpoint, such as the possibility of easily estimating the execution time for a given input and network architecture, thanks to the deterministic and data independent amount of computations involved in each layer [22], [23], [24], [28]. This calls for task-specific optimizations, which have been demonstrated capable of yielding largely superior results compared to generic approaches [22], [24].

One of the first works in this field is found in [22]. The authors propose a framework, called Neurosurgeon, that intelligently decides where to partition a CNN layer-wise, while accounting for connectivity conditions. Their observation is that CNN feature sizes shrink for deeper layers in the model. Therefore, computing a few layers on the edge and then sending the last computed feature tensor to the cloud can reduce latency and energy overheads deriving from wireless transmission.

A similar approach is proposed in BottleNet [23], but additionally, the network architecture is modified to favor partitioned execution. Specifically, a reduction unit is added after the layers assigned to be computed on the edge, to compress the output sent to the cloud, where a decompressor (restoration unit) restores the original tensor before continuing the computation. At training time, compression and decompression are approximated by an identity function.

JointDNN [24] extends the previous approaches to consider multiple split points, for DNNs where feature sizes are not monotonically decreasing, such as autoencoders. The authors map the problem of partitioning the computations in a DNN to shortest path finding in a graph, and provide an ILP formulation to constrain the resolution with limitations imposed by the edge device battery or by the cloud congestion. The authors of [26] combine layer-wise partitioning with an inference early-stopping mechanism to further speed-up the computation.

IONN [52] considers the scenario in which different devices in a distributed system do not store a copy of the same DNN model. The authors introduce a partitioning-

based offloading in which portions of layers' weights are transmitted to the cloud together with data, which allows to start a partial execution before the entire DNN model is uploaded. In [53] an improvement over IONN is proposed, based on a finer-grain uploading plan.

Layer-wise partitioning is not the only possibility. For example, DeepThings [25] splits each convolutional layer of a CNN into multiple parallel execution tasks to be distributed among edge node devices in a load balancing manner. The authors of [54] propose another form of partitioning, specific for applications that gather data from multiple IoT sensors. Their *hierarchical* inference algorithm uses a properly designed DNN architecture, which allows to decompose the first layers into slots, each one involving only the subset of features available on a given end-device. In this way, sensors themselves perform partial inference based on the data they can access locally, partially relieving the cloud of the computational burden, and send a low-sized amount of data to servers, with a decreased communication effort.

Similarly, collaborative systems are not limited to two levels (edge and cloud). For example, the authors of [27] propose a partitioning strategy that involves three offloading levels (end-devices, edge servers and cloud), and also consider the interaction between multiple independent and concurrent inference tasks.

To the best of our knowledge, existing collaborative inference strategies focus only on feed-forward networks, in particular on CNNs. Our work is the first specifically tailored for RNNs, which, as detailed in Section 3, require a completely different collaborative strategy. Indeed, merely extending a CNN-based solution to RNNs would not yield any benefit. Instead, what our method has in common with other collaborative solutions is being *fully orthogonal* to single-device optimizations (such as quantization, pruning, etc.) and freely combinable with them.

3 Motivation

Existing works on collaborative inference for feed-forward DNNs only propose *input-independent* policies. In other words, the partitioning point (e.g. a layer) may vary at runtime depending on connection status, cloud server load, etc, but it is not influenced by the processed datum. This is reasonable, given that the great majority of feed-forward DNNs and CNNs can only process *fixed-size* inputs, hence having a fixed computational complexity (i.e. number of MAC operations) per input.

In contrast, as seen in Section 2.1, RNN inference complexity grows linearly with input length. Therefore, an offloading strategy for these networks should consider the input length as a fundamental decision parameter. In the rest of this section, we first discuss why *partitioning* a single RNN inference among multiple devices is sub-optimal. Then, we discuss the potential benefits of mapping entire inferences onto a network of collaborating devices, in an input-dependent way.

3.1 Sub-optimality of RNN model partitioning

A key difference between feed-forward DNNs and RNNs is that, for most RNN-based tasks, *partitioning* of the inference phase is not beneficial. In contrast, the offloading decision should map the *entire* inference on one of the collaborating devices. When illustrating this, we refer to a slightly more detailed diagram of a 2-layer LSTM, shown in Figure 2, where the possible ways in which the inference execution could be partitioned are shown by dashed lines.

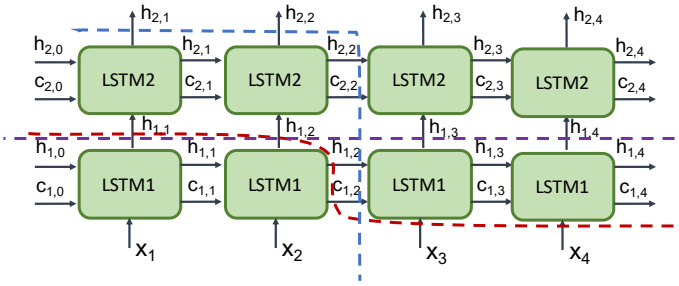


Fig. 2. A 2-layer LSTM network unrolled for $n = 4$, with three possible partitioning solutions shown by dashed lines: layer-wise (purple), time-wise (blue) and mixed (red).

The key unfavorable factor for partitioning is that RNN inputs are typically small, and often also smaller than intermediate outputs. As an example, inputs to Natural Language Processing (NLP) applications are sequences of word indexes in a vocabulary. Even if the latter is large (say 10k words), each index can be encoded with 2 bytes, and an entire sentence/paragraph, i.e. the typical inference input, requires few 100s of bytes. Similarly, for time-series processing applications [4], [8], inputs at each time-step are vectors of few tens of features. A *single* LSTM hidden/cell state vector, instead, is typically composed of few 100s of float elements per time-step [1].

Due to these small data sizes, the transmission time for inference offloading tends to be dominated by data-independent components [58]. To explain why, we use the following standard transmission time model, which is the same used by the proposed CRIME framework:

$$T_{tx,ij}(n) = T_{rtt,ij} + \frac{S(n)}{B_{ij}} \tag{2}$$

where B_{ij} is the *bandwidth* of the network connection between two devices i and j, and $T_{rtt,ij}$ is the connection *round-trip time*. $S(n)$ is the size in bytes to be transmitted, which in general depends on the processed input length n. This also includes the transmission of results (e.g. class labels) in the opposite direction.

As mentioned above, RNN inputs can be encoded with few hundreds of bytes. Assuming $S(n) \approx 100B$ and a relatively slow LTE link with $B_{ij} = 2$Mbps, the fractional term in (2) becomes 0.4ms. A typical value of $T_{rtt,ij}$ for LTE is around 70ms, i.e. ≈200x larger [58]. So, for most RNN inference applications, data offloading is bound by the round-trip time, which is virtually independent of the sequence length n. This is very different from what happens in CNNs [23], where both inputs and intermediate layer output sizes are orders of magnitude larger (e.g. even a small 256x256 RGB image requires ≈ 200kB).

These observations show why RNN inference partitioning is sub-optimal. That is, even if a partial local processing could yield some amount of data compression, the impact of

such compression on the overall transmission time $T_{tx,ij}(n)$ would be negligible, as the latter is roughly independent of the data size. In other words, with respect to directly offloading the inputs, the additional time and energy costs for partial local processing would be *wasted*, since they would not contribute to reducing the subsequent cost for transmission.

This is true both for layer-wise partitioning (the purple line in Figure 2) as well as for time-wise partitioning (blue line), and for mixed approaches (red line). In particular, the layer-wise and mixed approaches are almost surely sub-optimal, since, as mentioned, intermediate layer outputs are often larger in size compared to inputs. The compression ratio of the time-wise approach, instead, depends on a number of parameters, including the input size and length, the LSTM hidden size and its number of layers. However, regardless of the ratio, this compression could have an impact on the total transmission time only for very low-latency connections, which are not realistic for IoT applications.

