Revisit Long Short-Term Memory: An Optimization Perspective

Qi Lyu, Jun Zhu

State Key Laboratory of Intelligent Technology and Systems (LITS)
Tsinghua National Laboratory for Information Science and Technology (TNList)
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
qilyu.pub@gmail.com, dcszj@mail.tsinghua.edu.cn

Abstract

Long Short-Term Memory (LSTM) is a deep recurrent neural network architecture with high computational complexity. Contrary to the standard practice to train LSTM online with stochastic gradient descent (SGD) methods, we propose a matrix-based batch learning method for LSTM with full Backpropagation Through Time (BPTT). We further solve the state drifting issues as well as improving the overall performance for LSTM using revised activation functions for gates. With these changes, advanced optimization algorithms are applied to LSTM with long time dependency for the first time and show great advantages over SGD methods. We further demonstrate that large-scale LSTM training can be greatly accelerated with parallel computation architectures like CUDA and MapReduce.

1 Introduction

Long Short-Term Memory (LSTM) [1] is a deep recurrent neural network (RNN) well-suited to learn from experiences to classify, process and predict time series when there are very long time lags of unknown size between important events. LSTM consists of LSTM blocks instead of (or in addition to) regular network units. A LSTM block may be described as a "smart" network unit that can remember a value for an arbitrary length of time. This is one of the main reasons why LSTM outperforms other sequence learning methods in numerous applications such as achieving the best known performance in speech recognition [2], and winning the ICDAR handwriting competition in 2009. However, since each LSTM block has multiple nodes, with recurrent and cross connections between them, LSTM with many layers of blocks (and/or many blocks in one layer) will be extremely slow to train. This problem becomes more serious when trained with long sequences.

While the architecture of LSTM is continually evolving, the training methods have remained largely the same, i.e., the stochastic gradient descent (SGD) methods [1]. The momentum method is naturally incorporated into SGD methods later and remains the standard training regime for LSTM till now. Batch learning methods, which are popular in machine learning, are considered inapplicable for LSTM [3]. Despite the ease of implementation, SGD methods have several disadvantages that might limit the further improvements of training speed. One key disadvantage is that they require much manual tuning of hyper-parameters such as learning rate and convergence criteria. The second weakness is that unlike batch learning, SGD methods have little room left for serious optimization because the noisy nature of one sample per iteration renders the result unreliable to be utilized for further optimization. Most well-developed optimization algorithms such as Limited memory BFGS (L-BFGS) and Conjugate gradient (CG) methods can only work with batch learning. The third

---

[1]We stick to the notion of one sample at a time for stochastic gradient descent (SGD) methods, and use batch gradient descent methods to refer to others (including mini-batch gradient descent methods).
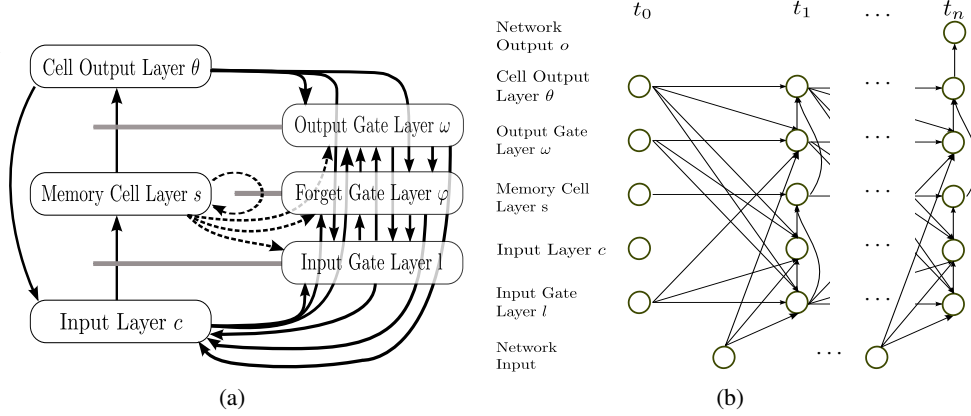
Figure 1: (a)An illustration of the layer-wise architecture of a LSTM block, where the dense lines denote cross connections between blocks in one layer; and the dashed lines denote peephole connections within a block.; (b)An illustration of the error flow through time in LSTM (forget gate omitted).

weakness is that SGD methods are inherently sequential: It is very difficult to parallelize them using GPUs or distribute them with a computer cluster.

