Reproducibility report: Walsh-Hadamard Variational Inference for Bayesian Deep Learning

David Nabergoj*

February 1, 2021

1 Introduction

In this report, we assess the reproducibility of the paper “Walsh-Hadamard Variational Inference for Bayesian Deep Learning” [1]. Our results are based on multiple readings of the original paper and its supplementary material, re-runs of the described experiments, as well as some key referenced literature [2, 3, 4, 5, 6]. This reproduction is not based on any published code, except the CUDA implementation of the fast Walsh-Hadamard transform by the paper’s authors. Our implementation of the proposed method, experiments and other files are available at https://github.com/davidnabergoj/WHVI.

1.1 Brief review of the paper

The authors propose Walsh-Hadamard Variational Inference (WHVI), where weight matrices in Bayesian neural networks are efficiently reparameterized to allow for linear space complexity and log-linear time complexity when transforming an input vector with a WHVI layer. The key idea is that weight matrices can be efficiently sampled by

$$\widetilde{\mathbf{W}} = \mathbf{S_1}\mathbf{H}\text{diag}(\widetilde{\mathbf{g}})\mathbf{H}\mathbf{S_2}, \quad \widetilde{\mathbf{g}} \sim q(\mathbf{g}). \tag{1}$$

Here, $\widetilde{\mathbf{W}}$ is a $D \times D$ weight matrix sample, $\mathbf{S_1}$ and $\mathbf{S_2}$ are deterministic diagonal matrices whose entries need to be optimized, $\mathbf{H}$ is the Walsh-Hadamard matrix, and $\widetilde{\mathbf{g}}$ is a sample from the distribution q. The variational posterior distribution q is a multivariate normal distribution with a diagonal covariance matrix, i.e. $q(\mathbf{g}) = \mathcal{N}(\mu, \mathbf{\Sigma})$.

This approach offers an advantage over other approaches like mean-field Gaussian variational inference, because it requires $O(D)$ instead of $O(D^2)$ parameters to represent weight matrices of size $D \times D$. Furthermore, the matrix-vector product $\mathbf{Hx}$ can be computed in $O(D \log D)$ time and $O(1)$ space using the in-place version of the Fast Walsh-Hadamard transform (FWHT). The reduced number of parameters causes the KL term to be less dominant during model training, which is otherwise a common problem in Bayesian deep learning. The described approach supports matrices of size $D \times D$ where $D = 2^p$ for $p > 0$ in its most basic form, however it is extended to support matrices of arbitrary size by concatenating smaller square matrices.

WHVI is applied to a toy example, several regression data sets, and is also tested on image classification tasks using Bayesian convolutional neural networks.

1.2 Reproducibility goals

Our main goal is to implement all required procedures for WHVI and run the described experiments.

We first try to reproduce results for the toy univariate regression example in section 3.1 of the original report. This primarily means obtaining qualitatively similar uncertainty estimates. We then focus on the regression data sets, as listed in Table 3 of the original report. The goal is to obtain similar WHVI test error and WHVI test MNLL (mean negative log-likelihood) estimates, both the mean and the standard deviation.

Due to long training times for Bayesian neural networks, complex convolutional neural network architectures, and the large number of parameters ($\sim$ 2.3M), we do not consider image classification experiments in this report. We believe they are not crucial to assessing the quality of the proposed approach, because linear layers (not involving convolution operations) are already evaluated using standard regression data sets, whereas the authors report that using WHVI for convolutional filters does not yield interesting results due to the small number

*University of Ljubljana, david.nabergoj@student.uni-lj.si

of parameters. We focus only on mean-field WHVI and not on the version involving normalizing flows, which are mentioned as an extension to the proposed method.

We attempt to reproduce the findings regarding WHVI inference time to a smaller degree, because the workstation used in the experiments of the original paper is significantly more powerful than the one we used in this reproduction study. We compare the speed of matrix multiplication and FWHT on the CPU and the GPU, but not the experiments regarding energy consumption. We test the method across different random seeds to empirically assess stability and convergence.

1.3 Report structure

We describe our implementation of the proposed method in Section 2. We compare our results to those in the original paper in Section 3, where we discuss the predictive quality We compare our measurements of compute performance to the original ones in Section 4. In Section 5, we conclude the report with an overall reproducibility assessment provide some suggestions for improvement.

2 Implementation details

In this section, we describe our implementation of WHVI and list assumptions for parts which were not described in detail in the original paper. We discuss the core classes, FWHT, setting the prior, and initializations for parameters.

