Reinforcement Learning for Adaptive Column Selection in Low-Rank Matrix Approximations

Arnav Deshmukh (2023101114)
Shreyas Mehta (2023101059)
Nidhish Jain (2023101071)

Mid-Evaluation Presentation

Overview

What's Done (Part 1)

- Enhanced randomized SVD method to match the current benchmarks.
- Added more dimensions to the current RL state to get more inferences.
- Added a new RL method A2C to compare it with the given DQN Method.
- Generated matrices of different sizes and trained DQN and A2C networks on them (low-rank noise added matrix).
  - Sizes: 100x80, 200x160, 400x320, 800x640
- Trained on different matrix types (size: 200x160) and saved DQN/A2C networks:
  - Types: low_rank_noise, exp_decay, clustered, ill_conditioned

What's Done (Part 2)

- Tested trained networks on newly generated matrices of each type.
    - Evaluated Frobenius norm error and explored alternative error metrics.
- Incorporated adaptive column selection and CUR decomposition method.
- Created a generalized robust model that handles both varying types and sizes of matrices.
- Currently working on parsing and analyzing Florida state-generated matrices.

Rank Approximation Methods

- **Deterministic SVD**:
  - Classical approach using full Singular Value Decomposition
  - Retains top-k singular components
- **Randomized SVD**:
  - Projects original matrix onto a low-dimensional subspace
  - Uses oversampling and QR decomposition
- **RL-based Column Selection (Epsilon-Greedy)**:
  - Selects columns sequentially using an exploration-exploitation tradeoff
  - Uses pseudo-inverse projection to approximate the original matrix

Baseline Performance Comparison

- **Dataset**:
  - Synthetic matrix of size 100×80
  - True rank $= 10$; noisy perturbations added
- **Results (Approx. Rank = 10)**:
  - **Deterministic SVD**: Error $= 0.7957$, Time $= 0.0237$s
  - **Randomized SVD**: Error $= 1.5565$, Time $= 0.0024$s
  - **RL Column Selection**: Error $= 2.0825$, Time $= 0.3120$s
- **Tradeoffs**:
  - Randomized method is fastest but less accurate
  - RL-based method is adaptive but slower

Deterministic SVD Approach

- **Goal**: Approximate the matrix A by keeping only the top-k singular values and vectors.
- **Steps**:
  - Perform full SVD: $A = U\Sigma V^T$
  - Truncate to rank-k: Keep only the first k singular values and vectors
  - Reconstruct: $A_r = U_k \Sigma_k V_k^T$
- **Advantages**:
  - High accuracy (optimal in Frobenius norm sense)
  - Simple and deterministic
- **Limitations**:
  - Computationally expensive for large matrices

Randomized SVD Approach

- **Goal**: Speed up SVD by working with a lower-dimensional approximation.
- **Steps**:
  - Generate a random matrix G of size $n \times (k + p)$
  - Form $Y = AG$ to project A to a smaller subspace
  - Perform QR decomposition: $Y = QR$
  - Project: $B = Q^T A$
  - Do SVD on B: $B = \hat{U}\Sigma V^T$
  - Reconstruct: $A_r = Q\hat{U}_k\Sigma_k V_k^T$
- **Advantages**:
  - Much faster for large matrices
  - Good accuracy with oversampling
- **Limitations**:
  - Accuracy depends on randomness and oversampling

RL-based Column Selection (Epsilon-Greedy)

- **Goal**: Select a subset of columns to approximate A using projection.
- **Steps**:
  - Initialize empty set of selected columns
  - Repeat until k columns are selected:
    - With probability ε, pick a random column
    - Otherwise, pick the column that minimizes reconstruction error
  - At each step, compute $A_r = A_c(A_c^\dagger A)$ using pseudo-inverse
- **Advantages**:
  - Adaptive to the matrix structure
  - Mimics reinforcement learning logic
- **Limitations**:
  - Slower than SVD methods
  - Greedy heuristic may get stuck in suboptimal solutions

Training Process

- **Environment Setup**:
  - State: Binary mask of selected columns
  - Actions: Choose next column (with constraints)
  - Rewards: Reduction in Frobenius norm error
  - Termination: Reached target rank or invalid action
- **Agent Architecture (Full DQN)**:
  - Q-network with 3 layers: Input $\rightarrow$ Hidden (ReLU) $\rightarrow$ Output
  - Experience Replay: Sampled mini-batches to stabilize learning
  - Target Network: Periodically updated for stable Q-value estimation
  - Action Masking: Prevents reselection of already chosen columns
- **Training Loop Highlights**:
  - Epsilon-greedy exploration with decay
  - Reward shaping via error reduction
  - Episode statistics show convergence and stability

Comparison of Methods

- **Original RL Approach**: No neural network, relied on direct greedy updates.
- **Issue**: Poor generalization, limited policy representation.
- **Solution**: Discussed with professor and transitioned to DQN-based RL.

