Tensors

Linas Vepstas

26 August 2020

Abstract

This chapter defines a tensor in a general setting, extending the conventional definition of a tensor valued in a field. The definition here allows the tensor to be valued over "anything", with only minor algebraic constraints on what "anything" can be. The tensor product can then be seen as a kind of concatenation or "forgetting", while the inner product becomes a "plugging together of connectors". The definition given here is more-or-less compatible with the idea of a "tensor category", but avoids category theory. The focus is different: the focus here is concerned with knowledge representation, computer algorithms and artificial intelligence.

This is a part of a sequence of articles exploring inter-related ideas; it is meant to provide details for a broader context. The current working title for the broader text is "Connectors and Variables".

Tensors

This text reviews the concept of a tensor, starting from the conventional definition, which gives its most narrow form, and broadening it to a general setting. It is hoped that the reader already knows what a tensor is, as otherwise, most of this may seem pointless. The explanation begins very simply, at the freshman level; do not be mislead by the simplicity, there are a few tricks in here.

Three properties of tensors will be exposed and articulated. These may seem strange and idiosyncratic, if you already know what a tensor is; but communicating the strangeness is why we write.

- A means of storing data in a particular form or shape.
- The tensor product as a kind of concatenation and "forgetting".
- The inner product as a kind of "plugging together of connectors".

These properties can be taken in contrast to conventional expositions, which emphasizes multi-linearity and and behavior under change-of-basis. The change in focus is required to build the abstract notion of a tensor.[1]

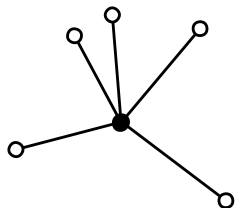
[1]There is a different abstract formalization of tensor products, that used in category theory. It is briefly reviewed in the appendix. It is far more abstract and daunting, thus we avoid it. Note also the definition of a tensor category does not include the notion of an inner product, so our focus here really is different.

Tensors as databases

The conventional definition of a tensor in science and engineering starts with the introduction of the concept of a scalar s: a single number usually taken to be a real number or a complex number or even, abstractly, taken from some ring R. A vector $v = [v_1, v_2, \cdots, v_n]$ is a sequence of numbers. A matrix $M = [M] = M_{ij} = M(i, j)$ is a square of numbers, indexed by rows and columns i, j. The equality signs here just suggest different ways (different notations) of writing the same thing. A 3-tensor $T_{ijk} = T(i, j, k)$ is a cube of numbers; a 4-tensor $T_{ijkm} = T(i, j, k, m)$ is a 4-dimensional "hypercube" of numbers, and so on.

That tensors can be used to "store data" sounds a bit silly, or perhaps painfully obvious, given the conventional definition. None-the-less, tensors are a form of "database", where "data" (the number) has been "stored" at a "location" determined by the indexes (which are taken as ordinal numbers). This becomes a slightly more interesting observation when one realizes that computer-science has a small armada of similar devices for storing data: arrays and vectors and lists and lookup tables and the like. In comp-sci, considerations such as speed, space, accessibility, mutability become important, and strongly affect algorithms. That a tensor has the form of a database becomes even more interesting when one ponders how to store extremely sparse tensors: those whose entries are mostly zero, with a scattering of non-zero entries. Consider a matrix with one million rows and one million columns: this has $10^6 \times 10^6 = 10^{12}$ locations; were we to splat a 64-bit floating-point number down at each location, this would require 8×10^{12} bytes of storage: 8 terabytes. If all of those locations were zero, except for possibly for 10^6 here or there ... that would be an insane waste of RAM: storing a million non-zero numbers requires only 8 megabytes, not 8 terabytes.

For sparse tensors, the preferred storage mechanism takes the form of a "seed", illustrated in previous sections. Lets repeat it:



The black dot at the center is the "data" (the number). The white dots are the "tensor indexes" – the ordinal-number valued tuple identifying the location of the data. The figure shows five connectors, and thus an entry in a 5-tensor.

Tensor products

We begin with the conventional definition of the tensor product, before generalizing it in a later section.

The tensor product is conventionally denoted with the "otimes" symbol $\otimes$. This intimidating symbol is used to emphasize the multi-linearity of the tensor product.

