Kevin Pan (kp639)

# ECE5545 Assignment 4

## 1) Convolutions (conv2d.py)

### im2col Implementation

- Given input x of shape (H, W) and kernel k of shape (Hk, Wk):
- Slide the kernel over the input, extract each patch, and flatten it into a column
- Stack all columns to form a 2D matrix of shape (HkWk, H_outW_out)
- Flatten the kernel to a vector of shape (HkWk,)
- Perform matrix multiplication and reshape the result to (H_out, W_out)
- Add bias

### Winograd Implementation

- For each 4x4 patch of the input, apply the Winograd input transformation
- Transform the 3x3 kernel using the Winograd kernel transformation
- Multiply the transformed patch and kernel element-wise
- Apply the Winograd output transformation to get a 2x2 output tile
- Place each output tile in the correct position to build the full output
- Add bias

### FFT Implementation

**1) Kernel Flipping**: The kernel is flipped both vertically and horizontally to match the mathematical definition of convolution

**2) Padding**: Both the input and the flipped kernel are zero-padded so that their shapes match the size required for a full linear convolution. Specifically, the padded size is the sum of the input and kernel dimensions minus one in each direction

**3) FFT Transformation**: The 2D FFT is computed for both the padded input and the padded, flipped kernel. This converts both to the frequency domain.

**4) Element-wise Multiplication**: The transformed input and kernel are multiplied element-wise in the frequency domain, which is equivalent to convolution in the spatial domain.

**5) Inverse FFT**: The inverse 2D FFT is applied to the product, converting the result back to the spatial domain.

**6) Cropping**: The output is cropped to the "valid" region, which matches the output size of a standard convolution

**7) Bias Addition**: Add bias to the output

# 2) Matrix Multiplications (matmul.py)

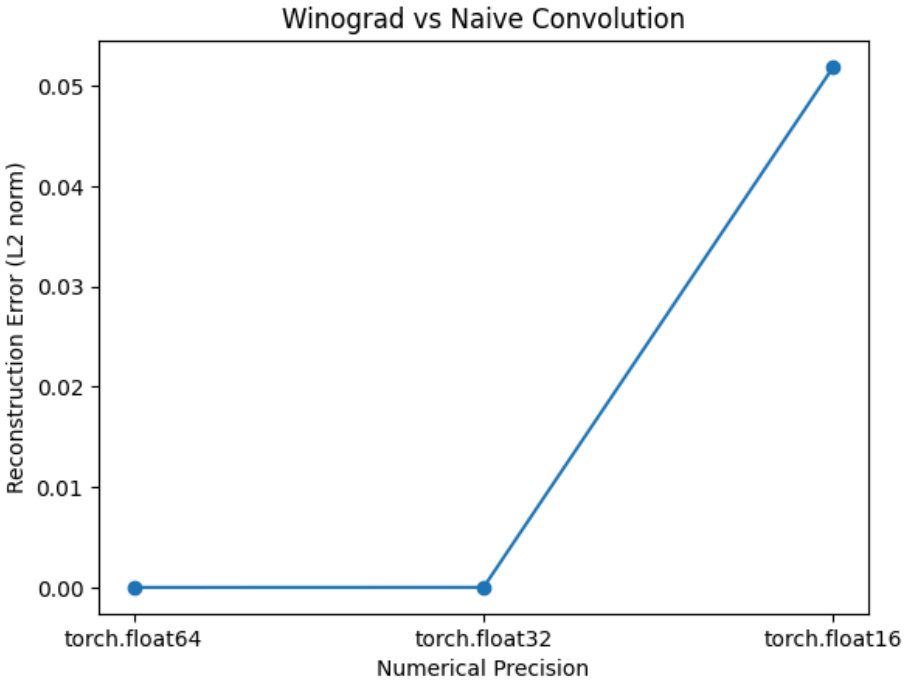
### SVD Implementation

- Decompose each input matrix using SVD
- If a rank is specified, truncate the SVD to keep only the top singular values and vectors, creating a low-rank approximation of the matrix
- Multiply the approximations of A and B together to get the result

