Understanding Algorithmic Differentiation

Moritz Wolter

March 16, 2025

The High-Performance Computing and Analytics Lab, University of Bonn

Overview

Course Intro

Welcome to Advanced Machine Learning

Course schedule:

- Day 01: Advanced Optimization
- Day 02: Reinforcement learning
- Day 03: CNNs, Very deep neural networks, residual connections
- Day 04: Bayesian Learning
- Day 05: Transformer for sequence-to-sequence processing, Vision Transformer, Transfer learning.

Course Team

- Lecturers: Elena Trunz, Lokesh Veeramacheneni, Moritz Wolter
    - trunz@cs.uni-bonn.de
    - lokiv@uni-bonn.de
    - moritz.wolter@uni-bonn.de
- Tutors: Zahra Ganji, Niklas Kerkfeld, Pauline Lion

Funding

This course is brought to you with support from the BMBF Projects BntrAInee and WestAI.

SPONSORED BY THE

Federal Ministry
of Education
and Research

Motivating Algorithmic Differentiation

Introduction

- Writing your own auto-grad engine is like programming your own compiler, most likely you will never have to do this, but it is the best way to understand what is going on behind the scenes.
- Parts of this presentation are based on the book “Evaluating Derivatives” by Griewank and Walther [GW08].

Terminology

The community does not agree on the terminology.

- According to [GW08] we should speak of Algorithmic Differentiation (AD). [GW08] call the two common modes reverse and forward mode.
- The reverse mode is also called backpropagation in the neural network literature.
- [Gro18] for example, speaks of automatic differentiation instead of algorithmic.
- The PyTorch developers, on the other hand, write autograd, when they mean autodiff or algodiff.

Generally, these terms are synonymous, we should be careful not to mix up forward and backward mode. We will study the backward case today. If you see forward AD, that is something else.

Libraries for Neural Network Optimization

An Overview from Old to New:

- Theano (Python) ($\approx$ 2009 - 2018)
- Torch (Lua) ($\approx$ 2012 - 2017)
- Pytorch (Python, C++, JIT) (2012 - ongoing)
- Tensorflow (C++, Python) (2015 - ongoing)
- Jax (Python, JIT) (2018 - ongoing)

All of these Libraries provide the means for algorithmic gradient computation.

Neural Network Optimization by Gradient Descent

Why are gradients so important?

Following the literature [GBC16, chapter 8], the vector θ denotes learnable parameters. Our network is f. Gradient descent computes,

$$\mathbf{g}_\tau = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} C(f(\mathbf{x}^i; \theta), \mathbf{y}^\mathbf{i}), \tag{1}$$

$$\theta_{\tau+1} = \theta_\tau - \epsilon \mathbf{g}_\tau. \tag{2}$$

With the step counter τ, gradient operator ∇, cost function C, inputs $\mathbf{x}$, and outputs $\mathbf{y}$. It is efficient to process multiple batches at once. m denotes the batch size and ϵ the step size.
We require accurate values for the gradient $\mathbf{g}_\tau$.

The gradient

The gradient lists partial derivatives with respect to all inputs in a vector. For a function $f : \mathbb{R}^n \to \mathbb{R}$ of n variables the gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is defined as

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}. \tag{3}$$

Computing the gradient of a paraboloid

A paraboloid is defined as

$$f(x_1, x_2) = x_1^2 + x_2^2. \tag{4}$$

We can find its gradient by hand

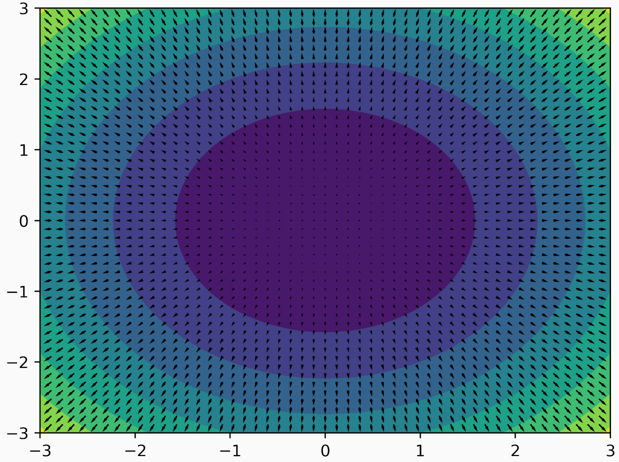
$$\begin{align} \nabla f(x_1, x_2) &= \nabla(x_1^2 + x_2^2) \tag{5} \\ &= \begin{pmatrix} 2x_1 \\ 2x_2 \end{pmatrix}. \tag{6} \end{align}$$

