Programming Assignment 1: Backpropagation

CS4787/5777 — Principles of Large-Scale ML — Fall 2023

Project Due: September 13, 2023 at 11:59pm

Late Policy: Up to two slip days can be used for the final submission.

Please submit all required documents to **Gradescope**. There are two separate assignments there, one for the code and one for the report. You need to submit **both**.

This is a group project. You can either work alone, or work **ONLY** with the other members of your group, which may contain AT MOST three people in total. Failure to adhere to this rule (by e.g. copying code) may result in an Academic Integrity Violation.

Overview: In this project, you will be implementing and testing a simple backprop engine. The backprop engine you'll implement differs from more sophisticated machine learning systems, such as PyTorch, in that it uses a simple object oriented structure (rather than the more functional approach that is typical of modern ML systems). This makes it easier for you to understand how control flow works within the backprop algorithm, at the cost of requiring some additional boilerplate which would otherwise be unnecessary

Part 1: A Simple Backprop System for Scalars In this programming assignment, we'll walk through a simple implementation of backprop. This implementation has the following characteristics.

- All scalars/vectors/matrices (arrays) used in backprop will be represented by Python objects of a type that inherits from `BackproppableArray`.
- If an array was produced by some computation that we can differentiate (backprop through), then the type of the array (a subtype of `BackproppableArray`) will indicate what operation produced the array. For example, the result of a an addition of two arrays will be of type `BA_Add`.
- If an array is an input variable or constant, it will just be of type `BackproppableArray`.
- Each array object "knows" how to backprop through it. To backprop through a `BackproppableArray` object, we call its `grad_fn` method. This method is overloaded in each subclass, and is a no-op in the base class (since constants require no backprop as they don't depend on anything).
- Each array object also "knows" which other backproppable arrays it depends on (the arrays that were used directly to compute it). These arrays are stored in its `dependencies` field.
- The actual work of scheduling the backpropagation is done in the `backward` method of the `BackproppableArray` class.

In this part, we're going to implement a simple scalar version of this backprop system.

1. First, implement the utility function `BackproppableArray.all_dependencies`. This function returns **all** the backproppable arrays on which an array depends in its computation, including indirectly. That is, this function should return a python list containing this array, any direct dependencies of this array (i.e. the arrays stored in the `dependencies` list), any dependencies of those dependencies, any dependencies of **those** dependencies, and so on indefinitely. To implement this utility function, you'll want to use some sort of graph search algorithm, e.g. breadth first search, using the `dependencies` field as the edges of graph.
2. Next, implement the backpropagation function `BackproppableArray.backward`. This function, when called on a scalar-valued `BackproppableArray` y, manifests, for each backproppable array x on which it depends, the gradient of y with respect to x in the field `x.grad`. Your implementation should perform the following steps, the first two of which are already implemented in the starter code:
    1. check that `backward` is being called on a scalar-valued tensor
    2. call `all_dependencies` to get a list of all dependencies of the expression we're differentiating
    3. sort the found dependencies so that the ones computed last go FIRST

- you can do this using the `.order` field of the `BackproppableArray` object, which is initialized from an always-increasing counter, so `BackproppableArray` objects constructed more recently have larger values for their `.order` field

4. initialize and zero out all the gradient accumulators (.grad) for all the dependencies
5. set the gradient accumulator of this array to 1, as an initial condition, since the gradient of a number with respect to itself is 1
6. call the `grad_fn` function for all the dependencies in the sorted reverse order (i.e. arrays that were computed LAST have their backprop `grad_fn` called FIRST)

