Automatic differentiation

Mathieu Blondel

February 12, 2021
Gradient-based learning

- Gradient-based training algorithms are the workhorse of modern machine learning.
- Deriving gradients by hand is tedious and error prone.
- This becomes quickly infeasible for complex models.
- Changes to the model require rederiving the gradient.
- Deep learning = GPU + data + autodiff
Automatic differentiation

- Evaluates the derivatives of a function at a given point.
- Not the same as numerical differentiation.
- Not the same as symbolic differentiation, which returns a “human-readable” expression.
- In a neural network context, reverse autodiff is often known as backpropagation.
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):
    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.sum((preds - targets)**2)

grad_fun = jit(grad(loss_fun))
```
Modern frameworks support higher-order derivatives

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

fp = grad(tanh)
fpp = grad(grad(tanh))
...```

Automatic differentiation
Outline

1 Numerical differentiation
2 Chain compositions
3 Computational graphs
4 Implementation
5 Advanced topics
6 Conclusion
Derivatives

- Definition of derivative of \( g: \mathbb{R} \rightarrow \mathbb{R} \):

\[
g'(a) = \frac{\partial g(a)}{\partial a} = \lim_{h \to 0} \frac{g(a + h) - g(a)}{h}
\]

- \( g'(a) \) is called Lagrange notation.

- \( \frac{\partial g(a)}{\partial a} \) is called Leibniz notation.

- Interpretations: instantaneous rate of change of \( g \), slope of the tangent of \( g \) at \( a \).
Gradient

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

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

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

- Applying the definition of derivative coordinate-wise:

$$[\nabla f(x)]_j = \frac{\partial f}{\partial x_j}(x) = \lim_{h \rightarrow 0} \frac{f(x + he_j) - f(x)}{h} \quad j \in \{1, \ldots, n\}$$

where $e_j = [0, 0, \ldots, 0, 1, 0, \ldots, 0]^\top \in \{0, 1\}^n$ is the $j^{th}$ standard basis vector.
Numerical gradient

- Finite difference:

\[
[\nabla f(\mathbf{x})]_j = \frac{\partial f}{\partial x_j}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \varepsilon \mathbf{e}_j) - f(\mathbf{x})}{\varepsilon} \quad j \in \{1, \ldots, n\}
\]

where \( \varepsilon \) is a small value (e.g., \( 10^{-6} \)).

- Central finite difference:

\[
[\nabla f(\mathbf{x})]_j = \frac{\partial f}{\partial x_j}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \varepsilon \mathbf{e}_j) - f(\mathbf{x} - \varepsilon \mathbf{e}_j)}{2\varepsilon} \quad j \in \{1, \ldots, n\}
\]

- Computing \( \nabla f(\mathbf{x}) \) approximately by (central) finite difference is \( n + 1 \) times (\( 2n \) times) as costly as evaluating \( f \).
Directional derivative

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

\[
D_v f(x) = \lim_{h \to 0} \frac{f(x + hv) - f(x)}{h} \in \mathbb{R}
\]

- Interpretation: rate of change of $f$ in the direction of $v$, when moving away from $x$.

- $[\nabla f(x)]_i$ is the derivative in the direction of $e_i$.

- Finite difference (and similarly for the central finite difference):

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

Only 2 calls to $f$ are needed, i.e., independent of $n$. 
Directional derivative

- **Fact.** The directional derivative is equal to the scalar product between the gradient and $\mathbf{v}$, i.e.,

$$D_{\mathbf{v}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$$

- **Proof.** Let $g(t) = f(\mathbf{x} + t\mathbf{v})$. We have

