DISTRIBUTED TRAINING OF LARGE SCALE EXPONENTIAL LANGUAGE MODELS

Abhinav Sethy, Stanley F. Chen, Bhuvana Ramabhadran

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA

{asethy, stanchen, bhuvana}@us.ibm.com

ABSTRACT

Shrinkage-based exponential language models, such as the recently introduced Model M, have provided significant gains over a range of tasks [1]. Training such models requires a large amount of computational resources in terms of both time and memory. In this paper, we present a distributed training algorithm for such models based on the idea of cluster expansion [2]. Cluster expansion allows us to efficiently calculate the normalization and expectations terms required for Model M training by minimizing the computational resources in terms of both time and memory. We also show how the algorithm can be implemented in a distributed environment, greatly reducing the memory required per process and training time.

Index Terms— Language modeling, exponential n-gram models, distributed training

1. INTRODUCTION

Exponential n-gram models comprise a powerful statistical framework that can model complex dependencies and utilize rich feature spaces [3]. In [4], we proposed a novel class-based exponential language model, Model M, that significantly outperforms state-of-the-art language models in automatic speech recognition word-error rate over a wide range of domains, including Wall Street Journal, Hub4 English Broadcast News and GALE Arabic [1]. However, this performance comes at a steep computational cost: our current training implementation requires around 30 hours of computation and 10GB of memory for 100M words of training data on a typical Xeon CPU. With training sets of a billion words and more not uncommon, training can easily become prohibitive.

In this paper, we describe an efficient distributed training algorithm for Model M which allows us to vastly reduce the memory required per process and training time. We begin by reviewing exponential language models and Model M in Section 2. We describe how we use cluster expansion [2] inside of the Model M training framework to compute feature expectations efficiently in Section 3. In Section 4, we present our parallelization strategies, one of which divides the model features across compute nodes and the other which divides the training data. In Section 5, we describe experiments and results on a 330M-word English Broadcast News corpus and a 1.6G-word GALE Arabic corpus, showing how our proposed method scales with training data size and number of compute nodes. We finish up with some discussion in Section 6.

2. BACKGROUND

In this section, we briefly review exponential language models in the context of Model M. For a set of target symbols $y \in Y$ and input symbols $x \in X$, an exponential model with parameters $\Lambda = \{\lambda_i\}$ and corresponding features $f_i(x, y)\ldots f_\ell(x, y)$ has the form

$$P_\Lambda(y|x) = \frac{\exp(\sum_i \lambda_i f_i(x, y))}{Z(x)}$$

(1)

$$Z(x) = \sum_y \exp(\sum_i \lambda_i f_i(x, y))$$

(2)

We refer to $x$ as the event history and $y$ as the target. In language models, the event history $x$ is typically composed of a series of words or tokens $w_1w_2\ldots w_{j-1}$.

In an exponential n-gram model (for $n = 3$), we have binary features $f_{(x,y)}(\cdot)$ for $(x, y)$ of the forms

$$(\epsilon, w_j), (w_{j-1}, w_j), (w_{j-2}w_j - 1, w_j)$$

where $f_{(x,y)}(x, y) = 1$ iff the event history $x$ ends in $x$ and the target word $y$ is $y$. Although the number of possible features for an n-gram model is $V^n$ where $V$ is the size of the word vocabulary, we only consider features that correspond to n-grams that occur at least once in the training corpus. Since the number of n-grams seen in the training data is typically much smaller than the number of possible n-grams, the feature space for n-gram models can be considered ‘sparse’. For any given event history $x$, we expect that for most target words there will be only one non-zero feature, corresponding to the history (i.e., a unigram feature).

We train exponential n-gram models using a combination of $\ell_1$ and $\ell_2^2$ regularization [5], i.e., parameters $\lambda_i$ are chosen to optimize

$$\mathcal{O}_{\ell_1+\ell_2^2}(\Lambda) = \log \text{PP}_{\text{train}} + \frac{\alpha}{D} \sum_i |\lambda_i| + \frac{1}{2\sigma^2 D} \sum_i \lambda_i^2$$

(3)

for some $\alpha$ and $\sigma$, where $\text{PP}_{\text{train}}$ is training set perplexity and $D$ is the size of the training set in words [1]. Updating the model parameters in each training iteration requires a comparison of expected feature counts and observed counts. For a feature $f_{(x,y)}(\cdot)$, the expected feature count can be computed as

$$E(f_{(x,y)}) = \sum_{(x_d, y_d) \in \mathcal{D}} \frac{\exp(\sum_{i=1}^{\ell} \lambda_i f_i(x_d, y_d))}{Z(x_d)}$$

(4)

where the training data $\mathcal{D}$ is expressed as a sequence of event histories and targets $(x_d, y_d)$. Given feature expectations, it is generally inexpensive to update the $\lambda_i$’s using iterative scaling [6] or another iterative update algorithm.

