A common trait of all computers is that they solve complex problems by executing trillions or more simple operations every second. To understand what computers can and cannot do it is useful to have an idealized model of a device that can perform arbitrary calculations by iterating many small simple steps. Circuits are one such model.

1 Circuits

To be concrete suppose you are given five numbers x_1, x_2, x_3, x_4, x_5 and you want to calculate the sum of the products of all distinct pairs of inputs, namely the value of the expression

$$y(x_1, x_2, x_3, x_4, x_5) = x_1x_2 + x_1x_3 + x_1x_4 + x_1x_5 + x_2x_3 + x_2x_4 + x_2x_5 + x_3x_4 + x_3x_5 + x_4x_5. \quad (1)$$

In any given step you are allowed to do one of two operations: add two numbers or multiply two numbers. How should you go about calculating y?

The formula (1) gives you one way of calculating y: Calculate the pairwise products one by one and keep adding the resulting terms one by one. We can represent this procedure by the following DAG:

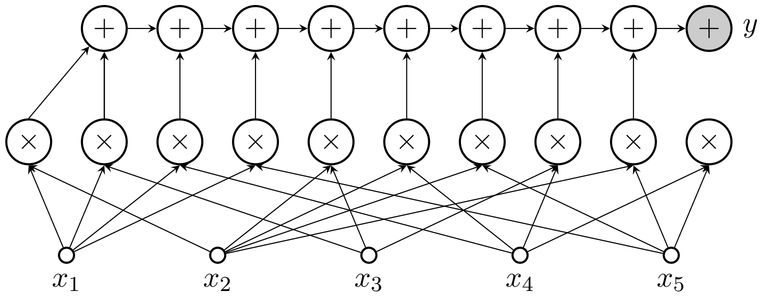
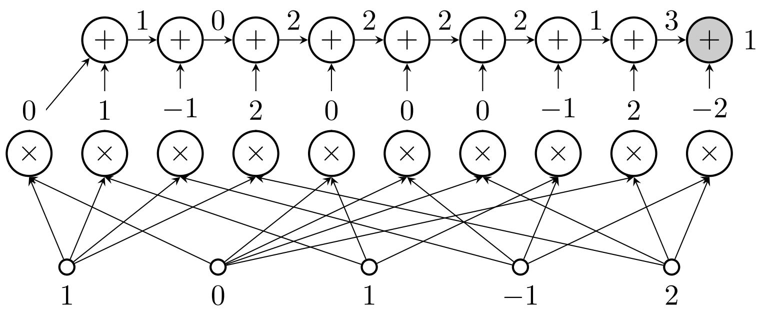


Figure 1: A circuit for $y(x_1, x_2, x_3, x_4, x_5)$.

Each vertex that is not a source represents an elementary operation. If we execute these operations in order that respects the direction of the edges the output y will be "stored" in the rightmost top vertex. This is an example of a circuit.

Given a finite set of basic operations, a *circuit* is a DAG in which every source vertex is labeled by an input variable x_i or a constant and every non-source vertex—called a *gate*—is labeled with a basic operation *op*. The in-degree of every such vertex must equal the number of arguments of *op*.

To evaluate the circuit on a given input, we topologically sort the vertices and evaluate the gates in order. For example, the computation $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, $x_4 = -1$, $x_5 = 2$ yields the following sequence of evaluations from left to right starting at the bottom and moving towards the top:



One natural measure of complexity is the number of gates. In this example the complexity is 19. Can we do better? Here is a another circuit that computes the same function y but has complexity 16: The two

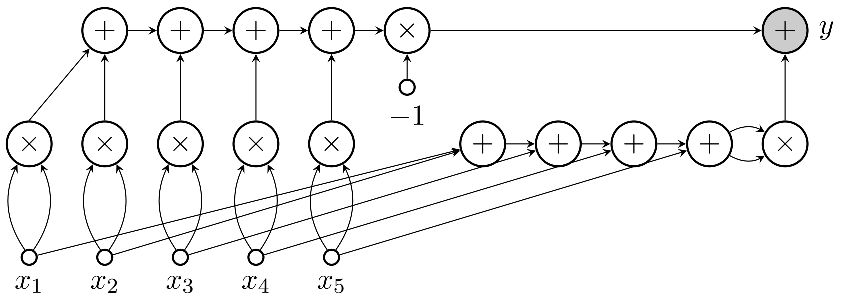


Figure 2: Another circuit for $y(x_1, x_2, x_3, x_4, x_5)$.