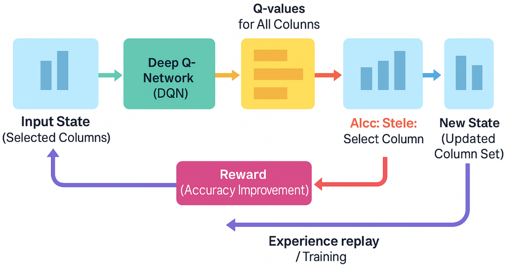
Results (on low-rank matrix with rank 10):

Method	Error (Frobenius)	Time (sec)
Deterministic SVD	0.7957	0.0016
Randomized SVD	1.5565	0.0004
RL-based (Greedy/Pseudo-RL)	2.0825	0.3120
RL-based (DQN, Neural Net)	1.7054	0.0046

Improvements in DQN

- **Better Policy Learning**: Deep network learns non-linear mapping from state to action.
- **Efficient Sampling**: Experience replay buffer improves sample efficiency.
- **Reduced Redundancy**: Action masking ensures valid and unique column selections.
- **Stability**: Use of target network prevents Q-value divergence.
- **Performance**: DQN reduced error by $\sim$18% over pseudo-RL, with 68x speed-up.

DQN Workflow



1. Enhanced Randomized SVD

What Changed:

- Added `n_iter` power-iteration loops.
- Exposed `return_components` flag.

Why:

- Power iterations amplify the dominant singular subspace $\rightarrow$ better accuracy.
- Returning (U, s, V^T) enables reuse of factors (e.g. leverage-score computation).

2. CUR Decomposition

What Added:

- New function `cur_decomposition(A, k)`.
- Samples actual columns and rows via leverage-score sampling.
- Forms approximation $C\,U^{\dagger}\,R$.

Why:

- Provides an interpretable subset of original rows/columns.
- Useful when you need actual features, not abstract bases.

3. Evaluation Suite & Adaptive Rank

What Added:

- `evaluate_approximation`: computes Frobenius, spectral, nuclear errors, effective rank.
- `adaptive_rank_selection`: automatically chooses smallest r to meet tolerance.

Why:

- Standardizes comparison across methods and metrics.
- Removes need to guess the target rank in advance.

4. Enhanced RL Environment

What Changed:

- New `EnhancedColumnSelectionEnv` with:
  - Multiple state representations: `mask`, `correlation`, `leverage`, `combined`.
  - Multiple reward types: `error_reduction`, `normalized`, `spectral`, `combined`.
  - Early stopping and max-step logic.

Why:

- Richer state/reward improves learning and generalization.
- Early stopping avoids wasted steps when approximation is "good enough."

5. Actor–Critic Agent (A2C)

What Added:

- `ActorCritic` network: shared body + actor (policy) head + critic (value) head.
- `A2CAgent`: on-policy updates, advantage estimation, entropy bonus.
- Online learning every step (no replay buffer).

Why:

- Actor–Critic often learns faster and handles richer state spaces.
- Entropy regularization encourages continued exploration.

6. Experimental Framework

What Changed:

- `generate_test_matrix`: synthesize low-rank, exp-decay, clustered, ill-conditioned matrices.
- `comprehensive_method_comparison`: loops over matrices, ranks, methods, metrics.
- `format_results_table`: prints results as Markdown/LaTeX tables.

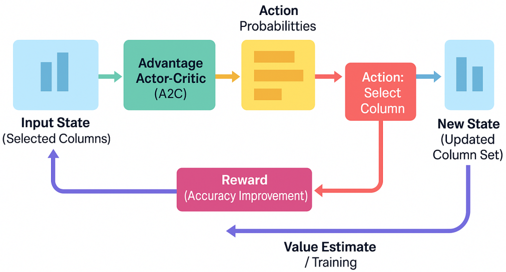
Why:

- Automates large-scale benchmarking.
- Makes it easy to compare classical vs. RL methods under varied conditions.

