CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 11

Tuesday, October 12th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

Automatic Differentiation: ***Forward Mode*** (basics)

- Evaluation trace
- The computational graph
- Computing derivatives of one variable using the forward mode
- Computing derivatives in higher dimensions using the forward mode

Beyond the basics:

- The Jacobian in forward mode
- What the forward mode actually computes
- Implementation approaches

OUTLINE

- Review of complex numbers and introduction of dual numbers
- Implementation of forward mode AD: operator overloading
- Reverse mode of AD
- Examples for application

REVIEW OF COMPLEX NUMBERS

REVIEW OF COMPLEX NUMBERS

A complex number has the form:

$$z = x + iy$$

- x: is the *real* part,
- y: is the *imaginary* part.

The *imaginary unit* i gives the complex number $z \in \mathbb{C}$ the special property that *defines* the square root of a ***negative*** number

$$i = \sqrt{-1},$$

such that $i^2 = -1$.

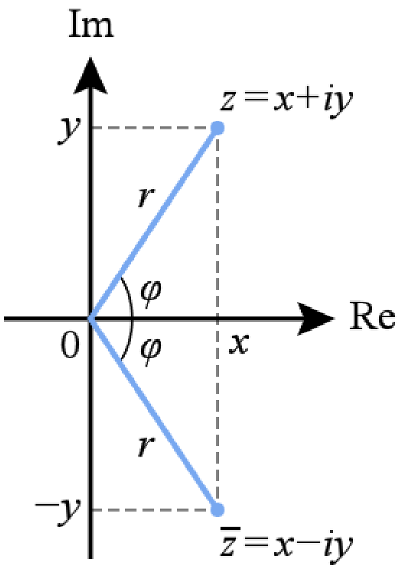
REVIEW OF COMPLEX NUMBERS

A complex number has the form:

$$z = x + iy$$

- x: is the *real* part,
- y: is the *imaginary* part.

- You can think of z as a *two-dimensional* vector.
- The imaginary unit i *extends the real line* with an orthogonal *imaginary* axis.



REVIEW OF COMPLEX NUMBERS

Complex numbers have several properties that we can use:

- Complex conjugate: $z^* = x - iy$
- Magnitude: $|z|^2 = zz^* = (x + iy)(x - iy) = x^2 + y^2$
- Polar form: $z = re^{i\varphi}$
    - r: is called *radius*, $r = |z|$
    - φ: is called *angle*, $\varphi = \arctan(y/x)$

If you compute the product

$$z = z_1 z_2$$

what happens to the radius and angle of z?

Can you see why $zz^* \in \mathbb{R}$ is a real number?

TOWARDS DUAL NUMBERS

A dual number has similarity to a complex number but the unit that gives the number its special property is defined differently.

A dual number consists of a real part and a *dual* part and is written as

$$z = a + b\epsilon,$$

where $a, b \in \mathbb{R}$ and ϵ is a special (nilpotent) number such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. ***Note:*** ϵ *is not* a real number.

Disclaimer: the following provides some ideas on how you can go about implementing an automatic differentiation code. You are free to make other choices for your project as long as you stick with the `python` programming language.

DUAL NUMBERS

Dual numbers have several useful properties:

- Dual conjugate: $z^* = a - b\epsilon$
- Magnitude: $|z|^2 = zz^* = (a + b\epsilon)(a - b\epsilon) = a^2$
- Polar decomposition: $z = a(1 + m\epsilon)$
  - where $m = \frac{b}{a}$ for $a \neq 0$
  - $e^{m\epsilon} = 1 + m\epsilon + \frac{1}{2}(m\epsilon)^2 + \ldots = 1 + m\epsilon$

What is more interesting: dual numbers have the following properties for addition and multiplication:

$$z_1 + z_2 = (a_1 + b_1\epsilon) + (a_2 + b_2\epsilon) \;= (a_1 + a_2) + (b_1 + b_2)\epsilon$$

$$z_1 z_2 = (a_1 + b_1\epsilon)(a_2 + b_2\epsilon) \;= (a_1 a_2) + (a_1 b_2 + a_2 b_1)\epsilon$$

DUAL NUMBERS

$$z_1 + z_2 = (a_1 + b_1\epsilon) + (a_2 + b_2\epsilon) = (a_1 + a_2) + (b_1 + b_2)\epsilon$$

$$z_1 z_2 = (a_1 + b_1\epsilon)(a_2 + b_2\epsilon) = (a_1 a_2) + (a_1 b_2 + a_2 b_1)\epsilon$$

Let us now introduce two functions $u(x)$ and $v(x)$ and let $u'(x)$ and $v'(x)$ be their *derivative* with respect to x.

