

Lecture 3

Neural Network

Artificial Intelligence Class
Ajou University

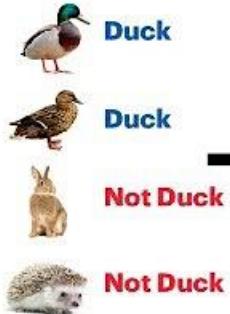
Jongbin Ryu



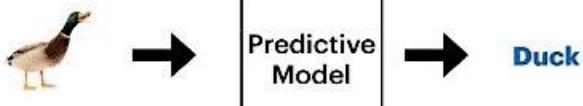
Supervised Learning vs Unsupervised Learning

Supervised Learning

Regression/Classification



Supervised Learning → Predictive Model



Unsupervised Learning

Clustering/Dim. Reduction

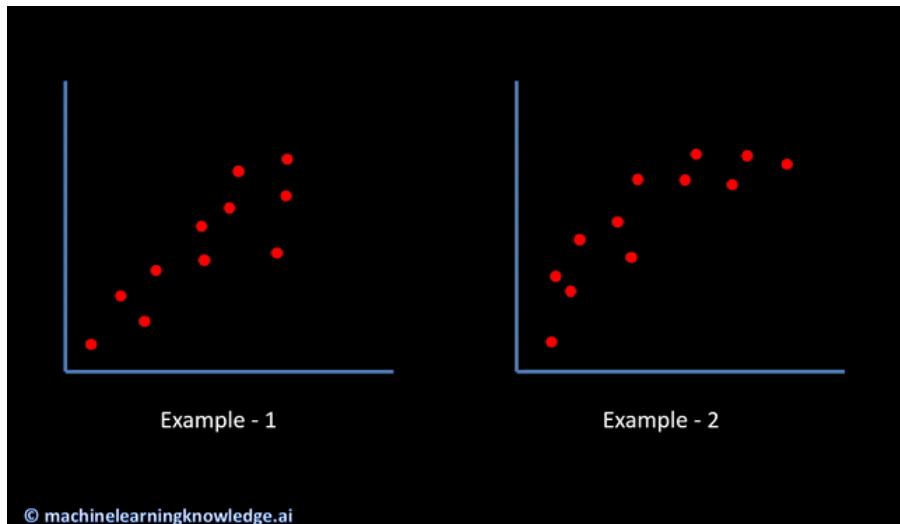


Unsupervised Learning → Clusters

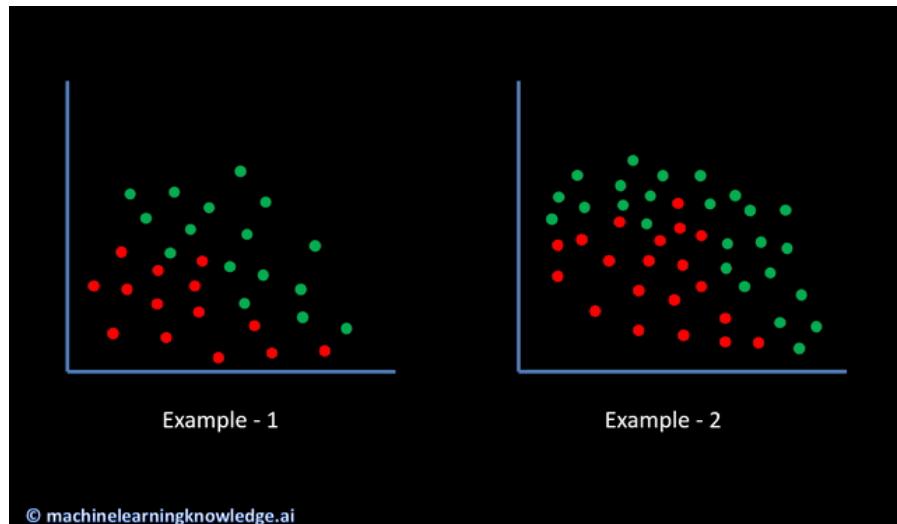


Supervised Learning

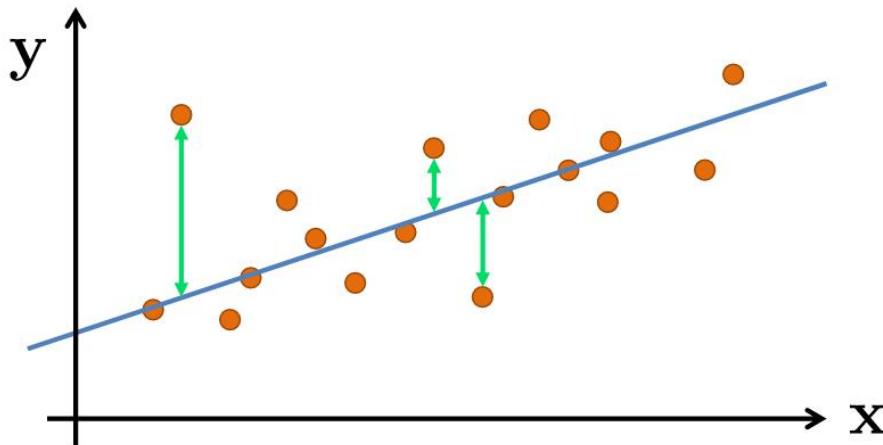
Regression



Classification



Linear Regression



Minimizing

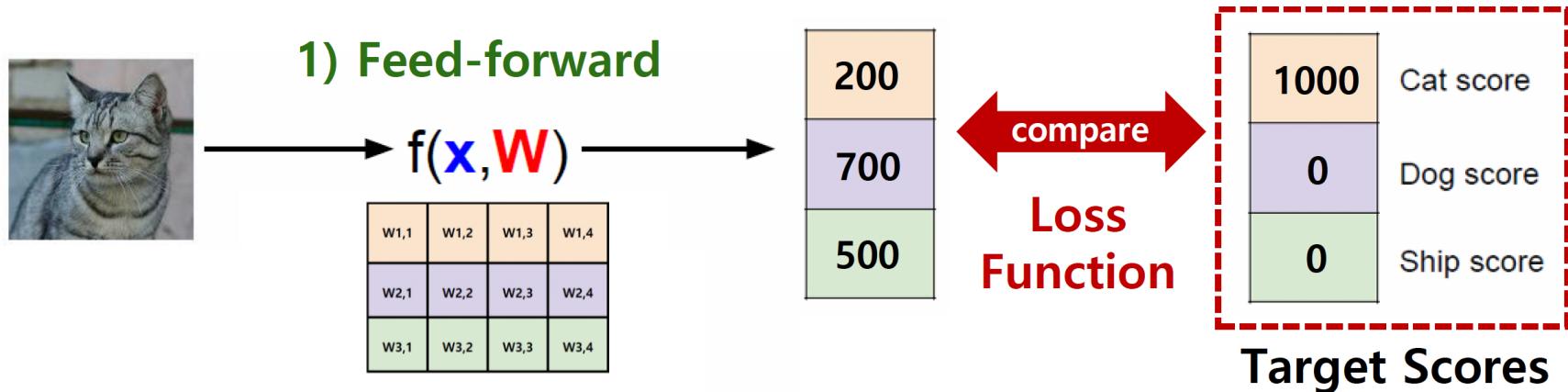
$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

MSE(Mean Squared Error)

Objective function
Energy
Cost function

48

Linear Classifier

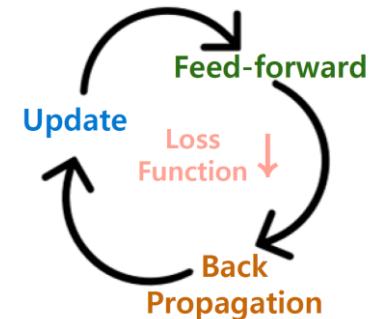


Goal

Set the W(weights) to
increase the correct class score

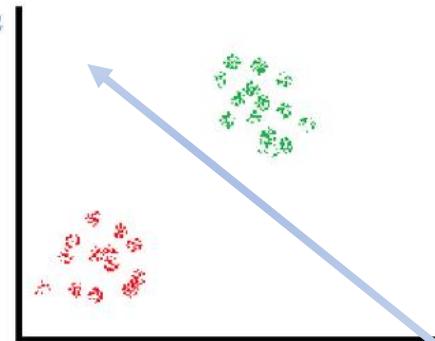


Set the W(weights) to
decrease the loss function

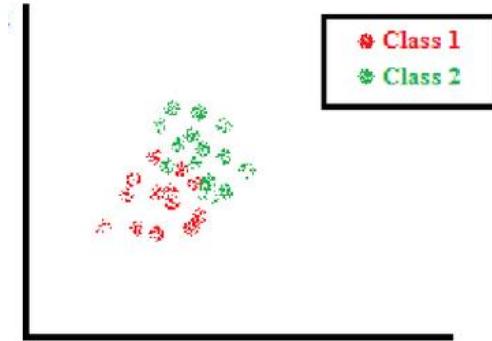


Challenges: Intra-class variation vs Inter-class variation

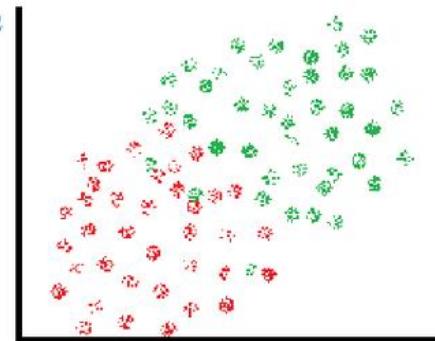
```
def classify_image(image):  
    # Some magic here?  
    return class_label
```



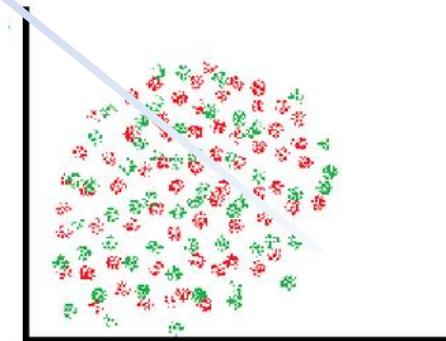
(a)



(b)



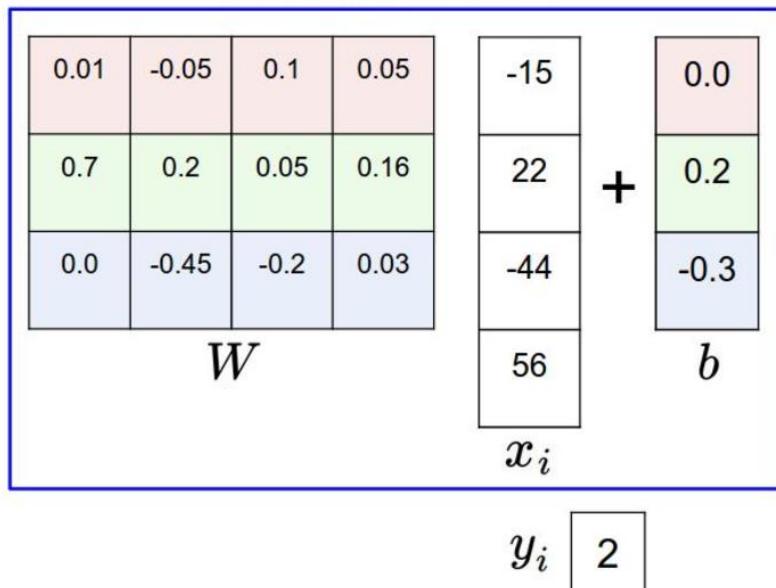
(c)



(d)

Softmax vs. SVM

matrix multiply + bias offset



hinge loss (SVM)

$$\max(0, -2.85 - 0.28 + 1) + \max(0, 0.86 - 0.28 + 1) = 1.58$$

cross-entropy loss (Softmax)

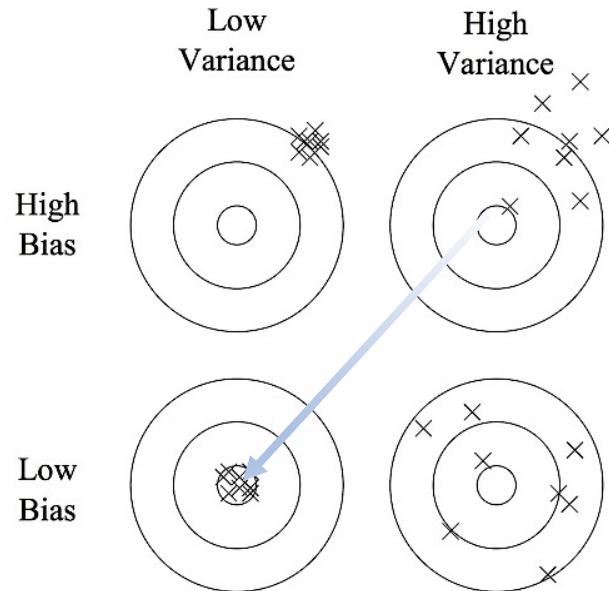
exp →

$$\begin{bmatrix} -2.85 \\ 0.86 \\ 0.28 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 0.058 \\ 2.36 \\ 1.32 \end{bmatrix} \xrightarrow[\text{(to sum to one)}]{\text{normalize}} \begin{bmatrix} 0.016 \\ 0.631 \\ 0.353 \end{bmatrix}$$
$$- \log(0.353) = 0.452$$

	Underfitting	Just right	Overfitting
Symptoms	<ul style="list-style-type: none"> • High training error • Training error close to test error • High bias 	<ul style="list-style-type: none"> • Training error slightly lower than test error 	<ul style="list-style-type: none"> • Very low training error • Training error much lower than test error • High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	<ul style="list-style-type: none"> • Complexify model • Add more features • Train longer 		<ul style="list-style-type: none"> • Perform regularization • Get more data

$$\text{Error}(X) = \text{noise}(X) + \text{bias}(X) + \text{variance}(X)$$

*irreducible error
reducible error*



Recap

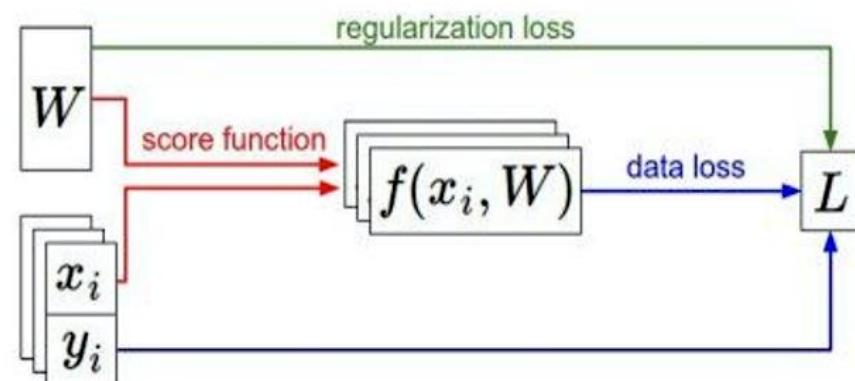
- We have some dataset of (x, y)
- We have a **score function**: $s = f(x; W) = Wx$ e.g.
- We have a **loss function**:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right) \quad \text{Softmax}$$

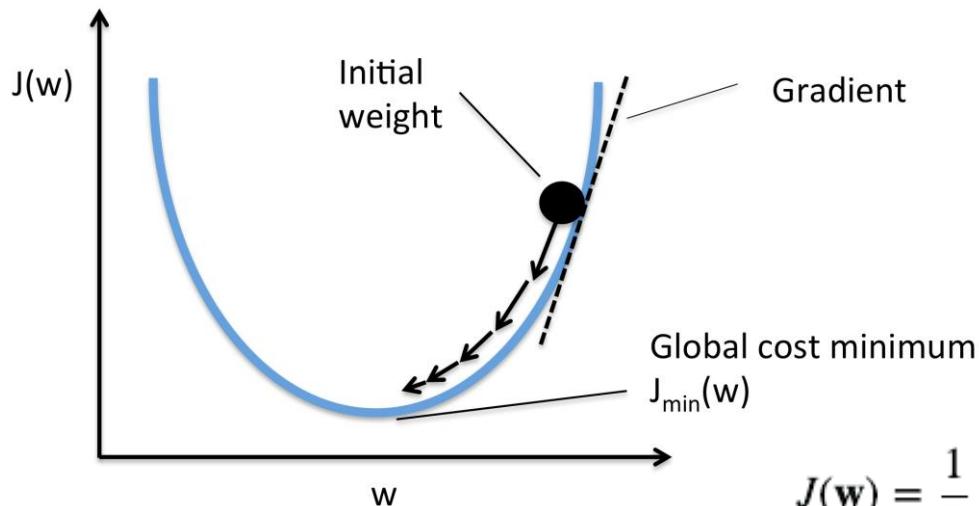
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W) \quad \text{Full loss}$$

How do we find the best W ?



Follow the slope

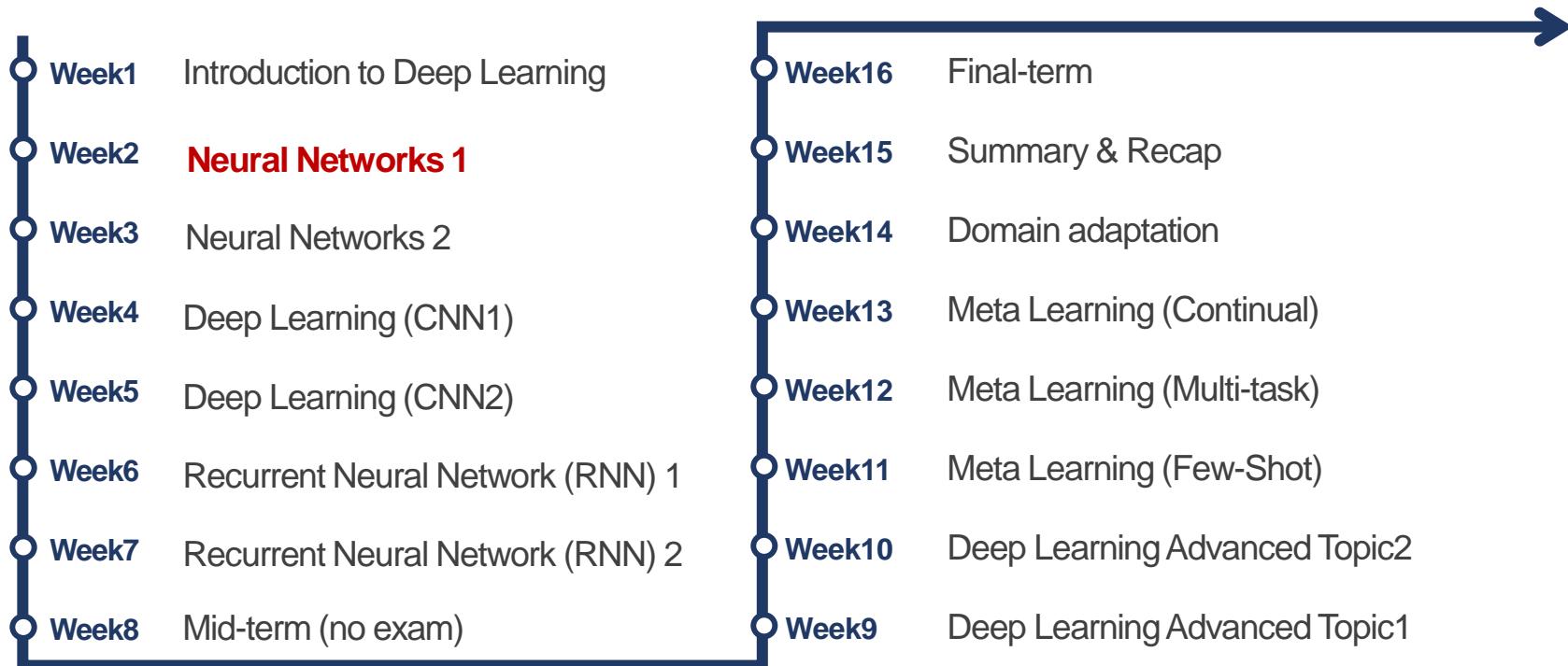


$$J(\mathbf{w}) = \frac{1}{2} \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \quad \Delta w_j = -\eta \frac{\partial J}{\partial w_j}$$

Schedule (인공지능)

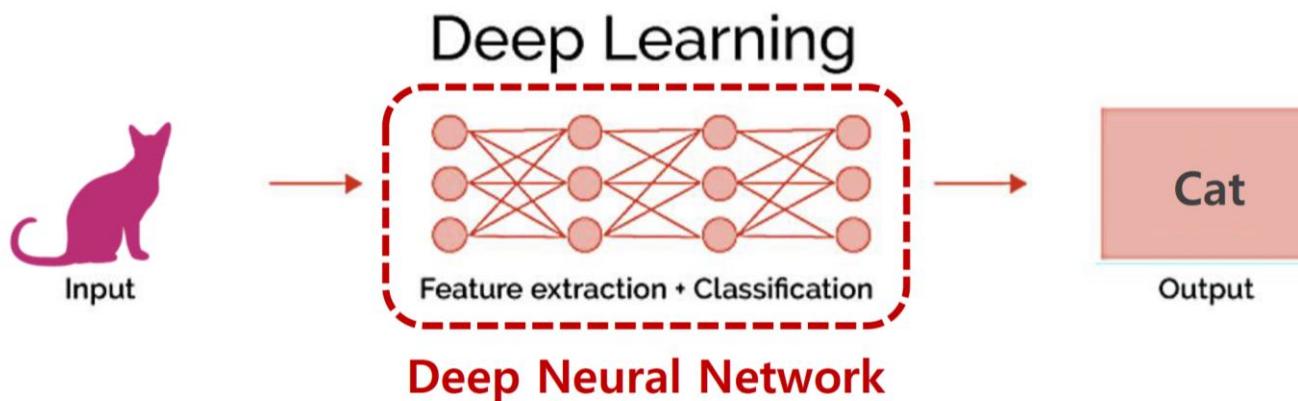
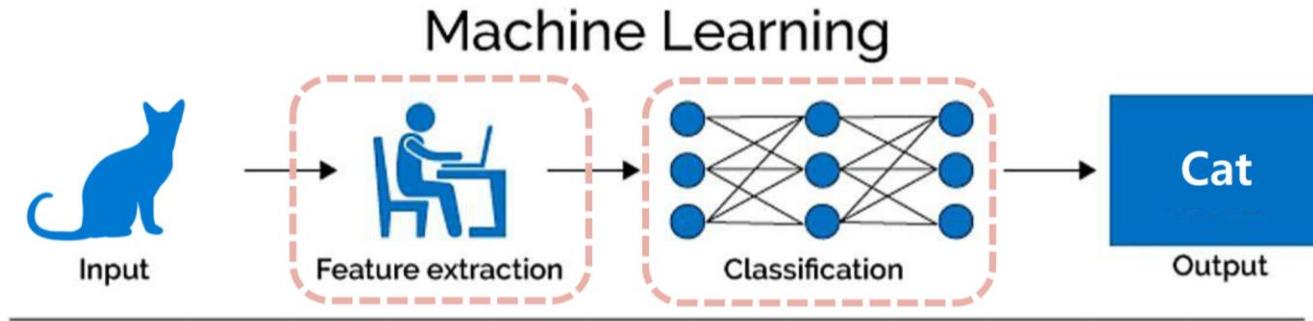
2th Week



Contents

- Traditional ML Approaches
- Introduction to Neural Networks
- Activation Functions (Nonlinearity)
- Gradient Descent
- Computational Graphs & Backpropagation

Machine Learning vs Deep Learning



Pixel Features

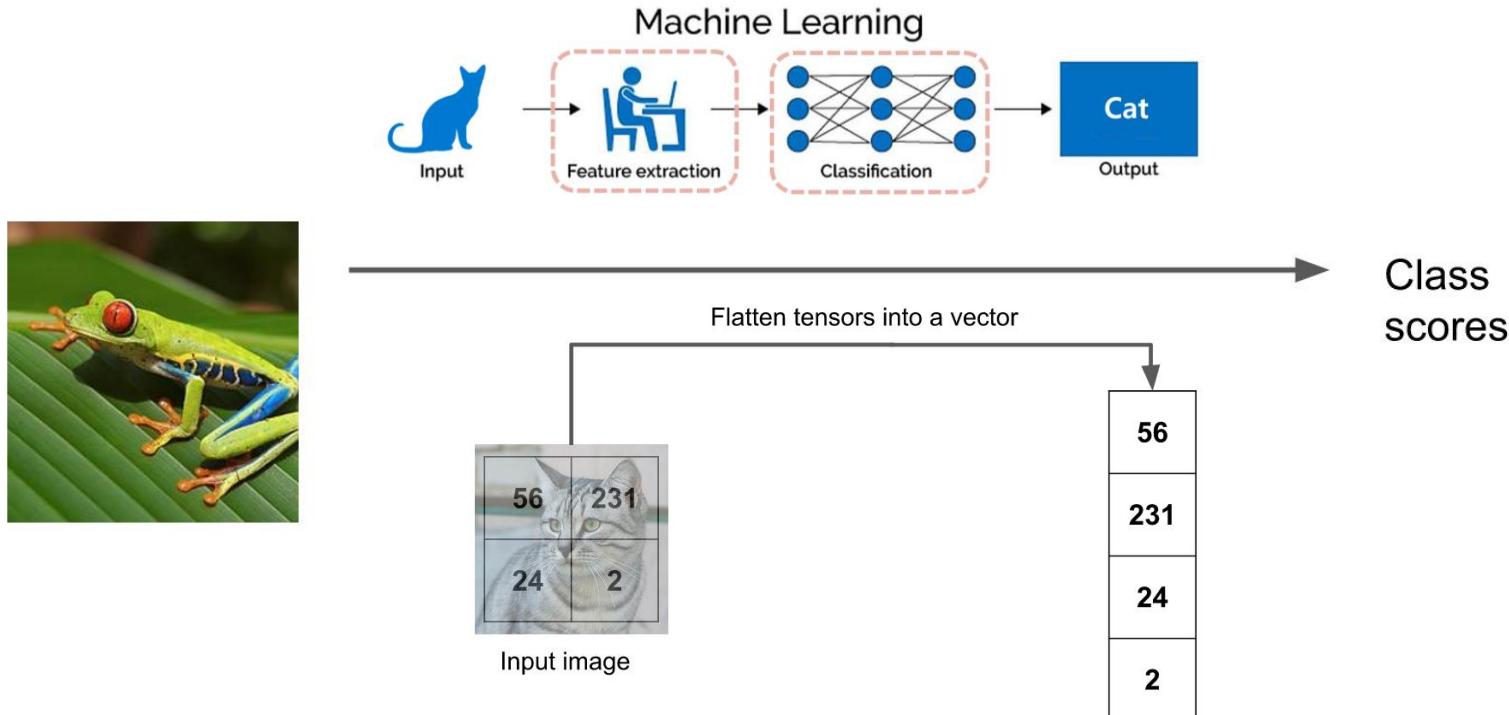
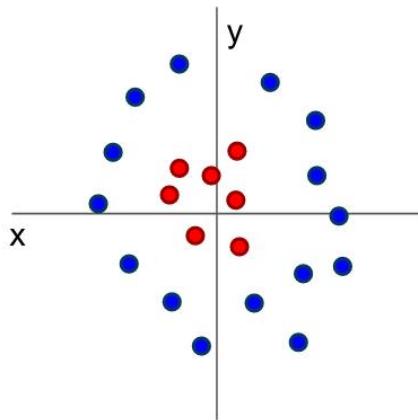
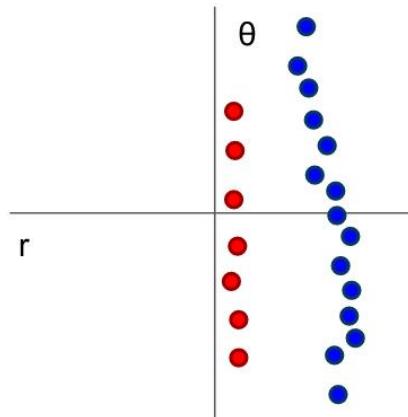


