0.1 11/29/2017 Research Notes

Notation
Let $[n] = \{1, \ldots, n\}$ and for $m < n$ let $[n]_m = \{m, m+1, \ldots, n-1, n\}$

0.1.1 Preliminary Adagrad Results

Over the break, I set up some batches to start training.Letting $\mathcal{P} \in \mathbb{R}^{n \times n \times T}$ denote the PMI tensor, $U \in \mathbb{R}^{n \times d}$ denote the shared embedding, and $\mathcal{C} \in \mathbb{R}^{\times \times \mathbb{T}}$ denote the core tensor The batches are minimizing

$$\|\mathcal{P} - \mathcal{C} \times_1 U \times_2 U\|_F^2 + \lambda_1 \|U\|_F^2 + \lambda_2 \|\mathcal{C}\|_F^2$$

with randomized mini-batch iterations of the Adagrad tensorflow (tf) optimizer with respect to U and $\mathcal{C}$. The experiments were hoping to show an affect of the regularizer parameter variable, by choosing values uniformly along a logarithmic scale (base 10). To minimize the keep the number of tests run linear $\lambda_1 = \lambda_2 = \lambda$ for all the tests and $\lambda = 10^i$. for $i \in [3]_{-4}$. All but on of the tests are using two time slices $T = 2$ with the years 2000 and 2001, with the exception of one test running on the years 2000 through 2004 to see how performance would change as T increased.

The embeddings when plugged into arithmetic word embedding software did not create not capture semantics as well as the svd embeddings using 50 singular vectors. Also when the eigenvalues of the core tensor slices were compared, the eigenvalues distribution shape were very similar. There were slight differences in the range of the values, but the differences were marginal.

$$INSERTPLOTSHERE$$

Though Adagrad did not offer very good results, this may lie within the fact that Adagrad's step size will continue to diminish monotonically as the step count increases. This is accounted for in the Adagrad-δ function, so perhaps a different optimizer will yield different results. Experiments have been set up to not only explore the effects of different optimizers, but to also help find optimal batch sizes. The tests are only using one time slice, reducing the minimization problem down to

$$\|P - UCU^T\|_F^2 + \lambda \|U\|_F^2 + \lambda \|C\|_F^2$$

which is a regularized SVD approximator. Here because there is no constraint relating C and U, C can be arbitary, and so letting $C = BB^T$ we let U absorb C by solving for UB. From this we hope to find results similar to the embeddings produced by the svd and compare different hyper parameters of the problem. We will consider the following hyperparameters

Optimizers: Gradient Descent, Adagrad, Adagrad-δ, and Adam

batch size: for values 10^i for $i \in [4]$

where we choose $\lambda = 1e-3$. One thing to note is that to make sure the run time of comparable for each run, we decrease the number of iterations run as we increase the batch size. we keep the product of the two constant at 10^6. This will not be perfect as tensorflow may have some hidden behavior under the hood, but it should offer reasonable normalization nonethe less.

0.2 11/15/2017 Research Notes

Still having challenges computing the svd for each of the individual slices due to memory capacity issues, decided that it would be better to tackle this from the approach of learning a symmetric embedding with a line search method based off of the gradient. Assuming that we have computed the fft along each of the 3-mode tubes then we are given a dense tensor $fft(P) \in \mathbb{C}^{n \times n \times \lceil \frac{T}{2} \rceil}$. We then hope to learn a d dimensional complex embedding $U \in \mathbb{C}^{n \times d}$ which is

0.3 11/12/2017 Research Notes

0.3.1 t-svd

for a given tensor $P \in \mathbb{R}^{n \times n \times T}$ we want to be able to compute the t-svd for the Tensor along the 3-mode. In our particular case, we are given a set of matrix slices which are all sparse, thus we know that the number of non-zeros of the tensor is proportional to $O(nT)$.
To compute the t-svd we must first compute the fft on each of the n^2 3rd mode tubes.

Once we compute the fft along the 3rd mode, then we can compute the svd for each of the $\frac{T}{2}$ slices, and then using this we can combine the slices back together to and compute the ifft along the 3rd mode of the tensor to get the final result of the t-svd.

Computational Issues

The primary issue is that this will yield a dense Tensor as the number of non-zeros will become proportional to $O(n^2T)$ as there is no guarantee that the sparsity pattern of the

Computational Speedups

- Since the domain of the fft is real, then there will be a symmetry along the T domain, which means that we can store the fourier coefficients in $O(\frac{T}{2})$ space.
- Each of the tensor slices are symmetric matrices, so we don't need to compute the fft for all the i, j tubes, we can compute them for the lower or upper triangular sections of the tubular matrices and then store the result in the opposite size, this would not be wasteful as each matrix is

instantiated with all the elements in the tensor. We also could specialize the operations or find a symmetric matrix svd algo/datatype to prevent the need for instantiating all the non-zeros of the tensor. Also because the matrix slices are symmetric, then the svd will be symmetric.

