An Investigation of the Response of a Three-Layer Neural Network to Randomized SVD and Truncated SVD

Benjamin Keene — benkeene@knights.ucf.edu

University of Central Florida — December 10, 2022

Contents

1 Introduction

In this project we test the performance of a neural network's ability to classify handwritten digits from MNIST. We will present the layout of the network, present its baseline performance without modifications. Then, we will show the ability of the network to withstand compression of its weights via SVD while maintaining relatively accurate classification.

1.1 Implementation and network design

Our neural network takes 28×28 resolution images from MNIST and returns a value from $\{0, 1, ..., 9\}$ as classification. The layers of the network have the following structure:

```
self.linear_relu_stack = nn.Sequential(
        nn.Linear(28*28, 15),
        nn.ReLU(),
        nn.Linear(15, 15),
        nn.ReLU(),
        nn.Linear(15, 10)
)
```

The number and size of the hidden layers is restricted by the platform we are training on.

Important to state, all layers (except for the final layer) have greater than 10 neurons. If any layer had at most 10 neurons, then after compression we're likely to lose ability in classification of the 10 distinct digits of MNIST.

That is, applying a dimension reduction filter which reduces the rank of a weight matrix below 10 seems counterproductive to classification.

For our optimizer, we use stochastic gradient descent with a learning rate $lr = 10^{-2}$.

1.2 Hardware

This network was trained on the following platform:

- OS: Windows 11 Pro
- CPU: Intel i7-12700k
- RAM: 48GB
- GPU: NVIDIA GTX 1080
- SSD: Samsung 980 NVMe

While capable, larger networks (combined with the SVD tensor operations) proved too cumbersome to reasonably analyze.

1.3 Randomized SVD

The paper *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions* by Halko, Martinsson, Tropp. (2010, December) demonstrates a novel method for constructing low-rank approximations.

While not the focus of this project, the algorithm for the randomized SVD is given.

Algorithm 1: `Randomized SVD`

Input: $A \in \mathbb{R}^{m \times n}$
Target rank: k
Oversampling parameter: p
Result: U, Σ, V^T in an SVD of QQ^TA

Stage **A**

1. Draw $k + p$ random normal vectors ω_i in $\mathbb{R}^n$, forming Ω
2. Form the product $Y = A\Omega$
3. Calculate the $m \times (k + p)$ matrix Q from the QR factorization of Y

Stage **B**

1. Calculate an SVD of $Q^TA = \tilde{U}\Sigma V^T$.
2. Left multiply this SVD by Q, where $Q\tilde{U} := U$

return U, Σ, V^T such that $A \approx U\Sigma V^T$.

Loosely, the randomized SVD first calculates the matrix Q with orthonormal columns whose range approximates that of A. Finding such a Q such that, given $A \in \mathbb{R}^{m \times n}$ and precision tolerance ϵ,

$$\|A - QQ^TA\| \leq \epsilon$$

is called the *fixed-precision approximation problem*.

Constructing this matrix Q with $k + p$ orthonormal columns that satisfy

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

is called the *fixed-rank approximation problem*.

It turns out, that with few requirements on the hyperparamter p (an oversampling parameter), we have very good bounds on the error above. In expectation, this error

$$\mathbb{E}\|A - QQ^T A\| \leq \left[1 + \frac{4\sqrt{k+p}}{p-1} \cdot \sqrt{\min\{m,n\}}\right] \sigma_{k+1}$$

is at worst a polynomial factor of the next singular value of A. Halko et al. present that with relatively small p, the probability that the above inequality fails is roughly proportional to $p^{-(k+1)}$. While out of scope of this review, we can rest assured that the randomized SVD algorithm behaves well for our purposes.

1.4 Truncated SVD

For fun, we implement the "truncated SVD", the first k rank-one matrices in the SVD corresponding to the k-dominant singular values $\sigma_1, \sigma_2, \ldots, \sigma_k$.

By Eckart-Young theorem, we have that this is the **best** rank k approximation of A.

