CS 4803 / 7643: Deep Learning

Topics:

- Optimization
- Computing Gradients

Dhruv Batra

Georgia Tech

Administrativia

- HW1 Reminder
  - Due: 09/26, 11:55pm
  - https://www.cc.gatech.edu/classes/AY2020/cs7643_fall/Z3o9P26CwTPZZMDXyWYDj3/hw1.pdf
  - https://www.cc.gatech.edu/classes/AY2020/cs7643_fall/Z3o9P26CwTPZZMDXyWYDj3/hw1-q8/
  - https://evalai.cloudcv.org/web/challenges/challenge-page/431/leaderboard/1200

Recap from last time

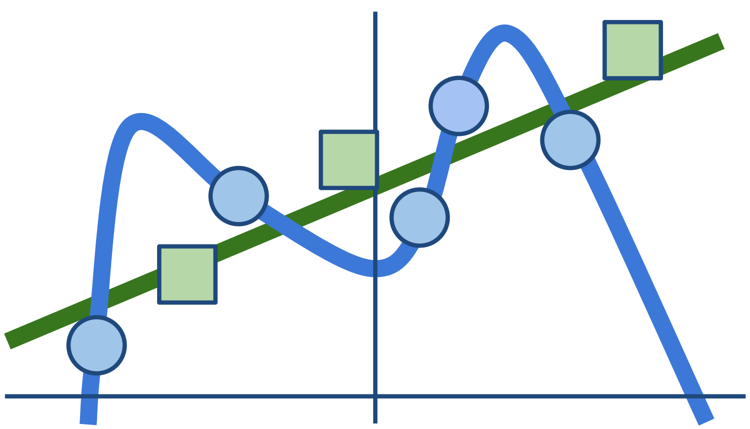
Regularization

λ = regularization strength (hyperparameter)

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

Data loss: Model predictions should match training data

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



Occam's Razor:
"Among competing hypotheses, the simplest is the best"
William of Ockham, 1285 - 1347

Regularization

λ = regularization strength (hyperparameter)

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

Data loss: Model predictions should match training data

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

Simple examples

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

More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

So far: Linear Classifiers

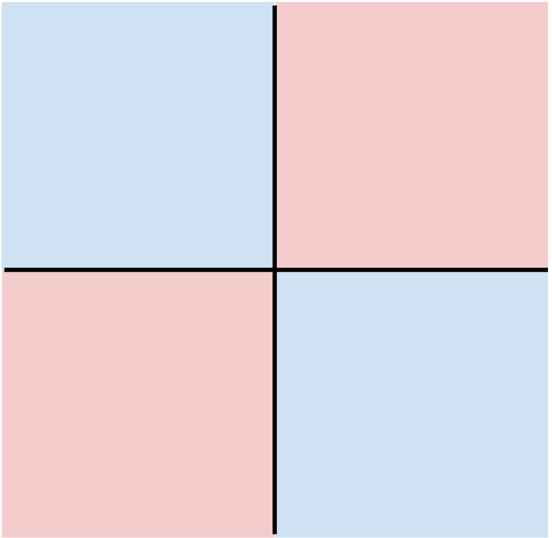
Class scores

$$f(x) = Wx$$

Hard cases for a linear classifier

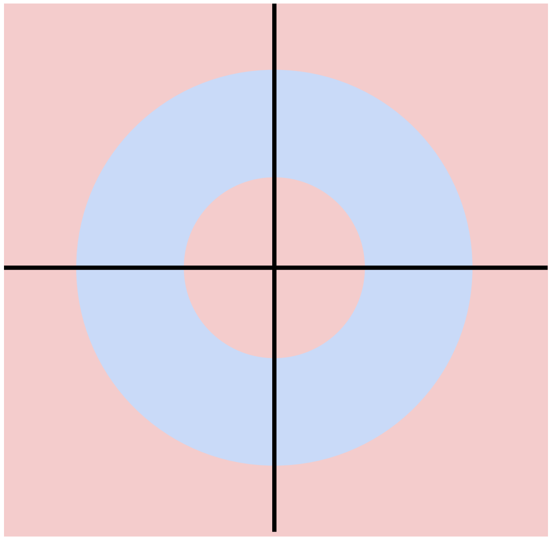
Class 1:
First and third quadrants

Class 2:
Second and fourth quadrants



Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



Class 1:
Three modes

Class 2:
Everything else

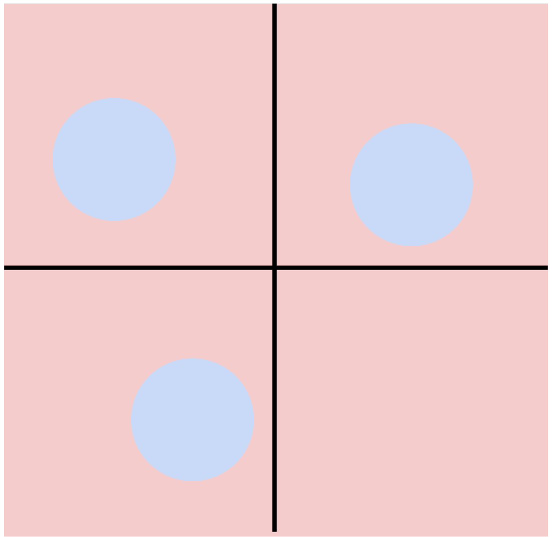
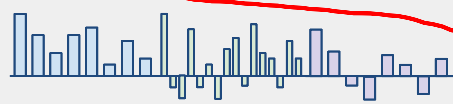


Image features vs Neural Nets

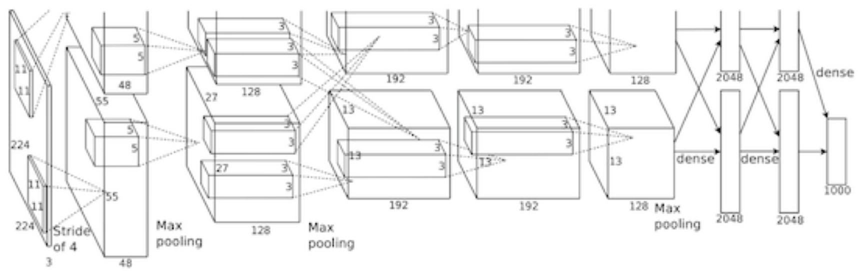
Feature Extraction



f

10 numbers giving scores for classes

training



10 numbers giving scores for classes

training

Neural networks: without the brain stuff

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

Neural networks: without the brain stuff

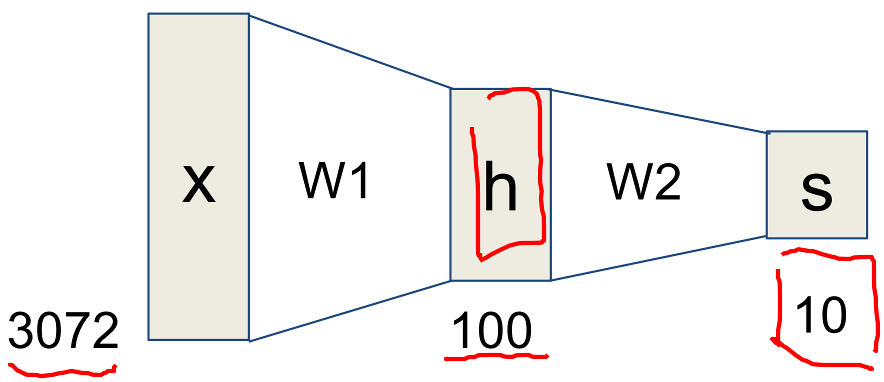
(**Before**) Linear score function: $f = Wx$

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

Neural networks: without the brain stuff

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

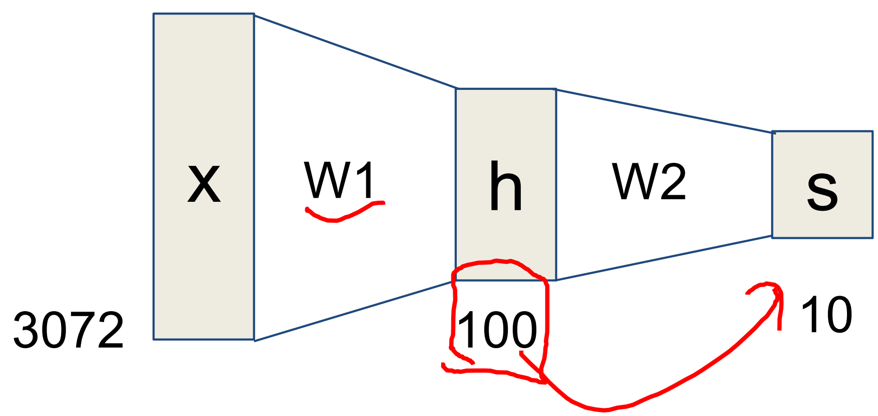
(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

