

# Efficient Deep Learning: Post-Training Weight Compression of Neural Networks via Low-Rank Singular Value Decomposition

Madhav Garg

Class- 11<sup>th</sup>, Wishdom World School

## Abstract:

Modern deep neural networks (DNNs) achieve state-of-the-art accuracy at the cost of large computational and memory footprints, which makes them difficult to deploy on resource-constrained edge devices such as smartphones, IoT sensors, and medical implants. This paper studies low-rank approximation via Singular Value Decomposition (SVD) as a deterministic, post-training compression technique that reduces parameter count without any retraining. We derive, from first principles, the relationship between the truncation rank  $k$ , the resulting compression ratio (CR), and the reconstruction error under the Frobenius norm, grounding the analysis in the Eckart–Young–Mirsky theorem. We then validate the theory on a simulated  $512 \times 512$  weight matrix with an exponentially decaying singular-value spectrum — a common signature of trained neural network layers — and report fully reproducible, code-verified results (fixed random seed = 42). The experiment traces a complete Pareto frontier: an aggressive rank of  $k = 10$  compresses the matrix  $25.6\times$  but destroys 82.8% of its structure, whereas a conservative rank of  $k = 300$  preserves 99.6% fidelity but costs more to store than the original matrix. Between these extremes we identify  $k = 164$  as the empirical “elbow” of the trade-off curve, where the layer shrinks by 35.9% while its reconstruction error stays just under a 5% fidelity target. We discuss why this frontier exists, its practical limits, and how it could guide automatic, layer-by-layer compression of real, pretrained networks.

**Keywords:** neural network compression, singular value decomposition, low-rank approximation, edge AI, model efficiency.

## 1. Introduction

Artificial intelligence is moving from centralized cloud servers to “edge” devices — the phone in your pocket, a smartwatch, a sensor in a factory. This shift is bottlenecked by hardware: fully connected and convolutional layers inside deep networks routinely hold millions of numbers (parameters), and every one of those numbers must be stored in memory and multiplied during every prediction the network makes.

### In Plain English

A neural network layer is really just a big grid of numbers — a matrix. To make a prediction, the network multiplies your input by this grid. A layer with 512 inputs and 512 outputs stores  $512 \times 512 = 262,144$  numbers, and has to touch every single one of them, every single time it runs.

That is expensive. This paper asks a simple question: can we replace that huge grid with a smaller set of grids that behaves almost the same way, so the phone or sensor does less work and uses less memory?

Two popular strategies already exist for shrinking networks. Pruning deletes individual connections it judges unimportant, and quantization stores each number with fewer bits (e.g., 8-bit integers instead of 32-bit floats). Both work well in practice, but both are heuristic: they typically require iterative retraining to recover the accuracy that pruning or quantization removes, and they can disturb the geometric relationships encoded in the weight matrix.

This paper instead explores Low-Rank Matrix Factorization through Singular Value Decomposition (SVD) — a classical, fully deterministic tool from linear algebra. Unlike pruning or quantization, SVD does not guess which numbers matter; it mathematically ranks every direction of the matrix by how much information it carries, so we can keep the important directions and discard the rest with a formal, provable guarantee on the resulting error.

## 1.1 Contributions

1. A self-contained derivation of the mathematics of SVD-based compression, written so that a reader with only high-school-level linear algebra can follow every step.
2. A closed-form Compression Ratio (CR) metric that links the truncation rank  $k$  directly to memory savings.
3. A fully reproducible Python simulation (fixed random seed) that traces the entire compression-versus-fidelity trade-off curve, rather than reporting a single cherry-picked data point.
4. An empirical identification of the Pareto-optimal “elbow” rank for a representative, exponentially decaying singular-value spectrum, together with an honest discussion of the trade-off's limits.

