Deep Learning

Lecture 3: Automatic differentiation

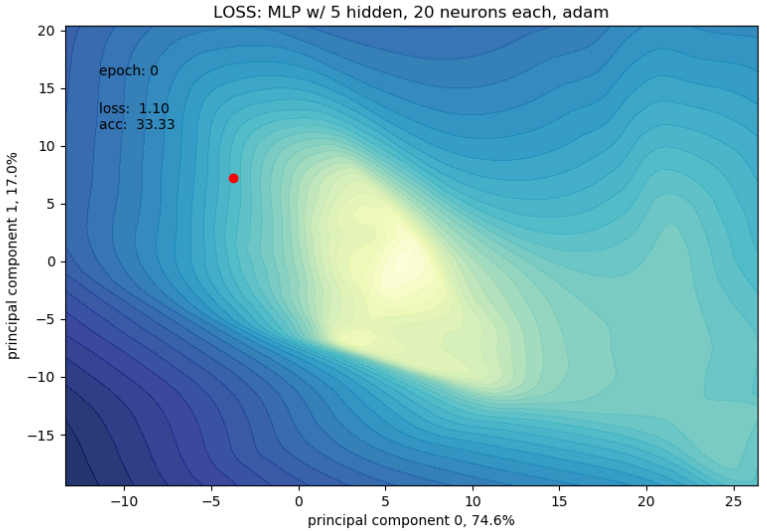
Prof. Gilles Louppe
g.louppe@uliege.be

Today

- Calculus
- Automatic differentiation
- Implementation
- Beyond neural networks

Implementing backpropagation by hand is like programming in assembly language. You will probably never do it, but it is important for having a mental model of how everything works.

Roger Grosse



Motivation

- Gradient-based training algorithms are the workhorse of deep learning.
- Deriving gradients by hand is tedious and error prone. This becomes quickly impractical for complex models.
- Changes to the model require rederiving the gradient.

Image credits: Visualizing optimization trajectory of neural nets, Logan Yang, 2020.

Programs as differentiable functions

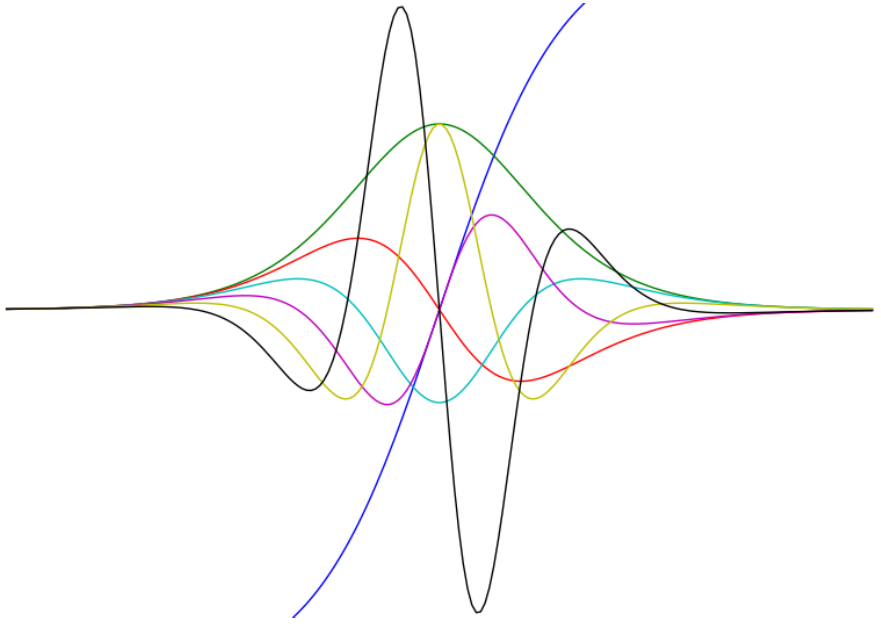
A program is defined as a composition of primitive operations that we know how to differentiate individually.

```
import jax.numpy as jnp
from jax import grad

def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs

def loss_fun(params, inputs, targets):
    preds = predict(params, inputs)
    return jnp.mean((preds - targets)**2)

grad_fun = grad(loss_fun)
```