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

and therefore $g'(0) = D_{\mathbf{v}}(\mathbf{x})$. By the chain rule, we also have

$$g'(t) = \nabla f(\mathbf{x} + t\mathbf{v}) \cdot \mathbf{v}.$$ 

Hence, $g'(0) = D_{\mathbf{v}}(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$. 

Mathieu Blondel
Automatic differentiation
The Jacobian of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$J_f(x) = \frac{\partial f(x)}{\partial x} = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n}
\end{bmatrix}
= \begin{bmatrix}
\frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n}
\end{bmatrix}
= \begin{bmatrix}
\nabla f_1(x)^T \\
\vdots \\
\nabla f_m(x)^T
\end{bmatrix}$$

The size of the Jacobian matrix is $m \times n$.

The gradient’s transpose is thus a “wide” Jacobian ($m = 1$).
Jacobian vector product ("JVP")

- Right-multiply the Jacobian with a vector \( \mathbf{v} \in \mathbb{R}^n \)

\[
J_f(\mathbf{x})\mathbf{v} = \begin{bmatrix}
\nabla f_1(\mathbf{x})^\top \\
\vdots \\
\nabla f_m(\mathbf{x})^\top
\end{bmatrix} \mathbf{v} \\
= \begin{bmatrix}
\nabla f_1(\mathbf{x}) \cdot \mathbf{v} \\
\vdots \\
\nabla f_m(\mathbf{x}) \cdot \mathbf{v}
\end{bmatrix}
\]

\[
= \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}
\]

- Finite difference (and similarly for the central finite difference):

\[
J_f(\mathbf{x})\mathbf{v} \approx \frac{f(\mathbf{x} + \varepsilon \mathbf{v}) - f(\mathbf{x})}{\varepsilon}
\]

- Computing the JVP approximately by (central) finite difference requires only 2 calls to \( f \).
Vector Jacobian Product (“VJP”)

- Left-multiply the Jacobian with a vector $u \in \mathbb{R}^m$

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

- Finite difference (and similarly for the central finite difference):

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

- Computing the VJP approximately by (central) finite difference requires $n + 1$ calls ($2n$ calls) to $f$. 

Mathieu Blondel
Automatic differentiation
Chain rule

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

- Alternatively, let $y = g(x)$ and $z = f(y)$, then
  $$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} = \frac{\partial z}{\partial y} \bigg|_{y=g(x)} \frac{\partial y}{\partial x} \bigg|_{x=x}$$

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

- and similarly using Leibniz notation
Assume $f: \mathbb{R}^n \to \mathbb{R}^m$ decomposes as follows:

$$o = f(x) = f_4 \circ f_3 \circ f_2 \circ f_1(x) = f_4(f_3(f_2(f_1(x))))$$

where $f_1: \mathbb{R}^n \to \mathbb{R}^{m_1}$, $f_2: \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}$, ..., $f_4: \mathbb{R}^{m_3} \to \mathbb{R}^m$.

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

\[
x_1 = x \\
x_2 = f_1(x_1) \\
x_3 = f_2(x_2) \\
x_4 = f_3(x_3) \\
\circ = f_4(x_4)
\]

- By the chain rule, we have

\[
\frac{\partial \circ}{\partial x} = \frac{\partial \circ}{\partial x_4} \frac{\partial x_4}{\partial x_3} \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial x} \\
= \frac{\partial f_4(x_4)}{\partial x_4} \frac{\partial f_3(x_3)}{\partial x_3} \frac{\partial f_2(x_2)}{\partial x_2} \frac{\partial f_1(x)}{\partial x} \\
= J_{f_4}(x_4) J_{f_3}(x_3) J_{f_2}(x_2) J_{f_1}(x)
\]
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 $j$th column

  $J_f(x)e_1 = \frac{\partial f}{\partial x_1}$

  $J_f(x)e_2 = \frac{\partial f}{\partial x_2}$

  $\vdots$

  $J_f(x)e_n = \frac{\partial f}{\partial x_n}$

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

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

\[