Image Features: Motivation



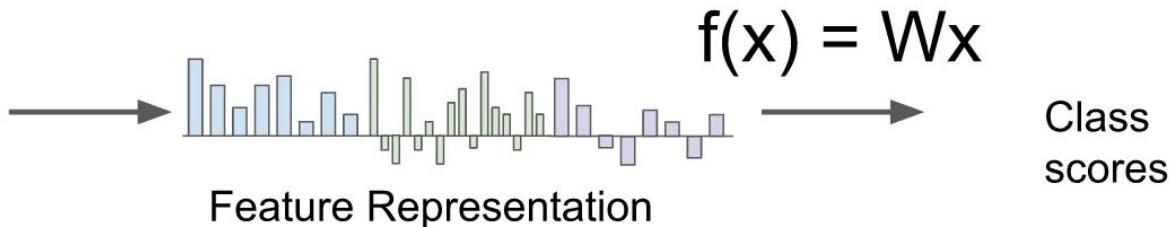
Cannot separate red
and blue points with
linear classifier

$$f(x, y) = (r(x, y), \theta(x, y))$$

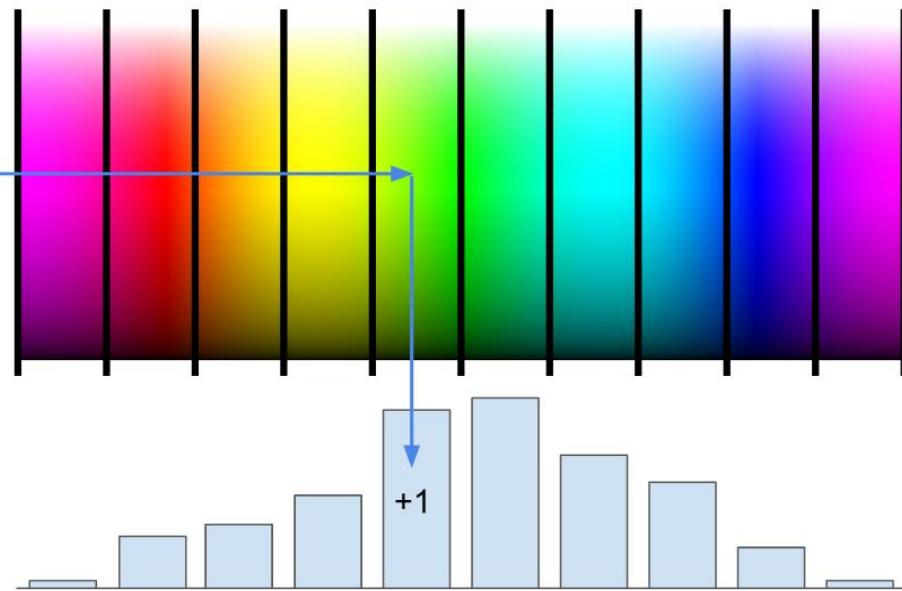


After applying feature
transform, points can
be separated by linear
classifier

Image Features



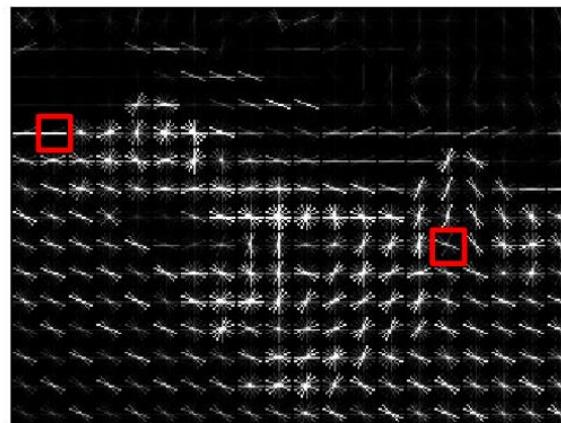
Example: Color Histogram



Example: Histogram of Oriented Gradients (HoG)



Divide image into 8x8 pixel regions
Within each region quantize edge
direction into 9 bins



Example: 320x240 image gets divided
into 40x30 bins; in each bin there are
9 numbers so feature vector has
 $30 \times 40 \times 9 = 10,800$ numbers

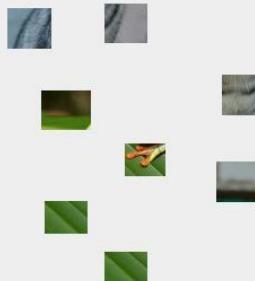
Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

Example: Bag of Words

Step 1: Build codebook



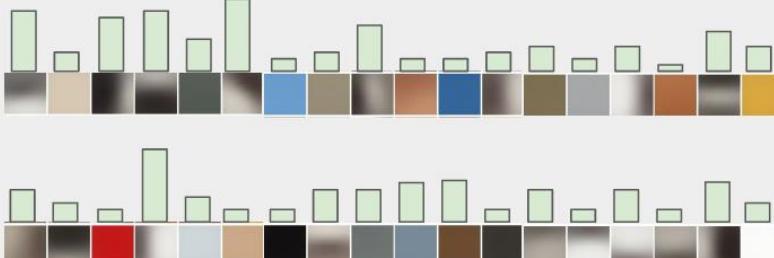
Extract random patches



Cluster patches to form “codebook” of “visual words”



Step 2: Encode images



Fei-Fei and Perona, "A bayesian hierarchical model for learning natural scene categories", CVPR 2005

Image Features

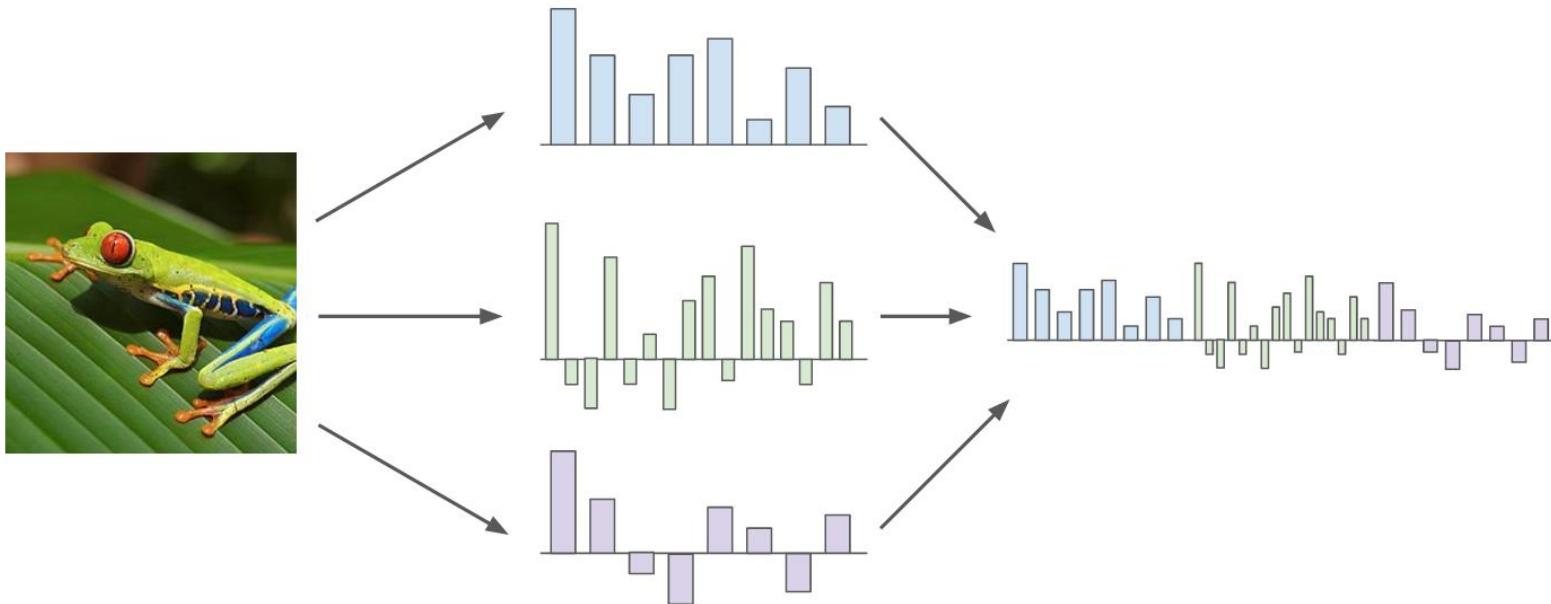
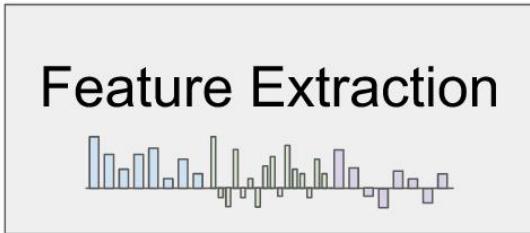


Image features vs ConvNets

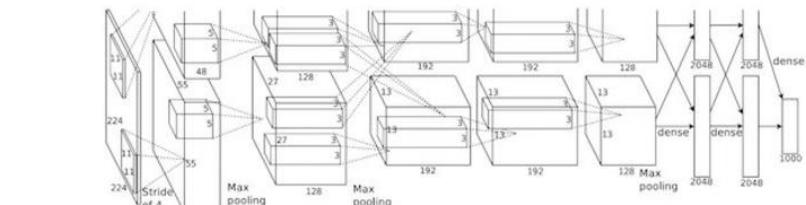


f



10 numbers giving
scores for classes

training



Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012.
Figure copyright Krizhevsky, Sutskever, and Hinton, 2012.
Reproduced with permission.

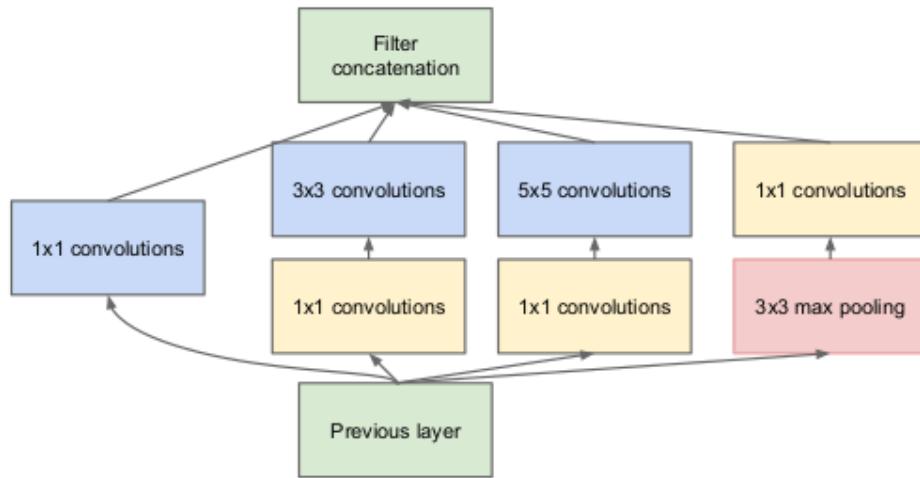
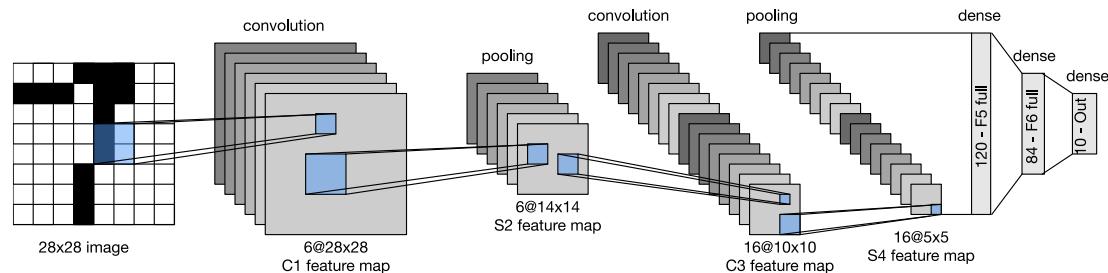
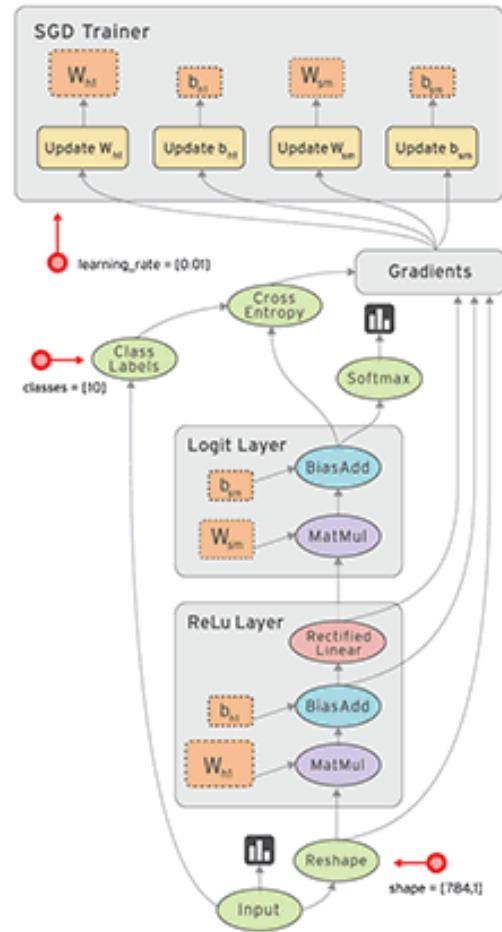


10 numbers giving
scores for classes

training

Contents

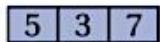
- Traditional ML Approaches
- **Introduction to Neural Networks**
- Activation Functions (Nonlinearity)
- Gradient Descent
- Computational Graphs & Backpropagation



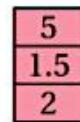
From Scalar to Tensor

(11)

SCALAR



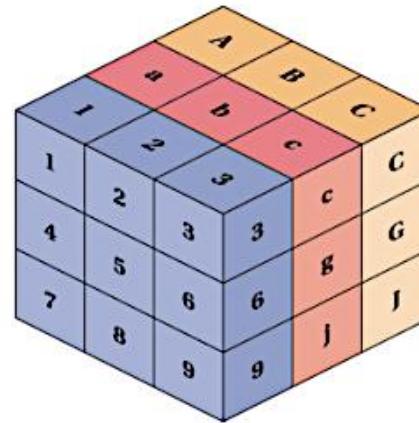
Row Vector
(shape 1x3)



Column Vector
(shape 3x1)

$$\begin{bmatrix} 4 & 19 & 8 \\ 16 & 3 & 5 \end{bmatrix}$$

MATRIX



TENSOR

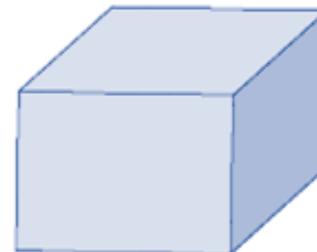
From Scalar to Tensor



1 d -Tensor



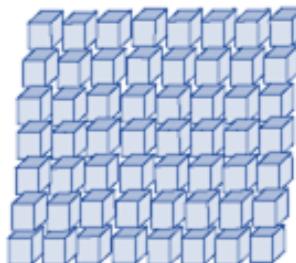
2 d -Tensor



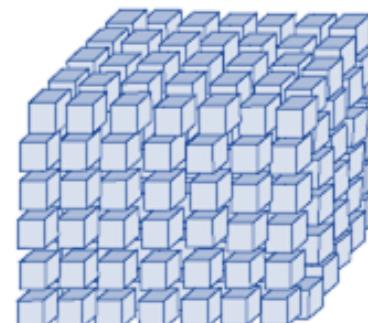
3 d -Tensor



4 d -Tensor



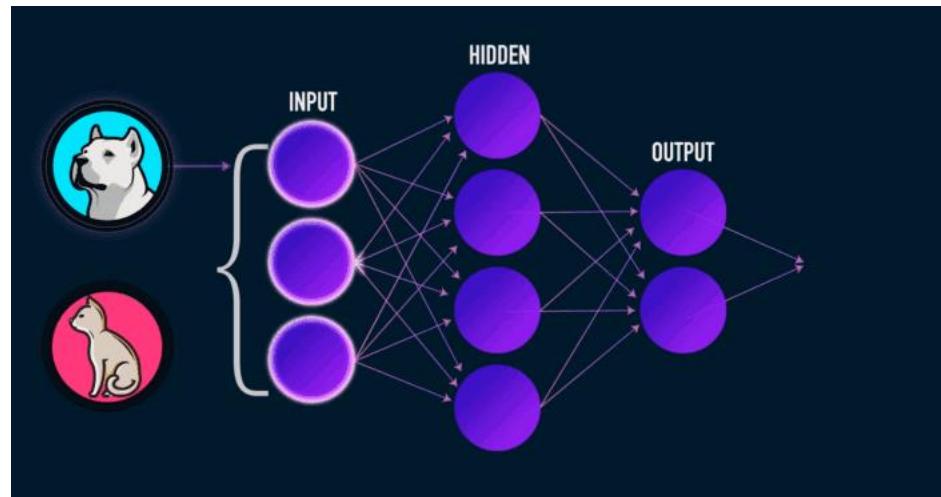
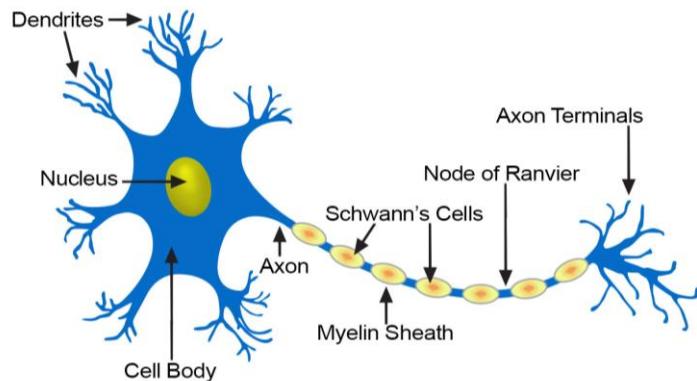
5 d -Tensor



6 d -Tensor

Neural Networks (NN)

Artificial neural networks (ANNs), usually simply called **neural networks (NNs)** are computing systems inspired by the biological neural networks



Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

“Neural Network” is a very broad term; these are more accurately called “fully-connected networks” or sometimes “multi-layer perceptrons” (MLP)

(In practice we will usually add a learnable bias at each layer as well)

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$
or 3-layer Neural Network

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

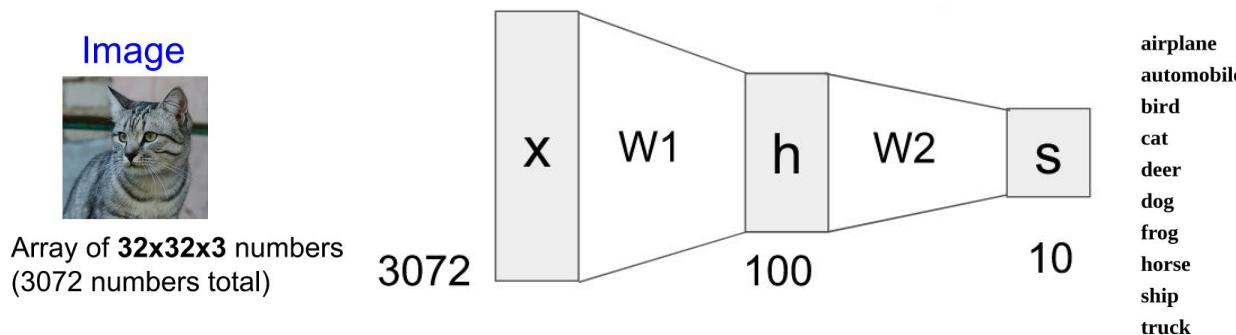
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

(In practice we will usually add a learnable bias at each layer as well)

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

The function $\max(0, z)$ is called the **activation function**.

Q: What if we try to build a neural network without one?

$$f = W_2 W_1 x$$

Neural networks: without the brain stuff

(Before) Linear score function: $f = Wx$

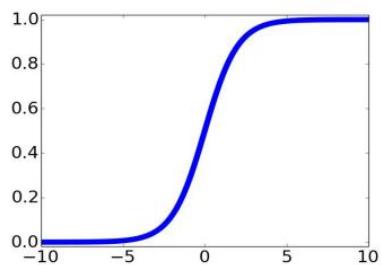
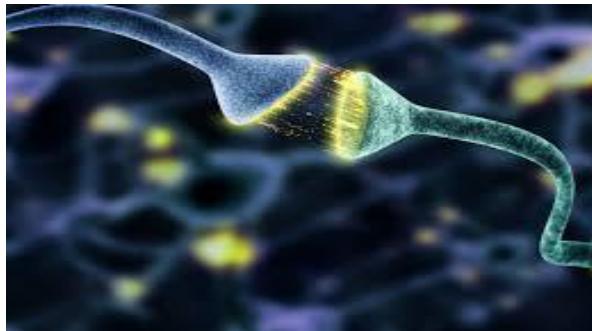
(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

The function $\max(0, z)$ is called the **activation function**.

Q: What if we try to build a neural network without one?

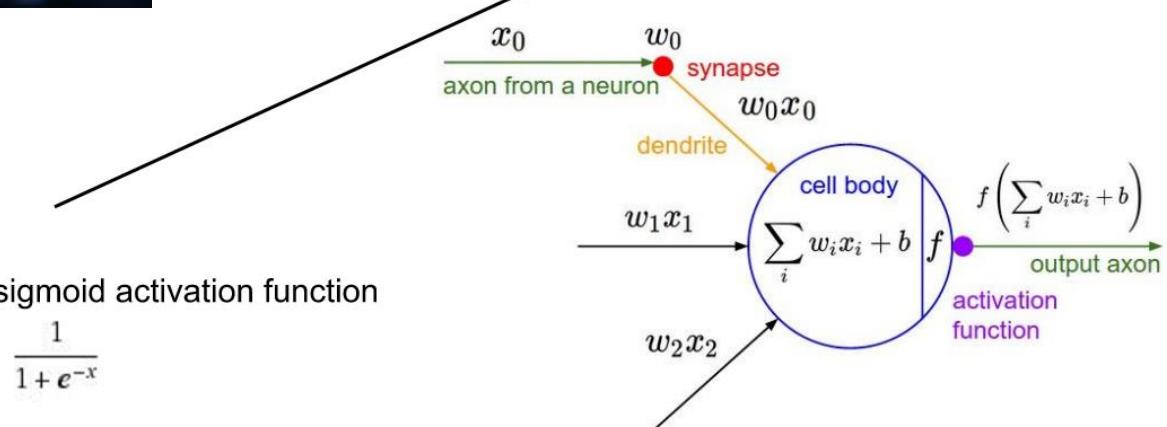
$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

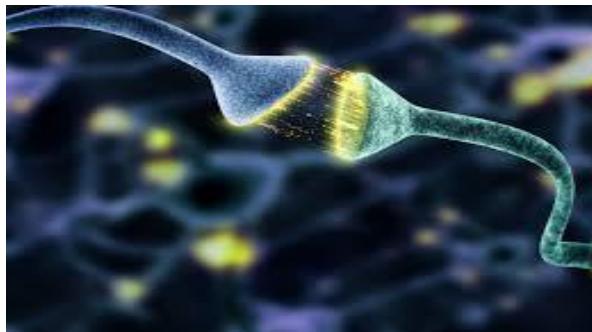
A: We end up with a linear classifier again!