Modern frameworks support higher-order derivatives.

```
def tanh(x):
    y = jnp.exp(-2.0 * x)
    return (1.0 - y) / (1.0 + y)

fp = grad(tanh)
fpp = grad(grad(tanh))  # what sorcery is this?!
...
```

Automatic differentiation

Automatic differentiation (AD) provides a family of algorithms for evaluating the derivatives of a function specified by a computer program.

- $\neq$ symbolic differentiation, which aims at identifying some human-readable expression of the derivative.
- $\neq$ numerical differentation (finite differences), which may introduce round-off errors.

Calculus

Derivative

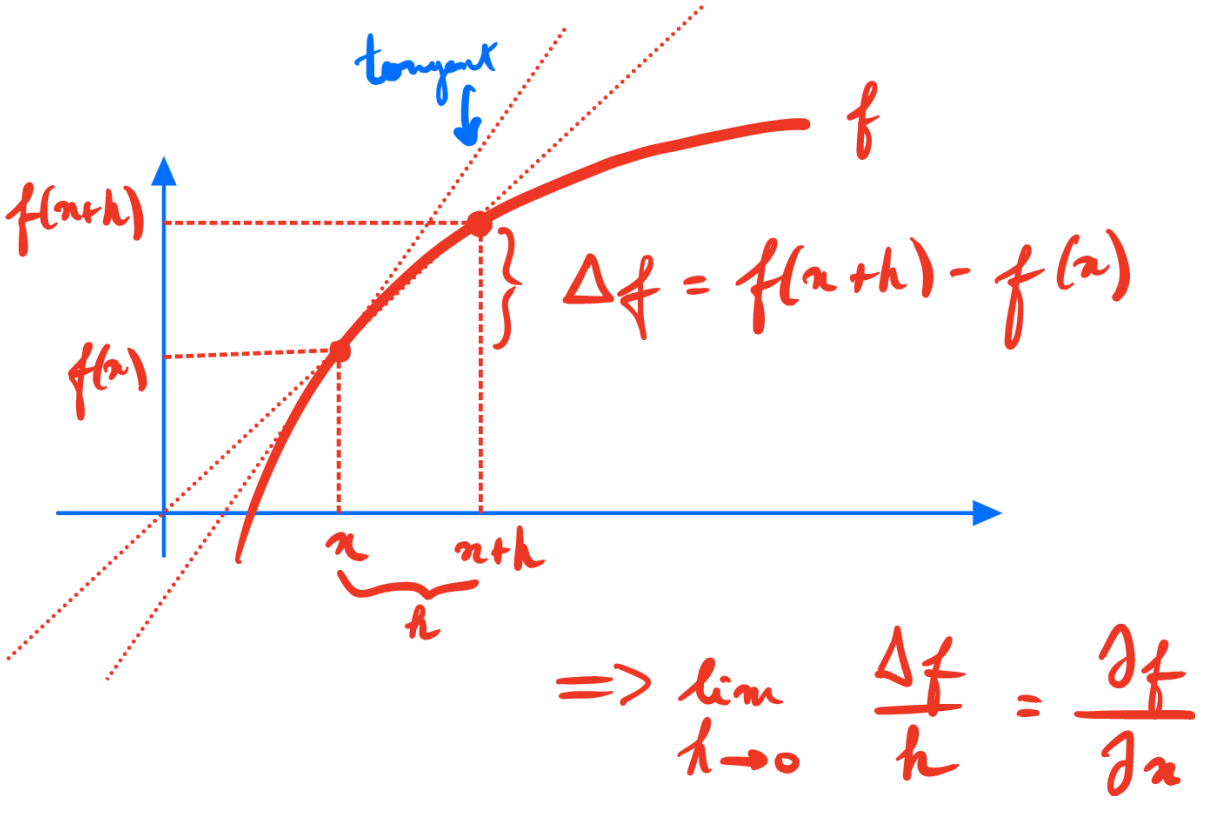
Let $f : \mathbb{R} \to \mathbb{R}$.

