7 Automatic Differentiation (AutoDiff)

Goal

Forward and reverse mode auto-differentiation.

Alert 7.1: Convention

Gray boxes are not required hence can be omitted for unenthusiastic readers.

This note is likely to be updated again soon.

Definition 7.2: Function Superposition and Computational Graph (Bauer 1974)

Let $\mathcal{BF}$ be a class of basic functions. A (vector-valued) function $g : \mathsf{X} \subseteq \mathbb{R}^d \to \mathbb{R}^m$ is a superposition of the basic class $\mathcal{BF}$ if the following is satisfied:

- There exist some DAG $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ where using topological sorting we arrange the nodes as follows:

$$\underbrace{v_1, \ldots, v_d}_{\text{input}}, \quad \underbrace{v_{d+1}, \ldots, v_{d+k}}_{\text{intermediate variables}}, \quad \underbrace{v_{d+k+1}, \ldots, v_{d+k+m}}_{\text{output}}, \qquad \text{and } (v_i, v_j) \in \mathscr{E} \implies i < j.$$

  Here we implicitly assume the outputs of the function g do not depend on each other. If they do, we need only specify the indices of the output nodes accordingly (i.e. they may not all appear in the end).

- For each node v_i, let $\mathscr{I}_i := \{u \in \mathscr{V} : (u, v_i) \in \mathscr{E}\}$ and $\mathscr{O}_i := \{u \in \mathscr{V} : (v_i, u) \in \mathscr{E}\}$ denote the (immediate) predecessors and successors of v_i, respectively. Clearly, $\mathscr{I}_i = \emptyset$ if $i \le d$ (i.e. input nodes) and $\mathscr{O}_i = \emptyset$ if $i > d + k$ (i.e. output nodes).

- The nodes are computed as follows: sequentially for $i = 1, \ldots, d + k + m$,

$$v_i = \begin{cases} x_i, & i \le d \\ f_i(\mathscr{I}_i), & i > d \end{cases}, \quad \text{where} \quad f_i \in \mathcal{BF}. \tag{7.1}$$

Our definition of superposition closely resembles the computational graph of Bauer (1974), who attributed the idea to Kantorovich (1957).

Bauer, F. L. (1974). "Computational Graphs and Rounding Error". *SIAM Journal on Numerical Analysis*, vol. 11, no. 1, pp. 87–96.

Kantorovich, L. V. (1957). "On a system of mathematical symbols, convenient for electronic computer operations". *Soviet Mathematics Doklady*, vol. 113, no. 4, pp. 738–741.

Exercise 7.3: Neural Networks as Function Superposition

Let $\mathcal{BF} = \{+, \times, \sigma, \text{constant}\}$. Prove that any multi-layer NN is a superposition of the basic class $\mathcal{BF}$.

Is exp a superposition of the basic class above?

Theorem 7.4: Automatic Differentiation (e.g. Kim et al. 1984)

Let $\mathcal{BF}$ be a basic class of differentiable functions that includes $+, \times$, and all constants. Denote $T(f)$ as the complexity of computing the function f and $T(f, \nabla f)$ the complexity with additional computation of the gradient. Let $\mathscr{I}_f$ and $\mathscr{O}_f$ be the input and output arguments and assume there exists some constant

$C = C(\mathcal{BF}) > 0$ so that

$$\forall f \in \mathcal{BF}, \quad T(f, \nabla f) + |\mathscr{I}_f||\mathscr{O}_f|[T(+) + T(\times) + T(\text{constant})] \leq C \cdot T(f).$$

Then, for any superposition $g : \mathbb{R}^d \to \mathbb{R}^m$ of the basic class $\mathcal{BF}$, we have

$$T(g, \nabla g) \leq C\gamma(m \wedge d) \cdot T(g),$$

where γ is the maximal output dimension of basic functions used to superpose g.

