

CS 4644-DL / 7643-A

DANFEI XU

Topics:

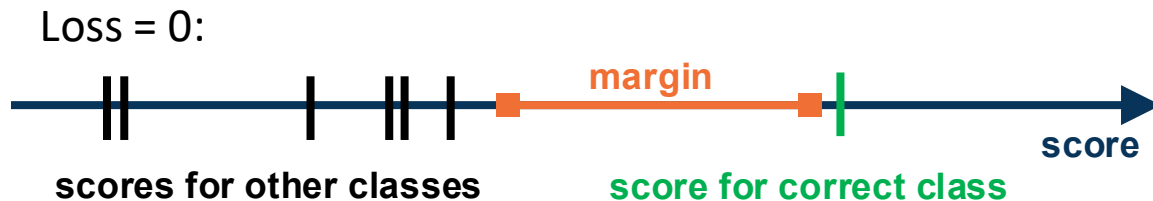
- Backpropagation
- Computation Graph and Automatic Differentiation

Administrative

- PS1 / HW1 is out. Due by Sep 21th 11:59pm (+48hr late days)
- Use Piazza for Q&A
- **Start early!**
- How to pick a project lecture (Sep 14)
- Project helping session (Sep 21)
- Project proposal due Oct 7 (no grace period)

Recap: Multiclass SVM loss

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

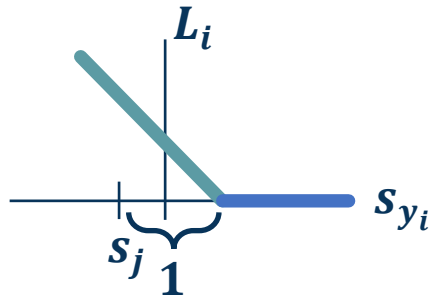


and using the shorthand for the
scores vector: $s = f(x_i, W)$

the SVM loss has the form:

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

“Hinge Loss”

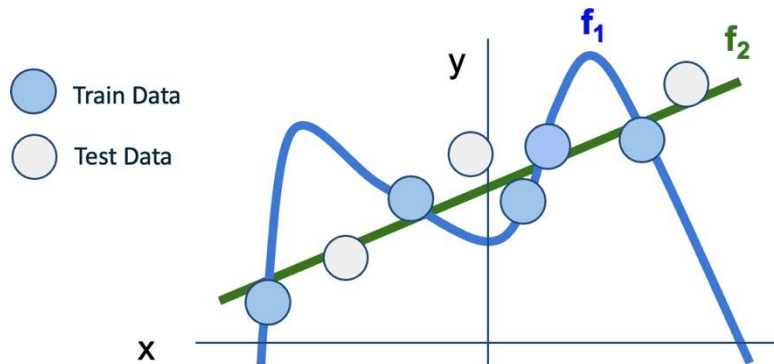


Recap: Regularization

Q: How do we pick between W and $2W$?

A: Opt for simpler functions to avoid overfit

How? Regularization!



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

λ = regularization strength
(hyperparameter)

Data loss: Model predictions should match training data

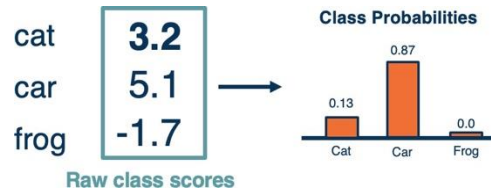
Regularization: Prevent the model from doing *too* well on training data

Recap: Softmax Classifier and Cross Entropy Loss

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

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

**Softmax
Function**



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

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: $\operatorname{argmax}_{\theta} \prod p_{\theta}(y_i | x_i)$

This is equivalent to:

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

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

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

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

Q: Why softmax?



Why this?



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

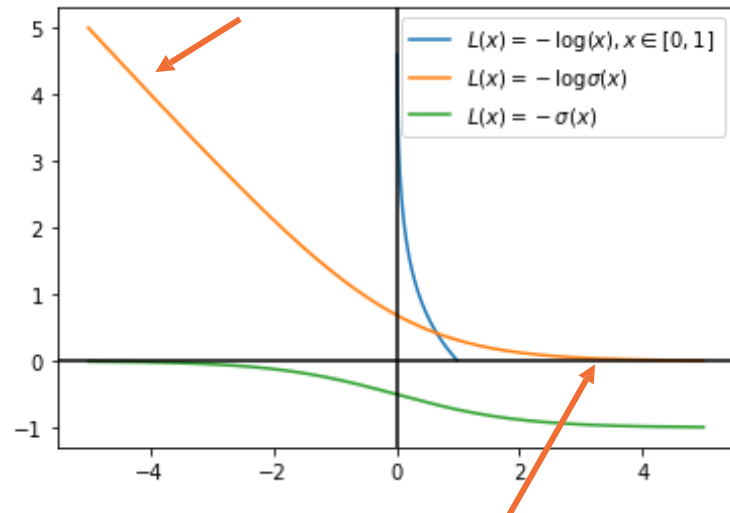
Use logistic function as example. Same as softmax but for binary classification

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

Consider the following three basis for NLL:

1. Squash and clip network value (x) to $(0, 1]$
2. (Negative) logistic function
3. NLL with logistic function

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

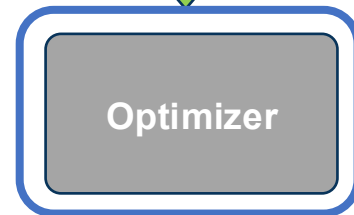
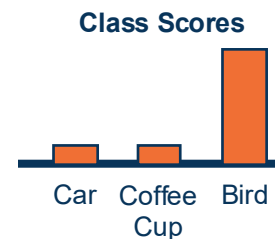
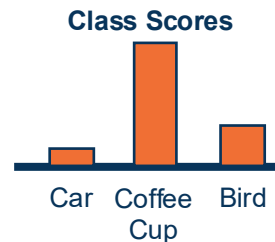
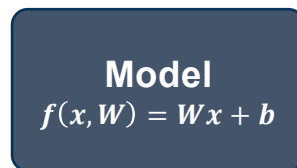


Only saturate at convergence, e.g. $\sigma(3) \approx 0.95$

- 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



Data: Image

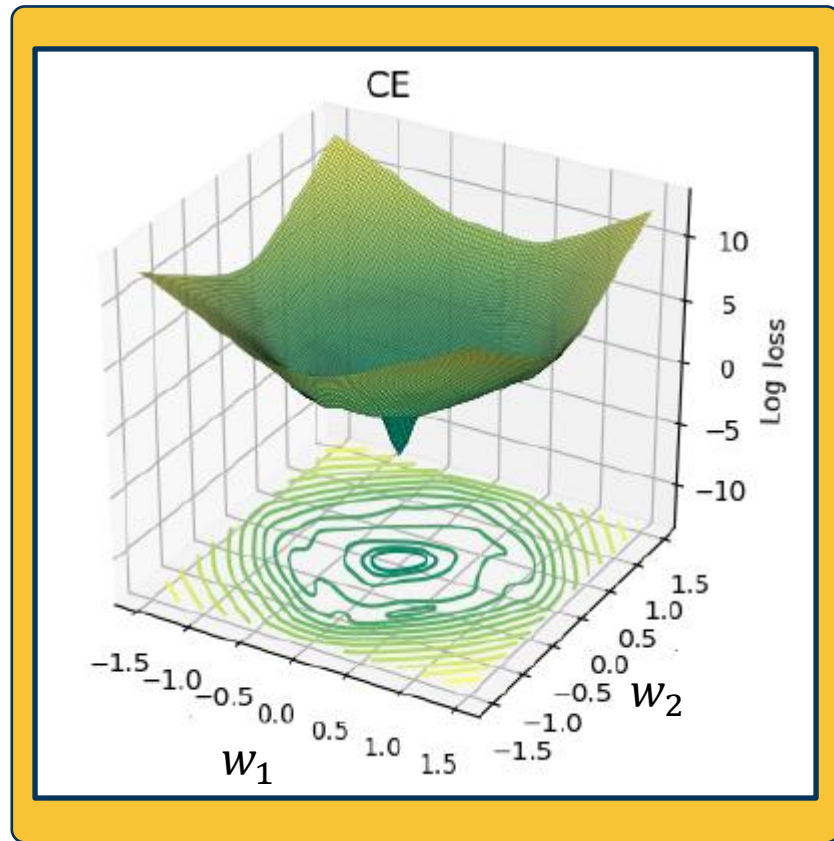


Gradient-based Optimization

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**



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

$$\frac{\partial f}{\partial w} = \lim_{h \rightarrow 0} \frac{f(w + h) - f(w)}{h}$$

- **Gradient** is multi-dimensional derivatives
- Notation: $\frac{\partial f}{\partial w}$ is the gradient of f (e.g., a loss function) with respect to variable w (e.g., a weight vector).
- $\frac{\partial f}{\partial w}$ is of the **same shape** as w
- **Intuitively**: Measures how the function changes as the variable w changes by a small step size
- Steepest descent direction is the **negative gradient**
- **Gradient descent**: Minimize loss by changing parameters

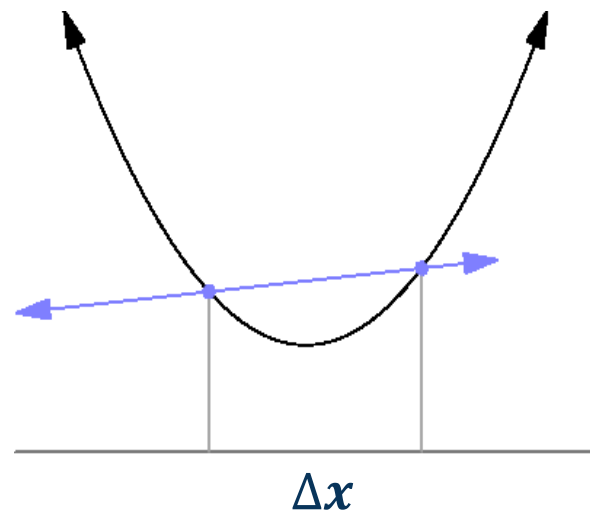
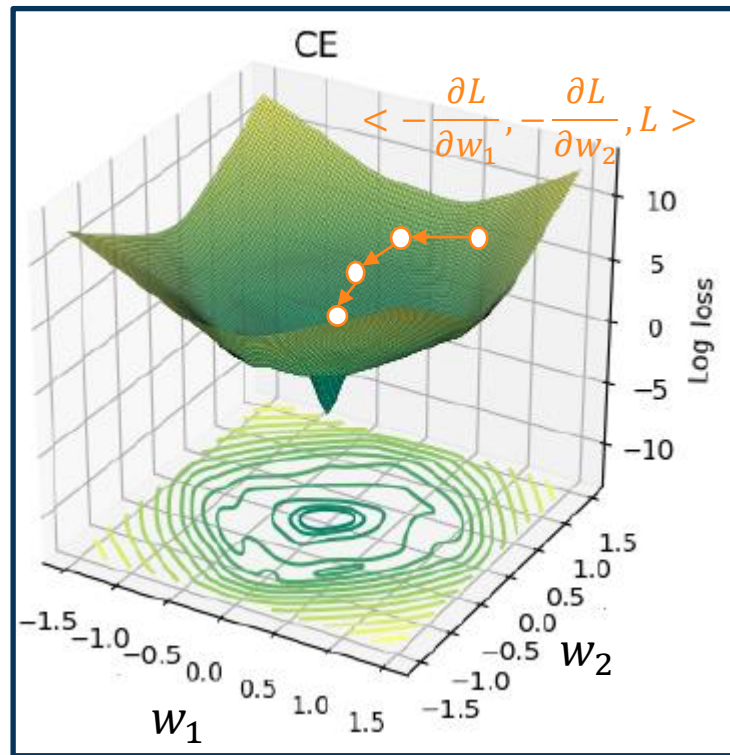


Image and equation from:
https://en.wikipedia.org/wiki/Derivative#/media/File:Tangent_animation.gif

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

$$\frac{\partial f}{\partial w} = \lim_{h \rightarrow 0} \frac{f(w + h) - f(w)}{h}$$

- **Gradient** is multi-dimensional derivatives
- Notation: $\frac{\partial f}{\partial w}$ is the gradient of f (e.g., a loss function) with respect to variable w (e.g., a weight vector).
- $\frac{\partial f}{\partial w}$ is of the **same shape** as w
- **Intuitively**: Measures how the function changes as the variable w changes by a small step size
- Steepest descent direction is the **negative gradient**
- **Gradient descent**: Minimize loss by changing parameters



Calculate gradients: finite differences

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,
?,...]

Calculate gradients: finite differences

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,...]

