What if Neural Networks had SVDs?

Anonymous Authors[1]

Abstract

Various Neural Networks employ time-consuming matrix operations like matrix inversion. Many such matrix operations are faster to compute given the Singular Value Decomposition (SVD). Techniques from (Zhang et al., 2018; Mhammedi et al., 2017) allow using the SVD in Neural Networks without computing it. In theory, the techniques can speed up matrix operations, however, in practice, they are not fast enough. We present an algorithm which is up to $27\times$ faster than a previous approach, fast enough to speed up several matrix operations.

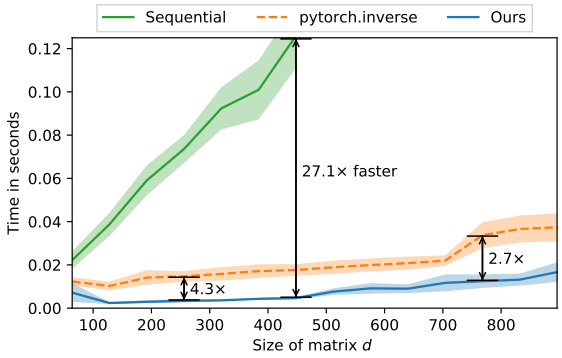


Figure 1. Time consumption of different approaches to matrix inversion in Neural Networks. The plot compares our algorithm against the sequential algorithm from (Zhang et al., 2018) and standard matrix inversion. Section 4.1 compares multiple matrix operations and a parallel algorithm from (Zhang et al., 2018).

1. Introduction

What could be done if the Singular Value Decomposition (SVD) of the weights in a Neural Network was given? Time-consuming matrix operations, such as matrix inversion (Hoogeboom et al., 2019), could be computed faster. Various Neural Networks employ such matrix operations, which often increase training time. Speeding up time-consuming matrix operations is thus very desirable.

For example, matrix inversion of $d \times d$ matrices could be computed and multiplied with a vector in $O(d^2)$ time instead of $O(d^3)$. Furthermore, Spectral Normalization (Miyato et al., 2018), often used by Generative Adversarial Networks (Goodfellow et al., 2014), could be done exactly in $O(d)$ time instead of approximated in $O(d^2)$. how

Both matrix operations take less time given the SVD. However, computing the SVD takes $O(d^3)$ time, which is no faster than computing either matrix operation. In Neural Networks, computing the SVD can be circumvented by the SVD reparameterization from (Zhang et al., 2018), which, in theory, allows speeding up both matrix operations.

However, in practice, we find that the previous approach to SVD reparameterization rarely attains any speed-ups for matrix operations on GPUs. This might not be surprising since the technique was not developed to speed up matrix operations. The difference in theory and practice occurs because the technique alters the forward pass of a fully connected layer to be highly sequential.

On a $d \times d$ weight matrix, the technique entails the computation of $O(d)$ sequential inner products, which is ill-fit for parallel hardware like GPUs. In practice, we find that the $O(d)$ sequential inner products take longer to compute than both matrix inversion and Spectral Normalization.

We introduce a novel algorithm that remedies the issue with sequential inner products. Our algorithm retains the same time complexity as the sequential algorithm from (Zhang et al., 2018) while reducing the number of sequential operations. On a mini-batch of size $m > 1$, our algorithm performs $O(d/m + m)$ sequential matrix-matrix operations instead of $O(d)$ sequential vector-vector operations.

In practice, our algorithm is faster than all algorithms from (Zhang et al., 2018), fast enough to speed up several matrix operations. For example, for matrix inversion in Neural Networks, our algorithm is up to $27\times$ faster than the sequential algorithm from (Zhang et al., 2018) and up to $4.3\times$ faster than standard matrix inversion, see Figure 1.

The remainder of the paper is structured as follows. In Section 2, we demonstrate the many benefits of SVDs for Neural Networks and outline the SVD technique from (Zhang et al., 2018). In Section 3, we present our new parallel algorithm and prove desirable theoretical guarantees. In Section 4, we present experiments, followed by related work in Section 5 and a conclusion in Section 7.

Code: see 'pythoncode.zip' attached to our submission.

2. Background

2.1. Fast Matrix Operations Using SVD

This subsection describes how several matrix operations, commonly used by Neural Networks, can be computed faster if the SVD is known. We consider a square weight matrix since 4 out of the 5 operations we consider are undefined for rectangular matrices. The SVD of a real square matrix $W \in \mathbb{R}^{d\times d}$ is $W = U\Sigma V^T$ where $\Sigma \in \mathbb{R}^{d\times d}$ is a diagonal matrix with $\Sigma_{ii} \geq 0$ and $U, V \in \mathbb{R}^{d\times d}$ are orthogonal matrices, i.e., $U^T = U^{-1}$ and $V^T = V^{-1}$.

Some of the matrix operations concern the special case where $W = W^T$. In this case, the SVD simplifies to $W = U\Sigma U^T$, however, the values of Σ_{ii} can be negative.

Lemma 1 states five well-known linear algebra results. Each result allows fast computation of a different matrix operation when the SVD is known. After the lemma, we describe how each matrix operation relates to Neural Networks.

Lemma 1. *Let W have a SVD $W = U\Sigma V^T$, it holds that*

1. ***Determinant.*** $|det(W)| = \prod_{i=1}^{d} \Sigma_{ii}$.
2. ***Inverse.*** $W^{-1} = V\Sigma^{-1}U^T$.
3. ***Largest singular value.*** $\max_{i=1}^{d} \Sigma_{ii}$.
4. ***Matrix exponential.*** *If $W = W^T$ then $W = U\Sigma U^T$,*

$$e^W := \sum_{k=0}^{\infty} \frac{1}{k!} W^k = U e^{\Sigma} U^T,$$

   where e^{Σ} is diagonal and $(e^{\Sigma})_{ii} = e^{\Sigma_{ii}}$. Similar is true when $W^T = -W$.

5. ***Cayley map.*** *If $W = W^T$ then $W = U\Sigma U^T$,*

$$\begin{aligned} C &:= (I - W)(I + W)^{-1} \\ &= U(I - \Sigma)(I + \Sigma)^{-1} U^T. \end{aligned}$$

   Similar is true when $W^T = -W$.

Proof. See the Supplementary Material 8.1. □

We now show how each result in Lemma 1 relates to the computation of matrix operations in Neural Networks.

Result 1 and 2 allow the computation of matrix determinant and matrix inversion in $O(d)$ time. Both operations are used by Normalizing Flows (Hoogeboom et al., 2019). Computing these operations with standard operations like PYTORCH.INVERSE(..) and PYTORCH.SLOGDET(..) (Paszke et al., 2019) take $O(d^3)$ time. Previous work suggests using the PLU and QR decomposition to circumvent the $O(d^3)$ standard operations (Kingma & Dhariwal, 2018; Hoogeboom et al., 2019). We discuss the advantages of the SVD approach in Section 5.

