Topics:

- Backpropagation
- Matrix/Linear Algebra view

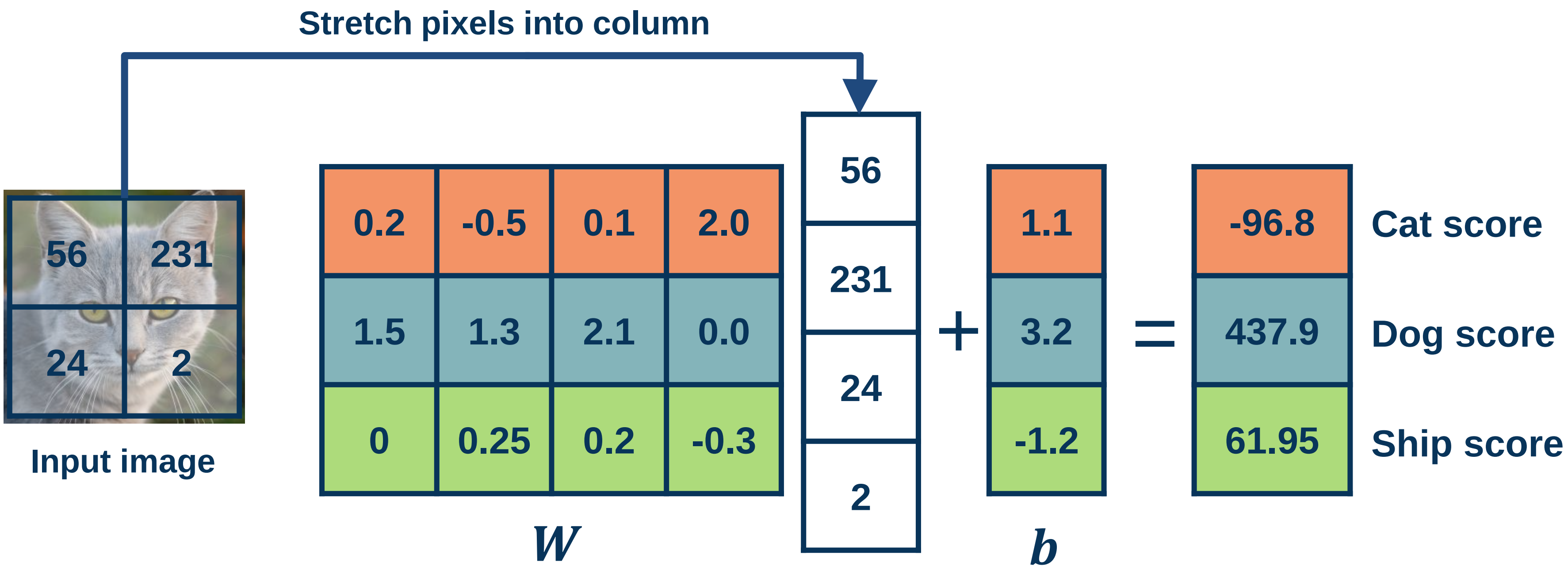
CS 4644-DL / 7643-A
ZSOLT KIRA

- **Assignment 1 out!**
  - **Due Feb 2nd (with grace period 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.
  - Assignment 1 (@57) and matrix calculus/computation graph (TBD)

- Piazza: Project teaming thread
  - Project proposal overview during my OH (Thursday 4pm ET)

Example with an image with **4 pixels**, and **3 classes** (**cat**/**dog**/**ship**)



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

- We can find the steepest descent direction by computing the **derivative (gradient):**

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

- Steepest descent direction is the **negative gradient**
- **Intuitively:** Measures how the function changes as the argument a changes by a small step size
  - As step size goes to zero
- **In Machine Learning:** Want to know how the **loss function** changes **as weights** are varied
  - Can consider each parameter separately by taking **partial derivative** of loss function with respect to that parameter

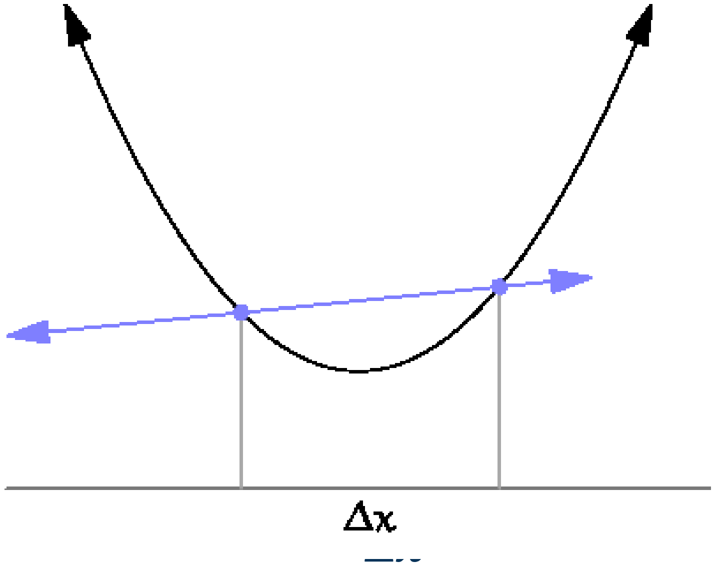
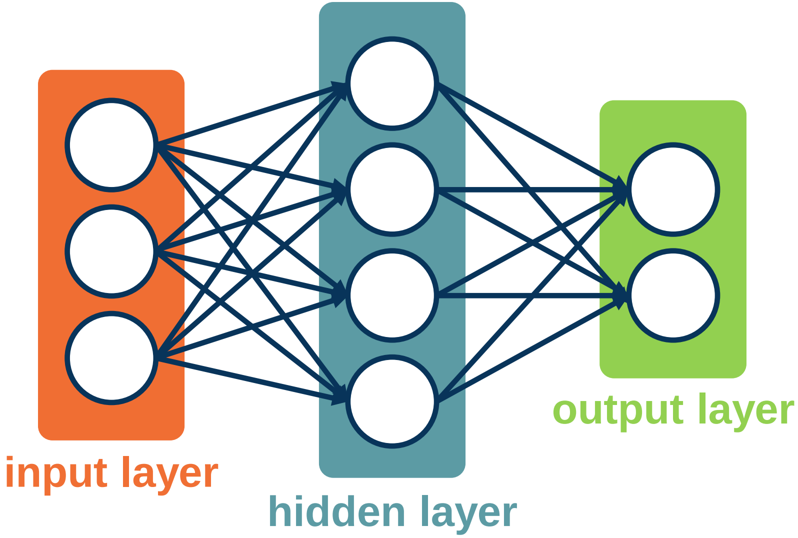