The derivative of f is

$$f'(x) = \frac{\partial f}{\partial x}(x) \triangleq \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

where

- $f'(x)$ is the Lagrange notation,
- $\frac{\partial f}{\partial x}(x)$ is the Leibniz notation.



The derivative $\frac{\partial f(x)}{\partial x}$ of f represents its instantaneous rate of change at x.

Gradient

The gradient of $f : \mathbb{R}^n \to \mathbb{R}$ is

$$\nabla f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^n,$$

i.e., a vector that gathers the partial derivatives of f.

Applying the definition of the derivative coordinate-wise, we have

$$[\nabla f(\mathbf{x})]_j = \frac{\partial f}{\partial x_j}(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{e}_j) - f(\mathbf{x})}{h},$$

where $\mathbf{e}_j$ is the j-th basis vector.

Jacobian

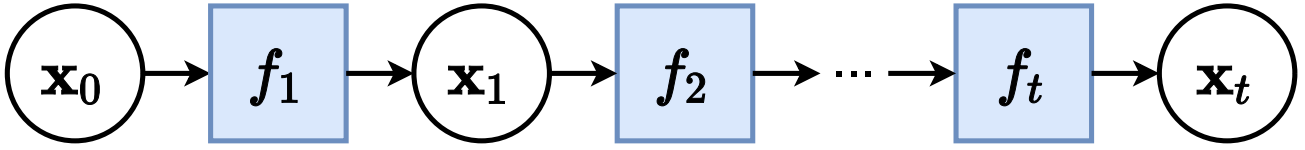
The Jacobian of $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is

$$
\begin{aligned}
J_{\mathbf{f}}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}) &\triangleq \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial f_m}{\partial x_n}(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^{m \times n} \\
&= \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial \mathbf{f}}{\partial x_n}(\mathbf{x}) \end{bmatrix} \\
&= \begin{bmatrix} \nabla f_1(\mathbf{x})^T \\ \vdots \\ \nabla f_m(\mathbf{x})^T \end{bmatrix}
\end{aligned}
$$

The gradient's transpose is thus a wide Jacobian ($m = 1$).

Automatic differentiation

Chain composition



Let us assume a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ that decomposes as a chain composition

$$\mathbf{f} = \mathbf{f}_t \circ \mathbf{f}_{t-1} \circ \ldots \circ \mathbf{f}_1,$$

for functions $\mathbf{f}_k : \mathbb{R}^{n_{k-1}} \to \mathbb{R}^{n_k}$, for $k = 1, \ldots, t$.

By the chain rule,

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_0} = \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_{t-1}} \underbrace{\frac{\partial \mathbf{x}_{t-1}}{\partial \mathbf{x}_0}}_{\text{recursive case}}$$

$$= \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_{t-1}} \frac{\partial \mathbf{x}_{t-1}}{\partial \mathbf{x}_{t-2}} \cdots \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}$$

Forward accumulation

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_0} = \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_{t-1}} \left(\frac{\partial \mathbf{x}_{t-1}}{\partial \mathbf{x}_{t-2}} \left(\dots \left(\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0} \right) \dots \right) \right)$$

Reverse accumulation

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_0} = \left(\left(\dots \left(\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_{t-1}} \frac{\partial \mathbf{x}_{t-1}}{\partial \mathbf{x}_{t-2}} \right) \dots \right) \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \right) \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}$$

Complexity

The time complexity of the forward and reverse accumulations are

$$\mathcal{O}\left(n_0 \sum_{k=1}^{t-1} n_k n_{k+1}\right) \quad \text{and} \quad \mathcal{O}\left(n_t \sum_{k=0}^{t-2} n_k n_{k+1}\right).$$

(Prove it!)

