HPNA Project - Randomized Singular Value Decomposition and its Applications

Kevin Sidak, 01249373

University of Vienna, Faculty of Computer Science

1 Introduction

It is a common technique to use the Singular Value Decomposition(SVD) to allow lower rank approximations for any given matrix. Eckart Carl and Young Gale proofed in 1936 [4] that any matrix's best approximation in a lower rank is given by the SVD. Since for most nowadays applications in the realm of machine learning we also need a dimensionality reduction and not only a lower rank approximation, the Principal Component Analysis is more commonly used. In order to speed up the SVD itself researchers looked into different techniques like randomization in order to shorten the time of the SVD with an accuracy loss as small as possible. In this report I will discuss these techniques and apply them for an image classification task and a sparse matrix approximation.

2 Theory

In this section I want to discuss the different algorithms, theoretical background and the general ideas of the two papers [3] and [5]. I will mostly discuss algorithms of the two papers I tested and implemented myself since in these cases I have experiments to backup any claims I make during the discussion. For an better overview I categorise the approaches in two stages, as proposed by [5]:

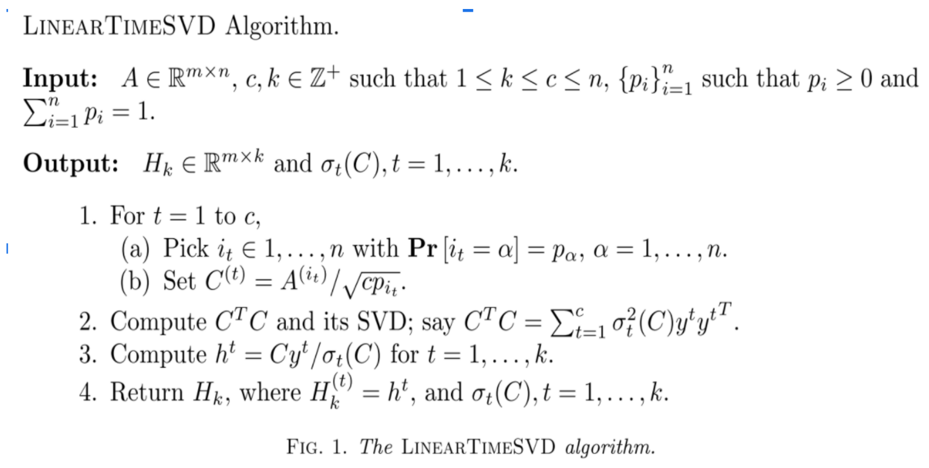
- Stage A: Finding an approximated matrix Q for the original matrix A. Q then will be used for the SVD.
- Stage B: Approximation of the SVD itself. Change the input matrix Q during the SVD itself or optimize the computation steps in the SVD like the matrix multiplications.

2.1 Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix

This paper only introduced new approaches for "Stage A". The two approaches are based on the approximation of A with row and columns extracted via a Monte-Carlo based distribution function dependent on the magnitude of individual rows and columns in the Frobenius norm.

Linear Time SVD The first proposed algorithm is the linear time SVD. Before using the algorithm we have to build our distribution function over the columns of our matrix A:

1. For each column c calculate: $p_i = |c|^2 \, / \, \|A\|_F^2$ where $_F$ is the Frobenius norm.
2. This gives us a probability p for each column which sum up to $\sum_{i=0}^{n} p_i = 1$.



LinearTimeSVD Algorithm.

Input: $A \in \mathbb{R}^{m\times n}$, $c, k \in \mathbb{Z}^+$ such that $1 \le k \le c \le n$, $\{p_i\}_{i=1}^{n}$ such that $p_i \ge 0$ and $\sum_{i=1}^{n} p_i = 1$.

Output: $H_k \in \mathbb{R}^{m\times k}$ and $\sigma_t(C), t = 1, \ldots, k$.