Image and equation from: https://en.wikipedia.org/wiki/Derivative#/media/File:Tangent_animation.gif

Derivatives

The same two-layered neural network **corresponds to adding another weight matrix**

- We will prefer the linear algebra view, but use some terminology from neural networks (& biology)



$$\boldsymbol{x} \quad \boldsymbol{W}_1 \quad \boldsymbol{W}_2$$

$$=$$

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{W}_2) = \boldsymbol{\sigma}(\boldsymbol{W}_2 \boldsymbol{\sigma}(\boldsymbol{W}_1 \boldsymbol{x}))$$

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

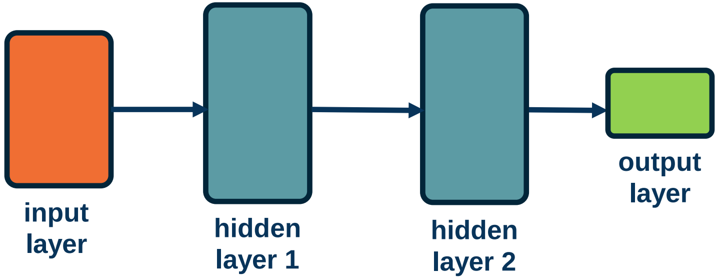
Adding More Layers!

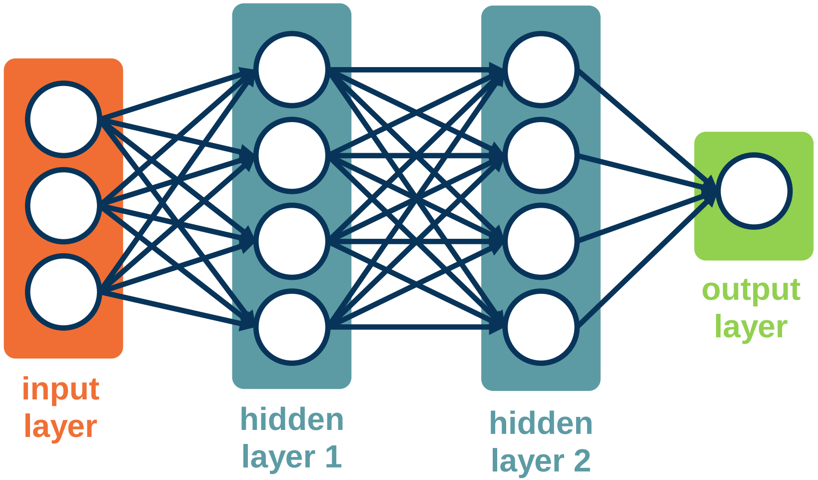
Large (deep) networks can be built by adding more and more layers

Three-layered neural networks can represent **any function**

- The number of nodes could grow unreasonably (exponential or worse) with respect to the complexity of the function

We will show them **without edges**:



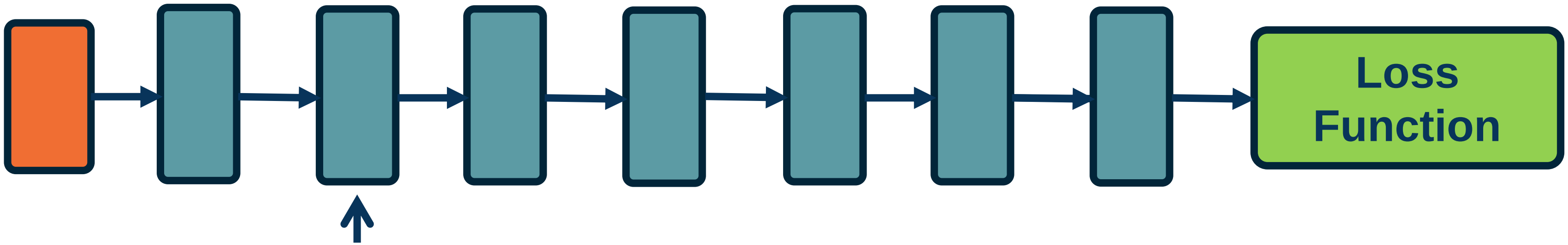


$$f(x, W_1, W_2, W_3) = \sigma(W_2\sigma(W_1x))$$

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

Computing Gradients in Complex Function

- We are learning **complex models** with significant amount of parameters (millions or billions)
- How do we compute the gradients of the **loss** (at the end) with respect to **internal** parameters?
- Intuitively, want to understand how **small changes** in weight deep inside **are propagated** to affect the **loss function** at the end



