Topics:

- Backpropagation / Automatic Differentiation
- Jacobians

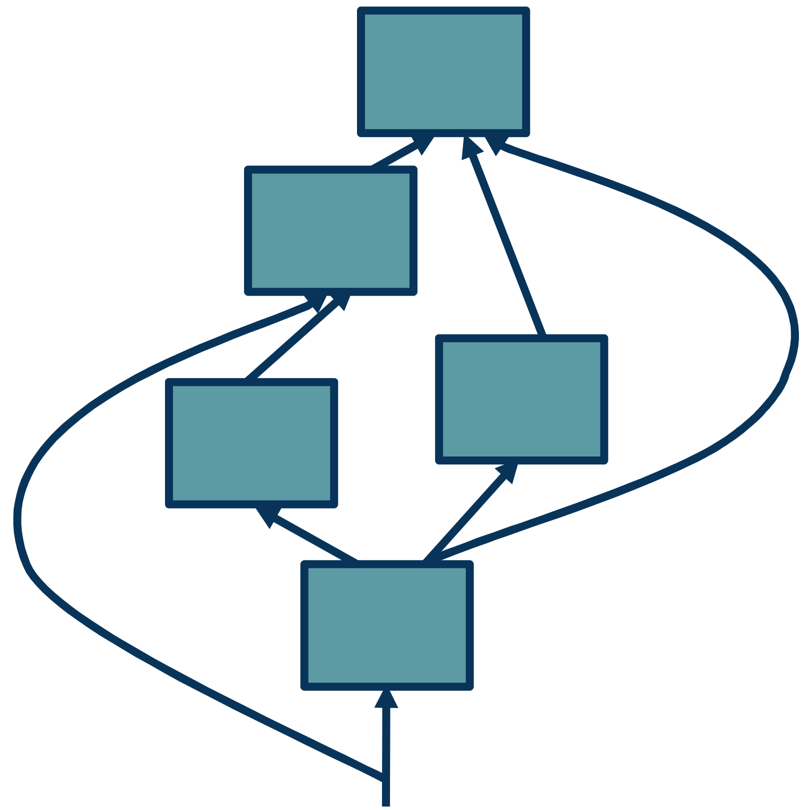
CS 4803-DL / 7643-A
ZSOLT KIRA

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**

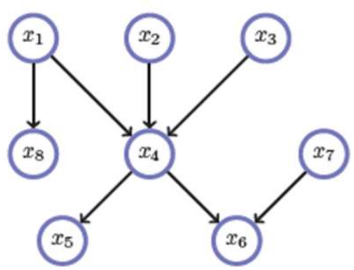


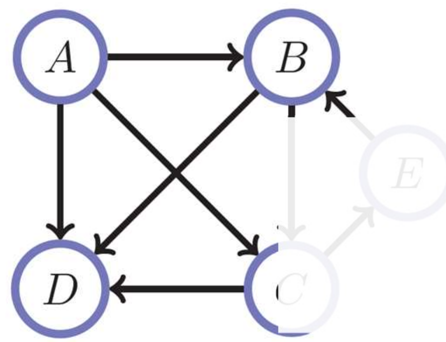
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

A General Framework

Directed Acyclic Graphs (DAGs)

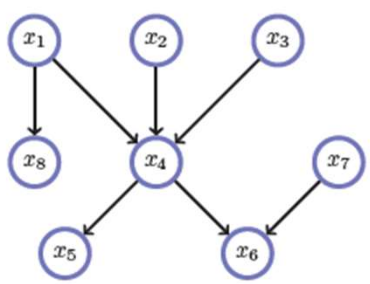
- Exactly what the name suggests
  - Directed edges
  - No (directed) cycles
  - Underlying undirected cycles okay

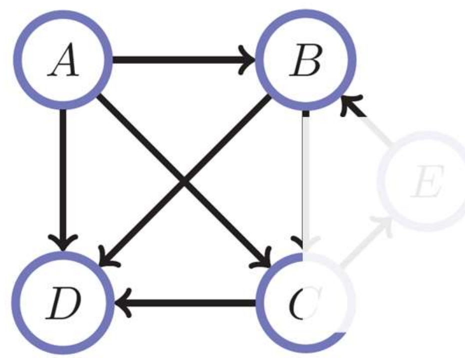




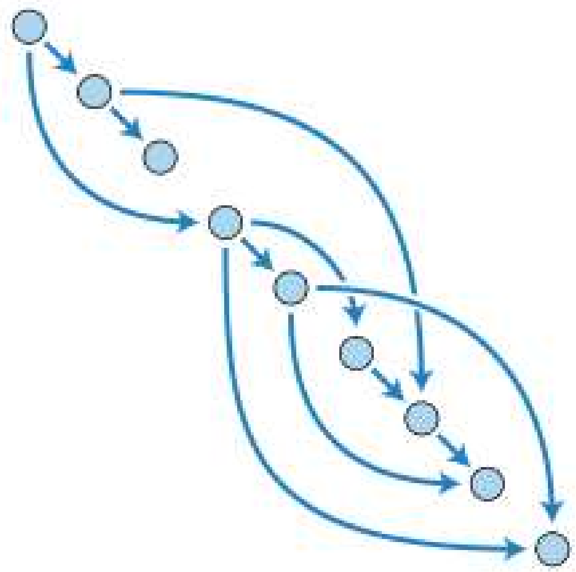
Directed Acyclic Graphs (DAGs)

- Concept
  - Topological Ordering

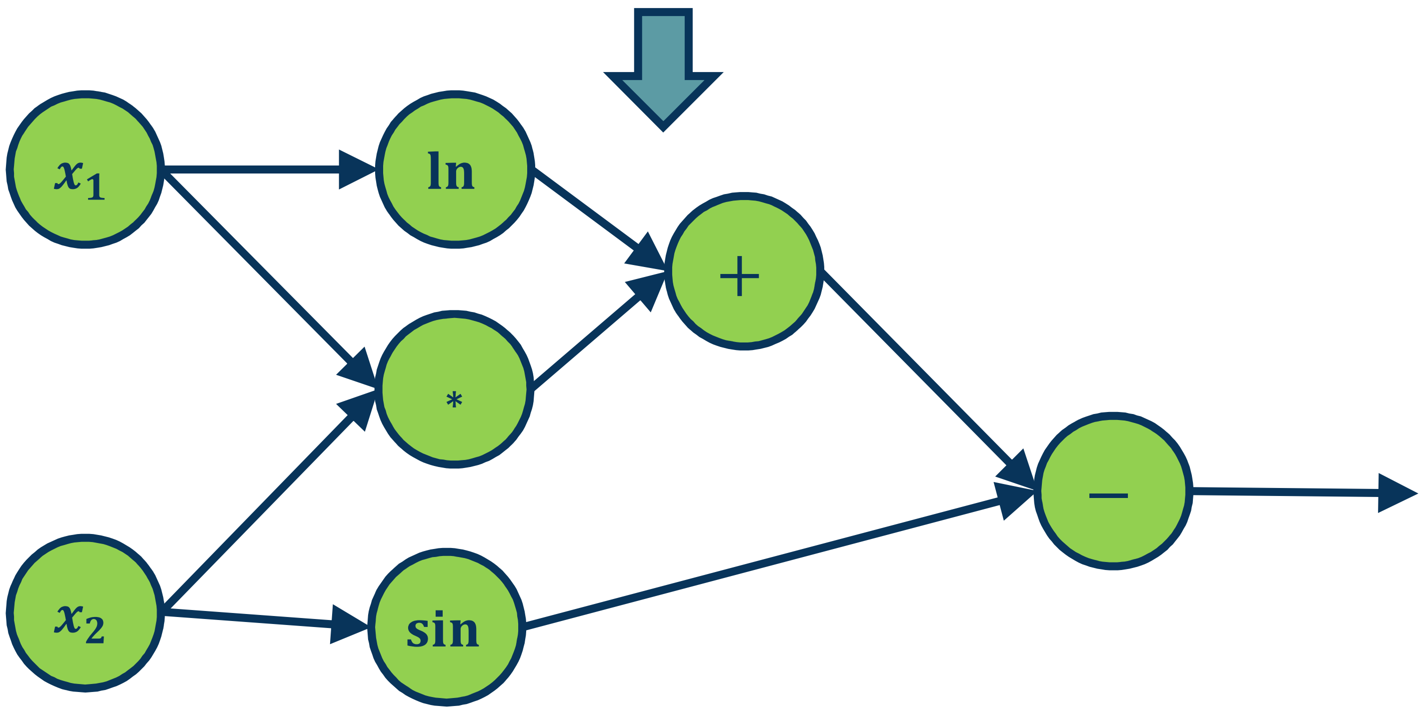




Directed Acyclic Graphs (DAGs)

