Write Loops Like Einstein

Using Einstein Summation Notation for Array Operations in Lisp

Bibek Panthi

November 20, 2020

Contents

1 Introduction

My target is to be able to write operations on vector, matrices, tensors directly in Einstein summation notation instead of loops.

```
;; d_i = L_k V^k_i [ 1 - (s^i)^2]
(loop for i from 0 below (length di) do
  (setf (aref d i)
        (* (- 1 (expt (aref s i) 2))
           (loop for k from 0 below (array-dimension V 1)
                 summing (* (aref (aref dL/do time) k)
                            (aref V k i))))))
```

In the end we should be able to replace above expression with:

```
(einsum (ik :to i)
        :into d
        (k L) (ki V) (- 1 (expt (i s) 2)))
```

We sum over the product of given terms such that indices 'i' & 'k' are reduce to just 'i' and the result is stored in 'd'. Notice how similar is the new code to the comment in the original code which describes the mathematics.

Inside the expressions the indices are associated with tensors using function call notation. These function are purely syntactical and are replaced by corresponding 'aref' on the arrays [1].

2 Einsum

```
(defmacro einsum ((rhs-indices to lhs-indices) &rest expr)
  "Expand expressions in Einstein's summation notation to loops"
  (assert (eql to :to))
  (if (eql (first expr) :into)
      (progn
        (assert (symbolp (second expr)))
        (apply #'einsum% (string rhs-indices) (string lhs-indices) (second expr) (cddr expr)))
      (apply #'einsum% (string rhs-indices) (string lhs-indices) nil expr)))
```

Inside the 'einsum' macro we only handle the syntactic structure and use 'einsum%' function for bulk of the source code transformation. Also using function makes for easier debuging experience and allows testing from bottom up approach.

3 Overview

Let see at the final result to get a general feel of how 'einsum' would work.

```
(macroexpand-1 '(einsum (ik :to i)
                        :into d
                        (k L) (ki V) (- 1 (expt (i s) 2))))
```

```
(LET ((#:|I-max754| (ARRAY-DIMENSION S 0)) (#:|K-max755| (ARRAY-DIMENSION V 0)))
  (ASSERT (= (ARRAY-DIMENSION S 0) (ARRAY-DIMENSION V 1)))
  (ASSERT (= (ARRAY-DIMENSION V 0) (ARRAY-DIMENSION L 0)))
  (LET ()
    (LOOP FOR #:I756 FROM 0 BELOW #:|I-max754|
          FOR #:|const-expr-758| = (- 1 (EXPT (AREF S #:I756) 2))
          DO (SETF (AREF D #:I756)
                   (* #:|const-expr-758|
                      (LOOP FOR #:K757 FROM 0 BELOW #:|K-max755|
                            FOR #:|const-expr-759| = (* (AREF L #:K757)
                                                        (AREF V #:K757
                                                              #:I756))
                            SUMMING #:|const-expr-759|))))
    D))
T
```

We can break down the above expanded code into few parts:

- first variable to store array dimesions are created
- assertions are added to check that the sizes of the arrays match up
- a vector with fill pointer is created that would store the result (if an result-array is provided, this vector is displaced to (i.e. points to) that array)

[1] Array in Common Lisp can be multi-dimensional. So in mathematical terms they can be vectors, matrices, or n-order tensors.

- loops over the indices are created. (looping variable are named '-dim-')
  - outer loops are over the output indices (in this example there is a single outer loops)
  - in the innermost portion of the outer loop a '(setf (aref ...)' is used to store the sums into the result array
  - the sums are computed in inner-loops (in this example there is only one inner loop)
  - expression which can be taken out of the loops are computed outside loops and stored in 'const-expr-' variable

4 Index Variables - Dynamic Extend

```
(defun einsum% (input-indices output-indices result-array &rest expr)
  (declare (optimize (debug 3)))
  (let* ((*indices* (map 'list #'identity input-indices))
         (*indices-max* (mapcar (lambda (i)
                                  (gensym (concatenate 'string (string i) "-max")))
                                *indices*))
         (*indices-vars* (mapcar (lambda (i) (gensym (string i))) *indices*))
```

We set the indices, variables to be used in loops (e.g. '#:|I-dim1252|' [2]) and variable to store the dimensions of indices (eg. '#:|I-max1262|')

Values that need to shared between functions are stored in special variables[3] and are bound inside the 'einsum%' function.