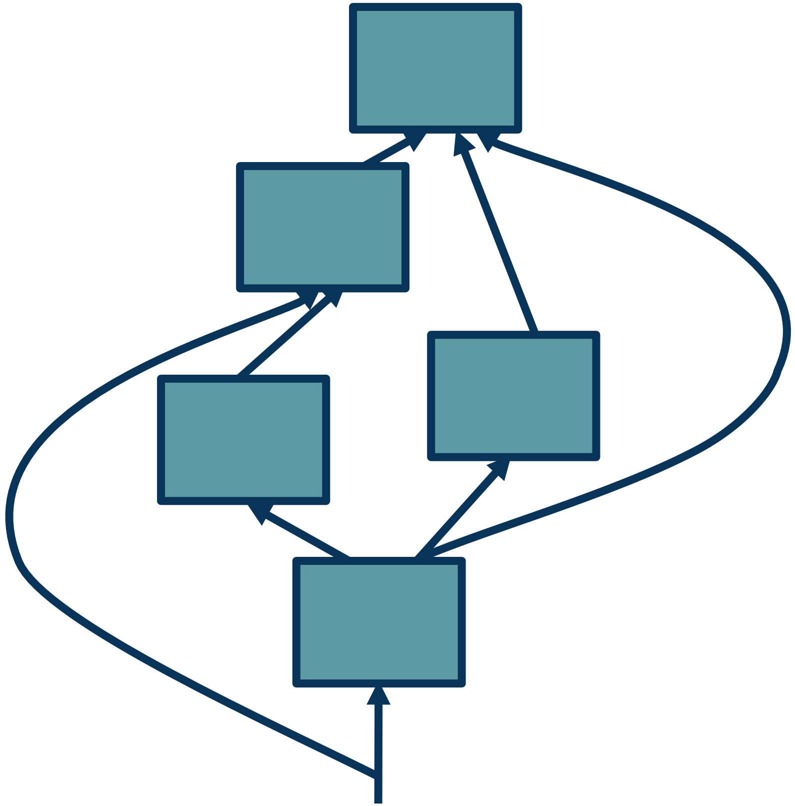
$$\frac{\partial L}{\partial w_i}?$$

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

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

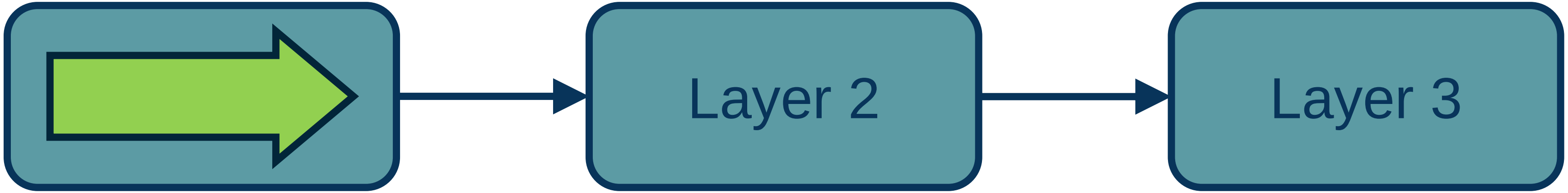
- Modules must be differentiable to support gradient computations for gradient descent

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



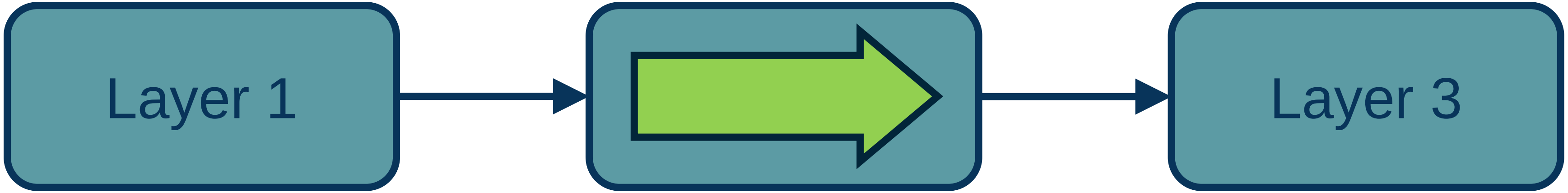
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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



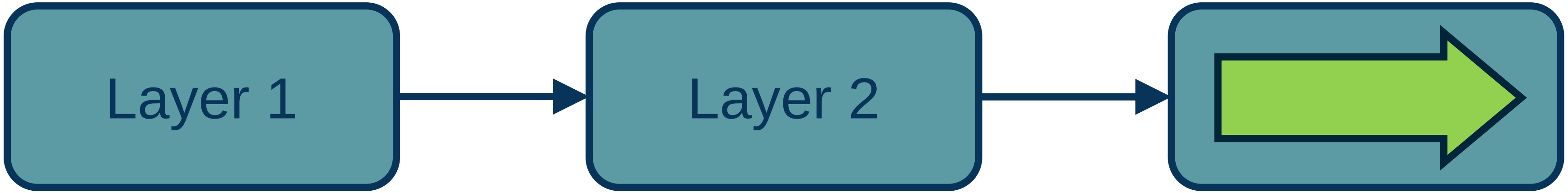
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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**



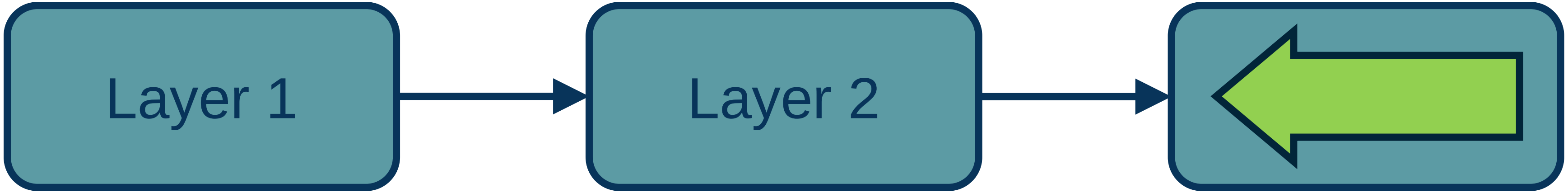
Note that we must store the **intermediate outputs of all layers**!