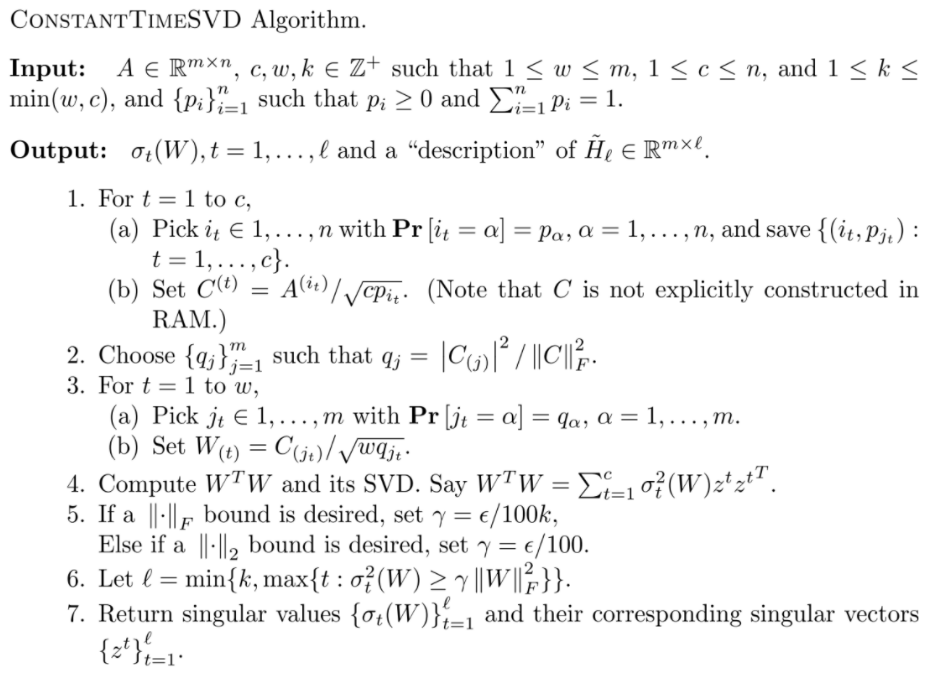
```
1. For $t = 1$ to $c$,
    (a) Pick $i_t \in 1, \ldots, n$ with $\mathbf{Pr}[i_t = \alpha] = p_\alpha$, $\alpha = 1, \ldots, n$.
    (b) Set $C^{(t)} = A^{(i_t)}/\sqrt{cp_{i_t}}$.
2. Compute $C^TC$ and its SVD; say $C^TC = \sum_{t=1}^{c} \sigma_t^2(C) y^t {y^t}^T$.
3. Compute $h^t = Cy^t/\sigma_t(C)$ for $t = 1, \ldots, k$.
4. Return $H_k$, where $H_k^{(t)} = h^t$, and $\sigma_t(C), t = 1, \ldots, k$.
```

Fig. 1. *The* LinearTimeSVD *algorithm.*

Fig. 1: Linear Time SVD [3]

After algorithm 1 we get for matrix A with dimensions $m \times n$ a matrix H with $m \times k$ where k is the number of columns we want to use. The runtime for this algorithm is than linear for O(max(m,n)).

Constant Time SVD The Constant Time SVD algorithm forms two distribution functions: one for the columns of A and one for the rows.



ConstantTimeSVD Algorithm.

Input: $A \in \mathbb{R}^{m \times n}$, $c, w, k \in \mathbb{Z}^+$ such that $1 \leq w \leq m$, $1 \leq c \leq n$, and $1 \leq k \leq \min(w, c)$, and $\{p_i\}_{i=1}^n$ such that $p_i \geq 0$ and $\sum_{i=1}^n p_i = 1$.

