3. Randomized SVD (RSVD)

3.1 Conceptual Overview

The goal of Randomized SVD (RSVD) is to efficiently compute a **low-rank approximation** of a large matrix $A \in \mathbb{R}^{m \times n}$:

$$ A \approx U_k \Sigma_k V_k^T, $$

where $k \ll \min(m, n)$. Instead of computing the full SVD (which costs $O(mn \min(m, n))$), RSVD uses **random projections** to quickly identify the dominant subspace of A, and then computes an exact SVD in that much smaller subspace.

High-level idea:

1. Hit A with a few random test vectors (columns of a random matrix Ω).
2. Collect the responses $Y = A\Omega$. These span (approximately) the same subspace as the top k singular vectors.
3. Orthonormalize Y to get a basis Q.
4. Project A into this subspace: $B = Q^T A$ is small.
5. Compute an **exact** SVD of the small matrix B, and lift the singular vectors back with Q.

This yields a near-optimal rank-k approximation at a cost roughly $O(mnk)$, which is much smaller than full SVD for $k \ll \min(m, n)$.

3.2 Algorithm Description

Given $A \in \mathbb{R}^{m \times n}$, target rank k, and oversampling parameter p (e.g. $p = 10$), set $\ell = k + p$.

Step 1: Draw a random test matrix

Draw a Gaussian random matrix

$$ \Omega \in \mathbb{R}^{n \times \ell} $$

whose entries are i.i.d. $\mathcal{N}(0, 1)$.

**Step 2: Sample the range of **A

Compute the sample matrix

$$ Y = A\Omega \in \mathbb{R}^{m \times \ell}. $$

Each column of Y is a random linear combination of the columns of A, with more weight on directions where A has larger singular values.

**Step 3: Compute an orthonormal basis for **$\mathrm{range}(A)$

Compute a QR factorization of Y:

$$ Y = QR, $$

where $Q \in \mathbb{R}^{m \times \ell}$ has orthonormal columns and $R \in \mathbb{R}^{\ell \times \ell}$ is upper triangular.

The columns of Q form an approximate basis for the column space spanned by the top singular vectors of A. Intuitively, Q captures the directions where A acts "strongly".

Step 4: Project A into the low-dimensional subspace

Form the smaller matrix

$$ B = Q^T A \in \mathbb{R}^{\ell \times n}. $$

Since Q has only $\ell = k + p$ columns, B is much smaller than A. Moreover,

$$ A \approx QB. $$

Step 5: Compute an exact SVD of the small matrix

Compute the (deterministic) SVD of B:

$$ B = \tilde{U} \Sigma V^T, $$

where $\tilde{U} \in \mathbb{R}^{\ell \times \ell}$, $\Sigma \in \mathbb{R}^{\ell \times n}$ (diagonal singular values), and $V \in \mathbb{R}^{n \times n}$.

Step 6: Lift singular vectors back to the original space

Define

$$ U = Q\tilde{U} \in \mathbb{R}^{m \times \ell}. $$

Then we have the approximate factorization

$$ A \approx U\Sigma V^T. $$

Truncate to the top k singular values/vectors:

- $U_k = U_{:,1:k}$

- $\Sigma_k = \Sigma_{1:k,1:k}$
- $V_k = V_{:,1:k}$

Yielding the rank-k approximation

$$ A_k = U_k \Sigma_k V_k^T. $$

3.4 Complexity Analysis

Assume $m \geq n$ for simplicity.

1. **Forming **$Y = A\Omega$
2. A is $m \times n$, Ω is $n \times \ell$.
3. Cost: $O(mn\ell)$.
4. **QR factorization of **Y
5. Y is $m \times \ell$, with $\ell = k + p \ll n$.
6. Cost: $O(m\ell^2)$.
7. **Forming **$B = Q^T A$
8. Q^T is $\ell \times m$, A is $m \times n$.
9. Cost: $O(mn\ell)$.
10. **SVD of **B
11. B is $\ell \times n$ with $\ell \ll n$.
12. Cost: $O(\ell^2 n)$.

Total leading cost:

$$ O(mn\ell + m\ell^2 + \ell^2 n) \approx O(mn\ell) = O(mnk), $$

for modest oversampling p.

By contrast, a classical deterministic SVD costs:

- $O(mn^2)$ if $m \geq n$,

- or $O(m^2 n)$ if $m < n$.

Hence, when $k \ll \min(m, n)$, RSVD provides a substantial computational savings.

With q steps of power iteration, the cost of forming Y and B scales like $O((2q+1)mn\ell)$ since each power step requires extra multiplications by A and A^T.

