Topics:

- Jacobians/Matrix Calculus continued
- Backpropagation / Automatic Differentiation

CS 4644 / 7643-A
ZSOLT KIRA

- **Assignment 1 out!**
  - **Due Feb 4th**
  - 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.
  - **Topic OH:** Assignment 1 and Matrix Calculus

- **In-class Quiz (30 mins) – Feb 11**

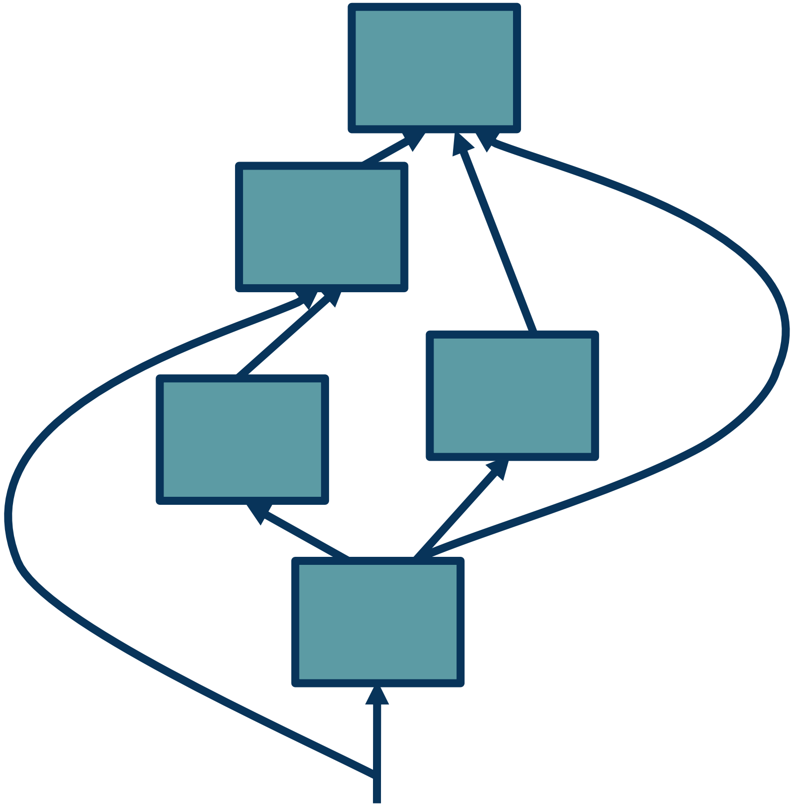
- Piazza: Project teaming thread
  - **Project Proposal: Feb. 14th**, **Project Check-in: Mar. 14th.**
  - Project proposal overview during my OH (Thursday 2pm ET, recorded)

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

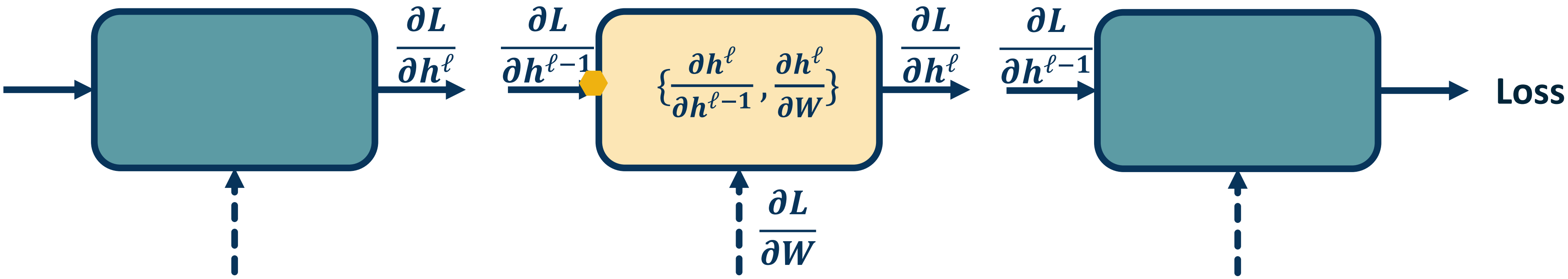
A **training algorithm** will then process this graph, **one module at a time**



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:

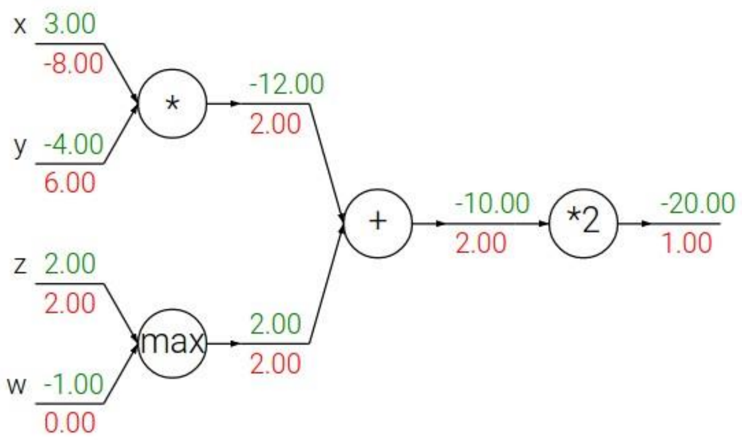
Chain Rule: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$

Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher



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

	S $[\ ]$	V $\begin{bmatrix} \\ \end{bmatrix}$	M $\begin{bmatrix} & \\ & \end{bmatrix}$
S	$\frac{\partial s_1}{\partial s_2}$ $[\ ]$	$\frac{\partial s}{\partial v}$ $[\quad]$	$\frac{\partial s}{\partial M}$ $\begin{bmatrix} & \\ & \end{bmatrix}$
V	$\frac{\partial v}{\partial s}$ $\begin{bmatrix} \\ \end{bmatrix}$	$\frac{\partial v_1}{\partial v_2}$ $\begin{bmatrix} & \\ & \end{bmatrix}$	
M	$\frac{\partial M}{\partial s}$ $\begin{bmatrix} & \\ & \end{bmatrix}$		

Tensors

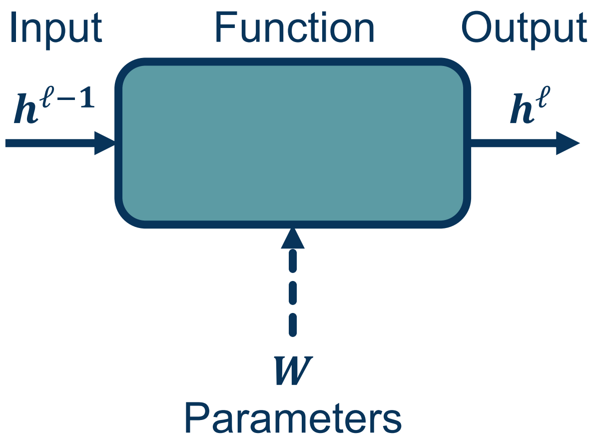
- What is the size of $\frac{\partial L}{\partial W}$?
  - Remember that loss is a **scalar** and W is a matrix:

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

$$W$$

Jacobian is also a matrix:

