Approximate Computing for Long Short Term Memory (LSTM) Neural Networks

Sanchari Sen and Anand Raghunathan, *Fellow, IEEE*

***Abstract*—Long Short Term Memory (LSTM) networks are a class of recurrent neural networks that are widely used for machine learning tasks involving sequences, including machine translation, text generation, and speech recognition. Large-scale LSTMs, which are deployed in many real-world applications, are highly compute intensive. To address this challenge, we propose AxLSTM, an application of approximate computing to improve the execution efficiency of LSTMs. An LSTM is composed of cells, each of which contains a cell state along with multiple gating units that control the addition and removal of information from the state. The LSTM execution proceeds in timesteps, with a new symbol of the input sequence processed at each timestep. AxLSTM consists of two techniques—Dynamic Timestep Skipping (DTS) and Dynamic State Reduction (DSR). DTS identifies, at runtime, input symbols that are likely to have little or no impact on the cell state and skips evaluating the corresponding timesteps. In contrast, DSR reduces the size of the cell state in accordance with the complexity of the input sequence, leading to a reduced number of computations per timestep. We describe how AxLSTM can be applied to the most common application of LSTMs, *viz.*, sequence-to-sequence learning. We implement AxLSTM within the TensorFlow deep learning framework and evaluate it on 3 state-of-the-art sequence-to-sequence models. On a 2.7 GHz Intel Xeon server with 128 GB memory and 32 processor cores, AxLSTM achieves 1.08 × −1.31× speedups with minimal loss in quality, and 1.12 × −1.37× speedups when moderate reductions in quality are acceptable.**

***Index Terms*—Approximate computing, long short term memory (LSTM) networks, sequence-to-sequence learning.**

I. Introduction

DEEP neural networks (DNNs) have transformed the field of machine learning by greatly advancing the state-of-the-art in a variety of image, video, text, and speech processing tasks [1]–[7]. Today, DNNs are deployed in a spectrum of real-world products and services including Facebook's DeepFace [8] and DeepText [9], Google's voice and image search [10], Apple's Siri [11], etc.

DNNs can broadly be classified into feed-forward neural networks and Recurrent Neural Networks (RNNs). Feed-forward networks like convolutional neural networks and multilayer perceptrons are suitable for tasks with fixed-length, temporally independent inputs and outputs, such as image classification. Feed-forward neural networks have information flowing strictly in the forward direction from the input layer to the output layer, and thus have no memory of previous inputs.

RNNs, on the other hand, can handle variable-length inputs and outputs that form sequences. Text and handwriting synthesis [12], speech recognition [13], Neural Machine Translation (NMT) [14], and image and video captioning [3] are a few examples of applications that deploy RNNs. These networks have a cyclic structure, allowing information to persist temporally in the form of memory in the network. The computation of an RNN can be thought of as proceeding in timesteps with a new element of the input sequence being fed to the network at each timestep. Recent interest in RNNs has been largely fueled by the success of Long Short Term Memory (LSTM) networks [15]. LSTMs and their derivatives have demonstrated an ability to learn long-term dependencies in sequences and are currently deployed in numerous real-world applications. An LSTM comprises of cells, each of which is associated with a cell state and multiple gate units that control the flow of information in and out of the cell.

State-of-the-art LSTMs are highly compute-intensive. For example, NMT of the WMT development set from English to French, using a quantized version of Google's NMT, takes 1322 seconds on a pair of Intel Haswell CPUs [16]. Previous efforts to improve the execution efficiency of LSTMs have explored three distinct directions. The first direction is to efficiently parallelize LSTMs on CPUs and GPUs [16]–[18]. The second direction develops accelerators that exploit the data access and compute patterns of LSTMs [19]–[23]. Finally, the last set of efforts applies model pruning and quantization to reduce the model size for execution on resource-constrained platforms [24]–[26].

We explore *approximate computing* techniques to accelerate the execution of LSTMs. The intrinsic error resilience of machine learning applications makes them excellent candidates for approximate computing, which attempts to reduce execution time and energy with minimal effect on the quality of outputs. Previous efforts have proposed approximate computing techniques for neural networks by utilizing variable bit precision and approximate arithmetic

Manuscript received April 3, 2018; revised June 8, 2018; accepted July 2, 2018. Date of publication July 23, 2018; date of current version October 18, 2018. This work was supported in part by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation Program, sponsored by DARPA and in part by the National Science Foundation under Grant 1423290. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Sanchari Sen.)*

The authors are with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA (e-mail: sen9@purdue.edu; raghunathan@purdue.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2858362

units [27]–[30]. Other works have also proposed approximate computing techniques for biologically inspired spiking neural networks [31]. However, these techniques provide benefits only on specialized hardware architectures or do not specifically exploit the unique characteristics of LSTMs. To the best of our knowledge, we are the first to propose hardware-agnostic approximate computing techniques for LSTMs.

We present AxLSTM, a general approach to approximate LSTMs.[1] AxLSTM consists of two techniques—Dynamic Timestep Skipping (DTS) and Dynamic State Reduction (DSR). DTS is based on the observation that some elements of the input sequence have little or no impact on the state of the LSTM (and hence the output produced). Therefore, identifying such unimportant elements of the input sequence allows the LSTM to skip evaluating them at runtime. On the other hand, DSR is based on the observation that, since the LSTM state is provisioned to capture all possible input sequences, many inputs sequences do not require the full dimensionality of the LSTM state. Thus, the computational complexity of an LSTM can be reduced by dynamically reducing the size of the LSTM state based on the input sequence.

In summary, the key contributions of this paper are as follows.

1) We propose AxLSTM, an approximate computing framework for LSTMs, which combines DTS and DSR.
2) We develop heuristics to dynamically modulate the number of timesteps in DTS, as well as the number of computations per timestep in DSR, based on the input sequence.
3) We implement AxLSTM within the TensorFlow deep learning framework and apply it to state-of-the-art sequence-to-sequence models for NMT and video caption generation. We achieve speedups of $1.08\times - 1.31\times$ with minimal loss in quality, and $1.12\times - 1.37\times$ when moderate reductions in quality are acceptable.

The rest of this paper is organized as follows. Section II presents related research efforts and Section III provides the necessary background on RNNs and LSTMs. The AxLSTM approach and attendant details are presented in Section IV, followed by the experimental methodology in Section V. Section VI presents the results of our experiments. Section VII concludes this paper.

II. Related Work

This paper merges two active fields of research, namely, efficient execution of LSTMs and approximate computing.

A. Efficient Execution of LSTMs

Previous efforts that improve the execution efficiency of LSTMs can broadly be grouped into three directions. The first direction explores efficient parallelization on CPUs and GPUs. These include software-pipelined GPU implementations that exploit coarse-grained parallelism between the computations for hidden states and outputs [17], multilayered implementations that are partitioned across multiple GPUs [16] and CPU-based implementations that merge matrix multiplication operations for more efficient execution [18].

The second direction explores specialized accelerators that improve the execution of LSTMs by exploiting their unique data access and compute patterns [19]–[23]. These implementations utilize dedicated compute units to perform the different operations of an LSTM, specialized communication modules feeding the compute units, and balanced pipeline structures that overlap computation and communication.

