# Compression of Neural Networks

Ittiam Systems

Pratyush Kant
pratyushkant@iisc.ac.in
Indian Institute of Science

July 30, 2025

## Contents

# Part I: Theoretical Foundations

# 1 Introduction and Problem Statement

Low-rank matrix approximations, such as the truncated singular value decomposition (SVD) and the rank-revealing QR decomposition, are fundamental tools in scientific computing, data analysis, and numerical linear algebra. This paper outlines a general framework for utilizing randomized algorithms to compute such approximations with high efficiency and accuracy. The central insight is that randomized methods can outperform traditional deterministic techniques in terms of computational speed, numerical stability, and parallel scalability, especially for large-scale problems arising in modern applications.

## 1.1 Core Problem

Let $A \in \mathbb{R}^{m \times n}$ be a given matrix. The goal is to compute a low-rank approximation of the form

$$A \approx BC,$$

where $B \in \mathbb{R}^{m \times k}$, $C \in \mathbb{R}^{k \times n}$, and the target rank $k$ satisfies $k \ll \min\{m, n\}$. Such a factorization enables significant data compression and allows for efficient matrix computations, particularly when $k$ is much smaller than the ambient dimensions $m$ and $n$.

## 1.2 Problem Formulations

Two common formulations of the low-rank approximation problem are considered:

1. **Fixed-precision problem:** Given a user-specified error tolerance $\varepsilon > 0$, find a matrix $Q \in \mathbb{R}^{m \times \ell}$ with orthonormal columns such that

   $$\|A - QQ^*A\| \leq \varepsilon,$$

   where $\|\cdot\|$ denotes the spectral norm (i.e., the operator 2-norm). The matrix $QQ^*A$ then represents an approximation of $A$ whose image lies in the $\ell$-dimensional subspace spanned by the columns of $Q$.

2. **Fixed-rank problem:** Given a target rank $k$ and an oversampling parameter $p \geq 0$, compute a matrix $Q \in \mathbb{R}^{m \times (k+p)}$ with orthonormal columns such that

   $$\|A - QQ^*A\| \approx \min_{\text{rank}(X) \leq k} \|A - X\|,$$

   where the right-hand side denotes the optimal error achievable by any rank-$k$ approximation of $A$. This formulation seeks a subspace that captures the dominant action of $A$ in the best possible rank-$k$ sense.

# 2 Two-Stage Algorithmic Framework

Randomized algorithms for low-rank approximation are typically organized into two computational phases:

- **Stage A (Randomized subspace identification):** Construct an orthonormal matrix $Q$ such that the range of $Q$ approximates the range of $A$. Specifically, one seeks $Q$ satisfying

  $$A \approx QQ^*A,$$

  where $Q$ is obtained via randomized sampling techniques, such as applying $A$ to a randomly generated test matrix.

- **Stage B (Deterministic post-processing):** Given the matrix $Q$ from Stage A, compute a standard low-rank factorization of $A$ (e.g., SVD, QR, interpolative decomposition) using deterministic numerical linear algebra applied to the smaller projected matrix $B = Q^*A$.

This two-stage structure offers a balance between efficiency and accuracy: Stage A leverages randomization to reduce dimensionality quickly, while Stage B refines the approximation using robust, well-established methods.

# 3 Stage A: Randomized Range Finding

The goal of Stage A is to identify an orthonormal matrix $Q \in \mathbb{R}^{m\times\ell}$ whose columns approximate the column space (range) of the input matrix $A \in \mathbb{R}^{m\times n}$, with $\ell = k + p$ where $k$ is the target rank and $p \geq 0$ is an oversampling parameter. This is achieved using randomized projections.

## 3.1 Basic Algorithms

### 3.1.1 Proto-Algorithm

The following high-level procedure outlines the essential idea behind randomized range finding. It provides a low-dimensional embedding of the column space of $A$.

**Algorithm 1** Proto-Algorithm for Randomized Range Finding

1: Draw a random test matrix $\Omega \in \mathbb{R}^{n\times(k+p)}$
2: Form the sample matrix $Y = A\Omega \in \mathbb{R}^{m\times(k+p)}$
3: Construct an orthonormal matrix $Q \in \mathbb{R}^{m\times(k+p)}$ whose columns span the range of $Y$

This proto-algorithm provides a simple and powerful approach to approximate the range of $A$, especially when $k \ll \min\{m, n\}$. It forms the basis of more refined techniques described below.

### 3.1.2 Randomized Range Finder

A concrete instantiation of the above idea is the following algorithm, which uses Gaussian random matrices to probe the range of $A$.

**Algorithm 2** Randomized Range Finder

1: **Input:** Matrix $A \in \mathbb{R}^{m\times n}$, target embedding dimension $\ell$
2: Draw a Gaussian random matrix $\Omega \in \mathbb{R}^{n\times\ell}$ with i.i.d. entries $\mathcal{N}(0, 1)$
3: Form the sample matrix $Y = A\Omega \in \mathbb{R}^{m\times\ell}$
4: Compute the thin QR factorization $Y = QR$, where $Q \in \mathbb{R}^{m\times\ell}$ has orthonormal columns
5: **Output:** Matrix $Q$ approximating the range of $A$

This algorithm captures the dominant subspace of $A$ with high probability when $\ell$ is slightly larger than the numerical rank $k$ of $A$.

**Computational Cost.** The approximate number of floating-point operations required by Algorithm 2 is

$$T_{\text{basic}} \sim \ell n \cdot T_{\text{rand}} + \ell \cdot T_{\text{mult}} + \ell^2 m,$$

where:

- $T_{\text{rand}}$ is the cost of generating a single Gaussian random number,
- $T_{\text{mult}}$ is the cost of computing a matrix-vector product involving $A$,
- the final term accounts for orthogonalizing the $\ell$ columns of $Y$.

**Implementation Remarks.**

- **Orthonormalization:** To ensure numerical stability, the matrix $Y$ should be orthonormalized using a stable method such as Householder QR, modified Gram-Schmidt with reorthogonalization, or Givens rotations.
- **Randomness quality:** The performance of the algorithm is largely insensitive to the quality of the random number generator. Gaussian random matrices with i.i.d. entries are sufficient to ensure near-optimal performance in most practical settings.
- **No need for pivoting:** Unlike deterministic QR-based techniques, randomized methods typically do not require pivoting. This is because the test matrix $\Omega$ is constructed with i.i.d. entries, ensuring that the sample matrix $Y = A\Omega$ consists of independent random linear combinations of the columns of $A$. Even when $A$ has highly dependent columns, the randomness ensures that the probability of generating nearly linearly dependent columns in $Y$ is negligible. As a result, $Y$ is numerically well-conditioned with high probability, and the span of its columns captures the dominant action of $A$.

### 3.2 Choice of Oversampling Parameter

To ensure that a low-rank approximation captures the dominant action of the matrix $A \in \mathbb{R}^{m\times n}$, it is typical to select a target embedding dimension $\ell = k + p$, where $k$ is the desired approximation rank and $p \geq 0$ is an oversampling parameter. The matrix $Q \in \mathbb{R}^{m\times \ell}$ should then satisfy the spectral-norm error guarantee:

$$\|(I - QQ^*)A\| \leq \varepsilon,$$

for some prescribed error tolerance $\varepsilon > 0$.

**Guidelines for Choosing $p$:**

- For Gaussian test matrices, oversampling by $p = 5$ or $p = 10$ is generally sufficient to ensure accurate approximation with high probability.
- Larger values of $p$ typically offer diminishing returns; choosing $p > k$ is rarely beneficial.
- Matrices with rapidly decaying singular spectra require less oversampling.
- For large-scale matrices, slightly larger $p$ may help guard against numerical instability or enhance robustness.

### 3.3 Adaptive Range Finding

In scenarios where the target rank $k$ is unknown a priori, one may construct the basis $Q$ incrementally until the residual energy falls below a specified tolerance. The following adaptive scheme, based on randomized sampling and residual estimation, accomplishes this goal.

**Algorithm 3** Adaptive Randomized Range Finder

**Input:** Matrix $A \in \mathbb{R}^{m\times n}$, error tolerance $\varepsilon > 0$, batch size $r$ (e.g., $r = 10$)
Draw $r$ independent Gaussian vectors $\omega^{(1)}, \ldots, \omega^{(r)} \in \mathbb{R}^n$
Compute $y^{(i)} = A\omega^{(i)}$ for $i = 1, \ldots, r$
Initialize $j = 0$, $Q^{(0)} = [\,]$
**while** $\max\left\{\|y^{(j+1)}\|, \ldots, \|y^{(j+r)}\|\right\} > \dfrac{\varepsilon}{10\sqrt{2/\pi}}$ **do**
    $j \leftarrow j + 1$
    $y^{(j)} \leftarrow (I - Q^{(j-1)}(Q^{(j-1)})^*)y^{(j)}$
    $q^{(j)} \leftarrow y^{(j)}/\|y^{(j)}\|$
    $Q^{(j)} \leftarrow [Q^{(j-1)}, q^{(j)}]$
    Draw new $\omega^{(j+r)}$, compute

$$y^{(j+r)} = (I - Q^{(j)}(Q^{(j)})^*)A\omega^{(j+r)}$$

    Orthogonalize all remaining $y^{(i)}$ against $q^{(j)}$
**end while**
**Output:** $Q = Q^{(j)}$

**Theoretical Guarantee.** With probability at least $1 - 10^{-r}$, the final output $Q$ satisfies

$$\|(I - QQ^*)A\| \le 10\sqrt{\frac{2}{\pi}} \cdot \max_{1\le i\le r} \left\|(I - QQ^*)A\omega^{(i)}\right\|.$$

### 3.4 Power Iteration for Slowly Decaying Spectra

When the singular values of $A$ decay slowly, the basic range finder may fail to capture the numerical rank efficiently. To enhance spectral decay and improve accuracy, one can apply a power iteration.

**Algorithm 4** Randomized Subspace Iteration (Power Scheme)

**Input:** Matrix $A \in \mathbb{R}^{m\times n}$, integers $\ell$ (embedding dimension), $q$ (power iteration exponent)
Draw a Gaussian test matrix $\Omega \in \mathbb{R}^{n\times \ell}$
Form $Y_0 = A\Omega$, compute QR factorization $Y_0 = Q_0R_0$
**for** $j = 1$ to $q$ **do**
    Form $\tilde{Y}_j = A^*Q_{j-1}$, compute QR: $\tilde{Y}_j = \tilde{Q}_j\tilde{R}_j$
    Form $Y_j = A\tilde{Q}_j$, compute QR: $Y_j = Q_jR_j$
**end for**
**Output:** $Q = Q_q$

**Key Properties:**

- This algorithm approximates the range of $(AA^*)^qA$, whose singular values are $\sigma_j(A)^{2q+1}$. Thus, the relative spectral decay is accelerated, improving approximation quality.
- The computational cost is approximately $(2q + 1)$ times that of the basic range finder.
- The error is reduced by a factor of $C^{1/(2q+1)}$, where $C$ is the approximation constant in the base case.

- For numerical stability, orthonormalization must be performed after every matrix application, as implemented in Algorithm 4.

### 3.5 Structured Random Matrices

To further improve efficiency, especially for large $n$, one may replace the dense Gaussian matrix $\Omega$ with a structured random matrix, such as a subsampled randomized Fourier transform (SRFT). This choice accelerates the matrix multiplication $Y = A\Omega$.

**Algorithm 5** Fast Randomized Range Finder Using SRFT

1: **Input:** Matrix $A \in \mathbb{R}^{m\times n}$, target dimension $\ell$
2: Construct SRFT test matrix $\Omega = \sqrt{\frac{n}{\ell}} DFR$, where:

- $D$ is a diagonal matrix with random phase entries on the unit circle,
- $F$ is the $n \times n$ discrete Fourier transform (DFT) matrix,
- $R$ samples $\ell$ columns uniformly at random (with or without replacement).

3: Compute $Y = A\Omega$ using fast Fourier transform (FFT)
4: Orthonormalize columns of $Y$ via QR to obtain $Q$
5: **Output:** Orthonormal matrix $Q \in \mathbb{R}^{m\times \ell}$

**Computational Cost.** The use of FFT in the matrix product $A\Omega$ reduces the computational cost to:

$$T_{\text{struct}} \sim mn \log(\ell) + \ell^2 n,$$

compared to the $\mathcal{O}(mn\ell)$ cost of using a Gaussian test matrix.

**Alternative Structured Matrices.** Instead of SRFT, one can use a structured matrix of the form:

$$\Omega = D''\Theta' D'\Theta DFR,$$

where $\Theta$ denotes a sparse matrix composed of random Givens rotations, and $D, D', D''$ are random diagonal matrices with signs or complex phases. This construction maintains desirable embedding properties while offering faster matrixvector multiplication.

## 4 Stage B: Standard Factorizations

After Stage A produces an orthonormal matrix $Q \in \mathbb{R}^{m\times \ell}$ satisfying

$$\|A - QQ^*A\| \le \varepsilon,$$

Stage B uses this basis to compute standard matrix factorizationsmost notably the SVD, eigenvalue decomposition, and Nystrm approximationon smaller projected matrices.

## 4.1 SVD-Based Approximations

### 4.1.1 Direct SVD Method

This method constructs an approximate SVD of $A$ via projection to a lower-dimensional subspace and subsequent post-processing.

**Algorithm 6** Direct SVD Approximation

1: **Input:** Matrix $A \in \mathbb{R}^{m\times n}$, orthonormal basis $Q \in \mathbb{R}^{m\times \ell}$ such that $\|A - QQ^*A\| \le \varepsilon$
2: Form the reduced matrix $B = Q^*A \in \mathbb{R}^{\ell\times n}$
3: Compute the SVD: $B = \tilde{U}\Sigma V^*$
4: Form $U = Q\tilde{U}$
5: **Output:** Approximation $A \approx U\Sigma V^*$ with error bounded by $\|A - U\Sigma V^*\| \le \varepsilon$

**Cost.** Dominated by forming $B = Q^*A$: $\mathcal{O}(mn\ell)$ flops for dense matrices.

### 4.1.2 Row Extraction Method

This variant uses an interpolative decomposition (ID) of $Q$ to avoid operating on the full matrix $A$, trading some accuracy for computational speed.

**Algorithm 7** Row Extraction SVD

1: Compute interpolative decomposition $Q = XQ_{(J,:)}$ for some index set $J$
2: Extract submatrix $A_{(J,:)} \in \mathbb{R}^{k\times n}$ and compute QR: $A_{(J,:)} = R^*W^*$
3: Form $Z = XR^*$
4: Compute SVD: $Z = U\Sigma\tilde{V}^*$
5: Form $V = W\tilde{V}$
6: **Output:** Approximation $A \approx U\Sigma V^*$

**Properties.**

$$\|A - U\Sigma V^*\| \le \left[1 + \sqrt{1 + 4k(n-k)}\right]\varepsilon$$

**Cost.** $\mathcal{O}(k^2(m+n))$ flops

**Advantage.** Fast, especially when $A$ is large; less accurate than direct SVD.

## 4.2 Hermitian Matrices

For Hermitian matrices $A = A^*$, the structure can be preserved during approximation:

$$A \approx Q(Q^*AQ)Q^*,$$

with error bounded by

$$\|A - Q(Q^*AQ)Q^*\| \le 2\varepsilon.$$

#### 4.2.1 Direct Eigenvalue Decomposition

**Algorithm 8** Direct Hermitian Eigendecomposition

1: Form $B = Q^*AQ$
2: Compute eigendecomposition: $B = V\Lambda V^*$
3: Form $U = QV$
4: **Output:** Approximation $A \approx U\Lambda U^*$

**Properties.**

$$\|A - U\Lambda U^*\| \leq 2\varepsilon$$

**Cost.** $\mathcal{O}(k^2 n)$ flops for forming $B$ and computing its eigenvalue decomposition.

### 4.3 Positive Semidefinite Matrices: Nystrm Method

If $A \in \mathbb{R}^{n\times n}$ is symmetric and positive semidefinite (PSD), then a Nystrm-type approximation is possible:

$$A \approx (AQ)(Q^*AQ)^{-1}(AQ)^* = FF^*, \quad F = (AQ)(Q^*AQ)^{-1/2}.$$

#### 4.3.1 Nystrm Eigenvalue Decomposition

**Algorithm 9** Nystrm Eigendecomposition

1: Form $B_1 = AQ$, $B_2 = Q^*B_1 = Q^*AQ$
2: Compute Cholesky factorization: $B_2 = C^*C$
3: Solve triangular system: $F = B_1C^{-1}$
4: Compute SVD: $F = U\Sigma V^*$
5: Set $\Lambda = \Sigma^2$
6: **Output:** Approximation $A \approx U\Lambda U^*$

**Advantage.** The error is never worse than that of the projection:

$$\|A - FF^*\| \leq \|A - QQ^*A\| \leq \varepsilon.$$

In practice, the Nystrm approximation often provides significantly better accuracy.

### 4.4 Single-Pass Algorithms

When matrix $A$ is accessed as a data stream or is too large to store in memory, single-pass algorithms offer a way to approximate spectral factorizations without revisiting $A$.

#### 4.4.1 Single-Pass Hermitian Eigendecomposition

**Algorithm 10** Single-Pass Eigendecomposition for Hermitian $A$

1: Given test matrix $\Omega$, form sample $Y = A\Omega$
2: Compute orthonormal basis $Q$ from $Y$
3: Solve least squares problem: $B_{\text{approx}}(Q^*\Omega) \approx Q^*Y$
4: Compute eigendecomposition: $B_{\text{approx}} = V\Lambda V^*$
5: Form $U = QV$
6: **Output:** Approximation $A \approx U\Lambda U^*$

**Numerical Consideration.** The matrix $Q^*\Omega$ may be ill-conditioned. Oversampling (increasing $\ell = k + p$) improves the conditioning and stabilises the least-squares step.

## 5 Theoretical and Computational Analysis

### Error Guarantees for Randomized Approximation

Let $A \in \mathbb{R}^{m\times n}$ and suppose $Q \in \mathbb{R}^{m\times \ell}$ is a matrix with orthonormal columns constructed using a randomized algorithm (e.g., with Gaussian test matrix $\Omega$). Then with high probability, the projection error $\|A - QQ^*A\|$ is controlled in terms of the $(k+1)$-st singular value $\sigma_{k+1}$ of $A$.

**Theorem 1 (Expected Error Bound)** *If $\Omega \in \mathbb{R}^{n\times(k+p)}$ is a Gaussian matrix, then*

$$\mathbb{E}\|A - QQ^*A\| \le \left(1 + \frac{4\sqrt{k+p}}{p-1} \cdot \sqrt{\min\{m,n\}}\right)\sigma_{k+1}.$$

**Corollary 1 (High-Probability Bound)** *With probability at least $1 - 6\cdot p^{-p}$,*

$$\|A - QQ^*A\| \le \left(1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}\right)\sigma_{k+1}.$$

These bounds justify the empirical recommendation that oversampling by $p = 5$ or 10 is generally sufficient for robust performance.

**Theorem 2 (Power Iteration Error Bound)** *Let $Q$ be computed using $q$ power iterations. Then,*

$$\mathbb{E}\|A - U\Sigma V^*\| \le \left(1 + 4\sqrt{\frac{2\min\{m,n\}}{k-1}}\right)^{1/(2q+1)}\sigma_{k+1},$$

*indicating that power iteration improves approximation quality exponentially in $q$.*

**Lemma 1 (Error Estimation via Gaussian Probes)** *Let $B$ be a real $m\times n$ matrix and $\{\omega^{(i)}\}_{i=1}^r$ be independent standard Gaussian vectors. Then for any $\alpha > 1$,*

$$\|B\| \le \alpha\sqrt{\frac{2}{\pi}}\max_{1\le i\le r}\|B\omega^{(i)}\|$$

*except with probability at most $\alpha^{-r}$.*

**Interpolative Decomposition.** When $Q = A_{(:,J)}X$ is computed via the GuEisenstat algorithm with parameter $f = 2$, the error satisfies

$$\|A - A_{(:,J)}X\| \leq \left(1 + \sqrt{1 + 4k(n-k)}\right) \cdot \varepsilon.$$

### Computational Complexity Comparison

| Method | Flops | Notes |
|---|---|---|
| Full SVD | $\mathcal{O}(mn \cdot \min\{m, n\})$ | Deterministic, accurate |
| Partial SVD | $\mathcal{O}(mnk)$ | Iterative, Lanczos or Krylov |
| Krylov-based | $kT_{\text{mult}} + k^2(m+n)$ | Efficient for sparse $A$ |
| Basic randomized | $\ell T_{\text{mult}} + \ell^2 m$ | $\ell = k + p$ |
| Structured randomized | $mn \log(\ell) + \ell^2 n$ | FFT-based methods |

Table 1: Complexity of standard and randomized low-rank approximations

### Representative Use-Case Scenarios

- **Dense matrices in core memory:**
  - **Randomized:** $T_{\text{rand}} \sim mn \log(k) + k^2(m+n)$ using structured test matrices and row extraction.
  - **Classical:** $T_{\text{RRQR}} \sim kmn$
  - **Error:** $\|A - U\Sigma V^*\| \lesssim n \cdot \sigma_{k+1}$
- **Fast matrix-vector products:**
  - **Basic:** $T = 2(k+p)T_{\text{mult}} + \mathcal{O}(k^2(m+n))$
  - **With power iteration:** $T = (2q+2)(k+p)T_{\text{mult}} + \mathcal{O}(k^2(m+n))$
  - **Error decay:** $(kn)^{1/(2q+1)} \cdot \sigma_{k+1}$
  - **Benefits:** Inherent stability, strong probabilistic guarantees, natural for parallelization
- **Out-of-core or streaming data:**
  - **Classical:** Requires $\mathcal{O}(k)$ full passes over data
  - **Randomized:** Achieves high accuracy in 12 passes using single-pass or power-iteration-based methods

## 6 Implementation Guidelines

The practical success of randomized low-rank approximation algorithms depends critically on careful parameter selection, computational efficiency, and numerical stability. This section outlines key implementation heuristics based on theoretical insights and empirical evidence.

### Parameter Selection

- **Oversampling parameter $p$:** For standard Gaussian test matrices, setting $p \in [5, 10]$ is typically sufficient to ensure accuracy and stability. For structured random matrices (e.g., SRFT), a more conservative choice of $p \in [10, 20]$ is recommended to offset the weaker concentration properties.
- **Power iteration steps $q$:** When the singular values of $A$ decay slowly, a small number of power iterations ($q = 1$ or 2) significantly improves accuracy by accelerating the decay of the singular spectrum of the sampled matrix.
- **Error estimation samples $r$:** In adaptive algorithms, $r = 10$ independent Gaussian vectors are typically sufficient to obtain accurate probabilistic estimates of the spectral norm using Monte Carlo methods.
- **Target dimension $\ell$:** For structured test matrices, set $\ell = k + p$ with $p \in [10, 20]$. This compensates for reduced randomness while still maintaining computational efficiency.

### Computational Efficiency Strategies

- **Matrix operations:** Use high-level BLAS3 routines and blocked matrix operations to exploit cache locality and hardware acceleration for dense matrix computations.
- **Parallelization:** Distribute matrix-vector and matrix-matrix products across multiple cores or compute nodes. Randomized methods are inherently parallel-friendly due to the independence of sample vectors.
- **FFT usage:** In structured random methods (e.g., SRFT), using a full FFT (rather than a subsampled one) can yield superior speed, particularly on hardware-optimised FFT libraries.
- **Progressive rank estimation:** In scenarios with unknown effective rank, progressively double the rank estimate $\ell$ (e.g., $\ell = 2^j$) until a stopping criterion is met. This retains asymptotic complexity while adapting to unknown structure.

### Numerical Stability Considerations

- **Orthogonalization:** Employ numerically stable procedures for computing orthonormal bases, such as Householder QR or modified GramSchmidt with reorthogonalization. Avoid nave single-pass GramSchmidt.
- **Power iteration care:** When using subspace iteration or power methods, orthogonalize the basis between successive applications of $A$ and $A^*$. This mitigates numerical instability and loss of orthogonality.
- **Adaptive schemes:** In adaptive range-finding algorithms, reproject new vectors against the current basis to maintain orthogonality and prevent drift due to accumulated numerical errors.
- **Finite-precision safety:** Monitor vector norms during iteration. Discard or regularize vectors whose norms fall below machine precision to avoid amplification of rounding errors.

