Topics:

- Backpropagation
- Matrix/Linear Algebra view

CS 4644-DL / 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!

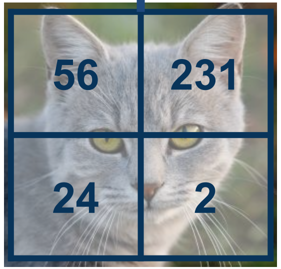
- Resources:
  - These lectures
  - Matrix calculus for deep learning
  - Gradients notes and MLP/ReLU Jacobian notes.
  - **Topic OH:** Assignment 1

- **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)

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

Stretch pixels into column



Input image

$$\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}}_{W} \begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix} + \underbrace{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}}_{b} = \begin{bmatrix} -96.8 \\ 437.9 \\ 61.95 \end{bmatrix} \begin{matrix} \text{Cat score} \\ \text{Dog score} \\ \text{Ship score} \end{matrix}$$

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

- If we use the softmax function to convert scores to probabilities, the right loss function to use is **cross-entropy**
- Can be derived by looking at the distance between two probability distributions (output of model and ground truth)
- Can also be derived from a maximum likelihood estimation perspective

$$s = f(x, W)$$ **Scores**

$$P(Y = k | X = x) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ **Softmax Function**

$$L_i = -\log P(Y = y_i | X = x_i)$$

Maximize log-prob of correct class = Maximize the log likelihood = Minimize the negative log likelihood

Performance Measure for Probabilities

Georgia Tech

- 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

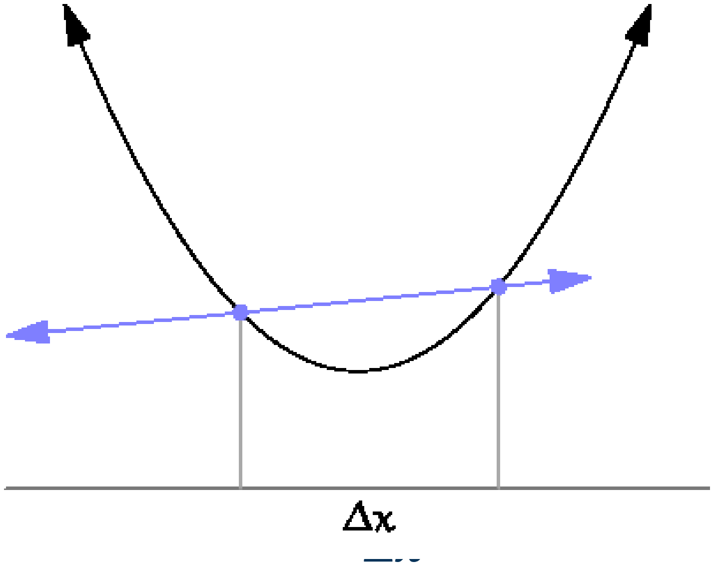


Image and equation from: https://en.wikipedia.org/wiki/Derivative#/media/File:Tangent_animation.gif

Derivatives

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

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

- In Deep Learning, gradient descent on **Loss** with respect to **parameters/weights**,

$$\mathbf{L} \in \mathbb{R} \quad , \mathbf{w} \in \mathbb{R}^m$$

$$\frac{\partial L}{\partial w} = \left[\frac{\partial L}{\partial w_1}, \dots, \frac{\partial L}{\partial w_m}\right]$$

- Update rule is for each weight $w_i = w_i - \frac{\partial L}{\partial w_i}$

- (but of course we can vectorize operations)

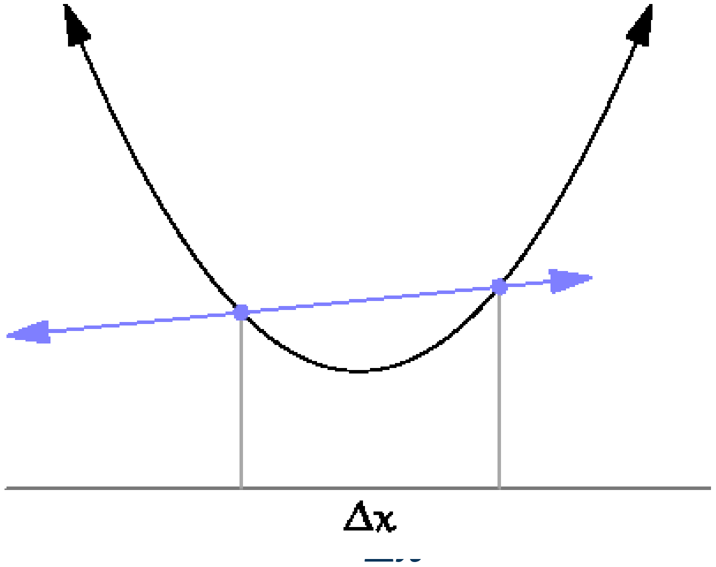
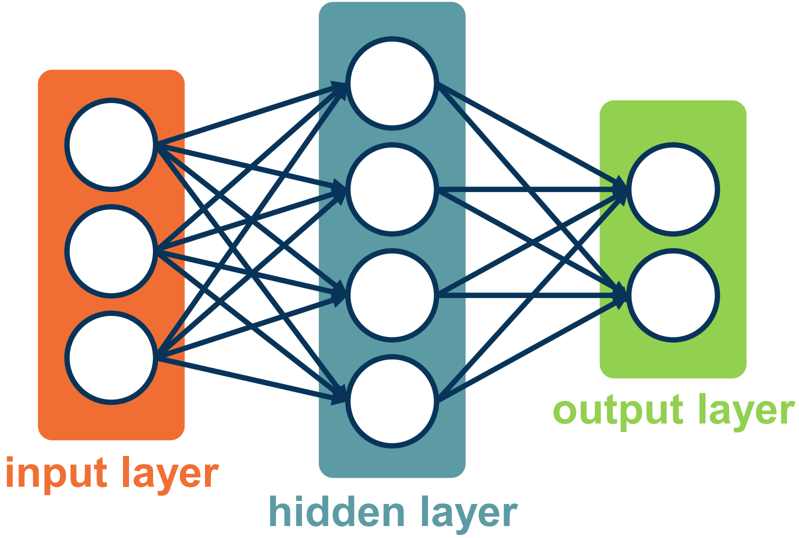


Image and equation from: https://en.wikipedia.org/wiki/Derivative#/media/File:Tangent_animation.gif