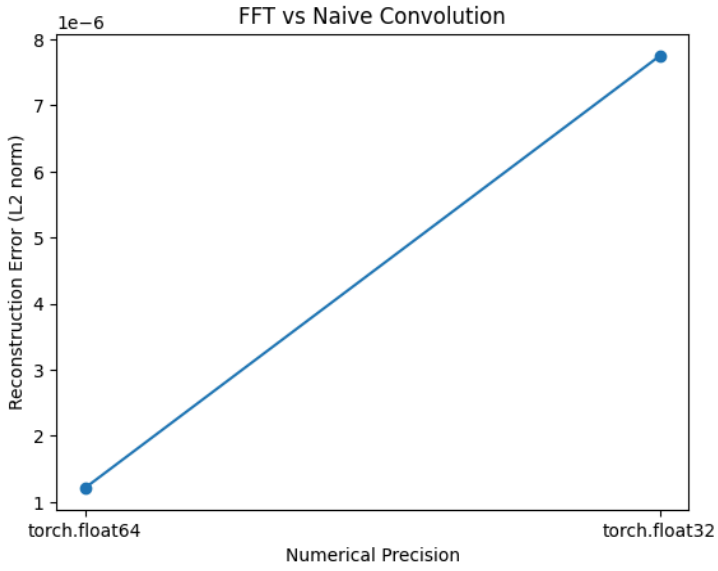
### LogMatMul Implementation

- Each matrix element is split into its sign and the logarithm of its absolute value
- Matrix multiplication is performed in the log domain by adding logs and then exponentiating
- The correct sign for each term is tracked and restored after exponentiation
- The final result is the sum over the shared dimension, matching standard matrix multiplication but using log-domain arithmetic

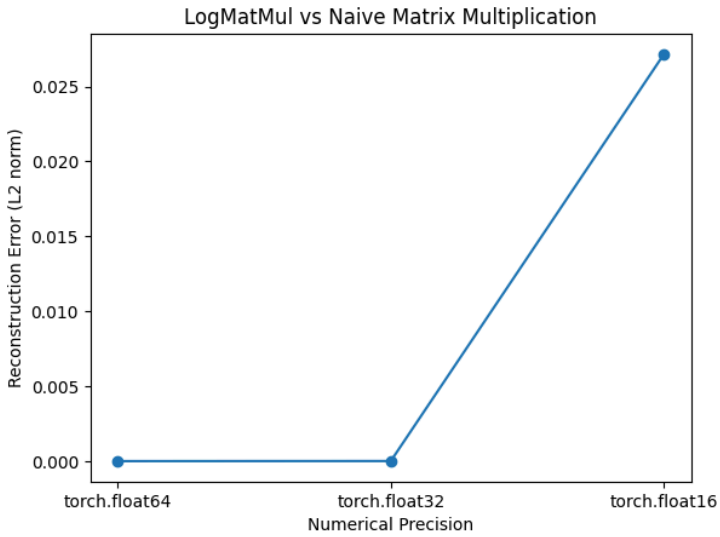
# 3) Numerical Precision




The plot shows that the reconstruction error between Winograd and the naive convolution is negligible for float64 and float32 precision, but increases noticeably for float16. This indicates that Winograd is numerically stable at standard precisions, but suffers from increased rounding errors at lower precision.