# 7 Applications and Performance

Randomized algorithms for low-rank approximation demonstrate strong empirical performance across a wide range of computational settings. Their efficiency, flexibility, and robustness make them well-suited for modern large-scale data problems.

### Optimal Application Scenarios

Randomized methods are particularly advantageous in the following settings:

- **Structured or sparse matrices:** Where matrix-vector products can be computed rapidly.
- **Rapid spectral decay:** When the singular values of $A$ decay quickly, randomized methods yield near-optimal approximations.
- **Low intrinsic rank:** When the desired rank $k$ satisfies $k \ll \min(m, n)$, significant dimensionality reduction is achieved.
- **Streaming or out-of-core data:** Algorithms requiring only a small number of passes over the data are critical in data-limited settings.
- **Parallel or distributed environments:** Random sampling and matrix multiplication are naturally parallelizable.

### Key Benefits

Randomized approaches offer several practical advantages over classical deterministic methods:

- **Streaming compatibility:** Algorithms can operate in a single pass, ideal for large-scale and online applications.
- **Parallelism:** Core operations such as sampling and matrix multiplication can be distributed across compute units.
- **Low memory footprint:** The algorithms do not require storing the entire input matrix in memory.
- **Algorithmic robustness:** Performance is reliable across a wide variety of matrix types and numerical conditions.

### Empirical Performance Highlights

Numerical experiments reported in the literature validate the efficiency and accuracy of randomized low-rank approximations:

- **Fast spectral decay:** For matrices arising in integral equations, randomized methods achieve errors comparable to truncated SVD at a fraction of the cost.
- **Slow spectral decay:** For data-mining and information retrieval matrices, power iteration with $q = 2$ typically restores near-optimal accuracy.
- **Structured transforms:** Algorithms using subsampled randomized Fourier transforms (SRFTs) yield $2\times$ to $5\times$ speedups over classical techniques for moderate problem sizes.

### Comparison with Classical Algorithms

Relative to traditional low-rank decomposition techniques (e.g., truncated SVD, RRQR, Krylov methods), randomized methods provide the following advantages:

- **Accuracy:** Often comparable to, or better than, rank-revealing QR factorizations, especially with modest oversampling.
- **Speed:** Superior scalability for large dense matrices; particularly effective in parallel computing environments.
- **Stability:** Less susceptible to roundoff errors and breakdowns than iterative Krylov subspace methods.
- **Adaptability:** Algorithms can be easily tailored to specific matrix structures or hardware architectures.

## 8 Enhanced Randomized SVD Algorithm

To improve accuracy in cases where the singular values of $A$ decay slowly, power iteration may be incorporated into the randomized SVD framework. The following algorithm applies $q$ steps of subspace iteration to amplify spectral decay prior to projection.

**Algorithm 11** Randomized SVD with Power Iteration

1: **Input:** Matrix $A \in \mathbb{R}^{m \times n}$, target rank $k$, power iteration steps $q$
2: Draw a Gaussian random matrix $\Omega \in \mathbb{R}^{n \times 2k}$
3: Form $Y = (AA^*)^q A\Omega$ (apply $q$ steps of $A$ and $A^*$)
4: Compute thin QR decomposition $Y = QR$
5: Form the projection $B = Q^*A$
6: Compute SVD of small matrix: $B = \tilde{U}\Sigma V^*$
7: Form $U = Q\tilde{U}$
8: **Output:** Approximation $A \approx U\Sigma V^*$

This procedure achieves accuracy close to the optimal truncated SVD while significantly reducing computational costs for large matrices. The power iteration amplifies the decay of smaller singular values by transforming $A \mapsto (AA^*)^q A$, without altering the dominant singular vectors.

## 9 Conclusion

Randomized algorithms offer a scalable and flexible framework for low-rank matrix approximation. The central insight is that *a small amount of oversampling*typically $p = 5$ to 10with random test matrices suffices to capture the dominant range of a matrix with high probability. This leads to substantial reductions in computational complexity and storage requirements compared to classical methods.

- **Classical SVD:** Computational cost of $O(\min\{mn^2, m^2n\})$
- **Randomized SVD:** Computational cost of $O(mn \log k + k^2(m+n))$

When $k \ll \min\{m, n\}$, the randomized approach provides dramatic acceleration with minimal loss in accuracy. Furthermore, it is particularly well-suited for streaming and out-of-core settings, requiring only storage of the sketch matrix $Q \in \mathbb{R}^{m \times k}$ and small projected matrices. Randomized methods thus offer an elegant trade-off between computational efficiency and approximation quality, with broad applicability in modern data analysis, machine learning, and scientific computing.

# Part II: Experimental Results and Observations

# 10 Toy Experiments

This section details the implementation and experimental validation of randomized Singular Value Decomposition (SVD) algorithms, drawing inspiration from the theoretical framework presented in Halko, Martinsson, and Tropp (2010). The experiments cover both synthetic matrix compression and neural network weight compression, analyzing the accuracy and performance of various randomized SVD methods against a deterministic baseline (implemented in "Experiment_1.ipynb").

## 10.1 Implementation of Randomized SVD Algorithms

The core of the experiment involves implementing several randomized SVD algorithms within the `RandomizedSVD` class. All implementations adhere to the two-stage algorithmic framework.

### 10.1.1 Core Randomized SVD Components

The `RandomizedSVD` class encapsulates the different randomized SVD variants, maintaining a consistent two-stage process:

- **Stage A (Randomized Range Finding)**: This stage focuses on constructing an approximate orthonormal basis $Q$ for the range of the input matrix $A$. Randomness is intentionally introduced here to efficiently capture the essential information within $A$.
- **Stage B (Deterministic Factorization)**: Once the approximate basis $Q$ is obtained, a smaller matrix $B$ is formed by computing $Q^T A$. A standard deterministic SVD (utilizing `scipy.linalg.svd`) is then applied to $B$ to compute its singular values $S$ and right singular vectors $V_t$. The left singular vectors $U$ of the original matrix are subsequently reconstructed as $Q$ multiplied by the left singular vectors of $B$ ($U_B$).

### 10.1.2 Algorithm 11: Randomized SVD with Power Iteration

The `power_iteration_svd` method directly implements Algorithm 11: Randomized SVD with Power Iteration. This algorithm is specifically designed to enhance approximation accuracy, particularly for matrices characterized by slowly decaying singular values, through the application of power iterations.

- **Generate Gaussian Matrix** $\Omega$: An $n \times l$ Gaussian random matrix $\Omega$ is generated, where $l = k+p$ (with $k$ as the target rank and $p$ as the oversampling parameter). The paper specifies $n \times 2k$ for $\Omega$, however, the implementation uses $n \times (k+p)$, a common practice in randomized SVD to incorporate oversampling.
- **Form** $Y = (AA^*)^q A\Omega$: The computation begins with $Y = A\Omega$. Subsequently, $q$ power iterations are performed. In each iteration, $Y$ is updated as $Y = A(A^T Y)$. This effectively computes $A(A^T A)^q \Omega$, which is equivalent to $(AA^*)^q A\Omega$ as described in the theoretical framework. This iterative process implicitly amplifies the dominant singular values, making the most significant subspace more pronounced.
- **Construct orthonormal** $Q$ **spanning range of** $Y$: A QR factorization of $Y$ is computed, yielding an orthonormal matrix $Q$ whose columns span the range of $Y$.
- **Form** $B = Q^* A$: The smaller matrix $B$ is formed by projecting $A$ onto the subspace spanned by $Q$.

- **Compute SVD:** $B = \tilde{U}\Sigma V^*$: A full SVD of $B$ is performed, where `full_matrices=False` ensures that only the singular vectors corresponding to the non-zero singular values are returned, optimizing for low-rank approximations.
- **Set** $U = Q\tilde{U}$: The final left singular vectors $U$ of the original matrix $A$ are constructed by multiplying $Q$ with $\tilde{U}$.
- **Output**: The top $k$ singular values and their corresponding singular vectors are returned.

### 10.1.3 Other Randomized SVD Algorithms Implemented

- **Basic Randomized SVD (Algorithm 2 from paper)**: This fundamental randomized SVD method involves generating a Gaussian test matrix $\Omega$, forming $Y = A\Omega$, computing an orthonormal basis $Q$ for $Y$ (via QR factorization), and then performing SVD on $B = Q^T A$. The oversampling parameter $p$ dictates the dimensions of $\Omega$ as $k + p$.
- **SRFT SVD (Algorithm 5 from paper)**: This method aims to leverage the Subsampled Random Fourier Transform (SRFT) for efficient range finding. While the theoretical description in the provided document outlines the construction of a structured $\Omega$ using diagonal ($D$), Discrete Fourier Transform ($F$), and random coordinate sampling ($R$) matrices for accelerated matrix-vector products, the implementation used for this experiment simplifies this by generating a standard Gaussian $\Omega$ and then proceeding with the standard randomized SVD steps. This simplification means the implementation does not fully exploit the "fast" properties of SRFT.
- **Error Estimator**: The error estimation technique implemented approximates the approximation error $\|A - QQ^*A\|_F$ using the Gaussian probes. Specifically, a set of random Gaussian vectors are generated, and for each, the norm of the residual (the difference between the original vector's projection and its approximation by $Q$) is computed. The maximum of these residual norms, scaled by $\sqrt{\frac{2}{\pi}}$, provides a robust estimate of the approximation error. This approach allows for practical assessment of the low-rank approximation quality without requiring full matrix computations.

## 10.2 Synthetic Matrix Generation

The experiment utilizes two distinct types of synthetic matrices to evaluate the algorithms under varying singular value decay characteristics:

- **Fast Decay Matrix**: This matrix type is designed such that its singular values decay exponentially. Such matrices are inherently well-suited for low-rank approximation methods, as a limited number of singular values capture the majority of the matrix's energy.
- **Slow Decay Matrix**: In contrast, this matrix is characterized by polynomially decaying singular values. These matrices present a greater challenge for basic low-rank approximation, as a larger set of singular values maintain significant contribution to the matrix's structure. This scenario is where the benefits of power iteration are particularly anticipated.

Both matrix types are constructed as $A = U\Sigma V^T$, where $U$ and $V$ are random orthonormal matrices, and $\Sigma$ is a diagonal matrix containing singular values that exhibit either exponential or polynomial decay.

## 10.3 Experimental Validation: Matrix Compression

The `run_matrix_compression_experiments` function systematically evaluates the performance of the implemented randomized SVD algorithms on the synthetic matrices. The results are provided in the figure 1.

### 10.3.1 Experiment Setup

- **Matrices**: A $500 \times 400$ fast-decay matrix and a $500 \times 400$ slow-decay matrix were generated. Their respective condition numbers, $1.41 \times 10^{18}$ and $1.21 \times 10^{2}$, reflect their spectral properties. The high condition number of the fast decay matrix indicates its strong compressibility.[1]
- **Target Ranks ($k$)**: The algorithms were tested across a range of target ranks from 5 to 100, incrementing by 5.
- **Oversampling ($p$)**: Basic randomized SVD was evaluated with oversampling parameters of 5, 10, and 20. For power iteration and SRFT, a default $p = 10$ was consistently applied.
- **Power Iterations ($q$)**: The power iteration SVD employed $q$ values of 1 and 2.
- **Baseline**: A standard deterministic SVD (via `scipy.linalg.svd`) served as the performance baseline, with the Frobenius norm of the approximation error ($||A - A_{\approx}||_F$) being the primary metric.

### 10.3.2 Observations and Results

- **Singular Value Decay**: The fast decay matrix exhibited a rapid decline in singular values, confirming its suitability for low-rank approximation. Conversely, the slow decay matrix showed a more gradual, linear-like decay, indicating that a larger number of singular values contribute significantly to its structure.
- **Approximation Errors for Fast Decay Matrix**: For the fast decay matrix, all randomized SVD methods (Basic, Power, SRFT) achieved approximation errors closely matching the deterministic SVD baseline, especially at higher ranks. The impact of varying the oversampling parameter $p$ for basic randomized SVD was minimal, consistent with theoretical expectations for rapidly decaying spectra. Power iteration did not provide a substantial advantage in this scenario, as the dominant subspace was readily captured.
- **Approximation Errors for Slow Decay Matrix**: The benefits of power iteration were distinctly observed for the slow decay matrix. Both $q = 1$ and $q = 2$ power iterations significantly reduced the approximation error compared to the basic randomized SVD, closely approaching the deterministic baseline. This corroborates the theoretical assertion that power iteration effectively amplifies dominant singular values for matrices with slowly decaying spectra.

[1] Condition number of a matrix $A$ is

$$\kappa(A) \coloneqq \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}.$$

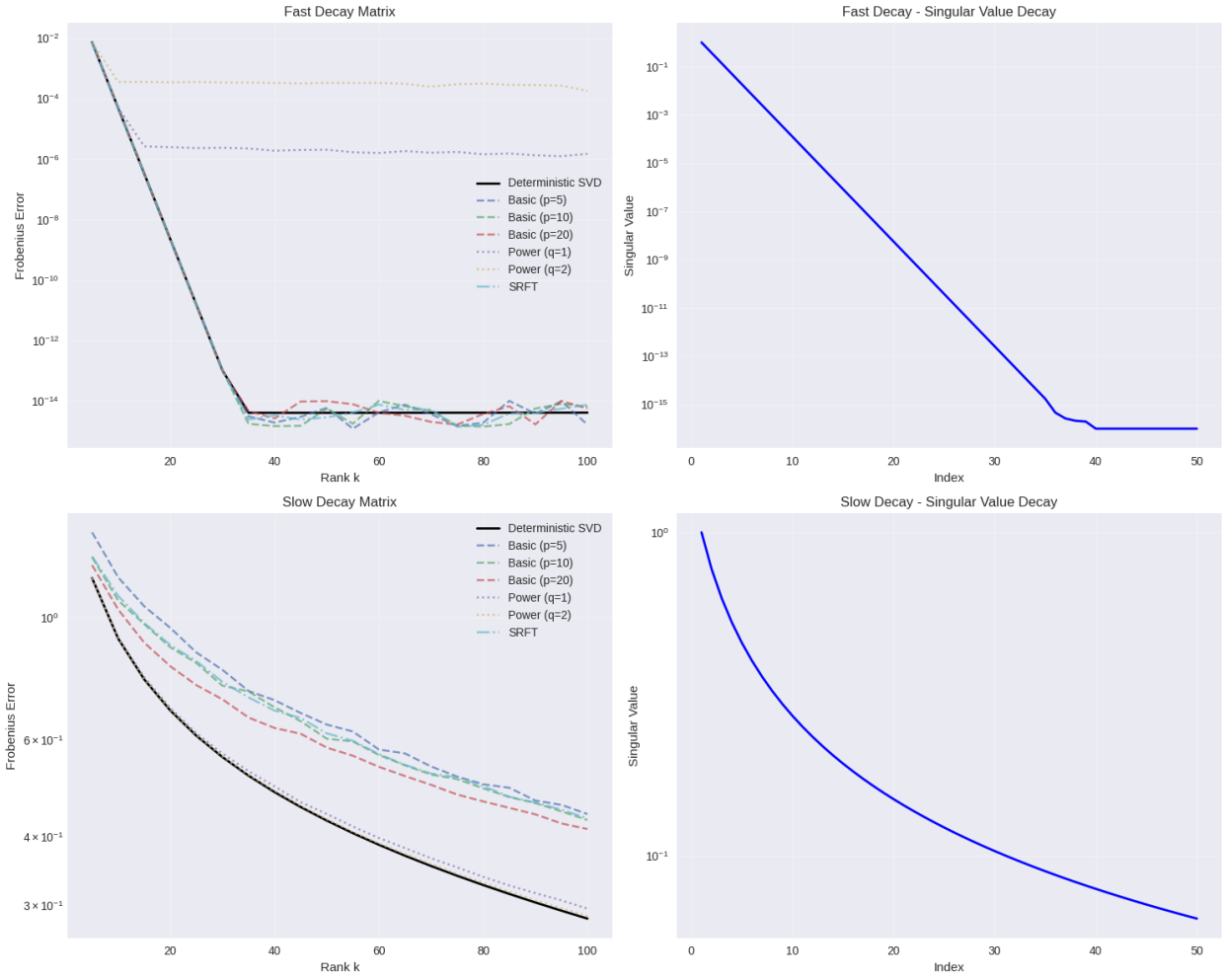



Figure 1: Matrix Compression Results: Frobenius Error vs. Rank and Singular Value Decay.

### 10.4 Multi layer Perceptorn Compression

The `run_neural_network_experiments` function investigates the application of randomized SVD to compress the weight matrices of a Multi-Layer Perceptron (MLP) trained on the MNIST dataset. The results are shown in the figure 2.

#### 10.4.1 Architecture

- **Input Layer:** 784 neurons (flattened $28 \times 28$ MNIST image).
- **Hidden Layer 1:** 512 neurons with **ReLU** activation.
- **Hidden Layer 2:** 256 neurons with **ReLU** activation.
- **Output Layer:** 10 neurons for digit classification. `CrossEntropyLoss` applies **Softmax**.
- **Fully Connected:** Each layer is densely connected to the next.

**Input:** 784 neurons
↓
**Fully Connected Layer 1:** $784 \rightarrow 512$
**Activation:** ReLU
↓
**Fully Connected Layer 2:** $512 \rightarrow 256$
**Activation:** ReLU
↓
**Fully Connected Layer 3 (Output):** $256 \rightarrow 10$
**Output:** 10 classes (digits 0–9)

#### 10.4.2 Experiment Setup

- **Model and Training**: A `CompressibleMLP` with two hidden layers (512 and 256 neurons) was trained on the MNIST dataset for 3 epochs. The original, uncompressed model served as a baseline, achieving a test accuracy of 97.22%.
- **Compression Parameters**: Compression ranks ($k$) of 10, 20, 50, 100, and 150 were explored. For all randomized methods, an oversampling parameter $p = 10$ was used. Power iteration SVD specifically used $q = 1$ iteration.
- **Methods**: 'basic', 'power', 'srft', and 'deterministic' SVD were employed to compress the MLP's weight matrices.
- **Evaluation**: For each compression method and rank, a copy of the trained model was utilized. Its weight matrices were compressed, and the model's accuracy on the MNIST test set, along with the average normalized Frobenius norm approximation error of the weights, were recorded ("average normalized Frobenius norm approximation error of the weights" refers to the mean of individual normalized Frobenius errors computed for each of the model's weight matrices).

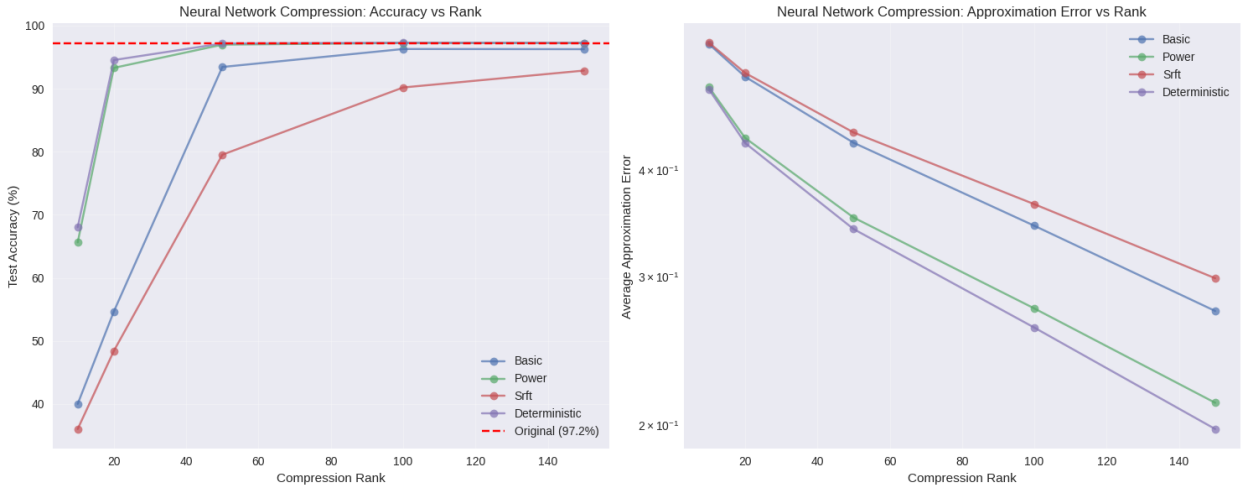



Figure 2: Neural Network Compression: Accuracy vs. Rank and Approximation Error vs. Rank.

**Remark 1** *There are no subsequent training steps applied to the compressed weights after SVD is performed.*

### 10.4.3 Observations and Results

- **Accuracy vs. Compression Rank**: Accuracy generally improved with increasing compression rank for all methods, approaching the original model's performance. The deterministic SVD consistently yielded the highest accuracy, as expected. Power iteration SVD performed remarkably close to the deterministic baseline, particularly at higher ranks (e.g., 97.24% accuracy at rank 100). This underscores the efficacy of power iteration for preserving neural network performance during compression. Basic randomized SVD and SRFT SVD generally exhibited lower accuracy compared to power iteration and deterministic SVD.

- **Approximation Error vs. Compression Rank**: The average normalized Frobenius norm approximation error decreased as the compression rank increased for all methods. Deterministic SVD achieved the lowest approximation errors. Power iteration SVD produced approximation errors very close to deterministic SVD, reinforcing its ability to find high-quality low-rank approximations (again "average normalized Frobenius norm approximation error" refers to the mean of individual normalized Frobenius errors computed for each of the model's weight matrices).

- **Detailed Analysis (Best Rank Performance)**:
  - **Basic**: Best rank = 100, Accuracy = 96.28%, Error = 0.3442.
  - **Power**: Best rank = 100, Accuracy = 97.24%, Error = 0.2749. This method's performance was notably close to the original model (97.22%) and the deterministic SVD, even with substantial compression.
  - **SRFT**: Best rank = 150, Accuracy = 92.86%, Error = 0.2984. While SRFT offers theoretical computational benefits for very large matrices, its accuracy in this specific neural network compression scenario did not surpass that of power iteration for the same rank.

– **Deterministic**: Best rank = 100, Accuracy = 97.27%, Error = 0.2609. This served as the optimal benchmark for SVD-based low-rank approximation.

## 10.5 Multi layer Perceptron Compression with Retraining

This section details the neural network compression experiments, specifically investigating the impact of subsequent retraining (fine-tuning) on the accuracy of models with compressed weights. The results are provided in the figure 3.

### 10.5.1 Experiment Setup

- **Model and Training**: A `CompressibleMLP` with the same architecture was trained on the MNIST dataset for 5 epochs. The original, uncompressed model achieved a test accuracy of 97.81%.
- **Compression Parameters**: Compression ranks ($k$) of 10, 20, 50, 100, and 150 were tested. For all randomized methods, an oversampling parameter $p = 10$ was used. Power iteration SVD specifically used $q = 1$ iteration.
- **Methods**: `basic`, `power`, `srft`, and `deterministic` SVD were employed to compress the MLP's weight matrices.
- **Retraining**: After compression, models were subjected to retraining for 0 (no retraining), 1, 3, or 5 epochs. A smaller learning rate (0.0005) was used during retraining for fine-tuning purposes.
- **Evaluation**: For each combination of compression method, rank, and retraining epochs, a fresh copy of the model (with applied compressed weights) was used. Its accuracy on the MNIST test set and the percentage of accuracy recovery (relative to the immediately compressed model's accuracy) were recorded.

### 10.5.2 Observations and Results

- **Compression Without Retraining (Figure 3, Top-Left)**:
  - As observed in previous experiments, accuracy generally improves with increasing compression rank for all methods.
  - Deterministic SVD consistently yields the highest accuracy without retraining, serving as the upper bound for SVD-based compression.
  - Power iteration performs significantly better than Basic and SRFT at lower ranks, particularly for ranks 10 and 20. For instance, at rank 20, Power achieved 94.32% accuracy, compared to Basic's 65.97% and SRFT's 74.21%. This demonstrates the effectiveness of power iteration in preserving model performance even without fine-tuning, especially for more aggressive compression ratios.
- **Effect of Retraining (Deterministic SVD) (Figure 3, Top-Right)**:
  - Retraining dramatically improves the accuracy of the compressed models, especially at lower compression ranks.

