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
[1] {xi, zuowen, longbiao, shengzhou, shih, tobi}@ini.uzh.ch
[2] chang.gao@tudelft.nl

Abstract

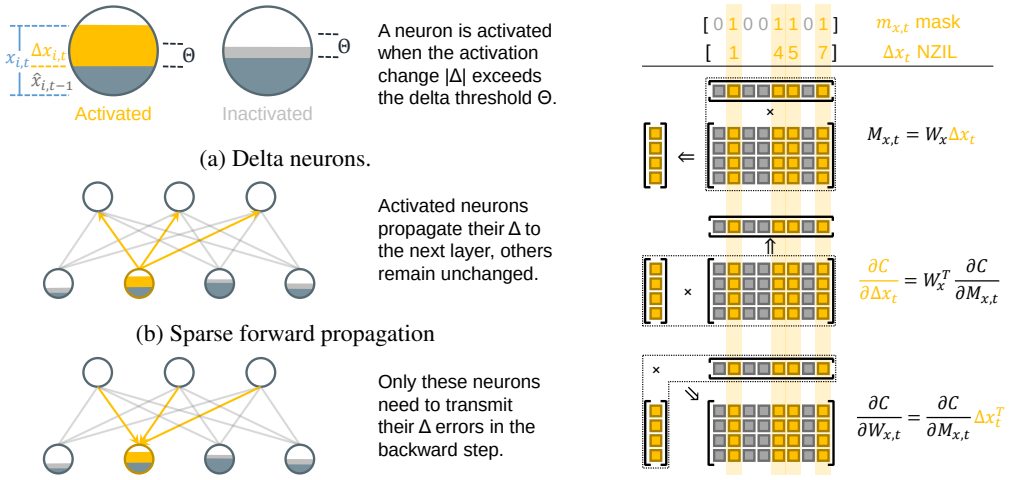
Recurrent Neural Networks (RNNs) are useful in temporal sequence tasks. However, training RNNs involves dense matrix multiplications which require hardware that can support a large number of arithmetic operations and memory accesses. Implementing online training of RNNs on the edge calls for optimized 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 $\sim$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 our training algorithm will be useful for online incremental learning on edge devices with limited computing resources.

1 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 [11, 21, 3, 14], a previously proposed biologically inspired network model named Delta Network (**DN**) [16], 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 [5–7]

37th First Workshop on Machine Learning with New Compute Paradigms at NeurIPS 2023(MLNPCP 2023).



(a) Delta neurons.

(b) Sparse forward propagation

(c) Sparse backward propagation

(d) Sparse MxV in training Delta RNNs.

Figure 1: Delta network concept and the sparse BPTT computations in traininig Delta RNNs.

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.
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 ∼80% training operations can be saved in an incremental learning setting of Delta RNNs on the Google Speech Command Dataset (**GSCD**).
4. Register Transfer Level (**RTL**) simulation results of the first Delta RNN training accelerator which show 2-10X speedup for an activation sparsity range of 50%-90%.

2 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 and memory costs.

2.1 Delta Network Formulation

In a vanilla RNN layer, the pre-activation vector Z_t is given by:

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

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. The hidden state vector is $h_t = \tanh(Z_t)$. In the DN formulation, Eq. (1) is calculated recursively by adding a new state variable vector, M_t holding a preactivation memory:

$$M_t = W_x \Delta x_t + W_h \Delta h_{t-1} + M_{t-1} \tag{2}$$

if we define delta vectors Δx and Δh as:

$$\Delta x_t = x_t - x_{t-1} \qquad \Delta h_t = h_t - h_{t-1} \tag{3}$$

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

Since the internal states of an RNN have temporal stability, the DN zeros out the changes in $|\Delta x_t|$ and $|\Delta h_t|$ which 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:

$$\hat{x}_{i,t} = \begin{cases} x_{i,t}, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ \hat{x}_{i,t-1}, & \text{otherwise} \end{cases} \qquad \Delta x_{i,t} = \begin{cases} x_{i,t} - \hat{x}_{i,t-1}, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

The hidden state h_t is updated the same way. Eqs. (2)-(4) summarize the delta principle in [16].

Until now, this principle has been applied only to the inference process, i.e., the Forward Propagation (FP) phase. Here we show that the delta sparsity can also be exploited in the Backward Propagation (BP) phase during the training process of DN.

The intuition comes from the fact that in the gradient-descent approach, the gradient of a state vector used to reduce the loss depends on how much it contributes to the network output. The changes of inactivated neurons are discarded in FP (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 BPTT formulations are given in Appendix A. Here we outline the proof for vanilla RNNs.

To allow later skipping the unchanging activations, we store a binary mask vector m_t during FP:

$$m_{i,t} = \begin{cases} 1, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

which indicates the neurons that are activated at the t-th time step. In BPTT, we only compute the gradients of the cost C w.r.t. the change of activated neurons Δx_t (for multi-layer RNNs):

$$\frac{\partial C}{\partial \Delta x_t} = \left(W_x^\intercal \frac{\partial C}{\partial M_t} \right) \odot m_t \tag{6}$$

and calculate the weight gradients of those activated neurons:

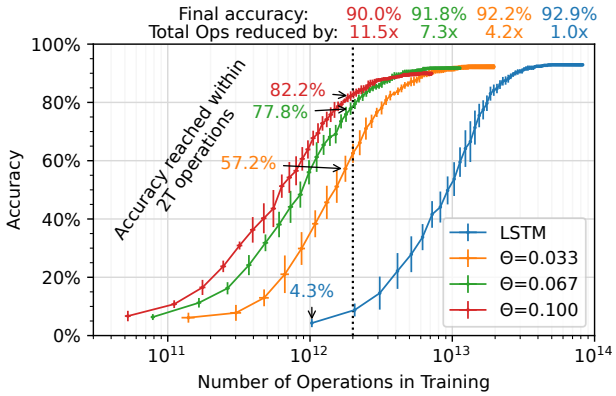
$$\frac{\partial C}{\partial W_x} = \sum_{t=1}^{T} \frac{\partial C}{\partial M_t} \Delta x_t^\intercal = \sum_{t=1}^{T} \frac{\partial C}{\partial M_t} (m_t \odot (x_t - \hat{x}_{t-1}))^T \tag{7}$$

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 BPTT due to the non-differentiability of Eq. (4) in the below-threshold case. So those values in $\frac{\partial C}{\partial \Delta x_t}$ are not needed and can be treated as zeros. This sparse version of BPTT (Eqs. 6 and 7) is equivalent to the dense version in DN, i.e., they result in exactly the same weight changes. Therefore, exploiting temporal sparsity in BPTT will not cause an extra accuracy loss when the delta threshold has already been applied in FP.

Fig. 1d illustrates how the sparse BPTT training method of DN exploits the temporal sparsity in the main matrix operations (Eqs. 2, 6 and 7). By storing a binary mask m_t or a Non-Zero Index List (**NZIL**) during FP, the calculations involving inactivated neurons and entire columns of their weights can be skipped in all these MxV operations. This partially regular sparsity pattern resembles structured sparsity and is hardware-friendly.

