Delft University of Technology

Exploiting Symmetric Temporally Sparse BPTT for Efficient RNN Training

Chen, Xi; Gao, Chang; Wang, Zuowen; Cheng, Longbiao; Zhou, Sheng; Liu, Shih Chii; Delbruck, Tobi

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Exploiting Symmetric Temporally Sparse BPTT for Efficient RNN Training

Xi Chen[1], Chang Gao[2], Zuowen Wang[1], Longbiao Cheng[1], Sheng Zhou[1],
Shih-Chii Liu[1], Tobi Delbruck[1]

[1]Sensors Group, Institute of Neuroinformatics, University of Zurich and ETH Zurich
[2]Department of Microelectronics, Delft University of Technology
{xi, zuowen, longbiao, shengzhou, shih, tobi}@ini.uzh.ch, chang.gao@tudelft.nl

Abstract

Recurrent Neural Networks (RNNs) are useful in temporal sequence tasks. However, training RNNs involves dense matrix multiplications which require hardware that can support a large number of arithmetic operations and memory accesses. Implementing online training of RNNs on the edge calls for optimized algorithms for an efficient deployment on hardware. Inspired by the spiking neuron model, the Delta RNN exploits temporal sparsity during inference by skipping over the update of hidden states from those inactivated neurons whose change of activation across two timesteps is below a defined threshold. This work describes a training algorithm for Delta RNNs that exploits temporal sparsity in the backward propagation phase to reduce computational requirements for training on the edge. Due to the symmetric computation graphs of forward and backward propagation during training, the gradient computation of inactivated neurons can be skipped. Results show a reduction of ∼80% in matrix operations for training a 56k parameter Delta LSTM on the Fluent Speech Commands dataset with negligible accuracy loss. Logic simulations of a hardware accelerator designed for the training algorithm show 2-10X speedup in matrix computations for an activation sparsity range of 50%-90%. Additionally, we show that the proposed Delta RNN training will be useful for online incremental learning on edge devices with limited computing resources.

Introduction

Recurrent Neural Networks (**RNN**) are widely used in applications involving temporal sequence inputs such as edge audio voice wakeup, keyword spotting, and spoken language understanding. These RNNs are commonly trained once and then deployed, but there is an opportunity to continually improve their accuracy and classification power without giving up privacy by incremental training on edge devices. Training of RNNs on the edge requires a hardware platform that has enough computing resources and memory to support the large number of arithmetic operations and data transfers. This is because the computation in RNNs consists mainly of Matrix-Vector Multiplications (**MxV**), which is a memory-bounded operation. An effective method to reduce the energy consumption for training RNNs is to minimize the number of memory accesses.

Among various approaches that exploit sparsity in RNN inference to improve efficiency (Kadetotad et al. 2020; Srivastava et al. 2019; Chen, Blair, and Cong 2022; Lindmar, Gao, and Liu 2022), a previously proposed biologically inspired network model named Delta Network (**DN**) (Neil et al. 2017), uses temporal sparsity to dramatically reduce memory access and Multiply-Accumulate (**MAC**) operations during inference. By introducing a delta threshold on neuron activation changes, the update of slow-changing activations can be skipped, thus saving a large number of computes while achieving comparable accuracy. Hardware inference accelerators that exploit this temporal sparsity (Gao et al. 2018, 2020; Gao, Delbruck, and Liu 2022) can achieve 5-10X better energy efficiency with a custom design architecture that performs zero-skipping on sparse delta vectors. However, these accelerators only do inference, i.e., the forward propagation. This paper proves for the first time that the identical forward delta sparsity can be used in the backward propagation of training RNNs without extra accuracy loss and extends the DN framework to the entire training process. The main contributions of this work are:

1. The first mathematical formulation for Delta RNN training, showing that Delta RNN training is inherently a type of sparse Backpropagation Through Time (**BPTT**), utilizing the identical temporal sparsity during both forward and backward propagation. Moreover, due to this consistent temporal sparsity, any speed improvements seen during Delta RNN inference can also be observed during the training of these networks.

2. Empirical results showing that for a fixed number of training epochs, a Delta RNN training uses 7.3X fewer training operations compared to the dense RNN with only a factor of 1.16X increase in error rate on the Fluent Speech Commands Dataset (**FSCD**).

3. Empirical results showing that on the frequently used Google Speech Command Dataset (**GSCD**) used for edge keyword spotting, ∼80% training operations can be saved in an incremental learning setting.

4. Register Transfer Level (**RTL**) simulation results of the first hardware accelerator designed for training Delta RNNs which can achieve 2-10X speedup for an activation sparsity range of 50%-90%.
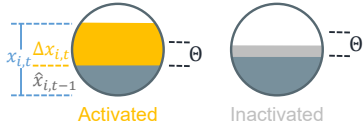
(a) Delta neuron is activated when the activation change $|\Delta|$ exceeds the delta threshold Θ at a certain timestep.
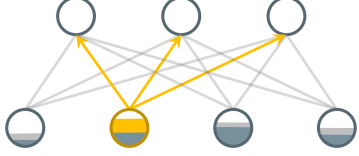
(b) Sparse forward propagation: Activated neurons propagate their Δ to the next layer, while other neurons remain unchanged.
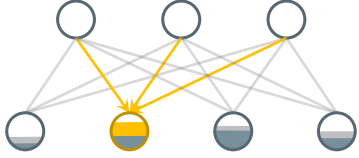
(c) Sparse backward propagation: Only these neurons need to transmit their Δ errors at that timestep in the backward phase.

