Backpropagation

Lesson 2

COMP6252 (Deep Learning Technologies)

ECS, University of Southampton

Computing the gradient

- An essential part of Deep Learning is computing the gradient of some loss function
- In most situations one cannot compute the gradient analytically
- PyTorch computes the gradient using three concepts
  1. Computational Graph
  2. Maintain a list of the derivatives of **primitive operations**
  3. Use the chain rule from calculus

Chain rule

- The chain rule allows us to compute the derivative of a composite function. Consider the following example

$$h = g(f(x))$$

- To compute $\frac{\partial h}{\partial x}$, the chain rule gives

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial g}\frac{\partial g}{\partial f}\frac{\partial f}{\partial x}$$

- The crucial point in the above is that if we know each individual derivative then the result could be computed as the product.

Example

Let $f(x) = x^2$, $g(f) = \log(f)$,$h = g + 1$ with $x = 3$. We have:

$$\frac{\partial f}{\partial x} = 2x = 6$$
$$\frac{\partial g}{\partial f} = \frac{1}{f} = \frac{1}{9}$$
$$\frac{\partial h}{\partial g} = 1$$

Therefore

$$\frac{\partial h}{\partial x} = 1 \times \frac{1}{9} \times 6 = \frac{2}{3}$$

An important part of the above is that we know the derivative of **primitive operations/functions**: even the most complicated of expressions can be broken down into a sequence of primitive operations.

How does PyTorch use the chain rule?

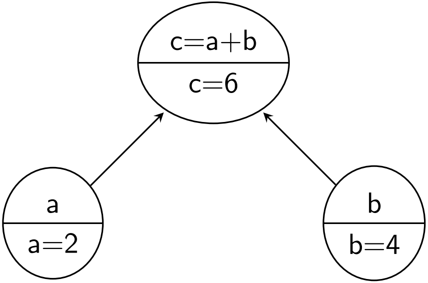
- Now that we know about the chain rule, how does PyTorch use it to compute the gradient?
- It constructs a graph where each primitive operation is represented by a node
- For each such node, it attaches auxillary nodes (actually functions) to compute its derivative
- As an example, let us see how PyTorch computes the following

$$
\begin{aligned}
a &= 2 \\
b &= 4 \\
c &= a + b \\
d &= \log(a) * \log(b) \text{ //not a primitive op. More later} \\
e &= c * d
\end{aligned}
$$