3.2 Potential benefits of input-dependent collaborative inference for RNNs

Despite the fact that RNNs partitioning is not beneficial, it is still possible to apply the principles of collaborative inference to these networks, by mapping *entire* inferences to a given device. The main technical challenge is that such mapping should be performed in an adaptive way with respect to the current input, due to the analysis of Section 2, as well as to the current computational load of the various devices and to the current speed of the network connection. Figure 3 shows visually the potential benefits of collaborative input-dependent RNN inference, using a 3-level system as an example. To obtain the graphs, we have measured the inference execution time of a popular LSTM architecture [55] on three different computational platforms, representative of typical end-nodes (e.g. a smartphone), edge gateways and cloud servers. The characteristics of the LSTM and of the devices are reported in the caption, and detailed in Section 5. For each device, the figure shows the average inference time for a given length (dots), the variability measured by the execution time standard deviation (shaded areas), and a linear regression model fitted on the data (lines). Additionally, the measurements relative to the gateway and server have been moved upwards to take into account the additional transmission time for offloading over a network, e.g. a WWAN link between end-node and gateway, and a multi-hop Internet connection between gateway and cloud.

The regression scores reported in the caption, confirm experimentally the analysis of Section 2.1 on the linear dependence of RNN inference execution time with respect to input length. Clearly, the dependencies have different slopes due to the different performance of the corresponding device. Data transfers, instead, affect the models' intercepts.

In its entirety, Figure 3 motivates our work. In fact, it shows that to minimize the total execution time for this particular example, inputs of length < 4 should be processed locally by the end-node, while lengths between 4 and 11 should be offloaded to the gateway, and longer inputs should be processed in the cloud.

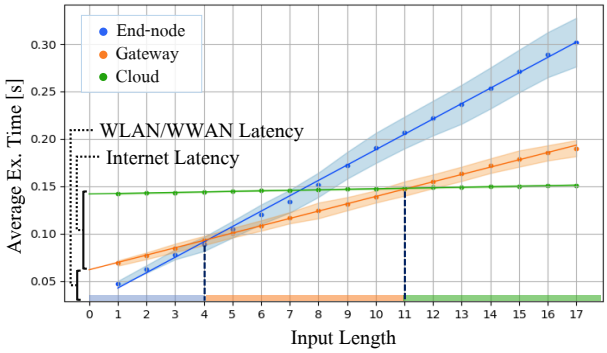


Fig. 3. Inference execution time versus input length for the CoVe network [55], measured on exemplificative processing devices for end-nodes (ARM Cortex A53), edge gateways (NVIDIA Jetson TX2) and cloud servers (NVIDIA Titan XP). Points and colored areas represent means and standard deviation intervals over 100 measurements. Lines are linear regression fits. Regression scores: edge $MSE = 6.08 \cdot 10^{-}4$, $R^2 = 0.997$; gateway $MSE = 6.98 \cdot 10^{-}5$ $R^2 = 0.998$; cloud $MSE = 1.45 \cdot 10^{-}6$ $R^2 = 0.985$

4 Collaborative RNN Inference Mapping

Based on the observations of Section 3, a smart collaborative strategy aiming at minimizing the overall inference time should execute RNNs on different devices depending on their relative performance, on the current speed of the communication links connecting them, and on the processed input's length. We propose CRIME, a *fully distributed* strategy to perform this selection, in which each device performs an independent decision to compute or to offload the current input, based on *local estimates* of the processing speed of all reachable devices and of the corresponding network links. Initially, we formulate our optimization with the goal of minimizing the *total inference execution time*. Later, in Section 4.4 we show that, with few adaptations, CRIME can also be applied to minimizing sensor nodes *energy consumption*.

4.1 Collaborative System Model

An overview of the underlying model used in CRIME is illustrated in Figure 4. As shown, we model a collaborative inference system as a Directed Acyclic Graph (DAG), $G(V, E)$ where nodes $V = \{v_i\}$ represent devices, and edges $E = \{e_{ij}\}$ represent the presence of a network connection between v_i and v_j. The DAG has a single *source* node v_0 representing the device (sensor, smartphone, smart speaker, etc.) where input data are gathered.

Our formulation does not require to directly consider the presence of multiple data sources. This is because, as explained below, the effect of other sources feeding inputs to the system is indirectly taken into account as a variation of the devices load. Similarly, CRIME does not impose any restriction on the topology and connectivity of the graph, but for the absence of cycles. The most realistic topology, and therefore the one considered in our experiments, is depicted in Figure 4; here, nodes are organized in *levels* and edges are only present between nodes in level l and $l + 1$. This is the scenario in which an end-device is connected to one or more intermediate *edge gateways*, which in turn are connected to one or more *cloud servers*. In principle, however, nothing forbids the presence of, for instance, *horizontal* connections between devices at the same level.

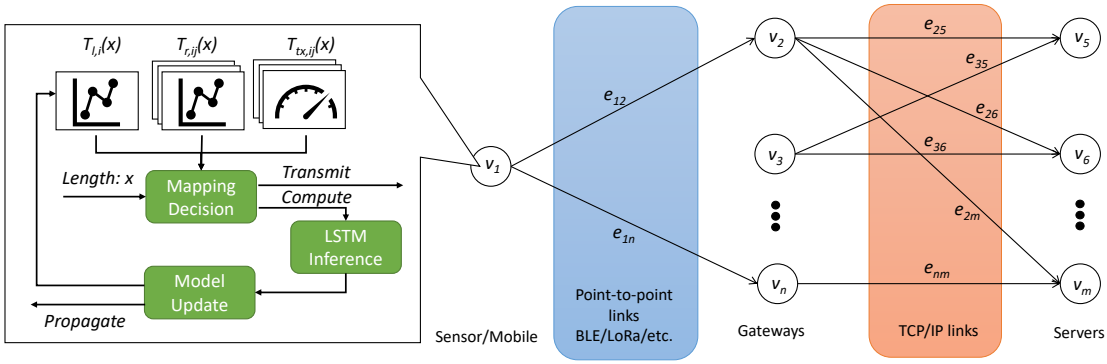


Fig. 4. Overview of the CRIME collaborative inference model, with the details of the main operations performed in a node. Although not shown for visualization purposes, the same operations depicted for the source node are performed in all other nodes.

The pre-conditions of CRIME are as follows. First, as mentioned in Section 3, for each input, the *entire* RNN inference is executed on a single node (i.e. our approach does not involve network *partitioning*). Therefore, each device must maintain a local copy of the RNN model and of its trained weights. In general, thanks to the many tool-independent NN storage formats and conversions tools available, devices can collaborate despite using different inference engines.

Second, each device v_i should maintain three pieces of information needed to perform a mapping decision:

- A *local model* $T_{l,i}(n)$ of its own inference execution time as a function of the length n of the processed input.
- A set of remote models $T_{r,ij}(n)\ \forall j : e_{ij} \in E$ of the inference execution time as a function of n for all directly connected nodes.
- A set of models $T_{tx,ij}(n)\ \forall j : e_{ij} \in E$ of the transmission time as a function of n for all directly connected nodes.

Given the linear dependency of RNN inference complexity with respect to input length, the local model $T_{l,i}(n)$ is obtained with a simple *linear regression*, as in the blue plot of Figure 3. Remote models $T_{r,ij}(n)$ instead, are in general piece-wise linear, as explained in Section 4.2. The total execution time for local inference is therefore simply estimated as:

$$T_{tot,i}(n) = T_{l,i}(n) = \alpha_{l,i} n + \beta_{l,i} \tag{3}$$

In contrast, when computations are offloaded to remote node v_j, the total execution time is estimated as:

$$T_{tot,i}(n) = T_{tx,ij}(n) + T_{r,ij}(n) \tag{4}$$

In turn, the transmission time $T_{tx,ij}(n)$ can be modeled as in (2), where both $T_{rtt,ij}$ and B_{ij} can have very different values depending on the type of link. However, the round-trip latency is dominant in the majority of cases, as explained in Section 3. This is why the gateway and cloud curves of Figure 3 just receive a fixed vertical offset as an effect of data transmission.

