M2 MODO
2025-2026

MACHINE LEARNING FOR OPTIMIZATION

November 20, 2025

Today: Introduction, Automatic differentiation / Backpropagation

Logistics:
- Course webpage
  https://www.lamsade.dauphine.fr/~croyer/teachMLO.html
- By email
  clement.royer@lamsade.dauphine.fr

Schedule (subject to uncertainties ...)

Nov 20	1.45 - 5pm :	Lecture
Nov 27	8.30 - 11.45pm :	Lab
Dec 8	1.45 - 5pm :	Lab
Dec 9	8.30 - 11.45pm :	Lab
Dec 16	8.30 - 11.45pm :	Lab

⚠ Assessment:
- Homework / Project (in groups of n students with $n \in \{1, 2\}$)
- Goal: Push lab sessions a bit further
- Intended deadline: Winter 2026 (for sure after the holidays)

Key concepts in ML for optimization

Main question: What can we do with ML tools and optimization solvers?

1) Try to learn something as you are solving an optimization problem and collecting data about the problem

Ex) In blackbox optimization, you collect a history of function values and you try to learn a (regression) model of the function to be optimized from those values

Giovanelli Sohab Vicente 2024: Neural networks do not necessarily give the best models in blackbox optimization

2) Learn on past problem instances (and hope that it generalizes to new instances of the problem)

Ex) Learning hidden constraint classifier in blackbox optimization

Tools (from ML) that people are using to perform 1) and 2)

↳ No LLMs (for now)

↳ But neural networks!

- Classical ones (RNNs, CNNs, ...)
- Less common ones (Graph neural networks, neural network with implicit layers)

→ These networks are used to solve supervised learning tasks, but also reinforcement learning and imitation learning

Basic mathematical formulation of a neural net

$$z_0 \longmapsto z_1 \longmapsto \cdots \longmapsto z_L \qquad \text{"L layers"}$$

$$\forall\, l = 1..L, \quad z_l = \phi\left(A_l z_{l-1} + b_l\right) \qquad \text{"}l^{\text{th}}\text{ layer"}$$

ϕ: Activation function (tanh, ReLU, ..) applied componentwise

A_l, b_l: parameters to be learned

$\mathrm{ReLU}(t) = \max(t, 0)$

$\mathrm{ReLU}(z) = [\max(t_i, 0)]_i$

Neural network (NN): Input $x = z_0$, get $NN(x) = z_L$ as output

Examples (of layers $z \mapsto y$)

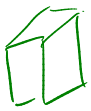
- Fully connected layers (Basic one)

$$y_j = \phi\left(\sum_j A_{ij} z_i + b_j\right)$$

$A = [A_{ij}]$, $b = \begin{bmatrix} b_1 \\ \vdots \end{bmatrix}$, $z = [z_i]$, $y = [y_i]$

- Convolutional layers (at the heart of CNNs)

$z \in \mathbb{R}^{d_0 \times d_0 \times c_0}$ ← three-way tensor



$$y_{a,b,c} = \sigma\left(\sum_{i,j,k} \left(A_{i,j,k,c}\, x_{a-i,b-j,k} + b_k\right)\right)$$

$y_{a,b,c} \in \mathbb{R}^{q_0 \times q_0 \times c_1}$

$A \in \mathbb{R}^{q_0 \times q_0 \times c_0 \times c_1}$ $\quad b \in \mathbb{R}^{c_1}$

Working with tensors: very important in ML

- Recurrent layers (RNNs) / ResNet

ResNet: $y = z + B(\mathrm{ReLU}(Az))$

↑ residual connection

RNN: $z_\ell = f(z_{\ell-1}) \quad \forall \ell$ (same function f at every layer)

- Graph neural networks (GNNs)

Full description:

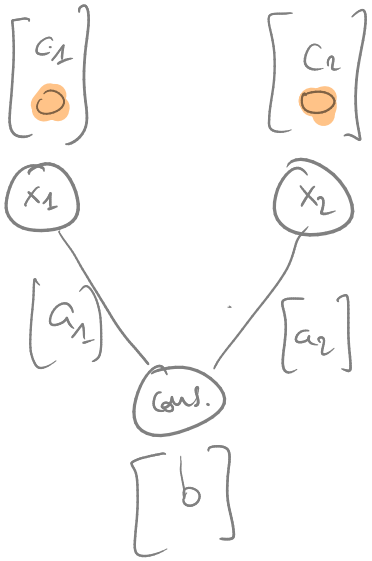
Graph $G = (V, E)$ undirected graph

Embedding:

$$\xi(G, \cdot): V \longrightarrow \mathbb{R}^d$$
$$v \longmapsto \xi(G, v)$$

Map a vertex to a vector of features (also possible for edges)

Ex) A vertex could represent a variable or a constraint in linear programming



$$\begin{cases} \text{minimize} & c_1 x_1 + c_2 x_2 \\ x \in \mathbb{R}^2 & \\ \text{s.t.} & a_1 x_1 + a_2 x_2 = b \quad \text{Constraint} \\ & x \geq 0 \end{cases}$$

→ The GNN structure considers in computing functions of the graph and its embedding.

Message Passing Operator

Start with an embedding ξ^0

Compute the recursion $\forall v \in V$ (and possibly $\forall$ edge $e \in E$)

$$\forall t = 0..T-1, \quad \xi^{t+1}(G,v) = \text{Comb}\left[\xi^t(G,v), \sum_{u \in N(v)} \xi^t(G,u) \right]$$

Comb: (ex) → Linear mapping → Neural network layer (of convolutional/fully connected type)

$N(v)$: Neighbors of v in the graph

NB: As we will (probably) see, implementing GNNs is challenging!

Typical abstraction: The parameters of the NN are gathered in a (big) vector $w \in \mathbb{R}^d$ $\quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}$

Training a NN: Given data $\{(x_i, y_i)\}_{i=1..n}$, find the best value of w, that is the value for which $NN(x_i, w) \approx y_i$

(want to output something close to y_i when given x_i as input)

$\Rightarrow$ This is an optimization problem!

One typical formulation:

$$(*) \quad \underset{w \in \mathbb{R}^d}{\text{minimize}} \ \frac{1}{n} \sum_{i=1}^{n} \text{loss}\left(NN(x_i, w), y_i \right)$$

$NN(x_i, w)$: output of the NN with input x_i and parameters w

loss: Loss function

(ex) In the blackbox notebook

- Linear regression
  $$\underset{w \in \mathbb{R}^d}{\text{minimize}} \ \frac{1}{n}\|Xw - y\|_2^2$$
- logistic regression
  $$\underset{w \in \mathbb{R}^d}{\text{minimize}} \ \frac{1}{n}\sum_{i=1}^{n} \log(1+\exp(-y_i x_i^T w))$$

(e.g. $\text{loss}(a,b) = \frac{1}{2}\|a-b\|_2^2$
$= \frac{1}{2}(a-b)^T(a-b)$
$= \frac{1}{2}\sum_{i=1}^{m}(a_i-b_i)^2$)

$x_i \mapsto x_i^T w$: "One-layer linear NN" (aka a linear model)

→ State of the art for problems of the form (*)

Apply some form of stochastic gradient technique (SGD, Adam, Adagrad, ...)

that involve computing the derivative of NN with respect to the parameters w

→ Called backpropagation in general, but sometimes the derivative calculation (the most important part) is called backpropagation

↳ Backpropagation is one of the reasons deep learning is successful!

Backpropagation / Automatic differentiation

General idea: Given a code that computes $f(w)$ where $f: \mathbb{R}^d \to \mathbb{R}$, design a code that evaluates the derivative of f at w, denoted by $\frac{\partial f}{\partial w}$

Theorem (informally): If you apply backward-mode automatic differentiation (aka backpropagation), the cost of computing $\frac{\partial f}{\partial w}$ is at most 6 times the cost of computing $f(w)$

Notation: $\frac{\partial f}{\partial w} \equiv \frac{\partial f}{\partial w}(w)$ fixed value rather than a function

$f: \mathbb{R} \to \mathbb{R}$, $w \mapsto f(w)$ $\qquad \frac{\partial f}{\partial w} \in \mathbb{R}$ (or $\mathbb{R}^{1\times 1}$, or $\mathbb{R}^{1\times 1\times 1}$, ...)

