Clarifying matrix algebra exceptions in Python visually with TensorSensor

Examples and implementation details

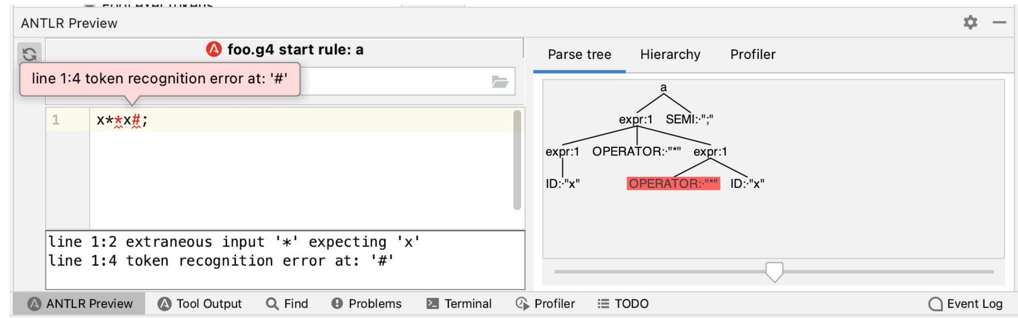
Terence Parr
University of San Francisco

See https://github.com/parrt/tensor-sensor and https://explained.ai/

UNIVERSITY OF SAN FRANCISCO

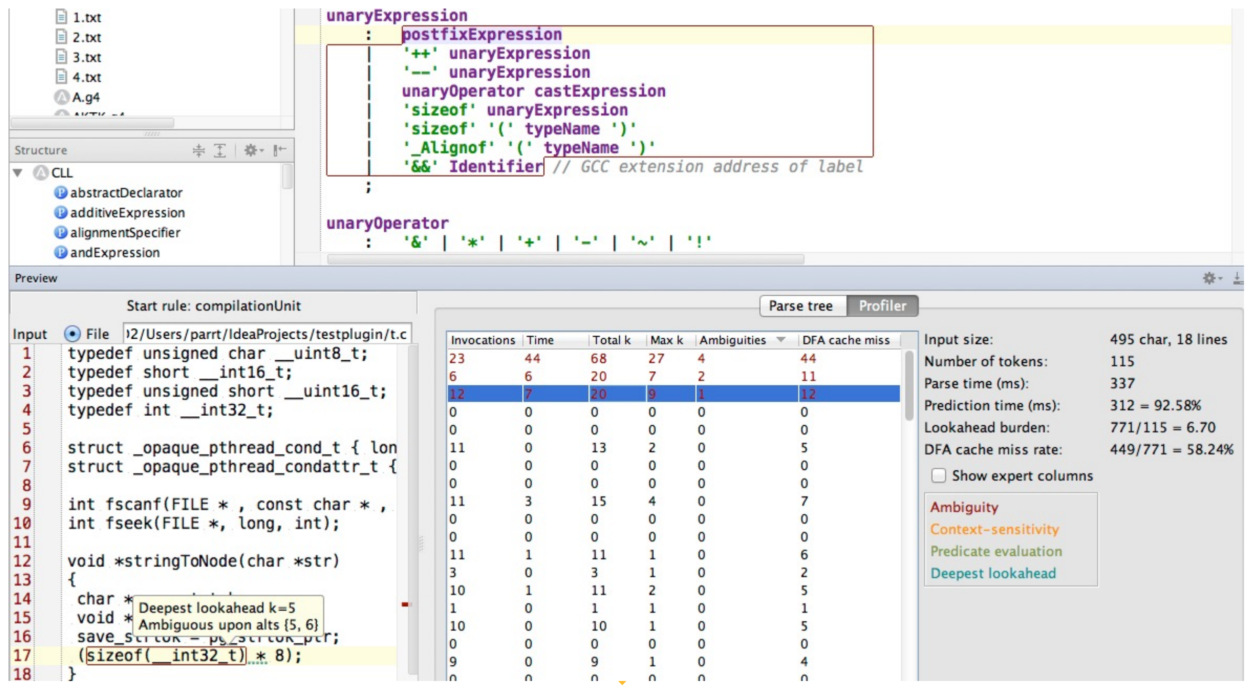