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)



$$x \qquad W_1 \qquad W_2$$

$$=$$

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

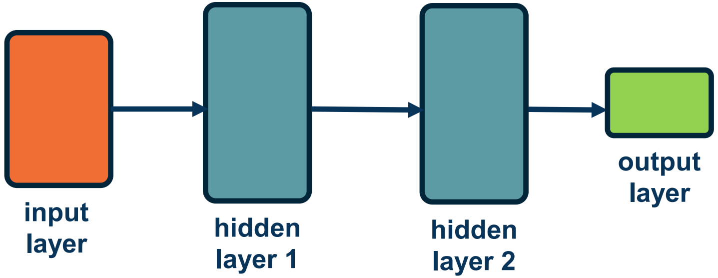
Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

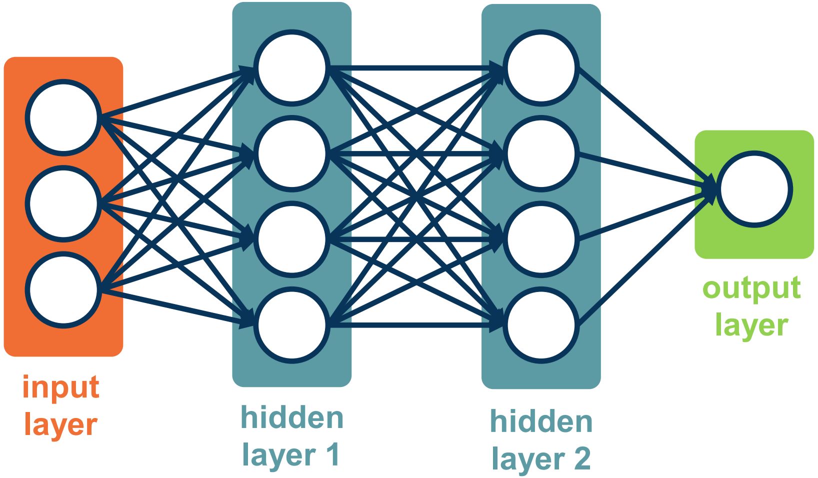
Large (deep) networks can be built by adding more and more layers

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**:



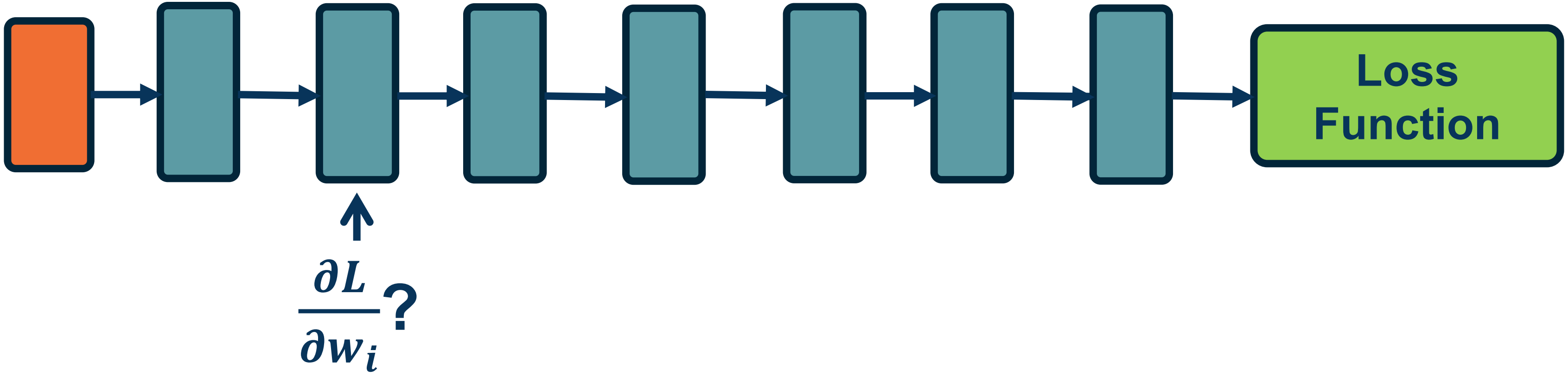


$$f(x, W_1, W_2, W_3) = \sigma(W_2\sigma(W_1x))$$

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

Adding More Layers!

