EINOPS: CLEAR AND RELIABLE TENSOR MANIPULATIONS WITH EINSTEIN-LIKE NOTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Tensor computations underlie modern scientific computing and deep learning. A number of tensor frameworks emerged varying in execution model, hardware support, memory management, model definition, etc. However, available tensor operations follow the same paradigm in all frameworks. Recent neural network architectures demonstrate demand for higher expressiveness of tensor operations. The current paradigm is not suited to write readable, reliable, or easy-to-modify code for multidimensional tensor manipulations. Moreover, some commonly used operations do not provide sufficient checks and can break a tensor structure. These mistakes are elusive as no tools or tests can detect them. Independently, API discrepancies complicate code transfer between frameworks. We propose `einops` notation: a uniform and generic way to manipulate tensor structure, that significantly improves code readability and flexibility by focusing on structure of input and output tensors. We implement `einops` notation in a Python package that efficiently supports multiple widely used frameworks and provides framework-independent minimalist API for tensor manipulations.

1 INTRODUCTION

Deep learning over the past decade achieved a significant progress in analysis and synthesis of images, audio and text (Bengio et al., 2017; Aggarwal et al., 2018; Foster, 2019). Tools relying on these methods became a commodity in production data pipelines.

Available research-and-production frameworks for deep learning, such as pytorch (Paszke et al., 2019), tensorflow (Abadi et al., 2015), mxnet (Chen et al., 2015), jax (Bradbury et al., 2018), and others, vary in numerous aspects, but their core functionality is built around efficient computations on n-dimensional arrays (tensors for brevity[1]). API exposed by frameworks for tensor operations are following the same approach, that combines high efficiency (specialized hardware can be utilized) and user convenience: computations can be expressed in high-level languages, e.g. Python, using a limited number of exposed (and frequently pre-compiled) operations.

Due to growing usage of deep learning in production and rising complexity of models, it becomes increasingly more important to provide programming interfaces that enable reliable and scalable development. We demonstrate that approach to define tensor operations taken by existing frameworks does not encourage writing code that is easy to interpret, maintain or modify; additionally, some of the core operations do not conduct sufficient checks and can lead to hard-to-catch mistakes. To address these problems, we propose Einstein-like notation for operations, called `einops`. We implemented this approach in a Python (Van Rossum & Drake, 2009) package to allow simple integration of notation into existing code across a variety of frameworks.

Outline We first briefly describe *mainstream approach for tensor operations* and point to its issues with examples. We review previously proposed ideas to resolve mentioned problems in *related works*. Our approach, *einops* – a verbose notation for tensor manipulation – is introduced next, followed by the code examples in *case studies*. We implement the notation in a Python package, and explain main design choices in *implementation details* section. We conclude with a *discussion* of `einops` role and common criticisms.

[1] Our sincere apologies to readers with mathematical and physical backgrounds for possible confusion.

2 MAINSTREAM APPROACH FOR TENSOR OPERATIONS

Nowadays tensor programming is dominating in deep learning and playing a crucial role in scientific computing. It first appeared in APL (Iverson, 1962), was popularized by MATLAB (Matlab, 1993) and was spread in Python community by `numpy` (Harris et al., 2020). Currently, tensor manipulations in all mainstream frameworks are based on the following assumptions:

- Tensor is an n-dimensional array with shape (e.g. $4 \times 6 \times 3$) and data type (e.g. float32).
- When matching between elements of tensors is expected, axes are aligned by order. Conventions regulate case of different dimensionalities, e.g. Numpy (2021).
- Multiple operations act differently on axes. Those operations either have conventions about axes order (e.g. convolutions, recurrent units: it is common for convolutions to expect input to be either BHWC or BCHW ordered) or should be provided with indices of axes that should be treated separately[2]:

  ```
  y = x.max(axis=1)       # some operations are provided with indices
  y = x.swap_axes(0, 1)   # of specially treated axes
  ```

  The mixed option is possible when operation has default for "special axes" that can be overridden during a call.

Benefits of the mainstream approach are simple API and a baseline implementation, as well as versatility: a lot of research code operates solely using this kind of operations for number crunching. The drawbacks are absence of semantics in an operation and no way to reflect expected input and output. To predict output after a chain of manipulations a researcher/engineer has to carefully keep in the memory (or in the code comments) shape and contents of each intermediate tensor and thus keep track of every operation.

In this listing a batch of images is collapsed into a single image by placing images in a row next to each other

```
# im_stack has shape [b, 3, h, w]
y1 = im_stack.transpose(2, 0, 3, 1).reshape(h, b * w, 3)
y2 = im_stack.transpose(2, 3, 0, 1).reshape(h, b * w, 3)
```

One of `y1`, `y2` contains a correct image, while the other is irrelevant — it requires some time to figure out which is what. A typical computation chain contains multiple tensor operations (only two in this example) and requires writing all intermediate steps on the paper to "debug" the code. Annoyingly, resulting tensors `y1` and `y2` have the same shapes and mistake in the code may stay under the radar for a long time since the code *never* errors out. The lack of stronger checks is a weak point of conventional operations.

