LEARNING RECURRENT BINARY/TERNARY WEIGHTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recurrent neural networks (RNNs) have shown excellent performance in processing sequence data. However, they are both complex and memory intensive due to their recursive nature. These limitations make RNNs difficult to embed on mobile devices requiring real-time processes with limited hardware resources. To address the above issues, we introduce a method that can learn binary and ternary weights during the training phase to facilitate hardware implementations of RNNs. As a result, using this approach replaces all multiply-accumulate operations by simple accumulations, bringing significant benefits to custom hardware in terms of silicon area and power consumption. On the software side, we evaluate the performance (in terms of accuracy) of our method using long short-term memories (LSTMs) on various sequential models including sequence classification and language modeling. We demonstrate that our method achieves competitive results on the aforementioned tasks while using binary/ternary weights during the runtime. On the hardware side, we present custom hardware for accelerating the recurrent computations of LSTMs with binary/ternary weights. Ultimately, we show that LSTMs with binary/ternary weights can achieve up to $12 \times$ memory saving and $10 \times$ inference speedup compared to the full-precision implementation on an ASIC platform.

1 INTRODUCTION

Convolutional neural networks (CNNs) have surpassed human-level accuracy in various complex tasks by obtaining a hierarchical representation with increasing levels of abstraction (Bengio 2009; Lecun et al. 2015). As a result, they have been adopted in many applications for learning hierarchical representation of spatial data. CNNs are constructed by stacking multiple convolutional layers often followed by fully-connected layers (Lecun et al. 1998). While the vast majority of network parameters (i.e. weights) are usually found in fully-connected layers, the computational complexity of CNNs is dominated by the multiply-accumulate operations required by convolutional layers (Yang et al. 2015). Recurrent neural networks, on the other hand, have shown remarkable success in modeling temporal data (Mikolov et al. 2010; Graves 2013; Cho et al. 2014a; Sutskever et al. 2014; Vinyals et al. 2014). Similar to CNNs, RNNs are over parameterized since they build on high-dimensional inputs/outputs and suffer from high computational complexity due to their recursive nature (Xu et al. 2018). As a result, the aforementioned limitations make the deployment of CNNs and RNNs difficult on mobile devices that require real-time processes with limited hardware resources.

Several techniques have been introduced in literature to address the above issues. In (Sainath et al. 2013; Jaderberg et al. 2014; Lebedev et al. 2014; Tai et al. 2015), it was shown that the weight matrix can be approximated using a lower rank matrix. In (Liu et al. 2015; Han et al. 2015; Wen et al. 2016; Arakani et al. 2016), it was shown that a significant number of parameters in DNNs are noncontributory and can be pruned without any performance degradation in the final accuracy performance. Finally, quantization approaches were introduced in (Courbariaux et al. 2015; Lin et al. 2015; Courbariaux & Bengio 2016; Kim & Smaragdis 2016; Hubara et al. 2016b; Rastegari et al. 2016; Hubara et al. 2016a; Zhou et al. 2016; Li & Liu 2016; Zhu et al. 2016) to reduce the required bitwidth of weights/activations. In this way, power-hungry multiply-accumulate operations are replaced by simple accumulations while also reducing the number of memory accesses to the off-chip memory.

Considering the improvement factor of each of the above approaches in terms of energy and power reductions, quantization has proven to be the most beneficial for hardware implementations. However,
all of the aforementioned quantization approaches focused on optimizing CNNs or fully-connected networks only. As a result, despite the remarkable success of RNNs in processing sequential data, RNNs have received the least attention for hardware implementations, when compared to CNNs and fully-connected networks. In fact, the recursive nature of RNNs makes their quantization difficult. In (Hou et al. (2016)), for example, it was shown that the well-known BinaryConnect technique fails to binarize the parameters of RNNs due to the exploding gradient problem (Courbariaux et al. (2015)). As a result, a binarized RNN was introduced in (Hou et al. (2016)), with promising results on simple tasks and datasets. However it does not generalize well on tasks requiring large inputs/outputs (Xu et al. (2018)). In (Xu et al. (2018); Hubara et al. (2016b)), multi-bit quantized RNNs were introduced. These works managed to match their accuracy performance with their full-precision counterparts while using up to 4 bits for data representations.

In this paper, we propose a method that eliminates the need for multiplications through learning recurrent binary and ternary weights in RNNs. In this way, all weights are constrained to \{+1, -1\} or \{+1, 0, -1\} in binary or ternary representations, respectively. Using the proposed approach, RNNs with binary and ternary weights can achieve the performance accuracy of their full-precision counterparts. In summary, this paper makes the following contributions:

- We introduce a method for learning recurrent binary and ternary weights during both forward and backward propagation phases.
- We perform a set of experiments on various sequential tasks, such as sequence classification and language modeling. We then demonstrate that our method can achieve near state-of-the-art results with greatly reduced computational complexity.
- We present custom hardware to accelerate the recurrent computations of RNNs with binary or ternary weights. The proposed dedicated accelerator can save up to 12× of memory elements/bandwidth and speed up the recurrent computations by up to 10×.