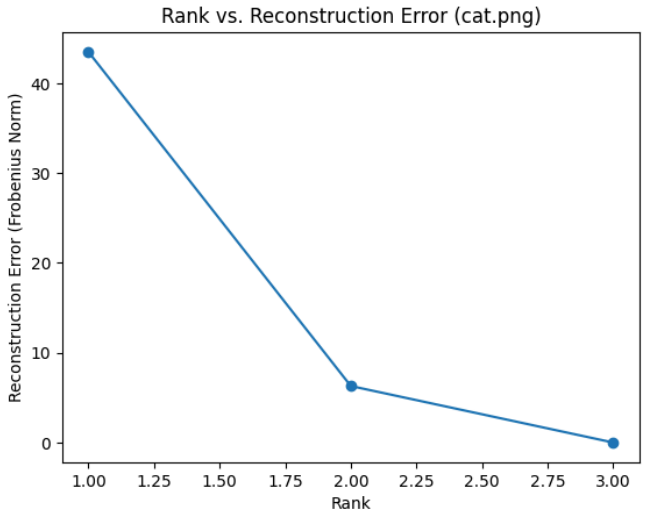



The plot shows that the reconstruction error between FFT-based convolution and the naive implementation is extremely low for both float64 and float32 precisions, indicating that FFT convolution is numerically stable at these precisions. The error is slightly higher for float32 than float64, as expected, but remains negligible. FFT-based convolution cannot be evaluated at float16 precision due to limitations in PyTorch's FFT implementation.

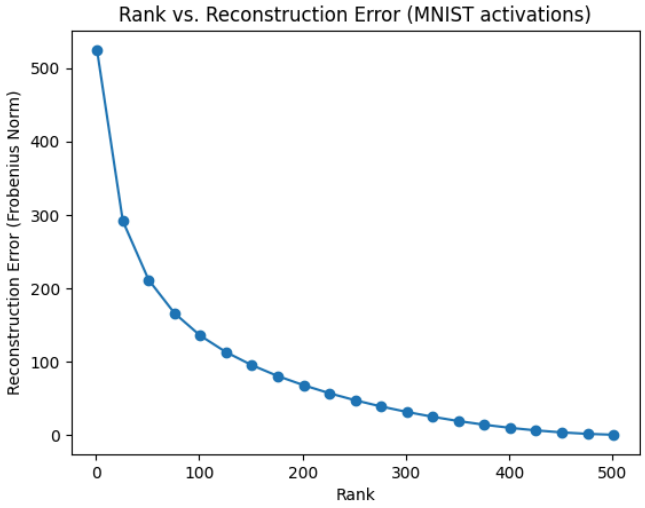



The plot shows that the reconstruction error between log-domain matrix multiplication and the naive method is negligible for float64 and float32, but increases sharply for float16. This indicates that logmatmul is numerically stable at standard precisions, but suffers from significant accuracy loss at lower precision, likely due to the limited representational range and increased rounding errors in half precision.

