GRADO EN INGENIERÍA MATEMÁTICA E INTELIGENCIA ARTIFICIAL

TRABAJO DE FINAL DE GRADO

APPLICATION OF TENSOR ALGEBRA TO NEURAL NETWORK HIGH-ORDER AUTOMATIC DIFFERENTIATION

Author: Miguel Montes Lorenzo
Director: David Alfaya Sánchez
Co-Director: Jaime Pizarroso Gonzalo

Madrid 2025

Contents

Abstract—**Automatic differentiation systems, widely used in machine learning, provide efficient first-order derivatives but lack a native mechanism for higher-order derivatives beyond nested applications of first-order rules. This nesting incurs large memory overhead, limited scalability, and incompatibility with non-sequential graphs. In this thesis, we present THOAD (Torch High-Order Automatic Differentiation), a PyTorch-compatible package implementing a tensor-algebraic approach to higher-order AD. The system reformulates the multivariable chain rule as an iterative symbolic contraction procedure, enabling propagation of arbitrary-order derivatives across unbalanced computational graphs. Optimizations based on batch unification and blockwise Schwarz symmetries reduce complexity significantly in batch size and derivative order. Benchmarks demonstrate substantial improvements in scalability over PyTorch's nested gradient evaluation. THOAD provides researchers and practitioners with an accessible, efficient, and extensible platform for higher-order derivative computations in modern deep learning.**

1 Introduction

1.1 Context and motivation

Automatic differentiation (AD) augments a numerical program with derivative logic so that exact derivatives, to machine precision, are obtained without symbolic manipulation or finite differencing. The core idea is to represent the computation as a directed acyclic graph: nodes store intermediate values, and edges encode algebraic dependencies. Traversing this graph in the same order as evaluation (forward mode) or in reverse (reverse mode, usually called backpropagation) yields first-order directional derivatives or full gradients with only a small constant overhead relative to the primal run time.

In machine-learning workloads, reverse-mode AD is indispensable because loss functions are scalars while the parameters to be optimized are high-dimensional tensors. More fundamentally, typical gradient-based optimization consumes only first-order gradients, not hessians or higher-order derivatives. Reverse-mode aligns exactly with this requirement by computing all parameter gradients from a scalar loss in a single reverse pass. As a result, the literature and engineering effort around AD has focused primarily on the efficient production of first-order derivatives.

Higher-order AD computation of Hessians, Jacobians, or their vector products has received far less attention. Mainstream libraries like (1) (2) (3) expose such quantities only by nesting the first-order AD process, what multiplies memory use and complicates graph scheduling. No widely adopted framework currently ships a purpose-built higher-order engine that remains compatible with arbitrary non-sequencial graphs.

Yet higher-order derivative information is not without value. It finds application in neural network sensitivity studies (4) (5) (6) and interpretability (7) enables genuine second-order or natural-gradient optimization ([8]), and is essential for physics-informed neural networks ([9]) ([10]) ([11]) that enforce differential constraints of order two or higher. A systematic study of native higher-order AD on nonlinear computational graphs therefore promises practical benefits in transparency, convergence speed, and physics-consistent learning.

1.2 Project contributions

The main contributions of this work can be summarized as follows:

- **Iterative method to obtain closed-form expressions for the derivatives of compositions of vector-valued functions:** We propose a contraction-based reformulation of the multivariable chain rule, equivalent to the Faà di Bruno formula ([12]), but expressed directly in tensorial form. This method enables the systematic construction of higher-order derivatives through successive symbolic derivations, producing compact closed-form expressions without resorting to brute-force expansion.
- **Procedure to propagate derivatives through a vectorial computational graph without requiring the graph to be sequential:** Within the literature on higher-order automatic differentiation systems, only functional proposals restricted to fully sequential graphs have been considered. Although any computational graph can be re-expressed as sequential through the introduction of fictitious variables, operating directly on the original graph provides a number of advantages.
- **Strategy to optimize the size and structure of derivatives by eliminating batch dimensions:** Batch dimensions are frequently present in the construction of modern artificial neural networks. Since they are processed independently, they induce specific properties in the internal structure of the network derivatives that allow them to be unified, thereby avoiding unnecessary allocation of elements and significantly reducing the computational cost of automatic differentiation.
- **Strategy to avoid redundant symmetric computations within the proposed backpropagation procedure:** A higher-order derivative of the composition of multivariable operators satisfying Schwarz's theorem exhibits a growing number of internal symmetries as the differentiation order increases. These symmetries can be exploited to avoid the explicit computation of redundant elements.

2 Present Research Context

2.1 Mathematical theoretical framework

Existing academic publications treat multivariable higher-order composition relying on the multivariable Faà di Bruno formula ([12]), a closed-form alternative to the brute-force repeated derivation of the chain rule. For a scalar output $f : \mathbb{R}^p \to \mathbb{R}$ composed with a vector-valued mapping $g : \mathbb{R}^m \to \mathbb{R}^p$, the n-th total derivative can be written in partition notation as

$$\frac{\partial^n}{\partial x_1 \dots x_n} f(g(x)) = \\ = \sum_{\pi \in \Pi_n} f^{(|\pi|)}(g(x)) \prod_{B \in \pi} \frac{\partial^{|B|} g}{\prod_{j \in B} \partial x_j}(x) \quad (2.1)$$

where Π_n denotes the set of all partitions of the index set $\{1, \dots, n\}$ and B runs over the blocks of each partition. While exact and elegant, this formulation does not allow one to leverage the benefits of parallel computation. Derivatives are expressed component-wise, and the combinatorial explosion of partitions forces any implementation to operate element by element. Consequently, adopting Faà di Bruno in practical software usually sacrifices the algebraic structure of vector spaces and forecloses the tensor-level optimizations that underpin efficient first-order systems.

2.2 Implementations

Mainstream frameworks equiped with automatic differentiation tools such as `PyTorch` [1] and `TensorFlow` [2] compute higher-order gradients by nesting their first-order engines. Forward-over-reverse or reverse-over-reverse nesting is straightforward to code but incurs multiplicative memory overhead, repeated graph traversals, and lack robustness confronted with non-differentiable control flow or stateful operators. Crucially, nested evaluation treats each derivative order as an independent pass rather than a unified algebraic object, preventing fusion across derivative orders and degrading parallel scalability.

Other existing alternative projects pursue closed composition rules. Modules like `jax.experimental.jet` [3] [13] implement higher-order pushforwards and pullbacks that mirror Faà di Bruno in code; however, they assume a linear (or univariate) computational graph in which all operations depend on a single tensor variable. Branching graphs violate those assumptions, limiting the approach to a narrow class of models - single input and fully sequential. Collectively, existing systems therefore either incur prohibitive cost through nesting or impose restrictive graph topologies through specialized composition logic-shortcomings that motivate the present work's search for a tensor-centric, order-agnostic alternative.

2.3 Applications

Higher-order partial derivatives of computational graphs have become indispensable in model interpretability. The NeuralSense package [4], for instance, leverages both Jacobian and Hessian tensors to quantify input-output sensitivities, enabling the computation of statistical measures such as local variation, curvature-based feature importance, and interaction effects. In a complementary direction, the study Understanding Black-box Predictions via Influence Functions [5] employs second-order derivative information to estimate how infinitesimal perturbations of individual training points affect model outputs, thereby attributing predictions to specific examples and uncovering hidden biases. Building on these ideas for structured data, Higher-Order Explanations of Graph Neural Networks via Relevant Walks [6] utilizes higher-order derivatives along computational paths to identify the most influential subgraphs and to generate walk-based explanations that capture complex relational dependencies in graph-structured domains.

Beyond interpretability, higher-order derivatives also enable rigorous certification of network properties. In [7], second-order information is used to bound the Lipschitz constants of partial derivatives and to certify partial monotonicity of already-trained networks over regions of the input space. This Lipschitz-based certification provides verifiable ordering guarantees and supports ethical-AI objectives by allowing safety conditions to be checked without imposing architectural constraints.

In physics-informed neural networks (PINNs), higher-order partial derivatives are central to embedding differential equations into the training objective. [9] incorporates residuals of governing equations, often involving second-order spatial or temporal derivatives-directly into the loss functional. Subsequent surveys such as [10] [11] further illustrate the necessity of accurately computing Hessian and higher-order Jacobian-vector products to enforce boundary conditions, satisfy variational integral constraints, and improve convergence and stability in complex physical systems.

Second-order derivative information likewise enhances optimization methods for deep models. In [8], efficient Hessian-vector products computed through automatic differentiation enable conjugate gradient updates without explicitly forming or storing the full Hessian matrix. By leveraging directional second-order information, this approach accelerates convergence, mitigates ill-conditioning, and scales second-order techniques to high-dimensional parameter spaces where naive implementations would be intractable.

Finally, higher-order partial derivatives inform the design of advanced cost functions and regularization strategies. The classic Optimal Brain Damage [14] method uses the diagonal of the Hessian matrix as a sensitivity metric to prune redundant parameters, producing compact models with minimal performance loss. Similarly, Sobolev Training [15] extends the loss to include derivative-matching terms, penalizing mismatches not only in output values but also in their gradients, thereby aligning the model's differential response with that of the target function and improving generalization.

3 Vector Approach to Multivariable Function Composition Derivative

3.1 Setting and notation

a) Vectorial set-up.

Let E_X, E_Y be finite-dimensional real vector spaces. We consider a smooth map

$$F_{Y,X} : E_X \longrightarrow E_Y.$$

If $E_X \cong \mathbb{R}^{N_X}$ we denote components of vectors as $\mathbf{v}^{\mathbf{X}} = (v_1^X, \dots, v_{N_X}^X)$.

b) First-order derivative.

For each $x \in E_X$, the pointwise (Fréchet) derivative is a linear operator

$$D^1 F_{Y,X}(x) \in \operatorname{Lin}(E_X, E_Y),$$
$$F(\mathbf{x}+\mathbf{h}) = F(\mathbf{x}) + D^1 F_{Y,X}(\mathbf{x})[\mathbf{h}] + o(\|\mathbf{h}\|).$$

The *symbolic* first derivative is the smooth map that assigns to every $x \in E_X$ its pointwise derivative:

$$D^1 F_{Y,X} : E_X \longrightarrow \operatorname{Lin}(E_X, E_Y).$$

In finite dimensions $\operatorname{Lin}(E_X, E_Y) \cong E_Y \otimes E_X^*$, hence

$$D^1 F_{Y,X} \in C^\infty\big(E_X,\ E_Y \otimes E_X^*\big).$$

That is, $D^1 F_{Y,X}$ (symbolic) takes $\mathbf{x} \in E_X$ and returns $D^1 F_{Y,X}(\mathbf{x})$ (pointwise), which is linear in the increment $h \in E_X$:

$$h \longmapsto D^1 F_{Y,X}(\mathbf{x})[\mathbf{h}] \in E_Y.$$

c) Order-m derivatives.