## 2 Related Work

During the binarization process, each element of the full-precision weight matrix \( W \in \mathbb{R}^{d_I \times d_J} \) is binarized by \( w_{i,j} = A_{i,j} w_{i,j}^B \), where \( A_{i,j} > 0 \), \( i \in \{1, \ldots, d_I\} \), \( j \in \{1, \ldots, d_J\} \) and \( w_{i,j}^B \in \{-1, +1\} \). In BinaryConnect (Courbariaux et al. (2015)), the binarized weight element \( w_{i,j}^B \) is obtained by the sign function while using a fixed scaling factor \( A \) for all the elements: \( w_{i,j}^B = A \times \text{sign}(w_{i,j}) \). In TernaryConnect (Lin et al. (2015)), values hesitating to be either +1 or -1 are clamped to zero to reduce the accuracy loss of binarization: \( w_{i,j} = A_{i,j} w_{i,j}^T \), where \( w_{i,j}^T \in \{-1, 0, +1\} \). To further improve the precision accuracy, TernaryConnect stochastically assigns ternary values to the weight elements by performing \( w_{i,j} = A \times \text{Bernoulli}(|w_{i,j}|) \times \text{sign}(w_{i,j}) \) while using a fixed scaling factor \( A \) for each layer. Ternary weight networks (TWNs) were then proposed to learn the factor \( A \) by minimizing the L2 distance between the full-precision and ternary weights for each layer. Zhou et al. (2016) introduced DoReFa-Net as a method that can learn different bitwidths for weights, activations, and gradients. Since the quantization functions used in the above works are not differentiable, the derivative of the loss \( L \) w.r.t the full-precision \( W \) is approximated by

\[
\frac{\partial L}{\partial W} = \frac{\partial L}{\partial W^B} = \frac{\partial L}{\partial W^T},
\]

where \( W^B \) and \( W^T \) denote binarized and ternarized weights, respectively.

Zhu et al. (2016) introduced the trained ternary quantization (TTQ) method that uses two asymmetric scaling parameters \( A \) for positive values and \( \beta \) for negative values to ternarize the weights. In loss-aware binarization (LAB) (Hou et al. (2016)), the loss of binarization was explicitly considered. More precisely, the loss w.r.t the binarized weights is minimized using the proximal Newton algorithm. Hou & Kwok (2018) extended LAB to support different bitwidths for the weights. This method is called loss-aware quantization (LAQ). Recently, Guo et al. (2017) introduced a new method that builds the full-precision weight matrix \( W \) as \( k \) multiple binary weight matrices: \( w_{i,j} = \sum_{z=1}^{k} A_{i,j}^z B_{i,j}^z \) where \( B_{i,j}^z \in \{-1, +1\} \) and \( A_{i,j}^z > 0 \). Xu et al. (2018) also uses a binary search tree to efficiently derive the

\[\text{sign}(w_{i,j})\]
binary codes $D^k_i,j$, improving the prediction accuracy. While using multiple binary weight matrices reduces the bitwidth by a factor of $32 \times$ compared to its full-precision counterpart, it increases not only the number of parameters but also the number of operations by a factor of $k$ (Xu et al. (2018)).

Among the aforementioned methods, only works of Xu et al. (2018) and Hou & Kwok (2018) targeted RNNs to reduce their computational complexity and outperformed all the aforementioned methods in terms of the prediction accuracy. However, they have shown promising results only on specific temporal tasks: the former targeted only the character-level language modeling task on small datasets while the latter performs the word-level language modeling task and matches the performance of the full-precision model when using $k = 4$. Therefore, there is no binary model that can match the performance of the full-precision model on the word-level language modeling task. More generally, a binary/ternary model that can perform all temporal tasks and achieve similar prediction accuracy to its full-precision counterpart is missing in literature.

3 Preliminaries

3.1 LSTM

Despite the remarkable success of RNNs in processing variable-length sequences, they suffer from the exploding gradient problem that occurs when learning long-term dependencies (Bengio et al. (1994); Pascanu et al. (2013)). Therefore, various RNN architectures such as Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber (1997)) and Gated Recurrent Units (GRUs) (Cho et al. (2014b)) were introduced in literature to mitigate the exploding gradient problem. In this paper, we focus on the LSTM architecture to learn recurrent binary/ternary weights due to their prevalent use in both academia and industry. The recurrent transition of LSTM is obtained by:

$$
\begin{pmatrix}
    f_t \\
    i_t \\
    o_t \\
    g_t
\end{pmatrix} =
\begin{pmatrix}
    \sigma \\
    \sigma \\
    \sigma \\
    \tanh
\end{pmatrix}
W_h h_{t-1} + W_x x_t + b,
$$

where $W_h \in \mathbb{R}^{d_h \times 4d_h}$, $W_x \in \mathbb{R}^{d_x \times 4d_h}$ and $b \in \mathbb{R}^{4d_h}$ denote the recurrent weights and bias. The parameters $h \in \mathbb{R}^{d_h}$ and $c \in \mathbb{R}^{d_h}$ are hidden states. The logistic sigmoid function and Hadamard product are denoted as $\sigma$ and $\odot$, respectively. The updates of the LSTM parameters are regulated through a set of gates: $f_t$, $i_t$, $o_t$ and $g_t$. Eq. (2) shows that the main computational core of LSTM is dominated by the matrix multiplications (i.e., $W_h h_{t-1} + W_x x_t$). The recurrent weight matrices $W_h$ and $W_x$ also contain the majority of the model parameters. As such, we aim to compensate the computational complexity of the LSTM cell and reduce the number of memory accesses to the energy/power-hungry DRAM by binarizing or ternarizing the recurrent weights.