In this paper, we propose to train LSTM with full Backpropagation Through Time (BPTT) under a matrix-based batch learning paradigm. Experiments show that our method enjoys faster convergence rate and better convergence performance than SGD methods. We further solve the state drifting issues as well as improving the overall performance for LSTM using revised activation functions for gates. With these changes, we make advanced optimization algorithms applicable to LSTM with long time dependency. Finally, our experiments with GPU and computer cluster demonstrate that large-scale LSTM training can be greatly accelerated with parallel computation architectures like CUDA and MapReduce using our matrix-based batch learning method.

2   LSTM Architecture

LSTM is a recurrent neural network (RNN) that can store and retrieve information from input streams and do complicated tasks like classifying, processing and predicting time series when there are very long time lags of an unknown size between important events. As illustrated in Fig. 1(a), LSTM is enabled to do so by utilizing memory cells that use logistic and linear units with multiplicative interactions with input and output gates. The information gets into (out of) the memory cell whenever a logistic input (output) gate is turned on. The memory cell state is kept from irrelevant information by keeping the input gate off, thus LSTM can have long term memory compared with simple recurrent networks without such kind of design.

Originally the gate is a logistic neuron that is activated by input units, memory cell outputs, gate units, or even conventional hidden units if there are any. With the extension of peephole connections, which are the direct connections from memory cell states to gates, the performance of LSTM is generally improved [4]. The other extension of forget gates is crafted to address the state drifting issues caused by inputs with a long time span. Because when inputs have a very long time span, the memory cell state often tends to grow (from initial status) linearly due to the leakage of information through a half-closed input gate. And in situations where the inputs are continuous streams, the memory cell states may grow in an unbounded fashion, leading to a saturation of the cell output squashing function, which apparently makes the memory cell defunct. This could be avoided if the memory cell state is controlled by a forget gate that will reset the cell state when it is turned off. Nevertheless we find this state drifting problem could be solved with a *more closed* input gate in Sec. 4. The true benefit of forget gates is resetting the network to do continuous prediction, so that the memory cells can keep the information when it is useful and forget the information when it is useless to leave space for new information to come in.

3 Batch Learning Methods for LSTM

There are two types of gradient-based techniques for LSTM training: truncated BPTT [1] and full BPTT [5]. Although they both can be applied to LSTM training with various architecture modifications in general, they are different in many ways. We analyze their differences in detail and then present our proposal of using full BPTT for LSTM under a batch learning paradigm.

3.1 Activation Dynamics

For clarity, we assume the LSTM network is densely connected without peephole connections or forget gates, and each block has only one memory cell (it is straightforward to generalize to other configurations). Vectors and matrices are in bold fonts. We will consider the LSTM blocks with five layers, including layer of the inputs (c), the input gates (l), the memory cells (s), the output gates (ω) and the cell outputs (θ). We will denote the final output layer by o. n is the imaginary layer consisting of the nodes that have direct connections with the nodes in question between time steps. For example, for layer l, n consists of l, ω, θ of the last time step, and input units, shown in the connections in Fig. 1(b) (one can get n for other layers similarly). $\boldsymbol{w}^{ab}$ denote the connection weights (in a matrix form) from layer b to layer a. For layer λ, $\boldsymbol{x}^\lambda$ are its inputs, f^λ is its activation function and $\boldsymbol{y}^\lambda$ are its outputs. $\boldsymbol{s}$ denote the statuses of the memory cell blocks. Unless stated otherwise, all network inputs, activations and partial derivatives are evaluated at time t, e.g. $\boldsymbol{y} \equiv \boldsymbol{y}_t$. The activation dynamics of an LSTM model at time step $t \in \{t_0, \cdots, t_n\}$ can be summarized as:

$$\boldsymbol{x}^\lambda = \boldsymbol{w}^{\lambda n} \boldsymbol{y}^n_{t-1}, \quad for \ \lambda \in \{l, c, \omega, o\} \tag{1}$$

