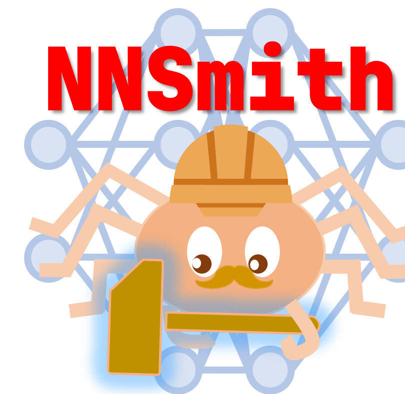
NNSmith

Generating Diverse and Valid Test Cases for Deep Learning Compilers

Jiawei Liu, Jinkun Lin (co-prim.), Fabian Ruffy
Cheng Tan, Jinyang Li, Aurojit Panda, Lingming Zhang

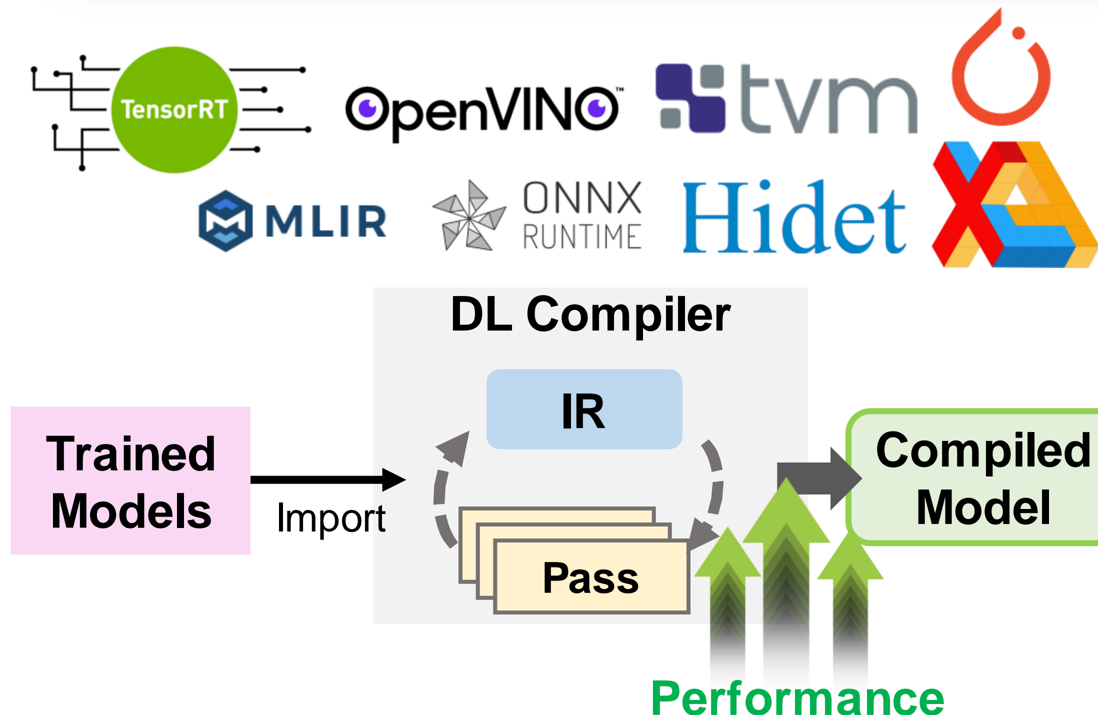


Deep-Learning models are being compiled!

DL compilers are being used to optimize model computation!

In five years, more than 350,000 developers across 27,500 companies in wide-ranging areas, including healthcare, automotive, finance and retail, have downloaded TensorRT nearly 2.5 million times. TensorRT applications can be deployed in hyperscale data centers, embedded or automotive product platforms.

https://nvidianews.nvidia.com/news/nvidia-inference-breakthrough-makes-conversational-ai-smarter-more-interactive-from-cloud-to-edge



PYTORCH 2.X: FASTER, MORE PYTHONIC AND AS DYNAMIC AS EVER

Today, we announce `torch.compile`, a feature that pushes PyTorch performance to new heights and starts the move for parts of PyTorch from C++ back into Python. We believe that this is a substantial new direction for PyTorch – hence we call it 2.0. `torch.compile` is a fully additive (and optional) feature and hence 2.0 is 100% backward compatible by definition.

https://pytorch.org/get-started/pytorch-2.0/

Fast and scalable

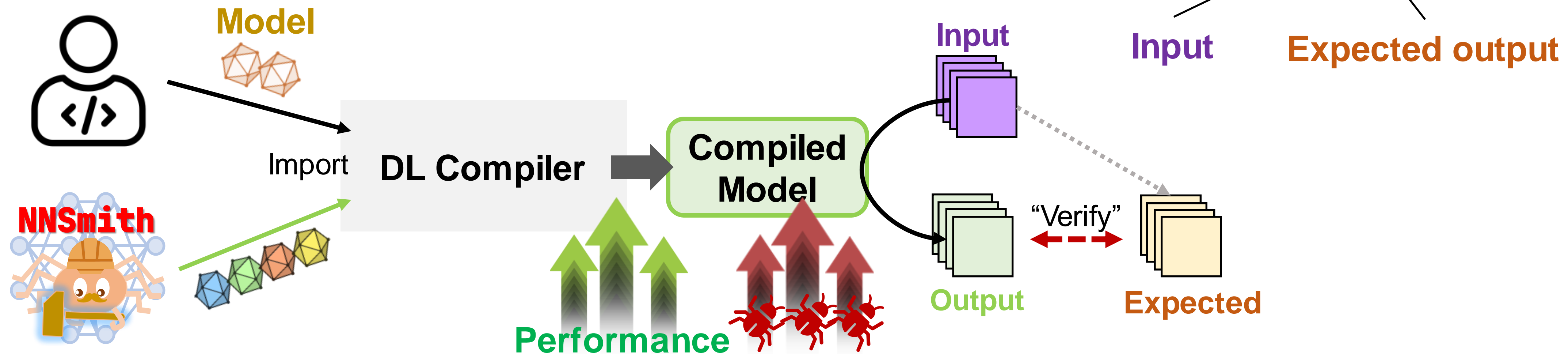
- XLA Compilation. We are focusing on XLA compilation and aim to make most model training and inference workflows faster on GPU and CPU, building on XLA's performance wins on TPU. We intend for XLA to become the industry-standard deep learning compiler,

https://blog.tensorflow.org/2022/10/building-the-future-of-tensorflow.html

Compiler correctness is crucial *but* challenging!