circuits compute the same function because y can also be expressed as

$$y(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2 + x_3 + x_4 + x_5)^2 - (x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2).$$

The effect of a good circuit choice on complexity can be quite significant when there are many inputs. For instance we can generalize both the Figure 1 and Figure 2 circuits to ones that compute the sum $\sum_{1 \le i < j \le n} x_i x_j$ of all distinct product pairs of n inputs. The circuit complexities would scale as $\Theta(n^2)$ and $\Theta(n)$, respectively. The Figure 2 type of circuit is clearly preferable.

Here is an even more dramatic example. Suppose you want to calculate the expression

$$z = \sum_{\substack{S \subseteq \{1,\ldots,20\} \\ |S|=10}} \prod_{i \in S} x_i = x_1 \cdots x_9 x_{10} + x_1 \cdots x_9 x_{11} + \cdots + x_1 \cdots x_9 x_{20} + \cdots + x_{11} \cdots x_{19} x_{20}.$$

which is the sum of all products of 10 distinct variables among the inputs $x_1, x_2, \ldots, x_{20}$. This is a sum of 184756 terms, suggesting that a circuit for it should have about that size. Yet it turns out that it admits a circuit of size less than 1000. The reason is that S can also be represented in this form:

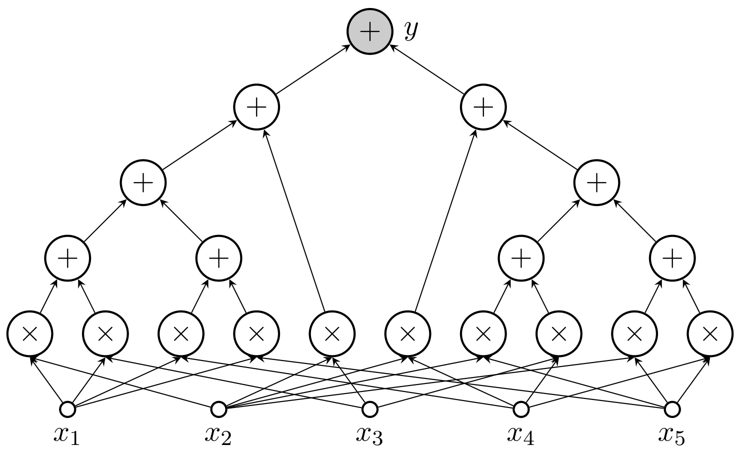
$$z = c_0 \cdot x_1 \cdots x_{20} + c_1 \cdot (x_1 + 1) \cdots (x_{20} + 1) + \cdots + c_{20} \cdot (x_1 + 20) \cdots (x_{20} + 20) \tag{2}$$

where the numbers $c_0, c_1, \ldots, c_{20}$ are

$$\begin{array}{llll}
c_0 = \frac{2965638101}{5518098432000} & c_1 = \frac{-46937467057}{4828336128000} & c_2 = \frac{1623151366349}{19313344512000} & c_3 = \frac{-742866335297}{1609445376000} \\
c_4 = \frac{23244571515317}{12875563008000} & c_5 = \frac{-716646421081}{134120448000} & c_6 = \frac{19976964872357}{1609445376000} & c_7 = \frac{-443829136609}{19160064000} \\
c_8 = \frac{226997057727689}{6437781504000} & c_9 = \frac{-106692491681983}{2414168064000} & c_{10} = \frac{40247351792213}{877879296000} & c_{11} = \frac{-95170162303973}{2414168064000} \\
c_{12} = \frac{180520357853249}{6437781504000} & c_{13} = \frac{-6601198952479}{402361344000} & c_{14} = \frac{199696061419}{25546752000} & c_{15} = \frac{-1201368202553}{402361344000} \\
c_{16} = \frac{425713023751}{476872704000} & c_{17} = \frac{-324027306227}{1609445376000} & c_{18} = \frac{622171058989}{19313344512000} & c_{19} = \frac{-15743145547}{4828336128000} \\
c_{20} = \frac{6063698587}{38626689024000} & & &
\end{array}$$

Parallel computation Modern computers have multiple cores which allow for fast parallel computation. A circuit can be naturally evaluated in parallel by coming up with a parallel schedule for the underlying DAG. The duration of the computation is then determined by the length of the longest path, assuming that enough parallel processing is available to evaluate all gates in a given layer in unit time. The *depth* of a circuit is the length of its longest path.