A quick self-intro…

Terence Parr
University of San Francisco



ANTLR

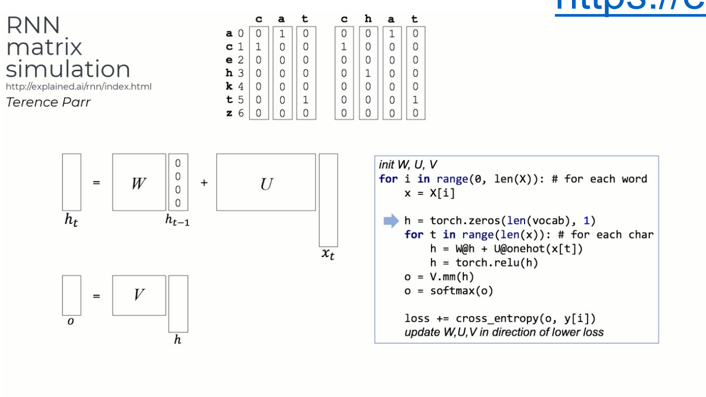
"Why program by hand in 5 days what you can spend 5 years of your life automating?"

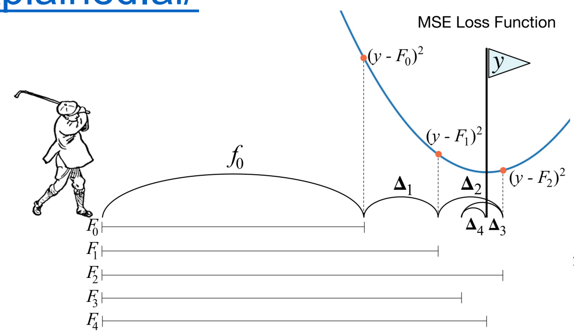


jGuru

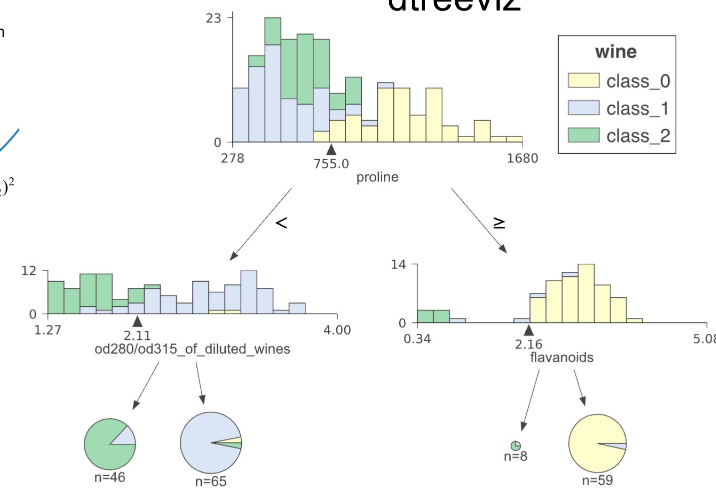
And now for something completely different…

