Input-dependent edge-cloud mapping of recurrent neural networks inference

Input-Dependent Edge-Cloud Mapping of Recurrent Neural Networks Inference

Daniele Jahier Pagliari*, Roberta Chiaro*, Yukai Chen*, Sara Vinco*, Enrico Macii† and Massimo Poncino*
*Department of Control and Computer Engineering, †Interuniversity Department of Regional and Urban Studies and Planning
Politecnico di Torino, Turin, Italy
Email: firstname.lastname@polito.it

***Abstract*—Given the computational complexity of Recurrent Neural Networks (RNNs) inference, IoT and mobile devices typically offload this task to the cloud. However, the execution time and energy consumption of RNN inference strongly depends on the length of the processed input. Therefore, considering also communication costs, it may be more convenient to process short input sequences locally and only offload long ones to the cloud. In this paper, we propose a low-overhead runtime tool that performs this choice automatically. Results based on real edge and cloud devices show that our method is able to simultaneously reduce the total execution time and energy consumption of the system compared to solutions that run RNN inference fully locally or fully in the cloud.**

I. Introduction

Recurrent Neural Networks (RNNs) are a type of Deep Neural Network (DNN) that has become state-of-the-art for advanced sequence analysis tasks, e.g. speech recognition, neural machine translation, sentiment analysis, etc [1]. The major leap in accuracy provided by these networks comes at the cost of high computational complexity for training and inference [2]. Currently, the standard approach is to run both tasks in cloud servers equipped with GPUs and multi-core CPUs. However, while training is often a one-time task and can therefore be left in the cloud, several benefits in terms of responsiveness, energy efficiency and security could derive from executing inference (fully or partially) in edge nodes, such as mobile or IoT devices [2]–[14].

Many researchers have proposed low-cost, fast and energy-efficient hardware accelerators that enable the complete execution of DNN inference at the edge [2]–[7]. While initially focusing mostly on Convolutional Neural Networks (CNNs), research on DNN acceleration has recently started to consider also RNNs [6], [7]. However, most mobile and IoT systems cannot afford dedicated inference hardware and must rely on general purpose embedded CPUs. With these devices, running the entire inference at the edge might be sub-optimal or even unfeasible due to performance and energy limitations [14]. Indeed, researchers have shown that, in these cases, the energy- and latency-optimal solution is often obtained *splitting the computation* between the edge node and the cloud [12]–[15]. By performing part of the inference locally and the rest remotely, the best compromise can be found between the time and energy consumed by the edge node's computation subsystem, and those consumed in transmitting intermediate data to the cloud.

Existing methods for edge-cloud inference mapping perform *input independent* design choices, i.e. the optimal selection among edge and cloud processing is performed independently of the considered input datum. However, recent research on DNN inference has shown that *input dependent* optimizations, both at the hardware and algorithm levels, can yield superior energy, latency, and accuracy results [8]–[11]. The rationale behind these approaches is that not all inputs are equally difficult to process from a computational standpoint. This is particularly true for RNNs, whose computational burden strongly depends on the length of the input sequence [10].

In this work, we propose a framework that applies input dependent optimizations to the problem of edge-cloud mapping of RNN inference. To the best of our knowledge, ours is the first work addressing this problem for RNNs. Our method decides at runtime whether to perform inference locally or in the cloud, *depending on the length of the processed input sequence, and on the current status of the system (e.g. network latency)*. Results based on real edge and cloud devices show that our method can simultaneously reduce the total inference time and energy consumption by up to 20-40% compared to a fully local and a fully remote solution.

II. Background and Related Work

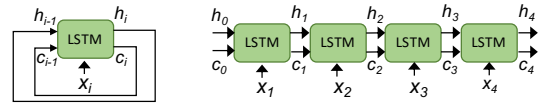
A. Recurrent Neural Network Inference

RNNs differ from standard feed-forward deep neural networks (such as CNNs) because of the presence of *feedback*, which allows them to process sequences of data and learn temporal relationships among inputs.