Finally, the third set of efforts reduce the model size of LSTMs through quantization and pruning to derive additional time and energy benefits [24]–[26]. These techniques enable the execution of LSTMs on storage-constrained devices, and reduce the bandwidth required to transfer models across a network.

B. Approximate Computing

The intrinsic error resilience of machine learning applications in different domains like recognition, vision, search, and data analytics allows them to produce outputs of satisfactory quality even when some of the computations are approximated. Approximate computing techniques applicable to different levels of the hardware stack, from circuits to architecture and software, have been proposed over the years [32].

A majority of previous efforts that apply approximate computing to neural networks [27]–[30] utilize reduced precision and approximate arithmetic units. As a result, they require specialized hardware implementations and do not benefit software implementations. Other works propose general approximate computing techniques for biologically inspired spiking neural networks [31] which are not applicable to LSTMs. This paper proposes new approximate computing techniques that improve the execution time of LSTMs by taking advantage of their unique structure and characteristics. The proposed approximations are generic in nature and applicable to both software and hardware implementations.

III. Background

This section provides a brief review of RNNs, LSTM networks and sequence-to-sequence models.

A. Recurrent Neural Networks

RNNs are a class of neural networks designed to process sequential information with the help of memory or state. Individual symbols of an input sequence are presented to the RNN at each processing *timestep*. These inputs are used to modify the network state, thereby accumulating information from the past.

The basic structure of an RNN is shown in Fig. 1(a). It operates on the tth symbol, x_t, of the input sequence, x, at timestep t and modifies the state h_t, before feeding it back to the network at time $t+1$. The network is trained through the backpropagation through time algorithm, performing backpropagation on an RNN unrolled into multiple timesteps [Fig. 1(b)]. Several RNN models, with varying levels of ability to model sequences, have been proposed over the years [15], [33], [34]. We focus on the most commonly used model, called LSTM

[1] Although we focus on LSTMs due to their widespread use, the proposed techniques are applicable to any RNN.

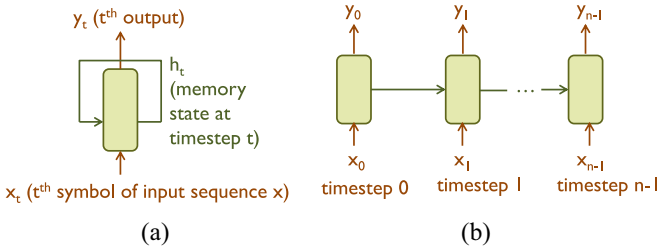

Fig. 1. (a) Basic RNN (b) Time unrolled RNN.

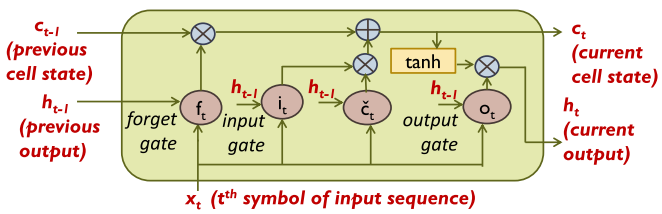

Fig. 2. LSTM cell.

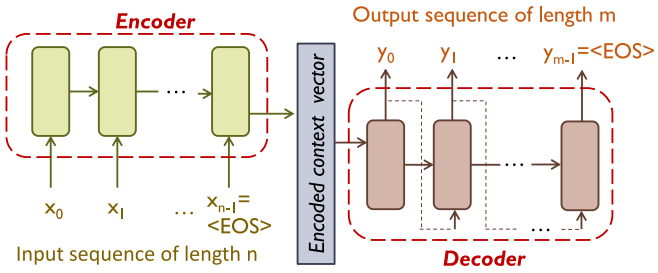

Fig. 3. Sequence-to-sequence model.

networks. However, our approximations can be applied to any RNN model that has clearly identifiable state and timesteps.

B. Long Short Term Memory Networks

LSTMs [15] represent a special class of RNNs known for their ability to effectively learn long-term dependencies in sequences. Recent interest in RNNs has been largely fueled by LSTMs, and LSTMs and their variants account for most practical applications of RNNs. An LSTM is composed of cells. The structure of a cell is illustrated in Fig. 2. Each cell has an associated state referred to as the cell state, c_t. Carefully regulated structures, called gates, control the addition and removal of information from the cell state. A gate is a neural network layer with a sigmoid activation function, followed by pointwise multiplication. The sigmoid output, with values between 0 and 1, dictates how much of each component should be let through for pointwise multiplication with the cell state. The forget gate, f_t, determines how much of the previous cell state, c_{t-1} should be passed on to the current time step. A new set of candidate values for the cell state, $\hat{c}_t$, is produced by a tanh layer. Subsequently, the input gate i_t determines the fraction of new candidate values to be added to the current cell state. Finally, the output gate, o_t, controls which parts of the cell state go to the output, h_t, produced at timestep t. Mathematically, the computations in an LSTM can be represented by the following equations:

$$\begin{aligned} f_t &= \sigma\left(W_f \times \left[h_{t-1}, x_t\right] + b_f\right) \\ \hat{c}_t &= \tanh\left(W_c \times \left[h_{t-1}, x_t\right] + b_c\right) \\ i_t &= \sigma\left(W_i \times \left[h_{t-1}, x_t\right] + b_i\right) \\ c_t &= f_t * c_{t-1} + i_t * \hat{c}_t \\ o_t &= \sigma\left(W_o \times \left[h_{t-1}, x_t\right] + b_o\right) \\ h_t &= o_t * \tanh(c_t). \end{aligned} \tag{1}$$

Here, W_f, W_c, W_i, and W_o are the matrices storing the weights of different gates; b_f, b_c, b_i, and b_o are the biases of the different gates; c_t is the current cell state and h_t is the current output. Matrix-vector multiplications are indicated by $\times$ and element wise multiplications are indicated by $*$.

LSTMs are most commonly used for sequence-to-sequence learning, which we discuss next.

C. Sequence-to-Sequence Learning

Sequence-to-sequence learning refers to the ability to learn a mapping from input sequences in one domain to output sequences in a different domain. Sequence-to-sequence models are deployed in a wide range of tasks including NMT [14], speech recognition [13], and video captioning [3]. As shown in Fig. 3, a sequence-to-sequence model comprises of two structures—an encoder and a decoder. The encoder converts the input sequence into a fixed dimensional context vector, which is then used by the decoder to generate the output sequence. For example, in a machine translation task, the encoder utilizes the words of the source sentence to produce a context vector that summarizes the semantics of the sentence. The decoder subsequently operates on this semantic vector to generate translated words in the target language. Advanced sequence-to-sequence models employ encoder and decoder networks with multiple layers and residual connections between layers to enable effective learning of more complicated sequences.

1) Evaluating Sequence-to-Sequence Models: Sequence-to-sequence models are evaluated by determining the quality of output sequences generated by a model. Unlike image recognition tasks, which usually have a unique golden output, most tasks performed by sequence-to-sequence models may allow multiple correct outputs for a given input. For example, there could be multiple correct translations for a given input sentence, which vary in the choice of words or phrases, as well as the ordering thereof. Human evaluations of these output sequences can be expensive. Multiple automatic evaluation methods have been proposed for sequence-to-sequence models that generate natural language as their output. A few examples are Bilingual Evaluation Understudy (BLEU) [35], METEOR [36], ROUGE [37], and CIDEr [38]. We utilize the most popular metric BLEU to evaluate the quality of our sequence-to-sequence models. BLEU scores indicate how similar the generated sentence is to the reference sentences, with higher values representing more similar sentences. A BLEU score of 100 indicates that the generated sentence is identical to one of the reference sentences.