In most cases we also cannot meaningfully visualize intermediate layers, so there is no way to narrow down searches for a problem source. Thus, an engineer has to check all the code after each failure (being restricted in time, one relies on the intuition which places are more error-prone).

Traditional operations make code less flexible: any change in the shape agreements between parts of the code is hard to align and all related code fragments (frequently located in different places) should be updated simultaneously. In the next code fragment we add (omitted) batch dimension to the code from the vision permutator (Hou et al., 2021), and then update code to support depth:

```
# pytorch-like code without batch dimension, as in the paper
x_h = x.reshape(H, W, N, S).permute(2, 1, 0, 3).reshape(N, W, H*S)
x_h = proj_h(x_h).reshape(N, W, H, S).permute(2, 1, 0, 3).reshape(H, W, C)
# with batch dimension
x_h = x.reshape(B, H, W, N, S).permute(0, 3, 2, 1, 4).reshape(B, N, W, H*S)
x_h = proj_h(x_h).reshape(B, N, W, H, S).permute(0, 3, 2, 1, 4).reshape(B, H, W, C)
# with batch and depth dimension
x_h = x.reshape(B, H, W, D, N, S).permute(0, 4, 2, 3, 1, 5).reshape(B, N, W, D, H*S)
x_h = proj_h(x_h).reshape(B, N, W, D, H, S).permute(0, 4, 2, 3, 1, 5).reshape(B, H, W,
    D, C)
```

[2]Pseudocode in the paper corresponds to `numpy` unless otherwise stated. There is no way to write cross-framework code. This problem we partially address with proposed `einops` notation.

Modifications are very error-prone: all indices should be recomputed and order of `reshape` operands should be verified. Uncommonly, this fragment is quite self-documenting since final `reshape` in each line hints the intended order of axes after the transposition. It is common to use `x.transpose(0, 3, 1, 2)` expecting other users to recognize a familiar pattern. Related, transposition in the code requires operating with *three contexts* in head (original order of axes, permutation, result order), and even when simplified to just permutation, it's unclear if permutation `(2 0 3 1)` is inverse to `(1 3 0 2)`.

Less critical, but still notable source of mistakes is 0-based enumeration of axes (Julia, MATLAB and R use 1-based), while we propose the framework not relying on axis numeration at all. Common phrase to say "first axis is batch", and there is no way to change this habit and avoid confusion between speech and code.

Finally, common tensor operations can easily break the tensor structure. For example, `reshape`, a common operation in the deep learning code, easily breaks the tensor structure because a whole tensor is treated as a sequence and no connection is assumed between axes in input and output, Figure 1.

(a) `x` shape = (92, 3, 96)

(b) `x.reshape(46, 3, -1)` shape = (46, 3, 192)

(c) `x.reshape(48, 3, -1)` shape = (48, 3, 184)

Figure 1: Example of breaking the tensor structure in 3d tensor. We use non-convential order HCW for visalizations: (a) original image; (b) partial mixing results in new colors; (c) mixing of all axes. Result shapes are shown underneath.

Aforementioned implementation mistakes can result in rejecting valuable research or drawing incorrect conclusion about the data. Moreover, there is no recipe to identify incorrect implementation, because the result tensor shape and data type are not affected by an error. Even the presence of an error can be obscured: poor performance can be misattributed to a number of other factors: data, hyperparameters, method, optimization, etc. All pointed issues have catastrophic importance for research costs and timeline in a setting where one experiment requires dozens to thousands of GPU-hours.

A number of recently proposed architectures (Wu et al., 2021; Tolstikhin et al., 2021; Jumper et al., 2021; Liu et al., 2021; Hou et al., 2021; Touvron et al., 2021) demonstrate that conservative approach with fixed ordering of axes (like BCHW for convolutional networks) is not sufficient. Existing frameworks carry this imprint that does not help with searches for new architectures.

3 Related works

A commonly taken approach to increase reliability is assigning names to tensor dimensions. Most known implementations are `xarray` (Hoyer & Hamman, 2017), `labeled_tensor` in tensorflow Hoyer (2016), namedtensor (2019) and named tensors in Pytorch (2019). Despite being around for many years, this idea got little adoption, and not used in the deep learning code: development stopped for labeled tensors and namedtensor, and pytorch named tensors are still experimental.

