Parallelizing Linear Recurrent Neural Nets Over Sequence Length

Anonymous authors
Paper under double-blind review

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 even with small minibatch size. We develop a parallel linear recurrence CUDA kernel and show that it can be applied to immediately speed up training and inference of several RNN architectures by up to 9x. We abstract recent work on linear RNNs into a new framework of linear surrogate RNNs and develop a linear surrogate model for the long short-term memory unit, the GILR-LSTM, that utilizies parallel linear recurrence. We push the limits of sequence learning by successfully training a GILR-LSTM on a synthetic sequence classification task with a one million timestep dependency. SOMETHING MEDICAL?

1 Introduction

Recurrent neural networks (RNNs) are widely used for sequence modelling tasks in domains such as natural language processing Sutskever et al. (2014), speech recognition Amodei et al. (2015), and reinforcement learning Hausknecht and Stone (2015). Most RNNs, including popular variants such as long short-term memories (LSTMs) Hochreiter and Schmidhuber (1997) and gated recurrent units (GRUs) Cho et al. (2014), 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 this speed-up brings several disadvantages. 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 Keskar et al. (2017). Given these effects, it is desirable to obtain high training throughput with small minibatches. Persistent RNNs Diamos et al. (2016) 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 (2016) 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. (2017), Kalchbrenner et al. (2016), Gehring et al. (2017), and van den Oord et al. (2016) 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.

A linear recurrence is a specific instance of a general form of computation known as a scan. 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 $\text{REDUCE}(\oplus, A, b)$ and is the final element of $\text{SCAN}(\oplus, A, b)$. Despite their dependent computation graph, algorithms exist to parallelize scans and reductions when $\oplus$ is associative Ladner and Fischer (1980).

Blelloch (1990) 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 $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

Considering the familiar operations in linear algebra, we see that the associative operation of vector addition ($x \oplus y = x + y$), the semiassociative operation of matrix-vector multiplication ($A \otimes x = Ax$) and the associative operation of matrix-matrix multiplication ($A \odot B = AB$) satisfy Blelloch's three conditions, allowing $h_t = \Lambda_t h_{t-1} + x_t$ to be evaluated in parallel over time steps t for vectors x_t and square matrices Λ_t.

We investigate this idea further and deliver the following contributions:

- We classify RNNs which satisfy the conditions above, and show that many RNNs used in practice such as Quasi-RNNs (QRNNs) Bradbury et al. (2017) are contained in this class

- We provide an implementation of the parallel linear recurrence algorithm as a CUDA kernel, and show that it speeds up training of QRNN and Simple Recurrent Unit (SRU) Lei and Zhang (2017) architectures by factors of up to 9x

- We describe how several recent linear RNNs can be described as linear surrogates for non-linear architectures. We introduce a linear surrogate for the LSTM and show that we are able to train it with a speedup of 5-10x compared to the CuDNN-LSTM when we use the parallel linear recurrence algorithm.

2 PARALLEL LINEAR RECURRENCE

As the method is essential to this work, Algorithm 1 presents the parallel linear recurrence algorithm for the interested reader.

2.1 THEORETICAL PERFORMANCE

The cost of a serial scan over a sequence of length T is $C_{\text{sscan}} \in \mathcal{O}((C_\otimes + C_\oplus)T)$, compared to the parallel scan cost $C_{\text{pscan}} \in \mathcal{O}(2(C_\odot + C_\otimes + C_\oplus)(T/p + \lg p))$ Blelloch (1990) on p processors. If h_t is a vector of dimension n then $C_\odot \in \mathcal{O}(n^3), C_\otimes \in \mathcal{O}(n^2), C_\oplus \in \mathcal{O}(n)$ giving $C_{\text{pscan}} \in \mathcal{O}(2(n^3 + n^2 + n)(T/p + \lg p))$ and $C_{\text{sscan}} \in \mathcal{O}((n^2 + n)T)$. The $\mathcal{O}(n^3)$ cost of the matrix multiplication in the parallel algorithm can counter-act 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 both matrix-matrix and matrix-vector multiplication have cost proportional to n and $C_{\text{pscan}} \in \mathcal{O}(6n(T/p + \lg p))$

