CSE 446: Machine Learning **Lecture**

Auto-Differentiation and Computation Graphs

Instructor: Sham Kakade

1 Auto-Differentiation (AD) in applied ML

The ability to automatically differentiate functions has recently become a core ML tool, providing us with ability to experiment with much richer models in the development cycle. One impressive (and remarkable!) mathematical result is that we can compute *all* of the partial derivatives of a function with (nearly) same computational runtime (within a factor of 5) of the function itself [Griewank(1989), Baur and Strassen(1983)].

Understanding the details of how auto-diff works is an important component in our ability to better utilize software like PyTorch, TensorFlow, etc...

2 The Computational Model

Suppose we seek to compute the derivative with respect to a real valued function $f(w) : \mathbb{R}^d \to \mathbb{R}$, i.e we seek to compute $\nabla_w f(w)$. The critical question: what is the time complexity of computing this derivative, particularly in the case where d is large?

First, let us state how specify the function f through a program. This model is (essentially) the algebraic complexity model.

2.1 An example

(This example is adapted from [Griewank and Walther(2008)]).

Let us start with an example: suppose we are interested in computing the function:

$$f(w_1, w_2) = (\sin(2\pi w_1/w_2) + 3w_1/w_2 - \exp(2w_2)) * (3w_1/w_2 - \exp(2w_2))$$

Let us now state a program which computes our function f.

input: $z_0 = (w_1, w_2)$

1. $z_1 = w_1/w_2$
2. $z_2 = \sin(2\pi z_1)$
3. $z_3 = \exp(2w_2)$
4. $z_4 = 3z_1 - z_3$
5. $z_5 = z_2 + z_4$

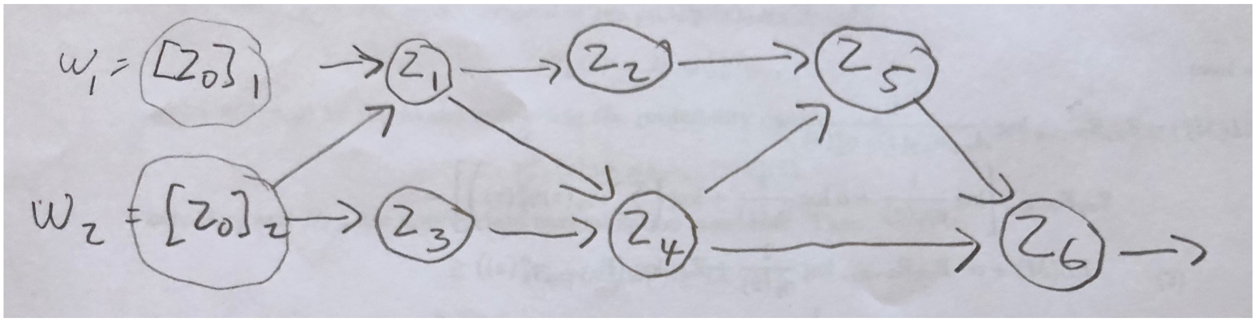

Figure 1: Computational graph of our example.

6. $z_6 = z_4 z_5$

return: z_6

Our "program" is sometimes referred to as an *evaluation trace*, when written in this manner. The computation graph is the flow of operations. Figure 1 shows this graph for our example; here z_1 points to z_2 and z_4; z_2 points to z_5; z_4 points to z_5 and z_6; etc. We say that z_2 and z_4 and children of z_1; z_5 is a child of z_2; etc.

2.2 The computation graph and evaluation traces

Now let us specify the model more abstractly. Suppose we have access to a set of differentiable real value functions $h \in \mathcal{H}$.

The computational model is one where we use our functions in $\mathcal{H}$ to create intermediate variables. Specifically, our evaluation trace will be of the form:

In our computational model, all intermediate variables $z_1, \ldots z_t$ will be scalars; each of the variables correspond to a node. *Only the input $z_0 = w$ will be a vector, and this input will be represented with d-nodes $[z_0]_1 = w_1, [z_0]_2 = w_2, \ldots [z_0]_3 = w_d$ in our graph.*

input: $z_0 = w$. We actually have d (scalar) input nodes where $[z_0]_1 = w_1, [z_0]_2 = w_2, \ldots [z_0]_3 = w_d$.