3.2 Batch Normalization

It is well-known that any change in the distribution of the inputs to a model affects the training efficiency (Ioffe & Szegedy (2015)). This phenomenon is referred to as covariate shift. A common approach to reducing the covariate shift is to use batch normalization (Ioffe & Szegedy (2015)). The batch normalization transform can be formulated as follows:

$$
\text{BN}(x; \alpha, \beta) = \beta + \alpha \odot \frac{x - \mathbb{E}(x)}{\sqrt{\mathbb{V}(x)} + \epsilon},
$$

where $x$ and $\epsilon$ denote the unnormalized vector and a regularization hyperparameter. The mean and standard deviation of the normalized vector are determined by model parameters $\alpha$ and $\beta$. The statistics $\mathbb{E}(x)$ and $\mathbb{V}(x)$ also denote the estimations of the mean and variance of the unnormalized vector for the current minibatch, respectively. Batch normalization is commonly applied to a layer where changing its parameters affects the distributions of the inputs to the next layer. This occurs frequently in RNN where its input at time $t$ depends on its output at time $t - 1$. Several works have investigated batch normalization in RNNs (Cooijmans et al. (2016); Laurent et al. (2016); Amodei et al. (2016)) to improve their convergence speed and performance.
4 Learning Recurrent Binary/Ternary Weights

[Hou et al. (2016)] showed that the methods ignoring the loss of the binarization process fail to binarize the weights in LSTMs despite their remarkable performance on CNNs and fully-connected networks. In BinaryConnect as an example, the weights are binarized during the forward computations by thresholding while Eq. (1) is used to estimate the loss w.r.t. the full-precision weights without considering the quantization loss. When the training is over, both the full-precision and binarized weights can be then used to perform the inference computations of CNNs and fully-connected networks ([Lin et al. (2015)]). However, using the aforementioned binarization approach in vanilla LSTMs fails to perform sequential tasks due to the gradient vanishing problem as discussed in [Hou et al. (2016)]. We attribute this problem to the covariate shift phenomenon. More specifically, the binarization process affects the distribution of the hidden states at each time step during the recurrent computations and causes an internal covariate shift. To address the above issue, we propose the use of batch normalization in order to learn binarized/ternarized recurrent weights.

The main goal of our method is to represent each element of the full-precision weight \( W \) either as
\[
    w_{i,j} = A w_{i,j}^B, \quad \text{or} \quad w_{i,j} = A w_{i,j}^T,
\]
where \( A \) is a fixed scaling factor for all the weights and initialized from Glorot & Bengio (2010). To this end, we first divide each weight by the factor \( A \) to normalize the weights. We then compute the probability of getting binary or ternary values for each element of the full-precision matrix \( W \) by
\[
    P(w_{i,j} = 1) = \frac{w_{i,j}^N + 1}{2}, \quad P(w_{i,j} = -1) = 1 - P(w_{i,j} = 1),
\]
(4)
for binarization and
\[
    P(w_{i,j} = 1) = P(w_{i,j} = -1) = |w_{i,j}^N|, \quad P(w_{i,j} = 0) = 1 - P(w_{i,j} = 1),
\]
(5)
for ternarization, where \( w_{i,j}^N \) denotes the normalized weight. Afterwards, we stochastically sample from the Bernoulli distribution to obtain binarized/ternarized weights as follows
\[
    w_{i,j}^B = \text{Bernoulli}(P(w_{i,j} = 1)) \times 2 - 1, \quad w_{i,j}^T = \text{Bernoulli}(P(w_{i,j} = 1)) \times \text{sign}(w_{i,j}).
\]
(6)
Finally, we batch normalize the vector-matrix multiplications between the input and hidden state vectors with the binarized/ternarized weights \( W_{h}^{B/T} \) and \( W_{x}^{B/T} \). More precisely, we perform the recurrent computations as
\[
    \begin{pmatrix}
    f_t \\
    i_t \\
    o_t \\
    g_t
    \end{pmatrix} = \begin{pmatrix}
    \sigma \\
    \sigma \\
    \sigma \\
    \tanh
    \end{pmatrix}
    \text{BN}(W_{h}^{B/T} h_{t-1}; \alpha_h, 0) + \text{BN}(W_{x}^{B/T} x_t; \alpha_x, 0) + b.
\]
(7)

In fact, batch normalization reduces the internal covariate shift imposed by the binarization/ternarization between time steps. Moreover, batch normalization regulates the scale of binarized/ternarized weights using its parameters \( \alpha_h \) and \( \alpha_x \) in addition to \( A \).

