LLM Sys

Deep Learning Framework and Auto Differentiation

Lei Li

Language Technologies Institute | Carnegie Mellon University School of Computer Science

Recap of GPU Acceleration

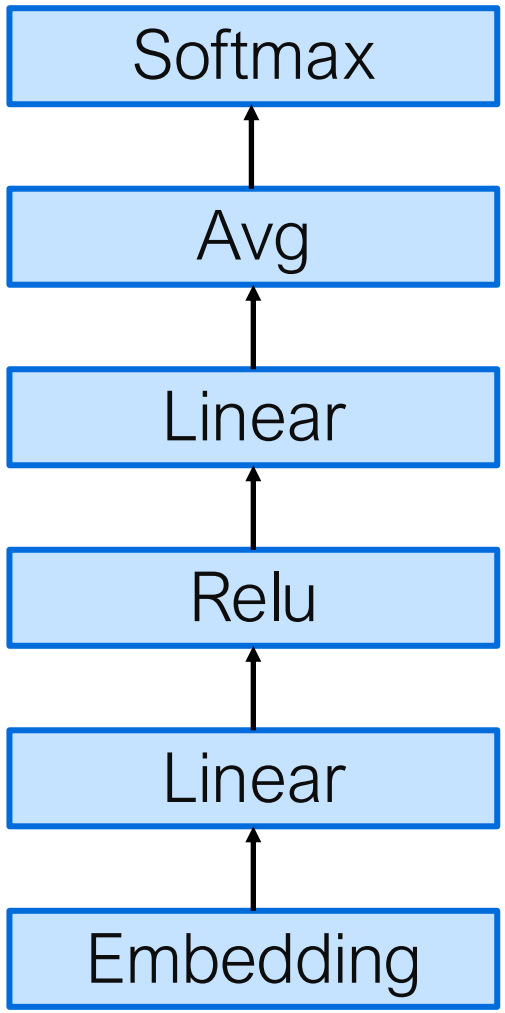
- Tiling for efficient matrix computation
- Coalesced memory access
- Sparse matrix representation and multiplication
- cuBLAS

Today's Topic

- Learning algorithm for Neural Network
- Computation Graph
- Auto Differentiation
- Put Together: Implementing a Deep Learning Framework

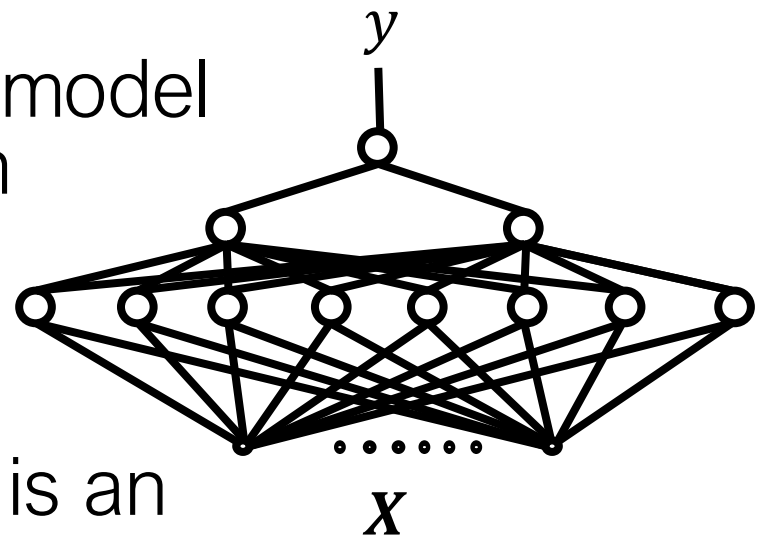
A Simple Feedforward Neural Network

- Neural network layers
  - Embedding (lookup table)
  - Linear
  - Relu
  - Average pooling
  - Softmax



The Learning Problem

- Given a training set of input-output pairs $D = \{(x_n, y_n)\}_{n=1}^{N}$
  - x_n and y_n may both be vectors
- To find the model parameters such that the model produces the most accurate output for each training input
  - Or a close approximation of it
- Learning the parameter of a neural network is an instance!
  - The network architecture is given



Training Loss for Classification

- (x_n, y_n) are data and label pairs for training
- Cross entropy

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^{N} -\log f(x_n)_{y_n}$$

- Pytorch CrossEntropyLoss is implemented as
  - Negative Likelihood on logits (instead of log of softmax)

Today's focus (using PyTorch Example)

```
loss = nn.CrossEntropyLoss()

output = loss(input_logits, target_labels)

grads = output.backward()
```

how is backward implemented?
how does it work on any network?

Generic Iterative Learning Algorithm

- Consider a generic function minimization problem, where x is unknown variable

$$\min_{x} f(x) \quad \text{where } f: \mathbb{R}^d \rightarrow \mathbb{R}$$

- Iterative update algorithm

$$x_{t+1} \leftarrow x_t + \Delta$$

- so that $f(x_{t+1}) \ll f(x_t)$
- How to find Δ

Gradient Descent

- $f(x_t + \Delta x) \approx f(x_t) + \Delta x^T \nabla f|_{x_t}$
- To make $\Delta x^T \nabla f|_{x_t}$ smallest
  - $\Rightarrow \Delta x$ in the opposite direction of $\nabla f|_{x_t}$ *i.e.* $\Delta x = -\nabla f|_{x_t}$
- Update rule: $x_{t+1} = x_t - \eta \nabla f|_{x_t}$
- η is a hyper-parameter to control the learning rate

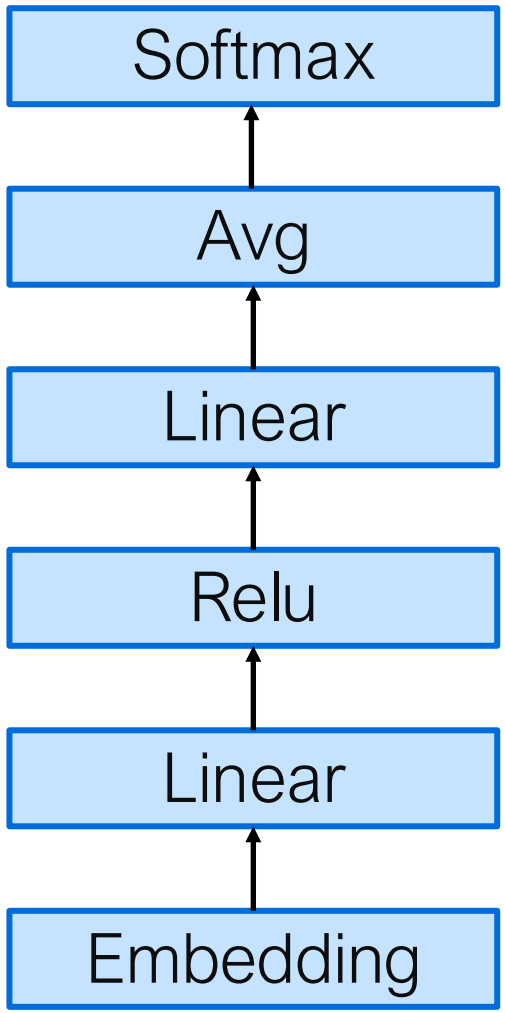
(Stochastic) Gradient Descent Algorithm

set learning rate eta.

1. set initial parameter $\theta \leftarrow \theta_0$
2. for epoch = 1 to maxEpoch or until converg:
3. for each batch in the data:
4. total_g = 0
5. for each data (x, y) in data batch:
6. compute error err(f(x; θ) - y)
7. compute gradient $g = \frac{\partial \mathrm{err}(\theta)}{\partial \theta}$
8. total_g += g
9. update θ = θ - eta * total_g / N

How to compute the gradient for every parameter in an "arbitrary network"?

