CS 7643: Deep Learning

Topics:

- Regularization
- Neural Networks
  - Modular Design
- Computing Gradients

Dhruv Batra

Georgia Tech

Recall from last time: Linear Classifier

Image

Array of **32x32x3** numbers (3072 numbers total)

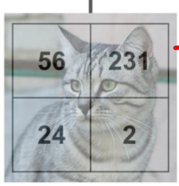
→ f(**x**,**W**) → **10** numbers giving class scores

↑

W
parameters or weights

$$f(x,W) = Wx + b$$

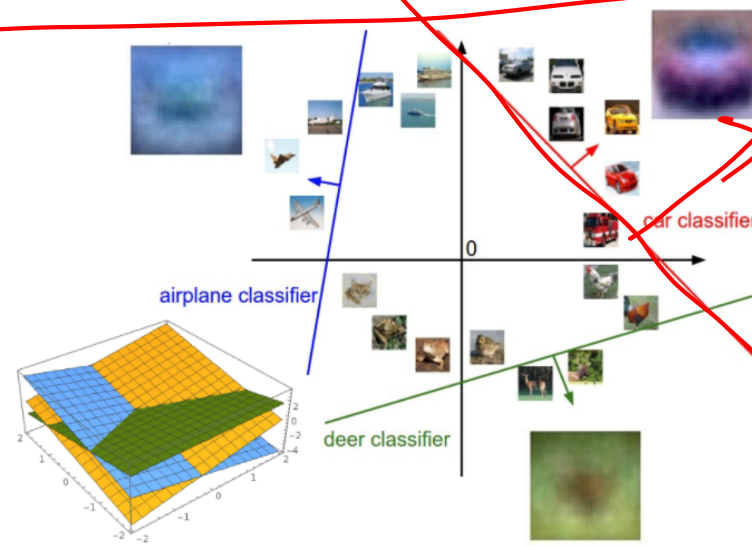
Stretch pixels into column



Input image

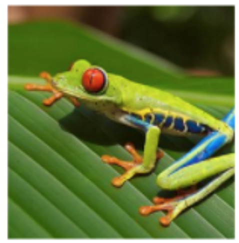
					x		b		=	
0.2	-0.5	0.1	2.0		56	+	1.1	=	-96.8	Cat score
1.5	1.3	2.1	0.0		231		3.2		437.9	Dog score
0	0.25	0.2	-0.3		24		-1.2		61.95	Ship score
					2					

W



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

Recall from last time: Linear Classifier

airplane	-3.45	-0.51	3.42
automobile	-8.87	**6.04**	4.64
bird	0.09	5.31	2.65
cat	**2.9**	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	**-4.34**
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

TODO:

1. Define a **loss function** that quantifies our unhappiness with the scores across the training data.

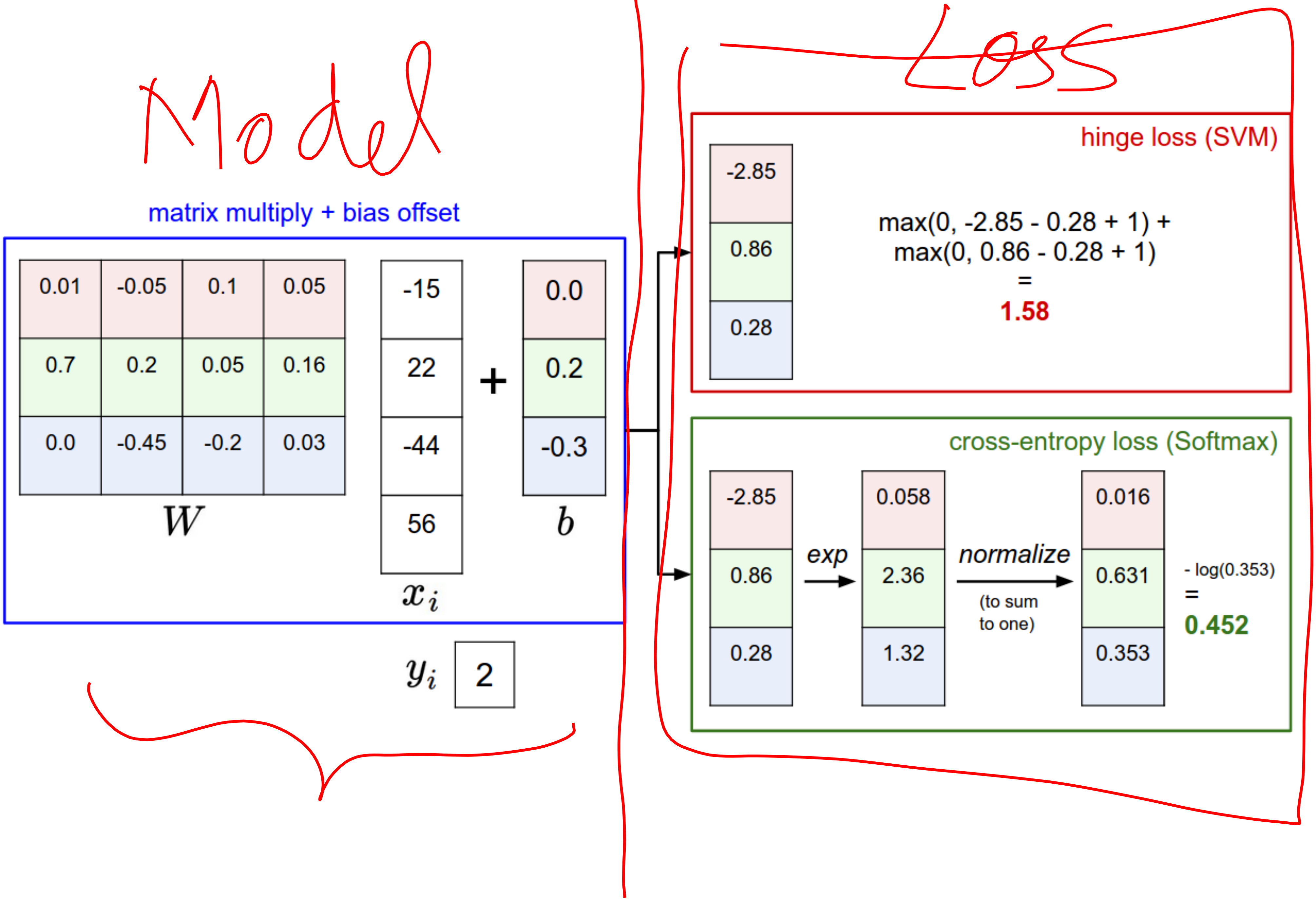
1. Come up with a way of efficiently finding the parameters that minimize the loss function. **(optimization)**

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

Softmax vs. SVM

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \qquad\qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$-\log P(Y = y_i \mid \vec{w}, x_i)$$



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

Plan for Today

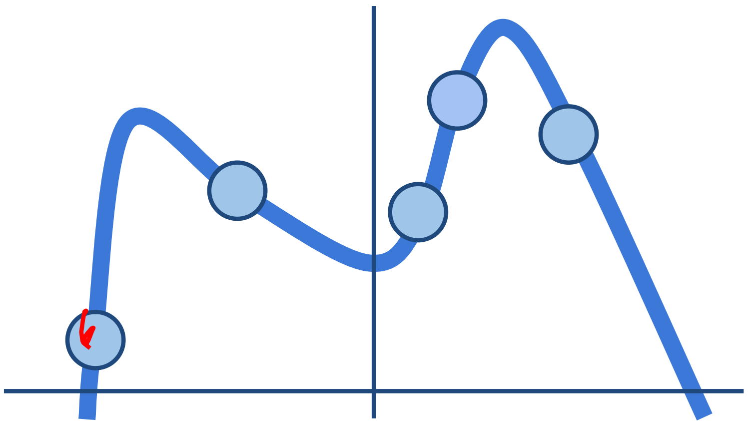
- Regularization
- Neural Networks
  - Modular Design
- Computing Gradients

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

Data loss: Model predictions should match training data

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

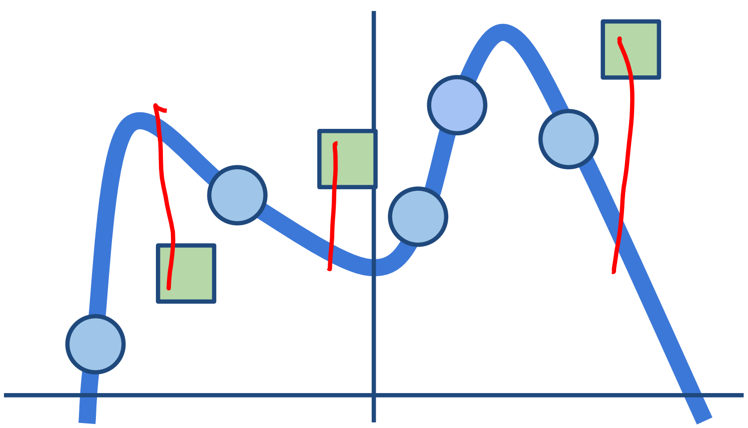
Data loss: Model predictions should match training data



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

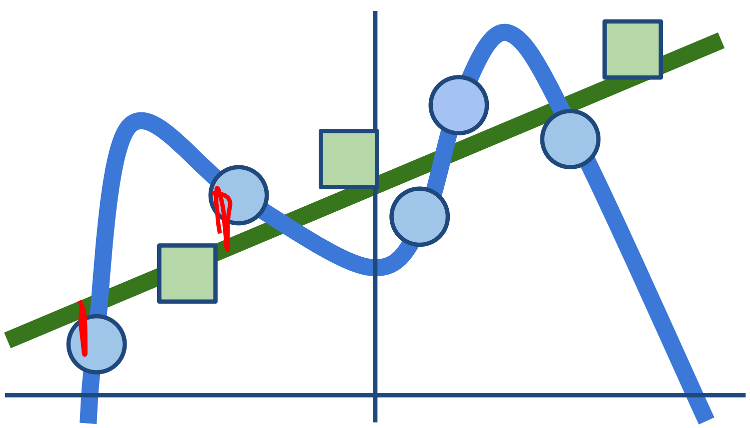
$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i)$$

