Dynamic Beam Width Tuning for Energy-Efficient Recurrent Neural Networks

Dynamic Beam Width Tuning for Energy-Efficient Recurrent Neural Networks

Daniele Jahier Pagliari, Francesco Panini, Enrico Macii and Massimo Poncino
Politecnico di Torino
Turin, Italy
name.first_surname@polito.it

ABSTRACT

Recurrent Neural Networks (RNNs) are state-of-the-art models for many machine learning tasks, such as language modeling and machine translation. Executing the inference phase of a RNN directly in edge nodes, rather than in the cloud, would provide benefits in terms of energy consumption, latency and network bandwidth, provided that models can be made efficient enough to run on energy-constrained embedded devices.

To this end, we propose an algorithmic optimization for improving the energy efficiency of encoder-decoder RNNs. Our method operates on the Beam Width (BW), i.e. one of the parameters that most influences inference complexity, modulating it depending on the currently processed input based on a metric of the network's "confidence".

Results on two different machine translation models show that our method is able to reduce the average BW by up to 33%, thus significantly reducing the inference execution time and energy consumption, while maintaining the same translation performance.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Hardware** → **Power estimation and optimization**; • **Computer systems organization** → *Embedded software.*

KEYWORDS

Energy Efficiency; Neural Networks; Deep Learning

1 INTRODUCTION

Deep learning models are increasingly being used in many machine learning applications [1]. In particular, Recurrent Neural Networks (RNNs) are now able to deliver state-of-the-art performance in sequence modeling tasks such as machine translation and image captioning [2]. Unlike traditional feed-forward deep neural networks, RNNs have memory and can handle variable-length inputs and outputs.

The high quality of the results achieved by these models, however, comes at the cost of high computational complexity. Currently, the execution of deep learning models is mostly performed in the cloud, using multicore CPUs and clusters of GPUs which consume hundreds of watts of power [3, 4]. On the other hand, there is an increasing demand for intelligent behavior in "edge" nodes (mobile devices, IoT sensors, wearables), which are typically battery-powered and have limited energy budget and computational power. Implementing neural networks directly on the device would provide clear benefits in terms of network bandwidth, response latency and ultimately energy consumption [1]. This applies in particular to the *inference* phase, i.e. the actual classification, whereas the *training* can conveniently be left to the cloud, as a unique (or sporadic, in the case of re-training) task [1, 5] As a result, the energy-efficient design of neural networks and the development of optimized inference methods are acquiring a key role for the development of new applications [4, 6].

In literature, many works have tackled this problem proposing different solutions [1, 3–15]. However, the great majority of these works focuses on Convolutional Neural Networks (CNNs), while only few consider RNNs [6, 14].

One approach that has proven effective for CNNs is to exploit the trade-off between quality of results (e.g. classification accuracy) and model complexity. Complexity reductions can be obtained in different ways, ranging from bit-width reductions in data and computations to algorithmic optimizations. Most of these studies propose *static* solutions, in which the complexity is kept constant throughout the whole inference. However, recent works have shown that a static approach is suboptimal, since for most classification tasks not all inputs are equally difficult to process. *Dynamic* solutions have therefore been devised to exploit this data dependency: here, the complexity of the model is adapted to the currently processed data at runtime, based on some metric of the "confidence" of the output produced by the network [3, 5, 7].

In this work, we take a similar approach, but we apply it for the first time to RNN models, focusing specifically on encoder-decoder networks for Neural Machine Translation (NMT). We propose a method to dynamically tune one of the main parameters of the network, the so-called Beam Width (BW), depending on the currently processed data. Increasing the BW is beneficial for the quality of the translations produced by a RNN, but also has a strong impact on the computational complexity of the inference task. In order to optimize this trade-off, we propose to monitor the translation "confidence" in each decoding step, and adapt the BW accordingly. This allows to reduce the average BW by up to 33% with respect to a fixed-BW baseline, while producing comparable or even better outputs. Considering a single-threaded SW implementation of the target RNNs, i.e. the most common scenario for embedded devices, this corresponds to a 25% reduction of the average inference execution time, which in turn translates into significant energy savings for a fixed translation throughput.

