O2Grad - A Pytorch Extension for 2nd Order Backpropagation

David S. Hippocampus*
Department of Computer Science
Cranberry-Lemon University
Pittsburgh, PA 15213
hippo@cs.cranberry-lemon.edu

May 23, 2022

Abstract

With the advent of deep learning, a substantial amount of work has gone into understanding what loss landscapes of neural networks look like and how their structure can be exploited, with the possible applications ranging from better generalization to sparsification, explainability and robustness to adversarial attacks. In this context, the Hessian of the network loss with respect to the parameters is paramount to the description of seconder-order information. However, since the Hessian is quadratic in the number of parameters of the network, computation and storage of the full Hessian is prohibitively expensive even for small datasets and toy models using current tools. In this paper, we present O2Grad, an extensible Python package on top of Pytorch which enables second order backpropagation for common building blocks of neural networks. Although impractical for large neural networks, we show that O2Grad can be used to considerably speed up the calculation of the full Hessian with respect to the current two-stage backpropagation method in Pytorch for small-sized networks. The theoretical considerations provide a starting point for building similar, more optimized tools, while the package itself can already be used out-of-the-box to speed up full Hessian calculation of toy models in research, paving the way for a more rigorous investigations of the Hessian in the future.

1 Introduction

2 Theoretical Preliminaries

2.1 Calculating Hessian Eigenpairs

Quickly calculating the eigenvalues and eigenvectors of neural network Hessians remains a tricky task, since the Hessian is a matrix of N^2 values if N is the number of network parameters, and modern neural networks are notorious for having a high number of parameters, OpenAI's GPT-3

*Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

being the most extreme example at the time of writing this thesis with 175 billion parameters [Brown et al., 2020]. This makes it practically unfeasible to calculate the full eigenvalue spectrum for larger models, at least not by virtue of first calculating the Hessian. A possible alternative is to use Power Iteration or improvements thereof such as Stochastic Power Iteration and Lanczos' algorithm to compute the eigenvalues and eigenvectors of the Hessian [Sa et al., 2017]. The underlying idea, in a nutshell, is to iteratively compute the dot product of the Hessian with a vector $\mathbf{v}$ until the vector converges against the eigenvector of the largest eigenvalue. Eigenvectors of successive eigenvalues can then be obtained by subtracting the previously obtained spectral components from the Hessian (making use of the spectral decomposition of the Hessian), which guarantees that the eigenvector obtained next will correspond to the next largest eigenvalue of the Hessian. Thankfully, calculating the dot product of the Hessian of an ANN with a vector $\mathbf{v}$ does not require explicitly computing the full Hessian of the network and can be done efficiently using B.A. Pearlmutter's $\mathcal{R}\{.\}$ technique without a quadratic bottleneck [Pearlmutter, 1994]. The modern Machine Learning framework *PyTorch* also provides a built-in function `hvp()` in the `autograd.functional` package that can in principle be used to calculate the Hessian of the network, which is exploited by the package *pytorch-hessian-eigenthings* [1] [Golmant et al., 2018]. However, while successive application of power iteration avoids the quadratic memory bottleneck of calculating the full Hessian, the eigenvectors and their eigenvalues must be calculated one after another in this approach, thus leading to a tradeoff in speed, since the method cannot be parallelized (or only to little degree). We believe that in terms of speed and for moderately sized neural networks, the approach of directly calculating the eigenvalues Hessian may be a better approach, even more so given the availability of fast SSDs that would also allow for the storage of larger Hessians. Furthermore, to reduce the memory and computational complexity, one might consider using **approximations of the Hessian** rather than the actual Hessian. A lightweight approximation of the Hessian can be attained by approximating the Hessian as a diagonal matrix and calculating only the diagonal entries, thus neglecting terms $\frac{\mathcal{L}}{\partial\theta_i\partial\theta_j}$ where parameters θ_i and θ_j are not the same. This approximation is used, for instance, in the *Optimal Brain Damage* algorithm [Cun et al., 1990] which is used for neural network parameter pruning. On the other hand, that particular algorithm is not concerned with the eigenvalues and eigenvectors of the Hessian, but rather with obtaining a computationally viable approximation of the Hessian vector product $\mathbf{H} \cdot v$. In general, the Hessian of a neural network cannot be expected to be diagonal, and thus the eigenvalues and eigenvectors of the Hessian should be expected to deviate considerably from the diagonal approximation (as our experiments in section **??** also show). However, it does have the advantage of having few non-zero values, allowing the Hessian to be stored with linear memory complexity $\mathcal{O}(N)$. Furthermore, the eigenvalues can also be computed in linear time, since they simply correspond to the values on the diagonal, while the eigenvectors are known in advance to be the canonical basis vectors of $\mathbb{R}^{\mathbb{N}}$.

A similar approach that uses more information on the curvature is to calculate a so-called *block diagonal approximation*: Assuming the neural network consists of layers $l^{(1)}, l^{(2)}, ..., l^{(T)}$, the elements of the Hessian will have the form

$$\frac{\partial\mathcal{L}}{\partial\theta_i^{(s)}\partial\theta_j^{(t)}}, \tag{1}$$

where $\theta_i^{(s)}$ and $\theta_j^{(t)}$ could be parameters from different layers $l^{(s)}$ and $l^{(t)}$. The idea of the block

[1] `https://github.com/noahgolmant/pytorch-hessian-eigenthings`

diagonal approximation, then, is to neglect cross-layer terms by setting the respective second-order derivatives to zero, giving rise to a block-matrix structure. This approximation contains more curvature information than just the diagonal of the Hessian and should therefore be better suited to compute eigenvalues and eigenvectors, although generally, the Hessian of neural networks can also not be expected to have a block-diagonal structure (see Appendix A.1). Assuming a network of T layers in which the largest layer has k parameters, the memory complexity of the block-diagonal approximation can be bounded by $\mathcal{O}(Tk^2)$, which is better than the lax upper bound of $\mathcal{O}(T^2k^2)$ for the actual Hessian. Furthermore, calculating the eigenvalues and eigenvectors of the block-diagonal approximation incurs a lower time complexity (and memory complexity, given a suitable representation) than for those of the actual Hessian. This stems from the fact that the 'blocks' in the block-diagonal matrix act on non-interacting subspaces of the latent space, allowing also for the eigenvalues and eigenvectors to be calculated on the respective subspaces. More formally:

Theorem 2.1. *Let $\boldsymbol{A} \in K^{N\times N}$ be the block-diagonal transformation matrix of a linear operator, i.e. the matrix can be written as direct sum $\boldsymbol{A} = \bigoplus_{i=1}^{n} \boldsymbol{A}_n$, where $\boldsymbol{A}_i \in K^{N_i}$ are the transformation matrices describing the linear operation on the respective subspaces and $\sum_{i=1}^{n} N_i = N$. Let us write $\boldsymbol{v} = (\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n) \in K^N$, with $\boldsymbol{v}_i \in K^{N_i}$ to denote a vector of the operator space. Then, the set of all eigenvectors of operator $\boldsymbol{A}$ are given by*

$$\begin{aligned} eigvec(\boldsymbol{A}) = \{(v_1, 0, ..., 0), v_1 \in eigvec(\boldsymbol{A}_1)\} \cup \{(0, v_2, ..., 0), v_2 \in eigvec(\boldsymbol{A}_2)\} \cup ... \\ \cup \{(0, 0, ..., \boldsymbol{v}_n), \boldsymbol{v}_n \in eigvec(\boldsymbol{A}_n)\} \end{aligned} \tag{2}$$

Proof: For the action of $\mathbf{A}$ on an arbitrary vector $\mathbf{v}$, we can write

$$\mathbf{A}\mathbf{u} = (\mathbf{A}_1 \oplus \mathbf{A}_2 \oplus ... \oplus \mathbf{A}_n)\mathbf{u} = (\mathbf{A}_1\mathbf{u}_1, \mathbf{A}_2\mathbf{u}_2, ..., \mathbf{A}_n\mathbf{u}_n) \tag{3}$$

Let $\mathbf{v}_i$ be an eigenvalue of $\mathbf{A}_i$ with eigenvalue λ_i, s.t. $\mathbf{A}_i\mathbf{v}_i = \lambda_i\mathbf{v}_i$. Then, the action of $\mathbf{A}$ on the vector $(\mathbf{v}_1, ..., \mathbf{v}_i, ..., \mathbf{v}_n)$ with $\mathbf{v}_j = \mathbf{0} \forall i \neq j$ is simply

$$\mathbf{A}(\mathbf{0}, ..., \mathbf{v}_i, ..., \mathbf{0}) = (\mathbf{A}_1\mathbf{0}, ..., \mathbf{A}_i\mathbf{v}_i, ..., \mathbf{A}_n\mathbf{0}) = (\mathbf{0}, ..., \lambda_i\mathbf{v}_i, ..., \mathbf{0}) \tag{4}$$

$$= \lambda_i(\mathbf{0}, ..., \mathbf{v}_i, ..., \mathbf{0}) \tag{5}$$

Thus, we know that for any subspace, the respective $\leq N_i$ eigenvectors of $\mathbf{A}_i$ can be extended to eigenvectors of $\mathbf{A}$, giving a total of $\leq \sum_{i=1}^{n} N_i = N$ eigenvectors constructed in this way. Furthermore, there can be no linearly independent eigenvectors other than these, for if that were the case, there would be an eigenvector $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ s.t.

$$(\mathbf{A}_1\mathbf{v}_1, ..., \mathbf{A}_n\mathbf{v}_n) = (\lambda\mathbf{v}_1, ..., \lambda\mathbf{v}_n), \tag{6}$$

implying $\mathbf{A}_i\mathbf{v}_i = \lambda\mathbf{v}_i$, thus λ being an eigenvalue on all subspaces where $\mathbf{v}_i$ is nonzero. But then $\lambda\mathbf{v}$ is a linear combination of those eigenvectors

$$\lambda\mathbf{v} = \lambda(\mathbf{v}_1, \mathbf{0}, ..., \mathbf{0}) + \lambda(\mathbf{0}, \mathbf{v}_2, ..., \mathbf{0}) + \lambda(\mathbf{0}, \mathbf{0}, ..., \mathbf{v}_n), \tag{7}$$

in contradiction to our assumption, and we are done.

Practically, this means that the time complexity for calculating all eigenvalues and eigenvectors of the block-diagonal approximation can be reduced to an upper bound $\mathcal{O}(Tk^3)$ rather than the lax upper bound of $\mathcal{O}(T^3k^3)$, which is a considerable improvement.

2.2 Second-Order Optimization

Due to the presence of the Hessian in the updates, our chaos-pruning optimization approach can be understood as a Second Order Optimization method. There have been numerous attempts to use second-order derivatives for neural network optimization in the past, with arguably most of them being rooted in Newton's method. In optimization, Newton's method on a variable $\mathbf{x}$ is given by the iteration formula:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \mathbf{H}(\mathbf{x}^{(t)})^{-1}\mathbf{J}(\mathbf{x}^{(t)}), \tag{8}$$

where $\mathbf{J}$ is the Jacobian of the loss w.r.t. the inputs, and $\mathbf{H}^{-1}$ is the inverse of the Hessian of the loss w.r.t. the inputs. For reasons already explained, calculating the actual Hessian, no less inverting it, is generally a practically unfeasible task, which is why Second Order Approximation Methods typically try introduce modifications to Newton's method or introduce some approximation of the Hessian that is easy to invert. Specifically, block-diagonal approximations of the Hessian have been used in deep learning methods recently, for instance in the *Practical Gauss-Newton Optimisation* approach [Botev et al., 2017] (A. Botev, H. Ritter, D. Barber) or in the *Block-diagonal Hessian-free Optimization* [Zhang et al., 2017] approach (H. Zhang et al). Both of the aforementioned approaches use the so-called generalized Gauss-Newton-Matrix instead, which has the useful property of being positive semidefinite and is defined as

$$\mathbf{G} := {\mathbf{J}_w^y}^T \mathbf{H}_y \mathbf{J}_w^y, \tag{9}$$

where $\mathbf{J}_w^y$ is the Jacobian of all intermediate outputs w.r.t. their respective layer parameters and $\mathbf{H}_y$ is the Hessian of the loss w.r.t model parameters. Another curvature matrix often used is the Fisher information matrix, which is equal to the negative expected Hessian of the log-likelihood

$$\mathbf{F} = -\mathbb{E}_{p(x|w)}\left[\mathbf{H}_{\log(p(x|w))}\right]. \tag{10}$$

Second order optimization approaches using the Fisher information matrix are studied under the label "natural gradient descent" [Zhang et al., 2017, p.5 l.39] [Amari, 1998] [Pascanu and Bengio, 2014] [Roux et al., 2007]. Notably, the more recent K-FAC method [Martens and Grosse, 2020] uses a sophisticated block-wise Kronecker-factored approach without the need of a block-diagonal matrix to compute an easily invertible approximation of the Fisher matrix.

3 Hessian Calculation

3.1 Two-stage Backpropagation

To our knowledge, the easiest out-of-the-box way to calculate the Hessians w.r.t. parameters in autodifferentiation packages such as PyTorch is *Two-Stage Backpropagation*. That is, let us say the autodifferentiation package provides a function `grad()`, which given a model f parametrized by $\mathbf{w}$, given loss function $\mathcal{L}$ and input $\mathbf{x}$ will calculate the gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$. Furthermore, let us assume that the function `grad()` is implemented using only operations supported for backpropagation, such that the calculation of the gradient gets recorded on a computational graph. The computational graph gives us a representation of the gradient function of the neural network for a specific input, and we can backpropagate through this secondary computation graph from each partial derivative of the gradient $\frac{\partial L}{\partial w_i}$ to obtain its respective column of the Hessian

$$\frac{\partial}{\partial w_j}\left(\frac{\partial L}{\partial w_i}\right) = \frac{\partial^2 L}{\partial w_j \partial w_i}. \tag{11}$$

Combining the gradients over all i yields the full Jacobian. Algorithm 1 provides a pythonic pseudocode implementation.

Algorithm 1 Pythonic pseudocode for the calculation of the Hessian from backpropagation in two stages (first from the loss, then from the gradients of the loss computed.)

```
def twostage_backpropagation(x, y, f, L):
   loss = L(f(x), y)
   dLdw = grad(loss)
   H = array(N, N)
   for i in range(N):
      H[:,i] = grad(dLdw[i])
   return H
```

Unfortunately, the complexity of this algorithm is rather difficult to estimate, since it depends on the exact implementation of the backpropagation algorithm. However, given a time cost of T for the regular backpropagation step (i.e. backpropagating from the loss L), and given that the computational graph for calculating the gradient $\frac{\partial L}{\partial \mathbf{w}}$ should contain the computation graph for calculating the loss L as a subgraph, we should expect the time complexity to be at least in the order of $NT + T = (N + 1)T$ if N is the number of parameters in the network.

3.2 Second Order Backpropagation

The method we implement for calculating the Hessian of the network using Second-Order Backpropagation is equivalent to the *Stagewise Second-Order Backpropagation* algorithm described by E. Mizutani and S.E. Dreyfus in [Mizutani et al., 2005]. However, we provide a more general formulation of the algorithm applicable not only to classical feed-forward ANNs such as the MLP (fully-connected layers and sigmoid nonlinear activations), but also to modern deep learning architectures and derive expressions for the local derivative tensors required to perform the calculation.

3.3 Diagonal Hessian Blocks

Consider a strictly sequential, feed-forward neural network with L layers, expressed mathematically as

$$\mathbf{y}^{(s+1)} = \mathbf{f}^{(s)}(\mathbf{y}^{(s)}), \quad \mathbf{f}^{(s)} : \mathbb{R}^{N_s} \to \mathbb{R}^{N_{s+1}} \quad \forall s \in \{0 \leq 1 \leq ... \leq L-1\}, \tag{12}$$

where $\mathbf{y}^{(0)}$ or layer 0 is the input to the network and $\mathbf{f}^{(s)}$ is a potentially parametrized transformation function (for example a linear transformation or a non-linear activation function) on which we impose the following conditions:

1. The Jacobian and Hessian

$$\mathbf{J}_x^{y,(s)}(\mathbf{x}) := \left\{ \frac{\partial f_i^{(s)}(\mathbf{x})}{\partial x_j} \right\}_{ij} \qquad \mathbf{H}_{xx}^{y,(n)}(\mathbf{x}) := \left\{ \frac{\partial^2 f_i^{(s)}(\mathbf{x})}{\partial x_j \partial x_k} \right\}_{ijk} \tag{13}$$

   with respect to the input should be defined for almost any $\mathbf{x}$. From now on, we refer to these as *Output-Input-Jacobian* (OIJ) and *Output-Input-Hessian* (OIH), respectively.

2. Additionally if $\mathbf{f}^{(s)}$ is a parametrized function $\mathbf{f}^{(s)}(\mathbf{x}) \equiv f(\mathbf{w}^{(s)}; \mathbf{x})$, the Jacobian and Hessian

$$\mathbf{J}_w^{y,(s)}(\mathbf{w}^{(s)}; \mathbf{x}) = \left\{ \frac{\partial f_i^{(s)}(\mathbf{x})}{\partial w_j^{(s)}} \right\}_{ij} \qquad \mathbf{H}_{ww}^{y,(s)}(\mathbf{w}^{(s)}; \mathbf{x}) = \left\{ \frac{\partial^2 f_i^{(s)}(\mathbf{x})}{\partial w_j^{(s)} w_k^{(s)}} \right\}_{ijk} \tag{14}$$

with respect to the function parameters should be defined for almost any $\mathbf{w}^{(s)}$. We shall refer to these as *Output-Parameter-Jacobian* (OPH) and *Output-Parameter-Hessian* (OPH), respectively.

3. Finally, if $\mathbf{f}^{(s)}$ is a parametrized function, we also demand that the mixed Hessian

$$\mathbf{H}_{xw}^{y,(s)}(\mathbf{w}^{(s)}; \mathbf{x}) = \left\{ \frac{\partial^2 f_i^{(s)}(\mathbf{x})}{\partial x_j \partial w_k^{(s)}} \right\}_{ijk} \tag{15}$$

is defined for almost any $\mathbf{x}$, $\mathbf{w}^{(s)}$. We shall call this tensor *mixed Output-Parameter-Hessian* (mOPH).

In practice, it has been observed that for typical neural network architectures, the network parameters are distributed in a ball around zero and therefore, it will suffice to demand that the parameter Hessians specified above are defined in a small subset of the entire parameter space. As a further remark, note that contrary to the Jacobian and Hessian of a scalar function (which are 1st order and 2nd order tensors, respectively), these are 2nd and 3rd order tensors.

Now, consider the Hessian of the network with respect to two parameters $w_i^{(s)}, w_j^{(t)}$. For now, let us set $s = t$, i.e. both are parameters of the same layer s. The Hessian can be rewritten as

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(s)}} = \frac{\partial}{\partial w_i^{(s)}} \left(\frac{\partial \mathcal{L}}{w_j^{(s)}} \right) \tag{16}$$

Using the chain rule of partial derivatives, we can write

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} w_j^{(s)}} = \frac{\partial}{\partial w_i^{(s)}} \left(\sum_k \frac{\partial y_k^{(s)}}{\partial w_j^{(s)}} \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} \right). \tag{17}$$

Using the sum rule of derivatives to apply the differential operator to each sum term, followed by the product rule, this can be further rewritten to

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} w_j^{(s)}} = \sum_k \frac{\partial}{\partial w_i^{(s)}} \left(\frac{\partial y_k^{(s)}}{\partial w_j^{(s)}} \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} \right) = \sum_k \frac{\partial^2 y_k^{(s)}}{\partial w_i^{(s)} \partial w_j^{(s)}} \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} + \frac{\partial y_k^{(s)}}{\partial w_j^{(s)}} \left(\frac{\partial}{\partial w_i^{(s)}} \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} \right) \tag{18}$$

Finally, we apply the chain rule of partial derivatives again and reorder the terms to obtain

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} w_j^{(s)}} = \sum_k \frac{\partial^2 y_k^{(s)}}{\partial w_i^{(s)} \partial w_j^{(s)}} \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} + \sum_k \frac{\partial y_k^{(s)}}{\partial w_j^{(s)}} \sum_l \frac{\partial y_l^{(s)}}{\partial w_i^{(s)}} \frac{\partial}{\partial y_l^{(s)}} \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} \tag{19}$$

$$= \sum_k \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} \frac{\partial^2 y_k^{(s)}}{\partial w_i^{(s)} \partial w_j^{(s)}} + \sum_{k,l} \frac{\partial y_l^{(s)}}{\partial w_i^{(s)}} \frac{\partial^2 \mathcal{L}}{\partial y_l^{(s)} \partial y_k^{(s)}} \frac{\partial y_k^{(s)}}{\partial w_j^{(s)}} \tag{20}$$

Comparing with equations (13) and (14), and substituting $\mathbf{x} \leftarrow \mathbf{y}^{(s)}$, observe that we have rewritten the Loss-Parameter-Hessian as a tensor dot product of (1) the Loss-Input-Gradient, (2) the Loss-Input-Hessian, (3) the OPH and (4) the OPJ:

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} w_j^{(s)}} = \left[\mathbf{g}^{(s)T} \cdot \mathbf{H}_{ww}^{y,(s)}(\mathbf{y}^{(s)}; \mathbf{w}^{(s)})\right]_{ij} + \left[\mathbf{J}_w^{y,(s)T}(\mathbf{y}^{(s)}; \mathbf{w}^{(s)}) \cdot \mathbf{H}^{(s)} \cdot \mathbf{J}_w^{y,(s)}(\mathbf{y}^{(s)}; \mathbf{w}^{(s)})\right]_{ij}. \quad (21)$$

Disregarding the functional dependencies and dropping the indices for more clarity:

$$\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}^{(s)} \mathbf{w}^{(s)}} = \mathbf{g}^{(s)T} \cdot \mathbf{H}_{ww}^{y,(s)} + \mathbf{J}_w^{y,(s)T} \cdot \mathbf{H}^{(s)} \cdot \mathbf{J}_w^{y,(s)}. \quad (22)$$

The Loss-Output-Hessian can be obtained in an analogous fashion as

$$\frac{\partial^2 \mathcal{L}}{\partial y_i^{(s-1)} \partial y_j^{(s-1)}} = \sum_k \frac{\partial \mathcal{L}}{\partial y_k^{(s)}} \frac{\partial^2 y_k^{(s)}}{\partial y_i^{(s-1)} \partial y_j^{(s-1)}} + \sum_{k,l} \frac{\partial y_l^{(s)}}{\partial y_i^{(s-1)}} \frac{\partial \mathcal{L}}{\partial y_l^{(s)} \partial y_k^{(s)}} \frac{\partial y_k^{(s)}}{\partial y_j^{(s-1)}}. \quad (23)$$

The derivation steps are completely analogous to those for the Loss-Parameter-Hessian, but replacing parameter $\mathbf{w}^{(s)}$ with $\mathbf{y}^{(s-1)}$, i.e. the input to layer s. Comparing the above equation with (13) and (14) and substituting $\mathbf{x} \leftarrow \mathbf{y}^{(s)}$, this can be written more succinctly as an expression depending on four distinct tensors, namely (1) gradient and (2) Hessian w.r.t. the output of layer s, as well as (3) OIJ and (4) OIH:

$$\mathbf{H}^{(s-1)} := \mathbf{g}^{(s)T} \cdot \mathbf{H}_{xx}^{y,(s)} + \mathbf{J}_x^{y,(s)T} \cdot \mathbf{H}^{(s)} \cdot \mathbf{J}_x^{y,(s)}. \quad (24)$$

Putting together equations (22) and (24), it becomes apparent that the respective matrices $\mathbf{H}^{(s-1)}$ and $\mathbf{H}_w^{(s-1)}$ for any layer s can be computed using an extension of the Vanilla feedforward and backpropagation step in regular neural networks (see algorithm 2), which involves calculating OIJ, OIH, OPJ and OPH in the feedforward step.

3.4 Off-Diagonal Hessian Blocks

The off-diagonal Hessian blocks, i.e. those blocks where $s \neq t$, are more difficult to compute. However, observe that due to the second derivative being symmetric,

$$\mathbf{H}^{(s,t)} = \left\{\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(t)}}\right\}_{ij} = \left\{\frac{\partial^2 \mathcal{L}}{\partial w_j^{(t)} \partial w_i^{(s)}}\right\}_{ij} = \mathbf{H}^{(t,s)}. \quad (25)$$

Thus only $\frac{T^2}{2} - T$ off-diagonal blocks need to be calculated and w.r.o.g. it suffices to consider the case $s < t$. We write

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(t)}} = \frac{\partial}{\partial w_i^{(s)}} \left(\frac{\partial^2 \mathcal{L}}{\partial w_j^{(t)}}\right) \quad (26)$$

As for the diagonal Hessian blocks, we start by using the chain rule to rewrite the partial derivatives w.r.t. the layer s parameter as

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(t)}} = \frac{\partial y_k^{(s+1)}}{\partial w_i^{(s)}} \frac{\partial}{\partial y_k^{(s+1)}} \left(\frac{\partial \mathcal{L}}{\partial w_j^{(t)}}\right) = \frac{\partial y_k^{(s+1)}}{\partial w_i^{(s)}} \frac{\partial}{\partial y_k^{(s+1)}} \left(\frac{\partial \mathcal{L}}{\partial y_l^{(t)}} \frac{\partial y_l^{(t)}}{\partial w_j^{(t)}}\right). \quad (27)$$

Algorithm 2 Pythonic pseudocode for the modified feedforward and backpropagation step required to compute the Loss-Parameter-Hessians for every diagonal block (here denoted `dL2dw2`).

```
def feedforward(x, t, layers, criterion):
   y, dydx, dy2dx2, dydw, dy2dw2 = {}
   y[0] = x
   for s in [1, 2, ..., T]:
      y[s] = layers[s](y[s-1])
      dydx[s] = layers[s].get_output_input_jacobian(y[s-1], y[s])
      dy2dx2[s] = layers[s].get_output_input_hessian(y[s-1], y[s])
      if layer[s].is_parametric():
         dydw[s] = layers[s].get_output_param_jacobian(y[s-1], y[s])
         dy2dw2[s] = layers[s].get_output_param_hessian(y[s-1], y[s])
   loss = criterion(y[T], t)

def backpropagation(loss, layers, criterion):
   dLdy, dL2dy2 = {}, {}
   dLdx[T+1] = criterion.get_output_input_jacobian(loss)
   dL2dx2[T+1] = criterion.get_output_input_hessian(loss)
   for s in [T, T-1, ..., 1]:
      dLdw = matmul(dLdy, dydw)
      dL2dy2 = dL2dx2[s+1]
      dLdy = dLdx[s+1]
      dL2dx2[s] = matmul(dLdy, dy2dx2[s])
      dL2dx2[s] += matmul(transpose(dydx[s]), dL2dy2, dydx[s])
      if layer[s].is_parametric():
         dL2dw2[s] = matmul(dLdy, dy2dw2[s])
         dL2dw2[s] += matmul(transpose(dydw[s]), dL2dy2, dydw[s])
```

Note that we have omitted the sum signs in the above equation following Einstein's Sum Notation to save space. Now, rather than directly using the product rule, which would give us derivatives for mixed layers s and t, we first apply the chain rule again to obtain

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(t)}} = \frac{\partial y_k^{(s+1)}}{\partial w_i^{(s)}} \frac{\partial y_m^{(t-1)}}{\partial y_k^{(s+1)}} \frac{\partial}{\partial y_m^{(t-1)}} \left(\frac{\partial \mathcal{L}}{\partial y_l^{(t)}} \frac{y_l^{(t)}}{\partial w_j^{(t)}} \right), \tag{28}$$

after which we apply the product rule, yielding

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(t)}} = \frac{\partial y_k^{(s+1)}}{\partial w_i^{(s)}} \frac{\partial y_m^{(t-1)}}{\partial y_k^{(s+1)}} \left(\left(\frac{\partial}{\partial y_m^{(t-1)}} \frac{\partial \mathcal{L}}{\partial y_l^{(t)}} \right) \frac{\partial y_l^{(t)}}{\partial w_i^{(t)}} + \frac{\partial \mathcal{L}}{\partial y_l^{(t)}} \frac{\partial^2 y_l^{(t)}}{\partial y_m^{(t-1)} \partial w_j^{(t)}} \right). \tag{29}$$

And using the chain rule one more time we obtain

$$\frac{\partial^2 \mathcal{L}}{\partial w_i^{(s)} \partial w_j^{(t)}} = \frac{\partial y_k^{(s+1)}}{\partial w_j^{(s)}} \underbrace{\frac{\partial y_m^{(t-1)}}{\partial y_k^{(s+1)}}}_{=: \mathbf{U}^{(s+1,t-1)}} \left(\frac{\partial y_o^{(t)}}{\partial y_m^{(t-1)}} \frac{\partial^2 \mathcal{L}}{\partial y_o^{(t)} \partial y_l^{(t)}} \frac{\partial y_l^{(t)}}{\partial w_i^{(t)}} + \frac{\partial \mathcal{L}}{\partial y_l^{(t)}} \frac{\partial^2 y_l^{(t)}}{\partial y_m^{(t-1)} \partial w_j^{(t)}} \right). \tag{30}$$

With the exception of the Output-Input-Jacobian defined as $\mathbf{U}^{(s+1,t-1)}$, this expression depends exclusively on local derivatives, i.e. such ones that can be calculated at the respective layers s or t given their inputs, outputs and parameters. In terms of equations (13) and (14), we can rewrite this expression more succinctly as

$$\mathbf{H}^{(s,t)} = \mathbf{J}_w^{y,(s+1)T} \cdot \mathbf{U}^{(s+1,t-1)} \cdot \underbrace{\left(\mathbf{g}^{(t)T} \cdot \mathbf{H}_{xw}^{(t)} + \mathbf{J}_x^{y,(t)T} \cdot \mathbf{H}^{(t)} \cdot \mathbf{J}_w^{y,(t)}\right)}_{=:\mathbf{V}^{(t)}}. \tag{31}$$

Now we still require a way of calculating what we call the *chained Output-Input-Jacobian* (cOIJ) $\mathbf{U}^{(s+1,t-1)}$, which can be obtained using the OIJs of the intermediate layers through iterated application of the chain rule:

$$\frac{\partial y_m^{(t-1)}}{\partial y_k^{(s+1)}} = \sum_{m_1} \sum_{m_2} ... \sum_{m_{t-2}} \frac{\partial y_{m_1}^{(s+2)}}{y_k^{(s+1)}} \frac{\partial y_{m_2}^{(s+3)}}{\partial y_{m_1}^{(s+2)}} ... \frac{\partial y_m^{(t-1)}}{\partial y_{m_{t-2}}^{(t-2)}} \tag{32}$$

$$\Rightarrow \mathbf{U}^{(s+1,t-1)} = \mathbf{J}_x^{y,(s+1)} \cdot \mathbf{J}_x^{y,(s+2)} \cdot ... \mathbf{J}_x^{y,(t-1)} \tag{33}$$

To avoid having to perform the full matrix chain multiplication for each $\mathbf{U}^{(s+1,t)}$, we can use a dynamic programming scheme:

$$\mathbf{V}^{(t,t)} := \mathbf{V}^{(t)}, \qquad \mathbf{V}^{(s,t)} = \mathbf{J}_x^{y,(s)} \cdot \mathbf{V}^{(s+1,t)} \tag{34}$$

Building upon the backpropagation step of Algorithm 2, this leads us to an updated backpropagation step as shown in Algorithm 3. Figure 1 illustrates the feedforward and backpropagation step of the algorithm.

