Distributed Language Models Using RNNs

Ting-Po Lee
tingpo@stanford.edu

Taman Narayan
tamann@stanford.edu

1 Introduction

Language models are a fundamental part of natural language processing. Given the prior words in a corpus, we'd like to accurately predict what comes next. Any artificial intelligence algorithm which seeks to understand text and, especially, output coherent text, needs to work off of an accurate language model.

The main way to train language models today is by running Recurrent Neural Networks ("RNNs") on massive corpora of text scraped from books, Wikipedia, newspapers, or even random websites. As these corpora grow larger, it becomes increasingly impractical to hope to train them on even the most powerful single machines. Yet the standard training approach involves sequentially feeding in batches of text into the model, updating weights and hidden states, and then feeding in these new values along with the next batch of text into the next iteration. Models can take weeks to train.

We explore how to leverage distributed file storage and computation to better train corpora. In particular, we implement a ground-up RNN in Apache Spark capable of different parallelization approaches to train a language model based on a corpus assumed to be stored in a Hadoop Distributed File System ("HDFS") or similar manner. Our two broad approaches are an embarassingly parallel approach of training the portions of text stored on different machines in a sequential fashion and a "sliding windows" approach of feeding in randomly-selected snippets of text in each iteration.

Full code can be found at `https://github.com/tblee/rnn_nlp`.

2 Background

2.1 Language Models

The objective function of language models is straightforward.

$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j}\log\hat{y}_{t,j} \tag{1}$$

where T is the number of words in the corpus, $|V|$ is the number of words in the vocabulary, $y_{t,j}$ is one if the tth word of the corpus is the jth word in the vocabulary and zero otherwise, and $\hat{y}_{t,j}$ is the predicted probability of that word.

In practice, the metric reported is the perplexity, which is simply 2^J.

In the past, the dominant methodlogy for language models was to use n-gram counts. That is, given the prior $n-1$ words, the next prediction would be based on how often each word in the vocabulary appears after those $n-1$ words.

Recently, researchers have found that RNNs have outperformed n-gram models both in accuracy and scalability.

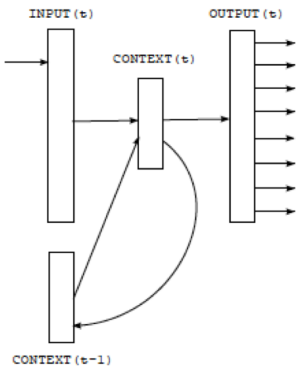


Figure 1: Information flow in an RNN. The input at the current time step is combined with the hidden state vector and produces an output prediction as well as an updated hidden state vector. Image taken from [3].

2.2 Recurrent Neural Networks

The key idea behind Recurrent Neural Networks is that the history of a corpus at any point in time can be represented by a state vector, which is then updated each time an additional word is seen.

In particular, the way this is most often implemented is with a series of matrix multiplies and non-linearities.

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b_h) \tag{2}$$
$$\hat{y}_t = softmax(W_o h_t + b_o) \tag{3}$$

The structure of the RNN lends itself very naturally to language models. Each time step produces a prediction for the next word which is based on the entire history of the text and incorporates the actual next word into its state that will predict subsequent words.

Numerous authors have found high degrees of success with RNNs in language models [3, 4]. An important consideration in getting them to 'work' on large corpora is to limit the scope of backpropagation. Technically, every prior word contributes to the current hidden state, so the first word in the corpus would have an influence on the 10,000th prediction. But realistically, updating weights based on the error for a particular prediction only based on the prior handful of words (e.g. 5-10) achieves similar results in a substantially more efficient fashion. Mikolov [3] only backpropagates one word back in time in his seminal paper on language models and still achieves excellent results.

Most relevantly from the point of view of our paper, Karpathy [2] makes a simpler character-level model with a limited backpropagation window. We use his ideas in our own character-level model and expand it to include support for parallelism.

