

Topics:

- Jacobians
- Optimization

CS 4803-DL / 7643-A
ZSOLT KIRA

- **Assignment Due Feb 5th**
- Resources:
 - These lectures
 - [Matrix calculus for deep learning](#)
 - [Gradients notes](#) and [MLP/ReLU Jacobian notes](#).
 - [Assignment](#) (@41) and [matrix calculus](#) (@46)
- **Project:** Teaming thread on piazza

- **Schedule:**

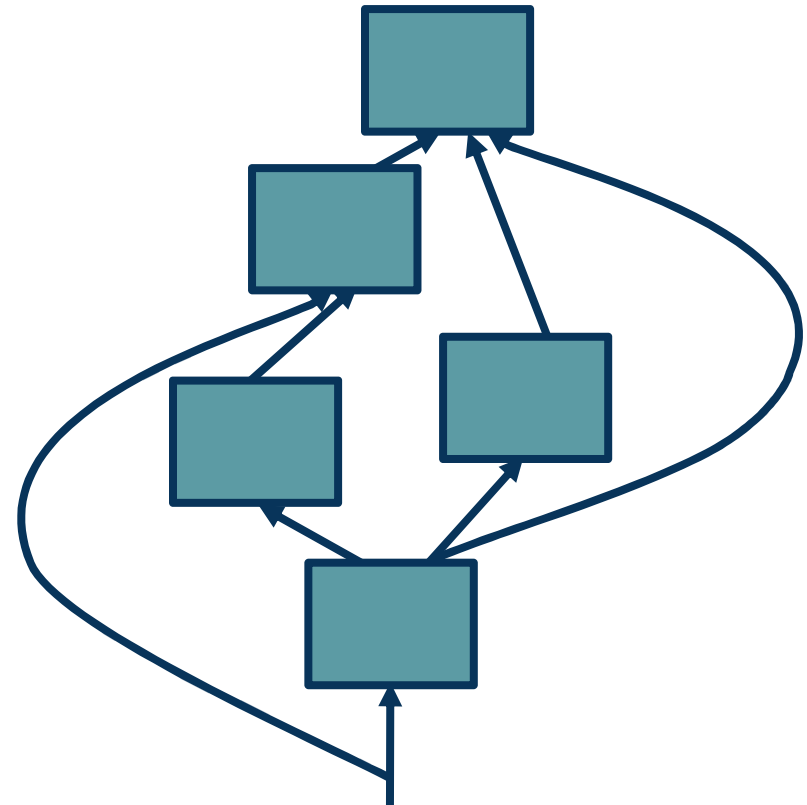
Graded Deliverabl	Release Date /	Due Date ES
Start of Term	10-Jan	
A1	14-Jan	5-Feb
A2	30-Jan	20-Feb
A3	20-Feb	6-Mar
Project Proposal	5-Feb	13-Mar
A4	9-Mar	3-Apr
Project Report	5-Apr	1-May

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

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**

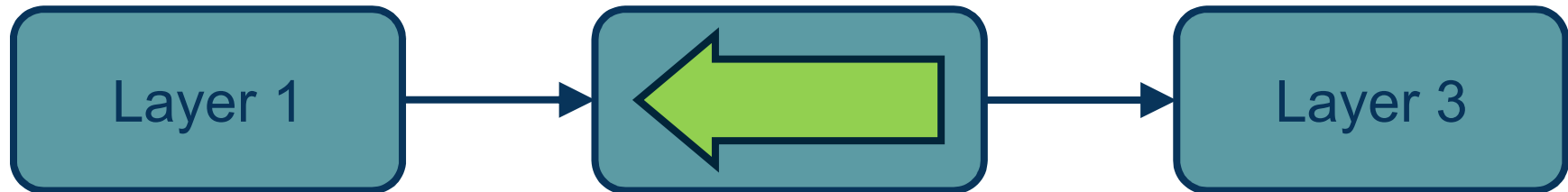
Step 2: Compute Gradients wrt parameters: **Backward Pass**



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



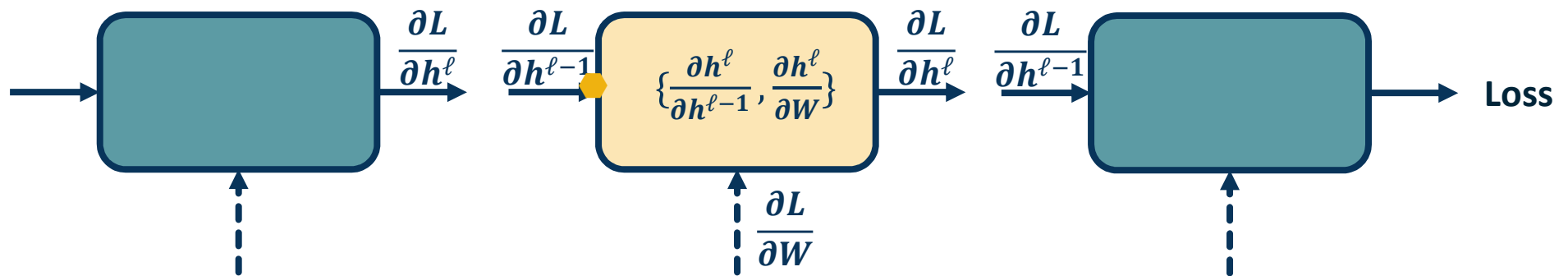
$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

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

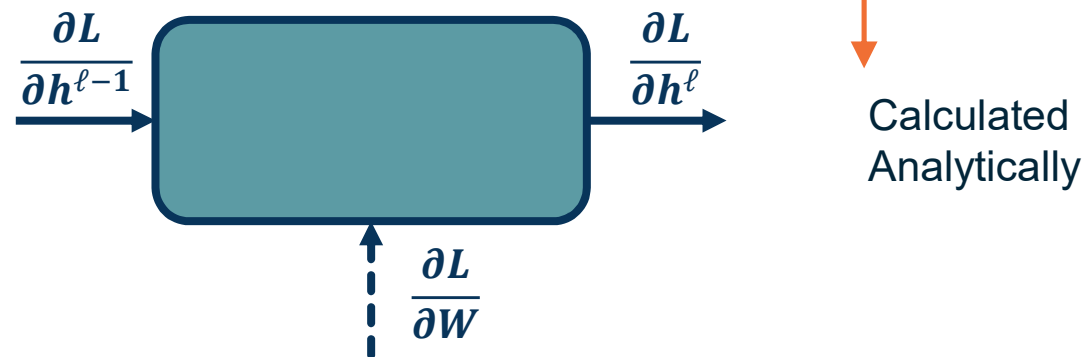


- We will use the *chain rule* to do this:

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

- 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:** $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Computing the Gradients of Loss

Conventions:

- Size of derivatives for scalars, vectors, and matrices:

Assume we have scalar $s \in \mathbb{R}^1$, vector $\mathbf{v} \in \mathbb{R}^m$, i.e. $\mathbf{v} = [v_1, v_2, \dots, v_m]^T$ and matrix $\mathbf{M} \in \mathbb{R}^{k \times \ell}$

	s $[\]$	\mathbf{v} $\left[\begin{array}{c} \end{array} \right]$	\mathbf{M} $\left[\begin{array}{cc} \end{array} \right]$
s	$\frac{\partial s_1}{\partial s_2}$ $[\]$	$\frac{\partial s}{\partial \mathbf{v}}$ $\left[\begin{array}{c} \end{array} \right]$	$\frac{\partial s}{\partial \mathbf{M}}$ $\left[\begin{array}{cc} \end{array} \right]$
\mathbf{v}	$\frac{\partial \mathbf{v}}{\partial s}$ $\left[\begin{array}{c} \end{array} \right]$	$\frac{\partial v_1}{\partial v_2}$ $\left[\begin{array}{c} \end{array} \right]$	Tensors
\mathbf{M}	$\frac{\partial \mathbf{M}}{\partial s}$ $\left[\begin{array}{cc} \end{array} \right]$		

$$\begin{aligned}
 \underline{x} \in \mathbb{R}^1 &\xrightarrow{g_1(\cdot)} z \in \mathbb{R}^1 \xrightarrow{g_2(\cdot)} \underline{y} \in \mathbb{R}^1 \quad \text{loss} \\
 &\quad \quad \quad y = g_2(g_1(x))
 \end{aligned}$$

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

Scalar
mult

Scalar Case

$$\begin{array}{ccccc} \vec{x} \in \mathbb{R}^d & \xrightarrow{g_1(\cdot)} & \vec{z} \in \mathbb{R}^m & \xrightarrow{g_2(\cdot)} & y \in \mathbb{R}^c \\ & \mathbb{R}^d \rightarrow \mathbb{R}^m & & \mathbb{R}^m \rightarrow \mathbb{R}^c & \end{array}$$

$$\left[\frac{\partial y}{\partial \vec{x}} \right] = \overset{\text{matrix}}{\left[\frac{\partial y}{\partial \vec{z}} \right]} \cdot \overset{\text{matrix}}{\left[\frac{\partial \vec{z}}{\partial \vec{x}} \right]}$$