2 BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Encoder-Decoder RNNs. A comprehensive description of the existing types of deep neural network can be found in [16]. Herein, we only focus on Recurrent Neural Networks based on the encoder-decoder structure, i.e. the main targets of this work, which are state-of-the-art models for sequence modeling and translation tasks [2].

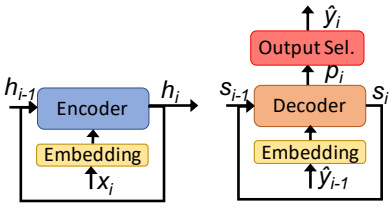


Figure 1: Architecture of an encoder-decoder RNN.

A high-level diagram of the architecture of an encoder-decoder RNN is reported in Figure 1. The *Encoder* and *Decoder* blocks contain two separate neural networks, with one or more layers of neurons each. The internal structure of these neurons is typically different from that of feed-forward networks; common models include the so-called *Vanilla RNN*, the *Gated Recurrent Unit* (GRU) and the *Long-Short Term Memory* (LSTM). The details of these models are immaterial for our method, and the reader is referred to [16] for the details. The *Embedding* layers at the input of both RNNs are typical in NMT applications; they are used to map words from the input/output vocabularies (e.g. English and German in the following example) onto a compact representation as arrays of floats, while respecting semantic similarities (i.e. similar words produce similar embeddings).

To perform a sequence-to-sequence mapping, first the Encoder and then the Decoder are executed multiple times. This process is described herein with the terminology of NMT (sentences, translations, etc.), but the same concepts apply also to other applications. To translate a sentence, initially the Encoder is iteratively fed with all input words $X = x^{<1>}, ..., x^{<T>}$ in order. In each iteration, the Encoder updates its *hidden state* vector (h), which is then fed-back to the network in the following step (similar to a FSM). This feedback, also present in the decoder, provides RNNs with memory and is their peculiarity with respect to feed-forward networks. When the last word has been processed, the corresponding Encoder hidden state contains a fixed-length representation of the input sentence, generally called *context* (C).

The context is then used to initialize the hidden state (s) of the Decoder, whose task is to produce the *predicted* output sentence $\hat{Y} = \hat{y}^{<1>}, ..., \hat{y}^{<T'>}$. Specifically, besides updating its internal state, the Decoder also produces an output $p^{<i>}$ at every step. This is an array of the same size as the output vocabulary, containing in its k-th position the likelihood of a sentence that contains all words predicted in previous steps, and that has the k-th word from the vocabulary as its i-th element. This likelihood is conditioned on the input sentence, represented by the context vector. Mathematically: $p^{<i>} = [p(\hat{y}^{<1>}, ...\hat{y}^{<i-1>}, y^{<i>} = y_k|X)], \forall k$. To model the dependency on previous predictions, the Decoder is also fed with the latest predicted word $y^{<i-1>}$, initialized at NULL in the first iteration. The block labeled *Output Sel.* is in charge of selecting the predicted output word(s) based on the likelihoods in $p^{<i>}$, as detailed in Section 2.1.2.

An example of the functionality of this type of RNNs is depicted in Figure 2a, where horizontal copies of the Encoder and Decoder represents executions of the *same* network in different iterations. In general, input and output sentences may have different lengths, so their termination is typically signaled by a special *End of Sentence* (<*EOS*>) value. During training, the parameters of the Encoder and Decoder RNNs are tuned to maximize $p(y^{<1>}, ..., y^{<T'>}|X)$, where $y^{<1>}, ..., y^{<T'>}$ are the elements of the *target* output sentence [16].

Several variations of this basic scheme have been proposed in order to improve performance. Notable examples include *bidirectional* RNNs, in which the hidden state can capture information coming both from previous and following words and *attention-based* RNNs, in which an additional layer is used to weigh the "importance" of each input element for the generation of a given output [2, 16]. The detailed description of these advanced models is out of the scope of this paper.

