CWI

{PyTorch} \ {Deep Learning}: a short tutorial

- Accelerated numerics
- Automatic differentiation
- Inverse problems

5 Sep 2024

Alexander Skorikov

PyTorch: main selling points

- A wide range of (GPU accelerated) computational primitives
- Automatic differentiation engine with support of dynamic computational graphs
- Flexible programming model with little restrictions: porting standard Python/NumPy workflows and prototyping is very easy
- Significant industry support for development, optimization and maintenance
- Large community: lots of well-supported libraries available for different domains and applications
- After all, it is one of the most popular machine learning libraries, so integrating deep learning methods in your workflow is very convenient

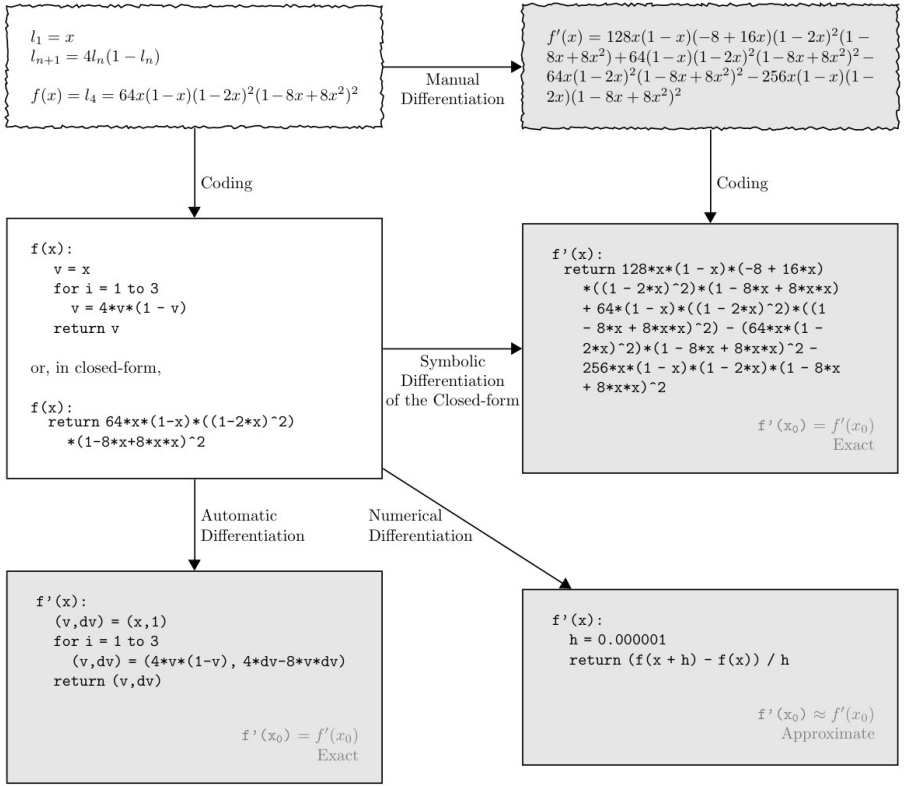
Interactive part 1: PyTorch basics

https://github.com/cicwi/mac-migs-tutorial

Automatic differentiation

is an efficient algorithm to compute derivatives of a function implemented as a sequence of computational steps.

Importantly, it **differs** from manual, symbolic and numerical differentiation, and produces **exact** results.



Baydin *et al.* // JMLR (2018)

CWI

Chain rule

$$y = f(g(h(x))) \longrightarrow \frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

Forward mode AD

Derivatives are computed in the same order as the computation:

$$y = f(g(h(x))) \longrightarrow \frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\left(\frac{\partial g}{\partial h}\left(\frac{\partial h}{\partial x}\right)\right)$$

Backward mode AD

Derivatives are computed in the reverse order compared to the computation:

$$y = f(g(h(x))) \longrightarrow \frac{\partial y}{\partial x} = \frac{\partial h}{\partial x}\left(\frac{\partial g}{\partial h}\left(\frac{\partial f}{\partial g}\right)\right)$$

Multiple dimensions

Jacobian-vector product (JVPs) – forward mode

$$\boldsymbol{F}'(\boldsymbol{x})\boldsymbol{v} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \vdots \\ \frac{\partial y_m}{\partial x_1} \end{bmatrix}$$

Vector-Jacobian products (VJPs) – backward mode

$$\boldsymbol{v}^T\boldsymbol{F}'(\boldsymbol{x}) = \begin{bmatrix} 1 & 0 \dots 0 \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \vdots \\ \frac{\partial y_1}{\partial x_n} \end{bmatrix}$$

Computational efficiency

Inputs $\gg$ outputs $\rightarrow$ backward mode is favoured

Reverse is also true

E.g. gradient of a scalar function $F : \mathbb{R}^n \rightarrow \mathbb{R}$

$$\boldsymbol{F}'(\boldsymbol{x})\boldsymbol{v} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \dots \frac{\partial y}{\partial x_n} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ \vdots \end{bmatrix} = \frac{\partial y}{\partial x_1} \qquad \text{(JVP × n times)}$$

$$\boldsymbol{v}^T \boldsymbol{F}'(\boldsymbol{x}) = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} \frac{\partial y}{\partial x_1} \dots \frac{\partial y}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{bmatrix} \qquad \text{(VJP × 1 time)}$$

CWI

Memory efficiency

Backward mode (VJP): need to store intermediate results.
See gradient checkpointing or invertible layers as methods to address this.

Hybrid mode

Mixed VJPs and JVPs: useful for computing Hessian-vector products.
See `torch.func.hessian`

CWI

Interactive part 2: Automatic differentiation

https://github.com/cicwi/mac-migs-tutorial

Inverse problems

$$K(u) = f$$

For a given f (measurement, observation), find u (underlying object, parameter of a system) satisfying the equation defined by K (model).

Applications:

- Computational imaging: CT, MRI, ultrasound, radioastronomy, …
- Image/signal processing: deconvolution, denoising, inpainting, …
- Machine learning and modeling
- …

Well- vs ill-posed inverse problems

$$K(u) = f$$

The problem is called well-posed if the solution u:

- Exists
- Is unique
- Is stable (small deviations in the observation lead to limited changes in the solution)

Linear discrete inverse problems

$$Ku = f^\delta, u \in \mathbb{R}^n, f^\delta \in \mathbb{R}^m, K \in \mathbb{R}^{m \times n}$$

There is a finite set of observations, and we seek for a solution using a discretized operator. For a linear model it can be represented as a matrix.

Let's analyze this problem following the exposition in:

https://tristanvanleeuwen.github.io/Masterclass_IP/linear_IP.slides.html#/2

Interactive part 3: Inverse problems

https://github.com/cicwi/mac-migs-tutorial

Summary

We've looked into parts of PyTorch that can be useful for your projects even if you are not doing deep learning, using inverse problems as an example of such an application. I hope you have found something interesting in this tutorial and that it will be useful for your research!

Suggested software to explore this topic further:

- DeepInverse (deepinv.github.io)
- PyXu (pyxu-org.github.io)