Figure 1a shows a conceptual block diagram of one of the most popular RNN variants, called *Long Short Term Memory* (LSTM). When processing the i-th input of a sequence, the green block (called *cell*) performs a fixed set of matrix-vector multiplications and non-linear operations (hyperbolic tangent and sigmoid function evaluations). This produces two outputs, the *cell state* (c_i) and the *hidden state* (h_i), that are then fed-back to the network at step $i+1$. Readers can refer to [1] for more details on RNN/LSTMs, omitted here for sake of space.

In practice, when performing inference on an input sequence of length N, the cell is *unrolled* (i.e. replicated) N times, as shown in Figure 1b for $N = 4$. Each copy of the cell performs

the same operations and shares the same weight matrices, learned during training. The final outputs (h_4 and c_4 in the figure) encode a representation of the input sequence and are typically fed to a classifier, such as a fully-connected NN.



a) Conceptual LSTM b) Example of unroll for N = 4

Fig. 1. Example of LSTM unrolling.

Figure 1b suggests that the computational complexity for inference in a RNN/LSTM grows linearly with N. Moreover, although each LSTM block performs a highly-parallel matrix-vector multiplication kernel, parallelism among different unrolled replicas is limited, as a given replica cannot start until the outputs from the previous step are ready [6], [7][1].

Given these considerations, inference execution time in RNNs can also be expected to grow linearly with respect to input length. Moreover, as each replica performs exactly the same operations, the power consumption of the hardware performing inference can be reasonably assumed constant throughout the process, especially for a single-task system (e.g. a smart sensor): consequently, *energy* consumption should also grow linearly with N. These dependencies will be confirmed by the measurements performed in Section III-A and III-B, and form the basis for our input-dependent optimization.

B. Methods for Collaborative Edge-Cloud Inference

Several papers have proposed approaches to split DNN inference between edge and cloud for latency or energy minimization (so-called *collaborative inference*). The work in [12] proposes a 3-level hierarchical framework focusing on applications that process data from multiple sources. In this framework, smart sensors, intermediate edge gateways and cloud servers each perform a *partial inference* step on their locally available data, leveraging results from the previous levels and forwarding theirs to the next levels. Thanks to this hierarchy, the amount of data transmitted on the network is reduced, positively impacting latency and energy consumption, at the cost of a possible decrease in accuracy.

In [13] the authors present a similar approach, but rather than splitting a task into multiple partial classifications, the architecture of a single NN is modified so that the first layers only process data from a single sensor, and can be executed at the edge. Subsequent layers aggregate layer outputs from multiple sensors and are executed in the cloud. This solution simplifies the design of the NN with respect to [12], enabling end-to-end training with standard back-propagation.

The authors of [14] focus on applications with a single data source and propose to perform CNN-based inference at the edge only up to a certain layer, and to complete it in the cloud. Thanks to the feature size compression of intermediate layers, they show that this partial local processing may reduce the total time/energy cost (including transmission), compared to a fully local/remote inference. A runtime environment is proposed to determine the optimal split-layer depending on the network conditions and on the load of the cloud server.

In [15], the authors apply the same type of layer-wise splitting for CNNs, but modify the network architecture to make it more partitioning-friendly, adding a so-called *bottleneck layer*. The first part of this layer, called *reduction* unit, compresses its inputs (e.g. through JPEG), while the following *restoration* unit decompresses them. At training time, this compression/decompression is approximated by an identity function, thus allowing standard back-propagation of gradients. At inference time, reduction is performed at the edge while restoration takes place in the cloud, allowing the transmission of compressed data to reduce energy and latency.

All these works perform *input independent* design choices. For example, the optimal split-layer in [14] is decided at runtime, based on the current status of the system (e.g. network speed, etc.) but independently from the considered input datum. This is motivated by the substantial independence on input values of the processing complexity in feed-forward NNs. However, the analysis of Section II-A demonstrates that RNNs require a completely different decision policy.

III. Proposed Method

A block-diagram of our framework is shown in Figure 2. We propose to add a simple runtime (called *mapping engine*) on the edge node, in charge of deciding whether to perform a RNN inference on the node itself or in a cloud server. We assume that both the edge node and the server maintain a local copy of the same RNN model and of its trained weights, so that they can both execute a *complete* inference when needed. In general, the two devices can use different inference engines, such as the light-weight ARM-NN for the edge node, and the more flexible TensorFlow for the cloud.