$$\boldsymbol{y}^\lambda = f^\lambda(\boldsymbol{x}^\lambda_{t-1}), \quad for \ \lambda \in \{l, c, \omega, o\}, \tag{2}$$

$$\boldsymbol{s} = \boldsymbol{s}_{t-1} + \boldsymbol{y}^l \circ \boldsymbol{y}^c, \tag{3}$$

$$\boldsymbol{y}^\theta = \boldsymbol{y}^\omega \circ f^s(\boldsymbol{s}), \tag{4}$$

where $\circ$ denotes Hadamard product (entrywise product) and $f(\boldsymbol{x})$ is a vector of the function values when applying f to each element of the vector $\boldsymbol{x}$. We can see that Eq.s (1) and (2) are similar to the standard computations in feedforward neural networks, while the activation dynamics for memory cell statuses and outputs are very different (Eq.s (3) and (4)). Specifically, the inputs to each memory cell block are multiplied with the according input gates for that block, and the final output of a cell block is the activation of the cell status multiplied by the output gate.

3.2 Learning Rules for Truncated BPTT

In truncated BPTT, the number of time steps considered for backpropagation is limited to a fixed number. For LSTM particularly, the number is 1. So errors arriving at input layer of memory blocks and their gates do not get propagated back further in time, although they do serve to change the incoming weights. Despite the truncation of a small fraction of errors, the temporal error information could be propagated forward by hybriding truncated BPTT with Real Time Recurrent Learning (RTRL) [6] for memory cells, as follows:

$$\frac{\partial \boldsymbol{s}}{\partial \boldsymbol{w}^{cn}} = \frac{\partial \boldsymbol{s}_{t-1}}{\partial \boldsymbol{w}^{cn}} + \left(\frac{\partial \boldsymbol{y}^c}{\partial \boldsymbol{x}^c} \circ \boldsymbol{y}^l\right)^\top \boldsymbol{y}^n, \tag{5}$$

$$\frac{\partial \boldsymbol{s}}{\partial \boldsymbol{w}^{ln}} = \frac{\partial \boldsymbol{s}_{t-1}}{\partial \boldsymbol{w}^{ln}} + \left(\frac{\partial \boldsymbol{y}^l}{\partial \boldsymbol{x}^l} \circ \boldsymbol{y}^c\right)^\top \boldsymbol{y}^n. \tag{6}$$

Eq.s (5) and (6) should be calculated at every time step continually. They are essentially a measure of the sensitivity of unit s at time t to a small change in the value of weights, taking into account the effect of such a change in the weight over the entire network trajectory from t_0 to t. When the target signal $\boldsymbol{O}$ arrive, the error E (we assume mean squared error $||\boldsymbol{y}^o - \boldsymbol{O}||^2/2$) will then be calculated and back propagated through the network to get the following error responsibilities δ for each layer:

$$\boldsymbol{\delta}^o = (\boldsymbol{y}^o - \boldsymbol{O}) \circ (f^o)'(\boldsymbol{x}^o) \tag{7}$$

$$\boldsymbol{\delta}^\theta = (\boldsymbol{w}^{o\theta})^T \boldsymbol{\delta}^o \tag{8}$$

$$\boldsymbol{\delta}^\omega = (f^\omega)'(\boldsymbol{x}^\omega) \circ f^s(\boldsymbol{s}) \circ \boldsymbol{\delta}^\theta \tag{9}$$

$$\boldsymbol{\delta}^\lambda = (f^s)'(\boldsymbol{s}) \circ \boldsymbol{y}^\omega \circ \boldsymbol{\delta}^\theta \ for \ \lambda \in \{l, c\}. \tag{10}$$

The calculated gradients are as follows :

$$\Delta \boldsymbol{w}^{\lambda n} = diag\left(\boldsymbol{\delta}^{\lambda}\right) \frac{\partial \boldsymbol{s}}{\partial \boldsymbol{w}^{\lambda n}}, \quad for \; \lambda \in \{l, c\}, \tag{11}$$

$$\Delta \boldsymbol{w}^{\lambda n} = \left(\boldsymbol{\delta}^{\lambda}\right)^{\top} \boldsymbol{y}^{n}, \quad for \; \lambda \in \{\omega, o\}, \tag{12}$$

while $diag(\boldsymbol{x})$ denotes the diagonal matrix with the diagonal entries being $\boldsymbol{x}$. If we want to do batch learning for truncated BPTT, memory cell statuses for different samples will be mixed in Eqs. (5) and (6). However, each samples should be calculated in parallel in matrix and independently before Eqs. (11). This very fact determines that the truncated BPTT for LSTM could not be vectorized[2] for thousands or millions of samples at the same time. If we want to apply batch learning to LSTM trained with the truncated BPTT, we have to loop through the samples in a batch, which can be much slower than vectorized operations (see Fig. 3(b) for an empirical justification).

