Damo

Ian Covert (icc2115): Project Manager
Alan Gou (ajg2233): Tester
Abhiroop Gangopadhyay (ag3661): System Architect
Hari Devaraj (sd2920): Language Guru

Introduction

Motivation

Our goal is to create a programming language that provides a superior way of modeling mathematical functions. Our system would rely on a graph to capture the dependencies of the many variables and intermediate values that make up a function.

The advantage of such a system is that unlike functions in most programming languages, our functions can be differentiated. That capability is extremely useful in machine learning applications, where algorithms like stochastic gradient descent might require differentiating a loss function with respect to several million tunable parameters.

Background

Nearly every programming language has the concept of a function. Functions can take an arbitrary number of arguments, and then use those arguments to compute and return a value. Such functions can be used for various purposes, but in particular they can act like mathematical functions.

Take Python as an example. In Python, we could write the following code, which models a relationship between seven mathematical variables:

```
1  # z is a function of x and y
2  def z_func(x, y):
3          return x + y
4
5  # x is a function of a and b
6  def x_func(a, b):
7          return a * b
8
9  # y is a function of c and d
10 def y_func(c, d):
11         return c - d
```

```
12
13 # Compute z as a function of a, b, c, d
14 def f1(a, b, c, d):
15         x = x_func(a, b)
16         y = y_func(c, d)
17         return z_func(x, y)
18
19 # Compute z as a function of x, y (bypassing a, b, c, d)
20 def f2(x, y):
21         return z_func(x, y)
22 f1(1, 2, 3, 4)     # Returns 1
23 f2(2, -1)          # Returns 1
```

The functions f1 and f2 do an adequate job of capturing the desired dependencies between our variables. However, in Python, as in most programming languages, the functions f1 and f2 could not be differentiated. That functionality is critical for certain use cases, and our language is constructed to support that feature natively.

Our language is inspired by a Python library called Theano, which was developed specifically to provide this kind of functionality. It serves as the back-end in several deep learning libraries, including Lasagne and Keras.

Machine Learning Use Case

To motivate the need for such a language, consider the following scenario. Suppose we're using a machine learning algorithm for which the loss function doesn't have a closed-form solution, as in the case of neural networks. We optimize our parameter choices to best fit the data by iteratively taking the gradient of the loss function and slightly tweaking the model's parameters. The process of taking a small batch of data, computing the gradient of a multinomial loss function, and shifting the parameters in the direction of the gradient is known as stochastic gradient descent, and has the objective of minimizing the loss function.

Our new language offers an easier way to optimize loss functions without a closed form solution by making the process of taking the gradient of a multinomial function a core feature.
Consider how the loss function is constructed: it is a function of both the data (perhaps a subset of the data) and the current values assigned to the tunable parameters. When computing the gradient of the loss function, the developer is interested only in the derivative with respect to the parameters, and not the derivative with respect to the data. In other words, from the perspective of the loss function, the data act as constants. Our language provides features that make this functionality possible.

Dependency Graph

Our dependency graph is restricted to binary outgoing edges to model the common binary mathematical operations. We can spawn additional temporary nodes in our dependency graph to model multinomial

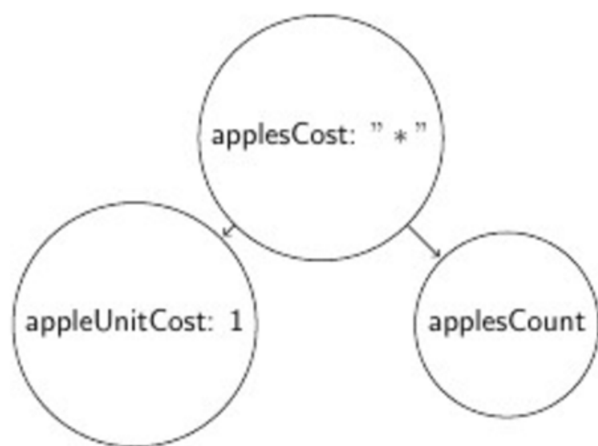
functions and maintain our binary dependency paradigm.