- The compiler stack is **complex**: various operators, passes, targets, etc.
- **~36/42/11%** code in PyTorch/TVM/TensorFlow are testing code*!
  - Developer-made testing models can be repetitive

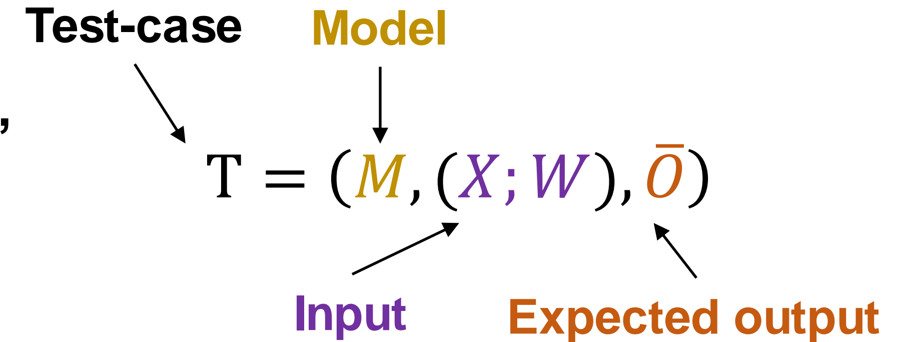
Can we *automatically* generate and diversify test-cases for DL compilers?



*cloc --include-lang=Python,C++ --match-d=test --exclude-dir=external,third_party,3rdparty .

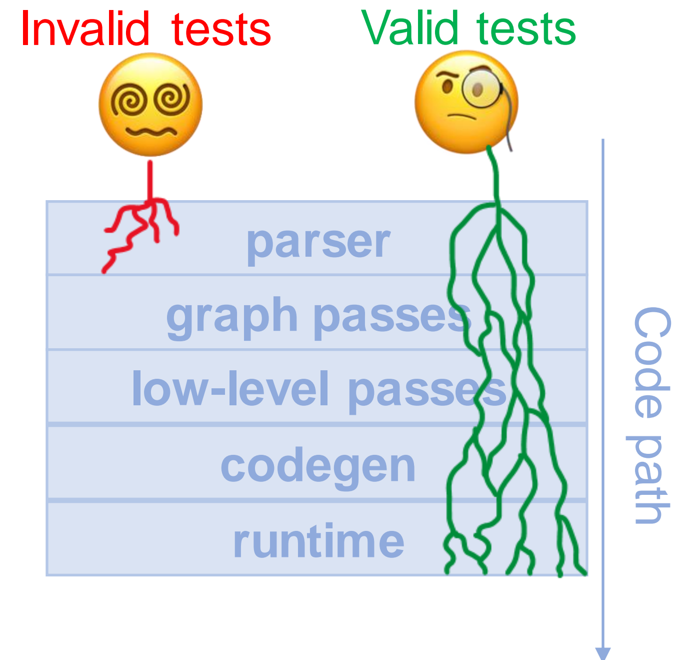
The test-case generation problem

- **Well-formedness** of M and $X; W$:
  - M: **validly constructed operators and "connections"**
  - $X; W$: avoiding NaN and Infinities



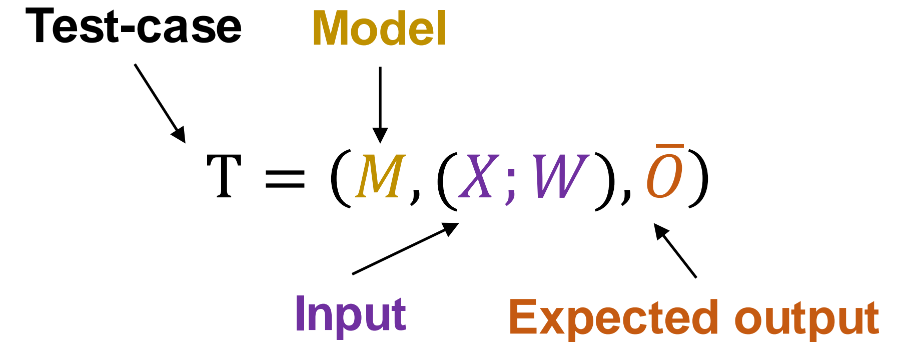
Invalid Model
`ksize` larger than input

```
x = … # shape=[1,3,32,32]
y = avg_pool(x, ksize=33)
```



The test-case generation problem (cont.)

- **Well-formedness** of M and $X; W$:
  - M: validly constructed operators and "connections"
  - $X; W$: **avoiding NaN and Infinities**



UB-inducing
Casting NaN to int is UB

```
x = … # has NaN
y = cast(x, to=int)
```

False positive

Bug-swallowing
NaN/Inf broadcasting

```
x = pow(…) # -> ∞
y = 2+x    # -> ∞
BUGGY_OP(∞)  -> ∞
```

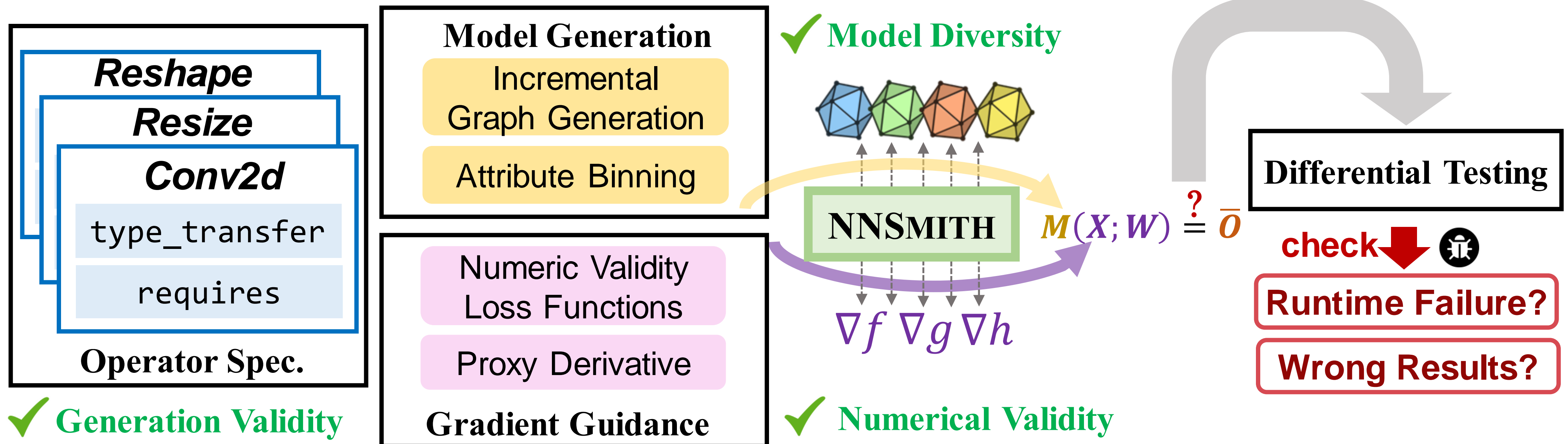
∞ broadcasts to outputs, not inducing the bug.