2.2 Theoretical Reduction in Computations and Memory Accesses

Proposition 1. *In training a vanilla RNN layer (described in Eq. 1), the computation 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.*



Dense/sparse BPTT comparison

Θ		-	0.067	
MxV type	FP	D	**Sp**	**Sp**
	BP	D	**D**	**Sp**
Accuracy (%)		92.9	**91.8**	**91.8**
MACs (K)	FP	53	8	8
	BP	107	107	16
Sparsity (%)	FP	-	**85.5**	**85.5**
	BP	-	0	**85.5**

Figure 2: Left: FSCD accuracy vs number of training operations for LSTMs and Delta LSTMs. Each point denotes an epoch where x and y are the number of MxV operations performed and the best accuracy reached up to this epoch respectively. Right: training results of dense/sparse BPTT.

The intuition for Proposition 1 is provided in Fig. 1d. We save the detailed proof in Appendix A. To compute the theoretical reduction of computation and memory access resulting from the sparse delta vectors, we define o_c to be the occupancy of a vector, i.e., the ratio of non-zero elements in a vector. According to the calculations in [16], the computation and memory access costs for sparse MxV in forward propagation is reduced to:

$$C_{\text{sparse}}/C_{\text{dense}} \approx (o_c \cdot n^2)/n^2 = o_c \tag{8}$$

where n is the size of the delta vector. Since the sparsity of the FP equation (4) is identical for the BP equations (6) and (7), the BP costs are also reduced to o_c (calculations given in Appendix A). This illustrates the nice property of Delta Networks: once we induce temporal sparsity in FP, it can be exploited in all the three MxVs in both FP and BP with nearly the same efficiency (Fig. 1d). For example, for a Δ sparsity of 90% (o_c = 10%), the computation and memory access costs are reduced to 10%, thus the theoretical speedup factor is $1/o_c$ = 10X for matrix computations in training.

3 Experiments

In this section, we first 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 Class-Incremental Learning (**CIL**). 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 FP and BP functions. The training experiments are conducted on a GTX 2080 Ti GPU. The hardware accelerator is implemented using Hardware Description Language (**HDL**) and is benchmarked in the Vivado simulator.

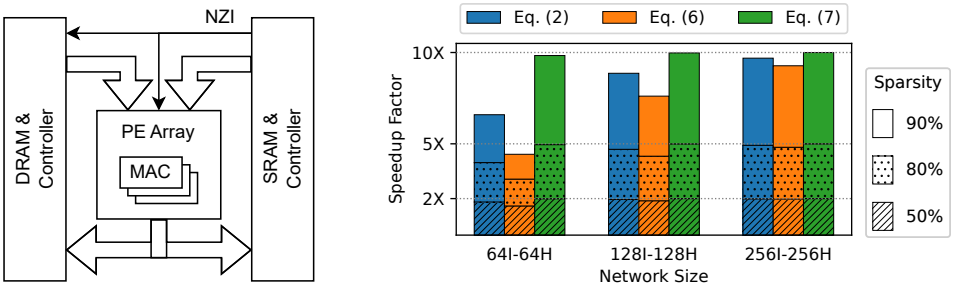
3.1 Spoken Language Training from Scratch Experiments

We use the FSCD [15] to evaluate the accuracy and cost of 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. 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 for 80 epochs with learning rate 1e-3 and batch size 32. Results are averaged from 5 random seeds.

Fig. 2 (left) compares the classification accuracy versus training computation 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. For 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 1.155X increase in error rate. In summary, when computing resources are limited, the Delta Network can offer efficient training with an acceptable accuracy loss.

Table 1: Test accuracy, sparsity, and number of operations for CIL with LSTM and Delta LSTM (Θ=0.1) models on GSCD. Every network model is 16-128H-12 with 73.7k parameters.

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**



(a) Accelerator block diagram.

(b) Speedup measured by RTL simulation of the accelerator.

Figure 3: System architecture and Vivado simulation results of the Delta RNN training accelerator.

The table in Fig. 2 (right) 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. The identical accuracy and sparsity in bold in the last two columns illustrates the mathematical equivalence of the masked BPTT equations with the original ones for Delta LSTM.

3.2 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 CIL tasks, we use GSCD v2 [23], a dataset frequently used for benchmarking ASIC keyword spotting implementations [12, 20, 8]. It contains 105,829 utterances of 35 English words. We employ iCaRL [19] as the incremental learning algorithm with exemplar set size $K = 2000$. Models are trained for 20 epochs with learning rate 1e-4 for each task. Test results are averaged over 5 runs with random permutations of classes.

Table 1 shows the results. The first column is the CIL setting, 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 forth 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 Delta LSTM models have $\sim$80% sparsity and can save this proportion of MAC operations during training. Moreover, the number of memory accesses for weights or weight gradients is also saved by the $\sim$80% when the batch size is 1. In an online learning setting, batch-1 training is natural and also desirable if the on-chip memory resources are too limited.

3.3 Hardware Simulation of Delta Training Accelerator

To demonstrate the speedup of the Delta RNN training method, we evaluated the potential performance of a hardware training accelerator in the RTL simulation. The accelerator (Fig. 3a) mainly

consists of a Processing Element (**PE**) array where each PE can perform an MAC operation in parallel at each clock cycle. To efficiently exploit the temporal sparsity of Delta RNNs, the accelerator stores sparse delta vectors in the format of NZIL and Non-Zero Value List (**NZVL**) [5] to enable the zero-skipping technique (Fig. 1d). The detailed computation flow is described in Appendix B.

Fig. 3a shows the architecture of the accelerator that we instantiated with 16 PEs. It computes Eqs. (2), (6) and (7), which are the three MxVs of training a Delta RNN. We tested three different network sizes: 64, 128 and 256. Random input data of fixed sparsities are generated to evaluate the performance of the accelerator. We measure computation time T_{measured} in clock cycles for each MxV equation, and calculate the speedup factor $F_{\text{speedup}} = T_{\text{dense}}/T_{\text{measured}}$, where T_{dense} is the theoretical computation time for a dense MxV equation of the same size in an ideal case.

Fig. 3b 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 speedup, which are the theoretical speedup factors calculated in Section 2.2. For smaller networks such as 64I-64H, the speedup factors for Eqs. (2) and (6) are lower due to the computation overhead. To conclude, the hardware accelerator would boost MxV operations in incremental batch-1 training by 5-10X with our sparse BPTT algorithm.

4 Related Works

The computational efficiency of neural networks can be improved by creating sparsity in the networks. [17] proposed an approach similar to DN for CNNs to accelerate the inference, but [1, 16] showed that it doubles the inference cost since CNNs are dominated by activation memory. By contrast, using DN on RNNs is beneficial because the fully-connected RNNs are weight-memory bounded, and the energy saving brought by temporal sparsity is much larger for RNNs. [7] and [10] 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 [13] 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 [2] 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 [22] 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. Their surrogate function makes the activation function differentiable but decreases the sparsity in the BP phase by a factor of 1.5X to 10X than in FP. [18] 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 BP can be different from those in FP. In contrast, in our work the mask vectors computed in FP can be directly reused in BP as the temporal sparsity is identical in both passes.

5 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 data, such as errors between predictions and measurements.

Acknowledgments and Disclosure of Funding

This project has received funding from Samsung Global Research "Neuromorphic Processor Project" (NPP), and the European Union's Horizon 2020 research and innovation programme under grant agreement No 899287 for the project "Neural Active Visual Prosthetics for Restoring Function" (NeuraViPeR).

