HESSIAN BACKPROPAGATION FOR BATCHNORM

Paul Fischer
University of Tübingen
paul.fischer@uni-tuebingen.de

ABSTRACT

We compute the Hessian backpropagation (HBP) equations for the popular Batchnorm layer (Ioffe & Szegedy, 2015) in order to efficiently formulate the matrix-free multiplication with the Hessian. It is known that performing this operation in generic automatic differentiation frameworks can be quite slow. We propose that, by leveraging the knowledge about the Hessian's structure as outlined in Dangel & Hennig (2019), efficiency can be improved with this structural knowledge.

On this document: *This document summarizes the work of a research internship carried out in Philipp Hennig's Methods of Machine Learning group in the winter term 2019/2020.*

1 INTRODUCTION

First-order methods such as Stochastic Gradient Descent are currently the most popular methods when it comes to choosing the optimizer for training deep neural networks. Second-order methods have several advantages over first-order methods such as better scaling to large mini-batch sizes and they take fewer updates for convergence (Zhang et al., 2017). However, with this approach one has to compute the Hessian which involves high computational costs. Based on the work of Zhang et al. (2017) and Martens (2010), the approach of Hessian-free methods is used which involves not computing the curvature matrix - the Hessian - explicitly but only the product of the curvature matrix and a vector (Dangel & Hennig, 2019). Especially for Batch Normalization layers, computing the Hessian-vector product becomes extremely slow (Zhang et al., 2017).

The goal of this work is to manually compute the module Hessian of the BN layer in order to formulate an efficient Hessian-free multiplication. First, the forward pass of the BN layer is explained such that it becomes clear why the first and second-order derivatives are not straightforward to compute. Afterwards it is explained what is necessary to compute for using the Hessian-free approach and also explicitly computed.

2 FORWARD PASS OF THE BATCH NORMALIZATION LAYER

Often the exact notation in computations is unnecessarily complicated. For this work, the exact index notation is necessary in order to compute the exact first and second-order derivatives of the BN layer. The basic idea of Batch Normalization is to normalize the elements of the batch such that the empirical mean is 0 and the standard deviation is 1. The layer output is a linear transformation of the points. Let $x_i \in \mathbb{R}^D$, $i = 1, ..., N$ be a vector where N is the size of the batch. The layer output z_i depends on different characteristic values of the batch such as the mean μ and the variance σ^2. The forward pass of the layer is computed as follows:

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}, \quad j = 1, ..., D$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2, \quad j = 1, ..., D.$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}, \quad i = 1, ..., N, \quad j = 1, ..., D$$
$$z_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Here, $\gamma, \beta \in \mathbb{R}^D$ are learnable parameters where γ is called weight and β bias.

3 Computing the first and second order derivatives of the backward pass

Backpropagating the gradient and curvature information of the loss with respect to the input requires computing the partial first-order derivative of the layer output with respect to the input $\partial z_{i,j}/\partial x_{k,l}$ - the Jacobian - and the second-order derivative of the layer output with respect to the input $\partial^2 z_{i,j}/\partial x_{k,l}\partial x_{m,n}$ - the Hessian. The HBP equation can equation can be written as follows (Dangel & Hennig, 2019):

$$\frac{\partial^2 E(x)}{\partial x_{k,l} \partial x_{m,n}} = \sum_{i,j,p,q} \frac{\partial z_{i,j}}{\partial x_{k,l}} \frac{\partial^2 E(z)}{\partial z_{i,j} \partial z_{p,q}} \frac{\partial z_{p,q}}{\partial x_{m,n}} + \sum_{i,j} \frac{\partial^2 z_{i,j}}{\partial x_{k,l} \partial x_{m,n}} \delta z_{i,j}. \tag{1}$$

The first term of the equation propagates the curvature information which involves computing the Jacobian and the transposed Jacobian of the layer output with respect to the layer input and the second term, called Residual, introduces second-order effects of the module itself which we are looking for and involves computing the Hessian of the layer output with respect to the layer input. Since the Hessian-free method is used, we only need to compute the Hessian-vector product (HVP), that is

$$\begin{aligned}\sum_{m,n} \left(\frac{\partial^2 E(x)}{\partial x_{k,l} \partial x_{m,n}} \right) v_{m,n} = \sum_{m,n} \Bigg(&\sum_{i,j,p,q} \frac{\partial z_{i,j}}{\partial x_{k,l}} \frac{\partial^2 E(z)}{\partial z_{i,j} \partial z_{p,q}} \frac{\partial z_{p,q}}{\partial x_{m,n}} \\ &+ \sum_{i,j} \frac{\partial^2 z_{i,j}}{\partial x_{k,l} \partial x_{m,n}} \delta z_{i,j} \Bigg) v_{m,n}.\end{aligned} \tag{2}$$

As one can see here, the HVP requires computing the Jacobian-vector product (JVP) and the product with the Residual term, the Residual-vector product (RVP).

