CSE 446: Machine Learning **Lecture**

Dynamic Computation Graphs

Instructor: Sham Kakade

1 Building the evaluation trace on the fly...

We provided a natural computational model to compute derivatives. In practice, we do not write code in a manner that is identical to the computational model. The question is how do we build this computation trace? And how generally the reverse mode of AD be applied?

Rather than formally defining when we can obtain correct derivatives in a more general mode (this itself is not obvious), we wil go through three examples to see how this is done in practice and why we need to 'dynamically' build evaluation traces.

Importantly, note that our computational model for AD, along with the proofs of correctness, does not permit branching and for loops. We will not look at reductions to construct evaluation traces when we do permit for loops and branching, and we will comment when such procedures are provably correct.

1.1 Example 1: "Pre-compiling" the evaluation trace

Suppose w is real valued and our program to compute $f(w)$ is:

0. **Input:** w
1. Initialize: $y = w$
2. For $i = \{1, 2, 3\}$ do
   $y \leftarrow y + i$
3. **Return:** y

Note that this program only uses intermediate variables, though it keeps overwriting this variable. This overwriting, and the For loop, means it is not written in the form of an evaluation trace. So we are not able to run the reverse mode 'automatically' when written in this form.

Instead, it is straightforward to see that this code can 'automatically' (say by a compiler) be written to be:

0. Initialize: $z_0 = w$
1. $z_1 = z_0 + 1$
2. $z_2 = z_1 + 2$
3. $z_3 = z_2 + 3$
4. **Return:** z_3

The above is a valid evaluation trace and the reverse mode can be applied to compute $\frac{df}{dw}$.

1.2 Example 2: Runtime compilation of the evaluation trace

Now, let us consider our common setting where we have loss function on a data point, e.g. $\ell(w, (x, y))$. Here, we are often interested in computing $\frac{d\ell(w,(x,y))}{dw}$, e.g. as is the case for SGD. Importantly, note that we regard $\ell(w, (x, y))$ *only* as a function of w when thinking about differentiation. There are implications of this for dealing with computation graphs, which we now consider.

Suppose we are dealing with an RNN, or a setting where we view x and y as a time series. The length of x and y may not be fixed in advanced; the length of x could be the length of our input sentence (and the length of y the length of our output sentence or output sequence of labels). Another example could be where x is an image of variable size. The key property these two examples share is that we are not able to define an evaluation trace until we obtain our input (x, y), as this defines the function $\ell(w, (x, y))$.

Let us consider a simple example of this for a function $f(w, N)$ where w is a real number and N is an integer. For example:

0. **Input:** w, N
1. Initialize: $y = w$
2. For $i = \{1, 2, \dots N\}$ do
   $y \leftarrow y + i$
3. **Return:** y

Note that here we are not able to determine the evaluation trace by *until* we actually obtain N, which is provided at runtime. Hence, in this case, the evaluation trace will be built at runtime when our auto-diff method (like PyTorch) 'watches' our code being executed.

In this case, the trace is:

0. Initialize: $z_0 = w$
1. $z_1 = z_0 + 1$
2. $z_2 = z_1 + 2$

...

3. $z_N = z_{N-1} + N$
4. **Return:** z_N

The following corollary immediately follows from our previous theorem. This is one case where branching is clearly correct and in which we build the trace "on the fly".

Corollary 1.1. *Assume $\ell(w, (x, y))$ is differentiable with respect to w for all possible inputs input (x, y). Suppose that for any given input (x, y), that we build a valid evaluation trace that computes $\ell(w, (x, y))$ (under our previous assumption). Then the Reverse Mode of AD correctly computes $\nabla\ell(w, (x, y))$ (and it computes this within a constant factor of the time it takes to compute $\ell(w, (x, y))$).*

1.3 Example 3: another (and more subtle) dynamic example

The last example, and most subtle one, is where we our program actually branches based on w itself. It is not clear what the guarantees we have are in this case nor what the correct answer should be for continuous functions.

Let us consider an example using a ReLu function, e.g. $f(w) = \max\{w, 0\} + 5$ where w is a scalar. One program for this is:

0. **Input:** w
1. If $w > 0$,
   Return: $w + 5$
2. If $w \leq 0$,
   Return: 5

Here, we can build an evaluation trace based on which branch is utilized at runtime. For example, if our input $w > 0$, at runtime, the evaluation trace on the branch taken would be:

0. Initialize: $z_0 = w$
1. $z_1 = w + 5$
2. **Return:** z_1

and if our input $w \leq 0$, at runtime, the evaluation trace on the branch taken would be:

0. Initialize: $z_0 = w$
1. $z_1 = 5$
2. **Return:** z_1

We can run the reverse mode of AD on each of these evaluation traces. In this particular case, we obtain a valid sub-gradient. What occurs more generally is less evident.

2 TensorFlow, PyTorch, and a Few Comments

The "pre-compilation" (before runtime) method of building the evaluation trace is what is fundamental to TensorFlow. In TensorFlow, when we write our code, we are actually building a computation graph ourselves; the semantics of TensorFlow is that we explicitly define the computation graph ourselves (which sometimes makes debugging confusing). The claimed advantage of this is that it may be possible to obtain runtime speed ups with a smart compiler that builds the trace in advance. The extent to which this happens in practice is less clear.

In PyTorch, we do not explicitly specify the computation graph. We write our code and PyTorch builds the graph at runtime, e.g. "on the fly". This means it is closer to running the true reverse mode of AD.

For the case in which our function is not differentiable, it is unclear what correctness claims are provided. In the non-convex case, we may hope to obtain a valid Clarke subdifferential.

2.1 Numpy and AutoGrad

Many of these numerical AD software are built on the AutoGrad library. Think of AutoGrad as provide automatic differentiation for NUMPY. However, for the differentiation many of the linear algebra primitives, this is not done with the reverse mode of AD (to some extent, this is because solutions of polynomial systems do live in our computational model). Instead, one can actually symbolically take derivatives of the decompositions like SVD, LU, QR. At first glance, doing this seem subtle though there is a body of tools to do this symbolically. See [**?**]. Note that the derivatives may not always exist of some of these decompositions (at some points).