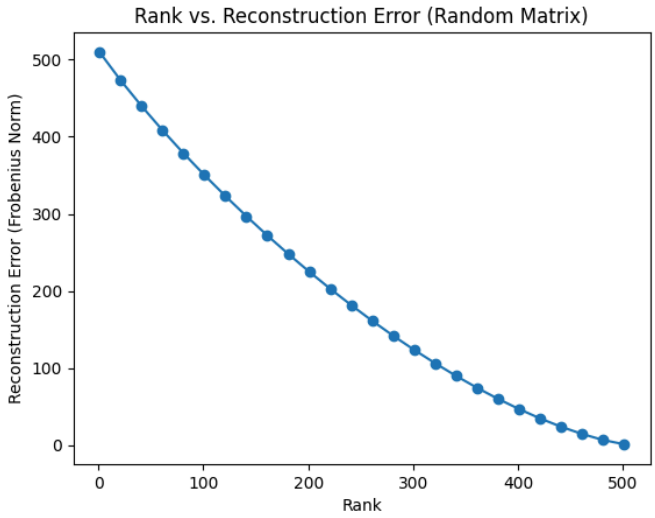
# 4) SVD Rank Analysis



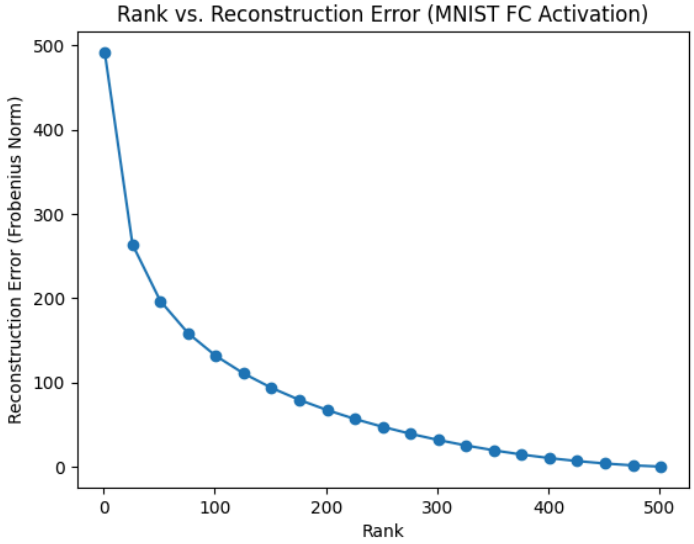

The reconstruction error decreases rapidly as the rank increases from 1 to 3. Most of the error is eliminated by including just the first two singular values, showing that the error drops quickly at low ranks. This is because most of the information is captured by the leading singular values. There is a clear "elbow" in the curve at rank 2. Below this, the error increases significantly, while above it, improvements are marginal. This data indicates that the matrix can be well approximated by a low-rank representation, and that most of the information is captured by the leading singular values.



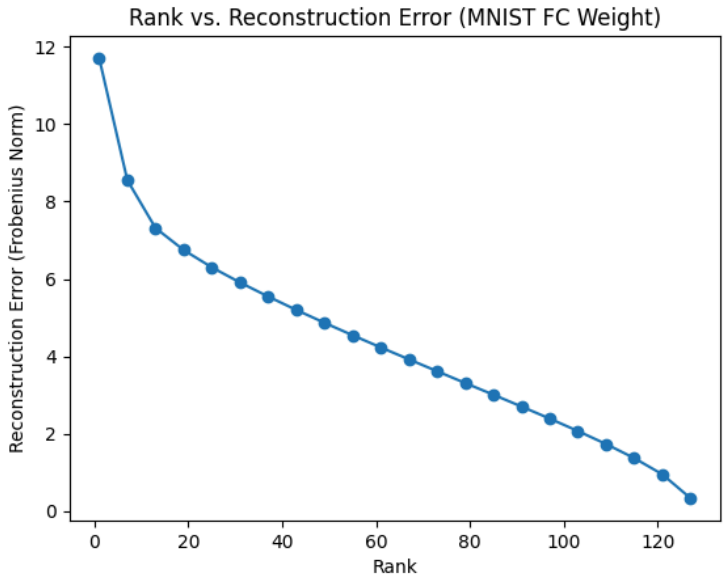

The reconstruction error decreases rapidly as the rank increases. This sharp initial drop indicates that the MNIST activation matrix is highly compressible, as the leading singular values capture most of the important information. After a certain point (around rank 100), further increases in rank yield only marginal improvements. Overall, this suggests that the data can be well-approximated by a low-rank matrix, enabling significant dimensionality reduction with minimal loss of information.