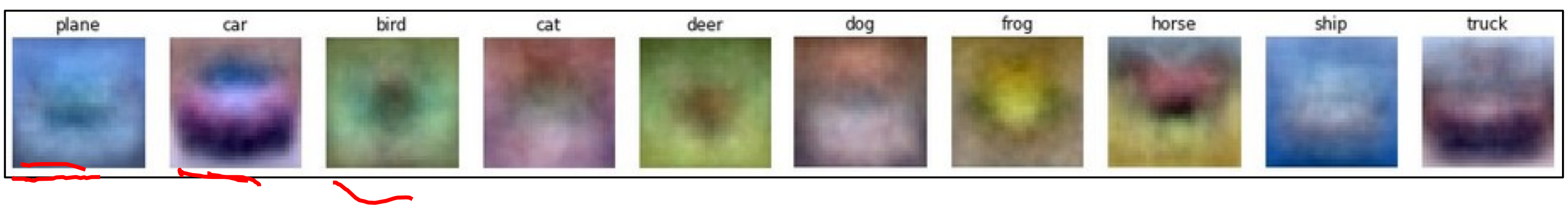


Neural networks: without the brain stuff

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

or 3-layer Neural Network

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

Multilayer Networks

MLP

- Cascaded "neurons"
- The output from one layer is the input to the next
- Each layer has its own sets of weights

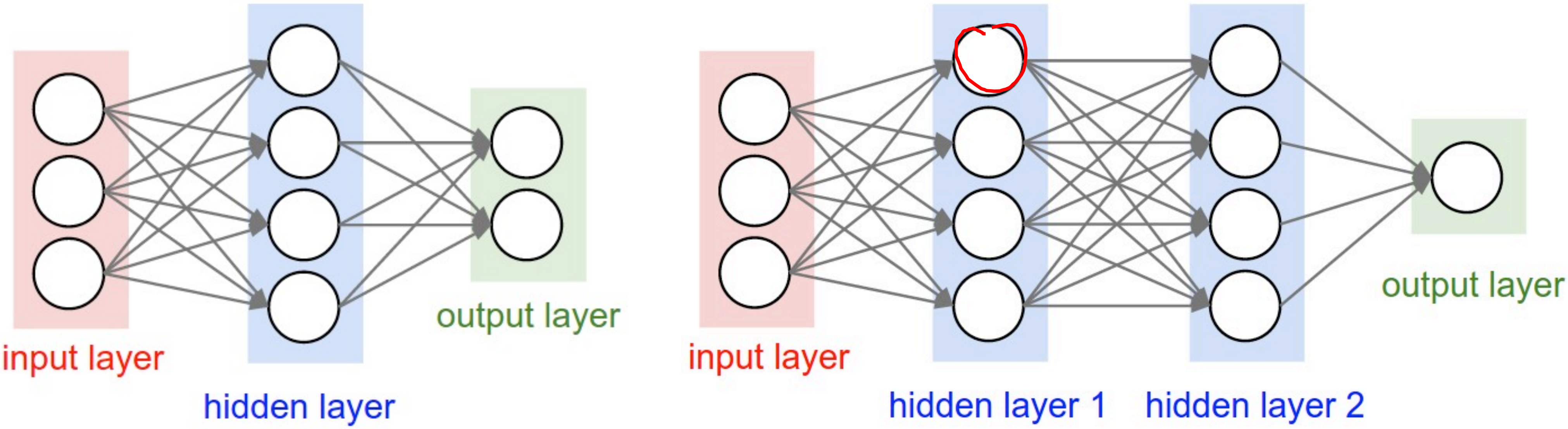
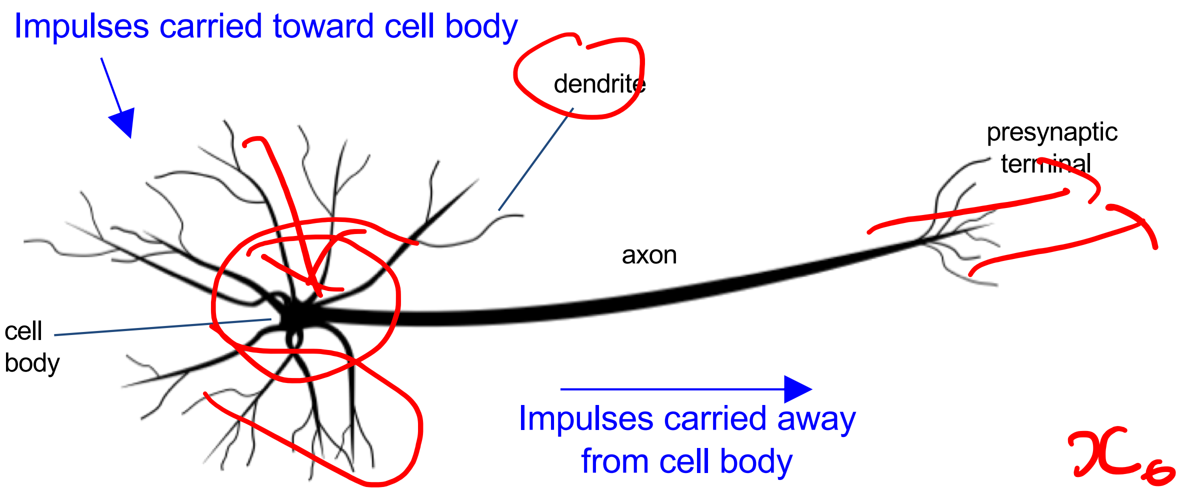
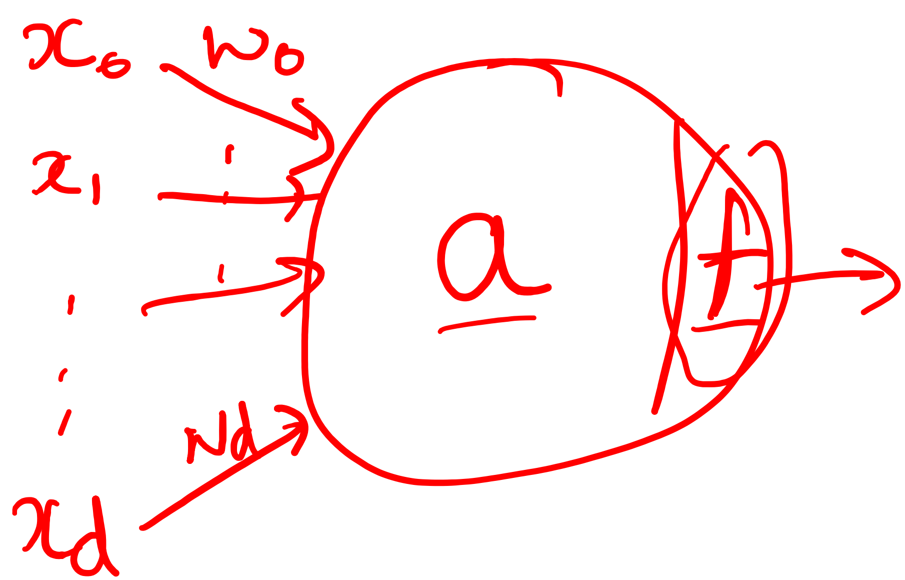


