Parallelizing Linear Recurrent Neural Nets Over Sequence Length

Eric Martin
eric@ericmart.in

Chris Cundy
c.cundy@berkeley.edu

Abstract

Recurrent neural networks (RNNs) are widely used to model sequential data but their non-linear dependencies between sequence elements prevent parallelizing training over sequence length. We show the training of RNNs with only linear sequential dependencies can be parallelized over the sequence length using the parallel scan algorithm, leading to rapid training on long sequences with small minibatch size. We abstract prior linear sequence models into a new framework of linear surrogate RNNs and develop a linear surrogate long short-term memory (LS-LSTM) powered by a parallel linear recurrence CUDA kernel we implemented. We evaluate the LS-LSTM on a long sequence noisy autoregressive task and find the LS-LSTM achieves slightly superior train and test performance to a similar sized LSTM in 4x less training time. We analyze latency and throughput of the LS-LSTM and find the LS-LSTM reaches up to 175x the throughput of the LSTM in the small minibatch long sequence regime.

1 Introduction

Recurrent neural networks (RNNs) are widely used for sequence modelling tasks in domains such as natural language processing [18], speech recognition [2], and reinforcement learning [10]. Most RNNs, including popular variants such as long short-term memories (LSTMs) [11] and gated recurrent units (GRUs) [6], contain a non-linear dependency between sequential inputs. These non-linear dependencies create a very flexible class of models but limit the feasibility of training RNNs on long sequences as each sequence element must be processed sequentially. Modelling sequences of thousands to millions of elements is important to domains such as robotics, remote sensing, control systems, speech recognition, medicine, and finance.

The RNN serial evaluation inefficiency problem is usually mitigated by parallelizing the forward and backward pass over a minibatch of inputs. Without minibatches, RNN evaluation is a sequence of matrix-vector multiplications. Minibatches transform RNN computation into a sequence of more efficient matrix-matrix multiplications, but minibatches within RNNs present many issues. RNN model size is often limited by GPU memory size, and running a forward and backward pass on a minibatch requires memory linear in the minibatch size. Grouping data into minibatches increases the latency of each pass and reduces the rate of optimization steps. Finally, training with larger minibatches damages generalization ability [13]. Persistent RNNs [7] use a novel implementation that can achieve high GPU utilization with very small minibatch sizes when the recurrent state is larger than 500 elements, but even persistent RNNs become limited by the serial evaluation inefficiency at smaller hidden sizes.

Numerous prior works have shown strong performance from neural sequential models with only linear dependence on earlier sequence elements. Balduzzi and Ghifary [3] investigated RNNs with only elementwise linear recurrence relations $h_t = \alpha_t \odot h_{t-1} + (1 - \alpha_t) \odot x_t$ and developed linear variants of LSTM and GRU that perform similarly to standard non-linear RNNs on text generation tasks. Bradbury et al. [5], Kalchbrenner et al. [12], Gehring et al. [8], and van den Oord et al. [19] have successfully applied networks of convolutions over sequences for tasks such as machine translation, language modelling, and audio generation. These works have observed up to an order of magnitude increase in training throughput compared to RNN alternatives. Convolutional sequence models typically rely on either an attention mechanism or a (possibly linear) recurrent layer to integrate information at scales larger than the filter width. Introduction of a recurrent layer prevents full parallelization over the sequence length while attention mechanisms are expensive to apply on long sequences in online inference use cases. One dimensional convolution can be viewed as a learnable linear finite impulse response (FIR) filter with a parallel evaluation algorithm, while linear recurrence is a learnable linear infinite impulse response (IIR). This work parallelizes evaluation of linear recurrences through application of the parallel scan algorithm.

Scans and reductions are computations involving repeated application of a binary operator $\oplus$ over an array of data. Computing the sum or maximum of an array is an example of a reduction, while a cumulative sum is a common example of a scan operation. Throughout this work, the scan of $\oplus$ with initial value b is defined as

$$\text{SCAN}(\oplus, [a_1, a_2, ..., a_n], b) = [(a_1 \oplus b), (a_2 \oplus a_1 \oplus b), ..., (a_n \oplus a_{n-1} ... \oplus a_1 \oplus b)]$$

The reduction of $\oplus$ over array A and initial value b is denoted REDUCE($\oplus, A, b$) and is the final element of SCAN($\oplus, A, b$). Despite their dependent computation graph, algorithms exist to parallelize scans and reductions when $\oplus$ is associative [15].