That is, if u, v and w are vectors, and a and b are constant numbers, then

$$(au+bv)\otimes w = au\otimes w + bv\otimes w$$

for the left side of the product, and likewise for the right: that this works on both sides is what makes it multi-linear. The $\otimes$ symbol is required because the Cartesian product symbol $\times$ does not work; it is not multi-linear. Consider, for example, the two ordered pairs $u\times w=(u,w)$ and $v\times w=(v,w)$. How can we multiply a scalar times an ordered pair? Conventionally, one re-scales all of the components. That is, $a.(u,w)=(a.u,a.w)$ but this fails spectacularly in terms of multi-linearity:

$$a.(u,w)+b.(v,w)=(a.u,a.w)+(b.v,b.w)=(au+bv,aw+bw)\neq(au+bv,w)$$

The tensor product $\otimes$ is not the Cartesian product $\times$. There is a way to construct the tensor product from the Cartesian product, this is developed next.

Tensor products as equivalence

The tensor product can be constructed from the Cartesian product by declaring an equivalence. From the multi-linearity property above, it's clear that we wish to say that $au\times w$ is the same thing as $u\times aw$. That is, we already have defined the tensor product so that $au\otimes w=u\otimes aw$ as strict equality; yet this is patently false for ordinary ordered pairs.

The procedure to rectify this situation is to introduce a new notion of equivalence. Using the symbol $\sim$ to denote "the same as", one writes $au\times w\sim u\times aw$. The usual laws of algebra should apply, so subtraction can be used to bring everything over to one side: $au\times w-u\times aw\sim 0$. That is, the difference of these two ordered pairs is the "same as" zero. Switching notation for ordered pairs, one may write $(au,w)-(u,aw)\sim 0$. The algebra is meant to behave "as expected", and so

$$(au,w)-(u,aw)=(au-u,w-aw)=((a-1)u,(1-a)w)\sim 0$$

But $a-1$ is just a number $c=a-1$, so the above says

$$(cu,-cw)\sim 0$$

for all numbers c. Such ordered pairs are "equivalent" to zero.

The formal way to write the tensor product is as a quotient over this equivalence. This is conventionally written as a quotient

$$U\otimes W=U\times W/\sim$$

which is meant to denote the set of all equivalent ordered pairs. In set notation,

$$u\otimes w=\{\text{all pairs }(s,t)\text{ such that }(s,t)=(u,w)+(cu,-cw)\text{ for some const }c.\}$$

This is to be read "the set of all pairs to which we have added the equivalent of zero". Formally, the set on the right is called "coset". By treating all members of this coset as "equivalent", we "forget" their identity and uniqueness (stemming from their origins as Cartesian pairs), and treat them as being all the same. Choosing any one exemplar from the coset will do; it is a form of "forgetting" of differences, or an "erasure" of origins. It is a denial of identity politics, it is a certain racial homogenization, it is a democratic notion that "all are created equal".

In a conventional exposition, there would be a verification to make sure that the result after quotienting is consistent: specifically, that the cosets are disjoint, that no exemplar ever belongs to two different cosets, and that the resulting mapping is one-to-one and onto – thus, an isomorphism. This can be taken for granted here; it needs verification only after the more abstract definition is given.

Tensor products as concatenation

The above was needlessly complicated. There is a wildly simpler way of saying the same thing, which, remarkably, arrives at the same place. Let $F(x,y,z)$ be a function of variables x,y,z which are understood to be ordinal numbers, that is, integers. Without any further restrictions (for the moment, as we ignore change-of-basis, for now), the function $F(x,y,z)$ is a tensor, and the x,y,z are the tensor indexes. Given another function aka tensor $G(s,t)$, the tensor product is simple the tensor

$$T(x,y,z,s,t) = F(x,y,z)\,G(s,t)$$

where the product is simply the scalar product (for whatever field or ring that F and G are valued in). This seems almost trivial in it's definition – how hard can a scalar product be? Its just the ordinary multiplication of numbers. This seems effectively trivial, but it hides a bit of trickery: there's a sleight-of-hand. The tensor on the left-hand side is written as $T(x,y,z,s,t)$ and not as $T((x,y,z),(s,t))$. If we look at (x,y,z) and (s,t) as two ordered lists, then (x,y,z,s,t) is the concatenation of those lists. It is NOT the Cartesian product of them!

