A zoo of matrix decompositions.

David G. Khachatrian

July 23, 2019

1 A zoo of matrix decompositions.

There are a great number of different decompositions of matrices that capture different aspects of the original matrix. Without the proper background, they might seem esoteric. The hope here is to provide descriptions that make some common decompositions more understandable and less daunting (or even worse – unmotivated!).

2 First things first: Matrices are linear transformations on column vectors.

An often-neglected point is that there's actually a *reason* why matrix multiplication works in the way that it does.

Let's say we have some matrix T with r rows and d columns (so it is $(r \times d)$). This matrix can only be multiplied by a vector/matrix with d rows (so it has dimension, say, $d \times c$). For now, let's say it's a vector x with dimension $d \times 1$. This would look something like

$$T = \begin{bmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,d} \\ t_{2,1} & t_{2,2} & \cdots & \\ \vdots & & \ddots & \\ t_{r,1} & & & t_{r,d} \end{bmatrix} = \begin{bmatrix} t_1 & t_2 & \cdots & t_d \end{bmatrix}, \quad x = \begin{bmatrix} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{d,1} \end{bmatrix}$$

So, what does Tx look like? The resulting matrix M (of shape $(r \times 1)$) will have entries

$$M_{i,1} = \sum_{k=1}^{d} T_{i,k} x_{k,1}$$

Let's look at this same sum a different way – grouping together by the columns of T (i.e., grouping by t_i), and considering the entire column of output of M:

$$M_1 = \sum_{k=1}^{d} T_k x_{k,1}$$

So what is the result? A linear combination of the columns of T, weighted by the corresponding entries of x. What does it mean? Consider how x was originally described – in terms of some basis vectors e_i:

$$\vec{x} = x_1\vec{e}_1 + x_2\vec{e}_2 + \cdots + x_d\vec{e}_d$$

where $\vec{x}$ lives in $\mathbb{R}^d$ (it has d coordinates), and so each $\vec{e}_i$ is a column vector with zeros in every one of d coordinates except at index i. Each $x_i\vec{e}_i$ represents the value of x's i'th coordinate.

What about afterward? The product $M = Tx$ can be described by

$$Tx = x_1\vec{t}_1 + x_2\vec{t}_2 + \cdots + x_d\vec{t}_d$$

So we literally just "hot-swapped" the e_i with t_i, which is where e_i lands if multiplied by T[1]. And in general, matrix multiplication of x by a matrix T can be thought of as answering, "If I were to move e_i to T_i via a linear transformation, where would the i'th basis vector of x (say called u_i) end up landing?" The reason why the computation doesn't really *look* that simple is because we end up converting the coordinates now described with basis vectors $T\vec{u}$ into the "standard basis vectors" $\hat{e}$ (vectors that often correspond to, e.g., the x, y, z axes – orthogonal, described by 1 alon gone coordinate and zeros everywhere else). The $T\vec{u}$ are often complicated in e coordinates – the columns of a matrix T usually aren't just one 1 and a bunch of zeros!

We consider this a *linear transformation of* x. And note that we may have changed where x "lives" – now it's described by a linear combination of t_i, which live in $\mathbb{R}^r$, so now Tx also lives in $\mathbb{R}^r$ (and it may be the case that $r \neq d$). For matrix-matrix multiplication TK with more than one column, you basically perform everything described above, just for each column of K.

An excellent, visual discussion of matrices as linear transformations (focusing on the case where $r = d$) can be found here.

3 One more thing: Matrices described as row operations.

There's another way of looking at what the entries of a matrix mean. It's usually not as helpful, but it'll be relevant in understanding Gaussian elimination and LU decomposition.

