Optimal Input-Dependent Edge-Cloud Partitioning for RNN Inference

Optimal Input-Dependent Edge-Cloud Partitioning for RNN Inference

Daniele Jahier Pagliari, Roberta Chiaro, Yukai Chen, Enrico Macii and Massimo Poncino
Politecnico di Torino, Turin, Italy
Email: name.surname@polito.it

***Abstract*—Recurrent Neural Networks (RNNs) such as those based on the Long-Short Term Memory (LSTM) architecture are state-of-the-art deep learning models for sequence analysis. Given the complexity of RNN-based inference, IoT devices typically offload this task to a cloud server. However, the complexity of RNN inference strongly depends on the length of the processed input sequence. Therefore, when communication time is taken into account, 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 decision automatically. Results based on performance profiling of real edge and cloud devices show that our method is able to reduce the total execution time of the system by up to 20% compared to solutions that execute the RNN inference fully locally or fully in the cloud.**

I. Introduction and Related Work

Deep Neural Networks (DNNs) have become the state-of-the-art models for a variety of machine learning (ML) tasks; in particular, Recurrent Neural Networks (RNNs) based on the Long-Short Term Memory (LSTM) model are increasingly used for advanced sequence analysis, e.g. in speech recognition, neural machine translation, sentiment analysis, etc.

The high accuracy achieved by DNNs on these tasks comes at the cost of a significant computational complexity for training and inference [1]. Therefore, currently both tasks are typically executed in high-performance cloud servers equipped with GPUs and multi-core CPUs. While this is acceptable for training, which is often a one-time task, 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 [1]–[14].

To enable the *complete* execution of DNN inference at the edge, many researchers have proposed fast and energy-efficient hardware accelerators [1]–[7]. These specialized designs exploit the parallelism of the matrix multiplication kernels that dominate DNN inference, and leverage techniques such as weight quantization and pruning to reduce the memory bottleneck. While initially being focused mostly on Convolutional Neural Networks (CNNs), research on DNN acceleration has recently started to consider also RNNs/LSTMs [6], [7].

More recently, however, other works have shown that, in most instances, the energy- and latency-optimal solution is not obtained performing inference entirely on the edge node or on the cloud, but rather partitioning of the computation between the two [12]–[14]. The work in [12] proposes a 3-level hierarchical ML framework for multiple-source data, in which sensors, 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. In such a way, the amount of data transmitted between levels is dramatically reduced, positively impacting latency and energy consumption, at the cost of a possible decrease in accuracy. In [13] the authors present a similar approach, in which 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, thus allowing the corresponding output activations to be produced at the edge. The subsequent layers aggregate activation vectors from multiple sensors and are executed in the cloud. This solution simplifies the design of the NN, which can be directly trained end-to-end with back-propagation. The authors of [14] focus on single-source computer vision applications and propose to perform CNN-based inference on the edge only up to a certain layer, and complete it in the cloud. A runtime environment is proposed to determine the optimal split-layer depending on the network conditions and on the load of the cloud server.

All these works perform *input-independent* design choices. However, it has been shown that *input-dependent* optimizations of DNN inference can yield superior energy, latency, and accuracy results [8]–[11]. In fact, different inputs may require a different number of features [8], a different weight quantization [9], a different inference algorithm [10] or even a different NN [11]. This is particularly true for RNNs, whose complexity strongly depends on input length [10].

In this work, we propose a framework that applies input-dependent optimizations to the problem of edge-cloud partitioning of DNN inference, focusing in particular on LSTM-based recurrent networks; to the best of our knowledge, ours is the first work addressing this problem. Our method decides at runtime whether to perform inference locally on the edge or remotely in the cloud, *depending on the length of the processed input sequence, as well as on the current context of the system (e.g. network latency)*. Results based on the profiling of real edge and cloud devices show that our method can reduce the total inference time by up to 20%, compared to solutions running fully locally or fully in the cloud.

II. Proposed Method

A conceptual scheme of our framework is shown in Figure 1. We propose to add a small runtime (*Mapping Engine*) on the edge node, in charge of deciding where to perform a RNN-based[1] inference between the node itself and 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, hence both can execute an entire inference independently. In general, the two devices can leverage different inference engines, such as the light-weight ARM-NN for the edge node, and the more flexible Tensorflow for the cloud.