- 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



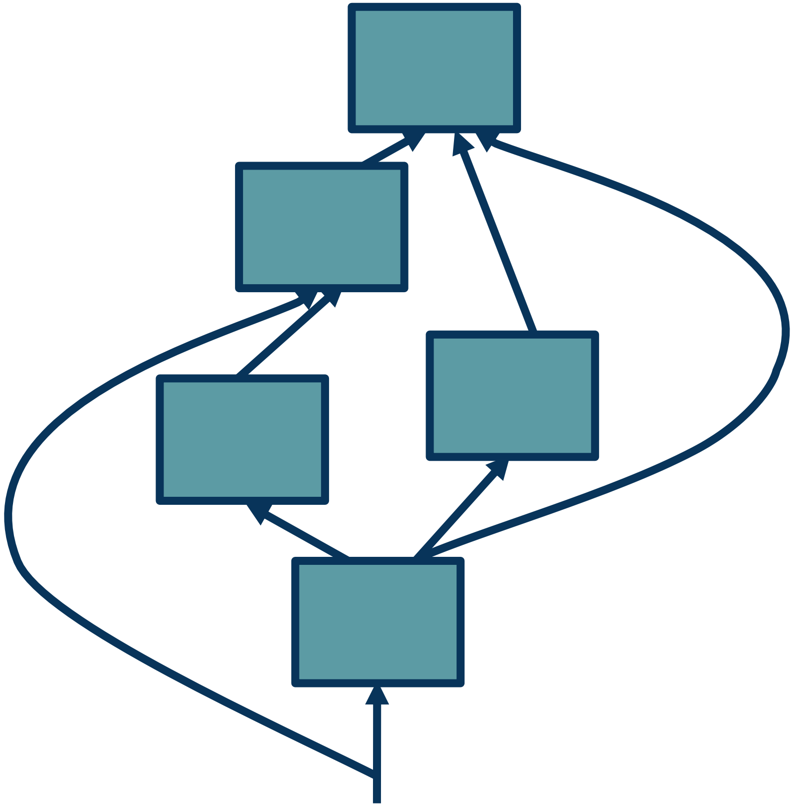
Computing Gradients in Complex Function

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

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

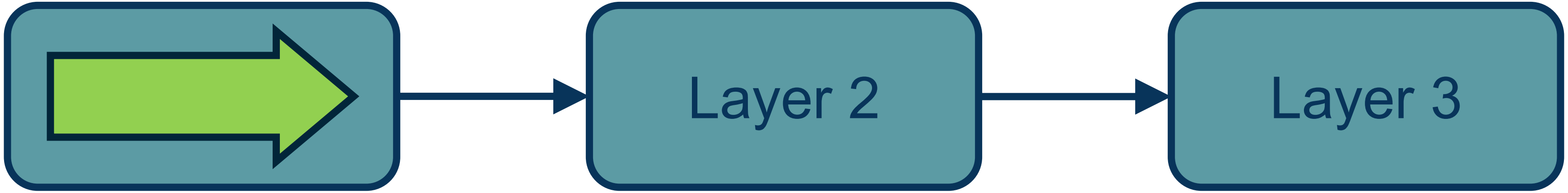
- Modules must be differentiable to support gradient computations for gradient descent

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**

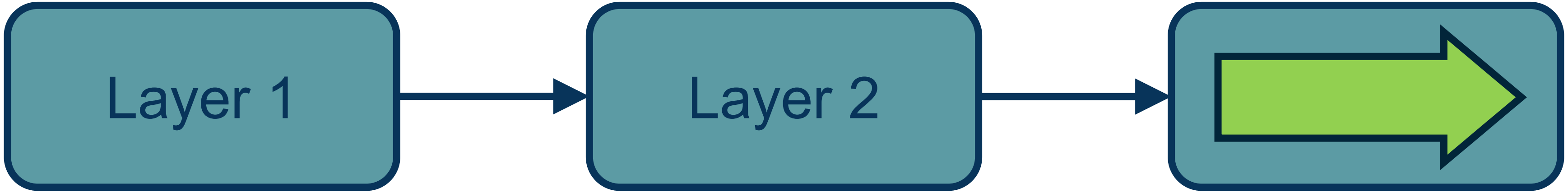


Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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**



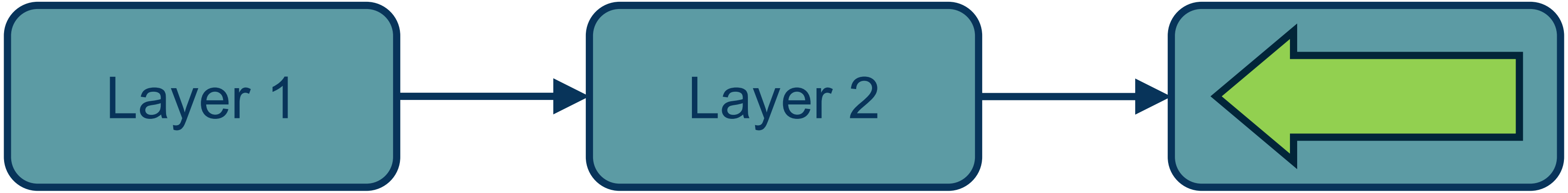
Note that we must store the **intermediate outputs of all layers**!

- 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**

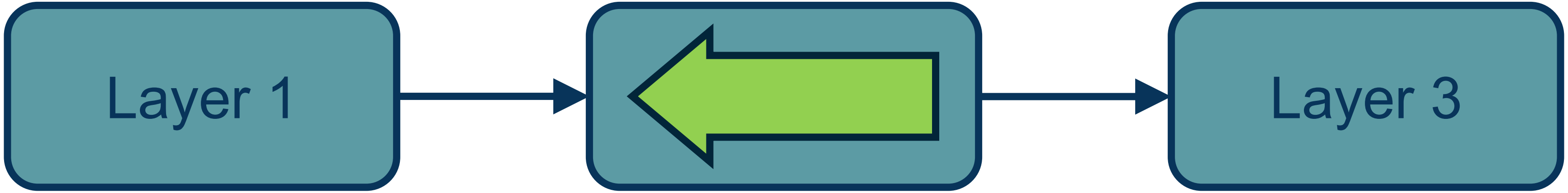
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

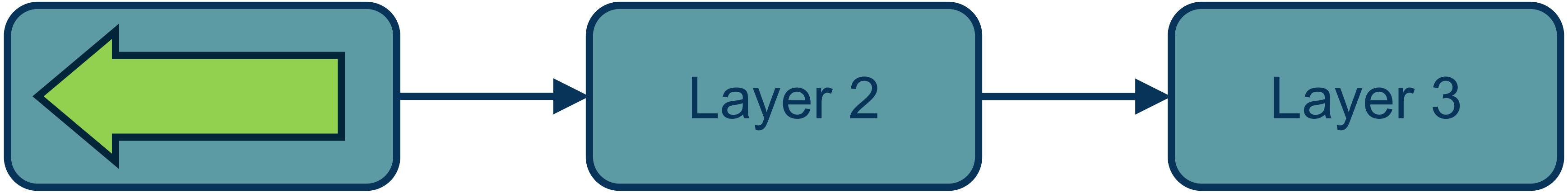
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

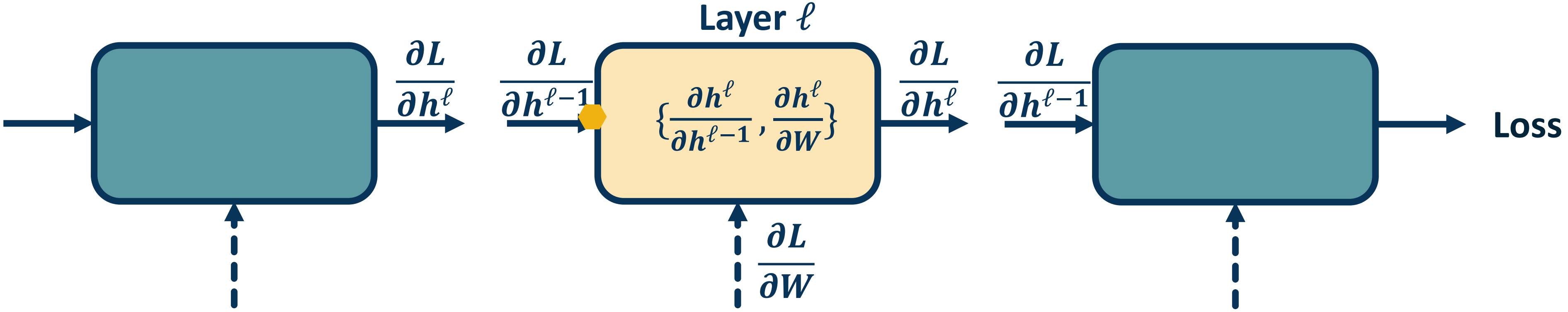
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