3.5 Approximation Quality and Error Intuition

Let the singular value decomposition of A be

$$ A = U \Sigma V^T, $$

with singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$.

The **optimal** rank-k approximation (truncated SVD) is

$$ A_k = U_k \Sigma_k V_k^T, $$

with error

$$ |A - A_k|_2 = \sigma_{k+1}, \quad |A - A_k|_F^2 = \sum_{i>k} \sigma_i^2. $$

RSVD, with oversampling p and possibly power iterations, achieves an approximation A_k^{rsvd} whose spectral and Frobenius norm errors are provably within a small factor of these optimal values (in expectation). Intuitively:

- If A has **fast-decaying singular values** (approximately low-rank), the sampled subspace captured in Q is almost identical to the span of U_k.
- Oversampling (p) and power iteration (q) make it increasingly unlikely that any important singular direction is missed.

In practice, for many ML and data matrices (neural network weights, embedding tables, user-item matrices), even a modest k and small p yield errors nearly indistinguishable from the exact truncated SVD.

3.6 Implementation Details (Python)

A practical RSVD implementation in Python typically follows these design choices:

- Use NumPy or PyTorch for matrix multiplications to leverage BLAS.
- Generate Ω as a dense Gaussian matrix or, for very large problems, a structured random matrix (e.g., subsampled randomized Fourier transform) to reduce costs.
- Apply QR factorization (`np.linalg.qr`) or a numerically stable variant to obtain Q.

- Use a standard SVD routine on the small matrix B.
- Carefully manage data types (float32 vs float64) to balance speed and numerical stability.

Pseudo-code sketch:

```
def rsvd(A, k, p=10, q=0):
    m, n = A.shape
    ell = k + p
    Omega = np.random.randn(n, ell)
    Y = A @ Omega
    for _ in range(q):
        Y = A @ (A.T @ Y)
    Q, _ = np.linalg.qr(Y, mode="reduced")
    B = Q.T @ A
    U_tilde, S, Vt = np.linalg.svd(B, full_matrices=False)
    U = Q @ U_tilde
    U_k = U[:, :k]
    S_k = S[:k]
    Vt_k = Vt[:k, :]
    return U_k, S_k, Vt_k
```

3.7 Experimental Workflow for RSVD

To study RSVD empirically, we consider the following matrix families:

1. **Dense Gaussian matrices** (no structure, worst case for low-rank approximation).
2. **Synthetic low-rank matrices**: $A = U_r \Sigma_r V_r^T$ with small rank r and optional noise.
3. **Sparse matrices**: generated with different sparsity levels to mimic graph or text applications.
4. **Neural-network-like matrices**: real weight matrices from fully connected layers or embeddings, which typically exhibit fast singular value decay.

For each matrix type and size, we:

- Compute the exact top-k truncated SVD using a deterministic method (for ground truth, when feasible).
- Compute RSVD with various k, oversampling p, and power iteration q.
- Measure:
- Spectral or Frobenius norm error $\|A - A_k^{\text{rsvd}}\| / \|A\|$.
- Runtime for RSVD vs full SVD.
- How error and runtime change as k, p, and q vary.

We then plot:

- Error vs. rank k for each matrix family.
- Error vs. oversampling p.

- Runtime vs. matrix size for RSVD and full SVD.

These experiments reveal that RSVD achieves near-optimal low-rank approximations with significantly lower runtime, especially on matrices with strong low-rank structure (neural network weights, recommender matrices, kernel matrices).

3.8 Practical Applications of RSVD

RSVD is widely applicable wherever low-rank structure is present:

- **Dimensionality reduction / PCA** on large datasets.
- **Recommender systems**: approximating large user-item matrices with a small number of latent factors.
- **Neural network compression**: approximating weight matrices to reduce parameters and computation.
- **Signal and image processing**: denoising and compression using low-rank models.

In our broader project, RSVD also serves as the **front-end** for low-rank factorized matrix multiplication: we first obtain U_k, Σ_k, V_k^T via RSVD, and then use these factors to speed up downstream matrix multiplications.

3.9 Conclusion

Randomized SVD provides an efficient, probabilistic alternative to classical SVD for computing low-rank approximations of large matrices. By combining random projections, orthonormalization, and a small deterministic SVD, RSVD recovers the dominant singular subspace with high accuracy at a cost proportional to $O(mnk)$. This makes it particularly powerful in modern applications where matrices are large but effectively low-rank, such as deep learning, recommender systems, and large-scale data analysis. It also forms a crucial building block for our subsequent low-rank factorized matrix multiplication algorithm.