Nonnegative/Binary Matrix Factorization with a D-Wave Quantum Annealer

Daniel O'Malley (EES-16), Velimir V. Vesselinov (EES-16)
Boian S. Alexandrov (T-1), Ludmil B. Alexandrov (T-6)
Los Alamos National Laboratory

D-Wave Qubits Users Conference
September, 27 2017

LA-UR-17-23437

Los Alamos
NATIONAL LABORATORY
EST.1943

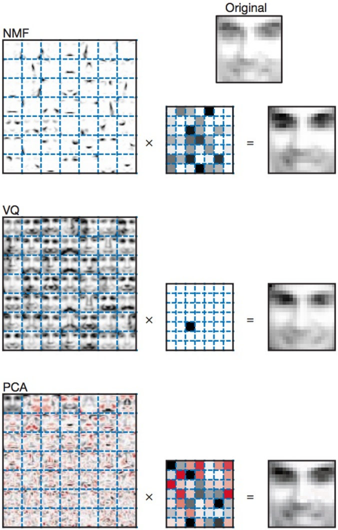
Matrix factorization is a fundamental applied math problem

- SVD: $A = U\Sigma V^*$ where Σ is diagonal, U, V are unitary
- QR: $A = QR$ where Q is orthogonal, R is upper triangular
- LU: $A = LU$ where L is lower triangular and R is upper triangular
- Cholesky: $A = LL^*$ where L is lower triangular
- NMF: $A \approx BC$ where $B_{ij} \geq 0$ and $C_{ij} \geq 0$
- D-Wave NMF: $A \approx BC$ where $B_{ij} \geq 0$ and $C_{ij} \in \{0, 1\}$

Low-rank matrix factorizations

$$\begin{bmatrix} \\ \\ \\ \\ & & & A & & & \\ \\ \\ \\ \end{bmatrix} \approx \begin{bmatrix} \\ \\ \\ \\ & B & \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} \\ & & & C & & & \\ \\ \end{bmatrix}$$

Unsupervised ML via matrix factorization

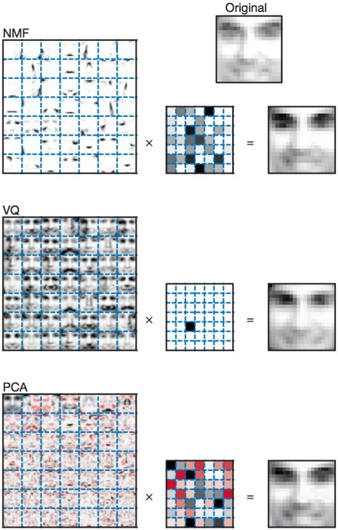


Lee & Seung (Nature, 1999)

$$A = BC$$

- Each column of A is a vectorized version of an image of a face
- Each row of A corresponds to a particular pixel in the images
- Each column of B is a "feature" that is used to reconstruct the image
- Each row of B corresponds to a particular pixel in the images
- Each column of C corresponds to an image and describes how each feature is present in the image
- Each row of C corresponds to a feature and describes how that feature is present in all the images

Unsupervised ML via matrix factorization on the D-Wave



Lee & Seung (Nature, 1999)

Are some of those features solid black? No

Pros/cons: D-Wave NMF versus classical NMF

Forget the D-Wave and just view this as a method

Pros

- The D-Wave NMF's C matrix is $\sim 85\%$ sparse, but classical NMF's C matrix is only $\sim 13\%$ sparse
- The components of the D-Wave NMF's C matrix require fewer bits than classical NMF's C matrix (1 bit vs. 64 bits)
- Viewed as lossy compression, the D-Wave NMF compresses more densely

Cons

- Classical NMF's reconstructions have slightly less than half as much error as D-Wave NMF's reconstructions
- Viewed as lossy compression, the D-Wave NMF loses more information
- The B matrices are about 40% sparse for classical NMF, but dense for D-Wave NMF

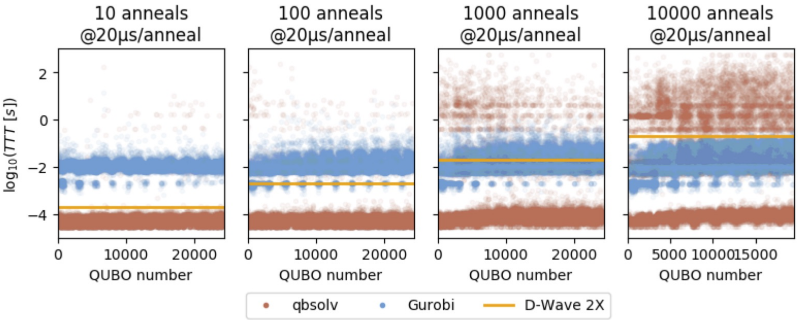
How to do it?

- Use "Alternating Least Squares"
  1. Randomly generate a binary C
  2. Solve $B = argmin_X ||A - XC||_F$ classically
  3. Solve $C = argmin_X ||A - BX||_F$ on the D-Wave
  4. Go to 2
- Step 3 is the interesting/D-Wave part
- In our analysis, A is 361×2429, B is 361×35 and C is 35×2429.
- C has $O(10^5)$ binary variables – far too many for the D-Wave, but. . .