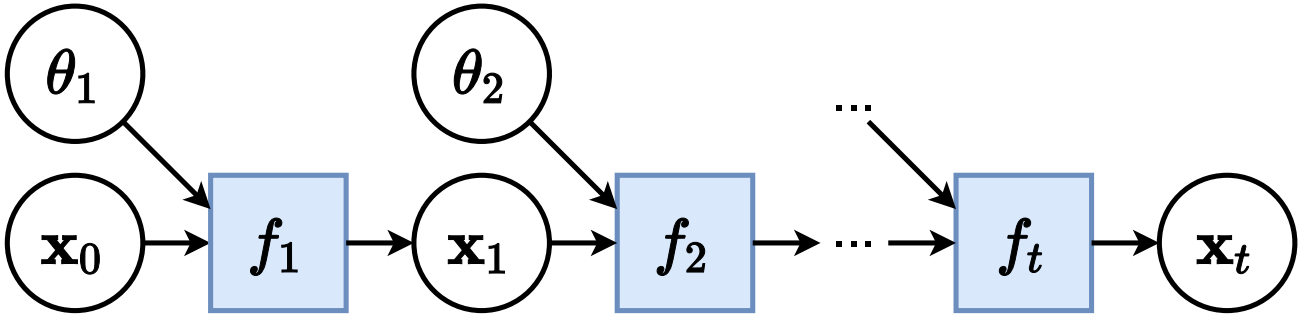
> If $n_t \ll n_0$ (which is typical in deep learning), then **backward accumulation is cheaper**. And vice-versa.

Multi-layer perceptron

Chain compositions can be generalized to feedforward neural networks of the form

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}, \theta_k)$$

for $k = 1, \ldots, t$, and where θ_k are vectors of parameters and $\mathbf{x}_0 \in \mathbb{R}^{n_0}$ is given. In supervised learning, $\mathbf{f}_t$ usually corresponds to a scalar loss ℓ, hence $n_t = 1$.



(whiteboard example)

AD on computer programs

Let $\mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_s)$ denote a generic function where

- $\mathbf{x}_1, \ldots, \mathbf{x}_s$ are the input variables,
- $f(\mathbf{x}_1, \ldots, \mathbf{x}_s)$ is implemented by a computer program producing intermediate variables $(\mathbf{x}_{s+1}, \ldots, \mathbf{x}_t)$,
- t is the total number of variables, with $\mathbf{x}_t$ denoting the output variable,
- $\mathbf{x}_k \in \mathbb{R}^{n_k}$, for $k = 1, \ldots, t$.

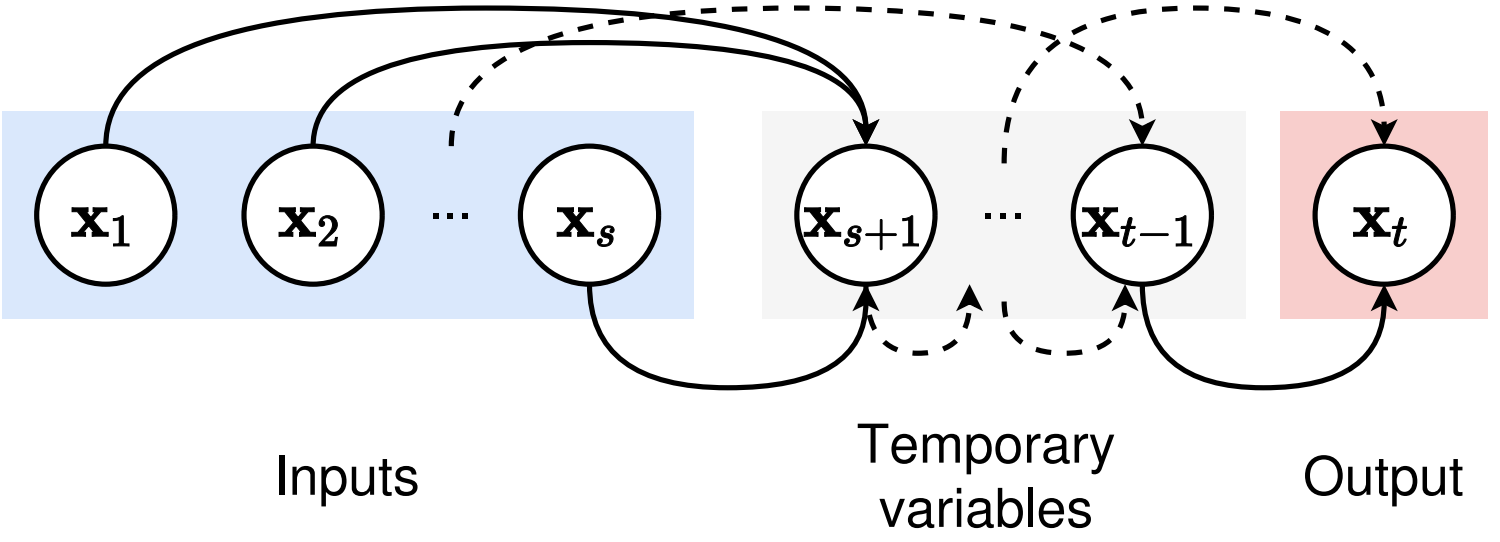
The goal is to compute the Jacobians $\frac{\partial \mathbf{f}}{\partial \mathbf{x}_k} \in \mathbb{R}^{n_t \times n_k}$, for $k = 1, \ldots, s$.

