In Lecture 2 we needed the gradient ∇f of the sum-of-squares loss function $f(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2$ in order to run gradient descent on it. We gave a formula for the gradient without explaining it. This lecture will be all about how to calculate gradients.

This may sound like a specialized problem. It turns out to be an important one. The choice of algorithm makes a huge difference.

1 Forward propagation

As a running example we'll look at the function

$$f(x, y) = (x - 2y)^2 + (x + y - 3)^2. \tag{1}$$

This is the sum-of-squares loss for the linear system $x - 2y = 0, x + y = 3$. You have probably spent a great deal of time in calculus class computing gradients of functions like this one. The main message of differential calculus is that no matter how complicated a function looks, its derivatives can be obtained by following a few simple rules like the sum rule, the product rule, and the chain rule. When these rules stop applying we resort to substitutions for "simple" functions like $de^x/dx = e^x$, $d\sin x/dx = \cos x$, and $dx/dx = 1$.

If we were to write a program to calculate gradients, how should we go about it? Experience suggests that a "top-down" approach tends to work. To calculate $\partial f/\partial x$ in (1), we would start with the sum rule to break up the task into two parts:

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}(x - 2y)^2 + \frac{\partial}{\partial x}(x + y - 3)^2. \tag{2}$$

The first term can then be handled using the chain rule

$$\frac{\partial}{\partial x}(x - 2y)^2 = 2(x - 2y) \cdot \frac{\partial}{\partial x}(x - 2y)$$

and the inner term now using the sum rule:

$$\frac{\partial}{\partial x}(x - 2y) = \frac{\partial}{\partial x}x + \frac{\partial}{\partial x}(-2y)$$

At this point we resort to the substitutions $\partial x/\partial x = 1$ and $\partial(\text{expression that does not contain } x)/\partial x = 0$. Thus $\partial(x - 2y)^2/\partial x$ equals $2(x - 2y)$. By identical logic $\partial(x + y - 3)^2/\partial x = 2(x + y - 3)$. Plugging both back into (2) we find

$$\frac{\partial f}{\partial x} = 2(x - 2y) + 2(x + y - 3) \tag{3}$$

and by the same logic

$$\frac{\partial f}{\partial y} = -4(x - 2y) + 2(x + y - 3). \tag{4}$$

I will call this algorithm *forward propagation*. (I am not sure what is the standard name for it.) The gradient of a complex function is obtained by "propagating" the gradients of its constitutent parts forward.

Algorithm: FP (Forward Propagation for polynomials)
Input: A polynomial $f(x, y, \ldots, z)$.
1 If f is a constant, output $\nabla f = (0, 0, \ldots, 0)$.
2 If f is a variable, say x, output $\nabla f = (1, 0, \ldots, 0)$.
3 If f is of the form $g + h$, output $FP(g) + FP(h)$ (sum rule).
4 If f is of the form $g \times h$, output $g \times FP(h) + h \times FP(g)$ (product rule).

To understand Forward Propagation we need to talk about the representation of functions like (1) and (2). In Lecture 4 we saw two representations of functions: the list representation and the Fourier representation. Neither of them tells us how to compute f or its derivatives.

2 Circuits

In computer science it is often useful to represent expressions like (1) and (3) not as equations but as labeled graphs called *circuits*. The circuit representation of f in (1) is shown in Figure 1a.

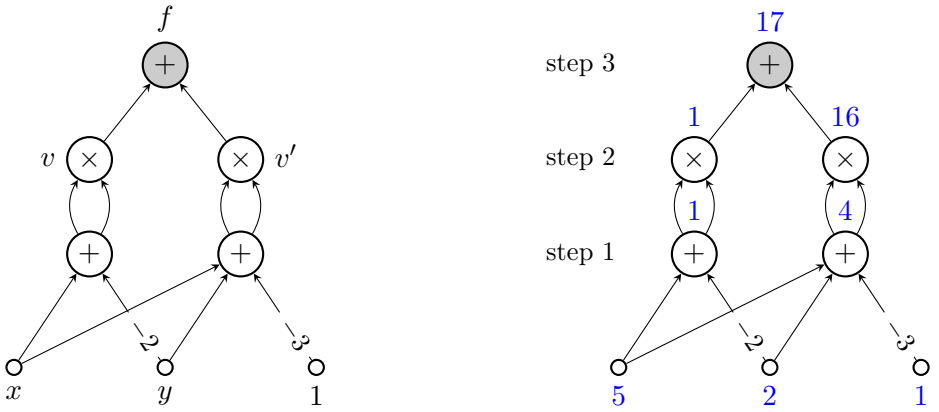
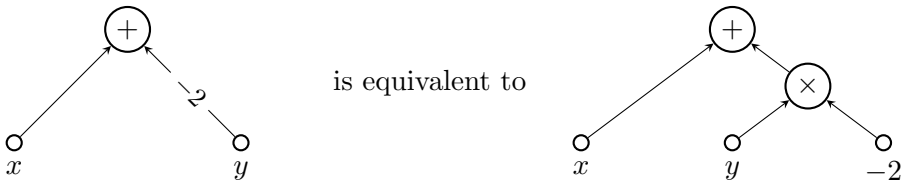


