Nabla

Tensor Operations and Automatic Differentiation

Kartik Srinivas - ES20BTECH11015
Tanmay Garg - CS20BTECH11063
Aayush Patel - CS20BTECH11001
Gautham Bellamkonda - CS20BTECH11017
Ketan Sabne - CS20BTECH11043
Ojjas Tyagi - CS20BTECH11060
Ganesh Bombatkar - CS20BTECH11016
August 25, 2022

Index

1 Introduction

1.1 What is Nabla?

Nabla is simple, domain specific, statically typed programming language, developed for simplifying tensors computation. Nabla supports automatic-differentiation via implementing a static computational graph in the backend.
Simple: Programming in Nabla is simple and intuitive. Tensors are provided as basic data types, and special operators are provided for operation on them.
Domain Specific: Nabla is designed to solve certain class of problem involving tensors and optimization, It has higher level of abstraction than other general purpose programming language.
Statically typed: Nabla is statically typed language, type of every variable is known at compile type. Program can be further optimised to give better performance.

1.2 Why Nabla?

The Greek letter Nabla is used in vector calculus as differential operator. Our language Nabla is designed for Automatic differentiation of a multivariable Tensor function. It additionally provides basic matrix and tensor operations. Autodiff gives more accurate numerical values of derivatives. Nabla is designed to give the users a direct touch of the computational graph by allowing to define the nodes easily, so that one can calculate gradients of complicated tensor functions easily.

2 Language Specifications

Nabla follows a standard similar to C, this makes it easier for programmers to learn our language and use some new cool features, data types and their operations in amazing ways. This section also describes how Nabla is different from C, C++ and Python.

2.1 Primitive Data-types

Nabla does not follow Object Oriented Programming *a.k.a.* OOPs, but we use very simplistic data types that are generally available in all languages. The data types are quite similar to C standard data types. The data types are ***int***, ***float***, ***char***, ***Tensors***, ***Bool***.

2.1.1 Integer Type

Integer numeric data types are **int**. There is no support for *unsigned int* kind of data types in Nabla.

2.1.2 Char Type

The **char** type is used to store a single character, using 8 bits.

2.1.3 Float Type

For real numbers numeric data types, there is a ***float*** data type which is 64 bit. The ***float*** follows the IEEE 754 specification for floating point values.

2.1.4 Tensor Type

Nabla supports *n-dimensional* Tensor data type similar to *TensorFlow* and *PyTorch*. The Tensor is made of only 1 data type and that is ***float***. Basically, Tensor is like a ***matrix*** of ***floats***.

2.1.5 Boolean Type

Boolean data types are also supported, which is binary in nature similar to that of other languages such as *C, C++, Python and Java*.

2.2 Tensors

In our internal system a Tensor is a vector or an array of floating point values. We currently support operations for nD Tensors such as all forms of arithmetic operations, trigonometric, exponential, and we also support some features of automatic differentiation as explained later based on the computational graph generated.

2.2.1 Arithmetic Addition

As mentioned above Tensor is 2D array of floating points.We can add two tensors and subtract as well just like we perform operations on two matrices. As we can assume tensor as any $m \times n$ matrix where $m \in \mathbb{N}$ *and* $n \in \mathbb{N}$.
Let's see an example. Let A and B be $m \times n$ two tensors

$$TensorC = A + B$$

$$\forall_{i,j} \quad C(i,j) = A(i,j) + B(i,j)$$
$$0 < i \leq m,\ 0 < j \leq n$$

.