Data loss: Model predictions should match training data



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

Data loss: Model predictions should match training data

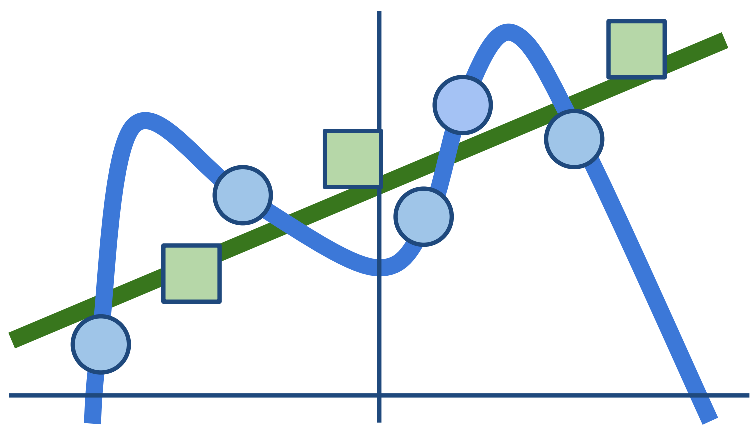


Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

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

Data loss: Model predictions should match training data

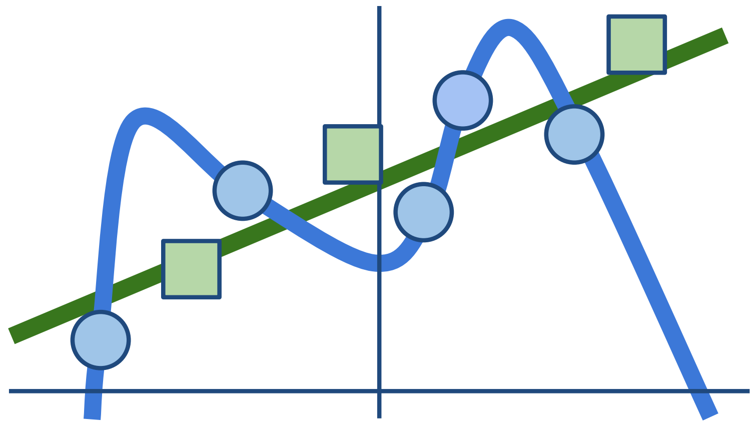
Regularization: Model should be “simple”, so it works on test data



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

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

Data loss: Model predictions should match training data



Regularization: Model should be “simple”, so it works on test data

Occam’s Razor:
“Among competing hypotheses, the simplest is the best”
William of Ockham, 1285 - 1347

Regularization

λ = regularization strength (hyperparameter)

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Dropout (will see later)

Fancier: Batch normalization, stochastic depth

L2 Regularization (Weight Decay)

$$x = [1, 1, 1, 1]$$

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization (Weight Decay)

$$x = [1, 1, 1, 1]$$

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$w_1 = [1, 0, 0, 0]$$
$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

(If you are a Bayesian: L2 regularization also corresponds MAP inference using a Gaussian prior on W)

$$w_1^T x = w_2^T x = 1$$

Recap

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

Softmax

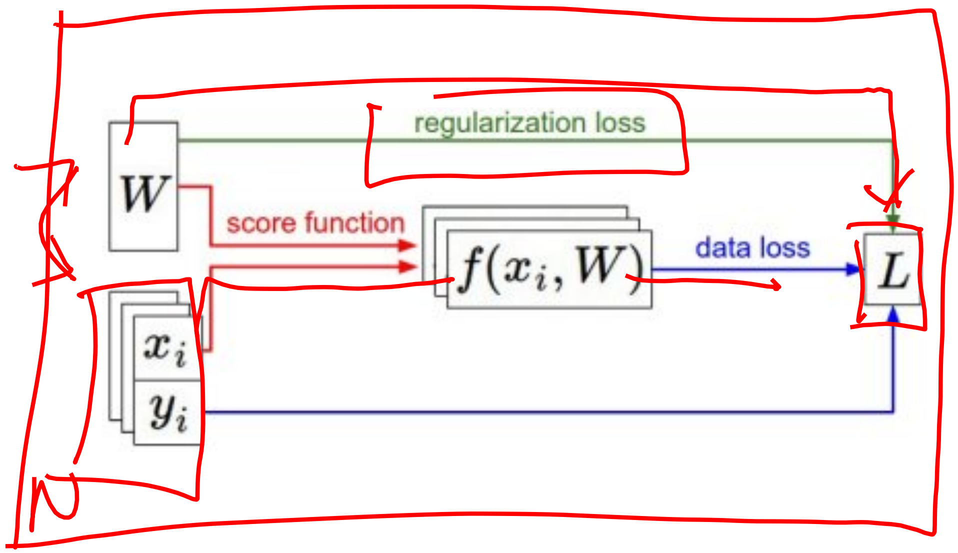
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM

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

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

Full loss



Recap

How do we find the best W?

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

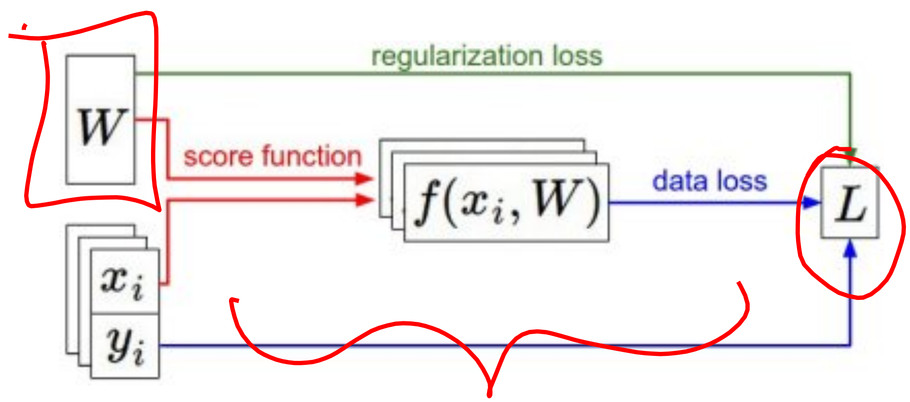
Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM

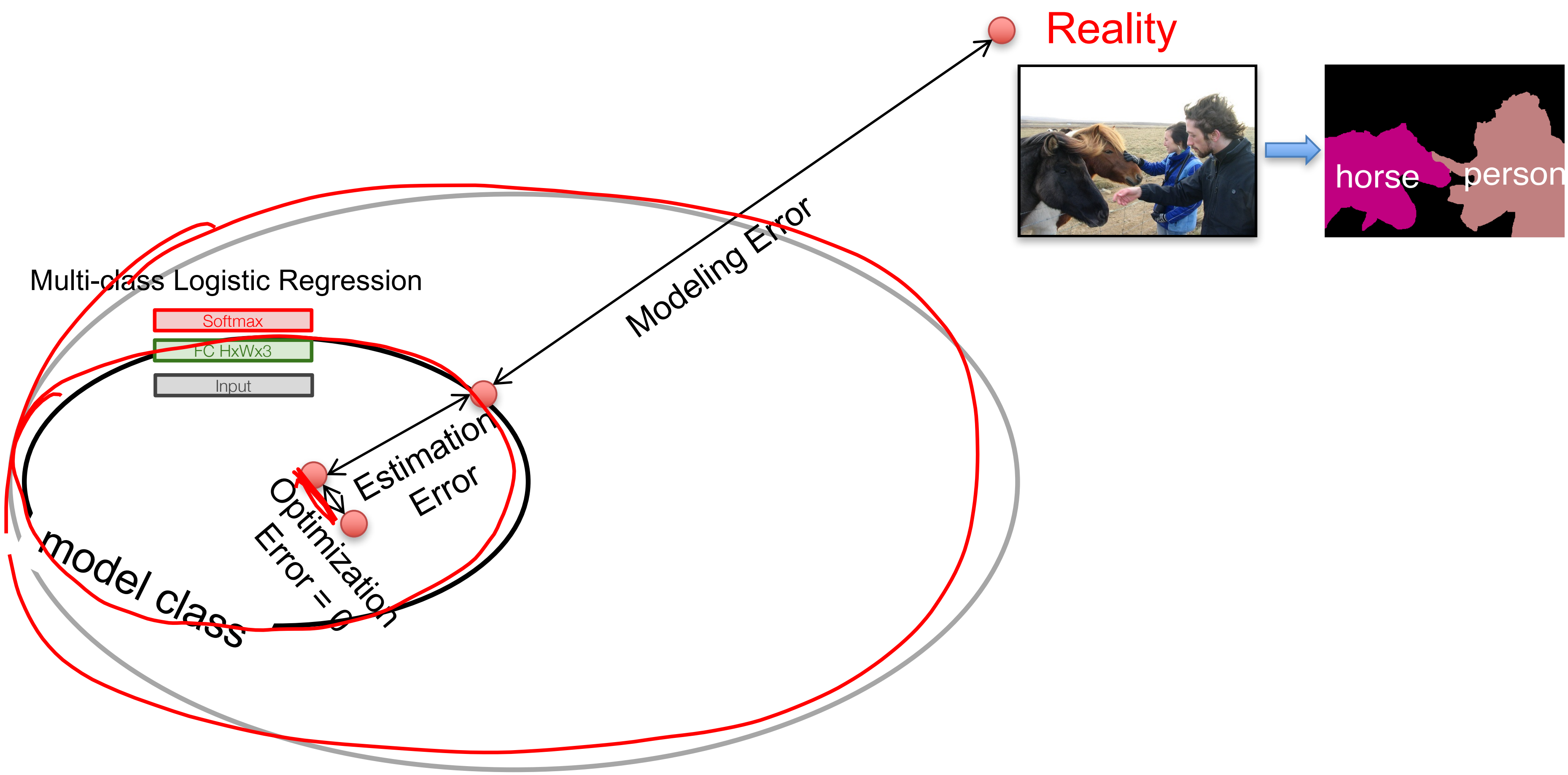
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

