11868 LLM Systems
Auto Differentiation

Lei Li

Carnegie Mellon University
Language Technologies Institute

Recap

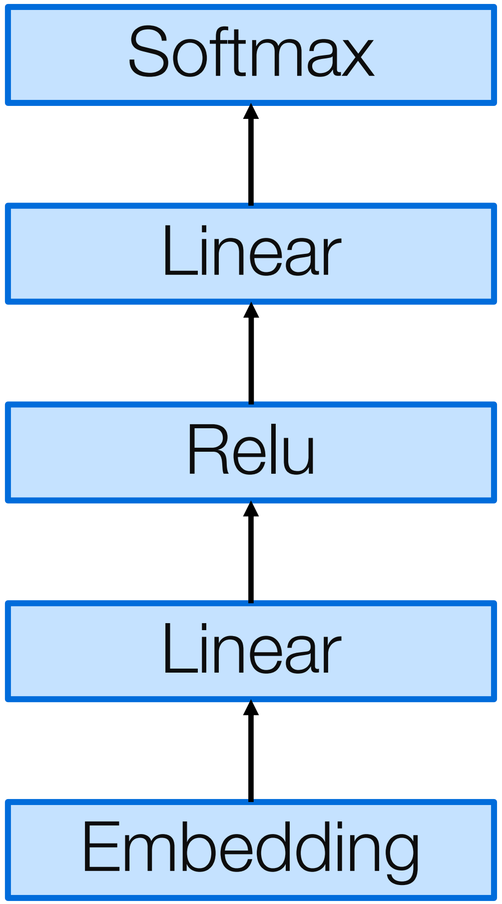
- Operators needed for Neural network
- GPU Architecture overview
  - GPU → SMs -> partitions
  - data transfer bandwidth
- Basic CUDA operations
  - lauch kernels as a grid of blocks of threads
- Matrix/Tensor Computation on GPU

Today's Topic

- Learning algorithm for Neural Network
- Computation Graph
- Auto Differentiation
- Gradient checking

A Simple Feedforward Neural Network

- Layers in FFN
  - Embedding (lookup table)
  - Linear
  - Relu
  - Softmax



It is a good movie

Loss for Classification

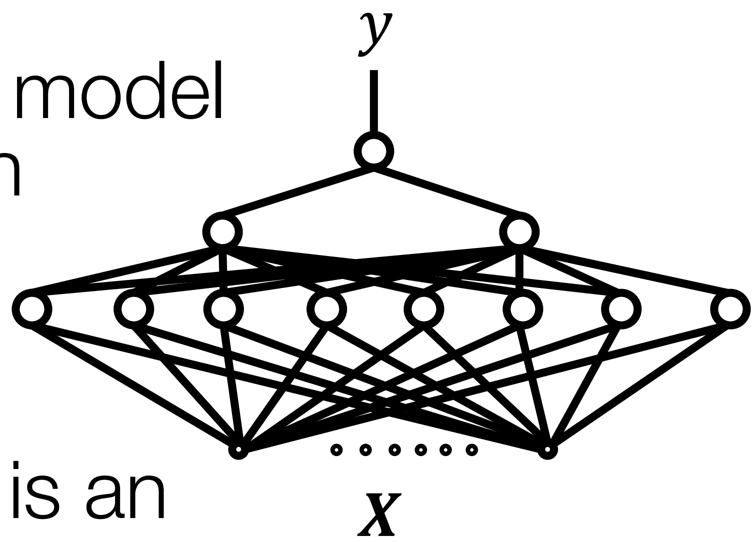
- Cross entropy

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

- Pytorch CrossEntropyLoss is implemented as
    - Negative Likelihood on Log(Softmax(h))
    - Should pass logit (linear before softmax) as input