False negative

Only manifests for non-inf inputs.

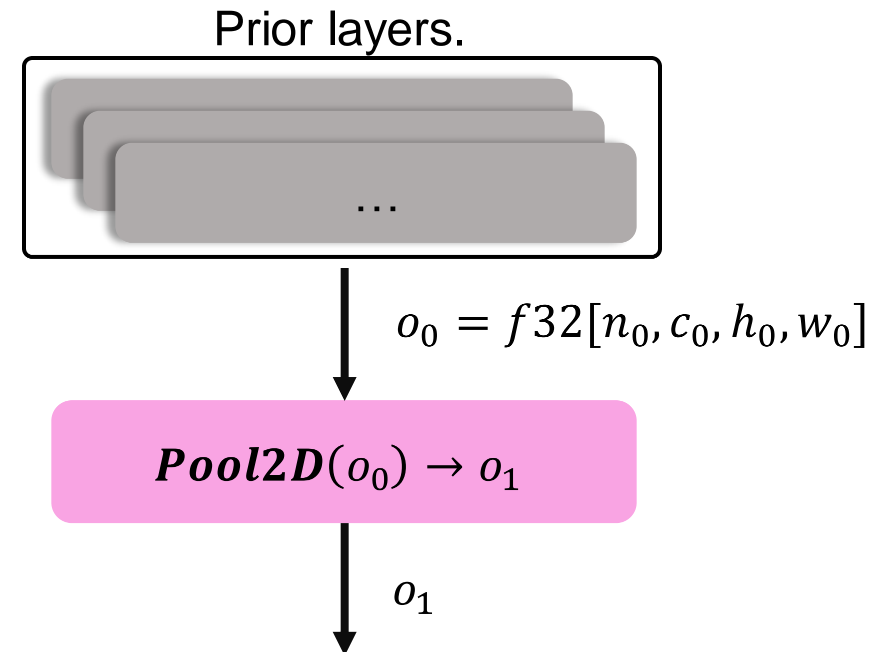
Introducing NNSMITH

- Randomized testing for DL compilers with **well-formed test-cases**:
  - Defining operator spec. for valid model generation.
  - Constructing diverse models incrementally w/ solvers.
  - Avoiding NaN/Inf with gradient guidance.
  - Differential testing to manifest bugs.



Operator specification :: Input constraints

- **Building valid models:** a valid model —validity-preserving insert operator→ a *larger* valid model.
- **Input constraints:** a set of predicates over input shape dims* & attributes.
- **Example (Pool2D)**:
  - (Input) shape dims: $o_0 = [n_0, c_0, h_0, w_0]$
  - Attributes: k{h,w}, pad_{h,w}, stride_{h,w}, …
  - **Constraints**:
    - $0 < \text{kh} \leq 2 \times \text{pad_h} + h_0$
    - $0 < \text{kw} \leq 2 \times \text{pad_w} + w_0$
    - n_0, c_0, h_0, w_0,stride,… > 0
    - …



*data types are omitted for clarity.

Operator specification :: Input constraints

- **Building valid models:** a valid model —validity-preserving insert operator→ a *larger* valid model.
- **Input constraints:** a set of predicates over input shape dims* & attributes
- **Example (Pool2D):**
  - (Input) shape dims: $o_0 = [n_0, c_0, h_0, w_0]$
  - Attributes: k{h,w}, pad_{h,w}, stride, …
  - **Constraints:**
    - $0 < \text{kh} \leq 2 \times \text{pad_h} + h_0$
    - $0 < \text{kw} \leq 2 \times \text{pad_w} + w_0$
    - n_0, c_0, h_0, w_0,stride,… > 0
    - …

Spec. code

```
class Pool2d(OpBase):
  def __init__(s, kh, kw, pad_h, ...):
    s.kh = kh  # height of kernel size
    …

  def requires(s, itensors) → List[Predicate]:
    ih, iw = itensors[0].shape[2:]
    return [
      0 < s.kh < 2 * s.pad_h + ih,
      0 < s.kw < 2 * s.pad_w + iw,
      0 < s.stride, … ]

  def type_transfer(s, itensors) → List[ATensor]:
    n, c, ih, iw = itensors[0].shape
    oh = (ih + 2*s.pad_h - s.kh) // s.stride + 1
    ow = (iw + 2*s.pad_w - s.kw) // s.stride + 1
    return [  ATensor(
                shape=(n, c, oh, ow),
                dtype=itensors[0].dtype)   ]
```

Operator specification :: Type propagation

How do we know the input shapes?

What is $n_0, c_0, h_0, w_0 \ldots$?

```
def requires(s, itensors ← o0):
  ih, iw = itensors[0].shape[2:]
  return [
    0 < s.kh < 2 * s.pad_h + ih,
    0 < s.kw < 2 * s.pad_w + iw,
    0 < s.stride, … ]
```