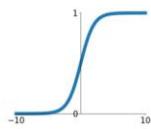
sigmoid activation function

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

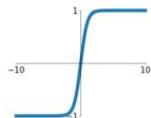




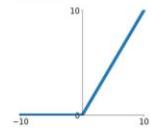
Sigmoid

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


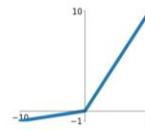
tanh

$$\tanh(x)$$


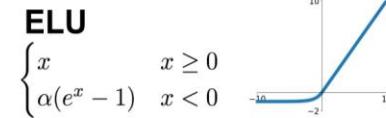
ReLU

$$\max(0, x)$$


Leaky ReLU

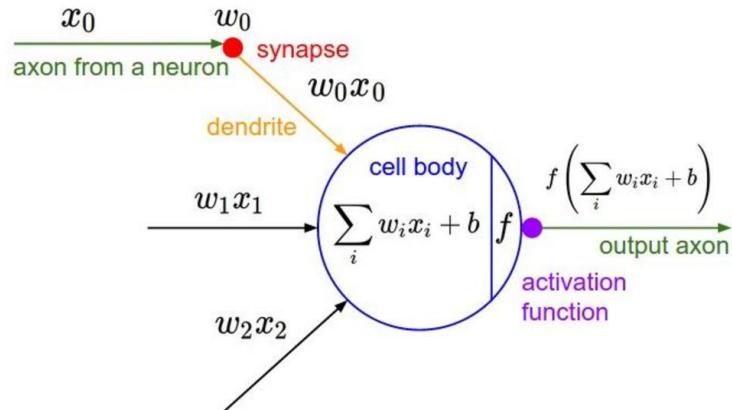
$$\max(0.1x, x)$$


Maxout

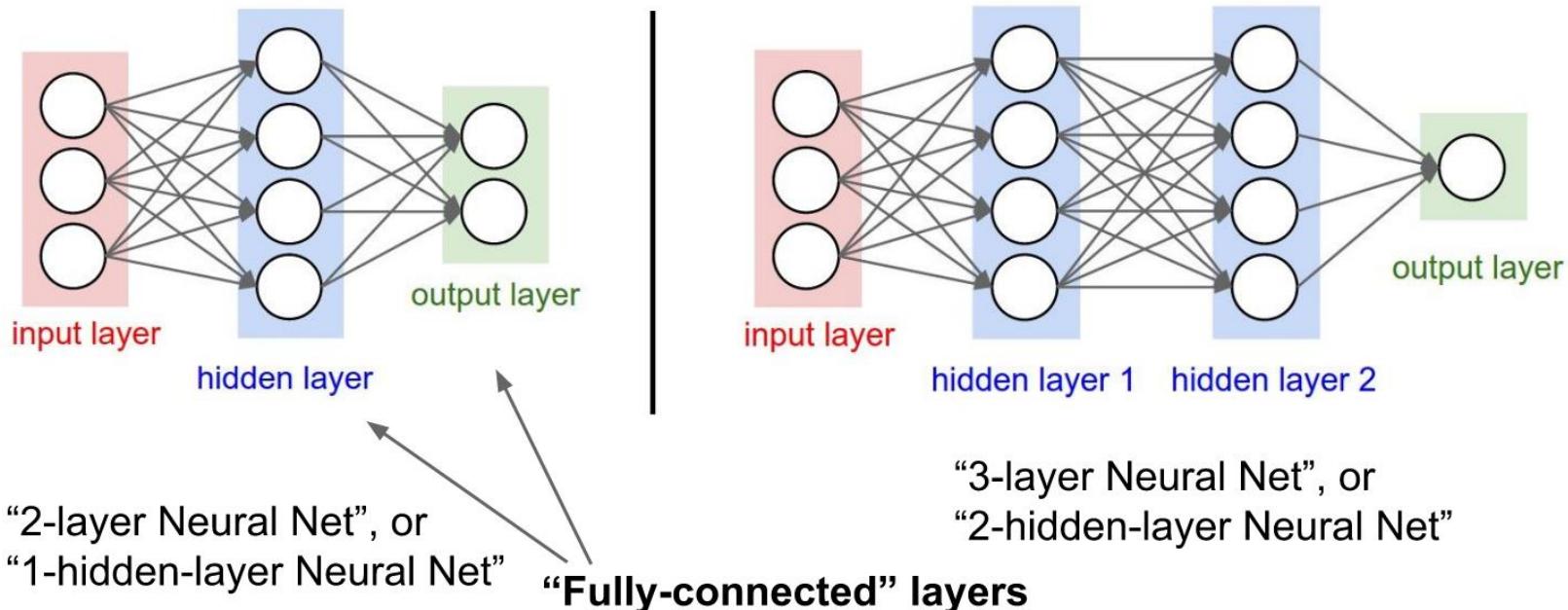
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$


ELU

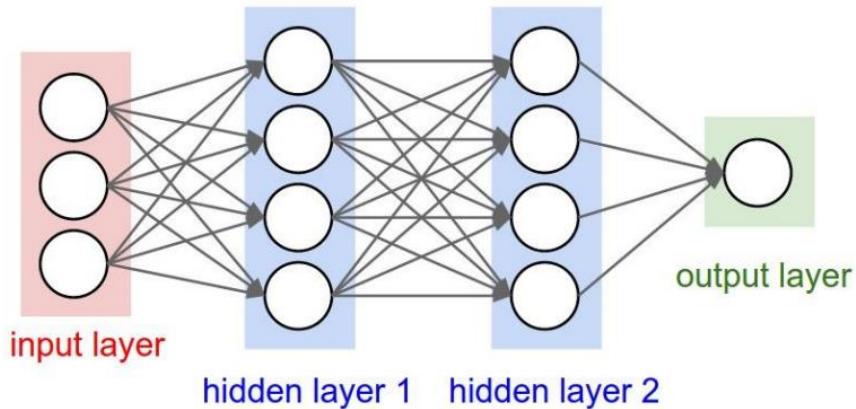
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural networks: Architectures



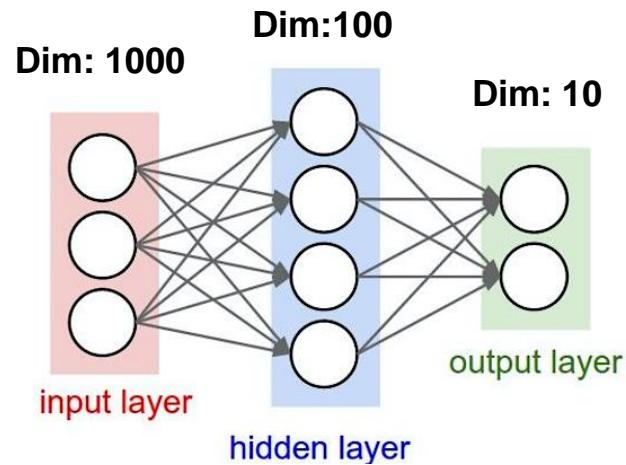
Example feed-forward computation of a neural network



```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```



Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

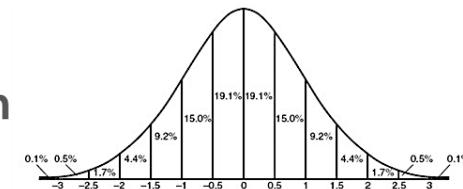
Define the network

numpy.random.randn

`numpy.random.randn(d0, d1, ..., dn)`

Return a sample (or samples) from the “standard normal” distribution.

If positive, int_like or int-convertible arguments are provided, `randn` generates an array of shape (d_0, d_1, \dots, d_n) , filled with random floats sampled from a univariate “normal” (Gaussian) distribution of mean 0 and variance 1 (if any of the d_i are floats, they are first converted to integers by truncation). A single float randomly sampled from the distribution is returned if no argument is provided.



Full implementation of training a 2-layer Neural Network needs ~20 lines:

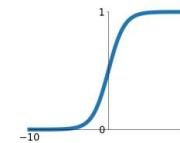
```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Sigmoid

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



$$SSE = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

predicted value actual value
test set

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

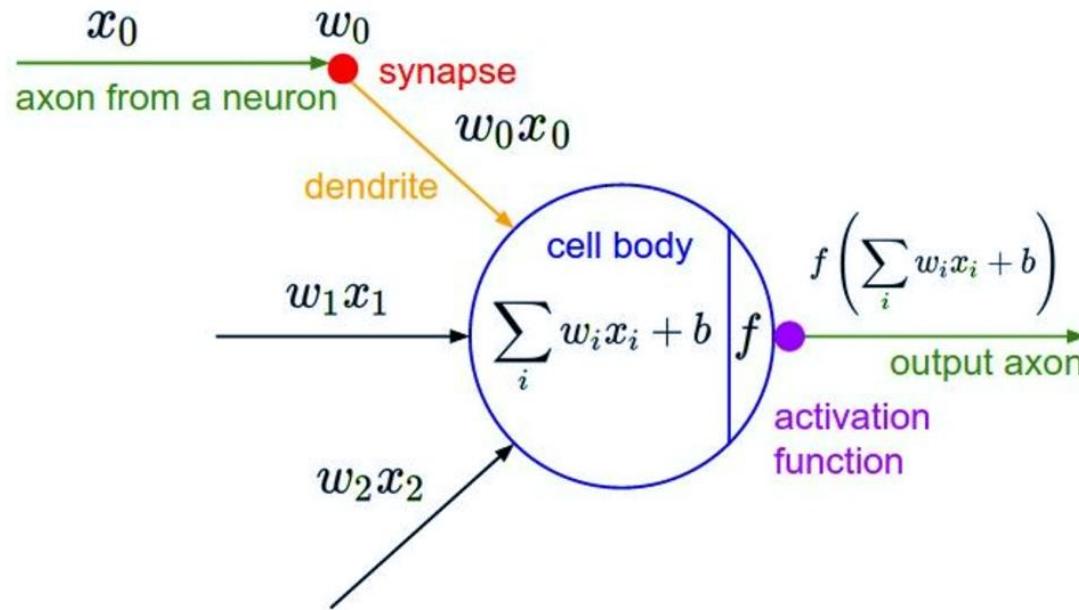
Calculate the analytical gradients

Gradient descent

Contents

- Traditional ML Approaches
- Introduction to Neural Networks
- **Activation Functions (Nonlinearity)**
- Gradient Descent
- Computational Graphs & Backpropagation

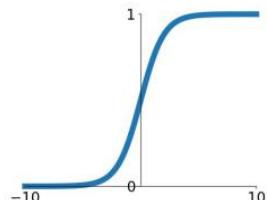
Activation Functions



Activation Functions

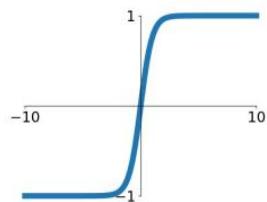
Sigmoid

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



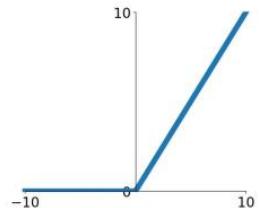
tanh

$$\tanh(x)$$



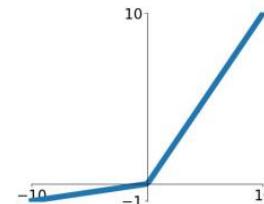
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

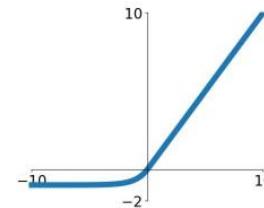


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

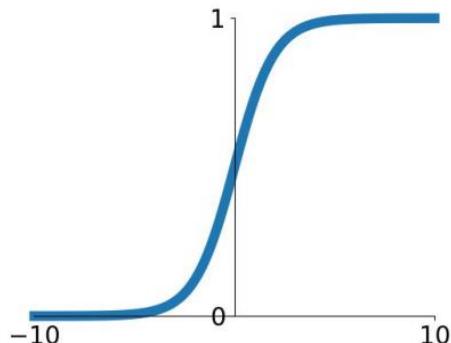
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

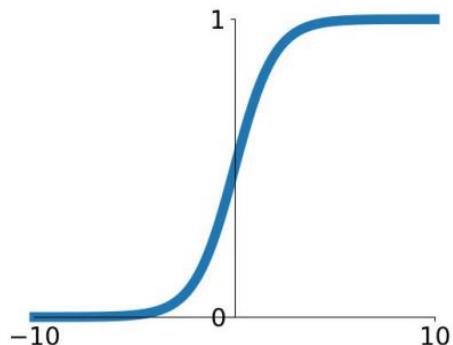
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



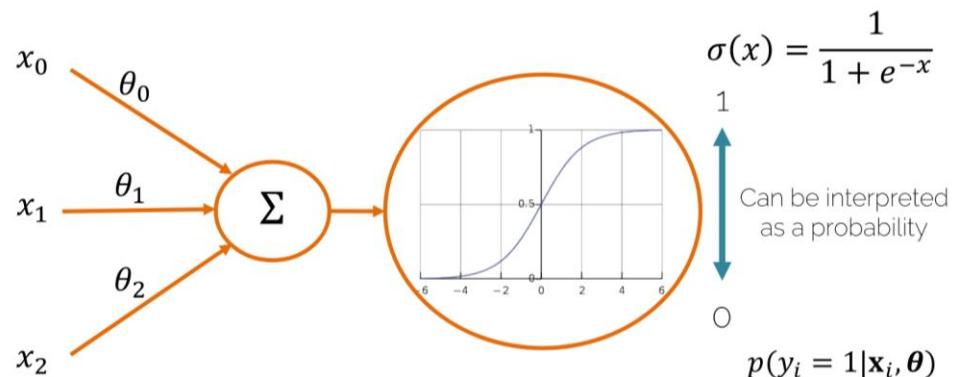
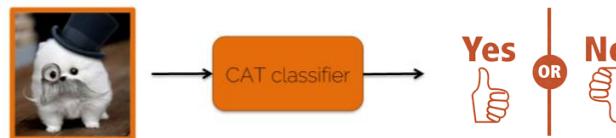
Sigmoid

Activation Functions



Sigmoid

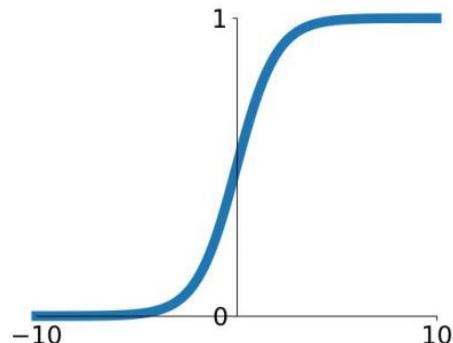
Logistic Regression



Activation Functions

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

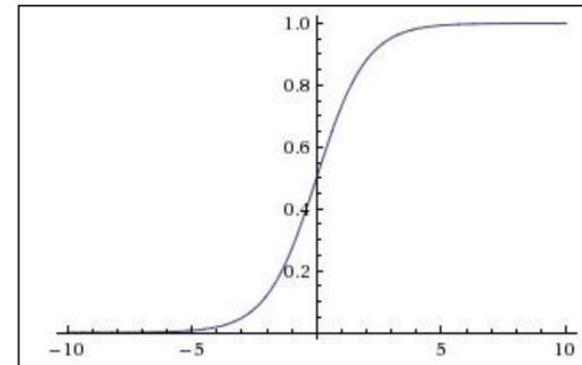
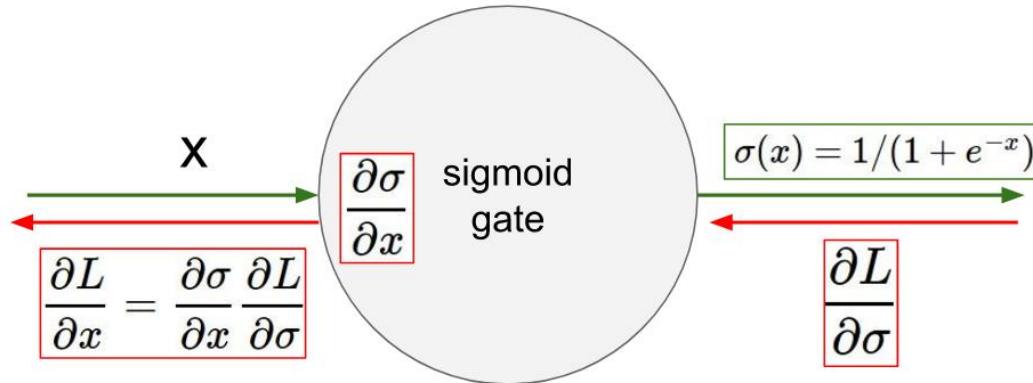
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



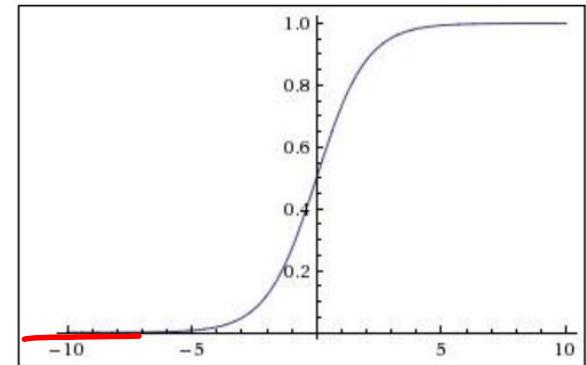
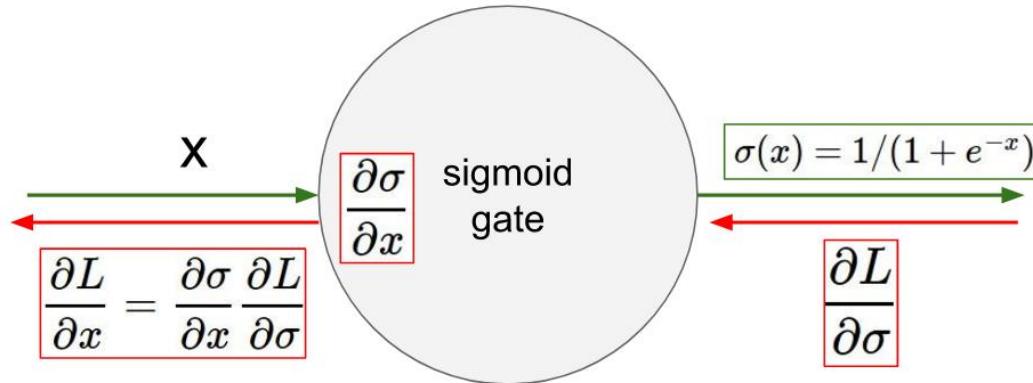
Sigmoid

3 problems:

1. Saturated neurons “kill” the gradients

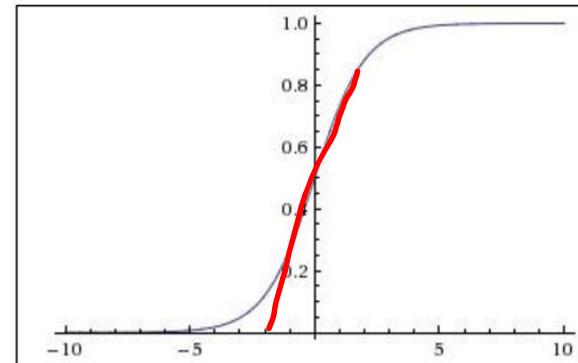
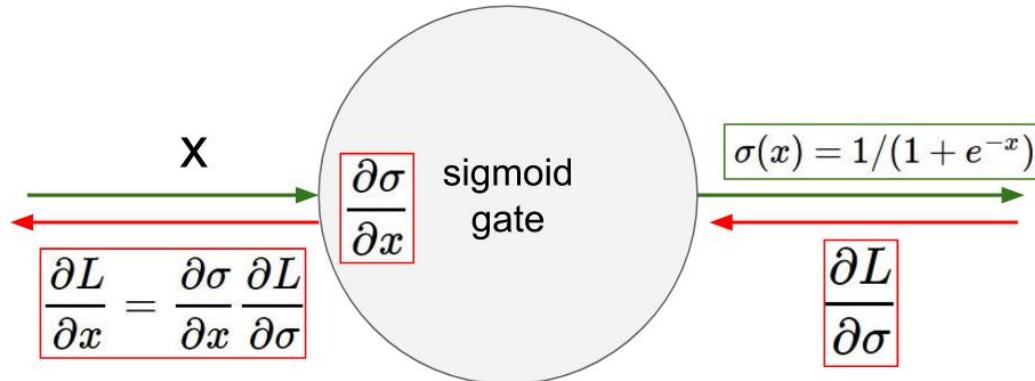


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

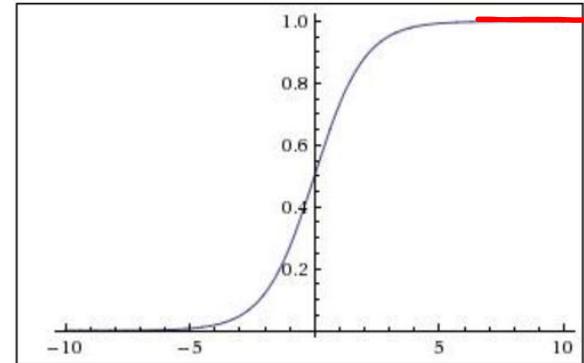
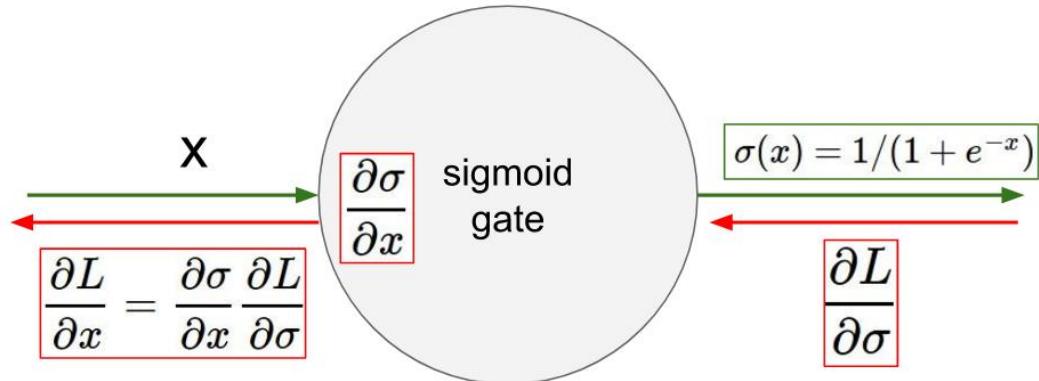
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

What happens when $x = 0$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

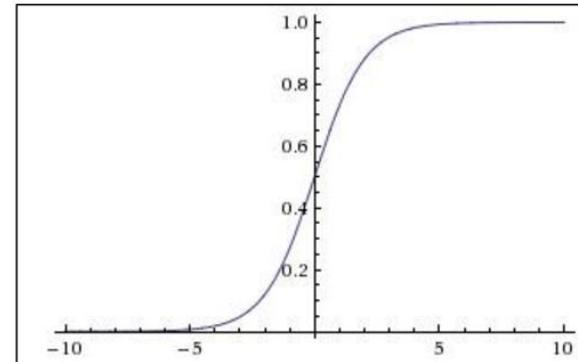
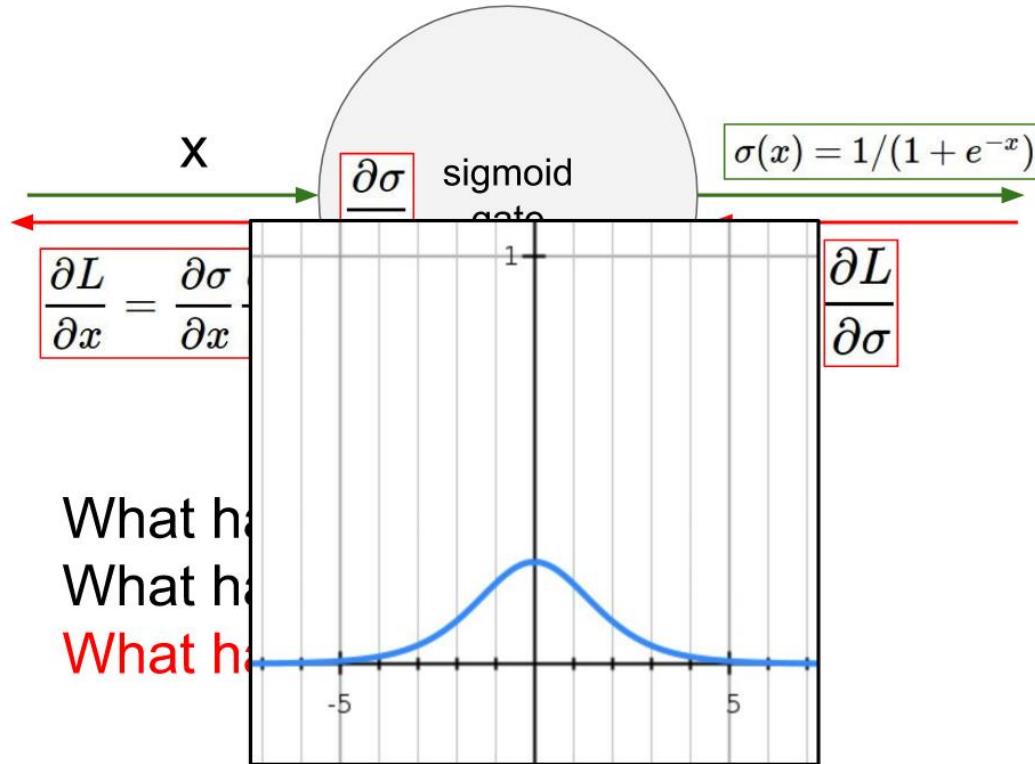


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

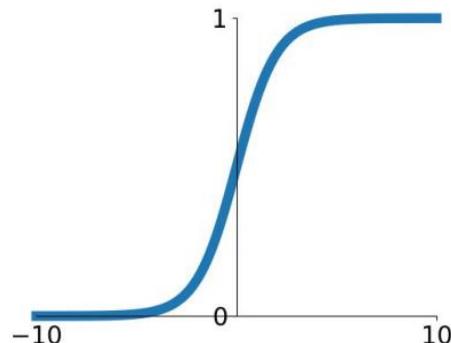


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

Activation Functions

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



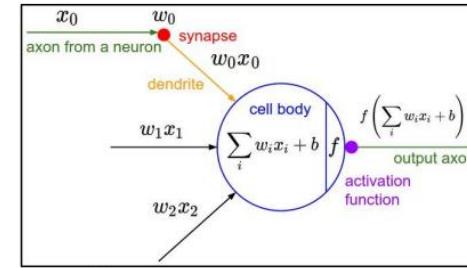
Sigmoid

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when the input to a neuron is always positive...

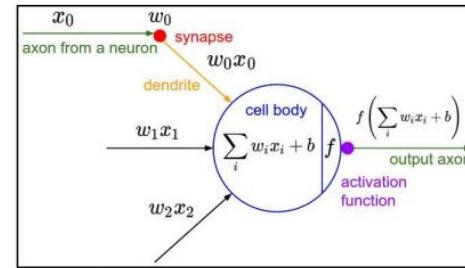
$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on w ?

Consider what happens when the input to a neuron is always positive...

$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on w ?