Result 3 allows the *exact* computation of the largest singular value in $O(d)$ time. The largest singular value is used for Spectral Normalization (Miyato et al., 2018). It is usually *approximated* in $O(d^2 r)$ time by running r rounds of the power iteration algorithm. Similar speed-ups are possible when W is rectangular.

Result 4 and 5 allow the computation of the matrix exponential and the Cayley map in $O(d)$ time. Both operations are used by (Casado, 2019), computed in $O(d^3)$ by Padé approximation and PYTORCH.SOLVE(..), respectively.

See Table 1 for a summary.

Similar bounds can be obtained for weight decay, pseudo-inverse, condition number and compression by low-rank approximation, see Supplementary Material 8.1.

Remark. All operations from Lemma 1 can be computed in $O(d)$ time, faster than computing a single matrix-vector multiplication $O(d^2)$. For example, if we compute $W^{-1}x$ for $x \in \mathbb{R}^d$ the time consumption is dominated by the multiplication and not the matrix inversion.

2.2. The SVD Technique

This subsection describes how (Zhang et al., 2018) allows using the SVD of the weights matrices in Neural Networks without computing them, and in particular, how this approach is limited by the computation of sequential inner

Table 1. Overview of the matrix operations from Lemma 1, comparing the time complexity of each operation with and without the SVD. The SVD of $W \in \mathbb{R}^{d\times d}$ is $W = U\Sigma V^T$ and r is the number of rounds used for the power iteration algorithm.

Operation		Complexity		Example Use Cases
Name	Given SVD	No SVD	With SVD	
Determinant	$\prod_{i=1}^{d} \Sigma_{ii}$	$O(d^3)$	$O(d)$	(Hoogeboom et al., 2019)
Inverse	$V\Sigma^{-1}U^T$	$O(d^3)$	$O(d)$	(Hoogeboom et al., 2019)
Largest singular value	$\max_{i=1}^{d} \Sigma_{ii}$	$O(d^2 r)$	$O(d)$	(Miyato et al., 2018)
Matrix Exponential	$Ue^{\Sigma}U^T$	$O(d^3)$	$O(d)$	(Casado, 2019)
Cayley map	$U(I-\Sigma)(I+\Sigma)^{-1}U^T$	$O(d^3)$	$O(d)$	(Casado, 2019)

products. Let $W = U\Sigma V^T$ be the SVD of a weight matrix W. The goal is to allow gradient descent updates of W while preserving the SVD. Consider updating U, Σ, V a small step $\eta \in \mathbb{R}$ in the direction of gradients $\nabla_U, \nabla_\Sigma, \nabla_V$.

$$\Sigma' = \Sigma - \eta\nabla_\Sigma, \quad U' = U - \eta\nabla_U, \quad V' = V - \eta\nabla_V.$$

While Σ' remains diagonal, both U' and V' are in general not orthogonal, which is needed to preserve the SVD. To this end (Zhang et al., 2018) suggest using a technique from (Mhammedi et al., 2017) which decomposes an orthogonal matrix as a product of d Householder matrices $H_1, \ldots, H_d$:

$$U = \prod_{i=1}^{d} H_i \qquad H_i = I - 2\frac{v_i v_i^T}{||v_i||_2^2} \qquad v_i \in \mathbb{R}^d. \quad (1)$$

Householder matrices satisfy several useful properties. In particular, the matrix U remains orthogonal under gradient descent updates $v_i = v_i - \eta\nabla_{v_i}$ (Mhammedi et al., 2017). Furthermore, all products of Householder matrices are orthogonal, and any $d \times d$ orthogonal matrix can be decomposed as a product of d Householder matrices (Uhlig, 2001). Householder matrices thus allow us to perform gradient descent over orthogonal matrices, which allows us to preserve the SVD of W during gradient descent updates.

Multiplication. One potential issue remains. The Householder decomposition might increase the time it takes to multiply UX for a mini-batch $X \in \mathbb{R}^{d\times m}$ during the forward pass. Consider computing

$$UX = H_1 \cdots (H_{d-1}(H_d \cdot X)). \quad (2)$$

The product UX can then be computed by d Householder multiplications. If done sequentially, as indicated by the parenthesis in Equation (2), each Householder multiplication can be computed in $O(dm)$ time (Zhang et al., 2018). All d multiplications can then be done in $O(d^2m)$ time. Therefore, the Householder decomposition does not increase the time complexity of computing UX.

Unfortunately, the $O(d^2m)$ time complexity comes at the cost of multiplying each Householder matrix sequentially, and each Householder multiplication entails computing an inner product, see Equation (1). The multiplication UX then requires the computation of $O(d)$ inner products sequentially. Such sequential computation is slow on parallel hardware like GPUs, much slower than normal matrix multiplication. To exploit GPUs (Zhang et al., 2018) suggests using a parallel algorithm that takes $O(d^3)$ time, but this is no faster than computing the SVD.

We are thus left with two options: (1) a $O(d^2m)$ sequential algorithm and (2) a $O(d^3)$ parallel algorithm. The first option is undesirable since it entails the sequential computation of $O(d)$ inner products. The second option is undesirable since it asymptotically takes the same time as the SVD, i.e., asymptotically, we might as-well compute the SVD. In practice, both algorithms usually achieve no speed-up for the matrix operations from Lemma 1, see Section 4.2.

Our main contribution is a novel parallel algorithm that resolves the issue with sequential inner products without increasing the time complexity. Our algorithm takes $O(d^2m)$ time but performs $O(d/m + m)$ sequential matrix-matrix operations instead of $O(d)$ sequential vector-vector operations (inner products). In practice, our algorithm is up to $6.2\times$ faster than the parallel algorithm and up to $27.1\times$ faster than the sequential algorithm, see Section 4.1.

Mathematical Setting. Time complexity is computed in the RAM model. The number of sequential matrix-matrix and vector-vector operations is simply counted. We count only once when other sequential operations can be done in parallel. For example, processing $v_1, ..., v_{d/2}$ sequentially while, in parallel, processing $v_{d/2+1}, ..., v_d$ sequentially, we count $d/2$ sequential vector-vector operations.

The Orthogonal Constraint. The SVD technique perform gradient descent over orthogonal matrices. This is possible with Householder matrices, however, other techniques exists. For example, techniques using the matrix exponential and the Cayley map (Casado, 2019; Li et al., 2020). For $d \times d$ matrices both techniques spend $O(d^3)$ time, no faster than computing the SVD.

3. A Parallel Algorithm

3.1. Forward Pass

Our goal is to create an $O(d^2m)$ algorithm with few sequential operations that solves the following problem: Given an input $X \in \mathbb{R}^{d\times m}$ with $d > m > 1$ and Householder matrices $H_1, ..., H_d$ compute the output $A = H_1 \cdots H_d X$. For simplicity, we assume m divides d.

Since each H_i is a $d \times d$ matrix, it would take $O(d^3)$ time to read the input $H_1, ..., H_d$. Therefore, we represent each Householder matrix H_i by its associated Householder vector v_i such that $H_i = I - 2v_iv_i^T/||v_i||_2^2$.