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



Generic Iterative Algorithm

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

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

- 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

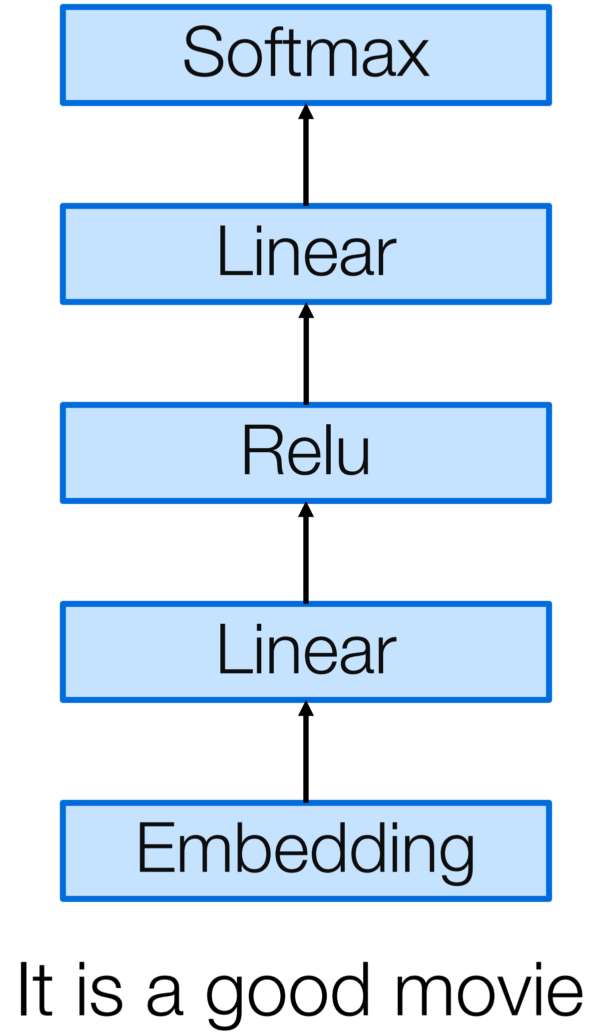
(Stochastic) Gradient Descent Algorithm

```
set learning rate eta.
1.set initial parameter θ ← θ_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 = ∂err(θ)/∂θ
8.         total_g += g
9.      update θ = θ - eta * total_g / N
```

$$\theta \leftarrow \theta_0, \qquad g = \frac{\partial \mathrm{err}(\theta)}{\partial \theta}$$

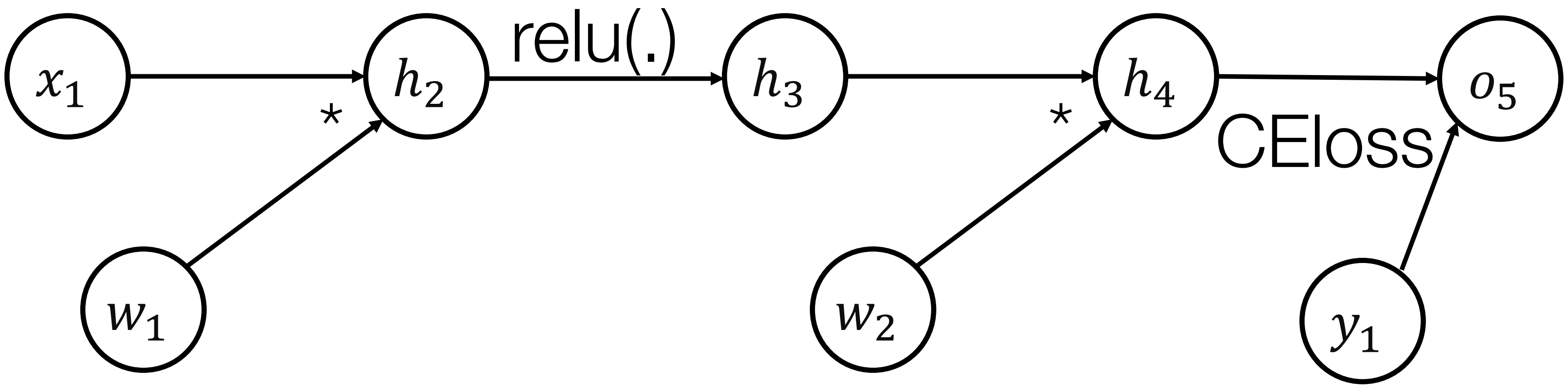
How to compute the gradient for every parameter?

- Goal: $\frac{\partial l}{\partial w_i}$
- Forward computation
- Backpropogation