Blelloch [4] shows that first order recurrences of the form $h_t = (\Lambda_t \otimes h_{t-1}) \oplus x_t$ can be parallelized with the parallel scan algorithm if three conditions are met:

1. $\oplus$ is associative: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
2. $\otimes$ is semiassociative: there exists a binary associative operator $\odot$ such that $a \otimes (b \otimes c) = (a \odot b) \otimes c$
3. $\otimes$ distributes over $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

Our primary contribution is the application of the parallel linear recurrence algorithm to RNNs.

2 Parallel linear recurrence

Using elementwise vector addition as $x \oplus y = x + y$, matrix-vector multiplication as $A \otimes x = Ax$, and matrix-matrix multiplication as $A \odot B = AB$ satisfies Blelloch's three conditions, thus allowing linear recurrence $h_t = \Lambda_t h_{t-1} + x_t$ to be evaluated in parallel over t for vectors x_t and square matrices Λ_t. As the method is essential to this work, Algorithm 1 presents the parallel linear recurrence algorithm for the interested reader.

Algorithm 1 Parallel linear recurrence on p processors

```
1:  Let $y = [(\Lambda_1, x_1), (\Lambda_2, x_2), ..., (\Lambda_T, x_T)]$
2:  Let binary operator $\bullet$ act as $(\Lambda, x) \bullet h = \Lambda h + x$
3:  Let $S_0 = 1, S_i < E_i, E_i + 1 = S_{i+1}, E_{p-1} = T$ for $i$ in $0, p-1$
4:
5:  parfor $i \leftarrow 0, p-1$ do
6:      $P_i = \text{REDUCE}(\odot, \Lambda_{S_i:E_i}, I)$
7:      $R_i = \text{REDUCE}(\bullet, y_{S_i:E_i}, 0)$
8:  end parfor
9:
10: Let $z = [(P_0, R_0), (P_1, R_1), ..., (P_p, R_p)]$.
11: $C = \text{SCAN}(\bullet, z, h_0)$          ▷ compute $C_i = P_i C_{i-1} + R_i$ with $C_{-1} = h_0$
12:
13: parfor $i \leftarrow 0, p-1$ do
14:     $h_{S_i:E_i} = \text{SCAN}(\bullet, y_{S_i:E_i}, C_{i-1})$
15: end parfor
16: return $h$
```

2.1 Theoretical performance

The cost of a serial scan over a sequence of length T is $C_{\text{sscan}} = (C_\otimes + C_\oplus)T$, compared to the parallel scan cost $C_{\text{pscan}} = 2(C_\odot + C_\otimes + C_\oplus)(T/p + \lg p)$ [4]. If h_t is a vector of dimension n then $C_\odot = n^3, C_\otimes = n^2, C_\oplus = n$ giving $C_{\text{pscan}} = 2(n^3 + n^2 + n)(T/p + \lg p)$ and $C_{\text{sscan}} = (n^2 + n)T$. The n^3 cost of the matrix multiplication in the parallel algorithm can destroy any parallel speedups for sufficiently large hidden states and lead to a slower algorithm overall.

To avoid this problem, we will only consider diagonal matrices Λ_t, in which case matrix-matrix and matrix-vector have cost n and $C_{\text{pscan}} = 6n(T/p + \lg p)$ and $C_{\text{sscan}} = 2nT$. Assuming $p \ll T$, then $C_{\text{pscan}} \leq C_{\text{sscan}}$ when $p \geq 3$. As we are only considering diagonal matrices, the linear recurrence will be written $h_t = \lambda_t \odot h_{t-1} + x_t$ where $\odot$ indicates elementwise multiplication.

Limiting Λ_t to be diagonal may seem like a severe constraint but there are several reasons to do so beyond the unfavorable parallelization performance. Relatively few neural network models use separate recurrent matrices for each sequence element and using these separate matrices would require potentially prohibitive n^2T memory. Applying the same matrix Λ to each sequence element is also unappealing considering that a matrix multiplication can be thought of as a rotation and a scaling. The same rotation at every element seems unlikely to be useful, and the scaling is exactly what's captured in diagonal vectors λ_t. Recurrent coefficient vectors λ_t provide enough flexibility to implement schemes such as exponential moving averages or a gating mechanism.

