Neural Network Preliminary
Tensor Computing

Weijie Zhao

Neural Network Preliminary
Tensor Computing

- Scalar
- Vector
- Matrix
- Tensor
  - Rank
  - Dimension

Weijie Zhao

- Scalar
- Vector
- Matrix
- Tensor
  - Rank
  - Dimension

Neural Network Preliminary
~~Tensor~~ Computing
Matrix

Weijie Zhao

- Matrix multiplication
- Non-linear activation
- Gradient descent

Neural Network Preliminary

~~Tensor~~ Computing

Matrix

- Scalar
- Vector
- Matrix
- Tensor
  - Rank
  - Dimension

Weijie Zhao

- Matrix multiplication
- Non-linear activation
- ~~Gradient descent~~ Graduate student descent

Neural Network Preliminary

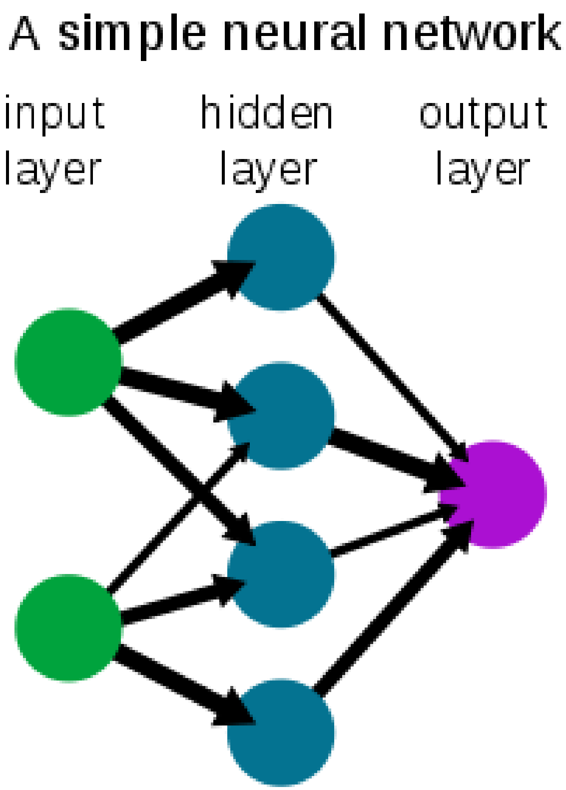
~~Tensor~~ Computing

Matrix

- Scalar
- Vector
- Matrix
- Tensor
  - Rank
  - Dimension

Weijie Zhao

Neural Networks



https://upload.wikimedia.org/wikipedia/commons/thumb/9/99/Neural_network_example.svg/330px-Neural_network_example.svg.png

Deep Learning Framework Implementation

- Knowing the things under the hood
- Deployment
- Deployment on emerging hardware

Deep Learning Language

- How to represent a deep neural network?

Deep Learning Language

- How to represent a deep neural network?
    - Abstraction

Deep Learning Language

- How to represent a deep neural network?
  - Abstraction
- How to implement/deploy the abstraction deep neural network?