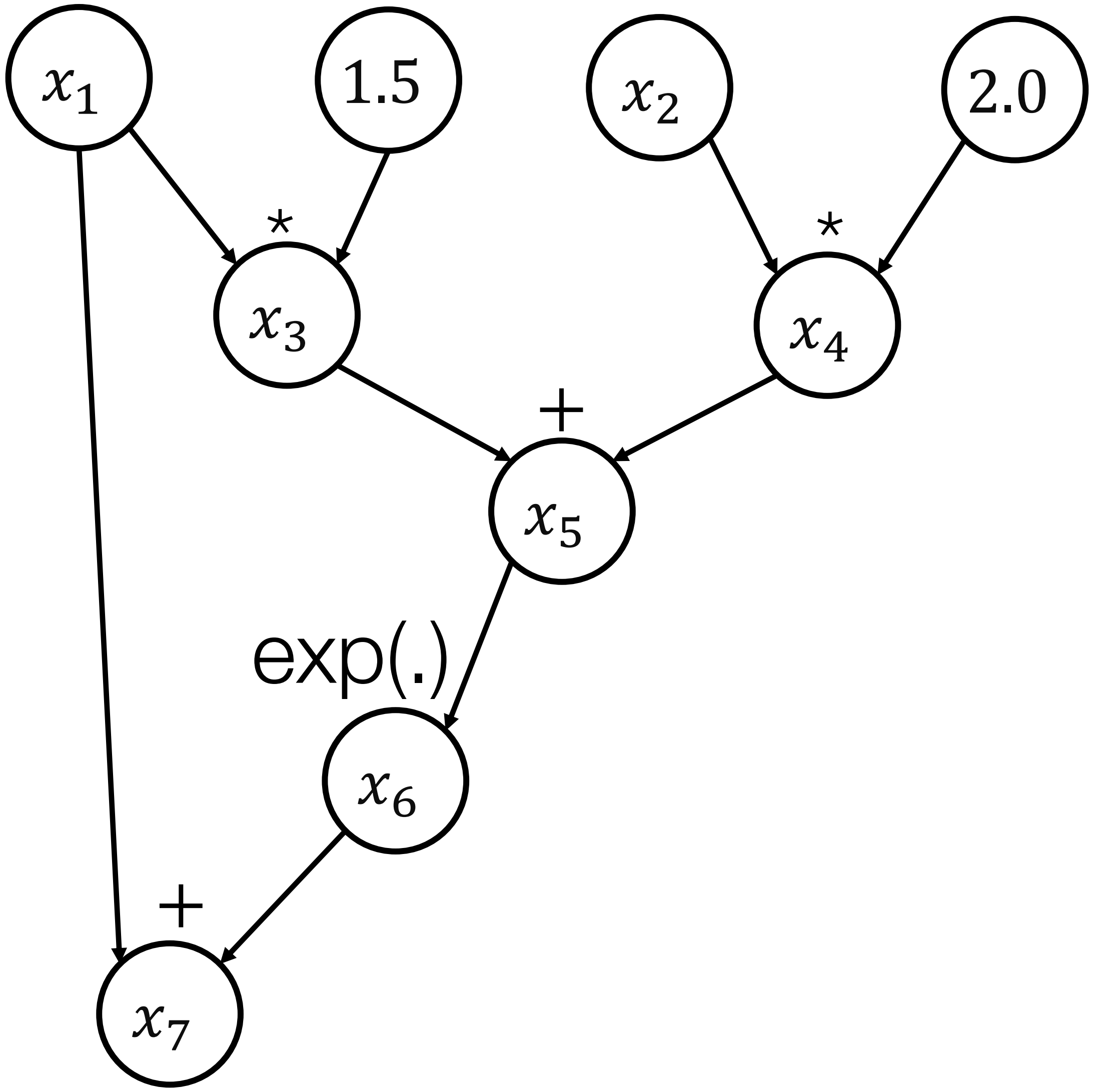
Computation Graph

- 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)



Computation:

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

Building Computation Graph

- Most autodiff systems, including Pytorch/Autograd, explicitly construct the computation graph.
- TensorFlow provide mini-languages for building computation graphs directly.
- Disadvantage: need to learn a totally new API.
- Autograd (JAX) instead builds them by tracing the forward pass computation (similar to numpy).