- 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**

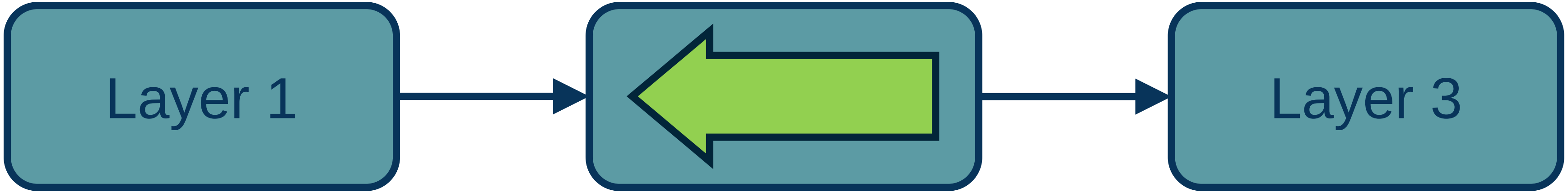
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

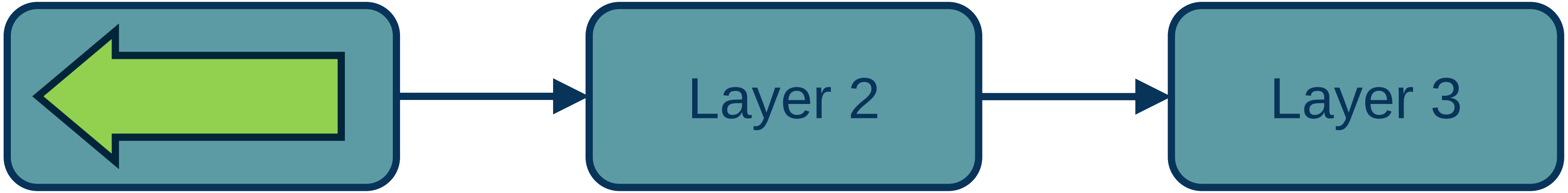
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

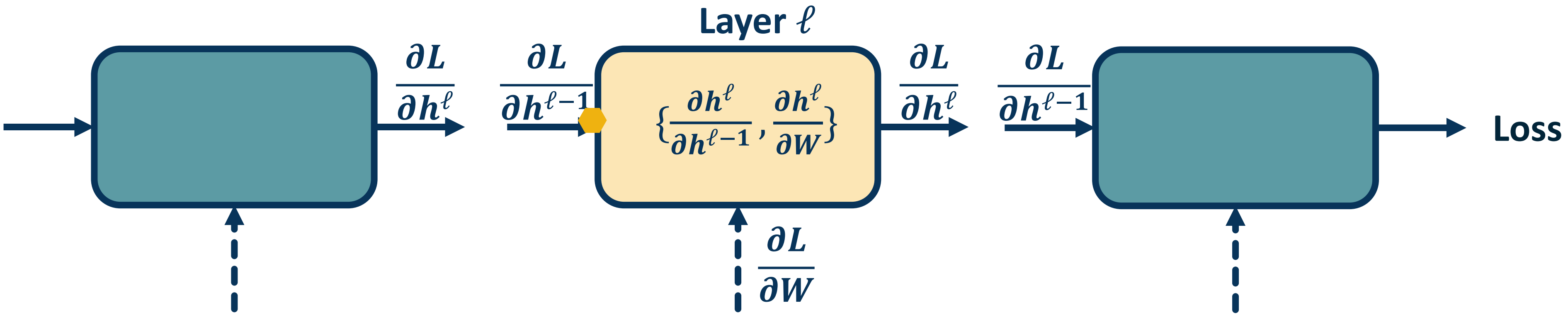
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

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



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

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

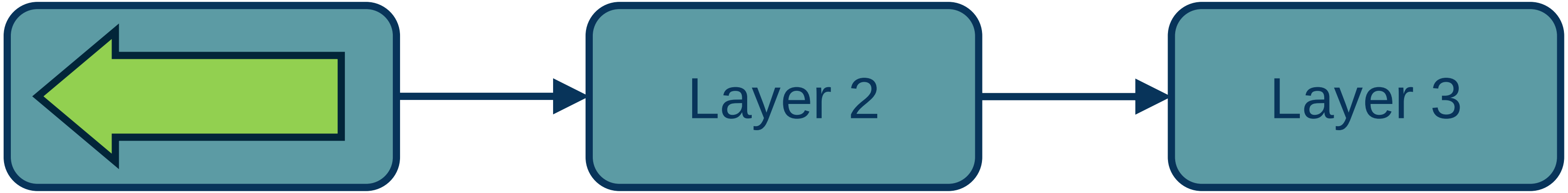
$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$$