$o_0 = f32[n_0, c_0, h_0, w_0]$

$\boldsymbol{Pool2D}(o_0) \rightarrow o_1$

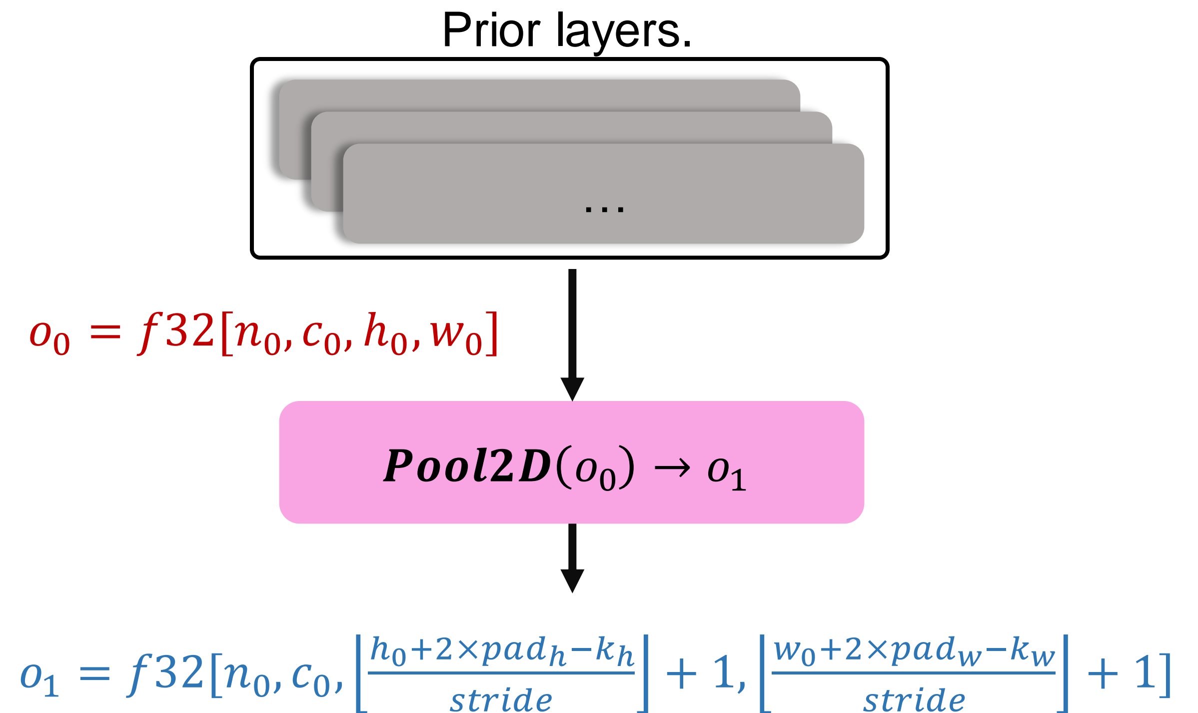
o_1

Operator specification :: Type propagation

- **Type propagation:**
  - A function maps input shape dims & attributes to output shape dims.

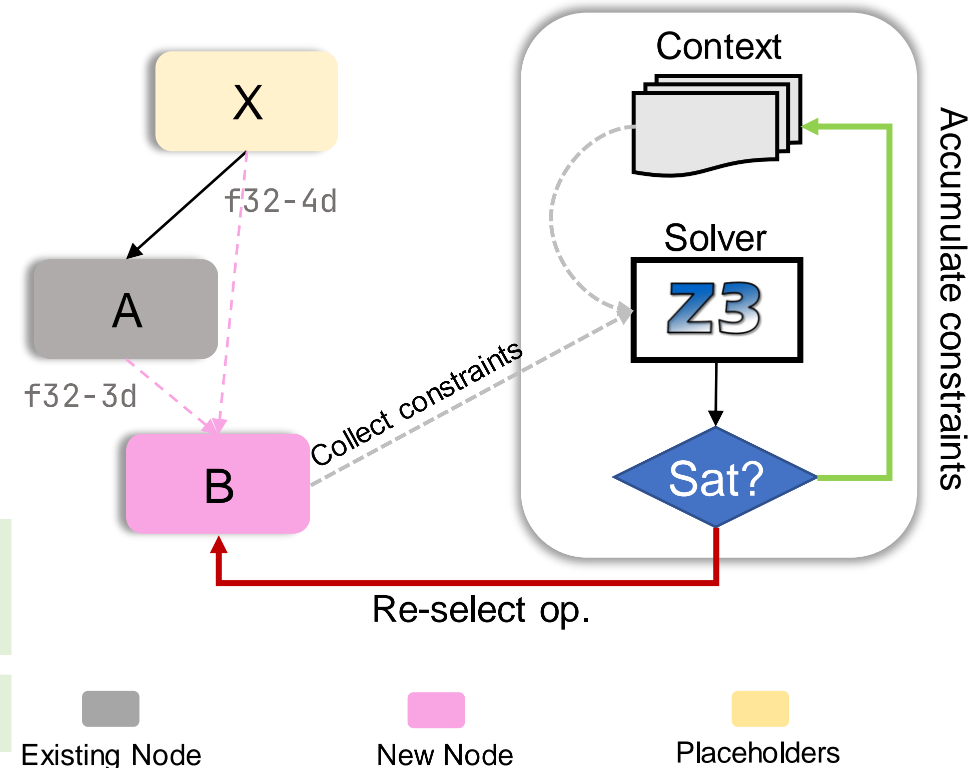
```
class Pool2d(OpBase):
  …
  def requires(s, itensors) → List[Predicate]:
    ih, iw = itensors[0].shape[2:]
    return [
      0 < s.kh < 2 * s.pad_h + ih,
      0 < s.kw < 2 * s.pad_w + iw,
      0 < s.stride, … ]

  def type_transfer(s, itensors) → List[ATensor]:
    n, c, ih, iw = itensors[0].shape
    oh = (ih + 2*s.pad_h - s.kh) // s.stride + 1
    ow = (iw + 2*s.pad_w - s.kw) // s.stride + 1
    return [  ATensor(
                shape=(n, c, oh, ow),
                dtype=itensors[0].dtype)  ]
```



Incremental model construction

- **Building valid models:** a valid model —validity-preserving insert operator→ a *larger* valid model.
  - *Incremental insertion* avoids making *un*connected graph.
- **Forward insertion:** Insert a leaf operator.
  - Select an operator (B is an **add** op.)
  - Select compatible producer (A, X)
    - *Data type*: A & X are both float32.
    - *Ranks*: 3- and 4-dims (broadcasting).
  - Solve constraints:
    - **Sat**: insert B and accum. constraints.
    - **Unsat**: retry other operator candidates.



Backward insertion: Similarly, operators can be inserted backward by replacing placeholders. (Detailed in paper)

Attr binning: Diversify op's attributes. (Detailed in paper)

What makes ill-formed model inputs?

- Computing **57%** of generated models (20-node) incurs **NaNs/Infinities.**
- **Where are NaNs/Infinities come from?**
  - Log(X), Asin(X), … when elements from X < 0.
  - Pow(X, Y) when elements in X and Y are too large.

NaN/Inf occurs when operators' inputs violate its *stable* domain.