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



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

Error Decomposition



Next: Neural Networks

Neural networks: without the brain stuff

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

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

Neural networks: without the brain stuff

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

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

$g_3(\vec{x}) = W_2 \vec{x}$ $= g_3(g_2(g_1(x)))$

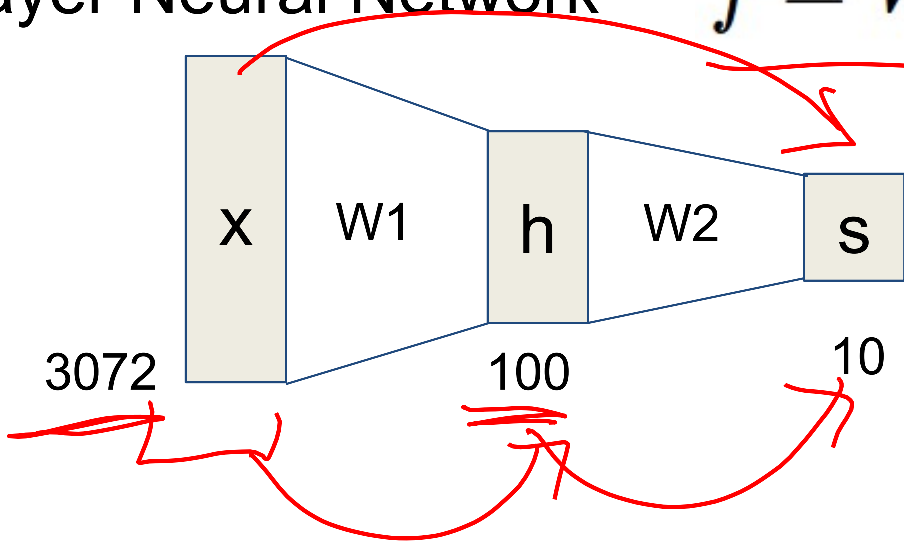
$g_1(\vec{x}) = W\vec{x}$

$g_2(\vec{x}) = \max\{0, \vec{x}\}$ ReLU

Neural networks: without the brain stuff

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

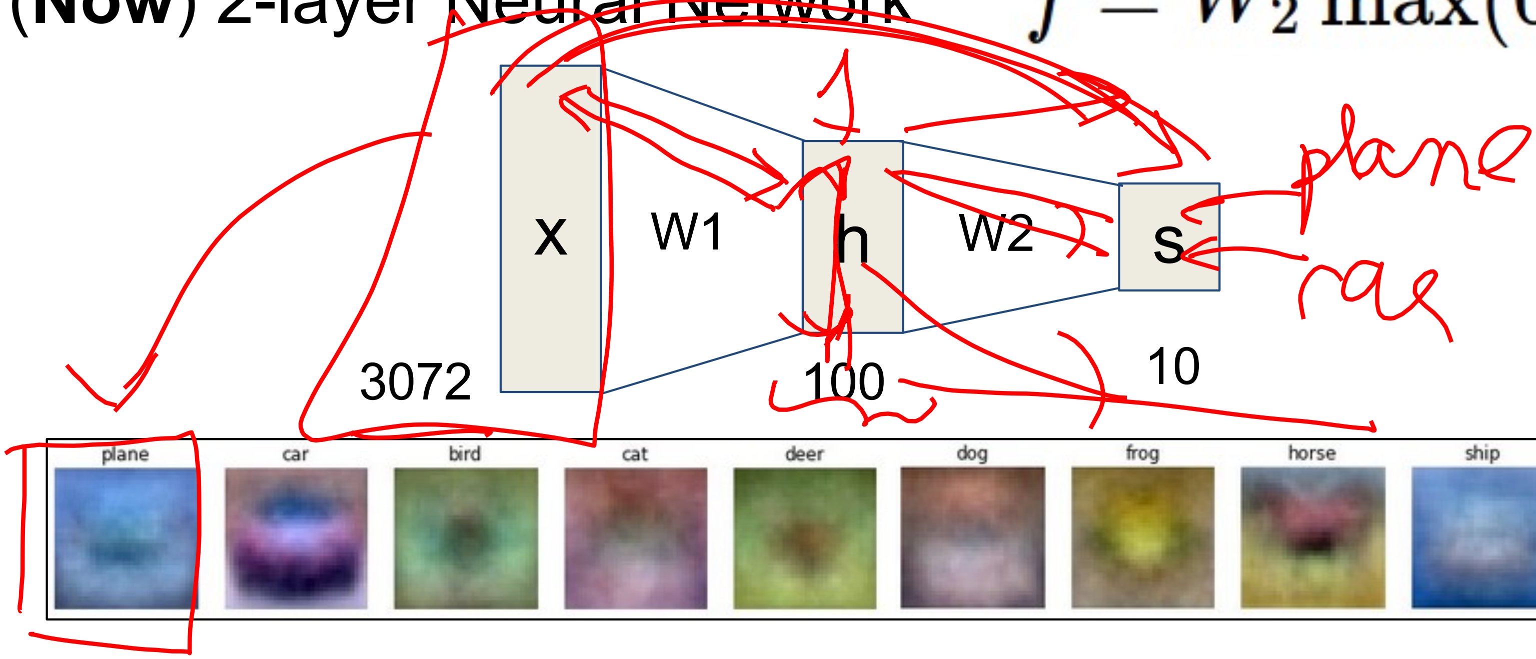
(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



Neural networks: without the brain stuff

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

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



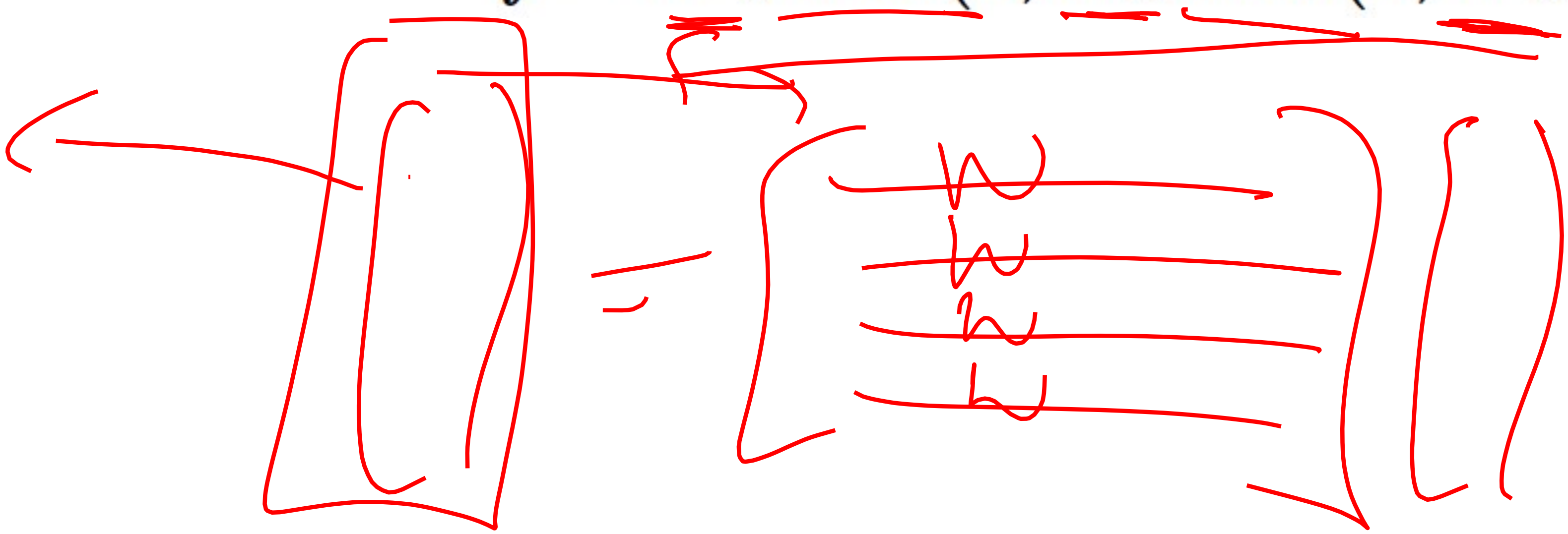
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

