The Journal of Open Source Software

hessQuik: Fast Hessian computation of composite functions

Elizabeth Newman[*1] **and Lars Ruthotto**[†1]

1 Emory University, Department of Mathematics

DOI: 10.21105/joss.04171

Software

- Review
- Repository
- Archive

Editor: Mehmet Hakan Satman

Reviewers:

- @GregaVrbancic
- @yhtang

Submitted: 08 February 2022
Published: 16 February 2022

Summary

hessQuik is a lightweight repository for fast computation of second-order derivatives (Hessians) of composite functions (that is, functions formed via compositions) with respect to their inputs. The core of hessQuik is the efficient computation of analytical Hessians with GPU acceleration. hessQuik is a PyTorch (Paszke et al., 2019) package that is user-friendly and easily extendable. The repository includes a variety of popular functions or layers, including residual layers and input convex layers, from which users can build complex models via composition. hessQuik layers are designed for ease of composition - users need only select the layers and the package provides a convenient wrapper to compose the functions properly. Each layer provides two modes for derivative computation and the mode is automatically selected to maximize computational efficiency. hessQuik includes easy-access, illustrative tutorials on Google Colaboratory (Bisong, 2019), reproducible experiments, and unit tests to verify implementations. hessQuik enables users to obtain valuable second-order informantion for their models simply and efficiently.

Statement of need

Deep neural networks (DNNs) and other composition-based models have become a staple of data science, garnering state-of-the-art results in, e.g., image classification and speech recognition (Goodfellow et al., 2016), and gaining widespread use in the scientific community, particularly as surrogate models to replace expensive computations (Anirudh et al., 2020). The unrivaled universality and success of DNNs is due, in part, to the convenience of automatic differentiation (AD) which enables users to compute derivatives of complex functions without an explicit formula. Despite being a powerful tool to compute first-order derivatives (gradients), AD encounters computational obstacles when computing second-order derivatives.

Knowledge of second-order derivatives is paramount in many growing fields, such as physics-informed neural networks (PINNs) (Raissi et al., 2019), mean-field games (Ruthotto et al., 2020), generative modeling (Ruthotto & Haber, 2021), and adversarial learning (Papernot et al., 2016). In addition, second-order derivatives can provide insight into the optimization problem solved to build a good model (O'Leary-Roseberry & Ghattas, 2020). Hessians are notoriously challenging to compute efficiently with AD and cumbersome to derive and debug analytically. Hence, many algorithms approximate Hessian information, resulting in suboptimal performance. To address these challenges, hessQuik computes Hessians analytically and efficiently with an implementation that is accelerated on GPUs.

[*] co-first author
[†] co-first author

`hessQuik` Building Blocks

`hessQuik` builds complex functions constructed via composition of simpler functions, which we call *layers*. The package uses the chain rule to compute Hessians of composite functions, assuming the derivatives of the layers are implemented analytically. We describe the process mathematically.

Let $f : \mathbb{R}^{n_0} \to \mathbb{R}^{n_\ell}$ be a twice continuously-differentiable function defined as

$$f = g_\ell \circ g_{\ell-1} \circ \cdots \circ g_1, \quad \text{where} \quad g_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i} \quad \text{for } i = 1, \ldots, \ell. \tag{1}$$

Here, g_i represents the i-th layer and n_i is the number of hidden features on the i-th layer. We call n_0 the number of input features and n_ℓ the number of output features. We note that each layer can be parameterized by weights which we can tune by solving an optimization problem. Because `hessQuik` computes derivatives for the network inputs, we omit the weights from our notation.

Implemented `hessQuik` Layers

`hessQuik` includes a variety of popular layers and their derivatives. These layers can be composed to form complex models. Each layer incorporates a nonlinear activation function, $\sigma : \mathbb{R} \to \mathbb{R}$, that is applied entry-wise. The `hessQuik` package provides several activation functions, including sigmoid, hyperbolic tangent, and softplus. Currently supported layers include the following:

- `singleLayer`: This layer consists of an affine transformation followed by pointwise nonlinearity

  $$g_{\text{single}}(\mathbf{u}) = \sigma(\mathbf{K}\mathbf{u} + \mathbf{b}) \tag{2}$$

  where $\mathbf{K}$ and $\mathbf{b}$ are a weight matrix and bias vector, respectively, that can be tuned via optimization methods. Multilayer perceptron neural networks are built upon these layers.
- `residualLayer`: This layer differs from a single layer by including a skip connection

  $$g_{\text{residual}}(\mathbf{u}) = \mathbf{u} + h\sigma(\mathbf{K}\mathbf{u} + \mathbf{b}) \tag{3}$$

  where $h > 0$ is a step size. Residual layers are the building blocks of residual neural networks (ResNets) (He et al., 2016). ResNets can be interpreted as discretizations of differential equations or dynamical systems (E, 2017; Haber & Ruthotto, 2017).
- `ICNNLayer`: The input convex neural network layer preserves convexity of the composite function with respect to the input features. Our layer follows the construction of (Amos et al., 2017).
- `quadraticLayer`, `quadraticICNNLayer`: These are layers that output scalar values and are typically reserved for the final layer of a model.

The variety of implemented layers and activation functions makes designing a wide range of `hessQuik` models easy.

Computing Derivatives with `hessQuik`

In `hessQuik`, we offer two modes, forward and backward, to compute the gradient $\nabla_{\mathbf{u}_0} f$ and the Hessian $\nabla^2_{\mathbf{u}_0} f$ of the function with respect to the input features. The cost of computing derivatives in each mode differs and depends on the number of input and output features. `hessQuik` automatically selects the least costly method by which to compute derivatives. We briefly describe the derivative calculations using the two methods.

First, it is useful to express the evaluation of f as an iterative process. Let $\mathbf{u}_0 \in \mathbb{R}^{n_0}$ be a vector of input features. Then, the function evaluated at $\mathbf{u}_0$ is

$$\begin{aligned} \mathbf{u}_1 &= g_1(\mathbf{u}_0) && \in \mathbb{R}^{n_1} && (4)\\ \mathbf{u}_2 &= g_2(\mathbf{u}_1) && \in \mathbb{R}^{n_2} && (5)\\ &\vdots \\ f(\mathbf{u}_0) \equiv \mathbf{u}_\ell &= g_\ell(\mathbf{u}_{\ell-1}) && \in \mathbb{R}^{n_\ell} && (6) \end{aligned}$$

where $\mathbf{u}_i$ are the hidden features on layer i for $i = 1, \dots, \ell - 1$ and $\mathbf{u}_\ell$ are the output features.

Forward Mode

Computing derivatives in forward mode means building the gradient and Hessian *during forward propagation*; that is, when we form $\mathbf{u}_i$, we simultaneously form the corresponding gradient and Hessian information. We start by computing the gradient and Hessian of the first layer with respect to the inputs; that is,

$$\begin{aligned} \nabla_{\mathbf{u}_0}\mathbf{u}_1 &= \nabla_{\mathbf{u}_0} g_1(\mathbf{u}_0) && \in \mathbb{R}^{n_0 \times n_1} && (7)\\ \nabla^2_{\mathbf{u}_0}\mathbf{u}_1 &= \nabla^2_{\mathbf{u}_0} g_1(\mathbf{u}_0) && \in \mathbb{R}^{n_0 \times n_0 \times n_1} && (8) \end{aligned}$$

We compute the derivatives of subsequent layers using the following mappings for $i = 1, \dots, \ell - 1$

$$\begin{aligned} \nabla_{\mathbf{u}_0}\mathbf{u}_{i+1} &= \nabla_{\mathbf{u}_0}\mathbf{u}_i \nabla_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i) && \in \mathbb{R}^{n_0 \times n_{i+1}} && (9)\\ \nabla^2_{\mathbf{u}_0}\mathbf{u}_{i+1} &= \nabla^2_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i) \times_1 \nabla_{\mathbf{u}_0}\mathbf{u}_i \times_2 \nabla_{\mathbf{u}_0}\mathbf{u}_i^\top \\ &\quad + \nabla^2_{\mathbf{u}_0}\mathbf{u}_i \times_3 \nabla_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i) && \in \mathbb{R}^{n_0 \times n_0 \times n_{i+1}} && (10) \end{aligned}$$

where $\times_k$ is the mode-k product (Kolda & Bader, 2009) and $\nabla_{\mathbf{u}_0}\mathbf{u}_\ell \equiv \nabla_{\mathbf{u}_0} f(\mathbf{u}_0)$ is the Hessian we want to compute. The Hessian mapping in Equation 10 is illustrated in Figure 1. For efficiency, we store $\nabla_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i)$ when we compute the gradient and re-use this matrix to form the Hessian. Notice that the sizes of the derivatives always depend on the number of input features, n_0.