A 'defparameter' in the top-level of the file makes those variable 'special'. Another approach could be to using '(declare (special ...))' but I choose the former style for easier testing and debugging.

```
(defparameter *indices* nil)
(defparameter *indices-vars* nil)
(defparameter *indices-max* nil)
(defparameter *constant-indices* nil)
```

```
(defun index-max (index)
  "Returns gensymed variable used to denote dimension of 'index'"
  (nth (position index *indices*) *indices-max*))

(defun index-var (index)
  "Returns gensymed variable used as looping variable of 'index'"
  (nth (position index *indices*) *indices-vars*))

(defun index-function? (symbol)
  "Checks if the 'symbol' could be name of a indexing function"
  (loop for char across (symbol-name symbol) do
        (unless (find char *indices*)
          (return nil))
        finally (return t)))
```

5 Array Dimensions

```
(defun einsum% (input-indices output-indices result-array &rest expr)
  (declare (optimize (debug 3)))
  (let* ((*indices* (map 'list #'identity input-indices))
```

[2] Yes! These are variable names (i.e. symbols). (They are randomly created (using 'gensym') during macroexpansion so that their names don't clash with other variables) (Also since they are randomly generated, the exact number at the end may not match within this document)

[3] Special Variables are variabes with dynamic extend. i.e. they can be assessed inside the other function without the need to be passed along with the function.

```
        (*indices-max* (mapcar (lambda (i)
                                 (gensym (concatenate 'string (string i) "-max")))
                               *indices*))
        (*indices-vars* (mapcar (lambda (i) (gensym (string i))) *indices*))

        (dimensions (walk-for-dimensions (cons '* expr)))
        (result (if result-array result-array (gensym "result"))))

    ;; assign max-vars to size of indices
    `(let (,@(loop for index in *indices*
                   for max-var = (index-max index)
                   for dim = (find index dimensions :key #'first)
                   collect `(,max-var (array-dimension ,(second dim) ,(third dim))))))

      ;; dimension assertions
      ,@(dimension-assertions dimensions)
```

'walk-for-dimensions' walks over the given expressinon to find the arrays, their shape and sizes. For example:

```
(let ((*indices* '(#\I #\K)))
  (walk-for-dimensions '(* (k L) (ki V) (- 1 (expt (j s) 2)))))
```

```
((#\I V 1) (#\K V 0) (#\K L 0))
```

This return value means that index 'i' is the 2nd dimensions of V, index K is the first dimension of V and L.

```
(defun walk-for-dimensions (expr &optional results)
  "Look at 'expr' and find out the which index corresponds to which dimension of which tensor
returns list of (index tensor axis)"
  (cond ((or (atom expr) (not *indices*)) nil)
        ((and (listp expr)
              (index-function? (first expr))
              (= (length expr) 2)
              (symbolp (second expr)))
         (loop for char across (symbol-name (first expr))
               for i from 0 do
                 (pushnew (list char (second expr) i)
                          results :test #'equal)))
        (t (loop for subexpr in expr do
          (setf results (walk-for-dimensions subexpr results)))))
  results)

(defun dimension-assertions (dimensions)
  "return assert forms; 'dimesions' is a list of (index tensor axis)"
  (loop for index in *indices*
        when (> (count index dimensions :key #'first) 1)
          collect `(assert (= ,@(remove-if #'not
                                           (mapcar (lambda (dims)
                                                     (if (eql (first dims) index)
                                                         `(array-dimension ,(second dims)
                                                                           ,(third dims))))
                                                   dimensions))))))
```