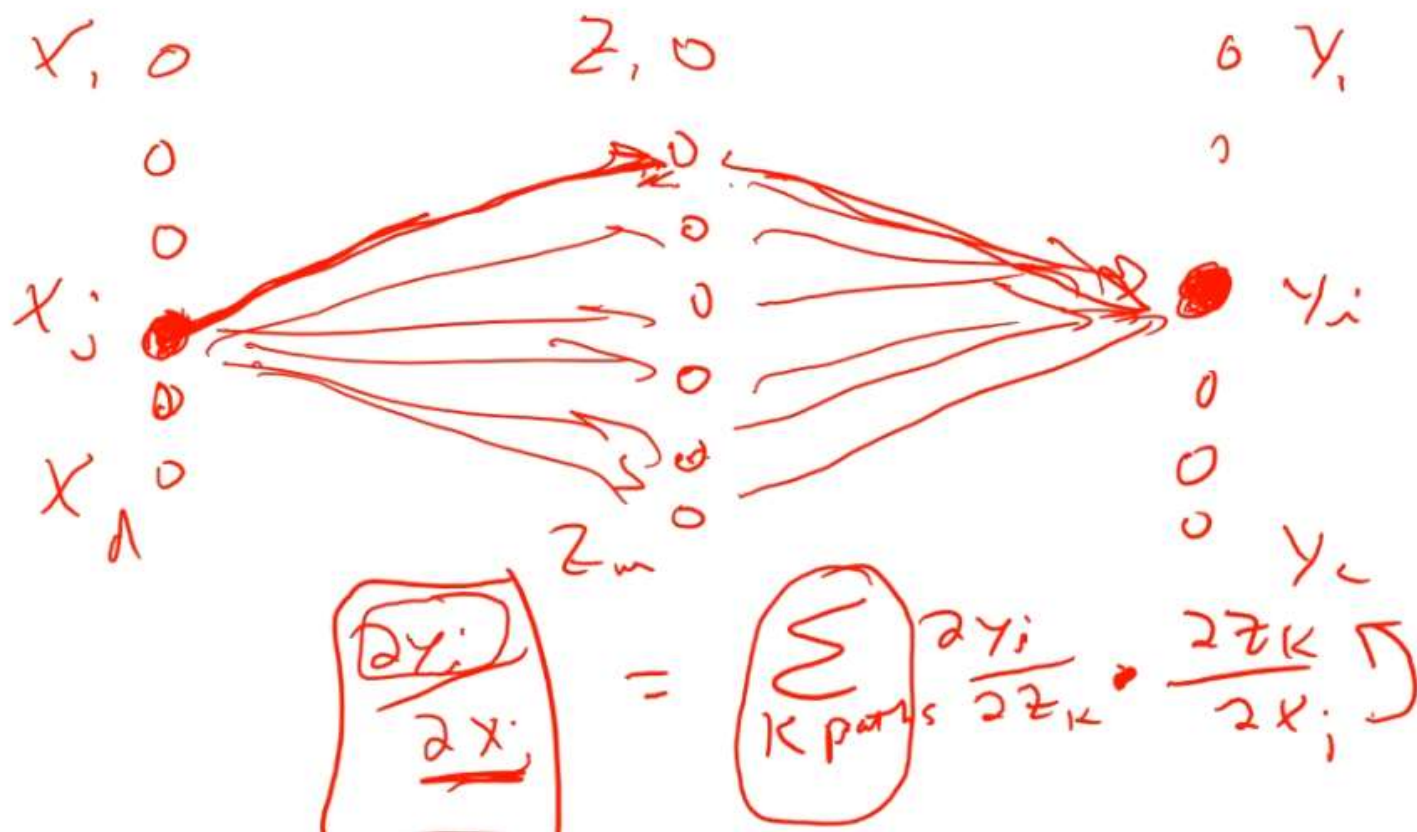
$$J_{g_2 \circ g_1} = J_{g_2} \cdot J_{g_1}$$

Vector Case

$$\begin{array}{c} \frac{\partial \vec{y}}{\partial \vec{x}} \\ \vdots \\ \boxed{\frac{\partial y_i}{\partial x_j}} \end{array} = \begin{array}{c} \frac{\partial \vec{y}}{\partial \vec{z}} \quad \frac{\partial \vec{z}}{\partial \vec{x}} \\ \left[\begin{array}{c} \text{---} \xrightarrow{\frac{\partial y_i}{\partial z_k}} \text{---} \end{array} \right] \left[\begin{array}{c} \vdots \\ \boxed{\frac{\partial z_k}{\partial x_j}} \\ \vdots \end{array} \right] \end{array}$$

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

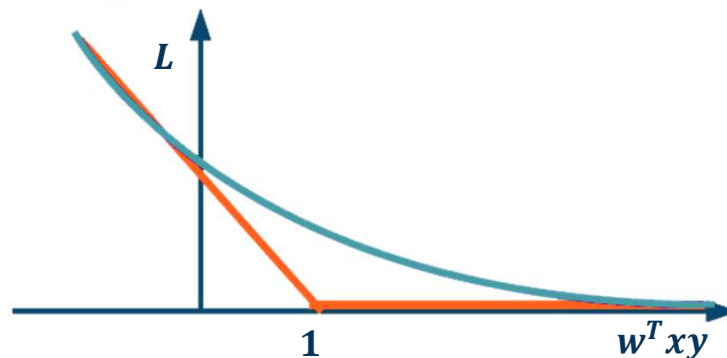
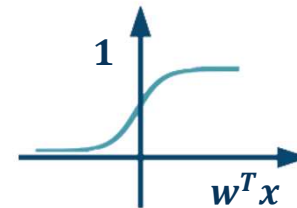
Jacobian View of Chain Rule



Graphical View of Chain Rule

Chain Rule: Cascaded

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



Log Loss

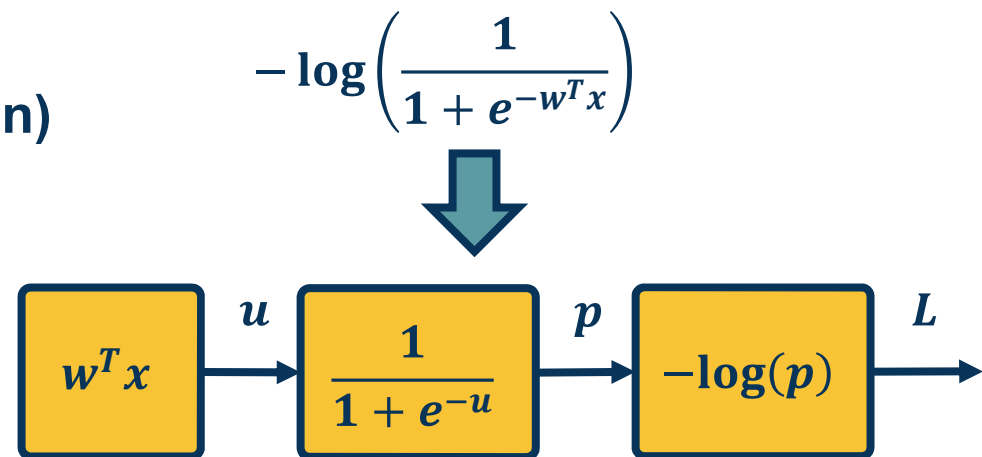
Adapted from slide by Marc'Aurelio Ranzato

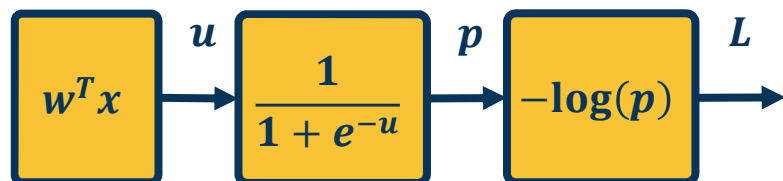
Linear Classifier: Logistic Regression

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!





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

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

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

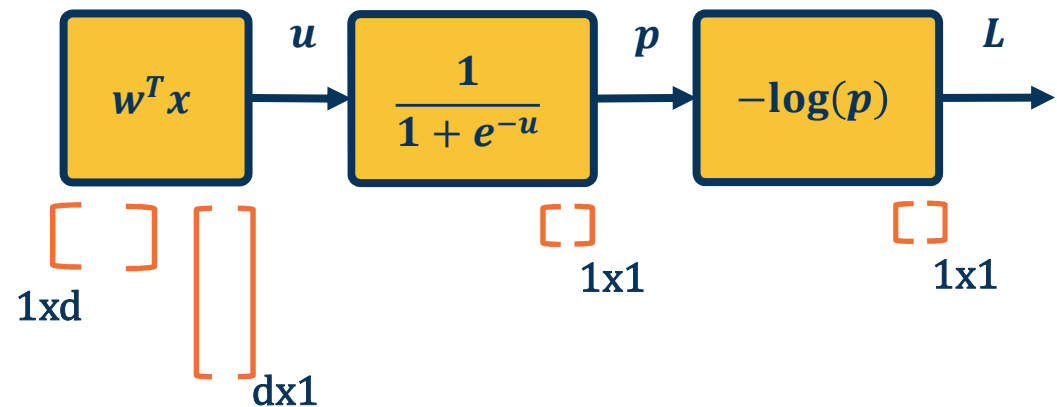
We can do this in a combined way to see all terms together:

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

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**

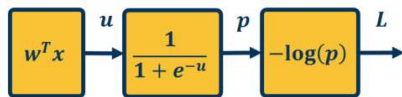


Extremely efficient in graphics processing units (GPUs)

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

The equation shows the vectorized computation of the weight update. The dimensions of the terms are indicated by brackets below them: $\frac{1}{\sigma(w^T x)}$ is 1×1 , $\sigma(w^T x)$ is 1×1 , $(1 - \sigma(w^T x))$ is 1×1 , and x^T is $1 \times d$.