References

[1] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, and Tobi Delbruck. 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, 2019. doi: 10.1109/TNNLS.2018.2852335.

[2] Víctor Campos, Brendan Jou, Xavier Giró i Nieto, Jordi Torres, and Shih-Fu Chang. Skip RNN: Learning to skip state updates in recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[3] Zhe Chen, Hugh T Blair, and Jason Cong. Energy-Efficient LSTM inference accelerator for Real-Time causal prediction. *ACM Trans. Des. Automat. Electron. Syst.*, 27(5):1–19, June 2022. ISSN 1084-4309. doi: 10.1145/3495006.

[4] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, 2014. doi: 10.3115/v1/D14-1179.

[5] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. DeltaRNN: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pp. 21–30, 2018. doi: 10.1145/3174243.3174261.

[6] Chang Gao, Antonio Rios-Navarro, Xi Chen, Shih-Chii Liu, and Tobi Delbruck. EdgeDRNN: Recurrent neural network accelerator for edge inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(4):419–432, 2020. doi: 10.1109/JETCAS.2020.3040300.

[7] Chang Gao, Tobi Delbruck, and Shih-Chii Liu. Spartus: A 9.4 TOp/s FPGA-based LSTM accelerator exploiting spatio-temporal sparsity. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[8] Juan Sebastian P Giraldo, Vikram Jain, and Marian Verhelst. Efficient execution of temporal convolutional networks for embedded keyword spotting. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(12):2220–2228, 2021.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, November 1997. doi: 10.1162/neco.1997.9.8.1735.

[10] Kevin Hunter, Lawrence Spracklen, and Subutai Ahmad. Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks. *Neuromorphic Computing and Engineering*, 2(3):034004, 2022.

[11] Deepak Kadetotad, Shihui Yin, Visar Berisha, Chaitali Chakrabarti, and Jae-sun Seo. An 8.93 TOPS/W LSTM recurrent neural network accelerator featuring hierarchical coarse-grain sparsity for on-device speech recognition. *IEEE Journal of Solid-State Circuits*, 55(7):1877–1887, 2020.

[12] Kwantae Kim, Chang Gao, Rui Graca, Ilya Kiselev, Hoi-Jun Yoo, Tobi Delbruck, and Shih-Chii Liu. A 23-μW keyword spotting IC with ring-oscillator-based time-domain feature extraction. *IEEE Journal of Solid-State Circuits*, 57(11):3298–3311, 2022.

[13] David Krueger, Tegan Maharaj, Janos Kramar, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Christopher Pal. Zoneout: Regularizing RNNs by randomly preserving hidden activations. In *International Conference on Learning Representations*, 2017.

[14] Johanna Hedlund Lindmar, Chang Gao, and Shih-Chii Liu. Intrinsic sparse LSTM using structured targeted dropout for efficient hardware inference. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 126–129, 2022.

[15] Loren Lugosch, Mirco Ravanelli, Patrick Ignoto, Vikrant Singh Tomar, and Yoshua Bengio. Speech model pre-training for end-to-end spoken language understanding. In *Proc. Interspeech 2019*, pp. 814–818, 2019. doi: 10.21437/Interspeech.2019-2396.

[16] Daniel Neil, Junhaeng Lee, Tobi Delbrück, and Shih-Chii Liu. Delta networks for optimized recurrent network computation. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, pp. 2584–2593, 2017.

[17] Peter O'Connor and Max Welling. Sigma delta quantized networks. In *International Conference on Learning Representations*, 2017.

[18] Nicolas Perez-Nieves and Dan F. M. Goodman. Sparse spiking gradient descent. In *Advances in Neural Information Processing Systems*, 2021.

[19] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. iCARL: Incremental classifier and representation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2001–2010, 2017.

[20] Weiwei Shan, Minhao Yang, Tao Wang, Yicheng Lu, Hao Cai, Lixuan Zhu, Jiaming Xu, Chengjun Wu, Longxing Shi, and Jun Yang. A 510-nW wake-up keyword-spotting chip using serial-FFT-based MFCC and binarized depthwise separable CNN in 28-nm CMOS. *IEEE Journal of Solid-State Circuits*, 56(1):151–164, 2020.

[21] Gaurav Srivastava, Deepak Kadetotad, Shihui Yin, Visar Berisha, Chaitali Chakrabarti, and Jae-Sun Seo. Joint optimization of quantization and structured sparsity for compressed deep neural networks. In *2019 Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1393–1397, 2019. doi: 10.1109/icassp.2019.8682791.

[22] Anand Subramoney, Khaleelulla Khan Nazeer, Mark Schöne, Christian Mayr, and David Kappel. Efficient recurrent architectures through activity sparsity and sparse back-propagation through time. In *The Eleventh International Conference on Learning Representations*, 2022.

[23] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

Appendix A Delta RNNs BPTT Formulations

This section shows the derivation of BPTT equations of Delta RNNs. First we show the proof of Proposition 1 in Section 2.2. Next we derive the BPTT formulation of Delta RNNs [16], then extend it to the two variants of RNNs: Gated Recurrent Unit (**GRU**) [4] and Long Short-Term Memory (**LSTM**) [9].

A.1 Proof of Proposition 1

In this subsection we give the proof of Proposition 1 in Section 2.2 for the gradient of weight W_x. Proof for the gradient of weight W_h is similar.

In a Delta RNN layer, the computation in the forward propagation phase is formulated as:

$$M_t = M_{x,t} + M_{h,t} + M_{t-1} \tag{9}$$

$$M_{x,t} = \boldsymbol{W_x} \Delta x_t \tag{10}$$

$$M_{h,t} = \boldsymbol{W_h} \Delta h_{t-1} \tag{11}$$

$$\hat{x}_{i,t} = \begin{cases} x_{i,t}, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ \hat{x}_{i,t-1}, & \text{otherwise} \end{cases} \tag{12}$$

$$\Delta x_{i,t} = \begin{cases} x_{i,t} - \hat{x}_{i,t-1}, & \text{if } |x_{i,t} - \hat{x}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

In the backward propagation phase, the cost function is the sum of the loss function at every timestep:

$$C = \sum_{t=1}^{T} L_t \tag{14}$$

Hidden states h_t is 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:

$$\frac{\partial C}{\partial W_x} = \frac{\partial (\sum_{t=1}^{T} L_t)}{\partial W_x} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_x} \tag{15}$$

For each timestep, according to the chain rule:

$$\frac{\partial L_t}{\partial W_x} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial W_x} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial M_t} \frac{\partial M_t}{\partial W_x} \tag{16}$$

From Eqs. (9)-(11) we can expand the last term:

$$\frac{\partial M_t}{\partial W_x} = \frac{\partial (W_x \Delta x_t + W_h \Delta h_{t-1} + M_{t-1})}{\partial W_x} = \Delta x_t^{\intercal} + \frac{\partial M_{t-1}}{\partial W_x} \tag{17}$$

Using telescoping we get

$$\frac{\partial M_t}{\partial W_x} = \Delta x_t^{\intercal} + \Delta x_{t-1}^{\intercal} + \cdots + \Delta x_1^{\intercal} = \sum_{i=1}^{t} \Delta x_i^{\intercal} \tag{18}$$

