Named Tensor Notation

David Chiang and Sasha Rush

December 9, 2020

Contents

1 Introduction

Most papers about neural networks use the notation of vectors and matrices from applied linear algebra. This notation is very well-suited to talking about vector spaces, but less well-suited to talking about neural networks. Consider the following equation (Vaswani et al., 2017):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V. \tag{1}$$

where Q, K, and V are sequences of query, key, and value vectors packed into matrices. Does the product $QK^\top$ sum over the sequence, or over the query/key features? We would need to know the sizes of Q, K, and V to know that it's taken over the query/key features. Is the softmax taken over the query sequence or the key sequence? The usual notation doesn't even offer a way to answer this question. With multiple attention heads, the notation becomes more complicated and leaves more questions unanswered. With multiple sentences in a minibatch, the notation becomes more complicated still, and most papers wisely leave this detail out.

Libraries for programming with neural networks (Harris et al., 2020; Paszke et al., 2019) provide multidimensional arrays, called tensors (although usually without the theory associated with tensors in linear algebra and physics), and a rich array of operations on tensors. But they inherit from math the convention of identifying indices by *position*, making code bug-prone. Quite a few libraries have been developed to identify indices by *name* instead: Nexus (Chen, 2017), tsalib (Sinha, 2018), NamedTensor (Rush, 2019), named tensors in PyTorch (Torch Contributors, 2019), and Dex (Maclaurin et al., 2019). (Some of these libraries also add types to indices, but here we are only interested in adding names.)

Back in the realm of mathematical notation, then, we want two things: first, the flexibility of working with multidimensional arrays, and second, the perspicuity of identifying indices by name instead of by position. This document describes our proposal to do both.

As a preview, the above equation becomes

$$\text{Attention}(Q, K, V) = \underset{\mathsf{seq}}{\text{softmax}} \left(\frac{Q \underset{\mathsf{key}}{\cdot} K}{\sqrt{d_k}} \right) \underset{\mathsf{seq}}{\cdot} V \tag{2}$$

making it unambiguous which axis each operation applies to. The same equation works with multiple heads and with minibatching.

More examples of the notation are given in §3.

The source code for this document can be found at `https://github.com/namedtensor/notation/`. We invite anyone to make comments on this proposal by submitting issues or pull requests on this repository.

2 Informal Overview

Let's think first about the usual notions of vectors, matrices, and tensors, without named indices.

Define $[n] = \{1, \ldots, n\}$. We can think of a size-n real vector v as a function from $[n]$ to $\mathbb{R}$. We get the ith element of v by applying v to i, but we normally write this as v_i (instead of $v(i)$).

Similarly, we can think of an $m \times n$ real matrix as a function from $[m] \times [n]$ to $\mathbb{R}$, and an $l \times m \times n$ real tensor as a function from $[l] \times [m] \times [n]$ to $\mathbb{R}$. In general, then, real tensors are functions from *tuples of natural numbers* to reals.

When we do linear algebra, it is sometimes confusing to keep track of whether we are the rows and columns, but since we have only two axes, this typically works out. From personal experience, one sometimes can confused whether a vector is supposed to be a row vector or column vector, and a matrix is supposed to be "short and fat" or "tall and skinny", but this since there is only two possibilities (you either need to transpose the vector/matrix or not) then this doesn't create as much of an issue. As an aside, in quantum mechanics, physicists use the bra-ket notation which makes the difference between column and row vectors clear ($\langle v|$ is a row vector, $|w\rangle$ is a column vector, $\langle v|w\rangle$ is their dot product which is a scalar, and $|w\rangle\langle v|$ is their outer product which is a matrix).

When we transition to more axes, this becomes more cumbersome, both in math and in code. One symptom of this is tensors of shapes such as $(n, m, 1)$, $(1, n, m)$ etc where the 1 there merely serves as a "placeholder" to make sure the axes are ordered correctly. Another is that fully specifying the axes that operators apply to in equations such as (1), not to mention adding a "dangling axis" such as a batch number makes, makes them become much more cumbersome.

The principles underlying our proposed notation for named tensors are the following: (These principles use some of our terms such as tensors, axis specifiers, axes, which are defined below.)

- Order of tensor indices should make no difference. If i, j, k are axis specifiers (ways to specify a particular axis in a tensor axis, see below) then $A_{i,j,k} = A_{j,i,i} = \cdots = A_{k,j,i}$. The *shape* of a tensor will be always an unordered set.
- Notation should highlight semantic differences. For example, in an order four tensor encodes a batch of images, then the notation should make it clear what is the horizontal (x) axis of the images, what is the vertical (y) axis, what is the channel axis, and what is the batch axis. Even if (for example) it happens to be the case that the number of batches is the same as the horizontal dimension of the image, the notation should make it hard to confuse between an axis into the former and an axis into the latter.
- On the flip side, the notation should make it easy to suppress or defer minor details, and make it easy to express general transformations in a uniform way. It should be possible to express general transformations such as convolutions and attention that are used repeatedly and instantiate them easily with different settings of hyper parameter. Notation should avoid unnecessary clutter, and only introduce terms when they are needed to disambiguate an expression.
- It should be clear how to translate an equation in this notation into code. This is both for practical reasons (we do often want to implement papers) and also because if you are unclear how an equation translated into code it's a sign you do not really understand what it means.
- The notation should make it easy to extend operations into tensors with "dangling" axes. For example, if we defined operations on an image or sentence tensor, it should be easy to consider the extension of these operations into a tensor that has an extra batch dimension corresponding to a batch of data points.

2.1 Terminology

In this section we introduce the terminology that we will use. The two main characters we study are *tensors* and *operators* on such tensors. In mathematics, we can think of an order d tensor over a field $\mathbb{F}$ as a map from a product set $I_1 \times I_2 \times \cdots \times I_d$ into $\mathbb{F}$. Often these sets $I_1, \ldots, I_d$ are intervals of integers. We will use naming to avoid relying on order for inputs of these tensors. We will now define the main concepts we use:

- An *axis* is a set of the form $\{(\text{ax}, \text{annot}, x) | x \in X\}$ where ax and annot are strings and X is a well ordered set that can be infinite or finite. (For example, X might be the integers, strings, tuples of integers or strings, or finite subsets of any of those.) The string ax is the *name* of the axis, the string annot is its annotation (can often be the empty string ""). An axis with name ax and empty annotation is denoted by ax. If there is an annotation then we use superscript, subscripts or primes to indicate it.

- The set X is the *support* of the axis, and is denoted by $\mathit{Supp}(\mathsf{name})$. We use the notation $\mathsf{ax} \underset{\text{axis}}{=} X$ to denote that ax is an axis of the form $\{(\text{ax}, "", x) | x \in X\}$. We use the notation $\mathsf{ax}^{annot} \underset{\text{axis}}{=} X$ to denote that $\mathsf{ax}^{\text{annot}}$ is an axis of the form $\{(\text{ax}, \text{annot}, x) | x \in X\}$. The support of an annotated axis ax^{annot} is always a subset of the support of the axis ax with an empty annotation. If two axes have the same name and annotation then they must have the same support. If ax^{annot} is an axis and $x \in \mathit{Supp}(X)$, then we denote by $\mathsf{ax}^{annot}[x]$ the element $(\text{ax}, \text{annot}, x) \in \mathsf{ax}^{annot}$. Our default notion for a supporting set will be the natural numbers $\mathbb{N}$ (without zero if we use one-based indexing) and so if we simply say ""ax is an ax" then we assume that its support is the natural numbers, and hence ax is the set $\{\mathsf{ax}[1], \mathsf{ax}[2], \ldots\}$.[1]

- An element of an axis ax is known as an *index*. We think of an axis name as describing a *type* that inherits from the type X. Therefore, if (for example) $\mathit{Supp}(\mathsf{ax})$ is the integers then we can use operations such as addition, subtraction, multiplication on indices. Even if two axes have the same support, we still require explicit casting to translate between one to the other, and so (for example) if $i \in \mathsf{batch}$ then it cannot be used as is to axis into a vector of shape $\{\mathsf{time}\}$. In contrast, if two axes differ only in annotation, then we can translate indices between one and the other without explicit casting. We can think of an annotated axis of the form $\mathsf{ax}_a nnot$ as a keyword argument of type ax and name $\mathsf{ax\_annot}$. (In this sense annotation is like "Hungarian notation" for parameter names.) So for example if A is a $\{\mathsf{width}, \mathsf{height}\}$ type tensor then it corresponds to a function of the form `def A(width: width , height: height): float` and if B is a $\{\mathsf{state}^{in}, \mathsf{state}^{out}\}$ type tensor then it corresponds to a function of the form `def B(state_in: state, state_out :state): float`.