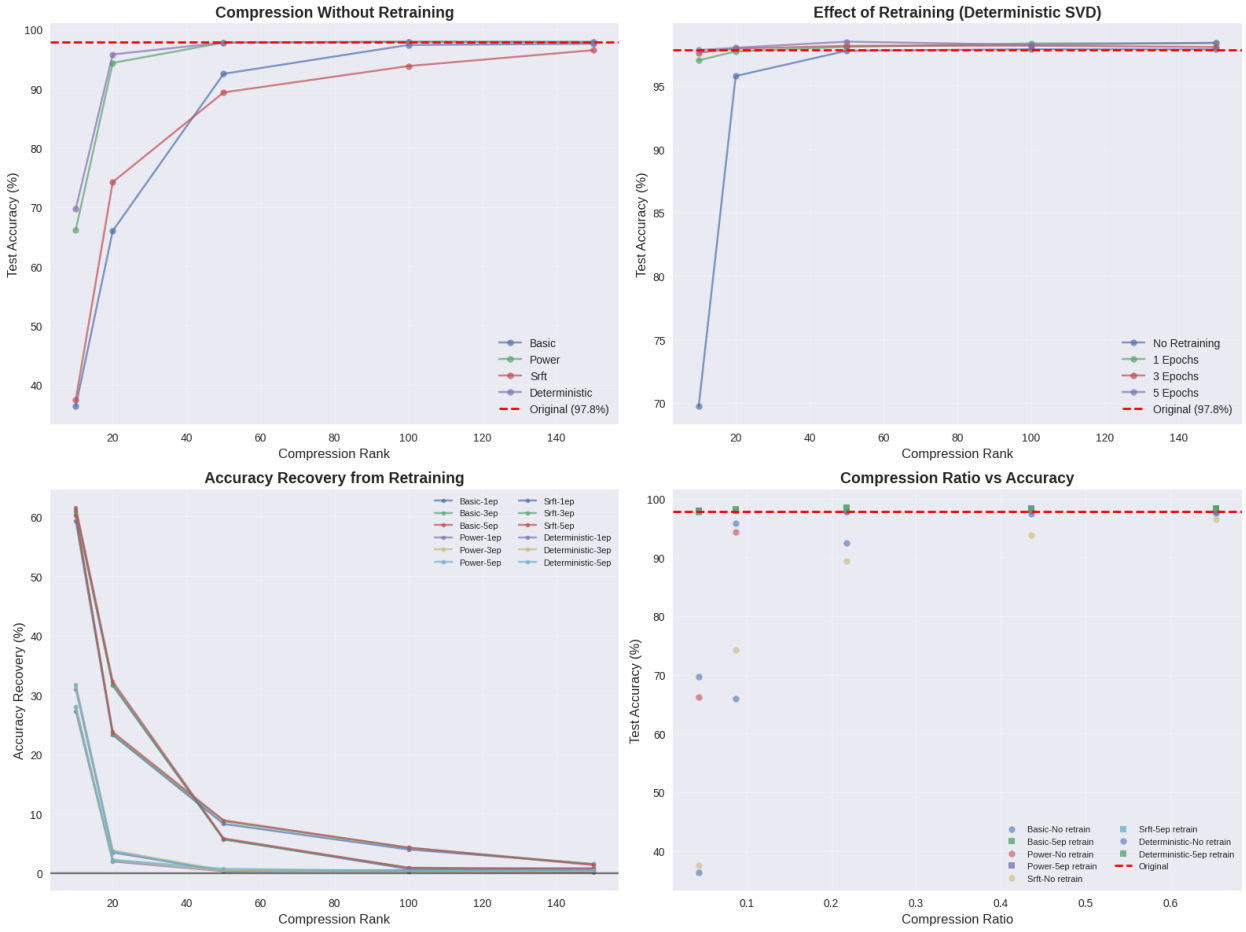



Figure 3: Multi Layer Perceptron Compression: Accuracy with and without Retraining.

  - Even with just 1 epoch of retraining, the accuracy for deterministic SVD at rank 10 jumps from 69.73% to 97.00%. With 3 or 5 epochs, the accuracy for highly compressed models (e.g., rank 10) can almost fully recover to the original model's accuracy (97.81% for original vs. 97.90% for Basic SVD at rank 10 after 5 epochs).
  - This highlights that retraining acts as a crucial fine-tuning step, allowing the compressed model to adapt and recover performance lost during the initial weight approximation.

- **Accuracy Recovery from Retraining (Figure 3, Bottom-Left)**:
  - For Basic and SRFT SVD, retraining leads to very substantial accuracy recovery, particularly at lower ranks where the initial accuracy drop was significant. For example, Basic SVD at rank 10 recovered over 60% accuracy after 1 to 5 epochs of retraining.
  - Power iteration and Deterministic SVD, which start with higher accuracies after compression, show smaller absolute recovery percentages, but this is because there is less accuracy to recover in the first place. Their initial accuracy is already quite high.
  - The recovery tends to diminish as the compression rank increases, as the models are already closer to the original performance without retraining.

- **Compression Ratio vs. Accuracy (Figure 3, Bottom-Right)**:
  - This plot directly illustrates the trade-off between model size reduction (compression ratio) and performance (accuracy).
  - Models with retraining (e.g., `5ep retrain`) maintain much higher accuracy even at very low compression ratios (high compression, small ratio) compared to their counterparts with no retraining.
  - At a compression ratio of approximately 0.05 (corresponding to rank 10), models without retraining show significantly degraded performance (e.g., Basic SVD at $\sim$36% accuracy). However, with 5 epochs of retraining, accuracy for all methods at this ratio (rank 10) is above 97%.

- **Accuracy increases from the base in some cases after retraining**: Retraining after compression is highly effective because it allows the model to "fine-tune" the newly compressed weights. When weights are compressed, some precision is inevitably lost, leading to an immediate drop in accuracy. Retraining acts like a quick adjustment, using the large initial errors to efficiently guide the model's parameters back to a better, more functional state within its new, smaller representation.

- **First epoch recovers most of the accuracy**: The significant accuracy recovery observed in the first retraining epoch occurs because the model is initially far from an optimal state after compression. This large performance gap results in substantial loss values and, consequently, large gradients. These large gradients drive the most impactful weight adjustments in the initial training steps, quickly pulling the model's performance up the steepest part of the accuracy curve. As training continues in subsequent epochs, the model gets closer to an optimum, and the remaining improvements become smaller and more subtle, leading to diminishing returns.

### 10.5.3 Detailed Analysis

**Original model accuracy**: 97.81%
**Original model parameters**: 401,930
The best accuracy was achieved at rank 50 and the detailed results are provided in the table 2.

Table 2: Best Results by Method (Rank 50)

| Method | Epochs | Accuracy (%) | Recovery (%) |
|---|---|---|---|
| BASIC | 0 | 92.48 | +0.00 |
| | 1 | 98.18 | +5.70 |
| | 3 | 98.11 | +5.63 |
| | 5 | 98.32 | +5.84 |
| POWER | 0 | 97.75 | +0.00 |
| | 1 | 98.09 | +0.34 |
| | 3 | 98.20 | +0.45 |
| | 5 | 98.05 | +0.30 |
| SRFT | 0 | 89.34 | +0.00 |
| | 1 | 97.61 | +8.27 |
| | 3 | 98.04 | +8.70 |
| | 5 | 98.22 | +8.88 |
| DETERMINISTIC | 0 | 97.75 | +0.00 |
| | 1 | 98.06 | +0.31 |
| | 3 | 98.14 | +0.39 |
| | 5 | 98.47 | +0.72 |

Observe that at rank 50, the model is compressed by more than 75%.

## 10.6 Conclusion

The experiments successfully validate the principles of randomized SVD for both general matrix compression and neural network weight compression.

- Algorithm 11 (Randomized SVD with Power Iteration) proved highly effective, particularly in scenarios resembling slow singular value decay and for neural network weight compression. It consistently achieved accuracy very close to the deterministic SVD baseline, demonstrating its ability to implicitly amplify dominant components for better approximation.
- The power iteration parameter ($q$) substantially improved the accuracy of randomized SVD, rendering it competitive with deterministic methods for challenging matrices and practical applications like neural network compression.
- The SRFT implementation, while conceptually designed for faster matrix-vector products, did not exhibit a distinct accuracy advantage over basic randomized SVD in these experiments due to its simplified implementation that did not fully leverage FFT-based acceleration.
- The experiments demonstrate that retraining (fine-tuning) is a critical step in preserving and even enhancing the accuracy of neural networks after weight compression. Even a few epochs

of retraining can recover most of the performance lost due to compression, especially at high compression ratios.

- Deterministic SVD, when combined with retraining, achieves accuracy exceeding that of the original uncompressed model at a substantial compression ratio. Power iteration also performs exceptionally well, highlighting the robustness of randomized methods when retraining is incorporated into the compression pipeline.

Overall, the results underscore that randomized SVD methods, especially when augmented with power iteration, offer a viable and often highly competitive alternative to classical deterministic SVD for large-scale low-rank approximation problems, including the practical application of neural network compression. The experiments align with the core concepts from the provided theoretical framework, demonstrating the practical utility and performance characteristics of these randomized SVD algorithms.

A remarkable noteworthy point which can be concluded is the rank selection strategy:

**Rank Selection Strategy: Uniform rank compression across all layers is suboptimal. An adaptive rank selection strategy, which tailors the rank based on each layer's importance, leads to improved performance.**

# 11 Compression of FC Layers in a CNN

This section provides a detailed analysis of Experiment 2, focusing on the application of randomized Singular Value Decomposition (SVD) algorithms for compressing the weights of Fully Connected (FC) layers in a Convolutional Neural Network (CNN). The experiment specifically leverages Algorithm 11: Randomized SVD with Power Iteration, as outlined in Halko, Martinsson, and Tropp (2010), alongside other randomized SVD variants for comparative analysis (implemented in "Experiment_2_fc_layer.ipynb").

## 11.1 Implementation Details

The `algorithm_11_randomized_svd_power_iteration` function directly implements Algorithm 11 from the theoretical framework. This algorithm is designed to enhance approximation accuracy, particularly for matrices with slowly decaying singular values, through the application of power iterations.

The steps implemented in the Python notebook reflect Algorithm 11 as follows:

1. **Generate Gaussian Matrix** $\Omega$: A Gaussian random matrix $\Omega \in \mathbb{R}^{n \times 2k}$ is generated, where $n$ is the number of columns of the input matrix $A$, and $k$ is the target rank. The notebook code uses `np.random.randn(n, 2 * k)` for this purpose.

2. **Form** $Y$ **and Construct Orthonormal Basis** $Q$: The process begins by forming $Y = A\Omega$. Subsequently, $q$ power iterations are performed using:

   `algorithm_4_randomized_subspace_iteration(A, 2 * k, q)`

   This effectively computes $(AA^T)^q A\Omega$, which amplifies the dominant singular values. A QR factorization is then computed to yield an orthonormal matrix $Q$ spanning the range of $Y$.

3. **Form** $B = Q^T A$: A smaller matrix $B$ is formed by projecting $A$ onto the subspace spanned by $Q$. This is implemented as $B = Q^T A$.

4. **Compute SVD of** $B$: A Singular Value Decomposition (SVD) is computed for the smaller matrix $B$:

   $(U_{\text{tilde}}, \Sigma, V^T) =$ `np.linalg.svd(B, full_matrices=False)`

5. **Set** $U = QU_{\textbf{tilde}}$: The final left singular vectors $U$ of the original matrix $A$ are constructed by multiplying $Q$ with $U_{\text{tilde}}$.

6. **Truncate**: The singular values and vectors are truncated to the target rank $k$.

The notebook also includes implementations of:

- `algorithm_4_randomized_subspace_iteration` (Algorithm 4 from the paper), which is utilized by Algorithm 11 for the range finding with power iterations, and
- `algorithm_2_basic_randomized_range_finder` (Algorithm 2 from the paper), used for the 'basic' randomised SVD comparison.

## 11.2 Experiment Setup and Implementation Details

The experiment uses a simple Convolutional Neural Network (CNN) architecture (`SimpleCNN`) with fully connected (FC) layers (`fc1` and `fc2`) and is trained on the CIFAR-10 dataset.

### 11.2.1 Model Architecture

The `SimpleCNN` model consists of:

- **Three convolutional layers** (`conv1`, `conv2`, `conv3`) with ReLU activation and max pooling.
- **Three fully connected layers**:
  - `fc1`: Input features $128 \times 4 \times 4$, output features 512
  - `fc2`: Input features 512, output features 256
  - `fc3`: Input features 256, output features 10 (for 10 CIFAR-10 classes)
- **Dropout layers** are applied after `fc1` and `fc2`.

### 11.2.2 Data Loading and Original Model Training

- **Dataset:** CIFAR-10, loaded with standard training and testing transforms including normalization.
- **Training:** The original `SimpleCNN` model is trained for 5 epochs using `nn.CrossEntropyLoss` and `torch.optim.Adam`.
- **Baseline Performance:** The original model achieved a test accuracy of **71.03%**. The original model has `1,276,234` trainable parameters.

### 11.2.3 FC Layer Compression

The `compress_fc_layer` function performs the compression. This function:

1. Extracts the weight matrix $W$ and bias from the specified FC layer (`layer_name`).
2. Determines the effective rank for compression (truncated to min(W.shape) − 1).
3. Applies the chosen SVD method (`'algorithm_11'`, `'algorithm_4'`, or `'basic'`) to $W$ to obtain $U$, $S$, and $V^t$.
4. Replaces the original FC layer in the model with a `CompressedFC` module, which reconstructs the linear operation using the compressed $U$, $S$, and $V^t$ components.
5. Calculates and prints the layer-wise compression ratio and the time taken for compression.

### 11.2.4 Comprehensive Experiment Parameters

The `comprehensive_fc_compression_experiment` function evaluates different compression methods and ranks, including retraining.

- **Ranks:** `[8, 16, 32, 64, 128]`
- **Retraining Epochs:** `[0, 1, 3, 5]`
- **Methods:** `algorithm_11`, `algorithm_4`, `basic`
- **Layer-Specific Compression:** The `fc1` layer is compressed to `rank` and `fc2` to `rank`·0.5. This indicates a strategy where the second FC layer is compressed more aggressively.
- **Retraining Learning Rate:** A smaller learning rate of `0.0001` is used for retraining.

## 11.3 Observations

The experiment's output provides detailed insights into the performance of each randomized SVD algorithm across various ranks and retraining epochs. The original model's accuracy is 71.03% with 1,276,234 parameters.

### 11.3.1 Analysis of FC Layer Spectra

The singular value spectrum of the FC layers was analyzed (Figure 4: FC1 Singular Values, FC1 Normalized Singular Values, FC2 Singular Values, FC2 Normalized Singular Values).

- **FC1 layer:** Shape (512, 2048), Rank: 512, Condition number: $2.09\text{e}+01$, Average log decay rate: $-0.0060$.
- **FC2 layer:** Shape (256, 512), Rank: 256, Condition number: $1.50\text{e}+01$, Average log decay rate: $-0.0106$.

These values indicate how quickly the singular values decay. A higher condition number and a less negative decay rate suggest a slower singular value decay, which is where power iteration methods are particularly beneficial.

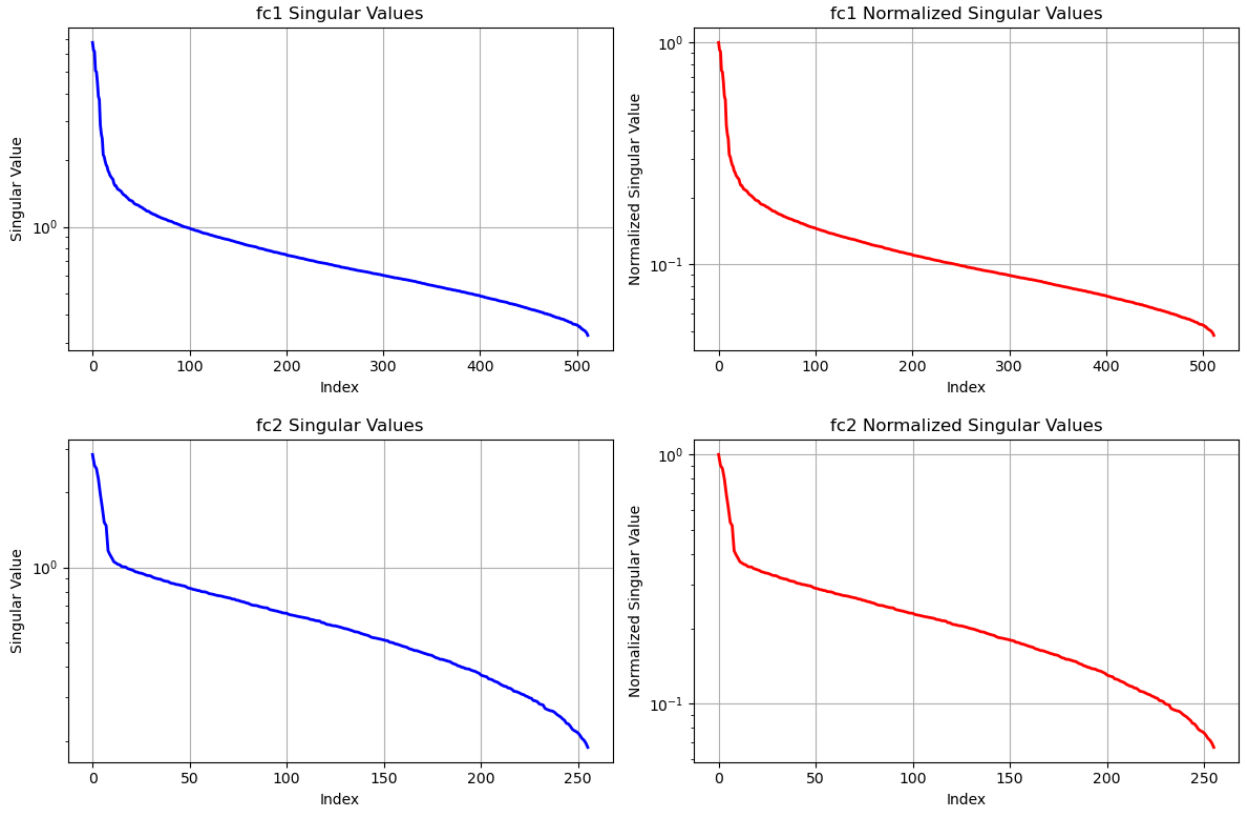



Figure 4: Singular value decay for the FC layers.

### 11.3.2 Compression Results by Algorithm and Rank

**Algorithm 11: Randomized SVD with Power Iteration**

- **Accuracy without Retraining:** This method consistently performs well, with accuracy ranging from 64.94% (Rank 8) to 70.85% (Rank 128) immediately after compression.
- **Retraining Impact:** Retraining significantly improves accuracy. For instance, at Rank 8, accuracy jumps from 64.94% to 71.56% after 5 epochs of retraining. At Rank 128, it increases from 70.85% to 75.68%.
- **Compression Ratio:** Achieves high layer compression ratios (e.g., 49.96x for `fc1` at Rank 8) and substantial overall model compression (e.g., 10.62x at Rank 8, down to 2.69x at Rank 128).
- **Parameter Reduction:** For Rank 8, model parameters reduce from $1,276,234$ to $120,150$. For Rank 128, they reduce to $473,610$.

**Algorithm 4: Randomized Subspace Iteration**

- Accuracy without Retraining: Similar to Algorithm 11, with accuracies from 64.73% (Rank 8) to 70.81% (Rank 128).
- Retraining Impact: Retraining shows comparable improvements, e.g., at Rank 8, accuracy improves from 64.73% to 71.80% after 5 epochs. At Rank 128, it rises from 70.81% to 75.54%.
- Compression Ratio: Compression ratios are identical to Algorithm 11, as the underlying strategy for determining compressed parameter counts is the same.

**Basic Randomized SVD (Algorithm 2)**

- Accuracy without Retraining: This method generally has lower baseline accuracy compared to Algorithm 11 and Algorithm 4, especially at lower ranks (e.g., 45.70% at Rank 8).
- Retraining Impact: Retraining provides the most dramatic improvements for this method due to its lower starting point. At Rank 8, accuracy leaps from 45.70% to 71.33% after 5 epochs. Even at Rank 128, accuracy improves from 70.43% to 75.08%.
- Compression Ratio: Compression ratios are identical to the other two methods.

These are summarised in the figures 5, 6 and 7.

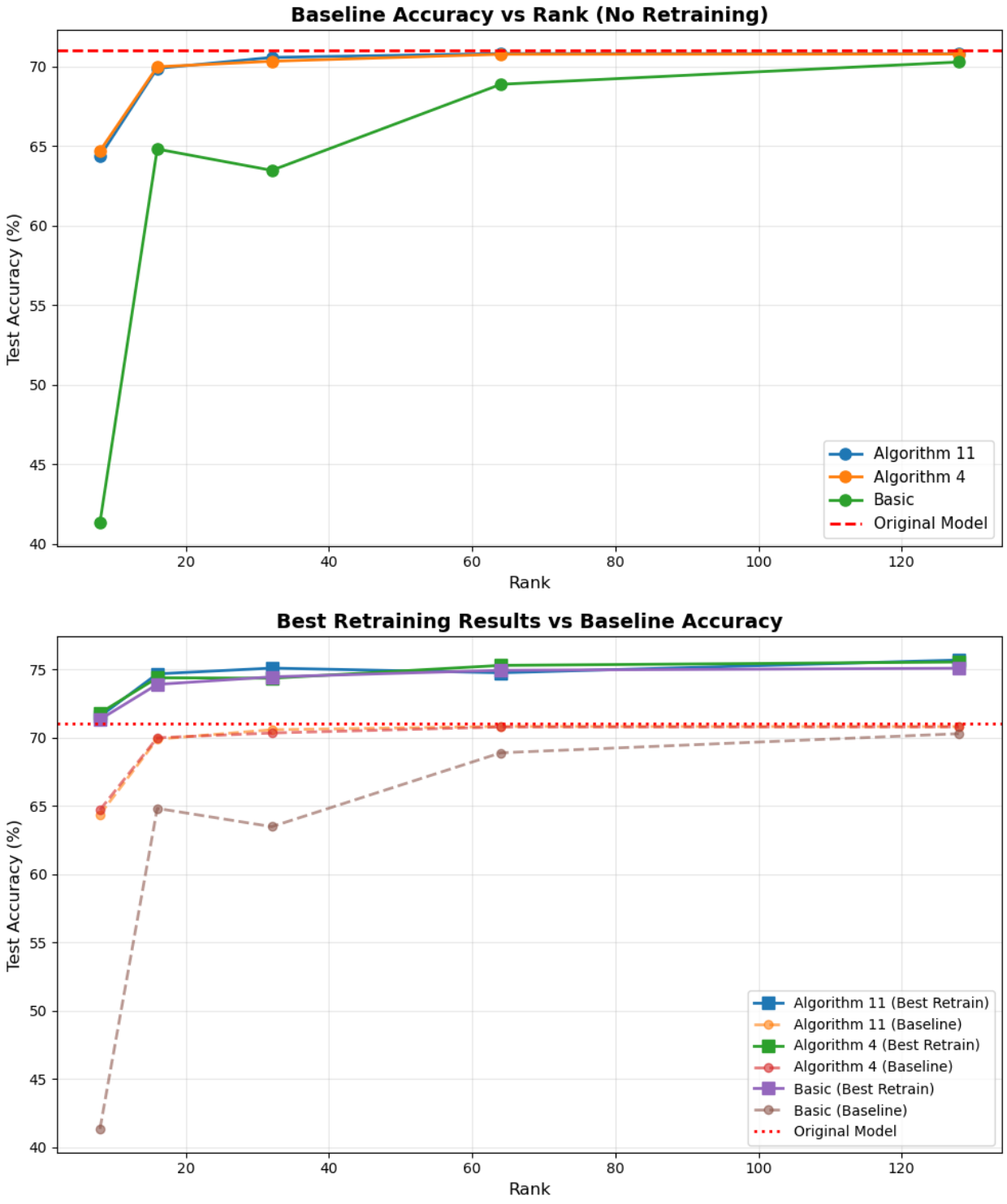



Figure 5: Baseline accuracy vs rank (before and after retraining)