Calculate gradients: finite differences

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,...]

loss 1.25322

gradient dW:

[-2.5,
?,
?,

$$(1.25322 - 1.25347)/0.0001 = -2.5$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

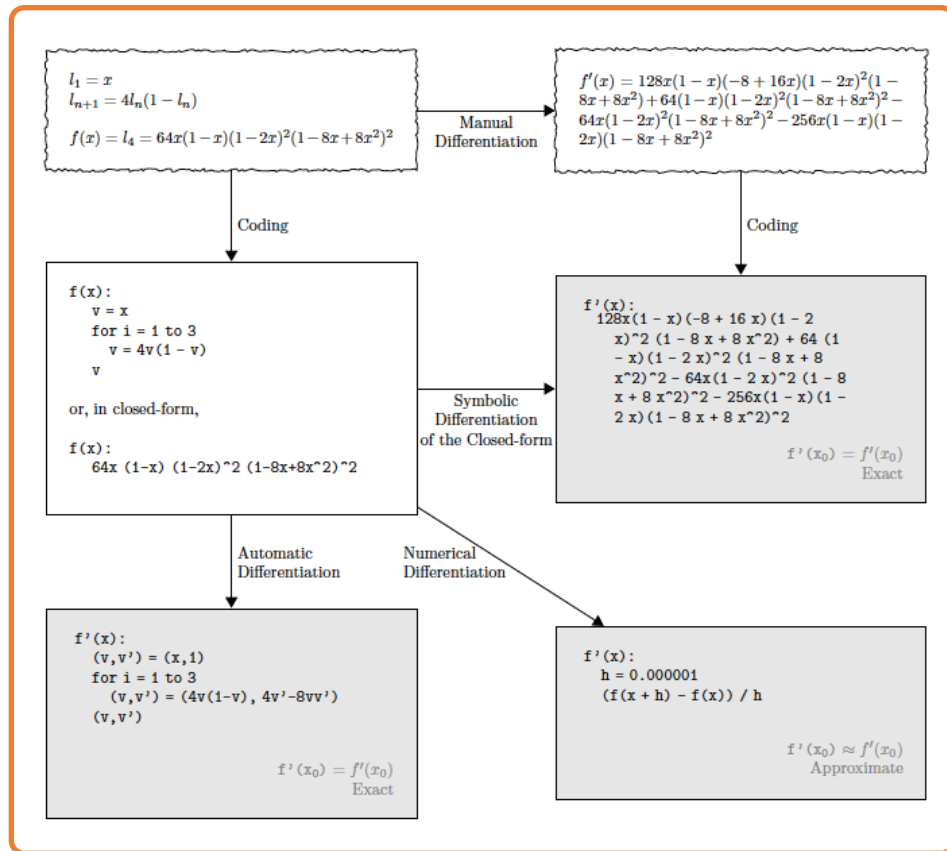
?,
?,...]

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

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

More on **autodiff**:

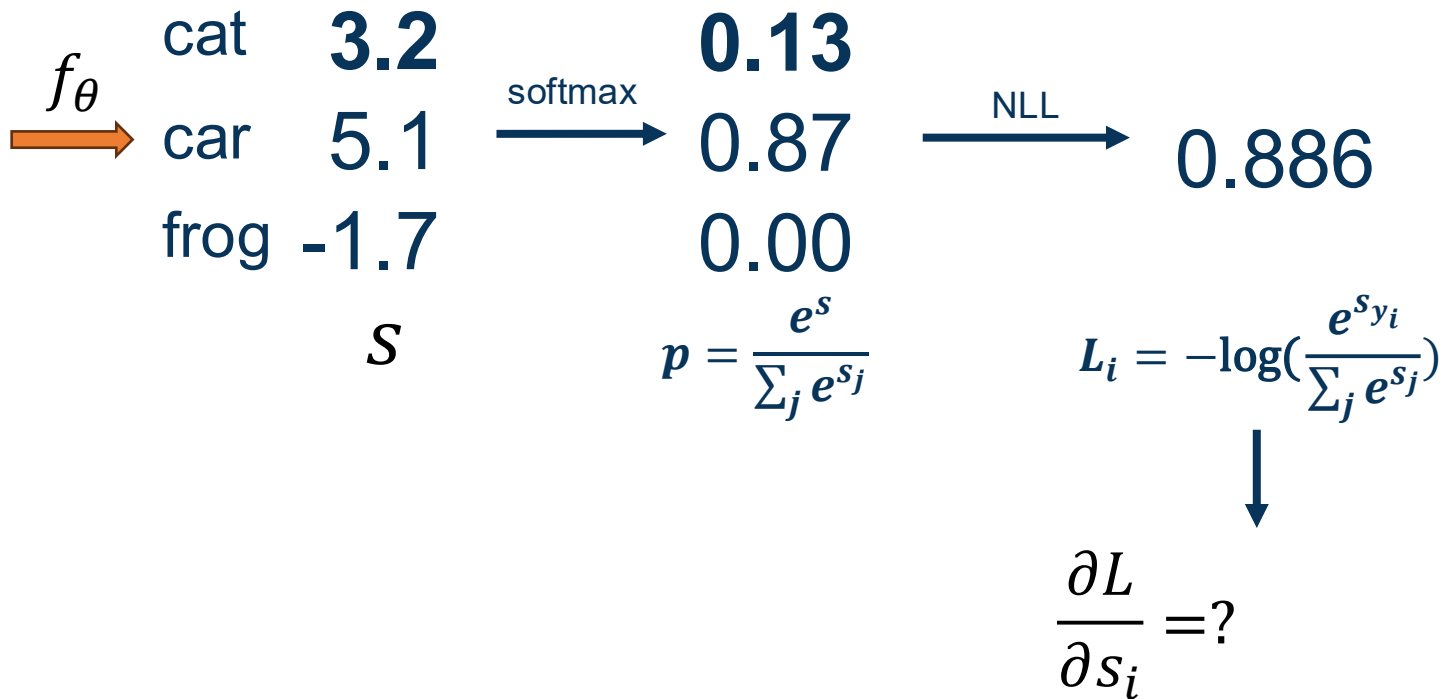
https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/readings/L06%20Automatic%20Differentiation.pdf



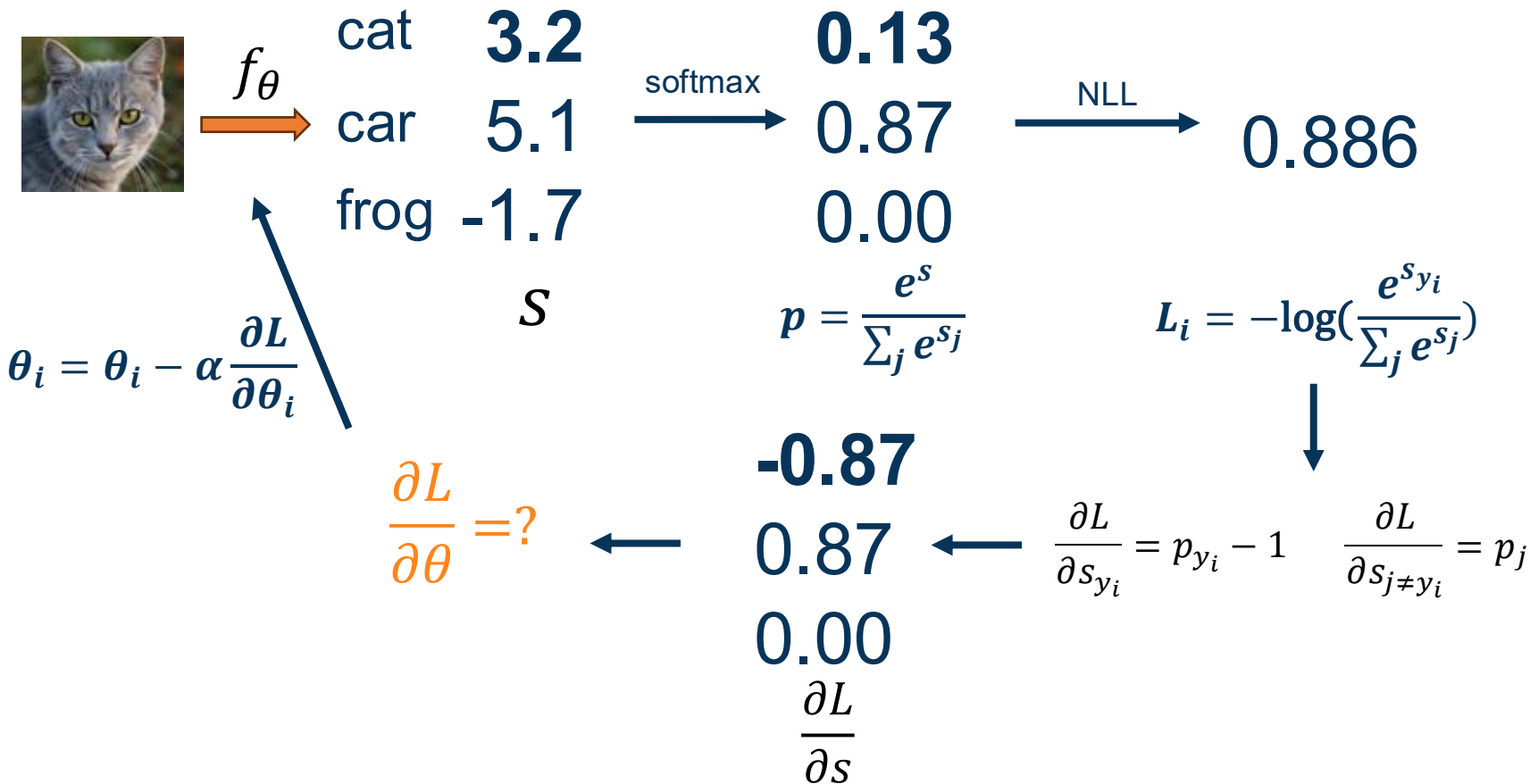
The gradient descent algorithm

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

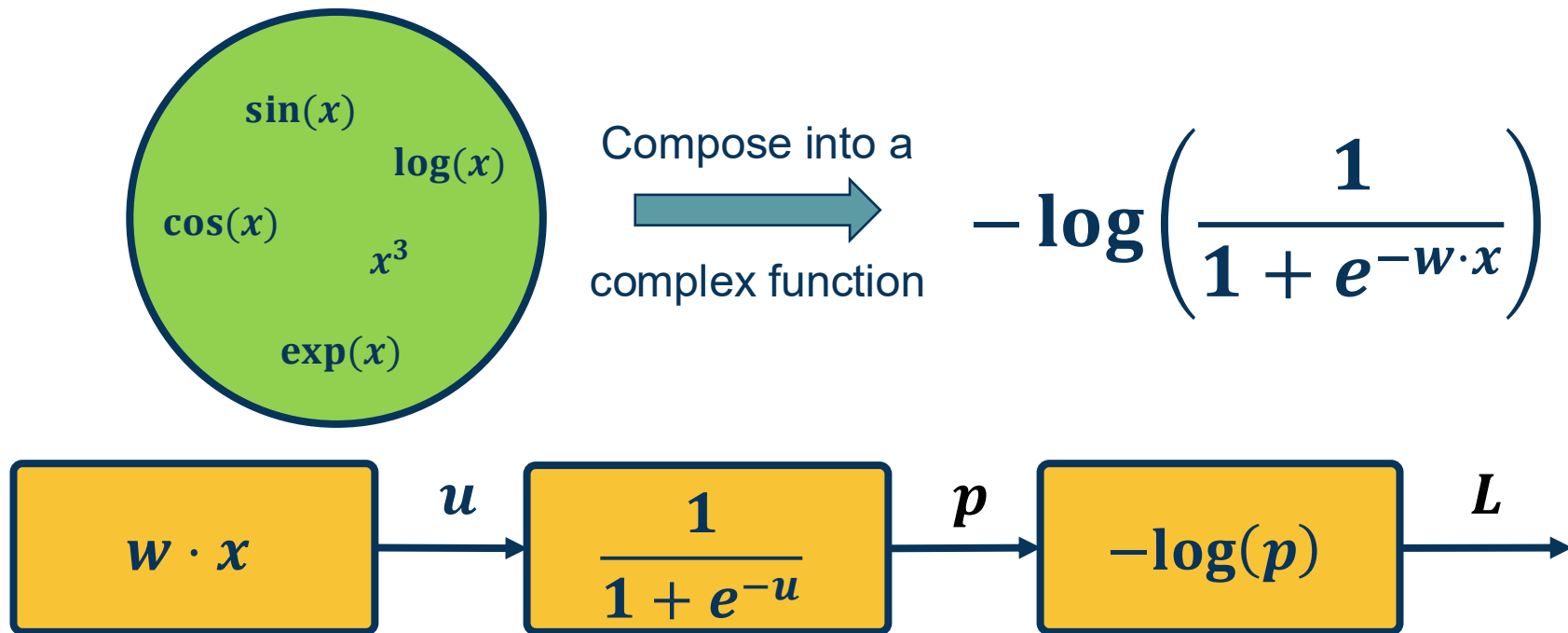
Gradient Descent on Softmax Classifier!