2.2 Backpropagation

$$\begin{aligned}
\nabla_{h_T} L &= \frac{\partial L}{\partial h_T} \\
\nabla_{h_t} L &= \frac{\partial h_{t+1}}{\partial h_t} \odot \nabla_{h_{t+1}} L + \frac{\partial L}{\partial h_t} \\
&= \lambda_{t+1} \odot \nabla_{h_{t+1}} L + \frac{\partial L}{\partial h_t} \\
\nabla_{\lambda_t} L &= \frac{\partial h_t}{\partial \lambda_t} \odot \nabla_{h_t} L = h_{t-1} \odot \nabla_{h_t} L \\
\nabla_{x_t} L &= \nabla_{h_t} L \\
\nabla_{h_0} L &= \frac{\partial h_1}{\partial h_0} \odot \nabla_{h_1} L = \lambda_1 \odot \nabla_{h_1} L
\end{aligned}$$

The backpropagation equations center around a linear recurrence over $\frac{\partial L}{\partial h_t}$ in the reverse order of the original sequence. This allows for parallelizing both the forwards and backwards pass of a linear RNN over the sequence length.

2.3 Implementation

A modern high-end NVIDIA GPU consists of between 640 and 3200 concurrently executing warps. Each warp operates on 32 single precision floating point numbers in parallel.

This work implemented parallel linear recurrence as a CUDA kernel with bindings into the TensorFlow [1] framework. Each warp acts as a processor, which means algorithmic p is up to 3200 and the theoretical parallelization speedup factor is up to several hundred. The 32 lanes of each warp work on different elements of the recurrence vector in parallel. These implementation details mean that peak performance is only obtained on sequences of at least several thousand steps on at least a 32 element vector.

3 Models

Parallel linear recurrence can be used to construct a wide variety of differentiable modules that can be evaluated in parallel. Common applications of linear recurrence include gating schemes and exponential moving averages. Although linear recurrence values can depend only linearly on previous elements, the stacking of linear recurrent layers separated by non-linearities allows for a non-linear dependence on the past. In this sense the non-linear depth of a linear recurrent network is the number of layers and not the sequence length.

3.1 Gated impulse linear recurrent layer

A gated impulse linear recurrent (GILR) layer transforms its m dimensional inputs x_t into a sequence of n dimensional hidden states h_t:

$$\begin{aligned} g_t &= \sigma(Ux_t + b_g) \\ i_t &= \tau(Vx_t + b_z) \\ h_t &= g_t \odot h_{t-1} + (1 - g_t) \odot i_t \end{aligned}$$

A GILR layer applies the same non-linear transform to each sequence element and then accumulates the sequence elements with a non-linear gating mechanism. Gate g_t uses the sigmoid activation function to have values in [0,1] for reasonable gating semantics, while impulse i_t can use any activation function τ. Stacking GILR layers allows for rich non-linear dependence on previous events while still taking advantage of fast parallel sequence evaluation.

3.1.1 Impact on effective "batch size"

Consider evaluating a vanilla RNN $h_t = \sigma(Uh_{t-1} + Vx_t + b)$ from m inputs to n hidden units on a sequence of length T with minibatch size b using a serial evaluation strategy. At each of T iterations, the naive approach performs two $(b, m)\mathbf{x}(m, n)$ matrix multiplications. Larger matrix multiplications achieve higher throughput due to less IO overhead, so the better approach computes Vx_t for all t ahead of time in a single $(bT, m)\mathbf{x}(m, n)$ matrix multiply. The non-linear recurrence forces even the better approach to perform T potentially small $(b, m)\mathbf{x}(m, n)$ matrix multiplications in serial which makes performance heavily dependent on minibatch size.

Now consider the GILR, noting that it has the same two matrix-vector multiplications per iteration as the vanilla RNN. g and i can each be evaluated for all t with a single $(bT, m)\mathbf{x}(m, n)$ matrix multiplication each. Given g and i, h can be computed using a parallel linear recurrence over T vectors each of bn elements. Rather than T small operations, the GILR can be evaluated over all sequence elements with two large matrix multiplies and a parallel linear recurrence. GILR performance is much less dependent on batch size as the matrix multiplies see an "effective batch size" of bT and T is typically large.