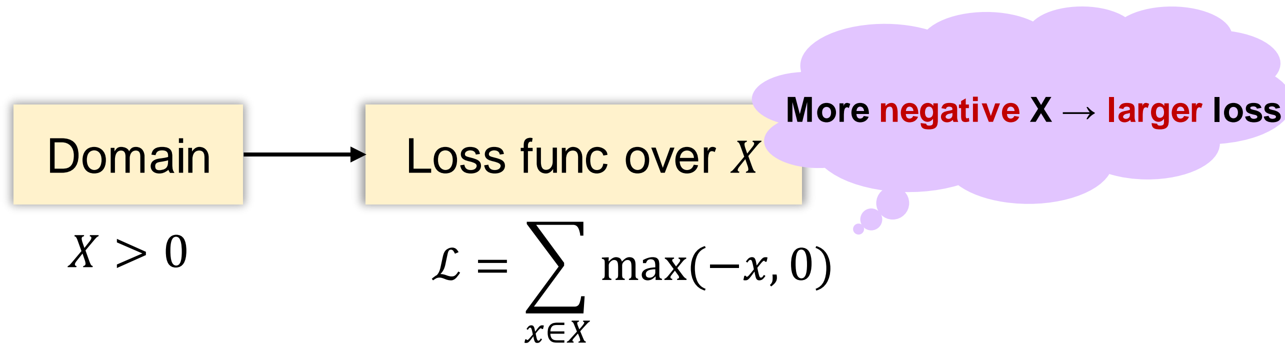
Vulnerable operator: operators with limited domain.

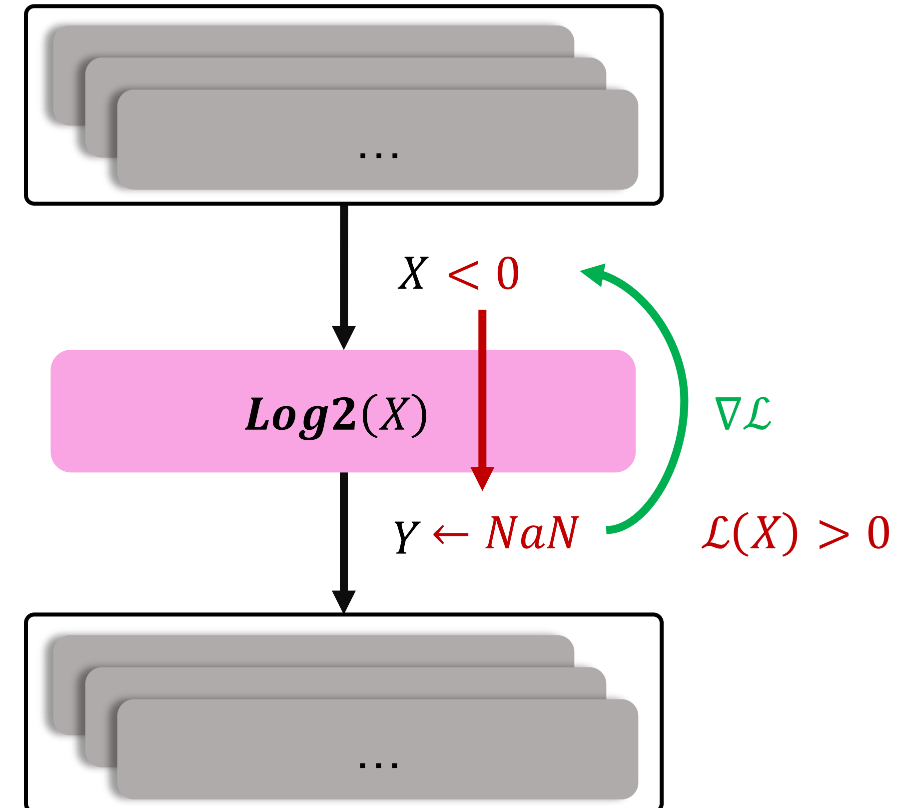
Operator	Domain	Violation
Asin(X)	$\|X\| \leq 1$	*NaN*
Div(X, Y)	$\|Y\| > 0$	*NaN*
Pow(X, Y)	$\begin{cases} X > 0 \\ Y \log(X) \leq 40 \end{cases}$	*NaN/Inf*
Log2(X)	$X > 0$	*NaN*

*Yan, Ming, et al. "Exposing numerical bugs in deep learning via gradient back-propagation."

Tuning model inputs via *gradients*

- use **loss functions** and **backprop** to tweak the weights in-domain.
  1. For $Y = \boldsymbol{VulOp}(X)$ check if $\mathrm{NaN} \,/\, \infty \in Y$
  2. If yes, map the *domain* to a *loss function* $\mathcal{L}$
  3. Apply $\mathcal{L}$ to the input (X) and do backward prop.





Some op. are non-differentiable or have 0-gradient regions!
Use "*proxy gradients*" to mimic the effect! (Detailed in paper)

Detecting *new* bugs

72 bugs found; 51 fixed.

- NNSMITH tests nightly versions of **3** compilers (and PT exporter by product)
- NNSMITH finds **72** bugs, **51** of which have been **fixed**
  - **17** are **semantic** bugs (others: crashing bugs)
  - **43** are caused by erroneous **passes** (others: converter bugs)

	Transformation	Conversion	Other	Total
ONNXRuntime (ORT)	10	0	2	**12**
TVM	29	11	0	**40**
TensorRT	4	2	4	**10**
PyTorch Exporter	-	10	-	**10**
Total	43	23	6	**72**

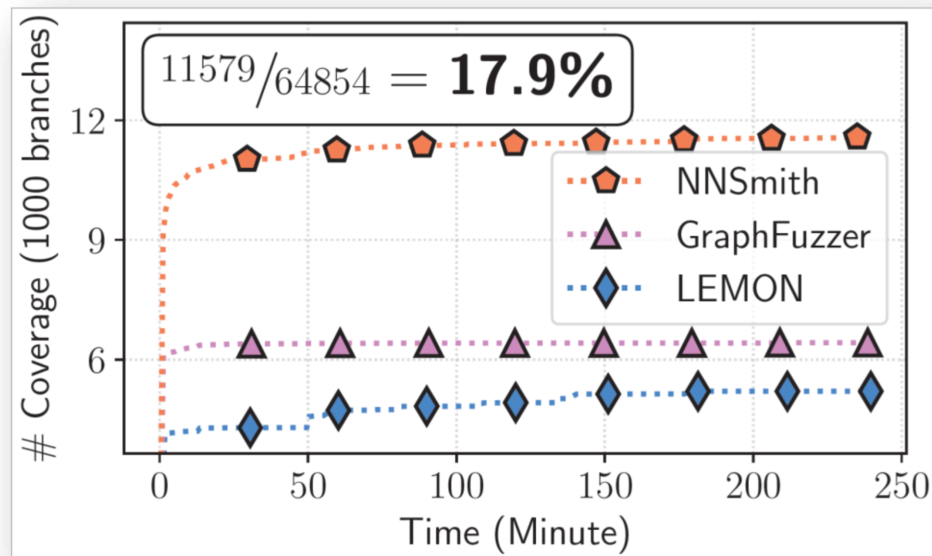
`FuseReLUClip` `SimplifyConsecutiveCast` `FuseMatMulScale` `GemmTransposeFusion`