Computational Graph

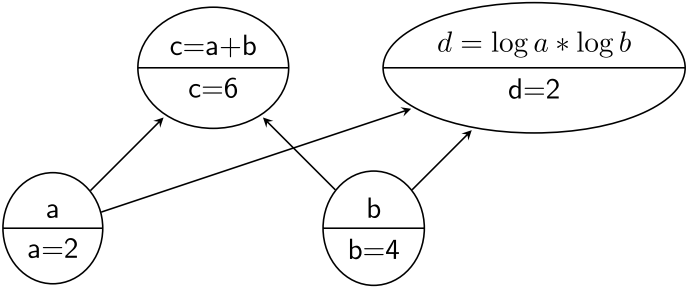
$$a = 2$$
$$b = 4$$

Computational Graph



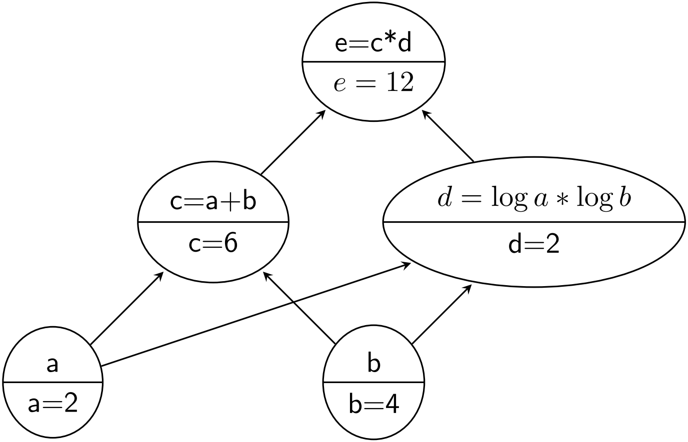
$$a = 2$$
$$b = 4$$
$$c = a + b$$

Computational Graph



$$a = 2$$
$$b = 4$$
$$c = a + b$$
$$d = \log(a) * \log(b)$$

Computational Graph



