Assignment 5

CAS CS 320: Concepts of Programming Languages

Due Thursday 3/5 by 8:00PM

1 Programming (100%)

This week we're building a calculator. So far, we've only built calculators for evaluting expressions over *scalars*. But ML/AI folks are all about *tensors*, and for good reason, they're pretty darn useful for things like deep learning.[1] It's valuable to know how they work. For all the folks interested in ML/AI, only a small percentage (at least this is my sense) know how to use Einstein summation notation (Einsums). Understanding how Einsums work is ultimately about understanding how to *interpret* certain kinds of expressions, and this is the purview of PL. So, we're going to be building a Einsum calculator. Some further motivation:

- I believe one of the "superpowers" of knowing how PLs work is being able to gain a deeper understanding of language constructs by *implementing* them. One of the best ways to verify that you understand something is to built it yourself.
- Einsums are a part of a larger area of research on tensor programming and tensor program compilation. This is a less-appreciated part of the ML/AI pipeline, the part that speeds up training while giving high-level control to the programmer.[2]
- Einsums generalize to *mapping* and *folding* operations over tensors, so we can think of Einsums as a form of higher-order programming on tensors.

What are tensors?

Tensors are a different things to different folks. We're gonna take the "CS" perspective, which is the simplest (and least interesting):

> *An n-tensor is an n-dimensional array whose shape along every dimension is uniform.*

So a 0-tensor is essentially a scalar, a 1-tensor is a vector, a 2-tensor is a matrix, a 3-tensor is a collection of matrices of the same shape, and so on. The image from Wikipedia in Figure 1 is instructive.

In this assignment, the representation of tensors is not terribly important, you'll be using an interface (in OCaml, a *module*) for working with and constructing tensors. If you're interested in the representation, see the file `tensor.ml`. In rough terms, a tensor consists of:

- a collection of entries, which in essence is an array-based S-expression with floating-point numbers (see the type `Tensor.entries`).

[1] That's right, even in PL we have to justify our existence via AI.

[2] Also worth noting a new professor who works on this just joined the department.

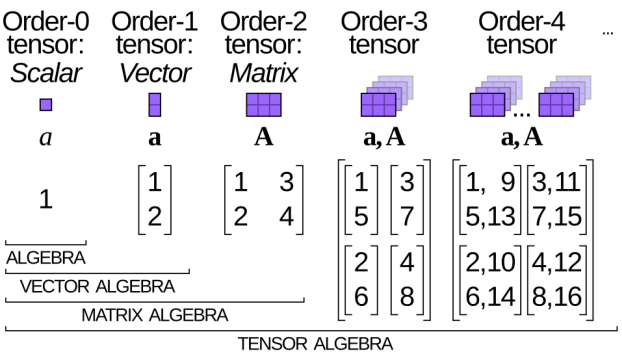


Figure 1: Visualization of tensor dimensions

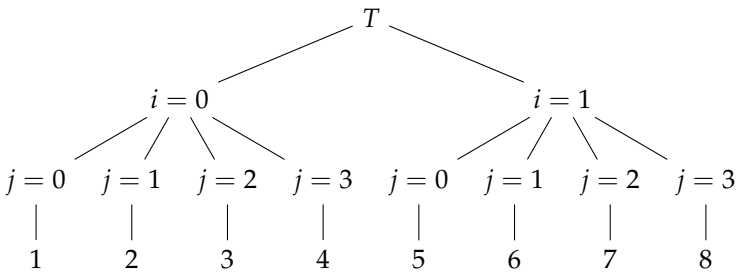
- An association list of type `(string * int) list` which describes the **index space** of the tensor, i.e., its **shape** together with labels given to each **axis**. As we'll see, it's convenient to assume axes have names. *We further require that axes have unique labels.*

The interface maintains the invariant the entries are consistent with the associated index space.