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: $\quad P_i = \text{REDUCE}(\odot, \Lambda_{S_i:E_i}, I)$
7: $\quad R_i = \text{REDUCE}(\bullet, y_{S_i:E_i}, 0)$
8: **end parfor**
9:
10: Let $z = [(P_0, R_0), (P_1, R_1), ..., (P_p, R_p)]$.
11: $C = \text{SCAN}(\bullet, z, h_0)$ $\qquad\qquad\qquad\qquad \triangleright$ 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: $\quad h_{S_i:E_i} = \text{SCAN}(\bullet, y_{S_i:E_i}, C_{i-1})$
15: **end parfor**
16: **return** h

---

and $C_{\text{sscan}} \in \mathcal{O}(2nT)$. This gives a parallel speedup factor of $\frac{pT}{3(T+\lg p)}$. Assuming $p \ll T$, then $C_{\text{pscan}} \leq C_{\text{sscan}}$ when $p \geq 3$.

As we are only considering diagonal matrices, we write the linear recurrence as $h_t = \lambda_t \odot h_{t-1} + x_t$ 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 $n^2 T$ 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

$$\nabla_{h_T} L = \frac{\partial L}{\partial h_T}$$

$$\nabla_{h_t} L = \frac{\partial h_{t+1}}{\partial h_t} \odot \nabla_{h_{t+1}} L + \frac{\partial L}{\partial h_t}$$

$$= \lambda_{t+1} \odot \nabla_{h_{t+1}} L + \frac{\partial L}{\partial h_t}$$

$$\nabla_{\lambda_t} L = \frac{\partial h_t}{\partial \lambda_t} \odot \nabla_{h_t} L = h_{t-1} \odot \nabla_{h_t} L$$

$$\nabla_{x_t} L = \nabla_{h_t} L$$

$$\nabla_{h_0} L = \frac{\partial h_1}{\partial h_0} \odot \nabla_{h_1} L = \lambda_1 \odot \nabla_{h_1} L$$

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

GPUs commonly used for deep learning in 2017 consist of between 640 and 3200 parallel processors known as 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 Abadi et al. (2016) framework. Each warp acts as a processor, which means the 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:

$$g_t = \sigma(U x_t + b_g)$$
$$i_t = \tau(V x_t + b_z)$$
$$h_t = g_t \odot h_{t-1} + (1 - g_t) \odot i_t$$

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 give 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 an RNN with recurrence $h_t = \sigma(U h_{t-1} + V x_t + b)$ 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 $(b, m) \times (m, n)$ matrix multiplications. Larger matrix multiplications achieve higher throughput due to less IO overhead, so the better approach computes $V x_t$ for all t ahead of time in a single $(bT, m) \times (m, n)$ matrix multiply. The non-linear recurrence forces even the better approach to perform T potentially small $(b, m) \times (m, n)$ matrix multiplications in serial. This makes serial RNN performance heavily dependent on minibatch size.

Now consider the GILR, noting that it has the same two matrix-vector multiplications per iteration as the above RNN. The intermediate variables g and i can be evaluated for all t with a single $(bT, m) \times (m, n)$ 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 multiplications and a parallel linear recurrence. GILR performance is much less dependent on batch size as the matrix multiplication kernel sees 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 dual purposes. 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. We can decouple these uses and introduce independent variables for each purpose: s_t is passed onto other layers of the network and we introduce the linear surrogate $\tilde{s}_t$ which is passed onto the next state, with $s_t = f(\tilde{s}_{t-1}, x_t)$. We are still able to choose a non-linear f, our only limitation being that $\tilde{s}_t$ must be linearly computable.

We refer to this class of model as a linear surrogate RNN (LS-RNN). QRNNs Bradbury et al. (2017) are LS-RNNs using $\tilde{h}_{t-1} = W_k x_{t-k} + \ldots W_1 x_{t-1}$ and strongly typed RNNsBalduzzi and Ghifary (2016) 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.

Consider an LSTM:

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

An LSTM has state $s_t = (h_t, c_t)$. Since c_t depends only linearly on c_{t-1}, 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. Introducing a GILR layer as the surrogate, we obtain the GILR-LSTM:

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

For m inputs and hidden size n, a GILR-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

We perform several experiments. First we find that our parallel linear recurrence kernel is able to able to achieve up to 40x higher throughput than a serial implementation when applied to long sequences. Secondly, we confirm that this kernel speedup translates to up to a 9x speedup to LS-RNNs such as QRNNs.