Figure 1: Delta network concept for vanilla RNN with recurrent connections omitted.

Methodology

This section summarizes the key concepts of the Delta Network, extends the theory to the BPTT process of RNN training, and shows its theoretical reduction in computation costs.

Delta Network Formulation

In a vanilla RNN layer, the pre-activation vector Z_t and the hidden state vector h_t at time step t are given by:

$$Z_t = W_x x_t + W_h h_{t-1} + b_h \tag{1}$$

$$h_t = \tanh(Z_t) \tag{2}$$

where W_x, W_h are the weight matrices for input and hidden states respectively, x_t is the input vector, and b_h is the bias vector. In the DN formulation, these are calculated recursively by adding a new state variable vector, M_t holding a preactivation memory:

$$M_t = W_x \Delta x_t + W_h \Delta h_{t-1} + M_{t-1} \tag{3}$$
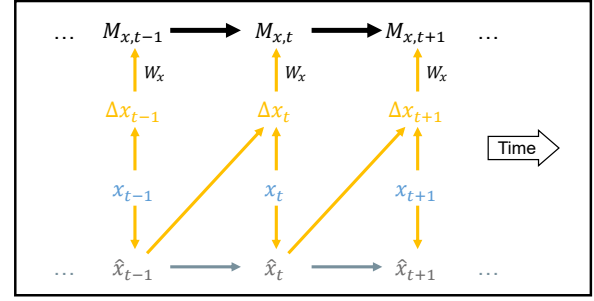
$$h_t = \tanh(M_t) \tag{4}$$

if we define delta vectors Δx and Δh as:
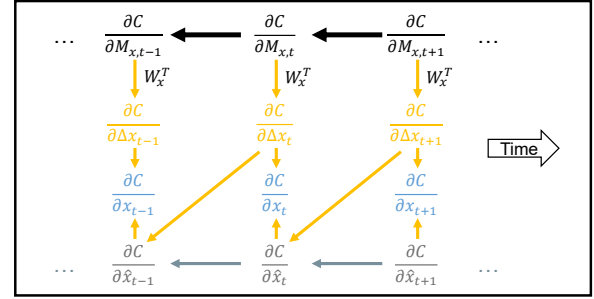
$$\Delta x_t = x_t - x_{t-1} \tag{5}$$

$$\Delta h_t = h_t - h_{t-1} \tag{6}$$

with the initial state $M_0 = b_h$, where M_{t-1} stores the preactivation from the previous time step.

Fig. 2a illustrates the x_t part of these inference equations as a graph to clarify the steps and data dependencies. It shows how $\hat{x}_{t-1}$ and x_t combine to form Δx_t and how it is multiplied by W_x to compute $M_{x,t}$.

(a) Forward computation graph illustrating input x_t.

(b) Backward computation graph illustrating input x_t.

Figure 2: Inference and training computation graphs of vanilla Delta RNNs illustrating input x_t. Graphs illustrating hidden states h_t are similar. Gold and grey arrows denote the paths for activated and inactivated neurons respectively, and black arrows include paths for all neurons.

Since the internal states of an RNN have temporal stability, the DN zeros out small changes in activations and hidden states, i.e. , the DN treats the values in $|\Delta x_t|$ and $|\Delta h_t|$ as zeros if they are smaller than a given delta threshold Θ, and those neurons are considered as "inactivated" (Fig. 1a). Then the DN only propagates the changes of those activated neurons (Fig. 1b), while the inactivated neurons keep current states until their changes go over the threshold later.

Formally, $\hat{x}_{i,t}$ denotes the latest value of the i-th element of the input vector at the t-th time step. The values $\hat{x}_{i,t}$ and $\Delta x_{i,t}$ will only be updated if the absolute difference between the current input $x_{i,t}$ and the previously stored state $\hat{x}_{i,t-1}$ is larger than the delta threshold Θ:

$$\hat{x}_{i,t} = \begin{cases} x_{i,t}, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ \hat{x}_{i,t-1}, & \text{otherwise} \end{cases} \tag{7}$$

$$\Delta x_{i,t} = \begin{cases} x_{i,t} - \hat{x}_{i,t-1}, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

The same update rules are applied to the hidden state h_t. Eqs. (3-8) summarize the delta principle in (Neil et al. 2017).

Until now, this principle has been applied only to the inference process, i.e., the forward propagation phase. Here we show that the delta sparsity can also be exploited in the backward propagation during the training process.