In order to select the most appropriate inference device for an input of length n, each node v_i performs the following local optimization. First, the best remote device $v_{\hat{j}}$ is selected as the one that yields the shortest *total* execution time (including transmission and remote processing):

$$\hat{j} = \arg \min_{j:e_{ij} \in E} (T_{tx,ij}(n) + T_{r,ij}(n)) \tag{5}$$

Then, the choice between local inference and offloading is simply performed as:

$$v_{target} = \begin{cases} v_i \text{ if } T_{l,i}(n) \leq T_{tx,i\hat{j}}(n) + T_{r,i\hat{j}}(n) \\ v_{\hat{j}} \text{ otherwise} \end{cases} \tag{6}$$

Notice that, to perform this choice, v_i only needs to have an *estimate* of the time taken by other connected nodes v_j to process the input. It does not need to be aware of *how* the result will be produced. This means that, when node v_i (e.g. a mobile device) selects remote offloading, node $v_{\hat{j}}$ (e.g. a gateway) will receive the input and in turn will decide whether to process it locally or offload it further (e.g. to a cloud server). All of this happens transparently for v_i, so that there is no need for a *global* orchestration of the inference, which would incur additional communication overheads.

4.2 Remote execution time models

Each CRIME node receives remote execution time models $T_{r,ij}(n)$ directly from its *successors* in the graph, i.e. models "flow" in the direction opposite to the graph edges. As an example, consider the DAG of Figure 5a. Nodes v_3 and v_4 are both *leaves* of the DAG. Therefore, when they receive an input, their only available choice is local execution, and the total execution time is simply:

$$T_{tot,3}(n) = T_{l,3}(n),\ T_{tot,4}(n) = T_{l,4}(n) \tag{7}$$

Both v_3 and v_4 will periodically transmit their local models to the common predecessor v_2, which will store them as $T_{r,23}(n)$ and $T_{r,24}(n)$ respectively. The details of this transmission are described in Section 4.3.

Node v_2 therefore has three mapping choices: it can execute the inference locally, or it can offload it either to v_3 or to v_4. In the most general scenario, the three mappings are optimal for different ranges of input length n, as shown in Figure 5b, where the offsets of the red and green curves are due to transmission times. Specifically, in the example the optimal inference node determined using (5) and (6) would be v_2 for $n \leq n_{23}$, v_3 for $n_{23} < n \leq n_{34}$ and v_4 for $n > n_{34}$.

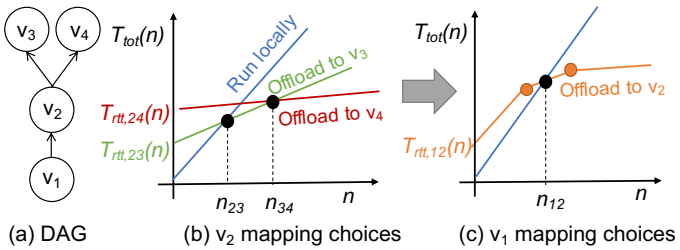


Fig. 5. (a) DAG representing a reference configuration of connected collaborating devices, (b)-(c) regression models associated with mapping choices available respectively for v2 and v1.

To indirectly inform v_1 of its multiple mapping choices, v_2 will transmit to v_1 the *minimum* of the blue, green and red lines of Figure 5b, which v_1 will internally store as $T_{r,12}(n)$. This model will have the piece-wise linear shape of the orange curve in Figure 5c. Here, the curve is further shifted upwards due to the transmission time $T_{tx,12}(n)$, to show the mapping trade-off available to v_1. In particular, the optimal choice for v_1 is to execute locally for $n \leq n_{12}$ and offload to v_2 for $n > n_{12}$.

If v_1 was not the source node of the graph, the minimum of the blue and orange curves of Figure 5 would be further propagated to its predecessors. Importantly, as anticipated in Section 4.1, with this propagation mechanism each node only has to store a limited number of remote models, corresponding to its direct successors, and does not have to be aware of the entire DAG topology nor of the choices performed by other nodes. This also simplifies system configuration changes during runtime. For example, adding a new node to the DAG, e.g. a previously not present intermediate gateway, or a new cloud server for load balancing, only requires informing its direct predecessors.

In CRIME, we assume that RNN parameters (i.e. weights) are loaded once at the beginning of the execution and kept in memory thereafter. This is a realistic scenario, since in IoT end-nodes (e.g. smart sensors, smart speakers, etc.), inference is typically executed as part of a task constantly running in the background, waiting from new inputs (e.g. sensor data, voice recordings, etc.) to be processed. In edge gateways and cloud servers, instead, inference is provided as a *service*, thus the corresponding task must be constantly in execution, listening to new requests from lower-level devices.

4.3 Model updating and propagation

If all execution time models were constant in time, it would be possible to compute the break-even lengths between different mapping choices (n_{12}, n_{23} and n_{34} in Figure 5) just once at design time, and the optimal mapping would reduce to a comparison with these thresholds. Unfortunately this is not the case, which is why storing and propagating models as described in Section 4.2 is needed. Taking again the example of Figure 5b, there are two main elements that can vary over time:

- The transmission times to offload an input to v_3 or v_4, i.e. $T_{tx,23}(n)$ and $T_{tx,24}(n)$, are affected by variations in the link speed.
- The local execution time $T_{l,2}(n)$ is influenced by *other tasks* running on the same device. For example, a server may receive inference requests from multiple devices, which affect its available compute resources and therefore its execution time for a given length n.

When one of these two types of variation happens, the remote models relative to v_2 stored in other nodes, i.e. $T_{r,i2}(n)$, become outdated and must be refreshed.

4.3.1 Transmission time updating

We combine two mechanisms to keep the information about the transition time between nodes up to date. First, every-time node v_i offloads a task to node v_j, both nodes attach time-stamps to the data upon their transmission/reception. With these time-stamps, when results are returned to v_i, the latter can estimate the parameters of (2) autonomously. Given the discussion of Section 3.1, the round-trip latency dominates the overall transmission time. Therefore, although adding these timestamps slightly increases the size of the transmitted data packets, the effect on $T_{tx,ij}(n)$ is negligible.

While the time-stamps mechanism is effective, it is not sufficient, as the transmission time is only updated when v_i offloads a task to v_j. To see why, consider again Figure 5b, and assume that v_2 offloads a task to v_4 during a momentary spike in the network latency, e.g. due to noise or traffic. As a consequence, the estimate of $T_{tx,24}(n)$ stored in v_2 will increase dramatically, i.e. the red curve in the figure will be moved upwards. Depending on the amount of this shift, the break-even length n_{34} might reach a value so high that it never happens in practice. In other words, v_2 will start to assume that offloading to v_4 is never convenient. As a result, even when the network latency decreases again, v_2 will never send tasks to process to v_4, and therefore will never correct its outdated transmission time estimate.

To solve this problem, every time no tasks have been offloaded to a particular destination for a time longer than T_{ping} (for example, more than 1 minute), the source will exchange a *ping* packet (i.e. a packet with no payload) with it, in order to update its round-trip latency estimate. Ping packets are transmitted rarely, in order to minimize their overhead in terms of transmission time and energy. In practice, for realistic graph sizes, the impact of these periodic latency updates on the overall execution time is limited, as shown in Section 5.

4.3.2 Local execution time updating

While IoT sensors are normally single-task systems, mobile devices, edge gateways and cloud servers handle multiple concurrent tasks at the same time, hence they are affected by *load variations*. This might influence the time required to locally process an input $T_{l,i}(n)$. Nonetheless, the dependency between inference time and input length remains linear, as shown in Figure 6.

