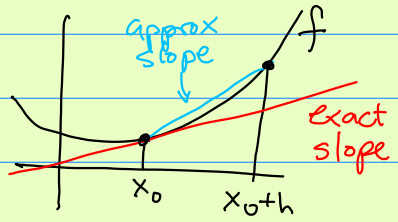
Day 19 Differentiation by finite differences, cont'd. Differentiation arithmetics.

Let's begin by reviewing the error bound for the forward difference quotient in machine arithmetic, and comparing it with the empirically observed error ...



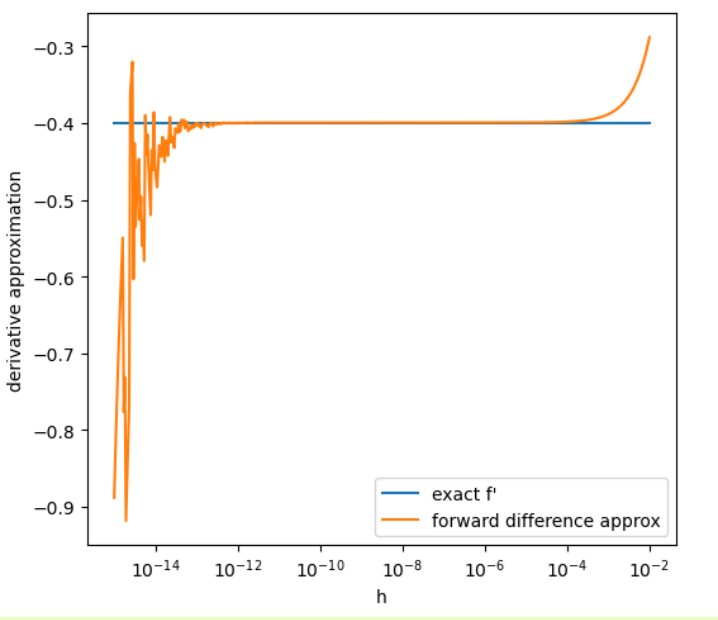
Using the methods we learned with Ch 1,
we obtained the leading order terms in the relative error are:

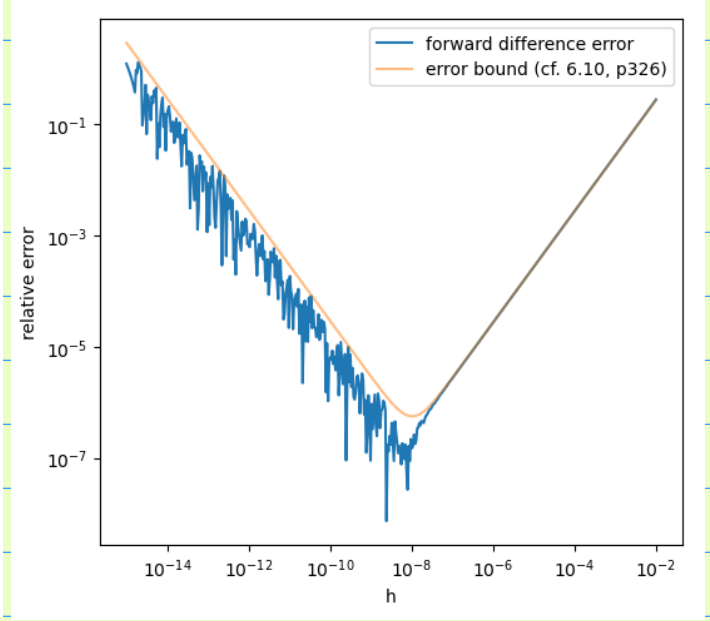
$$|\text{rel error in } f'| \lesssim \frac{1}{h}\left(1 + 2\mu\left|\frac{f}{f'}\right|\right)\frac{\varepsilon_{mach}}{2} + \left(3 + \mu + \left|\frac{f''}{f'}\right|\right)\frac{\varepsilon_{mach}}{2} + \frac{h}{2}\left|\frac{f''}{f'}\right|$$