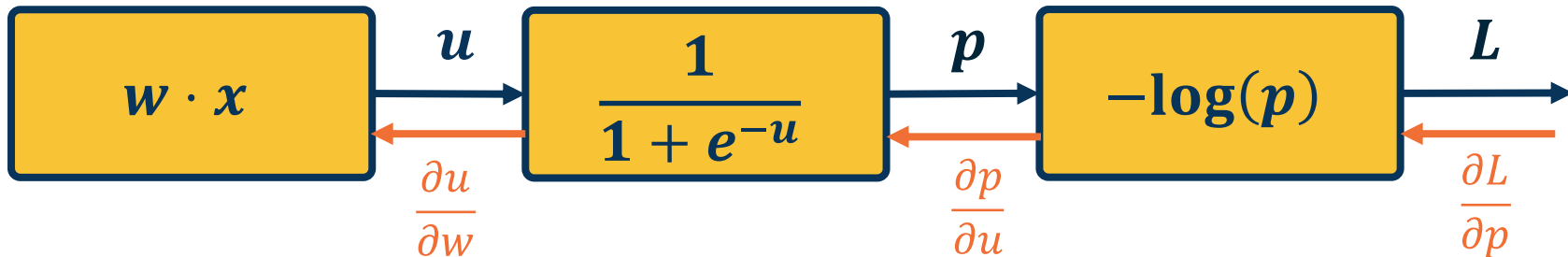
Gradient Descent on Softmax Classifier!



How to compute gradients for (deep) neural networks?



Backpropagation via chain rule



$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial u} \frac{\partial u}{\partial w} \quad \frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial s} \frac{\partial s}{\partial \theta}$$

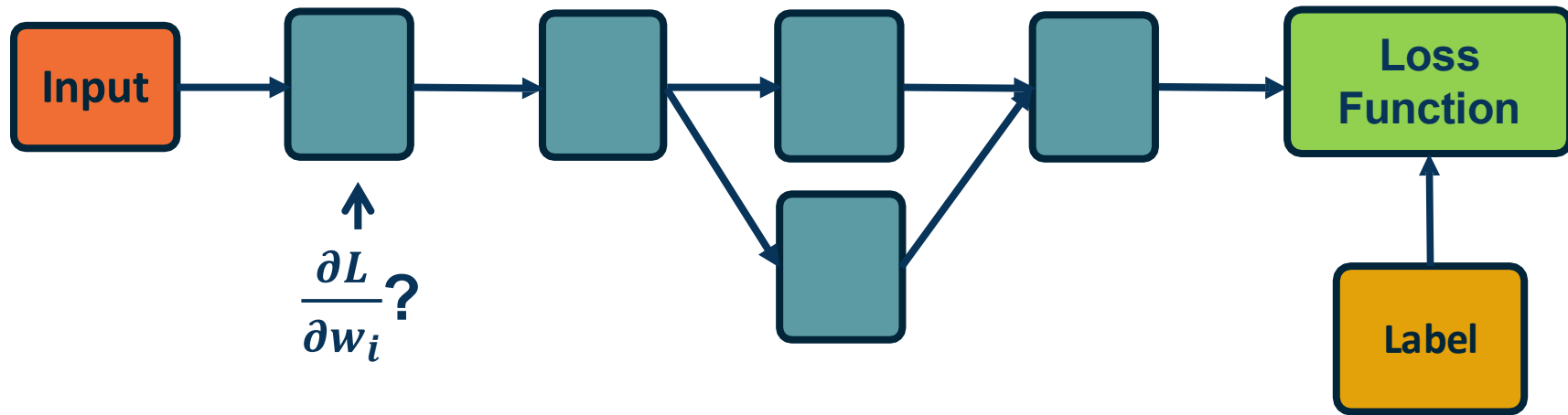
Backpropagation (roughly):

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.

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

$$f(x, W) = \sigma(W_5 \sigma(W_4 \sigma(W_3 \sigma(W_2 \sigma(W_1 x))))$$

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



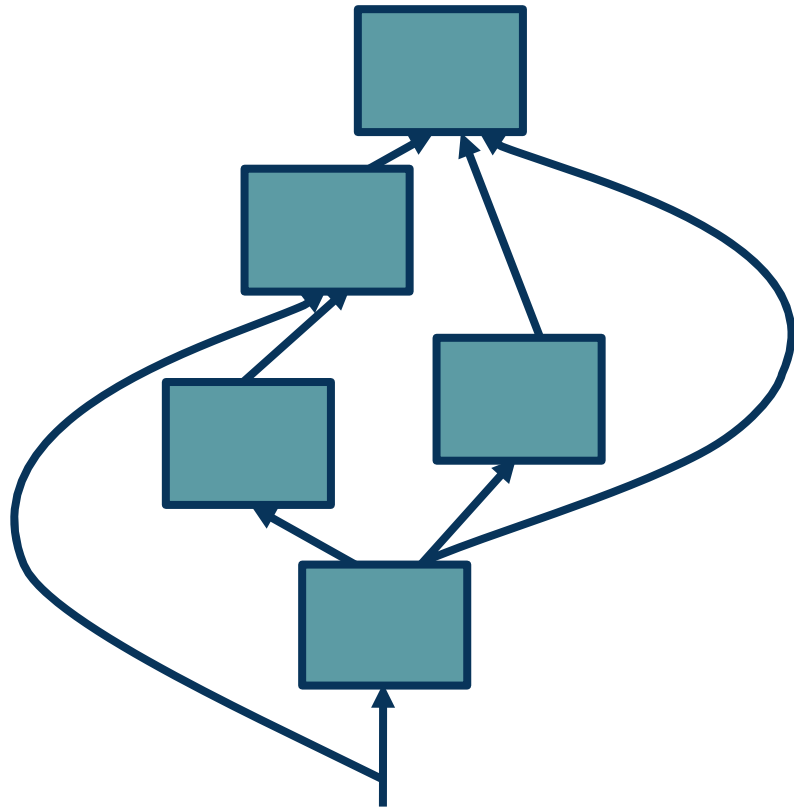
Computational Graph

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 node at a time**



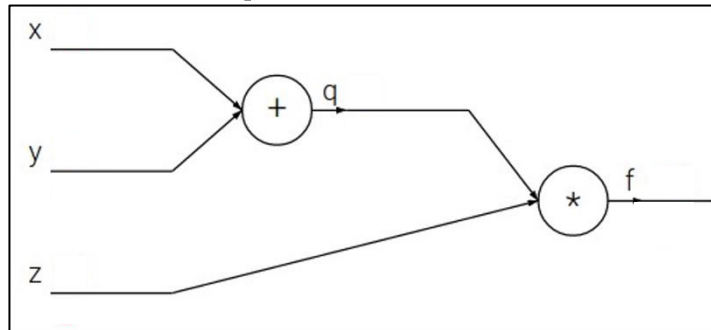
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

Backpropagation: a simple example

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

Backpropagation: a simple example

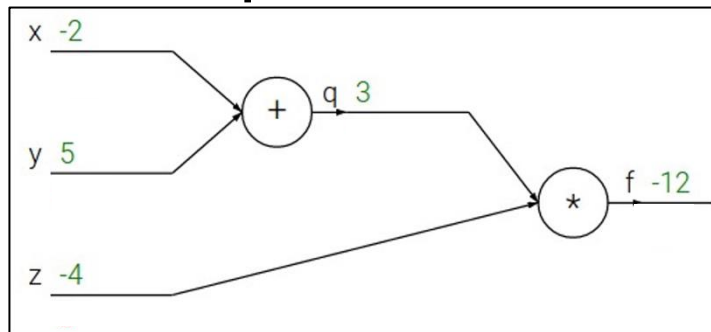
$$f(x, y, z) = (x + y)z$$



Backpropagation: a simple example

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

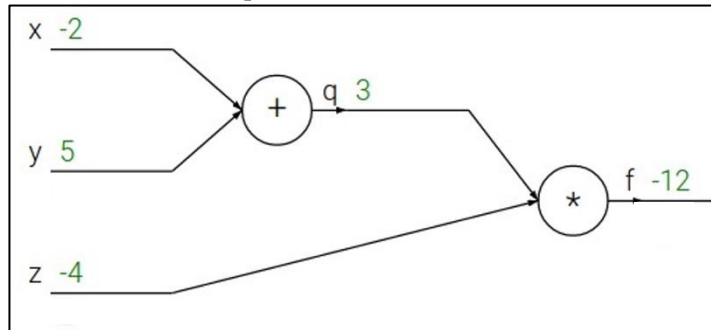
e.g. $x = -2$, $y = 5$, $z = -4$



Backpropagation: a simple example

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

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



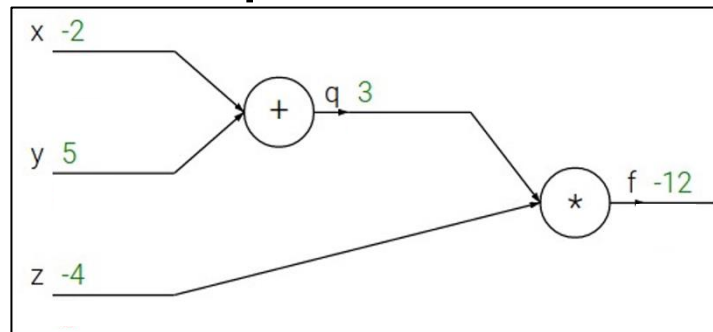
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Backpropagation: a simple example

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

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

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



1. Calculate local gradients

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

Backpropagation: a simple example

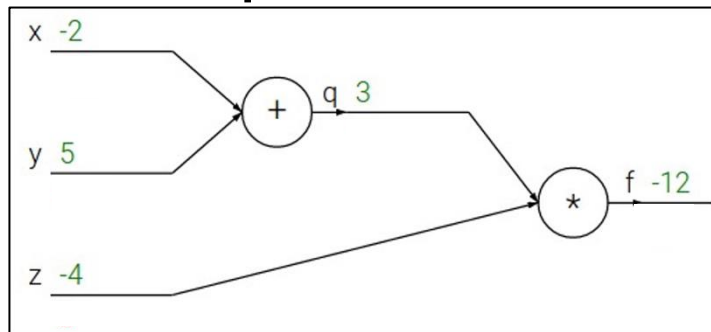
$$f(x, y, z) = (x + y)z$$

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

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

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

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



1. Calculate local gradients

Backpropagation: a simple example

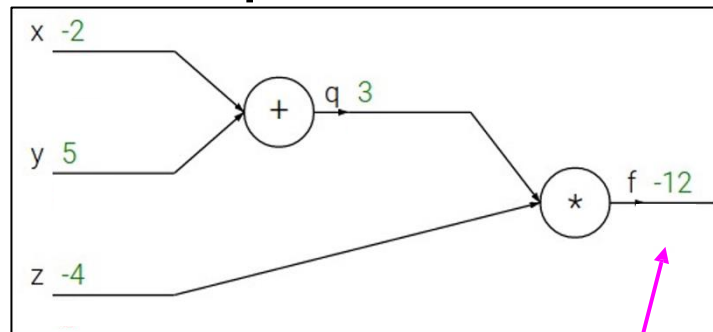
$$f(x, y, z) = (x + y)z$$

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

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

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

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

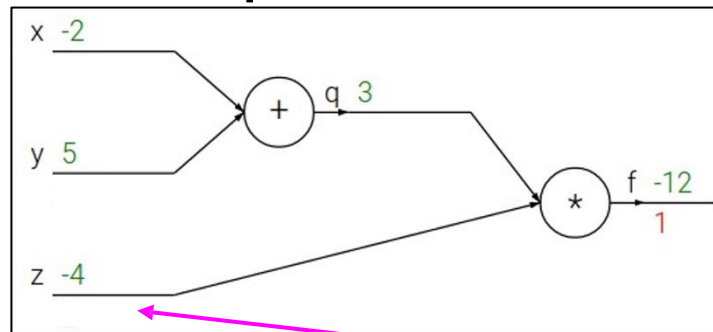
$$f(x, y, z) = (x + y)z$$

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

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

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

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



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

A pink arrow points from this box to the input z of the multiplication node in the computational graph.

