INTERSPEECH 2009 BRIGHTON

Back-Off Language Model Compression

Boulos Harb, Ciprian Chelba, Jeffrey Dean, Sanjay Ghemawat

Google, Inc.
1600 Amphiteatre Pkwy, Mountain View, CA 94043, USA
{harb,ciprianchelba,jeff,sanjay}@google.com

10.21437/Interspeech.2009-113

Abstract

With the availability of large amounts of training data relevant to speech recognition scenarios, scalability becomes a very productive way to improve language model performance. We present a technique that represents a back-off n-gram language model using arrays of integer values and thus renders it amenable to effective block compression. We propose a few such compression algorithms and evaluate the resulting language model along two dimensions: memory footprint, and speed reduction relative to the uncompressed one. We experimented with a model that uses a 32-bit word vocabulary (at most 4B words) and log-probabilities/back-off-weights quantized to 1 byte, respectively. The best compression algorithm achieves 2.6 bytes/n-gram at ≈18X slower than uncompressed. For faster LM operation we found it feasible to represent the LM at ≈4.0 bytes/n-gram, and ≈3X slower than the uncompressed LM. The memory footprint of a LM containing one billion n-grams can thus be reduced to 3-4 Gbytes without impacting its speed too much.

Index Terms: speech recognition, language model, compression, storage

1. Introduction

In recent years language modeling has witnessed a shift from advances in core modeling techniques (in particular, various n-gram smoothing algorithms) to a focus on scalability. The main driver behind this shift is the availability of significantly larger amounts of training data that are relevant to automatic speech recognition (ASR) scenarios. It is undoubtedly a very productive way of improving the performance of a language model (LM): our experiments on Google Search by Voice show that pruning a 3-gram LM to 10^{-3} of its original size doubles its perplexity. As a result, being able to store large LMs compactly while maintaining reasonable speed is a worthwhile goal, whether the LM is used in the 1-st pass or in lattice rescoring.

The paper presents an attempt at lossless compression of back-off n-gram LMs. To clarify, we are not concerned with pruning techniques [1, 2] that reduce the size of the LM at little degradation in performance, although integrating the two is certainly an area that deserves more exploration.

When attempting to compactly represent a language model, there are two orthogonal directions to be pursued:

- compress the *LM skeleton* representation—the set of n-grams stored in the model
- compress the *payload*—the log-probability (LogP) and back-off weight (BoW) stored with each n-gram entry

As an overall performance metric we use the *LM representation rate* (RR): the average number of bytes per n-gram (B/n-gram) of the compacted LM.

The rest of the paper is organized as follows: the next section reviews related work. Section 3 describes a twist on the well established trie data structure used for storing n-gram LMs. The LM skeleton is stored using arrays of integer values, and provides a dense mapping from n-gram to index in the range $[0 \ldots N-1]$, where N is the total number of n-grams in the LM. The *integer trie* (IT) representation is thus amenable to generic block compression techniques that operate on arrays of integers. In a similar fashion, one can compress the arrays of LogP/BoW values by first quantizing them to 8 bits, and then bundling 4 values as a 32-bit integer, as described in Section 3.2. Section 4 describes three methods explored for lossless block compression of integer arrays, followed by Section 5 describing our experiments. Section 6 concludes the paper and outlines future work directions.

2. Related Work

The use of a trie for storing n-gram back-off language models is well established: the CMU [3], SRILM [4] toolkits, as well as others [5, 6] all rely on it in one form or another. Its refinement using array indexes instead of pointers for the trie representation is also an established idea—it was implemented in later versions of the CMU [7] toolkit, as well as the more recent MITLM [6].

The quantization of LogP/BoW values is also an established procedure for reducing the storage requirements. The work in [5] applies both techniques, and in addition makes the important connection between LM pruning and compression/quantization. By representing LogP/BoWs on variable number of bits (codewords) at each n-gram order, quantizing recursively the differences between actual LogP value and quantized back-off estimate, and removing redundant n-grams using a similar criterion as [1, 2], the authors show that the LM performance on an ASR task can be preserved while dramatically reducing the memory footprint of the model.

All of the above methods represent the model exactly. More recent approaches achieve better compression by using lossy, randomized encoding schemes [8, 9]. These can store parameters in constant space per n-gram independent of either the vocabulary size or n-gram order but return an incorrect value for a 'random' subset of n-grams of tunable size: the more errors allowed, the more succinct the encoding. In the case of [9] n-grams can also be looked up in constant time independent of the compression rate. On the other hand, these schemes cannot easily store a list of all future words for each n-gram context as required by certain applications.

