Topics:

- Gradient Descent
- Neural Networks

# CS 4644-DL / 7643-A
# ZSOLT KIRA

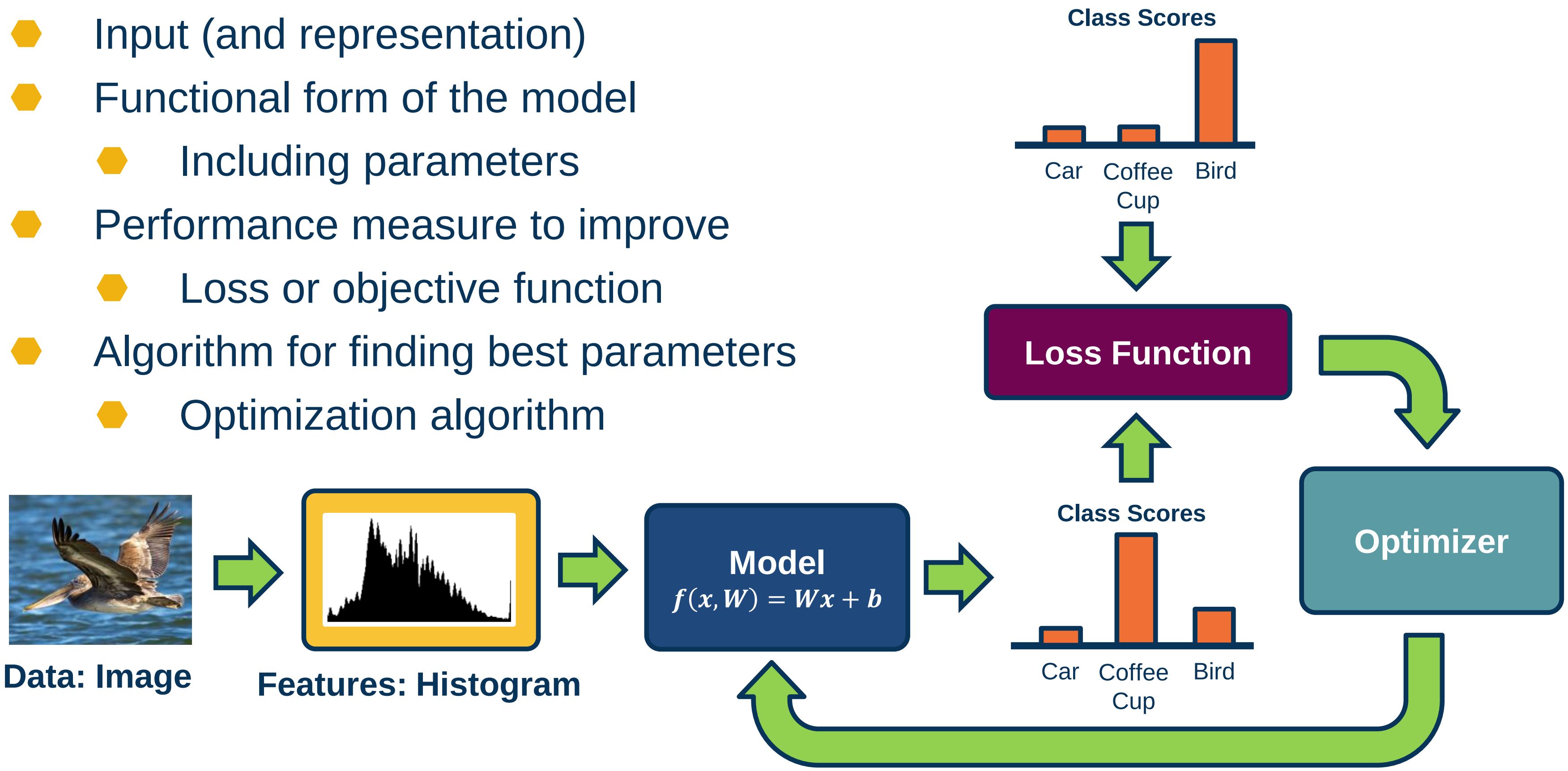
- **Assignment 1 out!**
  - **Due Feb 2nd (with grace period Feb 4th)**
  - Start now, start now, start now!
  - Start now, start now, start now!
  - Start now, start now, start now!

- **Piazza**
  - Be active!!!

- **Office hours**
  - Lots of special topics (e.g. Assignment 1, Matrix Calculus, etc. )

- Note: Course will start to get math heavy!

- Input (and representation)
- Functional form of the model
  - Including parameters
- Performance measure to improve
  - Loss or objective function
- Algorithm for finding best parameters
  - Optimization algorithm




# Components of a Parametric Model

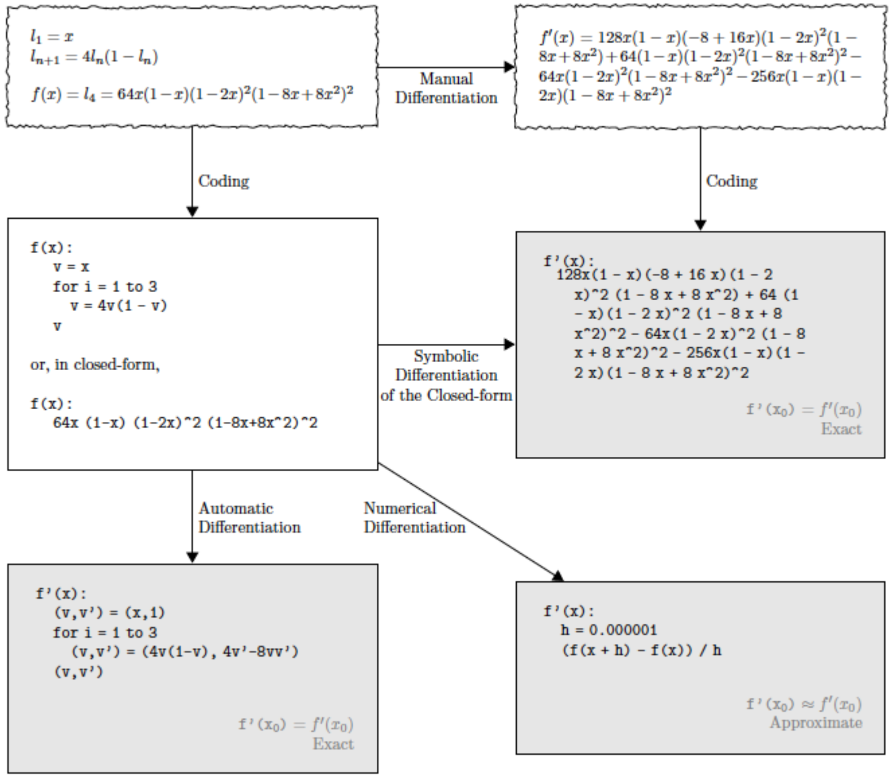
- Input**: Vector**
- Functional form of the model: **Softmax(Wx)**
- Performance measure to improve: **Cross-Entropy**
- Algorithm for finding best parameters: **Gradient Descent**
  - Compute $\frac{\partial L}{\partial w_i}$
  - Update Weights $w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$

# So far

We know how to compute the **model output and loss function**

**Several ways to compute $\frac{\partial L}{\partial w_i}$**

- Manual differentiation
- Symbolic differentiation
- Numerical differentiation
- Automatic differentiation




**Computing Gradients**

# For some functions, we can analytically derive the partial derivative

## Example:

**Function**

$$f(w, x_i) = w^T x_i$$

**Loss**

$$\sum_{i=1}^{N} (y_i - w^T x_i)^2$$

(Assume $w$ and $\mathbf{x_i}$ are column vectors, so same as $w \cdot x_i$)

**Dataset:** N examples (indexed by $i$)

**Update Rule**

$$w_j \leftarrow w_j + 2\alpha \sum_{i=1}^{N} \delta_i x_{ij}$$

## Derivation of Update Rule

L= $\sum_{i=1}^{N} (y_i - w^T x_i)^2$

Gradient descent tells us we should update $w$ as follows to minimize $L$:

$$w_j \leftarrow w_j - \alpha \frac{\partial L}{\partial w_j}$$

**So what's $\frac{\partial L}{\partial w_j}$?**

$$\frac{\partial L}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial}{\partial w_j} (y_i - w^T x_i)^2$$

$$= \sum_{i=1}^{N} 2(y_i - w^T x_i) \frac{\partial}{\partial w_j} (y_i - w^T x_i)$$

$$= -2 \sum_{i=1}^{N} \delta_i \frac{\partial}{\partial w_j} w^T x_i$$

…where… $\delta_i = y_i - w^T x_i$

$$= -2 \sum_{i=1}^{N} \delta_i \frac{\partial}{\partial w_j} \sum_{k=1} w_k x_{ik}$$

$$= -2 \sum_{i=1}^{N} \delta_i x_{ij}$$

# If we add a **non-linearity (sigmoid),** derivation is more complex

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

First, one can derive that: $\sigma'(x) = \sigma(x)(1-\sigma(x))$

$$\mathrm{f}(\mathrm{x}) = \sigma\left(\sum_k w_k x_k\right)$$

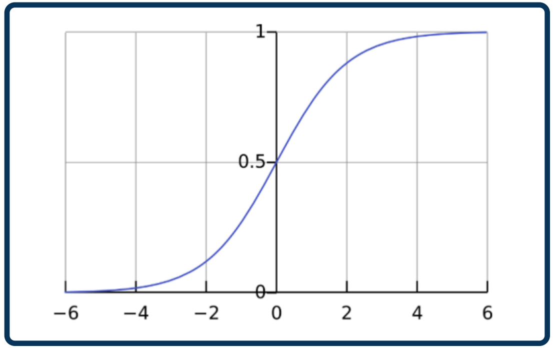
$$\mathrm{L} = \sum_i \left(y_i - \sigma\left(\sum_k w_k x_{ik}\right)\right)^2$$

$$\frac{\partial L}{\partial w_j} = \sum_i 2\left(y_i - \sigma\left(\sum_k w_k x_{ik}\right)\right)\left(-\frac{\partial}{\partial w_j}\sigma\left(\sum_k w_k x_{ik}\right)\right)$$

$$= \sum_i -2\left(y_i - \sigma\left(\sum_k w_k x_{ik}\right)\right)\sigma'\left(\sum_k w_k x_{ik}\right)\frac{\partial}{\partial w_j}\sum_k w_k x_{ik}$$

$$= \sum_i -2\delta_i \sigma(\mathrm{d}_i)(1-\sigma(\mathrm{d}_i))x_{ij}$$

where $\delta_i = y_i - \mathrm{f}(x_i)$ $\quad d_i = \sum w_k x_{ik}$




## The sigmoid perception update rule:

$$w_j \leftarrow w_j + 2\alpha \sum_{k=1}^{N} \delta_i \sigma_i (1-\sigma_i) x_{ij}$$