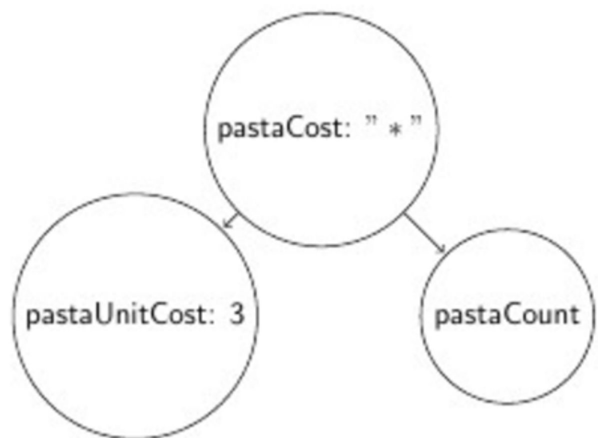
Shopping Cart Example

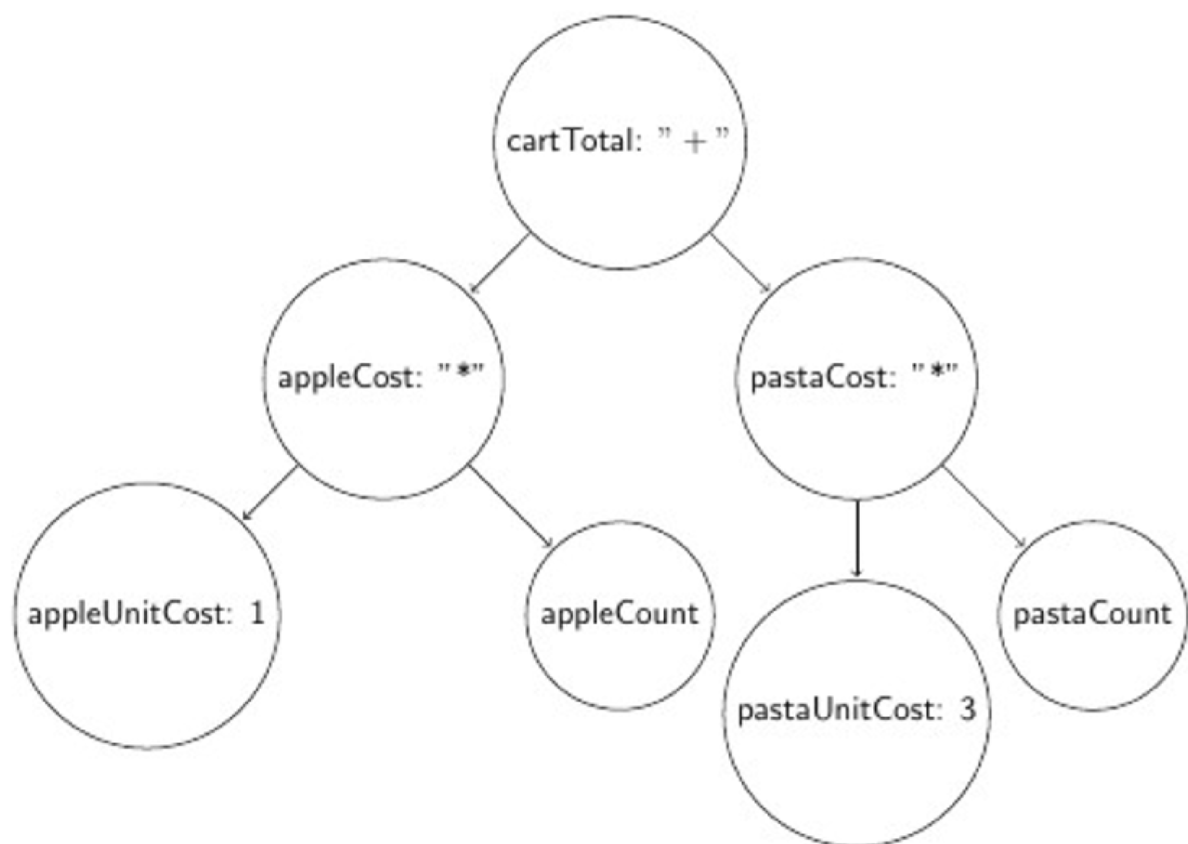
Consider the example of a shopping cart, and how the total cost of the items is calculated. In our language, we could write the following:

```
// Model total cost of apples
MathConstant apples_unit_cost = 1;
MathVariable apples_count;
MathVariable apples_cost;
apples_cost = apples_unit_cost * apples_count;

// Model total cost of pasta
MathConstant pasta_unit_cost = 3;
MathVariable pasta_count;
MathVariable pasta_cost;
pasta_cost = pasta_unit_cost * pasta_count;

// Model cost of shopping cart
MathVariable cart_total;
cart_total = apples_cost + pasta_cost;

// Create function that computes total cost
f = MathFunction([apples_count, pasta_count], cart_total);
MathFunction f = MathFunction([apples_count, pasta_count], cart_total);
f.eval([4, 5]);     // Returns 19
```