In order to illustrate that the linearization does not necessarily come at the cost of expressibility, we show that the GILR-LSTM architecture computed with the parallel linear recurrence algorithm is able to train significantly faster than an optimized LSTM implementation on a pathological long-term dependency problem on a pathological example from the original LSTM paper Hochreiter and Schmidhuber (1997). Finally we show that the GILR-LSTM is able to quickly achieve state of the art results on an existing medical data set.

I hope so...

4.1 THROUGHPUT BENCHMARKS

4.1.1 KERNEL PERFORMANCE

We first illustrate the throughput advantage of the parallel scan algorithm for evaluating the linear recurrence. For a minibatch comprised of b sequences of length T, we define the number of events as bT and the throughput as the number of events processed per second. We implement two CUDA kernels, one which evaluates the parallel linear recurrence described in algorithm 2, and one which evaluates the same linear recurrence on GPU in serial over sequence length and in parallel over features and minibatch. The performance of each kernel depends on two factors: the sequence length and the product of number of features and minibatch size. The performance measurements for this experiment are made directly at the kernel level, avoiding any overhead from TensorFlow. We find that the parallel kernel has a distinct advantage at long sequence lengths with a speedup factor of up

Table 1: Parallel kernel speedup on m features (minibatch size $= 1$)

Sequence Length	$m = 4$	$m = 32$	$m = 128$
16	0.06	0.06	0.05
256	0.22	0.22	0.86
4,096	1.02	2.94	3.36
65,536	38.5	41.8	17.5

Table 2: Parallel kernel speedup for a variety of LS-RNNs

Sequence Length	SRU	QRNN (filter size 2)	QRNN (filter size 10)	GILR-LSTM
16	0.28	0.38	0.78	0.61
256	0.84	0.86	0.99	0.91
4,096	1.38	1.18	1.05	0.98
65,536	9.21	6.68	2.05	1.41

to 40x, as shown in table 1. The parallel kernel does not perform well at short sequence lengths due to the overhead of multiple passes over data and communication between processors.

4.1.2 ACCELERATING EXISTING RNN ARCHITECTURES

Several recently introduced LS-RNNs can be accelerated with the parallel linear recurrence algorithm. We implemented SRUs, QRNNs (with filter width 2 and 10), and GILR-LSTMs that can be computed with either the standard serial linear recurrence algorithm or parallel linear recurrence. Both methods compute an identical recurrence, so switching from a serial to parallel implementation does not cause any numerical changes and takes only a single line of code changes. Notably, both SRUs and QRNNs claim an order of magnitude speedup compared to CuDNN LSTM when implemented with serial linear recurrence. Any further speedup from parallel linear recurrence applies on top of the existing speedup. We timed train throughput (forwards and backwards propagation), but the linear time of each pass also makes the results applicable to forwards (inference) performance. We controlled for GPU memory usage within these experiments by fixing $bT = 65536$ for minibatch size b and sequence length T.

Table 2 shows that throughput advantage from using parallel linear recurrence compared to serial linear recurrence. Simpler architectures (for which the linear recurrence is a higher proportion of the total computational load) are more affected by the switch to the parallel kernel. This is particularly clear in the case of the QRNN, where including wider convolutional filters results in more time spent outside of the linear recurrence and therefore speedup from linear recurrence parallelization.

4.1.3 GILR-LSTM THROUGHPUT

Finally, we compare the throughput of the GILR-LSTM to the current fastest LSTM implementation (which comes from the CuDNN library). Figure 1 shows that the GILR-LSTM has higher throughput for most of the small minibatch regime, with a 30x speedup at a sequence length of 65,536. The GILR-LSTM allows us to achieve high throughput for all choices of minibatch size and sequence length.

Unlike the previous case, these speedups do come at a cost. The GILR-LSTM architecture is different to the canonical LSTM, and it's possible that the loss of model capacity outweighs the speedup in training. The following experiment shows that the GILR-LSTM is able to outperform the LSTM on an example originally designed to illustrate the advantages of the LSTM.