As always, let's belabor the painfully obvious:

$$(x,y,z,s,t) \neq (x,y,z) \times (s,t) = ((x,y,z),(s,t))$$

The left side is a list. The right side is a list of lists. List concatenation "erases" the nested parenthesis that appear on the right-hand-side. It "forgets" where the indexes came from. Indeed, it might have been the case that the list was the result of concatenating (x,y) with (z,s,t) – we simply don't know, we forgot this bit of information. We "erased" it, the origins have been "democratized": we only have a list (x,y,z,s,t) but know not whence it came.[2]

To illustrate the correspondence, list concatenation can also be written with an equivalence principle. Using the symbol $\sim$ to denote "the same as", one writes

$$(x,y,z) \times (s,t) \sim (x,y) \times (z,s,t)$$

Using the equivalence $\sim$ to write a quotient space of concatenated products requires the development of some additional concepts. This is partly undertaken in the next section.

[2]The formal term for this property is that the product is "associative": re-arranging the parenthesis will not change the result.

This works for conventional tensors because F,G and T are considered to be maps to "numbers", elements of a field or maybe a ring. The multiplication of numbers (members of a field or ring) is also "forgetful": when we write "42", we don't know if it was constructed from the product of 6 and 7, or from the product of 2 and 21. It could have been either of these, or yet more. To put it differently, if $F(a,b,c)=6$ and $G(d,e)=7$ for fixed constants a,b,c,d,e, and we construct the product $T(a,b,c,d,e)=F(a,b,c)G(d,e)$, we no longer know where the 42 came from. It might have come from $T(a,b,c,d,e)=H(a,b)K(c,d,e)$ where $H(a,b)=2$ and $K(c,d,e)=21$. The forgetfulness of list concatenation goes hand-in-hand with the forgetfulness of multiplication in fields and rings. Tensors work because the exploit both of these properties.[3]

Cartesian products and lambda calculus

The simply-typed lambda calculus is famously the internal language of Cartesian-closed categories. Let's take a moment to unpack that statement. The treatment here is informal; our goal is not to teach category theory. Consider a class S of symbols. We call it a class because it could be a finite set, or an infinite set; it may be uncountable, or it may be so horridly structured that it cannot be expressed as a set. For the purposes of comp sci, things are always finite, so calling it a class S is merely conventional. The elements of S are "symbols" because, for the purposes here, symbols, and the connotation that they "stand for something", is an important property.

The Cartesian product of elements of S is an ordered list: we already wrote above that $a\times b\times c=(a,b,c)$ is two different ways of writing the same thing. It is convenient to drop the commas, and write $(a,b,c)=(a\,b\,c)$ as is standard in Lisp-dialect programming languages. Of course, one can likewise construct lists-of-lists, and so on. Consider now the collection (class) of all such nested lists-of-lists-of-lists... A typical exemplar might look like

$$(a\,b\ (c\,d)\ e\ (f\ g\ (h(j(k(m))))))$$

whence the acronym LISP - "Lots of InsidiouS Parenthesis" comes from. Here, the letters were taken to be symbols drawn from the class S. It is, however, convenient to introduce the notion of variables; for these, we write $x,y,z,\cdots$ as always, by convention. One can then consider the class of lists with embedded variables in them. OK, but things really get interesting when one then considers replacing variables by values. To do this, the special notation of λ is introduced, and one conventionally writes

$$(\lambda x.A)B\to A[x:=B]$$

where both A and B are lists (possibly containing variables), and the expression on the left-hand side of the arrow is a lambda-binding of the variable x such that any occurrence of the variable x in the list A is to be replaced by the list B. By convention, that is also what the expression on the right hand side of the arrow is supposed to

[3]This is effectively what the abstract definition of a tensor category (or monoidal category) in category theory is saying. This text mostly tries to avoid category theory, as there is a heavy conceptual overhead required to learn category theory, and that complexity conceals, hides the points we are trying to make explicit, here. That said, some category theory is unavoidable.

mean. The arrow itself is meant to denote beta-reduction, the actual act of plugging in or substitution of the variable x by the thing B that is to be plugged in.