$a = 2$

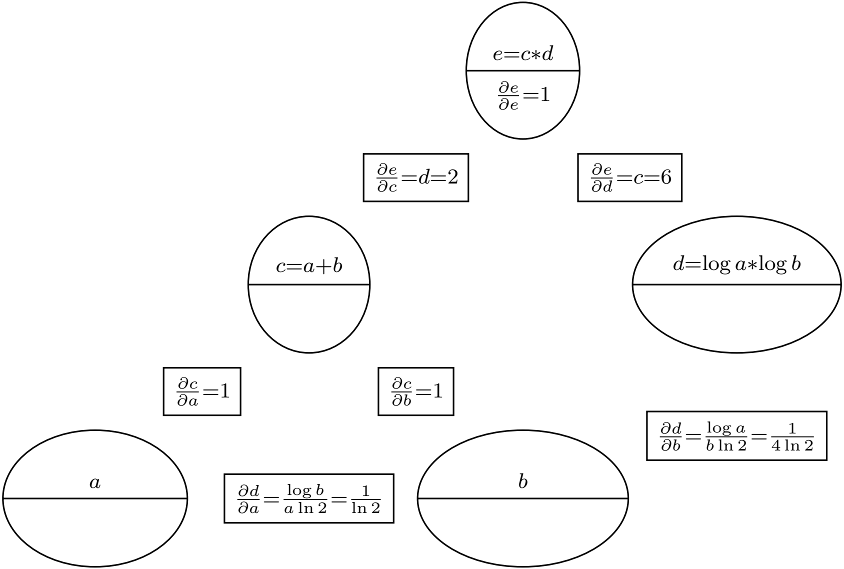
$b = 4$

$c = a + b$

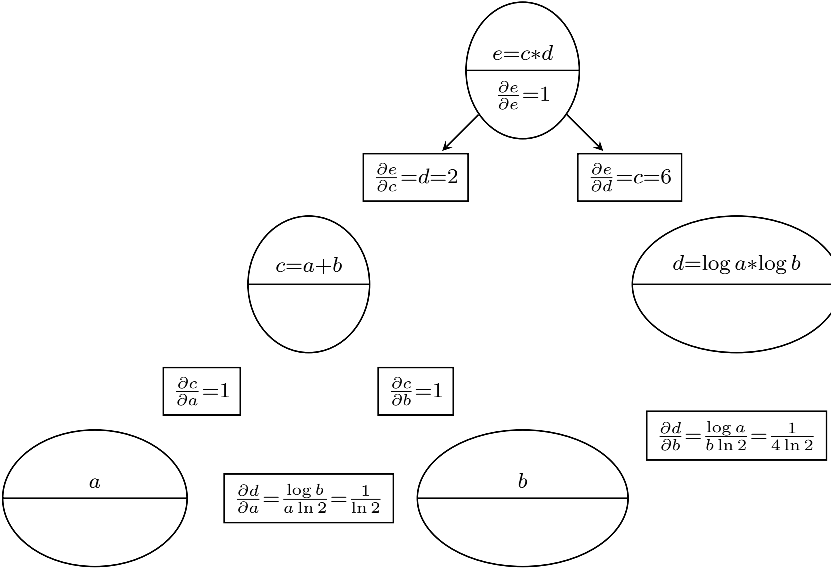
$d = \log(a) * \log(b)$

$e = c * d$

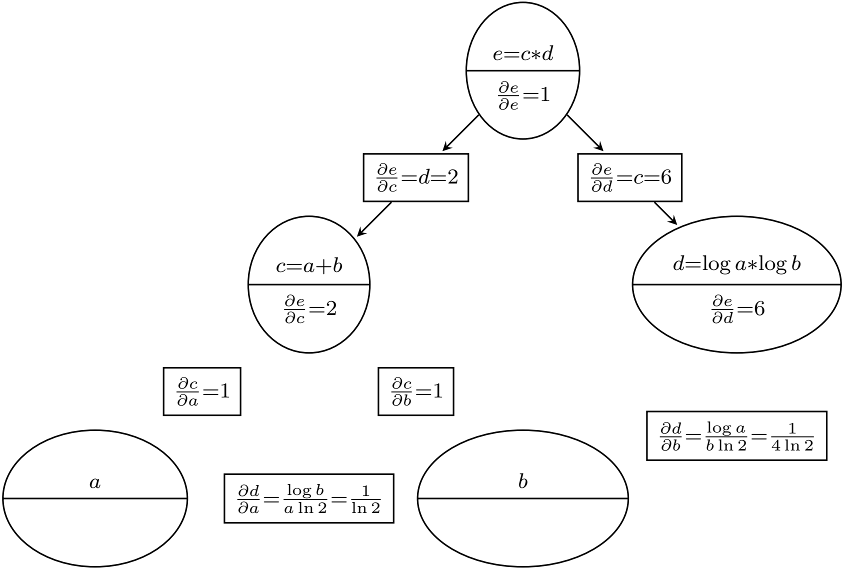
Backward pass



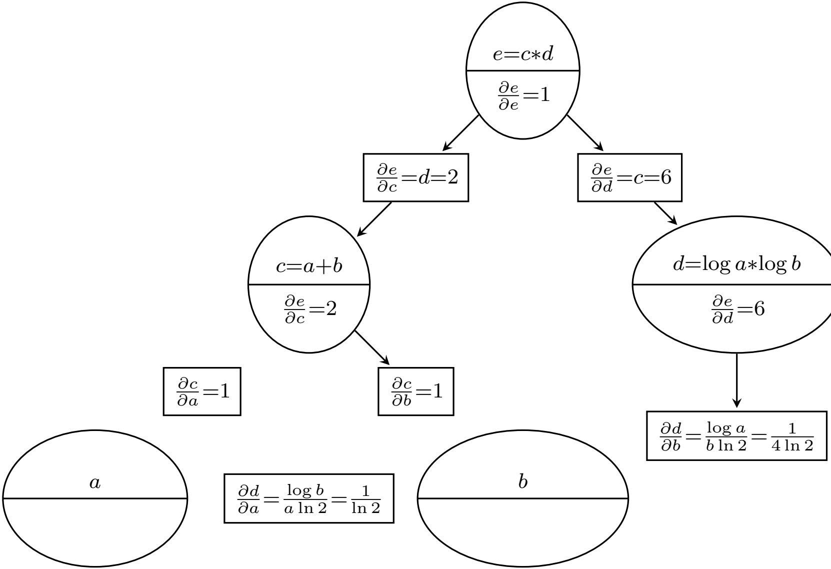
Backward pass



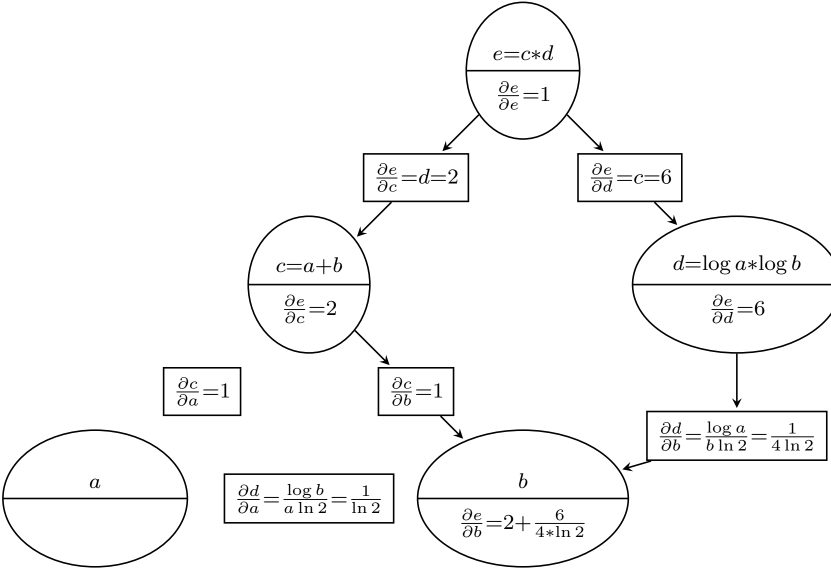