A simplified version of our algorithm proceeds as follows: divide the Householder product $H_1 \cdots H_d$ into smaller products $P_1 \cdots P_{d/m}$ so each P_i is a product of m Householder matrices:

$$P_i = H_{(i-1)\cdot m+1} \cdots H_{i\cdot m} \qquad i = 1, ..., d/m. \quad (3)$$

All d/m products P_i can be computed in parallel. The output can then be computed by $A = P_1 \cdots P_{d/m} X$ instead of $A = H_1 \cdots H_d X$, which reduces the number of sequential matrix multiplications from d to d/m.

This algorithm computes the correct A, however, the time complexity increases due to two issues. First, multiplying each product P_i with X takes $O(d^2m)$ time, a total of $O(d^3)$ time for all d/m products. Second, we need to compute all d/m products $P_1, ..., P_{d/m}$ in $O(d^2m)$ time, so each product P_i must be computed in $O(d^2m/(d/m)) = O(dm^2)$ time. If we only use the Householder structure, it takes $O(d^2m)$ time to compute each P_i, which is not fast enough.

Both issues can be resolved, yielding an $O(d^2m)$ algorithm. The key ingredient is a linear algebra result that dates back to 1987. The result is restated in Lemma 2.

Lemma 2. *(Bischof & Van Loan, 1987) For any m Householder matrices $H_1, ..., H_m$ there exists $W, Y \in \mathbb{R}^{d\times m}$ st.*

$$I - 2WY^T = H_1 \cdots H_m.$$

Both W and Y can be computed by m sequential Householder multiplications in $O(dm^2)$ time.

Proof. See (Bischof & Van Loan, 1987) Method 2. □

For completeness, we provide pseudo-code in Algorithm 1. Theorem 1 states properties of Algorithm 1 and its proof clarify how Lemma 2 solves both issues outlined above.

Algorithm 1 Forward Computation

Input: $X \in \mathbb{R}^{d\times m}$ and $H_1, ..., H_d \in \mathbb{R}^{d\times d}$.
Output: $A_1 = P_1 \cdots P_{d/m} X = H_1 \cdots H_d X$.

// Step 1
for $i = d/m$ **to** 1 **do in parallel**
  Compute $Y_i, W_i \in \mathbb{R}^{d\times m}$ such that ▷ $O(dm^2)$
    $P_i = I - 2W_iY_i^T$
  by using Lemma 2.
end for

// Step 2
$A_{d/m+1} = X$.
for $i = d/m$ **to** 1 **do sequentially**
  $A_i = A_{i+1} - 2W_i(Y_i^T A_{i+1})$. ▷ $O(dm^2)$
end for
return A_1.

Theorem 1. *Algorithm 1 computes $H_1 \cdots H_d X$ in $O(d^2m)$ time with $O(d/m + m)$ sequential matrix multiplications.*

Proof. **Correctness.** Each iteration of Step 2 computes

$$\begin{aligned} A_i &= A_{i+1} - 2W_i(Y_i^T A_{i+1}) \\ &= P_i A_{i+1}. \qquad \text{By Lemma 2} \end{aligned}$$

Therefore, at termination, $A_1 = P_1 \cdots P_{d/m} X$. In Step 1, we used Lemma 2 to compute the P_i's such that $A = H_1 \cdots H_d X$ as wanted.

Time complexity. Consider the for loop in Step 1. By Lemma 2, each iteration takes $O(dm^2)$ time. Therefore, the total time of the d/m iterations is $O(dm^2 d/m) = O(d^2m)$.

Consider iteration i of the loop in Step 2. The time of the iteration is asymptotically dominated by both matrix multiplications. Since A_{i+1}, X_i and Y_i all are $d \times m$ matrices, it takes $O(dm^2)$ time to compute both matrix multiplications. There are d/m iterations so the total time becomes $O(dm^2 d/m) = O(d^2m)$.

Number of Sequential Operations. Each iteration in Step 2 performs two sequential matrix multiplications. There are d/m sequential iterations which gives a total of $O(d/m)$ sequential matrix multiplications.

Each iteration in Step 1 performs m sequential Householder multiplications to construct P_i, see Lemma 2. Since each iteration is run in parallel, the algorithm performs no more than $O(d/m + m)$ sequential matrix multiplications. □

Remark. Section 3.2 extends the techniques from this section to handle gradient computations. For simplicity, this section had Algorithm 1 compute only A_1, however, in Section 3.2 it will be convenient to assume $A_1, ..., A_{d/m}$ are precomputed. Each $A_i = P_i \cdots P_{d/m} X$ can be saved during Step 2 of Algorithm 1 without increasing asymptotic memory consumption.

3.2. Backwards Propagation

This subsection extends the techniques from Section 3.1 to handle gradient computations. Our goal is to create an $O(d^2m)$ algorithm with few sequential operations that solves the following problem: Given $A_1, \ldots, A_{d/m+1}$, $P_1, ..., P_{d/m}$ and $\frac{\partial L}{\partial A_1}$ for some loss function L, compute $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial v_1}, ..., \frac{\partial L}{\partial v_d}$, where v_j is a Householder vector st. each Householder matrix is $H_j = I - 2v_jv_j^T/||v_j||_2^2$.

Since each P_i is a $d \times d$ matrix, it would take $O(d^3/m)$ time to read the input $P_1, ..., P_{d/m}$. Therefore, we represent each P_i by its WY decomposition $P_i = I - 2WY^T$.

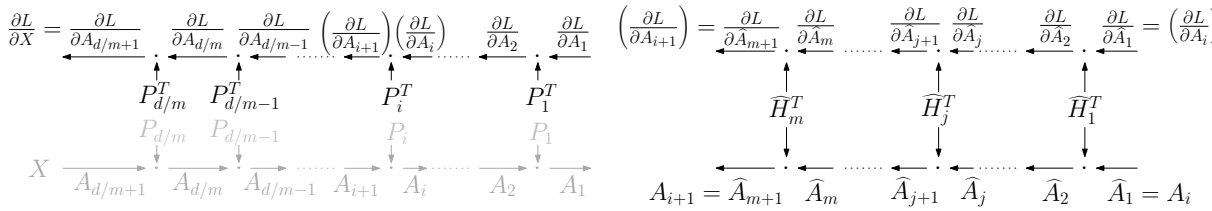
On a high-level our algorithm has two steps.

Step 1. Sequentially compute $\frac{\partial L}{\partial A_2}, \frac{\partial L}{\partial A_3}, ..., \frac{\partial L}{\partial A_{d/m+1}}$ by

$$\frac{\partial L}{\partial A_{i+1}} = \left[\frac{\partial A_i}{\partial A_{i+1}}\right]^T \frac{\partial L}{\partial A_i} = P_i^T \frac{\partial L}{\partial A_i} \tag{4}$$

This also gives the gradient wrt. X since $X = A_{d/m+1}$.

