Faster Orthogonal Parameterization with Householder Matrices

Alexander Mathiasen [1] **Frederik Hvilshøj** [1] **Jakob Rødsgaard Jørgensen** [1] **Anshul Nasery** [1 2] **Davide Mottin** [1]

Abstract

Orthogonal matrices have been used in several Normalizing Flows (Tomczak & Welling, 2016; van den Berg et al., 2018).Orthogonal matrices are attractive since they are easy to invert and have Jacobian determinant one. Their main downside is the additional computational resources required to perform gradient descent. We identify a computational bottleneck for previous work on Householder matrices, and introduce a novel algorithm, FastH, which circumvents the bottleneck and is up to 29× faster than a previous method.

1. Introduction

Normalizing Flows (NF) are a type of generative model with several attractive properties like exact likelihood, fast sampling and substantial memory savings (Kingma & Dhariwal, 2018). They are capable of representing complicated distributions by transforming a simple base distribution $z \sim P_z$ through an invertible neural network f to attain a model distribution $f(z) \sim P_{model}$. The exact likelihood $p_{model}(x)$ can then be computed through a change of variable formula due to the invertibility of f.

This has motivated much research into invertible neural networks. However, the change of variables formula also require the computation of the Jacobian determinant of f. Previous research thus attempt to design invertible neural networks which also allow efficient computation of their Jacobian determinant. This makes orthogonal matrices very attractive, since they are easy to invert $U^{-1} = U^T$ and have unit Jacobian determinant $|\det(\partial Ux/\partial x)| = 1$.

Perharps unsurprisingly, much previous work adopt orthogonal matrices in their construction of Normalizing Flows (Tomczak & Welling, 2016; van den Berg et al., 2018; Hoogeboom et al., 2019; Golinski et al., 2019). However, the use of orthogonal matrices introduce one complication.

[1]Aarhus University [2]Indian Institute of Technology, Bombay. Correspondence to: Alexander Mathiasen <alexander.mathiasen@gmail.com>.

Second workshop on *Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models* (ICML 2020)

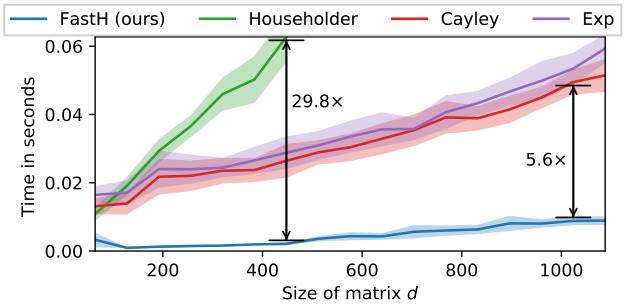


Figure 1. Comparison of previous approaches, see Section 4.

One needs to perform gradient descent wrt. weight matrices that are constrained to be orthogonal. Previous work solve this issue using different methods which can roughly be divided into three groups: matrix exponential, Cayley transform and Householder matrices.

The Householder method has the best time complexity for multiplying $U \cdot X$ where $U \in \mathbb{R}^{d \times d}$ is an orthogonal matrix and $X \in \mathbb{R}^{d \times m}$ is a mini-batch with m examples. In particular, the Householder methods take $O(d^2m)$ time compared to $O(d^3 + d^2m)$ time for both alternative methods. For $1 < m < d$ the Householder method is thus $O(d/m)$ times faster. Contrary to the superior time complexity, we find that methods based on the Householder method are often slower on GPUs in practice, see Figure 1.

We identify the algorithmic issue which causes the discrepancy between theory and practice. It turns out that the previous Householder methods entails the computation of $O(d)$ sequential inner products, which is ill-fit for parallel hardware like a GPU because the GPU cannot utilize all its cores. For example, if a GPU has 4000 cores and computes sequential inner products on 100-dimensional vectors, it can only utilize 100 cores simultaneously, leaving the remaining 3900 cores to run idle.

We introduce a novel algorithm, FastH, which increases core utilization, leaving less cores to run idle. FastH retains the desirable $O(d^2m)$ time complexity, while reducing the number of sequential operations. On a mini-batch of size $m > 1$, FastH performs $O(d/m + m)$ sequential matrix-matrix operations instead of $O(d)$ sequential vector-vector operations. In practice, FastH is faster than all previous methods, see Figure 1. Code: https://github.com/alexandermath/fasth.