In this approach, operations match tensor axes based on labels (common choice of label is string name), and rely on axis label instead of axes order, e.g.:

```
# x1 has axes (x, y, height) and x2 has axes (time, x, y)
x1 + x2 # result has axes (x, y, height, time), maybe in different order
x1.mean('height') # reduce over axis
```

Axes matching rules vary across implementations. However, we can describe some common issues why this approach did not gain wide support in the deep learning community:

- Operations focus on modified axes, and neither describe input nor output; a user has to remember axes labels for each of intermediate tensors.
- Less control over data layout: order of axes may significantly influence speed (Weber & Goesele, 2017), but is not transparent
- Names should be strictly defined and mistakes in names or their alignment may result in the wrong computations, not an exception, e.g. for `namedtensor` package:

```
# x1 has axes (height, width)
# x2 has axes (h, w)
# x3 has axes (heigt)
x1 + x2 + x3 # has axes (height width h w heigt) in some order
```

- Adoption requires strong buy-in decision, as all code should be axis label-centric. In contrary, transition to labeled tensors *only* in isolated code pieces (e.g. functions) does not protect most vulnerable pieces of the code – interaction between functions, but introduces constant conversions between the code styles. Third-party components need wrapping to support labels.
- Symmetric and antisymmetric tensors with multiple identical axes pose an additional challenge: all axes labels should be unique to allow matching and axis specification.

The broader problem is to come up with a concept of axis "label" that is indeed helpful in development, i.e. sets restrictions that help in detecting or preventing coding errors, while introducing limited performance overhead.

For illustrative purpose let us discuss how labeled axes could be added to convolution:[3]

- Should convolution stick to existing behavior and rely on axes order not names? If so, we get no additional checks, but create potential problems since user is expected to use axes names in other operations (e.g. reductions), but has to drop the names before every convolution, and reorder axes if necessary. Next, user has to set axes for convolution output, in agreement with input axes labels.
- Should convolution rely on axes names? If so, should axes names be fixed? For instance, convolution may require input to have dimensions ("channel", "height", "width") while all the other dimensions could be considered 'batch-like' and preserved in the output? In this case, almost all the tensors will have the same axes names, thus name checks become meaningless. If one decides to apply attention or e.g. recurrent unit, each switch will require renaming of axes.
- Should output names be predefined? Thus all tensors produced by convolutions would have the same labels, still preventing meaningful checks. That seems to produce more potential confusion than help.
- Should input axes names propagate to output? If yes – which names should be propagated? Output channels at first seem to be disconnected from input ones. However, in practice opposite is equally frequent: in depth-wise convolution each input has one-to-one correspondence with output. Setting up common rules for propagation of "height" and "width" labels meets similar questions and exceptions: compare `padding='same'` and `padding='valid'`.

Similar questions arise for other "building blocks" of deep learning. All design choices taken should be in agreement with each other, guided by some high-level logic that works for multiple models and applications. This is hardly feasible without a well-defined concept of axis label.

Proposed implementations of labelled tensors also break common principle of decomposition in software engineering: every function has its *own* scope with input names and names for intermediate variables. Everything that is shared between scopes should be described in the function signature. Whenever an internal structure of passed or returned entity should be shared between scopes, a type/class/interface/protocol is introduced to describe passed argument. However, the concept of labelled tensor breaks this logic: it is assumed that called and calling function agree on axes names, but no way to communicate these expectations from either side is proposed.

[3]Neither implementation has proposed integration of named axes into e.g. convolution.

Alternative approach to increase readability and reliability of tensor-operating code is to deliberately set interface restrictions only on large neural modules such as language models, encoders or decoders, as in NeMo Kuchaiev et al. (2019). While allowing to reuse and wrap existing code to glue large components, this approach does not improve internals of the modules where problems are harder to locate and code is less documented. These improvements of high-level interfaces still have their challenges, for example language model can expect to manipulate sequences of letters and thus expects axis "letter". However, surrounding code may try to use it for prediction of words, pixels or phonemes. Thus, relabelling of axes may be required to "connect" subsystems. This solution is still more reliable compared to common usage of bare transpositions to align axes order.

In 2011, M. Wiebe introduced an operation `einsum` in `numpy` package. With some simplifications (absence of covariant and contravariant indices, contracted dimension may be not repeated) `einsum` mimics Einstein summation rule commonly used in physics. `numpy.einsum` stands out from the rest of `numpy` API and for a long time rarely was mentioned in tutorials. However, function universality and verbosity were beneficial, and it was ported to other packages: tensorflow, pytorch, mxnet, chainer, etc.

```
numpy.einsum('ij,jk->ik', A, B) # matrix multiplication
numpy.einsum('ijk->ij', C)      # sum over last axis
numpy.einsum('ij,ji->', A, B)   # trace of matrix product
```

In our work we try to align interface with `einsum` to allow smooth simultaneous usage. However, interface adjustments (e.g. support for multi-character axes names) are necessary.