truncation error previously anticipated

Let us plot this bound along with the actual empirical error
for the same example used to illustrate symbolic differentiation:

```
#%matplotlib notebook
import sympy as sp
x = sp.symbols('x')
y = 3*x + sp.sin(7*x**2/(1+x))  # example from last time
yp = sp.diff(y,x).simplify()
ypp= sp.diff(y,x,x).simplify()
f        = sp.lambdify(x,y ,'numpy')
fp_exact = sp.lambdify(x,yp,'numpy')
fpp_exact = sp.lambdify(x,ypp,'numpy')

h = 10**np.linspace(-15,-2,500)
x = 1.1
fp_fd = (f(x+h)-f(x))/h  # forward difference approximation to derivative of f for many different h values
error = np.abs(fp_fd - fp_exact(x))
relerr = error/np.abs(fp_exact(x))
import matplotlib.pyplot as plt
plt.figure(figsize=(6,12))
plt.subplot(211)
plt.semilogx(h,fp_exact(x)+0*h,label='exact f\'')
plt.semilogx(h,fp_fd,label='forward difference approx')
plt.legend()
plt.xlabel('h'); plt.ylabel('derivative approximation')
plt.subplot(212)
plt.loglog(h,relerr,label='forward difference error')
emach = 2**(-52)
M0 = np.abs(f(x))
M1 = np.abs(fp_exact(x))
M2 = np.abs(fpp_exact(x))

mu = 2 # estimate of multiple of emach/2 in error of evaluating f

relerr_bound = h/2*M2/M1 + emach/2*(1+2*mu*M0/M1)/h  + (3+ mu + M2/M1 )*emach/2
plt.loglog(h,relerr_bound,label='error bound (cf. 6.10, p326)',alpha=0.5)

plt.legend()
plt.xlabel('h'); plt.ylabel('relative error');
```





* Zoom in to see how rounding errors vary with h in detail!

To find the optimal value of h, differentiate wrt h and set to 0:

$$\frac{d}{dh}\,\text{rel error bound} = -\frac{1}{h^2}\left(1 + 2\mu\left|\frac{f}{f'}\right|\right)\frac{\varepsilon_{mach}}{2} + \frac{1}{2}\left|\frac{f''}{f'}\right| \overset{\text{set}}{=} 0$$

$$h = \sqrt{\left|\frac{f'}{f''}\right|\left(1 + 2\mu\left|\frac{f}{f'}\right|\right)\varepsilon_{mach}} = \sqrt{\quad}\sqrt{\varepsilon_{mach}}$$

$\frac{A}{h^2} = B \quad h = \sqrt{\frac{A}{B}}$

Summarizing, the error bound on the forward difference $\frac{f(x+h)-f(x)}{h} \approx f'(x)$ is minimized at

$$h_{opt} = \sqrt{\quad}\sqrt{\varepsilon_{mach}}$$

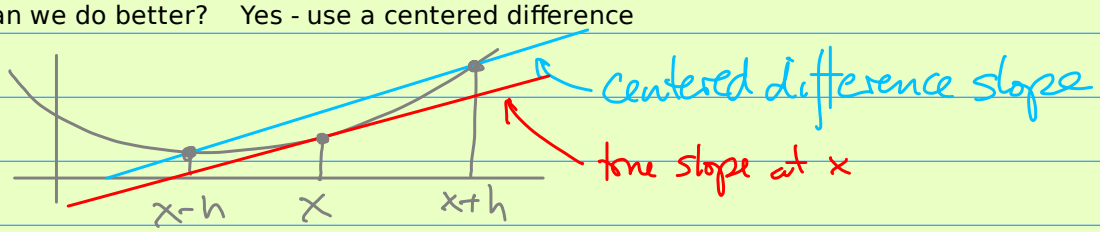
some number depending on f, maybe 5

and from above

$$\text{rel. error}\Big|_{h=h_{opt}} = \text{some number like 5}\ \sqrt{\varepsilon_{mach}}$$ ← huge compared to ε_{mach}

because $\sqrt{\varepsilon_{mach}} \gg \varepsilon_{mach}$!