Likewise, for $m \geq 2$ and each $\mathbf{x} \in E_X$,

$$D^m F_{Y,X}(\mathbf{x}) \in \operatorname{Mult}^m(E_X; E_Y),$$

i.e., an *m-linear* map in the m increments $h_1, \dots, h_m \in E_X$:

$$(\mathbf{h_1}, \dots, \mathbf{h_m}) \longmapsto D^m F_{Y,X}(\mathbf{x})[\mathbf{h_1}, \dots, \mathbf{h_m}] \in E_Y.$$

Via the canonical identification $\operatorname{Mult}^m(E_X; E_Y) \cong E_Y \otimes (E_X^*)^{\otimes m}$, we obtain the global (symbolic) tensor form in (3.5):

$$D^m F_{Y,X} \in C^\infty\big(E_X,\ E_Y \otimes (E_X^*)^{\otimes m}\big).$$

To simplify further development, we group all derivatives up to order k as

$$\overset{\oplus k}{\mathbb{E}}_{X,Y} := \bigoplus_{m=1}^{k} C^\infty\big(E_X,\ E_Y \otimes (E_X^*)^{\otimes m}\big), \tag{3.1}$$

3.2 Algorithm Proposition

a) From the chain rule to higher order (component view).

For components v_i^C of $F_{A,C}$ and v_j^A of E_A, the first- and second-order chain rules read (3.2).

$$\frac{\partial v_i^C}{\partial v_j^A} = \sum_{h=1}^{N_B} \frac{\partial v_i^C}{\partial v_h^B}\frac{\partial v_h^B}{\partial v_j^A}$$

$$\frac{\partial^2 v_i^C}{\partial v_j^A\,\partial v_k^A} = \frac{\partial}{\partial v_k^A}\left(\frac{\partial v_i^C}{\partial v_j^A}\right) =$$

$$= \sum_{h,t=1}^{N_B} \frac{\partial^2 v_i^C}{\partial v_h^B\,\partial v_t^B}\frac{\partial v_h^B}{\partial v_j^A}\frac{\partial v_\ell^B}{\partial v_k^A} + \sum_{h=1}^{N_B}\frac{\partial v_i^C}{\partial v_h^B}\frac{\partial^2 v_h^B}{\partial v_j^A\,\partial v_k^A} \tag{3.2}$$

By induction one obtains the general (Faà di Bruno–type) pattern. The tensorial method below recasts this iteration directly at the level of differentials, avoiding scalar expansion.

b) Vectorial composition notation.

Let E_A, E_B, E_C be finite-dimensional real vector spaces. Consider smooth maps

$$F_{A,B} : E_A \to E_B,$$
$$F_{B,C} : E_B \to E_C,$$
$$F_{A,C} \;=\; F_{B,C}\circ F_{A,B} : E_A \to E_C.$$

When convenient, we write

$$\mathbf{v}^{\mathbf{A}} \in E_A,$$
$$\mathbf{v}^{\mathbf{B}} = F_{A,B}(\mathbf{v}^{\mathbf{A}}) \in E_B,$$
$$\mathbf{v}^{\mathbf{C}} = F_{B,C}(\mathbf{v}^{\mathbf{B}}) \in E_C.$$

c) Notation used by the algorithm (symbols and types).

We fix an integer $k \geq 1$ (maximal derivative order). For each $m \geq 1$:

$$D^m F_{B,C} \;\in\; C^\infty\big(E_B,\; E_C \otimes (E_B^*)^{\otimes m}\big),$$
$$D^m F_{A,B} \;\in\; C^\infty\big(E_A,\; E_B \otimes (E_A^*)^{\otimes m}\big).$$

Ordered tuples and degree. For $m \geq 1$, an *ordered internal tuple* is $R = (r_1, \dots, r_m)$ with $r_t \in \mathbb{N}_{\geq 1}$. We write $|R| := \sum_{t=1}^m r_t$. The algorithm preserves order in R (no permutations). We use the shorthand $(1, R) := (1, r_1, \dots, r_m)$ and

$$R^{(t)} := (r_1, \dots, r_{t-1},\, r_t + 1,\, r_{t+1}, \dots, r_m) \qquad (1 \leq t \leq m).$$

Symbolic monomials. A *monomial* is the formal tensor product

$$M(m; R) \;:=\; D^m F_{B,C} \otimes D^{r_1} F_{A,B} \otimes \cdots \otimes D^{r_m} F_{A,B}.$$

At this stage $M(m; R)$ is *symbolic*: it specifies which differentials participate and in what order, but *no contraction has been applied yet.*

Evaluation and contraction. Let $a \in E_A$. First we *precompose* the external differential with $F_{A,B}$, so that all factors depend on a:

$$D^m F_{B,C} \circ F_{A,B} \;\in\; C^\infty\big(E_A,\; E_C \otimes (E_B^*)^{\otimes m}\big).$$

Then we apply the natural contraction of E_B/E_B^* dimensions with the ordered tuple of internal differentials. We denote this by the evaluation map

$$\mathrm{ev}_{F_{A,B}} :\; M(m; R) \longmapsto$$
$$\longmapsto \mathcal{C}_B[m; R]\big(D^m F_{B,C}\circ F_{A,B},\, \dots, D^{r_j} F_{A,B},\, \dots\big) \qquad (1 \leq j \leq m)\,,$$

which yields a tensor of type

$$\mathrm{ev}_{F_{A,B}}(M(m; R)) \;\in\; C^\infty\big(E_A,\; E_C \otimes (E_A^*)^{\otimes |R|}\big).$$

Here $\mathcal{C}_B[m; R]$ is the E_B-contraction that *respects the order* in R.

Symbolic polynomials and multiplicities. At iteration s ($1 \leq s \leq k$) we maintain a finite symbolic sum

$$\mathsf{T}_s \;=\; \sum_{(m,R)} c^{(s)}_{m,R}\, M(m; R), \qquad c^{(s)}_{m,R} \in \mathbb{N},$$

where $c^{(s)}_{m,R}$ counts the multiplicity of the monomial $M(m; R)$ produced so far. Only at the end we apply evaluation:

$$\mathrm{ev}_{F_{A,B}}(\mathsf{T}_k) \;=\; D^k F_{A,C} \;\in\; C^\infty\big(E_A,\; E_C \otimes (E_A^*)^{\otimes k}\big).$$

Elementary derivations. Let $J := D^1 F_{A,B}$. The total differential D acting on $\mathrm{ev}_{F_{A,B}}(M(m;R))$ (Leibniz) produces:

external derivation:

$$M(m;R) \;\mapsto\; M\big(m+1;\,(1,R)\big),$$

internal derivations:

$$M(m;R) \;\mapsto\; M\big(m;\,R^{(t)}\big) \quad (1 \le t \le m).$$

These are exactly the two generators used in the iteration.

Aggregation map. We will aggregate repeated monomials by a map

$$\mathrm{count}:\ \{M(m;R)\} \longrightarrow \mathbb{N},$$
$$\mathrm{key}(M(m;R)) := (m,R),$$

so that the next symbolic state is

$$\mathsf{T}_{s+1} \;=\; \sum_{(m,R)} \mathrm{count}_{s+1}(m,R)\, M(m;R).$$

d) algorithm description.

The algorithm we propose (1) employs an iterative process equivalent to that of (3.2), but designed from a vector variable approach instead of a component one.

It is important to note that, although the algorithm is iterative, the iteration takes place at the symbolic level and is therefore computationally negligible in practice, especially when compared to the subsequent evaluation of the resulting expressions on actual numerical tensors.

e) resulting vector expressions

The vector expressions equivalent to those in (3.2) resulting from the application of the procedure described in (1) would be the following:

Algorithm 1: Iterative construction of $D^k(F_{A,C})$ via ordered-tuples contraction (with explicit aggregation)

```
Input:
$k \ge 1$ (derivative order)
$\{D^m F_{A,B}\}_{m=1}^{k}$ (external derivatives)
$\{D^r F_{B,C}\}_{r=1}^{k}$ (internal derivatives)
Output: $\mathsf{T}_k = D^k(F_{A,C}) \in$
        $C^\infty\big(E_C,\ E_A \otimes (E_C^*)^{\otimes k}\big)$

Data model (terms as monomials).
$J \leftarrow D^1 F_{B,C}$;
$m \in \mathbb{N}$;
$R = (r_1,\dots,r_m)$ with $r_t \in \mathbb{N}_{\ge 1}$;
$M(m;R) :=$
  $D^m F_{A,B} \otimes D^{r_1} F_{B,C} \otimes \cdots \otimes D^{r_m} F_{B,C}$;
$\mathsf{T} = \sum c_{m,R}\, M(m;R)$.

Initialization ($k = 1$).
$\mathsf{T}_1 \;\leftarrow\; M(1;(1))$;   (semantics:
 $\mathrm{ev}_{F_{B,C}}(\mathsf{T}_1) = DF_{A,C}$)

Iterative step.
for $s \leftarrow 1$ to $k-1$ do
    /* Generate raw terms via Leibniz rule on monomials */
    raw_terms ← [ ]
    for $M(m;R) \in \mathsf{T}_s$ with $R = (r_1,\dots,r_m)$ do
        append $M(m+1;(1,r_1,\dots,r_m))$ to raw_terms
        for $t \leftarrow 1$ to $m$ do
            $R^{(t)} \leftarrow (r_1,\dots,r_{t-1},\ r_t + 1,\ r_{t+1},\dots,r_m)$
            append $M(m;R^{(t)})$ to raw_terms
    /* Aggregate identical monomials (multiset counting) */
    dict ← empty map $(m,R) \mapsto \mathbb{N}$
    foreach $M(m;R) \in$ raw_terms do
        dict[$(m,R)$] ← dict.get($m,R,0$) + 1
    $\mathsf{T}_{s+1} \leftarrow 0$
    foreach $(m,R)$ in keys(dict) do
        $\mathsf{T}_{s+1} \leftarrow \mathsf{T}_{s+1} + \mathrm{dict}[(m,R)] \cdot M(m;R)$

return $\mathrm{ev}_{F_{B,C}}(\mathsf{T}_k)$
```

$$\begin{aligned} D^1F_{A,C} &\in C^\infty\big(E_A,\ E_C\otimes E_A^*\big)\\ &= \big(D^1F_{B,C}\circ F_{A,B}\big)\ {}_{E_B^*}\!\otimes_{E_B}\ D^1F_{A,B}\\ D^2F_{A,C} &\in C^\infty\big(E_A,\ E_C\otimes E_A^*\otimes E_A^*\big)\\ &= \big(D^2F_{B,C}\circ F_{A,B}\big) \bigotimes_{E_B^*,\,E_B^*} \begin{matrix}\otimes_{E_B}\ D^1F^{(1)}_{A,B}\\ \otimes_{E_B}\ D^1F^{(2)}_{A,B}\end{matrix} +\\ &\quad + \big(D^1F_{B,C}\circ F_{A,B}\big)\ {}_{E_B^*}\!\otimes_{E_B}\ D^2F_{A,B}\,. \end{aligned}\tag{3.3}$$