Example.

$$\begin{array}{cc} & j \\ i & \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \end{array}$$

is a 2-tensor with an index space we'll write as $(i \mapsto 2) \times (j \mapsto 4)$ (and shape we'll write as 2×4). This means that an **index** of this tensor is of the form $\{i \mapsto k_i, j \mapsto k_j\}$ where k_i and k_j are integers such that $0 \leq k_i < i$ and $0 \leq k_j < j$, e.g., the entry at index $\{j \mapsto 2, i \mapsto 1\}$ is 7. We can also visualize tensors as decisions trees, where each node is labeled with a component along an axis, and every node at a given depth has the same axis and the same number of children, e.g.,



Einsums

Einstein summation notation provides a simple mechanism for defining new tensors from old ones. It's best understood by example; we typically define matrix multiplication as follows:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

where C is the product of A and B. Informally, this notation says that the ij-th entry of C is the sum over k of the ik-th entry of A times the kj-th entry of B. Our goal it to make more formal what's going on in this notation using operations on tensors:

- When we write A_{ik}, we'll think of this as giving the 2-tensor A with shape $m \times n$ new labels on its axes, so that it has index space $(i \mapsto m) \times (k \mapsto n)$.
- Likewise, we'll think of B_{kj} as giving the 2-tensor B of shape $n \times p$ new labels on axes so that it has index space $(k \mapsto n) \times (j \mapsto p)$.
- When we multiply A_{ik} and B_{kj}, we'll think of this as producing a 3-tensor with index space

  $$(i \mapsto m) \times (j \mapsto n) \times (k \mapsto p)$$

  whose ijk-th element is the ik-th element of A times the kj-th element of B.
- We'll think of $\sum_k A_{ik}B_{kj}$ as adding together the tensors along k-th axes of $A_{ik}B_{kj}$, yielding a 2-tensor with index space $(i \mapsto m) \times (j \mapsto n)$
- We'll think of the statement $C_{ij} = \sum_k A_{ik}B_{kj}$ as defining a new tensor C which is a 2-tensor with index space

  $$(i \mapsto m) \times (j \mapsto n)$$

  This tensor will ultimately be the matrix product of A and B.

Let's do a small concrete example. Consider the following two matrices, initially given the axis labels '0' and '1'.

$$A: \quad \begin{matrix} & 1 \\ 0 & \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \end{matrix}$$

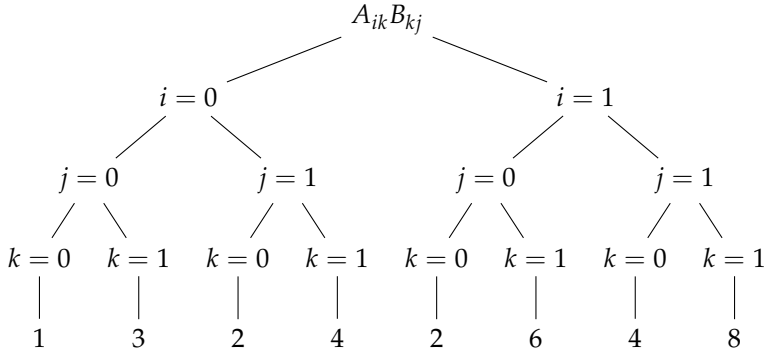
$$B: \quad \begin{matrix} & 1 \\ 0 & \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{matrix}$$

After relabeling indices, they become the following 2-tensors.

$$A_{ik}: \quad \begin{matrix} & k \\ i & \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \end{matrix}$$

$$B_{kj}: \quad \begin{matrix} & j \\ k & \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{matrix}$$

Multiplying these gives the following 3-tensor.