3.3 Learning Rules for Full BPTT

The BPTT approach could be clearly derived by unfolding the temporal operation of a network into a multi-layer feedforward network that grows by one layer at each time step. However, LSTM, with the LSTM block specifically, is hierarchical (Fig. 1(a)), which is manifested in the connections in the same time layer (Fig. 1(b)), so the backpropagation is different from normal BPTT: the backpropagation through time cannot be calculated at once for each time layer, but sequentially as follows (E and $\boldsymbol{\delta}^{o}$ are calculated at first, same as before):

$$\boldsymbol{\delta}^{\theta} = \left(\boldsymbol{w}^{o\theta}\right)^{\top} \boldsymbol{\delta}^{o} + \left(\boldsymbol{w}^{n\theta}\right)^{\top} \boldsymbol{\delta}_{t+1}^{n}, \tag{13}$$

$$\boldsymbol{\delta}^{\omega} = (f^{\omega})'(\boldsymbol{x}^{\omega}) \circ \left(f^{s}(\boldsymbol{s}) \circ \boldsymbol{\delta}^{\theta} + \left(\boldsymbol{w}^{n\omega}\right)^{\top} \boldsymbol{\delta}_{t+1}^{n}\right), \tag{14}$$

$$\frac{\partial E}{\partial \boldsymbol{s}} = \boldsymbol{\delta}^{\theta} \circ \boldsymbol{y}^{w} \circ (f^{c})'(\boldsymbol{x}^{c}) + \frac{\partial E_{t+1}}{\partial \boldsymbol{s}} \tag{15}$$

$$\boldsymbol{\delta}^{c} = (f^{c})'(\boldsymbol{x}^{c}) \circ \boldsymbol{y}^{l} \circ \frac{\partial E}{\partial \boldsymbol{s}}, \tag{16}$$

$$\boldsymbol{\delta}^{l} = (f^{l})'(\boldsymbol{x}^{l}) \circ \left(\frac{\partial E}{\partial \boldsymbol{s}} \circ f^{c}(\boldsymbol{x}^{c}) + \left(\boldsymbol{w}^{nl}\right)^{\top} \boldsymbol{\delta}_{t+1}^{n}\right). \tag{17}$$

The error responsibilities $\boldsymbol{\delta}$s are calculated in the reverse direction of error flow. The error of next time step backpropagated via the outcoming weights, together with the error of this time step (probably zero if no target is present at this time step), gives the summed full error of this very time step. The above equations can be derived directly from the connections given in Fig. 1(b). For continual prediction tasks, the target $\boldsymbol{O}$ is present at time step $t \in \{t_1, \cdots, t_n\}$, and the according error at every time step should be included into error responsibilities at each time step. Using the standard BPTT equation to accumulate the δs of different time steps, we get the gradients for weights updates:

$$\Delta \boldsymbol{w}^{\lambda n} = \sum_{t=t_0+1}^{t_n} \left(\boldsymbol{\delta}^{\lambda}\right)^{\top} \boldsymbol{y}_{t-1}^{n}, \quad for \; \lambda \in \{l, c, \omega, o\}. \tag{18}$$

We can see that the full BPTT maintains the network's nodes statuses (Eq. 18) at every time step, hence no matrices but vectors are needed for each sample. We can place different nodes in separate rows and different samples in separate columns, and use the resulted matrix to do the vectorized calculation for each layer. For samples with different lengths, we can simply fill zeros to make the length of samples in batch the same as the longest sample, and use a mask matrix to select the effective samples. The gradient in Eq. 18 should be divided by the number of samples. For tasks involving continuous prediction, the length of samples should also be divided individually to get the average gradient for one time step. In this way batch learning can work for LSTM learned with full BPTT for various tasks.

