Homework 4
APPM 4720/5720 Scientific Machine Learning, Fall 2024

Due date: Friday, Sep. 20 '24, before midnight, via Gradescope

Instructor: Prof. Becker

Revision date: 9/18/2024

Theme: Automatic Differentiation

Instructions Collaboration with your fellow students is OK and in fact recommended, although direct copying is not allowed. The internet **is allowed for basic tasks** (e.g., looking up definitions on wikipedia, looking at documentation, looking at basic tutorials) but it is not permissible to search for solutions to the exact problem or to *post* requests for help on forums such as `http://math.stackexchange.com/`.

Background If $F : \mathbb{R}^n \to \mathbb{R}^m$, we can write $F(\boldsymbol{x}) = (F_i(\boldsymbol{x}))_{i=1}^m$ for component functions $F_i : \mathbb{R}^n \to \mathbb{R}$, and we define the Jacobian J_F to be the $m \times n$ matrix of partial derivatives, so that the (i, j) entry of J_F is $\frac{\partial F_i}{\partial x_j}(\boldsymbol{x})$. *Note that if $m = 1$ then the Jacobian is just the transpose of the gradient.*
For multivariate functions, because derivatives (i.e., the Jacobian) are matrices and because matrix multiplication does not commute, we have to be careful with the order we write the chain rule in. The correct order is:

$$J_{f\circ g}(\boldsymbol{x}) = J_f(g(\boldsymbol{x})) \cdot J_g(\boldsymbol{x}).$$

Problem 1: We'll explore forward-mode and reverse-mode automatic differentiation. Implementing autodiff to work in general requires a lot of programming (especially for reverse-mode), so instead we'll specialize to one particular function, and choose a function with a straightforward "computational graph". Let

$$f : \mathbb{R}^{d_0} \to \mathbb{R}, \quad f(\boldsymbol{x}) = \text{sum}\left(\sigma(B \cdot \sigma(A \cdot \boldsymbol{x}))\right)$$

where $\boldsymbol{x} \in \mathbb{R}^{d_0}$, $A \in \mathbb{R}^{d_1 \times d_0}$, $B \in \mathbb{R}^{d_2 \times d_1}$ and $\sigma(\alpha) = (1+e^{-\alpha})^{-1}$ is the 1D logistic function (aka *the* "sigmoid" in ML terminology) and σ applied to a vector is done componentwise. Basically, this is a simple feed-forward neural net. We're going to compute the gradient of f, $\nabla f(\boldsymbol{x})$, aka $J_f(\boldsymbol{x})^\top$. *Be careful, in neural net training, we take gradients with respect to the weight matrices, but in this problem we are thinking of the weights as fixed and differentiating with respect to $\boldsymbol{x}$, since that's slightly simpler since it's a vector not a matrix*

a) Let $h_1(\boldsymbol{x}) = A \cdot \boldsymbol{x}$, $h_2(\boldsymbol{y}) = \sigma(\boldsymbol{y})$, $h_3(\boldsymbol{y}) = B \cdot \boldsymbol{y}$, $h_4 = h_2$, and $h_5(\boldsymbol{y}) = \text{sum}(\boldsymbol{y})$. Then

$$f(\boldsymbol{x}) = \text{sum}\Big(\overbrace{\sigma(\overbrace{B \cdot \sigma(\underbrace{A \cdot \boldsymbol{x}}_{\boldsymbol{y}_1})}^{\boldsymbol{y}_2})}^{\boldsymbol{y}_3} \Big)}^{\boldsymbol{y}_4} = h_5(h_4(h_3(h_2(h_1(\boldsymbol{x})))))$$

so we can write the Jacobian of F as

$$J_f(\boldsymbol{x}) = J_{h_5}(\boldsymbol{y}_4) \cdot J_{h_4}(\boldsymbol{y}_3) \cdot J_{h_3}(\boldsymbol{y}_2) \cdot J_{h_2}(\boldsymbol{y}_1) \cdot J_{h_1}(\boldsymbol{x}).$$

For part (a), mathematically work out what the Jacobian of each of the h_k functions is and write out your answer.