There two 2-tensors which are the result of fixing the k-component of $A_{ik}B_{jk}$

$$A_{ik}B_{kj} \text{ along } k = 0 : \qquad \begin{matrix} & j \\ i & \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \end{matrix}$$

$$A_{ik}B_{kj} \text{ along } k = 1 : \qquad \begin{matrix} & j \\ i & \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix} \end{matrix}$$

which can then added together entry-wise.

$$\textstyle\sum_k A_{ik}B_{kj} : \qquad \begin{matrix} & j \\ i & \begin{bmatrix} 4 & 6 \\ 8 & 12 \end{bmatrix} \end{matrix}$$

And this is, in fact, the product of A and B.

Syntax

We'll be using the following S-expression-based syntax.

<mat-lit>	::=	<float> | (<mat-lit> ... <mat-lit>)	$[\dots]$
<op>	::=	+ | *	
<expr>	::=	(<ident> (<ident> ... <ident>))	$A_{i\dots j}$
	|	(FOLD <op> <ident> <expr>)	$\sum_i E$, $\prod_i E$
	|	(MAP <op> <expr> <expr>)	$E + E'$, EE'
<stmt>	::=	(INIT <ident> (<int> ... <int>) <mat-lit>)	$C = [\dots]$
	|	(SET <ident> (<ident> ... <ident>) <expr>)	$C_{i\dots j} = E$

This grammar expresses that there are two kinds of statements. `INIT` statements initialize new tensors and are given a shape (no labels) and entries. `SET` statements create new tensors using Einsum notation. The above example can be written in this syntax as the following sequence of statements:

```
(INIT A (2 2)
  ((1 1)
   (2 2)))

(INIT B (2 2)
  ((1 2)
   (3 4)))

(SET C (i j) (FOLD + k (MAP * (A (i k)) (B (k j)))))
```

We can also construct an intermediate tensor directly:

```
(SET I (i j k) (MAP * (A (i k)) (B (k j))))
(SET C (i j) (FOLD + k (I (i j k))))
```

Which corresponds to the following Einsum statements:

$$I_{ijk} = A_{ik}B_{kj}$$
$$C_{ij} = \sum_k I_{ijk}$$

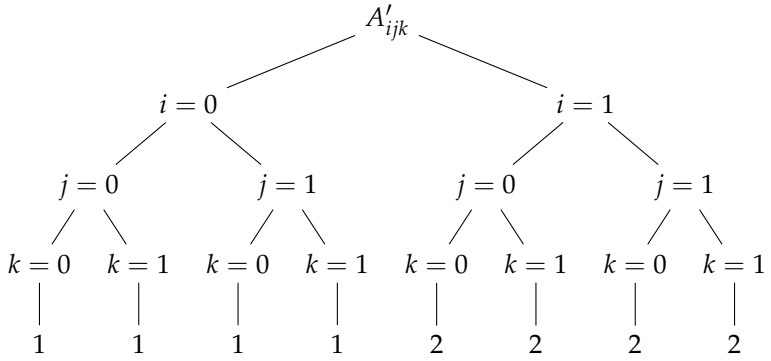
In this assignment, we've given you the parser, so you only need to understand this grammar insofar as you should be able to write examples in this language.

Aside: Why is is called `MAP`?

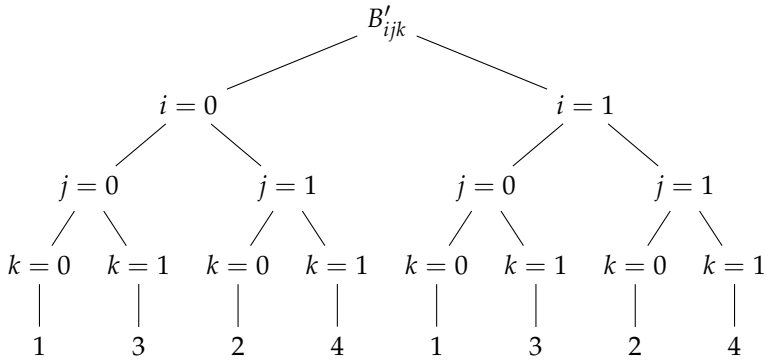
It's fairly clear why sums and products over an index are a form of folding, but based on the above example, it may not be clear why multiplying A_{ik} and B_{kj} is a form of mapping. The reason for this terminology is that the operands are implicitly **broadcasted** when we perform an operation like this. Before multiplying the tensors A_{ik} and B_{kj} above, we can imagine first converting them into 3-tensors so that they have the same index space, e.g., we could take A'_{ijk} to be the 3-tensor with index space