The circuit in Figure 1 has depth 10 as exhibited by the path that runs from x_1 through the leftmost $\times$ gate and the nine $+$ gates from left to right. Is it possible to represent y by a shallower circuit? One way to do this is to rewire the $+$ gates in a manner that takes advantage of the commutativity of addition:



The depth of this circuit for y goes down to 5.

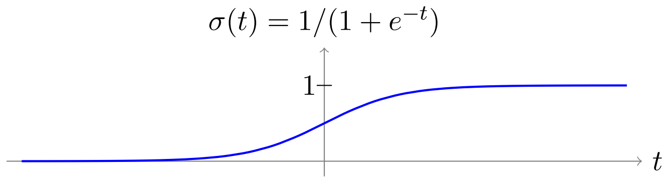
2 Perceptrons

When computers took off in the 1950s and 60s scientists began speculating that they can be used to shed light on the workings of human intelligence. Their premise was that the mind can be described as a circuit whose inputs are our sensory perceptions (vision, sound, smell, touch). What are the gates of these "circuits of the mind"? Neuroscience tells us that the basic blocks of the nervous system, including the brain, are neural cells, or neurons. These cells take in electrical signals from the sensory system or from other neurons and amplify or inhibit them depending on their function.

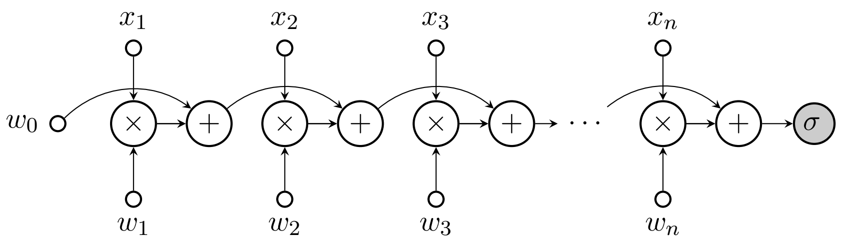
A perceptron is a model of a single neuron. It consists of two types of inputs: n *signals* x_1 up to x_n and an $n+1$ *weights* $w_0, w_1, \ldots, w_n$. It outputs the value

$$y = \sigma(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n),$$

where σ is a fixed real-valued function called the *activation function*. One popular choice of activation function is the *logistic function* $\sigma(t) = 1/(1 + e^{-t})$ which increases from 0 to 1:



The perceptron can be implemented naturally by this circuit:



For example, a perceptron can be used to decide whether the room is quiet or noisy based on the readings of five "noise sensors" x_1, x_2, x_3, x_4, x_5. Suppose a given sensor outputs 1 if it detects noise and 0 if it doesn't. The perceptron $\sigma(-5 + 2x_1 + 2x_2 + 2x_3 + 2x_4 + 2x_5)$ would then output the following overall noise estimate depending on the number k of sensors that output 1:

k	0	1	2	3	4	5
σ	0.01	0.05	0.27	0.73	0.95	0.99

Perceptrons are not only useful for modelling natural intelligence but also for endowing machines with the ability to make decisions from data. Suppose that you want to estimate your chances y of getting an A in Discrete Structures. This might depend on a variety of factors such as your midterm grade x_1, your quiz average x_2, and the number of hours x_3 that you studied each week. Your chances y can then reasonably be modeled as the output of some perceptron $\sigma(w_0 + w_1x_1 + w_2x_2 + w_3x_3)$. But how should you choose the weights w_0 to w_3?