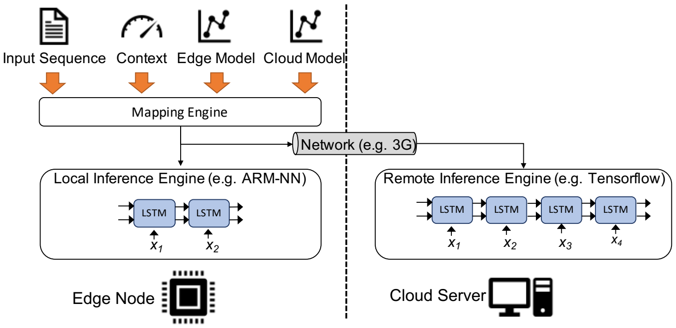


Fig. 1. Conceptual scheme of the proposed framework.

The runtime takes four inputs, i.e.:

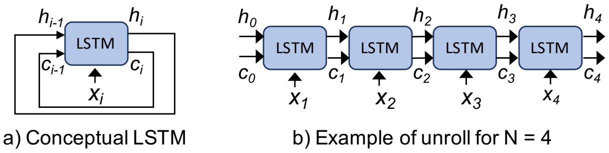
- The input sequence to be processed by the RNN.
- Two regression models to forecast the inference execution time, one for the edge node and one for the cloud server.
- Context information, such as the status of the network link (e.g. 3/4G or WiFi) connecting edge and cloud.

Using this information, it selects whether to perform inference locally or on the cloud for that input, with the goal of minimizing the total latency. The main features of the framework are described in the following sections.

A. *Edge and Cloud Execution Time Modeling*

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. As shown in Figure 2a for a LSTM, the two output vectors produced by the network when processing the i-th input of the sequence (x_i), called *cell state* (c_i) and *hidden state* (h_i), are fed-back to the network at step $i+1$. The functional details of RNN/LSTM *cells* (blue blocks) are omitted for sake of space; the reader can refer to [15]. In practice, when performing inference on an input sequence of length N, the LSTM is *unrolled* (i.e. replicated) N times, as shown in Figure 2b. Each copy of the LSTM performs the same operations and shares the same weight matrices (learned during training).

Figure 2b suggests that the computational complexity for inference in a LSTM grows linearly with N. Moreover, although each LSTM block performs a highly-parallel matrix 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]. Thus, inference execution time also grows linearly with respect to input length. Based on this analysis, our runtime should have at hand, for a given input length, an estimate for edge/cloud execution time, in order to determine where to run the inference. Such estimate can only be built empirically as it is device and NN-dependent. To this end, the target LSTM is first characterized on both devices, in order to extract two linear regression models (*Edge* and *Cloud Model* in Figure 1). Figure 3 shows the results of this characterization for two example devices. Each dot represents the mean execution time over 100 inferences for a given input length, for the 2-layer LSTM of [16].



a) Conceptual LSTM

b) Example of unroll for N = 4

Fig. 2. Example of LSTM unrolling.

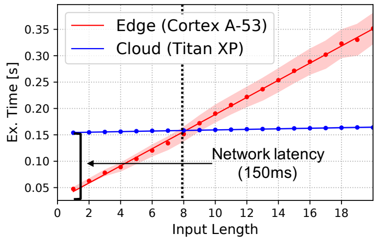


Fig. 3. Execution time versus input length for the CoVe network [16]. Points and colored areas represent means and standard deviation intervals over 100 inferences. Lines are linear regression fits. Regression scores: edge $MSE = 9.32 \cdot 10^{-6}$, $R^2 = 0.999$, cloud $MSE = 7.69 \cdot 10^{-8}$, $R^2 = 0.991$.

As an edge device (red dots and curve) we consider an ARM Cortex A-53 @ 1.2GHz, 1GB RAM, Linux OS. The shaded area around the dots defines the standard deviation of the execution time, showing the low variability of the results. The solid red line represents the best linear fit of the data; fitting scores are reported in the caption.