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:

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

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

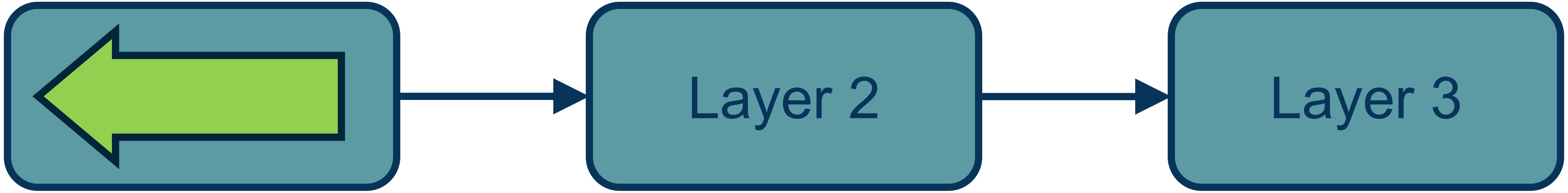
$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$$

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

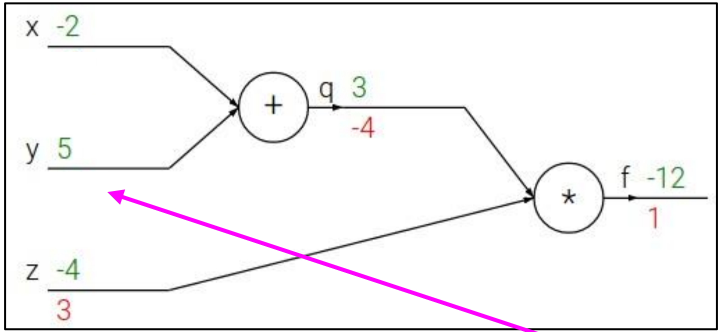
$$f(x, y, z) = (x + y)z$$

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

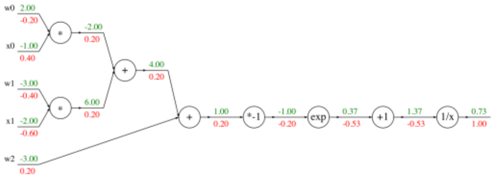
Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

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)
  - Backpropagation implementation on the graph

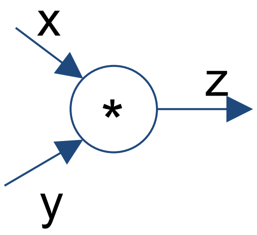
Modularized implementation: forward / backward API



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



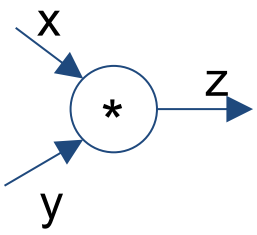
(x,y,z are scalars)

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

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

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

Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Example: Caffe layers

Branch: master ▾ caffe / src / caffe / layers /

Create new file | Upload files | Find file | History

shelhamer committed on GitHub Merge pull request #4630 from BlGene/load_hdf5_fix ... Latest commit e687a71 21 days ago

..

absval_layer.cpp	dismantle layer headers	a year ago
absval_layer.cu	dismantle layer headers	a year ago
accuracy_layer.cpp	dismantle layer headers	a year ago
argmax_layer.cpp	dismantle layer headers	a year ago
base_conv_layer.cpp	enable dilated deconvolution	a year ago
base_data_layer.cpp	Using default from proto for prefetch	3 months ago
base_data_layer.cu	Switched multi-GPU to NCCL	3 months ago
batch_norm_layer.cpp	Add missing spaces besides equal signs in batch_norm_layer.cpp	4 months ago
batch_norm_layer.cu	dismantle layer headers	a year ago
batch_reindex_layer.cpp	dismantle layer headers	a year ago
batch_reindex_layer.cu	dismantle layer headers	a year ago
bias_layer.cpp	Remove incorrect cast of gemm int arg to Dtype in BiasLayer	a year ago
bias_layer.cu	Separation and generalization of ChannelwiseAffineLayer into BiasLayer	a year ago
bnll_layer.cpp	dismantle layer headers	a year ago
bnll_layer.cu	dismantle layer headers	a year ago
concat_layer.cpp	dismantle layer headers	a year ago
concat_layer.cu	dismantle layer headers	a year ago
contrastive_loss_layer.cpp	dismantle layer headers	a year ago
contrastive_loss_layer.cu	dismantle layer headers	a year ago
conv_layer.cpp	add support for 2D dilated convolution	a year ago
conv_layer.cu	dismantle layer headers	a year ago
crop_layer.cpp	remove redundant operations in Crop layer (#5138)	2 months ago
crop_layer.cu	remove redundant operations in Crop layer (#5138)	2 months ago
cudnn_conv_layer.cpp	dismantle layer headers	a year ago
cudnn_conv_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_lcn_layer.cpp	dismantle layer headers	a year ago
cudnn_lcn_layer.cu	dismantle layer headers	a year ago
cudnn_lrn_layer.cpp	dismantle layer headers	a year ago
cudnn_lrn_layer.cu	dismantle layer headers	a year ago
cudnn_pooling_layer.cpp	dismantle layer headers	a year ago
cudnn_pooling_layer.cu	dismantle layer headers	a year ago
cudnn_relu_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_relu_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_sigmoid_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_sigmoid_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_softmax_layer.cpp	dismantle layer headers	a year ago
cudnn_softmax_layer.cu	dismantle layer headers	a year ago
cudnn_tanh_layer.cpp	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
cudnn_tanh_layer.cu	Add cuDNN v5 support, drop cuDNN v3 support	11 months ago
data_layer.cpp	Switched multi-GPU to NCCL	3 months ago
deconv_layer.cpp	enable dilated deconvolution	a year ago
deconv_layer.cu	dismantle layer headers	a year ago
dropout_layer.cpp	supporting N-D Blobs in Dropout layer Reshape	a year ago
dropout_layer.cu	dismantle layer headers	a year ago
dummy_data_layer.cpp	dismantle layer headers	a year ago
eltwise_layer.cpp	dismantle layer headers	a year ago
eltwise_layer.cu	dismantle layer headers	a year ago
elu_layer.cpp	ELU layer with basic tests	a year ago
elu_layer.cu	ELU layer with basic tests	a year ago
embed_layer.cpp	dismantle layer headers	a year ago
embed_layer.cu	dismantle layer headers	a year ago
euclidean_loss_layer.cpp	dismantle layer headers	a year ago
euclidean_loss_layer.cu	dismantle layer headers	a year ago
exp_layer.cpp	Solving issue with exp layer with base e	a year ago
exp_layer.cu	dismantle layer headers	a year ago