[1]This document uses a one-based indexing convention, but we can of course do everything with a zero-based indexing.

- A *slice* is a subset of an axis. (This does not need to be a strict subset, and so the ax itself is also a valid slice.) If ax^{annot} is an axis and $S \subseteq Supp(\mathsf{ax})$ then $\mathsf{ax}^{annot}[S]$ is the slice corresponding to all triples $(ax, annot, x) \in \mathsf{ax}$ such that $x \in S$. If $Supp(\mathsf{ax}^{annot})$ is a set of integers, then we use the notation $\mathsf{ax}^{annot}[..n]$ for the $\mathsf{ax}[\{1, \ldots, n\}]$. We will sometimes use annotation and slicing together and so use the notation such as $\mathsf{layer}_1 \underset{\text{axis}}{=} \mathsf{layer}[..128]$ to denote that layer_1 is the annotation of layer with support $\{1, .., 128\}$.

- A (tensor) *type* $\mathcal{T}$ is an unordered set of axes. If $\mathcal{T} = \{\mathsf{ax}_1, \ ldots, \mathsf{ax}_d\}$ is a type then a *shape* of type $\mathcal{T}$ is an unordered set of the form $\{\mathsf{ax}_1[S_1], \ldots, \mathsf{ax}_d[S_d]\}$. That is, a shape of type $\mathcal{T}$ is a set of slices of the axes of $\mathcal{T}$. For example, $\{\mathsf{width}, \mathsf{height}, \mathsf{batch}\}$ is a tensor type while $\{\mathsf{width}[32], \mathsf{height}[32], \mathsf{batch}[128]\}$ is a shape. Note in both shapes and types, there are no two axes or slices that have both identical names and annotations.

- For every shape $\mathcal{S} = \{S_1, \ldots, S_d\}$ (where the S_i's are slices, we denote by $ind\mathcal{S}$ the set $\{\{i_1, \ldots, i_d\} | i_1 \in S_1, \ldots, i_d \in S_d\}$. An element of $ind\mathcal{S}$ is called a *coordinate* (since it specified a unique point in all the axes). A (named) *tensor* with shape $\mathcal{S}$ is a map from $ind\mathcal{S}$ to $\mathbb{R}$. We denote by $\mathbb{R}^{\mathcal{S}}$ to be the set of tensors with shape $\mathcal{S}$ and $\mathbb{R}^{\mathcal{T}}$ to be the union of $\mathbb{R}^{\mathcal{S}}$ for all shapes $\mathcal{S}$ of type $\mathcal{T}$.

- If A is a tensor of type $\mathcal{T}$ and $\mathsf{ax} \in \mathcal{T}$ then $\mathsf{ax}(A)$ is the corresponding slice in A's shape. For example, if A has the shape $\{\mathsf{width}[..32], \mathsf{height}[..64]\}$ then $\mathsf{height}(A) = \mathsf{height}[..64]$. The length of ax in A is $|\mathsf{ax}(A)|$. (In some context this is known as dimension, but we don't use dimension here since it is an overloaded term.)

- An *operator* F is a function that takes as input one ore more tensors of a given type and outputs a tensor of a given type. For example, we can write $F : \mathbb{R}^{\mathcal{T}_1} \times \mathbb{R}^{\mathcal{T}_2} \to \mathbb{R}^{\mathcal{T}_3}$ for an operator that takes a tensor of type $\mathcal{T}_1$ and a tensor of type $\mathcal{T}_2$ and outputs a tensor of type $\mathcal{T}_3$.

2.2 In code

While we proposed to conceptually view an axis as a type, it would actually make sense in Python to think of it as an object (potentially a singleton element of a class). We do not think we want an axis as simply a string, since different packages might use a string "width" in different ways (for example maybe use different coordinate systems. Code might look something like

```
import axes

width = axes.width
width_in = width.annotate("_in")

rgb = axes.register("rgb",["red","green","blue"])
```

There would be standard constant axes that everyone can use, but also a way to add new axes. It would also be possible to register casting transformation that specify how we transform an index of one axis into another. For example translate from a coordinate system where $(0,0)$ is top left corner to one when its bottom left, or transfer a pair $(i,j) \in (\mathsf{width}, \mathsf{height})$ into $j \in \mathsf{layer}$ for flattening.

To define tensors we could use code such as

```
A = namedtensors.zeroes(size=[width[:64], height[:64],channel[:3]])
# the size is unordered, can be a list or also a set

print(A.shape)
# prints { width[:64], height[:64], channel[:3] }
```

We can also convert a standard tensor ‘T‘ to a named one.

```
A = namedtensor.tensor(T,[width, height])
# first axis becomes width and second height.
# lengths are inherited from T and so do not need to be specified.
```

If we define new functions for named tensors, we can use something like

```
  @signature(X=[width,height,channel],W=[width,height,channel],
                                   output=[width,height])
  def CONV(X,W):
      Y = namedtensor.tensor({(x,y) : sum((a,b) in zip(width(W),
                                        height(W)) X[x+a,y+b]*W[a,b
                                        ]} for (x,y) in zip(width(X
                                        ),height(X))))
```

We can use assertions to require some conditions on the lengths of the different axes.

We will also have a way in Python to extend existing functions and modules that were written for unnamed tensors. Maybe something like

```
g = named(X=[width,height], output = [layer] ,f)
```

when `f` is such a function that takes `X` as a tensor input. In this case width will be mapped to the first axis and height to the second one. As usual, the signature of an operator does not specify all the information about the relation of its input and output relation. For example, we can have the ”halving” operator $HALVE$ that drops the bottom half of an image.

$HALVE : \mathbb{R}^{\mathsf{width},\mathsf{height}} \to \mathbb{R}^{\mathsf{width},\mathsf{height}}$

The abstract types of the input and output tensors are the same but we can add the constraint that $|width(HALVE(X))| = \lfloor |width(X)/2| \rfloor$ separately.

Should also have such way to automatically transform pytorch modules of the form ‘nn.Module‘ to ones that use named tensors as their weights, inputs, and outputs. For example, something like

```
namedtensors.nn.Linear([ [width[:32],height[:32],channel[:3]],
                                layer[:70] ])
```

will create a linear transformation with weights and inputs of shape $\{\mathsf{width}[..32], \mathsf{height}[..32], \mathsf{channel}[..3]\}$ and output of shape $\{\mathsf{layer}[..70]\}$.

2.3 Example

If $\mathsf{foo}, \mathsf{bar}$ are axes (supported over the natural numbers) then the following is a tensor of shape $\{\mathsf{foo}[..2], \mathsf{bar}[..3]\}$ and type $\{\mathsf{foo}, \mathsf{bar}\}$:

$$A = \begin{array}{cc} & \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \end{bmatrix} \end{array}.$$

A is a map from $Ind\{\mathsf{foo}[..2], \mathsf{bar}[..3]\}$ to $\mathbb{R}$. Since order does not matter, we can also think of it as a map from $\mathsf{foo}[..2] \times \mathsf{bar}[..3]$ to $\mathbb{R}$. So, given $i \in namefoo[..2]$ and $j \in \mathsf{bar}[..3]$, we can write $A(i, j)$ or $A_{i,j}$ for the corresponding element of A. In particular $A_{\mathsf{foo}[1],\mathsf{bar}[3]} = 4$. We can also write this as $A_{\mathsf{foo}=1,\mathsf{bar}=4}$. In Python-speak, while this is not how we implement tennsors, we could think of A as a function that takes two parameters, named `foo` and `bar` and outputs a real number. The type of the parameter `foo` is `foo` and the type of the parameter `bar` is `bar`.

Conventions. We use uppercase italic letters for variables standing for named tensors. We don't mind if you use another convention, but urge you not to use different styles for tensors and their elements. For example, if $\mathbf{A}$ is a tensor, then an element of $\mathbf{A}$ is written as $\mathbf{A}_{\mathsf{foo}[2],\mathsf{bar}[3]}$ – not $A_{\mathsf{foo}[2],\mathsf{bar}[3]}$ or $a_{\mathsf{foo}[2],\mathsf{bar}[3]}$. What are good choices for axis names? We recommend meaningful *words* instead of single letters, and we recommend words that describe a *whole* rather than its parts. For example, a minibatch of sentences, each of which is a sequence of one-hot vectors, would be represented by a tensor with three indices, which we might name batch, seq, and vocab. Please see §3 for more examples.

