Prof. Ryan Cotterell

Assignment 1: Backpropagation

20/09/2022 - 09:54h

The first question of this assignment is to be solved in writing. Please submit a PDF file with your computer-written solutions (preferably LaTeX / Overleaf). The second question is a coding task. Please submit your code adhering to all naming and formatting requirements as detailed in the question description.

1 Theory

Question 1: Efficiently Computing the Hessian (50 pts)

Backpropagation on a computation graph can be used to efficiently obtain the derivatives of a function with respect to the input. In this problem, we consider a differentiable map $f : \mathbb{R}^n \to \mathbb{R}$. This question makes heavy use of the operator ∇ (pronounced nabla). We write $\nabla f(\mathbf{x})$ to denote the **Jacobian** of f evaluated at point $\mathbf{x}$. Importantly, in the special case that the co-domain of f is simply $\mathbb{R}$, we refer to the Jacobian of f as the **gradient** of f. When ∇f gives us a gradient, we will take it to be a row vector. Furthermore, we write $\nabla\nabla f(\mathbf{x})$ or $\nabla^2 f(\mathbf{x})$ to indicate the **Hessian** of f. Recall from Lecture 2 that $f(\mathbf{x})$'s Hessian is the matrix of second derivatives at point $\mathbf{x}$. This notation is intuitive because the Hessian of f is simply the Jacobian of the gradient of f.

You will now derive an algorithm to compute the Hessian $\nabla^2 f(\mathbf{x})$ using repeated calls to backpropagation.

a) Show that the following identity holds true:

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \nabla(\mathbf{e}_1 \nabla f(\mathbf{x})^\top) \\ \vdots \\ \nabla(\mathbf{e}_n \nabla f(\mathbf{x})^\top) \end{bmatrix} \tag{1}$$

where $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ are the standard basis vectors of $\mathbb{R}^n$. In words, you are asked to demonstrate that $\nabla(\mathbf{e}_i \nabla f(\mathbf{x})^\top)$ computes the i^{th} **row** of the Hessian $\nabla^2 f(\mathbf{x})$.

Then, give an expression to compute the i^{th} **column** of the Hessian that is similar to one for the i^{th} row derived above.

b) Use the identity in equation 1, to briefly describe an efficient (backpropagation) algorithm for computing $\nabla^2 f(\mathbf{x})$ and discuss the runtime of your algorithm. Take m to be the number of edges in the function's computation graph. Recall from class that this implies that the computation of f runs in $\mathcal{O}(m)$ time. Explain why this constitutes a faster algorithm than the more naïve way of computing the Hessian entry by entry and also give the runtime of this more naïve algorithm.

c) Now, briefly describe an algorithm to compute the tensor of 3$^{\text{rd}}$-order derivatives and discuss its runtime in terms of m. Then, explain how to produce the tensor of k^{th}-order derivatives and discuss its runtime in terms of m. The answer for the k^{th}-order derivative can be qualitative as it is tricky to come up with good notation.

d) Describe an $\mathcal{O}(m)$ algorithm for computing the *diagonal* of $\nabla^2 f(\mathbf{x})$. **Hint**: This is a tricky problem unless you look at it the right way. The general idea is to develop an analogue of the multivariate chain rule for elements of the Hessian's diagonal. Then, you can develop a dynamic program very similar to backpropogation itself that has the correct runtime. Recall that the multivariate chain rule says

$$\frac{\partial y_i}{\partial x_j} = \sum_{k=1}^{M} \frac{\partial y_j}{\partial z_k} \frac{\partial z_k}{\partial x_j}$$

and our goal is to compute $\frac{\partial^2 y_i}{\partial x_j \partial x_j}$, another notation for the diagonal elements of the Hessian.

2 Practice

Question 2: Coding Backpropagation (50 pts)

In this question, we ask you to code backpropagation in Python. You are not allowed to use any libraries other than `abc`, `argparse`, `ast`, `collections`, `csv`, `itertools`, `math`, `pyparsing`, `random`, `re` and `sys`. Please make sure you have version 3.0.9 of `pyparsing` installed.