For matrix multiplication $Y = TB$ (T is $r \times d$, B is $d \times c$), let's break down our matrices differently, this time via the rows of B and Y:

$$T = \begin{bmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,d} \\ t_{2,1} & t_{2,2} & \cdots & \\ \vdots & & \ddots & \\ t_{r,1} & & & t_{r,d} \end{bmatrix}, \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,c} \\ b_{2,1} & b_{2,2} & \cdots & \\ \vdots & & \ddots & \\ b_{d,1} & & & b_{d,c} \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_d \end{bmatrix}$$

$$Y = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{r,c} \\ y_{2,1} & y_{2,2} & \cdots & \\ \vdots & & \ddots & \\ y_{r,1} & & & y_{r,c} \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_r \end{bmatrix}$$

Now we can write the individual rows of Y as

$$Y_i = \sum_{k=1}^{d} t_{i,k} B_k$$

Notice the interesting reversal of roles here.

1. In our earlier interpretation, we would have said that the *columns* (d vectors of dimension $r \times 1$) of T (the *left* matrix) transform the basis vectors of *each column* (of original dimension d) of B (the *right* matrix). This suggests a "right-to-left" approach to interpreting transformation on columns: T, a $r \times d$ matrix, sends vectors in d dimensions to vectors in r dimensions.

2. In this second and entirely equivalent description, we have that the *rows* (d vectors of dimension $c \times 1$) of B (the *right* matrix) transform the basis vectors of *each row* (of original dimension d) of T (the *left* matrix). This suggests a "left-to-right" approach to interpreting transformations on rows: B, a $d \times c$ matrix, sends vectors in d dimensions to vectors in c dimensions.

While these two interpretations are both technically equally valid, the first method of interpretation (focusing on columns and "right-to-left" interpretation) is far more common when discussing vectors in a more "abstract" sense.

[1] This is the idea behind the matrix identity I_d being a diagonal of ones – those are the standard basis vectors of $\mathbb{R}^d$, and $TI_d = T$.

3.1 The transpose operator.

On a related note: the above discussion suggests that M, an $n \times m$ matrix, "sends" column vectors of dimension m to vectors of dimension n using M's columns. What if we *want* to use M's *rows* as a basis (and we want to apply them to column vectors and still go "right-to-left", rather than completely switching perspective from (1) to (2))? **This is what the seemingly innocuous *transpose operation* (^{T}) does.** By flipping the indices for all entries of a matrix, $M^T(m \times n)$ makes all the original row vectors of $M(n \times m)$ into column vectors – which can now serve as the basis for an incoming vector in n dimensions. (This interpretation can also help make sense of other properties of the transpose, e.g., $(AB)^T = B^T A^T$.)

Though the second interpretation is rarely considered, the *mechanics* of looking at rows is useful in at least one common scenario: solving a linear system of equations where each column represents the coefficients of one particular variable. In this case, row operations represent manipulating the different parts of the system of equations.

In particular, let's make the following note about T_{ij} in $Y = TB$. T_{ij} answers the question: "How many multiples of the j'th row of B should be in the i'th row of Y?"

4 Gaussian elimination as a form of LU decomposition.

Let's consider a system of n equations with n unknowns and write the system in matrix form: $Ax = b$, where each row of A has the n coefficients for the unknowns of one of the n equations.

In an introductory linear algebra course, you may have been instructed to perform *Gaussian elimination* to solve such a system. Essentially, you try to turn A to I_n by performing certain *row* operations, and whatever actions you did to A you also did to y. The row operations are usually described as three choices:

1. Multiply a row by a constant multiple.
2. Add a multiple of some row j to a row i.
3. Swap rows i and j.

Note that (1) is just (2) with $j = i$, so we really have two types of operations. The "swapping" operation in this context is normally called "pivoting".

Let's also consider that if we have an upper diagonal matrix U for A, we can quite easily solve the system of equations. This would correspond to a system like:

$$\begin{aligned} 3x + 2y \quad & -z = 3 \\ 0x - y \quad & +3z = 2 \\ 0x + 0y \quad & +5z = 10 \end{aligned}$$

Not exactly hard to solve for z in the last equation and then walk your way "back" up. Solving a system of equations associated with an upper-diagonal matrix in this way is called *back subsitution* (you're going "back" toward the top). An analogous method exists for lower-diagonal matrices, starting from the first equation and walking "forward" through the equations – this method is called *forward subsitution*.