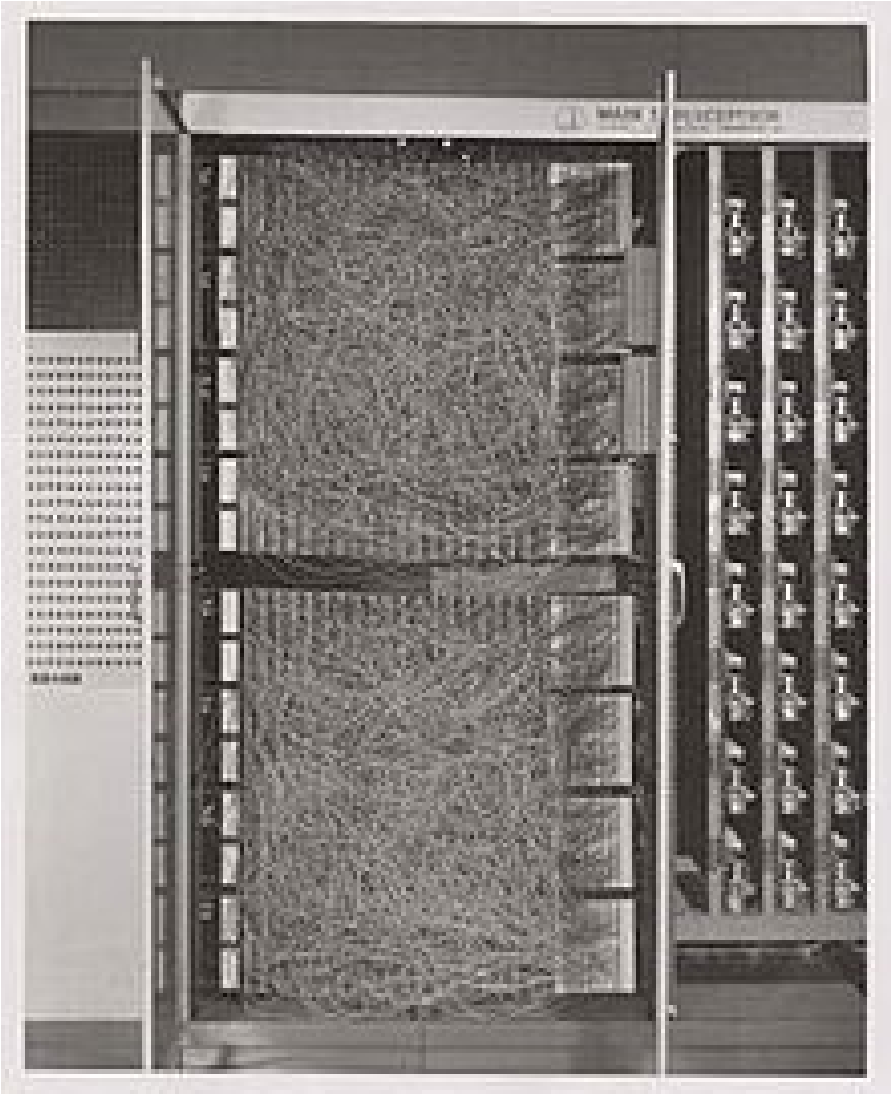
Deep Learning Language

- How to represent a deep neural network?
  - Abstraction
- How to implement/deploy the abstraction deep neural network?
  - Build tensor operations workflow
  - Implement high-performance low-level operations

Perceptron

- The perceptron was invented in 1943 by McCulloch and Pitts.
- The first implementation was a machine built in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$



https://upload.wikimedia.org/wikipedia/en/5/52/Mark_I_perceptron.jpeg

Perceptron

- Linear Layer
    - Matrix multiplication
    - Addition
- ReLU

Tensor Operations

- Element-wise add
- Element-wise plus
- Element-wise division
- Hadamard product
- Matrix multiplication
- Batched matrix multiplication
- More linear algebra operations…
- Collect, Scatter, Reduce…

Libraries

- Numpy
- Blas
- cuBlas
- cuSparse
- MKL
- TensorFlow
- PyTorch
- PaddlePaddle
- MXNet
- …

Lazy Evaluation and Code Generation

```
c = a + b
d = c * 2

for i = 1 to n do
    c[i] = a[i] + b[i]
for i = 1 to n do
    d[i] = c[i] * 2

for i = 1 to n do
    d[i] = (a[i] + b[i]) * 2
```

