AI535 Deep Learing – Homework 1

Background Math, Autograd, and Backpropagation

Overview and Objectives. In this homework, we are going to practice some of the mathematical skills we'll be using in class and learn more about techniques for autograd software by developing a small one ourselves.

How to Do This Assignment. All questions that require a response will be in blue-gray task boxes. We prefer solutions typeset in LaTeX but will accept scanned written work if it is legible. If a TA can't read your work, they can't give you credit. Submit your solutions to Canvas as a PDF.

Advice. Start early. You may be rusty on some of this material. Some of it might be new to you depending on your background – seek out resources to refresh yourself if so.

1 Flexing Our Mathematical Muscles

In this section, we will work through some mathematics to support what we will be learning in lecture. Training deep learning models involve optimizing the values of vector-valued functions to minimize some function measuring error – typically with first-order methods based on derivatives. So, let's play with these things.

▶ Q1 Convexity [10pts]. Suppose $f(x)$ is a convex function and let $a < b$. Show that:

$$\frac{f(x) - f(a)}{x - a} \leq \frac{f(b) - f(a)}{b - a} \leq \frac{f(b) - f(x)}{b - x} \tag{1}$$

for $x \in (a, b)$. Draw a sketch that illustrates this inequality.

▶ Q2 Linear Algebra [5 pts]. If A and B are positive definite matrices, prove that the matrix $\begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}$ is also positive definite. 0's here represent matrices of zeros of sufficient size to fill.

▶ Q3 Calculus I [10pts]. Compute the gradient $\nabla f(\mathbf{x})$ and Hessian $\nabla^2 f(x)$ for the function

$$f(\mathbf{x}) = (x_1 + x_2)(x_1 x_2 + x_1 x_2^2) \tag{2}$$

Find at least three stationary points of this function and compute the Hessian at each. Show that $\mathbf{x} = [3/8, -6/8]^T$ is a local maxima.

▶ Q4 Calculus II [10pts]. Show that the function

$$f(\mathbf{x}) = 8x_1 + 12x_2 + x_1^2 - 2x_2^2 \tag{3}$$

has only one stationary point, and that it is not a minimum nor a maximum, but a saddle point.

▶ Q5 Graphs [5pts]. A *topological order of a directed graph* $G = (V, E)$ is an ordering of its nodes such that for every edge $(v_i, v_j) \in E$, node i appears in the ordering before node j. Prove that every directed acyclic graph (DAG) has a topological order.

Hint: A DAG always has at least one node with no incoming edges.

2 Automatic Differentiation with Dynamic Computation Graphs

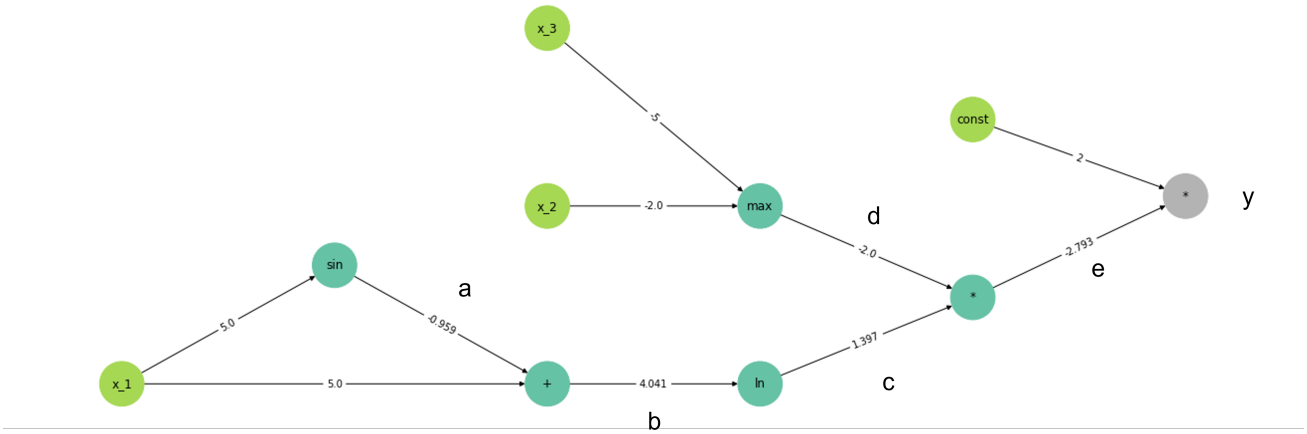
In practice, computing gradients manually is error-prone and time-consuming – especially when implementing complex deep learning architectures. Instead, algorithms that automatically keep track of computation and compute gradients have been developed. These are referred to as autograd or autodiff frameworks. In this section, we will develop a simple autograd framework for scalar values and a limited set of operations to better understand these systems.