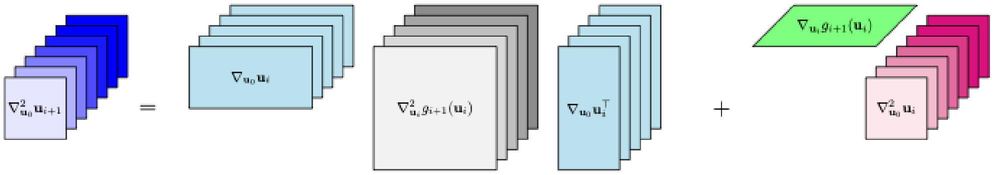
$\nabla^2_{\mathbf{u}_0}\mathbf{u}_{i+1}$
$=$
$\nabla_{\mathbf{u}_0}\mathbf{u}_i$
$\nabla^2_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i)$
$\nabla_{\mathbf{u}_0}\mathbf{u}_i^\top$
$+$
$\nabla_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i)$
$\nabla^2_{\mathbf{u}_0}\mathbf{u}_i$

Figure 1: Illustration of Hessian computation of $\nabla^2_{\mathbf{u}_0}\mathbf{u}_{i+1}$ in forward mode. Note that for the first term, the gray three-dimensional array $\nabla_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i)$ is treated as a stack of matrices. Then, the same Jacobian matrix $\nabla_{\mathbf{u}_0}\mathbf{u}_i$ is broadcast to each matrix in the stack, illustrated by the repeated cyan matrices. In the second term, the green matrix $\nabla_{\mathbf{u}_i} g_{i+1}(\mathbf{u}_i)$ is applied along the third dimension of the magenta three-dimensional array, $\nabla_{\mathbf{u}_0}\mathbf{u}_i$. Both of these operations can be parallelized and accelerated GPUs.

Backward Mode

Computing derivatives in backward mode is also known as *backward propagation* and is the method by which automatic differentiation computes derivatives. The process works as follows. We first forward propagate through the network *without computing gradients or Hessians*. After we forward propagate, we build the gradient and Hessian starting from the output layer and working backwards to the input layer. We start by computing derivatives of the final layer with respect to the previous features; that is,

$$\begin{aligned} \nabla_{\mathbf{u}_{\ell-1}} \mathbf{u}_\ell &= \nabla_{\mathbf{u}_{\ell-1}} g_\ell(\mathbf{u}_{\ell-1}) && \in \mathbb{R}^{n_{\ell-1}\times n_\ell} && (11)\\ \nabla^2_{\mathbf{u}_{\ell-1}} \mathbf{u}_\ell &= \nabla^2_{\mathbf{u}_{\ell-1}} g_\ell(\mathbf{u}_{\ell-1}) && \in \mathbb{R}^{n_{\ell-1}\times n_{\ell-1}\times n_\ell}. && (12) \end{aligned}$$

We compute derivatives of previous layers using the following mappings for $i = \ell-1, \ldots, 1$:

$$\begin{aligned} \nabla_{\mathbf{u}_{i-1}} \mathbf{u}_\ell &= \nabla_{\mathbf{u}_{i-1}} g_i(\mathbf{u}_{i-1}) \nabla_{\mathbf{u}_i} \mathbf{u}_\ell && \in \mathbb{R}^{n_{i-1}\times n_\ell} && (13)\\ \nabla^2_{\mathbf{u}_{i-1}} \mathbf{u}_\ell &= \nabla^2_{\mathbf{u}_i} \mathbf{u}_\ell \times_1 \nabla_{\mathbf{u}_{i-1}} g_i(\mathbf{u}_{i-1}) \times_2 \nabla_{\mathbf{u}_{i-1}} g_i(\mathbf{u}_{i-1})^\top && && \\ &\quad + \nabla^2_{\mathbf{u}_{i-1}} g_i(\mathbf{u}_{i-1}) \times_3 \nabla_{\mathbf{u}_i} \mathbf{u}_\ell && \in \mathbb{R}^{n_{i-1}\times n_{i-1}\times n_\ell}. && (14) \end{aligned}$$

For efficiency, we re-use $\nabla_{\mathbf{u}_{i-1}} g_i(\mathbf{u}_{i-1})$ from the gradient computation to compute the Hessian. Notice that the sizes of the derivatives always depend on the number of output features, n_ℓ.

