

Language engineering applied to matrix algebra debugging in Python

Clarifying and explaining matrix algebra visually with TensorSensor

Terence Parr
University of San Francisco

Let's start with the obvious

- In order to read code, understand exceptions, or debug we need to know the types of all variables involved, even in non-statically typed languages like Python

```
def f(x,y):  
    return x+y
```

In Python, this
could do anything
including erase
your hard drive

- Implementing deep learning neural networks means matching dimensions of layers and, more generally, matrix dimensions

Current programmer aids are weak

- It's easy to lose track of matrix/tensor dimensionality in even simple expressions (even in statically-typed languages)
- Upon error, we get less than helpful exception messages
- Here's a sample NumPy message:

```
...  
----> 10 Y = W @ X.T + b  
  
ValueError: matmul: Input operand 1 has a  
mismatch in its core dimension 0, with gufunc  
signature (n?,k),(k,m?)->(n?,m?) (size 764 is  
different from 100)
```

Which operator?

Which operand?

We could rerun using the debugger but...

- Python debuggers seem much slower than normal execution
- Even regular execution could take hours before faulting
- Sometimes it's hard to set a breakpoint on the right statement when it's in a loop
- Conditional breakpoints are challenging when the values are high-dimension matrices
- Debugger does not tell us which subexpression caused an exception, due to line-level granularity Python
(Must write down shape of all operands and lineup dimensions)

Introducing TensorSensor

Clarifying errors visually & textually



```
import tsensor
with tsensor.clarify():
    Y = W @ X.T + b
```



→ 10 Y = W @ X.T + b

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k), (k,m?)->(n?,m?) (size 764 is different from 100)

TensorSensor augments exception messages

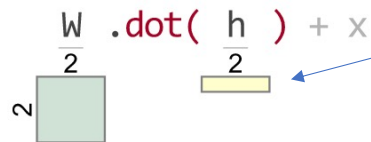
$$Y = \begin{matrix} & W & @ & X.T & + & b \\ & \overline{100} & & \overline{200} & & \\ 764 & \boxed{} & & 764 & \boxed{} & \end{matrix}$$

TensorSensor generates visual cues to matrix dimensions and offending operator / operands

```
...
ValueError: matmul: Input operand
...
Cause: @ on tensor operand W w/shape (764, 100) and operand X.T w/shape (764, 200)
```

Requirement: support multiple libraries

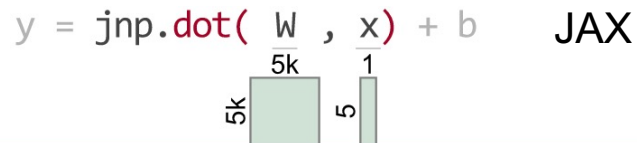
(1D vectors are yellow)



PyTorch

PyTorch says: 1D tensors expected, got 2D, 1
ors at /tmp/pip-req-build-as6281z5/aten/src/eric/THTensorEvenMoreMath.cpp:83

tsensor adds: Cause: W.dot(h) tensor arg h v
[2]



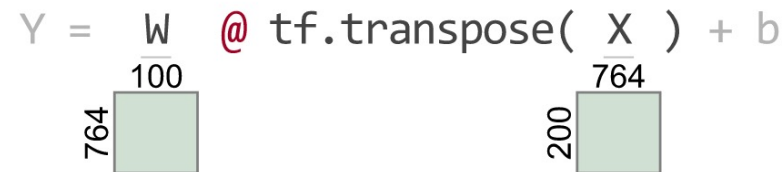
JAX

JAX says: Incompatible shapes for dot: got
5000) and (5, 1).

tsensor adds: Cause: jnp.dot(W,x) tensor arg W w/s
hape (5000, 5000), arg x w/shape (5, 1)