Backward pass



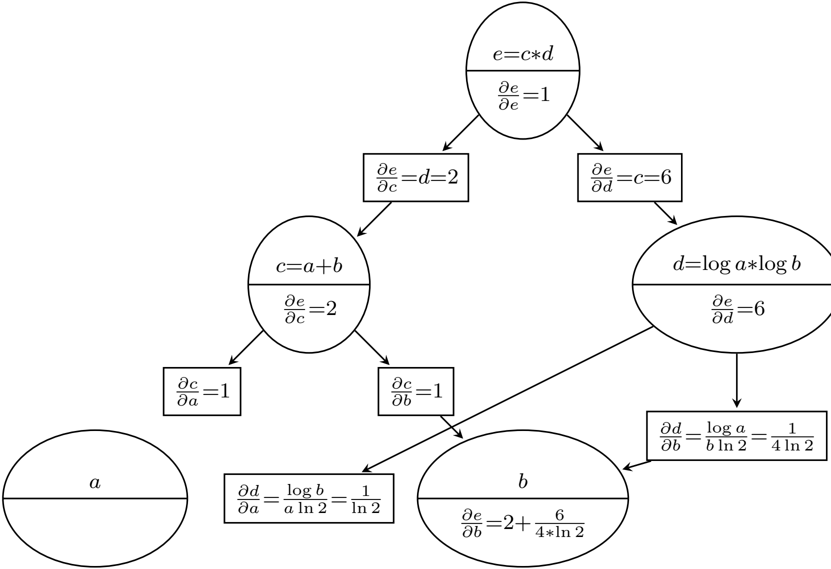
Backward pass



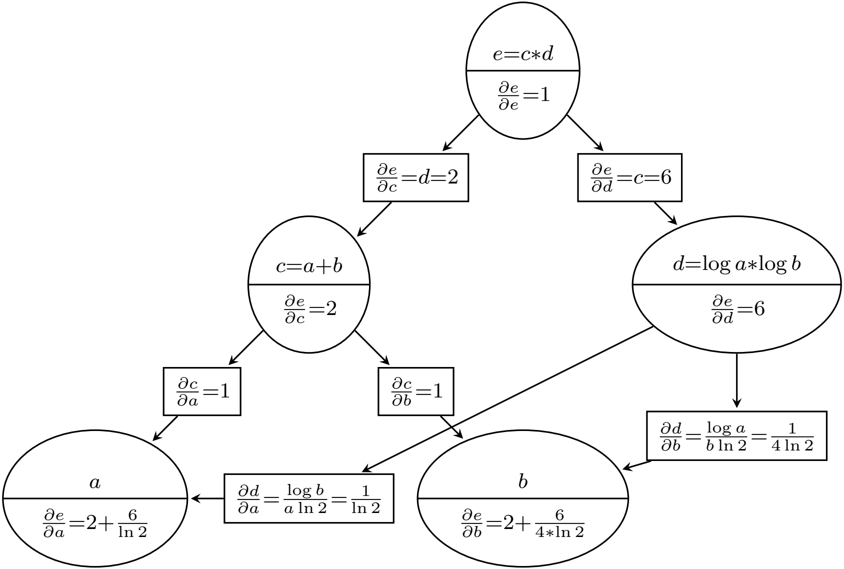
Backward pass



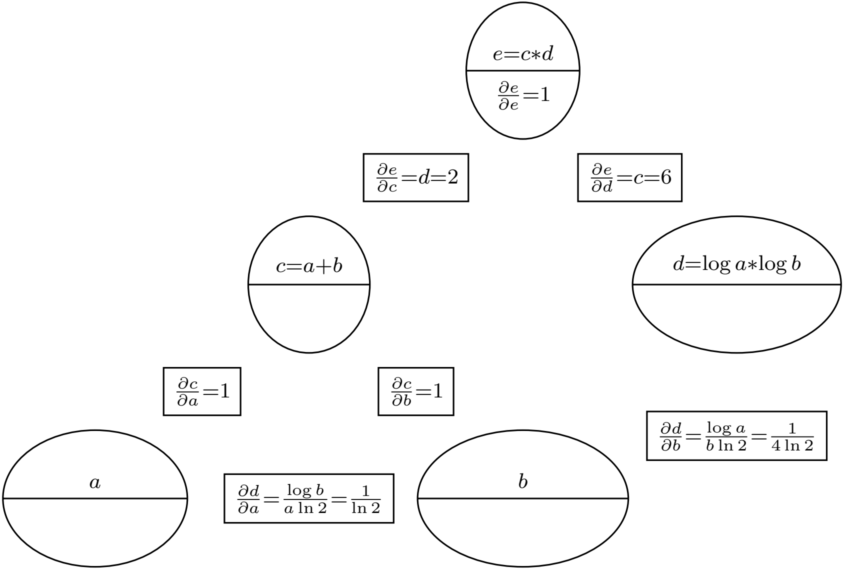
Backward pass



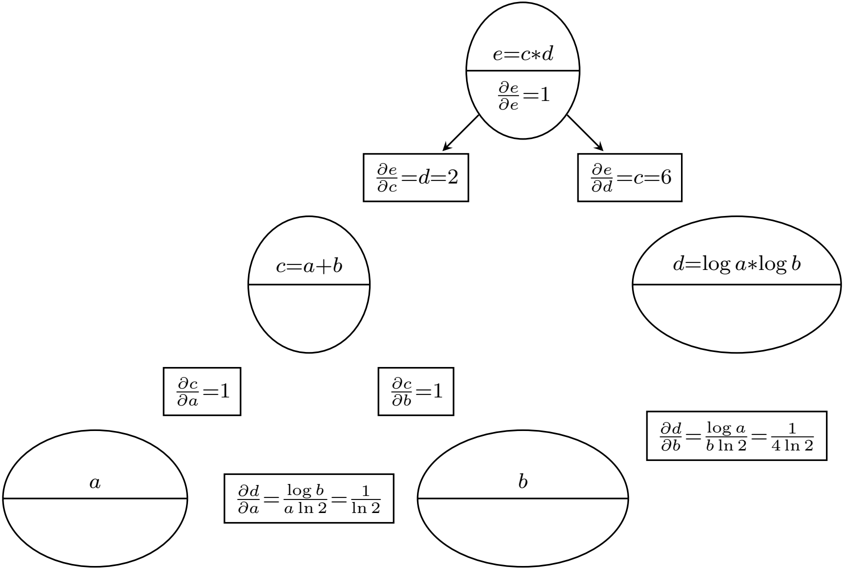
Backward pass



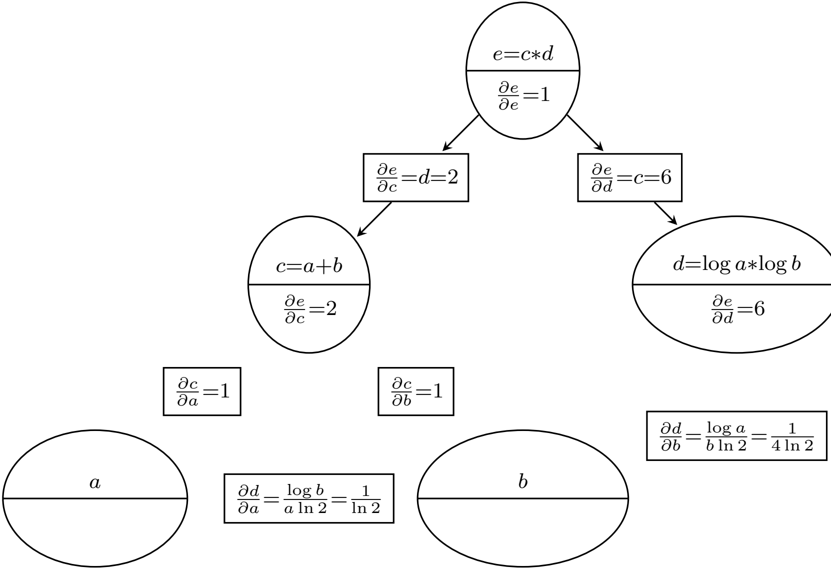