Output: $\sigma_t(W), t = 1, \ldots, \ell$ and a "description" of $\tilde{H}_\ell \in \mathbb{R}^{m \times \ell}$.

```
1. For $t = 1$ to $c$,
   (a) Pick $i_t \in 1, \ldots, n$ with $\mathbf{Pr}[i_t = \alpha] = p_\alpha$, $\alpha = 1, \ldots, n$, and save $\{(i_t, p_{j_t}) : t = 1, \ldots, c\}$.
   (b) Set $C^{(t)} = A^{(i_t)} / \sqrt{cp_{i_t}}$. (Note that $C$ is not explicitly constructed in RAM.)
2. Choose $\{q_j\}_{j=1}^m$ such that $q_j = \left|C_{(j)}\right|^2 / \|C\|_F^2$.
3. For $t = 1$ to $w$,
   (a) Pick $j_t \in 1, \ldots, m$ with $\mathbf{Pr}[j_t = \alpha] = q_\alpha$, $\alpha = 1, \ldots, m$.
   (b) Set $W_{(t)} = C_{(j_t)} / \sqrt{wq_{j_t}}$.
4. Compute $W^T W$ and its SVD. Say $W^T W = \sum_{t=1}^c \sigma_t^2(W) z^t {z^t}^T$.
5. If a $\|\cdot\|_F$ bound is desired, set $\gamma = \epsilon/100k$,
   Else if a $\|\cdot\|_2$ bound is desired, set $\gamma = \epsilon/100$.
6. Let $\ell = \min\{k, \max\{t : \sigma_t^2(W) \geq \gamma \|W\|_F^2\}\}$.
7. Return singular values $\{\sigma_t(W)\}_{t=1}^\ell$ and their corresponding singular vectors $\{z^t\}_{t=1}^\ell$.
```

Fig. 2: Constant Time SVD [3]

After selecting the columns in the same manner as in the linear time SVD we also select our rows in the same fashion like the columns of A. From algorithm 2 we then end up with an matrix W $w \times c$ and its corresponding SVD singular vectors within a desired error bound in the Frobenius or Spectral norm. It achieves a runtime bound of $O(c3 + cw2)$ which is constant if c and w are constant.

Runtime and Error Bounds Summary The authors of the first paper [3] proof the following error and runtime bounds for the two algorithms:

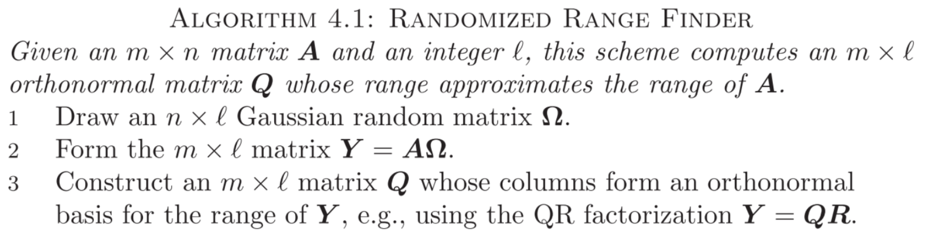
Additional error for:	LinearTimeSVD	ConstantTimeSVD	Ref. [16, 17]
$\|A - D^*\|_2^2$	$1/\epsilon^2$	$1/\epsilon^4$	k^4/ϵ^3
$\|A - D^*\|_F^2$	k/ϵ^2	k^2/ϵ^4	k^4/ϵ^3

Fig. 3: Summary table from [3]. k is the rank of the resulting matrix

2.2 Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions

The second paper introduced both, algorithms for Stage A and Stage B. For the image classification experiment I could only use algorithms from Stage A since Stage B relies heavily on row extraction techniques. The overall goal of the paper was to provide a "framework" of different randomization techniques for low rank approximations and the SVD in particular.

Randomized Range Finder The Randomized Range Finder is a simplified version of the Linear Time SVD algorithm 1. The main difference is that we don't calculate a distribution function ourselves. Instead we use an Gaussian random matrix Ω for that. Before the SVD we also use the QR factorization to construct a orthonormal basis from our matrix Q.