TensorFlow

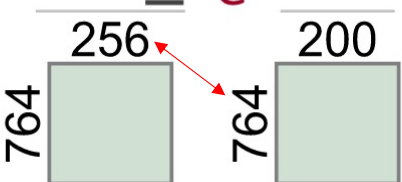
```
import tensorflow as tf
W = tf.random.uniform((d,n_neurons))
b = tf.random.uniform((n_neurons,1))
X = tf.random.uniform((n,d))
with tsensor.clarify():
    Y = W @ tf.transpose(X) + b
```



InvalidArgumentError: Matrix size-incompatible: In[0]:
[764,100], In[1]: [764,200] [Op:MatMul]
Cause: @ on tensor operand W w/shape (764, 100) and operand
tf.transpose(X) w/shape (764, 200)

Requirement: support complex expressions

- We need to know which operator and operands failed so we should highlight the offending elements:

$$h_ = \text{torch.tanh}(Whh_ @ (r*h) + \underbrace{Uxh_}_{764 \times 256} @ \underbrace{X.T}_{764 \times 200} + bh_)$$


- (Oops: The columns of Uxh_ must match the rows of X.T)

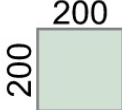
Requirement: support prebuilt layers

Should highlight error in user code not library code

```
L = torch.nn.Linear(d, n_neurons)
X = torch.rand(n,n) # oops! Should be n x d
with tsensor.clarify():
    Y = L(X)
```

Visualization

$Y = L(X)$



Augmented exception message

```
RuntimeError: size mismatch, m1: [200 x 200], m2:
[764 x 100] at /tmp/pip-req-build-
as6281z5/aten/src/TH/generic/THTensorMath.cpp:41
Cause: L(X) tensor arg X w/shape [200, 200]
```

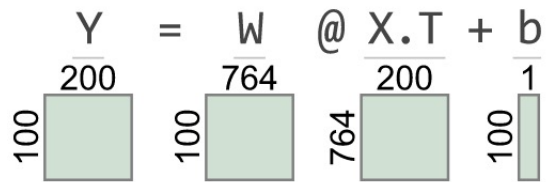
(TensorSensor augments message)

(Note unhelpful C++ reference)

Requirement: explain code w/o errors

```
with tsensor.explain():
```

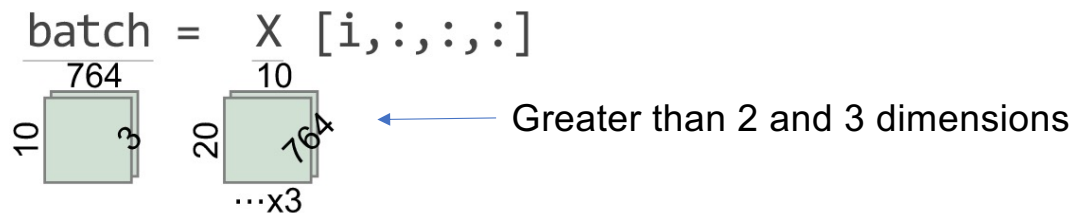
```
Y = W @ X.T + b
```



```
X = torch.rand(n_batches, batch_size, d, 3)
```

```
with tsensor.explain():
```

```
batch = X[i,:,:,:]
```



$y = \text{torch.dot}(b, b)$

Diagram illustrating the dimensions of the tensors in the equation $y = \text{torch.dot}(b, b)$:

- b : 100 rows, 1 column
- b : 100 rows, 1 column

$y = b @ b.T$

Diagram illustrating the dimensions of the tensors in the equation $y = b @ b.T$:

- y : 1 row, 1 column
- b : 100 rows, 1 column
- $b.T$: 1 row, 100 columns

Very helpful when trying to read
(even correct) code

Language engineering related implementation

Key questions to answer

1. Python traps exceptions at the statement level, so how can we identify offending operators and operand values?
2. How can we trap exceptions without requiring try/except blocks around statements with potential dimension problems?
(Also want to ignore non-matrix related exceptions; user shouldn't need to know Python internal execution details)
3. How can we generate matrix visualizations w/o explicit calls?
4. Can we make use of library zero cost until an exception?

In other words, how do we make this precise and unobtrusive?

Screams for code instrumentation, right?

