3D Computer Vision
Project Report

Author: Paul Hilt
TA: Jakob Kruse
Lecturer: Prof. Rother

September 13, 2020

1 Preliminaries

1.1 The Householder Transformation

The Householder transform is the matrix multiplication $T(x) = Hx$ of a vector $x \in \mathbb{R}^n$ and a matrix H. The resulting vector is a orthogonal reflection of x on a hyperplane through the origin.
Letting this hyperplane be defined by its normal vector $v \in \mathbb{R}^n$, H can be constructed by

$$H = I - \frac{2}{v^T v} v v^t. \tag{1}$$

Here $v^T v$ describes the inner product, vv^T is the outer product and I is the identity matrix. If v has unit length the equation simplifies to $H = I - 2vv^t$ and the reflection property can be seen using

$$Hx = x - 2vv^t x = x - 2v\langle v, x\rangle = (x - v\langle v, x\rangle) - v\langle v, x\rangle. \tag{2}$$

Here $x - v\langle v, x\rangle$ removes the part of x that is in the direction v, therefore projecting x on the hyperplane $v^\perp$. The last term "folds" the projection by the same part to the other side of $v^\perp$.
Some important Properties of the Householder transformation are

- it is symmetric, therefore $H = H^T$
- it is orthogonal, therefore $H^T H = I$
- from the property of linearity and orthogonality it follows that the jacobian determinant of $T(\cdot)$ equals to $det(DT(\cdot)) = \pm 1$
- the product of multiple Householder matrices is an orthogonal matrix.

Because the transformation is linear, the jacobian matrix of that transformation is $DT(x) = H$. From the orthogonality of H follows that its absolute determinant $det(H) = \pm 1$. This fact is important for the change of variables formula for integrals explained in the next section.

1.2 Change of Valiables

For a differentiable map $\phi(u) = v$ a infinitesimal change du in u relates to a change dv in v by $dv = |det(D\phi(u))|du$. Here $D\phi(u)$ is the jacobian matrix of the map ϕ with respect to the input u, $|\cdot|$ is the absolute operator and $det(\cdot)$ is an operator returning the determinant of a matrix.

Thm:

Let $U \subseteq \mathbb{R}^n$ and the function $\phi : U \to \mathbb{R}^n$ be injective and differentiable. Then for any real-valued, compactly supported, continuous function f

$$\int_{\phi(U)} f(v)dv = \int_U f(\phi(u))|det(D\phi(u))|du \tag{3}$$

This equation is referred to as the change of variables formula [1].

1.3 Normalizing Flows

Normalizing Flows make intrinsic use of the change of variables formula in order to obtain probability densities of transformed samples from a base distribution whose probability density is known. In other words, given a random variable V with probability density p_V and the diffeomorphic mapping $\phi(V) = U$. Using the change of variables p_U can be found by

$$P(U \in S) = \int_S p_U(u)du = \int_{\phi^{-1}(S)} p_V(v)dv \tag{4}$$

where $P(U \in S)$ denotes the probability that U takes a value in some set S. Using eq. (3) gives

$$P(U \in S) = \int_S p_V(\phi^{-1}(u))|det(D\phi^{-1}(u))|du \tag{5}$$

combining eq. (4) and eq. (5) and differentiating yields

$$p_U(u) = p_V(\phi^{-1}(u))|det(D\phi^{-1}(u))| \tag{6}$$

This means of course, that for a concatenation of N diffeomorphic mappings where $U_N = V$ and $U_{i-1} = \phi_i(U_i)$, $i = N...1$

$$p_{U_0}(u_0) = p_V(\phi_0^{-1}(\phi_1^{-1}(...\phi_N^{-1}(u_{N-1})))) \prod_{i=1}^{N} |det(D\phi_i^{-1}(u_i))| \tag{7}$$

Optimizing the function parameters of the transformations ϕ_i by e.g. a gradient method leads to a class of generative models called Normalizing Flows. Usually Normalizing

Flows are used to turn a unimodal and simple probability distribution into a more expressive one.
E.g. in Reinforcement Learning it makes sense to learn a simple action policy such as a Normal distribution where the reparameterization trick can be applied in order to backpropagate through the sampling process. However oftentimes a more optimal distribution is one that favors exploration of actions at multiple modes. In that case it makes sense to apply a normalizing flow on the samples of the base Normal distribution to arrive at a more expressive one. The authors in [2] provide a good review of Normalizing Flows and its applications.