$[\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ ]$ $m_{x,t}$ mask

$[\ \ \ 1\ \ \ \ \ 4\ 5\ \ \ 7\ ]$ Δx_t NZIL

$M_{x,t} = W_x \Delta x_t$

$\dfrac{\partial C}{\partial \Delta x_t} = W_x^T \dfrac{\partial C}{\partial M_{x,t}}$

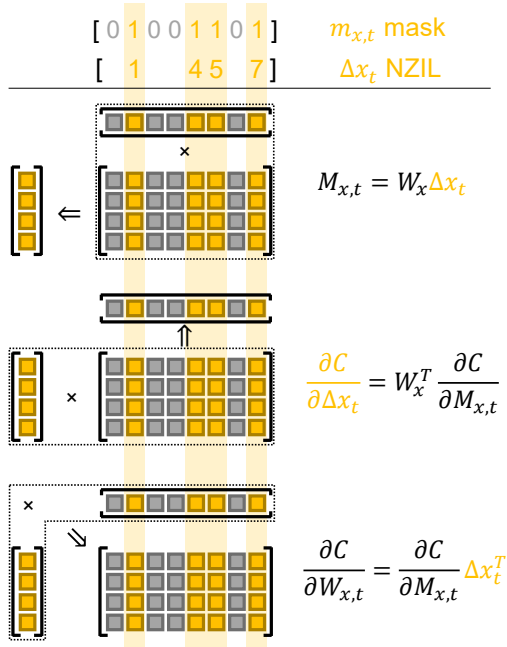$\dfrac{\partial C}{\partial W_{x,t}} = \dfrac{\partial C}{\partial M_{x,t}} \Delta x_t^T$

Figure 3: Sparse MxV in training Delta RNNs.

Intuition In the gradient-descent approach, we adjust the network weights to reduce the loss, and the gradient of a state vector depends on: a) how much the vector contributes to the network output; b) how much the output deviates from the ground truth. The changes of inactivated neurons are discarded in forward propagation (Fig. 1b) and make no contribution to the output, so their gradients are not needed. Therefore, we only need to propagate the errors of the changes of activated neurons, and calculate the weight gradients of the corresponding connections (Fig. 1c). Detailed mathematical proofs for vanilla RNN, Gated Recurrent Unit (**GRU**) (Cho et al. 2014), and Long Short-Term Memory (**LSTM**) (Hochreiter and Schmidhuber 1997) are given in Appendix A (Chen et al. 2023).

Proof Outline for Sparse BPTT in Vanilla Delta RNNs
In the backward phase, to allow skipping of the neurons with unchanging activations, we store a binary mask vector m_t during forward propagation:

$$m_{i,t} = \begin{cases} 1, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

which indicates the neurons that are activated at the t-th time step during forward propagation. Eq. (8) then reduces to:

$$\Delta x_t = m_t \odot (x_t - \hat{x}_{t-1}) \tag{10}$$

In backward propagation, as shown in Fig. 2b, we only compute the partial derivative of the cost C with respect to the change Δx_t of the activated neurons:

$$\frac{\partial C}{\partial \Delta x_t} = \left(W_x^\intercal \frac{\partial C}{\partial M_t} \right) \odot m_t \tag{11}$$

and calculate the weight gradients of those activated neurons:

$$\frac{\partial C}{\partial W_x} = \sum_{t=1}^T \frac{\partial C}{\partial M_t} \Delta x_t^\intercal = \sum_{t=1}^T \frac{\partial C}{\partial M_t} (m_t \odot (x_t - \hat{x}_{t-1}))^T \tag{12}$$

where $C = \sum_{t=1}^T L_t$ is the loss function values L_t summed across all time steps, $\odot$ denotes the element-wise multiplication, and T is the total number of time steps. The gradients for Δh_t can be derived similarly. More specifically, the vector $\frac{\partial C}{\partial \Delta x_t}$ is not sparse, but the gradients of the inactivated neurons will be zeroed out during backward propagation due to the non-differentiability of Eq. (8) in the below-threshold case. So those values in $\frac{\partial C}{\partial \Delta x_t}$ are not needed, and we can treat them as zeros using m_t and skip their computations.

The formulations in Appendix A (Chen et al. 2023) show that the sparse versions of the backward propagation equations (Eqs. 11 and 12) are equivalent to the dense versions for DN, i.e. they result in exactly the same weight changes. Therefore, exploiting temporal sparsity in backward propagation will not cause extra accuracy loss when the delta threshold has already been applied in forward propagation.

Fig. 3 illustrates how the DN uses temporal sparsity in the MxV operations arising from Δx_t in the forward (Eq. 3) and backward (Eqs. 11 and 12) computations. We can store the indices of activated neurons at each timestep as a binary mask m_t or a Non-Zero Index List (**NZIL**) during forward propagation. Since m_t can be applied to Eqs. (10)-(12), the MxV operations in backward propagation can share exactly the same sparsity. This sparsity allows for skipping entire columns of the weight matrices, thus the sparsity pattern is hardware-friendly.

Theoretical Reduction in Computations and Memory Accesses

Proposition 1. *In training a vanilla Delta RNN layer (model in Eqs. 3 and 4), the computational cost for calculating the gradients of weights $W_{x,t}$ or $W_{h,t}$ during each BPTT time step decreases linearly with the sparsity of delta input Δx_t or delta states Δh_t respectively. The total computation cost for the gradients of W_x or W_h with BPTT is the sum of terms proportional to the sparsity of Δx_t or Δh_t at each time step.*

Proof. In a Delta RNN layer, the computation in the forward propagation phase is formulated as Eqs. (3, 4, 7 & 8). In the backward propagation phase, the cost function is the sum of the loss function at every timestep: $C = \sum_{t=1}^T L_t$. Hidden states h_t are a function of M_t associated with any activation function and loss function L_t is a function of h_t and ground truth value at that time step. The gradient of weight W_x is the partial derivative of the cost C w.r.t. W_x:

$$\frac{\partial C}{\partial W_x} = \frac{\partial(\sum_{t=1}^T L_t)}{\partial W_x} = \sum_{t=1}^T \frac{\partial L_t}{\partial W_x} \tag{13}$$