Our approach enhances the standard use of integer arrays to represent the trie by applying block compression techniques in order to reduce the storage requirements for both skeleton and payload. The compression methods used are lossless.

3. Integer Trie for Language Model Representation

We can represent the skeleton for a back-off n-gram LM using two vectors:

- *rightmost word (rmw)*: stores the rightmost word for an n-gram, as we traverse the list of n-grams sorted by order (primary key) and by n-gram (secondary key, numeric sort, each word in an n-gram represented by its index in the vocabulary)
- *right diversity (div)*: stores the number of future words (child nodes in the trie) in a given n-gram context. The n-grams at the highest order have diversity 0, so there is no need to store a diversity count for them.

Trie traversal to identify the index of a given n-gram involves a sequence of $n-1$ binary searches over ranges of the *rmw* vector whose boundaries are specified by the cummulative diversity values. As such we prefer to store the *cummulative right diversity (acc)* vector instead of the *right diversity (div)* vector. The first entry $acc[0] = |V|$ is set equal to the vocabulary size $|V|$. The index assigned to a given n-gram $w_1 \dots w_n$ is the index of the last word w_n in the *rmw* vector traversal.

Listing the future words in a given n-gram context is as simple as pointing to the correct range in the *rmw* vector. The dense mapping from n-grams to integer values in the range $[0, N-1]$, N being the total number of n-grams in the model, also allows storing the LogP/BoW values as arrays of *float*, or b-bit integers if quantized. The inverse mapping from a given integer key to an n-gram involves binary searches over sorted ranges of the *acc* vector whose sizes are the number of n-grams at a given order, respectively, and is more expensive.

As for storage requirements, each n-gram requires two $\log_2(|V|)$ entries: one for the rightmost word, and one for the diversity count. In practice we use 32-bit integers to store the vocabulary, so the raw integer trie storage requirement is 8B/n-gram. Adding two *float* values—LogP and BoW—for each n-gram brings the RR ar 16B/n-gram. Quantizing the LogP/BoW to 1B each, reduces it to 10B/n-gram. Since the highest order n-grams do not have a *acc* entry and neither a BoW value, the actual RR values are slightly smaller, depending on the ratio of n-grams at the maximum order relative to the total number of n-grams in a given model.

3.1. Compressed Integer Trie

Representing each entry in the IT arrays using 32-bit (4B) integers is equivalent to assuming that the sequence of entries in each vector is output by a memoryless source. That is unlikely to be the case however, since there are many regularities in the n-gram stream. Even our toy example shows that each of the two vectors may contain runs of identical values. The *rmw* vector contains entries that are sorted within a sub-range. Similarly, the *acc* vector would be better represented by storing the smaller diversity values, which are very likely to require considerably less bits than $\log_2(|V|)$ as we get deeper in the trie, and are again likely to occur in runs of identical values.

In an attempt to exploit such regularities we explored three different block-compression techniques, which we detail in Section 4. Regardless of the compression technique, the resulting compact array is required to implement a common interface that allows efficient IT lookup:

- get block length
- get array element at index k
- get array elements in range $[\text{start}, \text{end})$
- find range $[v_{\text{start}}, v_{\text{end}})$ of values equal to a specified v, within a pre-specified range $[\text{start}, \text{end})$; the range $[\text{start}, \text{end})$ is assumed to be sorted

The implementation of the fourth method (*equal_range*) is not specific to any given block-compression scheme. In order to make it efficient we cache the array values at block starts and perform the binary search for v, first over blocks (no block decoding necessary) and then over the elements in the block (or range of blocks) that might contain the range $[v_{\text{start}}, v_{\text{end}})$. The size of the cached helper array is included in our computation of the RR.

3.2. Quantized Payload

Quantization of LogP/BoW values has been long used to save storage at very little decrease in language model performance as part of an ASR decoder. We use 8-bit linear quantization for the LogP/BoW values, computing one separate "codebook" for each. The $\min$ and $\max$ values are computed in a first pass, after which each value is represented by its nearest of the 256 8-bit codewords that span the $[\min, \max]$ range.

4. Block Compression for Integer Arrays

We have experimented with three different block compression schemes for arrays of 32-bit unsigned integers, which we detail in the next sections. We assume we are given an array A and a block length b. Each scheme generates a byte-array representation $\hat{A}$ (that is wrapped in an interface implementing the common functions described in Section 3.1).