Beta reduction takes actual effort and work, it is a computational problem. Done by hand, it requires transcribing symbols on sheets of paper. Done by computer, it requires copying and moving bytes. Vastly complex schemes have been developed to perform beta reduction rapidly. From the comp-sci perspective, it is a highly non-trivial process, no matter how obvious it may seem from the short expression above.

That's it. That's the effective definition of simply-typed lambda calculus. It is "simply typed" because all members of the class S are taken to be of the same type. There were only four ingredients in the construction:

- The Cartesian product, which allows the construction of lists (and of lists of lists ...).
- The use of variables as "placeholders".
- The use of a special symbol λ which is used to call out or bind or name a specific variable.
- The performance of substitution, named "beta-reduction", for historical reasons.

There is, of course, much more that can be said about lambda calculus; this hardly scratches the surface. But it does show the centrality of the Cartesian product to the construction.

It also helps highlight just how different list concatenation is from the Cartesian product of lists. This is a source of tremendous confusion for students of engineering and science, and so (as always) it is worth belaboring here. By convention, one is introduced to the notion of $\mathbb{R}^n$ as the n-dimensional Cartesian space. It is a coordinate space – the points of Cartesian space are labeled by n coordinates, taken to be real numbers, *i.e.* taken as elements of $\mathbb{R}$. By convention the Cartesian product of such spaces is $\mathbb{R}^m \times \mathbb{R}^n \cong \mathbb{R}^{m+n}$ where the $\cong$ symbol denotes isomorphism. By convention, this works as list concatenation: if one has spatial coordinates (x,y,z) in the 3D space $\mathbb{R}^3$ and the spatial coordinates $(s,t) \in \mathbb{R}^2$, then, "of course", $(x,y,z,s,t) \in \mathbb{R}^5 = \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$. Notationally, this is a bit unfortunate for our purposes, as it seems to be a form of list concatenation. Even worse, the right hand side is not a tensor product! Are you confused yet? Posed as a riddle, the solution to the riddle is to realize that when one writes a tensor T_{ijklm} of five indices, these indices are ordinal numbers: integers. They are not real-number-valued coordinates of Euclidean space. The value of T might be real, but the indexes are not. The abundance of notations - parenthesis of various shapes, and commas and what-not, can be a bit of a trap.

Tensor product, in general

We now have enough machinery developed to allow a general definition of the tensor product. Lets return to the form

$$T(x,y,z,s,t) = F(x,y,z)\,G(s,t)$$

as a simple product of $F(x,y,z)$ and $G(s,t)$. This time, we allow F and G to be "anything with a forgetful product" – not numbers, but anything which can be "multiplied" together, in some forgetful way, so that the origin of the factors is no longer identifiable. We had been previously vague as to whether the x,y,z,s,t were variables or constants; we may proceed with this vagueness, except that now we generalize them not to be ordinals, but to be members of some class S. The ordinal property of the indexes is not relevant, here.

Replacing numbers by things that can be forgetfully multiplied discards the addition of tensors. This, however, is an important property that we want to maintain. This can be done by going "meta-mathematical" in the conventional sense, and using the disjunctive-or to combine generalized tensors. The disjunctive-or is a "menu choice": it says "pick this or pick that, pick at least one, but don't pick both". If we have generalized tensors T and U, we can no longer write $T+U$ because there is no addition, but we can still combine them with disjunctive choice: I can present you with $T\vee U$ and demand that you pick either T or U. By convention, the disjunctive-or is denoted with $\vee$.

One can still use the $\otimes$ symbol to write the tensor product, or one can omit it entirely, as was done above, when we wrote $T=FG$. It is often useful keep a tensor product symbol, but, as we've generalized, it's conventional to use a slightly different symbol: the ampersand &, and so write $T=F\&G$. We now have two symbols: & and $\vee$ and can ask what the algebra of these symbols is. Note very carefully that it is NOT the Boolean algebra. We have one distributive property, but not the other. So,

$$(u\vee v)\&w=(u\&w)\vee(v\&w)$$

is the alternate (generalized) form of the conventional distributive property

$$(u+v)\otimes w=u\otimes w+v\otimes w$$

However, the other one, that would have made things Boolean, does not hold:

$$(u\&v)\vee w\neq(u\&w)\vee(v\&w)$$

This is hardly a surprise, since it is also the case that

$$(u\otimes v)+w\neq u\otimes w+v\otimes w$$

which is once again obvious and trivial: tensor products do not form a Boolean algebra.