Then Eq. (16) becomes:

$$\frac{\partial L_t}{\partial W_x} = \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial M_t} (\sum_{i=1}^{t} \Delta x_i^{\intercal}) \tag{19}$$

For simplicity we write

$$\frac{\partial L_t}{\partial h_t} = L'(h_t), \qquad \frac{\partial h_t}{\partial M_t} = h'(M_t)$$

Plug them into Eq. (15), we get:

$$
\begin{aligned}
\frac{\partial C}{\partial W_x} &= \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_x} = \sum_{t=1}^{T} \left[L'(h_t) h'(M_t) (\sum_{i=1}^{t} \Delta x_i^{\intercal}) \right] \\
&= \underbrace{L'(h_1) h'(M_1) \Delta x_1^{\intercal}}_{t=1} + \underbrace{L'(h_2) h'(M_2) (\Delta x_1^{\intercal} + \Delta x_2^{\intercal})}_{t=2} + \cdots + \underbrace{L'(h_T) h'(M_T) (\sum_{i=1}^{T} \Delta x_i^{\intercal})}_{t=T} \\
&= \sum_{t=1}^{T} L'(h_t) h'(M_t) \Delta x_1^{\intercal} + \sum_{t=2}^{T} L'(h_t) h'(M_t) \Delta x_2^{\intercal} + \cdots + \sum_{t=T}^{T} L'(h_t) h'(M_t) \Delta x_T^{\intercal} \\
&= \sum_{t=1}^{T} \underbrace{\left[\underbrace{\left(\sum_{i=t}^{T} L'(h_i) h'(M_i) \right)}_{①} \Delta x_t^{\intercal} \right]}_{②}
\end{aligned}
\tag{20}
$$

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 (Section 2.2), 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. (20) is the sum of ② across all timesteps, thus proportional to the sparsity of Δx_t at each timestep.

A.2 Delta RNN Computation and Memory Costs Calculation

For the BP phase in training, the computation cost for each timestep in Eq. (7) is

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

for a weight matrix W_x of size $n \times n$ and a sparse Δx_t vector of occupancy o_c.

The memory access cost 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. (6) can be derived analogously.

Therefore, the costs of matrix operations in BP (Eqs. 6 and 7) are also reduced to:

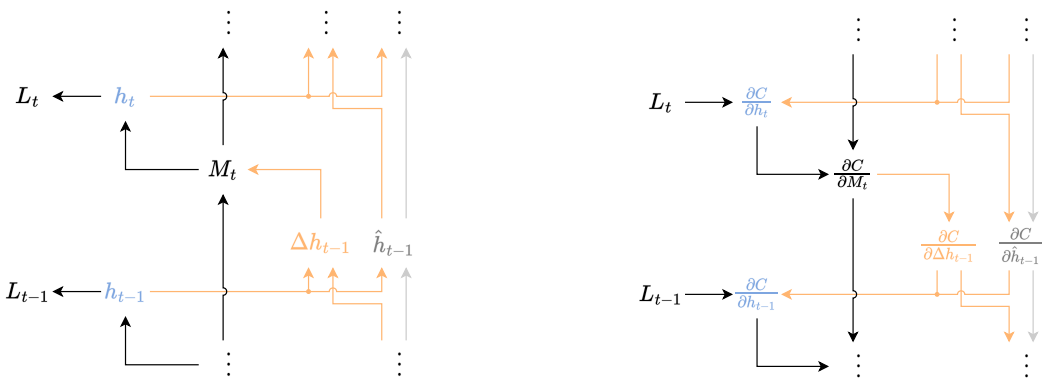
$$C_{\text{sparse}} / C_{\text{dense}} \approx (o_c \cdot n^2)/n^2 = o_c$$

The memory storage cost for the sparse DN training method is marginal. Each timestep in FP 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. Alternatively we may use an NZIL to memorize the indices of activated neurons at each timestep which occupies at most n words in the memory depending on the delta vector sparsity.

A.3 Delta RNN BPTT Formulation

The Forward Propagation (FP) equations of Delta RNNs [16] are shown in Section 2.1. The FP computation graph is shown in Fig. 4a. The lines in orange/gray color means that only values of activated/inactivated neurons are propagated along these paths. For clarity we only show the computation paths relevant to h_t. The other parts regarding x_t can be drawn in the similar way.

In BPTT, the loss function L_t is computed for each timestep, which sums up to the cost: $C = \sum_{t=1}^{T} L_t$. The BP computation paths are reversed (Fig. 4b) from the FP paths. The error is propagated backward through time, and the gradients are calculated along the paths according to the chain rule. The total gradient of a vertex in the graph is the sum of the gradients of all incoming paths.



(a) FP computation graph regarding h_t.

(b) BP computation graph regarding h_t.

Figure 4: Computation graphs of a Delta RNN layer. Weight matrices are omitted for clarity.

The pre-activation memory M_t has two incoming connections in the BP computation graph, one from M_{t+1} and the other from h_t. So the PD of M_t w.r.t. the cost C has two components:

$$\frac{\partial C}{\partial M_t} = \frac{\partial C}{\partial h_t}\frac{\partial h_t}{\partial M_t} + \frac{\partial C}{\partial M_{t+1}}\overline{\frac{\partial M_{t+1}}{\partial M_t}} = \frac{\partial C}{\partial h_t}\tanh'(M_t) + \frac{\partial C}{\partial M_{t+1}} \tag{21}$$

We use an overline to denote the part of a Partial Derivative (PD) that only takes the direct connection into account. For example, $\overline{\frac{\partial M_{t+1}}{\partial M_t}}$ is the PD coming from the path $M_{t+1} \to M_t$ directly, excluding any other path such as $M_{t+1} \to \Delta h_t \to h_t \to M_t$.

Similarly, we can derive the other gradients in the computation graph as follows.

$$\frac{\partial C}{\partial \Delta h_{t-1}} = \frac{\partial C}{\partial M_t}\frac{\partial M_t}{\partial \Delta h_{t-1}} = \boldsymbol{W}_{\boldsymbol{h}}^{\intercal}\frac{\partial C}{\partial M_t} \tag{22}$$

$$\frac{\partial C}{\partial \hat{h}_{t-1}} = \frac{\partial C}{\partial \hat{h}_t}\frac{\partial \hat{h}_t}{\partial \hat{h}_{t-1}} + \frac{\partial C}{\partial \Delta h_t}\frac{\partial \Delta h_t}{\partial \hat{h}_{t-1}} \tag{23}$$

$$\frac{\partial C}{\partial h_t} = \frac{\partial C}{\partial \hat{h}_t}\frac{\partial \hat{h}_t}{\partial h_t} + \frac{\partial C}{\partial \Delta h_t}\frac{\partial \Delta h_t}{\partial h_t} + \frac{\partial L_t}{\partial h_t} \tag{24}$$

$$\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_h}} = \sum_{t=1}^{T}\frac{\partial C}{\partial M_t}\frac{\partial M_t}{\partial W_h} = \sum_{t=1}^{T}\frac{\partial C}{\partial M_t}\Delta h_{t-1}^{\intercal} \tag{25}$$

The pre-activation memory is initialized as $M_0 = b_h$ in FP, so the gradient of the bias is $\frac{\partial C}{\partial b_h} = \frac{\partial C}{\partial M_0}$.