So far, we have only considered the forward computations. During the parameter update, we use full-precision weights since the parameter updates are small values. To update the full-precision weights, we use Eq. (1) to estimate its gradient since the binarization/ternarization functions are indifferentiable (See Algorithm 1 and its details in Appendix A).

5 Experimental Results and Discussions

In this section, we evaluate the performance of the proposed training algorithm with recurrent binary/ternary weights on different temporal tasks to show the generality of our method. We defer hyperparameters and tasks details for each dataset to Appendix B due to the limited space.

5.1 Character-Level Language Modeling

For the character-level modeling, the goal is to predict the next character and the performance is evaluated on bits per character (BPC) where lower BPC is desirable. We conduct quantization
Table 1: Testing character-level BPC values of quantized LSTM models and size of their weight matrices in terms of KByte.

<table>
<thead>
<tr>
<th></th>
<th>Linux Kernel</th>
<th>War &amp; Peace</th>
<th>Penn Treebank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Precision</td>
<td>Test Size</td>
<td>Test Size</td>
</tr>
<tr>
<td>LSTM (baseline)</td>
<td>Full-precision</td>
<td>1.73</td>
<td>5024</td>
</tr>
<tr>
<td>LSTM with binary weights (ours)</td>
<td>1.79</td>
<td>157</td>
<td>1.78</td>
</tr>
<tr>
<td>BinaryConnect (Courbariaux et al. (2015))</td>
<td>4.24</td>
<td>157</td>
<td>5.10</td>
</tr>
<tr>
<td>LAB (Hou et al. (2016))</td>
<td>1.88</td>
<td>157</td>
<td>1.86</td>
</tr>
<tr>
<td>LSTM with ternary weights (ours)</td>
<td>1.75</td>
<td>314</td>
<td>1.72</td>
</tr>
<tr>
<td>TWN (Li &amp; Liu (2016))</td>
<td>Ternary</td>
<td>1.85</td>
<td>314</td>
</tr>
<tr>
<td>TTQ (Zhu et al. (2016))</td>
<td>Ternary</td>
<td>1.88</td>
<td>314</td>
</tr>
<tr>
<td>LAQ (Hou &amp; Kwok (2018))</td>
<td>Ternary</td>
<td>1.81</td>
<td>314</td>
</tr>
<tr>
<td>LAQ (Hou &amp; Kwok (2018))</td>
<td>3 bits</td>
<td>1.84</td>
<td>471</td>
</tr>
<tr>
<td>LAQ (Hou &amp; Kwok (2018))</td>
<td>4 bits</td>
<td>1.90</td>
<td>628</td>
</tr>
<tr>
<td>DoReFa-Net (Zhou et al. (2016))</td>
<td>3 bits</td>
<td>1.84</td>
<td>471</td>
</tr>
<tr>
<td>DoReFa-Net (Zhou et al. (2016))</td>
<td>4 bits</td>
<td>1.90</td>
<td>628</td>
</tr>
</tbody>
</table>

experiments on Penn Treebank (Marcus et al. (1993)), War & Peace (Karpathy et al. (2015)) and Linux Kernel (Karpathy et al. (2015)) corpora. For Penn Treebank dataset, we use a similar LSTM model configuration and data preparation to Mikolov et al. (2012). For War & Peace and Linux Kernel datasets, we also follow the LSTM model configurations and settings in (Karpathy et al. (2015)). Table 1 summarizes the performance of our binarized/ternarized models compared to state-of-the-art quantization methods reported in literature. All the models reported in Table 1 use an LSTM layer with 1000, 512 and 512 units on a sequence length of 100 for the experiments on Penn Treebank (Marcus et al. (1993)), War & Peace (Karpathy et al. (2015)) and Linux Kernel (Karpathy et al. (2015)) corpora, respectively. The experimental results show that our model with binary/ternary weights outperforms all the existing quantized models in terms of prediction accuracy. Moreover, our ternarized model achieves the same BPC values on War & Peace and Penn Treebank datasets as the full-precision model (i.e., baseline) while requiring 32× less memory footprint. It is worth mentioning the accuracy loss of our ternarized model over the full-precision baseline is small.

In order to evaluate the effectiveness of our method on a larger dataset for the character-level language modeling task, we use the Text8 dataset which was derived from Wikipedia. For this task, we use one LSTM layer of size 2000 and train it on sequences of length 180. We follow the data preparation approach and settings of Mikolov et al. (2012). The test results are reported in Table 2. While our models use recurrent binary or ternary weights during runtime, they achieve acceptable performance when compared to the full-precision models.

Table 2: Test character-level performance of our quantized models on the Text8 corpus.