We know that local gradient of sigmoid is always positive

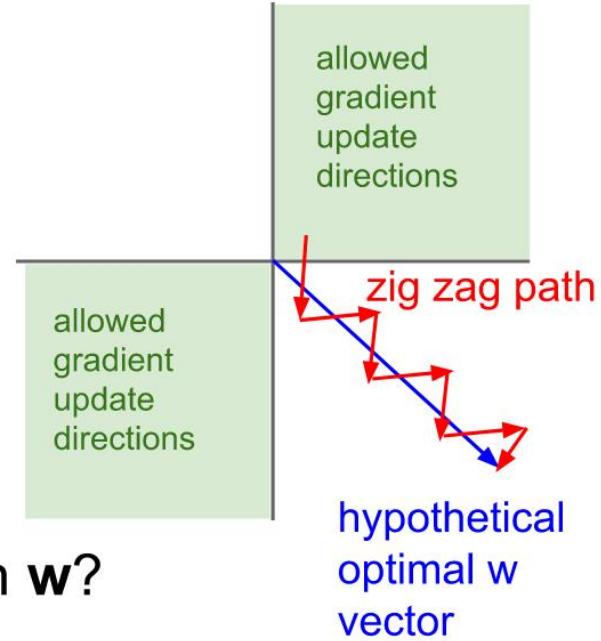
We are assuming x is always positive

So!! Sign of gradient for all w_i is the same as the sign of upstream scalar gradient!

$$\frac{\partial L}{\partial w} = \sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times \text{upstream_gradient}$$

Consider what happens when the input to a neuron is always positive...

$$f \left(\sum_i w_i x_i + b \right)$$

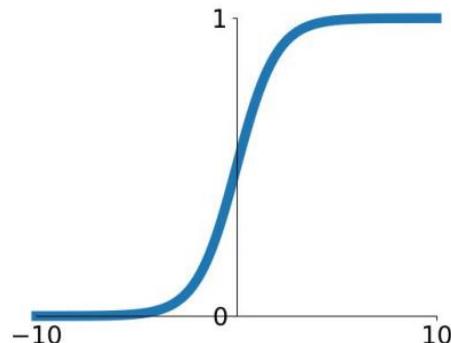


What can we say about the gradients on w ?
Always all positive or all negative :(

Activation Functions

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

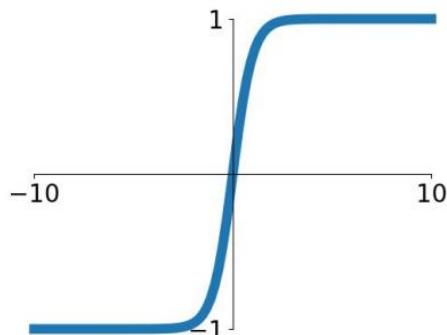


Sigmoid

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. $\exp()$ is a bit compute expensive

Activation Functions



- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

$$\tanh(x) = 2\sigma(2x) - 1$$

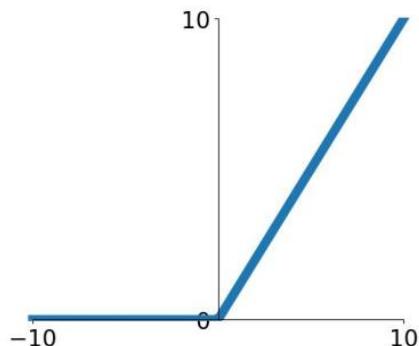
tanh(x)

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

$$\tanh'(x) = 1 - \tanh^2(x)$$

[LeCun et al., 1991]

Activation Functions

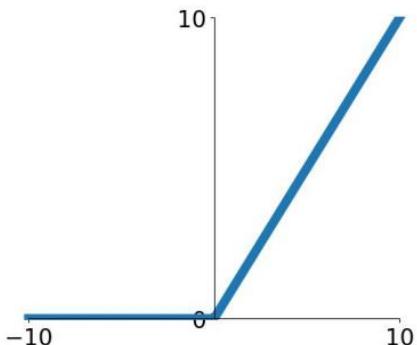


- Computes $f(x) = \max(0,x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

Activation Functions

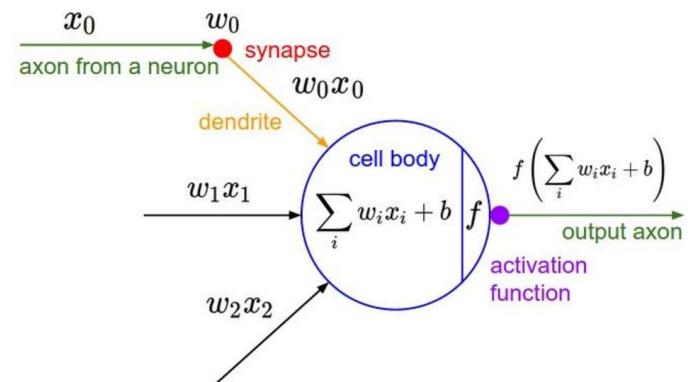
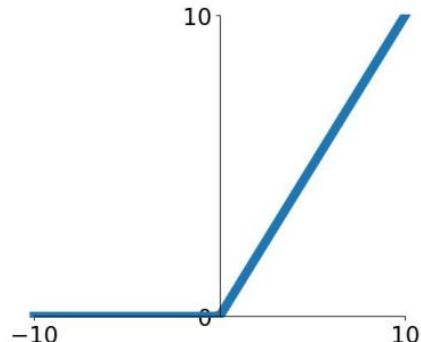
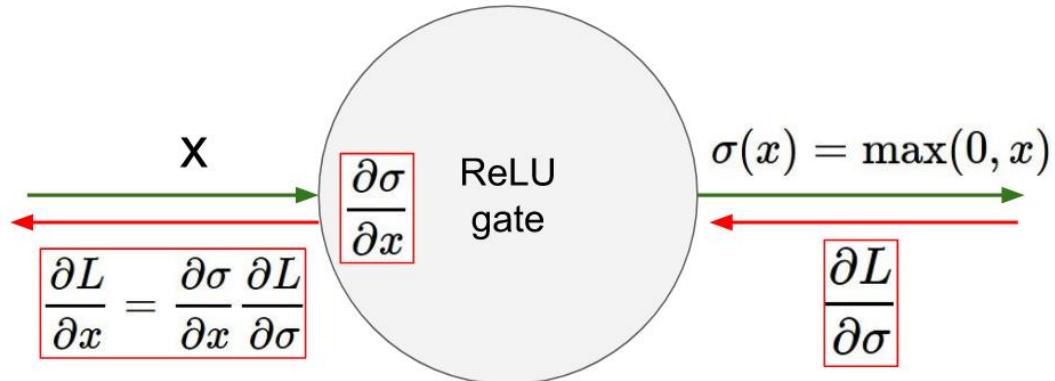


ReLU
(Rectified Linear Unit)

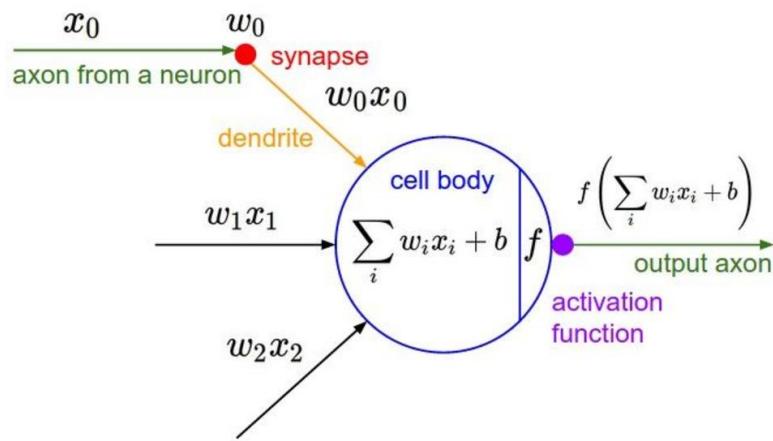
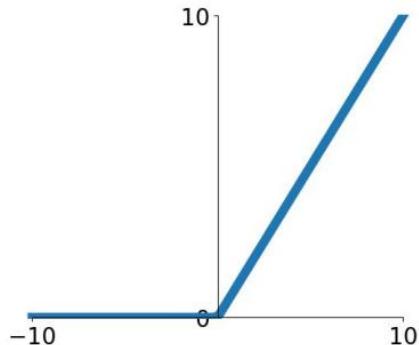
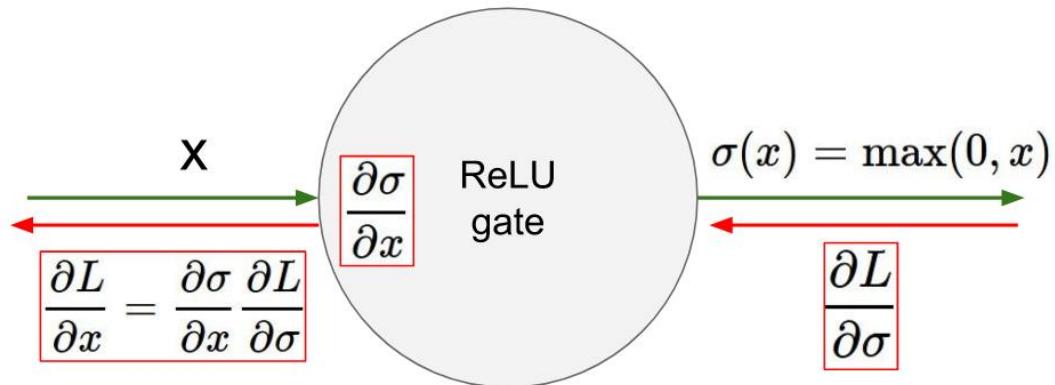
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

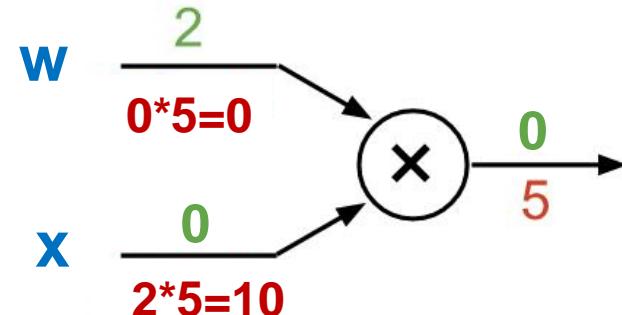
hint: what is the gradient when $x < 0$?



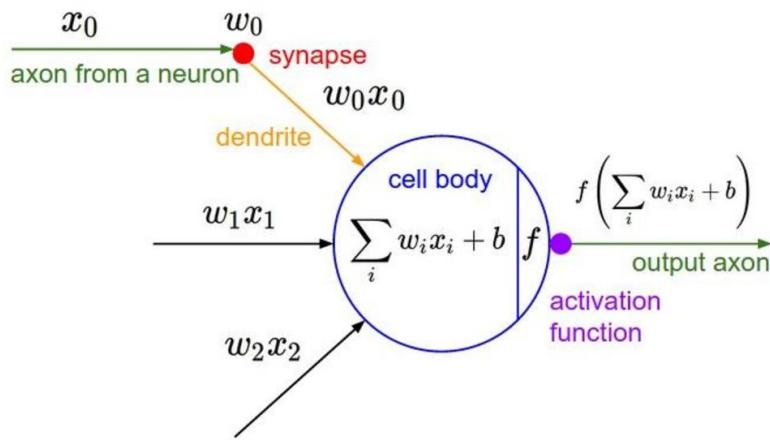
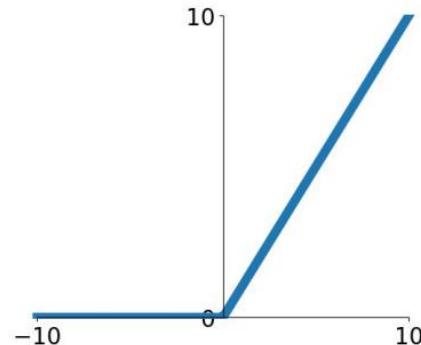
What happens when $x = -10$?
 What happens when $x = 0$?
 What happens when $x = 10$?



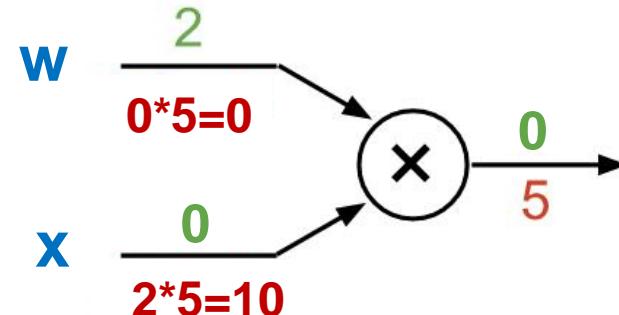
mul gate: “swap multiplier”



Initialize ReLU units with slightly positive biases (e.g., 0.001)



mul gate: “swap multiplier”



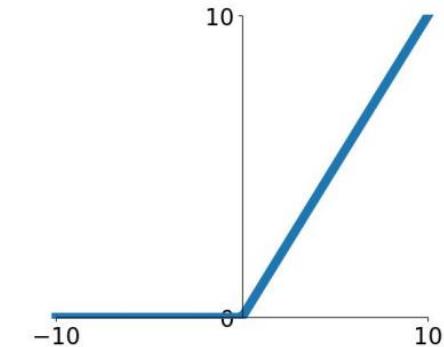
```

class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input

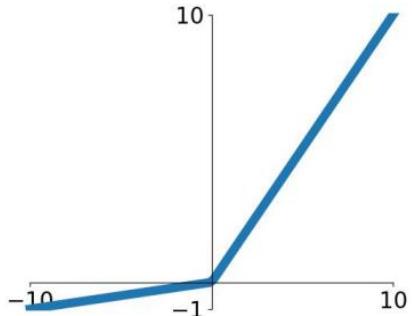
```



Computes $f(x) = \max(0, x)$

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



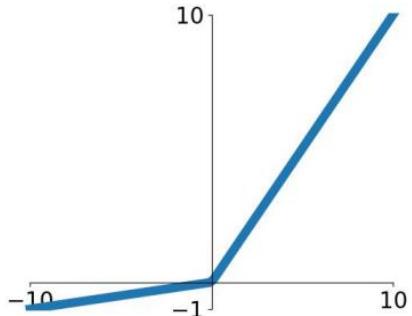
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

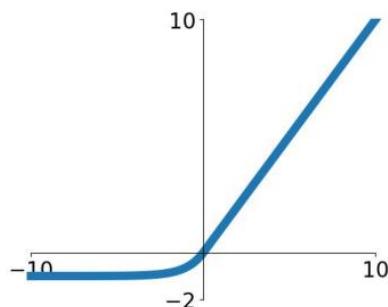
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into α
(parameter) learnable

Exponential Linear Units (ELU)



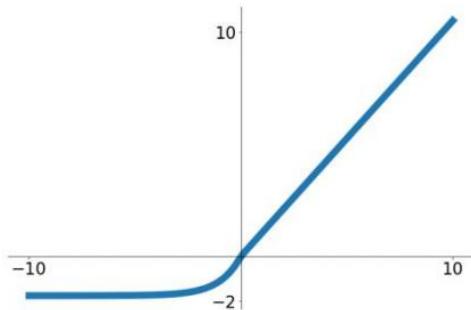
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(Alpha default = 1)

- Computation requires $\exp()$

Scaled Exponential Linear Units (SELU)



- Scaled version of ELU that works better for deep networks
- “Self-normalizing” property;
- Can train deep SELU networks without BatchNorm
 - (will discuss more later)

$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

$$\alpha = 1.6733, \lambda = 1.0507$$

Maxout “Neuron”

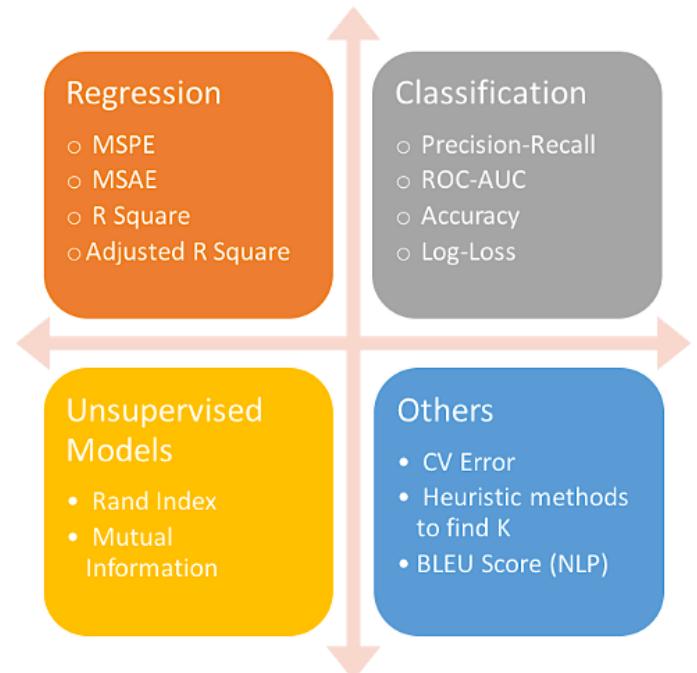
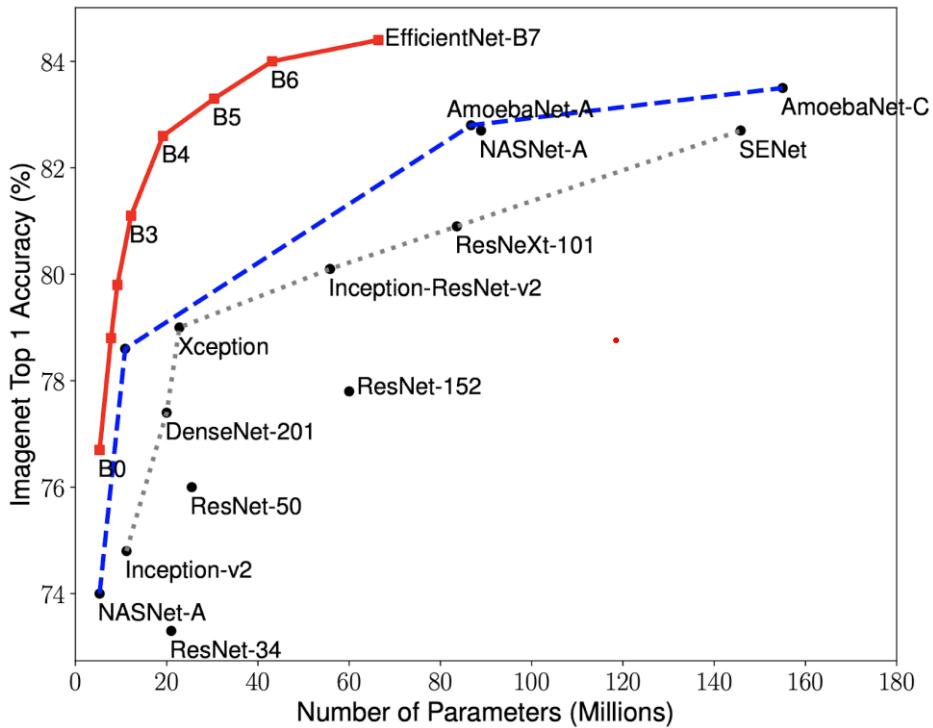
[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

of Parameters/FLOPs vs Evaluation Metric



TLDR: In practice:

- Use ReLU. Be careful about Dead ReLU Units
- Try out Leaky ReLU / Maxout / ELU / SELU
 - To squeeze out some marginal gains
- Don't use sigmoid or tanh

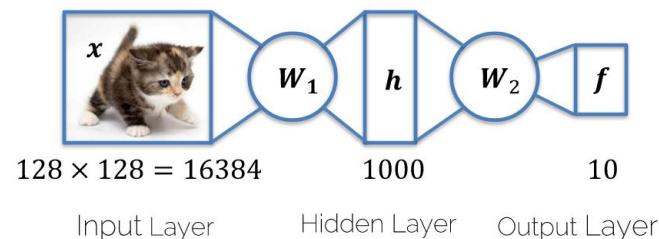
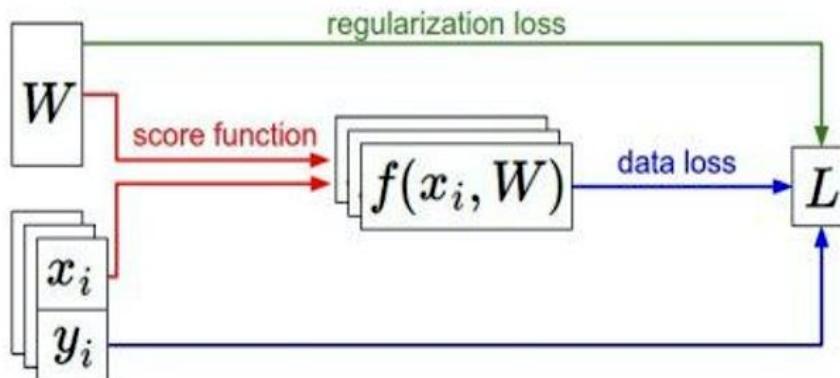
Contents

- Traditional ML Approaches
- Introduction to Neural Networks
- Activation Functions (Nonlinearity)
- **Gradient Descent**
- Computational Graphs & Backpropagation

Recap

How do we find the best W ?

- We have some dataset of (x, y)
- We have a **score function**: $s = f(x; W)$
- We have a **loss function**:

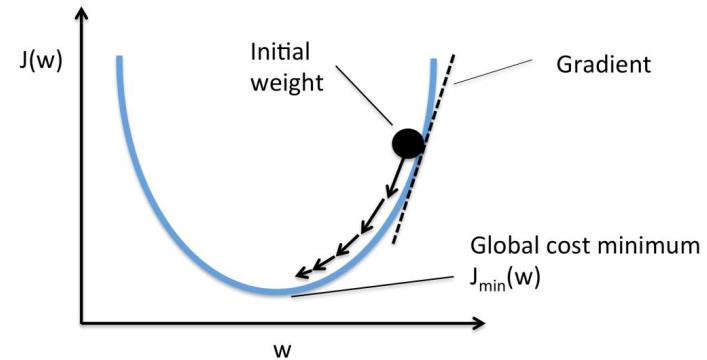


Recap

- We have some dataset of (x, y)
- We have a **score function**: $s = f(x; W)$
- We have a **loss function**:



How do we find the best W ?



$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w},$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} \quad J(\mathbf{w}) = \frac{1}{2} \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

Gradient

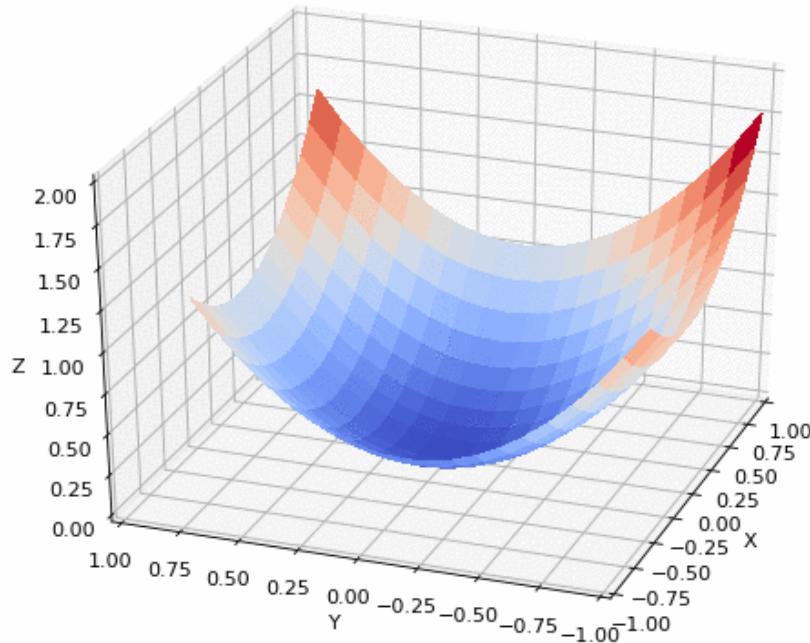
Gradient

$$\nabla f: \mathbb{R}^n \rightarrow \mathbb{R}$$

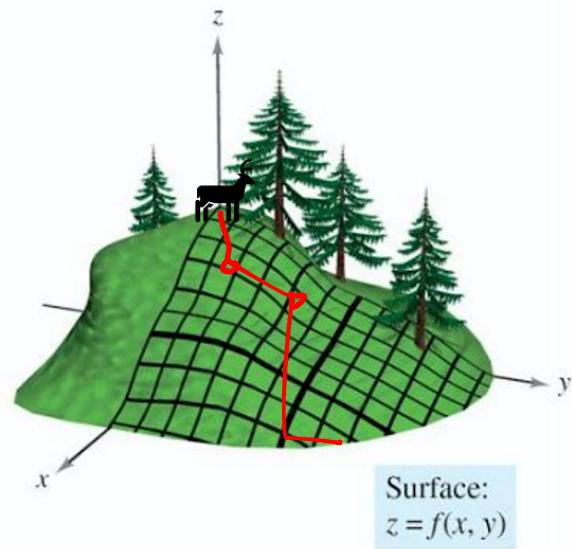
Partial derivative

$$x \rightarrow \nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \dots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix}$$

8



Gradient



$$f(x, y) = 3x^2y \quad \nabla f(x, y) = \left[\frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y} \right]$$

$$\frac{\partial}{\partial x} 3yx^2 = 3y \frac{\partial}{\partial x} x^2 = 3y2x = 6yx$$

$$\frac{\partial}{\partial y} 3x^2y = 3x^2 \frac{\partial}{\partial y} y = 3x^2 \frac{\partial y}{\partial y} = 3x^2 \times 1 = 3x^2$$

$$\nabla f(x, y) = \left[\frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y} \right] = [6yx, 3x^2]$$

Images from: <https://www.zhihu.com/question/36301367>

11

Problem: How to compute gradients?