2) Execution Time Breakdown for Sequence-to-Sequence Models: The overall execution time of a sequence-to-sequence model is a function of four major parameters as shown in equation (2)

$$InputSeqLen \times ComputeTimePerInputSymbol + OutputSeqLen \times ComputeTimePerOutputSymbol \quad (2)$$

where *InputSeqLen* and *OutputSeqLen* denote the lengths of the input and output sequences and *ComputeTimePerInputSymbol* and ComputeTimePer OutputSymbol denote the time required to process a single symbol of the input and output sequence. Overall, the first product term represents the encoding time while the second product term represents the decoding time. Three of the four parameters, namely, *InputSeqLen*, *ComputeTimePerInputSymbol*, and *ComputeTimePerOutputSymbol* are deterministic in nature and depend on the number of input symbols processed and the amount of computation in the encoder and decoder, respectively. The remaining parameter, *OutputSeqLen*, is nondeterministic and is influenced by the mapping performed by the sequence-to-sequence model. Reducing the execution time of sequence-to-sequence models amounts to reducing one or more of the four parameters mentioned above, without adversely affecting the overall quality of the model.

In summary, LSTMs and sequence-to-sequence models in particular have structures that are significantly different from feedforward neural networks. The exploration of techniques exploiting their specific characteristics is key to improving their execution efficiency.

IV. AxLSTM: Design Approach and Methodology

To improve the execution efficiency of LSTMs, we propose AxLSTM, a set of approximate computing techniques that exploit the key characteristics of LSTMs. In this section, we present the salient features of AxLSTM and describe its details in the context of sequence-to-sequence models.

A. AxLSTM: Overview

Fig. 4 outlines the approximation strategies adopted by AxLSTM and the targeted benefits. AxLSTM consists of two techniques—DTS and DSR. The motivation behind DTS is the fact that all symbols in an input sequence do not have an equal impact on the LSTM state. Consequently, the LSTM can skip evaluating symbols that are likely to have negligible effects on its state and thereby save execution time. For example, not all frames in a video sequence are equally important for an LSTM to generate an appropriate caption for the video. Similarly, some words of a sentence may turn out to be redundant in machine translation. Skipping the evaluation of any unimportant frame (or word) can help to save one entire timestep worth of computation, which involves multiple matrix multiplications for calculating the different gates of an LSTM.

In contrast, DSR is based on the principle that all input sequences are not semantically complex enough to require the entire dimensionality of the cell state. As a result, the state size can be dynamically modulated at runtime. A smaller state size reduces the sizes of matrices and thereby, the complexity of matrix multiplications involved in calculating the different gates of an LSTM. This, in turn, significantly decreases the time spent in evaluating each timestep of an LSTM.

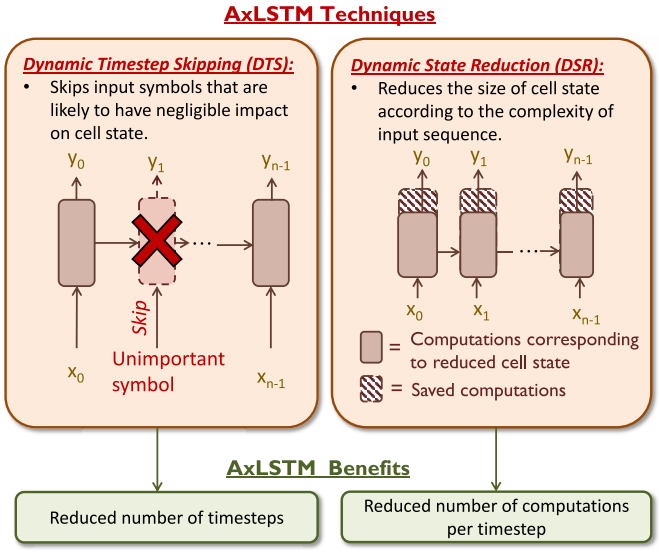

Fig. 4. Overview AxLSTM approximation strategies.

In summary, AxLSTM reduces the execution time of LSTMs by taking advantage of the two techniques, DTS and DSR, to reduce the number of timesteps as well as the number of computations per timestep.

B. Dynamic Timestep Skipping

The key challenge involved in DTS lies in developing an accurate, yet low overhead mechanism to identify unimportant symbols in an input sequence. To ensure savings, the time expended in evaluating this dynamic mechanism should be significantly less than that expended toward evaluating the LSTM itself.

We propose to augment an LSTM with a new structure, the *Input Analyzer* (IA). The IA acts as a filter to the LSTM and feeds only important input symbols of a sequence to it. It substitutes all unimportant symbols with a special symbol called the *Skip Symbol* (SS). Unlike other input symbols, the LSTM does not perform any computations on the SS. Instead, it simply moves on to processing the next timestep. The application of DTS to the encoder of a sequence-to-sequence model is shown in Fig. 5. In this case, DTS targets the encoding time and reduces the effective *InputSeqLen* observed by the encoder.

We propose two heuristics for the IA to dynamically identify unimportant input symbols. These heuristics, which are named *StateEffect* and *InputDiff*, are illustrated in Fig. 5. The StateEffect heuristic utilizes knowledge of each input symbol's expected or average effect on the encoder state. This effect is quantified as the L2 norm of the difference between the encoder state before and after processing a symbol. We rank the input symbols in increasing order of their effect on the encoder state for all examples in the training set. Next, we

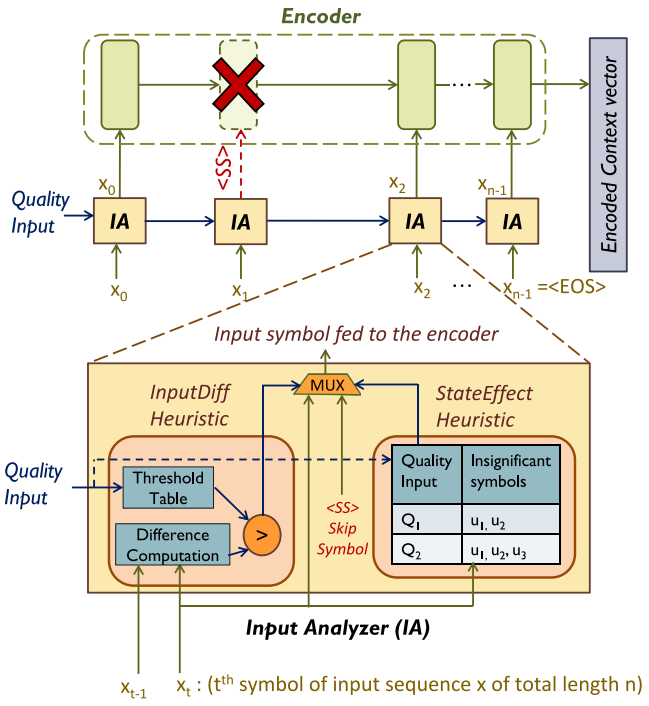

Fig. 5. DTS in sequence-to-sequence models.

skip each symbol and observe the effect on overall quality. Finally, we program the IA to store a list of input symbols that can be skipped for a given quality requirement. The size of the corresponding StateEffect table is determined by the number of available quality levels, the number of insignificant symbols per quality level and the encoding size for each symbol. In our evaluations, we observed that a table of size 12 kB is sufficient.