2.1 Core classes

We implemented WHVI in PyTorch [7], because it is also used in the original paper. The most important part is the `WHVISquarePow2Matrix` class, which contains as parameters the elements of $\mathbf{S}_1$, $\mathbf{S}_2$, and the posterior mean μ.

The authors do not state how they represent Σ. We do not directly use the elements of the diagonal covariance matrix Σ, but instead use a parameter vector ρ with D elements, which corresponds to Σ via a softplus transformation: $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_D)$ where $\sigma_i = \ln(1 + \exp(\rho_i))$. We choose the softplus transformation, because it was used in the seminal work on variational inference in neural networks [3]. The advantage is that we can optimize ρ_i across the real line and disregard positivity constraints in the gradient-based optimization, then ensure non-negativity via a simple transformation. We acknowledge that the referenced work uses softplus for individual elements of the weight matrix, whereas we use it for elements of vector $\mathbf{g}$, which is indirectly related to the weight matrix via Equation 1.

To transform a batch of input vectors, a weight matrix is sampled from the posterior according to Equation 1 and multiplied by a matrix whose rows are vectors from that batch. We implemented two options for this transformation. In the first, we sample the weight matrix directly according to Equation 1. In the second, we use the local reparameterization trick to sample the matrix-vector product $\mathbf{Wx} \in \mathbb{R}^D$ according to Equations 2 and 3.

$$\mathbf{Wx} = \overline{\mathbf{W}}(\mu)\mathbf{x} + \overline{\mathbf{W}}(\mathbf{\Sigma}^{1/2}\epsilon)\mathbf{x},\ \epsilon \sim \text{N}(\mathbf{0}, \mathbf{I}_D), \quad (2)$$

$$\overline{\mathbf{W}}(u) = \mathbf{S}_1\mathbf{H}\text{diag}(\mathbf{u})\mathbf{HS}_2 \quad (3)$$

We use the second option as it is not too slow when compared to the first, but has the advantage of decreasing the variance of stochastic gradients. For example, the number of epochs per second decreases from 190 to 155 on the toy dataset from Section 3.1 of the original paper.

The `WHVIStackedMatrix` class represents matrices of arbitrary size. It identifies how many smaller matrices of type `WHVISquarePow2Matrix` need to be stacked together to allow for the desired matrix multiplication, as well as the necessary padding of the input vector (see Algorithm 1 in the original report). These smaller matrices are stored in a `ModuleList` container as an attribute of `WHVIStackedMatrix`. When transforming a batch of input vectors, we generate one sample for each small matrix and concatenate these samples into a large matrix. The inputs are padded as necessary.

As a special case, we also implement the `WHVI-ColumnMatrix` class, which transforms a batch of input vectors with a single element into a batch of output vectors with many elements. It includes a smaller `WHVISquarePow2Matrix`, which is sampled at input transformation time and reshaped into a column. The last few elements of this column may be removed to accommodate for the desired output size. This method is recommended in the paper and indeed reduces the number of parameters from $O(D)$ to $O(\sqrt{D})$, while also reducing the single-vector transform time complexity from $O(D \log D)$ to $O(\sqrt{D} \log D)$. Note that by transposing the sampled matrix, this class can be used to map many inputs to a single output.

Finally, we create a `WHVILinear` layer class, which automatically selects the appropriate matrix based on the desired input and output dimensionalities. This is analogous to the traditional `Linear` layer in PyTorch, but also includes the computation of KL divergence $D_{\mathrm{KL}}\left(q(\mathbf{g}|\mu,\Sigma)\,||\,p(\mathbf{g})\right)$ from the prior p to the variational posterior q. The KL divergence of a `WHVILinear` layer is the sum of KL divergence terms for all of its descendants of type `WHVIStackedMatrix`.

2.2 Fast Walsh-Hadamard transform

An important contribution of the paper is the use of FWHT [4], which allows for log-linear vector transformation time. We implemented the transform in Python and C++, both implementations can be used on the CPU and the GPU. We also adapted and tested a CUDA kernel implementation, which is considerably faster than the Python and C++ implementations on the GPU. The performance is described more thoroughly in Section 4 with additional comparisons to regular matrix multiplication.

2.3 Priors and parameter initializations

According to the supplementary material for the original paper, we use a zero-mean prior with fully factorized covariance $\lambda\mathbf{I}$ for a particular layer, i.e. $\mathcal{N}(\mathbf{0}, \mathrm{diag}(\lambda,\ldots,\lambda)$ for a chosen $\lambda > 0$.

We believe that the original paper should describe the choices of prior variances in more detail. Currently, the supplement seems to suggest that a constant $\lambda = 10^{-5}$ was used in all layers of the deep Bayesian networks, but this is likely not the case. By reconsidering the statement in the supplement, we may interpret it as putting a low prior covariance on the last layer and possibly higher ones on the previous layers. Good choices of λ are thus essential at each layer separately.