2.1 Background

Computational Graphs. The first abstraction we need to understand is the notion of a *computational graph*. In this abstraction, we represent an equation (or other form of computation) by a graph of interconnected operations. For example, the equation

$$y = ln(x_1 + sin(x_1)) * max(x_2, x_3) * 2 \tag{4}$$

evaluated at $x_1 = 5$, $x_2 = -2$, and $x_3 = -5$ can be represented by the following graph



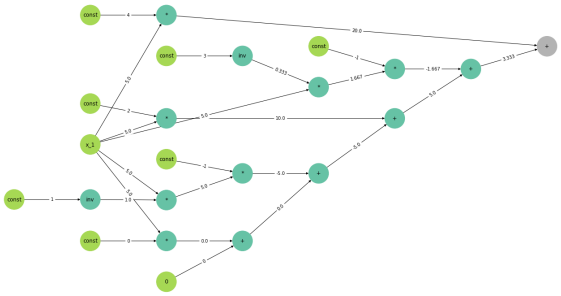
In this graph, inputs and constants are green nodes. Interior nodes correspond to operations and are colored cyan or grey. Edges show intermediate values of the computation and are labelled a to e for our discussion in the next section. For different values of x_1, x_2, and x_3, the edge values will change, but the structure is statically defined by Eq. 4.

In autograd frameworks, special datatypes are defined for numeric calculation and computation graphs can be dynamically generated as the computation executes. This is often nearly transparent to the user. For example, the following lines of code generated this graph in the simple autograd framework we developed for this assignment:

```
x1 = Variable(5.0, name="x_1")
x2 = Variable(-2.0, name="x_2")
x3 = Variable(-5, name="x_3")

y = ln(x1+sin(x1))*max(x2,x3)*2
y.drawGraph()
```

Aside from declaring the x's as `Variable` types, the computation itself looks like perfectly normal Python. In fact, autograd frameworks can be integrated fairly well with standard programming patterns like conditionals and iteration as in the example below. Note that the graph can get fairly complicated.

```
x1 = Variable(5.0, name="x_1")
y = Variable(0)

for i in range(5):
    if i % 2 == 0:
      y = y + x1*i
    else:
      y = y - x1/i

y.drawGraph()
```



This abstraction of computation graphs will be the primary data structure of our autograd framework. In the next two subsection, we'll discuss how to use this abstraction to compute gradients efficiently and automatically.

Chain Rule Follows Computational Graphs. Returning to the initial graph above labeled, suppose we rewrite Eq. 4 to match the labels of each intermediate operation:

$$
\begin{aligned}
a &= sin(x_1) && (5)\\
b &= x_1 + a && (6)\\
c &= ln(b) && (7)\\
d &= max(x_2, x_3) && (8)\\
e &= c * d && (9)\\
y &= e * 2 && (10)
\end{aligned}
$$

If we want to know the derivative of y with respect to x_2, we could apply chain rule and write