Can we do better? Yes - use a centered difference



If $f \in C^3$

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(\xi_+)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(\xi_-)$$

Thus

$$\frac{f(x+h)-f(x-h)}{2h} = \frac{2hf'(x) + \frac{h^3}{3}\left(\frac{f'''(\xi_+)+f'''(\xi_-)}{2}\right)}{2h}$$

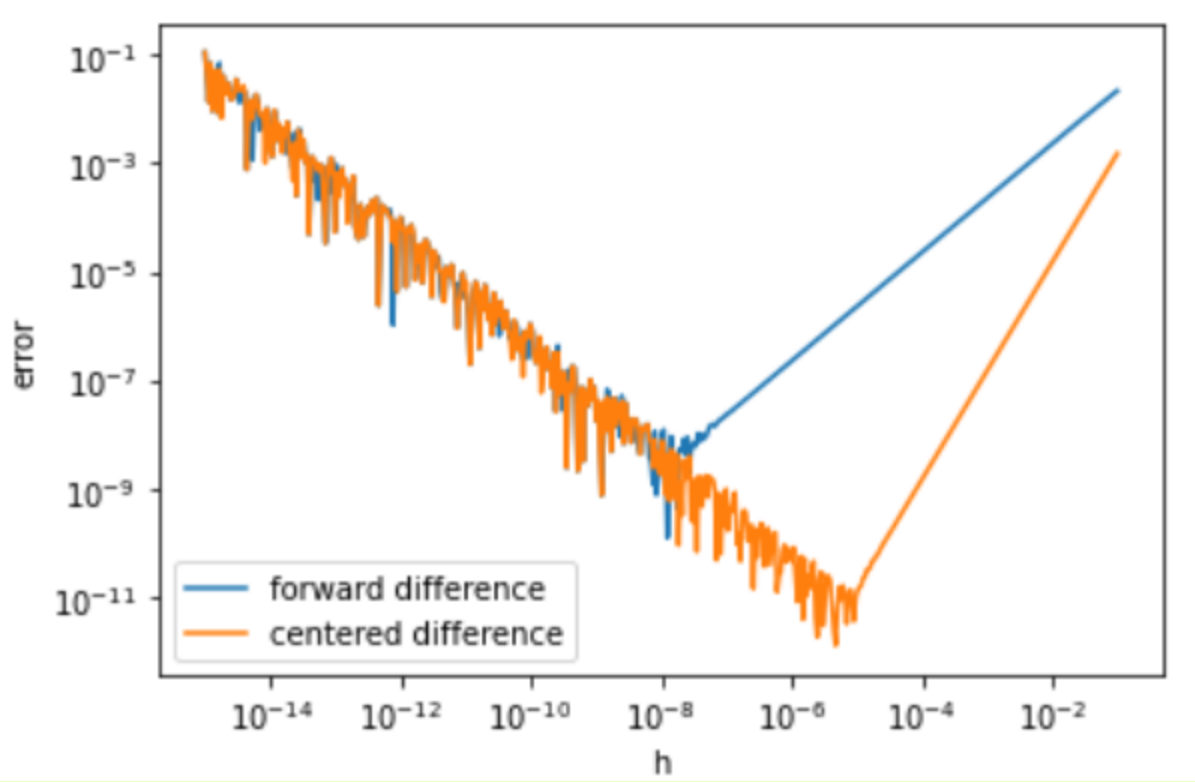
$$= f'(x) + \frac{h^2}{6}f'''(\xi) = f'(x) + O(h^2)$$

Compare with the O(h) error for the forward difference.

Let's do an empirical comparison of the errors in the forward and centered differences ...