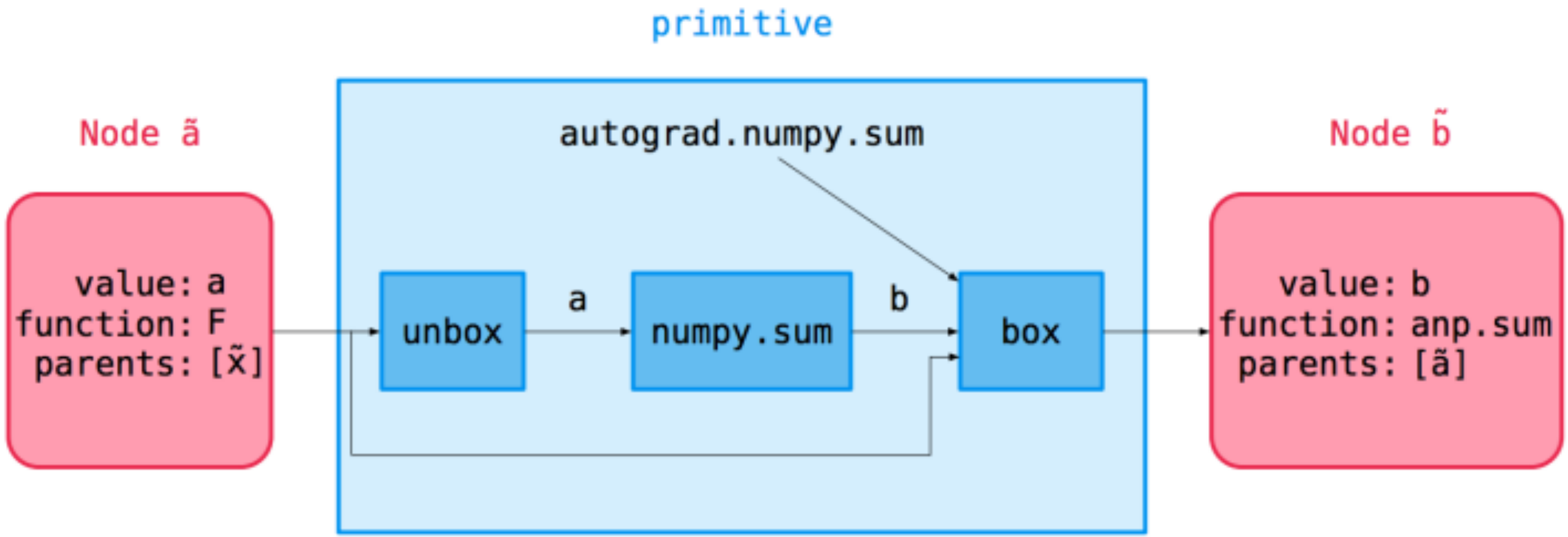
Implementation

- Node class, with attributes
  - value: the actual value computed on a particular set of inputs
  - fun: the primitive operation defining the node
  - args and kwargs: the arguments the op was called with
  - parents: the parent Nodes

https://github.com/mattjj/autodidact

Wrapper around Numpy

- Autograd's NumPy module provides primitive ops which look and feel like NumPy functions, but secretly build the computation graph.



Gradient Calculation

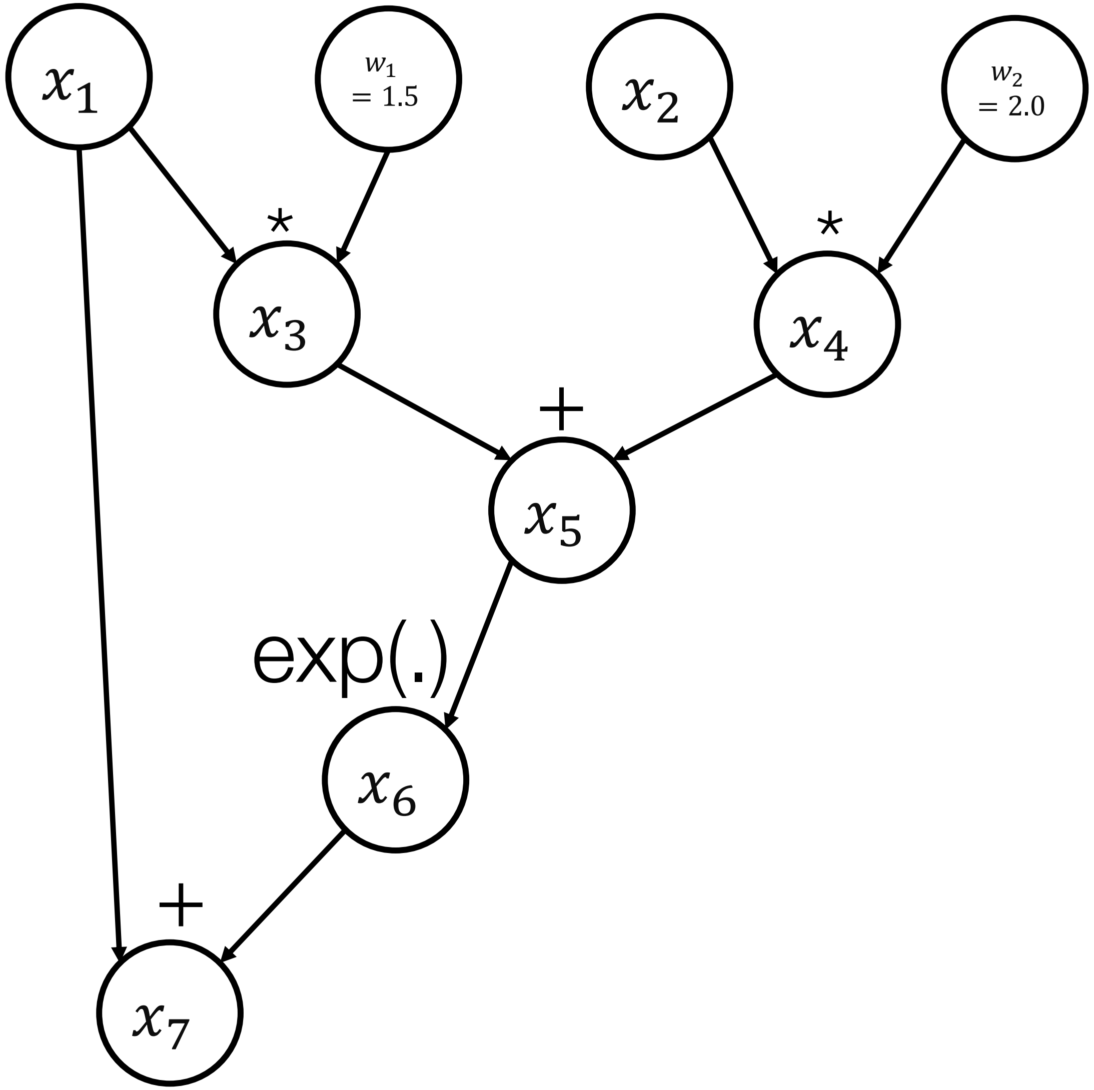
- To learn a neural network, we need gradient of loss function w.r.t. parameters.

- 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)

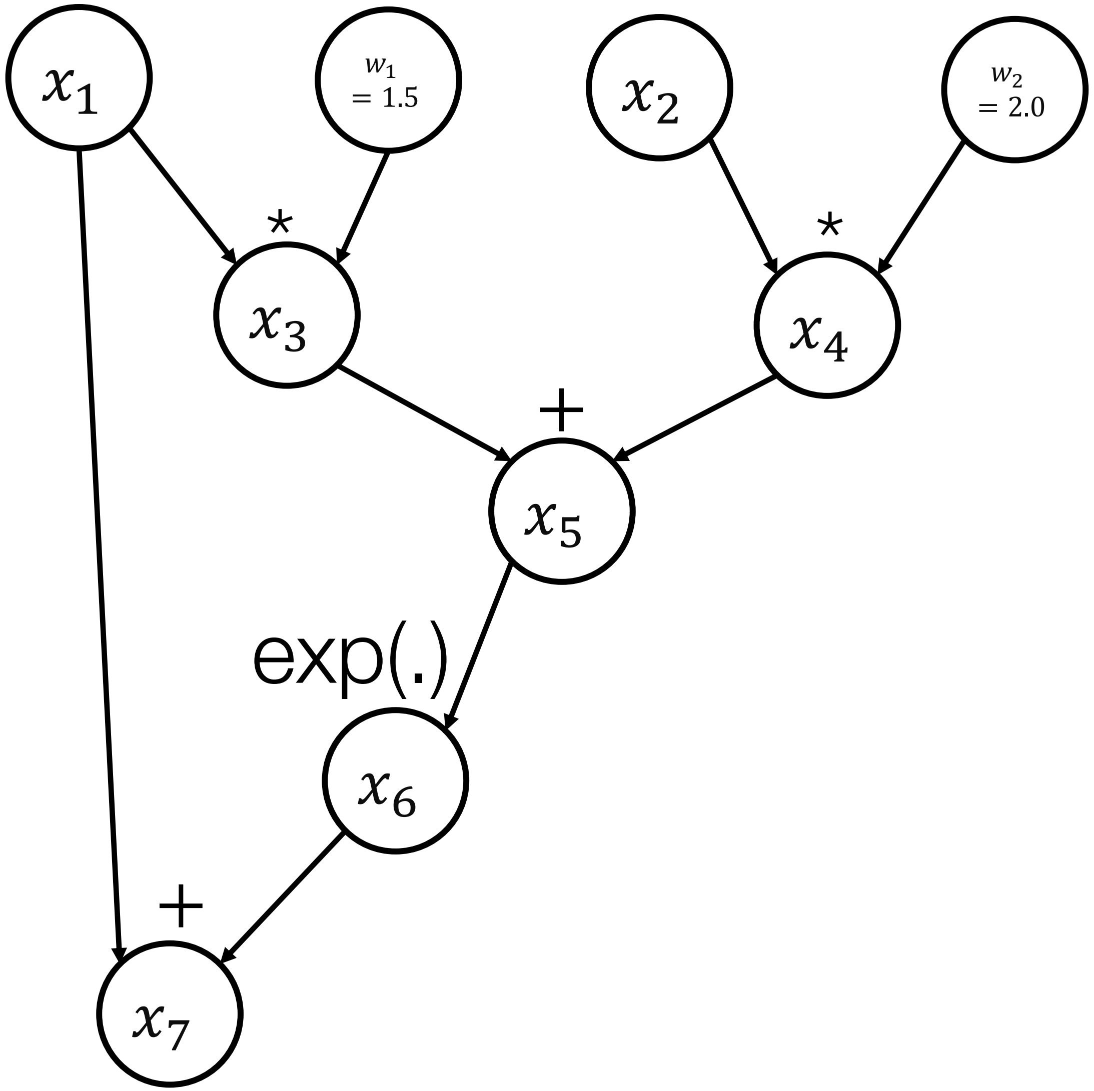


Computing the derivatives $\frac{\partial y}{\partial x_i}$

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

x1= 3, x2=0.5

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



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

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

$$\overline{x_7} = 1$$

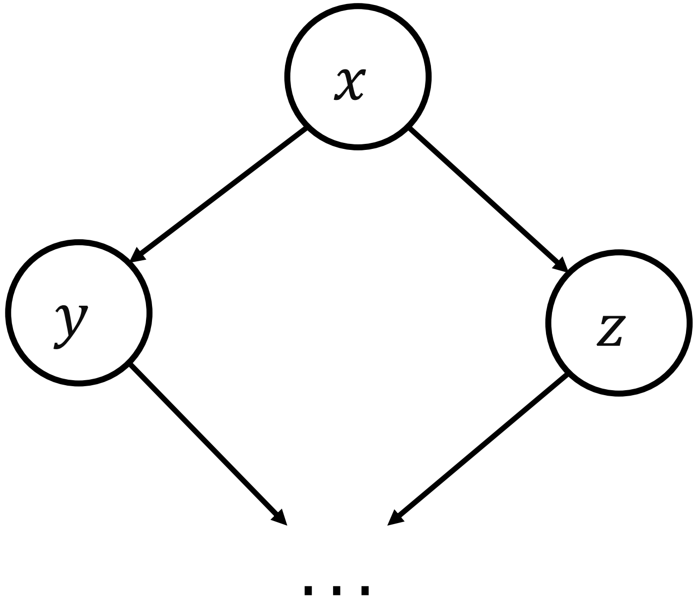
$$\overline{x_6} = 1$$

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

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

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

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



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

Partial derivatives for Vectors

Jacobian

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

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

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

Implementing Vector-Jacobian Product

- For each primitive operation, we must specify VJPs for each of its arguments.

- defvjp (defined in core.py) is a convenience routine for registering VJPs.

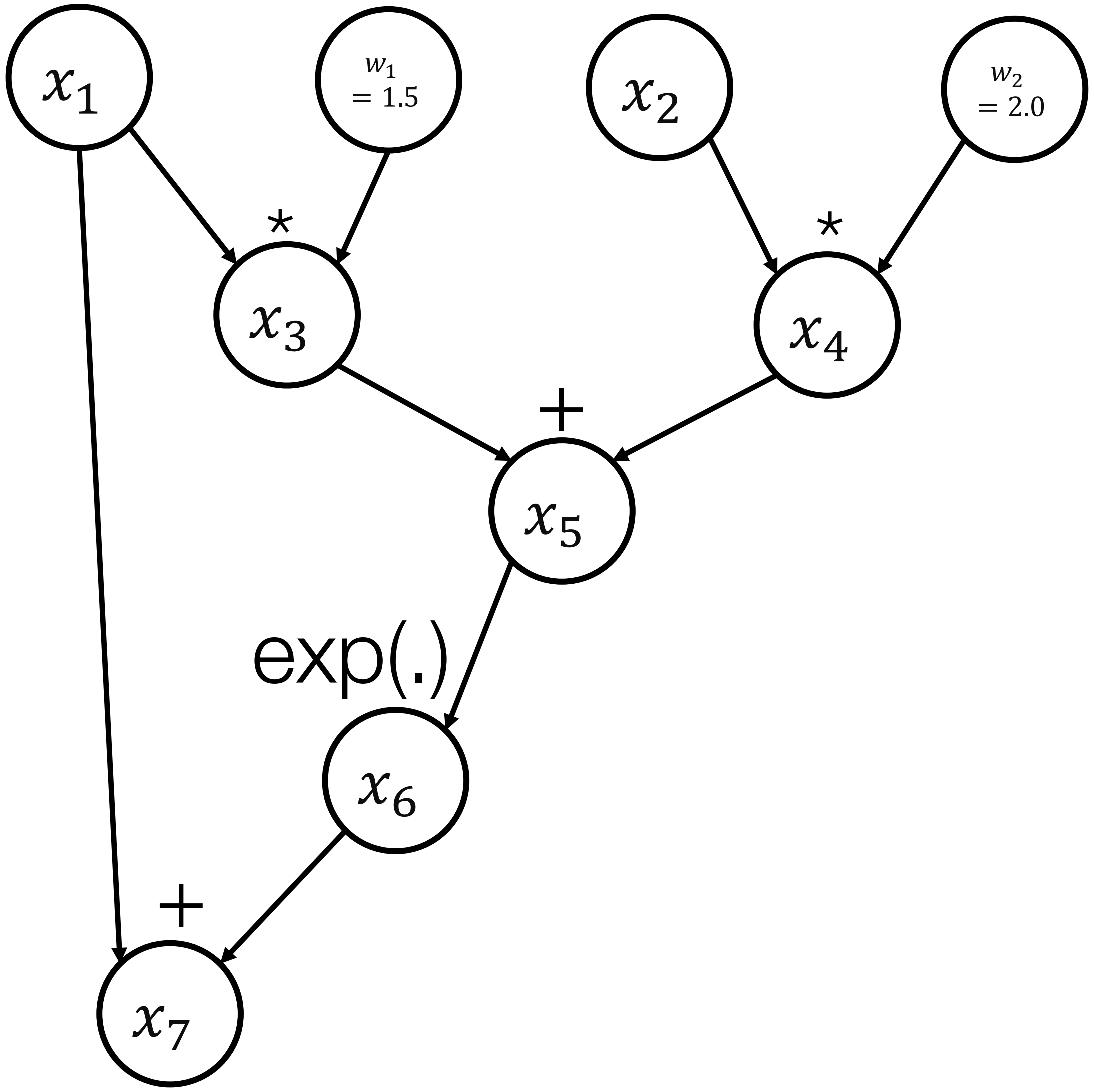
```
defvjp(anp.exp,    lambda g, ans, x: ans * g)
```

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)

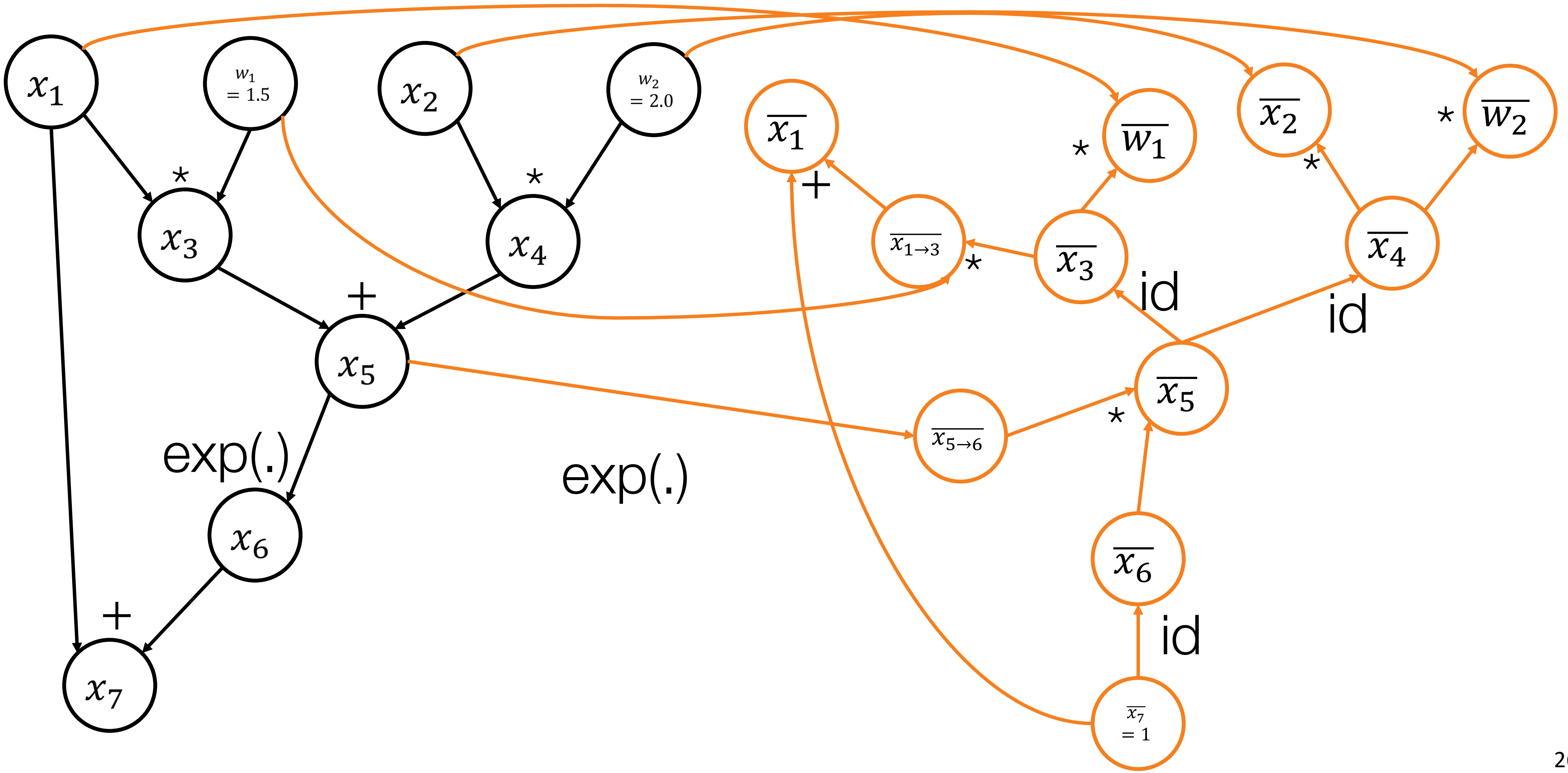


Computing the derivatives $\frac{\partial y}{\partial x_i}$

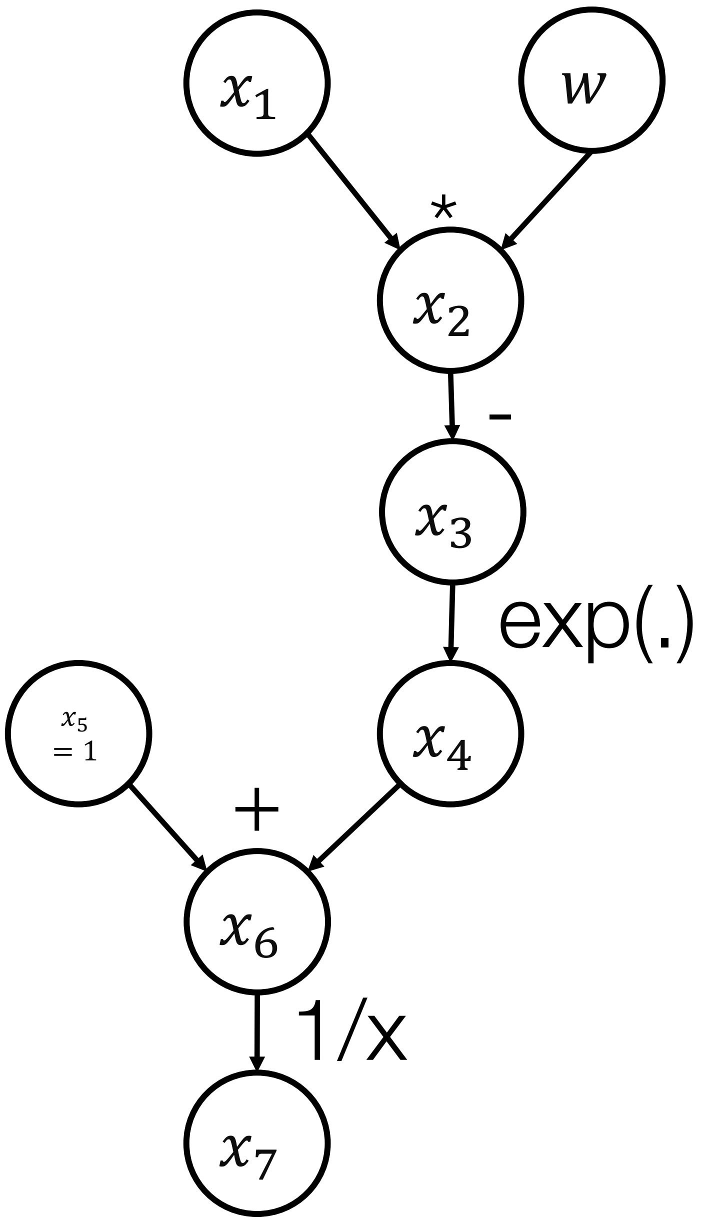
Define $\overline{x_i} = \frac{\partial y}{\partial x_i}$

x1= 3, x2=0.5

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



Exercise



Implementing Backward Pass

```python
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
```

```python
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

```python
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
```

```python
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
```

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 $h = 0.000001$
  - Compute the forward difference through the graph twice

Summary

- Learning algorithm for Neural Network
  - stochastic gradient descent
- Computation Graph
  - topological traversal along the DAG
- Auto Differentiation
  - building backward computation graph
- https://github.com/mattjj/autodidact/

Reading for Next Class

- TensorFlow: A System for Large-Scale Machine Learning, OSDI 2016.