Different colors represent different load conditions. Points represent the inference time for different input lengths on a real edge computing device, averaged over 100 inferences. Load variations have been simulated executing 1, 4 or 8 inferences in parallel, with randomized lengths orders. As shown, the effect of multiple concurrent processing

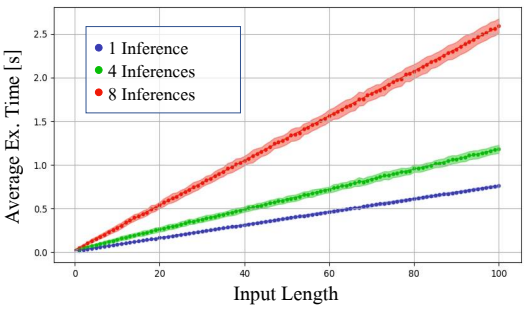


Fig. 6. Execution time versus input length for the CoVe network [55], measured on a NVIDIA Jetson TX 2 when performing a different number of concurrent inferences. Points and colored areas represent means and standard deviation intervals over 100 measurements. Regression scores: single inference $MSE = 5.71 \cdot 10^{-}5$, $R^2 = 0.998$; 4 concurrent inferences $MSE = 1.48 \cdot 10^{-}4$ $R^2 = 0.998$; 8 concurrent inferences $MSE = 6.96 \cdot 10^{-}4$ $R^2 = 0.998$

requests is just a change in the slope of the regression model, not in its shape. Given this observation, we account for load variations by *periodically refitting* the local $T_{l,i}(n)$ estimate. To this end, each node measures the execution time of all inferences performed locally, and stores these values in a sliding window. Once every M_l local inferences, the internal linear regression is updated. Model refitting clearly has an overhead as it involves additional computations. However, linear regression fitting on a small amount (e.g. 10-100) of time measurements is much simpler computationally than RNN inference, hence by appropriately tuning M_l, this overhead can be made negligible while keeping a relatively up-to-date execution time model.

4.3.3 Model propagation

In Sections 4.3.1 and 4.3.2 we have described how CRIME nodes update their *local* execution and transmission time models. These models, however, must be also *propagated* to each node's predecessors. Propagating models incurs an additional transmission overhead. Therefore, we do it only when (i) we can attach this information to another necessary transmission or (ii) the remote models of predecessors differ too much from the local version.

The first case corresponds to the *piggybacking* of model information: whenever v_i offloads an inference to v_j, the latter attaches its new model parameters to the *response* packet containing the inference outcome. The complete response packet format is shown in Figure 7. The first obvious

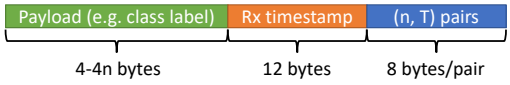


Fig. 7. Format of a data packet sent as a response to offloaded inference tasks in CRIME.

component is the *payload*, i.e. the inference result. This can be a single class label (e.g. a 4-byte integer) for applications such as sentiment analysis, or a sequence of n labels for applications such as question answering. Next, the packet contains the data reception timestamp, used to update v_i's estimate of the transmission time $T_{tx,ij}(n)$, as explained in Section 4.3.1. This can be represented, for example, as an 8-byte integer for the time-from-epoch, plus a 4-byte float to represent fractions of a second. The last element of the packet is the information about v_j's execution time model, i.e. the piece-wise linear curve described in Section 4.2. This is transmitted as a set of $(n, T(n))$ pairs, each represented as $(int, float)$, requiring 8 bytes.

Although these additional information significantly enlarge the total size of the packet, their impact on transmission time is negligible. Indeed, the total size is still only in the order of 10s-100s of bytes. Therefore, as explained in Section 3.1, the size-independent round-trip latency will still dominate the total transmission time.

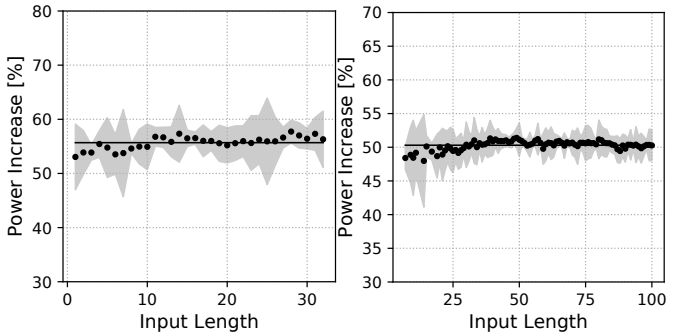
Besides piggybacking, updated models are also *broadcasted* to all predecessors when the difference between the last transmitted version and the current one exceeds a threshold. To this end, each node stores a copy of the latest model transmitted to its predecessors. After each internal update, the node computes the maximum time difference over all input lengths between the new model and predecessors' version. A broadcast is triggered when this value exceeds a threshold T_{th}. Broadcasting is especially useful when a device becomes "not appealing" for some time, e.g. because of a high load. In that scenario, given (5) and (6), predecessors will start to avoid offloading to that device. Therefore, even when its load decreases, the node would not receive inferences, and would not have the opportunity to piggyback its new model. Broadcasting solves this issue and ensures that relevant variations in the speed of a device are propagated through the system.

Notice that, with this scheme, models automatically propagate through multiple levels in the graph (possibly up to the source). In the example of Figure 5, a relevant load variation in v_3 (i.e. in the slope of the green curve of Figure 5b) would be propagated to v_2, This, in turn, would modify the shape of the piece-wise linear curve in Figure 5c. If this modification is greater than T_{th}, v_2 will then broadcast it to its predecessors (only v_1 in this case).

4.4 Energy Optimization

Up to this point, we have considered the task of minimizing the total response time of a collaborative RNN inference system. In this section, we discuss the differences when the goal is *energy minimization*. As noted in previous works [22], [24], energy is a critical goal mostly for battery-operated systems, such as IoT sensors. Therefore, in our work, we focus on minimizing energy only at the *source* node in the DAG of Figure 4.

In Section 2, we have anticipated that RNN inference power consumption should be approximately constant, since each time-step involves the exact same operations. This is confirmed in Figure 8, which shows the average power consumption for different input lengths when running two different RNNs. Power measurements are performed with a digital multimeter (HP 34401A) attached to a thermally stable shunt resistor, with a period of 1s, running 1000 inferences per each value of n. The target device is the same used as "end-node" in Figure 3. The graphs show the power increment in percentage with respect to the baseline consumption of the system, measured when the CPU is idle, with unused peripherals disabled. Power values in Watts are



(a) CoVe [55] (b) IMDB [56]

Fig. 8. Power consumption versus input length for two RNNs. Points and colored areas represent means and standard deviation intervals over 1000 inferences. The solid line is the best constant fit of the data. Baseline power: 1.81W, CoVe average increment: 1.00W, MSE = $0.44 \cdot 10^{-3}$W (a), IMDB average increment: 0.91W, MSE = $0.15 \cdot 10^{-3}$W (b).

reported in the caption. As expected, despite some fluctuations, the average power remains approximately constant, as shown by the small MSE obtained by the constant fits.

Given Figure 8, we estimates the source node energy for local inference as:

$$E_{tot,i}(n) = P_{l,i} \cdot T_{tot,i}(n) \tag{8}$$

where $P_{l,i}$ is a constant power increment measured as shown in the figure.

Communication energy clearly depends on the wireless protocol used by the source node. In general, however, the total energy consumed when the source node offloads a task to a remote device can be modeled as [58]:

$$E_{tot,ij}(n) = E_{tx0,ij}(n) + P_{tx,ij} \cdot T_{tot,i}(n) \tag{9}$$

In this equation, the first component $E_{tx0,ij}(n)$ is the energy required to actually upload (download) input (output) data. This component can have different relations with the input length n for different protocols. The second addend of (9), instead, is the additional energy spent by the radio chip while waiting for the response (e.g. for keeping the connection open). This second components shows that, regardless of the chosen protocol, once the source node decides to offload a task, spending $E_{tx0,ij}$ for uploading/downloading data, the faster the response is obtained (i.e. the smaller $T_{tot,i}$), the better for total energy.