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

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



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

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

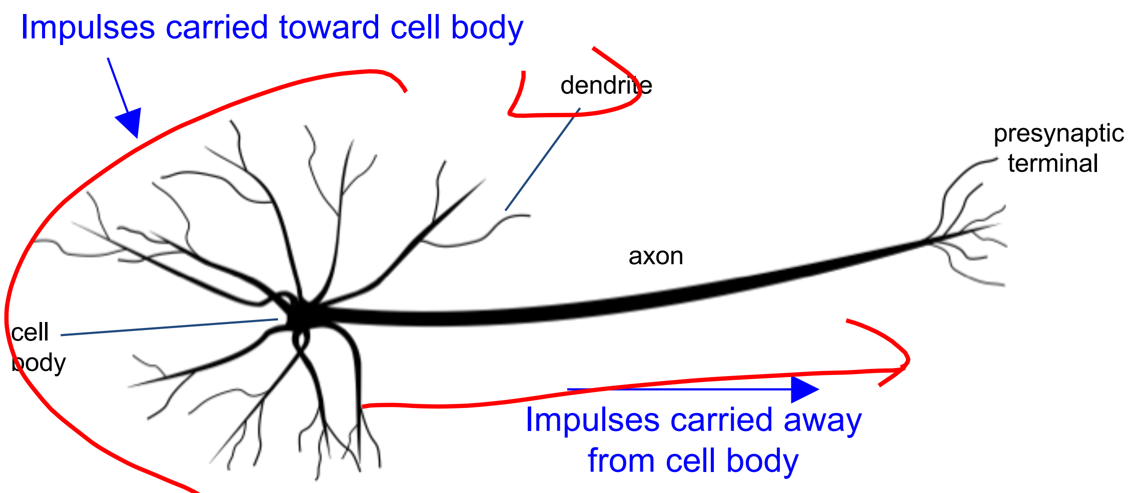
```
import numpy as np
from numpy.random import randn

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

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

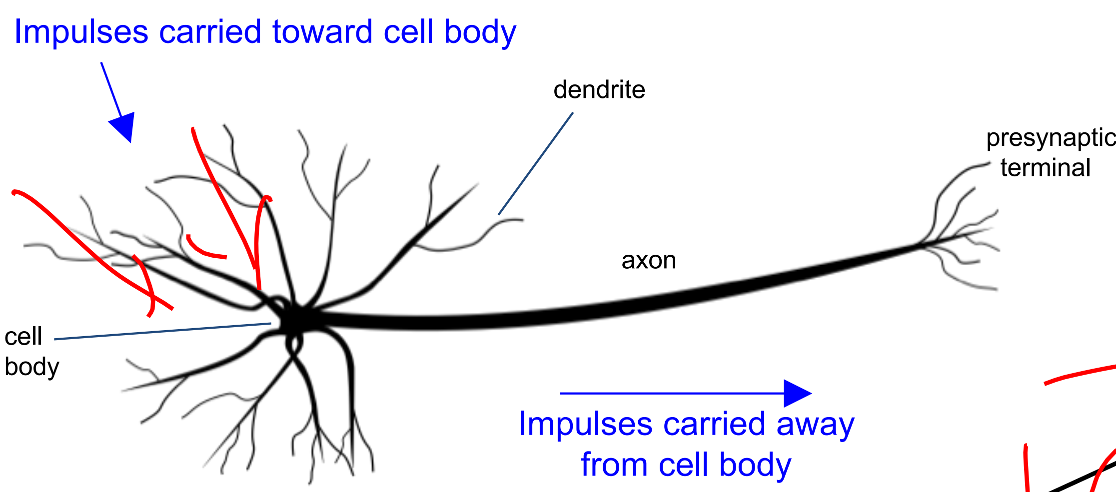
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

In Assignment 2: Writing a 2-layer

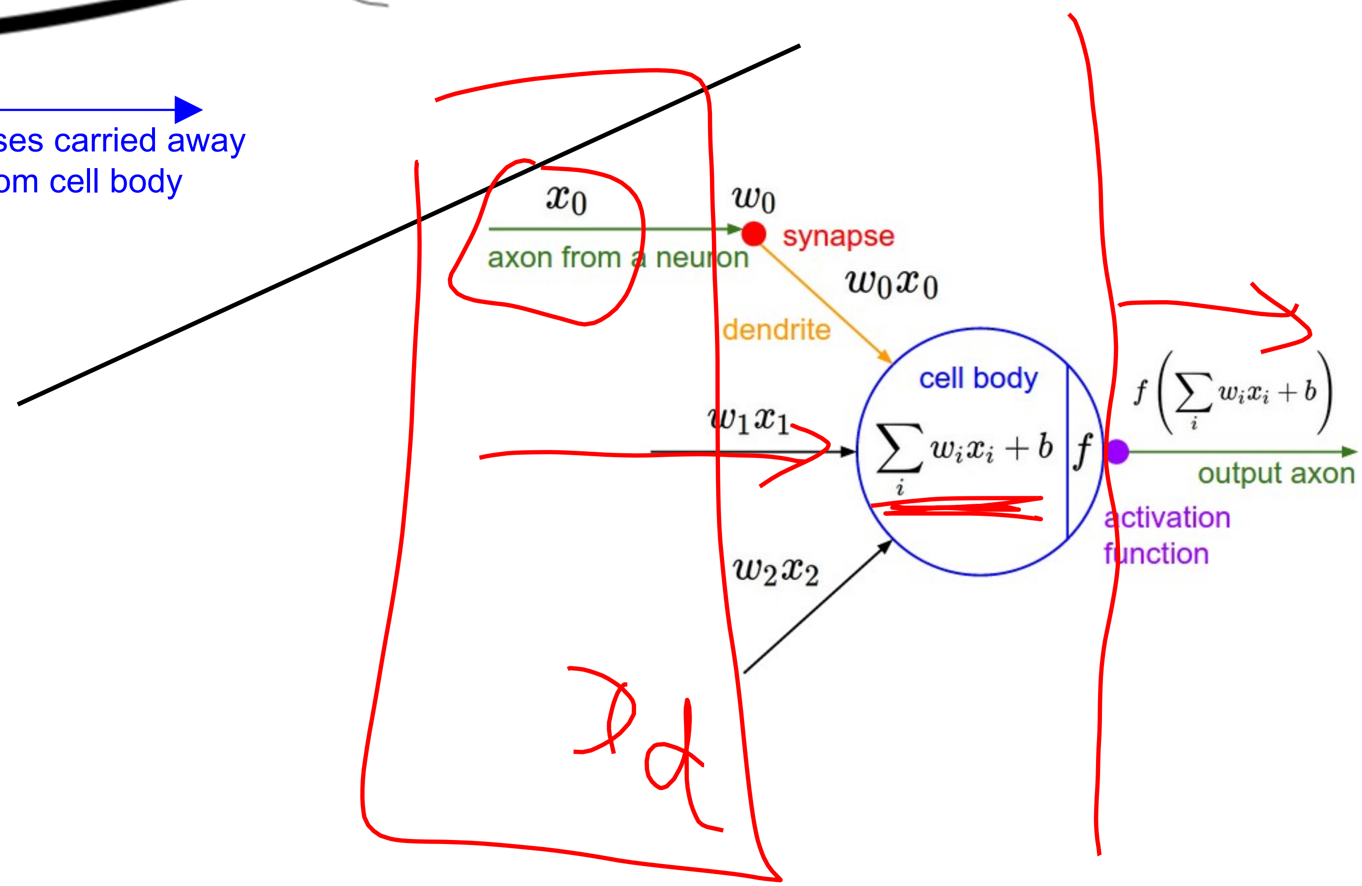
```
# receive W1,W2,b1,b2 (weights/biases), X (data)
# forward pass:
h1 = #... function of X,W1,b1
scores = #... function of h1,W2,b2
loss = #... (several lines of code to evaluate Softmax loss)
# backward pass:
dscores = #...
dh1,dW2,db2 = #...
dW1,db1 = #...
```

This image by Fotis Bobolas is licensed under CC-BY 2.0

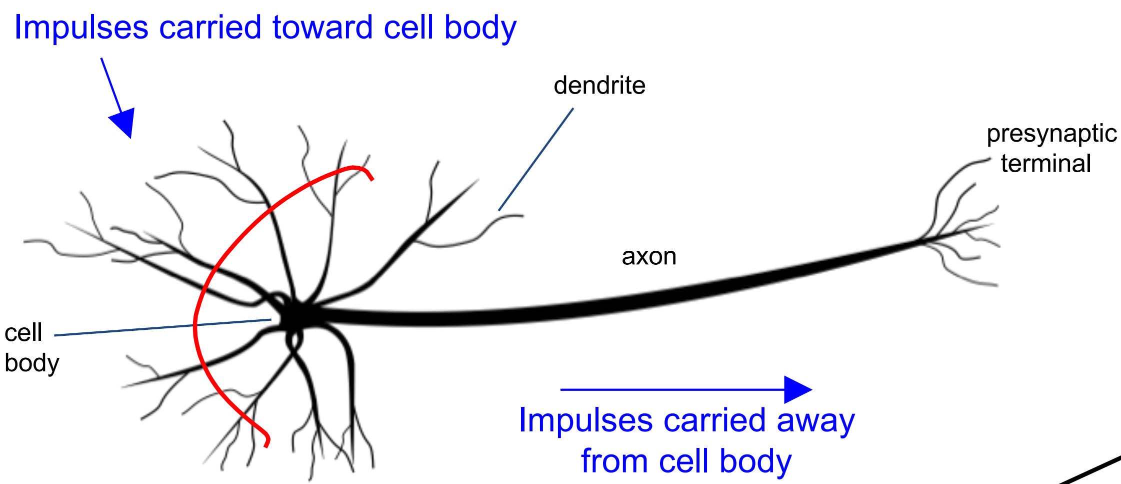

Impulses carried toward cell body
dendrite
presynaptic
terminal
axon
cell
body
Impulses carried away
from cell body

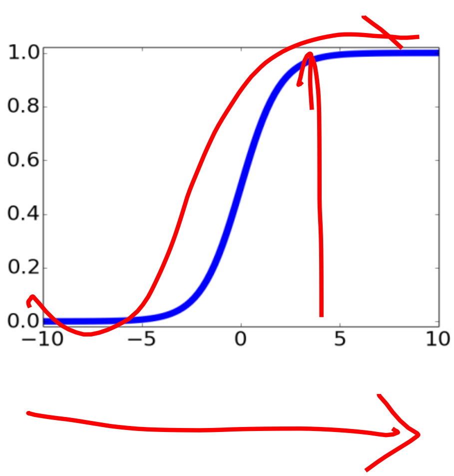


Impulses carried toward cell body
dendrite
presynaptic
terminal
axon
cell
body
Impulses carried away
from cell body



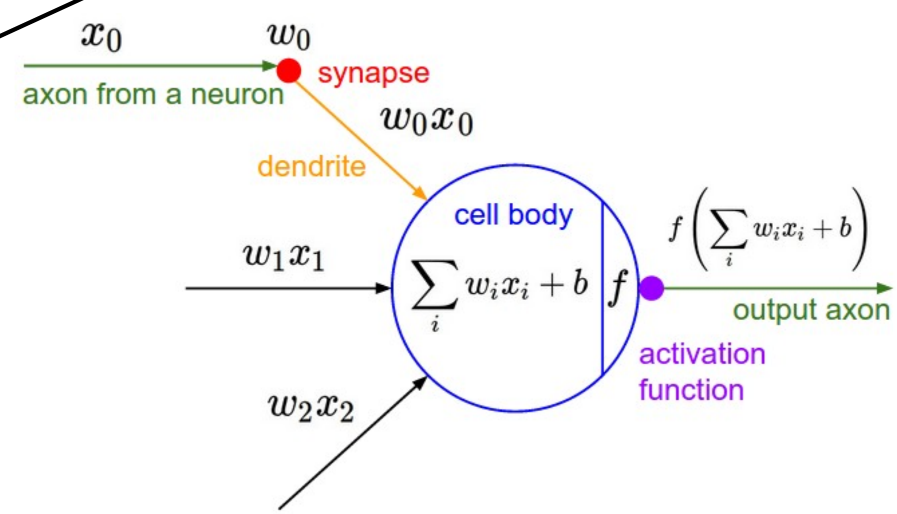
x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$w_1 x_1$
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
output axon
activation
function
$w_2 x_2$

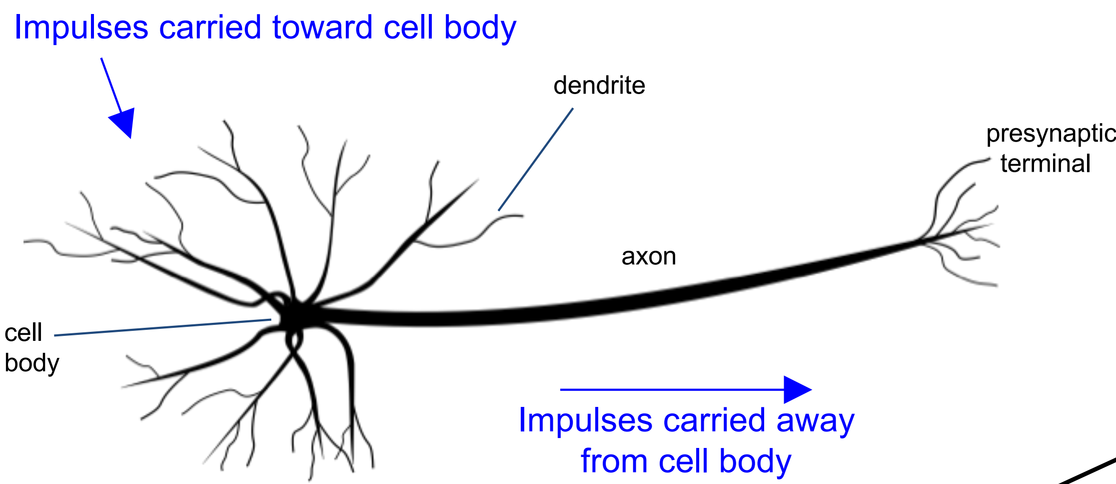


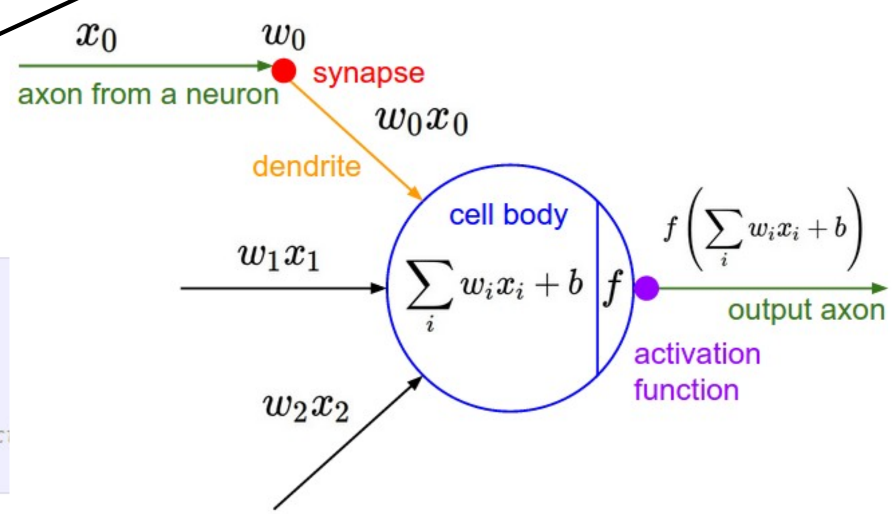


sigmoid activation function

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