---

[2]Vectorization refers to a powerful way to speed up algorithms For example, if $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{w} \in \mathbb{R}^n$ are vectors and one need to compute their inner product z, one can iterate each element of $\boldsymbol{x}$ and $\boldsymbol{w}$ and add their products to z, or simply calculate $\boldsymbol{w}'\boldsymbol{x}$ with vector-vector multiplication. Numerical computing and parallel computing researchers have put decades of work (packages like Basic Linear Algebra Subprograms (BLAS) etc.) into making certain vectorized numerical operations (e.g. matrices and vectors multiplications) fast.
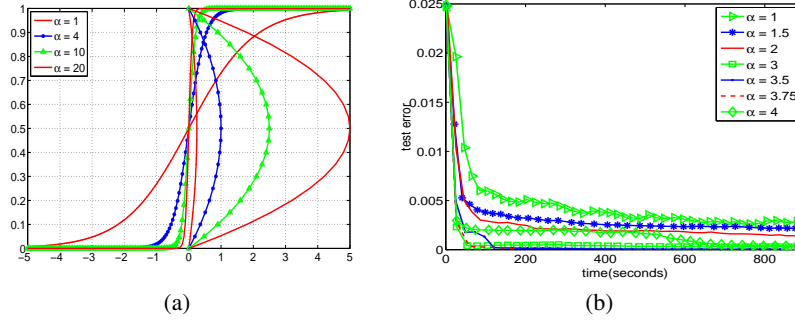
Figure 2: (a) Logistic sigmoid functions with different sharpness and their corresponding derivatives; (b) Experiments on multiplication tasks (Sec. 5.1) with different sigmoid functions (trained with L-BFGS).

From the above analysis, we can easily see that the update complexity per weight and time step is the same for the two methods, namely $O(1)$. Yet the full BPTT needs memory in proportion to the maximum time steps of training sequences, while the truncated BPTT only needs a fixed amount of memory. The truncated BPTT for LSTM is more suitable in online learning scenario, where the weights can be updated efficiently as the input stream flows in (no backpropagating through time is needed), and even an infinite length of input steps will not be a problem either. But there are more downsides of truncated BPTT that makes full BPTT a better choice. In the first place, the truncated BPTT actually hampers LSTM's ability to learn due to the loss of gradient information caused by truncation. Neither can LTSM with the truncated BPTT generalize to a deeper network with multiple layers of memory blocks, because the backpropagated error is truncated and will never reach the earlier layers of memory blocks. In practice, since the full gradient can be checked numerically, its implementation is also easier to debug. The most important attribute of the full BPTT for LSTM is that it can be learned with matrix-based batch learning. With truncated BPTT however, simply looping is the only choice for batch learning, which tends to be slower than SGD when a large batch (mini-batch) size is used. Besides, the time needed for tuning batch size can be intimidating. For these reasons, SGD turns out to be a better choice for LSTM trained with truncated BPTT. For tasks with large volumes of already collected data for training, SGD will be considerably slower compared to batch learning with all the accelerating extensions that will be discussed in Sec. 5.2, 5.3, and 5.4.