Tensor logic

To round out the algebraic constructions for the tensor product, we can do the same thing that was done to obtain (simply-typed) lambda calculus:

- Define the algebra of & and $\vee$ over a class of symbols S.
- Introduce variables as "placeholders".
- Employ the symbol λ to bind or call out a specific variable.

- Enable beta-reduction.

This glosses over the need for alpha-conversion and possibly eta-reduction. I will call this algebraic system "tensor logic", as I am not aware of any conventional name for it. It is similar to "linear logic", but is not the same, as it is missing an important ingredient, that of conjugation.

Inner product

Tensors become interesting in engineering and science only after the addition of one more ingredient: the inner product. The inner product allows tensors to be combined by "contracting tensor indices". For example, given tensors A_{ijk} and B_{lm}, one might consider the contraction

$$C_{ijm} = A_{ijk}B_{km} = \sum_k A_{ijk}B_{km}$$

where the middle expression uses the so-called "Einstein convention" of summing repeated indexes, whereas on the right the sum is explicit. For the moment, the distinction between covariant and contra-variant indices will be ignored; as any discussion of change-of-basis has been ignored, it remains appropriate to also ignore issues of contra/covariance.

The prototypical inner product is that of two vectors, say $\vec{a} = [a_1, a_2, \cdots, a_n]$ and $\vec{b} = [b_1, b_2, \cdots, b_n]$. One writes:

$$i\left(\vec{a}, \vec{b}\right) = \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

For the general case, one needs only to convert notation:

$$i\left(\vec{a}, \vec{b}\right) = a_1 \& b_1 \vee a_2 \& b_2 \vee \cdots \vee a_n \& b_n$$

which clearly has the same algebraic form despite a vastly different interpretation of the symbols.

Properties of the inner product

Inner products are compatible with tensors in that they are linear. For ordinary vectors, this is expressed as

$$i\left(\vec{a} + \vec{b}, \vec{c}\right) = i\left(\vec{a}, \vec{c}\right) + i\left(\vec{b}, \vec{c}\right)$$

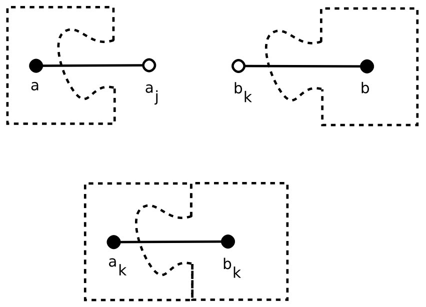
It is apparent that this property of linearity is consistent with the algebraic properties of & and $\vee$, in that the distributive property of & over $\vee$ is sufficient to guarantee linearity:

$$((a_1 \vee b_1) \& c_1) \vee ((a_2 \vee b_2) \& c_2) \vee \cdots = (a_1 \& c_1) \vee (b_1 \& c_1) \vee (a_2 \& c_2) \vee (b_2 \& c_2) \vee \cdots$$

Inner products over a field or a ring are taken to obey several additional properties: they are typically conjugate or symmetric; they are typically linear in multiplication by a scalar, and they are positive definite. These properties are discarded here. There is no conjugation or symmetry, as we have an express interest in the non-symmetric, non-conjugate case, as needed for linguistics. There is no natural scalar multiplication in our generalized definition of tensors above; it makes no sense to suddenly require one. Our generalized notion of tensors also lacks a partial order $\geq$, and so there is no way to express positive definiteness. There are cases where partial orders or pre-orders may be available; for example, string-rewriting systems (which include lambda calculus) do have a notion of "lexicographic order", and so allow ideas like "monoidal preorders" to be defined. These assumptions are not needed or relevant here. [4] Note that discarding all of these notions means that many derived properties, such as norms, homogeneity, triangle inequalities, polarization identities and so on all fall by the wayside.