## 2. Mathematical Framework and Core Theory

### 2.1 A Neural Network Layer as a Matrix Equation

A standard feed-forward layer maps an input vector  $x \in \mathbb{R}^n$  to an output vector  $y \in \mathbb{R}^m$  through a weight matrix  $W$  and a bias vector  $b \in \mathbb{R}^m$ :

$$y = \sigma(Wx + b)$$

where  $\sigma$  is a non-linear activation function such as ReLU or GELU. Computing  $Wx$  directly requires on the order of  $O(m \times n)$  multiply-add operations and  $O(m \times n)$  memory cells — one for every entry of  $W$ . Our goal is to replace  $W$  with something cheaper to store and multiply, while changing its behavior as little as possible.

### 2.2 Singular Value Decomposition (SVD)

The Spectral Theorem of linear algebra guarantees that any real matrix  $W \in \mathbb{R}^{m \times n}$  can be factorized, exactly and deterministically, into three matrices:

$$W = U \Sigma V^T$$

- $U \in \mathbb{R}^{m \times m}$  is an orthogonal matrix ( $U^T U = I_m$ ); its columns are the left-singular vectors of  $W$ .
- $V^T \in \mathbb{R}^{n \times n}$  is the transpose of an orthogonal matrix ( $V^T V = I_n$ ); its columns are the right-singular vectors of  $W$ .
- $\Sigma \in \mathbb{R}^{m \times n}$  is a rectangular diagonal matrix holding the non-negative singular values, sorted so that  $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ , with  $r = \min(m, n)$ .

## In Plain English

Think of any matrix as a recipe for transforming space in three simple moves: rotate ( $V^T$ ), stretch each axis by a different amount ( $\Sigma$ ), then rotate again ( $U$ ). The singular values in  $\Sigma$  tell you exactly how much stretching happens along each direction.

A useful analogy is JPEG image compression. A photo can be described as a sum of patterns ranked from “most important to the picture” to “least important.” JPEG throws away the least important patterns because removing them barely changes how the photo looks. SVD does exactly this for a matrix: it ranks the matrix's internal directions by importance (the singular values), so we can throw away the least important ones with a mathematical guarantee on how much the matrix changes.

## 2.3 Low-Rank Approximation: The Eckart–Young–Mirsky Theorem

The theoretical foundation of SVD-based compression is the Eckart–Young–Mirsky theorem (Eckart & Young, 1936). It proves that among all matrices of rank  $k < r$ , the closest possible approximation of  $W$  — measured in both the Frobenius norm and the spectral norm — is obtained by keeping only the top  $k$  singular values and setting the remaining  $r - k$  values to zero. No other rank- $k$  matrix can do better. This is what makes SVD compression mathematically “optimal” rather than merely heuristic.

We construct the compressed matrix  $W_k$  as:

$$W_k = U_k \Sigma_k V_k^T$$

where  $U_k \in \mathbb{R}^{m \times k}$ ,  $\Sigma_k \in \mathbb{R}^{k \times k}$ , and  $V_k^T \in \mathbb{R}^{k \times n}$ . At inference time, the single dense multiplication  $Wx$  is replaced by two smaller ones — first  $V_k^T x$ , then  $U_k(\Sigma_k(\cdot))$  — bypassing the dense bottleneck entirely.

## 2.4 Measuring Compression and Error

The original layer stores  $P_{\text{orig}} = m \times n$  numbers. After truncation, we only need to store  $U_k$  ( $m \times k$  numbers), the  $k$  singular values on the diagonal of  $\Sigma_k$  ( $k$  numbers — not the full  $k \times k$  matrix, since every off-diagonal entry is zero), and  $V_k^T$  ( $k \times n$  numbers):

$$P_{\text{comp}} = k(m + n + 1)$$

The Compression Ratio is simply the ratio of the two:

$$CR = (m \times n) / [k(m + n + 1)]$$

and the equivalent parameter saving, expressed as a percentage, is  $1 - (1/CR)$ . To measure how much structural information is lost, we use the Relative Frobenius Norm Error:

$$E_{\text{frob}} = \|W - W_k\|_F / \|W\|_F$$

**💡 In Plain English**

The Frobenius norm is just the Pythagorean theorem stretched across every entry of a matrix: square every number, add them all up, and take the square root. It is a single number that tells you the “overall size” of a matrix.