We substitute $a_1 = u, b_1 = u'$ and $a_2 = v, b_2 = v'$ and find:

$$z_1 + z_2 = (u + u'\epsilon) + (v + v'\epsilon) = (u + v) + (u' + v')\epsilon$$

$$z_1 z_2 = (u + u'\epsilon)(v + v'\epsilon) = (uv) + (uv' + u'v)\epsilon$$

Observe:

1. Adding dual numbers together resembles the *linearity* of addition and results in *adding the functions in the real part* and *adding the derivatives in the dual part.*
2. Multiplication results in multiplication of the functions in the real part and *the correct product rule for the derivatives in the dual part.*

DUAL NUMBERS

If you think of u and v as *intermediate* variables v_i and v_j in the primal trace of forward mode AD, then their derivatives correspond to the tangent trace $D_p v_i$ and $D_p v_j$.

A dual number can therefore be used as a *data structure* in automatic differentiation. In forward mode AD, we always evaluate v_j and $D_p v_j$ *simultaneously*, we carry them forward as a ***pair***, where the *real* part corresponds to the *primal trace* and the *dual* part corresponds to the *tangent trace*:

$$z_j = v_j + D_p v_j \epsilon$$

DUAL NUMBERS

So far, *dual numbers* seem to have all the properties we are looking for in a data structure that is useful for an automatic differentiation algorithm. But they are *useless* if we can not use them with the chain rule.

We can expand any analytic function $f(z_j)$, where z_j is a dual number, using a Taylor series expansion. In the following we expand the series around the point $\zeta_j = v_j + 0\epsilon$ which is the real part of z_j: (***notation:*** the κ-th derivative $f^{(\kappa)}(z_j)$ is with respect to z_j)

$$f(z_j) = f(v_j + D_p v_j \epsilon) = \sum_{\kappa=0}^{\infty} \frac{f^{(\kappa)}(\zeta_j)}{\kappa!}(z_j - \zeta_j)^{\kappa} = \sum_{\kappa=0}^{\infty} \frac{f^{(\kappa)}(v_j)}{\kappa!}(D_p v_j \epsilon)^{\kappa}$$

$$= f(v_j) + f'(v_j) D_p v_j \epsilon$$

All higher order terms vanish because of the definition $\epsilon^2 = 0$.

DUAL NUMBERS

$$f(z_j) = f(v_j + D_p v_j \epsilon) = \sum_{\kappa=0}^{\infty} \frac{f^{(\kappa)}(\zeta_j)}{\kappa!}(z_j - \zeta_j)^\kappa = \sum_{\kappa=0}^{\infty} \frac{f^{(\kappa)}(v_j)}{\kappa!}(D_p v_j \epsilon)^\kappa$$
$$= f(v_j) + f'(v_j) D_p v_j \epsilon$$

Recall: last lecture we were studying the forward primal and tangent traces of $f(x) = x - \exp\left(-2\left(\sin(4x)\right)^2\right)$. The first two *intermediate* variables are shown again below:

Forward primal trace	Forward tangent trace	Numerical value: $v_j; D_p v_j$
$v_0 = x_1 = \frac{\pi}{16}$	$D_p v_0 = 1$	1.963495e-01; 1.000000e+00
$v_1 = 4v_0$	$D_p v_1 = 4 D_p v_0$	7.853982e-01; 4.000000e+00
$v_2 = \sin(v_1)$	$D_p v_2 = \cos(v_1) D_p v_1$	7.071068e-01; 2.828427e+00