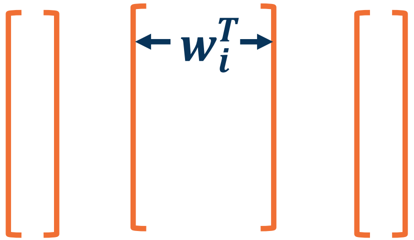
$$\begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \cdots & \frac{\partial L}{\partial w_{1m}} & \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial w_{21}} & \cdots & \cdots & \frac{\partial L}{\partial w_{2m}} & \frac{\partial L}{\partial b_2} \\ \cdots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



Define:

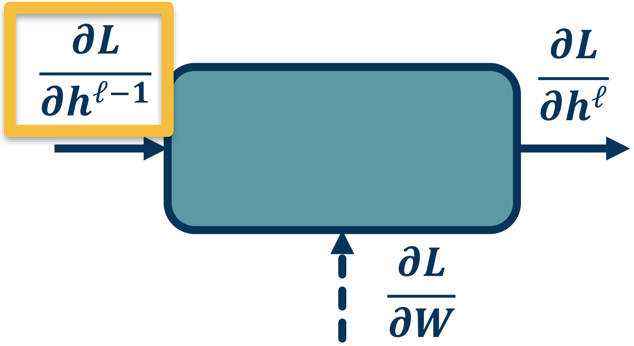
$$h_i^{\ell} = w_i^T h^{\ell-1}$$

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



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

Fully Connected (FC) Layer: Forward Function



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

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

Define:

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

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

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

Georgia Tech

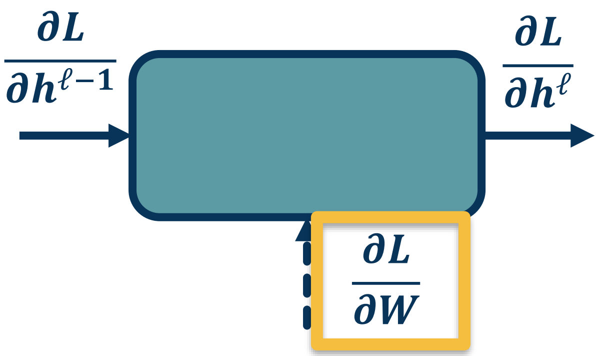
$$h^\ell = W h^{\ell-1}$$

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

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

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

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

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

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

$$\begin{bmatrix} \quad \end{bmatrix}$$

Iterate and populate

Note can simplify/vectorize!

Fully Connected (FC) Layer

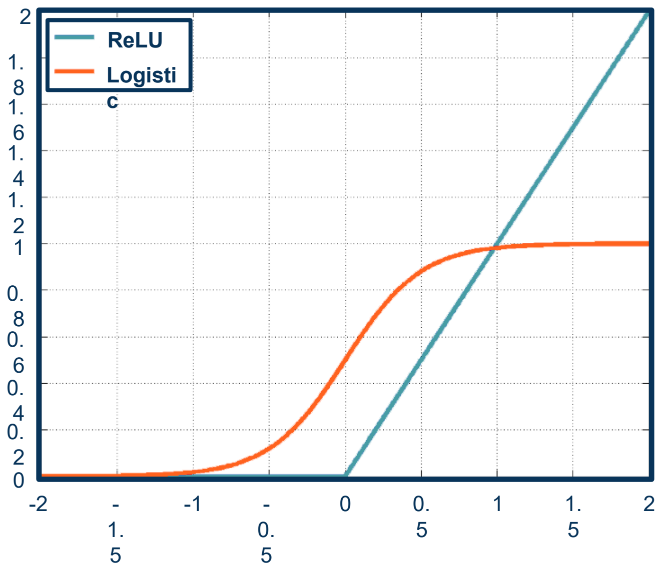
Rectified Linear Unit (ReLU)

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

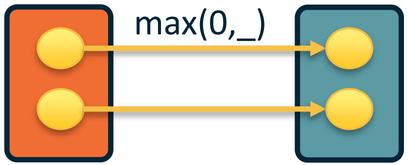
A common choice is the **Rectified Linear Unit**

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

How many parameters for this layer?



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



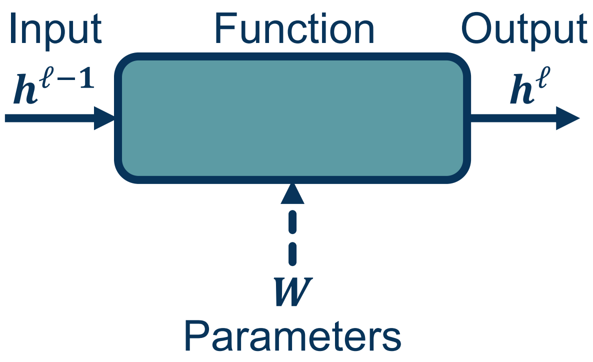
Jacobian of ReLU

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**

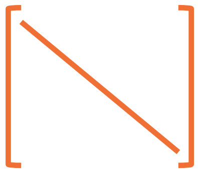
Max function **funnels gradients through selected max**