Notation. On the left of $\otimes$ stands the *external* derivative $D^mF_{B,C}\circ F_{A,B}$, a field on E_A with values in

$$E_C\ \otimes\ \underbrace{E_B^*\ \otimes\ \cdots\ \otimes\ E_B^*}_{m\text{ copies}}.$$

On the right, written vertically, appear the *internal* derivatives $D^rF_{A,B}$, each taking values in

$$E_B\ \otimes\ \underbrace{E_A^*\ \otimes\ \cdots\ \otimes\ E_A^*}_{r\text{ copies}}.$$

The labeled $\otimes$ indicates that the tensor dimensions E_B^* on the left are paired with the corresponding tensor dimensions E_B on the right, in the displayed order (top–bottom on the right matches left–right on the left). The superscripts $(1),(2)$ on $D^1F_{A,B}$ only label the first and second internal derivatives used in these pairings. After pairing, the remaining factors are

$$E_C\ \otimes\ (E_A^*)^{\otimes k},$$

which is the type of $D^kF_{A,C}$.

Why compose. Since $F_{A,C}=F_{B,C}\circ F_{A,B}:E_A\to E_C$, we need a tensor field over E_A. For $a\in E_A$, set $b=F_{A,B}(a)\in E_B$ and evaluate

$$\begin{aligned}\big(D^mF_{B,C}\circ F_{A,B}\big)(\mathbf{a}) &= D^mF_{B,C}\big(F_{A,B}(\mathbf{a})\big)\\ &\in\ E_C\otimes(E_B^*)^{\otimes m}.\end{aligned}$$

This places the external object over E_A so that its E_B^* factors can be paired with the E_B factors coming from $D^rF_{A,B}(a)$, yielding the desired tensor in $E_C\otimes(E_A^*)^{\otimes k}$.

3.3 Proof of Correctness

a) Setting and notation.

Let

$$\begin{aligned} F_{A,B}&:E_A\to E_B\\ F_{B,C}&:E_B\to E_C\\ F_{A,C}=F_{B,C}\circ F_{A,B}&\ :\ E_A\to E_C \end{aligned}\tag{3.4}$$

For each $m\ge 1$, the m-th differential of $F_{Y,X}$ is a smooth map

$$D^mF_{Y,X}\ \in\ C^\infty\big(E_X,\ E_Y\otimes(E_X^*)^{\otimes m}\big).\tag{3.5}$$

We group all derivatives up to order k as

$$\begin{aligned} \overset{\oplus k}{\mathbb{E}}_{C,B} &:= \bigoplus_{m=1}^{k} C^\infty\big(E_B,\ E_C\otimes(E_B^*)^{\otimes m}\big),\\ \overset{\oplus k}{\mathbb{E}}_{B,A} &:= \bigoplus_{m=1}^{k} C^\infty\big(E_A,\ E_B\otimes(E_A^*)^{\otimes m}\big). \end{aligned}\tag{3.6}$$

The k-th differential of the composition is a k-linear functional over E_A:

$$\begin{aligned} D^k(F_{B,C}\ \circ\ F_{A,B})\ &=\ D^kF_{A,C},\\ D^kF_{A,C}\ &\in\ C^\infty\big(E_A,\ E_C\otimes(E_A^*)^{\otimes k}\big). \end{aligned}\tag{3.7}$$

b) Contraction operator and evaluation.

Define the composition (chain-rule) operator

$$\begin{aligned} \mathcal{C}_B^{(k)}\ :\ \overset{\oplus k}{\mathbb{E}}_{C,B}\times\overset{\oplus k}{\mathbb{E}}_{B,A}&\longrightarrow\\ \longrightarrow C^\infty\big(E_A,\ E_C\otimes(E_A^*)^{\otimes k}\big),& \end{aligned}\tag{3.8}$$

which, for each order m, pairs an *external* derivative $D^mF_{B,C}$ with an *ordered tuple* of *internal* derivatives $(D^{r_1}F_{A,B},\dots,D^{r_m}F_{A,B})$, contracting the E_B^* factors of the external part with the E_B factors of the internal parts, in the written order.

c) Elementary monomials and evaluation map.

For $m \geq 0$ and an ordered tuple $R = (r_1, \dots, r_m)$ with $r_t \geq 1$, set the *symbolic monomial*

$$M(m; R) := D^m F_{B,C} \otimes \otimes D^{r_1} F_{A,B} \otimes \cdots \otimes D^{r_m} F_{A,B}. \quad (3.9)$$

At this level $M(m;R)$ is formal (no contrations aplicadas). Its *evaluation* at $a \in E_A$ is defined by first composing the external term with $F_{A,B}$ and then contracting E_B^* with E_B:

$$\mathrm{ev}_{F_{A,B}}\big(M(m;R)\big)(a) := \mathcal{C}_B\big(D^m F_{B,C} \circ F_{A,B}(a),\ \dots, D^{r_j} F_{A,B}(a), \dots\big) \in E_C \otimes (E_A^*)^{\otimes |R|} \qquad (1 \leq j \leq m).$$

Thus $\mathrm{ev}_{F_{A,B}}(M(m;R)) \in C^\infty(E_A,\ E_C \otimes (E_A^*)^{\otimes |R|})$.

d) Symbolic differentiation rule (Leibniz on evaluation).

Let $J := D^1 F_{A,B}$. For $a \in E_A$, the derivative of the evaluated monomial satisfies

$$D_a\, \mathrm{ev}_{F_{A,B}}\big(M(m;R)\big) = = \mathrm{ev}_{F_{A,B}}\big(M(m+1;\ (1,R))\big) + \sum_{t=1}^{m} \mathrm{ev}_{F_{A,B}}\big(M(m;\ R^{(t)})\big), \quad (3.10)$$

where $(1,R) = (1, r_1, \dots, r_m)$ and $R^{(t)} = (r_1, \dots, r_{t-1}, r_t{+}1, r_{t+1}, \dots, r_m)$. The first term corresponds to differentiating the external factor (which introduces a new ordered copy of J), and the sum corresponds to differentiating one internal factor (raising exactly one entry of R by 1). This is the Leibniz rule applied to the evaluation (a multilinear contraction).

e) Iterative procedure.

Let T_k denote the evaluated output after k steps. Initialization and recursion are

$$\mathsf{T}_1 = \mathcal{C}_B^{(1)}\big(D^1 F_{B,C},\ D^1 F_{A,B}\big) = D^1 F_{A,C}, \quad (3.11)$$

$$\mathsf{T}_{k+1} = D_a\, \mathsf{T}_k, \qquad k \geq 1, \quad (3.12)$$

where D_a acts termwise on $\mathrm{ev}_{F_{A,B}}(M)$ and expands according to (3.10) *without permuting* the ordered tuple R.

f) Proof of correctness.

We claim that for every $k \geq 1$,

$$\mathsf{T}_k = \mathcal{C}_B^{(k)}\Big(\bigoplus_{m \leq k} D^m F_{B,C},\ \bigoplus_{r \leq k} D^r F_{A,B}\Big) = D^k F_{A,C} \in C^\infty\big(E_A,\ E_C \otimes (E_A^*)^{\otimes k}\big). \quad (3.13)$$

Proof. The case $k = 1$ follows from (3.11) and the chain rule: $DF_{A,C} = DF_{B,C} \circ DF_{A,B}$, which matches (3.7) for $k = 1$. Assume (3.13) holds for some $k \geq 1$. Then, by (3.12),

$$\mathsf{T}_{k+1} = D_a\, \mathsf{T}_k = D_a\big(D^k F_{A,C}\big) = D^{k+1} F_{A,C}.$$

Expanding $D_a\, \mathsf{T}_k$ termwise and applying (3.10) shows that T_{k+1} is obtained from T_k by:

1) one external derivation that raises $m \mapsto m+1$ and introduces a new ordered copy of $D^1 F_{A,B}$;
2) all internal derivations that raise exactly one entry of each ordered tuple R by 1.

This is precisely the action encoded by $\mathcal{C}_B^{(k+1)}$ in (3.8). Hence (3.13) holds for $k{+}1$. By induction, the statement follows. □

4 Graph Backward Propagation of Derivatives

4.1 Reinterpretation of the computational graph

A computational graph is an unbalanced directed acyclic graph (DAG) representing a composition of computational operators applied to a set of variables. By *unbalanced* we mean that parallel branches do not necessarily contain the same number of nodes, as illustrated in 4.1 (A).

In the literature on computational graphs, there exist two different approaches regarding the identification of computational elements (variables and operators) with graph elements (nodes and edges). For example, PyTorch designates operators as nodes and variables as edges. In contrast, we adopt the opposite convention, identifying variables as nodes and operators as edges, since this perspective is more natural for our purposes.

As shown in figure 4.1 (B), the graph representation can be simplified to that of a balanced DAG by introducing identity operators and fictitious intermediate variables. This new representation is equivalent to a composition of successive multivariable functions defined over a sequence of variable collections. And by extension, it is also equivalent to a composition of multivariable operators defined over a sequence of vector spaces.