The InputDiff heuristic is based on the fact that an input symbol that is very similar to the previous input symbol does not convey any new information to the encoder. Accordingly, skipping its evaluation will minimally affect the overall output quality. An IA executing the InputDiff heuristic dynamically measures the difference between two successive input symbols. It is programmed to have a *Threshold Table* containing *DiffThres* values corresponding to each quality requirement. The IA decides to skip an input symbol if its difference from the previous symbol is less than DiffThres. In general, the suitability of the StateEffect or the InputDiff heuristic can vary across different applications and the IA can be programmed to execute either or both of them. The values in the threshold table are determined using the methodology described in Section IV-D.

In summary, by using the IA, the encoder of a sequence-to-sequence model is able to dynamically identify unimportant symbols of an input sequence and skip processing them, thereby reducing the execution time.

C. Dynamic State Reduction

In DSR, the original full dimensional state is converted to a smaller state by identifying and retaining only the significant state elements for a given input sequence. Correspondingly, the original weight matrices are sliced into smaller matrices by extracting only the values corresponding to the significant state elements.

In the context of sequence-to-sequence models, we apply DSR to the decoder and successfully reduce the contribution of *ComputeTimePerOutputSymbol* in (2). As shown in Fig. 6, we achieve this by adding a new structure called the State Minimizer (SM). In general, the SM can be invoked anywhere during the encoding or decoding process. However, invoking the SM at an intermediate timestep of the encoder and limiting the size of the encoder state size thereafter may lead to significant error since the subsequent input symbols have not yet been reflected in the encoder state. In addition, determining significant elements too frequently using the SM can turn out to be counter-productive due to the overhead of the significant index identification and matrix slicing steps. In order to avoid these challenges in our implementation, we invoke the SM only once at the well-defined boundary between the encoder and decoder network to identify a fixed set of significant indices before the decoding process starts. Specifically, the size of the hidden state in the encoder stays at the maximum value throughout the encoding process, producing a complete context vector. The SM analyzes the context vector and strips it down to a smaller vector containing only state elements that are deemed to be significant. The decoder weight matrices are also sliced by the SM based on the indices of the significant elements of the context vector. Each timestep of the decoder consumes and produces a context vector of this reduced size.

The SM examines the content of the encoded context vector to determine the significant elements. Two heuristics for identifying these significant elements are described below.

1) *Relative Thresholding:* The relative thresholding approach discards elements that are numerically far less than the maximum value in the context vector. For a given threshold relThres, an element i of the context vector C is identified to be insignificant if

$$C[i] < \text{relThres} \times \max_i C[i]. \tag{3}$$

2) *Absolute Thresholding:* The absolute thresholding approach discards elements whose magnitude is less than an absolute constant. For a given threshold of absThres, an element i of the context vector C is identified to be insignificant if

$$|C[i]| < \text{absThres}. \tag{4}$$

The relThres and absThres values in both criteria play an important role in achieving a favorable quality versus savings tradeoff. Lower values of relThres and absThres lead to smaller states as most of the context vector elements are discarded by the SM. However, this also leads to a potential loss of useful information and subsequently, a drop in quality. For a given user-defined quality constraint, the threshold values are determined using the methodology described in Section IV-D and are programmed into the SM before the AxLSTM model is deployed for inference.

For sequence-to-sequence models that perform inference after batching multiple input sequences, the SM adopts a conservative approach by forming a union of significant elements

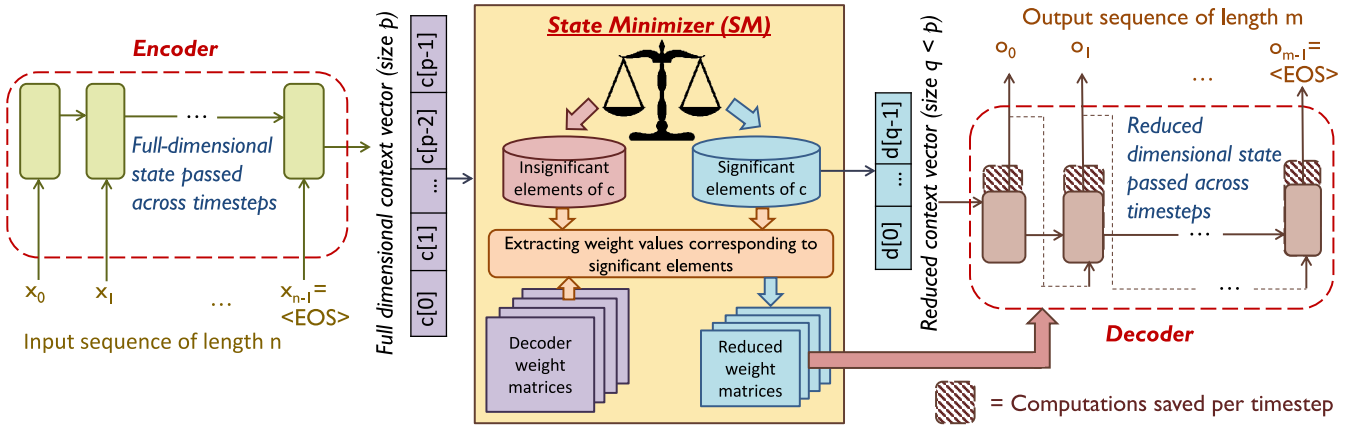

Fig. 6. DSR in Sequence-to-Sequence models.

across all input sequences in a batch. This leads to some amount of inefficiency as the final state size, determined by the number of significant elements, can eventually be higher than that required for a single sequence. However, batching of input sequences also provides an advantage by ensuring that the overhead of slicing states and weights is amortized over multiple sentences in the batch. A large number of state-of-the-art sequence-to-sequence models have multiple layers in the encoder and decoder. The SM is sensitive to the difference in the behavior of different layers and accordingly outputs different sets of important indices for different layers.

In summary, the SM equips an LSTM with the ability to reduce the state size dynamically at runtime based on the complexity of the input sequence. This leads to a reduction in the amount of computation per timestep.

D. AxLSTM: Design Methodology

Algorithm 1 presents a method to automatically design sequence-to-sequence models of specified quality levels with AxLSTM. For brevity of discussion, we restrict ourselves to the StateEffect heuristic of DTS and the absThres criterion for DSR. However, the method can be applied to other heuristics and critieria with minor modifications. The inputs to the method are a sequence-to-sequence model trained with conventional LSTMs (*Seq2Seq*), the training dataset (*TrainData*), the list of input symbols in the training dataset (*InputSym*) and the quality constraint (Q). The output is the approximate version of the *Seq2Seq* model (*AxSeq2Seq*), the list of unimportant input sequence symbols for DTS (DTS_{list}) and the DSR threshold for identifying unimportant elements of the context vector C for layer *Li* (DSR^{Li}_{thres}). The algorithm builds *AxSeq2Seq* by successively adding more input symbols to DTS_{list}, reducing DSR^{Li}_{thres} and retraining *AxSeq2Seq* until the quality drops below the specified threshold.