In other words, changing from time minimization to energy minimization only affects the decision performed by the source node. The latter must simply use (8) and (9) as the two elements of the comparison in (6). Once the source decides to offload an inference, however, all other nodes should perform their mapping decision with the goal of minimizing time, exactly as described in Section 4.1. In general, the decision performed by the source can also be based on a cost function that considers a *combination* of time and energy reduction, as described in [28].

5 Experimental Results

5.1 Experimental setup

In order to assess the effectiveness of CRIME, we perform several experiments considering three types of computing resources, representative of the different devices available within a collaborative network:

- ARM Cortex A-53@1.2GHz, 1GB RAM, as representative of a (powerful) end-node, such as a smartphone, smart speaker, etc.
- NVIDIA Jetson TX2, including a Pascal GPU with 256 CUDA cores, as an example of a high performance DL inference chip that could be equipped on an edge server.
- Dual Intel Xeon E5-2630@2.40GHz, 128GB RAM plus NVIDIA Titan XP GPU, to represent a typical cloud server.

All three devices run a Linux OS and exploit TensorFlow for RNN inference. Since our focus is only on inference, we use pre-trained RNNs for our experiments. In particular, we evaluate our approach on the CoVe network [55], composed of 2 stacked LSTM layers and used to pre-process sequences for a variety of NLP tasks, including sentiment analysis, question classification/answering, etc. In this work, we test CoVe against the Stanford Natural Language Inference (SNLI) Corpus and the Stanford Question Answering Dataset (SQuAD). Moreover, to show the independence of our method from RNN architecture details, we also consider a lighter network [56], composed of a single LSTM layer and trained on the IMDB dataset for sentiment classification.

The code of CRIME is written in Python. The time overhead of our engine ($<$ 1ms per inference on the Cortex A-53) is negligible, even compared to an inference of length 1. All inferences are performed on the real devices and execution times and energy consumption are measured using the "time" package and the HP 34401A multimeter respectively.

The CRIME code can support the autonomous execution of the entire distributed system, including data transmission, which is based on a REST API [57]. However, for the experiments of this paper, network transmissions are simulated, in order to have reproducible experiments and to assess the impact of different *predictable network conditions* on the effectiveness of our methodology. Indeed, it would be impossible to reproduce results obtained with a real network, especially for what concerns the connection to cloud servers, which is typically a multi-hop link over the Internet, whose latency and bandwidth are totally uncontrollable. The network simulator simply processes some connection profile files, formatted as time series of latency/bandwidth pairs. Clearly, these profiles can also be filled with real network data, as we do in Sections 5.2.2 and 5.3 to mimic a realistic (unpredictable) network. Importantly, the processing of each input still takes place on the device determined by CRIME, based on the simulated network's status. The only difference with respect to a stand-alone execution is therefore that, when computing the aggregated inference time and energy, the values of the real connection are replaced with the simulated ones at that instant.

Since CRIME is the first collaborative inference approach for RNNs, the baselines for comparison correspond to the execution of all inferences on a single device. Moreover, we also compare it to an *oracle policy*, i.e. one that always selects the best inference platform for each input. This oracle is not affected by all possible causes of wrong mapping in CRIME, such as errors in the local execution time estimate

(due to regression errors or to the unpredictable variability of compute times for the same length), outdated connection speed estimates and remote models, etc.

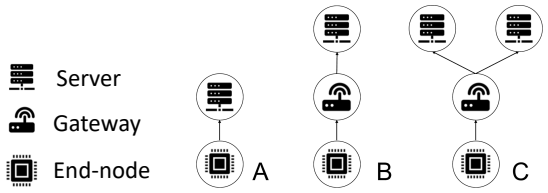


Fig. 9. DAG configurations considered in the experiments of Section 5: (a) two-level scenario (b) three-level scenario (c) three-level scenario with two cloud servers.

5.2 Two-level execution time minimization

In these experiments, we demonstrate the effectiveness of input-dependent collaborative inference for RNNs. For this, we consider the simplest collaborative system, composed of a single end-node (Cortex A53) and a single cloud server (Xeon + Titan XP), shown as "A" in Figure 9.

5.2.1 Fixed connection speed

First, we evaluate the execution time reduction obtained by CRIME compared to edge-only and cloud-only solutions, as a function of the connection conditions. Specifically, we fix the $B_{ij} = 2$Mbps and vary the round-trip latency $T_{rtt,ij}$ which as explained in Section 3.1 is the most influential parameter. For each latency value, we perform RNN inference on a large number of randomly sorted inputs and measure the total execution time.

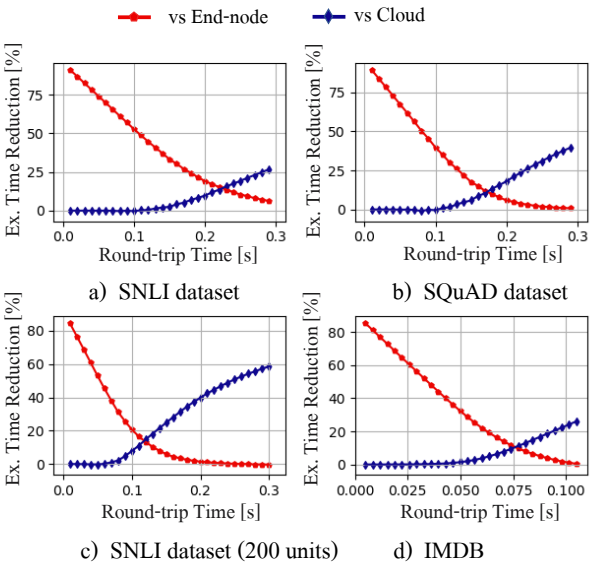


Fig. 10. Execution time reduction with respect to "edge-only" and "cloud-only" solutions. Results for different (fixed) network latencies and bandwidth fixed at 2Mbps. Comparison of the results on different datasets (a, b) and on different RNN architectures (c,d)

Figures 10a and 10b show the results obtained for CoVe, sampling 100k inputs from SNLI and SQuAD. For each value of $T_{rtt,i,j}$ on the x axis, the two curves show the reduction of the total execution time achieved by CRIME compared to a solution that performs all inferences locally (red curve) and to one that offloads everything to the cloud (blue). Clearly, the smaller $T_{rtt,i,j}$, the more cloud offloading is convenient, even for short inputs. In this condition, our policy reduces to the cloud-only approach, substantially speeding up the execution with respect to a fully on-edge approach. On the other hand, for very large latency values, local execution is always preferable, and our strategy tends to coincide with the edge-only scenario. The most interesting results are achieved for intermediate latency values, where our policy outperforms both baselines. For example, on SNLI and for $T_{rtt,ij} = 200$ms, total execution time is reduced by 19% and 10% with respect to edge-only and cloud-only approaches respectively.

Comparing Figure 10a and 10b shows the impact of the dataset on the performance of our method (for a fixed RNN). In particular, the benefits with respect to the cloud-only solution are greater for SQuAD, with an execution time reduction $> 40\%$ for large $T_{rtt,ij}$. This is because SQuAD contains generally shorter inputs (median length $= 9$, versus the SNLI median length $= 12$), allowing to exploit edge processing more frequently.

Similarly, Figure 10c shows the impact of the RNN size. The graph has been generated using the same data as Figure 10a, but using a modified version of CoVe, in which the hidden state vector size has been reduced to 200 units instead of the original 300. This RNN simplification makes local inference faster and therefore more convenient. As a consequence, the execution time reduction versus the cloud-only solution increases significantly for a given $T_{rtt,ij}$. Finally, Figure 10d refers to the IMDB dataset. The RNN used for this inference is significantly smaller than CoVe, even in its the reduced version. The purpose of this experiment is to show that the range of $T_{rtt,ij}$ for which our system obtains gains with respect to *both* baselines (x axis) varies significantly depending on the size of the RNN.