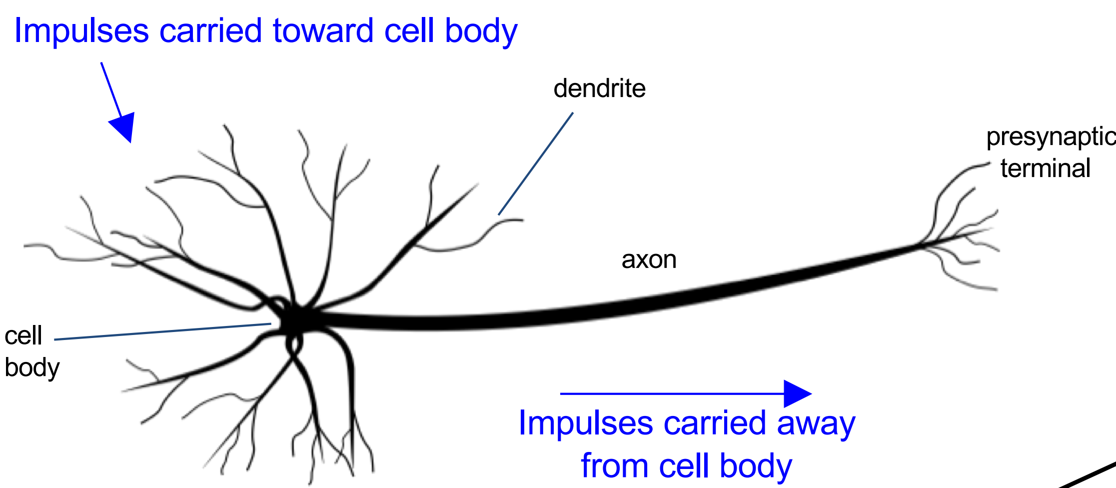
Image Credit: Andrej Karpathy, CS231n

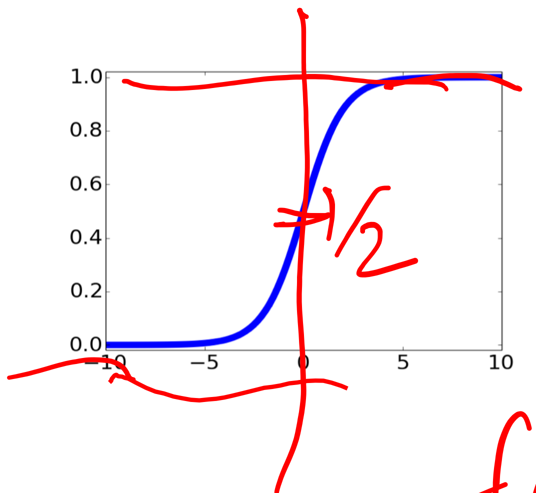


$$a = \sum_j w_j x_j = \vec{w}^T \vec{x}$$



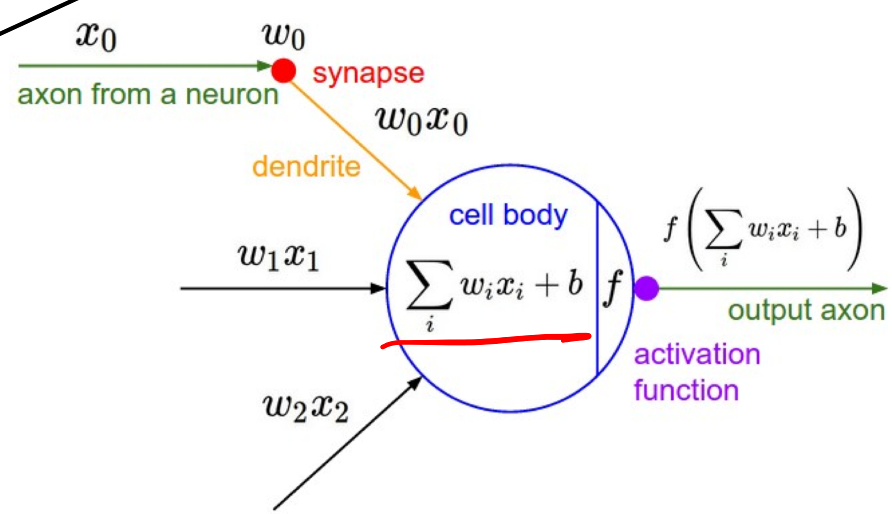
$$y = f(a)$$





sigmoid activation function

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

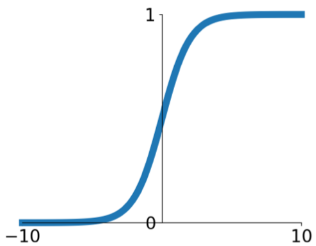


$f(a)$ $\frac{1}{1+e^{-a}}$

Activation functions

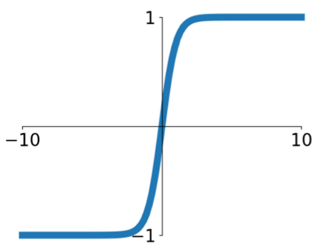
Sigmoid

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



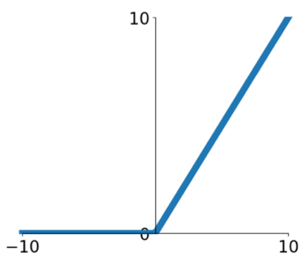
tanh

$\tanh(x)$



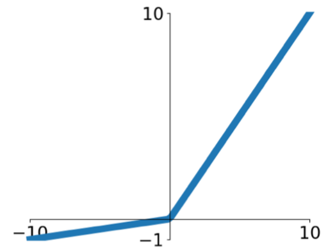
ReLU

$\boxed{\max(0, x)}$



Leaky ReLU

$\max(0.1x, x)$

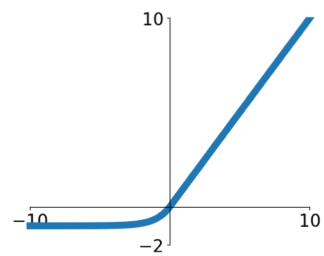


Maxout

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

ELU

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



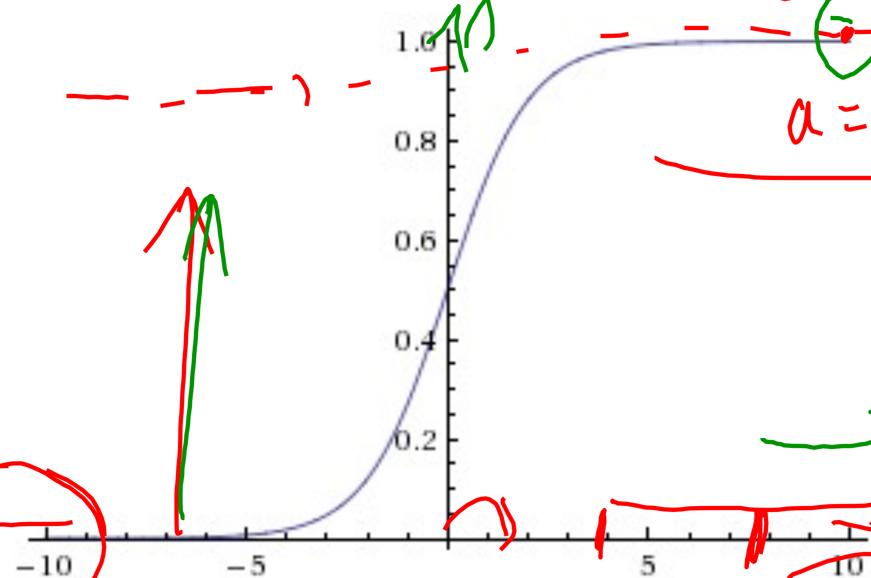
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Activation Functions