For table 2 how many layers and what was hidden size?

I think we should delete this sec-

LSTM

batch size	1	2	4	8	16	32	64	128	256
	14.3	18.4	37.1	90.8	170	305	534	695	772

LS-LSTM

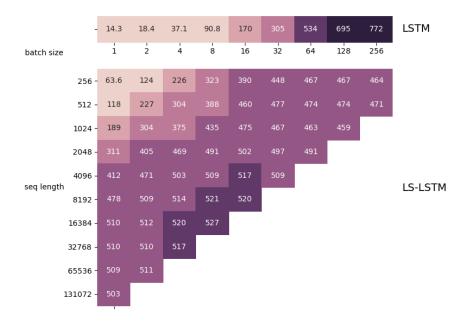| seq length \ batch size	1	2	4	8	16	32	64	128	256
256	63.6	124	226	323	390	448	467	467	464
512	118	227	304	388	460	477	474	474	471
1024	189	304	375	435	475	467	463	459	
2048	311	405	469	491	502	497	491		
4096	412	471	503	509	517	509			
8192	478	509	514	521	520				
16384	510	512	520	527					
32768	510	510	517						
65536	509	511							
131072	503								

Figure 1: Throughput comparison between LSTM-256-256 with GILR-LSTM-256-256, with a 32-dimensional input and a 2-dimensional output, for various batch sizes and sequence lengths. The LSTM only has a single row of data because its throughput is independent of sequence length. Entries are missing from the GILR-LSTM table because there was not enough memory on the GPU to handle such large batch sizes and sequences.

4.2 SYNTHETIC EXPERIMENT

In order to demonstrate how PLR may be used to speed up tasks that the LSTM is well-suited for, we tackle a pathological inference problem first introduced in Hochreiter and Schmidhuber's initial LSTM paper Hochreiter and Schmidhuber (1997). The input consists of sequences of length n with each sequence element a randomly chosen one-hot vector in p dimensional space. The first vector in each sequence is always the same, up to the sign of the component (i.e. it is $\pm p_0$). The sequential model must read in an entire sequence and then output the sign of the first sequence element. This sequence classification problem requires remembering the first element over the length of the sequence and is a canonical example of the long-term dependency problem. Due to the large throughput of the GILR-LSTM, we would expect that it would do better when the sequence length is large. In the original formulation of the problem (dealing in the regime with around one hundred timesteps) , p is set equal to n. As this would make the size of the input data grow impractically large as $\mathcal{O}(n^2)$ for long sequences, we fix $p = 128$ and vary n.

We generated sequences for n equal to 1,024, 8,192, and 1,048,576. For each of these we compared a two layer GILR-LSTM with 512 hidden units to a two layer LSTM with 512 hidden units per layer implemented by CuDNN.

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. A brief search over learning rate and batch size was carried out to find the parameters which allow the network to converge most rapidly for all runs. The criterion for convergence was five consecutive minibatches giving 100% accuracy. The learning curves in figure 2 give support to this being a reasonable convergence criteria. For the longest sequence length, we could not run long enough to observe the LSTM converge.
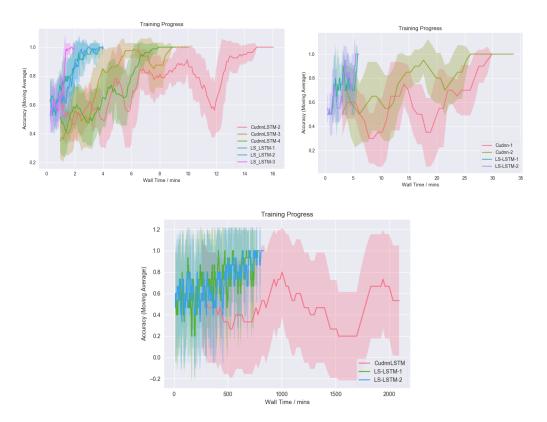
Figure 2: Training curves for GILR-LSTM and CUDNNLSTM architectures for various sequence lengths. (Clockwise from top-left: 1024, 8192, 1048576)

