GloVe on Spark

Alex Adamson
SUNet ID: aadamson

June 6, 2016

Introduction

Pennington et al. proposes a novel word representation algorithm called GloVe (Global Vectors for Word Representation) that synthesizes the two primary model families for learning vectors, matrix factorization methods over term-document matrices such as LSA (Deerwester et al., 1990) and context-window modeling methods such as Word2Vec (Mikolov et al., 2014). The goal of GloVe is to embed representations of words in a corpus into a continuous vector space in such a manner that the parallel semantic relationships between words are modelled by vector offsets between words. In other words, we desire that the produced vector space encodes equivalent semantic relationships via linear offsets:

$$W_{\text{queen}} - W_{\text{king}} \approxeq W_{\text{wife}} - W_{\text{husband}}$$
$$W_{\text{dog}} - W_{\text{puppy}} \approxeq W_{\text{cat}} - W_{\text{kitten}}$$

Similarly, we desire that the produced vector space encodes equivalent functional relationships via linear offsets:

$$W_{\text{king}} - W_{\text{kings}} \approxeq W_{\text{queen}} - W_{\text{queens}}$$
$$W_{\text{dog}} - W_{\text{dogs}} \approxeq W_{\text{cat}} - W_{\text{cats}}$$

GloVe begins by forming a word-word co-occurrence matrix X where X_{ij} is the distance-weighted co-occurrence count of word i and word j within a context window of size n where by distance weighted we mean that if in a context window word i is found m words from word j, we let that co-occurrence contribute $\frac{n-m+1}{n}$ to X_{ij}. To recover continuous vector space embeddings for the words, we seek to factor $\log X$ into $W^T\tilde{W} + b + \tilde{b}$ where $W \in \mathbb{R}^{n\times\|V\|}$ is taken as the final word embeddings, $\tilde{W} \in \mathbb{R}^{n\times\|V\|}$ are the context word embeddings, and b and $\tilde{b}$ are bias terms meant to account for co-occurrences caused by the overall frequency of a word in the corpus rather than by any relationship between the words.

Approach

We approach the problem by casting the factorization objective above into a least squares problem:

$$J = \sum_{i,j=1}^{\|V\|} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{1}$$

where f is a weighting function defined by

$$f = \begin{cases} (x/x_{\max})^\alpha & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases}$$

and intended to prevent very frequent words from dominating the descent direction.

Instead of solving via alternating least squares, we can take advantage of the fact that the co-occurrence matrix X is likely to be very sparse since even in large corpora, word frequencies roughly follow a power law and in turn $f(X)$ is likely to be very sparse. We instead solve by either stochastic descent where at each iteration we update the parameters based on the gradient observed for a particular word-word co-occurrence (as in Pennington et al.) or by full-batch methods (as we seek to develop here). This approach allows us to update all parameters in a single iteration rather than holding all but one parameter constant for the sake of casting each iteration as a legal quadratic program.

Pennington et al. trains the LBL using hogwild descent via Adagrad where each thread is assigned some subset of the word pairs that have a positive number of co-occurrences and in parallel performs (without locking) the updates to the descent direction and updates to the word matrices and biases which are stored in parameters shared by across all threads. In practice, this method has been shown to be numerically stable and robust to choices of model parameters (the learning rate η, $x_{\max}$, α, and the size of the word representations, n). Hence, there is no reason to prefer full-batch methods over the hogwild implementation other than possible speedup.

Distributing GloVe

We seek to distribute the training portion of the GloVe algorithm (i.e. developing the log-bilinear model given a co-occurrence matrix) using the Spark framework. Our main goal is to find a method to exploit the sparsity of the co-occurrence matrix to prevent performing full matrix-matrix multiplications while still performing full batch updates. Spark's data model makes it non-trivial to implement any sort of hogwild update because sharing parameters stored in distributed matrices or other data structures requires synchronization and hence removes the advantages of hogwild updates, so we instead seek to find a way to compute and perform the updates using operations on the full parameter matrices.