\mathbf{J}_f(x) v = \underbrace{\mathbf{J}_{f_4}(x_4)}_{m \times m_3} \underbrace{\mathbf{J}_{f_3}(x_3)}_{m_3 \times m_2} \underbrace{\mathbf{J}_{f_2}(x_2)}_{m_2 \times m_1} \underbrace{\mathbf{J}_{f_1}(x)}_{m_1 \times n} v
\]

Multiplication from right to left is more efficient.

- Cost of computing \( n \) JVPs:

\[
n(m m_3 + m_3 m_2 + m_2 m_1 + m_1 n)
\]

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

\[
O(n^3)
\]
Forward differentiation

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

Algorithm 1 Compute \( o = f(x) \) and \( J_f(x) \) alongside

1: **Input:** \( x \in \mathbb{R}^n \)
2: \( x_1 \leftarrow x \)
3: \( v_j \leftarrow e_j \in \mathbb{R}^n \quad j \in \{1, \ldots, n\} \)
4: **for** \( k = 1 \) to \( K \) **do**
5: \( x_{k+1} \leftarrow f_k(x_k) \)
6: \( v_j \leftarrow J_{f_k}(x_k)v_j \quad j \in \{1, \ldots, n\} \)
7: **end for**
8: **Returns:** \( o = x_{K+1} \), \( [J_f(x)]_{:,j} = v_j \quad j \in \{1, \ldots, n\} \)
Recall that $\nabla f_i(x)^\top \in \mathbb{R}^n$ is the $i^{\text{th}}$ row of $J_f(x)$.

Vector Jacobian product (VJP) with $e_i \in \mathbb{R}^m$ extracts the $i^{\text{th}}$ row

$$e_1^\top J_f(x) = \nabla f_1(x)^\top$$

$$e_2^\top J_f(x) = \nabla f_2(x)^\top$$

$$\vdots$$

$$e_m^\top J_f(x) = \nabla f_m(x)^\top$$

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

Backward differentiation

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

$$u^\top \begin{bmatrix} J_{f_4}(x_4) & J_{f_3}(x_3) & J_{f_2}(x_2) & J_{f_1}(x) \end{bmatrix}$$

Multiplication from left to right is more efficient.

- Cost of computing $m$ VJPs:

$$m(mm_3 + m_3m_2 + m_2m_1 + m_1n)$$

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

$$O(n^2)$$
Backward differentiation

- $o = f(x) = f_K \circ \cdots \circ f_2 \circ f_1(x)$
- $[J_f(x)]_{i,:} = e_i^\top J_{f_K}(x_K) \cdots J_{f_2}(x_2)J_{f_1}(x) \quad i \in \{1, \ldots, m\}$

Algorithm 2 Compute $o = f(x)$ and $J_f(x)$

1: **Input:** $x \in \mathbb{R}^n$
2: $x_1 \leftarrow x$, $u_i \leftarrow e_i \in \mathbb{R}^m \quad i \in \{1, \ldots, m\}$
3: for $k = 1$ to $K$ do
4: $x_{k+1} \leftarrow f_k(x_k)$
5: end for
6: for $k = K$ to 1 do
7: $u_i^\top \leftarrow u_i^\top J_{f_k}(x_k) \quad i \in \{1, \ldots, m\}$
8: end for
9: **Returns:** $o = x_{K+1}$, $[J_f(x)]_{i,:} = u_i^\top \quad i \in \{1, \ldots, m\}$
Feedforward networks

Each function can now have two arguments: $f_k(x_k, \theta_k)$, where $x_k$ is the previous output and $\theta_k$ are learnable parameters.

Example one hidden layer, one output layer, squared loss

$$f = f_4 \circ \cdots \circ f_1$$

$$x_2 = f_1(x, W_1) = W_1 x$$

$$x_3 = f_2(x_2, \emptyset) = \text{relu}(x_2)$$

$$x_4 = f_3(x_3, W_3) = W_3 x_3$$

$$o = f_4(x_4, y) = \frac{1}{2} \|x_4 - y\|^2$$
Feedforward network example

\[
x = x_1 \rightarrow f_1 \rightarrow x_2 \rightarrow f_2 \rightarrow x_3 \rightarrow f_3 \rightarrow x_4 \rightarrow f_4 \rightarrow o
\]

- Applying the chain rule once again we have

\[
\frac{\partial o}{\partial \theta_4} = \frac{\partial o}{\partial x_4} \frac{\partial x_4}{\partial \theta_3} \frac{\partial \theta_3}{\partial \theta_2} \frac{\partial \theta_2}{\partial \theta_1}
\]

\[
\frac{\partial o}{\partial \theta_3} = \frac{\partial o}{\partial x_4} \frac{\partial x_4}{\partial \theta_3}
\]

\[
\frac{\partial o}{\partial \theta_2} = \frac{\partial o}{\partial x_4} \frac{\partial x_4}{\partial \theta_3} \frac{\partial x_3}{\partial \theta_2}
\]

\[
\vdots
\]

- Apart from the last multiplication, the Jacobians \( \frac{\partial o}{\partial x_k} \) and \( \frac{\partial o}{\partial \theta_k} \) share the same computations!