For each timestep, according to the chain rule:

$$\frac{\partial L_t}{\partial W_x} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial W_x} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial M_t} \frac{\partial M_t}{\partial W_x} \tag{14}$$

From Eq. (3) we can expand the last term:

$$\frac{\partial M_t}{\partial W_x} = \frac{\partial(W_x \Delta x_t + W_h \Delta h_{t-1} + M_{t-1})}{\partial W_x}$$
$$= \Delta x_t^\mathsf{T} + \frac{\partial M_{t-1}}{\partial W_x} \qquad (15)$$

Using telescoping we get

$$\frac{\partial M_t}{\partial W_x} = \Delta x_t^\mathsf{T} + \Delta x_{t-1}^\mathsf{T} + \cdots + \Delta x_1^\mathsf{T} = \sum_{i=1}^{t} \Delta x_i^\mathsf{T} \quad (16)$$

Then Eq. (14) becomes:

$$\frac{\partial L_t}{\partial W_x} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial M_t} \left(\sum_{i=1}^{t} \Delta x_i^\mathsf{T} \right) \qquad (17)$$

For simplicity we write $\frac{\partial L_t}{\partial h_t} = L'(h_t)$ and $\frac{\partial h_t}{\partial M_t} = h'(M_t)$. By substituting them into Eq. (13), we get:

$$\frac{\partial C}{\partial W_x} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_x}$$
$$= \sum_{t=1}^{T} \left[L'(h_t) h'(M_t) \left(\sum_{i=1}^{t} \Delta x_i^\mathsf{T} \right) \right]$$
$$= \sum_{i=1}^{T} \sum_{t=i}^{T} L'(h_t) h'(M_t) \Delta x_i^\mathsf{T} \qquad (18)$$
$$= \sum_{t=1}^{T} \left[\underbrace{\left(\underbrace{\sum_{i=t}^{T} L'(h_i) h'(M_i)}_{①} \right) \Delta x_t^\mathsf{T}}_{②} \right]$$

① is a partial sum that has constant complexity in each timestep in BPTT. ② is a vector outer product performed at each timestep.

For a vector Δx_t of size n and occupancy o_c (the percentage of non-zero elements), the computation cost of ② is

$$C_{\text{comp,sparse}} = o_c \cdot n^2$$

which decreases linearly with the sparsity of Δx_t. The total cost of Eq. (18) is the sum of ② across all timesteps, thus proportional to the sparsity of Δx_t at each timestep.

The intuition for Proposition 1 is provided in Figs. 2 and 3. The computational complexities of most commonly used optimizers (SGD, Adam (Kingma and Ba 2015)) are linearly related to the complexity of computing the gradients of cost function w.r.t. the weights.

The memory access cost for ② is

$$C_{\text{mem,sparse}} = o_c \cdot n^2 + 3n$$

consisting of fetching $o_c \cdot n^2$ weights for W_x, reading n values for Δx_t and n values for the mask m_t, and writing n values for $\frac{\partial C}{\partial W_{x,t}}$. The costs for Eq. (11) can be derived analogously. Therefore, the cost of matrix operations in backward propagation is reduced to:

$$C_{\text{sparse}} / C_{\text{dense}} \approx (o_c \cdot n^2)/n^2 = o_c.$$

Threshold Θ		-		0.033		0.067	
MxV	FP	D	**Sp**	**Sp**	Sp	Sp	
type[*]	BP	D	**D**	**Sp**	D	Sp	
Accuracy (%)		92.9	**92.2**	**92.2**	91.8	91.8	
MACs	FP	53.2	13.4	13.4	7.7	7.7	
(K)	BP	106.5	106.5	26.8	106.5	15.5	
Sparsity	FP	-	**74.8**	**74.8**	85.5	85.5	
(%)	BP	-	0	**74.8**	0	85.5	

[*] "Sp" means sparse MxV operations and "D" is dense.

Table 1: Dense/sparse BPTT comparison for LSTM training.

Because the sparsity in Eqs. (11) and (12) is exactly the same as in Eq. (10), the computation and memory access costs of them are also reduced to o_c. For example, for a Δx_t sparsity of 90%, the occupancy o_c is 10%, the computation and memory access costs are reduced to 10%, thus the theoretical speedup factor is $1/o_c$ = 10X. This illustrates the nice property of Delta Networks, that is, once we induce temporal sparsity in forward propagation, it can be exploited in all the three MxVs in both forward and backward propagation with nearly the same efficiency.

The sparse DN training method requires marginal additional memory storage. Each timestep in forward propagation we store a binary mask m_t for Δx or Δh which only takes up n bits in the memory for a delta vector of size n.

Experiments

In this section, we first compare the accuracy and sparsity of original RNN models and Delta RNN models to verify the mathematical correctness of the sparse BPTT training method. Next, we evaluate the performance of Delta RNNs on speech tasks, for both batch-32 training from scratch and batch-1 incremental learning. Finally we establish the benchmark of a custom hardware accelerator designed for training Delta RNNs. For software experiments we implement Delta RNNs in Pytorch using custom functions for forward and backward propagation. Software experiments are conducted on a GTX 2080 Ti GPU. The hardware accelerator is implemented using Hardware Description Language (**HDL**) and benchmarked in the Vivado simulator.