Vectorized Computations



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

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

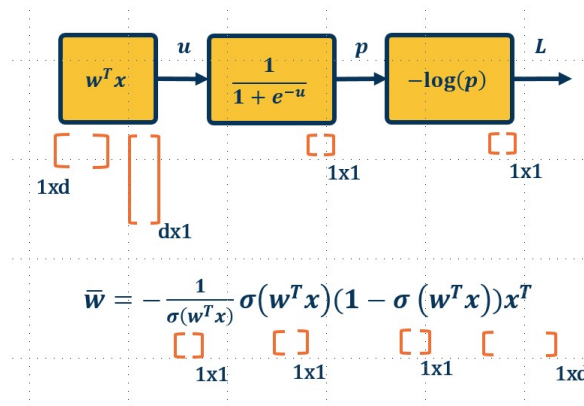
$$\bar{u} = \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} = \bar{p} \sigma(1 - \sigma)$$

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

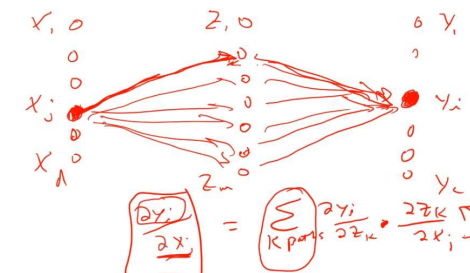
We can do this in a combined way to see all terms together:

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

This effectively shows gradient flow along path from L to w



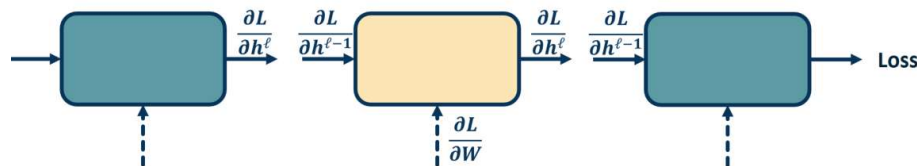
Computational / Tensor View



Graph View

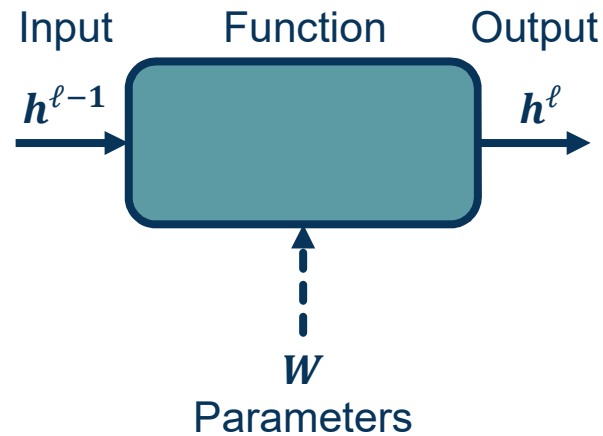
Computation Graph / Global View of Chain Rule

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



Backpropagation View
(Recursive Algorithm)

Different Views of Equivalent Ideas



Define:

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

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

$$\begin{array}{c}
 \left[\begin{array}{c} | \\ | \\ | \end{array} \right] \quad \left[\begin{array}{c} \leftarrow w_i^T \rightarrow \\ | \\ | \\ | \end{array} \right] \quad \left[\begin{array}{c} | \\ | \\ | \end{array} \right] \\
 |h^{\ell}| \times 1 \quad |h^{\ell}| \times |h^{\ell-1}| \quad |h^{\ell-1}| \times 1
 \end{array}$$

Fully Connected (FC) Layer: Forward Function

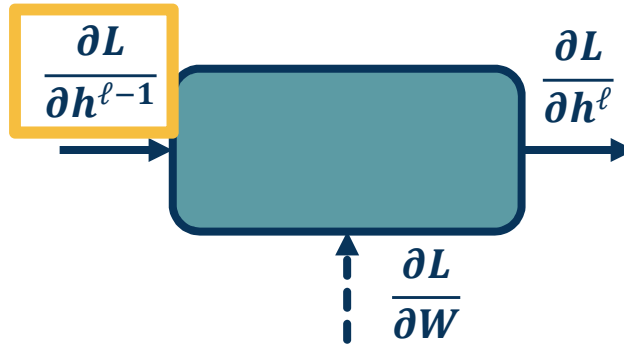
$$\mathbf{h}^\ell = \mathbf{W}\mathbf{h}^{\ell-1}$$

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

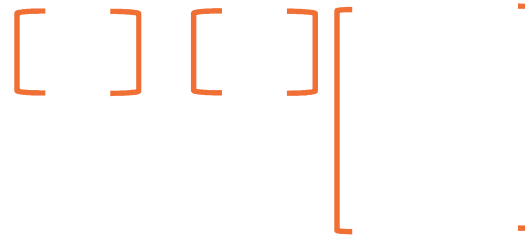
Define:

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

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



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



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

Fully Connected (FC) Layer

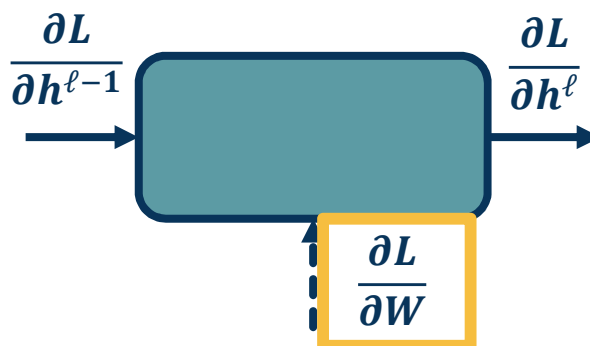
$$h^\ell = Wh^{\ell-1}$$

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

Define:

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

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



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

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

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

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

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

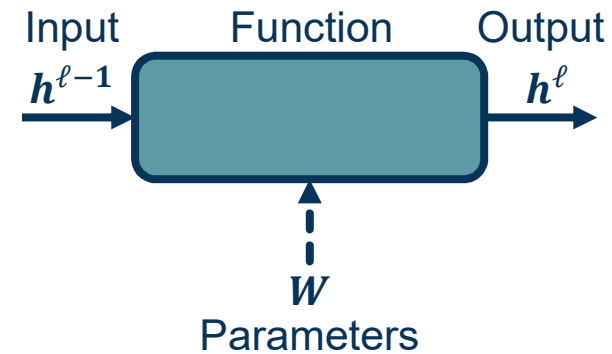
Fully Connected (FC) Layer

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

Diagram illustrating the Jacobian matrix structure for the ReLU layer. The matrix is shown as a square with a diagonal line, indicating that only the diagonal elements are non-zero. The matrix is labeled with dimensions $|h^{\ell} \times h^{\ell-1}|$.

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

Backpropagation and Automatic Differentiation

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

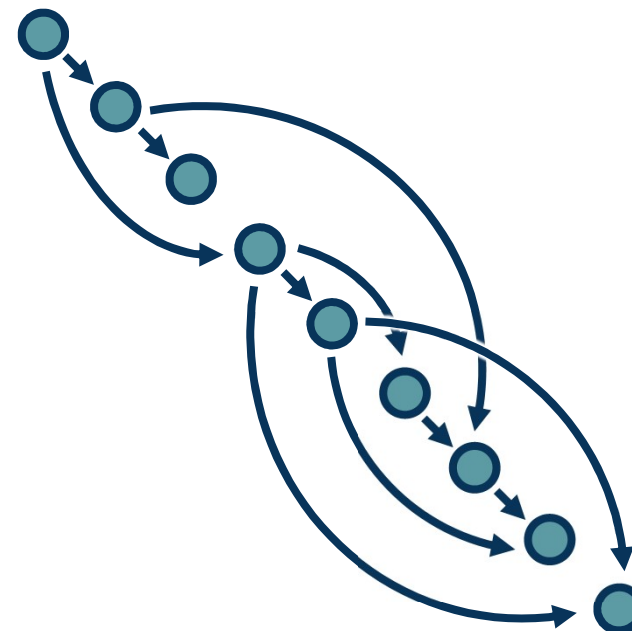
But the idea can be applied to **any directed acyclic graph (DAG)**

- 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

Computation = Graph

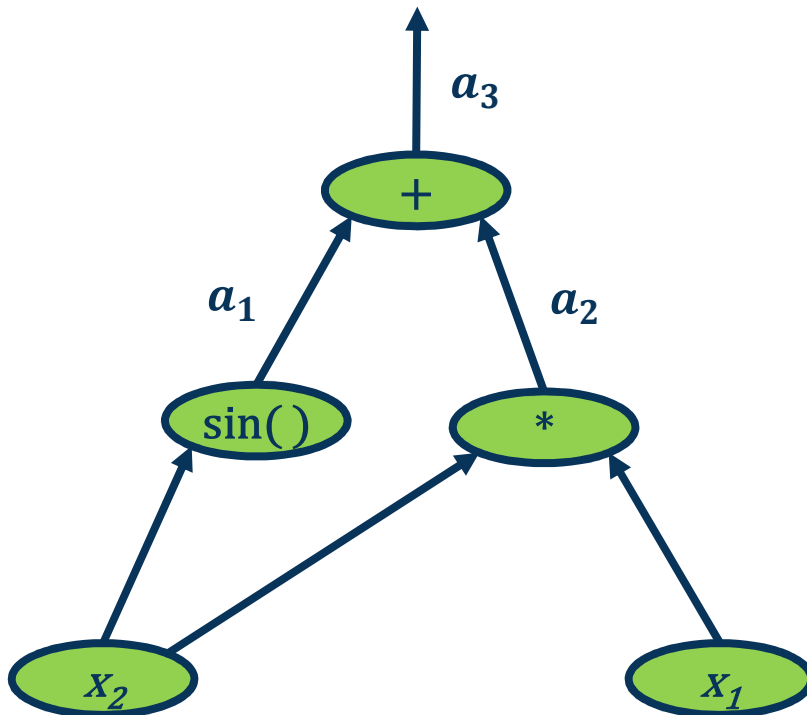
- ◆ Input = Data + Parameters
- ◆ Output = Loss
- ◆ Scheduling = Topological ordering