Backpropagation: a simple example

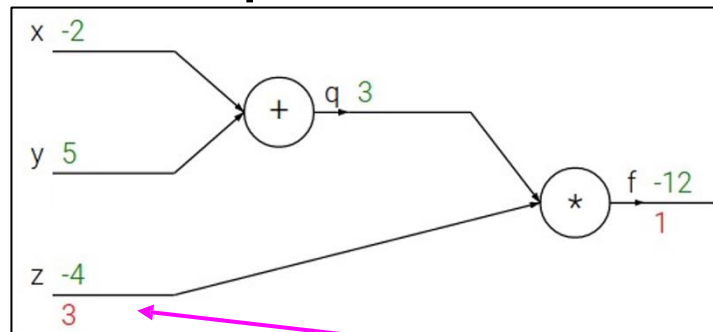
$$f(x, y, z) = (x + y)z$$

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

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

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

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

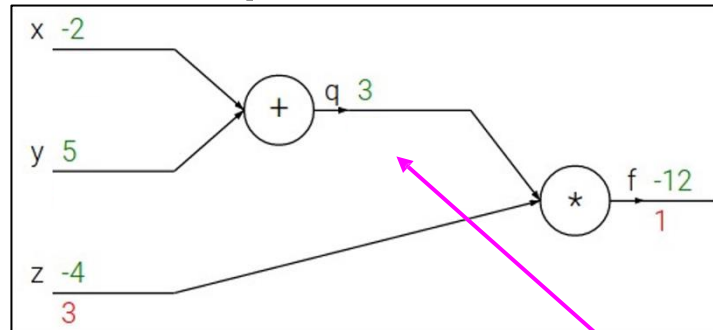
$$f(x, y, z) = (x + y)z$$

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

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

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

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

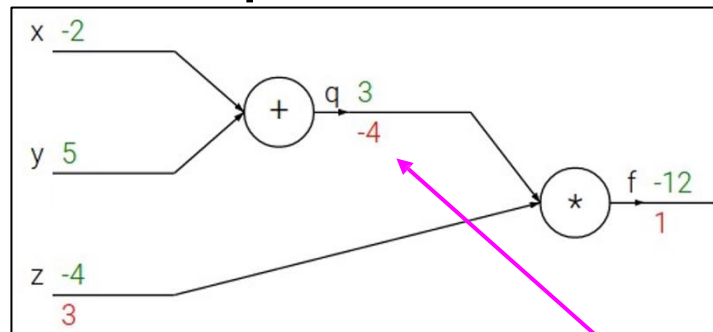
$$f(x, y, z) = (x + y)z$$

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

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

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

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

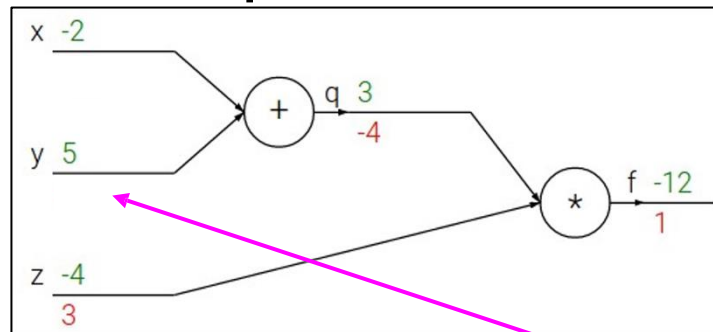
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain rule:

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

Upstream
gradient

Local
gradient

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

Backpropagation: a simple example

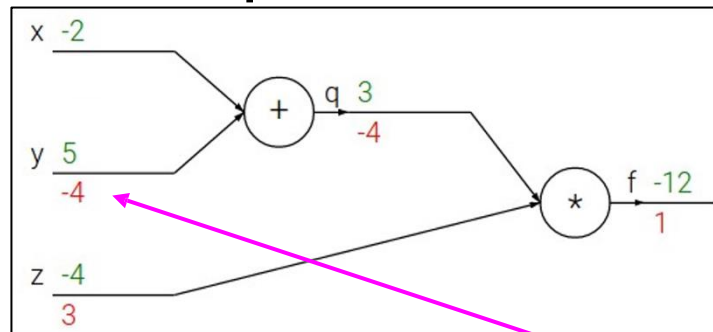
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain rule:

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

Upstream
gradient

Local
gradient

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

Backpropagation: a simple example

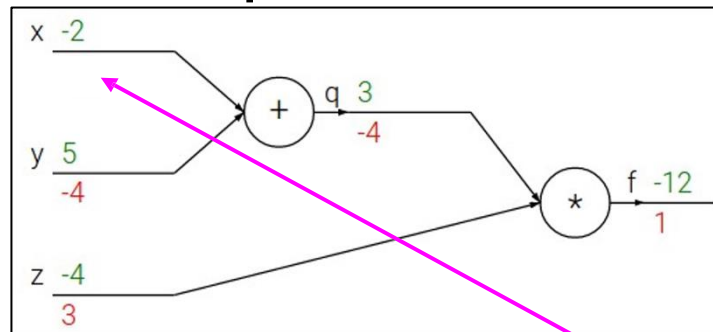
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain rule:

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

Upstream
gradient

Local
gradient

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

Backpropagation: a simple example

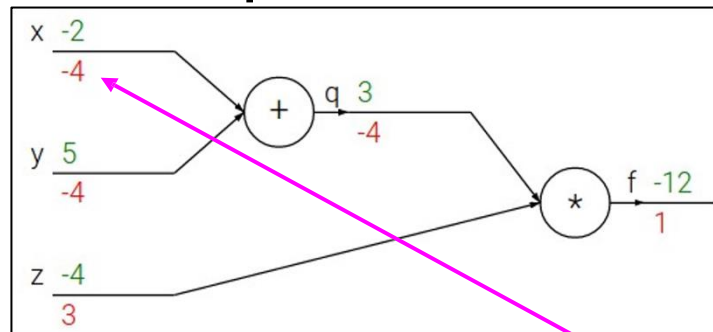
$$f(x, y, z) = (x + y)z$$

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

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

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

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



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

Chain rule:

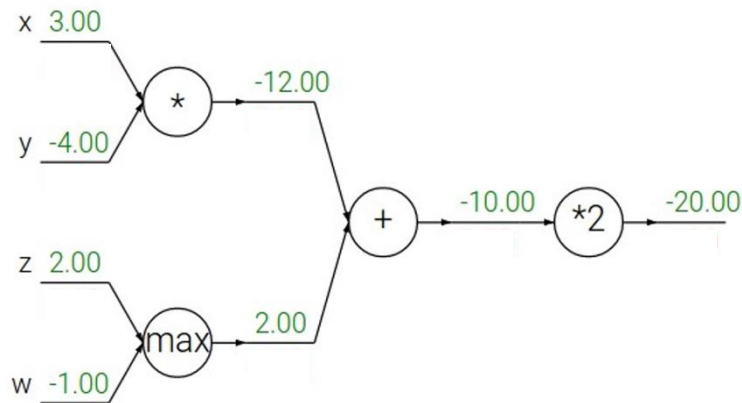
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Upstream
gradient

Local
gradient

Patterns in *backward* flow

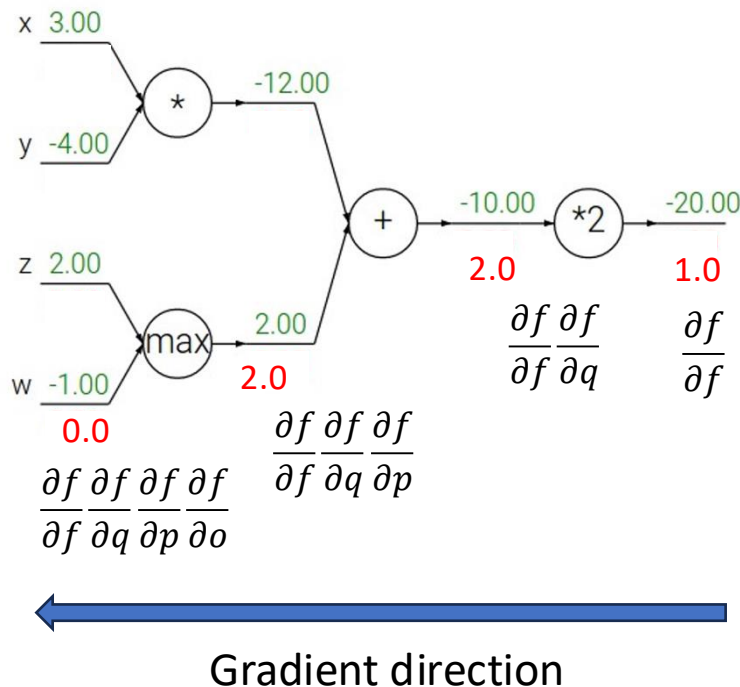
How does a **local gradient** modify the **upstream gradient**? $f = 2(xy + \max(z, w))$



Patterns in *backward* flow

How does a **local gradient** modify the **upstream gradient**? $f = 2(xy + \max(z, w))$

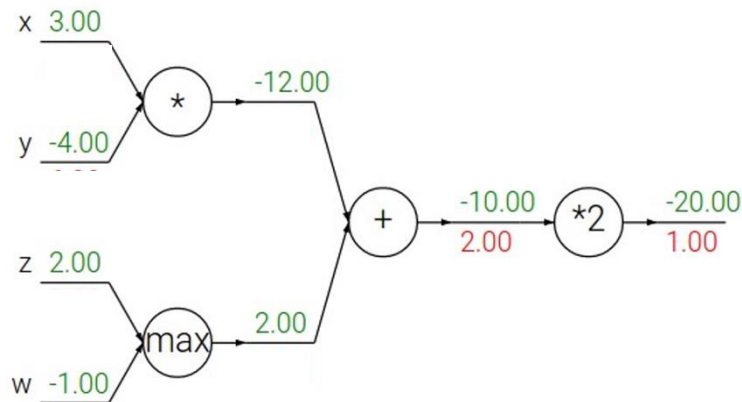
The derivative of the forward flow function (gate) determines how the gradient will be modified during backpropagation (chain rule)



Patterns in backward flow

How does a **local gradient** modify the **upstream gradient**? $f = 2(xy + \max(z, w))$

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



Gradient direction

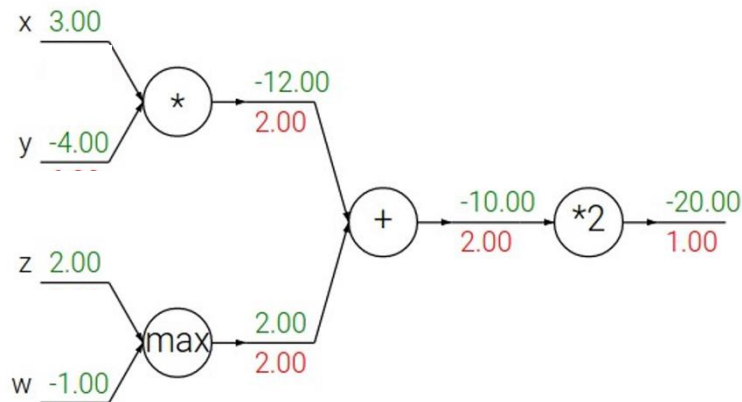
Patterns in backward flow

How does a **local gradient** modify the **upstream gradient**? $f = 2(xy + \max(z, w))$

add gate: gradient replicator

$$S = T + U$$

$$\frac{\text{NS}}{\text{NT}} = \frac{\text{NS}}{\text{NU}} = 1$$



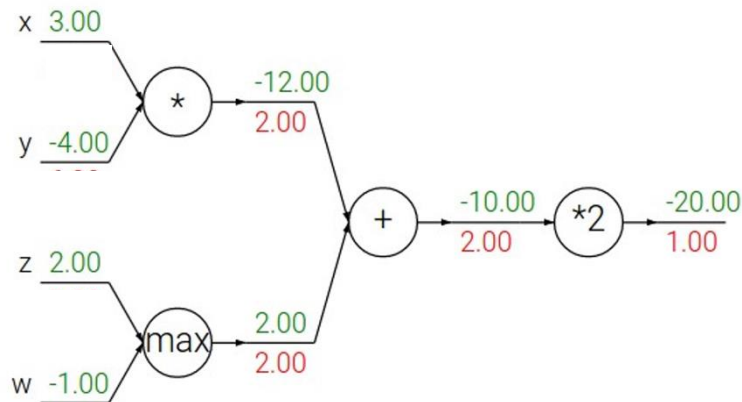
Gradient direction

Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(xy + \max(z, w))$

add gate: gradient replicator

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



Gradient direction

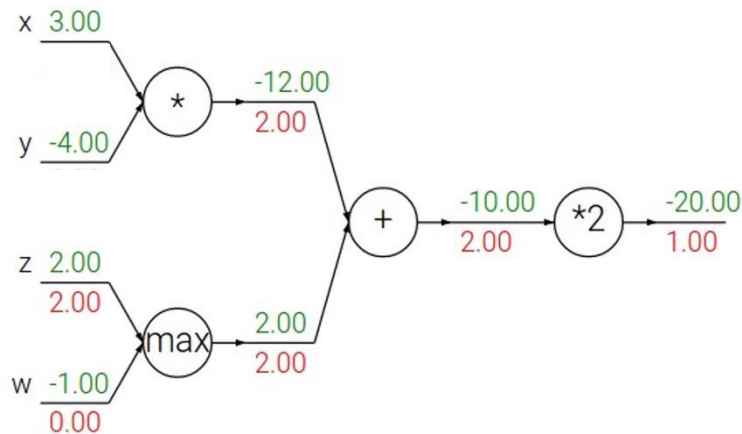
Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(xy + \max(z, w))$