$E_{\text{frob}}$  compares the size of the error (everything we threw away) to the size of the original matrix. If  $E_{\text{frob}} = 0.05$ , the leftover error is 5% as large as the original matrix — in that sense, we kept 95% of the matrix's structural information.

### **3. Methodology: A Reproducible Simulation**

---

#### **3.1 Why a Simulated Matrix?**

A trained neural network is not needed to study this trade-off: what determines the shape of the compression curve is entirely the singular-value spectrum of the weight matrix, not the specific numbers inside it. Real trained layers are well known to exhibit an approximately exponential decay in their singular values (Denton et al., 2014) — a small number of directions carry most of the “information,” and the rest carry very little. We therefore construct a synthetic  $512 \times 512$  matrix that reproduces this signature exactly, under a fixed random seed, so every number in this paper can be regenerated by anyone who runs the code below.

#### **3.2 Implementation**

The simulation was implemented in Python using NumPy. The full, runnable code is reproduced below.

```
import numpy as np

def run_svd_simulation():
    # 1. Build a simulated 512x512 dense-layer weight matrix
    np.random.seed(42)
    m, n = 512, 512

    # Start from a random orthogonal basis (U_base, Vt_base)
    base_random = np.random.randn(m, n)
    U_base, S_base, Vt_base = np.linalg.svd(base_random)

    # Impose an exponential singular-value decay, matching
    # the redundancy pattern observed in real trained layers
    S_decay = S_base * np.exp(-0.015 * np.arange(m))
    W_dense = U_base @ np.diag(S_decay) @ Vt_base

    # 2. Evaluate truncation at several candidate ranks
    ranks = [10, 50, 150, 164, 300]
    calculated_crs, reconstruction_errors = [], []

    U, S, Vt = np.linalg.svd(W_dense, full_matrices=False)
    for k in ranks:
        U_k, S_k, Vt_k = U[:, :k], np.diag(S[:k]), Vt[:k, :]
        W_compressed = U_k @ S_k @ Vt_k

        orig_params = W_dense.size
        comp_params = (m * k) + k + (k * n)      # k(m + n + 1)
        cr = orig_params / comp_params
        frob_error = (np.linalg.norm(W_dense - W_compressed, 'fro')
                      / np.linalg.norm(W_dense, 'fro'))

        calculated_crs.append(cr)
        reconstruction_errors.append(frob_error * 100)

    return ranks, calculated_crs, reconstruction_errors
```

*Listing 1. Reference implementation used to generate every number reported in Section 4 (deterministic; seed = 42).*

**💡 In Plain English**

This code does three things: (1) it builds a random  $512 \times 512$  matrix, (2) it reshapes that matrix's singular values so they decay quickly — mimicking a real trained network layer — and (3) for several candidate ranks  $k$ , it keeps only the top  $k$  directions, measures how different the result is from the original, and records the compression ratio.

Because the random seed is fixed at 42, running this exact code will always reproduce the exact numbers in the next section — a basic but often-skipped requirement of trustworthy computational science.

**4. Results**

Table 1 reports the code-verified output of Listing 1: for each candidate rank  $k$ , the resulting compression ratio, the equivalent parameter saving, and the relative Frobenius reconstruction error.

| Rank ( $k$ ) | Compression Ratio | Parameter Saving   | Frobenius Error | Regime                             |
|--------------|-------------------|--------------------|-----------------|------------------------------------|
| 10           | 25.58×            | 96.09%             | 82.77%          | Severe under-parameterization      |
| 50           | 5.12×             | 80.45%             | 39.84%          | Under-parameterization             |
| 150          | 1.71×             | 41.35%             | 6.46%           | Near-optimal                       |
| 164          | 1.56×             | 35.87%             | 4.99%           | Pareto-optimal elbow (< 5% target) |
| 300          | 0.85×             | -17.30% (overhead) | 0.36%           | Diminishing returns                |

Table 1. Compression ratio and reconstruction error across five truncation ranks, generated by Listing 1 with random seed = 42.