Auto-Diff

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

Deep Learning = Differentiable Programming

$$f(x_1, x_2) = x_1x_2 + \sin(x_2)$$



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

- Assign intermediate variables

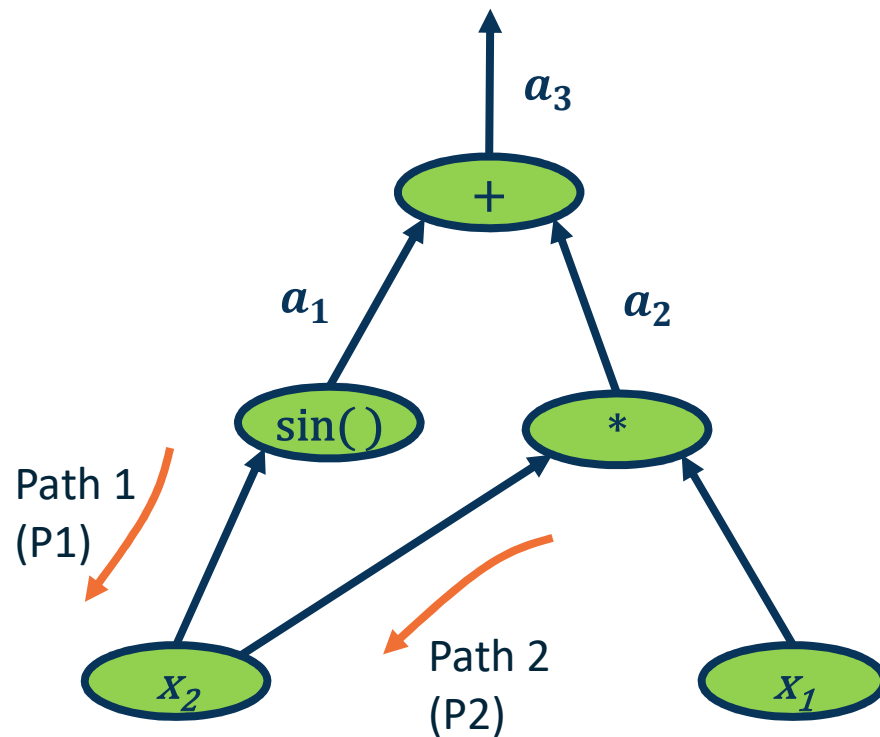
Simplify notation:

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

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

Example

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



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

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

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

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

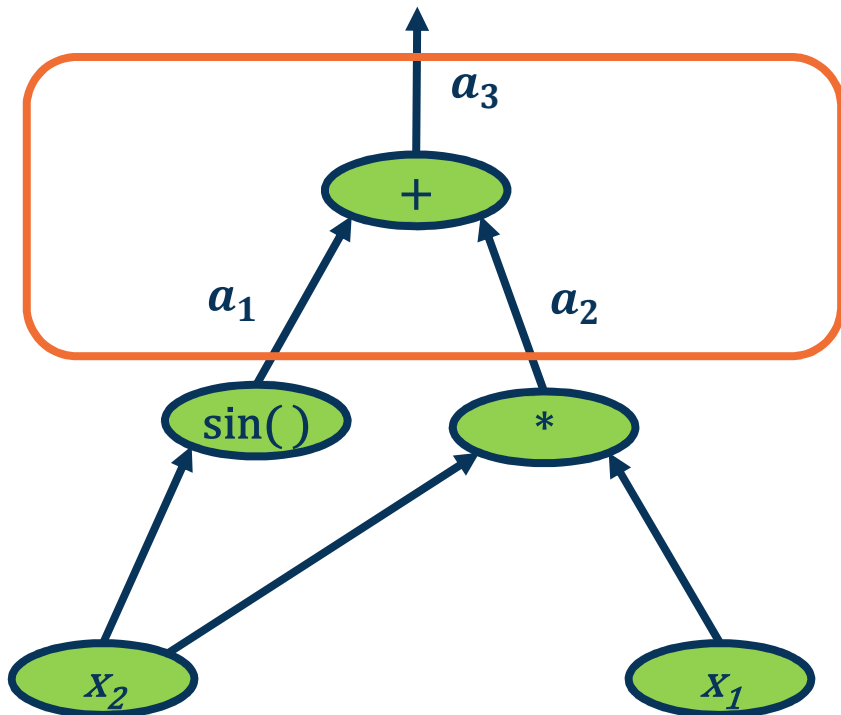
$$\overline{x_2^{P2}} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1 x_2)}{\partial x_2} = \overline{a_2} x_1$$

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

Gradients
from multiple
paths
summed

Example

$$f(x_1, x_2) = x_1x_2 + \sin(x_2)$$



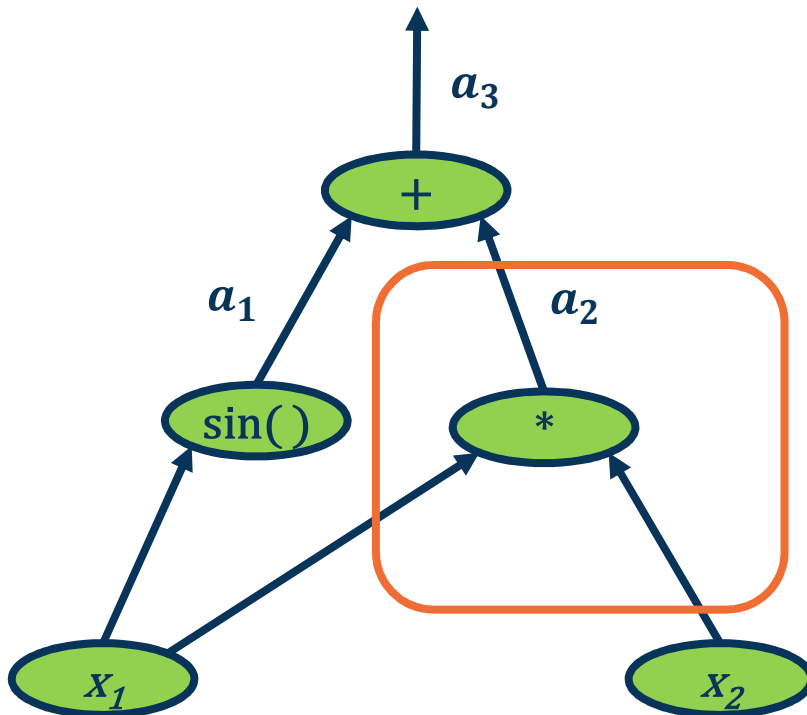
$$\overline{a_1} = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} \quad 1 = \overline{a_3}$$

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

Addition operation distributes gradients along all paths!

Patterns of Gradient Flow: Addition

$$f(x_1, x_2) = x_1x_2 + \sin(x_2)$$



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

$$\overline{x_2} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1x_2)}{\partial x_2} = \overline{a_2}x_1$$

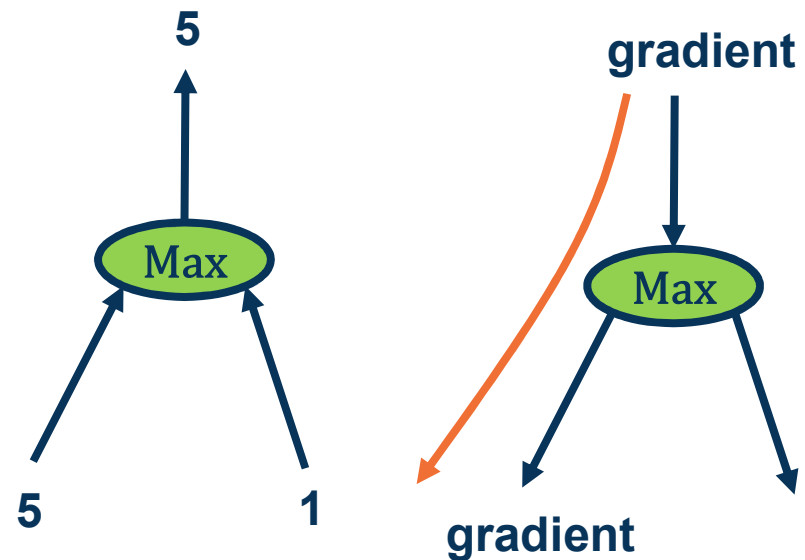
$$\overline{x_1} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = \overline{a_2}x_2$$

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

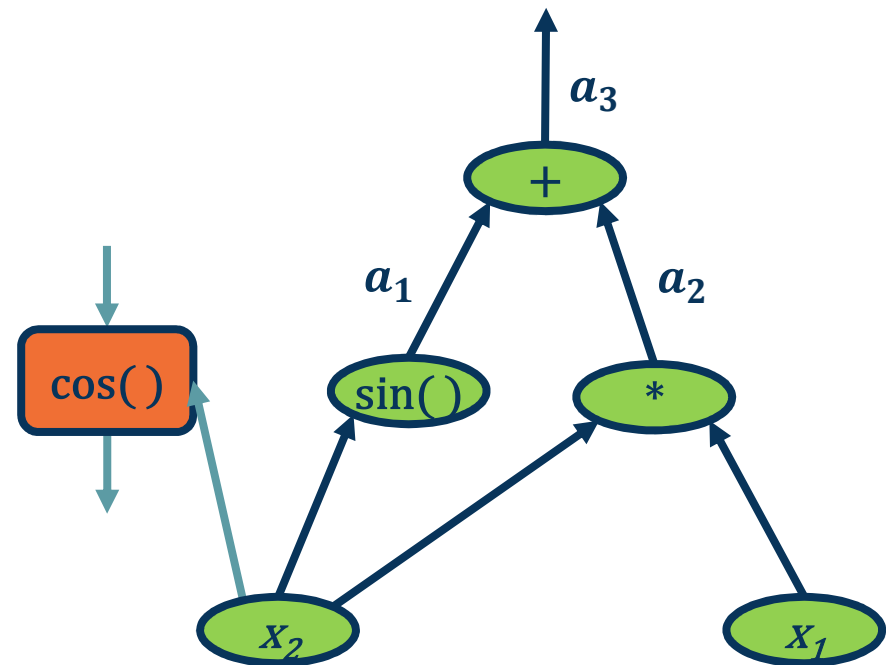


The flow of gradients is one of the **most important aspects** in deep neural networks