Just as the set of all size-n real vectors is written $\mathbb{R}^n$, and the set of all $m \times n$ real matrices is often written $\mathbb{R}^{m \times n}$ (which makes sense because one sometimes writes Y^X for the set of all functions from X to Y), we write $\mathbb{R}^{\mathsf{foo}[..2],\mathsf{bar}[..3]}$ for the set of all tensors with shape $\{\mathsf{foo}[..2], \mathsf{bar}[..3]\}$. We write $\mathbb{R}^{\mathsf{foo},\mathsf{bar}}$ for the set of all tensors whose shape has the form $\{\mathsf{foo}[S], \mathsf{bar}[T]\}$ for some subsets S, T of the natural numbers.

We also allow *partial indexing*:

$$A_{\mathsf{foo}[1]} = \begin{array}{c} \mathsf{bar} \\ \begin{bmatrix} 3 & 1 & 4 \end{bmatrix} \end{array}$$
$$A_{\mathsf{bar}[3]} = \begin{array}{c} \mathsf{foo} \\ \begin{bmatrix} 4 & 9 \end{bmatrix} \end{array}.$$

In general, if A is a tensor of shape $\mathcal{S}$ where $d = |\mathcal{S}|$, and $i_1, \ldots, i_\ell$ are elements in distinct slices $S_1, \ldots, S_\ell \in \mathcal{S}$ then $A(i_1, \ldots, i_\ell)$ is the tensor of shape $\mathcal{S} \setminus \{S_1, \ldots, S_\ell\}$ that maps $j_1, \ldots, j_d - \ell$ into the $A(i_1, \ldots, i_\ell, j_1, \ldots, j_{d-\ell})$.

Proposed convention: We propose that if i is an axis belonging to an axis *not* in *type*(A) then $A(i) = A$. This can be convenient for example to use a $\{\mathsf{width}, \mathsf{height}\}$ type tensor in a function that expects a $\{\mathsf{width}, \mathsf{height}, \mathsf{batch}\}$ tensor by essentially treating the former as if it has a trivial (only one coordinate) batch axis.

2.4 Named tensor operations

2.5 General operations

An *operator* takes as input one or more tensors and outputs as input a tensor. For example:

$CONV : \mathbb{R}^{\mathsf{width},\mathsf{height}} \times \mathbb{R}^{\mathsf{width},\mathsf{height}} \to \mathbb{R}^{\mathsf{width},\mathsf{height}}$

is the two dimensional convolution defined as following: $CONV(W, X) = Y$ of shape $\{\mathsf{width}(X), \mathsf{height}(X)\}$ such that for every $(i, j) \in (\mathsf{width}, \mathsf{height})$,

$Y_{x,y} = \sum_{(a,b) \in (\mathsf{width}(W), \mathsf{height}(W))} X_{x+a,y+b} W_{a,b}$

Another way to write it is that

$Y_{x,y} = X_{\mathsf{width}[x-\ell:x+\ell],\mathsf{height}[y-\ell,y+\ell]} \cdot W$ where $|\mathsf{width}(W)| = |\mathsf{height}(W)| = 2\ell + 1$.

(We are thinking of the axes of W here as indexed by integers $\{-\ell, -\ell + 1, \ldots, 0, 1, \ldots, \ell\}$. We assume here that any "out of bound" value defaults to zero.)

Dangling and aligned axes We use the following conventions for an operator F taking one or more tensors $X_1, X_2, \ldots, X_\ell$ when these tensors contain axes that do not appear in the signature of F:

- If an axis ax appears only in one of the X_i's and not in any other, then it is called *dangling*. In such a case, we execute F in parallel for every slice of the form $X_i(\mathsf{ax} = i)$ and the output tensor will contain the axis ax with the same support as that of this X_i with the corresponding slice having the output of F on that slice.

- If an axis ax that does not appear in the signature appears in more than one of the X_i's, then the two appearances must have the same support (unless we explain how aligning them can happen). In such a case these axes are *aligned*. In such cases, for every i in the support of this axis, we run F on the corresponding slices of all tensors in which the axis appears.

For the purposes of the above, if a tensor A's type contains an axis ax but $|\mathsf{ax}(A)| = 1$ then we consider it as if A does not have this axis.

For example, consider the following operator $F : \mathbb{R}^{\mathsf{layer}} \times \mathbb{R}^{\mathsf{layer}} \to \mathbb{R}$ defined as $F(X, W) = ReLU(X \cdot W)$. If we execute this operator on $X \in \mathbb{R}^{\mathsf{layer},\mathsf{batch}}$ and $W \in \mathbb{R}^{\mathsf{layer},\mathsf{output}}$ then the output will be a tensor in $\mathbb{R}^{\mathsf{batch},\mathsf{output}}$.

2.5.1 Elementwise operations

Any function from scalars to scalars can be applied elementwise to a named tensor:

$$\exp A = \begin{array}{c c}& \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} \exp 3 & \exp 1 & \exp 4 \\ \exp 1 & \exp 5 & \exp 9 \end{bmatrix}\end{array}.$$

More elementwise unary operations:

kA	scalar multiplication by k
$-A$	negation
$\exp A$	elementwise exponential function
$\tanh A$	hyperbolic tangent
$\sigma(A)$	logistic sigmoid
$\mathrm{ReLU}(A)$	rectified linear unit

Any function or operator that takes two scalar arguments can be applied elementwise to two named tensors with the same shape. If A is as above and

$$B = \begin{array}{c c}& \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 2 & 7 & 1 \\ 8 & 2 & 8 \end{bmatrix}\end{array}$$

then

$$A + B = \begin{array}{c c}& \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 3+2 & 1+7 & 4+1 \\ 1+8 & 5+2 & 9+8 \end{bmatrix}\end{array}.$$

But things get more complicated when A and B don't have the same shape. If A and B each have an axis with the same name (and size), the two indices are *aligned*, as above. But if A has an axis named i and B doesn't (or it does but it is of size one), then we do *broadcasting*, which means effectively that we replace

B with a new tensor B' that contains a copy of B for every value of axis i.

$$A + 1 = \begin{array}{c c} & \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 3+1 & 1+1 & 4+1 \\ 1+1 & 5+1 & 9+1 \end{bmatrix} \end{array}$$

$$A + B_{\mathsf{foo}[1]} = \begin{array}{c c} & \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 3+2 & 1+7 & 4+1 \\ 1+2 & 5+7 & 9+1 \end{bmatrix} \end{array}$$

$$A + B_{\mathsf{bar}[3]} = \begin{array}{c c} & \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 3+1 & 1+1 & 4+1 \\ 1+8 & 5+8 & 9+8 \end{bmatrix} \end{array}.$$

Similarly, if B has an axis named i and A doesn't, then we effectively replace A with a new tensor A' that contains a copy of A for every value of axis i. If you've programmed with NumPy or any of its derivatives, this should be unsurprising to you.

More elementwise binary operations:

$A + B$	addition
$A - B$	subtraction
$A \odot B$	elementwise (Hadamard) product
A/B	elementwise division
$\max\{A, B\}$	elementwise maximum
$\min\{A, B\}$	elementwise minimum

2.5.2 Reductions

If A is a tensor and $\mathsf{ax} \in \mathit{shape}(A)$ then we use $\sum_{\mathsf{ax}} A$ as shorthand for $\sum_{x \in \mathsf{ax}(A)} A_x$.

The same rules for alignment and broadcasting apply to functions that take tensor as arguments or return tensors. The gory details are in §5.3, but we present the most important subcases here. The first is *reductions*, which are functions from vectors to scalars. Unlike with functions on scalars, we always have to specify which axis these functions apply to, using a subscript. (This is equivalent to the `axis` argument in NumPy and `dim` in PyTorch.)