2. Background

In this section, we sketch the method based on Householder matrices, and explain its computational bottleneck. A matrix $U \in \mathbb{R}^{d\times d}$ is orthogonal if $U^T = U^{-1}$. All orthogonal $d \times d$ matrices can be decomposed into a product of d Householder matrices $H_1, \dots, H_d$ (Uhlig, 2001):

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

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 all orthogonal matrices.

Multiplication. Normal matrix multiplication UX for matrices $U \in \mathbb{R}^{d\times d}, X \in \mathbb{R}^{d\times m}$ takes $O(d^2m)$ time. This is also true when U is a product of d Householder matrices. The product $UX = H_1 \cdots (H_{d-1}(H_d \cdot X))$ can be computed by d Householder multiplications. If done sequentially, as indicated by the parenthesis, 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, no more than computing UX normally.

However, the $O(d^2m)$ time complexity requires us to multiply each Householder matrix sequentially. Each Householder multiplication entails computing an inner product, see Equation (1), which means the multiplication UX requires the computation of $O(d)$ inner products sequentially. Such sequential computation of inner products is slow on parallel hardware like GPUs.

Our main contribution is a novel parallel algorithm, FastH, which resolves the issue with sequential inner products without increasing the time complexity. FastH 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, FastH is up to $29\times$ faster than the normal sequential Householder method, see Figure 1 and Section 4.1.

Mathematical Setting. 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.

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 as in Equation (1).

A simplified version of FastH 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:

$$P_i = H_{(i-1)\cdot m+1} \cdots H_{i\cdot m} \qquad i = 1, ..., d/m. \quad (2)$$

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 1.

Lemma 1. *(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 1 solves both issues outlined above.

Theorem 1. *Algorithm 1 computes*

$$H_1 \cdots H_d X$$

in $O(d^2m)$ time with $O(d/m + m)$ sequential matrix multiplications.

Algorithm 1 FastH Forward

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 $P_i = I - 2W_i Y_i^T$ by using Lemma 1. ▷ $O(dm^2)$
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.

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

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

Therefore, at termination, $A_1 = P_1 \cdots P_{d/m} X$. In Step 1, we used Lemma 1 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 1, each iteration takes $O(dm^2)$ time. Therefore, the total time of the d/m iterations is $O(dm^2 d/m) = O(d^2 m)$.

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 $O(dm^2 d/m) = O(d^2 m)$.

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 1. Since each iteration is run in parallel, the algorithm performs no more than $O(d/m + m)$ sequential matrix multiplications. □

Remark. Supplementary Material 8.1 extends this sections techniques to compute gradient, see Algorithm 2. For simplicity, this section had Algorithm 1 compute only A_1, however, in Algorithm 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. 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. The new algorithm 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. The number of sequential matrix multiplications $O(d/k+k)$ is minimized when $k = O(\sqrt{d})$. For a constant $c > 1$, we can find the best $k \in \{2, 3, ..., c\lceil\sqrt{d}\rceil\}$ by trying all $O(\sqrt{d})$ values. The search needs to be done only once and takes $O(\sqrt{d}(d^2 k + d^2 m)) = O(d^3 + d^{2.5} m)$ time. In practice, this time is negligible, e.g., on the hardware we describe in Section 4 it took less than $1s$ for $d = 784$.

4. Experiments

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 investigates the time it takes for FastH to perform a gradient descent update wrt. an orthogonal matrix. The time of FastH is compared against four alternative algorithms. The first two alternatives are methods based on the matrix exponential and the Cayley map respectively (Golinski et al., 2019). The next two alternatives are the sequential and parallel algorithms from (Zhang et al., 2018), which both rely on Householder matrices like FastH. Both articles open-sourced their implementations which we use in our experiments.[1][2]

The performance of the sequential algorithm is particularly interesting, because it is the same algorithm most previous work on Normalizing Flows adopt (Tomczak & Welling, 2016; van den Berg et al., 2018; Hoogeboom et al., 2019). The only difference is that (Zhang et al., 2018) implemented the algorithm in CUDA instead of PyTorch (Paszke et al., 2019) or TensorFlow (Abadi et al., 2015).