In each of our block-based encoding schemes below, we first divide the array into a set B of non-overlapping blocks of length b, encode each block, and store the sequence of encoded blocks. We also store an offset array on the side that maps the block number to the beginning of the encoded block. Sections 4.1, 4.2, and 4.3 below focus on how blocks are encoded in each scheme.

A lookup proceeds by computing the block number, looking up the block offset, and finding and decoding the appropriate entry in the block.

4.1. RandomAccess

The *RandomAccess* encoding maintains $O(1)$ access into the encoded array $\hat{A}$. However, as implemented, *RandomAccess* can only encode arrays of monotonically non-decreasing values. Extending it to encode any array is straightforward. Each block is encoded by storing the first value in the block–or the *anchor*–using 4 bytes; then storing the difference between each subsequent element in the block and the anchor using a fixed number β of bytes. For a given block, β is the minimum number of bytes needed to store the difference between the largest (also the last) element in the block and the anchor. Maintaining random access during lookup is now easy since we can compute β for a block by taking the difference between its offset and the following block's offset and dividing by the block size b.

As mentioned in Section 3.1, we can cache the anchors for faster access time especially when performing binary search. The cached anchors add $4|B|$ bytes to the size of $\hat{A}$.

4.2. GroupVar

One drawback of the RandomAccess encoding is that as the block length b increases, even though the number of blocks (thus the number of anchors and boundaries) decreases, the

number of bytes needed to store block elements may increase. In the *GroupVar* encoding we present here the size of the representation $\hat{A}$ gets better as b increases; however, the time required for decoding an element becomes $O(b)$ in the worst case.

This encoding uses a custom representation of delta-encoded values to improve decoding speed. Each value is represented as a two-bit length (0-3 indicating 1-4 bytes), plus that many bytes of data. In order to make decoding fast, the numbers are stored in groups of four, as follows:

length tags: 1 byte
value 1: 1-4 bytes
value 2: 1-4 bytes
value 3: 1-4 bytes
value 4: 1-4 bytes

where the first byte contains the two-bit lengths for the four values. Hence, each group of four values is represented in 5 to 17 bytes. Because the length tags are all in one byte, a 256-element lookup table can give the appropriate masks and offsets for decoding these four values.

4.3. CompressedArray

Each block is converted to a sequence of symbols from the following alphabet:

- `TOGGLE(N)` : toggle last value's N-th bit
- `ADD(N)` : add N to last value
- `ESCAPE(N)` : next N bits form the value
- `REPEAT_LAST(N)` : last value is repeated for next N index entries
- `EXPLICIT(N)` : N is the next value
- `MRU(N)` : use the N-th most recent symbol value

This encoding assumes that the value just before the beginning of each block is zero, so for example an ADD(7) at the beginning of a block expands to 7.

We compute a single Huffman code for the entire array and encode all blocks using this code. The conversion of numbers to symbols is done by making multiple passes over the input. In the first pass, we try all possible encodings of a value and increment a count associated with each possible encoding. E.g., if a value can be encoded as ADD(1) or TOGGLE(1), we increment the counts for these symbols. (The counts can be maintained probabilistically if the input array is large.) At the end of this pass, we build a Huffman table from these counts.

In the next few passes, we refine the count distribution: For each value we pick the minimum length encoding using the Huffman table we built in the previous pass. E.g., if the Huffman table assigned a smaller code to ADD(1) than TOGGLE(1), we increment the count for ADD(1) in the table built by this pass. At the end of the pass we build a new Huffman table with the new count distribution.

In the final pass we encode the values using the Huffman table generated by the previous pass.

5. Experiments

Our experiments used two sets of language models/test data:

- *Switchboard (SWB)*: 4-gram LM containing 908,359 n-grams: 30,170/429,898/245,961/202,330 1/2/3/4-grams, respectively; evaluated on 5,879 sentences/57,657 word tokens; n-gram hit ratios: 0.11/0.40/0.31/0.18
- *Google Search by Voice (GSV)*: 3-gram LM containing 13,552,097 n-grams: 998,846/8,229,305/4,323,946 1/2/3-grams, respectively; evaluated on 10,557 sentences/38,997 word tokens; n-gram hit ratios: 0.08/0.57/0.35

Compression Technique	Block Length	Relative Time	RR (B/n-gram)
None	—	1.0	14.09
Quantized	—	1.0	8.75
CMU 32b, Quantized	—	1.0	7.2
CMU 24b, Quantized	—	1.0	6.2
GroupVar	8	2.4	6.79
	64	3.9	5.09
	256	8.3	4.91
RandomAccess	8	2.6	6.69
	64	3.5	4.98
	256	6.2	5.12
CompressedArray	8	5.3	5.18
	64	18.3	3.38
	256	56.2	3.18

Table 1: Switchboard LM: speed and representation rate (RR) for various compression techniques and block lengths.