3 Incorporating Parallelism

The main barrier to incorporating parallelism into language model training is the importance of maintaining the relevant hidden state at any point in time, which theoretically should include contributions from words in all previous time steps. Breaking the corpus and separately training different parts (say, sentences) would reduce the ability of the model to understand context across those breaks because the initial predictions would not have a meaningful hidden state.

Chelba et al. [1] highlight another weakness of using distributed models to train language models, namely the communication and I/O costs of MapReduce. Fortunately, Apache Spark overcomes many of these fixed costs of MapReduce by avoiding repeated I/O, keeping data in memory rather than on disk to the extent practicable, and optimizing broadcasting and All Reduces.

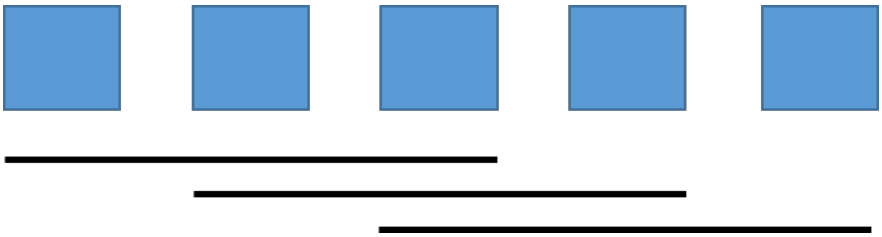


Figure 2: An illustration of sliding windows applied to a corpus. There is a window of length T starting with every character.

The most straightforward way to parallelize language model training would simply be to train sequential models using the text stored on each machine. After each pass through the data, parameters could be aligned across machines with an All Reduce. Given enough data per machine, the "breaks" in the training procedure (from when the text switches machines) would be minor and this approach would be a direct extension of the sequential pass through the full data. A natural extension, furthermore, would be to evenly chunk up the text between the cores of the machine and train from there. In the case where paragraphs are generally long, sequentially training on paragraphs could approximate this approach.

Other mechanisms of splitting up the text in fixed ways, such as separately training sentences, would be more problematic. The lengths of different sentences vary quite widely, which in an RNN context can seriously affect training quality and also pose implementation concerns. Trying to solve this problem by feeding in smaller, fixed chunks of text would be an even worse idea, since splitting consistently within sentences will hinder the ability of the model to learn complete thoughts.

A related but more promising idea would be to train on shorter chunks of data that overlap with one another. To make this more concrete, the text on each machine could be divided into overlapping chunks of T characters each, and those chunks could be trained instead of sequentially passing through the full data.

There are a few possible advantages to this approach. First, especially after introducing sampling of these windows in each training iteration, the model could finish each iteration much more quickly. Second, this approach makes better use of the corpus by presenting each word in multiple different contexts and therefore hidden states, potentially improving generalizability to new texts. Third, it could quickly incorporate new information from all parts of the corpus, improving convergence time.

Finally, compared to other methods of training small sequences in parallel in an SGD-like fashion, sliding windows eliminate the problem of fixed corpus break points. Williams et al. [4] use a similar sliding-window format in a different context of filtering down a large corpus to make it more amenable to training. There, the authors observe that randomly sampling sentences to train on would throw away too much valuable information about how sentences fit together and so instead sample from "rolling windows" of sentences.

We expect the more sequential models to perform very well due to their richer hidden states and proven success. However, the proposed benefits from the sampling-from-overlapping-windows approach may turn out to boost performance in a distributed setting.

4 Algorithmic Analysis

Out of the numerous techniques described above, we focus on a theoretical analysis of 3: sequentially passing through all data on a machine per iteration, training on all sliding windows in an iteration, and sampling from sliding windows each iteration.

In the following analysis, we define N to be the corpus size in characters, P to be the number of machines, C to be the number of cores per machine, L to be the minibatch size when sequential training, T to be the length of each sliding window, H to be the dimenson of the hidden state, and V to be the encoding size of each character (here, we use one-hot encoding and so V is also the number of unique characters). As in Equation 1, we use J to refer to the loss function.

4.1 Per Iteration Cost

First, we start with a sequential training of the data on each machine.