$$(i \mapsto m) \times (j \mapsto n) \times (k \mapsto p)$$

so that the ijk-th entry of A'_{ijk} is the ij-th entry of A for all k. In other words, we repeat the 2-tensor A_{ik} along the j-axis:



Likewise for B'_{ijk} except we repeat B_{kj} along the i-axis:



Then $A_{ik}B_{kj}$ is the same as $A'_{ijk}B'_{ijk}$ but in the latter case, it's simply a matter of multiplying two tensors element-wise, i.e., mapping the binary multiplication function over two tensors.

Tasks

In this assignment you'll be implementing dimensionality checking and evaluation of Einsums. This means implementing the following two functions (along with any auxiliary functions you need).

```
val dim_check : (string * tensor) list -> expr -> (string * int) list option
val eval : (string * tensor) list -> expr -> tensor
```

Dimensionality Checking

Not all Einsums expressions make sense to evaluate. We cannot, according to the definition of matrix multiplication, compute the tensor C_{ij} above if A and B both have shape 2×3; there are dimensionality rules we must follow in order to apply certain tensor operations. These are kind of like "typing" rules in that they can be checked before evaluating an Einsum. You'll need to implement the function:

```
val dim_check : (string * tensor) list -> expr -> (string * int) list option
```

so that `dim_check env e` is `Some idx_space` if and only if `e` is a valid expression with respect to the environment `env` (i.e., previously defined tensors) with index space `idx_space`. We can formally describe these dimensionality rules as follows.

$$\frac{A \text{ is in env. with index space } \mathcal{I}}{A : \mathcal{I}} \text{ AXIOM}$$

$$\frac{A : (j_1 \mapsto n_1) \times \cdots \times (j_k \mapsto n_k) \qquad i_1, \ldots, i_k \text{ are distinct}}{(\ A\ i_1\ \ldots\ i_k\ ) : (i_1 \mapsto n_1) \times \cdots \times (i_k \mapsto n_k)} \text{ RELABEL}$$

$$\frac{A : \mathcal{I} \qquad B : \mathcal{J} \qquad \mathcal{I} \text{ and } \mathcal{J} \text{ are consistent}}{(\ \texttt{MAP}\ o\ A\ B\ ) : \mathcal{I} \cup \mathcal{J}} \text{ MAP} \qquad \frac{A : \mathcal{I}}{(\ \texttt{FOLD}\ o\ i\ A) : \mathcal{I} \setminus i} \text{ FOLD}$$

A couple notes about these rules:

- Two index spaces are *consistent* they are the same size along axes with the same label. For example, $(i \mapsto 2) \times (k \mapsto 3)$ is consistent with $(k \mapsto 3) \times (j \mapsto 4)$.
- We write $\mathcal{I} \cup \mathcal{J}$ for the union of two consistent index spaces $\mathcal{I}$ and $\mathcal{J}$. For example,

  $$((i \mapsto 2) \times (k \mapsto 3)) \cup ((k \mapsto 3) \times (j \mapsto 4)) = (i \mapsto 2) \times (j \mapsto 4) \times (k \mapsto 3)$$

  For the purposes of this assignment, the order of the axes of the resulting index space does not matter.
- We write $\mathcal{I} \setminus i$ for $\mathcal{I}$ with the axis labeled by i removed. For example,

  $$((i \mapsto 2) \times (j \mapsto 4) \times (k \mapsto 3)) \setminus k = (i \mapsto 2) \times (j \mapsto 4)$$

  Again, for the purposes of this assignment, the order of the axes of the resulting index space does not matter.