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.

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

Figure 2 depicts the running time on the y-axis, as the size of the $d \times d$ matrices increases on the x-axis. For $d > 64$, FastH is faster than all previous approaches. At $d = 64$ FastH is faster than all previous approaches, except the parallel algorithm. FastH is even faster at $d = 3000$ than the Sequential algorithm at $d = 400$. To put the matrix sizes into perspective, we mention that previous work employ, e.g., $d = 192$ in (Kingma & Dhariwal, 2018) or in $d = 784$ (Zhang et al., 2018).

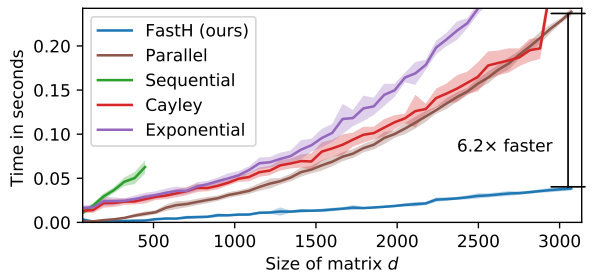


Figure 2. Running time of different algorithms for $d \times d$ matrices. FastH is fastest when $d > 64$. The sequential algorithm from (Zhang et al., 2018) crashed when $d > 448$.

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.

FastH takes the same asymptotic time as the sequential algorithm, however, it performs less sequential matrix operations, making it up to $29\times$ faster in practice. Since FastH 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, FastH 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.

Different Orthogonal Parameterizations. Previous work explore different approaches to orthogonal parameterizations, including methods based on Householder matrices (Mhammedi et al., 2017), the matrix exponential (Lezcano-Casado & Martínez-Rubio, 2019) and the Cayley map (Golinski et al., 2019).

(Golinski et al., 2019) raised a theoretical concern about the use of Householder matrices. The methods based on the matrix exponential and the Cayley map have desirable provable guarantees, which currently, it is not known whether the Householder decomposition possess. This might make it desirable to use methods based on the matrix exponential or the Cayley map instead of using methods based on Householder matrices. However, the methods based on matrix exponential and Cayley map use $O(d^3)$ time to construct the orthogonal matrix, which the Householder method circumvents. This allows Householder methods to perform faster multiplications. In particular, for a mini-batch $X \in \mathbb{R}^{d \times m}$, methods based on Householder matrices can compute the product in $O(d^2 m)$ time, $O(d/m)$ times faster.

Determinants and Matrix Decompositions. (Kingma & Dhariwal, 2018) propose to speed up determinant computations by using the PLU decomposition $W = PLU$ where P is a permutation matrix and L, U are lower and upper triangular. This allows 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 FastH can speed up.

6. Code

To make FastH widely accessible, we wrote a PyTorch implementation 'nn.Orthogonal' which can be used like 'nn.Linear'. Besides implementing the default 'forward' function, it also contains functions for inverse and log Jacobian determinant[3]. While implementing FastH, we found that Python did not provide an adequate level of parallelization. We therefore implemented FastH in CUDA to fully utilize the parallel capabilities of GPUs. Our code can be found at https://github.com/alexandermath/fasth.

7. Conclusion

We identified an algorithmic issue with the previous use of Householder matrices in Neural Networks. FastH mitigates the issue, and is up to $29\times$ faster in practice, without introducing any loss of quality. In other words, FastH computes the same thing as the previous algorithms, just faster. FastH can thus be used to speed up Neural Networks like (Tomczak & Welling, 2016; van den Berg et al., 2018; Hoogeboom et al., 2019) without any downsides.

[3]It just needs to return $0 = \lg |\det(\partial Ux/\partial x)|$.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

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

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.

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.

Lezcano-Casado, M. and Martínez-Rubio, D. Cheap Orthogonal Constraints in Neural Networks: A Simple Parametrization of the Orthogonal and Unitary Group. In *ICML*, 2019.

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

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.

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. 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 $A_1, \dots, A_{d/m+1}$, $P_1, ..., P_{d/m}$ 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 FastH 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

$$\frac{\partial L}{\partial A_{i+1}} = \left[\frac{\partial A_i}{\partial A_{i+1}}\right]^T \frac{\partial L}{\partial A_i} = P_i^T \frac{\partial L}{\partial A_i} \tag{3}$$

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

Step 2. 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 $\frac{\partial L}{\partial A_i}$.