- We also don't need to compute the full svd of each of each of the slices, we can compute the top k singular vectors and compute each (i, j) as needed in order to not explicitly form the dense n^2T tensor. With this in mind we could use randomize projection methods in order to diminish the dimensionality of the $n \times n$ matrix slices, into something that a process could fit into it's virtual memory. Another alternative is to consider is the use of Krylov methods in order to simply think in terms of the matrix vector multiplications to the matrix slices, perhaps using a distributed matrix vector computation.

Questions

- How much memory does each process spawned by the multiprocessing python module have access to? Does it get the full capacity of a regular process, or does it have to share resources with the parent process?

- how many dimensions can we project down to using the randomized projection method and safely maintain the information of the top k singular vectors?

0.4 10/31/2017 Research Notes

Currently using the tensorflow routine for computing the embedding minimizing the objective function with the Adagrad optimizer with an initial learning rate of .01 .

$$\arg\min_{U\in\mathbb{R}^{n\times d}} \|P - UBU^T\|_F - \lambda_1\|U\|_F$$

Currently hoping to impose B be symmetric positive definite. We impose symmetry by starting with a symmetric initial matrix and then noting that the gradient will be symmetric now that we're using U for both the word and context embeddings. So B will naturally stay symmetric. To impose positive definiteness, I have implemented a method to project B onto its positive eigenspaces at each iteration. I've done this by computing the eigendecomposition and only keeping the eigenvectors associated with the positive eigenvalues.

Projection Sanity Check:
Let $B = V\Lambda V^T$ be the eigendecomposition for B where V is an orthonormal basis for the eigenspaces. With this, note that we can partition.order the eigenpairs such that $V = [V_p\ \tilde{V}]$ where V_p are the eigenvectors associated with the positive eigenvalues and $\tilde{V}$ are all the other eigenvalues. Similarly we can partition/order the eigenvalues in the block diagonal matrix $\Lambda = \begin{bmatrix}\Lambda_p & \\ & \tilde{\Lambda}\end{bmatrix}$, using similar notation to denote the positive eigenvalues.

Thus we can write the projection onto the positive eigenspaces as $P = V_pV_p^T$ which means that we can compute PA as

$$\begin{aligned} PA &= V_pV_p^T(V\Lambda V^T) = V_pV_p^T([V_p\ \tilde{V}]\Lambda V^T) \\ &= V_p[I_p\ \vec{0}]\begin{bmatrix}\Lambda_p & \\ & \tilde{\Lambda}\end{bmatrix}\begin{bmatrix}V_p^T \\ \tilde{V}^T\end{bmatrix} = [V_p\ \vec{0}]\begin{bmatrix}\Lambda_pV_p^T \\ \tilde{\Lambda}\tilde{V}^T\end{bmatrix} \\ &= V_p\Lambda_pV_p^T. \end{aligned}$$

However this method is not proving successful in maintaining the positive definiteness of the matrix, as the next update re-introduces the negative eigenspaces. There is also another key issue with this method as even after removing the negative eigenvalued eigenspaces, which is that this makes the matrix B rank deficient, and thus it is positive-semi definite. Since the number of negative eigenvalues of B has been around 50. So at best, this imposes a semi-norm into our loss functions, where each of the equivalence classes will be quite large because of the size of the kernel of B.

However having discussed this problem with a friend of Shuchin and I, he suggested that we simply multiply the matrix to its transpose to enforce positive definiteness. So I have adjusted the objective functions to minimize

$$\arg\min_{U\in\mathbb{R}^{n\times d}} \|P - UBB^TU^T\|_F - \lambda_1\|U\|_F$$

. However because there are no constraints on U or B, this will not lead to very successful embeddings. I have since realized my mistake and currently implementing an objective function to learn a shared embedding across multiple PMI matrices P_k. minimizing the objective function

$$\arg \min_{U \in \mathbb{R}^{n \times d}} \sum_k \|P_k - U B_k B_k^T U^T\|_F - \lambda_1 \|U\|_F.$$

I will also add in regularizers for the B_k terms in the objective functions.

0.5 Experiment Results

The following subsections are labeled by their file name templates on the dell37 server.All of the embeddings are normalized row-wise. None of the embeddings computed produced significant results. I also produced experiments varying λ_1 on a logarithmic scale, but the also did not show any improvements. I believe the mistake is not using multiple time slices to create a proper core, and without a positive definite B, I have to use a row normalized U, rather than $UV\sqrt{\Lambda}$ which means when testing the embeddings I'm leaving out a critical part of the factorization.

0.5.1 wordPairPMI_2000_iterations_10000_lambda1_-1.0_lambda2_-1.0_dimensions_50_2tf[U,B]

This experiment ran for 10 times as long as any of the other embeddings that I've been running. This function included the projection method for B. However the extra iterations did not improve the quality of the embedding at all. Due to the fact that B was not positive definite, I could not compute a cholesky decomposition or compute the square root of the eigenvalues without introducing imaginary terms.

Note that the λ_2 term in the file name is a relic of the regularizer for the V matrix, which is no longer present in the most recent versions of the software pushed up to the repo.

0.5.2 wordPairPMI_2000_iterations_1000_lambda1_-1.0_lambda2_-1.0_dimensions_50_2tf[U,B].npy

This experiment was run before I had implemented the projection method for B. However this embedding did not produce significant results either.