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



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

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

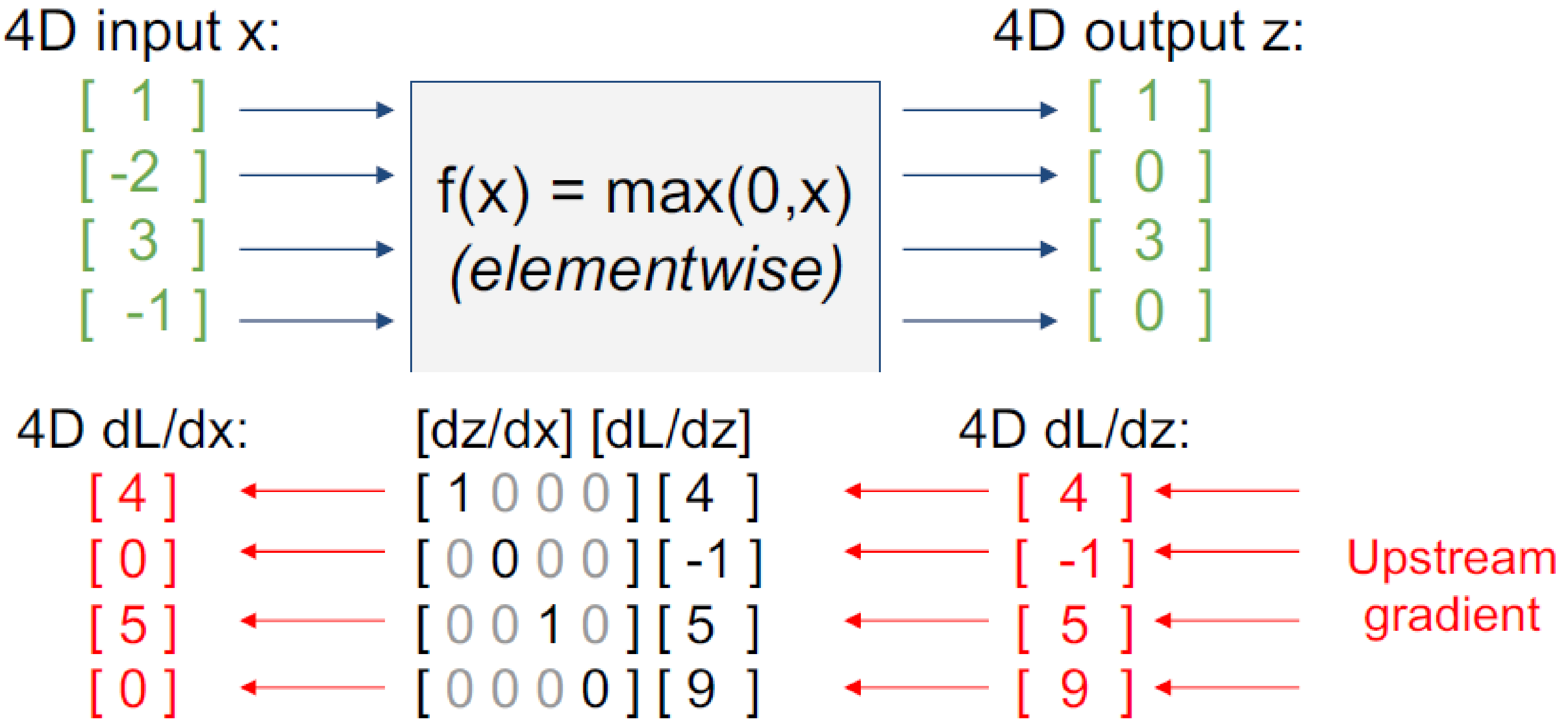


$|h^{\ell} \times h^{\ell-1}|$

For diagonal

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

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):

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

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

(Many of these will be parameterized)

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

$$\mathbf{x} \in \mathbb{R}^1 \xrightarrow{\boldsymbol{g_1}} \mathrm{z} \in \mathbb{R}^1 \xrightarrow{\boldsymbol{g_2}} \mathrm{y} \in \mathbb{R}^1$$

$$\boldsymbol{y} = \boldsymbol{g_2}(\boldsymbol{g_1}(\boldsymbol{x}))$$

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

Scalar Multiplication

Scalar Case

$$\vec{x}\{\in \mathbb{R}^d \xrightarrow[g_1]{} \vec{z}\{\in \mathbb{R}^m \xrightarrow[g_2]{} \vec{y}\{\in \mathbb{R}^c$$

$$\mathbb{R}^d \rightarrow \mathbb{R}^m \qquad \mathbb{R}^m \rightarrow \mathbb{R}^c$$

$$\underbrace{\left[\ \frac{\partial \vec{y}}{\partial \vec{x}}\ \right]}_{J_{g_1 \circ g_2}} = \underbrace{\left[\ \frac{\partial \vec{y}}{\partial \vec{z}}\ \right]}_{J_{g_1}} \underbrace{\left[\ \frac{\partial \vec{z}}{\partial \vec{x}}\ \right]}_{J_{g_2}}$$

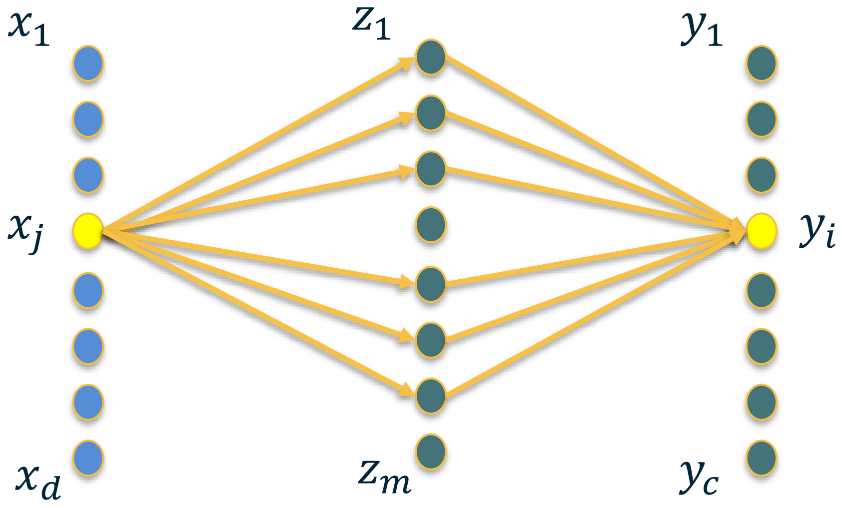
Matrix Multiplication

Vector Case

$$\left[\quad \frac{\partial y_i}{\partial x_j} \quad\right] = \left[\quad \frac{\partial y_i}{\partial z_k} \quad\right] \left[\quad \frac{\partial z_k}{\partial x_j} \quad\right]$$

$$\frac{\partial y_i}{\partial x_j} = \sum_k \frac{\partial y_i}{\partial z_k} * \frac{\partial z_k}{\partial x_j}$$

Jacobian View of Chain Rule



$$\frac{\partial y_i}{\partial x_j} = \sum_k \frac{\partial y_i}{\partial z_k} * \frac{\partial z_k}{\partial x_j}$$

k paths

Graphical View of Chain Rule

$$h^0 \in \mathbb{R}^d \longrightarrow \mathrm{h}^1 \in \mathbb{R}^{\boldsymbol{d}} \longrightarrow \quad \dots \quad \longrightarrow \mathrm{h}^l \in \mathbb{R}^{\boldsymbol{d}}$$

$$\frac{\partial h^l}{\partial h^1} = \frac{\partial h^l}{\partial h^{l-1}} \frac{\partial h^{l-1}}{\partial h^{l-2}} \dots \frac{\partial h^2}{\partial h^1}$$

$$\left[\quad\right] = \left[\quad\right]\left[\quad\right]\left[\quad\right]$$

Chain Rule: Cascaded

$$h^0 \in \mathbb{R}^d \longrightarrow h^1 \in \mathbb{R}^d \longrightarrow \quad \dots \quad \longrightarrow h^l \in \mathbb{R}^d \longrightarrow L \in \mathbb{R}^1$$

$$\frac{\partial L}{\partial h^1} = \frac{\partial L}{\partial h^l} \frac{\partial h^l}{\partial h^{l-1}} \frac{\partial h^{l-1}}{\partial h^{l-2}} \dots \frac{\partial h^2}{\partial h^1}$$

$$[\quad] = [\quad][\quad][\quad][\quad]$$

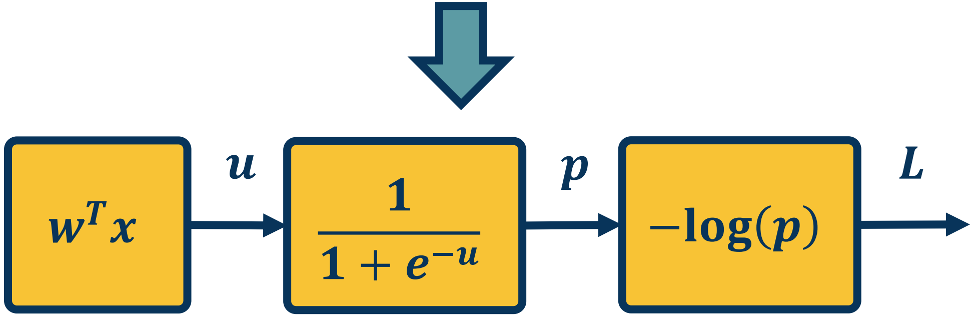
Which directions is more efficient to multiply?