The authors did not describe the initialization of $\mathbf{S}_1$, $\mathbf{S}_2$, Σ. We draw initial elements of $\mathbf{S}_1$ and $\mathbf{S}_2$ i.i.d. from $\mathrm{N}(0, 0.01)$ and initial elements of ρ i.i.d. from $\mathrm{Uniform}(-3,-2)$.

3 Model testing

In this section, we test our WHVI implementation, specifically the inference and predictive quality of Bayesian deep neural networks that use WHVI linear layers. We first focus on the toy example from Section 3.1 and then discuss regression experiments from Section 3.2 of the original paper.

3.1 Toy example

We consider the toy example from Section 3.1 in the original paper. The authors do not provide an explicit formula for the function. To replicate the experiment as closely as possible, we look at the plot and roughly observe some (x, y) pairs – extrema and points between them. We then use polynomial interpolation with a Vandermonde matrix to obtain a similar-looking function. This function is visualized in Figure 1.

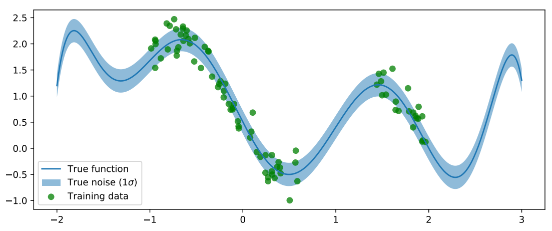

Figure 1: Polynomial approximation to the toy function in the original paper. The exact form of this function up to two decimals is $f(x) = 0.50 - 3.45x + 1.14x^2 + 4.36x^3 - 0.93x^4 - 1.77x^5 + 0.39x^6 + 0.22x^7 - 0.06x^8$. The noise is normally distributed with $\sigma = \sqrt{\exp(-3)}$.

The cosine activation function is no longer suitable given this different form of the function. We check this by training a non-Bayesian model, which has the exact same architecture as the WHVI one (two hidden layers with 128 hidden units). We found that the sigmoid activation performs better and decided to use it in the WHVI experiment instead. A comparison of the two non-Bayesian models with different activations can be seen in Figure 2.

We trained the model with WHVI layers and found that the results depend heavily on initial λ values in each layer as well as the initial scale σ_0 of the Gaussian likelihood. The result is visualized in Figure 3.

Many parameter choices resulted in slow convergence and significantly underestimated uncertainty. We suggest setting λ to values close to 1 and set σ_0 to be sufficiently larger than zero.[1] In particu-

[1]We base this on empirical observations for this toy exam-

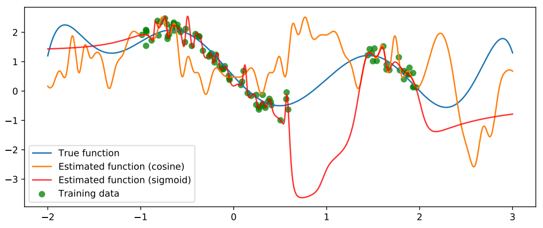

Figure 2: Comparison of the cosine and the sigmoid activation in a non-bayesian network. We decided to use sigmoid instead of cosine, because it seemed to capture the function more nicely on the left chunk of the data. The overfitting is deliberate and shows that a neural network has difficulties with cosine, but not with sigmoid as its activation.

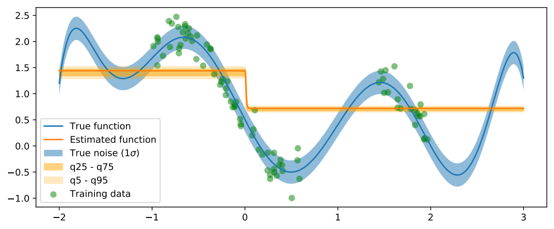

Figure 3: Estimated function and the corresponding uncertainty. The dark-shaded region represents quantiles $q_{25} - q_{75}$, whereas the bright-shaded one represents $q_5 - q_{95}$. The underlying function is not modeled well. The model remains stuck in this local minimum even if we increase the number of epochs from 50000 to 200000.

lar, our choices for modeling the toy data set were $\lambda_1 = 1, \lambda_2 = 2.5, \lambda_3 = 5, \sigma_0 = 5$. When setting λ_3 to 10^{-5} as proposed in the paper, the predicted value was just the mean. We observed that the KL term in ELBO would often be much larger than the MNLL term at the beginning of the optimization. Conversely, the MNLL term would get stuck at sub-optimal value and be unable to escape this local minimum even after several thousand epochs. After optimization, we observed several times that all but one pair of μ and Σ parameters were almost exactly equal to zero and λ. This suggested that the KL term was too dominant and drew the variational posterior too close to the prior.

ple, but it should theoretically not affect the sampled weight matrix too much. This is because the weight matrix is computed using optimized matrices $\mathbf{S}_1$ and $\mathbf{S}_2$ which should adjust for suitable results.

