

# Autograd: A cool software trick for avoiding maths

Pashmina Cameron

Microsoft Research, Cambridge ([aka.ms/pashmina](https://aka.ms/pashmina))

# Why?

Most parameter optimization involves taking gradients

$$f(x) = \frac{1}{x} * e^x$$

$$f'(x) = \frac{e^x}{x} \left( 1 - \frac{1}{x} \right)$$

```
def f(x):  
    return (1/x) * exp(x)
```

```
def df(x):  
    return (exp(x)/x)*(1-1/x)
```

```
y = f(x)  
z = df(x)
```

# Why?

Most parameter optimization involves taking gradients

$$f(x) = \frac{1}{x} * e^x$$

$$f'(x) = \frac{e^x}{x} \left( 1 - \frac{1}{x} \right)$$

*Get away with writing  
that instead of  
implementing df(x)*

```
def f(x):  
    return (1/x) * exp(x)
```

```
def df(x):  
    return (exp(x)/x)*(1-1/x)
```

```
df = grad(f) # Autodiff!
```

```
y = f(x)  
z = df(x)
```

# What AutoDiff isn't

## Finite differences

- Expensive (Multiple evaluations for each partial derivative)
- Unstable (Involves division of a small number by a small number)

## Symbolic differentiation

- Returns actual derivatives, not an expression for them (Mathematica)
- Keeps track of repeated expression evaluation

# What AutoDiff is

A source of exact analytical gradients

- Returns actual derivatives

Efficient

- Uses memoization (stores intermediate values for later reuse)