3.5 Local Derivatives

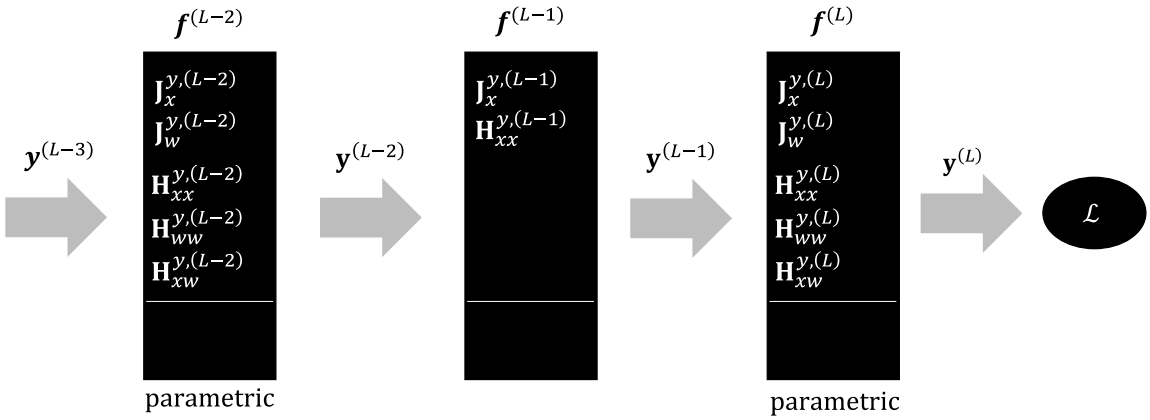
Next, we provide explicit expressions for the derivative tensors from equations (13), (14) and (15) - OIJ, OIH, OPJ, OPH, mOPH - for commonly used layers in modern deep learning architectures. As we will show in the implementation section, although the Jacobians and Hessians of a layer can be calculated implicitly in *PyTorch* using the autograd functions `jacobian()` and `hessian()` when applying some tricks, the respective function evaluations are far too slow (especially when feeding the network high-dimensional inputs) for running a performing Second-Order Backpropagation. Thus, it is more efficient to compute explicit expressions for the respective tensors and generate them on demand.

3.6 Parametric Layers

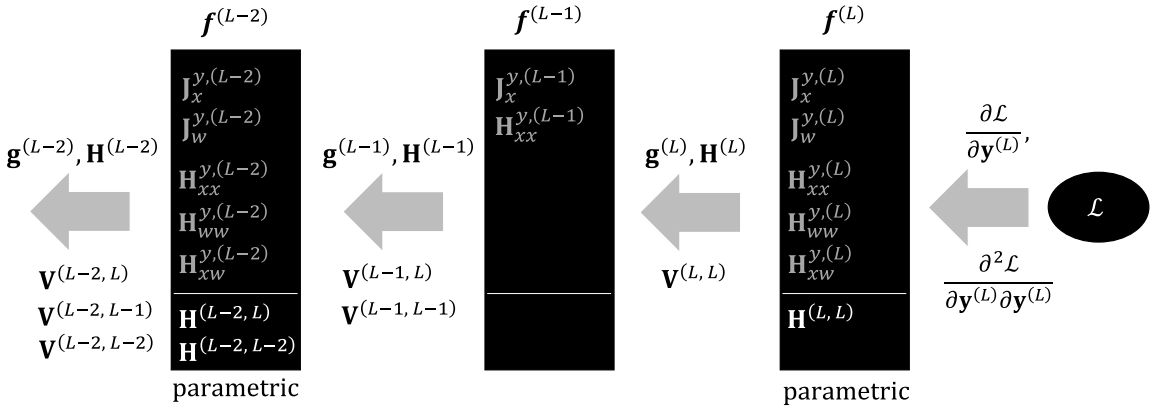
3.6.1 Fully Connected Layers

The earliest specified and most common type of parametric layers are linear, a.k.a. fully-connected layers, which for an input $\mathbf{x} \in \mathbb{R}^D$, weights $\mathbf{w} \in \mathbb{R}^{D' \times D}$ and bias $\mathbf{b} \in \mathbb{R}^{D'}$ operate as:

$$y_i = f_i(\mathbf{x}) = \sum_{j=0}^{D-1} w_{ij} x_j, + b_i. \tag{35}$$



(a) Feedforward



(b) Backpropagation

Figure 1: Illustration of the Stagewise Second-Order Backpropagation algorithm and the local derivatives computed at every layer, for a network of L layers. Here, layers $L-2$ and L are parametric layers, while $L-1$ is a non-parametric layer.

We arrive at the following expressions for the Jacobians and Hessians:

$$\text{OIJ}: \quad \frac{\partial y_i}{\partial x_k} = \sum_j w_{ij}\delta_{jk} = w_{ik}, \quad \text{OIH}: \quad \frac{\partial^2 y_i}{\partial x_k \partial x_l} = 0 \tag{36}$$

$$\text{OPJ}: \quad \frac{\partial y_i}{\partial w_{kl}} = \sum_j \delta_{(ij),(kl)} x_j = \sum_j \delta_{ik}\delta_{jl}x_j = \delta_{ik}x_l, \quad \frac{\partial y_i}{\partial b_k} = \delta_{ik} \tag{37}$$

$$\text{OPH}: \quad \frac{\partial^2 y_i}{\partial w_{mo}\partial w_{kl}} = 0, \quad \frac{\partial^2 y_i}{\partial b_l \partial b_k} = 0, \quad \frac{\partial^2 y_i}{\partial w_{lm}\partial b_k} = 0 \tag{38}$$

$$\text{mOPH}: \quad \frac{\partial^2 y_i}{\partial w_{lm}\partial x_k} = \delta_{(ik),(lm)} = \delta_{il}\delta_{km}, \quad \frac{\partial^2 y_i}{\partial x_k \partial b_l} = 0 \tag{39}$$

Algorithm 3 Pythonic pseudocode for the modified backpropagation step required to compute the full Loss-Parameter-Hessian (here denoted `dL2dw2`) including off-diagonal blocks.

```
def backpropagation(loss, layers):
   dLdy, dL2dy2 = {}, {}
   dLdx[T+1] = criterion.get_output_input_jacobian(loss)
   dL2dx2[T+1] = criterion.get_output_input_hessian(loss)
   for s in [T, T-1, ..., 1]:
      dLdw = matmul(dLdy, dydw)
      dL2dy2 = dL2dx2[s+1]
      dLdy = dLdx[s+1]
      dL2dx2[s] = matmul(dLdy, dy2dx2[s])
      dL2dx2[s] += matmul(transpose(dydx[s]), dL2dy2, dydx[s])
      for t in V[s+1].keys():
         V[s][t] = matmul(transpose(dydx[s]), V[s+1][t])
      if layer[s].is_parametric():
         V[s][s] = matmul(dLdy, dy2dxdw[s])
         V[s][s] += matmul(transpose(dydx), dL2dy2, dydw)
         for t in V[s+1].keys():
            dL2dw2[s][t] = matmul(transpose(dydw), V[s+1][t])
         dL2dw2[s][s] = matmul(dLdy, dy2dw2[s])
         dL2dw2[s][s] += matmul(transpose(dydw[s]), dL2dy2, dydw[s])
```

The above expressions are valid assuming single-sample inputs. For a batch of inputs, the network function is applied to every sample of the batch individually and there are no interactions between different batch samples, so for $\mathbf{x} \in \mathbb{R}^{B \times D}$ if B is the batch and D is the number of input dimensions:

$$\frac{\partial y_{b'i}}{\partial x_{bk}} = \sum_j w_{ij} \frac{\partial x_{b'j}}{\partial x_{bk}} = \sum_j w_{ij} \delta_{(b'j),(bk)} = \sum_j w_{ij} \delta_{b'b} \delta_{jk} = w_{ik} \delta_{b'b} \tag{40}$$

So for all local derivative tensors that depend on the input, the expressions for batched inputs can simply be obtained by computing the tensor product of the single-sample derivative tensors with a Kronecker delta over the two batch dimensions, e.g. $\tilde{\mathbf{J}}_x^y = \mathbf{J}_x^y \otimes \mathbf{I}_B$. This applies to all layers discussed in this section with the only exception of Batch Normalization layers.

3.6.2 1D Convolutions

Convolution Layers [LeCun et al., 1998] are a sparse type of Linear layers that have become popular for ML tasks on input data with highly localized information, such as audio tracks or images. When the index counts from zero, the Vanilla convolution in 1D without bias can be written as:

$$y_i = f_i(\mathbf{x}) = \mathbf{K} \star \mathbf{x} = \sum_{k=0}^{M-1} K_k x_{i+k} \tag{41}$$

The sum formula is to be interpreted as follows: Given a position i of the output vector, start at the same position i in the input vector and compute the dot product of the next m fields (size of the

convolution kernel) of $\mathbf{x}$ with kernel $\mathbf{K}$. If the convolution is *strided* with a stride s, the convolution kernel will be shifted by s over the input at every step, so for output y_i, the first field of the input to be scalar multiplied with the kernel will be at $x_{s\cdot i}$. If the kernel is *dilated* by a factor d, any two successive kernel fields are $d-1$ empty fields apart, and thus the k-th field of $\mathbf{x}$ to be multiplied with the k-th field of the kernel is $x_{d\cdot k}$. Taking these two together, the adapted convolution formula becomes:

$$y_i = f_i(\mathbf{x}) = \mathbf{K} \star \mathbf{x} = \sum_{k=0}^{M-1} K_k x_{s\cdot i + d\cdot k} \tag{42}$$

Furthermore, we need to account for *padding*. If a convolution is applied with a symmetric padding of size P to the input, the convolution is applied to a padded input vector $\tilde{\mathbf{x}}$ defined as follows:

$$\tilde{x}_i = \begin{cases} p^l_i, & i < P \\ x_{i-P}, & P \leq i < n+P \\ p^r_{i-n-P}, & n+P \leq i < n+2P \end{cases}, \tag{43}$$

where $\mathbf{p}^l, \mathbf{p}^r \in \mathbb{R}^P$. A common choice is to use a *random initialization* or *zero-padding*, i.e. setting $\mathbf{p}^l = \mathbf{p}^r = \mathbf{0}$. As a final complication, convolution layer in deep learning are usually implemented with an additional *channel* dimension and a bias. Therefore, the single-sample inputs become 2D tensors $\mathbf{x} \in \mathbb{R}^{C\times D}$, the kernels become 3D tensors $\boldsymbol{K} \in \mathbb{R}^{C'\times C\times M}$, and we introduce a bias vector $\mathbf{b} \in \mathbb{R}^{C'}$, allowing for the action of the convolution layer to be written now as

$$y^\alpha_i = f^\alpha_i(\mathbf{x}) = \sum_{k=0}^{M-1}\sum_{\beta=0}^{C-1} K^{\alpha\beta}_k \tilde{x}^\beta_{s\cdot i+d\cdot k} + b^\alpha. \tag{44}$$

Using this, we can now proceed to calculate the derivatives. We start with the OIJ:

$$\frac{\partial y^\alpha_i}{\partial x^\gamma_j} = \sum_{k=0}^{M-1}\sum_{\beta=0}^{C-1} K^{\alpha\beta}_k \frac{\partial \tilde{x}^\beta_{s\cdot i+d\cdot k}}{\partial x^\gamma_j}, \tag{45}$$

where the partial derivative of $\tilde{\boldsymbol{x}}$ can be expanded with equation (43) as

$$\frac{\partial \tilde{x}^\beta_{s\cdot i+d\cdot k}}{\partial x^\gamma_j} = \begin{cases} \frac{\partial p^{l,\beta}_{s\cdot i+d\cdot k}}{\partial x^\gamma_j}, & i < P \\ \frac{\partial \tilde{x}^\beta_{s\cdot i+d\cdot k-P}}{\partial x^\gamma_j}, & P \leq i < n+P \\ \frac{\partial p^{r,\beta}_{s\cdot i+d\cdot k-n-P}}{\partial x^\gamma_j}, & n+P \leq i < n+2P \end{cases}. \tag{46}$$

Since the padding values are independent from the inputs x_j, the respective derivatives would become zero and we are thus left with

$$\begin{aligned} \frac{\partial \tilde{x}^\beta_{s\cdot i+d\cdot k}}{\partial x^\gamma_j} &= I\left((s\cdot i + d\cdot k - P = j) \wedge (\beta = \gamma) \wedge (P \leq i < n+P)\right) \\ &= I(s\cdot i + d\cdot k - P = j)\cdot \delta^{\beta\gamma} \cdot I(P \leq i < n+P), \end{aligned} \tag{47}$$

where I is the indicator function. Inserting this back into equation (42) and omitting the last tensor, we obtain

$$\begin{aligned}\frac{\partial y_i^\alpha}{\partial x_j^\gamma} &= \sum_{k=0}^{M-1}\sum_{\beta=0}^{C-1} K_k^{\alpha\beta} I(s\cdot i + d\cdot k - P = j)\delta^{\beta\gamma} \\ &= K_k^{\alpha\gamma} I((s\cdot i + d\cdot k - P = j) \wedge (0 \leq k < M)) \\ &= K_k^{\alpha\gamma} I\left(k = \frac{j - s\cdot i + P}{d}\right) I(0 \leq k < M),\end{aligned} \tag{48}$$

and it is clear that applying the derivative w.r.t. the input one more time yields 0 since the kernel does not depend explicitly on the input.
The OPJ with respect to kernel parameters can be derived similarly:

$$\frac{\partial y_i^\alpha}{\partial K_j^{\gamma\delta}} = \sum_{k=0}^{M-1}\sum_{\beta=0}^{C-1} \frac{\partial K_k^{\alpha\beta}}{\partial K_j^{\gamma\delta}} \tilde{x}_{s\cdot i+d\cdot k}^\beta = \sum_{k=0}^{M-1}\sum_{\beta=0}^{C-1} \delta^{\alpha\gamma}\delta^{\beta\delta}\delta_{jk}\tilde{x}_{s\cdot i+d\cdot k}^\beta = \delta^{\alpha\gamma}\tilde{x}_{s\cdot i+d\cdot j}^\delta \tag{49}$$

From equation (48), applying the derivative w.r.t. the kernel parameters to obtain the mOPH yields

$$\begin{aligned}\frac{\partial^2 y_i^\alpha}{\partial K_l^{\delta\eta}\partial x_j^\gamma} &= \delta^{\alpha\delta}\delta^{\gamma\eta}\delta_{kl} I\left(k = \frac{j - s\cdot i + P}{d}\right) I(0 \leq k < M) \\ &= \delta^{\alpha\beta}\delta^{\gamma\eta} I\left(l = \frac{j - s\cdot i + P}{d}\right).\end{aligned} \tag{50}$$

The second indicator function $I(0 \leq l < M)$ resulting from application of the Kronecker delta δ_{kl} drops out, since l is an index of the kernel on the left side, and thus the indicator function's predicate is guaranteed to always be true. From equation (49) for the OPJ, it is clear that the OPH must be 0, since the padded input does not depend on the kernel. As for the derivatives w.r.t. the bias, these are equivalent to the ones for the linear layer. Putting everything together, we can summarize the local derivatives of the convolution layer as follows:

$$\text{OIJ}: \frac{\partial y_i^\alpha}{\partial x_j^\gamma} = K_k^{\alpha\gamma} I\left(k = \frac{j - s\cdot i + P}{d}\right) I(0 \leq k < M), \quad \text{OIH}: \frac{\partial^2 y_i^\alpha}{\partial x_l^\delta \partial x_k^\gamma} = 0 \tag{51}$$

$$\text{OPJ}: \frac{\partial y_i^\alpha}{\partial K_j^{\gamma\delta}} = \delta^{\alpha\gamma}\tilde{x}_{s\cdot i+d\cdot j}^\delta, \quad \frac{\partial y_i^\alpha}{\partial b^\beta} = \delta^{\alpha\beta} \tag{52}$$

$$\text{OPH}: \frac{\partial^2 y_i^\alpha}{\partial K_l^{\eta\kappa}\partial K_j^{\gamma\delta}} = 0, \quad \frac{\partial^2 y_i^\alpha}{\partial K_l^{\gamma\delta}\partial b^\beta} = 0, \quad \frac{\partial^2 y_i^\alpha}{\partial b^\gamma \partial b^\beta} = 0 \tag{53}$$

$$\text{mOPH}: \frac{\partial^2 y_i^\alpha}{\partial K_l^{\delta\eta}\partial x_j^\gamma} = \delta^{\alpha\delta}\delta^{\gamma\eta} I\left(l = \frac{j - s\cdot i + P}{d}\right), \quad \frac{\partial^2 y_i^\alpha}{\partial b^\delta \partial x_j^\gamma} = 0 \tag{54}$$

3.6.3 2D Convolutions

2D convolution layers are almost identical to 1D convolutions, but differ in that their inputs are expected to be 3D tensors, i.e. $\mathbf{x} \in \mathbb{R}^{C\times D_1\times D_2}$ and the convolution kernel is a 4D tensor $\mathbf{K} \in \mathbb{R}^{C'\times C\times M_1\times M_2}$ that is slided over two dimensions of the input and dot multiplied with the

third (the channel dimension). The equations look exactly as for the 1D convolution, but with an extra dimension:

$$y^{\alpha}_{i_1 i_2} = f^{\alpha}_{i_1 i_2}(\mathbf{x}) = \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} \sum_{\beta=0}^{C-1} K^{\alpha\beta}_{k_1 k_2} \tilde{x}^{\beta}_{s_1 \cdot i_1 + d_1 \cdot k_1, s_2 \cdot i_2 + d_2 \cdot k_2} + b^{\alpha} \tag{55}$$

Note that a 2D convolution kernel operates completely independently on the two convolved input dimensions, and the convolution layer can have different strides $s_1, s_2 \in \mathbb{N}^+$, different dilations $d_1, d_2 \in \mathbb{N}^+$ and even different paddings $P_1, P_2 \in \mathbb{N}$ applied to the two dimensions of the convolution. As a consequence, the derivatives can be decomposed into non-interacting terms for the first and for the second convolution dimension. To better illustrate this, consider the OPJ which can be written as

$$\begin{aligned}
\frac{\partial y^{\alpha}_{i_1 i_2}}{\partial K^{\gamma\delta}_{j_1 j_2}} &= \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} \sum_{\beta=0}^{C-1} \frac{\partial K^{\alpha\beta}_{k_1 k_2}}{\partial K^{\gamma\delta}_{j_1 j_2}} \tilde{x}^{\beta}_{s_1 \cdot i_1 + d_1 \cdot k_1, s_2 \cdot i_2 + d_2 \cdot k_2} \\
&= \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} \sum_{\beta=0}^{C-1} \delta^{\alpha\gamma} \delta^{\beta\delta} \delta_{k_1 j_1} \delta_{k_2 j_2} \tilde{x}^{\beta}_{s_1 \cdot i_1 + d_1 \cdot k_1, s_2 \cdot i_2 + d_2 \cdot k_2} \\
&= \delta^{\alpha\gamma} \tilde{x}^{\delta}_{s_1 \cdot i_1 + d_1 \cdot j_1, s_2 \cdot i_2 + d_2 \cdot j_2}
\end{aligned} \tag{56}$$

For the remaining tensors, the derivation is analogous. In total, the derivatives can be summarized as follows:

$$\begin{aligned}
\text{OIJ}: \frac{\partial y^{\alpha}_{i_1 i_2}}{\partial x^{\gamma}_{j_1 j_2}} &= K^{\alpha\gamma}_{k_1 k_2} I\left(k_1 = \frac{j_1 - s_1 \cdot i_1 + P_1}{d_1}\right) I\left(k_2 = \frac{j_2 - s_2 \cdot i_2 + P_2}{d_2}\right) \cdot \\
&\quad I(0 < k_1 \leq M_1) I(0 < k_2 \leq M_2)
\end{aligned} \tag{57}$$

$$\text{OIH}: \frac{\partial y^{\alpha}_{i_1 i_2}}{\partial x^{\delta}_{l} \partial x^{\gamma}_{j_1 j_2}} = 0 \tag{58}$$

$$\text{OPJ}: \frac{\partial y^{\alpha}_{i}}{\partial K^{\gamma\delta}_{j_1 j_2}} = \delta^{\alpha\gamma} \tilde{x}^{\delta}_{s_1 \cdot i_1 + d_1 \cdot j_1, s_2 \cdot i_2 + d_2 \cdot j_2}, \quad \frac{\partial y^{\alpha}_{i_1, i_2}}{\partial b^{\beta}} = \delta^{\alpha\beta} \tag{59}$$

$$\text{OPH}: \frac{\partial^2 y^{\alpha}_{i}}{\partial K^{\eta\kappa}_{l_1 l_2} \partial K^{\gamma\delta}_{j_1 j_2}} = 0, \quad \frac{\partial^2 y^{\alpha}_{i}}{\partial K^{\gamma\delta}_{l_1 l_2} \partial b^{\beta}} = 0, \quad \frac{\partial^2 y^{\alpha}_{i}}{\partial b^{\gamma} \partial b^{\beta}} = 0 \tag{60}$$

$$\begin{aligned}
\text{mOPH}: \frac{\partial^2 y^{\alpha}_{i_1 i_2}}{\partial K^{\delta\eta}_{l_1 l_2} \partial x^{\gamma}_{j_1 j_2}} &= \delta^{\alpha\delta} \delta^{\gamma\eta} I\left(l_1 = \frac{j_1 - s \cdot i_1 + P_1}{d_1}\right) I\left(l_2 = \frac{j_2 - s \cdot i_2 + P_2}{d_2}\right), \\
\frac{\partial^2 y^{\alpha}_{i_1 i_2}}{\partial b^{\delta} \partial x^{\gamma}_{j_1 j_2}} &= 0.
\end{aligned} \tag{61}$$

3.6.4 Transposed Convolutions

Transposed Convolution Layers [Dumoulin and Visin, 2018] (also known as Deconvolution Layers) were introduced to meet the need for upsampling layers in architectures such as Autoencoders and act as an inverse operation to regular convolutions. As seen in the previous two sections, in a convolution any valid subwindow of the input is dot multiplied with a convolution kernel $\mathbf{K}$ striding

over the input image to a single output element. In Transposed Convolutions, the convolution kernel strides over the output and each valid subwindow of the output maps back to a single input element multiplied with multiple kernel elements. More formally, while in a 1D convolution with stride s, dilation d and padding P, an output y_i is connected to an input x_j iff

$$j = s \cdot i + d \cdot k - P, \tag{62}$$

in a transposed convolution the connection relation can be obtained through inversion of output and input, i.e. an output y_i is connected to an input x_j iff

$$i = s \cdot j + d \cdot k - P. \tag{63}$$

Solving for j and replacing the index of the input x in equation (41) with the expression j, it follow we can also write a simple 1D transposed convolution as

$$y_i = f_i(\mathbf{x}) = \mathbf{K} \overset{T}{\star} \mathbf{x} + \mathbf{b} = \sum_{k=0}^{M-1} K_k x_{\lfloor \frac{i-k\cdot d+P}{s} \rfloor} + b_i \tag{64}$$

Thus, the full formula for transposed convolutions becomes:

$$y_i^\alpha = f_i^\alpha(\mathbf{x}) = \sum_{k=0}^{M-1} K_k^{\alpha\beta} \tilde{x}^\beta_{\lfloor \frac{i-k\cdot d}{s} \rfloor} + b^\alpha. \tag{65}$$

Aside from the flipped output and input dimension indices, the expressions for the local derivatives of the convolution layer are therefore analogous to those of the 1d convolution layer and take the form

$$\text{OIJ} : \frac{\partial y_i^\alpha}{\partial x_j^\gamma} = K_k^{\alpha\gamma} I\left(k = \frac{j \cdot s - i + P}{d}\right) I(0 \leq k < M), \quad \text{OIH} : \frac{\partial^2 y_i^\alpha}{\partial x_l^\delta \partial x_k^\gamma} = 0 \tag{66}$$

$$\text{OPJ} : \frac{\partial y_i^\alpha}{\partial K_j^{\gamma\delta}} = \delta^{\alpha\gamma} \tilde{x}^\delta_{\lfloor \frac{i-j\cdot d}{s} \rfloor}, \quad \frac{\partial y_i^\alpha}{\partial b^\beta} = \delta^{\alpha\beta} \tag{67}$$

$$\text{OPH} : \frac{\partial^2 y_i^\alpha}{\partial K_l^{\eta\kappa} \partial K_j^{\gamma\delta}} = 0, \quad \frac{\partial^2 y_i^\alpha}{\partial K_l^{\gamma\delta} \partial b^\beta} = 0, \quad \frac{\partial^2 y_i^\alpha}{\partial b^\gamma \partial b^\beta} = 0 \tag{68}$$

$$\text{mOPH} : \frac{\partial^2 y_i^\alpha}{\partial K_l^{\delta\eta} \partial x_j^\gamma} = \delta^{\alpha\delta} \delta^{\gamma\eta} I\left(l = \frac{j \cdot s - i + P}{d}\right), \quad \frac{\partial^2 y_i^\alpha}{\partial b^\delta \partial x_j^\gamma} = 0 \tag{69}$$

For 2D transposed convolutions, the kernel operates independently on each input dimension (just like for regular 2D convolutions) and thus, the formulas for a 2D convolution layer can be transferred to the formulas for a 2D transposed convolution layer in the same way as for 1D equivalents. For the sake of brevity, we will omit the 2D formulas here.