Step 2. We then use $\frac{\partial L}{\partial A_1}, ..., \frac{\partial L}{\partial A_{d/m}}$ from Step 1 to compute the gradient $\frac{\partial L}{\partial v_j}$ for all j. This problem can be split into d/m subproblems, which can be solved in parallel, one subproblem for each $\frac{\partial L}{\partial A_i}$.



(a) Step 1: Sequential part of Algorithm 2.

(b) Step 2: The i'th subproblem in Algorithm 2.

Figure 2. Computational graph of Step 1 and the i'th subproblem in Step 2 from Algorithm 2.

For completeness, we state pseudo-code in Algorithm 2, which we now explain with the help of Figure 2.

Figure 2a depicts a computational graph of Step 1 in Algorithm 2. In the figure, consider $\frac{\partial L}{\partial A_1}$ and P_1^T, which both have directed edges to a multiplication node (denoted by $\cdot$). The output of this multiplication is $\frac{\partial L}{\partial A_2}$ by Equation (4). This can be repeated to obtain $\frac{\partial L}{\partial A_2}, ..., \frac{\partial L}{\partial A_{d/m+1}}$.

Step 2 computes the gradient of all Householder vectors $\frac{\partial L}{\partial v_j}$. This computation is split into d/m distinct subproblems that can be solved in parallel. Each subproblem concerns $\frac{\partial L}{\partial A_i}$ and the product P_i, see line 10-12 in Algorithm 2.

To ease notation, we index the Householder matrices of P_i by $P_i = \widehat{H}_m \cdots \widehat{H}_1$. Furthermore, we let $\widehat{A}_{m+1} := A_{i+1}$ and $\widehat{A}_j := \widehat{H}_j \widehat{A}_{j+1}$. The notation implies that $\widehat{A}_1 = \widehat{H}_1 \cdots \widehat{H}_m \widehat{A}_{m+1} = P_i A_{i+1} = A_i$. The goal of each subproblem is to compute gradients wrt. the Householder vectors $\widehat{v}_m, ..., \widehat{v}_1$ of $\widehat{H}_m, ..., \widehat{H}_1$. To compute the gradient of $\widehat{v}_j$, we need $\widehat{A}_{j+1}$ and $\frac{\partial L}{\partial \widehat{A}_j}$, which can be computed by:

$$\widehat{A}_{j+1} = \widehat{H}_j^{-1} \widehat{A}_j = \widehat{H}_j^T \widehat{A}_j \tag{5}$$

$$\frac{\partial L}{\partial \widehat{A}_{j+1}} = \left[\frac{\partial \widehat{A}_j}{\partial \widehat{A}_{j+1}}\right]^T \frac{\partial L}{\partial \widehat{A}_j} = \widehat{H}_j^T \frac{\partial L}{\partial \widehat{A}_j} \tag{6}$$

Figure 2b depicts how $\widehat{A}_2, ..., \widehat{A}_{m+1}$ and $\frac{\partial L}{\partial \widehat{A}_2}, ..., \frac{\partial L}{\partial \widehat{A}_{m+1}}$

Given $\widehat{A}_{j+1}$ and $\frac{\partial L}{\partial \widehat{A}_j}$, we can compute $\frac{\partial L}{\partial \widehat{v}_j}$ as done in (Zhang et al., 2018; Mhammedi et al., 2017). For completeness, we restate the needed equation in our notation, see Equation (7). Let $a^{(l)}$ be the l'th column of $\widehat{A}_{j+1}$ and let $g^{(l)}$ be the l'th column of $\frac{\partial L}{\partial \widehat{A}_j}$. The sum of the gradient over a mini-batch of size m is then:

$$-\frac{2}{||\widehat{v}_j||_2^2} \sum_{l=1}^{m} (\widehat{v}_j^T a^{(l)}) g^{(l)} + (\widehat{v}_j^T g^{(l)}) a^{(l)} - \frac{2}{||\widehat{v}_j||_2^2} (\widehat{v}_j^T a^{(l)})(\widehat{v}_j^T g^{(l)}) \widehat{v}_j \tag{7}$$

Theorem 2 states properties of Algorithm 2.

Algorithm 2 Backwards Computation

1: **Input:** $A_1, ..., A_{d/m+1}, P_1, ..., P_{d/m}$ and $\frac{\partial L}{\partial A_1}$.
2: **Output:** $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial v_k}$ for all k where $H_k = I - 2\frac{v_k v_k^T}{||v_k||_2^2}$.
3:
4: // Step 1
5: **for** $i = 1$ **to** d/m **do sequentially**
6: $\quad \frac{\partial L}{\partial A_{i+1}} = P_i^T \frac{\partial L}{\partial A_i}$ eq. (4). $\triangleright\ O(dm^2)$
7: **end for**
8:
9: // Step 2
10: **for** $i = 1$ **to** d/m **do in parallel**
11: $\quad$ Let $\frac{\partial L}{\partial \widehat{A}_1} = \left(\frac{\partial L}{\partial A_i}\right)$.
12: $\quad$ To ease notation, let $P_i = \widehat{H}_m \cdots \widehat{H}_1$.
13: $\quad$ **for** $j = 1$ **to** m **do**
14: $\quad\quad$ Compute $\widehat{A}_{j+1}$, $\frac{\partial L}{\partial \widehat{A}_j}$, eqs. (5) and (6). $\triangleright\ O(dm)$
15: $\quad\quad$ Compute $\frac{\partial L}{\partial \widehat{v}_j}$ using $\widehat{A}_{j+1}$, $\frac{\partial L}{\partial \widehat{A}_j}$, eq. (7). $\triangleright\ O(dm)$
16: $\quad$ **end for**
17: **end for**
18: **return** $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial A_{d/m+1}}$ and $\frac{\partial L}{\partial v_k}$ for all $k = 1, ..., d$.

Theorem 2. *Algorithm 2 computes $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial v_1}, ..., \frac{\partial L}{\partial v_d}$ in $O(d^2 m)$ time with $O(d/m + m)$ sequential matrix multiplications.*

Proof. See the Supplementary Material 8.2. □

3.3. Extensions

Trade-off. Both Algorithm 1 and Algorithm 2 can be extended to take a parameter k that controls a trade-off between *total time complexity* and *the amount of parallelism*. This can be achieved by changing the number of Householder matrices in each product P_i from the mini-batch size m to an integer $k \in \{2, ..., d-1\}$. The resulting algorithms takes $O(d^2 k + d^2 m)$ time, $O(d^2 m/k)$ space and has $O(d/k + k)$ sequential matrix multiplications. This extension has the practical benefit that one can try different values of k and choose the one that yields superior performance on a particular hardware setup.