6 Allocate Result array

If the array to store the results ('result-array') is provided then new result array is not created otherwise a new array is allocated with size given by product of the dimensions of the output-indices.

```
;; allocate resulting array or reuse given array
(let ,(unless result-array
```

```
            `((,result ,(if output-indices
                            `(make-array (* ,@(map 'list #'index-max output-indices)))))))
```

7 Loops

7.1 Loop Over

To write the outer and inner loops with ease, a helper function 'loop-over' is defined as follows:

```
(defun loop-over% (index expr then-function constant-product)
  (let ((*constant-indices* (cons index *constant-indices*)))
    (multiple-value-bind (const-expr remaining-expr) (extract-constant-expr expr)
      (if const-expr
          (let ((var (gensym "const-expr-")))
            `(loop for ,(index-var index) from 0 below ,(index-max index)
                   for ,var = ,(if constant-product
                                   (expand-arefs `(* ,@const-expr ,constant-product))
                                   (if (= (length const-expr) 1)
                                       (expand-arefs (first const-expr))
                                       (expand-arefs `(* ,@const-expr))))
                   ,@(funcall then-function remaining-expr var)))
          `(loop for ,(index-var index) from 0 below ,(index-max index)
                 ,@(funcall then-function remaining-expr constant-product))))))

(defun loop-over (index &key checking-constants-in then with-constant)
  "Return a loop form taking care of any expression in `checking-constants-in'
 that can be taken out of the loop
 (loop for index-var from 0 below index-max
       for new-constant = (* with-constant ...)
     ,@(then remaining-expr new-constant)"
  (loop-over% index checking-constants-in then with-constant))
```

As an example see this:

```
(let* ((input-indices "IK")
       (*indices* (map 'list #'identity input-indices))
       (*indices-max* (mapcar (lambda (i)
                                (gensym (concatenate 'string (string i) "-max")))
                              *indices*))
       (*indices-vars* (mapcar (lambda (i) (gensym (string i))) *indices*)))
  (loop-over #\I
             :checking-constants-in '((k L) (ki V) (- 1 (expt (i s) 2)))
             :then (lambda (remaining-expr const)
                     `(do (print ,remaining-expr ,const)))
             :with-constant nil))
```

```
(LOOP FOR #:|I-dim1312| FROM 0 BELOW #:|I-max1313|
      FOR #:|const-expr-1311| = (- 1 (EXPT (AREF S #:|I-dim1314|) 2))
      DO (PRINT ((K L) (KI V)) #:|const-expr-1311|))
```

'loop-over' was smart enough to identify that the expression '(- 1 (EXPT (AREF S #:|I-dim1314|) 2))' is constant for given 'i' so it stored that in a variable name '#:|const-expr-1311|' and passed the remaining expressions and the name of this variable to the ':then' function.

In this 'loop-over' function few other small functions are used that perform small and easy tasks.

7.1.1 Utilites used

```
(defun expand-arefs (expr)
  "Repalce indexing functions with aref in the expression `expr'"
  (cond ((atom expr)
```

```
          expr)
         ((and (listp expr)
               (index-function? (first expr))
               (= (length expr) 2))
          `(aref ,(second expr) ,@(loop for index across (symbol-name (first expr))
                                        collect (index-var index))))
         ((listp expr)
          (mapcar (lambda (e)
                    (expand-arefs e))
                  expr))
         (t expr)))

(defun constant-expr? (e)
  "Returns true if expression `e' is constant when indices in `*constant-indices*' are given"
  (cond ((atom e) t)
        ((and (listp e)
              (index-function? (first e)))
         (loop for i across (symbol-name (first e)) do
           (if (not (find i *constant-indices*))
               (return nil))
               finally (return t)))
        ((listp e)
         (every (lambda (e)
                  (constant-expr? e))
                (rest e)))
        (t t)))

(defun extract-constant-expr (expr)
  "Return constant and non-constant parts in `expr' under given `*constant-indices*'"
  (let* ((non-constant-expr (remove-if (lambda (e)
                                         (constant-expr? e))
                                       expr))
         (constant-expr (set-difference expr non-constant-expr)))
    (values constant-expr non-constant-expr)))
```