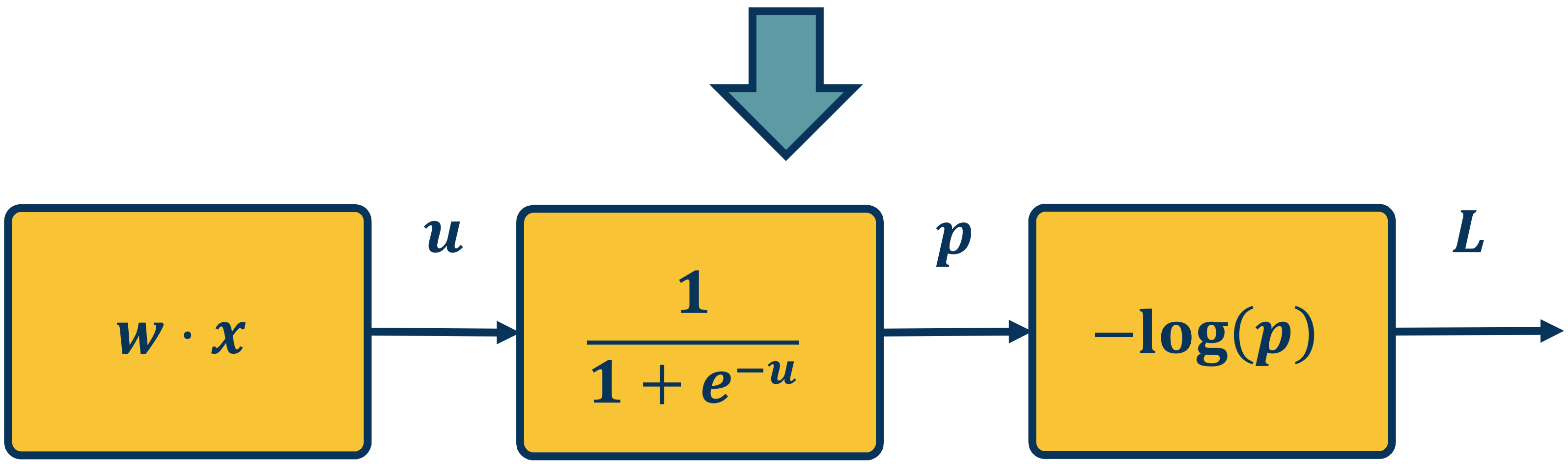


$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$



Example

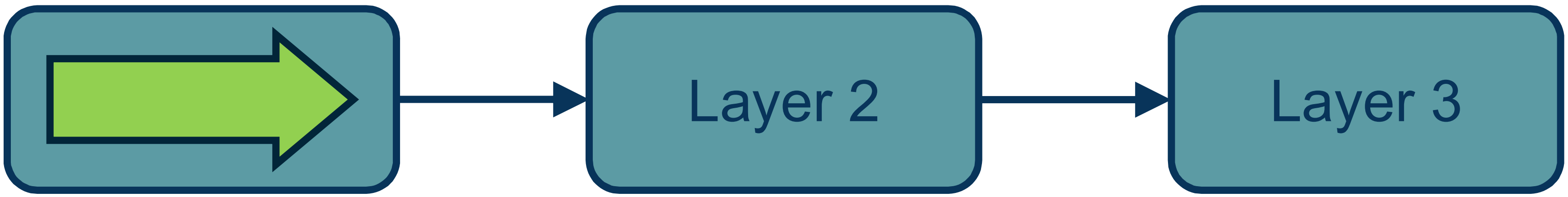
$$-\log\left(\frac{1}{1+e^{-w\cdot x}}\right)$$



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

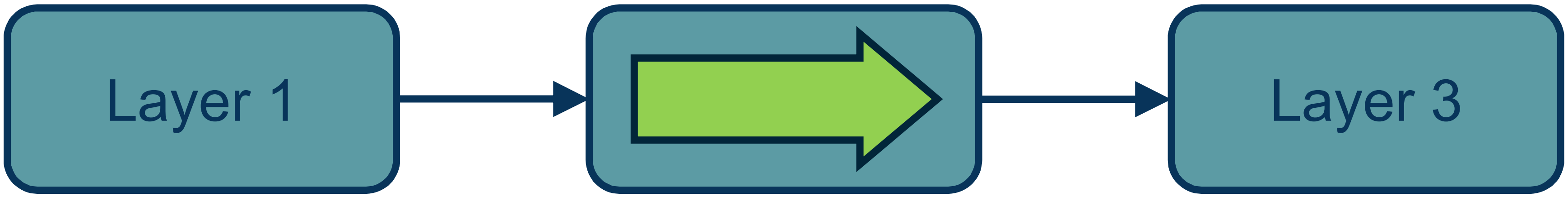
Backpropagation

Step 1: Compute Loss on Mini-Batch: **Forward Pass**



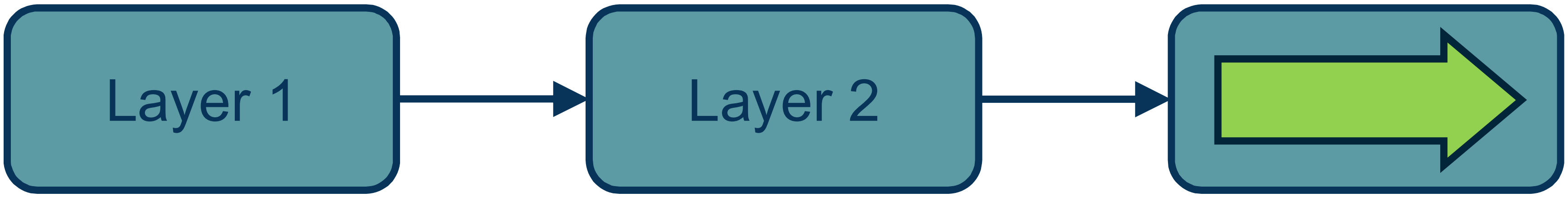
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Step 1: Compute Loss on Mini-Batch: **Forward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Step 1: Compute Loss on Mini-Batch: **Forward Pass**



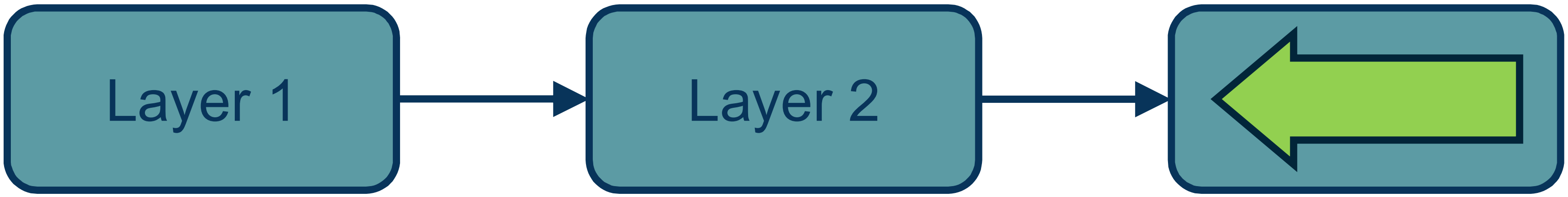
Note that we must store the **intermediate outputs of all layers**!

