Natural Language Processing – Lecture 1

Lecturer: Ryan Cotterell, ETH Zurich

Arrangement: Tongyu Lu

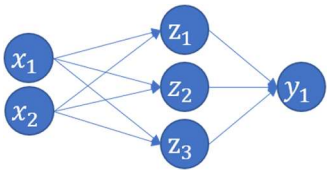
How to implement gradient-based optimization in almost any model? Backpropagation (BP). We may have been pretty familiar with using BP, but if you are asked to throw away PyTorch and implement it yourself, are you able to do that? If yes, then this lecture note will be trivial for you. If not, it is worth reading.

Backpropagation

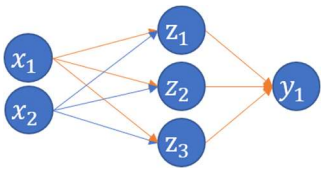
Computing Derivative in Computation Graph

Consider $y = y(z), z = z(x)$, then $\frac{\partial y_j}{\partial x_j} = \sum_{k=1}^{m} \frac{\partial y_j}{\partial z_k} \frac{\partial z_k}{\partial x_j}$

The forward computation from x to y could be represented as such (and dimensions are assumed as such in the graph):



We can see that $\frac{\partial y_1}{\partial x_1} = \frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial y_1}{\partial z_2}\frac{\partial z_2}{\partial x_1} + \frac{\partial y_1}{\partial z_3}\frac{\partial z_3}{\partial x_1}$ is the summation back of the three orange paths.

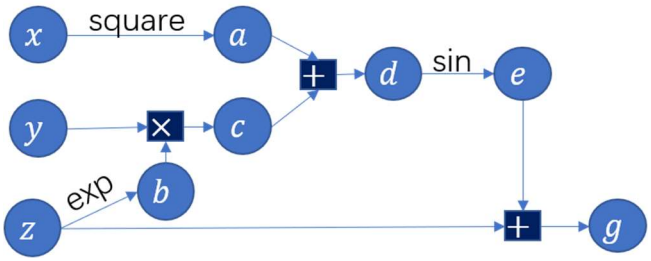


This is actually a general rule to use summation over paths to calculate derivatives, which is stated as the Bauer's formula. This will be stated very soon.

Before getting to the theory, let's have another example of practical chain rule. Consider a composite function:

$$f(x, y, z) = z + \sin(x^2 + y \exp(z))$$

It is easy to draw a computation graph for it:



Now, we focus on the form of this graph: it is a labelled directed acyclic graph (DAG). Actually, it is a **hypergraph**. Each node is a variable and each hyperedge is labeled with a function. What is a hypergraph and a hyperedge?

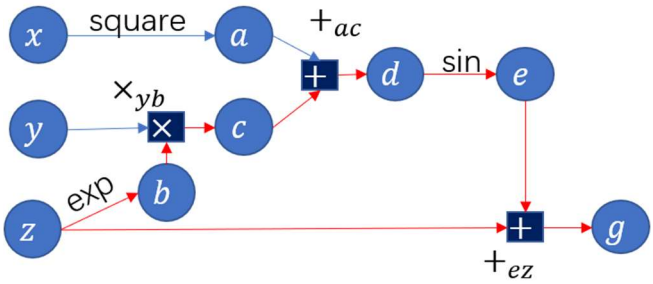
A hypergraph is a graph in which exists an edge (hyperedge) $e = ([V_1, V_2 \dots], V')$ that links a vertex V' to more than one vertices $[V_1, V_2 \dots]$. In the computation graph, such edges are operators which involves more than one argument, like addition and multiplication.

When we have a hyperedge, we could introduce an auxiliary node, like the $+$ or $\times$ nodes shown in the computation graph. And then, the hyperedges are transformed into a node with several normal edges. And the hypergraph is transformed into a graph.

Now, we want to do derivative on it. Say, we want to compute $\partial g/\partial z$. From the expression, it is not hard to do it:

$$\frac{\partial g}{\partial z} = 1 + y\exp(z)\cos(x^2 + y\exp(z))$$

We check if the "tracing all the way back" rule applies to this: we find two paths from g to



According to the graph,

$$\frac{\partial g}{\partial z} = \left[\frac{\partial g}{\partial +_{ez}}\frac{\partial +_{ez}}{\partial z}\right] + \left[\frac{\partial g}{\partial +_{ez}}\frac{\partial +_{ez}}{\partial e}\right]\frac{\partial e}{\partial d}\left[\frac{\partial d}{\partial +_{ac}}\frac{\partial +_{ac}}{\partial c}\right]\left[\frac{\partial c}{\partial \times_{yb}}\frac{\partial \times_{yb}}{\partial b}\right]\frac{\partial b}{\partial z}$$