The central dogma of Machine Learning is that you can estimate unknown parameters like w_0, w_1, w_2, w_3 from data using a *training algorithm.* From talking to your friends who took the course last year you gathered the following data:

name	x_1	x_2	x_3	A?
Alice	39	31	12	yes
Bob	45	11	3	no
Charlie	43	25	6	yes
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Zack	50	40	0	yes

For any given student s in the table, you would expect that the "indicator value" $\hat{y}(s)$ for the event that the student received an A should be close to the perceptron's estimate of this value:

$$\hat{y}(s) \approx \sigma(w_0 + w_1x_1(s) + w_2x_2(s) + w_3x_3(s)).$$

This approximation will never be exact; the left-hand side is a binary value (0 or 1) while the right hand side is some real number between 0 and 1. A natural error measure is the square loss

$$\ell(s) = \big(\hat{y}(s) - \sigma(w_0 + w_1x_1(s) + w_2x_2(s) + w_3x_3(s))\big)^2.$$

It is sensible to try to pick the weights w_0, w_1, w_2, w_3 so as to miminize the sum z of the individual losses:

$$\begin{aligned} z &= \ell(\text{Alice}) + \ell(\text{Bob}) + \cdots + \ell(\text{Zack}) \\ &= \big(1 - \sigma(w_0 + 39w_1 + 31w_2 + 12w_3)\big)^2 \\ &\quad + \big(0 - \sigma(w_0 + 45w_1 + 11w_2 + 3w_3)\big)^2 \\ &\quad \vdots \\ &\quad + \big(1 - \sigma(w_0 + 50w_1 + 40w_2 + 0w_3)\big)^2. \end{aligned}$$

Minimizing such expressions can be quite difficult. There is however a natural strategy to try: Start with an initial guess for the unknowns w_0, w_1, w_2, w_3, then keep moving the "point" (w_0, w_1, w_2, w_3) in the direction in which the value of z decreases most rapidly. We know from calculus that this direction is the opposite of the gradient

$$\nabla z = \left(\frac{\partial z}{\partial w_0}, \frac{\partial z}{\partial w_1}, \frac{\partial z}{\partial w_2}, \frac{\partial z}{\partial w_3}\right).$$

To summarize, one general recipe for "learning" the weights w_0, w_1, w_2, w_3 is to start with an initial guess, then move a bit in the direction of $-\nabla z(w_0, w_1, w_2, w_3)$, and so on, until a "local minimum" in which $\nabla z \approx 0$ is reached. This type of minimization method is called gradient descent.

To turn gradient descent into an actual algorithm you must specify several things, including your choice of initial guess, the amount by which you move in the direction of $-\nabla z$, and so on. You can learn more about it in courses on optimization and machine learning. An indispensable part of any implementation is, however, the calculation of the gradient ∇z at various points (w_0, w_1, w_2, w_3).

The function z that describes the sum of losses for the perceptron is simple enough that we can do this calculation by hand. We can do it by adding up the gradients of the individual losses:

$$\nabla z = \nabla \ell(\text{Alice}) + \nabla \ell(\text{Bob}) + \cdots + \nabla \ell(\text{Zack}).$$

For any fixed student, the partial derivative $\partial \ell / \partial w_i$ is

$$\frac{\partial}{\partial w_i}\big(\hat{y} - \sigma(w_0 + w_1x_1 + w_2x_2 + w_3x_3)\big)^2 = -2x_i\big(\hat{y} - \sigma(w_0 + w_1x_1 + w_2x_2 + w_3x_3)\big)\sigma'(w_0 + w_1x_1 + w_2x_2 + w_3x_3).$$

(When $i = 0$ we omit x_0.) Thus we can evaluate ∇z at any point (w_0, w_1, w_2, w_3) by summing up these expressions over the data table, for example

$$\begin{aligned}\frac{\partial z}{\partial w_1} = &-68(1 - \sigma(w_0 + 39w_1 + 31w_2 + 12w_3))\sigma'(w_0 + 39w_1 + 31w_2 + 12w_3)\\ &- 90(0 - \sigma(w_0 + 45w_1 + 11w_2 + 3w_3))\sigma'(w_0 + 45w_1 + 11w_2 + 3w_3)\\ &\vdots\\ &- 100(1 - \sigma(w_0 + 50w_1 + 40w_2))\sigma'(w_0 + 50w_1 + 40w_2 + 0w_3).\end{aligned}$$

Here σ' is the derivative of the logistic function, namely $\sigma'(t) = e^{-t}/(1 - e^{-t})^2$.