Proof. Applying the chain rule to the recursive formula (7.1) it is clear that any superposition g is differentiable too. We split the proof into two parts: a forward mode and a backward mode.

Forward mode: Let us define the block matrix $V = [V_1, \ldots, V_d, V_{d+1}, \ldots, V_{d+k}, V_{d+k+1}, \ldots, V_{d+k+m}] \in \mathbb{R}^{d \times \sum_i d_i}$, where each column block V_i corresponds to the gradient $\nabla v_i = \frac{\partial v_i}{\partial \mathbf{x}} \in \mathbb{R}^{d \times d_i}$, where d_i is the output dimension of node v_i (typically 1). By definition of the input nodes we have

$$V_i = \mathbf{e}_i, \quad i = 1, \ldots, d,$$

where $\mathbf{e}_i$ is the standard basis vector in $\mathbb{R}^d$. Using the recursive formula (7.1) and chain rule we have

$$V_i = \sum_{j \in \mathscr{I}_i} V_j \cdot \nabla_j f_i, \quad \text{where} \quad \nabla_j f_i = \frac{\partial f_i}{\partial v_j} \in \mathbb{R}^{d_j \times d_i}.$$

In essence, by differentiating at each node, we obtained a square and sparse system of linear equations, where $\nabla_j f_i$ are known coefficients and V_i are unknown variables. Solving the linear system yields $V_{d+k+1}, \ldots, V_{d+k+m}$, the desired gradient of g. Thanks to the topological ordering, we can simply solve V_i one by one. Let $\gamma = \max_i d_i$ be the maximum output dimension of any node. We bound the complexity of the forward mode as follows:

$$\begin{aligned} T(g, \nabla g) &\leq \sum_{i \in \mathscr{V}} T(f_i, \nabla f_i) + \sum_{j \in \mathscr{I}_i} d d_i d_j [T(+) + T(\times) + T(\text{constant})] \\ &\leq d\gamma \sum_{i \in \mathscr{V}} T(f_i, \nabla f_i) + |\mathscr{I}_{f_i}||\mathscr{O}_{f_i}|[T(+) + T(\times) + T(\text{constant})] \leq d\gamma C T(g). \end{aligned}$$

Reverse mode: Let us rename the outputs $y_i = v_{d+k+i}$ for $i = 1, \ldots, m$. Similarly we define the block matrix $V = [V_1; \ldots; V_d; V_{d+1}; \ldots; V_{d+k}; V_{d+k+1}; \ldots; V_{d+k+m}] \in \mathbb{R}^{\sum_i d_i \times m}$, where each row block V_i corresponds to the transpose of the gradient $\nabla v_i = \frac{\partial \mathbf{y}}{\partial v_i} \in \mathbb{R}^{m \times d_i}$, where d_i is the output dimension of node v_i (typically 1). By definition of the output nodes we have

$$V_{d+k+i} = \mathbf{e}_i, \quad i = 1, \ldots, m, \quad \mathbf{e}_i \in \mathbb{R}^{1 \times m}.$$

Using the recursive formula (7.1) and chain rule we have

$$V_i = \sum_{j \in \mathscr{O}_i} \nabla_i f_j \cdot V_j, \quad \text{where} \quad \nabla_i f_j = \frac{\partial f_j}{\partial v_i} \in \mathbb{R}^{d_i \times d_j}.$$

Again, by differentiating at each node we obtained a square and sparse system of linear equations, where $\nabla_i f_j$ are known coefficients and V_i are unknown variables. Solving the linear system yields $V_1, \ldots, V_d$, the desired gradient of g. Thanks to the topological ordering, we can simply solve V_i one by one backwards, after a forward pass to get the function values at each node. Similar as the forward mode, we can bound the complexity as $m\gamma C T(g)$. □

Thus, surprisingly, for real-valued superpositions ($m = \gamma = 1$), computing the gradient, which is a $d \times 1$ vector, costs at most constant times that of the function value (which is a scalar), if we operate in the reverse mode! The common misconception is that the gradient has size $d \times 1$ hence if we compute one component at a time we end up d times slower. This is wrong, because we can recycle computations. Note also that even reading the input already costs $O(d)$. However, this time complexity gain, as compared to that of the forward mode, is achieved through a space complexity tradeoff: in reverse mode we need a forward pass first to collect and store all function values at each node, whereas in the forward mode these function values can be computed on the fly.