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

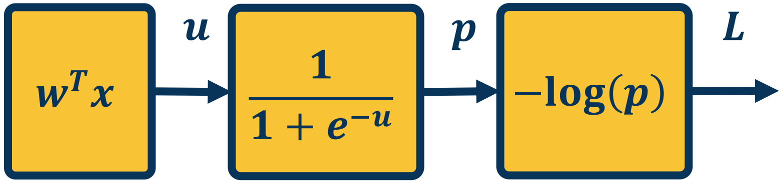
Machine Learning functions **(input -> model -> loss function)** is also a computation graph

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:

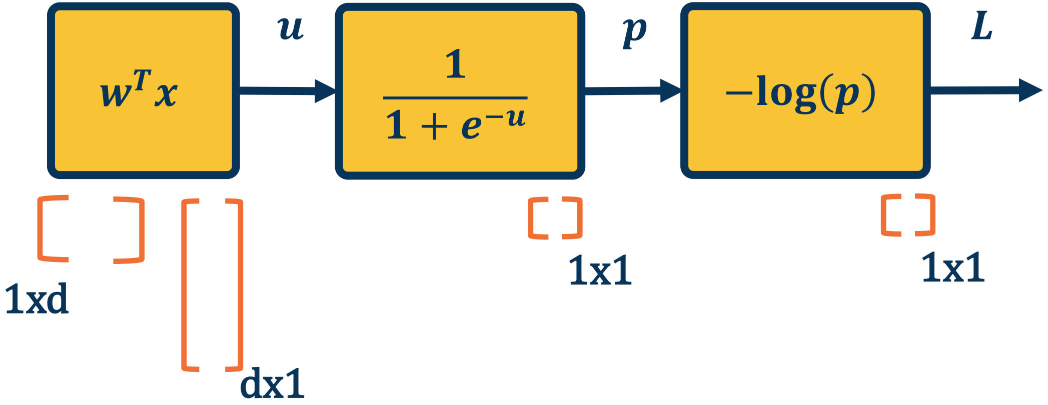
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = \bar{L}\,\bar{p}\,\bar{u} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma\,(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

This effectively shows gradient flow along path from L to w

Example Gradient Computations

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)

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

Many **standard regularization methods** still apply!

L1 Regularization

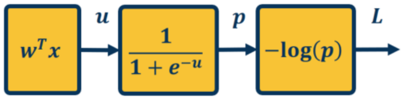
$$L = |y - Wx_i|^2 + \lambda|W|$$

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

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

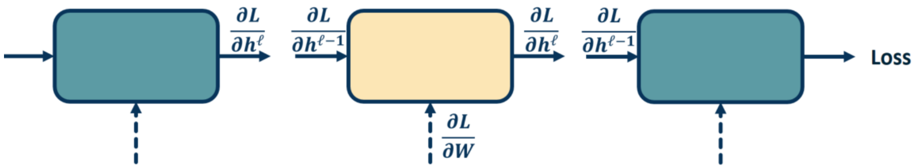
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

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

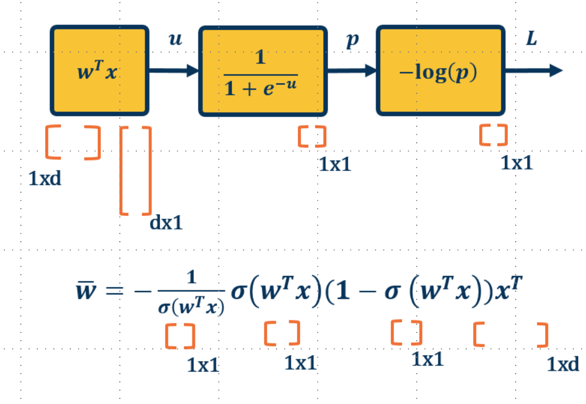
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

This effectively shows gradient flow along path from L to w

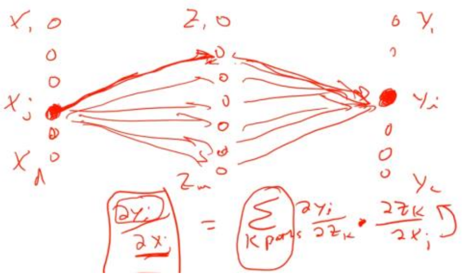
Computation Graph of primitives (automatic differentiation)



Backpropagation View (Recursive Algorithm)



Computational / Tensor View



Graph View

Different Views of Equivalent Ideas

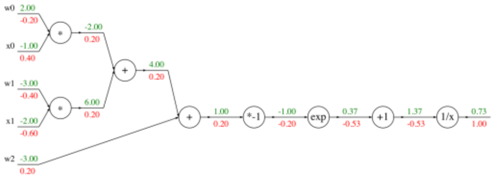
Georgia Tech

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

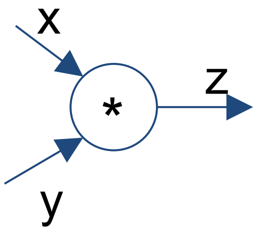


Graph (or Net) object *(rough psuedo code)*

```python
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
```


Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

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]
```

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

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

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

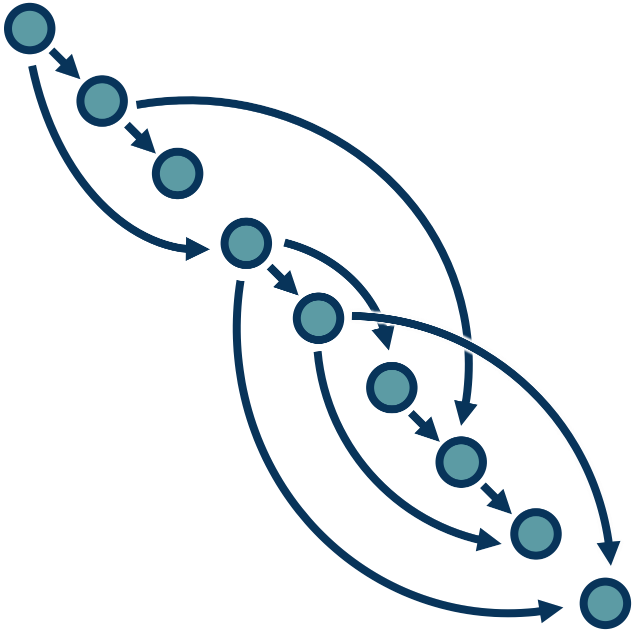
But the idea can be applied to **any directed acyclic graph (DAG)**

- 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**



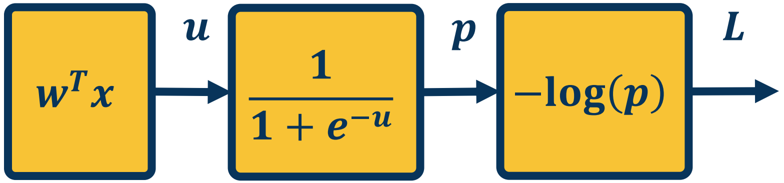
A General Framework

Computation = Graph

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

Auto-Diff

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



Example Gradient Computations

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:

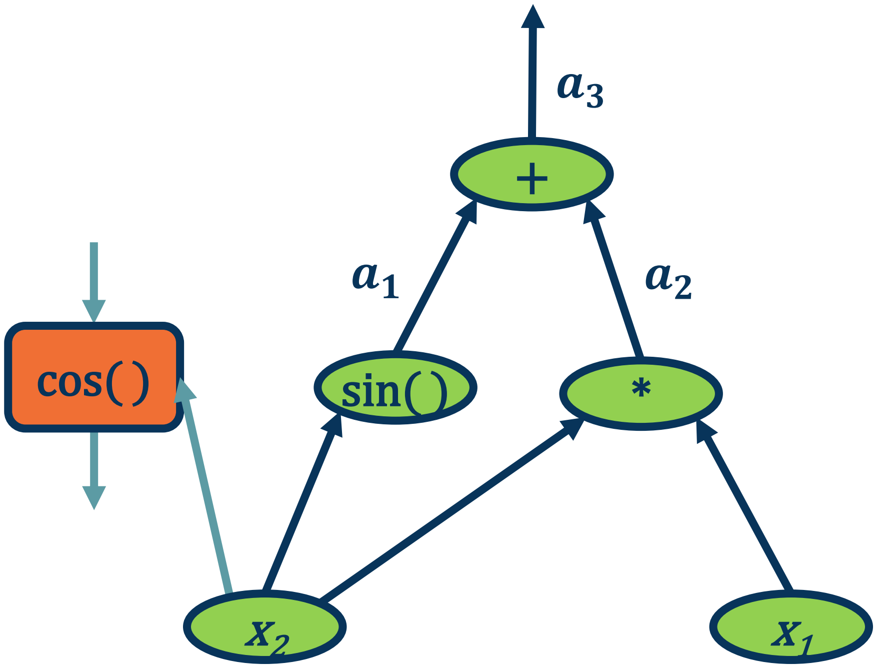
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

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

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

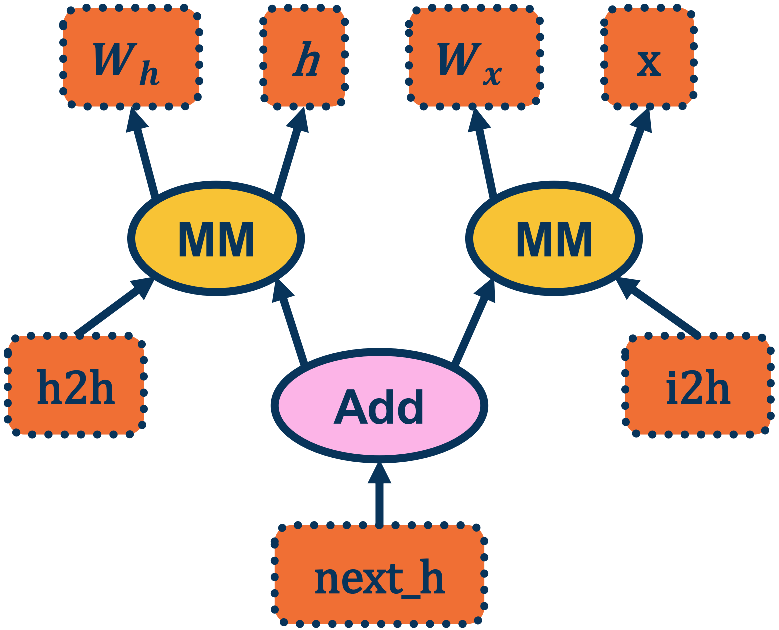
- Key idea is to **explicitly store computation graph** in memory and **corresponding gradient functions**

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



A graph is created on the fly

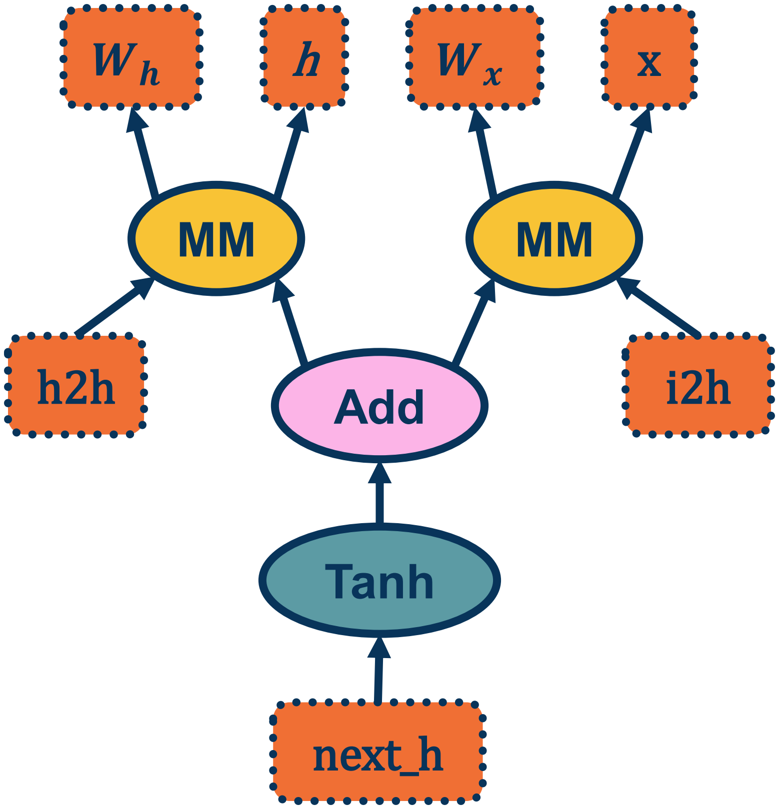
```
from torch.autograd import Variable

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
```



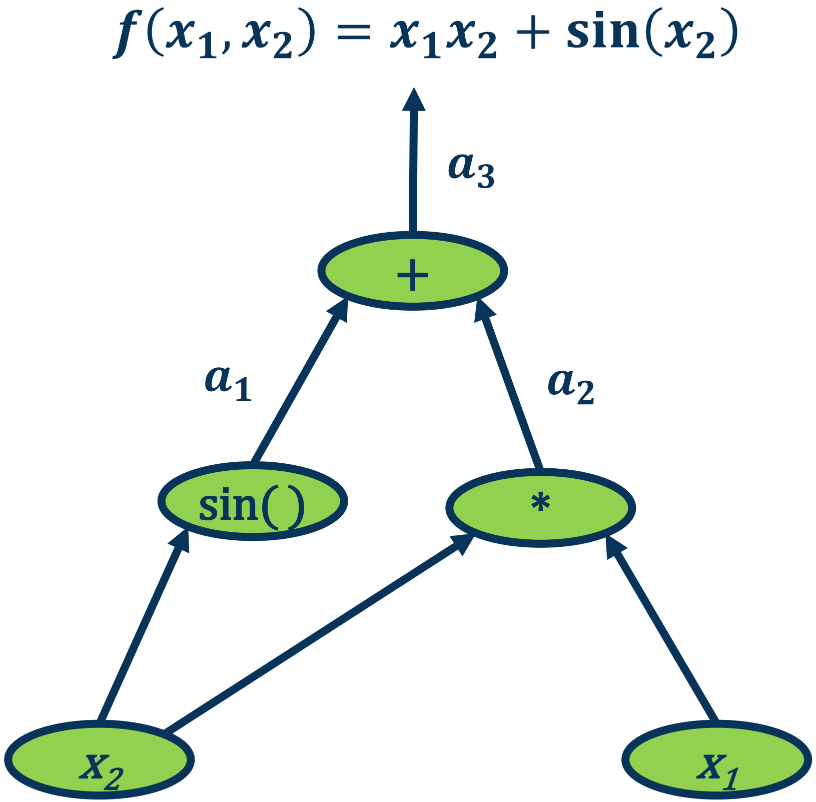
(Note above)

Back-propagation uses the dynamically built graph