Estimating cloud execution time is less straightforward, as besides computation, communication also has an impact. The total time for cloud processing T_{cloud} can be modeled as:

$$T_{cloud} = T_{rt} + \frac{S(N,M)}{B} + T_{server}(N) \qquad (1)$$

where T_{rt} is the round-trip network latency, $S(N,M)$ is the size in bytes of the input (N elements) and output (M elements) data, B is the network bandwidth and $T_{server}(N)$ is the server inference time. Due to the typical small input/output sizes of RNNs (in the order of 100s of Bytes [15]), T_{cloud} is dominated by T_{rt}. As an example, assuming $S(N,M) = 100$ Bytes are transmitted on a 3G link with $B = 1$ Mbps, the second term in (1) becomes 0.8 ms. Assuming a typical value of T_{rt} of 150 ms, this is $\approx$ 200x larger than the transmission time. Therefore, for most RNN applications, total communication time will be bound by network latency, which is virtually *independent of the input size*.

[1] In this work we focus on LSTM networks, but our framework also applies to other types RNNs (e.g. vanilla RNNs and GRUs). Therefore, we will use the terms LSTM an RNN interchangeably.

This is shown in the blue curve of Figure 3. As an example of cloud device, we use a NVIDIA Titan XP GPU on a server-class platform, i.e. 32-thread Intel Xeon E5-2630 CPU @ 2.40GHz, 128GB RAM, Linux OS. When drawing the curve, we assumed the network parameters described above, which shift all execution times up on the y axis due to the effect of latency. 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 plaform, but the dependence is still almost perfectly linear, as shown by the scores in caption. For this device, the execution time standard deviation interval is too small to be visible in the figure.

Overall, Figure 3 shows the purpose of our runtime: for short input sequences ($N < 8$ in this specific instance), edge processing is faster, whereas cloud offloading becomes preferable for longer ones.

B. Online Adaptation

The exact *break-even* point of the previous analysis depends on the state of the system, and in particular on network status, which varies over time. In order to adapt our runtime's decisions to variations in the network connection we use two different mechanisms. First, the latency information is updated every time the runtime decides to perform an inference in the cloud, using timestamps added by the edge node and by the cloud server upon sending/receiving the 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 latency estimate. This second mechanism is needed in cases when a sporadically large increase in latency shifts the blue curve of Figure 3 so high that the break-even point moves towards input sizes that never occur in practice, and therefore our runtime starts to *always* select local processing. In that situation, the timestamp mechanism would never take place, and the runtime would not notice when the latency decreases again.

In this work, we assume edge and cloud inference times to be constant. However, our framework can be extended to also support load variations in the two devices, e.g. using the methods proposed in [14], as another form of context information. This extension will be object of our future work.

III. Experimental Results

We tested the proposed methodology on the edge and cloud platforms described in Section II, i.e. ARM Cortex A-53 CPU and Intel Xeon + NVIDIA Titan XP respectively. We selected the CoVe network as a baseline architecture [16]. This NN is composed of a 2-layer LSTM and is used to process sequences for a variety of NLP tasks, such as sentiment analysis, question classification/answering, etc. For our experiments, we used two of the datasets considered by the original authors of [16], SNLI (entailment) and SQuAD (question answering). The same CoVe architecture and weights have been loaded in the edge and cloud devices using TensorFlow. As our method has no direct competitor, we compared it against the two trivial solutions, i.e. running inference fully in the edge or in the cloud.

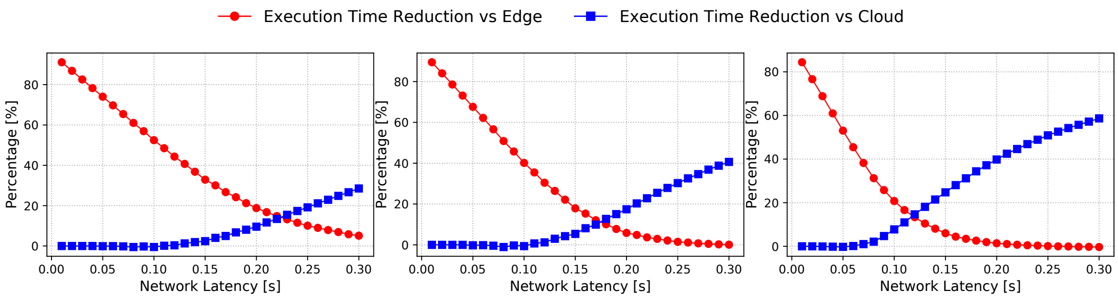
We used a Python-based simulator to evaluate the impact of our runtime on the total execution time of the system. With respect to testing with the real on-device runtime, the simulator allows us to assess the impact of different *predictable network conditions* on the effectiveness of our methodology.