Within each machine, the algorithm steadily computes L new hidden states and makes L predictions before updating the weights via backpropagation through those L time steps. It then moves on to the next L characters. Note that we use the limited backpropagation trick as described above, since technically the first word contributes to the gradient coming from every prediction in the text.

The relevant computations are:

Forward Pass (same as Equations 2 and 3)

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b_h) \tag{4}$$
$$\hat{y}_t = softmax(W_o h_t + b_o) \tag{5}$$

Backward Pass (let $z_t^{(1)} = W_x x_t + W_h h_{t-1} + b_h$ and $z_t^{(2)} = W_o h_t + b_o$)

$$\frac{\partial J_t}{\partial z_t^{(2)}} = \hat{y}_t - y_t = \delta_t^{(2)} \tag{6}$$
$$\frac{\partial J_t}{\partial W_o} = \delta_t^{(2)} h_t^T \tag{7}$$
$$\frac{\partial J_t}{\partial b_o} = \delta_t^{(2)} \tag{8}$$
$$\frac{\partial J_t}{\partial h_t} = W_o^T \delta_t^{(2)} \tag{9}$$
$$\frac{\partial J_t}{\partial z_t^{(1)}} = \frac{\partial J_t}{\partial h_t} \odot \left(1 - h_t^2\right) = \delta_t^{(1)} \tag{10}$$
$$\frac{\partial J_t}{\partial h_{t-1}} = W_h^T \delta_t^{(1)} \tag{11}$$
$$\left.\frac{\partial J_t}{\partial W_x}\right|_t = \delta_t^{(1)} x_t^T \tag{12}$$
$$\left.\frac{\partial J_t}{\partial W_h}\right|_t = \delta_t^{(1)} h_t^T \tag{13}$$
$$\left.\frac{\partial J_t}{\partial b_h}\right|_t = \delta_t^{(1)} \tag{14}$$

The reason that Equations 12, 13, and 14 are evaluated at time t is because via the chain rule, those gradients will be further updated via $\frac{\partial J_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W_x}$ and so on.

Observe that a naive implementation would therefore be quadratic in the size of the minibatch L since computing $\frac{\partial J_t}{\partial W_x}$ would require a full pass through the minibatch computing all of the partial derivatives with respect to each hidden state. This would occur for J_1 through J_L.

With a little bit of care, though, all of the gradient updates can occur in a single pass through the data. The key is the computation in Equation 11 of $\frac{\partial J_t}{\partial h_{t-1}}$. Then, as we pass backwards in time, we can add this result to that of Equation 9, thereby computing $\frac{\partial J_t}{\partial h_{t-1}} + \frac{\partial J_{t-1}}{\partial h_{t-1}}$. It's clear, then, that as we unroll the time steps, each of the subsequent computations will in fact be computing, e.g., $\sum_{l=t}^{L} \frac{\partial J_l}{\partial W_x}|_t$ and so the gradients will be fully updated by the time we reach the first time step.

The forward pass equations provide the runtime bounds at a single time step: $O(VH + H^2)$, where the dominating term depends on the size of V compared to the size of H.

Thus, in each minibatch we achieve a cost of $O(VHL + H^2 L)$. Overall, given that N words are distributed over P machines and C clusers per machine, this yields a cost of $O(\frac{VHN}{CP} + \frac{H^2 N}{CP})$ if the sequences are split evenly across cores and trained sequentially on each core.

Note that if we instead wanted to train sequentially on the full text per machine (without introducing the additional C break points per machine), we could then use the C cores to speed up the vector-matrix multiplies. The depth of those operations would be bounded by $O(\log H)$ from Equation 4 or $O(\log V)$ from Equation 9 using the tree-based parallel summation procedure, leading to a cost on C cores of $O(\frac{H^2}{C} + \log H + \frac{VH}{C} + \log V)$.

That would lead to an overall cost of $O(\frac{N}{P}(\frac{H^2}{C} + \log H) + \frac{N}{P}(\frac{VH}{C} + \log V))$.

