Backpropagation

Instructor: Yutong Wang

October 4, 2024

This is a tutorial for backpropagation written for CS 577 at Illinois Institute of Technology.

1 Autograd with Scalars

In class, we've manually calculated the gradients for the function $J(\boldsymbol{\theta})$ for several different types of models. We want to automate this process. Let's forget about machine learning for a second and consider a simple function like

$$f(x, y) = \log(1 + e^{-(x+y)}) \cdot (x + y). \tag{1}$$

We will use (1) as our running example. After years of math in grade school, we can compute $f(x, y)$ in a snap by "plug-and-chug". But since we are computer scientists, we are interested in how to *compute* the function *algorithmically*. To do so, we need two ingredients:

- *Quantities* including
  - the *leaf*[1] quantities x and y, but also
  - the *internal quantities* such as $x + y$, $-(x + y)$, $e^{-(x+y)}$ and so on.
- *primitive operations* *(ops)* on quantities including
  - the unary op "`log`" and,
  - the binary addition "`+`" or "`add`" op.

Let's first focus on quantities.

1.1 Quantities = candidates

The quantities x and y are special in the sense that they are not defined in terms of anything else. We call them "leaves" or "leaf quantity". The quantities $x + y$, $-(x + y)$, and so on, do depend on other quantities. Before moving on, we note that...

> ...throughout, we focus on *scalar* quantities. Defining an auto-differentiation library for vector quantities is more involved and we will get into it later.

[1] This terminology is in accordance with that of Pytorch. See `https://pytorch.org/docs/stable/generated/torch.Tensor.is_leaf.html`. They are leaves in the computational graph in the graph-theoretic sense, which will be discussed below.

Quantity = any value or intermediate result

Quantity → Leaf quantities: They are not defined in terms of other quantities. For example x only

Quantity → Internal quantities: These are intermediate values that result from applying operations to other quantities. For example x+y

So it sounds like quantities are just numbers. What's the big deal? The answer is that we are not interested in only computing the quantity $f(x, y)$, but also the partial derivatives $\frac{\partial f}{\partial x}(x, y)$ and $\frac{\partial f}{\partial y}(x, y)$. Moreover, we want to compute these partial derivatives *automatically*. To this end, we need a data structure that supports this:

```
class ag: # AutoGrad
    # lines skipped [...]
    class Scalar: # Scalars with grads
        def __init__(self,
                     value,
                     op="",
                     _backward= lambda : None, # the do-nothing function
                     inputs=[],
                     label=""):

            self.value = float(value)
            self.grad = 0.0

            self._backward = _backward
            self.inputs = inputs

            self.op = op
            self.label = label
```

This `ag.Scalar` is how we represent quantities that enables automatic differentiation, as we will see below. It is a class that wraps around a `value`, but it also contains additional information such as

- `grad` that keeps track of the gradient
- `inputs` that keeps track of the inputs that lead to a particular value
- `op` — the label string for the operation that produced the quantities from the inputs (for visualization only)
- `_backward` — the function for propagating the grad to the inputs
- `label` — the label string for drawing the computational graph (for visualization only)

Next, we go into depth about quantities in of themselves, starting with the leaves.

1.1.1 Leaf quantities

A bookkeepping matter: We could be in a situation where we have *so many* leaf quantities that we run out of alphabets (for instance, for the $J(\boldsymbol{\theta})$ function for a very large neural network, the leaves are all the weights in $\boldsymbol{\theta}$). So let's assign our leaves to the following symbols:

$$z_1, z_2, \ldots, z_\ell.$$

where ℓ is the total number of leaf quantities. In our running example (1), we have $\ell = 2$, $z_1 = x$ and $z_2 = y$. To represent these leaves in code, we write

```
x = ag.Scalar(2.0, label="z1\nleaf(x)") # the label is just for our own benefit
y = ag.Scalar(3.0, label="z2\nleaf(y)")

z1 = x
z2 = y
```

The values 2.0 and 3.0 that are passed in are arbitrary. The label "`z1 leaf(x)`" are chosen to aid visualization. The code for drawing the graph is looking for the "`leaf`" substring for styling purposes.