$f: \mathbb{R}^d \to \mathbb{R}$, $w \mapsto f(w)$ $\qquad \frac{\partial f}{\partial w} \in \mathbb{R}^{1\times d}$ "row vector" $[v_1 \dots v_d]$

$\frac{\partial f}{\partial w} = \nabla f(w)^T$ $\qquad \nabla f(w)$: gradient $\in \mathbb{R}^d = \mathbb{R}^{d\times 1}$ $\begin{bmatrix} v_1 \\ \vdots \\ v_d \end{bmatrix}$

$f: \mathbb{R}^d \to \mathbb{R}^q$
$w \mapsto f(w) \in \mathbb{R}^q$

$\frac{\partial f}{\partial w} \in \mathbb{R}^{q \times d}$ "Jacobian matrix"

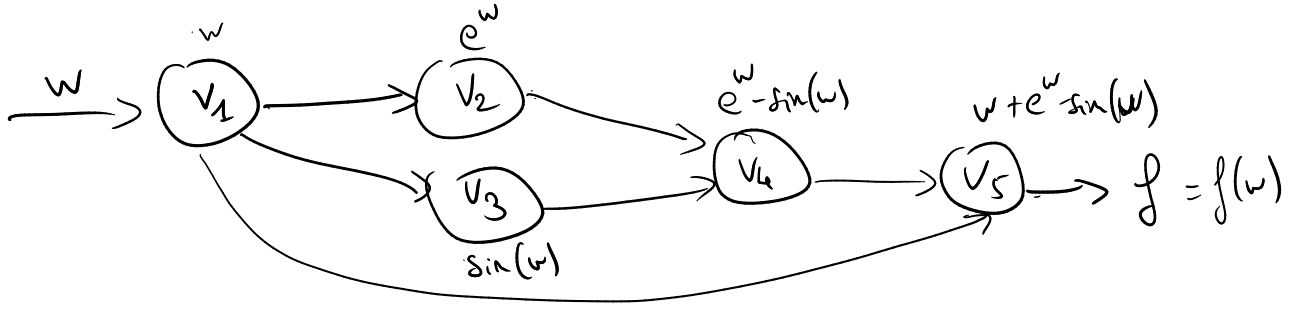
$f: \mathbb{R}^{d_1 \times \dots \times d_r} \to \mathbb{R}^{q_1 \times \dots \times q_s}$
$w \mapsto f(w)$

$\frac{\partial f}{\partial w} \in \mathbb{R}^{(q_1 \times \dots \times q_s) \times (d_1 \times \dots \times d_r)}$ (equivalently $\mathbb{R}^{q_1 \times \dots \times q_s \times d_1 \times \dots \times d_r}$)

Example of applying automatic differentiation

$f: \mathbb{R} \to \mathbb{R}$
$w \mapsto w + e^w - \sin(w)$

(1) Build a computational graph for computing $f(w)$



$\Rightarrow$ this graph is always a DAG (Directed Acyclic Graph)

(2) Compute $f(w)$ by doing a "forward pass" on the graph

Actual calculation of $f = f(w)$:

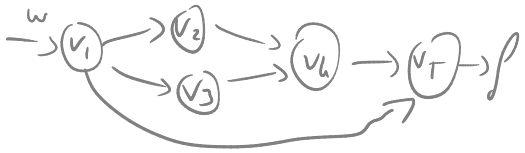
$v_1 = w$
$v_2 = e^{v_1}$
$v_3 = \sin(v_1)$
$v_4 = v_2 - v_3$
$v_5 = v_1 + v_4$
$f = v_5$

(+) Store the dependencies between the v_i's that are linked by an arc

$\frac{\partial v_i}{\partial v_j}$ $\forall$ arc (j, i)

→ After the forward pass, every node i contains $v_i(w)$ (a value)

③ Go through the graph backwards ("backward pass") to compute $\frac{\partial f}{\partial v_i}$ $\forall i$:



stored at node v_5 → $\frac{\partial f}{\partial v_5} = 1$

$$\frac{\partial f}{\partial v_4} = \frac{\partial f}{\partial v_5}\frac{\partial v_5}{\partial v_4} = 1 \times 1 = 1$$