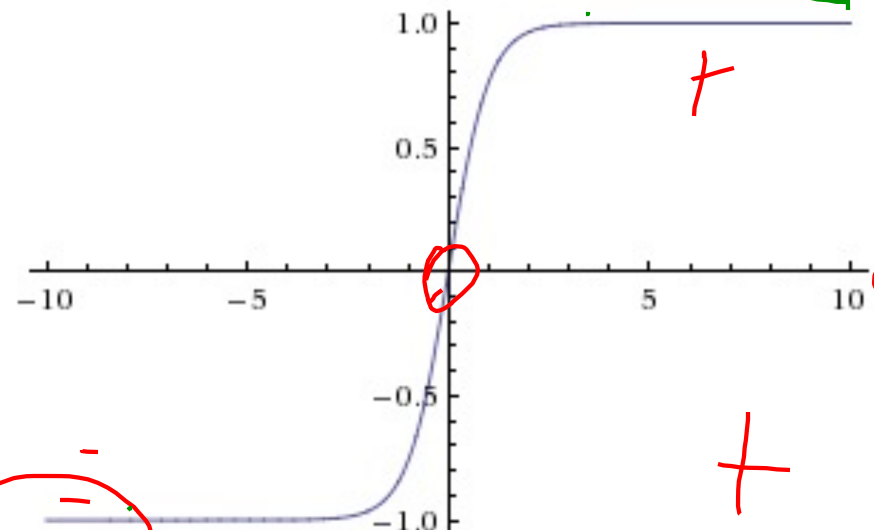
$\sigma(a)$

- sigmoid vs tanh

$\frac{1}{1+e^{-a}}$ $\quad 2\sigma(2a) - 1$



$a = 10^5$



$\sigma(\Sigma\sigma(\Sigma\sigma(x)))$

A quick note

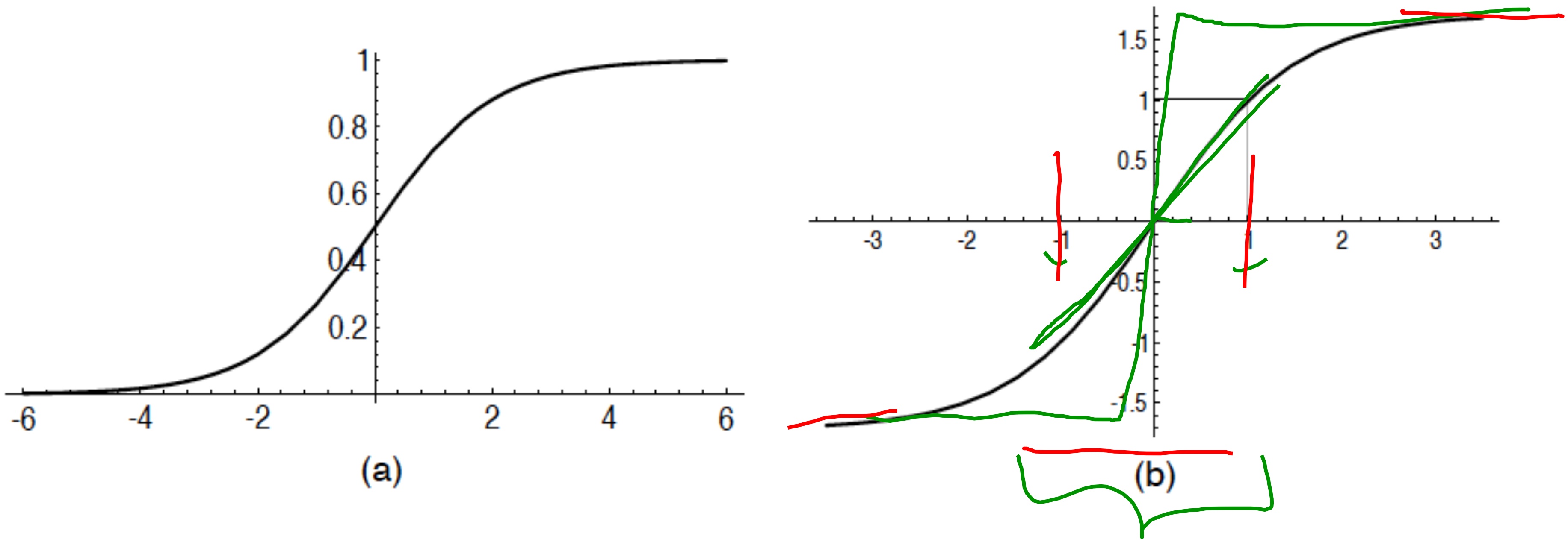
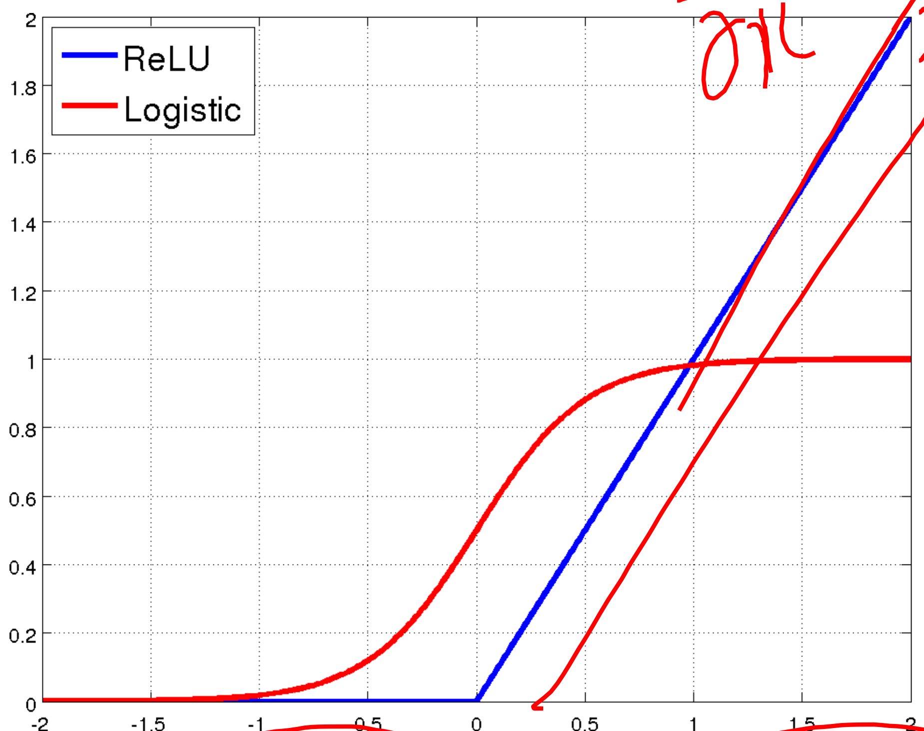


Fig. 4. (a) Not recommended: the standard logistic function, $f(x) = 1/(1 + e^{-x})$. (b) Hyperbolic tangent, $f(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$.

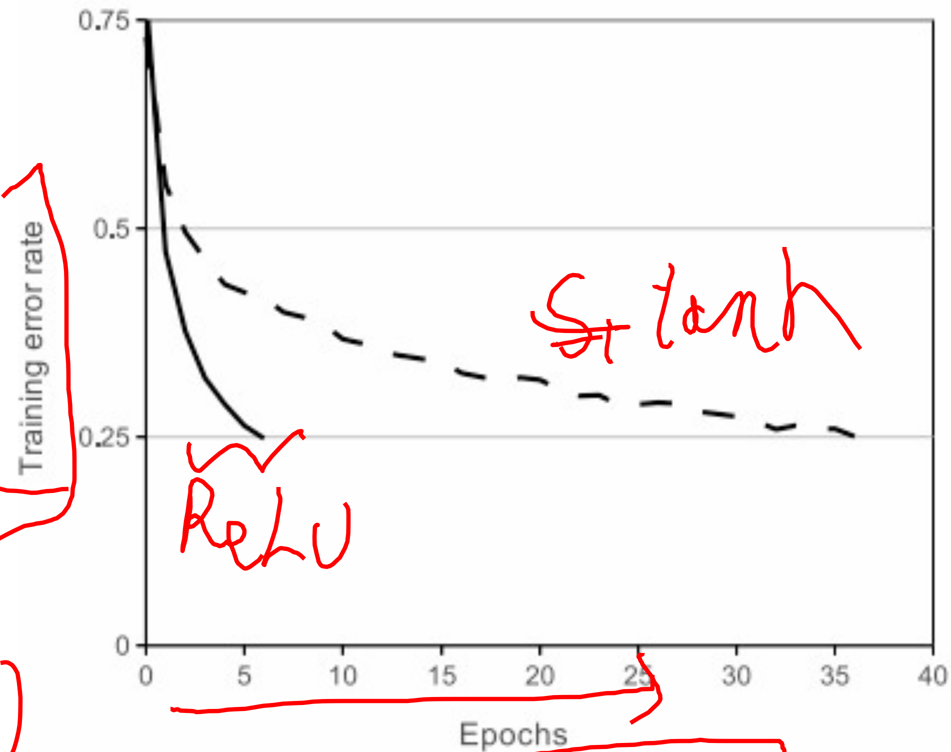
Rectified Linear Units (ReLU)

$\frac{\partial h}{\partial x} = x$

$=1$



0



tanh

ReLU

[Krizhevsky et al., NIPS12]

Plan for Today

- Optimization
- Computing Gradients

Optimization

Supervised Learning

- Input: x (images, text, emails…)
- Output: y (spam or non-spam…)

- (Unknown) Target Function
  - f: X → Y (the "true" mapping / reality)

- Data
  - $(x_1,y_1), (x_2,y_2), \ldots, (x_N,y_N)$

- Model / Hypothesis Class
  - {h: X → Y}
  - e.g. $y = h(x) = \text{sign}(w^T x)$

Linear → NN

- Loss Function
  - How good is a model wrt my data D?

hinge

softmax CE

- Learning = Search in hypothesis space
  - Find best h in model class.