Empirical observation of error in centered finite difference (and comparison with one-sided difference):

```
import numpy as np
def f(x): return np.cos(x)
def fp_exact(x): return -np.sin(x)

x = 1.1

h = 10**np.linspace(-15,-1,500)

fp_fd = (f(x+h)-f(x))/h  #2 forward difference approximation to derivative of f for many different h values
fp_cd =(f(x+h)-f(x-h))/(2*h)  # forward difference approximation to derivative of f for many different h val

error = np.abs(fp_fd - fp_exact(x))
error2 = np.abs(fp_cd - fp_exact(x))

import matplotlib.pyplot as plt
plt.loglog(h,error ,label='forward difference')
plt.loglog(h,error2,label='centered difference')
plt.legend()
plt.xlabel('h'); plt.ylabel('error');
```



● "automatic" differentiation (AD), differentiation arithmetics

Would be great if we had another method that has
(i) no truncation error
(ii) no catastrophic round-off error

There is such a thing: we use a "differentiation arithmetic".

Suppose you have 2 functions f & g, and define $h = f\,g$ (their product).

Question: What do we need to know in order to compute say h'(5.7)?

Answer: $f(5.7),\ g(5.7),\ f'(5.7),\ g'(5.7)$

$$h'(5.7) = f'(5.7)\,g(5.7) + f(5.7)\,g'(5.7) \quad \text{(product rule)}$$

for f for g

So develop an arithmetic of objects like < f(5.7), f'(5.7) >, < g(5.7), g'(5.7) >,

for example < 3.4, -1.3 >, <7, -2>,

by defining the product of two objects like this:

$$\langle f_{val}, f_{der}\rangle * \langle g_{val}, g_{der}\rangle = \langle f_{val}\,g_{val},\ f_{der}\,g_{val} + f_{val}\,g_{der}\rangle$$

Compare complex arithmetic: $(f_{re} + i f_{im})(g_{re} + i g_{im})$

$$\langle f_{re}, f_{im}\rangle * \langle g_{re}, g_{im}\rangle$$

$$= \langle f_{re}g_{re} - f_{im}g_{im},\ f_{re}g_{im} + f_{im}g_{re}\rangle$$

Other arithmetic operations, and elementary transcendental functions, similarly ...

$\pm$ $\quad \langle f_{val}, f_{der}\rangle \pm \langle g_{val}, g_{der}\rangle = \langle f_{val} \pm g_{val},\ f_{der} \pm g_{der}\rangle$

$$\langle f_{der}, f_{val}\rangle / \langle g_{val}, g_{der}\rangle = \langle f_{val}/g_{val},\ \frac{g_{val} f_{der} - f_{val} g_{der}}{g_{val}^2}\rangle$$

$$\sin(\langle f_{val}, f_{der}\rangle) = \langle \sin(f_{val}),\ \cos(f_{val}) \cdot f_{der}\rangle$$

[space for other operations in a first-order single-variable differentiation arithmetic]

Example:

Let's do this example, first symbolically so we can check our AD answer.

An example for illustrating a Differentiation Arithmetic

```
x = sp.symbols('x')
y = x**2/(x-13)
y
```

$$\frac{x^2}{x - 13}$$

```
yp = sp.diff(y,x)
yp
```

$$-\frac{x^2}{(x - 13)^2} + \frac{2x}{x - 13}$$

```
# evaluate y and y' at 3
y.subs({x:3}), yp.subs({x:3})    # use this to validate our AD answer
```

```
(-9/10, -69/100)
```

By AD: rules are:

$$\langle f, f' \rangle \pm \langle g, g' \rangle = \langle f \pm g, f' \pm g' \rangle$$

$$\langle f, f' \rangle * \langle g, g' \rangle = \langle fg, f'g + fg' \rangle$$

$$\langle f, f' \rangle / \langle g, g' \rangle = \langle f/g, (f'g - g'f)/g^2 \rangle$$

$x = \langle 3, 1 \rangle$

$x^2 =$

Now let's implement this Differentiation Arithmetic in code ...