Computer programs as computational graphs

A numerical algorithm is a succession of instructions of the form

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_1, \ldots, \mathbf{x}_{k-1})$$

for $k = s+1, \ldots, t$, and where $\mathbf{f}_k$ is a function which only depends on the previous variables.



This computation can be represented by a directed acyclic graph where

- the nodes are the variables $\mathbf{x}_k$,
- an edge connects x_i to x_k if x_i is an argument of $\mathbf{f}_k$.

The evaluation of $\mathbf{x}_t = \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_s)$ thus corresponds to a forward traversal of this graph.

Forward mode

The forward mode of automatic differentiation consists in computing

$$\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_1} \in \mathbb{R}^{n_k \times n_1}$$

for all variables $\mathbf{x}_k$, iteratively from $k = s + 1$ to $k = t$.

Initialization

Set the Jacobians of the input nodes with

$$\begin{aligned}
\frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_1} &= 1_{n_1 \times n_1} \\
\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} &= 0_{n_2 \times n_1} \\
&\dots \\
\frac{\partial \mathbf{x}_s}{\partial \mathbf{x}_1} &= 0_{n_s \times n_1}
\end{aligned}$$

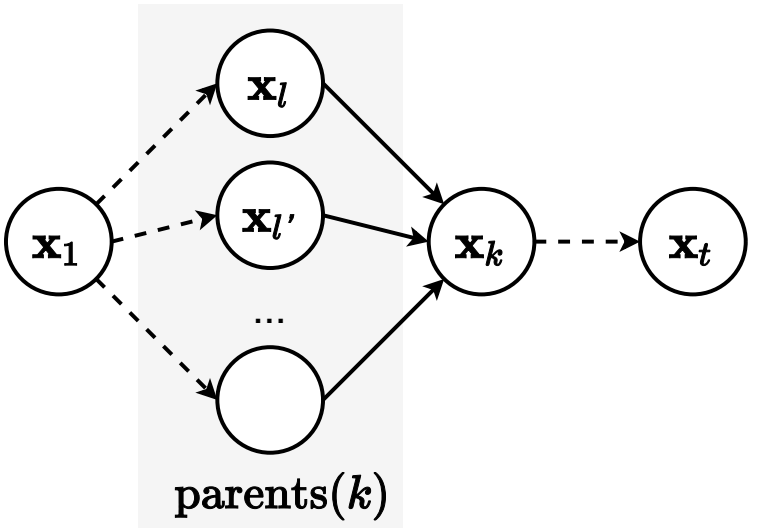
Forward recursive update

For all $k = s+1, \dots, t$,

$$\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_1} = \sum_{l \in \text{parents}(k)} \left[\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_l}\right] \times \frac{\partial \mathbf{x}_l}{\partial \mathbf{x}_1},$$

where

- $\left[\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_l}\right]$ denotes the on-the-fly computation of the Jacobian locally associated to the primitive $\mathbf{f}_k$,
- $\frac{\partial \mathbf{x}_l}{\partial \mathbf{x}_1}$ is obtained from the previous iterations (in topological order).