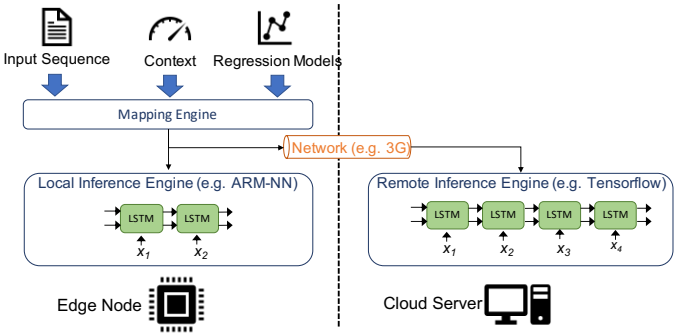


Fig. 2. Conceptual scheme of the proposed framework.

The mapping engine takes three inputs, i.e.:

- The input sequence to be processed by the RNN.
- A set of regression models to forecast the inference execution time on the edge node and on the cloud server, as well as the energy consumption of the edge node.

[1]The same analysis also applies to most variations of this basic architecture proposed in literature (e.g. bi-directional RNNs/LSTMs), and to different kinds of cells (e.g. Gated Recurrent Units). Therefore, although the rest of the paper focuses on LSTM networks, our solution also applies to other types of RNNs, and we will use the terms RNN and LSTM interchangeably.

- Context information, such as the status of the network link (e.g. 3/4G or WiFi) connecting edge and cloud.

Using this information, the mapping engine selects whether to perform inference locally or on the cloud for that input, with the goal of minimizing an objective function. Importantly, for a given input, inference is always *entirely* executed either in the edge node or in the cloud, and never *partitioned* between the two devices. This is because, as explained below, input data sizes for RNNs are typically much smaller than for CNNs, hence the "data compression" benefit deriving from partitioned inference (see Section II-B) would be minimal.

As in previous literature [14], [15], we consider the edge node as the only target for energy minimization, and neglect cloud consumption. However, contrary to most previous works, we do not optimize either the total execution time of the system (T_{tot}) or the energy consumption of the edge device (E_{edge}) individually. Rather, we consider the more general case of a *combination* of the two metrics, i.e. we search for:

$$\min_{D}(T_{tot}(D,N,C) + w \cdot E_{edge}(D,N,C)) \tag{1}$$

where D is the selected device (edge or cloud), N is the input length, C is the context information and w is a user-defined parameter. Evidently, $w = 0$ corresponds to pure execution time minimization whereas $w \to \infty$ is pure energy minimization. If both T_{tot} and E_{edge} are normalized (e.g. with respect to the execution of a sentence of length $N = 1$ on the edge device), w can be interpreted as the "importance" given to energy with respect to time.

The main features of our framework are described in the following sections.

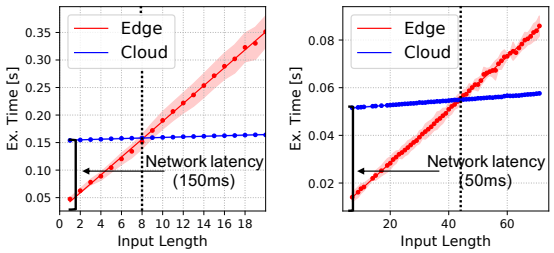
A. Edge and Cloud Execution Time Modeling

The proposed runtime uses an estimate of edge and cloud execution times in order to determine where to run the inference. Such estimate can only be built by characterization, as it is device and NN-dependent. In particular, based on the analysis of Section II-A, we use two *linear regression* models to approximate the dependency between the input length N and the edge/cloud execution time.