For speed measurements, we used a standard Google microbenchmarking setup. For each run we made 10 passes over the test data to perform the standard perplexity calculation. The model and test data loading time are excluded from the calculation. We did not pre-compute any sufficient statistics from the test data either, namely each benchmark iteration involved a LM LogP calculation for each of the tokens indicated above.

The last two paragraphs in Section 2 of [5] detail a technique used in the CMU toolkit which allows one to store the cummulative diversity counts on 2B only. By denoting the total number of n-grams in a model with N, the number of n-grams at the maximum order as N_{max}, the number of 1-grams by N_1 and assuming 1B quantized LogP/BoW values, we arrive at RR$= 8 - 4 \cdot N_1/N - 3 \cdot N_{max}/N$ (B/n-gram) for a LM that uses a 32-bit vocabulary—we assume that the vocabulary is not stored as part of the *rmw* vector, to make sure the comparison with the IT is correct. We also note that while the term due to the vocabulary size $4 \cdot N_1/N$ is relatively significant for the GVS LM (0.3B/n-gram), as the total number of n-grams increases towards 100M-1B, it becomes negligible. Allowing a 24-bit vocabulary (sufficient in practice) reduces the RR by 1B/n-gram; we report this rate as well, although not directly comparable with our experiments since we use 32-bit vocabularies.

We compared the performance of each IT compression scheme at various block lengths. Tables 1, 2 show the results for Switchboard, Google Search by Voice, respectively; Figures 1, 2 summarize the RR as a function of LM speed, measured relative to the uncompressed one.

In its raw form, the integer trie requires slightly less than[1] 16B/n-gram—≈8B/n-gram for the LM skeleton, and ≈8B/n-gram for the payload, assuming *float* representation for LogP/BoWs. Linear quantization of LogP/BoWs to 8-bit codewords reduces the payload to slightly less than 2B/n-gram, and the RR to 8-9B/n-gram. Further reductions can be achieved by block compression of the integer arrays in the integer trie, RR≈3-5B/n-gram. The best result on the LMs we experimented with was RR 3.1B/n-gram for block length 256. The speed of n-gram lookups is negatively impacted by using stronger compression schemes and larger block lengths—16X and 56X slower than uncompressed for the *GSV* and *SWB* models, respectively. A better trade-off for the *GSV* LM achieves

[1] The n-grams at the highest order in a back-off LM do not have any children in the trie, and no associated BoW, only a LogP value.

Compression Technique	Block Length	Relative Time	RR (B/n-gram)
None	—	1.0	13.15
Quantized	—	1.0	8.11
CMU 32b, Quantized	—	1.0	6.7
CMU 24b, Quantized	—	1.0	5.8
GroupVar	8	1.4	6.32
	64	1.9	4.77
	256	3.4	4.60
RandomAccess	8	1.5	6.24
	64	1.8	4.64
	256	3.0	4.59
CompressedArray	8	2.3	4.99
	64	5.6	3.25
	256	16.4	3.07

Table 2: Google Search by Voice LM: speed and representation rate (RR) for various compression techniques and block lengths.

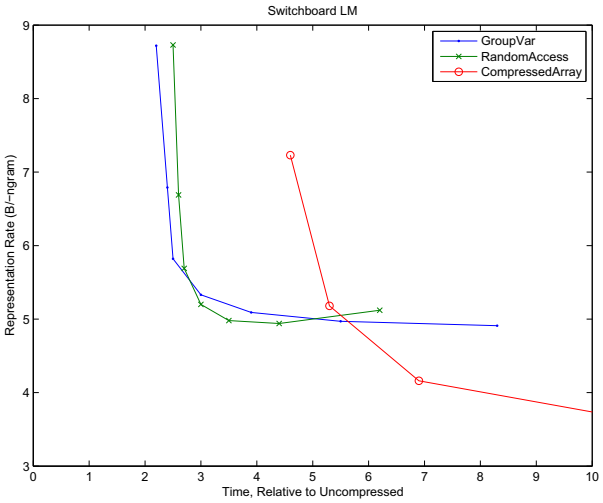

Figure 1: *Switchboard LM compression vs. speed, measured relative to uncompressed LM.*

RR≈4.5B/n-gram, operating ≈3X slower than uncompressed.