We first measure the average effect on encoder state for all input symbols in the training dataset (lines 1 and 2) by calculating the L2 difference between the state vector before and after processing the symbol. Next, we sort the input symbols in order of their increasing effect on state (line 3) and store them in *SortedSym*. The DSR_{thres} for each layer is initialized to the maximum absolute value of the corresponding context vector (lines 4 and 6). The DTS_{list} is initialized to be a null list (line 7) and the weight and bias values of *AxSeqtoSeq* are initialized to be equal to that of the original *Seq2Seq*. At each iteration of the algorithm, new symbols are copied from the *SortedSym* list to DTS_{list} (line 11). The DSR_{thres} values for each layer of the network are reduced by a small constant Δ (line 13). Finally, the network is retrained for T training iterations to further improve its quality level (line 14). This process

Algorithm 1 Designing Sequence-to-Sequence Models With AxLSTM

Input: *Seq2Seq*: Trained sequence-to-sequence model,
TrainData: Training dataset,
InputSym: List of input sequence symbols,
Q: Quality constraint

Output: *AxSeq2Seq*: Approximate sequence-to-sequence model,
DTS_{list}: List of unimportant input sequence symbols,
DSR^{Li}_{thres}: Threshold for identifying unimportant elements of context vector C for layer *Li*

1: **For** each symbol $S \in$ *InputSym*:
2: Compute average effect on encoder state, *Effect*[*S*]
3: Sort *InputSym* in increasing order of *Effect*, *SortedSym*
4: **For** each Layer *Li*:
5: Compute Max. Abs(Context vector C), C^{Li}_{max}
6: Initialize $DSR^{Li}_{thres} = C^{Li}_{max}$
7: $DTS_{list} = \emptyset$
8: *AxSeq2Seq* = *Seq2Seq*
9: *numSkipped* = 0
10: **while** (1) **do**
11: $DTS_{list} = DTS_{list} \cup SortedSym[numSkipped]$
12: **For** each Layer *Li*:
13: Set DSR^{Li}_{thres} -= Δ
14: *AxSeq2Seq* = Retrain(*AxSeq2Seq*, DTS_{list}, DSR_{thres}, *TrainData*, T training iterations)
15: **if**($Q_{AxSeq2Seq} < Q$) **break**
16: *numSkipped* = *numSkipped*+1
17: **end while**
18: **return** *AxSeq2Seq*, DTS_{list} **and** DSR^{Li}_{thres}

Model	NMT1	NMT2	S2VT
Task	Neural Machine Translation	Neural Machine Translation	Video Captioning
Dataset	IWSLT15	WMT16	Youtube Corpus (MSVD)
Input Sequence	English sentence	German sentence	Video
Output Sequence	Vietnamese sentence	English sentence	English sentence
No. of layers	2	4	2
LSTM state size	512	1024	256

Fig. 7. Application benchmarks.

is stopped when the quality level $Q_{AxSeq2Seq}$ of the *AxSeq2Seq* drops below the specified quality level Q.

In summary, by selectively skipping unimportant symbols in the encoder and reducing the decoder state size, AxLSTM improves the execution efficiency of LSTMs.

V. Experimental Methodology

In this section, we present the methodology utilized in our experiments to evaluate AxLSTM.

A. Performance Evaluation

We implemented AxLSTM within TensorFlow [39], a popular deep learning framework. The files used by the Python API of TensorFlow were modified to incorporate the DTS and DSR techniques. We evaluated the performance benefits of AxLSTM in software by measuring application level runtimes with and without the proposed techniques on a 2.7 GHz Intel Xeon server with 128 GB memory and 32 processor cores.

B. Application Benchmarks

Our benchmark suite, listed in Fig. 7, consists of three sequence-to-sequence models with applications in NMT and video captioning. These models vary considerably in the nature of their input sequences, output sequences and their computational complexity, as indicated by the number of layers and LSTM state size. The two NMT models are based on the models available in [40]. The first NMT model performs English–Vietnamese translation and is trained on a parallel corpus of TED talks (133K sentence pairs) provided by the IWSLT Evaluation Campaign. The next NMT model performs German–English translation and is trained on the German–English parallel corpus (4.5M sentence pairs) provided by the WMT evaluation campaign. On the other hand, the video captioning model is based on [3], trained on the Microsoft Video Description Corpus [41] and used to generate captions on Youtube clips collected on Amazon Mechanical Turk. We utilize BLEU scores, a method for automatically evaluating machine translations, as a metric to evaluate application quality. The baseline implementations were trained on the training datasets using conventional LSTM networks, whereas the AxLSTM implementations were obtained using the methodology mentioned in Section IV-D with the DTS and DSR techniques in place. The application level runtimes and BLEU scores of both implementations were measured on a separate validation dataset.

VI. Results

In this section, we present the results of our experiments that evaluate the benefits of AxLSTM, and analyze the sources of these benefits.

A. Performance Benefits Versus Accuracy

Fig. 8 shows the normalized execution time benefits achieved by AxLSTM across all benchmarks. For each benchmark, models with different quality levels, or equivalently, different BLEU scores, are obtained by varying how aggressively input symbols are skipped in DTS and how aggressively the state size is reduced using DSR. AxLSTM reduces the execution time by 13%, 19.1%, and 23.5% for average drops in BLEU scores of 0.9, 2.5, and 5.8, respectively. This models were obtained by applying quality constraints of <1, <3, and <6 drops in BLEU scores, respectively, in the methodology described in Section IV-D. For each of these cases, we also calculated the theoretical reduction in compute operations by taking into account the number of skipped symbols and the reduction in state size. The corresponding values are shown in Fig. 9. We observe that AxLSTM achieves 22.2%, 29.8%, and 36.29% reduction in operations for average drops in BLEU scores of 0.9, 2.5, and 5.8, respectively. The reduction in compute operations do not translate entirely into the execution time benefits because of the overheads associated with DTS and DSR, which are discussed in more detail in Section VI-B.

Figs. 8 and 9 also show the individual execution time versus quality tradeoffs observed with DTS and DSR. The speedups and compute reductions observed with DTS depend on the amount of redundancy present in the input sequence. In general, frames in a video have more redundancy than words in a sentence. Specifically, consecutive frames in a video have a higher chance of conveying similar redundant information than consecutive words in a sentence. As a result, DTS can demonstrate higher benefits on sequence-to-sequence models with videos as input sequences. In Fig. 8, DTS achieves 10.5% reduction in execution time on the S2VT model as opposed to 4.7% and 2.9% reductions on NMT1 and NMT2, respectively, for similar drops in BLEU scores. This corresponds to 17.8%, 6.4%, and 4.1% reductions in compute operations in the S2VT, NMT1, and NMT2, respectively, as shown in Fig. 9.