Algorithm 4.1: Randomized Range Finder

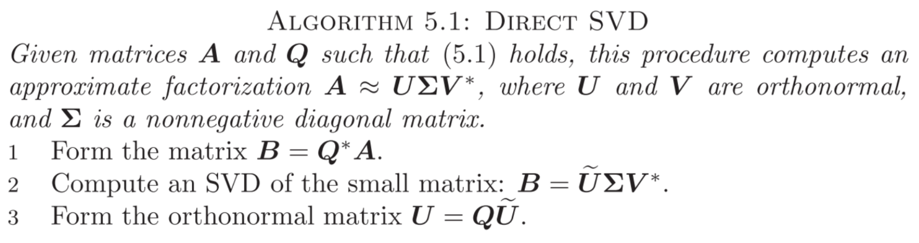
Given an $m \times n$ matrix $\boldsymbol{A}$ and an integer ℓ, this scheme computes an $m \times \ell$ orthonormal matrix $\boldsymbol{Q}$ whose range approximates the range of $\boldsymbol{A}$.

```
1  Draw an $n \times \ell$ Gaussian random matrix $\boldsymbol{\Omega}$.
2  Form the $m \times \ell$ matrix $\boldsymbol{Y} = \boldsymbol{A\Omega}$.
3  Construct an $m \times \ell$ matrix $\boldsymbol{Q}$ whose columns form an orthonormal
   basis for the range of $\boldsymbol{Y}$, e.g., using the QR factorization $\boldsymbol{Y} = \boldsymbol{QR}$.
```

Fig. 4: Randomized Range Finder [5]

After algorithm 4 similar to algorithm 1 and we end up with a matrix Q $m \times l$ where l is the number of columns we want to use.

Direct SVD The simplest approach fig 5 for the SVD is the Direct SVD from our approximated matrix Q from algorithm 4.


Algorithm 5.1: Direct SVD

Given matrices $\boldsymbol{A}$ and $\boldsymbol{Q}$ such that (5.1) holds, this procedure computes an approximate factorization $\boldsymbol{A} \approx \boldsymbol{U\Sigma V}^*$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, and $\boldsymbol{\Sigma}$ is a nonnegative diagonal matrix.

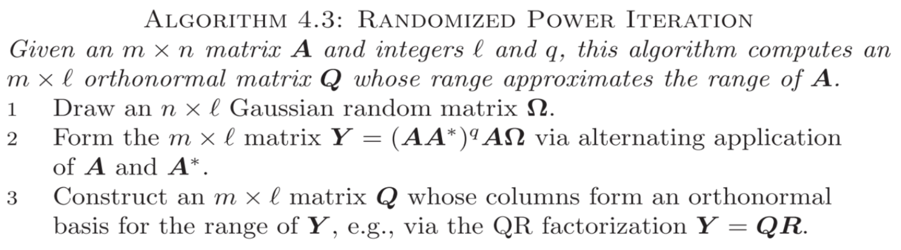
```
1  Form the matrix $\boldsymbol{B} = \boldsymbol{Q}^*\boldsymbol{A}$.
2  Compute an SVD of the small matrix: $\boldsymbol{B} = \widetilde{\boldsymbol{U}}\boldsymbol{\Sigma V}^*$.
3  Form the orthonormal matrix $\boldsymbol{U} = \boldsymbol{Q}\widetilde{\boldsymbol{U}}$.
```

Fig. 5: Direct SVD [5]

For the randomized range finder as Stage A and the direct SVD as Stage B we get the following bounds:

- Runtime: $(k+p)T_{mul} + O(k^2 m)$ flops
- Error: $\|A - U\Sigma V^*\| < \sqrt{kn \cdot \sigma_{k+1}}$