Computation of the first order derivative: the Jacobian

To compute the first-order derivative of the output $z_{i,j}$ with respect to the input $x_{k,l}$ SymPy was used (Meurer et al., 2017). The resulting Jacobian is a four-dimensional tensor. The $ijkl$-th entry of the Jacobian can be computed by using the chain rule:

$$\begin{aligned}\frac{\partial z_{i,j}}{\partial x_{k,l}} &= \sum_{m,n} \frac{\partial z_{i,j}}{\partial \hat{x}_{m,n}} \left(\sum_p \frac{\partial \hat{x}_{m,n}}{\partial \mu_p} \frac{\partial \mu_p}{\partial x_{k,l}} + \sum_q \frac{\partial \hat{x}_{m,n}}{\partial \sigma_q^2} \left(\sum_s \frac{\partial \sigma_q^2}{\partial \mu_s} \frac{\partial \mu_s}{\partial x_{k,l}} + \frac{\partial \sigma_q^2}{\partial x_{k,l}} \right) + \frac{\partial \hat{x}_{m,n}}{\partial x_{k,l}} \right) \\ &= \frac{\gamma_j}{N\sqrt{\sigma_j^2 + \varepsilon}} \delta_{j,l} \left(N\delta_{i,k} - 1 - \hat{x}_{i,j} \hat{x}_{k,j} \right)\end{aligned}$$

where $\delta_{j,l}$ is the Kronecker delta. Next we want to show that the Jacobian is symmetric. This a desired property since in equation 1 we only have to compute the Jacobian once and not the Jacobian and the transpose Jacobian and therefore in equation 2 the JVP and transpose JVP are the same. The Jacobian is symmetric if the equation $\partial z_{i,j}/\partial x_{k,l} \overset{!}{=} \partial z_{k,l}/\partial x_{i,j}$ is satisfied:

$$\begin{aligned}\frac{\partial z_{i,j}}{\partial x_{k,l}} &= \frac{\gamma_j}{N\sqrt{\sigma_j^2 + \varepsilon}} \delta_{jl} \left(N\delta_{ik} - 1 - \hat{x}_{i,j} \hat{x}_{k,j} \right) \\ &\overset{j=l}{=} \frac{\gamma_l}{N\sqrt{\sigma_l^2 + \varepsilon}} \delta_{lj} \left(N\delta_{ki} - 1 - \hat{x}_{k,l} \hat{x}_{i,l} \right) = \frac{\partial z_{k,l}}{\partial x_{i,j}}\end{aligned}$$

Therefore the Jacobian is symmetric. Eventually, we do not need to compute the Jacobian matrix explicitly but rather the JVP which is defined as $\sum_{k,l} \left(\partial z_{i,j}/\partial x_{k,l}\right) v_{k,l}$. Computing the JVP gives:

$$\begin{aligned}\sum_{k,l} \frac{\partial z_{i,j}}{\partial x_{k,l}} v_{k,l} &= \sum_{k,l} \frac{\gamma_j}{N\sqrt{\sigma_j^2+\varepsilon}} \delta_{jl} \left(N\delta_{ik} - 1 - \hat{x}_{i,j}\hat{x}_{k,j}\right) v_{k,l} \\ &= \frac{\gamma_j}{N\sqrt{\sigma_j^2+\varepsilon}} \left(N v_{i,j} - \sum_k v_{k,j} - \hat{x}_{i,j} \sum_k \hat{x}_{k,j} v_{k,j} \right)\end{aligned}$$