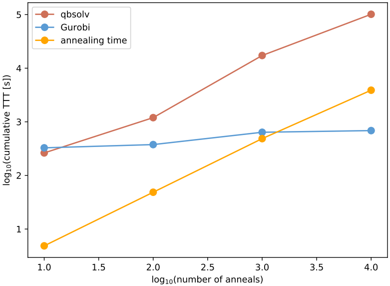
Step 3 in more detail

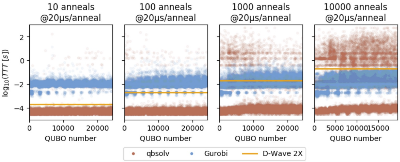
- $C = argmin_X ||A - BX||_F$ where C and X are 35×2429
- Step 3 is formulated above as a problem in 35×2429 binary variables, but it decomposes ("partitions") into 2429 problems with 35 binary variables each
- $C_i = argmin_x ||A_i - Bx||_2$ where C_i is the i^{th} column of C and x consists of 35 binary variables
- 35 binary variables fit on the D-Wave easily (can go to 49 with the VFYC)
- Imagine a Beowulf cluster of these...

What about performance?



What about performance?





- The D-Wave wins the cumulative time-to-targets modest number of anneals are used (up to 1000), but loses to Gurobi when 10,000 anneals are used
- qbsolv wins most problems, but loses very badly when it loses
- Gurobi takes too long to get rolling on the short time scales, but wins over longer times

What about performance including non-annealing time?

- Solving 2429 QUBOs repeatedly can take a long time unless you are careful
- Performance roadblocks
  - ThreeQ.jl "symbolic" mode
  - SAPI embedding
  - SAPI `async_solve_qubo`+`await_completion`+`p.result()`
- By overcoming the performance roadblocks, executing "Step 3" on a 361×2429 matrix can be done in a few minutes
- In the cumulative time-to-targets benchmark, qbsolv could sometimes lose even when I/O time was included

What about performance including non-annealing time?

ThreeQ.jl "symbolic" mode

```
function setupsmallqubo(A, B, j)
 m = ThreeQ.Model(...)
 @ThreeQ.defvar m Ccolj[1:size(B, 2)]
 for k = 1:size(B, 2)
  lincoeff = 0.0
  for i = 1:size(A, 1)
   lincoeff += B[i, k] * (B[i, k] - 2 * A[i, j])
  end
  @ThreeQ.addterm m lincoeff * Ccolj[k]
  for l = 1:k - 1
   quadcoeff = 0.0
   for i = 1:size(A, 1)
    quadcoeff += 2 * B[i, k] * B[i, l]
   end
   @ThreeQ.addterm m quadcoeff * Ccolj[k] *
Ccolj[l]
  end
 end
 return m, Ccolj

end
```

```
function setupsmallqubo(A, B, j)
 Q = zeros(size(B, 2), size(B, 2))
 for k = 1:size(B, 2)
  for i = 1:size(A, 1)
   Q[k, k] += B[i, k] * (B[i, k] - 2 * A[i, j])
  end
  for l = 1:k - 1
   for i = 1:size(A, 1)
    Q[k, l] += 2 * B[i, k] * B[i, l]
   end
  end
 end
 return Q

end
```

What about performance including non-annealing time?

SAPI embedding

- SAPI's `embed_problem` uses a "one-step" embedding process
  - Works great if you only have to embed the problem once
  - Slow if you have to embed the same problem repeatedly
- Wrote a custom replacement for SAPI's `embed_problem` using a "two-step" embedding process
  1. Find the couplings that are used as part of the embedding and determine how the coefficient will be spread across the couplers/qubits. Do this once.
  2. Use the result from step 1 to perform the embedding. Do this repeatedly.
- Also important to call `find_embedding` only once (obviously)

What about performance including non-annealing time?

SAPI async_solve_qubo+await_completion+p.result()

- Downloading 2429 results from the D-Wave system in serial is slow
  - I.e., 2429 calls to `p.result()` in serial is slow
- Use multiple processes to download results in parallel
  - Use `async_solve_qubo` then `await_completion` to wait for `nworkers()` results to be ready
  - Effectively perform one `p.result()` for each worker process to download the results in parallel
  - Actually, had to reimplement `p.result()` from scratch, probably due to an issue with julia's python interface
- Probably a lot of room for improvement here
  - For example, often don't need to download all the samples – just the best will do
  - Would be great to do the computation closer to the D-Wave to reduce round-trip time

Conclusions

- Utilized the D-Wave to solve a practical, unsupervised, machine-learning problem
- The D-Wave outperforms two state-of-the-art classical codes in a cumulative time-to-target benchmark when a low-to-moderate number of samples are used
  - Limitations in getting problems into/out of the D-Wave make these benefits hard to leverage, but the situation should improve with future D-Wave hardware
  - Custom heuristics would likely beat the D-Wave even in this benchmark
- Large datasets can be analyzed on the D-Wave with this algorithm
  - We factored a 361×2429 matrix for consistency with Lee & Seung (Nature, 1999), but going larger is not a problem
- The D-Wave only limits the rank of the factorization
  - Not a major limitation, because we *want* the rank to be small

Preview: PDE-constrained optimization on the D-Wave

2D elliptic PDE using a custom embedding that leverages the
virtual full yield chimera solver