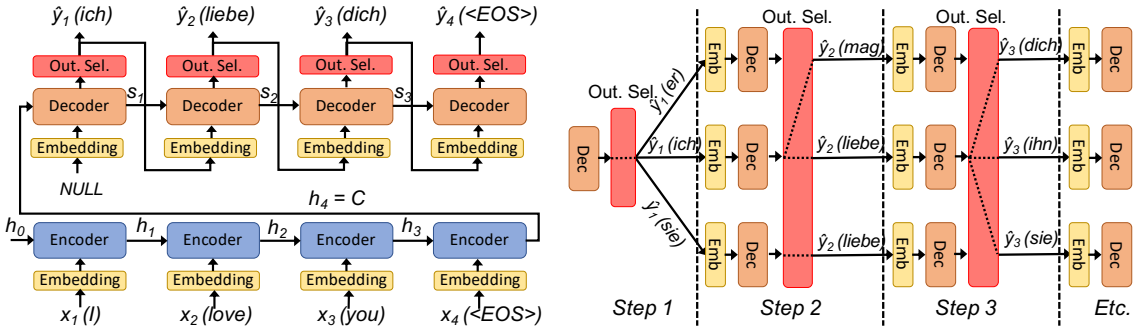
2.1.2 Beam Search. Basic encoder-decoder RNNs, such as the one of Figure 2a, produce output sentences using the so-called *Greedy Search* algorithm. That is, the predicted word at every iteration is selected simply as the one that generates the "partial sentence" with the largest likelihood. While this method produces good-enough outputs on average, it can be proven mathematically that the most likely partial sentence at a given decoding step does not necessarily correspond to the beginning of the most likely sentence overall [16]. Finding the most likely translation is actually a NP-complete problem, which involves searching through all possible word combinations. In practice, most real applications of RNNs utilize an intermediate solution called *Beam Search* [16]. Rather than greedily picking a single word, Beam Search considers a fixed number of most likely partial sentences as translation candidates in each decoding step; this number is called the *Beam Width* (BW). Once the decoding phase has completed (i.e. all sentences in the "beam" have reached <EOS>), the final translation is selected as the candidate with the highest joint probability.

An example of Beam Search decoding for BW = 3 is depicted in Figure 2b, where the encoding phase is not reported, being identical to that of Figure 2a. As shown, Beam Search corresponds to expanding a tree of possible translations, adding a level at each iteration and keeping the number of vertices in each level equal to BW. The Embedding and Decoder blocks are identical to those of Figure 2a, but they are both executed BW times per iteration. The main difference is in the output selection, which now takes the probabilities generated by the BW Decoder executions, and selects the words that generate the BW most likely partial sentences. The latter can be generated by any combination of the decoders, as shown by the dashed lines in the figure.

2.2 Related Work

A popular approach to improve the energy efficiency of neural networks inference consists in designing dedicated hardware accelerators for FPGAs [14, 15] or ASICs [3, 6, 8–10]. These designs combine algorithm-level, architecture-level and circuit-level techniques to optimize the most energy consuming operations in the network. The vast majority of these solutions, however, focus solely on feed-forward models and in particular on Convolutional Neural Networks (CNNs) for image recognition. Despite their importance from an application perspective, fewer efforts have been devoted to improving the efficiency of RNNs by acceleration [6, 14].

Despite its effectiveness, hardware acceleration is mostly destined to high-end devices, that can afford specialized SoCs. Vice



(a) Translation example with greedy search.

(b) Beam search with BW = 3.

Figure 2: High-level view of an encoder-decoder RNN.

versa, lower-end embedded systems typically execute RNN inference on general purpose CPUs. While our work focuses on the latter scenario, the proposed algorithmic optimization is agnostic of the underlying platform, and would be equally effective even if applied to the hardware of [6, 14].

At the algorithm level, many studies (again mostly focusing on feed-forward models) apply *approximate computing* techniques to reduce neural networks complexity, exploiting their intrinsic *error resilience*, which allows the injection of significant approximations without dramatically impacting output quality [1, 4, 12, 17]. Common approaches include *quantization*, i.e. the replacement of floating-point data with reduced bit-width fixed-point [4, 11] and *pruning*, in which redundant computations (e.g. entire neurons) that do not affect output quality are eliminated [12, 13]. The former approach is brought to the extreme by Binarized Neural Networks[8, 17] , in which most arithmetic computations are replaced by bit-wise operations, while still retaining good output quality.