Convolutional Layers. So far, we have considered the SVD technique for matrices which corresponds to fully connected layers. The matrix case extends to convolutions by, e.g., 1×1 convolutions (Kingma & Dhariwal, 2018) or invertible periodic convolutions (Hoogeboom et al., 2019). The SVD technique can be used for such convolutions without increasing the time complexity. On an input with height h and width w our algorithm performs $O(d/m + mhw)$ sequential matrix multiplications instead of the $O(d)$ sequential inner products of the previous algorithm.

Recurrent Layers. The SVD technique was developed for Recurrent Neural Networks (RNNs) (Zhang et al., 2018). Let r be the number of recurrent applications of the RNN. Our algorithm performs $O(d/m + rm)$ sequential matrix operations instead of the $O(d)$ sequential inner products.

4. Experiments

This section contains two experiments. Section 4.1 compares the running time of our algorithm against alternatives. Section 4.2 shows our algorithm speeds up matrix operations. To simulate a realistic machine learning environment, we performed all experiments on a standard machine learning server using a single NVIDIA RTX 2080 Ti.

4.1. Comparing Running Time

This subsection compares the running time of our algorithm against four alternative algorithms. We compare the time all algorithms spend on gradient descent with a single orthogonal matrix, since such constrained gradient descent dominates the running time of the SVD technique.

We first compare our algorithm against the parallel and sequential algorithm from (Zhang et al., 2018), all three algorithms rely on the Householder decomposition. For completeness, we also compare against approaches that does not rely on the Householder decomposition, in particular, the matrix exponential and the Cayley map (Casado, 2019). See Supplementary Material 8.3 for further details.

We measure the time of a gradient descent step with a weight matrix $W \in \mathbb{R}^{d\times d}$ and a mini-batch $X \in \mathbb{R}^{d\times m}$, where $m = 32$ and $d = 1 \cdot 64, 2 \cdot 64, ..., 48 \cdot 64$. We ran each algorithm 100 times, and we report mean time μ with error bars $[\mu - \sigma, \mu + \sigma]$ where σ is the standard deviation of running time over the 100 repetitions.

Figure 3 depicts the running time on the y-axis, as the size of the $d \times d$ matrices increases on the x-axis. For $d > 64$, our algorithm is faster than all previous approaches. At $d = 64$ our algorithm is faster than all previous approaches, except the parallel algorithm. Previous work employ sizes $d = 192$ in (Kingma & Dhariwal, 2018) and $d = 784$ in (Zhang et al., 2018).

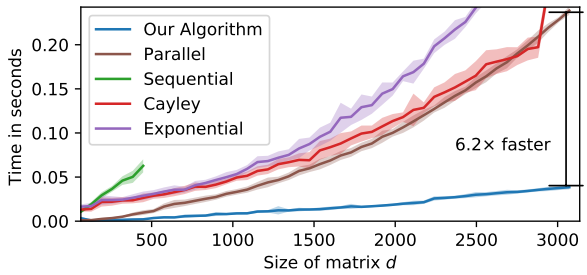


Figure 3. Running time of different algorithms for $d \times d$ matrices. Our algorithm is fastest when $d > 64$. The sequential algorithm from (Zhang et al., 2018) crashed when $d > 448$.

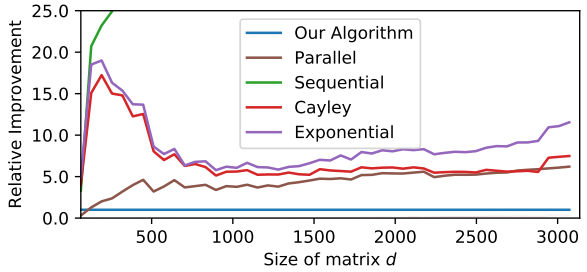


Figure 4. Improvement of our algorithm relative to previous algorithms, i.e., the mean time of a previous algorithm divided by the mean time of our algorithm.

Figure 4 depicts how much faster our algorithm is relative to the previous algorithms, i.e., the mean time of a previous algorithm divided by the time of our algorithm, which we refer to as relative improvement. For $d > 500$, the relative improvement of our algorithm increase with d.

4.1.1. Additional Insights

Householder Decomposition. At $d = 448$ our algorithm is roughly $25\times$ faster than the sequential algorithm. Our algorithm is faster with $d = 3072$ than the sequential algorithm with $d = 448$. Previous work like (Hoogeboom et al., 2019; van den Berg et al., 2018; Mhammedi et al., 2017) use the Householder decomposition with the sequential algorithm. Since our algorithm computes the same thing as the sequential algorithm, it can speed-up their computation without degrading performance in any way.

Implementations of Previous Algorithms. For the matrix exponential and the Cayley map we used the open-source implementation[1] from (Casado, 2019). For the parallel and sequential algorithm we used the open-source implementation[2] from (Zhang et al., 2018).

[1] https://github.com/Lezcano/expRNN
[2] https://github.com/zhangjiong724/spectral-RNN

4.2. Using the SVD to Compute Matrix Operations

This subsection investigates whether the matrix operations from Lemma 1 achieves speed-ups in practice. The matrix operations are usually used during the forward pass of a Neural Network, changing the subsequent gradient computations. Therefore, we measure the sum of the time it takes to compute the matrix operation, the forward pass and the subsequent gradient computations.

For example, with matrix inversion, we measure the time it takes to compute the matrix operation Σ^{-1}, the forward pass $W^{-1}X = V\Sigma^{-1}U^TX$ and the subsequent gradient computation wrt. U, Σ, V, X. The measured time is compared with a standard approach like PYTORCH.INVERSE (Paszke et al., 2019). Again, we measure the time of the matrix operation PYTORCH.INVERSE(W), the forward pass $W^{-1}X$, and the subsequent gradient computation wrt. W, X.

We use the following standard approaches, inspired by (Kingma & Dhariwal, 2018) and (Casado, 2019):

Determinant:	PYTORCH.SLOGDET(W)
Inverse:	PYTORCH.INVERSE(W)
Cayley:	PYTORCH.SOLVE(I - W, I + W)
Exponential:	Padé Approximation.

We run the SVD technique with three different algorithms: our algorithm, the sequential and the parallel algorithm from (Zhang et al., 2018). For each matrix operation, we consider matrices $V, \Sigma, U, W \in \mathbb{R}^{d\times d}$ and $X \in \mathbb{R}^{d\times M}$, where $m = 32$ and $d = 1 \cdot 64, 2 \cdot 64, ..., 48 \cdot 64$. We repeat the experiment 100 times, and report the mean time μ with error bars $[\mu - \sigma, \mu + \sigma]$ where σ is the standard deviation of the running times over the 100 repetitions.

We plot the time of the SVD technique for three different algorithms: ours, parallel and sequential. To avoid clutter, we plot only the time of our algorithm for the matrix operation it is slowest to compute, and the time of the previous algorithms for the matrix operations they were fastest to compute. See Supplementary Material 8.4 for details.