5.2.2 Variable connection speed

In this experiment, we consider a time-varying connection, to evaluate how CRIME adapts to changes in the transmission time. First, we consider an artificial time-varying network profile: we let $T_{rtt,ij}$ increase from 100ms to 200ms at 1/3 of the inference process and from 200ms to 300ms at 2/3, with B_{ij} always fixed at 2Mbps. Second, following the method in [59] we mimic a real latency profile using data from RIPE Atlas, an open source database of Internet measurements. We extrapolate a 3-hour-long record of connection speed between two random nodes in the database and use it as time-varying $T_{rt,ij}$ profile ($T_{rt,ij}$ ranges from 100ms to 245ms). Results are reported in Table 1.

The significant speed-ups obtained for the artificial profile are explained by the fact that the edge-only solution is sub-optimal in the initial phase, when latency is small, whereas at the end the cloud-only approach suffers from high transmission delays. On the contrary, CRIME selects the best inference target dynamically, based on the network conditions. An exception is the IMDB experiment, for which given the small size of the RNN, edge processing remains close to optimal throughout the experiment. With the real network profile from RIPE Atlas the savings are slightly smaller due to the less dramatic variation of $T_{rt,ij}$. However,

TABLE 1
Experimental results for variable network latency. RIPE ATLAS Meas. ID: 1437285, Probe ID: 6222, Date and time: May 3rd 2018, 3-6 p.m

Profile	Test	Ex. time reduction [%] vs end-node	Ex. time reduction [%] vs cloud	Ex. time increment [%] vs oracle
Artificial	SNLI	25.66	17.29	0.28
	SQuAD	15.67	25.99	0.40
	$SNLI_{200}$	20.93	7.84	0.18
	IMDB	1.51	23.47	0.01
RIPE Atlas	SNLI	21.35	8.64	0.24
	SQuAD	8.06	15.72	0.28
	$SNLI_{200}$	3.48	36.51	0.032
	IMDB	0.12	56.23	0.002

they still show the effectiveness of CRIME in adapting to varying connection statuses, RNN sizes and datasets. In both cases, the effectiveness of transmission time model updating is demonstrated by the small execution time increment ($< 1\%$) with respect to the oracle policy.

5.3 Execution time minimization on larger systems

5.3.1 Multiple levels

This experiment considers the case of a three-level system, including also an intermediate edge server (Jetson TX2), shown as configuration "B" in Figure 9. We generally refer to this server as "gateway" in the following. This experiment assesses the effectiveness of model propagation through multiple levels as described in Section 4.3. In this case, we use a real Bluetooth Low Energy (BLE) connection between the end-node and the gateway (GW), while we resort again to RIPE Atlas for the link between gateway and server.

Results for the same RNNs and datasets of Section 5.2 are reported in the leftmost 5 columns of Table 2. As shown, the speedups with respect to single-device mappings vary significantly depending on the RNN and dataset, but there is at least one condition for which CRIME reduces the execution time by $> 35\%$ compared to each device. Speedups are on average larger compared to Table 1, showing that CRIME is even more effective in a 3-level scenario. The IMDB experiment reports a negative time reduction with respect to a solution that executes all inferences on the end-node device. This is because for that RNN and dataset, and given the device compute speeds and connection profiles used in this experiment, the end-node is indeed the optimal device for all inputs. Intuitively, the GW device is not "fast enough" to justify offloading, despite the relatively short latency of the BLE connection, and sending inputs to the cloud is equally not convenient because it requires two "hops". The main reason why CRIME performs slightly worse than the Oracle in this experiment is the additional time overhead of periodic ping transmissions needed to obtain updated network status estimates, and to the reception of propagated regression models from higher-level devices. Wrong offloading choices due to network fluctuations or to regression errors, instead, have a negligible impact, meaning that CRIME does make the correct mapping decision for the great majority of inputs. Overall, despite the negative result, this experiment is still useful in showing the small overheads introduced by CRIME. In fact, the small time increment due to model propagations and periodic "pings" remains $< 1\%$ even in this corner case.

The main advantage of CRIME lies in its input-dependent selection of the inference target. Therefore, in Figure 11 we show an example of how inputs are distributed between these three levels based on their length, for the SQuAD dataset. The graph shows one bar for each input length present in the tested dataset. Bars are normalized to 100%, and colors show the percentage of inputs of that length that are processed on each device. As expected, short inputs are executed locally, intermediate inputs are processed by the gateway, and longer inputs are offloaded to the cloud. For a large portion of these lengths the choice is not univocal, because CRIME dynamically adapts to connection speed variations. For example, length-15 inputs are offloaded to the cloud when the GW-to-cloud link $T_{rt,ij}$ is small, and retained by the gateway otherwise.

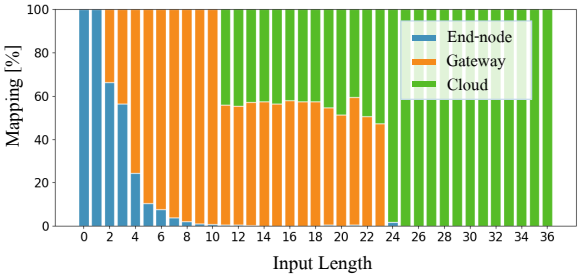


Fig. 11. Distribution of 100k inputs over end-node, gateway and cloud determined by CRIME for the SQuAD test.

5.3.2 Competing offloading devices and variable load

In this experiment we consider a complete collaborative inference system, shown as "C" in Figure 9. Using this configuration, we demonstrate two other features of CRIME; the selection among two competing offloading devices at the same level (the two servers shown in the DAG) and the response to load variations. The latter in particular are induced in the gateway: after processing half of the inputs, the gateway is artificially slowed down by starting 7 additional inference tasks in parallel, simulating requests from other end-nodes.

The results are shown in the rightmost part of Table 2. Comparing these numbers with the ones obtained in the previous experiment, the most relevant difference is an increase in the execution time reduction compared to a gateway-only approach. The reason is that executing entirely on the gateway becomes much less effective due to the load increment. In contrast, CRIME promptly adapts, starting to process more inputs locally and forwarding those that reach the GW to one of the two cloud servers.

Despite the fact that execution time reductions with respect to the two cloud servers are similar, CRIME does not select them randomly. Since this is not apparent from Table 2, Figure 12 summarizes visually the entire test for the SNLI dataset. The topmost chart shows the round-trip latency profiles of all three connection links in the system, while the central and lowermost charts visualize the mapping choices performed by CRIME and by the oracle policy. The time axis is divided in slots of 15s, and the colored bars

TABLE 2
Experimental results on a three-level collaborative system, and on a system with two alternative cloud servers. RIPE ATLAS Meas.ID: 1437285, Probe ID: 6222, Date and Time: May 3rd 2018, 12-16 p.m [GW to cloud1] and May 6th 2018, 7:30-11:30 a.m [GW to cloud2]

Test	Three-Levels				Three-Levels and Two-Servers				
	Ex. time reduction [%]			Ex. time incr. [%]	Ex. time reduction [%]				Ex. time incr. [%]
	vs end-node	vs gateway	vs cloud	vs oracle	vs end-node	vs gateway	vs cloud1	vs cloud2	vs oracle
SNLI	35.57	5.93	25.33	0.32	35.56	45.38	26.11	25.13	0.72
SQuAD	26.40	1.49	31.92	0.99	23.22	35.12	32.98	29.51	1.18
$SNLI_{200}$	4.52	20.22	37.48	0.85	4.75	50.87	44.73	37.69	1.15
IMDB	-0.46	68.49	59.22	0.46	-0.70	80.45	64.35	59.12	0.71

represent the percentage of inputs processed in that slot that are mapped to each device. The orange background area represents the portion of the test in which the gateway load has been increased.