All aforementioned works implement *static* energy-vs-quality tradeoffs, in which the amount of approximation (e.g. the fixed point bit-width or the amount of pruning) are decided at design time. More recently, *dynamic* approaches have been proposed [3, 5, 7], starting from the observation anticipated in Section 1 that a static approach may be sub-optimal when inputs are not all equally hard to process. In such cases, a static network would either over-approximate complex inputs, hence producing poor results or under-approximate simple ones, resulting in a waste of energy.

Most dynamic techniques work at the algorithm-level and are hardware independent. The authors of [7] propose a sort of "Big/Little" network, in which two CNNs of different size and complexity are used for an image classification task. When a new input is received, at first the "Little" (least complex) network is executed. According to the confidence of the classification produced, the result is committed as is, or the "Big" network is triggered to provide a more accurate classification. As long as the larger network is employed rarely, total energy consumption is reduced. However, the total model size (number of weights to be stored) increases significantly, and the training effort is substantially doubled. To cope with these limitations, in [3], "Little" networks are generated starting from "Big" ones and using only a portion of each layer. Finally, in [5], a similar dynamic solution is adopted for tuning the bit-width of fixed-point operations depending on the considered input complexity. These dynamic approaches are the most similar to our proposed methodology, yet they are focused on CNNs; to the best of our knowledge, this work is the first to apply data-dependent optimizations for energy efficiency on encoder-decoder RNNs.

3 PROPOSED METHODOLOGY

3.1 Motivation: RNNs Execution Time Analysis

In this work, we focus on optimizing the execution of the inference process on *single thread* CPUs, i.e. the most common computational devices available in edge nodes. As mentioned in Section 2, encoding and decoding in RNNs are iterative processes, involving multiple calls to the same network. Since each step involves the same operations, the power consumption of a CPU will remain approximately constant throughout each phase. Therefore, the most straight-forward way to improve the energy efficiency of the network at the algorithm level is to reduce its execution time.

Based on the analysis of Section 2.1.2, the Beam Width parameter is expected to strongly influence the overall execution time of the decoding phase. In fact, a BW of k corresponds to performing the k decoding operations at each step (Figure 2b); in a single-threaded CPU, these k executions must be carried out sequentially. Typical values of BW found in state-of-the-art models are in the range of 3 to > 5 [18]. Consequently, reducing the BW configures as a promising way to speed up the entire network execution.

To confirm this intuition, we have characterized the dependency on BW of the execution time of two complex RNN models for NMT applications; the networks and the experimental platform are presented in more detail in Section 4. This experiment has been performed on a single core and with a batch size of 1 (i.e. no concurrent processing of multiple sentences) to simulate an embedded scenario. Results are reported in Figure 3, where the vertical axes of the two graphs report the average execution time per sentence over the entire validation subsets, normalized to the result of greedy decoding (i.e. BW=1).

A first observation is that the execution time of the encoding phase, which is expectedly independent from BW, is also negligible with respect to decoding. This gives even more value to the choice of BW as our target. Vice versa, the decoder accounts for a significant portion of the total execution in both networks, and is

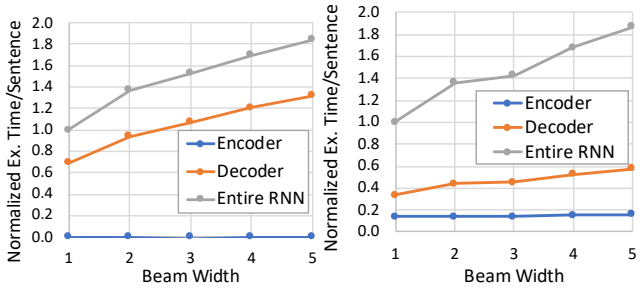


Figure 3: CPU execution time vs. Beam Width for two RNNs.

significantly influenced by BW. The dependency is not perfectly linear due to optimizations in the adopted computational framework; nonetheless, going from BW=1 to BW=5 causes an increase of 84% (EN-DE) and 87% (DE-EN) in the *total* execution time of the networks.