The equations for gradients w.r.t. x_t are similar, except that there is no direct connection between x_t and the loss L_t. So Eq. (24) can be modified for x_t as:

$$\frac{\partial C}{\partial x_t} = \frac{\partial C}{\partial \hat{x}_t}\frac{\partial \hat{x}_t}{\partial x_t} + \frac{\partial C}{\partial \Delta x_t}\frac{\partial \Delta x_t}{\partial x_t} \tag{26}$$

To compute the first two terms in Eqs. (23) and (24), we need to look back at the FP equations:

$$\hat{h}_{i,t} = \begin{cases} h_{i,t}, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ \hat{h}_{i,t-1}, & \text{otherwise} \end{cases} \tag{27}$$

$$\Delta h_{i,t} = \begin{cases} h_{i,t} - \hat{h}_{i,t-1}, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{28}$$

Eq. (27) has two branches, thus the PDs of $\hat{h}_t$ w.r.t. h_t and $\hat{h}_{t-1}$ also have two branches:

$$\frac{\partial \hat{h}_{i,t}}{\partial h_{i,t}} = \begin{cases} 1, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \qquad \frac{\partial \hat{h}_{i,t}}{\partial \hat{h}_{i,t-1}} = \begin{cases} 0, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ 1, & \text{otherwise} \end{cases} \tag{29}$$

From Eq. (28) the PDs of Δh_t w.r.t. h_t and $\hat{h}_{t-1}$ are:

$$\frac{\partial \Delta h_{i,t}}{\partial h_{i,t}} = \begin{cases} 1, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \qquad \frac{\partial \Delta h_{i,t}}{\partial \hat{h}_{i,t-1}} = \begin{cases} -1, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{30}$$

For convenience we use a mask vector $m_{h,t}$ to memorize whether a neuron is activated during FP:

$$m_{h,i,t} = \begin{cases} 1, & \text{if } |h_{i,t} - \hat{h}_{i,t-1}| > \Theta \\ 0, & \text{otherwise} \end{cases} \tag{31}$$

Then Eqs. (27) and (28) can be rewritten as:

$$\hat{h}_t = h_t \odot m_{h,t} + \hat{h}_{t-1} \odot (1 - m_{h,t}) \tag{32}$$

$$\Delta h_t = (h_t - \hat{h}_{t-1}) \odot m_{h,t} \tag{33}$$

where $\odot$ denotes the element-wise multiplication.

And the PDs in Eqs. (29)- (30) can be rewritten as:

$$\frac{\partial \hat{h}_t}{\partial h_t} = m_{h,t} \qquad \frac{\partial \hat{h}_t}{\partial \hat{h}_{t-1}} = 1 - m_{h,t}$$

$$\frac{\partial \Delta h_t}{\partial h_t} = m_{h,t} \qquad \frac{\partial \Delta h_t}{\partial \hat{h}_{t-1}} = -m_{h,t}$$

Plug them into Eqs. (23) and (24), we have:

$$\frac{\partial C}{\partial \hat{h}_{t-1}} = \frac{\partial C}{\partial \hat{h}_t} \odot (1 - m_{h,t}) - \frac{\partial C}{\partial \Delta h_t} \odot m_{h,t} \tag{34}$$

$$\frac{\partial C}{\partial h_t} = \frac{\partial C}{\partial \hat{h}_t} \odot m_{h,t} + \frac{\partial C}{\partial \Delta h_t} \odot m_{h,t} + \frac{\partial L_t}{\partial h_t} \tag{35}$$

The term $\frac{\partial C}{\partial \Delta h_t}$ is multiplied with the mask $m_{h,t}$ for all the equations relevant to it. In other words, only values at the indices of those neurons who are activated during FP are used in BP. So we only need to calculate the values in $\frac{\partial C}{\partial \Delta h_t}$ for activated neurons in Eq. (22) and discard other values, just like we discard the changes of inactivated neurons Δh_t in FP. In fact, Eq. (22) can be calculated as:

$$\frac{\partial C}{\partial \Delta h_{t-1}} = \left(\boldsymbol{W}_{\boldsymbol{h}}^{\mathsf{T}} \frac{\partial C}{\partial M_t} \right) \odot m_{h,t-1} \tag{36}$$

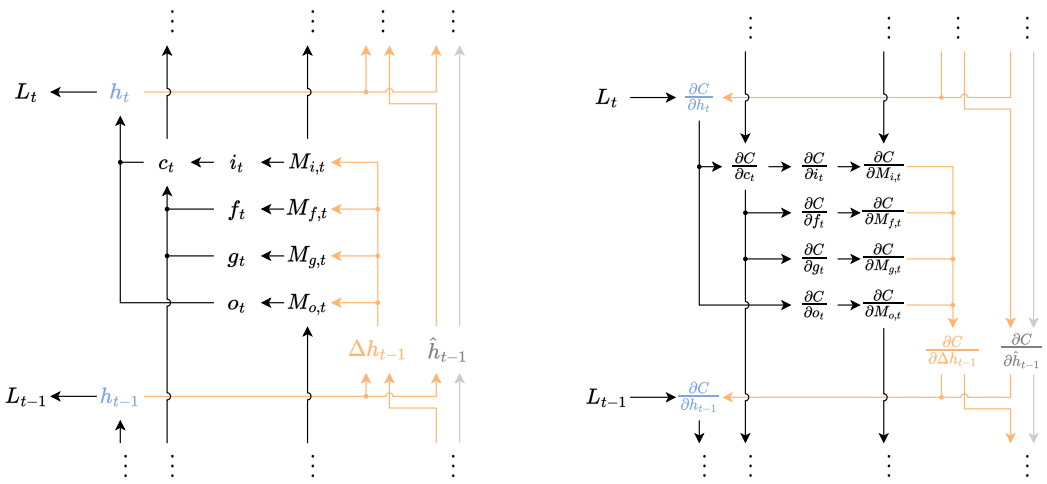
which would give exactly the same results Eqs. (34) and (35) and other equations in BPTT.

This can also be seen from Eq. (28). Δh_t is a constant 0 at the below-threshold branch which is not differentiable, so the PDs of Δh_t w.r.t. h_t and $\hat{h}_{t-1}$ are 0 at this branch. According to the chain rule, any gradients of Δh_t passing through this branch will be multiplied by these PDs, effectively zeroing out those gradients. Consequently, we only need the Δh_t gradients for the above-threshold branch, which can be expressed by Eq. (36).

Therefore, once the delta threshold Θ is introduced in FP, the accuracy of the network may decrease due to the loss of some information, but the temporal sparsity created in Δx and Δh during FP can be leveraged in BP without causing any extra accuracy loss.

The main computations regarding the hidden state vector h_t in BP are the vector outer products in Eq. (25) and the MxVs in Eq. (36), both of them sparsified by the delta mask $m_{h,t}$. The gradients regarding the input vector x_t can be derived in a similar way, which would also results in sparse MxVs with the mask $m_{x,t}$.

In conclusion, in the BPTT phase of Delta RNNs, we only need to propagate the errors of the changes of activated neurons, and calculate the weight gradients of the corresponding connections. The three MxV operations during the training of Delta RNNs, which are expressed by Eqs. (11), (25), and (36), share the same sparsity (Figs. 1d). A large number of operations can be reduced by skipping computations of inactivated neurons in both forward and backward propagation using binary mask vectors or NZILs.