Computing the Gradients of Loss

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

Backpropagation: a simple example

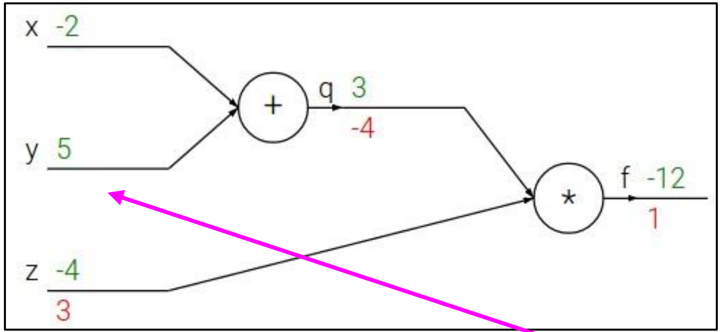
$$f(x, y, z) = (x + y)z$$

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

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

Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher

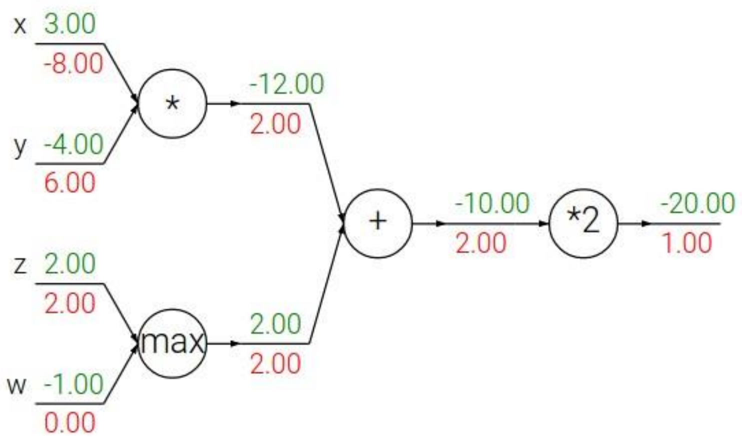


Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

- 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?
  - How can we implement this algorithmically to make these calculations automatic? **Automatic Differentiation**

Summary

Georgia Tech

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: $[c \times (m+1)]$ $[(m+1) \times 1]$

Where c is number of classes

m 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 $\boldsymbol{M} \in \mathbb{R}^{m_1 \times m_2}$

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

Conventions:

- Size of derivatives for scalars, vectors, and matrices:
  Assume we have scalar $\boldsymbol{s} \in \mathbb{R}^{1}$, vector $\boldsymbol{v} \in \mathbb{R}^{\boldsymbol{m}}$, i.e. $\boldsymbol{v} = [\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_m}]^T$
  and matrix $\boldsymbol{M} \in \mathbb{R}^{\boldsymbol{m_1} \times \boldsymbol{m_2}}$

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

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

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

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

Conventions:

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

Col j

Row i

$$\begin{bmatrix} \frac{\partial v_1^1}{\partial v_1^2} & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial v_i^1}{\partial v_1^2} & \cdots & \frac{\partial v_i^1}{\partial v_j^2} & \cdots & \frac{\partial v_i^1}{\partial v_{m_2}^2} & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix}$$

$m_1 \times m_2$

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

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

Conventions:

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

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

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

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:

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

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

$$= [w_1, \ldots, w_m]$$

because $\frac{\partial(\sum_k w_k x_k)}{\partial x_i} = w_i$

$$= w^T$$

Example 3:

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

Col j

Row i
$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \frac{\partial y_i}{\partial x_j} & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix} = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & w_{ij} & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix}$$

$$y_i = \sum_j w_{ij} x_j$$

Example 4:

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

- 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:

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

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

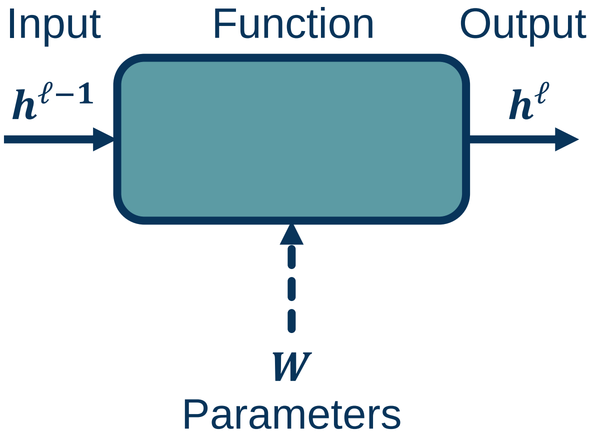
- Instead, flatten input to a vector and get a vector of derivatives!
- This can also be done for partial derivatives between two vectors, two matrices, or two tensors

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

Flatten

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

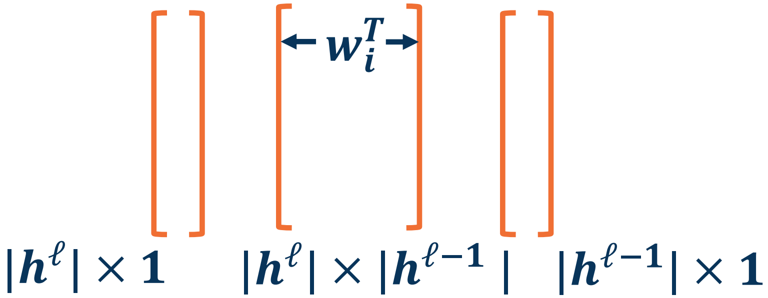
Jacobians of Batches



Define:

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

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



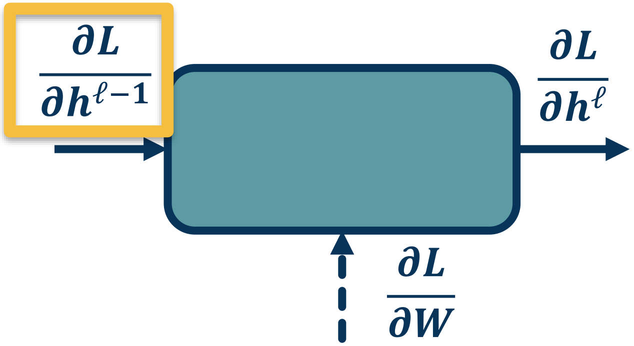
Fully Connected (FC) Layer: Forward Function

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

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

Define:

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



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

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

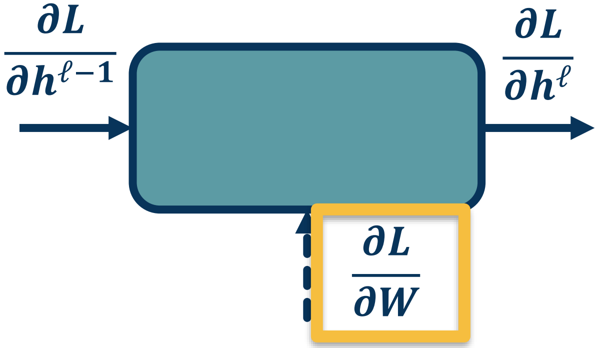
Fully Connected (FC) Layer

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

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

Define:

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



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

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

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

Fully Connected (FC) Layer

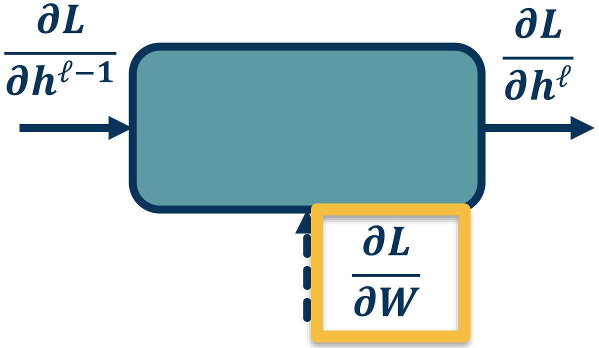
$$h^{\ell} = W h^{\ell-1}$$

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

Define:

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

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



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

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

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

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

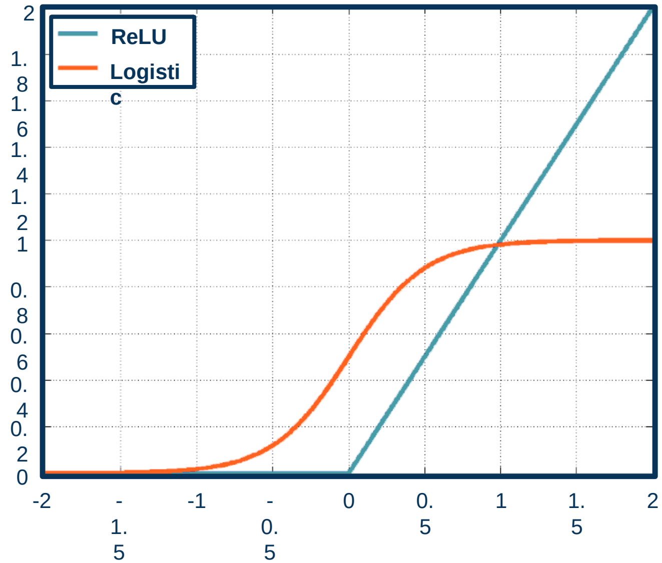
$1 \times |h^{\ell-1}| \quad 1 \times |h^{\ell}| \quad |h^{\ell}| \times |h^{\ell-1}|$

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

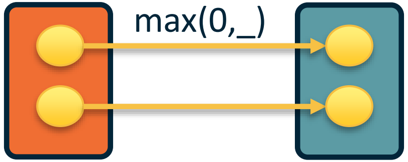
A common choice is the **Rectified Linear Unit**

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

How many parameters for this layer?



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



Rectified Linear Unit (ReLU)

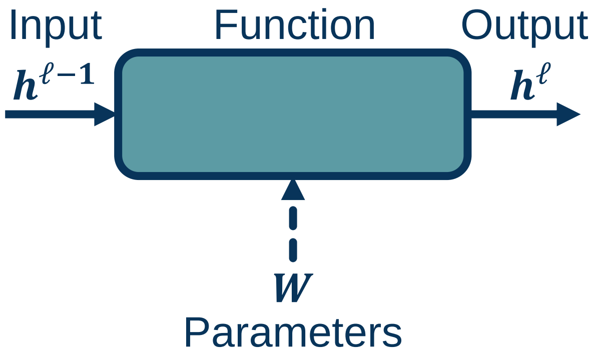
Jacobian of ReLU

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**

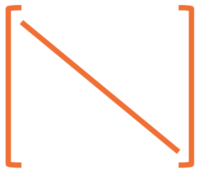
Max function **funnels gradients through selected max**