4 Increasing Sharpness of Gates' Activation Functions

The input gates, forget gates and output gates, as they are so named, should behave like gates that are either open or closed. Any leakage of unwanted information flow will interfere with the memory cell states or directly changes output. However, a nonsmooth function like binary step function should be avoided for improper derivative of back propagation learning algorithms. Like in ordinary neural networks, the gates use the logistic sigmoid function $f(x) = \frac{1}{1+e^{-\alpha*x}}$ as their activation functions, whose response is ranged from 0 to 1. Here α is the sharpness (slope parameter), which is usually set to 1. Fig. 2(a) shows the logistic sigmoid function with different α and the corresponding derivatives. We can see that the standard practice to assign 1 for α has a quite wide region that will return values far away from 0 and 1. To avoid this problem, we can increase α of the gate excitation function to make it more like a binary step function. We tried different values of α and found that for long term dependency problems that have sequence lengths more than tens or hundreds of time steps, bigger α can lead to faster and better convergence (Fig. 2(b)). It is important to note that when dealing with samples that have a long time span, optimization algorithms in Sec. 5.3 cannot progress with a small α such as 1. However, too large an α will contribute to the exploding gradient problem in backpropagation (e.g. if the sharpness is above 4, the derivatives may exceed 1), and sometimes can even cause severe numerical problems. A large value of α will also impede the error signal from getting into the cell and getting out to the output, which will make the gradient value small and slow down the learning process at the beginning on the contrary. If necessary, we can gradually increase the α (which is an adaptive parameter that could be tuned) by training epochs, and use the previous parameters as initialization. In theory the upper bound of α could be infinity when the network is
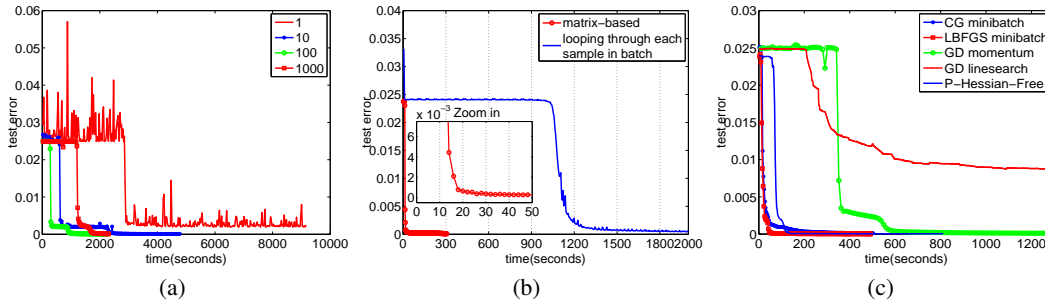
Figure 3: (a) batch learning with gradient descent; (b) vectorization comparison for (mini-)batch learning, where the batch size is 100 and samples of 10 time steps; and (c) comparison with optimization algorithms.

perfectly trained on noiseless data and no error is backpropagated. In practice we can simply keep using the initial setting for stability against noises in data.

5 Experiments

5.1 Experiment Settings

For hardware, we use an AMD CPU at 4.33GHz [3] and a GeForce GTX 770 GPU. We perform experiments using MATLAB and its built-in GPU computing implementation in MATLAB's Parallel Computing Toolbox. For experiment with computer cluster we use the toolbox as in [7]. Since the vanilla LSTM is a non-generative model, and the lack of good benchmarks for measuring a recurrent network's ability, we focus here on LSTM's primary goal, i.e., the memorization of information. It can be integrated with other models as a memory module to improve their performances, or be combined with probabilistic layers directly for outputs as in [8] as an enhanced model, which we leave as future extensions. All the experiments are conducted on the classical multiplication problem, which can only be solved with RNN with HF and LSTM (see some other experiments in Supplementary Material[4]). In this problem, the input to the RNN is a sequence of random numbers, and its target output, which is located at the end of the sequence, is the multiplication of the two marked numbers. The marked numbers are far from the end of the sequence. Moreover, they are distant from each other with various positions. We refer to [1] for a formal description of the multiplication problem. In all experiments we use 10,000 training samples and 3,000 testing samples generated according to the task with a minimal length of 100. α for the gate activation function is set to 3.75, and the (mini-)batch size is set to 1,000 and 100 for optimization algorithms and gradient descent methods with momentum respectively. The stopping criterion is that the mean absolute test set error is below 0.01. The LSTM network uses 2 memory cell blocks, each with 2 memory cells, and all the gates are densely connected. We use peephole connections but omit forget gates since this task is not a continual prediction task as discussed in Sec. 4. This is the general settings for all the experiments if not specified otherwise. It is significant that while RNN with HF need 50,000,000 and original LSTM need 1,273,000 training samples respectively, with revised gate activations and batch learning, our experiments only need 10,000 training samples or less and produce better performance.