Figure 3 shows the results of this characterization for two different RNNs (detailed in Section IV) and for two example devices. As an edge device (red dots and curves) we consider an ARM Cortex A-53@1.2GHz, 1GB RAM, Linux OS, whereas as a cloud device we use a NVIDIA Titan XP GPU on a server-class platform, i.e. 32-thread Intel Xeon E5-2630@2.40GHz, 128GB RAM, Linux OS. Each dot in the figure represents the mean execution time over 1000 inferences for a given input length. Shaded areas around the dots (hardly visible for the cloud device) define the standard deviation of the execution time, showing the low variability of the results. Solid lines are the linear fits of the data, whose scores are reported in the caption.

When inference is performed on the edge, the total execution time of the system is simply estimated as:

$$T_{tot}(D = edge, N, C) = T_{edge}(N) \tag{2}$$



(a) CoVe [16] (b) IMDB [17]

Fig. 3. Execution time versus input length for two RNNs. Points and colored areas represent means and standard deviation intervals over 1000 inferences. Lines are linear regression fits. CoVe regression scores: edge $MSE = 9.32 \cdot 10^{-6}$, $R^2 = 0.999$, cloud $MSE = 7.69 \cdot 10^{-8}$, $R^2 = 0.991$. IMDB regression scores: edge $MSE = 2.37 \cdot 10^{-7}$, $R^2 = 0.999$, cloud $MSE = 4.25 \cdot 10^{-9}$, $R^2 = 0.998$.

where T_{edge} is the output of the (red) linear regression model. Estimating cloud execution time is less straightforward, as besides computation, communication also has an impact. Therefore, total execution time can be modeled as:

$$T_{tot}(D = cloud, N, C) = T_{tx}(N, C) + T_{cloud}(N) \tag{3}$$

where T_{tx} is the total communication time and $T_{cloud}(N)$ is the inference time on the cloud server. In turn:

$$T_{tx}(N, C) = T_{rt}(C) + \frac{S(N)}{B(C)} \tag{4}$$

where T_{rt} is the round-trip network latency, S is the size in bytes of the transmitted sequence and B is the network bandwidth. RNN inputs are typically encoded with a few Bytes per element, thus leading to S values in the order of 100s of Bytes [1]. Therefore, communication time tends to be dominated by T_{rt} [18]. As an example, assuming that 100 Bytes are transmitted on a 3G link with $B = 1$Mbps, the fractional term in (4) becomes 0.8ms. Assuming a typical value of T_{rt} of 150ms [18], this is ≈200x larger than the transmission time. Therefore, for most RNN applications, T_{tx} will be bound by network latency, which is virtually *independent of the input size*.

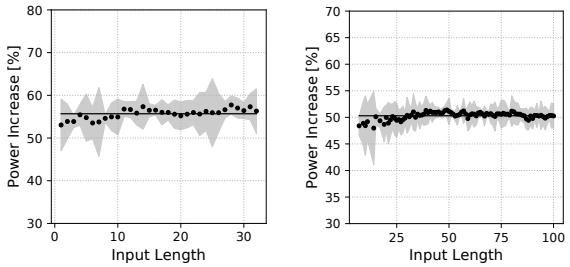
To visualize this effect, we offset the blue curves in Figure 3 by two different T_{rt} values for the two RNNs. The slope of the curve (measuring inference time as a function of input size) is much smaller than for the edge device due to the higher performance of the cloud platform, but the dependence is still almost perfectly linear, as shown by the scores in caption.

Overall, Figure 3 serves as a motivation for the proposed runtime, at least in the case of pure execution time optimization, i.e. $w = 0$ in (1): indeed, for short input sequences (left of the dashed black line in the two graphs) edge processing is faster, whereas cloud offloading becomes preferable for longer ones. The figure also shows that, depending on the target RNN and dataset, inference execution times (y axes) and input length ranges (x axes) may vary significantly, leading to very different trade-offs, even for the same edge/cloud devices.

B. Edge Energy Modeling

Similarly to execution time, we measure power consumption on the edge device to validate the assumptions of Section II-A.

We sample the power consumption of the same edge device used for Figure 3 with a digital multimeter (HP 34401A) and a period of 1s, running 1000 inferences per each value of N. Figure 4 reports the results of these measurements. 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 reported in the caption. Despite some fluctuations, the average power remains approximately constant, as shown by the small MSE obtained by the constant fits.