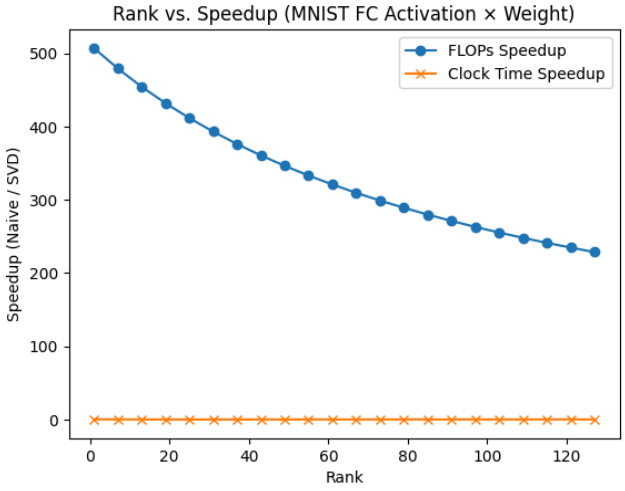

For the random matrix, the reconstruction error decreases steadily as the rank increases, but unlike structured data, the curve is very smooth. The error reduction is gradual and nearly linear, indicating that the singular values are more evenly distributed and no small subset dominates. This means random matrices are not easily compressible, and a high rank is required to achieve a low reconstruction error. As a result, low-rank SVD approximations are less effective for random data, since most of the information is spread across many singular values.




The reconstruction error for the MNIST fully connected activation matrix decreases rapidly as the rank increases, with a sharp drop at low ranks. This indicates that most of the information in the activations is captured by a relatively small number of principal components, reflecting the network's ability to learn compact, low-dimensional representations.

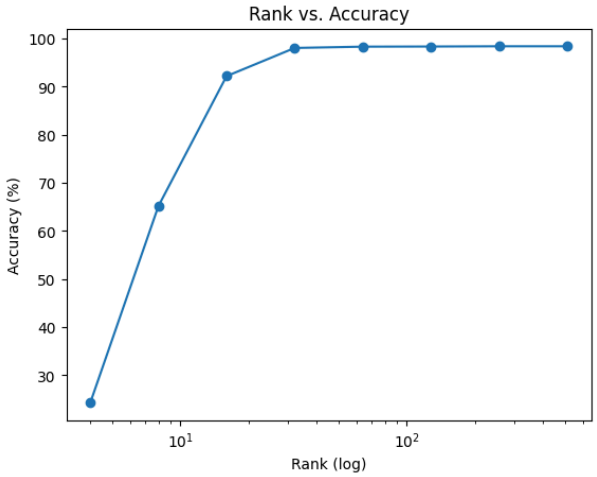


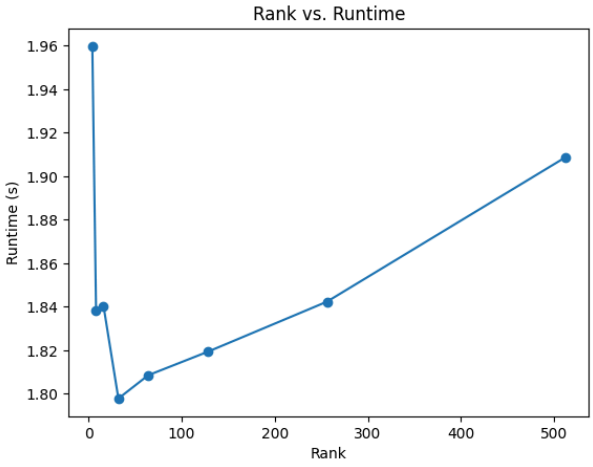

The reconstruction error for the MNIST fully connected layer's weight matrix decreases rapidly as the rank increases, with a steep drop at low ranks and a more gradual decline after. This indicates that the weight matrix is highly structured and much of its information is captured by the leading singular values. This suggests that a low-rank approximation can represent the matrix well, while further increases in rank yield diminishing returns. This highlights the potential for compressing neural network weights with minimal loss of information.