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

Caffe Sigmoid Layer

```
#include <cmath>
#include <vector>

#include "caffe/layers/sigmoid_layer.hpp"

namespace caffe {

template <typename Dtype>
inline Dtype sigmoid(Dtype x) {
  return 1. / (1. + exp(-x));
}

template <typename Dtype>
void SigmoidLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  const int count = bottom[0]->count();
  for (int i = 0; i < count; ++i) {
    top_data[i] = sigmoid(bottom_data[i]);
  }
}

template <typename Dtype>
void SigmoidLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
  if (propagate_down[0]) {
    const Dtype* top_data = top[0]->cpu_data();
    const Dtype* top_diff = top[0]->cpu_diff();
    Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
    const int count = bottom[0]->count();
    for (int i = 0; i < count; ++i) {
      const Dtype sigmoid_x = top_data[i];
      bottom_diff[i] = top_diff[i] * sigmoid_x * (1. - sigmoid_x);
    }
  }
}

#ifdef CPU_ONLY
STUB_GPU(SigmoidLayer);
#endif

INSTANTIATE_CLASS(SigmoidLayer);


}  // namespace caffe
```


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

$(1-\sigma(x))\sigma(x)$ * top_diff (chain rule)

- 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

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

W $\quad$ x

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

Where c is number of classes

m is dimensionality of input

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 $\boldsymbol{v} \in \mathbb{R}^{m}$, i.e. $\boldsymbol{v} = [v_1, v_2, \dots, v_m]^T$
  and matrix $\boldsymbol{M} \in \mathbb{R}^{m_1 \times m_2}$

- 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:

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

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

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

$$= w^T$$

Example 3:

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

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:

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

Jacobians of Batches

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

Examples:

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

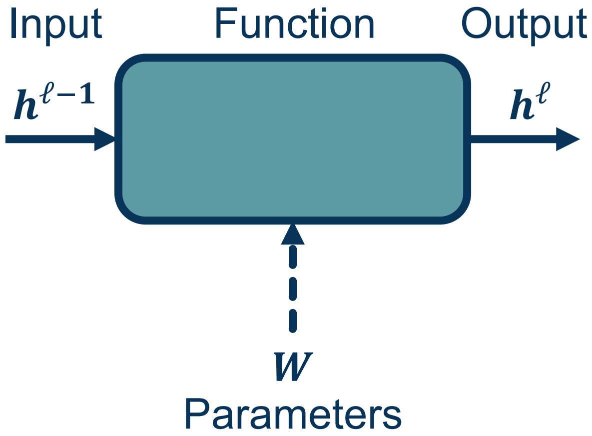
Jacobians become tensors which is complicated

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

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

Flatten

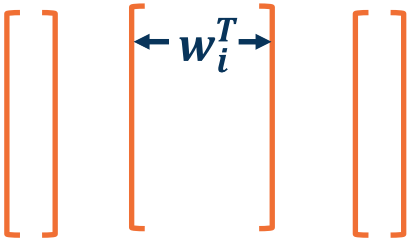
$$\begin{bmatrix} x_{11} \\ x_{12} \\ \vdots \\ x_{21} \\ x_{22} \\ \vdots \\ x_{n1} \\ \vdots \\ x_{nn} \end{bmatrix}$$



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

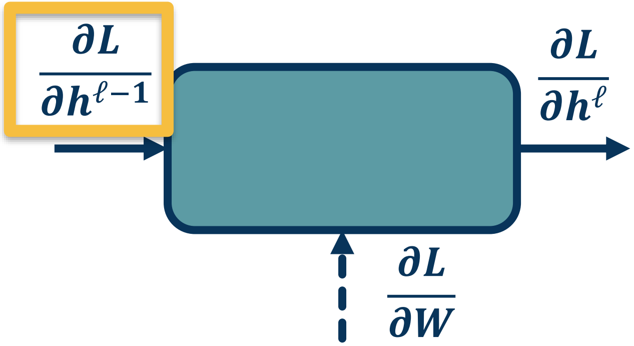
Georgia Tech

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

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

Define:

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



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

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

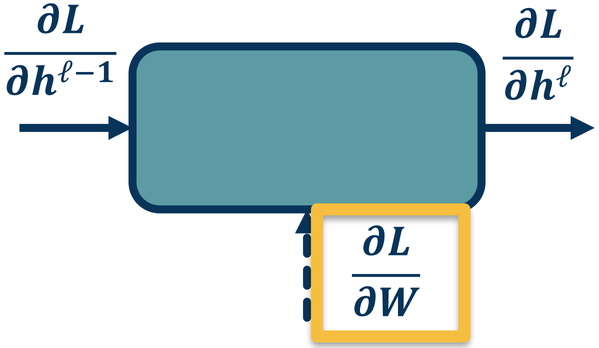
Fully Connected (FC) Layer

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

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

Define:

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

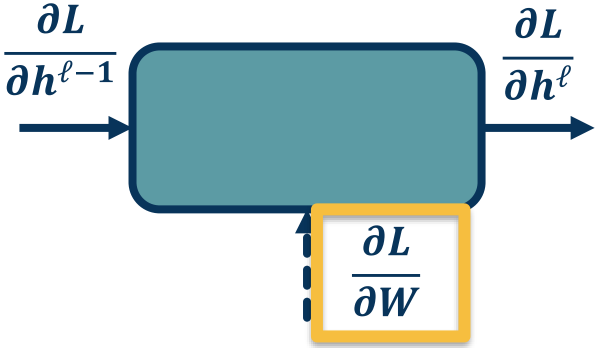


$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^\ell}\frac{\partial h^\ell}{\partial W}$$