Inner product as connection

Diagrammatically, the contraction of tensor indices was depicted earlier in this series with a puzzle-shaped diagram. This is repeated below, with slightly different labeling.



The diagram is slightly awkward here; the upper row is meant to depict the two vectors, with naked, uncontracted indexes. The lower part depicts a scalar, with Einstein-convention repeated indexes that are summed over. The two different mating puzzle-tab shapes hint at a hidden contra/covariance. They are also meant to indicate compatible types. That is to say, the type of a_j is compatible with the type of b_k, and so can be multiplied: so, they might both be numbers, or both be something else, as long as they can compatibly be multiplied with each-other using the product & between them. The next chapter will return to types; the tensors defined here remain simply-typed. The point of the diagram is to re-assert the claim that the inner product is just a form of "connecting things together", this time made notationally explicit.

Beta reduction as inner product

Earlier chapters claimed that beta-reduction was a form of connecting things together. How can this be? To recap the argument so far: when one plugs in 42 for x in $f(x)$ to get $f(42)$, one is "connecting" 42 to $f(x)$. This is intuitively obvious, and one can even intuit how the jigsaw-puzzle diagram captures this idea. Yet it is not (yet) obviously an inner product.

[4]That is, we are appealing to the existence of an inner product as a "universal property" of the tensor product. The universal property is the mechanism by which all of the excess baggage of symmetry and partial orders can be discarded.

To demonstrate that it is, one may appeal to the most basic type theory to elucidate. Here, 42 is an exemplar of the set of "integers", and $f(x)$ is a "function that takes integers". They can mate together because the types are compatible. We can write an integer as

$$1 \vee 2 \vee 3 \vee \cdots$$

which is a menu choice - the disjoint union, again. It says that "you should choose either 1 or choose 2 or choose 3 ... but choose at least one, and chose no more than one". Likewise, the result of plugging in is

$$f(1) \vee f(2) \vee f(3) \vee \cdots$$

This begins to look like an inner product; the resemblance can be completed by writing it as

$$f_1 \& 1 \vee f_2 \& 2 \vee f_3 \& 3 \vee \cdots$$

where, of course, $f_1 \& 1 = f(1)$ is how one writes "the value of f at one can be obtained by mating together the value one, as a jigsaw-puzzle-tab, with f_1 as the corresponding matching jigsaw-puzzle mate". With the revised notation, its now clear that beta-reduction is a form of the inner product.

To write it this way, there was a slight abuse of the notation &. Previously, it was defined as the "product of two things of the same type"; yet here it is used to denote "the mating of matching types". It is convenient to read it both ways; for the moment we sweep this under the rug, along with the co/contravariance issue. They are not entirely unrelated. Later chapters will return to these topics.

Appendix/FAQ

A few related points and commonly-raised questions are addressed here.

Monoidal categories

The definitions above correspond with the conventional definition of tensor products in monoidal categories; the goal has been not to invent something new, but rather to change perspective. Thus, the forgetfulness of the placement of parenthesis, *viz.* that $(a(bc)) \sim ((ab)c)$ is more commonly called the "associative property of a binary operation". The definition of the monoidal category elaborates on associativity: the tensor product only needs to be associative "up to isomorphism". The isomorphism that does this is called the "associator" α and it is a one-to-one, onto map $\alpha : A \otimes (B \otimes C) \mapsto (A \otimes B) \otimes C$.

The category must contain also have a "unit object" or "identity object" I; multiplication by the unit, on either the left or the right, must preserve the multiplied object, "up to isomorphism". These two isomorphisms are called the "left and right unitors"; for example, the right unitor is the one-to-one, onto map $\rho : A \otimes I \mapsto A$.

With explicit left and right unitors, one must then verify several "coherence conditions": there is a triangle diagram, which verifies that the re-arrangement of parenthesis involving the identity object gives "what is expected". Adding the associator into the mix requires the verification of a pentagon-shaped diagram. The Wikipedia article on monoidal categories provides an adequate summary of the definition of a monoidal category. Of course, much more can be said: there is a 350-page book on the topic[1].

Strict monoidal categories are those where the associator and the unitors are identities; that is all that is called upon, in the above. There is also the concept of a "lax monoidal category", where the associativity properties are slightly weakened. Some of the concepts discussed in this collection of chapters are properly lax monoidal categories; no particular effort is made to call attention to this, and it seems harmless, as every such lax category has a corresponding equivalent strict category.