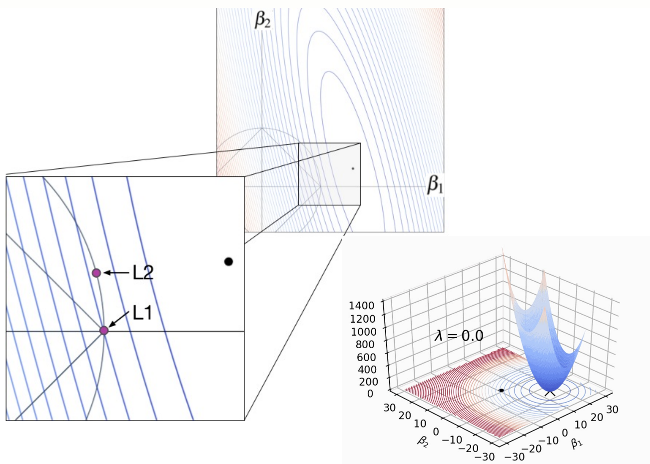
https://explained.ai/



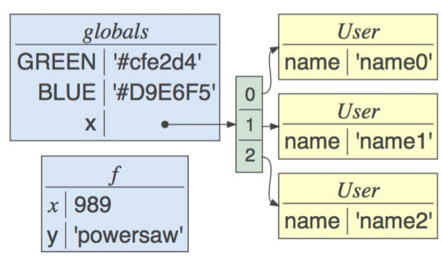


dtreeviz

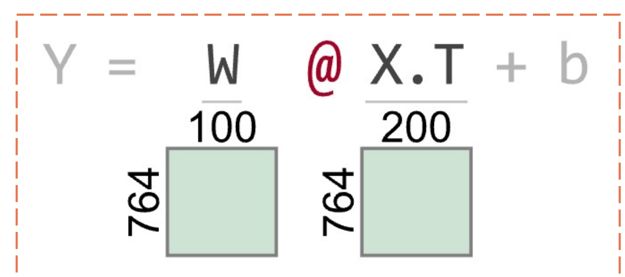




lolviz



TensorSensor



UNIVERSITY OF SAN FRANCISCO

The problem

```python
import torch

n = 200
d = 764
nhidden = 256

Whh = torch.eye(nhidden, nhidden)
Uxh = torch.randn(d, nhidden)
bh  = torch.zeros(nhidden, 1)
h = torch.randn(nhidden, 1)
r = torch.randn(nhidden, 1)
X = torch.rand(n,d)

# Following code raises an exception
h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)
```

- It's easy to lose track of matrix/tensor dimensionality in matrix algebra expressions (even in statically-typed languages)
- Upon error, we often get less than helpful exception messages, such as this (PyTorch) message
- The offending operator and operands are not identified, since Python exceptions occur at the line level rather than the operator level

```
RuntimeError: mat1 and mat2 shapes cannot be multiplied (764x256 and 764x200)
```

We could rerun using the debugger but…

- The debugger still does not tell us which subexpression caused the exception, due to line-level granularity of Python exceptions
- We must write down shape of all operands then line up and compare dimensions on all subexpressions manually
- Besides
  - 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

What most people do (in notebooks)

- Most data scientists laboriously inject code and rerun to isolate:

Or, they stop here

```
print(Whh.shape, r.shape, h.shape, Uxh.shape, X.shape, bh.shape)
print((r*h).shape)
print((Whh@(r*h)).shape)
print((Uxh@X.T).shape) # <-- exception!
print((Whh@(r*h)+Uxh@X.T).shape)
print((Whh@(r*h)+Uxh@X.T+bh).shape)
h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)
```

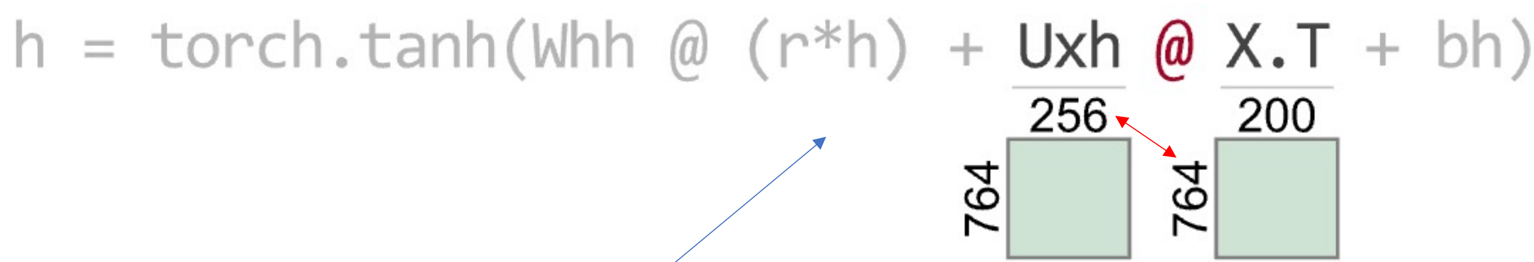
```
RuntimeError                              Traceback (most recent call last)
<ipython-input-2-b5160030ac99> in <module>
     16 print((r*h).shape)
     17 print((Whh@(r*h)).shape)
---> 18 print((Uxh@X.T).shape)
     19 print((Whh@(r*h)+Uxh@X.T).shape)
     20 print((Whh@(r*h)+Uxh@X.T+bh).shape)
```

What TensorSensor proposes

- First, augment the exception message to identify the op/opnds:

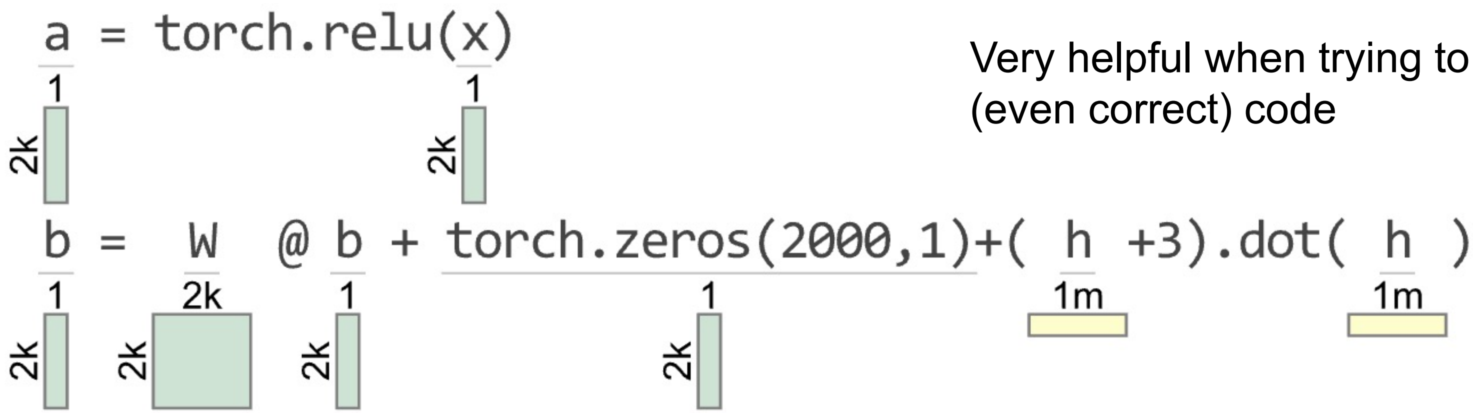
```
RuntimeError: mat1 and mat2 shapes cannot be multiplied (764x256 and 764x200)
Cause: @ on tensor operand Uxh w/shape [764, 256] and operand X.T w/shape [764, 200]
```

- But we can help programmers even more…
- The key is to line up dimensions, so let's show that visually!

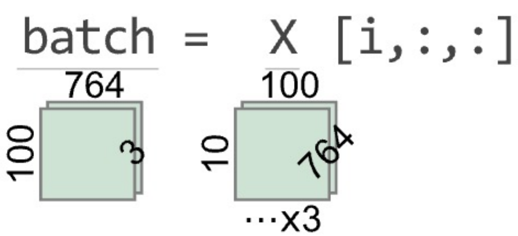


Believe it or not, this is all **matplotlib**

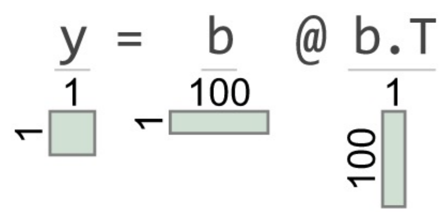
A nice-to-have feature: viz correct code