FIGURE 1: (a) The circuit representation of (1). (b) A sample evaluation.

A circuit is a directed acyclic graph in which every source vertex is labeled by an input variable or a constant. Every other vertex—called a *gate*—is labeled by a basic operation *op*. You may think of these basic operations as instructions that can be executed in hardware at unit cost. The basic operations of an *arithmetic circuit* are plus and times. Richer circuit models may allow others like divide and exponentiate.

In this example some of the edges incident to a + gate are labeled by constants describing a scaling factor. If they bother you, you can think of these factors as "syntactic sugar" for multiplication by a constant. For example:



The primary measure of computational complexity for a circuit is its number of gates. If each gate takes one time unit to evaluate then the size of the circuit is the time it takes to do the computation sequentially. The circuit in Figure 1 has size 7. A secondary complexity measure is the *depth* of the circuit. This is the length of its longest directed path. It is the minimum time it takes to evaluate the circuit in parallel, assuming each gate takes unit time. The circuit in Figure 1 has depth 3. An example evaluation is shown in Figure 1b.

3 Analysis of Forward Propagation

Forward propagation is friendly to circuit representations. Given a circuit computing f as input, it produces a circuit for its partial derivatives as output. The $\partial f/\partial x$ part of the result of running forward propagation on the circuit in Figure 1 is shown in Figure 2.

The sub-circuit computing $\partial f/\partial y$ has the same structure. The only differences are the factors on the wires coming out of the 1 node.

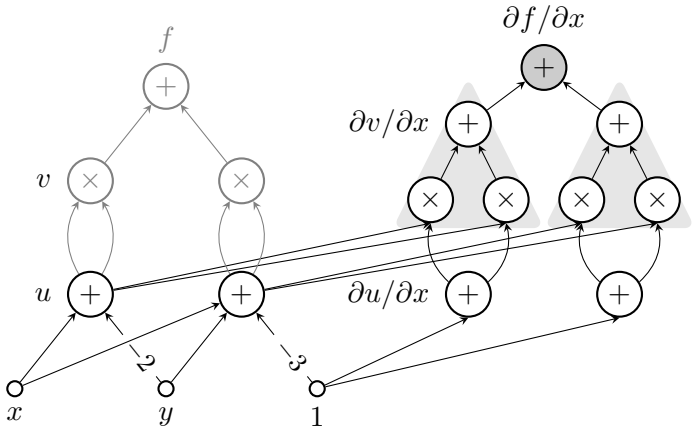


FIGURE 2: A portion of circuit produced by Forward Propagation on input (1).

The overall size of the circuit for ∇f in Figure 2 is $2 + 2 \cdot 9 = 20$. The part of the circuit computing f has 2 gates and there are 9 additional gates participating in the computation of each partial derivative. In general, if $f(x_1, \ldots, x_n)$ has circuit size s, the circuit that calculates each partial derivative $\partial f/\partial x_i$ has size proportional to s, or $\Theta(s)$. To calculate all n partial derivatives, the size grows to $\Theta(ns)$.

There is a more general variant of Forward Propagation. To explain it we'll need some notation. In a circuit C, each gate g computes a function of the inputs. This function is described recursively: It is obtained by applying the gate operation, which we will denote by $[g]$, to the functions computed by its children. For example, in the circuit in Figure 1, f is derived by applying the gate operation $[f]$ to its inputs v and v': $f = [f](v, v')$. In this example, the gate operation is addition, so $[f](v, v') = v + v'$.