There are several important aspects of the definition of monoidal categories that are relevant to the current text:

- The definition of a monoidal category makes no appeal to the concept of "addition", or anything "additive". This is not required to create a monoidal category.
- Many tensor categories in mathematics are "symmetric", in that they have a natural isomorphism $A \otimes B \cong B \otimes A$. This is manifestly *not* the case here; we are very much concerned with the case where such symmetry is entirely lacking. This is precisely why the concept of tensors can be applied to linguistics: sentences are not palindromes.
- No mention is made of inner products. The concept of a tensor category is unrelated to the concept of an inner product.

Change of basis and isometries

Discarding the concept of a norm in the definition of the inner product has serious consequences. In particular, all conceptions of change-of-basis and isometries are thrown out the window. It's worth exploring this a bit.

Consider the conventional case of Euclidean space. A change of basis involves multiplication by an orthonormal matrix M having the property $M^T M = I$ so that $M \in SO(n)$. Then for some basis vector $\vec{e}_k$, the transformed basis vector is $\vec{e}'_k = M\vec{e}_k$. Writing out indexes, one has $\left[\vec{e}_k\right]_j = \delta_{kj}$ and of course $[M]_{ij} = M_{ij}$ so that $\left[\vec{e}'_k\right]_i = [M]_{ij}\left[\vec{e}_k\right]_j = M_{ij}\delta_{jk} = M_{ik}$. What happens if instead, we only have $\&$ and $\vee$ as operators? Attempting to reproduce the above gives

$$\left[\vec{e}'_k\right]_i = M_{i1} \& \left[\vec{e}_k\right]_1 \vee M_{i2} \& \left[\vec{e}_k\right]_2 \vee M_{i3} \& \left[\vec{e}_k\right]_3 \vee \cdots$$

The construction on the right reads, as always "choose $M_{i1} \& \left[\vec{e}_k\right]_1$ or choose $M_{i2} \& \left[\vec{e}_k\right]_2$ ... choose one and only one" and so the item on the left, the symbol $\left[\vec{e}'_k\right]_i$ is not some reduced thing in the same class as $\left[\vec{e}_k\right]_j$ but is an entirely different beast: its a list of choices, or rather, just a name for a list of choices. Of course, this can be made a bit more consistent, if one claims that $\left[\vec{e}_k\right]_j$ is nothing but a singleton, with the commandment "here is exactly one thing, and one you choose one and only one thing". Thus, here, a single change-of-basis merely creates an accretion or conglomeration: some initial collection of $\vec{e}_k$ gets an extra M stuck onto the side. Connecting connectors cause things to stick to each-other; the result is still typically some thing with connectors that are still exposed.

The inner product is also conventionally used to define a norm, and then a metric, allowing concepts of a metric space to be layered on. Here, we've discarded these notions. There is one, though, that naturally remains: the ultrametric. We can take, for example, the ultrametric to be a count, a cardinal number, of the number of connectors that have been connected, or the distance between two points, by counting the number of times the & symbol appears between them. This is fine, and an interesting thing to do, but this count is naturally a number (an integer), and we have not defined any kind of natural scalar product between integers and the elements of the algebra. One may exist; only that we've not defined one.

For example, when writing $M^T M = I$, the thing on the right hand side is the identity, and so this equation states that when two M's are connected in some appropriate way, the identity results. If the algebra was, for example, the algebra of first-order logic, then the associated ultrametric might be the number of steps required to prove something, or the number of rewriting steps needed to convert one expression into another. Performing a sequence of operations *e.g.* applying a sequence of inference rules in natural deduction, and then reversing them, can be interpreted as some transformation M and it's inverse. There does not seem top be any particular insight here. One might ask for the shortest proof: the proof with the smallest number of steps from here to there, and take this as the (ultra-)metric on the space of proofs. This is interesting, but lies outside the immediate need for building up machinery of knowledge representation.

References

[1] Pavel Etingof, Shlomo Gelaki, Dmitri Nikshych, and Victor Ostrik. *Tensor Categories*. American Mathematical Society, 2010.