Now this is where our observation about rows comes back into place! In fact, every one of these row operations can be described by a matrix! For example, swapping the first and second rows of a 3x3 set of equations can be written as $P_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, and subtracting two multiples of the first row to the third row can be written as $S_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$.

If we only ever modify "lower" rows with multiples of "upper" rows and never need to swap rows to get A to be upper-diagonal U, we can fully describe a Gaussian elimination as a bunch of lower-diagonal matrices $L_1, L_2, \cdots$ – which, when multiplied together, happen to be lower-diagonal! This would *decompose* A such that $A = LU$. Once we have this expression, we can easily solve $Ax = b$ for any b:

1. Decompose $A = LU$.
2. Solve $y = Ux$ for x (via forward substitution).
3. Solve $L(Ux) = Ly = b$ for y (via back substitution).

Now of course, we normally aren't so lucky to not have to swap rows. But we can get the appropriate pivot matrix P ("row-swapper") and have $PA = LU$. Then all that changes is a few swaps: $Pb = PAx = LUx$.

So there's nothing super esoteric about LU decomposition – it's just Gaussian elimination!

For more specifics about algorithm implementation, one can check out the Wikipedia entry.

5 Symmetric positive-definite matrices: Definitely our favorite kinds of matrices.

Now, there's a subset of matrices of particular interest[2]: symmetric matrices A such that $x^TAx = \langle x, Ax\rangle > 0$ for all $x \neq \mathbf{0}$. Such matrices are called *(symmetric) positive-definite*.

This definition is a bit suggestive – what other function is positive for all nonzero y? One answer is $\langle y, y\rangle = \|y\|_2^2$, the squared norm of the vector y.

Hmm... The quadratic form is so *very* close to describing a norm. Maybe we could decompose $A = K^TK$ for some K – then we could write $x^TAx = x^T(K^TK)x = (Kx)^T(Kx) = \|Kx\|^2$, which is clearly positive for nonzero Kx (and so for nonzero x if K is full-rank). Can we find such a K? (Spoilers: we can!)

Before that though, a tidbit that may help make sense of positive-definite matrices. The *covariance matrix* Σ of a vector-valued random variable $\vec{\theta}$ is always positive semidefinite, and is positive definite assuming there are no perfect collinearities between coordinates (e.g. $\theta_1 + \theta_2 = c$ – this is to ensure Σ is full-rank).

Perhaps even more suggestively, let us say X is an $n \times p$ data matrix consisting of n observations of a random variable with p features/parameters. Let X_c be the centered version of this data (i.e., each column (representing a coordinate) is centered around zero). Then we can described the covariance matrix of X as $\Sigma = \tilde{X}_c^T\tilde{X}_c$ (where $\tilde{X} = \frac{1}{\sqrt{n-1}}X$).

This fits exactly the type of decomposition we want! Does it hold in general for any symmetric positive-definite matrix?

6 The Cholesky Decomposition: Fancy (and really convenient) Gaussian elimination.

Turns out, the answer is yes!

In particular, one can perform a variant of Gaussian elimination and end up with $A = LU = LL^T$. In particular, the idea is that "any Gaussian-elimination row operation L_i you perform to the left of A, you perform analogously"from the right" with L_i^T". This hearkens back to the idea of the transpose operator"interfacing" between row-vector and column-vector interpretations of matrix multiplication, while using the symmetry of the matrix A.

6.1 Covariances, Mercer's Theorem, the kernel trick.

A related fun fact: every positive semidefinite matrix describes a covariance matrix and vice-versa. This is an implication of Mercer's theorem and is usually invoked to explain why the kernel trick works. Essentially, the kernel trick involves reframing an objective function in terms of a measure of similarity between two vectors via an inner product $\left\langle x^{(i)}, x^{(j)}\right\rangle$. Now, if we define a *kernel function* $K(x_i, x_j)$ that describes

[2]Technically, this is more accurately described as "matrix representations of specific types of *quadratic forms*". More on quadratic forms here.

the similarity between two points and replace our inner product with our kernel function – voila, we've changed mapped x into a new feature space!