5.2 Gradient Descent

We conduct experiments to compare the SGD method (one sample per iteration) with gradient descent methods with various mini-batch sizes for LSTM, using a fixed learning rate of 0.1 as in [1] and a momentum of 0.9. We can see that with batch learning, the learning curve will be steadier and less error-prone, and the convergence can be much faster because of the robust estimate of gradient. An appropriately tuned batch size (e.g., 100 in this case) could make the learning even faster (Fig. 3(a)). In this comparison, we used matrix-based batch learning for vectorization. If not, we

---

[3]We ran experiments on one core for fair comparison because matrix-base batch learning will be further speed up in proportion to number of cores in CPU by multithreading in MATLAB.

[4]https://bitbucket.org/huashiyiqike/nips-workshop/src

have to loop through each sample in a batch for batch learning, which can be slower in proportional to the number of samples in a batch, as shown in Fig. 3(b). With mini-batch size of 100, matrix-based batch learning is about 100 times faster than simply looping, which is the only way of batch learning for truncated BPTT for LSTM.

5.3 Beyond Gradient Descent

Batch methods, such as Limited memory BFGS (L-BFGS) or Conjugate Gradient (CG), with the presence of a line search procedure, are usually much more stable to train and easier to check for convergence. This has already been testified in deep learning [7]. LSTM with its complicated hierarchical architecture, however, can be harder to train than deep learning with the same number of nodes. For sequences of a long time span, a large α for activation function of gates as described in section 4 is a premise for fast convergence. The results in Fig 3(c) show that batch optimization algorithms like L-BFGS and CG[5] indeed have a prominent gain in learning speed.

Hessian Free (HF) greatly alleviate the exploding/vanishing gradient problems for the training of standard RNN, and even endowed it with some ability of long term memory with structural damping for hidden layer [3]. It is interesting to investigate whether LSTM learning will benefit from the second order gradient information of HF. For investigation purpose, we use numerical computed Hessian-Vector products instead of approximation like Gauss-Newton matrix and discard negative curvatures, at price of higher computational complexity. We adopt L-BFGS matrix as pre-conditioner for speeding up CG in the inner loop of HF. Even with these as the best settings, HF does not perform better in convergence: the mean absolute test set error is below 0.01 only 4 times in 15 runs, while L-BFGS and CG can succeed in each run. HF also did not show advantages in solving extremely hard problems like XOR problem (see Supplementary Materials) with long dependencies. However, there is possibility by setting up a special damping scheme taking into consideration LSTM's layer-wise structure with gate functionality, LSTM can converge better when the weights are close to optimum with HF.

5.4 Large-Scale LSTM Batch Training with Parallel Architectures

Now we turn to the hardware extensions for accelerating large-scale LSTM batch learning. In the following experiments, the weight updates at every iteration are the same as before, yet the calculation is no longer done on a single CPU, but in multiple threads in GPU or with multiple machines at the same time. The speed up for each iteration equals to the speed up for convergence, so we compare not the convergence rate but the average running time (in seconds) per iteration. There are also extra time costs for transferring data and network parameters. Because data transferring only needs to be done at the beginning for once, we only consider the transfer time for network weight parameters in our experiments. In GPU computation we can use global variables to trivialize the transferring time between CPU and GPU rams, and in cluster computations we can reduce the transferring time by additional layers of mergers as described in Map-Reduce-Merge [9]. The parameter change is achieved by increasing the gate number (increasing input or output size works the same way). However, a densely connected large-scale LSTM with the full BPTT batch training can be very memory consuming because the memory of node statuses (plus mask variables if the sequences are different in length) is proportional to the number of gates, batch size and maximum sequence length. This problem could be alleviated with a large ram GPU, sliced training sequences, distributed computation techniques to be presented in the Sec. 5.4.2, or multiple GPUs like in[10]. The sparse connected LSTM in [5] is another option. We reduced the minimal sequence length to 10 in the following experiments and report the average iteration time with 5 runs.

5.4.1 Training LSTM with CUDA

Our experience shows that MATLAB's built-in GPU computing implementation in MATLAB's Parallel Computing Toolbox with arrayfun function specifically for element-wise operation works even better than Jacket[6], and runs up to 6.5 times faster on the GTX 770 GPU than on the 4.33 GHz

---

[5]We used LBFGS and CG in minFunc by Mark Schmidt (http://www.di.ens.fr/~mschmidt/Software/minFunc.html), which are fairly optimized implementations.