There is an ongoing research to create languages for low-level definition of tensor operations with explicit indexing, e.g. Tensor Comprehensions (Vasilache et al., 2018) and Dex (Paszke et al., 2021).

4 `einops`

`einops` (Einstein-Inspired Notation for OPerationS) is our proposal to address previously mentioned problems. The core of our approach is a new notation for transformation patterns based on the following rules:

- axis present only in the input (the left hand side) is reduced (e.g. with max-reduction)
- axis present only in the output is "repeated" (tensor values are the same for all index values of new axes)
- all axis identifiers on either side of expression should be unique (`numpy.einsum` allows repeats to cover traces).

Examples of transformation patterns are

```
'b c h w -> b h w c' # transpose
'b c h w -> b c'     # reduce on h, w
'b c -> b c h w'     # repeat on h, w
'(h1 h2) (w1 w2) c -> (h1 w1) h2 w2 c' # split image to patches, stack them
```

The main novelty is the composition and decomposition of axes within patterns. (De)composition uses C-ordering convention, intuitively associated with digits in a number:

```
# x is of shape (10, 10, 10, 10), then x[6, 2, 4, 9] == y[6249]
y = rearrange(x, 'a b c d -> (a b c d)')
```

Changing the rightmost of "digits" changes composed index in "small steps", while any change in leftmost results in "large steps", even when axes are not decimal:

```
# Rearrange pattern that composes 3 axes into one: i1 i2 i3 -> (i1 i2 i3)
# Let original array have a shape of [2, 3, 2], result has a length 2x3x2=12
i1                0  0  0  0  0  0  1  1  1  1  1  1
i2                0  0  1  1  2  2  0  0  1  1  2  2
i3                0  1  0  1  0  1  0  1  0  1  0  1
final position    0  1  2  3  4  5  6  7  8  9 10 11
```

Reverse pattern `(i1 i2 i3) -> i1 i2 i3` uses the same correspondence to decompose axis into three. Since axis can be decomposed in multiple ways (e.g. 12 could be represented as $2 \times 3 \times 2$ or $1 \times 12 \times 1$ or $3 \times 1 \times 4$, etc), additional axis size specifications are required during decomposition. The following rule is helpful for C-ordered arrays (default ordering in all current backends): in case the order of axes does not change, result of rearrange is still a view of the original data. E.g. this rearrange returns a view, and no copy is required: `'(a b c) (d e f) (g h) -> a b (c d) e (f g h)'`.

Pattern composition and decomposition became particularly helpful to leverage existing operations for data of higher dimensionality. E.g. if an attention function accepts tensors `k, q, v` of shape `[batch, seq, channel]`, one can turn it into multi-head attention for 3-dimensional data by composing height, width and depth to a single dimension, and grouping head and batch dimension to ensure independent processing of attention heads: `b h w d head c -> (b head) (h w d) c`. Likewise, other neural blocks can be "upgraded" by rearranging inputs and outputs.

`einops` provides three functions shown below in examples (additional axes specifications are provided with `**kwargs`) :

```
# organize 16 images in 4x4 grid
rearrange(im, '(b1 b2) h w c -> (b1 h) (b2 w) c', b1=4, b2=4)
# max-pooling with kernel of size 2x2
reduce(im, 'b c (h h2) (w w2) -> b c h w', 'max', h2=2, w2=2)
# 2x upsampling of individual image by repeating pixels
repeat(im, 'h w c -> (h h2) (w w2) c', h2=2, w2=2, c=3)
```

While all patterns could be handled by a single function instead of three, we made an explicit choice to separate scenarios of "adding dimensions" (repeat), "removing dimensions" (reduce) and "keeping number of elements the same" (rearrange). This makes code more verbose and helps in producing more specific error messages when a user provides a wrong pattern.

Compared to `numpy.einsum`, we allow multi-character names (including underscores) to help readability. Arbitrary capitalization of axes names is also allowed. Names of axes as well as axes groups in parenthesis are space-separated.

Proposed notation addresses different problems of the mainstream approach:

- Both input and output are described in the operation definition: tensor dimensionality and expected order of axes. This makes `einops`-based code more declarative and self-documenting. A user is not required to remember or infer shapes of tensors after every operation.
- Input is checked for a number of dimensions and divisibility of corresponding dimensions. The length of dimension is checked if provided.
- Tensor structure cannot be broken by design, because the notation connects input axes (or their constituents) to output axes.
- Axis enumeration is not used, so no way to make one-off mistake.
- User does not need to compute permutation of axes, those are computed from pattern.
- `einops` alleviates the need to track memory layout with patterns[4].
- `einops` and `einsum` "document" inputs and outputs, thus implicitly annotating the code in the nearest proximity, better seen in the large code samples.

