Automatic Differentiation

Alexandre Gramfort

Master 2 Data Science, Univ. Paris Saclay
Optimisation for Data Science
course based on slides from Mathieu Blondel.

Introduction

- Gradient-based training algorithms are the backbone of modern machine learning.
- Deriving gradients by hand is:
  - Tedious.
  - Error-prone.
  - Quickly infeasible for complex models.
  - But a very good exercise for master students!
- Key ingredients of deep learning:
  - GPUs.
  - Large datasets.
  - Automatic differentiation (autodiff).

What is Automatic Differentiation?

- Computes the derivatives of a function at a given point.
- Different from:
  - Numerical differentiation: Approximate results.
  - Symbolic differentiation: Produces human-readable expressions.
- In neural networks, reverse autodiff $=$ backpropagation.

Gradient

- For $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is:

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

- Coordinate-wise:

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

Numerical Gradient

- Finite difference approximation:

$$[\nabla f(x)]_j \approx \frac{f(x + \epsilon e_j) - f(x)}{\epsilon}$$

- Central finite difference:

$$[\nabla f(x)]_j \approx \frac{f(x + \epsilon e_j) - f(x - \epsilon e_j)}{2\epsilon}$$

- Requires n + 1 (2n) evaluations of f.
- Computationally expensive for high-dimensional x.
- Not very accurate for small ϵ.
- Available with `scipy.optimize.approx_fprime`
- What is used by the scipy function `scipy.optimize.check_grad` function.

Example of symbolic differentiation

Using the sympy library in Python:

```
import sympy as sp

# Define symbolic variables
x = sp.Symbol('x')
# Define a function
f = sp.sin(x) + x**2
# Compute derivatives
df_dx = sp.diff(f, x)  # Derivative with respect to x
# Print the result
print(df_dx)
# Output: 2*x + cos(x)
```

Automatic differentiation

- A program is defined as the composition of primitive operations that we know how to derive.
- The user can focus on the forward computation / model.

```python
import jax.numpy as jnp
from jax import grad, jit

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

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

grad_fun = jit(grad(loss_fun, argnums=0))
```

$\rightarrow$ See notebook.

Directional Derivative

- Derivative in the direction $v \in \mathbb{R}^n$:

$$D_v f(x) = \lim_{h \to 0} \frac{f(x + hv) - f(x)}{h}$$

- Interpretation: Rate of change of f in the direction v.
- Finite difference approximation:

$$D_v f(x) \approx \frac{f(x + \epsilon v) - f(x)}{\epsilon}$$

- Requires 2 evaluations of f (independent of n).

Directional Derivative

- The directional derivative is equal to the scalar product between the gradient and v, i.e.,

$$D_v f(x) = \nabla f(x) \cdot v$$

- **Proof:** Let $g(t) = f(x + tv)$. Then:

$$g'(t) = \lim_{h \to 0} \frac{f(x + (t+h)v) - f(x + tv)}{h}$$

At $t = 0$:

$$g'(0) = D_v f(x)$$

By the chain rule:

$$g'(t) = \nabla f(x + tv) \cdot v$$

Hence:

$$g'(0) = D_v f(x) = \nabla f(x) \cdot v$$

Jacobian

Definition

For

$$\begin{cases} f : \mathbb{R}^n & \to \mathbb{R}^m \\ \quad x & \mapsto (f_1(x), \dots, f_m(x)) \end{cases}$$

the Jacobian matrix is:

$$J_f(x) = \frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right] = \begin{bmatrix} \nabla f_1(x)^\top \\ \vdots \\ \nabla f_m(x)^\top \end{bmatrix}$$

- The gradient is the transpose of the Jacobian for $m = 1$ (a “wide” Jacobian).

Jacobian-Vector Product (JVP)

The Jacobian-vector product is computed as:

$$J_f(x)v = \begin{bmatrix} \nabla f_1(x)^\top \\ \vdots \\ \nabla f_m(x)^\top \end{bmatrix} v = \begin{bmatrix} \nabla f_1(x) \cdot v \\ \vdots \\ \nabla f_m(x) \cdot v \end{bmatrix} \in \mathbb{R}^m$$

Finite Difference Approximation

$$J_f(x)v \approx \frac{f(x + \epsilon v) - f(x)}{\epsilon}$$

Requires only 2 function calls for (central) finite difference.