Algorithm 2: `Truncated SVD`

Input: $A \in \mathbb{R}^{m \times n}$
Target rank: k
Result: $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$

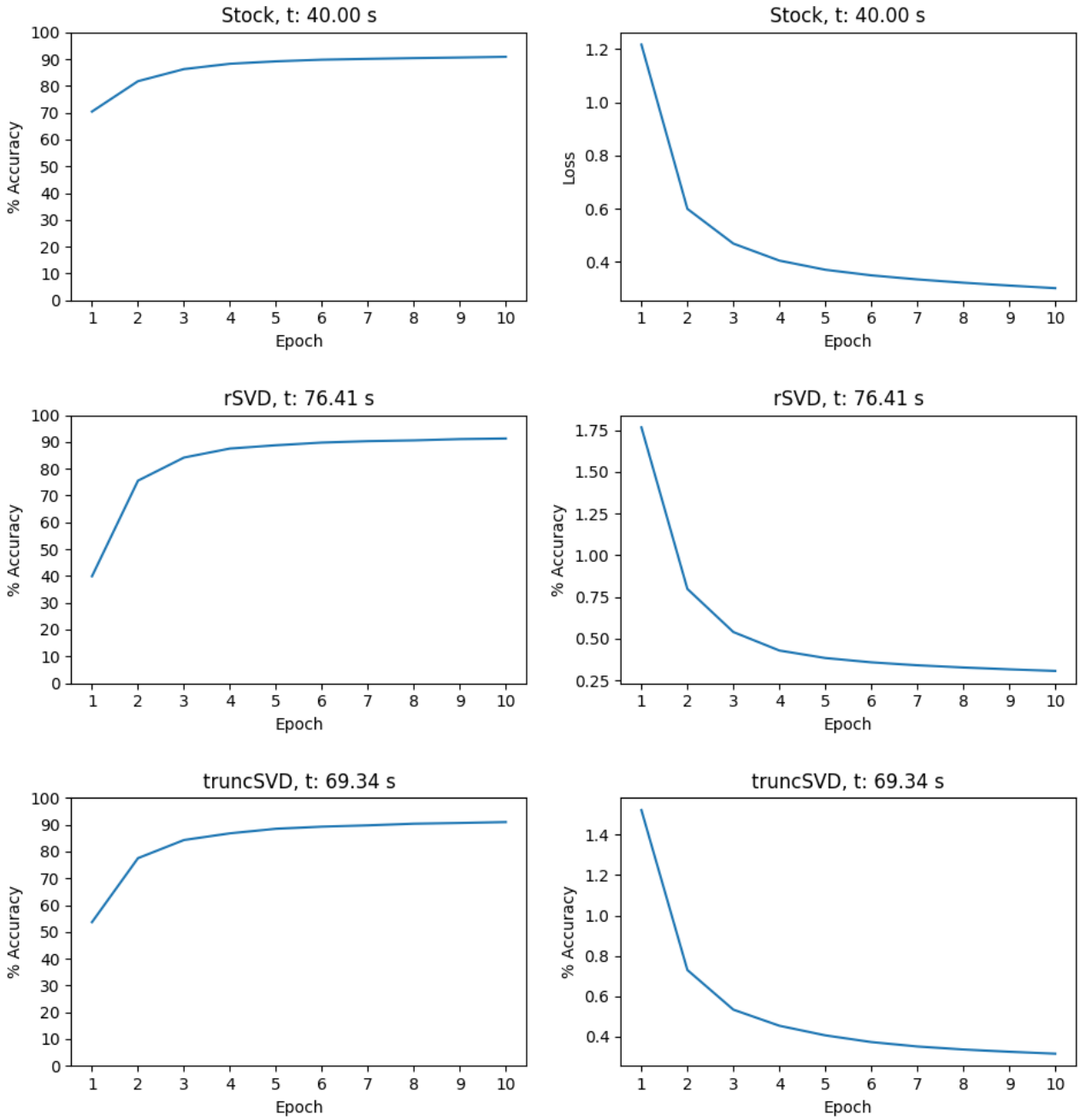
1. Calculate the SVD $A = U\Sigma V^T$
2. Set $[\Sigma]_{ii} = 0$ for $i > k$
3. Set $A_k = U\Sigma V^T$

return A_k

2 Performance Analysis

Presented below is the percent error, loss, and runtime of each implementation. The following parameters for randomized SVD and truncated SVD were used.