- Goal: compute $\frac{\partial l}{\partial w_i}$ for every parameter
- Forward computation
- Backpropogation



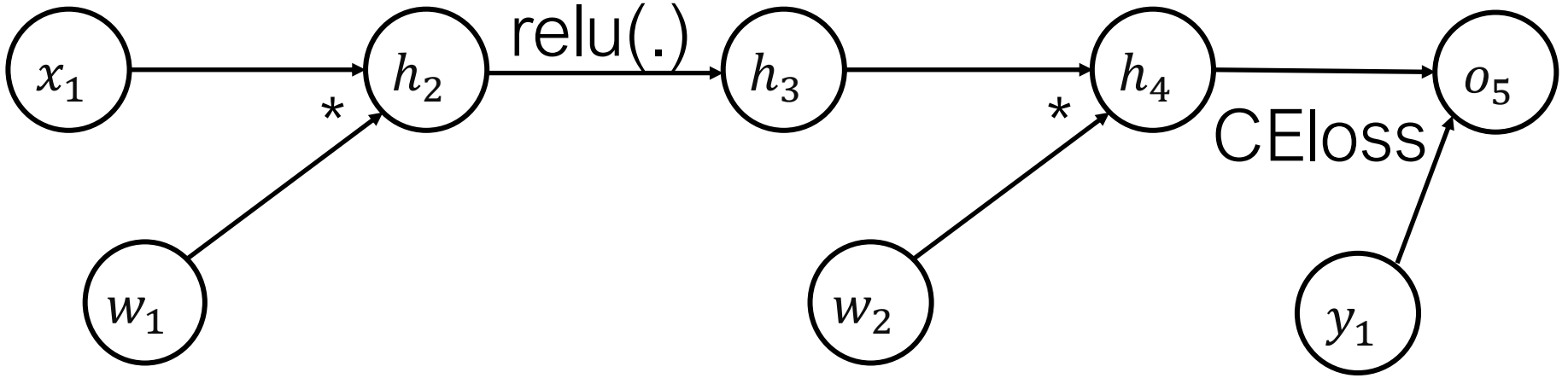
"It is a good movie"

Today's Topic

- Learning algorithm for Neural Network
- Computation Graph
- Auto Differentiation
- Put Together: Implementing a Deep Learning Framework

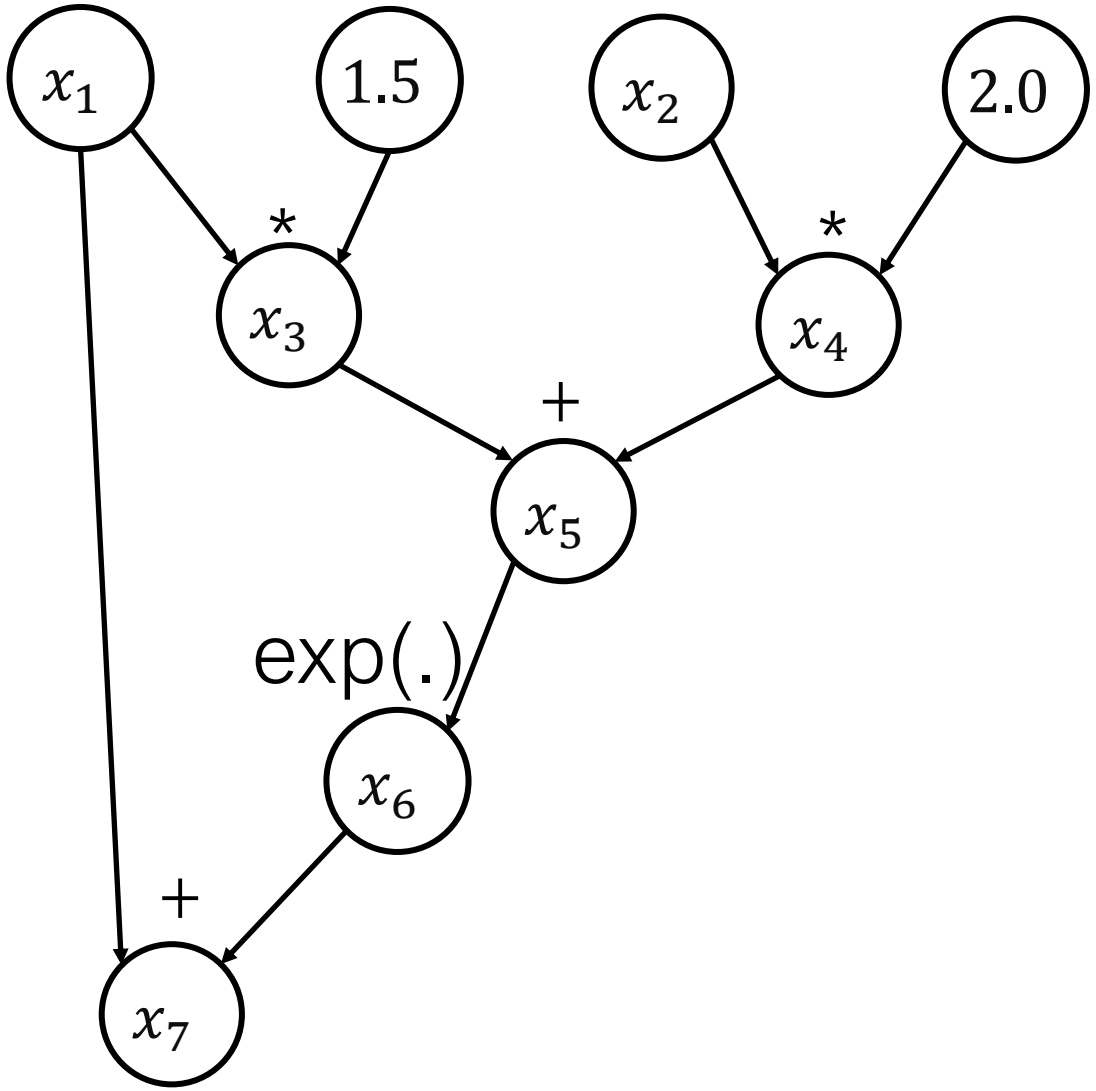
Computation Graph

- Calculations on a neural network can be defined using computation graph
- Each node denotes a variable or an operation
- Directed edges to connect nodes, indicating the input values for operations.



