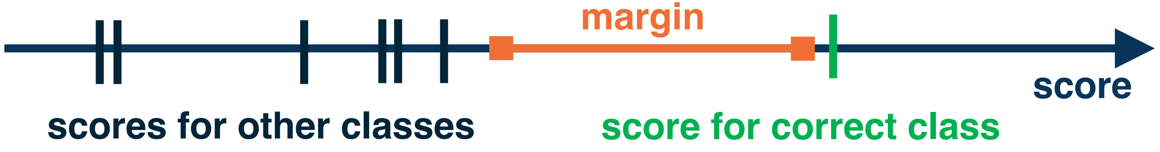
CS 4644-DL / 7643-A
DANFEI XU
(SLIDE CREDIT: PROF. ZOLT KIRA)

Topics:

- Backpropagation
- Computation Graph and Automatic Differentiation

Recap: Multiclass SVM loss

Loss = 0:



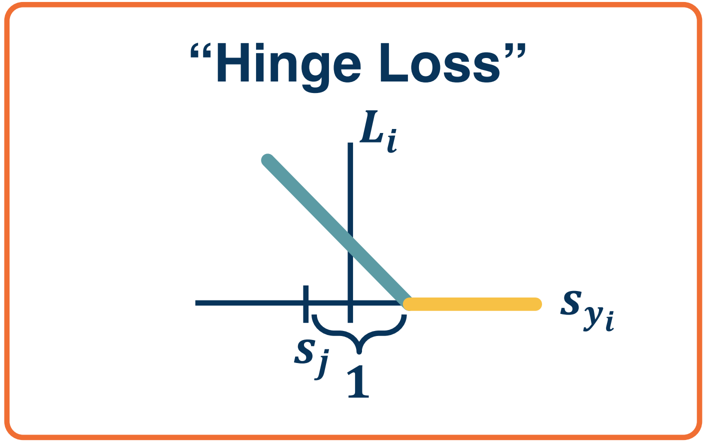
Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the
scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

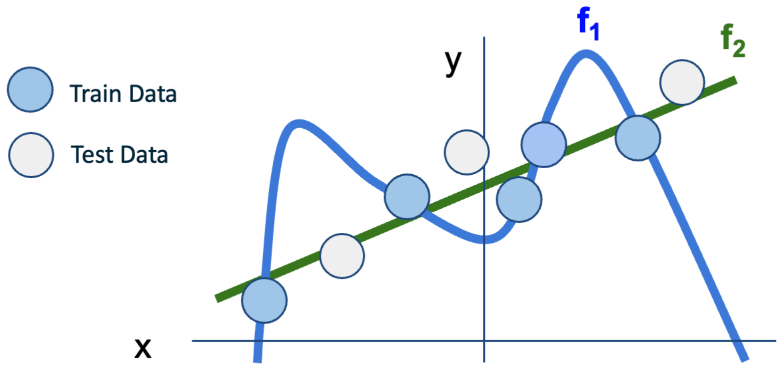
$$= \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$



Recap: Regularization

Q: How do we pick between W and 2W?
A: Opt for simpler functions to avoid overfit

How? Regularization!



$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

λ = regularization strength (hyperparameter)

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Recap: Softmax Classifier and Cross Entropy Loss

Want to interpret raw classifier scores as **probabilities**

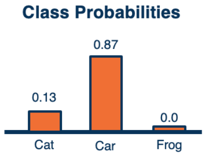
$$\boxed{p_\theta(\boldsymbol{Y} = \boldsymbol{y_i} | \boldsymbol{X} = \boldsymbol{x_i}) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}}$$

Softmax Function

cat	**3.2**
car	5.1
frog	-1.7

Raw class scores

→



How do we optimize the classifier? We maximize the probability of $\boldsymbol{p_\theta(y_i|x_i)}$!

1. Maximum Likelihood Estimation (MLE):
Choose weights to maximize the likelihood of observed data. In this case, the loss function is the **Negative Log-Likelihood (NLL)**.

Finding a set of weights θ that maximizes the probability of correct prediction: $\underset{\theta}{\operatorname{argmax}} \prod p_\theta(y_i|x_i)$

This is equivalent to:

$$\underset{\theta}{\operatorname{argmax}} \sum \ln p_\theta(y_i|x_i)$$

$$L_i = -\ln p_\theta(y_i|x_i) = -\ln\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

2. Information theory view:
Derive NLL from the cross entropy measurement. Also known as the cross-entropy loss

Cross Entropy: $H(p,q) = -\sum p(x) \ln q(x)$

Cross Entropy Loss -> NLL

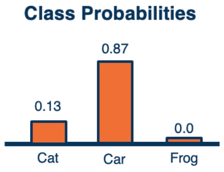
$$\boldsymbol{H_i(p, p_\theta)} = -\sum_{y \in Y} \boldsymbol{p(y|x_i)} \ln \boldsymbol{p_\theta(y|x_i)}$$
$$= -\ln \boldsymbol{p_\theta(y_i|x_i)}$$

$$\boldsymbol{L} = \sum \boldsymbol{H_i(p, p_\theta)} = -\sum \ln \boldsymbol{p_\theta(y_i|x_i)} \equiv \boldsymbol{NLL}$$

Q: Why softmax?

cat	3.2
car	5.1
frog	-1.7

Raw class scores



Why this?

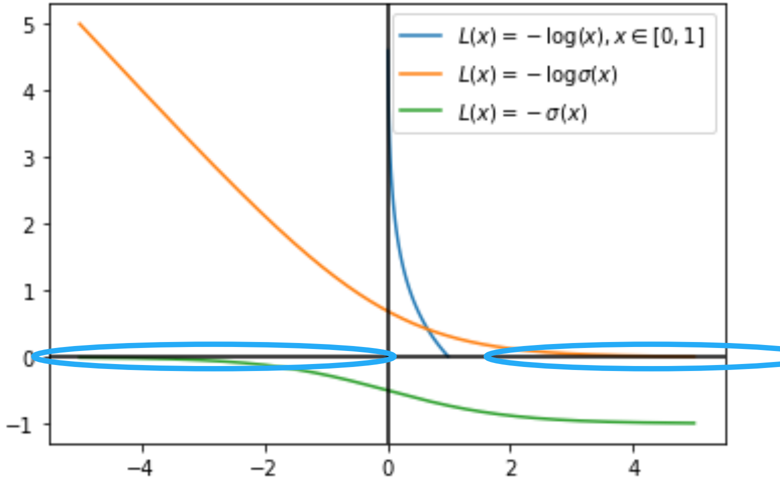
$$p_\theta(Y = y_i | X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$$

Use logistic function as example. Same as softmax but for binary classification

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

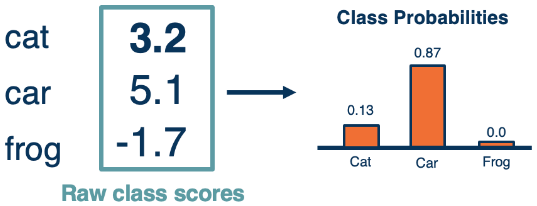
Consider the following three basis for NLL:

1. Squash and clip value to (0, 1]
2. Logistic function
3. Logistic function but no log (just negative likelihood)



1. Squash & clip: no loss, no learning!

Q: Why softmax?



Why this?

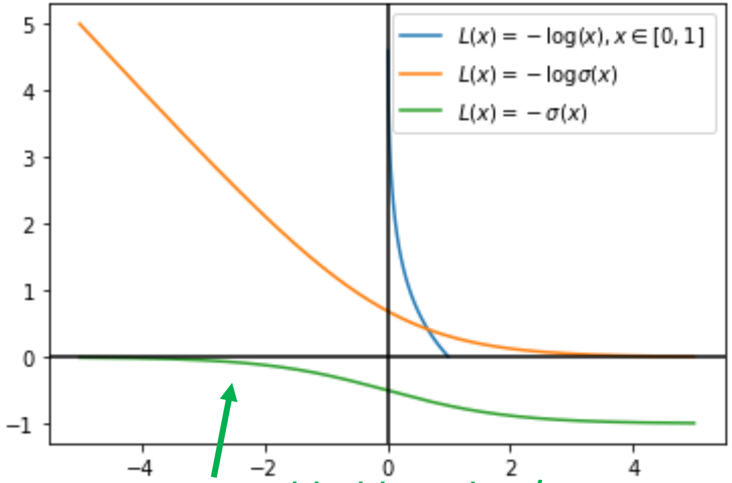
$$p_\theta(Y = y_i | X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$$

Use logistic function as example. Same as softmax but for binary classification

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

Consider the following three basis for NLL:

1. Squash and clip value to (0, 1]
2. Logistic function
3. Logistic function but no log (just negative likelihood)

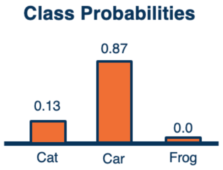


3. Negative likelihood w/ logistic function: saturated loss when classifier is very wrong

Q: Why softmax?

cat	**3.2**
car	5.1
frog	-1.7

Raw class scores



Why this?