Deep Dive: Actor–Critic Network

- **Shared Backbone**:
  - Two fully-connected layers (state_dim → hidden_dim → hidden_dim) with ReLU
  - Extracts features common to policy and value estimation
- **Actor Head (Policy)**:
  - FC layer (hidden_dim → hidden_dim) + ReLU
  - Final FC (hidden_dim → action_dim) + Softmax
  - Outputs a probability distribution over columns
- **Critic Head (Value)**:
  - FC layer (hidden_dim → hidden_dim) + ReLU
  - Final FC (hidden_dim → 1)
  - Estimates expected return from current state

A2C Workflow



A2C Agent: Learning Mechanics

- **On-Policy Updates**:
  - Collect one (state, action, reward, next_state) at a time
  - Immediately update network—no replay buffer
- **Advantage Computation**:

$$\delta = r + \gamma\, V(s') - V(s)$$

- **Losses**:
  - Critic: $\mathrm{MSE}\big(V(s),\, r + \gamma V(s')\big)$
  - Actor: $-\log \pi(a \mid s)\, \delta$
  - Entropy: $-\beta \sum_a \pi(a \mid s) \log \pi(a \mid s)$
- **Entropy Bonus** (β):
  - Prevents policy collapse
  - Encourages exploration of less-visited columns

Enhanced Environment: State Representations

- **Mask** (baseline):
  - Binary vector of length n (0=available, 1=selected)
- **Leverage Scores**:
  - Precomputed $\ell_i = \sum_j U_{ij}^2$ from approximate SVD
  - Captures "importance" of each column
- **Correlation**:
  - Pairwise cosine similarities between columns
  - State entry = average correlation to selected set
- **Combined**:
  - Concatenates mask, leverage, correlation into a single vector

Enhanced Environment: Reward Variants

- **Error Reduction**:

$$r = \|A - A_{t-1}\|_F - \|A - A_t\|_F$$

- **Normalized**:

$$r = \frac{\|A - A_{t-1}\|_F - \|A - A_t\|_F}{\|A\|_F}$$

- **Spectral**:

$$r = \|A - A_{t-1}\|_2 - \|A - A_t\|_2$$

- **Combined**:
  - Mix of normalized error reduction + small efficiency bonus
  - Penalty if improvement ¡ threshold

Why These Enhancements Matter

- **Richer State** $\rightarrow$ agent sees more context (importance, redundancy)
- **Reward Variants** $\rightarrow$ tailor learning to different objectives (speed vs. accuracy)
- **Actor–Critic** $\rightarrow$ stable and efficient policy/value learning
- **Early Stopping** $\rightarrow$ avoids unnecessary steps once good approximation reached

Updated Performance Results

- **Test Setup**:
  - Synthetic 100×80 matrix, true rank $= 10$
  - Approximation rank $= 10$
- **Results**:

Method	**Error (Frobenius)**	**Time (sec)**
Deterministic SVD	0.7957	0.0232
Randomized SVD	0.7957	0.0007
RL-based Column Selection (DQN)	1.7054	0.0043

Training Curve (DQN):

- Episode 50: Total Reward $= 267.73$, $\varepsilon = 0.7783$
- Episode 100: Total Reward $= 256.28$, $\varepsilon = 0.6058$
- ...
- Episode 500: Total Reward $= 265.68$, $\varepsilon = 0.1000$

Training Pipeline for Multiple Matrix Sizes

- **Objective**: Pre-train RL agents on matrices of varying dimensions
- **Sizes**: $(100 \times 80), (200 \times 160), (400 \times 320), (800 \times 640)$
- **True Rank & Target Rank**: 10

Key Steps in `train_matrix_size_models()`:

1. `generate_test_matrix(m,n,10,'low_rank_noise')`
2. **DQN Agent**:
   - Create `ColumnSelectionEnv` for $m \times n$
   - Train for 300 episodes: `train_dqn(...)`
   - Save Q-network weights to `models/dqn_model_{m}x{n}.pt`
3. **A2C Agent**:
   - Create `EnhancedColumnSelectionEnv` (state=combined, reward=combined)
   - Train for 300 episodes: `train_a2c(...)`
   - Save actor-critic model to `models/a2c_model_{m}x{n}.pt`

Why Pre-Training on Multiple Sizes?