x1= 3, x2=0.5

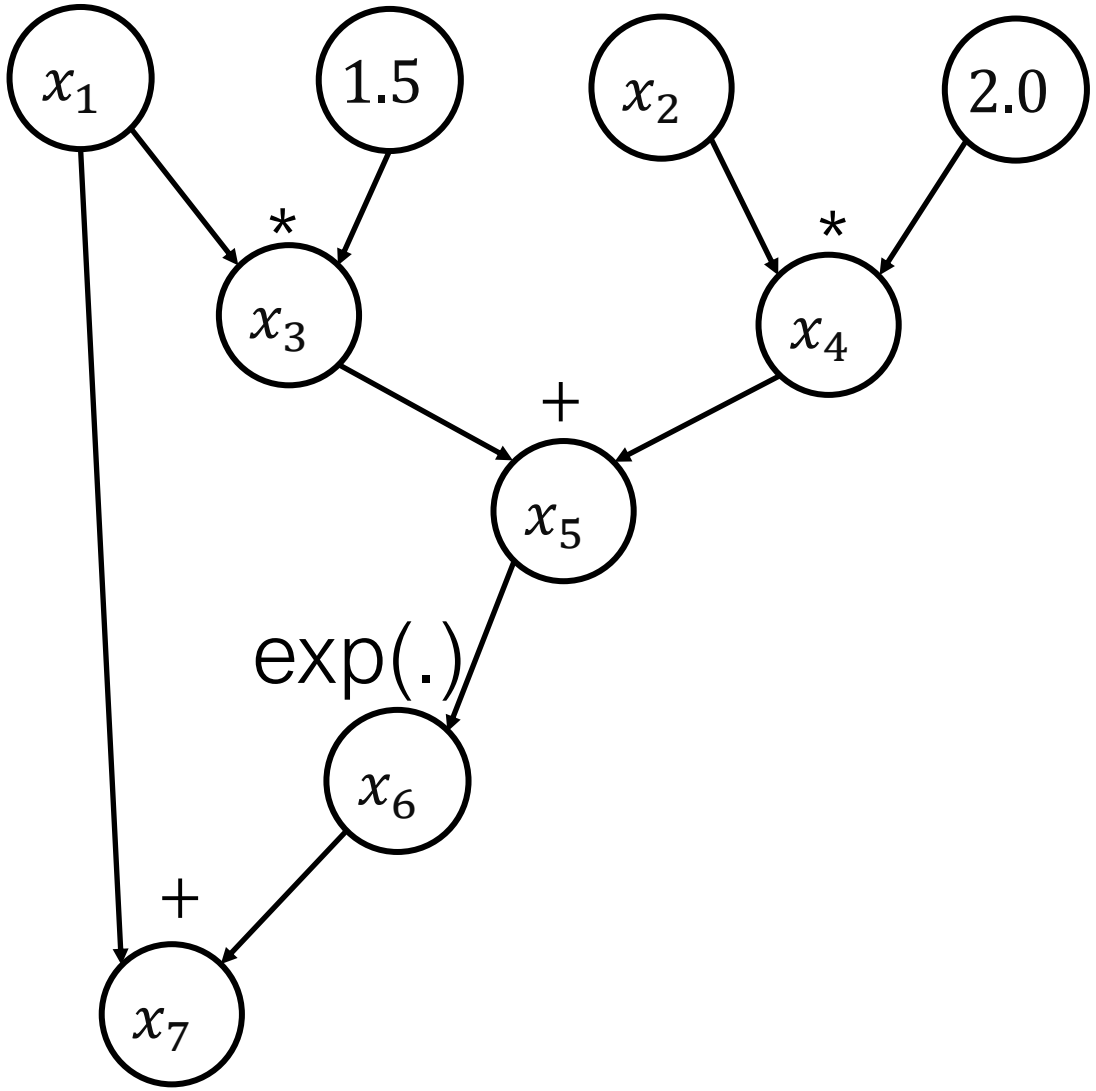
f = x1 + exp(1.5 * x1 + 2.0 * x2)



To perform Computation:

1. Topological sorting of all nodes
2. Calculate the value for each node given its input

Topological Sort



- Put all nodes into un-processed queue.
- Repeatedly, find a node without incoming edges from un-processed nodes
  - evaluate its value based on operation
  - remove the node from the queue and add it to processed queue

Today's Topic

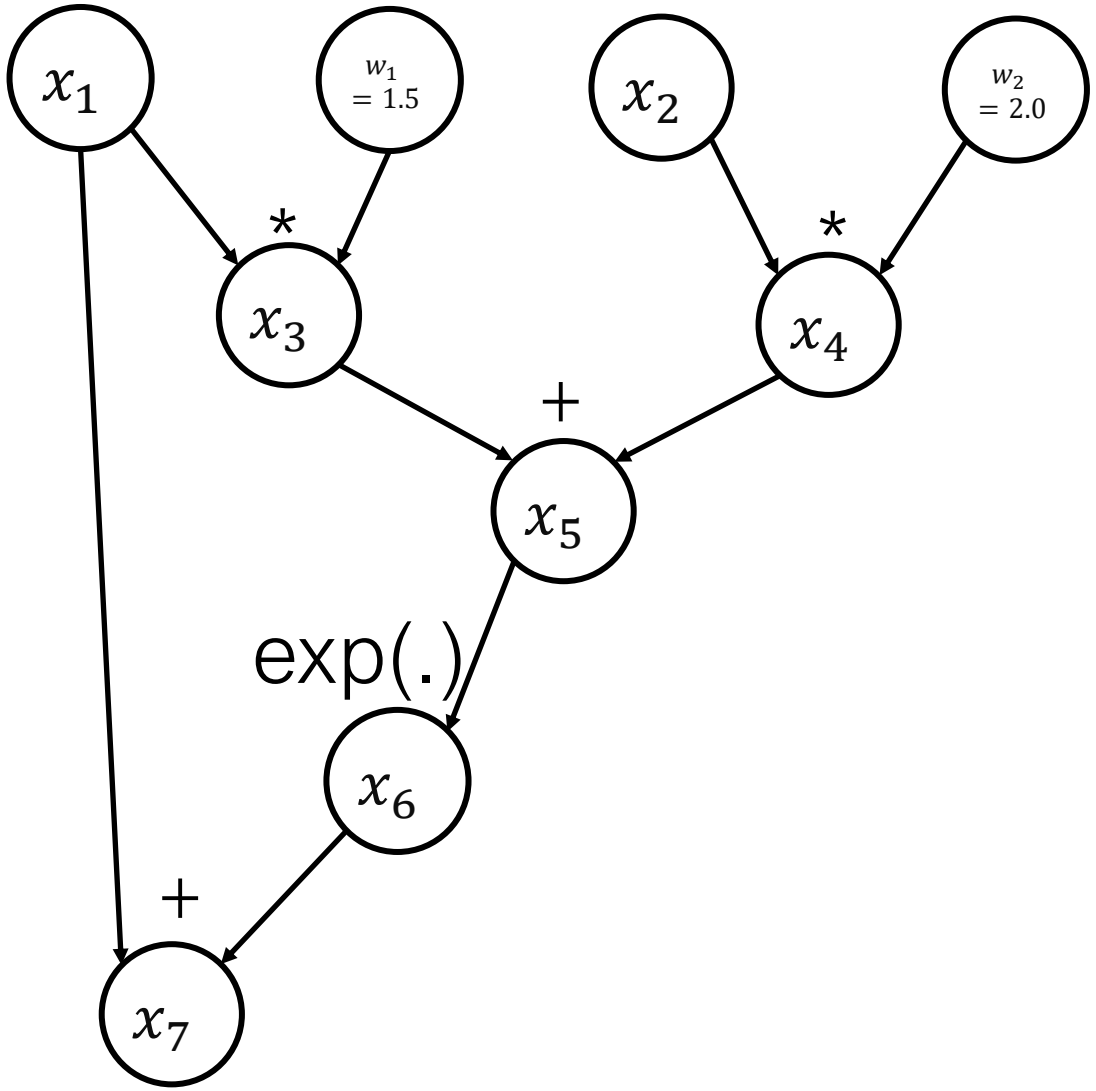
- Learning algorithm for Neural Network
- Computation Graph
- Auto Differentiation
- Put Together: Implementing a Deep Learning Framework

Gradient Calculation

- To learn a neural network, we need gradient of loss function w.r.t. parameters.
- Parameters are also variables, and represented as nodes in the computation graph.
- Chain rule => backpropogation