We define $\left[\frac{\partial z}{\partial +_{xy}}\frac{\partial +_{xy}}{\partial x}\right] = \left[\frac{\partial z}{\partial +_{xy}}\frac{\partial +_{xy}}{\partial y}\right] = 1$, $\left[\frac{\partial z}{\partial \times_{xy}}\frac{\partial \times_{xy}}{\partial x}\right] = y$ and $\left[\frac{\partial z}{\partial \times_{xy}}\frac{\partial \times_{xy}}{\partial y}\right] = x$ (this definition is reasonable), we will have:

$$\frac{\partial g}{\partial z} = 1 + 1 \times \frac{\partial e}{\partial d} \times 1 \times y\frac{\partial b}{\partial z} = 1 + y\frac{\partial e}{\partial d}\frac{\partial b}{\partial z} = 1 + y\cos d \exp z$$

And this is exactly what we have got from the chain rule in mathematics.

Now, we are convinced to welcome the following theorem:

Theorem (Bauer's formula): given a function f, generate its computation graph $G = (V, E)$ (where auxiliary nodes for multi-argument operations are introduced and the corresponding derivative rules are defined), then given any $z_n, z_k \in V$, we have

$$\frac{\partial z_n}{\partial z_k} = \sum_{p \in \mathcal{P}(k,n)} \prod_{(i,j)\in p} \frac{\partial z_j}{\partial z_i}$$

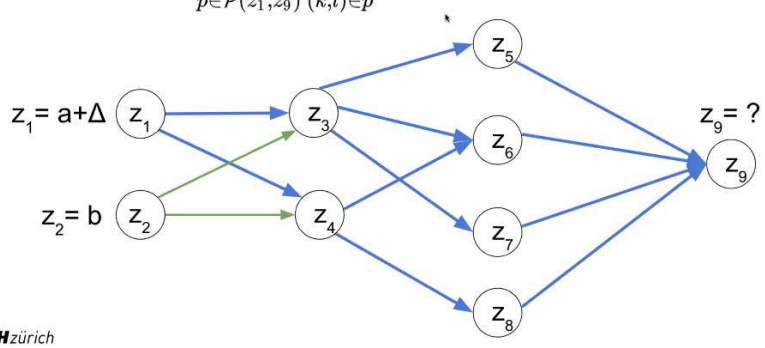
where $\mathcal{P}(k,n)$ is the set of all paths from z_k to z_n, and (i,j) is an individual edge from z_i to z_j.

No proof for Bauer's formula! To be simple, it is exactly the chain rule.

We call such paths $p \in \mathcal{P}(k,n)$ as Bauer paths. If we compute this formula naively, we will sum up exponential number of paths. As an example, in the following computation graph, the Bauer paths are highlighted in blue:

- The derivative is a sum over all of the Bauer paths:

$$\frac{\partial z_9}{\partial z_1} = \sum_{p \in \mathcal{P}(z_1, z_9)} \prod_{(k,l)\in p} \frac{\partial z_l}{\partial z_k}$$



ETH zürich

This graph is screenshot from the lecture slide. This shows that there is a total of $2 \times 4 \times 1$ paths. It could be imagined the number of paths is given by a series of multiplication. This is why we say that it is in exponential level.

Note that in this graph, the auxiliary vertices are removed, which is replaced by multiple edges. For example, there is a 2-argument computation from $[z_3, z_4]$ to z_6. In the computation hypergraph, it is a hyperedge. Now, it is represented as two edges.

The introduction of auxiliary vertices is just for clear notations in math. In the next section, a computation graph is treated as a hypergraph without auxiliary vertices, where each hyperedge $g_j: [\mathrm{Pa}(z_i)] \to z_i$ is labelled as a function from parent nodes of z_i to node z_i.

In lecture discussion: Bipartite graph? It is the graph which has nodes on the two parts, in which the nodes in each part are not linked directly by edges. Currying?

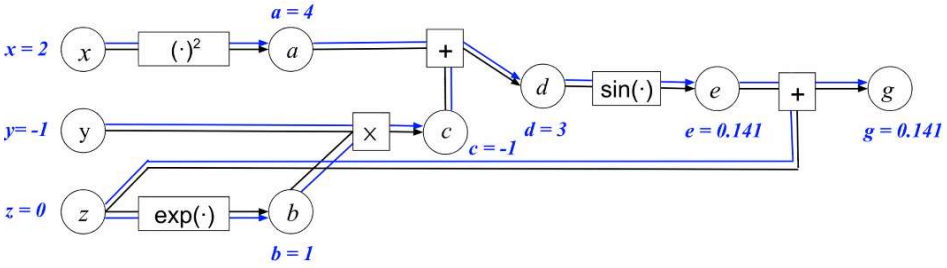
Reverse-mode Automatic Differentiation

Now, as we have known how to represent an arbitrary derivative in a computation graph, we want to then find an efficient way to calculate it.