3.2 Linear surrogate RNNs

RNNs learn a transition function $s_t = f(s_{t-1}, x_t)$ which combines previous state s_{t-1} with input x_t to compute current state s_t. Non-linear f prevents application of the parallel linear recurrence algorithm and forces slow serial evaluation. To work around this inefficiency, note that s_t serves a dual purpose. In $s_t = f(s_{t-1}, x_t)$, s_{t-1} serves as an input to f summarizing the previous inputs while s_t serves as the output of f to be passed to other layers of the network. If we decouple these uses, we can instead compute $s_t = f(\tilde{s}_{t-1}, x_t)$ with $\tilde{s}_t$ as a linearly computable surrogate for s_t. With this linear surrogate, non-linear f can still be evaluated. We refer to this class of model as a linear surrogate RNN (LS-RNN). Quasi-RNNs [5] are LS-RNNs using $\tilde{h}_{t-1} = W_k x_{t-k} + ... W_1 x_{t-1}$ and strongly typed RNNs[3] are LS-RNNs with $\tilde{h}_t = x_{t-1}$. Although not a rule, LS-RNNs can often be parallelized over sequence length with either convolution or linear recurrence.

As an example LS-RNN, consider an LSTM:

$$\begin{aligned} f_t, i_t, o_t &= \sigma(U_{f,i,o} h_{t-1} + V_{f,i,o} x_t + b_{f,i,o}) \\ z_t &= \tau(U_z h_{t-1} + V_z x_t + b_z) \\ c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\ h_t &= o_t \odot c_t \end{aligned}$$

An LSTM has state $s_t = (h_t, c_t)$. c_t depends only linearly on c_{t-1}, so no surrogate is needed for c_t. h_t has a non-linear dependence on h_{t-1}, so h_t needs a linear surrogate. With a GILR layer as surrogate, the linear surrogate LSTM (LS-LSTM) is

$$\begin{aligned}
f_t, i_t, o_t &= \sigma(U_{f,i,o}\tilde{h}_{t-1} + V_{f,i,o}x_t + b_{f,i,o}) \\
z_t &= \tau(U_z\tilde{h}_{t-1} + V_z x_t + b_z) \\
c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\
h_t &= o_t \odot c_t \\
g_t &= \sigma(V_g x_t + b_g) \\
\tilde{h}_t &= g_t \odot \tilde{h}_{t-1} + (1 - g_t) \odot \tau(W x_t + b_h)
\end{aligned}$$

For m inputs and hidden size n, the LS-LSTM contains $2n(n + m)$ more parameters than the equivalently sized LSTM to handle the mapping from x to $\tilde{h}$. More generally, a LS-RNN contains all of the same parameters as the underlying RNN as well as some additional parameters to compute the linear surrogate.

4 Experiments

Experiments were performed on the N-MNIST [17] dataset. N-MNIST captures the MNIST digit dataset [16] by panning an event driven camera over each digit. Each example in N-MNIST is a sequence of single pixel events (x, y, polarity, timestamp) where x and y each indicate a pixel position in [0, 33], polarity indicates whether the pixel was switching on or off, and timestamp is the time of the event in microseconds. Videos produced from the event data show that positive polarity events are often located on the leading edge of the digit motion and negative polarity events on the trailing edge. Between the 60,000 digits in the training set, N-MNIST contains approximately 250 million pixel events with sequence lengths ranging from 500 to 8000 and averaging 4000 events.