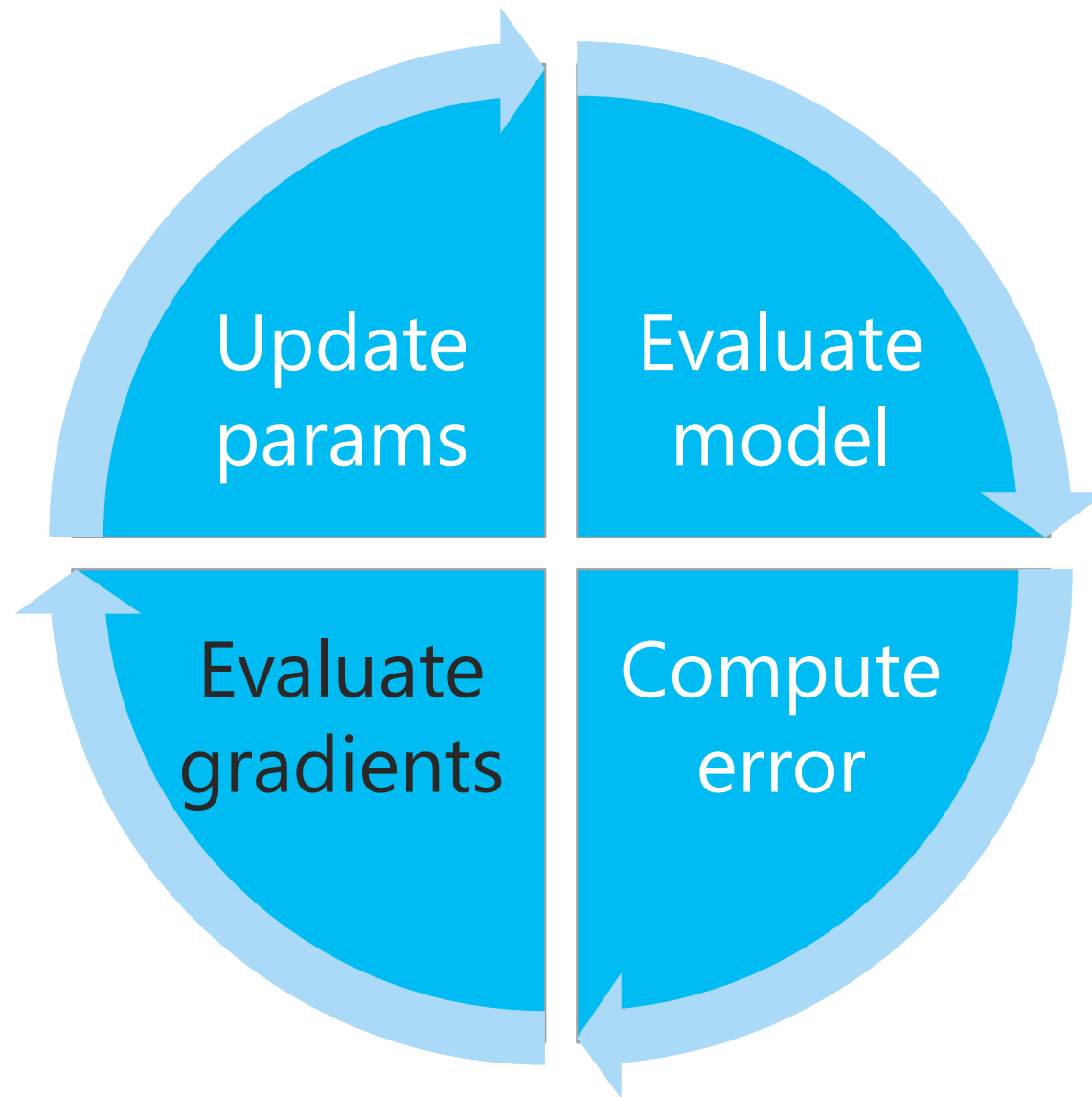
# Terminology

## Back propagation (Backprop)

- Optimizes model parameters using repeated application of the chain rule of derivatives
- Can be implemented efficiently using reverse-mode differentiation

