CmpE 260 - Principles of Programming Languages
Spring 2021 - Project 2

Deadline: 18 June 2021 23:59

Alper Ahmetoğlu
ahmetoglu.alper@gmail.com

1 Introduction

In this project, you will implement an automatic differentiation [1]. Automatic differentation is a computation technique to calculate partial gradients of an expression by a repetitive application of the chain rule of calculus [2]. This technique makes the gradient calculation of any expression very trivial. This is one of the very important methods (if not the most) for machine learning research. After completing this project, you will have a very minimal implementation of something like TensorFlow [3] or PyTorch [4] in which you can actually train an artificial neural network (albeit a bit slowly compared to them).

2 The Data Structure

We will first define a data structure called `num` which will have two fields: `value` and `grad`. `value` will hold the value of the number, and `grad` is for holding the partial gradient of a function with respect to this parameter. Don't worry, you will not calculate the gradient by yourself, this will be automatically handled with some tricks.

```
;; plain, simplest version of this struct.
> (struct num (value grad))
> (define mynumber (num 12.3 1.0))
> (num-value mynumber)
12.3
> (num-grad mynumber)
1.0
> mynumber
#<num>
```

I have added some additional descriptions to the `num` struct for printing so that the output will be `(num 12.3 1.0)` instead of `#<num>`.

3 Helpers

You will implement the following two procedures which you can think as a warm-up exercise.

3.1 (get-value *num-list*)

This procedure will return the values of numbers in `num-list` as a list. If `num-list` is a `num`, then the return result should be a number.

Examples:

```
> (get-value '((num 3 1) (num 4 2) (num 0.0001 0.0)))
'(3 4 0.0001))
> (get-value (num 10 2))
10
```

3.2 (get-grad *num-list*)

This procedure will return the gradient field of numbers in `num-list` as a list. If `num-list` is a `num`, then the return result should be a number.

Examples:

```
> (get-grad '((num 3 1) (num 4 2) (num 0.0001 0.0)))
'(1 2 0.0))
> (get-grad (num 10 2))
2
```

4 Operators

You will extend `+`, `-`, and `*` operators so that they will also compute the gradients of their arguments. These operators will return a `num` struct. There will be normal return values of these operators in the `value` field of the number. In the `grad` field, there will be the derivative of the operator with respect to all of its arguments. So, given two numbers $< v_1, g_1 >$, $< v_2, g_2 >$, an operator f will return:

$$< f(v_1, v_2), \frac{\partial f}{\partial v_1}(v_1, v_2) + \frac{\partial f}{\partial v_2}(v_1, v_2) > \tag{1}$$

Note that there may be more than two numbers for `+`, `-`, and `*` (see the examples below). If you feel like you cannot do it for n terms, just implement it for two terms for partial credit.

4.1 (add *num1 num2* ...)

The derivative of `add` operator is the sum of gradients of its terms. This is rather easy, for the value, you will sum the values of numbers, and for the gradient, you will sum the gradients.

Examples:

```
> (add (num 5 1) (num 2 0.2))
(num 7 1.2)
> (add (num 5 1) (num 4 3) (num 6 2))
(num 15 6)
```

In these results, the first field is the value, and the second field is the gradient.

4.2 (`mul` *`num1`* *`num2`* `...`)

For the value, you will multiply all the numbers' values. For the gradient, the derivative of `mul` operator is the sum of multiplication of each term's gradient with others' values. For example:

$$mul(x, y, z, t) = xyzt \tag{2}$$

$$\sum_{arg} \frac{\partial mul}{\partial arg}(x, y, z, t) = x'yzt + xy'zt + xyz't + xyzt' \tag{3}$$

Hint: you can implement this recursively for n arguments. Again, if you have a hard time, implement it for only two arguments for partial credit. That would be rather easy.

Examples:

```
> (mul (num 5 1) (num 2 1))
(num 10 7)
> (mul (num 5 1) (num 4 3) (num 6 2))
(num 120 154)
```

4.3 (`sub` *`num1`* *`num2`*)

For the value, you will subtract *`num2`* from *`num1`*. For the gradient, the derivative of `sub` is the subtraction of *`num2`*'s gradients from *`num1`*'s gradients.

Examples:

```
> (sub (num 5 1) (num 2 1))
(num 3 0)
> (sub (num 5 0) (num 2 1))
(num 3 -1)
```

4.4 `relu` and `mse`

After you implement the primitive operations, you no longer need to implement the derivative for other stuff. You can basically calculate almost everything[1] using `+` and `*`. I have implemented rectified linear unit (ReLU, $f(x) = max(x, 0)$) and mean square error function (mse, $f(x, y) = (x - y)^2$) for you to use in the proceeding procedures.