(a) CoVe [16] (b) IMDB [17]

Fig. 4. 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, IMDB average increment: 0.91W, MSE = $0.15 \cdot 10^{-3}$W.

Given Figure 4, our runtime uses the following model to predict energy consumption when inference is run at the edge:

$$E_{edge}(D = edge, N, C) = P_{edge} \cdot T_{edge}(N) \quad (5)$$

where P_{edge} is the constant power increment measured as described above. To model the power consumed in communication (P_{tx}) we use the results of [15], where power is estimated as a linear function of the transmission bandwidth, with different parameters depending on the connection type (3G, 4G or WiFi). Therefore, energy consumption (of the *edge* device) when inference is run in the cloud is estimated as:

$$E_{edge}(D = cloud, N, C) = P_{tx}(C) \cdot T_{tx}(N, C) \cdot \quad (6)$$

C. Online Adaptation

The exact *break-even* value of N for which cloud processing becomes more convenient than edge processing in terms of time, energy or a combination of the two, clearly depends on the state of the system, and in particular on the time-varying status of the network (especially T_{rt}). In order to adapt our runtime's decisions to variations in the network connection, we use two different mechanisms.

First, the T_{rt} estimate is updated every time the runtime decides to perform an inference in the cloud, adding timestamps to the transmitted/received input sequence. Second, whenever the latest cloud inference took place too far in time (e.g. more than 1 minute ago), the edge node pings the cloud server to update its T_{rt} estimate. To see why this second mechanism is needed, consider the case of pure execution time optimization ($w = 0$). With reference to Figure 3, a momentary large increase in T_{rt} would shift the blue curve so high that the break-even point would move towards input sizes that never occur in practice, and therefore our runtime would start to *always* select edge processing. Consequently, the timestamp mechanism would never take place, and the runtime would not notice any future decrease of T_{rt}.

In this work, although we account for random execution time and power fluctuations, we assume the mean inference time for a given N and the mean CPU power consumption in the edge node are constant. However, our framework can be extended to support load and power variations (e.g. due to other processes) as another type of context information. This extension will be object of our future work.

IV. Experimental Results

We test the proposed methodology on the edge and cloud platforms described in Section III, i.e. ARM Cortex A-53 and Intel Xeon + NVIDIA Titan XP respectively. We experiment on two RNNs and three datasets (described later) using TensorFlow as inference engine. Our method has no direct competitor, since all previous solutions for collaborative edge-cloud intelligence only target feed-forward NNs. Thus, we compare against the two trivial solutions, i.e. running inference fully in the edge or in the cloud.

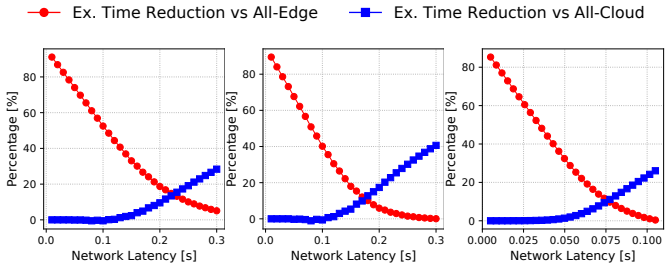
In all experiments, we measure computation times and power (T_{edge}, T_{cloud} and P_{edge}) on the real devices. To model the impact of communication, we use a Python-based simulator, as opposed to actually transmitting data on the network, in order to be able to assess the impact of different *predictable network conditions* on the effectiveness of our methodology. The simulator receives as input a network connection profile, formatted as a time series of latency/bandwidth pairs. Clearly, this profile can also be filled with real network data, as we do in Section IV-D, to mimic a realistic (unpredictable) network.

We perform a sequence of inferences, using our mapping engine to select the device (edge or cloud) that minimizes (1) for each input. The impact of this selection is then evaluated by measuring the *real* time and energy values for each sentence on the selected device. This allows to evaluate the impact of inference time/power variability and of regression errors on the effectiveness of the proposed runtime. We also account for the fact that network information can be outdated, as it is only updated when cloud inference is selected or every minute through a ping. We conservatively assume that pings are executed sequentially with respect to inferences on the edge and we account for their contribution on total time and energy. Notice that the time overhead of the runtime itself ($<$ 1ms per inference on the target edge device) is negligible, even compared to the execution of 1-input sequences.