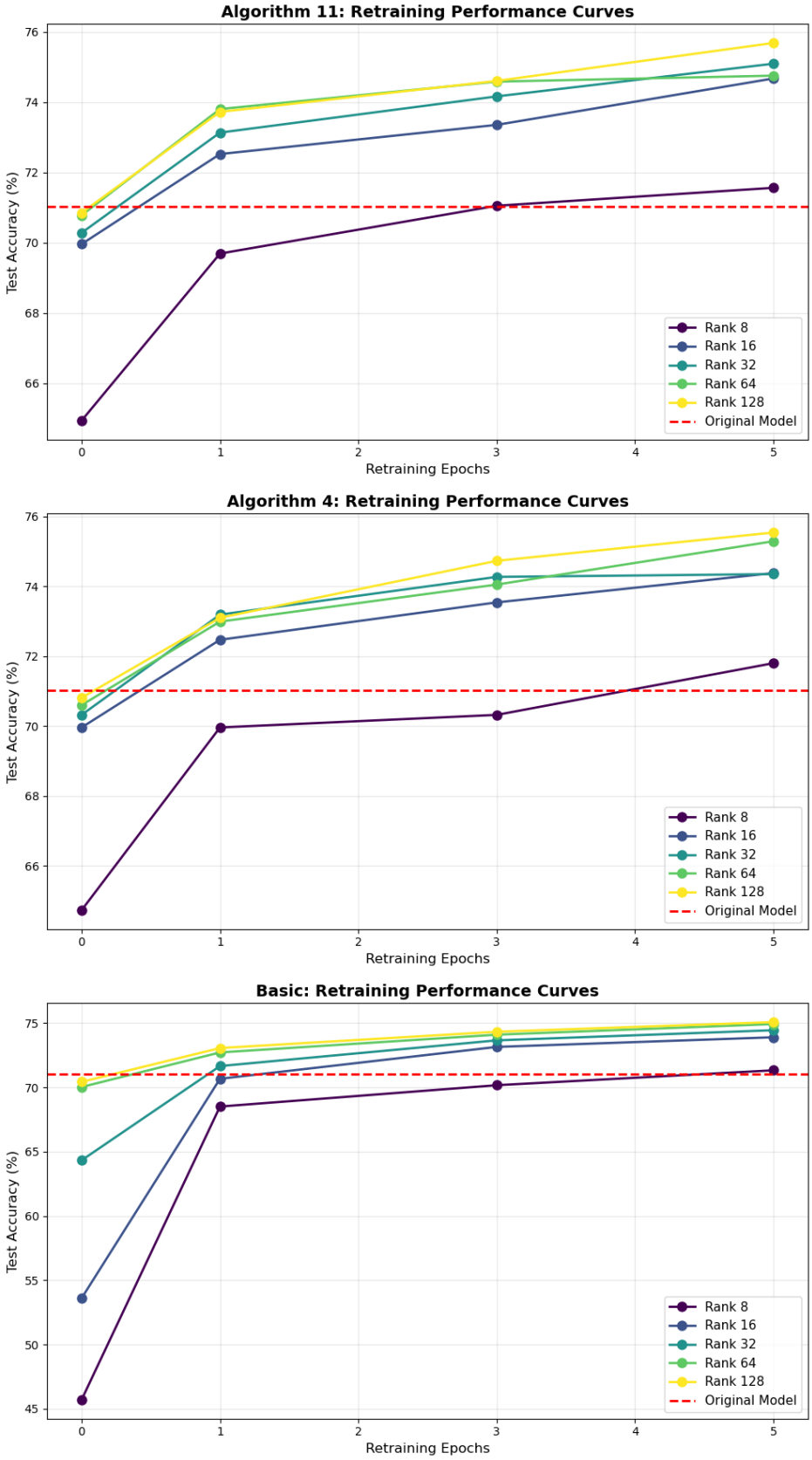
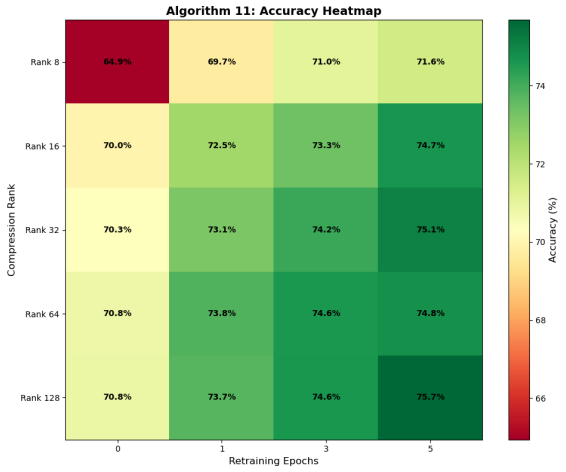
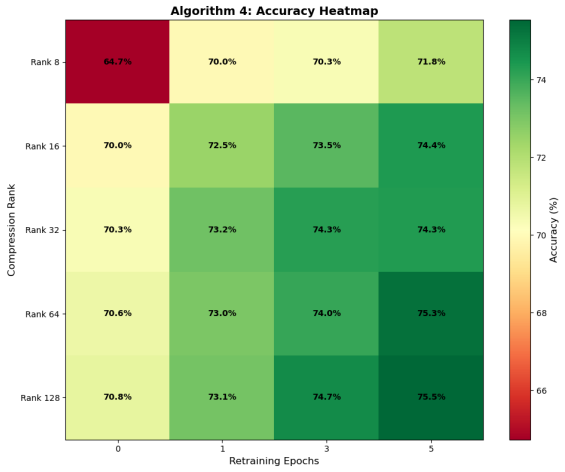


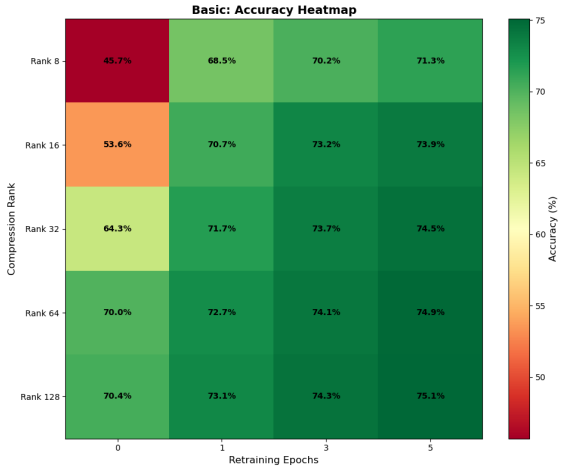

Figure 6: Accuracy after retraining vs epochs



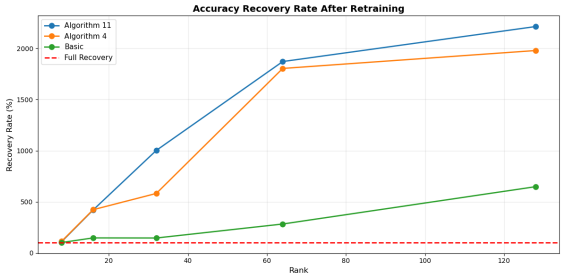

(a) Heatmap for Algorithm 11




(b) Heatmap for Algorithm 4




(c) Heatmap for Basic (Algorithm 2)




(d) Accuracy recovery against rank, a comparison of all the algorithms

Figure 7: The heatmaps are before the retraining.

Finally, the neural architecture used is described in more detail as:

$$
\begin{aligned}
\textbf{Input:} &\quad (3, 32, 32) \\
&\downarrow \\
\textbf{Conv Layer 1:} &\quad (3, 32, 32) \to (32, 32, 32) \quad \text{[Kernel: } 3 \times 3\text{, Padding: 1]} \\
\textbf{Activation:} &\quad \text{ReLU} \\
&\downarrow \\
\textbf{Conv Layer 2:} &\quad (32, 32, 32) \to (64, 32, 32) \quad \text{[Kernel: } 3 \times 3\text{, Padding: 1]} \\
\textbf{Activation:} &\quad \text{ReLU} \\
&\downarrow \\
\textbf{Max Pooling:} &\quad (64, 32, 32) \to (64, 16, 16) \quad \text{[Kernel: } 2 \times 2\text{, Stride: 2]} \\
&\downarrow \\
\textbf{Conv Layer 3:} &\quad (64, 16, 16) \to (128, 16, 16) \quad \text{[Kernel: } 3 \times 3\text{, Padding: 1]} \\
\textbf{Activation:} &\quad \text{ReLU} \\
&\downarrow \\
\textbf{Max Pooling:} &\quad (128, 16, 16) \to (128, 8, 8) \quad \text{[Kernel: } 2 \times 2\text{, Stride: 2]} \\
&\downarrow \\
\textbf{Flatten:} &\quad (128, 8, 8) \to (8192) \\
&\downarrow \\
\textbf{Fully Connected Layer 1:} &\quad 8192 \to 512 \\
\textbf{Activation:} &\quad \text{ReLU} \\
\textbf{Dropout:} &\quad p = 0.5 \\
&\downarrow \\
\textbf{Fully Connected Layer 2:} &\quad 512 \to 256 \\
\textbf{Activation:} &\quad \text{ReLU} \\
\textbf{Dropout:} &\quad p = 0.5 \\
&\downarrow \\
\textbf{Fully Connected Layer 3 (Output):} &\quad 256 \to 10 \\
\textbf{Output:} &\quad 10 \text{ classes}
\end{aligned}
$$

- Retraining is Crucial: Even a few epochs of retraining (e.g., 1-3 epochs) significantly recover and often exceed the accuracy of the immediately compressed model. For instance, Algorithm 11 at Rank 8 sees a +7.2% improvement after 5 epochs of retraining.
- Algorithm 11 and Algorithm 4 Performance: Both algorithms, which incorporate power iteration, consistently achieve higher baseline accuracies and generally reach higher final accuracies after retraining compared to the 'basic' method. This aligns with the theoretical benefits of power iteration for matrices with slowly decaying singular values, such as those typically found in neural network weight matrices. The Basic algorithm shows significant degradation at lower ranks (like 8), but it surpasses the other algorithms at higher ranks (at 128).

- Trade-off between Compression and Accuracy: As rank increases (lower compression ratio), accuracy generally improves. Retraining allows for much higher compression ratios while maintaining, or even exceeding, original model accuracy. For example, Algorithm 11 at Rank 128 (2.69× overall compression) achieves 75.68% accuracy after 5 epochs, surpassing the original model's 71.03%.
- Parameter Efficiency: All methods achieve significant parameter reduction in the FC layers, leading to substantial overall model compression.

**Remark 2** *There can be two possible reasons behind the very small drop in accuracy:*

- *The convolution layers, which capture the spatial data, are intact.*
- *The FC3 layer, which also plays a crucial part, is also kept intact.*

*In the next to next experiment, we will see that changing the first and the last layers or the convolutional layers degrades the performance drastically.*

**Remark 3** *The network was already way too complex for the dataset, and hence was able to achieve a very high compression ratio (more than* 10*), retaining the same accuracy. This is later confirmed when the network, after removing* `fc1` *and* `fc2` *layers, achieves an accuracy of* 71.86% *after just training for* 5 *epochs (implemented in "Exp_2_part_2.ipynb").*

# 12 Compression of Convolutional Layer in a CNN

This experiment implements and evaluates three randomized SVD methods for convolutional layer compression (implemented in "Experiment_3_conv_layer.ipynb"):

- Algorithm 11: Randomized SVD with Power Iteration: An enhanced method designed for matrices with slowly decaying singular value spectra, using power iterations to improve the accuracy of the computed low-rank approximation.
- Algorithm 4: Randomized Subspace Iteration: A core component of Algorithm 11, used to construct an approximate basis for the range of a matrix.
- Basic Randomized SVD (Algorithm 2 + Direct SVD): A simpler randomized SVD approach used as a baseline for comparison.

The compression is applied to a specific convolutional layer within a custom CNN trained on the CIFAR-10 dataset. The impact on model accuracy, compression ratio, and computational time is analyzed.

## 12.1 Experimental Setup and Implementation Details

The experiment is structured into several key components:

### 12.1.1 Randomized SVD Algorithms Implementation

The core of the compression relies on the Python implementations of **randomized** SVD algorithms using `numpy` for numerical operations.

- `algorithm_4_randomized_subspace_iteration(A, ell, q)`:
  - **Purpose:** This function implements Algorithm 4, "Randomized Subspace Iteration," which is crucial for finding an orthonormal matrix `Q` that approximates the range of a given matrix `A`. This algorithm is particularly effective for matrices with slowly decaying singular values, as it applies power iterations to enhance the accuracy of the subspace approximation.
  - **Inputs:**
    - `A`: The input matrix (2D NumPy array) to which the **randomized** subspace iteration is applied.
    - `ell`: The target dimension of the subspace (sample size). Its typically chosen as `k + oversampling`, where `k` is the target rank for SVD.
    - `q`: The number of power iterations. Higher `q` values generally lead to better accuracy in capturing the dominant subspace but increase computational cost.
  - **Steps (as per Algorithm 4):**
    1. **Step 1: Generate Gaussian Test Matrix ($\Omega$)**: A random Gaussian matrix $\Omega$ of shape `(n, ell)` is generated, where `n` is the number of columns in `A`. This matrix serves as a random probe to capture information about the column space of `A`.
    2. **Step 2: Form Initial Sample Matrix ($Y\_0$) and QR Factorization**: `Y_0` is computed as the matrix product of `A` and $\Omega$ (`A @ Omega`). A QR **factorization** of `Y_0` is then performed to obtain an initial orthonormal basis `Q_j` (denoted as `Q_0` in the algorithm description).

3. **Step 3: Power Iteration Loop**: This loop runs `q` times to refine the orthonormal basis. In each iteration `j`:
   * `Y_tilde_j` is formed by multiplying the transpose of `A` with the current orthonormal basis `Q_j` (`A.T @ Q_j`).
   * A QR factorization of `Y_tilde_j` is computed to get `Q_tilde_j`. This step essentially projects the row space of `A` onto the current subspace.
   * `Y_j` is formed by multiplying `A` with `Q_tilde_j` (`A @ Q_tilde_j`).
   * Another QR factorization of `Y_j` is computed to get the updated orthonormal basis `Q_j`. This step projects the column space of `A` onto the refined subspace.
4. **Step 4: Output**: The final orthonormal matrix `Q_j` (after $q$ iterations) is returned. This `Q_j` spans an approximate range of $A$.

- `algorithm_11_randomized_svd_power_iteration(A, k, q=2)`:
  - **Purpose:** This function implements Algorithm 11, "Randomized SVD with Power Iteration," which computes a low-rank SVD approximation ($U\Sigma V^T$) of a matrix `A`. It leverages Algorithm 4 to efficiently find an approximate range.
  - **Inputs:**
    * `A`: The input matrix (2D NumPy array) for which to compute the SVD.
    * `k`: The target rank for the low-rank approximation.
    * `q`: The number of power iterations to use in Algorithm 4 (default is 2).
  - **Steps (as per Algorithm 11):**
    1. **Step 1: Generate Gaussian Test Matrix ($\Omega$)**: A random Gaussian matrix $\Omega$ of shape `(n, 2 * k)` is implicitly generated within the call to

       `algorithm_4_randomized_subspace_iteration`.

       The `2 * k` is used as the `ell` parameter for oversampling, typically `k + p` where `p` is an oversampling parameter (here `p = k`).
    2. **Step 2: Form Y and Construct Q**: Instead of explicitly forming $Y = (AA^*)^q A$, the implementation directly calls

       `Q = algorithm_4_randomized_subspace_iteration(A, 2 * k, q)`.

       This `Q` is an orthonormal matrix spanning the approximate range of `A` after `q` power iterations.
    3. **Step 3: Form B**: `B` is computed as the product of the transpose of `Q` and `A` (`Q.T @ A`). This projects `A` onto the smaller subspace spanned by `Q`.
    4. **Step 4: Compute SVD of B**: A standard (deterministic) SVD is performed on the smaller matrix `B` to obtain `U_tilde`, `Sigma`, and `Vt`.
    5. **Step 5: Set U**: The final `U` matrix is computed by multiplying `Q` with `U_tilde` (`Q @ U_tilde`).
    6. **Step 6: Truncate to Rank k**: The `U`, `Sigma`, and `Vt` matrices are truncated to the desired rank `k`.
  - The low-rank SVD components `U`, `Sigma`, and `Vt`.

- `algorithm_2_basic_randomized_range_finder(A, ell)`:

- **Purpose:** Implements Algorithm 2, "Basic Randomized Range Finder," to find an orthonormal matrix `Q` that approximates the range of `A`. This is a simpler version without power iterations.
- **Inputs:** `A` (input matrix), `ell` (sample size).
- **Steps:**
  1. Generate `n x ell` Gaussian random matrix `Omega`.
  2. Form `Y = A @ Omega`.
  3. Compute QR factorization `Y = QR` to get orthonormal `Q`.
- **Output:** `Q`.

- `basic_randomized_svd(A, rank, oversampling=10)`:
  - **Purpose:** A wrapper that combines Algorithm 2 with a direct SVD to perform basic **randomized** SVD.
  - **Inputs:** `A` (input matrix), `rank` (target rank), `oversampling` (parameter for `ell`, default 10).
  - **Steps:**
    1. Calculates `ell = min(rank + oversampling, min(A.shape))`.
    2. Calls `algorithm_2_basic_randomized_range_finder` to get `Q`.
    3. Forms `B = Q.T @ A`.
    4. Performs standard SVD on `B` and returns the truncated `U`, `S`, `Vt`.

### 12.1.2 Convolutional Layer Matrix Unfolding

To apply SVD to a convolutional layers weight tensor (which is 4D: `[out_channels, in_channels, kh, kw]`), it must be reshaped into a 2D matrix. This is done using an `im2col`-like operation conceptually.

- `conv_to_matrix(weight, input_shape, stride=1, padding=0)`:
  - **Purpose:** Converts a 4D convolutional weight tensor into a 2D matrix suitable for SVD. The conceptual idea is to flatten the `in_channels`, kernel height (`kh`), and kernel width (`kw`) dimensions into a single dimension.
  - **Input:** `weight` (PyTorch tensor of shape `[out_channels, in_channels, kh, kw]`). `input_shape` is used to infer the full context, though only `weight.shape` is directly used for the matrix conversion.
  - **Implementation:** `weight.view(out_channels, -1)` reshapes the tensor. The `-1` automatically calculates the size of the second dimension (`in_channels * kh * kw`). The result is then converted to a NumPy array for SVD.
  - **Output:** A 2D NumPy array of shape `[out_channels, in_channels * kh * kw]`.

- `matrix_to_conv(U, S, Vt, original_shape)`:
  - **Purpose:** Reconstructs a 4D convolutional weight tensor from its 2D SVD components.
  - **Input:** `U, S, Vt` (NumPy arrays from SVD), `original_shape` (the original 4D shape of the weight tensor).

- **Implementation:** First, the 2D matrix is reconstructed using `U @ np.diag(S) @ Vt`. Then, this reconstructed matrix is converted to a PyTorch tensor and reshaped back to `original_shape` using `.view()`.
- **Output:** A reconstructed 4D PyTorch tensor.

### 12.1.3 Compressed Convolutional Layer

The custom `CompressedConv2D` PyTorch module replaces a standard `nn.Conv2d` layer with its low-rank approximation. The low-rank approximation of a weight matrix $W \approx U\Sigma V^T$ can be implemented as two sequential convolutional layers:

- A first convolutional layer with weights $V^T$ (reshaped). This layer effectively projects the input channels into the lower-dimensional rank space.
- A second convolutional layer ($1 \times 1$ kernel) with weights $U\Sigma$ (reshaped). This layer expands the rank-dimensional features back to the original output channels.
- `__init__(self, U, S, Vt, original_shape, stride=1, padding=0, bias=None)`:
  - Initializes two `nn.Conv2d` layers: `conv1` and `conv2`.
  - `conv1`: `in_channels` to `rank`, with the original kernel size `(kh, kw)`. It does **not** apply `stride` or bias, as its role is to project to the rank space.
  - `conv2`: `rank` to `out_channels`, with a 1x1 kernel. This layer applies the original `stride` and handles the bias.
- `_initialize_from_svd(U, S, Vt)`: This helper method sets the weights of `conv1` and `conv2` using the SVD components.
  - `conv1.weight.data` is set to `V_reshaped` (from `Vt`).
  - `conv2.weight.data` is set to `US` (from `U @ np.diag(S)`), reshaped to `[out_channels, rank, 1, 1]`.
- `forward(self, x)`: Defines the forward pass by sequentially applying `conv1` and `conv2`.

More implementation details are:

- **CNN Architecture** The architecture used is same as the one used in the preceding section.
- The CIFAR-10 dataset is used for training and evaluation. Standard data augmentation (random crop, horizontal flip) and normalization are applied.
- **Training and Evaluation Functions**
  - `train_model(model, trainloader, testloader, epochs, lr, print_freq)`: Standard PyTorch training loop using Adam optimizer and CrossEntropyLoss. Prints training and test accuracy periodically.
  - `evaluate_model(model, dataloader)`: Evaluates the model's accuracy on a given DataLoader in evaluation mode.

#### 12.1.4 Compression and Fine-tuning Logic

- `compress_conv_layer(model, layer_name, rank, method, q)`:
  - **Purpose:** Orchestrates the compression of a specified convolutional layer.
  - **Steps:**
    1. Retrieves the original `nn.Conv2d` layer's weights and bias.
    2. Converts the 4D weight tensor to a 2D matrix using `conv_to_matrix`.
    3. Applies the chosen **randomized** SVD method (`algorithm_11`, `algorithm_4`, or `basic`) to the 2D matrix to get `U`, `S`, `Vt`.
    4. Replaces the original layer in the model with an instance of `CompressedConv2D initialized` with the SVD components.
    5. Calculates and returns the compression ratio (original parameters / compressed parameters) and the time taken for decomposition.
- `fine_tune_model(model, trainloader, testloader, epochs, lr)`:
  - **Purpose:** Retrains the compressed model for a few epochs to recover accuracy lost during compression.
  - **Implementation:** Similar to `train_model` but typically uses a smaller learning rate and fewer epochs.

#### 12.1.5 Analysis and Plotting

- `analyze_conv_layer_spectrum(model, layer_name, plot=True)`:
  - Purpose: Visualizes and quantifies the singular value spectrum of a convolutional layer. This helps understand the inherent low-rank nature of the layer.
  - Implementation: Converts the layer's weight to a matrix, computes full SVD, and then plots:
    * Singular values (log scale)
    * Normalized singular values (log scale)
    * Cumulative energy (to determine rank for a given energy retention).
    * Prints statistics like condition number and rank for 90% energy.
- `plot_compression_results(results, original_accuracy)`: Generates various plots to compare the performance of different compression methods across different ranks:
  - Accuracy vs. Rank (Before Fine-tuning)
  - Accuracy vs. Rank (After Fine-tuning)
  - Accuracy Drop vs. Compression Ratio
  - Decomposition Time vs. Rank
  - Memory vs. Accuracy Trade-off
  - Fine-tuning Recovery Rate

### 12.1.6 Additional Analysis Functions

- `compare_compression_methods_detailed()`: Analyzes singular value spectra of multiple convolutional layers (`conv1, conv2, conv3`) to identify potential ranks for compression based on energy thresholds.
- `memory_analysis()`: Calculates and reports the total and convolutional layer parameter counts and memory usage (in MB and KB) for the original and compressed models at different ranks.
- `robustness_test()`: Investigates the impact of varying $q$ (power iterations) and oversampling ratios on the compression quality (accuracy and decomposition time).

## 12.2 Experiment Execution and Observations

The main execution block (`if __name__ == "__main__":`) orchestrates the entire experiment:

1. **Reproducibility**: Sets random seeds for PyTorch and NumPy.
2. **Data Loading**: Loads the CIFAR-10 dataset.
3. **Original Model Training**: Trains an uncompressed `ConvTestCNN` model for 15 epochs to establish a baseline accuracy.
   - Observation: The original model achieved a test accuracy of approximately **79.25%**.
4. `conv2` **Layer Spectrum Analysis**: Analyzes the singular value spectrum of the `conv2` layer.
   - Observation: The `conv2` layer (matrix shape 64x288) has a rank of 64. Its singular values decay, and **90% of the energy is captured at rank 39**. This suggests that significant compression might be possible around this rank without a drastic loss of information.
5. **Comprehensive Compression Experiment**:
   - Iterates through different target ranks (`[8, 16, 24, 32]`) and compression methods (`'algorithm_11'`, `'algorithm_4'`, `'basic'`).
   - For each combination:
     - A fresh copy of the original trained model is created.
     - The `conv2` layer is compressed using the specified method and rank.
     - Accuracy is evaluated before fine-tuning.
     - The compressed model is fine-tuned for 5 epochs with a smaller learning rate.
     - Accuracy is evaluated after fine-tuning.
     - Compression ratio and decomposition/fine-tuning times are recorded.