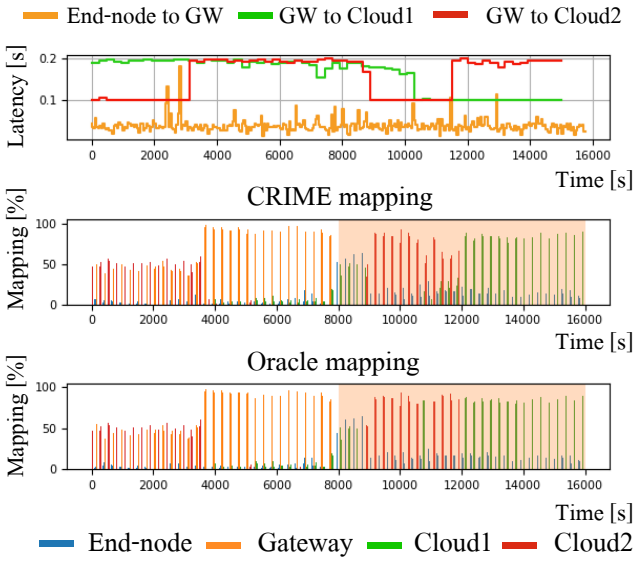


Fig. 12. CRIME mapping and Oracle mapping for the SNLI test with a three-level system and two competing cloud servers. The shaded orange area indicates the portion of the test in which the gateway is subjected to an higher load.

Overall, the figure clearly shows how CRIME adapts to changing conditions. For instance, in the initial portion of the test, cloud server 2 is selected more often, since the round-trip latency to reach it is low (and in particular lower than that of server 1). When the transmission time to server 2 increases (around 3000s), the gateway starts to process most inputs locally. In turn, when the GW load increases at about 8000s, performing inference there stops being convenient, and CRIME maps an increasing percentage of inputs to the end-node and to both cloud servers. Towards the end of the test, cloud server 1 is the most selected device due to the lower transmission latency.

Comparing the two mapping charts of Figure 12 evidently shows the similarity between the selections performed by CRIME and by the Oracle policy. This is quantitatively confirmed by the small overheads reported in Table 2.

Figure 13 shows the impact of T_{ping}, the maximum time that each CRIME node waits without offloading inputs to a given successor before sending an empty ping packet to update its transmission latency estimate. We discuss this parameter in this section since the larger DAG makes T_{ping}

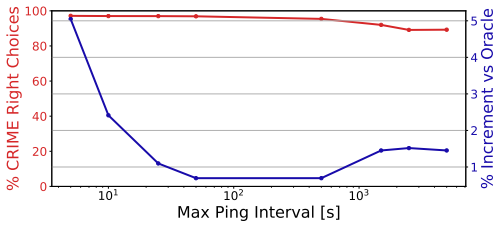


Fig. 13. Impact of T_{ping} on the percentage of correct CRIME mappings and on the total execution time overhead, for the SNLI experiment.

expiration more probable. Specifically, the graph reports the percentage of correct mappings identified by CRIME and the total execution time overhead with respect to the oracle as functions of T_{ping}. The time curve clearly identifies a trade-off: for too small T_{ping}, the overhead associated with ping packets causes a not negligible increment in the total inference time of CRIME (about 5%). At the other extreme, a too large T_{ping} causes again an overhead, this time because of the wrong mapping decisions performed by CRIME due to outdated transmission time estimates. The optimum is obtained for intermediate values, approximately in the range 1-10 minutes.

5.4 Energy minimization

As a final experiment, we assess the effectiveness of CRIME for energy minimization. We refer again to the experimental setup described in Section 5.2 but this time using (8) and (9) in the optimization performed by the source node. To estimate transmission energy, we use the model of [58] for 4G LTE technology. We compare the achieved energy reduction with the results provided by edge-only and cloud-only solutions for different round-trip latency conditions. Results are presented only for the SNLI and SQuAD tests for sake of space, and are shown in Figure 14.

Energy reduction trends mirror those obtained for execution time minimization (see Figure 10) because of the close relationship between energy usage and inference time described in Section 4.4. However, due to the large transmission consumption of the selected WWAN technology, offloading becomes a more costly operation when energy is the target metric. Therefore, CRIME tends to favor local processing even if this is does not yield the smallest response time. As a consequence, with respect to Figures 10a and 10b, the saving curves are shifted to the left.

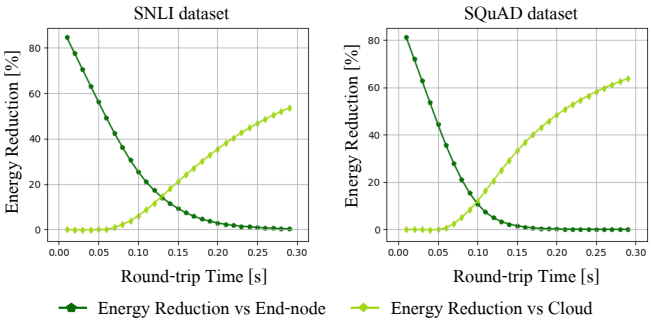


Fig. 14. Energy reduction with respect to "edge-only" and "cloud-only" solutions. Results for different (fixed) network latency values and bandwidth fixed at 2Mbps.

6 Conclusion

We have presented CRIME, a light-weight and distributed mapping engine for collaborative RNN inference. In CRIME each collaborating device performs a local optimization, greadily selecting the fastest (or least consuming) mapping between local processing and remote offloading. This selection is based on estimates of the total processing time both locally and in the directly connected devices (including transmission time), which are constantly updated and propagated through the network. Importantly, these estimates account for the length of the processed input, which has a significant influence on RNN processing time.

With experiments on three different datasets and three different RNNs, we have shown that CRIME yields significant execution time and energy reductions with respect to any static mapping solution. These results are very close to those achieved by an Oracle policy that always selects the most appropriate target for inference. In our future work, we plan on extending the CRIME framework to support other types of deep learning models for sequence processing, such as attention-based transformers, which show different dependencies between input length and inference complexity.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, The MIT Press, 2016.

[2] V. Sze et al, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proc. of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[3] Z. Zhou et al, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing," in *Proc. of the IEEE*, pp. 1738–1762, 2019.

[4] A. Kusupati et al, "FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network," *Proc. of NIPS*, 2018, pp. 9017–9028.

[5] J. Chauhan et al, "Breathing-Based Authentication on Resource-Constrained IoT Devices using Recurrent Neural Networks," *Computer*, vol. 51, no. 5, pp. 60–67, 2018.

[6] Zia Uddin, "A wearable sensor-based activity prediction system to facilitate edge computing in smart healthcare system", *Journal of Parallel and Distributed Computing*, vol. 123, pp. 46–53, 2019.

[7] N. Wadhwani et al, "IOT Based Biomedical Wireless Sensor Networks and Machine Learning Algorithms for Detection of Diseased Conditions," *Proc. of i-PACT*, 2019, pp. 1-7.

[8] W. Zhang et al., "LSTM-Based Analysis of Industrial IoT Equipment," *IEEE Access*, vol. 6, pp. 23551-23560, 2018

[9] H. Shi et al, "Deep Learning for Household Load Forecasting—A Novel Pooling Deep RNN," *IEEE TSG*, vol. 9, no. 5, pp. 5271-5280, 2018.

[10] J. Chen and X. Ran, "Deep Learning With Edge Computing: A Review," *Proc. of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[11] W. Shi et al, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[12] Y. H. Chen et al, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE JSSC*, vol. 52, no. 1, pp. 127–138, 2017.

[13] T. Mealey et al, "Accelerating inference in long short-term memory neural networks," *Proc. of NAECON*, 2018, pp. 382–390.

[14] S. Cao et al, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *Proc. of FPGA*, 2019, pp. 63–72.

[15] J. Kung et al, "Peregrine: A Flexible Hardware Accelerator for LSTM with Limited Synaptic Connection Patterns," in *Proc. of DAC*, 2019, pp. 209:1–209:6.