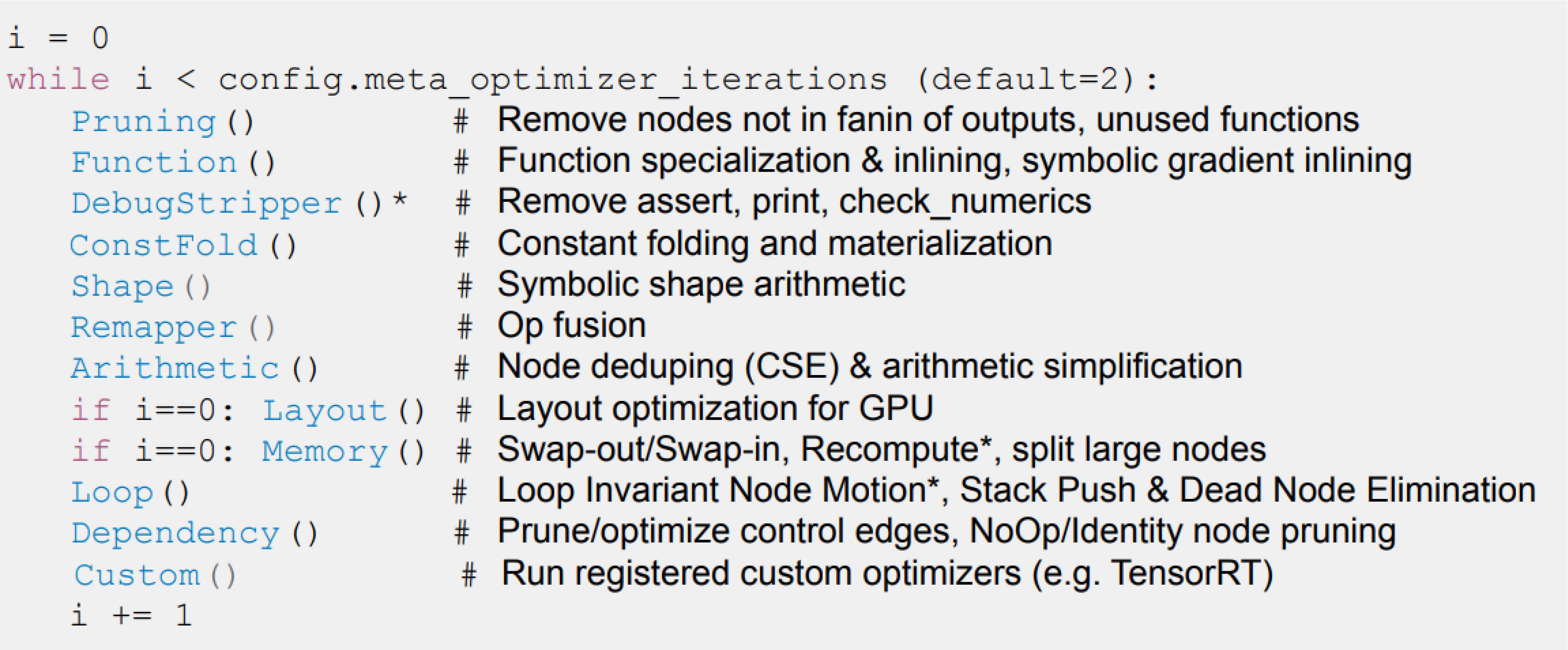
Optimizations

- Graph minimization and canonicalization
  - Constant Folding
  - Common subexpression elimination
  - Remove unnecessary operations
- Algebraic simplification and reassociation
- Copy propagation

Graph Optimizer

- Graph minimization and canonicalization
  - Constant Folding
  - Common subexpression elimination
  - Remove unnecessary operations
- Algebraic simplification and reassociation
- Copy propagation