3.2 Input-Dependent Beam Width Tuning

In standard implementations of encoder-decoder RNNs, the Beam Width is kept constant throughout the inference phase. Therefore, this parameter is typically chosen conservatively, so that good enough translations are generated even for the most complex sentences. In consideration of the results presented in Section 3.1, however, the BW should be reduced as much as possible to contain the inference computational cost.

In order to do so without sacrificing translation performance, we propose a novel dynamic Beam Search algorithm, in which *the BW is varied according to the evolution of a translation*. This allows the network to self-tune, at each step, the effort required to produce a good translation. Our method is based on the intuitive concept that not all inputs, sentences or part of them, are equally difficult to translate. For easier, unambiguous sentences, a small BW could be sufficient, and vice versa.

More specifically, we propose to evaluate the outputs of the decoder at each step, and use them as an indicator on how to adjust the Beam Width for the following step. As explained in Section 2.1, decoder outputs represent the likelihoods of partially formed sentences. Therefore, their *distribution* can provide information on the difficulty of a translation. When there are one or few highly likely partial sentences, one of them is probably going to correspond to the final correct output; hence, there is no need to keep many elements in the beam, and BW should be reduced. Vice versa, when the highest likelihoods are all similar (indicating that the network is striving to select a word, e.g. among verbs with different tenses or synonyms), BW should be increased to avoid losing the correct output in favor of partial sentences with higher temporary probability. The details of our proposed policy for mapping decoder outputs to the next BW are presented in Section 3.3.

Figure 4 shows an example of the proposed Dynamic Beam Search, where the blocks labeled *Out. Sel. + Policy* encompass the selection of the predicted word(s) and of the next BW. For example, during Step 2, the decoder may have produced an output with much higher probability than all others, so the policy sets a BW of 1. Then, in Step 3, the distribution of the likelihood is more uniform, so the policy increases the BW to 3.

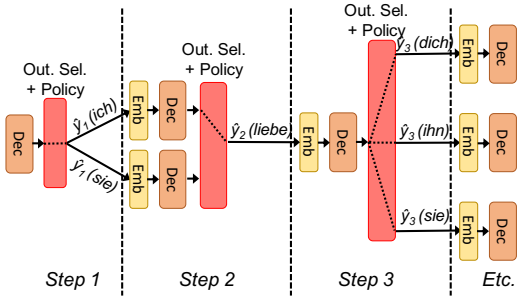


Figure 4: Dynamic Beam Search example.

With this dynamic strategy, we aim at reducing the execution time of the Decoder, while producing comparable or even better results with respect to a standard implementation using a fixed BW. Notice that, although this idea is inspired by previous work on CNNs [3, 5, 7], the knob used to adapt the complexity versus performance trade-off (i.e. the BW in our case) is completely different, as well as the proposed decision policy and off course the type of neural network.

3.3 Beam-Width Selection Policy

Many different policies can be used to map the decoder outputs to a value of BW. In this work, we introduce a simple yet effective approach based on analyzing the distribution of the largest likelihoods; more advanced policies will be object of future work.

The proposed policy considers at each steps the top-k scores produced by the decoder, where $k = BW_{max}$ is the maximum allowed BW and is a parameter of the algorithm. The standard deviation σ of these scores is then computed to evaluate their dispersion; large values of σ correspond to large differences among the top-k scores, and therefore will be mapped to smaller values of BW, and vice versa. Specifically, the mapping is performed using a piece-wise linear equation having σ as the independent variable, shown in Figure 5.

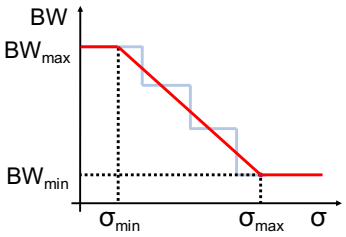


Figure 5: Beam Width selection policy.

Between a minimum and a maximum standard deviation value (σ_{min} and σ_{max}) BW is decreased proportionally to the increase of σ (red diagonal curve). Clearly, BW can not assume fractional values, hence it is rounded to the nearest integer, producing the blue stair-like curve. For values of σ smaller than σ_{min} or larger than σ_{max}, the BW is saturated to BW_{max} and BW_{min} respectively.