### 12.2.1 Results and Observations

The singular value analysis of the convolutional layers is shown in the figure 8. The rest of the plots are shown in the figure 9. More detailed results are given in the table 3.

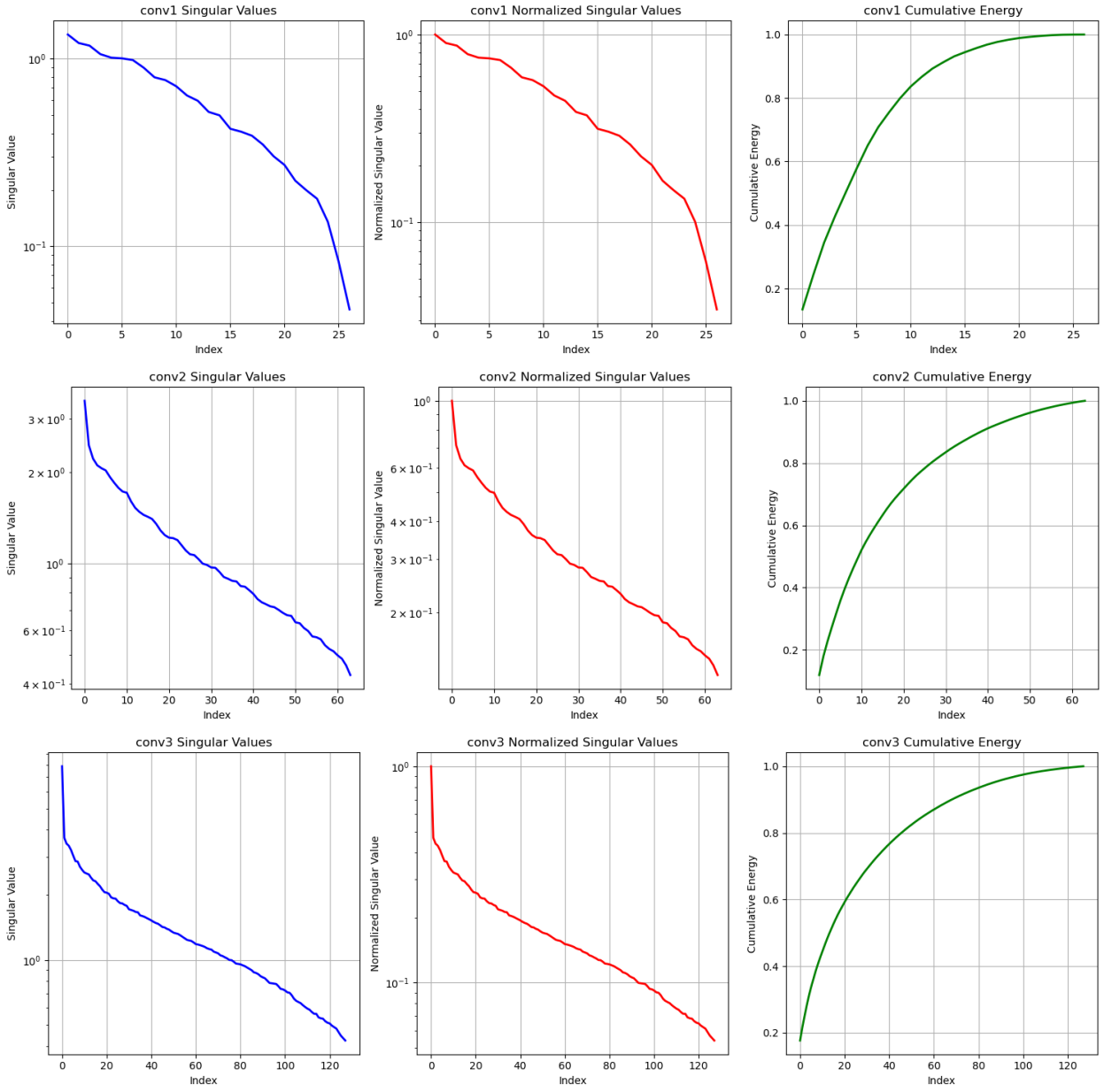
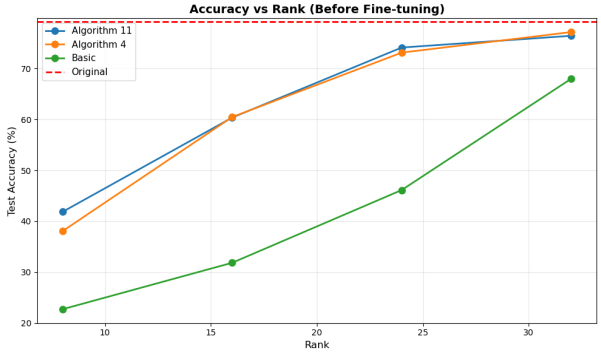


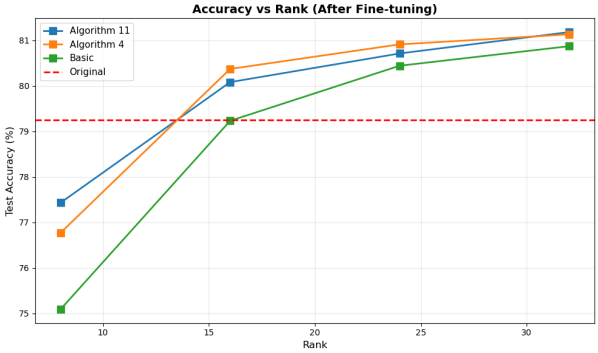

Figure 8: Singular values for the decomposed convolutional layers in the network.

Table 3: Performance Metrics of Different Compression Methods, compression ratio is just for the second conv layer not overall compression.

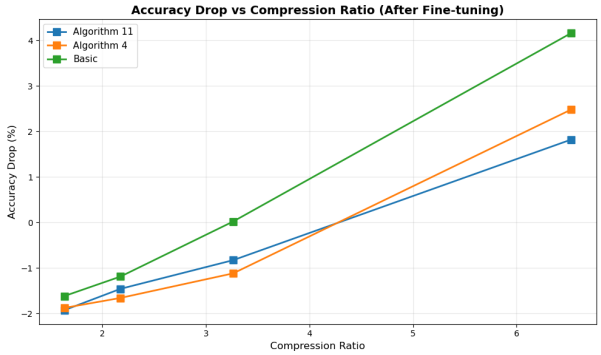
| **Method** | **Rank** | **Accuracy Before (%)** | **Accuracy After (%)** | **Compression Ratio (x)** | **Decomp Time (s)** | **Finetune Time (s)** |
|---|---|---|---|---|---|---|
| ALGORITHM 11 | 8 | 41.85 | 77.43 | 6.5 | 0.001 | 44.6 |
| ALGORITHM 11 | 16 | 60.36 | 80.08 | 3.3 | 0.003 | 43.7 |
| ALGORITHM 11 | 24 | 74.13 | 80.71 | 2.2 | 0.004 | 45.9 |
| ALGORITHM 11 | 32 | 76.44 | 81.18 | 1.6 | 0.004 | 47.1 |
| ALGORITHM 4 | 8 | 38.06 | 76.77 | 6.5 | 0.001 | 44.6 |
| ALGORITHM 4 | 16 | 60.47 | 80.37 | 3.3 | 0.002 | 43.7 |
| ALGORITHM 4 | 24 | 73.13 | 80.91 | 2.2 | 0.002 | 45.9 |
| ALGORITHM 4 | 32 | 77.17 | 81.13 | 1.6 | 0.003 | 36.5 |
| BASIC | 8 | 22.71 | 75.09 | 6.5 | 0.001 | 35.3 |
| BASIC | 16 | 31.83 | 79.23 | 3.3 | 0.001 | 35.2 |
| BASIC | 24 | 46.12 | 80.44 | 2.2 | 0.002 | 36.6 |
| BASIC | 32 | 67.99 | 80.87 | 1.6 | 0.002 | 37.3 |



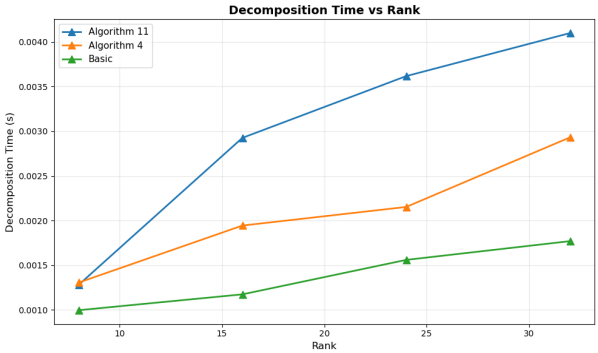

(a) Accuracy-Rank (before retraining)



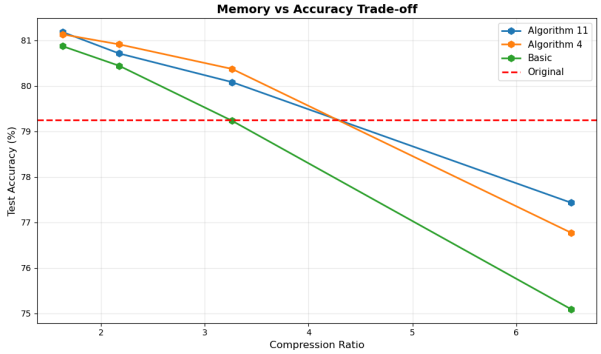

(b) Accuracy-Rank (after retraining)



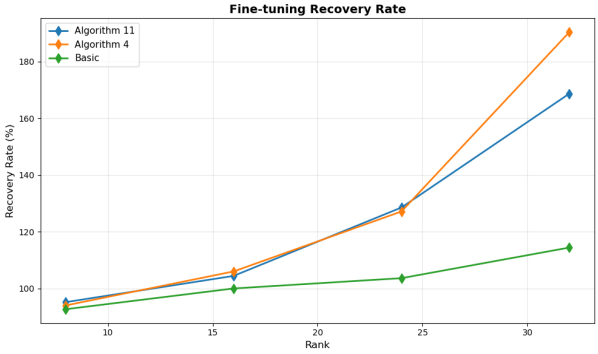

(c) Accuracy drop-compression ratio




(d) Decomposition time - Rank




(e) Accuracy-Compression Ratio




(f) Recovery rate - Rank

Figure 9: Plots generated for the experiment.

#### 12.2.2 Observations

- **Accuracy vs. Rank:** Higher ranks generally yield better accuracy, both before and after fine-tuning.
- **Impact of Fine-tuning:**
  - Fine-tuning is crucial for recovering accuracy.
  - For lower ranks (e.g., rank 8), the drop in accuracy before fine-tuning is substantial, but a significant portion is recovered afterward.
  - For higher ranks, the initial drop is smaller, and fine-tuning still provides a boost.
  - In some cases, fine-tuning even improves accuracy beyond the original model (e.g., Algorithm 11 at rank 32 achieves 81.18% vs. 79.25% original).
  - This improvement may be attributed to low-rank approximation acting as a form of regularization.
- **Compression Ratio:**
  - All methods achieve the same compression ratios for a given rank, as they share the same low-rank factorization mechanism.
  - For rank 8, a substantial 6.5× compression is achieved.
- **Decomposition Time:**
  - Decomposition times are extremely fast (in milliseconds) across all methods and ranks.
  - Algorithm 11 and Algorithm 4 are slightly slower than BASIC due to the overhead of power iterations.
- **Method Comparison (Accuracy):**
  - **Algorithm 11** typically achieves the highest or comparable accuracy, especially at higher ranks. It reaches 81.18% at rank 32.
  - **Algorithm 4** is highly competitive, often producing similar post-finetuning accuracies.
  - **BASIC** starts with lower pre-finetuning accuracy, especially at lower ranks. Fine-tuning improves it significantly, but it generally lags behind the other two.
  - This indicates the effectiveness of power iterations (used in Algorithms 4 and 11) in producing better initial approximations.
- **Best Configurations Analysis:**
  - **Algorithm 11 (Rank 32):** Highest accuracy (81.18%), 1.6× compression, total time: 47.1s.
  - **Algorithm 4 (Rank 32):** Very close performance (81.13%), slightly faster total time: 36.5s.
- **Robustness Testing** Increasing $q$ from 1 to 4 for Algorithm 11 (at rank 16) generally improves the accuracy before fine-tuning and slightly improves the final accuracy. For instance, $q = 1$ yields 79.55% after fine-tuning, while $q = 4$ yields 79.93%. This confirms that power iterations help in better capturing the dominant subspace, as stated in the "Power Iteration Error Bound" theorem. The decomposition time remains consistently low. $q = 2$ yields

Table 4: Fine-tuning Results with Varying Oversampling

| Oversampling | Accuracy Before (%) | Accuracy After (%) |
|---|---|---|
| 0 | 24.08 | 76.63 |
| 5 | 32.25 | 78.14 |
| 10 | 41.60 | 77.78 |
| 15 | 58.39 | 79.25 |

79.89% after fine-tuning and $q = 3$ yields 79.93% (same as $q = 4$) after fine-tuning, suggesting increasing $q$ beyond 2 is rarely beneficial.

- **Oversampling Ratios (for Basic Randomized SVD)** Increasing the oversampling ratio from 0 to 15 (at rank 16) significantly improves accuracy, especially before fine-tuning. For example, oversampling 0 yields 24.08% before fine-tuning, while oversampling 15 yields 58.39%. This highlights the importance of the oversampling parameter $p$. The decomposition time remains minimal. Detailed results are given in the table 4.

### 12.3 Conclusions

The experiment successfully demonstrates the feasibility and effectiveness of compressing convolutional layers using randomized SVD algorithms, particularly Algorithm 11: Randomized SVD with Power Iteration.

- **Significant Compression:** Substantial compression ratios (e.g., 6.5x) can be achieved for convolutional layers with minimal impact on final model accuracy after fine-tuning.
- **Accuracy Recovery:** Fine-tuning is a critical step that effectively recovers and sometimes even improves the accuracy of compressed models, potentially due to the regularization effect of low-rank approximation.
- **Efficiency of Randomized SVD:** The decomposition times for all randomized SVD methods are remarkably fast, making them practical for large-scale model compression.
- **Superiority of Power Iterations:** Algorithms 4 and 11, which incorporate power iterations, consistently provide better initial approximations and generally lead to higher final accuracies compared to the basic randomized SVD, especially at lower ranks. This confirms the theoretical advantages of power iteration for matrices with non-rapidly decaying singular values.
- **Importance of Oversampling:** The robustness tests highlight that appropriate oversampling is crucial for the performance of randomized SVD, even for the basic method.

This experiment validates the practical applicability of randomized SVD for efficient neural network compression, offering a valuable tool for deploying models in resource-constrained environments.

# 13 Layer-wise Compression Impact Analysis

This experiment applies randomized SVD algorithms to compress individual layers of a pre-trained CNN, evaluating the trade-offs between model size reduction and accuracy. The focus is on understanding which layers are more critical to the model's performance and how effectively a compressed model can recover its accuracy through brief fine-tuning.

## 13.1 Experimental Setup and Technical Implementation

The experiment is implemented in Python using the PyTorch framework for neural network operations and NumPy for numerical computations, particularly for SVD.

### 13.1.1 Reproducibility and Environment

To ensure reproducibility, all random seeds for PyTorch (CPU and CUDA) and NumPy are set to 42 at the beginning of each run. The experiment is configured to run on a CUDA-enabled GPU if available, otherwise it defaults to CPU.

The number of random runs (`N_RANDOM_SEEDS`) for statistical analysis is set to 10, with a confidence level of 0.95.

### 13.1.2 Model Architecture: `LayerAnalysisCNN`

A custom CNN architecture, `LayerAnalysisCNN`, is defined for this analysis. This model is designed to be representative of a typical CNN, featuring:

- **Convolutional Layers**: `conv1` (3→64), `conv2` (64→128), `conv3` (128→256), `conv4` (256→512) with $3 \times 3$ kernel, padding=1, and followed by ReLU and $2 \times 2$ MaxPool.
- **Fully Connected Layers**: `fc1` (`512*2*2 -> 1024`), `fc2` (`1024 -> 512`), `fc3` (`512 -> num_classes`). Dropout (0.5) is applied after `fc1` and `fc2`.
- **Layer Information**: An `OrderedDict` (`self.layer_info`) stores references to each trainable layer (convolutional and linear) along with its type ('`conv`' or '`fc`') and a qualitative position ('early', 'middle', 'late', 'final'). This facilitates systematic layer-wise processing.

### 13.1.3 Data Loading and Model Training

The experiment utilizes both CIFAR-10 and CIFAR-100 datasets to assess the compression impact across different levels of classification complexity.

- **Data Preprocessing**: Standard transformations are applied: random crop, random horizontal flip, ToTensor, and normalization using CIFAR-specific mean and standard deviation.
- **DataLoaders**: `DataLoader` instances are created for training and testing with a batch size of 128 and 2 workers for parallel data loading.
- **Model Training**: The `train_model` function trains the `LayerAnalysisCNN` using the function `nn.CrossEntropyLoss` and `torch.optim.Adam`. The model is trained for 12 epochs for CIFAR-10 and 20 epochs for CIFAR-100. The best test accuracy is tracked and returned.
- **Model Evaluation**: The `evaluate_model` function computes the top-1 accuracy of the model on a given `DataLoader` in evaluation mode (`model.eval()`) without gradient computation.

### 13.1.4 Randomized SVD Implementation

The core compression algorithm used is **Algorithm 11: Randomized SVD with Power Iteration**. The Python implementation (`algorithm_11_randomized_svd_power_iteration`) closely follows the steps:

1. **Generate Gaussian Matrix** $\Omega$: `Omega = np.random.randn(n, 2 * k)` creates an $n \times 2k$ Gaussian random matrix, where $k$ is the target rank and $2k$ accounts for oversampling (as suggested in the algorithm description for $2k$ columns).
2. **Form** $Y = (AA^*)^q A\Omega$: The power iteration is implemented by repeatedly multiplying $A$ and $A^T$ with $Y$. `Y = A @ Omega` is the initial step, followed by `Y = A @ (A.T @ Y)` for $q$ iterations. In this experiment, $q$ is set to 2.
3. **Construct orthonormal** $Q$: `Q, R = np.linalg.qr(Y)` performs QR decomposition to obtain $Q$ with orthonormal columns spanning the range of $Y$.
4. **Form** $B = Q^*A$: `B = Q.T @ A` projects $A$ onto the orthonormal basis $Q$.
5. **Compute SVD of** $B$: `U_tilde, Sigma, Vt = np.linalg.svd(B, full_matrices=False)` computes the SVD of the smaller matrix $B$.
6. **Set** $U = Q\tilde{U}$: `U = Q @ U_tilde` transforms `U_tilde` back to the original matrix's column space.
7. **Truncation**: The singular values `Sigma` and corresponding `U` and `Vt` are truncated to the desired `rank` (which is $k$).

### 13.1.5 Layer-to-Matrix Conversion

To apply SVD to convolutional and linear layers, their weight tensors are reshaped into 2D matrices:

- `conv_to_matrix(weight, input_shape, stride=1, padding=0)`: For a `Conv2d` layer, the weight tensor `(out_channels, in_channels, kh, kw)` is reshaped into a 2D matrix of dimension `(out_channels, in_channels * kh * kw)`. This is a common practice for applying SVD to convolutional filters by treating them as fully connected layers.
- `linear_to_matrix(weight)`: For a Linear layer, the weight tensor is already 2D, so it's directly returned.

### 13.1.6 Compressed Layer Modules

Custom PyTorch modules are created to represent the compressed layers:

- `CompressedConv2D`: This module replaces an original `nn.Conv2d` layer. It uses two sequential `nn.Conv2d` layers to approximate the original convolution via low-rank decomposition. The first `conv1` layer has `in_channels` to `rank` output channels with the original kernel size, and its weights are initialized from $Vt$. The second `conv2` layer has `rank` input channels to `out_channels` with a 1x1 kernel, and its weights are initialized from $U$ @ `np.diag(S)`.
- `CompressedLinear`: This module replaces an original `nn.Linear` layer. It uses two sequential `nn.Linear` layers. The first `linear1` layer maps `in_features` to `rank` features (initialized from $Vt$), and the second `linear2` layer maps `rank` features to `out_features` (initialized from $U$ @ `np.diag(S)`).

- **Initialization from SVD**: Both compressed modules initialize their weights directly from the $U$, $S$, and $Vt$ components obtained from the randomized SVD. This ensures that the compressed layer immediately approximates the original layer's function.

### 13.1.7 Layer-wise Compression and Fine-tuning

The `compress_single_layer` function takes a model, a layer name, and a target rank. It performs the following steps:

1. Retrieves the original layer's weight and bias.
2. Converts the weight to a 2D matrix using `conv_to_matrix` or `linear_to_matrix`.
3. Applies `algorithm_11_randomized_svd_power_iteration` to the layer's weight matrix to get $U$, $S$, $Vt$ for the specified `rank`.
4. Instantiates the appropriate `CompressedConv2D` or `CompressedLinear` module with the SVD components and replaces the original layer in the model.
5. Calculates and returns the compression ratio (original parameters / compressed parameters).

The `fine_tune_compressed_model` function performs a brief fine-tuning step on the compressed model. It uses `nn.CrossEntropyLoss` and `torch.optim.Adam` for 3 epochs.

## 13.2 Statistical Analysis and Reporting

The `layer_wise_compression_analysis_statistical` function orchestrates the main experiment:

1. **Baseline Accuracy**: It first calculates the baseline accuracy of the original (uncompressed) model by evaluating it `N_RANDOM_SEEDS` times. This provides a robust baseline mean, standard deviation, and confidence interval.
2. **Layer Spectrum Analysis**: For each layer, it analyzes the singular value spectrum using `analyze_layer_spectrum`. This function computes the singular values, determines the rank required to retain specific energy (e.g., 70%, 80%, 90%, 95%, 99% of the Frobenius norm squared), calculates the condition number, and effective rank. The Frobenius norm of the weight matrix is also calculated for later normalization.
3. **Statistical Compression Runs**: For each layer and for predefined `compression_levels` (80%, 90%, 95% energy retention), the following steps are repeated `N_RANDOM_SEEDS` times:
    - A deep copy of the original trained model is made to ensure each compression experiment starts from the same state.
    - The specific layer is compressed to the rank determined by the energy retention level.
    - The accuracy is evaluated before fine-tuning.
    - The model is briefly fine-tuned.
    - The accuracy is evaluated after fine-tuning.
    - Metrics such as accuracy drop (from baseline), and recovery (accuracy after - accuracy before fine-tuning) are recorded.

4. **Statistical Aggregation**: For each layer and compression level, the mean, standard deviation, and 95% confidence interval are calculated for `acc_before`, `acc_after`, `acc_drop`, and `recovery` metrics across all random runs.

5. **Normalized Sensitivity**: A 'normalized sensitivity' is calculated as `mean_accuracy_drop / Frobenius_norm_of_layer_weight`. This helps to compare the impact of compression across layers with different scales of weights.

## 13.3 Results and Observations

The experiment provides detailed statistical results and visualisations for both the CIFAR-10 and CIFAR-100 datasets.

### 13.3.1 CIFAR-10 Analysis

- **Baseline Accuracy:** The baseline accuracy for CIFAR-10 is approximately 79.53%.
- **Layer Sensitivity:**
  - `conv1` (Early Conv): Shows positive accuracy drops (e.g., -2.85% at 80% energy retention, -3.21% at 90% energy retention, -3.21% at 95% energy retention), indicating it is sensitive to compression.
  - `fc3` (Final FC): Shows positive accuracy drops (e.g., -3.24% at 80% energy retention, -3.26% at 90% energy retention, -3.40% at 95% energy retention), indicating sensitivity.
  - **Other Layers** (`conv2`, `conv3`, `conv4`, `fc1`, `fc2`): Consistently show positive accuracy drops (e.g., -3.20% to -3.46% for 80% to 95% energy retention), indicating sensitivity.
- **Normalized Sensitivity:** `conv1` and `fc3` have the highest normalized sensitivity, confirming their critical role.
- **Recovery:** All layers show positive recovery values, indicating that brief fine-tuning is effective in mitigating the initial performance degradation caused by compression and, in many cases, even improving upon it.