3 Backpropagation

While perceptrons are convenient to work with they are inadequate for more complex data-driven decision tasks. Suppose you want to know whether an image has a cat in it. The inputs $x_1, x_2, \ldots, x_n$ are the pixels and the output y is supposed to equal 1 if $x_1, \ldots, x_n$ form a cat and 0 if they don't. A perceptron would try to base its decision on the value $\sigma(w_0 + w_1x_1 + \cdots + w_nx_n)$ for some weights that represent the importance of different pixels. A weighted sum of pixels cannot take into account high-level features of vision such as edges between objects, foreground and background layers, and so on.

A more expressive model can be obtained by taking multiple perceptrons and organizing them in layers. By analogy with biological neurons we may expect that neurons closer to the inputs should be adequate for lower-level perception tasks such as edge-detection, while neurons closer to the output level may perform more cognitively demanding roles such as object detection and classification (e.g. is it an animal?)

There is a steep price to pay for this complexity: The model now consists of not one but many perceptrons so the number of unknown weights that needs to be estimated from the training data becomes very large. Moreover, the circuits representing these models become more complex. For the training to complete in a reasonable amount of time it is essential to have a systematic and efficient way to evaluate the partial derivatives of the loss. As the loss is usually some simple function of the model, which is itself a circuit whose inputs are the unknown "weights", this task can be captured by the following problem:

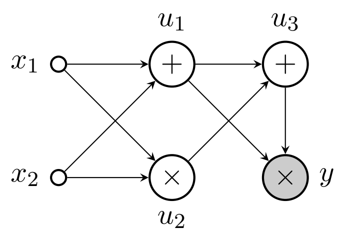
> Given a circuit C with inputs $x_1, \ldots, x_n$ and a designated output gate y, design a circuit ∇C that computes all partial derivatives $\partial y/\partial x_1, \ldots, \partial y/\partial x_n$.

There are two algorithms for this problem. Both solve the problem inductively in order determined by a topological sort of the gates of C. The more natural forward propagation algorithm visits the vertices in order from sources to sink. In general, if C has n inputs and g gates, forward propagation produces a circuit ∇C of size $O(ng)$. Modern neural networks can easily have n and g in the millions resulting in a prohibitively large circuit. In contrast, the *backpropagation algorithm* produces a circuit of size $O(n + g)$ only.

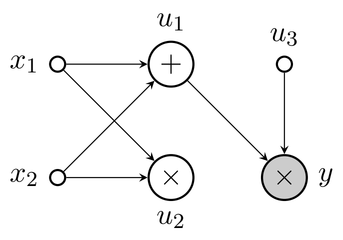
The backpropagation algorithm calculates partial derivatives by visiting the vertices in reverse topological order, starting from the output y and ending with the sources $x_1, \ldots, x_n$. Upon visiting node z, backpropagation constructs a new gate that calculates the partial derivative $\partial y/\partial z$. To explain what this means we need to define the partial derivative of one circuit gate y with respect to another gate z.

Definition 1. Assume C is a circuit, y is a gate, and z is a vertex (input or gate). Let $(C - z)$ be the circuit obtained by removing all edges that point to z from C and turning z into an input (if it is not already one). Then y computes some function $f(z, \text{other inputs})$ in $(C - z)$. The partial derivative $\partial y/\partial z$ is the derivative of f with respect to z.

