Write Up 1

Bryant Har

September 13, 2023

1 Programming Assignment Summary

Summary of Observations

Overall, I observed through practice the overarching structure of backpropagation and auto-differentiation. The process involves procedurally generating, storing, and accumulating gradients until the desired gradient is computed. This method naturally has a basis in symbolic differentiation and can be approximated with numerical differentiation. Further, I observed that non-symbolic differentiation nevertheless can accumulate floating point errors that may influence the fidelity of results but that usually remain manageable for reasonably constrained and well-behaved inputs and tests. Finally, I saw how backpropagation is significantly faster than numerical differentiation and far easier to compute than manual or symbolic differentiation.

1.5

Testing was done by executing a loop with 1,000 random inputs ranging from -10 to 10. Then, the results for auto, numerical, and symbolic differentiation are computed. I asserted that their absolute difference be less than some threshold (usually 10^-5) to check that they were effectively equal. This verified that every trial for all three functions passed.

I observed while testing that, while all three generated the same values, some computations' precision suffered from floating point precision. For example, when I reduced comparison tolerance to around 7 decimal places, some of the test cases failed, because one computation would evaluate to 0, while another would be extremely close but not equal to zero (e.g. 0.000056). Very trivially, I also noticed that test cases could fail if you tested inputs in the range of 1000s, since it would cause an overflow in numerical differentiation. I observed that all 1000 tests passed near instantly.

2.3

The problem with the starter code add is incompatible shapes and the inability to handle/combine gradients of different shapes. The starter code is not robust to these multidimensional operations. This can be solved by implementing a custom broadcasting function, as I did in the code, named "reshape_()". For example, when we conduct an add operation between a scalar-valued gradient and a vector-valued gradient, we must broadcast the values so that they match each other and can consequently be combined and subsequently accumulated.

2.4

g_1 and g_2, are tested similarly to 1.5. Random values from -10 to 10 were generated as inputs, and numerical results were compared against the auto-differentiated results. 1,000 trials were conducted for each function and compared against the numerical differentiation output. The acceptable tolerance was, as before, 10^{-5}. I observed that all 1000 tests passed near instantly.

I noticed generally the same things as 1.5, though the discrepancy between the numerical and our result seemed to increase with function complexity. Additionally, I noticed that since the derivative of g_2 involves log values, exceedingly small values led to extremely high magnitude outputs, which decreased precision. As the input goes to zero, the differential goes to negative infinity. Therefore, I clipped the range of random valued inputs to be at least 0.01, so that the tolerance could be reasonably made around 1e-5. I experimentally determined this value, which generally prevented overflow or the accumulation of excessive error. I observed that all 1000 tests passed near instantly.

2.6

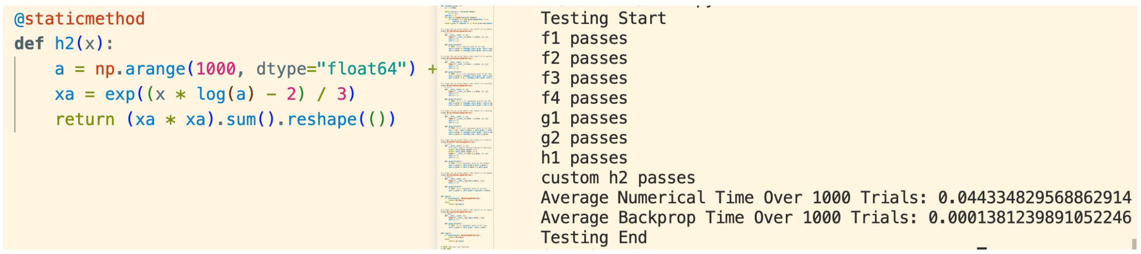
Testing for h was performed similarly to g and f. However, the inputs generated were random vectors, instead of scalars. To achieve this, I generated random numbers from 0 to 10, and randomly assigned negative values. I used these inputs for

1,000 trials, and compared the numerical differentiation results to our results, and ensured that they were always within some tolerance of each other, 10^{-5} as usual. Specifically, I took the maximum element of both results, and ensured that the results were no more than 10^{-5} relative to each other.

There were no particular things notable about testing that I didn't document in g and f. I observed that all 1000 tests passed near instantly, though noticeably slower than f and g, perhaps by half a second.

2.7

I wrote a function requiring an 1000-dimensional input and that involved complex functions like logs and exponentiation. I observed that the auto-differentiation I implemented was significantly faster than numerical differentiation.
Auto-differentiation offers a significant speedup, which critically expedites the many computations required to train models. The times averaged over 1000 trials are featured below, with numerical and auto taking 0.443 and 0.000138s respectively. This corresponds to an approximate 321x speedup.



```
@staticmethod
def h2(x):
    a = np.arange(1000, dtype="float64") +
    xa = exp((x * log(a) - 2) / 3)
    return (xa * xa).sum().reshape(())
```

```
Testing Start
f1 passes
f2 passes
f3 passes
f4 passes
g1 passes
g2 passes
h1 passes
custom h2 passes
Average Numerical Time Over 1000 Trials: 0.044334829568862914
Average Backprop Time Over 1000 Trials: 0.00013812398910522246
Testing End
```