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM Loss on predictions}$$

$$R(W) = \sum_k W_k^2 \quad \text{Regularization}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

If we can compute $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$ then we can learn W_1 and W_2

(Bad) Idea: Derive $\nabla_W L$ on paper

$$s = f(x; W) = Wx$$

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

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

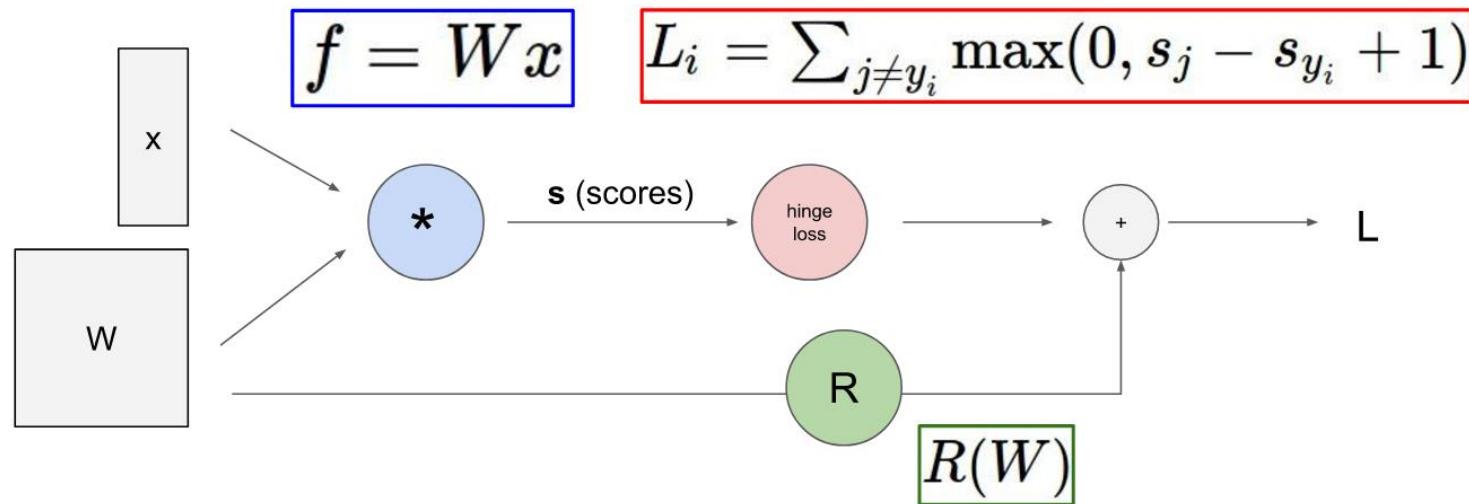
Problem: What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch =(

Problem: Not feasible for very complex models!

Contents

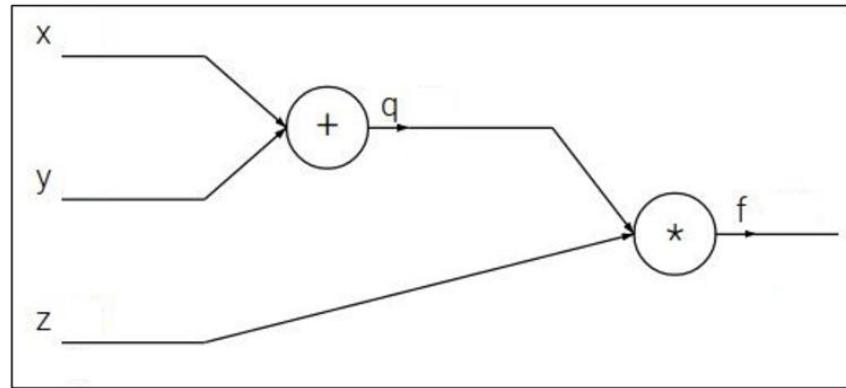
- Traditional ML Approaches
- Introduction to Neural Networks
- Activation Functions (Nonlinearity)
- Gradient Descent
- Computational Graphs & Backpropagation

Better Idea: Computational graphs + Backpropagation



Backpropagation: a simple example

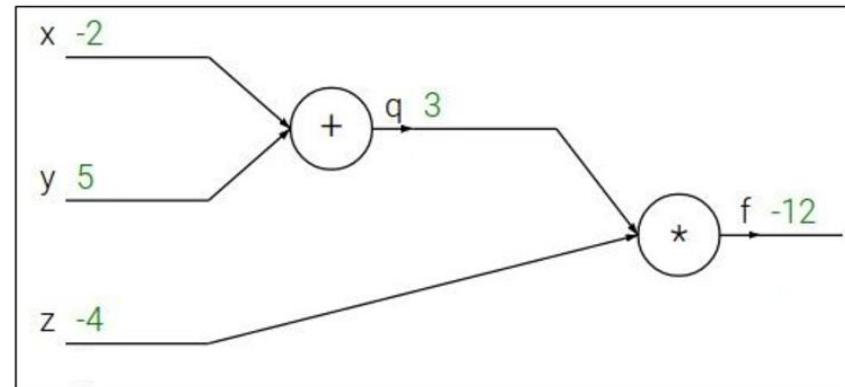
$$f(x, y, z) = (x + y)z$$



Backpropagation: a simple example

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

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

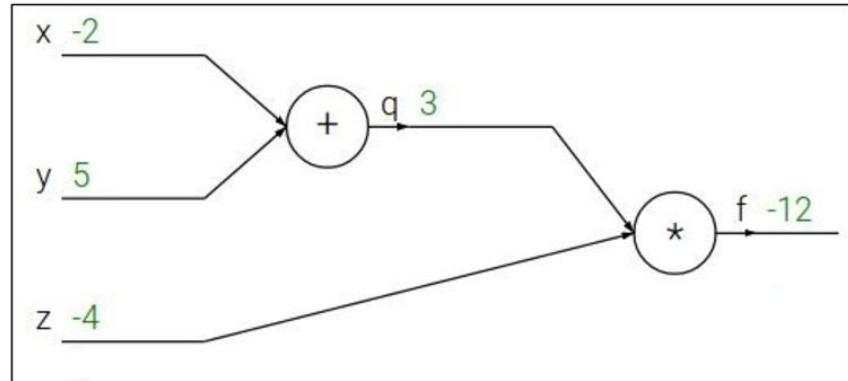


Backpropagation: a simple example

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

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

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

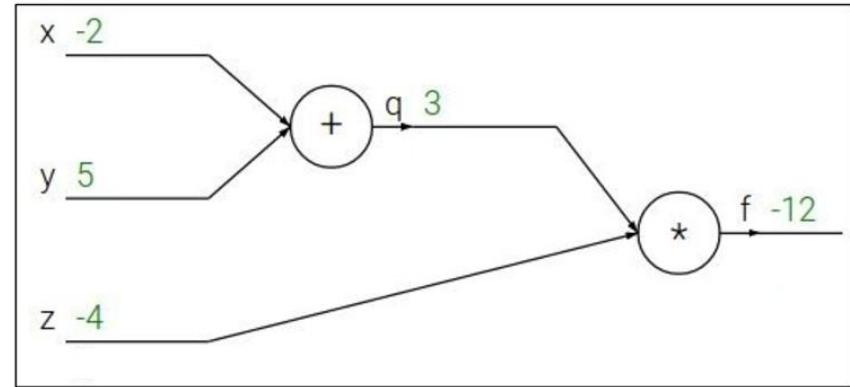


Backpropagation: a simple example

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

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

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



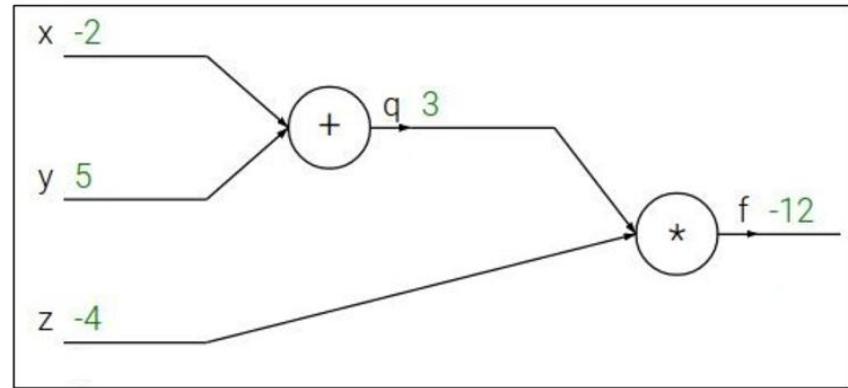
$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

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

Backpropagation: a simple example

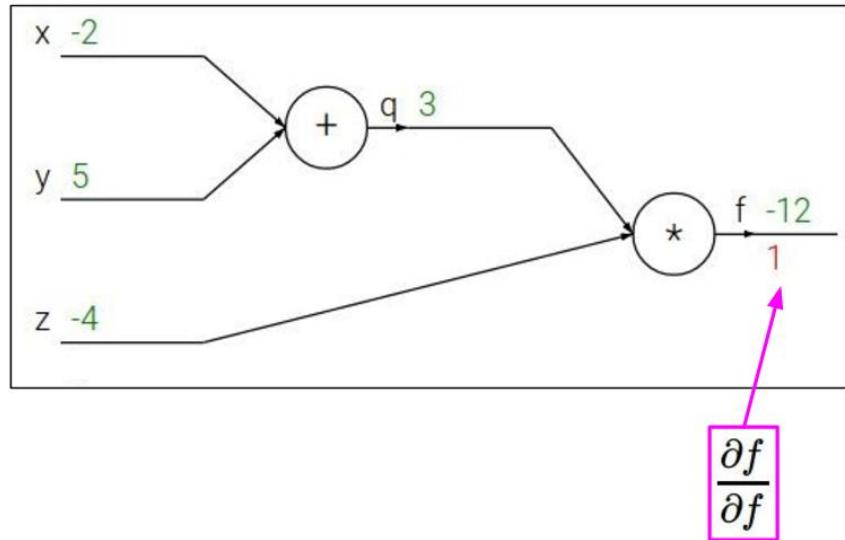
$$f(x, y, z) = (x + y)z$$

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



Backpropagation: a simple example

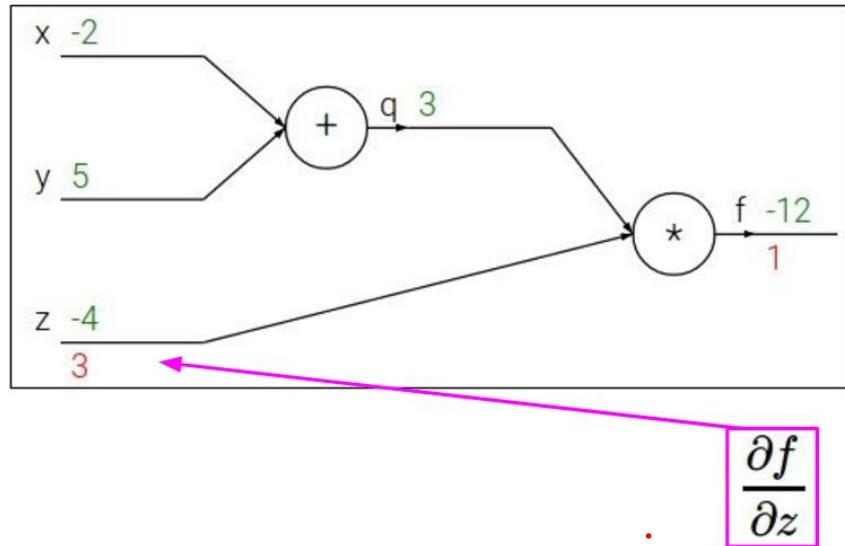
$$f(x, y, z) = (x + y)z$$

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

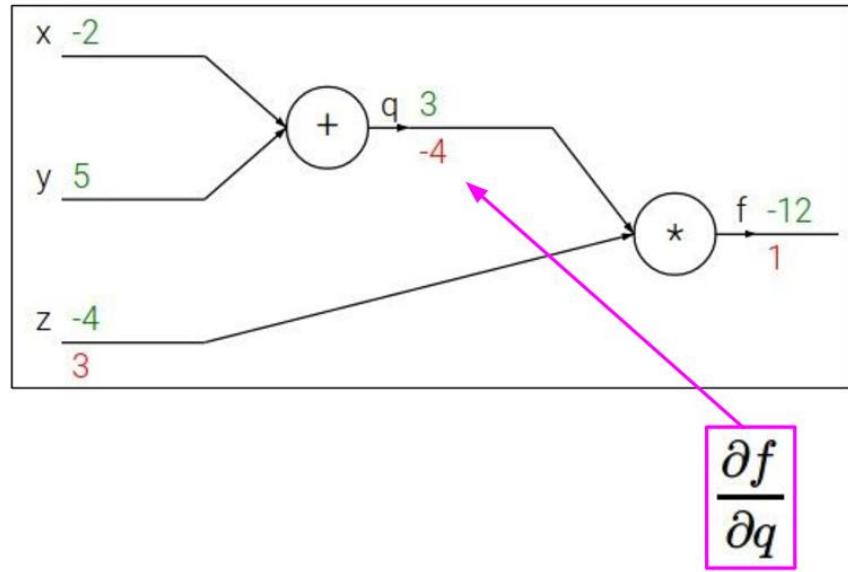
$$f(x, y, z) = (x + y)z$$

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



Backpropagation: a simple example

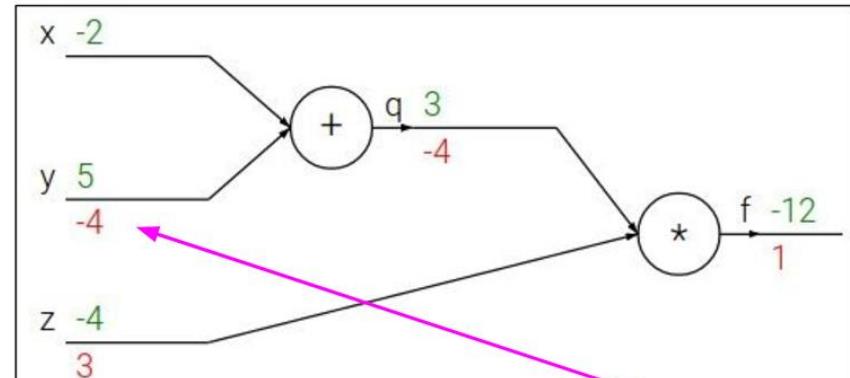
$$f(x, y, z) = (x + y)z$$

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

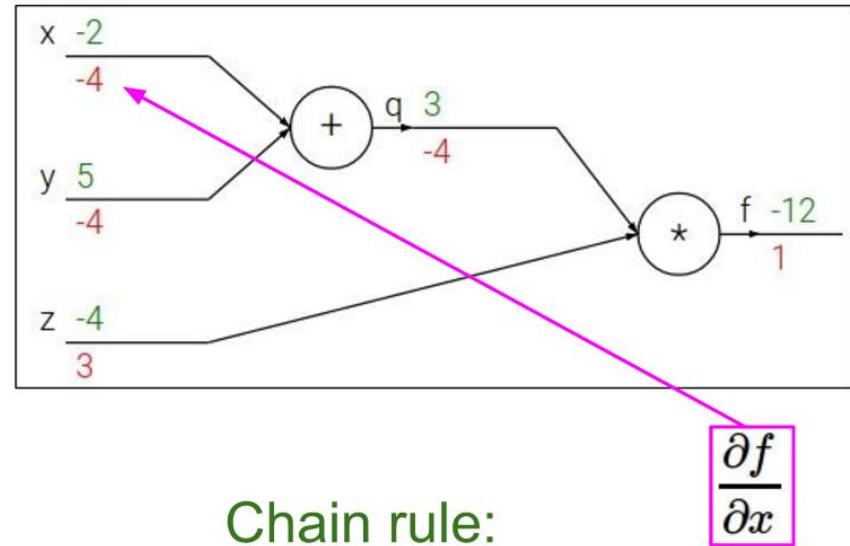
$$f(x, y, z) = (x + y)z$$

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



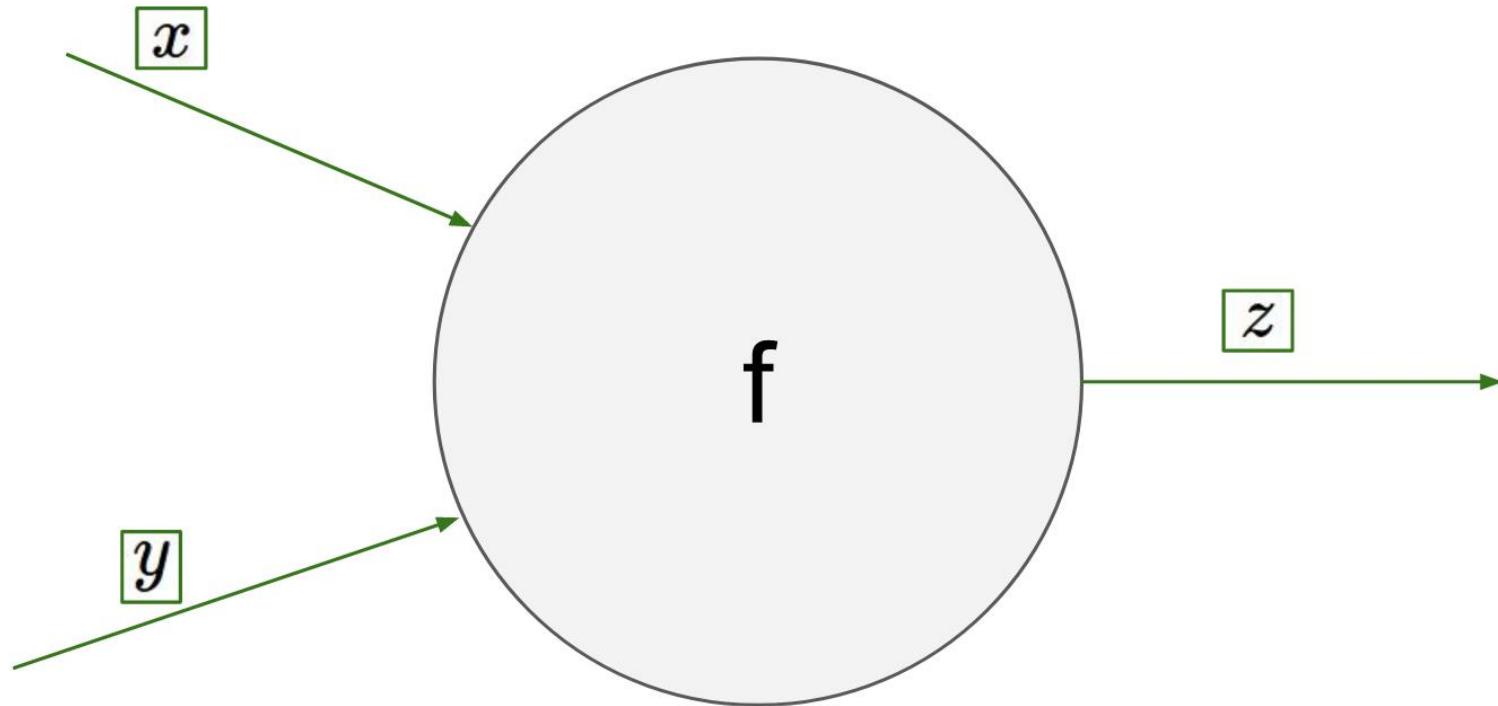
Chain rule:

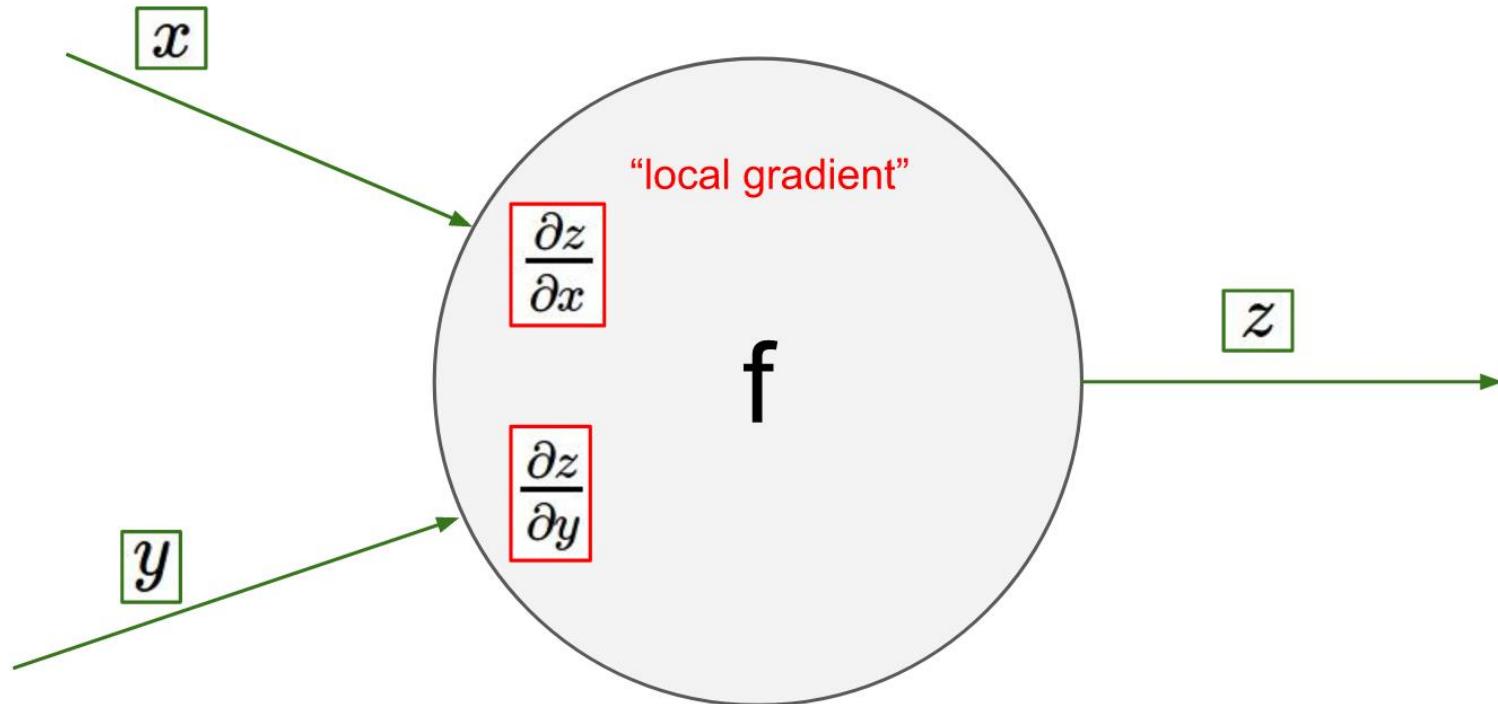
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