```python
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation func
    return firing_rate
```

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

Be very careful with your brain analogies!

Biological Neurons:

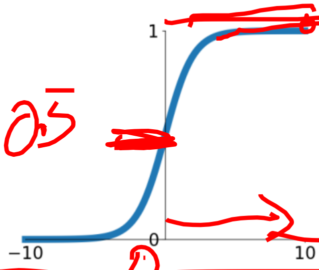
- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

[Dendritic Computation. London and Hausser]

Activation functions

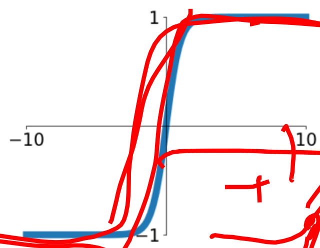
Sigmoid

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



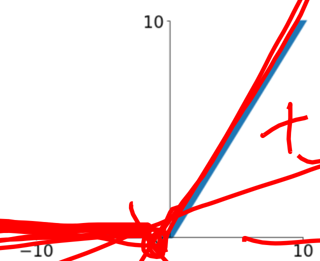
tanh

$\tanh(x)$



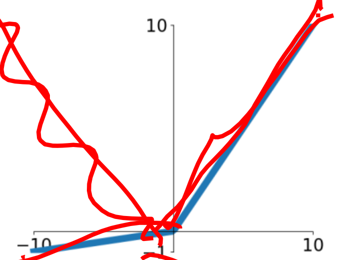
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

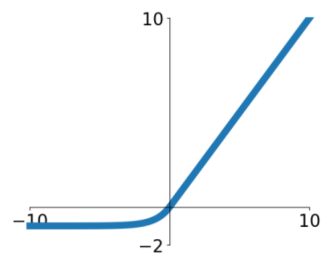


Maxout

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

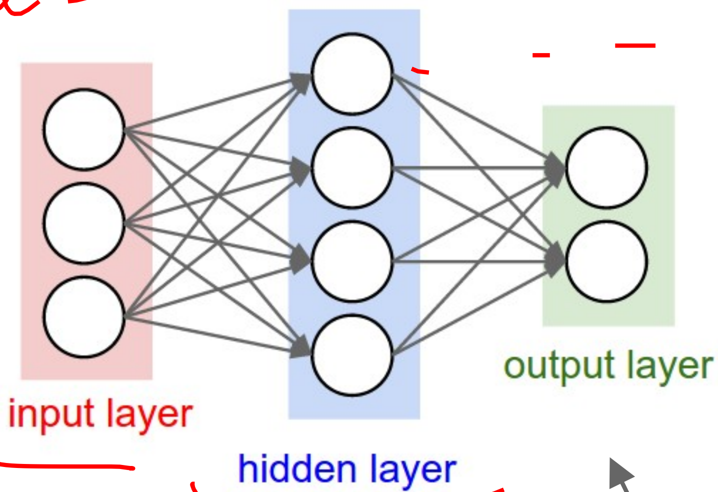
ELU

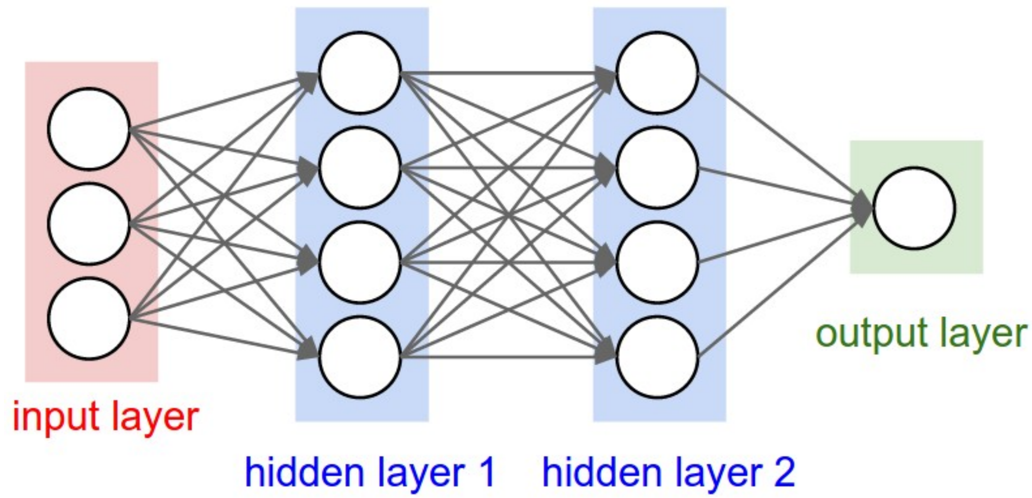
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



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

Neural networks: Architectures





"2-layer Neural Net", or
"1-hidden-layer Neural Net"

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

"Fully-connected" layers

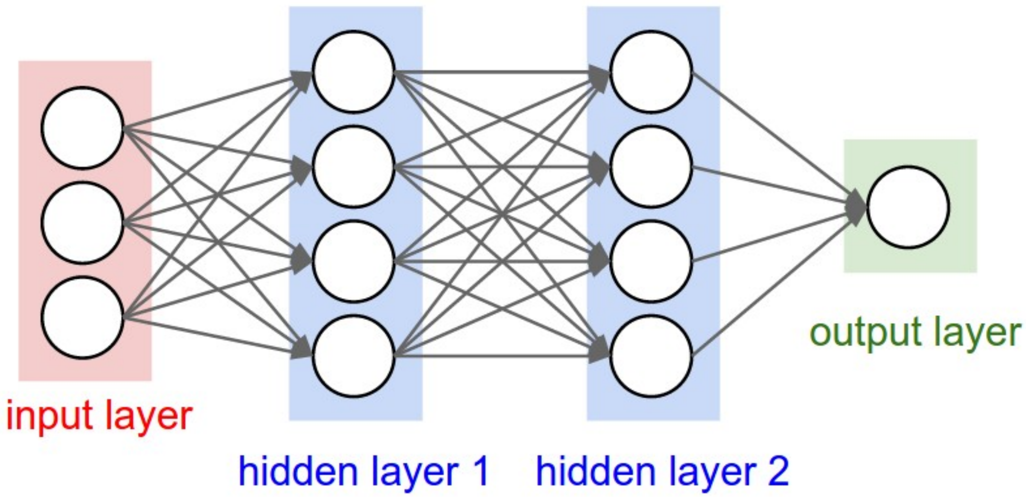
Inner Prod

Example feed-forward computation of a neural network

```
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

We can efficiently evaluate an entire layer of neurons.