Another important observation is that although the representation rates are similar, the speed of the two language models is impacted very differently by compression. The n-gram hit ratios at the highest order for the *SWB* and *GSV* models are 0.18 and 0.35, respectively, leading us to believe that the speed difference is due to additional trie traversals on *SWB*.

In a separate batch of experiments that we do not fully report due to lack of space, we have found that one of our compression techniques (Compressed Array, Section 4.3) also reduced the size of the arrays of quantized LogP/BoW values by 0.5B/n-gram at block length 256, bringing the RR to 2.7 and 2.6 B/n-gram for the *SWB* and *GSV* models, at 60X and 19X slower than uncompressed, respectively. Clearly, there are more storage savings possible from careful quantization/compression of the LogP/BoW values.

6. Conclusions and Future Work

Assuming that a 3X slowdown in the LM can be tolerated (in practice the LM calls in an ASR decoder take a small percentage of the overall decoding time), one can conceive using 1B n-grams in the first pass of an ASR system at a LM memory footprint of ≈4GB.

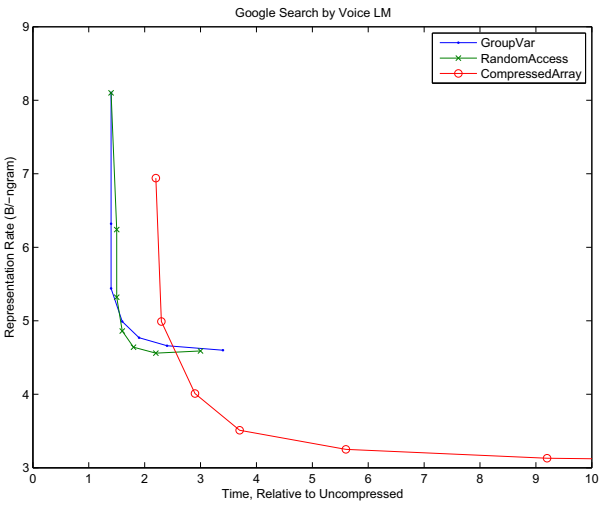

Figure 2: *Google Search by Voice LM compression vs. speed, measured relative to uncompressed LM.*

If time-bound, one can stay very close to uncompressed at high block length values by using full block GroupVar, and decoding schemes that are sub-linear in b. The offset and size for a given value can be computed efficiently from the header of a GroupVar block—see the masking approach in Section 4.2. More optimizations may be possible depending on the exact LM use in the decoder: if it needs to list probabilities of words in a given context, block decodes can be shared.

7. Acknowledgements

The authors thank Thorsten Brants for many useful discussions.

8. References

[1] A. Stolcke, "Entropy-based pruning of back-off language models," in *Proceedings of News Transcription and Understanding Workshop*. Lansdowne, VA: DARPA, 1998, pp. 270–274.

[2] K. Seymore and R. Rosenfeld, "Scalable back-off language models," in *Proceedings ICSLP*, vol. 1, Philadelphia, 1996, pp. 232–235.

[3] R. Rosenfeld, "The CMU statistical language modeling toolkit and its use in the 1994 ARPA CSR evaluation," in *Proceedings of the Spoken Language Systems Technology Workshop*, 1995, pp. 47–50.

[4] A. Stolcke, "SRILM - an extensible language modeling toolkit," in *Proceedings of the International Conference on Spoken Language Processing*, Denver, CO, September 2002, pp. 901–904.

[5] E. Whittaker and B. Raj, "Quantization-based language model compression," Mitsubishi Electric Research Laboratories, Tech. Rep. TR-2001-41, December 2001.

[6] B. Hsu and J. Glass, "Iterative Language Model Estimation: Efficient Data Structure & Algorithms," in *Proc. Interspeech*. Brisbane, Australia: ISCA, September 2008.

[7] P. Clarkson and R. Rosenfeld, "Statistical language modeling using the CMU-Cambridge toolkit," in *Fifth European Conference on Speech Communication and Technology*. ISCA, 1997.

[8] D. Talbot and M. Osborne, "Smoothed Bloom filter language models: Tera-scale LMs on the cheap," in *Proceedings of the 2007 Joint Conference on EMNLP and CoNLL*, 2007, pp. 468–476.

[9] D. Talbot and T. Brants, "Randomized language models via perfect hash functions," in *Proceedings of ACL-08: HLT*. Columbus, Ohio: Association for Computational Linguistics, June 2008, pp. 505–513.