The authors also state that they obtained a model with 1541 parameters for the toy example, however we obtained one with 1537 parameters. We were not able to identify the missing parameters and we believe that they are not hyperparameters, because the authors refer to hyperparameters separately in a supplement section (i.e. not as "parameters"). We believe that these missing parameters are the primary reason for the observed problems in our experiments, since experimenting with a wide variety of different settings and initializations for $\lambda_i, \sigma_0, \rho, \mathbf{S}_1$, and $\mathbf{S}_2$ did not improve the results.

As an attempt to improve performance, we added bias columns (to be optimized) to WHVI layers. This addition was not enough to obtain the desired results and also introduced a considerably large number of additional parameters. We suggest further research regarding parameter initialization. A possible solution would be to explore hyperpriors for μ and ρ to allow for greater flexibility in the sampled weights. We found the choice of random seed not to improve results by observing the fit with seeds $5000k, k = \{1, \ldots, 10\}$. To conclude, we were not able to reproduce the uncertainty estimates, presented in the original paper.

3.2 Regression experiments

This section refers to experiments in section 3.2 and Table 3 of the original paper. We use the experimental setup as described in section 3.2 of the original paper and section D.1 of the supplement. Again, the paper does not state the prior covariances λ except for the last layer, where $\lambda = 10^{-5}$. We choose $\lambda = 3$ for all previous layers.

The comparison between our results and the ones from the paper are presented in Table 1.

We believe that the discrepancies are due to the missing parameters (see Subsection 3.1) and possibly due to different values of λ. It is also possible that the authors invested more time into hyperparameter tuning to achieve such results, however we were unable to do the same given the considerable time and computational resources required for the experiments. We do not evaluate WHVI performance further, because it is clear that the missing parameters, as well as the choice prior and parameter initializations are causing poor results.

	Original RMSE	Our RMSE	Original MNLL	Our MNLL
Boston	3.14 (0.71)	11.97 (0.64)	4.33 (1.80)	24188 (15730)
Concrete	4.70 (0.72)	19.15 (1.21)	3.17 (0.37)	47347 (45087)
Energy	0.58 (0.07)	15.54 (3.36)	2.00 (0.60)	45671 (23931)
KIN8NM*	0.08 (0.00)	1.74 (NA)	-1.19 (0.04)	11343 (NA)
Yacht	0.69 (0.16)	16.46 (2.25)	1.80 (1.01)	3719 (956)

Table 1: WHVI regression experiments results. RMSE and MNLL are computed on test data sets. The result format is *mean (std)*, where the sample mean and standard deviation are computed across eight independent random 9:1 train-test splits of the data sets. We used $\lambda_1 = \lambda_2 = 3, \lambda_3 = 10^{-5}$ on all data sets. We only ran KIN8NM for one iteration that took 35 hours, however it was clear from this result and the others that the described method will not achieve the necessary results. For this reason, we also skipped the Naval, Power plant, and Protein data sets.

4 Performance testing

In this section, we discuss the computational efficiency of our implementation of WHVI and FWHT. The regression experiments were run on Linux Ubuntu 20.04 with an Intel I7-6700K CPU and a GeForce GTX 970 GPU.

It is not clear from the original paper to what extent the in-place version of FWHT was utilized. We found that using in-place operations caused problems in the PyTorch automatic differentiation engine so that the computed gradient was incorrect. We consequently cloned the input batch before using the transform, but this did not noticeably decrease processing speed.

The CUDA implementation of FWHT was not explained in the original paper or the supplement. We adapted an implementation of the kernel from the original paper[2] and used it in our experiments. We suggest the paper by Bikov and Bouyukliev (2018) [8] for a clear and concise FWHT implementation reference, as well as the detailed implementation strategies, described by Arndt (2010) [9].

The following benchmarks were run on Windows 10 with an AMD Ryzen 9 3900X CPU and a GeForce RTX 2070S GPU. We noticed that using a non-vectorized Python/C++ CPU implementation of FWHT yielded consistently slower results than matrix multiplication on the CPU with no sign of change. However, a vectorized Python implementation yields the same results as those in the original paper, i.e. the break-even point occurs at approximately $D = 2^{11}$. The results are shown in Figure 4.