b) Implement the function f in code, and use an existing automatic differentiation package (I suggest PyTorch) to get the gradient, which we will later use to check the correctness of our code. *I suggest choosing moderate values for d_0, d_1, d_2 and make these values different to help find bugs in your code. The matrices A and B can be arbitrary, e.g., random.*

c) Implement a gradient for your function in the forward-mode style. That is, calculate

$$J_f(\boldsymbol{x}) = J_{h_5}(\boldsymbol{y}_4) \cdot \left(J_{h_4}(\boldsymbol{y}_3) \cdot \left(J_{h_3}(\boldsymbol{y}_2) \cdot \left(J_{h_2}(\boldsymbol{y}_1) \cdot J_{h_1}(\boldsymbol{x}) \right) \right) \right).$$

This gradient function should be in the same function that also calculates $f(\boldsymbol{x})$, so now have that function return two values, $f(\boldsymbol{x})$ and $J_f(\boldsymbol{x})$. Comparing with the autodiff software (PyTorch) at one or more points $\boldsymbol{x}$, make sure you get the right answer. *Note: autodiff software like PyTorch works in single precision by default, so you shouldn't expect your answer to be more than about 5 to 8 digits the same. Try using double precision, and so you should have 10 to 15 digits the same.*

d) Implement a gradient for your function in the reverse-mode style. That is, calculate

$$J_f(\boldsymbol{x}) = \left(\left(\left(J_{h_5}(\boldsymbol{y}_4) \cdot J_{h_4}(\boldsymbol{y}_3) \right) \cdot J_{h_3}(\boldsymbol{y}_2) \right) \cdot J_{h_2}(\boldsymbol{y}_1) \right) \cdot J_{h_1}(\boldsymbol{x}).$$

Again, implement this in the same function that calculates $f(\boldsymbol{x})$, since you will want to save some intermediate values on the forward pass. Comparing with the autodiff software, make sure you get the right answer.

e) What is the computational complexity for the forward-mode style (in terms of d_0, d_1, d_2)? What about for reverse-mode style?

f) Set $d_0 = d_1 = d_2 = 8000$ and choose A, B to be random matrices (double-precision). Time how long it takes your code to run in forward-mode, and how long it takes to run in reverse-mode, and also compare with how long it takes the autodiff package to run.

For this entire problem, please turn in relevant written answers as well as source code, nicely formatted.

Problem 2: Students in 5720 only We'll consider finding the derivative of the 1D function $f(x) = \sqrt{x}$, but we'll calculate the square root using Heron's method, an iterative method, in order to illustrate how autodiff can work with `for` loops. The method is simple: letting x be the input

Require: Parameter $N \in \mathbb{N}$

$y \leftarrow 1$ {Any positive initialization works, but let's use 1}
for $k = 1, 2, \ldots, N$ **do**
  $y \leftarrow \frac{1}{2}\left(y + \frac{x}{y}\right)$
end for
return y

and is actually Newton's method applied to the problem $r(y) \stackrel{\text{def}}{=} y^2 - x = 0$.

a) Implement Heron's method and compare with established methods (e.g., `numpy.sqrt`) to check your answer. *Small N should work, like $N = 10$, but it's more exciting for this problem to choose larger N, like $N = 100$ or 1000.*

b) Implement forward-mode style automatic differentiation for this function and check your answer using the true value $f'(x) = \frac{1}{2}x^{-\frac{1}{2}}$.

Hint: Pick something concrete for N for now, like $N = 3$. We can conceptually think of

of the algorithm as computing

$$
\begin{aligned}
y_1 &= \frac{1}{2}(1+x) \\
y_2 &= \frac{1}{2}\left(y_1 + \frac{x}{y_1}\right) \\
y_3 &= \frac{1}{2}\left(y_2 + \frac{x}{y_2}\right).
\end{aligned}
$$

We want $\frac{dy_3}{dx}$, and we can get there using the total derivative (sometimes covered in the *implicit differentiation* section of your calculus book) which is

$$
\frac{dy_3}{dx} = \frac{\partial y_3}{\partial y_2}\frac{dy_2}{dx} + \frac{\partial y_3}{\partial x}
$$

and we'll find $\frac{dy_2}{dx}$ in a similar (recursive) way. When you actually do this in code, your method should allow for any value of N. In fact, your method could even work with *adaptive* stopping rules (like when the change in y between iterations is less than some tolerance)!

c) **Bonus: not required** Implement reverse-mode style automatic differentiation for this problem.