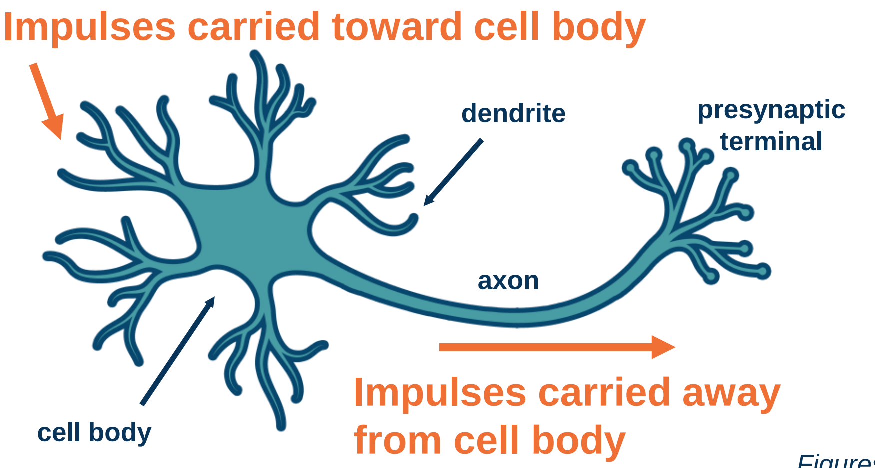
where $\sigma_i = \sigma\left(\sum_{j=1}^{d} w_j x_{ij}\right)$

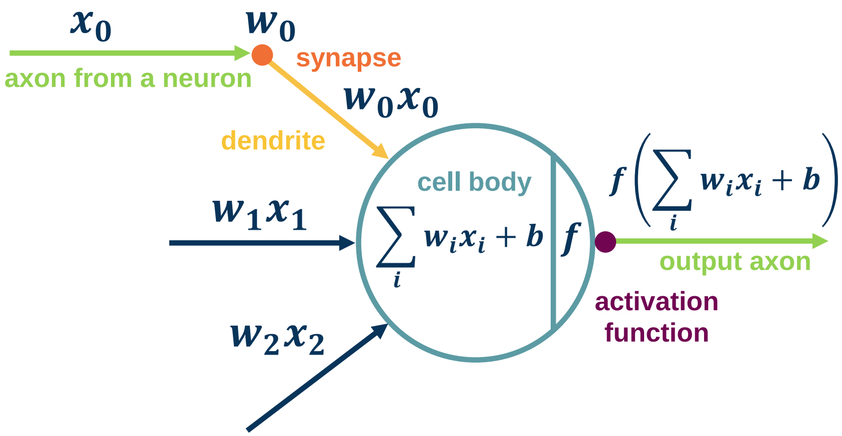
$$\delta_i = y_i - \sigma_i$$

# Neural Network View of a Linear Classifier

# A simple **neural network** has similar structure as our linear classifier:

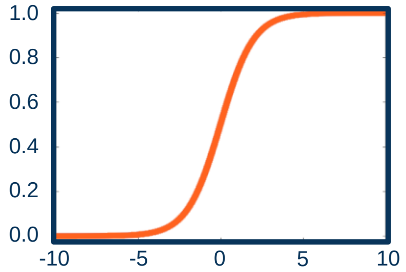
- A neuron takes input (firings) from other neurons **(-> input to linear classifier)**
- The inputs are summed in a weighted manner **(-> weighted sum)**
  - Learning is through a modification of the weights
- If it receives enough input, it "fires" (threshold or if weighted sum plus bias is high enough)



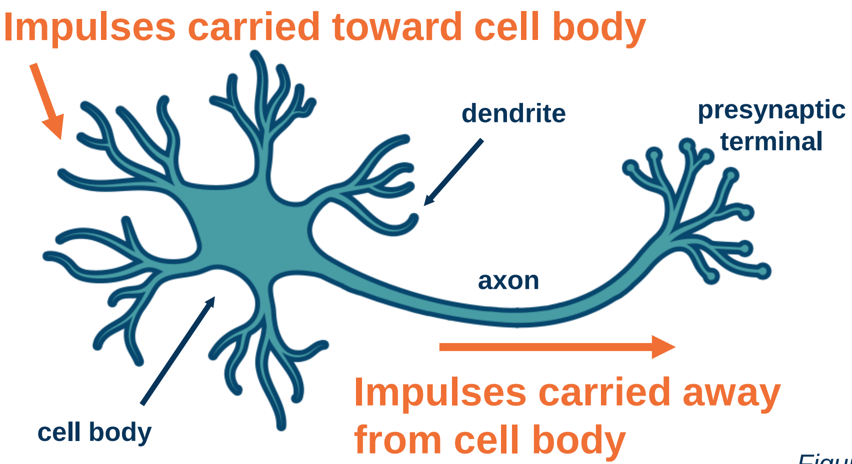

*Figures adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

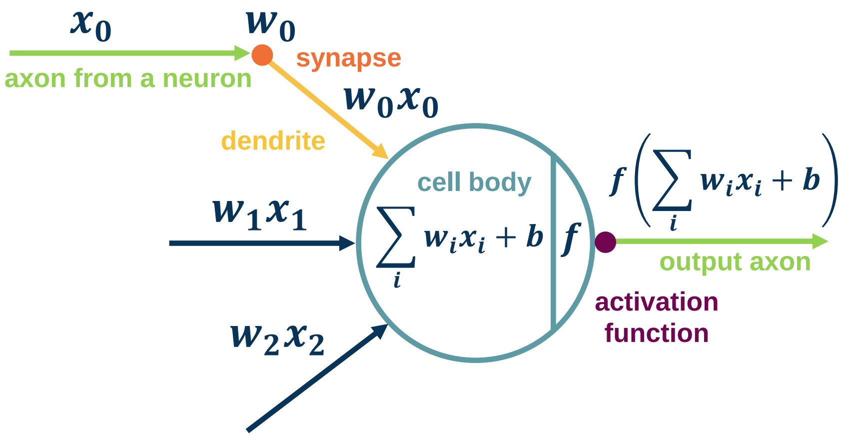
As we did before, the output of a neuron can be modulated by a non-linear function (e.g. sigmoid)




**Sigmoid Activation Function**

$$\frac{1}{1+e^{-x}}$$




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

## Adding Non-Linearities

Georgia Tech

We can have **multiple** neurons connected to the same input

Corresponds to a multi-class classifier

- Each output node outputs the score for a class

$$f(x, W) = \sigma(\, Wx + b) \quad \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b2 \\ w_{21} & w_{22} & \cdots & w_{3m} & b3 \end{bmatrix}$$

- Often called fully connected layers
  - Also called a linear *projection layer*

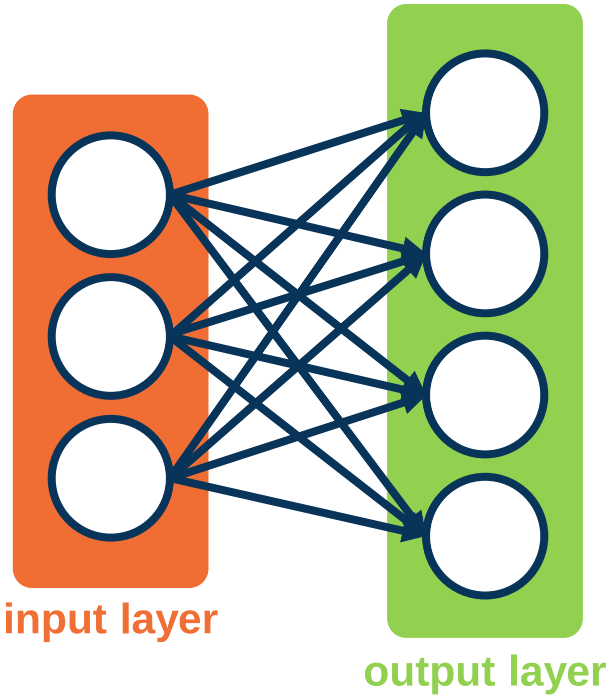



*Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

- Each input/output is a **neuron (node)**
- A linear classifier (+ optional non-linearity) is called a **fully connected** layer
- Connections are represented as **edges**
- Output of a particular neuron is referred to as **activation**
- This will be expanded as we view computation in a neural network as a **graph**

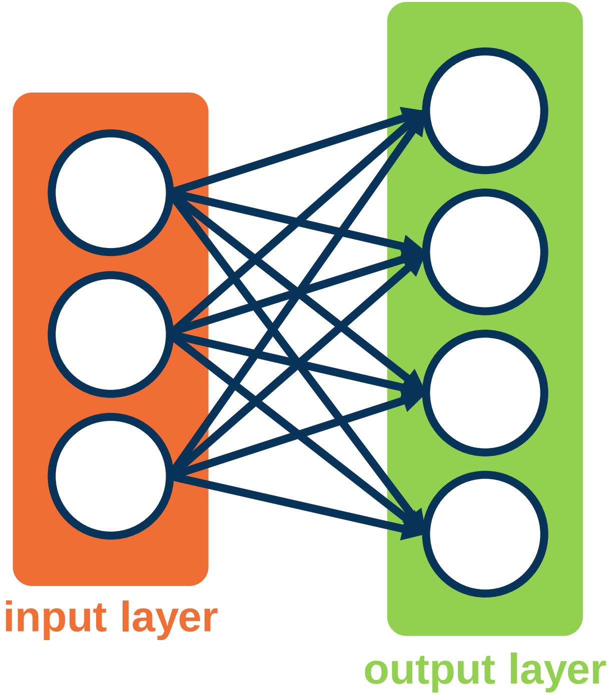



*Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# Neural Network Terminology

We can **stack** multiple layers together

- Input to second layer is output of first layer

Called a **2-layered neural network** (input is not counted)

Because the middle layer is neither input or output, and we don't know what their values represent, we call them **hidden** layers

- We will see that they end up learning effective features

This **increases** the representational power of the function!