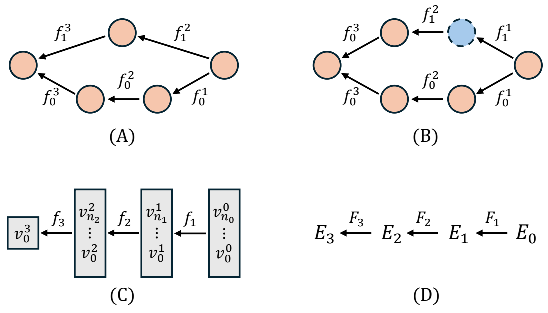


Fig. 4.1: Equivalent computational graph interpretations

4.2 Dependency from propagated nodes derivatives

In the backward pass, executed *after* the forward pass, the computational graph has already been evaluated: every node carries a realized primal value. Hence all derivatives used below are *pointwise* (Fréchet) derivatives evaluated at those values. We write

$$\begin{aligned} a_j &\in E_j, \\ a_{j+1} &= F_{j,j+1}(a_j) \in E_{j+1}, \\ a_N &= F_{j+1,N}(a_{j+1}) \in E_N. \end{aligned}$$

Definition 4.1 (External derivative (pointwise))**.** *For $m \geq 1$, the m-th derivative of the map $F_{j+1,N} : E_{j+1} \to E_N$,* evaluated at the forward value a_{j+1}*, is the tensor*

$$D^m F_{j+1,N}(a_{j+1}) \in E_N \otimes (E_{j+1}^*)^{\otimes m}.$$

This is the pointwise counterpart obtained by applying (3.6) to $F_{j+1,N}$ and then evaluating at a_{j+1}.

Definition 4.2 (Internal derivative (pointwise))**.** *For $m \geq 1$, the m-th derivative of the map $F_{j,j+1} : E_j \to E_{j+1}$,* evaluated at the forward value a_j*, is the tensor*

$$D^m F_{j,j+1}(a_j) \in E_{j+1} \otimes (E_j^*)^{\otimes m}.$$

Again, this is (3.6) applied to $F_{j,j+1}$ and evaluated at a_j.

Grouping the pointwise derivatives up to order k gives the pointwise analogues of (3.6):

$$\begin{aligned} \overset{\oplus k}{\mathbb{E}}_{j+1,N}(a_{j+1}) &:= \bigoplus_{m=1}^{k} \left(E_N \otimes (E_{j+1}^*)^{\otimes m}\right), \\ \overset{\oplus k}{\mathbb{E}}_{j,j+1}(a_j) &:= \bigoplus_{m=1}^{k} \left(E_{j+1} \otimes (E_j^*)^{\otimes m}\right). \end{aligned} \tag{4.1}$$

The k-th derivative of the composition at a_j is

$$\begin{aligned} D^k\big(F_{j+1,N} \circ F_{j,j+1}\big)(a_j) &= D^k F_{j,N}(a_j) \\ &\in E_N \otimes (E_j^*)^{\otimes k}. \end{aligned} \tag{4.2}$$

We use the (pointwise) tensorial chain-rule contraction to combine the evaluated external and internal tensors:

$$\begin{aligned}
\mathcal{C}_j^{(k)}(a_j) : \ & \overset{\oplus k}{\mathbb{E}}_{j+1,N}(a_{j+1}) \times \overset{\oplus k}{\mathbb{E}}_{j,j+1}(a_j) \\
& \longrightarrow E_N \otimes (E_j^*)^{\otimes k}, \\
\mathcal{C}_j^{(k)}(a_j)\Big(& \bigoplus_{m=1}^{k} D^m F_{j+1,N}(a_{j+1}) \ , \\
& \bigoplus_{m=1}^{k} D^m F_{j,j+1}(a_j)\Big) = \\
& = D^k F_{j,N}(a_j).
\end{aligned}$$

In words: at the concrete primal values (a_j, a_{j+1}) coming from the forward pass, the k-th composite derivative is obtained by contracting the E_{j+1}/E_{j+1}^* factors between the pointwise external tensors $D^m F_{j+1,N}(a_{j+1})$ and the ordered tuple of pointwise internal tensors $D^r F_{j,j+1}(a_j)$ whose total order sums to k.

4.3 Block-divided tensor derivatives

Modern Deep Learning frameworks can operate with sequences of one-vector layers, but they are not constraint to that. Neural Networks frequently branches of parallel vector pipelines. In other words, subsets of scalar variables, are not grouped into single vectors, but into multiple ones. This grouping also affects the structure of the output derivatives with respect to each layer, which can be partitioned into independent *blocks*, as shown in figure 4.2. However, this deviation from the presented graph reformulation does not constitute a problem, in as much it can be reached with a simple concatenation of the layer vectors.

When the maximum differentiation order is one, cross-block terms appear. Consequently, the computation of the derivative of a block with respect to a vector in the composition depends only on the external derivative blocks whose vectors are functions of that specific vector. Grouping all scalar variables belonging to the same vector into a node, the derivative with respect to each node

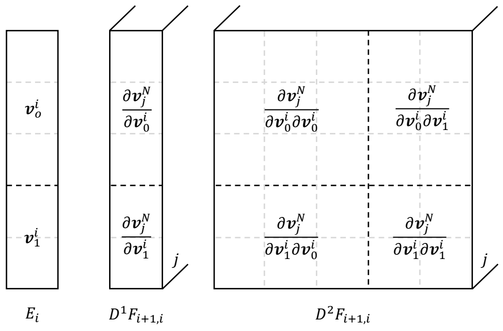

v_0^i
v_1^i
$\frac{\partial v_j^N}{\partial v_0^i}$
$\frac{\partial v_j^N}{\partial v_1^i}$
j
$\frac{\partial v_j^N}{\partial v_0^i \partial v_0^i}$
$\frac{\partial v_j^N}{\partial v_0^i \partial v_1^i}$
$\frac{\partial v_j^N}{\partial v_1^i \partial v_0^i}$
$\frac{\partial v_j^N}{\partial v_1^i \partial v_1^i}$
j
E_i
$D^1 F_{i+1,i}$
$D^2 F_{i+1,i}$

Fig. 4.2: Equivalent computational graph interpretations

depends only on the derivatives with respect to the nodes directly connected to it. Internal derivative blocks from any unconnected node are zero, and no cross-blocks exist. Thus, for first-order derivatives, backpropagation can be carried out node by node without expanding the entire layer and without introducing fictitious nodes. The PyTorch autograd system, for instance, performs a dependency-driven, priority-queued traversal of the backward graph: a node is expanded exactly when all its required gradient contributions have arrived, and among ready nodes, the one with the highest priority (largest forward sequence number) is selected from a per-device queue.

4.4 Propagation procedure

The technique used by PyTorch cannot be extended to derivatives of order greater than one. Addressing this case requires to use the composition procedure presented in section 3. However, there exist two different alternatives regarding the method used to manage derivative blocks:

1) Concatenate vectors of each layer and perform backpropagation on the resulting series graph.
2) Avoid vector concatenation and meticulously manage block combinatorics while

computing iterated tensor products of vector spaces.

In our implementation, we opt for the second approach for several reasons.

1) Derivative tensors, when subdivided into blocks, are asymptotically smaller with respect to the number of blocks.
2) The subdivision enables us to exploit specific properties of external and internal derivatives to avoid repeated allocation of batch dimensions.
3) The subdivision allows us to exploit symmetries among partial derivatives to prevent recomputation of equivalent blocks.

The main drawback of this approach is that it does not provide the possibility to exploit scalar-level symmetries. In any case, doing so would require specialized kernels and would preclude the use of highly optimized tensor contraction operators such as *matmul* or *batched-matmul*.

Propagation through the graph proceeds by sequentially expanding layers of nodes, with each node representing a vector block. Frontier expansion must occur at every step (described at algorithm (2)) across all traversed branches. This requirement causes join nodes to act as blocking gates, passable only after every parent node has connected. Fictitious balancing nodes can be created by preprocessing the graph, however, this requires to go over the graph twice. A better approach is to materialize the nodes on demand simply by completing with identity transformations every node present among composed derivatives and not targeted by any $internal\_fns$.

The expression describing the tensorial derivative of the composition, as established by (1), consists of a summed series of tensor contractions. When derivative-block division is included, computing the full derivative requires

Algorithm 2: Node selection and expansion along the graph

```
Input:
$expanded\_nodes$ (set)
$expanded\_edges$ (set)
Output:
$expanded\_nodes$ (updated)
$expanded\_edges$ (updated)
Initialization:
$active \leftarrow \varnothing$
$candidates \leftarrow \varnothing$
$frontier \leftarrow \varnothing$
Update active nodes:
$deps \leftarrow \varnothing$
foreach $u \in expanded\_nodes$ do
    if $u.edge \neq \varnothing$ and $u.edge \notin expanded\_edges$ then
        $active \leftarrow active \cup \{u\}$
        $deps \leftarrow deps \cup (\text{dependencies}(u) \setminus \{u\})$
$active \leftarrow active \setminus deps$
Collect expansion candidates (edges):
$edges \leftarrow \{\, u.edge \;:\; u \in active \,\}$
foreach $e \in edges$ do
    if $\forall s \in e.sources :\; s \in active$ then
        $candidates \leftarrow candidates \cup \{e\}$
Expand candidates in batch (contractive update):
if $|candidates| > 0$ then
    $internal\_fns \leftarrow \varnothing$
    foreach $e \in candidates$ do
        $internal\_fns[e.fn] \leftarrow (e.sources,\ e.targets)$
        $frontier \leftarrow frontier \cup \{\, t \;:\; t \in e.targets \,\}$
    $composed\_derivatives.\text{update}(internal\_fns)$
    $expanded\_nodes \leftarrow expanded\_nodes \cup frontier$
    $expanded\_edges \leftarrow expanded\_edges \cup candidates$
    $candidates \leftarrow \varnothing$
return $expanded\_nodes,\ expanded\_edges$
```

iterating not only over the summation and contraction terms, but also over the external and the internal blocks. Algorithm (3) presents our pseudo-code for the process.

5 Expression of Tensor Contractions

5.1 Requirement of permutations

Algorithm (1) shows how to compute the derivative of the composition of vector-valued functions. However, modern Deep Learning frameworks, which represent the main application of such derivatives, do not operate directly on vector variables, but more generally on tensor variables. At the same time, these frameworks provide operators that treat each tensor dimension differently. Although every derivative with respect to a tensor variable can be reformulated as a derivative with respect to a vector variable, due to the mentioned heterogeneous dimension treatment, this reformulation is not always computationally convenient.

Batch dimensions, in particular, are treated independently throughout both the evaluation of the graph and the backpropagation of derivatives. For the structure of derivatives, this implies that all cross-elements between repetitions of the same batch dimension across different differentiations are null. This property can be formalized as follows.

