# Auto-Differentiation

## At the Intersection of Nifty and Obvious

A.C.

March 28, 2021

# Outline

# Univariate Derivatives

- The instantaneous rate of change of $f$ in response to infinitesimal perturbations in $x$.
- The slope of the tangent line through $(x, f(x))$.

## Univariate Derivatives

- The instantaneous rate of change of $f$ in response to infinitesimal perturbations in $x$.
- The slope of the tangent line through $(x, f(x))$.

### Definition

Let $f : \mathbb{R} \to \mathbb{R}$. We say that $f$ is differentiable wherever the limit

$$f' = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

exists and we call $f'$ the derivative of $f$.

# Derivatives of Transformations

Linearity:

$$(\alpha f + \beta g)' = \alpha f' + \beta g'$$

# Derivatives of Transformations

Linearity:

$$(\alpha f + \beta g)' = \alpha f' + \beta g'$$

Product rule:

$$(fg)' = f'g + fg'$$

## Derivatives of Transformations

Linearity:

$$(\alpha f + \beta g)' = \alpha f' + \beta g'$$

Product rule:

$$(fg)' = f'g + fg'$$

Quotient rule:

$$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$

# Derivatives of Transformations

Linearity:

$$(\alpha f + \beta g)' = \alpha f' + \beta g'$$

Product rule:

$$(fg)' = f'g + fg'$$

Quotient rule:

$$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$

Chain rule:

$$(f \circ g)' = g' \cdot (f' \circ g)$$

## Gradients

- Lots of interesting functions operate on multiple inputs.

## Gradients

- Lots of interesting functions operate on multiple inputs.
- Idea: take the derivative with respect to one input at at time.
  - Pretend the other inputs don't exist, or rather, are constant.
  - Collate it all into a vector at the end.

# Gradients

- Lots of interesting functions operate on multiple inputs.
- Idea: take the derivative with respect to one input at at time.
  - Pretend the other inputs don't exist, or rather, are constant.
  - Collate it all into a vector at the end.

## Definition

Let $f : \mathbb{R}^n \to \mathbb{R}$, and $u_i$ be the $i$th Cartesian unit vector in $\mathbb{R}^n$. We say that $f$ is differentiable wherever the limit

$$\frac{\partial f}{\partial x_i} = \nabla f_i = \lim_{h \to 0} \frac{f(x + hu_i) - f(x)}{h}$$

exists for all $i \in [1, n]$. We call $\nabla f$ the gradient of $f$.

# Jacobians

- Some functions have multiple outputs, too.
- Similar trick:
  - Compute the gradient for each output
  - Glue the gradients to one another
  - Give the resulting matrix a fancy name

# Jacobians

- Some functions have multiple outputs, too.
- Similar trick:
  - Compute the gradient for each output
  - Glue the gradients to one another
  - Give the resulting matrix a fancy name

## Definition

Let $f : \mathbb{R}^n \to \mathbb{R}^m$, be differentiable in each of its outputs at $x$. We define the Jacobian of $f$ to be the matrix such that

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

# Outline

# First Piece

- Differentiation is a flowchart.
  - An *easy* flowchart: `https://xkcd.com/2117/`

# First Piece

- Differentiation is a flowchart.
  - An *easy* flowchart: `https://xkcd.com/2117/`
- Flowcharts are programs.

# Second Piece

- Numerical programs may include loops, branches, etc
  - Not always easy to express in closed form

# Second Piece

- Numerical programs may include loops, branches, etc
  - Not always easy to express in closed form
- But when it executes it has to boil down to some finite composition of ALU-executable operations.
  - We know how to differentiate sums, products, differences, etc

# Eureka

- Apply the "differentiation flowchart" to the composite function defined by the *execution* of a numerical program.

# Eureka

- Apply the "differentiation flowchart" to the composite function defined by the *execution* of a numerical program.
- Compute $J_f(x)$ at the same time as $f(x)$!

# Dependent Variables

- Many intermediate calculations in most computer programs.

# Dependent Variables

- Many intermediate calculations in most computer programs.
- Treat intermediate calculations as anonymous dependent variables.