- Two layered networks can represent any continuous function

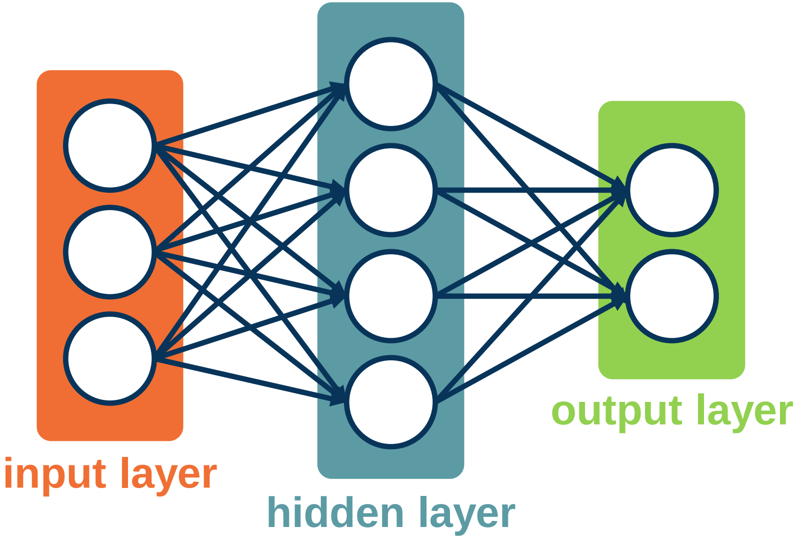
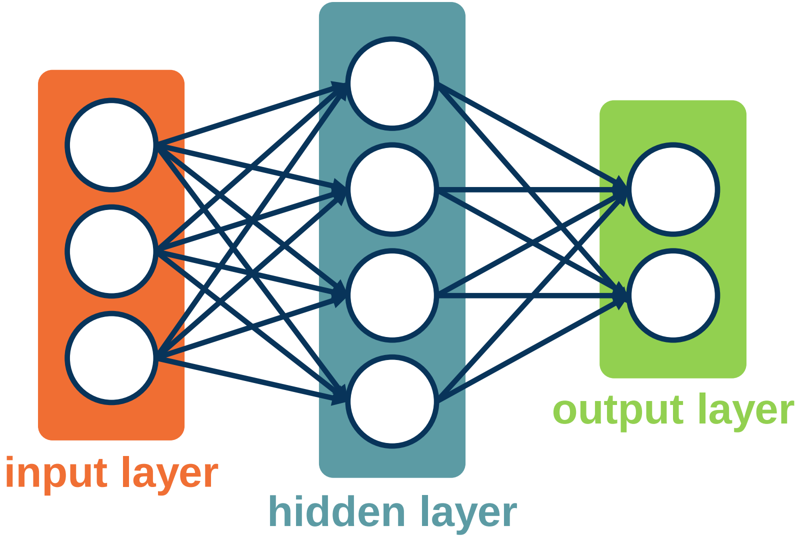



*Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# Connecting Many Layers

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)




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

$$=$$

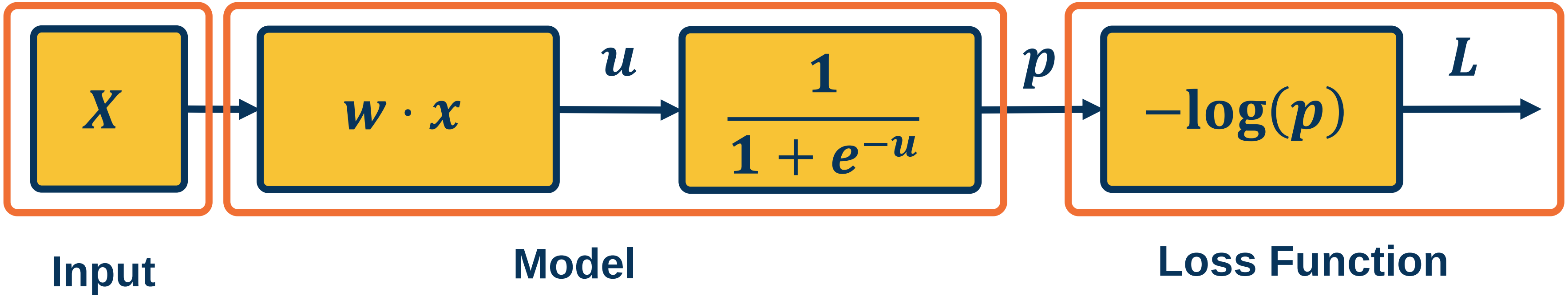
$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{W}_2) = \boldsymbol{\sigma}(\boldsymbol{W}_2 \boldsymbol{\sigma}(\boldsymbol{W}_1 \boldsymbol{x}))$$

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

A **linear classifier** can be broken down into:

- Input
- A function of the input
- A loss function

It's all just one function that can be decomposed into building blocks

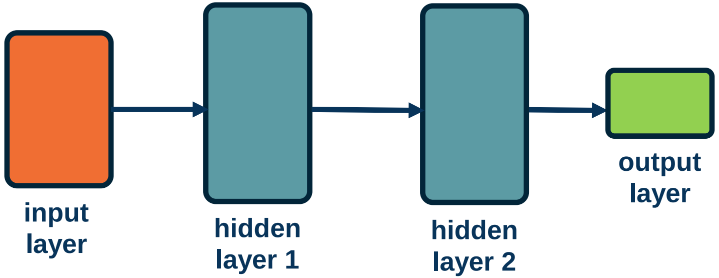


**Large (deep) networks** can be built by adding more and more layers

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**:



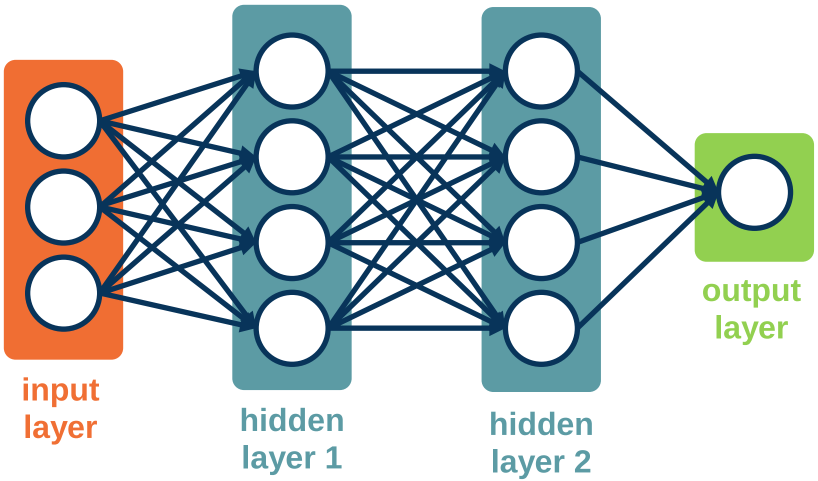




*Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

## Adding More Layers!

# Computation Graphs

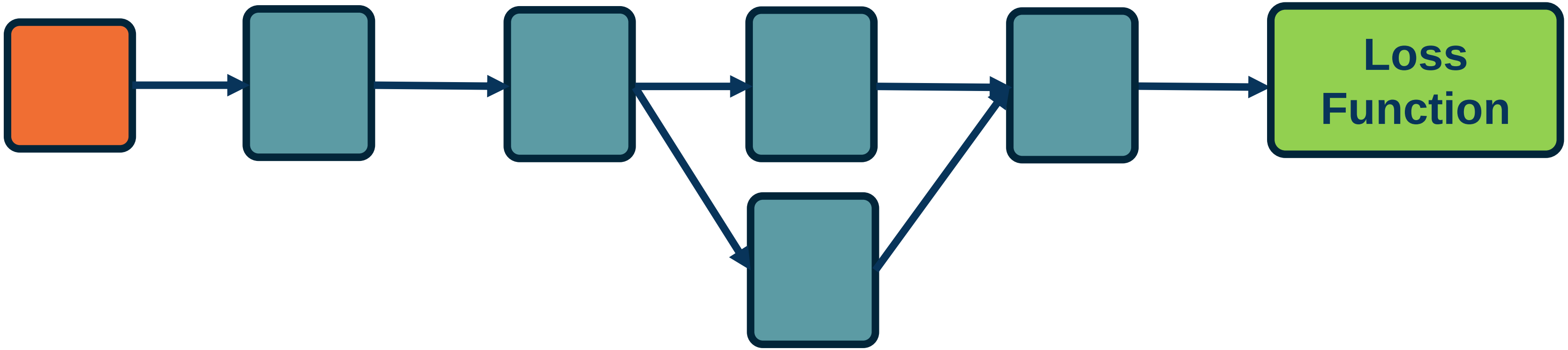
Functions can be made **arbitrarily complex** (subject to memory and computational limits), e.g.:

$$f(x, W) = \sigma(W_5\sigma(W_4\sigma(W_3\sigma(W_2\sigma(W_1x))$$

We can use **any type of differentiable function (layer)** we want!