[2]See the paper's repository at https://github.com/srossi93/vardl and a different implementation of the transform at https://github.com/HazyResearch/structured-nets.

5 Conclusion and discussion

We implemented Walsh-Hadamard variational inference as proposed in the original paper and its supplement. The main contribution of the method is the significantly reduced number of parameters needed to represent a weight matrix in a Bayesian neural network and the use of the fast Walsh-Hadamard transform for time and space-efficient computations.

We were not able to reproduce the findings in the original paper with our implementation. We believe that this is primarily because of missing parameters, which we were unable to identify. Another possibility is the choice of hyperparameters, parameter initialization strategies, and prior selection, which could have been discussed somewhat more in the original paper. We realize that some of these details and other common knowledge in variational inference might have been omitted from the published paper and the supplement due to space constraints.

We contacted the authors regarding the issues, and they were kind enough to link the Github repository (https://github.com/srossi93/vardl) with some supporting code for the paper. We would like to express our gratitude for this help. While the linked repository contains code for some methods, it is still in development and yet to be thoroughly documented. Unfortunately, the notebooks for the studied paper are also not available. Due to time constraints, we could not study the code in detail at the time of writing. However, after checking some related classes, we saw some procedures that were not described in the paper. For example, there is support for biases in the WHVI layers, but because there are no notebooks, it is unclear which of the experiments used biases and which did not. We will

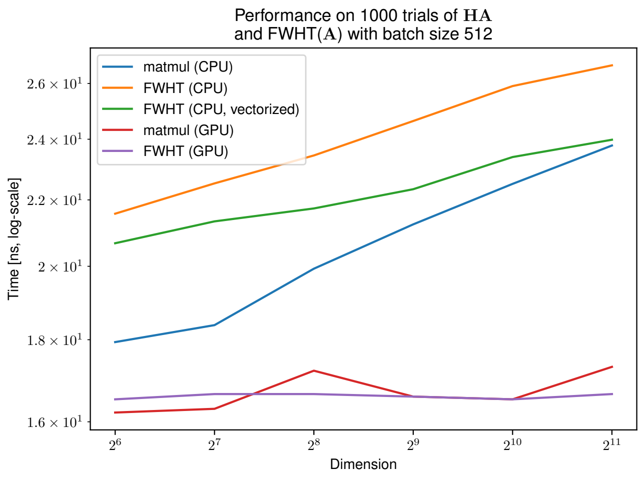

Figure 4: FWHT and matrix multiplication performance comparison on the CPU and GPU. Results are based on 1000 transformations of 512 vectors with standard normal random entries, each vector consisting of D elements (shown as "Dimension" on the x-axis). The vectors are arranged into the matrix $\mathbf{A}$. The vectorized CPU implementation beats matrix multiplication at around $D = 2^{11}$ in compute time, but the non-vectorized implementation is consistently worse. Both operations are much faster on the GPU, and FWHT outperforms matrix multiplications at moderate values of D. These results agree with the ones in the supplement for the original paper.

update our code and this report once the authors publish additional materials for the paper. In conclusion, we believe that WHVI is a very interesting method and probably works well given the amount of evidence in the original paper, but needs further clarification with respect to some important missing details.

References

[1] Simone Rossi, Sebastien Marmin, and Maurizio Filippone. Walsh-Hadamard Variational Inference for Bayesian Deep Learning. *arXiv preprint arXiv:1905.11248*, 2019.

[2] Quoc Viet Le, Tamás Sarlós, and Alexander Johannes Smola. Fastfood: Approximate Kernel Expansions in Loglinear Time. *arXiv preprint arXiv:1408.3060*, 2014.

[3] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight Uncertainty in Neural Network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR, 2015.

[4] Bernard J. Fino and V. Ralph Algazi. Unified Matrix Treatment of the Fast Walsh-Hadamard Transform. *IEEE Transactions on Computers*, 25(11):1142–1146, 1976.

[5] Diederik P. Kingma, Tim Salimans, and Max Welling. Variational Dropout and the Local Reparameterization Trick. *arXiv preprint arXiv:1506.02557*, 2015.

[6] Simone Rossi, Pietro Michiardi, and Maurizio Filippone. Good initializations of variational Bayes for deep models. In *International Conference on Machine Learning*, pages 5487–5497. PMLR, 2019.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[8] Dusan Bikov and Iliya Bouyukliev. Parallel fast Walsh transform algorithm and its implementation with CUDA on GPUs. *Cybernetics and Information Technologies*, 18(5):21–43, 2018.

[9] Jörg Arndt. *Matters Computational: ideas, algorithms, source code*. Springer Science & Business Media, 2010.