1. $z_1 = h_1$(a fixed subset of the parent variables in w)
   ...

t. $z_t = h_t$(a fixed a subset of the parent variables in $z_{1:t-1}, w$)
   ...

T. $z_T = h_T$(a fixed a subset of the parent variables in $z_{1:T-1}, w$)

return: z_T.

Let us say every $h \in \mathcal{H}$ is one of the following:

1. an affine transformation of the inputs (e.g. step 4 in our example)
2. a product of variables, to some power (e.g. step 1, step 6 in our example. we could also have $z_8 = z_1^4 * z_5^7 z_6^{-1}$).

3. h could lie in some fixed set of one dimensional differentiable functions. Examples include $\sin(\cdot)$, $\cos(\cdot)$, $\exp(\cdot)$, $\log(\cdot)$, etc. Implicitly, we are assuming that we can "easily" compute the derivatives for each of these one dimensional functions h (we specify this precisely later on). For example, we could have $z_8 = \sin(2z_3)$. We do not allow $z_8 = \sin(2z_3 + 7z_5 + z_6)$; for the latter, we would have to create another intermediate variable for $2z_3 + 7z_5 + z_6$. This restriction is to make our computations as efficient as possible.

Remark: We don't really need the functions of type 3. In a very real sense, all our transcendental functions like $\sin(\cdot)$, $\cos(\cdot)$, $\exp(\cdot)$, $\log(\cdot)$, etc. are all implemented (in code) through using functions of type 1 and 2, e.g. when you call the $\sin(\cdot)$ function, it is computed through a polynomial.

Relation to Backprop and a Neural Net: In the special case of neural nets, note that our computation graph should not be thought of as being the same as the neural net graph. With regards to the computation graph, the input nodes are w. In a neural net, we often think of the input as x. Note that for neural nets which are not simple MLPS (say you have skip connections or one which is more generally a DAG), then there are multiple ways ot execute the computation, giving rise to different computational graphs, and this order is relevant in how we execute the reverse mode.

3 The Reverse Mode of Automatic Differentiation

The key in in understanding auto-diff is understanding the chain rule, where the insight is that z_t effects the target function only through its functional dependence on its children. This will allow us to work backwards. Clearly,

$$\frac{dz_T}{dz_T} = 1\,.$$

Now we will proceed recursively by computing the $\frac{dz_T}{dz_t}$ given the derivatives of z_T with respect to the children of t. Specifically, we make use of the chain rule:

$$\frac{dz_T}{dz_t} = \sum_{c \text{ is a child of } t} \frac{dz_T}{dz_c}\frac{\partial z_c}{\partial z_t} \tag{1}$$

where the sum is over all children of t. Here, a *child* is a node in the computation graph which z_t directly points to.

A note on the chain rule. The subtle point in understanding auto-diff is understanding the chain rule due to that all z_t are dependent variables on $z_{1:t-1}$ and w. It is helpful to think of z_T as a function of both a single grandparent z_t along with w as follows (slightly, abusing notation):

$$z_T = z_T(w, z_t)$$

where think of z_t as a free variable. In particular, this means we think of z_T as being computed by following the evaluation trace (our program) *except* that at node t it uses the value z_t; this node ignores its inputs and is "free" to use another value z_t instead. In this sense, we think of z_t as a free variable (not depending on w or on any of its parents). We will be interested in computing the derivatives (again, slightly abusing notation):

$$\frac{dz_T}{dz_t} := \frac{dz_T(w, z_t)}{dz_t}$$

for all the variables z_t.

The "reverse mode" of AD. The reverse mode of AD algorithm is as follows.