An exmple of 'extract-constant-expr' in action:

```
(let ((*indices* '(#\I #\K))
      (*constant-indices* '(#\I)))
  (extract-constant-expr '((k L) (ki V) (- 1 (expt (i s) 2)))))
```

```
((- 1 (EXPT (I S) 2)))
((K L) (KI V))
```

An example of 'expand-arefs' in action

```
(let ((*indices* '(#\I #\K)))
    (expand-arefs '(* (k L) (ki V) (- 1 (expt (i s) 2)))))
```

```
(* (AREF L #:|K-dim1394|) (AREF V #:|K-dim1395| #:|I-dim1396|)
   (- 1 (EXPT (AREF S #:|I-dim1397|) 2)))
```

7.2 Outer and Inner loops

Now that we have the convenient loop-over function we can use it to generate the outer and inner loops inside 'einsum%'

```
;; now loop!! :)
,(labels ((outer-loop (indices expr const)
            (loop-over
             (first indices)
             :with-constant const
             :checking-constants-in expr
             :then
```

```
                (lambda (remaining-expr const)
                  (if (> (length indices) 1)
                      `(do ,(outer-loop (rest indices)
                                        remaining-expr
                                        const))
                      (cond
                        ((and const remaining-expr)
                         `(do (setf (aref ,result ,@(map 'list #'index-var output-indices))
                                    (* ,const ,(inner-loop remaining-expr)))))
                        (remaining-expr
                         `(do (setf (aref ,result ,@(map 'list #'index-var output-indices))
                                    ,(inner-loop remaining-expr))))
                        (const
                         `(do (setf (aref ,result ,@(map 'list #'index-var output-indices))
                                    ,const)))))))

          (inner-loop% (indices expr const)
            (loop-over
             (first indices)
             :checking-constants-in expr
             :with-constant const
             :then
             (lambda (remaining-expr const)
               (if (> (length indices) 1)
                   (if const
                       `(summing (* ,const ,(inner-loop% (rest indices)
                                                         remaining-expr nil)))
                       `(summing ,(inner-loop% (rest indices)
                                               remaining-expr nil)))
                   (cond
                     ((and const remaining-expr)
                      `(summing (* ,const ,(expand-arefs `(* ,@remaining-expr)))))
                     (remaining-expr
                      `(summing ,(expand-arefs `(* ,@remaining-expr))))
                     (const
                      `(summing ,const)))))))

          (inner-loop (expr)
            (inner-loop% (set-difference *indices* (map 'list #'identity output-indices))
                         expr nil)))
    (outer-loop (map 'list #'identity output-indices)
                expr
                nil))
```

The 'outer-loop' recursive create a loop form for each outer index. In the ':then' argument if the length of outer indices is greated that 1 i.e. when some outer loops are still to be created 'outer-loop' recursively calls itself. Otherwise, it returns a 'do' clause with a '(setf (aref result-array indices...) sums)' operation and calls 'inner-loop' to calculate the sums.

The 'inner-loop' computes the indices that need to be summed over (which is the set difference of all indices with the output-indices). Then calls 'inner-loop%' which, similar to how 'outer-loop' function operates, creates the inner loops recursively summing the product till the end.

8 Return Result

Finally the resulting array is returned.

```
;; return the results
,(or result-array result)))))
```