Meta Optimizer



```
i = 0
while i < config.meta_optimizer_iterations (default=2):
    Pruning()              # Remove nodes not in fanin of outputs, unused functions
    Function()             # Function specialization & inlining, symbolic gradient inlining
    DebugStripper()*       # Remove assert, print, check_numerics
    ConstFold()            # Constant folding and materialization
    Shape()                # Symbolic shape arithmetic
    Remapper()             # Op fusion
    Arithmetic()           # Node deduping (CSE) & arithmetic simplification
    if i==0: Layout()      # Layout optimization for GPU
    if i==0: Memory()      # Swap-out/Swap-in, Recompute*, split large nodes
    Loop()                 # Loop Invariant Node Motion*, Stack Push & Dead Node Elimination
    Dependency()           # Prune/optimize control edges, NoOp/Identity node pruning
    Custom()               # Run registered custom optimizers (e.g. TensorRT)
    i += 1
```

https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf

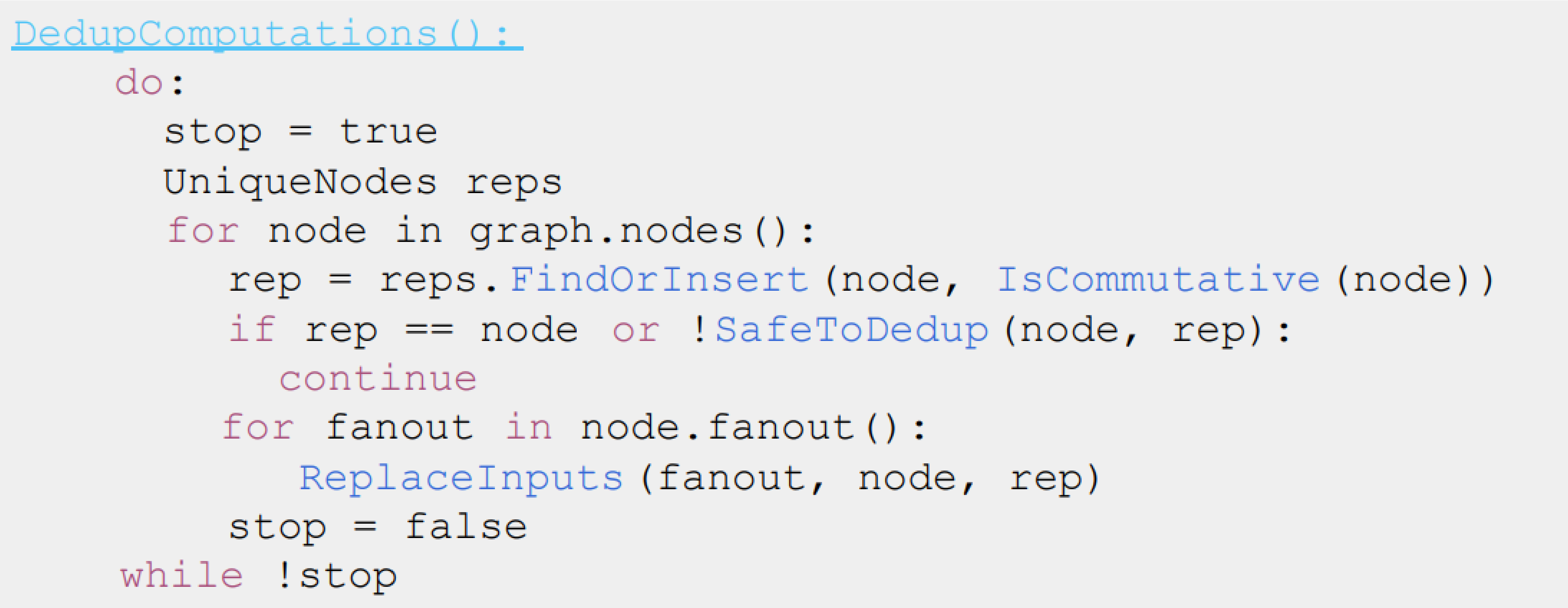
Constant Folding Optimizer

```
do:
   InferShapesStatically() # Fixed-point iteration with symbolic shapes
   graph_changed = MaterializeConstants() # grad broadcast, reduction dims
   q = NodesWithKnownInputs()
   while not q.empty():
     node = q.pop()
     graph_changed |= FoldGraph(node, &q) # Evaluate node on host
   graph_changed |= SimplifyGraph()
while graph_changed
```