Computation of the second order derivative: the Hessian

The module Hessian of the BN layer $\partial^2 z_{i,j}/\partial x_{k,l}\partial x_{m,n}$ is again computed with SymPy (Meurer et al., 2017) which gives us the following expression for the *ijklmn*-th entry:

$$\begin{aligned}\frac{\partial^2 z_{i,j}}{\partial x_{k,l}\partial x_{m,n}} = \frac{\gamma_j \delta_{jl}\delta_{jn}}{N(\sigma_j^2+\varepsilon)} \Bigg(&- \left(\delta_{ik} - \frac{1}{N}\right)\hat{x}_{m,j} - \left(\delta_{im} - \frac{1}{N}\right)\hat{x}_{k,j} - \left(\delta_{km} - \frac{1}{N}\right)\hat{x}_{i,j} \\ &+ \frac{3}{N}\hat{x}_{i,j}\hat{x}_{k,j}\hat{x}_{m,j} \Bigg)\end{aligned}$$

Testing for positive definiteness, that is checking for $\sum_{k,l,m,n} v_{m,n} \partial^2 z_{i,j}/\partial x_{k,l}\partial x_{m,n} v_{k,l} \geq 0$, gives that the Hessian is indefinite. Since the module Hessian for BN is not diagonal, there is no easy way to make it positive semi-definite, which is desired for optimization. The Hessian can now be used to compute the residual and therefore the residual-vector product where the *kl*-th entry is defined as $\sum_{m,n} \left(\sum_{ij} \partial^2 z_{i,j}/\partial x_{k,l}\partial x_{m,n} \delta z_{ij} \right) v_{m,n}$ and computing this term gives us

$$\begin{aligned}\sum_{m,n} \left(\sum_{i,j} \frac{\partial^2 z_{i,j}}{\partial x_{k,l}\partial x_{m,n}} \delta z_{ij} \right) v_{m,n} = \frac{\gamma_l}{N(\sigma_l^2+\varepsilon)} \Bigg(&- \delta z_{k,l} \sum_m \hat{x}_{m,l} v_{m,l} \\ &+ \frac{1}{N} \sum_i \delta z_{i,l} \sum_m \hat{x}_{m,l} v_{m,l} \\ &- \hat{x}_{k,l} \sum_m \delta z_{m,l} v_{m,l} \\ &+ \frac{1}{N} \hat{x}_{k,l} \sum_i \delta z_{i,l} \sum_m v_{m,l} \\ &- v_{k,l} \sum_i \hat{x}_{i,l} \delta z_{i,l} \\ &+ \frac{1}{N} \sum_i \hat{x}_{i,l} \delta z_{i,l} \sum_m v_{m,l} \\ &+ \frac{3}{N} \hat{x}_{k,l} \sum_m \hat{x}_{m,l} v_{m,l} \sum_i \hat{x}_{i,l} \delta z_{i,l} \Bigg).\end{aligned} \tag{3}$$

4 Conclusion

The goal of this work was to compute the module Hessian and the Residual-vector product for the Batch Normalization layer in order to provide an efficient formulation of the HPB equation. By looking closer at equation 3, one can see that there are several values that occur more than once and therefore only have to be computed once. A benchmark comparison using a numpy implementation against automatic differentiation with the Autograd package by Maclaurin et al. (2015) showed that the computations for the Hessian of the BN layer are significantly faster than automatic differentiation which can be seen in Appendix F figure 1. This results can now be used to include them into BackPACK (Dangel et al., 2020) which provides Hessian-free second-order extensions to current deep-learning software.

REFERENCES

Felix Dangel and Philipp Hennig. A Modular Approach to Block-diagonal Hessian Approximations for Second-order Optimization Methods. *arXiv preprint arXiv:1902.01813*, 2019.

Felix Dangel, Frederik Kunstner, and Philipp Hennig. BackPACK: Packing more into backprop. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJlrF24twB`.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

James Martens. Deep learning via Hessian-free optimization. In *ICML*, volume 27, pp. 735–742, 2010.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3: e103, January 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL `https://doi.org/10.7717/peerj-cs.103`.

Huishuai Zhang, Caiming Xiong, James Bradbury, and Richard Socher. Block-diagonal hessian-free optimization for training neural networks. *arXiv preprint arXiv:1712.07296*, 2017.