1. Stock: No manipulation of weight matrices during training
2. rSVD: After each epoch, approximate each weight matrix by a rank 10 approximation
3. truncSVD: After each epoch, approximate the each weight matrix by the first 10 terms of its singular value decomposition



Notice that each implementation (stock, rSVD, and truncSVD) reasonably learn to classify the digits. The runtime however, is very poor in the rSVD and truncSVD implementations.

3 Manipulating tensors in CUDA memory

Shown below are the functions used to modify the tensors while preserving the gradient. This was done via saving the model's state dictionary, manipulating the tensors directly, and reloading the altered weight matrices as the weights of a new model.

```
def rSVD(model, q):
    sd = model.state_dict()

    stack0 = torch.svd_lowrank(sd['linear_relu_stack.0.weight'], q)
    stack2 = torch.svd_lowrank(sd['linear_relu_stack.2.weight'], q)
    stack4 = torch.svd_lowrank(sd['linear_relu_stack.4.weight'], q)

    sd['linear_relu_stack.0.weight'] = (
        stack0[0] @ torch.diagflat(stack0[1]) @ torch.transpose(stack0[2], 0, 1))

    sd['linear_relu_stack.2.weight'] = (
        stack2[0] @ torch.diagflat(stack2[1]) @ torch.transpose(stack2[2], 0, 1))

    sd['linear_relu_stack.4.weight'] = (
        stack4[0] @ torch.diagflat(stack4[1]) @ torch.transpose(stack4[2], 0, 1))

    model.load_state_dict(sd)
    return
```

```
def truncSVD(model, q1):
    sd = model.state_dict()

    U0, S0, Vh0 = torch.linalg.svd(
        sd['linear_relu_stack.0.weight'], full_matrices=False)
    U2, S2, Vh2 = torch.linalg.svd(
        sd['linear_relu_stack.2.weight'], full_matrices=False)
    U4, S4, Vh4 = torch.linalg.svd(
        sd['linear_relu_stack.4.weight'], full_matrices=False)

    for j in range(15):
        if j > q1:
            S0[j] = 0
        if j > q1:
            S2[j] = 0
    for j in range(10):
        if j > q1:
            S4[j] = 0

    sd['linear_relu_stack.0.weight'] = torch.matmul(
        torch.matmul(U0, torch.diag(S0)), Vh0)
    sd['linear_relu_stack.2.weight'] = torch.matmul(
        torch.matmul(U2, torch.diag(S2)), Vh2)
    sd['linear_relu_stack.4.weight'] = torch.matmul(
        torch.matmul(U4, torch.diag(S4)), Vh4)

    model.load_state_dict(sd)
    return
```

4 Remarks on how this might be useful

We have shown how to implement Halko et al.'s randomized SVD as a compression algorithm against the weight matrices of a neural network, manipulating them in CUDA memory. While we gain no extra

accuracy by doing so, perhaps this is a feasible method for revealing the optimal rank (and further, the dimension) of a neural network's matrices. Presented below is a possible heuristic:

Algorithm 3: `NN Rank Revealer`

Input: Weight matrices $A_1, A_2, ..., A_n$, tolerances $\epsilon_1, \epsilon_2, ..., \epsilon_n$

Result: $k_1, k_2, ..., k_n$, the optimal rank of a low rank approximation of each matrix

1. For each $A_i \in \mathbb{R}^{m \times n}$ and $j \in \min\{m, n\}$:

   Calculate the loss, ℓ_{ij}, of the network with A_i approximated by its rank j approximation via rSVD or truncSVD

   If $\ell_{ij} - \ell_{i(j+1)} \geq \epsilon_i$, then assign A_i its rank j approximation via rSVD or truncSVD

return $A_1, A_2, ..., A_n$

This algorithm is clearly unfeasible for large n, but interesting to think about.

5 Conclusion

To conclude, we've demonstrated how to manipulate the weight matrices of a neural network while training. For the classification between 10 labels, the network behaves well when its matrices are replaced by rank 10 approximations via randomized SVD and truncated SVD.

6 How to reproduce results and view source code

Clone from https://github.com/benkeene/Compressed-Neural-Networks, and run `project.py`.

The repository is roughly 120 MB in size, due to the MNIST datasets used.