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

- 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**

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

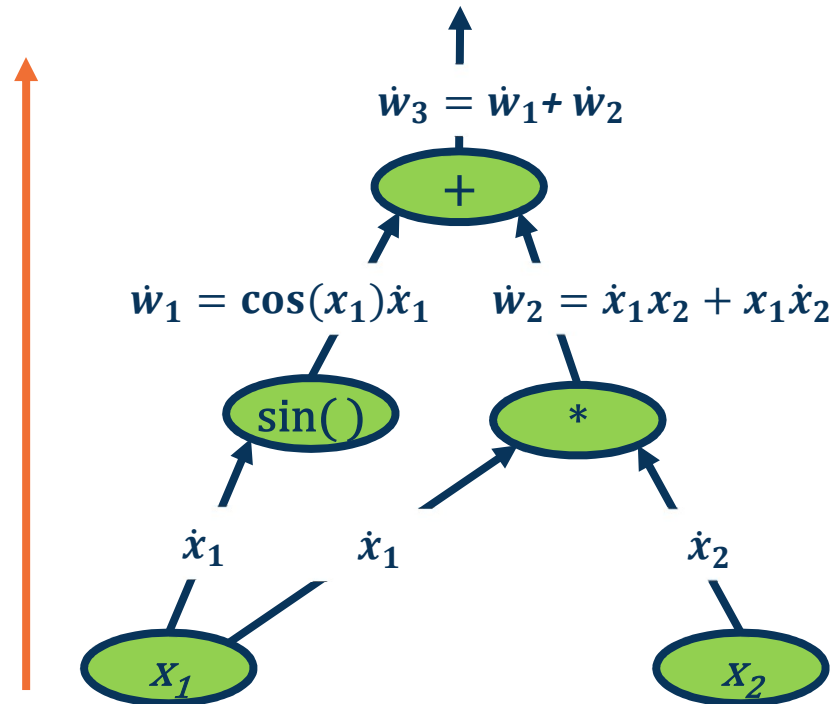


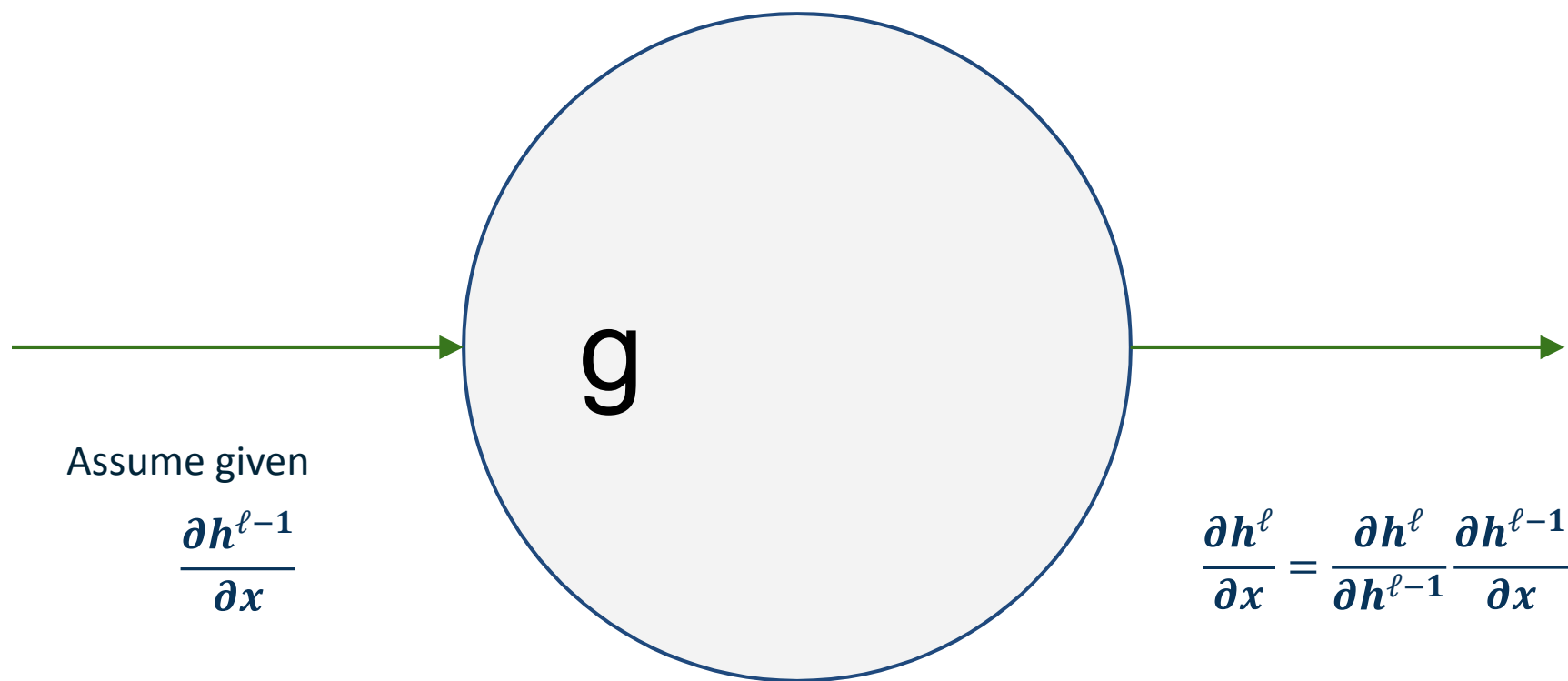
Note that we can also do **forward mode** automatic differentiation

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





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

Forward Mode Autodifferentiation

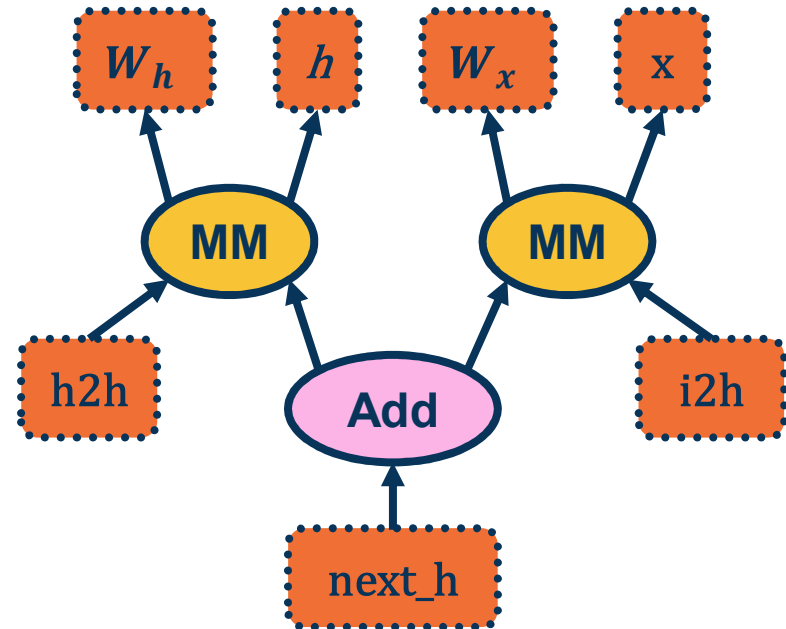
A graph is created on the fly

```
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)



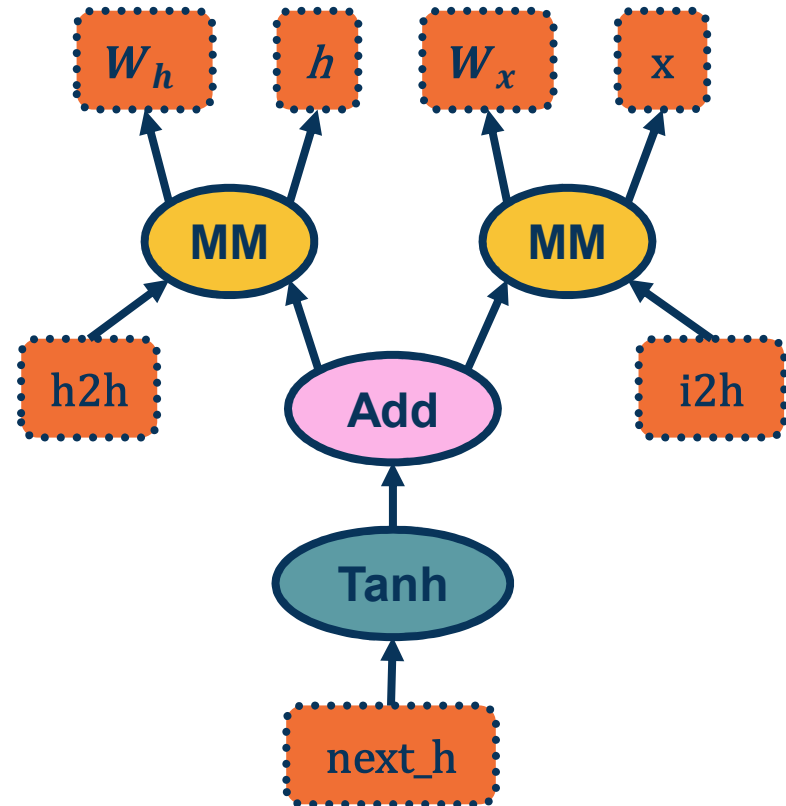
Back-propagation uses the dynamically built graph

```
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)