Spoken Language Training from Scratch Experiments

We use the FSCD (Lugosch et al. 2019) to verify the mathematical correctness of the sparse version of BPTT, and to evaluate the accuracy and cost of the Delta RNNs on Spoken Language Understanding (**SLU**) tasks. The dataset contains 30,043 utterances from 97 speakers for controlling smart-home appliances or virtual assistants. It has 248 SLU phrases mapping to 31 intents with three slots: action, object, and location, e.g. "Make it softer" is the same as "Turn down the volume". The network model is a two-layer LSTM or Delta LSTM that both have 64 neurons, followed by one fully connected layer for classification. The model is trained
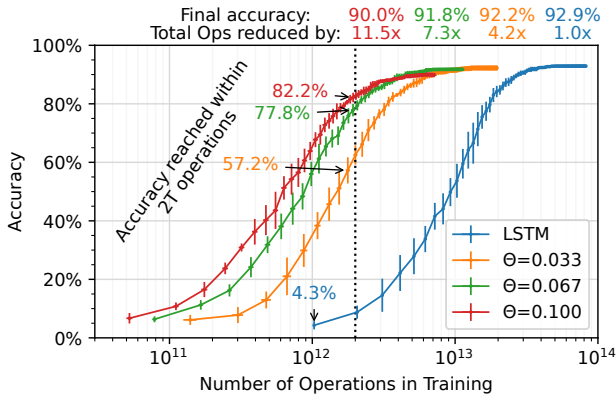
Figure 4: FSCD test accuracy vs number of training operations for LSTMs and Delta LSTMs. For each point, the x-coordinate is the number of MxV operations performed up to this epoch, and the y-coordinate is the test accuracy at this epoch. The top and bottom data labels show the best accuracy achieved within 80 epochs and the reduction factor in operations compared with LSTM.
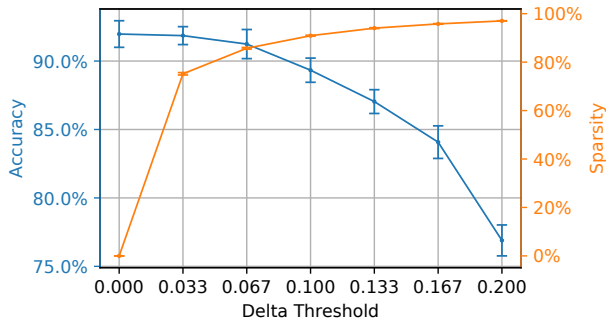

Figure 5: FSCD accuracy and sparsity vs delta threshold.

for 80 epochs with learning rate 1e-3 and batch size 32. We use cosine annealing scheduler, ADAM optimizer, and weight decay coefficient of 1e-2. Test results are averaged over 5 runs with different random seeds.

Results   Table. 1 compares the training results of dense/sparse BPTT (D/Sp on the second row) for Delta LSTM. The third column shows the baseline of the original LSTM model with regular dense training (automatic differentiation). The identical accuracy and sparsity in bold in the two columns illustrates the mathematical equivalence of the masked BPTT equations with the original ones for Delta LSTM.

Fig. 4 compares the classification accuracy versus training cost of a standard LSTM model against Delta LSTMs with various delta thresholds. When the delta threshold Θ increases, the number of operations needed to train the model to a given accuracy decreases dramatically, but the accuracy of Delta LSTM only slightly decreases. After a set 80 epochs of training, the Delta LSTM with Θ=0.067 (green curve) requires 7.3X fewer training operations than the LSTM with only a factor of 1.155X increase in error rate. When computing resources are limited, the Delta Network can offer

accurate training with an acceptable accuracy loss.

Fig. 5 shows how accuracy and sparsity change with respect to the delta threshold. By carefully choosing a delta threshold from experiments, one can obtain a good trade-off between accuracy and sparsity, where the training can be significantly sped up without hurting accuracy.

Incremental Keyword Learning Experiments

An attractive application of DNs is for online incremental training, where new labeled data become available in the field to an edge device and must be incorporated into the RNN to personalize or improve accuracy. To evaluate the performance of Delta RNNs on Class-Incremental Learning (**CIL**) tasks, we use GSCD v2 (Warden 2018), a dataset frequently used for benchmarking ASIC and FPGA keyword spotting implementations (Shan et al. 2020; Giraldo, Jain, and Verhelst 2021). The dataset contains 105,829 utterances of 35 English words, and it is divided into train/test sets with the ratio 8:2. We employ iCaRL (Rebuffi et al. 2017) as the CIL algorithm and use its evaluation method. The test accuracy is averaged over 5 runs with random permutation of classes, so for each run the order of classes learned by the model is different. The network model is a one-layer LSTM or Delta LSTM that has 128 neurons, followed by one fully-connected layer for classification. We pretrain the model to learn 20 classes in 20 epochs with learning rate 1e-3 for batch-32 and learning rate 1e-4 for batch-1. Then we retrain the model to learn several new classes in 20 epochs with the same learning rates for each CIL task, during which the model only has access to a limited set of 2000 exemplars of the previously learned classes, until the model finishes all tasks and learns all 35 classes. We use ADAM optimizer and weight decay coefficient of 1e-2.

