Representation learning notes

Representation learning or feature extraction is a way to create kind of data compression applied to general data. Given a set of data x_i, $i = 1,2,\ldots,N$ with $x_i \in \mathbb{R}^n$, we want to find functions $\varphi_j \colon \mathbb{R}^n \to \mathbb{R}$, $j = 1,2,\ldots,m$ where the inputs x_i can be accurately reconstructed from $\varphi_j(x_i)$, $j = 1,2,\ldots,m$. Let $\varphi(x) = [\varphi_1(x), \varphi_2(x), \ldots, \varphi_m(x)]^T \in \mathbb{R}^m$. Essentially we are creating autoencoders.

We assume that we will use linear reconstructions. That is, we want to find a matrix Z where

$$Z\varphi(x_i) \approx x_i \qquad \text{for all } i.$$

This corresponds to an autoencoder where the decoder layer is linear. If we want to compute $Z \in \mathbb{R}^{n\times m}$ that minimizes

$$\sum_{i=1}^{N} \|Z\varphi(x_i) - x_i\|_2^2,$$

this involves solving a least squares problem, which can be carried out efficiently. If we write $\varphi_j(x) = \varphi(x; w_j)$ for a suitable vector or parameters $w_j \in \mathbb{R}^p$ (such as might describe the weights for the input of a node in a neural network, for example). We could put $\varphi(x;w) = \sigma(w^T x)$ with σ a suitable activation function. If we let $W = [w_1, w_2, \ldots, w_m] \in \mathbb{R}^{p\times m}$, then we can write the problem for Z as a least squares problem

$$\min_Z \|Z\Phi(W) - X\|_F^2$$

where $\Phi(W) = [\varphi(x_1;W), \varphi(x_2;W), \ldots, \varphi(x_N;W)] \in \mathbb{R}^{m\times N}$ with

$$\varphi(x;W) = \begin{bmatrix} \varphi(x;w_1) \\ \varphi(x;w_2) \\ \vdots \\ \varphi(x;w_m) \end{bmatrix}, \qquad \text{and } X = [x_1, x_2, \ldots, x_N] \in \mathbb{R}^{n\times N}.$$

This least squares problem can be solved by either the normal equations

$$Z\Phi(W)\Phi(W)^T = X\Phi(W)^T,$$

or by means of a QR factorization $\Phi(W)^T = QR$ with Q orthogonal $N \times N$ and R upper triangular $N \times m$. In dealing with the common case where $N \gg m$, we should use a reduced QR factorization $\Phi(W)^T = Q_1 R_1$ where Q_1 is $N \times m$ has

orthonormal columns and R_1 is $m \times m$ upper triangular. The reduced QR factorization takes $\mathcal{O}(Nm^2)$ floating point operations, which is also the asymptotic cost of computing $\Phi(W)\Phi(W)^T \in \mathbb{R}^{m\times m}$ as $N, m \to \infty$ with $N \geq m$.

The solution is given by

$$\begin{aligned} Z R_1^T R_1 &= X Q_1 R_1, \qquad \text{or equivalently} \\ Z &= X Q_1 \left(R_1^T\right)^{-1}. \end{aligned}$$

If you want to do this in parallel, there are parallel QR factorization algorithms.

Permutation symmetry and non-convexity

Note that if we permute the w_j vectors, then we permute the features, but do not change or create or destroy any features. Thus the result is essentially equivalent. This can be represented in matrix terms by supposing that if P is a permutation matrix (every entry is either zero or one, and every row and every column has exactly one entry that is a one) then

$$\begin{aligned} \varphi(x; WP^T) &= P\varphi(x; W), \qquad \text{and} \\ \Phi(WP^T) &= P\Phi(W). \end{aligned}$$

The corresponding reconstruction matrix is ZP^T.

A consequence of this permutation symmetry is a lack of convexity in the overall optimization problem except for trivial (and useless) cases. Suppose that

$$\psi(W) := \min_Z \|Z\Phi(W) - X\|_F^2$$

were convex in W. Then as $\psi(WP^T) = \psi(W)$ for every permutation matrix P, if W^* is a global minimizer, the average

$$\overline{W} = \frac{1}{m!} \sum_{P\in\mathcal{S}_m} W^* P^T$$

must satisfy

$$\psi(\overline{W}) \leq \frac{1}{m!} \sum_{P\in\mathcal{S}_m} \psi(W^* P^T) = \frac{1}{m!} \sum_{P\in\mathcal{S}_m} \psi(W^*) = \psi(W^*).$$

In other words, $\overline{W}$ must also be a global minimizer. But $\overline{W}P^T = \overline{W}$ for all permutation matrices P, and so $w_j^* = w_k^*$ for all j and k. That is, all features are the same. This is clearly not a reasonable outcome, and in almost all circumstances would definitely *not* be a global minimizer of ψ.