Given a gate operation $[g]$ with d inputs and one output, its gradient $[\nabla g]$ can be viewed as some other gate operation with (at most) d inputs and d outputs (see Figure 3). For instance, if $[g](u, v) = u + v$ then $[\nabla g](u, v, w) = (1, 1)$, i.e., two copies of the constant 1. If $[g](u, v, w) = uvw$, then $[\nabla g](u, v, w) = (vw, wu, uv)$. If $[g](u) = \exp u$ then $[\nabla g](u) = \exp u$ and if $[g](u) = 1/u$ then $[\nabla g](u) = -1/u^2$.

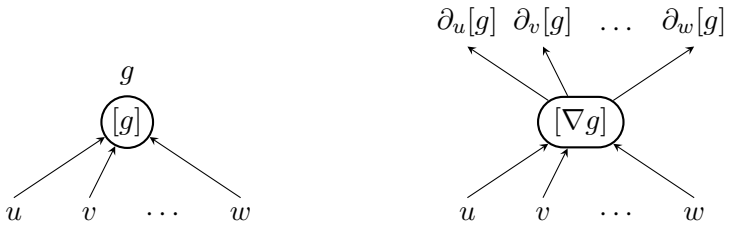


FIGURE 3: (a) A gate $[g]$ (b) The gate $[\nabla g]$ representing its gradient.

The partial derivatives of g with respect to the input variables $x, y, \ldots, z$ can be computed from the partial derivatives of u, v, w using the *chain rule*. For example, if $[g]$ is a multiplication gate with three inputs u, v, w then

$$\frac{\partial g}{\partial x} = \frac{\partial u}{\partial x} \cdot (vw) + \frac{\partial v}{\partial x} \cdot (wu) + \frac{\partial w}{\partial x} \cdot (uv).$$

This is the sum of products between the partial derivatives of the functions represented by the children of g and the gradient of g evaluated at those children:

$$\begin{aligned} \frac{\partial g}{\partial x} &= \left(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}, \ldots, \frac{\partial w}{\partial x}\right) \cdot [\nabla g](u, v, \ldots, w) \\ &= \frac{\partial u}{\partial x} \cdot \partial_u[g] + \frac{\partial v}{\partial x} \cdot \partial_v[g] + \cdots + \frac{\partial w}{\partial x} \cdot \partial_w[g] \end{aligned} \tag{5}$$

Here, $\partial_v[g]$ stands for the v-th component of the gradient gate $[\nabla g]$. This is the partial derivative of the gate with respect to argument v:

$$[\nabla g](u, v, \ldots, w) = (\partial_u[g], \partial_v[g], \ldots, \partial_w[g]).$$

The chain rule (5) is valid for any gate. The sum and product rules are special cases of it. Forward Propagation applies it systematically to transform a circuit C into its gradient ∇C.

Algorithm: *FP* (General Forward Propagation)
Input: A circuit C with n inputs.
1 For every gate g in C with children $u, v, \ldots, w$,
2 Create a new gate ∇g with inputs $u, v, \ldots w$ and outputs $\partial_u[g], \partial_v[g], \ldots, \partial_w[g]$.
3 For every input variable x,
4 Create a new node $\partial g/\partial x$.
5 Connect $\partial u/\partial x, \partial v/\partial x, \ldots, \partial w/\partial x$ and ∇g to $\partial g/\partial x$ using the chain rule (5).
6 Output the resulting circuit ∇C.

We can apply the chain rule inductively to conclude that Forward Propagation correctly calculates all the partial derivatives.

Theorem 1. *Each gate $\partial g/\partial x$ in ∇C computes the partial derivative of g with respect to input variable x.*

How large is the resulting circuit? In step 4 each gate g of C is augmented with $\Theta(n)$ new gates, one for every partial derivative. In step 5 each wire (edge) of C contributes an extra times gate to each of these partial derivatives. If C has s gates and w wires then ∇C will have $\Theta(n \cdot (s + w))$ gates.

Can we do better? As an example, the output of Forward Propagation for C in Figure 1 is quite a bit larger than C: ∇C has size 20, while C has only size 5. In this example a much smaller circuit for the gradient of the same function exists. It is shown in Figure 4.

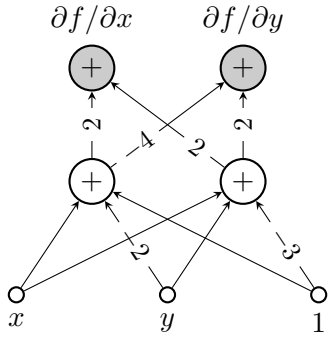


FIGURE 4: A small circuit for the gradient of (1).