Let us now define the *dual number* $z_1 = v_1 + D_p v_1 \epsilon$. To compute $z_2 = f(z_1) = \sin(z_1)$ we apply the result from the Taylor series above (chain rule):

$$z_2 = \sin(z_1) = \underbrace{\sin(v_1)}_{v_2} + \underbrace{\cos(v_1) D_p v_1}_{D_p v_2} \epsilon$$

DUAL NUMBERS: EXERCISE

We are given the following function $f(x) : \mathbb{R} \mapsto \mathbb{R}$:

$$f(x) = \frac{\sin(x)}{\left(\cos(x)\right)^2 + 1}$$

Perform the following tasks (~20 minutes, use the next slide for your solution):

1. Draw the computational graph for $f(x)$. The last intermediate variable is $v_5 = f(x_1)$ (x_1 is the point where we evaluate f).
2. Show that $D_p v_5$ takes the form

$$D_p v_5 = \frac{1}{v_4^2}\left(v_4 D_p v_1 - v_1 D_p v_4\right)$$

   for $v_5 = g(v_1, v_4)$ (***hint:*** chain rule).
3. Compute the last intermediate state with dual numbers $z_5 = g(z_1, z_4)$. Note that the function g is the same as in item 2 above, we just replace the intermediate (primal) variable v_j with dual numbers z_j. Depending on how you draw the graph, the arguments to g may have different subscripts.

DUAL NUMBERS: EXERCISE SOLUTION

IMPLEMENTATION: OPERATOR OVERLOADING

There are different implementation techniques for automatic differentiation. Two techniques often used are

1. *Code translation* on the level of intermediate representation (IR). This happens on the *compiler* level and is therefore very efficient.
2. *Operator-overloading* on the software level. We have already touched this topic when we studied the `python` data model.

The first technique is not in the scope of this class. We will focus on *operator-overloading* which is related to the special (dunder) methods that we already know about.

IMPLEMENTATION: OPERATOR OVERLOADING

What is meant by "operator overloading"?

- *You are already very familiar with it.* Consider for example $\sin(x)$: here the sine is a *mathematical operator* that *acts* on an argument x.
- The operator $\sin$ acts on a real number $x \in \mathbb{R}$, this is your common perception of the operator $\sin$.
- We have just learned about dual numbers. What should be the action of the $\sin$ operator acting on a dual number z. That is, what should be the result of $\sin(z)$? (By now we know the answer to this question.)
- Operator overloading is a *form of polymorphism* where an operator may have different implementations depending on the argument it acts on.

IMPLEMENTATION: OPERATOR OVERLOADING

- *Recall Lecture 7:* we were adding our custom `Thing`'s together.
- Let us revisit what we did there using a `Complex` type instead:

```python
class Complex:
    """Complex number type"""
    def __init__(self, real, imag):
        """Construct a complex number from real and imaginary parts"""
        self.real = real
        self.imag = imag
```

```
>>> z1 = Complex(1, 1)
>>> z2 = Complex(2, 2)
>>> z3 = z1 + z2
Traceback (most recent call last):
  File "/home/fabs/CS107/lecture11/code/complex.py", line 14, in <module>
    z3 = z1 + z2
TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

- You already knew that this does not work!

IMPLEMENTATION: OPERATOR OVERLOADING

- The fix is easy:

```
class Complex:
    """Complex number type"""
    def __init__(self, real, imag):
        """Construct a complex number from real and imaginary parts"""
        self.real = real
        self.imag = imag

    def __add__(self, other):
        """Adding complex numbers together"""
        return Complex(self.real + other.real, self.imag + other.imag)
```