(whiteboard example)

Forward mode automatic differentiation needs to be repeated for $k = 1, \ldots, s$. For a large s, this is prohibitive.

However, the cost in terms of memory is limited since temporary variables can be freed as soon as their child nodes have all been computed.

Backward mode

Instead of evaluating the Jacobians $\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_1} \in \mathbb{R}^{n_k \times n_1}$ for $k = s+1, \ldots, t$, the reverse mode of automatic differentation consists in computing

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \in \mathbb{R}^{n_t \times n_k}$$

recursively from $k = t$ down to $k = 1$.

Initialization

Set the Jacobian of the output node to

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_t} = 1_{n_t \times n_t}.$$

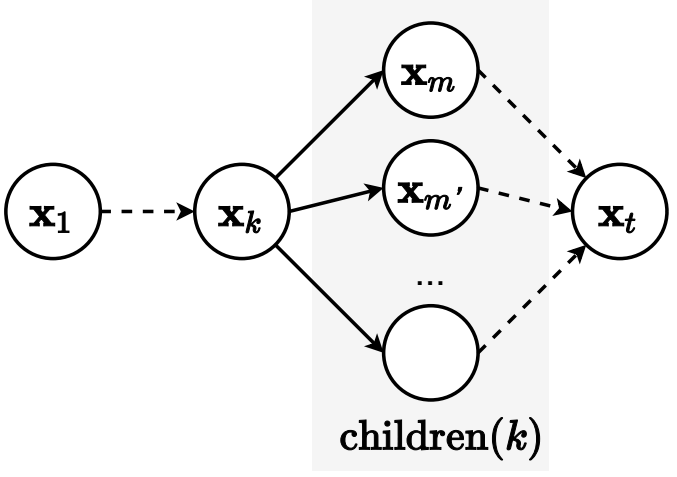
Backward recursive update

For all $k = t-1, \ldots, 1$,

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \sum_{m \in \text{children}(k)} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_m} \times \left[\frac{\partial \mathbf{x}_m}{\partial \mathbf{x}_k} \right]$$

where

- $\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_m}$ is obtained from previous iterations (in reverse topological order) and is known as the adjoint,
- $\left[\frac{\partial \mathbf{x}_m}{\partial \mathbf{x}_k} \right]$ denotes the on-the-fly computation of the Jacobian locally associated to the primitive $\mathbf{f}_m$.



(whiteboard example)

The advantage of backward mode automatic differentiation is that a single traversal of the graph allows to compute all $\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k}$.

However, the cost in terms of memory is significant since all the temporary variables computed during the forward pass must be kept in memory.