The range of allowed Beam Widths can be set depending on the target application: a reasonable value for BW_{min} is 1 (although this

is not mandatory), whereas BW_{max} can be set to any value typically adopted in the target domain. For identifying good minimum and maximum values for σ, instead, designers should collect the standard deviation of the top-k likelihoods produced by the original (fixed BW) network when inferring on representative data. The distribution of these standard deviations can provide a good starting point for exploring the parameters space. For example, selecting as initial σ_{min} the 5-th percentile of the collected standard deviations corresponds to using BW_{max} only in the ≈5% most difficult cases; similarly, a σ_{max} equal to the 50-th percentile corresponds to using BW_{min} in ≈50% iterations. Tuning these parameters requires some exploration, which however can be done offline using multi-core CPUs or GPUs.

4 EXPERIMENTAL RESULTS

4.1 Setup

We have tested our methodology on two complex encoder-decoder RNNs for NMT applications, using the *OpenNMT* framework in its PyTorch implementation [18]. This choice was motivated by the fact that the dynamic computational graph model of PyTorch makes it easier than in other frameworks to dynamically change the BW at runtime. The two selected RNNs perform English to German (Deutsch) and German to English translation, and are hereafter referred to as EN-DE and DE-EN respectively. Both networks are made available by OpenNMT online as pre-trained models. The EN-DE network is composed of a total of 6 layers, each composed of 512 LSTM neurons, whereas the DE-EN network is composed of 2 layers of 500 LSTM neurons. More details on the network architectures and datasets can be found in [18]. The default Beam Width for both models is set to 5.

Parameter explorations have been performed on a workstation equipped with a NVIDIA Titan XP GPU, whereas the measurement of execution times with and without our dynamic Beam Width mechanism have been performed on a laptop equipped with an Intel Core i7 CPU and 32GB of RAM. This choice is due to the fact that no porting of the OpenNMT framework onto embedded devices is currently available. However, in order to get a more accurate estimate of the execution times benefits obtainable on an embedded node, we have constrained all executions to use only one CPU thread and a batch size of 1. Therefore, while absolute execution times collected would not be representative of the actual implementation on an embedded CPU, relative trends should be a reasonable proxy. All tests have been performed on the validation sets of each network. Translation quality has been evaluated using two common machine translation scores, i.e. the *BiLingual Evaluation Understudy* (BLEU) and the *Perplexity* (PPL) [16].

4.2 Comparison with Fixed Beam Width

As we do not have dynamic BW tuning methods to compare against, we use as reference a standard RNN inference using a fixed Beam Width. Specifically, we have compared against BWs ranging from 1 (greedy) to 5.

The results of this comparison are shown in Figure 6. For each RNN we report three graphs; the leftmost two show the trade-off among the average inference execution time per sentence over the validation set, and the average BLEU and PPL scores. Execution times are normalized to the one of the original network with BW = 1. The rightmost graph shows on the horizontal axis the *average Beam Width* over all decoding iterations. For fixed BW inference, this value simply corresponds to the selected BW, whereas for our method, it is the result of the choices made by the mapping policy. This plot has particular relevance since it reports a platform-independent result, which does not depend on the target CPU. Similar plots for the PPL score are not reported for sake of space.

As expected, on the original network (blue curve), increasing the BW generally has a positive effect on both BLEU (for which larger is better) and PPL (smaller is better). It is worth emphasizing that, although the absolute differences in the two scores for the extremes of the range (BW=1 and BW=5) appear to be relatively small, they correspond to significant improvements of the translation quality. As a matter of fact, most state-of-the-art models use BW in the range 3 to 5, indicating that even such small improvements in the metrics are relevant. However, even a small increase of these metrics requires a large increase of execution time. For instance, for the EN-DE network, increasing the BLEU of 0.6 requires a 40% longer execution on average (from BW=1 to BW=2). In contrast, our proposed dynamic BW tuning allows to obtain many more fine-grain settings. This is shown by the orange triangles in the figure, which correspond to some of the Pareto points obtainable by experimenting with policy parameters. The numerical values of the selected parameters (BW_{min}, BW_{max}, σ_{min} and σ_{max}, described in Section 3.3) and the corresponding results are reported in Table 1.