Evaluation

Once we've checked that the expression we want to evaluate makes sense dimensionally, we need determine the actual tensor it represents. This means implementing the function:

```
val eval : (string * tensor) list -> expr -> tensor
```

so that `eval env e` is the value of `e`. The behavior of this function undefined if `e` does not pass dimensionality checking. We're intentionally leaving this one vague, you're gonna have to think about how to generalize the example given above. At a high level:

- `(` $A\ i_1\ \ldots\ i_k$ `)` should be the tensor A but with the axes relabeled.
- `(MAP` $o\ A\ B$ `)` is the result of applying the operator o element-wise to A and B after broadcasting to the union of their index spaces.[3]
- `(FOLD` $o\ i\ A$`)` is the result of folding over the i-th axes of A with the operator o.

You should put all of your programming solutions in the file `assign5/lib/assign5.ml`.

The `Tensor` Module

Finally, part of the point of this assignment is that you should not need to know how tensors are implemented. Your implementation of the above two functions should be agnostic to the representation of tensors, and should depend only of the following four functions.

- `val Tensor.idx_space : tensor -> (string * int) list` is defined so that `idx_space t` is the index space of `t`.
- `val Tensor.relabel_axes : tensor -> string list -> tensor` is defined so that the expression `Tensor.relabel_axes t labels` evaluates to `t` with the labels of its index space made to be the labels from `labels`, in the order given. The behavior of the function is undefined if the length of `labels` is not the same as the length of `idx_space t`.
- `val Tensor.get : tensor -> (string * int) list -> float` is defined so that `Tensor.get t i` is the `i`-th entry of `t`. It will raise an `Out_of_bounds` exception if `i` is not in the index space of `t`. So, for example, if `b` is defined as B_{kj} above, then `Tensor.get b [("j", 0); ("k", 1)]` is `3.`.

  Important. Note that the order of the mappings in `i` does not matter. Furthermore, `Tensor.get` is defined so that it can be applied to an index from an index space which *contains* `idx_space t` and is consistent with it. That is, if T is a tensor with index space $\mathcal{I}$, then `Tensor.get` can be used on T with any index in $\mathcal{I} \cup \mathcal{J}$, where $\mathcal{I}$ and $\mathcal{J}$ are consistent. For example, if `b` is defined as B_{kj} above, then `Tensor.get b [("j", 0); ("foo", 42); ("k", 1); ("bar"; 100)]` is also `3.`.
- `val Tensor.init : (string * int) list -> ((string * int) list -> float) -> tensor` is defined so that `Tensor.init idx_space f` is the tensor T whos `i`-th entry is `f i` for all indices `i` in the index space of T. For example, the tensor B_{kj} above can be constructed as follows:

[3] Note that this is not necessarily how you should implement it, it's just a description of the expected value.

```
let idx_space = [("k",  2); ("j", 2)] in
let mk idx =
  let k_comp = List.assoc "k" idx in
  let j_comp = List.assoc "j" idx in
  match k_comp, j_comp with
  | 0, 0 -> 1.
  | 0, 1 -> 2.
  | 1, 0 -> 3.
  | 1, 1 -> 4.
  | _ -> assert false
in
Tensor.init idx_space mk
```

You should use this function to construct intermediate tensors in the process of evaluating an Einsum expression. The behavior of this function is undefined if `idx_space` is not a valid index space (e.g., if the axes sizes are negative, or if the index labels are not distinct).

That's where we'll leave it. A couple final remarks.

- There's as much code reading in this assignment as there is code writing. Make sure you take some time to understand what each part of the code does.
- Tutorials on Einsums are all over the internet, they could be worth looking at for more perspectives and examples (e.g., this one and this one).
- If you have any questions, please ask on Piazza or come to office hours.

Happy coding.