Backward pass



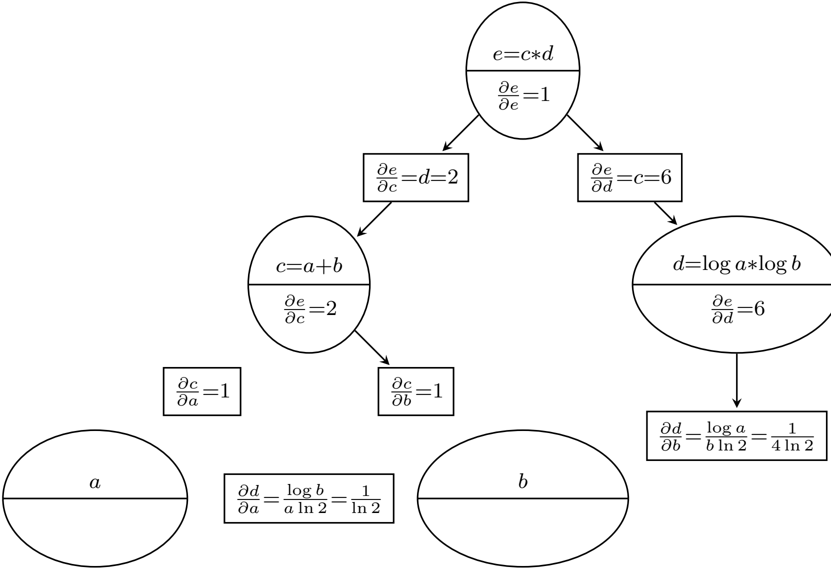
Backward pass



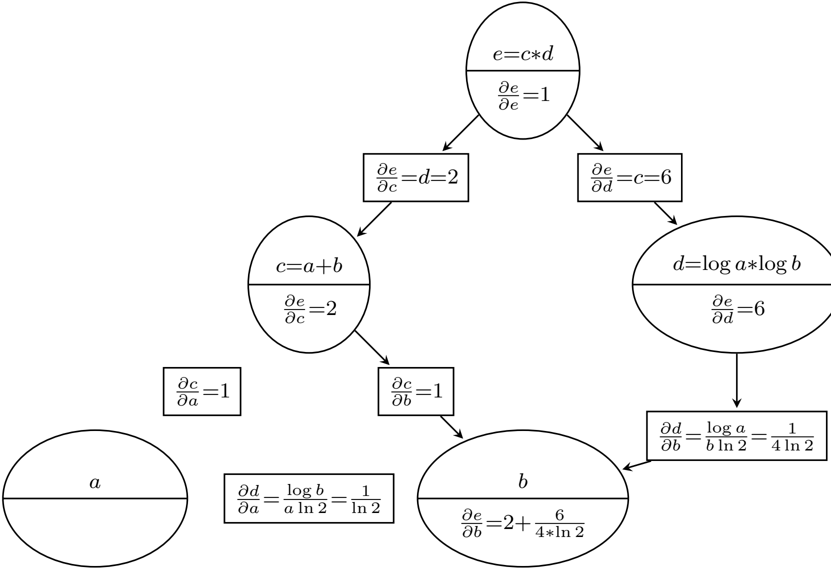
Backward pass



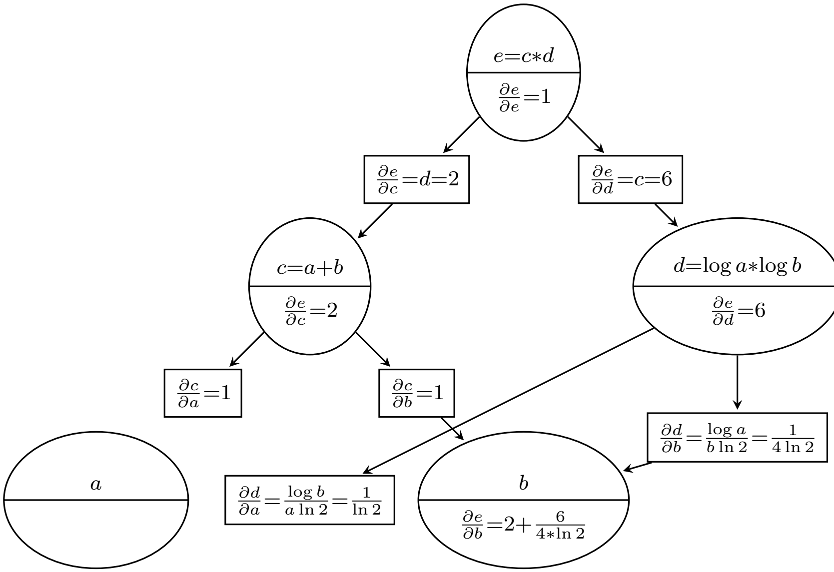
Backward pass



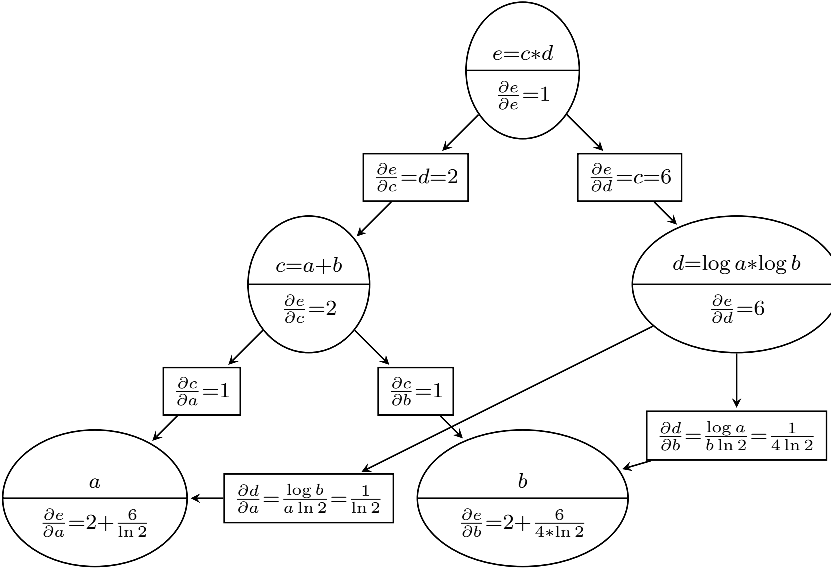
Backward pass



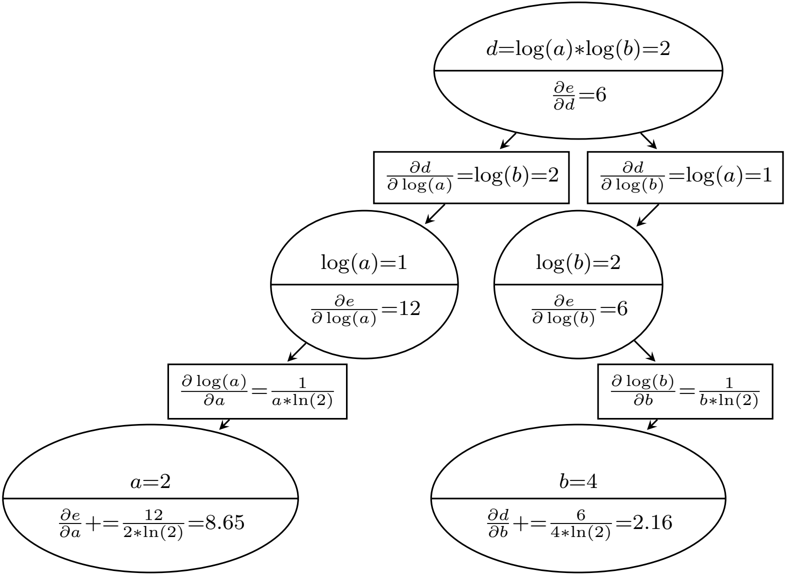
Backward pass



Backward pass



Details for node d



PyTorch Details

- PyTorch implements the above using a graph of gradient functions
- We can map our graph to PyTorch implementation as follows
  1. The oval nodes represent the gradient functions
  2. The rectangular nodes represent the outputs of those functions
- The starting point is `e.grad_fn` with input 1. The output of `e.grad_fn(1.)` is `[2,6]` as expected.
- The output `[2,6]` is used as input to the functions `e.grad_fn.next_functions`
- The code for the whole graph is shown on the corresponding notebook