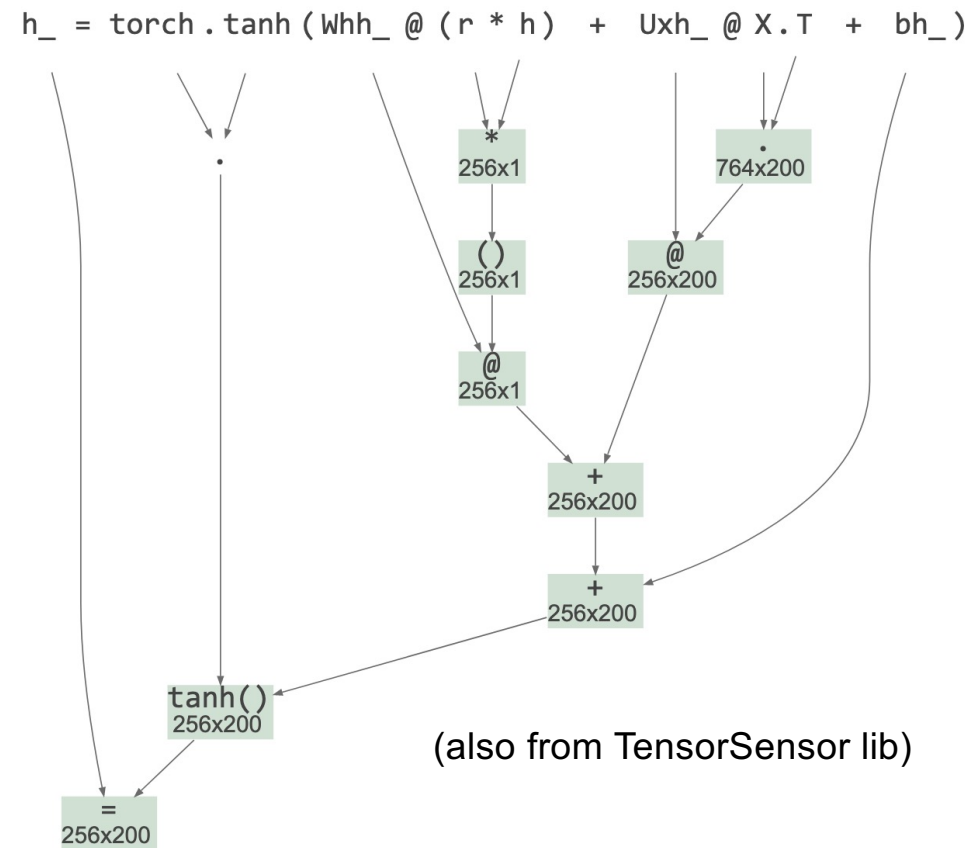
- I considered injecting Python bytecode to track subexpression evaluation and to get finer-grained exception handling
- But, that might require a separate tool and a preprocessing step, rather than simply importing a library
- Injected code could slow down entire program execution significantly
- Injected code might affect Python optimization / concurrency
- It seemed like a lot of work to learn and implement for a skill I probably would never need again

Getting operator-level exceptions w/o bytecode instrumentation requires a total hack

- We have: (1) the full execution stack and (2) the offending line of source code
- To identify the individual operator and operands that triggered an exception use brute-force:
 - reevaluate each operation in the line, piece-by-piece, in proper order, and in the correct execution context (must pick correct stack frame)
- Wait for an operator to cause an exception, report op/opnds
- Zero cost until error, but assumes side-effect free functions
- Even if side-effecting, who cares (usually)? Program is about to terminate