Algorithm 3 Compute $o = f(x, \theta_1, \ldots, \theta_K)$ and its Jacobians.

1: **Input:** $x \in \mathbb{R}^n$, $\theta_1, \ldots, \theta_K$
2: $x_1 \leftarrow x$
3: $u_i \leftarrow e_i \in \mathbb{R}^m \quad i \in \{1, \ldots, m\}$
4: **for** $k = 1$ to $K$ **do**
5: \hspace{1em} $x_{k+1} \leftarrow f_k(x_k, \theta_k)$
6: **end for**
7: **for** $k = K$ to $1$ **do**
8: \hspace{1em} $j_{i,k} \leftarrow u_i^\top \frac{\partial f_k(x_k, \theta_k)}{\partial \theta_k} \quad i \in \{1, \ldots, m\}$
9: \hspace{1em} $u_i^\top \leftarrow u_i^\top \frac{\partial f_k(x_k, \theta_k)}{\partial x_k} \quad i \in \{1, \ldots, m\}$
10: **end for**
11: **Returns:** $o = x_{K+1}$, $[\frac{\partial o}{\partial x}]_{i,:} = u_i^\top$, $[\frac{\partial o}{\partial \theta_k}]_{i,:} = j_{i,k} \quad i \in \{1, \ldots, m\}, \quad k \in \{1, \ldots, K\}$
Examples of VJPs

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

- $f(x) = g(x)$ (element-wise)
  - $f$ maps $\mathbb{R}^b$ to $\mathbb{R}^b$
  - $J_f(x) = J_f(x)^\top = \text{diag}(g'(x))$ maps $\mathbb{R}^b$ to $\mathbb{R}^b$, i.e., $b \times b$ matrix
  - $u^\top J_f(x) = J_f(x)^\top u = u \ast g'(x) \in \mathbb{R}^b$, where $\ast$ means element-wise multiplication

- $f(x) = Wx$
  - $f$ maps $\mathbb{R}^b$ to $\mathbb{R}^a$
  - $J_f(x) = W$ maps $\mathbb{R}^b$ to $\mathbb{R}^a$, i.e., $a \times b$ matrix
  - $J_f(x)^\top = W^\top$ maps $\mathbb{R}^a$ to $\mathbb{R}^b$, i.e., $b \times a$ matrix
  - $u^\top J_f(x) = J_f(x)^\top u = W^\top u \in \mathbb{R}^b$
Examples of VJPs

- $f(W) = Wx$
  - $f$ maps $\mathbb{R}^{a \times b}$ to $\mathbb{R}^a$
  - $J_f(W)$ maps $\mathbb{R}^{a \times b}$ to $\mathbb{R}^a$, i.e., $a \times (a \times b)$ matrix
  - $J_f(W)^\top$ maps $\mathbb{R}^a$ to $\mathbb{R}^{a \times b}$, i.e., $(a \times b) \times a$ matrix
  - $J_f(W)^\top u = ux^\top$

VJPs make things easier when dealing with matrix or tensor inputs.
Summary: Forward vs. Backward

- **Forward**
  - Uses Jacobian vector products (JVPs)
  - Each JVP call builds one column of the Jacobian
  - Efficient for tall Jacobians \((m \geq n)\)
  - Need not store intermediate computations

- **Backward**
  - Uses vector Jacobian products (VJPs)
  - Each VJP call builds one row of the Jacobian
  - Efficient for wide matrices \((m \leq n)\)
  - Needs to store intermediate computations
Most objectives in machine learning can be written in the form

\[ \min_{x \in \mathbb{R}^n} f(x) = \sum_{i=1}^{N} \ell_i(f_i(x)) \]

where \( f : \mathbb{R}^n \rightarrow \mathbb{R}^M \) and \( \ell_i : \mathbb{R}^M \rightarrow \mathbb{R} \).

The minimization needs to be w.r.t. a scalar valued loss.

This corresponds to the \( m = 1 \) setting, for which backward differentiation is more efficient.

This explains the immense success of reverse autodiff in machine learning.
Outline

1. Numerical differentiation
2. Chain compositions
3. Computational graphs
4. Implementation
5. Advanced topics
6. Conclusion
Computational graph

\[ f(x_1, x_2) = x_2 e^{x_1} \sqrt{x_1 + x_2 e^{x_1}} \]

- Operations in topological order
  
  \[ x_3 = f_3(x_1) = e^{x_1} \]
  
  \[ x_4 = f_4(x_2, x_3) = x_2 x_3 \]
  
  \[ x_5 = f_5(x_1, x_4) = x_1 + x_4 \]
  
  \[ x_6 = f_6(x_5) = \sqrt{x_5} \]
  
  \[ x_7 = f_7(x_4, x_6) = x_4 x_6 \]

- Directed acyclic graph traversal

[Diagram showing the directed acyclic graph with nodes labeled as \( f_3, f_4, f_5, f_6, f_7 \) and edges connecting them in the order of operations.]
Forward differentiation example

\[ x_1 \rightarrow f_3 \rightarrow f_5 \rightarrow f_6 \rightarrow f_7 \rightarrow x_7 = 0 \]