In contrast, the benefits observed with DSR depend primarily on the fraction of semantically complex input sequences in a dataset. A higher proportion of semantically simpler inputs allows S2VT to derive higher DSR benefits than the other two models. As shown in Fig. 8, S2VT experiences 20.4% reduction in execution time with DSR whereas NMT1 and NMT2 experience reductions of 18.1% and 7.6%, respectively, for similar drops in BLEU scores. The reduction in compute operations follows a similar trend as illustrated in Fig. 9. DSR achieves 30.9% reduction in compute operations

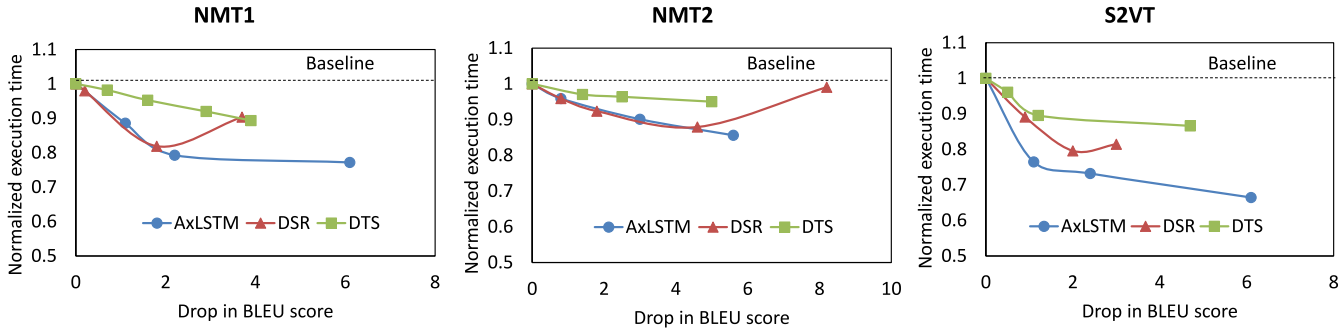

Fig. 8. Normalized execution time versus drop in quality using AxLSTM for sequence-to-sequence models.

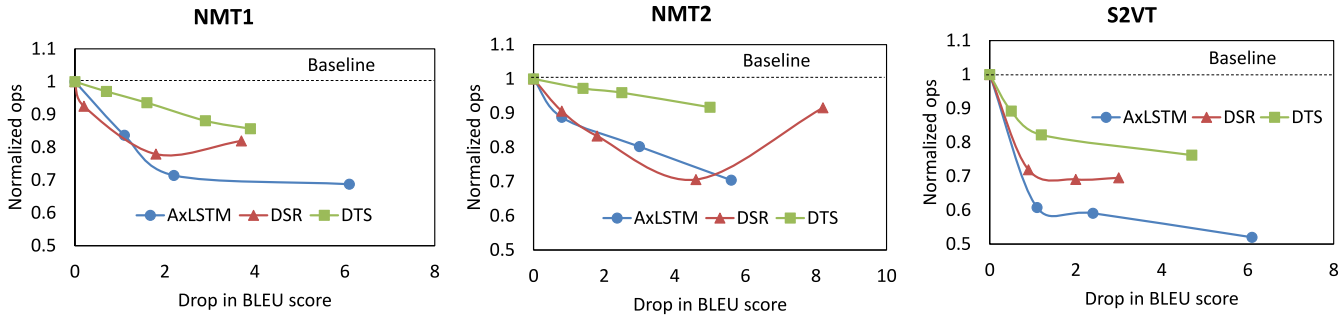

Fig. 9. Normalized compute operations versus drop in quality using AxLSTM for sequence-to-sequence models.

on S2VT as opposed to 22.1% and 16.7% reductions on NMT1 and NMT2 models, respectively.

Figs. 8 and 9 present an interesting behavior exhibited by models with DSR-based approximations. We observe that the execution time and compute operations initially drop with a reduction in BLEU score but subsequently increase with further reduction in BLEU score. This counterintuitive behavior can be attributed to the fact that these inferior quality models suffer from the problem of lack of coverage, in which the smaller state vector leads to loss of useful information. This results in over-translation, where some words are unnecessarily translated multiple times [42]. This causes *OutputSeqLen* in (2) to increase, outweighing the reduction in *ComputeTimePerOutputSymbol*, resulting in a net increase in the compute operations and decoding time.

B. Benefits Breakdown and Overhead Analysis

In this section, we analyze the benefits observed with AxLSTM by providing a breakdown in terms of the individual execution time benefits observed during the encoding and decoding process. Fig. 10 shows this breakdown across the three benchmarks for an average of 0.9 drop in BLEU score. On average, AxLSTM reduces the encoding and decoding time by 9.8% and 14.6%, respectively, which translates to 13.9% reduction in overall execution time. It is important to note here that the decoding process accounts for a larger fraction of the overall execution time in both the baseline and the AxLSTM-based implementations of all three models.

The figure also highlights the overheads encountered with AxLSTM. We observe an average of 6.4% and 2.08% overheads during the encoding and decoding process, respectively,

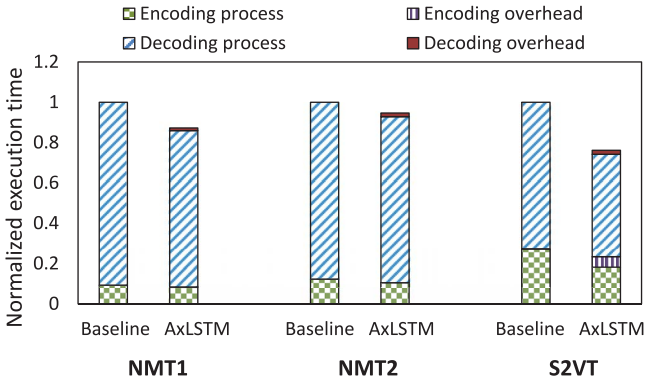

Fig. 10. Execution time benefits breakdown with AxLSTM.

which in turn leads to an overall overhead of 3.5%. The encoding and decoding overheads are directly attributable to DTS and DSR, respectively. The encoding time overhead is equal to the time taken by the IA in DTS to determine the unimportant symbols on the fly. We observe that this overhead is a strong function of the heuristic adopted by IA. In general, the time consumed by the StateEffect heuristic is substantially less than that consumed by the InputDiff heuristic. Consequently, the encoding overhead for NMT1 and NMT2, which utilize the StateEffect heuristic, is <0.1% as opposed to an encoding overhead of 19.2% in the S2VT model, which utilizes the InputDiff heuristic. This difference stems from the fact that the computations involved in calculating the sum of absolute difference between pixels of consecutive video frames in S2VT are significantly more expensive than the quality table lookup performed in the other two models. On the other hand, the decoding overhead is proportional to the time expended by

