CS 542G: Cholesky, Interpolation with Errors

Robert Bridson

September 29, 2008

1 Cholesky Factorization

1.1 SPD Matrices

Last time we started looking at a special (but very common, and very important) class of matrices: symmetric positive definite (SPD) matrices. Today we'll see we can modify LU to run faster and better on this class.

First, to recap: a matrix A is symmetric if it is equal to its transpose: $A = A^T$, i.e. $a_{ij} = a_{ji}$. A matrix is positive definite if $x^T Ax > 0$ for all nonzero vectors x. An SPD matrix A therefore enjoys several additional properties:

- Just from symmetry, it has all real eigenvalues and a complete orthonormal set of eigenvectors. If x is one of these eigenvectors ($Ax = \lambda x$) then positive definiteness implies $x^T Ax > 0$, i.e. $\lambda \|x\|^2 > 0$, which means every eigenvalue is positive.
- Since all eigenvalues are positive, A is nonsingular.
- The 2-norm of A is its maximum eigenvalue. See below.
- Every diagonal entry of A is positive: if you form x as the vector of all zeros except for a 1 in entry i, then $x^T Ax$ is the i'th diagonal entry: $x^T Ax = a_{ii}$. Positive definiteness requires it be positive.
- Generalizing the positive diagonal entries, every principal submatrix of A is also SPD. That is, if you pick a subset $\{i_1, i_2, \ldots i_k\}$ of $\{1, \ldots, n\}$ and form the $k \times k$ submatrix B with rows and columns induced by this subset, then B is SPD too. (Obviously it's symmetric, and $x^T Bx = y^T Ay > 0$ where y is x expanded with zeros for the entries not included in B.)

Properties like this make life much, much nicer than in the general matrix case, and provides for a wealth of elegant and powerful algorithms.

1.1.1 The 2-Norm

Let's take a closer look at one of those properties, the fact that $\|A\|_2$ is equal to the largest eigenvalue of A. Recall the definition

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}$$

We'll directly figure out a vector x that maximizes this ratio. From symmetry, A has a complete set of orthonormal eigenvectors: let's call them $v_1, v_2, \ldots, v_n$. Any arbitrary vector x can be written as a linear combination of them:

$$x = \alpha_1 v_1 + \ldots + \alpha_n v_n$$

We'll rewrite that as

$$x = V\alpha$$

where V is an $n \times n$ matrix containing the eigenvectors as columns:

$$V = \left(\begin{array}{c|c|c|c} v_1 & v_2 & \cdots & v_n \end{array} \right)$$

(in class I used U instead of V, but I want to make sure there's no confusion with the completely different U from LU factorization), and $\alpha = (\alpha_1, \ldots, \alpha_n)$ is the vector of coefficients. Note that V is in fact an orthogonal matrix, since its columns are orthonormal: its transpose is its inverse, $V^T V = I$, since $V^T V$ contains the dot-products of each column with the other columns.

Let's evaluate $\|x\|_2$ in terms of α:

$$\begin{aligned} \|x\|_2^2 &= \|V\alpha\|_2^2 \\ &= (V\alpha)^T (V\alpha) \\ &= (\alpha^T V^T)(V\alpha) \\ &= \alpha^T (V^T V)\alpha \\ &= \alpha^T \alpha \\ &= \|\alpha\|_2^2 \end{aligned}$$

Near the end we used the fact $V^T V = I$. This is of course the classic result that multiplying by an orthogonal matrix doesn't change the 2-norm of a vector.

Note that

$$\begin{aligned} Ax &= A(\alpha_1 v_1 + \ldots + \alpha_n v_n) \\ &= \alpha_1(Av_1) + \ldots + \alpha_n(Av_n) \\ &= \alpha_1\lambda_1 v_1 + \ldots + \alpha_n\lambda_n v_n \end{aligned}$$

Therefore $\|Ax\|_2^2 = (\alpha_1\lambda_1)^2 + \ldots + (\alpha_n\lambda_n)^2$. Therefore, the 2-norm of the matrix is:

$$\|A\|_2^2 = \max_{\alpha \neq 0} \frac{\alpha_1^2\lambda_1^2 + \ldots + \alpha_n^2\lambda_n^2}{\alpha_1^2 + \ldots + \alpha_n^2}$$

You can view the numerator as a weighted version of the sum of squares of α's in the denominator, with weights equal to the squared eigenvalues. Assume without loss of generality that the eigenvalues are in increasing order, i.e. λ_n is the biggest. It should be fairly obvious that to maximize this ratio with respect to the α's, you should make α_n nonzero and the rest zero—their weights are smaller. This gives:

$$\|A\|_2^2 = \lambda_n^2$$

Since A is SPD, so all its eigenvalues are positive, we know $\|A\|_2 = \lambda_n$. (For general symmetric matrices, the same steps can be used to show $\|A\|_2$ is the largest absolute value of any eigenvalue.)