2.2.2 Trigonometric

We can perform trigonometric functions on tensors in Nabla. Such as

$$Tensor\, T = [-0.5461, 1.3547, -2.7486, -0.2746]$$

Upon applying sin() function on our *Tensor*

$$\sin(T) = [-0.5194, 0.9747, -0.3829, -0.2711]$$

Upon applying cos() function on our *Tensor*

$$\cos(T) = [0.8545, 0.2144, -0.9237, 0.9625]$$

In similar way, we have all other trigonometric functions for tensors. All trigonometric operations are element-wise operations and return modified *Tensor*

2.2.3 Exponential

In Nabla, there is support for exponential function on Tensors. This exponential function also works element-wise on elements of the Tensors

$$Tensor\, T = [a, b, c]$$

Upon applying exp() function on our *Tensor*

$$\exp(T) = [e^a, e^b, e^c]$$

It will return modified *Tensor*. Similarly, there is support for logarithmic function where base is e

$$Tensor\, T = [a, b, c]$$

Upon applying ln() function on our *Tensor*

$$\ln(T) = [\ln(a), \ln(a), \ln(a)]$$

2.2.4 Multiplications

Nabla supports Tensor Multiplication operation with a new defined operator @, which performs matrix multiplication after dimension checks of both Tensors

$$T = [1 \quad 2]$$

$$S = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$T@S = \begin{bmatrix} 7 & 10 \end{bmatrix}$$

2.2.5 Element-wise multiplication

Nabla supports element-wise multiplication of two Tensors when both are of same dimensions

$$\forall_{i,j} \quad m_{ij} = t_{ij} * s_{ij}$$

$$0 < i \leq m \; and \; 0 < j \leq n$$

where t_{ij} denotes the element at i, j coordinate of $Tensor\; T$

2.3 Comments

Comments in Nabla follows C like syntax. They are added to make code more readable to developers.

2.3.1 Single Line Comments

Single line comments start with // symbol.

2.3.2 Multi line Comments

Comments start with character /* and end with the character */. Nested comments are not allowed and they can not be inside string literals.

```
// Single line comment

/*
    Multi line comment
*/
```

Figure 1: Example of Comments

2.4 Operators

2.4.1 Arithmetic Operators

Operator	Description	Associativity
@	Tensor Multiplication	Left
*	Multiplication	Left
/	Division	Left
%	Modulo	Left
+	Addition	Left
-	Subtraction	Left

2.4.2 Logical operators

Operator	Description	Associativity
&&	Logical AND	Left
||	Logical OR	Left

2.4.3 Comparison operator

Operator	Description	Associativity
==	Equal to	Left
!=	Not Equal to	Left
>	greater than	Left
>=	greater than or equal to	Left
<	Less than	Left
<=	Less than or equal to	Left

2.4.4 Assignment Operator

Operator	Description	Associativity
=	assign AND	Right
*=	multiply by OR	Right
+=	increment by OR	Right
-=	decrement by OR	Right
%=	modulo by OR	Right
/=	divide by OR	Right

2.4.5 Operator Precedence

Operators in decreasing orders of precedence
1. @
2. * / %
3. + -
4. == != > >= < <=
5. && ||
6. = *= /= %= += -=

2.5 Control-flow

Nabla has conditional commands for control flow of the program. Nabla has basic selection statements such as *if - elif -else* We plan to enforce the curly braces inorder to to avoid the ambiguity problem that arises in this grammar.

2.5.1 Loop

The loop that will be made will incorporate the C - style syntax. This is for simple iterations

```
loop(initialize; boolean statement; increment)
{
    expression;
}
```

Figure 2: Example of Loop Statement

2.5.2 If Statements

Below a code example has been provided

```
if(boolean statement)
{
    expression;
}
elif(boolean statement)
{
    expression;
}
else
{
    expression;
}
```

Figure 3: Example of If Statement

2.6 Constants

The constants that will be present are going to be mainly of the following types:-

1. **Integer Constants:** These constants contain digits (0-9) and do not contain decimal point and may be positive or negative or zero. Eg: $1, -5, 0$
2. **Char Constants:** These constants contain a single character. Eg: 'a', 'Z', '0'

3. **Float Constants:** They are numeric constants that contain a decimal point. $1.0, -2.56, 3.14$

4. **Tensor Constants:** These constants represent vector/matrix constants. Each element in the Tensor is a float constant. Eg: [1 2], [[1, 2], [3, 4]];

5. **Bool Constants:** These constants represent only two values: True or False.