Before we begin the training iterations, we calculate and cache $f(X)$ in block matrix form and do the same for $\log X$. Both operations take $O((|V|/p)^2)$ time where p is the number of partitions and require no communication (since we have already partitioned X and the operation here is just a map-values).

We explore how to efficient calculate and apply the full-batch updates. First, note that we can find the gradient with respect to the factorization expression $W^T\tilde{W}+B+\tilde{B}=\log X$ as follows:

$$\frac{\partial J}{\partial(W^T\tilde{W}+B+\tilde{B}-\log X)} \propto f(X)\circ(W^T\tilde{W}+B+\tilde{B}-\log X) \tag{2}$$

Having calculated this, we then update W using gradient descent as

$$\begin{aligned} W &\leftarrow W-\eta\cdot\frac{\partial J}{\partial(W^T\tilde{W}+B+\tilde{B}-\log X)}\cdot\frac{\partial(W^T\tilde{W}+B+\tilde{B}-\log X)}{\partial W} \\ &= W-\eta\cdot(f(X)\circ(W^T\tilde{W}+B+\tilde{B}-\log X)\cdot\tilde{W}) \end{aligned}$$

and update $\tilde{W}, b, \tilde{b}$ analogously.

The matrix-matrix operations present here are the elementwise multiply between $f(X)$ and the factorization expression, the elementwise additions between B, $W^T\tilde{W}$, $\tilde{B}$, and $-\log X$, and the matrix-matrix multiplies $W^T\tilde{W}$ and the partial cost gradient and $\tilde{W}$. We would like to minimize both the communication and the computation required during the matrix-matrix multiplies.

Here we exploit the sparsity of f. We model $f(X)$ as a block matrix and partition it by storing a block on a machine with its neighboring blocks. Note that before building X, we have sorted the words in the corpus such that the word i is the ith most frequent word in the corpus and so we expect the bottom right corner of X (the co-occurrence counts between the rarest words) to be relatively sparse. With sufficiently small block size, we will avoid having to calculate the local matrix multiplies for several blocks.

We partition $(W^T\tilde{W}+B+\tilde{B}-\log X)$ identically to $f(X)$ and lazily evaluate it after we apply the elementwise product with $f(X)$. We implement the elementwise product by performing an inner join over the blocks of the two matrices (represented as ((row block index, column block index), and performing a local elementwise product on the sub-matrices. Note that since the matrices are partitioned identically, no network communication is required. Since we do not explicitly represent empty entries, we preserve the block-level sparsity pattern in the result.

Since we lazily evaluate the local matrices that compose the blocks of $(W^T\tilde{W}+B+\tilde{B}-\log X)$ and in particular lazily evaluate the blocks of $W^T\tilde{W}$, we avoid having to actually calculate the local matrix multiplies or communicate the contents of the blocks matrices over the network for all multiplies involving a completely sparse block.

Let b_c and b_r be the number of row and column blocks in the left matrix respectively (so b_r and b_c is the number of row and column blocks in the right matrix) and assume both matrices are partitioned into a grid where each block in the grid is on the same machine. Assume that the left matrix is in $\mathbb{R}^{|V|\times n}$ and the right matrix is in $\mathbb{R}^{n\times|V|}$. Before accounting for the sparsity of $f(X)$, the shuffle size (in matrix elements) for the matrix-multiply is $\frac{|V|\cdot n}{b_c}(\frac{b_c}{p}-1)+\frac{|V|\cdot n}{b_r}(\frac{b_r}{p}-1)$ where p is the number of machines across which the blocks are partitioned. Suppose s is the number of sparse blocks in the left matrix (which are distributed randomly across the matrix). Then, in expectation, the shuffle size for the matrix multiply decreases to $\frac{s}{b_c b_r}(\frac{|V|\cdot n}{b_c}(\frac{b_c}{p}-1)+\frac{|V|\cdot n}{b_r}(\frac{b_r}{p}-1))$

n	$\lvert V\rvert$	Time	Cost
4000	25	2m2.938s	0.0615
8000	25	2m12.139s	0.0291
16000	25	5m27.746s	0.0481
4000	50	2m16.826s	0.0546
8000	50	4m55.088s	0.0477
16000	50	6m51.855s	0.0481