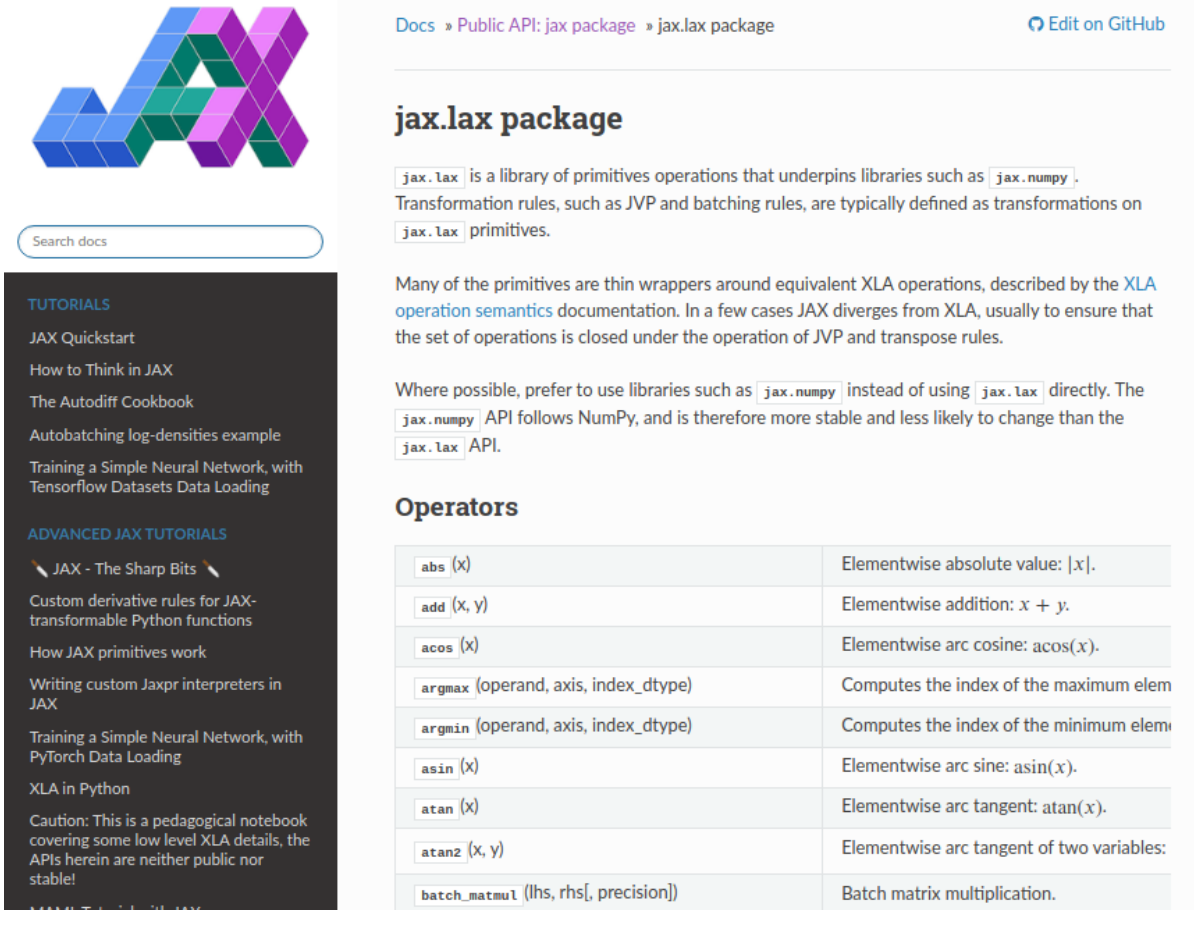
Implementations


PyTorch


TensorFlow

Primitives

Most automatically-differentiable frameworks are defined by a collection of composable primitive operations.

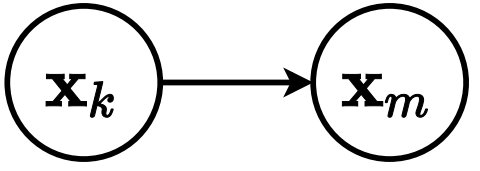


Composing primitives

Primitive functions are composed together into a graph that describes the computation. The computational graph is either built

- ahead of time, from the abstract syntax tree of the program or using a dedicated API (e.g., Tensorflow 1), or
- just in time, by tracing the program execution (e.g., Tensorflow Eager, JAX, PyTorch).

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = ? \quad \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_m}$$



VJPs

In the backward recursive update, in the situation above, we have when $\mathbf{x}_t \in \mathbb{R}$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \underbrace{\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_m}}_{1 \times n_m} \underbrace{\left[\frac{\partial \mathbf{x}_m}{\partial \mathbf{x}_k}\right]}_{n_m \times n_k}$$

- Therefore, each primitive only needs to define its vector-Jacobian product (VJP). The Jacobian $\left[\frac{\partial \mathbf{x}_m}{\partial \mathbf{x}_k}\right]$ is never explicitly built. It is usually simpler, faster, and more memory efficient to compute the VJP directly.
- Most reverse mode AD systems compose VJPs backward to compute $\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_1}$.