Implementation of a differentiation arithemetic

with a gentle introduction to object-oriented programming in Python

```
import numpy as np

class ad:

    def __init__(self,val,der=0):   # allows c = ad(3)
        self.val = val
        self.der = der

    def __repr__(self):
        return f'< {self.val}, {self.der} >'

    def __add__(self,other):
        return ad( self.val + other.val  , self.der + other.der   )

    def __sub__(self,other):
        return ad( self.val - other.val  , self.der - other.der   )

    def __mul__(self,other):
        return ad( self.val * other.val  , self.der*other.val + self.val*other.der    )

    def __truediv__(self,other):
        return ad( self.val / other.val  , ( self.der*other.val - self.val*other.der )/ other.val**2  )

    def sin(self):
        return ad( np.sin( self.val ), np.cos(self.val)*self.der )

# to allow use of syntax sin(f)
def sin(x):
    if isinstance(x,ad):
        return x.sin()
    else:
        return np.sin(x)

# what we want to be able to do:

f = ad(3.4,-1.3)
g = ad(7,-2)
c = ad(3)
display(f)
display(g)
display(c)

f + g
print('product f*g')
display( f*g )
```

Check against symbolic differentiation

```
x = ad(3,1)
x*x
```

```
< 9, 6 >
```

```
x/x
```

```
< 1.0, 0.0 >
```

```
x + x
```

```
< 6, 2 >
```

```
x - x
```

```
< 0, 0 >
```

```
seven = ad(7,0)
seven*x
```

```
< 21, 7 >
```

```
z = ad(np.pi/2)
sin(z)
```

```
< 1.0, 0.0 >
```

```
x = sp.symbols('x')
y = 3*x + sp.sin(7*x**2/(1+x))
y
```

$$3x + \sin\left(\frac{7x^2}{x+1}\right)$$

```
yprime = sp.diff(y,x)
yprime
```

$$\left(-\frac{7x^2}{(x+1)^2} + \frac{14x}{x+1}\right)\cos\left(\frac{7x^2}{x+1}\right) + 3$$

```
y.subs({x:2.}),yprime.subs({x:2.})
```

```
(6.09131723555475, -3.19622486179360)
```

symbolic

```
x = ad(2.,1.)
three = ad(3,0)
seven = ad(7,0)
one   = ad(1,0)
y = three*x + sin(seven*x*x/(one+x))y = 3*x + sp.sin(7*x**2/(1+x))

y
```

agree

```
< 6.091317235554749, -3.1962248617936035 >
```

AD

Extending the DA idea:

It's straightforward to extend the idea to
(i) higher-order AD
(ii) AD in multiple variables

(i) < f, f', f'' > * < g, g', g'' > =

$$\langle fg,\ f'g+fg',\ f''g+2f'g'+fg'' \rangle$$

2nd der: $(f'g+fg')' = f''g+f'g'+f'g'+fg''$

x = < 3, 1, 0 >

$x^2 = xx = \langle 9,\ 1\cdot 3+3\cdot 1,\ 0\cdot 3+1\cdot 1+1\cdot 1+3\cdot 0 \rangle$

$= \langle 9, 6, 2 \rangle$

Symbolically $y = x^2$, $y' = 2x$, $y'' = 2$

x = 3
9
6
2

(ii) Suppose $f: \mathbb{R}^2 \to \mathbb{R}$

Represent f by $\langle f, \partial_1 f, \partial_2 f \rangle$

And the product rule would be

$$\langle f, \partial_1 f, \partial_2 f \rangle * \langle g, \partial_1 g, \partial_2 g \rangle$$

$$= \langle fg,\ (\partial_1 f)g + f(\partial_1 g),\ (\partial_2 f)g + f(\partial_2 g) \rangle$$

And for $f: \mathbb{R}^n \to \mathbb{R}$

represent f by $\langle f, \nabla f \rangle$

with product rule $\langle f, \nabla f \rangle * \langle g, \nabla g \rangle = \langle fg,\ \nabla f \cdot g + f \cdot \nabla g \rangle$

Machine learning, reverse-mode AD aka back-propagation

.