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
<th>Test (BPC)</th>
<th>Size (MByte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM (baseline)</td>
<td>Full-precision</td>
<td>1.46</td>
<td>64.9</td>
</tr>
<tr>
<td>LSTM with binary weights (ours)</td>
<td>Binary</td>
<td>1.54</td>
<td>2.0</td>
</tr>
<tr>
<td>LSTM with ternary weights (ours)</td>
<td>Ternary</td>
<td>1.51</td>
<td>4.0</td>
</tr>
<tr>
<td>BinaryConnect (Courbariaux et al. (2015))</td>
<td>Binary</td>
<td>2.45</td>
<td>2.0</td>
</tr>
</tbody>
</table>

5.2 Word-level Language Modeling

Similar to the character-level language modeling, the main goal of word-level modeling is to predict the next word. However, this task deals with large vocabulary sizes, making the model quantization
Table 3: Test performance of the proposed LSTM with recurrent binary/ternary weights on the Penn Treebank (PTB) corpus.

<table>
<thead>
<tr>
<th></th>
<th>Word-PTB</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Model</td>
<td>Precision</td>
<td>Test (Perplexity)</td>
<td>Size (KByte)</td>
</tr>
<tr>
<td>Small LSTM (baseline)</td>
<td>Full-precision</td>
<td>91.5</td>
<td>2880</td>
<td>1.4</td>
</tr>
<tr>
<td>Small LSTM with binary weights (ours)</td>
<td>Binary</td>
<td>92.2</td>
<td>90</td>
<td>1.4</td>
</tr>
<tr>
<td>Small LSTM with ternary weights (ours)</td>
<td>Ternary</td>
<td>90.7</td>
<td>180</td>
<td>1.4</td>
</tr>
<tr>
<td>Small BinaryConnect LSTM (Courbariaux et al. (2015))</td>
<td>Binary</td>
<td>125.9</td>
<td>90</td>
<td>1.4</td>
</tr>
<tr>
<td>Small Alternating LSTM (Xu et al. (2018))</td>
<td>2 bits</td>
<td>103.1</td>
<td>180</td>
<td>2.9</td>
</tr>
<tr>
<td>Small Alternating LSTM (Xu et al. (2018))</td>
<td>3 bits</td>
<td>93.8</td>
<td>270</td>
<td>4.3</td>
</tr>
<tr>
<td>Small Alternating LSTM (Xu et al. (2018))</td>
<td>4 bits</td>
<td>91.4</td>
<td>360</td>
<td>5.8</td>
</tr>
<tr>
<td>Medium LSTM (baseline)</td>
<td>Full-precision</td>
<td>87.6</td>
<td>27040</td>
<td>6.7</td>
</tr>
<tr>
<td>Medium LSTM with binary weights (ours)</td>
<td>Binary</td>
<td>87.2</td>
<td>422</td>
<td>6.7</td>
</tr>
<tr>
<td>Medium LSTM with ternary weights (ours)</td>
<td>Ternary</td>
<td>86.1</td>
<td>845</td>
<td>6.7</td>
</tr>
<tr>
<td>Medium BinaryConnect LSTM (Courbariaux et al. (2015))</td>
<td>Binary</td>
<td>108.4</td>
<td>422</td>
<td>6.7</td>
</tr>
</tbody>
</table>

difficult. [Xu et al., 2018] introduced a multi-bit quantization method, referred to as alternating method, as a first attempt to reduce the complexity of the LSTMs used for this task. However, the alternating method only managed to almost match its performance with its full-precision counterpart using 4 bits (i.e., \(k = 4\)). However, there is a huge gap in the performance between its quantized model with 2 bits and the full-precision one. To show the effectiveness of our method over the alternating method, we use a small LSTM of size 300 similar to [Xu et al., 2018] for a fair comparison. We also examine the prediction accuracy of our method over the medium model introduced by Mikolov et al., 2010: the medium model contains an LSTM layer of size 650. We also use the same settings described in [Mikolov et al., 2010] to prepare and train our model. Table 3 summarizes the performance of our models in terms of perplexity. The experimental results show that our binarized/ternarized models outperform the alternating method using 2-bit quantization in terms of both perplexity and the memory size. Moreover, our medium-size model with binary weights also has a substantial improvement over the alternating method using 4-bit quantization. Finally, our models with recurrent binary and ternary weights yield a comparable performance compared to their full-precision counterparts.

5.3 SEQUENTIAL MNIST

We perform the MNIST classification task ([Le et al., 2015)] by processing each image pixel at each time step. In this task, we process the pixels in scanline order. We train our models using an LSTM with 100 nodes, followed by a softmax classifier layer. Table 4 reports the test performance of our models with recurrent binary/ternary weights. While our binary model uses a lower bit precision and fewer operations for the recurrent computations compared to the alternating models, its loss of accuracy is small. On the other hand, our ternary model requires the same memory size and achieves the same accuracy as the alternating method while requiring 2 \times fewer operations.

Table 4: Test accuracy of the proposed LSTM with recurrent binary/ternary weights on the pixel by pixel MNIST classification task.