(a) FP computation graph regarding h_t.

(b) BP computation graph regarding h_t.

Figure 5: Computation graphs of a Delta LSTMs layer.

A.4 Delta LSTM BPTT Formulation

Figs. 5a and 5b show the FP and BP computation graphs pertaining to the hidden state h_t of a Delta LSTM layer. The delta updating rules described by Eqs. (27) and (28) in Section A.3 are also applied to the Delta LSTM. The pre-activation memory vector M_t consists of four components that correspond to the four gates of LSTM: input gate i_t, forget gate f_t, input modulation gate g_t, and output gate o_t. It also has a memory cell state c_t.

The FP equations of the Delta LSTM are shown below.

$$M_{i,t} = \boldsymbol{W_{xi}}\Delta x_t + \boldsymbol{W_{hi}}\Delta h_{t-1} + M_{i,t-1} \tag{37}$$

$$M_{f,t} = \boldsymbol{W_{xf}}\Delta x_t + \boldsymbol{W_{hf}}\Delta h_{t-1} + M_{f,t-1} \tag{38}$$

$$M_{g,t} = \boldsymbol{W_{xg}}\Delta x_t + \boldsymbol{W_{hg}}\Delta h_{t-1} + M_{g,t-1} \tag{39}$$

$$M_{o,t} = \boldsymbol{W_{xo}}\Delta x_t + \boldsymbol{W_{ho}}\Delta h_{t-1} + M_{o,t-1} \tag{40}$$

$$i_t = \sigma(M_{i,t}) \tag{41}$$

$$f_t = \sigma(M_{f,t}) \tag{42}$$

$$g_t = \tanh(M_{g,t}) \tag{43}$$

$$o_t = \sigma(M_{o,t}) \tag{44}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{45}$$

$$h_t = o_t \odot \tanh(c_t) \tag{46}$$

Eqs. (37)-(40) can also be expressed by Eqs. (9)-(11) if we concatenate the weight matrices and the pre-activation memory vectors of the four gates. The memory vectors are initialized to the corresponding biases: $M_{i,0} = b_i$, $M_{f,0} = b_f$, $M_{g,0} = b_g$, $M_{o,0} = b_o$.

The BP equations are derived according to the computation graph (Fig. 5b).

$$\frac{\partial C}{\partial c_t} = \frac{\partial C}{\partial h_t}\frac{\partial h_t}{\partial c_t} + \frac{\partial C}{\partial c_{t+1}}\overline{\frac{\partial c_{t+1}}{\partial c_t}} = \frac{\partial C}{\partial h_t} \odot o_t \odot \tanh'(c_t) + \frac{\partial C}{\partial c_{t+1}} \tag{47}$$

$$\frac{\partial C}{\partial M_{i,t}} = \frac{\partial C}{\partial c_t}\frac{\partial c_t}{\partial i_t}\frac{\partial i_t}{\partial M_{i,t}} + \frac{\partial C}{\partial M_{i,t+1}}\overline{\frac{\partial M_{i,t+1}}{\partial M_{i,t}}} = \frac{\partial C}{\partial c_t} \odot g_t \odot \sigma'(M_{i,t}) + \frac{\partial C}{\partial M_{i,t+1}} \tag{48}$$

$$\frac{\partial C}{\partial M_{f,t}} = \frac{\partial C}{\partial c_t}\frac{\partial c_t}{\partial f_t}\frac{\partial f_t}{\partial M_{f,t}} + \frac{\partial C}{\partial M_{f,t+1}}\overline{\frac{\partial M_{f,t+1}}{\partial M_{f,t}}} = \frac{\partial C}{\partial c_t} \odot c_{t-1}\sigma'(M_{f,t}) + \frac{\partial C}{\partial M_{f,t+1}} \tag{49}$$

$$\frac{\partial C}{\partial M_{g,t}} = \frac{\partial C}{\partial c_t}\frac{\partial c_t}{\partial g_t}\frac{\partial g_t}{\partial M_{g,t}} + \frac{\partial C}{\partial M_{g,t+1}}\overline{\frac{\partial M_{g,t+1}}{\partial M_{g,t}}} = \frac{\partial C}{\partial c_t} \odot i_t \odot \tanh'(M_{g,t}) + \frac{\partial C}{\partial M_{g,t+1}} \tag{50}$$

$$\frac{\partial C}{\partial M_{o,t}} = \frac{\partial C}{\partial h_t}\frac{\partial h_t}{\partial o_t}\frac{\partial o_t}{\partial M_{o,t}} + \frac{\partial C}{\partial M_{o,t+1}}\overline{\frac{\partial M_{o,t+1}}{\partial M_{o,t}}} = \frac{\partial C}{\partial h_t} \odot \tanh(c_t) \odot \sigma'(M_{o,t}) + \frac{\partial C}{\partial M_{o,t+1}} \tag{51}$$

The term with an overline denotes the part of a partial derivative that only takes the direct connection into account. For example, $\overline{\frac{\partial c_{t+1}}{\partial c_t}}$ is the PD that comes from the path $c_{t+1} \rightarrow c_t$ directly, excluding any other paths such as $c_{t+1} \rightarrow i_{t+1} \rightarrow M_{i,t+1} \rightarrow \Delta h_t \rightarrow h_t \rightarrow c_t$.

The PD of the cost w.r.t. Δh_{t-1} is:

$$\frac{\partial C}{\partial \Delta h_{t-1}} = \frac{\partial C}{\partial M_{i,t}}\frac{\partial M_{i,t}}{\partial \Delta h_{t-1}} + \frac{\partial C}{\partial M_{f,t}}\frac{\partial M_{f,t}}{\partial \Delta h_{t-1}} + \frac{\partial C}{\partial M_{h,t}}\frac{\partial M_{g,t}}{\partial \Delta h_{t-1}} + \frac{\partial C}{\partial M_{o,t}}\frac{\partial M_{o,t}}{\partial \Delta h_{t-1}} \tag{52}$$

The gradient vectors for the pre-activation memories of the four gates can also be concatenated into a longer vector $\frac{\partial C}{\partial M_t}$. Then Eq. (52) becomes:

$$\frac{\partial C}{\partial \Delta h_{t-1}} = \frac{\partial C}{\partial M_t}\frac{\partial M_t}{\partial \Delta h_{t-1}} = \boldsymbol{W_h^\intercal}\frac{\partial C}{\partial M_t} \tag{53}$$

The PDs of the cost w.r.t. $\hat{h}_{t-1}$ and h_t are:

$$\frac{\partial C}{\partial \hat{h}_{t-1}} = \frac{\partial C}{\partial \hat{h}_t}\frac{\partial \hat{h}_t}{\partial \hat{h}_{t-1}} + \frac{\partial C}{\partial \Delta h_t}\frac{\partial \Delta h_t}{\partial \hat{h}_{t-1}} = \frac{\partial C}{\partial \hat{h}_t} \odot (1 - m_{h,t}) - \frac{\partial C}{\partial \Delta h_t} \odot m_{h,t} \tag{54}$$

$$\frac{\partial C}{\partial h_t} = \frac{\partial C}{\partial \hat{h}_t}\frac{\partial \hat{h}_t}{\partial h_t} + \frac{\partial C}{\partial \Delta h_t}\frac{\partial \Delta h_t}{\partial h_t} + \frac{\partial L_t}{\partial h_t} = \frac{\partial C}{\partial \hat{h}_t} \odot m_{h,t} + \frac{\partial C}{\partial \Delta h_t} \odot m_{h,t} + \frac{\partial L_t}{\partial h_t} \tag{55}$$