# Dependent Variables

- Many intermediate calculations in most computer programs.
- Treat intermediate calculations as anonymous dependent variables.
- Keep track of every dependent variable's gradient WRT inputs.
  - Propagate forward with product rule, quotient rule, etc.

# Dependent Variables

- Many intermediate calculations in most computer programs.
- Treat intermediate calculations as anonymous dependent variables.
- Keep track of every dependent variable's gradient WRT inputs.
  - Propagate forward with product rule, quotient rule, etc.
- Use independent variables to bootstrap the calculation
  - $\frac{\partial x_i}{\partial x_j} = I_{i=j}$

# Dep Vars Example

Consider the function $f(x, y) = x^2y + 2y$, and let

$$g_1 = x \cdot x \implies \nabla g_1 = x \cdot (1, 0)^T + (1, 0)^T \cdot x$$

# Dep Vars Example

Consider the function $f(x, y) = x^2 y + 2y$, and let

$$g_1 = x \cdot x \implies \nabla g_1 = x \cdot (1, 0)^T + (1, 0)^T \cdot x$$
$$g_2 = g_1 \cdot y \implies \nabla g_2 = g_1 \cdot (0, 1)^T + \nabla g_1 \cdot y$$

# Dep Vars Example

Consider the function $f(x, y) = x^2 y + 2y$, and let

$$g_1 = x \cdot x \implies \nabla g_1 = x \cdot (1, 0)^T + (1, 0)^T \cdot x$$
$$g_2 = g_1 \cdot y \implies \nabla g_2 = g_1 \cdot (0, 1)^T + \nabla g_1 \cdot y$$
$$g_3 = 2y \implies \nabla g_3 = 2 \cdot (0, 1)^T$$

# Dep Vars Example

Consider the function $f(x, y) = x^2 y + 2y$, and let

$$
\begin{aligned}
g_1 &= x \cdot x \implies \nabla g_1 = x \cdot (1,0)^T + (1,0)^T \cdot x \\
g_2 &= g_1 \cdot y \implies \nabla g_2 = g_1 \cdot (0,1)^T + \nabla g_1 \cdot y \\
g_3 &= 2y \implies \nabla g_3 = 2 \cdot (0,1)^T \\
f &= g_2 + g_3 \implies \nabla f = \nabla g_2 + \nabla g_3
\end{aligned}
$$

Exercise for the reader: confirm that $\nabla f = (2xy, x^2 + 2)^T$

# Code I

```
class _FDepVar(object):
    def __init__(self, val, grad):
        self.val = val
        self.grad = grad

    def __add__(f, g):
        if not isinstance(g, _FDepVar):
            return _FDepVar(f.val + g, f.grad)
        return _FDepVar(f.val + g.val, f.grad + g.grad)

    def __mul__(f, g):
        if not isinstance(f, _FDepVar):
            return _FDepVar(f.val*g, f.grad*g)
        return _FDepVar(f.val*g.val,
                        f.grad*g.val + f.val*g.grad)
```

# Code II

```
def forward(f):
    def f_J(*args):
        wrapped = []
        for i, arg in enumerate(args):
            grad = np.zeros(len(args))
            grad[i] = 1
            wrapped.append(_FDepVar(arg, grad))
        out = f(*wrapped)
        try:
            return ([o.val for o in out],
                    [o.grad for o in out])
        except TypeError:
            return out.val, out.grad
    return f_J
```

# Caveats

- Every scalar operation in our program now a vector op
- Calculation is $n$ times more expensive, where $n$ is our number of inputs.
  - No big deal for auto-diffing $f(x, y, z)$
  - But if $f$ takes thousands of parameters ... hoo boy

# Key Idea

- Derivatives don't have to be taken WRT input variables

# Key Idea

- Derivatives don't have to be taken WRT input variables
- Forward mode: take $\frac{dg}{dx}$, $\frac{dh}{dx}$, build $\frac{df}{dx}$
  - Derivatives of different functions with respect to $x$