- \( x_4 \) is influenced by \( x_3 \) and \( x_2 \), therefore

\[ \frac{\partial x_4}{\partial x_1} = \frac{\partial x_4}{\partial x_3} \frac{\partial x_3}{\partial x_1} + \frac{\partial x_4}{\partial x_2} \frac{\partial x_2}{\partial x_1} \]

- \( x_7 \) is influenced by \( x_4 \) and \( x_6 \), therefore

\[ \frac{\partial x_7}{\partial x_1} = \frac{\partial x_7}{\partial x_4} \frac{\partial x_4}{\partial x_1} + \frac{\partial x_7}{\partial x_6} \frac{\partial x_6}{\partial x_1} \]
Forward differentiation example

- Recurse in topological order

\[
\frac{\partial x_1}{\partial x_1} = \text{Id}_n \\
\frac{\partial x_2}{\partial x_2} = \text{Id}_n \\
\frac{\partial x_3}{\partial x_1} = \frac{\partial x_3}{\partial x_1} \frac{\partial x_1}{\partial x_1} \\
\frac{\partial x_4}{\partial x_1} = \frac{\partial x_4}{\partial x_3} \frac{\partial x_3}{\partial x_1} + \frac{\partial x_4}{\partial x_2} \frac{\partial x_2}{\partial x_1} \\
\vdots
\]

- Everything can be computed in terms of JVPs
Forward differentiation

In the general case, we have

$$\frac{\partial x_j}{\partial x_1} = \sum_{i \in \text{Parents}(j)} \frac{\partial x_j}{\partial x_i} \frac{\partial x_i}{\partial x_1}$$

- $\frac{\partial x_j}{\partial x_i}$ is easy to compute as $f_j$ is a direct function of $x_i$.
- $\frac{\partial x_i}{\partial x_1}$ is obtained from the previous iterations in topological order.
Backward differentiation example

\[
\begin{align*}
\frac{\partial o}{\partial x_5} &= \frac{\partial o}{\partial x_6} \frac{\partial x_6}{\partial x_5} \\
\frac{\partial o}{\partial x_4} &= \frac{\partial o}{\partial x_5} \frac{\partial x_5}{\partial x_4} + \frac{\partial o}{\partial x_7} \frac{\partial x_7}{\partial x_4}
\end{align*}
\]

- \(x_5\) influences only \(x_6\), therefore
- \(x_4\) influences \(x_5\) and \(x_7\), therefore
Backward differentiation example

- Recurse in reverse topological order

\[
\frac{\partial o}{\partial x_7} = \frac{\partial x_7}{\partial x_7} = 1d_m \\
\frac{\partial o}{\partial x_6} = \frac{\partial o}{\partial x_7} \frac{\partial x_7}{\partial x_6} \\
\frac{\partial o}{\partial x_5} = \frac{\partial o}{\partial x_6} \frac{\partial x_6}{\partial x_5} \\
\frac{\partial o}{\partial x_4} = \frac{\partial o}{\partial x_5} \frac{\partial x_5}{\partial x_4} + \frac{\partial o}{\partial x_7} \frac{\partial x_7}{\partial x_4} \\
\vdots
\]

- Everything can be computed in terms of VJPs
In the general case, we have

\[
\frac{\partial o}{\partial x_j} = \sum_{k \in \text{Children}(j)} \frac{\partial o}{\partial x_k} \frac{\partial x_k}{\partial x_j}
\]

\(\frac{\partial o}{\partial x_k}\) is obtained from previous iterations (reverse topological order) and is known as “adjoint”.

\(\frac{\partial x_k}{\partial x_j}\) is easy to compute as \(f_k\) is a direct function of \(x_j\).
Outline

1. Numerical differentiation
2. Chain compositions
3. Computational graphs
4. Implementation
5. Advanced topics
6. Conclusion
Obtaining the computational graph

- **Ahead of time**
  - Read from source or abstract syntax tree (AST). Ex: **Tangent**.
  - API for composing primitive operations (the graph is fully built before the program is evaluated). Ex: Tensorflow.

- **Just in time**
  - Tracing: monitor the program execution (the graph is built while the program is being executed). Ex: Tensorflow Eager, **JAX**, PyTorch.

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

def add(a, b):
    return a + b

a = jnp.array([1, 2, 3])
b = jnp.array([4, 5, 6])

print(grad(add)(a, b))
```
Key components of an implementation