Randomized Power Iteration Another algorithm proposed for Stage A was the Randomized Power Iteration algorithm 6. The authors promote this algorithm for sparse matrices since the matrix multiplication can be evaluated rapidly.



```
ALGORITHM 4.3: RANDOMIZED POWER ITERATION
Given an $m \times n$ matrix $\boldsymbol{A}$ and integers $\ell$ and $q$, this algorithm computes an
$m \times \ell$ orthonormal matrix $\boldsymbol{Q}$ whose range approximates the range of $\boldsymbol{A}$.
1  Draw an $n \times \ell$ Gaussian random matrix $\boldsymbol{\Omega}$.
2  Form the $m \times \ell$ matrix $\boldsymbol{Y} = (\boldsymbol{A}\boldsymbol{A}^*)^q \boldsymbol{A}\boldsymbol{\Omega}$ via alternating application
   of $\boldsymbol{A}$ and $\boldsymbol{A}^*$.
3  Construct an $m \times \ell$ matrix $\boldsymbol{Q}$ whose columns form an orthonormal
   basis for the range of $\boldsymbol{Y}$, e.g., via the QR factorization $\boldsymbol{Y} = \boldsymbol{Q}\boldsymbol{R}$.
```

Fig. 6: Randomized Power Iteration [5]

3 Setup

For my experiments I used python as programming language and jupyter notebooks as coding enviorments. I also used the following libraries:

- SciPy [13]
- Numpy [9]
- Matplotlib [7]
- Scikit-learn [10]
- Pytorch [1]

The experiments where conducted on an Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz. The test matrices were taken from Matrix Market [8], Labeled Faces in the Wild database [6] and the university of Florida sparse matrix collection [2].

4 Experiments

In my experiments I wanted to focus on real life problems where the SVD is or could be used. The first on is an Image Classification Task where we can use the SVD to reduce our feature space. The second one is the SVD on sparse matrices.

4.1 Image Classification Task

For the first experiment I trained a support vector machine on the Labeled Faces in the Wild database. as tanning's and validation set. I tried to use so called "Eigenfaces", which are the left singular values of the SVD, as features to classify the different images of the persons. Eigenfaces for face recognition were proposed by Turk and Pentland in 1991 [12]. They claimed that most face features are encoded in the left singular values of the SVD. As base I used the proposed SVM image classifier from scikit-learn [11]. For the reduction of feature space in the training's set I used the following algorithms:

- Linear Time SVD fig 1
- Standard PCA of Numpy [9]
- Randomized Range Finder fig 4
- Randomized Power Iteration fig 6

All algorithms were tested with float16, float32 and float64 precision. But since BLAS for python converts every input to float32 the data was only perturbated but no speedup was gained. The runtime and accuracy of the SVM classification over the number of features on the test dataset is visualized in figures 7 and 8.