`MergeShapeInfo` `SoftmaxSchedule` `ConcatSchedule` `SplitModConst` `NarrowDataType`

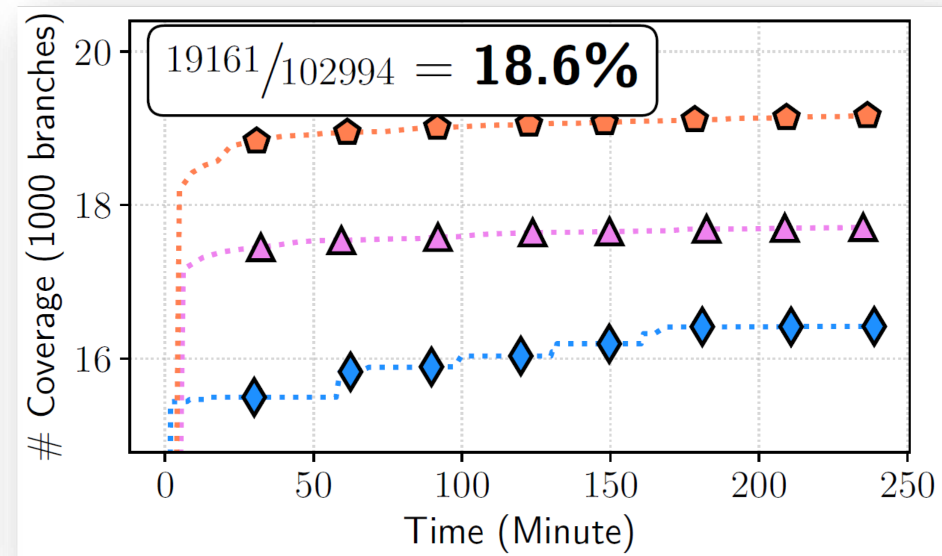
`BinaryBroadcastLayoutHelper` `ReduceInferCorrectLayout` `GetStoreRule` `StridedSliceInferCorrectLayout`

Branch coverage

- Fuzzing ORT/TVM for **4 hours** and record **branch coverage**
- **18~19%** system-wide branch coverage
- **ORT**'s improvement is ***larger*** as it's pattern-sensitive (w/ more graph passes)



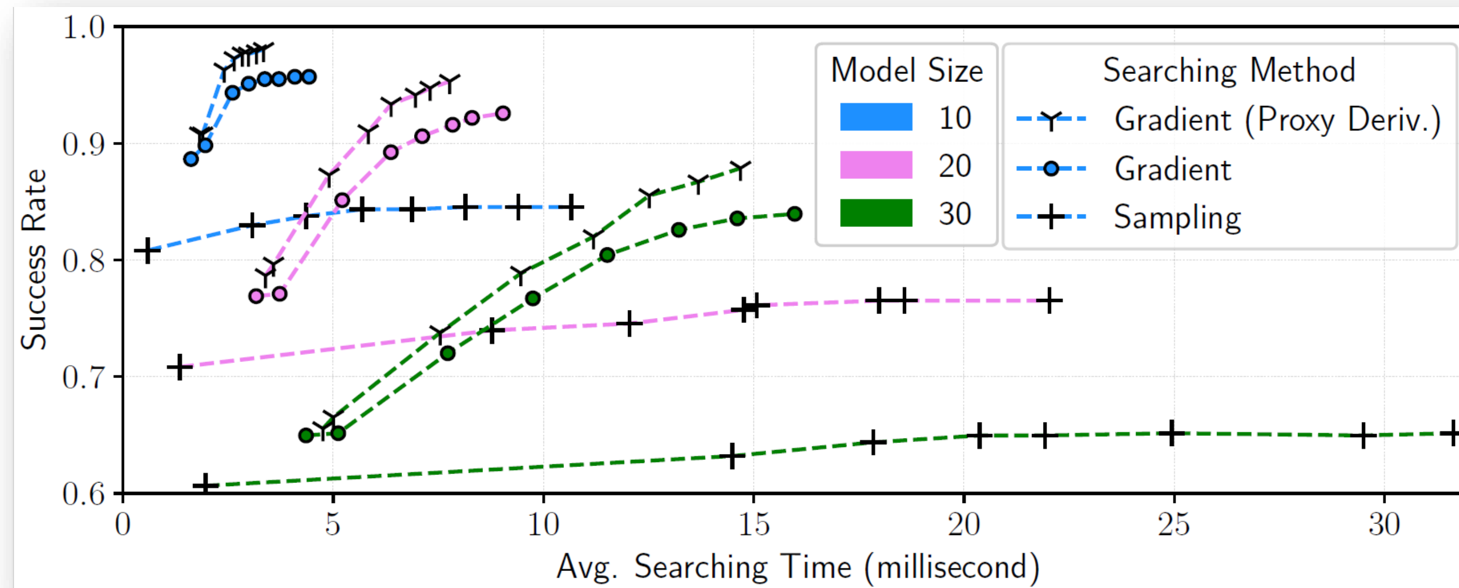
ONNXRuntime@CPU



TVM@LLVM

Validity rate of model inputs/weights

- Gradient guidance finds NaN/∞-free input for **98%** models in **3.5ms** (each)*.
- Input validity becomes **more challenging** to preserve for **lager-sized** models.



*Base of 512 randomly generated 10-node models.