$$\frac{dy(z)}{dx} = \frac{dy(z)}{dz} \cdot \frac{dz}{dx}$$

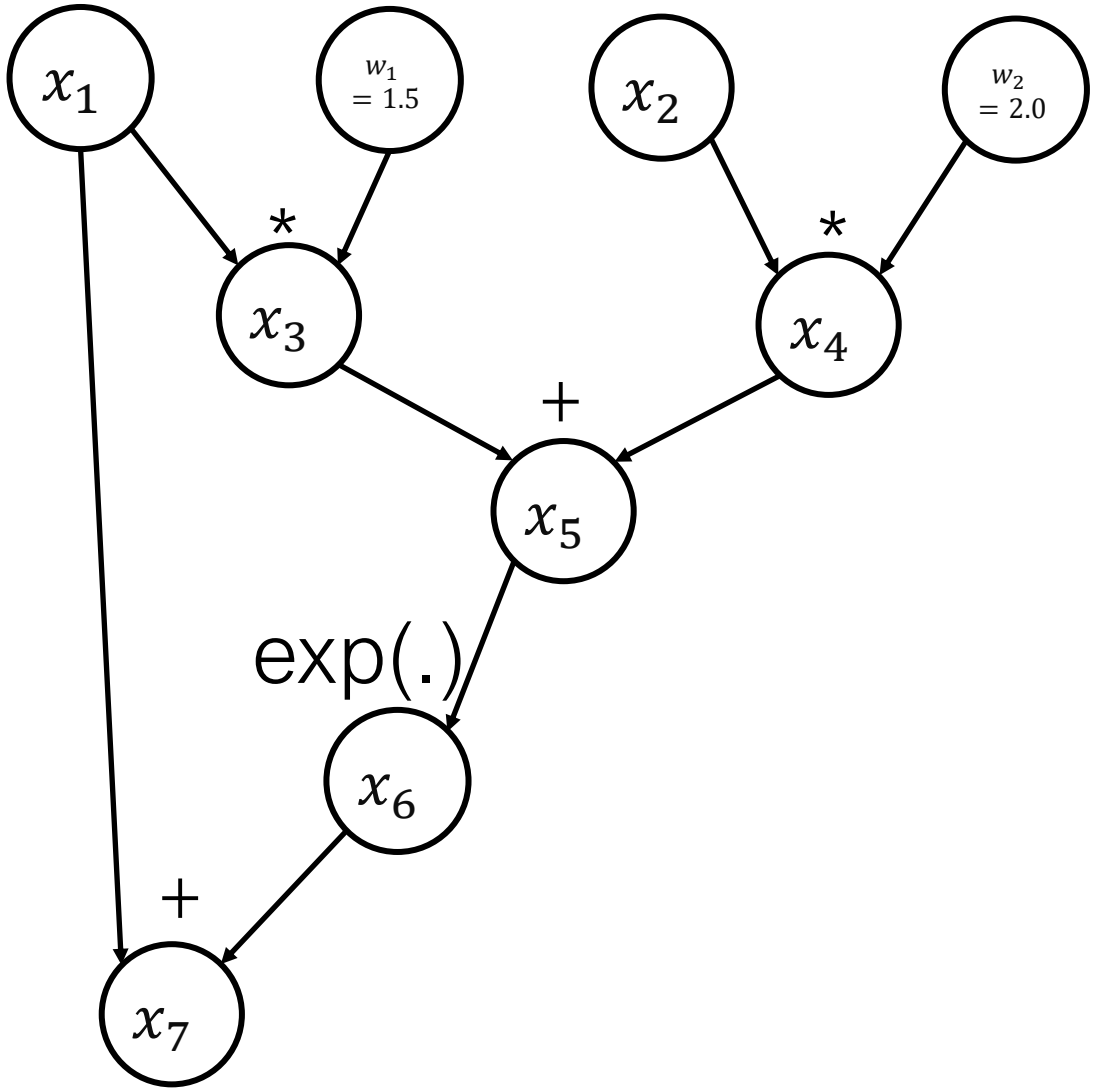
x1= 3, x2=0.5
y=x1 + exp(1.5 * x1 + 2.0 * x2)



Computing the derivatives $\frac{\partial y}{\partial x_i}$

Define $\overline{x_i} = \frac{\partial y}{\partial x_i}$

x1= 3, x2=0.5
y=x1 + exp(1.5 * x1 + 2.0 * x2)



Computing the derivatives $\frac{\partial y}{\partial x_i}$

Define $\overline{x_i} = \frac{\partial y}{\partial x_i}$

$$\overline{x_7} = 1$$

$$\overline{x_6} = 1$$

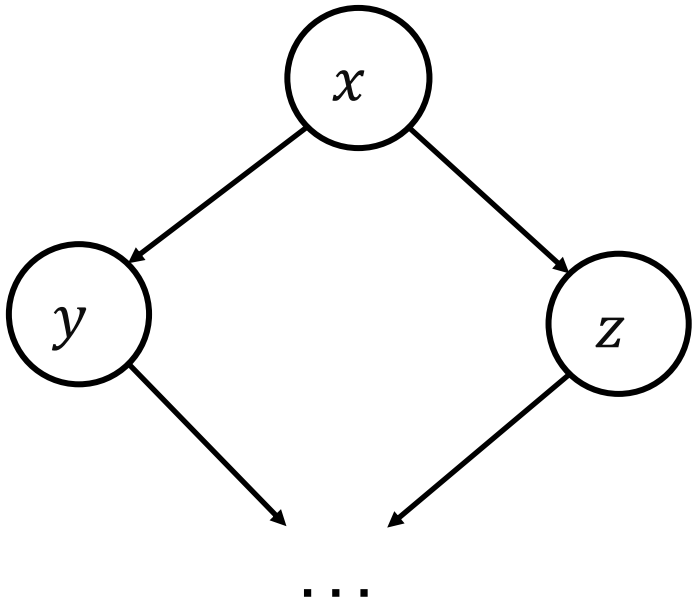
$$\overline{x_5} = \frac{\partial y}{\partial x_6} \cdot \frac{\partial x_6}{\partial x_5} = \overline{x_6} \cdot \exp(x_5)$$

$$\overline{x_4} = \frac{\partial y}{\partial x_5} \cdot \frac{\partial x_5}{\partial x_4} = \overline{x_5}$$

$$\overline{x_3} = \frac{\partial y}{\partial x_5} \cdot \frac{\partial x_5}{\partial x_3} = \overline{x_5}$$

$$\overline{w_2} = \frac{\partial y}{\partial x_4} \cdot \frac{\partial x_4}{\partial w_2} = \overline{x_4} \cdot x_2$$

Node with multiple outgoing edges



$$\bar{x} = \bar{y} \cdot \frac{\partial y}{\partial x} + \bar{z} \cdot \frac{\partial z}{\partial x}$$

Partial derivatives for Vectors

Jacobian

$$J = \frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{pmatrix}$$

row: keep y index, iter x index

col: keep x index, iter y index

Vector Jacobian Product

$$J = \frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{pmatrix}$$

- computing the partial derivative for each node (vector)