I constructed this circuit "by hand" after inspecting (3) and (4). The partial derivatives share many common terms that can be reused in the circuit for ∇f. Can this construction procedure be automated for general f?

4 Backpropagation

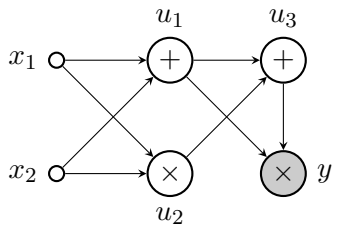
Backpropagation is an ingenious alternative algorithm for calculating the gradient of a circuit C. If C has n inputs, s nodes, and w wires, Backpropagation produces a circuit for ∇C of size $O(n + s + w)$. This is a big improvement over Forward Propagation for circuits with many inputs.

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,

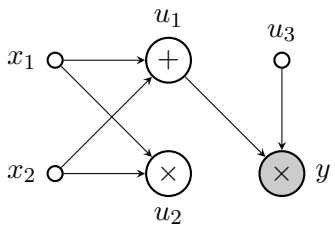
backpropagation constructs a new gate that calculates the partial derivative $\partial y/\partial z$. To explain what this means we need to define the partial derivative of one circuit gate y with respect to another gate z.

Definition 2. 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.

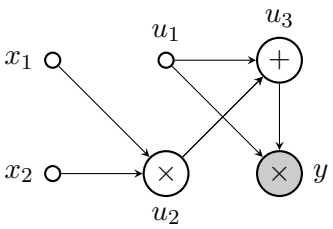
Let's work out an example. Consider the following circuit that computes the function $y(x_1, x_2) = (x_1 + x_2 + x_1x_2)(x_1 + x_2)$. The ordering $x_1, x_2, u_1, u_2, u_3, y$ is a topological sort of its vertices. To its right are the circuits $(C - u_3)$ and $(C - u_1)$.



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 $u_1 \cdot u_3^* = u_1 \cdot (u_1 + u_2) = u_1 \cdot (u_1 + x_1x_2)$. 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).

Here is the general method. Suppose we have a gate g whose out-edges point to its *parents* $u, v, \ldots, w$. In the circuit $(C - g)$, the output y depends on g via the gates $u, v, \ldots, w$ (one of which could be y itself), each of which depends on g. The chain rule now expresses $\partial y/\partial g$ as a small circuit of the partial derivatives with respect to its parents $\partial y/\partial u, \partial y/\partial v, \ldots, \partial y/\partial w$:

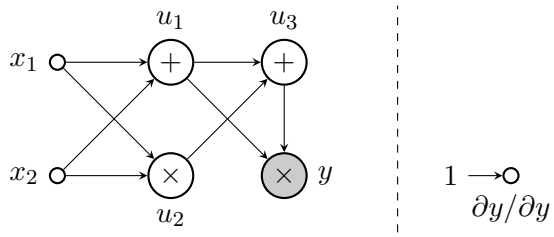
$$\frac{\partial y}{\partial g} = \frac{\partial y}{\partial u} \cdot \partial_g[u] + \frac{\partial y}{\partial v} \cdot \partial_g[v] + \cdots + \frac{\partial y}{\partial w} \cdot \partial_g[w]. \tag{6}$$

If we had circuits calculating the latter, formula (6) tells us how to combine them to calculate the former. Backpropagation does that systematically.

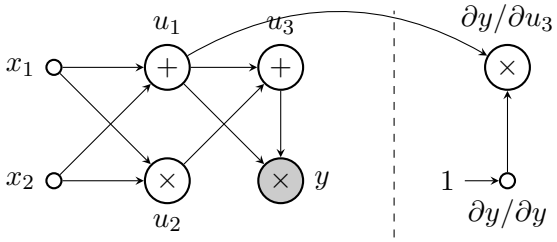
Algorithm *BP* (Backpropagation)
Input: A circuit C.
1 Compute a reverse topological sort s of C's vertices starting with y.
2 For every g in s, construct a partial circuit for $\partial y/\partial g$ using the chain rule (6).
3 Output the resulting circuit ∇C.