```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



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

- Assign intermediate variables

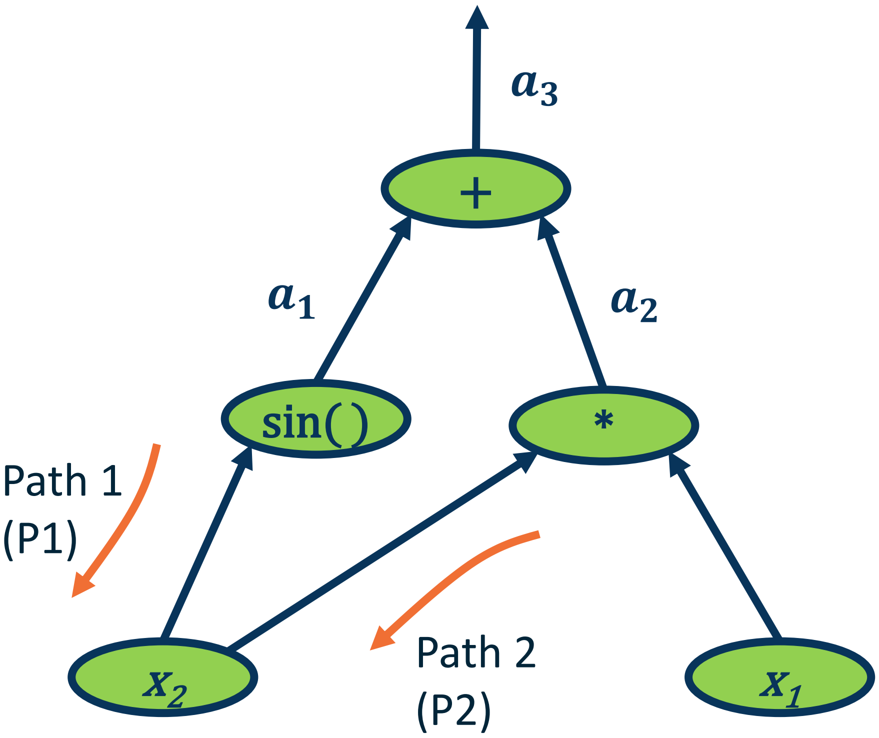
  Simplify notation:

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

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

Example

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



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

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

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

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

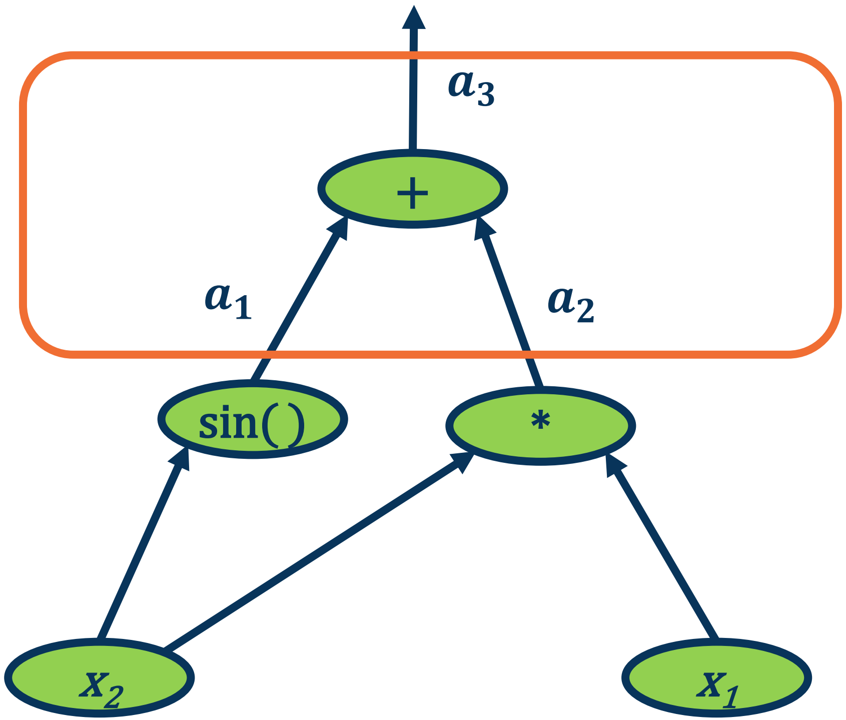
$$\overline{x_2^{P2}} = \frac{\partial f}{\partial a_2}\frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2}\frac{\partial (x1x2)}{\partial x_2} = \overline{a_2} x_1$$

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

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



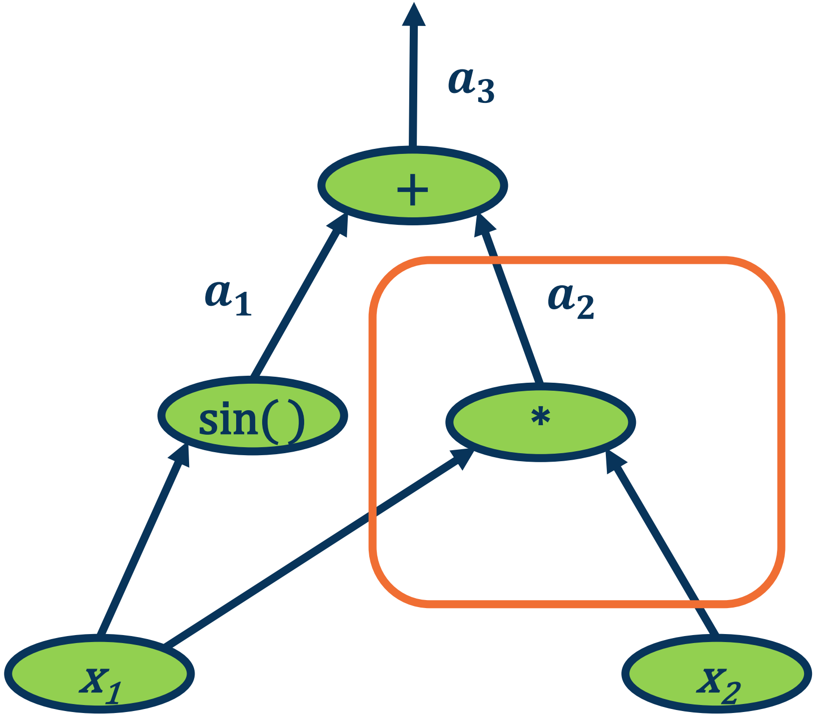
$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3}\frac{\partial(a_1+a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = \overline{a_3}$$

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

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)

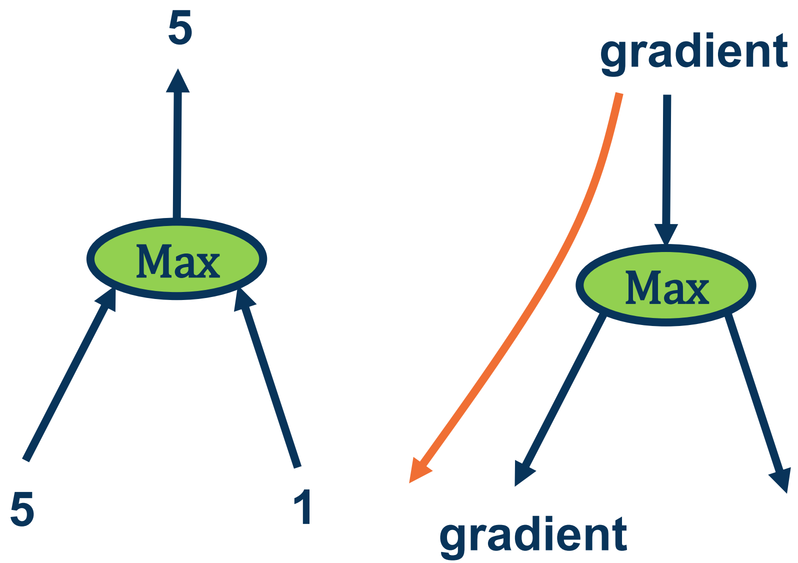
$$\overline{x_2} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x1x2)}{\partial x_2} = \overline{a_2} x_1$$