add gate: gradient replicator

max gate: gradient router

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



Gradient direction

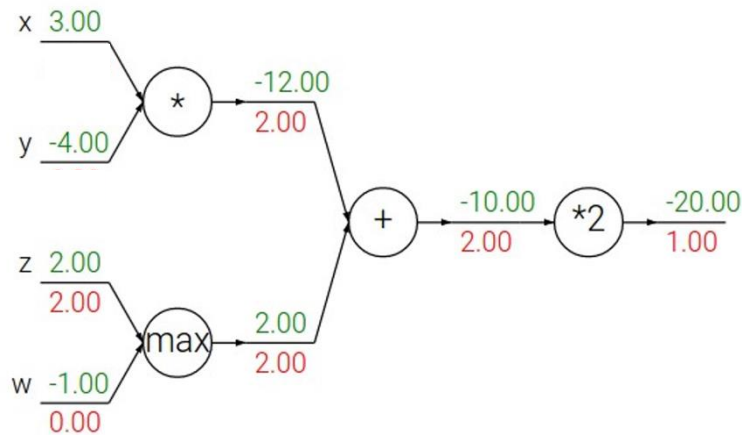
Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(xy + \max(z, w))$

add gate: gradient replicator

max gate: gradient router

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



Gradient direction

Patterns in backward flow

How does a local gradient modify the upstream gradient? $f = 2(xy + \max(z, w))$

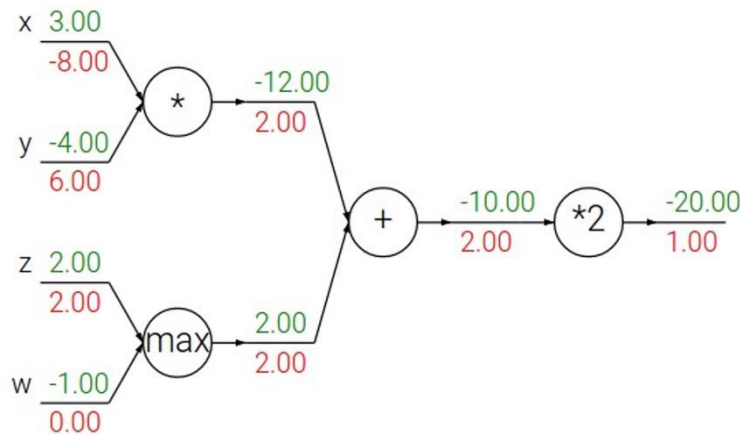
add gate: gradient replicator

max gate: gradient router

mul gate: gradient switcher

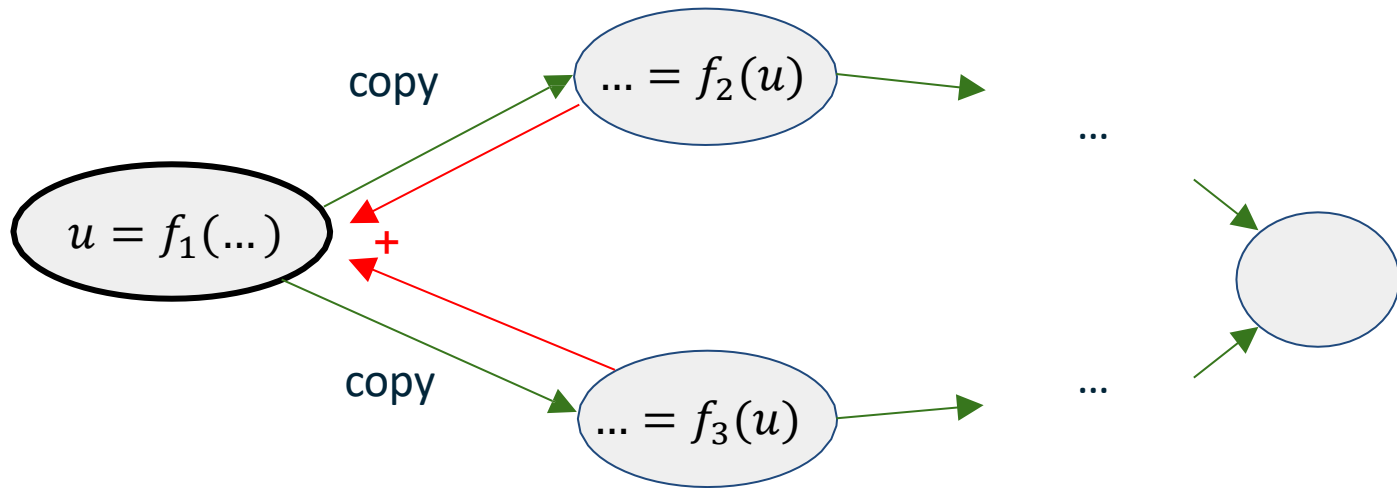
$$S = T V U$$

$$\frac{NS}{NT} = U \quad \frac{NS}{NU} = T$$



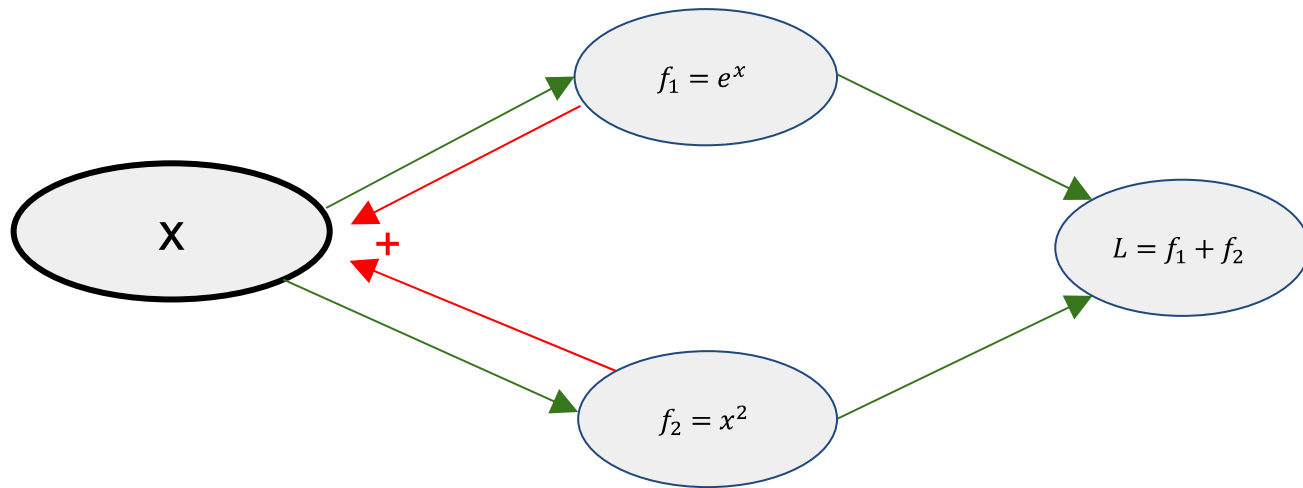
Gradient direction

Upstream gradients **add** at *fork branches*



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial L}{\partial f_2} \frac{\partial f_2}{\partial x}$$

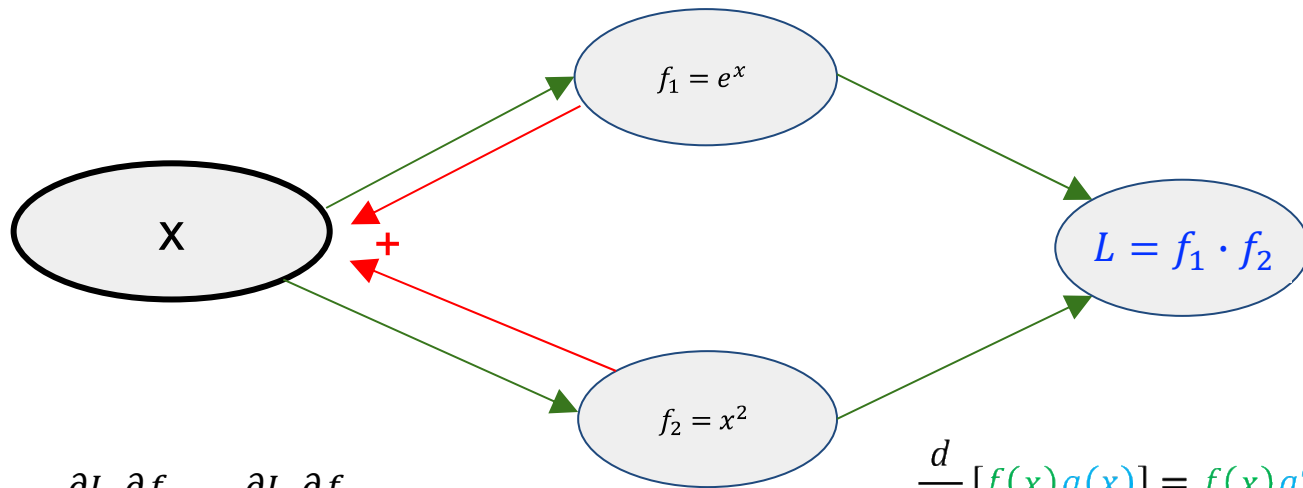
Upstream gradients **add** at *fork branches*



Claim:
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial L}{\partial f_2} \frac{\partial f_2}{\partial x}$$
$$= 1 \cdot e^x + 1 \cdot 2x = e^x + 2x$$

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

Upstream gradients add at fork branches



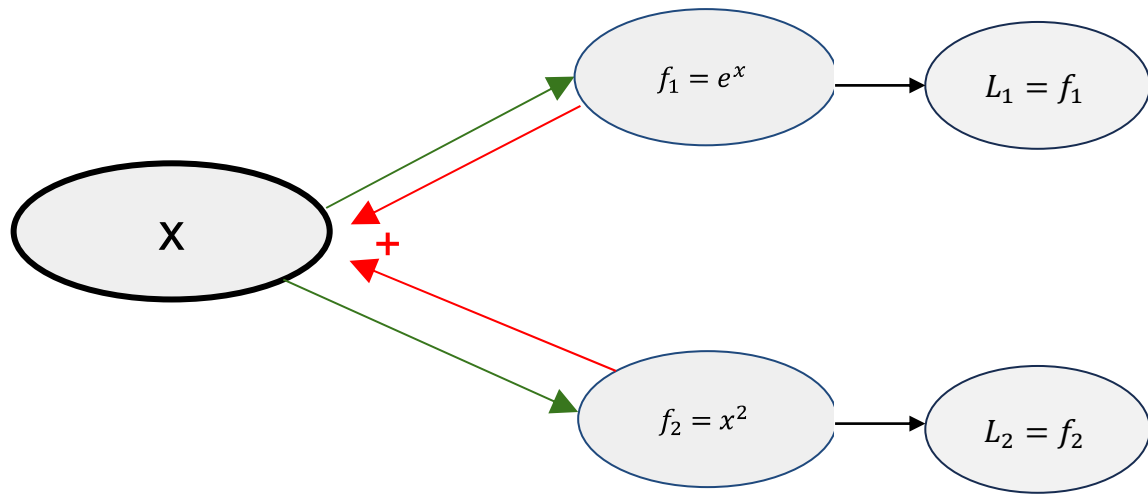
Claim:
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial L}{\partial f_2} \frac{\partial f_2}{\partial x}$$
$$= x^2 \cdot e^x + e^x \cdot 2x$$

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

Derivation: $L = e^x \cdot x^2$

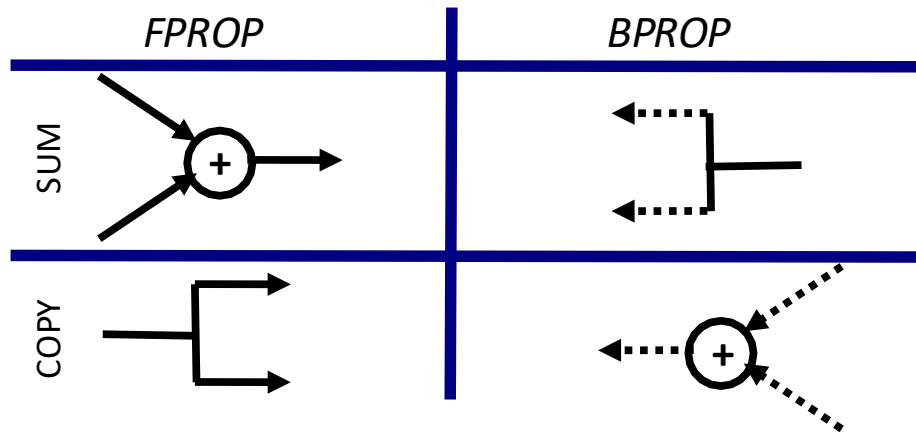
$$\frac{\partial L}{\partial x} = e^x \cdot 2x + e^x \cdot x^2$$