```
>>> z1 = Complex(1, 1)
>>> z2 = Complex(2, 2)
>>> z3 = z1 + z2
>>> vars(z3)
{'real': 3, 'imag': 3}
```

- The interface is of course very incomplete. What about multiplication or division?
- Another operation that you may come across often may be this: `z4 = 1 + z3`.

IMPLEMENTATION: OPERATOR OVERLOADING

- Another operation that you may come across often may be this: `z4 = 1 + z3`.
- You know that `python` resolves the right-hand side to `1.__add__(z3)`.
- It is unlikely that integer objects support your custom `Complex` type. *This operation will fail with a `NotImplementedError`.*
- In that case `python` checks if the other object implements the `__radd__` special method which will then be called instead. (The "r" stands for *reflected* or swapped.)
- You can simply call `__add__` from within `__radd__` ***if the operator is commutative***, but be careful to handle the type of `other` correctly (here `other` is an *integer* and not a `Complex` type).
- You may want to checkout the `isinstance` built-in function.

IMPLEMENTATION: OPERATOR OVERLOADING

- The last example implements multiplication of `Complex` numbers:

```python
class Complex:
    """Complex number type"""
    def __init__(self, real, imag):
        """Construct a complex number from real and imaginary parts"""
        self.real = real
        self.imag = imag

    def __add__(self, other):
        """Adding complex numbers together"""
        return Complex(self.real + other.real, self.imag + other.imag)

    def __mul__(self, other):
        """Multiplying complex numbers together"""
        r1, r2 = self.real, other.real
        i1, i2 = self.imag, other.imag
        return Complex(r1 * r2 - i1 * i2, r1 * i2 + r2 * i1)
```

