Pitfalls of Automatic Differentiation

UCSD CSE 291 Differentiable Programming
Tzu-Mao Li
Today

Understanding Automatic Differentiation Pitfalls

Jan Hückelheim¹ | Harshitha Menon² | William Moses³
| Bruce Christianson⁴ | Paul Hovland¹ | Laurent Hascoët⁵
Take home message

- Sometimes, you can’t just autodiff your program and walk away

\[ L(T, \omega) = \int_0^T \cos(\omega t) dt \]

\[ \text{AD: } \frac{\partial}{\partial \omega} (L(T, \omega)) \]

\[ \text{FD: } \frac{1}{2h} (L(T, \omega + h) - L(T, \omega - h)), h = 0.1 \]
4 Pitfalls of AD

- unexpected derivatives
- bad approximation
- branching
- numerical accuracy

**Code with fast path**

```python
def g(x):
    if x == 0:
        return 0
    else:
        return x

def g_d(x, xd=1.0):
    if x == 0:
        return 0  # wrong!
    else:
        return xd
```
4 Pitfalls of AD

- unexpected derivatives
- bad approximation
- branching
- numerical accuracy

Code with fast path:

```python
def g(x):
    if x == 0:
        return 0
    else:
        return x

def g_d(x, xd=1.0):
    if x == 0:
        return 0  # wrong!
    else:
        return xd
```
Chaos in physics can lead to ill-behaved derivatives

Lorenz attractor

A slight change of \( r \) can completely change the physical trajectory

\[
\begin{align*}
\dot{x} &= \sigma(y - x) \\
\dot{y} &= x(r - z) - y \\
\dot{z} &= xy - bz
\end{align*}
\]
Chaos in physics can lead to ill-behaved derivatives

Lorenz attractor

the average of X position over time T is differentiable with r

Average X: $L_x(r, T) = \int_0^T l_x(r, t) dt$
Chaos in physics can lead to ill-behaved derivatives

the average of X position over time T is differentiable with r

\[
L_x(r, T) = \int_0^T l_x(r, t) dt
\]

but the derivative w.r.t. r grows exponentially with T!!!
Chaos in physics can lead to ill-behaved derivatives

Lorenz attractor

the average of X position over time T is differentiable with r

Average X: $L_x(r, T) = \int_0^T l_x(r, t) dt$

finite differences, on the other hand, is a lot more well behaved
Non-chaotic functions can also have similar behaviors

\[ L(T, \omega) = \int_0^T \cos(\omega t) dt \]
Non-chaotic functions can also have similar behaviors.

\[ L(T, \omega) = \int_0^T \cos(\omega t) dt \]

AD: \[ \frac{\partial}{\partial \omega} (L(T, \omega)) \]

FD: \[ \frac{1}{2h} (L(T, \omega + h) - L(T, \omega - h)), h = 0.1 \]
A possible remedy: tame the integrand using a smooth falloff

\[ \int_0^T \cos(\omega t) \, dt \quad \longrightarrow \quad \int_0^T e^{-\frac{t^2}{T^2\sigma^2}} \cos(\omega t) \, dt \]

often called “windowing” or “apodization” in signal processing
4 Pitfalls of AD

- unexpected derivatives
- bad approximation
- branching
- numerical accuracy

Code with fast path:

```python
def g(x):
    if x == 0:
        return 0
    else:
        return x

def g_d(x, xd=1.0):
    if x == 0:
        return 0 # wrong!
    else:
        return xd
```
Differentiating an approximation of a function can lead to arbitrarily bad derivatives.

- **A smooth primal function**: A smooth, continuous function.
- **A piecewise approximation**: A function that is defined piece by piece with different equations over different intervals.
- **Derivative of a piecewise approximation (bad!)**: The derivative of a piecewise function can exhibit poor behavior, such as discontinuities or spurious spikes.
Differentiating an approximation of a function can lead to arbitrarily bad derivatives

- similar problems happen with integrals & root solving, we have talked about them in previous lectures

```python
def g(t):
    sum = 0
    for i in range(N):
        if rand() < t:
            sum += 1
    return sum / N

def dg(t, dt):
    sum = 0
dsum = 0
    for i in range(N):
        if rand() < t:
            sum += 1
dsum += 0
    return dsum / N
```

Figure 3: Jacobian estimate errors. Empirical error of implicit differentiation follows closely the theoretical upper bound. Unrolling achieves a much worse error for comparable iterate error.
A potential remedy: differentiate the primal function, then make the approximation

\[ \int_{0}^{1} [x < t] \, dx \]

\[ \int_{0}^{1} \delta(t - x) \, dx \]