3.6.5 Batch Normalization

Batch Normalization is a widespread layer used in Deep Learning architectures and was proposed by Sergey Ioffe and Christian Szegedy to solve the problem of covariate shits in deeper neural network layers and thereby increase model performance[Ioffe and Szegedy, 2015]. Although newer

research seems to suggest that Batch Normalization does little to reduce covariate shift and the performance benefits may instead come from other mechanisms such as a smoothing of the loss landscape [Santurkar et al., 2019], it has still found its way into the default toolbox of Deep Learning researchers and practitioners. It works by computing the mean and standard deviation of the neural network activations over a minibatch of samples and then removing the mean and renormalizing the activations to end up with a mean-free, normalized distribution of activations. At training time, for an input $\mathbf{x} \in \mathbb{R}^{B\times C}$ and parameters $\beta, \gamma \in \mathbb{R}^C$ [2], the output of a Batch Normalization layer can be described as

$$y_i^k = \gamma^k \hat{x}_i^k + \beta^k, \qquad \hat{x}_i^k = \frac{x_i^k - \mu_{\mathcal{B}}^k}{\sqrt{(\sigma_{\mathcal{B}}^k)^2 + \epsilon}}, \tag{70}$$

where $\mu_{\mathcal{B}}^k$, $\sigma_{\mathcal{B}}^k$ are the mean and standard deviation calculated over all elements of the minibatch for the given channel k, i.e.

$$\mu_{\mathcal{B}}^k = \sum_{b=1}^{B} \frac{1}{B} x_b^k, \qquad \sigma_{\mathcal{B}}^2 = \sum_{b=1}^{B} \frac{1}{B} \left(x_b^k - \mu_{\mathcal{B}}^k\right)^2. \tag{71}$$

Note that in all calculations of the Output-Input-Jacobian up until this point, we have been calculating partial derivatives $\frac{\partial y_i}{\partial x_j}$, because we implicitly assumed that the expression for the outputs contained no hidden dependencies on the inputs through other variables, i.e. $\mathbf{y} = f(\mathbf{x})$. However, since the expression for the output in equation (70) contains variables $\mu_{\mathcal{B}}^K$ and $\sigma_{\mathcal{B}}^k$ which are implicitly dependent on the inputs - $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\mu}_{\mathcal{B}}(\mathbf{x}))$,

The last term can be omitted due to the derivative of the variance with respect to the mean being zero:

$$\frac{\partial(\sigma_{\mathcal{B}}^m)^2}{\partial \mu_{\mathcal{B}}^m} = \sum_{b=1}^{B} \frac{2}{B}(x_b^m - \mu_{\mathcal{B}}^m) = 2\left(\sum_{b=1}^{B} \frac{x_b^m}{B} - \sum_{b=1}^{B} \frac{\mu_{\mathcal{B}}^m}{B}\right) = 2\left(\mu_{\mathcal{B}}^m - B\frac{\mu_{\mathcal{B}}^m}{B}\right) = 0. \tag{72}$$

Writing $\nu_{\mathcal{B}}^k := ((\sigma_{\mathcal{B}}^k)^2 + \epsilon)$, we can resolve the above derivatives

$$\frac{\partial \hat{x}_i^k}{\partial x_j^l} = (\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} \frac{\partial x_i^k}{\partial x_j^l} = (\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} \delta_{ij}\delta^{kl} \tag{73}$$

$$\frac{\partial \hat{x}_i^k}{\partial(\mu_{\mathcal{B}}^m)} = (\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} \frac{-\partial \mu_{\mathcal{B}}^k}{\partial \mu_{\mathcal{B}}^m} = -(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} \delta^{km} \tag{74}$$

$$\frac{\partial \mu_{\mathcal{B}}^m}{\partial x_j^l} = \frac{1}{B}\sum_{b=1}^{B} \frac{\partial x_b^m}{\partial x_j^l} = \frac{\delta^{ml}}{B} \tag{75}$$

$$\frac{\partial \hat{x}_i^k}{\partial(\sigma_{\mathcal{B}}^m)^2} = (x_i^k - \mu_{\mathcal{B}}^k)\left(-\frac{1}{2}\right)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\delta^{km} \tag{76}$$

$$\frac{\partial(\sigma_{\mathcal{B}}^m)^2}{\partial x_j^l} = \frac{1}{B}\sum_{b=1}^{B} \frac{\partial(x_b^m - \mu_{\mathcal{B}}^m)^2}{\partial x_j^l} = \frac{1}{B}\sum_{b=1}^{B} 2(x_b^m - \mu_{\mathcal{B}}^m)\frac{\partial x_b^m}{\partial x_j^l} = \frac{2}{B}(x_j^l - \mu_{\mathcal{B}}^l)\delta^{ml}, \tag{77}$$

[2] Although we have used symbol γ for the learning rate earlier, this γ is a parameter of the Batch Normalization layer, left unchanged out of notation convention.

and inserting back into equation (**??**), we obtain

$$\begin{aligned}\frac{d\hat{x}_i^k}{dx_j^l} &= (\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}\delta_{ij}\delta^{kl} - \frac{1}{B}\sum_m (\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}\delta^{km}\delta^{ml} - \frac{1}{B}\sum_m (x_i^k - \mu_{\mathcal{B}}^k)(x_j^l - \mu_{\mathcal{B}}^l)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\delta^{km}\delta^{ml} \\ &= \delta^{kl}\left(\delta_{ij} - \frac{1}{B}\right)(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} - \frac{1}{B}\delta^{kl}(x_i^k - \mu_{\mathcal{B}}^k)(x_j^k - \mu_{\mathcal{B}}^k)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}. \end{aligned} \tag{78}$$

For the Output-Input-Hessian, we have two derivatives to compute.

$$\frac{d^2\hat{x}_i^k}{dx_s^r dx_j^l} = \delta^{kl}\left(\delta_{ij} - \frac{1}{B}\right)\frac{d}{dx_s^r}(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} - \frac{1}{B}\delta^{kl}\frac{d}{dx_s^r}(x_i^k - \mu_{\mathcal{B}}^k)(x_j^k - \mu_{\mathcal{B}}^k)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}. \tag{79}$$

The derivative for the square root term can be resolved by using the total differential rule again. Note that the terms involving transitive dependencies of the variance on the inputs is obviated, since we have already found the partial derivative of the standard deviation with respect to the mean to be zero in equation (72). Furthermore, since the square root term does not contain any explicit dependency on the inputs or the means (only implicit), the first two terms become 0 and we are left with two derivatives (1) and (2) to compute:

$$\frac{d(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}}{dx_s^r} = \underbrace{\frac{\partial(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}}{\partial x_s^r}}_{=0} + \sum_m \frac{\partial\mu_{\mathcal{B}}^m}{\partial x_s^r}\underbrace{\frac{\partial(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}}{\partial\mu_{\mathcal{B}}^m}}_{=0} + \sum_m \underbrace{\frac{\partial(\sigma_{\mathcal{B}}^m)^2}{\partial x_s^r}}_{(1)}\underbrace{\frac{\partial(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}}{\partial(\sigma_{\mathcal{B}}^m)^2}}_{(2)}. \tag{80}$$

For the first derivative term, we have already derived an expression in (77) for which we only have to substitute the indices, and the second derivative term can be simplified to obtain

$$(1): \quad \frac{\partial(\sigma_{\mathcal{B}}^m)^2}{\partial x_s^r} = \frac{2}{B}(x_s^r - \mu_{\mathcal{B}}^r)\delta^{mr} \tag{81}$$

$$(2): \quad \frac{\partial(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}}{\partial(\sigma_{\mathcal{B}}^m)^2} = -\frac{1}{2}(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\frac{\partial(\sigma_{\mathcal{B}}^k)^2}{\partial(\sigma_{\mathcal{B}}^m)^2} = -\frac{1}{2}(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\delta^{km}. \tag{82}$$

Inserting back into (80), we find

$$\frac{d(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}}}{dx_s^r} = -\frac{1}{B}\sum_m (x_s^r - \mu_{\mathcal{B}}^r)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\delta^{km}\delta^{mr} = -\frac{1}{B}(x_s^r - \mu_{\mathcal{B}}^r)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\delta^{kr}. \tag{83}$$

The second derivative term of equation (79) is a bit more complex to compute:

$$\frac{d}{dx_s^r}\left((x_i^k - \mu_{\mathcal{B}}^k)(x_j^k - \mu_{\mathcal{B}}^k)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\right) = \underbrace{\frac{\partial(...)}{\partial x_s^r}}_{(1)} + \sum_m \underbrace{\frac{\partial\mu_{\mathcal{B}}^m}{\partial x_s^r}}_{(2)}\underbrace{\frac{\partial(...)}{\partial\mu_{\mathcal{B}}^m}}_{(3)} + \sum_m \underbrace{\frac{\partial(\sigma_{\mathcal{B}}^m)^2}{\partial x_s^r}}_{(4)}\underbrace{\frac{\partial(...)}{\partial(\sigma_{\mathcal{B}}^m)^2}}_{(5)} \tag{84}$$

For derivative (4), we can use (81) without any modifications. For derivative (2), we can substitute the indices in expression (75) to arrive at

$$\frac{\partial\mu_{\mathcal{B}}^m}{\partial x_s^r} = \frac{1}{B}\delta^{mr} \tag{85}$$

We resolve the unknown derivatives (1), (3), and (5) as follows:

$$(1): \quad \frac{\partial}{\partial x_s^r}\frac{(x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} = \frac{\delta_{is}\delta^{kr}(x_j^k-\mu_{\mathcal{B}}^k)+\delta_{js}\delta^{kr}(x_i^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} \tag{86}$$

$$(3): \quad \frac{\partial}{\partial \mu_{\mathcal{B}}^m}\frac{(x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} = \frac{-\delta^{km}(x_j^k-\mu_{\mathcal{B}}^k)-\delta^{km}(x_i^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} \tag{87}$$

$$(5): \quad \frac{\partial}{\partial (\sigma_{\mathcal{B}}^m)^2}\frac{(x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} = \left(-\frac{3}{2}\right)\frac{(x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{5}{2}}}\delta^{km} \tag{88}$$

Inserting the expressions for derivatives (1), (2), (3), (4), and (5) back into equation (84), we obtain

$$\begin{aligned}\frac{d}{dx_s^r}\left((x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}\right) &= \delta^{kr}\frac{\delta_{is}(x_j^k-\mu_{\mathcal{B}}^k)+\delta_{js}(x_i^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} \\ &\quad - \frac{1}{B}\delta^{kr}\frac{(x_j^k-\mu_{\mathcal{B}}^k)+(x_i^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} \\ &\quad - \frac{3}{B}\delta^{kr}\frac{(x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)(x_s^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{5}{2}}}.\end{aligned} \tag{89}$$

In total:

$$\begin{aligned}\frac{d^2\hat{x}_i^k}{dx_s^r dx_j^l} = -\frac{1}{B}\delta^{kl}\delta^{kr}\Bigg[&\left(\delta_{ij}-\frac{1}{B}\right)\frac{(x_s^r-\mu_{\mathcal{B}}^r)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} - \Bigg(\frac{\delta_{is}(x_j^k-\mu_{\mathcal{B}}^k)+\delta_{js}(x_i^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} \\ &-\frac{1}{B}\frac{(x_j^k-\mu_{\mathcal{B}}^k)+(x_i^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{3}{2}}} - \frac{3}{B}\frac{(x_i^k-\mu_{\mathcal{B}}^k)(x_j^k-\mu_{\mathcal{B}}^k)(x_s^k-\mu_{\mathcal{B}}^k)}{((\sigma_{\mathcal{B}}^k)^2+\epsilon)^{\frac{5}{2}}}\Bigg)\Bigg].\end{aligned} \tag{90}$$

Using these, we can write the entirety of the local derivatives as follows:

$$\text{(OIJ):} \quad \frac{dy_i^k}{dx_j^l} = \gamma^k\frac{d\hat{x}_i^k}{dx_j^l}, \qquad \text{(OIH):} \quad \frac{d^2y_i^k}{dx_s^r dx_j^l} = \gamma^k\frac{d^2\hat{x}_i^k}{dx_s^r dx_j^l} \tag{91}$$

$$\text{(OPJ):} \quad \frac{dy_i^k}{d\gamma^l} = \delta^{kl}\hat{x}_i^k, \quad \frac{dy_i^k}{d\beta^k} = \delta^{kl} \tag{92}$$

$$\text{(OPH):} \quad \frac{d^2y_i^k}{d\gamma^r d\gamma^l\partial} = 0, \quad \frac{d^2y_i^k}{d\gamma^r d\beta^l} = 0, \quad \frac{d^2y_i^k}{d\beta^r d\beta^l} = 0 \tag{93}$$

$$\text{(mOPH):} \quad \frac{d^2y_i^k}{d\gamma^r dx_j^l} = \delta^{kr}\frac{d\hat{x}_i^k}{dx_j^l}, \quad \frac{d^2y_i^k}{d\beta^r dx_j^l} = \delta^{kr}\frac{d\hat{x}_i^k}{dx_j^l} \tag{94}$$

For 1D Batch Normalization with an additional input dimension (i.e. operating on input tensors $x \in \mathbb{R}^{B\times C\times D}$), the derivation of the expression is equivalent, but the statistics are calculated not only over the batch dimension, but also over the additional input dimension. Thus, the expressions can be obtained simply by substituting the batch indices with 2-dimensional multi-indices, and the number of elements B with BC, as demonstrated below for the OIJ (for pre-output $\hat{x}$):

$$\frac{d\hat{x}_{i_1,i_2}^k}{dx_{j_1,j_2}^l} = \delta^{kl}\left(\delta_{i_1,j_1}\delta_{i_2,j_2}-\frac{1}{BC}\right)(\nu_{\mathcal{B}}^k)^{-\frac{1}{2}} - \frac{1}{BC}\delta^{kl}(x_{i_1,i_2}^k-\mu_{\mathcal{B}}^k)(x_{j_1,j_2}^k-\mu_{\mathcal{B}}^k)(\nu_{\mathcal{B}}^k)^{-\frac{3}{2}}. \tag{95}$$

For 2D and higher-order Batch Normalization, the expressions are analogous, but with $(D+1)$-dimensional multi-indices instead.

3.7 Nonparametric Layers

3.7.1 Activation Functions

For our experiments, we use two different activation functions: Sigmoid and ReLU. Although many other activation functions exist, we focus on these as they are arguably the most relevant and common activation functions in modern deep learning.

$$\mathbf{f}(\mathbf{x}) = \mathrm{ReLU}(\mathbf{x}) = \max(0, \mathbf{x}) \tag{96}$$
$$\mathbf{J}_x^y = \mathrm{ReLU}'(\mathbf{x}) = \Theta(\mathbf{x}) \tag{97}$$
$$\mathbf{H}_{xx}^y = \mathrm{ReLU}''(\mathbf{x}) = 0, \tag{98}$$

where Θ is the Heaviside step function.

$$\mathbf{f}(\mathbf{x}) = \mathrm{Sigmoid}(\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{x})} \tag{99}$$
$$\mathbf{J}_x^y = \mathrm{Sigmoid}'(\mathbf{x}) = \mathrm{Sigmoid}(\mathbf{x})(1 - \mathrm{Sigmoid}(\mathbf{x})) \tag{100}$$
$$\mathbf{H}_{xx}^y = \mathrm{Sigmoid}''(\mathbf{x}) = \mathrm{Sigmoid}(\mathbf{x})(1 - \mathrm{Sigmoid}(\mathbf{x}))(1 - 2\,\mathrm{Sigmoid}(\mathbf{x})) \tag{101}$$

All of the above functions should be interpreted as element-wise, i.e. $y_i = f(x_i)$.

3.7.2 Pooling Layers

A special case of non-parametric functions are Pooling layers such as AveragePooling and MaxPooling layers. These are comparable to convolutions in that they can be understood as a function moving over the input, but contrary to convolution layers there are no learnable parameters (no learnable kernel and no bias at all) and there is no action on a channel dimension. The action of an AveragePooling layer of window size M, stride s, dilation d and padding P on an input $\boldsymbol{y} \in \mathbb{R}^D$ is given by

$$y_i = f_i(\mathbf{x}) = \frac{1}{M} \sum_{k=0}^{M-1} \tilde{x}_{s\cdot i + d\cdot k}, \tag{102}$$

where $\tilde{\boldsymbol{x}}$ is the padded input vector defined as in equation (43). This leads to the local derivatives

$$\begin{aligned} \mathrm{OIJ}: \frac{\partial y_i}{\partial x_j} &= \frac{1}{M} \sum_{k=0}^{M-1} \frac{\partial \tilde{x}_{s\cdot i + d\cdot k}}{\partial x_j} = \frac{1}{M} I\left(j = s\cdot i + d\cdot k \wedge 0 \le k < M\right) \\ &= \frac{1}{M} I\left(0 \le \frac{j - s\cdot i}{d} < M\right) \end{aligned} \tag{103}$$
$$\mathrm{OIH}: \frac{\partial y_i}{\partial x_k \partial x_j} = 0. \tag{104}$$

For MaxPooling, the layer action can be expressed as

$$y_i = f_i(\mathbf{x}) = \max_{k \in \{0,1,\ldots,M-1\}} (\tilde{x}_{s \cdot i + d \cdot k}) \,, \tag{105}$$

i.e. each output node is connected to exactly one input node from its respective subwindow, namely the node with the maximum value, thus giving us the local derivatives

$$\text{OIJ} : \frac{\partial y_i}{\partial x_j} = I\left(j = \operatorname*{argmax}_{k \in \{0,1,\ldots,M-1\}} (\tilde{x}_{s \cdot i + d \cdot k})\right), \quad \text{OIH} : \frac{\partial^2 y_i}{\partial x_k \partial x_j} = 0. \tag{106}$$

In 2D pooling layers, the window operates independently on both input dimensions, as was the case with 2D convolution layers. Therefore, for a window size $M_1 \times M_2$, the respective local derivatives of the 2D AveragePool layer are

$$\text{OIJ} : \frac{\partial y_i}{\partial x_{j_1 j_2}} = \frac{1}{M_1 M_2} I\left(0 \leq \frac{j_1 - s_1 \cdot i}{d_1} < M_1\right) I\left(0 \leq \frac{j_2 - s_2 \cdot i}{d_2} < M_2\right), \tag{107}$$

$$\text{OIH} : \frac{\partial^2 y_i}{\partial x_{k_1 k_2} \partial x_{j_1 j_2}} = 0. \tag{108}$$

The expressions for the 2D Maxpooling layer are obtained analogously.

3.8 Beyond Sequential Architectures

3.8.1 Parallel Layers

So far, we have described Second Order Backpropagation for purely linearly sequential feedforward ANN architectures. While this covers a great deal of applications, it may be interesting to consider architectures where the output of one layer is forked to be fed as input to not just one, but several subsequent layers, i.e. there is a layer $n-1$ and subsequent layers $(1,n),(2,n),...,(N,n)$ all satisfying

$$\mathbf{y}^{(\alpha,n)} = \mathbf{f}^{(\alpha,n)}(\mathbf{x}^{(n-1)}). \tag{109}$$

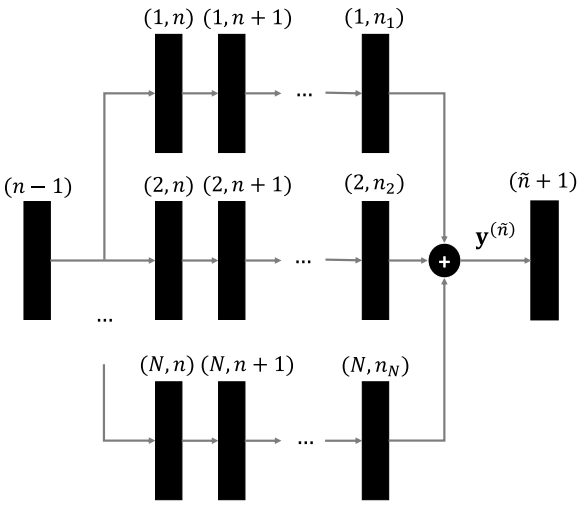


Figure 2: Feedforward ANN with parallel layer structure.

In First Order Backpropagation, this is easy enough, with the gradients at the fork simply adding up. Unfortunately, for the Hessian with respect to the loss in Second Order Backpropagation this is not the case, since the expression contains a quadratic interaction term of the parallel layers. Consider a neural network with a layer structure as featured in Figure 2, with N parallel, sequential subnetworks, each subnetwork having a depth of n_α layers. Let us define $\tilde{n} := \max\{n_\alpha | 1 \leq \alpha \leq N\}$. As already shown, the Loss-Input-Hessian can be written as

$$\frac{\partial^2 \mathcal{L}}{\partial x_i \partial x_j} = \sum_k \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial^2 y_k}{\partial x_i \partial x_j} + \sum_{k,l} \frac{\partial y_l}{\partial x_i} \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \frac{\partial y_k}{\partial x_j}. \tag{110}$$

Next, we expand the expressions using $y_k = \sum_\alpha y_k^\alpha$ to find:

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial x_i \partial x_j} &= \sum_k \frac{\partial^2 \mathcal{L}}{\partial y_k} \left(\frac{\partial}{\partial x_i \partial x_j} \sum_\alpha y_k^\alpha \right) + \sum_{k,l} \left(\frac{\partial}{\partial x_i} \sum_\beta y_l^\beta \right) \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \left(\frac{\partial}{\partial x_j} \sum_\alpha y_k^\alpha \right) \\
&= \sum_\alpha \sum_k \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial^2 y_k^\alpha}{\partial x_i \partial x_j} + \sum_{k,l} \sum_{\alpha,\beta} \frac{\partial y_l^\beta}{\partial x_i} \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \frac{\partial y_k^\alpha}{\partial x_j} \\
&= \sum_\alpha \underbrace{\left(\sum_k \frac{\partial \mathcal{L}}{\partial y_k} \frac{\partial^2 y_k^\alpha}{\partial x_i \partial x_j} + \sum_{k,l} \frac{\partial y_l^\alpha}{\partial x_i} \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \frac{\partial y_k^\alpha}{\partial x_j} \right)}_{=:T_1} + \underbrace{\sum_{\alpha \neq \beta} \sum_{k,l} \frac{\partial y_l^\beta}{\partial x_i} \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \frac{\partial y_k^\alpha}{\partial x_j}}_{=:T_2}
\end{aligned} \tag{111}$$

Now, for more clarity let us define $\mathbf{y}^{(\alpha,n-1)} := \mathbf{x}$ and $\mathbf{y}^{(\alpha,n_\alpha)} := \mathbf{y}^\alpha$ to denote that $\mathbf{x}$ is the input to subnetwork α, while $\mathbf{y}^\alpha$ is the output at layer n_α of subnetwork α. We observe that the first terms T_1 would correspond to the Hessian at the input layer in a purely linearly sequential subnetwork α,

i.e.

$$T_1 = \sum_k \frac{\partial \mathcal{L}}{\partial y_k^{(\alpha,n_\alpha)}} \frac{\partial^2 y_k^{(\alpha,n_\alpha)}}{\partial y_i^{(\alpha,n-1)} \partial y_j^{(\alpha,n-1)}} + \sum_{k,l} \frac{\partial y_l^{(\alpha,n_\alpha)}}{\partial y_i^{(\alpha,n-1)}} \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \frac{\partial y_k^{(\alpha,n_\alpha)}}{\partial y_j^{(\alpha,n-1)}} \equiv \mathbf{H}^{(\alpha,n-1)} \tag{112}$$

Furthermore, we notice we can rewrite the second term using the cOIJs $\mathbf{U}^{(\alpha,0),(\alpha,n_\alpha)}$ and $\mathbf{U}^{(\beta,0),(\beta,n_\beta)}$. Writing $\mathbf{U}^\alpha$ and $\mathbf{U}^\beta$ for short, we arrive at

$$T_2 = \sum_{k,l} \frac{\partial y_l^{(\beta,n_\beta)}}{\partial y_i^{(\beta,n-1)}} \frac{\partial^2 \mathcal{L}}{\partial y_l \partial y_k} \frac{\partial y_k^{(\alpha,n_\alpha)}}{\partial y_j^{(\alpha,n-1)}} = (\mathbf{U}^\beta)^T \cdot \mathbf{H}^{(\tilde{n})} \cdot \mathbf{U}^\alpha. \tag{113}$$

Introducing the matrix

$$\boldsymbol{U}^{\boldsymbol{\Sigma}} := \sum_\alpha \mathbf{U}^\alpha, \tag{114}$$

we can put everything together to find

$$\begin{aligned}
\left\{ \frac{\partial^2 \mathcal{L}}{\partial x_i \partial x_j} \right\}_{ij} &= \sum_\alpha \mathbf{H}^{(\alpha,n-1)} + \sum_{\beta \neq \alpha} (\mathbf{U}^\beta)^T \mathbf{H}^{(\tilde{n})} \mathbf{U}^\alpha \\
&= \sum_\alpha \mathbf{H}^{(\alpha,n-1)} + \left(\sum_\beta (\mathbf{U}^\beta)^T \right) \cdot \mathbf{H}^{(\tilde{n})} \cdot \left(\sum_\alpha \mathbf{U}^\alpha \right) - \sum_\alpha (\mathbf{U}^\alpha)^T \mathbf{H}^{(\tilde{n})} \mathbf{U}^\alpha \\
&= \sum_\alpha \mathbf{H}^{(\alpha,n-1)} + (\boldsymbol{U}^{\boldsymbol{\Sigma}})^T \cdot \mathbf{H}^{(\tilde{n})} \cdot \boldsymbol{U}^{\boldsymbol{\Sigma}} - \sum_\alpha (\mathbf{U}^\alpha)^T \cdot \mathbf{H}^{(\tilde{n})} \cdot \mathbf{U}^\alpha.
\end{aligned} \tag{115}$$

For an algorithmic implementation, this means that each layer should multiply their OIJ with the next layer's cOIJ to obtain its own cOIJ.

3.9 Residual Connections

Residual connections represent a particularly simple instance of a parallel network architecture. Let $\mathbf{h}^{(n)}, ..., \mathbf{h}^{(n+m)}$ be successive layers of a neural network and let $\mathbf{F} := \mathbf{h}^{(n+m)} \circ ... \mathbf{h}^{(n+1)} \circ \mathbf{h}^{(n)}$ define the concatenation of these layers. Then, the output of this subnetwork $\mathbf{F}$ with an additional residual connection is given by

$$\mathbf{y}^{(n+m)} = \mathbf{y}^{(n)} + \mathbf{F}(\mathbf{y}^{(n)}). \tag{116}$$

This means we have two parallel networks $\mathbf{f}^{(n,1)} = \mathbf{F}$ and the identity network $\mathbf{f}^{(n,2)} = \mathbf{id}$. Thus, setting $\tilde{n} = n + m$, the matrices for the residual connection itself are:

$$\mathbf{U}^2 = \mathbf{J}_x^{(n_2)} = \mathbf{I} \tag{117}$$

$$\boldsymbol{H}_x^{(2,n_2)} = \mathbf{g}^{(n_2)T} \cdot \underbrace{\mathbf{H}_{xx}^{y,(n_2)}}_{=\mathbf{0}} + \mathbf{J}_x^{(n_2)T} \cdot \mathbf{H}^{(n_2)} \cdot \mathbf{J}_x^{(n_2)} = \mathbf{H}^{(\tilde{n})}. \tag{118}$$

Using the above equation and inserting it into the expression for $\boldsymbol{U}^{\boldsymbol{\Sigma}}$ and subsequently into equation (115), and making use of the symmetry of $\mathbf{H}^{(\tilde{n})}$ such that $\boldsymbol{A}^T \mathbf{H}^{(\tilde{n})} = (\mathbf{H}^{(\tilde{n})} \mathbf{A})^T$:

$$\left\{ \frac{\partial^2 \mathcal{L}}{\partial x_i \partial x_j} \right\}_{ij} = \boldsymbol{H}^{(1,n-1)} + \mathbf{H}^{(\tilde{n})} + \mathbf{I} \cdot \mathbf{H}^{(\tilde{n})} \cdot \mathbf{U}^1 + \mathbf{U}^{1T} \cdot \mathbf{H}^{(\tilde{n})} \cdot \mathbf{I} \tag{119}$$

$$= \mathbf{H}^{(1,n-1)} + \mathbf{H}^{(\tilde{n})} + \mathbf{H}^{(\tilde{n})} \cdot \mathbf{U}^1 + (\mathbf{H}^{(\tilde{n})} \cdot \mathbf{U}^1)^T \tag{120}$$

3.10 Exclusions

Although we have done our best to cover a wide array of layers currently used in Deep Learning architectures, a multitude of other layers exist, among them LayerNormalization [Ba et al., 2016], Attention Layers [Vaswani et al., 2017], and different kinds of sampling layers used in generative models to name just a few. These have been excluded from the thesis' body of work both due to time constraints and for practical reasons: Attention Layers in particular compute tensors with third-order dependencies on three tensor inputs (dot multiplied with keys, value and query matrices respectively), which would lead to local derivatives with few non-zero elements and as we will see in the next chapter, this is a critical requirement for realizing an implementation that can compete with ordinary second-order backpropagation techniques.

4 Implementation

In this chapter, we discuss the implementation of Stagewise Second Order Backpropagation that we will use in the experiments of the subsequent chapter to calculate the Hessian and Hessian approximations.

4.1 O2Grad - A Package for Second Order Backprop

To assess the practical viability of using Second Order Backpropagation to calculate the Hessian (and approximations thereof), we implemented a package based on PyTorch which we call **O2Grad** - a combination of *autograd*, the autodifferentiation package used in PyTorch, and *O2*, which is short for *Order Two*. The motivation for implementing this package is that at the time of writing the thesis, none of the currently leading ML frameworks support Second Order Backpropagation (as investigated in [Mizutani et al., 2005] and expanded upon in this thesis) out of the box. Our package was implemented with the following requirements in mind:

1. **Extensibility**: It should be possible to extend the package to support layers and architectures beyond the ones discussed in the previous section.
2. **Reach**: The package was supposed to be usable for as many ML researchers as possible.
3. **Ease of use**: The package should integrate as smoothly as possible with pre-existing ML frameworks and workflows.

With these in mind, we decided to implement the package building upon PyTorch rather than competing autodifferentiation ML frameworks Tensorflow/Keras [3] and JAX [4]. The reasons for this decision were as follows: (1) We are **familiar with the API** of PyTorch and thus expected the implementation to be the simplest there. (2) At the moment we started working on the thesis, **Tensorflow/Keras** was **lacking flexibility** compared to PyTorch, not supporting some critical functionality for our use case. For instance, it was only possible to exchange layers with wrapped layers in a model only to a limited extent (a requirement which is important for design reasons that we will discuss later). (3) Although **JAX** can be argued to have the same level of functionality as PyTorch, it is still a comparatively **young and experimental** ML framework. It also does not

[3] `https://github.com/tensorflow/tensorflow`
[4] `https://github.com/google/jax`

provide implementations for neural networks out of the box (although libraries building on top of JAX such as FLAX [5] and Haiku [6] do). (4) Finally, the code for a majority of papers currently being published in major ML conferences up until recently has been written using PyTorch [He, 2019], which seems to suggest that **PyTorch** is **currently the most popular framework** for ML research.

A version of the package is available along with the rest of the code written for this thesis on a public GitHub repository [7]. Possible future versions will be provided on a separate repository [8].

4.2 Design

Adhering to the requirement of extensibility, we wanted the workflow using O2Grad to diverge as little as possible from using regular PyTorch. However, the layers provided in Pytorch had to be expanded with functions to compute the local derivatives discussed in section 3.3, since there is no out-of-the-box way to do this in Pytorch. Therefore, we designed the general workflow to be as follows:

```
model = nn.Sequential(...)    #Define a sequential model.
criterion = nn.MSELoss()    #Define a loss criterion.
o2model = O2Model(model, criterion)    #Prepare the model.
o2model.zero_grad() #Set the model gradients to zero.
yhat = o2model(x)    #Compute prediction from feed-forward step on input.
loss = o2model.criterion(yhat, y)    #Compute prediction loss.
loss.backward(create_graph=True)    #Backpropagate loss
hessian = o2model.get_hessian()    #Retrieve Hessian of the model.
```

With the exception of steps 3 and 8, these steps are exactly how one would go about computing the gradients of a model in Vanilla PyTorch. Step 3 is required to wrap the model's layers and the loss criterion in matching O2Grad layers and loss criteria that will allow for Second Order Backpropagation. Three mayor base classes are required to make this possible: `O2Loss` and `O2Module`, and `O2Model` to wrap everything together, with the second class branching into further subclasses `O2Layer`, `O2ParametricLayer` and `O2Container`. Figure 3 illustrates the relationships between the classes of the O2Grad module graphically.

4.2.1 O2Loss

`O2Loss` is a wrapper class that extends from PyTorch's `torch.nn.Module`. Its instances are created by passing an instance of `torch.nn.Module` that computes a loss. It is used to enable Second Order Backpropagation for Pytorch Loss modules and exposes the three core functions

1. `def get_loss_input_jacobian(self, x: torch.Tensor, t: torch.Tensor) -> torch.Tensor`

[5] https://github.com/google/flax
[6] https://github.com/deepmind/dm-haiku
[7] https://github.com/luisherrmann/thesis2021
[8] https://github.com/luisherrmann/o2grad

2. `def get_loss_input_hessian(self, x: torch.Tensor, t: torch.Tensor) -> torch.Tensor | SparseSymmetricMatrix`

3. `def get_loss_input_hessian_cached(self) -> Optional[torch.Tensor]`

which get an input tensor and a target tensor to calculate the respective OIJ and OIH, respectively. The two quantities can be calculated by fetching the input x in the forward step and calculating the Loss-Input-Jacobian and the Loss-Input-Hessian using the integrated PyTorch functions `jacobian()` and `hessian()` from the `autograd.functional` subpackage.