$$\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial x_2} \tag{11}$$

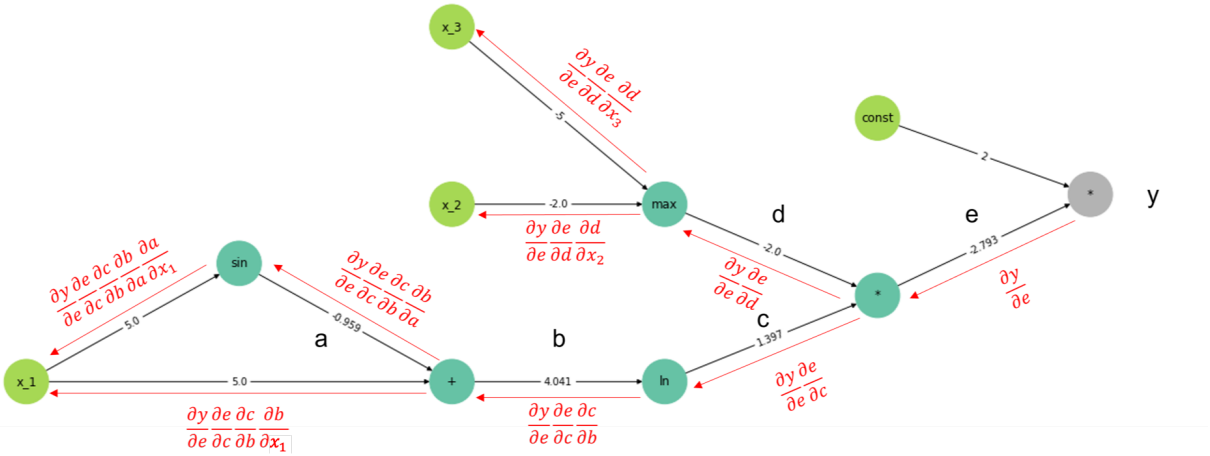
by following paths from y to x_2 in the graph above. Likewise, for x_1 we could follow *all* paths from y and write:

$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial e}\frac{\partial e}{\partial c}\frac{\partial c}{\partial b}\left(\frac{\partial b}{\partial x_1} + \frac{\partial b}{\partial a}\frac{\partial a}{\partial x_1}\right) \tag{12}$$

where the final parenthetical terms account for both routes through which x_1 affects b.

Notably, these expressions we've derived in Eq. 11 and Eq. 12 depend only on a product or sum of partial derivatives of our atomic operations at the nodes. For example, the `ln` node produces c via the natural logarithm of b such that $\partial c/\partial b$ is simply $1/b$. This suggests that if we define the partial derivative for each operation, we should be able to compute gradients by tracing paths on the computational and accumulating the product of gradients.

Backpropagation on Computational Graphs. Building on this intuition, we present the *backpropagation* algorithm on our running example computational graph below. We will refer to the back arrows/labels and corresponding evaluation of the function as the *forward pass*. The red arrows/labels and corresponding computation of gradients will be referred to as the *backward pass*.



Examining the right-most product node that computes $e = d * c$, we can observe a key insight. During the backward pass, this node takes in the gradient of y with respect to its output e ($\partial y/\partial e$) and this term persists in what it passes to both its children nodes. $\partial y/\partial e$ is multiplied with the partial of e with respect to each input respectively.

With this, we can characterize the forward and backward behavior of each node in general. During the forward pass, a node f operates by computing its output given the input.

$$output = f_{forward}(input) \tag{13}$$

During the backward pass, the node f operates by taking the gradient of y with respect to f's output and producing the gradient of y with respect to its inputs.

$$\frac{\partial y}{\partial input} = f_{backward}\left(\frac{\partial y}{\partial output}\right) = \frac{\partial y}{\partial output}\frac{\partial output}{\partial input} \tag{14}$$

Notably, this requires each node to collect gradients from all its parents in the backward pass *before* executing and passing gradients to nodes before it. Computing an ordering for processing the nodes that has this property requires a topological sort of the graph.

2.2 Implementing Autograd

With the background material covered, let's implement an autograd framework. The `DIYAutograd.py` file provides Python skeleton code that marks out the structure of our implementation. The `numpy` package is required. If you want to visualize the computational graphs, you'll need to install the `networkx`, `matplotlib`, and `pydot` packages. If your environment does not include `graphviz` (e.g. Windows), you may need to install it from https://graphviz.org/download/ and add it to your path. Alternatively, if you don't want to visualize the graphs – simply comment out any lines where `drawGraph()` is called.

Variable. We represent our computational graph using the `Variable` class. These represent both input variables and intermediate computations. `Variables` have the following class variable:

- `id` – a unique id
- `value` – the numerical value of the node's output
- `children` – a list of the input `Variable` used in the computation. The ordering matters for each operation.
- `func` – the function used for computing the output from the inputs. As we'll discuss later, this also determines the function used during the backward pass.
- `grad` – a scalar where the gradient will be stored after the backward pass
- `name` – an optional string to identify the `Variable` when plotting the computational graph

`Variables` make up the nodes in our computation graph and the directed edges are defined by the `children` variables. Note that most of the common mathematical operations are overridden in `Variable` to make use of the specialized version we describe next.