APPENDIX A: DERIVING THE *ijkl*-TH ENTRY OF THE JACOBIAN

$$
\begin{aligned}
\frac{\partial z_{i,j}}{\partial x_{k,l}} &= \sum_{m,n} \frac{\partial z_{i,j}}{\partial \hat{x}_{m,n}} \frac{\partial \hat{x}_{m,n}}{\partial x_{k,l}} \\
&= \sum_{m,n} \frac{\partial z_{i,j}}{\partial \hat{x}_{m,n}} \left(\sum_p \frac{\partial \hat{x}_{m,n}}{\partial \mu_p} \frac{\partial \mu_p}{\partial x_{k,l}} + \sum_q \frac{\partial \hat{x}_{m,n}}{\partial \sigma_q^2} \frac{\partial \sigma_q^2}{\partial x_{k,l}} + \frac{\partial \hat{x}_{m,n}}{\partial x_{k,l}} \right) \\
&= \sum_{m,n} \frac{\partial z_{i,j}}{\partial \hat{x}_{m,n}} \left(\sum_p \frac{\partial \hat{x}_{m,n}}{\partial \mu_p} \frac{\partial \mu_p}{\partial x_{k,l}} + \sum_q \frac{\partial \hat{x}_{m,n}}{\partial \sigma_q^2} \left(\sum_s \frac{\partial \sigma_q^2}{\partial \mu_s} \frac{\partial \mu_s}{\partial x_{k,l}} + \frac{\partial \sigma_q^2}{\partial x_{k,l}} \right) + \frac{\partial \hat{x}_{m,n}}{\partial x_{k,l}} \right)
\end{aligned}
$$

APPENDIX B: PROPERTIES OF KRONECKER DELTAS

The Kronecker delta $\delta_{i,j}$ is defined as

$$
\delta_{i,j} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} .
$$

Note that $\delta_{i,j} = \delta_{j,i}$. To simplify the expression of the derivatives, we need some computational properties of the Kronecker delta:

$$
\begin{aligned}
\sum_j \delta_{i,j} a_j &= a_i \\
\sum_k \delta_{i,k} \delta_{k,j} &= \delta_{i,j}
\end{aligned}
$$

APPENDIX C: COMPUTING THE EXPLICIT *ijkl-th* ENTRY OF THE JACOBIAN

$$
\begin{aligned}
\frac{\partial z_{i,j}}{\partial x_{k,l}} &= \gamma_j \left(\frac{\delta_{i,k}\delta_{j,l} - \frac{\sum_i \delta_{i,k}\delta_{j,l}}{N}}{\sqrt{\sigma_j^2 + \varepsilon}} - \frac{(x_{i,j} - \mu_j)\sum_h \left(2\delta_{h,k}\delta_{j,l} - \frac{2\sum_i \delta_{i,k}\delta_{j,l}}{N}\right)(x_{h,j} - \mu_j)}{2N(\sigma_j^2 + \varepsilon)^{3/2}} \right) \\
&= \gamma_j \left(\frac{\delta_{i,k}\delta_{j,l} - \frac{\delta_{j,l}}{N}}{\sqrt{\sigma_j^2 + \varepsilon}} - \frac{(x_{i,j} - \mu_j)\delta_{j,l}(x_{h,j} - \mu_j)}{N(\sigma_j^2 + \varepsilon)^{3/2}} \right) \\
&= \frac{\gamma_j}{N\sqrt{\sigma_j^2 + \varepsilon}} \delta_{j,l} \left(N\delta_{i,k} - 1 - \hat{x}_{i,j}\hat{x}_{k,j} \right)
\end{aligned}
$$

APPENDIX D: CODE TO VERIFY THE EXACT COMPUTATION OF THE JACOBIAN

```
import autograd.numpy as np
from autograd import jacobian

# define our input for which we want to test

test_input = np.array([[0.00629718, 0.09731993],
                       [0.08979499, 0.07512464],
                       [0.06931812, 0.09813456]])

# define the forward pass

eps = 0.0

# mean
def mu(x):
    return np.mean(x, axis=0)

# variance
def sigma_sq(x):
    return np.var(x, axis=0)

# z-score
def x_hat(x):
    N, D = x.shape
    return (x-mu(x))/(np.sqrt(sigma_sq(x) + eps))

# linear transformation
def y(x):
    N,D = x.shape
    # fix gamma and beta just to test
    gamma, beta = np.ones(D), np.zeros(D)
    return gamma*x_hat(x) + beta

# autograd Jacobian
jac_y = jacobian(y)

# define kronecker delta for the derivative
def kron_delta(i,j):
    if(i==j):
        return 1
    else:
        return 0
```

```
def my_jacobian(x):
    N,D = x.shape
    # fix gamma as before
    gamma = np.ones(D)
    var = sigma_sq(x)
    x_hats = x_hat(x)
    # using for-loops just to check for correctness
    der = np.zeros((N,D,N,D))
    for i in range(N):
        for j in range(D):
            for k in range(N):
                for l in range(D):
                    factor = gamma[j]*kron_delta(j,l)/
                             (N * np.sqrt(var[j] + eps))
                    der[i,j,k,l] = factor * ( N * kron_delta(i,k)
                                       - 1
                                       - x_hats[i][j] * x_hats[k][j])
    return der
```