$$\bar{x} = J^T \bar{y}$$

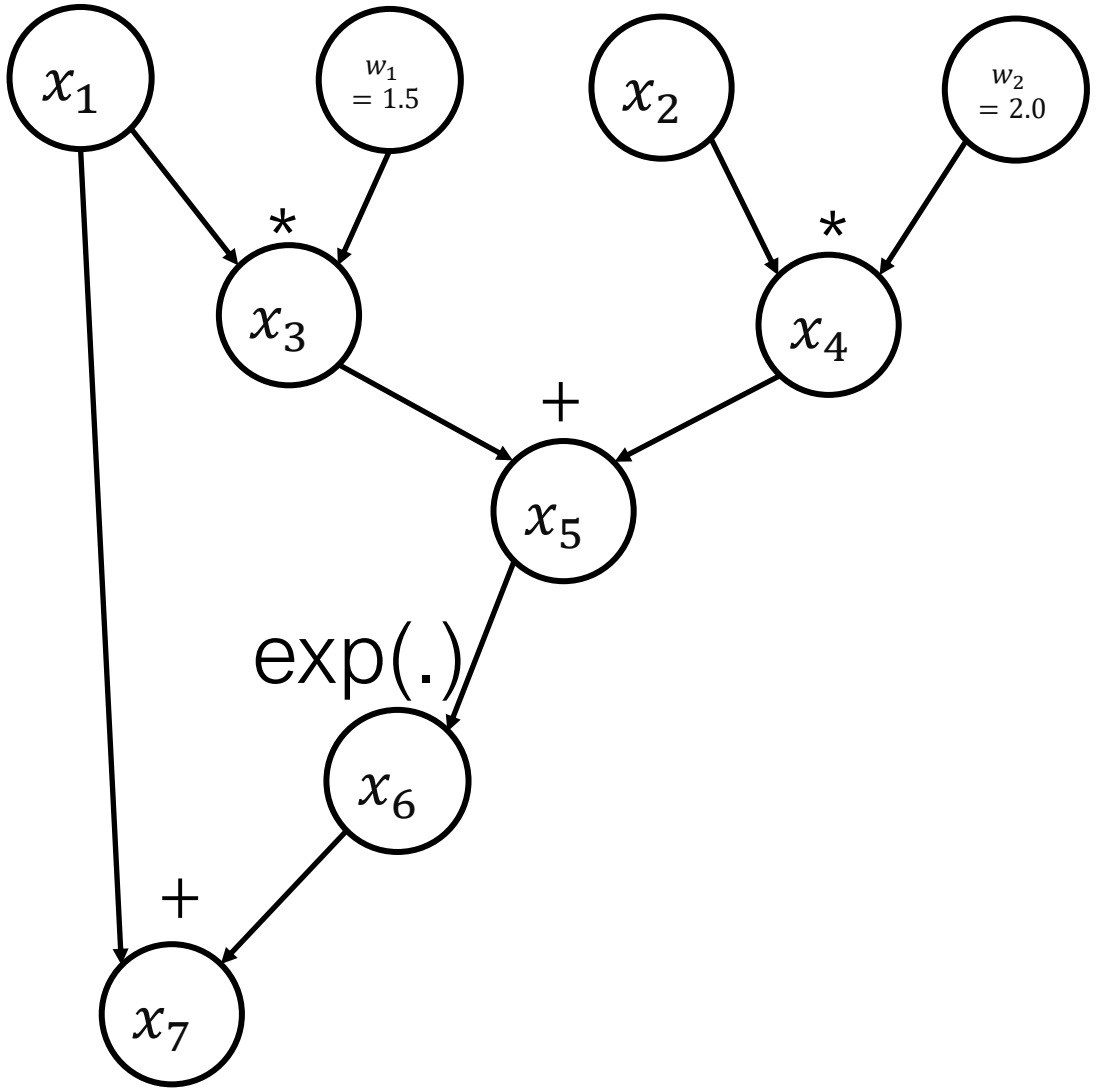
Example

$$y = Wx$$
$$\bar{x} = W^T \bar{y}$$

Auto Differentiation

- Instead of explicitly computing the derivatives (gradients) for each data sample following the backward direction
- Construct a computation graph for gradient calculation for every node
- Applicable to any input data (and output=loss)

x1= 3, x2=0.5
y=x1 + exp(1.5 * x1 + 2.0 * x2)

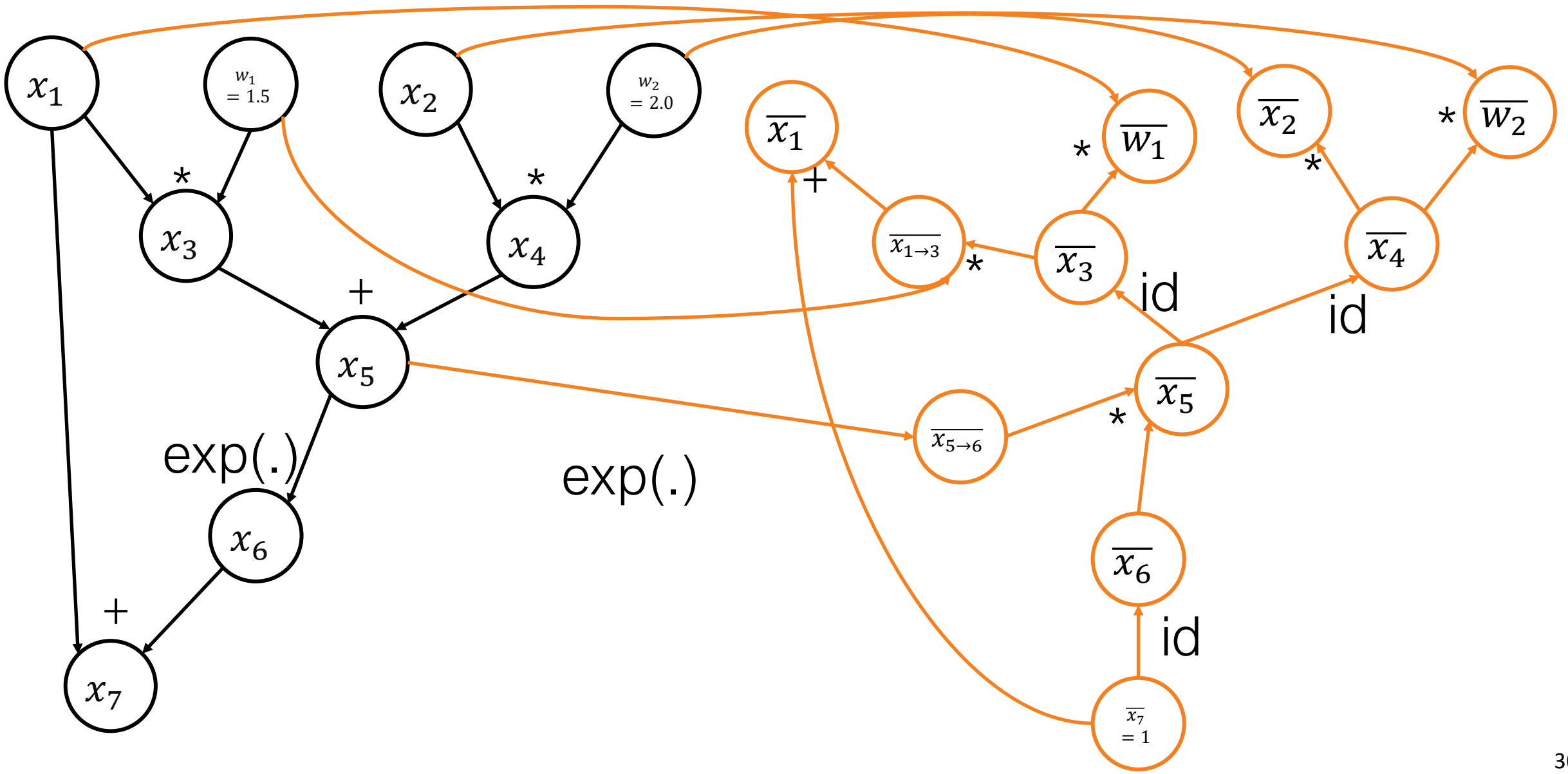


Computing the derivatives $\frac{\partial y}{\partial x_i}$

Define $\overline{x_i} = \frac{\partial y}{\partial x_i}$

x1= 3, x2=0.5

y=x1 + exp(1.5 * x1 + 2.0 * x2)



Implementing Backward Pass (important for HW2)

```
def backward_pass(g, end_node):
    """Backpropagation.

    Traverse computation graph backwards in topological order from the end node.
    For each node, compute local gradient contribution and accumulate.
    """
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            # Lookup vector-Jacobian product (gradient) function for this
            # function/argument.
            vjp = primitive_vjps[fun][argnum]

            # Compute vector-Jacobian product (gradient) contribution due to
            # parent node's use in this function.
            parent_grad = vjp(outgrad, value, *args, **kwargs)

            # Save vector-Jacobian product (gradient) for upstream nodes.
            # Sum contributions with all others also using parent's output.
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad
```

```
def add_outgrads(prev_g, g):
    """Add gradient contributions together."""
    if prev_g is None:
        return g
    return prev_g + g
```

Build the AutoDiff Graph

```
def make_vjp(fun, x):
    """Make function for vector-Jacobian product.

    Args:
      fun: single-arg function. Jacobian derived from this.
      x: ndarray. Point to differentiate about.

    Returns:
      vjp: single-arg function. vector -> vector-Jacobian[fun, x] proc
      end_value: end_value = fun(start_node)

    """
    start_node = Node.new_root()
    end_value, end_node = trace(start_node, fun, x)
    if end_node is None:
        def vjp(g): return np.zeros_like(x)
    else:
        def vjp(g): return backward_pass(g, end_node)
    return vjp, end_value
```

```
def grad(fun, argnum=0):
    """Constructs gradient function.

    Given a function fun(x), returns a function fun'(x) that returns the
    gradient of fun(x) wrt x.

    Args:
      fun: single-argument function. ndarray -> ndarray.
      argnum: integer. Index of argument to take derivative wrt.

    Returns:
      gradfun: function that takes same args as fun(), but returns the gradient
        wrt to fun()'s argnum-th argument.
    """
    def gradfun(*args, **kwargs):
        # Replace args[argnum] with x. Define a single-argument function to
        # compute derivative wrt.
        unary_fun = lambda x: fun(*subval(args, argnum, x), **kwargs)

        # Construct vector-Jacobian product
        vjp, ans = make_vjp(unary_fun, args[argnum])
        return vjp(np.ones_like(ans))
    return gradfun
```