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)
```

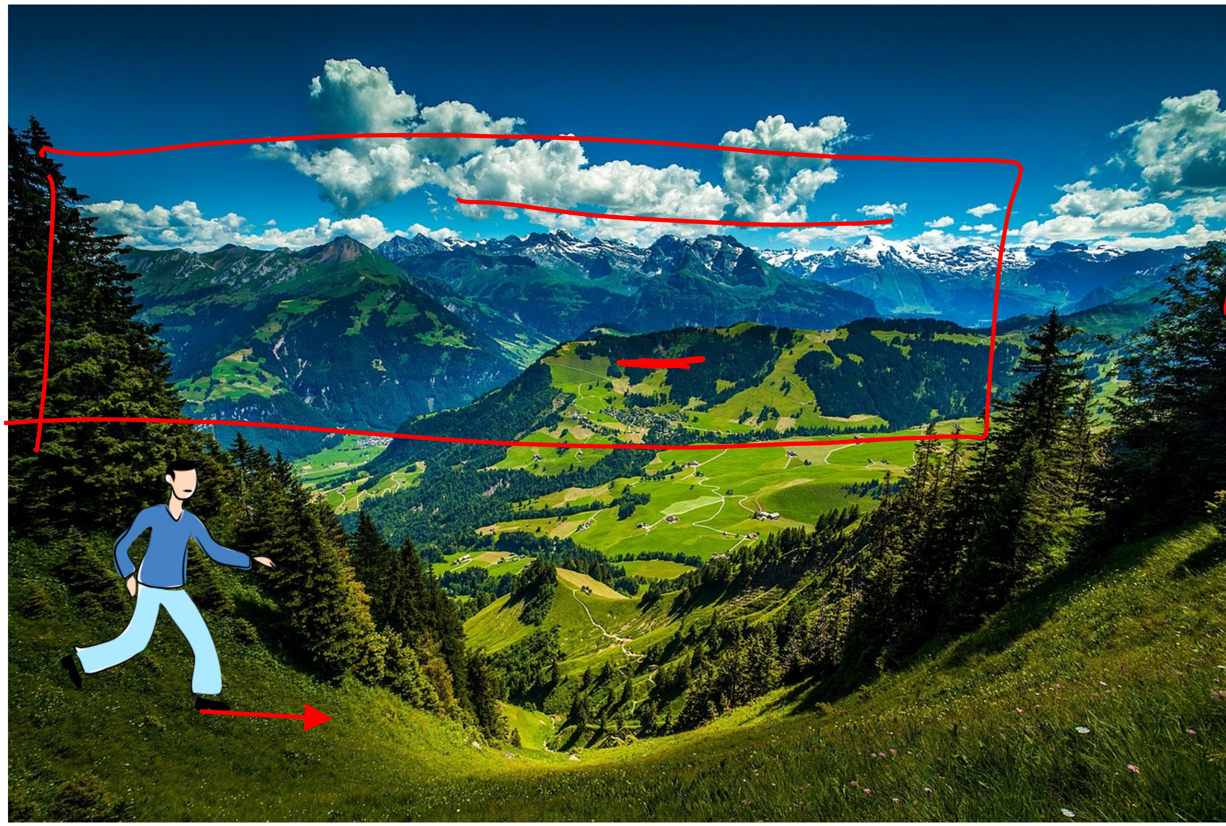
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Optimization

This image is CC0 1.0 public domain

Strategy: **Follow the slope**

$$f(\vec{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix}$$



Strategy: **Follow the slope**

In 1-dimension, the derivative of a function:

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

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

Gradient Descent

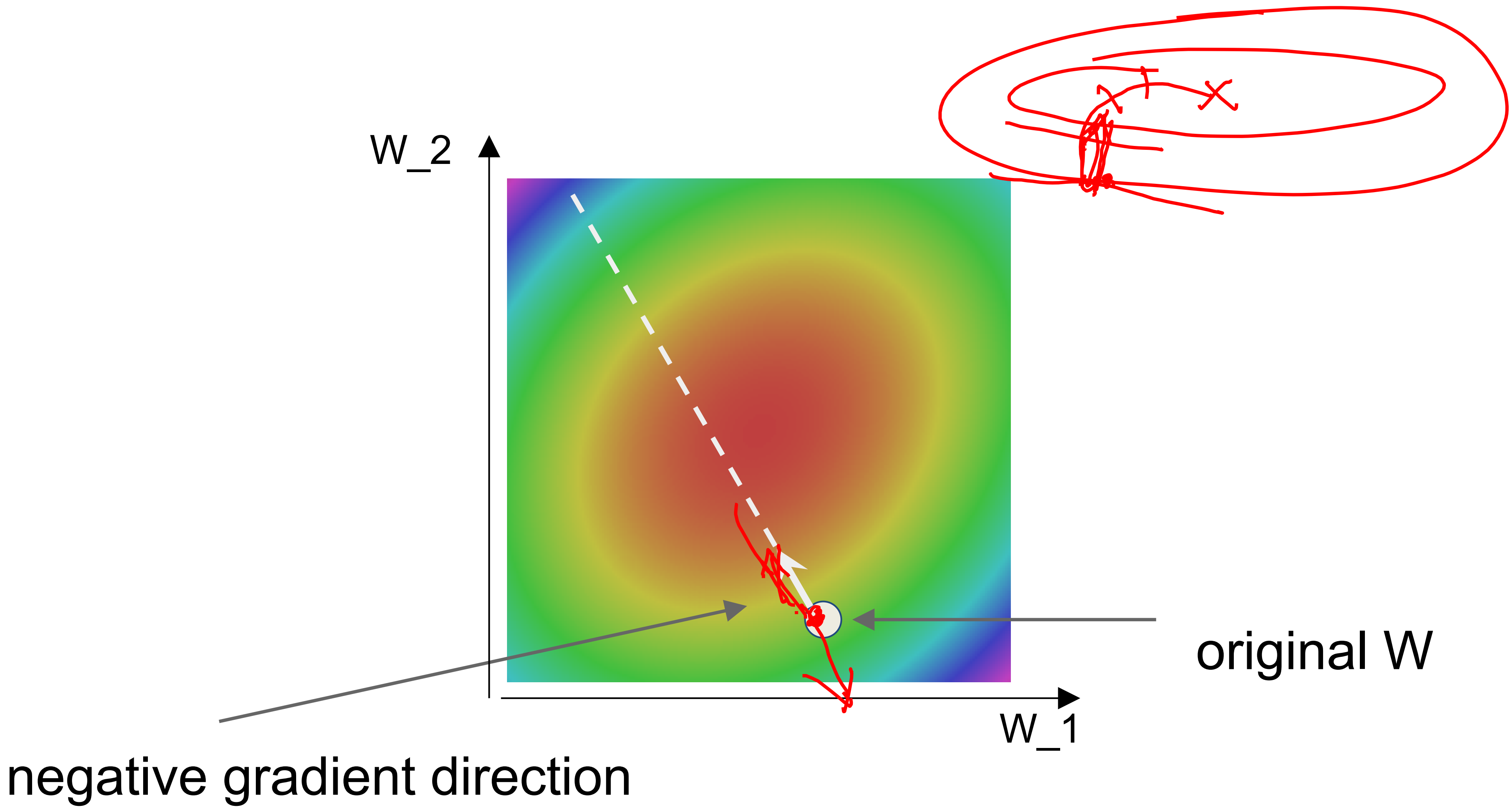
$$L = \frac{1}{N}\sum L_i + R(W)$$

$$\frac{\partial L}{\partial W}$$

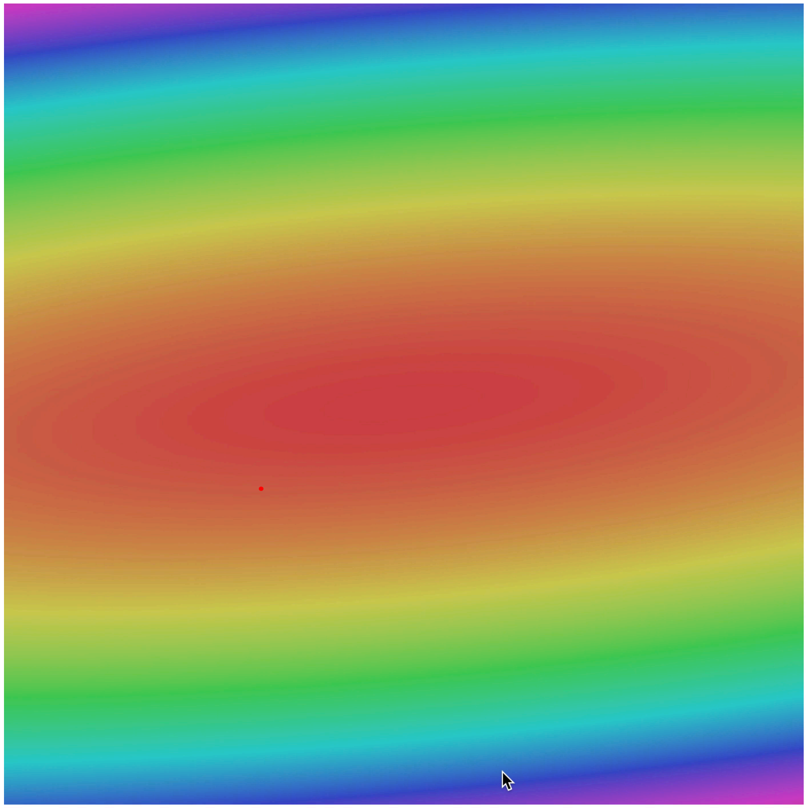
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

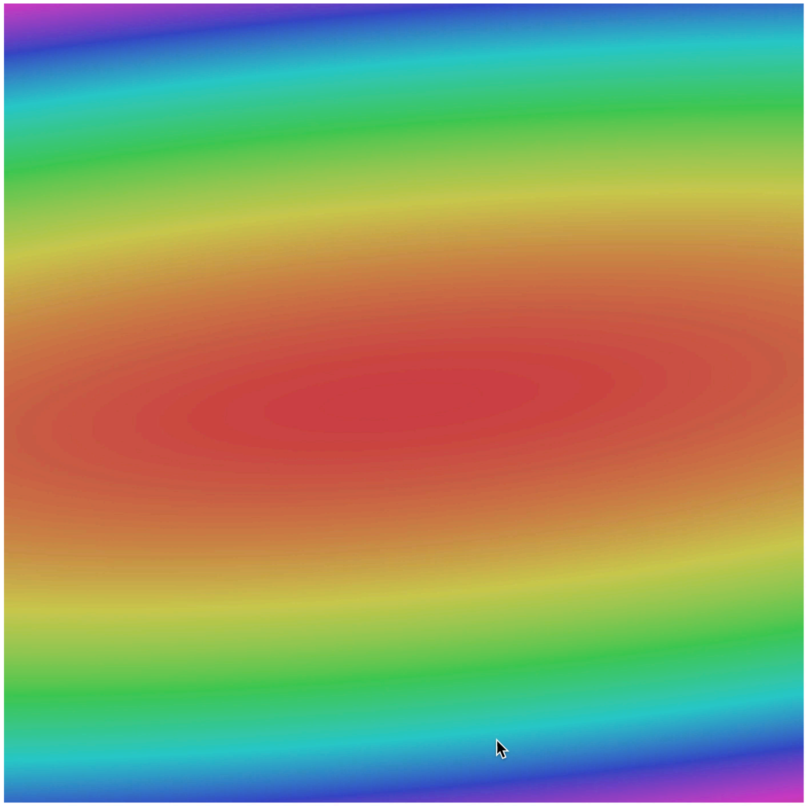
backprop

$$W^{(t)} \leftarrow W^{(t-1)} - \eta \frac{\partial L}{\partial W}$$



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





Stochastic Gradient Descent (SGD)

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

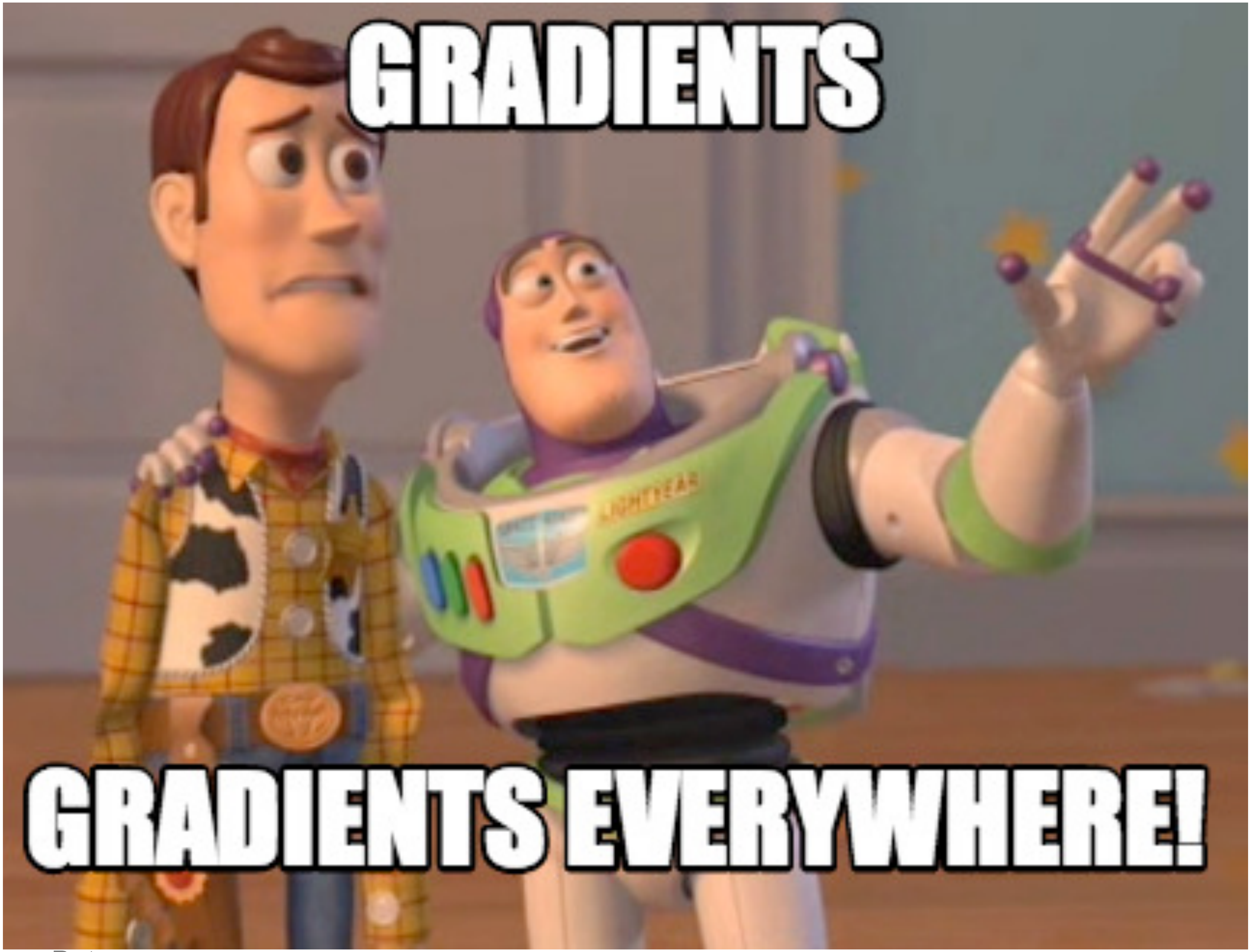
$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

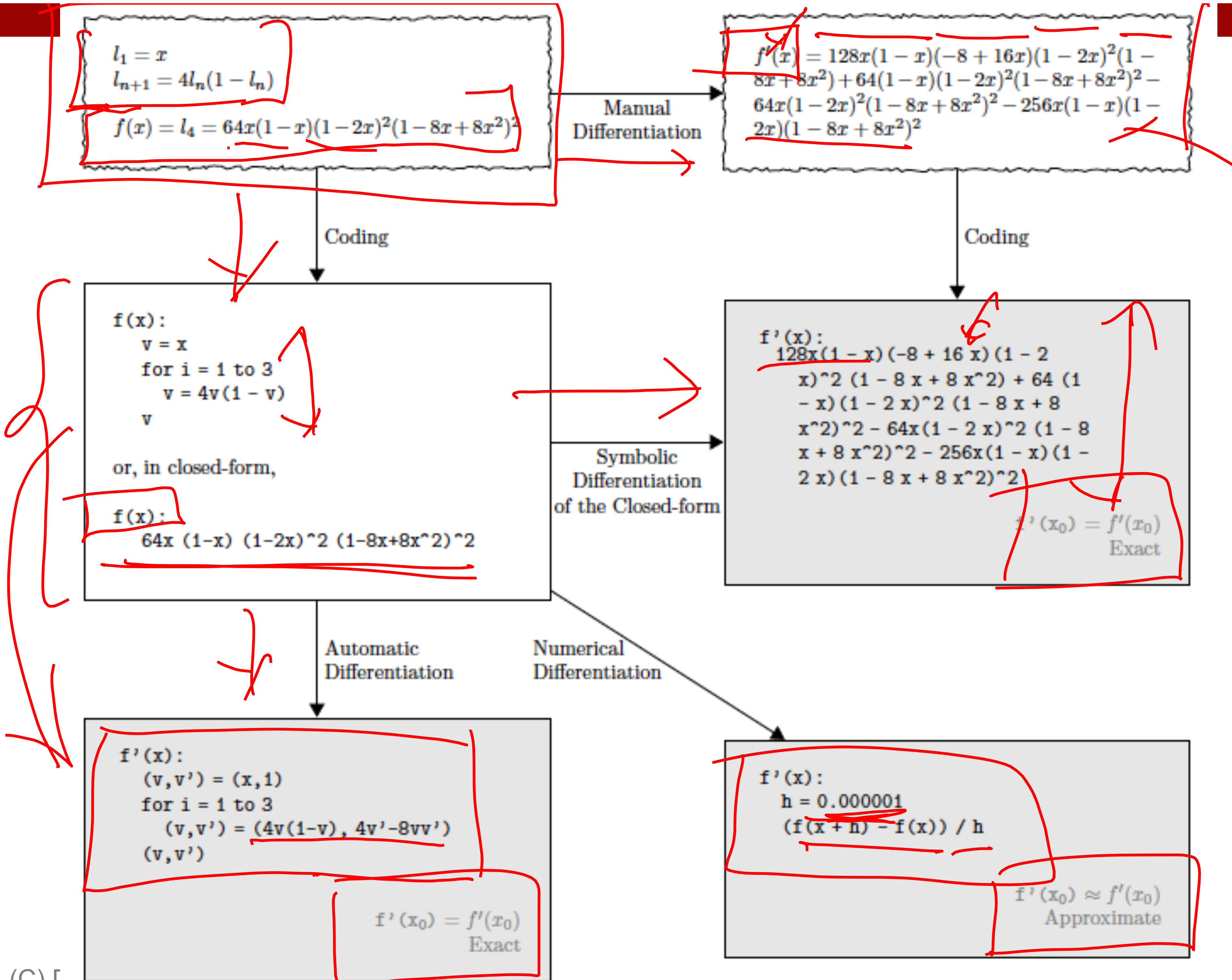
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n


GRADIENTS
GRADIENTS EVERYWHERE!

How do we compute gradients?

- Manual Differentiation
- Symbolic Differentiation
- Numerical Differentiation
- Automatic Differentiation
  - Forward mode AD
  - Reverse mode AD
    - aka “backprop”

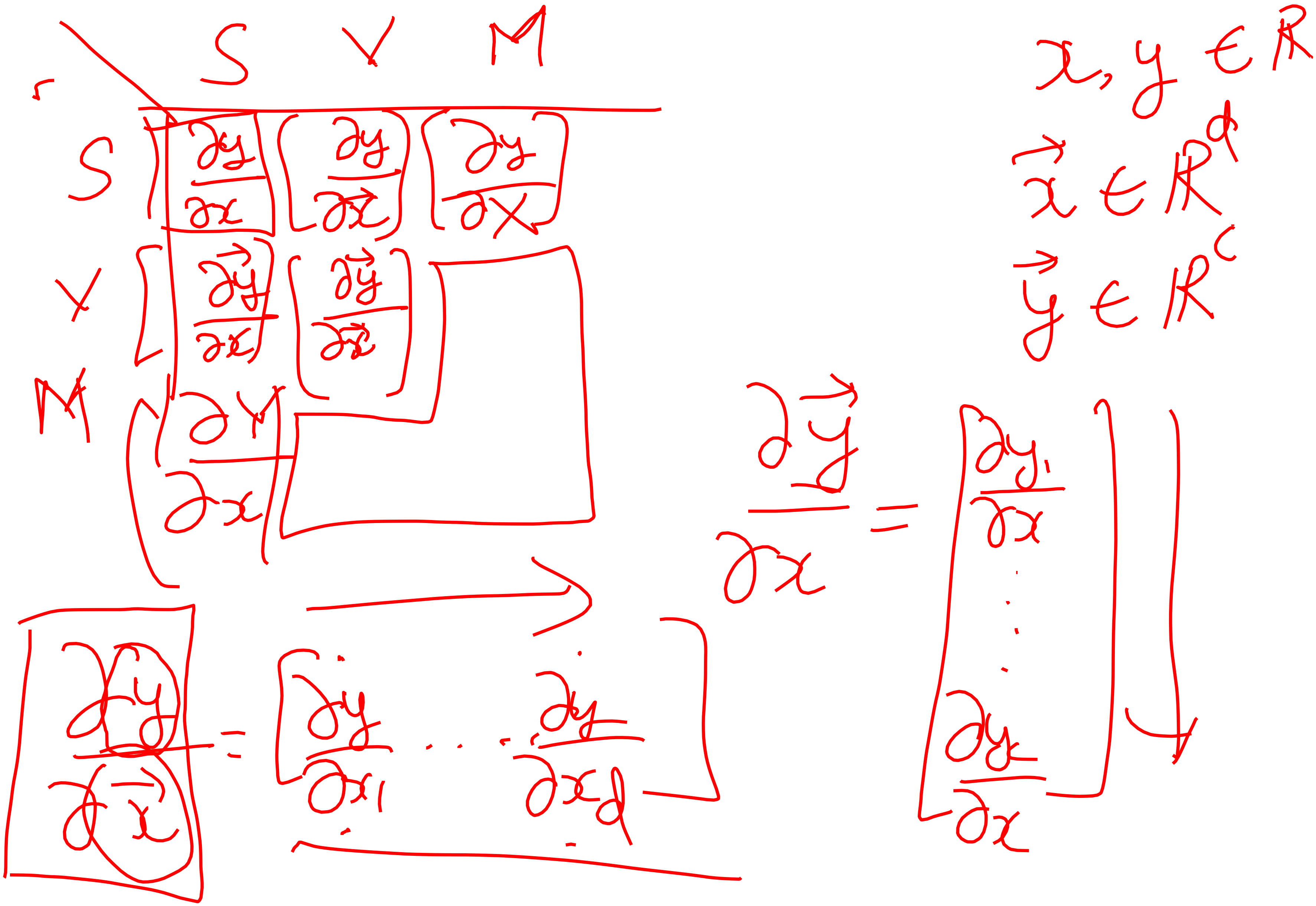