Stiching all the pieces together we have the 'einsum%' function:

```
(defun einsum% (input-indices output-indices result-array &rest expr)
  (declare (optimize (debug 3)))
  (let* ((*indices* (map 'list #'identity input-indices))
         (*indices-max* (mapcar (lambda (i) (gensym (concatenate 'string (string i) "-max"))) *indices*))
         (*indices-vars* (mapcar (lambda (i) (gensym (string i))) *indices*))
```

```
6
7           (dimensions (walk-for-dimensions (cons '* expr)))
8           (result (if result-array result-array (gensym "result"))))
9
10      ;; assign max-vars to size of indices
11      `(let (,@(loop for index in *indices*
12                     for max-var = (index-max index)
13                     for dim = (find index dimensions :key #'first)
14                     collect `(,max-var (array-dimension ,(second dim) ,(third dim)))))
15
16        ;; dimension assertions
17        ,@(dimension-assertions dimensions)
18
19        ;; allocate resulting array or reuse given array
20        (let ,(unless result-array
21                `((,result ,(if output-indices
22                                `(make-array (* ,@(map 'list #'index-max output-indices)))))))
23          ;; now loop!! :)
24          ,(labels ((outer-loop (indices expr const)
25                      (loop-over
26                       (first indices)
27                       :with-constant const
28                       :checking-constants-in expr
29                       :then
30                       (lambda (remaining-expr const)
31                         (if (> (length indices) 1)
32                             `(do ,(outer-loop (rest indices)
33                                               remaining-expr
34                                               const))
35                             (cond
36                               ((and const remaining-expr)
37                                `(do (setf (aref ,result ,@(map 'list #'index-var output-indices))
38                                           (* ,const ,(inner-loop remaining-expr)))))
39                               (remaining-expr
40                                `(do (setf (aref ,result ,@(map 'list #'index-var output-indices))
41                                           ,(inner-loop remaining-expr))))
42                               (const
43                                `(do (setf (aref ,result ,@(map 'list #'index-var output-indices))
44                                           ,const)))))))))
45
46                    (inner-loop% (indices expr const)
47                      (loop-over
48                       (first indices)
49                       :checking-constants-in expr
50                       :with-constant const
51                       :then
52                       (lambda (remaining-expr const)
53                         (if (> (length indices) 1)
54                             (if const
55                                 `(summing (* ,const ,(inner-loop% (rest indices)
56                                                                   remaining-expr nil)))
57                                 `(summing ,(inner-loop% (rest indices)
58                                                         remaining-expr nil)))
59                             (cond
60                               ((and const remaining-expr)
61                                `(summing (* ,const ,(expand-arefs `(* ,@remaining-expr)))))
62                               (remaining-expr
63                                `(summing ,(expand-arefs `(* ,@remaining-expr))))
64                               (const
65                                `(summing ,const)))))))
66
67                    (inner-loop (expr)
68                      (inner-loop% (set-difference *indices* (map 'list #'identity output-indices))
69                                   expr nil)))
70              (outer-loop (map 'list #'identity output-indices)
71                          expr
72                          nil))
73          ;; return the results
74          ,(or result-array result)))))
```

9 Lets see usecases

For use in following examples lets define a matrix and two vector

```
(defparameter M #2A((1 2 3) (4 5 6)))
(defparameter V (vector 10 2.2 3e3))
(defparameter U (vector 9 4))
```

9.1 Transpose

```
(defun transpose (matrix)
  (destructuring-bind (m n) (array-dimensions matrix)
    (let ((new-matrix (make-array (list n m))))
      (loop for i from 0 below m do
        (loop for j from 0 below n do
          (setf (aref new-matrix j i)
                (aref matrix i j))))))
```

```
      new-matrix)))

(transpose M)
```

```
#2A((1 4) (2 5) (3 6))
```

Equivalent einsum expression

```
(einsum (ij :to ji) (ij M))
```

```
#2A((1 4) (2 5) (3 6))
```

9.2 Outer Product

```
(defun incf-outer-product (place vec-a vec-b)
  "Add the outer product of `vec-a' and `vec-b' into `place'"
  (let ((n (array-dimension place 0))
        (m (array-dimension place 1)))
    (assert (= n (length vec-a)))
    (assert (= m (length vec-b)))
    (loop for i from 0 below n do
      (loop for j from 0 below m do
        (incf (aref place i j)
              (* (aref vec-a i)
                 (aref vec-b j)))))
    place))

;; M^i_j += U^i V_j
(let ((M (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))))
  (incf-outer-product M
                      U V))
```

```
#2A((91 21.800001 27003.0) (44 13.8 12006.0))
```