2.6.1 Identifiers

The identifiers used in our language are as follows:-

2.6.2 Keywords

Nabla reserves the following keywords.

char	***int***	***float***	***Tensor***
Bool	***if***	***elif***	***else***
loop	***char***	***cns***	***return***
var	***sizeof***	***void***	***function***

2.7 The Autodiff Mode

One of the features that we want to build into Nabla is the ability to perform automatic differentiation by building **dynamic computational graphs.** This idea is inspired from the way Tensorflow is built. The computational graph will be built at **runtime** Therefore we allow support for if-conditions and control-flow changes. The graph that has been built is therefore un-fixed at compile time. The computational graph can be built in C++, once the parse tree has been made Here is an example of how the code will look like:-

```
var Tensor a[2][2]  = [[2.0, 3.0],[4.0, 5.0]];
var Tensor b[2][1]  = [3, 1];
var Tensor c[2][1]  = [4,6];
var Tensor z[2][1];
cns float d = 6.5;
var float m = 7.0;
z = a@b + c;
print(z)
cns int k  = 5;
cns int z = k >> 1;
if(k > 0 || a[0][1]){
        c  = a + b;
        if(b > = 0){
            b  = b*25;
        }
}
elif(k < 0){
        c = a - b;
}
else{
        c = a;
}
loop(int i = 0; i< 10; i++){
    c = c + a;
}
float k = 1.0;
var Tensor p [2][1];
var Tensor Final[2][1];
p = x@y;
Final = cos(p + z);
backward(Final);
print(grad(a));
print(grad(m));
```

Figure 4: Autodiff Rough Example

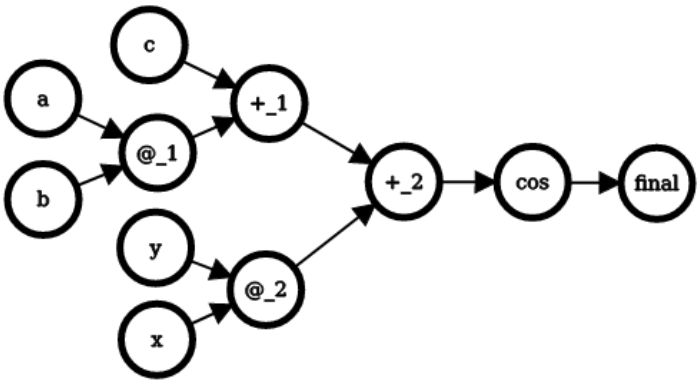
2.7.1 The Computational Graph

A computational graph is a graph that has been built for the purpose of computing gradients with respect to intermediate variables.

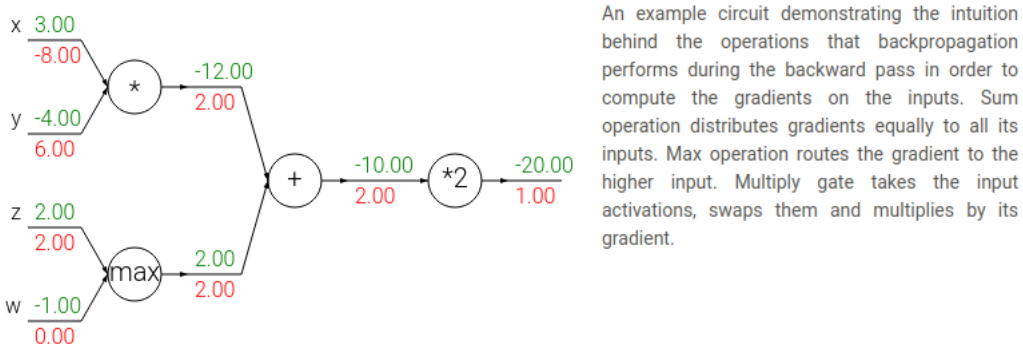
The picture below shows tensor(1x1) variables x, y, z, w and some operations performed on them, the values of the output are shown in green. What is required, is the gradient of the output variable with respect to the input variable.