### 13.3.2 CIFAR-100 Analysis

- **Baseline Accuracy:** The baseline accuracy for CIFAR-100 is approximately 43.24%.
- **Layer Sensitivity:**
  - `conv1` (Early Conv): Shows positive accuracy drops (e.g., -1.27% at 80% energy retention, -1.88% at 90% energy retention, -2.15% at 95% energy retention), indicating sensitivity.
  - `fc3` (Final FC): Shows positive accuracy drops (e.g., -0.89% at 80% energy retention, -1.84% at 90% energy retention, -2.31% at 95% energy retention), indicating sensitivity.
  - **Other Layers** (`conv2`, `conv3`, `conv4`, `fc1`, `fc2`): Consistently show positive accuracy drops (e.g., -2.49% to -2.80% for 80% to 95% energy retention), indicating sensitivity.

All these results are shown for CIFAR-10 are shown in the figures 10, 11, 12, and for CIFAR-100 are shown in the figures 13, 14 and 15.

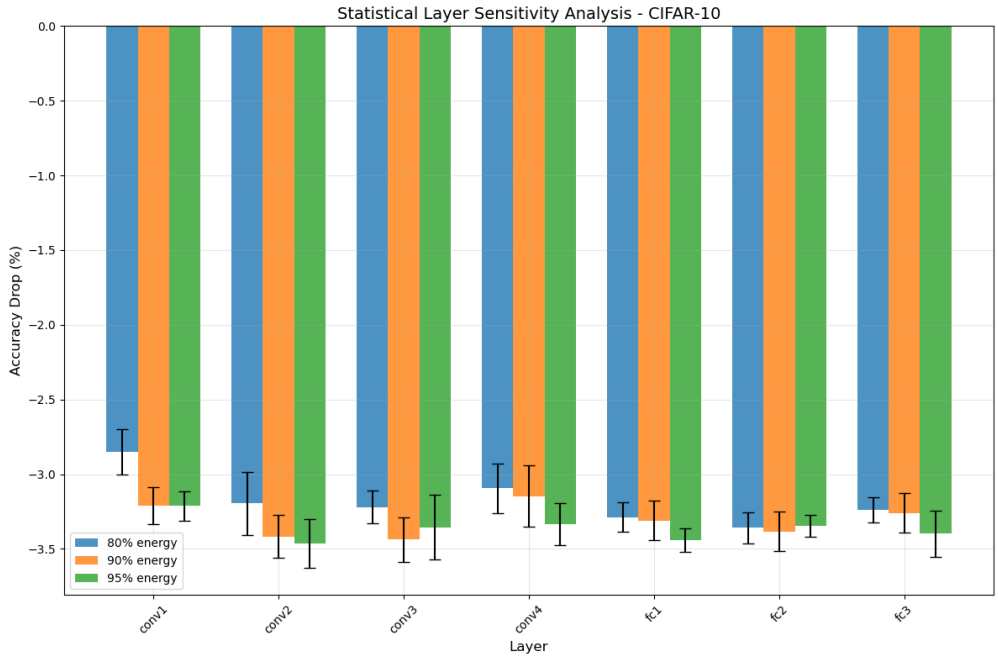



Figure 10: Statistical layer sensitivity (CIFAR-10)

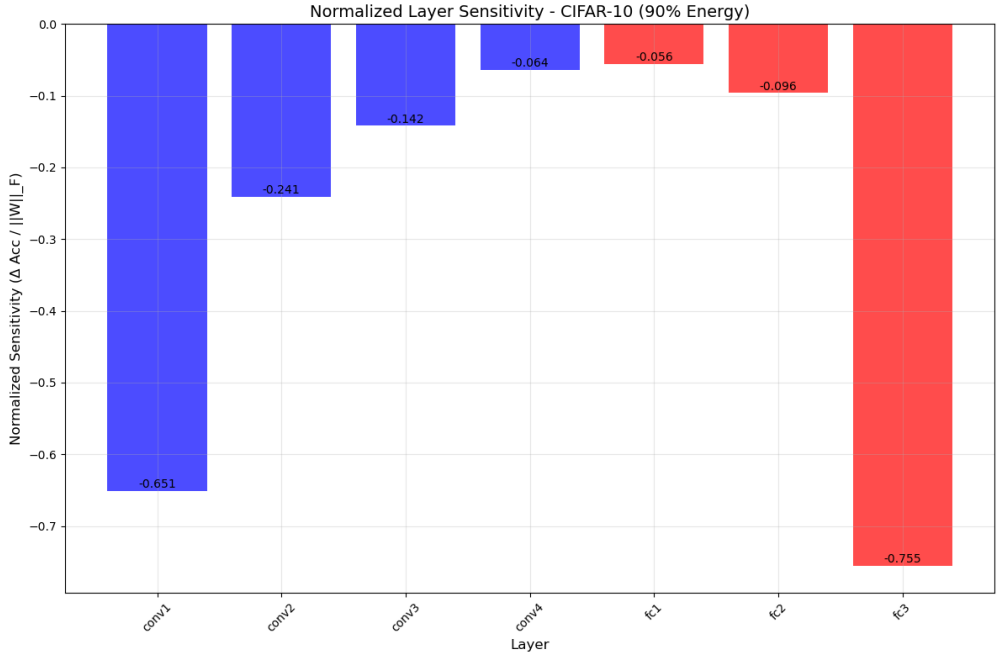



Figure 11: Normalized layer sensitivity (CIFAR-10)

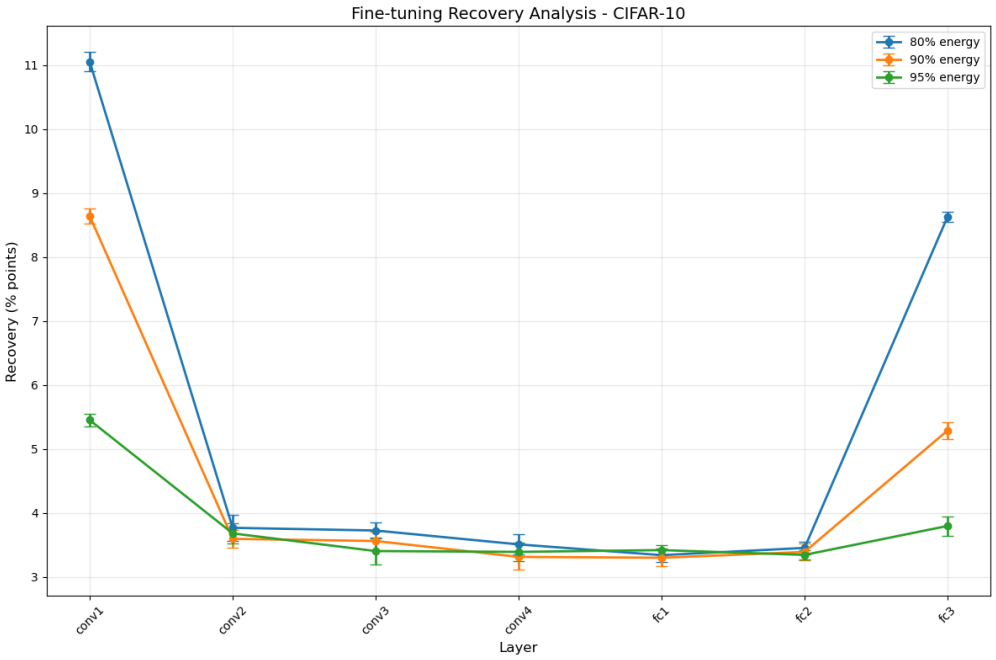



Figure 12: Fine-tuning recovery analysis across energies (CIFAR-10)

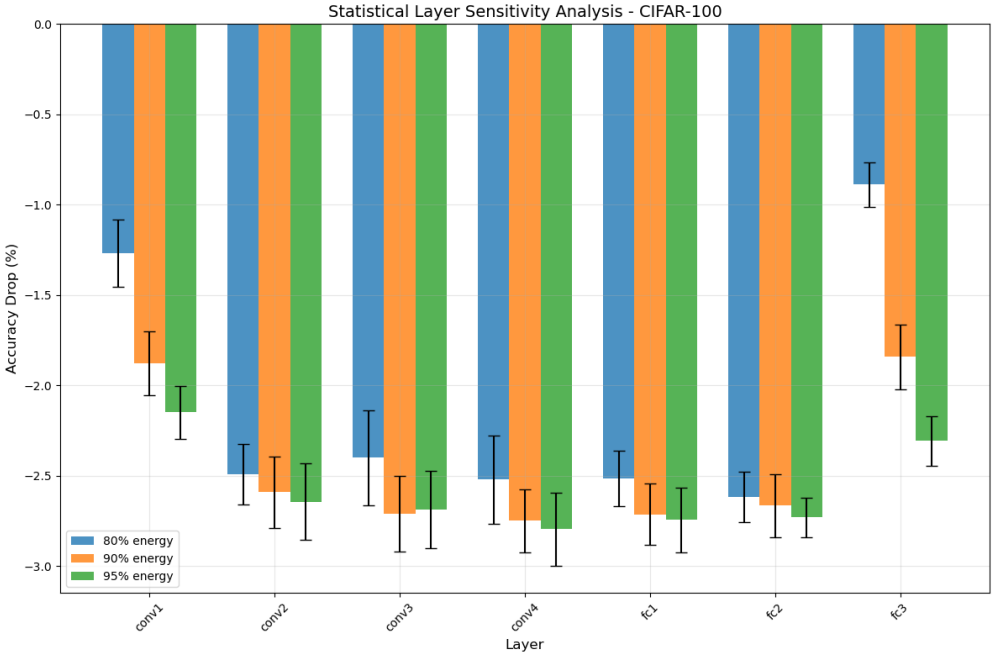



Figure 13: Statistical layer sensitivity (CIFAR-100)

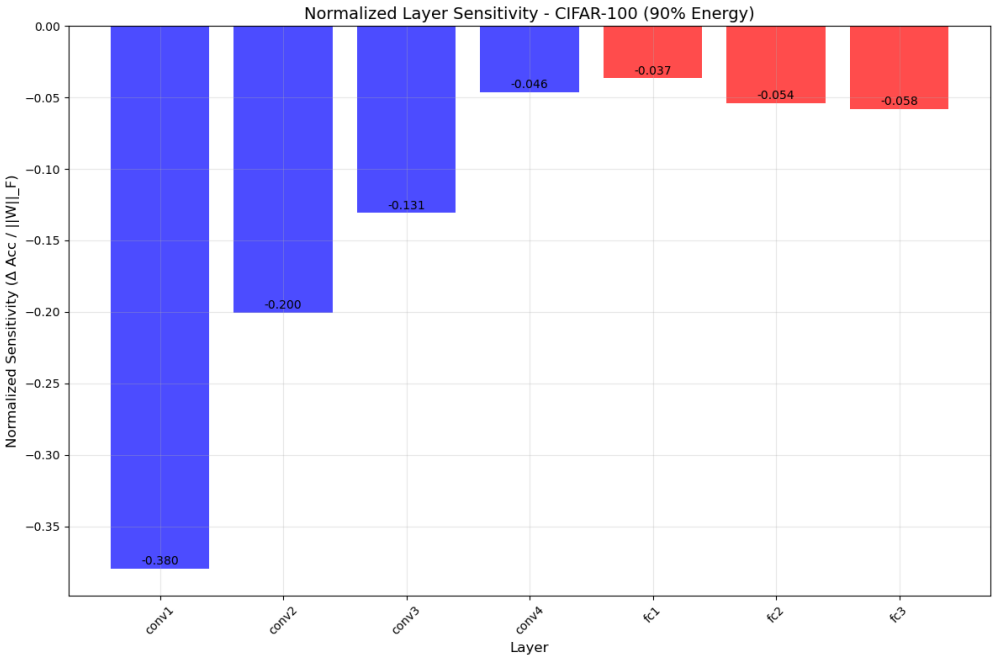



Figure 14: Normalized layer sensitivity (CIFAR-100)

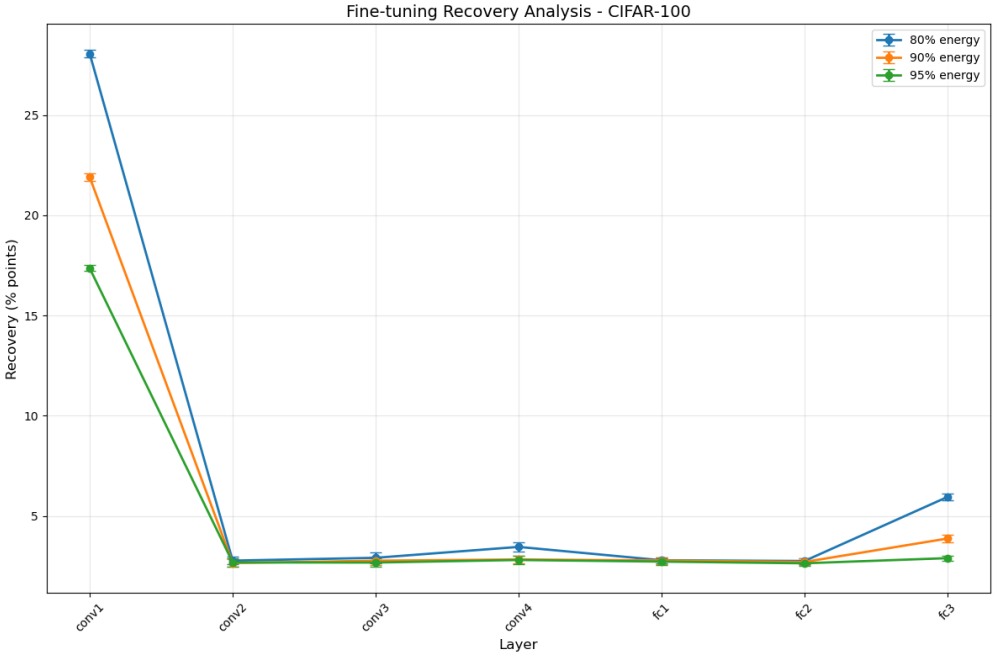



Figure 15: Fine-tuning recovery analysis across energies (CIFAR-100)

### 13.3.3 Cross-Dataset Comparison

- `conv1`: Shows a higher drop in accuracy for CIFAR-10 (-3.21%) compared to CIFAR-100 (-1.88%) at 90% energy retention.
- `conv4`: Shows a higher drop in accuracy for CIFAR-10 (-3.15%) compared to CIFAR-100 (-2.75%) at 90% energy retention.
- `fc1`: Shows a higher drop in accuracy for CIFAR-10 (-3.31%) compared to CIFAR-100 (-2.71%) at 90% energy retention.
- `fc3`: Shows a higher drop in accuracy for CIFAR-10 (-3.26%) compared to CIFAR-100 (-1.84%) at 90% energy retention.

### 13.3.4 Compression Strategies Comparison

The report compares three compression strategies based on the average accuracy drop per layer:

1. **Uniform Compression:** Compressing all layers uniformly. For CIFAR-10, this leads to average drops of -4.03% (80% energy), -4.86% (90% energy), and -5.05% (95% energy). For CIFAR-100, this leads to average drops of -5.98% (80% energy), -7.34% (90% energy), and -7.84% (95% energy).
2. **Selective Compression (Robust Layers Only):** Compressing only the bottom 50% least sensitive layers. For CIFAR-10, this includes `fc2`, `conv2`, `fc1`, resulting in average drops of -4.47% (80% energy), -4.38% (90% energy), and -4.64% (95% energy). For CIFAR-100, this includes `conv4`, `fc2`, `fc1`, with average drops of -7.22% (80% energy), -7.61% (90% energy), and -7.81% (95% energy). This strategy generally shows similar or slightly worse performance than uniform compression, but can be useful for targeted compression in specific scenarios.
3. **Avoid Critical Layers:** Compressing all layers except the top 2 most critical layers. For CIFAR-10, these avoided layers are `conv4` and `conv1`, and the compressed layers are `conv2`, `conv3`, `fc1`, `fc2`, `fc3`. This results in average drops of -4.18% (80% energy), -4.68% (90% energy), and -4.60% (95% energy). For CIFAR-100, the avoided layers are `conv1` and `fc3`, and the compressed layers are `conv2`, `conv3`, `conv4`, `fc1`, `fc2`. This results in average drops of -7.26% (80% energy), -7.72% (90% energy), and -7.74% (95% energy). This strategy also shows similar or slightly worse performance than uniform compression.

## 13.4 Conclusions

The experiment provides several key insights into layer-wise compression using Randomized SVD with Power Iteration:

1. **All Layers are Sensitive to Compression:** Contrary to some expectations, all layers in the tested CNN model exhibit a drop in accuracy when compressed, even after fine-tuning. This indicates that while compression reduces model size, it consistently introduces some performance degradation. However, as indicated in figures 12 and 15, the highest variance across accuracy recovery is for the first and last layers, indicating the most criticality.
2. **Fine-tuning is Crucial for Recovery:** Brief fine-tuning after compression is highly effective in mitigating the initial performance degradation. The "recovery" metric consistently

shows positive values, meaning that the fine-tuned compressed model performs better than the compressed model before fine-tuning. This highlights the importance of an adaptation phase to help the model adjust to the compressed weights.

3. **Dataset Complexity Influences Sensitivity:** The magnitude of accuracy drop varies between CIFAR-10 and CIFAR-100. Layers in the model trained on CIFAR-100 generally experience larger accuracy drops compared to CIFAR-10 for the same compression levels. This indicates that for more complex tasks, the network's parameters are more critical and less tolerant of aggressive compression. Further, compression ratios for CIFAR-100 are higher than that of CIFAR-10 (more complex dataset) compared to CIFAR-10. This indicates that for more challenging tasks, the network becomes more and more redundant.

4. **No Significant Advantage of Targeted Compression Strategies:** The comparison of compression strategies (uniform, selective, and avoiding critical layers) did not show a clear advantage for selective or avoided strategies over uniform compression in terms of overall accuracy drop. This suggests that for this specific model and datasets, uniformly compressing all layers to the desired energy retention level is as effective as more complex targeted approaches. This finding simplifies practical compression strategies, as a single compression ratio can be applied across all layers without significant performance penalties.

Detailed results are provided in the table 5.

| Dataset | Strategy | Energy | Avg Drop | Notes |
|---|---|---|---|---|
| CIFAR-10 | Uniform Compression | 80% | -4.03% ± 0.27 | All layers compressed uniformly (2.11 avg compression) |
| | | 90% | -4.86% ± 0.51 | All layers compressed uniformly (1.47 avg compression) |
| | | 95% | -5.05% ± 0.40 | All layers compressed uniformly (1.17 avg compression) |
| | Selective Compression (Robust Layers Only) | 80% | -4.47% ± 0.42 | Robust layers: `fc2`, `conv2`, `fc1` (2.34 avg compression) |
| | | 90% | -4.38% ± 0.53 | Same layers (1.53 avg compression) |
| | | 95% | -4.64% ± 0.56 | Same layers (1.17 avg compression) |
| | Avoid Critical Layers | 80% | -4.18% ± 0.22 | Avoided: `conv4`, `conv1`; Compressed: `conv2`, `conv3`, `fc1`, `fc2`, `fc3` (2.18 avg compression) |
| | | 90% | -4.68% ± 0.43 | Same layers (1.50 avg compression) |
| | | 95% | -4.60% ± 0.51 | Same layers (1.18 avg compression) |
| CIFAR-100 | Uniform Compression | 80% | -5.98% ± 1.54 | All layers compressed uniformly (2.65 avg compression) |
| | | 90% | -7.34% ± 1.50 | All layers compressed uniformly (1.64 avg compression) |
| | | 95% | -7.84% ± 1.61 | All layers compressed uniformly (1.26 avg compression) |
| | Selective Compression (Robust Layers Only) | 80% | -7.22% ± 1.35 | Robust layers: `conv4`, `fc2`, `fc1` (2.60 avg compression) |
| | | 90% | -7.61% ± 1.65 | Same layers (1.78 avg compression) |
| | | 95% | -7.81% ± 1.63 | Same layers (1.40 avg compression) |
| | Avoid Critical Layers | 80% | -7.26% ± 1.46 | Avoided: `conv1`, `fc3`; Compressed: `conv2`, `conv3`, `conv4`, `fc1`, `fc2` (2.65 avg compression) |
| | | 90% | -7.72% ± 1.72 | Same layers (1.71 avg compression) |
| | | 95% | -7.74% ± 1.55 | Same layers (1.32 avg compression) |

Table 5: Comparison of Compression Strategies on CIFAR-10 and CIFAR-100. Avg Drop values represent average test accuracy drop at epoch 6 relative to baseline (mean ± std).

**Overall Conclusion:** This experiment demonstrates that Randomized SVD with Power Iteration (Algorithm 11) is a viable technique for CNN compression. While compression consistently

leads to some accuracy drop, fine-tuning effectively mitigates this. Furthermore, for the tested model and datasets, simple uniform compression is as effective as more complex selective compression strategies.

## 13.5 Accuracy drop Analysis

This section is based on the file "Experiment_5_accuracy_plots.ipynb". We explore accuracy drop (before and after training) based on two compression strategies:

- Ratio-based Compression (`compress_all_layers_by_ratio`): For each eligible layer (convolutional and linear), the target rank ($k$) is determined such that the compression ratio of that individual layer (original parameters / compressed parameters) is as close as possible to a specified `target_ratio`. This approach attempts to uniformly compress the parameter count across layers.
- Energy-based Compression (`compress_all_layers_by_energy`): For each layer, a full SVD is performed. The rank ($k$) is then chosen to retain a specified `energy_retention` percentage of the total energy (sum of squared singular values) of the original weight matrix. This method aims to preserve the informational content of the layer while reducing its size.

### 13.5.1 Training and Evaluation Pipeline

1. **Dataset:** The CIFAR-10 dataset was used for this experiment, loaded with standard data augmentation (random crop, horizontal flip) for training and normalization for both training and testing.
2. **Model Architecture:** The same architecture used in the preceding section was used here as well.
3. **Baseline Training:** An uncompressed `LayerAnalysisCNN` model was trained from scratch for 15 epochs on CIFAR-10 to establish a performance baseline.
4. **Compression and Fine-tuning Loop:**
   - For each target compression ratio (or energy retention level), a deep copy of the pre-trained baseline model was created.
   - The `compress_all_layers_by_ratio` or `compress_all_layers_by_energy` function was called to modify the model copy in-place by replacing original layers with their compressed counterparts.
   - The accuracy of the compressed model was evaluated immediately after compression (before fine-tuning).
   - The compressed model was then fine-tuned for a few epochs (6 for ratio-based, 5 for energy-based) with a smaller learning rate to recover lost accuracy. This is a common practice in model compression to mitigate performance degradation.
   - The final accuracy after fine-tuning was recorded.
5. **Parameter Calculation:** The `calculate_total_parameters` function was used to accurately count the trainable parameters of both original and compressed layers/models, enabling the computation of actual compression ratios.

### 13.5.2 Results and Observations

The analysis generated two sets of results: one for ratio-based compression and another for energy-based compression, each detailing accuracy before and after fine-tuning, accuracy drop from baseline, and accuracy recovery. The baseline accuracy for the uncompressed model was 80.90%.

- Ratio-based Compression Analysis:
  - **Accuracy vs. Overall Compression Ratio:** As the target compression ratio increased, the accuracy before fine-tuning generally decreased significantly. For instance, at 1.25x compression, accuracy was 64.60%, dropping to 12.92% at 6.00x compression. This indicates that initial compression, especially at higher ratios, severely impacts performance.
  - **Accuracy Recovery:** A notable observation is the substantial accuracy recovery after fine-tuning. For lower compression ratios (e.g., 1.25x to 1.49x), the final accuracy after fine-tuning (`Acc_After`) often exceeded the baseline accuracy (e.g., 83.40% at 1.25x vs. 80.90% baseline), suggesting that the compression process, followed by fine-tuning, might have a regularization effect, potentially helping the model generalize better or escape local optima. For higher compression ratios (e.g., 2.94x and above), while accuracy before fine-tuning was very low (e.g., 22.55% at 2.94x), the recovery was remarkably high (e.g., +56.63% at 2.94x), bringing the accuracy back into a respectable range (e.g., 79.18%).
  - **Accuracy Drop After Fine-tuning:** The accuracy drop after fine-tuning remained relatively small for a wide range of compression ratios. For instance, up to approximately 2.86x compression, the drop from baseline was less than 1%. Even at 3.98x compression, the drop was only 4.07%. This highlights the effectiveness of fine-tuning in mitigating the performance loss due to aggressive compression.
  - **Highest Compression:** The highest compression achieved was 5.97x, with a final accuracy of 59.93%, representing a significant drop from baseline (20.97%). This shows that while high compression is possible, there's an eventual trade-off in accuracy.
  - **Best Final Accuracy:** The best final accuracy (83.95%) was achieved at 1.33x compression, which is even higher than the baseline, reinforcing the potential regularization benefit of the compression-fine-tuning pipeline.
- Energy Retention Analysis:
  - **Accuracy vs. Energy Retention:** As the energy retention percentage decreased (implying higher compression), the accuracy before fine-tuning also decreased. For example, retaining 99% energy resulted in 80.05% accuracy (before fine-tuning), while 50% energy retention yielded only 17.10% accuracy.
  - **Actual Overall Compression Ratio:** There's an inverse relationship between energy retention and the actual overall compression ratio. Higher energy retention (e.g., 99%) leads to lower compression (0.90x, which is technically expansion), while lower energy retention (e.g., 50%) results in very high compression (7.28x). This is expected, as preserving more energy means retaining more parameters.
  - **Accuracy Recovery:** Similar to ratio-based compression, fine-tuning significantly recovered accuracy across all energy retention levels. The recovery was particularly high for lower energy levels (higher compression), such as +58.31% at 50% energy retention.