In this paper, our focus is scalable, distributed training for Model M, an exponential class-based n-gram model that can be viewed as the result of shrinking an exponential word n-gram model using
word classes [1, 4]. If we assume each word \( w \) is mapped to a single class \( c(w) \), we can write
\[
p(w_1 \cdots w_l) = \prod_{j=1}^{l} p(c_j|c_1 \cdots c_{j-1}, w_1 \cdots w_{j-1}) \times 
\prod_{j=1}^{l} p(w_j|c_1 \cdots c_j, w_1 \cdots w_{j-1})
\]
(5)
where \( c_{l+1} \) is the end-of-sentence token. Let \( p_{bg}(y|\lambda) \) denote an exponential \( n \)-gram model as defined earlier. Let \( p_{bg}(y|\lambda_1, \lambda_2) \) denote a model containing all features in \( p_{bg}(y|\lambda_1) \) and \( p_{bg}(y|\lambda_2) \). Then, we can define (the trigram version of) Model M as
\[
p(c_j|c_1 \cdots c_{j-1}, w_1 \cdots w_{j-1}) = \prod_{j=1}^{l} p_{bg}(c_j|c_{j-2}c_{j-1}, w_{j-2}w_{j-1})
\]
\[
= \prod_{j=1}^{l} p_{bg}(w_j|w_{j-2}w_{j-1}c_j)
\]
(6)
That is, Model M is composed of two exponential models, a class prediction model containing features from two exponential \( n \)-gram models, and a word prediction model that is a simple exponential \( n \)-gram model; these models can be trained independently.

3. EFFICIENT EXPECTATION COMPUTATION

Here, we will describe our approach for efficient training on a single machine and then present parallelization strategies for training in a distributed environment in the next section. As noted earlier, the bulk of the training computation involves using eq. (4) to compute feature expectations for each feature in each training iteration. Note that this entails computing the normalization term \( Z(x_d) \) for each training set event history \( x_d \), where \( Z(x_d) \) is a sum over scores for all words in the target vocabulary as given in eq. (2). Thus, a naive implementation requires \( |Y| \) operations per training iteration where \( |Y| \) is the number of unique event histories and \( |Y| \) is the size of the target vocabulary. Similarly, a direct implementation of eq. (4) to compute feature expectations would also require \( |Y| \) operations per training iteration. For large training sets, \( |Y| \) can be 100M or more and \( |Y| \) can be 100k or more, and thus the naive implementation is impractical.

Feature expectations and normalization terms can be computed efficiently using cluster expansion, introduced by Lafferty and Suhm [2] for maximum entropy models. In [3, 7], the authors describe an efficient scheme based on cluster expansion for training exponential \( n \)-gram models where they propagate the normalization sums from lower order to higher order \( n \)-grams and expectations from higher order to lower order \( n \)-grams. Our approach for computing expectations is similar, but instead of using the nesting of lower and higher order \( n \)-grams for efficient computation, we rely on sorting the \( n \)-gram events in the corpus in a particular order and computing only the differentials between consecutive sorted events. This allows us to train the model without explicitly keeping the full tree of \( n \)-gram features in memory. Our approach also leads to a simpler formulation for models containing multiple sets of \( n \)-gram features (like the class prediction model in Model M), which is handled in the previous approach by considering conjunctions of features between \( n \)-gram feature sets. The two approaches, however, are comparable in terms of their computational complexity, which is linear in the number of features in the model.

3.1. Normalization Term Computation

We first describe our approach to cluster expansion for the normalization term computation. For any two event histories \( x_1 \) and \( x_2 \), we can write the difference between their associated normalization terms as
\[
Z(x_1) - Z(x_2) = \sum_{y \in Y} (\alpha(x_1, y) - \alpha(x_2, y))
\]
(7)
where we define \( \alpha(x, y) = \exp(\sum_{i=1}^{n} \lambda_i f_i(x, y)) \), or the numerator on the right-hand side in eq. (1).