We can say more than that ψ is not convex. If W^* (the global minmizer) is a strict local minimizer and no two columns of W^* are identical, then there must be $m! - 1$ other strict local minimizers. This makes for a highly non-convex optimization problem with many local minimizers.

Overall problem

The overall problem is to determine the optimal way to reconstruct the input data. With errors measured in the 2-norm, this becomes

$$\min_{W,Z} \|Z\Phi(W) - X\|_F^2 .$$

Since the problem of minimizing over Z is straightforward, we can simply solve that directly as a least squares problem. The bigger problem is finding W that minimizes

$$\psi(W) := \min_{Z} \|Z\Phi(W) - X\|_F^2 .$$

For that we can properly consider using heuristics for minimizing over W.

We can compute gradients for ψ: $\nabla\psi(W)$. The code for doing this is somewhat complex, but definitely possible. Furthermore, the cost of this is $\mathcal{O}(Nm^2)$, asymptotically the same cost as for solving the least squares problem in the first place.

If we use metaheuristics that avoid computing derivatives (such as PSO, GSA, GA's), then we just need to compute $\psi(W)$ for any given W via solving the least squares problem.

If the number of data points N is "too big" for doing these computations (e.g., because of memory limits), then we should take a subsample of the data. Without taking a subsample of the data, we cannot even hope to determine $\|Z\Phi(W) - X\|_F^2$ for any given W or Z. We can create a number of nested subsamples by choosing increasing subsets of $\{1,2,\ldots,N\}$. We can choose subsets $\mathscr{D}_1 \subset \mathscr{D}_2 \subset \cdots \subset \mathscr{D}_M = \{1,2,\ldots,N\}$; once the solution for subset $\mathscr{D}_1$ is obtained, the Z and W found will be starting points for the problem with subset $\mathscr{D}_2$, etc.

Gradient computations

Derivative computations show that for $\delta W \approx 0$,

$$\psi(W+\delta W)-\psi(W)=Z^T(Z\Phi(W)-X)\bullet(\Phi(W+\delta W)-\Phi(W))+\mathscr{O}(\|\delta W\|^2)$$

where $A\bullet B=\text{trace}(A^TB)=\sum_{i,j}a_{ij}b_{ij}$. That is,

$$\delta\left[\psi(W)\right]=Z^T(Z\Phi(W)-X)\bullet\delta\left[\Phi(W)\right].$$

Now $\phi_{ij}(W)=\varphi(x_j;w_i)$ so

$$\phi_{ij}(W+\delta W)-\phi_{ij}(W)=\nabla_w\varphi(x_j;w_i)^T\delta w_i+\mathscr{O}(\|\delta W\|^2).$$

Then

$$\begin{aligned}A\bullet(\Phi(W+\delta W)-\Phi(W))&=\sum_{i,j}a_{ij}\nabla_w\varphi(x_j;w_i)^T\delta w_i+\mathscr{O}(\|\delta W\|^2)\\&=\sum_i\left(\sum_j a_{ij}\nabla_w\varphi(x_j;w_i)\right)^T\delta w_i+\mathscr{O}(\|\delta W\|^2)\\&=G\bullet\delta W+\mathscr{O}(\|\delta W\|^2)\end{aligned}$$

where column i of G is $\sum_j a_{ij}\nabla_w\varphi(x_j;w_i)$. This is a sum over all data points. Let H be the matrix with column j being $\nabla_w\varphi(x_j;w_i)$. Then

$$g_{ki}=e_k^T\sum_j a_{ij}\nabla_w\varphi(x_j;w_i)=\sum_j a_{ij}e_kHe_j=\sum_j a_{ij}h_{kj}=\left(HA^T\right)_{ki}$$

and so $G=HA^T$. For $A=Z^T(Z\Phi(W)-X)$ this gives

$$G=H\left(Z\Phi(W)-X\right)^TZ.$$

Note that G is the "matrix gradient" of $\psi(W)$ with respect to W. To compute this we need clearly to compute the solution Z of the matrix least squares problem.