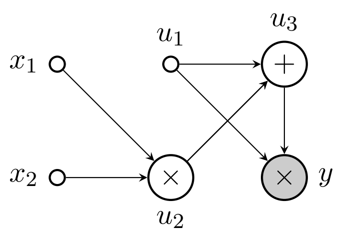
Let's work out an example. Consider the following circuit that computes the function $y(x_1, x_2) = (x_1 + x_2 + x_1x_2)(x_1 + x_2)$. The ordering $x_1, x_2, u_1, u_2, u_3, y$ is a topological sort of its vertices. To its right are the circuits $(C - u_3)$ and $(C - u_1)$.



circuit C



circuit $(C - u_3)$



circuit $(C - u_1)$

The circuit $(C - u_3)$ has inputs x_1, x_2, u_3. The function computed by y in $(C - u_3)$ is $u_1 \cdot u_3 = (x_1 + x_2) \cdot u_3$. The partial derivative $\partial y/\partial u_3$ is then $\partial(u_1 \cdot u_3)/\partial u_3 = u_1 = x_1 + x_2$. The last simplification was possible because u_1 does not depend on u_3 (as it precedes it in the topological sort).

The circuit $(C - u_1)$ has inputs x_1, x_2, u_1. The function computed by y in $(C - u_1)$ is $u_1 \cdot u_3^* = u_1 \cdot (u_1 + u_2) = u_1 \cdot (u_1 + x_1 + x_2)$. Here u_3^* stands for the function computed by u_3 in $(C - u_1)$: This is not the same as the function computed by u_3 in C! The input u_1 appears twice in this expression, once explicitly as an argument of the product gate y and once implicitly via u_3^* which itself depends on u_1. We can calculate the partial derivative of this function using the product rule for derivatives:

$$\frac{\partial y}{\partial u_1} = \frac{\partial(u_1 \cdot u_3^*)}{\partial u_1} = \frac{\partial u_1}{\partial u_1} \cdot u_3^* + u_1 \cdot \frac{\partial u_3^*}{\partial u_1} = 1 \cdot (u_1 + u_2) + u_1 \cdot \frac{\partial(u_1 + u_2)}{\partial u_1} = (u_1 + u_2) + u_1 = 2u_1 + u_2.$$

This expresses the desired partial derivative as a small circuit in terms of the gates u_1 and u_2, which are already present in C. We can compute $\partial y/\partial u_3$ from the existing gates for u_1 and u_2 and a bit of extra work (one addition and one multiplication). Is there a general method for this?

Let us first rework the expression for $\partial y/\partial u_1$ in a more systematic way. To do this it will be useful to introduce another piece of notation. For a gate g taking inputs a, b, let $\partial_a[g]$ denote the partial derivative of the gate operation $[g]$ with respect to argument a. For example, for a product gate $[g](a, b) = a \cdot b$ we have $\partial_a[g] = b$. In general, $\partial_a[g]$ is not the same as $\partial g/\partial a$. In the circuit $(C - u_1)$, $\partial_{u_1}[y] = \partial(u_1u_3^*)/\partial u_1 = u_3^*$, while $\partial y/\partial u_1 = u_3^* + u_1$. The reason is that u_1 affects y partially through the edge (u_1, y) and partially through the path (u_1, u_3, y). In this example we can represent y as $y = [y]([u_3](u_1, u_2), u_1)$. By the chain rule,

$$\frac{\partial y}{\partial u_1} = \frac{\partial y}{\partial u_3} \cdot \partial_{u_1}[u_3] + \partial_{u_1}[y].$$

As we already calculated, $\partial y/\partial u_3 = u_1$. As for the other terms, u_3 is a sum gate so $\partial_{u_1}[u_3] = 1$ and u_1 is a product gate so $\partial_{u_1}[y] = u_3 = u_1 + u_2$. We obtain again $\frac{\partial y}{\partial u_1} = 2u_1 + u_2$.

In general, suppose we have a gate z whose out-edges point to gates $z_1, \ldots, z_t$. In the circuit $(C - z)$, the output y depends on z via the gates $z_1, \ldots, z_t$, each of which depends on z (one of which could be y itself). Using the chain rule for derivatives,

$$\frac{\partial y}{\partial z} = \frac{\partial y}{\partial z_1} \cdot \partial_z[z_1] + \cdots + \frac{\partial y}{\partial z_t} \cdot \partial_z[z_t]. \tag{3}$$

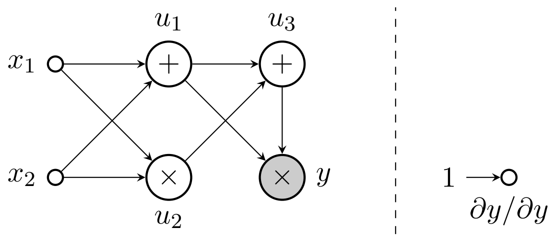
This expresses $\partial y/\partial z$ as a small circuit that depends on partial derivatives of y with respect to gates that succeed y in the topological sort and some derivatives of the gates, suggesting the following iterative algorithm for partial derivatives.