RNN	BW_{min}/ BW_{max}	σ_{min}/ σ_{max}	Avg. BW	Ex. Time	BLEU	PPL
EN-DE	1/2	0.1/2.2	1.36	1.13	32.45	1.568
	1/2	0.1/3.1	1.55	1.17	32.70	1.563
	1/3	0.1/1.7	1.48	1.13	32.58	1.564
	1/3	0.1/3.1	2.02	1.37	32.80	1.554
	2/4	0.1/1.7	2.27	1.40	32.94	1.552
	2/4	0.1/1.3	2.77	1.51	33.02	1.545
	2/5	0.1/1.7	3.33	1.59	33.13	1.543
DE-EN	1/2	0.1/2.2	1.51	1.17	31.16	1.623
	1/2	0.1/3.1	1.69	1.22	31.38	1.612
	1/3	0.1/2.2	1.88	1.27	31.48	1.605
	1/3	0.1/3.1	2.19	1.34	31.61	1.596
	2/4	0.1/0.6	2.35	1.34	31.69	1.600
	2/4	0.1/1.7	2.86	1.51	31.79	1.590
	2/5	0.1/1.7	3.59	1.54	31.80	1.586

Table 1: Numerical results.

Most importantly, the figure shows that our method allows to consistently achieve a better trade-off compared to a fixed BW solution, thanks to the fact that the BW is automatically tuned to the "difficulty" of the current translation. In practice, comparable or even better performance can be achieved (according to both scores) while requiring a lower BW on average. For example, for the EN-DE model, an average BW=3.33 is sufficient to reach a BLEU score of 33.13, superior even to the one achieved by the standard network with BW=5 (33% reduction). On the target platform, this causes a reduction of the average execution time of ≈25%, for the same performance. Similarly, on the DE-EN network, a PPL of 1.61 is obtained with an average BW of 1.69 as opposed to a fixed BW of 2, causing an execution time reduction of ≈11%. As mentioned in Section 3, these execution time reductions correspond to proportional energy savings.