Let's continue the example in the previous section:

$$f(x, y, z) = z + \sin(x^2 + y\exp(z))$$

Suppose we plug in an input and get an output, as screenshot in the following figure:



$$f(2, -1, 0) = g = 0.141$$

Complexity: $O(E)$

Now, try to write this process into an algorithm:

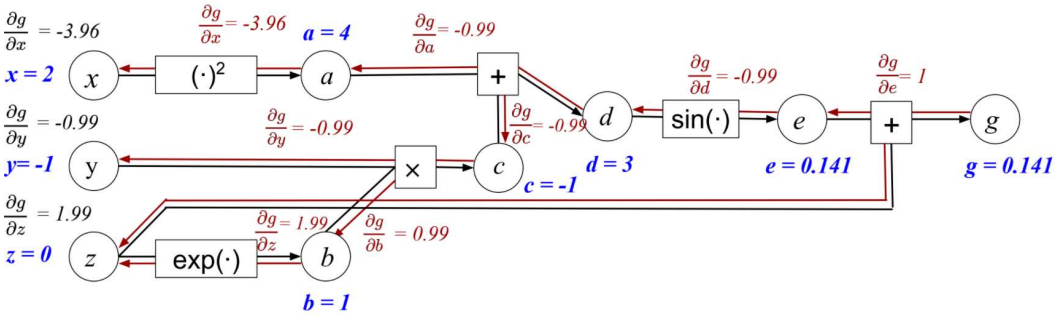
Algorithm: forward propagate

Input: function $f: \mathbb{R}^n \to \mathbb{R}^m$, input $x \in \mathbb{R}^n$

1. Create computation hypergraph, with N hyperedges. (As one hyperedge links one vertex in this DAG, the total number of vertices is $M = N + n$)
2. Initialize $z_i = \begin{cases} x_i, 1 \le i \le n \\ 0, n < i \le M \end{cases}$
3. Denote each hyperedge as $g_i: [\mathrm{Pa}(i)] \to z_i$ (notation $[\mathrm{Pa}(i)]$ means the ordered parent vertices of z_i)
4. for $i = n + 1, \dots, M$:
5. $\quad z_i \leftarrow g_i([\mathrm{Pa}(i)])$

Output: $[z_1, \dots, z_M]$

Then, we do derivative in that same example computation graph, and here is what we get:



$$\nabla f(2, -1, 0) = \langle \frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial z} \rangle = \langle -3.96, -0.99, 1.99 \rangle$$

Complexity: $O(E)$

We mimic the forward propagate algorithm, and try to write down the "back propagate" algorithm:

Algorithm: back propagate (BP)

Input: function $f: \mathbb{R}^n \to \mathbb{R}^m$, input $x \in \mathbb{R}^n$

1. $[z_1, \dots, z_M] = \mathrm{forward\_propagate}(f, x)$

2. Initialize $\frac{\partial f}{\partial z_i} = \begin{cases} 1, i = M \\ 0, i = 1, \dots, M-1 \end{cases}$
3. for $i = M-1, \dots, 1$:
4. $\frac{\partial f}{\partial z_i} \leftarrow \sum_{j \in \mathrm{Ch}(i)} \frac{\partial f}{\partial z_j} \frac{\partial}{\partial z_i} g_j([\mathrm{Pa}(j)])$ (notation $\mathrm{Ch}(i)$ means the child vertices of z_i. If $j \in \mathrm{Ch}(i)$, then $j > i$)

Output: $\left[\frac{\partial f}{\partial z_1}, \dots, \frac{\partial f}{\partial z_M}\right]$

We see that: although BP is implementation of chain rule, it is actually a linear-time dynamic program for computing derivatives. This is the art of dynamic programming: use storage to trade computation complexity.

From the algorithm introduced above, we say that BP is the "reverse-mode automatic differentiation". Are there other forms of automatic differentiation? Yes, there are. For example, forward-mode. But they are not so efficient as the reverse-mode. Therefore, they are left for literature reviews, in case you are interested.