1.4 The WY Representation

The WY representation as introduced in [3] is a representation of a product of k Householder matrices

$$Q_k = \prod_{i=1}^{k} H_i \tag{8}$$

where $H_i = I + u_i v_i^T$ with $u = -2v$ and v is has a unit L2 norm.
Eq. (9) can be written in the form

$$Q_k = I + W_k Y_k^T \tag{9}$$

where W_k and Y_k are m-by-k matrices. Obviously $W_1 = u_1$ and $Y_1 = v_1$. An iterative method to obtain W_{k+1} and Y_{k+1} from W_k and Y_k goes as follows:

$$\begin{aligned}
Q_k = Q_{k-q} H_k &= (I + W_{k-1} Y_{k-1}^T)(I + u_k v_k^T) && (10)\\
&= I + W_{k-1} Y_{k-1}^T u_k v_k^T + u_k v_k^T + W_{k-1} Y_{k-1}^T && (11)\\
&= I + (W_{k-1} Y_{k-1} + I) u_k v_k^T + W_{k-1} Y_{k-1}^T && (12)\\
&= I + W_{k-1} Y_{k-1}^T + Q_{k-1} u_k v_k^T && (13)\\
&= I + [W_{k-1}, Q_{k-1} u_k][Y_{k-1}, v_k]^T && (14)\\
&= I + W_{k-1} Y_{k-1} (u_k v_k^T + I) + u_k v_k^T && (15)\\
&= I + W_{k-1} Y_{k-1}^T H_k + u_k v_k^T && (16)\\
&= I + [W_{k-1}, u_k][H_k Y_{k-1}, v_k]^T && (17)
\end{aligned}$$

Here $[\cdot, \cdot]$ denotes the column wise concatenation of two matrices. Note, that the equalities in eq. (14) and eq. (17) hold due to the the definition of matrix multiplication and addition.
This result gives rise to the two equivalent update methods

- Method 1:

$$\begin{aligned}
W_k &= [W_{k-1}, Q_{k-1} u_k] && (18)\\
Y_k &= [Y_{k-1}, v_k] && (19)
\end{aligned}$$

- Method 2:

$$W_k = [W_{k-1}, u_k] \tag{20}$$
$$Y_k = [H_k Y_{k-1}, v_k] \tag{21}$$

2 Introduction

Householder matrices are attractive because a product of Householder matrices can be used to generate an orthogonal matrix with learnable parameters where the othogonality of the resulting matrix holds by construction. Orthogonal matrices can be seen as a generalization of permutation matrices, because they neither change lenghts of vectors nor relative angles between vectors. In invertible neural networks orthogonal matrices are commonly used for "soft" learned permutations.
It is important to note that orthogonal matrices are not able to change densities (results from the fact that the absolute determinant jacobian is 1 and eq. (6)), therefore only orthogonal transformations are not sufficient to represent a Normalizing Flow that changes probability densities.
The naive method to compute a product of K, m-by-m Householder matrices has time complexity $O(m^2K)$. The authors in [4] propose a method based on the WY representation for Householder products that retains this complexity but reduces the amount of involved sequential operations, using accelerated GPU computing.

3 Method

The goal is to define a fast algorithm that solves

$$U = \prod_{i=1}^{K} H_i. \tag{22}$$

Using the WY representation, any sub product of s matrices in eq. (22) can be written as

$$P_i = I - W_i Y_i^T = \prod_{j=si}^{s(i+1)} H_j \ , \qquad i = 1, 2, ..., \left\lfloor \frac{K}{s} \right\rfloor \tag{23}$$

where W_i and Y_i are obtained using method 2 in section 1.4. The parameter s defines a stride that controls the amount of parallel operations. If $\frac{K}{s}$ is not integral and letting $e = \lfloor \frac{K}{s} \rfloor$, $E = e + \left\lfloor \frac{K-es}{2} \right\rfloor$ then the remaining multiplications can be obtained by