Figure 5 depicts the measured running time on the y-axis with the size of the $d \times d$ matrices increasing on the x-axis. Each matrix operation is plotted as a dashed line, and the different algorithms are plotted as solid lines. In all cases, our algorithm is faster than the standard approach. For example, with $d = 768$, our algorithm is $3.1\times$ faster than the Cayley map, $4.1\times$ faster than the matrix exponential, $2.7\times$ faster than inverse and $3.5\times$ faster than matrix determinant. At $d = 768$, the parallel algorithm provides a speed up for only one operation, a $1.1\times$ speed-up for the matrix exponential. The sequential algorithm is not fast enough to speed up any matrix operation.

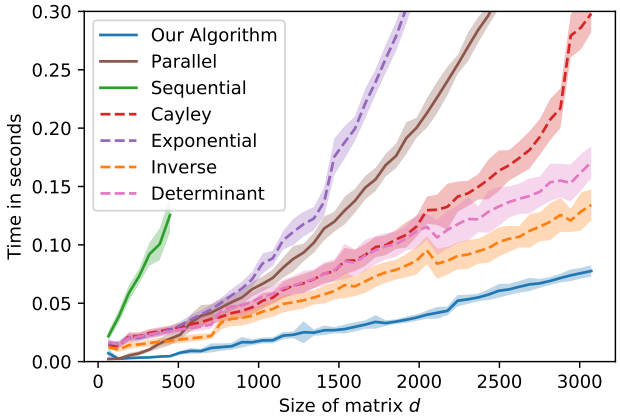


Figure 5. Running time of matrix operations. Solid lines depict approaches which use the SVD technique and dashed lines depict standard approaches like PYTORCH.INVERSE.

4.2.1. Additional Insights

Previous work on Matrix Determinant. Previous work suggested speeding up matrix determinant by using the PLU decomposition (Kingma & Dhariwal, 2018) and the QR decomposition (Hoogeboom et al., 2019). Our algorithm can speed up the QR decomposition, which (Hoogeboom et al., 2019) introduced to fix a limitation of the PLU decomposition. This is possible because the QR decomposition uses an orthogonal matrix, which, in (Hoogeboom et al., 2019), is done by using the Householder decomposition. See the Related Work in Section 5 for details.

Spectral Normalization. Previous work (Miyato et al., 2018) uses the power iteration algorithm to approximate the largest singular value of a matrix W. Power iteration takes an initial random vector v_0, and iteratively computes powers $v_{i+1} = Wv_i/||Wv_i||$ until a sufficiently good approximation is reached. In Neural Networks, v_0 is usually initialized by the result from the previous mini-batch update. In this case, one power iteration usually gives a sufficiently good approximation. We found the increased multiplication time caused by the SVD technique incurs a larger overhead than one power iteration, i.e., no speed-up.

Pay For One, Get Spectral Normalization For Free. The time consumption of the SVD technique is dominated by the forward pass and the subsequent gradient computations. For example, when $d = 3072$, computing Σ^{-1} takes just 0.7% of the time it takes to compute $V\Sigma^{-1}V^TX$. If we also compute Spectral Normalization, we only need to compute $\Sigma/\max_i \Sigma_{ii}$, with a negligible increase in total computation time. The same is true for matrix determinant $\prod_i \Sigma_{ii}$, weight decay $\sum_i \Sigma_{ii}^2$ and condition number $\max_i \Sigma_{ii}/\min_i \Sigma_{ii}$.

5. Related Work

The Householder Decomposition. The Householder decomposition of orthogonal matrices has been used in much previous works, for example, (Tomczak & Welling, 2016; Mhammedi et al., 2017; Zhang et al., 2018; van den Berg et al., 2018; Hoogeboom et al., 2019). Previous work typically use a type of sequential algorithm that performs $O(d)$ sequential inner products. To circumvent the resulting long computation time on GPUs, previous work often suggest limiting the number of Householder matrices, which limits the expressiveness of the orthogonal matrix, introducing a trade-off between computation time and expressiveness.

Our algorithm takes the same asymptotic time as the sequential algorithm, however, it performs less sequential matrix operations, making it up to $27\times$ faster in practice. Since our algorithm computes the same output as the previous sequential algorithms, it can be used in previous work without degrading the performance of their model. In particular, our algorithm can be used to either (1) increase expressiveness at no additional computational cost or (2) speed up previous applications at the same level of expressiveness.

SVDs in Neural Networks. The authors of (Zhang et al., 2018) introduced a technique that provides access to the SVD of the weights in a Neural Network without computing the SVD. Their motivation for developing this technique was the exploding/vanishing gradient issue in RNNs. In particular, they use the SVD technique to force all singular values to be within the range $[1 \pm \epsilon]$ for some small ϵ.

We point out that their technique, in theory, can be used to speed up matrix operations, and, furthermore, that their algorithms are too slow to speed-up most matrix operations in practice. To mitigate this problem we introduce a new algorithm that is more suitable for GPUs, which allows us to speed up several matrix operations.

Different Orthogonal Parameterizations. The SVD technique by (Zhang et al., 2018) uses the Householder decomposition to perform gradient descent with orthogonal matrices. Their work was followed by (Golinski et al., 2019) that raises a theoretical concern about the use of Householder decompositions. Alternative approaches based on the matrix exponential and the Cayley map have desirable provable guarantees, which currently, it is not known whether the Householder decomposition possesses. This might make it desirable to use the matrix exponential or the Cayley map together with the SVD technique from (Zhang et al., 2018). However, previous work spend $O(d^3)$ time to compute or approximate the matrix exponential and the Cayley map. These approaches are thus undesirable for SVD since they asymptotically take the same time as computing the SVD.

Normalizing Flows. Normalizing Flows (Dinh et al., 2015) is a type of generative model that, in some cases (Kingma & Dhariwal, 2018; Hoogeboom et al., 2019), entails the computation of matrix determinant and matrix inversion. (Kingma & Dhariwal, 2018) propose to use the PLU decomposition $W = PLU$ where P is a permutation matrix and L, U are lower and upper triangular. This allow the determinant computation in $O(d)$ time instead of $O(d^3)$. (Hoogeboom et al., 2019) point out that a fixed permutation matrix P limits flexibility. To fix this issue, they suggest using the QR decomposition where R is a rectangular matrix and Q is orthogonal. They suggest making Q orthogonal by using the Householder decomposition which our algorithm can speed up. Alternatively, one could use the SVD decomposition instead of the QR or PLU decomposition.

6. Code

During implementation of our algorithm, we found that Python did not provide an adequate level of parallelization. We therefore implemented our algorithm in CUDA to fully utilize the parallel capabilities of GPUs. To make the code widely accessible, we wrote accompanying Python code that allows using our algorithm in PyTorch (Paszke et al., 2019). For example: code that has a fully connected Neural Network which use 'nn.Linear' simply needs to change 'nn.Linear' to 'LinearSVD' after importing our code.

Further details can be found in 'pythoncode.zip' attached with our submission. We plan to publish a revised version of the code on Github to ease future use of our algorithm.

