Autograd for Algebraic Expressions

Date:2024.3.22

Chapter 1:Introduction

The automatic differentiation problem requires us to differentiate a given expression (composed of symbols and operators, such as a, b, c,+- */...) for each variable (i.e. a, b, c, d...) and output them separately, with the derivative represented by other variables. For example ,If you input a * b+c, the output will be a: b, b: a, c: 1. The symbols we need to handle are+- */, and the variable is a string (which may also be aa, xx, etc.)

Chapter 2 Algorithm Specification

Description in pseudo-code form:

- **Tokenize(string s)**: Breaks down the input expression string into a list of tokens.
    - For each character **c**:
        - If **c** is a space, ignore it.
        - If **c** is a parenthesis or operator, end the current token, add it to the list, and add **c** as a new token.
        - Otherwise, add **c** to the current token.
- **InfixToPostfix(vector<string> tokens)**: Converts an infix token list to postfix (Reverse Polish) form.
    - Use a stack to handle operator precedence and parentheses.
    - For each token:

        - If it's a number or letter, add it to the output list.
        - If it's "(", push it onto the stack.
        - If it's ")", pop the stack until "(" is encountered.
        - If it's an operator, pop the stack while the precedence is higher or equal to the current operator, then push the current operator onto the stack.
    - Pop any remaining operators from the stack to the output list.
- **BuildTree(vector<string> postfix)**: Builds an expression tree using the postfix token list.
    - For each postfix token:
        - If it's an operator, pop two nodes from the stack as right and left children, create a new operator node, and push it onto the stack.
        - If it's a number or variable, create a node and push it onto the stack.
    - The node at the top of the stack is the root of the tree.
- **Diff(Node node, string s)**:Computes the derivative of the given expression tree relative to variable **s**.
    - If the node is null or the operator is unsupported, return "0".
    - If the node is the variable **s**, return "1".
    - For addition and subtraction, recursively derive the left and right child nodes and connect them with the same operator.
    - For multiplication, apply the product rule.
    - For division, apply the quotient rule.
    - For power functions, simplify handling the case where the exponent is a constant.
- **ExpressionToString(Node node)**:Converts an expression tree to a string representation.
    - For leaf nodes, return the node value.
    - For internal nodes, recursively convert left and right children, possibly adding parentheses based on the operator, and combine them.

Main Function

- Read the expression string **s** from user input.
- Tokenize **s** and convert it to postfix form.
- Build the expression tree **exprTree** using the postfix tokens.
- For each variable token:
    - Compute the derivative of **exprTree** relative to that variable, constructing the derivative tree **derivativeTree**.
    - Convert **derivativeTree** to string **derivativeExpr**.
    - Simplify **derivativeExpr**.
    - Print the simplified derivative expression.

Data Structure specification:

Node: Represents a node in the expression tree, containing a string value (operator, variable, or constant) and pointers to the left and right child nodes.

Data structure trees (binary trees) and **stacks, arrays, strings,and pointers** are also used in this program.

Chapter 3:Testing result

Test case 1:

Description:only contain "+" operation,with the variable that only contain one character.

Intention:test the fundamental function of the program,including the basic+function.

Status:Passed

Test case 2:

Description:including the * and / operation,together with the + and – operation.

Intention:To test the further performance of the program in more complex invoronment.

Status:Passed.

Test case 3:

Description:including the ^ operation,which requires the program to output “In”,for example,a^b requires the program to output b*a^(b-1) and a^b*In(a).

Intention:To test the exponent function of the program,along with other function combined with each other.

Status:Passed.

Attention: Due to some technical issues,some additional operator and extra “0” will appear in the output,but they don’t influence the correct answer of the input,so just ignore them .Sorry for the matter.

Chapter 4 Analysis and Comments

Tokenization (Tokenize function)

Time Complexity: O(N), where N is the length of the input string. Each character is visited once.

Space Complexity: O(N), as a vector of tokens is created. In the worst case, each character could be a separate token (e.g., a series of operators and operands with no multi-character variables).

Infix to Postfix Conversion (infixtopostfix function)

Time Complexity: O(N), assuming the number of tokens is proportional to the length of the input string. Each token is processed once. The while loop inside the else block may seem to increase complexity, but each element is pushed and popped from the stack only once, maintaining O(N) overall.

Space Complexity: O(N) for storing the stack and the resulting postfix vector.

Binary Expression Tree Construction (buildtree function)

Time Complexity: O(N), where N is the number of tokens. Each token is processed once to build the tree.

Space Complexity: O(N) for the stack and O(N) for the tree itself, leading to O(N) overall. The depth of the tree, and hence the stack size, is limited by the structure of the expression but does not exceed the number of tokens.

Differentiation (Diff function)

Time Complexity: The Diff function is recursive and visits each node in the expression tree. Its complexity is O(T) for a tree with T nodes. In the worst case, T is proportional to N, giving O(N). However, for each node, especially for "*" and "^" nodes, the function may create multiple new nodes, leading to a multiplier effect. This can increase the complexity, but it's still bounded by a constant factor of the tree size.

Space Complexity: O(H) for recursive stack space, where H is the height of the tree. The space for the newly created nodes in the differentiation process also needs to be considered, but since nodes are created and not stored in a data structure that grows with N, the space complexity remains O(H).

Expression String Generation (expressionToString function)

Time Complexity: O(T), as it traverses each node in the tree once.

Space Complexity: O(H) due to recursion, where H is the height of the tree. The string concatenation may temporarily increase space usage, but this is bounded by the depth of the recursion.

Conclusion

The time complexity of the entire program is dominated by the parts that operate on the expression tree (building, differentiating, and converting to string), each being O(N) in terms of the number of tokens/nodes.

The space complexity is mainly influenced by the size of the expression tree (O(N)) and the depth of recursive calls (O(H)), where H is the tree height. In the worst case, the tree can become linear (e.g., a long chain of operations), making H proportional to N, but this is not the typical case for balanced expressions.

Further possible improvements

For this automatic differentiation program, we can certainly add more features, such as taking derivatives of sine, cosine, and logarithmic functions, to make its functionality more comprehensive. For the form of output results, it can also be simplified by removing unnecessary symbols and extra zeros, making the results more beautiful. Due to the author's own level and time constraints of the assignment, these improvements cannot be completed within a limited time. We apologize for any inconvenience this may cause and the author will make improvements in the future.

I hereby declare that all the work done in this project titled " Autograd for Algebraic Expressions" is of my independent effort.