```python
>>> z1 = Complex(1, 1)
>>> z2 = Complex(1, -1)
>>> z3 = z1 * z2
>>> vars(z3)
{'real': 2, 'imag': 0}
```

- *Don't forget that your AD library must also handle overloaded elementary transcendental functions like* $\sin$, $\cos$, $\exp$ *or* $\ln$ *for example.*

AUTOMATIC DIFFERENTIATION: REVERSE MODE

References for automatic differentiation:

- P. H.W. Hoffmann, *A Hitchhiker's Guide to Automatic Differentiation*, Springer 2015, doi:10.1007/s11075-015-0067-6 (You can access this paper through the Harvard network.)
- Griewank, A. and Walther, A., *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM 2008, Vol. 105
- Nocedal, J. and Wright, S., *Numerical Optimization*, Springer 2006, 2nd Edition
- Baydin, A., Pearlmutter, B., Radul, A. and Siskind, J., *Automatic Differentiation in Machine Learning: A Survey*, Journal of Machine Learning 2017

AUTOMATIC DIFFERENTIATION: REVERSE MODE

- The reverse mode of automatic differentiation is a *two-pass* process as opposed to the m-pass forward mode.
- Reverse mode does not evaluate v_j and $D_p v_j$ simultaneously!
- For this reason the useful properties of dual numbers in forward mode *are not* useful in reverse mode.
- Reverse mode recovers the *partial* derivatives of the i-th *output* f_i with respect to the n variables v_{j-m} with $j = 1, 2, \ldots, n$ by traversing the computational graph *backwards*. The partial derivatives describe the *sensitivity* of the output with respect to the intermediate variable v_{j-m}:

$$\bar{v}_{j-m} = \frac{\partial f_i}{\partial v_{j-m}}.$$

We call $\bar{v}_{j-m}$ the *adjoint* of v_{j-m}.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

- Reverse mode recovers the *partial* derivatives of the i-th *output* f_i with respect to the n variables v_{j-m} with $j = 1, 2, \ldots, n$ by traversing the computational graph *backwards*. The partial derivatives describe the *sensitivity* of the output with respect to the intermediate variable v_{j-m}:

$$\bar{v}_{j-m} = \frac{\partial f_i}{\partial v_{j-m}}.$$

  We call $\bar{v}_{j-m}$ the *adjoint* of v_{j-m}.

- ***Recall:*** we have defined $\boxed{v_{j-m} = x_j \text{ for } j = 1, 2, \ldots, m}$. So the first m adjoints in the reverse mode *are the m components of the gradient* ∇f_i (the same gradient we get from the forward mode).

AUTOMATIC DIFFERENTIATION: REVERSE MODE

What is the difference between forward and reverse mode?

- Forward mode computes the gradient ∇f with respect to the *independent* variable x.
- Reverse mode computes the sensitivity $\bar{v}_{j-m}$ of f with respect to the independent *and* intermediate variables v_{j-m}. We therefore recover the gradient ∇f in reverse mode as well.
- Compared to the forward mode, the reverse mode has a *significantly smaller arithmetic operation count* for mappings of the form $f(x) : \mathbb{R}^m \mapsto \mathbb{R}$ if m is very large. *Artificial neural networks have exactly this property.*
- ***There is no free lunch:*** we have to store the full computational graph in reverse mode.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Forward pass

Computes the primal values v_j *and* the partial derivatives $\frac{\partial v_j}{\partial v_i}$ *with respect to its parent node(s)* v_i. ***Note:*** the partial derivatives here are *the **factors** that show up in the chain rule*, not the chain rule itself. ***We do not need to apply the chain rule explicitly in reverse mode, we will "build it up" in the reverse pass instead!*** That is why we *do not* compute $D_p v_j$ in the forward pass of the reverse mode.

Compare what is being computed:

- ***Forward mode:*** $v_j = \sin(v_i)$ *and* $D_p v_j = \frac{\partial v_j}{\partial v_i} D_p v_i = \cos(v_i) D_p v_i$ (chain rule)
- ***Forward pass in reverse mode:*** $v_j = \sin(v_i)$ *and* $\frac{\partial v_j}{\partial v_i} = \cos(v_i)$ (this ***is not*** the chain rule)

In reverse mode, we must know the relationship between ***parent*** and ***child***:

The partial derivative $\frac{\partial v_j}{\partial v_i}$ describes the change in a *child* node with respect to its *parent* node v_i. This is not the chain rule.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

The two passes in reverse mode: Reverse pass

In the reverse pass we ***reconstruct*** the chain rule that we *ignored* in the forward pass.

The goal is to compute the following quantity for each node v_i:

$$\bar{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j \text{ a child of } i} \frac{\partial f}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \sum_{j \text{ a child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

The partial derivatives $\frac{\partial v_j}{\partial v_i}$ are computed during the forward pass. At the start of the reverse pass, we initialize $\bar{v}_i = 0$ and update the values with

$$\bar{v}_i = \bar{v}_i + \frac{\partial f}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \bar{v}_i + \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

as we iterate over the children j of node i. Once *all* contributions from child nodes are accumulated in node i, we can proceed with updating its parent node(s). If $\bar{v}_j$ for a particular child node ***is not complete*** we can not proceed with $\bar{v}_i$ and must continue with another node instead.

AUTOMATIC DIFFERENTIATION: REVERSE MODE

*The two passes in reverse mode: **Reverse pass***

Recall that for the very last intermediate state we have $v_{n-m} = f(x)$ with $x \in \mathbb{R}^m$ and *this last node obviously has no children* (***recall:*** n is the sum of the *independent* variables (the number m) and *dependent* variables).

We therefore know the initial value of the adjoint $\bar{v}_{n-m}$:

$$\bar{v}_{n-m} = \frac{\partial f}{\partial v_{n-m}} = \frac{\partial v_{n-m}}{\partial v_{n-m}} = 1$$

Which we need to get started as in the reverse pass we traverse the computational graph ***backwards***, from the right (outputs) to the left (inputs).

During the reverse pass, we *only* work with ***numerical values*** not with formulae or overloaded operators.

REVERSE MODE: EXAMPLE

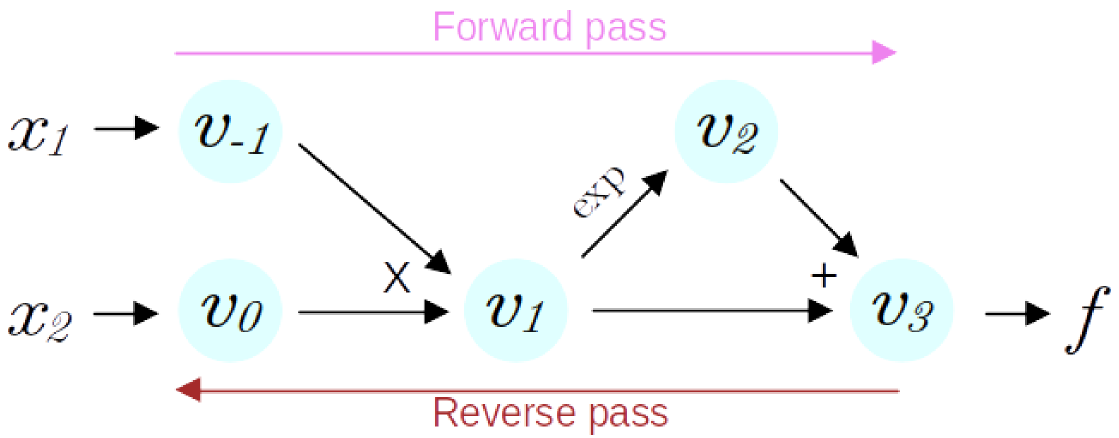
Consider the following function $f(x) : \mathbb{R}^2 \mapsto \mathbb{R}$

$$f(x) = x_1 x_2 + e^{x_1 x_2}$$

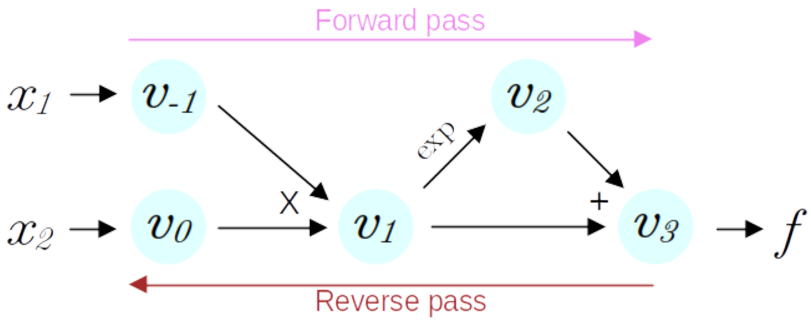
We want to evaluate the gradient ∇f at the point $x = [1, 2]^\intercal$. Computing the gradient by hand is easy:

$$\nabla f = (1 + e^{x_1 x_2}) \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = (1 + e^2) \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Its computational graph is given by:



REVERSE MODE: EXAMPLE



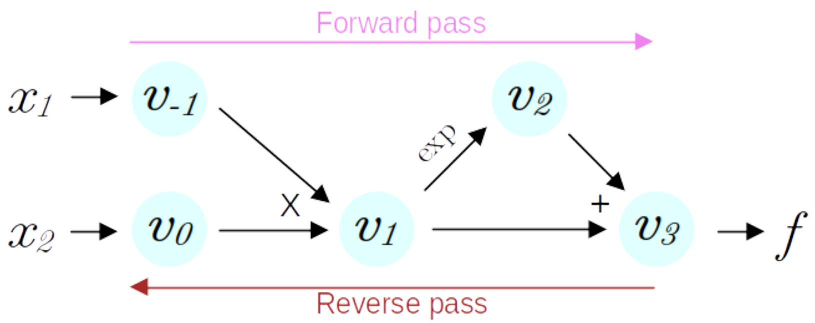
Let's do ***forward mode*** first:

Forward primal trace	Forward tangent trace	Pass with $p = [1,0]^\intercal$	Pass with $p = [0,1]^\intercal$
$v_{-1} = x_1 = 1$	$D_p v_{-1} = p_1$	$D_p v_{-1} = 1$	$D_p v_{-1} = 0$
$v_0 = x_2 = 2$	$D_p v_0 = p_2$	$D_p v_0 = 0$	$D_p v_0 = 1$
$v_1 = v_{-1} v_0 = 2$	$D_p v_1 = v_0 D_p v_{-1} + v_{-1} D_p v_0$	$D_p v_1 = 2$	$D_p v_1 = 1$
$v_2 = e^{v_1} = e^2$	$D_p v_2 = e^{v_1} D_p v_1$	$D_p v_2 = 2e^2$	$D_p v_2 = e^2$
$v_3 = v_1 + v_2$	$D_p v_3 = D_p v_1 + D_p v_2$	$D_p v_3 = 2 + 2e^2$	$D_p v_3 = 1 + e^2$

independent variables — *dependent* variables

Note that we need $m = 2$ passes in ***forward mode*** to compute the gradient ∇f