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

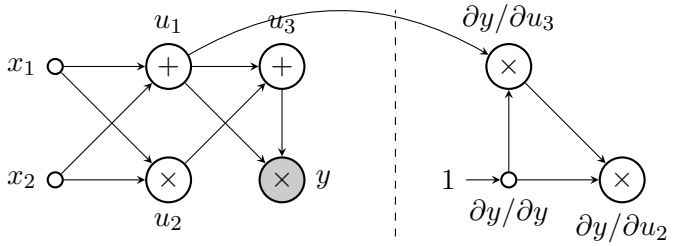
Our reverse topological sort s is the ordering $y, u_3, u_2, u_1, x_2, x_1$. The first vertex y is isolated in the circuit $(C - y)$, so $\partial y/\partial y = 1$, as expected.



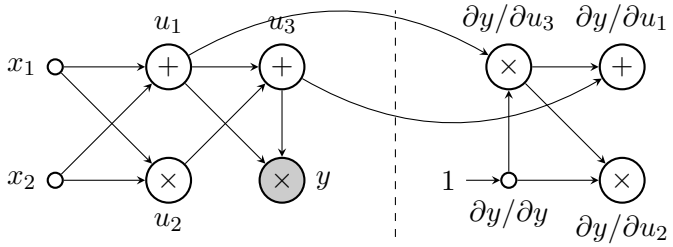
Next in s is u_3. As u_3 has only y as parent, (6) gives $\partial y/\partial u_3 = \partial y/\partial y \cdot \partial_{u_3}[y] = \partial y/\partial y \cdot u_1$:



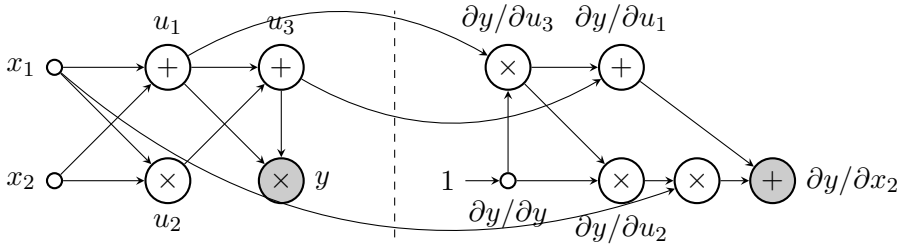
Next is u_2, which has a single parent u_3. From (6) we get $\partial y/\partial u_2 = \partial y/\partial u_3 \cdot \partial_{u_2}[u_3] = \partial y/\partial u_3 \cdot 1$.



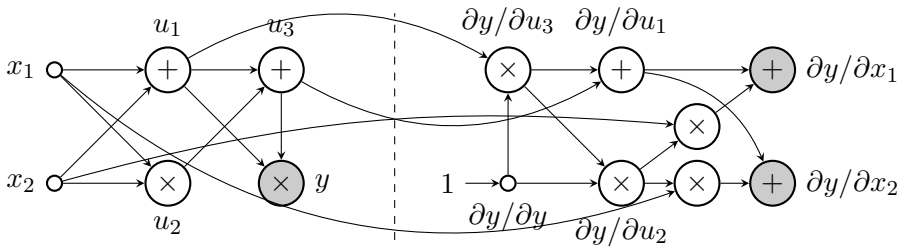
Next is u_1, which has two parents u_3 and y. Using (6) we get $\partial y/\partial u_1 = \partial y/\partial u_3 \cdot \partial_{u_1}[u_3] + \partial y/\partial y \cdot \partial_{u_1}[y] = \partial y/\partial u_3 \cdot 1 + \partial y/\partial y \cdot u_3$. To keep the picture simple we'll use $\partial y/\partial y = 1$ and omit the multiplicaton gates.



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



Finally we construct the gate $\partial y/\partial x_1$ using (6) one last time.



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

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

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 3. *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 node u of C. The number of gates in ∇C is at most three times the number of wires plus the number of nodes in C.*

The proof is by strong induction with respect to the ordering s. The gate count comes from formula (6): Each wire from a node g to a parent u in C contributes one gate $\partial_u[g]$ in ∇C, one multipilication $\partial y/\partial u \cdot \partial_u[g]$, and at most one addition in the chain rule.