1.1.2 Building SPD Matrices

I've already asserted that SPD matrices are one of the most common and important classes of matrices, but not justified why. We'll see a real-world example next lecture, but for now I'll stick to the general principle from which virtually all examples arise.

Let $B \in \mathbb{R}^{m \times n}$ be a matrix with full column rank, i.e. whose columns are all linearly independent. (This implies that $m \geq n$, so if B isn't square it must be taller than it is wide.) Note that for any nonzero vector x, this implies $Bx \neq 0$: otherwise x would tell us coefficients of a nonzero linear combination of columns of B that is zero, which would mean they weren't linearly independent. Therefore $\|Bx\|_2 > 0$, and expanding this out gives:

$$\begin{aligned} \|Bx\|_2^2 &= (Bx)^T(Bx) \\ &= x^T(B^TB)x \end{aligned}$$

This clearly shows that the matrix $A = B^TB$ is positive definite; it's also clearly symmetric. This is the usual recipe for SPD matrices, though it may take some digging to figure out what B is for a particular example.

1.2 Cholesky

For triangular factorization, it turns out SPD means the **Cholesky** factorization exists:

$$A = LL^T$$

where L is a lower triangular matrix. This is not the same L as in LU, since that L had all ones on the diagonal and the Cholesky factor might not. Also note that L^T is upper triangular, but again it's not quite the same as the U from LU. However, it can be shown there is a very close relationship that amounts to rescaling rows or columns.

In the 1×1 case, Cholesky factorization is just the square root: if $a > 0$ then there is a number l with $a = l^2$. By convention we choose the positive square root, $l = \sqrt{a}$, and similarly for the full Cholesky factorization we choose L to have positive diagonal entries.

For SPD matrices larger than 1×1 it's not immediately obvious that the Cholesky factorization does exist. We saw last time how for general matrices, LU might not exist and thus we turned to pivoting; we'll now show that in the SPD case LL^T always exists without need for pivoting.

We can do this inductively or recursively using our block approach again, with the 1×1 case as the base of induction/recursion. In particular, let's write down (formally) the blocks as:

$$\begin{aligned} \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & L_{22}^T \end{pmatrix} \\ &= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix} \end{aligned}$$

This gives us three matrix equations (well, four, but one is just the transpose of another):

- $A_{11} = L_{11}L_{11}^T$, i.e. the Cholesky factorization of the principal submatrix A_{11}. We know this is SPD and of smaller dimension, therefore this should succeed.
- $A_{21} = L_{21}L_{11}^T$, which resolves as $L_{21} = A_{21}L_{11}^{-T}$ (using the superscript $-T$ as a convenient abbreviation to indicate the inverse of the transpose). We already know L_{11} exists, and must be invertible since A_{11} is. So we have no problems with this step.
- $A_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T$, or equivalently $(A_{22} - L_{21}L_{21}^T) = L_{22}L_{22}^T$.

The last step is the Cholesky factorization of the matrix $A_{22} - L_{21}L_{21}^T$; however, while it's clear that A_{22} is SPD and $L_{21}L_{21}^T$ is at least symmetric positive semi-definite, it's not clear that their difference is SPD. (For

example, it's easy to pick two positive numbers whose difference is negative, in general.) It's obviously symmetric, but to show that it is in fact positive definite, i.e. $x^T(A_{22} - L_{21}L_{21}^T)x > 0$ for $x \neq 0$, we need to do some work.

One way of doing this is with the following partial inverse factor, F:

$$F = \begin{pmatrix} L_{11}^{-1} & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix}$$

This is designed so that:

$$\begin{aligned} FAF^T &= \begin{pmatrix} I & 0 \\ 0 & A_{22} - A_{21}A^{-1}A_{21}^T \end{pmatrix} \\ &= \begin{pmatrix} I & 0 \\ 0 & A_{22} - L_{21}L_{21}^T \end{pmatrix} \end{aligned}$$

Note that F is invertible, so if $y \neq 0$, then $F^T y \neq 0$: this implies $y^T(FAF^T)y = (F^T y)^T A(Fy) > 0$, so FAF^T is SPD as well. If we take $y = (0, x)$ for a given nonzero x, then

$$\begin{aligned} x^T(A_{22} - L_{21}L_{21}^T)x &= y^T(FAF^T)y \\ &> 0 \end{aligned}$$

Therefore the Cholesky factorization can proceed!

1.3 Advantages of Cholesky

Cholesky gets a few advantages over LU:

- Only one triangular factor needs to be calculated and stored, saving roughly a factor of two in memory.
- The number of arithmetic operations is roughly half what LU requires, speeding it up by roughly a factor of two.
- No pivoting is required, simplifying the code, allowing a wider range of algorithm variants, and reducing unpredicatble memory access since no permutations take place. This further improves performance.