$$p_\theta(Y = y_i | X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$$

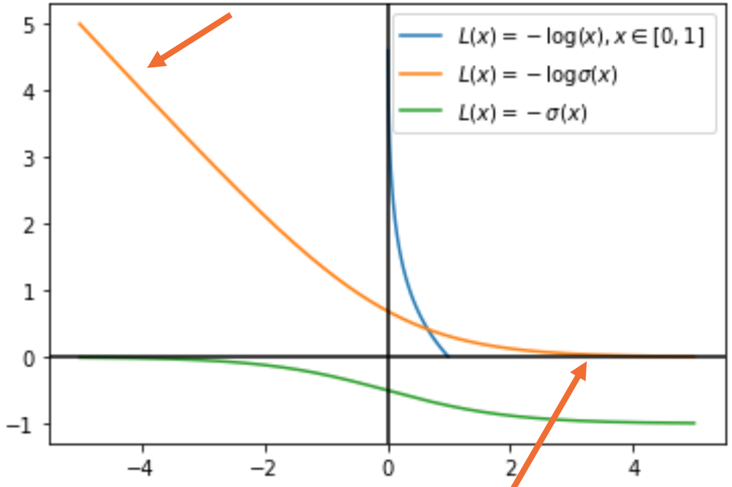
Use logistic function as example. Same as softmax but for binary classification

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

Consider the following three basis for NLL:

1. Squash and clip value to (0, 1]
2. Logistic function
3. Logistic function but no log (just negative likelihood)

2. NLL w/ logistic: Strong guidance when classifier is wrong

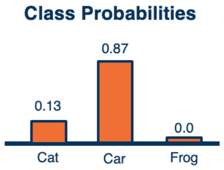


Only saturate at convergence, e.g., $\sigma(3) \approx 0.95$

Q: Why softmax?

cat	**3.2**
car	5.1
frog	-1.7

Raw class scores



Why this?

$$p_\theta(Y = y_i | X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$$

Use logistic function as example. Same as softmax but for binary classification

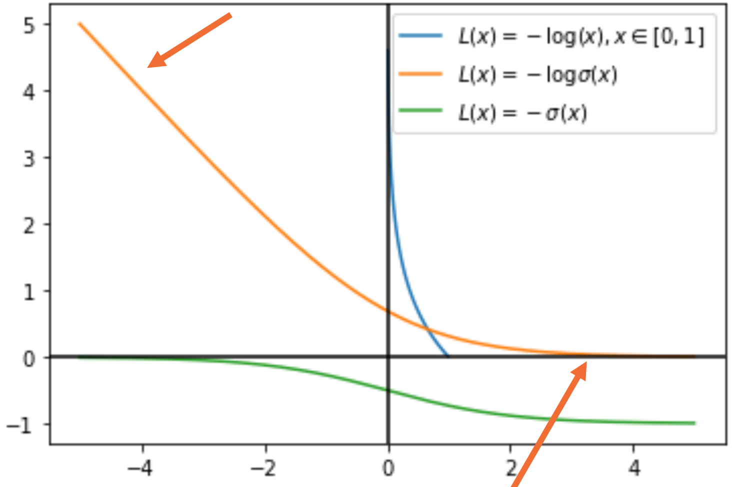
$$\sigma(x) = \frac{e^x}{1 + e^x}$$

Consider the following three basis for NLL:

1. Squash and clip value to (0, 1]
2. Logistic function
3. Logistic function but no log (just negative likelihood)

A: Many ways to get probabilities. Logistic function / softmax make the NLL loss behave well for optimization.

2. NLL w/ logistic: Strong guidance when classifier is wrong



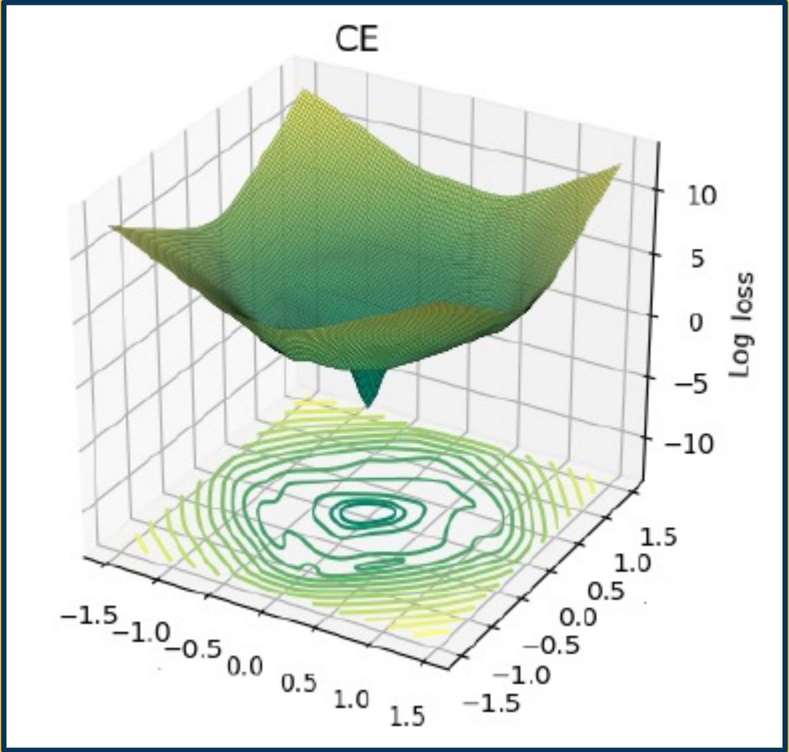
Only saturate at convergence, e.g., $\sigma(3) \approx 0.95$

Recap: gradient-based optimization

As weights change, the gradients change as well

- This is often somewhat-smooth locally, so small changes in weights produce small changes in the loss

We can therefore think about **iterative algorithms** that take **current values of weights and modify them** a bit



Recap: The gradient descent algorithm

- 1. Choose a model: $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{W}) = \mathrm{Wx}$
- 2. Choose loss function: $\boldsymbol{L_i} = |\boldsymbol{y} - \boldsymbol{W}\boldsymbol{x_i}|^2$
- 3. Calculate partial derivative for each parameter: $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w_i}}$
- 4. Update the parameters: $\boldsymbol{w_i} = \boldsymbol{w_i} - \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w_i}}$
- 5. Add learning rate to prevent too big of a step: $\boldsymbol{w_i} = \boldsymbol{w_i} - \boldsymbol{\alpha}\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{w_i}}$
- **Repeat 3-5**

Recap: calculating gradients

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

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

- **Gradient** is multi-dimensional derivatives
- Steepest descent direction is the **negative gradient**
- **Intuitively:** Measures how the function changes as the argument a changes by a small step size
- **In Machine Learning:** Want to know how to minimize loss by changing parameters
  - Can consider each parameter separately by taking **partial derivative** of loss function with respect to that parameter

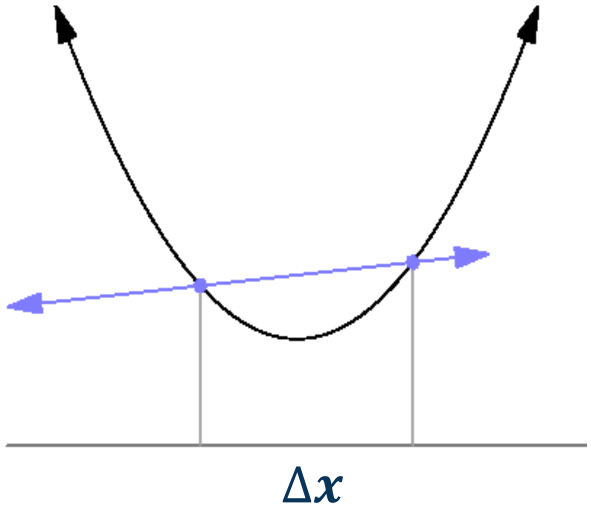


Image and equation from: https://en.wikipedia.org/wiki/Derivative #/media/File:Tangent_animation.gif

Hard to calculate analytical gradients for complex functions!

$\sin(x)$ $\log(x)$ $\cos(x)$ x^3 $\exp(x)$

Compose into a complicate function

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

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

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

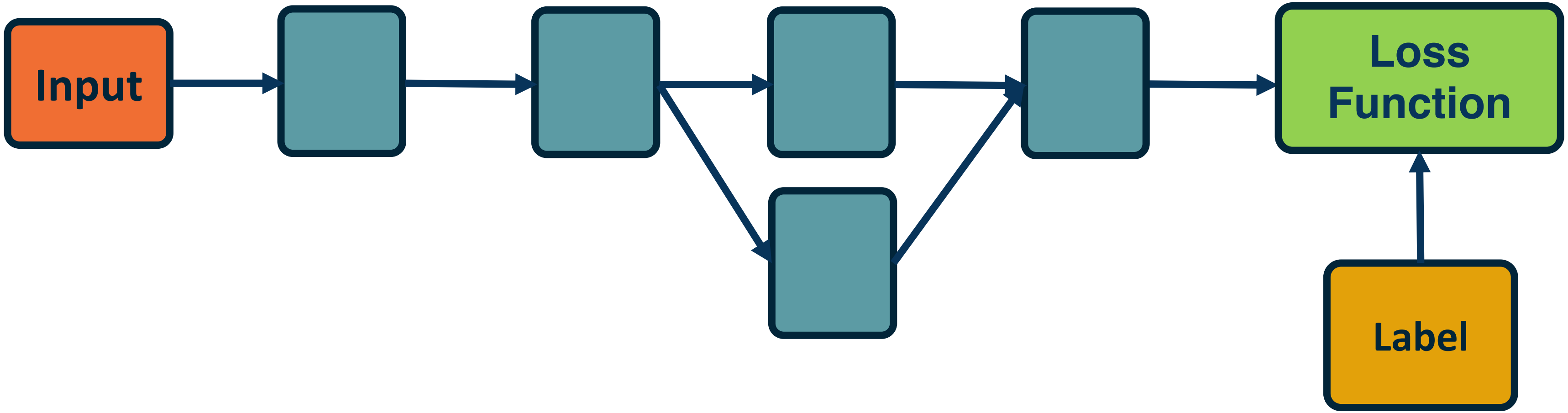
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w}$$

This time: Chain rule and Backpropagation!