<table>
<thead>
<tr>
<th></th>
<th>MNIST</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Model</td>
<td>Precision</td>
<td>Test (%)</td>
<td>Size (KByte)</td>
</tr>
<tr>
<td>LSTM (baseline)</td>
<td>Full-precision</td>
<td>98.9</td>
<td>162</td>
<td>80.8</td>
</tr>
<tr>
<td>LSTM with binary weights (ours)</td>
<td>Binary</td>
<td>98.6</td>
<td>5</td>
<td>80.8</td>
</tr>
<tr>
<td>LSTM with ternary weights (ours)</td>
<td>Ternary</td>
<td>98.8</td>
<td>10</td>
<td>80.8</td>
</tr>
<tr>
<td>BinaryConnect (Courbariaux et al. (2015))</td>
<td>Binary</td>
<td>68.3</td>
<td>5</td>
<td>80.8</td>
</tr>
<tr>
<td>Alternating LSTM (Xu et al. (2018))</td>
<td>2 bits</td>
<td>98.8</td>
<td>10</td>
<td>161.6</td>
</tr>
</tbody>
</table>
Table 5: Test error rate of Attentive Reader with recurrent binary/ternary weights on CNN question-answering task.

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
<th>Test (%)</th>
<th>Size (MByte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attentive Reader (baseline) Full-precision</td>
<td>40.19</td>
<td>7471</td>
<td></td>
</tr>
<tr>
<td>Attentive Reader with binary weights (ours)</td>
<td>Binary</td>
<td>40.78</td>
<td>233</td>
</tr>
<tr>
<td>Attentive Reader with ternary weights (ours)</td>
<td>Ternary</td>
<td>39.97</td>
<td>467</td>
</tr>
<tr>
<td>BinaryConnect Attentive Reader (Courbariaux et al. 2015)</td>
<td>Binary</td>
<td>94.66</td>
<td>233</td>
</tr>
</tbody>
</table>

5.4 QUESTION ANSWERING

[Hermann et al. 2015] recently introduced a challenging task that involves reading and comprehension of real news articles. More specifically, the main goal of this task is to answer questions about the context of the given article. To this end, they also introduced an architecture, called Attentive Reader, that exploits an attention mechanism to spot relevant information in the document. Attentive Reader uses two bidirectional LSTMs to encode the document and queries. To show the generality and effectiveness of our quantization method, we train Attentive Reader with our method to learn recurrent binary/ternary weights. We perform this task on the CNN corpus (Hermann et al. 2015) by replicating Attentive Reader and using the setting described in (Hermann et al. 2015). Table 5 shows the error rate of binarized/ternarized Attentive Reader. The simulation results show that our Attentive Reader with binary/ternary weights yields similar error rate to its full-precision counterpart while requiring 32× smaller memory footprint.

5.5 DISCUSSIONS

As discussed in Section 4, the training models ignoring the quantization loss fail to quantize the weights in LSTM while they perform well on CNNs and fully-connected networks. To address this problem, we proposed the use of batch normalization during the quantization process. To justify the importance of such a decision, we have performed different experiments over a wide range of temporal tasks and compared the accuracy performance of our binarization/ternarization method with binaryconnect as a method that ignores the quantization loss. The experimental results showed that binaryconnect method fails to learn binary/ternary weights. On the other hand, our method not only learns recurrent binary/ternary weights but also outperforms all the existing quantization methods in literature. It is also worth mentioning that the models trained with our method achieve a comparable accuracy performance w.r.t. their full-precision counterpart.

Figure 1 shows a histogram of the binary/ternary weights of the LSTM layer used for character-level language modeling task on the Penn Treebank corpus. In fact, our model learns to use binary or ternary weights by steering the weights into the deterministic values of -1, 0 or 1. Despite the CNNs or fully-connected networks trained with binary/ternary weights that can use either real-valued or binary/ternary weights, the proposed LSTMs trained with binary/ternary can only perform the

![Figure 1: Distribution of $W_h$ used in the LSTM layer on the Penn Treebank corpus.](image-url)
Figure 2 illustrates the learning curves and generalization of our method to longer sequences on the validation set of the Penn Treebank corpus. In fact, the proposed training algorithm also tries to retain the main features of using batch normalization, i.e., fast convergence and good generalization over long sequences. Figure 2(a) shows that our model converges faster than the full-precision LSTM for the first few epochs. After a certain point, the convergence rate of our method decreases, that prevents the model from early overfitting. Figure 2(b) also shows that our training method generalizes well over longer sequences. More specifically, the models with recurrent binary and ternary weights imitate the behavior of the baseline model when performing the temporal tasks over long sequences.