Definition 5.1 (Batch independence)**.** *Let V, W be finite-dimensional vector spaces, and let $B \cong \mathbb{R}^n$ denote the batch space with canonical basis $\{e_b\}_{b\in\mathcal{B}}$ and dual basis $\{e_b^*\}_{b\in\mathcal{B}}$. For $m \geq 1$, define the diagonal batch tensor of order m by*

$$\Delta_m^B = \sum_{b\in\mathcal{B}} \mathbf{e}_\mathrm{b} \otimes (\mathbf{e}_\mathrm{b}^*)^{\otimes m} \in B \otimes (B^*)^{\otimes m}. \tag{5.1}$$

Algorithm 3: Block-wise computation of the composite derivative

```
Input:
$block\_counts = (B_{\text{ext}}, B_{\text{int}})$
$composite\_block\_indices$ (tuple int)
$expression$
$external\_differentials$
    $\in \overset{\oplus k}{\mathbb{E}}_{N,j+1}$ divided by external blocks
$internal\_differentials$
    $\in \overset{\oplus k}{\mathbb{E}}_{j+1,j}$ divided by internal blocks
Output:
$result \in \mathbb{E}_{N,j}^{(\leq k)}$ divided by internal blocks
Initialization:
$result \leftarrow 0$
foreach
 $contraction \in expression.contractions$ do
    $order \leftarrow contraction.external.order$
      $E\_block\_indices \leftarrow$
      $set(range(block\_counts[0]))^{order}$
    foreach $position \in E\_positions$ do
        $E \leftarrow external\_differentials.get($
          $E\_block\_indices)$
        if $E$ is None then
            continue
        $Is\_block\_indices \leftarrow$
          $contraction.internals$
        $pairs \leftarrow$
          zip($E\_block\_indices$, $Is\_block\_indices$)
        $Is \leftarrow [\,]$
        $block\_disarrangement \leftarrow [\,]$
        $missing \leftarrow$ false
        foreach
          $(E\_block\_index, I\_block\_indices) \in$
          $pairs$ do
            $key \leftarrow$
              $(E\_block\_index, I\_block\_indices)$
            $I \leftarrow$
              $internal\_differentials.get(key)$
            if $I$ is None then
                $missing \leftarrow$ true; break
            append $I$ to $Is$
            extend $block\_disarrangement$
              with $I\_block\_indices$
        if $missing$ then
            continue
        $block\_permutation \leftarrow$
          $reverse(block\_disarrangement)$
        $term \leftarrow$
          contract($E$, $Is$, $block\_permutation$)

        $result \leftarrow result + term$
return $result$
```

Consider a batched map

$$F : B \otimes V \longrightarrow B \otimes W, \qquad F = \mathrm{Id}_B \otimes \hat{F},$$
$$u = \sum_{b \in \mathcal{B}} \mathbf{e_b} \otimes \mathbf{v_b} \in B \otimes V, \quad (5.2)$$

and take derivatives with respect to the V–components only *(batch coordinates are held fixed). Then, for $m \geq 1$, the m-th* pointwise *derivative at u factorizes as*

$$D_V^m F(u) = \sum_{b \in \mathcal{B}} \mathbf{e_b} \otimes (\mathbf{e_b^*})^{\otimes m} \otimes D^m \hat{F}(\mathbf{v_b})$$
$$\in B \otimes (B^*)^{\otimes m} \otimes W \otimes (V^*)^{\otimes m}. \quad (5.3)$$

In components,

$$\big(D_V^m F(u)\big)_{b,\alpha;\, b_1,\beta_1;\ldots;\, b_m,\beta_m} =$$
$$= \Big(\prod_{j=1}^{m} \delta_b^{b_j}\Big) \big(D^m \hat{F}(\mathbf{v_b})\big)_{\alpha;\, \beta_1,\ldots,\beta_m}.$$

In particular, all cross-components with distinct batch indices vanish, reflecting batch independence.

The computation of higher-order derivatives over tensors with batch dimensions naturally leads to expressions combining dimension permutations, element-wise multiplications, and tensor contractions. Although definition (5.1) shows one single batch dimension at first place, these dimensions may appear in arbitrary number and positions within both external and internal derivatives. Consequently, tensors must first be reorganized by suitable permutations that align the batch dimensions before efficient contractions can be applied.

Once dimensions are aligned, interactions among those that share batch positions can be represented as element-wise multiplications, while the remaining interactions between distinct dimensions are expressed as tensor contractions. This decomposition is essential, as it shows that any derivative propagation in the presence of batch reduces to a sequence of dimension permutations and standard contractions, without the need to introduce additional operators.

5.2 Esinstein sum and notation

The Einstein notation is a well known notation designed to describe multi-linear operators. It identifies element-wise multiplication with indices repeated an arbitrary number of times including output. And it identifies contraction with indices repeated twice omitting output. This notation provides a natural language to describe the previously described operations. By suppressing summation symbols over contracted indices and identifying each index through its repetition, the notation encodes the minimal information required to specify unambiguously a permutation, element-to-element multiplication and contraction. This way, a single formula in Einstein notation can represent what would otherwise require a greater number of steps.

Any combination of permutations, element-wise multiplications, and tensor contractions can conveniently be rewritten as a sequence of permutations and batch matrix multiplications, which belong to the optimized core of nearly all Deep Learning backends. The main challenge, in practice, lies in scheduling the optimal sequence of permutations and matrix multiplications. Determining the order that minimizes the number of operations or the memory cost is, in general, NP-hard ([16]) ([17]), as the search space grows exponentially with the number of indices, making exact solutions intractable for large contractions.

The `einsum` operator in PyTorch provides the ideal interface to express contractions declaratively using Einstein notation. Moreover it can be integrated with external planners such as *opt_einsum* ([18]) which implement heuristic and limited exact algorithms to approximate the optimal contraction sequence. To guarantee maximum efficiency in the computation of derivative expressions for compositions, our implementation performs all tensor product contractions using exclusively this operator.

5.3 Einstein notation for tensorial composition derivative

The structure of the contractions resulting from the iterative procedure in (1) is always that of an external derivative contracted against an arbitrary number of internal derivatives. Within this structure, the specific way in which the dimensions of each of these internal derivatives behave depends on the operators associated with them. Each internal derivative has its own three-part Einstein notation indicating how its dimensions are contracted with the dimensions of the corresponding differentiation (group of dimensions) of the external derivative. This set of Einstein notations is unified into a single one that describes the multiple contraction to be performed, allowing the 'einsum' operator to internally decompose it in whatever way it considers most efficient.

The use of Einstein notation in the composition involves a series of difficulties related to the ordering of the differentiations of the resulting external derivative. Simply contracting each internal derivative with its corresponding differentiation of the incoming external derivative does not guarantee that the ordering of the resulting differentiations (arising from the internal derivatives) will be correct. The differentiations of the external derivatives resulting from multiple contractions generally require a final permutation to reach their expected ordering. Naturally, this final permutation of differentiations is also incorporated into the single 'einsum' operation.

To carry out this final permutation, the dimensions of the internal Einstein notations are separated into their own internal differentiations. In this way, the internal Einstein notations consist of:

1) A first group of indices referencing the dimensions of the external differentiation to be contracted.
2) A second group of indices referencing the dimensions of the internal derivative.
3) A third tuple of groups of indices referencing the dimensions of each of the internal differentiations.

Having this packaging of the resulting dimensions by differentiation makes it possible to easily perform the necessary final permutations.

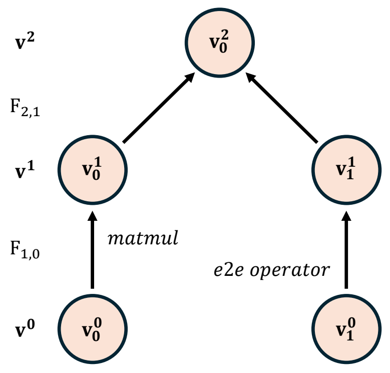


Fig. 5.1: Example of tree-shaped vector computational graph

Take, for example, the graph illustrated in figure 5.1 with the contraction (5.4) of the derivative of the composition $F_{2,1} \circ F_{1,0}$. The unification of the internal einsums into a single notation would be carried out as shown in (5.5).

$$\left(D^2 F_{\mathbf{E}^{\mathbf{1}}_{[\mathbf{0},\mathbf{1}]},\mathbf{E}^{\mathbf{2}}_{[\mathbf{0}]}} \circ F_{E^0_{[0,1]},E^1_{[0,1]}}\right)(\mathbf{v}^0) \underset{E_0^1{}^*,\,E_1^1{}^*}{\bigotimes} \begin{matrix} \otimes_{E_0^1} \\ \otimes_{E_1^1} \end{matrix}$$

$$\underset{E_0^1{}^*,\,E_1^1{}^*}{\bigotimes} \begin{matrix} \otimes_{E_0^1} & D^2 F_{\mathbf{E}^{\mathbf{0}}_{[\mathbf{0},\mathbf{0}]},\mathbf{E}^{\mathbf{1}}_{[\mathbf{0}]}}(\mathbf{v}^0) \\ \otimes_{E_1^1} & D^1 F_{\mathbf{E}^{\mathbf{0}}_{[\mathbf{1}]},\mathbf{E}^{\mathbf{1}}_{[\mathbf{1}]}}(\mathbf{v}^0) \end{matrix} \tag{5.4}$$

Explanation. This displays one contraction term of the second-order composition along the chain $E^0 \to E^1 \to E^2$, evaluated pointwise at $\mathbf{v}^0 \in E^0$. The external factor uses the order-2 derivative of $F_{1,2}$ with respect to the two blocks $E^1_{[0]}$ and $E^1_{[1]}$, mapping into block $E^2_{[0]}$, and is precomposed with $F_{0,1}$ to live over E^0. On the right, the vertically listed internal factors are the derivatives of $F_{0,1}$ taken at $\mathbf{v}^0$ and contracted, respectively, in the $E^1_{[0]}$ and $E^1_{[1]}$ dimensions. Brackets $[\,\cdot\,]$ indicate the block index inside each layer vector.

external indices:

$$[\,\underbrace{[0,1]}_{\text{diff } \mathbf{v}^1_0},\ \underbrace{[\,2\,]}_{\text{diff } \mathbf{v}^1_1}\,]$$

internal einsums:

$$\begin{aligned} &\text{matmul:} && [0,1],\ [3,4] \ \to\ [[3,1],\ [0,4]] \\ &\text{e2e:} && [\,2\,],\ [\,2\,] \ \to\ [[\,2\,]] \end{aligned}$$

unified einsum:

$$\begin{aligned} &[[0,1],\ [2]]\,,\ [3,4],\ [2] \ \to \\ &\to\ [\underbrace{[3,1]}_{\text{diff } \mathbf{v}^2_0},\ \underbrace{[\,2\,]}_{\text{diff } \mathbf{v}^2_1},\ \underbrace{[0,4]}_{\text{diff } \mathbf{v}^2_0}] \end{aligned} \tag{5.5}$$

6 Batch Management and Optimizations

6.1 Batch unification

Batch dimensions, as established in the definition of batch independence (5.1), introduce redundant elements into derivative tensors. Since batch dimensions are pairwise diagonal, they can be unified into a single one by means of an appropriate operator (6.1), thereby reducing the size of the derivatives significantly. As shown in (6.4), this unification decreases the derivative size asymptotically with respect to the batch size.