The above process is called "forward-mode" AD.
It is accurate to machine epsilon, and easy to program.
But it is not optimally efficient.

Machine learning - training a neural network - involves minimizing a real-valued function called the "loss", which is a measure of how poorly the network is performing the desired task.

The network is characterized by a large set of parameters called "weights",

$$w_0, w_1, w_2, \ldots, w_{1000000000} .$$

loss $y = N(\vec{w})$. Want to minimize y with respect to $\vec{w}$.
($\vec{w}$: large vector of weights)

The standard method is to move repeatedly in the direction of steepest descent. gradient descent

That direction is $-\nabla_w N$

Because of the huge dimension of $\vec{w}$, we need to be as efficient as possible (as well as accurate).

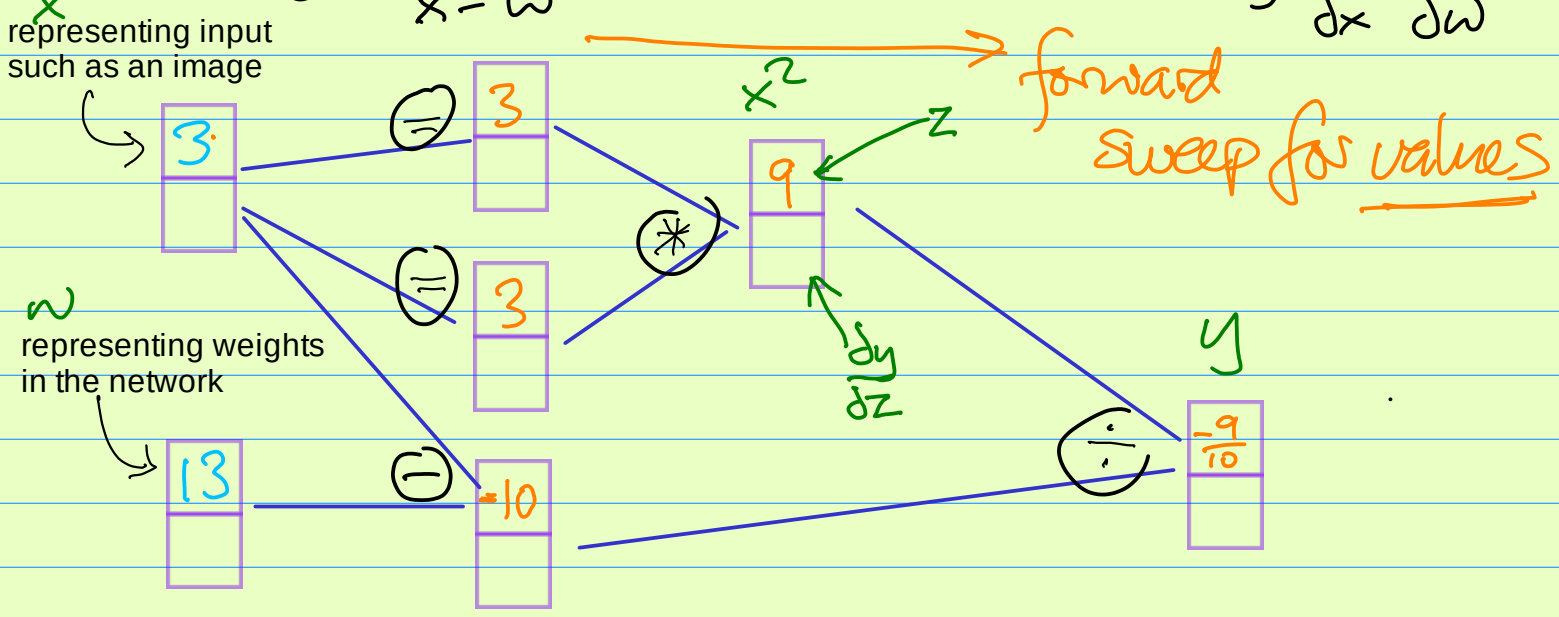
"Reverse-mode AD" or "back-propagation" is more efficient than forward-mode AD.