We show versatility of `einops` by expressing common `numpy` (`np`) operations:[5]:

```
np.transpose(x, [0, 3, 1, 2])              rearrange(x, 'b h w c -> b c h w')
np.reshape(x,                              rearrange(x, 'h w c -> (h w) c ')
 [x.shape[0]*x.shape[1], x.shape[2]])
np.squeeze(x, 0)                           rearrange(x, '() h w c -> h w c')
np.expand_dims(x, -1)                      rearrange(x, 'h w c -> h w c ()')
np.stack([r, g, b], axis=2)                rearrange([r, g, b], 'c h w -> h w c')
```

[4]New users frequently continue trying to imagine the tensors layout in memory. It takes time and practice to stop thinking about data layout and accept patterns as a new representation, simple and sufficient.

[5]Some of examples include anonymous axes and list inputs, user conveniences that we did not introduce.

```
np.concatenate([r, g, b], axis=0)          rearrange([r, g, b], 'c h w -> (c h) w')
np.flatten(x)                              rearrange(x, 'b t c -> (b t c) ')
np.swap_axes(x, 0, 1)                      rearrange(x, 'b t c -> t b c')
left, right = np.split(image, 2, axis=1)   rearrange(x, 'h (lr w) c -> lr h w c', lr=2)
np.max(x, [1, 2])                          reduce(x, 'b h w c -> b c', 'max')
np.mean(x)                                 reduce(x, 'b h w c ->', 'mean')
np.mean(x, axis=(1, 2), keepdims=True)     reduce(x, 'b h w c -> b () () c', 'mean')
np.reshape(x, [-1, 2]).max(axis=1)         reduce(x, '(h 2) -> h', 'max')
np.repeat(x, 2, axis=1)                    repeat(x, 'h w -> h (w 2)')
np.tile(x, 2, axis=1)                      repeat(x, 'h w -> h (2 w)')
np.tile(x[:, :, np.newaxis], 3, axis=2)    repeat(x, 'h w -> h w 3')
```

5 CASE STUDIES

We again consider fragments from the vision permutator (Hou et al., 2021). Two examples below only differ in what axis is mixed.

```
# vision permutator - mixing along h
x_h = x.reshape(B, H, W, N, S).permute(0, 3, 2, 1, 4).reshape(B, N, W, H*S)
x_h = proj_h(x_h).reshape(B, N, W, H, S).permute(0, 3, 2, 1, 4).reshape(B, H, W, C)
# vision permutator - mixing along w
x_w = x.reshape(B, H, W, N, S).permute(0, 1, 3, 2, 4).reshape(B, H, N, W*S)
x_w = proj_w(x_w).reshape(B, H, N, W, S).permute(0, 1, 3, 2, 4).reshape(B, H, W, C)
```

When parts of the code responsible for such manipulation are not located together, but separated into functions, implicit agreements appear, and engineers tend to avoid changing or repurposing such code. While previously simultaneous update was required in *four* operations, in `einops` changes are limited to swapping `h` and `w` before and after projection. And to remove e.g. batch dimension, one just removes axis `b` from patterns.

```
# einops: mixing along h
x_h = rearrange(x, 'b h w (n s) -> b n w (h s)', s=S)
x_h = rearrange(proj_h(x), 'b n w (h s) -> b h w (n s)', s=S)
# einops: mixing along w. We swapped h and w before and after projection
x_h = rearrange(x, 'b h w (n s) -> b n h (w s)', s=S)
x_h = rearrange(proj_w(x), 'b n h (w s) -> b h w (n s)', s=S)
```

The next fragment is derived from OpenAI's implementation of Glow (Kingma & Dhariwal, 2018). As other neural flows, Glow heavily relies on rearrangements.

```
def unsqueeze2d(x, factor=2):
    assert factor >= 1
    if factor == 1:
        return x
    shape = x.get_shape()
    height = int(shape[1])
    width = int(shape[2])
    n_channels = int(shape[3])
    assert n_channels >= 4 and n_channels % 4 == 0
    x = tf.reshape(
        x, (-1, height, width, int(n_channels/factor**2), factor, factor))
    x = tf.transpose(x, [0, 1, 4, 2, 5, 3])
    x = tf.reshape(x, (-1, int(height*factor),
                       int(width*factor), int(n_channels/factor**2)))
    return x

# same in einops, no function introduced
rearrange(x, 'b h w (c h2 w2) -> b (h h2) (w w2) c', h2=factor, w2=factor)
```

As for the original implementation, function name is confusing and non-descriptive. To reflect actual transformation, a name should be changed to `rearrange_by_squeeze_channels_and_unsqueeze_h_and_w`. `einops` alleviates necessity to introduce a function, as arguments describe input, output and the transformation itself. This example

also demonstrates how `einops` performs transformations, as under the hood it makes the same sequence of transformations (reshape-transpose-reshape) as in the original code. Reversing this rearrangement is non-trivial with the mainstream approach, but in `einops` it amounts to swapping input and output parts of the pattern.