REVERSE MODE: EXAMPLE



Now ***reverse*** ***mode***:

Forward pass:

Intermediate	Partial Derivatives
$v_{-1} = x_1 = 1$	
$v_0 = x_2 = 2$	
$v_1 = v_{-1} v_0 = 2$	$\frac{\partial v_1}{\partial v_{-1}} = v_0 = 2$ $\frac{\partial v_1}{\partial v_0} = v_{-1} = 1$
$v_2 = e^{v1} = e^2$	$\frac{\partial v_2}{\partial v_1} = e^{v_1} = e^2$
$v_3 = v_1 + v_2 = 2 + e^2$	$\frac{\partial v_3}{\partial v_1} = 1$ $\frac{\partial v_3}{\partial v_2} = 1$

Reverse pass:

Adjoint
$\bar{v}_{-1} = \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial v_{-1}} = \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = (1 + e^2) \cdot 2 = \frac{\partial f}{\partial v_{-1}} = \frac{\partial f}{\partial x_1}$
$\bar{v}_0 = \frac{\partial f}{\partial v_1} \frac{\partial v_1}{\partial v_0} = \bar{v}_1 \frac{\partial v_1}{\partial v_0} = 1 + e^2 = \frac{\partial f}{\partial v_0} = \frac{\partial f}{\partial x_2}$
$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_2} \frac{\partial v_2}{\partial v_1} = \bar{v}_1 + \bar{v}_2 \frac{\partial v_2}{\partial v_1} = 1 + e^2$ (second child update)
$\bar{v}_1 = \frac{\partial f}{\partial v_3} \frac{\partial v_3}{\partial v_1} = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = 1$ (first child)
$\bar{v}_2 = \frac{\partial f}{\partial v_3} \frac{\partial v_3}{\partial v_2} = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1$
$\bar{v}_3 = \frac{\partial f}{\partial v_3} = \frac{\partial v_3}{\partial v_3} = 1$

independent variables

dependent variables

REVERSE MODE: EXAMPLE

Now ***reverse mode***:

Forward pass: Intermediate	Partial Derivatives	*Reverse pass:* Adjoint
$v_{-1} = x_1 = 1$		$\bar{v}_{-1} = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial v_{-1}} = \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = (1+e^2)\cdot 2 = \frac{\partial f}{\partial v_{-1}} = \frac{\partial f}{\partial x_1}$
$v_0 = x_2 = 2$		$\bar{v}_0 = \frac{\partial f}{\partial v_1}\frac{\partial v_1}{\partial v_0} = \bar{v}_1 \frac{\partial v_1}{\partial v_0} = 1 + e^2 = \frac{\partial f}{\partial v_0} = \frac{\partial f}{\partial x_2}$
$v_1 = v_{-1}v_0 = 2$	$\frac{\partial v_1}{\partial v_{-1}} = v_0 = 2$ $\frac{\partial v_1}{\partial v_0} = v_{-1} = 1$	$\bar{v}_1 = \bar{v}_1 + \frac{\partial f}{\partial v_2}\frac{\partial v_2}{\partial v_1} = \bar{v}_1 + \bar{v}_2 \frac{\partial v_2}{\partial v_1} = 1 + e^2$ (second child update)
$v_2 = e^{v1} = e^2$	$\frac{\partial v_2}{\partial v_1} = e^{v_1} = e^2$	$\bar{v}_1 = \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_1} = \bar{v}_3 \frac{\partial v_3}{\partial v_1} = 1$ (first child)
$v_3 = v_1 + v_2 = 2 + e^2$	$\frac{\partial v_3}{\partial v_1} = 1$ $\frac{\partial v_3}{\partial v_2} = 1$	$\bar{v}_2 = \frac{\partial f}{\partial v_3}\frac{\partial v_3}{\partial v_2} = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1$
		$\bar{v}_3 = \frac{\partial f}{\partial v_3} = \frac{\partial v_3}{\partial v_3} = 1$

independent variables | *dependent* variables

We only need 1 ***reverse mode*** pass to compute the gradient ∇f (forward + reverse pass is considered *one* reverse mode pass). Compare this to forward mode *if* $m \gg 1$.

REVERSE MODE: EXAMPLE

Observations:

- ***Forward mode*** computes the gradient with respect to the *independent* variables: $\boxed{\nabla_x f}$.
- ***Reverse mode*** computes the gradient with respect to the coordinates v: $\boxed{\nabla_v f}$. Because we have chosen $v_{j-m} = x_j$ for $j = 1, 2, \ldots, m$, the gradient $\nabla_x f$ *is a subset of* $\nabla_v f$*!*
- The *computational cost* of forward mode depends on the number of *independent* variables m. The computational cost of reverse mode is independent of that number.