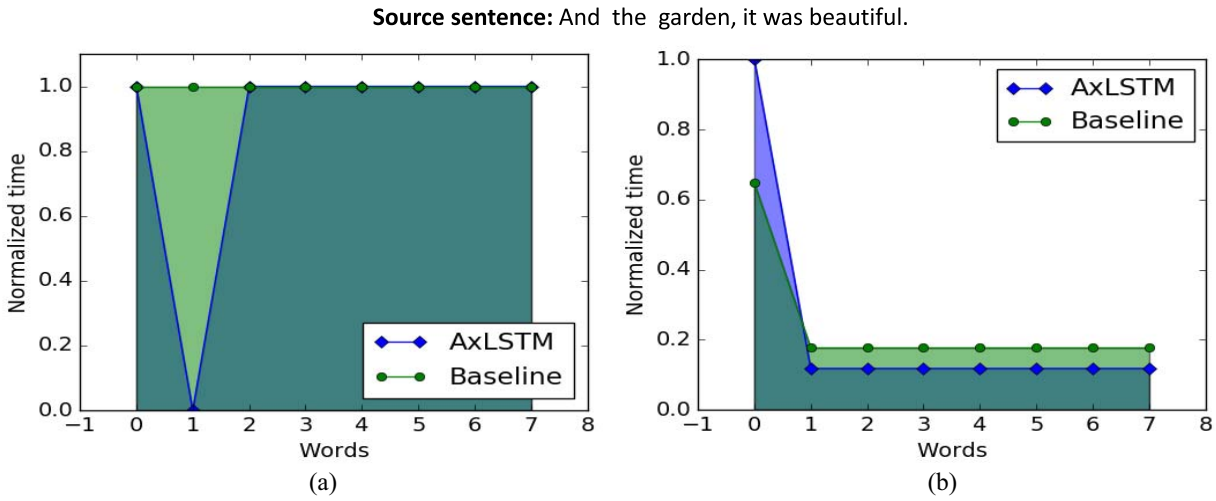

Fig. 11. (a) Normalized encoding time per input word and (b) Normalized decoding time per output word for a semantically simple sentence with and without AxLSTM.

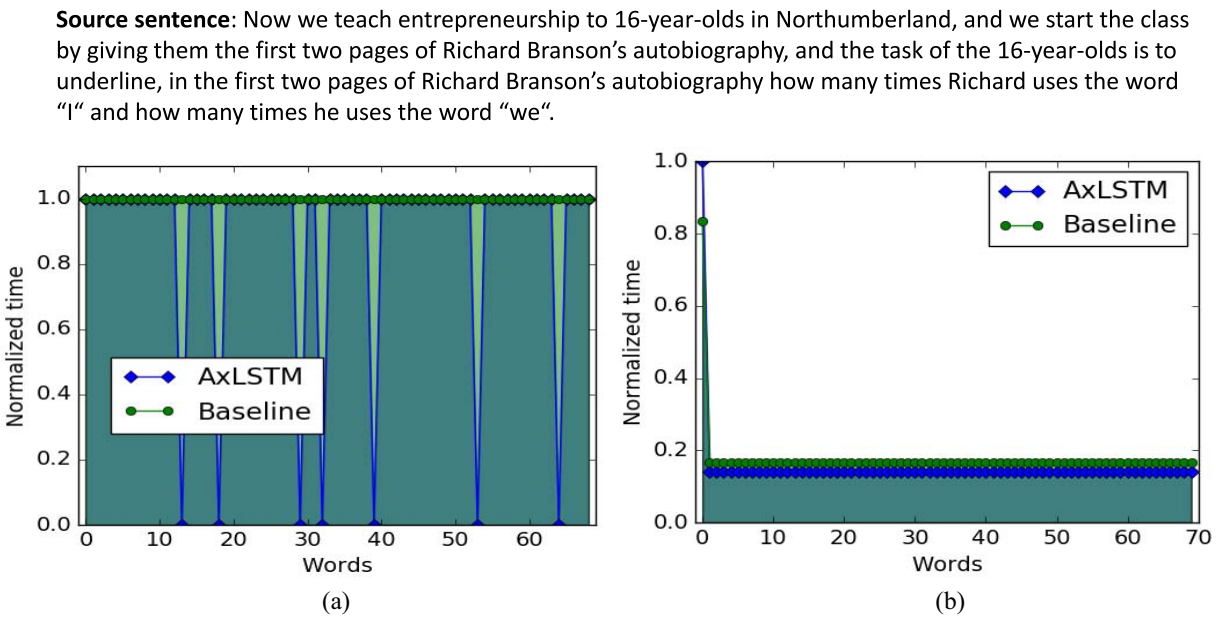

Fig. 12. (a) Normalized encoding time per input word and (b) Normalized decoding time per output word for a semantically complex sentence with and without AxLSTM.

the SM to extract important elements and slice matrices as part of the DSR process. The observed values vary according to the number of important elements and the matrix sizes involved.

In summary, the individual benefits and overheads of DTS and DSR combine to determine the overall speedups observed with AxLSTM.

C. Input Adaptive Approximations in Action

As described in Section IV, the two techniques of AxLSTM, DTS, and DSR, reduce two different components of (2), *viz.*, *InputSeqLen* and *ComputeTimePerOutputSymbol*. We illustrate these reductions in the NMT1 model with the help of two different sentences from the IWSLT dataset. For both sentences AxLSTM produces translations that are identical to the baseline LSTM, i.e., there is no drop in output quality.

Fig. 11(a) shows the *ComputeTimePerInputSymbol* with and without AxLSTM for each input symbol in a semantically simple source sentence comprising of eight words, or equivalently, eight input symbols. The specified quality constraint allows DTS to skip all instances of "the" present in the sentence, leading to zero processing time for the second word in the case of AxLSTM. The *ComputeTimePerInputSymbol* for the remaining words is not affected by AxLSTM because the quality constraint does not allow DTS to skip them. Overall, the area under the curves denote the total encoding time and we observe that AxLSTM successfully reduces the encoding time from the baseline implementation.

Fig. 11(b) shows the *ComputeTimePerOutputSymbol* with and without AxLSTM for each symbol during the decoding process of the same sentence. For all output symbols other than the first symbol, the *ComputeTimePerOutputSymbol* observed with AxLSTM is less than that observed with LSTM because of the reduced state size with DSR. The increase in time for the first symbol with AxLSTM can be attributed to the

overhead associated with the matrix slicing process in DSR. However, the total area under the AxLSTM curve is less than that under the baseline curve, indicating that AxLSTM reduces the decoding time for this sentence.

The *ComputeTimePerInputSymbol* and *ComputeTimePerOutputSymbol* for a semantically complex source sentence with 69 words are shown in Fig. 12(a) and (b). We observe that DTS is able to extract higher benefits in this case because of a more frequent occurrence of unimportant words. Specifically, the compute time reduces to zero for seven words. However, the complex nature of this sentence dictates the use of a larger decoder state in DSR and accordingly, we observe a lower reduction in time during the decoding process. Nevertheless, the area under the AxLSTM curves during both the encoding and decoding process are still less than the baseline, indicating that AxLSTM is beneficial.

In summary, AxLSTM is able to successfully extract execution time benefits in the form of reduced encoding and decoding times for input sequences of widely varying complexity.

VII. Conclusion

LSTM networks, a special class of RNNs, have attracted widespread attention due to their success in a range of machine learning applications involving sequences, including text generation, machine translation, and speech recognition. We address the computational challenges posed by LSTMs by proposing AxLSTM, an application of approximate computing to improve the execution efficiency of LSTMs. AxLSTM comprises of two techniques—DTS and DSR—that exploit the unique structure and computational characteristics of LSTMs. DTS reduces the effective number of timesteps of an LSTM by dynamically identifying input symbols that have little or no impact on the LSTM state and skipping them. DSR reduces the computations in each timestep of an LSTM by dynamically reducing the size of the LSTM state. Both these techniques are intrinsically input-adaptive, i.e., they modulate the overall computational effort of an LSTM based on the complexity of the input sequences. We describe how AxLSTM can be applied in the context of sequence-to-sequence models. We implement AxLSTM within the TensorFlow deep learning framework and evaluate its benefits on three state-of-the-art sequence-to-sequence benchmarks. Our evaluations on an Intel Xeon Server reveal that AxLSTM achieves speedups of $1.08\times - 1.31\times$ with minimal loss in quality, and $1.12\times - 1.37\times$ when moderate reductions in quality are acceptable.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, Dec. 2015.