How did that happen?

By definition, the output of the kernel function K must form a *positive (semi)-definite matrix* Σ_n for any set of points $\{x^{(1)}, x^{(2)}, \cdots x^{(n)}\}$. So the kernel function is essentially an *uncountably infinite-dimensional analog to a covariance matrix*[3]. In most practical settings, you could explicitly create the implied covariance of your data Σ_n – which we just said can be decomposed to LL^T. So in a way, instead of calculating the inner product in the original space via $\left\langle x^{(i)}, x^{(j)} \right\rangle$, we're transforming $x \to L^T x$ and taking *its* inner product $\left\langle L^T x^{(i)}, L^T x^{(j)} \right\rangle$ instead. By specifying K, we implicitly specify a Σ_n and therefore a L^T for our data.

This cascade of implications (hopefully) intuitively make sense in words – by specifying how "close" two datapoints are (K), we're implicitly describing how to quantify their "difference"/"variance" (Σ_n), which would have to be describable by features of the datapoints (L^T).

[gaussian_process]: This is kind of like *"the Gaussian process of positive semi-definite matrices"*. In a Gaussian process, any finite subset of points must be describable as a multivariate Gaussian distribution. A good reference is here. (Note that the Gaussian process itself requires a kernel function, so the connection I described is a bit recursive – but hopefully it makes sense if one already knows about Gaussian processes.)

6.2 Cholesky decomposition and simulation convenience.

A commonly lauded practical use of the Cholesky decomposition is that it can be used to make a set of random variables have a specific covariance matrix. How?

For convenience, say our "starting" *column* vector $u \in \mathbb{R}^d$ which has covariance matrix $\Sigma_1 = I_d$. Our goal is to have a transformed u $T(u)$ so that it has covariance Σ.

Well, using Cholesky decomposition, we have $\Sigma = LL^T$. And by the Delta Method[4], we have that the covariance matrix of Lu is $\Sigma' = \nabla_u(Lu)^T \Sigma_1 \nabla_u(Lu) = L\Sigma_1 L^T = LL^T = \Sigma$. So left-multiply u by L and you're done. Nice and easy!

7 The Gram-Schmidt Process as QR decomposition.

Let's remember the Gram-Schmidt process. It's meant to turn an input set of vectors v a set of orthonormal vectors q that span the same subspace. It basically works recursively:

1. For $i = 1$, define $q_1 = v_1 / \|v_1\|$.
2. Define the i'th basis vector q_i as v_i with all the previously defined basis vectors subtracted out: $q_i \propto v_i - \sum_{j<i} \langle q_j, v_i, q \rangle_j$. Normalize q_i to be of unit length.

Perform this process on a matrix A, storing q_i into the i'th column of Q and the values of $\langle q_j, v_i \rangle$ in $R_{j,i}$, and bam – there's your QR decomposition! (The goal of this decomposition is to have an orthogonal matrix Q and a right-diagonal matrix R.)

(Worth noting: The Gram-Schmidt process, while geometrically pleasing and intuitive, is not the most numerically stable way to decompose a matrix. Take a look at the Wikipedia entry for more details.)

8 Singular-value decomposition and eigendecomposition.

Here I'll quote from when I described SVD as it compares to UV factorization (used in the problem of collaborative filtering, e.g. for a recommendation algorithm).

[3] This is kind of like *"the Gaussian process of positive semi-definite matrices"*. In a Gaussian process, any finite subset of points must be describable as a multivariate Gaussian distribution. A good reference is here. (Note that the Gaussian process itself requires a kernel function, so the connection I described is a bit recursive – but hopefully it makes sense if one already knows about Gaussian processes.)

[4] Why does the Delta Method apply? It applies for any function of a consistent estimator approaching some other function in distribution. (Some good notes available here.) And surely, $u \xrightarrow[n\to\infty]{(d)} u$. We then take $g(u) = Lu$.