$$f(x_1, x_2) = x_1 \cdot x_2 + \sin(x_1)$$


l_1 = x
l_{n+1} = 4l_n(1 − l_n)
f(x) = l_4 = 64x(1−x)(1−2x)^2(1−8x+8x^2)^2
Manual Differentiation
f'(x) = 128x(1 − x)(−8 + 16x)(1 − 2x)^2(1 − 8x + 8x^2) + 64(1 − x)(1 − 2x)^2(1 − 8x + 8x^2)^2 − 64x(1 − 2x)^2(1 − 8x + 8x^2)^2 − 256x(1 − x)(1 − 2x)(1 − 8x + 8x^2)^2
Coding
Coding
f(x):
v = x
for i = 1 to 3
v = 4v(1 - v)
v
or, in closed-form,
f(x):
64x (1-x) (1-2x)^2 (1-8x+8x^2)^2
Symbolic Differentiation of the Closed-form
f'(x):
128x(1 - x)(-8 + 16 x)(1 - 2 x)^2 (1 - 8 x + 8 x^2) + 64 (1 - x)(1 - 2 x)^2 (1 - 8 x + 8 x^2)^2 - 64x(1 - 2 x)^2 (1 - 8 x + 8 x^2)^2 - 256x(1 - x)(1 - 2 x)(1 - 8 x + 8 x^2)^2
f'(x0) = f'(x0)
Exact
Automatic Differentiation
Numerical Differentiation
f'(x):
(v,v') = (x,1)
for i = 1 to 3
(v,v') = (4v(1-v), 4v'-8vv')
(v,v')
f'(x0) = f'(x0)
Exact
f'(x):
h = 0.000001
(f(x + h) - f(x)) / h
f'(x0) ≈ f'(x0)
Approximate

How do we compute gradients?

- Manual Differentiation
- Symbolic Differentiation
- Numerical Differentiation
- Automatic Differentiation
  - Forward mode AD
  - Reverse mode AD
    - aka “backprop”

Matrix/Vector Derivatives Notation



Matrix/Vector Derivatives Notation

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{bmatrix} & \vdots & \\ \cdots & \frac{\partial y_i}{\partial x_j} & \cdots \\ & \vdots & \end{bmatrix}_{c \times d}$$

$$\frac{\partial (\vec{w}^T \vec{x})}{\partial \vec{w}} = \left[\frac{\partial (x^T w)}{\partial w_1} \cdots \frac{\partial (\;)}{\partial w_d} \right] = [x_1 \cdots x_d] = x^T$$