Demo Time

- https://playground.tensorflow.org

$$\min_{\vec{w}} L(\vec{w}, \underline{D})$$

$$\underline{L(\vec{w})} = \frac{1}{N}\sum_i L_i(w)$$

Strategy: **Follow the slope**

Strategy: **Follow the slope**

In 1-dimension, the derivative of a function:

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

$$\frac{\partial f(x_1, \ldots x_d)}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots x_i + h, \ldots x_d) - f(x_1, \ldots x_d)}{h}$$

$$\nabla f = \left[\frac{\partial f}{\partial x_1} \cdots \frac{\partial f}{\partial x_d} \right]$$ Gradient

Strategy: **Follow the slope**

In 1-dimension, the derivative of a function:

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

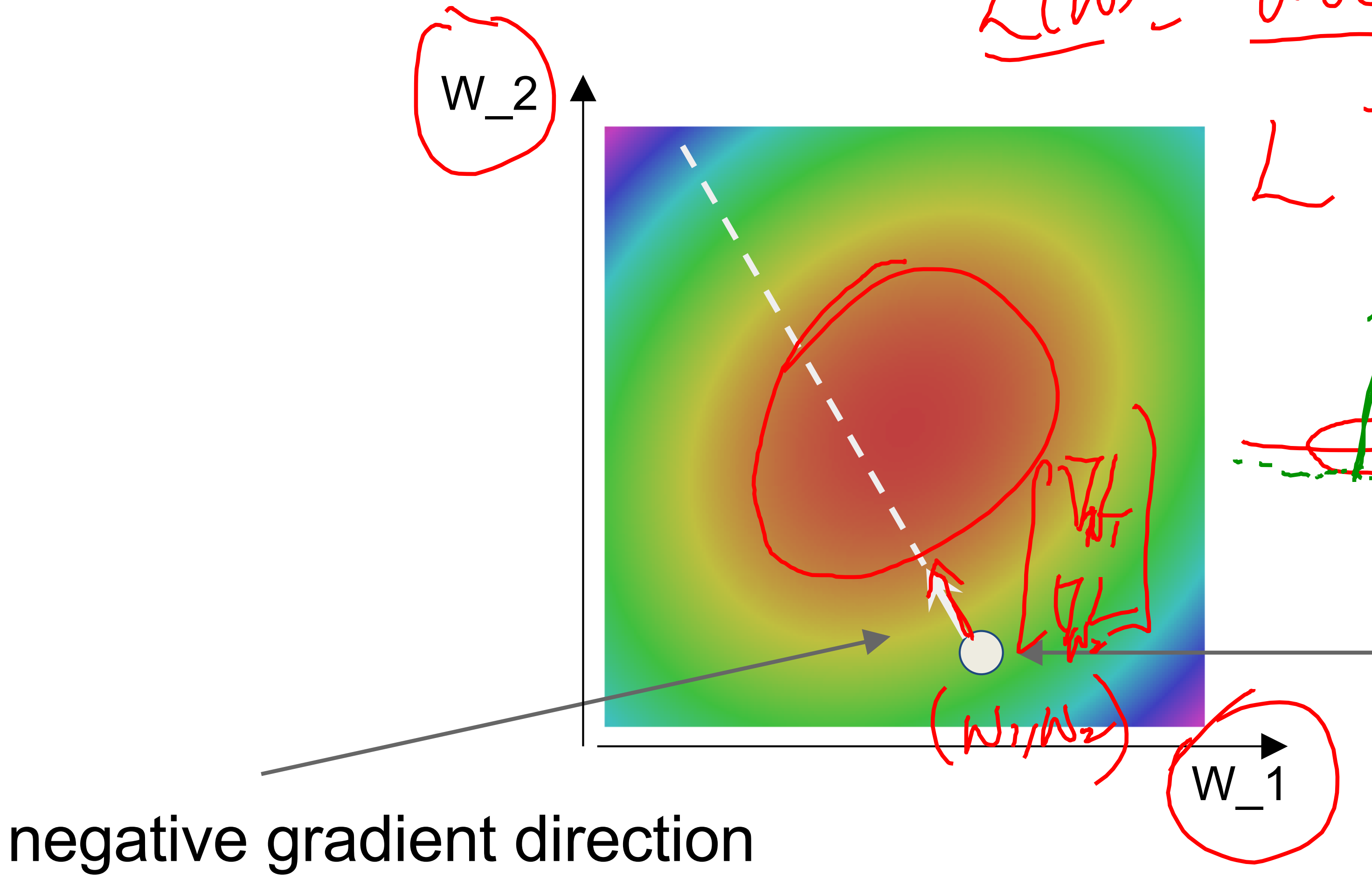
In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

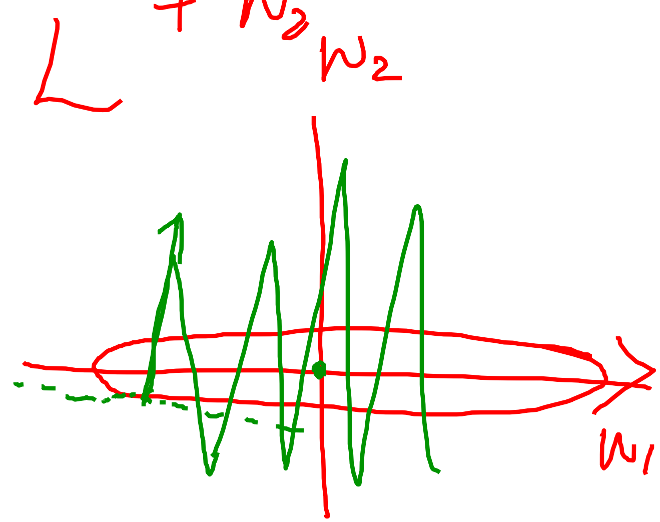
The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