- At the end, **add the loss function**

Composition can have **some structure**



The world is **compositional**!

We want our **model** to reflect this

Empirical and theoretical evidence that it makes **learning complex functions easier**

Note that **prior state of art engineered features** often had this compositionality as well

**VISION**

pixels → edge → texton → motif → part → object

**SPEECH**

sample → spectral band → formant → motif → phone → word

**NLP**

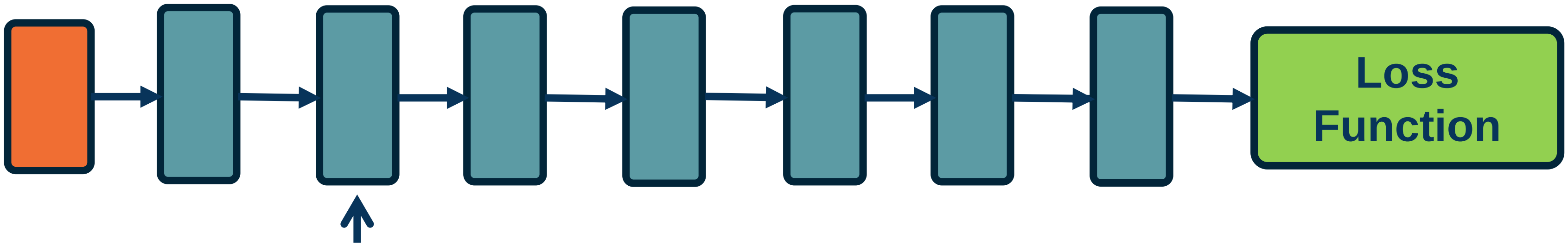
character → word → NP/VP/.. → clause → sentence → story

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

- **Pixels -> edges -> object parts -> objects**

# Compositionality

- 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




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

Given a library of simple functions

$\sin(x)$

$\log(x)$

$\cos(x)$

$x^3$

$\exp(x)$

Compose into a

complicate function

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

$$\boxed{w\cdot x} \xrightarrow{u} \boxed{\frac{1}{1+e^{-u}}} \xrightarrow{p} \boxed{-\log(p)} \xrightarrow{L}$$

*Adapted from slides by: Marc'Aurelio Ranzato, Yann LeCun*

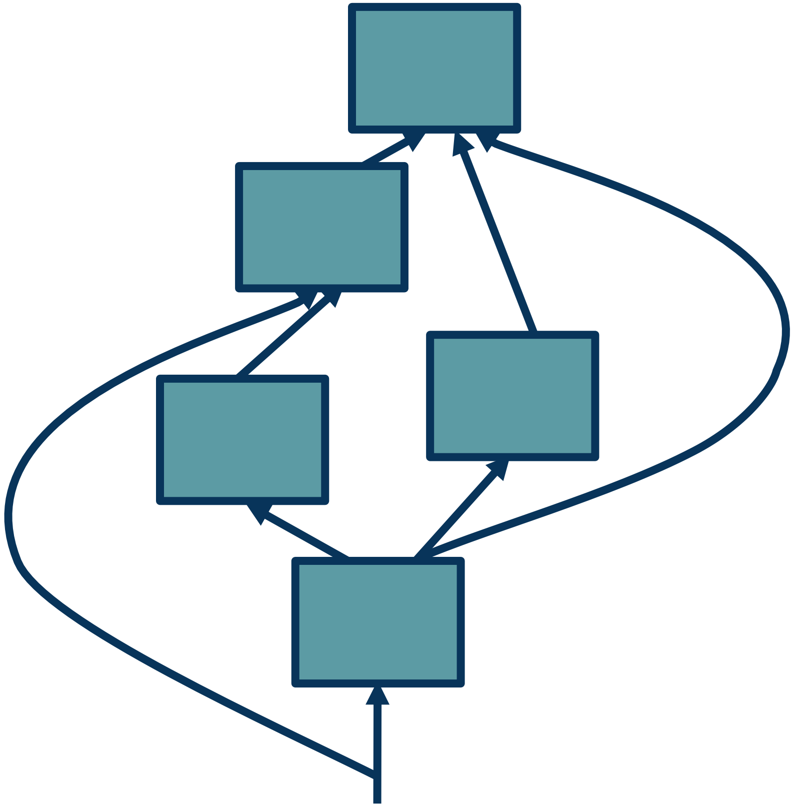
# Decomposing a Function

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

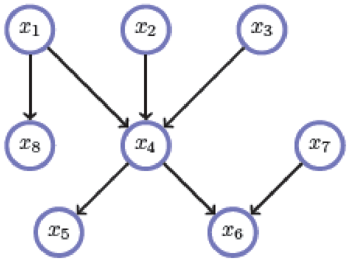
A **training algorithm** will then process this graph, **one module at a time**

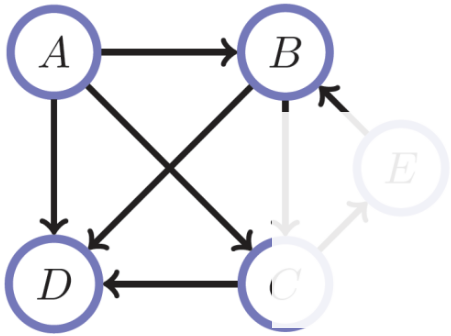


*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

# Directed Acyclic Graphs (DAGs)

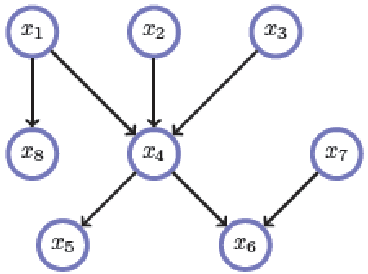
- Exactly what the name suggests
  - Directed edges
  - No (directed) cycles
  - Underlying undirected cycles okay

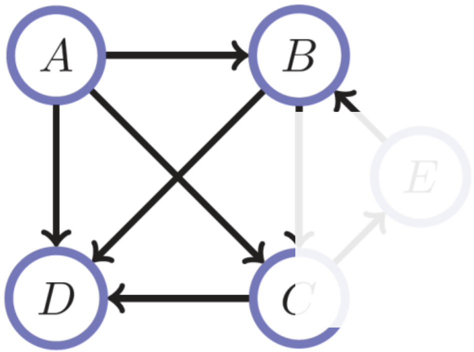


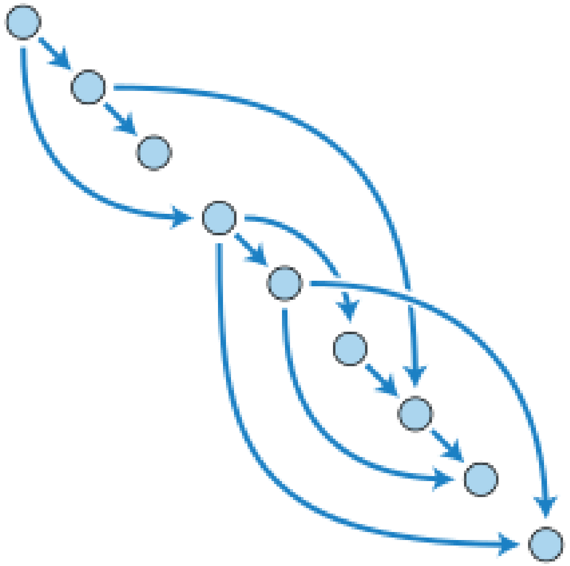
# Directed Acyclic Graphs (DAGs)

- Concept
  - Topological Ordering



# Directed Acyclic Graphs (DAGs)



# Backpropagation

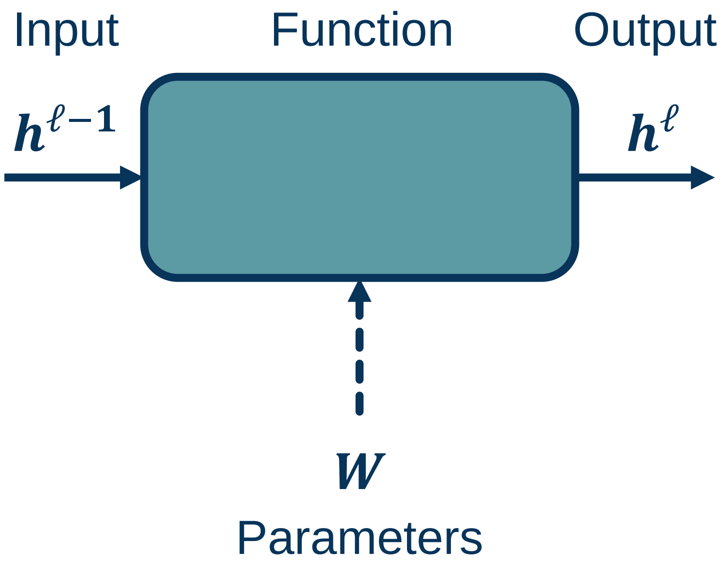
Given this computation graph, the training algorithm will:

- Calculate the current model's outputs (called the **forward pass**)
- Calculate the gradients for each module (called the **backward pass**)

Backward pass is a recursive algorithm that:

- Starts at **loss function** where we know how to calculate the gradients
- Progresses back through the modules
- Ends in the **input layer** where we do not need gradients (no parameters)