Vector Derivative Example

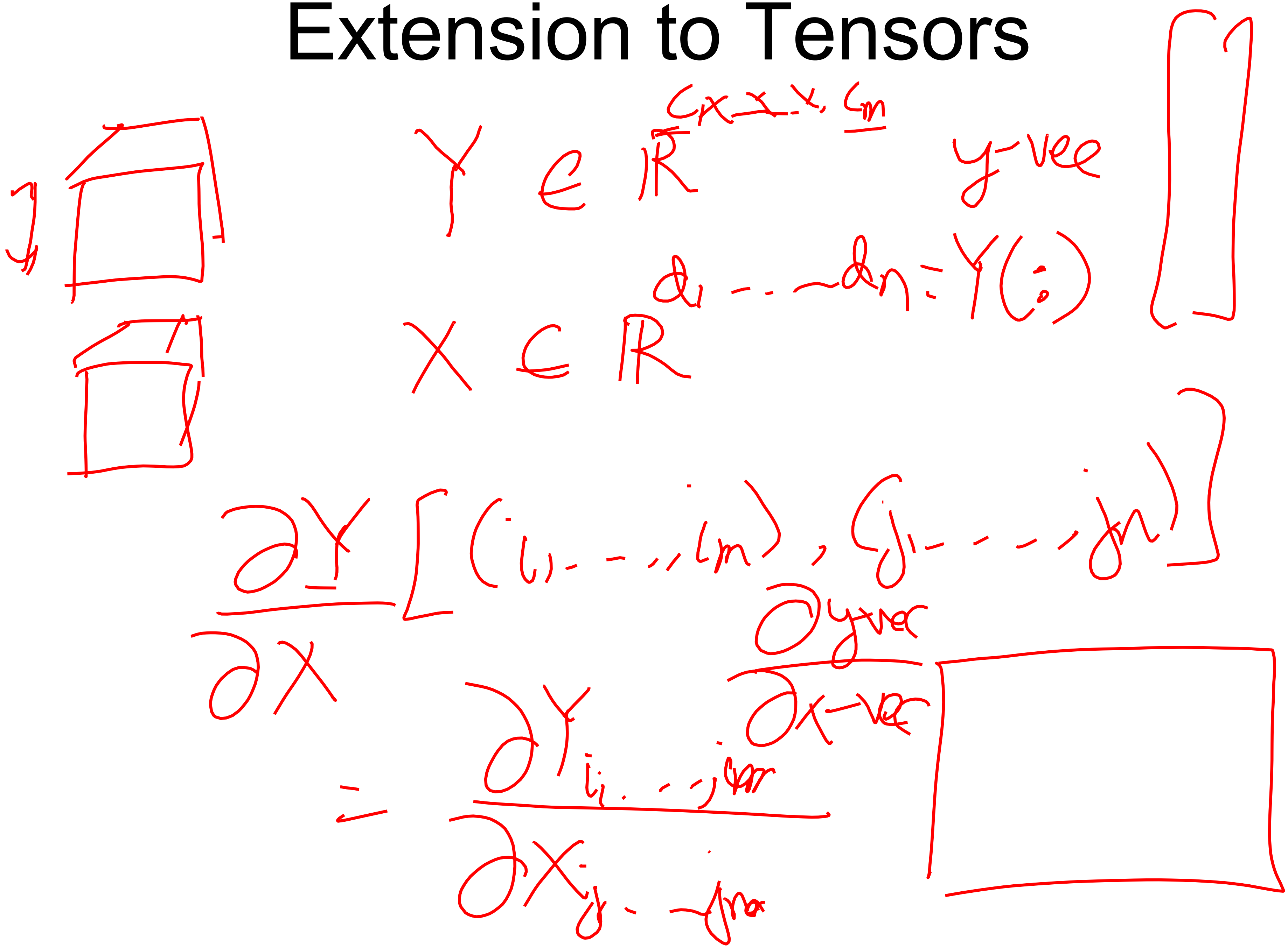
$$\frac{\partial (\vec{w}^T A \vec{w})}{\partial \vec{w}} = 2 w^T A$$

$$\vec{y} = A\vec{x}$$

$$\frac{\partial \vec{y}}{\partial \vec{x}} = A$$

$$\left[a_{ij} = \left(\frac{\partial y_i}{\partial x_j} \right) \right]$$

Extension to Tensors



Chain Rule: Composite Functions

$$L(x) = f(g(x)) = (f \circ g)(x)$$

$$(g \circ f)(x)$$

$$(g_1 \circ g_2 \circ \dots \circ g_l)(x)$$

$$(g_l \circ g_{l-1} \circ \dots \circ g_1)(x)$$

Chain Rule: Scalar Case

$x \quad z \quad y$

$$L(x) = (f \circ g)(x)$$

$$z = g(x)$$

$$y = f(z)$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x}$$

Chain Rule: Vector Case

$\vec{x} \in \mathbb{R}^d \qquad g: \mathbb{R}^d \to \mathbb{R}^m \qquad f(g(\vec{x}))$

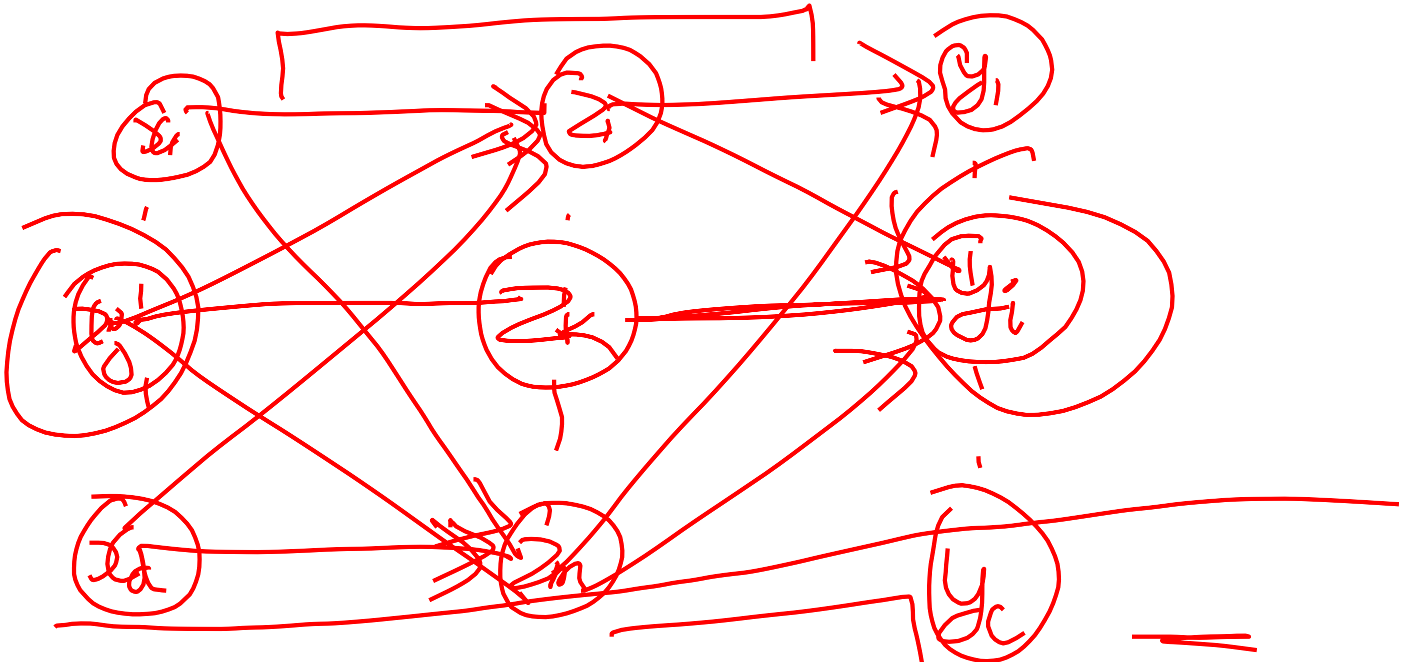
$\vec{z} \in \mathbb{R}^m \qquad f: \mathbb{R}^m \to \mathbb{R}^c$

$\vec{y} \in \mathbb{R}^c$

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \frac{\partial \vec{y}}{\partial \vec{z}} \cdot \frac{\partial \vec{z}}{\partial \vec{x}}$$

$$i \left[\frac{\partial y_i}{\partial x_j} \right]_{c \times d} = i \left[\cdots \frac{\partial y_i}{\partial z_k} \cdots \right] \left[\begin{matrix} \vdots \\ \frac{\partial z_k}{\partial x_j} \\ \vdots \end{matrix} \right]$$

Chain Rule: Jacobian view



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

Chain Rule: Tensors