We provide five scripts:

(i) **parsing**: the `Parser` class is converting a math problem into infix notation that we can perform subsequent operations on. In particular, it is taking a math problem as a string input, as well as optional initialization values of the problem's variables. The parser is provided and should not be changed.

$$\texttt{exp(x) - (y * 2)} \longrightarrow \texttt{[['exp', 'x'], '-', ['y', '*', 2]]} \tag{2}$$

(ii) **building**: the `Builder` class is taking the infix notation math problem as input and instantiates the class variable `self.graph` with a computation graph. Class initialization accepts the input variables as an additional, optional argument to avoid naming conflicts with the intermediate and output variables which may be arbitrarily chosen from the set of unicode chars. You are free in choosing your *own data structure* for representing the computation graph. As a simplification, we *do not require optimized parallelization* — the order of operations should be correct and free of conflicts pertaining variable-sharing. However, operations that can be executed in parallel may also be consecutively executed.

(iii) **executing**: the `Executor` class takes two dictionaries as input: the computation graph and the initialized input variables. The class should implement the functions `forward` and `backward`, which use the input variables to run a forward and backward pass on the computation graph respectively. After executing both, two class variables of the `Executor` class should be initialized: `self.output` should hold the output of the forward pass as a float value; `self.derivative` should hold the dictionary of first-order partial derivatives of all input variables sorted alphabetically. Given the input variables `in_vars` = {'x': 2.0, 'y': -2.0} and the computation graph `b.graph` as constructed in the `Builder` class (item (ii)), the outputs should be:

```
e = executing.Executor(graph = b.graph, in_vars = in_vars)
e.forward()
e.backward()
print(e.output) ## 11.39
print(e.derivative) ## {'x': 7.39, 'y': -2.0}
```

(iv) **operations**: the Executor should make use of basic operations defined via the abstract `Operator` class. You should implement the functions

```
self.fn_map = {"log": Log(), "exp": Exp(), "+": Add(),\
"-": Sub(), "^": Pow(), "sin": Sin(), "*": Mult(), "/": Div()}
```

where log is the natural logarithm. In particular, each class inheriting from `Operator` must have a function method `self.f()` and a first-order partial derivative method

`self.df()`. Both take one or two variables as an input for unary or binary operations respectively. While `self.f()` always returns a single float, `self.df()` returns a single-element list of floats for unary and a two-element list of floats, i.e. a Jacobian vector, for binary operations.

(v) **main**: For grading your submission, we will run the `main.py` script which loads a set of math problems and executes the above pipeline of `parsing`, `building` and `executing`. The class `main.py` must be executable as

```
python3 ../backprop/main.py -problems ../backprop/problems.tsv
```

where the first argument is the path to the main script and the second argument is an optional path to a *tsv* file of math problems. We provide such a *tsv* file with exemplary math problems, one per row, in `problems.tsv`. The `main` script loads the *tsv* file into a list of dicts. Each dict is a math problem and consists of 4 key-value pairs: `problem:` math equation as string, `in_vars:` dict of input variables, `output:` true output solution of the math problem as float and `derivative:` dict of the input variables' partial derivatives. In particular, we compare if the `Executor` class variables `self.derivative` and `self.output` match the true solution. Therefore, we round all floats to two decimal places in `main.py`. For testing, we run the main script, but pass another *tsv* file with different math problems. For this reason, you should use `main.py` to develop and test your code and *please verify that your code is runnable via the above terminal line command*! This means, beware of correct relative path imports of script functions.

Please do not change anything in the scripts `main.py` and `parsing.py`. In any other script, you can of course add more class functions that are not already laid out in the code and even add a helper script if needed. You may also change function signatures if the change does not require any changes in `main.py` and `parsing.py`. Note that we will run software on your submissions to *test for plagiarism*.