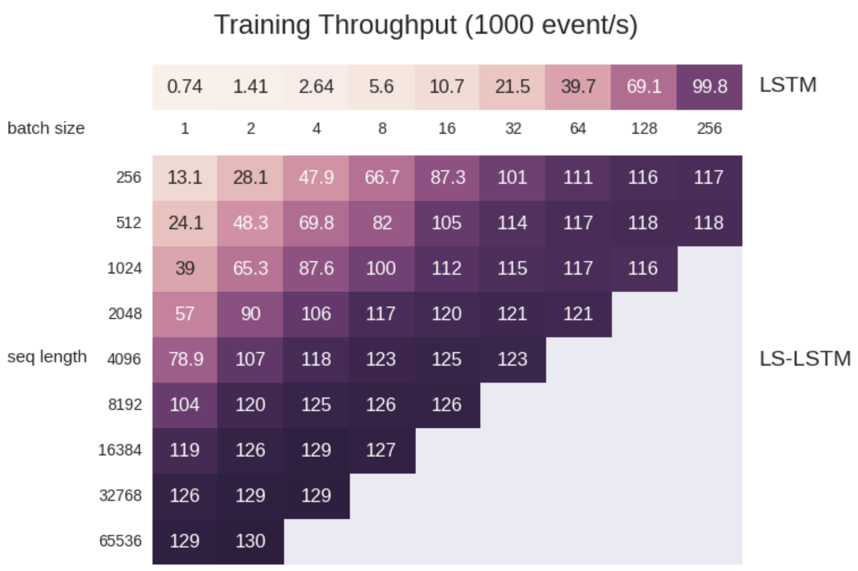
We attempt to forecast 50 events ahead in N-MNIST using a two layer LSTM with 256 units per layer and a two layer LS-LSTM with 234 units per layer. The LS-LSTM layer size was selected so that it had slightly fewer parameters than the LSTM. Each model transformed the incoming event position into a 40-dimensional embedding vector which was then combined with the polarity to produce a 41 dimensional input. Both models output a 2x34^2 matrix containing two future event location probability distributions conditioned on the polarity of the future event. The training algorithm only considers the probability distribution of the true future polarity and uses the cross entropy loss function. The Adam [14] optimization algorithm and Glorot [9] initialization scheme were used. Training on a minibatch of size b with sequence length T consisted of uniformly sampling b N-MNIST sequences and then extracting a single random T element subsequence (and its 50 element ahead forecast) from the sequence. Sequences less than T elements were padded out to length T, and there was a "burn-in" of 30 events at the start of each sequence where no predictions were made. An epoch was defined as a pass over as many pixel events as there are in the full training set.

All experiments were performed using TensorFlow 1.0 [1] on a single Nvidia K80 GPU running for up to 18 hours. The LSTM model was computed using TensorFlow's dynamic_rnn and BasicLSTMCell routines which are slower than but algorithmically similar to the cuDNN LSTM implementation.

4.1 Computational performance

The computational performance of the LSTM and LS-LSTM models were compared across a wide range of minibatch sizes and sequence lengths. Although the 234 unit LS-LSTM has a similar number of parameters to the 256 unit LSTM, we measured the computational performance of a 256 unit LS-LSTM to observe any impact of the linear surrogate calculation. We define throughput as events/s and a minibatch of size b and length T to be bT events. The serial evaluation of the LSTM causes its throughput to be independent of the sequence length as a doubling of sequence length causes a doubling of runtime.

Figure 1 shows the LS-LSTM model achieves a throughput between 1.17x and 175x that of the LSTM, with the greatest relative advantages occurring at small minibatch sizes and long sequences. Notably, the LS-LSTM can achieve a higher throughput by running on one 8192 event sequence at a time than an LSTM running with 256 sequence minibatches. Similar speedups were found for inference. These speedups indicate the use of parallel linear recurrence through a linear surrogate can massively accelerate RNN training.



	1	2	4	8	16	32	64	128	256
LSTM	0.74	1.41	2.64	5.6	10.7	21.5	39.7	69.1	99.8
256	13.1	28.1	47.9	66.7	87.3	101	111	116	117
512	24.1	48.3	69.8	82	105	114	117	118	118
1024	39	65.3	87.6	100	112	115	117	116	
2048	57	90	106	117	120	121	121		
4096	78.9	107	118	123	125	123			
8192	104	120	125	126	126				
16384	119	126	129	127					
32768	126	129	129						
65536	129	130							

Figure 1: Throughput comparison between LSTM-256-256 with LS-LSTM-256-256. The LSTM only has a single row of data because its throughput is independent of sequence length. Entries are missing from the LS-LSTM table because there was not enough memory on the GPU to handle such large batch sizes and sequences.

4.2 Training performance

The speed of neural net training is not solely determined by the training throughput but also by the frequency of optimization steps. As an example, a batch method may process inputs at the same rate as a minibatch method, but the minibatch method will generally converge much faster on large datasets. On an infinite dataset, the batch method never takes a single optimization step regardless of its throughput. With this example in mind, it is clear that achieving fast training is a balancing act between training data throughput and optimization step latency and that training performance should be evaluated with training curves and not just throughput numbers such as time per epoch.

The LSTM and LS-LSTM offer differ latency and throughput tradeoffs. LSTM throughput depends only on minibatch size but LSTM latency depends on both minibatch size and sequence length. Experiments were conducted training the LSTM models with minibatch size 256 and sequence lengths 128, 256, 512, and 1024. Figure 2 shows the smaller sequence lengths led to faster initial learning but inferior final performance.