These lines of code would result in the following dependency graphs:


applesCost: " * "
appleUnitCost: 1
applesCount


pastaCost: " * "
pastaUnitCost: 3
pastaCount


cartTotal: " + "
appleCost: "*"
pastaCost: "*"
appleUnitCost: 1
appleCount
pastaUnitCost: 3
pastaCount

Note that the function f does not require apples_unit_cost or pasta_unit_cost to be passed as inputs, as they are constant values. Also note that the function acts as a wrapper around the dependency graph, but it could be wrapped around the dependency graph in a different manner:

```
MathFunction f2 = MathFunction([apples_cost, pasta_cost], cart_total);
f2.eval([4, 5]);      // Returns 9
```

Notes on how the dependency graph is constructed:

- It's a directed acyclic graph.
- Each node represents a variable or a constant.
- Each node has at most two outgoing edges.
- A function acts as a wrapper around the expression tree. A valid function computes an output based on a sufficient set of inputs.

Notes on nodes in the dependency graph:

- A node corresponding to a variable has at most two children. A node corresponding to a constant value has no children.
- A node can have an arbitrary number of incoming edges.
- Nodes corresponding to variables keep track of the nodes that they depend on, as well as the operation used to combine those nodes (e.g. addition, multiplication).

Types

Name	Description	Examples
int	A basic 32 bit integer. Used primarily for array indexing, not for mathematical functions.	`int x = 43`
num	Like a double in C. All values assigned to variables and constants are presumed to be of this type.	`num y = 43.0`
boolean	A value that is either true or false.	`boolean z = tr`
array	A statically sized sequence of items, which must be of the same type.	`int[] arr1 = [` `num[] arr2 = [`
String	A string	`String s = "He`
MathVariable	A data type representing a mathematical variable. It may depend on other values (MathVariables and/or MathConstants), and other MathVariables might depend on it. MathVariables do not store numerical values, but can have values assigned to them when a function is evaluated or differentiated.	`MathVariable x` `MathVariable z`
MathConstant	A data type representing a constant value. It cannot depend on other values (including both MathVariables and MathConstants), but MathVariables may depend on it. It stores a value, but that value may be changed.	`MathConstant G` `G = 3.7`
MathFunction	A function acts as a wrapper around a part of a dependency graph. It is	`MathConstant x`

	the only way of evaluating a mathematical function modeled by the dependency graph.	`MathVariable y`
		`MathVariable z`
		`MathFunction f`
		`MathFunction([`

Operators

Binary Mathematical Operators

Our binary mathematical operators can be used to combine ints, nums, MathVariables and MathConstants. The behavior can vary depending on which types are being combined. For example, combining two ints will result in an int, combining two nums will result in a num, and combining two MathVariables would result in a new MathVariable.

Operator	Description	Examples

+	Addition	`1 + 2`
		`1.0 + 2.0`
		`MathVariable x, y`
		`MathVariable z = x + y`

-	Subtraction	`1 - 2`
*	Multiplication	`2 * 3`
/	Division	`3 / 2` // Returns 1 `3.0 / 2` // Returns 1.5 `3.0 / 2.0` // Returns 1.5
^	Exponentiation	`3 ^ 2`
%	Modulation	`3%2` //Returns 1

Comparison Operators

Comparison operators are meant to be used on ints and nums.