We note that for “sparse” models, many (if not most) of the terms in the sum on the right of eq. (7) will be 0 for most \( x_1 \) and \( x_2 \). For example, for an infrequent word \( y \), only the unigram feature \( f_{1,1}(\cdot) \) will be active for most \( x_1 \) and \( x_2 \), in which case \( \alpha(x_1, y) = \alpha(x_2, y) \). As noted earlier, typical \( n \)-gram models can be considered to be sparse. Then, for the sequence of training set histories \( x_1, x_2, \ldots, x_D \), we can compute normalization terms efficiently by just computing the difference between \( Z(x_d) \) and \( Z(x_{d+1}) \). To reduce computation further, we sort the corpus such that consecutive histories differ only in the words farthest from the current position, thus reducing the number of target words to sum over.

Specifically, let us define a feature history \( x \) for each \( x \) occurring in a feature of the form \( f_{1\times Y}(\cdot) \). For an exponential \( n \)-gram model, for instance, there will be a feature history for each bigram, unigram, and empty history (i.e., \( e \)) occurring in the training data. The set of \( n \)-gram feature histories can be viewed as forming a tree with the empty history at the root. We assign each feature history a unique ID by doing a prefix traversal of the tree, so that lower-order histories are assigned lower ID’s. Then, with each training event history \( x_d \), we can associate a list of active feature history ID’s, sorted in ascending order. We sort event histories by sorting the associated (sorted) feature history ID lists lexicographically. In this way, consecutive event histories will tend to have very similar sets of active feature histories, so that \( Z(x_{d+1}) - Z(x_d) \) can be computed efficiently. That is, consecutive event histories will tend to differ only on feature histories with high ID’s, which tend to correspond to higher-order \( n \)-gram histories, which tend to co-occur in fewer features \( f_{1\times Y}(\cdot) \).

Once event histories are sorted, we can compute all normalization terms \( Z(x_d) \) in each training iteration by processing each event history in turn. For each event history \( x_d \), we keep track of the values \( \alpha(x_d, y) \) for all \( y \in Y \) as well as \( Z(x_d) = \sum_{y \in Y} \alpha(x_d, y) \). Moving from one event history to the next, we identify all feature histories \( x \) that differ, and compute \( \alpha(x_{d+1}, y) - \alpha(x_d, y) \) for only those \( x \) such that the feature \( f_{1\times Y}(\cdot) \) exists for one of these \( x \). For all other \( y \), we have \( \alpha(x_{d+1}, y) = \alpha(x_d, y) \), and \( Z(x_{d+1}) \) can be computed efficiently using eq. (7).

3.2. Expectation Computation

Expectations can be efficiently calculated using cluster expansion, using a similar approach as in the normalization term computation. Given a feature \( f_{1\times Y}(\cdot) \), from eq. (4) we see that its expectation is the sum of \( \alpha(x_d, y)/Z(x_d) \) over those training event histories \( x_d \) where the feature history \( x \) is active. Now, imagine we have a sequence of event histories \( x_{d_1}, \ldots, x_{d_2} \) such that \( x \) is active in each history and \( \alpha(x_d, y) \) is constant. Then, we have
\[
\sum_{d=d_1}^{d_2} \frac{\alpha(x_d, y)}{Z(x_d)} = \alpha(x_{d_1}, y) \times \sum_{d=d_1}^{d_2} Z^{-1}(x_d)
\]
(8)
By taking advantage of this property, instead of updating feature expectations for every \( y \) at every event history \( x_d \), we need only update

---

1In models with multiple sets of \( n \)-gram features, we can form a single tree by connecting each separate feature tree to a common root node.
feature expectations for those \( y \) such that \( \alpha(x_{d}, y) \neq \alpha(x_{d-1}, y) \).

(More precisely, we actually need to do this for all \( y \) such that the set of features active for the event \((x_{d}, y)\) changes.) Note that the sum \( \sum_{k=0}^{d} Z^{-1}(x_{d-k}) \) is independent of \( y \) and thus can be shared across target words. In addition, note that the sorting of event histories performed for the normalization term computation attempts to change as few \( \alpha(x_{d}, y) \) as possible between consecutive event histories; thus, this sorting makes the expectation computation more efficient as well.