The simulator receives as input the two regression models (generated as discussed in Section II-A) and a network connection profile, formatted as a time series of latency/bandwidth pairs. It then analyzes a set of sequences as if they were fed to the system one after another, and for each sequence it selects the optimal inference target (edge vs. cloud) based on the current network status and on the two regression models. The impact of the selection is then evaluated by computing the total processing time for each sentence (including communication time), accounting for the difference between the regression estimate (t_{pred}) and the real inference time (t_{real}). The former is obtained from the models, whereas the latter is *measured on the real edge and cloud devices*; thus, we account for both inference time variability and regression errors.

We also simulate the fact that network information 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. The input-dependent impact of network bandwidth on communication time, although negligible, is also taken into account. Finally, the time overhead of the runtime itself is negligible, even compared to the execution of 1-input sequences on the edge node, as it only consists in a look-up of the two regression models and a simple equation evaluation.

As a first experiment, Figures 4a and 4b show the results of simulating the inference of 100k random sentences from the SNLI and SQuAD datasets. Specifically, each graph reports the *reduction of the total inference time* with respect to a fully edge or cloud solution, for different values of network latency (assumed *fixed* for the entire simulation). We consider a reasonable latency range for connections such as 3G (10s-100s of ms), which are the most commonly used in IoT. 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.

Figure 4c shows the impact of the *size* of the LSTM model; this graph has been obtained as for Figure 4a, but using a version of CoVe that only contains 200 LSTM units in each layer instead of the original 300. The smaller NN makes inference faster on both platforms, but in particular on the edge. Therefore, the execution time reduction curves tend to



(a) SNLI Dataset

(b) SQuAD Dataset

(c) SNSLI Dataset (200 Units)

Fig. 4. Total execution time reduction versus “all edge” and “all cloud” solutions for the CoVe network [16]. Results for different (fixed) network latencies; bandwidth fixed at 1Mbps (download) and 200kbps (upload).

shift to the left, as a consequence of the fact that edge-side inference for short input sequences becomes convenient even for smaller network latency values. As a result, the benefit with respect to a full cloud solution increases for large latency values (59% at 300ms versus 28% of Figure 4a). 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 with an embedded GPU. Thus, the range of network latency for which our method yields benefits compared to *both* trivial solutions will vary significantly depending on the size of the LSTM and 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, for each run, the difference in inference time between the choice returned 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 three scenarios of Figure 4.

TABLE I
RESULTS FOR VARIABLE NETWORK LATENCY

Dataset	N. Units	Reduction wrt Cloud	Reduction wrt Edge
SNLI	300	17.3	25.3
SNLI	200	43.7	7.2
SQUAD	300	26.0	15.2
SQUAD	200	52.8	2.4

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. To show this, we have performed inference with the same data used for Figure 4, but assuming a time-varying network profile. Specifically, we kept the bandwidth fixed at 1Mbps (download) and 200kbps (upload) and let the latency increase from 100ms to 200ms at about 1/3 of the simulation and from 200ms to 300ms at 2/3. The reduction of execution time in this scenario for the two datasets and two LSTM sizes are reported in Table I. Our method achieves significant speed-ups with respect to the two trivial solutions, as both are strongly sub-optimal in different moments: initially, executing on the edge is not convenient due to the small latency, whereas at the end of the simulation the cloud solution suffers from long delays. In contrast, our method adapts to the current network status and selects the best approach at all times, i.e. always cloud when latency is 100ms, always edge for 300ms, and a “mix” depending on input length for 200ms.

IV. Conclusion

We proposed a novel runtime framework to perform collaborative LSTM 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 extend our runtime to also consider energy minimization as an objective, and to react to variations in the cloud server load, as in [14].

References

[1] 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.

[2] 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.

[3] 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 *ISSCC*, 2017, pp. 246–247.

[4] D. Jahier Pagliari and M. Poncino, “Application-Driven Synthesis of Energy-Efficient Reconfigurable-Precision Operators,” in *ISCAS*, 2018, pp. 1–5.

[5] D. Jahier Pagliari et al, “A methodology for the design of dynamic accuracy operators by runtime back bias,” in *DATE*, 2017, pp. 1165–1170.

[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*, 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] I. Goodfellow et al, *Deep Learning*. The MIT Press, 2016.

[16] B. McCann et al, “Learned in translation: Contextualized word vectors,” *CoRR*, vol. abs/1708.00107, 2017.