6 HARDWARE IMPLEMENTATION

In order to evaluate the effectiveness of LSTMs with recurrent binary/ternary weights, we build our binary/ternary architecture over the dataflow presented in (Chen et al. (2014)) as a baseline which has proven to be the most efficient dataflow for DNNs with sigmoid/tanh functions (Moshovos et al. (2018)). In this dataflow, a DRAM is used to store all the weights/activations and provide the required memory bandwidth for each multiply-accumulate (MAC) unit. For evaluation purposes, we consider two different architectures implementing Eq. 2: low-power implementation and high-speed inference engine. We build these two architectures based on the aforementioned dataflow. For the low-power implementation, we use 100 MAC units. We also use a 12-bit fixed-point representation for both weights and activations of the full-precision model as a baseline architecture. As a result, 12-bit multipliers are required to perform the recurrent computations. For the LSTMs with recurrent binary/ternary weights, a 12-bit fixed-point representation is only used for activations and multipliers in the MAC units are replaced with low-cost multiplexers. We implemented our low-power inference engine for both the full-precision and binary/ternary-precision models in TSMC 65-nm CMOS technology. The synthesis results are summarized in Table 6. They show that using recurrent binary/ternary weights results in up to 9× lower power and 10.6× lower silicon area compared to the baseline when performing the inference computations at 400 MHz.

Table 6: Implementation results of the proposed binary/ternary models vs full-precision models.

<table>
<thead>
<tr>
<th></th>
<th>Low-Power</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>Full-Prec</td>
<td>Binary</td>
<td>Ternary</td>
<td>Full-Prec</td>
</tr>
<tr>
<td># MAC Units</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Throughput (GOps/sec)</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Silicon Area (mm²)</td>
<td>2.56</td>
<td>0.24</td>
<td>0.42</td>
<td>2.56</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>336</td>
<td>37</td>
<td>61</td>
<td>336</td>
</tr>
</tbody>
</table>
For the high-speed design, we consider the same silicon area and power consumption for both the full-precision and binary/ternary-precision models. Since the MAC units of the binary/ternary-precision model require less silicon area and power consumption as a result of using multiplexers instead of multipliers, we can instantiate up to $10 \times$ more MAC units, resulting in up to $10 \times$ speedup compared to the full-precision model (see Table 6). It is also worth noting that the models using recurrent binary/ternary weights also require up to $12 \times$ less memory bandwidth than the full-precision models. More details on the proposed architecture are provided in Appendix C.

7 Conclusion

In this paper, we introduced a training algorithm that learns recurrent binary/ternary weights and eliminates most of the full-precision multiplications of the recurrent computations during the inference. We showed that the proposed training method generalizes well over long sequences and across a wide range of temporal tasks such as word/character language modeling and pixel by pixel classification tasks. We also showed that learning recurrent binary/ternary weights brings a major benefit to custom hardware implementations by replacing full-precision multipliers with hardware-friendly multiplexers and reducing the memory bandwidth. For this purpose, we introduced two hardware implementations: low-power and high-throughput implementations. The former architecture can save up to $9 \times$ power consumption and the latter speeds up the recurrent computations by a factor of 10.

References


APPENDIX A

Learning recurrent binary/ternary weights are performed in two steps: forward propagation and backward propagation.

Forward Propagation: A key point to learn recurrent binary/ternary weights is to batch-normalize the result of each vector-matrix multiplication with binary/ternary recurrent weights during the forward propagation. More precisely, we first binarize/ternarize the recurrent weights. Afterwards, the unit activations are computed while using the recurrent binarized/ternarized weights for each time step and recurrent layer. The unit activations are then normalized during the forward propagation.

Backward Propagation: During the backward propagation, the gradient with respect to each parameter of each layer is computed. Then, the updates for the parameters are obtained using a learning rule. During the parameter update, we use full-precision weights since the parameter updates are small values. More specifically, the recurrent weights are only binarized/ternarized during the forward propagation. Algorithm 1 summarizes the training method that learns recurrent binary/ternary weights. It is worth noting that batch normalizing the state unit $c$ can optionally be used to better control over its relative contribution in the model.

Algorithm 1: Training with recurrent binary/ternary weights. $J$ is the cross entropy loss function. $B/T(u)$ specifies the binarization/ternarization function. The batch normalization transform is also denoted by $BN(\cdot; \gamma, \beta)$. $L$ and $T$ are also the number of LSTM layers and time steps, respectively.

Data: Full-precision LSTM parameters $W_h, W_x$ and $b$ for each layer. Batch normalization parameters for hidden-to-hidden and input-to-hidden states (i.e., $\alpha_h, \beta_h, \alpha_x, \beta_x, \alpha_c$ and $\beta_c$). The classifier parameters $W_s$ and $b_s$. Input data $x^1$, its corresponding targets $y$ for each minibatch.