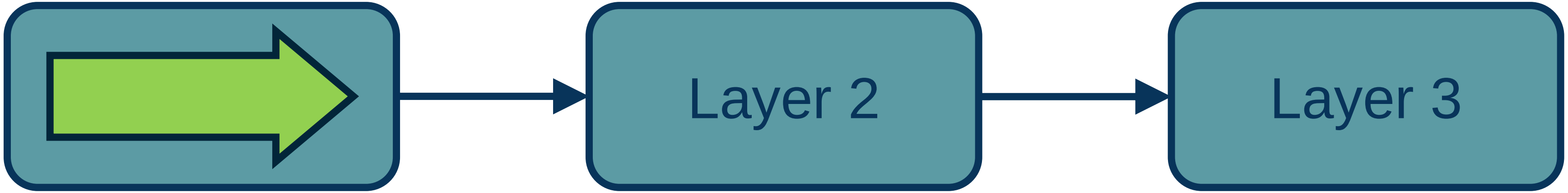
This algorithm is called **backpropagation**




*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

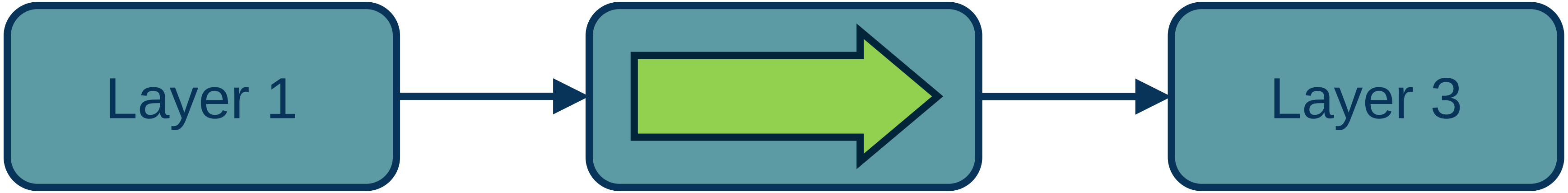
# Overview of Training

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



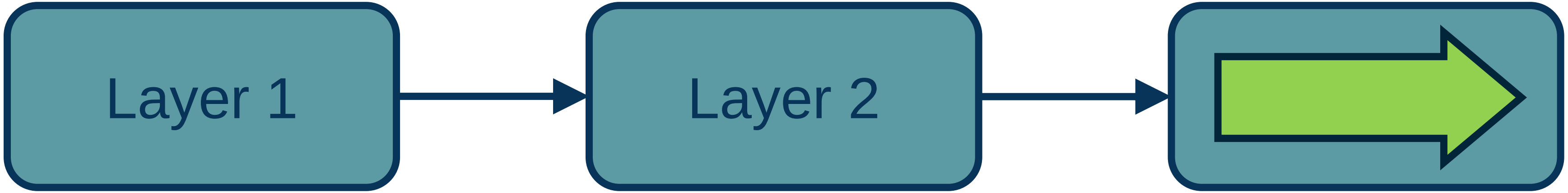

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

## **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**



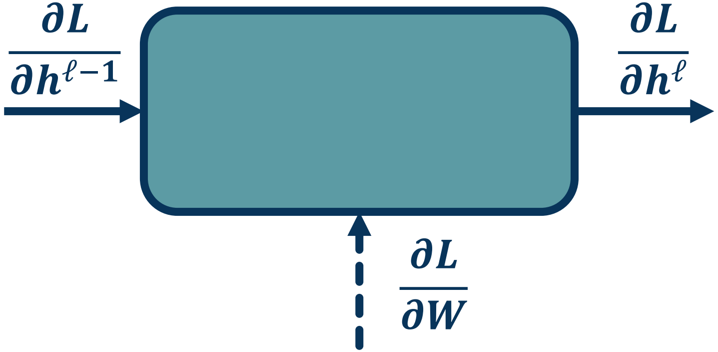

Note that we must store the **intermediate outputs of all layers**!

- 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*

In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module's parameters

- Assume that we have the gradient of the loss with respect to the **module's outputs** (given to us by upstream module)
- We will also pass the gradient of the loss with respect to the **module's inputs**
  - This is not required for update the module's weights, but passes the gradients back to the previous module



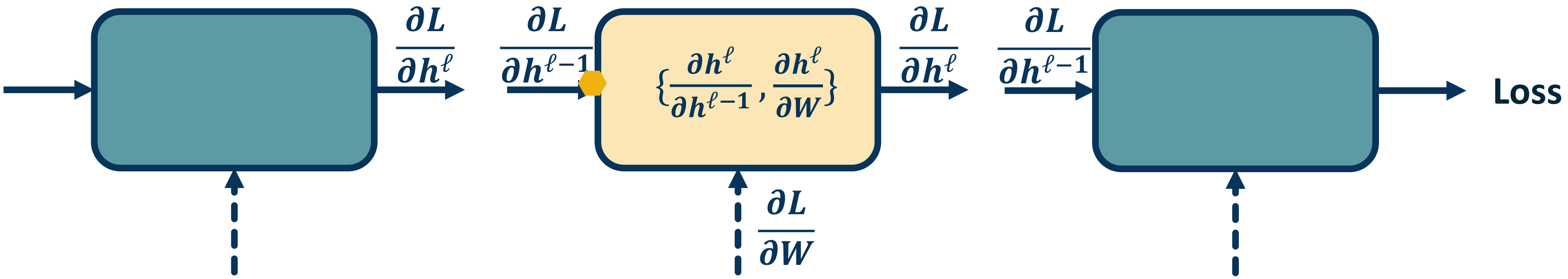

**Problem:**

- We can compute local gradients: $\{\frac{\partial h^{\ell}}{\partial h^{\ell-1}}, \frac{\partial h^{\ell}}{\partial W}\}$
- We are given: $\frac{\partial L}{\partial h^{\ell}}$
- Compute: $\{\frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W}\}$

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

# Backward Pass Computations

- 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 can compute **local gradients**: $\{\frac{\partial \boldsymbol{h}^{\ell}}{\partial \boldsymbol{h}^{\ell-1}}, \frac{\partial \boldsymbol{h}^{\ell}}{\partial \boldsymbol{W}}\}$

- This is just the **derivative of our function** with respect to its parameters and inputs!

**Example:** If $\boldsymbol{h}^{\ell} = \boldsymbol{W}\boldsymbol{h}^{\ell-1}$

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

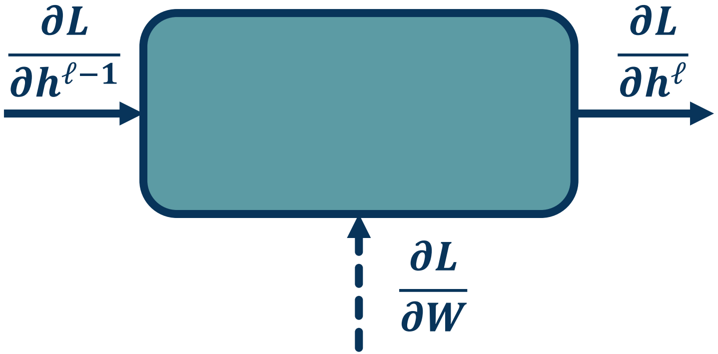
and $\frac{\partial \boldsymbol{h}_i^{\ell}}{\partial \boldsymbol{w}_i} = \boldsymbol{h}^{\ell-1,T}$

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

Given by upstream module **(upstream gradient)**

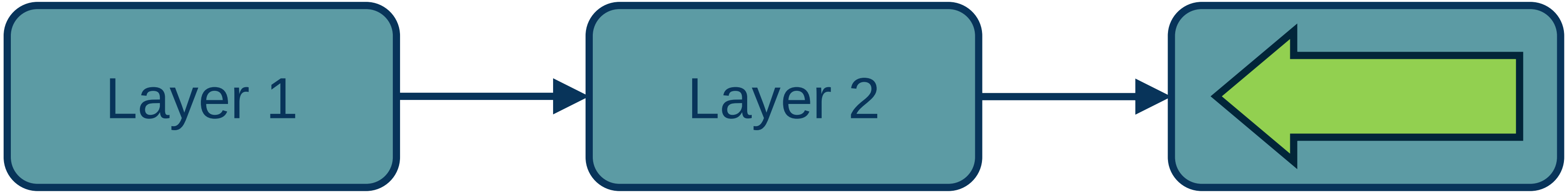
- **Gradient of loss w.r.t. weights:** $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial W}$




*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

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

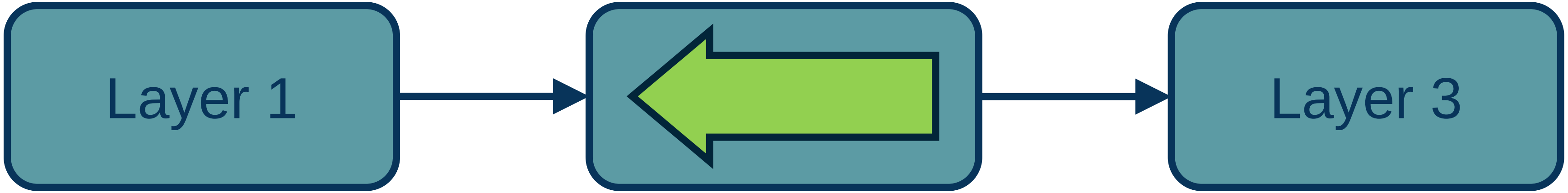
**Step 2:** Compute Gradients wrt parameters: **Backward Pass**