Gradients at points

Equation 5 is defined for all $x_1, x_2 \in \mathbb{R}$. We can evaluate the expression by choosing values for both dimensions. In other words, for every point $\mathbf{p} = (x_1, x_2, \ldots, x_n)$ we can write

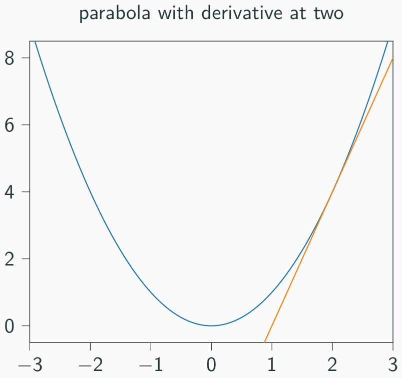
$$\nabla f(\mathbf{p}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{p}) \\ \frac{\partial f}{\partial x_2}(\mathbf{p}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{p}) \end{pmatrix}. \tag{7}$$

Gradients on the Paraboloid



Finite differences?

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{8}$$



Finite differences?

- Finite differences are computed by inserting a numerical value into h instead of taking the limit.
- Can't we use these instead?
- Choose for example $x = 2$ and $h = 10^{-12}$.
- Numerical evaluation with $f(x) = x^2$ yields 4.000355602329364, which is not quite 4.
- If we want to train large networks this precision difference matters!

Finite differences?

- We risk underflow if we choose an h that is too small, since depending on the precision we might observe $f(x+h) \approx f(x)$. Unfortunately, increasing h also increases the error.
- Furthermore, numerical errors can accumulate if the input dimension of vector-valued functions increases.

Finite Differences Error Accumulation Example

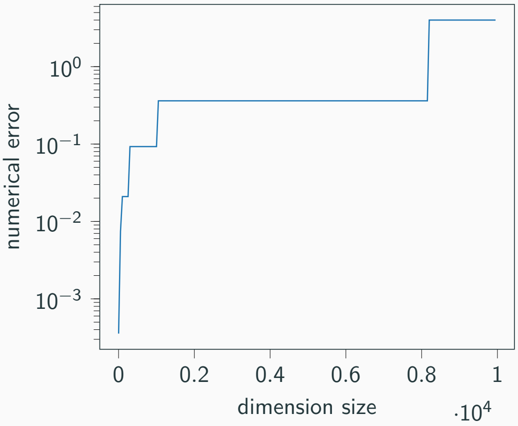


Figure: Numerical error versus dimension the finite difference approximation of $f(\mathbf{x}) = \sum_{i=1}^{N} x_i^2$'s first derivative. $\mathbf{x} = \mathbf{2}$ and $h = 10^{-12}$.

Summary

- Gradients are the key to artificial neural network optimization.
- Finite differences are not good enough [GW08].
- What now?

Backward Algorithmic Differentiation

Differentiation Rules

Chain Rule:

$$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x) \tag{9}$$

Product Rule:

$$\frac{d}{dx}(f(x) \cdot g(x)) = \frac{df(x)}{dx} \cdot g(x) + f(x) \cdot \frac{dg(x)}{dx} \tag{10}$$

Sum Rule:

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx} f(x) + \frac{d}{dx} g(x) \tag{11}$$

with $x \in \mathbb{R}$.

Partial Derivatives

Functions with more than a single input have partial derivatives. Given a function $f(x_1, x_2, \ldots x_n)$ we write $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}$. When computing a partial derivative we apply the usual rules for the variable in the denominator and treat all other inputs as constant.

Example

To illustrate what the formulae mean let's consider

$$f(x_1, x_2) = \sigma(x_1 \cdot x_2) + x_1 \cdot x_2 \tag{12}$$

with $\sigma(x) = \frac{1}{1+\exp(-x)}$ and $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$. This function has the partials,

$$\frac{\partial f}{\partial x_1} = \sigma'(x_1 \cdot x_2) \cdot x_2 + x_2 \tag{13}$$

$$\frac{\partial f}{\partial x_2} = \sigma'(x_1 \cdot x_2) \cdot x_1 + x_1 \tag{14}$$

The Jacobian

The Jacobian is an important tool, that allows the formulation of the chain rule in its multivariate form. For the *n* input and *m* output-case [Gro18],

$$\frac{\partial}{\partial \mathbf{x}} y(\mathbf{x}) = \mathbf{J} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \tag{15}$$