4.2.2 O2Layer

`O2Layer` is an abstract class that extends from PyTorch's `torch.nn.Module`. It is used to provide an interface for atomic layers. It exposes the core functions

1. `def get_output_input_jacobian(self, x: torch.Tensor) -> torch.Tensor`

2. `def get_output_input_hessian(self, x: torch.Tensor) -> torch.Tensor`

3. `def get_loss_input_hessian(self, dLdy: torch.Tensor, dL2dy2: torch.Tensor | SparseSymmetricMatrix, dydx: torch.Tensor, dy2dx2: torch.Tensor, delete: Sequence[str] = []) -> torch.Tensor`

4. `def get_chained_output_input_jacobian_cached(self) -> Optional[torch.Tensor]`

For `get_output_input_jacobian()` and `get_output_input_hessian()`, the abstract class provides default implementations using the functions `jacobian()` and `hessian()`, which can be used to obtain the respective Jacobian and Hessian using automatic differentiation, but we observed that this base implementation is very slow compared to directly computing the Jacobian and Hessian. Therefore, the idea is that modules for specific layers with O2Grad support extend from this abstract class and provide their own implementation of these two core functions. The method `get_loss_input_hessian()` is used to compute the LIH of the layer, and a default implementation is also provided here. However, classes implementing this abstract class should not override this method unless a more efficient implementation specific to the respective layer class is desired. The second method is important for composed layers and will be discussed further below. Currently supported modules that extend directly from this class are

1. `O2Reshape`
2. `O2Sigmoid`, `O2ReLU`
3. `O2MaxPool1d`, `O2AvgPool1d`
4. `O2MaxPool2d`, `O2AvgPool2d`

4.2.3 O2ParametricLayer

`O2ParametricLayer` is an abstract class that extends from `O2Layer` and, in addition to all methods exposed by `O2Layer`, also exposes the four core methods

1. ```
   def get_output_param_jacobian(self,
       x: torch.Tensor) -> torch.Tensor
   ```

2. ```
   def get_output_param_hessian(self,
       x: torch.Tensor) -> torch.Tensor
   ```

3. ```
   def get_mixed_output_param_hessian(self,
       x: torch.Tensor) -> torch.Tensor
   ```

4. ```
   def get_loss_param_hessian(self, dLdy: torch.Tensor,
       dL2dy2: torch.Tensor | SparseSymmetricMatrix,
       dydw: torch.Tensor,
       dy2dw2: torch.Tensor,
       delete: Sequence[str] = []
   ) -> torch.Tensor | SparseSymmetricMatrix
   ```

with the first three method returning the OPJ, OPH and mOPH, respectively. Unlike for the methods to compute the OIJ and OIH, a default implementation is not provided here, since there is no general way to compute these tensors using the `jacobian()` and the `hessian()` function. Thus, implementing these methods is mandatory when extending from `O2ParametricModule`. The fourth method is used to compute the Loss-Parameter-Hessian for the respective parameter and provides a default implementation that should not be overridden when implementing this abstract class except to provide a more efficient implementation for that specific layer class. Currently supported modules that extend directly from this class are

1. `O2Linear`
2. `O2Conv1d`, `O2TransposeConv1d`
3. `O2Conv2d`, `O2TransposeConv2d`
4. `O2BatchNorm1d`, `O2BatchNorm2d`

4.2.4 O2Container

In order to build complex models, we need composed layers that can be used to build complex models. This functionality is covered by the `O2Container` class, which is an abstract class exposing the API for composed layers. An `O2Container` may consist of other `O2Module` instances, either `O2Layer` or `O2Container`, but it must not include regular PyTorch modules since this would mess with the ability to run a Second Order Backpropagation through the full model. The class exposes the following core methods:

1. ```
   def get_output_input_jacobian(self) -> torch.Tensor
   ```

2. ```
   def get_output_input_jacobian_cached(self) -> torch.Tensor
   ```

3. `def get_loss_input_hessian(self,`
   `dL2dy2: torch.Tensor | SparseSymmetricMatrix`
   `) -> torch.Tensor | SparseSymmetricMatrix`

4. `def get_loss_input_hessian_cached(self`
   `) -> Optional[torch.Tensor | SparseSymmetricMatrix]`

5. `def set_chain_output_input_jacobian(self, value: bool) -> None`

6. `def set_chain_end_output_input_jacobian(self, value: bool) -> None`

7. `def get_chained_output_input_jacobian_cached(self`
   `) -> Optional[torch.Tensor]`

The functionality of the first four methods should be clear from the verbose names and the previous class descriptions. The last three are needed to enable calculating of cOIJs, which in turn are required for parallel layer compositions such as skip blocks. How exactly these methods interact is clarified in section 4.2.7 on module nesting. As far as implementations go, since we have constrained the theoretical analysis of this thesis to sequential architectures and ones with parallel connections, we restrict the implementations to sequential layers and residual blocks, which are reasonably complex and common building blocks in modern Deep Learning architectures. We provide two composite layers

1. `O2Sequential` and
2. `O2Residual`,

where the former is an O2grad-compatible wrapper for `torch.nn.Sequential`, and the latter is a wrapper for a custom class extending `torch.nn.Module` called `Residual`, which simply calculates $x + f(x)$ if composed of an `O2Module` that would otherwise calculate a function f.
An important detail to note is that the API expects no such thing as composite parametric layers, since we believe these can be worked around by implementing suitable subclasses of `O2ParametricLayer` and composing them accordingly. The idea underlying the `O2Container` class is merely to provide a way to combine layers and pass the loss Hessian back to previous layers using the local derivatives calculated in its sublayers.

4.2.5 O2Model: Putting Everything Together

Finally, the `O2Model` class is where everything comes together. An `O2Model` instance is created by passing an instance of (1) `torch.nn.Module` representing a PyTorch model, and (2) a second instance of `torch.nn.Module` representing a loss criterion, for example `torch.nn.MSELoss()`. The submodules of the model must all be `torch.nn.Module` instances supported by `O2Grad` or `O2Module` instances. Upon instantiation, the model uses the method `replace_with_o2modules()` from the `o2grad.recursive` package to replace all submodules with `O2Module` instances, if needed. In addition, it prepares the modules for backpropagation by (1) distributing the model settings as well as the Hessian shared object, (2) setting the `next_layer` attribute in all modules, (3) adding backward hooks and (4) enabling Second Order Backpropagation in the atomic modules. API methods intended for user interaction include:

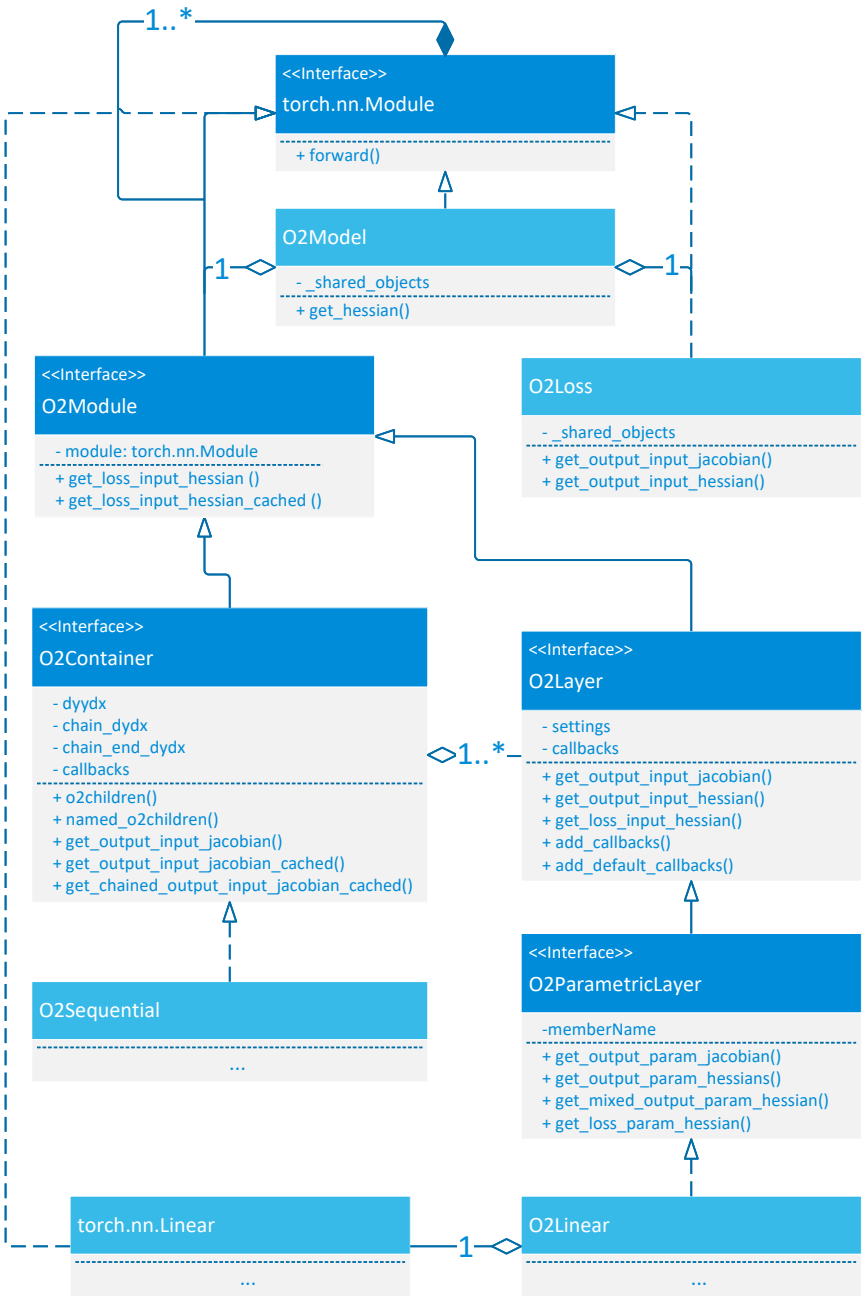


Figure 3: UML class diagram illustrating relationships between the main interfaces/classes, along with some of most important attributes and methods. For instance, an `O2Linear` layer extends from `O2ParametricLayer` and wraps around a `torch.nn.Linear` layer to extend the latter with Second Order Backpropagation functionality.

1. ```
   def distribute_settings(self,
       settings=None,
       condition: Callable[[nn.Module], bool] = lambda m: True
   ) -> None
   ```

   Can be used to distribute a dictionary with settings among all layers satisfying a given condition.

2. ```
   def enable_o2backprop(self) -> None
   ```

   Enables Second Order Backpropagation.

3. ```
   def disable_backprop(self) -> None
   ```

   Disables Second Order Backpropagation. Calling `.backward()` on a loss computed from the model while Second Order Backprop is disabled in this model will not trigger any *O2Grad* methods, but run a simple backpropagation.

4. ```
   def clear_cache(self) -> None
   ```

   Forcibly deletes any temporarily stored derivative tensors calculated in the backpropagation step to free up memory.

Apart from these, the following methods are available to retrieve second order derivative information:

1. ```
   def get_hessian_as_dict(self,
       as_type: str = 'dense',
       diagonal_blocks=False
   ) -> Dict[str, torch.Tensor | SparseSymmetricMatrix]
   ```

2. ```
   def get_hessian_from_input_as_dict(self, input: torch.Tensor,
       target: torch.Tensor,
       as_type: str = 'dense',
       diagonal_blocks=False,
       per_batch=False
   ) -> Dict[str, torch.Tensor | SparseSymmetricMatrix]
   ```

3. ```
   def get_hessian(self,
       as_type: str = 'dense',
       as_file=False,
       diagonal_blocks=False
   ) -> torch.Tensor | SparseSymmetricMatrix
   ```

4. ```
   def get_hessian_from_input(self, input: torch.Tensor,
       target: torch.Tensor,
       as_type: str = 'dense',
       as_file=False,
       diagonal_blocks=False,
       per_batch=False) -> torch.Tensor | SparseSymmetricMatrix
   ```

```
5. def get_hessian_eigs_from_input(self, input: torch.Tensor,
       target: torch.Tensor,
       diagonal_blocks = False
   ) -> Tuple[torch.Tensor, torch.Tensor]
```

Methods with the optional parameter `diagonal_blocks` will return a block diagonal approximation of the Hessian if the parameter is set to `True`. The last method will return the eigenvalues and eigenvectors either of the full Hessian or of the block diagonal approximation, and uses Theorem 2.1 to compute the eigenvalues more efficiently.

4.2.6 Backpropagating Through Simple Models

The starting point for the backpropagation is provided by the `O2Loss` module. This module's OIH is in fact the same as the local LIH, since the output of the module is the loss, and the calculation of the LIH can be done efficiently in the forward pass of this module using the `jacobian()` and `hessian()` functions of the `torch.autograd.functional` package.
For all other modules, the calculation of the local derivatives is done in the backpropagation step rather than in the forward pass as suggested in our original algorithm formulation (see Algorithm 3), since this allows us to generate and keep in memory the respective tensors only as long as they are needed, therefore reducing the maximum memory usage of the whole Hessian calculation from start to end. For atomic layers the backpropagation is the most straight-forward, largely following the pseudocode formulation of the backpropagation discussed earlier. However, other than the pseudocode might suggest, we do not use a `for`-loop to implement the (outer) iteration over the layers, since it is a well-known fact that `for`-loop performance is usually not very good in Python, and PyTorch's autograd engine does already iterate through all layers anyway. Instead, we use the PyTorch `torch.nn.Module` method `register_backward_hooks()` to trigger the Second Order BP step (see Figure 4). This allows us to register a callback that fires after the module's `backward()` method has been called and receives a module reference and a `torch.Tensor` representing the backpropagated loss gradients for that module (with respect to the module's output, i.e. the LOJ). Assuming the next layer's LIH - the current layer's LOH - has already been calculated and stored by the next module, the current layer can retrieve this tensor using the `next_layer` pointer and calling that module's `get_loss_input_hessian_cached()` method. If the layer is configured to chain the OIJ (and is not a chaining end), it will additionally retrieve the next module's cOIJ by calling its `get_chained_output_input_hessian_cached()` method. Then, the module proceeds to compute its LIH and the cOIH, and tries to store the tensors to its cache using the `try_cache()` method, before trying to delete the tensors retrieved from the next module from that module's cache. This is done to immediately free up memory resources used for storing those tensors in an effort to cope with the memory bottleneck of the Second Order BP algorithm.
Naturally, if the module is an instance of `O2ParametricLayer`, its local Hessian and the respective mixed Hessians are also calculated when calling `backprop_step()`.

4.2.7 Backpropagating Through Nested Models

In strictly sequential models without nesting, it is clear which module has to be assigned as `next_layer` to any given layer, because a layer's LOH is the LIH of the next module in the sequence. However, when the model is nested, it might seem like there are modules on several nesting levels that qualify as next module for a layer. As an example, in Figure 5, both *comp2* and *fc2* are

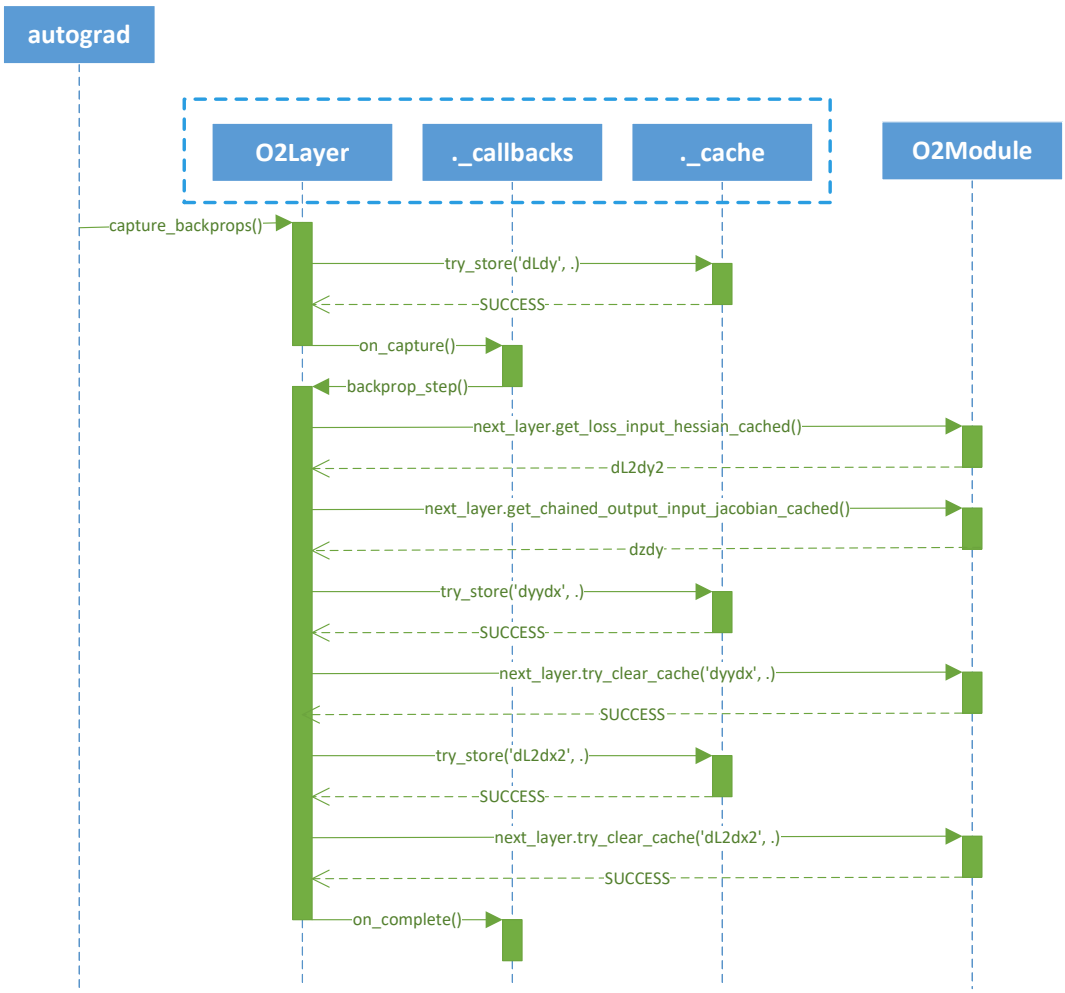


Figure 4: UML sequence diagram illustrating the interactions between an `O2Layer` and the next module during backpropgation. The objects denoted `._callbacks` and `._cache` are attributes of O2Module, but are included with separate lifelines for more clarity. The backpropagation step is triggered by the autograd engine calling the backward hook `capture_backprops()`, and exited by calling the `O2Layer` callback `oncomplete()`. In a purely sequential architecture, the `next_layer` is an instance of `O2Loss` or `O2Layer` or `O2Container`.

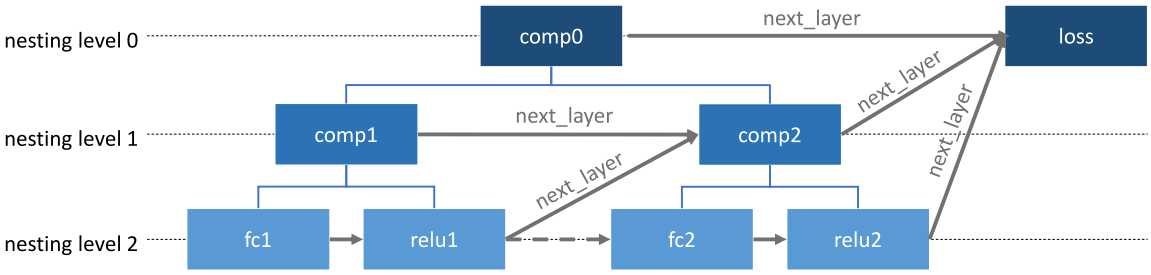


Figure 5: A nested module, with blue lines denoting parent-child relationship (e.g. *fc1* is a child of *comp1*) and grey arrows denoting `next_layer`-relationships, where LOH(*layer*) = LIH(*layer.next_layer*). The dashed arrows point to modules that are direct feedforward successors on the same nesting level, but which do not satisfy said relationship.

modules that come after *relu1* and hence, both might seem like suitable candidates for succession. Indeed, if *comp2* were an `O2Sequential` layer, this would be true, since the input would be passed unmodified to *comp2*, hence not affecting the LIH. In an `O2Residual` however, this would not be the case (see 120). Generally, the input to an `O2Container` always precedes the input to one of its sublayers. Hence, the LOH of any given module must be the LIH of that successive layer with the lowest nesting level. More formally:

Theorem 4.1. *Let l_1, l_2 be successive (atomic) `O2Layers`, and let p be their maximum-depth common ancestor (i.e. there is no node v with $depth(v) >= depth(p)$ s.t. both $l1$ and $l2$ are its children). Furthermore, let p_1, p_2 be direct children of p and let l_1 be on a branch of p_1, l_2 on a branch of p_2. Then, the LOH of l_1 corresponds to the LIH of p_1.*

Assigning the next layer to any module can therefore be done through a simple recursion as featured in Algorithm 4.

Algorithm 4 Recursive function Python code for setting the next layer from which to backpropagate the LIH of a module in nested architecture.

```
def set_next_layer(module, layer) -> None:
        if isinstance(module, O2Layer):
                module.next_layer = layer
        elif isinstance(module, O2Container):
                module.next_layer = layer
                children = [*module.o2children(), layer]
        for i in range(len(children)-1):
                set_next_layer(children[i], children[i+1])
```

However, there are two remaining problems that need to be tackled in order for the Second Order Backpropagation to work: (1) The O2Container layer needs to compute the OIH and cOIJ using the results of the submodules and (2) the execution order of backward hooks during backpropagation is **strictly sequential**, but **not strictly hierarchical**. By *hierarchical execution order*, we mean that backward hooks of a composed module (such as `torch.nn.Sequential`) get executed only after

the backward hooks of all its submodules have been executed. Unfortunately, in PyTorch, this is generally not the case. For instance, consider a composed PyTorch module as in Figure 4.2.7. If the execution of the backward hooks were hierarchical, the order of backward hook execution would be as follows: `model['fc2']` -> `model['fc1']` -> `model`. However, our experiments show that the actual execution order is consistently `model['fc2']` -> `model` -> `model['fc1']`. So while the order of backward hook execution is strictly **sequential for modules on the same nesting level**, the order is not hierarchical across nesting levels. The above toy example is merely designed to bring forth our point, but we have confirmed this kind of behavior also with more complex models, with the composed module backward hooks being executed after the last submodule's backward hooks. This is a problem, since - taken together with point (1) - it means we cannot use backward hooks to run Second Order BP through composed modules, at least not out-of-the-box.

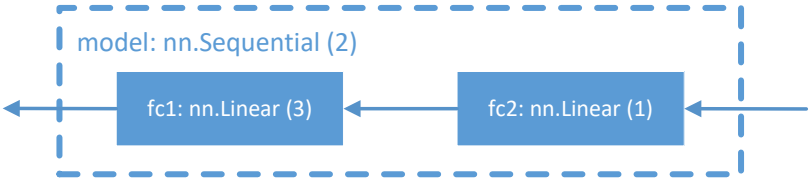


Figure 6: A nested PyTorch model annotated with the module's execution order of backward hooks.

The solution we devised for this problem was to use a callback chaining system to compute `O2Containers`' local derivatives only after those of all their `O2Layer` children have been computed. The sequence diagram of Figure 8 provides a graphical description of how the callback system works. `O2Container` modules can add callbacks to their children which point to an own method or callback. In this way, an `O2Container` can register a function controlled by its own scope with one of its `O2Layer` children. So whenever a `O2Layer`'s backpropagation step completes, it calls the `on_complete()` callback, where the parent `O2Container` has registered its own `on_child_complete()` callback during initialization. The registered callback calls the `children_complete()` function to evaluate whether all children of the `O2Container` module have finished evaluating. If so, the `on_child_complete()` callback triggers the `on_children_complete()` callback, which starts the `backprop_step()` on the `O2Container` module. For example, the `O2Sequential` would register its `on_child_complete()` callback with the first `O2Layer` in its layer sequence and always return `True` in `children_complete()`, because the first module in the sequence being complete implies that the other modules in the sequence have also completed. An `O2Residual` layer simply registers its `on_children_complete()` callback with its only direct `O2Layer` child, and thus also always returns `True` in `children_complete()`. While these two `O2Container` realizations do not require more than one child to finish, a composite layer with n parallel layers as discussed in section 3.8.1 would require all of the n parallel layers to notify their completion, thus need for different callbacks `on_child_complete()` and `on_children_complete()`.

In the backpropagation step, the `O2Container` module goes essentially through the same steps as an `O2Layer` instance, accessing the next module's LIH and cOIJ. However, while an atomic layer requires only the cached input to compute its own OIJ and LIH, an `O2Container` may have to access its children cOIJ and LIH as well. Such is the case with the `O2Residual` layer, which requires

the cOIJ of its only direct child to calculate its OIJ, and both the cOIJ and the LIH of its only direct child to calculate its own LIH (see Section 3.9). The `O2Container` also handles the deletion of those cached tensors in the children from which they are extracted, since by the time they have been used by the parent's backpropagation step, the tensors are no longer required: Previous layers, even those on the same nesting level as the children of the `O2Container`, will require the tensors either from the `O2Container` itself or from a parent of the `O2Container`, thus having `next_layer` point to one of those.

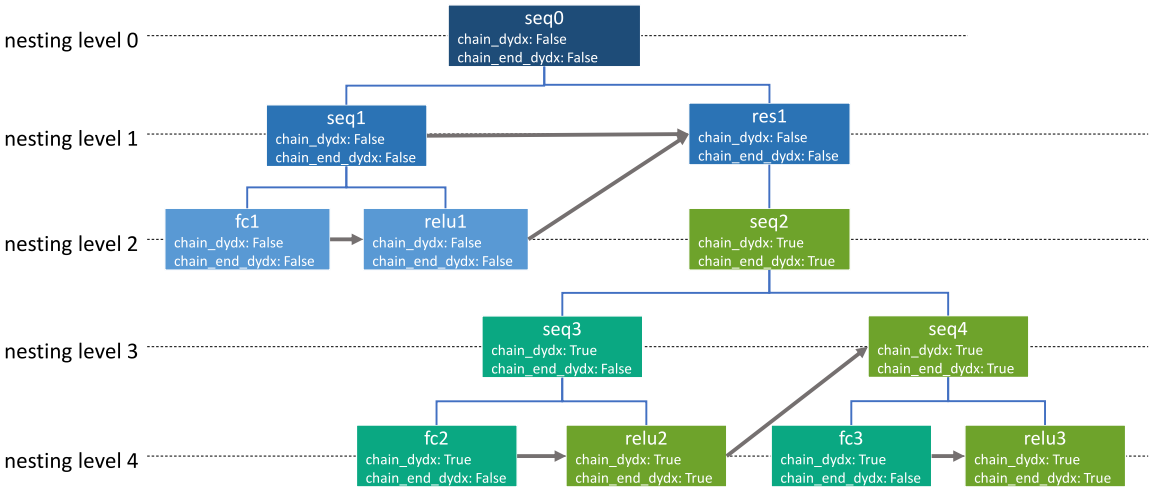


Figure 7: A nested model with OIJ chaining in the `O2Residual` layer *res1*. Layers that chain their OIJ to the next module (where `chain_dydx = True` and `chain_end_dydx = False`) are highlighted in teal, OIJ chaining ends (where `chain_dydx = True` and `chain_end_dydx = True`) are highlighted in lime green.

A detail that we have not explained so far is how the sublayers of a container handle chaining of the OIJs. Consider, for instance, an architecture as in Figure 7. The root-level `O2Sequential` does not need to chain the OIJs of its children, because it is the root-level layer and thus it does not have to backpropagate anything. On nesting level 1, the `O2Sequential` *seq1* does not need to chain its children's OIJs either, because computing its own OIJ (which would be the chained OIJ over all its direct children) is not required anywhere in its backpropagation step. However, the `O2Residual` layer *res1* requires the OIJ of its direct child, *seq2*, in order to calculate its own LIH for the purpose of backpropagating it to *seq1* and its children. Therefore, the `O2Container` must enable chaining in *seq2*, since it can only calculate its own OIJ by chainin the OIJs of its children. But *seq3* and *seq4*, too, can only calculate their own OIJ by chaining their respective children's OIJs. That is, all direct and indirect children of *res1* need to enable chaining. This is what the attribute `chain_dydx`, `chain_end_dydx` and the respective setter methods

1. `def set_chain_output_input_jacobian()` and
2. `def set_chain_end_output_input_jacobian()`

are for. On initialization, the `O2Residual` module will call the first setter with argument `True` on *seq2*. This will set `seq2.chain_dydx` to `True`, and call the setter method on all children of *seq2*, and

so on, therefore recursively setting all children of *res1* to chaining mode. Additionally, the last child (in feedforward order) of any `O2Container` below *res1* is marked as chaining end: The layers need to know if they are an intermediate or an end point of the OIJ chaining, since this will determine the iteration step. That is, *res1* needs to tell *seq2* it's a chaining end point, which in turn needs to tell *seq4*, which finally tells *relu3*, and *seq3* tells *relu2*. Note that it would not suffice to set the last child on every nesting level below `res1` as a chaining end point (in the example only *relu3* on nesting level 4), since in that case, *relu2* would chain its OIJ to the cOIJ coming from *seq4*. But we don't want this to happen, because it would interfere with the OIJ calculation of the parent, *seq3*. Hence, each composed layer needs to set its last layer in feedforward order to be a chaining end.

4.3 Exploiting Sparsity

4.3.1 General Considerations

The Second Order Backpropagation algorithm should, in general, be expected to suffer from a memory bottleneck, since naively, the size of the local derivatives scales quadratically (for the Jacobians as $\mathcal{O}(mn)$, where m is the size of the 'dividend' and n the size of the 'divisor' tensor in the derivative) and even cubically (for the Hessians as $\mathcal{O}(mno)$ if n, o are the sizes of the two 'divisor' variables' tensors) with the total size of the tensors involved. At least, this is the case when the local derivatives are represented as *dense* tensors (i.e. a dense tensor of Dth order is represented as a D-dimensional cube of numbers). This kind of representation would quickly render the mere storage of the local derivatives unpractical even for relatively small neural networks. For instance, consider a fully-connected layer with an input dimension of 1000 and and output dimension of 1000. Furthermore, let us assume we feed the layer a minibatch with 64 samples, such that the input and the output tensor of the layer have dimensions 64×1000. In that case, the OIJ of the linear layer would be a tensor with $64^2 \cdot 1000^2 = 4.096 \cdot 10^9$ entries, which would correspond to at least 16.384GB of memory assuming 32bit float representation. Taking into account the other tensors that need to be stored for the duration of the Second Order Backpropagation, this is a task that is currently only feasible for computers with lots of RAM, or with a very fast access to swap memory, allowing the relevant elements of the tensors to be loaded into memory on demand for tensor operations. However, the tensors of all the local derivatives discussed in this thesis have a high **sparsity** - meaning that they have a high proportion of entries that are 0:

$$sparsity(T) = 1 - \frac{\#nonzero\_elements(T)}{\#elements(T)} \tag{121}$$

One of the less sparse examples is the OIJ of the linear layer, where every output value is connected to every input value, but only for the same sample, since linear layers do not mix elements across the batch dimension! Thus, given input dimension M, output dimension N and batch size B, the number of nonzero elements in the OIJ is no greater than $B \times M \times N$, which corresponds to a sparsity of $\frac{B-1}{B}$. So asymptotically, as the batch size goes to infinity, the sparsity goes to 1 and the proportion of nonzero values vanishes. Needless to say, it is a waste of memory to store every single value of a tensor where most of the values are zero anyway, and it would be more efficient to store the non-zero values only, along with some information regarding their position in the tensor.