- VJP for all primitive operations
- Node class
- Topological sort
- Forward pass
- Backward pass

We will now briefly review each component using a rudimentary implementation (link to code).
VJPs for primitive operations

```python
def dot(x, W):
    return np.dot(W, x)

def dot_make_vjp(x, W):
    def vjp(u):
        return W.T.dot(u), np.outer(u, x)
    return vjp

dot.make_vjp = dot_make_vjp

def add(a, b):
    return a + b

def add_make_vjp(a, b):
    gprime = np.ones(len(a))

    def vjp(u):
        return u * gprime, u * gprime
    return vjp

add.make_vjp = add_make_vjp
```
class Node(object):

    def __init__(self, value=None, func=None, parents=None, name=""):
        # Value stored in the node.
        self.value = value
        # Function producing the node.
        self.func = func
        # Inputs to the function.
        self.parents = [] if parents is None else parents
        # Unique name of the node (for debugging and hashing).
        self.name = name
        # Gradient / Jacobian.
        self.grad = 0
        if not name:
            raise ValueError("Each node must have a unique name.")

    def __hash__(self):
        return hash(self.name)

    def __repr__(self):
        return "Node(%s)" % self.name
A good implementation would support tracing, instead of building the DAG manually.
Topological sort

```python
def dfs(node, visited):
    visited.add(node)
    for parent in node.parents:
        if not parent in visited:
            # Yield parent nodes first.
            yield from dfs(parent, visited)
            # And current node later.
    yield node

def topological_sort(end_node):
    visited = set()
    sorted_nodes = []

    # All non-visited nodes reachable from end_node.
    for node in dfs(end_node, visited):
        sorted_nodes.append(node)

    return sorted_nodes
```

Mathieu Blondel

Automatic differentiation
def evaluate_dag(sorted_nodes):
    for node in sorted_nodes:
        if node.value is None:
            values = [p.value for p in node.parents]
            node.value = node.func(*values)
    return sorted_nodes[-1].value
def backward_diff_dag(sorted_nodes):
    value = evaluate_dag(sorted_nodes)
    m = value.shape[0]  # Output size