*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

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

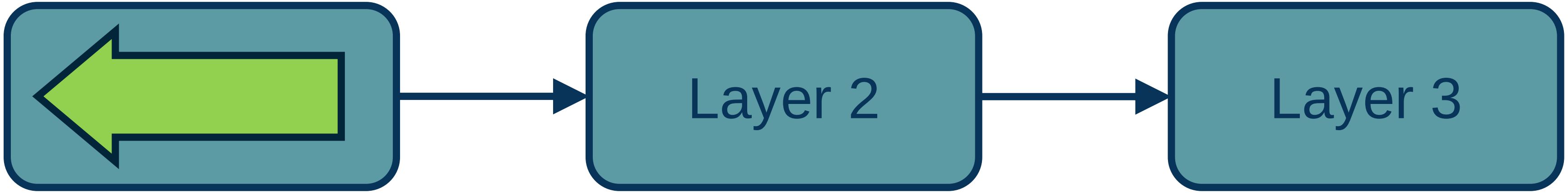
**Step 2:** Compute Gradients wrt parameters: **Backward Pass**




*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**

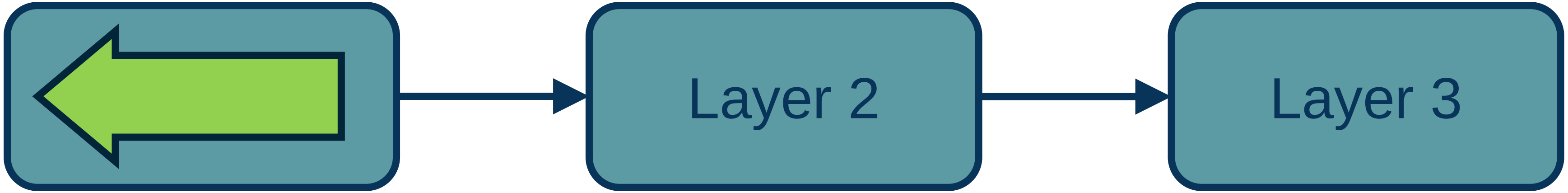



*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

**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!**

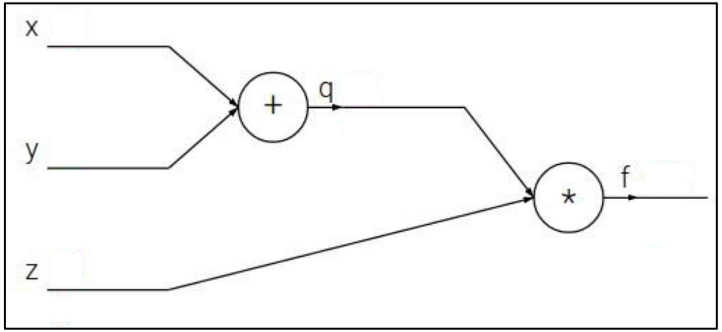
*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*

# Backpropagation: a simple example

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

# Backpropagation: a simple example

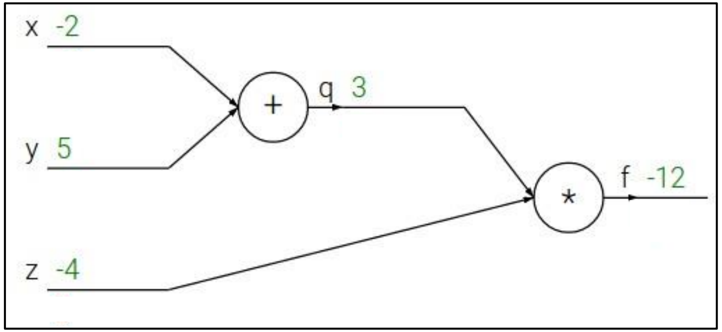
$$f(x, y, z) = (x + y)z$$



# Backpropagation: a simple example

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

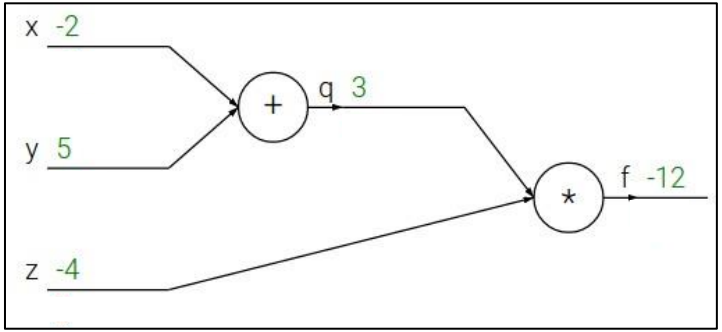
e.g. x = -2, y = 5, z = -4



# Backpropagation: a simple example

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

e.g. x = -2, y = 5, z = -4




Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



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

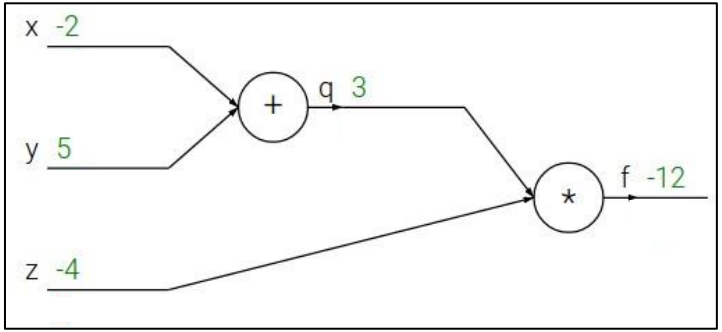
# Backpropagation: a simple example

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

e.g. x = -2, y = 5, z = -4

$$\boxed{q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1}$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



# Backpropagation: a simple example

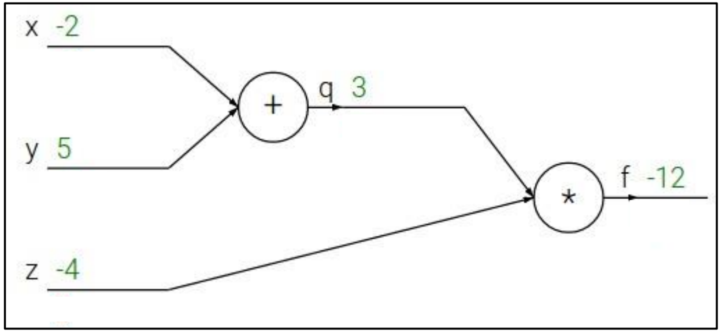
$$f(x, y, z) = (x + y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



# Backpropagation: a simple example

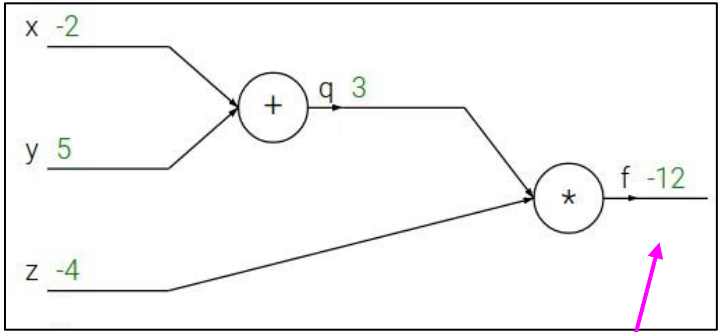
$$f(x,y,z) = (x+y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial f}$$

# Backpropagation: a simple example

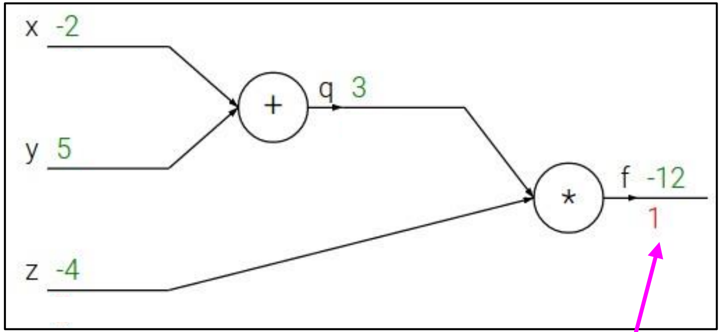
$$f(x, y, z) = (x + y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial f}$$

# Backpropagation: a simple example

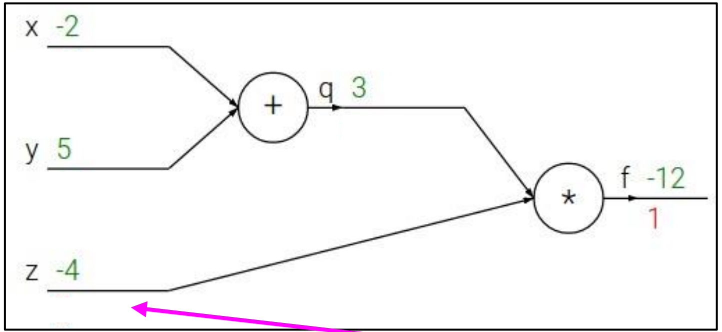
$$f(x, y, z) = (x + y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial z}$$

# Backpropagation: a simple example

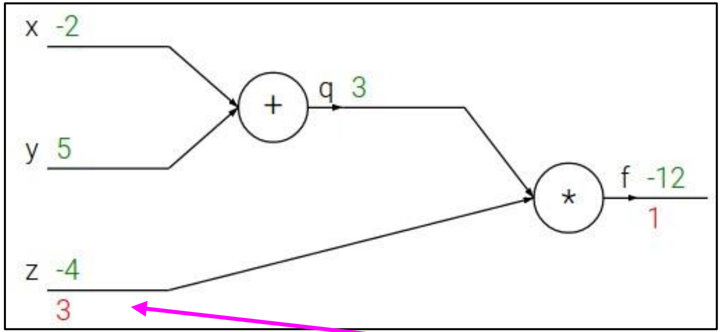
$$f(x, y, z) = (x + y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial z}$$