Very helpful when trying to read (even correct) code

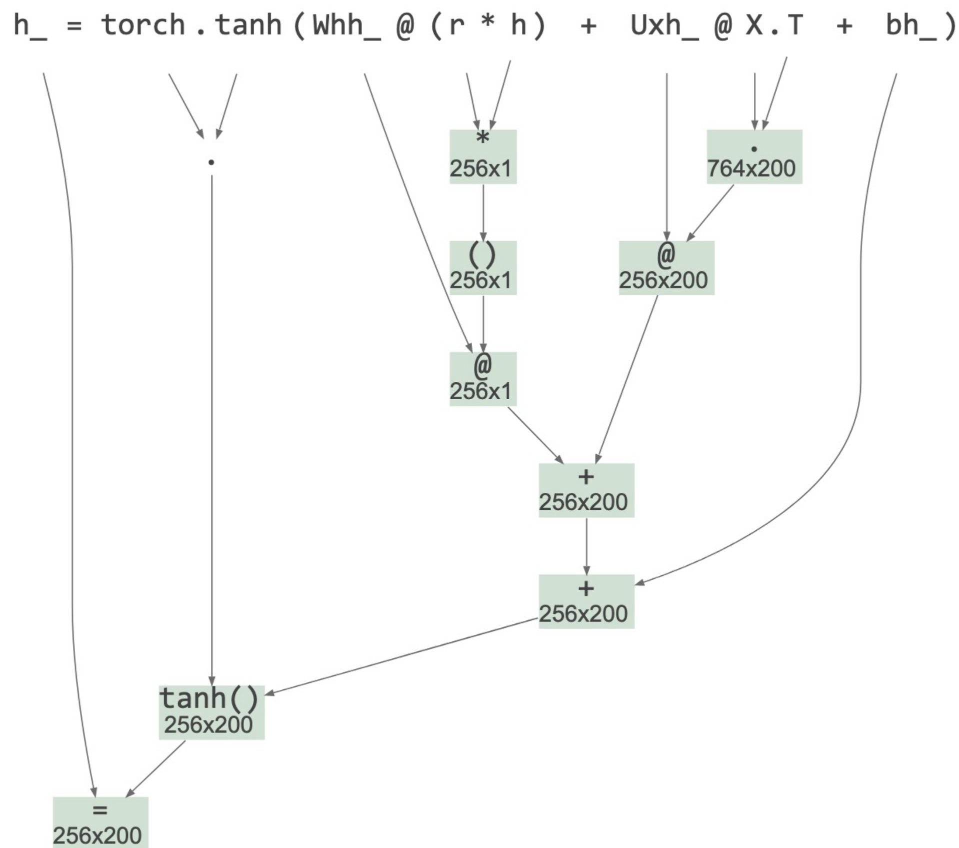


Greater than 2 and 3 dimensions



A nice-to-have feature:
Abstract syntax trees with dimensions

Shows tensor dimensions for all subexpression partial results (currently must call explicit function)

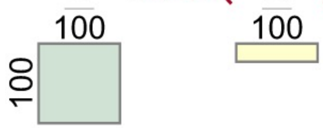


Oh, and support multiple libraries

(1D vectors are yellow)

PyTorch

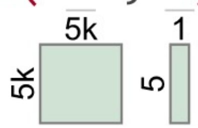
```
W .dot( h ) + x
```




```
RuntimeError: 1D tensors expected, but got
2D and 1D tensors
Cause: W.dot(h) tensor arg h w/shape [100]
```

JAX

```
y = jnp.dot( W , x) + b
```




```
TypeError: Incompatible shapes for dot: got (5000, 5000)
and (5, 1).
Cause: jnp.dot(W, x) tensor arg W w/shape (5000, 5000),
arg x w/shape (5, 1)
```

TensorFlow

```
Y = W @ tf.transpose( X ) + b
```


```
InvalidArgumentError: In[0] mismatch In[1] shape:
100 vs. 764: [764,100] [764,200] 0 0 [Op:MatMul]
Cause: @ on tensor operand W w/shape (764, 100) a
nd operand tf.transpose(X) w/shape (764, 200)
```

Design goals

(knowing what to build is as important as knowing how to build)

- Should be as unobtrusive as possible with least user effort
- Users shouldn’t have to reorganize code
- Trap just matrix-related exceptions
- Avoid need for an external tool or translator
- Avoid spewing output until exception occurs
- Can we avoid CPU cost until an exception?
- Ideally, implementation would be small and straightforward (at least to language engineers)

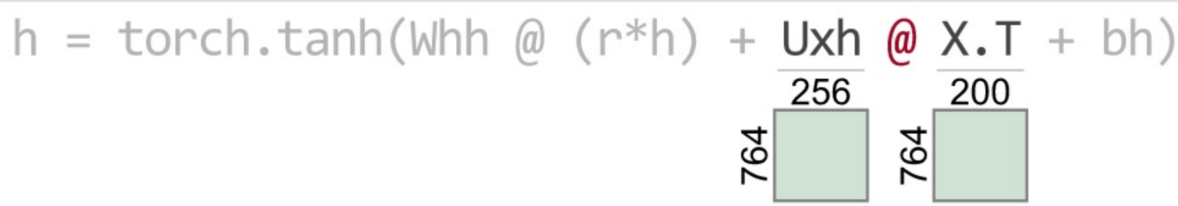
TensorSensor programmer interface

- The Python **with** statement gives us the hooks we need

```
import tsensor

with tsensor.clarify():
    h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)
```

(From within a Jupyter notebook)