Equivalent einsum expression:

```
(let ((M (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))))
  (einsum (ij :to ij) :into M
          (+ (ij M) (* (i U) (j V)))))
```

```
#2A((91 21.800001 27003.0) (44 13.8 12006.0))
```

Lets see the code generated by einsum:

```
(macroexpand '(einsum (ij :to ij) :into M
               (+ (ij M) (* (i U) (j V)))))
```

```
(LET ((#:|I-max730| (ARRAY-DIMENSION U 0)) (#:|J-max731| (ARRAY-DIMENSION V 0)))
  (ASSERT (= (ARRAY-DIMENSION U 0) (ARRAY-DIMENSION M 0)))
  (ASSERT (= (ARRAY-DIMENSION V 0) (ARRAY-DIMENSION M 1)))
  (LET ()
    (LOOP FOR #:I732 FROM 0 BELOW #:|I-max730|
          DO (LOOP FOR #:J733 FROM 0 BELOW #:|J-max731|
                   FOR #:|const-expr-734| = (+ (AREF M #:I732 #:J733)
                                               (* (AREF U #:I732)
                                                  (AREF V #:J733)))
                   DO (SETF (AREF M #:I732 #:J733) #:|const-expr-734|)))
    M))
T
```

Both the handwritten code and generated code are similar.

9.2.1 Lets see the timing.

```
(time (let ((M (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))))
                (loop repeat 10000000 do
                      (incf-outer-product M U V))
                M))
```

```
Evaluation took:
  2.090 seconds of real time
  2.088960 seconds of total run time (2.088960 user, 0.000000 system)
  99.95% CPU
  4,814,051,628 processor cycles
  0 bytes consed

#2A((900000001 1.7338182e8 2.9209824e11) (400000004 8.1764024e7 1.046401e11))
```

einsum:

```
(time (let ((M (make-array '(2 3) :initial-contents '((1 2 3) (4 5 6)))))
              (loop repeat 10000000 do
                (einsum (ij :to ij) :into M
                        (+ (ij M) (* (i U) (j V)))))
              M))
```

```
Evaluation took:
  2.020 seconds of real time
  2.023754 seconds of total run time (2.023754 user, 0.000000 system)
  100.20% CPU
  4,663,633,418 processor cycles
  0 bytes consed

#2A((900000001 1.7338182e8 2.9209824e11) (400000004 8.1764024e7 1.046401e11))
```

9.3 Transpose then dot

```
(defun matrix-T-dot-vector (matrix vector)
  "Multiply transpose of `matrix` with `vector`"
  (destructuring-bind (m n) (array-dimensions matrix)
    (assert (= m (length vector)))
    (let ((result (make-array n)))
      (loop for j from 0 below n do
        (setf (aref result j)
              (loop for i from 0 below m
                    summing (* (aref matrix i j)
                               (aref vector i)))))
      result)))

(matrix-T-dot-vector M U)
```

```
#(25 38 51)
```

```
(einsum (ij :to j)
        (ij M) (i U))
```

```
#(25 38 51)
```

9.4 Sth complicated

```
;; d_i = d_m, prev W^m_i [ (1 - (s^i)^2)]
(let ((d (vector 1 2 3))
      (s (vector 9 8 4))
      (W #2A((1 2 3) (3 4 5) (5 6 7))))

  (map-into d
            (lambda (dW_i s^i)
              (* dW_i (- 1 (expt s^i 2))))
            (matrix-t-dot-vector W d)
            s))
```

```
#(-1760 -1764 -510)
```

```
;; d_m = d_m, prev W^m_i [ (1 - (s^i)^2)]
(let ((d (vector 1 2 3))
      (s (vector 9 8 4))
      (W #2A((1 2 3) (3 4 5) (5 6 7))))

  (einsum (im :to i)
          (* (m d) (mi W) (- 1 (expt (i s) 2)))))
```

```
#(-1760 -1764 -510)
```

Notice how the einsum expression is similar to the mathematical notation given in the comment.