Better Living with Automatic Differentiation

clab lecture: September 2, 2014

Chris Dyer
Empirical Risk Minimization

• Define a model that predicts something given some inputs and modulated by parameters

• Define a loss function that compares the model prediction to the “truth” (e.g., squared error, log loss, …)

• Get some training data

• Fit the parameters to minimize the empirical loss on the training data, probably with regularization
Learning: Minimization

- Write down the the **loss** (fixed) on **your data** (fixed) as a function of your parameters (variable)
- Differentiate this function with respect to your parameters
- Write code to compute the derivatives
- Chose one: LBFGS, AdaGrad, CG, ...
The Challenge

• We spend a lot of our lives differentiating and writing code to compute derivatives.

• It is very easy to get this wrong.

• (Yes, they look sort of fancy in papers; yes, they can provide intuitions you might miss)

• **We should stop doing this.** We have better things to do: experiments and new models.
Alternatives

- **Numerical Differentiation** (ND)
  - Implement the loss function, for every parameter, change it a bit and recompute. Problems: scales $O(n)$ ($n=$ # parameters), rounding errors

- **Symbolic Differentiation** (SD)

- **Automatic Differentiation** (AD)
Automatic Differentiation

• Write code to compute the loss function as you normally would

  • For structured models, this will mostly just be “the forward/inside algorithm”

  • For neural networks, this will mostly just be “forward propagation”

• “Augment” your code with AD data types or classes (hint: use a good language like Julia, C++, Python, Haskell, … that will do it with operator overloading)

• You will never have to write the backward algorithm / outside algorithm / back propagation again!
Any function?

• Any complex function is the composition of simpler functions

• Compilers/interpreters turn complex functions into simple functions (CPU operations)

• The chain rule for derivatives says how you differentiate compositions of simple functions

• **AD automatically tracks the extra information you need to compute derivatives**
Any Function?

- Most libraries handle a really broad range of things
  - max, min, fabs compute sub derivatives
  - conditionals work
  - Linear algebra routines (SVD, matrix inverse) work
- But if your function isn’t subdifferentiable, you can’t make it so with AD. :p
Let’s start with a really simple example.

\[ y = \log \sin^2 x \]

What is the derivative at \( x_0 \)?

<table>
<thead>
<tr>
<th>components</th>
<th>range</th>
<th>differential</th>
<th>d-range</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = f(u) = \log u )</td>
<td>( \mathbb{R} )</td>
<td>( \frac{dy}{du} = \frac{1}{u} )</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>( u = g(v) = v^2 )</td>
<td>( \mathbb{R} )</td>
<td>( \frac{du}{dv} = 2v )</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>( v = h(x) = \sin x )</td>
<td>( \mathbb{R} )</td>
<td>( \frac{dv}{dx} = \cos x )</td>
<td>( \mathbb{R} )</td>
</tr>
</tbody>
</table>

\[
\frac{dy}{dx} \bigg|_{x=x_0} = \frac{dy}{du} \bigg|_{u=g(h(x_0))} \cdot \frac{du}{dv} \bigg|_{v=h(x_0)} \cdot \frac{dv}{dx} \bigg|_{x=x_0}
\]
In general, for our applications $\mathbf{x}$ in $f(\mathbf{x})$ will be a vector.

$$y = \sum_{i=1}^{n} (W \exp \mathbf{x})_i$$

where $\mathbf{x} \in \mathbb{R}^d$ and $W \in \mathbb{R}^{n \times d}$

<table>
<thead>
<tr>
<th>components</th>
<th>range</th>
<th>differential</th>
<th>d-range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = f(\mathbf{u}) = \sum_{i=1}^{n} u_i$</td>
<td>$\mathbb{R}$</td>
<td>$\frac{\partial y}{\partial \mathbf{u}} = 1$</td>
<td>$\mathbb{R}^{1 \times n}$</td>
</tr>
<tr>
<td>$\mathbf{u} = g(\mathbf{v}) = W \mathbf{v}$</td>
<td>$\mathbb{R}^n$</td>
<td>$\frac{\partial \mathbf{u}}{\partial \mathbf{v}} = \mathbf{W}$</td>
<td>$\mathbb{R}^{n \times d}$</td>
</tr>
<tr>
<td>$\mathbf{v} = h(\mathbf{x}) = \exp \mathbf{x}$</td>
<td>$\mathbb{R}^d$</td>
<td>$\frac{\partial \mathbf{v}}{\partial \mathbf{x}} = \text{diag}(\exp \mathbf{x})$</td>
<td>$\mathbb{R}^{d \times d}$</td>
</tr>
</tbody>
</table>

$$\left. \frac{dy}{dx} \right|_{x=x_0} = \frac{dy}{du} \bigg|_{u=g(h(x_0))} \cdot \frac{du}{dv} \bigg|_{v=h(x_0)} \cdot \frac{dv}{dx} \bigg|_{x=x_0}$$
Two Evaluation Strategies

\[ \frac{dy}{dx} \bigg|_{x=x_0} = \frac{dy}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx} \bigg|_{x=x_0} \]

“Forward”

“Backward” or “Adjoint”
Two Evaluation Strategies

Which is better?
Which is better?

- When you have vector inputs, **backward (adjoint) mode is usually more efficient** (often, by a lot).
  - Rather than intermediate **matrices**, you just have **vectors**
  - Depending on the problem (number of inputs, outputs, layers), one or the other may be better
  - Forward mode is easier to implement
  - Fortunately, many tools exist…
Implementation

- Operator/function overloading
  - Rather than using `double` use `adouble` (or whatever your AD library calls it)
  - It secretly keeps track of the operation stack (for backward mode) or partial results for forward mode

- Source code transformation
  - For languages that don’t support OO
Example: adept

```cpp
double loss(const double x[2]) {
  double y = 4;
  double s = 2.0 * x[0] + 3.0 * x[1] * x[1];
  y *= sin(s);
  return y / exp(x[0] + x[1]);
}
```

```cpp
#include “adept.h”
using namespace adept;

adouble loss(const adouble x[2]) {
  adouble y = 4;
  adouble s = 2.0 * x[0] + 3.0 * x[1] * x[1];
  y *= sin(s);
  return y / exp(x[0] + x[1]);
}
```
```cpp
#include <iostream>
#include "adept.h"

using namespace adept;
using namespace std;

adouble loss(const adouble x[2]) {
  ...
}

int main() {
  Stack s;
  adouble x[2] = { 1.1, -0.2 };
  s.new_recording();
  adouble y = loss(x);
  y.set_gradient(1.0);
  s.compute_adjoint();
  cout << "loss(x) = " << y.value() << endl;
  cout << "d loss / dx0 = " << x[0].get_gradient() << endl;
  cout << "d loss / dx1 = " << x[1].get_gradient() << endl;
}
```

```bash
allegro$ ./ex
loss(x) = 1.19081
d loss / dx0 = -3.40599
d loss / dx1 = 0.138292
```
CRF & LBL examples

allegro$ module load adept-1.0
allegro$ module load eigen-3.2.2
allegro$ git clone git@github.com:clab/clab-autodiff-examples.git