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



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

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

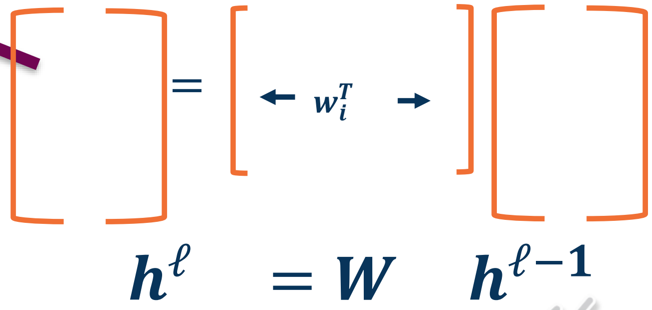
$$\frac{\partial h^{\ell}}{\partial h^{\ell-1}} = W$$

Define:

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

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

Find $\frac{\partial h^{\ell}}{\partial w_i^T}$



Fully Connected (FC) Layer

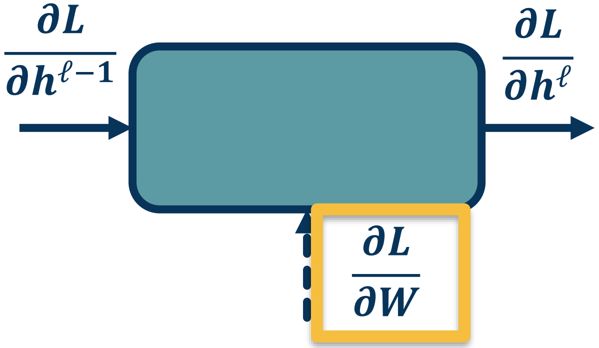
$$h^{\ell} = W h^{\ell-1}$$

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

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

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

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

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

Iterate and populate

Note can simplify/vectorize!

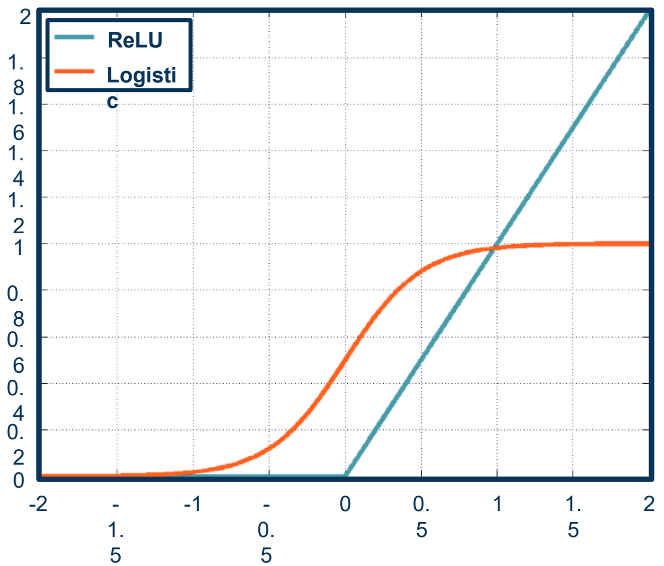
Fully Connected (FC) Layer

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

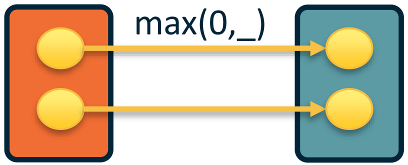
A common choice is the **Rectified Linear Unit**

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

How many parameters for this layer?



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



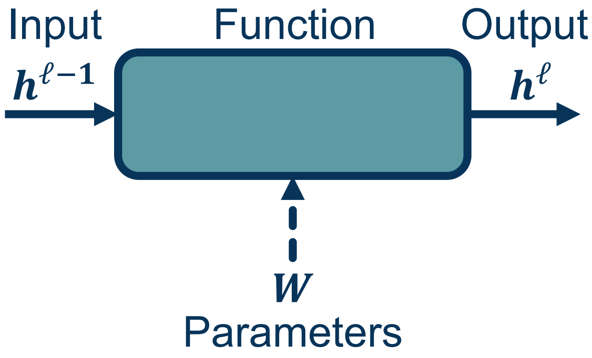
Jacobian of ReLU

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**

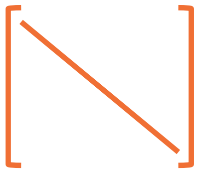
Max function **funnels gradients through selected max**