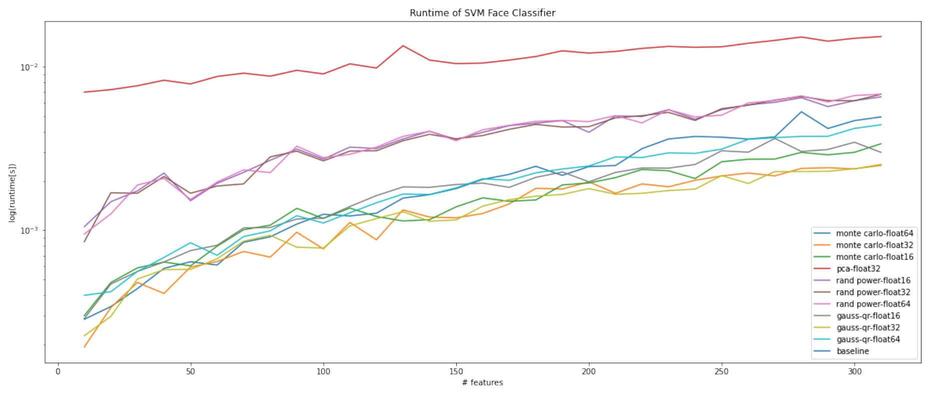


Fig. 7: Runtimes of the SVM Classification

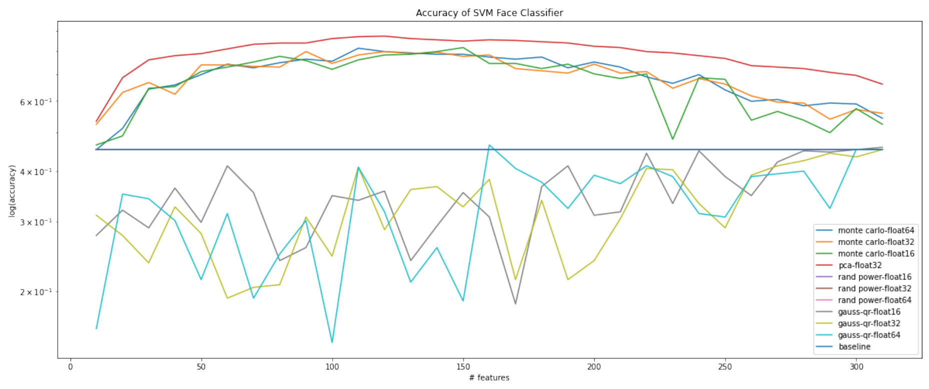


Fig. 8: Accuracy of the SVM Classification

4.2 Sparse Matrices

In the second task I wanted to test the performance of the proposed algorithms on sparse matrices. For this I used the "bundle1" matrix from the university of Florida sparse matrix collection [2]. In this test I used the same algorithms as in the image classification task. All algorithms were again tested with float16, float32 and float64 precision. The runtime and the relative forward error of the SVD dependent on the number of features are visualized in figures 9 and 10.

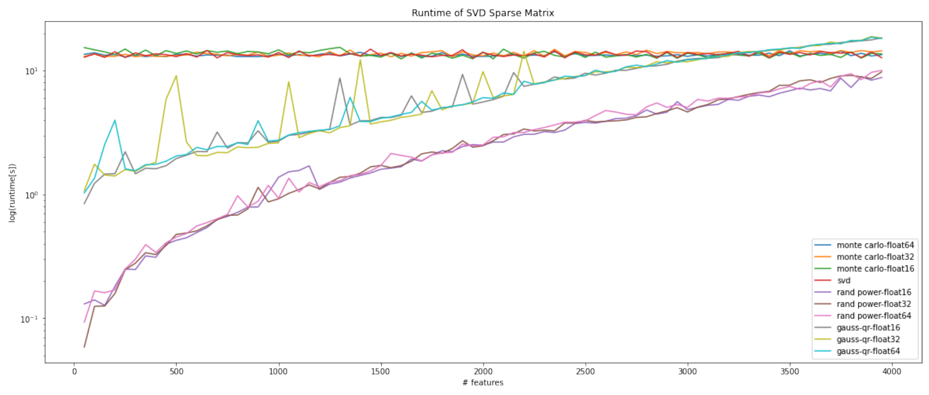


Fig. 9: Runtimes of the Sparse SVD

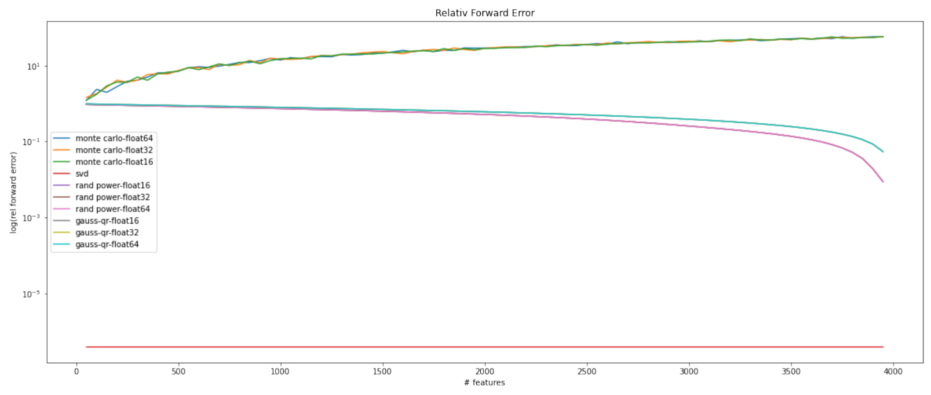


Fig. 10: Relative Forward Error of the Sparse SVD

The randomized power iteration implemented in torch [1] produced a near constant error near the baseline on the logarithmic scale.