In Appendix A we analyze and rewrite a larger fragment of code: a multi-head attention module derived from the popular implementation (Huang, 2018).

6 IMPLEMENTATION DETAILS

We implement proposed notation in a python package `einops`.

Support of multiple frameworks. `einops` supports a diverse set of deep learning frameworks (e.g. pytorch, tensorflow, chainer, jax, gluon) as well as frameworks for tensor computations: numpy, cupy (Okuta et al., 2017). We refer to them as backends. The major challenge in the support of multiple backends is the absence of common API: even simple operations like repeat, view, or transpose are defined inconsistently.

```
np.transpose(x, [2, 0, 1])         # numpy
x.transpose(2, 0, 1)               # numpy
tf.transpose(x, [2, 0, 1])         # tensorflow
K.permute_dimensions(x, [2, 0, 1]) # keras
mx.nd.transpose(x, (2, 0, 1))      # mxnet
x.permute(2, 0, 1)                 # torch

rearrange(x, 'h w t -> t h w')     # einops (any backend)
```

This inconsistency makes projects like `einops` more valuable for users, as they minimize framework specifics when user does not need it. Proposal (Data-Apis, 2021) may help in convergence on shared API and simplify development of cross-framework applications, including `einops`.

Backend recognition in `einops` does not use wrapped imports, which are commonly used in Python to handle optional dependencies:

```
def is_numpy_tensor(x): # einops does not use this approach
    try:
        import numpy as np
        return isinstance(x, np.ndarray)
    except ImportError as _:
        return False
```

Instead, `einops` keeps dictionary that maps a tensor type to a backend. When a type not found in a dictionary, a pass through backends is done, and before importing module, `einops` confirms that the backend was *previously* loaded in `sys.modules`, since `einops` will not receive tensors from non-imported modules.

The main reasons for this strategy: a) some backends take a lot of memory and time to load (e.g. tensorflow), which may result in an unforeseen usage of resources if a backend is installed but not used b) it isn't rare that a backend is installed incorrectly (e.g. wrong binary) and import drives to non-catchable segmentation faults c) checking backends one-by-one incurs unnecessary overhead, which we skip by dictionary lookup.

`einops` has shared logic for parsing and conversion into operations for every framework. This conversion is very efficient: every `einops` pattern is converted into at most 4 tensor operations by backend, even if dimensionality of tensor is large[6]. When a backend supports returning views and result of rearrangement can be a view of the original data, operations are designed to return view and avoid copying the data.

Support for backends. Each backend is represented by a proxy class that provides a minimal set of operations necessary to manipulate tensors. In addition, several supplementary methods are required and implemented to allow generic testing: most of tests can be run against any backend.

[6]There are unavoidable exceptions: e.g. some backends cannot reduce multiple axes at once.

In addition, `einops` implements **layers** for backends with layers support. This code base heavily relies on stateless operations and has some backend-specific code.

Caching plays an important role in `einops` high performance. There are two layers of caching: i) cache for pattern and provided axes; ii) cache for pattern, provided axes, and input shape. When the same pattern is applied to a new shape, only shape verification and computation of unknown axes sizes is done. When a pattern is applied to input of the same shape (quite typical for iterative algorithms, and very common in deep learning), even checks are skipped, and `einops` executes sequence of commands with cached parameters.

Exceptions are detailed to provide information about performed operation including pattern and provided axes sizes. This makes debugging easier compared to the common error messages ("operands could not be broadcast together with shapes ...").

7 DISCUSSION

Speaking of criticism, `einops` is sometimes described as "stringly-typed". We should point that it does not make `einops` less reliable compared to the mainstream approach:

```
numpy.transpose: [ndarray, Tuple[int]] -> np.ndarray
rearrange: [string, ndarray] -> ndarray, (ndarray can be polymorphic)
```

So input and output types are not any different except for string not being tuple. In frameworks tensor types do not describe number of dimensions, and existing type system cannot set restrictions on shape components. Static analysis is identically ignorant to mistakes in patterns and e.g. in axes order. There is no static check that prevents user from passing tuple with wrong number of components, or repeats, or just ridiculously large numbers. Interestingly, `numpy.einsum` can accept axis indices not string pattern, but it makes operation less expressive, less reliable and almost never used.