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



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

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

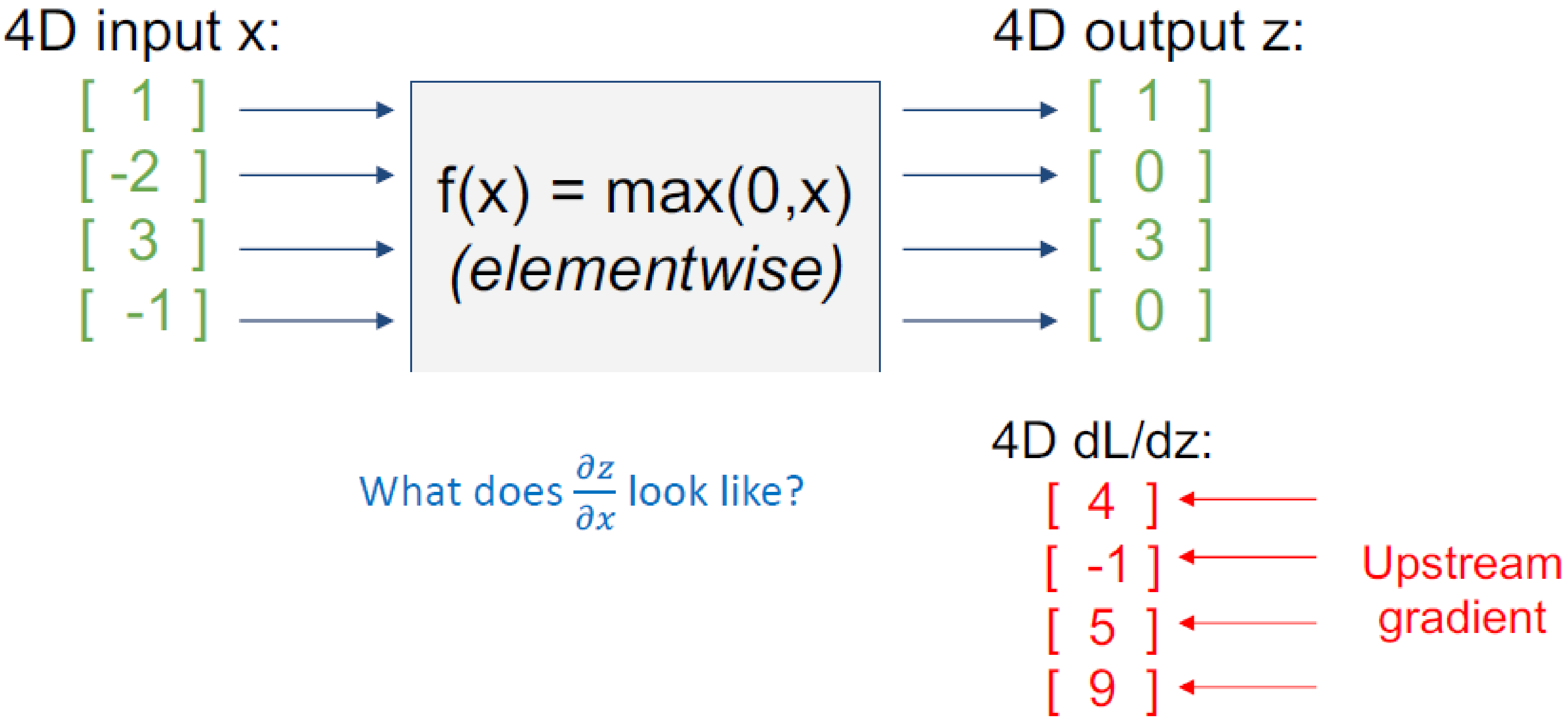


$|h^{\ell} \times h^{\ell-1}|$

For diagonal

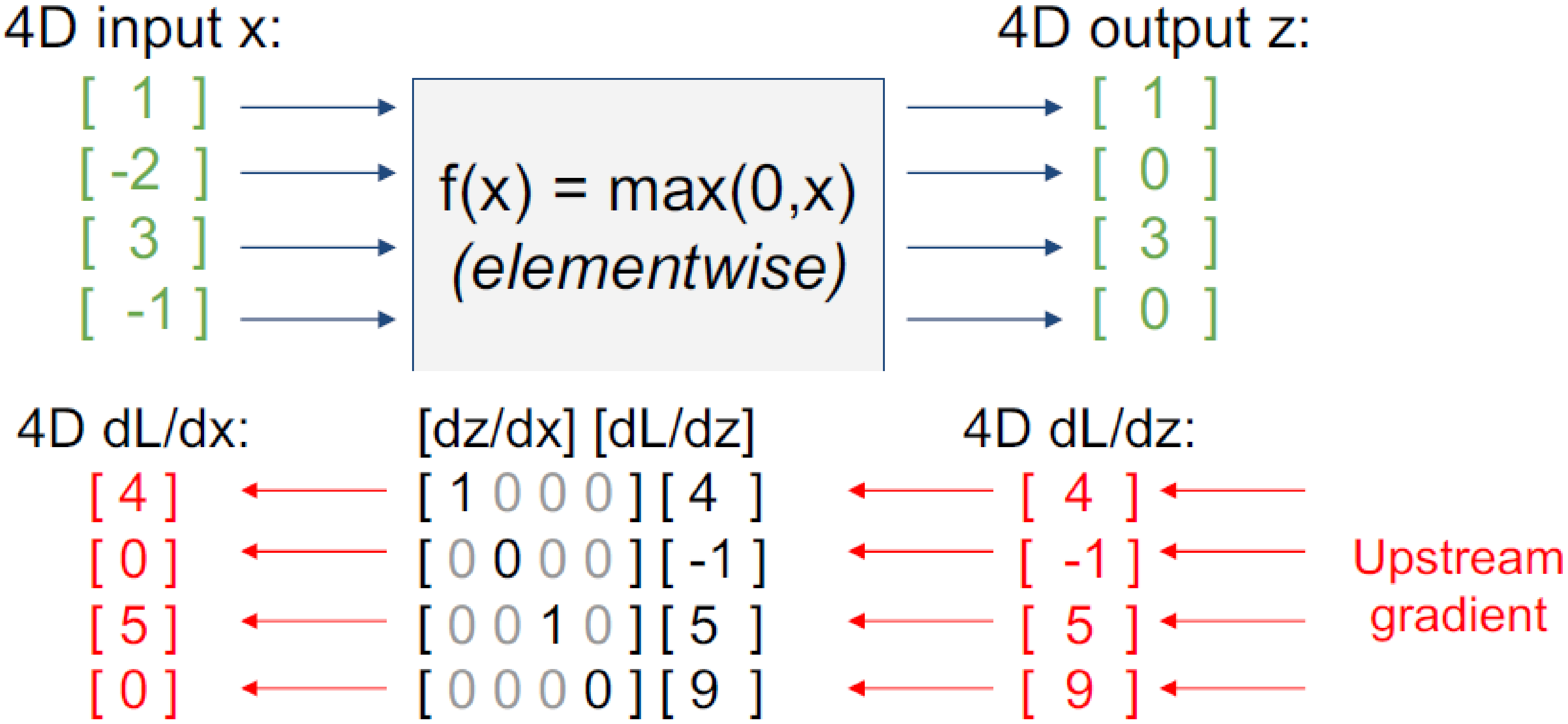
$$\frac{\partial h^{\ell}}{\partial h^{\ell-1}} = \begin{cases} 1 & if\ h^{\ell-1} > 0 \\ 0 & otherwise \end{cases}$$

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



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

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

Summary

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

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

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

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

(Many of these will be parameterized)

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

$$\mathbf{x} \in \mathbb{R}^1 \xrightarrow{\boldsymbol{g}_1} \mathrm{z} \in \mathbb{R}^1 \xrightarrow{\boldsymbol{g}_2} \mathrm{y} \in \mathbb{R}^1$$

$$\boldsymbol{y} = \boldsymbol{g}_2\left(\boldsymbol{g}_1(\boldsymbol{x})\right)$$