Information available after the forward pass

"Chain rule"

$v_5 = v_4 + v_4$

$$\frac{\partial f}{\partial v_3} = \frac{\partial f}{\partial v_4}\frac{\partial v_4}{\partial v_3} = 1 \times (-1) = -1$$

$v_4 = v_2 - v_3$

$$\frac{\partial f}{\partial v_2} = \frac{\partial f}{\partial v_4} \times \frac{\partial v_4}{\partial v_2} = 1 \times 1 = 1$$

$v_2 = e^{v_1}$

$v_3 = \cos v_1$

$$\frac{\partial f}{\partial v_1} = \sum_{j:(1,j)\,\text{arc}} \frac{\partial f}{\partial v_j} \times \frac{\partial v_j}{\partial v_1} = \frac{\partial f}{\partial v_2} \times \frac{\partial v_2}{\partial v_1} + \frac{\partial f}{\partial v_3} \times \frac{\partial v_3}{\partial v_1} + \frac{\partial f}{\partial v_5} \times \frac{\partial v_5}{\partial v_1}$$

$$= 1 \times e^{v_1} + -1 \times \cos(v_1) + 1 \times 1$$

$$= e^{v_1} - \cos(v_1) + 1$$

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial v_1} \times \underbrace{\frac{\partial v_1}{\partial w}}_{1} = e^{v_1} - \cos(v_1) + 1 = e^{w} - \cos(w) + 1$$

→ The approach extends to functions with multiple variables (and multiple outputs), as well as functions defined in a sequential

way (like NN)

$$x \to z_0 \mapsto \cdots \mapsto z_\ell = NN(x)$$

$\to$ AD computes $\frac{\partial z_\ell}{\partial z_{\ell-1}}$ $\forall \ell$

$\to$ Easy to compute $\frac{\partial NN}{\partial x}$ by AD, and you also get the derivatives with respect to all parameters by doing so

Example: Two-layer neural network

$$f(x, y, W_1, W_2) = \frac{1}{2} \| W_2 \sigma(W_1 x) - y \|_2^2$$

Neural network part
$x \mapsto W_2 \sigma(W_1 x)$
h: hidden dimension

$x \in \mathbb{R}^{d_x}$, $y \in \mathbb{R}^{d_y}$, $W_1 \in \mathbb{R}^{h \times d_x}$, $W_2 \in \mathbb{R}^{d_y \times h}$

$\sigma : \mathbb{R}^h \to \mathbb{R}^h$
$v \mapsto [\tanh(v_i)]_{i=1..h}$

Goal: Compute $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial W_1}$

Forward pass (looks like the actual "forward" function in PyTorch / Jax)

$\in \mathbb{R}^h \to z_1 = W_1 x$

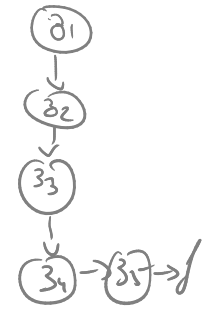
$\mathbb{R}^h \to z_2 = \sigma(z_1)$

$\mathbb{R}^{d_y} \to z_3 = W_2 z_2$

$\mathbb{R}^{d_y} \to z_4 = z_3 - y$

$\mathbb{R} \to z_5 = \frac{1}{2} \| z_4 \|_2^2$

$f = z_5$



Backward pass

$$\frac{\partial f}{\partial z_5} = 1$$

$$\frac{\partial f}{\partial z_4} = \underbrace{\frac{\partial f}{\partial z_5}}_{1} \underbrace{\frac{\partial z_5}{\partial z_4}}_{1\times d_y} = 1 \times z_4^T \qquad z_4 \in \mathbb{R}^{d_y},\ z_4^T \in \mathbb{R}^{1\times d_y}$$

$$\frac{\partial f}{\partial z_3} = \frac{\partial f}{\partial z_4} \times \underbrace{\frac{\partial z_4}{\partial z_3}}_{I \in \mathbb{R}^{d_y \times d_y}} = z_4^T \in \mathbb{R}^{1\times d_y}$$