In the diagram below,
the upper slots are for the *values* of initial, intermediate, and final quantities in the calculation,
the lower slots are for the *derivative of the output value y with respect to the quantity*.

Our goal is to fill in all the slots in two sweeps:
(i) a forward sweep to fill in all the *values*
(ii) a reverse sweep (back propagation) to fill in all the partial derivatives,

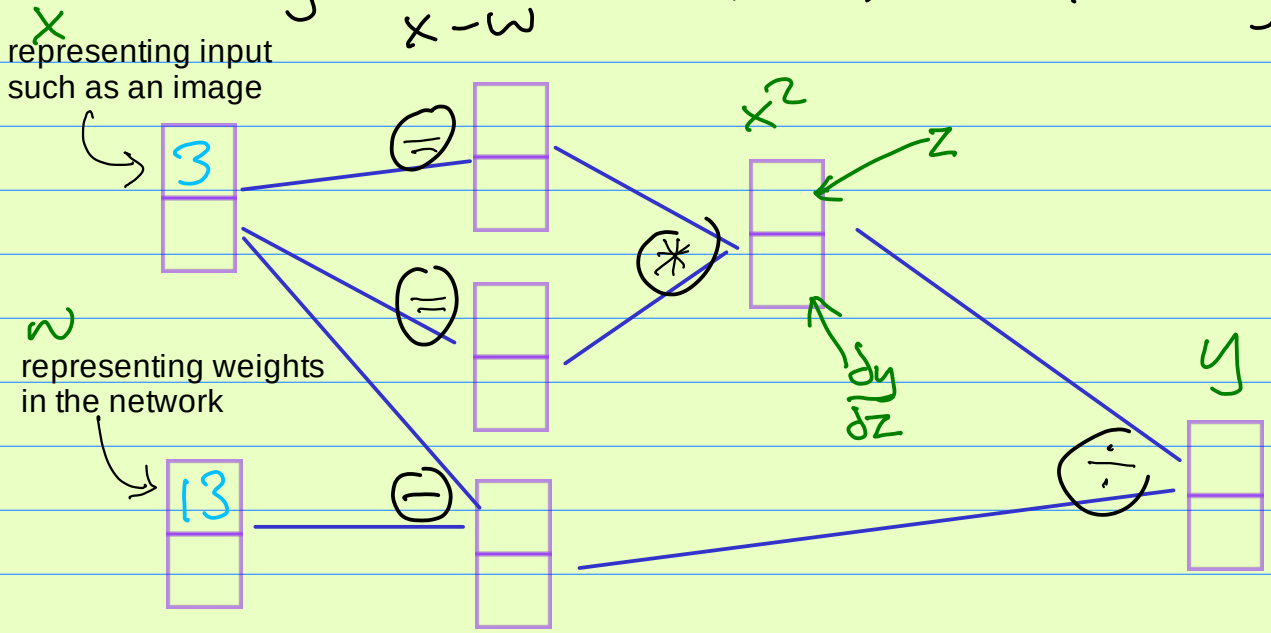
The last numbers we compute in the reverse sweep will be the desired components of the gradient of y.

Example: $y = \dfrac{x^2}{x - w}$ at $x = 3, w = 13$. Seek $y, \dfrac{dy}{dx}, \dfrac{dy}{dw}$.