AutoGrad is a particular AutoDiff package

# Optimization using backprop



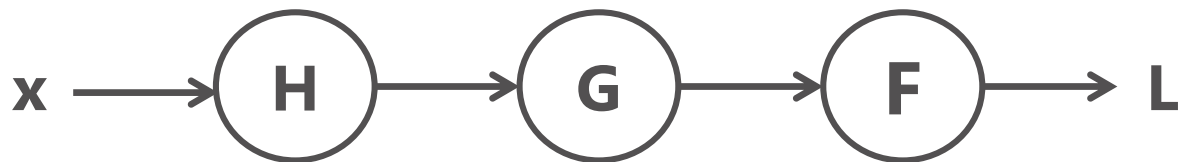
# An example

Given a nested function

$$L = F(G(H(x)))$$

Derivative using Chain rule

$$\frac{dL}{dx} = \frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x}$$



## Forward mode

$$dL = \frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x} * dx$$

←  
Output                      Input



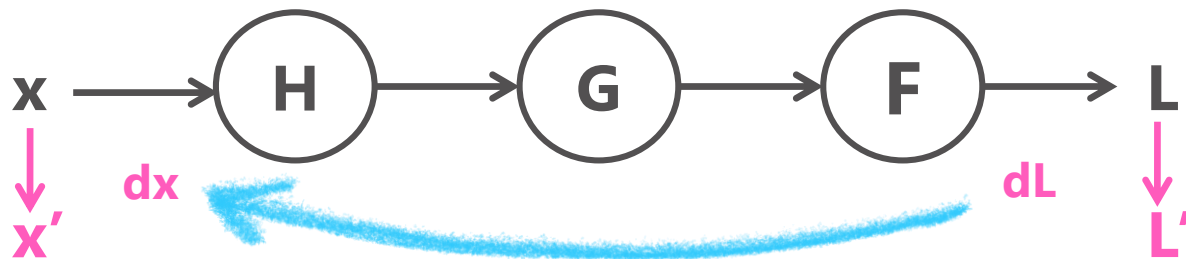
# An example

Given a nested function

$$L = F(G(H(x)))$$

Derivative using Chain rule

$$\frac{dL}{dx} = \frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x}$$



## Forward mode

$$dL = \frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x} * dx$$

←  
Output                      Input

## Reverse mode

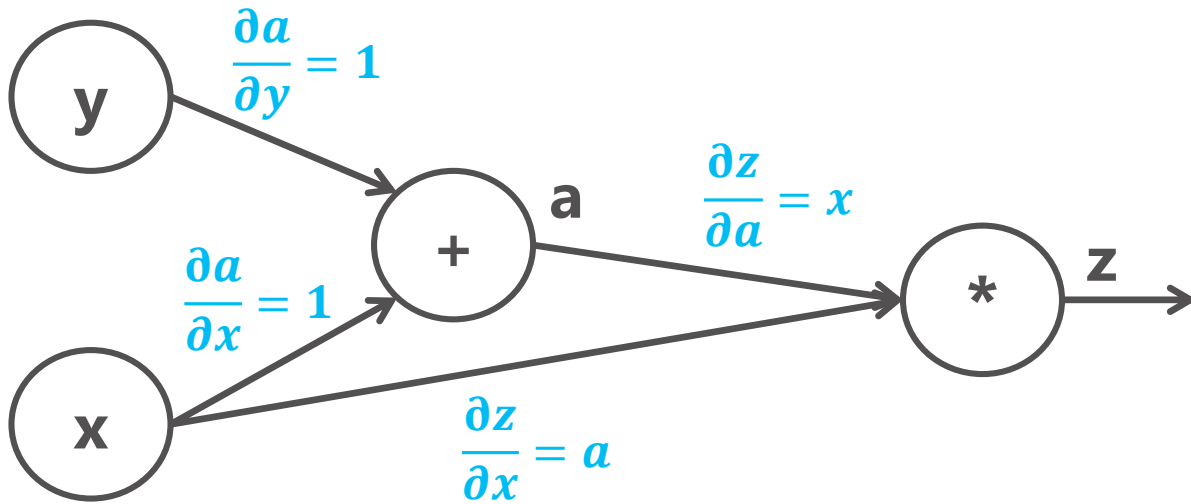
$$\frac{d}{dx} = \frac{d}{dL} * \frac{\partial F}{\partial G} * \frac{\partial G}{\partial H} * \frac{\partial H}{\partial x}$$

→  
Output                      Input

# Multipath case

$$a(x, y) = x + y$$

$$z(a, x) = a * x$$



Computes derivative of outputs wrt an input

$$\nabla z = \left( \frac{\partial z(x, y, a)}{\partial x}, \frac{\partial z(x, y, a)}{\partial y} \right)$$

$$\frac{\partial z(x, y, a)}{\partial x} = \frac{\partial z}{\partial x} + \frac{\partial z}{\partial a} * \frac{\partial a}{\partial x}$$

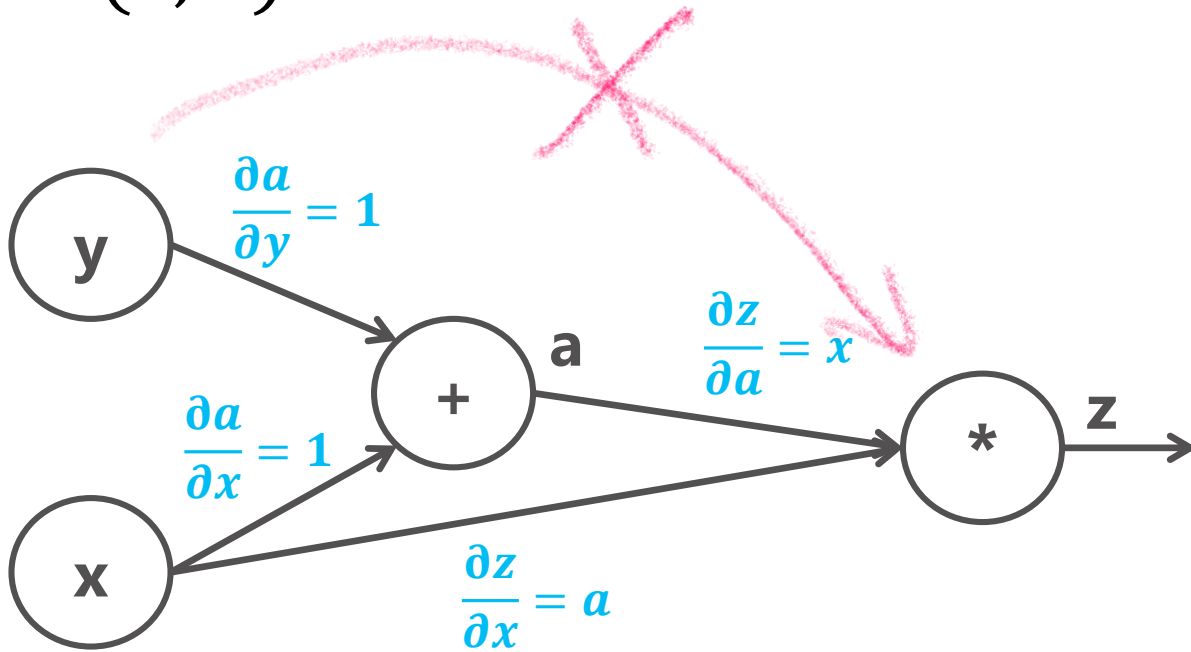
$$\frac{\partial z(x, y, a)}{\partial y} = \frac{\partial z}{\partial y} + \frac{\partial z}{\partial a} * \frac{\partial a}{\partial y}$$