$$I = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix}$$

$z_3 \in \mathbb{R}^{d_y}$, $z_2 \in \mathbb{R}^{h}$

$$z_3 = W_2 z_2$$

$$\frac{\partial z_3}{\partial z_2} = W_2$$

$$\frac{\partial f}{\partial z_2} = \frac{\partial f}{\partial z_3} \times \frac{\partial z_3}{\partial z_2} = \underbrace{z_4^T}_{1\times d_y} \underbrace{W_2}_{d_y \times h} \qquad \frac{\partial f}{\partial z_2} : 1\times h$$

$$\frac{\partial f}{\partial z_1} = \frac{\partial f}{\partial z_2} \times \frac{\partial z_2}{\partial z_1} = z_4^T W_2 \underbrace{\Sigma}_{h\times h}$$

$$\Sigma = \begin{bmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_h \end{bmatrix}$$

$$\sigma_i = \tanh'([z_1]_i) = 1 + \tanh^2([z_1]_i)$$

a) $$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z_1} \times \frac{\partial z_1}{\partial x} = \underbrace{z_4^T W_2 \Sigma W_1}_{1\times d_x} = (W_2 \sigma(W_1 x) - y)^T W_2 \Sigma W_1$$

(W_1 over $\frac{\partial z_1}{\partial x}$)

b) $$\frac{\partial f}{\partial W_1} = \frac{\partial f}{\partial z_1} \times \frac{\partial z_1}{\partial W_1} = \underbrace{z_4^T W_2 \Sigma}_{1\times h} \underbrace{\frac{\partial z_1}{\partial W_1}}_{h\times(h\times d_x)} = \underbrace{\Sigma W_2^T z_4 x^T}_{h\times h\ \ h\times d_y\ \ d_y\times 1\ \ 1\times d_x}$$

$W_1 \in \mathbb{R}^{h\times d_x}$, $\frac{\partial f}{\partial W_1} : h\times d_x$

Implicit layers in neural networks

↳ Standard layer choices in NNs are explicit

$$z_\ell = \tanh(A z_\ell + b)$$

which enables backpropagation.

↳ Implicit layer approach: Define a layer in terms of joint conditions involving the input and the output

→ De-couple what the layer does from the actual calculation

Examples

- Algebraic equation as a layer:

  z_ℓ computed from $z_{\ell-1}$ so that $c(z_\ell, z_{\ell-1}) = 0$

  (Deep Equilibrium Models)

- Differential equations

  $z_\ell = \varphi(1)$ where φ solution of an ordinary differential equation

$$\begin{cases} \frac{\partial \varphi}{\partial t}(t) = f(\varphi(t)) & \forall t \in [0,1] \\ \varphi(0) = z_{\ell-1} \end{cases}$$

  (Neural ODE, 2018-)

- Optimization problem

$$z_l \in \underset{z}{\operatorname{argmin}}\ f(z; z_{l-1}) \quad \text{s.t.} \quad z \in \mathcal{Z}(z_{l-1})$$

- $\underset{z}{\operatorname{argmin}}$: set of solutions
- $f(z; z_{l-1})$: objective function
- s.t.: subject to
- $\mathcal{Z}(z_{l-1})$: feasible set

"Optimization Layers"

Used in 2 ways:

(1) For problems that are easy to solve

$$\forall z \in \mathbb{R}^q,\ \text{ReLU}(z) = \left[\max(z_i, 0)\right]_{i=1..q}$$

$$= \underset{y}{\operatorname{argmin}} \frac{1}{2}\|y - z\|_2^2 \ \text{s.t.}\ \begin{matrix} y \geq z \\ y \geq 0 \end{matrix}$$

(2) For embedding optimization solvers in a ML pipeline

z_{l-1} : parameters/data for the solver

z_l : solution/output information from the solver

$\Rightarrow$ Want to train a NN on a task which involves solving an optimization problem through a solver (which you do not have explicit description of)

Key: Computing $\frac{\partial z_l}{\partial z_{l-1}}$!

→ Involves optimality conditions for the optimization problem

→ Has to be done "by hand" because solvers are not part of the classical AD rules