It is quite straightforward to extend this analysis to a sliding windows-based algorithm. Instead of proceeding through a sequential block of text in batches of size L, each window is of size T and is processed via the same forward and back-propagation scheme. We assume that separate map tasks are performed sequentially on cores.

It therefore follows directly that since we will have $O(NT)$ total sliding windows due to the windows overlapping, the total runtime when processing all sliding windows is $O(\frac{VHNT}{CP} + \frac{H^2NT}{CP})$, a factor of T more runtime than the sequential train version.

Sampling α percent of the sliding windows to train in each iteration will just add an α to the above runtimes: $O(\alpha\frac{VHNT}{CP} + \alpha\frac{H^2NT}{CP})$.

4.2 Communication Cost

One nice property of the neural network setup is that the only communication that needs to occur is an All Reduce on the weight matrices in between iterations. The gradients are computed separately on each core of each machine and, in the case of the longer sequential trains, are used to locally update the weight matrices during an iteration. Then, the resulting weight matrices need to be averaged across cores and machines, which is where the All Reduce comes in.

The total amount of data that crosses the network per iteration is $O(VHP + H^2P)$, assuming that local combiners are used to average the weight matrices from the different machine cores. Since the communication occurs in a BitTorrent-like manner, the time cost of this communication is $O(VH\log_2 P + H^2\log_2 P)$.

Note that if there were no combiners and every mapper sent its output onto the network, the communication costs would vary between the different methods. A per-core sequential train, for example, would send $O(VHPC + H^2PC)$ data onto the network since each machine core would have output. Meanwhile, a sampled sliding windows approach would send out $O(VH\alpha\frac{NT}{C} + H^2\alpha\frac{NT}{C})$ data onto the network, again corresponding to the number of map tasks.

Observe that in practice, there isn't necessarily a need to keep the weight matrices in sync across machines and cores every single operation. If bandwidth is low and/or latency is high, a better option may be to continue locally updating weight matrices for multiple iterations before pursuing the All Reduce. One of the empirical experiments we run is how this affects convergence.

4.3 Optimization

All of the analysis we've done so far has been on a per-iteration basis. How many iterations will it take to converge? Note that in practice, language models are not formally trained with a stopping criterion of convergence but rather are kept running until the results look good.

It's worth noting up front that there isn't a direct mapping with the standard gradient descent / stochastic gradient descent literature here. Most analyses assume that the gradient is separable across observations, which is not the case here due to the sequential nature of the model. Thus, every version of the parallelized model, and even the standard sequential models with truncated backpropagation, are only approximating the gradient over thheir particular sequence of data.

Besides the non-separability of the gradient, we also violate the standard model of GD/SGD by sometimes performing local parameter updates within an iteration instead of just calculating gradients on fixed inputs and weights.

That being said, the ultimate averaging of weights comes pretty close to a "true" gradient descent step in practice, so it makes sense to approximate the number of iterations it takes to convergence us-

Successful reproduction of sentence	Unique combinations from corpus
And yet the bullshit you choose may be harder to eliminate than the bullshit that's forced on you.	Otherwise these people are litably forced on you, the bullshit that sneaks into your life by tricking you is no one's fault but your own.

Table 1: Example of character-level RNN output.

ing the standard optimization theory. We avoid considering the possibility of Hogwild optimization since our implementation in Spark synchronously updates parameters each step.

Thus, we estimate that the full gradient approximation of sequentially training on each machine will require $O(\log \frac{1}{\epsilon})$ iterations and the SGD approximation of sampling from sliding windows will take $O(\frac{1}{\epsilon})$ iterations, where ϵ is the desired convergence level. The cost of averaging the parameters requires $O(VHP + H^2P)$ work and $O(\log P)$ depth, again assuming combiners, which pales in comparison to the cost of an iteration.

5 Experiments

We implement our character-level RNN model on Spark using Scala. In the following experiments, the RNN model is given a document as corpus and the task is to recover vocabulary and writing style on a character-by-character basis. A sample of our RNN model input and output are shown in Table. 1 where the results are obtained after training our model with Paul Graham's article.