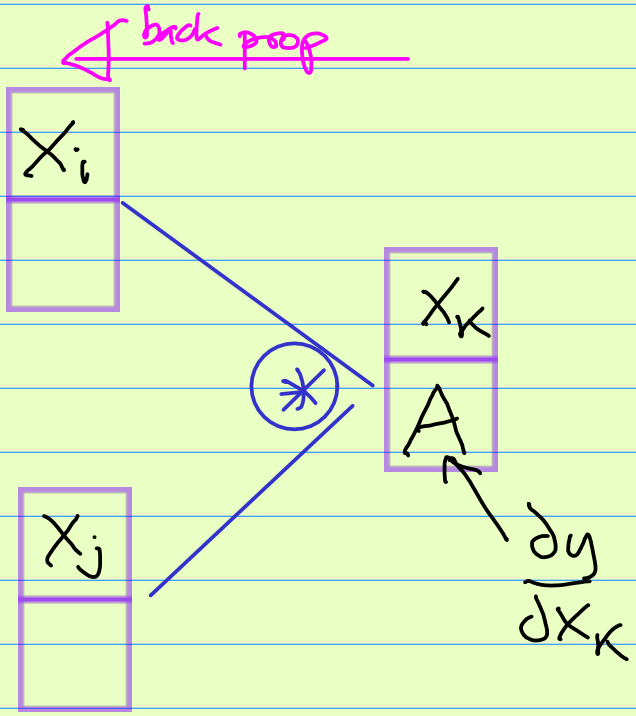
Forward sweep to fill in the values:

$y = \frac{x^2}{x - w}$ at $x = 3, w = 13$. Seek $y, \frac{dy}{dx}, \frac{dy}{dw}$.



For the reverse sweep to fill in the partial derivatives of y, we need to develop some rules ...

Let's look at multiplication.



Chain rule is

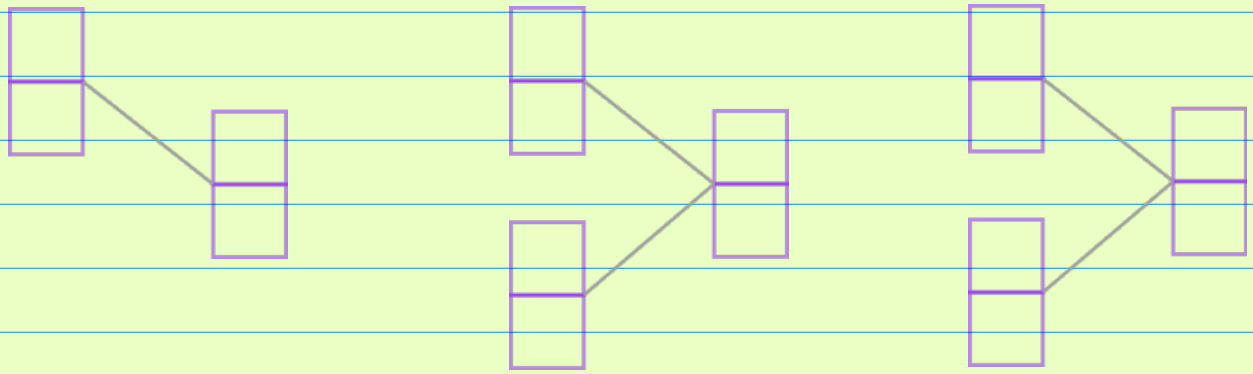
$$\frac{dy}{dx_i} = \frac{dy}{dx_k} \frac{dx_k}{dx_i}$$

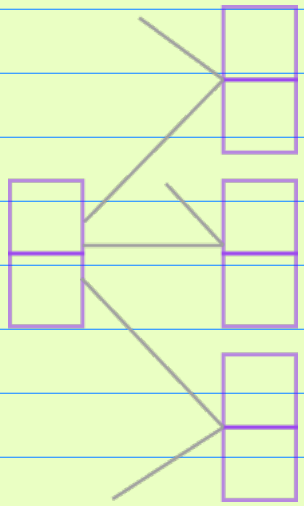
$$=$$

$$=$$

Exercise for you:

Find the back prop rules for the operations of $=, -, \div$ and branching.





Finally let's apply our back prop rules to compute the gradient of our example expression.