Constant Folding Optimizer: SimplifyGraph()

- Removes trivial ops, e.g. identity Reshape, Transpose of 1-d tensors, Slice(x) = x, etc.
- Rewrites that enable further constant folding
- Arithmetic rewrites that rely on known shapes or inputs, e.g.
  - Constant push-down:
    - Add(c1, Add(x, c2)) => Add(x, c1 + c2)
    - ConvND(c1 * x, c2) => ConvND(x, c1 * c2)
  - Partial constfold:
    - AddN(c1, x, c2, y) => AddN(c1 + c2, x, y),
    - Concat([x, c1, c2, y]) = Concat([x, Concat([c1, c2]), y)
  - Operations with neutral & absorbing elements:
    - x * Ones(s) => Identity(x), if shape(x) == output_shape
    - x * Ones(s) => BroadcastTo(x, Shape(s)), if shape(s) == output_shape
    - Same for x + Zeros(s) , x / Ones(s), x * Zeros(s) etc.
    - Zeros(s) - y => Neg(y), if shape(y) == output_shape
    - Ones(s) / y => Recip(y) if shape(y) == output_shape

Arithmetic Optimizer



```
DedupComputations():
    do:
      stop = true
      UniqueNodes reps
      for node in graph.nodes():
        rep = reps.FindOrInsert(node, IsCommutative(node))
        if rep == node or !SafeToDedup(node, rep):
          continue
        for fanout in node.fanout():
            ReplaceInputs(fanout, node, rep)
        stop = false
    while !stop
```

Arithmetic Optimizer

- Arithmetic simplifications
  - Flattening: a+b+c+d => AddN(a, b, c, d)
  - Hoisting: AddN(x * a, b * x, x * c) => x * AddN(a+b+c)
  - Simplification to reduce number of nodes:
    - Numeric: x+x+x => 3*x
    - Logic: !(x > y) => x <= y
- Broadcast minimization
  - Example: (matrix1 + scalar1) + (matrix2 + scalar2) => (matrix1 + matrix2) + (scalar1 + scalar2)
- Better use of intrinsics
  - Matmul(Transpose(x), y) => Matmul(x, y, transpose_x=True)
- Remove redundant ops or op pairs
  - Transpose(Transpose(x, perm), inverse_perm)
  - BitCast(BitCast(x, dtype1), dtype2) => BitCast(x, dtype2)
  - Pairs of elementwise involutions f(f(x)) => x (Neg, Conj, Reciprocal, LogicalNot)
  - Repeated Idempotent ops f(f(x)) => f(x) (DeepCopy, Identity, CheckNumerics...)
- Hoist chains of unary ops at Concat/Split/SplitV
  - Concat([Exp(Cos(x)), Exp(Cos(y)), Exp(Cos(z))]) => Exp(Cos(Concat([x, y, z])))
  - [Exp(Cos(y)) for y in Split(x)] => Split(Exp(Cos(x), num_splits)

Tensor

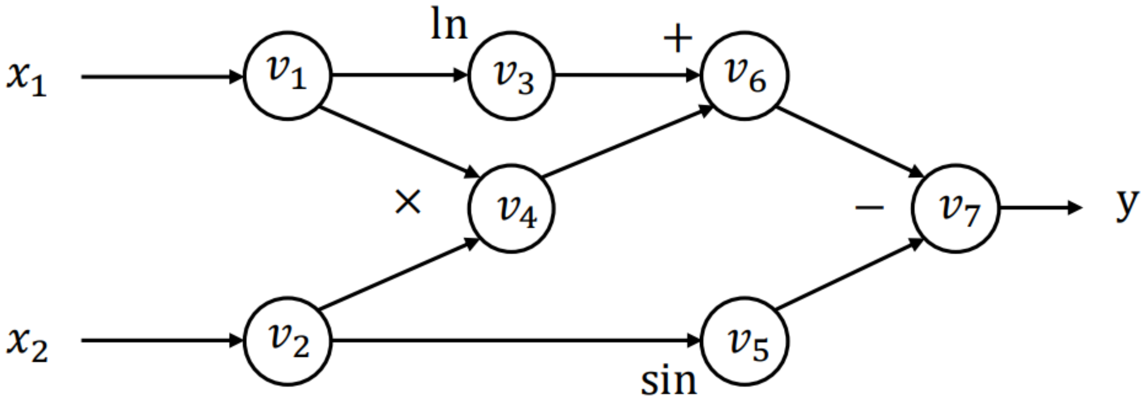
- Dense
  - Column major
  - Row major
  - Stride
- Sparse
  - Compressed representation
  - Set intersection