No need to inject code
No need to rerun

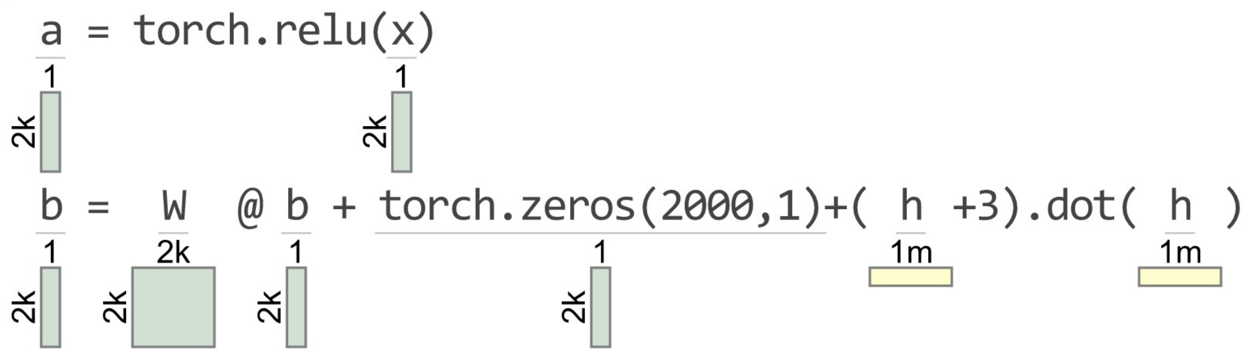
```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-13-b9a515efa8ef> in <module>
      2
      3 with tsensor.clarify():
----> 4     h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (764x256 and 764x200)
Cause: @ on tensor operand Uxh w/shape [764, 256] and operand X.T w/shape [764, 200]
```

Explaining correct matrix code

- **clarify**() has no effect unless tensor code triggers an exception
- But **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)
```



Approaches I rejected

- Python decorators: would require wrapping user code in functions
- Try/except blocks
- Program rewriting is complex and requires a separate tool
- Bytecode injection:
  - slows down entire program
  - could require huge cache of subexpression partial results
  - function-level granularity

```
try:
    … my code …
except Exception as e:
    tsensor.do_everything(e)
```

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

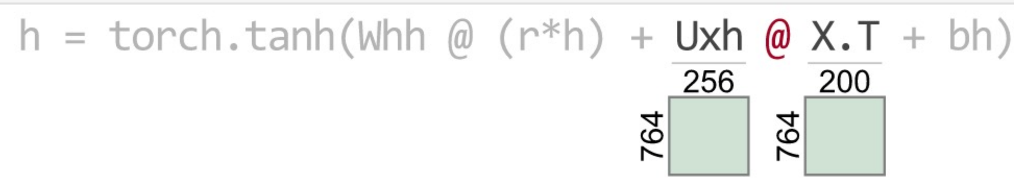
TensorSensor implementation

Impl. relies on “context manager” objects

- Python "**with b**" blocks call **__enter__**(), **__exit__**() on object **b** and exit method receives exception object and execution stack
- **clarify()** needs the exception object to augment messages and the execution stack to obtain subexpression values, identify offending operator
- **__exit__**() can automatically gen visualization in notebook or pop-up a window if run outside a notebook
- There’s no cost unless exception occurs within the **with** block