The second one is the hardest to see: it's a good exercise to derive, for example, the right-looking version of Cholesky and estimate the number of arithmetic operations in each step compared to LU. Note in particular that the rank-1 update need only be applied to the lower (or upper) triangle of A, since by symmetry we can ignore one triangle of A (accessing a_{ij} when we need a_{ji} if necessary).

1.4 Stability of Cholesky

We saw above that the Cholesky factorization exists for every SPD matrix without need for pivoting, at least in exact arithmetic. It turns out it also leads to wonderfully stable algorithms for solving SPD linear systems. As a brief peek at the proof of this, we can estimate the condition number of the factor L in terms of the condition of A.

Remember the Frobenius norm, which up to a small constant factor is equivalent to any other standard matrix norm, and how it can be phrased using the trace:

$$\|L\|_F^2 = \mathrm{tr}(LL^T)$$

In this case, LL^T is A, and recalling the trace is always equal to the sum of the eigenvalues, we get:

$$\|L\|_F^2 = \lambda_1 + \ldots + \lambda_n$$

This can be bounded in terms of the largest eigenvalue λ_n of A: the sum is less than or equal to $n\lambda_n$. Also remember that $\|A\|_2 = \lambda_n$. Therefore,

$$\|L\|_F^2 \leq n\|A\|_2$$

or, taking the square root:

$$\|L\|_F \leq \sqrt{n\|A\|_2}$$

We can do almost exactly the same bound for L^{-1} using $A^{-1} = L^{-T}L^{-1}$:

$$\|L^{-1}\|_F \leq \sqrt{n\|A^{-1}\|_2}$$

Therefore, the condition number of the factor is bounded as follows:

$$\kappa_F(L) \leq n\sqrt{\kappa_2(A)}$$

In particular, solving $Ax = b$ by doing forward and backwards substitution with L and L^T should give an answer accurate to within $\kappa(L)^2$ which, with a small factor dependent on n, is proportional to $\kappa(A)$. Therefore we will get a reliable answer.

1.5 Extension to Symmetric Indefinite Problems

While Cholesky works beautifully for SPD matrices, it fails on symmetric indefinite matrices: at some point it may need to take the square root of a negative number or divide by zero. (Obviously, as we've seen, LL^T is SPD, so it can't work for a matrix that isn't SPD.) In general, some form of numerical pivoting

is required to successfully factor a symmetric indefinite matrix. However, the forms of partial pivoting and complete pivoting we saw earlier won't help since they destroy matrix symmetry in general.

The first thought for dealing with these matrices is to introduce symmetric pivoting: if the entry a_{11} is zero or too small, swap both row 1 and row i (for some i) and column 1 and column i. This brings diagonal entry a_{ii} to the $(1,1)$ position. If we pick i to get the largest entry on the diagonal, we might be able to proceed with factorization.

Unfortunately, as we saw with the RBF matrix, in some cases every entry on the diagonal may be zero, but the matrix as whole is still invertible and even well-conditioned. Further tricks are required, the most common one being to allow the use of 2×2 block pivots.

More specifically, we look for an LDL^T factorization, where L is unit lower triangular and D is block diagonal—with diagonal blocks all 1×1 or 2×2. At each stage of factorization, the algorithm has to decide if a 1×1 pivot chosen from the diagonal will work, or if a 2×2 block pivot is necessary. The 2×2 blocks are treated with partial pivoting when they must be inverted. LAPACK uses a particular variant called Bunch-Kaufman, which is analogous to partial pivoting in the LU case.

2 Interpolating with Error

We've now covered the basics of what we need for RBF interpolation; let's go back and make the problem more interesting and see where that takes us. In particular, so far we have assumed we need to interpolate data points which were known exactly: there was some underlying smooth function (presumably) we are trying to reconstruct, and we know its exact value at the sample points. It made sense to look for the smoothest function possible which exactly goes through those data points.

However, it's very typical for there to be errors made in measurement. It's not always the case, but quite often those errors are roughly uncorrelated: the error made in measuring at one sample point bears no relation to the error made at another sample point—even one that's very close to the first. Think of each measured data value f_i as the sum of the exact value $\hat{f}_i$ and an error e_i. While we will continue to assume the exact function $\hat{f}(x)$ is smooth, the error is definitely not smooth; it's not even continuous.

Trying to pass a smooth-as-possible function exactly through these non-smooth data points will naturally result in a considerably wiggly, badly behaved interpolant, reflecting the nature of the error and obscuring the smooth true function. Ideally we would through out the error term and interpolate the exact function instead; we can't do that, but it does make it clear that we needn't reconstruct a function which exactly goes through the measured data points.

Instead, we need to find a good trade-off between a smooth function which is not wiggly but also closely approximates the data. We'll tackle this next by restricting the space of possible approximating functions to rule out wiggliness completely, and then search out the function from this space that comes closest to the measured data.