`einops` was also referred once as "a good intermediate solution", with an implicit assumption that there will be some *final solution* that would allow completely integrated analysis of shapes. Unfortunately, design of such system is *much* harder than it sounds: previous ideas failed when it comes to a) static propagation of axes labels/shape agreements b) reflecting expectations in signatures and interfaces, specially when it comes to axis length-modifying operations like padding c) handling polymorphism with respect to the number of dimensions. A good test of such system would be implementation of Sequential – a function that combines several (or at least two) modules and produces a new module (input and output of each module is a single tensor for simplicity). The system should be capable of figuring out if provided modules are combinable: number of dimensions, and compatibility of shapes. We did not touch indexing issues, but those should be also resolved in the *final solution*. These restrictions seem to be irresolvable without a new language or language features. To become practically valuable, the system should not be too sophisticated. If resulting solution is significantly harder to integrate compared to running every module with several test inputs, it is unlikely to get a wide adoption.

Independently of code usage, we observed that `einops` notation gets picked up for describing tensors with packed dimensions, showing its suitability as a mental model.

We intentionally omit discussion of user conveniences provided by `einops` package: anonymous axes, ellipsis, list inputs and neural layers.

8 CONCLUSION

We introduce `einops` — a minimalist notation for tensor manipulations that leverages patterns for describing structure of input and output tensors. `einops` mitigates a number of issues common to the conventional tensor manipulation routines, represents a number of commonly used functions with a small API, and makes code more readable and reliable. We implement notation in a Python package that provides identical API across a variety of tensor frameworks. Cross-framework support, expressiveness of code, additional checks and simple integration into existing projects make `einops` a convenient tool for researchers and engineers.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Charu C Aggarwal et al. Neural networks and deep learning. *Springer*, 10:978–3, 2018.

Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. MIT press Massachusetts, USA:, 2017.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Data-Apis. Consortium for Python Data API Standards, Array API standard. https://github.com/data-apis/array-api/, 2021. [Online; accessed 1-Oct-2021].

David Foster. *Generative deep learning: teaching machines to paint, write, compose, and play*. O'Reilly Media, 2019.

Charles R. Harris, K. Jarrod Millman, Stéfan 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 Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

Qibin Hou, Zihang Jiang, Li Yuan, Ming-Ming Cheng, Shuicheng Yan, and Jiashi Feng. Vision permutator: A permutable mlp-like architecture for visual recognition. *arXiv preprint arXiv:2106.12368*, 2021.

Stephan Hoyer. tf.contrib.labeled_tensor. https://github.com/tensorflow/tensorflow/commit/9d20f4ea4b0b5792bf88ef886d0143b7aa780522, 2016. [Online; accessed 1-Oct-2021].

Stephan Hoyer and Joe Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017. doi: 10.5334/jors.148. URL http://doi.org/10.5334/jors.148.

Yu-Hsiang Huang. Implementation of "attention is all you need" paper. https://github.com/jadore801120/attention-is-all-you-need-pytorch/blob/2077515a8ab24f4abdda9089c502fa14f32fc5d9/transformer/SubLayers.py, 2018. [Online; accessed 1-Oct-2021].

Kenneth E Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pp. 345–351, 1962.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

Diederik P. Kingma and Prafulla Dhariwal. Glow, official implementation. https://github.com/openai/glow/blob/91b2c577a5c110b2b38761fc56d81f7d87f077c1/tfops.py, 2018. [Online; accessed 1-Oct-2021].

Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, et al. Nemo: a toolkit for building ai applications using neural modules. *arXiv preprint arXiv:1909.09577*, 2019.

Hanxiao Liu, Zihang Dai, David R So, and Quoc V Le. Pay attention to mlps. *arXiv preprint arXiv:2105.08050*, 2021.

Matlab. Toolbox, symbolic math and others. *Mathworks Inc*, 1993.

namedtensor. namedtensor python package. https://github.com/harvardnlp/namedtensor, 2019. [Online; accessed 1-Oct-2021].

Numpy. Numpy broadcasting rules. https://numpy.org/doc/stable/user/basics.broadcasting.html, 2021. [Online; accessed 1-Oct-2021].

Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. URL http://learningsys.org/nips17/assets/papers/paper_16.pdf.

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. 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 (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Adam Paszke, Daniel Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. *arXiv preprint arXiv:2104.05372*, 2021.

Pytorch. Documentation for Pytorch named tensors. https://pytorch.org/docs/stable/named_tensor.html, 2019. [Online; accessed 1-Oct-2021].

Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, et al. Mlp-mixer: An all-mlp architecture for vision. *arXiv preprint arXiv:2105.01601*, 2021.

Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training. *arXiv preprint arXiv:2105.03404*, 2021.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

Nicolas Weber and Michael Goesele. Matog: Array layout auto-tuning for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–26, 2017.