Kim, K. V., Yuri E. Nesterov, and B. V. Cherkasskii (1984). "An estimate of the effort in computing the gradient". *Soviet Mathematics Doklady*, vol. 29, no. 2, pp. 384–387.

Algorithm 7.5: Automatic Differentiation (AD) Pesudocode

We summarize the forward and reverse algorithms below. Note that to compute the gradient-vector multiplication $\nabla g \cdot \mathbf{w}$ for some compatible vector $\mathbf{w}$, we can use the forward mode and initialize V_i with $\mathbf{w}$. Similarly, to compute $\mathbf{w} \cdot \nabla g$, we can use the reverse mode with proper initialization to V_{d+k+i}.

Algorithm: Forward Automatic Differentiation for Superposition.

```
Input: $\mathbf{x} \in \mathbb{R}^d$, basic function class $\mathcal{BF}$, computational graph $\mathscr{G}$
Output: gradient $[V_{d+k+1}, \ldots, V_{d+k+m}] \in \mathbb{R}^{d\times m}$
1 for $i = 1, \ldots, d$ do                    // forward: initialize function values and derivatives
2     $v_i \leftarrow x_i$
3     $V_i \leftarrow \mathbf{e}_i \in \mathbb{R}^{d\times 1}$
4 for $i = d+1, \ldots, d+k+m$ do              // forward: accumulate function values and derivatives
5     compute $v_i \leftarrow f_i(\mathscr{I}_i)$
6     for $j \in \mathscr{I}_i$ do
7         compute partial derivatives $\nabla_j f_i(\mathscr{I}_i)$
8     $V_i \leftarrow \sum_{j\in\mathscr{I}_i} V_j \cdot \nabla_j f_i$
```

Algorithm: Reverse Automatic Differentiation for Superposition.

```
Input: $\mathbf{x} \in \mathbb{R}^d$, basic function class $\mathcal{BF}$, computational graph $\mathscr{G}$
Output: gradient $[V_1; \ldots; V_d] \in \mathbb{R}^{d\times m}$
1 for $i = 1, \ldots, d$ do                    // backward: initialize function values and derivatives
2     $v_i \leftarrow x_i$
3     $V_{d+k+i} \leftarrow \mathbf{e}_i \in \mathbb{R}^{1\times m}$
4 for $i = d+1, \ldots, d+k+m$ do                           // forward: accumulate function values
5     compute $v_i \leftarrow f_i(\mathscr{I}_i)$
6 for $i = d+k, \ldots, 1$ do                  // backward: accumulate function values and derivatives
7     $V_i \leftarrow \sum_{j\in\mathscr{O}_i} \nabla_i f_j \cdot V_j$
```

We remark that, as suggested by Wolfe (1982), one effective way to test AD (or manually programmed derivatives) and locate potential errors is through the classic finite difference approximation.

Wolfe, Philip (1982). "Checking the Calculation of Gradients". *ACM Transactions on Mathematical Software*, vol. 8, no. 4, pp. 337–343.

Exercise 7.6: Matrix multiplication

To understand the difference between forward-mode and backward-mode differentiation, let us consider the simple matrix multiplication problem: Let $A_\ell \in \mathbb{R}^{d_\ell \times d_{\ell+1}}, \ell = 1, \ldots, L$, where $d_1 = d$ and $d_{L+1} = m$. We are interested in computing

$$A = \prod_{\ell=1}^{L} A_\ell.$$

- What is the complexity if we multiply from left to right (i.e. $\ell = 1, 2, \ldots, L$)?
- What is the complexity if we multiply from right to left (i.e. $\ell = L, L-1, \ldots, 1$)?
- What is the optimal way to compute the product?