6.2 Batch unification

Definition 6.1 (Batch unification operator (pointwise form)). *Let $E_j \cong B \otimes V_j$ with batch space $B \cong \mathbb{R}^n$, canonical basis $\{e_b\}_{b\in\mathcal{B}}$ and dual $\{e^*_b\}_{b\in\mathcal{B}}$. For $m \geq 1$, set*

$$\begin{aligned} \Delta^B_m &:= \sum_{b\in\mathcal{B}} e_b \otimes (e^*_b)^{\otimes m} \ \in\ B \otimes (B^*)^{\otimes m}, \\ \mathcal{U}^B_m(x) &:= \big\langle \Delta^B_m,\, x \big\rangle_{(B^*)^{\otimes m}}, \end{aligned} \tag{6.1}$$

so $\mathcal{U}^B_m : B \otimes (B^*)^{\otimes m} \to B$ contracts the m dual batch dimensions against Δ^B_m. Suppose $F_{j,j+1} = \mathrm{Id}_B \otimes \hat{F}_{j,j+1}$ and $F_{j+1,N} = \mathrm{Id}_B \otimes \hat{F}_{j+1,N}$ are batch–wise (element–wise in B). Then, differentiating only with respect to feature variables, for $\mathbf{v}^j \in V_j$ and $\mathbf{v}^{j+1} \in V_{j+1}$ one has the pointwise factorizations

$$\begin{aligned} D^m F_{j,j+1}(\mathbf{v}^j) &= \Delta^B_m \ \otimes\ D^m \hat{F}_{j,j+1}(\mathbf{v}^j) \\ &\in\ B \otimes V_{j+1} \otimes (V^*_j)^{\otimes m}, \\ D^m F_{j+1,N}(\mathbf{v}^{j+1}) &= \Delta^B_m \otimes D^m \hat{F}_{j+1,N}(\mathbf{v}^{j+1}) \\ &\in\ B \otimes V_N \otimes (V^*_{j+1})^{\otimes m}. \end{aligned} \tag{6.2}$$

In components,

$$\begin{aligned} &\big(D^m F(\mathbf{v})\big)_{b,\alpha;\, b_1,\beta_1;\ldots;b_m,\beta_m} = \\ &\qquad = \Big(\prod_{t=1}^{m} \delta^{b_t}_b\Big)\, \big(D^m \hat{F}(\mathbf{v})\big)_{\alpha;\, \beta_1,\ldots,\beta_m}, \end{aligned}$$

so all cross–batch components with distinct indices b_t vanish.

$$\begin{aligned} &\mathcal{C}^{(k)}_j\Big(\bigoplus_{m=1}^{k} \mathcal{U}^B_m\big(D^m F_{j+1,N}(\mathbf{v}^{j+1})\big), \\ &\qquad \bigoplus_{m=1}^{k} \mathcal{U}^B_m\big(D^m F_{j,j+1}(\mathbf{v}^j)\big)\Big) = \\ &\qquad\qquad = \mathcal{U}^B_k\big(D^k F_{j,N}(\mathbf{v}^j)\big), \end{aligned} \tag{6.3}$$

Brief justification. Since both $F_{j,j+1}$ and $F_{j+1,N}$ act element–wise on B, the chain rule and the multilinear contraction $\mathcal{C}^{(k)}_j$ decompose batch–wise: for each $b \in \mathcal{B}$, the b–th slice of the left–hand side equals the (feature–space) composition at batch b. The tensor Δ^B_m enforces equality of the batch indices along the external and internal derivatives, hence applying $\mathcal{U}_m^B$ to each factor and $\mathcal{U}_k^B$ to the result preserves the composition and removes redundant batch duals, yielding (6.3).

Proposition 6.1 (Computational cost reduction via batch unification)**.** *Let $C_X(m)$ denote the cost of contracting the feature–space part (independent of $|B|$), and assume batch dimensions operate element–wise. Then the m–th order composition term has*

$$\begin{aligned} T\big(\mathcal{C}_j^{(m)}\big) &= \Theta\big(|B|^{1+2m}\, C_X(m)\big), \\ T\big(\mathcal{U}^B \circ \mathcal{C}_j^{(m)}\big) &= \Theta\big(|B|\, C_X(m)\big), \end{aligned} \tag{6.4}$$

so the asymptotic reduction factor is

$$\frac{T\big(\mathcal{C}_j^{(m)}\big)}{T\big(\mathcal{U}^B \circ \mathcal{C}_j^{(m)}\big)} = \Theta\big(|B|^{2m}\big). \tag{6.5}$$

Sketch. *Without unification, the external factor contributes one batch dimension and each of the m internal derivatives contributes one dual batch slot; element–wise evaluation over B yields the $|B|^{1+2m}$ factor. After unification, all batch duals are collapsed once, leaving a single batch traversal ($|B|$); the feature–space contraction cost $C_X(m)$ is unchanged.*

6.3 Overview of batch annotation

Unifying batch dimensions in internal derivatives has a drawback. Some operators have primal dimensions in internal derivatives that do not contract with all dual dimensions of the external derivatives. In such cases, batch dimensions from internal derivatives may interact element-wise with external dual dimensions. And if thodse do not satisfy batch independence, the operation requires undoing unification by applying $\mathrm{inv}(\mathcal{U}_m^B)$.

Implementing batch unification during automatic differentiation requires maintaining annotations on the batch-independence condition of all dimensions in each node (assuming nodes represent tensor variables). Our implementation initially assumes that all output tensor dimensions are batch-independent. During backpropagation, annotations are updated dynamically according to the way each operator treats its dimensions. When a dimension transitions from independent to non-independent, the inverse unification operator is applied to all occurrences of that dimension across the external derivatives.

This strategy, however, has a practical limitation. Neural network computational graphs typically terminate in a 0-dimensional tensor (a scalar) representing the loss. Since no output dimension is marked as batch-independent, batch unification cannot be applied, even though the network may have treated some dimension as batch and only reduced it in the final operator. To address this limitation, we adopt a strategy that allows annotating some newly introduced dimensions during backpropagation as batch-independent under specific circumstances.

The key lies in the independent dual dimensions introduced by internal derivatives. If a new dimension is independent of all others within the internal derivatives, then within the composition it satisfies the conditions of a batch-independent dimension. Independence of a new dimension can always be ensured when the maximum nonzero external derivative order is 1 and the dimension is independent of all others introduced by the same internal derivative, since in this case that derivative is the only one involved in the contraction.

Definition 6.2 (Independent dual dimension)**.** *Let V be a finite-dimensional vector space with canonical basis $\{e_i\}_{i\in I}$ and dual $\{e_i^*\}_{i\in I}$. A dual slot of type V^* is an* independent dual dimension *if, for every operator F and every order $m \geq 1$, the derivative tensor* (6.6).

$$D^m F \in C^\infty\big(V,\, W \otimes (V^*)^{\otimes m}\big) \tag{6.6}$$

contains this slot only as repeated factors of the same basis element e_i^*, i.e. (6.7)

$$\left(D^m F\right)_{\alpha;\, i_1,\dots,i_m} = 0 \quad \textit{whenever not all } i_1 = \cdots = i_m = i \tag{6.7}$$

for some fixed $i \in I$. Equivalently, such a dimension never produces cross-components with any other dimension across derivatives of any order. Hence, an independent dual dimension always appears as a single factor in V^*, replicated m times in $D^m F$, with no coupling to other dimensions.

Considering this annotation-based methodology, along with the correction to incorporate dynamically new batch-independent dimensions, the algorithm for batch management during back-propagation can be implemented as in (4).

Algorithm 4: Backpropagation with batch-unification awareness

```
Input:
output_node (graph output node)
k (differentiation order)
graph (computational graph)
Output:
external_derivatives (w.r.t. graph inputs)
Initialization:
foreach dim ∈ output_node.dims do
    dim.batch_independent ← true
external_derivatives ← [$Id$, $0_2$, …, $0_k$]
Iterative step:
foreach layer ∈ graph do
    E ← layer.external_nodes
    I ← layer.internal_nodes
    foreach EN ∈ E do
        foreach IN ∈ I do
            op ← edge(EN, IN)
            foreach dim ∈ IN.dims do
                if
                  ( dim ∉ EN.dims ) ∨ ( dim ∉ op.batch_dims ) then
                    if $k = 1$ then
                        dim.batch_independent ←
                          ( dim ∈ op.independent_duals )
                    else
                        dim.batch_independent ← false
    external_derivatives ←
      desunify_batch(external_derivatives, I)
    internal_derivatives ←
      compute_internal_derivatives(E, I)
    external_derivatives ←
      compute_composite_derivatives(
      external_derivatives, internal_derivatives
      )
return external_derivatives
```

7 Block-Symmetries and Optimization

7.1 Schwarz condition symmetries and potential optimization

The *Schwarz theorem* states that, under suitable regularity assumptions on the operators (in particular, continuity of higher-order derivatives), the order in which partial derivatives are applied does not affect the result. Formally, for any sufficiently smooth function $F : \mathbb{R}^n \to \mathbb{R}$ and for every pair of indices i, j, satisfies equation (7.1).

$$\frac{\partial^2 F}{\partial x_i \partial x_j} = \frac{\partial^2 F}{\partial x_j \partial x_i} \tag{7.1}$$

and more generally, derivatives of order m are symmetric with respect to permutations of their indices

$$D^m F(x)[\mathbf{v}_{\sigma(1)}, \dots, \mathbf{v}_{\sigma(m)}] = = D^m F(x)[\mathbf{v_1}, \dots, \mathbf{v_m}], \quad \forall \sigma \in S_m \tag{7.2}$$

7.2 Block-wise symmetry optimization

where S_m denotes the symmetric group of order m. This property implies that, to compute the full derivative tensor of order m, it is sufficient to evaluate only a fraction of its independent components, as the rest can be obtained by symmetry. The number of independent components is reduced approximately by a factor of $(1/2)^{m-1}$ with respect to the total number of components.