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



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

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

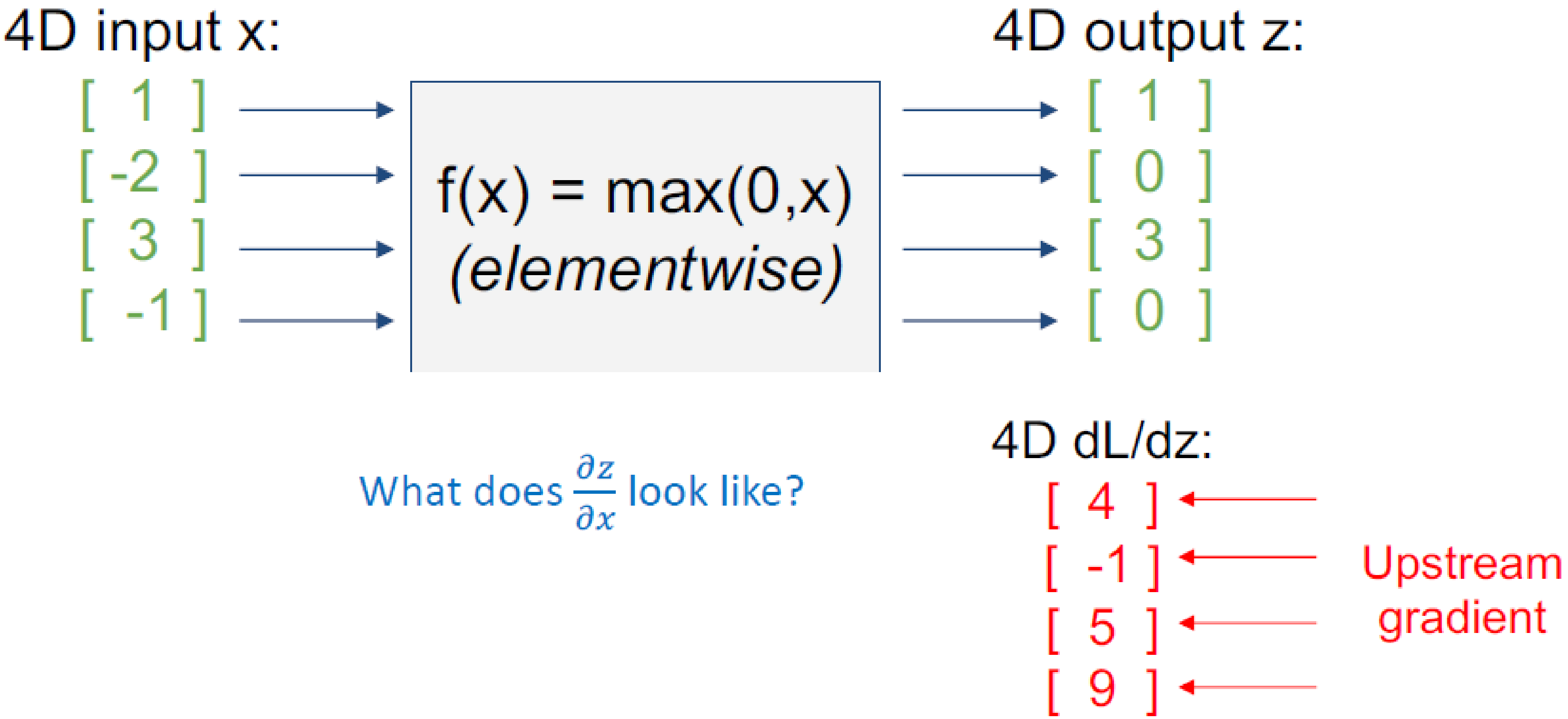


$|h^{\ell} \times h^{\ell-1}|$

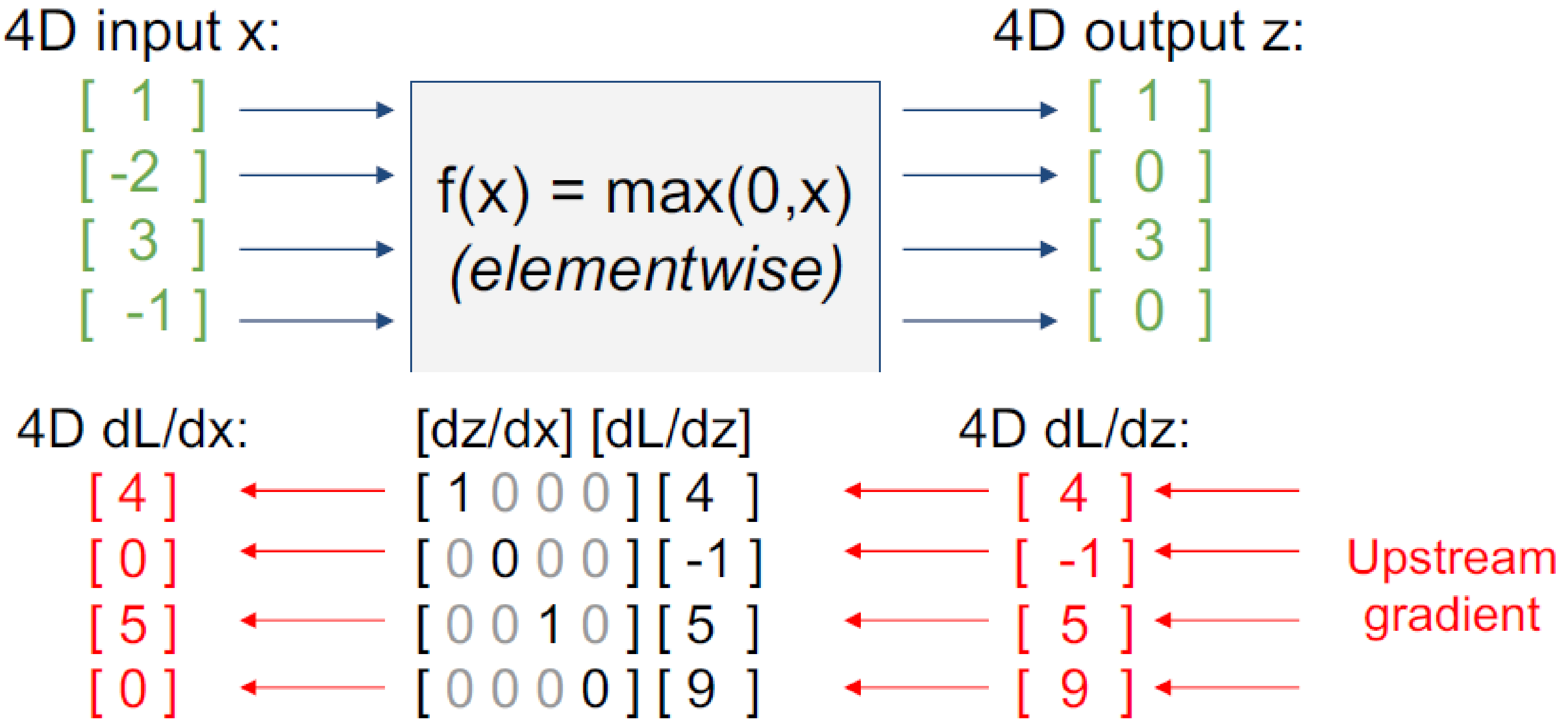
For diagonal

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

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



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



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? **Next!**
  - How can we implement this algorithmically to make these calculations automatic? **Automatic Differentiation**

Summary

Georgia Tech

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!)