The gradients of weight matrices obtained at the t-th timestep are given by:

$$\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hi,t}}} = \frac{\partial C}{\partial M_{i,t}}\frac{\partial M_{i,t}}{\partial W_{hi,t}} \qquad \frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hf,t}}} = \frac{\partial C}{\partial M_{f,t}}\frac{\partial M_{f,t}}{\partial W_{hf,t}}$$
$$\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hg,t}}} = \frac{\partial C}{\partial M_{g,t}}\frac{\partial M_{g,t}}{\partial W_{hg,t}} \qquad \frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{ho,t}}} = \frac{\partial C}{\partial M_{o,t}}\frac{\partial M_{o,t}}{\partial W_{ho,t}}$$

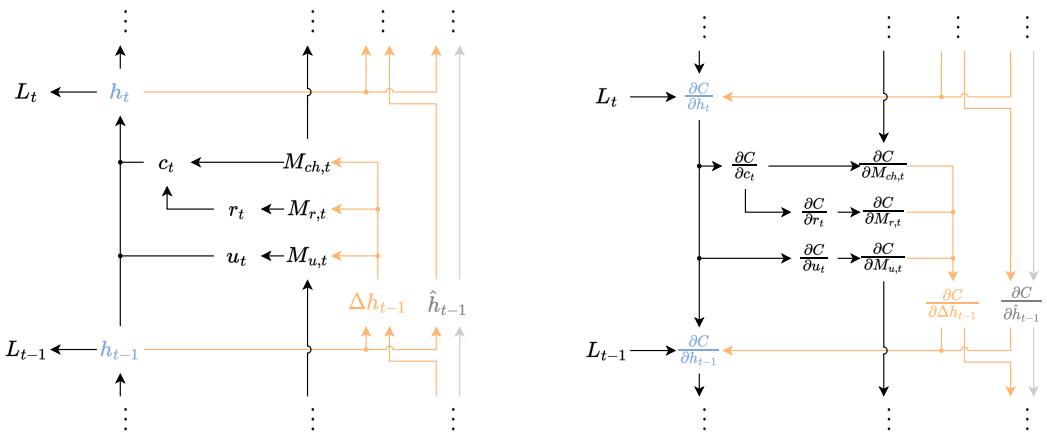
The gradients of biases are:

$$\frac{\partial C}{\partial b_i} = \frac{\partial C}{\partial M_{i,0}} \qquad \frac{\partial C}{\partial b_f} = \frac{\partial C}{\partial M_{f,0}}$$
$$\frac{\partial C}{\partial b_g} = \frac{\partial C}{\partial M_{g,0}} \qquad \frac{\partial C}{\partial b_o} = \frac{\partial C}{\partial M_{o,0}}$$

By concatenating the four weight gradient matrices into $\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_h}}$, we have:

$$\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_h}} = \sum_{t=1}^{T}\frac{\partial C}{\partial M_t}\frac{\partial M_t}{\partial W_h} = \sum_{t=1}^{T}\frac{\partial C}{\partial M_t}\Delta h_{t-1}^\intercal \tag{56}$$

It can be seen that Eqs. (53)-(56) are the same as Eqs. (22)-(25). Only the values at the indices of activated neurons in $\frac{\partial C}{\partial \Delta h_{t-1}}$ are used in Eqs. (54) and (55), so Eq. (53) can be made sparse using the mask vector $m_{h,t}$. Eq. (56) is also sparse since the term $\frac{\partial C}{\partial M_t}$ is multiplied with the sparse vector Δh_{t-1}. The sparsity in the gradient computations for the input x can be obtained in the same way. Therefore, the temporal sparsity created in Delta LSTMs during FP also exist in the matrix multiplications during BP as shown in Eqs. (53) and (56). Other claims about Delta RNNs in Section A.3 are also true for Delta LSTMs.



(a) FP computation graph regarding h_t.

(b) BP computation graph regarding h_t.

Figure 6: Computation graphs of a Delta GRU layer.

A.5 Delta GRU BPTT Formulation

The GRU has a reset gate r, an update gate u, and a candidate activation vector c. In Delta GRU we apply the delta updating rules described by Eqs. (27) and (28) in Section A.3 to the input x and the hidden state h. Figs. 6a and 6b show the FP and BP computation graphs relevant to the hidden state h_t of a Delta GRU layer.

The FP equations of the Delta GRU are shown below.

$$M_{r,t} = \boldsymbol{W_{xr}}\Delta x_t + \boldsymbol{W_{hr}}\Delta h_{t-1} + M_{r,t-1} \tag{57}$$

$$M_{u,t} = \boldsymbol{W_{xu}}\Delta x_t + \boldsymbol{W_{hu}}\Delta h_{t-1} + M_{u,t-1} \tag{58}$$

$$M_{cx,t} = \boldsymbol{W_{xc}}\Delta x_t + M_{cx,t-1} \tag{59}$$

$$M_{ch,t} = \boldsymbol{W_{hc}}\Delta h_{t-1} + M_{ch,t-1} \tag{60}$$

$$M_{c,t} = M_{cx,t} + r_t \odot M_{ch,t} \tag{61}$$

$$r_t = \sigma(M_{r,t}) \tag{62}$$

$$u_t = \sigma(M_{u,t}) \tag{63}$$

$$c_t = \tanh(M_{c,t}) \tag{64}$$

$$h_t = (1 - u_t) \odot h_{t-1} + u_t \odot c_t \tag{65}$$

The pre-activation memory vectors are initialized to their corresponding biases: $M_{r,0} = b_r$, $M_{u,0} = b_u$, $M_{cx,0} = b_c$, $M_{ch,0} = 0$.

The BP equations are derived according to the computation graph (Fig. 6b).

$$\frac{\partial C}{\partial c_t} = \frac{\partial C}{\partial h_t}\frac{\partial h_t}{\partial c_t} = \frac{\partial C}{\partial h_t} \odot u_t \tag{66}$$

$$\frac{\partial C}{\partial M_{c,t}} = \frac{\partial C}{\partial c_t}\frac{\partial c_t}{\partial M_{c,t}} = \frac{\partial C}{\partial c_t} \odot \tanh'(M_{c,t}) \tag{67}$$

$$\frac{\partial C}{\partial M_{ch,t}} = \frac{\partial C}{\partial M_{c,t}}\frac{\partial M_{c,t}}{\partial M_{ch,t}} + \frac{\partial C}{\partial M_{ch,t+1}}\overline{\frac{\partial M_{ch,t+1}}{\partial M_{ch,t}}} = \frac{\partial C}{\partial M_{c,t}} \odot r_t + \frac{\partial C}{\partial M_{ch,t+1}} \tag{68}$$

$$\frac{\partial C}{\partial M_{r,t}} = \frac{\partial C}{\partial M_{c,t}}\frac{\partial M_{c,t}}{\partial r_t} + \frac{\partial C}{\partial M_{r,t+1}}\overline{\frac{\partial M_{r,t+1}}{\partial M_{r,t}}} = \frac{\partial C}{\partial M_{c,t}} \odot M_{ch,t} + \frac{\partial C}{\partial M_{r,t+1}} \tag{69}$$

$$\frac{\partial C}{\partial M_{u,t}} = \frac{\partial C}{\partial h_t}\frac{\partial h_t}{\partial u_t}\frac{\partial u_t}{\partial M_{u,t}} + \frac{\partial C}{\partial M_{u,t+1}}\overline{\frac{\partial M_{u,t+1}}{\partial M_{u,t}}} = \frac{\partial C}{\partial h_t} \odot (c_t - h_{t-1}) \odot \sigma'(M_{u,t}) + \frac{\partial C}{\partial M_{u,t+1}} \tag{70}$$