Vector-Jacobian Product (VJP)

The vector-Jacobian product is computed as:

$$u^\top J_f(x) = u^\top \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right] = \left[u \cdot \frac{\partial f}{\partial x_1}, \ldots, u \cdot \frac{\partial f}{\partial x_n}\right] \in \mathbb{R}^n$$

Finite Difference Approximation

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + \epsilon e_i) - f(x)}{\epsilon}$$

Requires $n + 1$ function calls for finite difference, or $2n$ for central finite difference.

Chain Rule

- Let $f(x) = h(g(x)) = h \circ g(x)$, where $h, g : \mathbb{R} \to \mathbb{R}$. Then,

$$f'(x) = h'(g(x))g'(x)$$

- Alternatively, let $y = g(x)$ and $z = h(y)$, then

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = \left.\frac{\partial z}{\partial y}\right|_{y=g(x)} \left.\frac{\partial y}{\partial x}\right|_{x=x}$$

- Let $f(x) = h(g(x))$, where $g : \mathbb{R}^n \to \mathbb{R}^d$ and $h : \mathbb{R}^d \to \mathbb{R}$. Then,

$$\underbrace{\nabla f(x)}_{n\times 1} = (\underbrace{\nabla h(g(x))^\top}_{1\times d} \underbrace{J_g(x))^\top}_{d\times n} = \underbrace{J_g(x)^\top}_{n\times d} \underbrace{\nabla h(g(x))}_{d\times 1}$$

Chain Rule

- Assume $f \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ decomposes as follows:

$$\begin{aligned} o &= f(x) \\ &= f_4 \circ f_3 \circ f_2 \circ f_1(x) \\ &= f_4\left(f_3\left(f_2\left(f_1(x)\right)\right)\right) \end{aligned}$$

- where $f_1 : \mathbb{R}^n \rightarrow \mathbb{R}^{m_1}, f_2 : \mathbb{R}^{m_1} \rightarrow \mathbb{R}^{m_2}, \ldots, f_4 : \mathbb{R}^{m_3} \rightarrow \mathbb{R}^m$.
- How to compute the Jacobian $J_f(x) = \frac{\partial o}{\partial x} \in \mathbb{R}^{m \times n}$ efficiently?

Chain Rule

- Sequence of operations

$$\begin{aligned} x_1 &= x \\ x_2 &= f_1(x_1) \\ x_3 &= f_2(x_2) \\ x_4 &= f_3(x_3) \\ o &= f_4(x_4) \end{aligned}$$

- By the chain rule, we have:

$$\begin{aligned} \frac{\partial o}{\partial x} &= \frac{\partial o}{\partial x_4}\frac{\partial x_4}{\partial x_3}\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x} \\ &= \frac{\partial f_4(x_4)}{\partial x_4}\frac{\partial f_3(x_3)}{\partial x_3}\frac{\partial f_2(x_2)}{\partial x_2}\frac{\partial f_1(x)}{\partial x} \\ &= J_{f_4}(x_4) J_{f_3}(x_3) J_{f_2}(x_2) J_{f_1}(x) \end{aligned}$$

Forward Differentiation

- Recall that $\frac{\partial f}{\partial x_j} \in \mathbb{R}^m$ is the j^{th} column of $J_f(x)$
- Jacobian vector product (JVP) with $e_j \in \mathbb{R}^n$ extracts the f^{th} column

$$
\begin{aligned}
J_f(x)e_1 &= \frac{\partial f}{\partial x_1} \\
J_f(x)e_2 &= \frac{\partial f}{\partial x_2} \\
&\vdots \\
J_f(x)e_n &= \frac{\partial f}{\partial x_n}
\end{aligned}
$$

- Computing a gradient (m=1) requires n JVPs with $e_1, \ldots, e_n$.

Forward Differentiation

- Jacobian-vector product with $v \in \mathbb{R}^n$

$$J_f(x)v = \underbrace{J_{f_4}(x_4)}_{m \times m_3} \underbrace{J_{f_3}(x_3)}_{m_3 \times m_2} \underbrace{J_{f_2}(x_2)}_{m_2 \times m_1} \underbrace{J_{f_1}(x)}_{m_1 \times n} v$$

Multiplication from right to left.