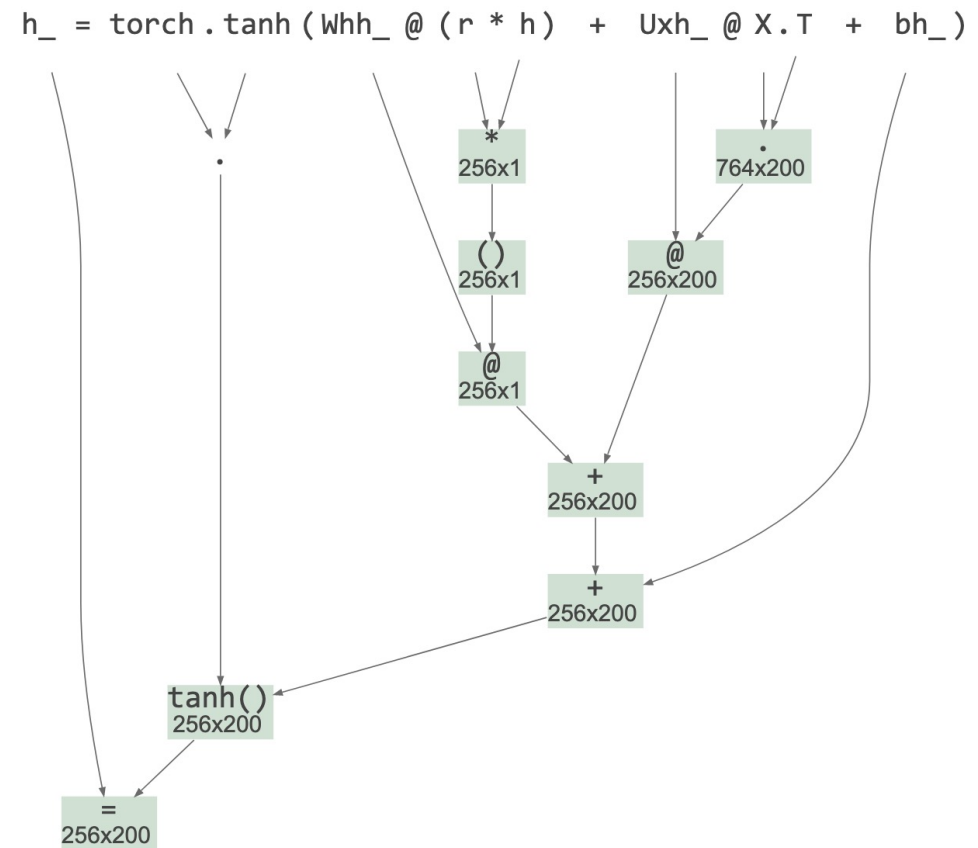
Reevaluation mechanism

- First, check if exception is tensor-related or if exec stack descends into a known tensor lib
- If so, find and parse deepest *user-level* offending statement and build an appropriate AST with operators as subtree roots
- Uses built-in Python tokenizer
- Uses handbuilt Python parser for subset of statements / exprs
- Avoided ANTLR to avoid introducing a lib dependency 😂
- Avoided built-in Python parser since reorg'ing its AST is same work as rolling my own "parrser"



Reevaluation mechanism continued

- Evaluate operators of AST bottom-up in proper exec order
- Call **eval()** on Python source of subexpressions using the appropriate execution contexts, saving results in associated nodes
- Trap and absorb exception from **eval()**, record that exception and offending AST node
- Augment original exception message with info derived this new exception, op, operands



Picking the right execution context

- The goal is to identify user code not library code that (eventually) triggers a tensor-related exception
- TensorSensor **clarify**() descends into any user code function calls, stopping only when it reaches a tensor library function
- Source file prefix indicates user code boundary, such as:
`.../lib/python3.8/site-packages/tensorflow/...`
- Boundary frame is any whose package is in `{numpy, torch, tensorflow, jax}`

Example: Picking the execution frame boundary

$$\text{return } \underbrace{\frac{W}{2}}_{\sim} @ \underbrace{\frac{x}{1}}_{\sim} + b$$

t.py source

```
def f(x):  
    W = tf.constant([[1, 2], [3, 4]])  
    b = tf.reshape(tf.constant([[9, 10]]), (2, 1))  
    → return W @ x + b # line 4  
  
with tsensor.clarify():  
    x = tf.reshape(tf.constant([[8, 5, 7]]), (3, 1))  
    y = f(x) # line 8
```