Theorem: reverse-mode AD can compute gradient $\partial f / \partial x$ in the same time complexity as computing $f(x)$.
Actually, we have "informally" proved this by introducing the BP algorithm. We see that there is only a for-loop.

Theorem: implementing BP is equivalent to implementing Bauer's formula naively
Proof: we use math induction procedure. In the proof, we use notation $z_1, \dots, z_n$

- Base case: $\frac{\partial z_n}{\partial z_n} = 1$
- Inductive hypothesis: assume that for $k > m$, we have the local chain rule:

$$\frac{\partial z_n}{\partial z_k} = \sum_{j \in \mathrm{Ch}(k)} \frac{\partial z_j}{\partial z_k} \frac{\partial z_n}{\partial z_j}$$

- Inductive step: for $k = m$, we have

$$\frac{\partial z_n}{\partial z_k} = \sum_{p \in \mathcal{P}(k,n)} \prod_{(i,j) \in p} \frac{\partial z_j}{\partial z_i} = \sum_{p \in \mathcal{P}(k,n)} \frac{\partial z_l}{\partial z_k} \prod_{\substack{(i,j) \in p \\ j \neq l}} \frac{\partial z_j}{\partial z_i} = \sum_{l \in \mathrm{Ch}(k)} \frac{\partial z_l}{\partial z_k} \left(\sum_{p \in \mathcal{P}(l,n)} \prod_{(i,j) \in p} \frac{\partial z_j}{\partial z_i} \right) = \sum_{l \in \mathrm{Ch}(k)} \frac{\partial z_l}{\partial z_k} \frac{\partial z_n}{\partial z_l}$$

The first equality is Bauer's formula;
the second equality is by extracting the first factor in the multiplication;
the third equality is distributive property ($|\mathrm{Ch}(k)|$ terms in total);
the fourth equality is using the inductive hypothesis because $l > k = m$.

This finishes the proof. What we did in the BP algorithm is to iteratively calculate

$$\frac{\partial z_n}{\partial z_k} = \sum_{j \in \mathrm{Ch}(k)} \frac{\partial z_j}{\partial z_k} \frac{\partial z_n}{\partial z_j},$$

$$\frac{\partial z_n}{\partial z_j} = \sum_{j' \in \mathrm{Ch}(j)} \frac{\partial z_{j'}}{\partial z_j} \frac{\partial z_n}{\partial z_{j'}},$$

…

$$\frac{\partial z_n}{\partial z_?} = \frac{\partial z_n}{\partial z_?} \frac{\partial z_n}{\partial z_n}$$

And so forth, we approach back to $\frac{\partial z_n}{\partial z_n} = 1$, and calculate $\frac{\partial z_n}{\partial z_?} = \frac{\partial z_n}{\partial z_?} \frac{\partial z_n}{\partial z_n}$. Because we know how to calculate $\frac{\partial z_n}{\partial z_?}$ (this is done by $\frac{\partial z_n}{\partial z_?} = \frac{\partial}{\partial z_?} g_n\big(\mathrm{Pa}(z_n)\big)$, where g_n is the label of the n^{th} edge), we are successful in calculating $\frac{\partial z_n}{\partial z_k}$. If we trace all the way down and calculate the derivative recursively, we get the forward-mode automatic differentiation. But if we start

from $\frac{\partial z_n}{\partial z_n}$ and calculate $\frac{\partial z_n}{\partial z_?}$ one by one, we get the recursive-mode automatic differentiation, which is BP.

And from the proof, we could say that: BP is a kind of recursive rearrangement of the Bauer's formula.

The Role of BP

Differentiation in computer has a lot of forms.

- Form 1: symbolic differentiation. This is what MATLAB has done. But BP does not do this.
- Form 2: numerical differentiation: to calculate the differentiation using perturbation, getting the numerical result. BP does not do this, but can have the same result as this.
  A typical numerical differentiation is done by defining another function, e.g.
  $$\frac{\partial f}{\partial z}(x, y, z) = \frac{f(x, y, z + h) - f(x, y, z)}{h}$$
- Form 3: automatic differentiation: it calculates the numerical result, given a function input. It gets the same result as the numerical method, but approaches through computation graph. The function form is not obtained analytically, but as a graph.

  In other words, the computation hypergraph gives the function $\frac{\partial f}{\partial z}(x, y, z)$.

  And by plugging in an input (x, y, z), we have the numerical value.
  If we define a set of relationships between input arguments and the output, we establish a graph (which is equivalent to analytical form), even if we did not plug in any numerical inputs. The computer does not have to compute the analytical form of the derivatives, because the analytical form is exactly the graph structure (in backward propagation).

Exercise: implement a code for BP.