Forward Mode vs. Backward Mode

The computational efficiency of computing derivatives is proportional to the number of input features n_0 and the number of output features n_ℓ. The heuristic we use is if $n_0 < n_\ell$, we compute derivatives in forward mode, otherwise we compute derivatives in backward mode. Our implementation automatically selects the mode of derivative computation based on this heuristic. Users have the option to select their preferred mode of derivative computation if desired.

Testing Derivative Implementations

The `hessQuik` package includes methods to test derivative implementations and corresponding unit tests. The main test employs Taylor approximations; for details, see (Haber, 2014).

Efficiency of `hessQuik`

We compare the time to compute the Hessian of a neural network with respect to the input features of three approaches: `hessQuik` (our AD-free method), `PytorchAD` which uses automatic differentiation following the implementation in (Huang et al., 2021), and `PytorchHessian` which uses the built-in Pytorch Hessian function.

We compare the time to compute the gradient and Hessian of a network with an input dimension $d = 2^k$ where $k = 0, 1, \ldots, 10$. We implement a residual neural network (He et al., 2016) with the width is $w = 16$, the depth is $d = 4$, and various numbers of output features, n_ℓ. For simplicity, the same network architecture is used for every timing test.

For reproducibility, we compare the time to compute the Hessian using Google Colaboratory (Colab) Pro and provide the notebook in the repository. For CPU runtimes, Colab Pro uses an Intel(R) Xeon(R) CPU with 2.20GHz processor base speed. For GPU runtimes, Colab Pro uses a Tesla P100 with 16 GB of memory.

In Figure 2 and Figure 3, we compare the performance of three approaches to compute Hessians of a neural network. In our experiments, we see faster Hessian computations using `hessQuik` and noticeable acceleration on the GPU, especially for networks with larger input and output dimensions. Specifically, Figure 2 shows that for a model with a scalar output, the timing using the `hessQuik` implementation scales better with the number of input features than either of the AD-based methods. Additionally, Figure 3 demonstrates that the `hessQuik` timings remain relatively constant as the number of output features changes whereas the `PytorchAD` timings significantly increase as the number of output features increases. Note that we only compare to `PytorchAD` for vector-valued outputs because `PytorchHessian` was noticeably slower for the scalar case.

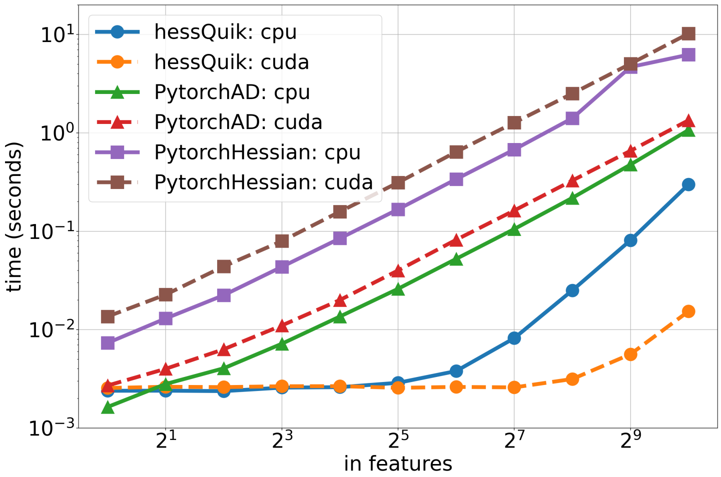

Figure 2: Average time over 10 trials to evaluate and compute the Hessian with respect to the input features for one output feature ($n_\ell = 1$). Solid lines represent timings on the CPU and dashed lines are timings on the GPU. The circle markers are the timings obtained using `hessQuik`.