Upstream gradients don't have to come from the same loss



$$\frac{\partial L_1}{\partial x} + \frac{\partial L_2}{\partial x} = \frac{\partial L_1}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial L_2}{\partial f_2} \frac{\partial f_2}{\partial x}$$

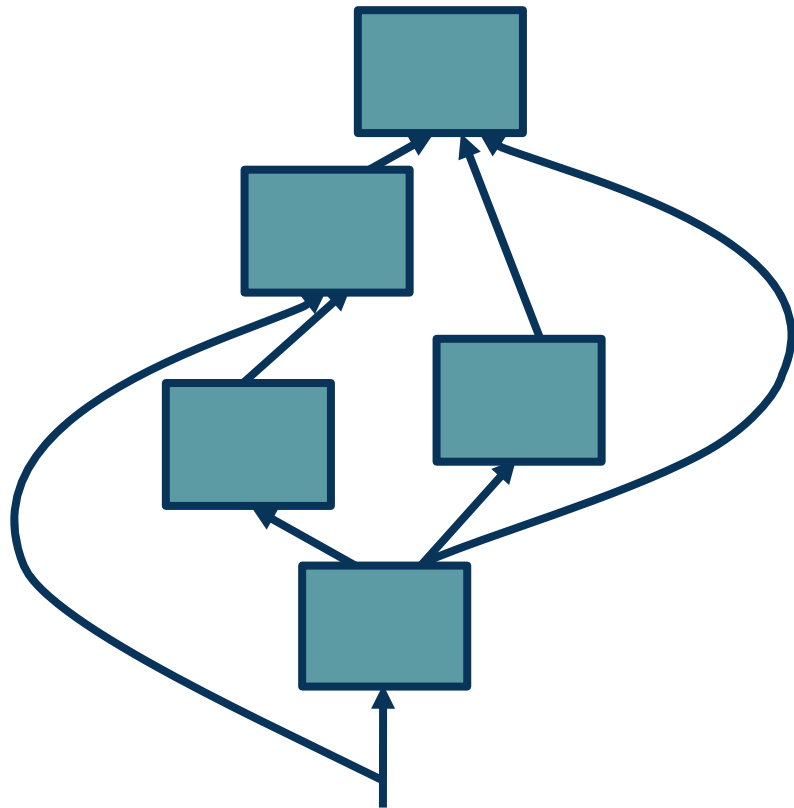
Duality in F(orward)prop and B(ack)prop



Can we backpropagate on *any* computation graph?

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

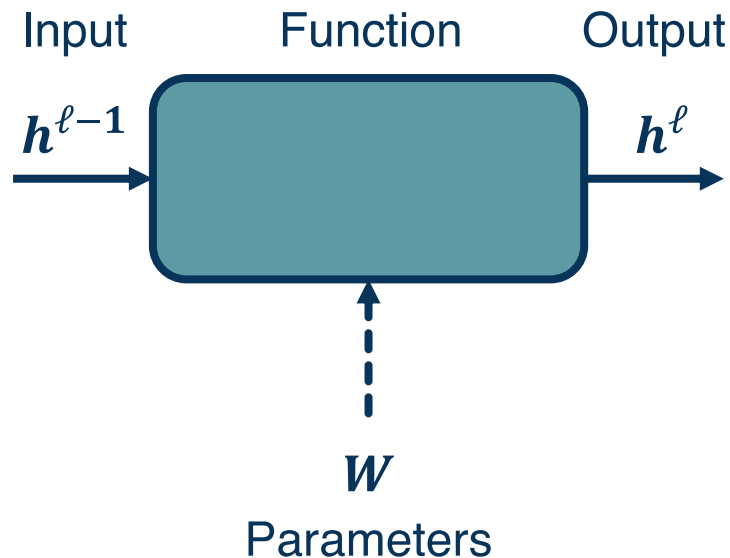
The **backpropagation algorithm** will then process this graph, **one node at a time**



Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

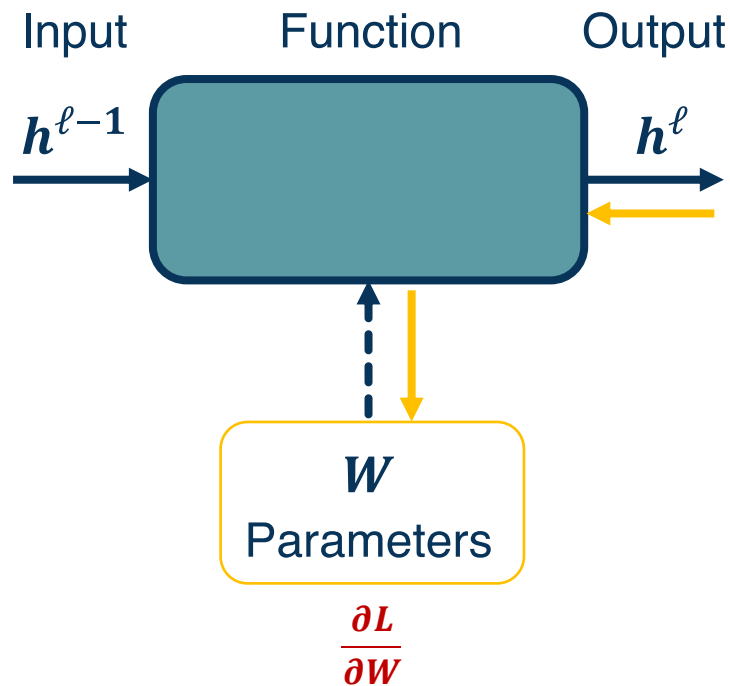
Given this computation graph, the training algorithm will:

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



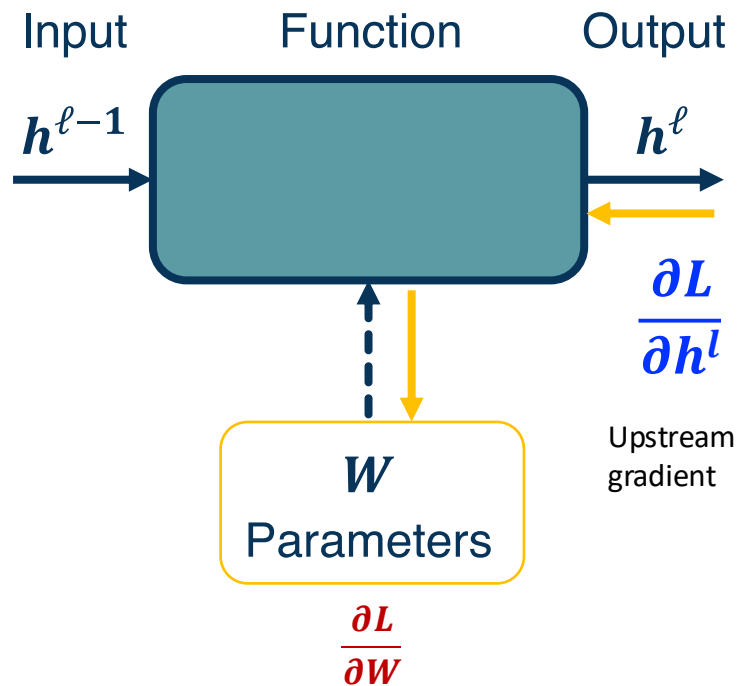
Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun

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



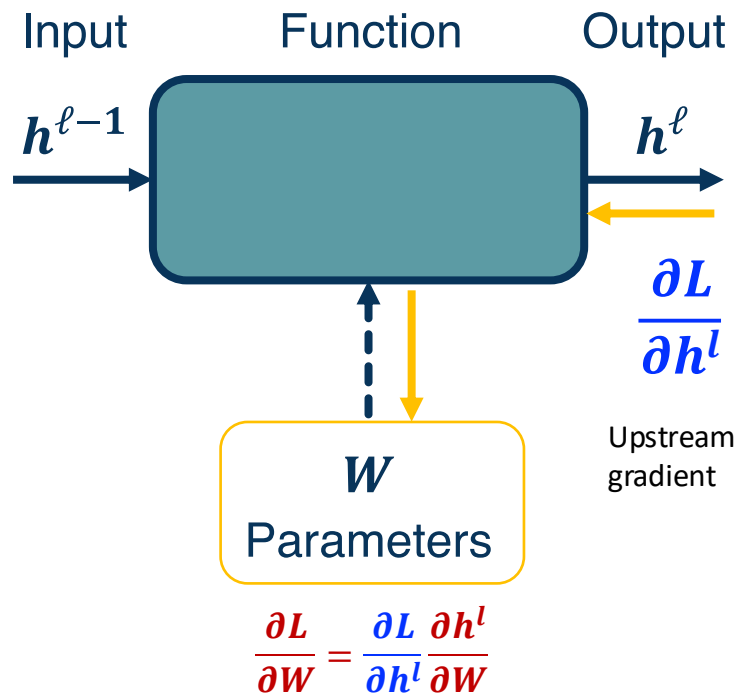
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module's parameters

- Assume that we have the gradient of the loss with respect to the **module's outputs** (given to us by upstream module)



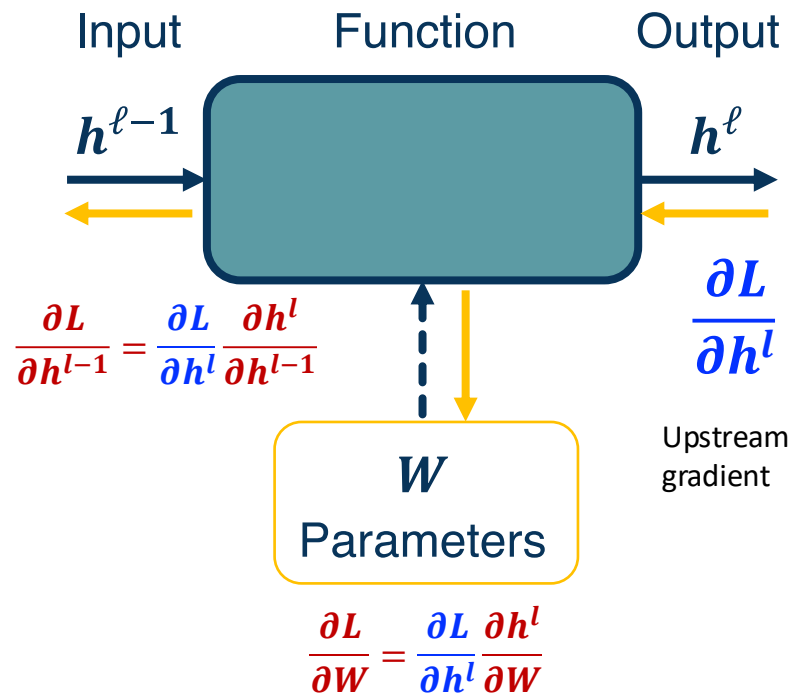
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module's parameters

- Assume that we have the gradient of the loss with respect to the **module's outputs** (given to us by upstream module)
- We can calculate the gradient of the loss with respect to the module's weights



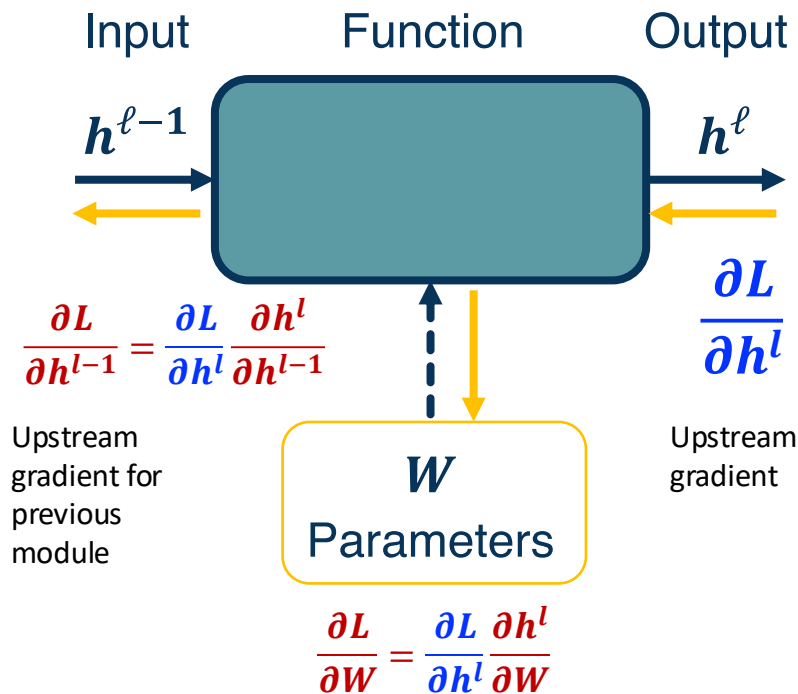
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module's parameters