In our framework, derivatives are represented with respect to an explicit vector basis. The operators employed in the implementation (matrices, convolutions, batched matrix multiplications) act in parallel over all tensor elements, without the capability to adapt element-wise computations to exploit symmetries. Consequently, the combinatorial reduction of components derived from the Schwarz theorem cannot be directly applied at the scalar variable level.

Nevertheless, when working with a block partitioning, symmetry can be exploited effectively. Each block corresponds to a vectorial subset of variables jointly processed by a multivariable operator. The Schwarz symmetry ensures that the computation of cross-derivatives between two different blocks is redundant: each ordered pair of blocks needs to be computed only once, with the result reused by symmetry. Thus, instead of reducing the number of scalar elements computed, the method reduces the number of derivative blocks evaluated. The reduction factor remains $(1/2)^{m-1}$, but applied to the number of blocks involved in derivatives of order m.

In theory, the computational cost of evaluating derivatives of maximum order is reduced exactly by this factor. In practice, however, the benefit is partially offset by two considerations: first, the overhead associated with the implementation of block-symmetry deduplication; and second, the fact that for lower-order derivatives the proportion of reduction is smaller, such that the overall weighted saving in the full computation is lower than the theoretical ideal. Despite these limitations, the application of the Schwarz symmetry at the block level provides a significant optimization for the evaluation of higher-order derivatives, while preserving compatibility with standard vectorized operators and without requiring specialized kernels to exploit scalar-level symmetries.

8 Results: THOAD Package

8.1 Presentation of the package

Throughout this project we have developed a package implementing an automatic differentiation system of arbitrary order over a vectorial (or more precisely, tensorial) computational graph, using the techniques discussed throughout this paper. The package is named **THOAD** (Torch High Order Automatic Differentiation), since it delegates to PyTorch both the implementation of tensor operators and the construction of the computational graph. It is fully written in Python 3.12 and relies exclusively on PyTorch 2.2+ as its only dependency. At present, it is available for download via GitHub ([19]) or PyPI.

A main design priority of the package is to provide a user interface that enables seamless integration with PyTorch. Figure 8.2 illustrates how to define a computational graph and run the higher-order automatic differentiation process with PyTorch. Figure 8.1, shows how to do the same with THOAD.

8.2 Package tests

THOAD is compatible with 70 different PyTorch-operator backward functions (internal classes used to compute an operator's internal

```
import torch

# Model setup
X = torch.rand(
      size=(64,10),
      requires_grad=False

)
W1 = torch.rand(
      size=(10,10),
      requires_grad=True
)
W2 = torch.rand(
      size=(10,10),
      requires_grad=True,
)

T = torch.relu(
      input=(X @ W1),
)
T = torch.softmax(
      input=(T @ W2),
      dim=1,
)

# First-order gradient
T.sum().backward()
```

Fig. 8.1: First order back-propagation with PyTorch

derivatives). In practice, this means that it is compatible with many more than 70 PyTorch operators, since most complex PyTorch operators implement their backward process through the composition of simpler backward functions. All implemented backward functions have been tested against PyTorch's first- and second-order derivatives across all their inputs. The structure of the tests performed can be found in the appendix figure 1.

8.3 Impact of optimizations

As mentioned earlier in the report, the two strategies proposed to improve the performance of the automatic differentiation process represent a significant part of the contributions of this work. Consequently, as part of the performance analysis of the package, we have conducted comparisons between the execution of derivative propagation with these strategies enabled and disabled. As shown in figure 8.3, both the batch optimization and the Schwarz optimization yield asymptotical reductions in computational cost

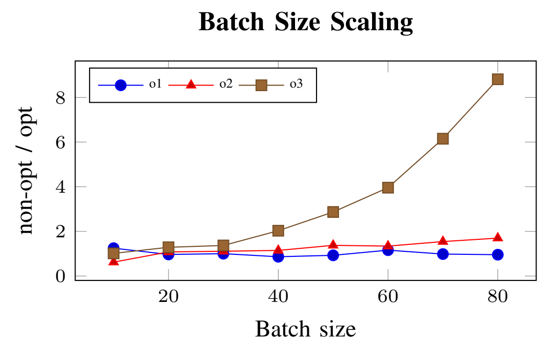
```
import torch
import thoad

# same model setup as above

# - O.sum().backward()
order = 1  # define arbitrary
      differentiation order
thoad.backward(
      tensor=T,  # accepts non-scalar
         tensors
      order=order,
)
```

Fig. 8.2: Back-propagation with thoad

with respect to batch size and derivative order, respectively.



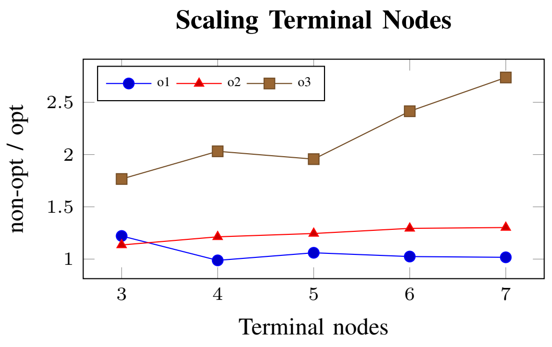


Fig. 8.3: Scalability benchmarks: ratio of non-optimized THOAD runtime to optimized THOAD runtime as a function of batch size (top) and number of terminal nodes (bottom). Benchmark implementations can be found in appendix figures 6 and 7 respectively.

8.4 Benchmarks vs PyTorch autograd

Furthermore, Figure 8.4 presents benchmark results that compare the computational scalability of Hessian computation in THOAD with PyTorch, which performs this operation differently using nested automatic differentiations. As can

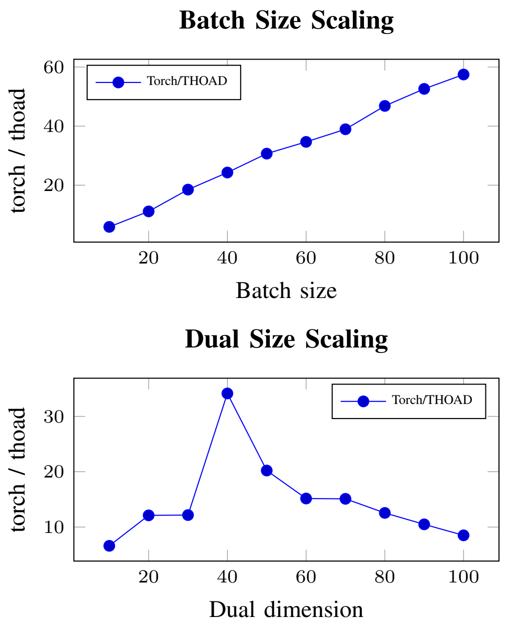


Fig. 8.4: Scalability benchmarks: ratio of PyTorch runtime to THOAD runtime as a function of batch dimension size (top), dual dimension size (bottom). Benchmark implementations can be found in appendix figures 3 and 4 respectively.

be observed, thanks to the optimization techniques discussed, THOAD scales asymptotically better in computational cost with respect to increases in batch dimension size. By contrast, it scales worse w.r.t. parameter dimension size,

9 Conclusion and Future Work

To conclude this report, we present below several ideas that we consider particularly relevant for a potential extension or complement to this work.

- **Exploring the exploitation of derivative symmetries at the element level:** As discussed in Section 7 of this report, taking computational advantage of symmetries at the element level when working with vectorial or tensorial variables is challenging, and may in fact be impossible with the tools provided by high-level vector calculus libraries. Nevertheless, this is a direction to which we have not devoted any research effort, and on which, consequently, we cannot make any categorical statements.
- **Incorporating efficient support for convolutions into the package:** The entire mathematical framework presented in this report relies on the assumption that internal derivatives possess a tensor structure with primal, dual, and batch dimensions. The internal derivative of a convolution can indeed be expressed in this way, but doing so requires combining in that derivative one dimension corresponding to the output feature space with one or more dimensions corresponding to the input feature space, i.e. $(CHW)_{out} \otimes (CHW^*)_{in}^{\otimes m}$. As a result, the resulting tensors become extremely large and highly sparse. Should convolution support be incorporated into the package, it would be advisable to do so in a more optimized manner.
- **Complementing the package with an API for developing and applying optimizers that leverage higher-order derivatives:** Modifying the values of the leaf tensors of the graph according to the values of their gradients is a key process in computational optimization, forming the cornerstone of Deep Learning. In certain subdomains of Deep Learning research, such as Physics-Informed Neural Networks (PINNs), it may be useful to involve derivatives of order higher than one in the optimization process. Providing the package with an API for designing and applying optimizers of arbitrary order would substantially enhance its usability.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” in *NIPS-W*, 2017.
[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.
[3] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: Composable transformations of Python+NumPy programs.” http://github.com/jax-ml/jax, 2018. Version 0.3.13.
[4] J. Pizarroso, J. Portela, and A. Muñoz, “Neuralsens: sensitivity analysis of neural networks,” *arXiv preprint arXiv:2002.11423*, 2020.
[5] P. W. Koh and P. Liang, “Understanding black-box predictions via influence functions,” in *International conference on machine learning*, pp. 1885–1894, PMLR, 2017.
[6] T. Schnake, O. Eberle, J. Lederer, S. Nakajima, K. T. Schütt, K.-R. Müller, and G. Montavon, “Higher-order explanations of graph neural networks via relevant walks,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 11, pp. 7581–7596, 2021.
[7] A. Polo-Molina, D. Alfaya, and J. Portela, “A mathematical certification for positivity conditions in neural networks with applications to partial monotonicity and ethical ai,” *arXiv preprint arXiv:2406.08525*, 2024.
[8] J. Martens *et al.*, “Deep learning via hessian-free optimization.,” in *Icml*, vol. 27, pp. 735–742, 2010.
[9] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations,” *arXiv preprint arXiv:1711.10561*, 2017.
[10] X. W. Klapa Antonion, M. Raissi, and L. Joshie, “Machine learning through physics–informed neural networks: Progress and challenges,” *Academic Journal of Science and Technology*, vol. 9, no. 1, p. 2024, 2024.
[11] E. Kharazmi, Z. Zhang, and G. E. Karniadakis, “Variational physics-informed neural networks for solving partial differential equations,” *arXiv preprint arXiv:1912.00873*, 2019.
[12] G. Constantine and T. Savits, “A multivariate faa di bruno formula with applications,” *Transactions of the American Mathematical Society*, vol. 348, no. 2, pp. 503–520, 1996.
[13] J. Bettencourt, M. J. Johnson, and D. Duvenaud, “Taylor-mode automatic differentiation for higher-order derivatives in jax,” in *Program Transformations for ML Workshop at NeurIPS 2019*, 2019.
[14] Y. LeCun, J. Denker, and S. Solla, “Optimal brain damage,” *Advances in neural information processing systems*, vol. 2, 1989.
[15] W. M. Czarnecki, S. Osindero, M. Jaderberg, G. Swirszcz, and R. Pascanu, “Sobolev training for neural networks,” *Advances in neural information processing systems*, vol. 30, 2017.
[16] J. Xu, H. Zhang, L. Liang, L. Deng, Y. Xie, and G. Li, “Np-hardness of tensor network contraction ordering,” *arXiv preprint arXiv:2310.06140*, 2023.
[17] L. Chi-Chung, P. Sadayappan, and R. Wenger, “On optimizing a class of multi-dimensional loops with reduction for parallel execution,” *Parallel Processing Letters*, vol. 7, no. 02, pp. 157–168, 1997.
[18] D. G. a. Smith and J. Gray, “opt_einsum - a python package for optimizing contraction order for einsum-like expressions,” *Journal of Open Source Software*, vol. 3, no. 26, p. 753, 2018.
[19] “thoad: Pytorch high order autodifferentiator.” https://github.com/mntsx/thoad, 2025.