Table 3: Performance of the GILR-LSTM compared to CUDA-optimised CudnnLSTM implementation on problem 2b from (cite '97). We see that the GILR-LSTM has a clear advantage in training speed as the sequence length increases. We did not observe the Cudnn-LSTM converging for the longest task, even after several days. *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*	
	CuDNN	GILR	CuDNN	GILR	CuDNN	GILR
Iterations (1000s)	1.0 ± 0.4	0.55 ± 0.04	0.44 ± 0.05	0.56 ± 0.16	-	14 ± 3
Wall Clock time (hours)	0.28 ± 0.08	0.031 ± 0.002	0.58 ± 0.06	0.10 ± 0.03	-	9.7 ± 1.7

The results illustrate that the GILR-LSTM is able to converge much more quickly than the fastest available LSTM implementation. This is somewhat surprising given the LSTM was specifically constructed for problems of this sort. This GILR-LSTM convergence leads to the conclusion that the non-linearities present in LSTM are not necessary for solving this instance of the long-term dependency problem. The time to convergence further leads to the conclusion that inclusion of a non-linearity at every step incurs a significant training time slowdown. Furthermore, the GILR-LSTM is able to learn to carry dependencies over a one million element sequence. As far as we know, this one million step sequence experiment is the longest sequential learning problem to be handled by neural networks to date.

4.3 ABNORMAL HEART SOUND DETECTION

Medical waveform data consists of long sequences and is a reasonable application area for RNNs. We applied the GILR-LSTM on the problem presented in the 2016 PhysioNet/Computing in Cardiology

Challenge 2016. The aim of the Classification of Normal/Abnormal Heart Sound Recordings challenge is classify electroencephalogram (EEG) recordings as coming from a normal or abnormal heart. The waveforms are recorded on several sets of sensors and split into a training and testing set. The testing set is not publicly available, but can be evaluated through a web portal. The top results cover a large range of techniques, but generally rely on manual featurization of the raw EEG records. The GILR-LSTM's ability to handle long sequences makes it a candidate for training directly on the raw EEG signal. The EEGs are sampled at 2,000 Hz and range in length from several seconds to more than one hundred seconds, giving a max sequence length of over 200,000. Since the testing server is not equipped with a GPU, we cannot run PLR in the inference. We resample the records to 500 Hz so that we are able to evaluate the test results in a reasonable amount of time, padding all records that are shorter than the longest.

The training set of data is unbalanced, with 75% of the data corresponding to normal hearts. We rebalance the data by oversampling the abnormal data by a factor of three, including each abnormal heart three times. We train a network feeding the one dimensional input signal into three stacked GILR-LSTM layers of 16 hidden units each. Although reasonably small, 16 hidden units allows developing develop representations. Parallel linear recurrence allows us to rapidly train the network over the entire long sequence length, so we don't have to work on truncated samples, downsampled sequenes, or spend lots of time manually developing EEG specific features. The model obtained perfect classification on the train dataset after a few hours of training. On the unseen test data we obtained a score of PLACEHOLDER.

how many?

If we get good results (doesn't have to be best ever, but can be on par), then keep

5 CONCLUSION

Parallel linear recurrence can accelerate existing RNNs and be used in the design of new architectures such as the GILR-LSTM. Parallel linear recurrence enables rapid training with small minibatches, which further enables training models on extremely long sequence lengths. The impressive results in Lei and Zhang (2017) and other recent work comprehensively show that linear RNNs can often be used in place of traditional non-linear RNNs. Our results on the synthetic example continue this trend, showing that the GILR-LSTM is able to handle sequence length regimes that are longer than any we have seen seriously suggested in the literature. We feel that the framework of describing linear RNNs as linear surrogates for non-linear RNNs is a fruitful approach, and that this, paired with the PLR algorithm, will substantially advance the ability to train RNNs, and to understand timeseries data.

If we don't get good results, or don't have results, we will need to kick this section

REFERENCES

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.

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.

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

G. E. Blelloch. Prefix sums and their applications. 1990.

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

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.

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.

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

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

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

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

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.

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.

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

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

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

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.

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.

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.

T. Lei, Y. Zhang, Training RNNs as fast as CNNs. *arXiv preprint arXiv:1709.02755* , 2017.

Goldberger et al. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals. In *Circulation* 101(23):e215-e220; 2000 (June 13). PMID: 10851218; doi: 10.1161/01.CIR.101.23.e215