- **Generalization**: Agents learn policies robust across matrix dimensions
- **Efficiency**: Pre-trained models can be fine-tuned or deployed directly
- **Benchmarking**: Compare performance scaling (error vs. time) as size grows

Storage and Deployment

- **Model Directory**: `models/`
- **Saved Files**:
  - `dqn_model_100x80.pt`, ..., `dqn_model_800x640.pt`
  - `a2c_model_100x80.pt`, ..., `a2c_model_800x640.pt`
- **Next Steps**:
  - Load pre-trained agents for rapid inference on new matrices
  - Fine-tune on specialized datasets (e.g. real-world sparse matrices)

DQN Training Rewards by Matrix Size

- **Episodes = 300, ε decayed from 1.0 to 0.22**
- **Total Reward at Episode 300**:

Matrix Size	**Reward @300**
100×80	252.84
200×160	528.27
400×320	1114.03
800×640	2149.86

A2C Training Performance by Matrix Size

- **Episodes = 300, Steps per Episode = 10**
- **Final Episode Metrics**:

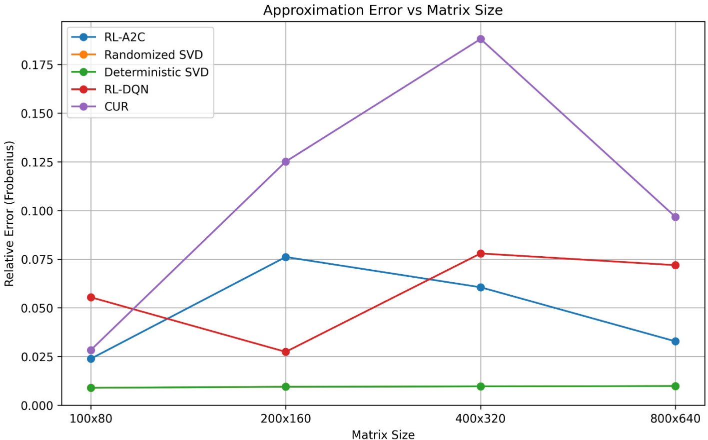
Size	Reward @300	Error @300
100×80	1.2172	0.0828
200×160	1.0819	0.2181
400×320	1.2490	0.0510
800×640	1.2737	0.0263

Saved Models

- **DQN Models**: `models/dqn_model_100x80.pt`, ..., `dqn_model_800x640.pt`
- **A2C Models**: `models/a2c_model_100x80.pt`, ..., `a2c_model_800x640.pt`
- **Next Steps**:
  - Fine-tune on real datasets
  - Evaluate inference speed accuracy tradeoffs

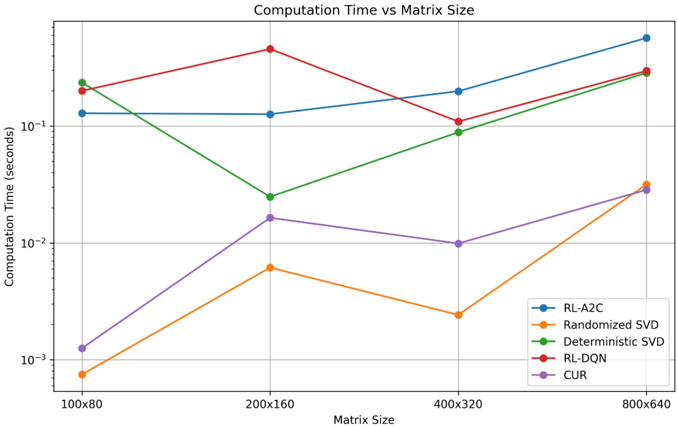
Relative Error by Matrix Size

- **Error Metric:** $\|A - \hat{A}\|_F / \|A\|_F$
- **Compared Methods:** SVD (Deterministic, Randomized), CUR, RL-based (DQN, A2C)



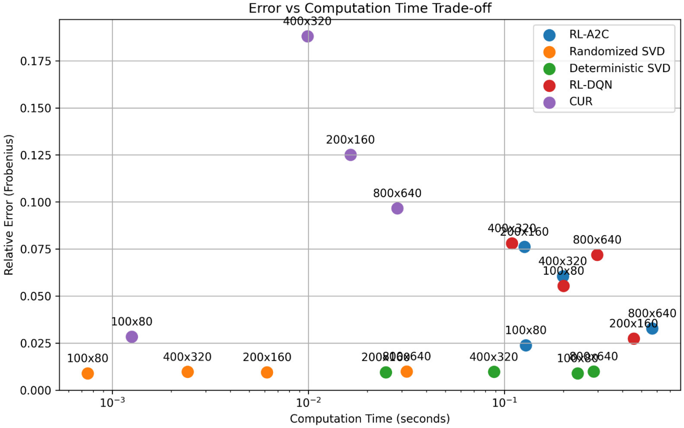
Computation Time by Matrix Size

- **Log Scale Used**
- **Time in seconds per method per matrix**



Error vs Time Trade-off

- **Each point = 1 matrix size for a method**
- **Goal: minimize both error and time**