8.1 Quoting myself...

The thought process behind this matrix factorization of X into U and V^T is probably one of the clearest explanations I've seen for what the components of singular value decomposition (SVD) are. There are just a few differences between what we discussed in lecture and how SVD works:

1. In lecture, we assumed the $n \times m$ matrix X to be of some specific rank k. In SVD, we don't explicitly make that assumption (though we still have a way of determining the relative importance via the *singular values* – more on that in a second).
2. In lecture, the professor mentioned how U and V^T can vary by scaling when writing $X = UV^T$. In SVD, we "capture" all of the scaling into a diagonal matrix Σ by writing $X = U\Sigma V^T$. The entries of Σ are the *singular values*, that describe the "strength of association" between the relevant entries in U and V^T. Small singular values mean that the association between the two are nearly non-existent – singular values equal to zero mean that X is not full-rank.

Besides that, it's basically as was described in lecture, and the intuitive explanation given by the example was great! Recapitulating for the most part: If X_{ai} is what person a thought about movie i and we want to describe things with k "concepts", then

- the a'th row of $U(n \times k)$ describes Person a's relative affinity for k unknown/"latent" concepts, and
- the i'th column of $V^T(k \times m)$ describes how relevant these k latent concepts are to movie i.
- When using SVD, the $\Sigma_{j,j}$ entry of the matrix Σ describes: if Person a likes concept j and Movie i is highly relevant to concept j, how much effect does that actually have on the final rating X_{ai}?

Technically, what's described above is a *truncated SVD*. The *full SVD* is slightly hairier (and this extra hairiness is probably what makes SVD feel so opaque). For "full" SVD, you need to have Σ be an $n \times m$ rectangular-diagonal matrix, and U and V^T change shape accordingly (and be made *orthogonal/unitary* matrices $U(n \times n)$ and $V^T(m \times m)$ – i.e., again, making sure Σ captures all of the scaling). Why? The number of nonzero entries in Σ is the rank of X, which is between 0 and $\min(n, m)$, so this ensures you're "safe" regardless of the rank of X.

But if you *know* the rank of X to be k – or *want* the matrix of rank-k that is "close" of X – then you can

1. do the truncated SVD with Σ as a $k \times k$ matrix and (appropriate shapes for U and V^T), as described above. When the rank of X is in fact $\leq k$, this gives the "same" result as full SVD. When $\text{rank}(X) > k$, I believe this gives the "closest" rank-k matrix to X, but it does so by "mixing in" the less important "concepts" into the k concepts we have kept. Or;
2. do the full SVD, reorder the matrices so that the largest k singular values are at the top, and drop the rest of the rows/columns. This may be slightly less "close" compared to (1), but it doesn't perform any "extra" mixing like (1) does.

Which one you prefer would probably depend on the sizes of n, m, and k.

7/14/19 UPDATE

The good point made by @khanhedx in their response made me want to emphasize something:

SVD only works on matrices with *filled-in data*. Trying to perform SVD on Y directly (which is mostly filled with unknown entries) will cause errors or give erroneous results (if your marker for "unknown data" is a number, e.g., -1).

Also, ***different assumptions are made*** when imputing X via truncated SVD ($X = U\Sigma V^T$, rows of U and columns of V^T have unit norm) compared to $X = UV^T$. In SVD, as mentioned before, *all* scaling is captured by Σ and shared across all rows of U (people) and columns of V^T (movies), which all have *unit norm*. What this means is, for example, there is no way for an X factorized according to the truncated SVD to capture the idea that, say, "Person a rates all types of movies more highly in general compared to Customer b" (mathematically, $U_{aj} > U_{bj}, j \in \{1, 2, \cdots, k\}$).

One could regain this capability by relaxing the restriction of unit norms – at which point you can drop Σ altogether and recover the matrix factorization $X = UV^T$ discussed in lecture. Or you could perhaps keep Σ and play with regularization terms to allow but punish deviations of U and V^T from unit norm ("you're allowed to say that some users just like all movies more than others, but I don't want you to use that reason willy-nilly").

The proposed changes to the objective function above suggest different assumptions about how you expect the data to be structured and/or how you want the data to be described. This should be a good reminder of just how important it is to specify your objective function properly.