JVPs

Similarly, when $n_1 = 1$, the forward recursive update

$$\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_1} = \underbrace{\left[\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_l}\right]}_{n_k \times n_l} \underbrace{\frac{\partial \mathbf{x}_l}{\partial \mathbf{x}_1}}_{n_l \times 1}$$

is usually implemented in terms of Jacobian-vector products (JVP) locally defined at each primitive.

Checkpointing

Checkpointing consists in marking intermediate variables for which the forward values are not stored in memory. These are recomputed during the backward pass, which can save memory at the cost of recomputation.

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(100, 200)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(200, 10)

    def forward(self, x):
        x = checkpoint(self.layer1, x) # x is not stored
                                       # it will be recomputed
                                       # during the backward pass

        x = self.relu(x)
        x = self.layer2(x)
        return x
```

Higher-order derivatives

```
def tanh(x):
    y = jnp.exp(-2.0 * x)
    return (1.0 - y) / (1.0 + y)

fp = grad(tanh)
fpp = grad(grad(tanh))    # what sorcery is this?!
...
```

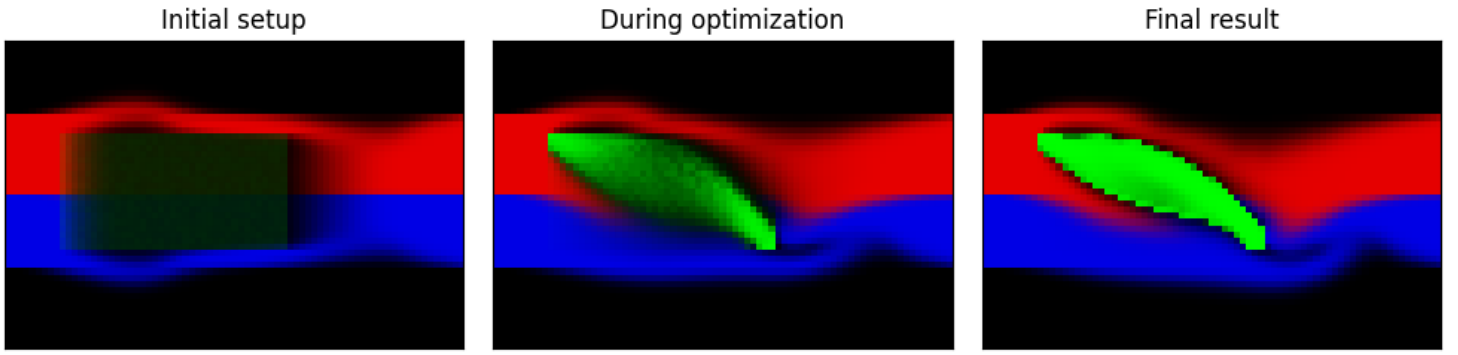
> The backward pass is itself a composition of primitives. Its execution can be traced, and reverse mode AD can run on its computational graph!

(demo of `code/lec3-autodiff.ipynb`)

AD beyond neural networks

You should be using automatic differentiation (Ryan Adams, 2016)

Differentiable simulation for system identification and visuomotor control
(Murthy Jatavallabhula et al, 2021)



Optimizing a wing (see `code/lec3-differentiable-fluid.ipynb`)

Summary

- Automatic differentiation is one of the keys that enabled the deep learning revolution.
- Backward mode automatic differentiation is more efficient when the function has more inputs than outputs.
- Applications of AD go beyond deep learning.

The end.

References

Slides from this lecture have been largely adapted from:

- Mathieu Blondel, Automatic differentiation, 2020.
- Gabriel Peyré, Course notes on Optimization for Machine Learning, 2020.