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

# Backpropagation: a simple example

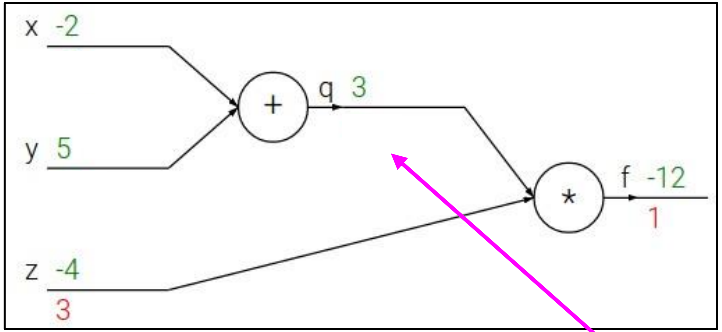
$$f(x, y, z) = (x + y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial q}$$

# Backpropagation: a simple example

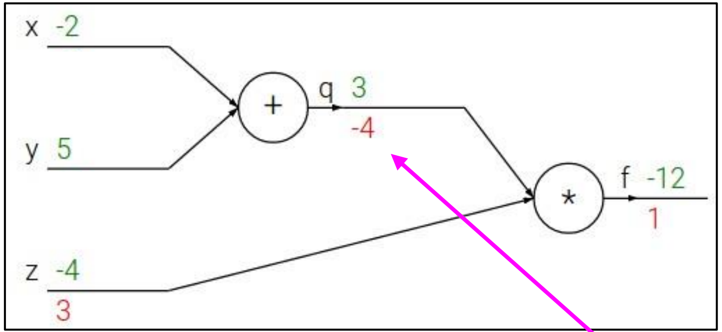
$$f(x, y, z) = (x + y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial q}$$

# Backpropagation: a simple example

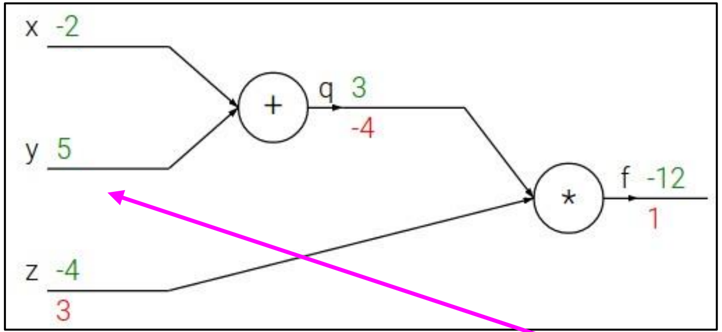
$$f(x, y, z) = (x + y)z$$

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:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}}$$

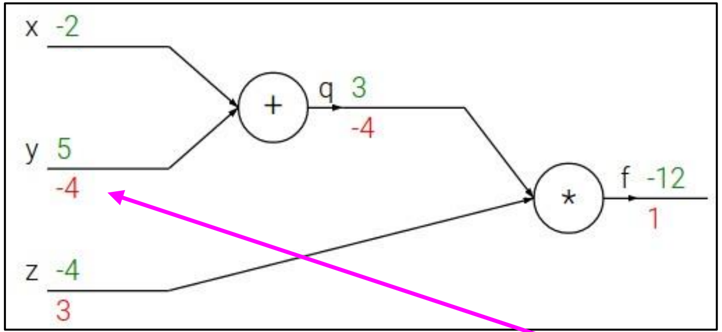
Upstream gradient

Local gradient

# 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: $\dfrac{\partial f}{\partial x}, \dfrac{\partial f}{\partial y}, \dfrac{\partial f}{\partial z}$

$$\boxed{\frac{\partial f}{\partial y}}$$

Chain rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}}$$

Upstream gradient

Local gradient

# Backpropagation: a simple example

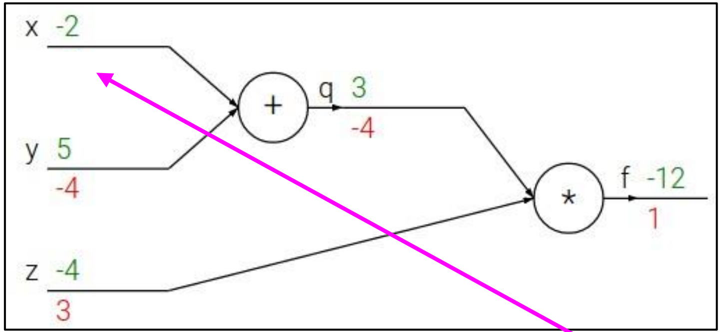
$$f(x,y,z) = (x+y)z$$

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: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




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

Chain rule:

$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}}$$

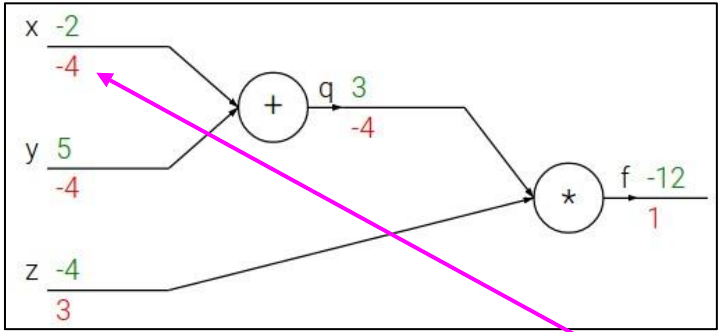
Upstream gradient

Local gradient

# Backpropagation: a simple example

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

e.g. x = -2, y = 5, z = -4




$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$\frac{\partial f}{\partial x}$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

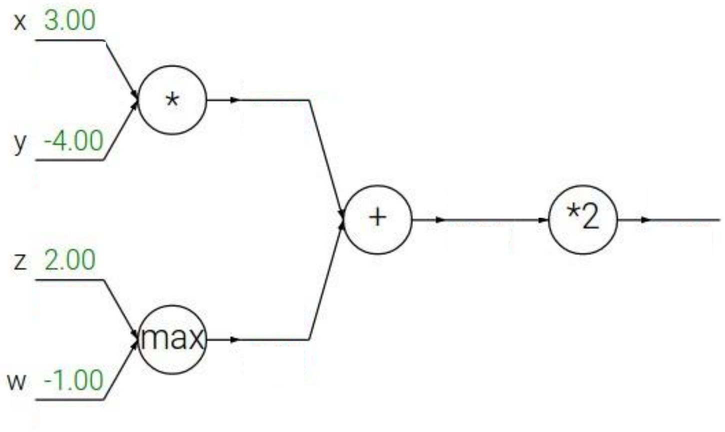
$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}}$$

Upstream gradient

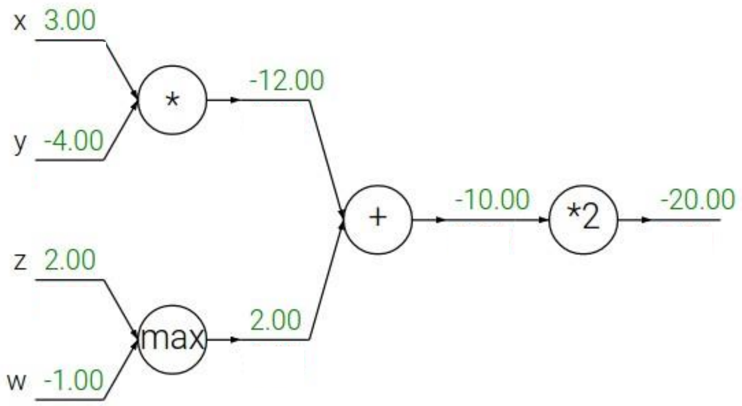
Local gradient

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

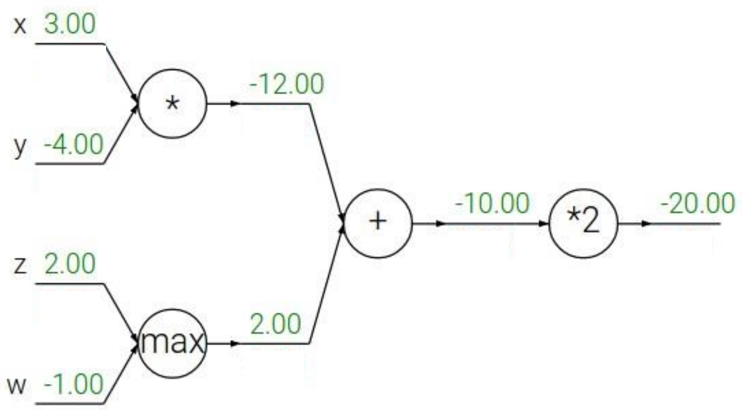
# Backpropagation: a simple example



# Backpropagation: a simple example

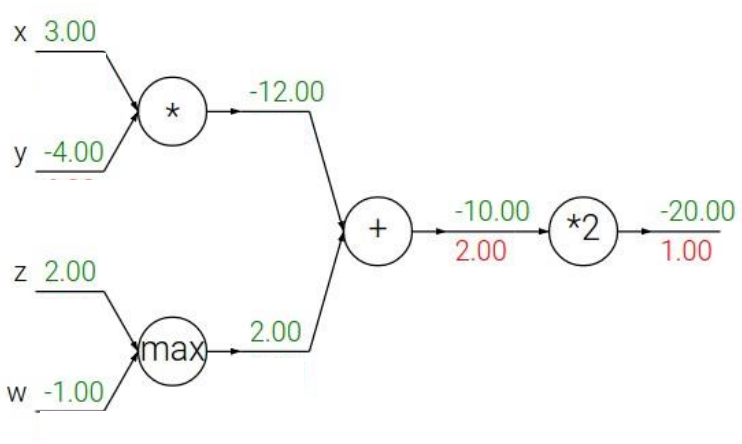


# Patterns in backward flow



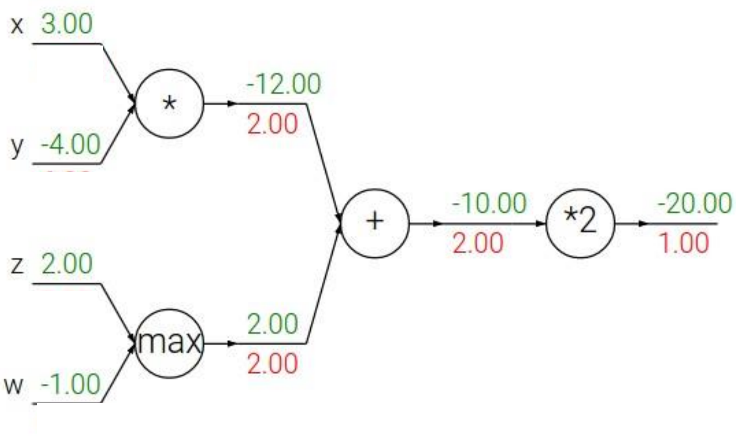
# Patterns in backward flow

Q: What is an **add** gate?