A. Execution Time Optimization

In this section, we consider pure execution time optimization, i.e. we set $w = 0$ in (1). We then assess the effectiveness of our method for different values of the network latency T_{rt}, which is one of the most important parameters that determine the selection between edge and cloud. In a first experiment, we consider the "CoVe" network [16], a 2-layer LSTM used to process sequences for a variety of NLP tasks. Specifically,

we use two of the datasets considered in [16], i.e. SNLI (entailment) and SQuAD (question answering). The edge and cloud regression models used in our mapping engine for this network correspond to Figures 3a and 4a. Figures 5a and 5b show the results of using our framework to map the inference of 100k random sentences from each dataset. The graphs show the *reduction of the total inference time* with respect to edge-only and cloud-only solutions, for different values of T_{rt} (assumed *fixed* for the entire simulation).



(a) CoVe + SNLI (b) CoVe + SQuAD (c) IMDB

Fig. 5. Total execution time reduction versus "all edge" and "all cloud" solutions for the RNNs in [16] and [17]. Results for different (fixed) network latencies; bandwidth fixed at 1Mbps.

As expected, for small latency values our runtime offloads all inferences to the cloud and performs exactly as a cloud-only solution, being thus much faster than a edge-only solution. As latency increases, however, inferences corresponding to the shortest input sequences start to be executed locally, saving time with respect to a cloud-only approach. For instance, for a 200ms latency our framework outperforms *both* edge- and cloud-only solutions by $\approx$ 19% and 10%, respectively. Clearly, when latency increases even more, our strategy tends to coincide with the edge-only case.

Comparing Figures 5a and 5b also shows the impact of the dataset on our optimization. In fact, SQuAD inputs have a smaller median length compared to SNLI ($N_{median} = 9$ vs 12), hence the mapping engine can exploit edge processing more often, thus obtaining a larger saving vs the cloud-only approach for a given T_{rt}.

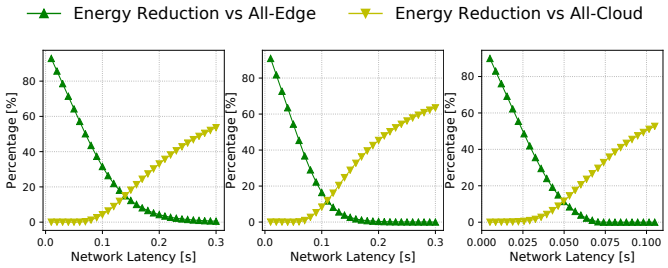
Figure 5c, instead, shows the impact of the network *size*. This graph refers to the smaller (1-layer) LSTM from [17], trained on the IMDB dataset for sentiment classification. Regression models for this NN are depicted in Figures 3b and 4b. The graph is obtained running inference on all sentences of the dataset with $N < 100$. While the dependency on T_{rt} is very similar to Figures 5a and 5b, the absolute latency values for which edge processing starts to become convenient are significantly smaller. Indeed, the smaller network size makes edge inference faster. Therefore, local processing of short input sequences becomes convenient even for smaller T_{rt} values. Although not shown for sake of space, a similar effect would be obtained by having a different ratio of computational power between edge and cloud devices, e.g., by using using a faster edge node equipped with an embedded GPU or a dedicated accelerator.

In conclusion, the range of network latency for which our method yields benefits compared to *both* trivial solutions varies significantly depending on the RNN and dataset, as well as on the relative speeds of the edge/cloud devices.

Due to regression errors, execution time variability and outdated network status information, our runtime may sometime make wrong decisions on where to allocate inference, especially for input sequence lengths around the break-even point. To evaluate this error, we have measured the difference in execution time between the mapping performed by our runtime and an "oracle" policy that always takes the correct decision. On average, this difference is 0.41%, 0.50% and 0.32% for the experiments of Figure 5.