The multivariate chain rule and Vector-Jacobian Products

We are interested in the contributions of each of the inputs $x_1, x_2, \dots x_n$ to the change of an output $\mathbf{y}$. In order to obtain a column vector we re-express the chain rule as [Gro18]

$$\delta \mathbf{x}_c = \mathbf{J}^T \delta \mathbf{y}. \tag{16}$$

In the literature the partials of an intermediate vector $\mathbf{h}$ with respect to the cost function is sometimes written as $\dot{\mathbf{h}}$ [GW08], $\bar{\mathbf{h}}$ [Gro18] or $\delta\mathbf{h}$ [Gre+16]. The transpose of the right side leads to a row vector,

$$\delta \mathbf{x}_r = \delta \mathbf{y}^T \mathbf{J} \tag{17}$$

this form is often expressed as a sum to obtain a formula for each input partial,

$$\delta x_j = \sum_i \delta y_i \frac{\partial y_i}{\partial x_j}. \tag{18}$$

Vector-Jacobian Product examples [Gro18]

A linear layer is typically defined as $\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$. Considering the matrix-vector product leads to,

$$\mathbf{h} = \mathbf{W}\mathbf{x}, \mathbf{J} = \mathbf{W}, \delta\mathbf{x} = \mathbf{W}^T\delta\mathbf{h}. \tag{19}$$

For the element-wise function, we have,

$$\mathbf{y} = \sigma(\mathbf{h}), \mathbf{J} = \begin{pmatrix} \sigma'(h_1) & & 0 \\ & \ddots & \\ 0 & & \sigma'(h_m) \end{pmatrix}, \delta\mathbf{h} = \sigma'(\mathbf{h}) \odot \delta\mathbf{y}, \tag{20}$$

with the element-wise product $\odot$, the sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)}$ and $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$.

Automating the process

Neural networks are expressed as long chains of operations. A fully connected network can be expressed as a composition of additions, multiplications and element-wise activation functions.

Given an incoming inner derivative δh or seed value, we can write down their partial derivatives.

Derivative flow through summation

To allow gradients to flow through summation we compute the partials

$$y = x_1 + x_2 \tag{21}$$

$$\rightarrow \delta x_1 = \frac{\partial(x_1 + x_2)}{\partial x_1} \cdot \delta y = 1 \cdot \delta y \tag{22}$$

$$\rightarrow \delta x_2 = \frac{\partial(x_1 + x_2)}{\partial x_2} \cdot \delta y = 1 \cdot \delta y, \tag{23}$$

with δy as the inner derivative or seed value.

Derivative flow through products

To allow gradients to flow through products we compute the partials

$$
\begin{aligned}
y &= x_1 \cdot x_2 && (24)\\
&\rightarrow \delta x_1 = \frac{\partial(x_1 \cdot x_2)}{\partial x_1} \cdot \delta y = x_2 \cdot \delta y && (25)\\
&\rightarrow \delta x_2 = \frac{\partial(x_1 \cdot x_2)}{\partial x_2} \cdot \delta y = x_1 \cdot \delta y, && (26)
\end{aligned}
$$

with δy as the inner derivative or seed value.

Derivative flow through activation functions

Similarly for an element-wise function f,

$$y = f(x) \tag{27}$$

$$\rightarrow \delta x = f'(x)\delta y. \tag{28}$$

Matrix multiplication

Computing the output of a fully connected linear layer $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ requires evaluating a matrix-vector product. We can re-write this to use only scalar sums and products,

$$y_i = \sum_j w_{ij} x_j + b_i \tag{29}$$

Python's built-in `sum` function will help you here.

Magic functions in Python

Backward mode Algorithmic Differentiation requires us to keep track of all operations and record what to do when users trigger the backward pass. An elegant way of achieving this goal is by re-implementing the `__add__` and `__mul__` functions. These two functions allow you to control the effect of `+` and `*` operations.

Conclusion

We have seen the elements we require to implement an autograd engine in Python. Today's exercise asks you to try for yourself.

Literature i

References

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. ***Deep learning*.** MIT press, 2016.

[Gre+16] Klaus Greff, Rupesh K Srivastava, Jan Koutnik, Bas R Steunebrink, and Jürgen Schmidhuber. **"LSTM: A search space odyssey."** In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

Literature ii

[GW08] Andreas Griewank and Andrea Walther. ***Evaluating derivatives: principles and techniques of algorithmic differentiation*.** SIAM, 2008.

[Gro18] Roger Grosse. ***CSC321 Lecture 10: Automatic Differentiation*.** `https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/slides/lec10.pdf`. 2018.