Use AutoGrad

```
# Define training objective
def objective(params, iter):
    idx = batch_indices(iter)
    return -log_posterior(params, train_images[idx], train_labels[idx], L2_reg)

# Get gradient of objective using autograd.
objective_grad = grad(objective)
```

```
def neural_net_predict(params, inputs):
    """Implements a deep neural network for classification.
       params is a list of (weights, bias) tuples.
       inputs is an (N x D) matrix.
       returns normalized class log-probabilities."""
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs - logsumexp(outputs, axis=1, keepdims=True)

def log_posterior(params, inputs, targets, L2_reg):
    log_prior = -L2_reg * l2_norm(params)
    log_lik = np.sum(neural_net_predict(params, inputs) * targets)
    return log_prior + log_lik
```

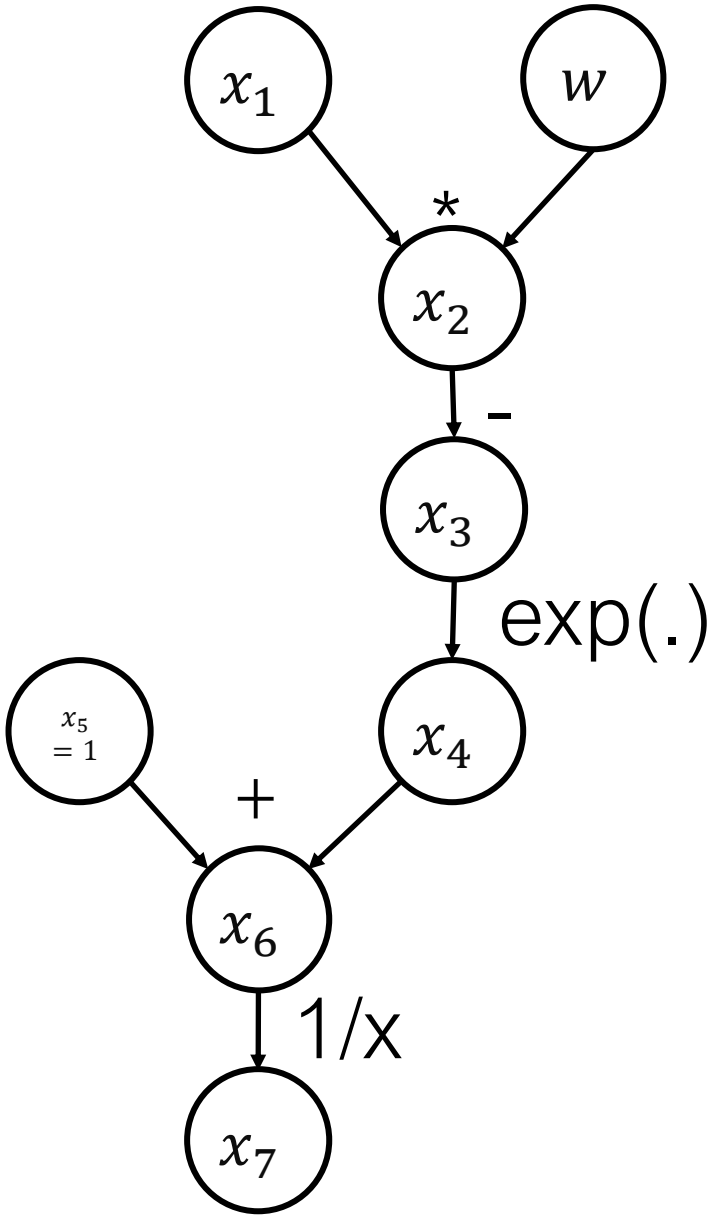
How to check the correctness of gradient

- use finite differences to check our gradient calculations

$$\frac{\partial f(x_1, x_2)}{\partial x_1} = \frac{f(x_1 + h, x_2) - f(x_1 - h, x_2)}{2h}$$

- Care the precision!
  - Use double precision (fp64)
  - Pick a small $h = \mathbf{0.000001}$
  - Compute the forward difference through the graph twice

Quiz



Today's Topic

- Learning algorithm for Neural Network
- Computation Graph
- Auto Differentiation
- Put Together: Implementing a Deep Learning Framework

Deep Learning Frameworks (also for LLMs)

- expressive to specify any neural networks
  - support future custom operators/layers
- productive for ML engineers
  - hide low-level details (no need to write cuda)
  - automatic differentiation (no need to derive gradient calculation manually)
- efficient in large-scale training and inference
  - automatically scale to data and model size
  - automatic hardware acceleration

Aspect	PyTorch	TensorFlow	JAX	NumPy
Primary Use	Deep learning	Deep learning	numerical and ML computing	numerical computing
Programming Paradigm	Dynamic (eager execution)	Static (Graph mode, or Eager)	Functional transformations	Procedural
Autograd	dynamic comp graph	static comp graph	Functional-based with grad/jit	Not available
Hardware Support	CPU, GPU, TPU	CPU, GPU, TPU	CPU, GPU, TPU	CPU only
Ease of Use	Pythonic	a bit learning curve	Pythonic and functional	Very easy, native python
Ecosystem	PyTorch Lightning, TorchVision	TensorBoard, TensorFlow Extended	integrates with NumPy	NA
Parallelism	Multi-GPU with DataParallel or DDP	Multi-GPU/TPU via tf.distribute	Multi-GPU/TPU via pmap	No parallelism

Deep Learning Framework Design Principles

- Dataflow graphs (computation graph) of primitive operators

- Deferred execution (two phases)

  1. Define program i.e., symbolic dataflow graph w/ placeholders, essentially constructing the computation graph

  2. Executes optimized version of program on set of available devices

Basic Components (follows tensorflow)

- A Computation Graph, which contains these nodes
  - Placeholder: to store the input data (as tensors/multi-dim array)
  - Variable: to store the network parameters
  - Constant: some static data
  - Operation: the mathmatical operations for each neural network layer, the input are any of these nodes, result is stored in output
    - each operation needs to define forward and backward operation
- Session: execution environment
  - Perform computation via the topological sort of the nodes

All data are represented Tensor

- A tensor is a multi-dimensional array. generalization to vector and matrix

tf.constant([[1, 2], [3, 4]])

is a 2x2 tensor with element type int32

tf.Tensor([[2 3] [4 5]], shape=(2, 2), dtype=int32)

Pytorch:
torch.tensor([[1., 2.], [3., 4.]])