B. Energy Optimization

In this experiment, we assess the effectiveness of our framework when the main objective is energy minimization. To this end, we repeat the same experiments of Section IV-A, but setting $w = 1000$ in (1). For transmission power, we use the 3G model of [15], as this type of connection is still the most common for IoT devices. Results are shown in Figure 6.



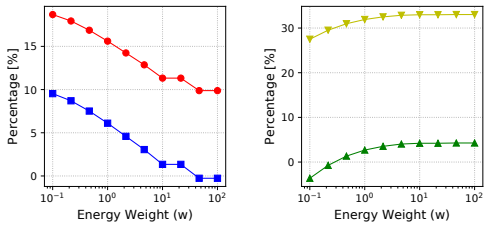
(a) CoVe + SNLI (b) CoVe + SQuAD (c) IMDB

Fig. 6. Energy reduction versus "all edge" and "all cloud" solutions for the for the RNNs in [16] and [17]. Results for different (fixed) network latencies; bandwidth fixed at 1Mbps.

As shown, energy reduction trends with respect to T_{rt} are very similar to the case of time minimization. This is because, for a fixed network bandwidth, transmission power computed with the model of [15] is a constant, similarly to computation power. In particular, with the parameters used in this experiment and a fixed bandwidth of 1Mbps, we obtain $P_{tx} \approx 1.9$W, whereas $P_{edge} \approx 1$W (from Figure 4). Therefore, minimizing energy reduces to minimizing the time spent actively computing/transmitting, with the difference that the least consuming operation (i.e. local computation) is selected more often by the mapping engine. As a consequence, the benefits with respect to a solution that always transmits (i.e. the cloud-only approach) increase. The average energy overheads compared to an oracle policy are 0.06%, 0.07% and 0.32% for this experiment.

C. Combined Optimization

Energy and time minimization can be combined in the proposed runtime by using intermediate values of w in (1). To show the effect of this parameter, we run multiple inferences on the CoVe network with the SNLI dataset, varying w from 0.1 to 100 (on a logarithmic scale), while setting fixed network latency and bandwidth values (200ms and 1Mbps). The results of this experiment are shown in Figure 7.



(a) Ex. Time (b) Energy

Fig. 7. Total execution time and energy reduction versus "all edge" and "all cloud" solutions for CoVe on the SNLI dataset. Results for different values of the objective function parameter w. Network latency and bandwidth fixed at 0.2s and 1Mbps. Same legend of Figures 5 and 6.

For small values of w the mapping engine favors execution time reduction (20% w.r.t edge and 10% w.r.t. cloud) at the expense of energy, which actually *increases* with respect to an edge-only solution (left of Figure 7b). In other words, the mapping engine decides to transmit long input sequences to the cloud for fast processing, even if this is less efficient than computing locally. As w increases, however, the energy benefits do as well, whereas time benefits progressively reduce. Notice that, for many intermediate points, our system yields simultaneous benefits in both quantities.

D. Online Adaptation

The advantage of our framework is even more evident when the connection status changes over time, as the runtime can adapt and dynamically change the selected device for a given input length. We show this through two additional experiments on the CoVe network.

First, we generate an artificial time-varying network profile, keeping the bandwidth fixed at 1Mbps and increasing the latency from 100ms to 200ms at about 1/3 of the simulation and from 200ms to 300ms at 2/3. This controlled profile allows us to make considerations on the adaptability of our solution.

Second, we use the method in [19] to mimic real T_{rt} values using data from RIPE Atlas, an open source database of Internet measurements. Specifically, we use a record of ping times among two random nodes in the database, selected simply because they yielded a time-varying T_{rt} profile. We use a 3-hour-long T_{rt} record to cover the entire time required for running 100k inferences on SNLI/SQuAD, even with the two trivial approaches (edge-only and cloud-only). In this time-span the measured T_{rt} varies between 100ms and 245ms.

In these experiments, the energy weight in (1) is set to $w = 1$ to obtain a balanced reduction of both energy and time. The results are reported in Table I, and the exact parameters of the RIPE Atlas query are listed in the caption.