Results   Table 2 shows the results. The first column is the CIL setting, and we group the experiment results according to 3 different settings. The first setting "35" is the baseline where the network model learns all 35 classes directly. "20+3x5" means pretraining the model to learn 20 classes and then retraining the model to learn 3 additional classes each step for 5 times. The fourth column shows the final accuracy after learning all 35 classes. The second last column shows the number of MAC operations per timestep in the LSTM layer. The last column shows the number of DRAM accesses for weights or weight gradients per timestep per batch in the LSTM layer.

It is clear from the table that the Delta LSTM models have ~80% sparsity and can save this proportion of MAC operations during training. Moreover, because the weight columns of inactivated neurons are not used during training and the memory access of them can be skipped as shown in Fig. 3, we can also save ~80% memory access in the batch-1 training of Delta RNNs. This significantly reduces energy consumption during training, because memory access consumes at least 10X more energy than arithmetic operations with the same bit width (Horowitz 2014). In an online setting, updating the weights using a batch size of 1 is natural and also desirable if the on-chip memory resources are too limited.

CIL setting	Model	Batch size	Accuracy (%)	Sparsity (%)	MACs (K)	DRAM W accesses (K words)
35	LSTM	32	90.8	-	221.2	221.2
	Delta LSTM	32	90.5	81.7	40.4	221.2
20+3x5	LSTM	1	82.3	-	221.2	221.2
	Delta LSTM	**1**	80.3	**79.8**	**44.7**	**44.7**
20+1x15	LSTM	1	76.6	-	221.2	221.2
	Delta LSTM	**1**	74.8	**79.8**	**44.7**	**44.7**

Table 2: Test accuracy, sparsity, and number of operations for CIL with LSTM models and Delta LSTM models ($\Theta=0.1$) on GSCD. For all models the network size is 16-128H-12 with 73.7k parameters.
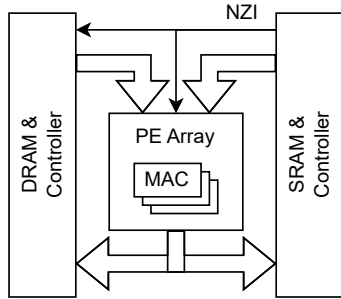
(a) Accelerator block diagram.
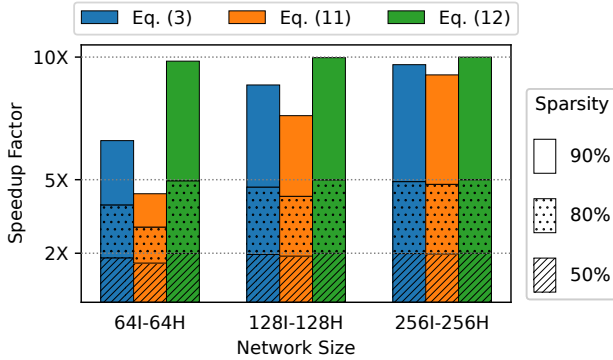
(b) Speedup measured by RTL simulation of the accelerator.

Figure 6: System architecture and Vivado simulation results of hardware accelerator for training Delta RNNs.

Hardware Simulation of Delta Training Accelerator

To demonstrate the speedup of Delta RNNs, we evaluated the potential performance of a hardware training accelerator, through the clock-cycle accurate RTL simulation of the design frequently used for logic design on FPGAs and custom ASICs. The RTL simulation closely emulates the actual hardware latency and logic costs of a custom silicon implementation. The accelerator (Fig. 6a) mainly consists of a Processing Element (**PE**) array where each PE can perform a MAC operation in parallel at each clock cycle. To efficiently exploit the temporal sparsity of Delta RNNs, the accelera-

tor stores sparse activation vector data in its Static Random Access Memory (**SRAM**) in the format of NZIL and Non-Zero Value List (**NZVL**), which enables the accelerator to skip computation (and Dynamic Random Access Memory (**DRAM**) memory access of entire weight columns if the batch size is 1) for zero elements in delta vectors (Fig. 3) (Gao et al. 2018). Dynamically skipping weight columns ideally matches properties of burst mode DRAM memory access, where addressing DRAM columns is slow but reading them out is fast. The accelerator employs the PE architecture in (Chen et al. 2021) so that it maintains this burst-mode memory access pattern for weight matrices during the forward and backward phases.

Accelerator Computation Flow

- Eq. (3), FP: The accelerator reads the NZIL of input Δx_t, which is a list containing indices of activated neurons at t-th timestep, and fetches the corresponding weight columns from DRAM, then multiply it with the Nonzero Values (**NZV**) of Δx_t.
- Eq. (11), BP input gradient: The same weight columns are fetched using the Δx_t NZIL, and they are multiplied with the gradient of pre-activation $\frac{\partial C}{\partial M_t}$ to produce the gradients of activated neurons in Δx_t.
- Eq. (12), BP weight gradient: The accelerator fetches the gradient of pre-activation $\frac{\partial C}{\partial M_t}$ and multiply it with the NZVs of Δx_t, producing the gradient of weight columns $\frac{\partial C}{\partial W_x}$ for activated neurons. The NZIL is used for output indexing in this step.