3. Now we've implemented the core of the backprop algorithm...but we still need to implement the functions that do the differentiating. For each of the computation types `BA_Sub`, `BA_Mul`, `BA_Div`, `BA_Exp`, and `BA_Log`, implement its `grad_fn` method **for scalar-valued arrays** (i.e. do not worry about vectors, matrices, et cetera yet). We've already implemented the `BA_Add` method, which should give you a sense of what is needed. The `grad_fn` method should:
    1. take as input the `.grad` field of the backproppable array on which it is called
    2. compute, for each of its inputs, the gradient of the output with respect to that input
    3. add each of those gradients to the `.grad` field of the corresponding input
    4. concretely, if the function this class is associated with is F (e.g. if we're doing an add then F is $F(x, y) = x + y$) and g is the gradient in the `.grad` field, then we want to add $\nabla_x(g^T F(x, y))$ to the `x.grad` field and add $\nabla_y(g^T F(x, y))$ to the `y.grad` field.
4. Now we're going to test your implementation! First, implement the functions `TestFxs.df1dx`, `TestFxs.df2dx`, and `TestFxs.df3dx` by symbolically differentiating (by hand) the functions given in `TestFxs.f1`, `TestFxs.f2`, and `TestFxs.f3`, respectively.
5. Next, for each of the functions f_1, f_2, and f_3, compare (1) your symbolic derivative implementation from Part 1.4, (2) the numerical derivative output by `numerical_diff` in the starter code, and (3) your automatic differentiation result as output by `backprop_diff` from the starter code. Also compare the latter two for f_4 (which is similar, except I didn't ask you to differentiate it by hand because that would be a bit tedious). **Include a description of how you performed the test and what you observed in your report.**

Part 2: Backprop on Tensors Next, we're going to expand your implementation to work on more general arrays, rather than just on scalars. We now suppose that `.data` and `.grad` can hold numpy arrays, not just numpy scalars.

1. For each of the computation types `BA_MatMul`, `BA_Sum` (for simplicity, suppose that we are only handling sums with `keepdims=True`), `BA_Reshape`, and `BA_Transpose`, implement its `grad_fn` method.
2. Using backprop in Python depends on Python's operator overloading capabilities. In Part 1, we used operator overloading that was already written in the starter code, so that when we write something like `x + y` for backproppable arrays x and y, we'd produce a new object of type `BA_Add`. But analogous code is not implemented for matrix multiply using the `@` operator. Implement the necessary operator overloading for the `@` operator to produce behavior analogous to `+`, `-`, and $*$. (Hint: you may need to reference the Python documentation here).
3. Now that we're working with vectors, the starter code implementation of `BA_Add.grad_fn` and your previous implementations of `BA_Sub.grad_fn`, `BA_Mul.grad_fn`, and `BA_Div.grad_fn` may be wrong because they don't handle broadcasting correctly. You may find the test functions g_1 and g_2 to be useful to debug this. Fix these implementations. **Include a description of what's wrong with the starter code implementation of add and how you fixed it (and the other methods) in your report.**
4. Now we're going to test your implementation! For each of the functions g_1 and g_2, compare (1) the numerical derivative output by `numerical_diff` in the starter code, and (2) your automatic differentiation result as output by `backprop_diff` from the starter code. **Include a description of how you performed the test and what you observed in your report.**
5. We want to further test your implementation for functions that take vectors as input. To do this, we need a vector version of `numerical_diff` to compare against as a baseline. Implement the function `numerical_grad`, which

approximates a gradient numerically by computing $\frac{f(x+\epsilon e_i)-f(x-\epsilon e_i)}{2\epsilon}$ for each standard basis vector e_i in the input, and storing it in the appropriate entry of the output gradient vector. (This function should contain a Python for-loop over the indicies of the input array.)

6. For each the function h_1, compare (1) the numerical derivative output by your `numerical_grad` and (2) your automatic differentiation result as output by `backprop_diff` from the starter code. **Include a description of how you performed the test and what you observed in your report.**
7. Now write your own function with a relatively high-dimensional input (e.g. $d = 1000$) and scalar-valued output. Use both your `numerical_grad` and `backprop_diff` to compute its gradient (for some input). Measure the time taken to compute the gradients for both methods. What does this suggest about the relative speed of these two approaches? **Include what you observed and your conclusions in your report.**

What to submit:

1. An implementation of the functions in main.py.
2. A lab report containing:
   - A one-paragraph summary of what you observed during this programming assignment.
   - The description of your approach and observations from Part 1.5.
   - The description of what's wrong and how you fixed it from Part 2.3.
   - The description of your approach and observations from Part 2.4.
   - The description of your approach and observations from Part 2.6.
   - Your measurements and conclusions from Part 2.7.

Setup:

1. Run `pip3 install -r requirements.txt` to install the required python packages