Summary of NNSMITH

- **Well-formed-oriented test-case generation** for **DL compilers**!
  - Operator specification for expressing validity essentials.
  - Generating random & valid models incrementally.
  - Searching well-formed model inputs with gradients.
- Finding new bugs in real world

72 bugs found; 51 fixed

- Test your DL compiler with NNSMITH!

paper

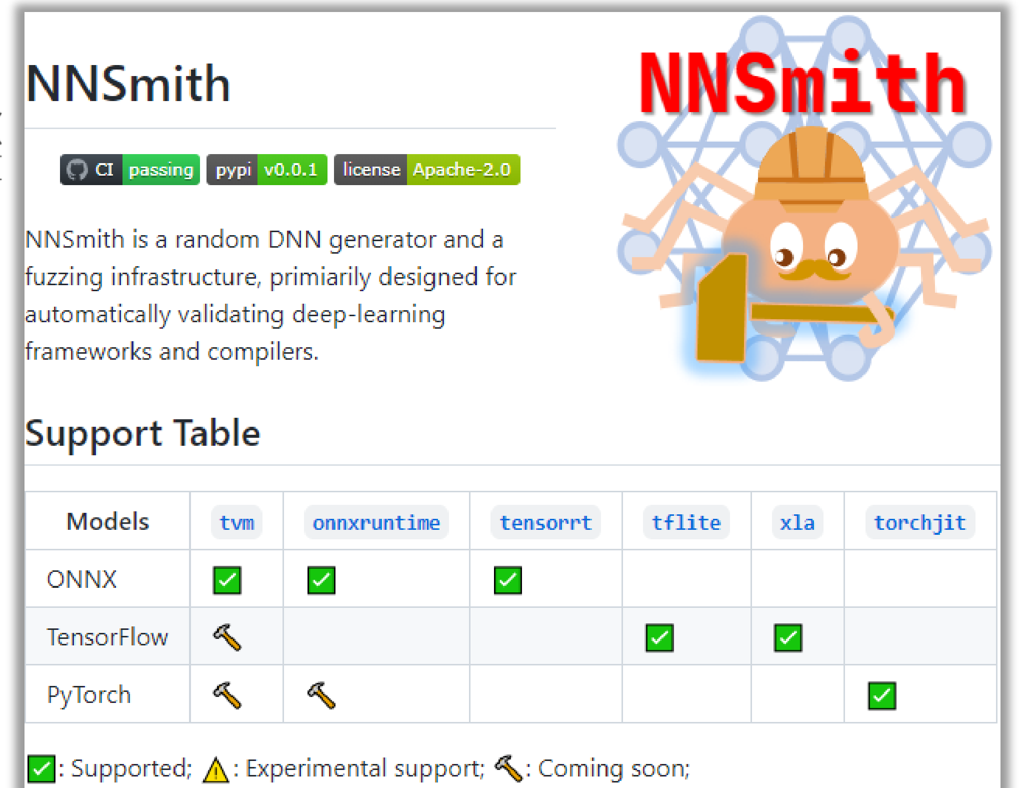
nnsmith

ise-uiuc/nnsmith

code

ganler/nnsmith-asplos23-ae

nnsmith-asplos.rtfd.io



NNSmith

CI passing | pypi v0.0.1 | license Apache-2.0

NNSmith is a random DNN generator and a fuzzing infrastructure, primiarily designed for automatically validating deep-learning frameworks and compilers.

Support Table

Models	tvm	onnxruntime	tensorrt	tflite	xla	torchjit
ONNX	✅	✅	✅			
TensorFlow	🔨			✅	✅	
PyTorch	🔨	🔨				✅

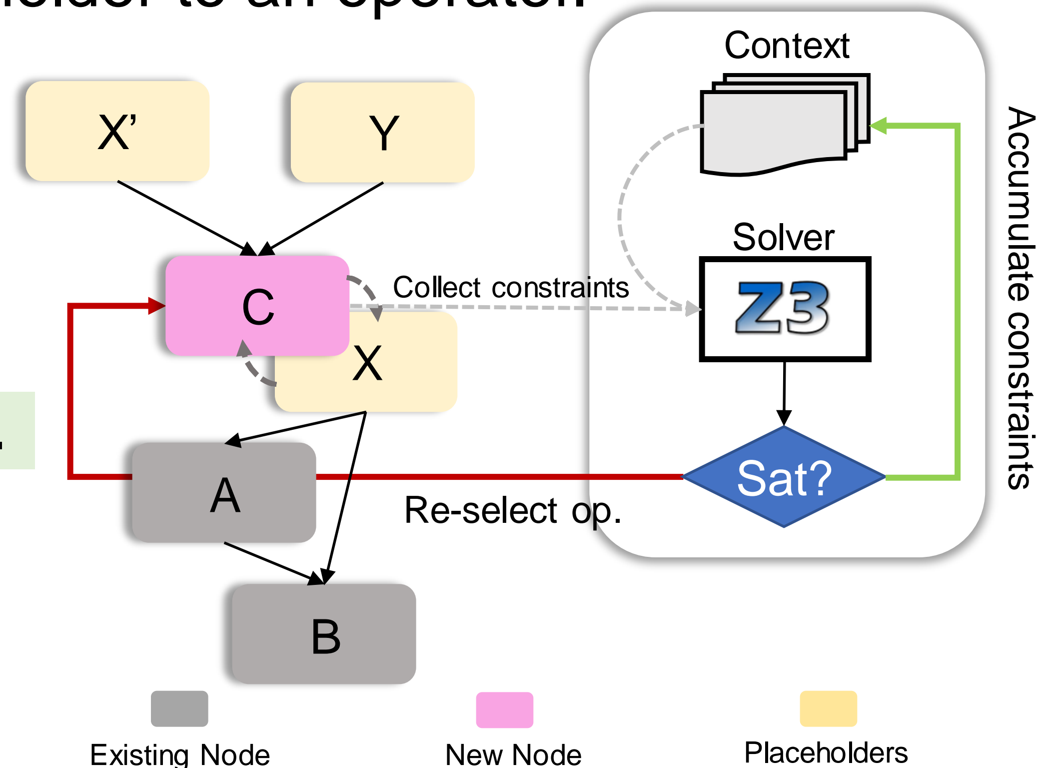
✅: Supported; ⚠: Experimental support; 🔨: Coming soon;

Bonus Slides (===Splitter===)

Incremental model construction

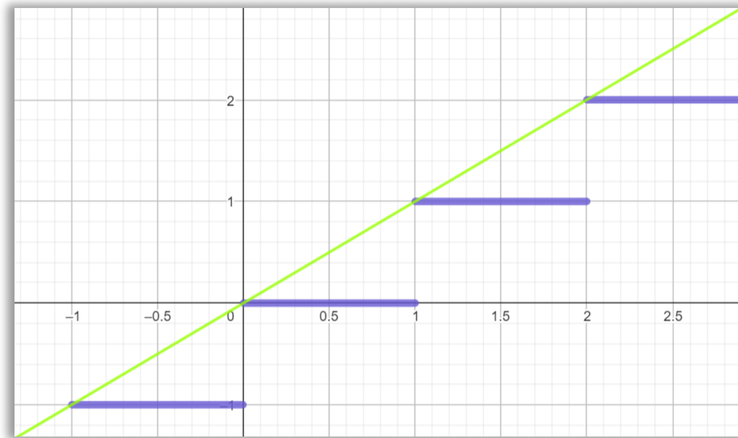
- **Building valid models:** a valid model —validity-preserving insert operator→ a *larger* valid model.
- **Forward insertion:** Insert a leaf operator.
- **Backward insertion:** Replace a placeholder to an operator.
  - Select an operator (C is an **add** op.)
  - Select compatible placeholder (X)
    - **add**'s output can be 4-dim float32 tensor.
  - If **sat**, replace X with the new node.
  - Construct new placeholders for X.

