*Prof. Ryan Cotterell*

# Assignment 1: Backpropagation

29/09/2022 - 16:21h

The first half of the assignment is a series of theoretical questions about the material. When you have answered the questions to your satisfaction, you should upload a pdf file with your solutions (preferably written in LaTeX) to Moodle. The second question is a coding task that is to be solved in the released Google Colab notebook. You have to copy the notebook to your own drive in order to edit it. When submitting your solution, please execute all cells in your copied notebook, then save it and include a link to your notebook in your PDF file submission. We will run software on your submission to *test for plagiarism.*

**Important:** Please ensure, that all cells in the notebook are already **executed** and that the notebook is accessible via the shared link in **Editor mode**! The notebook must also be executable from top to bottom without throwing errors.

## 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 $\nabla$ (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 $\nabla f$ gives us a gradient, we will take it to be a column 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 can assume that $f$ is twice differentiable. You will now derive an algorithm to compute the Hessian $\nabla^2 f(\mathbf{x})$ using repeated calls to backpropagation.

a) **(10 pts)** Show that the following identity holds true:

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

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

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

b) **(15 pts)** Suppose $f$ can be computed in $\mathcal{O}(m)$ time. Use the identity in equation 1, to briefly describe an efficient algorithm, based on backpropogation, for computing $\nabla^2 f(\mathbf{x})$. Discuss the runtime of your algorithm. 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) **(15 pts)** Now, briefly describe an algorithm to compute the tensor of 3rd-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) **(10 pts)** 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 backpropagation 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 in a Google Colab notebook:

`https://colab.research.google.com/drive/1lSgujLn0ETFUMssy_ClRAb3PlbhtUoQU?usp=sharing`

There are a several cells, marked with "[do not change]", that you are not allowed to modify! In other other cells, you may of course add more (class) functions. However, do not change the signatures of existing functions, such as class initializations. Also, you are not allowed to import any libraries other than the ones that are being automatically imported in the cell "Library Imports".

The notebook allows you to select one of four math problems in a drop-down menu. Each math problem is represented as dict 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.

The provided `Parser` class is converting the math problem into infix notation that we can perform subsequent operations on. In particular, it is taking a math problem as a string input (`problem`), as well as optional initialization values `in_vars` of the problem's input variables and returning a list `infix`. As a simplification, you can expect operations to be grouped, e.g. $x + y + z$ will be $(x + y) + z$, so that the parser always returns lists (and sublists) of length 3 at most and individual operations (such as +) will expect 2 variables at most, as discussed in item (ii). See for instance:

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

There are three cells for three classes with open ToDos for you to implement:

(i) **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.

```
g = Builder(infix: list, in_vars: dict)
print(g.graph)
```

(ii) **Operations**: the `Operator` class defines functions that are used by the `Executor` class (item (iii)), which is defined next. You are asked to implement several child classes that all inherit from the abstract `Operator` class and that define the following 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.

(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. 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 (i)), the outputs should be:

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

For grading your submission, we will run all cells in your notebook from top to bottom. In particular, we will consider the cell "Test Function for Grading". This loads a set of math problems and executes the full pipeline of `parsing`, `building` and `executing`. 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.

**Note:** Please ensure that you share your copied notebook via a link in **Editor mode**. The notebook must be fully executable and all cells should be executed already!