Operator	Description	Examples
<	Less than	`1 < 2`
<=	Less than or equal	`1 <= 1`
>	Greater than	`2 > 1`

>=	Greater than or equal	`2 >= 2`
==	Equal to	`3.0 == 3`
!=	Not equal to	`2 != 3`

Boolean Operators

Boolean operators are intended to operate on one or more boolean values.

Operator	Description	Examples
and	Logical and	`true and true` `true and false`
or	Logical or	`true or false`
not	Logical not	`not false`

Features

MathFunctions

MathFunctions are used to wrap around a part of a dependency graph. If a MathVariable can be uniquely computed given a set of input MathVariables and MathConstants, that part of the dependency graph can be turned into a MathFunction.

The validity of the MathFunction is determined at the time of creation (e.g. a function would be invalid if the output couldn't be computed without additional inputs being specified).

Two operations can be performed on a MathFunction.

Evaluate
The evaluate method is a simple functional evaluation. Pass in a list of values for the input MathVariables, and the function will be evaluated by propagating values through our back-end dependency graph.

Gradient
A method that differentiates the function with respect to the input MathVariables. The compiler should not have to perform any checking at this point because the correctness checking should be done at function creation time. Returns a one dimensional MathVariable (or vector)

MathVariables

A primitive data type that's used to model variables in mathematical expressions. MathVariables can be defined as a function of other MathVariables and MathConstants. The MathVariable type does not store a value; rather values are assigned at the time of MathFunction evaluation, or the evaluation of a MathFunction's gradient.

MathConstants
A primitive data type that's used to model constant values in mathematical expressions. MathConstants cannot depend on other MathConstants or MathVariables. A crucial difference with MathVariables is that MathConstants store a value. As such, a MathConstant's value does not need to be specified at the time of a MathFunction's evaluation, and does not even need to be explicitly specified as an input.

With this design choice comes the desirable feature that a MathFunction can be differentiated with respect to all of its inputs, which would include MathVariables but not MathConstants.

Methods

Methods are our built in representation of programmatic functions. Return statements are optional.

I/O

Data can be printed to standard output using print(). Input can be read from files using functions like from_csv().

Control Flow

Standard implementation of loops and conditionals (for and while loops and if/elif/else supported)

Not Supported

Tensor algebra is not supported in our dependency graph. While our language would be much more powerful if it allowed tensor algebra, that would be a project in itself.

Syntax and Language Conventions

Comments

```
// for single line comments
/* for multiline comments */
```

Method Definitions

```
def trivial_function(<parameter1>, <parameter2>) : <returnType> {
        /* instructions */
        return 1;
}
```

Methods are defined using the def keyword. Return type is denoted after the colon following the function parameters. Curly braces are required around all function definitions.

Primitive Types

Variables of a primitive types must be declared, and their data type must be specified.

```
int a = 5
num b = 2.0
String s = “hello world”
MathVariable a, b, c;
MathConstant d, e, f;
// Shorthand for assigning values to MathConstants
MathConstant g = 5;
// Shorthand for expressing dependencies of MathVariables
MathVariable h = a + b + c - g;
// Functions cannot be declared without expressing their inputs and outputs
MathFunction my_function = MathFunction([a, b, c], h);
// MathFunction my_other_function would be illegal
```

Sample Programs:

Basic Hello World

Prints hello world, like you’d expect.

```
def hello_world() : {
  print("Hello world");
}
hello_world() // prints "Hello world"
```

Basic GCD function

Calculates the greatest common denominator of two integers.

```
def gcd(int a, int b) : int {
  while (b > 0){
   b = a % b;
   a= a / b;
  }
  return b;
}
```

Calculating Euclidean Distance

Creates a dependency graph by combining several MathVariables, and then wraps their relationship in a MathFunction. It then creates two methods that call the evaluation and gradient methods of the MathFunction.

```
MathVariable x, y;
MathVariable z = (x^2 + y^2)^0.5;
MathFunction my_function = MathFunction([x, y], z);
def euclidean_distance(int a, int b) : int {
  return my_function.eval([a, b]);
}
def euclidean_diff(int a, int b) : int {
  return my_function.grad([a, b]);
}
euclidean_distance(3, 4); // returns 5
euclidean_diff(3, 4); // returns [0.6, 0.8]
```