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

http://demonstrations.wolfram.com/VisualizingTheGradientVector/

$$L(\vec{w}) = 0.00001\, w_1^2 + w_2^2$$





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

$\min_{\vec{w}} L(\vec{w})$

Gradient Descent

```
# Vanilla Gradient Descent

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

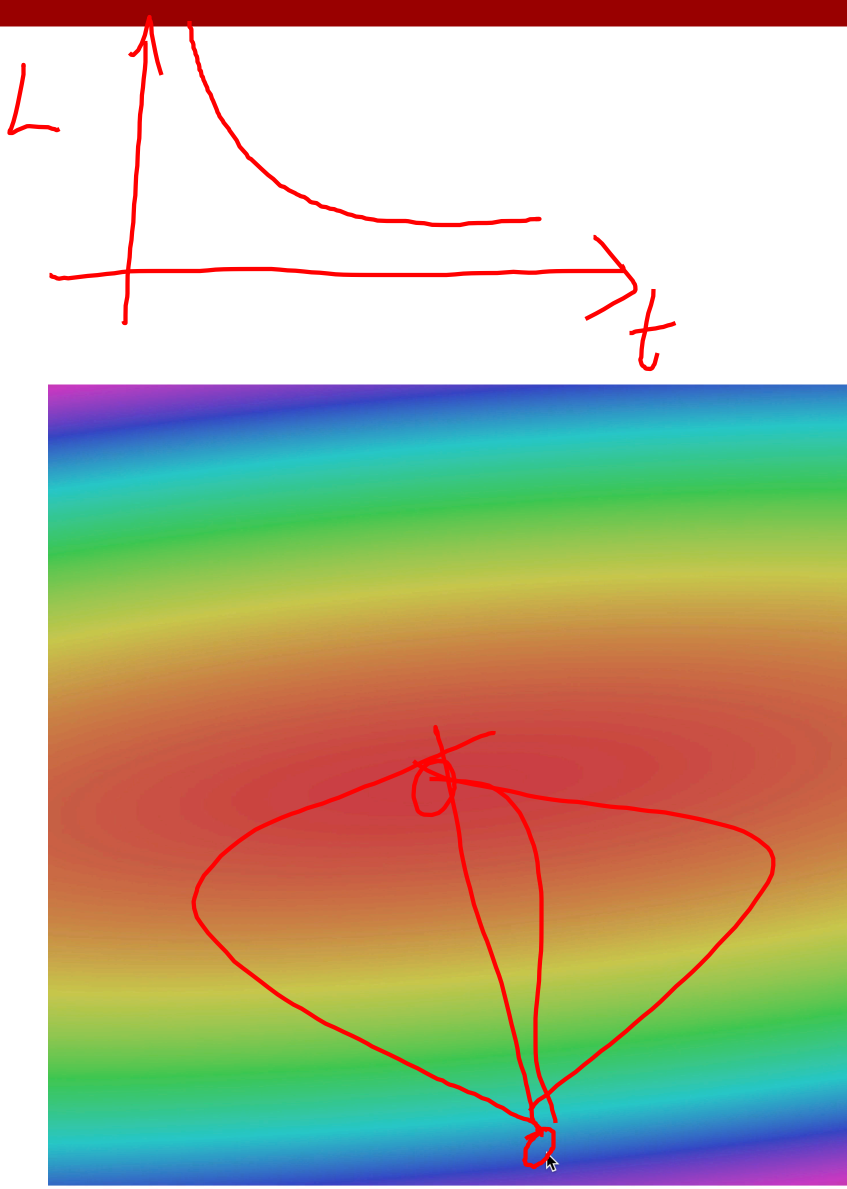
$\vec{w}^{(0)}$ = Initialize

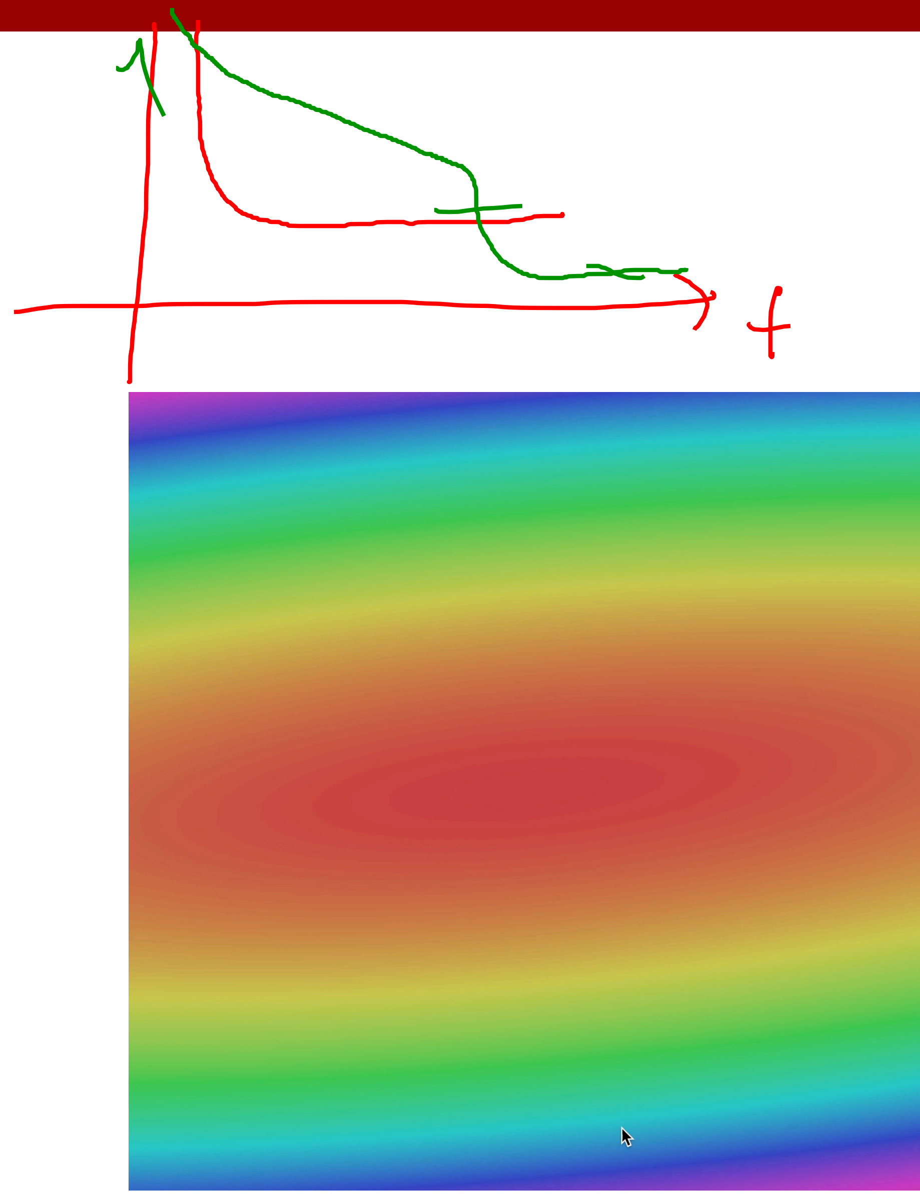
for $t = 1, \ldots,$ tired

$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \boxed{\eta}\, \nabla_{\vec{w}} L(\vec{w}^{(t)})$

η: 0.001

step size / learning rate





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

Gradient Descent has a problem

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

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

Full sum expensive
when N is large!

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

N = 1000

```
# 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
```

Stochastic Gradient Descent (SGD)

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

$$\boxed{\nabla_W L(W)} = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) = \text{mean-over-N}\left(\nabla_W L_i\right)$$

$$\boxed{\approx} \ \text{mean-over-B}(\ \cdot\ )$$

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

Stochastic Gradient Descent (SGD)

$$\boxed{L(W)} = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) \approx \mathbb{E}_{x,y\sim p^*}\left[L(w,x,y)\right]$$

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

$$\nabla_w \mathbb{E}\left[L(w,x,y)\right]$$

$$= \boxed{\mathbb{E}_{p^*}\left[\nabla_w L(w,x,y)\right]}$$

B

Stochastic Gradient Descent (SGD)

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

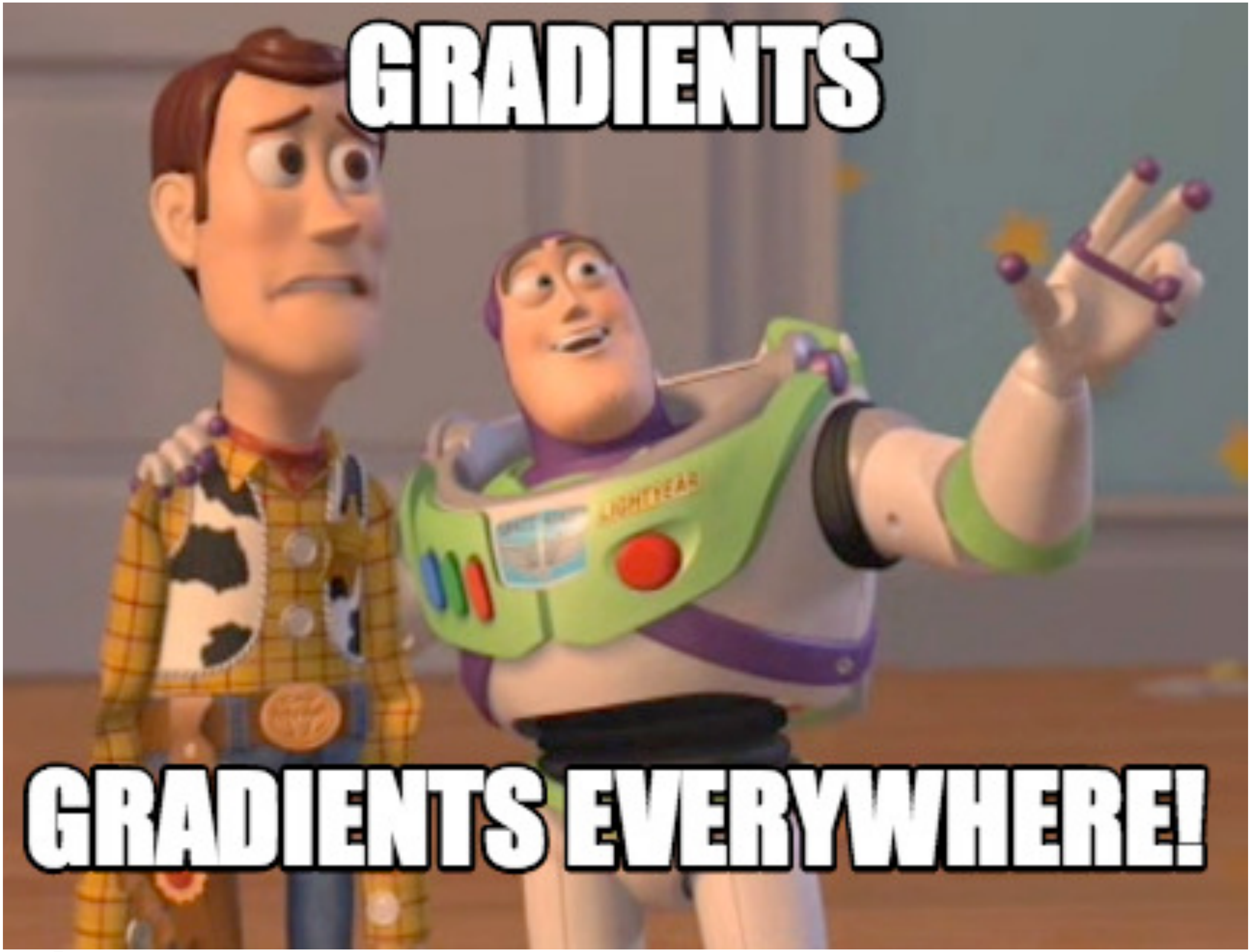
$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

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
```


GRADIENTS
GRADIENTS EVERYWHERE!