\([0 < t < 1]\)
4 Pitfalls of AD

- unexpected derivatives
- bad approximation
- branching
- numerical accuracy

**Code with fast path**

```python
def g(x):
    if x == 0:
        return 0
    else:
        return x

def g_d(x, xd=1.0):
    if x == 0:
        return 0  # wrong!
    else:
        return xd
```
Branching can lead to undefined behavior

def sillyID_0(x):
    return max(0, x) - max(-x, 0)

def sillyID_1(x):
    return max(x, 0) - max(-x, 0)

def sillyID_2(x):
    return max(0, x) - max(0, -x)

d/dx for x=0 in PyTorch

example from

Distribution Theoretic Semantics for Non-Smooth Differentiable Programming

PEDRO H. AZEVEDO DE AMORIM, Cornell University, United States
CHRISTOPHER LAM, University of Illinois at Urbana-Champaign, United States
Branching can lead to undefined behavior

def sillyID_0(x):
    return max(0, x) - max(-x, 0)

def sillyID_1(x):
    return max(x, 0) - max(-x, 0)

def sillyID_2(x):
    return max(0, x) - max(0, -x)

d/dx for x=0 in PyTorch

1
0
2

example from

Distribution Theoretic Semantics for Non-Smooth Differentiable Programming

PEDRO H. AZEVEDO DE AMORIM, Cornell University, United States
CHRISTOPHER LAM, University of Illinois at Urbana-Champaign, United States
The undefined behavior only occurs in a measure zero set of the input space hence, a practical hack is to simply perturb the input slightly only behave weirdly at $x=0$

def sillyID_0(x):
    return max(0, x) - max(-x, 0)
def sillyID_1(x):
    return max(x, 0) - max(-x, 0)
def sillyID_2(x):
    return max(0, x) - max(0, -x)

see proof here
However, non-differentiability can lead to numerical issues in general

\[ f(x, y) = \sqrt{x^2 + y^2} \]

\[ Df(0,0)(dx, dy) = ? \]

most autodiff systems will throw NaN at you
However, non-differentiability can lead to numerical issues in general

\[ f(x, y) = \sqrt{x^2 + y^2} \]

\[ Df(0,0)(dx, dy) = ? \]

most autodiff systems will throw NaN at you

\[ Df(10^{-10}, 10^{-10})(dx, dy) = ? \]
Potential remedy: no definite answer yet!

- one random idea from me: maybe AD systems should issue warnings when an input is close to a non-differentiable point?

- any other ideas?

```
def sillyID_0(x):
    return max(0, x) - max(-x, 0)

def sillyID_1(x):
    return max(x, 0) - max(-x, 0)

def sillyID_2(x):
    return max(0, x) - max(0, -x)
```

detecting whether different programs are semantically equivalent is undecidable!
4 Pitfalls of AD

- unexpected derivatives
- bad approximation
- branching
- numerical accuracy

Code with fast path

```python
def g(x):
    if x == 0:
        return 0
    else:
        return x

def g_d(x, xd=1.0):
    if x == 0:
        return 0 # wrong!
    else:
        return xd
```
The derivative operation can be less numerically accurate than primal ones.

**Single precision**

**Double precision**

\[
\begin{align*}
\text{Relative error} & \quad 10^{-2} \quad 10^{-5} \quad 10^{-8} \quad 10^{-11} \quad 10^{-14} \\
\text{x} & \quad 10^{-5} \quad 10^{-3} \quad 10^{-1} \quad 10^{-3} \quad 10^{-1} \\
\end{align*}
\]
A potential remedy: Herbie

doesn’t work for large scale programs but is excellent for small operations

<table>
<thead>
<tr>
<th>Initial Program: 5.9% accurate, 1.0× speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{e^x}{e^{x^2}} = 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative 1: 99.1% accurate, 41.0× speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{1}{x} + 0.5 )</td>
</tr>
<tr>
<td>▶ Derivation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative 2: 98.0% accurate, 58.3× speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\frac{1}{x}}{x} )</td>
</tr>
<tr>
<td>▶ Derivation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative 3: 3.2% accurate, 205.0× speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
</tr>
<tr>
<td>▶ Derivation</td>
</tr>
</tbody>
</table>

[https://herbie.uwplse.org/](https://herbie.uwplse.org/)
Another potential remedy: error bounds tracking through uncertainty propagation

\[ \begin{align*}
    a \oplus b &= \text{round}(a + b) \in (a + b)(1 \pm \epsilon_m) \\
    &= [(a + b)(1 - \epsilon_m), (a + b)(1 + \epsilon_m)].
\end{align*} \]

Keep track of the lower bound and upper bound for each operation to track the floating point cancellation error.

\[ \begin{align*}
    (((a \oplus b) \oplus c) \oplus d) &\in (((a + b)(1 \pm \epsilon_m)) + c)(1 \pm \epsilon_m) + d)(1 \pm \epsilon_m) \\
    &= (a + b)(1 \pm \epsilon_m) + c(1 \pm \epsilon_m)^2 + d(1 \pm \epsilon_m).
\end{align*} \]
4 Pitfalls of AD

- unexpected derivatives
- bad approximation
- branching
- numerical accuracy
Debugging strategies

- finite differences

- though sometimes, finite differences are not to be trusted since they are also susceptible to numerical errors

- compare between forward and reverse modes

- in general, having language support for debugging AD issues is an open research problem
Take home message

• Sometimes, you can’t just autodiff your program and walk away

\[ L(T, \omega) = \int_0^T \cos(\omega t) dt \]

AD: \[ \frac{\partial}{\partial \omega} (L(T, \omega)) \]

FD: \[ \frac{1}{2h} (L(T, \omega + h) - L(T, \omega - h)) \]

\( h = 0.1 \)