Haiping Wu, Bin Xiao, Noel Codella, Mengchen Liu, Xiyang Dai, Lu Yuan, and Lei Zhang. Cvt: Introducing convolutions to vision transformers. *arXiv preprint arXiv:2103.15808*, 2021.

A CASE STUDY: MULTI-HEAD ATTENTION

Below we provide shortened implementation of multi-head attention based on Huang (2018). For brevity, we remove weights initialization and mask support.

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, temperature, attn_dropout=0.1):
        super().__init__()
        self.temperature = temperature
        self.dropout = nn.Dropout(attn_dropout)
        self.softmax = nn.Softmax(dim=2)

    def forward(self, q, k, v):
        attn = torch.bmm(q, k.transpose(1, 2)) / self.temperature
        attn = self.softmax(attn)
        attn = self.dropout(attn)
        output = torch.bmm(attn, v)

        return output, attn


class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1):
        super().__init__()
        self.n_head = n_head
        self.d_k = d_k
        self.d_v = d_v

        self.w_qs = nn.Linear(d_model, n_head * d_k)
        self.w_ks = nn.Linear(d_model, n_head * d_k)
        self.w_vs = nn.Linear(d_model, n_head * d_v)
        self.attention = ScaledDotProductAttention(temperature=np.power(d_k, 0.5))
        self.layer_norm = nn.LayerNorm(d_model)
        self.fc = nn.Linear(n_head * d_v, d_model)
        self.dropout = nn.Dropout(dropout)


    def forward(self, q, k, v):
        d_k, d_v, n_head = self.d_k, self.d_v, self.n_head

        sz_b, len_q, _ = q.size()
        sz_b, len_k, _ = k.size()
        sz_b, len_v, _ = v.size()

        residual = q

        q = self.w_qs(q).view(sz_b, len_q, n_head, d_k)
        k = self.w_ks(k).view(sz_b, len_k, n_head, d_k)
        v = self.w_vs(v).view(sz_b, len_v, n_head, d_v)

        q = q.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k) # (n*b) x lq x dk
        k = k.permute(2, 0, 1, 3).contiguous().view(-1, len_k, d_k) # (n*b) x lk x dk
        v = v.permute(2, 0, 1, 3).contiguous().view(-1, len_v, d_v) # (n*b) x lv x dv

        output, attn = self.attention(q, k, v)

        output = output.view(n_head, sz_b, len_q, d_v)
        output = output.permute(1, 2, 0, 3).contiguous().view(sz_b, len_q, -1) # b x lq
     x (n*dv)

        output = self.dropout(self.fc(output))
        output = self.layer_norm(output + residual)

        return output, attn
```

Original implementation demonstrates a number of issues that we previously discussed: unchecked and hard-to-track transformations, necessity to keep comments about the shape. Frequent usage of -1 in reshapes makes it simple to introduce errors. The critical part of computations (attention) is offloaded to a separate module, which does not check input and which is not documented: dimensions of inputs and outputs are not defined, just as the connection between them. Thus, provided `ScaledDotProductAttention` cannot be considered as an independent, self-containing or reusable module.

Inlining of `ScaledDotProductAttention` inside `MultiHeadAttention` could improve the situation, but would also make the problem with shapes more obvious, as more comments would be required for inlined variables.

We can compare that with `einops` implementation, where all computations are done in a single module. Each axis is easy to track throughout the code. An axis index is needed three times (explicitly in `softmax`, implicitly in `fc` and `layer_norm`), but those are easy to find from the code without any additional comments – which demonstrates how `einops` "implicitly annotates" code in the proximity.

```
class MultiHeadAttentionNew(nn.Module):
    def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1):
        super().__init__()
        self.n_head = n_head

        self.w_qs = nn.Linear(d_model, n_head * d_k)
        self.w_ks = nn.Linear(d_model, n_head * d_k)
        self.w_vs = nn.Linear(d_model, n_head * d_v)
        self.fc = nn.Linear(n_head * d_v, d_model)

        self.dropout = nn.Dropout(p=dropout)
        self.attn_dropout = nn.Dropout(p=0.1)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, q, k, v):
        residual = q
        q = rearrange(self.w_qs(q), 'b l (h k) -> h b l k', h=self.n_head)
        k = rearrange(self.w_ks(k), 'b t (h k) -> h b t k', h=self.n_head)
        v = rearrange(self.w_vs(v), 'b t (h v) -> h b t v', h=self.n_head)
        attn = torch.einsum('hblk,hbtk->hblt', [q, k]) / np.sqrt(q.shape[-1])
        attn = self.attn_dropout(attn.softmax(dim=-1))
        output = torch.einsum('hblt,hbtv->hblv', [attn, v])
        output = rearrange(output, 'h b l v -> b l (h v)')
        output = self.dropout(self.fc(output))
        output = self.layer_norm(output + residual)
        return output, attn
```