5.1 Data

We compare the performance of distributed training algorithms proposed in sections 3 and 4 using two different corpora: the *Paul Graham corpus* and *Wikicorpus* [5]. The *Paul Graham corpus* is an article by Paul Graham which contains 8.5 thousand characters; we associate this as a small corpus. The *Wikicorpus* contains Wikipedia contents based on a 2006 dump. We took a portion of *Wikicorpus* which contains 22.3 million characters as our training corpus. We consider this a large corpus in terms of character-level models.

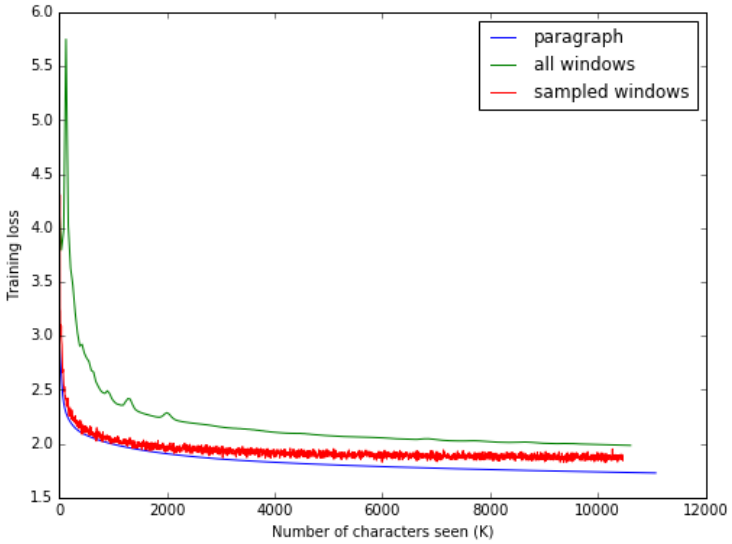
5.2 Distributed Training Algorithms

We experiment with three different distributed training algorithms. The first algorithm distributes corpora as an RDD of paragraphs. Within each paragraph we train sequentially. As discussed above, we use this as an easily implementable approximation of the sequential train per machine approach. The second algorithm transforms corpora into an RDD of sliding windows as proposed in previous sections; sliding windows are trained in parallel with a gradient descent manner. The third algorithm modifies the previous where a random sample of sliding windows are trained in each iteration.

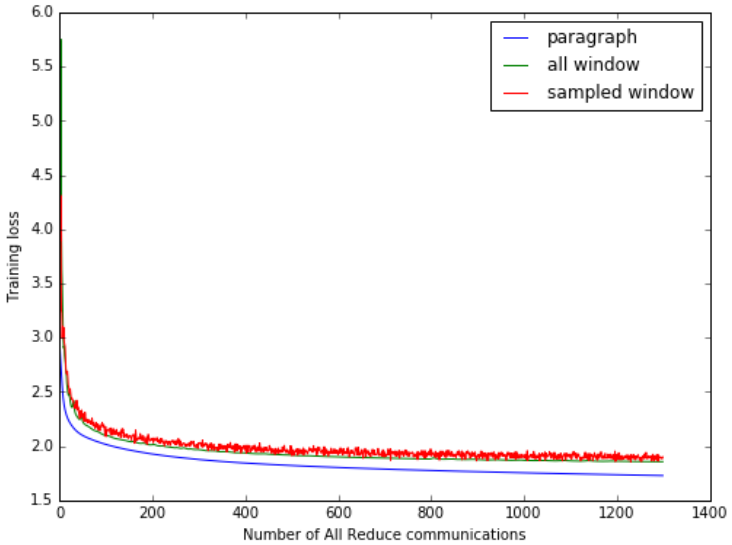
We first run the *Paul Graham corpus* with $8.5K$ characters on sequential paragraph training, all sliding window training, and sampled sliding window training. The results are shown in Fig. 3. We measure the performance of algorithms by average per-character training loss. Since our goal is training efficiency it's reasonable to focus on training loss only.

