Lecture 3: Automatic Differentiation

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

Still looking at our first principle from the intro lecture...

Principle #1: Write your learning task as an optimization problem, and solve it via fast algorithms that update the model iteratively with easy-to-compute steps using numerical linear algebra.

Gradients

Many, if not most, machine learning training algorithms use gradients to optimize a function.

What is a gradient?

Suppose I have a function f from $\mathbb{R}^d$ to $\mathbb{R}$. The gradient, ∇f, is a function from $\mathbb{R}^d$ to $\mathbb{R}^d$ such that

$$(\nabla f(w))_i = \frac{\partial}{\partial w_i} f(w) = \lim_{\delta \rightarrow 0} \frac{f(w + \delta e_i) - f(w)}{\delta},$$

that is, it is the **vector of partial derivatives of the function**.

Another, perhaps cleaner (and basis-independent), definition is that $\nabla f(w)^T$ is the linear map such that for any $\Delta \in \mathbb{R}^d$

$$\nabla f(w)^T \Delta = \lim_{a \rightarrow 0} \frac{f(w + a\Delta) - f(w)}{a}.$$

More informally, it is the unique vector $\nabla f(w_0)$ such that $f(w) \approx f(w_0) + (w - w_0)^T \nabla f(w_0)$ for w nearby w_0.

Let's derive some gradients!

$f(x) = x^T A x$

$$\lim_{a \rightarrow 0} \frac{(x + a\Delta)^T A (x + a\Delta) - x^T A x}{a}$$

$$\lim_{a \to 0} \frac{x^T A x + a \Delta^T A x + a x^T A \Delta + a^2 \Delta^T A \Delta - x^T A x}{a}$$

$$\lim_{a \to 0} \Delta^T A x + x^T A \Delta + a \Delta^T A \Delta$$

$$\Delta^T A x + x^T A \Delta$$

$$\Delta^T A x + \Delta^T A^T x$$

$$\Delta^T (A x + A^T x)$$

$$\nabla f(x) = A x + A^T x$$

$f(x) = \|x\|_2^2 = \sum_{i=1}^d x_i^2 = x^T x = x^T I x$

$$\frac{\partial f}{\partial x_i} = \frac{\partial}{\partial x_i} x_i^2 = 2 x_i$$

$$\nabla f(x) = 2x$$

$$f(x + a\Delta) = (x + a\Delta)^T (x + a\Delta)$$

$f(x) = \|x\|_1 = \sum_{i=1}^d |x_i|$

$$\frac{\partial f}{\partial x_i} = \frac{\partial}{\partial x_i} |x_i| = \operatorname{sign}(x_i) = \frac{x_i}{|x_i|}$$

$$\nabla f(x) = \operatorname{sign}(x)$$

Symbolic differentiation

What we just did is called *symbolic differentiation*.

- Write your function as a single mathematical expression.
- Apply the chain rule, product rule, et cetera to differentiate that expression.
- Execute the expression as code.

Symbolic differentiation: Problems

The naive symbolic differentiation approach has a couple of problems.

- Converting code into a mathematical expression is not trivial! You'd need some sort of reflection capability.
- There's no guarantee that the computational structure of our original code will be preserved.

- Expressions you get from naive symbolic differentiation can be larger and more complicated than the thing we're differentiating.

An Illustrative Example

Imagine that we have some functions f, g, and h, and we are interested in computing the derivative of the function $f(g(h(x)))$ with respect to x. By the chain rule, $$

```
\frac{d}{dx} f(g(h(x)))
=
f'(g(h(x))) \cdot g'(h(x)) \cdot h'(x).
```

$$ Imagine that we just copy-pasted this math into python. We'd get

In [1]:
```
def grad_fgh(x):
    return grad_f(g(h(x))) * grad_g(h(x)) * grad_h(x)
```

This code recomputes $h(x)$ twice!

To avoid this, we could imagine writing something like this instead:

In [2]:
```
def better_grad_fgh(x):
    hx = h(x)
    return grad_f(g(hx)) * grad_g(hx) * grad_h(x)
```

This code only computes $h(x)$ once. But we had to do some manual work to pull out common subexpressions, work that goes beyond what the plain chain rule gives us.

Now you might ask the very reasonable question: **So what?** We just recomputed one function. **Could this really have a significant impact on performance?**

To answer this question, suppose that we now have functions $f_1, f_2, \ldots, f_k$, and consider differentiating the more complicated function $f_k(f_{k-1}(\cdots f_2(f_1(x))\cdots))$. The chain rule will give us

$$\begin{aligned}\frac{d}{dx} f_k(f_{k-1}(\cdots f_2(f_1(x))\cdots)) \\ &= f_k'(f_{k-1}(\cdots f_2(f_1(x))\cdots)) \cdot f_{k-1}'(f_{k-2}(\cdots f_2(f_1(x))\cdots)) \\ &\quad \cdots f_2'(f_1(x)) \cdot f_1'(x).\end{aligned}$$

If each function f_i or f_i' takes $O(1)$ time to compute, how long can this derivative take to compute if

- ...I just copy-paste naively into python?
- ...I do something to save redundant computation?