Specifically, we compute feature expectations by processing each event history \( x_{d} \) in turn, keeping track of active features for each \((x_{d}, y)\) (usually just the unigram feature for most words). For each \( x_{d} \), we first compute \( Z(x_{d}) \) using the algorithm described earlier. Let \( Z_{d}^{-1} = \sum_{d'=0}^{d} Z_{d'}^{-1}(x_{d'}) \), the sum of inverse normalizers so far, and let \( Z_{d'}^{-1}(y) = Z_{d'}^{-1} \) for the most recent event history \( x_{d'} \) such that \((x_{d'}, y)\) and \((x_{d}, y)\) have different sets of features active. Then, for each \( y \) such that \((x_{d}, y)\) and \((x_{d+1}, y)\) have different sets of features active and for each feature \( f_{(x,y)}(\cdot) \) active for event \((x_{d}, y)\), we update the running total for \( E(f_{(x,y)}) \) by adding the quantity \( \alpha(y)/Z_{d}^{-1} - Z_{d'}^{-1}(y) \). This has the effect of identifying maximal event history sequences where eq. (8) can be applied, and applying it. Once we have the feature expectations \( E(f_{(x,y)}) \) for a training iteration, we can use unnormalized iterative scaling [1] or some other update algorithm to update the parameters \( \lambda_{i} \) to optimize the objective function described in eq. (3).

3.3. Creating the corpus of sorted event histories

To sort training set event histories for the normalization and expectation computations, we associate a sorted list of feature history ID’s with each event history (to be used as the sort key). To compute the list of feature history ID’s for an event history, we need to be able to quickly map from a feature history to its ID. To store this mapping compactly, we use a minimal perfect hashing function (MPHF) [8]. Minimal perfect hashing functions are a class of functions that map \( N \) unique keys to the range 1, \ldots, \( N \) with no collisions. We first create the list of all unique feature histories and build a minimal perfect hashing function. Since the training data is a closed set of events, we do not require any signature bits to avoid hash collisions. However, the MPHF assigns random ID’s to feature histories, and recall that we require that lower-order feature histories receive lower ID’s. Thus, we build a lookup table to map the ID’s assigned by the MPHF to a new set of ID’s, where histories occurring in more events in the training corpus are assigned lower ID’s. Using the MPHF allows us to build the sorted corpus in a memory-efficient way since the MPHF can be represented using around 4 bits/entry.

4. PARALLELINIZING MODEL M TRAINING

Cluster expansion makes the training of binary-valued nested exponential language models practical by reducing the computation required from \( O(|H| \cdot |Y|) \) operations per training iteration (where \(|H|\) is the number of distinct event histories and \(|Y|\) the target vocabulary size) to a factor of the number of features in the model. Still, the computational requirements are pretty demanding when training large language models on a single machine. However, the training process can be parallelized by splitting the target word vocabulary or the training corpus across machines. Splitting just the expectation computation across several machines by partitioning the training corpus and summing expectations for each partition has been previously investigated in [9, 10].

The computation of normalization terms as well as feature expectations can be distributed by splitting the target vocabulary into a number of partitions. The normalization term computation requires a sum over all target words and can be computed by summing the normalization terms computed over each vocabulary partition. Note that each of our features \( f_{(x,y)}(\cdot) \) is active for only a single target word \( y \); thus, we can use the vocabulary partition to also partition our feature set. The expectation computation can thus be split perfectly by splitting the target words across machines, with each machine computing expectations for only those features corresponding to its subset of the vocabulary. Since we need only store a subset of the model features and parameters on each machine, the memory required on each machine is reduced by a factor equal to the number of machines. In addition, with iterative scaling, parameter reestimation can also be split across machines since the feature expectation computations are disjoint. However, we still need a master process to merge the partial normalization terms from each machine and to broadcast the complete normalization terms back.

In addition to splitting the target vocabulary, we can also partition the (sorted) training corpus. We distribute each partition of the training corpus to a different machine. Unlike in the previous approach, the normalization term computation for a given event history can be done on a single machine with no merge required. Each worker machine needs to load parameters for the features corresponding to the set of event histories it will process. Since there is considerable overlap in the lower-order features active for each subcorpus (e.g., all partitions will need access to all unigram features), the memory required for storing the model is not partitioned efficiently. In addition, since the features are not perfectly partitioned, we need to merge feature expectations across multiple machines, and parameter reestimation also cannot be split across machines.