A possible way of realizing this is using a sparse COO (Coordinate list) representation, in which for each non-zero value, the value and the multi-index giving the position of the element in the

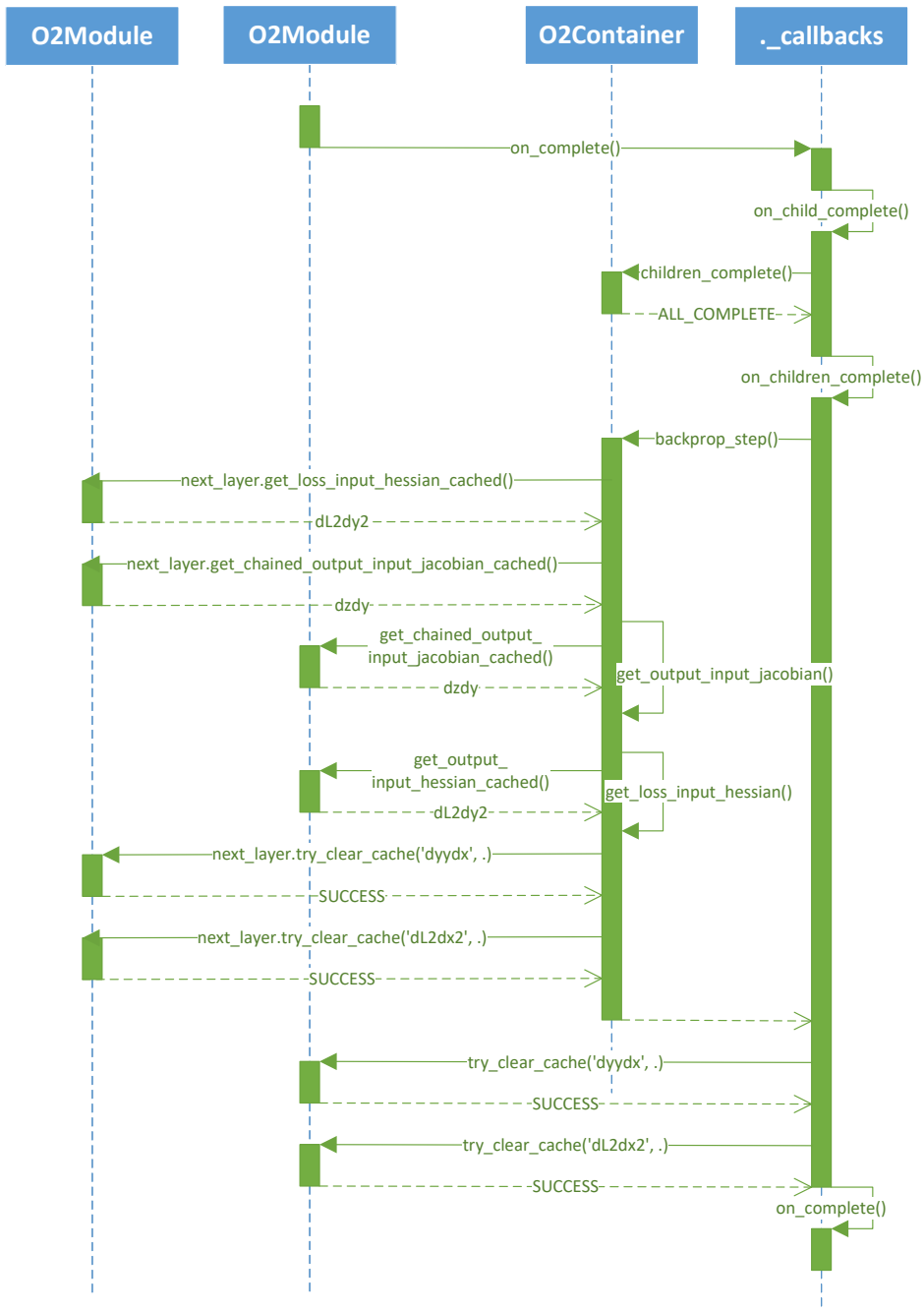


Figure 8: UML sequence diagram illustrating the interactions between an `O2Container` module, the submodule from which its backprop step is triggered and the next module, from right to left. Caching is not included.

Layer	OIJ	OIH	OPJ	OPH	mOPH
linear	BMN	1	BMN	1	BMN
conv1d	$\frac{BC_{in}C_{out}(M-K+2P)K}{S}$	1	$BC_{in}C_{out}MN$	1	$\frac{BC_{in}C_{out}(M-K+2P)}{S}$
bn1d	CB^2N^2	CB^3N^3	BNC	1	CB^2N^2
relu	BN	1	/	/	/
sigmoid	BN	BN	/	/	/

Table 1: $\mathcal{O}(.)$ upper bounds of memory complexity of the local derivate tensors for selected layers assuming a sparse COO representation. The expressions are derived making use of the Kronecker deltas and indicator functions appearing in the expressions for the respective local derivative tensors, derived in the previous chapter.

tensor is stored. For instance:

$$A = \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} \quad \Rightarrow \quad \text{indices: } \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \quad \text{values: } \begin{pmatrix} 1 & 2 \end{pmatrix} \tag{122}$$

So given n values, a tensor of order D can be represented using at most $(D+1)N$ elements. By compressing the tensor to a vector, the tensor can even be represented using only $2N$ elements (but implementing some tensor operations such as the tensor dot product becomes more difficult this way). Returning to the previous example, the OIJ of the examplary linear layer is a 2D tensor (a matrix), and could thus be represented using $3BMN = 192 \cdot 10^6$ entries, which is equivalent to 1.28GB of memory, assuming the indices are represented as 64bit integers - a clear improvement by one order of magnitude. Table 1 contains an overview of the memory complexity for the local derivatives of selected atomic layers discussed.

Conveniently, COO sparse representation is also supported by PyTorch through the package `torch.sparse` [9], which provides support for sparse COO tensor representation and operations (with GPU support). Unfortunately, at the time of writing this thesis PyTorch does not support matrix multiplications between sparse matrices out of the box, but this functionality is provided by the open source *pytorch_sparse* [10] package, more specifically through the `spspmm()` function. In O2Grad, we rely on `torch.sparse` and the *pytorch_sparse* package to provide the required functionality.

4.3.2 SparseSymmetricMatrix

The memory efficiency can be improved even further by making use of the symmetry property of the Hessian, i.e. $\frac{\partial^2 L}{\partial x_i \partial x_j} = \frac{\partial^2 L}{\partial x_j \partial x_i}$. A matrix $\mathbf{H} \in \mathbb{R}^{n \times n}$ is symmetric iff it can be decomposed as a sum

$$\mathbf{H} = \mathbf{L} + \mathbf{D} + \mathbf{L}^T, \quad \mathbf{L}, \mathbf{D} \in \mathbb{R}^{n \times n}, L_{ij} = H_{ij} \forall j < i, D_{ii} = H_{ii} \tag{123}$$

where $\mathbf{L}$ is a lower triangular matrix with its entries corresponding to the subdiagonal values of $\mathbf{H}$ and $\mathbf{D}$ is a matrix containing only the diagonal values of $\mathbf{H}$. From this representation, it follows that

[9] `https://pytorch.org/docs/1.9.0/sparse.html`
[10] `https://github.com/rusty1s/pytorch_sparse`

the memory usage of Hessian matrices can be cut approximately in half by storing tensors $\mathbf{L}$ and $\mathbf{D}$ instead. The tensor operations relevant for our application (sum and dot product) can be realized without ever having to store $\mathbf{L}$ and $\mathbf{L}^T$ in memory at the same time. For $\mathbf{S}$ a sparse symmetric matrix and $\mathbf{T}$ a non-sparse matrix, the sum can be calculated as $\mathbf{S} + \mathbf{T} = (\mathbf{L} + \mathbf{D} + \mathbf{T}) + \mathbf{L}^T$, where transposition of $\mathbf{L}$ is done in-place (freeing up the memory reserved for $\mathbf{L}$) after computing the expression in the brackets. The original tensor $\mathbf{L}$ can be restored by transposing $\mathbf{L}^T$ again, which will slightly increase the computation cost as payoff for the reduced memory usage. If both $\mathbf{S}$ and $\mathbf{T}$ are sparse symmetric matrices with $\mathbf{T} = \mathbf{R} + \mathbf{E} + \mathbf{R}^T$, the sum can be written as

$$\mathbf{S} + \mathbf{T} = (\mathbf{L} + \mathbf{R}) + (\mathbf{D} + \mathbf{E}) + (\mathbf{L} + \mathbf{R})^T, \tag{124}$$

which is again a sparse symmetric matrix, so it suffices to compute the sums $\mathbf{L} + \mathbf{R}$ and $\mathbf{D} + \mathbf{E}$ and store these as subdiagonal and diagonal of the resulting sparse symmetric matrix. For matrix dot products we generally have $(\mathbf{L} + \mathbf{D} + \mathbf{L}^T) \cdot \mathbf{T} = (\mathbf{L} \cdot \mathbf{T} + \mathbf{D} \cdot \mathbf{T}) + \mathbf{L}^T \cdot \mathbf{T}$. Again, holding both $\mathbf{L}$ and $\mathbf{L}^T$ in storage can be avoided by first evaluating the expression inside the brackets and then transposing $\mathbf{L}$ to evaluate the rest. If $\mathbf{T}$ is also a sparse matrix, the expression stemming from the above representation is more complex:

$$\begin{aligned} \mathbf{S}^T \cdot \mathbf{T}^T = &\mathbf{L} \cdot \mathbf{R} + \mathbf{L} \cdot \mathbf{E} + \mathbf{L} \cdot \mathbf{R}^T \\ &+ \mathbf{D} \cdot \mathbf{R} + \mathbf{D} \cdot \mathbf{E} + \mathbf{D} \cdot \mathbf{R}^T \\ &+ \mathbf{L}^T \cdot \mathbf{R} + \mathbf{L}^T \cdot \mathbf{E} + \mathbf{L}^T \cdot \mathbf{R}^T \end{aligned} \tag{125}$$

Evaluating the above expression without increasing the memory usage will require transposing $\mathbf{L}$ and $\mathbf{R}$ inplace at least 1 or 5 times, respectively. Note that the resulting tensor may be sparse, but will generally not be a symmetric tensor. However, for what we call a *twin dot product* $\mathbf{T}^T \cdot \mathbf{S} \cdot \mathbf{T}$ between a matrix $\mathbf{S}$ and a matrix $\mathbf{T}$, the result is always symmetric if $\mathbf{S}$ is symmetric. This follows simply from

$$(\mathbf{T} \cdot \mathbf{S} \cdot \mathbf{T}^T)_{il} = \sum_{j,k} T_{ij} S_{jk} T^{\mathrm{T}}_{kl} = \sum_{j,k} = T_{lk} S_{kj} T^T_{ji} = (\mathbf{T} \cdot \mathbf{S} \cdot \mathbf{T}^T)_{li} \tag{126}$$

Since there is no class in PyTorch already providing the required functionality, we implement an own class `SparseSymmetricMatrix` that supports sums and dot products with instances of `torch.Tensor` (both strided and sparse) and other instances of the class. In addition, we implement a function `linalg.twin_matmul_mixed(S, T)` which returns an instance of `SparseSymmetricMatrix` when matrix $\mathbf{S}$ is also an instance of the class.

In the next section, we investigate empirically how the implementation of the Second Order BP algorithm fares in terms of time and memory usage, and try to pinpoint whether the algorithm for computing the Hessian using Second Order Backpropagation is competitive with the best PyTorch-based algorithm.

5 Measurements

For the measurements analyzed in this section, we use *snuffles*, a server kindly provided by Prof. Dr. Tim Landgraf's lab. The system features an AMD Ryzen Threadripper 1950X processor, 4x Nvidia

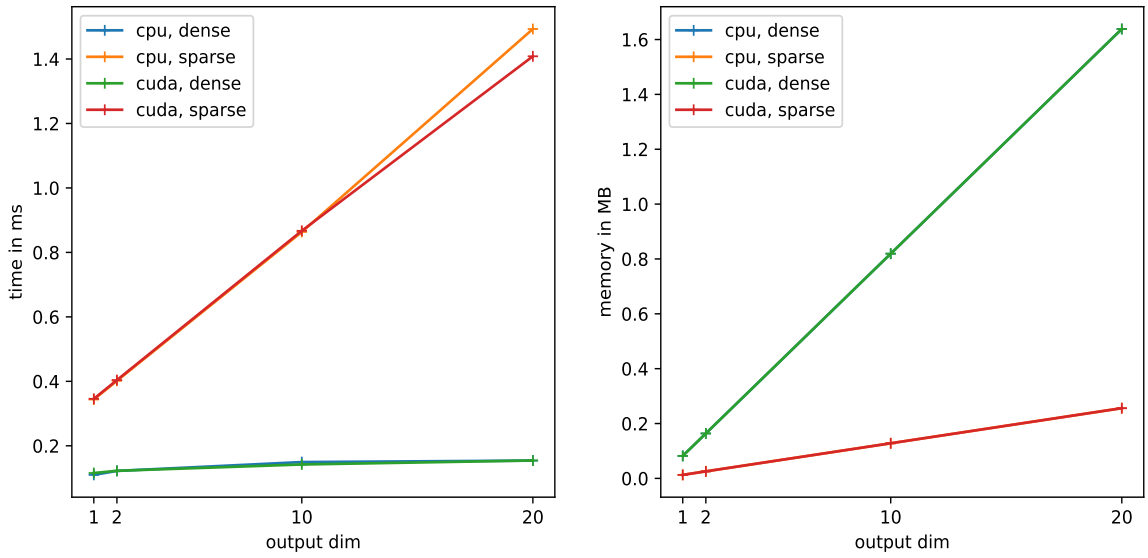
GeForce RTX 2080 TI GPUs (Driver version 460.91.03) with 11GB VRAM, and 64GB RAM. The system runs on Debian 5.10.46-4 (2021-08-03) x86_64. All GPU measurements are run on a single GPU node, since our *O2Grad* package does currently not support multi-GPU setups. To set up the Python environment, we use an installation of miniconda (4.10.3) and create an environment with PyTorch 1.9.0 and CUDA version 11.2 in the accompanying cudatoolkit. The root folder of the project repository for our thesis contains an 'environments.yml' file for creating a conda environment clone and reproducing the measurements discussed in this section. The time measurements are performed using the `timeit`[11] package, while the memory measurements are performed using the `memory_usage()` function of the `memory_profiler`[12] package. While PyTorch does have an own profiling package `torch.profiler`, we deliberately use the above packages instead, since the memory profiler turned out to be very slow, such that using it would have slowed down our measurements considerably. Unfortunately, this means that the reliability of the memory usage measurements might not be as high as they would be using the profiler provided by PyTorch. Furthermore, since the `memory_profiler` module only tracks CPU memory usage, the memory measurements performed on CUDA devices will only reflect memory usage on the CPU, but not on the CUDA devices. We perform the measurements nonetheless, since they can be used to infer information about the memory usage of the algorithm in combination with the measurements on CPU.

5.1 Tensor Generation

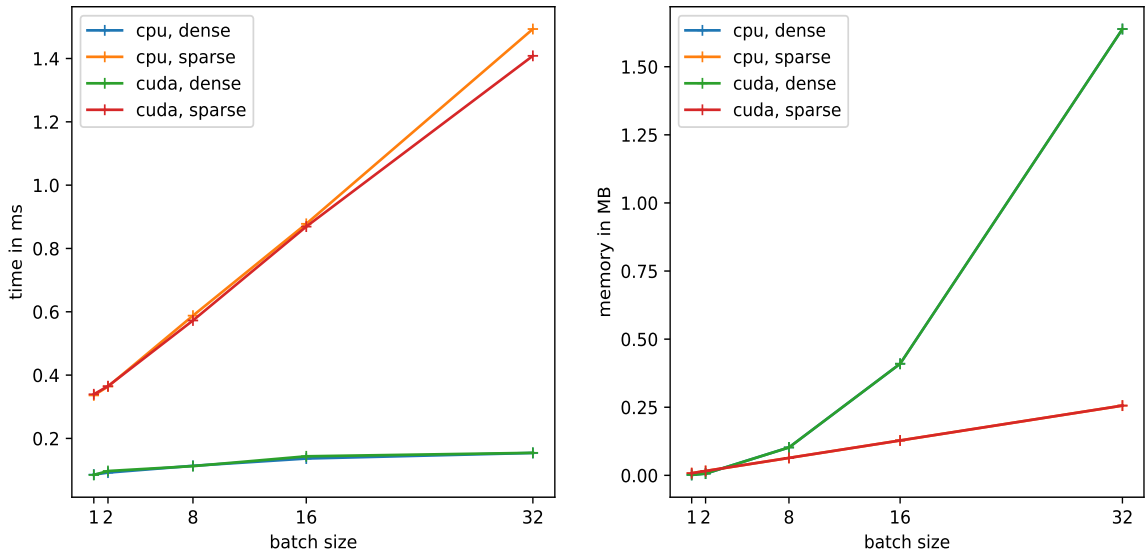
An important question to be addressed is whether the tensors should be generated in a dense or sparse layout. While our theoretical analysis showed that the memory complexity of the sparse representations should be expected to scale favorably compared to a dense representation, we want to confirm this finding experimentally. To do so, we prepare a grid of batch sizes, input sizes and output sizes for a single instance of `O2Linear` and compute the layer's local derivatives using the respective API functions, while measuring the time and estimating the memory usage from the number of elements in the tensor. As Figure 9 demonstrates, there is a time-memory tradeoff as a function of the input and layer dimensions for generating the OIJ of a linear layer with a sparse versus a dense layout, although the generation times for both layouts remain roughly in the same order of magnitude. For the dimensions investigated here, the memory sizes of the OIJ are in an unproblematic domain. For the OIH, on the other hand, both the time for generating the tensor and the memory required in dense layout increase so quickly that using a sparse layout is the only scalable option for working with larger layers, as seen in Figure 10, since the OIH is zero everywhere. The OPJ, OPH and mOPH offer a similar picture (see Table 20, 21 and 22 in Appendix C). Tables 5, 7, 9, and 10 with the measurement values can be found in Appendix B.

To see how the return layouts of the local derivative functions impact the memory usage and computation time when used in the actual Second Order Backpropagation algorithm, we compute the Hessian on a 100-20-10 MLP with sigmoid as activation functions and inputs of batch size 16, for all possible configurations of return layouts. As we can see in Table 2, the highest memory usage of 10.5GB is unsurprisingly obtained when exclusively using strided tensors. While the time required to compute the Hessian is also high at approximately 4.3s in this case, the worst configuration in terms of runtime is obtained when only the OIJ and the OPJ are returned as sparse tensors. This makes sense because the OIJ and the OPJ benefit the least from having a sparse representation, and indeed the memory usage even seems to increase for the dimensions chosen in this particular

[11] `https://docs.python.org/3/library/timeit.html`
[12] `https://github.com/pythonprofilers/memory_profiler`



(a) (b)



(c) (d)

Figure 9: Time-Memory tradeoff of generating the dense/sparse OIJ tensors of a `O2Linear` layer.

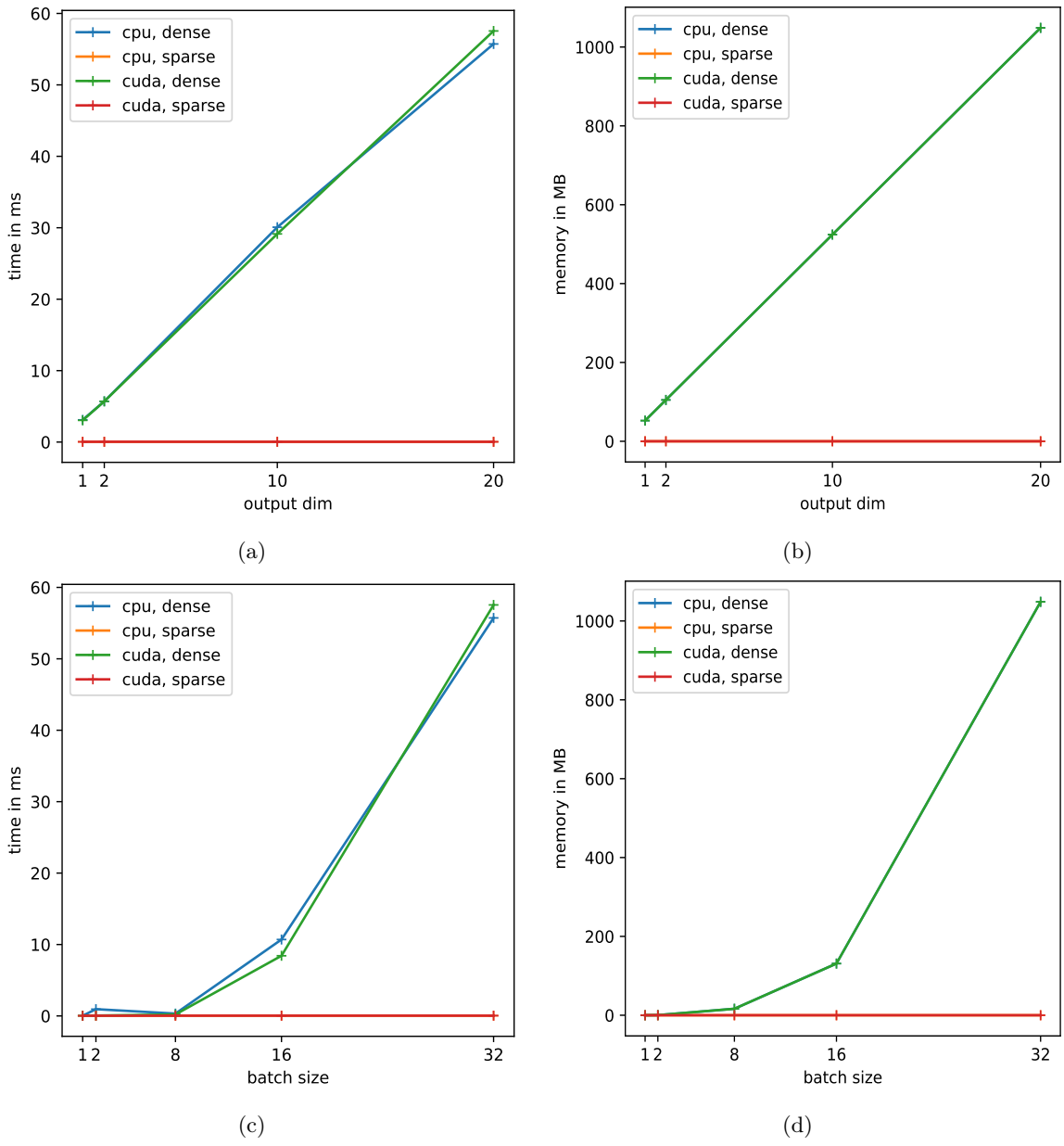


Figure 10: Time and memory usage of generating the dense/sparse OIH tensors of a `O2Linear` layer. In the sparse case, these are close to zero, since the OIH is the zero matrix.

example. At same time, the sparse representation incurs a computational overhead, which is only worth paying when the sparse representation strongly reduces the number of elements stored. This also explains why a similar configuration, with the OIH also sparse, leads to the worst runtime of 4.5s. Meanwhile, the lowest memory usage is obtained when all tensors except for the OPJ are sparse, at a mere 1.1GB, which is an improvement by one order of magnitude, and it is much faster, too, taking only 0.19s to compute - an improvement by a factor of 21×! By returning both the OPJ and the OIH in strided layout, the computation time can be reduced even further to a mere 0.09s, which is an improvement by a factor of 45×, while only using slightly more memory (1.173GB vs 1.184GB). We therefore use the latter configuration as the default for the return layouts in `O2Linear` layers.

While the above results already show impressive performance improvements, we can also see that despite all our optimizations, it takes a considerable amount of almost 1.2GB of memory to run the Second Order Backpropagation algorithm on this rather small model, already hinting at the fact that we can probably only ever hope to use this algorithm as a scientific tool on toy models.

5.2 MLP

To see just how far we can take the algorithm, we investigate how the computation time and memory usage scale when variating (a) input size and (b) batch size. To do so, we prepare a N - N - N MLP, where N is the input size: Since we are interested in seeing how the algorithm performance scales with network size, the intermediate and output layer size are always set to equal the input layer size and are not varied as well. For each tuple `(input_size, batch_size)`, we run the experiment for all 4 possible configurations: {`'autograd'`, `'o2grad'`} × {`'cpu'`, `'gpu'`}. Furthermore, given any tuple `(input_size, batch_size)` and for all 4 possible configurations, we calculate the maximum absolute deviation of the Hessian computed w.r.t. the baseline configuration `('autograd', 'gpu')` - the error for the baseline configuration is zero - to (1) check that the Hessians computed are actually correct and (2) see if there is a dependence of the quality of the Hessian computed on network size. Table 12 contains all results (they are also available in the `results` folder in the official repository).

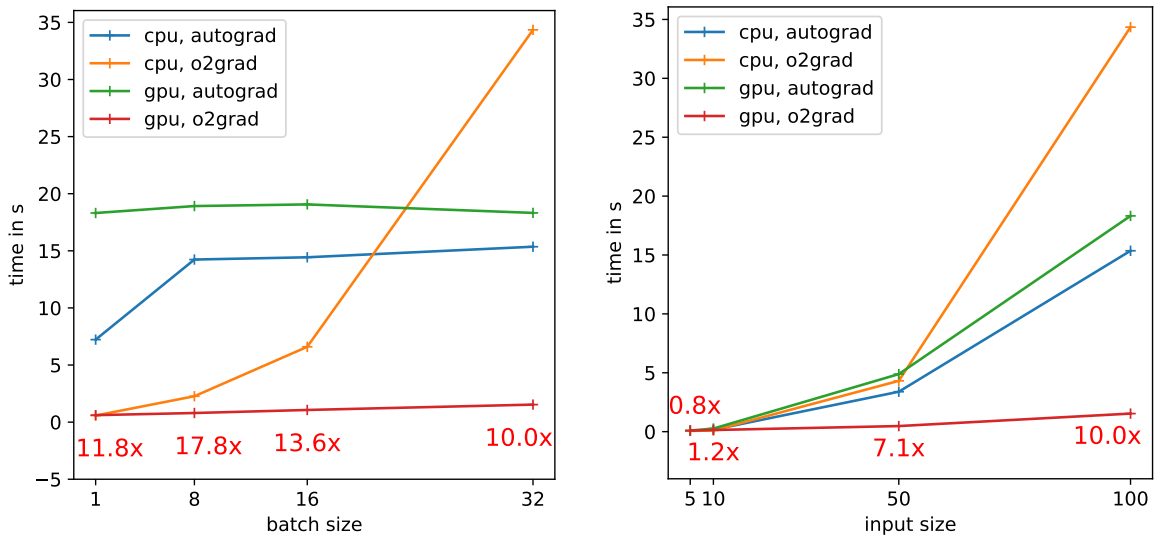
Batch Size

As we can see in Figure 11a, for an input size of 100, *O2Grad* on GPU is consistently faster than the autograd methods and the absolute time remains small within the inspected range. For the maximum batch and input size of 32 and 100, respectively, *O2Grad* takes 1.5s, while the autograd methods take 15.4s and 18.3s on CPU and GPU, respectively. That's not an improvement as big as for the 100-20-10 network, but still a significant one if one wants to compute the Hessian of a network frequently during the training, since calculating the Hessian would by far be the most expensive step. Interestingly, the computation time for *O2Grad* on CPU grows far more quickly than on GPU, so the initial advantage over the autograd methods at batch size 1 quickly melts away and the autograd methods likely surpass *O2Grad* somewhere between a batch size of 20 and 25. This may be because the architecture of the GPU is better suited to deal with operations on big tensors through vectorization.

The results for the memory usage are very unintuitive, since the memory usage appears to decrease with increasing batch size for the *O2Grad* methods (see Figure 12a). While the parameter Hessian's size itself is indeed independent from the batch size, the intermediate tensors calculated in the algorithm are not, and therefore we would expect the memory usage to increase along with the batch size. The behaviour shown might make sense for *O2Grad* on GPU, since the intermediate

dydx	dy2dx2	dydp	dy2dp2	dy2dxdp	time in s	mem in MB
sparse	sparse	sparse	sparse	sparse	2.191	1389
sparse	sparse	sparse	sparse	strided	4.485	5684
sparse	sparse	sparse	strided	sparse	2.268	5314
sparse	sparse	sparse	strided	strided	**4.564**	9711
sparse	sparse	strided	sparse	sparse	0.191	**1173**
sparse	sparse	strided	sparse	strided	2.510	6195
sparse	sparse	strided	strided	sparse	1.195	4571
sparse	sparse	strided	strided	strided	3.638	9908
sparse	strided	sparse	sparse	sparse	0.957	2739
sparse	strided	sparse	sparse	strided	3.327	6409
sparse	strided	sparse	strided	sparse	2.005	5931
sparse	strided	sparse	strided	strided	4.333	**10471**
sparse	strided	strided	sparse	sparse	0.811	2807
sparse	strided	strided	sparse	strided	3.274	6575
sparse	strided	strided	strided	sparse	1.834	5893
sparse	strided	strided	strided	strided	4.173	10427
strided	sparse	sparse	sparse	sparse	0.199	1183
strided	sparse	sparse	sparse	strided	2.554	6247
strided	sparse	sparse	strided	sparse	1.224	4570
strided	sparse	sparse	strided	strided	3.477	10196
strided	sparse	strided	sparse	sparse	**0.092**	1184
strided	sparse	strided	sparse	strided	2.364	6234
strided	sparse	strided	strided	sparse	1.146	4489
strided	sparse	strided	strided	strided	3.446	10001
strided	strided	sparse	sparse	sparse	0.851	2695
strided	strided	sparse	sparse	strided	3.249	6328
strided	strided	sparse	strided	sparse	1.908	5843
strided	strided	sparse	strided	strided	4.285	10170
strided	strided	strided	sparse	sparse	0.755	2774
strided	strided	strided	sparse	strided	3.165	6613
strided	strided	strided	strided	sparse	1.862	5544
strided	strided	strided	strided	strided	4.027	10504

Table 2: Time and memory usage for calculating the Hessian of a 100-20-10 MLP with sigmoid as activation function on CPU, given all possible configurations of the return layout of the different local derivative methods. Highest time and memory usage are highlighted in **bold** red, lowest time and memory usage are highlighted in **bold** green.