Number of characters seen is an estimation of total work incurred by each algorithm. With the same amount of work, the sampled sliding window algorithm outperforms the all sliding window algorithm. The result comes with no surprise as sliding windows overlap with each other and sampling among sliding windows boosts computational efficiency while sacrificing little in terms of training efficacy. Sequential paragraph training has the best loss performance, showing the importance of preserving sequence order in RNN models.

To compare our algorithms in a distributed environment where communication is the bottleneck, we estimate communication cost by the number of All Reduce communications incurred and demonstrate the comparison in Fig. 3b. With the same number of All Reduce communications, the all sliding window and sampled sliding window algorithms have similar loss performance. However, taking into consideration the smaller communication size for the sampled approach (in the case of



(a) Performance measurement using number of characters seen. Number of characters seen is an estimation of total work.

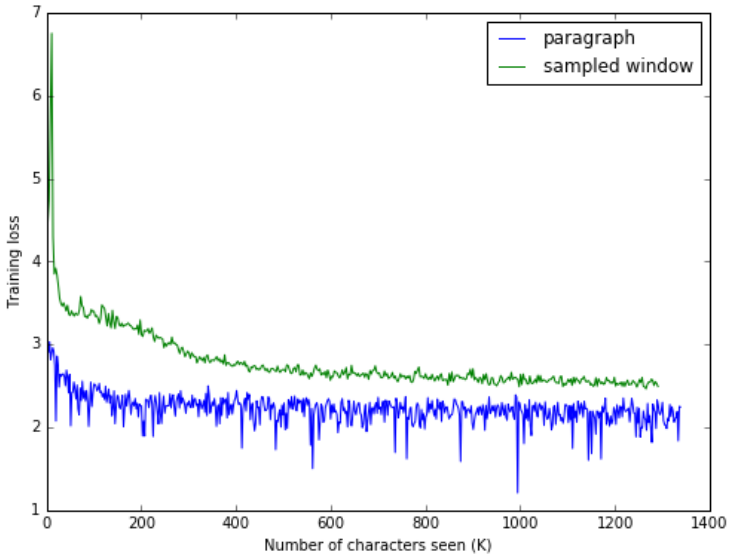


(b) Performance measurement by number of All Reduce communications incurred.

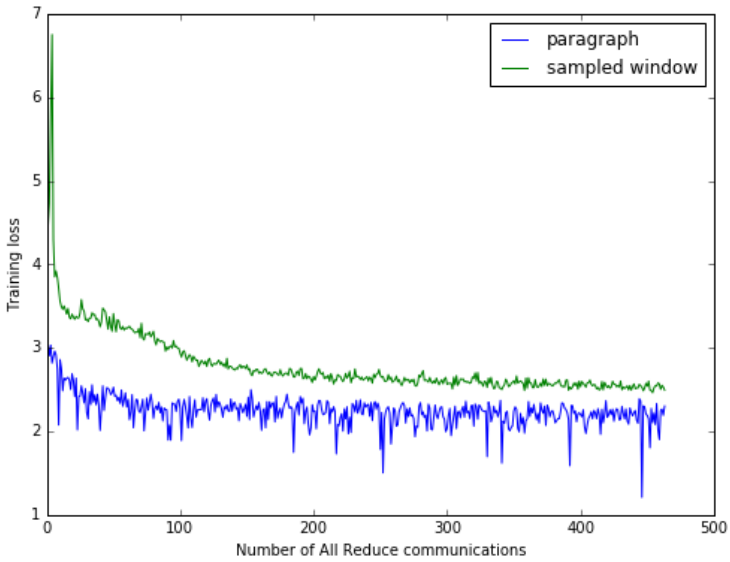
Figure 3: All RNN models have 1 layer and 25 hidden units. Paragraph training takes a subsequence of length 25 each time. Sliding windows have length 5. Random sampling samples 1% of sliding windows each time.

no combiners) or reduced combiner workload (in the case of combiners), sampled sliding windows is preferred compared to all sliding windows.