5. EXPERIMENTS

We present results on the Hub4 English Broadcast News and Arabic Gale task. In this paper our goal is to show that the distributed Model M training can efficiently handle large data sets and thus we present build times/memory requirements when building a single flat Model M using all the data available in these tasks instead of interpolating individual sources. For the best WER results obtained with Model M using interpolated models built on different sources on both the English BN and Arabic Gale task the reader is referred to [1].

The LM training text for the English Hub4 system consists of 335M words from the following data sources: 1996 CSR Hub4 Language Model data, EARS BN03 closed captions, GALE Phase 2 Distillation GNG Evaluation Supplemental Multilingual data, Hub4 acoustic model training transcripts, TDT4 closed captions, TDT4 newswire, and GALE Broadcast Conversations and GALE Broadcast News. For Arabic, the language model data was gathered by LDC for the purpose of Arabic broadcast transcription for DARPA’s Global Autonomous Language Exploitation (GALE) program. The total amount of text was about 1.6 billion words, collected from 20 sources such as transcripts of audio data, parts of the Arabic Gigaword corpus, Arabic side of machine translation training text, news groups, and web logs, etc.

We start by comparing a direct implementation of normalization sum computation against our proposed cluster expansion approach. The compute time required for computing feature expectations in one iteration of training the word prediction model are presented in Table 1. The corpus used for the results in Table 1 is the Hub4 AM training corpus with 1.5M words and a vocabulary of 87K. The unigram caching approach presented in Table 1 precomputes the nor-
malization sum contribution and feature expectations for unigram features. Since for any give event history most target words will only have the unigram feature active, this simple approach achieves a considerable speedup over the naive implementation. However using cluster expansion as described in Section 3 provides an order of magnitude speed up over unigram caching.

<table>
<thead>
<tr>
<th>Table 1. Compute time (in seconds) for estimating feature expectations with a direct implementation of Equation 4, unigram feature caching, and the proposed method (Section 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compute time</strong></td>
</tr>
<tr>
<td>Direct implementation</td>
</tr>
<tr>
<td>Unigram caching</td>
</tr>
<tr>
<td>Proposed Approach</td>
</tr>
</tbody>
</table>

We now present the compute node memory requirements and model build time for both the English Broadcast news and Arabic Gale task using parallelization by partitioning of target vocabulary as described in Section 4. The alternate approach of splitting by corpus provided comparable compute time speed up but as expected was less efficient in reducing memory. For all our experiments feature expectations on individual compute nodes are computed using the approach described in Section 3 while model parameters are updated using the unnormalized iterative scaling method with regularization as described in [1]. For both the tasks we built a model with 500 classes. The vocabulary size of the English model was 80K and 795K for the Arabic model. For the English Model M the class prediction model had 238M features with 86M distinct event histories and the word prediction model has 276M features with 97M distinct event histories. For the Arabic Gale Model M the word prediction model has 940M features with 410M distinct event histories and the class prediction model has 750M features with 390M distinct event histories. The compute node memory requirements are shown in Table 2 for both the tasks. As we can see the memory requirement for building both the class and the word prediction model scales down almost linearly with the number of compute nodes.

<table>
<thead>
<tr>
<th>Table 2. Memory requirement (GB) per compute node</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of compute nodes</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>English BN (Class Model)</td>
</tr>
<tr>
<td>English BN (Word Model)</td>
</tr>
<tr>
<td>Arabic Gale (Class Model)</td>
</tr>
<tr>
<td>Arabic Gale (Word Model)</td>
</tr>
</tbody>
</table>

Table 3 shows the compute time in hours for training word and class prediction components of Model M.

<table>
<thead>
<tr>
<th>Table 3. Compute time in hours for training word and class prediction components of Model M</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of compute nodes</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>English BN (Class Model)</td>
</tr>
<tr>
<td>English BN (Word Model)</td>
</tr>
<tr>
<td>Arabic Gale (Class Model)</td>
</tr>
<tr>
<td>Arabic Gale (Word Model)</td>
</tr>
</tbody>
</table>

6. CONCLUSION

In this paper we presented a scalable distributed training algorithm for exponential n-gram models especially the recently introduced class based language model, Model M. We presented a cluster expansion scheme similar to [7] which allows us to efficiently compute the normalization sum and expectations required for training Model M. We also show how the algorithm can be used in a distributed computation environment allowing us to reduce the training memory requirements by a factor of the number of available machines and also a get sub linear reduction in compute time.

7. REFERENCES