(a) `input_dim` = 100

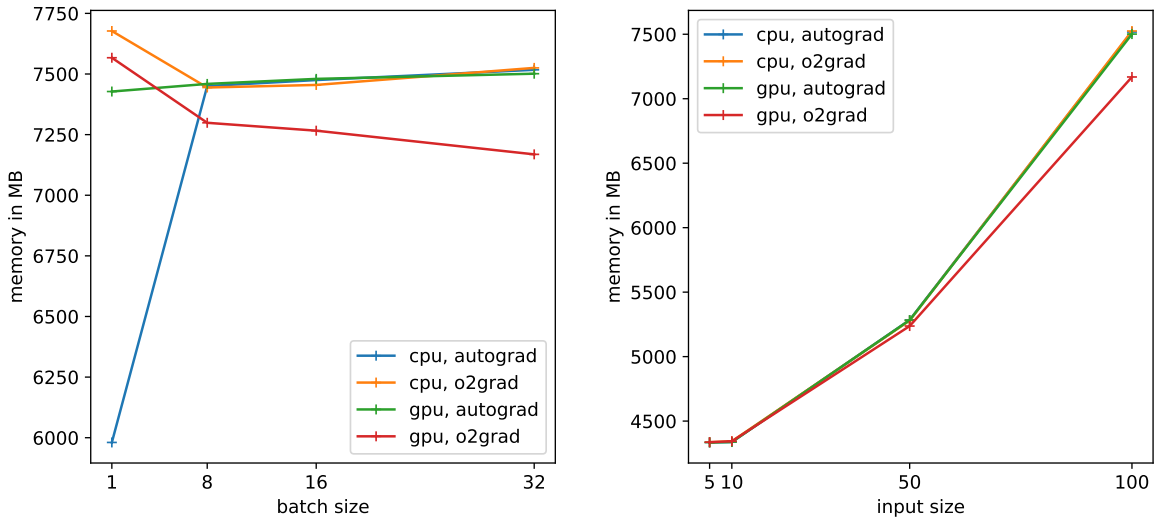
(b) `batch_size` = 32

Figure 11: Time required to compute the Hessian of an N- N - N MLP, as a function of (a) batch size and (b) input size N, for fix `input_dim` and `batch_size`, respectively. Annotated with minimum speedups of *O2Grad* on GPU over the autograd methods.

tensors will all be generated and used for computation on the GPU, which is not tracked by `memory_profiler`, but this does not explain why the memory usage also decreases for *O2Grad* on CPU. Further investigation is required to identify the cause. The errors of the *O2Grad* methods do not appear to show any dependency on the batch size and are negligible at an order of magnitude of 10^{-8} (see Figure 13a).

Input Size

As we would expect, the input dimension, which for this configuration scales the number of parameters in the network quadratically. Thus, we should expect time and memory usage to increase quadratically with the input dimension. Consider the computation time results for a fixed batch size of 32 (see Figure 11b). *O2Grad* on CPU scales least favourably in the inspected range, increasing in an accelerated fashion and overtaking the autograd methods somewhere around an input size of 50. The increase in computation time of *O2Grad* on GPU also accelerates, but at a smaller rate. Naturally, where the impact of the increasing input sizes shows most is in the memory usage (see Figure 12b). The memory usage increases for all methods at approximately the same rate, since the biggest consumer of CPU memory is the Hessian being generated at the end of the calculation and all earlier computation steps stay below the total memory required at the end of the computation. The maximum error seems to be approximately constant at a magnitude of 10^{-9} (see Figure 13b).



(a) `input_dim` = 100

(b) `batch_size` = 32

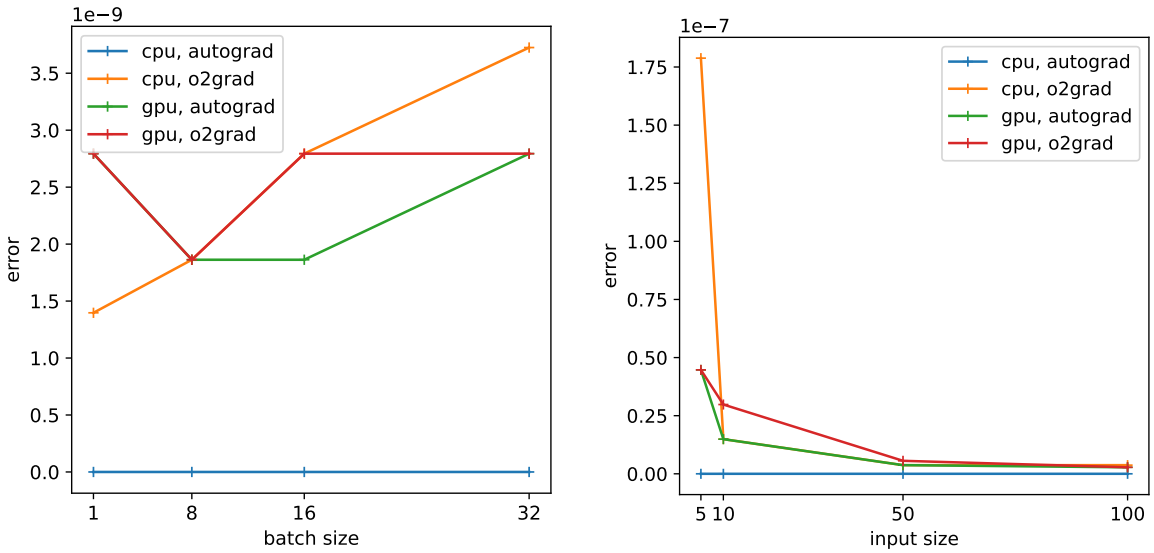
Figure 12: Memory used to compute the Hessian of an N- N - N MLP, as a function of (a) batch size and (b) input size N, for fix `input_dim` and `batch_size`, respectively.

MNIST

Finally, we want to prepare a model that is suitable for training on the MNIST handwritten digit dataset [13]. The dataset contains 28px × 28px greyscale images of digits with their respective labels. While MNIST is a simple problem by today's standards, it is suitable for training and analyzing toy models because it is a low-dimensional problem with natural, easy-to-visualize data. For this experiment series, we set a fix input dimension of 784, fix an output dimension of 10 and variate only the intermediate dimension and the batch size. All measurements are featured in Table 13.

The measurement results give us an idea of the algorithm's limitations. At a batch size of 32 and intermediate dimension of 20, *O2Grad* on GPU is no longer capable of calculating the Hessian. This also applies to an intermediate dimension of 30, irrespective of the batch size. *O2Grad* on CPU is still faster than the autograd methods in this range, but takes 8.1s for an intermediate dimension of 30 and a batch size of 32, which is a speedup of just 2.1× compared to the autograd method on CPU, while also requiring an extra 3.5GB in memory. However, we confirmed experimentally that an MLP can achieve a reasonable training performance for MNIST with just 20 intermediate neurons and for a batch size of 16, and *O2Grad* on GPU takes just 0.5s for calculating the Hessian of an MLP given these hyperparameters, while autograd on CPU takes a full 9.9s seconds. Of course, these findings further reinforce the observation that *O2Grad* is nowhere close to being useful for practical applications: Currently, the best non-convolutional ANN architecture is a 6-layer ANN with a 784-2500-2000-1500-1000-500-10 architecture according to the MNIST homepage, and the number of parameters in this network would be impossible for *O2Grad* to tackle. But optimal performance is not necessarily relevant in research concerned with understanding the inner workings

[13] `http://yann.lecun.com/exdb/mnist/`



(a) `input_dim` = 100 (b) `batch_size` = 32

Figure 13: Maximum absolute errors of computing the Hessian of an N- N - N MLP, as a function of (a) batch size and (b) input size N, for fix `input_dim` and `batch_size`, respectively. The reference Hessian is the one calculated with the autograd method on CPU.

of ANNs, while speed is often a concern.

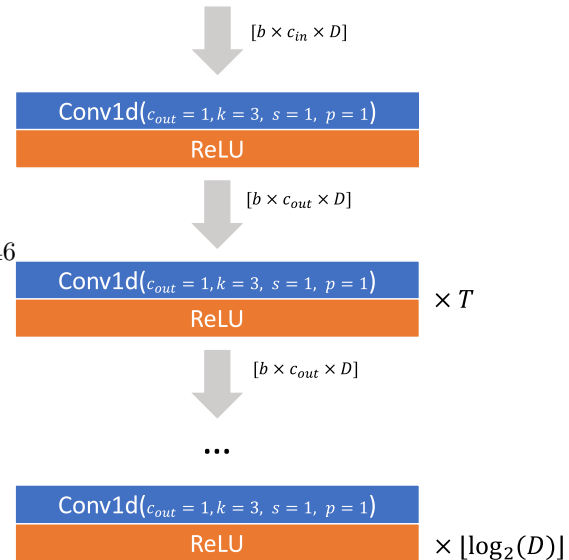
5.3 1D CNN

So far, we have seen that *O2Grad* leads to a considerable reduction of Hessian calculation time for small MLPs, while also incurring a high memory cost. However, one might hope for the results for 1D convolutional architectures to be better, since we have shown in section 3.6.2 that the local derivatives of 1D convolutions are very sparse. To test this idea, we set up a 1D convolutional architecture that is determined by 4 hyperparameters: **input dimension**, **output channels**, **block count** and **batch size**. The architecture we use for testing is shown in Figure 14. Note that we don't include Batch Normalization layers even though they are a common layer type used in CNN architectures, because the local derivatives are less sparse than those of other layer types and thus restrict the dimensions of the networks we can efficiently calculate with *O2Grad*. Also, note that all local derivate tensors are generated as sparse tensors, since we noticed that the algorithm runs into CUDA OOM exceptions very quickly otherwise, even for low-dimensional layers.

The full results are available in Table 14.

Input Size

For the 1D CNN , we expected the computation time as a function of the input dimension



to scale differently due to the sparse connectivity of the convolution layers. In addition, the greater the input size, the more downsampling layers our CNN architecture will have. However, this expectation was mostly not met (see Figure 15a). For the parameters considered, the computation time with autograd on GPU remains constant or exhibits small growth, as was the case with the input size for the MLP, while the computation time on the CPU seems to drop between an input dimension 16 and 32, before slowly (linearly?) increasing thereafter. This time drop is rather counterintuitive and we are not able to provide any fundamental explanation for it. Therefore, we have to assume this has something to do with internal details of PyTorch's implementation. For *O2Grad*, the computation time matches the behaviour of the input size in the MLP case: On the GPU, the time increases very slowly (linearly) in the measurement range, while on the CPU it increases quadratically. More importantly however, we notice that *O2Grad* on GPU provides once again an enormous boost in computation time over the autograd methods, with an impressive improvement by a factor of $62\times$ for the smallest input size of 16, and a more moderate improvement by a factor of $11\times$ for the largest input size of 100. Interestingly, memory usage seems to slightly decrease for all methods as input dimension increases (see Figure 16a), which is a theoretically unexpected result and must therefore be related to some internal of PyTorch's backpropagation, or possibly to Python's garbage collection. The error appears to be stable in an order of 10^{-8} (see Figure 23a).

Output Channels

Since neuron connectivity over the channel dimension of convolution layers is exactly as for a dense linear layer, we expect the computation time as a function of channel count to scale exactly as for the input size in the MLP, i.e. quadratically. Our measurements approximately match the expectation (see Figure 15b). For *O2Grad* on GPU, computation times for the number of output channels increases seemingly linearly as a function of channels in the inspected range. However, the memory usage is difficult to explain, peaking at 10 channels, but dropping again at 15 channels (especially for autograd on CPU, with a drop of memory usage by over 50%) (see Figure 16b). Since this happens regardless of the method used, we suspect this must be related to some internal implementation detail of autograd's backpropagation for this kind of architecture, with the memory usage improving

for a certain number of output channels in the `O2Conv1d` layer. Another possible explanation is that this is in some way related to Python's garbage collector. Interestingly, a similar peak can be observed in the error at a block count of 5 (see Figure 23b) and for all of the methods compared to the reference method, so this must be an intrinsic numerical error coming from some operation in the default autograd backpropagation. In any case, since the errors stay within a perfectly acceptable order of magnitude of 10^{-8} to 10^{-7}, but it would be interesting to understand the origin of the drop in memory for the output channels, since it could pave a way to an improvement in memory efficiency of all methods.

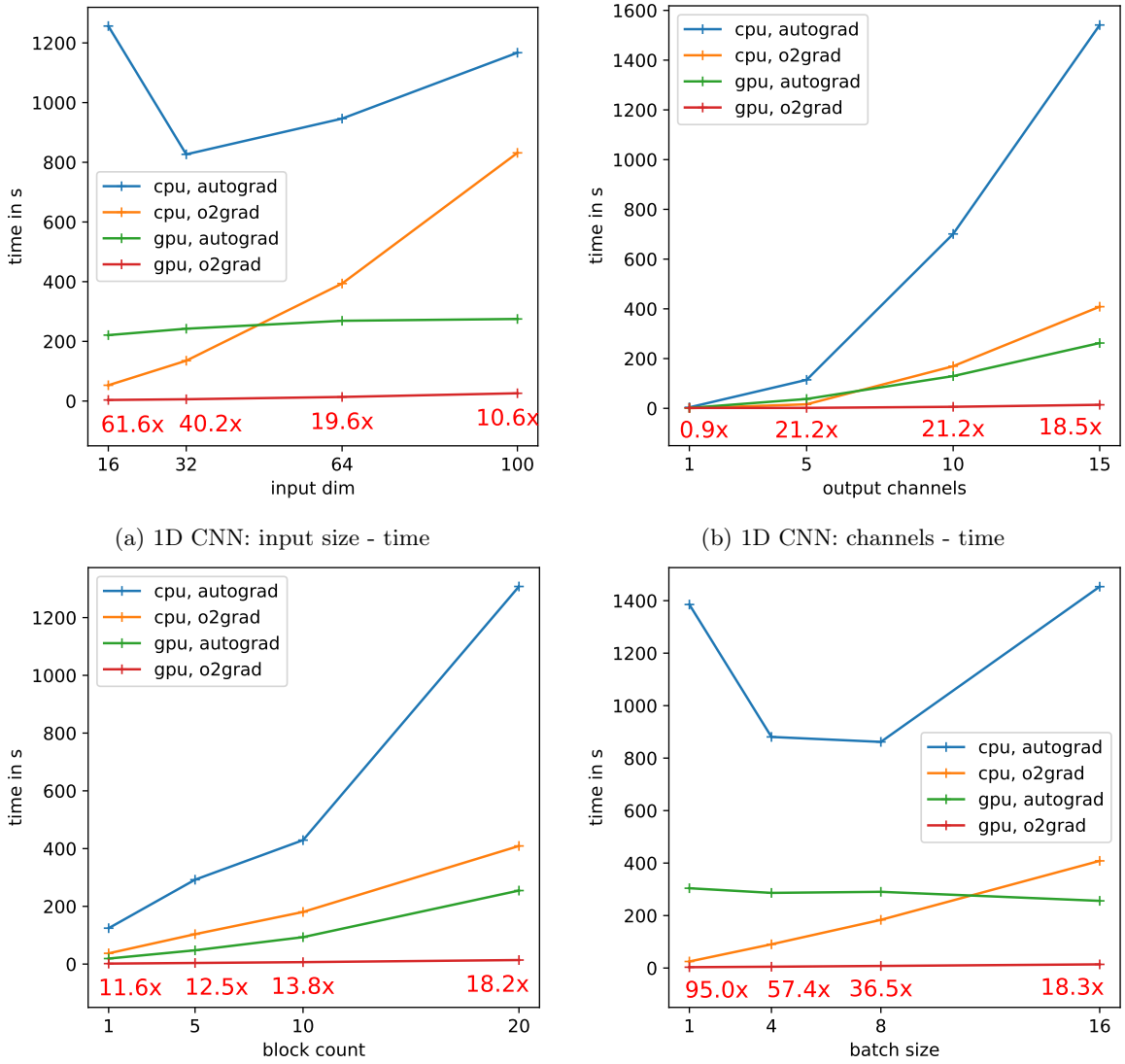
Block Count

For the block count, we expect the computation time to scale quadratically in our chosen architecture, since every block contains a parametric layer, and assuming those layers have the same number of parameters, the number of Hessian blocks `H[s][t]` to compute increases quadratically with the number of parametric layers and thus with the number of blocks. Our observations appear to be in line with our theoretical expectations (see Figure 15c). Both for the memory usage and the error, we observe a similar dependence on block count (see Figure 16c and 23c) as on channel count, and we must assume the same explanation applies here as well.

Batch Size

As we would expect, we observe a dependence of computation time on batch size (see Figure 15d) similar to that of the MLP, although the curvature of the quadratic increase is small for both *O2Grad* methods and barely appreciated in the plot. The dependence of memory usage on batch size shows, again, a similar behaviour as the dependence on channels and block count, with the memory usage dropping for a batch size of 16 (see Figure 16d). We assume the underlying source is the same as for the drop in memory usage observed in the output channels and block count plots, thus making further investigation necessary. For the error, we observe what appears to be an approximately linear increase as a function of batch size in both *O2Grad* methods, while the error for the autograd methods remain stable (see Figure 23d). However, this may only be an apparent trend and the error might drop to lower levels, as observed in the error plots for output channels and block count.

Although our results show that the computation time of the Hessian still benefits greatly from the usage of *O2Grad* on GPU, we have yet to find out whether this benefit can be sustained for 2D CNNs. An input size of 64 may be a reasonable size for 1D toy problems, but since 2D convolution layers have two input dimensions, this translates to an input size of merely 8 per dimension, which we expect to be on the lower end of what can be considered a suitable problem for 2D CNNs.



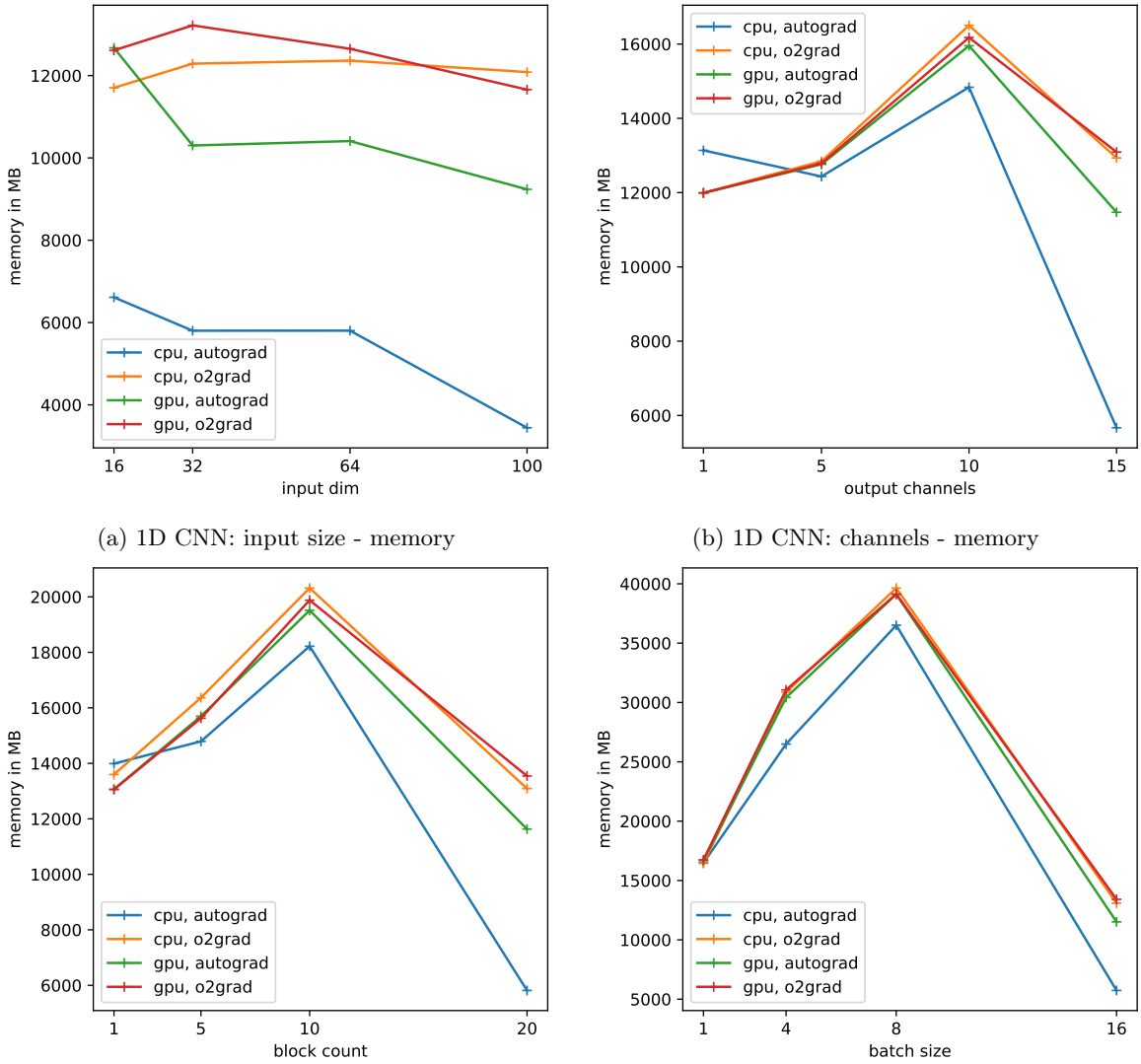
(a) 1D CNN: input size - time

(b) 1D CNN: channels - time

(c) 1D CNN: block count - time

(d) 1D CNN: batch size - time

Figure 15: Time to compute the Hessian of a 1D CNN, as a function of (a) input size, (b) output channels, (c) block count and (d) batch size, for a fix parameter baseline of (`input_size` = 64, `output_channels` = 15, `block_cnt` = 20, `batch_size` = 16. The parameters that are not subject to variation in the respective graphic correspond to the baseline. All data points are averaged over 3 measurements (except when the input size is 8, in which case the data points come from a single measurement). Annotated with smallest speedups over the autograd methods.



(a) 1D CNN: input size - memory

(b) 1D CNN: channels - memory

(c) 1D CNN: block count - memory

(d) 1D CNN: batch size - memory

Figure 16: Maximum absolute errors of computing the Hessian of a 1D CNN, as a function of (a) input size, (b) output channels, (c) block count and (d) batch size, for a fix parameter baseline of (`input_size` = 64, `output_channels` = 15, `block_cnt` = 20, `batch_size` = 16. The parameters that are not subject to variation in the respective graphic correspond to the baseline. All data points are averaged over 3 measurements (except when the input size is 8, in which case the data points come from a single measurement).

5.4 2D CNN

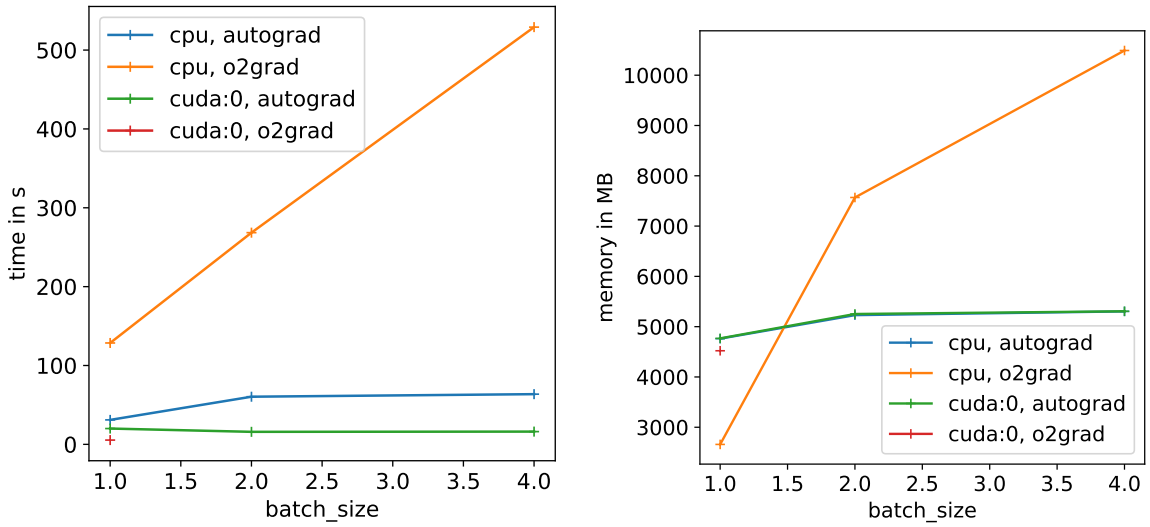
Unfortunately, the Second Order BP algorithm seems to be less suited for 2D CNNs. To test the viability of using the algorithm for calculating the Hessian on a small 2D CNN, we use a convnet using the same structure as the one defined in the previous section, but replace all 1D layers with their respective 2D layer equivalents. We analyze the performance of Hessian calculations subject to MNIST-sized input. We choose a block size of 1, channel dimensions of 10 and batch sizes of 1, 2 and 4. The results can be seen in in Figure 17 and Table 3: Using the GPU and *O2Grad*, we are barely able to compute the Hessian of the network for batch size 1, and for minibatches of size 2 and 4, the algorithm runs out of GPU memory and fails with a CUDA OOM exception. Although for a batch size of 1 and at a 5.4s computation time, it is still 3.7× faster than autograd (at a 20.0s), the speedup is also much smaller than the ones measured for the fully connected network. A plausible explanation for this decrease in speedup is that autograd backpropagates much faster through sparse layers like 2d convolutional layers than through dense layers such as fully connected layers, thus decreasing the time advantage of *O2Grad*. Unfortunately, the situation does not improve when using *O2Grad* on a CPU, as the algorithm consistently takes >5× more time than its autograd counterparts, while also using far more memory as the batch size increases. Unfortunately, this renders *O2Grad* useless for anything other than very small toy examples when working with 2D (fully) convolutional architectures, but given how time-efficient autograd is at performing a Second-Order BP through convolutional architectures in comparison, it might be better to rely on autograd in medium- to higher-dimensional cases.

Since the prospect of calculating the Hessian online not just for shallow MLPs and 1D CNNs, but also for 2D CNNs is an interesting research objective, we try measuring the performance for a dataset with even smaller inputs. The USPS handwritten digit dataset [14] contains 4649 greyscale images of handwritten digits, sized 16 × 16 pixels each, making it a suitable candidate. Table 15 gives us an impression of what can be calculated with *O2Grad* on GPU in this range of dimensions with the hardware available.

Input Dimension

We observe similar behaviour in computation time for the 1D CNN and for the 2D CNN (see Figure 18a). The autograd method on CPU has the same drop in computation time from input size 4 to input size 8, and shows signs of linearly increasing thereafter, while autograd on GPU grows linearly at a very small rate. *O2Grad* on CPU also features superlinear growth, which is more stark in the 2D convnet than in the 1D convnet however, as this method takes more time than any other method for an input size of 20. The *O2Grad* method takes less time than the other methods, but fails with an OOM exception at an input size of 20. The memory usage curves are quite different compared to those of the 1D CNN, memory usage never being as high, and the memory usage does not mysteriously drop from input size 4 to input size 8, but rather increase at same time that the computation time increases. The errors appear to be more or less stable within an order of 10^{-7}, with the exception of autograd on GPU spiking up to an order of 10^{-5} at an input size of 20, but there is no reason to suspect this is anything but an outlier. If anything, this reinforces that *O2Grad* has a high accuracy within what we can reasonably expect from these methods.

[14] http://www.gaussianprocess.org/gpml/data/



(a) (b)

Figure 17: Time (left) and memory (right) usage of calculating the Hessian of a 2D CNN for MNIST, built from two conv - relu blocks and 4 conv - relu - pool downsampling blocks, for batch sizes 1, 2 and 4. For *O2Grad* with CUDA, the algorithm is barely able to handle a batch size of 1, but fails for all higher batch sizes (thus only one measurement shows up in the figure).

Output Channels

1D convolution layers and 2D convolutions layers operate the same way on the channel dimension and thus should have a similar dependency. *O2Grad* on CPU and autograd on GPU show a clearly accelerated increase of computation time, while for autograd on CPU it is less clear from our measurements this time, but we can assume this trend to be present here as well. The *O2Grad* method on GPU consistently takes less time, but fails with an OOM exception at 15 output channels. The memory usage also accelerates as a function of output channel count, but does not drop as the memory usage goes up to 15. Just as for the 1D CNNs, the error spikes for 5 output channels, further supporting the theory that is due to some intrinsic numerical error during autograd backpropagation.

Block Count

As in the 1D CNN case, our measurements reflect the expected linear increase of computation time for the *O2Grad* method, and a quadratic increase for the autograd methods. However, for the 2D CNN, the quadratic increase is more pronounced, as the computation time increases considerably from a block count of 15 to a block count of 20 (see Figure 18c). Interestingly, the measurement for *O2Grad* on GPU fails with an OOM exception. This result is surprising, because the memory required to calculate the Hessian can only increase as a function of block count through the additional number of cOIJs that need to be stored in every step, but those get moved to the CPU in every step to save memory. Further investigation is required to understand why this happens. Memory usage increases linearly as expected (see Figure 19c), and the error stays well-behaved within a magnitude

block_cnt	batch_size	o2grad	device	time in	mem in MB
2	1	True	cpu	128.564604	2658.899248
2	1	True	cuda:0	5.418229	4520.993395
2	1	False	cpu	31.015664	4761.333078
2	1	False	cuda:0	20.044696	4767.152813
2	2	True	cpu	268.452337	7568.507811
2	2	True	cuda:0	NaN	NaN
2	2	False	cpu	60.421726	5229.026406
2	2	False	cuda:0	15.881729	5252.066406
2	4	True	cpu	529.110169	10491.128717
2	4	True	cuda:0	NaN	NaN
2	4	False	cpu	63.658702	5303.443054
2	4	False	cuda:0	16.176676	5305.862811

Table 3: Time and memory usage of calculating the Hessian of a 2D convnet for MNIST, built from two conv - relu blocks and 4 conv - relu - pool downsampling blocks, for batch sizes 1, 2 and 4. For *O2Grad* with CUDA, the algorithm is barely able to handle a batch size of 1, but fails for higher batch sizes (these correspond to the rows with NaNs).