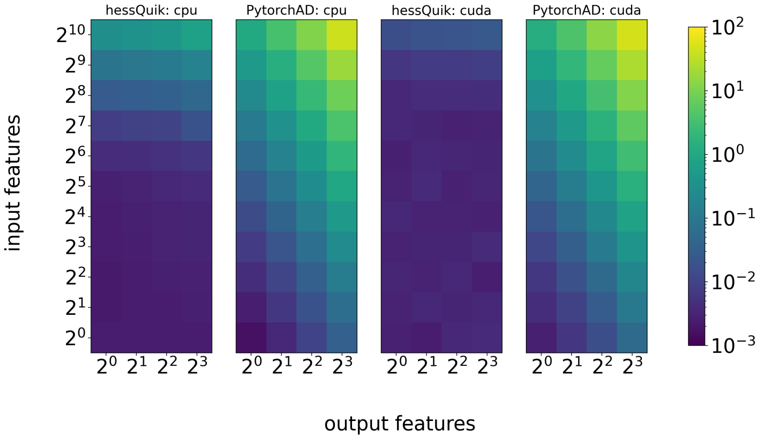

Figure 3: Average time over 10 trials to compute the Hessian with respect to the input features with variable number of input and output features. Each row corresponds to a number of input features, n_0, each column corresponds to a number of output features, n_ℓ, and color represents the amount of time to compute (in seconds).

Conclusions

hessQuik is a simple, user-friendly repository for computing second-order derivatives of composite functions with respect to their inputs. This PyTorch package includes many popular built-in layers, tutorial repositories, reproducibile experiments, and unit testing for ease of use. The implementation scales well in time with various input and output feature dimensions and performance is accelerated on GPUs, notably faster than automatic-differentiation-based second-order derivative computations. We hope the accessibility and efficiency of this package will encourage researchers to use and contribute to hessQuik in the future.

Acknowledgements

The development of hessQuik was supported in part by the US National Science Foundation under Grant Number 1751636, the Air Force Office of Scientific Research Award FA9550-20-1-0372, and the US DOE Office of Advanced Scientific Computing Research Field Work Proposal 20-023231. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

Amos, B., Xu, L., & Kolter, J. Z. (2017). *Input convex neural networks*. http://arxiv.org/abs/1609.07152

Anirudh, R., Thiagarajan, J. J., Bremer, P.-T., & Spears, B. K. (2020). Improved surrogates in inertial confinement fusion with manifold and cycle consistencies. *Proceedings of the National Academy of Sciences*, *117*(18), 9741–9746. https://doi.org/10.1073/pnas.1916634117

Bisong, E. (2019). Google colaboratory. In *Building machine learning and deep learning models on google cloud platform: A comprehensive guide for beginners* (pp. 59–64). Apress. https://doi.org/10.1007/978-1-4842-4470-8_7

E, W. (2017). A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, *5*(1), 1–11. https://doi.org/10.1007/s40304-017-0103-z

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Haber, E. (2014). *Computational methods in geophysical electromagnetics*. Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9781611973808

Haber, E., & Ruthotto, L. (2017). Stable architectures for deep neural networks. *Inverse Problems*, *34*(1), 014004. https://doi.org/10.1088/1361-6420/aa9a90

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. https://doi.org/10.1109/CVPR.2016.90

Huang, C.-W., Chen, R. T. Q., Tsirigotis, C., & Courville, A. (2021). Convex potential flows: Universal probability distributions with optimal transport and convex optimization. *International Conference on Learning Representations*. https://openreview.net/forum?id=te7PVH1sPxJ

Kolda, T. G., & Bader, B. W. (2009). Tensor decompositions and applications. *SIAM Review*, *51*(3), 455–500. https://doi.org/10.1137/07070111X

O'Leary-Roseberry, T., & Ghattas, O. (2020). *Ill-posedness and optimization geometry for nonlinear neural network training*. http://arxiv.org/abs/2002.02882

Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., & Swami, A. (2016). The limitations of deep learning in adversarial settings. *2016 IEEE European Symposium on Security and Privacy (EuroS p)*, 372–387. https://doi.org/10.1109/EuroSP.2016.36

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., … Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Raissi, M., Perdikaris, P., & Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics, Elsevier*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Ruthotto, L., & Haber, E. (2021). *An introduction to deep generative modeling*. https://doi.org/10.1002/gamm.202100008

Ruthotto, L., Osher, S. J., Li, W., Nurbekyan, L., & Fung, S. W. (2020). A machine learning framework for solving high-dimensional mean field game and mean field control problems. *Proceedings of the National Academy of Sciences*, *117*(17), 9183–9193. https://doi.org/10.1073/pnas.1922204117