Remark 7.7: Further insights on AD

If we associate an edge weight $w_{ij} = \frac{\partial v_j}{\partial v_i}$ to $(i, j) \in \mathscr{E}$, then the desired gradient

$$\frac{\partial g_i}{\partial x_j} = \sum_{\text{path } P: v_j \to v_i} \prod_{e \in P} w_e. \tag{7.2}$$

However, we cannot compute the above naively, as the number of paths in a DAG can grow exponentially quickly with the depth. The forward and reverse modes in the proof of Theorem 7.4 correspond to two dynamic programming solutions. (Incidentally, this is exactly how one computes the graph kernel too.)

Naumann (2008) showed that finding the optimal way to compute (7.2) is NP-hard.

Naumann, Uwe (2008). "Optimal Jacobian accumulation is NP-complete". *Mathematical Programming*, vol. 112, no. 2, pp. 427–441.

Remark 7.8: Tightness of dimension dependence in AD (e.g. Griewank 2012)

The dimensional dependence $m \wedge d$ cannot be reduce in general. Indeed, consider the simple function $\mathbf{f}(\mathbf{x}) = \sin(\mathbf{w}^\top \mathbf{x})\mathbf{b}$, where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{b} \in \mathbb{R}^m$. Computing $\mathbf{f}$ clearly costs $O(d + m)$ (assuming sin can be evaluated in $O(1)$) while even outputting the gradient costs $O(dm)$.

Griewank, Andreas (2012). "Who Invented the Reverse Mode of Differentiation?" *Documenta Mathematica*, vol. Extra Volume ISMP, pp. 389–400.

Exercise 7.9: Backpropogation (e.g. Rumelhart et al. 1986)

Apply Theorem 7.4 to multi-layer NNs and recover the celebrated backpropogation algorithm. Distinguish two cases:

- Fix the network weights $W_1, \ldots, W_L$ and compute the derivative w.r.t. the input $\mathbf{x}$ of the network. This is useful for constructing adversarial examples.
- Fix the input $\mathbf{x}$ of the network and compute the derivative w.r.t. the network weights $W_1, \ldots, W_L$. This is useful for training the network.

Suppose we know how to compute the derivatives of $f(x, y)$. Explain how to compute the derivative of $f(x, x)$?

- Generalize from above to derive the backpropogation rule for convolutional neural nets (CNN).
- Generalize from above to derive the backpropogation rule for recurrent neural nets (RNN).

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning representations by back-propagating errors". *Nature*, vol. 323, pp. 533–536.

Remark 7.10: Fast computation of other derivatives (Kim et al. 1984)

Kim et al. (1984) pointed out an important observation, namely that the proof of Theorem 7.4 only uses the chain-rule property of differentiation:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial x}{\partial y}.$$

In other words, we could replace differentiation with any other operation that respects the chain rule and obtain the same efficient procedure for computation. For instance, the relative differential in numerical analysis or the directional derivative can both be efficiently computed in the same way. Similarly, one can compute the Hessian-vector multiplication efficiently as it also respects the chain rule (Møoller 1993; Pearlmutter 1994).

Kim, K. V., Yuri E. Nesterov, V. A. Skokov, and B. V. Cherkasskii (1984). "An efficient algorithm for computing derivatives and extremal problems". *Ekonomika i matematicheskie metody*, vol. 20, no. 2, pp. 309–318.

Møoller, M. (1993). *Exact Calculation of the Product of the Hessian Matrix of Feed-Forward Network Error Functions and a Vector in $O(N)$ Time*. Tech. rep. DAIMI Report Series, 22(432).

Pearlmutter, Barak A. (1994). "Fast Exact Multiplication by the Hessian". *Neural Computation*, vol. 6, no. 1, pp. 147–160.