Table 1: Results for Hogwild implementation

n	$\lvert V\rvert$	Time	Cost
4000	25	2m30.405s	0.00444
8000	25	3m49.019s	0.00213
16000	25	8m11.923s	0.00178
4000	50	2m40.416s	0.00173
8000	50	4m26.961s	0.00173
16000	50	10m16.377s	0.00173

Table 2: Results for Spark implementation using a block size of 128 rows/cols per block

Results

Our Spark implementation scales significantly worse with the size of the vocabulary than the implementation provided by the GloVe authors. This makes sense considering our implementation depended on leveraging the sparsity of $f(X)$ by avoiding communication and computation involving completely empty blocks, but for all but the blocks involving the rarest words, it is unlikely that the block is completely sparse and hence we fail to avoid these computations and shuffles if even one entry in the block is nonzero. Even with a block size of 128, if we're using the 8000 most common words in a corpus of over a million sentences and over 72000 words with a window size of 15 (as we are doing here), it is staggeringly unlikely that no pair of words in the 128 least common words of the 8000 ever cooccur. For instance, for the case where we are using 16000 words, of the 15625 possible blocks, 15623 of them have a nonzero value. Additionally, as we lower the block size, the size of the join during the matrix multiplication (as in number of blocks output by the join) increases cubically (although the number of local matrices sent over the network remains the same), so there is a trade off between the lowered communication of raw matrices afforded by a lower block size and the increased number of local matrix routines that need to called (i.e. the number of records that are output by the join/cogroup in the matrix multiply).

The Spark implementation scales substantially better with the size of the word representations than the Hogwild implementation. This is likely because the word representation size is smaller than the block size we use (128) so increasing the size of the word representation does not lead to the shuffling of more elements or more calls to BLAS routines (although it does increase the cost of the sparse matrix multiplies in BLAS). We use BLAS via netlib-java when evaluating the product of blocks whereas the Hogwild implementation computes the analogous result ($W_i^T\tilde{W}_j$ for each cooccurring pair) via a basic dot-product implementation rather than SIMD or some other optimization.

Finally, note that our algorithm appears to converge much quicker than the Hogwild implementation so in some sense while the runtime of an iteration of our algorithm does not scale as well as an iteration of the Hogwild implementation, we do more work towards the objective per iteration (and per unit time) than the Hogwild implementation. [1]

To illustrate how the algorithm scales with sparsity, we run the algorithm after zeroing all entries in the cooccurrence matrix that fall below a threshold.

$x_{\min}$	% sparse blocks	Time	Cost
0.01	0.961	2m41.287s	1.126E-16
0.025	0.677	1m12.681s	8.11E-17
0.05	0.305	40.601s	5.76E-17

Table 3: Results for Spark implementation using a block size of 128 rows/cols per block, 4000 words, 25 dimensional vectors, varying the minimum threshold for clipping the cooccurrence matrix entries. Note that these costs are not comparable to those from the tables above since the clipping in general should make the regression easier (there are fewer terms in the summation)

The results as we increase the block sparsity of X are as expected: even for modest increases in the block sparsity, we see significant speedups. We expect that for larger vocabularies than we are using here (i.e. the sort of vocabularies that would be of actual interest in NLP tasks),

Future Work

Given more time, we would have liked to compare the scalability of the two implementations on much larger vocabularies (relative to the size of the corpus, since that ratio is the primary factor in the sparsity of the cooccurrence matrix) since our expectation is that the Spark implementation's scaling relative to the vocabulary size would dovetail with that of the Hogwild implementation as the number of sparse entries in sparse blocks approaches the number of sparse entries.

References

Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391, 1990.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. word2vec, 2014.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

[1]We have not peformed an exhaustive gridsearch and it is possible that with the correct learning rate, the iterations to convergence of both algorithms is closer