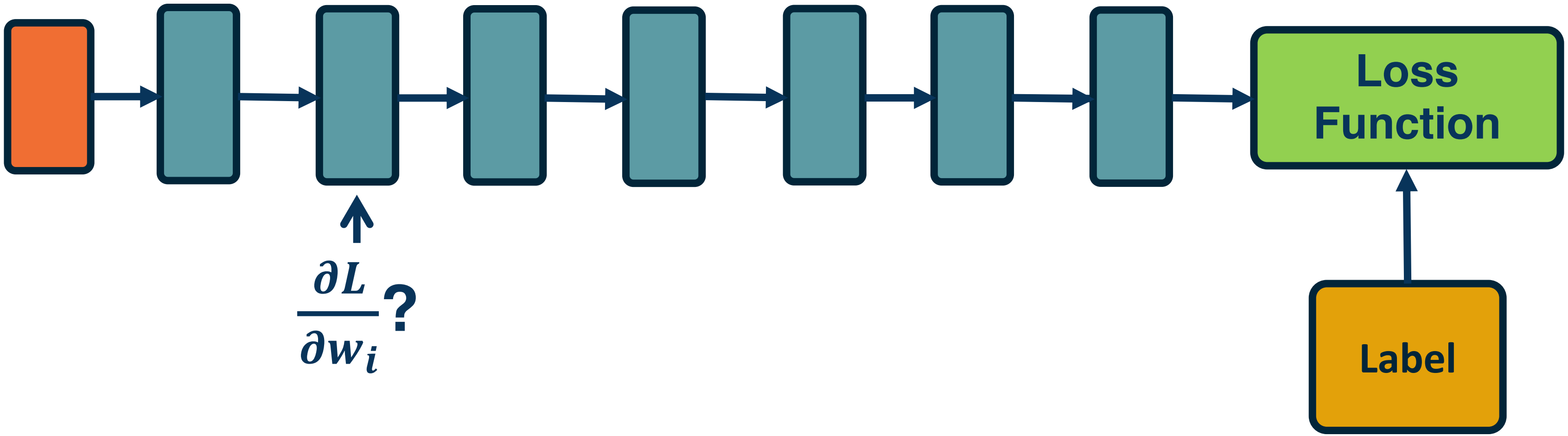
Functions can be made **arbitrarily complex** (subject to memory and computational limits), e.g.:

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

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



- 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 **are propagated** to affect the **loss function** at the end

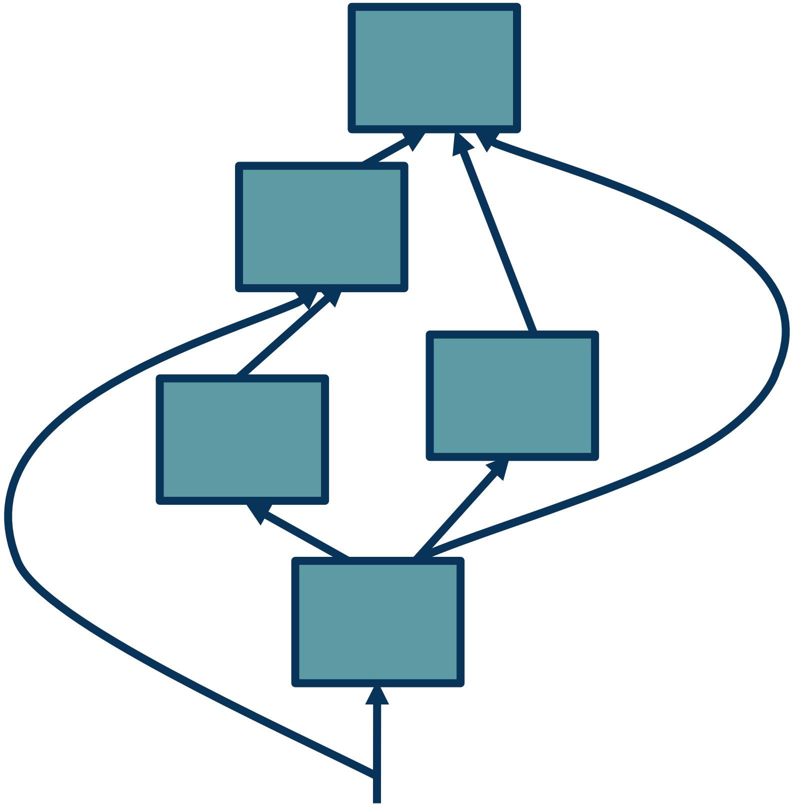


To develop a general algorithm for this, we will view the function as a **computation graph**

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

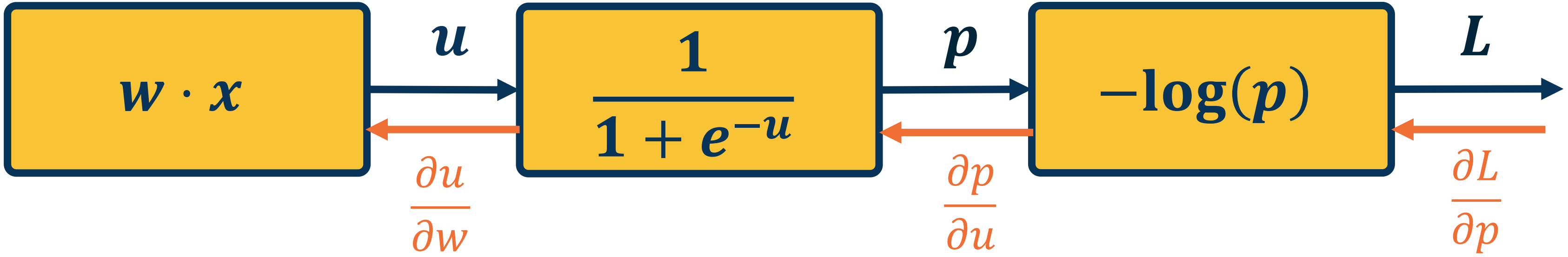
- Modules must be differentiable to support gradient computations for gradient descent

The **backpropagation algorithm** will then process this graph, **one module at a time**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

This is a computation graph!



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial u}\frac{\partial u}{\partial w}$$

Backpropagation (roughly):

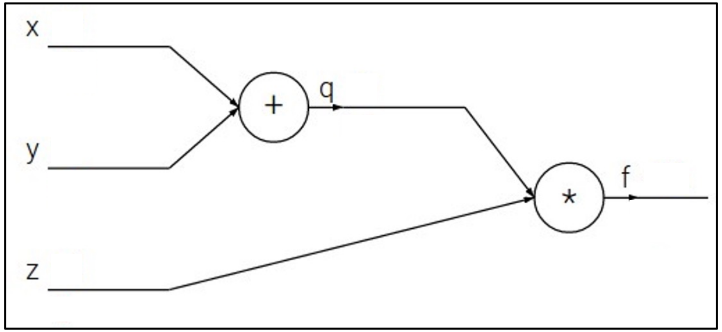
1. Calculate local gradients for each node (e.g., $\frac{\partial u}{\partial w}$)
2. Trace the computation graph (backward) to calculate the global gradients for each node w.r.t. to the loss function.

Backpropagation: a simple example

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

Backpropagation: a simple example

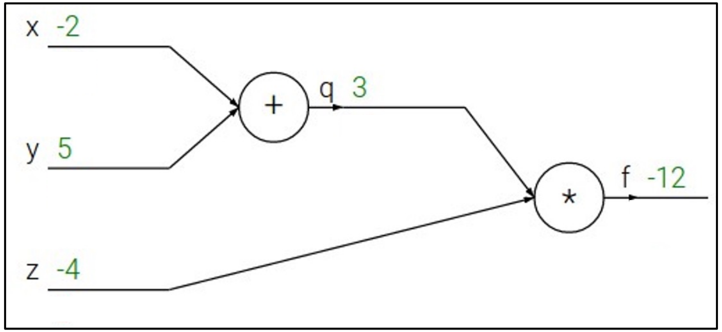
$$f(x, y, z) = (x + y)z$$



Backpropagation: a simple example

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

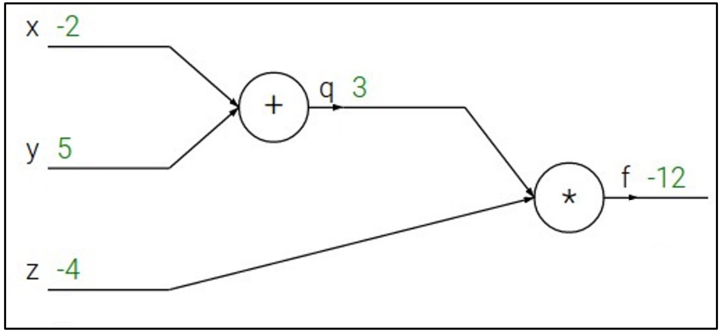
e.g. x = -2, y = 5, z = -4



Backpropagation: a simple example

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

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



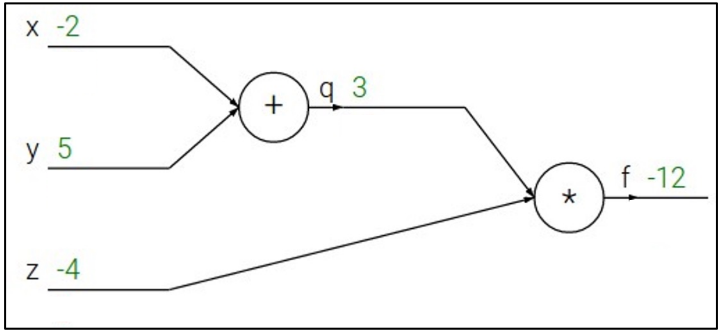
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Backpropagation: a simple example