    # Initialize recursion.
    sorted_nodes[-1].grad = np.eye(m)

    for node_k in reversed(sorted_nodes):
        if not node_k.parents:
            # We reached a node without parents.
            continue

        # Values of the parent nodes.
        values = [p.value for p in node_k.parents]

        # Iterate over outputs.
        for i in range(m):
            # A list of size len(values) containing the vjps.
            vjps = node_k.func.make_vjp(*values)(node_k.grad[i])

            for node_j, vjp in zip(node_k.parents, vjps):
                node_j.grad += vjp

    return sorted_nodes
Checkpointing (best seen in presentation mode)

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.

![Diagram showing checkpointing process](image-url)
Checkpointing (best seen in presentation mode)

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.
Checkpointing (best seen in presentation mode)

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.

\[ x \xrightarrow{f_1} f_2 \xrightarrow{f_3} f_4 \xrightarrow{f_5} f_6 \xrightarrow{o} \]
Checkpointing

During the forward pass, save computations at intermediate locations only (checkpoints).

During the backward pass, recompute other locations on the fly, starting from the checkpoints.

Tradeoff between memory and computation time.
During the forward pass, save computations at intermediate locations only (checkpoints).

During the backward pass, recompute other locations on the fly, starting from the checkpoints.

Tradeoff between memory and computation time.

Mathieu Blondel
Automatic differentiation
Checkpointing (best seen in presentation mode)

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.
Checkpointing (best seen in presentation mode)

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.
Checkpointing (best seen in presentation mode)

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.
Checkpointing

During the forward pass, save computations at intermediate locations only (checkpoints).

During the backward pass, recompute other locations on the fly, starting from the checkpoints.

Tradeoff between memory and computation time.
JAX

- NumPy and SciPy compatible
- Automatic differentiation (grad)
- Just-in-time compilation (jit)
- Automatic vectorization (vmap)
- Code transformations are composable
- Actively developed by Google
- Gaining a lot of popularity among ML and science researchers
The matrix gathering second-order derivatives

\[ \nabla^2 f = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{bmatrix} \]

Hessian vector product = gradient of directional derivative

\[ \nabla^2 f(\mathbf{x}) \mathbf{v} = \nabla(\nabla f(\mathbf{x}) \cdot \mathbf{v}) \]

JAX supports fully closed tracing: we can “trace through tracing”
Recovering JVPs from VJPs

- Suppose we already have a VJP routine for computing $u^\top J_f(x)$
- By linearity we have
  \[ \frac{\partial u^\top J_f(x)}{\partial u} = J_f(x)^\top \]
- and therefore
  \[ v^\top \frac{\partial u^\top J_f(x)}{\partial u} = v^\top J_f(x)^\top = (J_f(x)v)^\top \]
- The VJP w.r.t. $u$ of the VJP w.r.t. $x$ is equal to the transpose of the JVP w.r.t. $x$.
- The trick does not work in the other direction!
Differentiating min problems

- Consider the function

\[ f(\theta) = \min_x E(x, \theta) = E(x^*(\theta), \theta) \]

- From Danskin’s theorem (a.k.a. envelope theorem)

\[ \nabla f(\theta) = \nabla_2 E(x^*(\theta), \theta) \]

where \( \nabla_2 \) indicates the gradient w.r.t. the second argument.

- Informally, the theorem says that we can treat \( x^*(\theta) \) as if it did not depend on \( \theta \).
Differentiating argmin problems

- Now, consider the function

\[ x^*(\theta) = \arg\min_x E(x, \theta) \]

\[ f(\theta) = L(x^*(\theta), \theta) \]

- By the chain rule, we have

\[ \nabla f(\theta) = (J x^*(\theta))^\top \nabla_1 L(x^*(\theta), \theta) + \nabla_2 L(x^*(\theta), \theta) \]

- How to compute \( J x^*(\theta) = \frac{\partial x^*(\theta)}{\partial \theta} \)?
Fixed points