- Cost of computing n JVPs:

$$n\left(mm_3 + m_3 m_2 + m_2 m_1 + m_1 n\right)$$

- Cost of computing a gradient $(m = 1, m_3 = m_2 = m_1 = n)$:

$$O\left(n^3\right)$$

Forward Differentiation

- $o = f(\mathrm{x}) = f_K \circ \cdots \circ f_2 \circ f_1(\mathrm{x})$
- $[J_f(x)]_{:j} = J_{f_K}(x_K) \ldots J_{f_2}(x_2) J_{f_1}(x) e_j \quad j \in \{1, \ldots, n\}$

Require: $x \in \mathbb{R}^n$

1: $x_1 \leftarrow x$
2: $v_j \leftarrow e_j \in \mathbb{R}^n \quad j \in \{1, \ldots, n\}$
3: **for** $k = 1$ to K **do**
4: $\quad x_{k+1} \leftarrow f_k(x_k)$
5: $\quad v_j \leftarrow J_{f_k}(x_k) v_j \quad j \in \{1, \ldots, n\}$
6: **end for**
7: **return** $o = x_{K+1}, [J_f(x)]_{:j} = v_j \quad j \in \{1, \ldots, n\}$

Backward Differentiation (a.k.a. Reverse Mode)

- Recall that $\nabla_i(x)^\top \in \mathbb{R}^n$ is the i^{th} row of $J_f(x)$.
- Vector Jacobian product with $e_i \in \mathbb{R}^m$ extracts the i^{th} row:

$$e_i^\top J_f(x) = \nabla f_i(x)^\top$$

- Computing the gradient (m=1) requires only 1 VJP with $e_1 \in \mathbb{R}^1$.

Backward Differentiation

- Vector Jacobian product with $u \in \mathbb{R}^m$

$$u^\top \underbrace{J_{t_4}(x_4)}_{m \times m_3} \underbrace{J_{f_3}(x_3)}_{m_3 \times m_2} \underbrace{J_{t_2}(x_2)}_{m_2 \times m_1} \underbrace{J_{f_1}(x)}_{m_1 \times n}$$

  Multiplication from left to right.

- Cost of computing m VJPs:

$$m\,(mm_3 + m_3 m_2 + m_2 m_1 + m_1 n)$$

- Cost of computing a gradient ($m = 1, m_3 = m_2 = m_1 = n$):

$$O\left(n^2\right)$$

- It is more efficient than forward differentiation if $m = 1$ (for $m < n$).

Algorithm: Backward Differentiation

- $o = f_K \circ \cdots \circ f_1(x)$
- $[J_f(x)]_{i,:} = e_i^\top J_{f_K}(x_K) \dots J_{f_1}(x) \quad i \in \{1, \dots, m\}.$

Require: $x \in \mathbb{R}^n$

1: $x_1 \leftarrow x$ $u_1 \leftarrow e_i \in \mathbb{R}^m \quad i \in \{1, \dots, m\}$
2: **for** $k = 1$ to K **do**
3: $\quad x_{k+1} \leftarrow f_k(x_k)$ (Store the intermediate results)
4: **end for**
5: **for** $k = K$ to 1 **do**
6: $\quad u_i^\top \leftarrow u_i^\top J_{f_k}(x_k) \quad i \in \{1, \dots, m\}$ (Iterate from K to 1)
7: **end for**
8: **return** $o = x_{K+1}, [J_f(x)]_{i,:} = u_i^\top \quad i \in \{1, \dots, m\}$

Remark: You trade computation for memory as you need to store the intermediate results.

Examples of VJPs

Let $W \in \mathbb{R}^{a \times b}, u \in \mathbb{R}^a, x \in \mathbb{R}^b$.

- For $f(x) = g(x)$ element-wise:
  - f maps $\mathbb{R}^b$ to $\mathbb{R}^b$.

$$J_f(x) = \mathrm{diag}(g'(x)) \in \mathbb{R}^{b \times b}$$

  - VJP:

$$u^\top J_f(x) = u * g'(x) \qquad (* \text{ means element-wise multiplication}).$$

- For $f(x) = Wx$:
  - f maps $\mathbb{R}^b$ to $\mathbb{R}^a$.
  - $J_f(x) = W$ maps $\mathbb{R}^b$ to $\mathbb{R}^a$.
  - VJP:

$$u^\top J_f(x) = W^\top u \in \mathbb{R}^b$$

- For $f(W) = Wx$?

Summary: Forward vs. Backward Differentiation

Forward Differentiation

- Uses Jacobian-Vector Products (JVPs).
- Efficient for tall Jacobians ($m \geq n$).
- Does not store intermediate computations.

Backward Differentiation

- Uses Vector-Jacobian Products (VJPs).
- Efficient for wide Jacobians ($m \leq n$).
- Stores intermediate computations.

References

Two minimalist implementations of autodiff:

- Autodidact, by Matthew Johnson.
  `https://github.com/mattjj/autodidact`
- Micrograd, by Andrej Karpathy.
  `https://github.com/karpathy/micrograd`