[1] For example, you can implement Babylonian square-root with summation and division (we omitted the division here but you get the idea), or you can implement cosine using Taylor expansion, and so on.

```
(define relu (lambda (x) (if (> (num-value x) 0) x (num 0.0 0.0))))
(define mse (lambda (x y) (mul (sub x y) (sub x y))))
```

5 Higher-level procedures

In this part, you will implement some high-level helper functions that will help you parse the input and transform it to a valid expression, so that everything can be automated.

5.1 (create-hash *names values var*)

Given a list of symbols in *names*, respective values in *values*, and variable at interest in *var*, you will create a hash which contains numbers for each symbol with appropriate gradients. Here, initial gradients of each number will be 0 except for *var*, for which it will be 1.

Examples:

```
> (create-hash '(a b c d) '(1 3 3 7) 'b)
'#hash((a .  (num 1 0.0)) (b .  (num 3 1.0)) (c .  (num 3 0.0)) (d .  (num 7 0.0)))
> (create-hash '(a b c d) '(1 3 3 7) 'c)
'#hash((a .  (num 1 0.0)) (b .  (num 3 0.0)) (c .  (num 3 1.0)) (d .  (num 7 0.0)))
```

This way, we will keep track of variables that we want to take the gradient. Numbers will 0 gradient will be automatically treated as constants.

5.2 (parse *hash expr*)

In this procedure, you will implement a simple parser which will translate the symbolic expression in *expr* into an appropriate expression by converting symbols in *expr* into their respective numbers in *hash*. For example, if we see `x` in *expr*, we will convert it into its number in *hash*. Operations in *expr* can be one of the following: `+`, `*`, `-`, `mse`, `relu`. Here, you have to convert `*` to `mul`, and so on. If you encounter a number in *expr*, you will construct the number with 0 gradient.

Examples:

```
> (parse '#hash((x .  (num 10 .  1.0)) (y .  (num 20 0.0))) '(+ x y))
'(add (num 10 1.0) (num 20 0.0))
> (parse (create-hash '(x y) '(10 20) 'x) '(+ x y))
'(add (num 10 1.0) (num 20 0.0))
> (eval (parse '#hash((x .  (num 10 .  1.0)) (y .  (num 20 0.0))) '(+ x y)))
(num 30 1.0)
> (parse (create-hash '(x y) '(10 20) 'x) '(+ (* (+ x y) x) (+ y x 5)))

'(add
  (mul (add (num 10 1.0) (num 20 0.0)) (num 10 1.0))
  (add (num 20 0.0) (num 10 1.0) (num 5 0.0)))
```

Here is a pseudocode to ease your work:

1. If the expression is an empty list, return empty list.
2. If the expression is a list, apply `parse` to the element of the list, apply `parse` to the rest of the list, and concatenate these two results.
3. If the expression is is one of these: `+`, `*`, `-`, `mse`, `relu`, convert it to one of these: `'add`, `'mul`, `'sub`, `'mse`, `'relu`.
4. If the expression is a number, construct a number struct with 0 gradient.
5. Else, it should be a symbol in our hash. Retrieve the number from the hash.

5.3 (`grad` *names values var expr*)

Given a list of symbols in *names* with respective values in *values*, take partial derivative of the *expr* with respect to *var* and return it. If you have implemented all the other parts, this procedure should be easy; you have all the tools you need.

Examples:

```
> (grad '(x y) '(10 20) 'x '(+ (* (+ x y) x) (+ y x)))
41.0
> (grad '(x) '(7) 'x '(* x x))
14.0
```

5.4 (`partial-grad` *names values vars expr*)

This is similar to `grad` except now we return a list of partial derivatives of *expr* with respect to each symbol in *names*. Here, if a symbol in *names* is not in *vars*, we treat it as a constant and its gradient should be 0. It should be easy to implement this procedure using `grad` (hint: for all elements in *name*, take `grad` if it is in *vars*, otherwise fill with 0).