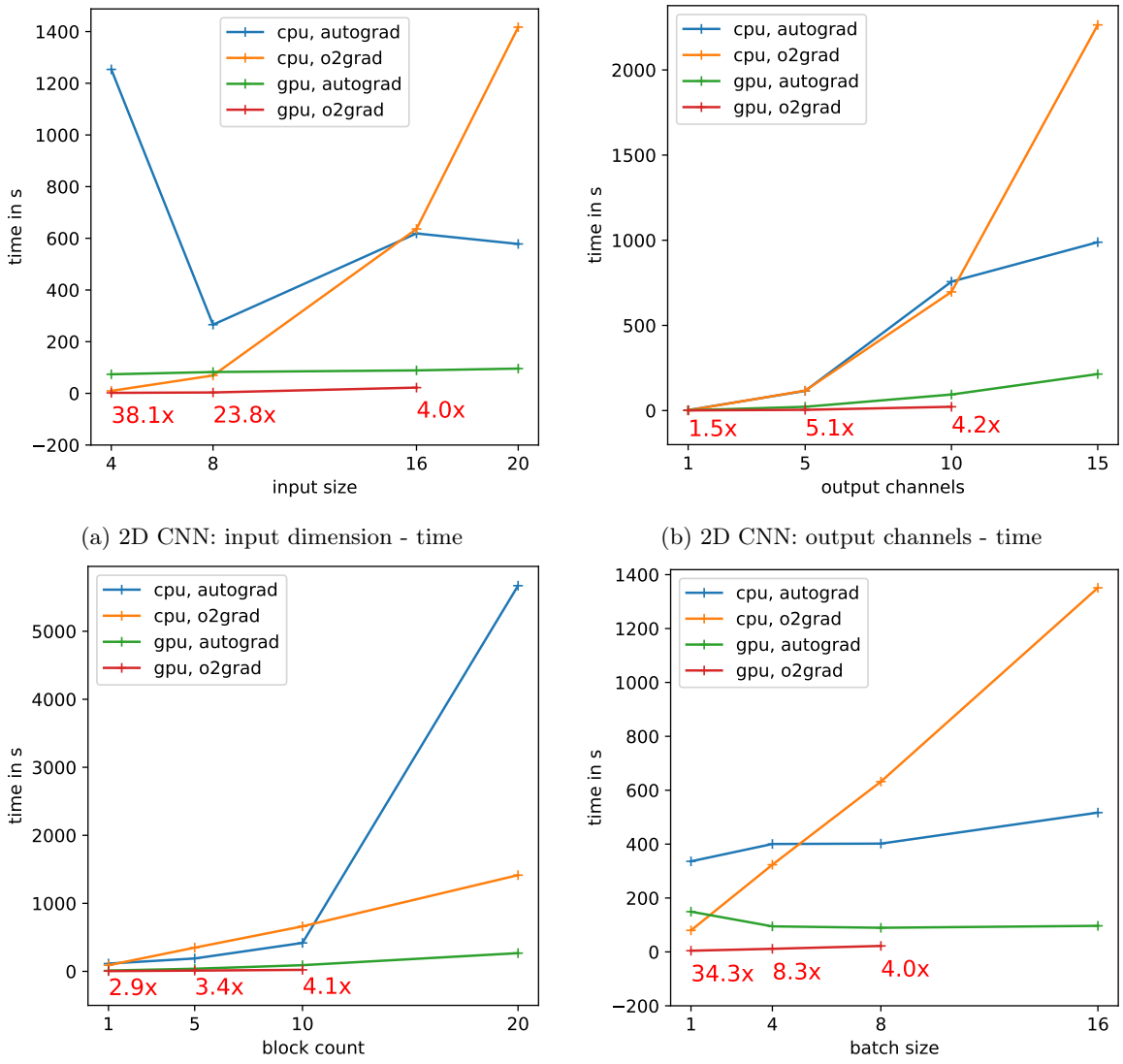
of 10^{-8} (see Figure 24c).

Batch Size

The time-dependency mirrors that of the 1D CNNs (see Figure 18d). This is what we expect, since 1D and 2D convolution layers operate exactly the same way on the batch dimension. Memory usage increases as a function of batch size (see Figure 19d), but appears to increase at a slower rate for higher batch sizes, but why this happens is not clear. As for the error, it seems to increase linearly as a function of a batch size with the inspected range, just like in the 1D CNN case. However, this seems to be the case not only for the *O2Grad* methods, but also for the autograd method on GPU, suggesting the source is an accumulating numerical error in the normal backpropagation step (possibly in gradient calculation).

5.5 Conclusion

In conclusion, we have shown that *O2Grad* can be used to greatly increase the speed of the Hessian calculation of both shallow MLPs and deeper 1D and 2D CNN architectures, but only with tight constraints. While using *O2Grad* can lead to speedups far greater than 10× for MLPs and 1D CNN architectures, for the sparser 2D CNNs the speedup shrinks down to a single-digit factor. Furthermore, *O2Grad* on the CPU will often be slower than the autograd method on GPU, making only *O2Grad* on the GPU significantly faster. However, because the intermediate tensors calculated can become very big and GPU memory is far more limited than RAM, it can only be used to calculate the Hessian of small-dimensional network architectures before running out of memory. Finally, another problem of *O2Grad* is that there are some parts of its time and memory complexity as functions of the model architecture that we don't yet fully understand, at least not given our current measurements. Further analysis will be necessary to remove the remaining uncertainties.



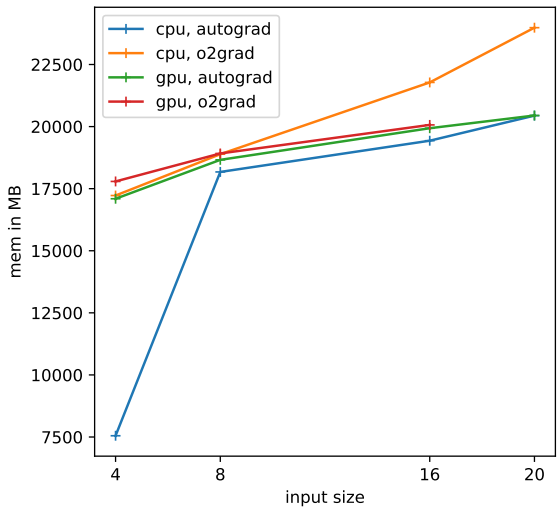
(a) 2D CNN: input dimension - time

(b) 2D CNN: output channels - time

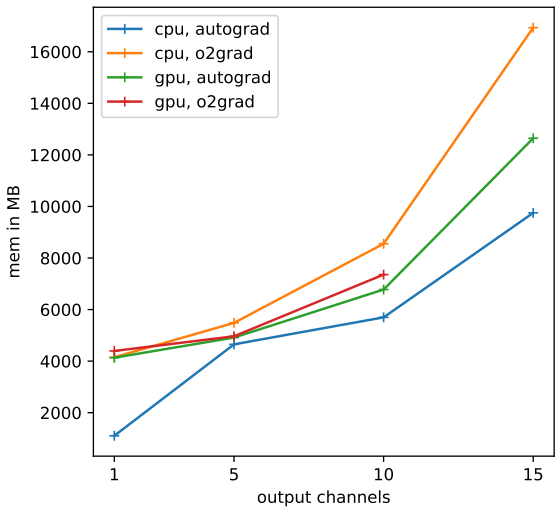
(c) 2D CNN: block count - time

(d) 2D CNN: batch size - time

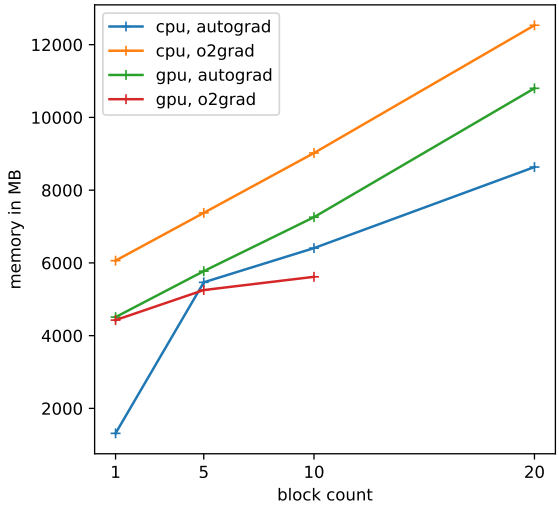
Figure 18: Time to compute the Hessian of a 2D CNN, as a function of (a) input size, (b) output channels, (c) block count and (d) batch size, for a fix parameter baseline of (`input_size` = 16, `output_channels` = 10, `block_cnt` = 10, `batch_size` = 8. The parameters that are not subject to variation in the respective graphic correspond to the baseline. All data points are averaged over 3 measurements. Annotated with smallest speedups over the autograd methods.



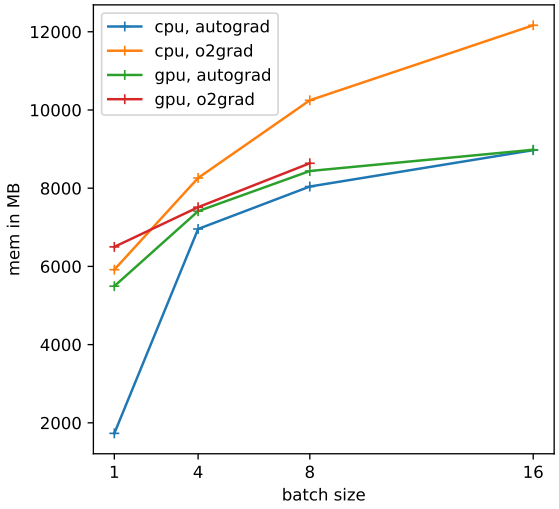
(a) 2D CNN: input dimension - memory usage



(b) 2D CNN: output channels - memory usage



(c) 2D CNN: block count - memory usage



(d) 2D CNN: batch size - memory usage

Figure 19: Memory usage to compute the Hessian of a 2D CNN, as a function of (a) input size, (b) output channels, (c) block count and (d) batch size, for a fix parameter baseline of (`input_size` = 16, `output_channels` = 10, `block_cnt` = 10, `batch_size` = 8. The parameters that are not subject to variation in the respective graphic correspond to the baseline. All data points are averaged over 3 measurements.

While not suitable for productive use, it can be a useful tool for research on small toy models, and this is just what we need for our experiments on chaotic training, to be discussed in the next section.

References

[Amari, 1998] Amari, S.-i. (1998). Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2):251–276.

[Ba et al., 2016] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.

[Botev et al., 2017] Botev, A., Ritter, H., and Barber, D. (2017). Practical gauss-newton optimisation for deep learning.

[Brown et al., 2020] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

[Cun et al., 1990] Cun, Y. L., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann.

[Dumoulin and Visin, 2018] Dumoulin, V. and Visin, F. (2018). A guide to convolution arithmetic for deep learning.

[Golmant et al., 2018] Golmant, N., Yao, Z., Gholami, A., Mahoney, M., and Gonzalez, J. (2018). pytorch-hessian-eigenthings: efficient pytorch hessian eigendecomposition.

[He, 2019] He, H. (2019). The state of machine learning frameworks in 2019. *The Gradient*.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition.

[Martens and Grosse, 2020] Martens, J. and Grosse, R. (2020). Optimizing neural networks with kronecker-factored approximate curvature.

[Mizutani et al., 2005] Mizutani, E., Dreyfus, S., and Demmel, J. (2005). Second-order backpropagation algorithms for a stagewise-partitioned separable hessian matrix. volume 2, pages 1027 – 1032 vol. 2.

[Pascanu and Bengio, 2014] Pascanu, R. and Bengio, Y. (2014). Revisiting natural gradient for deep networks.

[Pearlmutter, 1994] Pearlmutter, B. A. (1994). Fast Exact Multiplication by the Hessian. *Neural Computation*, 6(1):147–160.

[Roux et al., 2007] Roux, N. L., Manzagol, P.-A., and Bengio, Y. (2007). Topmoumoute online natural gradient algorithm. In *NIPS*.

[Sa et al., 2017] Sa, C. D., He, B., Mitliagkas, I., Ré, C., and Xu, P. (2017). Accelerated stochastic power iteration.

[Santurkar et al., 2019] Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2019). How does batch normalization help optimization?

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

[Zhang et al., 2017] Zhang, H., Xiong, C., Bradbury, J., and Socher, R. (2017). Block-diagonal hessian-free optimization for training neural networks.

A Proofs

A.1 Block-Diagonal

The Hessian of an ANN can generally not be expected to be block-diagonal. To see this, consider a simple 2-layer network with 2 weight neurons, no bias, no activation functions and Mean Squared Error (MSE) as loss. The network is described by equations:

$$x_1 = w_1 x_0, \qquad y = x_2 = w_2 x_1, \qquad l(t, y) = \frac{1}{2}(t - y)^2 = \frac{1}{2}(t - w_2 w_1 x_0)^2 \tag{127}$$

Consider the off-diagonal block $\frac{\partial^2 l}{\partial w_1 \partial w_2}$. We have

$$\frac{\partial l}{\partial w_2} = (t - w_2 w_1 x_0) \cdot w_1 x_0. \tag{128}$$

Differentiating for w_1 and using the product rule, we obtain

$$\frac{\partial^2 l}{\partial w_1 \partial w_2} = (-w_1 x_0) \cdot w_1 x_0 + (t - w_2 w_1 x_0) \cdot x_0 = -(w_1 x_0)^2 + (t - w_2 w_1 x_0) x_0, \tag{129}$$

which is generally not 0, e.g. for $w_1, w_2 = 1$ and $x_0 \neq 0$ and $t \neq 2x_0$:

$$\frac{\partial^2 l}{\partial w_1 \partial w_2} = (t - 2x_0) x_0 \neq 0 \tag{130}$$

B Tables

B.1 Tensor Generation

Output-Input-Jacobian (OIJ)

batch size	input dim	output dim	device	time in s dense	time in s sparse	mem in MB dense	mem in MB sparse
1	20	20	cpu	0.000086	0.000336	1600	8000
1	20	20	cuda:0	0.000085	0.000339	1600	8000
2	20	20	cpu	0.000092	0.000364	6400	16000
2	20	20	cuda:0	0.000097	0.000365	6400	16000
8	20	20	cpu	0.000114	0.000588	102400	64000
8	20	20	cuda:0	0.000113	0.000573	102400	64000
16	20	20	cpu	0.000136	0.000878	409600	128000
16	20	20	cuda:0	0.000144	0.000869	409600	128000
32	1	20	cpu	0.000117	0.000351	81920	12800
32	1	20	cuda:0	0.000116	0.000353	81920	12800
32	2	20	cpu	0.000221	0.000416	163840	25600
32	2	20	cuda:0	0.000223	0.000409	163840	25600
32	10	20	cpu	0.000178	0.000853	819200	128000
32	10	20	cuda:0	0.000179	0.000871	819200	128000
32	20	1	cpu	0.000110	0.000344	81920	12800
32	20	1	cuda:0	0.000116	0.000346	81920	12800
32	20	2	cpu	0.000122	0.000402	163840	25600
32	20	2	cuda:0	0.000122	0.000404	163840	25600
32	20	10	cpu	0.000150	0.000863	819200	128000
32	20	10	cuda:0	0.000142	0.000867	819200	128000
32	20	20	cpu	0.000154	0.001493	1638400	256000
32	20	20	cuda:0	0.000154	0.001408	1638400	256000

Table 5: Time and memory for OIJ generation of `O2Linear` layer, for selected layer hyperparameters.

Output-Input-Hessian (OIH)

batch size	input dim	output dim	device	time in s dense	time in s sparse	mem in MB dense	mem in MB sparse
1	20	20	cpu	0.000030	0.000025	32000	0
1	20	20	cuda:0	0.000030	0.000025	32000	0
2	20	20	cpu	0.000946	0.000033	256000	0
2	20	20	cuda:0	0.000053	0.000032	256000	0
8	20	20	cpu	0.000329	0.000037	16384000	0
8	20	20	cuda:0	0.000208	0.000035	16384000	0
16	20	20	cpu	0.010700	0.000042	131072000	0
16	20	20	cuda:0	0.008425	0.000037	131072000	0
32	1	20	cpu	0.000055	0.000027	2621440	0
32	1	20	cuda:0	0.000048	0.000027	2621440	0
32	2	20	cpu	0.000089	0.000028	10485760	0
32	2	20	cuda:0	0.000071	0.000027	10485760	0
32	10	20	cpu	0.014681	0.000038	262144000	0
32	10	20	cuda:0	0.014307	0.000038	262144000	0
32	20	1	cpu	0.003106	0.000037	52428800	0
32	20	1	cuda:0	0.003043	0.000037	52428800	0
32	20	2	cpu	0.005703	0.000037	104857600	0
32	20	2	cuda:0	0.005699	0.000044	104857600	0
32	20	10	cpu	0.030076	0.000038	524288000	0
32	20	10	cuda:0	0.029140	0.000039	524288000	0
32	20	20	cpu	0.055728	0.000039	1048576000	0
32	20	20	cuda:0	0.057546	0.000039	1048576000	0

Table 7: Time and memory for OIH generation of `O2Linear` layer, for selected layer hyperparameters.

Output-Parameter-Jacobian (OPJ)

batch size	input dim	output dim	device	time in s dense	time in s sparse	mem in MB dense	mem in MB sparse
1	20	20	cpu	0.000075	0.000373	33600	8400
1	20	20	cuda:0	0.000121	0.000862	33600	8400
2	20	20	cpu	0.000079	0.000416	67200	16800
2	20	20	cuda:0	0.000116	0.000887	67200	16800
8	20	20	cpu	0.000127	0.000667	268800	67200
8	20	20	cuda:0	0.000118	0.000907	268800	67200
16	20	20	cpu	0.000170	0.001034	537600	134400
16	20	20	cuda:0	0.000115	0.000936	537600	134400
32	1	20	cpu	0.000087	0.000460	102400	25600
32	1	20	cuda:0	0.000118	0.000905	102400	25600
32	2	20	cpu	0.000208	0.000504	153600	38400
32	2	20	cuda:0	0.000117	0.000895	153600	38400
32	10	20	cpu	0.000193	0.001054	563200	140800
32	10	20	cuda:0	0.000117	0.000920	563200	140800
32	20	1	cpu	0.000063	0.000391	2688	13440
32	20	1	cuda:0	0.000115	0.000939	2688	13440
32	20	2	cpu	0.000103	0.000667	10752	26880
32	20	2	cuda:0	0.000115	0.000898	10752	26880
32	20	10	cpu	0.000173	0.001444	268800	134400
32	20	10	cuda:0	0.000167	0.001122	268800	134400
32	20	20	cpu	0.000185	0.001706	1075200	268800
32	20	20	cuda:0	0.000153	0.000937	1075200	268800

Table 9: Time and memory for OPJ generation of a `O2Linear` layer, for selected layer hyperparameters.

Output-Parameter-Hessian (OPH)

batch size	input dim	output dim	device	time in s dense	time in s sparse	mem in MB dense	mem in MB sparse
1	20	20	cpu	0.000420	0.000043	14112000	0
1	20	20	cuda:0	0.000261	0.000044	14112000	0
2	20	20	cpu	0.001403	0.000045	28224000	0
2	20	20	cuda:0	0.001063	0.000046	28224000	0
8	20	20	cpu	0.007626	0.000046	112896000	0
8	20	20	cuda:0	0.007034	0.000046	112896000	0
16	20	20	cpu	0.014682	0.000049	225792000	0
16	20	20	cuda:0	0.014232	0.000046	225792000	0
32	1	20	cpu	0.000075	0.000036	4096000	0
32	1	20	cuda:0	0.000063	0.000035	4096000	0
32	2	20	cpu	0.000095	0.000037	9216000	0
32	2	20	cuda:0	0.000079	0.000037	9216000	0
32	10	20	cpu	0.008087	0.000046	123904000	0
32	10	20	cuda:0	0.008237	0.000046	123904000	0
32	20	1	cpu	0.000045	0.000036	56448	0
32	20	1	cuda:0	0.000045	0.000036	56448	0
32	20	2	cpu	0.000058	0.000036	451584	0
32	20	2	cuda:0	0.000056	0.000036	451584	0
32	20	10	cpu	0.003565	0.000046	56448000	0
32	20	10	cuda:0	0.003395	0.000047	56448000	0
32	20	20	cpu	0.027337	0.000048	451584000	0
32	20	20	cuda:0	0.025825	0.000046	451584000	0

Table 10: Time and memory for OPH generation of a `O2Linear` layer, for selected layer hyperparameters.

Mixed Output-Parameter-Hessian (mOPH)

batch size	input dim	output dim	device	time in s dense	time in s sparse	mem in MB dense	mem in MB sparse
1	20	20	cpu	0.000192	0.000307	672000	8000
1	20	20	cuda:0	0.000182	0.000306	672000	8000
2	20	20	cpu	0.001048	0.000292	2688000	16000
2	20	20	cuda:0	0.000879	0.000285	2688000	16000
8	20	20	cpu	0.012410	0.000470	43008000	64000
8	20	20	cuda:0	0.011628	0.000469	43008000	64000
16	20	20	cpu	0.047692	0.000695	172032000	128000
16	20	20	cuda:0	0.048436	0.000699	172032000	128000
32	1	20	cpu	0.000286	0.000275	3276800	12800
32	1	20	cuda:0	0.000278	0.000272	3276800	12800
32	2	20	cpu	0.001192	0.000341	9830400	25600
32	2	20	cuda:0	0.001371	0.000342	9830400	25600
32	10	20	cpu	0.050721	0.000700	180224000	128000
32	10	20	cuda:0	0.051608	0.000699	180224000	128000
32	20	1	cpu	0.000525	0.000278	1720320	12800
32	20	1	cuda:0	0.000541	0.000275	1720320	12800
32	20	2	cpu	0.001840	0.000334	6881280	25600
32	20	2	cuda:0	0.001680	0.000332	6881280	25600
32	20	10	cpu	0.084024	0.000704	172032000	128000
32	20	10	cuda:0	0.084163	0.000700	172032000	128000
32	20	20	cpu	0.332656	0.001248	688128000	256000
32	20	20	cuda:0	0.327588	0.001175	688128000	256000

Table 11: Time and memory for mOPH generation of a `O2Linear` layer, for selected layer hyperparameters.

B.2 MLP

MLP Grid

output dim	input dim	batch size	o2grad	device	time in s	mem in MB	err
5	5	1	False	cpu	0.080	1075.0	0.00e+00
5	5	1	False	cuda:2	0.174	4087.0	1.49e-08
5	5	1	True	cpu	0.050	4093.8	1.49e-08
5	5	1	True	cuda:2	0.092	4333.6	1.49e-08
5	5	8	False	cpu	0.061	4333.7	0.00e+00
5	5	8	False	cuda:2	0.109	4333.7	2.98e-08
5	5	8	True	cpu	0.049	4333.8	2.98e-08
5	5	8	True	cuda:2	0.065	4333.9	2.98e-08
5	5	16	False	cpu	0.077	4333.9	0.00e+00
5	5	16	False	cuda:2	0.100	4333.9	5.96e-08
5	5	16	True	cpu	0.063	4334.3	5.96e-08
5	5	16	True	cuda:2	0.077	4334.3	5.96e-08
5	5	32	False	cpu	0.078	4334.4	0.00e+00
5	5	32	False	cuda:2	0.093	4334.4	4.47e-08
5	5	32	True	cpu	0.064	4336.0	1.79e-07
5	5	32	True	cuda:2	0.093	4336.3	4.47e-08
10	10	1	False	cpu	0.107	4336.4	0.00e+00
10	10	1	False	cuda:2	0.271	4336.4	7.45e-09
10	10	1	True	cpu	0.048	4336.4	1.49e-08
10	10	1	True	cuda:2	0.061	4336.5	1.49e-08
10	10	8	False	cpu	0.110	4336.5	0.00e+00
10	10	8	False	cuda:2	0.220	4336.5	1.49e-08
10	10	8	True	cpu	0.052	4336.5	1.49e-08
10	10	8	True	cuda:2	0.079	4336.6	1.49e-08
10	10	16	False	cpu	0.114	4336.6	0.00e+00
10	10	16	False	cuda:2	0.261	4336.6	1.49e-08
10	10	16	True	cpu	0.065	4337.2	2.98e-08
10	10	16	True	cuda:2	0.097	4337.6	1.49e-08
10	10	32	False	cpu	0.167	4337.6	0.00e+00
10	10	32	False	cuda:2	0.262	4337.6	1.49e-08
10	10	32	True	cpu	0.109	4343.2	1.49e-08
10	10	32	True	cuda:2	0.139	4343.6	2.98e-08
50	50	1	False	cpu	1.758	5064.2	0.00e+00
50	50	1	False	cuda:2	4.496	5164.4	3.73e-09
50	50	1	True	cpu	0.087	5160.3	3.73e-09
50	50	1	True	cuda:2	0.111	5136.1	3.73e-09
50	50	8	False	cpu	3.224	5157.8	0.00e+00
50	50	8	False	cuda:2	4.299	5157.4	5.59e-09
50	50	8	True	cpu	0.255	5150.6	3.73e-09
50	50	8	True	cuda:2	0.181	5126.0	3.73e-09

50	50	16	False	cpu	3.173	5167.4	0.00e+00
50	50	16	False	cuda:2	4.402	5169.1	3.73e-09
50	50	16	True	cpu	0.740	5210.0	3.73e-09
50	50	16	True	cuda:2	0.275	5236.8	3.73e-09
50	50	32	False	cpu	3.391	5283.0	0.00e+00
50	50	32	False	cuda:2	4.887	5283.0	3.73e-09
50	50	32	True	cpu	4.314	5279.5	3.73e-09
50	50	32	True	cuda:2	0.475	5237.0	5.59e-09
100	100	1	False	cpu	7.225	5980.5	0.00e+00
100	100	1	False	cuda:2	18.311	7427.7	2.79e-09
100	100	1	True	cpu	0.577	7677.4	1.40e-09
100	100	1	True	cuda:2	0.611	7567.1	2.79e-09
100	100	8	False	cpu	14.236	7450.2	0.00e+00
100	100	8	False	cuda:2	18.919	7459.1	1.86e-09
100	100	8	True	cpu	2.267	7444.3	1.86e-09
100	100	8	True	cuda:2	0.798	7298.8	1.86e-09
100	100	16	False	cpu	14.433	7475.0	0.00e+00
100	100	16	False	cuda:2	19.062	7479.9	1.86e-09
100	100	16	True	cpu	6.596	7454.9	2.79e-09
100	100	16	True	cuda:2	1.064	7266.3	2.79e-09
100	100	32	False	cpu	15.358	7517.3	0.00e+00
100	100	32	False	cuda:2	18.320	7501.3	2.79e-09
100	100	32	True	cpu	34.348	7525.2	3.73e-09
100	100	32	True	cuda:2	1.534	7168.6	2.79e-09

Table 12: Time, memory usage and computation errors for the Hessian of a N - N - N network, for different input sizes N and batch sizes of the input. The method indicates whether *O2Grad* or False is used to calculate the Hessian. Given a fix input and batch size, the error is the average absolute deviation of the Hessian resulting from a `(True, device)` configuration w.r.t. to the one resulting from the baseline configuration `('False', 'cpu')`. The time results are averaged over 3 runs.

MLP MNIST

input dim	inter dim	output dim	batch size	o2grad	device	time in s	mem in MB	err
784	10	10	1	False	cpu	2.586	1194.3	0.00e+00
784	10	10	1	False	cuda:2	7.406	4448.8	1.49e-08
784	10	10	1	True	cpu	0.109	4471.2	7.45e-09
784	10	10	1	True	cuda:2	0.375	4713.5	1.49e-08
784	10	10	8	False	cpu	3.701	4709.6	0.00e+00
784	10	10	8	False	cuda:2	6.640	4713.6	1.49e-08
784	10	10	8	True	cpu	0.222	4889.7	1.49e-08
784	10	10	8	True	cuda:2	0.137	4677.9	1.49e-08
784	10	10	16	False	cpu	3.884	4731.9	0.00e+00
784	10	10	16	False	cuda:2	8.177	4737.4	1.49e-08
784	10	10	16	True	cpu	0.552	5170.5	2.98e-08
784	10	10	16	True	cuda:2	0.198	4885.5	7.45e-09
784	10	10	32	False	cpu	5.080	4834.9	0.00e+00
784	10	10	32	False	cuda:2	6.994	4840.8	1.49e-08
784	10	10	32	True	cpu	1.839	6857.3	2.24e-08
784	10	10	32	True	cuda:2	0.400	4884.8	2.98e-08
784	20	10	1	False	cpu	5.254	5227.7	0.00e+00
784	20	10	1	False	cuda:2	16.914	5962.5	2.24e-08
784	20	10	1	True	cpu	0.315	6733.7	1.12e-08
784	20	10	1	True	cuda:2	0.365	6485.6	2.24e-08
784	20	10	8	False	cpu	9.597	5944.7	0.00e+00
784	20	10	8	False	cuda:2	13.731	5957.5	1.49e-08
784	20	10	8	True	cpu	0.692	6718.7	1.49e-08
784	20	10	8	True	cuda:2	0.430	6522.2	1.49e-08
784	20	10	16	False	cpu	9.877	5959.5	0.00e+00
784	20	10	16	False	cuda:2	13.950	5977.3	1.49e-08
784	20	10	16	True	cpu	1.282	7121.4	2.98e-08
784	20	10	16	True	cuda:2	0.512	6467.8	1.49e-08
784	20	10	32	False	cpu	10.499	5953.3	0.00e+00
784	20	10	32	False	cuda:2	16.594	5955.6	2.24e-08
784	20	10	32	True	cpu	3.750	8458.7	2.24e-08
784	20	10	32	True	cuda:2	NaN	NaN	NaN
784	30	10	1	False	cpu	8.128	6610.4	0.00e+00
784	30	10	1	False	cuda:2	27.296	7833.0	2.24e-08
784	30	10	1	True	cpu	0.687	10001.8	7.45e-09
784	30	10	1	True	cuda:2	NaN	NaN	NaN
784	30	10	8	False	cpu	15.633	7818.5	0.00e+00
784	30	10	8	False	cuda:2	23.265	7859.4	1.49e-08
784	30	10	8	True	cpu	1.252	10053.7	1.49e-08
784	30	10	8	True	cuda:2	NaN	NaN	NaN
784	30	10	16	False	cpu	16.247	7784.2	0.00e+00

784	30	10	16	False	cuda:2	25.157	7805.2	1.49e-08
784	30	10	16	True	cpu	2.538	10010.7	2.98e-08
784	30	10	16	True	cuda:2	NaN	NaN	NaN
784	30	10	32	False	cpu	17.169	7787.1	0.00e+00
784	30	10	32	False	cuda:2	21.281	7821.8	2.24e-08
784	30	10	32	True	cpu	8.093	11312.5	2.24e-08
784	30	10	32	True	cuda:2	NaN	NaN	NaN

Table 13: Time, memory usage and computation errors for the Hessian of a N - N - N network, for different input sizes N and batch sizes of the input. The method indicates whether *O2Grad* or False is used to calculate the Hessian. Given a fix input and batch size, the error is the average absolute deviation of the Hessian resulting from a `(True, device)` configuration w.r.t. to the one resulting from the baseline configuration `('False', 'cpu')`. The time results are averaged over 3 runs.