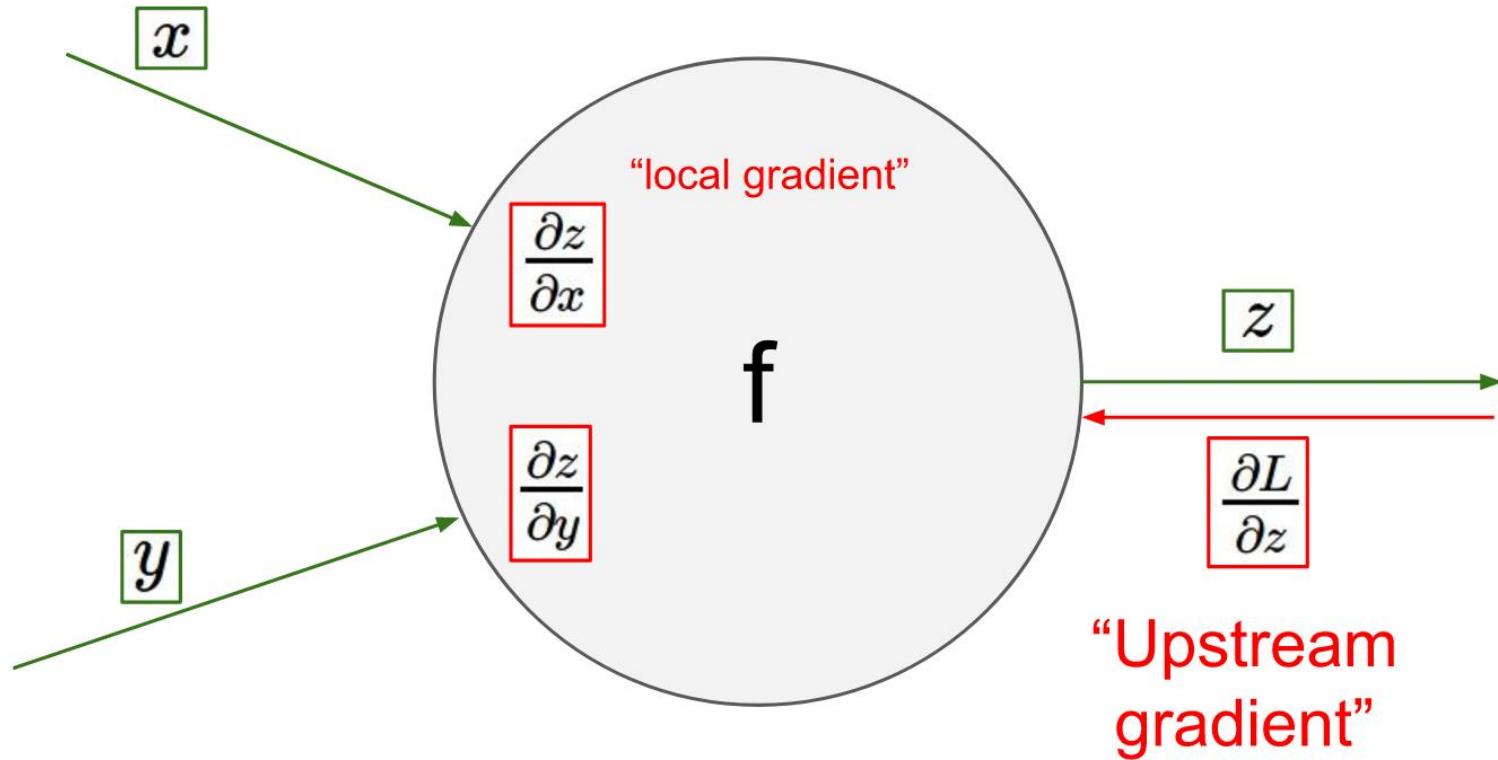
Upstream
gradient

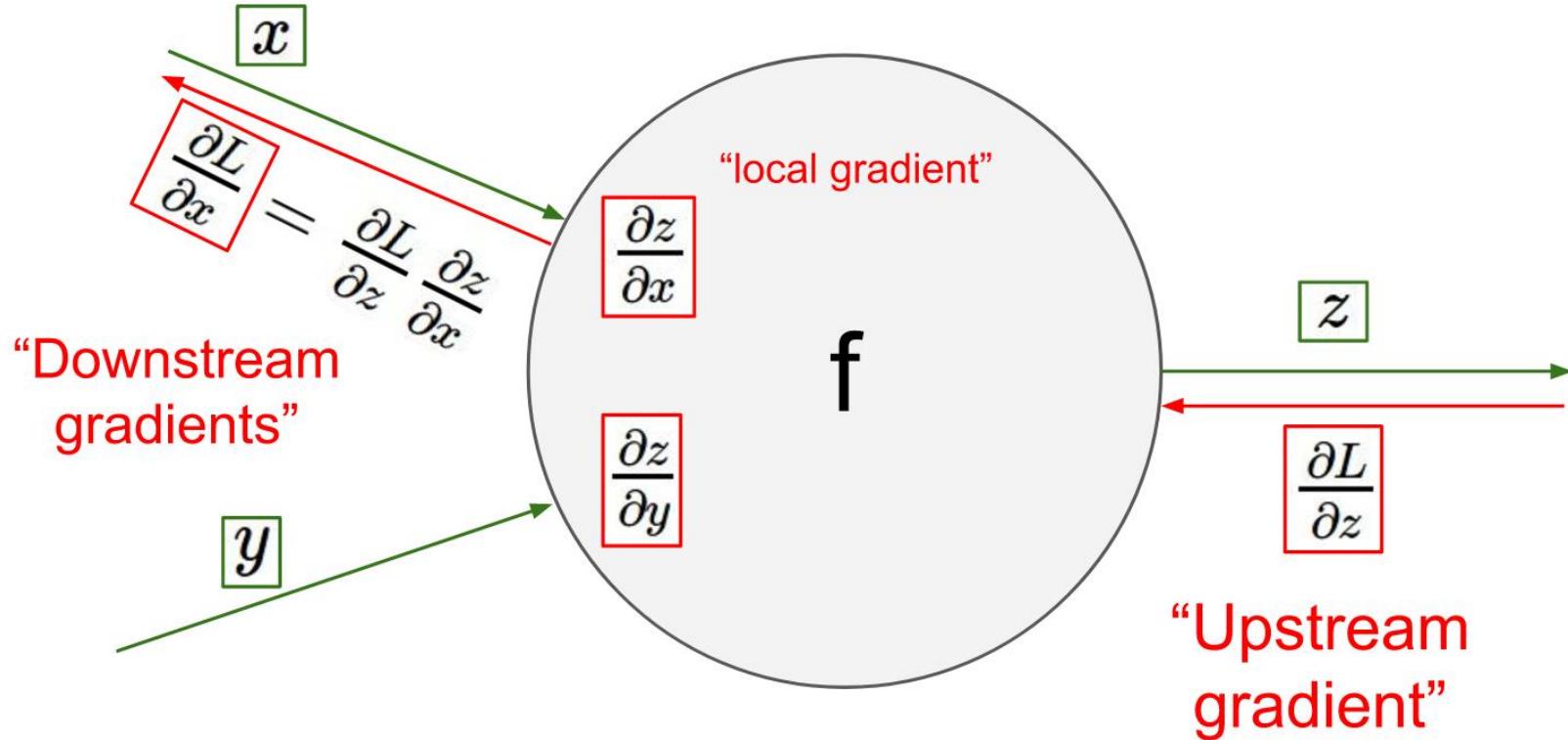
Local
gradient

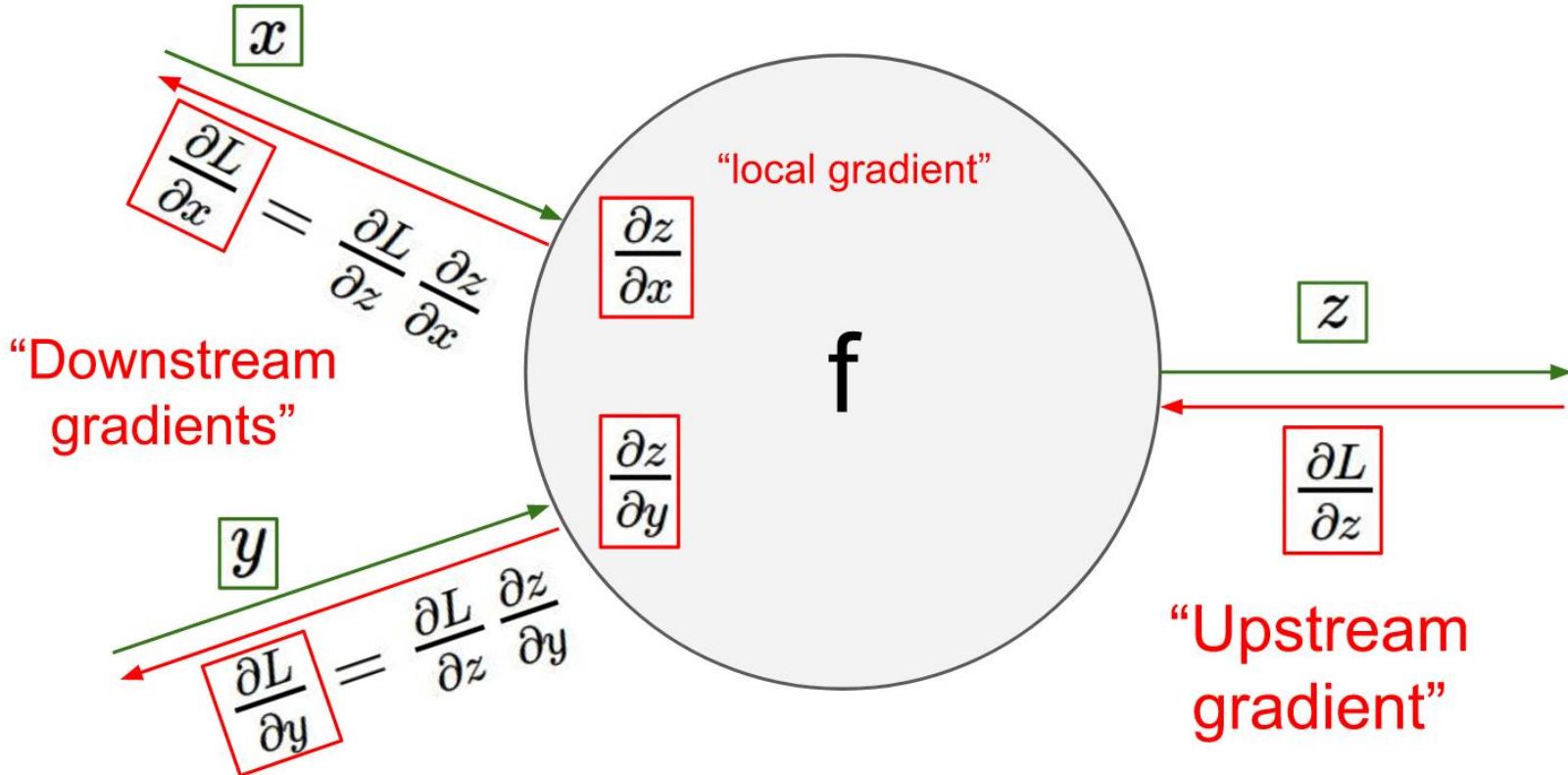
$$\frac{\partial f}{\partial x}$$

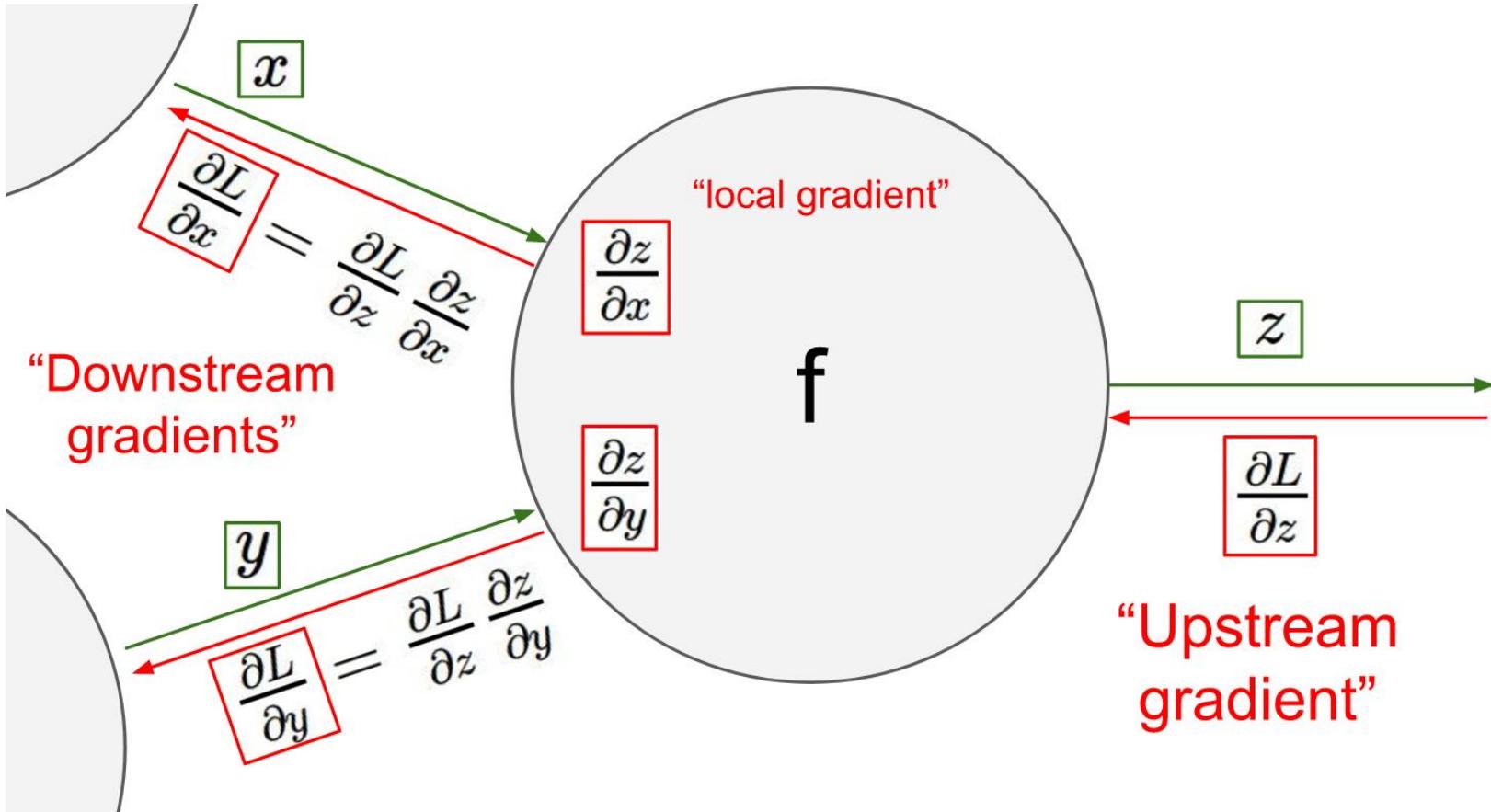






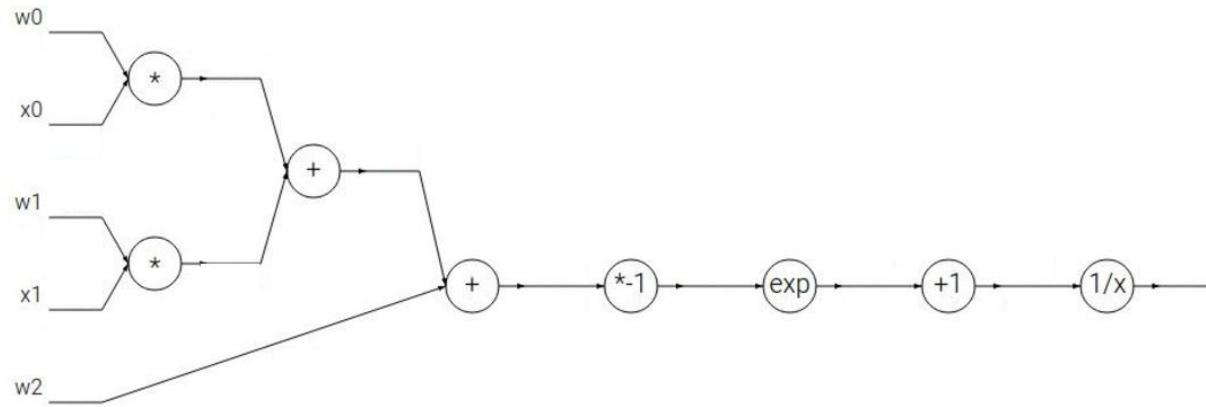






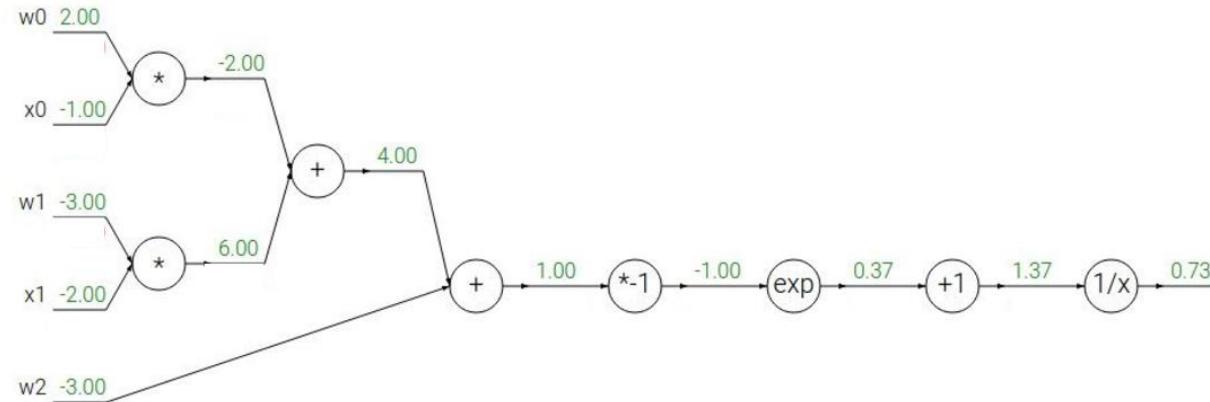
Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

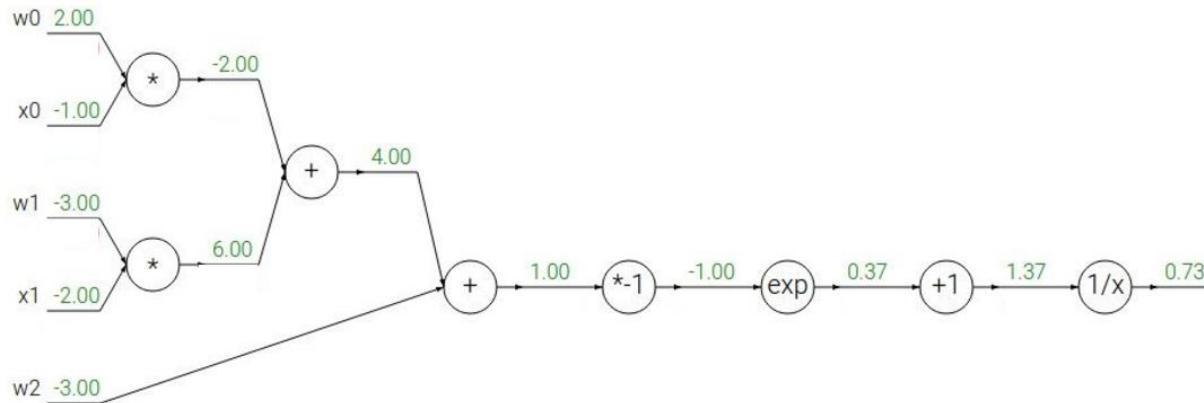


Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

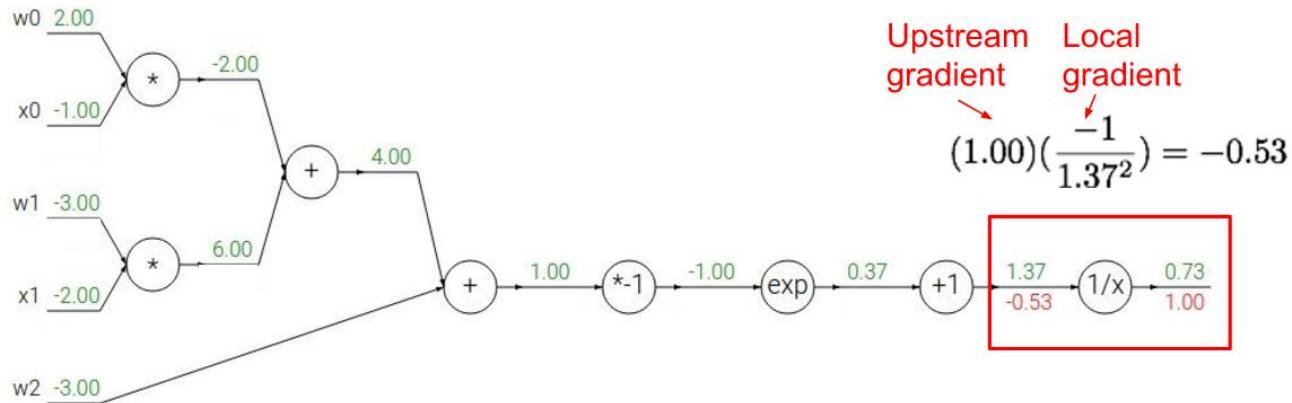
$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

Another example:

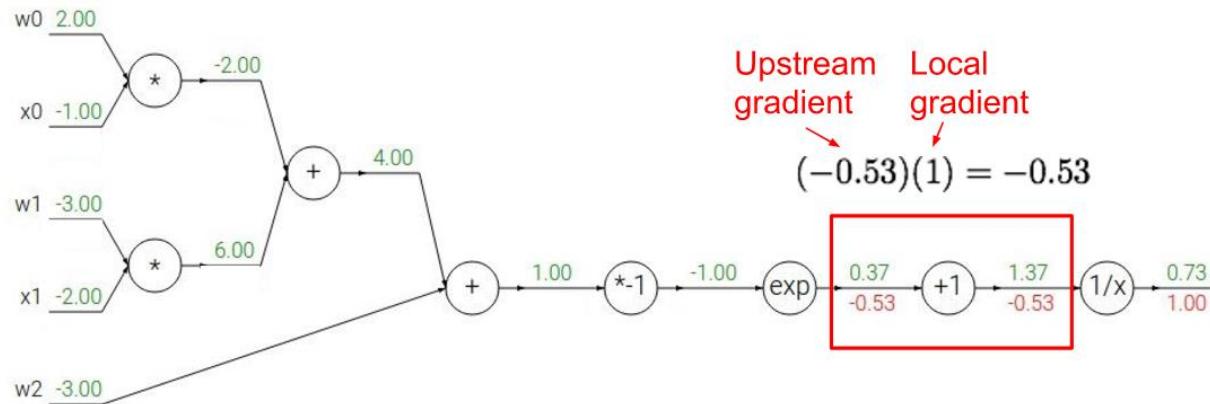
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$\begin{aligned} f(x) &= e^x & \rightarrow & \quad \frac{df}{dx} = e^x \\ f_a(x) &= ax & \rightarrow & \quad \frac{df}{dx} = a \end{aligned}$$

$$\begin{array}{c|c} \boxed{\begin{aligned} f(x) &= \frac{1}{x} & \rightarrow & \quad \frac{df}{dx} = -1/x^2 \\ f_c(x) &= c + x & \rightarrow & \quad \frac{df}{dx} = 1 \end{aligned}} & \end{array}$$

Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



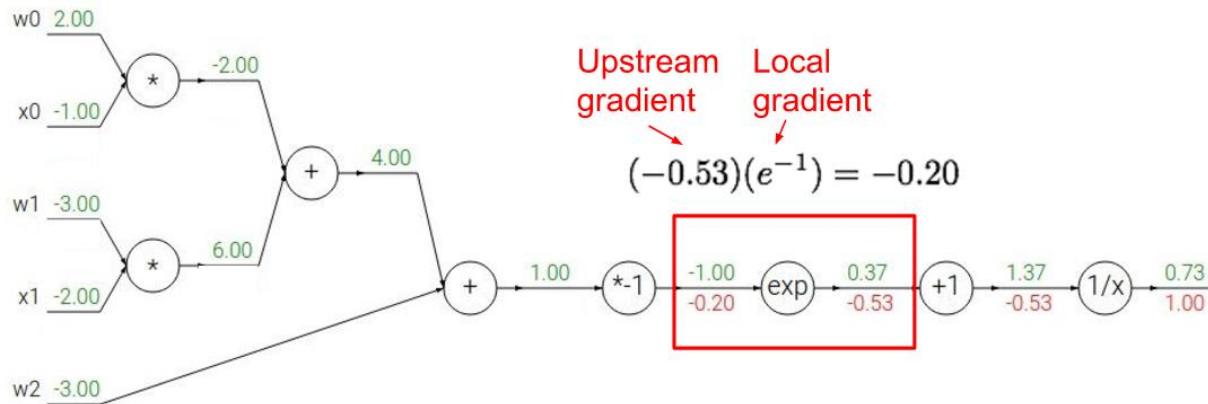
$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



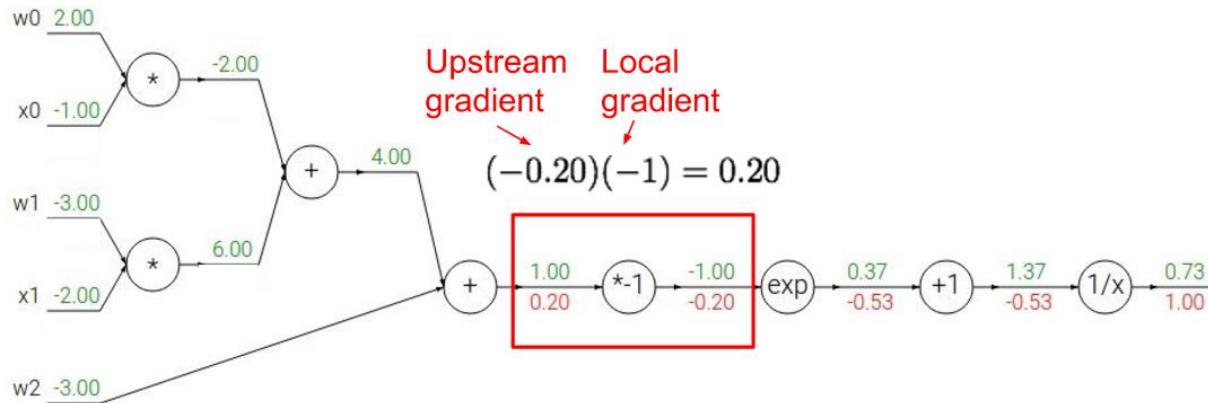
$$\boxed{f(x) = e^x} \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$

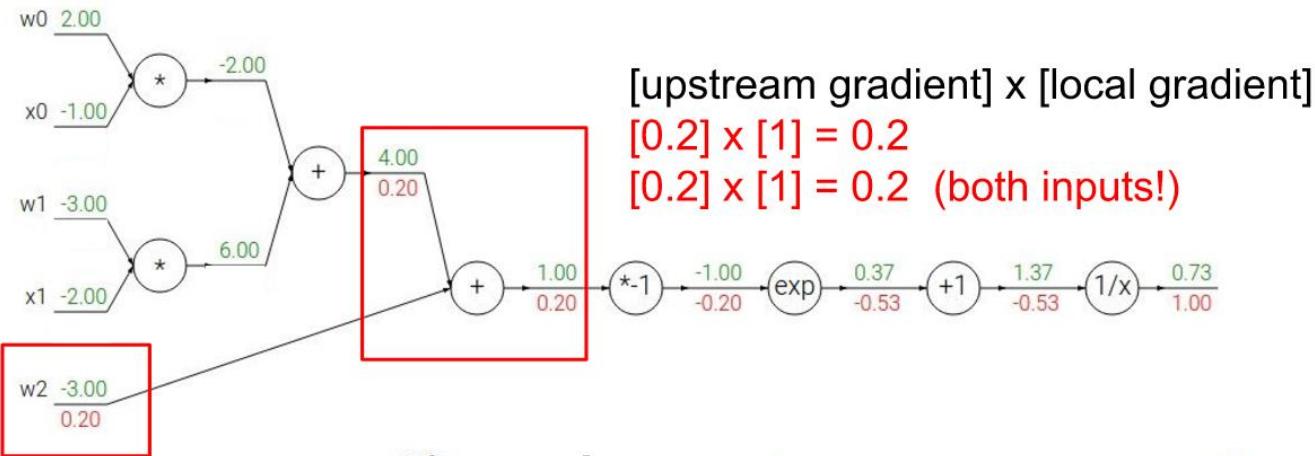


$$\begin{aligned} f(x) &= e^x & \rightarrow & \frac{df}{dx} = e^x \\ f_a(x) &= ax & \rightarrow & \frac{df}{dx} = a \end{aligned}$$

$$\begin{aligned} f(x) &= \frac{1}{x} & \rightarrow & \frac{df}{dx} = -1/x^2 \\ f_c(x) &= c + x & \rightarrow & \frac{df}{dx} = 1 \end{aligned}$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

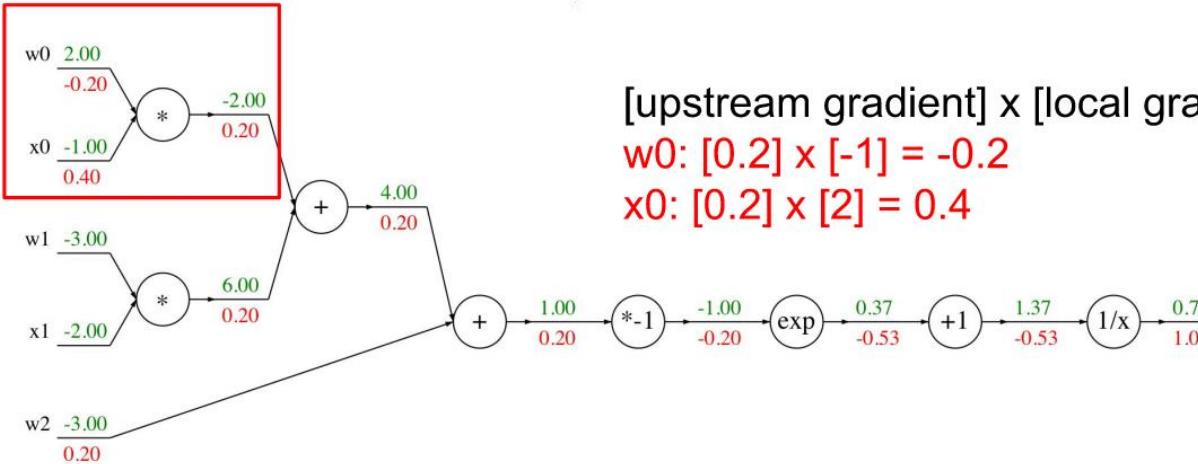
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



[upstream gradient] x [local gradient]
 $w_0: [0.2] \times [-1] = -0.2$
 $x_0: [0.2] \times [2] = 0.4$