For example, using the same example tensor A from above,

$$\sum_{\mathsf{foo}} A = \begin{array}{c} \mathsf{bar} \\ \begin{bmatrix} 3+1 & 1+5 & 4+9 \end{bmatrix} \end{array}$$

$$\sum_{\mathsf{bar}} A = \begin{array}{c} \mathsf{foo} \\ \begin{bmatrix} 3+1+4 & 1+5+9 \end{bmatrix} \end{array}.$$

More reductions: If A has shape $\{\mathsf{i}[..X], \ldots\}$, then

$$\sum_{\mathsf{foo}} A = \sum_{x \in X} A_{\mathsf{foo}[x]} = \begin{array}{c}\mathsf{bar}\\ \begin{bmatrix}4 & 6 & 13\end{bmatrix}\end{array}$$

$$\operatorname*{norm}_{\mathsf{foo}} A = \sqrt{\sum_{\mathsf{foo}} A^2} = \begin{array}{c}\mathsf{bar}\\ \begin{bmatrix}\sqrt{10} & \sqrt{26} & \sqrt{97}\end{bmatrix}\end{array}$$

$$\min_{\mathsf{foo}} A = \min_{x \in X} A_{\mathsf{foo}[x]} = \begin{array}{c}\mathsf{bar}\\ \begin{bmatrix}1 & 1 & 4\end{bmatrix}\end{array}$$

$$\max_{\mathsf{foo}} A = \max_{x \in X} A_{\mathsf{foo}[x]} = \begin{array}{c}\mathsf{bar}\\ \begin{bmatrix}3 & 5 & 9\end{bmatrix}\end{array}$$

$$\operatorname*{mean}_{\mathsf{foo}} A = \frac{1}{|X|} A = \begin{array}{c}\mathsf{bar}\\ \begin{bmatrix}2 & 3 & 6.5\end{bmatrix}\end{array}$$

$$\operatorname*{var}_{\mathsf{foo}} A = \frac{1}{|X|} \sum_{\mathsf{i}} (A - \operatorname*{mean}_{\mathsf{foo}} A)^2 = \begin{array}{c}\mathsf{bar}\\ \begin{bmatrix}1 & 4 & 6.25\end{bmatrix}\end{array}$$

(Note that max and min are overloaded; with multiple arguments and no subscript, they are elementwise, and with a single argument and a subscript, they are reductions.)

You can also write multiple names to perform the reduction over multiple indices at once.

2.5.3 Contraction

The vector dot product (inner product) is a function from *two* vectors to a scalar, which generalizes to named tensors to give the ubiquitous *contraction* operator, which performs elementwise multiplication, then sums along an axis. It can be used, for example, for matrix multiplication:

$$C = \begin{array}{cc} & \mathsf{baz} \\ \mathsf{bar} & \begin{bmatrix}1 & -1\\ 2 & -2\\ 3 & -3\end{bmatrix}\end{array}$$

$$A \underset{\mathsf{bar}}{\cdot} C = \begin{array}{cc} & \mathsf{baz} \\ \mathsf{foo} & \begin{bmatrix}17 & -17\\ 53 & -53\end{bmatrix}\end{array}$$

In python we might write the product of A and B with shared axis bar as `dot(A,B,axis=bar)` or `dot(A[bar],B[bar])`.

If two tensors A and B have only a single shared axis, then we don't have to specify the axis, but it can still help for clarity. If we want to align together two different axes (using implicit) casting.

For example if A and B are matrices of type $\{\mathsf{state}^{in}, \mathsf{state}^{out}\}$ corresponding to transition matrices, we may want to multiply them by matching the state^{out} of A with the state^{in} layer of B. We will write this as

$$C = B \underset{\mathsf{state^{in}}|\mathsf{state^{out}}}{\cdot} A$$

we will also use the shorthand

$$B_{in\ out}A$$

for this. In Python we can use `dot(B,A,left_axis=state_in, right_axis=state_out)` or `dot(B[state_in],A[state_out]`.

Note that (like vector dot-product, but unlike matrix multiplication) the generalized dot product operator is commutative, but not associative! Specifically, if

$$A \in \mathbb{R}^{\mathsf{foo}[..m]}$$
$$B \in \mathbb{R}^{\mathsf{foo}[..m],\mathsf{bar}[..n]}$$
$$C \in \mathbb{R}^{\mathsf{foo}[..m],\mathsf{bar}[..n]}$$

then $(A \underset{\mathsf{foo}}{\cdot} B) \underset{\mathsf{bar}}{\cdot} C$ and $A \underset{\mathsf{foo}}{\cdot} (B \underset{\mathsf{bar}}{\cdot} C)$ don't even have the same shape.

2.5.4 Vectors to vectors

A very common example of a function from vectors to vectors is the softmax:

$$\underset{\mathsf{foo}}{\mathrm{softmax}}\, A = \frac{\exp A}{\sum\limits_{\mathsf{foo}} \exp A} \approx \begin{array}{cc} & \mathsf{bar} \\ \mathsf{foo} & \begin{bmatrix} 0.731 & 0.002 & 0.953 \\ 0.269 & 0.998 & 0.047 \end{bmatrix} \end{array}.$$

And it's also very handy to have a function that renames an axis.

$$[A]_{\mathsf{bar}\to\mathsf{baz}} = \begin{array}{cc} & \mathsf{baz} \\ \mathsf{foo} & \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \end{bmatrix} \end{array}$$

We assume that this operation will use any *casting* as needed. So for example, we can use $\mathsf{widthtl}$ and $\mathsf{heighttl}$ for width and height when the origin is at the top left corner, and $\mathsf{widthbl}$ and $\mathsf{heightbl}$ for these axes when the origin is in the bottom left corner. The renaming operation will automatically perform the appropriate casting.

Concatenation combines two vectors into one:

$$A \underset{\mathsf{foo}}{\oplus} B = \begin{array}{c} \\ \mathsf{foo} \end{array} \begin{array}{c} \mathsf{bar} \\ \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 7 & 1 \\ 8 & 2 & 8 \end{bmatrix} \end{array}$$

$$A \underset{\mathsf{bar}}{\oplus} B = \begin{array}{c} \\ \mathsf{foo} \end{array} \begin{array}{c} \mathsf{bar} \\ \begin{bmatrix} 3 & 1 & 4 & 2 & 7 & 1 \\ 1 & 5 & 9 & 8 & 2 & 8 \end{bmatrix} \end{array}$$

2.5.5 Matrices

Finally, we briefly consider functions on matrices, for which you have to give *two* axis names (and the order in general matters). Let A be a named tensor with shape $\{\mathsf{foo}[..2], \mathsf{bar}[..2], \mathsf{baz}[..2]\}$:

$$A_{\mathsf{foo}[1]} = \begin{array}{c} \\ \mathsf{bar} \end{array} \begin{array}{c} \mathsf{baz} \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{array}$$

$$A_{\mathsf{foo}[2]} = \begin{array}{c} \\ \mathsf{bar} \end{array} \begin{array}{c} \mathsf{baz} \\ \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \end{array}$$

$$\underset{\mathsf{bar},\mathsf{baz}}{\det} A = \begin{array}{c} \mathsf{foo} \\ \begin{bmatrix} \det \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \det \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \end{bmatrix} \end{array}$$

$$\underset{\mathsf{baz},\mathsf{bar}}{\det} A = \begin{array}{c} \mathsf{foo} \\ \begin{bmatrix} \det \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} & \det \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} \end{bmatrix} \end{array}$$

$$\underset{\mathsf{foo},\mathsf{bar}}{\det} A = \begin{array}{c} \mathsf{baz} \\ \begin{bmatrix} \det \begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix} & \det \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \end{bmatrix} \end{array}$$

For matrix inverses, there's no easy way to put a subscript under $\cdot^{-1}$, so we recommend writing $\underset{\mathsf{foo},\mathsf{bar}}{\mathrm{inv}}$.

3 Examples

3.1 Multi layer perceptron

If we want to express a depth d fully connected network, we can do so as follows. Assume for example the input has type $\{\mathsf{width}, \mathsf{height}, \mathsf{channel}\}$. We can use annotated types $\mathsf{layer}^1, \ldots, \mathsf{layer}^d$. The weight vectors will be a tuple $(W^0, \ldots, W^{d-1})$ where W^i has type $\{\mathsf{layer}^i, \mathsf{layer}^{i+1}\}$. The value is computed as follows:

1. Set $X^0 = X_{\{\textsf{width},\textsf{height},\textsf{channel}\}\to\textsf{layer}^0}$ (casting the three indices into a single index - flattening)
2. For $i \in \{0, 1, \ldots, d-1\}$ do:
   - $X^{i+1} = ReLU(W^i \cdot X^i)$. Since they share the axis $\textsf{layer}^i$ the dot product is done along this axis. The type of X^{i+1} is $\{\textsf{layer}^{i+1}\}$.
3. Output X^d

If X has an extra batch dimension then it will automatically be carried out to the output.