1. Compute $f(w)$ and store in memory all the intermediate variables $z_{0:T}$.

2. Initialize:
$$\frac{dz_T}{dz_T} = 1$$

3. Proceeding recursively, starting at $t = T - 1$ and going to $t = 0$
$$\frac{dz_T}{dz_t} = \sum_{c \text{ is a child of } t} \frac{dz_T}{dz_c} \frac{\partial z_c}{\partial z_t}$$

4. return $\frac{dz_T}{dz_0} = \frac{df}{dw}$

Note that $\frac{dz_T}{dz_0} = \frac{df}{dw}$ by the definition of z_T and z_0.

4 Time complexity

The following theorem, referred to as the Baur-Strassen theorem, states that the computational runtime for computing all d-partial derivatives of a scalar function $f(w)$. is nearly the same as computing the function itself.

Theorem 4.1. *([Baur and Strassen(1983), Griewank(1989)]) Assume that every $h(\cdot)$ is specified as in our computational model (with the aforementioned restrictions). Furthermore , for $h(\cdot)$ of type 3, let us assume that we can compute the derivative of $h'(z)$ in time that is within a factor of 5 of computing $h(z)$ itself. Using a given evaluation trace, let T be the time it takes to compute $f(w)$ at some input w, then the reverse mode computes both $f(w)$ and $\frac{df}{dw}$ in time $5T$. In other words, we compute all d partial derivatives of f in essentially the same time as computing f itself.*

Proof. First, let us show the algorithm is correct. The equation to compute $\frac{dz_T}{dz_t}$ follows from the chain rule. Furthermore, based on the order of operations, at (backward) iteration t, we have already computed $\frac{dz_T}{dz_c}$ for all children c of t. Now let us observe that we can compute $\frac{\partial z_c}{\partial z_t}$ using the variables stored in memory. To see this, consider our three cases (and let us observe the computational cost as well):

1. If h is affine, the derivative is simply the coefficient of z_t.
2. If h is a product of terms (possibly with divisions), then $\frac{\partial z_c}{\partial z_t} = z_c(\alpha/z_t)$, where alpha is the power of z_t. For example, for $z_5 = z_2 z_4^2$ we have that $\frac{\partial z_5}{\partial z_4} = z_5 * (2/z_4)$.
3. If $z_c = h(z_t)$ (so it is a one dim function of just *one* variable), then $\frac{\partial z_c}{\partial z_t} = h'(z_t)$.

Hence, the algorithm is correct, and the derivates are computable using what we have stored in memory.

Now let us verify the claimed time complexity. The compute time T for $f(w)$ is simply the sum of times required to compute z_1 to z_T. We will relate this time to the time complexity of the reverse mode. In the reverse mode, note that since $\frac{\partial z_c}{\partial z_t}$ is used precisely once: it is computed when we hit node t. Now let us show that the compute time of z_c and the compute time for computing *all* the derivatives $\{\frac{\partial z_c}{\partial z_t} : t \text{ which are parents of } c\}$ are of the same order. If z_c is an affine function of its parents — suppose there are M parents — then z_c takes time $O(M)$ time to compute and computing all the partial derivatives also takes $O(M)$ in total: each $\frac{\partial z_c}{\partial z_t}$ is $O(1)$ (since the derivative is just a constant) there are M such derivatives. A similar argument can be made for case 2. For case 3, computing $\frac{\partial z_c}{\partial z_t}$ (for the only parent t) is the same order as computing z_c by assumption. Hence, we have show that computing z_c and computing *all* the derivatives $\{\frac{\partial z_c}{\partial z_t} : t \text{ which are parents of } c\}$ are of the same order. This accounts for all the computation required to compute all the $\frac{\partial z_c}{\partial z_t}$'s. It is now straightforward to see that the remaining computation of all the $\frac{dz_T}{dz_t}$'s using these partial derivatives, is also of order T, since each $\frac{\partial z_c}{\partial z_t}$ occurs just once in some sum.

The factor of 5 is simply more careful book-keeping of the costs. $\square$

References

[Griewank(1989)] Andreas Griewank. On automatic differentiation. In *IN MATHEMATICAL PROGRAMMING: RECENT DEVELOPMENTS AND APPLICATIONS*, pages 83–108. Kluwer Academic Publishers, 1989.

[Baur and Strassen(1983)] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[Griewank and Walther(2008)] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2008.