- This is because we will need them to **compute the gradients** (the gradient equations will have terms with the output values in them)

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Step 1: Compute Loss on Mini-Batch: **Forward Pass**

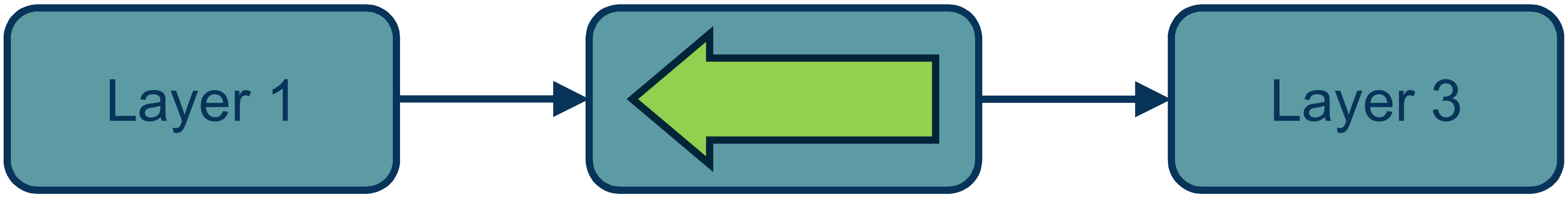
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Step 1: Compute Loss on Mini-Batch: **Forward Pass**

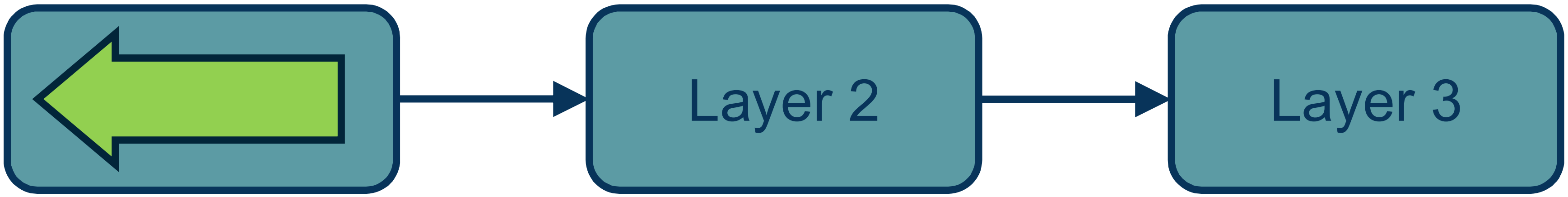
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

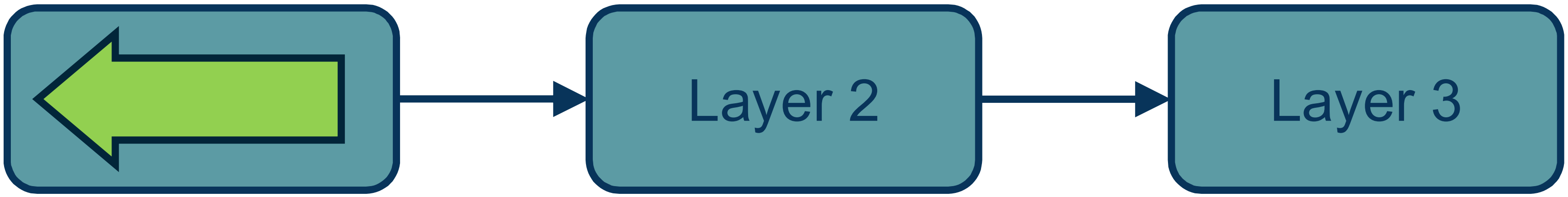


Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Step 3: Use **gradient** to update **all parameters** at the end

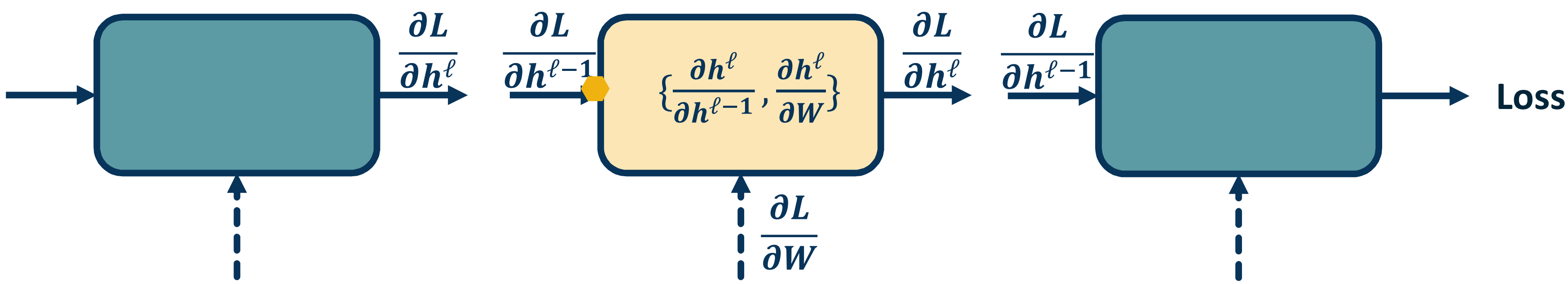


$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

Backpropagation is the application of gradient descent to a computation graph via the chain rule!

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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



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

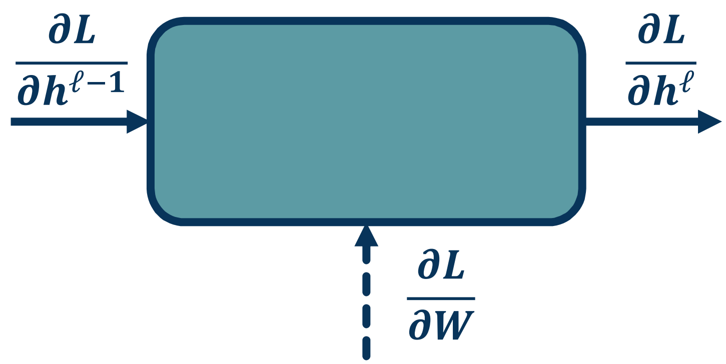
Chain Rule: $$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Computing the Gradients of Loss

- We will use the **chain rule** to compute: $\{\frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W}\}$
- **Gradient of loss w.r.t. inputs:** $\frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial h^{\ell-1}}$
- **Gradient of loss w.r.t. weights:** $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$

Given by upstream module **(upstream gradient)**

Calculated Analytically



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Computing the Gradients of Loss

Backpropagation: a simple example

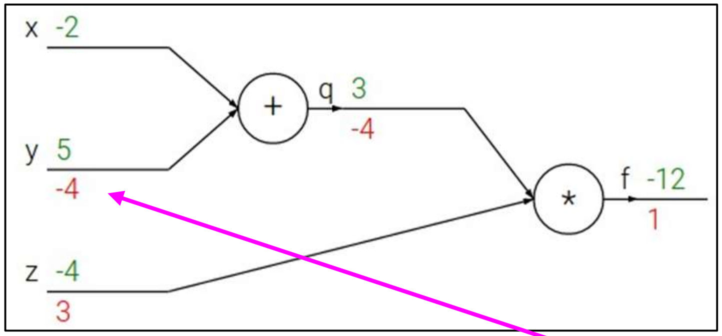
$$f(x,y,z) = (x+y)z$$

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



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

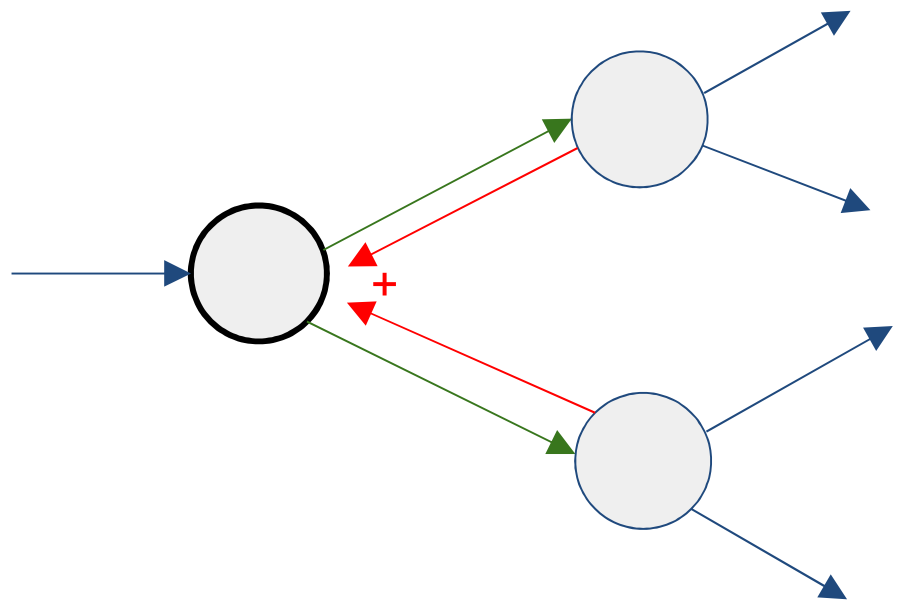
Chain rule:

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

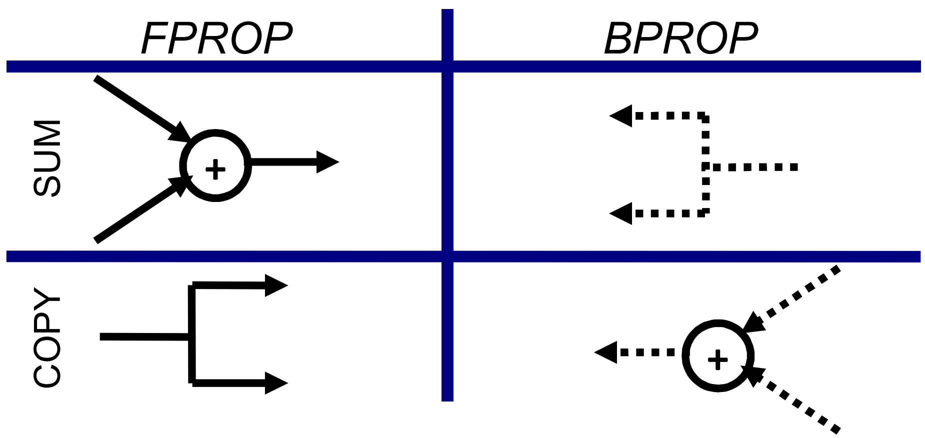
Upstream gradient

Local gradient

Gradients add at branches



Duality in Fprop and Bprop



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)

Linear Algebra View: Vector and Matrix Sizes

Georgia Tech

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b2 \\ w_{31} & w_{32} & \cdots & w_{3m} & b3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{bmatrix}$$

$\boldsymbol{W}$ $\boldsymbol{x}$

Sizes: $[\boldsymbol{c} \times (\boldsymbol{d}+\mathbf{1})]$ $[(\boldsymbol{d}+\mathbf{1}) \times \mathbf{1}]$

Where $\boldsymbol{c}$ is number of classes

$\boldsymbol{d}$ is dimensionality of input

Closer Look at a Linear Classifier

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. $\boldsymbol{v} = [v_1, v_2, \ldots, v_m]^T$
  and matrix $\boldsymbol{M} \in \mathbb{R}^{k \times \ell}$

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

Conventions:

- Size of derivatives for scalars, vectors, and matrices:
  Assume we have scalar $\boldsymbol{s} \in \mathbb{R}^{\mathbf{1}}$, vector $\boldsymbol{v} \in \mathbb{R}^{\boldsymbol{m}}$, i.e. $\boldsymbol{v} = [\boldsymbol{v}_{\mathbf{1}}, \boldsymbol{v}_{\mathbf{2}}, \ldots, \boldsymbol{v}_{\boldsymbol{m}}]^{\boldsymbol{T}}$
  and matrix $\boldsymbol{M} \in \mathbb{R}^{\boldsymbol{k}\times\boldsymbol{\ell}}$

- What is the size of $\frac{\partial v}{\partial s}$? $\mathbb{R}^{m\times 1}$ (column vector of size m)

$$\begin{bmatrix} \frac{\partial v_1}{\partial s} \\ \frac{\partial v_2}{\partial s} \\ \vdots \\ \frac{\partial v_m}{\partial s} \end{bmatrix}$$

- What is the size of $\frac{\partial s}{\partial v}$? $\mathbb{R}^{1\times m}$ (row vector of size m)

$$\begin{bmatrix} \frac{\partial s}{\partial v_1} & \frac{\partial s}{\partial v_1} & \cdots & \frac{\partial s}{\partial v_m} \end{bmatrix}$$

Conventions:

- What is the size of $\frac{\partial v^1}{\partial v^2}$? A matrix:

$$
\begin{array}{cc}
 & \text{Col } j \\
\text{Row } i & \begin{bmatrix}
\frac{\partial v_1^1}{\partial v_1^2} & \cdots & \cdots & \cdots & \cdots & \\
\cdots & & \cdots & \cdots & \cdots & \cdots \\
\frac{\partial v_i^1}{\partial v_1^2} & \cdots & \frac{\partial v_i^1}{\partial v_j^2} & \cdots & \frac{\partial v_i^1}{\partial v_{m_2}^2} & \\
\cdots & \cdots & \cdots & \cdots & \cdots & \\
\cdots & \cdots & \cdots & \cdots & \cdots &
\end{bmatrix}_{m_1 \times m_2}
\end{array}
$$

- This matrix of partial derivatives is called a **Jacobian**

(Note this is slightly different convention than on Wikipedia). Also, computationally other conventions are used.

Dimensionality of Derivatives

Conventions:

- What is the size of $\frac{\partial s}{\partial M}$? A matrix:

$$\begin{bmatrix} \frac{\partial s}{\partial m_{[1,1]}} & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \frac{\partial s}{\partial m_{[i,j]}} & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix}$$

Example 1:

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x \\ x^2 \end{bmatrix} \qquad \frac{\partial y}{\partial x} = \begin{bmatrix} 1 \\ 2x \end{bmatrix}$$

Example 2:

$$y = w^T x = \sum_k w_k x_k$$

$$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x_1}, \dots, \frac{\partial y}{\partial x_m}\right]$$

$$= [w_1, \dots, w_m]$$ because $$\frac{\partial(\sum_k w_k x_k)}{\partial x_i} = w_i$$

$$= w^T$$

Example 3:

$\frac{\partial(wAw)}{\partial w} = 2w^T A$ (assuming A is symmetric)

Example 4:

$$y = Wx \qquad\qquad \frac{\partial y}{\partial x} = W$$

$$\begin{array}{cc} & \begin{array}{ccccc} & & \textbf{Col } j & & \end{array} \\ \textbf{Row } i & \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \frac{\partial y_i}{\partial x_j} & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix} \end{array} = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & w_{ij} & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix} \qquad y_i = \sum_j w_{ij} x_j$$

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

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

Jacobian is also a matrix:

$$W$$

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

Dimensionality of Derivatives in ML

Jacobians of Batches

Batches of data are **matrices** or **tensors** (multi-dimensional matrices)

Examples:

- Each instance is a vector of size m, our batch is of size $[\boldsymbol{B} \times \boldsymbol{m}]$
- Each instance is a matrix (e.g. grayscale image) of size $\boldsymbol{W} \times \boldsymbol{H}$, our batch is $[\boldsymbol{B} \times \boldsymbol{W} \times \boldsymbol{H}]$
- Each instance is a multi-channel matrix (e.g. color image with R,B,G channels) of size $\boldsymbol{C} \times \boldsymbol{W} \times \boldsymbol{H}$, our batch is $[\boldsymbol{B} \times \boldsymbol{C} \times \boldsymbol{W} \times \boldsymbol{H}]$

Jacobians become tensors which is complicated

- Instead, flatten input to a vector and get a vector of derivatives!
- This can also be done for partial derivatives between two vectors, two matrices, or two tensors

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix}$$

Flatten

$$\begin{bmatrix} x_{11} \\ x_{12} \\ \vdots \\ x_{21} \\ x_{22} \\ \vdots \\ x_{n1} \\ \vdots \\ x_{nn} \end{bmatrix}$$

Vectorization and Jacobians of Simple Layers

Composition of Functions: $\boldsymbol{f}\left(\boldsymbol{g}(\boldsymbol{x})\right) = (\boldsymbol{f} \circ \boldsymbol{g})(\boldsymbol{x})$

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

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}_{\ell}\left(\boldsymbol{g}_{\ell-1}\left(\ldots \boldsymbol{g}_{1}(\boldsymbol{x})\right)\right)$$

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}_{\ell} \circ \boldsymbol{g}_{\ell-1} \ldots \circ \boldsymbol{g}_{1}$$

(Many of these will be parameterized)

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

Scalar Case

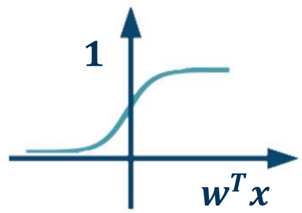
Vector Case

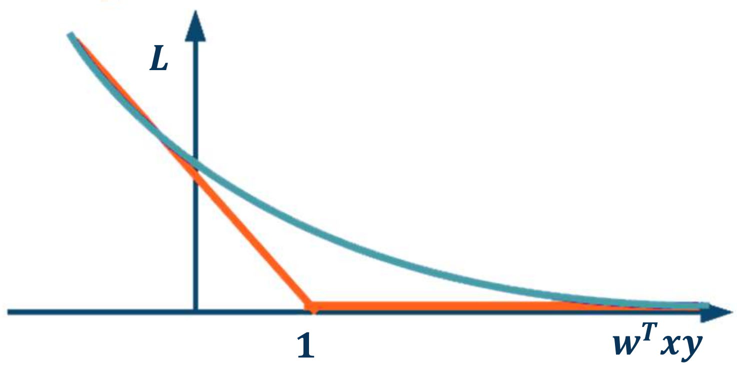
Jacobian View of Chain Rule

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: $\boldsymbol{L} = \frac{\mathbf{1}}{\mathbf{2}}||\boldsymbol{w}||^{2} - \boldsymbol{\lambda}\mathbf{log}\big(\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x})\big)$





Log Loss

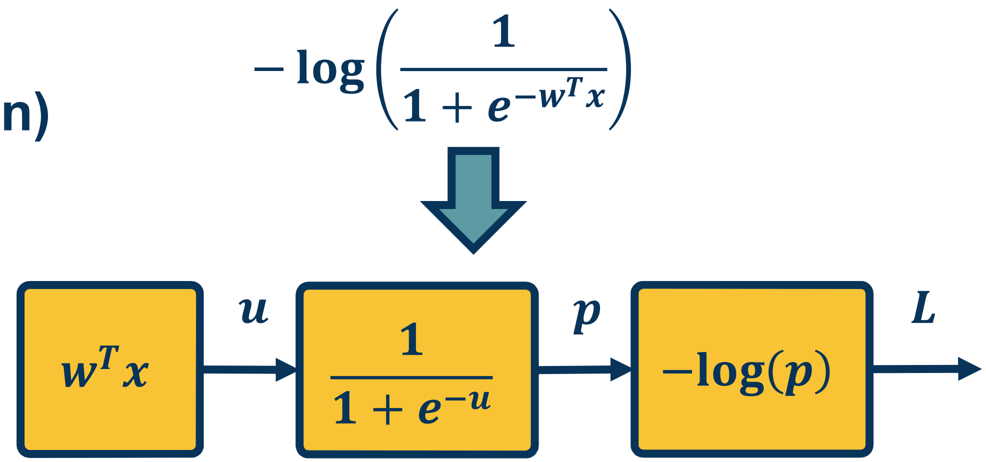
Adapted from slide by Marc'Aurelio Ranzato

Linear Classifier: Logistic Regression

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!



Neural Network Computation Graph

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

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:

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

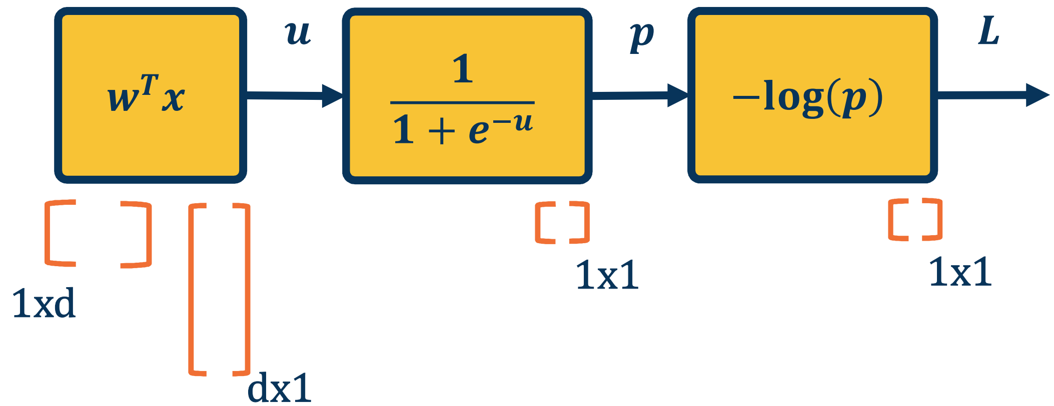
This effectively shows gradient flow along path from L **to** w

Example Gradient Computations

Georgia Tech

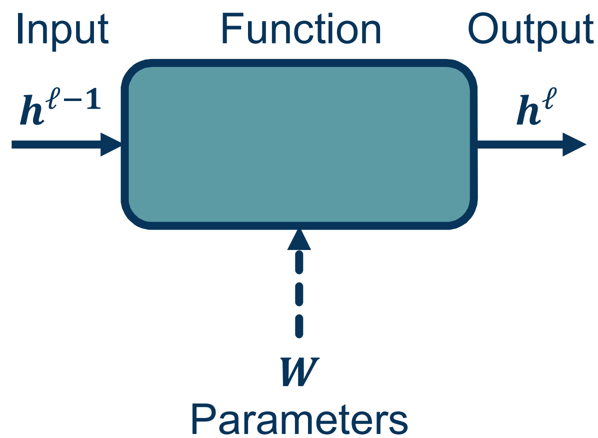
Vectorized Computations

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



Extremely efficient in graphics processing units (GPUs)

$$\bar{w} = \underbrace{-\frac{1}{\sigma(w^T x)}}_{1x1}\underbrace{\sigma(w^T x)}_{1x1}\underbrace{(1-\sigma(w^T x))}_{1x1}\underbrace{x^T}_{1xd}$$



Define:

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

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

$\leftarrow w_i^T \rightarrow$

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

Fully Connected (FC) Layer: Forward Function

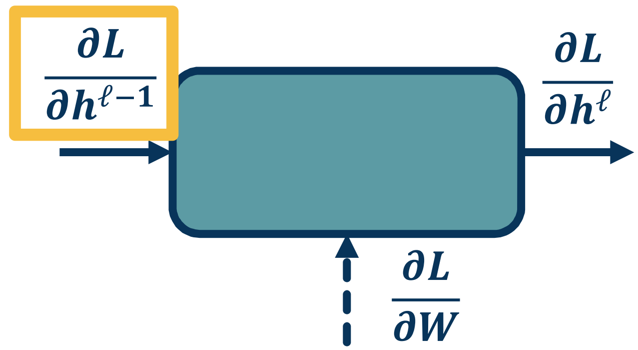
$$h^{\ell} = W h^{\ell-1}$$

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

Define:

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

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

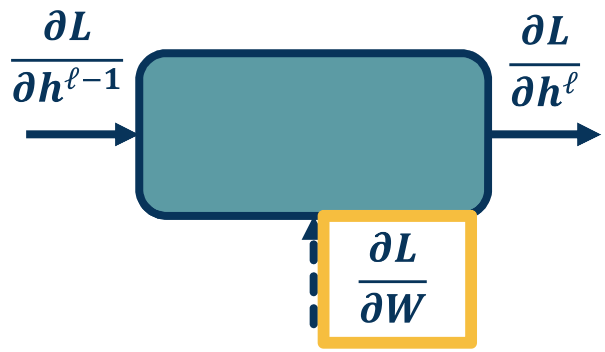


$$\boxed{\frac{\partial L}{\partial h^{\ell-1}}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial h^{\ell-1}}$$

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

Fully Connected (FC) Layer

Georgia Tech



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

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

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

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

Define:

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

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

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

$$\underbrace{\begin{bmatrix} \quad \end{bmatrix}}_{1 \times |h^{\ell-1}|} \underbrace{\begin{bmatrix} \quad \end{bmatrix}}_{1 \times |h^{\ell}|} \underbrace{\begin{bmatrix} \leftarrow & 0 & \rightarrow \\ \leftarrow & \frac{\partial h_i^{\ell}}{\partial w_i} & \rightarrow \\ \leftarrow & 0 & \rightarrow \end{bmatrix}}_{|h^{\ell}| \times |h^{\ell-1}|}$$

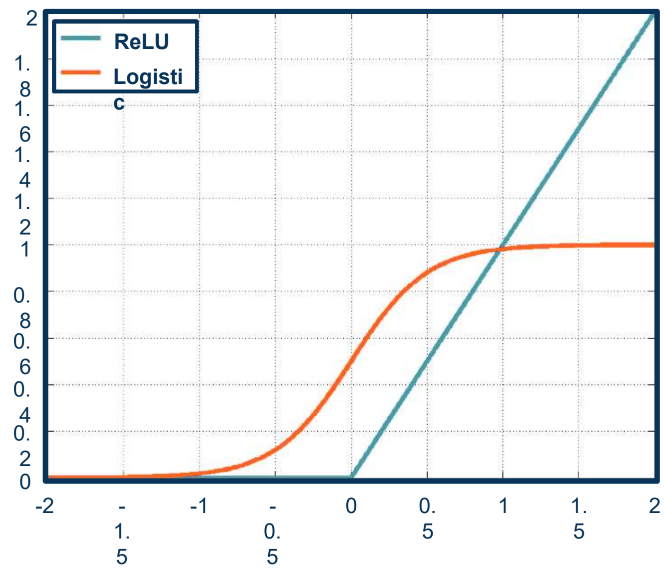
Fully Connected (FC) Layer

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

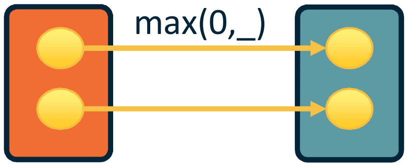
A common choice is the **Rectified Linear Unit**

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

How many parameters for this layer?



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



Rectified Linear Unit (ReLU)

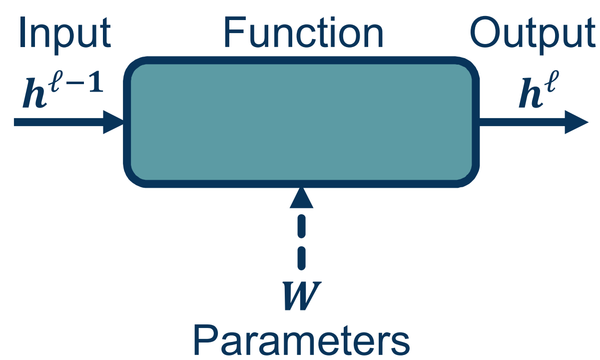
Jacobian of ReLU

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**

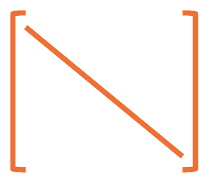
Max function **funnels gradients through selected max**

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



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

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



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

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

Backpropagation and Automatic Differentiation

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

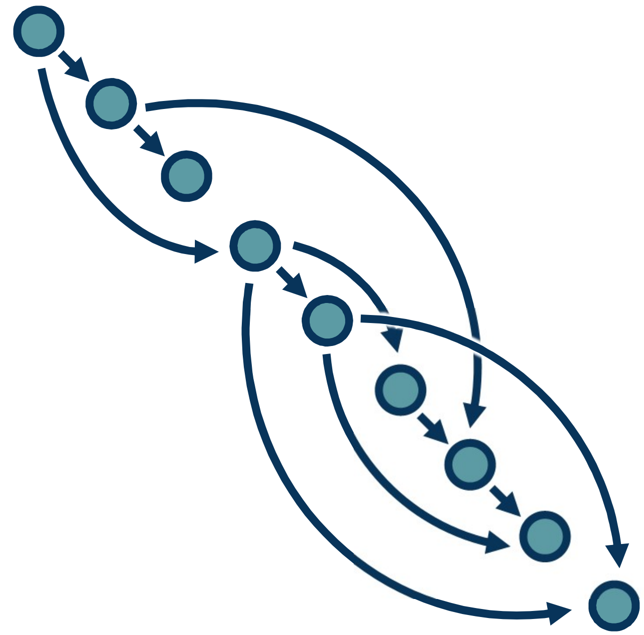
But the idea can be applied to **any directed acyclic graph (DAG)**

- 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

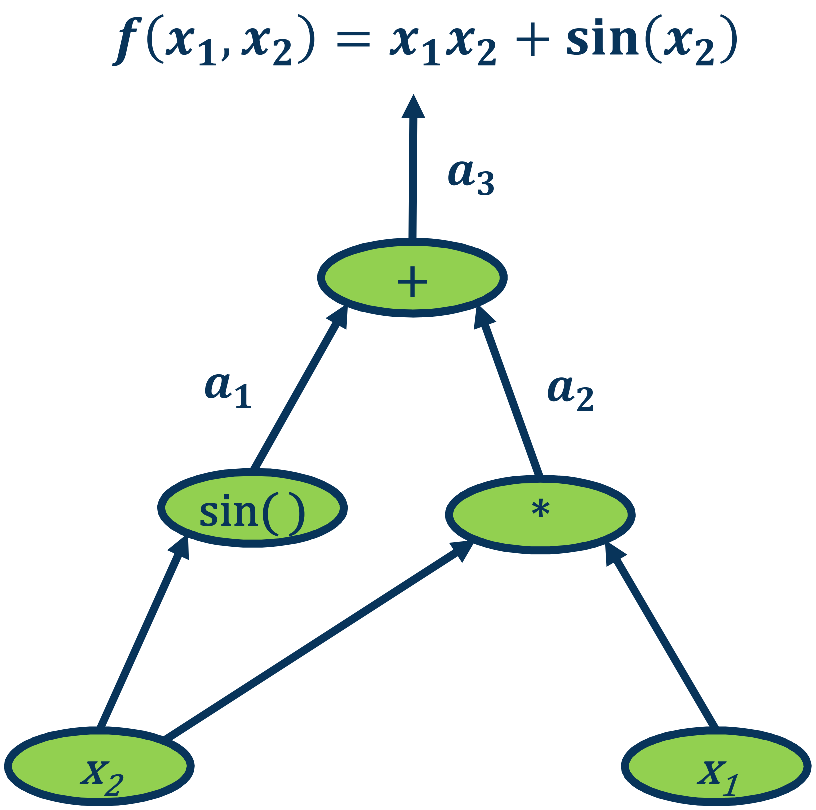
Georgia Tech

Computation = Graph

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

Auto-Diff

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



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

- Assign intermediate variables

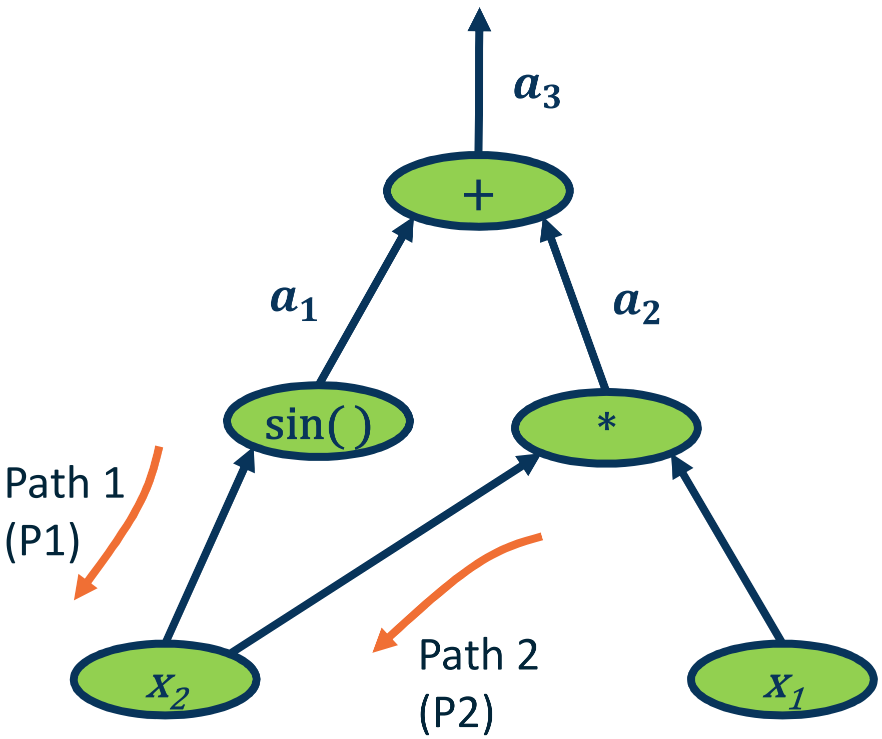
  Simplify notation:

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

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

Example

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



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

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

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

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

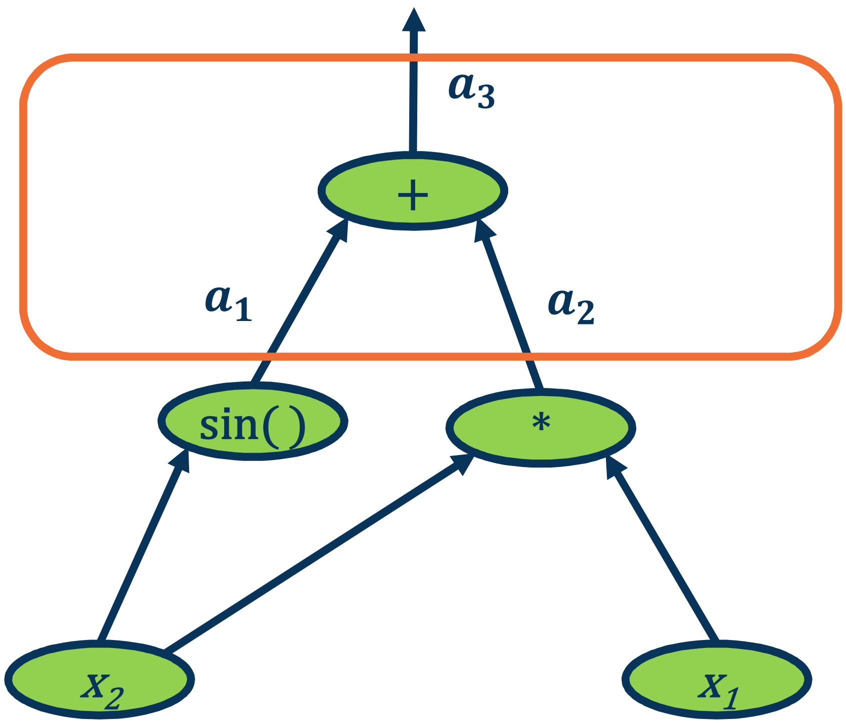
$$\overline{x_2^{P2}} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2}\frac{\partial (x1x2)}{\partial x_2} = \overline{a_2} x_1$$

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

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



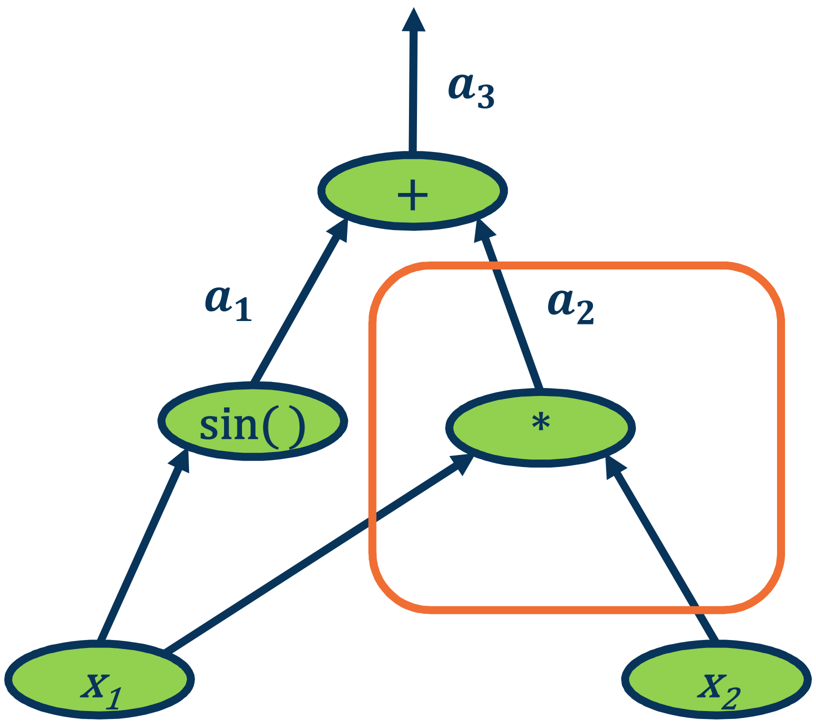
$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} \, 1 = \overline{a_3}$$

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

Addition operation distributes gradients along all paths!

Patterns of Gradient Flow: Addition

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



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

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

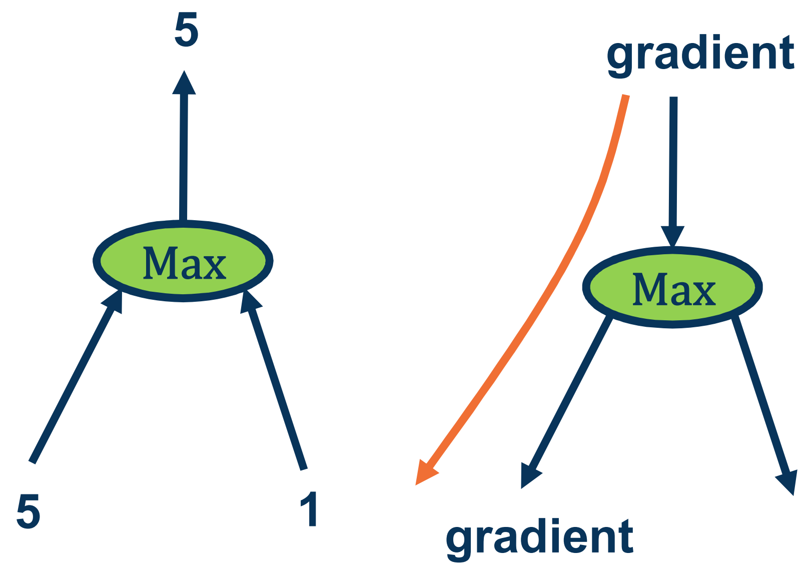
$$\overline{x_1} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = \overline{a_2} x_2$$

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

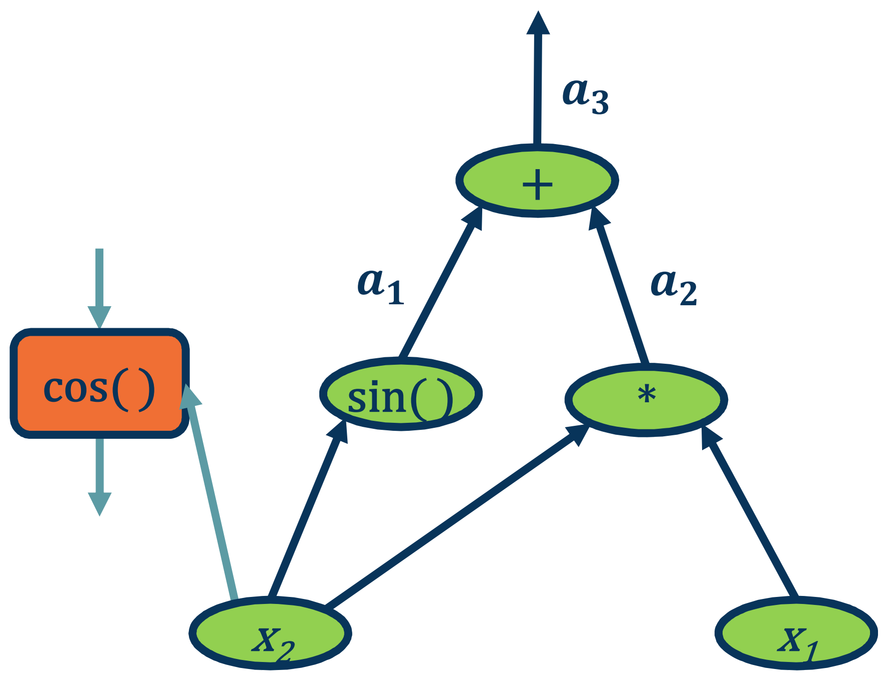
- If gradients **do not flow backwards properly,** learning slows or stops!

Patterns of Gradient Flow: Other

- 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**

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



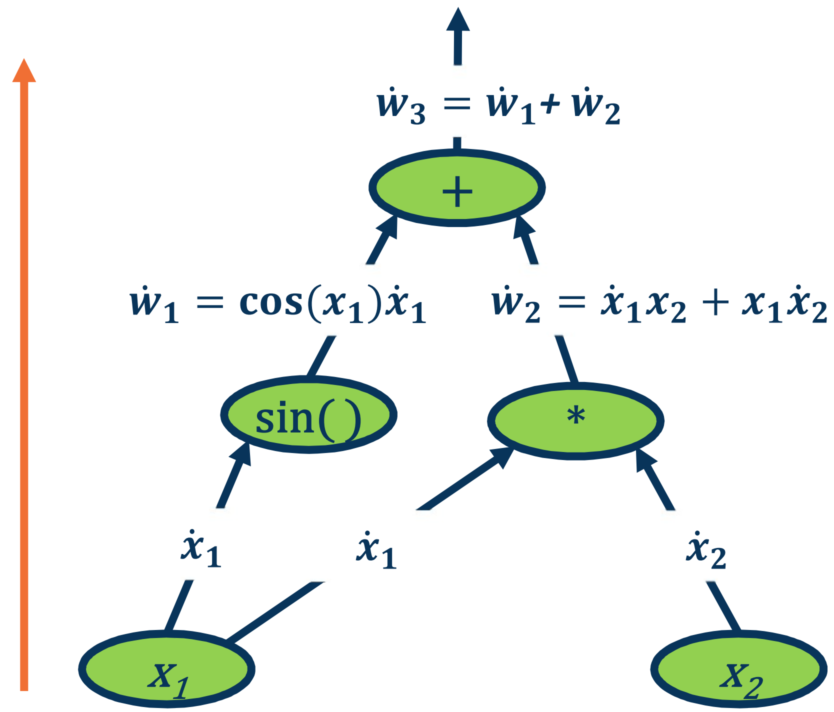
Computational Implementation

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



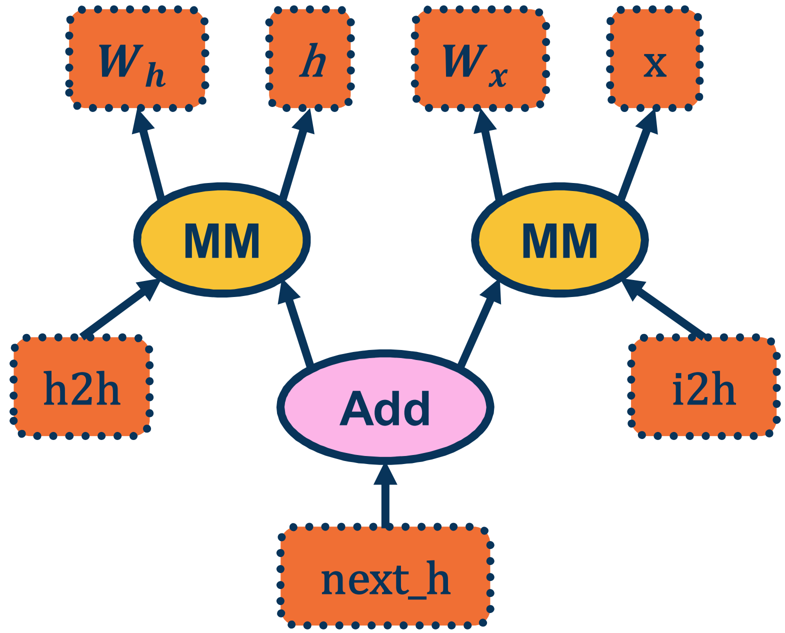
Automatic Differentiation

A graph is created on the fly

```
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)

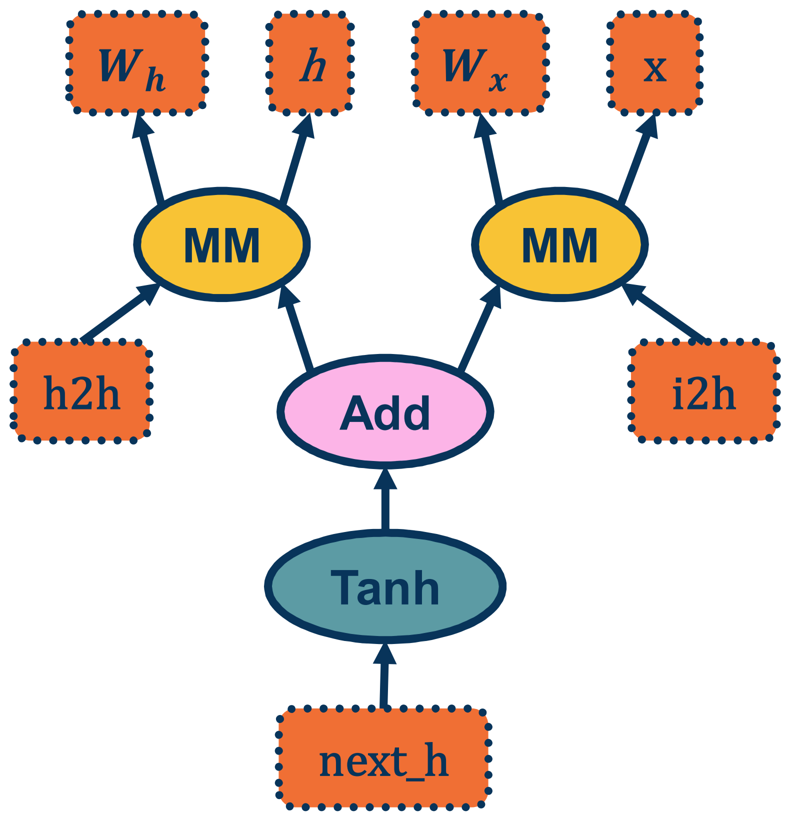
Back-propagation uses the dynamically built graph

```
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

Convolutional network (AlexNet)

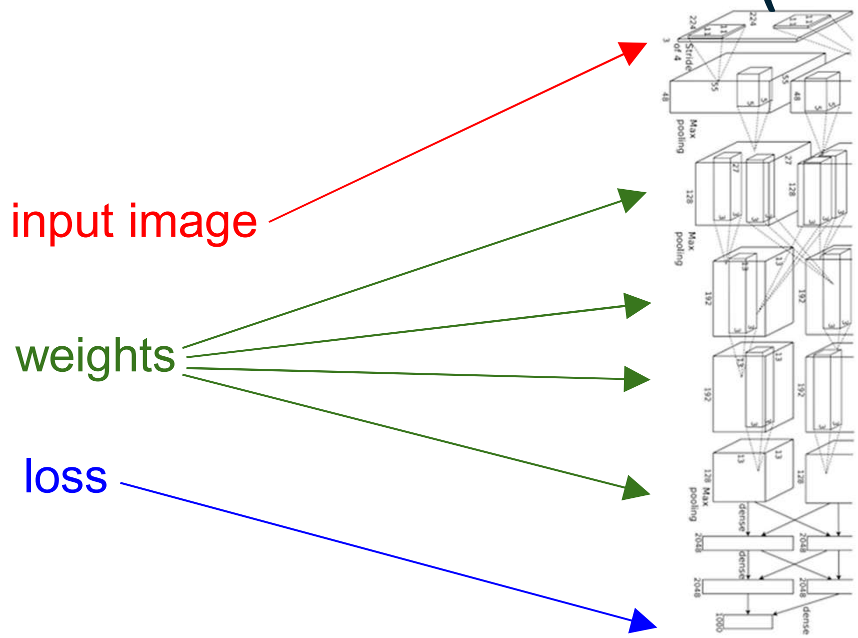


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Neural Turing Machine

input image

loss

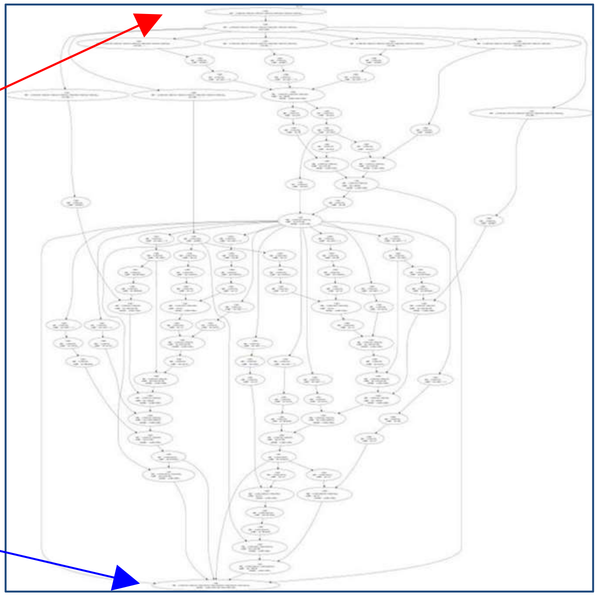
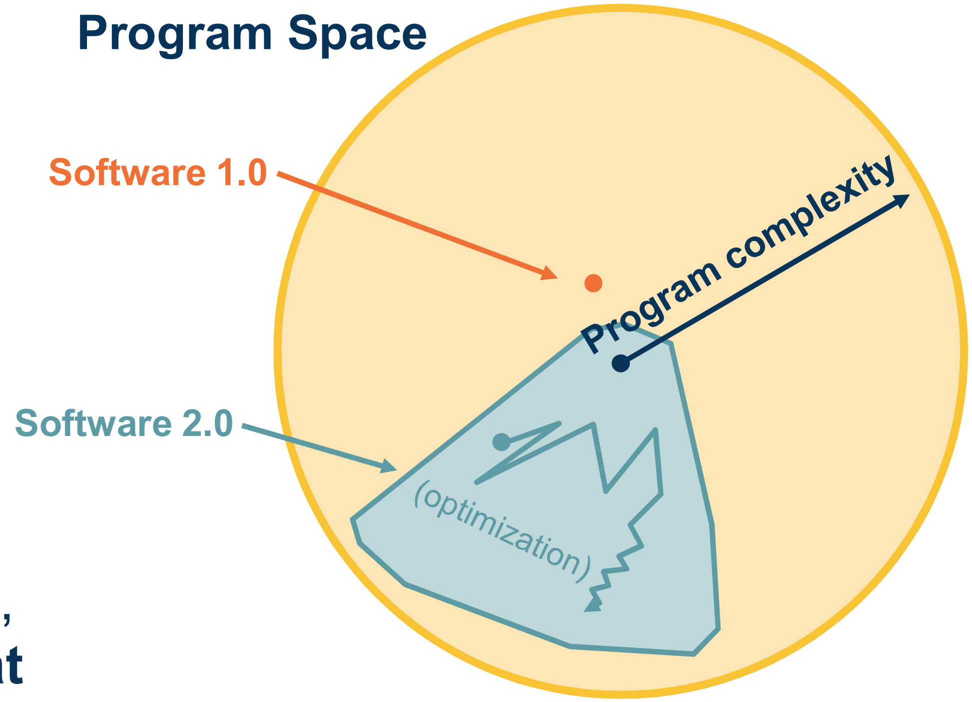


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

- 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