3.2 Attention

We can express attention as follows:

$$\text{Att}\colon \mathbb{R}^{\textsf{key}} \times \mathbb{R}^{\textsf{seq},\textsf{key}} \times \mathbb{R}^{\textsf{seq}} \to \mathbb{R}$$
$$\text{Att}(Q, K, V) = \text{softmax}\left(\frac{Q \underset{\textsf{key}}{\cdot} K}{\sqrt{|\textsf{key}(Q)|}}\right) \underset{\textsf{seq}}{\cdot} V$$

(We don't really need to specify the axes key and seq we take the dot products over, since they are only aligned ones, but it doesn't hurt to be explicit)

The function requires the precondition that $|\textsf{key}(Q)| = |\textsf{key}(K)|$ and $|\textsf{seq}(K)| = |\textsf{seq}(V)|$. We only follow the rules with "dangling" and "aligned" axes for the axes that are *not contained in the signature of the function.* This means that if we evaluate this function with the first input Q being a two dimensional vector of type $\{\textsf{seq}, \textsf{key}\}$ (instead of a single vector of type $\{\textsf{key}\}$) then because seq is not in the signature of the function, it will be treated as a dangling axis. Similarly if V has the shape $\{\textsf{seq}, \textsf{val}\}$ instead of only name $\{\textsf{seq}\}$ then it is a dangling axis as well. Hence in the case Q is of type $\{\textsf{seq}, \textsf{key}\}$ and V is of type $\{\textsf{seq}, \textsf{val}\}$ then the output of $\text{Att}(Q, K, V)$ will have type $\{\textsf{val}, \textsf{seq}\}$. The output Y satisfies the postconditions $|\textsf{seq}(Y)| = |\textsf{seq}(Q)|$ and $|\textsf{val}(Y)| = |\textsf{val}(V)|$.

Boaz: needs to continue here

In self-attention, Q, K, and V are all computed from the same sequence. Let emb denote the embedding dimension. And write:

$$W^Q \in \mathbb{R}^{\textsf{emb},\textsf{key}}$$
$$W^K \in \mathbb{R}^{\textsf{emb},\textsf{key}}$$
$$W^V \in \mathbb{R}^{\textsf{emb},\textsf{val}}$$
$$W^O \in \mathbb{R}^{\textsf{emb},\textsf{val}}$$

By our rules on dangling and aligned axes, if we take a dot product of $X \in \mathbb{R}^{\mathsf{seq},\mathsf{emb}}$ with W^Q then we will get $Q \in \mathbb{R}^{\mathsf{seq},\mathsf{key}}$. To emphasize that the dot product is taken along the emb axis we could write this as $W^Q \underset{\mathsf{emb}}{\cdot} X$ (though this is implied since this is the only aligned direction). So, if W^Q, W^K, W^V satisfy the appropriate pre-conditions, we can define

Then define

$$\begin{aligned}
\text{SelfAtt} &\colon \mathbb{R}^{\mathsf{seq},\mathsf{emb}} \to \mathbb{R}^{\mathsf{seq},\mathsf{emb}} \\
\text{SelfAtt}(X; W^Q, W^K, W^V, W^O) &= W^O \underset{\mathsf{val}}{\cdot} [\text{Att}(Q, K, V)]_{\mathsf{seq}' \to \mathsf{seq}}
\end{aligned}$$

where

$$\begin{aligned}
Q &= W^Q \underset{\mathsf{emb}}{\cdot} X \\
K &= W^K \underset{\mathsf{emb}}{\cdot} X \\
V &= W^V \underset{\mathsf{emb}}{\cdot} X.
\end{aligned}$$

To change this to multi-head self-attention with h attention heads, simply add a head axis to the W's and then write

$$\begin{aligned}
\text{MultiSelfAtt} &\colon \mathbb{R}^{\mathsf{seq},\mathsf{emb}} \to \mathbb{R}^{\mathsf{seq},\mathsf{emb}} \\
\text{MultiSelfAtt}(X; W^Q, W^K, W^V, W^O) &= \sum_{\mathsf{head}} \text{SelfAtt}(X; W^Q, W^K, W^V, W^O).
\end{aligned}$$

BOAZ: STOPPED HERE

3.3 RNN

As a second example, let's define a simple (Elman) RNN. Let d be a positive integer.

$$\begin{aligned}
x^{(t)} &\in \mathbb{R}^{\mathsf{emb}[..d]} && t = 1, \dots, n \\
h^{(t)} &\in \mathbb{R}^{\mathsf{state}[..d]} && t = 0, \dots, n \\
A &\in \mathbb{R}^{\mathsf{state}[..d],\mathsf{state}'[..d]} \\
B &\in \mathbb{R}^{\mathsf{emb}[..d],\mathsf{state}'[..d]} \\
c &\in \mathbb{R}^{\mathsf{state}'[..d]} \\
h^{(t+1)} &= \left[\tanh\left(A \underset{\mathsf{state}}{\cdot} h^{(t)} + B \underset{\mathsf{emb}}{\cdot} x^{(t)} + c\right)\right]_{\mathsf{state}' \to \mathsf{state}}
\end{aligned}$$

The renaming is necessary because our notation doesn't provide a one-step way to apply a linear transformation (A) to one axis and put the result in the same axis. For possible solutions, see §6.

3.4 Fully-Connected Layers

Fully-connected layers are bit more verbose, but make more explicit which parameters connect which layers.

$$\begin{aligned}
V &\in \mathbb{R}^{\mathsf{output}[..o],\mathsf{hidden}[..h]} & c &\in \mathbb{R}^{\mathsf{output}[..o]} \\
W &\in \mathbb{R}^{\mathsf{hidden}[..h],\mathsf{in}[..i]} & b &\in \mathbb{R}^{\mathsf{hidden}[..h]} \\
X &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{in}[..i]} \\
Y &= \sigma\left(W \underset{\mathsf{in}}{\odot} X + b\right) \\
Z &= \sigma\left(V \underset{\mathsf{hidden}}{\odot} Y + c\right)
\end{aligned}$$

3.5 Deep Learning Norms

These three functions are often informally described using the same equation, but they each correspond to very different functions. They differ by which axes are normalized.

Batch Norm

$$\begin{aligned}
X &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{channels}[..c],\mathsf{hidden}[..h]} \\
\gamma, \beta &\in \mathbb{R}^{\mathsf{batch}[..b]} \\
\operatorname{batchnorm}(X; \gamma, \beta) &= \frac{X - \underset{\mathsf{batch}}{\operatorname{mean}}(X)}{\sqrt{\underset{\mathsf{batch}}{\operatorname{var}}(X) + \epsilon}} \odot \gamma + \beta
\end{aligned}$$

Instance Norm

$$\begin{aligned}
X &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{channels}[..c],\mathsf{hidden}[..h]} \\
\gamma, \beta &\in \mathbb{R}^{\mathsf{hidden}[..h]} \\
\operatorname{instancenorm}(X; \gamma, \beta) &= \frac{X - \underset{\mathsf{hidden}}{\operatorname{mean}}(X)}{\sqrt{\underset{\mathsf{hidden}}{\operatorname{var}}(X) + \epsilon}} \odot \gamma + \beta
\end{aligned}$$

Layer Norm

$$\begin{aligned}
X &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{channels}[..c],\mathsf{hidden}[..h]} \\
\gamma, \beta &\in \mathbb{R}^{\mathsf{channels}[..c],\mathsf{hidden}[..h]} \\
\operatorname{layernorm}(X; \gamma, \beta) &= \frac{X - \underset{\mathsf{hidden},\mathsf{channels}}{\operatorname{mean}}(X)}{\sqrt{\underset{\mathsf{hidden},\mathsf{channels}}{\operatorname{var}}(X) + \epsilon}} \odot \gamma + \beta
\end{aligned}$$

3.6 Continuous Bag of Words

A continuous bag-of-words model classifies by summing up the embeddings of a sequence of words X and then projecting them to the space of classes.

$$\begin{aligned} X &\in \{0,1\}^{\mathsf{seq}[..n],\mathsf{vocab}[..v]} & \sum_{\mathsf{vocab}} X &= 1 \\ E &\in \mathbb{R}^{\mathsf{vocab}[..v],\mathsf{hidden}[..h]} \\ W &\in \mathbb{R}^{\mathsf{class}[..c],\mathsf{hidden}[..h]} \\ \text{cbow}(X; E, W) &= \underset{\mathsf{class}}{\text{softmax}}(W \underset{\mathsf{hidden}}{\cdot} E \underset{\mathsf{vocab}}{\cdot} X) \end{aligned}$$

Here, the two contractions can be done in either order, so we leave the parentheses off.