In practice, x and y will be replaced by weights of some neural network and the values are replaced by the random initialization of those weights. Next, let's talk about the internal quantities $x + y$ and so on.

1.1.2 Internal quantities

Let's refer to the internal quantities by $z_{\ell+1}, \ldots, z_n$.

The last quantity z_n will denote the ultimate[2] quantity: $f(x, y)$ itself. In our running example, we can put $z_3 := z_1 + z_2 = x + y$,

```
z3 = z_1+z_2
```

and $z_4 := -z_3$. But wait, we haven't defined addition between `ag.Scalar` yet. We'll get to that in the next section. Now, let's set some rules. We enforce the following for assigning internal quantities to the symbols $z_{\ell+1}, z_{\ell+2}, \ldots$:

Property 1. For each $j = \ell + 1, \ldots, n$, the quantity z_j only depends on the prior symbols:

$$z_1, z_2, \ldots, z_{j-1}.$$

The notion of "dependency" is exactly made rigorous by "operations".

1.2 Operations

As we said earlier, ops include taking logarithm of a quantity or adding two quantities. Let's rewind back to our earlier symbol assignment $z_3 := z_1 + z_2$. Thinking graphically, we can visualize this as three nodes consisting of z_1, z_2 and z_3 with two edges flowing from z_1, z_2 to z_3:

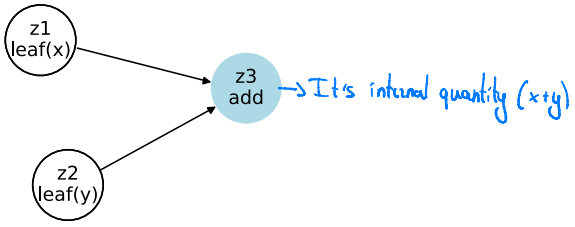


Figure 1: The "add" operation.

```
        def __add__(self, other):
            if not is instance(other, ag.Scalar):
                other = ag.Scalar(other, label=f"{other}\nconst")
        z3 --> output = ag.Scalar(self.value + other.value,
                                  inputs=[self, other], op="add")
            # lines skipped [...]
            return output
```

[2] I mean "ultimate" in the sense that the quantity is "final".

Note, we haven't gotten to `_backward` yet. This is a good time to link our cartoon representation Figure 1 to the jargons that we've introduced:

- Leaf quantities are denoted by white disks with black border.
- Internal quantities are denoted by blue-filled disks.
- The directed edges goes from inputs to the output `ag.Scalar` quantity (in this case z_3)
- The operation is labeled on the output disk.

Let's complete our forward pass.

1.2.1 The forward pass and the computational graph

Continuing to obey Property 1, we assign the remaining internal quantities:

$$z_4 := -(z_3) = -(x + y)$$

This operation is simply the negation of z_3. We can think of this as applying a unary operation "`neg`" on z_3.

```
        def __neg__(self):
            output = ag.Scalar(-self.value, inputs=[self], op="neg")

            # lines skipped [...]

            return output
```

Continuing, we have

1. $z_5 := e^{z_4} = e^{-(x+y)}$ ("`exp`")
2. $z_6 := z_5 + 1 = e^{-(x+y)} + 1$ ("`add`")
3. $z_7 := \log(z_6) = \log(1 + e^{-(x+y)})$ ("`log`")
4. $z_8 := z_7 \cdot z_3 = \log(1 + e^{-(x+y)}) \cdot (x + y)$ ("`mul`")

We won't show the (foward portion) code for the other ops since they are similar to "`add`" for binary ops and to "`neg`" for unary op.

Thus, $z_n = z_8$ is the ultimate quantity, i.e., $f(x, y)$. At this point, we have completed the *forward pass*, i.e., an evaluation of the function $f(x, y)$. As a by-product, we have created a sequence of quantities $z_1, z_2, \ldots, z_8$, even though we are only interested in z_8. These quantities are nodes in a graph often called the *computational graph* of a function Figure 2. The edges of this directed acylic graph captures the relationship between the quantities.