APPENDIX E: CODE TO VERIFY THE EXACT COMPUTATION OF THE HESSIAN

```
# compute the Hessian with autograd
hess = jacobian(jac_y)

# compute the explicit Hessian
def my_hessian(x):
    var = sigma_sq(x)
    x_hats = x_hat(x)
    N,D = x.shape
    gamma = np.ones(D)
    beta = np.zeros(D)
    result = np.zeros((N,D,N,D,N,D))
    # again for loops just for verification
    for i in range(N):
        for j in range(D):
            for k in range(N):
                for l in range(D):
                    for m in range(N):
                        for n in range(D):
                            factor = (gamma[j] * kron_delta(j,l)
                                      * kron_delta(j,n))/(N *
                                      (var[j] + eps))
                            sum_1 = (kron_delta(i,k) - 1/N)
                                    * x_hats[m][j]
                            sum_2 = (kron_delta(i,m) - 1/N)
                                    * x_hats[k][j]
                            sum_3 = (kron_delta(k,m) - 1/N)
                                    * x_hats[i][j]
                            sum_4 = (3/N) * x_hats[i][j] *
                                    x_hats[k][j] * x_hats[m][j]
                            result[i,j,k,l,m,n] = factor *
                                                  (- sum_1
                                                   - sum_2
                                                   - sum_3
                                                   + sum_4)
    return result
```

APPENDIX F: BENCHMARK AUTOGRAD HESSIAN VS. MY IMPLEMENTATION

```
import perfplot
perfplot.show(
    setup=lambda n: np.random.rand(2, n),
    kernels=[
        lambda a: hess(a),
        lambda a: my_hessian(a)
    ],
    labels=["autograd Hessian", "explicit Hessian"],
    n_range=[k for k in range(1, 11)],
    xlabel="dimension D in a (2xD) matrix",
    # More optional arguments with their default values:
    # title=None,
    # logx="auto",  # set to True or False to force scaling
    # logy="auto",
    # equality_check=numpy.allclose,  # set to None to disable
    # "correctness" assertion
    # automatic_order=True,
    # colors=None,
    # target_time_per_measurement=1.0,
    # time_unit="s",  # set to one of
    # ("auto", "s", "ms", "us", or "ns") to force plot units
    # relative_to=1,  # plot the timings relative to one
    # of the measurements
    # flops=lambda n: 3*n,  # FLOPS plots
)
```

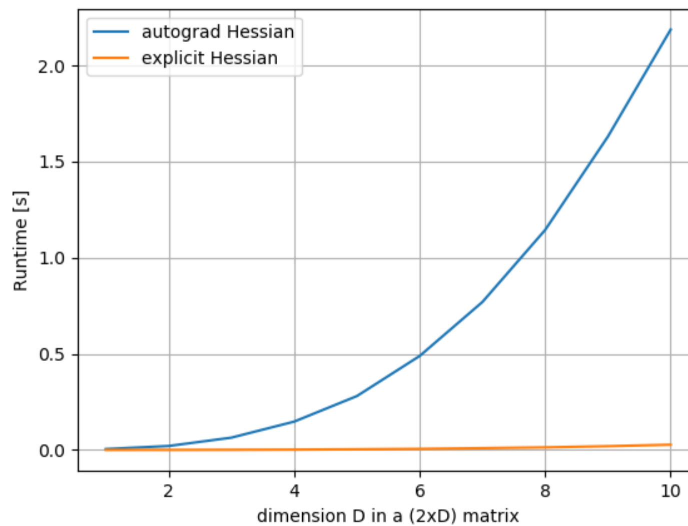

Figure 1: Benchmark autograd package vs. explicit implementation