```
import tsensor

with tsensor.clarify():
    h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)
```



TensorSensor’s explain() mechanism
(Visualizing correct 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 same mechanism as **clarify**() to identify operand shapes
- Even in a loop within **with** block, statements visualized just once
- Slows down execution (a lot) but it's still useful

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

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

- We have: (1) the full execution stack from which we can get (2) the offending line of source code (**inspect.getframeinfo()**)
- 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
- Assumes side-effect free operations
- Even if side-effecting, who cares (usually)? The program is about to terminate

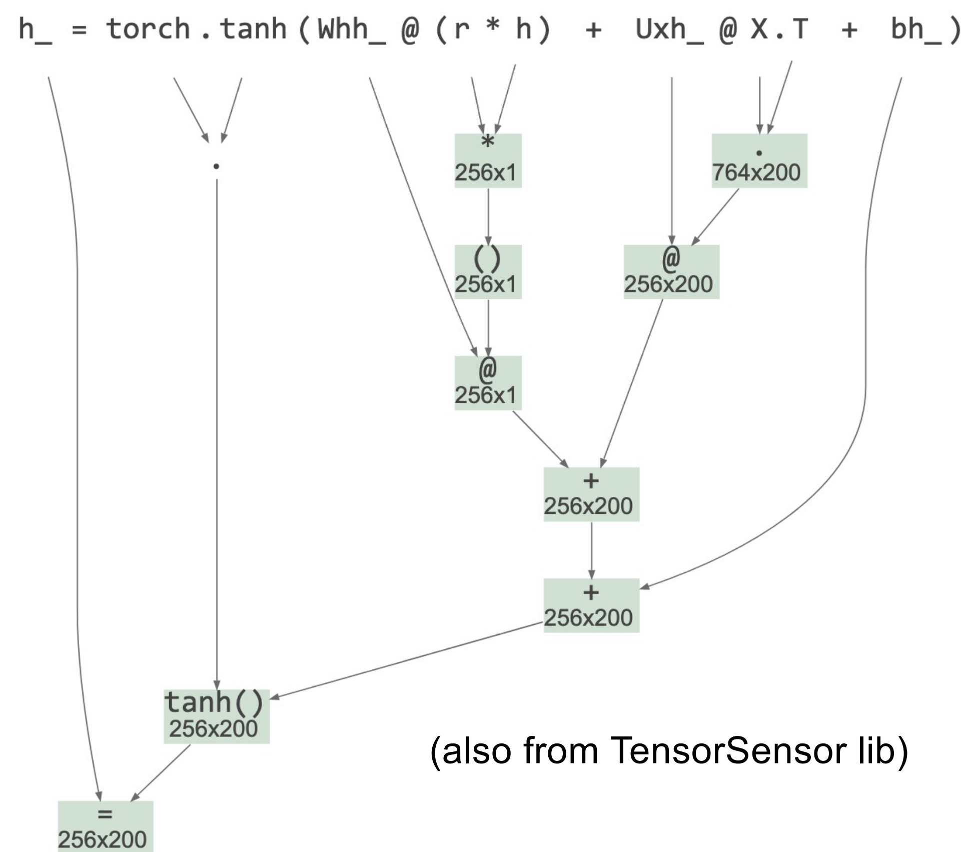
Warning:

Prints "hi" twice:

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

Reevaluation mechanism

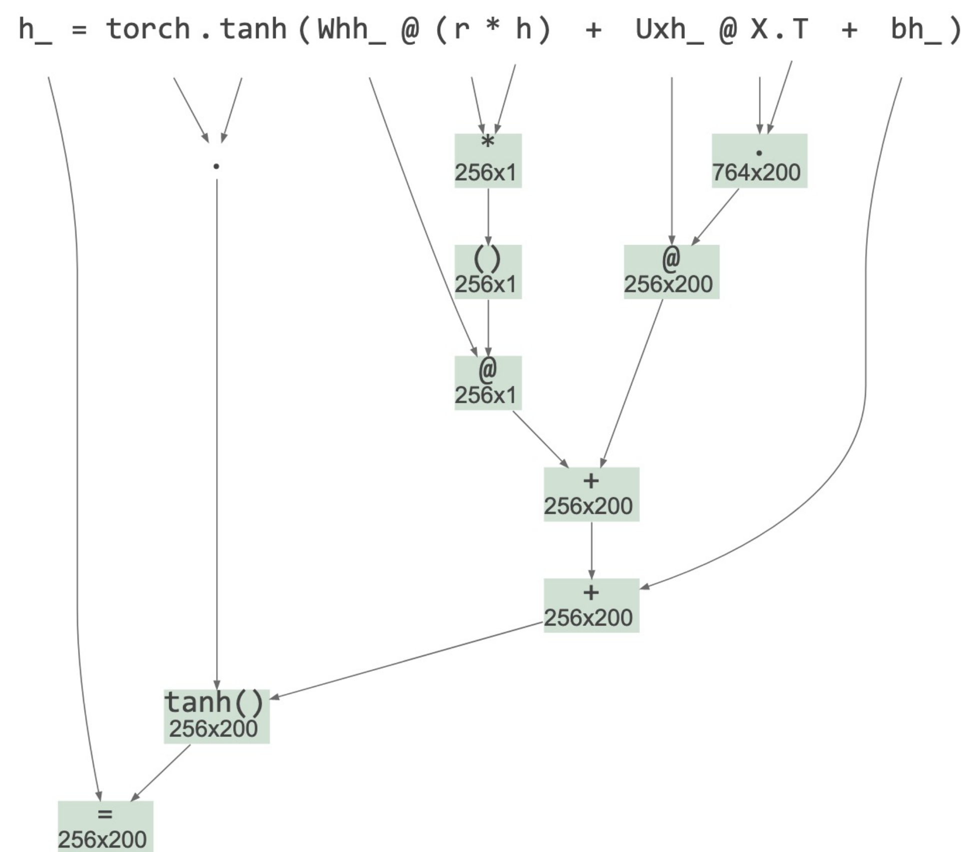
- First, check if exception is tensor-related or if exec stack descends into a known tensor lib
- If so, scan stack to find and parse deepest *user-level* offending statement and build an appropriate AST with operators as subtree roots
- Uses built-in Python tokenizer
- Uses handbuilt Python parser for 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"



(also from TensorSensor lib)

Reevaluation mechanism continued

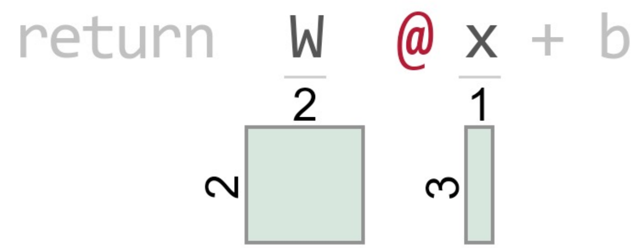
- Evaluate operators of AST bottom-up in proper exec order
- Call **eval()** on Python source of subexpressions using the appropriate execution contexts, saving results in associated nodes
- Trap and absorb exception from **eval()**, record that exception and offending AST node
- Augment original exception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



`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
```