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

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

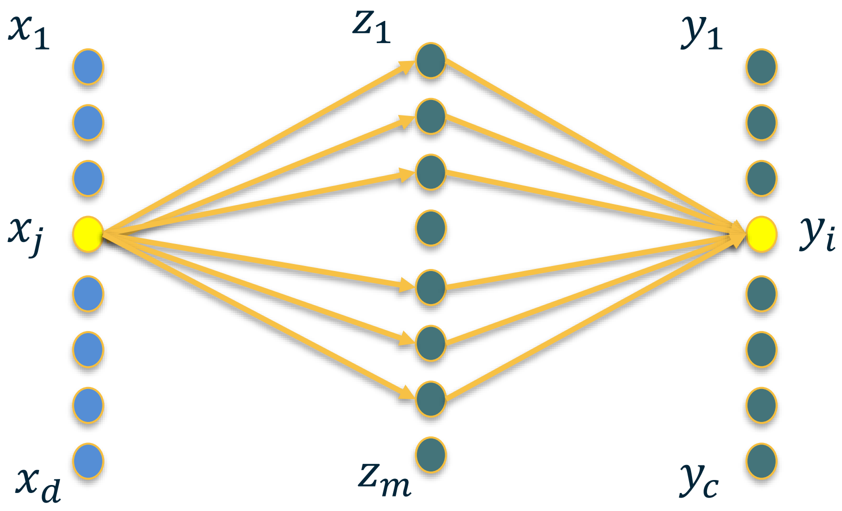
Vector Case

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

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

Jacobian View of Chain Rule

Georgia Tech



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

k paths

Graphical View of Chain Rule

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

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

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

Chain Rule: Cascaded

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

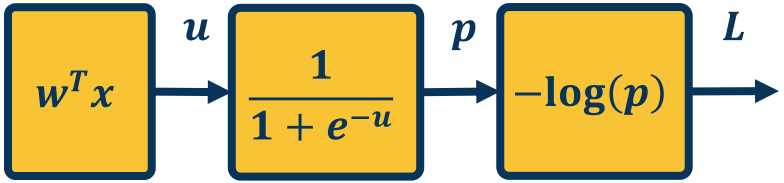
$$\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{h}^{1}} = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{h}^{l}} \frac{\partial h^l}{\partial h^{l-1}} \frac{\partial h^{l-1}}{\partial h^{l-2}} \cdots \frac{\partial h^2}{\partial h^1}$$

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

Which directions is more efficient to multiply?

Chain Rule: Cascaded

Georgia Tech



$\bar{L} = 1$

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

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

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

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

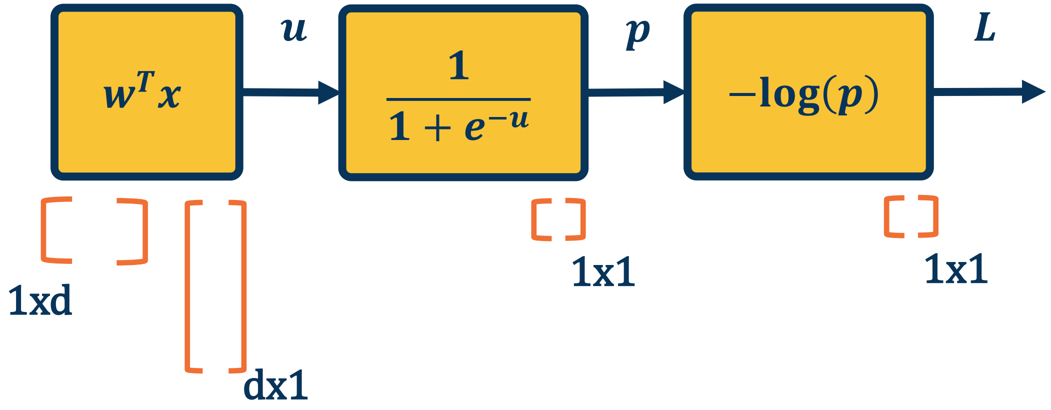
We can do this in a combined way to see all terms together:

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

This effectively shows gradient flow along path from L to w

Example Gradient Computations

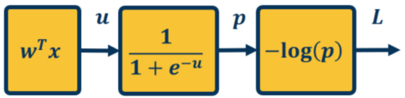
The chain rule can be computed as a **series of scalar, vector, and matrix linear algebra operations**



Extremely efficient in graphics processing units (GPUs)

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

Vectorized Computations



$L = 1$

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

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

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

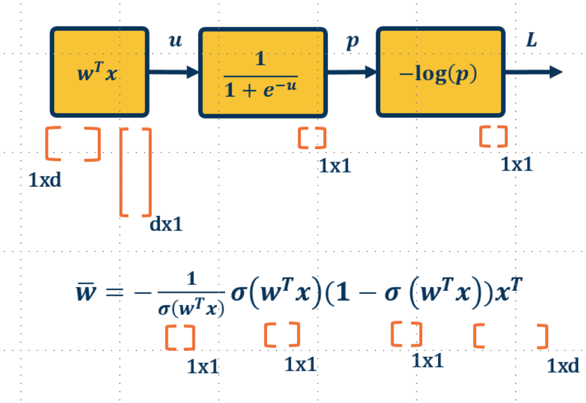
$\bar{w} = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w} = \bar{u}x^T$

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

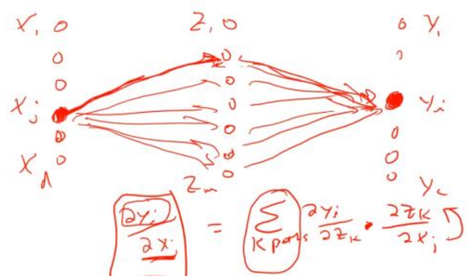
$$\bar{w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w} = -\frac{1}{\sigma(w^T x)}\sigma(w^T x)(1-\sigma(w^T x))x^T$$
$$= -\left(1-\sigma(w^T x)\right)x^T$$

This effectively shows gradient flow along path from L to w

Computation Graph / Global View of Chain Rule

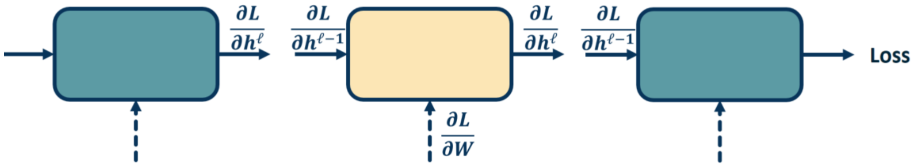


Computational / Tensor View



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

- **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