- Assume that we have the gradient of the loss with respect to the **module's outputs** (given to us by upstream module)
- We can calculate the gradient of the loss with respect to the module's weights
- 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



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 can calculate the gradient of the loss with respect to the module's weights
- 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



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

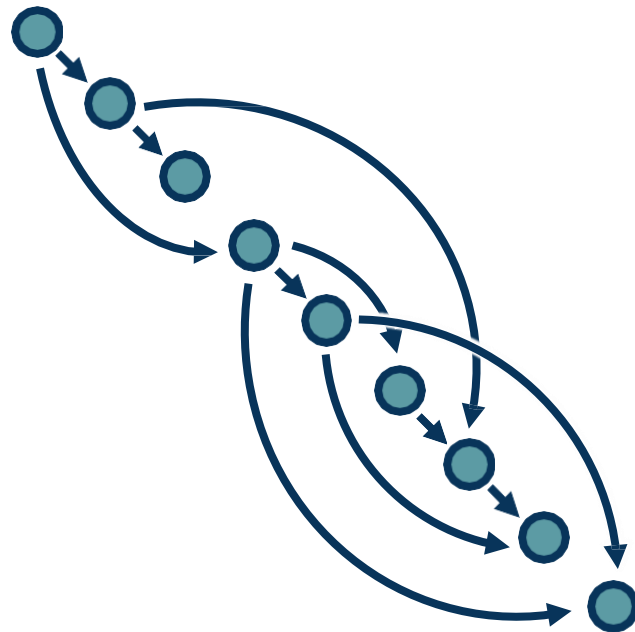
But the idea can be applied to **any directed acyclic graph (DAG)**

- 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**



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)

PyTorch: Computational Graphs

input image

loss

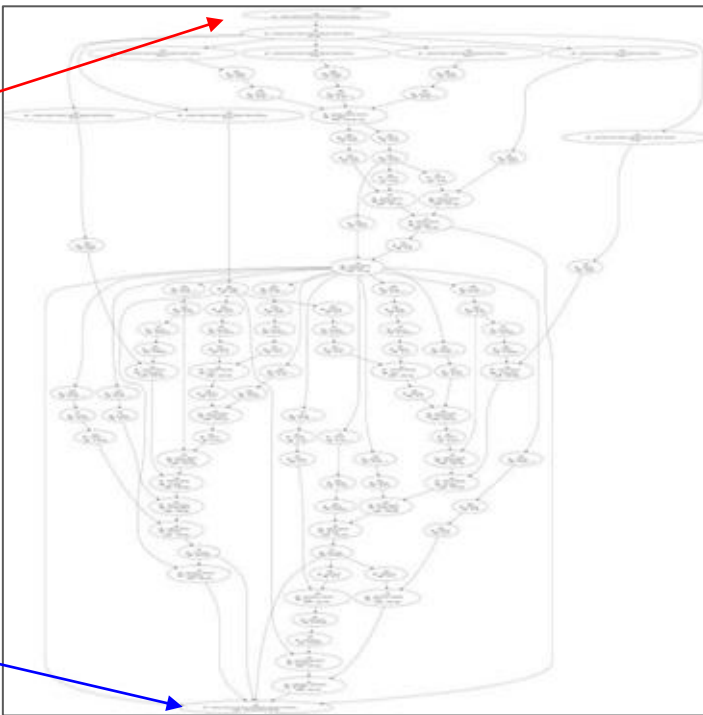


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

PyTorch: Tensors

Running example: Train
a two-layer ReLU
network on random data
with L2 loss

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Tensors

Create random tensors
for data and weights



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Tensors

Forward pass: compute predictions and loss

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

Calculate gradient for
each node (weight and
intermediate variables)
in the graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

Make gradient step on weights, then zero them. Torch.no_grad means “don’t build a computational graph for this part”

PyTorch: Autograd

PyTorch methods that end in underscore modify the Tensor in-place; methods that don't return a new Tensor

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Use ctx object to “cache” values for the backward pass

```
class MyReLU(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x):  
        ctx.save_for_backward(x)  
        return x.clamp(min=0)  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        x, = ctx.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input
```

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward functions for Tensors

Use ctx object to “cache” values for the backward pass

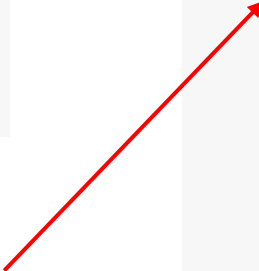
Define a helper function to make it easy to use the new function

```
class MyReLU(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x):  
        ctx.save_for_backward(x)  
        return x.clamp(min=0)  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        x, = ctx.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input  
  
def my_relu(x):  
    return MyReLU.apply(x)
```

PyTorch: New Autograd Functions

```
class MyReLU(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x):  
        ctx.save_for_backward(x)  
        return x.clamp(min=0)  
  
    @staticmethod  
    def backward(ctx, grad_y):  
        x, = ctx.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input  
  
def my_relu(x):  
    return MyReLU.apply(x)
```

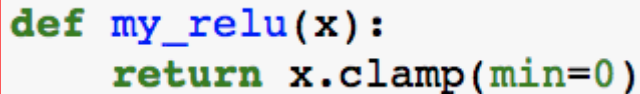
Can use our new autograd
function in the forward pass



```
N, D_in, H, D_out = 64, 1000, 100, 10  
  
x = torch.randn(N, D_in)  
y = torch.randn(N, D_out)  
w1 = torch.randn(D_in, H, requires_grad=True)  
w2 = torch.randn(H, D_out, requires_grad=True)  
  
learning_rate = 1e-6  
for t in range(500):  
    y_pred = my_relu(x.mm(w1)).mm(w2)  
    loss = (y_pred - y).pow(2).sum()  
  
    loss.backward()  
  
    with torch.no_grad():  
        w1 -= learning_rate * w1.grad  
        w2 -= learning_rate * w2.grad  
        w1.grad.zero_()  
        w2.grad.zero_()
```

PyTorch: New Autograd Functions

```
def my_relu(x):  
    return x.clamp(min=0)
```



In practice you almost never need to define new autograd functions! Only do it when you need custom backward. In this case we can just use a normal PyTorch function

```
N, D_in, H, D_out = 64, 1000, 100, 10  
  
x = torch.randn(N, D_in)  
y = torch.randn(N, D_out)  
w1 = torch.randn(D_in, H, requires_grad=True)  
w2 = torch.randn(H, D_out, requires_grad=True)  
  
learning_rate = 1e-6  
for t in range(500):  
    y_pred = my_relu(x.mm(w1)).mm(w2)  
    loss = (y_pred - y).pow(2).sum()  
  
    loss.backward()  
  
    with torch.no_grad():  
        w1 -= learning_rate * w1.grad  
        w2 -= learning_rate * w2.grad  
        w1.grad.zero_()  
        w2.grad.zero_()
```

PyTorch: Dynamic Computation Graphs

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

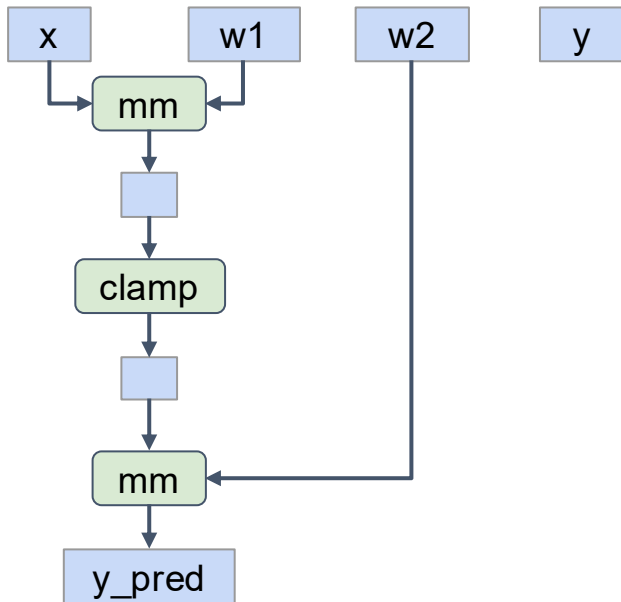
```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Create Tensor objects

PyTorch: Dynamic Computation Graphs



```
import torch

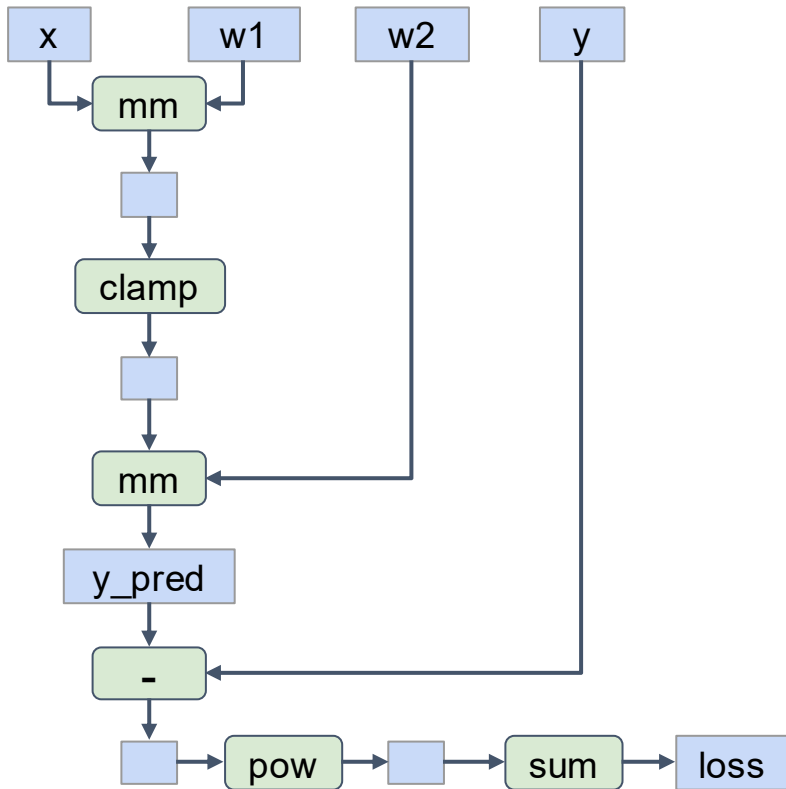
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

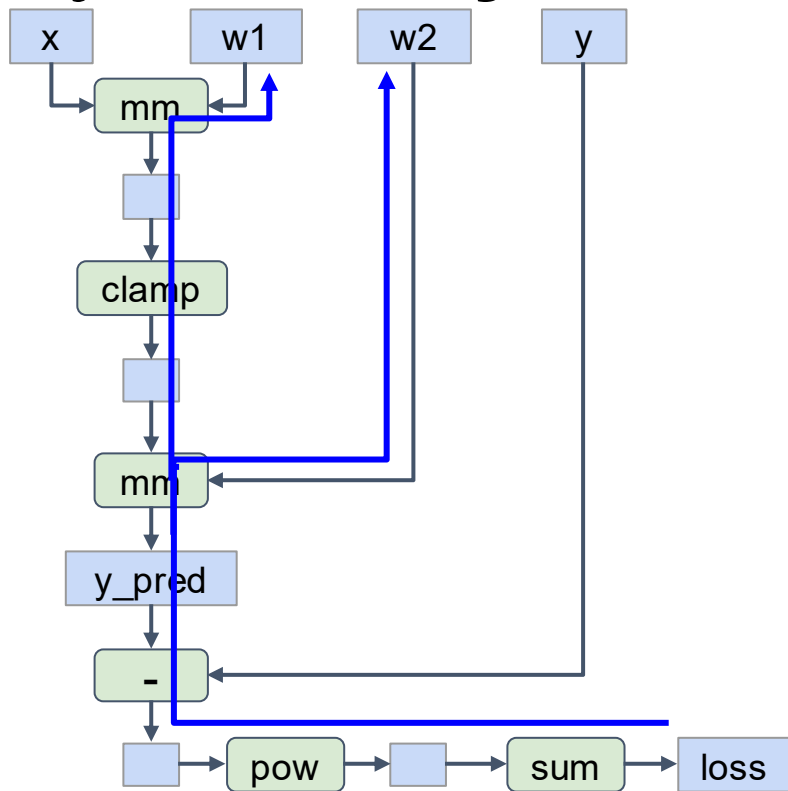
```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: Dynamic Computation Graphs

x

w1

w2

y