GRADO EN INGENIERÍA MATEMÁTICA E INTELIGENCIA ARTIFICIAL

TRABAJO DE FINAL DE GRADO

APPENDIX

Author: Miguel Montes Lorenzo
Director: David Alfaya Sánchez
Co-Director: Jaime Pizarroso Gonzalo

Madrid 2025

```
# First derivative
X_shapes = [(4, 6), (1, 6), (4, 1), (4, 6), (4, 6), (6,), (4, 6)]
Y_shapes = [(4, 6), (4, 6), (4, 6), (1, 6), (4, 1), (4, 6), (6,)]
for reqx, reqy in [(True, True), (True, False), (False, True)]:
    for xs, ys in zip(X_shapes, Y_shapes):
        for alpha in [1.0, 0.5]:
            X = torch.rand(size=(3, 4), requires_grad=reqx, device=device
                )
            Y = torch.rand(size=(3, 4), requires_grad=reqy, device=device
                )
            O = torch.add(input=X, other=Y)
            O = O.sum() ** 2
            backward(tensor=O, order=2)
            O.backward()
            if reqx:
                assert torch.allclose(
                    X.hgrad[0].flatten(), X.grad.flatten(), atol=1e-4
                )
            if reqy:
                assert torch.allclose(
                    Y.hgrad[0].flatten(), Y.grad.flatten(), atol=1e-4
                )

# Second derivative
X = torch.rand(size=(4, 6), requires_grad=True, device=device)
Y = torch.rand(size=(4, 6), requires_grad=True, device=device)
O = torch.add(input=X, other=Y)
O = O.sum()**2
ctrl = backward(tensor=O, order=2, crossings=True)
def f(a_ref: Tensor, b_ref: Tensor):
    return (a_ref + b_ref).sum()**2
full_hessian = torch.autograd.functional.hessian(f, (X, Y))
H00, _ = ctrl.fetch_hgrad([X, X], keep_batch=False)
H01, _ = ctrl.fetch_hgrad([X, Y], keep_batch=False)
H10, _ = ctrl.fetch_hgrad([Y, X], keep_batch=False)
H11, _ = ctrl.fetch_hgrad([Y, Y], keep_batch=False)
assert torch.allclose(H00.flatten(), full_hessian[0][0].flatten(), atol=1
    e-4)
assert torch.allclose(H01.flatten(), full_hessian[0][1].flatten(), atol=1
    e-4)
assert torch.allclose(H10.flatten(), full_hessian[1][0].flatten(), atol=1
    e-4)
assert torch.allclose(H11.flatten(), full_hessian[1][1].flatten(), atol=1
    e-4)
```

Figure 1: Example of THOAD autodifferentiation test for torch.add operator

```
import timeit
import torch
import thoad
from torch import Tensor

def foward_pass(X: Tensor, *params) -> Tensor:
    T: Tensor = X
    for i, P in enumerate(params):
        last_step: bool = i == (len(params) - 1)
        T = T @ P
        T = torch.softmax(T, dim=1) if last_step else torch.relu(T)
    return T.sum()

def time_autograd_hessian(param_grad: bool, reps: int, X: Tensor, *params
    ) -> float:
    def _fixed_forward_pass(X) -> Tensor:
        return foward_pass(X, *params)
    def _foward_and_backward() -> None:
        if param_grad:
            torch.autograd.functional.hessian(func=foward_pass, inputs=(X
                , *params))
        else:
            torch.autograd.functional.hessian(func=_fixed_forward_pass,
                inputs=X)
        return None
    time: float = timeit(
        lambda: _foward_and_backward(),
        number=reps,
    )
    return time

def time_thoad_hessian(param_grad: bool, reps: int, X: Tensor, *params)
    -> float:
    X.requires_grad_(True)
    params: list[Tensor] = [P.requires_grad_(param_grad) for P in params]
    def _foward_and_backward() -> None:
        T: Tensor = foward_pass(X, *params)
        ctrl: thoad.Controller = thoad.backward(tensor=T, order=2,
            crossings=param_grad, keep_batch=True)
        ctrl.clear()
        return None
    time: float = timeit(
        lambda: _foward_and_backward(),
        number=reps,
    )
    return time
```

Figure 2: Helper functions for PyTorch vs THOAD benchmarks

```python
torch_times: list[float] = []
thoad_times: list[float] = []
for batch_size in [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]:
    param_size: int = 10
    x_shape: Tuple[int, int] = (batch_size, param_size)
    p_shape: Tuple[int, int] = (param_size, param_size)

    X: Tensor = torch.rand(size=x_shape, device=dev)
    params: list[Tensor] = [torch.rand(size=p_shape, device=dev) for _ in
        range(3)]

    reps: int = 1000 // batch_size
    autograd_time: float = time_autograd_jacobian(False, reps, X, *params
        )
    thoad_time: float = time_thoad_jacobian(False, reps, X, *params)

    torch_times.append(autograd_time)
    thoad_times.append(thoad_time)
```

Figure 3: PyTorch vs THOAD benchmark scaling batch size (using helper functions presented in (2))

```python
torch_times: list[float] = []
thoad_times: list[float] = []
for param_size in [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]:
    batch_size: int = 10
    x_shape: Tuple[int, int] = (batch_size, param_size)
    p_shape: Tuple[int, int] = (param_size, param_size)

    X: Tensor = torch.rand(size=x_shape, device=dev)
    params: list[Tensor] = [torch.rand(size=p_shape, device=dev) for _ in
        range(3)]

    reps: int = 1000 // param_size
    autograd_time: float = time_autograd_jacobian(False, reps, X, *params
        )
    thoad_time: float = time_thoad_jacobian(False, reps, X, *params)

    torch_times.append(autograd_time)
    thoad_times.append(thoad_time)
```

Figure 4: PyTorch vs THOAD benchmark scaling dual size (using helper functions presented in (2))

```python
import timeit
import torch
import thoad
from torch import Tensor

def foward_pass(X: Tensor, *params) -> Tensor:
    T: Tensor = X
    for i, P in enumerate(params):
        last_step: bool = i == (len(params) - 1)
        T = T @ P
        T = torch.softmax(T, dim=1) if last_step else torch.relu(T)
    return T.sum()

def time_differentiation(
        reps: int,
        param_grad: bool,
        keep_batch: bool,
        keep_schwarz: bool,
        order: int,
        X: Tensor,
        *params,
    ) -> float:
    X.requires_grad_(True)
    params: list[Tensor] = [P.requires_grad_(param_grad) for P in params]
    def _foward_and_backward() -> None:
        T: Tensor = foward_pass(X, *params)
        ctrl: thoad.Controller = thoad.backward(
            tensor=T,
            order=order,
            crossings=param_grad,
            keep_batch=keep_batch,
            keep_schwarz=keep_schwarz,
        )
        ctrl.clear()
        return None
    time: float = timeit(
        lambda: _foward_and_backward(),
        number=reps,
    )
    return time
```

Figure 5: Helper functions for optimization benchmarks

```python
baseline_times: list[float] = []
optimized_times: list[float] = []
for o in [1, 2, 3]:
    baseline_times.append([])
    optimized_times.append([])
    for batch_size in [10, 20, 30, 40, 50, 60, 70, 80]:
        param_size: int = int(10 * (1 / o) * TENSOR_SCALE)
        x_shape: Tuple[int, int] = (batch_size, param_size)
        p_shape: Tuple[int, int] = (param_size, param_size)

        X: Tensor = torch.rand(size=x_shape, device=dev)
        params: list[Tensor] = [torch.rand(size=p_shape, device=dev) for
            _ in range(3)]

        config.SCHWARZ_OPTIMIZATION = False
        reps: int = int(1000 * (1/batch_size) * (1/order)

        config.BATCH_OPTIMIZATION = False
        regular_time: float = time_differentiation(
            reps, True, True, False, o, X, *params
        )
        config.BATCH_OPTIMIZATION = True
        optimized_time: float = time_differentiation(
            reps, True, True, False, o, X, *params
        )

        baseline_times[-1].append(baseline_time)
        optimized_times[-1].append(optimized_time)
```

Figure 6: benchmark for optimizations scaling batch size (using helper functions presented in (5))

```python
baseline_times: list[float] = []
optimized_times: list[float] = []
for o in [1, 2, 3]:
    baseline_times.append([])
    optimized_times.append([])
    for depth in range(2, 7):
        batch_size: int = 10
        param_size: int = int(10 * (1 / o) * TENSOR_SCALE)
        x_shape: Tuple[int, int] = (batch_size, param_size)
        p_shape: Tuple[int, int] = (param_size, param_size)

        X: Tensor = torch.rand(size=x_shape, device=dev)
        params: list[Tensor] = [torch.rand(size=p_shape, device=dev) for
            _ in range(depth)]

        config.BATCH_OPTIMIZATION = False
        reps: int = int(500 * (1/depth) * (1/order)

        config.SCHWARZ_OPTIMIZATION = False
        baseline_time: float = time_differentiation(
            reps, True, False, True, o, X, *params
        )
        config.SCHWARZ_OPTIMIZATION = True
        optimized_time: float = time_differentiation(
            reps, True, False, True, o, X, *params
        )

        baseline_times[-1].append(baseline_time)
        optimized_times[-1].append(optimized_time)
```

Figure 7: benchmark for optimizations scaling batch size (using helper functions presented in (5))