How do we compute gradients?

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

$$l_1 = x$$
$$l_{n+1} = 4l_n(1 - l_n)$$
$$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

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

```
f(x):
  v = x
  for i = 1 to 3
    v = 4*v*(1 - v)
  return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

Coding

Symbolic Differentiation of the Closed-form

```
f'(x):
  return 128*x*(1 - x)*(-8 + 16*x)
    *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
    + 64*(1 - x)*((1 - 2*x)^2)*((1
    - 8*x + 8*x*x)^2) - (64*x*(1 -
    2*x)^2)*(1 - 8*x + 8*x*x)^2 -
    256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
    + 8*x*x)^2
```

f'(x_0) $= f'(x_0)$
Exact

Automatic Differentiation

```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

f'(x_0) $= f'(x_0)$
Exact

Numerical Differentiation

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

f'(x_0) $\approx f'(x_0)$
Approximate

$l_1 = x$

$l_{n+1} = 4l_n(1 - l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

L(x)

Manual Differentiation

∇L(x)

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

Coding

```
f(x):
  v = x
  for i = 1 to 3
    v = 4*v*(1 - v)
  return v
```

L(x)

or, in closed-form,

```
f(x):
 return 64*x*(1-x)*((1-2*x)^2)
   *(1-8*x+8*x*x)^2
```

Symbolic Differentiation of the Closed-form

Coding

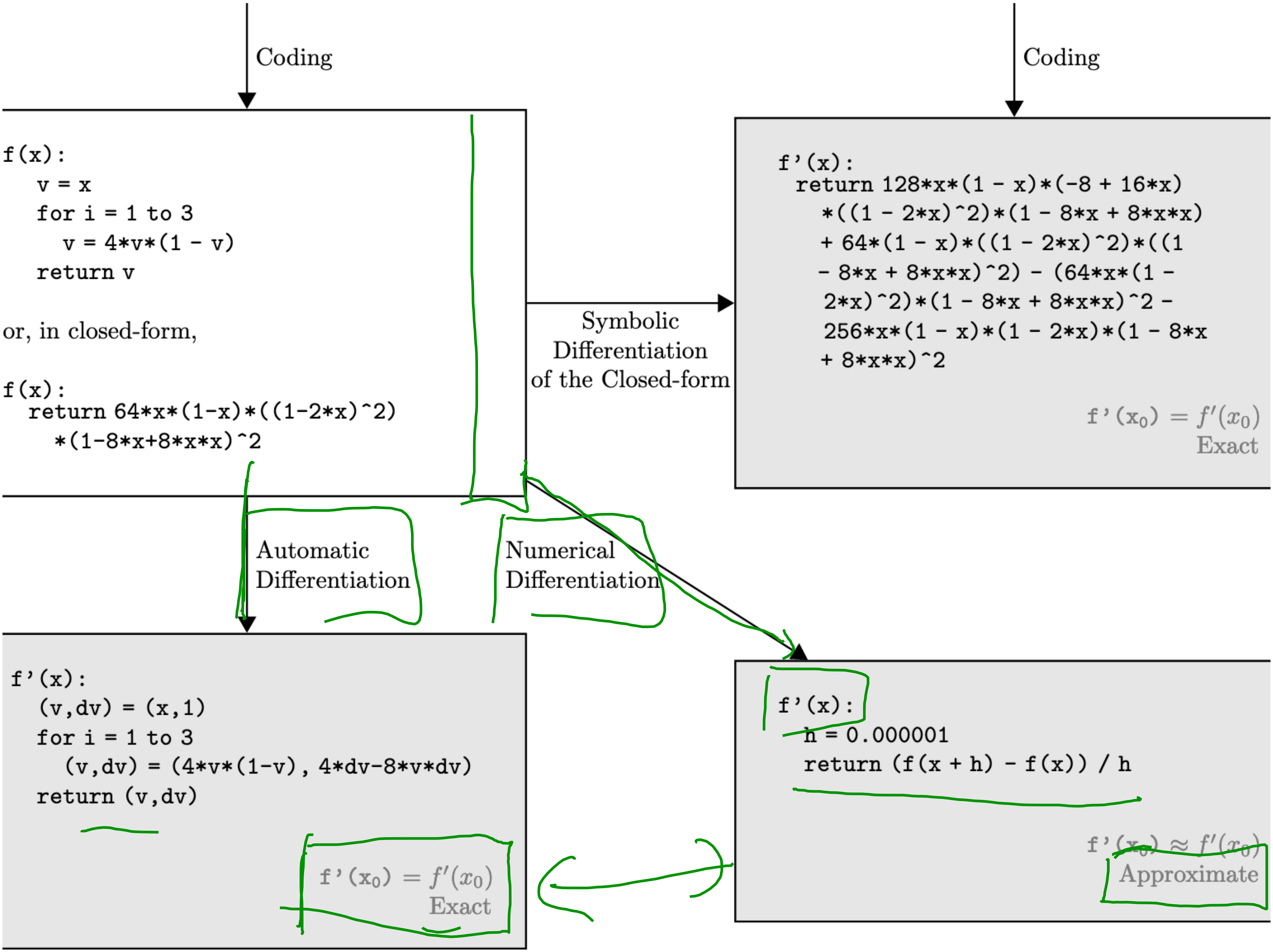
```
f'(x):
 return 128*x*(1 - x)*(-8 + 16*x)
   *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
   + 64*(1 - x)*((1 - 2*x)^2)*((1
   - 8*x + 8*x*x)^2) - (64*x*(1 -
   2*x)^2)*(1 - 8*x + 8*x*x)^2 -
   256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
   + 8*x*x)^2
```