5 Discussion

5.1 Image Classification

In my experiments I could observe that for real life examples like image classification the linear time SVD algorithm gives as a pretty good estimate compared with the standard PCA and even better than an classification over all features. It remains unclear why the authors of the second paper [5] didn't discuss Monte-Carlo as viable random approximation strategy for their range finder or power iteration. Overall the floating point precision had no noticeable impact in runtime or accuracy. The torch implemented power iteration was extremely close to the base line but had also the second worst runtime. The range finder is really fast but provides an accuracy that is close to simply randomly picking features from the dataset. Since the runtime is almost equal with Monte Carlo, a Monte Carlo based distribution is preferable if image data is classified.

5.2 Sparse Matrices

In this experiment the range finder has the best time to error ratio. One interesting result is that Monte Carlo actually takes less time the large the matrix is. This due to the fact that we only need to calculate the distribution function one time and picking from the distribution is constant. Monte Carlo would eventually be the fastest algorithm if the matrix is large enough.

6 Future Work

In this section I want to discuss several ideas I had will working on this project but had no time implementing them.

6.1 SVD/PCA in Neural Networks

It could be feasible to decompose weights in order to achieve higher batch sizes in neural networks. Nowadays batch size is mostly hardware bound by the GPU's onboard memory. the bound would be that:

- Better runtime with larger batch sizes and better generalization.
- The decomposition must not exceeds the loading from the disk of the batches.
- How large is the error we introduce.
- For other than dense neural networks we need to think about how to implement forward pass and gradient descent.

6.2 SVD/PCA vs. Autoencoders

I noticed will researching this topic that nowadays there could be a shift towards using autoencoders or even variational autoencoders for their feature extraction. One benefit here is that a neural network learns to tune the hyperparameters of the dimensionality reduction. An idea I had was to make an VAE and the randomized SVD's comparable. Since simply comparing the forward error wouldn't produce a meaningfully result. The autoencoder's encoder trains towards providing the most significant features for the decoder not for numerical accuracy. Training an SVM for a task with the reduced datasets from the SVD and the autoencoder might actually give us some insight for which datatypes which approach is the best.

References

[1] R. Collobert, K. Kavukcuoglu, and C. Farabet. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, NIPS Workshop*. 2011.

[2] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Transactions on Mathematical Software* 38.1 (Nov. 2011), pp. 1–25. ISSN: 00983500. DOI: 10.1145/2049662.2049663. URL: https://dl.acm.org/doi/10.1145/2049662.2049663.

[3] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. "Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix". In: *SIAM Journal on Computing* 36.1 (2006), pp. 158–183. ISSN: 00975397. DOI: 10.1137/S0097539704442696.

[4] Carl Eckart and Gale Young. "The approximation of one matrix by another of lower rank". In: *Psychometrika* 1.3 (Sept. 1936), pp. 211–218. ISSN: 00333123. DOI: 10.1007/BF02288367. URL: https://link-springer-com.uaccess.univie.ac.at/article/10.1007/BF02288367.

[5] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288. ISSN: 00361445. DOI: 10.1137/090771806.

[6] Gary Huang et al. “Labeled Faces in the Wild: A Database forStudying Face Recognition in Unconstrained Environments”. In: *Tech. rep.* (Oct. 2008).

[7] John D Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in science & engineering* 9.3 (2007), pp. 90–95.

[8] National Institute of Standards and Technology. *Matrix Market.* `https://math.nist.gov/MatrixMarket/`.

[9] Travis E Oliphant. *A guide to NumPy.* Vol. 1. Trelgol Publishing USA, 2006.

[10] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] scikit-learn developers. *Faces recognition example using eigenfaces and SVMs.* `https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html`.

[12] Matthew A Turk and Alex P Pentland. *Face Recognition Using Eigenfaces.* Tech. rep.

[13] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* (2020).