For corpora as large as *Wikicorpus* we utilize sampling techniques to enhance training efficiency. We run a modified sequential paragraph training algorithm where a sample of paragraphs are trained in each iteration. We compare sampled paragraph training and sampled sliding window training as shown in Fig. 4. The comparisons are based on number of characters seen and number of communications. Sampled paragraph training achieves better loss performance in terms of number of characters seen and number of communications, again demonstrating the importance of preserving sequence order and passing memory states within paragraphs.



(a) Performance measurement using number of characters seen.



(b) Performance measurement by number of All Reduce communications incurred.

Figure 4: All RNN models have 1 layer and 25 hidden units. Paragraph training takes a subsequence of length 25 each time. Sliding windows have length 25. Random sampling samples 0.01% of paragraphs and 0.0005% of sliding windows each time.

5.3 Modified Paragraph Training

It is possible to further enhance the performance of sequential paragraph training by controlling communication. Before communicating with other machines, each machine can iterate through its paragraph multiple times such that overall communication cost can be reduced. With *Wikicorpus* we experiment on how the number of iterations before communication affects loss performance. The results are shown in Fig. 5.

Experiment results show multiple iterations before communication may slightly slow down the rate of loss convergence. However, as the training process goes on the difference between single iteration

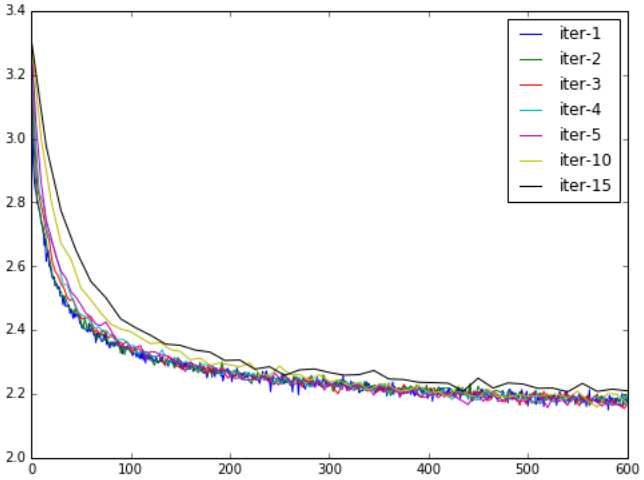


Figure 5: Loss performance for different number of iterations before All Reduce communication under sequential paragraph training algorithm.

and multiple iteration narrows. This indicates a design choice in designing RNN training algorithms. With different environments and different network latency conditions, it is possible to adjust the number of iterations before communication to achieve optimal efficiency.

6 Conclusion

We examined the efficacy of a variety of methods to take advantage of distributed computing in the training of RNN-based language models. Theoretically, there are reasons to believe that naively parallel sequential models and sliding window-based models might be faster and more accurate than each other, as well as a directly sequential model. We test these techniques' efficiencies using a complete ground-up implementation of an RNN in Apache Spark. We find that naively parallel sequential models outperform sliding window approaches in terms of loss convergence performance on total work and number of communications. We propose further enhancement of parallel sequential algorithms by controlling the number of local iterations before communication. In an environment where communication costs are high, it is beneficial to iterate multiple times locally before communication.

References

[1] Chelba, C., Mikolov, T., Schuster, M., Ge, Q., and Brants, T. (2013). One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. arXiv preprint. arXiv:1312.3005

[2] Karpathy, A. (2015). Minimal Character-Level Language Model with a Vanilla Recurrent Neural Network. https://gist.github.com/karpathy/d4dee566867f8291f086

[3] Mikolov, T., Karafiat, M., Burget, L., Cernocky, J., and Khudanpur, S. (2010). Recurrent Neural Network Based Language Model. INTERSPEECH 2, 3.

[4] Williams, W., Prasad, N., Mrva, D., Ash, T., and Robinson, T (2015). Scaling Recurrent Neural Network Language Models. arXiv preprint. arXiv:1502.00512v1

[5] Samuel R., Gemma B., Montse C., Lluis P., German R. (2010). *Wikicorpus*: A Word-Sense Disambiguated Multilingual Wikipedia Corpus. LREC'10