Execution stack

t.py:8 (in main)
t.py:4 (in f)
math_ops.py:1124
dispatch.py:201
math_ops.py:3253
gen_math_ops.py:5624
ops.py:6843

Raises exception

Simplifying the user code interface

- My prototype used try/except but was unwieldy for the user
- And, the code looked like error handling, which obscured purpose
- Python "**with b**" blocks call `__enter__()`, `__exit__()` on object **b** and exit method receives exception object and execution stack
- Exception handling code can automatically gen visualization

Prototype

```
try:
    myprogram()
except Exception as e:
    tsensor.do_everything(e)
```

(might not be able to hide everything here, like reraising e)

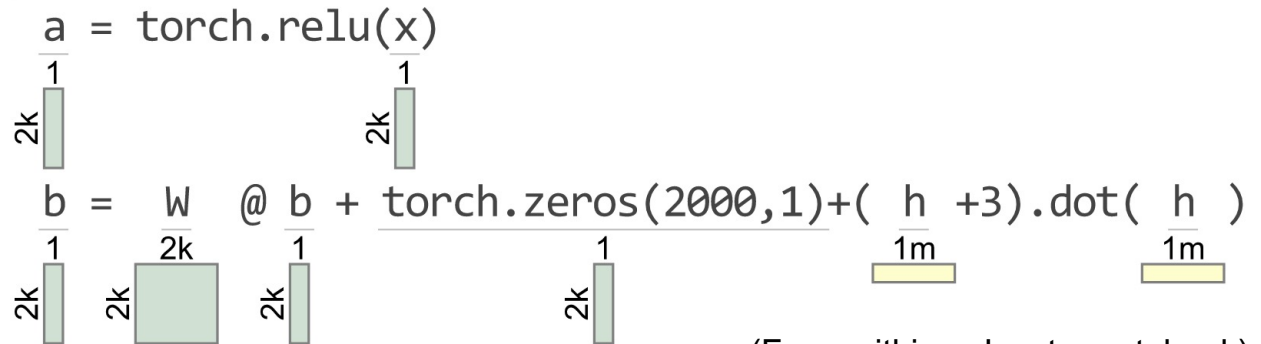
Current

```
with tsensor.clarify():
    myprogram()
```

Explaining correct matrix code

- **clarify()** has no effect unless tensor code triggers an exception
- To aid reading matrix code, **explain()** gens a visualization for each statement within the block

```
import torch
import tsensor
W = torch.rand(size=(2000,2000))
b = torch.rand(size=(2000,1))
h = torch.rand(size=(1_000_000,))
x = torch.rand(size=(2000,1))
with tsensor.explain():
    a = torch.relu(x)
    b = W @ b + torch.zeros(2000,1)+(h+3).dot(h)
```



(From within a Jupyter notebook)

Visualizing Python code on-the-fly

- **explain()** object's **__enter__()** method creates a tracer object and registers it with Python via **sys.settrace()** [1]
- The tracer is notified upon each source line execution
- Using the same mechanism as **clarify()**: parse source line, build AST, evaluate AST subexpressions to identify operand shapes
- Even in loop within **with** block, statements visualized just once
- Slows down execution (a lot) but it's still useful
- Watch out for side effects; this prints "hi" twice:

```
with tsensor.explain():  
    print("hi")
```

[1] <https://docs.python.org/3/library/sys.html#sys.settrace>

Summary

- Language engineering is useful far beyond building compilers and interpreters
- TensorSensor users think that visualization was the hard part, but that was just painful not hard (I abused matplotlib horribly!)
- The tricky bit was getting fine-grained exceptions from Python
 - The key idea is to reevaluate the offending line operator-by-operator and wait for the exception to happen again
 - Involves extracting source line, parsing into an AST, then calling **eval()**
- Finding and implementing an unobtrusive mechanism also took a lot of experimentation
- Article: <https://explained.ai/tensor-sensor/index.html>
- Code: <https://github.com/parrt/tensor-sensor>