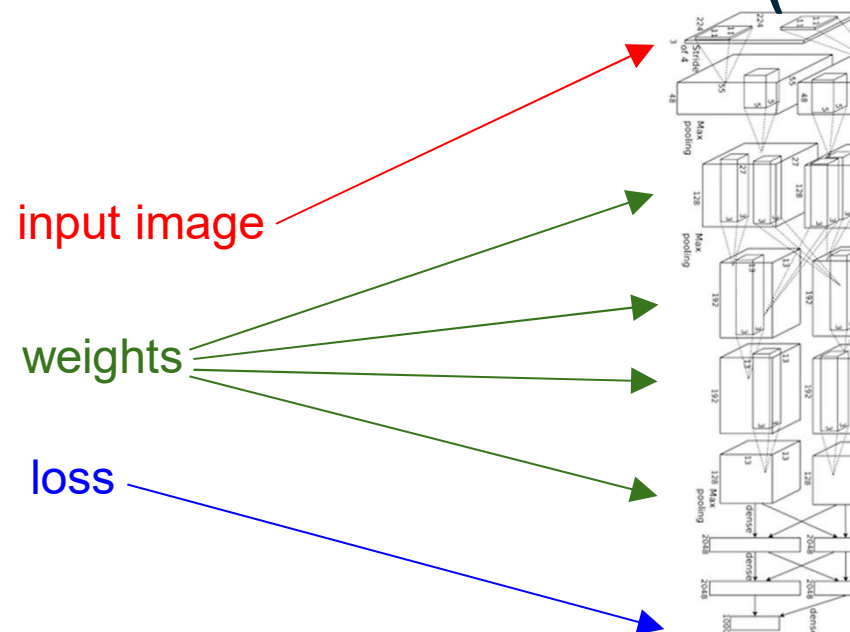


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Neural Turing Machine

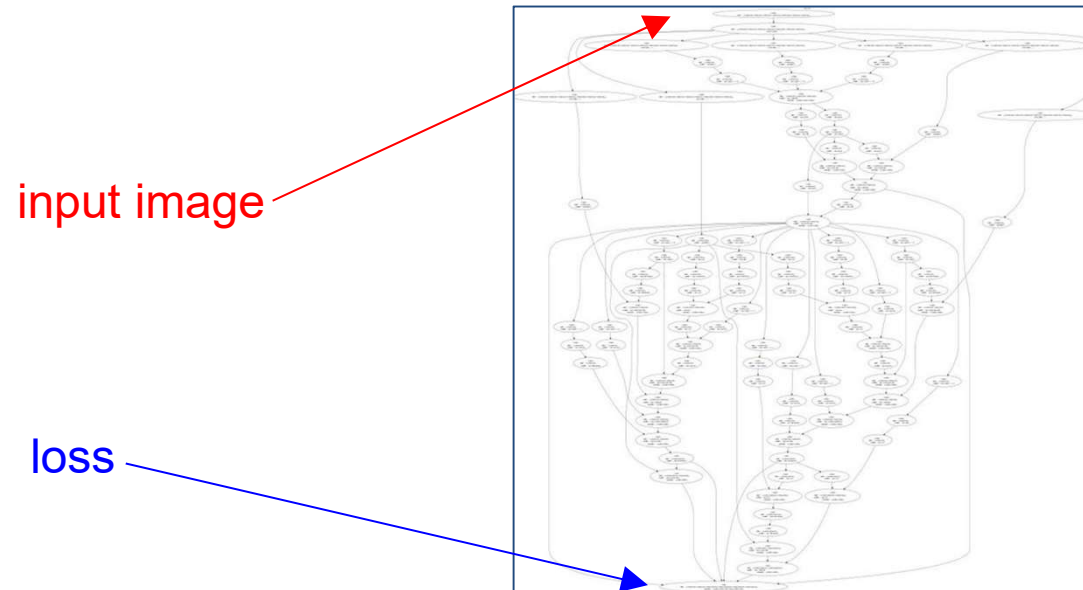
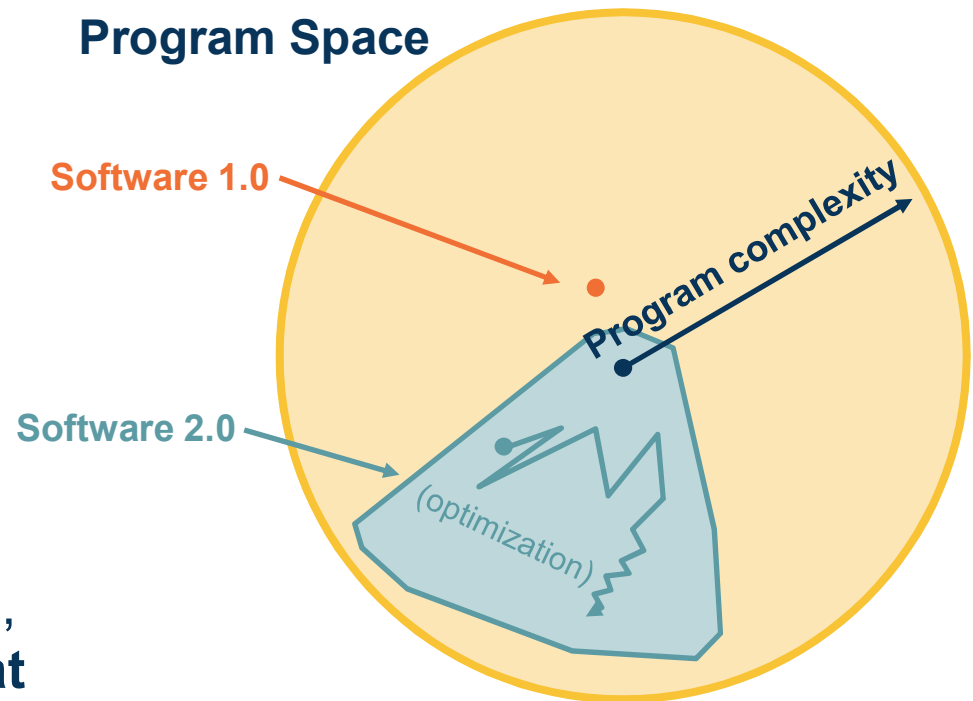


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

- 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**

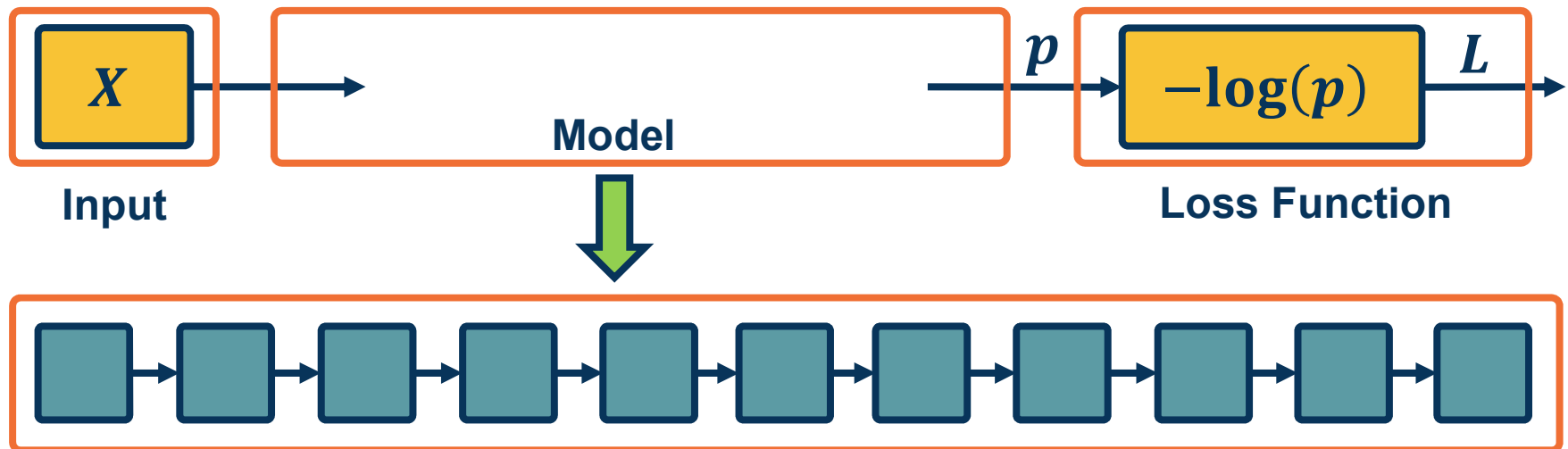


Adapted from figure by Andrej Karpathy

Optimization of Deep Neural Networks Overview

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**

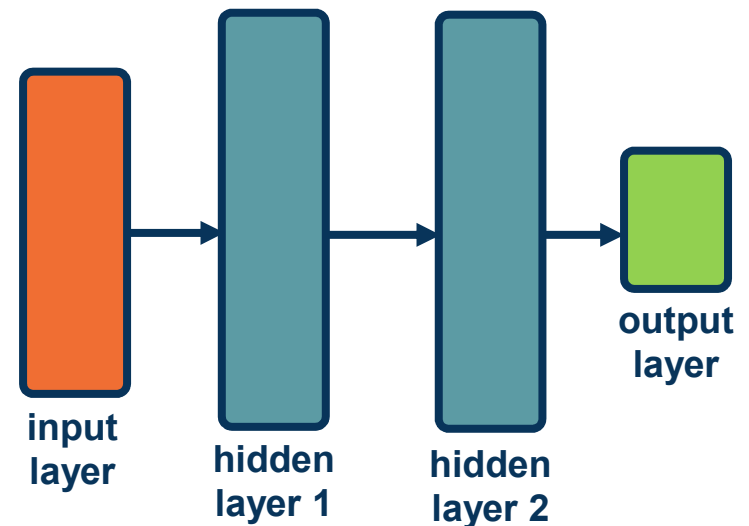


The Power of Deep Learning

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

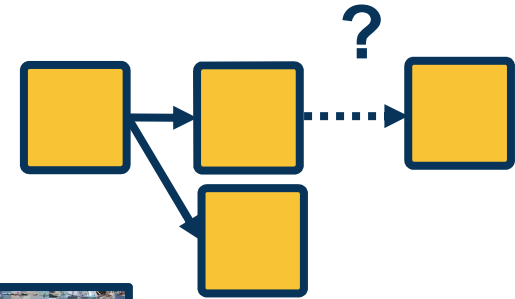
Depth is important:

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



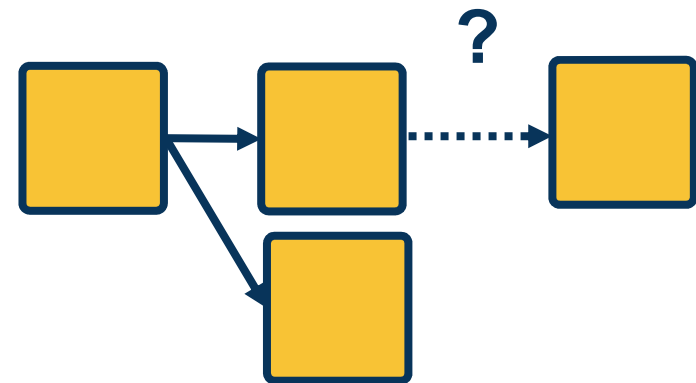
There are still many design decisions that must be made:

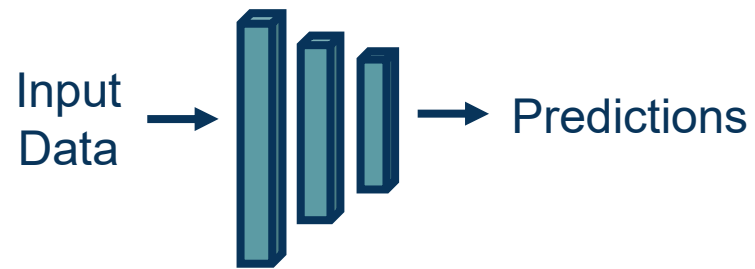
- ✦ **Architecture**
- ✦ **Data Considerations**
- ✦ **Training and Optimization**
- ✦ **Machine Learning Considerations**



We must design the **neural network architecture**:

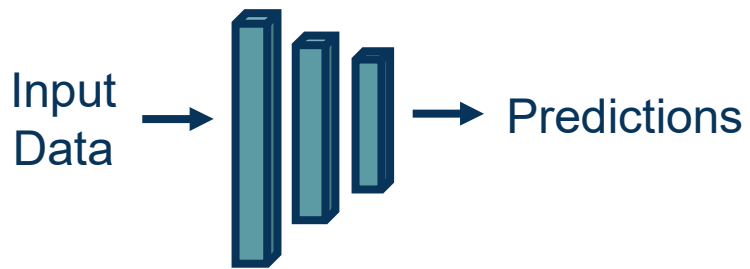
- What **modules (layers)** should we use?
- How should they **be connected together**?
- Can we use our **domain knowledge** to add architectural biases?



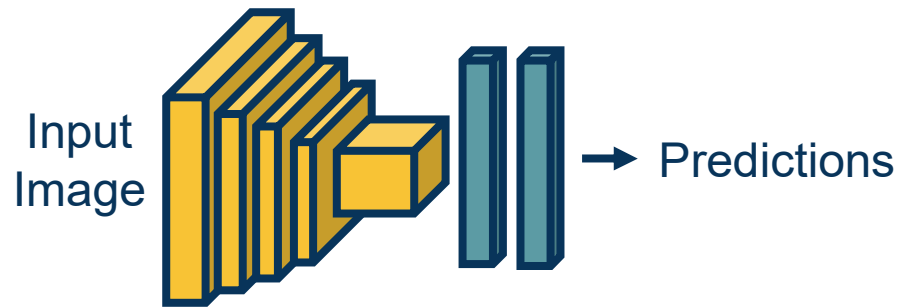


**Fully Connected
Neural Network**

Example Architectures

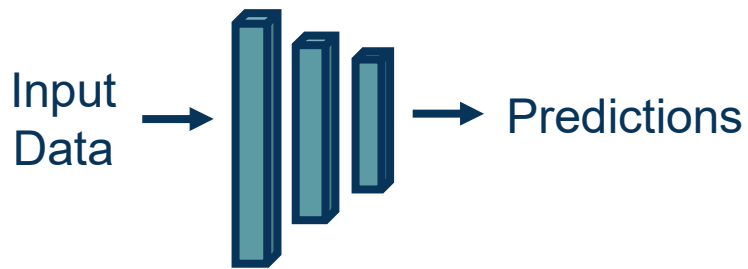


**Fully Connected
Neural Network**



**Convolutional Neural
Networks**

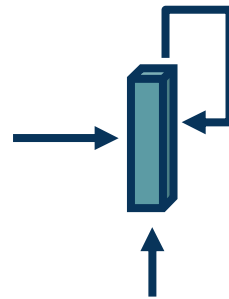
Example Architectures



**Fully Connected
Neural Network**



**Convolutional Neural
Networks**



Recurrent Neural Network

**Different architectures
are suitable for different
applications or types of
input**

Example Architectures

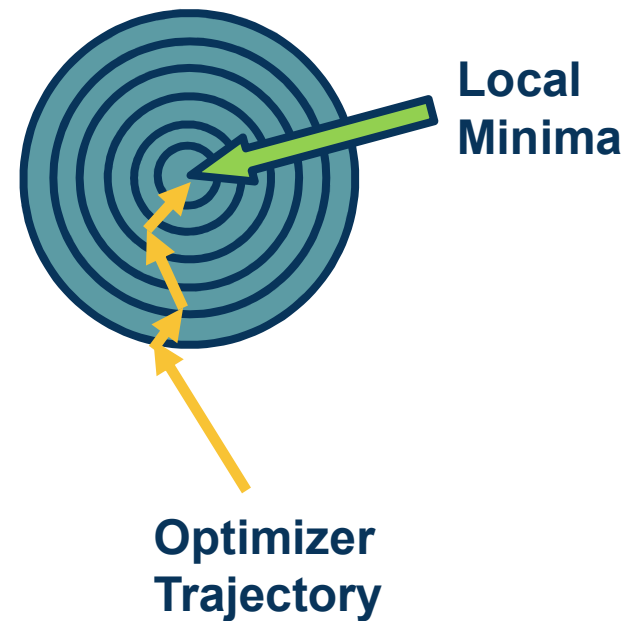
As in traditional machine learning, **data** is key:

- ◆ Should we **pre-process** the data?
- ◆ Should we **normalize** it?
- ◆ Can we **augment** our data by adding noise or other perturbations?



Even given a good neural network architecture, we need a **good optimization algorithm to find good weights**

- What **optimizer** should we use?
 - Different optimizers make **different weight updates** depending on the gradients
- How should we **initialize** the weights?
- What **regularizers** should we use?
- What **loss function** is appropriate?



Machine Learning Considerations

The practice of machine learning is **complex**: For your particular application you have to **trade off** all of the considerations together

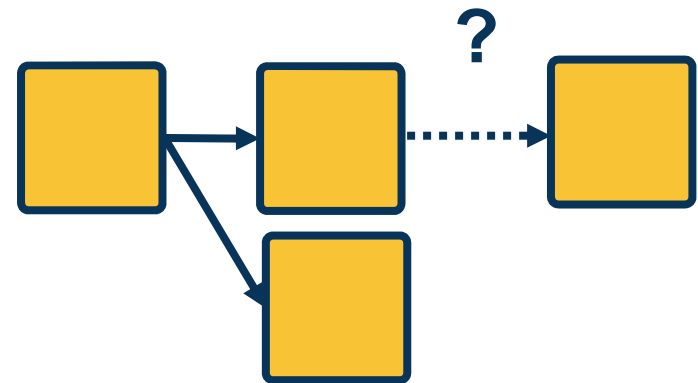
- Trade-off between **model capacity** (e.g. measured by # of parameters) and **amount of data**
- Adding **appropriate biases** based on knowledge of the domain



Architectural Considerations

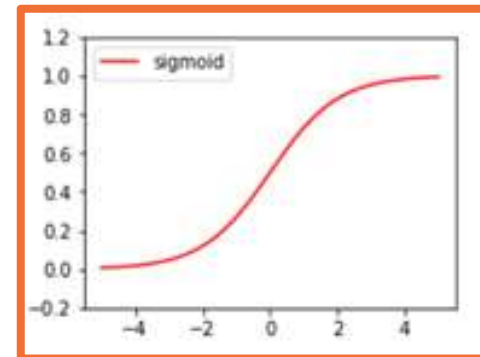
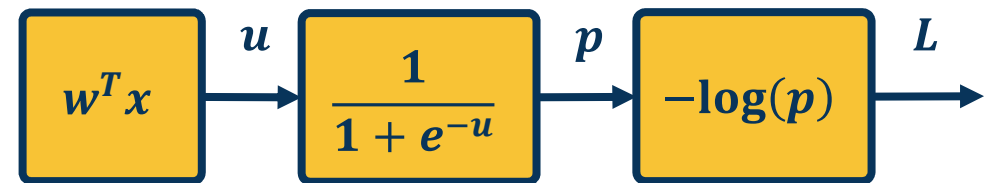
Determining what modules to use, and how to connect them is part of the **architectural design**

- ✦ Guided by the **type of data used** and its **characteristics**
 - ✦ Understanding your data is always the first step!
- ✦ **Lots of data types (modalities)** already have good architectures
 - ✦ Start with what others have discovered!
- ✦ **The flow of gradients** is one of the key principles to use when analyzing layers