# Patterns in backward flow

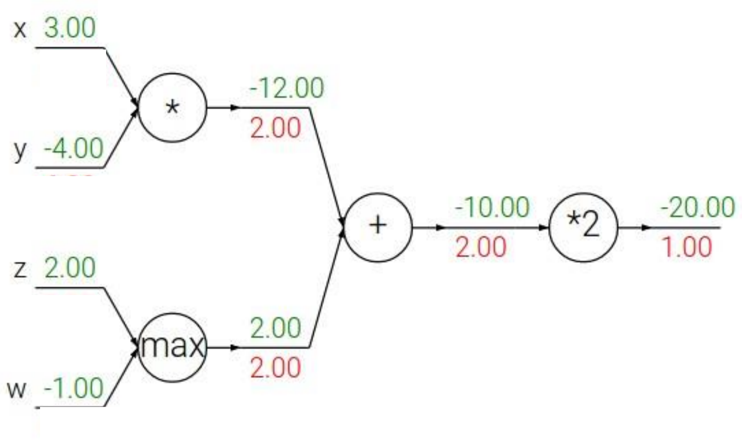
**add** gate: gradient distributor



# Patterns in backward flow

**add** gate: gradient distributor

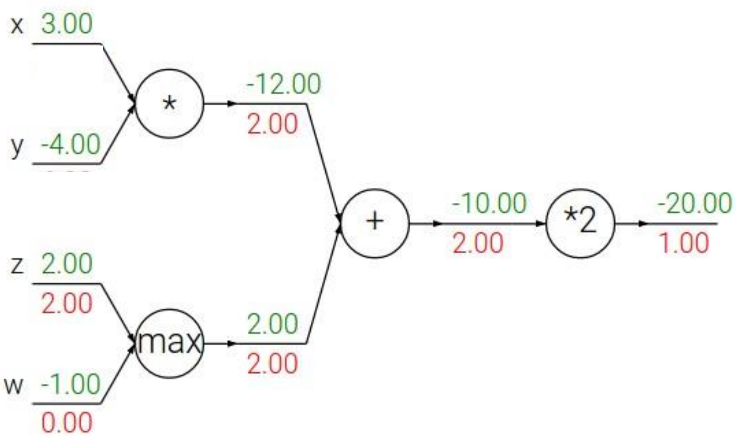
Q: What is a **max** gate?



# Patterns in backward flow

**add** gate: gradient distributor

**max** gate: gradient router

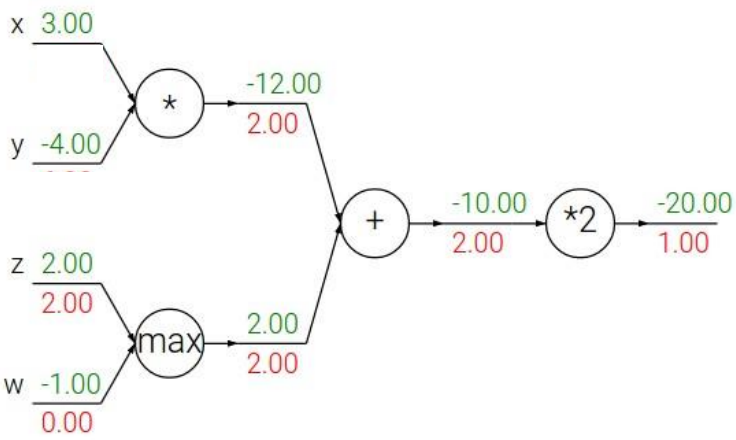


# Patterns in backward flow

**add** gate: gradient distributor

**max** gate: gradient router

Q: What is a **mul** gate?

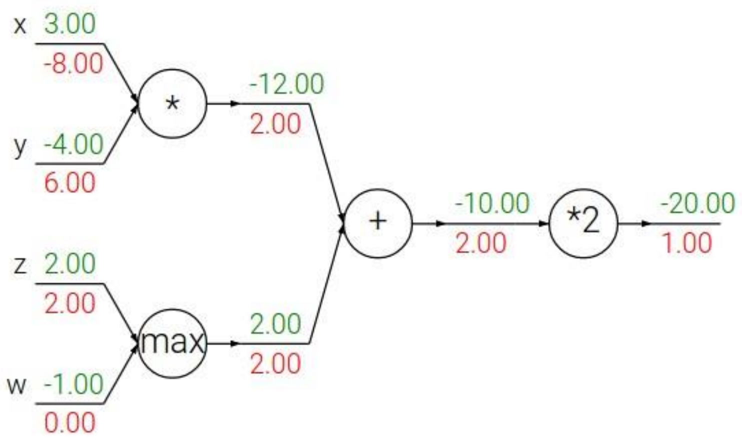


# Patterns in backward flow

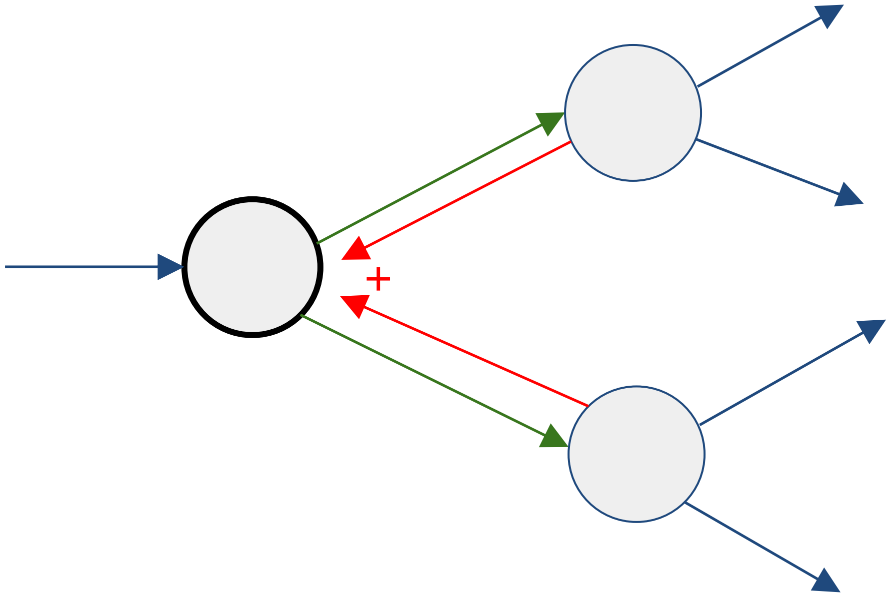
**add** gate: gradient distributor

**max** gate: gradient router

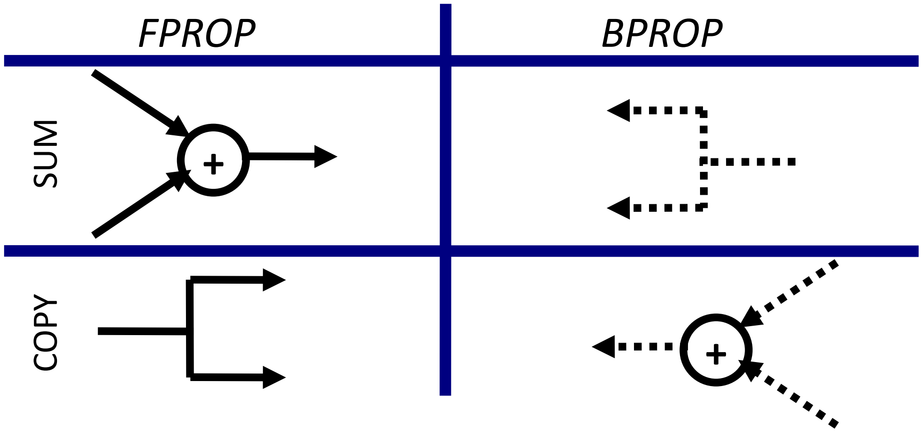
**mul** gate: gradient switcher



# Gradients add at branches



# Duality in Fprop and Bprop

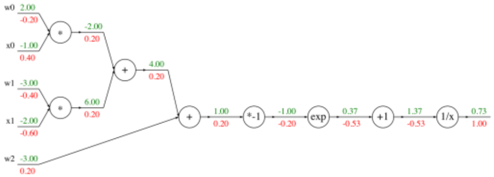


# Deep Learning = Differentiable Programming

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

- What do we need to do?
  - Generic code for representing the graph of modules
  - Specify modules (both forward and backward function)

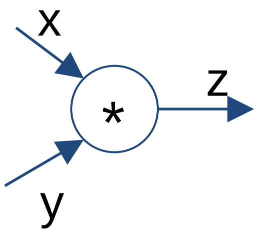
# Modularized implementation: forward / backward API




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

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

# Modularized implementation: forward / backward API



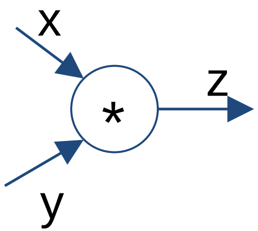

(x,y,z are scalars)

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

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

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

# 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 BIGene/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 |

# Caffe Sigmoid Layer

```cpp
#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)