Functions. To define the forward and backward computation at each node, we define `Functions`. The function responsible for multiplication is shown below:

```
class Prod:
  label="*" # Symbol to be shown when visualizing the graph

  # Given a and b, compute a*b and add a new corresponding Variable
  # to the computational graph. Return this new Variable
  def forward(a,b):
    a = Variable.convert(a) # Converts to a Variable if not already
    b = Variable.convert(b) # Converts to a Variable if not already
    return Variable(a.value*b.value, [a,b], Prod) # Adds the computation to the graph

  # parent -- Variable resulting from forward of this Prod function
  # children -- list of Variables that were input during forward
  # Return a list of partial derivatives of the output with respect to each input
  def backward(parent, children):
    return [children[1].value, children[0].value] #[d(a*b)/da=b, d(a*b)/db=a]
```

This function defines the forward operation of multiplying Variables `a` and `b` by generating a new Variable with value `a.value*b.value`, recording that the `Prod` function was used, and noting that Variables `a` and `b` were inputs. In the backward pass, it returns the gradient of its output with respect to its inputs.

▶ Q6 Implement Unfinished Functions [5pts]. The code has a number of unfinished classes for the functions listed below. Implement these using the `Prod` code above as a guide.

- Inverse `Inv` – Given input `x`, compute `1/x`.
- Addition `Add` – Given input `a` and `b`, compute `a+b`.
- Natural Logarithm `Ln` – Given input `a`, compute `ln(a)`
- Exponentiation `Pow` –Given input `a` and `b`, compute a^b.
- Sine `Sin` –Given input `a`, compute `sin(a)`.
- Cosine `Cos` –Given input `a`, compute `cos(a)`.
- Tangent `Tan` –Given input `a`, compute `tan(a)`.
- Minimum `Min` – Given input `a` and `b`, compute `min(a,b)`.
- Maximum `Max` – Given input `a` and `b`, compute `max(a,b)`.

After completing these correctly, all unit tests in `functionUnitTests()` should pass.

Backpropagation. Calling `y.backward()` on a `Variable` `y` initiates the backpropagation algorithm to compute the derivative of y with respect the other nodes in the graph. This is done as described in the background section.

▶ Q7 Finish Backpropagation Code. [5pt] The current code for backpropagation is incomplete. A helper function has already performed a topological sort of the nodes in the computational graph, but computing and passing the gradient has not been implemented. The skeleton code is shown below.

```
def backward(self):

    # Compute a topological ordering so each node is processed
    # after all parents have sent gradients to it
    nodes = self.topoSort()

    # Execute the backward pass for each node in the topological sort

    self.grad=1 # dy/dy = 1

    # Work through the graph backwards
    for parent in nodes:
        dy_dparent = parent.grad

        # TODO: Update children's gradient
```

To finish the `TODO`, implement the following:

- Perform the backward pass of `parent`'s function to compute ∂`parent`$/\partial$`inputs`.
- For each child c of `parent`, add $\partial y/\partial$`parent` $*$ ∂`parent`$/\partial$`c` to `c`'s gradient.

After completing this correctly, all unit tests in `backpropUnitTests()` should pass.

Closing the Loop. As we said in the beginning, gradients are an important part of optimization in deep networks. To get a sense for how autograd frameworks like the one we've develop make that easier, let's reconsider Eq. 2 from Q3.

$$f(\mathbf{x}) = (x_1 + x_2)(x_1 x_2 + x_1 x_2^2) \tag{15}$$

In the `main()` function of the skeleton code, a simple gradient ascent demo is provided for this function. Starting from $x_1 = 1$ and $x_2 = -1/2$, we adjust x_1 and x_2 to move in the direction of their gradients. This process seeks a local maxima. After 500 iterations, the code outputs the resulting x_1 and x_2 values which should converge to $3/8$ and $-6/8$ from this starting point. In our neural networks, we will do something very similar to minimize a loss function that measures error on our training set.

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?