Differentiation

- Numerical differentiation
- Symbolic differentiation
  - Chain rules
- Forward mode auto differentiation
- Reverse mode auto differentiation

Computational Graph

$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$



Forward evaluation trace

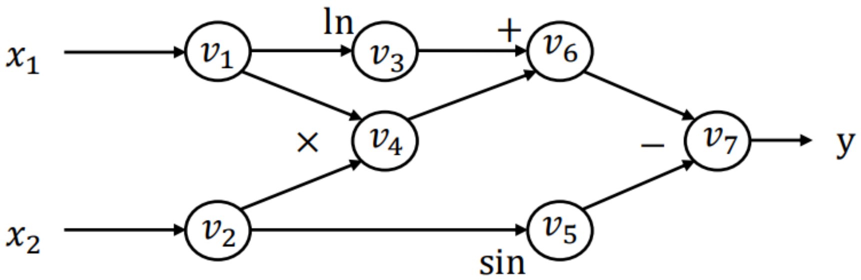
$$
\begin{aligned}
v_1 &= x_1 = 2\\
v_2 &= x_2 = 5\\
v_3 &= \ln v_1 = \ln 2 = 0.693\\
v_4 &= v_1 \times v_2 = 10\\
v_5 &= \sin v_2 = \sin 5 = -0.959\\
v_6 &= v_3 + v_4 = 10.693\\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652\\
y &= v_7 = 11.652
\end{aligned}
$$

Each node represent an (intermediate) value in the computation. Edges present input output relations.

https://dlsyscourse.org/slides/4-automatic-differentiation.pdf

Forward Mode Auto Differentiation

$y = f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin x_2$



Define $\dot{v_i} = \frac{\partial v_i}{\partial x_1}$

We can then compute the $\dot{v_i}$ iteratively in the forward topological order of the computational graph

Forward evaluation trace

$v_1 = x_1 = 2$
$v_2 = x_2 = 5$
$v_3 = \ln v_1 = \ln 2 = 0.693$
$v_4 = v_1 \times v_2 = 10$
$v_5 = \sin v_2 = \sin 5 = -0.959$
$v_6 = v_3 + v_4 = 10.693$
$v_7 = v_6 - v_5 = 10.693 + 0.959 = 11.652$
$y = v_7 = 11.652$