For the artificial T_{rt} profile, our method achieves significant speed-ups and energy reductions with respect to the two trivial solutions, as both are strongly sub-optimal at different times: initially, offloading to cloud is convenient due to the small latency, whereas at the end of the simulation the cloud solution suffers from long delays and consequently also consumes significantly more. In contrast, our method adapts to the current network status and selects the best approach at all times. Although results on the RIPE Atlas profile are slightly less good (due to the less dramatic variation of T_{rt} in the selected interval), trends are very similar, demonstrating the effectiveness of our optimization in a realistic scenario.

TABLE I
RESULTS FOR VARIABLE NETWORK LATENCY. RIPE ATLAS MEAS. ID: 1437285, PROBE ID: 6222, DATE AND TIME: MAY 3RD 2018, 3-6 P.M.

Profile	Dataset	Ex. Time Reduction [%]		Energy Reduction [%]	
		vs Edge	vs Cloud	vs Edge	vs Cloud
Artificial	SNLI	23.43	14.81	11.25	37.90
	SQUAD	13.78	24.30	4.94	47.88
RIPE Atlas	SNLI	16.33	8.02	5.28	33.90
	SQUAD	5.16	16.58	1.67	45.43

V. CONCLUSIONS

We proposed a novel runtime framework to perform collaborative RNN inference between edge and cloud. Our method selects the optimal device on which to perform inference based on a characterization of the NN, on the length of the input to be processed and on the current status of the communication network. In the future, we plan to test our runtime on faster edge devices (e.g. a smartphone with an integrated GPU). We also plan to extend our runtime to react to variations in the cloud server load.

REFERENCES

[1] I. Goodfellow et al, *Deep Learning*, 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] 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.
[4] B. Moons et al, "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI," in *IEEE ISSCC*, feb 2017, pp. 246–247.
[5] D. Jahier Pagliari and M. Poncino, "Application-Driven Synthesis of Energy-Efficient Reconfigurable-Precision Operators," in *IEEE ISCAS* 2018, pp. 1–5.
[6] J. Kung et al, "Peregrine: A Flexible Hardware Accelerator for LSTM with Limited Synaptic Connection Patterns," in *DAC* 2019, pp. 209:1–209:6.
[7] S. Cao et al, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *FPGA* 2019, pp. 63–72.
[8] H. Tann et al, "Runtime configurable deep neural networks for energy-accuracy trade-off," in *CODES '16* 2016, pp. 1–10.
[9] D. Jahier Pagliari et al, "Dynamic Bit-width Reconfiguration for Energy-Efficient Deep Learning Hardware," in *ISLPED* 2018, pp. 47:1—-47:6.
[10] D. Jahier Pagliari et al, "Dynamic Beam Width Tuning for Energy-Efficient Recurrent Neural Networks," in *GLSVLSI* 2019, pp. 69–74.
[11] B. Taylor et al, "Adaptive Deep Learning Model Selection on Embedded Systems," in *LCTES* 2018, pp. 31–43.
[12] H. Yin et al, "A Hierarchical Inference Model for Internet-of-Things," *IEEE TMSCS*, vol. 4, no. 3, pp. 260–271, 2018.
[13] A. Thomas et al, "Hierarchical and Distributed Machine Learning Inference Beyond the Edge," in *ICNSC* 2019, pp. 1004–1009.
[14] Y. Kang et al, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," in *ASPLOS* 2017, pp. 615–629.
[15] A. E. Eshratifar et al, "BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services," *CoRR*, vol. abs/1902.0, 2019.
[16] B. McCann et al, "Learned in translation: Contextualized word vectors," *CoRR*, vol. abs/1708.00107, 2017.
[17] https://github.com/keras-team/keras
[18] M.-R. Ra et al, "Odessa: Enabling Interactive Perception Applications on Mobile Devices," in *MobiSys* 2011, pp. 43–56.
[19] M. Mouchet et al, "Statistical Characterization of Round-Trip Times with Nonparametric Hidden Markov Models", in *IEEE IM*, 2019, pp. 43–48.