Automatic Differentiation

Jonathon Hare

Vision, Learning and Control
University of Southampton

Much of this material is based on this blog post:
`https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation`

What is Automatic Differentiation (AD)?

To solve optimisation problems using gradient methods we need to compute the gradients (derivatives) of the objective with respect to the parameters.

- In neural nets we're talking about the gradients of the loss function, $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$: $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$

What is Automatic Differentiation (AD)?

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

What is Automatic Differentiation (AD)?

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

Problems

Static - can't "differentiate algorithms"

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

Problems

Numerical errors - will compound in deep nets

What is Automatic Differentiation (AD)?

Computing Derivatives

There are three ways to compute derivatives:

- Symbolically differentiate the function with respect to its parameters
  - by hand
  - using a CAS
- Make estimates using finite differences
- Use Automatic Differentiation

What is Automatic Differentiation (AD)?

Automatic Differentiation is:

- a method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards.
  - Takes code that computes a function and uses that to compute the derivative of that function.
  - The goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.

Lets think about differentiation and programming

Example (Math)

$$
\begin{aligned}
x &= ? \\
y &= ? \\
a &= x\,y \\
b &= \sin(x) \\
z &= a + b
\end{aligned}
$$

Lets think about differentiation and programming

Example (Math)

$$x = ?$$
$$y = ?$$
$$a = x\,y$$
$$b = \sin(x)$$
$$z = a + b$$

Example (Code)

```
x = ?
y = ?
a = x * y
b = sin(x)
z = a + b
```

The Chain Rule of Differentiation

Recall the chain rule for a variable/function z that depends on y which depends on x:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

The Chain Rule of Differentiation

Recall the chain rule for a variable/function z that depends on y which depends on x:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

In general, the chain rule can be expressed as:

$$\frac{\partial w}{\partial t} = \sum_i^N \frac{\partial w}{\partial u_i}\frac{\partial u_i}{\partial t} = \frac{\partial w}{\partial u_1}\frac{\partial u_1}{\partial t} + \frac{\partial w}{\partial u_2}\frac{\partial u_2}{\partial t} + \cdots + \frac{\partial w}{\partial u_N}\frac{\partial u_N}{\partial t}$$

where w is some output variable, and u_i denotes each input variable w depends on.

Applying the Chain Rule

Let's differentiate our previous expression with respect to some yet to be given variable t:

$$\begin{aligned}
\frac{\partial x}{\partial t} &= ? \\
\frac{\partial y}{\partial t} &= ? \\
\frac{\partial a}{\partial t} &= x\frac{\partial y}{\partial t} + y\frac{\partial x}{\partial t} \\
\frac{\partial b}{\partial t} &= \cos(x)\frac{\partial x}{\partial t} \\
\frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}
\end{aligned}$$

Applying the Chain Rule

Let's differentiate our previous expression with respect to some yet to be given variable t:

$$\begin{aligned}\frac{\partial x}{\partial t} &= ?\\ \frac{\partial y}{\partial t} &= ?\\ \frac{\partial a}{\partial t} &= x\frac{\partial y}{\partial t} + y\frac{\partial x}{\partial t}\\ \frac{\partial b}{\partial t} &= \cos(x)\frac{\partial x}{\partial t}\\ \frac{\partial z}{\partial t} &= \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}\end{aligned}$$

If we substitute $t = x$ in the above we'll have an algorithm for computing $\partial x/\partial x$. To get $\partial z/\partial y$ we'd just substitute $t = y$.

Translating to code I

We could translate the previous expressions back into a program involving *differential variables* {dx, dy, ...} which represent $\partial x/\partial t, \partial y/\partial t, \ldots$ respectively:

```
dx = ?
dy = ?
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

Translating to code I

We could translate the previous expressions back into a program involving *differential variables* {dx, dy, ...} which represent $\partial x/\partial t, \partial y/\partial t, \ldots$ respectively:

```
dx = ?
dy = ?
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

What happens to this program if we substitute $t = x$ into the math expression?

Translating to code II

```
dx = 1
dy = 0
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

Translating to code II

```
dx = 1
dy = 0
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

The effect is remarkably simple: to compute $\partial z/\partial x$ we just seed the algorithm with dx=1 and dy=0.

Translating to code III

```
dx = 0
dy = 1
da = y * dx + x * dy
db = cos(x) * dx
dz = da + db
```

To compute $\partial z / \partial y$ we just seed the algorithm with dx=0 and dy=1.

Making Rules

- We've successfully computed the gradients for a specific function, but the process was far from automatic.

Making Rules

- We've successfully computed the gradients for a specific function, but the process was far from automatic.
- We need to formalise a set of rules for translating a program that evaluates an expression into a program that evaluates its derivatives.

Making Rules

- We've successfully computed the gradients for a specific function, but the process was far from automatic.
- We need to formalise a set of rules for translating a program that evaluates an expression into a program that evaluates its derivatives.
- We have actually already discovered 3 of these rules:

```
c = a + b    =>  dc = da + db
c = a * b    =>  dc = b * da + a * db
c = sin(a)   =>  dc = cos(a) * da
```

More rules

These initial rules:

```
c=a+b      =>  dc=da+db
c=a*b      =>  dc=b*da+a*db
c=sin(a)   =>  dc=cos(a)*da
```

can easily be extended further using multivariable calculus:

```
c=a-b      =>  dc=da-db
c=a/b      =>  dc=da/b-a*db/b**2
c=a**b     =>  dc=b*a**(b-1)*da+log(a)*a**b*db
c=cos(a)   =>  dc=-sin(a)*da
c=tan(a)   =>  dc=da/cos(a)**2
```

Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.

Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.
- The order of computation remains unchanged: if a statement K is evaluated before another statement L, then the differential analogue of K is evaluated before the analogue statement of L.

Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.
- The order of computation remains unchanged: if a statement K is evaluated before another statement L, then the differential analogue of K is evaluated before the analogue statement of L.
- This is **Forward-mode Automatic Differentiation**.

Interleaving differential computation

A careful analysis of our original program and its differential analogue shows that its possible to interleave the differential calculations with the original ones:

```
x  = ?
dx = ?

y  = ?
dy = ?

a  = x * y
da = y * dx + x * dy

b  = sin(x)
db = cos(x) * dx

z  = a + b
dz = da + db
```

Interleaving differential computation

A careful analysis of our original program and its differential analogue shows that its possible to interleave the differential calculations with the original ones:

```
x  = ?
dx = ?

y  = ?
dy = ?

a  = x * y
da = y * dx + x * dy

b  = sin(x)
db = cos(x) * dx

z  = a + b
dz = da + db
```

Dual Numbers

- This implies that we can keep track of the value and gradient at the same time.

Interleaving differential computation

A careful analysis of our original program and its differential analogue shows that its possible to interleave the differential calculations with the original ones:

```
x  = ?
dx = ?

y  = ?
dy = ?

a  = x * y
da = y * dx + x * dy

b  = sin(x)
db = cos(x) * dx

z  = a + b
dz = da + db
```

Dual Numbers

- This implies that we can keep track of the value and gradient at the same time.
- We can use a mathematical concept called a "Dual Number" to create a very simple direct implementation of AD.

Backward Mode AD

A bit more information about this