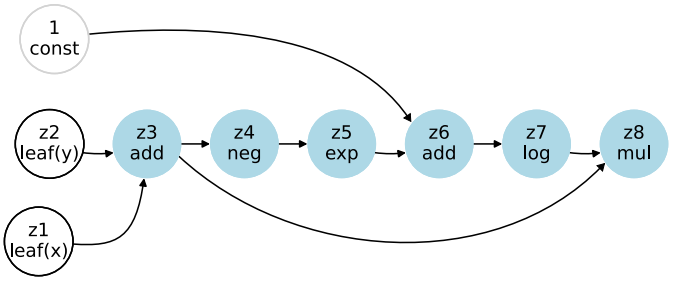


Figure 2: The computational graph of our running example.

The forward phase code is simply

```
z1 = x
z2 = y
z3 = z1+z2
z4 = -z3
z5 = ag.exp(z4)
z6 = z_5+1
z7 = ag.log(z6)
z8 = z7*z3
```

where

```
class ag: # AutoGrad
    # lines skipped [...]
    def log(input):
        output = ag.Scalar(math.log(input.value), inputs=[input], op="log")
        # lines skipped [...]
        return output

    def exp(input):
        output = ag.Scalar(math.exp(input.value), inputs=[input], op="exp")
        # lines skipped [...]
        return output
```

Note, we also have this "`const`" plate. It is convenient to just add a number without wrapping it in `ag.Scalar` first. This was used to define z_6.

So there you have it: the **computational graph**. Property 1 essentially forces the computational graph to be a directed acyclic graph, i.e., there are no (directed) loops. Some terminologies:

- **Sink** — all the edges eventually lead to z_8. If we think of these edges as "flow", then we can call z_8 the sink.
- **Upstream** — since we view the edges as flow, we can say "going upstream from z_6, we can reach z_4", or "z_4 is upstream of z_6".
- **Downstream** — the same as above, but backwards.

Recall that the nodes are "enriched" with additional information such as `grad`, and `_backward` which, so far, we have not touched upon. This is the subject of the next section.

1.3 Backpropagation

Backpropagation is an algorithm for computing all the partial derivatives

$$\frac{\partial f}{\partial z_j}(x, y) \tag{2}$$

with respect to each $j = 1, \ldots, \ell, \ell + 1, \ldots, n$ starting with $j = n$ and traversing backward the depth-first-search of the computational graph:

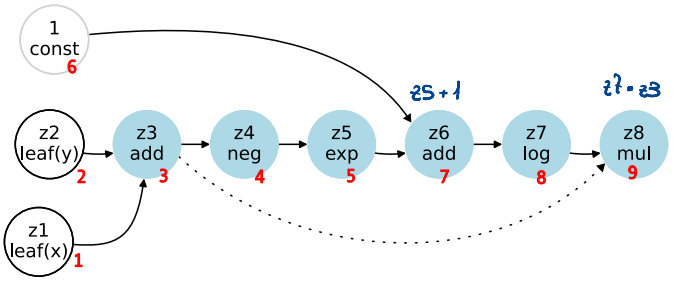


Figure 3: Depth-first-search order for traversing the z_is

The partial derivatives $\frac{\partial f}{\partial z_1}(x,y), \ldots, \frac{\partial f}{\partial z_8}(x,y)$ are what `z1.grad`, ..., `z8.grad` are meant to hold. In the initialization of an `ag.Scalar`, the grad is zero, i.e., `self.grad = 0.0`. This is obviously incorrect. The job of *backpropagation* (BP) is to populate these `ag.Scalar.grad` with the correct partial derivative, while traversing the computational graph via reversed DFS.