Backward insertion makes multi-input patterns.



Estimating gradients via "proxy derivative"

- Gradients cannot be backward propagated at some operators:
  - Zero-gradient regions: ReLU(X) where X < 0
  - Non-differentiable: Floor, Ceil, Integer operators, etc.
- To proceed, we *re-define* the derivatives of such operators (i.e., STE):
  - For each *non-diff* or *zero-grad* point x, let $\frac{dF(x)}{dx} = \pm\alpha$ (a constant)
  - The sign for α complies with overall trend at x.

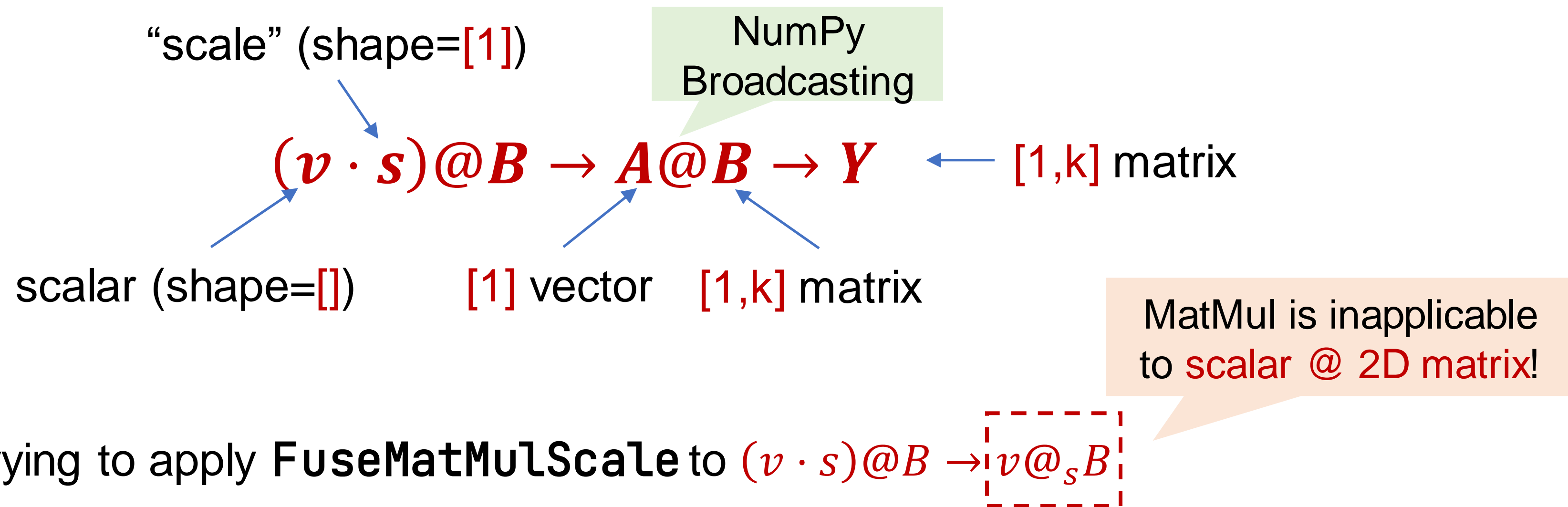


Let the proxy derivative of $Y = \lfloor X \rfloor$ to be $+\alpha$ (say $\alpha = 1$)

"Straight-Through Estimator": Bengio, Yoshua, et al. "Estimating or propagating gradients through stochastic neurons for conditional computation."

Exemplary crash bug

- `FuseMatMulScale` (ORT) fuses $(s \cdot A)@B$ into one kernel as if $A@_sB$:
- How ORT captures an optimizable pattern:
  - Find a graph pattern of $(\square_1 \cdot \square_2)@(\square_3)$
  - Either $\square_1$ or $\square_2$ is "scale", which is a single-element vector (shape = [1])
- Consider the bug-inducing case below (#10950):

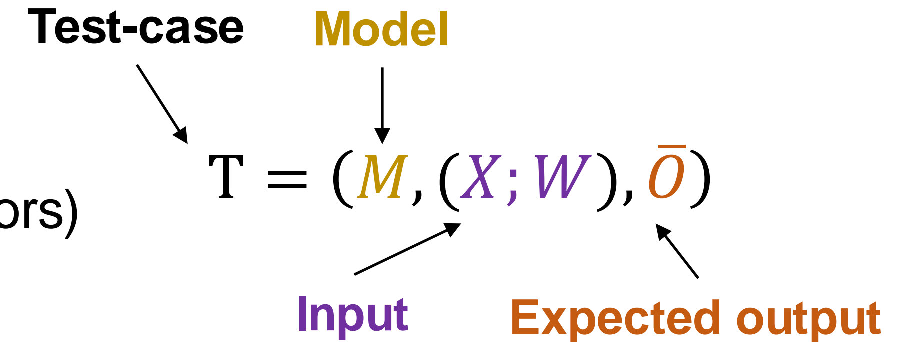


Exemplary semantic bug

- Pass: `SimplifyConsecutiveCast` in TVM
- Consecutive casting: `cast<int32>( cast<bool>(-1i64) )`
  - Interpreter: **1**
  - Optimized (tvm #13048): **-1**
- Consecutive casting should be “folded” when intermediate type has fewer bits.
  - e.g., bool as fewer bits than i64.

The test-case generation problem

- **A test-case in DL compiler**
  - A model M, *i.e.*, a "graph" of operators*
  - Model's computational inputs $X;W$ (input & weight tensors)
  - Expected outputs $\bar{O}$
- **Oracles**
  - Successful execution: Run{ $M(X;W)$ } → OK
  - Expected^ output: $M(X;W) = \bar{O}$



*Assume they are all (normalized to be) *functional*.
^Within tolerant floating-point errors.