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

Several other patterns as well, e.g.:

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

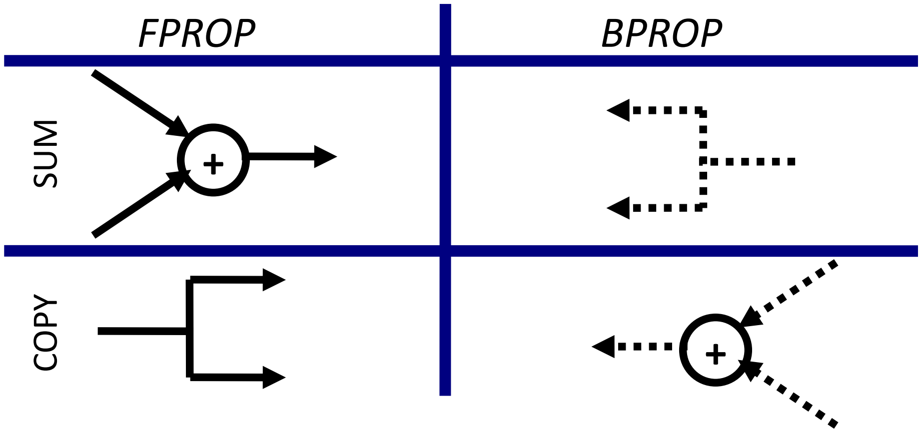
- Gradient flows along the path that was "selected" to be max
- This information must be recorded in the forward pass



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

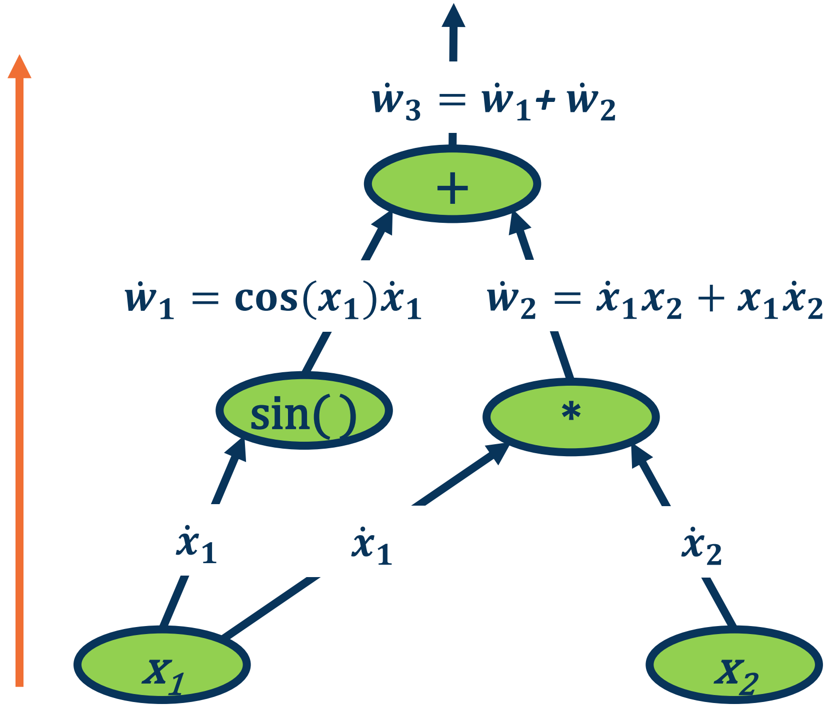


Note that we can also do **forward mode** automatic differentiation

Start from **inputs** and propagate gradients forward

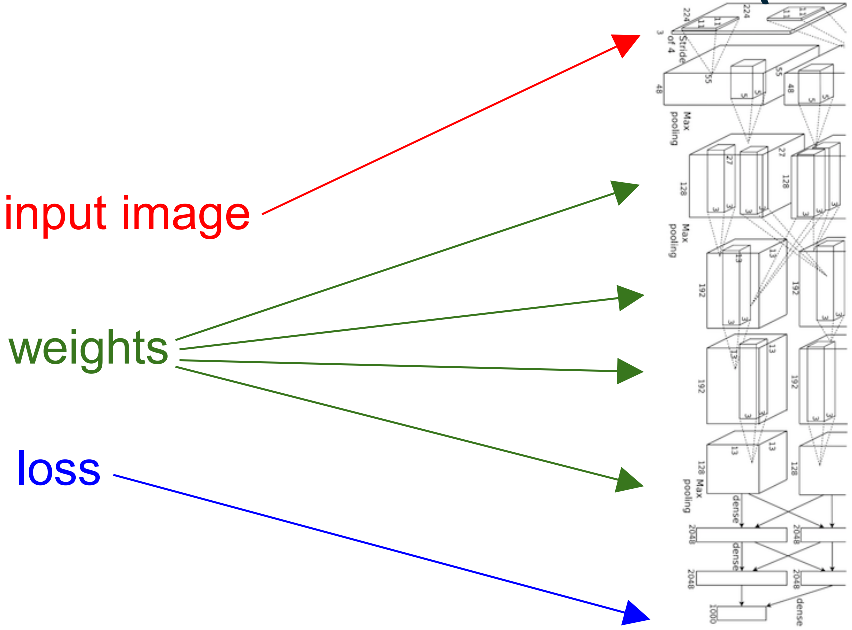
Complexity is proportional to input size

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



Forward Automatic Differentiation

Convolutional network (AlexNet)



