Topics:

- Backpropagation / Automatic Differentiation
- Jacobians

CS 4644 / 7643-A
ZSOLT KIRA

- **Assignment 1 out!**
  - **Due Feb 3rd (with grace period 5th)**
  - Start now, start now, start now!
  - Start now, start now, start now!
  - Start now, start now, start now!

- Resources:
  - These lectures
  - Matrix calculus for deep learning
  - Gradients notes and MLP/ReLU Jacobian notes.
  - Assignment 1 (@67) and matrix calculus (@86), convex optimization (@89)

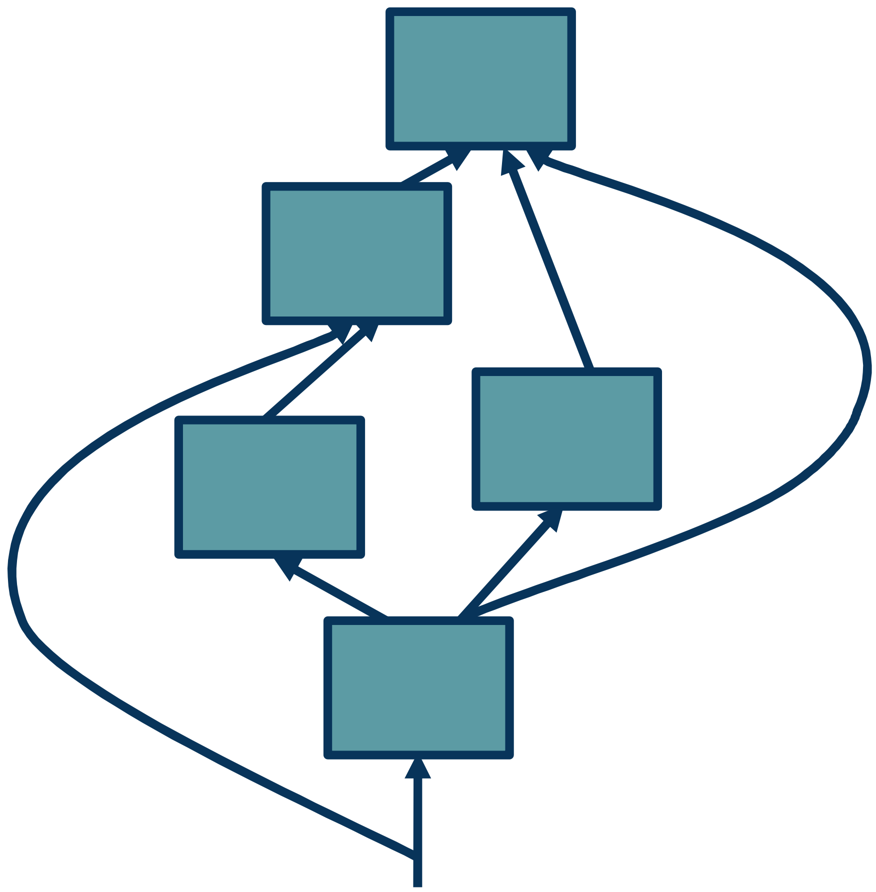
- Piazza: Project teaming thread
  - Will post video of project overview

To develop a general algorithm for this, we will view the function as a **computation graph**

Graph can be any **directed acyclic graph (DAG)**

- Modules must be differentiable to support gradient computations for gradient descent

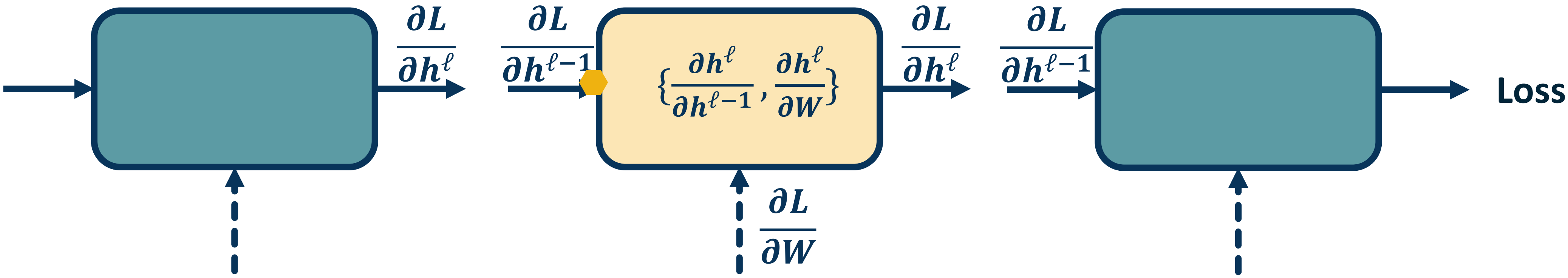
A **training algorithm** will then process this graph, **one module at a time**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

A General Framework

- We want to to compute: $\left\{\frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W}\right\}$



- We will use the *chain rule* to do this:

Chain Rule: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$

Backpropagation: a simple example

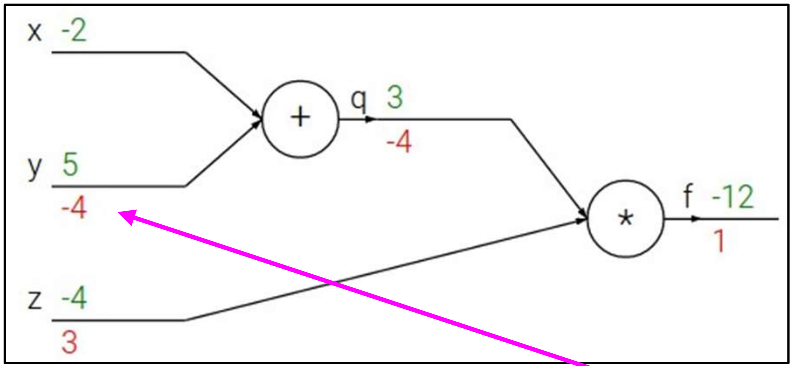
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial y}$$

Chain rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}}$$

Upstream gradient

Local gradient

Conventions:

- Size of derivatives for scalars, vectors, and matrices:
  Assume we have scalar $\boldsymbol{s} \in \mathbb{R}^{1}$, vector $\boldsymbol{v} \in \mathbb{R}^{m}$, i.e. $\boldsymbol{v} = [\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_m}]^{T}$ and matrix $\boldsymbol{M} \in \mathbb{R}^{k \times \ell}$

	$\boldsymbol{S}$	$\boldsymbol{V}$	$\boldsymbol{M}$
$\boldsymbol{S}$	$\frac{\partial s_1}{\partial s_2}$	$\frac{\partial s}{\partial v}$	$\frac{\partial s}{\partial M}$
$\boldsymbol{V}$	$\frac{\partial v}{\partial s}$	$\frac{\partial v_1}{\partial v_2}$	Tensors
$\boldsymbol{M}$	$\frac{\partial M}{\partial s}$	Tensors	Tensors

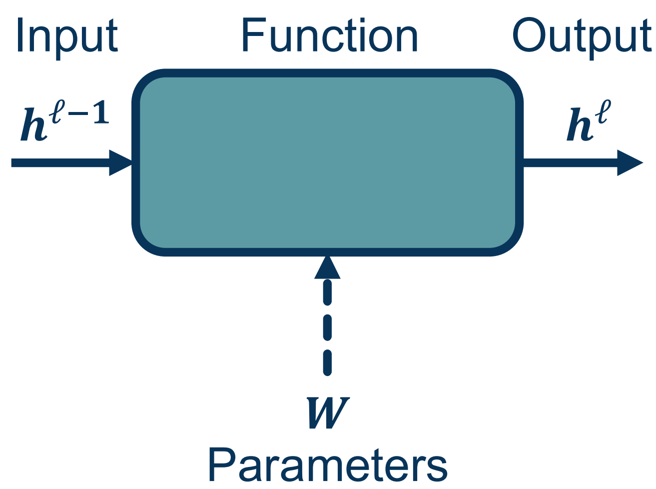
- What is the size of $\frac{\partial L}{\partial W}$?
  - Remember that loss is a **scalar** and W is a matrix:

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b2 \\ w_{31} & w_{32} & \cdots & w_{3m} & b3 \end{bmatrix}$$

$$W$$

Jacobian is also a matrix:

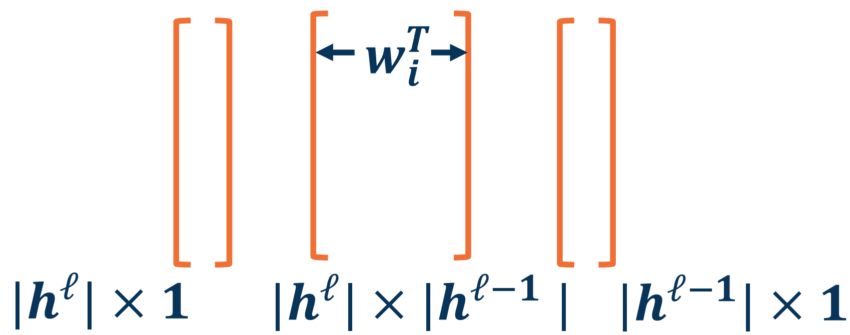
$$\begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \cdots & \frac{\partial L}{\partial w_{1m}} & \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial w_{21}} & \cdots & \cdots & \frac{\partial L}{\partial w_{2m}} & \frac{\partial L}{\partial b_2} \\ \cdots & \cdots & \cdots & \frac{\partial L}{\partial w_{3m}} & \frac{\partial L}{\partial b_3} \end{bmatrix}$$



Define:

$$h_i^{\ell} = w_i^T h^{\ell-1}$$

$$h^{\ell} = W h^{\ell-1}$$



Fully Connected (FC) Layer: Forward Function

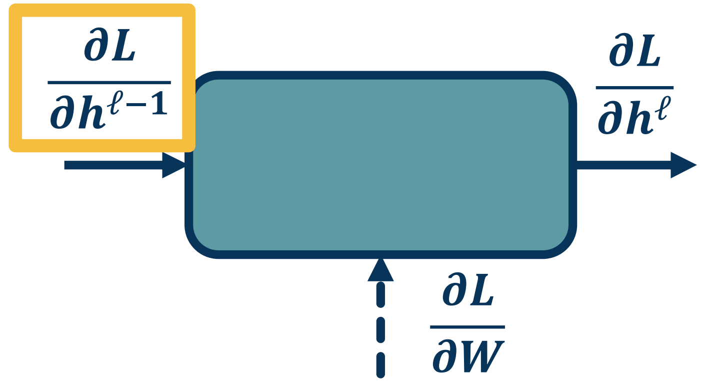
Fully Connected (FC) Layer

$$h^\ell = W h^{\ell-1}$$

$$\frac{\partial h^\ell}{\partial h^{\ell-1}} = W$$

Define:

$$h_i^\ell = w_i^T h^{\ell-1}$$



$$\boxed{\frac{\partial L}{\partial h^{\ell-1}}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^\ell}{\partial h^{\ell-1}}$$

$1 \times |h^{\ell-1}| \quad 1 \times |h^{\ell}| \quad |h^\ell| \times |h^{\ell-1}|$

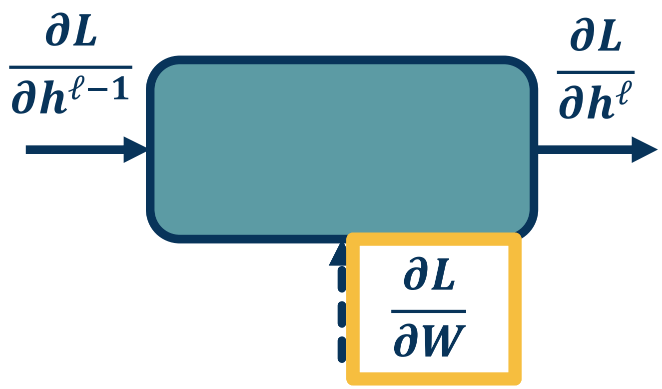
$$h^{\ell} = W h^{\ell-1}$$

$$\frac{\partial h^{\ell}}{\partial h^{\ell-1}} = W$$

Define:

$$h_i^{\ell} = w_i^T h^{\ell-1}$$

$$\frac{\partial h_i^{\ell}}{\partial w_i^T} = h^{(\ell-1),T}$$



Note doing this on full W matrix would result in Jacobian tensor!

But it is *sparse* – each output only affected by corresponding weight row

$$\boxed{\frac{\partial L}{\partial w_i^T}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial w_i^T}$$

$$\begin{bmatrix} \quad \end{bmatrix} \begin{bmatrix} \quad \end{bmatrix} \begin{bmatrix} \leftarrow & 0 & \rightarrow \\ \leftarrow & \frac{\partial h_i^{\ell}}{\partial w_i^T} & \rightarrow \\ \leftarrow & 0 & \rightarrow \end{bmatrix}$$

$$1 \times |h^{\ell-1}| \quad 1 \times |h^{\ell}| \quad |h^{\ell}| \times |h^{\ell-1}|$$

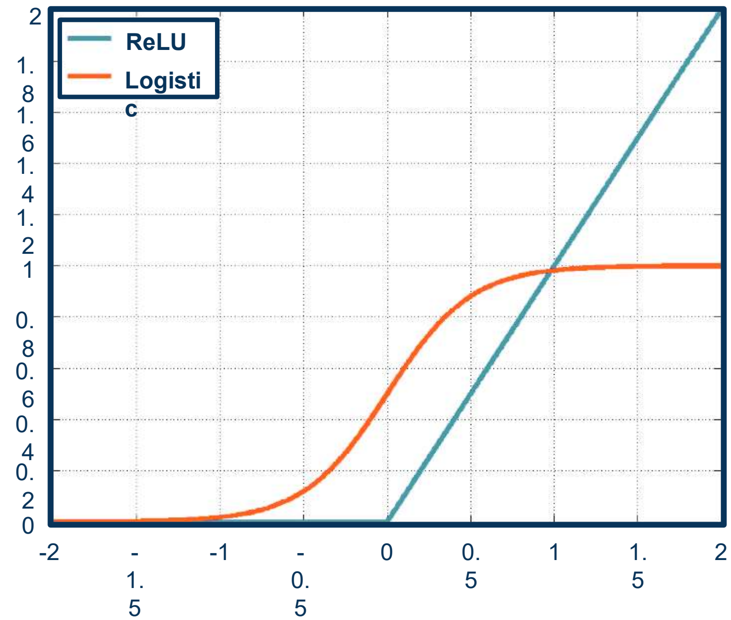
Fully Connected (FC) Layer

We can employ **any differentiable (or piecewise differentiable) function**

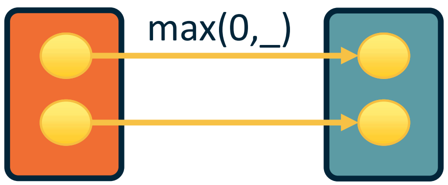
A common choice is the **Rectified Linear Unit**

- Provides non-linearity but better gradient flow than sigmoid
- Performed **element-wise**

How many parameters for this layer?



$$h^{\ell} = \max(0, h^{\ell-1})$$

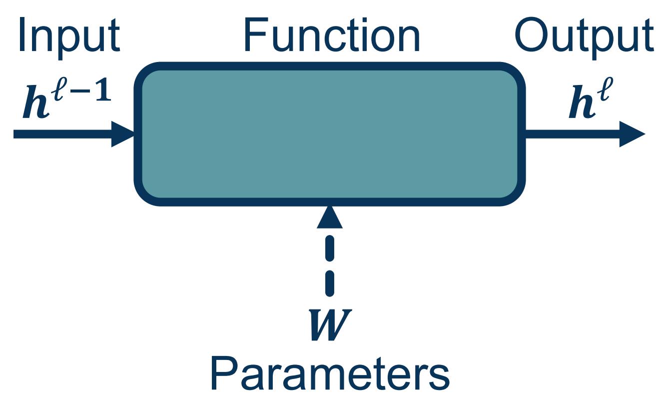


Full Jacobian of ReLU layer is **large** (output dim x input dim)

- But again it is **sparse**
- Only **diagonal values non-zero** because it is element-wise
- An output value affected only by **corresponding input value**

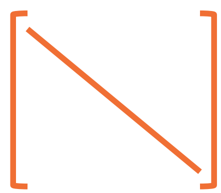
Max function **funnels gradients through selected max**

- Gradient will be **zero** if input **<= 0**



Forward: $h^{\ell} = \max(0, h^{\ell-1})$

Backward: $\frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial h^{\ell-1}}$



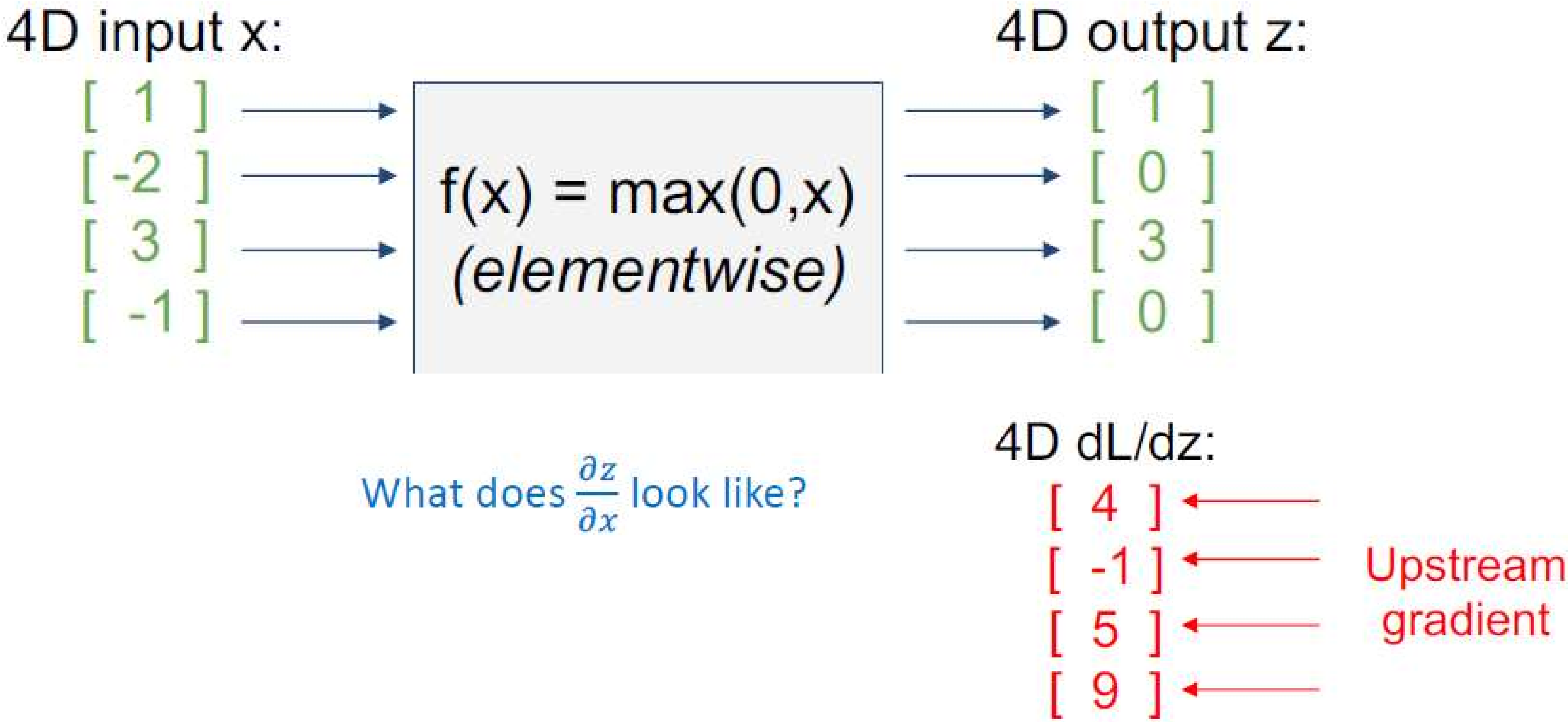
$|h^{\ell} \times h^{\ell-1}|$

For diagonal

$$\frac{\partial h^{\ell}}{\partial h^{\ell-1}} = \begin{cases} 1 & if\ h^{\ell-1} > 0 \\ 0 & otherwise \end{cases}$$

Jacobian of ReLU

Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n



What does $\frac{\partial z}{\partial x}$ look like?

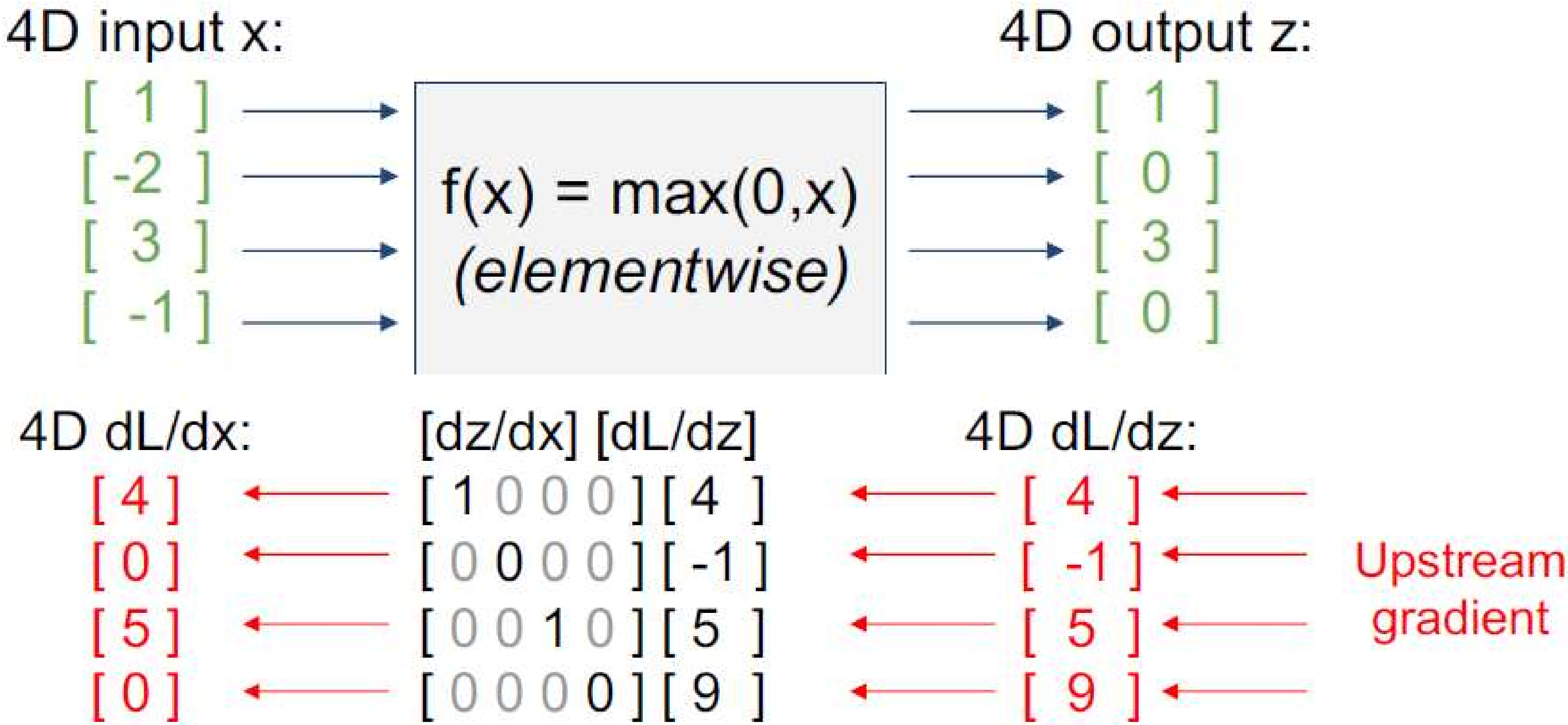


Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

For element-wise ops, jacobian is **sparse**: off-diagonal entries always zero!
Never **explicitly** form Jacobian -- instead use elementwise multiplication

- Neural networks involves composing simple functions into a **computation graph**

- Optimization (updating weights) of this graph is through backpropagation
  - Recursive algorithm: Gradient descent (partial derivatives) plus chain rule

- Remaining questions:
  - How does this work with vectors, matrices, tensors?
    - Across a composed function? **This Time!**
  - How can we implement this algorithmically to make these calculations automatic? **Automatic Differentiation**

Summary

Georgia Tech

Vectorizaiton in Function Compositions

Composition of Functions: $f\big(g(x)\big) = (f \circ g)(x)$

A complex function (e.g. defined by a neural network):

$$f(x) = g_\ell\left(g_{\ell-1}\big(\ldots g_1(x)\big)\right)$$

$$f(x) = g_\ell \circ g_{\ell-1} \ldots \circ g_1(x)$$

(Many of these will be parameterized)

(Note you might find the opposite notation as well!)

Scalar Case

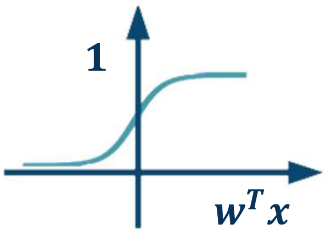
Vector Case

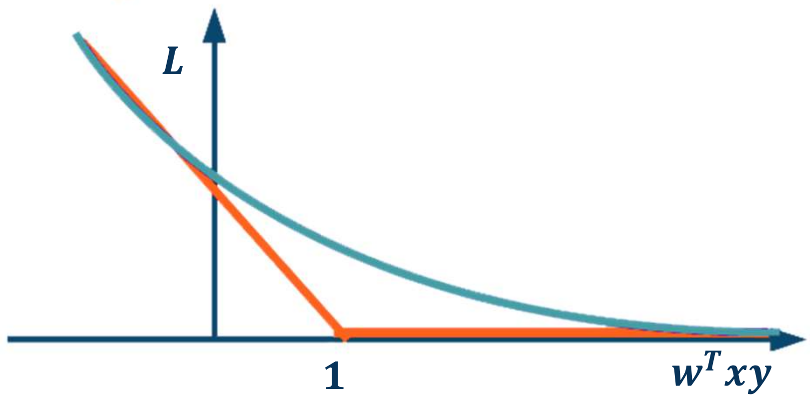
Jacobian View of Chain Rule

Graphical View of Chain Rule

Chain Rule: Cascaded

- Input: $\boldsymbol{x} \in \boldsymbol{R}^{\boldsymbol{D}}$
- Binary label: $\boldsymbol{y} \in \{-\mathbf{1}, +\mathbf{1}\}$
- Parameters: $\boldsymbol{w} \in \boldsymbol{R}^{\boldsymbol{D}}$
- Output prediction: $\boldsymbol{p}(\boldsymbol{y} = \mathbf{1}|\boldsymbol{x}) = \frac{\mathbf{1}}{\mathbf{1}+\boldsymbol{e}^{-\boldsymbol{w}^T\boldsymbol{x}}}$
- Loss: $\boldsymbol{L} = \frac{1}{2}\|\boldsymbol{w}\|^2 - \boldsymbol{\lambda}\mathbf{log}\big(\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x})\big)$





Log Loss

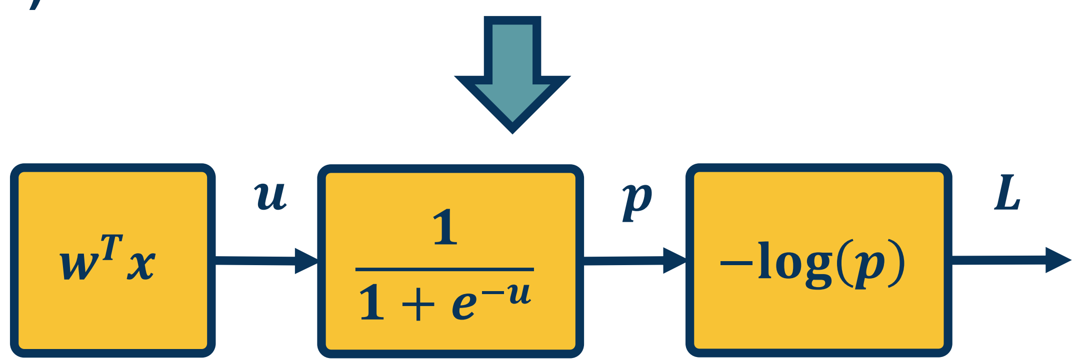
Adapted from slide by Marc'Aurelio Ranzato

We have discussed **computation graphs for generic functions**

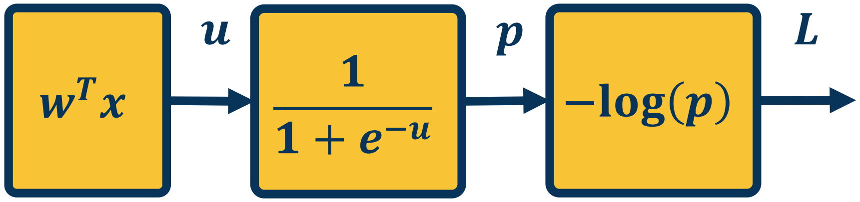
Machine Learning functions **(input -> model -> loss function)** is also a computation graph

We can use the **computed gradients from backprop/automatic differentiation** to update the weights!

$$-\log\left(\frac{1}{1+e^{-w^T x}}\right)$$



Neural Network Computation Graph



$\bar{L} = 1$

$$\bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p}$$

where $p = \sigma(w^T x)$ and $\sigma(x) = \frac{1}{1+e^{-x}}$

$$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p}\ \frac{\partial p}{\partial u} = \bar{p}\ \sigma(1-\sigma)$$

$$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\ \frac{\partial u}{\partial w} = \bar{u} x^T$$

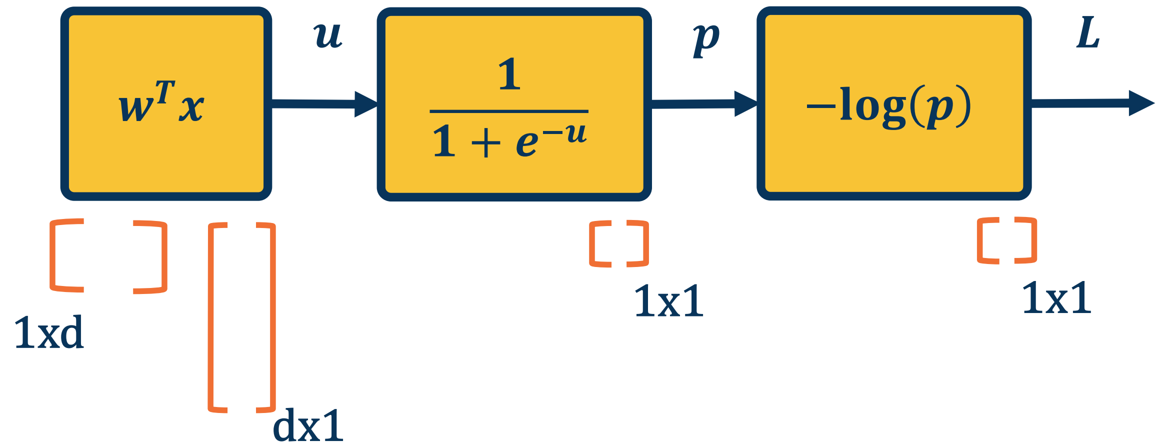
We can do this in a combined way to see all terms together:

$$\bar{w} = \frac{\partial L}{\partial p}\ \frac{\partial p}{\partial u}\ \frac{\partial u}{\partial w} = \bar{L}\ \bar{p}\ \bar{u} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma\ (w^T x))x^T$$
$$= -\left(1 - \sigma(w^T x)\right) x^T$$

This effectively shows gradient flow along path from L to w

Example Gradient Computations

The chain rule can be computed as a **series of scalar, vector, and matrix linear algebra operations**



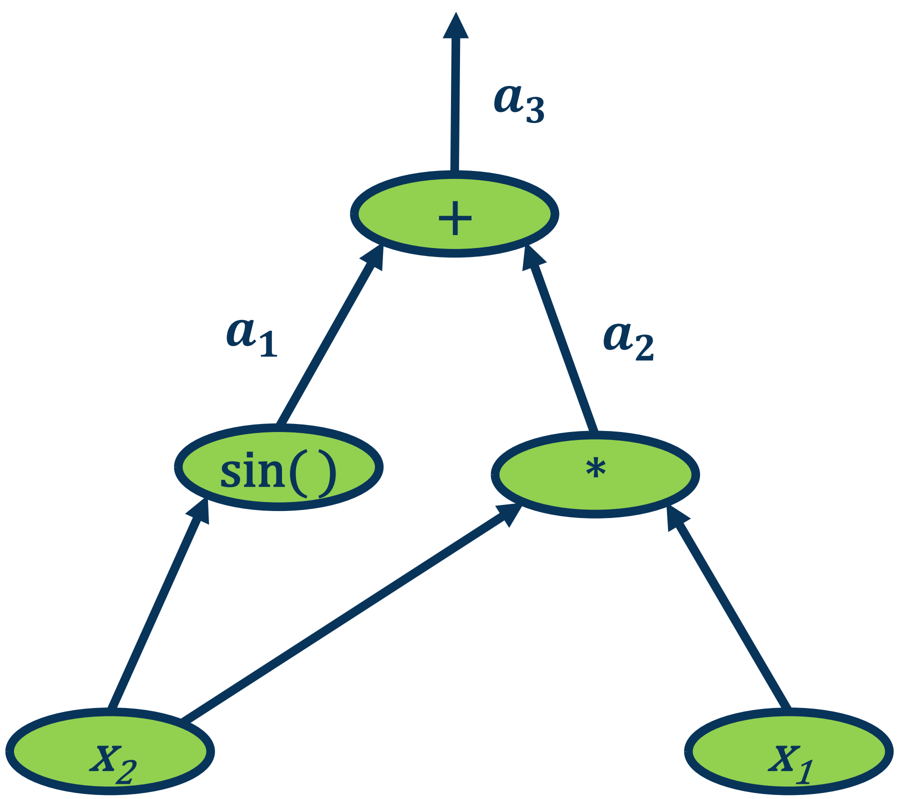
Extremely efficient in graphics processing units (GPUs)

$$\bar{w} = -\underbrace{\frac{1}{\sigma(w^T x)}}_{1\text{x}1}\,\underbrace{\sigma(w^T x)}_{1\text{x}1}\underbrace{(1-\sigma(w^T x))}_{1\text{x}1}\underbrace{x^T}_{1\text{x}d}$$

Vectorized Computations

Example

$$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$$



We want to find the **partial derivative of output f** (output) with respect to **all intermediate variables**

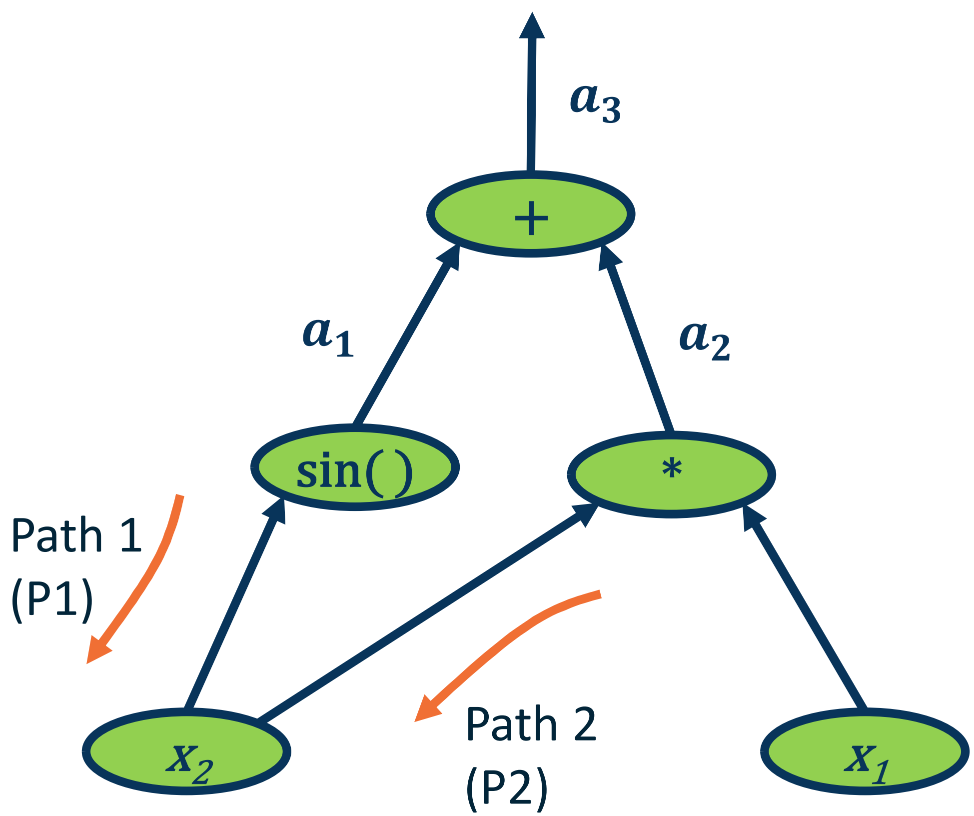
- Assign intermediate variables

  Simplify notation:

  Denote bar as: $\overline{a_3} = \frac{\partial f}{\partial a_3}$

- Start at **end** and move **backward**

$$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$$



$$\overline{a_3} = \frac{\partial f}{\partial a_3} = 1$$

$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial (a_1+a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3}\ 1 = \overline{a_3}$$

$$\overline{a_2} = \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial a_2} = \overline{a_3}$$

$$\overline{x_2^{P1}} = \frac{\partial f}{\partial a_1}\frac{\partial a_1}{\partial x_2} = \overline{a_1}\ \cos(x_2)$$

$$\overline{x_2^{P2}} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2}\frac{\partial (x1x2)}{\partial x_2} = \overline{a_2}x_1$$

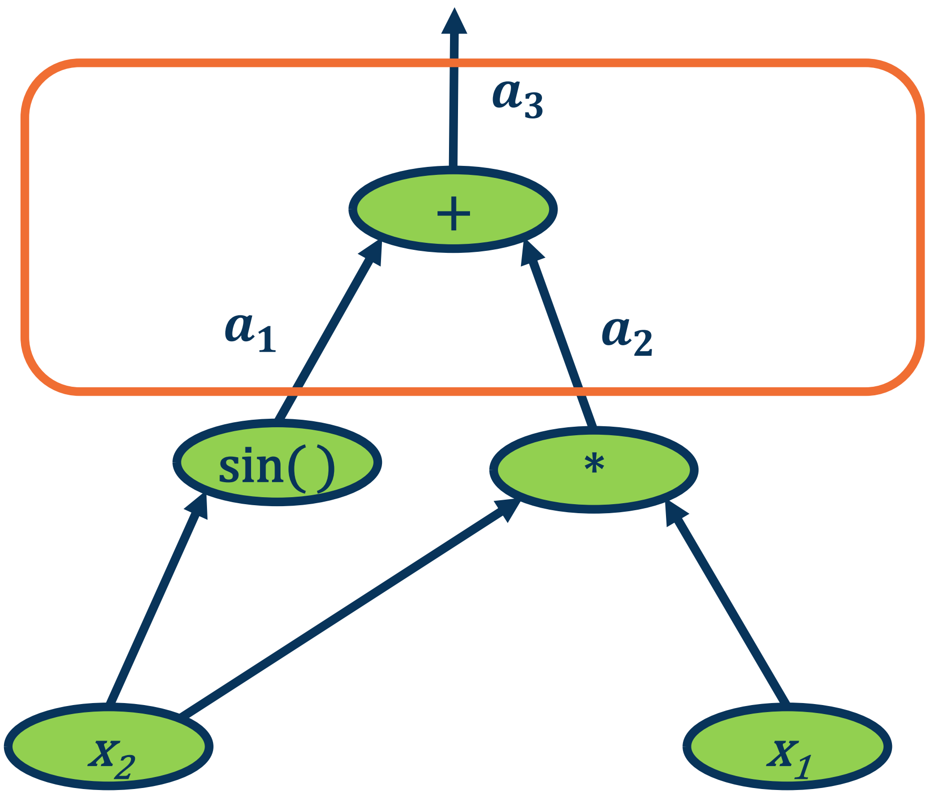
Gradients from multiple paths **summed**

$$\overline{x_1} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_1} = \overline{a_2}x_2$$

Example

Patterns of Gradient Flow: Addition

$$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$$

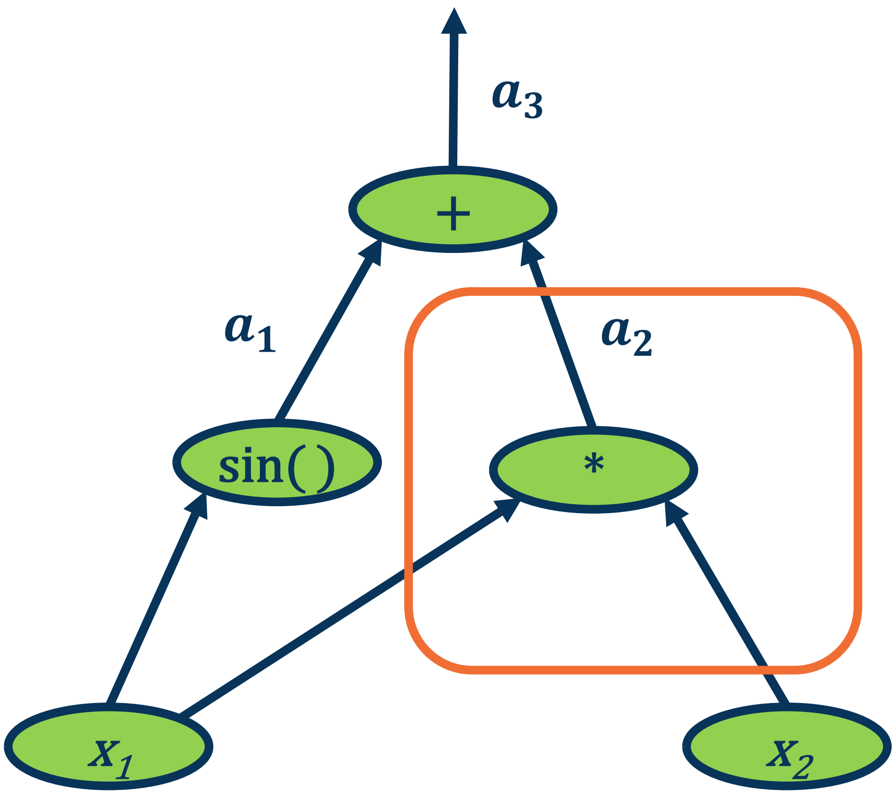


$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3}\ \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3}\ \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3}\ 1 = \overline{a_3}$$

$$\overline{a_2} = \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3}\ \frac{\partial a_3}{\partial a_2} = \overline{a_3}$$

Addition operation distributes gradients along all paths!

$$f(x_1, x_2) = x_1 x_2 + \sin(x_2)$$



Multiplication operation is a gradient switcher (multiplies it by the values of the other term)

$$\overline{x_2} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x1x2)}{\partial x_2} = \overline{a_2} x_1$$

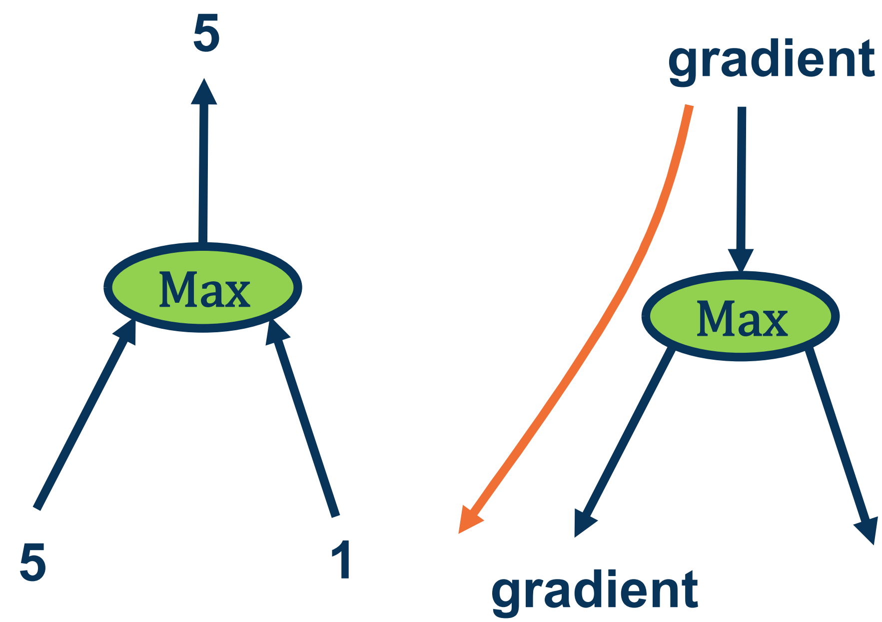
$$\overline{x_1} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = \overline{a_2} x_2$$

Patterns of Gradient Flow: Multiplication

Several other patterns as well, e.g.:

Max operation **selects** which path to push the gradients through

- Gradient flows along the path that was “selected” to be max
- This information must be recorded in the forward pass

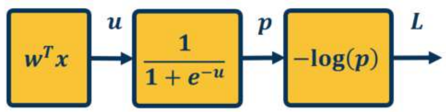


The flow of gradients is one of the **most important aspects** in deep neural networks

- If gradients **do not flow backwards properly,** learning slows or stops!

Patterns of Gradient Flow: Other

- We want to to compute: $\{\frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W}\}$



$L = 1$

$\bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p}$

where $p = \sigma(w^T x)$ and $\sigma(x) = \frac{1}{1+e^{-x}}$

$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u} = \bar{p}\,\sigma(1-\sigma)$

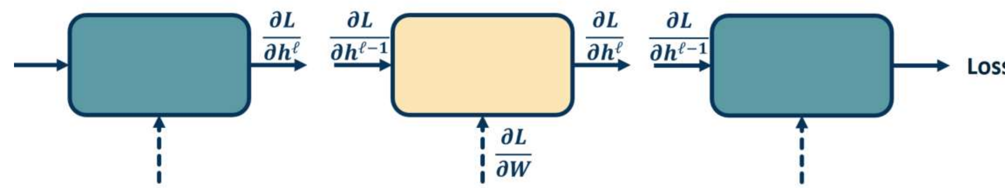
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

We can do this in a combined way to see all terms together:

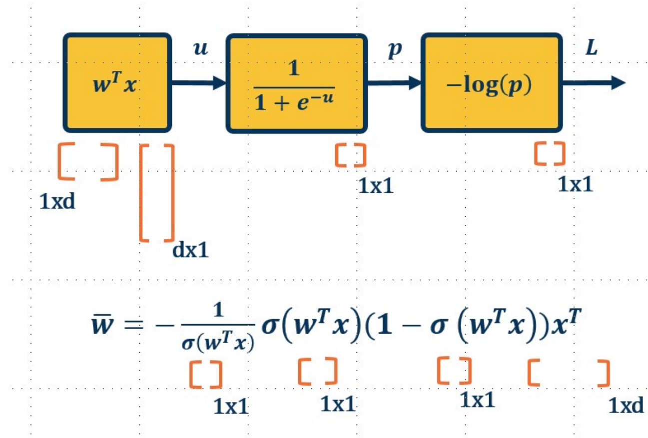
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

This effectively shows gradient flow along path from L to w

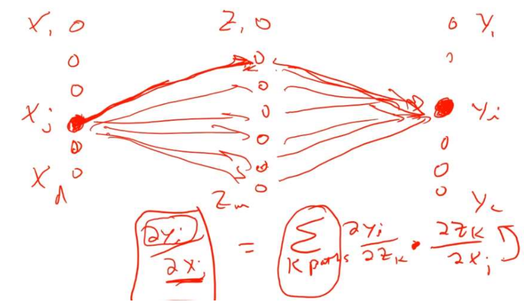
Computation Graph of primitives (automatic differentiation)



Backpropagation View (Recursive Algorithm)



Computational / Tensor View



Graph View

Different Views of Equivalent Ideas

Georgia Tech

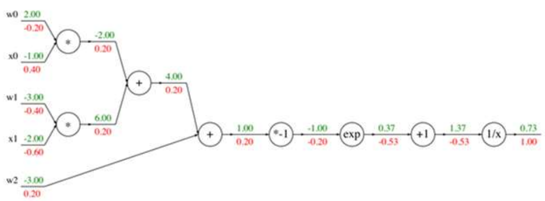
Backpropagation and Automatic Differentiation

Deep Learning = Differentiable Programming

- Computation = Graph
  - Input = Data + Parameters
  - Output = Loss
  - Scheduling = Topological ordering

- What do we need to do?
  - Generic code for representing the graph of modules
  - Specify modules (both forward and backward function)

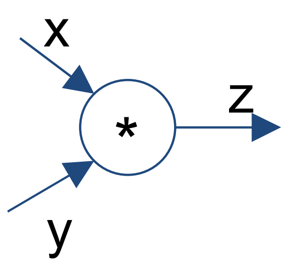
Modularized implementation: forward / backward API



Graph (or Net) object *(rough psuedo code)*

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Modularized implementation: forward / backward API



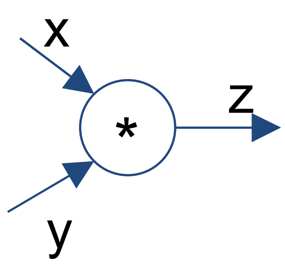
(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$$\boxed{\frac{\partial L}{\partial z}}$$

$$\boxed{\frac{\partial L}{\partial x}}$$

Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Example: Caffe layers

Branch: master ▾ caffe / src / caffe / layers / — Create new file | Upload files | Find file | History

shelhamer committed on GitHub Merge pull request #4630 from BlGene/load_hdf5_fix … Latest commit e687a71 21 days ago

..		
absval_layer.cpp	dismantle layer headers	a year ago
absval_layer.cu	dismantle layer headers	a year ago
accuracy_layer.cpp	dismantle layer headers	a year ago
argmax_layer.cpp	dismantle layer headers	a year ago
base_conv_layer.cpp	enable dilated deconvolution	a year ago
base_data_layer.cpp	Using default from proto for prefetch	3 months ago
base_data_layer.cu	Switched multi-GPU to NCCL	3 months ago
batch_norm_layer.cpp	Add missing spaces besides equal signs in batch_norm_layer.cpp	4 months ago
batch_norm_layer.cu	dismantle layer headers	a year ago
batch_reindex_layer.cpp	dismantle layer headers	a year ago
batch_reindex_layer.cu	dismantle layer headers	a year ago
bias_layer.cpp	Remove incorrect cast of gemm int arg to Dtype in BiasLayer	a year ago
bias_layer.cu	Separation and generalization of ChannelwiseAffineLayer into BiasLayer	a year ago
bnll_layer.cpp	dismantle layer headers	a year ago
bnll_layer.cu	dismantle layer headers	a year ago
concat_layer.cpp	dismantle layer headers	a year ago
concat_layer.cu	dismantle layer headers	a year ago
contrastive_loss_layer.cpp	dismantle layer headers	a year ago
contrastive_loss_layer.cu	dismantle layer headers	a year ago
conv_layer.cpp	add support for 2D dilated convolution	a year ago
conv_layer.cu	dismantle layer headers	a year ago
crop_layer.cpp	remove redundant operations in Crop layer (#5138)	2 months ago
crop_layer.cu	remove redundant operations in Crop layer (#5138)	2 months ago
cudnn_conv_layer.cpp	dismantle layer headers	a year ago
cudnn_conv_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago

cudnn_lcn_layer.cpp	dismantle layer headers	a year ago
cudnn_lcn_layer.cu	dismantle layer headers	a year ago
cudnn_lrn_layer.cpp	dismantle layer headers	a year ago
cudnn_lrn_layer.cu	dismantle layer headers	a year ago
cudnn_pooling_layer.cpp	dismantle layer headers	a year ago
cudnn_pooling_layer.cu	dismantle layer headers	a year ago
cudnn_relu_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_relu_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_sigmoid_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_sigmoid_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_softmax_layer.cpp	dismantle layer headers	a year ago
cudnn_softmax_layer.cu	dismantle layer headers	a year ago
cudnn_tanh_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_tanh_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
data_layer.cpp	Switched multi-GPU to NCCL	3 months ago
deconv_layer.cpp	enable dilated deconvolution	a year ago
deconv_layer.cu	dismantle layer headers	a year ago
dropout_layer.cpp	supporting N-D Blobs in Dropout layer Reshape	a year ago
dropout_layer.cu	dismantle layer headers	a year ago
dummy_data_layer.cpp	dismantle layer headers	a year ago
eltwise_layer.cpp	dismantle layer headers	a year ago
eltwise_layer.cu	dismantle layer headers	a year ago
elu_layer.cpp	ELU layer with basic tests	a year ago
elu_layer.cu	ELU layer with basic tests	a year ago
embed_layer.cpp	dismantle layer headers	a year ago
embed_layer.cu	dismantle layer headers	a year ago
euclidean_loss_layer.cpp	dismantle layer headers	a year ago
euclidean_loss_layer.cu	dismantle layer headers	a year ago
exp_layer.cpp	Solving issue with exp layer with base e	a year ago
exp_layer.cu	dismantle layer headers	a year ago

Caffe Sigmoid Layer

```cpp
#include <cmath>
#include <vector>

#include "caffe/layers/sigmoid_layer.hpp"

namespace caffe {

template <typename Dtype>
inline Dtype sigmoid(Dtype x) {
  return 1. / (1. + exp(-x));
}

template <typename Dtype>
void SigmoidLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  const int count = bottom[0]->count();
  for (int i = 0; i < count; ++i) {
    top_data[i] = sigmoid(bottom_data[i]);
  }
}

template <typename Dtype>
void SigmoidLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
  if (propagate_down[0]) {
    const Dtype* top_data = top[0]->cpu_data();
    const Dtype* top_diff = top[0]->cpu_diff();
    Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
    const int count = bottom[0]->count();
    for (int i = 0; i < count; ++i) {
      const Dtype sigmoid_x = top_data[i];
      bottom_diff[i] = top_diff[i] * sigmoid_x * (1. - sigmoid_x);
    }
  }
}

#ifdef CPU_ONLY
STUB_GPU(SigmoidLayer);
#endif

INSTANTIATE_CLASS(SigmoidLayer);


}  // namespace caffe
```


$$\boxed{\sigma(x) = \frac{1}{1+e^{-x}}}$$

$\boxed{(1-\sigma(x))\,\sigma(x)}$ * top_diff (chain rule)

Backpropagation does not really spell out how to **efficiently** carry out the necessary computations

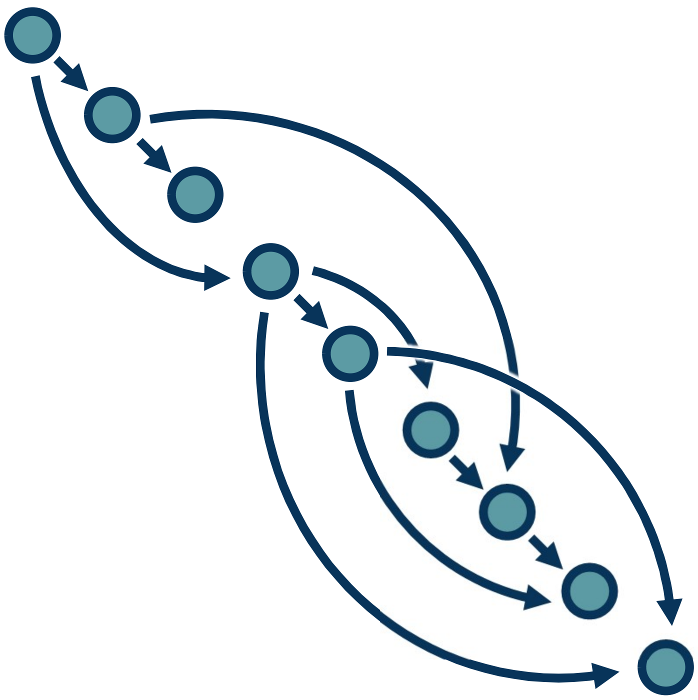
But the idea can be applied to **any directed acyclic graph (DAG)**

- Graph represents an **ordering constraining** which paths must be calculated first

Given an ordering, we can then iterate from the last module backwards, **applying the chain rule**

- We will store, for each node, its **gradient outputs for efficient computation**
- We will do this **automatically** by computing backwards function for primitives and as you write code, express the function with them

This is called reverse-mode **automatic differentiation**



A General Framework

Computation = Graph

- Input = Data + Parameters
- Output = Loss
- Scheduling = Topological ordering

Auto-Diff

- A family of algorithms for implementing chain-rule on computation graphs

$$w^T x \xrightarrow{u} \frac{1}{1+e^{-u}} \xrightarrow{p} -\log(p) \xrightarrow{L}$$

Automatic differentiation:

- Carries out this procedure for us on arbitrary graphs
- Knows derivatives of primitive functions
- As a result, we just define these (forward) functions **and don't even need to specify the gradient (backward) functions!**

$$\bar{L} = 1$$
$$\bar{p} = \frac{\partial L}{\partial p} = -\frac{1}{p}$$

where $p = \sigma(w^T x)$ **and** $\sigma(x) = \frac{1}{1+e^{-x}}$

$$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \bar{p}\, \sigma(1-\sigma)$$

$$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u} \frac{\partial u}{\partial w} = \bar{u} x^T$$

We can do this in a combined way to see all terms together:

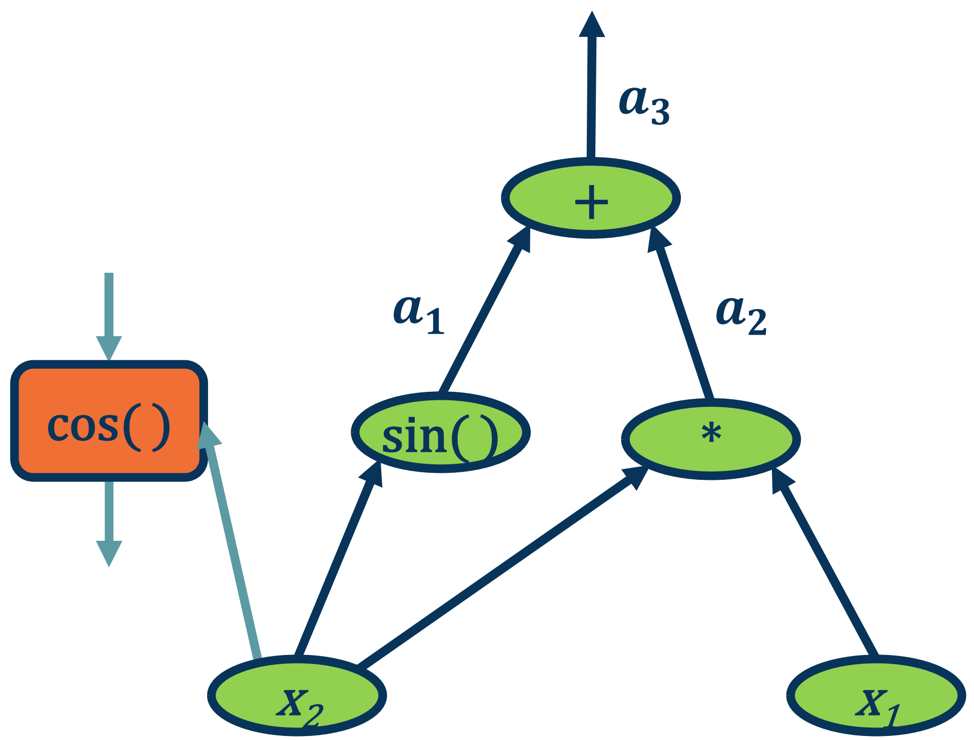
$$\begin{aligned} \bar{w} &= \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)} \sigma(w^T x)(1-\sigma(w^T x)) x^T \\ &= -\left(1-\sigma(w^T x)\right) x^T \end{aligned}$$

This effectively shows gradient flow along path from L **to** w

Example Gradient Computations

- Key idea is to **explicitly store computation graph** in memory and **corresponding gradient functions**
- **Nodes** broken down to **basic primitive computations** (addition, multiplication, log, etc.) for which **corresponding derivative is known**

$$\overline{x_2} = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = \overline{a_1} \quad \cos(x_2)$$

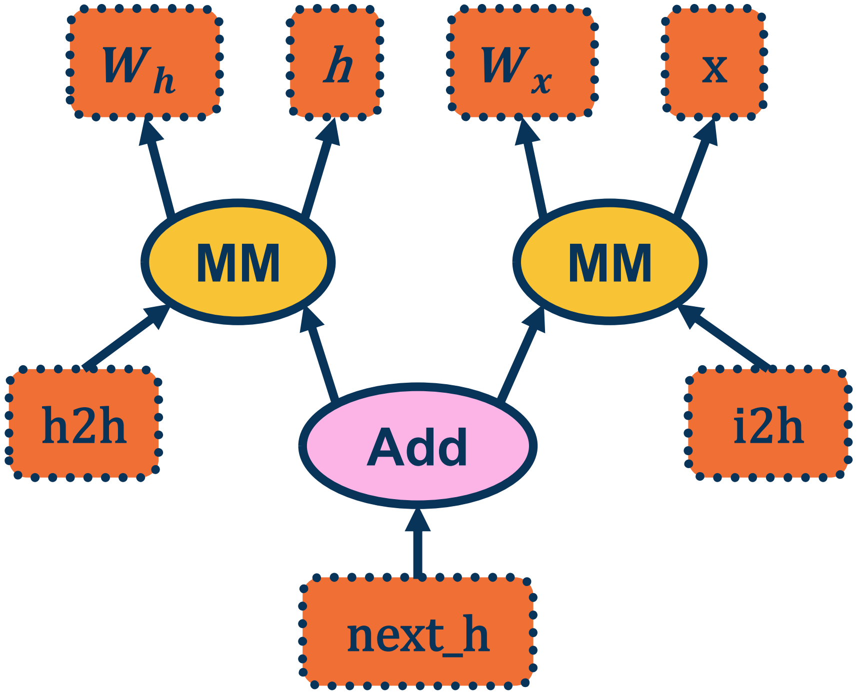


A graph is created on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



(Note above)

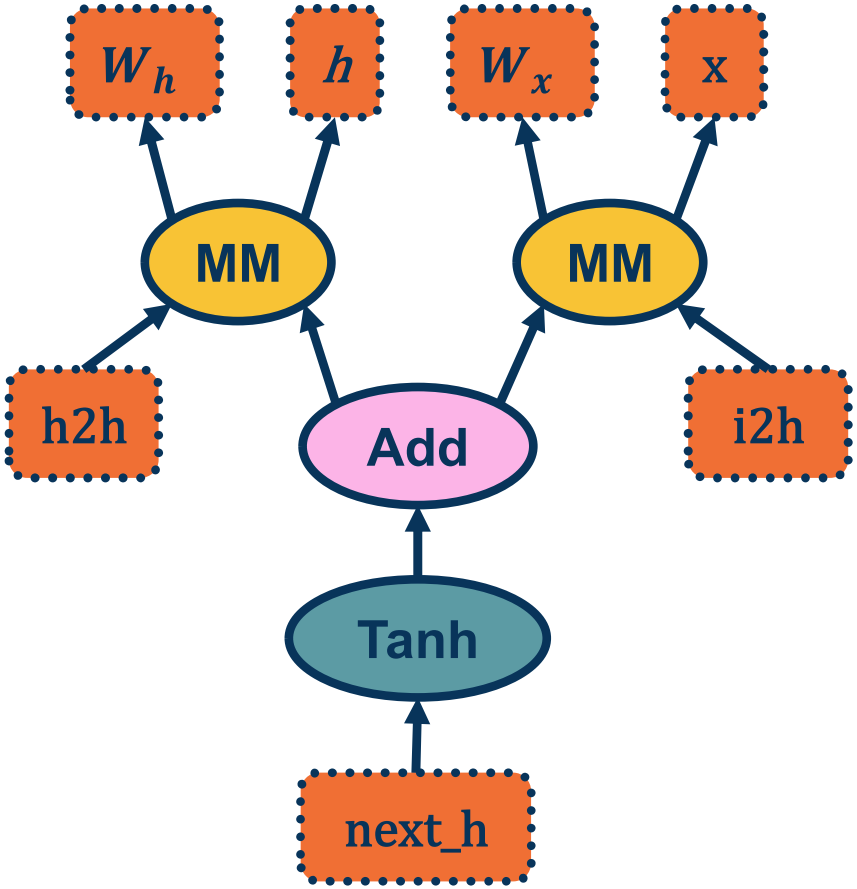
Back-propagation uses the dynamically built graph

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



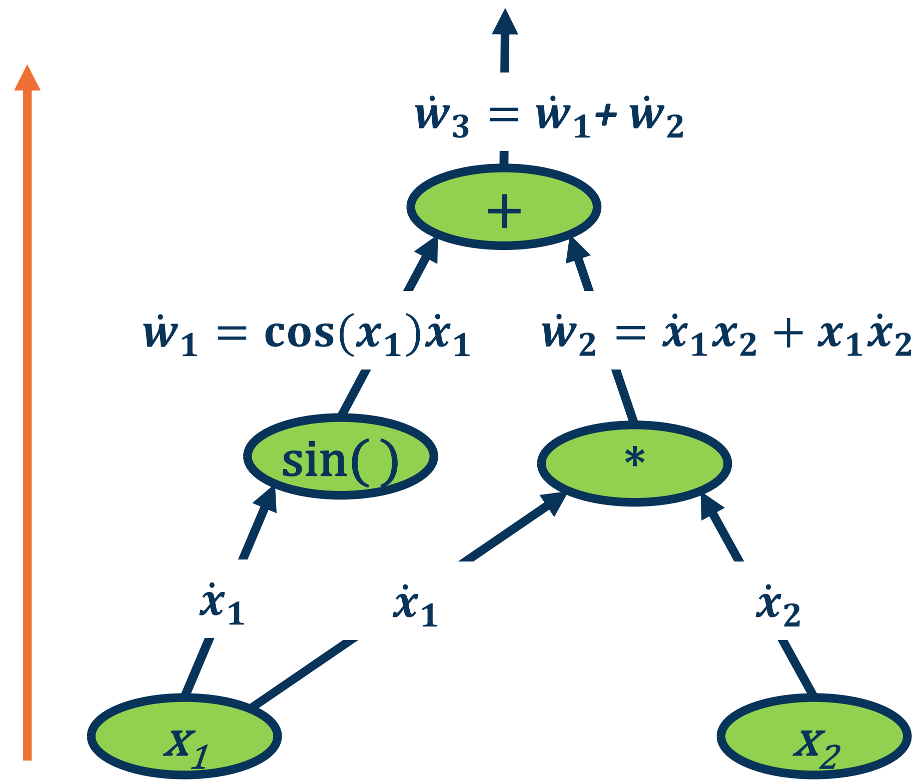
From pytorch.org

Note that we can also do **forward mode** automatic differentiation

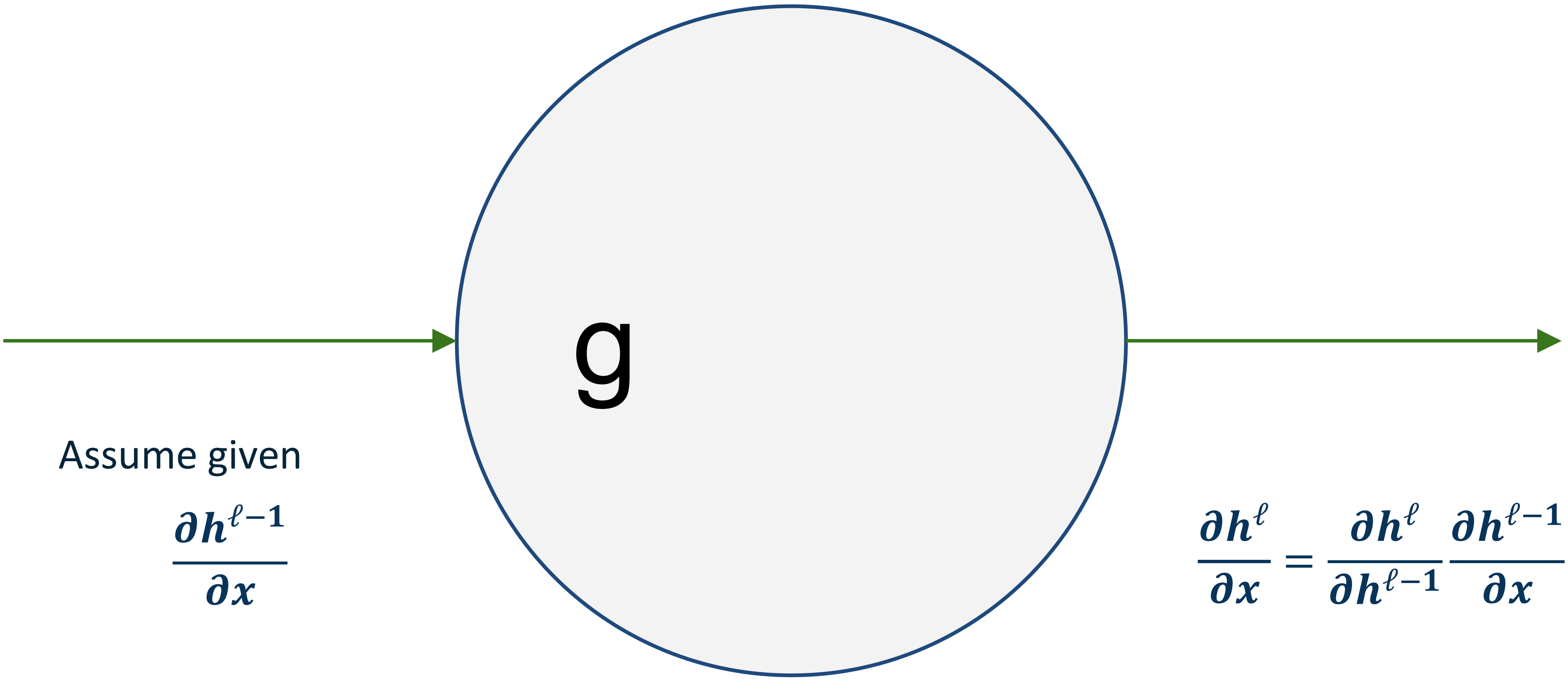
Start from **inputs** and propagate gradients forward

Complexity is proportional to input size

- Memory savings (all forward pass, no need to store activations)
- However, in most cases our **inputs** (images) are large and **outputs** (loss) are small



Automatic Differentiation



Assume given

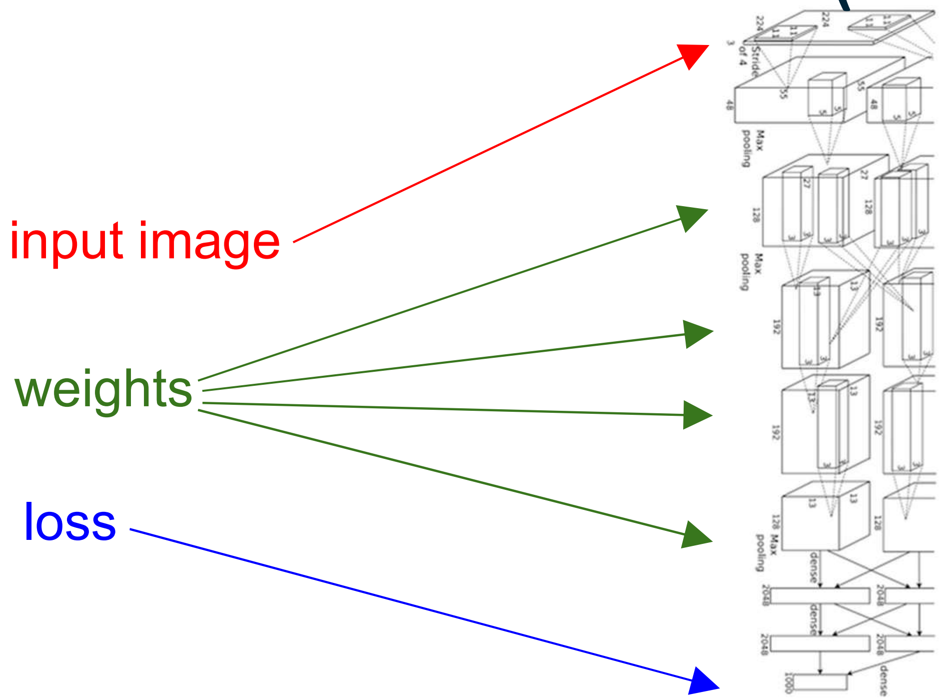
$$\frac{\partial h^{\ell-1}}{\partial x}$$

$$\frac{\partial h^{\ell}}{\partial x} = \frac{\partial h^{\ell}}{\partial h^{\ell-1}} \frac{\partial h^{\ell-1}}{\partial x}$$

See https://www.cc.gatech.edu/classes/AY2020/cs7643_spring/slides/autodiff_forward_reverse.pdf

Forward Mode Autodifferentiation

Convolutional network (AlexNet)



Neural Turing Machine

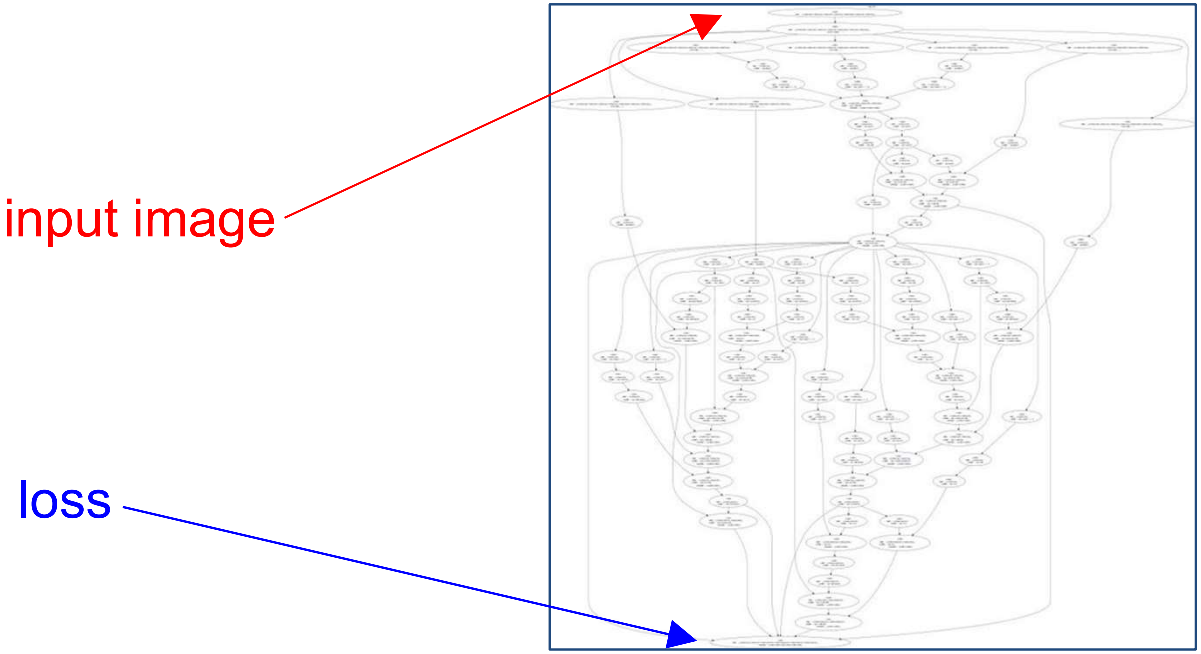
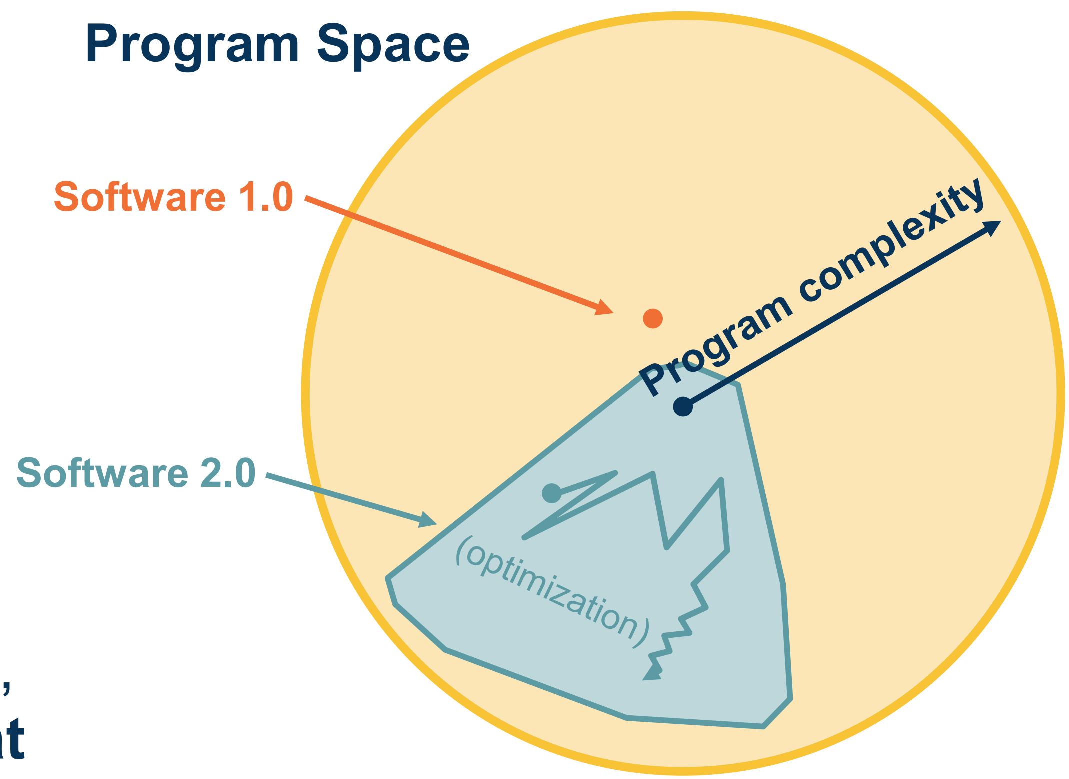


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

- Computation graphs are **not limited to mathematical functions!**
- Can have **control flows** (if statements, loops) and **backpropagate** through **algorithms**!
- Can be done **dynamically** so that **gradients are computed**, then **nodes are added, repeat**
- **Differentiable programming**



Adapted from figure by Andrej Karpathy

Power of Automatic Differentiation