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

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

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



1. Calculate local gradients

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

Backpropagation: a simple example

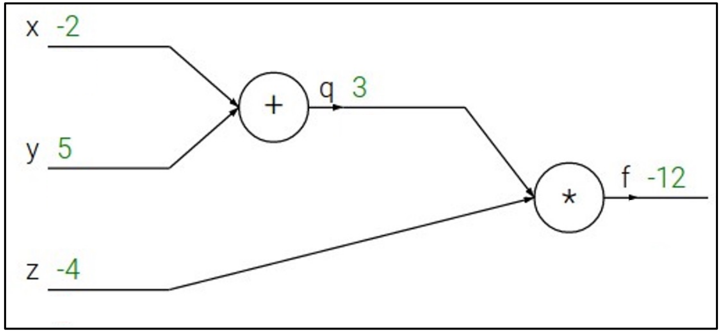
$$f(x, y, z) = (x + y)z$$

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

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

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

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



1. Calculate local gradients

Backpropagation: a simple example

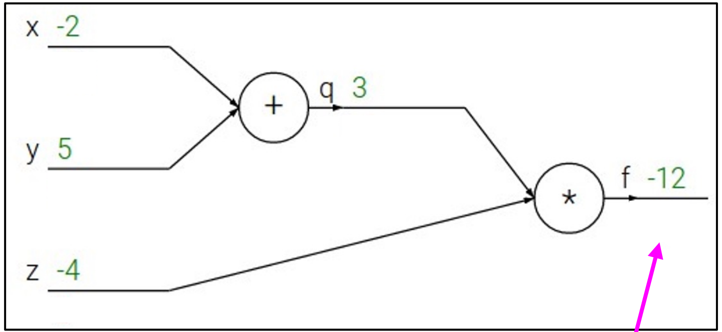
$$f(x, y, z) = (x + y)z$$

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

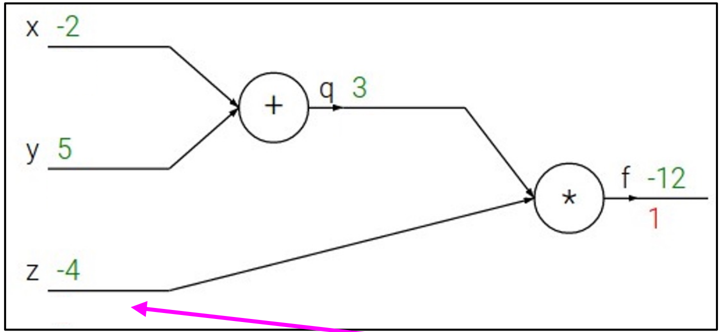
$$f(x,y,z) = (x+y)z$$

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

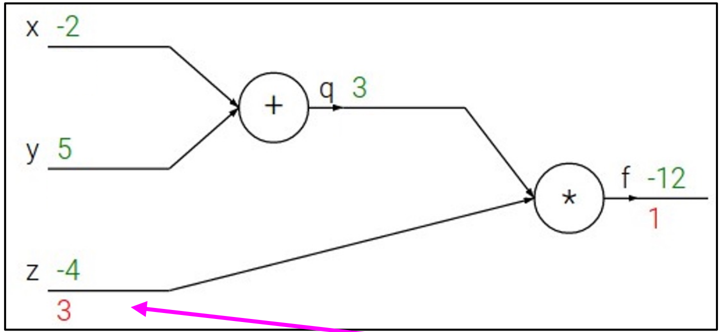
$$f(x, y, z) = (x + y)z$$

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



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

Backpropagation: a simple example

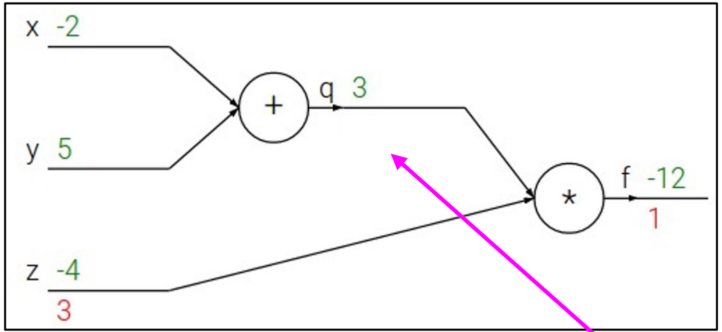
$$f(x, y, z) = (x + y)z$$

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

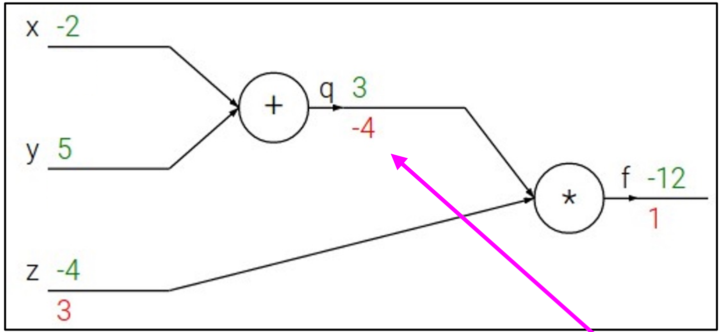
$$f(x, y, z) = (x + y)z$$

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

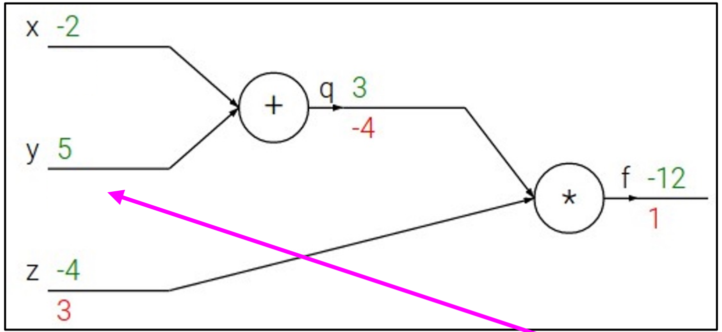
$$f(x, y, z) = (x + y)z$$

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:

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

Upstream gradient

Local gradient

Backpropagation: a simple example

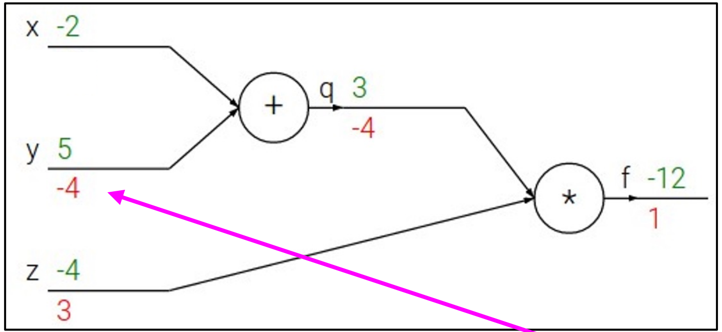
$$f(x, y, z) = (x + y)z$$

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



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

Chain rule:

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

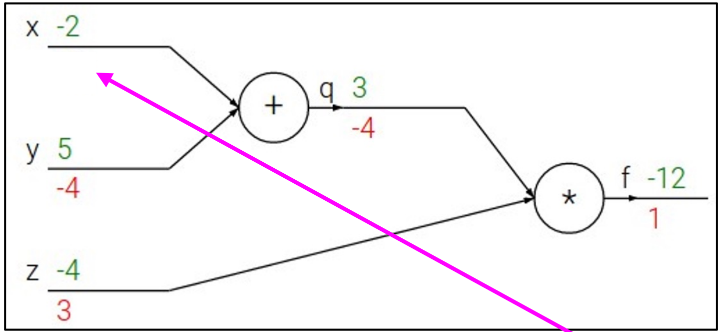
Upstream gradient

Local gradient

Backpropagation: a simple example

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

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



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

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

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

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

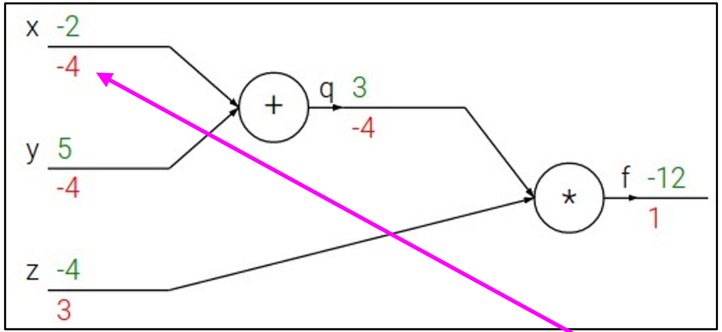
$$f(x, y, z) = (x + y)z$$

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



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

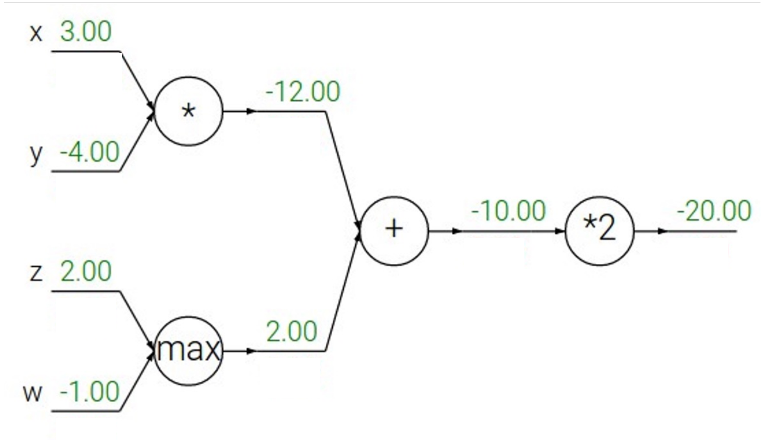
Chain rule:

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

Upstream gradient

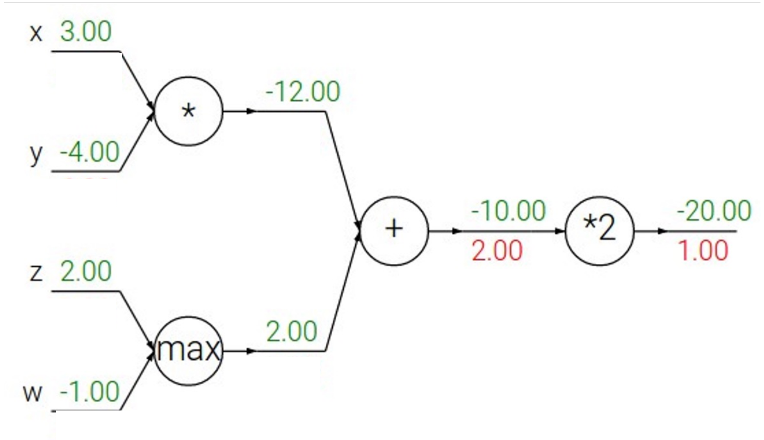
Local gradient

Patterns in backward flow



Patterns in backward flow

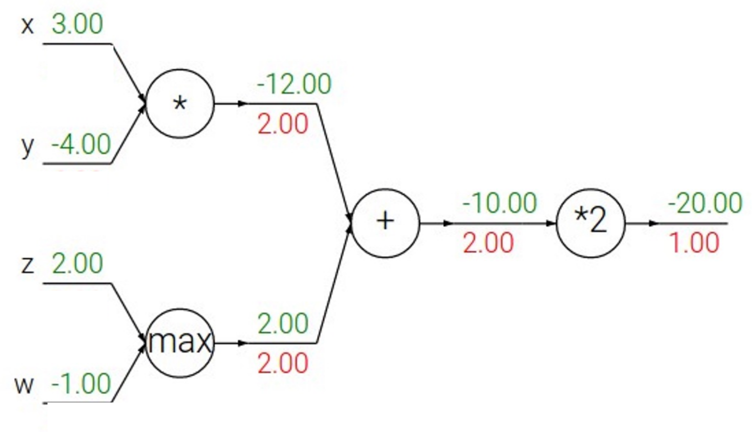
Q: What is an **add** gate?



Patterns in backward flow

add gate: gradient distributor

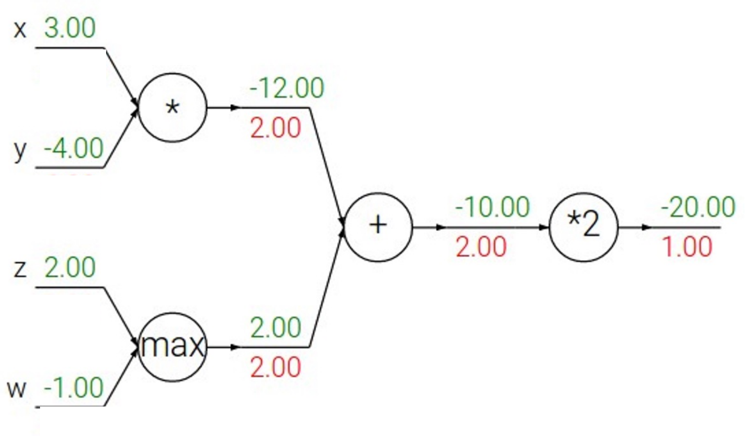
$$f = a + b$$
$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial b} = 1$$