The throughput and latency of the LS-LSTM are both influenced by batch size and sequence length. Several LS-LSTMs were trained with batch size 16 and sequence length 1024. This combination was chosen because of the nearly maximum throughput, the low latency, and the ease of building a minibatch given the distribution of sequence lengths in the data set. The 234 unit LS-LSTM reaches a better training loss in roughly 4.5 hours than any of the LSTMs could reach in 18 hours. This experimental evidence indicates the LS-LSTM is as powerful of a model as the LSTM and can be trained in a fraction of the time.

4.3 Test performance

Beyond cross entropy on train and test, we also evaluated top-k accuracy for $k = 5, 20$. A probability distribution is top-k correct if it assigns the realized location one of the k largest probabilities. N-MNIST contains 1156 pixel positions, so top-5 and top-20 accuracy are equivalent to localizing the future to 0.43% and 1.7% of pixels.

No regularization was attempted and all of the models overfit, as indicated by the best test performance from the model with the worst train performance. Table 1 contains the full test results. Our focus

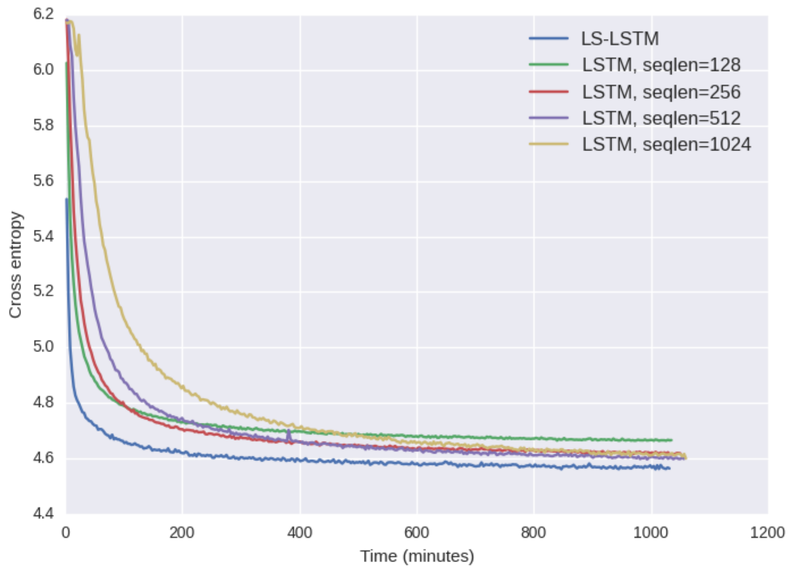


Figure 2: The LSTM models had a direct relationship between sequence length and training curve: the shorter the sequence length, the faster the initial learning and the higher the final loss. The faster initial learning is explained by the decreased latency of optimization steps and the higher final loss is explained by the inability to learn long dependencies. The LS-LSTM has 234 units per hidden layer to have the same number of parameters as the 256 unit LSTMs.

Table 1: N-MNIST test results

	Train Cross Entropy	Test Cross Entropy	Top-5 Accuracy	Top-20 Accuracy
LS-LSTM	**4.567**	4.856	10.51%	36.70%
LSTM, seqlen=128	4.664	**4.846**	**10.62%**	**37.08%**
LSTM, seqlen=256	4.620	4.906	10.11%	35.40%
LSTM, seqlen=512	4.602	4.883	10.45%	36.32%
LSTM, seqlen=1024	4.611	4.880	10.32%	35.92%

was the fast training of powerful models, and we leave regularizing parallel linear recurrences and LS-RNNs to future work. Although not tested, it is possible that the LS-LSTM generalized better than the LSTMs trained on long sequences due to the much smaller minibatch size of LS-LSTM leading to a wider minima.

4.4 Synthetic Example

In order to demonstrate how PLR may be used to speed up tasks that the LSTM is well-suited for, we tackle the pathological example 2 from cite Hochreiter and Schmidhtbuer which involves storing a number for a very long sequence of time steps. We have an alphabet of size p with each character represented as a particular one-hot vector in this space. We input a sequence of n of these vectors, chosen at random (with replacement). The first vector in the sequence is always the same, up to the sign of the component (i.e. it is $\pm p_0$.

The two signs on the first component separate the sequences into two sets. The whole sequence is fed into the LSTM and we aim to learn to classify the sequences. This requires remembering the first element over the length of the sequence. Due to the large throughput of the LS-LSTM, we would expect that it would do better when the sequence length is large. In the original formulation of the problem, p is set equal to n. Since this would make the size of the input data grow impracticaly large as $\mathcal{O}(n^2)$ for long sequences, we fix $p = 128$ and vary n.