3.7 Bayes' Rule

Named indices are very helpful for working with discrete random variables, because each random variable can be represented by an axis with the same name. For instance, if A and B are random variables, we can treat $p(\mathsf{B} \mid \mathsf{A})$ and $p(\mathsf{A})$ as tensors:

$$\begin{aligned} p(\mathsf{B} \mid \mathsf{A}) &\in [0,1]^{\mathsf{A}[..a],\mathsf{B}[..b]} & \sum_{\mathsf{B}} p(\mathsf{B} \mid \mathsf{A}) &= 1 \\ p(\mathsf{A}) &\in [0,1]^{\mathsf{A}[..a]} & \sum_{\mathsf{A}} p(\mathsf{A}) &= 1 \end{aligned}$$

Then Bayes' rule is just:

$$p(\mathsf{A} \mid \mathsf{B}) = \frac{p(\mathsf{B} \mid \mathsf{A}) \odot p(\mathsf{A})}{p(\mathsf{B} \mid \mathsf{A}) \underset{\mathsf{A}}{\cdot} p(\mathsf{A})}.$$

3.8 Sudoku ILP

Sudoku puzzles can be represented as binary tiled tensors. Given a grid we can check that it is valid by converting it to a grid of grids. Constraints then ensure that there is one digit per row, per column and per sub-box.

$$\begin{aligned} X &\in \{0,1\}^{\mathsf{row}[..9],\mathsf{col}[..9],\mathsf{assign}[..9]} \\ \text{check}(X) &= \left(\sum_{\mathsf{assign}} Y = \sum_{\mathsf{Height},\mathsf{height}} Y = \sum_{\mathsf{Width},\mathsf{width}} Y = \sum_{\mathsf{height},\mathsf{width}} Y = 1 \right) \\ Y &\in \{0,1\}^{\mathsf{Height}[..3],\mathsf{Width}[..3],\mathsf{height}[..3],\mathsf{width}[..3],\mathsf{assign}[..9]} \end{aligned}$$

$$Y_{\mathsf{Height}[r],\mathsf{height}[r'],\mathsf{Width}[c],\mathsf{width}[c']} = X_{\mathsf{height}[r \times 3 + r' - 1],\mathsf{width}[c \times 3 + c' - 1]},$$

3.9 Max Pooling

Max pooling used in image recognition takes a similar form as the Sudoku example.

$$\begin{aligned}
X &\in \mathbb{R}^{\mathsf{height}[..h],\mathsf{width}[..w]} \\
\mathrm{maxpool2d}(X, kh, kw) &= \max_{\mathsf{kh},\mathsf{kw}} U \\
U &\in \mathbb{R}^{\mathsf{height}[..h/kh],\mathsf{width}[..w/kw],\mathsf{kh}[..kh],\mathsf{kw}[..kw]} \\
U_{\mathsf{height}[i],\mathsf{width}[j],\mathsf{kh}[di],\mathsf{kw}[dj]} &= X_{\mathsf{height}[i\times kh+di-1],\mathsf{width}[j\times kw+dj-1]}
\end{aligned}$$

3.10 1D Convolution

1D Convolution can be easily written by unrolling a tensor and then applying a standard dot product.

$$\begin{aligned}
X &\in \mathbb{R}^{\mathsf{channels}[..c],\mathsf{seq}[..n]} \\
W &\in \mathbb{R}^{\mathsf{out\_channels}[..c'],\mathsf{channels}[..c],\mathsf{kw}[..k]} \\
\mathrm{conv1d}(X, W) &= W \underset{\mathsf{channels},\mathsf{kw}}{\cdot} U \\
U &\in \mathbb{R}^{\mathsf{channels}[..c],\mathsf{seq}[..n-k+1],\mathsf{kw}[..k]} \\
U_{\mathsf{seq}[i],\mathsf{kw}[j]} &= X_{\mathsf{seq}[i+j-1]}
\end{aligned}$$

3.11 K-Means Clustering

The following equations define one step of k-means clustering. Given a set of points X and an initial set of cluster centers C,

$$\begin{aligned}
X &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{space}[..k]} \\
C &\in \mathbb{R}^{\mathsf{clusters}[..c],\mathsf{space}[..k]}
\end{aligned}$$

we compute cluster assignments

$$\begin{aligned}
Q &= \underset{\mathsf{clusters}}{\mathrm{argmin}}\ \underset{\mathsf{space}}{\mathrm{norm}}(C - X) \\
&= \lim_{\alpha\to-\infty} \underset{\mathsf{clusters}}{\mathrm{softmax}} \left(\alpha \underset{\mathsf{space}}{\mathrm{norm}}(C - X)\right)
\end{aligned}$$

then we recompute the cluster centers:

$$C \leftarrow \sum_{\mathsf{batch}} \frac{Q \odot X}{Q}.$$

3.12 Beam Search

Beam search is a commonly used approach for approximate discrete search. Here H is the score of each element in the beam, S is the state of each element in the beam, and f is an update function that returns the score of each state transition. Beam step returns the new H tensor.

$$
\begin{aligned}
H &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{beam}[..k]} \\
S &\in \{0,1\}^{\mathsf{batch}[..b],\mathsf{beam}[..k],\mathsf{state}[..s]} \qquad \sum_{\mathsf{state}} S = 1 \\
f &\colon \{0,1\}^{\mathsf{state}[..s]} \to \mathbb{R}^{\mathsf{state'}[..s]} \\
\text{beamstep}(H, S) &= \underset{\mathsf{beam},\mathsf{state'}}{\text{maxk}} \left(\underset{\mathsf{state'}}{\text{softmax}}(f(S)) \odot H \right)
\end{aligned}
$$

3.13 Multivariate Normal

In our notation, the application of a bilinear form is more verbose than the standard notation $((X-\mu)^\top \Sigma^{-1}(X-\mu))$, but also makes it look more like a function of two arguments (and would generalize to three or more arguments).

$$
\begin{aligned}
X &\in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{d}[..k]} \\
\mu &\in \mathbb{R}^{\mathsf{d}[..k]} \\
\Sigma &\in \mathbb{R}^{\mathsf{d1}[..k],\mathsf{d2}[..k]} \\
\mathcal{N}(X; \mu, \Sigma) &= \frac{\exp\left(-\frac{1}{2}\left(\underset{\mathsf{d1},\mathsf{d2}}{\text{inv}}(\Sigma) \underset{\mathsf{d1}}{\odot} [X-\mu]_{\mathsf{d}\to\mathsf{d1}}\right) \underset{\mathsf{d2}}{\odot} [X-\mu]_{\mathsf{d}\to\mathsf{d2}}\right)}{\sqrt{(2\pi)^k \underset{\mathsf{d1},\mathsf{d2}}{\det}(\Sigma)}}
\end{aligned}
$$

3.14 Attention with Causal Masking

When the Transformer is used for generation, it is necessary to have an additional mask to ensure the model does not look at future words. This can be included in the attention definition with clear names.

$$
\begin{aligned}
Q &\in \mathbb{R}^{\mathsf{key}[..d_v],\mathsf{seq'}[..n]} \\
K &\in \mathbb{R}^{\mathsf{head}[..h],\mathsf{key}[..d_k],\mathsf{seq}[..n]} \\
V &\in \mathbb{R}^{\mathsf{head}[..h],\mathsf{val}[..d_v],\mathsf{seq}[..n]} \\
\text{attention}(Q,K,V) &= \underset{\mathsf{seq}}{\text{softmax}} \left(\frac{Q \underset{\mathsf{key}}{\odot} K}{\sqrt{d_k}} + M \right) \underset{\mathsf{seq}}{\odot} V \\
M &\in \mathbb{R}^{\mathsf{seq}[..n],\mathsf{seq'}[..n]} \\
M_{\mathsf{seq}[i],\mathsf{seq'}[j]} &= \begin{cases} 0 & i \le j \\ -\infty & \text{otherwise} \end{cases}
\end{aligned}
$$

3.15 Full Examples: Transformer and LeNet

As further proof of concept, we have written the full models for Transformer (`https://namedtensor.github.io/transformer.html`) and LeNet (`https://namedtensor.github.io/convnet.html`).