RL Agents Trained on Diverse Matrix Types

- **Matrix Size:** 200×160, **True Rank = 10**
- **Trained on 4 Matrix Types**:
    1. Low-Rank + Noise
    2. Exponentially Decaying Singular Values
    3. Clustered Columns
    4. Ill-Conditioned Matrix ($\kappa = 10^4$)
- **Episodes:** 300 for each model

DQN Model Training (Matrix Types)

- **Training Setup:**
  - Environment: ColumnSelectionEnv
  - State Dim: n, Action Dim: n
  - ε-greedy exploration
- **Saved Models:**
  - `models/dqn_type_low_rank_noise.pt`
  - `models/dqn_type_exp_decay.pt`
  - `models/dqn_type_clustered.pt`
  - `models/dqn_type_ill_conditioned.pt`

A2C Model Training (Matrix Types)

- **Training Setup:**
  - Environment: EnhancedColumnSelectionEnv
  - State: `combined` (raw, leverage, top-k mask)
  - Reward: `combined` (rank + reconstruction gain)
- **Saved Models:**
  - `models/a2c_model_low_rank_noise.pt`
  - `models/a2c_model_exp_decay.pt`
  - `models/a2c_model_clustered.pt`
  - `models/a2c_model_ill_conditioned.pt`

DQN Training Performance by Matrix Type

- **Episodes = 300, ε decayed from 1.0 to 0.22**
- **Total Reward at Episode 300**:

Matrix Type	Reward @300
Low-Rank + Noise	485.23
Exp. Decay	1.24
Clustered	197.84
Ill-Conditioned	17894.58

A2C Training Performance by Matrix Type

- **Episodes = 300, Steps per Episode = 10**
- **Final Episode Metrics**:

Matrix Type	**Reward @300**	**Error @300**
Low-Rank + Noise	1.0754	0.1246
Exp. Decay	1.0815	0.1185
Clustered	1.2804	0.1196
Ill-Conditioned	—	—

- Ill-Conditioned matrix training still in progress

Approximation Error vs Matrix Type

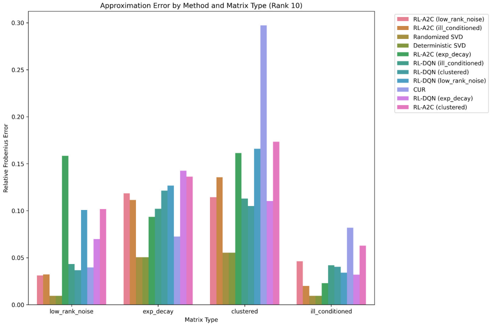


Figure: Relative Frobenius norm error for different methods across matrix sizes.

- Deterministic SVD and Randomized SVD perform consistently well.
- CUR and RL-based methods show more variance with matrix size.

Computation Time vs Matrix Type

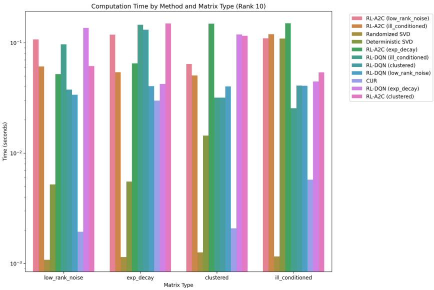


Figure: Computation time for different types. Log scale used for visibility.

- Deterministic SVD has the highest runtime growth.
- Randomized SVD and CUR offer good trade-offs in time.
- RL methods (DQN, A2C) are fast at inference but limited to small matrices.

Error vs Computation Time Trade-off

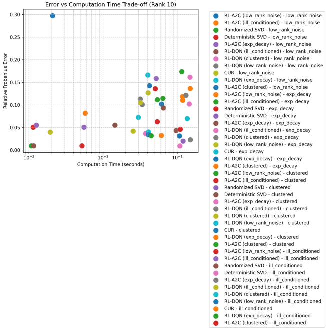


Figure: Trade-off between accuracy and runtime. Log scale on time.

- RL-based methods offer low error with low runtime for small matrices.
- Randomized SVD balances well across all matrix sizes.
- Deterministic SVD is accurate but slower.