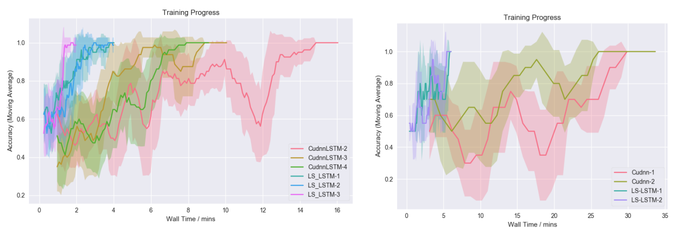


Figure 3: Training curves for LS-LSTM and CUDNNLSTM architectures for various sequence lengths.

Table 2: Performance of the LS-LSTM compared to CUDA-optimised CudnnLSTM implementation on problem 2b from (cite '97). We see that the LS-LSTM has a clear advantage in training speed as the sequence length increases. *For the longest sequence length, the number of hidden units was decreased to 64 for both architectures so that the net could fit in memory.

Sequence Length	**1,024**		**8,192**		**1,048,576****	
	CudnnLSTM	LS-LSTM	CudnnLSTM	LS-LSTM	CudnnLSTM	LS-LSTM
Iterations (1000s)	1.0 ± 0.4	0.55 ± 0.04	0.44 ± 0.05	0.56 ± 0.16		14 ± 3
Wall Clock time (hours)	0.278 ± 0.083	0.0314 ± 0.0019	0.58 ± 0.06	0.10 ± 0.03		9.7 ± 1.7

We generated three sets of data: for n equal to 1,024, 8,192, and 1,048,576. For each of these we compared a two-layer LS-LSTM with 512 hidden units to a canonical LSTM network with roughly the same number of parameters. We could imagine using a two-layer LSTM with 512 hidden units in each layer, or a one-layer LSTM with 1024 hidden units.

Initial experiments showed that the two-layer LSTM was much quicker to converge than the one-layer LSTM, so here we compare the LS-LSTM to the two-layer CudnnLSTM. We ran all experiments on a NVIDIA K80 GPU, choosing the largest minibatch size which fit into the GPU memory, with five runs per configuration allowing us to find the average and standard deviation of the time and number of iterations to convergence. For all CUDA runs, a brief search over learning rate and batch size was carried out to find the parameters which allow the network to converge most rapidly. The criterion for convergence was five consecutive minibatches giving 100% accuracy. As can be seen from the learning curves in figure 3, this was a reasonable criterion.

5 Conclusion

Parallel linear recurrence is an extremely powerful algorithm and the LS-LSTM is just one of many possible models that can be built with it. Future applications of parallel linear recurrence could include sequences orders of magnitude longer than N-MNIST, the development of parallel computable differentiable memory modules, and the combination of linear recurrence with convolutional sequence models. Besides future research, existing models such as Quasi-RNNs and strongly typed RNNs that already contain linear recurrences can immediately benefit from parallel linear recurrence. This work demonstrates the LS-LSTM significantly accelerates the training of small to medium sized LSTMs. Although similar techniques have been used before, the now explicit concept of linear surrogacy provides a framework for future development and analysis of fast sequence models.

We intend to expand upon the experiments section and open-source the parallel linear recurrence kernel in the near future.

Acknowledgments

We would like to acknowledge Kevin Bowers, Alex Meiburg, JD Co-Reyes, Carson McNeil, and several others for fruitful conversations and guidance.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

[3] D. Balduzzi and M. Ghifary. Strongly-typed recurrent neural networks. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 1292–1300, 2016.

[4] G. E. Blelloch. Prefix sums and their applications. 1990.

[5] J. Bradbury, S. Merity, C. Xiong, and R. Socher. Quasi-recurrent neural networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[6] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[7] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*, pages 2024–2033, 2016.

[8] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*, 2017.

[9] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[10] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

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

[12] N. Kalchbrenner, L. Espeholt, K. Simonyan, A. v. d. Oord, A. Graves, and K. Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

[13] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. 2017.

[14] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4): 831–838, 1980.

[16] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits, 1998.

[17] G. Orchard, A. Jayawant, G. Cohen, and N. Thakor. Converting static image datasets to spiking neuromorphic datasets using saccades. *arXiv preprint arXiv:1507.07629*, 2015.

[18] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[19] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499*, 2016.