The Backpropagation Algorithm. Given a circuit C as an input,

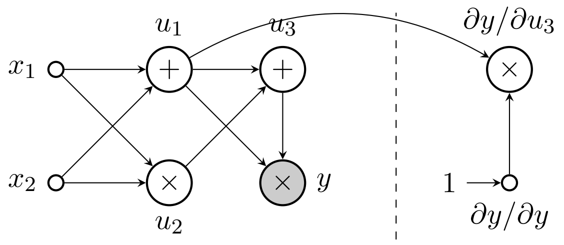
```
    Compute a reverse topological sort $s$ of $C$'s vertices starting with $y$.
    For every $z$ in $s$, construct a subcircuit for $\partial y/\partial z$ using formula (3).
    Output the resulting circuit $\nabla C$.
```

We demonstrate an execution of backpropagation on our example circuit C. It is useful to visualize the circuit ∇C by placing the gate $\partial y/\partial z$ in a position mirroring the gate z.

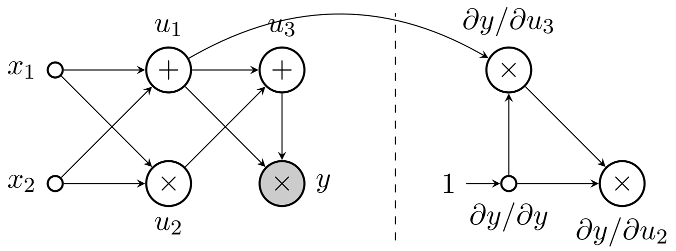
Our reverse topological sort s is the ordering $y, u_3, u_2, u_1, x_2, x_1$. The first vertex y is isolated in the circuit $(C - y)$, so $\partial y/\partial y = 1$.



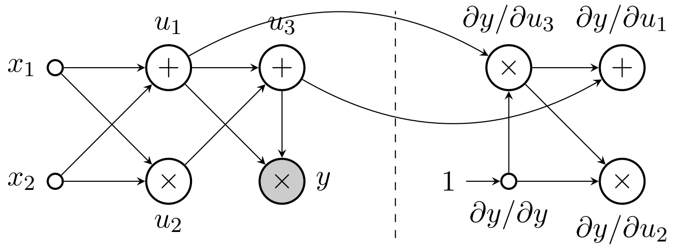
Next in s is u_3. As u_3 has a single out-edge going into y, (3) gives $\partial y/\partial u_3 = \partial y/\partial y \cdot \partial_{u_3}[y] = \partial y/\partial y \cdot u_1$:



Next is u_2, which has a single edge pointing to u_3. From (3) we get $\partial y/\partial u_2 = \partial y/\partial u_3 \cdot \partial_{u_2}[u_3] = \partial y/\partial u_3 \cdot 1$.

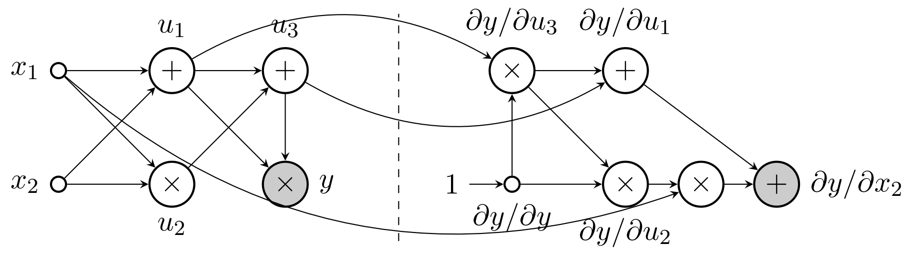


Next is u_1, which has outgoing edges to u_3 and y. Using (3) we get $\partial y/\partial u_1 = \partial y/\partial u_3 \cdot \partial_{u_1}[u_3] + \partial y/\partial y \cdot \partial_{u_1}[y] = \partial y/\partial u_3 \cdot 1 + \partial y/\partial y \cdot u_3$. To keep the picture simple we'll use $\partial y/\partial y = 1$ and omit the multiplicaton gates.