The PD of the cost w.r.t. Δh_{t-1} is:

$$\frac{\partial C}{\partial \Delta h_{t-1}} = \frac{\partial C}{\partial M_{u,t}} \frac{\partial M_{u,t}}{\partial \Delta h_{t-1}} + \frac{\partial C}{\partial M_{r,t}} \frac{\partial M_{r,t}}{\partial \Delta h_{t-1}} + \frac{\partial C}{\partial M_{ch,t}} \frac{\partial M_{ch,t}}{\partial \Delta h_{t-1}} \tag{71}$$

The gradient vectors $\frac{\partial C}{\partial M_{r,t}}$, $\frac{\partial C}{\partial M_{u,t}}$, and $\frac{\partial C}{\partial M_{ch,t}}$ can be concatenated into a longer vector $\frac{\partial C}{\partial M_{h,t}}$. Their weight matrices $\boldsymbol{W_{hr}}$, $\boldsymbol{W_{hu}}$, and $\boldsymbol{W_{hc}}$ can also be concatenated as $\boldsymbol{W_h^\intercal}$. Then Eq. (71) becomes:

$$\frac{\partial C}{\partial \Delta h_{t-1}} = \frac{\partial C}{\partial M_{h,t}} \frac{\partial M_{h,t}}{\partial \Delta h_{t-1}} = \boldsymbol{W_h^\intercal} \frac{\partial C}{\partial M_{h,t}} \tag{72}$$

The PDs of the cost w.r.t. $\hat{h}_{t-1}$ and h_t are:

$$\frac{\partial C}{\partial \hat{h}_{t-1}} = \frac{\partial C}{\partial \hat{h}_t} \frac{\partial \hat{h}_t}{\partial \hat{h}_{t-1}} + \frac{\partial C}{\partial \Delta h_t} \frac{\partial \Delta h_t}{\partial \hat{h}_{t-1}} = \frac{\partial C}{\partial \hat{h}_t} \odot (1 - m_{h,t}) - \frac{\partial C}{\partial \Delta h_t} \odot m_{h,t} \tag{73}$$

$$\begin{aligned} \frac{\partial C}{\partial h_t} &= \frac{\partial C}{\partial \hat{h}_t} \frac{\partial \hat{h}_t}{\partial h_t} + \frac{\partial C}{\partial \Delta h_t} \frac{\partial \Delta h_t}{\partial h_t} + \frac{\partial C}{\partial h_{t+1}} \overline{\frac{\partial h_{t+1}}{\partial h_t}} + \frac{\partial L_t}{\partial h_t} \\ &= \frac{\partial C}{\partial \hat{h}_t} \odot m_{h,t} + \frac{\partial C}{\partial \Delta h_t} \odot m_{h,t} + \frac{\partial C}{\partial h_{t+1}} \odot (1 - u_{t+1}) + \frac{\partial L_t}{\partial h_t} \end{aligned} \tag{74}$$

The gradients of weight matrices obtained at the t-th timestep are given by:

$$\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hr,t}}} = \frac{\partial C}{\partial M_{r,t}} \frac{\partial M_{r,t}}{\partial W_{hr,t}} \qquad \frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hu,t}}} = \frac{\partial C}{\partial M_{u,t}} \frac{\partial M_{u,t}}{\partial W_{hu,t}} \qquad \frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hc,t}}} = \frac{\partial C}{\partial M_{ch,t}} \frac{\partial M_{ch,t}}{\partial W_{hc,t}}$$

The gradients of biases are:

$$\frac{\partial C}{\partial b_r} = \frac{\partial C}{\partial M_{r,0}} \qquad \frac{\partial C}{\partial b_u} = \frac{\partial C}{\partial M_{u,0}} \qquad \frac{\partial C}{\partial b_c} = \frac{\partial C}{\partial M_{cx,0}}$$

By concatenating the weight gradient matrices $\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hr}}}$, $\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hu}}}$, and $\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_{hc}}}$ into $\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_h}}$, we have:

$$\frac{\boldsymbol{\partial C}}{\boldsymbol{\partial W_h}} = \sum_{t=1}^{T} \frac{\partial C}{\partial M_{h,t}} \frac{\partial M_{h,t}}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial C}{\partial M_{h,t}} \Delta h_{t-1}^\intercal \tag{75}$$

It can be seen that Eqs. (72)-(75) are essentially the same as Eqs. (22)-(25), except that there is an additional term in Eq. (74) derived from the self-recursion of h_t in Eq. (65). But the values at the indices of inactivated neurons in $\frac{\partial C}{\partial \Delta h_{t-1}}$ are still unused in BP, so Eq. (72) can still be sparsified with the mask vector $m_{h,t}$. Eq. (75) is also sparse as the term $\frac{\partial C}{\partial M_{h,t}}$ is multiplied with the sparse vector Δh_{t-1}. The sparsity in the gradient computations for the input x can be obtained in the same manner. Therefore, the temporal sparsity created in Delta GRUs during FP also exist in the matrix multiplications during BP as shown in Eqs. (72) and (75). Other claims about Delta RNNs in Section A.3 are also true for Delta GRUs.

Appendix B Experiment Details

This section provides more details about the experiments in Section 3.

B.1 Delta RNN Training Accelerator

B.1.1 Computation Flow

The training accelerator described in Section 3.3 stores sparse delta vectors in its Static Random Access Memory (**SRAM**) in the format of NZIL and NZVL [5], which enables the accelerator to skip computations and memory accesses of entire weight columns for zero elements in delta vectors. This zero-skipping technique can efficiently exploit the temporal sparsity in Delta RNNs (Fig. 1d).

In the FP equation (2), 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 Dynamic Random Access Memory (**DRAM**), then multiply it with the Nonzero Values (**NZV**) of Δx_t.

In BP, for Eq. (6) 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.

For Eq. (7), 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 weight gradients are accumulated for all timesteps.

The same operations are performed for hidden states Δh.

The accelerator processes input samples one by one, i.e., the batch size of training is 1, so it can skip both the DRAM access of weight columns of inactivated neurons and the computation for them. 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.

B.1.2 Measurements in Simulation

In the simulation, we measure the computation time T_{measured} 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. The speedup factor is calculated as $F_{\text{speedup}} = T_{\text{dense}}/T_{\text{measured}}$, where T_{dense} is the theoretical computation time for a dense MxV equation of the same size in an ideal case. More specifically, assuming that there is no memory latency or communication overhead, and PEs are fully utilized, the computation time (in clock cycles) for a dense MxV is: $T_{\text{dense}} = N_l * N_{l-1} * T_s/P$ 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.