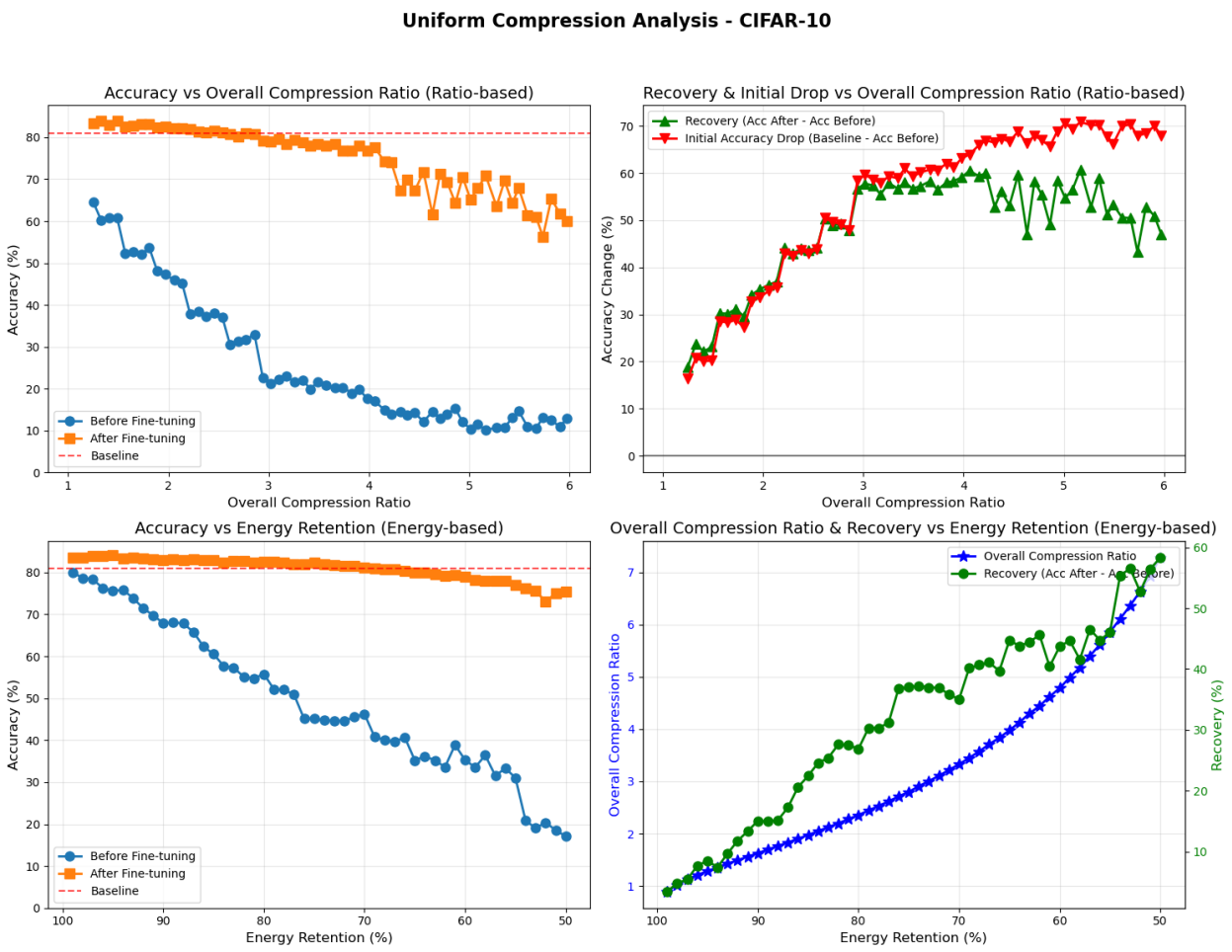



Figure 16: Uniform Compression Analysis (CIFAR-10)

- **Best Energy Setting:** The best final accuracy (84.16%) was observed when retaining 95% of the energy, achieving a 1.29x compression ratio. This again surpassed the baseline accuracy.
- **Trade-off Comparison:** The trade-off plots effectively visualize the relationship between accuracy drop and compression ratio for both methods. Both methods show an increasing accuracy drop as compression ratio increases, but fine-tuning dramatically reduces this drop.

The plots are shown in the figure 16 and figure 17.

## 13.6 Detailed Architecture

Finally, a more detailed description of the used neural architecture is provided below:

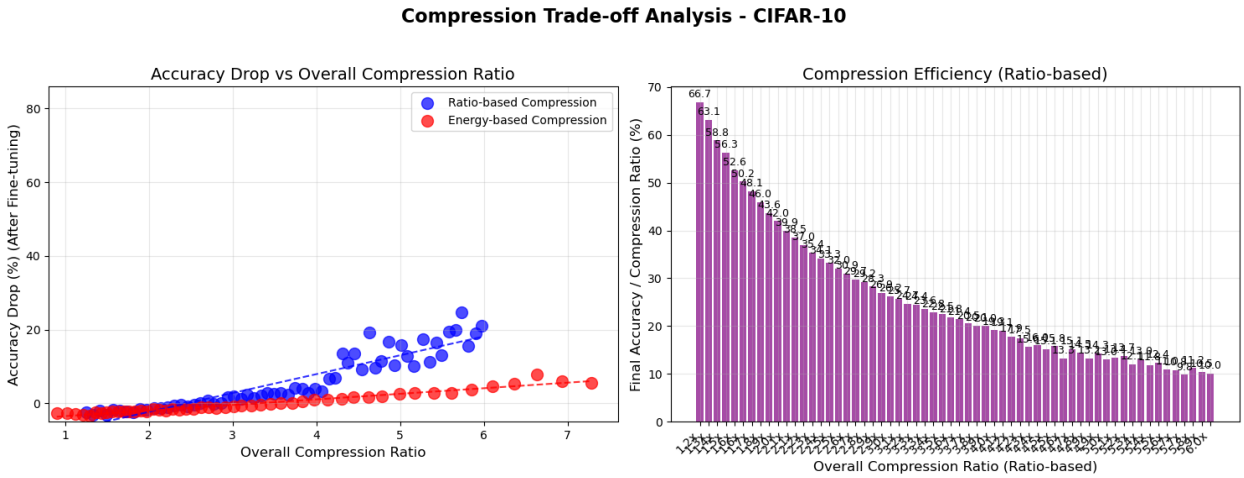



Figure 17: Compression-Tradeoff Analysis (CIFAR-10)

**Input:** $(3, 32, 32)$

$\downarrow$

**Conv1:** $(3, 32, 32) \rightarrow (64, 32, 32)$, Kernel: $3 \times 3$, Pad: 1

ReLU $\rightarrow$ MaxPool($2 \times 2$, Stride: 2) $\rightarrow (64, 16, 16)$

**Conv2:** $(64, 16, 16) \rightarrow (128, 16, 16)$, Kernel: $3 \times 3$, Pad: 1

ReLU $\rightarrow$ MaxPool($2 \times 2$, Stride: 2) $\rightarrow (128, 8, 8)$

**Conv3:** $(128, 8, 8) \rightarrow (256, 8, 8)$, Kernel: $3 \times 3$, Pad: 1

ReLU $\rightarrow$ MaxPool($2 \times 2$, Stride: 2) $\rightarrow (256, 4, 4)$

**Conv4:** $(256, 4, 4) \rightarrow (512, 4, 4)$, Kernel: $3 \times 3$, Pad: 1

ReLU $\rightarrow$ MaxPool($2 \times 2$, Stride: 2) $\rightarrow (512, 2, 2)$

**Flatten:** $(512 \times 2 \times 2) = 2048$

$\downarrow$

**FC1:** $2048 \rightarrow 1024$, ReLU, Dropout ($p = 0.5$)

**FC2:** $1024 \rightarrow 512$, ReLU, Dropout ($p = 0.5$)

**FC3:** $512 \rightarrow$ num_classes

## 13.7 Conclusions

- **Effectiveness of Randomized SVD**: Randomized SVD is an effective technique for compressing neural network layers, significantly reducing the number of parameters while maintaining competitive accuracy.
- **Importance of Fine-tuning**: Fine-tuning is critical for recovering accuracy lost during the compression process. In many cases, especially at moderate compression ratios, fine-tuning can even lead to accuracy improvements beyond the original baseline, suggesting a potential regularization effect from the low-rank approximation.

- **Compression vs. Accuracy Trade-off**: There is a clear trade-off between the degree of compression and the final model accuracy. While high compression ratios can be achieved, they eventually lead to more significant accuracy drops, even after fine-tuning.
- **Comparison of Strategies**: Both ratio-based and energy-based compression strategies are viable. The energy-based approach provides a more direct control over the information retained from the original weight matrix, while the ratio-based approach directly targets parameter reduction. The results indicate similar trends in accuracy recovery and overall performance for comparable compression levels.

### 13.7.1 Phased Accuracy Decline with Compression

Upon detailed examination of the accuracy plots, particularly the "Accuracy vs. Overall Compression Ratio (Ratio-based)" and "Accuracy vs. Energy Retention (Energy-based)" graphs from the experiment, a recurring pattern is observed for the accuracy performance. Instead of a consistently smooth or linear decline, the accuracy often exhibits phases of relative stability where values remain more or less consistent across a range of increasing compression ratios (or decreasing energy retention), followed by a more pronounced, significant drop in accuracy. This pattern then appears to repeat as compression becomes more aggressive.

### 13.7.2 Conclusion from this Observation

This phased accuracy decline suggests that neural networks, when compressed using randomized SVD and subsequently fine-tuned, possess a degree of robustness or tolerance to parameter reduction up to a certain point. Within these "stable phases", the redundant information or less critical components of the original weight matrices are effectively pruned without significantly compromising the model's overall performance. This is likely because the randomized SVD, especially with power iterations (Algorithm 11), efficiently identifies and retains the most critical singular values and corresponding subspaces that capture the majority of the matrix's "energy" or information.

The "significant drops" that occur after these stable phases indicate that the compression has surpassed a critical threshold where the removed parameters are no longer redundant but are essential for the model's performance. Beyond this threshold, the low-rank approximation begins to discard increasingly vital information, leading to a more noticeable degradation in accuracy that even fine-tuning cannot entirely compensate for.

This observation is highly valuable for practical model deployment. It implies that there might be an "optimal sweet spot" or a range of compression ratios/energy retention levels where significant model size reduction can be achieved with minimal to no accuracy loss, and potentially even slight improvements due to regularization. Beyond this sweet spot, further compression will come at a more considerable cost to performance. Identifying these thresholds for specific models and tasks is crucial for efficient model optimization.

# 14 Compression Strategy

The experiments suggest a phased approach to compression, focusing on preserving the most critical layers while aggressively compressing others, and crucially, incorporating fine-tuning at each stage to recover and even enhance accuracy.

## 14.1 Identify and Preserve Critical Layers:

- Early Convolutional Layers (e.g., `conv1`): The `conv1` layer is consistently identified as the most sensitive to compression across both CIFAR-10 and CIFAR-100 datasets. Compressing it too aggressively can lead to substantial accuracy drops, especially for more complex datasets like CIFAR-100. Strategy: Initially, avoid compressing `conv1`, or compress it very cautiously (e.g., retaining 95% or more energy) and with immediate fine-tuning.
- Final Fully Connected Layer (e.g., `fc3`): The final FC layer (`fc3`) also shows some sensitivity, particularly at lower energy retention levels, though less so than `conv1`. Strategy: Similar to `conv1`, compress `fc3` with caution, prioritizing higher energy retention (e.g., 90% or 95%).

## 14.2 Aggressively Compress Robust Middle Layers (with Fine-tuning):

Middle Convolutional and Fully Connected Layers (e.g., `conv2`, `conv3`, `conv4`, `fc1`, `fc2`): These layers consistently exhibit a "negative accuracy drop" (meaning accuracy improves after compression and fine-tuning). This suggests that low-rank approximation acts as a regularization effect for these layers, potentially leading to better generalization.

Strategy: For these layers, more aggressive compression ratios can be applied (e.g., targeting 80% energy retention or higher compression ratios). The crucial step it to always follow compression with fine-tuning (even a few epochs like 1-3) for these layers to fully leverage the regularization effect and recover/improve accuracy. The first epoch of retraining is particularly effective at recovering most of the accuracy.

## 14.3 Fine-tuning is Essential Throughout

Fine-tuning dramatically improves accuracy, especially at lower compression ranks, and can even lead to accuracy exceeding the original uncompressed model. Incorporate fine-tuning steps after compressing any layer or set of layers. Using a smaller learning rate (e.g., 0.0005 or 0.0001) is recommended for fine-tuning.

## 14.4 Proposed Robust Compression Method

1. Train the Original Model: Train your full, uncompressed CNN to establish a performance baseline.
2. Analyze Layer Sensitivity:
   (a) Perform singular value spectrum analysis for each layer to understand their decay characteristics and inherent low-rank nature.
   (b) Identify the "critical" layers (like `conv1` and `fc3`) that show higher accuracy drops upon compression, even with fine-tuning.
   (c) Identify "robust" layers (e.g., `conv2`, `conv3`, `conv4`, `fc1`, `fc2`) that show accuracy improvements after compression and fine-tuning

3. Initial Targeted Compression (Critical Layers):
    (a) First Convolutional Layers (`conv1`): Apply minimal compression, if any (e.g., 95% energy retention or higher, or skip compression entirely for this layer if extreme accuracy is paramount)
    (b) Last Fully Connected Layers (`fc3`): Apply moderate compression (e.g., 90-95% energy retention).
    (c) Fine-tune the Entire Model: After compressing these critical layers, fine-tune the entire model for a few epochs (e.g., 3-5 epochs) with a small learning rate. This helps the network adapt to the initial changes.
4. Aggressive Compression (Robust Layers):
    (a) Middle Convolutional and Fully Connected Layers (`conv2`, `conv3`, `conv4`, `fc1`, `fc2`): For these layers, more aggressive compression ratios can be applied (e.g., aiming for 80% energy retention or higher parameter compression ratios).
    (b) Fine-tune the Entire Model: After compressing this second set of layers, perform another round of fine-tuning on the entire network. This is where the "regularization effect" is most likely to yield accuracy improvements
5. Iterative Shedding: Try to iteratively shed as many parameters as can be shed without significantly damaging the accuracy.

This strategy leverages the insights from the experiments, acknowledging the varying sensitivity of different layers and the power of post-compression fine-tuning to maintain and even enhance model performance.

# 15 Iterative Layer-Sensitive Compression Strategy for CNNs

This section details the implementation and evaluation of an iterative layer-sensitive compression strategy for Convolutional Neural Networks (CNNs), building upon the randomized Singular Value Decomposition (SVD) framework, specifically **Algorithm 11: Randomized SVD with Power Iteration**. The primary goal is to assess the trade-offs between model size reduction and classification accuracy on CIFAR-10 and CIFAR-100 datasets, with a strong emphasis on preventing data leakage between the validation and test sets. The methodology involves an adaptive, iterative compression approach that applies varying energy retention levels to different layers based on their sensitivity, followed by fine-tuning.

A validation set is exclusively used for model selection, early stopping during training, and guiding compression decisions, while the test set is reserved for a single, final evaluation of the compressed model's performance (implemented in "Experiment_6_compression_pipeline_cifar_10.ipynb" and "Experiment_6_compression_pipeline_cifar_100.ipynb".

## 15.1 Implementation Details

- **Model Architecture:** The `LayerAnalysisCNN` implements the architecture as:
  - Convolutional Layers: `conv1 (364)`, `conv2 (64128)`, `conv3 (128256)`, `conv4 (256512)`. Each convolutional layer uses a `3`3 kernel with padding=1, followed by a ReLU activation function and `2`2 Max Pooling.
  - Fully Connected Layer: A single fully connected layer, `fc3`, connects directly from the flattened output of `conv4` (`51222=2048` features) to `num_classes` (10 for CIFAR-10, 100 for CIFAR-100). A Dropout layer with p=0.5 is applied before `fc3`.
  - Layer Information: An `OrderedDict` named `self.layer_info` stores references to each trainable layer (`conv1, conv2, conv3, conv4, fc3`) along with its type and a qualitative position (`early, middle, final`). This dictionary facilitates systematic layer-wise processing.
- **Data Loading and Model Training:** The experiments utilize the CIFAR-10 and CIFAR-100 datasets to evaluate compression effectiveness across different classification complexities.
  - **Data Preprocessing:** Standard transformations like `RandomCrop(32, padding=4)`, `RandomHorizontalFlip()`, `ToTensor()`, and normalization using dataset-specific mean and standard deviation.
    - For CIFAR-10: The mean used is `(0.4914, 0.4822, 0.4465)` and the standard deviation used is `(0.2023, 0.1994, 0.2010)`.
    - For CIFAR-100: The mean used is `(0.5071, 0.4867, 0.4408)` and the standard deviation used is `(0.2675, 0.2565, 0.2761)`.
  - **DataLoaders:** `DataLoader` instances are created for training, validation, and testing with a `batch_size=128` and `num_workers=2`. A `validation_split` of 0.1 (10% of the training data) is used to create the validation set. This ensures that the test set remains untouched during the model selection and compression phases.
  - **Model Training:** The `train_model` function trains the `LayerAnalysisCNN` using the loss function `nn.CrossEntropyLoss` and the optimizer `torch.optim.Adam`. The model is trained for `epochs=30` for CIFAR-10 and `epochs=60` for CIFAR-100, with a `patience=5`

for early stopping based on validation accuracy. The best model state based on validation accuracy is saved.

- **Model Evaluation:** The `evaluate_model` function computes the top-1 accuracy of the model on a given `DataLoader` in evaluation mode (`model.eval()`) without gradient computation.

- **Algorithm 11: Randomized SVD with Power Iteration:** Implemented as described in the preceding section.
- **Layer-to-Matrix Conversion:** `conv_to_matrix` and `linear_to_matrix` implemented as described in the preceding section.
- **Compressed Layer Modules:** Custom PyTorch modules are created to represent the compressed layers:
  - `CompressedConv2D`: This module replaces an original `nn.Conv2d` layer. It uses two sequential `nn.Conv2d` layers to approximate the original convolution via low-rank decomposition. The first `conv1` layer projects input channels to `rank` dimensions with the original kernel size, and its weights are initialized from $V^T$ (reshaped from `Vt`). The second `conv2` layer expands from `rank` dimensions to `out_channels` with a $1 \times 1$ kernel, applying the original stride and handling the bias; its weights are initialized from $U\Sigma$ (reshaped from `U @ np.diag(S)`).
  - `CompressedLinear`: This module replaces an original `nn.Linear` layer. It uses two sequential `nn.Linear` layers. The first `linear1` layer maps `in_features` to `rank` features (initialized from `Vt`), and the second `linear2` layer maps `rank` features to `out_features` (initialized from $U\Sigma$).
  - **Initialization from SVD:** Both compressed modules initialize their weights directly from the $U$, $S$, and $V_t$ components obtained from the randomized SVD. This ensures that the compressed layer immediately approximates the original layer's function.
- **Singular Value Spectrum Analysis:** The `analyze_singular_value_spectrum` function analyzes the singular value spectrum for each layer of the trained model. This analysis helps in understanding the inherent low-rank nature of the layers and informing compression decisions. It computes singular values, normalized singular values, and cumulative energy. It also estimates the effective rank and decay rate for each layer's singular values

## 15.2 Compression Strategy

### 15.2.1 Layer-wise Compression and Fine-tuning Logic

- The `compress_single_layer(model, layer_name, energy_retention, spectrum_analysis)` function orchestrates the compression of a specified layer. It retrieves the original layer's weight and bias, converts the weight to a 2D matrix, applies **Algorithm 11: Randomized SVD with Power Iteration** to obtain $U, S, V_t$ for the specified `energy_retention` level (determining the `rank` using pre-computed `spectrum_analysis`), and then instantiates the appropriate `CompressedConv2D` or `CompressedLinear` module to replace the original layer in the model. It also calculates the layer's compression ratio.

- `fine_tune_compressed_model(model, trainloader, valloader, epochs, lr, patience)`: This function performs a brief fine-tuning step on the compressed model. It uses the same loss function `nn.CrossEntropyLoss` and optimizer `torch.optim.Adam`. Early stopping is applied based on validation accuracy, with a default `patience=3`. This is crucial as low-rank approximation introduces some error, and fine-tuning helps the model adapt to the compressed weights and recover performance.

### 15.2.2 Adaptive Iterative Compression Strategy

The `adaptive_iterative_compression` function implements the core compression pipeline. It operates as follows:

1. **Baseline Accuracy:** It first evaluates the original model's accuracy on the validation set to establish a baseline `original_accuracy`. A `min_accuracy` threshold is then set by subtracting the `max_accuracy_drop` from the `original_accuracy`.
2. **Fixed Initial Compressions:** Based on empirical observations and insights from the previous experiments, `conv1` (early convolutional layer) and `fc3` (final fully connected layer) are treated with fixed, relatively high energy retention levels at the beginning of the process.
    - `conv1` is compressed with 95% energy retention.
    - `fc3` is compressed with 90% energy retention.
    - The model is fine-tuned for 3 epochs after these initial fixed compressions.
3. **Adaptive Layer Compression:** The remaining layers (`conv4`, `conv3`, `conv2`) are compressed iteratively in a predefined order (from later to earlier layers, as found to be more robust in the previous experiments conducted).
    - For each layer, a "coarse search" is performed across a range of `energy_retention` levels (0.99 down to 0.30) using `find_optimal_layer_compression`. This function evaluates the model's validation accuracy after compressing the specific layer and fine-tuning it briefly (3 epochs).
    - The most aggressive `energy_retention` (lowest value) that still keeps the validation accuracy above `min_accuracy` is selected for that layer.
    - The chosen `energy_retention` is applied to the layer, and the entire compressed model is fine-tuned again for 5 epochs.
    - An early stopping mechanism is included: if the `current_accuracy` (validation accuracy) falls below `min_accuracy + 0.5`, the compression process stops to prevent excessive performance degradation.
4. **Final Evaluation:** After all layers are potentially compressed, the overall compression ratio, final validation accuracy, and accuracy drop are calculated and reported.

The `max_accuracy_drop` parameter controls the aggressiveness of the compression strategy. Different `max_accuracy_drop` values lead to different `layer_targets` (minimum and maximum retention levels for `conv2`, `conv3`, `conv4`):

- `max_accuracy_drop <= 1.0`: Conservative compression strategy.
- `max_accuracy_drop <= 3.0`: Moderate compression strategy.

- `max_accuracy_drop <= 5.0`: Aggressive compression strategy.
- `max_accuracy_drop > 6.0`: Very aggressive compression strategy.

### 15.2.3 Experiment Procedure

The `run_improved_compression_experiments` function orchestrates the entire experimental flow.