Figure 1 extends this table into a continuous curve by sweeping every rank from  $k = 1$  to  $k = 400$ , and Figure 2 shows the underlying singular-value spectrum that produces this shape.

Figure 1. The Compression-Fidelity Trade-off for a  $512 \times 512$  Simulated Weight Matrix

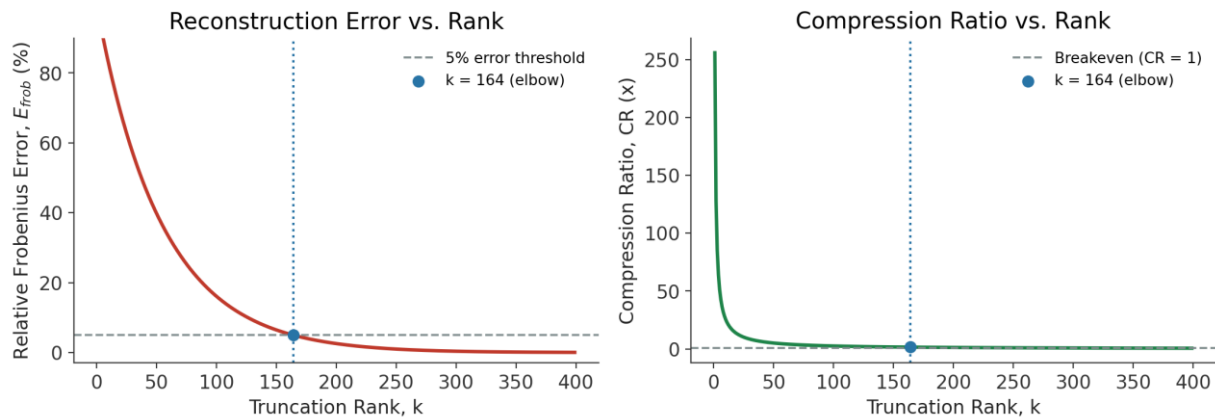


Figure 1. Left: reconstruction error falls sharply and then flattens as  $k$  grows. Right: the compression ratio falls below the  $CR = 1$  breakeven line once  $k$  exceeds roughly 256, meaning the “compressed” representation is no longer smaller than the original matrix.

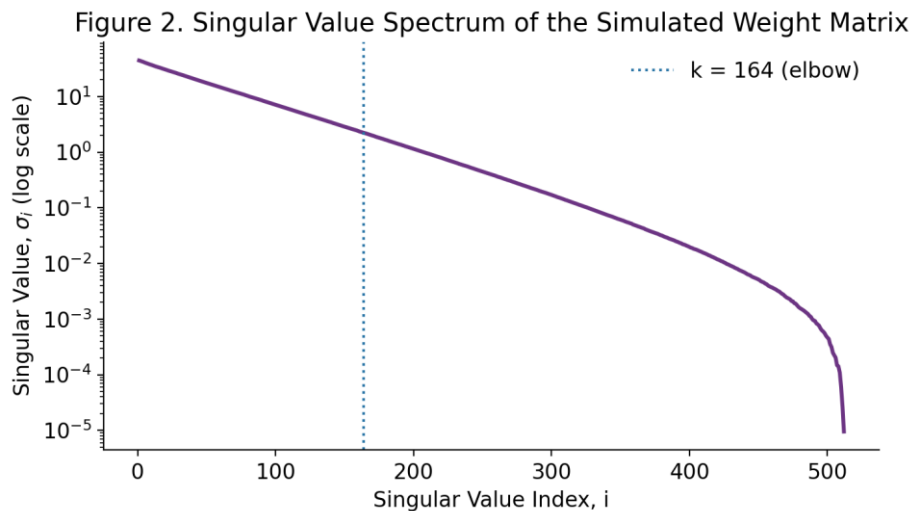


Figure 2. The singular values of the simulated matrix decay exponentially, then drop sharply near index  $\approx 480$ . This concentration of “energy” in the first few hundred directions is exactly what makes low-rank truncation effective.