Backpropagation is kickstarted in our code by

```
z8.backward()
```

where `backward` is a method of the `ag.Scalar` class:

```
    class Scalar: # Scalars with grads
        # lines skipped [...]
        def backward(self):
            self.grad = 1.0

            topo_order = self.topological_sort()

            for node in reversed(topo_order):
                node._backward()

        def topological_sort(self):
            topo_order = []
            visited = set()

            def dfs(node):
                if node not in visited:
                    visited.add(node)
                    for input in node.inputs:
                        dfs(input)
                    topo_order.append(node)

            dfs(self)
            return topo_order
```

If we print out the

```
z8.topological_sort()
```

we get

```
['z1:leaf(x)',
 'z2:leaf(y)',
 'z3:add',
 'z4:neg',
 'z5:exp',
 '1:const',
 'z6:add',
 'z7:log',
 'z8:mul']
```

We haven't discussed `_backward` yet. Let's hold that thought. First, what is the *mathematical* intuition behind the partial derivative Equation (2)?

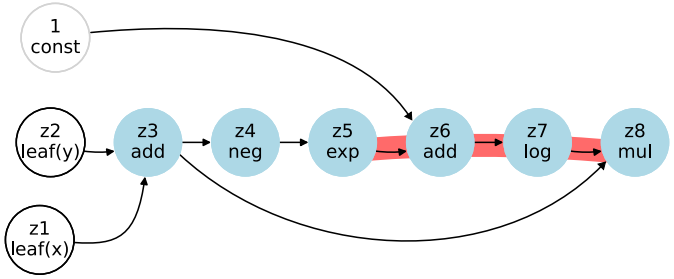
1.3.1 Intuition for partial derivative

$\frac{\partial f}{\partial z}$ = how much changes $f(x,y)$ if z changes a tiny amount

The partial derivative intuitively says this:

> If we change z_j by a tiny amount (let's write it as ∂z_j), how much does $f(x, y)$ change (let's write it as $\partial f(x, y)$)? In other words, it is how much z_j "influence" the value at the sink.

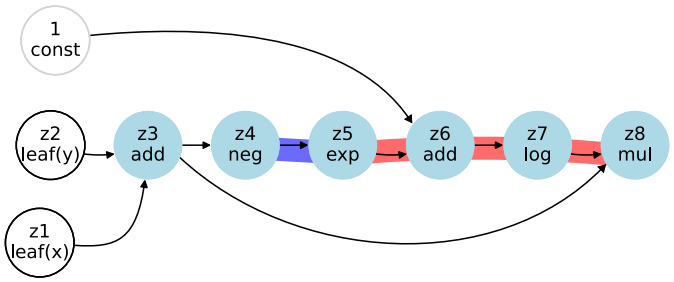
Since $f(x, y) = z_8$, we can visualize this "influence" $\boxed{\frac{\partial f}{\partial z_5}(x, y)}$ as a (red) path from z_5 to z_8:



Idea: The influence (gradient) z_5 on z_8 is tracked through the operations on the computation graph

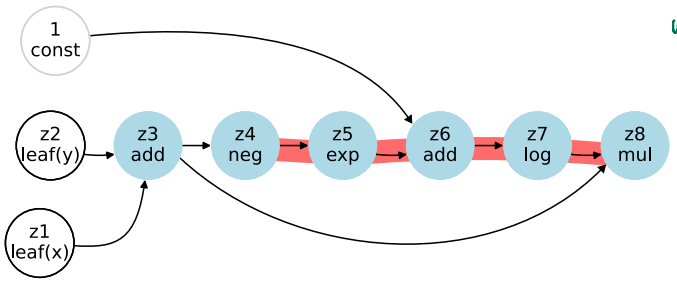
In a similar vein, we can draw $\boxed{\frac{\partial z_5}{\partial z_4}}$ as the short (blue) path from z_4 to z_5:

Idea: z_5 depends directly on z_4

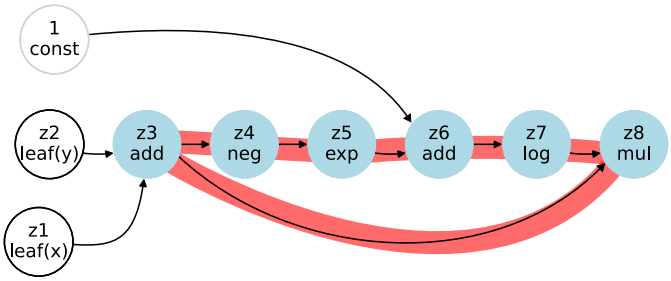


Chain rule says that we can glue these together $\boxed{\frac{\partial z_5}{\partial z_4}\frac{\partial f}{\partial z_5}(x, y) = \frac{\partial f}{\partial z_4}(x, y)}$

Idea: The chain rule allows us to propagate gradients through the graph



Chain rule can help us extend our calculation of the partial derivative upstream to the leaves. But it's also possible for this influence to be over several different paths:



So it's important to keep track of the different "streams of influence" and add them together. In the next section, we make this intuition rigorous.