Running backpropagation on the sum-of-squares loss $C(\mathbf{x}) = \|A\mathbf{x} - b\|^2$ produces the circuit $\nabla C(\mathbf{x}) = 2A^T(A\mathbf{x} - b)$, provided the product gates in C are replaced with square gates ($[g](x) = x^2, [\nabla g](x) = 2x$). As an exercise I recommend trying it on the circuit in Figure 1. The output of backpropagation will be the circuit in Figure 4.

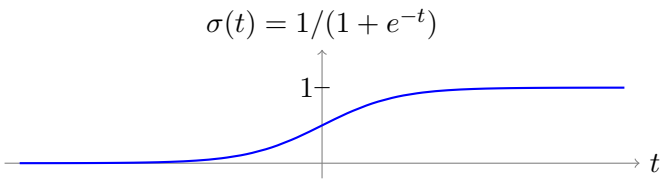
5 Neural networks

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.

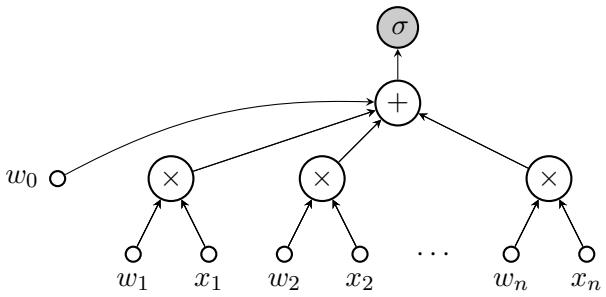
A perceptron is a model of a single neuron. It consists of two types of inputs: n *signals* x_1 up to x_n and an $n + 1$ *weights* $w_0, w_1, \ldots, w_n$. It outputs the value

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

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 by a circuit with multiplications, an addition, and a σ gate:



For example, a perceptron can be used to decide whether the room is dim or bright based on the readings of five "sensors" x_1, x_2, x_3, x_4, x_5. Suppose a given sensor outputs 1 if it detects light and -1 if it doesn't. The perceptron $\sigma(x_1 + x_2 + x_3 + x_4 + x_5)$ would then output an overall brightness 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 this course. This might depend on a variety of factors such as your homework average x_1, your project grade 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
	⋮	⋮	⋮	⋮
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 f of the individual losses:

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

Minimizing such expressions can be quite difficult. Gradient Descent is a natural algorithm to try. To run it we need to calculate the gradient at various inputs. That is where a circuit for the gradient comes in handy. In this example f is a function of only four model parameters w_0, w_1, w_2, w_3, but one of fairly large circuit size. Forward Propagation and Backpropagation will produce circuits for ∇f of comparable size. In models with dozens or more parameters, however, Backpropagation yields orders of magnitude savings in efficiency.

Deep networks

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 gradient of the loss. Backpropagation is indispensable for training such deep networks effectively.

6 Circuit complexity

In calculus class we learn that the derivative of a univariate function f is the limiting value of $(f(x+\epsilon) - f(x))/\epsilon$ as ϵ approaches zero. The definition suggests a third "algorithm" for approximating the gradient: Estimate each partial derivative $\partial f/\partial x_i$ by $(f(x_1, \ldots, x_i + \epsilon, \ldots, x_n) - f(x_1, \ldots, x_n))/\epsilon$ for some small ϵ. This algorithm is tricky to implement. First, is unclear how to choose a suitable ϵ. Even if we did, calculating a ratio of numbers close to zero is notoriously sensitive to precision errors. Small changes in numerator and denominator cause the output to swing widely.

Nevertheless, suppose you had a magic wand that would make these issues go away. This "dream" algorithm would still be *less* efficient than Backpropagation: It would require evaluating f at $n+1$ different points, leading to a running time of $\Theta(ns)$. It would have the same asymptotic complexity as Forward Propagation.

What is remarkable about Backpropagation is that it evaluates all n partial derivatives of f in much less time than it takes to calculate f at n different inputs. We usually expect that the amount of effort it takes to solve n problems should be n times larger than that for a single problem. Backpropagation

bucks this intuition by virtue of its *white-box* nature: It ingeniously exploits the circuit representation of its input f.