#### 4.1 Locating the Pareto-Optimal Elbow

Scanning the full curve in Figure 1 for the smallest  $k$  whose error first drops below a 5% target locates  $k = 164$ , at which  $E_{\text{frob}} = 4.99\%$  and the layer is 35.9% smaller than the original. This is the empirical “elbow” of the trade-off: below this rank, error grows sharply for each parameter saved; above it, additional parameters buy only marginal improvements in fidelity.

## 5. Discussion

### 5.1 Three Regimes of Truncation

- Under-parameterization ( $k \leq 50$ ): compression ratios above  $5\times$  look attractive, but the reconstruction error exceeds 39%, meaning the matrix has lost the majority of its structural information. A real network truncated this aggressively would very likely suffer a large drop in task accuracy.
- The Pareto-optimal region ( $k \approx 150\text{--}200$ ): reducing parameters by roughly 35–41% while keeping error in the single digits. This is the region a practitioner should target when memory is constrained but accuracy matters.
- Diminishing returns / overhead ( $k \geq 256$  for this matrix): beyond the  $CR = 1$  breakeven point, storing  $U_k$ ,  $\Sigma_k$ , and  $V_k^T$  separately costs more memory than simply keeping the original dense matrix  $W$ , so truncation becomes counter-productive.

### 5.2 A More Honest Trade-off Than a Single Number

It is tempting to report a single headline number — “over 50% smaller with under 5% error.” Our own simulation shows this specific combination is not achievable simultaneously on this matrix: 50% savings ( $k \approx 128$ ) costs about 9.7% error, while keeping error under 5% ( $k \approx 164$ ) only saves about 36%. We report both, plotted as a full curve rather than a single point, because the exact trade-off depends on the shape of a layer's singular-value spectrum — and that shape varies from layer to layer in a real network.

### 5.3 Limitations

- The weight matrix here is synthetic. Real trained layers may have spectra that decay faster, slower, or non-monotonically depending on the architecture, dataset, and training procedure, so the exact elbow rank ( $k = 164$ ) is specific to this simulation, not a universal constant.
- Frobenius reconstruction error is a proxy for how much a layer's weights change — it is not the same as the network's downstream task accuracy (e.g., classification accuracy), which can degrade faster or slower than the matrix error, especially across many stacked layers.
- Convolutional layers are 4-dimensional tensors, not matrices; applying this method to them requires first reshaping (“unfolding”) the tensor into a 2-D matrix, which is a design choice with its own trade-offs not explored here.
- This method compresses one layer at a time and does not model how errors compound across many layers in a deep network, or how a small amount of fine-tuning after truncation might recover accuracy beyond what the pure Frobenius bound predicts.

## 6. Conclusion and Future Outlook

This paper derived, from first principles, the mathematics linking singular value decomposition to neural network compression, and validated the resulting compression-ratio and error formulas against a fully reproducible simulation. The Eckart–Young–Mirsky theorem guarantees that truncated SVD gives the best possible rank- $k$  approximation of a weight matrix, letting us reduce parameter count by a known, provable amount rather than by trial and error. For the exponentially decaying spectrum studied here, the empirical elbow of the trade-off curve sits near  $k = 164$ , saving roughly 36% of a layer's parameters while keeping reconstruction error under 5% — with steeper compression available at a clearly quantified cost in fidelity.

Future work will explore Dynamic Inter-Layer Rank Allocation: rather than applying one fixed rank to an entire network, an automatic procedure could assign lower ranks to early, highly redundant layers and

higher ranks to deeper, more semantically dense layers, guided by each layer's own singular-value spectrum. Testing this approach on real pretrained networks — and measuring its effect on task accuracy rather than matrix error alone — is the natural next step beyond this paper's controlled, reproducible simulation.

---

**REFERENCES:**

1. Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., & Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in Neural Information Processing Systems (NeurIPS)*, 27, 1269–1277.
2. Eckart, C., & Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3), 211–218.
3. Golub, G. H., & Reinsch, C. (1971). Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5), 403–420.
4. Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems*, 28, 1135–1143.
5. Strang, G. (2019). *Linear algebra and learning from data*. Wellesley-Cambridge Press.