Example Computation Graph in Tensorflow

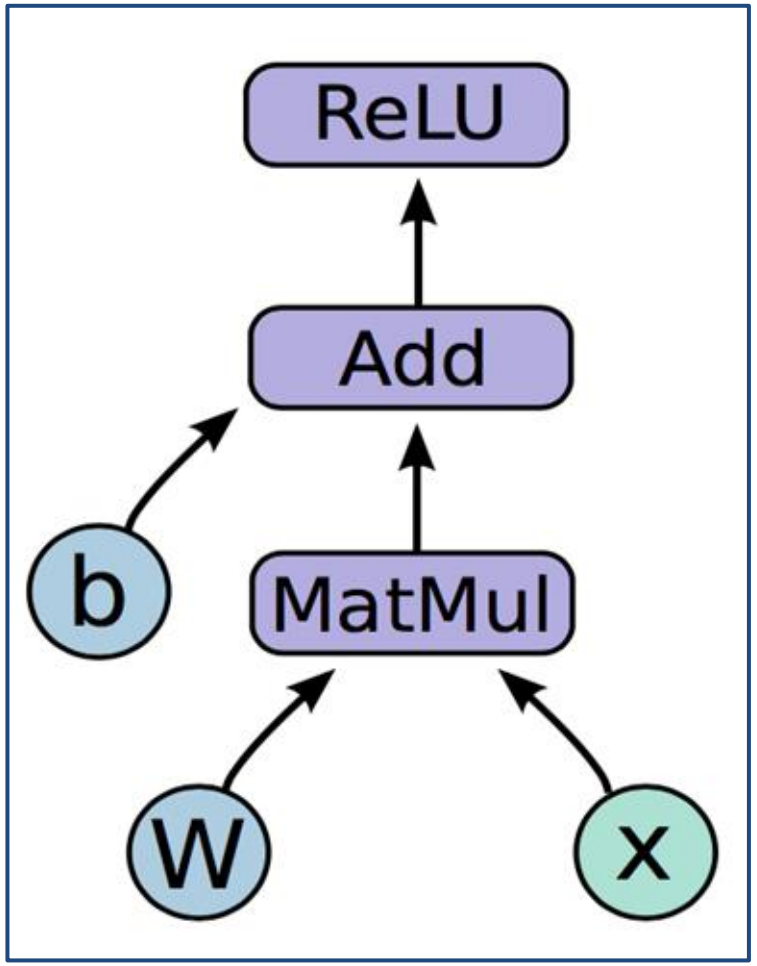
$$h = RELU(Wx + b)$$

```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100), -1, 1))

x = tf.placeholder(tf.float32, (1, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```

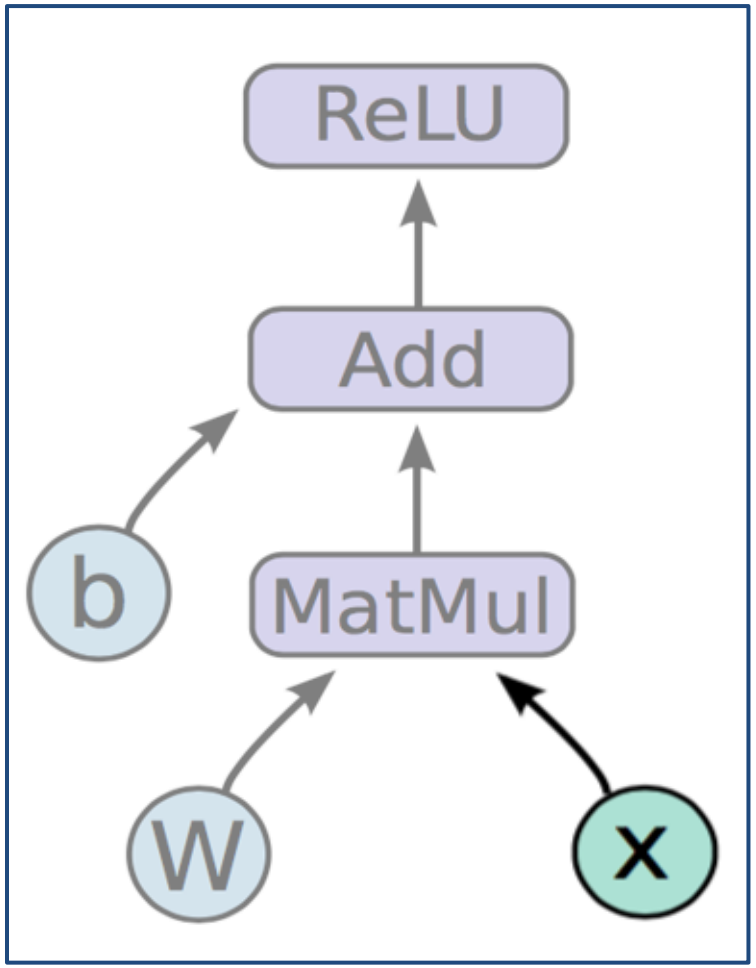


Placeholder Node (Tensorflow v1)

$$h = RELU(Wx + b)$$

```
x = tf.placeholder(tf.float32, (1, 784))
```

- Represent Inputs, Labels, …
- value is fed in at execution time
- No need to explicitly define Placeholder in Tensorflow v2

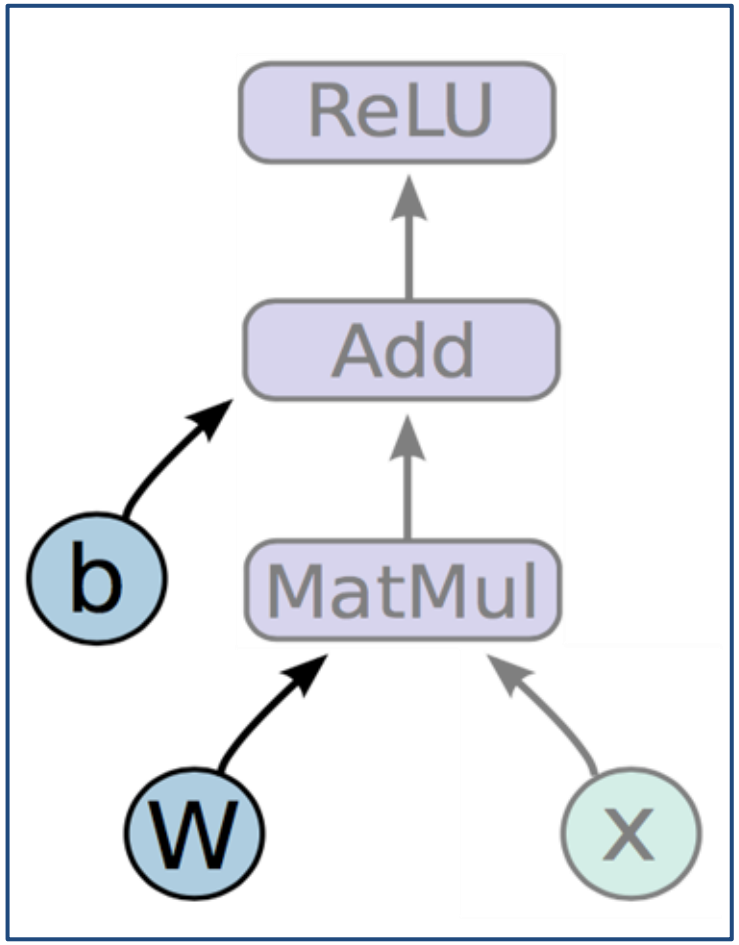


Variable Node

$$h = RELU(Wx + b)$$