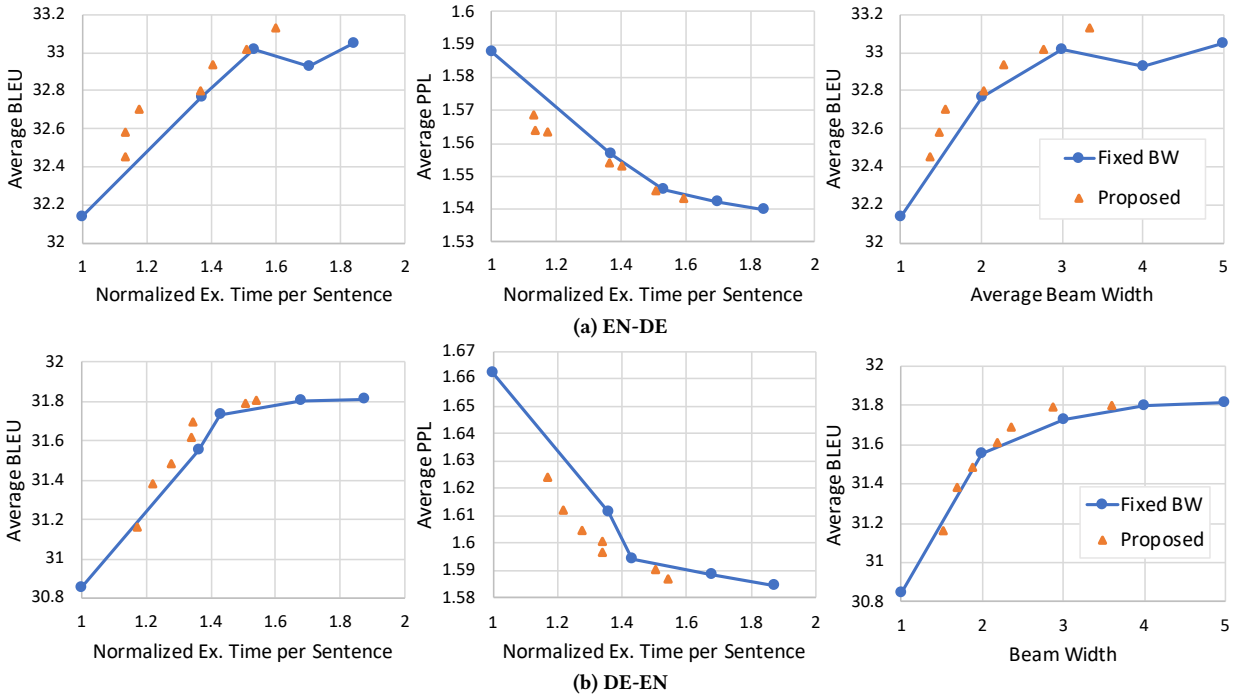


Figure 6: Comparison with a fixed Beam Width approach.

5 CONCLUSIONS

We have proposed a methodology for reducing inference complexity in an encoder-decoder RNN, by dynamically tuning the Beam Width depending on the currently processed input. By applying this technique, we have been able to significantly speedup the translation process (hence also reducing its energy consumption) while achieving comparable or even better output quality. Importantly, our approach can be applied to existing networks without re-training. In future work, we plan on experimenting with more policies for setting the Beam Width, and on integrating our approach with other knobs for exploring the energy-quality trade-off (e.g. quantization).

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] A. Vaswani et al, "Attention is all you need," *Proc. NIPS*, 2017, pp. 5998–6008.
[3] H. Tann et al., "Runtime configurable deep neural networks for energy-accuracy trade-off," *Proc. IEEE/ACM CODES + ISSS*, 2016, pp. 1–10.
[4] B. Moons et al., "Energy-efficient convnets through approximate computing," *Proc. IEEE WACV*, 2016, pp. 1–8.
[5] D. Jahier Pagliari, E. Macii, and M. Poncino, "Dynamic bit-width reconfiguration for energy-efficient deep learning hardware," *Proc. IEEE/ACM ISLPED*, 2018, pp.47:1–47:6.
[6] F. Silfa et a., "E-pur: An energy-efficient processing unit for recurrent neural networks," *arXiv:1711.07480*, 2017.
[7] E. Park et al, "Big/little deep neural network for ultra low power inference," *Proc. IEEE/ACM CODES + ISSS*, 2015, pp. 124–132.
[8] R. Andri et al., "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE TCAD*, vol. 37, no. 1, pp. 48–60, 2018.
[9] J. Zhu et al., "Sparsenn: An energy-efficient neural network accelerator exploiting input and output sparsity," *Proc. DATE*, 2018, pp. 241–244.
[10] Y. Chen, J. Emer and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *Proc. ACM/IEEE ISCA*, 2016, pp. 367–369.
[11] F. Sun, J. Lin and Z. Wang, "Intra-layer nonuniform quantization of convolutional neural network", *Proc. WCSP*, 2016, pp. 1-5.
[12] T. Yang, Y. Chen and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning," *Proc. IEEE CVPR*, 2017, pp. 6071–6079.
[13] Y. Guo, A. Yao and Y. Chen, "Dynamic Network Surgery for Efficient DNNs", *Proc. NIPS*, 2016, pp. 1379–1387.
[14] C. Gao et al., "DeltaRNN: A Power-efficient Recurrent Neural Network Accelerator," *Proc. ACM FPGA*, 2018, pp. 21–30.
[15] C. Zhang et al, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," *Proc. ACM FPGA*, 2015, pp. 161–170.
[16] G. Ian, B. Yoshua, and C. Aaron, *Deep Learning*, MIT Press, 2016.
[17] I. Hubara et al, "Binarized neural networks," *Proc. NIPS*, 2016, pp. 4107–4115.
[18] G. Klein et al, "OpenNMT: Open-source toolkit for neural machine translation," *arXiv:1701.02810*, 2017.