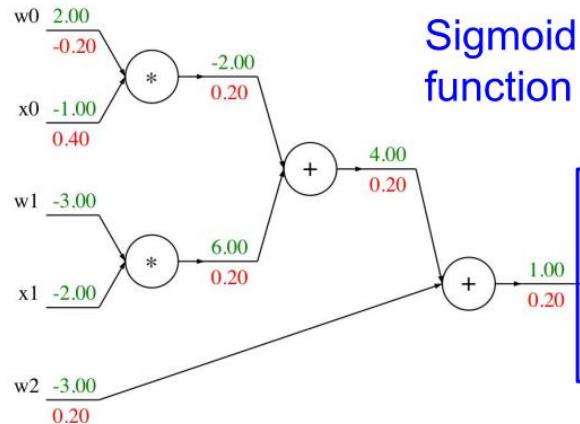
$$\begin{aligned} f(x) &= e^x & \rightarrow & \frac{df}{dx} = e^x \\ f_a(x) &= ax & \rightarrow & \frac{df}{dx} = a \end{aligned}$$

$$\frac{df}{dx} = e^x$$

$$\begin{aligned} f(x) &= \frac{1}{x} & \rightarrow & \frac{df}{dx} = -1/x^2 \\ f_c(x) &= c + x & \rightarrow & \frac{df}{dx} = 1 \end{aligned}$$

Another example:

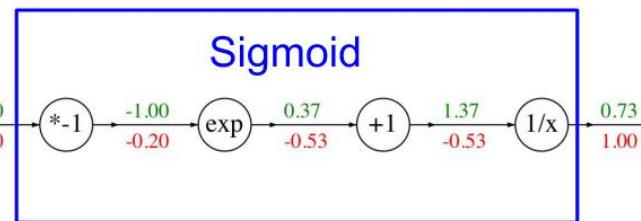
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Sigmoid
function

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

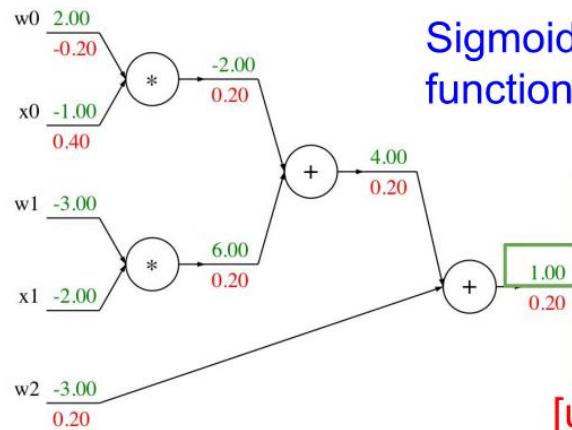


Sigmoid local
gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$

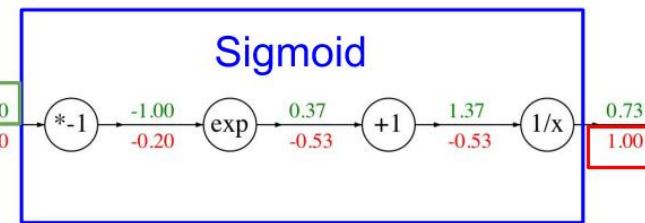
Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Sigmoid function

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



$$[\text{upstream gradient}] \times [\text{local gradient}] \\ [1.00] \times [(1 - 1/(1+e^1)) (1/(1+e^1))] = 0.2$$

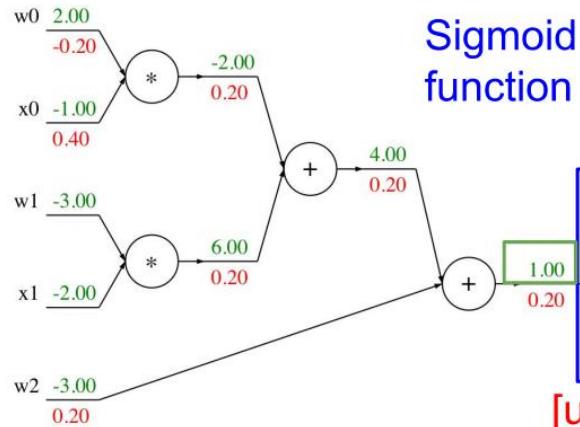
Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

Another example:

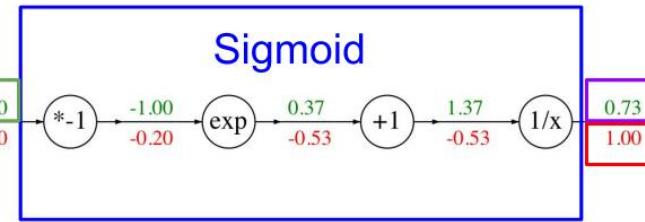
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Sigmoid function

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



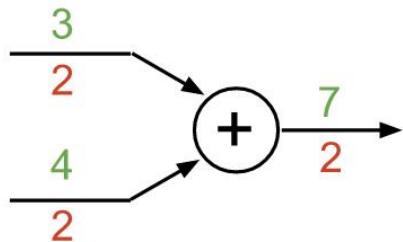
$$[\text{upstream gradient}] \times [\text{local gradient}] \\ [1.00] \times [(1 - 0.73) (0.73)] = 0.2$$

Sigmoid local gradient:

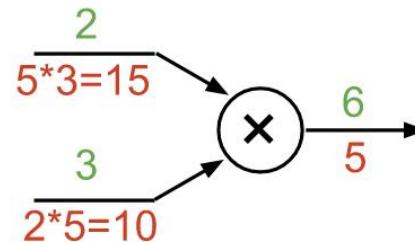
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

Patterns in gradient flow

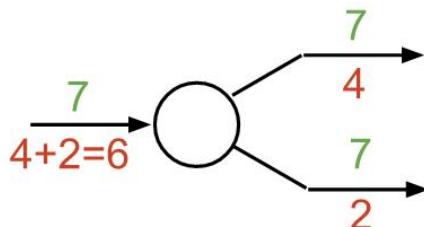
add gate: gradient distributor



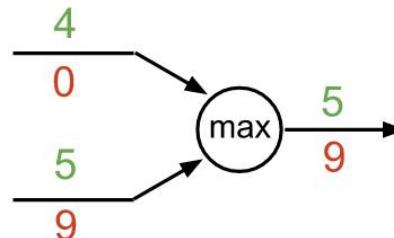
mul gate: “swap multiplier”



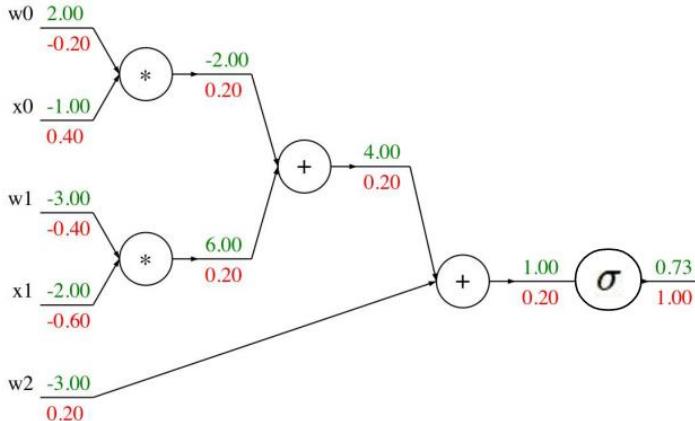
copy gate: gradient adder



max gate: gradient router



Backprop Implementation: “Flat” code



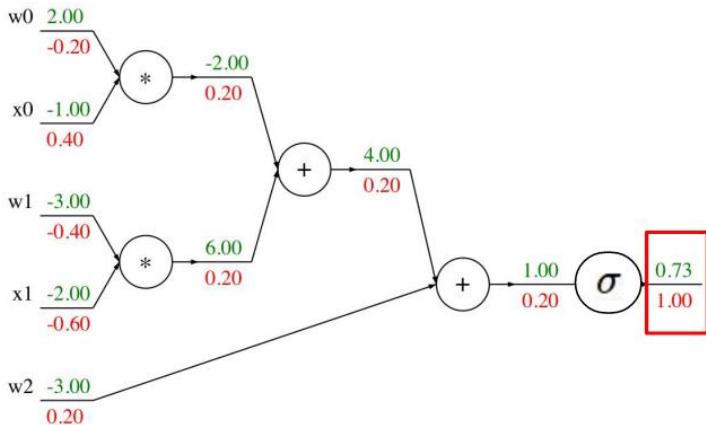
Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Backward pass:
Compute grads

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

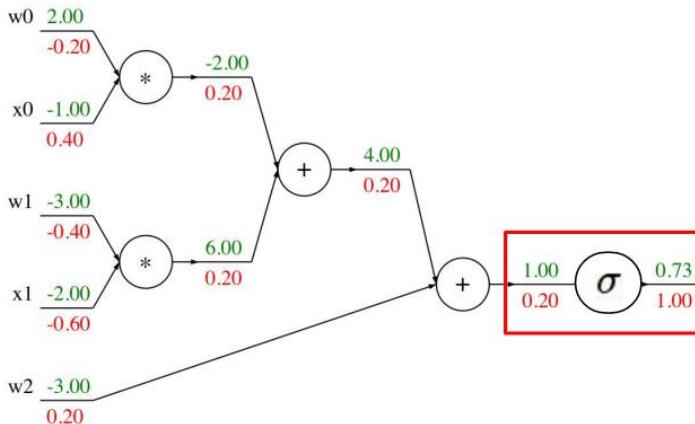
```
s0 = w0 * x0
s1 = w1 * x1
s2 = s0 + s1
s3 = s2 + w2
L = sigmoid(s3)
```

```
grad_L = 1.0
```

```
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Base case

Backprop Implementation: “Flat” code



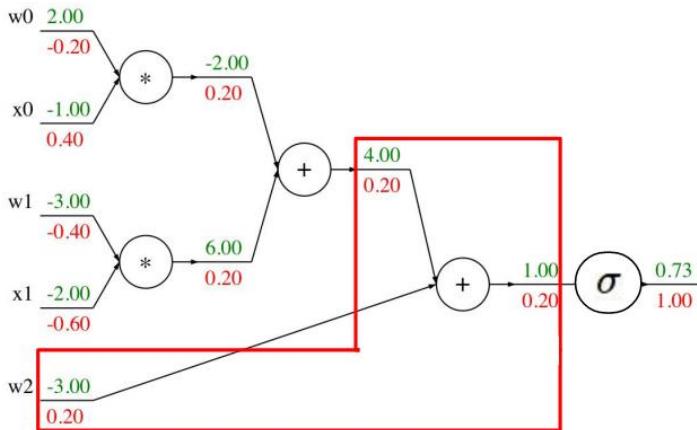
Forward pass:
Compute output

Sigmoid

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



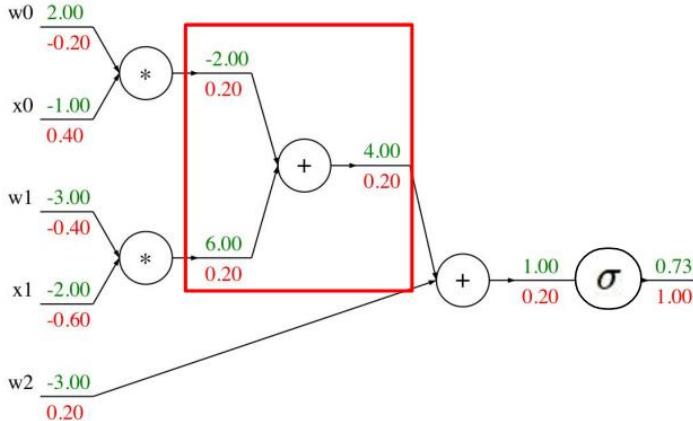
Forward pass:
Compute output

Add gate

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



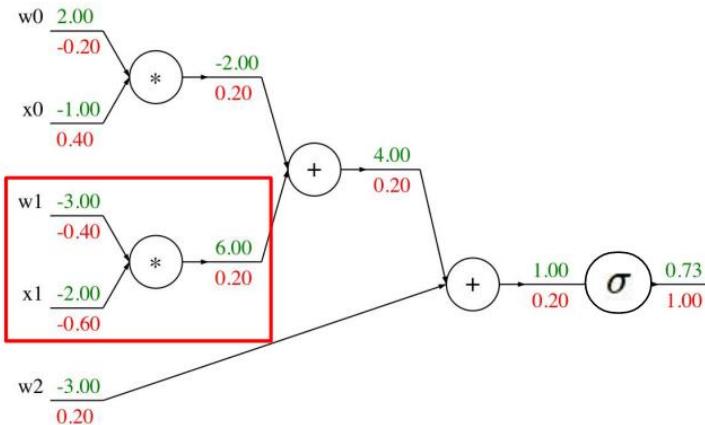
Forward pass:
Compute output

Add gate

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



Forward pass:
Compute output

```

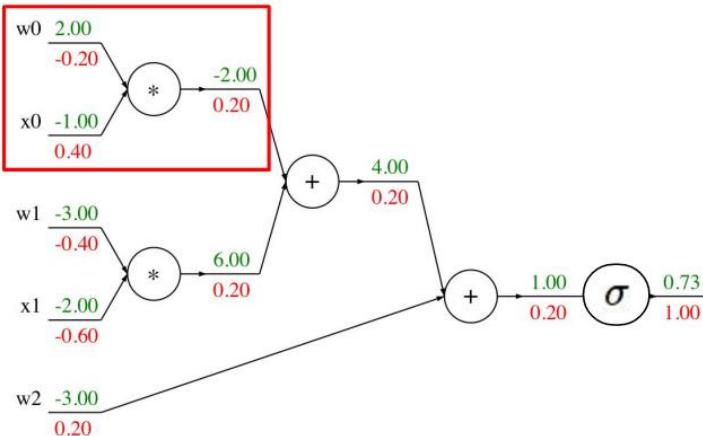
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
  
```

```

grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
  
```

Multiply gate

Backprop Implementation: “Flat” code



Forward pass:
Compute output

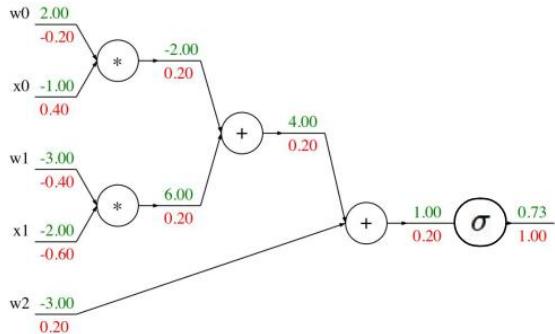
```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Multiply gate