```
b = tf.Variable(tf.zeros((100,)))
tf.Variable(initial_value=None,
  trainable=None,
  name=None)
```

- Variables are stateful nodes which output their current value.
- State is retained across multiple executions of a graph
- mostly parameters



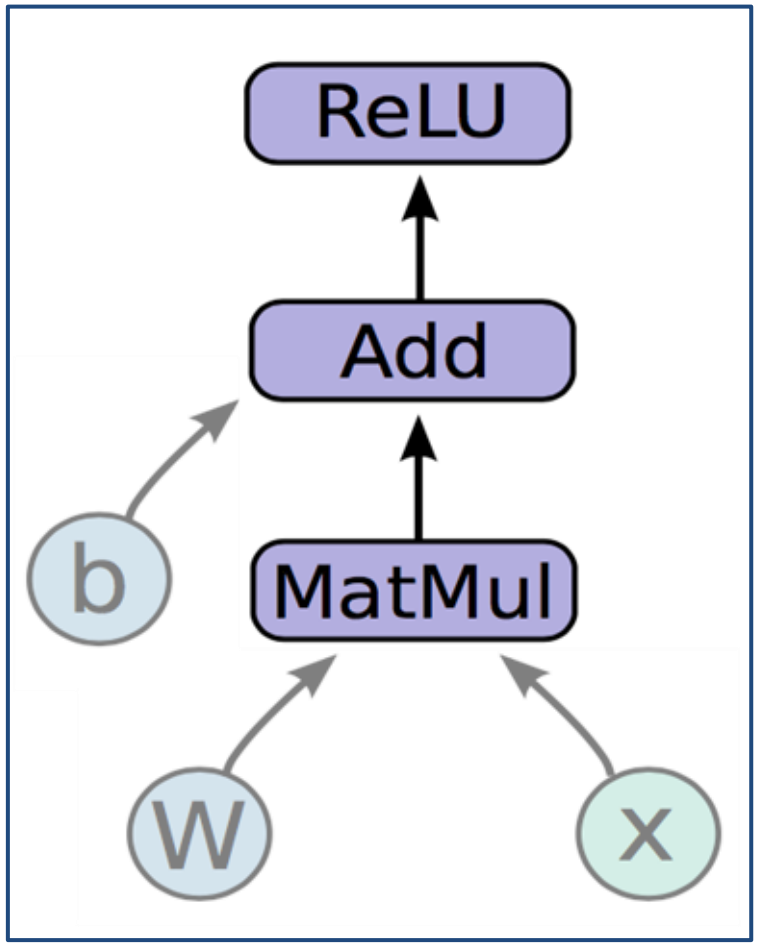
Operation Node

$$h = RELU(Wx + b)$$

tf.linalg.matmul(a, b): multiply two matrices

tf.math.add(a, b): Add elementwise

tf.nn.relu(a): Activate with elementwise rectified linear function $ReLu(x) = \begin{cases} 0, & x <= 0 \\ x, & x > 0 \end{cases}$



Pytorch:

torch.matmul(a, b)

torch.add(a, b)

torch.nn.ReLU(a)

Implementing Operation Node

- need to define the input nodes and forward/backward func

```
class AddOperation(Operation):
 # define Add operation a+b
 def __init__(self, a, b):
  # a, b are input nodes.
  super().__init__([a, b])

 def forward(self, a, b): # calculating the result of op
  return a + b

 def backward(self, upstream_grad):
  …
```

Defining Loss as a node

- Use **placeholder** for **labels**
- Build loss node using labels and **prediction**

```
prediction = tf.nn.softmax(...)  #Output of neural network
label = tf.placeholder(tf.float32, [100, 10])

cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)
```

Gradient Computation

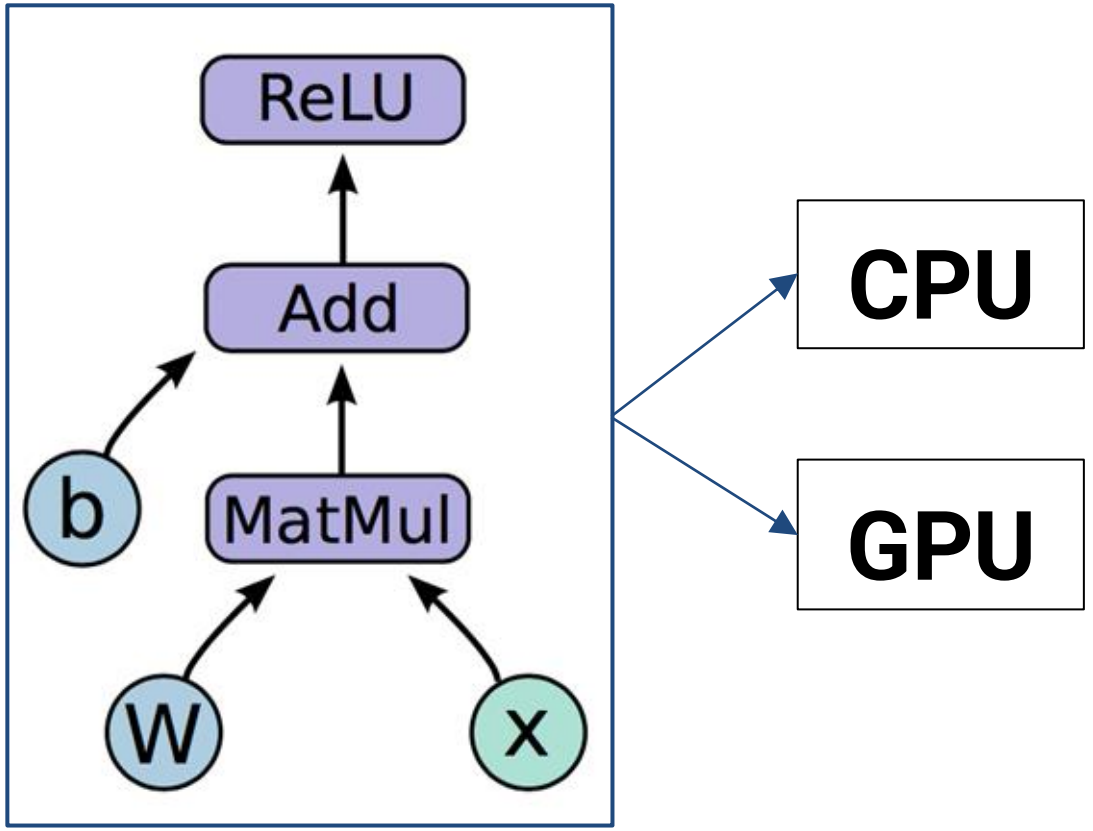
```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

- tf.train.GradientDescentOptimizer is an Optimizer object
- tf.train.GradientDescentOptimizer(lr).minimize(cross_entropy) adds optimization operation to computation graph
- TensorFlow graph nodes have attached gradient operations
- Gradient with respect to parameters computed with Auto Differentiation (recall previous)

Session

In TF v1, to deploy graph with a session: a binding to a particular execution context (e.g. CPU, GPU)

```
with tf.Session() as s:
  …
  s.run()
```



```python
import tensorflow as tf

with tf.Session() as sess:
    # Phase 1: constructing the graph
    a = tf.constant(15, name="a")
    b = tf.constant(5, name="b")
    prod = tf.multiply(a, b, name="Multiply")
    sum = tf.add(a, b, name="Add")
    res = tf.divide(prod, sum, name="Divide")

    # Phase 2: running the session
    out = sess.run(res)
    print(out)
```

Implementing Topological Sort

- For each Operation node (using isinstance to check)
  - recursively find the incoming nodes, visit them first and add node to visited nodes.

Implementing Session

- Apply topological sort on the computation graph starting from the final operation node
- Feeding data using a dictionary that maps Placeholder to actual data array
- Compute value for each node:
  - If a node is a placeholder, it should take value from feed_dict
  - If a node is variable or constant, it just use the node's value
  - If a node is an operation, it should get the node's input_nodes, and then apply forward

Code Practice: Implement Computation Graph

https://github.com/llmsystem/llmsys_code_examples/tree/main/mini_tensorflow

Please follow the instructions and fill in the code in

https://github.com/llmsystem/llmsys_code_examples/blob/main/mini_tensorflow/mini_tensorflow.ipynb

The full code is provided in

https://github.com/llmsystem/llmsys_code_examples/blob/main/mini_tensorflow/mini_tensorflow_full.ipynb

Summary

- Learning parameters of an NN needs gradient calculation
- Computation Graph
  - to perform computation: topological traversal along the DAG
- Auto Differentiation
  - building backward computation graph for gradient calculation
- Put together: Deep Learning Framework

  1. Define program i.e., symbolic computation graph w/ placeholders/variable/operation nodes
  2. Executes (optimized) computation graph on a set of available devices

Additional Reading

- Auto Diff survey, https://arxiv.org/abs/1502.05767
- The Elements of Differentiable Programming (Book), https://arxiv.org/abs/2403.14606
- TensorFlow: A System for Large-Scale Machine Learning, OSDI 2016.

Quiz

- on canvas

HW2

- https://llmsystem.github.io/llmsystemhomework/assignment_2/
- join the recitation session this Friday to learn about mini_torch framework for HW2
  - please bring your laptop for coding practice