8.2 Some extra points.

A few points to make after that to round things off.

First, eigendecomposition and SVD are very closely linked. If X is a square matrix, then the singular-value decomposition *is* the eigendecomposition of X. But even in the non-square case, the SVD contains a lot of interesting information. Take $X = U\Sigma V^T$, and recall that U and V are both orthogonal matrices and that orthogonal matrices K satisfy $K^T = K^{-1}$. Then:

1. $X^TX = (U\Sigma V^T)^T(U\Sigma V^T) = (V\Sigma^T U^T)(U\Sigma V^T) = V(\Sigma^T\Sigma)V^T$ is the eigendecomposition of X^TX. So V describes the eigenvectors of X^TX and $\Sigma^T\Sigma$ hold the eigenvalues.
2. $XX^T = (U\Sigma V^T)(U\Sigma V^T)^T = (U\Sigma V^T)(V\Sigma^T U^T) = (U\Sigma\Sigma^T U^T)$ is the eigendecomposition of XX^T. So U describes the eigenvectors of XX^T and $\Sigma\Sigma^T$ hold the eigenvalues.

(Note that the nonzero values of $\Sigma^T\Sigma$ and $\Sigma\Sigma^T$ are the same – one of the matrices just have more dimensions and zeros.)

Finally, a generalization and a gripe: the general form of an eigendecomposition may not have orthogonal eigenvectors but may still have an eigendecomposition $X = V\Lambda V^{-1}$ if X is full-rank. And technically, since SVD always deals with orthogonal matrices, we can write $X = U\Sigma V^{-1}$. Personally, I feel the inverse operator (V^{-1} means "do the operation that exactly counteracts V") is more intuitive to grasp/visualize than the transpose operator. Hopefully, this makes it a bit easier to sound out what "actions" a vector v hit by X undergoes, going right-to-left (we'll talk through eigendecomposition first):

1. First, we align v to have coordinates along the eigenvectors through V^{-1} (remember, this "undoes" the usual V, which would map the basis vectors to the eigenvectors – here, we map the eigenvectors to the basis vectors).
2. Next, we stretch along the axes according to how X would stretch the basis vectors via Λ.
3. Finally, we realign v to the proper output space with V.

An analogous description can be made for SVD, except it might feel slightly "weirder" because our output space may be different from our input space. But recall the discussion above: the idea is that there's a "true" space in which the vectors reside where all coordinates are orthogonal to each other, and the two other matrices are alternate "views" of the "true" vector in different coordinate systems (and potentially different dimension).

9 Jeez, tl;dr!

Alright, so this was longer than expected. But there's a reason for the length!

1. We explain how matrices are linear transformations. We highlight both the usual "right-to-left column-vector basis" interpretation as well as the more rarely used "left-to-right row vector basis" interpretation (which is useful for understanding LU decomposition).
    1. At this point, we described what the *transpose operator* actually "means" conceptually – letting our row-vectors serve as a basis while still doing multiplication right-to-left.
2. We discuss how LU decomposition is essentially Gaussian elimination (making sense of the matrices using the "row-vector" interpretation).

3. We discuss how the Cholesky decomposition is essentially Gaussian elimination with a clever trick that uses information about A being symmetric positive-definite.
    1. While there, we discuss the connections among covariance matrices, symmetric positive-definite matrices, and kernel functions.
    2. We also describe why a common practical use of Cholesky decomposition – covariance modeling – works.
4. We discuss how QR decomposition can be thought of as just the Gram-Schmidt process described as matrices.
5. We discuss how SVD involves describing the "latent" subspace on which the column-vectors of a matrix reside. We then map an input into this space, stretch along the axes, then map it to its output space.
    1. We also discuss the connections between SVD and eigendecomposition – how the U and V in SVD for X show up as the matrix of eigenvectors in the eigendecompositions of XX^T and X^TX, respectively.

It took a while to do all that discussing! But hopefully the post was not too prolix[5], and the read is worth the time and effort.

- DK

[5] ~~Anything for alliteration.~~