4 LaTeX Macros

Many of the LaTeX macros used in this document are available in the style file `https://namedtensor.github.io/namedtensor.sty`. To use it, put

```
\usepackage{namedtensor}
```

in the preamble of your LaTeX source file (after `\documentclass{article}` but before `\begin{document}`).

The style file contains a small number of macros:

- Use `\name{foo}` to write an axis name: foo.
- Use `A \ndot{foo} B` for contraction: $A \underset{\mathsf{foo}}{\odot} B$. Similarly, use `A \ncat{foo} B` for concatenation.
- Use `\nsum{foo} A` for summation: $\sum\limits_{\mathsf{foo}} A$.
- Use `\nfun{foo}{qux} A` for a function named qux with a name under it: $\underset{\mathsf{foo}}{\text{qux}}\, A$.
- Use `\nmov{foo}{bar}{A}` for renaming: $[A]_{\mathsf{foo}\rightarrow\mathsf{bar}}$.

5 Formal Definitions

5.1 Named tuples

A *named tuple* is a set of pairs, written as $\{\mathsf{i}_1[x_1], \ldots, \mathsf{i}_r[x_r]\}$, where $\mathsf{i}_1, \ldots \mathsf{i}_r$ are pairwise distinct *names*. We write both names and variables ranging over names using sans-serif font.

If t is a named tuple, we write $\operatorname{dom} t$ for the set $\{\mathsf{i} \mid (\mathsf{i}[x]) \in t \text{ for some } x\}$. If $\mathsf{i} \in \operatorname{dom} t$, we write $t.\mathsf{i}$ for the unique x such that $(\mathsf{i}[x]) \in t$. We write the empty named tuple as $\emptyset$.

We define a partial ordering $\sqsubseteq$ on named tuples: $t_1 \sqsubseteq t_2$ iff for all i, x, $(\mathsf{i}[x]) \in t_1$ implies $(\mathsf{i}[x]) \in t_2$. Then $t_1 \sqcap t_2$ is the greatest lower bound of t_1 and t_2, and $t_1 \sqcup t_2$ is their least upper bound.

A *shape* is a set of pairs written as $\{\mathsf{i}_1[X_1], \ldots, \mathsf{i}_r[X_r]\}$ where $X_1, \ldots, X_r$ are sets. We use shapes to define sets of named tuples:

$$\operatorname{ind}\{\mathsf{i}_1[X_1], \ldots, \mathsf{i}_r[X_r]\} = \{\{\mathsf{i}[1] = x_1, \ldots, \mathsf{i}[r] = x_r\} \mid x_1 \in X_1, \ldots, x_r \in X_r\}.$$

We define $\sqsubseteq$, $\sqcup$, and $\sqcap$ on shapes just as with named tuples.

If $t \in \operatorname{ind} \mathcal{T}$ and $\mathcal{S} \sqsubseteq \mathcal{T}$, then we write $t|_{\mathcal{S}}$ for the named tuple $\{(\mathsf{i}[x]) \in t \mid \mathsf{i} \in \operatorname{dom} S\}$.

5.2 Named tensors

Let $[n] = \{1, \ldots, n\}$. We deal with shapes of the form $\{\mathsf{i}[..1] : [n_1], \ldots, \mathsf{i}[..r] : [n_r]\}$ so frequently that we define the shorthand $\{\mathsf{i}[..1] : n_1, \ldots, \mathsf{i}[..r] : n_r\}$.

Let F be a field and let $\mathcal{T}$ be a shape. Then a *named tensor over F with shape $\mathcal{T}$* is a mapping from $\operatorname{ind} \mathcal{T}$ to F. We write the set of all named tensors with shape $\mathcal{T}$ as $F^{\mathcal{T}}$. To avoid clutter, in place of $F^{\{\mathsf{i}_1[X_1], \ldots, \mathsf{i}_r[X_r]\}}$, we usually write $F^{\mathsf{i}_1[X_1], \ldots, \mathsf{i}_r[X_r]}$.

We don't make any distinction between a scalar (an element of F) and a named tensor with empty shape (an element of $F^{\emptyset}$).

If $A \in F^{\mathcal{T}}$, then we access an element of A by applying it to a named tuple $t \in \operatorname{ind} \mathcal{T}$; but we write this using the usual subscript notation: A_t rather than $A(t)$. To avoid clutter, in place of $A_{\{\mathsf{i}_1[x_1], \ldots, \mathsf{i}_r[x_r]\}}$, we usually write $A_{\mathsf{i}_1[x_1], \ldots, \mathsf{i}_r[x_r]}$. When a named tensor is an expression like $(A + B)$, we surround it with square brackets like this: $[A + B]_{\mathsf{i}_1[x_1], \ldots, \mathsf{i}_r[x_r]}$.

We also allow partial indices. Let $\mathcal{U}$ be a shape such that $\mathcal{U} = \mathcal{S} \sqcup \mathcal{T}$ and $\mathcal{S} \sqcap \mathcal{T} = \emptyset$. If A is a tensor with shape $\mathcal{S}$ and $s \in \operatorname{ind} \mathcal{S}$, then we define A_s to be the named tensor with shape $\mathcal{T}$ such that, for any $t \in \operatorname{ind} \mathcal{T}$,

$$[A_s]_t = A_{s \sqcup t}.$$

(For the edge case $\mathcal{S} = \mathcal{U}$ and $\mathcal{T} = \emptyset$, our definitions for indexing and partial indexing coincide: one gives a scalar and the other gives a tensor with empty shape, but we don't distinguish between the two.)

5.3 Extending functions to named tensors

In §2, we described several classes of functions that can be extended to named tensors. Here, we define how to do this for general functions.

Let $f\colon F^{\mathcal{S}} \to G^{\mathcal{T}}$ be a function from tensors to tensors. For any shape $\mathcal{U}$ such that $\mathcal{S} \sqcap \mathcal{U} = \emptyset$ and $\mathcal{T} \sqcap \mathcal{U} = \emptyset$, we can extend f to:

$$\begin{aligned} f &: F^{\mathcal{S} \sqcup \mathcal{U}} \to G^{\mathcal{T} \sqcup \mathcal{U}} \\ [f(A)]_u &= f(A_u) \qquad \text{for all } u \in \operatorname{ind} \mathcal{U}. \end{aligned}$$

If f is a multary function, we can extend its arguments to larger shapes, and we don't have to extend all the arguments with the same names. We consider just the case of two arguments; three or more arguments are analogous. Let $f\colon F^{\mathcal{S}} \times G^{\mathcal{T}} \to H^{\mathcal{U}}$ be a binary function from tensors to tensors. For any shape $\mathcal{S}'$, $\mathcal{T}'$ such that $\mathcal{U}' = \mathcal{S}' \sqcup \mathcal{T}'$ exists, and

$$\begin{aligned} \mathcal{S} \sqcap \mathcal{S}' &= \emptyset \\ \mathcal{T} \sqcap \mathcal{T}' &= \emptyset \\ \mathcal{U} \sqcap \mathcal{U}' &= \emptyset \end{aligned}$$

we can extend f to:

$$\begin{aligned} f &: F^{\mathcal{S} \sqcup \mathcal{S}'} \times G^{\mathcal{T} \sqcup \mathcal{T}'} \to H^{\mathcal{U} \sqcup \mathcal{U}'} \\ [f(A, B)]_u &= f\left(A_{u|_{\mathcal{S}'}}, B_{u|_{\mathcal{T}'}}\right) \qquad \text{for all } u \in \operatorname{ind} \mathcal{U}'. \end{aligned}$$

All of the tensor operations described in §2.4 can be defined in this way. For example, the contraction operator extends the following "named dot-product":

$$\begin{aligned} \underset{\mathsf{i}}{\cdot} &: F^{\mathsf{i}[..n]} \times F^{\mathsf{i}[..n]} \to F \\ A \underset{\mathsf{i}}{\cdot} B &= \sum_{i=1}^{n} A_{\mathsf{i}[i]} B_{\mathsf{i}[i]}. \end{aligned}$$

6 Duality

In applied linear algebra, we distinguish between column and row vectors; in pure linear algebra, vector spaces and dual vector spaces; in tensor algebra, contravariant and covariant indices; in quantum mechanics, bras and kets. Do we need something like this?

In §3.3 we saw that defining an RNN requires renaming of indices, because a linear transformation must map one axis to another axis; if we want to map an axis to itself, we need to use renaming.