Total derivative rule

# Multipath case

$$a(x, y) = x + y$$

$$z(a, x) = a * x$$



Computes derivative of outputs wrt an input

$$\nabla z = \left( \frac{\partial z(x, y, a)}{\partial x}, \frac{\partial z(x, y, a)}{\partial y} \right)$$

$$\frac{\partial z(x, y, a)}{\partial x} = \frac{\partial z}{\partial x} + \frac{\partial z}{\partial a} * \frac{\partial a}{\partial x}$$

$$\frac{\partial z(x, y, a)}{\partial y} = \cancel{\frac{\partial z}{\partial y}} + \frac{\partial z}{\partial a} * \frac{\partial a}{\partial y}$$

Total derivative rule

# Prod

```
class ProdNode(object):
```

```
    def forward(self, a, x):
```

```
        self.x = x      # store a,x for use
```

```
        self.a = a      # in backward later
```

```
        z = a * x
```

```
        return z
```

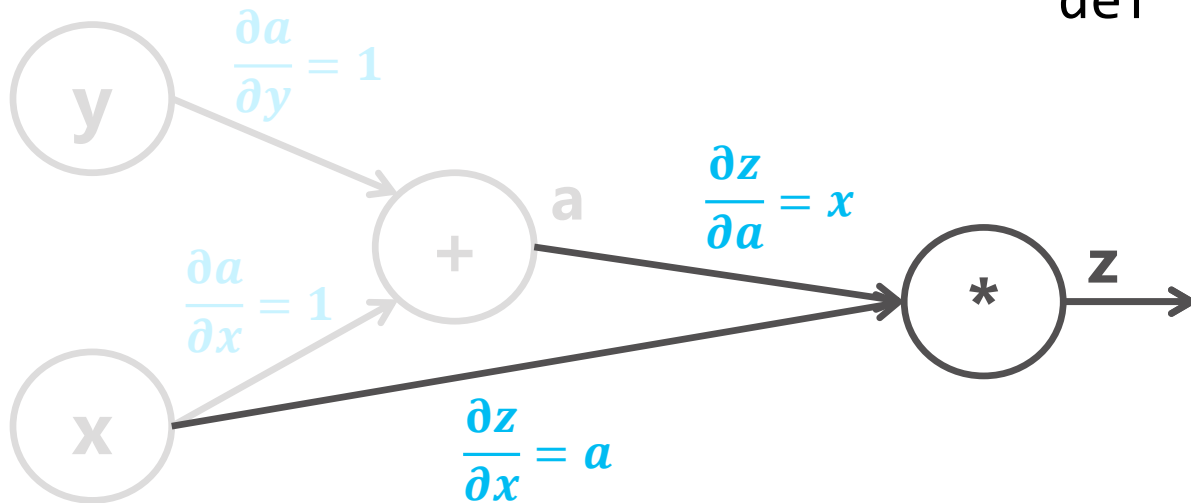
$$z = a * x$$

```
    def backward(self, dz):
```

```
        da = self.x * dz
```

```
        dx1 = self.a * dz
```

```
        return da, dx1
```



Memoization needed to access  $x$  and avoid recomputing  $a$

# Add

```
class AddNode(object):
```

```
    def forward(self, x, y):
```

```
        a = x + y
```

```
        return a
```

$$a = x + y$$

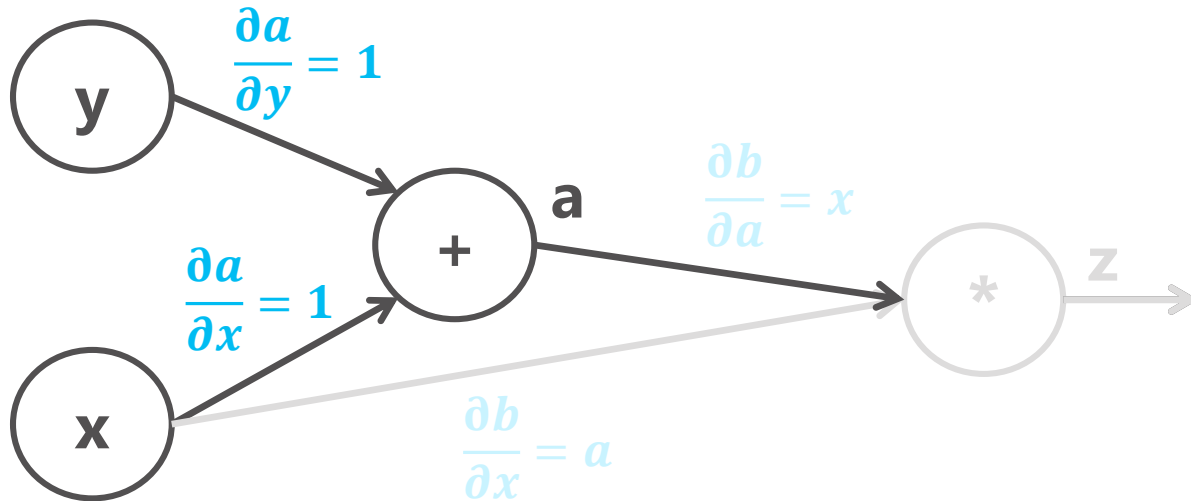
```
    def backward(self, da):
```

```
        dx2 = 1 * da
```

```
        dy = 1 * da
```

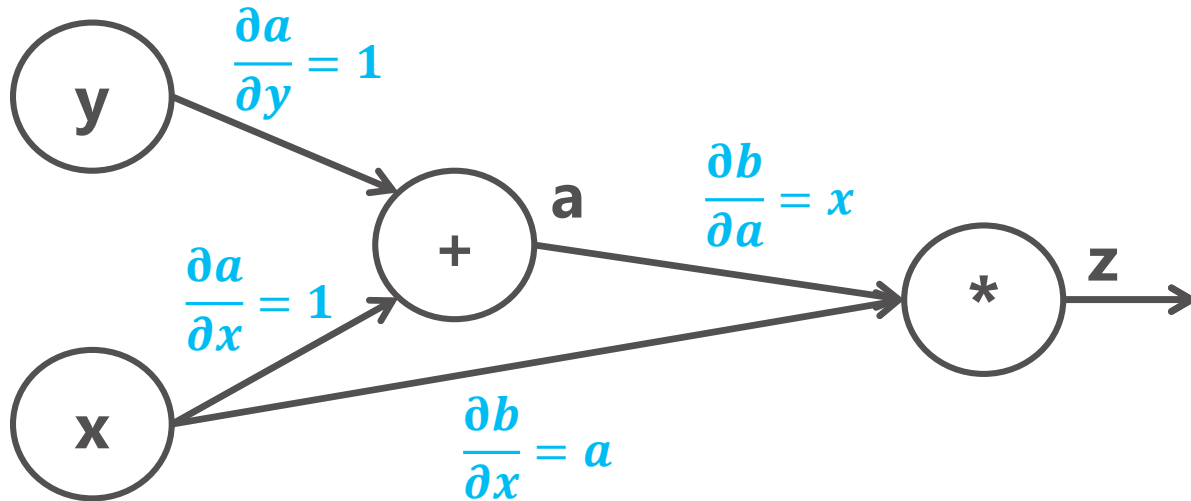
```
        return dx2, dy
```

No memoization needed in this case



# Combine

$$a = x + y$$



```
for i in range(num_steps):  
    # Evaluate at current params  
    a = add.forward(x,y)  
    z = prod.forward(a,x)  
    dz = z - z_hat  
    # Compute param update  
    da, dx2 = prod.backward(dz)  
    dx1, dy = add.backward(da)  
    dx = dx1 + dx2  
    # Update params  
    x = x - lr*dx  
    y = y - lr*dy
```

# Autograd package

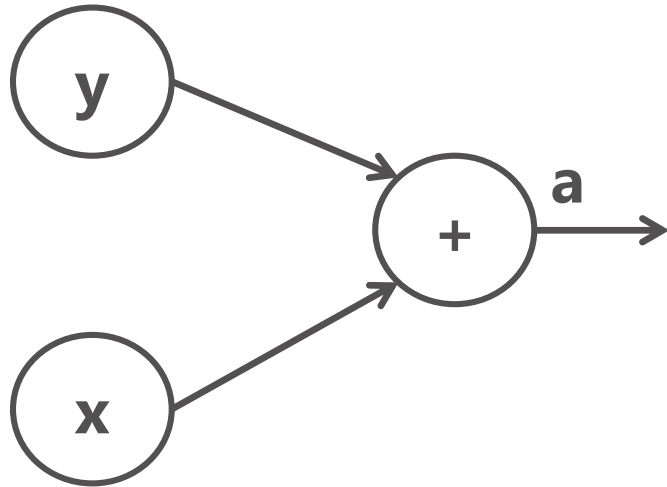
- Lightweight automatic differentiation package that automates the computation of gradients of a function
- Function must be expressed using common python, numpy and scipy primitives (those with `.deriv` defined)

## Alternatives

- AutoDidact, only 200 lines of Python
- PyTorch, JAX: GPU support

# Autograd: Simple

$$a = x + y$$



```
from autograd import grad
def add(x, y):
    return x + y
```

```
grad_add = grad(add, (0,1))
```

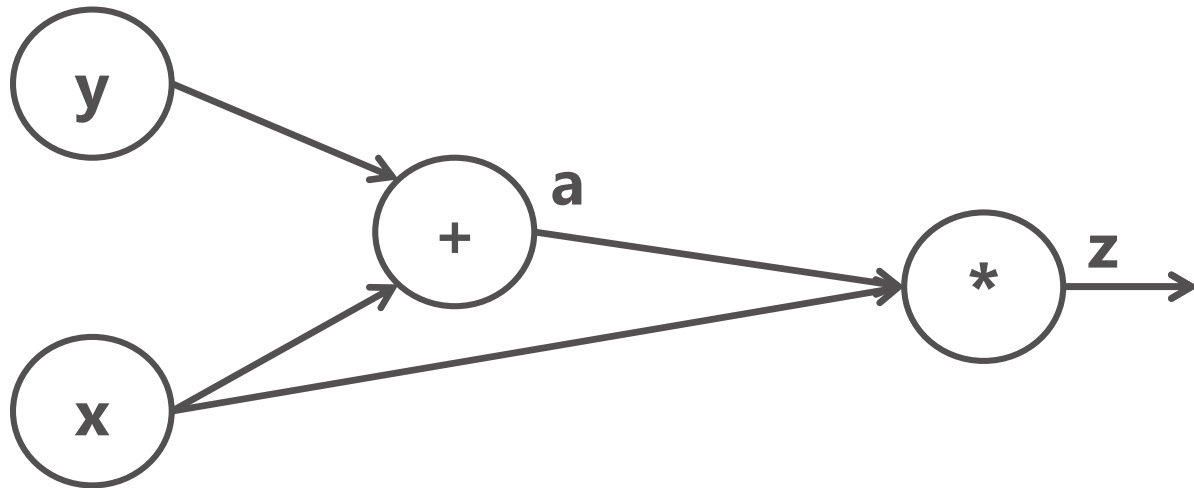
```
for i in range(num_steps):
    # Evaluate at current params
    a = add(x,y)
    da = np.float64(a - a_hat)
    # Compute param update
    grads = grad_add(x,y)
    dx, dy = (g*da for g in grads)
    # Update params
    x = x - lr*dx
    y = y - lr*dy
```



# Autograd: Nested

$$a = x + y$$

$$z = x * y$$



```
from autograd import grad
add = lambda x, y: x + y
node = lambda x, y: x * add(x, y)
```

```
grad_node = grad(node, (0, 1))
```

```
for i in range(num_steps):
    # Evaluate at current params
    z = node(x, y)
    dz = np.float64(z - z_hat)
    # Compute param update
    grads = grad_node(x, y)
    dx, dy = (g*dz for g in grads)
    # Update params
    x = x - lr*dx
    y = y - lr*dy
```

# PyTorch

*# AutoGrad*

```
for i in range(50):  
    z = net(x,y)  
    dz = np.float64(z - z_hat)  
    # assumes dz = 1.0  
    # i.e. returns dx/da and dy/da  
    # so post-multiply by dz  
    grads = grad_net(x,y)  
    dx, dy = (g*dz for g in grads)  
  
    x = x - lr*dx  
    y = y - lr*dy
```