Backprop Implementation: Modularized API

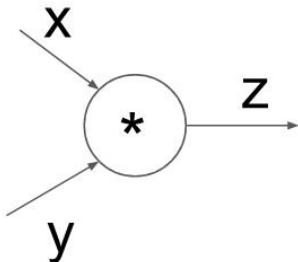
Graph (or Net) object (*rough pseudo code*)



```
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

Gate / Node / Function object: Actual PyTorch code



(x, y, z are scalars)

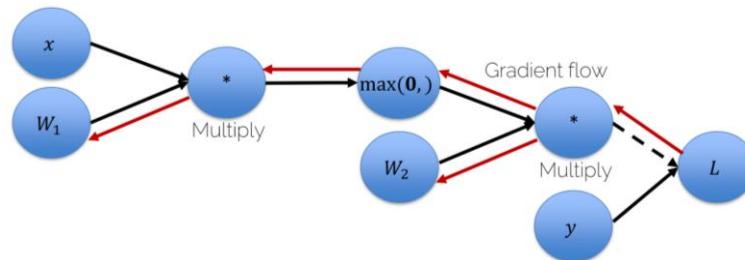
```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y) ← Need to stash some values for use in backward  
        z = x * y  
        return z  
    @staticmethod  
    def backward(ctx, grad_z): ← Upstream gradient  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z # dz/dx * dL/dz  
        grad_y = x * grad_z # dz/dy * dL/dz  
        return grad_x, grad_y
```

Need to stash some values for use in backward

Upstream gradient

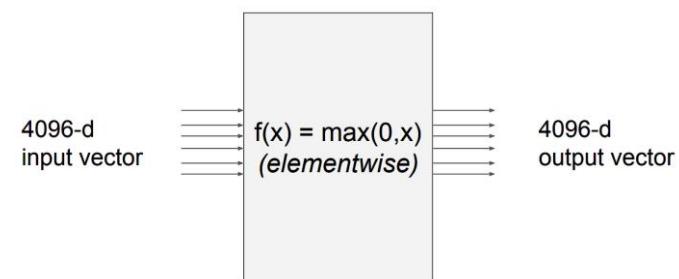
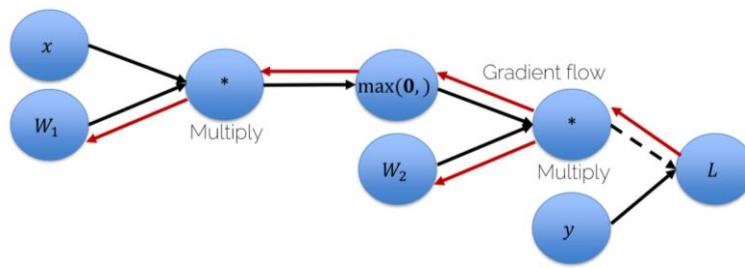
Multiply upstream and local gradients

So far: backprop with scalars



So far: backprop with scalars

What about vector-valued functions?



Recap: Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Vector to Scalar

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will y change?

Vector to Vector

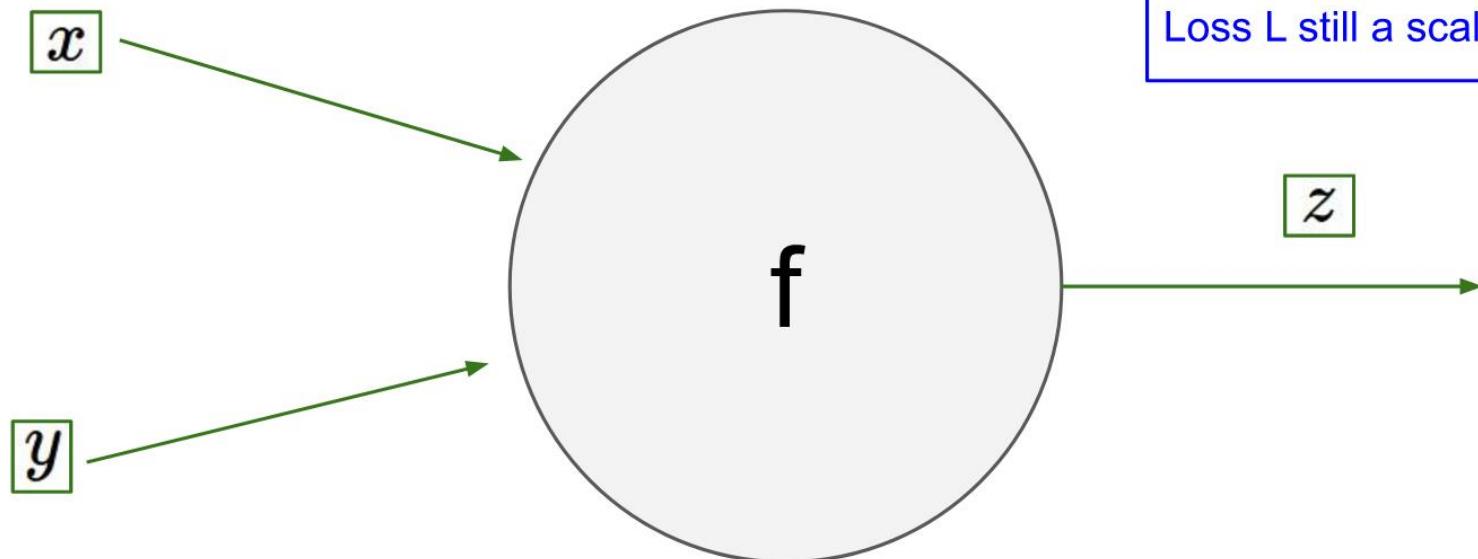
$$x \in \mathbb{R}^N, y \in \mathbb{R}^M$$

Derivative is **Jacobian**:

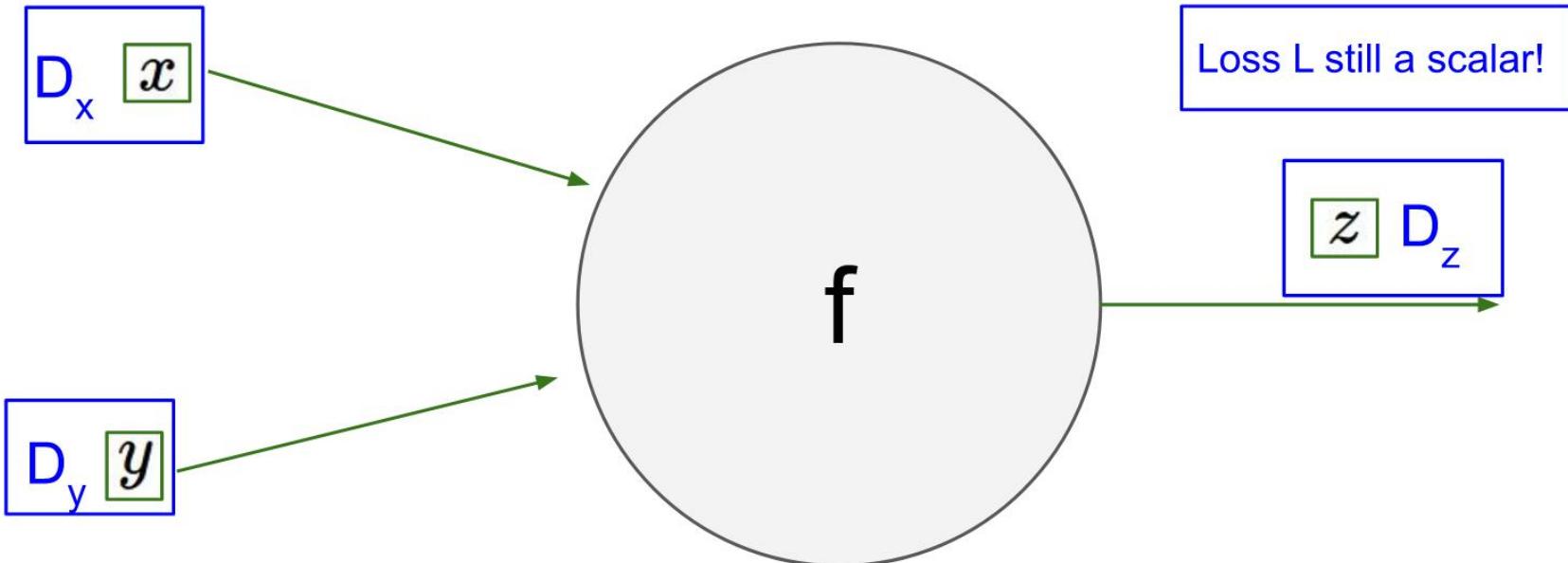
$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x} \right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will each element of y change?

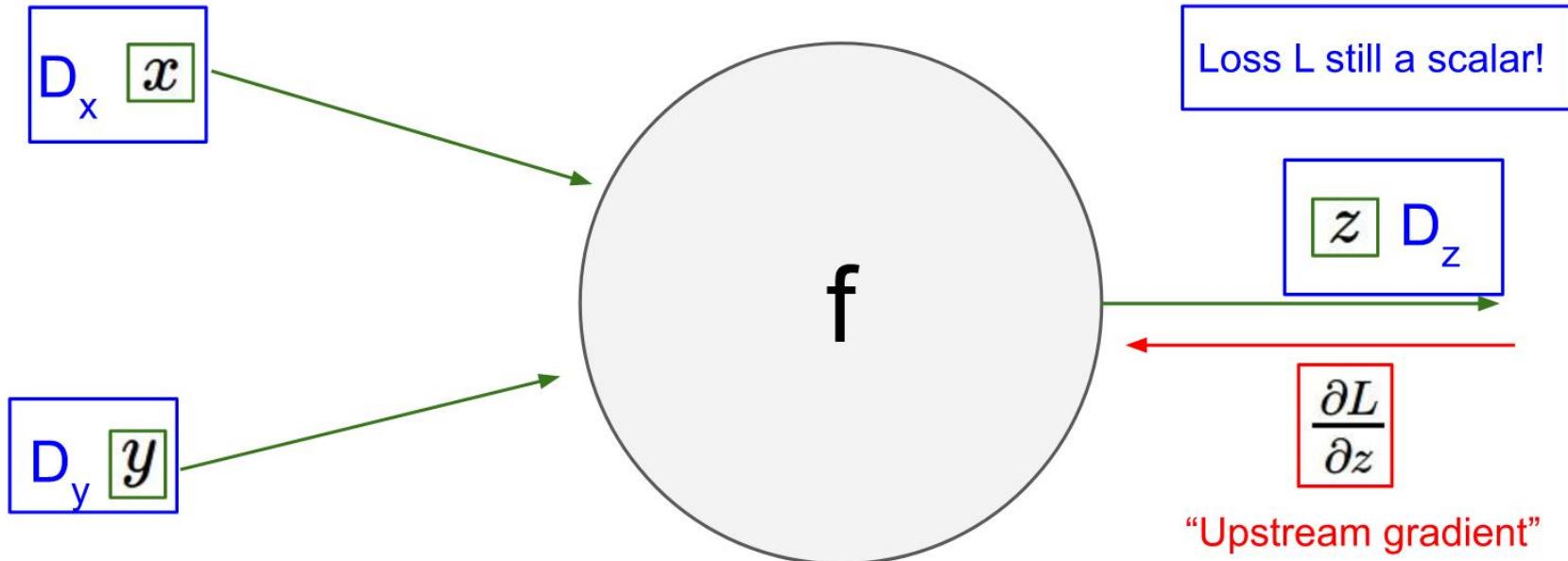
Backprop with Vectors



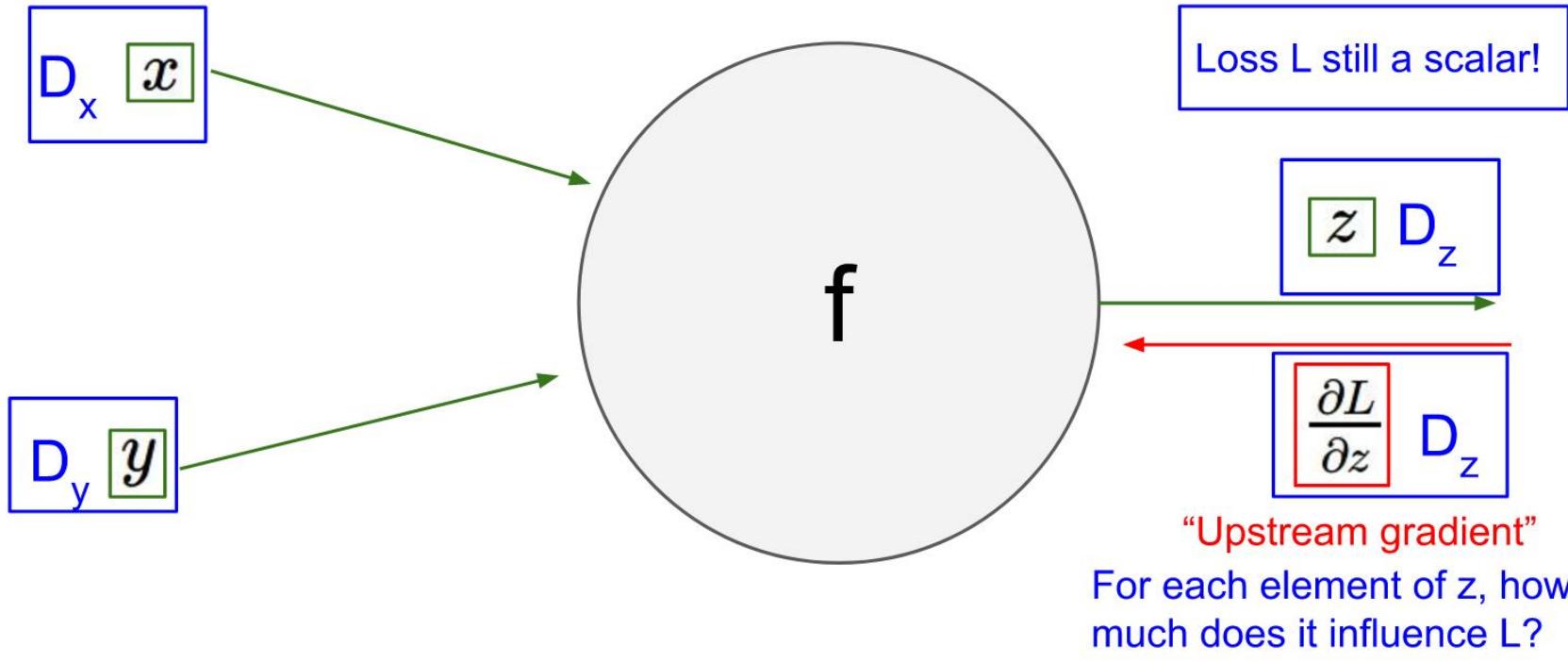
Backprop with Vectors



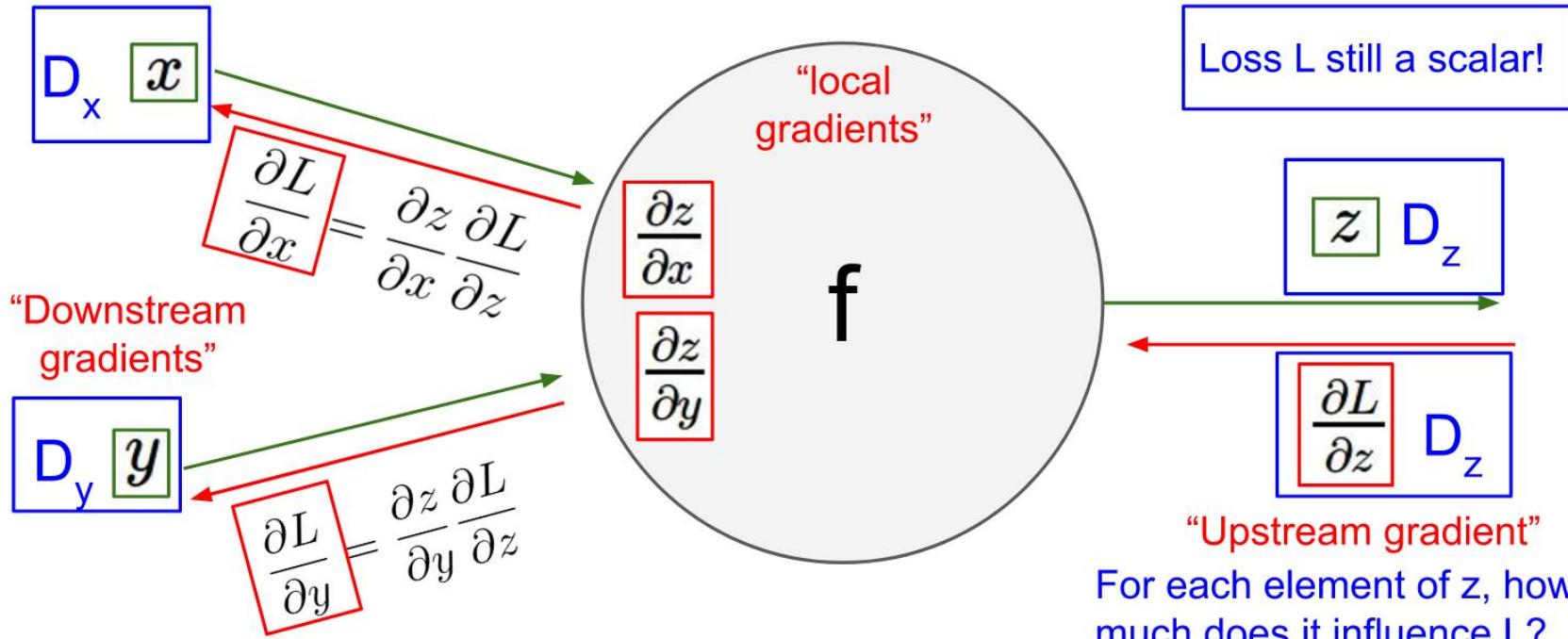
Backprop with Vectors



Backprop with Vectors

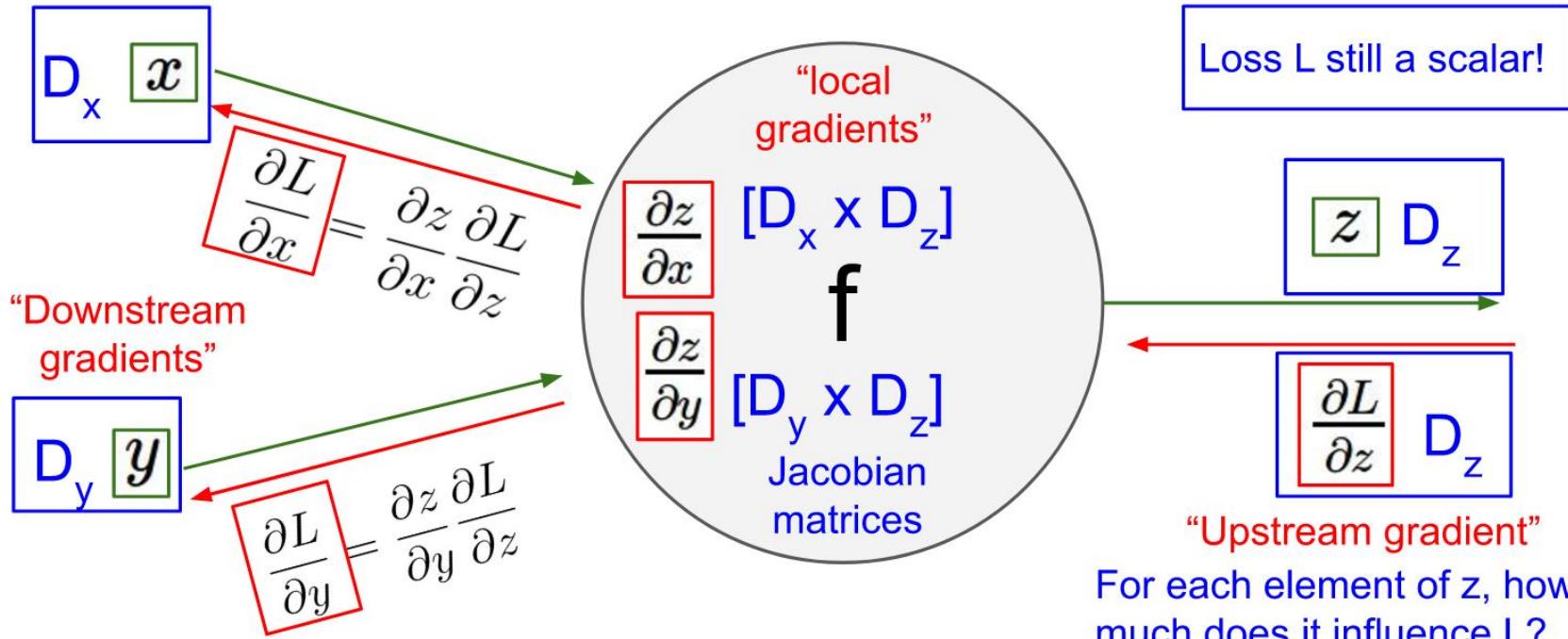


Backprop with Vectors

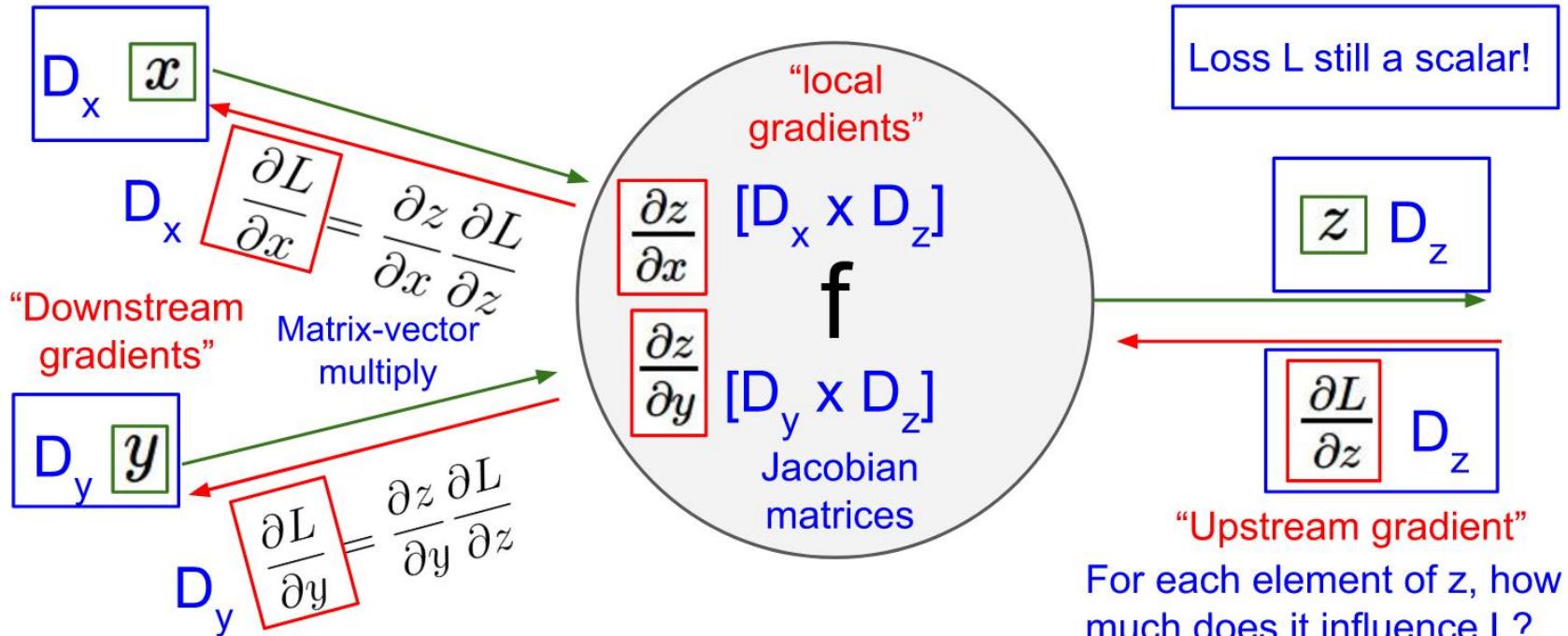


"Upstream gradient"
For each element of z , how
much does it influence L ?

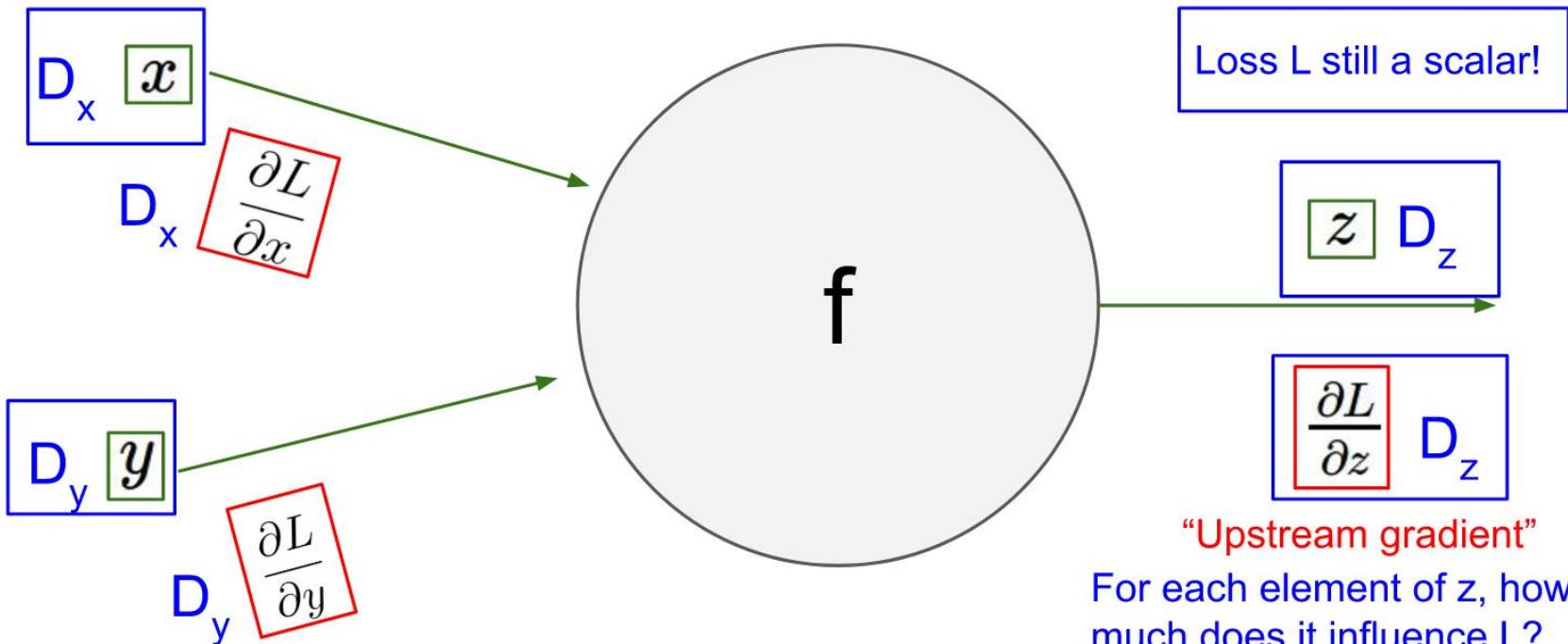
Backprop with Vectors



Backprop with Vectors



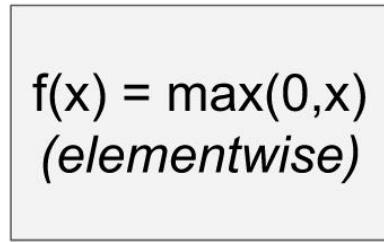
Gradients of variables wrt loss have same dims as the original variable



Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$$



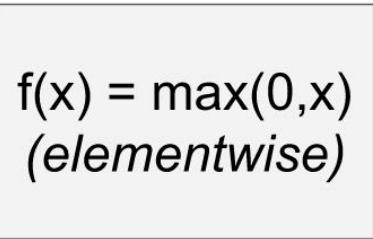
4D output z:

$$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$$



4D output z:

$$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D dL/dz :

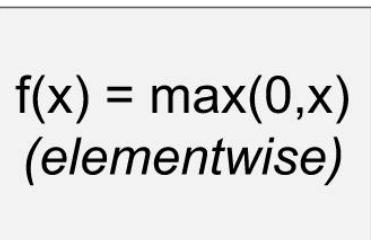
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

Upstream
gradient

Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$$



4D output z:

$$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

Jacobian dz/dx

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

4D dL/dz :

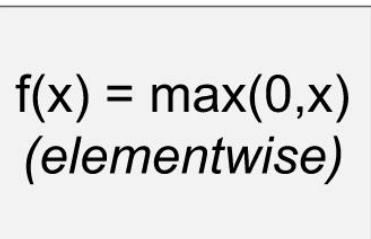
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

Upstream
gradient

Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$$



4D output z:

$$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

[$\frac{dz}{dx}$] [$\frac{dL}{dz}$]

$$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

4D $\frac{dL}{dz}$:

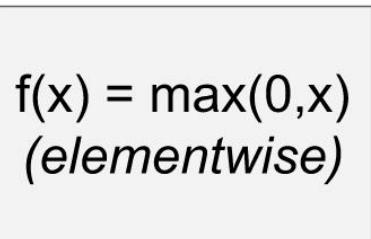
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

Upstream
gradient

Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$$



4D output z:

$$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D dL/dx :

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

4D dL/dz :

$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

Upstream
gradient

Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$$

$$f(x) = \max(0, x)$$

(elementwise)

4D output z:

$$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D dL/dx :

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow$$

$[dz/dx] [dL/dz]$

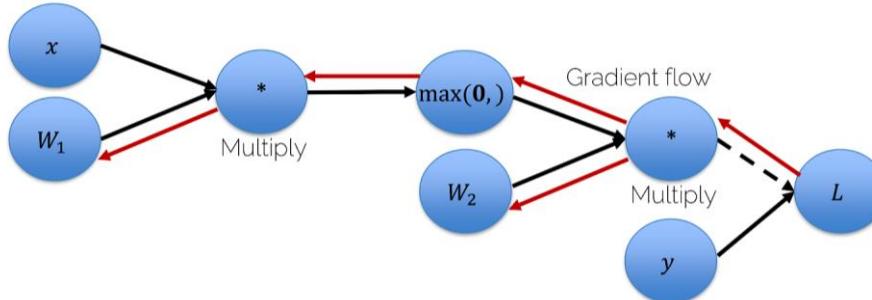
$$\left(\frac{\partial L}{\partial x} \right)_i = \begin{cases} \left(\frac{\partial L}{\partial z} \right)_i & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

4D dL/dz :

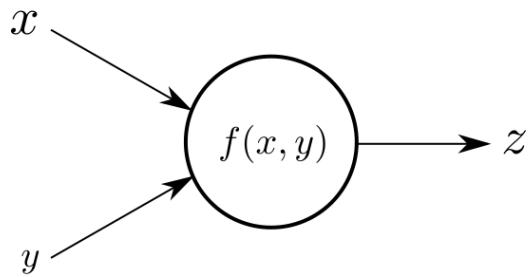
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \leftarrow$$

Upstream
gradient

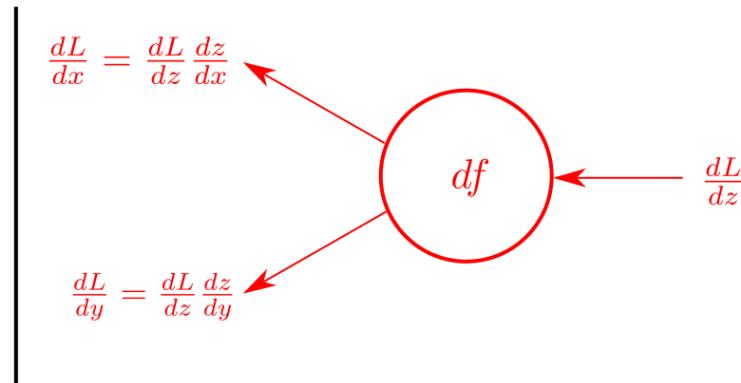
Computational Graphs & Backpropagation



Forwardpass



Backwardpass



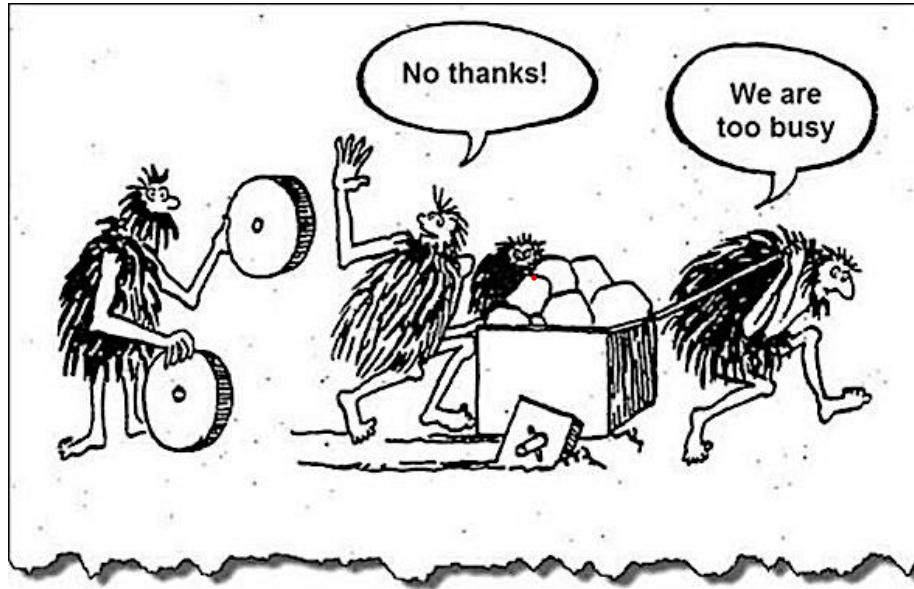
Computational Graphs & Backpropagation

		Variable
	Scalar x	Vector $\mathbf{x} \in \mathbb{R}^n$
Scalar f	Derivative $\frac{\partial f}{\partial x}$	Gradient $\left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right]^T$
Function	Partial Derivative $\left[\frac{\partial f_1(x)}{\partial x}, \dots, \frac{\partial f_m(x)}{\partial x} \right]^T$	Jacobian Matrix
		$\begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{bmatrix}$

Chain Rule

$$\frac{dy}{dx} = \frac{dy}{dt} \times \frac{dt}{du} \times \frac{du}{dx}$$

Technical/Knowledge Debt



- Bag of Words/Features
- Logistic Regression
- Backprop Implementation



Papers With Code

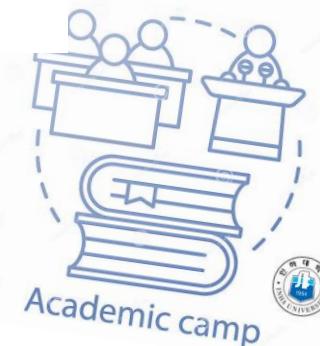


reddit



Top Conferences for Machine Learning & Arti. Intelligence

Ranking is based on Conference H5-index>=12 provided by Google Scholar Metrics				
		Show Due only	All Categories	All Countries
1	240		CVPR : IEEE/CVF Conference on Computer Vision and Pattern Recognition Jun 16, 2020 - Jun 18, 2020 - Seattle , United States https://cvpr2020.thecvf.com/	Conference Details
2	169		NeurIPS : Neural Information Processing Systems (NIPS) Dec 10, 2019 - Dec 12, 2019 - Vancouver , Canada https://nips.cc	
3	137		ECCV : European Conference on Computer Vision Aug 23, 2020 - Aug 28, 2020 - Glasgow , United Kingdom https://eccv2020.eu/	Deadline : to be confirmed
4	135		ICML : International Conference on Machine Learning (ICML) Jul 13, 2020 - Jul 18, 2020 - Vienna , Austria https://icml.cc/Conferences/2020	Deadline : to be confirmed
5	129		ICCV : IEEE/CVF International Conference on Computer Vision Oct 27, 2019 - Nov 3, 2019 - Seoul , South Korea https://iccv2019.thecvf.com/	
6	106		ACL : Meeting of the Association for Computational Linguistics (ACL) Jul 5, 2020 - Jul 10, 2020 - Seattle , United States https://acl2020.org/	Deadline : Mon 09 Dec 2019



arxiv-sanity.com

Arxiv Sanity Preserver



Course Project

Refer

Overview

The Course Project is an opportunity for you to apply what you have learned in class to a problem of your interest. Potential projects usually fall into these two tracks:

- **Applications.** If you're coming to the class with a specific background and interests (e.g. biology, engineering, physics), we'd love to see you apply ConvNets to problems related to your particular domain of interest. Pick a real-world problem and apply ConvNets to solve it.
- **Models.** You can build a new model (algorithm) with ConvNets, or a new variant of existing models, and apply it to tackle vision tasks. This track might be more challenging, and sometimes leads to a piece of publishable work.

One **restriction** to note is that this is a Computer Vision class, so your project should involve pixels of visual data in some form somewhere. E.g. a pure NLP project is not a good choice, even if your approach involves ConvNets.

To get a better feeling for what we expect from CS231n projects, we encourage you to take a look at the project reports from previous years:

- [Spring 2017](#)
- [Winter 2016](#)
- [Winter 2015](#)

To inspire ideas, you might also look at recent deep learning publications from top-tier conferences, as well as other resources below.

- [CVPR](#): IEEE Conference on Computer Vision and Pattern Recognition
- [ICCV](#): International Conference on Computer Vision
- [ECCV](#): European Conference on Computer Vision
- [NIPS](#): Neural Information Processing Systems
- [ICLR](#): International Conference on Learning Representations
- [ICML](#): International Conference on Machine Learning
- Publications from the [Stanford Vision Lab](#)
- [Awesome Deep Vision](#)
- [Past CS229 Projects](#): Example projects from Stanford's machine learning class
- [Kaggle challenges](#): An online machine learning competition website. For example, a [Yelp classification challenge](#).

For applications, this type of projects would involve careful data preparation, an appropriate loss function, details of training and cross-validation and good test set evaluations and model comparisons. Don't be afraid to think outside of the box. Some successful examples can be found below:

- [Teaching Deep Convolutional Neural Networks to Play Go](#)
- [Playing Atari with Deep Reinforcement Learning](#)
- [Winning the Galaxy Challenge with convnets](#)