# Key Idea

- Derivatives don't have to be taken WRT input variables
- Forward mode: take $\frac{dg}{dx}, \frac{dh}{dx}$, build $\frac{df}{dx}$
  - Derivatives of different functions with respect to $x$
- Backward mode: take $\frac{df}{dg}, \frac{df}{dh}$, build $\frac{df}{dx}$
  - Derivatives of $f$ with respect to different variables.
  - Multivariate chain rule is central:

$$\frac{df(g(x), h(x))}{dx} = \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial h} \cdot \frac{dh}{dx}$$

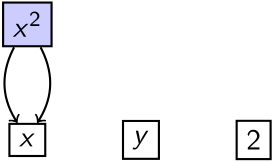
# Calculation Graph

Calculations naturally form a DAG as they feed into one another:

$\boxed{x}$ $\boxed{y}$ $\boxed{2}$

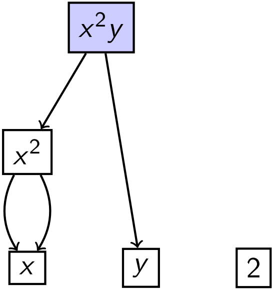
# Calculation Graph

Calculations naturally form a DAG as they feed into one another:



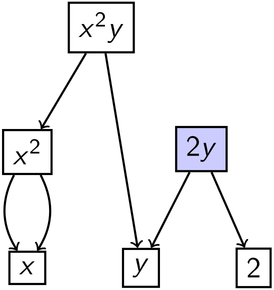
# Calculation Graph

Calculations naturally form a DAG as they feed into one another:



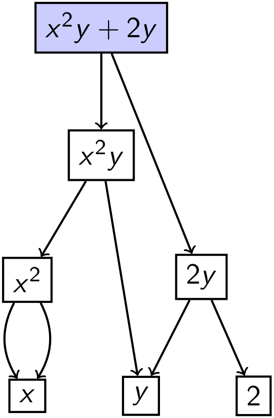
# Calculation Graph

Calculations naturally form a DAG as they feed into one another:



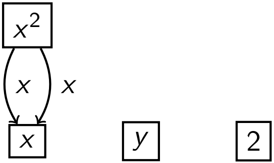
# Calculation Graph

Calculations naturally form a DAG as they feed into one another:



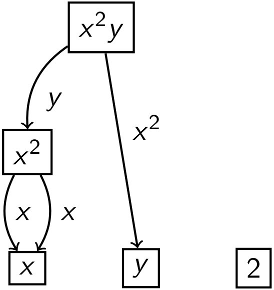
# Edge Weights

Edges provide a convenient place to record our various $\frac{\partial g_i}{\partial g_j}$s:



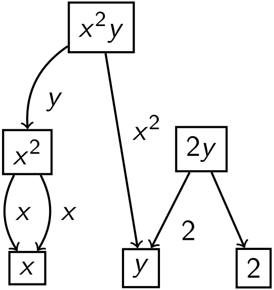
# Edge Weights

Edges provide a convenient place to record our various $\frac{\partial g_i}{\partial g_j}$s:



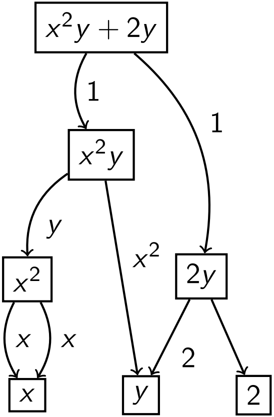
# Edge Weights

Edges provide a convenient place to record our various $\frac{\partial g_i}{\partial g_j}$s:



# Edge Weights

Edges provide a convenient place to record our various $\frac{\partial g_i}{\partial g_j}$s:



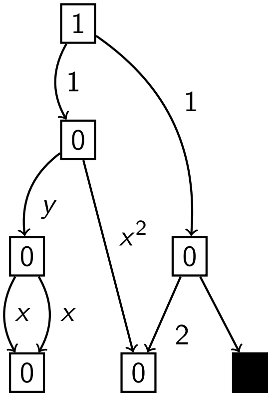
# Propagation

- If we know all of $\frac{df}{d\text{parent}}$, then by the chain rule $\frac{df}{d\text{self}}$ is just the weighted sum of its parents.
  - Root node always knows its derivative: $\frac{df}{df} = 1$

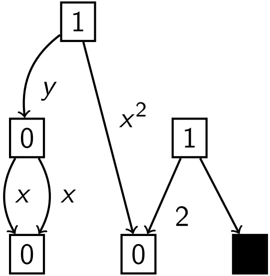
# Propagation

- If we know all of $\frac{df}{d\text{parent}}$, then by the chain rule $\frac{df}{d\text{self}}$ is just the weighted sum of its parents.
  - Root node always knows its derivative: $\frac{df}{df} = 1$
- Note: DAG-order traversal, not just a DFS.
  - we can't visit a node until all its parents are done.
  - we have to traverse all edges, not visit every node.

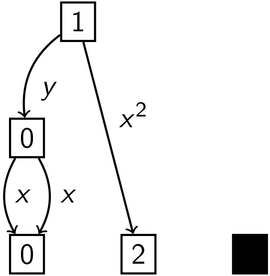
# Propagation Example



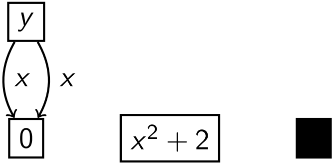
# Propagation Example



# Propagation Example



# Propagation Example



# Propagation Example

$\boxed{2xy}$ $\boxed{x^2 + 2}$

# Code

- The code for backward mode is a little less slide-deck friendly than for forward mode, so we'll skip it here.
- A (crude) implementation is available in this presentation's git repository for those interested in perusing it.

# Caveats

- Have to propagate derivatives backward for every output.
  - Calculation is $m$ times more expensive, where $m$ is our number of outputs.

# Caveats

- Have to propagate derivatives backward for every output.
  - Calculation is $m$ times more expensive, where $m$ is our number of outputs.
- Have to remember the calculation graph.
  - Requires additional space proportional to the length of the computation.

# Outline

# Disclaimer

*ACHTUNG*: Aggressive, true-from-10k-feet-but-not-below-that simplifications ahead.

# ML $\approx$ Optimization

- Most ML boils down to one of
  - Classification: Find me the best decision boundary for all this the training data I've marked as positive or negative.
  - Regression: Find me the function that best matches these data points I measured.

# ML $\approx$ Optimization

- Most ML boils down to one of
  - Classification: Find me the best decision boundary for all this the training data I've marked as positive or negative.
  - Regression: Find me the function that best matches these data points I measured.
- If we define a loss function $\mathcal{L}$, and model parameters $\theta$ for tuning it, and call our training data $X$, $Y$, then we can formulate them as

$$\arg\min_{\theta} \mathcal{L}(X, Y; \theta)$$

  - E.g. in linear regression, $\mathcal{L} = ||X^T\theta - y||_2^2$

## Gradient Descent in a Nutshell

- When you're on a hill and want to get to the bottom, *walk downhill*.

## Gradient Descent in a Nutshell

- When you're on a hill and want to get to the bottom, *walk downhill*.
- The gradient of your loss function tells you which way is downhill.

$$\theta_n = \theta_{n-1} - h_n \nabla \mathcal{L}(\theta_{n-1})$$

# Neural Networks & Backpropagation

- Neural Networks are commonly optimized (trained) using gradient descent.

# Neural Networks & Backpropagation

- Neural Networks are commonly optimized (trained) using gradient descent.
- But formulating the gradient is a PITA, sure would be nice if we could generate the gradient at a point programmatically and efficiently . . .

# Neural Networks & Backpropagation

- Neural Networks are commonly optimized (trained) using gradient descent.
- But formulating the gradient is a PITA, sure would be nice if we could generate the gradient at a point programmatically and efficiently . . .
- Which is exactly what autodiff does.
  - Neural networks have many, many parameters to train on, but we only have one loss function, so backward mode is a no-brainer.
  - Frequently called backpropagation or backprop for historical reasons.

## References and Additional Materials

- The source for this presentation is hosted at `https://github.com/alan-christopher/autodiff-edu`.
- A recorded presentation of this slide deck can be found at `https://youtu.be/Fntkxhcs3mU`

# Questions?