This equivalence between the complexity of a function and its gradient has a surprising consequence regarding the circuit complexity of matrix computations. The goal of circuit complexity is to find the smallest possible circuit for computing a given function. For example, the second symmetric polynomial

$$\sigma_2(x_1, \ldots, x_n) = \sum_{i<j} x_i x_j = x_1x_2 + x_1x_3 + \cdots + x_1x_n + x_2x_3 + \cdots + x_{n-1}x_n$$

is expressed here as a sum of $\binom{n}{2}$ products. The circuit implementing it has size $\binom{n}{2} + 1 = \Theta(n^2)$. Is there a smaller circuit for it (when n is large)? Indeed there is: We can rewrite σ_2 as

$$\sigma_2(x_1, \ldots, x_n) = \tfrac{1}{2}(x_1 + \cdots + x_n)^2 - \tfrac{1}{2}(x_1^2 + \cdots + x_n^2),$$

which computes the same σ_2 in size $n + O(1)$.

Figuring out the smallest possible circuit for a given function is very difficult. An important example is the Fourier Transform function FT. FT takes as its input a list representation f of size N and outputs the Fourier representation $\hat{f}$ also of size N. The Fast Fourier Transform can be implemented as a circuit of size $N \log N$ known as the butterfly. Is this the smallest possible circuit for computing FT? The answer is not known.[1]

Another important example in linear algebra is the matrix INVerse. A circuit for INV takes the n^2 entries of a square matrix X as inputs and produces the n^2 entries of the matrix X^{-1} as outputs. (To allow inversion the circuit has the gate $[g](x) = 1/x$ at its disposal.) Calculating the matrix inverse is closely related to solving linear equations. With a bit of work Gaussian Elimination can be turned into a circuit for INV of similar complexity, namely $\Theta(n^3)$. Can we do better?

Computer scientist Volker Strassen answered this question affirmatively in 1969. He came up with a beautiful recursive algorithm for inverting n by n matrices using $\Theta(n^{\log 7}) \approx \Theta(n^{2.81})$ operations. Strassen's algorithm was the leading shot for what became a war of exponents. The current world record for matrix inversion is $\Theta(n^{2.371117})$.[2] These algorithms are complicated and not particularly practical. Yet there is no reason to doubt that further improvements and simplifications are in the cards.

What does this have to do with Backpropagation? To explain its relevance we need to talk about yet another problem: computing the DETerminant of an $n \times n$ matrix X. The determinant of a matrix X is some complicated polynomial in its n^2 entries. How hard is it to calculate?

Gaussian Elimination can also be used to calculate determinants using $\Theta(n^3)$ operations. This is not a coincidence: In his 1969 paper Strassen showed that calculating the determinant of a matrix can be "reduced" to inverting some related matrices. Thus the complexity of calculating the determinant is asymptotically bounded by the complexity of calculating the inverse. If anyone figures out how to invert $n \times n$ matrices in quadratic time—the holy grail of matrix algorithms—they will have also figured out how to calculate the determinant in quadratic time.

This does not sound terribly surprising. After all, the determinant of a matrix is just one number, while its inverse is a whole matrix with n^2 numbers. Surely calculating the determinant should be *easier* than calculating the inverse. It is not! If there is a size-s circuit for calculating the $n \times n$ matrix determinant out there, there will also be a size-$O(s)$ circuit for the inverse.

The reason for this is Backpropagation. By Cramer's rule, the entries of the inverse matrix can be calculated by dividing the determinants of the minors by the determinant of the matrix:

$$(i,j)\text{-th entry of } INV(X) = \frac{DET(X \text{ with } i\text{-th row and } j\text{-th column removed})}{DET(X)}.$$

[1] On October 6, 2026, OpenAI claimed to have discovered a circuit of size $O(N(\log N)^{1-10^{-13}})$. The jury is still out on it.
[2] OpenAI claimed $n^{2.25}$ on the same date. The jury is still out on it.

The determinant of X with the i-th row and j-th column removed is nothing but the partial derivative of $DET(X)$ with respect to variable X_{ij}:

$$(i,j)\text{-th entry of } INV(X) = \frac{\partial DET(X)/\partial X_{ij}}{DET(X)}.$$

Collecting all these entries into the matrix $INV(X)$ gives the beautiful formula

$$INV(X) = \frac{\nabla DET(X)}{DET(X)}$$

The circuit complexity of the INVerse is therefore at most that of the DETerminant plus that of its gradient (plus n^2 divisions). Theorem 3 tells us that if DET has a size-s circuit then ∇DET has a size-$O(s)$ circuit, and so does INV.[3]

[3]This reasoning assumes all circuit gates have *bounded fan-in*, namely they take a fixed number of arguments, guaranteeing that the number of wires is proportional to the number of nodes.