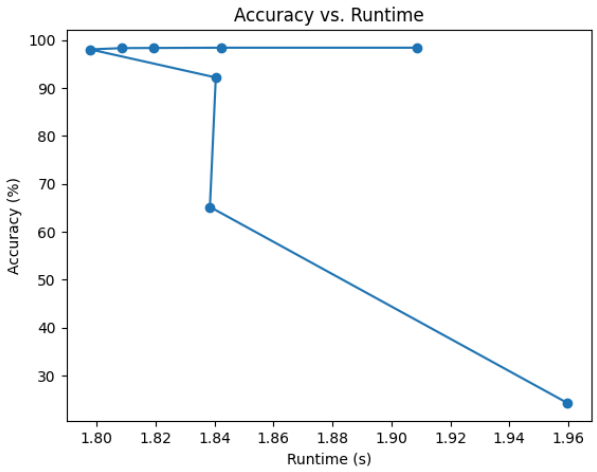



The plot shows that the theoretical speedup from using SVD-based multiplication is highest at low ranks and decreases as the rank increases, reflecting the reduced computational cost of low-rank approximations. However, the actual clock time speedup is minimal and nearly flat, indicating that in practice, the overhead of SVD computation and Python execution outweighs the theoretical gains for these matrix sizes. This highlights the difference between theoretical and practical speedup, and suggests that SVD-based methods are most beneficial for very large matrices or when the SVD can be reused multiple times.

# 5) SVD in Neural Network

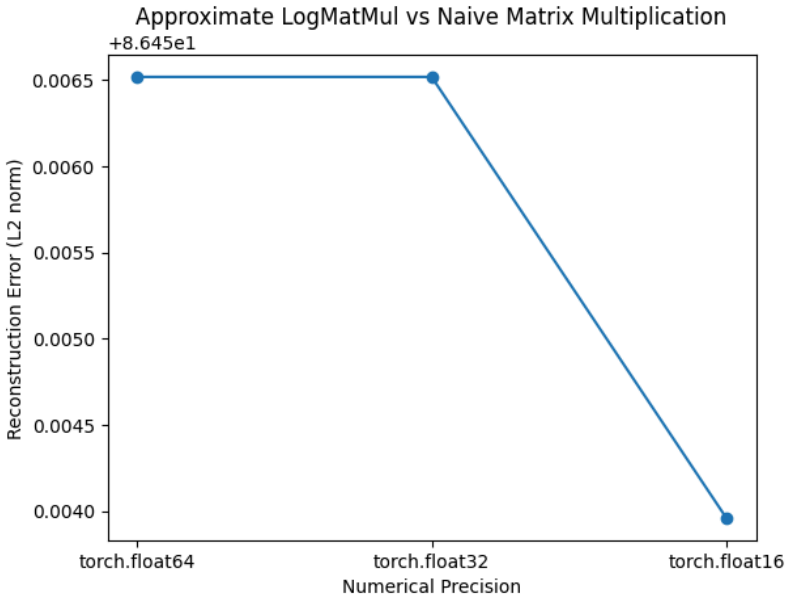



The plots show that as the SVD rank increases, model accuracy improves rapidly and plateaus near the original performance. This trend indicates that most of the network's predictive power is retained even at moderate ranks. The runtime remains nearly constant across all ranks, suggesting that SVD compression does not significantly speed up inference for this model size and hardware. The "Accuracy vs. Runtime" plot further highlights that substantial compression (low rank) leads to a sharp drop in accuracy, but once a moderate rank is reached, accuracy saturates with little to no runtime penalty. Overall, SVD compression enables significant model size reduction with minimal loss in accuracy, but practical speedup is limited in this setting.

# 6) Extra Credit




## Analysis

The plot demonstrates that NVidia's approximate log-domain addition yields very low reconstruction error across all tested precisions, with only a minor loss in accuracy compared to the exact logmatmul.
This validates the effectiveness of the hardware-friendly approximation for practical matrix multiplication tasks.

## Approximate LogMatMul Implementation (NVidia)

To efficiently perform matrix multiplication in the log domain, we implemented NVidia's patented approximate log-addition. Instead of the exact log-sum-exp, we use the hardware-friendly formula:

$\log(a+b) \approx \max(\log a, \log b) + \mathrm{ReLU}(C - |\log a - \log b|)$, where $C = \log(2)$.

This approximation is applied iteratively when summing terms in the log domain. Our implementation decomposes each matrix element into sign and log-absolute value, accumulates products using the approximate log-add, and restores the sign at the end. This approach yields very low reconstruction error across numerical precisions, while being much more efficient for hardware or large-scale computation.