*# PyTorch*

```
model = Net().to(device)  
optimiser = optim.SGD(model.parameters(), lr=lr, momentum=momentum)  
model.train()  
for i, (data, target) in enumerate(train_loader):  
    data, target = data.to(device), target.to(device)  
    # sets w.grad = 0 for all params w  
    # this step does not affect the computational graph,  
    # only changes the values of params w  
    optimiser.zero_grad()  
    # computational graph is created here, during the fwd pass  
    # the computation graph is available through loss.grad_fn  
    output = model(data)  
    loss = F.nll_loss(output, target)  
    # computes dloss/dw for every param w with requires_grad=true  
    # the computational graph in loss.grad_fn  
    # is used to compute gradients  
    loss.backward()  
    # updates the value of w using w.grad  
    # For SGD,  $w \leftarrow w - lr * w.grad$   
    # this step does not affect the computational graph  
    optimiser.step()
```

# AutoGrad applications

## Fluid simulation

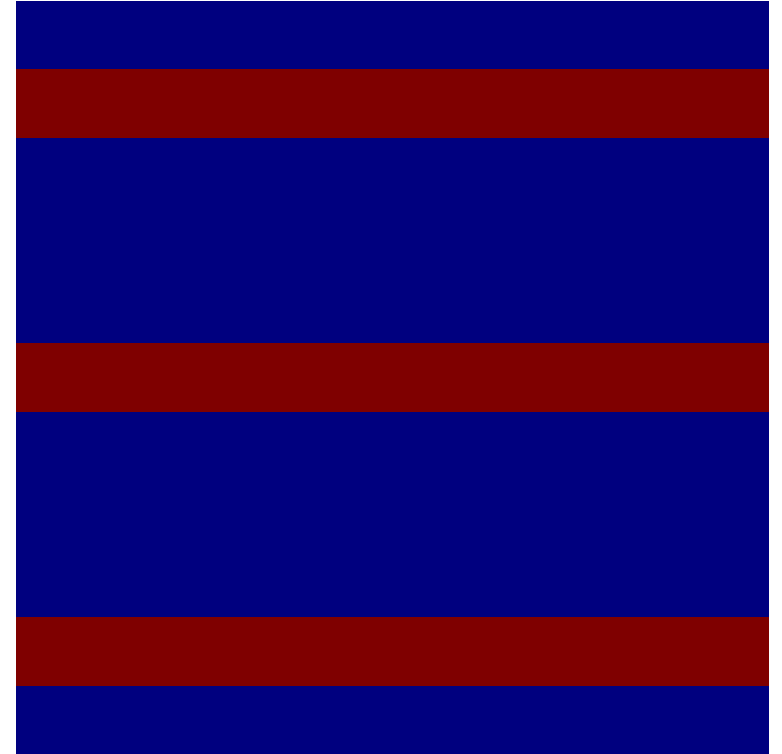
Simulates realistic fluid flows by solving Navier-Stokes equations for optimal visual quality

Source: [github.com/HIPS/autograd/](https://github.com/HIPS/autograd/)

## Electromagnetic simulation

Calculates the effect of a single input parameter on simulation output using forward mode

Forward-Mode Differentiation of Maxwell's Equations [arXiv:1908.10507](https://arxiv.org/abs/1908.10507) PyPI: ceviche