This gradient is calculated **recursively using the chain rule.** The intermediate gradients are stored in red. The gradients flow backwards.

To compute the backward flow of gradients we **must perform a forward pass first.** Therefore this mode will be implemented post the implementation of the basic tensor operations.



(a)



(b)

Figure 5: (a) Graph for above code above (b) Backward Gradient Flow in CG

2.7.2 Operators Supported

Nabla supports the following elementary operations for the creation of computation graphs.

+	-	@
*	/	max()
min()	Trigonometric functions	Exponential functions

3 Grammar

3.1 Start

start $\longrightarrow$ *declarations*

3.2 Miscellaneous

exp $\longrightarrow$ *assignment_exp* | *exp* $'$,$'$ *assignment_exp*
assignment_exp $\longrightarrow$ *unary_exp assignment_operator assignment_exp*
assignment_operator $\longrightarrow$
$'=$$'$
| *MUL_ASSIGN*
| *DIV_ASSIGN*
| *MOD_ASSIGN*
| *ADD_ASSIGN*
| *SUB_ASSIGN*
| *LEFT_ASSIGN*
| *RIGHT_ASSIGN*
| *AND_ASSIGN*
| *XOR_ASSIGN*
| *OR_ASSIGN*
| *AT_ASSIGN*

unary_exp $\longrightarrow$ *postfix_exp*
| *INC_OP unary_exp*
| *DEC_OP unary_exp*
| *lib_funcs* $'$ ($'$*type_name*$'$) $'$

lib_funcs $\longrightarrow$
COS
| *LOG*
| *BACKWARD*

primary_expression $\longrightarrow$
identifier
| *constant*
| *string*
| *(expression)*

$postfix\_expression \longrightarrow$
$primary\_expression$
$| \; postfix\_expression[expression]$
$| \; postfix\_expression++$
$| \; postfix\_expression--$
$| \; postfix\_expression.identifier$
$| \; postfix\_expression(argument\_expression\_list_{opt})$

$argument\_expression\_list \longrightarrow$
$assignment\_expression$
$| \; argument\_expression\_list', ' \; assignment\_expression$

$storage\_class\_specifier \longrightarrow$
VAR
$| \; CNS$

$statement \longrightarrow$
$compound\_statement$
$| \; expression\_statement$
$| \; iteration\_statement$

$iteration\_statement \longrightarrow$
$LOOP \; '(' \; expression\_statement \; expression\_statement \; expression \; ')'$

$expression\_statement \longrightarrow$
$';'$
$| \; expression \; ';'$

$compound\_statement \longrightarrow$
$"\{""\}"$
$| \; "\{" \; statement\_list"\}"$
$| \; "\{"'declaration\_list"\}"$
$| \; "\{"declaration\_list \; statement\_list"\}"$

$declaration \longrightarrow declaration\_specifiers \; ';'$
$| \; declaration\_specifiers \; init\_declarator_list \; ';'$

$declaration\_specifiers \longrightarrow storage\_class\_specifier$
$| \; storage\_class\_specifierdeclaration\_specifiers$

$|\ type\_specifier$
$|\ type\_specifier declaration\_specifiers$

$init\_declarator\_list \longrightarrow init\_declarator$
$|\ init\_declarator\_list' ,' init\_declarator$

$init\_declarator \longrightarrow declarator$
$|\ declarator\ '='\ \ initializer$

$initializer \longrightarrow$
$assignment\_expression$
$|\ "["initializer\_list"]"$
$|\ "["initializer\_list' ,' "]"$

3.3 Constants

$constant \longrightarrow$
$integer\_constant$
$|\ character\_constant$
$|\ floating\_constant$
$|\ bool\_constant$

4 References

1. PyTorch - https://arxiv.org/pdf/1912.01703.pdf
2. TensorFlow - https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
3. Kernighan and Ritchie C Reference Manual
4. Tensor Comprehensions