Training Procedure

- **DQN:**
  - Environment: `ColumnSelectionEnv`
  - State: Selected column mask
  - Actions: Select next column
- **A2C:**
  - Environment: `EnhancedColumnSelectionEnv`
  - State: Combined (mask + singular value profile + residuals)
  - Reward: Combined error reduction and column diversity
- **Episodes:** 300 per agent per matrix type

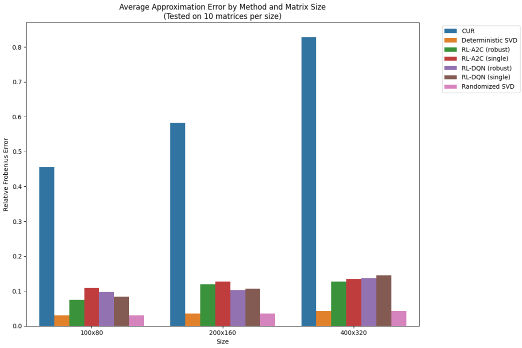
Robust Model Training and Testing Methodology

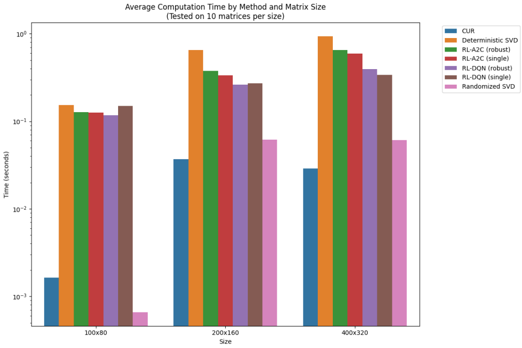
- **Objective**: Train RL models on multiple matrices to improve generalization
- **Training Details**:
  - Train on 10 different matrices for each size (100×80, 200×160, 400×320)
  - Different matrix types (low_rank_noise, exp_decay, clustered)
  - Fewer episodes per matrix (100) but more diverse training data

Model Type	Training Matrices	Testing Matrices
Single-Matrix	1 per size	10 new matrices
Robust Model	10 per size	10 new matrices

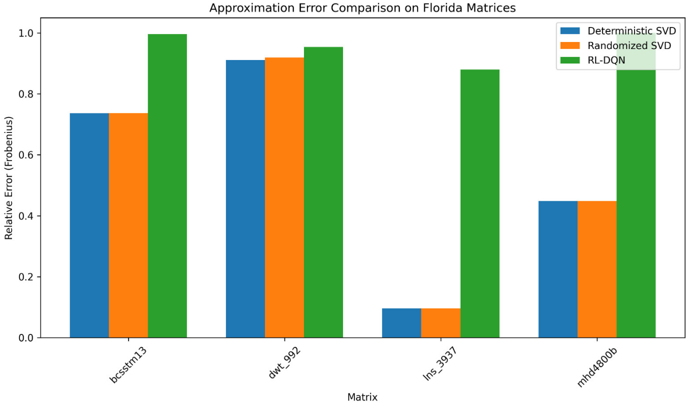
- **Testing Metrics**: Error reduction - We observed error reduction as compared to single models

Results for Robust Models



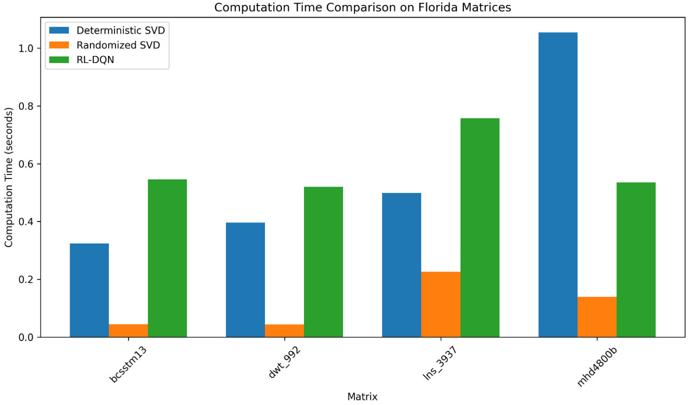


Approximation Error by Method for Florida Matrices



Relative Frobenius Error comparison across different Florida matrices

Computation Time by Method for Florida Matrices



Execution time (log scale) comparison across methods

Problems and Doubts

- What should be the number of matrices on which we should train?
- How to explore for florida matrices as they are of different sizes and types. So should we continue with out test-train partition or something else?