```
import torch

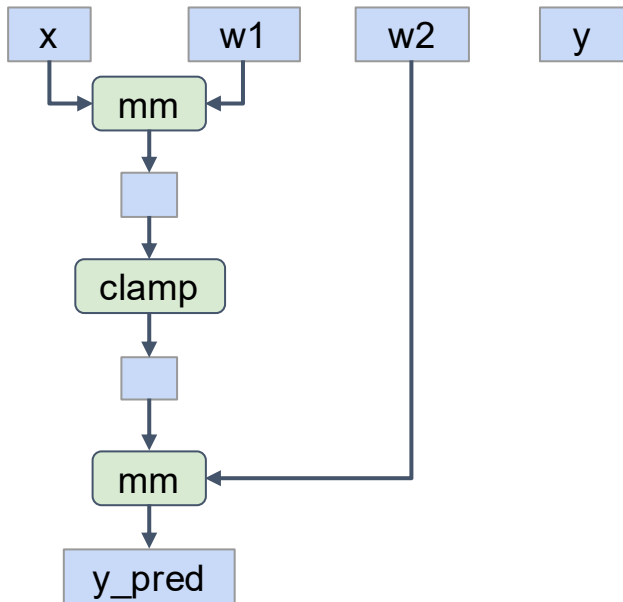
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Throw away the graph, backprop path, and rebuild it from scratch on every iteration

PyTorch: Dynamic Computation Graphs



```
import torch

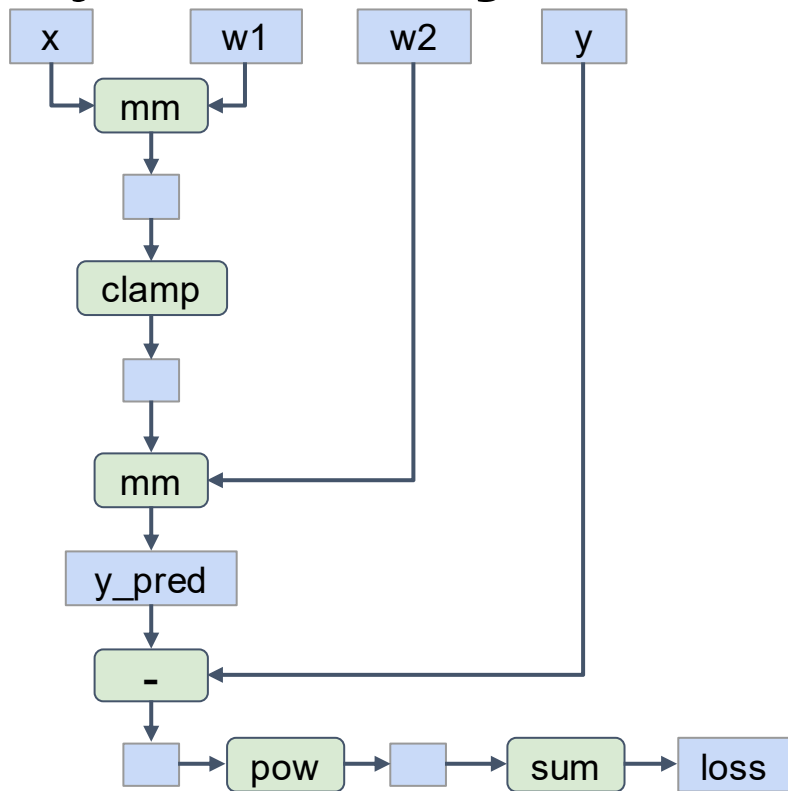
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

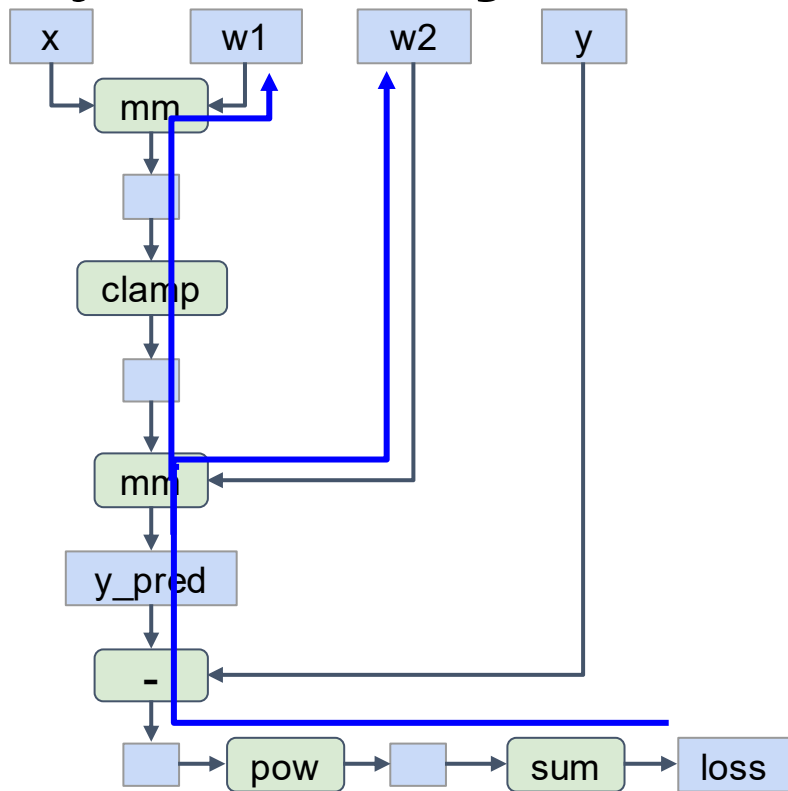
```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Build graph data structure AND
perform computation

PyTorch: Dynamic Computation Graphs



```
import torch
```

```
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)
```

```
w1 = torch.randn(D_in, H, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, requires_grad=True)
```

```
learning_rate = 1e-6
```

```
for t in range(500):
```

```
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
```

```
    loss = (y_pred - y).pow(2).sum()
```

```
    loss.backward()
```

Search for path between loss and w1, w2
(for backprop) AND perform computation

PyTorch: **Dynamic** Computation Graphs

Building the graph and
computing the graph happen at
the same time.

Seems inefficient, especially if we
are building the same graph over
and over again...

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

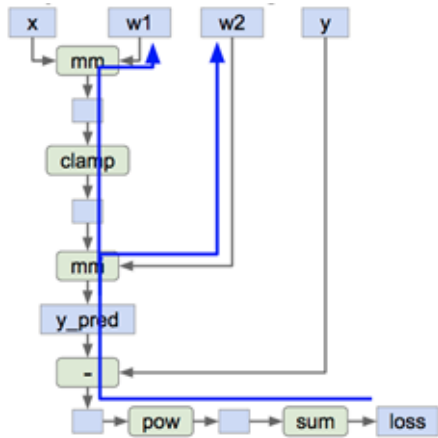
    loss.backward()
```

Static Computation Graphs

Alternative: **Static** graphs

Step 1: Build computational graph describing our computation (including finding paths for backprop)

Step 2: Reuse the same graph on every iteration



```
graph = build_graph()
```

```
for x_batch, y_batch in loader:  
    run_graph(graph, x=x_batch, y=y_batch)
```

@tf.function: compile static graph

tf.function decorator
(implicitly) compiles
python functions to
static graph for better
performance

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```



```
@tf.function
def model_func(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

for t in range(50):
    with tf.GradientTape() as tape:
        y_pred, loss = model_func(x, y)
    gradients = tape.gradient(
        loss, model.trainable_variables)
    optimizer.apply_gradients(
        zip(gradients, model.trainable_variables))
```

@tf.function: compile static graph

Here we compare the forward-pass time of the same model under dynamic graph mode and static graph mode

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss

def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))

dynamic graph: 0.02520249200000535
static graph: 0.03932226699998864
```

@tf.function: compile static graph

Static graph is *in theory* faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss
```

```
def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph: ", timeit.timeit(lambda: model_dynamic(x, y), number=10))
print("static graph: ", timeit.timeit(lambda: model_static(x, y), number=10))

dynamic graph: 0.02520249200000535
static graph: 0.03932226699998864
```

Run the same ops 10 times

@tf.function: compile static graph

Static graph is *in theory* faster than dynamic graph, but the performance gain depends on the type of model / layer / computation graph.

```
N, D, H = 64, 1000, 100
x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,), activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)
```

```
@tf.function
def model_static(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
    return y_pred, loss
```

```
def model_dynamic(x, y):
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)

print("dynamic graph:", timeit.timeit(lambda: model_dynamic(x, y), number=1000))
print("static graph:", timeit.timeit(lambda: model_static(x, y), number=1000))

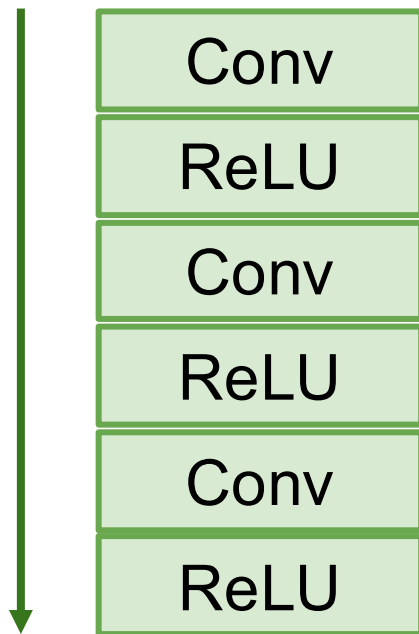
dynamic graph: 2.3648411540000325
static graph: 1.1723986679999143
```

Run the same ops 1000 times

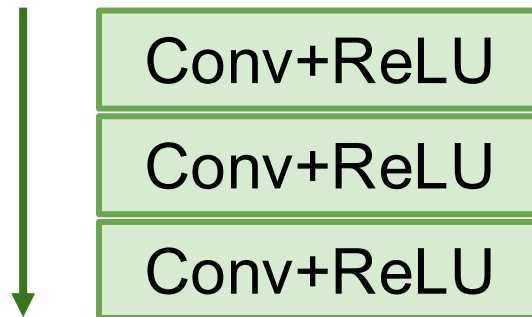
Static vs Dynamic: Optimization

With static graphs, framework can **optimize** the graph for you before it runs!

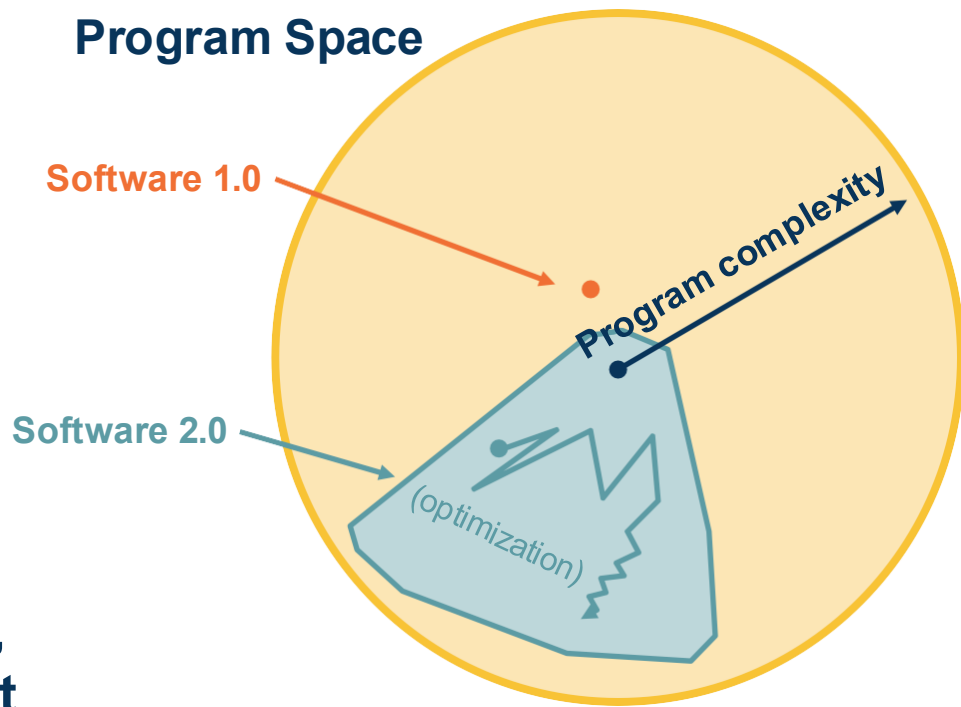
The graph you wrote



Equivalent graph with **fused operations**



- 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](#)

Next time:

- Math on backprop but for (shallow) neural nets!
- Jacobians
- Activation functions