Patterns in backward flow

add gate: gradient distributor

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

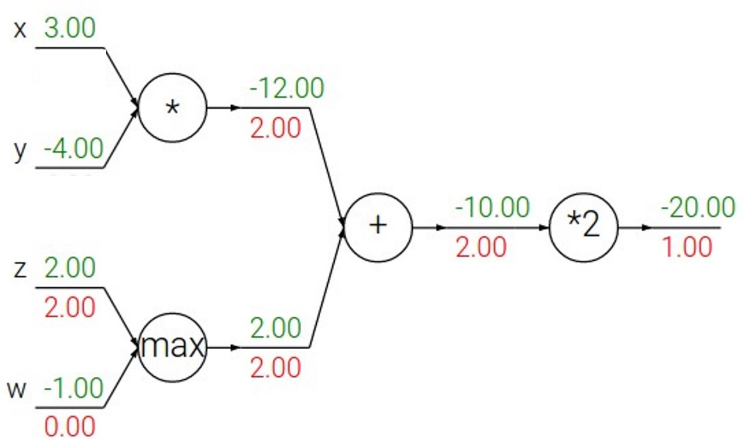


Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

only the path selected by the max operator gets the upstream gradient

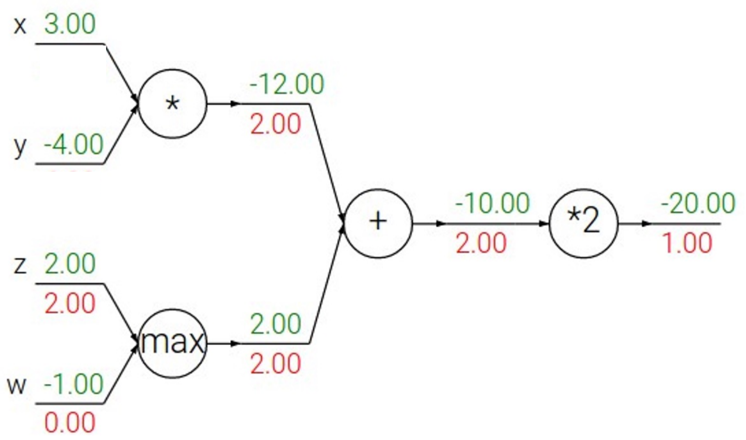


Patterns in backward flow

add gate: gradient distributor

max gate: gradient router

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



Patterns in backward flow

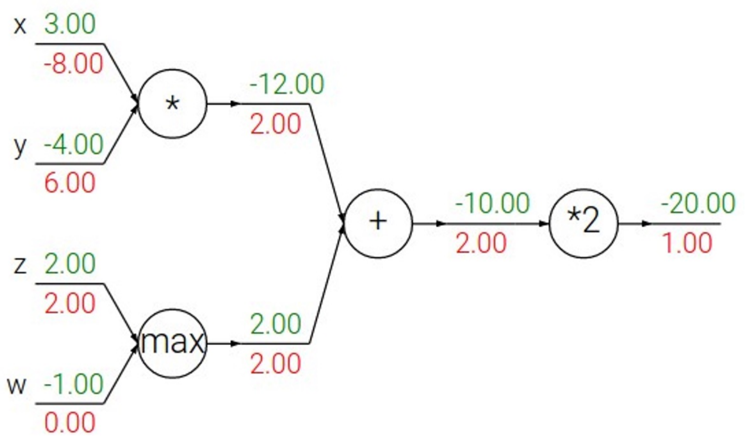
add gate: gradient distributor

max gate: gradient router

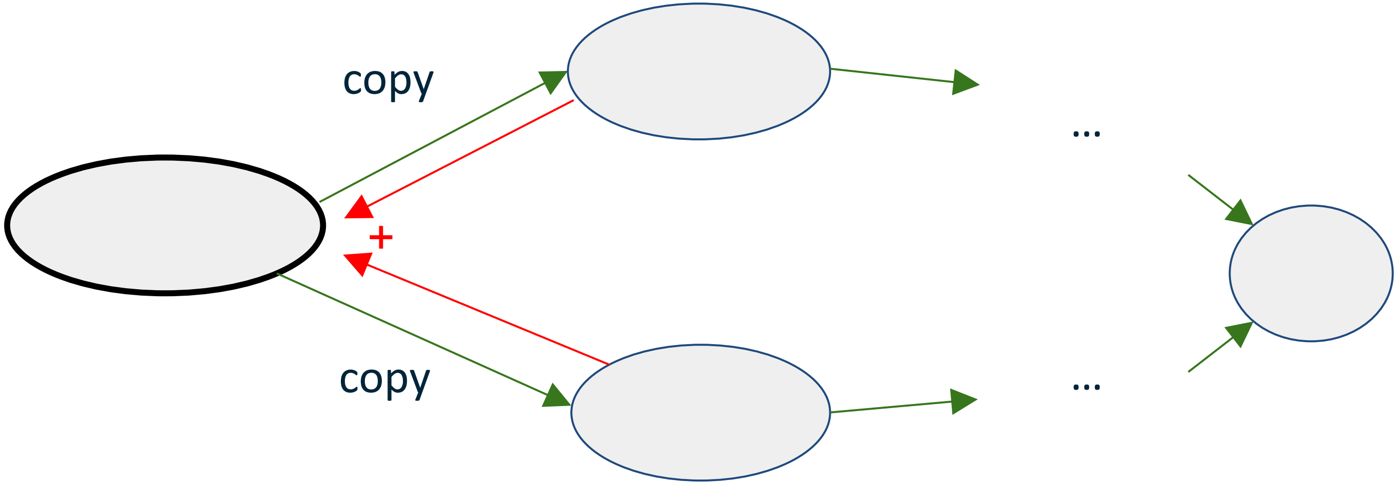
mul gate: gradient switcher

$$f = a \cdot b$$

$$\frac{\partial f}{\partial a} = b \quad \frac{\partial f}{\partial b} = a$$

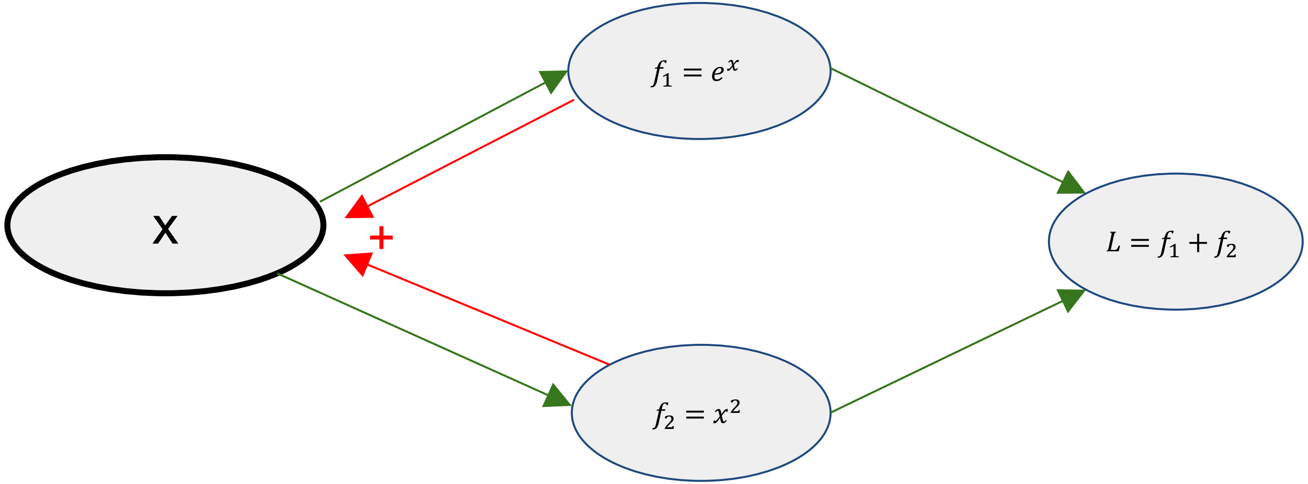


Upstream gradients add at fork branches



… as long as the branches join at some point in the graph

Upstream gradients add at fork branches



Claim: $\frac{\partial L}{\partial q} = \frac{\partial L}{\partial f_1}\frac{\partial f_1}{\partial q} + \frac{\partial L}{\partial f_2}\frac{\partial f_2}{\partial q}$

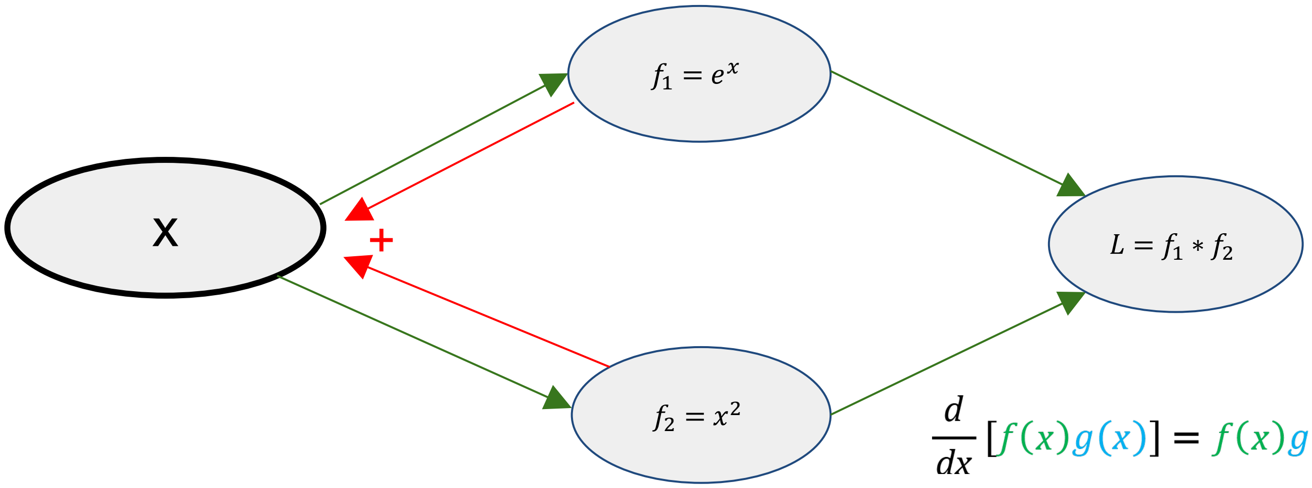
$= 1 \cdot e^x + 1 \cdot 2x$

$= e^x + 2x$

Derivation: $L = e^x + x^2$

$$\frac{\partial L}{\partial q} = e^x + 2x$$

Upstream gradients add at fork branches

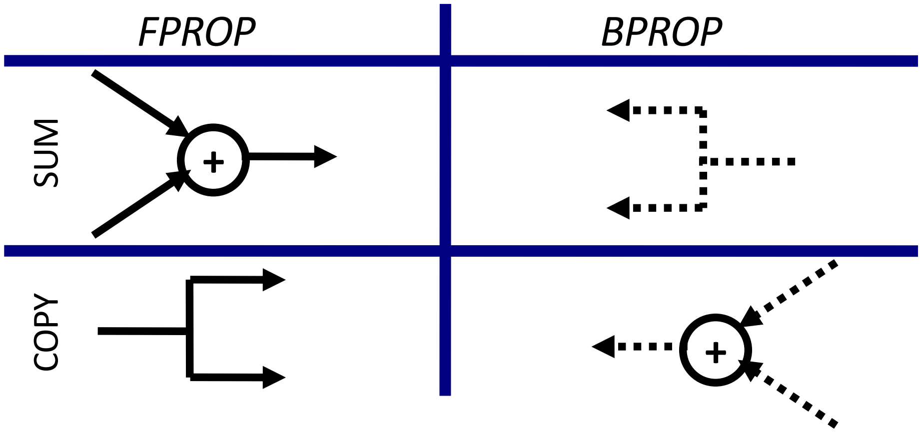


$$\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + f'(x)g(x)$$

Claim: $\frac{\partial L}{\partial q} = \frac{\partial L}{\partial f_1}\frac{\partial f_1}{\partial q} + \frac{\partial L}{\partial f_2}\frac{\partial f_2}{\partial q}$
$= x^2 \cdot e^x + e^x \cdot 2x$

Derivation: $L = e^x * x^2$
$$\frac{\partial L}{\partial q} = e^x \cdot 2x + e^x \cdot x^2 = x^2 \cdot e^x + e^x \cdot 2x$$

Duality in F(orward)prop and B(ack)prop



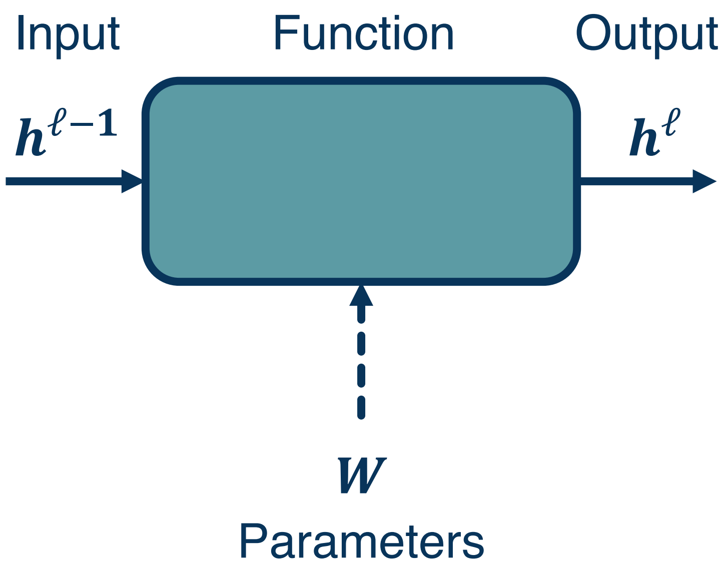
Given this computation graph, the training algorithm will:

- 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:

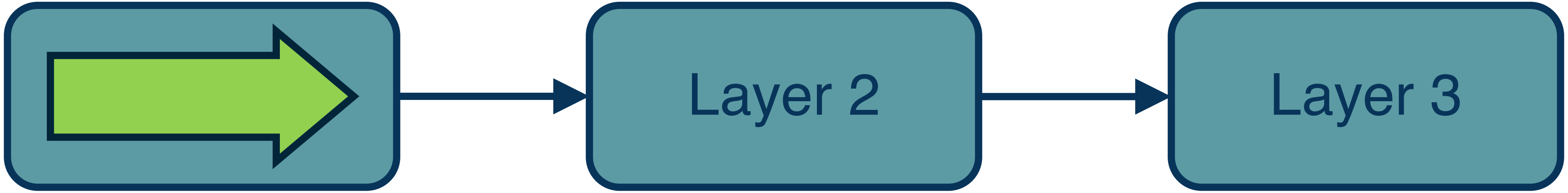
- Starts at **loss function** where we know how to calculate the gradients
- Progresses back through the modules
- Ends in the **input layer** where we do not need gradients (no parameters)

This algorithm is called **backpropagation**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

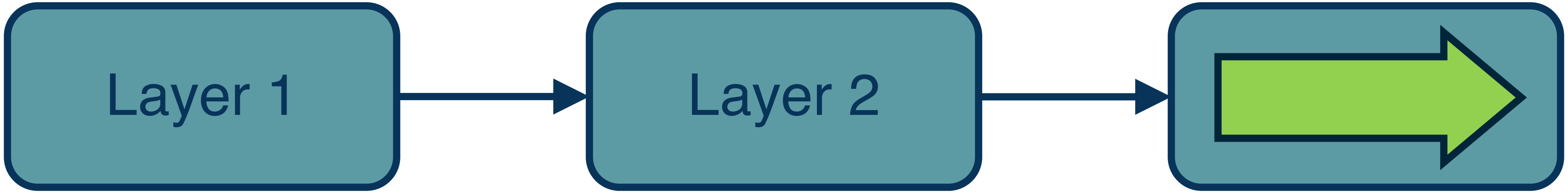


Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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**

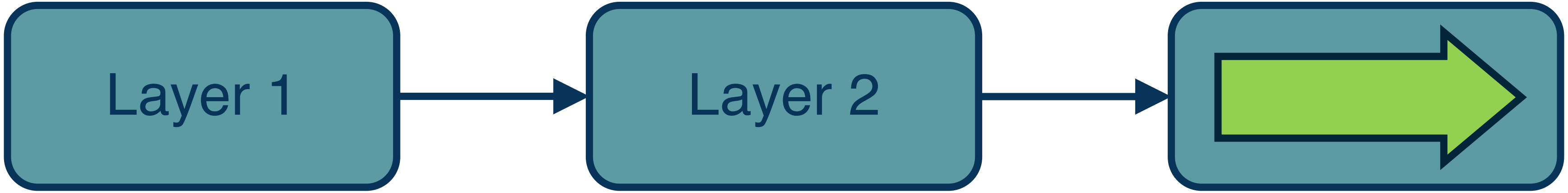


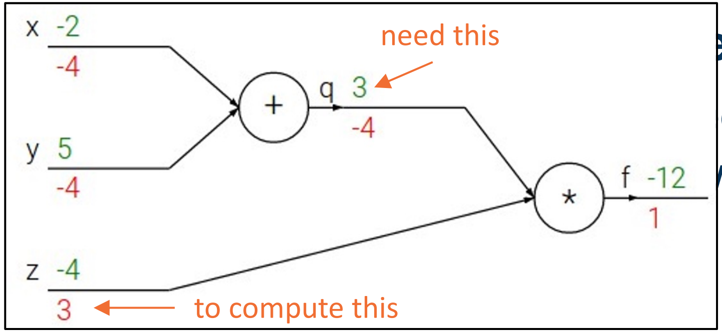
Note that we must store the **intermediate outputs of all layers**!

- 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**





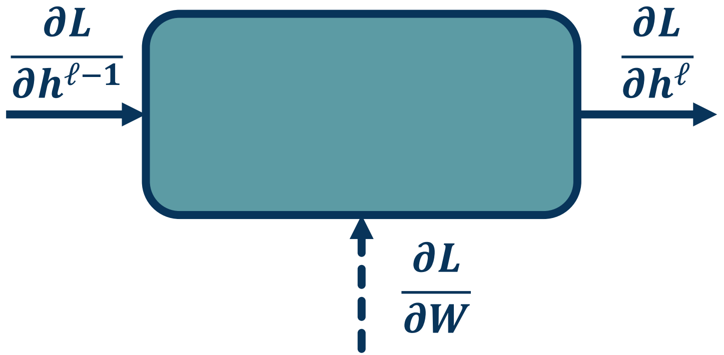
ermediate outputs of all layers!

d them to **compute the gradients** (the gradient
ith 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
  - Becomes the **upstream gradient** for the previous module



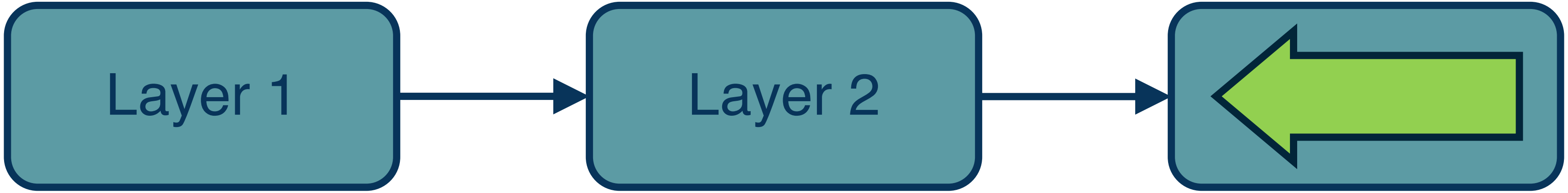
Problem:

- 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

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

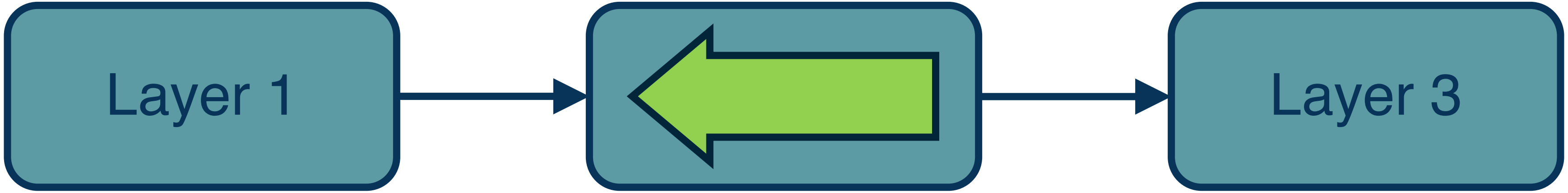
Step 2: Compute Gradients wrt parameters: **Backward Pass**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

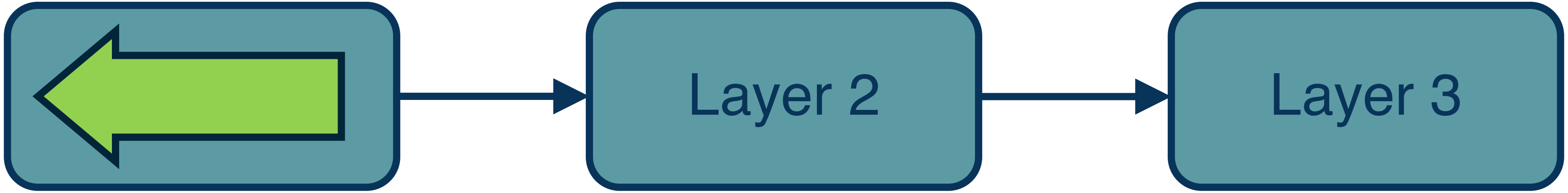
Step 2: Compute Gradients wrt parameters: **Backward Pass**



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**

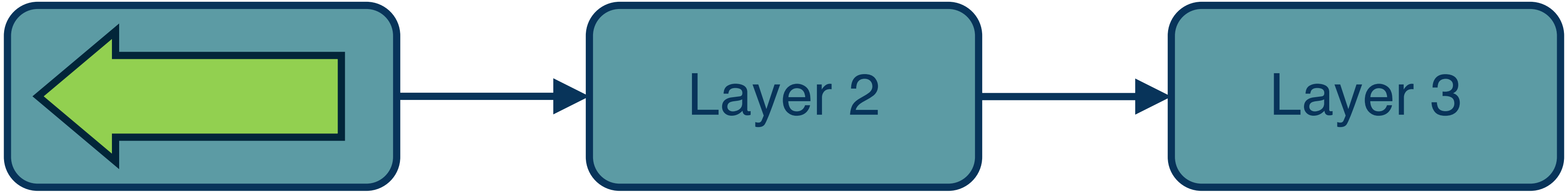


Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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

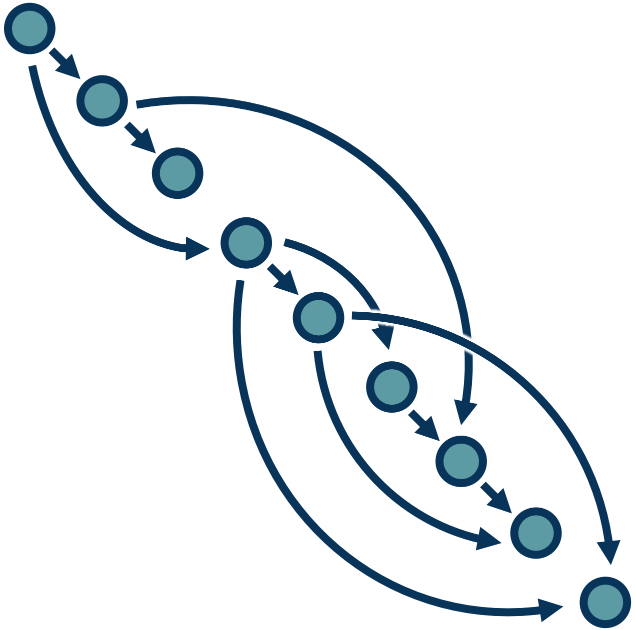
But the idea can be applied to **any directed acyclic graph (DAG)**

- Graph represents an **ordering constraining** which paths must be calculated first

Given an ordering, we can then iterate from the last module backwards, **applying the chain rule**

- We will store, for each node, its **gradient outputs for efficient computation**
- We will do this **automatically** by tracing the entire graph, aggregate and assign gradients at each function / parameters, from output to input.