[6] http://www.accelereyes.com/. Jacket is no longer supported and is integrated into MATLAB's Parallel Computing Toolbox.
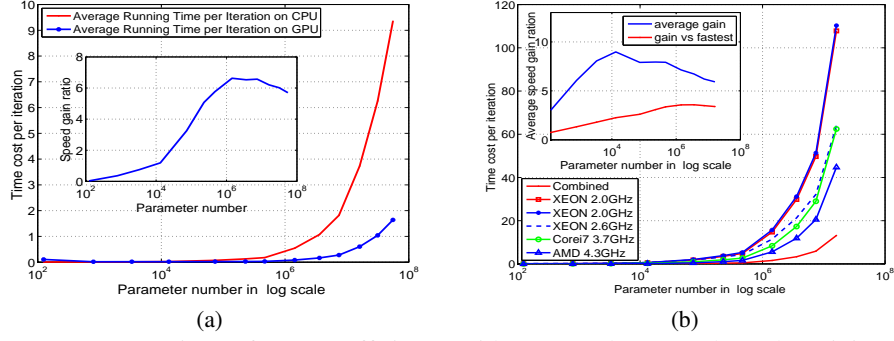
Figure 4: (a) Comparison of LSTM efficiency with CPU and GPU, where the minimal sample length is 10 for GPU memory issue; (b) Comparison of LSTM efficiency with MapReduce, where the batch size is 5,000 as the sum of 5 client with 1,000 samples each. The comparison is conducted with the same batch size for each client alone. The minimal sample length is 10.

CPU (Fig. 4(a)). When the network is small the memory transfer cost counterbalanced the benefit of parallel running on GPU. When network is larger, e.g., having more than 50 memory cells, which is common in practice, the GPU surpasses by a large margin (Fig 4(a)). However, there are certain issues we found in practice. The single float precision commonly used for GPU computation is not very stable when applying optimization algorithms to LSTM batch training. This can be natural because the computed gradient will be backpropagated though a long time span and many layers, making a small truncation error blow up easily. We did the experiment with double precision, which actually slows down the GPU computation for jobs of converting between double and single precisions inside CUDA kernels.

5.4.2 Training LSTM with MapReduce

In MapReduce [11], every client machine only needs to process its fraction of the total samples. The clients calculate their own gradients and send these values to the server via a local network. The reducer then reduces the values to get an averaged gradient as the combined batch gradient for adjusting model parameters. In this way, the computation time can be greatly reduced if one has a large training set. We conduct the experiments on a cluster with heterogeneous computing clients, i.e., the clients with different CPU architectures and speeds. The differences between clients can make the synchronization difficult for parallelizing, because the faster computer has to wait the slower clients for each update. With these unfavorable factors, the acceleration is still quite satisfactory: With 5 clients, we achieve almost 3 times speed up compared to the fastest client alone, and up to 8 times faster than average (Fig.4(b)). The achieved gain can be superimposed on the gain of using GPU implementations demonstrated in the previous section. SGD methods however, cannot benefit much from them, because GPU computation excels only when there are lots of matrix manipulation, and in MapReduce the communications between client computers and server take more time than processing for a sample per iteration. The distributed Downpour SGD [12] may speak for SGD, but it only shows advantages with huge datasets at the expense of tens of thousands of CPU cores, and applies to extremely large decomposable networks without hierarchy in its architecture, which does not apply here.

6   Conclusions

In this paper, we propose a matrix-based batch learning method for LSTM with full Backpropagation Through Time (BPTT). We then solve the state drifting issues as well as improving the overall performance for LSTM using revised activation functions for gates. With these changes, the advanced optimization algorithms can be applied to LSTM with long time dependency and show great advantages over SGD methods. The experiments with CUDA and MapReduce show that there is much room left for accelerating large-scale LSTM training. In all the experiments we also use sparse initialization which generally helps, but further investigation is needed to reveal the best practice for initialization.

Acknowledgements

References

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[2] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE, 2013.

[3] James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning*, pages 1033–1040, 2011.

[4] Felix A Gers and Jürgen Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[5] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[6] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[7] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 265–272, 2011.

[8] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[9] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[10] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V Le, Mark Z Mao, Marc'Aurelio Ranzato, Andrew W Senior, Paul A Tucker, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1232–1240, 2012.