In this section, we describe three possible solutions to this problem, and welcome comments about which (if any) would be best.

6.1 Contracting two names

We define a version of the contraction operator that can contract two indices with different names. If $\mathsf{i} \in \operatorname{dom} \mathcal{A}$ and $\mathsf{j} \in \operatorname{dom} \mathcal{B}$ and $\mathcal{A}.\mathsf{i} = \mathcal{B}.\mathsf{j} = X$, then we define

$$A \underset{\mathsf{i}|\mathsf{j}}{\cdot} B = \sum_{x \in X} A_{\mathsf{i}[x]} \, B_{\mathsf{j}[x]}$$

For example, the RNN would look like this.

$$\begin{aligned}
x^{(t)} &\in \mathbb{R}^{\mathsf{emb}[..d]} \\
h^{(t)} &\in \mathbb{R}^{\mathsf{state}[..d]} \\
A &\in \mathbb{R}^{\mathsf{state}[..d], \mathsf{state}'[..d]} \\
B &\in \mathbb{R}^{\mathsf{emb}[..d], \mathsf{state}[..d]} \\
c &\in \mathbb{R}^{\mathsf{state}[..d]} \\
h^{(t+1)} &= \tanh\left(A \underset{\mathsf{state}'|\mathsf{state}}{\cdot} h^{(t)} + B \underset{\mathsf{emb}}{\cdot} x^{(t)} + c \right)
\end{aligned}$$

6.2 Starred axis names

If i is a name, we also allow a tensor to have an axis $\mathsf{i}*$ (alternatively: superscript i). Multiplication contracts starred indices in the left operand with non-starred indices in the right operand.

$$\begin{aligned}
x^{(t)} &\in \mathbb{R}^{\mathsf{emb}[..d]} \\
h^{(t)} &\in \mathbb{R}^{\mathsf{state}[..d]} \\
A &\in \mathbb{R}^{\mathsf{state}*[..d], \mathsf{state}[..d]} \\
B &\in \mathbb{R}^{\mathsf{emb}*[..d], \mathsf{state}[..d]} \\
c &\in \mathbb{R}^{\mathsf{state}[..d]} \\
h^{(t+1)} &= \tanh\left(A \underset{\mathsf{state}}{\cdot} h^{(t)} + B \underset{\mathsf{emb}}{\cdot} x^{(t)} + c \right)
\end{aligned}$$

In general, if $\mathsf{i}* \in \operatorname{dom} \mathcal{A}$ and $\mathsf{i} \in \operatorname{dom} \mathcal{B}$ and $\mathcal{A}.\mathsf{i}* = \mathcal{B}.\mathsf{i} = X$, then we define

$$A \underset{\mathsf{i}}{\cdot} B = \sum_{x \in X} A_{\mathsf{i}*[..x]} \, B_{\mathsf{i}[..x]}$$

There are a few variants of this idea that have been floated:

1. $\cdot$ (no subscript) contracts every starred axis in its left operand with every corresponding unstarred axis in its right operand. Rejected.
2. $\underset{\mathsf{i}}{\cdot}$ contracts i with i, and we need another notation like $\underset{\mathsf{i}(*)}{\cdot}$ or $\times_{\mathsf{i}}$ for contracting $\mathsf{i}*$ with i.
3. $\underset{\mathsf{i}}{\cdot}$ always contracts $\mathsf{i}*$ with i; there's no way to contract i with i.

6.3 Named and numbered indices

We allow indices to have names that are natural numbers $1, 2, \ldots$, and we define "numbering" and "naming" operators:

A_{i}	rename axis i to 1
$A_{\mathsf{i},\mathsf{j}}$	rename axis i to 1 and j to 2
$A_{\rightarrow\mathsf{i}}$	rename axis 1 to i
$A_{\rightarrow\mathsf{i},\mathsf{j}}$	rename axis 1 to i and 2 to j

The numbering operators are only defined on tensors that have no numbered indices.

Then we adopt the convention that standard vector/matrix operations operate on the numbered indices. For example, vector dot-product always uses axis 1 of both its operands, so that we can write

$$C = A_{\mathsf{i}} \cdot B_{\mathsf{i}}$$

equivalent to $C = A \underset{\mathsf{i}}{\cdot} B$.

Previously, we had to define a new version of every operation; most of the time, it looked similar to the standard version (e.g., max vs $\max_{\mathsf{i}}$), but occasionally it looked quite different (e.g., matrix inversion). With numbered indices, we can use standard notation for everything. (This also suggests a clean way to integrate code that uses named tensors with code that uses ordinary tensors.)

We also get the renaming operation for free: $A_{\mathsf{i}\rightarrow\mathsf{j}} = [A_{\mathsf{i}}]_{\rightarrow\mathsf{j}}$ renames axis i to j.

Finally, this notation alleviates the duality problem, as can be seen in the definition of a RNN:

$$\begin{aligned}
x^{(t)} &\in \mathbb{R}^{\mathsf{emb}[..d]} \\
h^{(t)} &\in \mathbb{R}^{\mathsf{state}[..d]} \\
A &\in \mathbb{R}^{\mathsf{state}[..d],\mathsf{state}'[..d]} \\
B &\in \mathbb{R}^{\mathsf{state}[..d],\mathsf{emb}[..d]} \\
c &\in \mathbb{R}^{\mathsf{state}[..d]} \\
h^{(t+1)}_{\mathsf{state}} &= \tanh\left(A_{\mathsf{state},\mathsf{state}'}\, h^{(t)}_{\mathsf{state}} + B_{\mathsf{state},\mathsf{emb}}\, x^{(t)}_{\mathsf{emb}} + c_{\mathsf{state}}\right)
\end{aligned}$$

or equivalently,

$$h^{(t+1)} = \tanh\left(A_{\mathsf{state'}} \cdot h^{(t)}_{\mathsf{state}} + B_{\mathsf{emb}} \cdot x^{(t)}_{\mathsf{emb}} + c\right)$$

Attention:

$$\text{Att}\colon \mathbb{R}^{\mathsf{seq'}[..n'],\mathsf{key}[..d_k]} \times \mathbb{R}^{\mathsf{seq}[..n],\mathsf{key}[..d_k]} \times \mathbb{R}^{\mathsf{seq}[..n],\mathsf{val}[..d_v]} \to \mathbb{R}^{\mathsf{seq'}[..n'],\mathsf{val}[..d_v]}$$

$$\text{Att}(Q,K,V) = \text{softmax}\left[\frac{Q_{\mathsf{key}} \cdot K_{\mathsf{key}}}{\sqrt{d_k}}\right]_{\mathsf{seq}} \cdot V_{\mathsf{seq}}$$

Multivariate normal distribution:

$$X \in \mathbb{R}^{\mathsf{batch}[..b],\mathsf{d}[..k]}$$
$$\mu \in \mathbb{R}^{\mathsf{d}[..k]}$$
$$\Sigma \in \mathbb{R}^{\mathsf{d}[..k],\mathsf{d'}[..k]}$$
$$\mathcal{N}(X;\mu,\Sigma) = \frac{\exp\left(-\frac{1}{2}[X-\mu]_{\mathsf{d}}^{\top}\, \Sigma^{-1}_{\mathsf{d},\mathsf{d'}}\, [X-\mu]_{\mathsf{d}}\right)}{\sqrt{(2\pi)^k\, \det \Sigma_{\mathsf{d},\mathsf{d'}}}}$$

Because this notation can be a little more verbose (often requiring you to write axis names twice), we'd keep around the notation $A \underset{\mathsf{i}}{\cdot} B$ as a shorthand for $A_{\mathsf{i}} \cdot B_{\mathsf{i}}$. We'd also keep named reductions, or at least $\underset{\mathsf{i}}{\text{softmax}}$.

Acknowledgements

Thanks to Ekin Akyürek, Colin McDonald, Chung-chieh Shan, and Nishant Sinha for their input to this document (or the ideas in it).

References

Tongfei Chen. 2017. Typesafe abstractions for tensor operations. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 45–50.

Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature*, 585(7825):357–362.

Dougal Maclaurin, Alexey Radul, Matthew J. Johnson, and Dimitrios Vytiniotis. 2019. Dex: array programming with typed indices. In *NeurIPS Workshop on Program Transformations for ML*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Alexander Rush. 2019. Named tensors. Open-source software.

Nishant Sinha. 2018. Tensor shape (annotation) library. Open-source software.

Torch Contributors. 2019. Named tensors. PyTorch documentation.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc.