Topics:

- Jacobians/Matrix Calculus continued
- Backpropagation / Automatic Differentiation

# CS 4644 / 7643-A

ZSOLT KIRA

- **Assignment 1 out!**
  - **Due Jan 31st (with grace period Feb 2nd)**
  - Hopefully you have started!

- Resources:
  - These lectures
  - Matrix calculus for deep learning
  - Gradients notes and MLP/ReLU Jacobian notes.
  - Assignment 1 (@57) and matrix calculus (@80), convex optimization (@82)

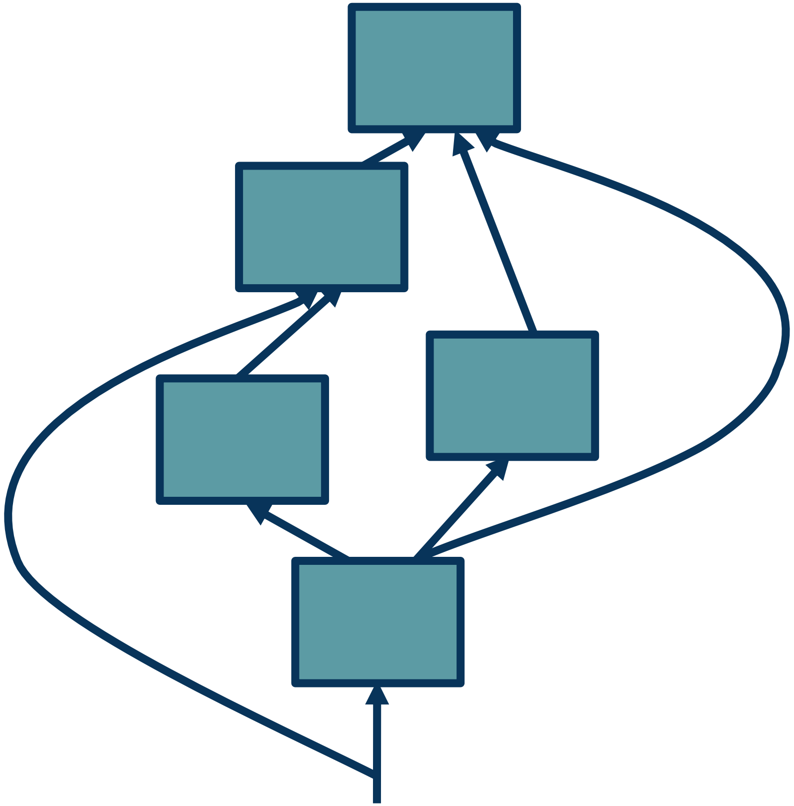
- Project:
  - Project proposal overview out, due by **Feb 14th**
  - Recorded office hours

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

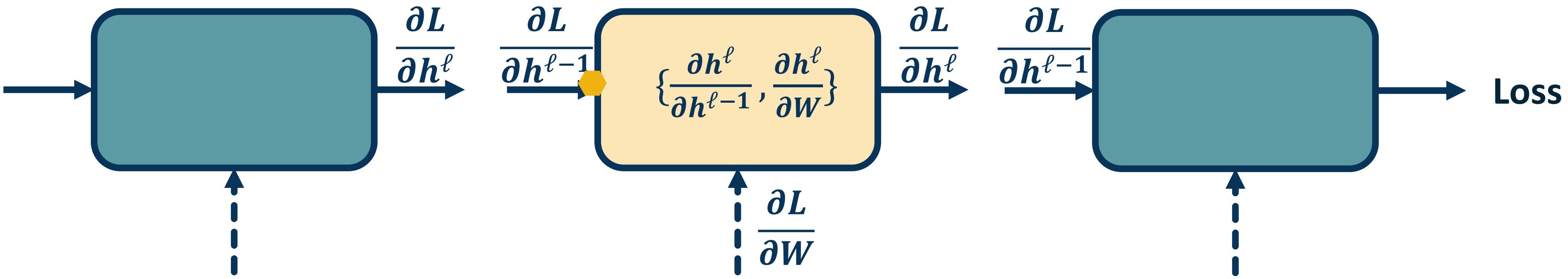
A **training algorithm** will then process this graph, **one module at a time**



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

# Computing the Gradients of Loss

# 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** |

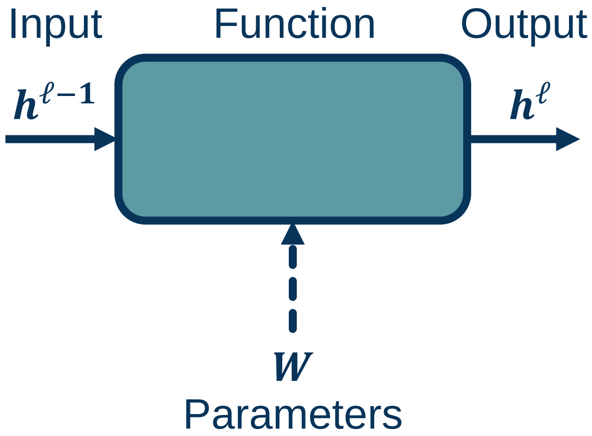
- What is the size of $\frac{\partial L}{\partial W}$ ?
  - Remember that loss is a **scalar** and $W$ is a matrix:

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

Jacobian is also a matrix:

$$W$$

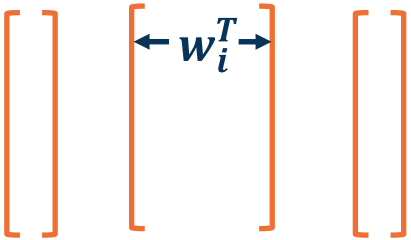
$$\begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \cdots & \frac{\partial L}{\partial w_{1m}} & \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial w_{21}} & \cdots & \cdots & \frac{\partial L}{\partial w_{2m}} & \frac{\partial L}{\partial b_2} \\ \cdots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




**Define:**

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

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




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

# Fully Connected (FC) Layer: Forward Function

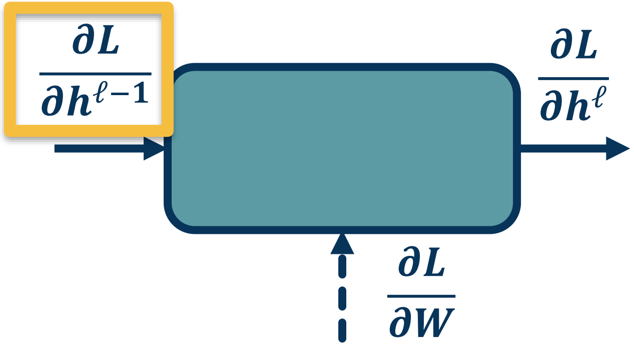
# Fully Connected (FC) Layer

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

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

**Define:**

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




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

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

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

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

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:**

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

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

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

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

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

$$\frac{\partial L}{\partial W} \begin{bmatrix} \qquad \end{bmatrix}$$

Iterate and populate

Note can simplify/vectorize!

Fully Connected (FC) Layer

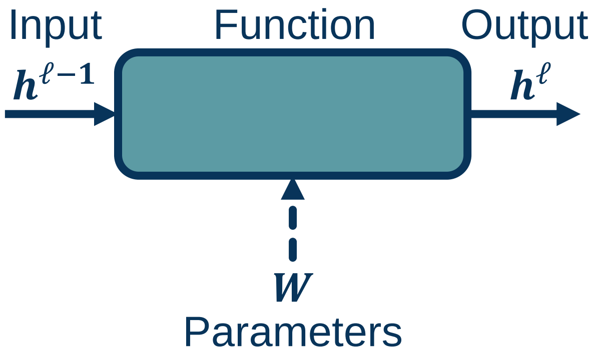
Georgia Tech

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**

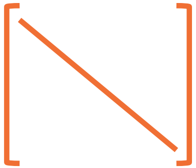
Max function **funnels gradients through selected max**

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




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

**Backward:** $\dfrac{\partial L}{\partial \boldsymbol{h}^{\ell-1}} = \dfrac{\partial L}{\partial \boldsymbol{h}^{\ell}} \; \dfrac{\partial \boldsymbol{h}^{\ell}}{\partial \boldsymbol{h}^{\ell-1}}$



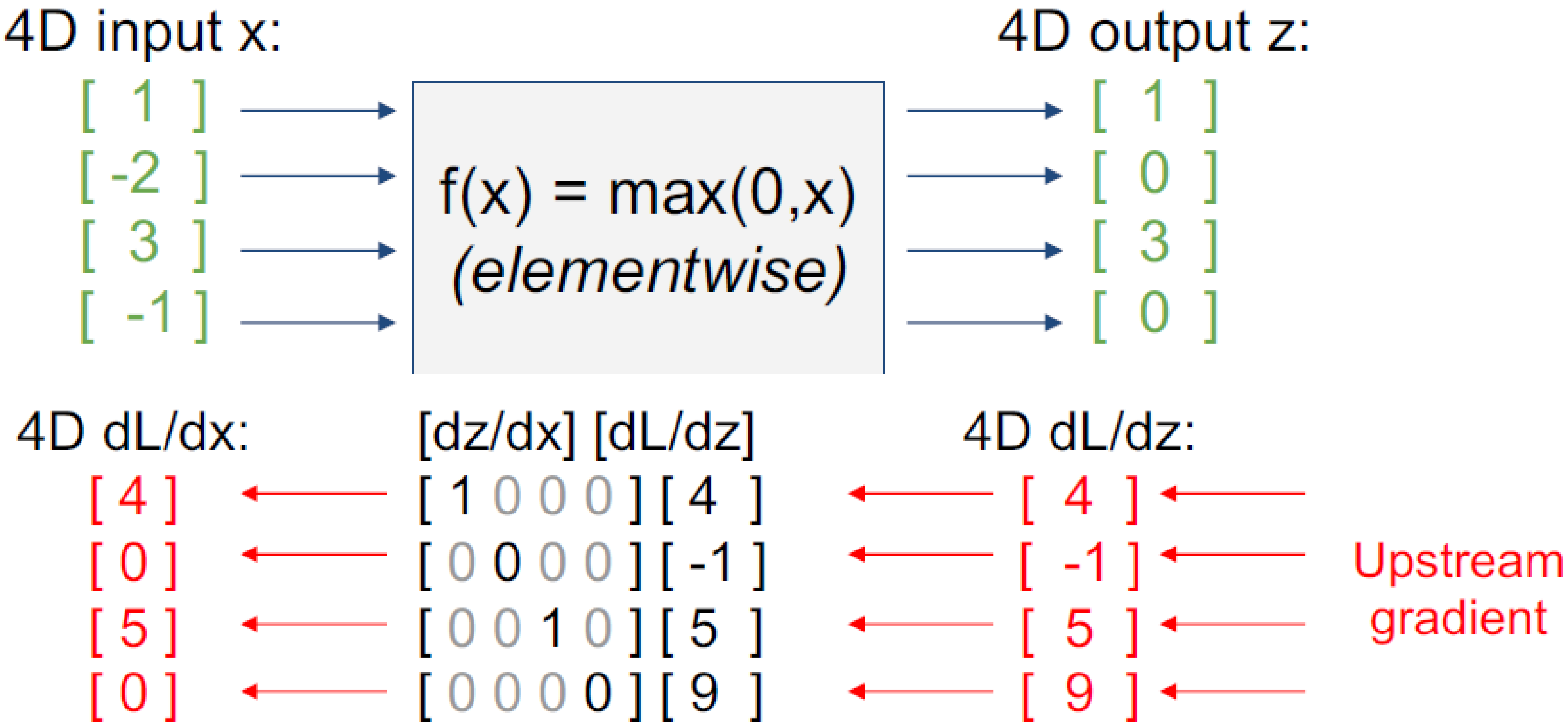
$|\boldsymbol{h}^{\ell} \times \boldsymbol{h}^{\ell-1}|$

For diagonal

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

# Jacobian of ReLU

*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

# Vectorization in Function Compositions

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

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

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

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

(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

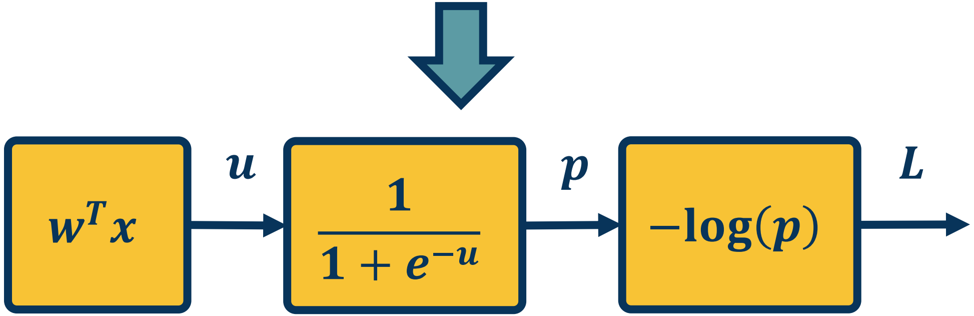
# Chain Rule: Cascaded

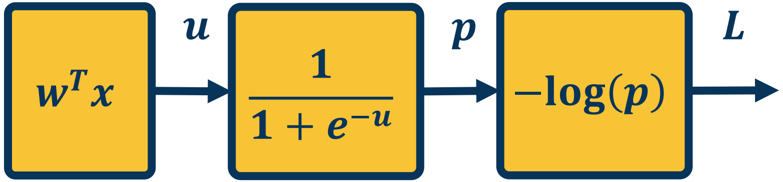
We have discussed **computation graphs for generic functions**

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!

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






$$\bar{L} = 1$$

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

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

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

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

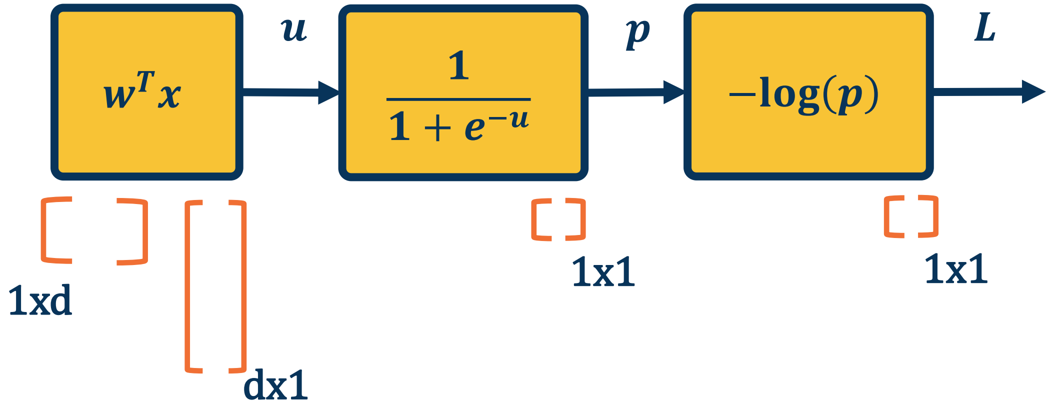
We can do this in a combined way to see all terms together:

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

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)

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

1x1 1x1 1x1 1xd

Many **standard regularization methods** still apply!

**L1 Regularization**

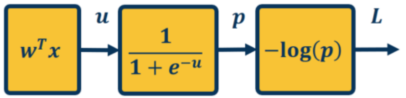
$$L = |y - Wx_i|^2 + \lambda|W|$$

where $|W|$ is element-wise

**Example regularizations:**

- L1/L2 on weights (encourage small values)
- L2: $L = |y - Wx_i|^2 + \lambda|W|^2$ (weight decay)
- Elastic L1/L2: $|y - Wx_i|^2 + \alpha|W|^2 + \beta|W|$

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




$\bar{L} = 1$

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

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

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

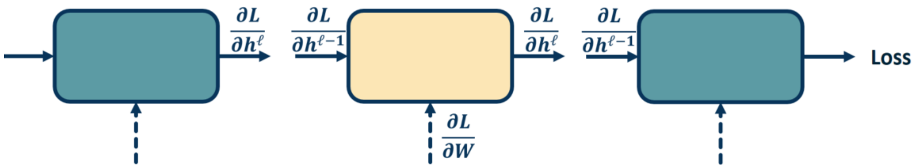
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

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

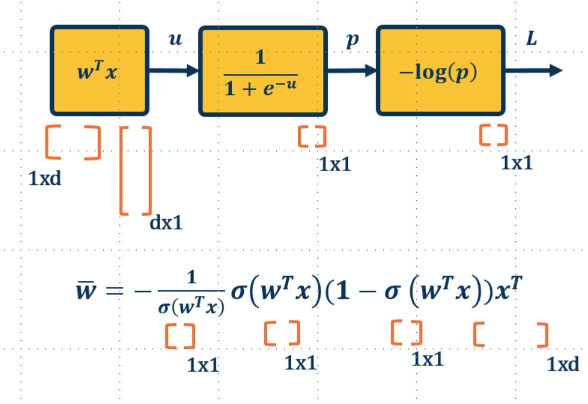
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

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

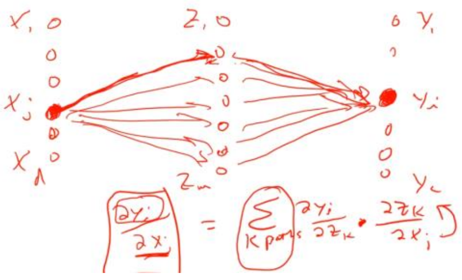
## Computation Graph of primitives (automatic differentiation)




## Backpropagation View (Recursive Algorithm)




## Computational / Tensor View




## Graph View

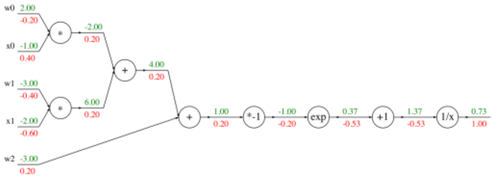
# Different Views of Equivalent Ideas

# 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)

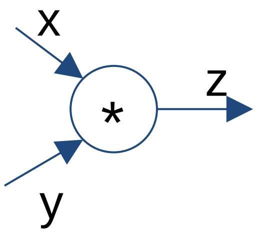
# Modularized implementation: forward / backward API




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)

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

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

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

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

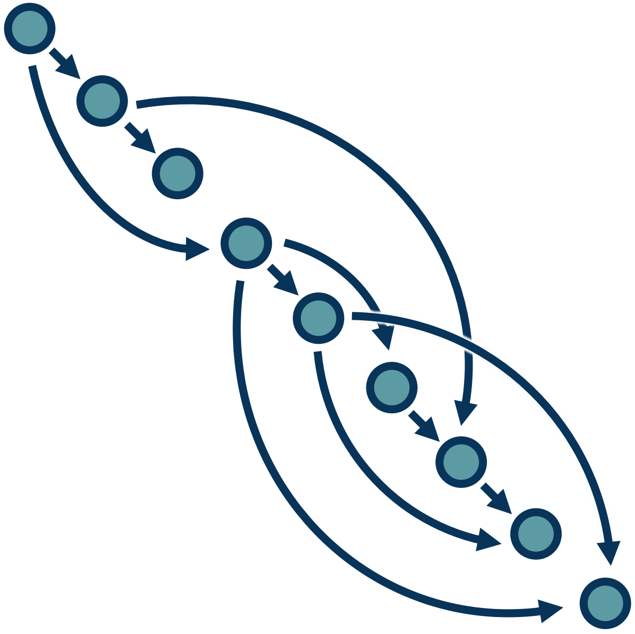
But the idea can be applied to **any directed acyclic graph (DAG)**

- 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 **local gradient function/computation for efficiency**
- 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**



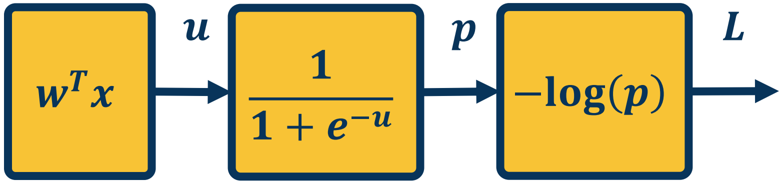
# A General Framework

## Computation = Graph

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

## Auto-Diff

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




$$\bar{L} = 1$$

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

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 $w$**

**Automatic differentiation:**

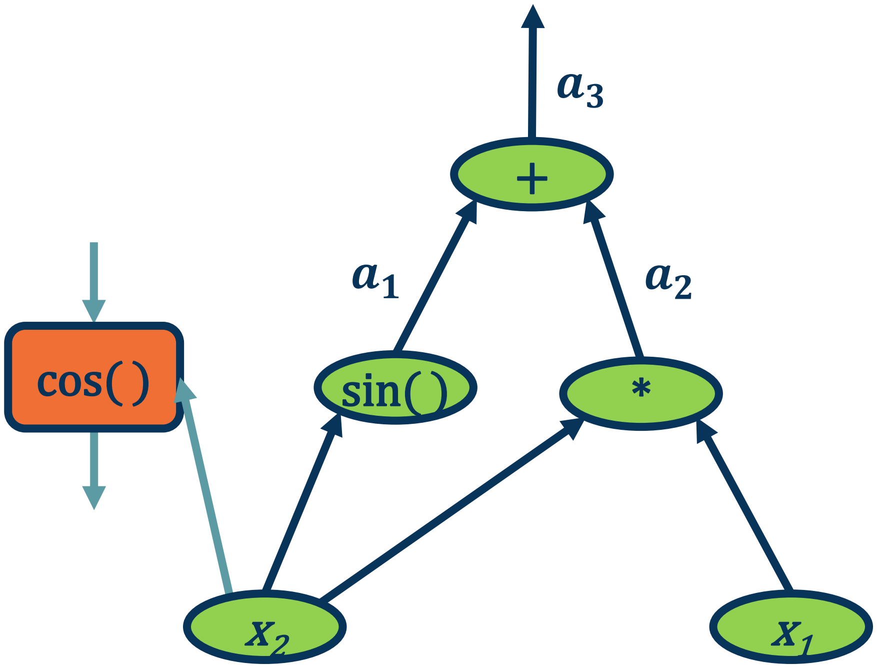
- Carries out this procedure for us on arbitrary graphs
- Knows derivatives of primitive functions
- As a result, we just define these (forward) functions **and don't even need to specify the gradient (backward) functions!**

# Example Gradient Computations

- Key idea is to **explicitly store computation graph** in memory and **corresponding gradient functions**

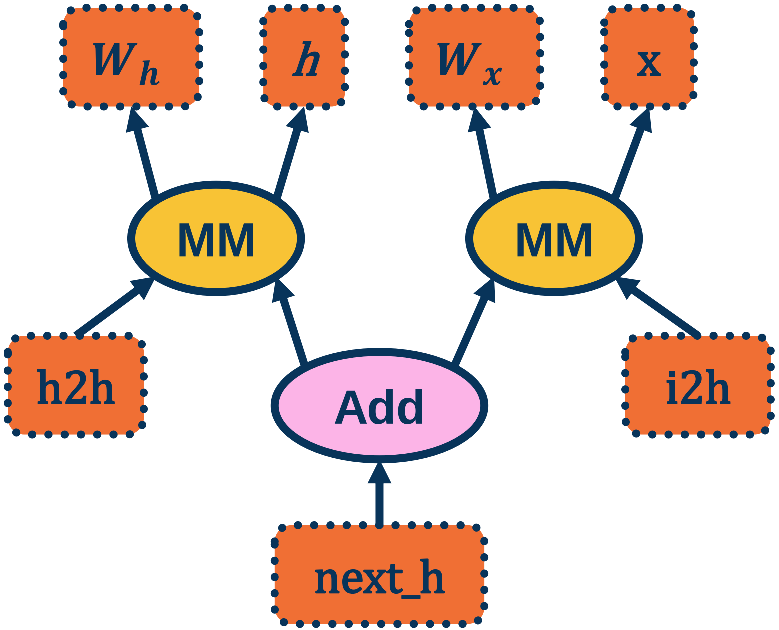
- **Nodes** broken down to **basic primitive computations** (addition, multiplication, log, etc.) for which **corresponding derivative is known**

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



## A graph is created on the fly

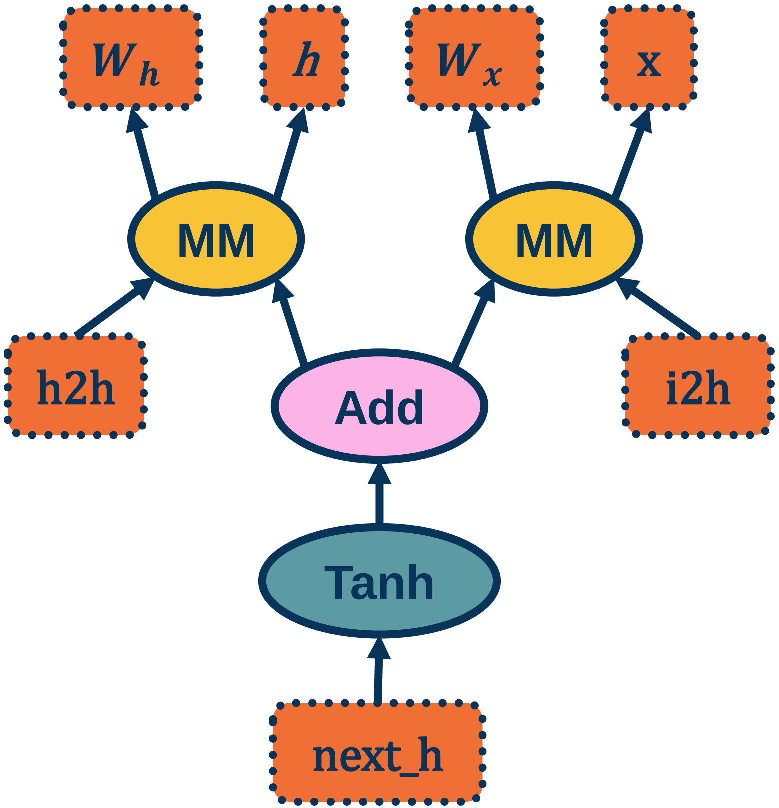
```python
from torch.autograd import Variable

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
```



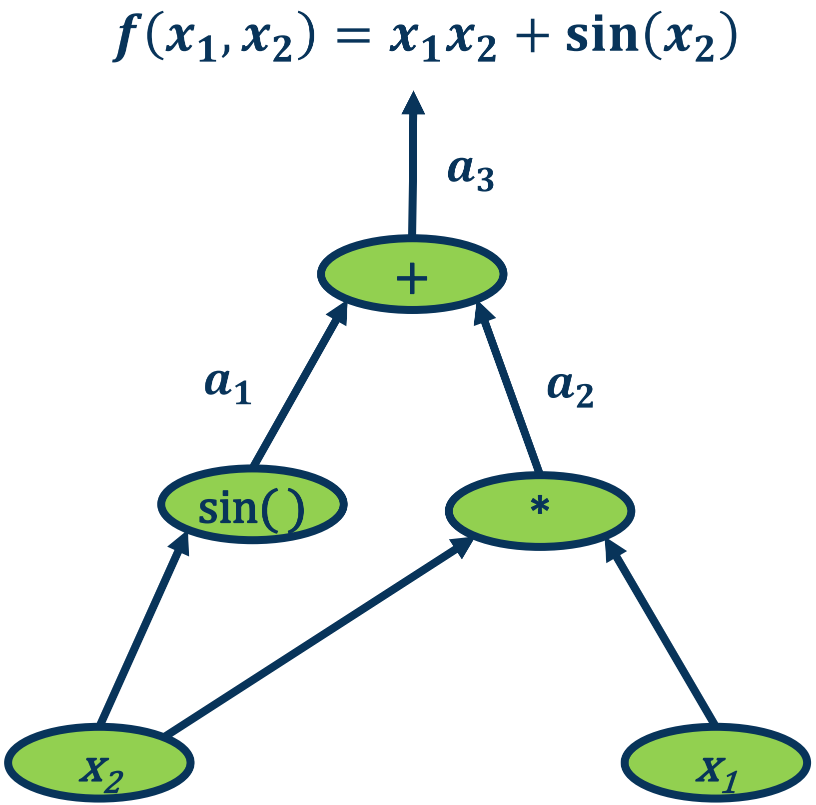

(Note above)

# Back-propagation uses the dynamically built graph

```
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




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

- Assign intermediate variables

  **Simplify notation:**

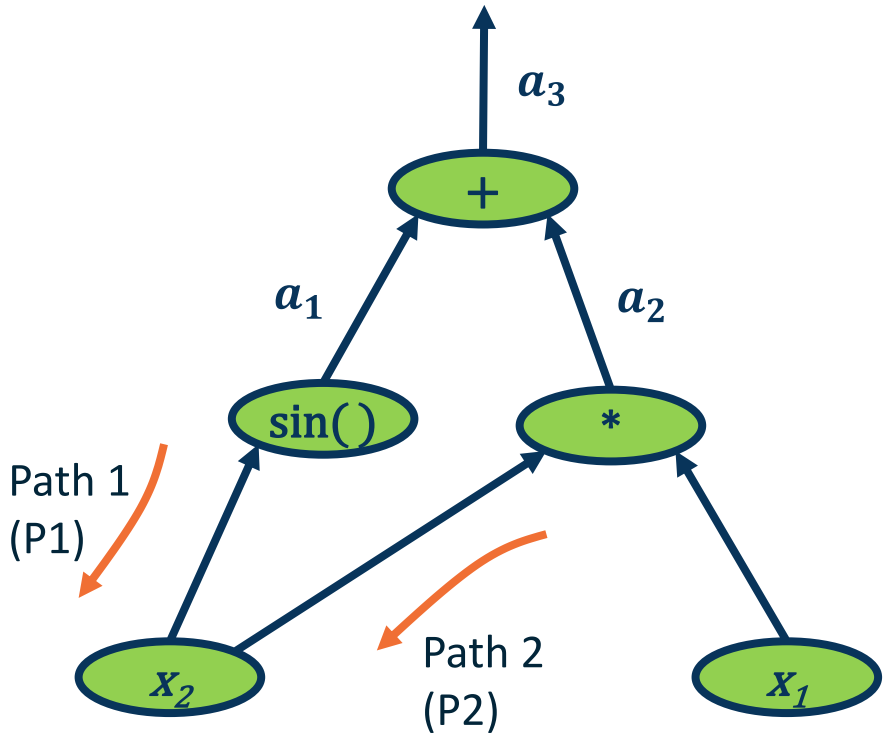
  **Denote bar as:** $\overline{a_3} = \frac{\partial f}{\partial a_3}$

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

## Example

# Example

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




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

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

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

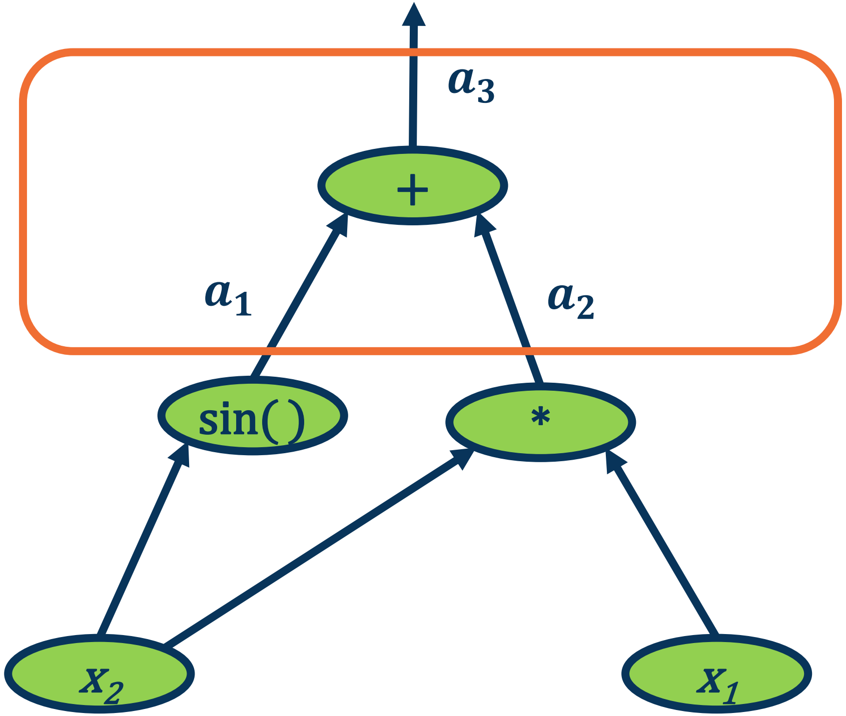
$$\overline{x_2^{P1}} = \frac{\partial f}{\partial a_1}\frac{\partial a_1}{\partial x_2} = \overline{a_1}\;\cos(x_2)$$

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

Gradients from multiple paths **summed**

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

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

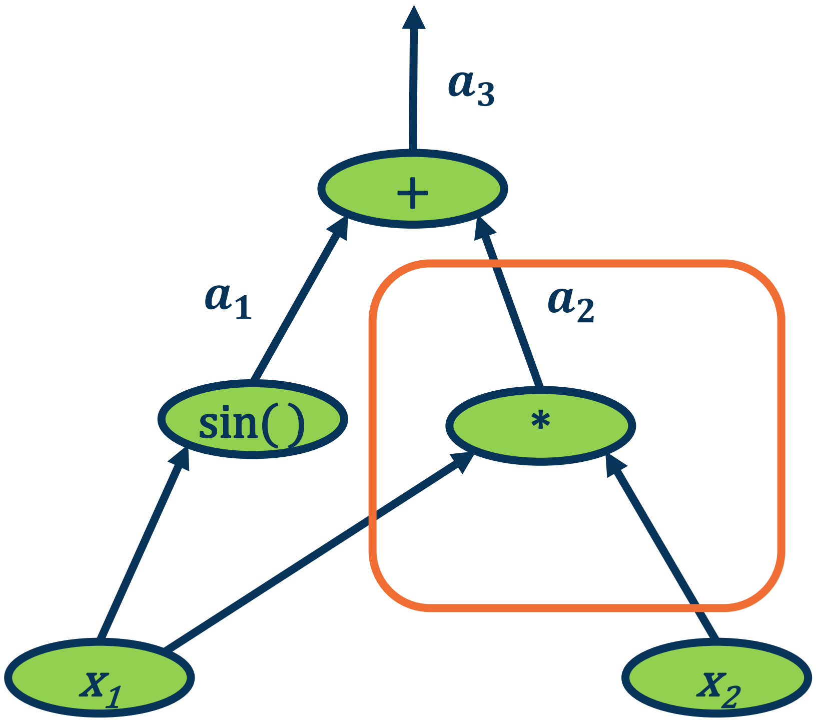



$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial(a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = \overline{a_3}$$

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

**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)

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

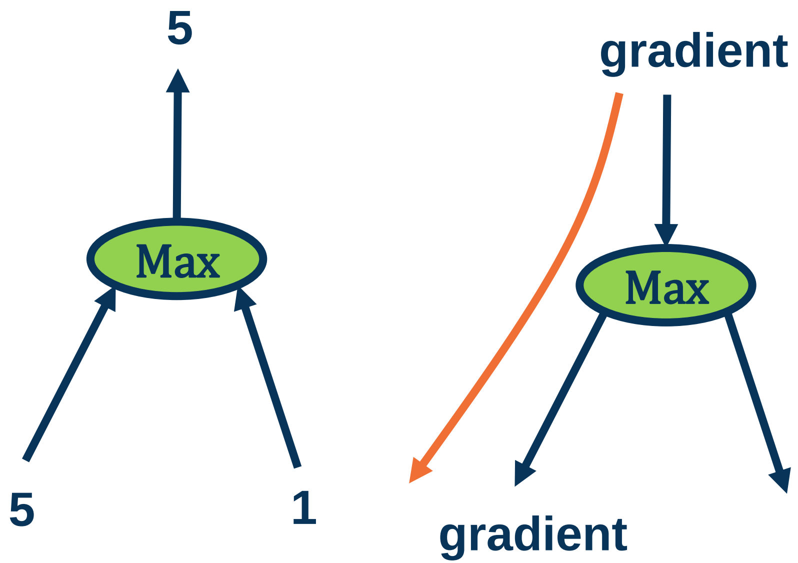
$$\overline{x_1} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_1} = \overline{a_2}x_2$$

# Patterns of Gradient Flow: Multiplication

**Several other patterns** as well, e.g.:

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

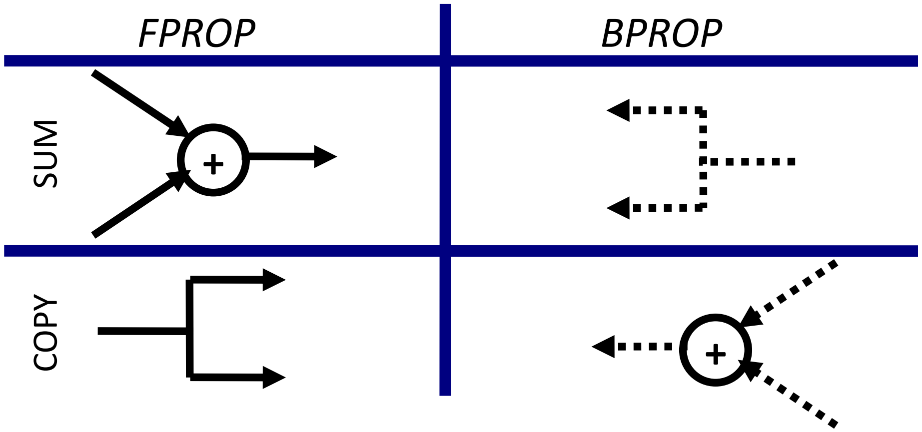
- Gradient flows along the path that was “selected” to be max
- This information must be recorded in the forward pass




**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!

# Duality in Fprop and Bprop



# Convolutional network (AlexNet)

input image

weights

loss

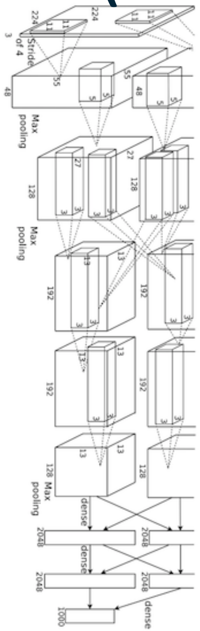


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Neural Turing Machine

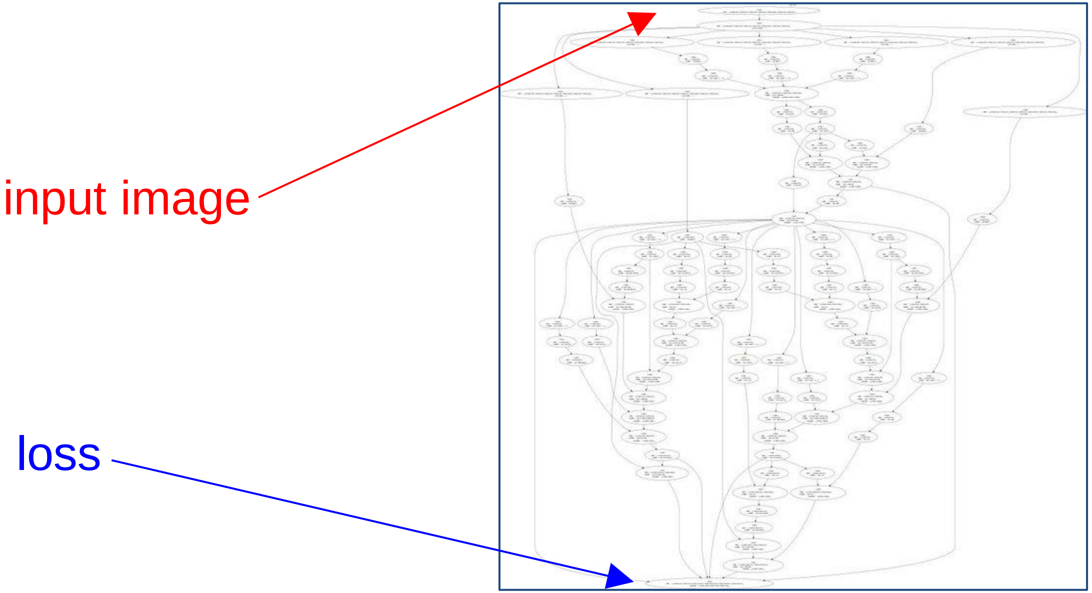
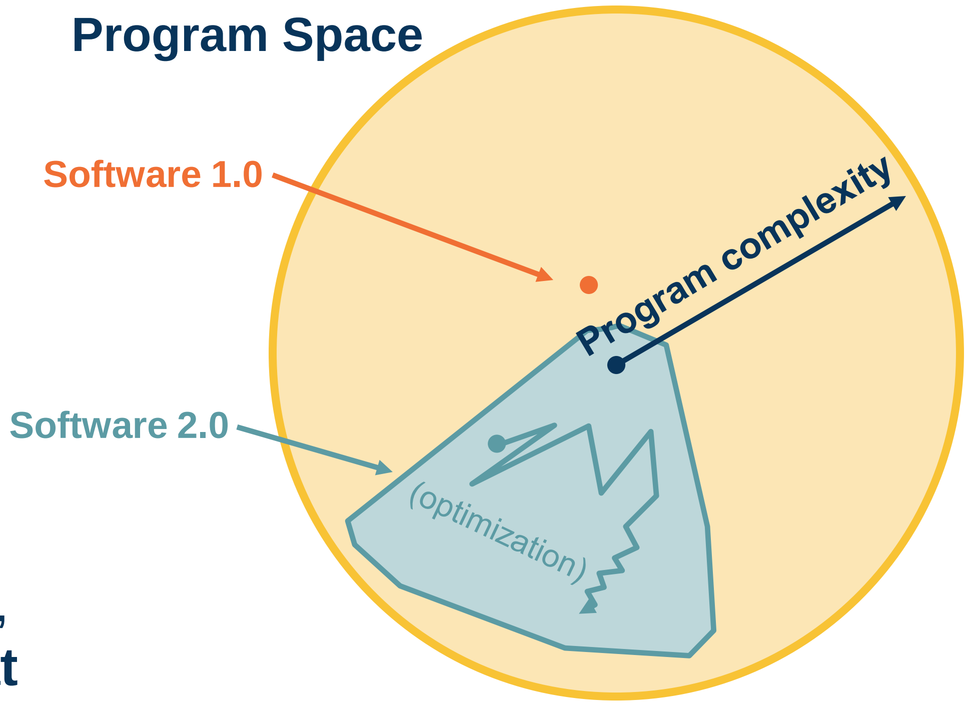



Figure reproduced with permission from a Twitter post by Andrej Karpathy.

- 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*