1. Forward Computations

   for $l = 1$ to $L$ do
   \[ W_{h_l}^T \leftarrow B/T(W_{h_l}^l) \]
   \[ W_{x_l}^T \leftarrow B/T(W_{x_l}^l) \]
   for $t = 1$ to $T$ do
   \[ \begin{pmatrix} f_l^t \\ i_l^t \\ o_l^t \\ g_l^t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} \cdot BN(W_{h_l}^T h_{l-1}^t; \alpha_h^l, 0) + BN(W_{x_l}^T x_l^t; \alpha_x^l, 0) + b_l \]
   \[ c_l^t = f_l^t \odot c_{l-1}^t + i_l^t \odot g_l^t \]
   \[ h_l^t = o_l^t \odot \tanh(BN(c_l^t; \alpha_c^l, \gamma_c^l)) \]
   \[ x_{l+1} = h_l^t \]
   end
   end

   \[ \hat{y} = \text{softmax}(W_s h^L + b_s) \]

2. Backward Computations

   Compute the cross entropy loss function $J$ knowing $\hat{y}$ and $y$

   Obtain the updates $\Delta W_s$ and $\Delta b_s$ by computing $\frac{\partial J}{\partial W_s}$ and $\frac{\partial J}{\partial b_s}$, respectively

   for $l = 1$ to $L$ do
   Obtain the updates $\Delta W_h^l$ and $\Delta W_x^l$ by computing $\frac{\partial J}{\partial W_h^l}$ and $\frac{\partial J}{\partial W_x^l}$, respectively
   Obtain the updates $\Delta b^l, \Delta h_0^l$ and $\Delta c_0^l$ by computing $\frac{\partial J}{\partial b^l}, \frac{\partial J}{\partial h_0^l}$ and $\frac{\partial J}{\partial c_0^l}$, respectively
   Obtain the updates $\Delta \alpha_h^l, \Delta \alpha_x^l, \Delta \alpha_c^l$ and $\Delta \beta_h^l, \Delta \beta_x^l, \Delta \beta_c^l$ by computing $\frac{\partial J}{\partial \alpha_h^l}, \frac{\partial J}{\partial \alpha_x^l}, \frac{\partial J}{\partial \alpha_c^l}$ and $\frac{\partial J}{\partial \beta_h^l}, \frac{\partial J}{\partial \beta_x^l}, \frac{\partial J}{\partial \beta_c^l}$, respectively
   end

   Update the network parameters using their updates
APPENDIX B

B.1 CHARACTER-LEVEL LANGUAGE MODELING

**Penn Treebank**: Similar to Mikolov et al. (2012), we split the Penn Treebank corpus into 5017k, 393k and 442k training, validation and test characters, respectively. For this task, we use an LSTM with 1000 units followed by a softmax classifier. The cross entropy loss is minimized on minibatches of size 64 while using ADAM learning rule. We use a learning rate of 0.002. We also use the training sequence length of size 100.

**Linux Kernel and Leo Tolstoy’s War & Peace**: Linux Kernel and Leo Tolstoy’s War and Peace corpora consist of 6,206,996 and 3,258,246 characters and have a vocabulary size of 101 and 87, respectively. We split these two datasets similar to Karpathy et al. (2015). We use one LSTM layer of size 512 followed by a softmax classifier layer. We use an exponentially decaying learning rate initialized with 0.002. ADAM learning rule is also used as the update rule.

**Text8**: This dataset has the vocabulary size of 27 and consists of 100M characters. Following the data preparation approach of Mikolov et al. (2012), we split the data into training, validation and test sets as 90M, 5M and 5M characters, respectively. For this task, we use one LSTM layer of size 2000 and train it on sequences of length 180 with minibatches of size 128. The learning rate of 0.001 is used and the update rule is determined by ADAM.

B.2 WORD-LEVEL LANGUAGE MODELING

**Penn Treebank**: Similar to Mikolov et al. (2010), we split the Penn Treebank corpus with a 10K size vocabulary, resulting in 929K training, 73K validation, and 82K test tokens. We start the training with a learning rate of 20. We then divide it by 4 every time we see an increase in the validation perplexity value. The model is trained with the word sequence length of 35 and the dropout probability of 0.5 and 0.65 for the small and medium models, respectively. Stochastic gradient descent is also used to train our model while the gradient norm is clipped at 0.25.

B.3 SEQUENTIAL MNIST

**MNIST**: MNIST dataset contains 60000 gray-scale images (50000 for training and 10000 for testing), falling into 10 classes. For this task, we process the pixels in scanline order: each image pixel is processed at each time step similar to Le et al. (2015). We train our models using an LSTM with 100 nodes, a softmax classifier layer and ADAM step rule with learning rate of 0.001.

![Figure 3: Latency of the proposed accelerator over full-precision, binary and ternary models.](image)
B.4 QUESTION ANSWERING

**CNN**: For this task, we split the data similar to Hermann et al. (2015). We adopt Attentive Reader architecture to perform this task. We train the model using bidirectional LSTM with unit size of 256. We also use minibatches of size 128 and ADAM learning rule. We use an exponentially decaying learning rate initialized with 0.003.

APPENDIX C

We implemented our binary/ternary architecture in VHDL and synthesized via Cadence Genus Synthesis Solution using TSMC 65nm GP CMOS technology. Figure 3 shows the latency of the proposed binary/ternary architecture for each time step and temporal task when performing the vector-matrix multiplications on binary/ternary weights. The simulation results show that performing the computations on binary and ternary weights can speed up the computations by factors of 10× and 5× compared to the full-precision models.