[2] C. Szegedy *et al.*, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, Sep. 2014.

[3] S. Venugopalan *et al.*, "Sequence to sequence—Video to text," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 4534–4542.

[4] A. Y. Hannun *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *CoRR*, vol. abs/1412.5567, Dec. 2014.

[5] X. Zhang and Y. LeCun, "Text understanding from scratch," *CoRR*, vol. abs/1502.01710, Apr. 2015.

[6] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[7] J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan, "A persona-based neural conversation model," *CoRR*, vol. abs/1603.06155, 2016. [Online]. Available: http://arxiv.org/abs/1603.06155

[8] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "DeepFace: Closing the gap to human-level performance in face verification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 1701–1708.

[9] (2016). *Introducing DeepText: Facebook's Text Understanding Engine: Facebook Code*. [Online]. Available: https://code.facebook.com/posts/181565595577955/introducing-deeptext-facebook-s-text-understanding-engine/

[10] *Improving Photo Search: A Step Across the Semantic Gap*, Google Res. Blog, Mountain View, CA, USA, 2013. [Online]. Available: https://ai.googleblog.com/2013/06/improving-photo-search-step-across.html

[11] *Apple Is Turning Siri Into a Next-Level Artificial Intelligence*, Mashable, New York, NY, USA, 2016. [Online]. Available: http://mashable.com/2016/06/13/siri-sirikit-wwdc2016-analysis/hLMSxZKVnEqO

[12] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, Jun. 2013.

[13] A. Graves, A.-R. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, May 2013, pp. 6645–6649.

[14] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 2. Cambridge, MA, USA, 2014, pp. 3104–3112.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[16] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, Oct. 2016.

[17] B. Li *et al.*, "Large scale recurrent neural network on GPU," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2014, pp. 4062–4069.

[18] J. Devlin, "Sharp models on dull hardware: Fast and accurate neural machine translation decoding on the CPU," *CoRR*, vol. abs/1705.01991, May 2017.

[19] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 1–12.

[20] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *CoRR*, vol. abs/1511.05552, Mar. 2015.

[21] M. Lee *et al.*, "FPGA-based low-power speech recognition with recurrent neural networks," in *Proc. IEEE Int. Workshop Signal Process. Syst. (SiPS)*, Oct. 2016, pp. 230–235.

[22] S. Li *et al.*, "FPGA acceleration of recurrent neural network based language model," in *Proc. IEEE 23rd Annu. Int. Symp. Field Program. Custom Comput. Mach.*, Vancouver, BC, Canada, May 2015, pp. 111–118.

[23] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Proc. 22nd Asia South Pac. Design Autom. Conf. (ASP-DAC)*, Jan. 2017, pp. 629–634.

[24] A. See, M.-T. Luong, and C. D. Manning, "Compression of neural machine translation models via pruning," in *Proc. CoNLL*, 2016, pp. 291–301.

[25] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2017, pp. 75–84.

[26] S. Shin, K. Hwang, and W. Sung, "Fixed-point performance analysis of recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Mar. 2016, pp. 976–980.

[27] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, 2014, pp. 27–32.

[28] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: An approximate computing framework for artificial neural network," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2015, pp. 701–706.

[29] Z. Du *et al.*, "Leveraging the error resilience of neural networks for designing highly energy efficient accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 8, pp. 1223–1235, Aug. 2015.

[30] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, 2016, pp. 168–173.
[31] S. Sen, S. Venkataramani, and A. Raghunathan, "Approximate computing for spiking neural networks," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2017, pp. 193–198.
[32] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.
[33] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017.
[34] J. Chung, C. Gülcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, Dec. 2014.
[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguist. (ACL)*, 2002, pp. 311–318.
[36] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proc. 9th Workshop Stat. Mach. Transl.*, 2014, pp. 376–380.
[37] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Text Summarization Branches Out Workshop (ACL)*, Jul. 2004, pp. 74–81.
[38] R. Vedantam, C. L. Zitnick, and D. Parikh, "CIDEr: Consensus-based image description evaluation," *CoRR*, vol. abs/1411.5726, Jun. 2014.
[39] M. Abadi. *et al.* (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: https://tensorflow.org
[40] M.-T. Luong, E. Brevdo, and R. Zhao. (2017). *Neural Machine Translation (Seq2Seq) Tutorial*. [Online]. Available: https://github.com/tensorflow/nmt
[41] D. L. Chen and W. B. Dolan, "Collecting highly parallel data for paraphrase evaluation," in *Proc. Assoc. Comput. Linguist.*, Jan. 2011, pp. 190–200.
[42] Z. Tu, Z. Lu, Y. Liu, X. Liu, and H. Li, "Coverage-based neural machine translation," *CoRR*, vol. abs/1601.04811, Aug. 2016.

Sanchari Sen received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology Kharagpur, Kharagpur, India. She is currently pursuing the Ph.D. degree with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA.

Her current research interests include approximate computing, energy efficient architectures for deep learning, and software and hardware techniques for accelerating deep learning on different platforms.

Ms. Sen was a recipient of the Institute Silver Medal for her academic performance in IIT Kharagpur and the Prestigious Ross Fellowship from Purdue University in 2015.

Anand Raghunathan (F'12) received the B.Tech. degree in electrical and electronics engineering from the Indian Institute of Technology Madras, Chennai, India, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, USA.

He is currently a Professor and the Chair of the VLSI area with the School of Electrical and Computer Engineering, Purdue University, where he directs research in the Integrated Systems Laboratory. He holds the Distinguished Visiting Chair in Computational Brain Research with the Indian Institute of Technology Madras. He was a Senior Researcher and a Project Leader with NEC Laboratories America, Princeton, NJ, USA and held a visiting position with Princeton University. He is the Co-Founder and the Director of hardware with High Performance Imaging, Inc., West Lafayette, IN, USA, a company commercializing Purdue innovations in the area of computational imaging. He has co-authored a book, eight book chapters, and over 250 refereed journal and conference papers, and holds 24 U.S. patents and 16 international patents. His current research interests include brain-inspired cognitive computing, energy-efficient machine learning and artificial intelligence, system-on-chip design and computing with post-CMOS devices.

Prof. Raghunathan was a recipient of eight best paper awards and five best paper nominations at premier IEEE and ACM conferences for his publications, the Patent of the Year Award, and two Technology Commercialization Award from NEC, the Distinguished Alumnus Award from IIT Madras, the IEEE Meritorious Service Award and Outstanding Service Award. He was chosen by MIT's Technology Review among the TR35 (top 35 innovators under 35 years, across various disciplines of science and technology) in 2006. He has chaired four premier IEEE/ACM conferences, and served on the editorial boards of various IEEE and ACM journals in his areas of interest. He is a Golden Core Member of the IEEE Computer Society.