The same operations are performed for hidden states Δh. The accelerator process input samples one by one, i.e., the batch size of training is 1. This way, the accelerator can skip both the DRAM access of weight columns of inactivated neurons and the computation for them.

Experiment Setup Fig. 6a shows the architecture of the accelerator that we instantiated with 16 PEs. It computes Eqs. (3), (11) and (12), which are the three MxVs of training a Delta RNN. We tested three different network sizes: 64, 128 and 256. For each network, the input size equals the hidden layer size, which also equals the number of timesteps of input data. Random input data of different sparsities (50%, 80%, and 90%) are generated to evaluate the performance of the accelerator. We measure the computation time in clock

cycles for each MxV equation, starting from the time when the input data is loaded into memory, to the time when PE array outputs the last data.

If there is no memory latency or communication overhead, and PEs are fully utilized, the computation time (in clock cycles) for a dense MxV during RNN training is

$$T_{\text{dense}} = \frac{N_l * N_{l-1} * T_s}{P}$$

where N_l, T_s, and P denote the size of the l-th layer, the length of the s-th input sequence, and the number of PEs in the accelerator respectively. The speedup factor is given by

$$F_{\text{speedup}} = \frac{T_{\text{dense}}}{T_{\text{measured}}}$$

where T_{measured} is the measured computation time in simulation. We compare the ideal result with the realistic RTL simulation of the accelerator.

Experiment Results Fig. 6b shows the speedup factors measured in the RTL simulations for different network sizes and activation sparsities. For input data of 50%, 80%, and 90% sparsity, the 256I-256H Delta RNN nearly achieves 2X, 5X, and 10X training speedup, which are the theoretical speedup factors calculated in Section . For smaller networks such as 64I-64H, the speedup factors for Eqs. (3) and (11) are lower, especially when the sparsity is high. This is because the two equations are calculated timestep by timestep, and the computation time of each timestep is short for small network or highly sparse data, making the overhead of communication and computation relatively more apparent in this case. The weight gradient calculation for all timesteps is executed at the end of BPTT, so the speedup factors of Eq. (12) are close to the theoretical values for all tested cases. In a custom training Delta RNN hardware accelerator using the NZIL-NZVL data format, we can see a significant boost in the speedup factor with high activation sparsity. The Delta RNN training accelerator would improve RNN incremental batch-1 training by a factor of about 5-10X compared to a dense RNN training accelerator.

Related Works

The computational efficiency of neural networks can be improved by introducing sparsity into the networks. (O'Connor and Welling 2016) proposed an approach similar to Delta Network for CNNs to reduce the inference cost but (Aimar et al. 2019; Neil et al. 2017) showed that using DN requires doubled activation memory access because it must be read to check if it has changed, and then written, which ends up doubling the CNN inference cost because it is dominated by activation memory. By contrast, using the DN on RNNs is beneficial because the fully-connected RNNs are weight-memory bounded, and the energy savings brought by temporal sparsity is much larger for RNNs. (Gao, Delbruck, and Liu 2022) and (Hunter, Spracklen, and Ahmad 2022) exploit both DN activation sparsity and weight sparsity to achieve impressive inference performance on hardware.

Another method that can create sparsity in neural networks is conditional computation, or skipping operations.

Zoneout (Krueger et al. 2017) randomly selects whether to carry forward the previous hidden state or update it with the current hidden state during training. It aims to prevent overfitting in RNNs and provides only limited sparsity.

Skip RNN (Campos et al. 2017) uses a gating mechanism to learn whether to update or skip hidden states at certain timesteps, which is trained to optimize the balance between computational efficiency and accuracy. The skipping is applied to the whole RNN layers, rather than element-wise on activation vectors as in Delta RNNs.

EGRU (Subramoney et al. 2022) uses event-based communication between RNN neurons, resulting in sparse activations and a sparse gradient update. While this method resembles DN, its activation sparsity is different from the temporal sparsity of our work, and its sparsity in backward propagation is smaller by a factor of 1.5X to 10X than in forward propagation, unlike our Delta RNNs where the sparsity is identical for FP and BP. This asymmetry in forward and backward sparsity results from their surrogate gradient function which is non-zero within a certain range around the threshold to allow the errors near that point to pass through. Their surrogate gradient function makes the activation function differentiable, resulting in more accurate inference, but it can greatly decrease the sparsity in the backward pass.

(Perez-Nieves and Goodman 2021) also touches on sparse BPTT but uses Spiking Neural Networks (**SNN**). In a similar spirit, the authors show that computations can be saved by calculating the gradients only for active neurons (i.e. neurons producing a spike as defined by a threshold). However, the set of active neurons in backward propagation can be different from those in forward propagation. In contrast, the temporal sparsity in our work is identical in both the forward and backward propagation, thus the mask vectors computed in the forward pass can be directly reused in the backward pass. Their asynchronous SNN cannot efficiently use DRAM for weight memory because the weight memory accesses are unpredictable.