REVERSE MODE: EXAMPLE

Observations:

- In machine learning, the objective function is a *scalar* function with possibly a very large number m of input arguments.
- The *gradient* of the objective function is needed to train the model. A popular and efficient algorithm for this task is called *back-propagation*, which is a special case of reverse mode AD. Special in the sense that the function is scalar and it represents an error between the computed output (hence we compute v_j in the forward pass too) and an expected output.
- If there are many more outputs $n \gg m$ forward mode AD is more efficient.
- If there are many more inputs $m \gg n$ reverse mode AD is more efficient.

AUTOMATIC DIFFERENTIATION: EXERCISE

Given the function $f(x): \mathbb{R}^5 \mapsto \mathbb{R}$ with

$$f(x) = x_1 x_2 x_3 x_4 x_5,$$

compute the gradient ∇f evaluated at the point $x = [2, 1, 1, 1, 1]^\intercal$.

1. Draw the computational graph.
2. Compute the gradient using forward mode. Note: you need $m = 5$ passes with different seed vectors. Write your solution in a evaluation table similar to what we did earlier.
3. Compute the gradient using reverse mode. Write your results in another evaluation table (with possibly fewer columns than forward mode above).
4. For both, forward and reverse mode, calculate the number of arithmetic operations (addition, subtraction, multiplication, division).

You may use the next two pages to write down your solution.
Work together with your neighbors.

AUTOMATIC DIFFERENTIATION: EXERCISE (SOLUTION)

AUTOMATIC DIFFERENTIATION: EXERCISE (SOLUTION)

EXAMPLES FOR EXTENSIONS AND APPLICATIONS

- Up to this point we have discussed the math behind automatic differentiation (chain rule and splitting up a function (evaluation) into elementary parts resulting in computational graph).
- Many extensions and applications exist for an automatic differentiation algorithm.
- We will outline a few here to give you some ideas for your project.

EXAMPLES FOR EXTENSIONS

- Higher order and mixed derivatives:
  - Laplacian operator $\Delta f = \nabla \cdot (\nabla f)$
  - Mixed derivatives $\frac{\partial^2 f}{\partial x_1 \partial x_2}$
  - Hessian matrix which is the Jacobian of the gradient of a scalar function f, that is $\nabla(\nabla f)$
- Computational optimizations:
  - Efficient graph storage and data structure design/traversal
  - Hybrid graph storage model: writing parts of a large graph to (slow) disks and keeping "hot" graph parts in memory
- Combining forward mode and reverse mode
- Exploiting sparsity in the Jacobian and/or Hessian matrices (graph coloring)
- Non-differentiable functions

EXAMPLES FOR APPLICATIONS

There are many applications of AD, below are just a few. See also autodiff.org.

- Numerical solution of Ordinary Differential Equations (ODEs):
  - integration of *stiff* ode systems
  - Newton's method for the solution of non-linear systems of equations (requires Jacobian-vector products)
- Optimization:
  - Optimize an object function (also know as loss or cost function)
  - These techniques require the gradient of the loss function with respect to its parameters
- Solution of linear systems:
  - Iterative methods are powerful algorithms for solving linear systems
  - Some iterative methods require information obtained through derivatives, for example, steepest gradient descent, conjugate gradient or biconjugate gradient methods.

RECAP

- Review of complex numbers and introduction of dual numbers
- Implementation of forward mode AD: operator overloading
- Reverse mode of AD
- Examples for application