Examples:

```
> (partial-grad '(x y) '(10 20) '(x) '(+ (* (+ x y) x) (+ y x)))
'(41.0 0.0)
> (partial-grad '(x y) '(10 20) '(y) '(+ (* (+ x y) x) (+ y x)))
'(0.0 11.0)
> (partial-grad '(x y) '(10 20) '(x y) '(+ (* (+ x y) x) (+ y x)))
'(41.0 11.0)
> (partial-grad '(x y) '(10 20) '(x y) '(+ 1 2))
'(0.0 0.0)
```

5.5 (`gradient-descent` *names values vars lr expr*)

We are approaching the end. Now that we have everything in our toolbox, we can start optimizing our differentiable parameters (i.e. parameters that are in *vars*) to minimize an objective. In this procedure, you will implement a single iteration of the following equation for our parameter set:

$$values = values - lr * \nabla_{values}(\texttt{expr}) \tag{4}$$

Here, we change values of *names* in the direction that minimizes the value of *expr* [2]. If you have successfully implemented `partial-grad`, what you will return is just *values* - *lr* * `partial-grad`(*names*, *values*, *vars*, *expr*) (of course, in a functional programming style).

Examples:

```
> (gradient-descent '(x y) '(10 20) '(x) 0.1 '(+ x y))
'(9.9 20.0) ;; x is updated to minimize x+y
> (gradient-descent '(x y) '(10 20) '(x y) 0.1 '(+ x y))
'(9.9 19.9) ;; now both are updated to minimize x+y since both are in vars
> (gradient-descent '(x y) '(10 20) '(x y) 0.001 '(+ x y))
'(9.999 19.999) ;; a smaller learning rate (lr)
> (gradient-descent '(x y) '(10 20) '(x) 0.1 '(mse x y))
'(12.0 20.0) ;; now we try to bring x closer to y
```

5.6 (optimize *names values vars lr k expr*)

In this procedure, you will run `gradient-descent` iteratively *k* times. That is, the returning result of `gradient-descent` call should be the new *values*. In another words, you will call `gradient-descent` recursively *k* times and at each iteration you will its result for `values` in the next call. Here is an example for *k*=2:

```
(gradient-descent names (gradient-descent names vars lr expr) vars lr expr)
```

Examples:

```
> (optimize '(x y) '(10 20) '(x) 0.1 100 '(+ x y))
'(1.8790524691780774e-14 20.0) ;; x is something closer to 0
> (optimize '(x y) '(10 20) '(x y) 1 5 '(+ x y))
'(5.0 15.0)
;; 1000 iteration might take some time
> (optimize '(x y) '(10 20) '(x) 0.01 1000 '(mse x y))
'(19.99999998317034 20.0) ;; don't have to be exact
> (optimize '(x y w1 w2 b) '(1 20 0.1 0.3 5) '(x) 0.001 1000
'(mse (+ (* x x w1) (* x w2) b) y))
'(10.83825591226222 20.0 0.1 0.3 5.0)
;; check for 0.1x^2 + 0.3x + 5 = 20
```

6 Submission

You will submit two files: `autodiff.rkt`, `feedback.txt`. Your code should be in one file named `autodiff.rkt`. First four lines of your `autodiff.rkt` file must have exactly the lines below since it will be used for compiling and testing your code automatically:

[2] We effectively change values of *vars* since gradient of other variables will be zero, but it will be more easy to use *names* with `partial-grad`

```
% name surname
% student id
% compiling: yes
% complete: yes
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether you completed all of the project, which must be `no` if you're doing a partial submission. This whole part must be lowercase and include only the English alphabet. Example:

```
% alper ahmetoglu
% 2012400147
% compiling: yes
% complete: yes
```

We are interested in your feedback about the project. In the `feedback.txt` file, please write your feedback. This is optional, you may leave it empty and omit its submission.

7 Prohibited Constructs

The following language constructs are ***explicitly prohibited***. You *will not get any points* if you use them:

1. Any function or language element that ends with an `!`.
2. Any of these constructs: `begin`, `begin0`, `when`, `unless`, `for`, `for*`, `do`, `set!-values`.
3. Any language construct that starts with `for/` or `for*/`.
4. Any construct that causes any form of mutation [3] (impurity).

8 Tips and Tricks

- You can use higher-order functions like `map`, `folder`, `foldl`, `foldr`. You are also encouraged to use anonymous functions with the help of `lambda`.
- You can use Racket reference, either from DrRacket's menu: Help ¿ Racket Documentation, or from the following link `https://docs.racket-lang.org/reference/index.html`.
- A useful link for the difference between `let`, `let*`, `letrec`, and `define`: `https://stackoverflow.com/questions/53637079/when-to-use-define-and-when-to-use-let-in-racket`.

[3] Mutation means creating any kind of change, either via changing value of a variable, a memory slot, or an element of a container; or via doing input/output.

References

[1] Automatic Differentiation. `https://en.wikipedia.org/wiki/Automatic_differentiation`.

[2] Chain rule. `https://en.wikipedia.org/wiki/Chain_rule`.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.