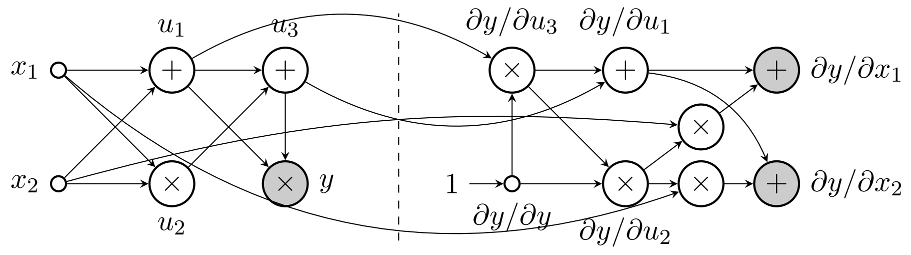


u_1
u_3
$\partial y/\partial u_3$
$\partial y/\partial u_1$
x_1
x_2
u_2
y
1
$\partial y/\partial y$
$\partial y/\partial u_2$

The second to last vertex is x_2, with edges pointing to u_1 and u_2. We have $\partial y/\partial x_2 = \partial y/\partial u_1 \cdot \partial_{x_2}[u_1] + \partial y/\partial u_2 \cdot \partial_{x_2}[u_2] = \partial y/\partial u_1 + \partial y/\partial u_2 \cdot x_1$.


u_1
u_3
$\partial y/\partial u_3$
$\partial y/\partial u_1$
x_1
x_2
y
1
$\partial y/\partial x_2$
u_2
$\partial y/\partial y$
$\partial y/\partial u_2$

Finally we construct the gate $\partial y/\partial x_1$ using (3) one last time.


u_1
u_3
$\partial y/\partial u_3$
$\partial y/\partial u_1$
x_1
$\partial y/\partial x_1$
x_2
y
1
$\partial y/\partial x_2$
u_2
$\partial y/\partial y$
$\partial y/\partial u_2$

At this point it is a good idea to check that this circuit produces correct outputs:

$$\begin{aligned}\frac{\partial y}{\partial x_1} &= \frac{\partial y}{\partial u_2}\cdot x_2 + \frac{\partial y}{\partial u_1}\\ &= \frac{\partial y}{\partial u_3}\cdot x_2 + \left(u_3 + \frac{\partial y}{\partial u_3}\right)\\ &= u_1 x_2 + (u_3 + u_1)\\ &= (x_1+x_2)x_2 + (2u_1 + u_2)\\ &= (x_1+x_2)x_2 + (2x_1 + 2x_2 + x_1x_2)\\ &= 2x_1 + 2x_2 + 2x_1x_2 + x_2^2.\end{aligned}$$

You can verify that this is indeed the partial derivative of $(x_1 + x_2 + x_1x_2)(x_1 + x_2)$ with respect to x_1.

We can now state the correctness of the backpropagation algorithm. Given a set of basic operations B, let ∇B be the set of all partial derivatives of all operations in B.

Theorem 2. *For any circuit C with designated output y and operations coming from B, the output ∇C of Backpropagation is a circuit with operation set $B \cup \nabla B \cup \{+, \times\}$ that contains gates computing $\partial y/\partial u$ for every vertex u of C. The number of gates in ∇C is at most three times the number of edges plus the number of vertices in C.*

The proof is by strong induction with respect to the ordering s. The gate count comes from formula (3): Each edge (z, z_i) in G contributes one gate $\partial_z[z_i]$ in ∇G, one multipilication $\partial y/\partial z_i \cdot \partial_z[z_i]$, and at most one addition in the chain rule.

References

Formula (2) is a special case of an unpublished construction of Ben-Or. The history of backpropagation looks complicated. It seems to have been discovered, forgotten, and rediscovered several times.