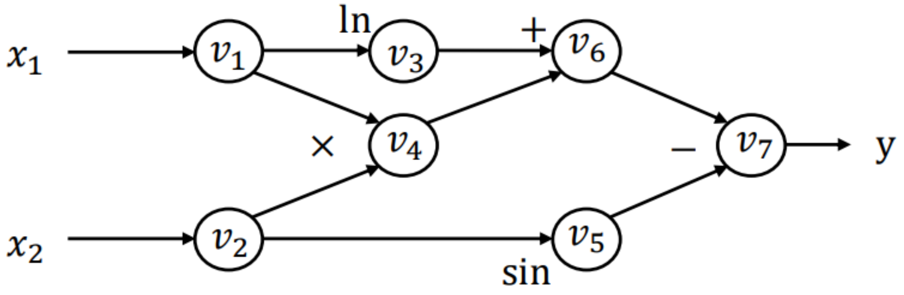
Forward AD trace

$\dot{v_1} = 1$
$\dot{v_2} = 0$
$\dot{v_3} = \dot{v_1}/v_1 = 0.5$
$\dot{v_4} = \dot{v_1}v_2 + \dot{v_2}v_1 = 1 \times 5 + 0 \times 2 = 5$
$\dot{v_5} = \dot{v_2}\cos v_2 = 0 \times \cos 5 = 0$
$\dot{v_6} = \dot{v_3} + \dot{v_4} = 0.5 + 5 = 5.5$
$\dot{v_7} = \dot{v_6} - \dot{v_5} = 5.5 - 0 = 5.5$

Now we have $\frac{\partial y}{\partial x_1} = \dot{v_7} = 5.5$

Reverse Mode Auto Differentiation

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$



Forward evaluation trace

$$
\begin{aligned}
v_1 &= x_1 = 2 \\
v_2 &= x_2 = 5 \\
v_3 &= \ln v_1 = \ln 2 = 0.693 \\
v_4 &= v_1 \times v_2 = 10 \\
v_5 &= \sin v_2 = \sin 5 = -0.959 \\
v_6 &= v_3 + v_4 = 10.693 \\
v_7 &= v_6 - v_5 = 10.693 + 0.959 = 11.652 \\
y &= v_7 = 11.652
\end{aligned}
$$

Define adjoint $\bar{v}_i = \frac{\partial y}{\partial v_i}$

We can then compute the $\bar{v}_i$ iteratively in the **reverse** topological order of the computational graph