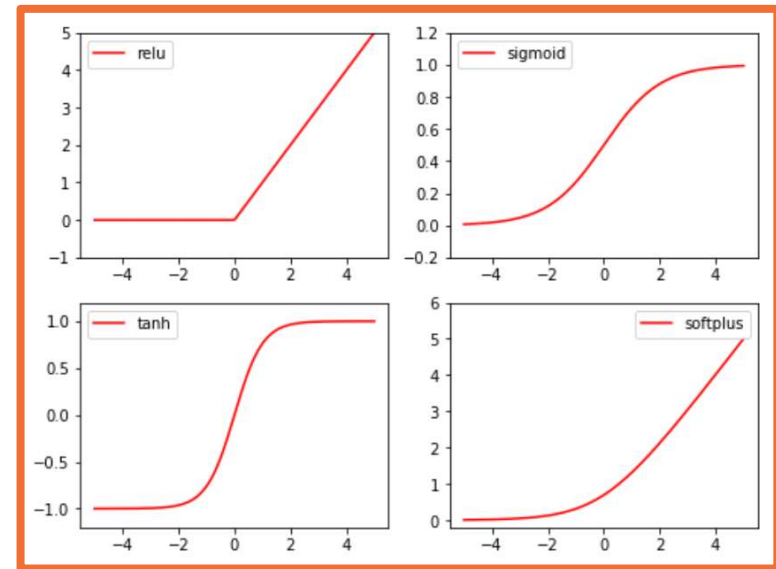
- Combination of linear and non-linear layers
- Combination of **only** linear layers has same representational power as one linear layer
- Non-linear layers** are crucial
 - Composition of non-linear layers **enables complex transformations of the data**

$$w_1^T(w_2^T(w_3^T x)) = w_4^T x$$



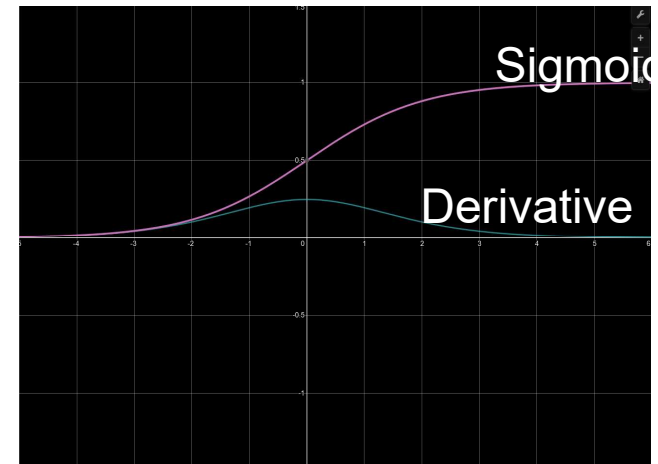
Several aspects that we can **analyze**:

- Min/Max
- Correspondence between input & output statistics
- **Gradients**
 - At initialization (e.g. small values)
 - At extremes
- Computational complexity



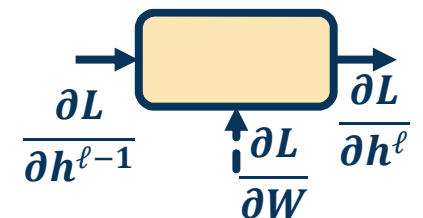
Analysis of Non-Linear Function

- Min: 0, Max: 1
- Output **always positive**
- Saturates at **both ends**
- Gradients**
 - Vanishes at both end
 - Always positive
- Computation: Exponential term**



$$h^\ell = \sigma(h^{\ell-1})$$

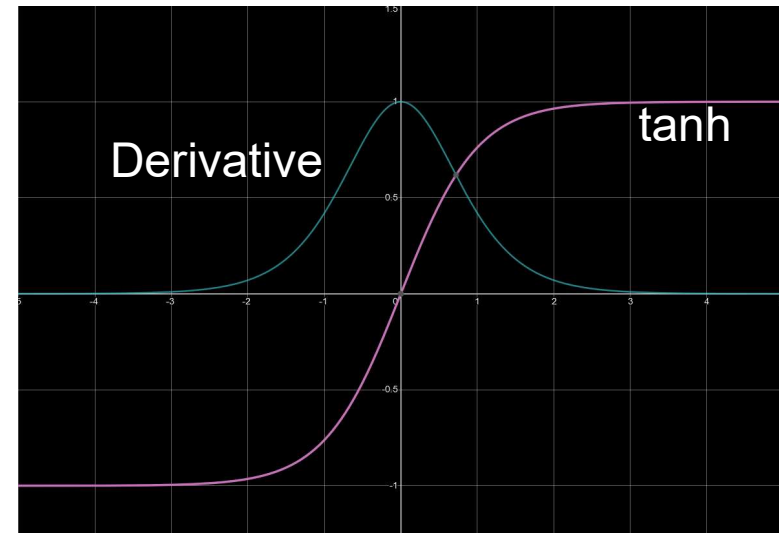
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial W}$$

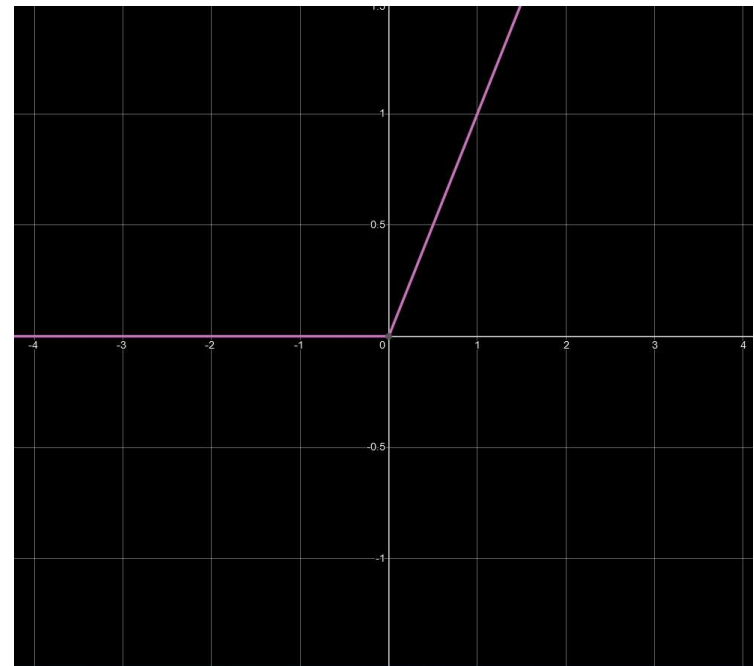
Sigmoid Function

- **Min: -1, Max: 1**
- **Centered**
- Saturates at **both ends**
- **Gradients**
 - Vanishes at both end
 - Always positive
- **Still somewhat computationally heavy**



$$h^\ell = \tanh(h^{\ell-1})$$

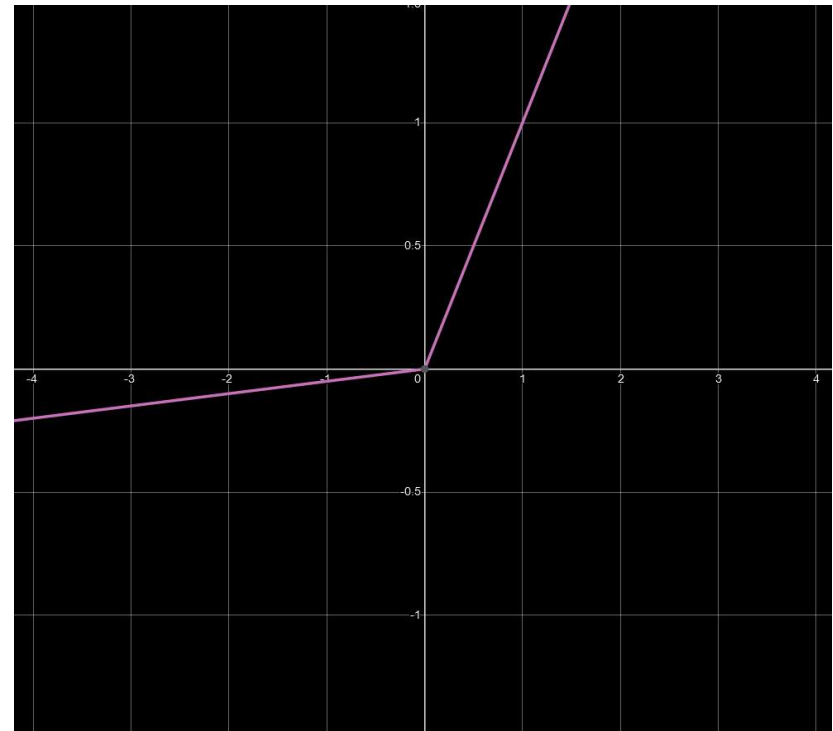
- ⬢ **Min: 0, Max: Infinity**
- ⬢ Output always **positive**
- ⬢ **No saturation** on positive end!
- ⬢ **Gradients**
 - ⬢ 0 if $x \leq 0$ (dead ReLU)
 - ⬢ Constant otherwise (does not vanish)
- ⬢ **Cheap to compute (max)**



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

Rectified Linear Unit

- ⬢ **Min: -Infinity, Max: Infinity**
- ⬢ **Learnable parameter!**
- ⬢ **No saturation**
- ⬢ **Gradients**
 - ⬢ No dead neuron
- ⬢ **Still cheap to compute**



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

Leaky ReLU

Selecting a Non-Linearity

Which **non-linearity** should you select?

- ✦ Unfortunately, **no one activation function is best** for all applications
- ✦ **ReLU** is most common starting point
 - ✦ Sometimes leaky ReLU can make a big difference
- ✦ **Sigmoid** is typically avoided unless clamping to values from $[0, 1]$ is needed



- **Backpropagation:** Recursive, modular algorithm for chain rule + gradient descent
- **When we move to vectors and matrices:**
 - Computation graph (composition of functions) => Multiplication of Jacobians along path
- **Automatic differentiation:**
 - Reduction of modules to simple operations we know (simple multiplication, etc.)
 - Automatically build computation graph in background as write code
 - Automatically compute gradients via backward pass
- **We now have a generic algorithm! Considerations:**
 - Architecture
 - Data Considerations
 - Training and Optimization
 - Machine Learning Considerations

Summary