1.3.2 Initializing backpropagation

Calculating `z8.grad` is the easy case. This is because, z_8 is *equal* to $f(x, y)$. If we change $z_8 = f(x, y)$ by a tiny amount, then $f(x, y)$ will change by that same amount. This is a *tautology*. Thus, $\frac{\partial f}{\partial z_8}(x, y) = 1$ always and so

key idea

```
        def backward(self):
            self.grad = 1.0
            # lines skipped [...]
```

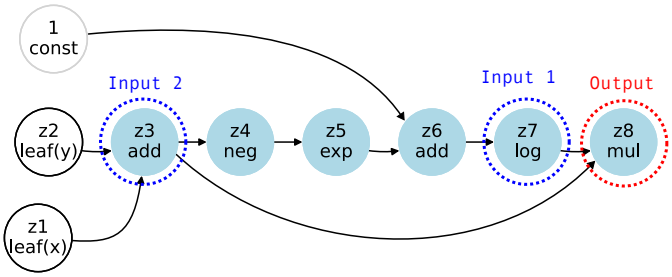
← At the begining

And just like that, we've already ensured that the `z8.grad` is set correctly. We now enter...

1.3.3 The main loop of backpropagation: beginning of the first step

```
            for node in reversed(topo_order):
                node._backward()
```

In the first iteration, `node = z8`, where the operation is `mul`.



The reader may recall that we have not defined `_backward` for any of the operations introduced thus far. Now is a good time to discuss the backward function for multiplication. Note that by construction:

$$z_8 = z_7 * z_3.$$

How can we calculate $\boxed{\frac{\partial f}{\partial z_7}(x, y)}$ given the correct value of $\frac{\partial f}{\partial z_8}(x, y)$ (which is equal to 1 as we saw)? By the chain rule

$$\frac{\partial f}{\partial z_7}(x, y) = \frac{\partial f}{\partial z_8}(x, y)\frac{\partial z_8}{\partial z_7} = \frac{\partial f}{\partial z_8}(x, y)\frac{\partial (z_7 * z_3)}{\partial z_7} = \frac{\partial f}{\partial z_8}(x, y)\left(\frac{\partial z_7}{\partial z_7}\right) z_3 = \frac{\partial f}{\partial z_8}(x, y) z_3.$$

1

But wait, shouldn't it be

$$\frac{\partial(z_7 * z_3)}{\partial z_7} = \frac{\partial z_7}{\partial z_7} z_3 + z_7 \frac{\partial z_3}{\partial z_7}$$

by the product rule? Yes! But

$$\frac{\partial z_3}{\partial z_7} = 0$$

by Property 1: z_3 does not depend on z_7!

Next, how can we calculate $\boxed{\frac{\partial f}{\partial z_3}(x, y)}$ given $\frac{\partial f}{\partial z_8}(x, y)$? Note that

$$\frac{\partial f}{\partial z_3}(x, y) = \frac{\partial f}{\partial z_8}(x, y) \frac{\partial z_8}{\partial z_3} = \frac{\partial f}{\partial z_8}(x, y) \frac{\partial (z_7 * z_3)}{\partial z_3} = \underbrace{z_7 \frac{\partial f}{\partial z_8}(x, y)}_{\text{term 1}} + \underbrace{\frac{\partial f}{\partial z_8}(x, y) \frac{\partial z_7}{\partial z_3}}_{\text{term 2}}.$$