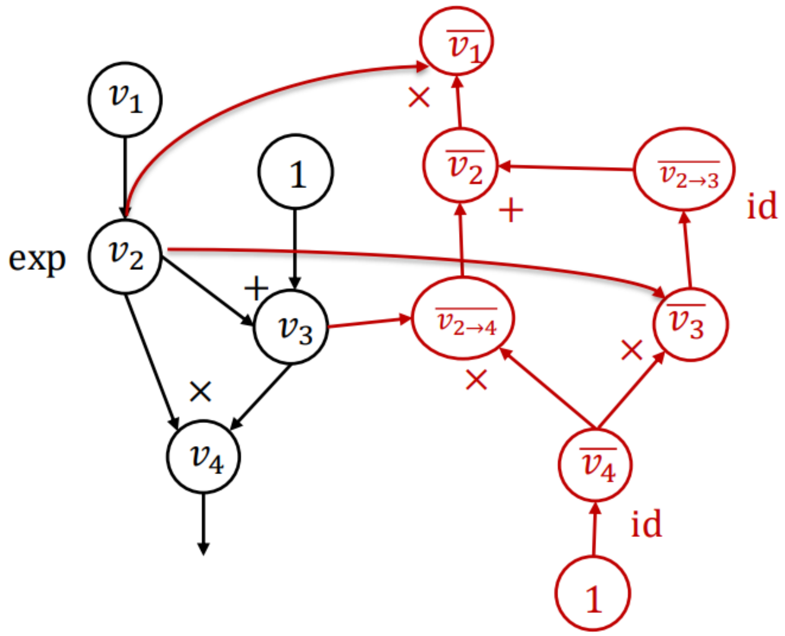
Reverse AD evaluation trace

$$
\begin{aligned}
\overline{v_7} &= \frac{\partial y}{\partial v_7} = 1 \\
\overline{v_6} &= \overline{v_7} \frac{\partial v_7}{\partial v_6} = \overline{v_7} \times 1 = 1 \\
\overline{v_5} &= \overline{v_7} \frac{\partial v_7}{\partial v_5} = \overline{v_7} \times (-1) = -1 \\
\overline{v_4} &= \overline{v_6} \frac{\partial v_6}{\partial v_4} = \overline{v_6} \times 1 = 1 \\
\overline{v_3} &= \overline{v_6} \frac{\partial v_6}{\partial v_3} = \overline{v_6} \times 1 = 1 \\
\overline{v_2} &= \overline{v_5} \frac{\partial v_5}{\partial v_2} + \overline{v_4} \frac{\partial v_4}{\partial v_2} = \overline{v_5} \times \cos v_2 + \overline{v_4} \times v_1 = -0.284 + 2 = 1.716 \\
\overline{v_1} &= \overline{v_4} \frac{\partial v_4}{\partial v_1} + \overline{v_3} \frac{\partial v_3}{\partial v_1} = \overline{v_4} \times v_2 + \overline{v_3} \frac{1}{v_1} = 5 + \frac{1}{2} = 5.5
\end{aligned}
$$

Reverse Mode Auto Differentiation

```
def gradient(out):
  node_to_grad = {out:  [1]}
  for $i$ in reverse_topo_order(out):
    $\overline{v_i} = \sum_j \overline{v_{i\to j}}$ = sum(node_to_grad[$i$])
    for $k \in inputs(i)$:
        compute $\overline{v_{k\to i}} = \overline{v_i} \frac{\partial v_i}{\partial v_k}$
        append $\overline{v_{k\to i}}$ to node_to_grad[$k$]
  return adjoint of input $\overline{v_{input}}$
```

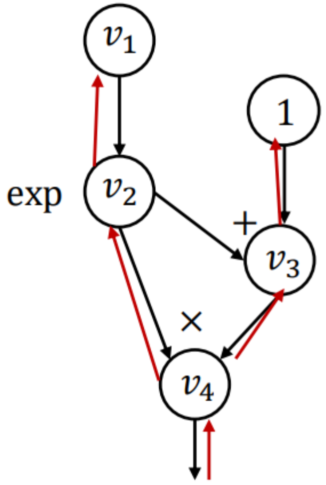
```
$i = 2$
node_to_grad: {
  1: [$\overline{v_1}$]
  2: [$\overline{v_{2\to4}}$, $\overline{v_{2\to3}}$]
  3: [$\overline{v_3}$]
  4: [$\overline{v_4}$]
}
```



NOTE: id is identity function

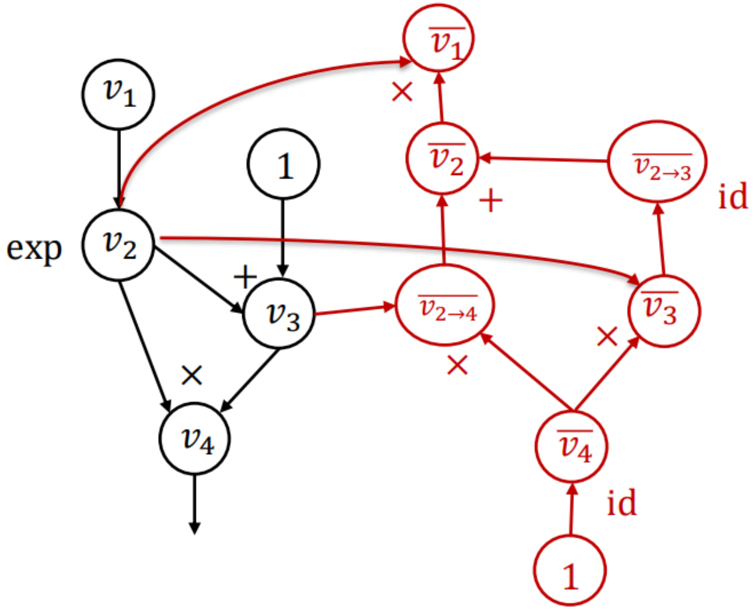
Backprop V.S. Reverse Mode AD

Backprop



- Run backward operations the same forward graph
- Used in first generation deep learning frameworks (caffe, cuda-convnet)

Reverse mode AD by extending computational graph



- Construct separate graph nodes for adjoints
- Used by modern deep learning frameworks