Details. For completeness, we state pseudo-code in Algorithm 2, which we now explain with the help of Figure 3.

Figure 3a 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 (3). 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 $P_i = \widehat{H}_1 \cdots \widehat{H}_m$. 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:

$$\widehat{A}_{j+1} = \widehat{H}_j^{-1} \widehat{A}_j = \widehat{H}_j^T \widehat{A}_j \tag{4}$$

$$\frac{\partial L}{\partial \widehat{A}_{j+1}} = \left[\frac{\partial \widehat{A}_j}{\partial \widehat{A}_{j+1}}\right]^T \frac{\partial L}{\partial \widehat{A}_j} = \widehat{H}_j^T \frac{\partial L}{\partial \widehat{A}_j} \tag{5}$$

Algorithm 2 FastH Backward

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. (3). $\quad \triangleright\ O(dm^2)$
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 $P_i = \widehat{H}_1 \cdots \widehat{H}_m$.
13: $\quad$ **for** $j = 1$ **to** m **do**
14: $\quad\quad$ Compute $\widehat{A}_{j+1}$, $\frac{\partial L}{\partial \widehat{A}_j}$, eqs. (4) and (5). $\quad \triangleright\ O(dm)$
15: $\quad\quad$ Compute $\frac{\partial L}{\partial \widehat{v}_j}$ using $\widehat{A}_{j+1}$, $\frac{\partial L}{\partial \widehat{A}_j}$, eq. (6). $\quad \triangleright\ O(dm)$
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 $k = 1, ..., d$.

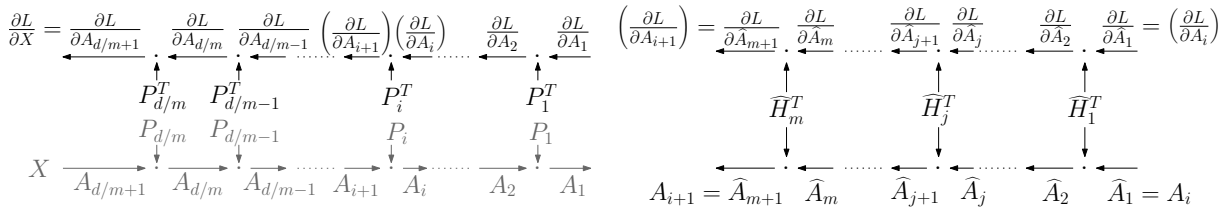
Figure 3b depicts how $\widehat{A}_2, ..., \widehat{A}_{m+1}$ and $\frac{\partial L}{\partial \widehat{A}_2}, ..., \frac{\partial L}{\partial \widehat{A}_{m+1}}$ can be computed given $\widehat{A}_1$ and $\frac{\partial L}{\partial \widehat{A}_1}$. Given $\widehat{A}_{j+1}$ and $\frac{\partial L}{\partial \widehat{A}_j}$, 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 (6). 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:

$$-\frac{2}{||\widehat{v}_j||_2^2}\sum_{l=1}^{m}(\widehat{v}_j^T a^{(l)})g^{(l)} + (\widehat{v}_j^T g^{(l)})a^{(l)} - \frac{2}{||\widehat{v}_j||_2^2}(\widehat{v}_j^T a^{(l)})(\widehat{v}_j^T g^{(l)})\widehat{v}_j \tag{6}$$

Theorem 2 states properties of Algorithm 2.

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

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



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

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

Figure 3. Computational graph of Step 1 and the i'th subproblem in Step 2 from Algorithm 2.

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

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial A_{d/m+1}} = P_{d/m}^T \cdots P_1^T \frac{\partial L}{\partial A_1}$$
$$= H_d^T \cdots H_1^T \frac{\partial L}{\partial A_1}. \qquad eq.\ (2)$$

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 (4) 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 $O(d^2 m)$ time.

Step 2 line 14 performs two Householder matrix multiplications which take $O(dm)$ time, see Equations (4) and (5). In line 15 the gradient is computed by Equation (6), 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 $O(d^2 m)$.

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. □

8.2. 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 FastH is faster than a lower bound on the running time of the parallel algorithm.