$$P_i = I - W_i Y_i^T = \prod_{j=2(i-e)}^{2((i-e)+1)} H_{j+es} \ , \qquad i = \left\lfloor \frac{K}{s} \right\rfloor + 1, \left\lfloor \frac{K}{s} \right\rfloor + 2, ..., E \tag{24}$$

if $\frac{K-es}{2}$ is not integral then

$$P_E = P_E H_K \tag{25}$$

and finally

$$U = \prod_{i=1}^{K} H_i = \prod_{i=1}^{E} P_i. \tag{26}$$

The important property to notice here is that all P_i can be obtained in parallel and therefore can be seen as one operation concerning the time complexity. Using method 1 in section 1.4 has higher time complexity because it is rich in matrix multiplication opposed to method 2 which is merely one outer product per step k (calculating H_k from its vector representation). It seems that reducing s results in more parallel operations and is therefore faster but a reduction of s also results in more sequential operations in eq. (26) since it produces more P_i. The optimal value for s also depends on the amount of memory of the GPU and has therefore no strict analytical solution.

4 Experiments and Results

The method in section 3 has been implemented in python under heavy use of the pytorch library which allows for accelerated GPU computing. Using advanced indexing methods, the parallelizations in eq. (23) and eq. (24) can be achieved, relying on the parallelism induced by optimizations of batched matrix multiplication in pytorch. The method was tested and evaluated against the naive method of multiplying Householder matrices on a GeForce RTX 2080 Ti GPU under full workload.
The stride parameter was cross validated for the value grid $s = [1..15]$,
$m = [10, 20, 30, ..., 550]$ and $K = m$ with 100 iterations. There is some dependence between the "fastest" stride s' and m but this dependence seems not to be linear which is why it is difficult to find an optimal s for a given m. For the range of m that was used in the test, s' was always between 2 and 12. A good default value is $s' = 2$. If m is so large, that there can be no parallelism because it would not fit on the GPU than $s' = 1$ is always the fastest which induces no additional batching and therefore no additional parallelism.

5 Conclusion

The authors in [4] set the parameter s to the number of samples in the mini batch or columns in the matrix, that U is multiplied with and constraint their method to $K = m$. Further they assume $\frac{K}{s}$ is integral therefore they do not need eq. (25-27). There is no apparent reasoning why s should be dependent on the mini batch size. The method implemented in this work has no constraints on s or K. However a empirical evaluation for an optimal value for $s = s'$ shows that $s' = 2$ is a good value.
Method 1 in section 1.4 has been implemented as well. While it has the advantage of

producing the intermediate vale $Q_k = P_i$ which saves the calculation of P_i, it showed to be slower than method 2 because of its inherent matrix multiplications.
I could not find an API where the optimization methods in pytorch's matrix multiplication are explained. However in some blogs and posts it is mentioned, that pytorch always uses the full available GPU space for batched (easy to parallelize) operations such as matrix multiplication. To strengthen that assumtion, an explicit parallelization method, that uses pytorch streams was implemented and showed to be slower than the batched matrix multiplications.
The authors in [4] also provide a CUDA implementation with python bindings of their method but this library could not be imported because of clashing CUDA versions. Building it from sources also failed, probably due to clashing nvcc versions.

References

[1] Wikipedia contributors, "Substitution for multiple variables — Wikipedia, the free encyclopedia," 2020, [Online; accessed 13-Sptember-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Integration_by_substitution#Substitution_for_multiple_variables

[2] I. Kobyzev, S. Prince, and M. Brubaker, "Normalizing flows: An introduction and review of current methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 11, 2020. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2020.2992934

[3] C. Bischof, Christian H.; Van Loan, "The wy representation for products of householder matrices," *computer science; technical report*, 1985. [Online]. Available: https://ecommons.cornell.edu/handle/1813/6521

[4] J. R. J. A. N. D. M. Alexander Mathiasen, Frederik Hvilsh, "Faster orthogonal parameterization with householder matrices," 2020. [Online]. Available: https://invertibleworkshop.github.io/accepted_papers/pdfs/10.pdf