Conclusion

Training RNNs with BPTT involves a huge number of arithmetic operations and especially memory accesses, which leads to inefficient deployment on edge platforms. This paper shows that the temporal sparsity introduced in the Delta Network inference can also be applied during training leading to a sparse BPTT process. The MxVs operations in BPTT can be significantly sped up by skipping the computation and propagation of gradients for inactivated neurons. Our experiments and digital hardware simulations demonstrate that the number of matrix multiplication operations in training RNNs can be reduced by 5-10X with marginal accuracy loss on speech tasks. Furthermore, the number of memory accesses can also be reduced by the same factor if training with a batch size of 1 on a hardware accelerator specifically designed for Delta RNNs, saving substantial energy consumption. Therefore, our proposed training method would be particularly useful for continuous online learning on resource-limited edge devices that can exploit self-supervised information, such as errors between predictions and measurements.

Acknowledgments

References

Aimar, A.; Mostafa, H.; Calabrese, E.; and others. 2019. NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Transactions on Neural Networks and Learning Systems*, 30(3): 644–656.

Campos, V.; Jou, B.; Giró-i Nieto, X.; Torres, J.; and Chang, S.-F. 2017. Skip RNN: Learning to skip state updates in recurrent neural networks. *arXiv preprint arXiv:1708.06834*.

Chen, X.; Gao, C.; Delbruck, T.; and Liu, S.-C. 2021. EILE: Efficient incremental learning on the edge. In *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 1–4.

Chen, X.; Gao, C.; Wang, Z.; Cheng, L.; Zhou, S.; Liu, S.-C.; and Delbruck, T. 2023. Exploiting symmetric temporally sparse BPTT for efficient RNN training. arXiv:2312.09391.

Chen, Z.; Blair, H. T.; and Cong, J. 2022. Energy-efficient LSTM inference accelerator for real-time causal prediction. *ACM Trans. Des. Automat. Electron. Syst.*, 27(5): 1–19.

Cho, K.; van Merrienboer, B.; Gulcehre, C.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*.

Gao, C.; Delbruck, T.; and Liu, S.-C. 2022. Spartus: A 9.4 TOp/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity. *IEEE Transactions on Neural Networks and Learning Systems*.

Gao, C.; Neil, D.; Ceolini, E.; Liu, S.-C.; and Delbruck, T. 2018. DeltaRNN: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18.

Gao, C.; Rios-Navarro, A.; Chen, X.; Liu, S.-C.; and Delbruck, T. 2020. EdgeDRNN: Recurrent neural network accelerator for edge inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(4): 419–432.

Giraldo, J. S. P.; Jain, V.; and Verhelst, M. 2021. Efficient execution of temporal convolutional networks for embedded keyword spotting. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(12): 2220–2228.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780.

Horowitz, M. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 10–14.

Hunter, K.; Spracklen, L.; and Ahmad, S. 2022. Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks. *Neuromorphic Computing and Engineering*, 2(3): 034004.

Kadetotad, D.; Yin, S.; Berisha, V.; Chakrabarti, C.; and Seo, J. 2020. An 8.93 TOPS/W LSTM recurrent neural network accelerator featuring hierarchical coarse-grain sparsity for on-device speech recognition. *IEEE J. Solid-State Circuits*, 55(7): 1877–1887.

Kingma, D.; and Ba, J. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*. San Diego, CA, USA.

Krueger, D.; Maharaj, T.; Kramár, J.; Pezeshki, M.; Ballas, N.; Ke, N. R.; Goyal, A.; Bengio, Y.; Courville, A.; and Pal, C. 2017. Zoneout: Regularizing RNNs by randomly preserving hidden activations. arXiv:1606.01305.

Lindmar, J. H.; Gao, C.; and Liu, S.-C. 2022. Intrinsic sparse LSTM using structured targeted dropout for efficient hardware inference. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 126–129.

Lugosch, L.; Ravanelli, M.; Ignoto, P.; Tomar, V. S.; and Bengio, Y. 2019. Speech model pre-training for end-to-end spoken language understanding. In *Proc. Interspeech 2019*, 814–818.

Neil, D.; Lee, J.; Delbrück, T.; and Liu, S.-C. 2017. Delta networks for optimized recurrent network computation. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, 2584–2593.

O'Connor, P.; and Welling, M. 2016. Sigma delta quantized networks. *arXiv preprint arXiv:1611.02024*.

Perez-Nieves, N.; and Goodman, D. F. M. 2021. Sparse spiking gradient descent. In Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems*.

Rebuffi, S.-A.; Kolesnikov, A.; Sperl, G.; and Lampert, C. H. 2017. iCARL: Incremental classifier and representation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2001–2010.

Shan, W.; Yang, M.; Wang, T.; Lu, Y.; Cai, H.; Zhu, L.; Xu, J.; Wu, C.; Shi, L.; and Yang, J. 2020. A 510-nW wake-up keyword-spotting chip using serial-FFT-based MFCC and binarized depthwise separable CNN in 28-nm CMOS. *IEEE J. of Solid-State Circuits*, 56(1): 151–164.

Srivastava, G.; Kadetotad, D.; Yin, S.; Berisha, V.; Chakrabarti, C.; and Seo, J.-S. 2019. Joint optimization of quantization and structured sparsity for compressed deep neural networks. In *2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1393–1397.

Subramoney, A.; Nazeer, K. K.; Schöne, M.; Mayr, C.; and Kappel, D. 2022. Efficient recurrent architectures through activity sparsity and sparse back-propagation through time. In *The Eleventh International Conference on Learning Representations*.

Warden, P. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*.