Neural Turing Machine

input image

loss

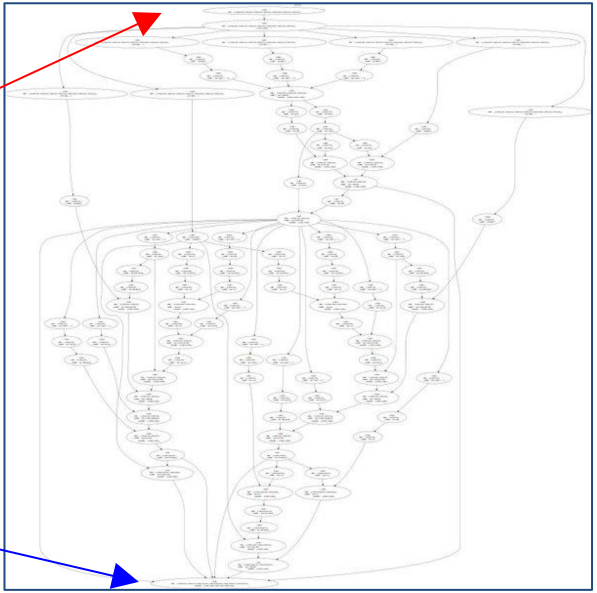
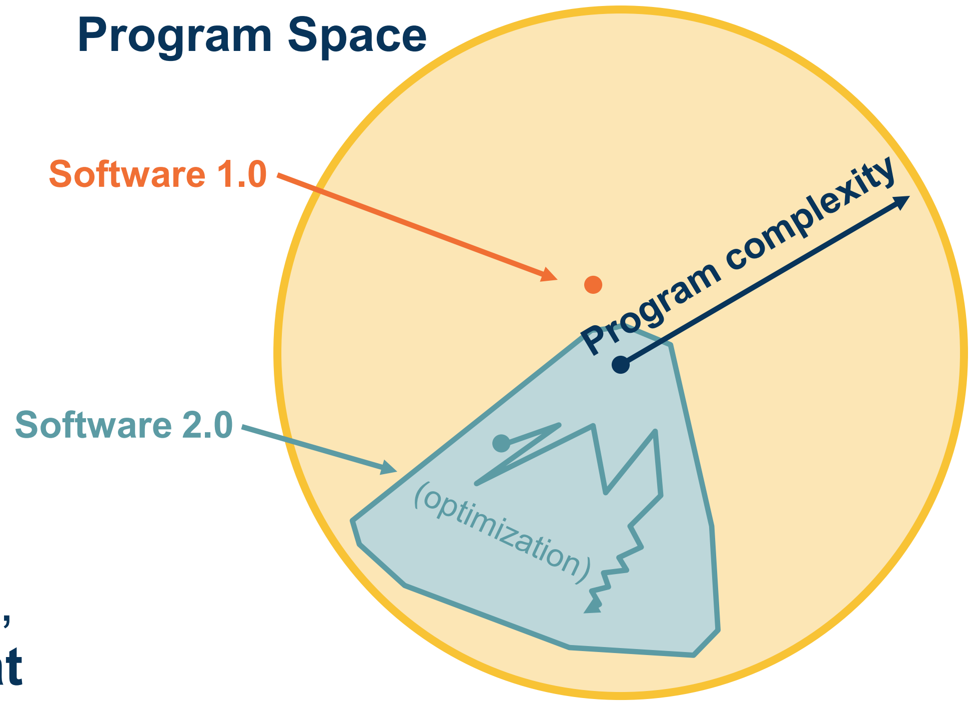


Figure reproduced with permission from a [Twitter post] by Andrej Karpathy.

- 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**

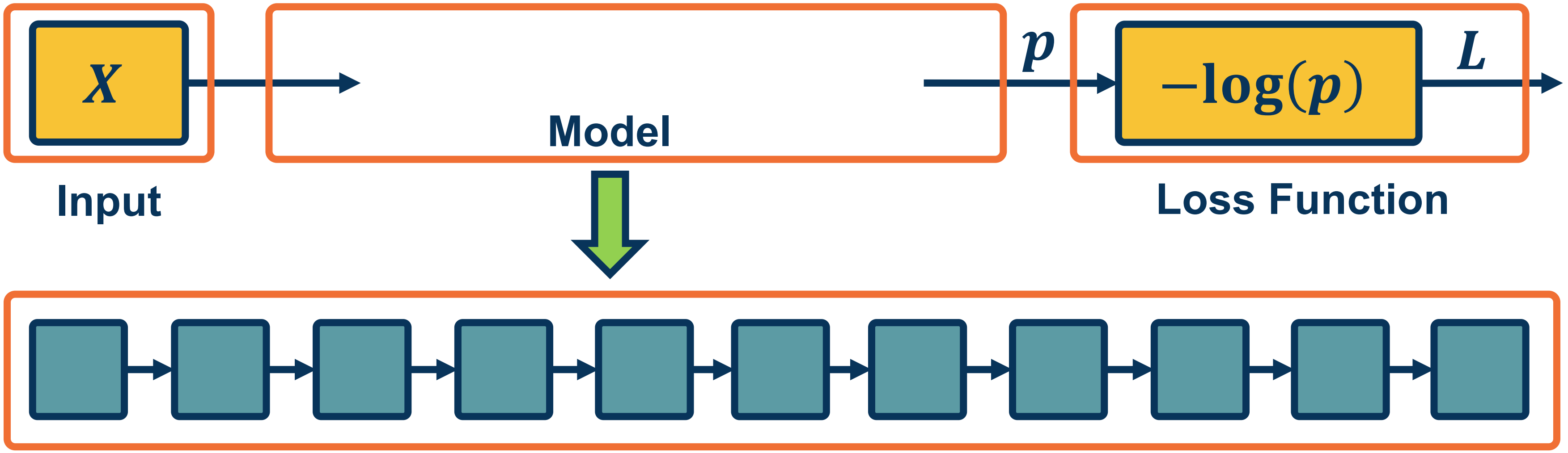


Adapted from figure by Andrej Karpathy

Power of Automatic Differentiation

Backpropagation, and automatic differentiation, allows us to optimize **any** function composed of differentiable blocks

- **No need to modify** the learning algorithm!
- The complexity of the function is only limited by **computation and memory**

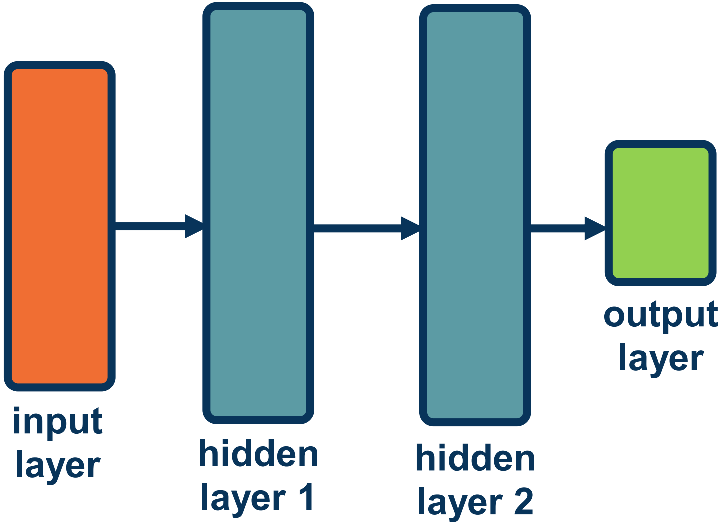


The Power of Deep Learning

A network with two or more hidden layers is often considered a **deep** model

Depth is important:

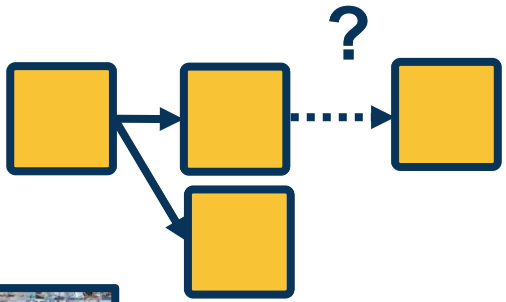
- Structure the model to represent an inherently compositional world
- Theoretical evidence that it leads to parameter efficiency
- Gentle dimensionality reduction (if done right)

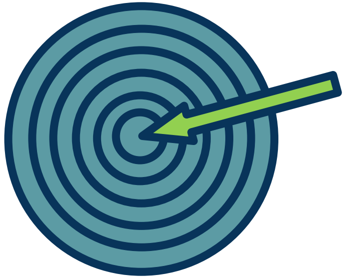


Importance of Depth

There are still many design decisions that must be made:

- **Architecture**
- **Data Considerations**
- **Training and Optimization**
- **Machine Learning Considerations**





Local Minima