Note that H and X are $n\times N$, $\Phi(W)$ is $m\times N$ while Z is $n\times m$. Assuming $m\ll n\ll N$, the we can determine an optimal order for computing the matrix products. Computing $R:=\left(Z\Phi(W)-X\right)^T$ takes $\mathscr{O}(nmN)$ operations to compute an $N\times n$ matrix. Then we can either compute $(HR)Z$ or $H(RZ)$. The first takes $\sim 2nNn+2n^2m$ operations while the second takes $\sim 2Nnm+2nNm$ operations. Then the

optimal order of products is $H(RZ)$ provided $n > 2m$. Otherwise the optimal order is $(HR)Z$. The ordering of the products, however, only gives a factor of two difference in the cost.

The efficient computation of gradients means that there are many other algorithms that can be efficiently implemented. These include conjugate gradients and limited memory quasi-Newton methods. The limited memory quasi-Newton methods can include limited memory BFGS and limited memory symmetric rank-1 (SR1) methods. The SR1 based methods should use trust region methods. Using limited memory quasi-Newton methods does mean that the amount of memory needed is at least an order of magnitude (a factor of 10) larger than the memory needed for the solution, but should not be two orders of magnitude (a factor of 100). This means that applicability may be a bit more limited than simpler methods, like gradient descent, but is still mostly linear in the number of data points N.

Parallelizing

Parallelizing over data points

Since we expect N to be much larger than m and n, it makes sense to parallelize over the data points. We can distribute the data points across processors $1, 2, \ldots, p$ (p is the number of processors).

Partition $\{1, 2, \ldots, N\} = \mathscr{P}_1 \cup \mathscr{P}_2 \cup \cdots \cup \mathscr{P}_p$ where $\mathscr{P}_k$ is the set of indexes of data points on processor k. We can let $X_k = [x_i \mid i \in \mathscr{P}_k]$ be the data points on processor k. Note that we mean that x_i is a column of X_k provided $i \in \mathscr{P}_k$. Correspondingly, set $\Phi_k(W) = [\varphi(x_i; W) \mid i \in \mathscr{P}_k]$ to be the feature vectors for data points on processor k. Again, we mean that $\varphi(x_i; W)$ is a column of $\Phi_k(W)$ provided $i \in \mathscr{P}_k$. We can write $X = [X_1 \mid X_2 \mid \cdots \mid X_p]$ and $\Phi(W) = [\Phi_1(W) \mid \Phi_2(W) \mid \cdots \mid \Phi_p(W)]$.

The biggest challenge with this is that computing the QR factorization of

$$\Phi(W)^T = \begin{bmatrix} \Phi_1(W)^T \\ \Phi_2(W)^T \\ \vdots \\ \Phi_p(W)^T \end{bmatrix} \qquad (N \times m).$$

We can do this as a "tall and skinny" QR factorization which can be parallelized. See, for example [1], which considers MapReduce parallel architectures. An earlier paper deals with MapReduce architectures, but is an R-only version of the QR

factorization [2]. The algorithm in [1] claims to provide an algorithm that can stably compute the QR factorization "in only slightly more than 2 passes over the data".

Parallelizing over features

If m is large as well, we can also consider further parallelism with respect to features. Partition the features $\{1,2,\ldots,m\} = \mathscr{F}_1 \cup \cdots \cup \mathscr{F}_q$ This can be done by means of a block subdivision into a $p \times q$ array of processors where processor (k,ℓ) contains x_i and $\varphi_j(x_i; w_j)$ for $(i,j) \in \mathscr{P}_k \times \mathscr{F}_\ell$.

The parallelization would be considered to be the outer parallelization while parallelism over the data points would be the inner parallelization. So we could perform a parallel QR factorization of the first block using the above parallel algorithm. The Q factor would need to be stored as a product of Householder reflectors, or as a WY representation of Q. The Householder vectors or WY matrices would be stored in distributed memory according to the Then Q^T is multiplied by the other block columns of the $\Phi(W)$ matrix. This step is the most expensive step of the factorization, and where parallelism is advantageous.

Block-cyclic distribution of the features would be the most efficient. Even so, parallelism over features would only be of secondary benefit.

References

[1] Austin R. Benson, David F. Gleich, and James Demmel. Direct QR factorizations for tall-and-skinny matrices in mapreduce architectures. *CoRR*, abs/1301.1071, 2013.

[2] Paul G. Constantine and David F. Gleich. Tall and skinny QR factorizations in MapReduce architectures. In *Proceedings of the second international workshop on MapReduce and its applications - MapReduce '11*. ACM Press, 2011.