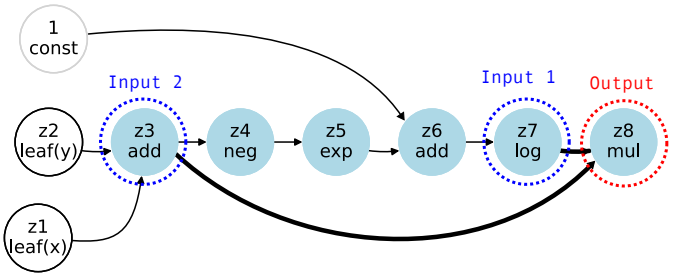
There is an important difference: $\boxed{\frac{\partial z_7}{\partial z_3} \neq 0}$. We can calculate term 1 right now, but term 2 will have to wait until `z4` is reached in the reverse DFS, at which point `z3.grad` will be correctly computed. In the code, `self` is `z7` (since it is the left multiplicand) and `other` is `z3` (the right multiplicand). So for now, we store only $z_7 \frac{\partial f}{\partial z_8}(x, y)$ into `z3.grad`. This directly leads to:

```
        def __mul__(self, other):
            # lines skipped [...]
            output = ag.Scalar(self.value * other.value,
                               inputs=[self, other], op="mul")

            def _backward():
                self.grad += other.value * output.grad
                other.grad += self.value * output.grad # term 1 only!
                return None

            output._backward = _backward

            return output
```

A natural question at this point is why "`+=`", i.e., accumulated sum? Why not just "`=`"? The answer is precisely to deal with the case where a node has multiple outgoing edge like z_3.

1.3.4 The main loop of backpropagation: end of the first step

In conclusion, $\frac{\partial f}{\partial z_7}(x, y)$ is now correctly computed, while only one summand of $\frac{\partial f}{\partial z_3}(x, y)$ has been computed. We will use thickened edges to show our progress in computing the partial derivative. After our first step, the situation is as follows:



Once all edges leaving a node is thickened, the partial derivative at that node is correctly computed. Thus, `z3.grad` is not yet correctly computed since the $z_3 \to z_4$ edge is not thickened.

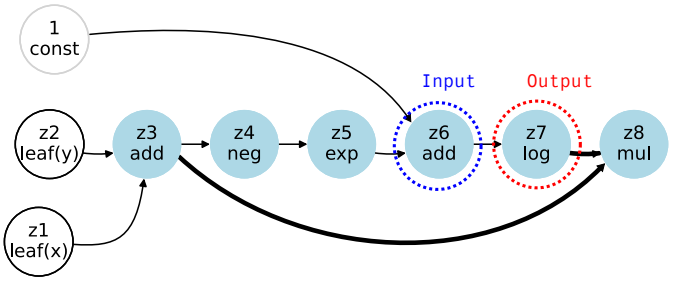
The key is that in the next iteration, `node = z7` and `z7.grad` is now correctly computed.

1.3.5 The main loop of backpropagation: beginning of $z_7 \to z_6$

Suppose now that we are at some internal node whose `grad` has been correctly computed. For simplicity, let's just say `node = z7` in our running example. Indeed, from the previous step we know that

$$\frac{\partial f}{\partial z_7}(x, y)$$

has already been calculated correctly and is now stored in `z7.grad`.



How can we calculate $\frac{\partial f}{\partial z_6}(x, y)$ given correct value of $\frac{\partial f}{\partial z_7}(x, y)$? First, by the chain rule:

$$\frac{\partial f}{\partial z_6}(x, y) = \frac{\partial f}{\partial z_7}(x, y)\frac{\partial z_7}{\partial z_6}.$$

Now, since $z_7 = \log(z_6)$, we have

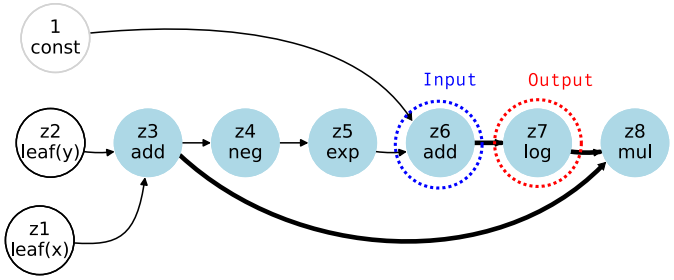
$$\frac{\partial z_7}{\partial z_6} = \frac{1}{z_6} \quad \text{which implies} \quad \frac{\partial f}{\partial z_6}(x, y) = \frac{1}{z_6}\frac{\partial f}{\partial z_7}(x, y).$$