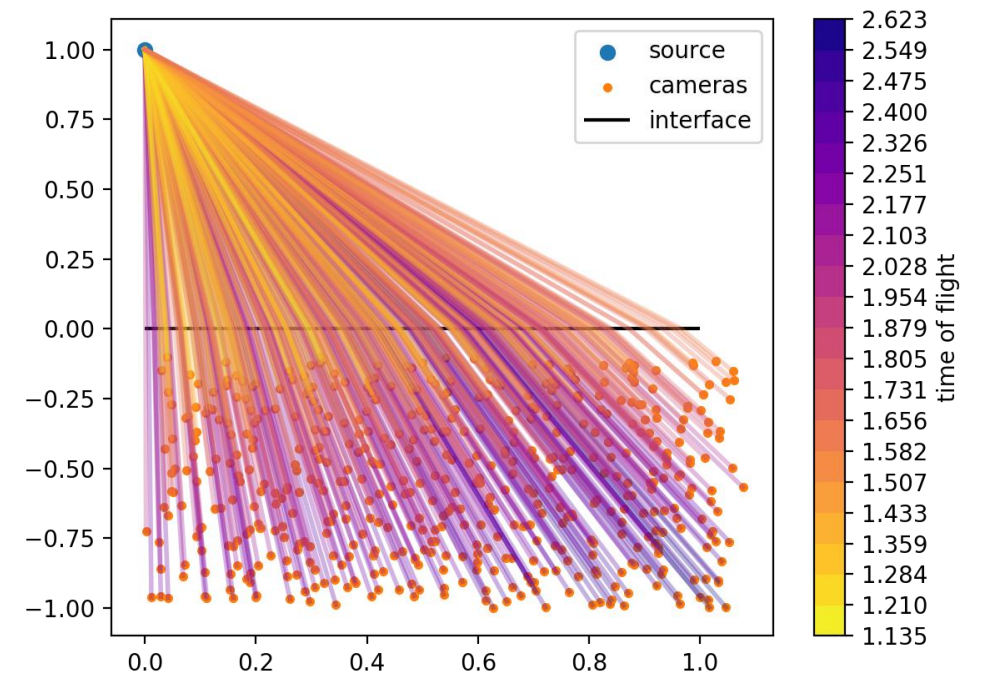


# AutoGrad applications

## Ray tracing

Finds ray with minimum time of flight

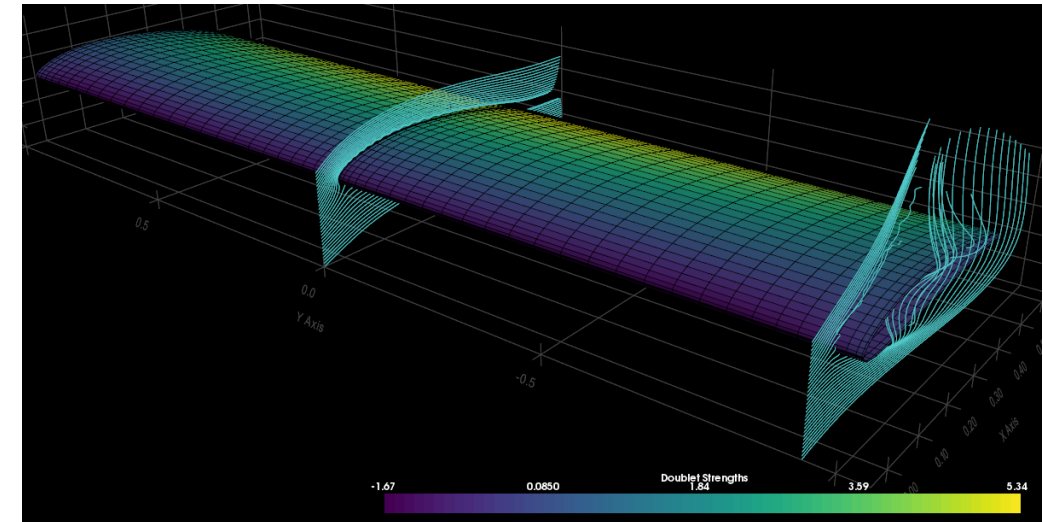
Source: <http://flothesof.github.io/ray-tracing-numpy-autograd.html>



## Aircraft design

Calculates aerodynamic performance & sensitivity wrt large number of design variables

Source: [github.com/peterdsharpe/AeroSandbox](https://github.com/peterdsharpe/AeroSandbox)



# Recent advances

- *PyTorch (Python)*  
Direct/Indirect access to AutoGrad, GPU implementation
- *JAX (Python)*  
AutoGrad with XLA-jit for GPU and TPU
- *Message passing (C#)*  
Generalization of approximate automatic differentiation  
Useful when functions don't have closed form analytical derivatives

# References

1. Autograd <https://github.com/HIPS/autograd>
2. AutoDidact <https://github.com/mattjj/autodidact>  
AutoGrad light (200 lines of Python)
3. PyTorch <https://pytorch.org/>  
AutoGrad + GPU support
4. JAX <https://github.com/google/jax>  
AutoGrad + GPU/TPU JIT compilation
5. T Minka, From automatic differentiation to message passing  
<https://tminka.github.io/papers/acml2019/> ACMLL 2019  
Approximate automatic differentiation

Code: <https://notebooks.azure.com/pashmina-cameron/projects/autodiff>