Consider the following fixed point iteration

\[ x^*(\theta) = g(x^*(\theta), \theta) \iff h(x^*(\theta), \theta) = 0 \]

where \( h(x, \theta) = x - g(x, \theta) \)

By the implicit function theorem

\[ J x^*(\theta) = -(J_1 h(x^*(\theta), \theta))^{-1} J_2 h(x^*(\theta), \theta) \]

where \( J_1 \) and \( J_2 \) are the Jacobians w.r.t. the 1st and 2nd variables
Differentiating argmin problems

- Recall that
  \[ x^*(\theta) = \arg\min_x E(x, \theta) \]

- We have the fixed point iteration (gradient descent)
  \[ x^*(\theta) = x^*(\theta) - \nabla_1 E(x^*(\theta), \theta) \]

- Choosing \( h(x, \theta) = \nabla_1 E(x, \theta) \), we get
  \[
  J x^*(\theta) = -(J_1 \nabla_1 E(x^*(\theta), \theta))^{-1} J_2 \nabla_1 E(x^*(\theta), \theta) \\
  = -(\nabla_1^2 E(x^*(\theta), \theta))^{-1} J_2 \nabla_1 E(x^*(\theta), \theta)
  \]

- In practice, we need to replace \( x^*(\theta) \) by an approximate solution.
Differentiating argmin problems

- Example: hyper-parameter optimization for ridge regression

\[ E(x, \theta) = \frac{1}{2} \| Ax - b \|^2 + \frac{\theta}{2} \| x \|^2 \in \mathbb{R} \]

\[ \nabla_1 E(x, \theta) = A^\top (Ax - b) + \theta x \in \mathbb{R}^d \]

\[ \nabla_1^2 E(x, \theta) = A^\top A + \theta I \in \mathbb{R}^{d \times d} \]

\[ J_2 \nabla_1 E(x, \theta) = x \in \mathbb{R}^{d \times 1} \]

\[ x^*(\theta) = (A^\top A + \theta I)^{-1} A^\top b \]

- \( J \, x^*(\theta) \) is therefore obtained by solving the following linear system

\[ (A^\top A + \theta I)[J \, x^*(\theta)] = -x^*(\theta) \]
Differentiating argmin problems

- An alternative idea to obtain $J \mathbf{x}^*(\theta)$ is to backpropagate through gradient descent:

$$\mathbf{x}^{t+1}(\theta) = \mathbf{x}^t(\theta) - \eta_t \nabla_1 E(\mathbf{x}^t(\theta), \theta)$$

- No longer needs to solve a linear system...
- ...but needs to store intermediate iterates $\mathbf{x}^t(\theta)$ or checkpoints
- Possibility to use truncated backpropagation
- Possibility to use reversible dynamics in some cases
Inference in graphical models

- **Gibbs distribution**

  \[ P(Y = y; \theta) \propto \exp(y \cdot \theta) \]

  where \( y \in \mathcal{Y} \subset \{0, 1\}^n \)

- **Log-partition function**

  \[ f(\theta) = \log \sum_{y \in \mathcal{Y}} \exp(y \cdot \theta) \]

- **Fact.**

  \[ (P(Y_i = 1; \theta))_{i=1}^n = \mathbb{E}[Y] = \nabla f(\theta) \]

- If we know how to compute \( f(\theta) \), we can get expectations / marginal probabilities by autodiff! Recovers forward-backward algorithms as special case. For a proof, see e.g. this paper.
Outline

1. Numerical differentiation
2. Chain compositions
3. Computational graphs
4. Implementation
5. Advanced topics
6. Conclusion
Summary

- Automatic differentiation is one of the keys that enabled the deep learning “revolution”.

- Backward / reverse differentiation is more efficient when the function has more inputs than outputs.

- Which is the de-facto setting in machine learning!

- Even if you use TensorFlow / JAX / PyTorch, implementing a rudimentary autodiff library is a very good exercise.
References

The following tutorials have been a great inspiration:


Two minimalist implementations of autodiff:

- Autodidact, by Matthew Johnson.
- Micrograd, by Andrej Karpathy.