This is called reverse-mode **automatic differentiation**



A General Framework

Computation = Graph

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

Auto-Diff

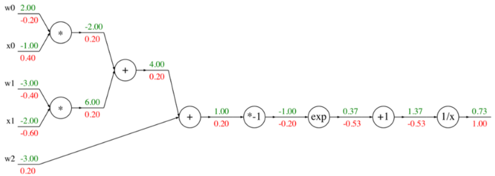
- A family of algorithms for
  implementing chain-rule on computation graphs

Deep Learning Framework = Differentiable Programming Engine

- 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)

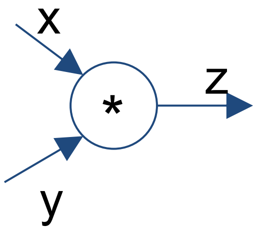
Modularized implementation: forward / backward API



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



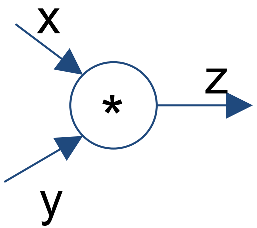
(x,y,z are scalars)

```python
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]
```

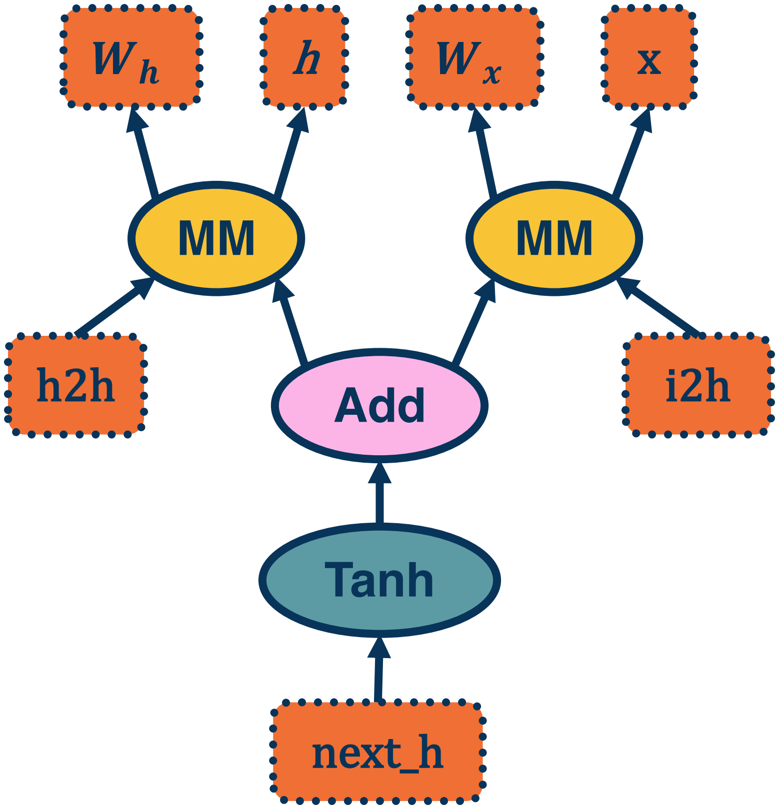
Writing code == building graph

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



From pytorch.org

Neural Turing Machine

input image

loss

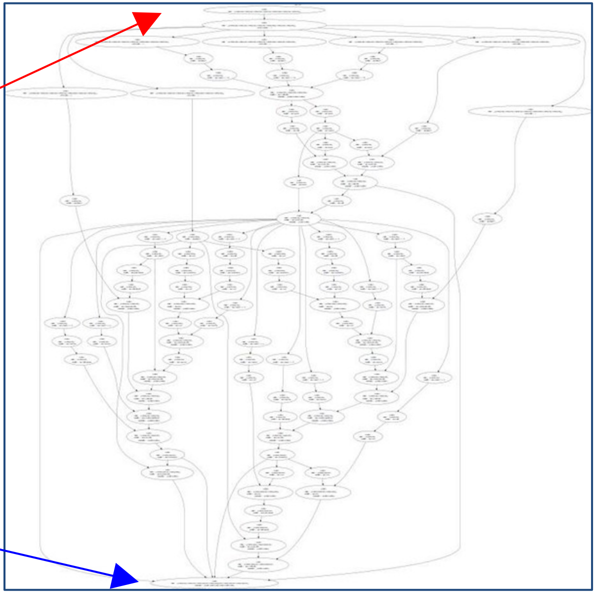
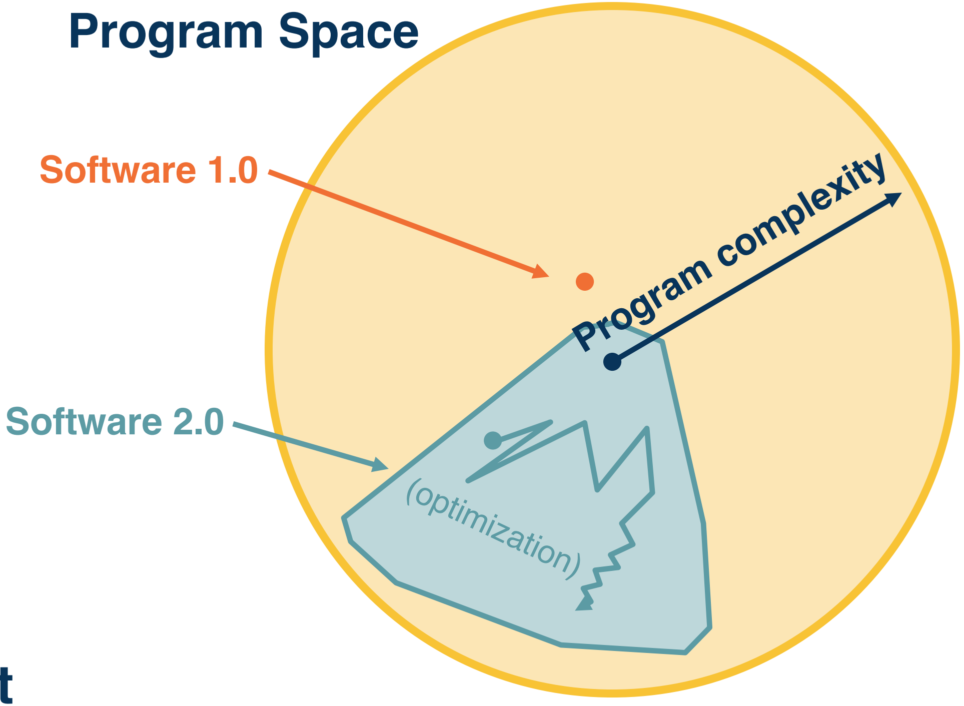


Figure reproduced with permission from a Twitter post by Andrej Karpathy.

- Computation graphs are **not limited to mathematical functions!**
- Can have **control flows** (if statements, loops) and **backpropagate** through **algorithms**!
- Can be done **dynamically** so that **gradients are computed**, then **nodes are added, repeat**



Adapted from figure by Andrej Karpathy

- Autodiff from scratch: micrograd repo, video tutorial

Linear Algebra View: Vector and Matrix Sizes

Georgia Tech

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

W x

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

Where c is number of classes

d 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 $\boldsymbol{M} \in \mathbb{R}^{k \times \ell}$

- 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 $\boldsymbol{m}$)

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

Conventions:

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

$$\begin{array}{cc} & \begin{array}{cccccc} & & \textbf{Col } j & & & \end{array} \\ \textbf{Row } i & \begin{bmatrix} \frac{\partial v_1^1}{\partial v_1^2} & \cdots & \cdots & \cdots & \cdots & \\ \cdots & & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \frac{\partial v_i^1}{\partial v_j^2} & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \end{bmatrix} \end{array}$$

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

(Note this is slightly different convention than on Wikipedia)

Conventions:

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

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

- 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:

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

Dimensionality of Derivatives

Jacobians of Batches

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

Examples:

- Each instance is a vector of size $\boldsymbol{m}$, 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

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