Now, since $\frac{\partial f}{\partial z_7}(x, y)$ is stored in `z7.grad`, `z7` is `output`, and `input` is `z6`. The value z_6 is represented by `input.value`. Thus, "`output.grad / input.value`" = $\frac{1}{z_6}\frac{\partial f}{\partial z_7}(x, y)$., and we have the following code for `ag.log`:

```
    def log(input):
        output = ag.Scalar(math.log(input.value), inputs=[input], op="log")

        def _backward():
            input.grad += output.grad / input.value
            return None

        output._backward = _backward
        return output
```

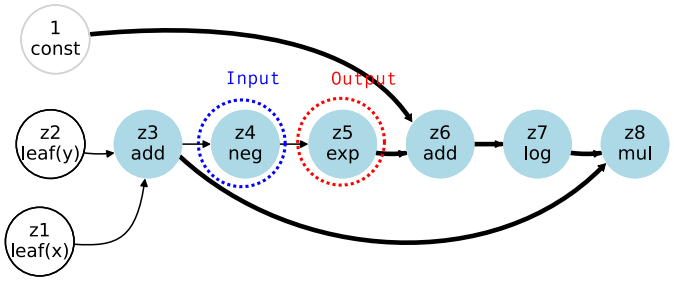
1.3.6 The main loop of backpropagation: end of $z_7 \to z_6$

At this point, we can also thicken the edge:



Let's skip a few iterates until we get to...

1.3.7 The main loop of backpropagation: $z_5 \to z_4$



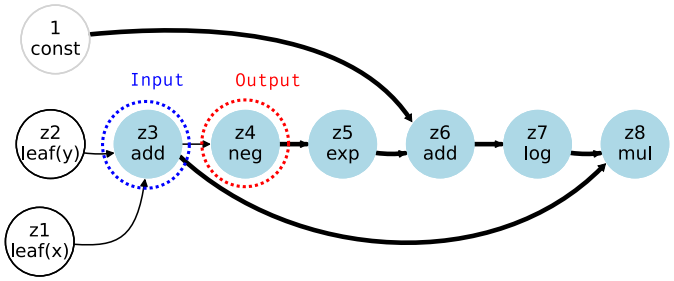
How can we calculate $\frac{\partial f}{\partial z_4}(x, y)$ given correct value of $\frac{\partial f}{\partial z_5}(x, y)$? (Hint: see Section 1.3.5 for example).

```
    def exp(input):

        output = ag.Scalar(math.exp(input.value), inputs=[input], op="exp")

        def _backward():
            input.grad += output.grad * output.value
            return None

        output._backward = _backward
        return output
```

$$\frac{\partial f}{\partial z_4} = \frac{\partial f}{\partial z_5}(x,y) \cdot \frac{\partial z_5}{\partial z_4}$$

1.3.8 The main loop of backpropagation: $z_4 \to z_3$

Continuing in the loop, we will eventually reach `node = z4`:



- What is currently stored in `z3.grad` right before `z4._backward()` is called? (Hint: see Section 1.3.3).

After `z4._backward()` is called, the partial derivative $\frac{\partial f}{\partial z_3}(x, y)$ is now correctly computed.

2 Lists of `ag.Scalar`

How about vectors? We can use lists of `ag.Scalar`!

```
d = 3

x = [ag.Scalar(i, label=f"z{i}\nleaf(x[{i}])")  for i in range(d)]
y = [ag.Scalar(i, label=f"z{i+5}\nleaf(y[{i}])")  for i in range(d)]
final_output = x[0]*y[0]
for i in range(1,d):
    final_output += x[i]*y[i]
```