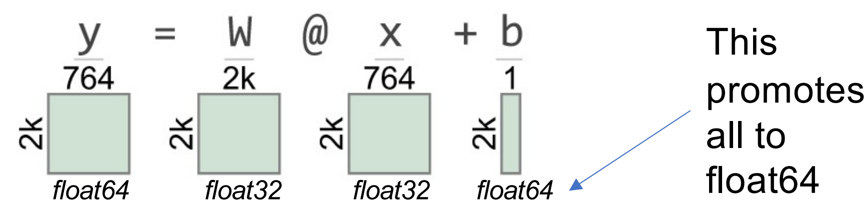
Execution stack

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

Examples of future work

- Lots of meat still on the bone
- Add tensor element type to messages and visualizations
  - we don't want integers becoming floats if they are used as indexes
  - might need to restrict to 32 bits



- Viz errors in predefined layers; currently highlights **model(X)** not layer in **nn.Sequential**

```
model = nn.Sequential(
    nn.Linear(784, n_neurons), # 28x28 flattened image
    nn.ReLU(),
    nn.Linear(10, n_neurons),  # 10 output classes (0-9)
    nn.Softmax(dim=1)
)
X = torch.rand(n,784) # n instances of feature vectors w
with tsensor.clarify():
    Y = model(X)
```

Optimally, we'd have static typing

- Can we capture shape as part of the type in this special case? (dimensions are dynamic so capture variable names?)

```
Uxh = torch.randn(d, nhidden)   ←  Capture tensor('d','nhidden')
X   = torch.rand(n, d)          ←  Capture tensor('n','d')
```

- Then, we can give good error messages statically!

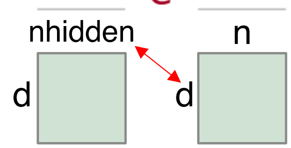
Option 1

```
h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)
                                 ^
```

Incompatible types: @ on tensor operand Uxh w/shape [d, nhidden] and operand X.T w/shape [d, n]

Option 2

```
h = torch.tanh(Whh @ (r*h) + Uxh @ X.T + bh)
```



Summary

- Finding and implementing an unobtrusive mechanism took a lot of experimentation (and had to learn about Python's rich runtime)
- 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 the source line, parsing into an AST, then calling **eval**()
- Language engineering is useful far beyond building compilers and interpreters
- Article: https://explained.ai/tensor-sensor/index.html
- Code: https://github.com/parrt/tensor-sensor