7. Conclusion

We showed that, in theory, the techniques from (Zhang et al., 2018; Mhammedi et al., 2017) allow speeding-up matrix operations. However, in practice, we demonstrated that the techniques are not fast enough on GPUs for moderately sized use-cases. We proposed a novel algorithm that remedies the issues with both algorithms from (Zhang et al., 2018), which is up to $27\times$ faster than the previous sequential algorithm. Our algorithm introduces no loss of quality, it computes the same thing as the previous algorithms, just faster. Our algorithm has two uses:

- It can speed up the algorithms from (Zhang et al., 2018), so much, that it is fast enough to speed up matrix inversion, matrix determinant, matrix exponential and the Cayley map.
- It can speed up previous work that employ the Householder decomposition as done in e.g. (Tomczak & Welling, 2016; Mhammedi et al., 2017; van den Berg et al., 2018; Hoogeboom et al., 2019).

References

Bischof, C. and Van Loan, C. The WY Representation for Products of Householder Matrices. *SIAM Journal on Scientific and Statistical Computing*, 1987.

Casado, M. L. Trivializations for Gradient-Based Optimization on Manifolds. In *NeurIPS*, 2019.

Dinh, L., Krueger, D., and Bengio, Y. NICE: Non-Linear Independent Components Estimation. In *ICLR (Workshop)*, 2015.

Golinski, A., Lezcano-Casado, M., and Rainforth, T. Improving Normalizing Flows via Better Orthogonal Parameterizations. In *ICML Workshop on Invertible Neural Networks and Normalizing Flows*, 2019.

Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. The Reversible Residual Network: Backpropagation Without Storing Activations. In *NIPS*, 2017.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative Adversarial Nets. In *NIPS*, 2014.

Hoogeboom, E., van den Berg, R., and Welling, M. Emerging Convolutions for Generative Normalizing Flows. In *ICML*, 2019.

Kingma, D. P. and Dhariwal, P. Glow: Generative Flow with Invertible 1x1 Convolutions. In *NeurIPS*. 2018.

Li, J., Li, F., and Todorovic, S. Efficient Riemannian Optimization on the Stiefel Manifold via the Cayley Transform. In *ICLR*, 2020.

Mhammedi, Z., Hellicar, A., Rahman, A., and Bailey, J. Efficient Orthogonal Parametrisation of Recurrent Neural Networks Using Householder Reflections. In *ICML*, 2017.

Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. Spectral Normalization for Generative Adversarial Networks. In *ICLR*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 2019.

Tomczak, J. M. and Welling, M. Improving Variational Auto-Encoders using Householder Flow. *arXiv preprint*, 2016.

Uhlig, F. Constructive Ways for Generating (Generalized) Real Orthogonal Matrices as Products of (Generalized) Symmetries. *Linear Algebra and its Applications*, 2001.

van den Berg, R., Hasenclever, L., Tomczak, J., and Welling, M. Sylvester Normalizing Flows for Variational Inference. In *UAI*, 2018.

Xue, J., Li, J., and Gong, Y. Restructuring of Deep Neural Network Acoustic Models with Singular Value Decomposition. 2013.

Zhang, J., Lei, Q., and Dhillon, I. Stabilizing Gradients for Deep Neural Networks via Efficient SVD Parameterization. In *ICML*, 2018.

8. Supplementary Material

8.1. Matrix Operations

Proof of Lemma 1.

Proof. (1) $|\det(W)| = |\det(U)| \cdot |\det(\Sigma)| \cdot |\det(V^T)|$ since the determinant of a product is equal to the product of determinants. But the determinant of orthogonal matrices U and V^T is ± 1, so $|\det(W)| = |\det(\Sigma)|$. But Σ is diagonal and all entries are positive so $|\det(W)| = \det(\Sigma) = \prod_{i=1}^{d} \Sigma_{ii}$.

(2) Let us check that $W \cdot W^{-1} = I$. Recall that $V^T = V^{-1}$ and $U^T = U^{-1}$.

$$\begin{aligned} WW^{-1} &= U\Sigma V^T V \Sigma^{-1} U^T \\ &= U\Sigma\Sigma^{-1}U^T \\ &= UU^T = I \end{aligned}$$

(3) The *singular* values are defined to be the entries of Σ, the diagonal matrix of the *Singular* Value Decomposition.

(4) If W is symmetric then $W = U\Sigma U^T$. Inserting this decomposition into the matrix exponential, we get

$$\begin{aligned} e^{U\Sigma U^T} &= \sum_{k=0}^{\infty} \frac{1}{k!}(U\Sigma U^T)^k \\ &= \sum_{k=0}^{\infty} \frac{1}{k!} U\Sigma^k U^T \\ &= U\left(\sum_{k=0}^{\infty} \frac{1}{k!}\Sigma^k\right) U^T \\ &= Ue^{\Sigma}U^T \end{aligned}$$

If W is skew-symmetric, $W^T = -W$, we get $W = U\Sigma U^T$ but for complex U.

(5) If W is symmetric then $W = U\Sigma U^T$. Inserting this decomposition into the Cayley transform yields

$$\begin{aligned} C &= (I - W)(I + W)^{-1} \\ &= (UIU^T - U\Sigma U^T)(UIU^T - U\Sigma U^T)^{-1} \\ &= U(I - \Sigma)U^T U(I + \Sigma)^{-1}U^T \\ &= U(I - \Sigma)(I + \Sigma)^{-1}U^T. \end{aligned}$$

If W is skew-symmetric, $W^T = -W$, we get $W = U\Sigma U^T$ but for complex U. □

Weight Decay. Weight decay is a regularizer that adds the Frobenious norm of a weight matrix to the loss function. The Frobenious norm is $||W||_F^2 = \sum_{ij} W_{ij}^2 = \sum_{i=1}^{d} \Sigma_{ii}^2$. If the SVD is given the norm can be computed in $O(d)$ instead of $O(d^2)$.

Compression with Truncated SVD. Neural Networks can be compressed by truncating the SVD of all weight matrices, see e.g. (Xue et al., 2013). This usually requires computing the SVD in $O(d^3)$ time. If the SVD is given, we only need to compute the largest k singular values and discard the remaining singular values/vectors. Computing the k largest singular values could be done by sorting all singular values in $O(d \lg d)$ time. It is possible to get $O(d)$ time by using the selection algorithm to get the k'th largest singular value in $O(d)$ time, then partition around the k'th largest element.

Pseudo-Inverse. The pseudo-inverse of a rectangular matrix $M \in \mathbb{R}^{m \times n}$ is usually computed by using the SVD to compute the reciprocal of the singular values. The time consumption is dominated by computing the SVD which, if $m > n$, takes $O(m^2 n)$ time. If the SVD is given it takes $O(n)$ time to compute the reciprocal of the singular values.

Condition Number. The condition number of a square matrix is $\kappa = \max_i \Sigma_{ii} / \min_i \Sigma_{ii}$, normally computed by first computing the SVD in $O(d^3)$ time. If the SVD is already given it can be computed in $O(d)$ time.