[16] Y. Guan et al, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *ASP DAC* 2017, pp. 629–634.

[17] C. Gao et al, "EdgeDRNN: Enabling low-latency recurrent neural network edge inference," in *AICAS* 2020, p. 41-45.

[18] P. Gysel, "Hardware-Oriented Approximation of Convolutional Neural Networks," *CoRR*, http://arxiv.org/abs/1605.06402, 2016.

[19] D. Jahier Pagliari et al, "Dynamic Bit-width Reconfiguration for Energy-Efficient Deep Learning Hardware," in *Proc. of ISLPED*, 2018, pp. 47:1—-47:6.

[20] S. Han et al, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proc. of ICLR*, 2015, pp. 1–14.

[21] T. Yang et al, "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning," in *Proc. of CVPR*, 2017, pp. 6071–6079.

[22] Y. Kang et al, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," *Proc. of ASPLOS*, 2017, pp. 615–629.

[23] A. E. Eshratifar et al, "BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services," *CoRR*, http://arxiv.org/abs/1902.01000, 2019.

[24] A. E. Eshratifar et al, "Jointdnn: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE TMC*, vol. PP, 01 2018.

[25] Z. Zhao et al, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE TCAD*, vol. 37, no. 11, pp. 2348–2359, 2018.

[26] E. Li et al, "Edge AI : On-Demand Accelerating Deep Neural Network Inference via Edge Computing," *CoRR*, https://arxiv.org/abs/1910.05316, 2019.

[27] Y. Huang et al, "Deepar: A hybrid device-edge-cloud execution framework for mobile deep learning applications," in *Proc. of INFOCOM Workshops*, 2019, pp. 892–897.

[28] D. Jahier Pagliari et al, "Input-dependent edge-cloud mapping of recurrent neural networks inference," in *Proc. of DAC*, 2020, pp. 1–6.

[29] Y. Han et al, "Convergence of edge computing and deep learning: A comprehensive survey," *CoRR*, https://arxiv.org/abs/1907.08349, 2019.

[30] S. Han et al, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proc. of ACM* 2017, pp. 75–84.

[31] Q. Cao et al, "Mobirnn: Efficient recurrent neural network execution on mobile gpu," in *Proc. of EMDL@MobySys*, 2017, pp. 1–6.

[32] C. Holmes et al, "Grnn: Low-latency and scalable rnn inference on gpus," in *Proc. of Eurosys* 2019. pp. 1-16.

[33] P. Gao et al. "Low latency rnn inference with cellular batching," in *Proc of EuroSys* 2018, pp.1-15.

[34] L. Lai et al, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *CoRR*, http://arxiv.org/abs/1801.06601, 2018.

[35] J. Ott et al, "Recurrent Neural Networks With Limited Numerical Precision," *CoRR*, http://arxiv.org/abs/1611.07065, 2016.

[36] B. Reagen et al, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ACM SIGARCH*, vol. 44, no. 3, pp. 267–278, 2016.

[37] S. Liu et al, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proc. of ACM*, 2018, pp. 389–400.

[38] D. Jahier Pagliari et al, "Energy-Efficient Digital Processing via Approximate Computing," in *Smart Systems Integration and Simulation. Springer, Cham*, Chapter 4, pp. 55–89, 2016.

[39] J. Yu et al, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proc. of ISCA*, 2017 pp. 548-560.

[40] E. Park et al, "Big/little deep neural network for ultra low power inference," in *Proc. of CODES+ISSS*, 2015, pp. 124–132.
[41] H. Tann et al, "Runtime configurable deep neural networks for energy-accuracy trade-off," in *Proc. of CODES*, 2016, pp. 1–10.
[42] B. Taylor et al, "Adaptive Deep Learning Model Selection on Embedded Systems," in *Proc. of LCTES*, 2018, pp. 31–43.
[43] D. Jahier Pagliari et al., "Dynamic Beam Width Tuning for Energy-Efficient Recurrent Neural Networks," in *Proc. of GLSVLSI*, 2019, pp. 69–74.
[44] D. Jahier Pagliari et al., "Sequence-To-Sequence Neural Networks Inference on Embedded Processors Using Dynamic Beam Search," *Electronics*, vol. 9, no. 2, p. 337, feb 2020.
[45] J. Jo et al, "Similarity-Based LSTM Architecture for Energy-Efficient Edge-Level Speech Recognition," *Proc. of ISLPED*, 2019, pp. 1–6.
[46] S. Teerapittayanon et al, "Branchynet: Fast inference via early exiting from deep neural networks," *ICPR*, 2016, pp. 2464–2469.
[47] H. Chen et al, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proc. of ACM Sensys*, 2015.
[48] U. Drolia et al, "Cachier: Edge-caching for recognition applications," in *Proc. of IEEE ICDCS*, 2017.
[49] P. Guo et al, " Foggycache: Cross-device approximate computation reuse," in Proc. of ACM Mobicom," in *Proc. of ACM Mobicom*, 2018.
[50] E. Cuervo et al, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proc. of MobiSys*, 2010, pp. 49–62.
[51] M.-R. Ra et al, "Odessa: Enabling Interactive Perception Applications on Mobile Devices," in *Proc. of MobiSys*, 2011, pp. 43–56.
[52] H.-J. Jeong et al, "IONN: Incremental Offloading of Neural Network Computations from Mobile Devices to Edge Servers," in *Proc. of SoCC*, 2018, pp. 401–411.
[53] K. Shin et al, "Enhanced partitioning of dnn layers for uploading from mobile devices to edge servers," in *Proc. of EMDL@MobiSys*, 2019, pp. 35–40.
[54] A. Thomas et al, "Hierarchical and Distributed Machine Learning Inference Beyond the Edge," in *Proc. of ICNSC 2019*, 2019, pp. 1004–1009.
[55] B. McCann et al, "Learned in Translation: Contextualized Word Vectors," *CoRR*, http://arxiv.org/abs/1708.00107, 2017.
[56] Online: https://github.com/keras-team/keras, Accessed March 2020.
[57] Online: https://docs.cherrypy.org, Accessed July 2020.
[58] J. Huang et al, "A close examination of performance and power characteristics of 4g lte networks," in *Proc. of MobiSys*, 2012, pp. 225–238.
[59] M. Mouchet et al, "Statistical Characterization of Round-Trip Times with Nonparametric Hidden Markov Models," in *Proc. of IM*, 2019, pp. 43–48.

Daniele Jahier Pagliari (M'15) received the M.Sc. and Ph.D. degrees in computer engineering from Politecnico di Torino, Torino, Italy, in 2014 and 2018, respectively. He is currently an Assistant Professor in the same institution. His research interests include computer-aided design of digital systems, approximate computing and low-power optimizations for embedded systems, with particular focus on embedded machine learning.

Roberta Chiaro (M'20) is a Research Assistant at Politecnico di Torino. She received a degree in Physics of Complex Systems from Politecnico di Torino, in 2013. Her research interests concern machine learning for embedded systems and industry 4.0.

Enrico Macii (SM'02-F'05) is a Full Professor of Computer Engineering with the Politecnico di Torino, Torino, Italy. He holds a Laurea degree in electrical engineering from the Politecnico di Torino, a Laurea degree in computer science from the Università di Torino, Turin, and a PhD degree in computer engineering from the Politecnico di Torino. His research interests are in the design of electronic digital circuits and systems, with a particular emphasis on low-power consumption aspects. He is a Fellow of the IEEE.

Massimo Poncino (SM'12-F'18) is a Full Professor of Computer Engineering with the Politecnico di Torino, Torino, Italy. His current research interests include several aspects of design automation of digital systems, with emphasis on the modeling and optimization of energy-efficient systems. He received a PhD in computer engineering and a Dr.Eng. in electrical engineering from Politecnico di Torino. He is a Fellow of the IEEE.