1. **Dataset Loading and Training:**
   - CIFAR-10 or CIFAR-100 dataset is loaded with a 90/10 train/validation split.
   - A `LayerAnalysisCNN` model is initialized and trained on the training data, with early stopping based on validation accuracy.
   - **CIFAR-10:** Best validation accuracy: 82.56%. Baseline test accuracy: 83.70%. Total parameters: 1,571,466.
   - **CIFAR-100:** Best validation accuracy: 51.62%. Baseline test accuracy: 52.46%. Total parameters: 1,755,876.
2. **Singular Value Spectrum Analysis:**
   - The `analyze_singular_value_spectrum` function is called on the trained model. This pre-computation provides the `energy_cumsum` for each layer, which is crucial for determining the rank based on `energy_retention` during compression. Plots are not saved for these experiments.
3. **Iterative Compression Experiments:**
   - The `adaptive_iterative_compression` function is executed for a range values for the quantity `max_accuracy_drop` thresholds (0%, 1%, 3%, 4%, 5%).
   - For each threshold, the compression pipeline is applied: fixed compression for `conv1` (95% energy) and `fc3` (90% energy), followed by adaptive compression and fine-tuning for `conv4`, `conv3`, and `conv2`.
   - **Crucially, all compression decisions and early stopping during fine-tuning are based only on the validation set to prevent data leakage.**
4. **Final Test Set Evaluation:**
   - **Only after the entire compression process** (including all fine-tuning steps for a given `max_accuracy_drop` threshold) is complete, the final compressed model is evaluated **once** on the unseen test set. This ensures that the reported test accuracy is an unbiased measure of the compressed model's generalization performance.

## 15.3 Results and Observations

The summary table below presents the results for different `max_accuracy_drop` thresholds on CIFAR-10:

**Baseline Test Accuracy for CIFAR-10: 83.70%**

| Acc Drop (%) | Val Orig Acc (%) | Val Final Acc (%) | Test Final Acc (%) | Val Drop (%) | Test Drop (%) | Ratio | Orig Params | Comp Params |
|---|---|---|---|---|---|---|---|---|
| 0.0 | 83.08 | 84.10 | 84.42 | -1.02 | -0.72 | 2.91 | 1,571,466 | 539,291 |
| 1.0 | 83.02 | 82.40 | 82.46 | 0.62 | 1.24 | 6.32 | 1,571,466 | 248,603 |
| 3.0 | 83.16 | 80.52 | 80.79 | 2.64 | 2.91 | 9.32 | 1,571,466 | 168,539 |
| 4.0 | 82.66 | 79.82 | 79.87 | 2.84 | 3.83 | 10.27 | 1,571,466 | 153,051 |
| 5.0 | 82.70 | 79.22 | 80.12 | 3.48 | 3.58 | 10.41 | 1,571,466 | 150,939 |

Table 6: Compression results for CIFAR-10 under different `max_accuracy_drop` thresholds

**Observations for CIFAR-10:**

- **Accuracy Improvement at 0% Drop:** Surprisingly, when targeting a 0% maximum accuracy drop, the validation and test accuracies **increased** (e.g., Validation: 83.08% to 84.10%, Test: 83.70% to 84.42%). This phenomenon, where compression combined with fine-tuning acts as a regularizer, leading to better generalization, has been observed in previous experiments as mentioned in the report. The aggressive `conv4` compression (0.400 energy retention for a 7.88x layer ratio) might contribute to this regularization. This resulted in an overall compression ratio of 2.91x.
- **Moderate Compression (1% Drop)**: Allowing a 1% accuracy drop yields a significantly higher overall compression ratio (6.32x) with a validation accuracy drop of 0.62% and a test accuracy drop of 1.24%. The model still performs very well, indicating a good trade-off between compression and accuracy.
- **Aggressive Compression (3-5% Drop)**: As the allowed accuracy drop increases, the compression ratio significantly improves (up to 10.41x). The actual validation and test accuracy drops are generally within or close to the specified threshold. For example, at a 5% drop tolerance, a 10.41x compression is achieved with a 3.58% test accuracy drop. This shows the effectiveness of the adaptive strategy in pushing for higher compression while attempting to stay within acceptable accuracy bounds.
- Fixed Compressions: The initial fixed compressions on `conv1` (95% energy) and `fc3` (90% energy) are applied consistently across all thresholds. These layers are considered critical for overall model performance.

The summary table below presents the results for different `max_accuracy_drop` thresholds on CIFAR-100:

| Acc Drop (%) | Val Orig Acc (%) | Val Final Acc (%) | Test Final Acc (%) | Val Drop (%) | Test Drop (%) | Ratio | Orig Params | Comp Params |
|---|---|---|---|---|---|---|---|---|
| 0.0 | 51.06 | 52.18 | 53.05 | -1.12 | -0.59 | 4.17 | 1,755,876 | 421,519 |
| 1.0 | 51.46 | 51.00 | 52.43 | 0.46 | 0.03 | 4.81 | 1,755,876 | 365,199 |
| 3.0 | 50.90 | 48.72 | 51.15 | 2.18 | 1.31 | 6.35 | 1,755,876 | 276,495 |
| 4.0 | 50.78 | 48.54 | 49.48 | 2.24 | 2.98 | 6.78 | 1,755,876 | 258,895 |
| 5.0 | 51.52 | 48.90 | 50.29 | 2.62 | 2.17 | 6.69 | 1,755,876 | 262,415 |

Table 7: Compression results for CIFAR-100 under different `max_accuracy_drop` thresholds

**Baseline Test Accuracy for CIFAR-100: 52.46%**

**Observations for CIFAR-100:**

- **Accuracy Improvement at 0% Drop:** Similar to CIFAR-10, an accuracy increase is observed for CIFAR-100 when targeting a 0% maximum drop (Validation: 51.06% to 52.18%, Test: 52.46% to 53.05%). This again supports the regularization effect of compression and fine-tuning. A 4.17x compression ratio is achieved.

- **Moderate Compression (1% Drop):** For a 1% drop tolerance, a 4.81x compression is achieved with minimal impact on test accuracy (0.03% drop). This is a highly favorable outcome, demonstrating that significant compression can be achieved almost without sacrificing performance on a more complex dataset.
- **Aggressive Compression (3-5% Drop)**: Higher compression ratios (up to 6.78x) are achieved for larger accuracy drop tolerances. The test accuracy drops are generally well-controlled and fall within or near the specified thresholds.
- **Lower Compression Ratios Compared to CIFAR-10:** For similar `max_accuracy_drop` settings, the overall compression ratios achieved on CIFAR-100 are generally lower than on CIFAR-10 (e.g., 6.78x vs. 10.41x for 4-5% drop). This is expected because CIFAR-100 is a more complex dataset with 100 classes, requiring the model to retain more information (and thus more parameters) to maintain performance. The layers for CIFAR-100 might have less redundancy to exploit for compression.
- **Consistency of Fixed Compressions:** As with CIFAR-10, `conv1` (95% energy) and `fc3` (90% energy) are consistently compressed first.

# 16 Compression of Regression Model

This section details Experiment 7 (implemented in "Experiment_7.ipynb"), focusing on applying randomized Singular Value Decomposition (SVD) algorithms to compress the weight matrices of a feedforward neural network (`LayerAnalysisRegressor`) used for regression. The experiment explores an adaptive iterative compression strategy and evaluates its impact on performance metrics like $R^2$ score, RMSE, and MAE, alongside the achieved compression ratios. The study draws inspiration from the theoretical framework of randomized SVD and previous neural network compression experiments.

## 16.1 Implementation Details

The core of this experiment involves implementing and integrating several key components:

### 16.1.1 Model Architecture

The `LayerAnalysisRegressor` is a feedforward neural network designed for regression tasks. It consists of five fully connected (FC) layers, with dropout layers applied after the first three to prevent overfitting:

- **Input Layer:** `input_dim` neurons.
- **Fully Connected Layer 1 (fc1):** `input_dim` $\rightarrow$ 256 neurons with ReLU activation.
- **Fully Connected Layer 2 (fc2):** 256 $\rightarrow$ 128 neurons with ReLU activation.
- **Fully Connected Layer 3 (fc3):** 128 $\rightarrow$ 64 neurons with ReLU activation.
- **Fully Connected Layer 4 (fc4):** 64 $\rightarrow$ 32 neurons with ReLU activation.
- **Fully Connected Layer 5 (fc5 - Output):** 32 $\rightarrow$ `output_dim` (default 1) neuron.

The model also maintains an `OrderedDict (self.layer_info)` to store references to each layer, its type (`'fc'`), and a qualitative position (`'early'`, `'middle'`, `'final'`) to facilitate systematic layer-wise processing and analysis.

The algorithm used is again `algorithm_11_randomized_svd_power_iteration`, again implemented similarly as in the preceding sections.

### 16.1.2 Compressed Linear Layer Module

A custom PyTorch module, `CompressedLinear`, is implemented to replace original `nn.Linear` layers with their low-rank approximations. The low-rank approximation of a weight matrix $W \approx U\Sigma V^T$ is implemented as two sequential linear layers:

- A first linear layer (`linear1`) that maps `in_features` to `rank` features. Its weight is initialized from $V^T$.
- A second linear layer (`linear2`) that maps `rank` features to `out_features`. Its weight is initialized from $U\Sigma$. This layer also handles the bias from the original layer.

The `_initialize_from_svd` helper method sets the weights of `linear1` and `linear2` directly from the SVD components, ensuring that the compressed layer immediately approximates the original layer's function.

### 16.1.3 Helper Functions

- `linear_to_matrix(weight)`: Converts a linear layer's weight tensor to its 2D NumPy array form suitable for SVD.
- `calculate_total_parameters(model)`: sCalculates the total number of trainable parameters in a PyTorch model.
- `comprehensive_evaluate_model(model, dataloader, scaler_y=None)`: Evaluates the model performance using multiple metrics ($R^2$ score, RMSE, MAE). It supports inverse transformation of predictions if targets were standardized during training.
- `train_model(...)`: Trains the regression model with an Adam optimizer and MSELoss, incorporating early stopping based on validation $R^2$ to prevent overfitting.
- `fine_tune_compressed_model(...)`: Performs a brief fine-tuning step on the compressed model. This is crucial for recovering performance lost during compression and is similar to `train_model` but typically uses a smaller learning rate and fewer epochs.
- `get_optimal_rank(singular_values, energy_retention, min_rank=1)`: Finds the target rank for SVD based on a specified energy retention level. It calculates the cumulative sum of squared singular values and finds the smallest rank that retains the desired percentage of total energy.
- `compress_single_layer(model, layer_name, energy_retention, spectrum_analysis)`: Compresses a single layer in the model using the randomised power iteration algorithm to the rank determined by the `energy_retention` level. It then replaces the original layer with a `CompressedLinear` module.
- `analyze_singular_value_spectrum(model, save_plots=True)`: Analyzes and plots the singular value spectrum (normalized singular values, cumulative energy) for each layer in the model. It also calculates and prints statistics like decay rate and effective rank for various energy retention levels.
- `find_optimal_layer_compression(...)`: Conducts a coarse search to find the most aggressive compression (lowest energy retention) for a given layer that maintains an $R^2$ score above a specified minimum threshold after fine-tuning. This function is key to the adaptive compression strategy.
- `adaptive_iterative_compression(model, spectrum_analysis, trainloader, valloader, max_r2_drop)`: Implements the adaptive compression strategy. It initially applies fixed compressions to `fc1` (95% energy retention) and `fc5` (90% energy retention), then iteratively compresses `fc4`, `fc3`, and `fc2` based on `max_r2_drop` (conservative, moderate, aggressive, or very aggressive strategies). After each layer's compression, the entire model is fine-tuned to recover accuracy.
- `compare_compression_levels(results_list, baseline_metrics)`: Performs t-tests to determine if the $R^2$ scores of compressed models are significantly different from the baseline, providing statistical validation for the compression levels.
- `plot_enhanced_compression_curves(results_summary)`: Generates scatter plots to visualize the $R^2$ drop, final $R^2$ score, final RMSE, and a Pareto frontier of $R^2$ vs. compression ratio, allowing for a comprehensive analysis of the compression trade-offs.

### 16.1.4 Dataset Loading and Preprocessing

The `load_regression_dataset` function supports loading `'california_housing'` and `'diabetes'` datasets. It performs standard scaling on features (`X`) and optionally on targets (`y`) using `StandardScaler`. The data is then split into training, validation, and testing sets, and PyTorch `DataLoader`s are created. For the California Housing dataset used in this experiment, targets were standardized.

## 16.2 Experimental Setup

The experiment was structured as follows:

1. **Dataset:** The California Housing dataset was used, with features and targets standardized.
2. **Model:** A `LayerAnalysisRegressor` model was initialized with the appropriate input and output dimensions.
3. **Baseline Training:** The uncompressed model was trained for 100 epochs with early stopping (patience=15) on the training set, evaluated on a validation set. The final performance was then reported on the test set.
    - **Baseline Test Performance ($R^2$):** 0.8214
    - **Baseline Test Performance (RMSE):** 0.4860
    - **Baseline Test Performance (MAE):** 0.3304
4. **Singular Value Spectrum Analysis:** The singular value spectrum for each layer of the trained baseline model was analyzed and plotted to understand their inherent low-rank nature and decay characteristics.
5. **Adaptive Iterative Compression Study:** The `adaptive_iterative_compression` function was run with different `max_r2_drop` thresholds (0.00, 0.02, 0.03, 0.05, 0.08) to simulate various compression aggressiveness levels.
    - For each `max_r2_drop` level, `n_trials` (set to 10) independent compression trials were conducted.
    - In each trial:
        - A deep copy of the baseline model was used.
        - Fixed compressions were applied to `fc1` (95% energy retention) and `fc5` (90% energy retention), followed by a fine-tuning step.
        - Then, `fc4`, `fc3`, and `fc2` were iteratively compressed. For each of these layers, `find_optimal_layer_compression` was used to identify the most aggressive compression that kept the current validation $R^2$ above `min_r2` (original $R^2$ - `max_r2_drop`). After compressing each layer, the entire model underwent another fine-tuning phase.
        - Finally, the compressed model's performance was evaluated on the test set.
6. **Statistical Analysis and Plotting:** The results from all trials and compression levels were aggregated, and statistical tests (t-tests against baseline $R^2$) were performed. Enhanced compression curves were plotted to visualize the trade-offs.

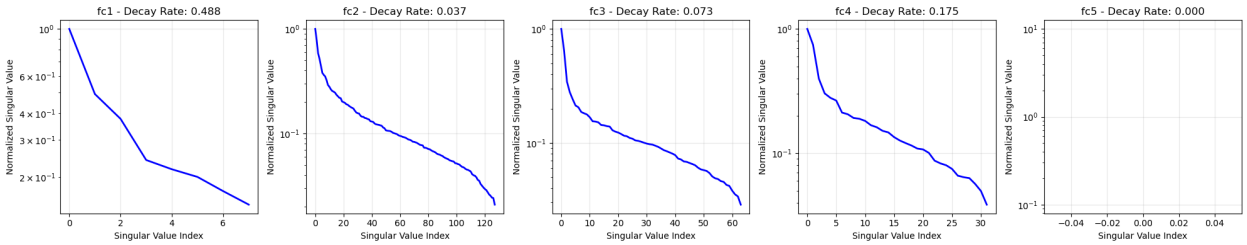



Figure 18: Singular values for the layers

## 16.3 Observations and Results

### 16.3.1 Singular Value Spectrum Analysis

The singular value spectrum analysis for the baseline `LayerAnalysisRegressor` model revealed the following decay rates and effective ranks for 99% energy retention:

- `fc1`: Decay rate: 0.488, Effective rank: 8, Rank for 99% energy: 8
- `fc2`: Decay rate: 0.037, Effective rank: 128, Rank for 99% energy: 99
- `fc3`: Decay rate: 0.073, Effective rank: 64, Rank for 99% energy: 52
- `fc4`: Decay rate: 0.175, Effective rank: 32, Rank for 99% energy: 26
- `fc5`: Decay rate: 0.000, Effective rank: 1, Rank for 99% energy: 1

**Observation:** `fc5` (output layer) has a very low effective rank and rapid singular value decay, indicating it is highly compressible. `fc1` also shows a relatively fast decay. `fc2`, `fc3`, and `fc4` exhibit slower decay rates, suggesting they might be more challenging to compress aggressively without significant performance loss, hence the need for power iteration and adaptive strategies. The same is shown in the image 18.

### 16.3.2 Adaptive Iterative Compression Results

The comprehensive study evaluated different `max_r2_drop` thresholds. The key metrics reported were Mean Test $R^2$, Standard Deviation of $R^2$, Mean Test RMSE, and Mean Compression Ratio.

| Max $R^2$ Drop | Mean $R^2$ | $R^2$ Std | Mean RMSE | Compression (x) | Significant? |
|---|---|---|---|---|---|
| 0.000 | 0.8063 | 0.0008 | 0.5061 | 1.03 | Yes* |
| 0.020 | 0.8106 | 0.0022 | 0.5004 | 4.63 | Yes* |
| 0.030 | 0.8109 | 0.0015 | 0.5000 | 5.53 | Yes* |
| 0.050 | 0.8072 | 0.0013 | 0.5049 | 9.15 | Yes* |
| 0.080 | 0.7939 | 0.0008 | 0.5220 | 11.62 | Yes* |

Table 8: Adaptive compression results across various `max_r2_drop` thresholds.

*Baseline (No compression):* $R^2 = 0.8214$, RMSE = 0.4860
* *Significantly worse than baseline* ($p < 0.05$)

### 16.3.3 Observations

1. **Consistent $R^2$ Drop:** All tested compression levels (even the most conservative with `max_r2_drop=0.000`) resulted in a statistically significant drop in $R^2$ compared to the uncompressed baseline model on the test set. This suggests that for this specific regression task and model architecture, compression using this method inherently leads to some performance degradation, even with adaptive strategies and fine-tuning.

2. **Trade-off between Compression Ratio and Performance:**

   - **Low Compression Ratios (1.03x):** The most conservative strategy with the maximum $R^2$ drop being zero (`max_r2_drop=0.000`) achieved a minimal compression ratio of 1.03x, indicating very slight parameter reduction. The mean $R^2$ remained relatively high (0.8063) but was still significantly lower than the baseline. This aligns with the "CONSERVATIVE" strategy in the code, which aims for higher energy retention.
   - **Moderate Compression Ratios (4.63x – 5.53x):** With `max_r2_drop` at 0.020 and 0.030, the model achieved significantly higher compression (4.63x and 5.53x, respectively). Interestingly, the mean $R^2$ for these levels (0.8106 and 0.8109) was higher than the $R^2$ achieved at 1.03x compression. This counter-intuitive result could imply that some initial aggressive pruning acts as a form of regularization, leading to slightly better generalization on the test set within this range, despite being statistically worse than the uncompressed baseline.
   - **High Compression Ratios (9.15x - 11.62x)**: As `max_r2_drop` increased to 0.050 and 0.080, the compression ratios became much higher (9.15x and 11.62x). However, this came with a more pronounced drop in $R^2$ (0.8072 and 0.7939, respectively), though the magnitude of the drop was not as severe as one might expect for such high compression, thanks to the adaptive fine-tuning.

3. **Effectiveness of Adaptive Strategy and Fine-tuning:** Even with the consistent $R^2$ drop, the adaptive compression strategy, combined with iterative fine-tuning after each layer's compression, allowed the model to achieve substantial parameter reduction (up to 11.62x overall) while maintaining $R^2$ scores that are still reasonably close to the original baseline. This highlights the crucial role of the fine-tuning steps in recovering lost performance.

4. **Fc1 and Fc5 Compression:** The strategy began by compressing `fc1` (95% energy retention) and `fc5` (90% energy retention). `fc5` is the output layer, and its low effective rank from the spectrum analysis confirms it can be heavily compressed without major impact. `fc1` is the first hidden layer, and its more conservative compression target (95%) aligns with the sensitivity of early layers often observed in CNNs (though this is an MLP, similar principles can apply to feature extraction layers).

5. **Iterative Compression of Middle Layers:** The iterative compression of `fc4`, `fc3`, and `fc2` (in that order) based on validation $R^2$ allowed for a nuanced approach. The helper function `find_optimal_layer_compression` enables finding the most aggressive compression for each layer that still meets the performance threshold, ensuring that the model doesn't fall below the acceptable $R^2$ until necessary.

## 16.4 Conclusions

1. **Feasibility of Compression for Regression Models:** The study confirms that neural networks for regression can be significantly compressed using low-rank approximation techniques. Overall compression ratios of up to 11.62x were achieved.

2. **Trade-off and Regularization:** While all compression levels resulted in a statistically significant $R^2$ drop compared to the uncompressed baseline, the magnitude of the drop was often small, especially at moderate to high compression ratios (e.g., 4.63x to 5.53x). The slight improvement in $R^2$ at these levels compared to the most conservative compression suggests that judicious pruning via SVD and subsequent fine-tuning may have a regularization effect, leading to a more robust model.

3. **Crucial Role of Fine-tuning:** The adaptive iterative strategy, which includes fine-tuning after each compression step, is essential. It allows the model to adapt to the modified weights and recover a substantial portion of the lost performance, making aggressive compression viable.

4. **Adaptive Strategy Effectiveness:** The adaptive approach, by setting $R^2$ drop thresholds and iteratively compressing layers, provides a practical method for balancing compression ratio and performance. It allows for more aggressive compression of less sensitive layers while preserving critical ones.

5. **Layer Sensitivity in Regression Models:** Similar to classification CNNs, different layers in the regression MLP exhibited varying singular value decay characteristics, influencing their compressibility. The `fc5` (output) layer was highly compressible, while middle layers (`fc2`, `fc3`, `fc4`) required more careful compression.

## Future Developments

This report demonstrates the promising potential of randomized SVD for neural network compression, particularly when combined with strategic fine-tuning. However, several avenues for future research and development can further enhance the effectiveness, efficiency, and applicability of these techniques:

- **Automated Layer-wise Rank Selection:** Current methods often rely on predefined energy retention thresholds or manual experimentation to determine the optimal compression rank for each layer. Future work should focus on developing automated, data-driven approaches for layer-wise rank selection. This could involve:
  - Reinforcement learning agents that learn optimal compression policies based on model performance and compression ratio targets.
  - Bayesian optimization or evolutionary algorithms to search for the ideal rank distribution across layers.
  - Incorporating sensitivity analysis directly into the rank selection process to dynamically allocate more capacity to critical layers and aggressively prune robust ones.
- **Progressive Compression Strategies:** Instead of compressing a pre-trained model in a single step, progressive compression gradually reduces the rank of layers throughout the training process. This could involve:
  - Starting with a full-rank model and incrementally reducing ranks over epochs, allowing the network to adapt more smoothly to the reduced capacity.
  - Integrating SVD-based compression directly into the optimization loop, potentially as a regularizer.
  - Exploring differentiable compression techniques that allow the rank and decomposition to be jointly optimized with the network weights.
- **Architecture-Aware Compression Methods:** The effectiveness of compression can vary significantly across different neural network architectures (e.g., CNNs, RNNs, Transformers) and even within different types of layers (e.g., convolutional vs. linear). Future research should explore:
  - Developing specialized randomized SVD variants or tensor decomposition methods that inherently exploit the unique structural properties of different layer types and network architectures.
  - Investigating how specific architectural components (e.g., skip connections, attention mechanisms) interact with compression and devising strategies that maintain their functional integrity.
  - Tailoring compression for emerging architectures like Vision Transformers or Graph Neural Networks.
- **Advanced Tensor Reshaping for Convolutional Layers:** The current approach of reshaping 4D convolutional weight tensors into 2D matrices for SVD may not capture all structural information. Future developments should explore:

  - Directly applying higher-order tensor decomposition (e.g., Tucker decomposition, Tensor Train decomposition) to convolutional kernels, which can preserve more of the inherent multi-dimensional structure.
  - Investigating novel matrix unfolding techniques that are more amenable to low-rank approximation while retaining spatial and channel correlations.
  - Developing hybrid methods that combine matrix SVD with other tensor-aware compression techniques.

- **Theoretical Underpinnings of Regularization Effect:** The observation that compression, particularly for certain layers, can lead to accuracy improvements after fine-tuning suggests a regularization effect. Future work should include:
  - A deeper theoretical analysis to rigorously explain this phenomenon and quantify its benefits.
  - Developing new theoretical bounds that account for the regularization properties of low-rank approximations in neural networks.
  - Investigating the interplay between model capacity, data complexity, and the extent of this regularization.

- **Hardware-Aware Compression and Deployment:** To facilitate real-world deployment, future research should consider the computational and memory constraints of target hardware platforms. This involves:
  - Developing compression algorithms that are optimized for specific hardware accelerators (e.g., TPUs, FPGAs, edge AI devices).
  - Exploring methods for efficient inference with compressed models, including specialized runtime environments and compilers.
  - Quantifying the energy efficiency and latency improvements of compressed models on real-world hardware.

- **Robustness to Adversarial Attacks and Domain Shift:** Investigating how low-rank compression affects the robustness of neural networks to adversarial attacks and their generalization performance under domain shift conditions.