8.2. Proof of Theorem 2.

Theorem. *Algorithm 2 computes* $\frac{\partial L}{\partial X}$ *and* $\frac{\partial L}{\partial v_1}, ..., \frac{\partial L}{\partial v_d}$ *in* $O(d^2m)$ *time with* $O(d/m + m)$ *sequential matrix multiplications.*

Proof. **Correctness.** Our algorithm computes gradients by the same equations as (Zhang et al., 2018), so in most cases, we show correctness by clarifying how our algorithm computes the same thing, albeit faster.

Consider $\frac{\partial L}{\partial X}$ computed in Step 1:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial A_{d/m+1}} = P_{d/m}^T \cdots P_1^T \frac{\partial L}{\partial A_1}$$
$$= H_d^T \cdots H_1^T \frac{\partial L}{\partial A_1}. \qquad eq.\ (3)$$

This is the same as that computed in (Zhang et al., 2018).

Consider Step 2. Both $\frac{\partial L}{\partial \widehat{v}_j}$ and $\frac{\partial L}{\partial \widehat{A}_j}$ are computed as done in (Zhang et al., 2018). $\widehat{A}_{j+1}$ is computed using Equation (5) similar to backpropagation without storing activations, (Gomez et al., 2017), but using the fact that $\widehat{H}_j^T = \widehat{H}_j^{-1}$.

Time Complexity. In Step 1 the for loop performs d/m matrix multiplications. Due to the WY decomposition $P_i^T = (I - 2WY^T)^T = I - 2YW^T$ which can be multiplied on $\frac{\partial L}{\partial A_i} \in \mathbb{R}^{d\times m}$ in $O(dm^2)$ time since $W, Y \in \mathbb{R}^{d\times m}$. The computation is repeated d/m times, and take a total of $O(d^2m)$ time.

Step 2 line 14 performs two Householder matrix multiplications which take $O(dm)$ time, see Equations (5) and (6). In line 15 the gradient is computed by Equation (7), this sum also takes $O(dm)$ time to compute. Both computations on line 14 and 15 are repeated $d/m \cdot m$ times, see line 10 and line 13. Therefore, the total time is $O(d^2m)$.

Number of Sequential Operations. Step 1 performs $O(d/m)$ sequential matrix operations. Lines 13-16 of Step 2 perform $O(m)$ sequential matrix multiplications. Since each iteration of line 10-17 is run in parallel, the algorithm performs no more than $O(d/m + m)$ sequential matrix multiplications. □

Algorithm 3 Backwards Computation

1: **Input:** $A_1, ..., A_{d/m+1}, P_1, ..., P_{d/m}$ and $\frac{\partial L}{\partial A_1}$.
2: **Output:** $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial v_j}$ for all j where $H_j = I - 2\frac{v_j v_j^T}{||v_j||_2^2}$.
3:
4: // Step 1
5: **for** $i = 1$ **to** d/m **do sequentially**
6: $\quad \frac{\partial L}{\partial A_{i+1}} = P_i^T \frac{\partial L}{\partial A_i}$ eq. (4). ▷ $O(dm^2)$
7: **end for**
8:
9: // Step 2
10: **for** $i = 1$ **to** d/m **do in parallel**
11: $\quad$ Let $\frac{\partial L}{\partial \widehat{A}_1} = \frac{\partial L}{\partial A_i}$.
12: $\quad$ To ease notation, let $P_i = \widehat{H}_m \cdots \widehat{H}_1$.
13: $\quad$ **for** $j = 1$ **to** m **do**
14: $\quad\quad$ Compute $\widehat{A}_{j+1}, \frac{\partial L}{\partial \widehat{A}_j}$, eqs. (5) and (6). ▷ $O(dm)$
15: $\quad\quad$ Compute $\frac{\partial L}{\partial \widehat{v}_j}$ using $\widehat{A}_{j+1}, \frac{\partial L}{\partial \widehat{A}_j}$, eq. (7). ▷ $O(dm)$
16: $\quad$ **end for**
17: **end for**
18: **return** $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial A_{d/m+1}}$ and $\frac{\partial L}{\partial v_j}$ for all j.

8.3. Comparing Running Time

This subsection clarifies how the matrix exponential and the Cayley map was used in combination with the SVD technique from (Zhang et al., 2018). It also provides further details on the exact computations we timed in the experiment. These details were left out of the main article as they require the introduction of some notation regarding a *reparameterization function*.

Let $V \in \mathbb{R}^{d\times d}$ be a weight matrix and let ϕ be a function that reparameterizes V so $\phi(V)$ is orthogonal and we can perform gradient descent wrt. V. The Householder decomposition can be used to construct such a function ϕ, by letting the columns of V be Householder vectors and $\phi(V)$ be the product of the resulting Householder matrices.

There exist alternative ways of constructing ϕ which does not rely on the Householder decomposition. For example, the matrix exponential approach where $\phi_{exp}(V) = e^V$ and the Cayley map approach where $\phi_C(V) = (I - V)(I + V)^{-1}$ (Casado, 2019).

We record the joint time it takes to compute $\phi(V)X$ and the gradients wrt. to V and X for a dummy input $X \in \mathbb{R}^{d\times M}$. To simplify the gradient computation of V, we use a dummy gradient $G \in \mathbb{R}^{d\times M}$ st. the gradient wrt. V is $[\frac{\partial \phi(V)\cdot X}{\partial V}]^T G$. It might be useful to think of G as the gradient that arises by back-propagating through a Neural Network.

Both the dummy input and the dummy gradient have normally distributed entries $X_{ij}, G_{ij} \sim N(0, 1)$.

Implementation details. The parallel algorithm from (Zhang et al., 2018) halted for larger values of d. The failing code was not part of the main computation. This allowed us to remove the failing code and still get a good estimate of the running time of the parallel algorithm. We emphasize that removing the corresponding code makes the parallel algorithm faster. The experiments thus demonstrated that our algorithm is faster than a lower bound on the running time of the parallel algorithm.

8.4. Using the SVD to Compute Matrix Operations

This section requires first reading Section 4.1 and Section 4.2. Recall that we in Section 4.2 want to measure the time it takes to compute the matrix operation, the forward pass and the gradient computations. For example, with matrix inversion, we want to compute the matrix operation Σ^{-1}, the forward pass $V\Sigma^{-1}U^TX$ and the gradient computations wrt V, Σ, U, X.

The time of the forward pass and gradient computations is no more than two multiplications and two gradient computations, which is exactly two times what we measured in Section 4.1. We re-used those measurements, and add the time it takes to compute the matrix operation, e.g., Σ^{-1}.

Over Estimating the Time of Our Algorithm. The matrix exponential and the Cayley map require one orthogonal matrix instead of two, i.e., $U\Sigma U^T$ instead of $U\Sigma V^T$. The WY decomposition then only needs to be computed for U and not both U and V. By re-using the data we measure the time of two orthogonal matrices, this thus estimates an upper-bound of the real running time of our algorithm.