Symbolic Differentiation: Take-away point

Computing a derivative by just applying the chain rule and then naively copying the expression into code can be **asymptotically slower** than other methods.

Numerical Differentiation

$$\nabla f(w)^T \Delta = \lim_{a \to 0} \frac{f(w + a\Delta) - f(w)}{a}.$$

Therefore, for small $\epsilon > 0$,

$$\nabla f(w)^T \Delta \approx \frac{f(w + \epsilon\Delta) - f(w)}{\epsilon}.$$

It's often better to use the symmetric difference quotient instead:

$$\nabla f(w)^T \Delta \approx \frac{f(w + \epsilon\Delta) - f(w - \epsilon\Delta)}{2\epsilon}.$$

Let's Test Numerical Differentiation

In [5]:
```
import math

def f(x):
    return 2*x*x*x-1

def dfdx(x):
    return 6*x*x

def numerical_derivative(f, x, eps = 1e-4):
    return (f(x+eps) - f(x-eps))/(2*eps)

print(dfdx(2.0))
print(numerical_derivative(f, 2.0))
```

```
24.0
24.00000002002578
```

Problems with Numerical Differentiation

- numerical imprecision (bigger problem as number of operations increases)
- if f is not smooth, then wrong results
- unclear how to set ϵ
- for a vector -> scalar function, you have to compute each partial individually, meaning $O(d)$ blowup in cost
- if you set ϵ too small, you can get zeros

Automatic Differentiation

Compute derivatives automatically without any overheads or loss of precision.

Two rough classes of methods:

- Forward mode
- Reverse mode

Forward Mode AD

- Fix one input variable x over $\mathbb{R}$.
- At each step of the computation, as we're computing some value y, also compute $\frac{\partial y}{\partial x}$.
- We can do this with a *dual numbers* approach: each number y is replaced with a pair $(y, \frac{\partial y}{\partial x})$.

Forward Mode AD Demo

In [6]:

```
import numpy as np

def to_dualnumber(x):
    if isinstance(x, DualNumber):
        return x
    elif isinstance(x, float):
        return DualNumber(x)
    elif isinstance(x, int):
        return DualNumber(float(x))
    else:
        raise Exception("couldn't convert {} to a dual number".format(x))

class DualNumber(object):
    def __init__(self, y, dydx=0.0):
        super().__init__()
        self.y = y
        self.dydx = dydx

    def __repr__(self):
        return "(y = {}, dydx = {})".format(self.y, self.dydx)

    # operator overloading
    def __add__(self, other):
        other = to_dualnumber(other)
        return DualNumber(self.y + other.y, self.dydx + other.dydx)
    def __sub__(self, other):
        other = to_dualnumber(other)
        return DualNumber(self.y - other.y, self.dydx - other.dydx)
    def __mul__(self, other):
        other = to_dualnumber(other)
        return DualNumber(self.y * other.y, self.dydx * other.y + self.y * oth
    def __pow__(self, other):
        other = to_dualnumber(other)
        return DualNumber(self.y ** other.y,
            self.dydx * other.y * self.y ** (other.y - 1) + other.dydx * (self
    def __radd__(self, other):
        return to_dualnumber(other).__add__(self)
    def __rsub__(self, other):
        return to_dualnumber(other).__sub__(self)
    def __rmul__(self, other):
        return to_dualnumber(other).__mul__(self)

def exp(x):
    if isinstance(x, DualNumber):
        return DualNumber(np.exp(x.y), np.exp(x.y) * x.dydx)
    else:
        return np.exp(x)

def sin(x):
    if isinstance(x, DualNumber):
        return DualNumber(np.sin(x.y), -np.cos(x.y) * x.dydx)
    else:
        return np.sin(x)

def forward_mode_diff(f, x):
    x_dual = DualNumber(x, 1.0)
    return f(x_dual).dydx
```

In [8]:
```
print(dfdx(3.0))
print(numerical_derivative(f, 3.0))
print(forward_mode_diff(f, 3.0))
```

```
54.0
54.00000002012462
54.0
```

In [14]:
```
# def g(x):
#     return 12 + 0*x

# print(numerical_derivative(g, 2.0))
# print(forward_mode_diff(g, 2.0))

def h(x):
    return x**5.0

def dhdx(x):
    return 5*x**4

print(dhdx(2.0))
print(forward_mode_diff(h, 2.0))
```

```
80.0
80.0
```

Problems with Forward Mode AD

- Differentiate with respect to **one** input
- If we are computing a gradient of $f : \mathbb{R}^d \to \mathbb{R}$, blow-up proportional to the dimension d

Reverse-Mode AD

- Fix one output ℓ over $\mathbb{R}$
- Compute partial derivatives $\frac{\partial \ell}{\partial y}$ for each value y

- Need to do this by going *backward* through the computation

Problem: can't just compute derivatives as-we-go like with forward mode AD.

```
In [ ]:
```