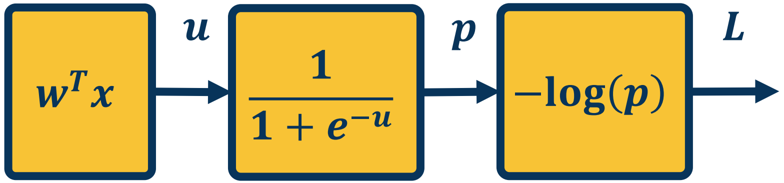
Scalar Case

Vector Case

Jacobian View of Chain Rule

Graphical View of Chain Rule

Chain Rule: Cascaded



$\bar{L} = 1$

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

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

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

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

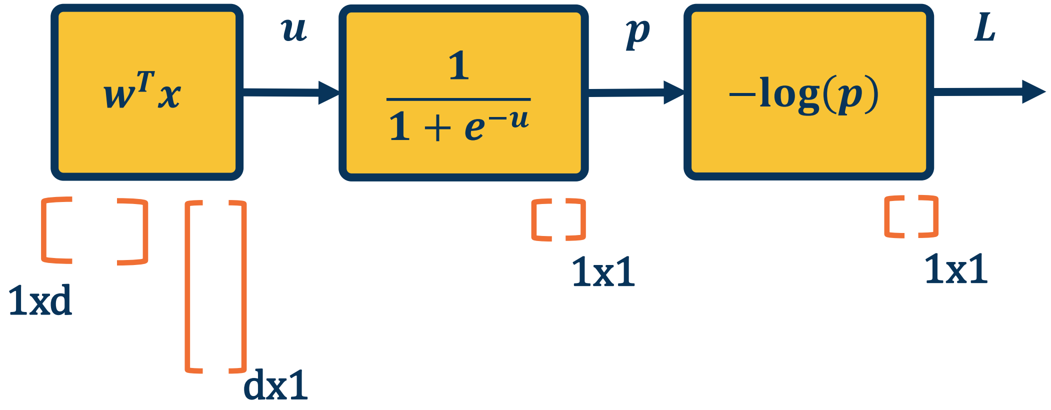
We can do this in a combined way to see all terms together:

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

This effectively shows gradient flow along path from L to w

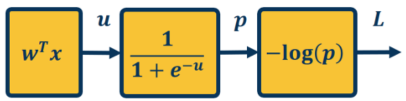
Example Gradient Computations

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



Extremely efficient in graphics processing units (GPUs)

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



$\bar{L} = 1$

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

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

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

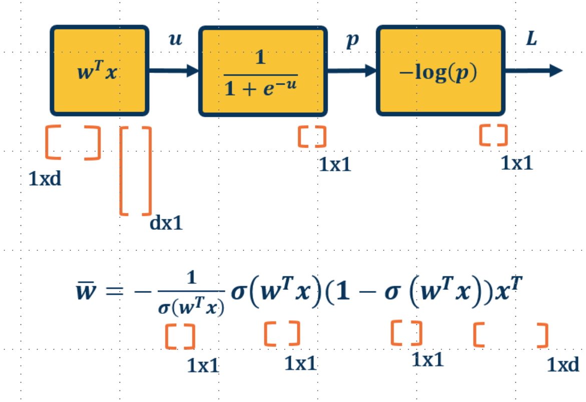
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

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

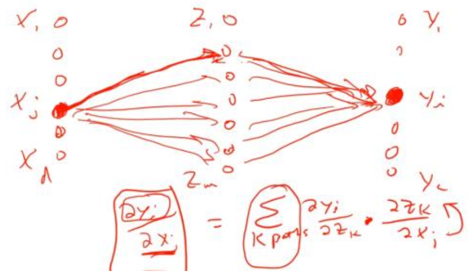
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

This effectively shows gradient flow along path from L to w

Computation Graph / Global View of Chain Rule

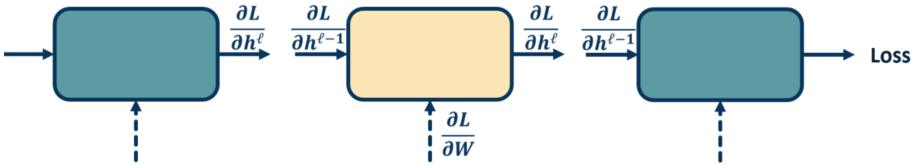


Computational / Tensor View



Graph View

- We want to 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

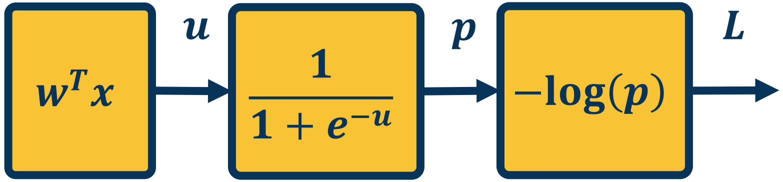
Georgia Tech

- **Backpropagation:** Recursive, modular algorithm for chain rule + gradient descent

- **When we move to vectors and matrices:**
  - Composition of functions (scalar)
  - Composition of functions (vectors/matrices)
  - Jacobian view of chain rule
  - Can view entire set of calculations as linear algebra operations (matrix-vector or matrix-matrix multiplication)

- **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

Summary



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!**

Automatic Differentiation