B.3 1D CNN

input dim	out channels	block cnt	batch size	o2grad	device	time	mem	err
100	15	20	16	False	cpu	1167.166	5668.7	0.00e+00
100	15	20	16	False	cuda:3	275.005	11413.0	1.12e-08
100	15	20	16	True	cpu	831.573	13723.0	5.22e-08
100	15	20	16	True	cuda:3	26.063	13369.9	1.49e-08
16	15	20	16	False	cpu	1256.558	17200.1	0.00e+00
16	15	20	16	False	cuda:3	220.948	19061.5	1.49e-08
16	15	20	16	True	cpu	52.866	19111.1	3.73e-08
16	15	20	16	True	cuda:3	3.588	19026.0	2.98e-08
32	15	20	16	False	cpu	826.634	23247.4	0.00e+00
32	15	20	16	False	cuda:3	242.393	25049.0	1.12e-08
32	15	20	16	True	cpu	135.184	25377.8	4.47e-08
32	15	20	16	True	cuda:3	6.026	25082.6	2.98e-08
64	15	20	16	False	cpu	946.427	29153.7	0.00e+00
64	15	20	16	False	cuda:3	268.940	30929.2	1.12e-08
64	15	20	16	True	cpu	393.714	31832.2	4.10e-08
64	15	20	16	True	cuda:3	13.726	31207.8	2.98e-08
64	15	20	16	False	cpu	1541.286	5668.5	0.00e+00
64	15	20	16	False	cuda:2	262.523	11471.3	1.12e-08
64	15	20	16	True	cpu	408.648	12933.8	4.10e-08
64	15	20	16	True	cuda:2	14.209	13092.4	2.98e-08
64	1	20	16	False	cpu	3.966	13137.7	0.00e+00
64	1	20	16	False	cuda:2	1.556	11990.5	0.00e+00
64	1	20	16	True	cpu	0.713	11992.0	0.00e+00
64	1	20	16	True	cuda:2	1.709	11992.4	0.00e+00
64	5	20	16	False	cpu	114.485	12433.9	0.00e+00
64	5	20	16	False	cuda:2	37.578	12762.0	5.96e-08
64	5	20	16	True	cpu	16.092	12843.6	1.79e-07
64	5	20	16	True	cuda:2	1.769	12787.7	1.79e-07
64	10	20	16	False	cpu	701.130	14836.5	0.00e+00
64	10	20	16	False	cuda:2	129.631	15952.8	1.49e-08
64	10	20	16	True	cpu	169.355	16502.4	7.45e-08
64	10	20	16	True	cuda:2	6.120	16175.7	7.45e-08
64	15	20	16	False	cpu	1307.781	5815.4	0.00e+00
64	15	20	16	False	cuda:3	254.639	11631.1	1.12e-08
64	15	20	16	True	cpu	409.102	13090.4	4.10e-08
64	15	20	16	True	cuda:3	14.021	13543.1	2.98e-08
64	15	1	16	False	cpu	124.388	13989.6	0.00e+00
64	15	1	16	False	cuda:3	19.193	13066.8	1.49e-08
64	15	1	16	True	cpu	37.302	13597.3	2.98e-08
64	15	1	16	True	cuda:3	1.658	13055.2	2.98e-08
64	15	5	16	False	cpu	292.324	14790.5	0.00e+00

64	15	5	16	False	cuda:3	47.777	15698.9	1.12e-08
64	15	5	16	True	cpu	103.573	16362.9	2.98e-08
64	15	5	16	True	cuda:3	3.828	15625.0	2.98e-08
64	15	10	16	False	cpu	429.251	18219.3	0.00e+00
64	15	10	16	False	cuda:3	93.289	19513.6	9.04e-08
64	15	10	16	True	cpu	180.529	20316.7	2.98e-08
64	15	10	16	True	cuda:3	6.746	19879.8	9.04e-08
64	15	20	16	False	cpu	1453.507	5736.4	0.00e+00
64	15	20	16	False	cuda:3	256.063	11516.4	1.12e-08
64	15	20	16	True	cpu	408.450	13088.5	4.10e-08
64	15	20	16	True	cuda:3	13.998	13413.3	2.98e-08
64	15	20	1	False	cpu	1385.397	16490.2	0.00e+00
64	15	20	1	False	cuda:3	304.188	16674.5	1.12e-08
64	15	20	1	True	cpu	24.984	16438.8	7.45e-09
64	15	20	1	True	cuda:3	3.201	16747.1	7.45e-09
64	15	20	4	False	cpu	880.805	26495.0	0.00e+00
64	15	20	4	False	cuda:3	286.427	30416.5	1.49e-08
64	15	20	4	True	cpu	90.504	30859.6	1.49e-08
64	15	20	4	True	cuda:3	4.988	31057.9	1.49e-08
64	15	20	8	False	cpu	861.878	36507.1	0.00e+00
64	15	20	8	False	cuda:3	290.506	39121.7	1.49e-08
64	15	20	8	True	cpu	183.696	39641.1	2.24e-08
64	15	20	8	True	cuda:3	7.953	39128.8	1.49e-08

Table 14: Time and memory usage of calculating the Hessian of a 1D convnet with architecture as in Figure 14. The label 'o2grad' indicates whether *O2Grad* or autograd is used to calculate the Hessian. Given a fix input and batch size, the error is the average absolute deviation of the Hessian resulting from a given `(o2grad, device)` configuration w.r.t. to the Hessian resulting from the baseline configuration `('False', 'cpu')`. Cells with NaN indicate the calculation failed with a CUDA OOM exception. The time results are averaged over 3 runs.

B.4 2D CNN

input dim	out channels	block cnt	batch size	o2grad	device	time in s	mem in MB	err
4	10	10	8	False	cpu	1253.586	7550.6	0.00e+00
4	10	10	8	False	cuda:2	73.795	17094.2	1.54e-08
4	10	10	8	True	cpu	9.600	17223.9	7.45e-08
4	10	10	8	True	cuda:2	1.939	17791.3	7.45e-08
8	10	10	8	False	cpu	265.745	18173.7	0.00e+00
8	10	10	8	False	cuda:2	82.546	18660.5	2.79e-08
8	10	10	8	True	cpu	69.098	18888.9	7.45e-08
8	10	10	8	True	cuda:2	3.471	18917.9	7.45e-08
16	10	10	8	False	cpu	618.851	19431.1	0.00e+00
16	10	10	8	False	cuda:2	88.851	19934.2	3.73e-08
16	10	10	8	True	cpu	636.120	21777.9	9.31e-08
16	10	10	8	True	cuda:2	22.284	20067.5	7.45e-08
20	10	10	8	False	cpu	578.127	20441.9	0.00e+00
20	10	10	8	False	cuda:2	95.924	20446.3	5.26e-05
20	10	10	8	True	cpu	1417.006	23982.8	1.08e-07
20	10	10	8	True	cuda:2	NaN	NaN	NaN
16	1	10	8	False	cpu	2.880	1105.0	0.00e+00
16	1	10	8	False	cuda:1	1.556	4132.5	0.00e+00
16	1	10	8	True	cpu	1.305	4150.1	0.00e+00
16	1	10	8	True	cuda:1	1.044	4394.9	0.00e+00
16	5	10	8	False	cpu	115.448	4651.1	0.00e+00
16	5	10	8	False	cuda:1	22.115	4914.8	3.90e-07
16	5	10	8	True	cpu	117.124	5483.1	1.49e-07
16	5	10	8	True	cuda:1	4.372	4963.3	3.90e-07
16	10	10	8	False	cpu	756.945	5697.2	0.00e+00
16	10	10	8	False	cuda:1	94.235	6776.4	3.73e-08
16	10	10	8	True	cpu	696.913	8553.0	9.31e-08
16	10	10	8	True	cuda:1	22.273	7356.4	7.45e-08
16	15	10	8	False	cpu	988.530	9750.6	0.00e+00
16	15	10	8	False	cuda:1	214.304	12645.8	4.47e-08
16	15	10	8	True	cpu	2264.342	16934.6	8.94e-08
16	15	10	8	True	cuda:1	NaN	NaN	NaN
16	10	1	8	False	cpu	113.893	1313.7	0.00e+00
16	10	1	8	False	cuda:2	11.749	4509.8	2.98e-08
16	10	1	8	True	cpu	90.481	6061.9	7.45e-08
16	10	1	8	True	cuda:3	4.119	4427.2	0.00e+00
16	10	5	8	False	cpu	189.418	5465.8	0.00e+00
16	10	5	8	False	cuda:2	38.428	5773.0	2.24e-08
16	10	5	8	True	cpu	348.447	7375.3	7.45e-08
16	10	5	8	True	cuda:3	11.423	5253.3	0.00e+00
16	10	10	8	False	cpu	418.202	6406.5	0.00e+00

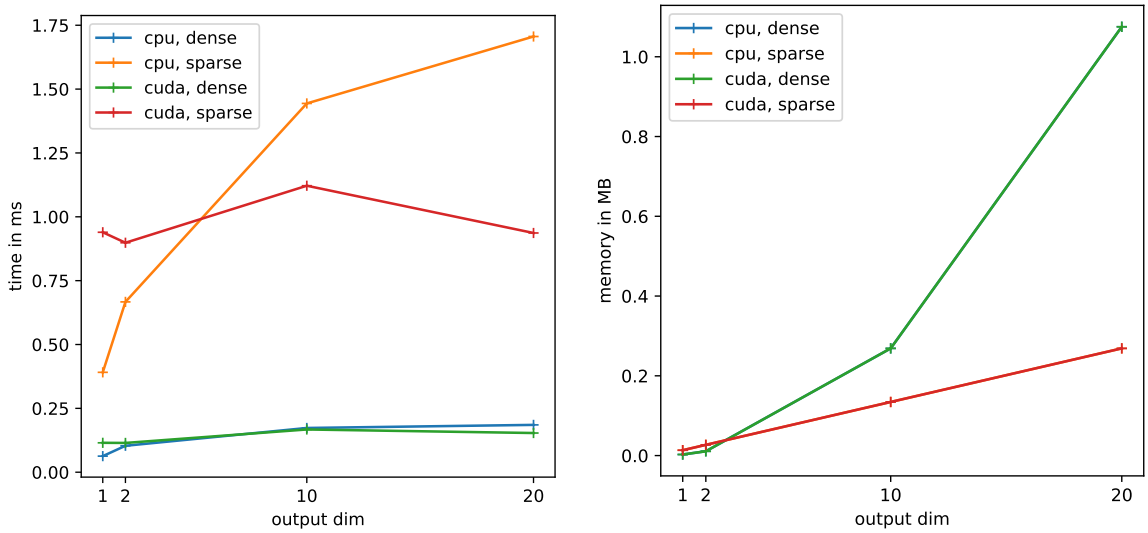
16	10	10	8	False	cuda:2	90.843	7257.0	3.73e-08
16	10	10	8	True	cpu	661.707	9021.1	9.31e-08
16	10	10	8	True	cuda:3	22.333	5617.2	0.00e+00
16	10	20	8	False	cpu	5670.095	8635.1	0.00e+00
16	10	20	8	False	cuda:2	267.971	10800.0	2.24e-08
16	10	20	8	True	cpu	1414.708	12532.9	7.45e-08
16	10	20	8	True	cuda:3	NaN	NaN	NaN
16	10	10	1	False	cpu	336.113	1730.7	0.000e+00
16	10	10	1	False	cuda:1	149.339	5494.5	2.980e-08
16	10	10	1	True	cpu	79.890	5916.3	3.725e-08
16	10	10	1	True	cuda:1	4.348	6499.1	2.980e-08
16	10	10	4	False	cpu	400.430	6957.0	0.000e+00
16	10	10	4	False	cuda:1	94.778	7412.1	2.235e-08
16	10	10	4	True	cpu	323.528	8260.9	7.823e-08
16	10	10	4	True	cuda:1	11.437	7513.5	2.980e-08
16	10	10	8	False	cpu	401.846	8042.5	0.000e+00
16	10	10	8	False	cuda:1	89.503	8439.4	3.725e-08
16	10	10	8	True	cpu	631.924	10247.7	9.313e-08
16	10	10	8	True	cuda:1	22.103	8637.4	7.451e-08
16	10	10	16	False	cpu	516.955	8972.6	0.000e+00
16	10	10	16	False	cuda:1	96.721	8983.0	7.945e-08
16	10	10	16	True	cpu	1351.234	12166.0	1.788e-07
16	10	10	16	True	cuda:1	NaN	NaN	NaN

Table 15: Time and memory usage of calculating the Hessian of a 2D convnet with architecture as in Figure 14, but with 2D layers instead. The label 'o2grad' indicates whether *O2Grad* or autograd is used to calculate the Hessian. Given a fix input and batch size, the error is the average absolute deviation of the Hessian resulting from a given `(o2grad, device)` configuration w.r.t. to the Hessian resulting from the baseline configuration `('False', 'cpu')`. Cells with NaN indicate the calculation failed with a CUDA OOM exception. The time results are averaged over 3 runs.

C Figures

C.1 Measurements

C.1.1 Tensor Generation



(a) (b)

(c) (d)

Figure 20: Time-Memory tradeoff of generating the dense/sparse OPJ tensors of a `O2Linear` layer.

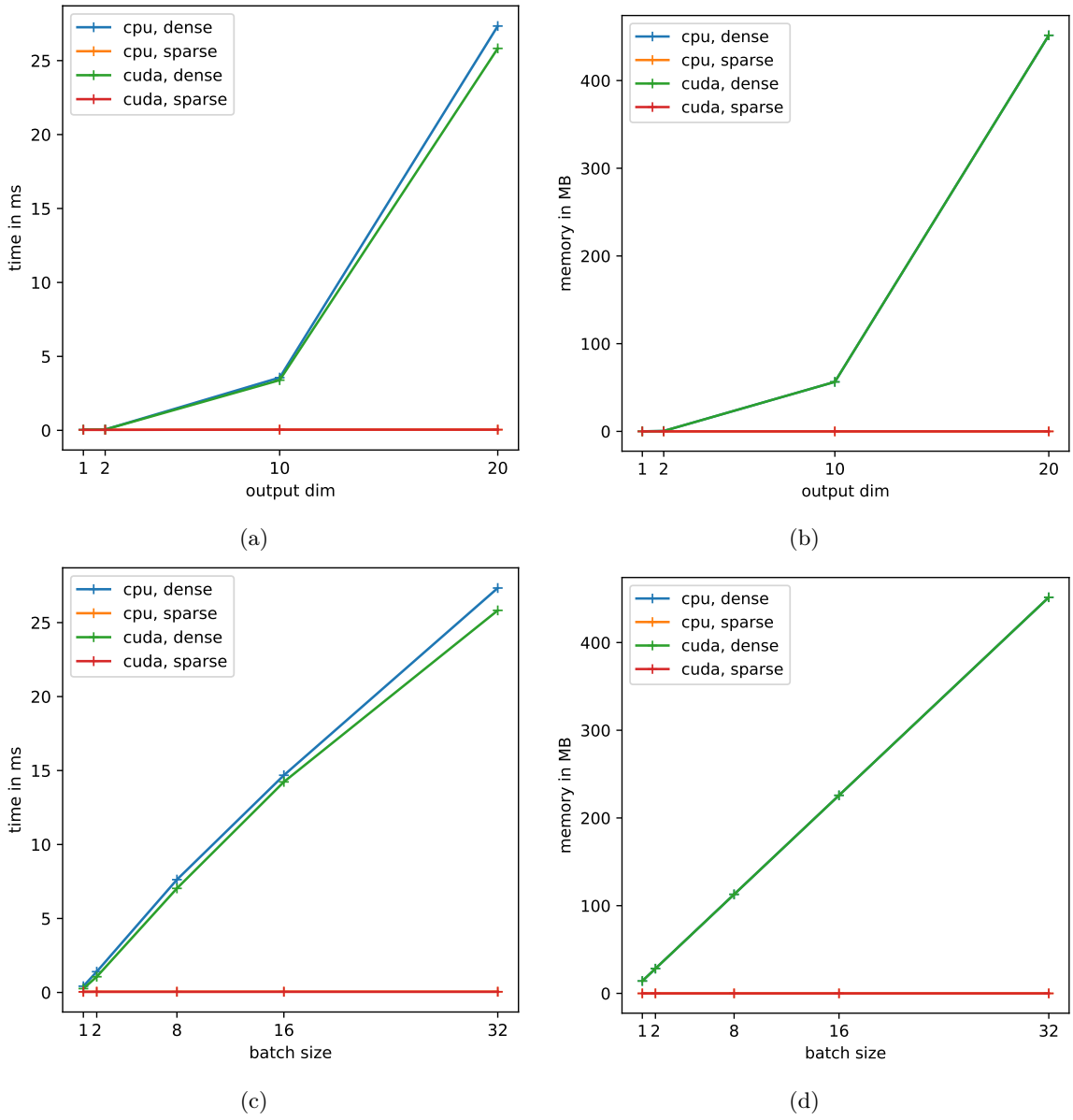


Figure 21: Time and memory usage of generating the dense/sparse OPH tensors of a `O2Linear` layer. In the sparse case, these are close to zero, since the OPH is a zero matrix.

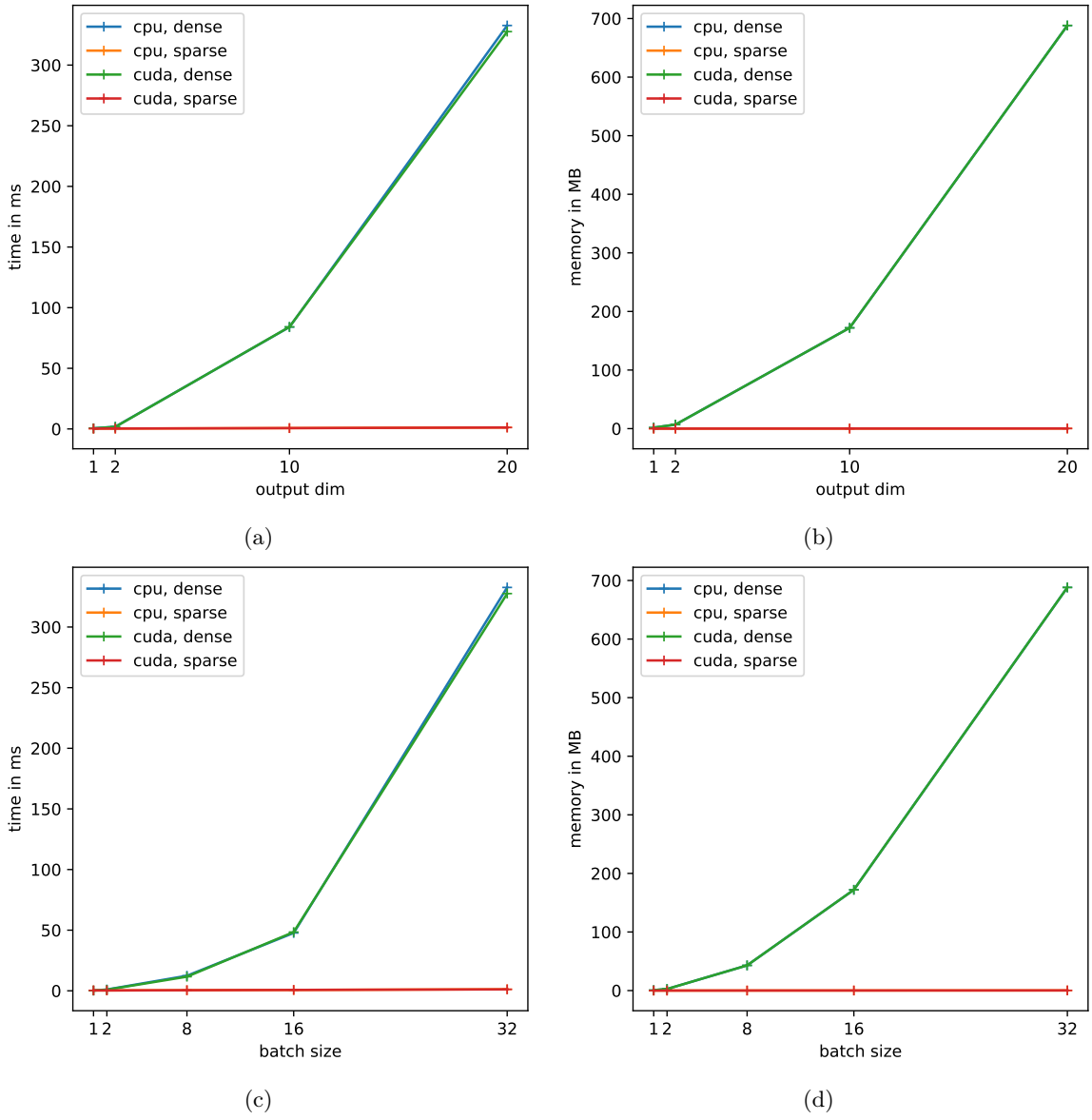
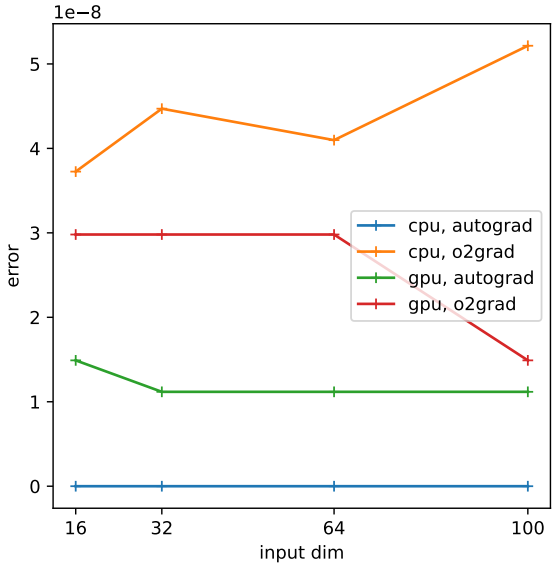
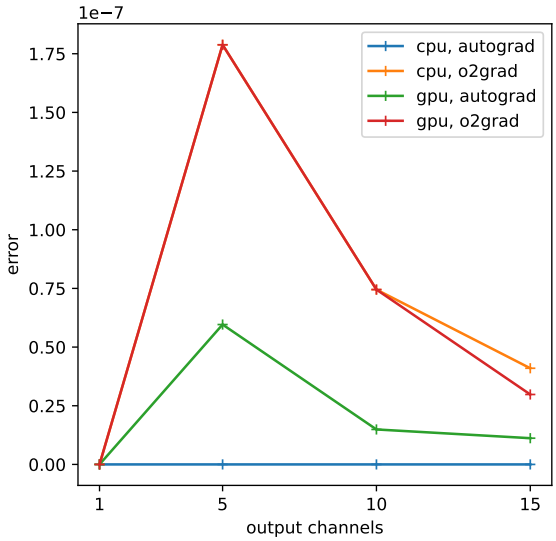


Figure 22: Time and memory usage of generating the dense/sparse mOPH tensors of a `O2Linear` layer.

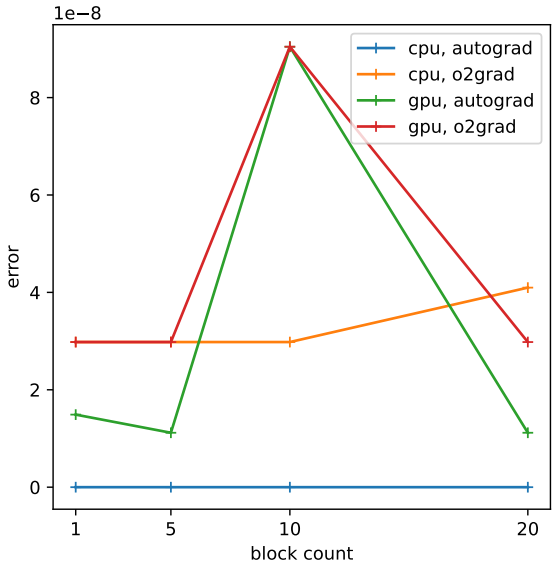
C.1.2 1D CNN



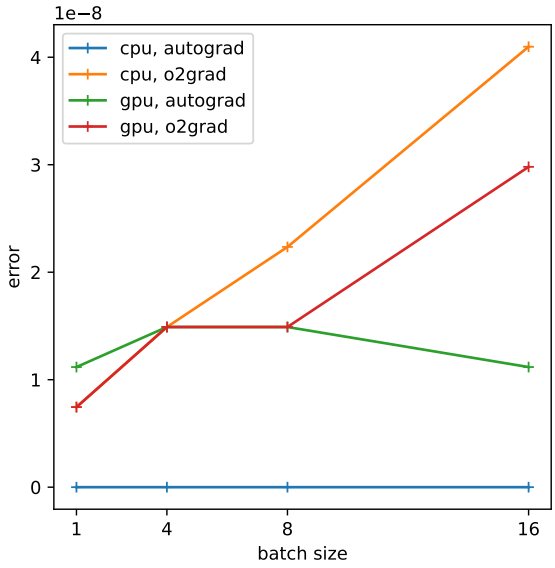
(a) 1D CNN: input size - error



(b) 1D CNN: channels - error



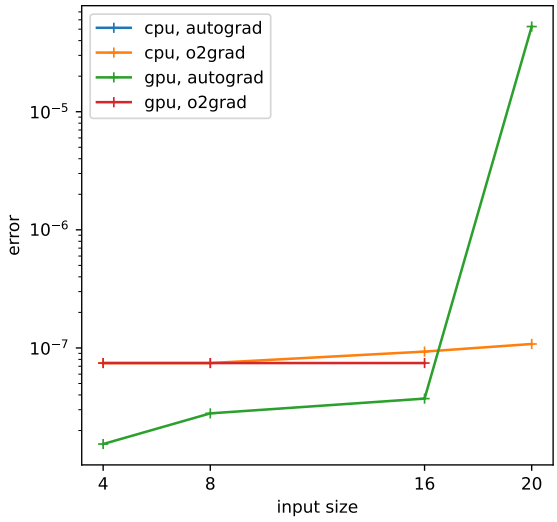
(c) 1D CNN: block count - error



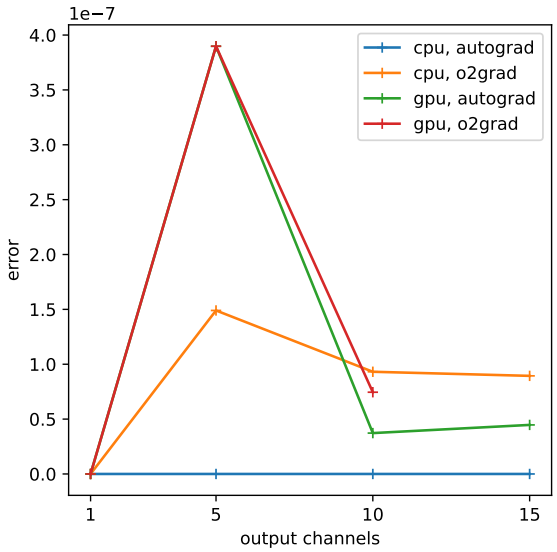
(d) 1D CNN: batch size - error

Figure 23: Absolute maximum error of calculating the Hessian of a 1D CNN, as a function of (a) input size, (b) output channels, (c) block count and (d) batch size, for a fix parameter baseline of (`input_size` = 64, `output_channels` = 15, `block_cnt` = 20, `batch_size` = 16. The parameters that are not subject to variation in the respective graphic correspond to the baseline. All data points are averaged over 3 measurements.

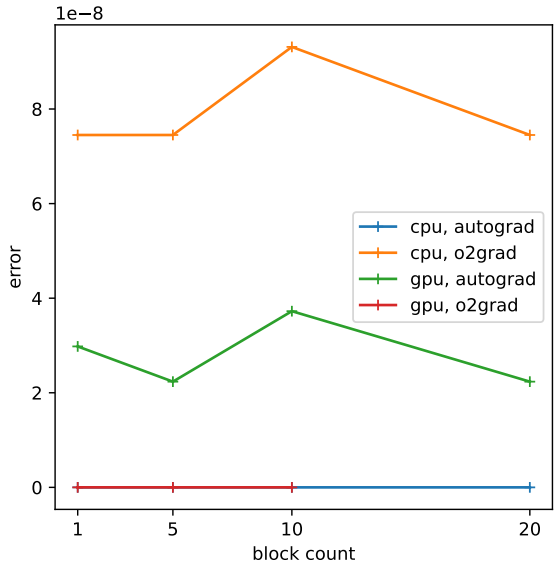
C.1.3 2D CNN



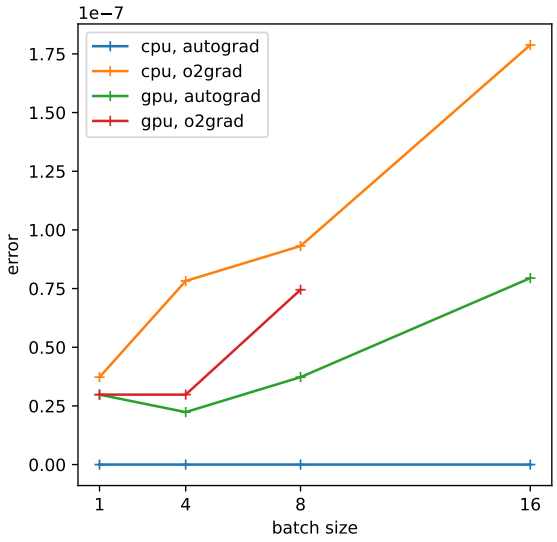
(a) 2D CNN: input dimension - error



(b) 2D CNN: output channels - error



(c) 2D CNN: block count - error



(d) 2D CNN: batch size - error

Figure 24: Absolute maximum error of calculating the Hessian of a 2D CNN, as a function of (a) input size, (b) output channels, (c) block count and (d) batch size, for a fix parameter baseline of (`input_size` = 16, `output_channels` = 10, `block_cnt` = 10, `batch_size` = 8. The reference Hessian is the one calculated using the autograd method on CPU.