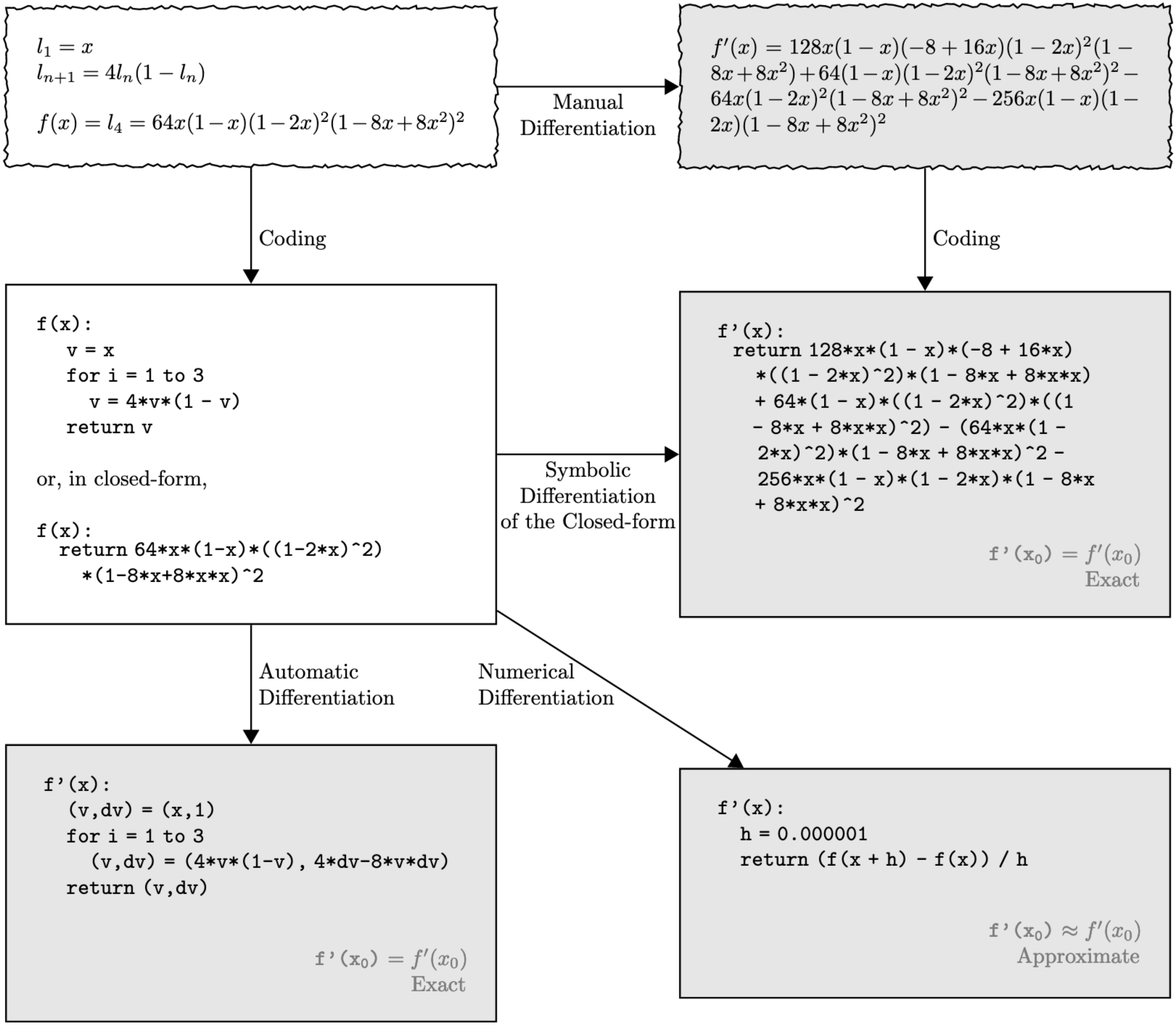
f'(x_0) $= f'(x_0)$
Exact

Automatic Differentiation

Numerical Differentiation

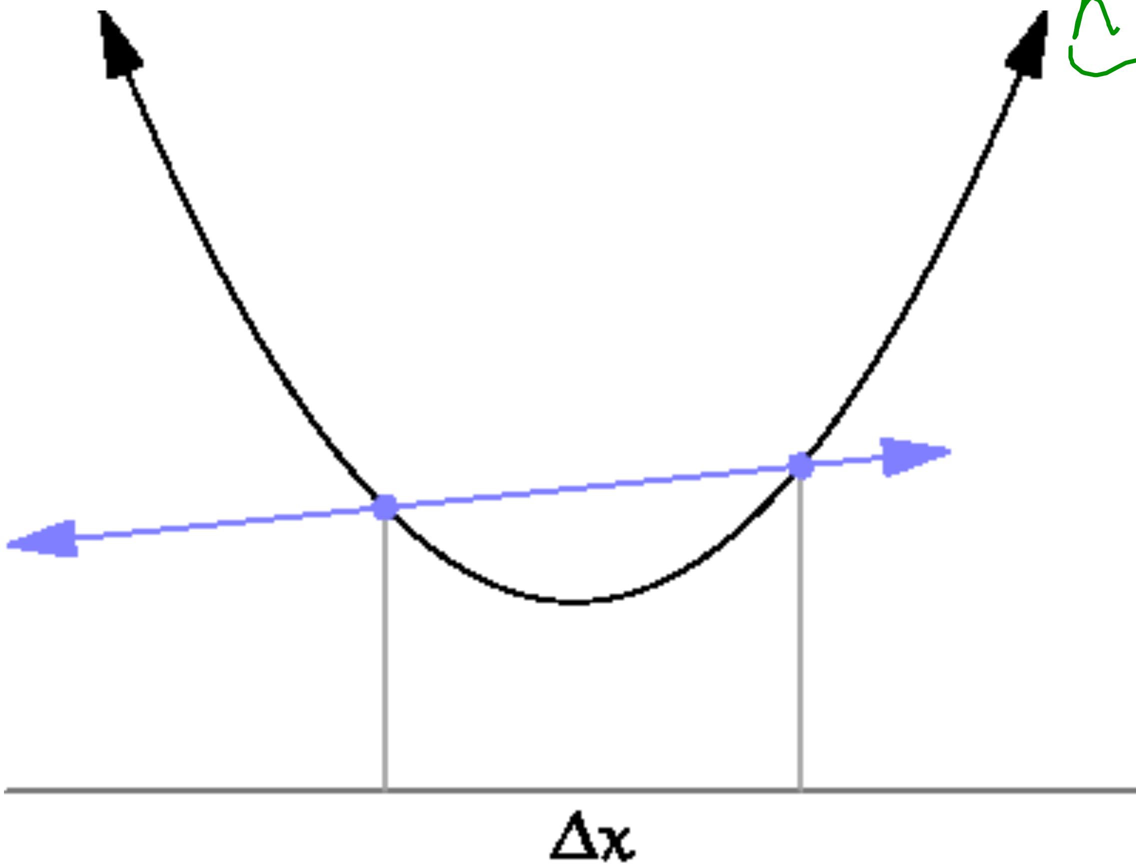
```
f'(x):
```


Coding
Coding
f(x):
v = x
for i = 1 to 3
v = 4*v*(1 - v)
return v
or, in closed-form,
f(x):
return 64*x*(1-x)*((1-2*x)^2)
*(1-8*x+8*x*x)^2
Symbolic
Differentiation
of the Closed-form
f'(x):
return 128*x*(1 - x)*(-8 + 16*x)
*((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
+ 64*(1 - x)*((1 - 2*x)^2)*((1
- 8*x + 8*x*x)^2) - (64*x*(1 -
2*x)^2)*(1 - 8*x + 8*x*x)^2 -
256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
+ 8*x*x)^2
f'(x0) = f′(x0)
Exact
Automatic
Differentiation
Numerical
Differentiation
f'(x):
(v,dv) = (x,1)
for i = 1 to 3
(v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
return (v,dv)
f'(x0) = f′(x0)
Exact
f'(x):
h = 0.000001
return (f(x + h) - f(x)) / h
f'(x0) ≈ f′(x0)
Approximate


$l_1 = x$
$l_{n+1} = 4l_n(1 - l_n)$
$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$
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 = 4*v*(1 - v)
return v
or, in closed-form,
f(x):
return 64*x*(1-x)*((1-2*x)^2)
*(1-8*x+8*x*x)^2
Symbolic Differentiation of the Closed-form
f'(x):
return 128*x*(1 - x)*(-8 + 16*x)
*((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
+ 64*(1 - x)*((1 - 2*x)^2)*((1
- 8*x + 8*x*x)^2) - (64*x*(1 -
2*x)^2)*(1 - 8*x + 8*x*x)^2 -
256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
+ 8*x*x)^2
f'(x$_0$) = $f'(x_0)$
Exact
Automatic Differentiation
Numerical Differentiation
f'(x):
(v,dv) = (x,1)
for i = 1 to 3
(v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
return (v,dv)
f'(x$_0$) = $f'(x_0)$
Exact
f'(x):
h = 0.000001
return (f(x + h) - f(x)) / h
f'(x$_0$) $\approx$ $f'(x_0)$
Approximate

How do we compute gradients?

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



$\lim_{h \to 0}$

$$\left[\frac{\partial L}{\partial w_1} \cdots \frac{\partial L}{\partial w_J}\right]$$

current W:

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

loss 1.25347

$L(W)$

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

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

loss 1.25347

$L(\vec{w})$

W + h (first dim)**:**

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

loss 1.25322

$L(w_1 + h, w_2 \ldots w_d)$

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

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

?,
?,…]

current W:

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

loss 1.25347

W + h (second dim)**:**

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

loss 1.25353

gradient dW:

[-2.5,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (second dim)**:**

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

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

?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

current W:

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

loss 1.25347

d

W + h (third dim):

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

loss 1.25347

$2x \quad \underline{L(W)}$

gradient dW:

[-2.5,
0.6,
0,
?,

(1.25347 - 1.25347)/0.0001
= 0

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

?,…]

$2d \quad \underline{|L(W)|}$

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

Numerical vs Analytic Gradients

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

Numerical gradient: slow :(, approximate :(, easy to write :)
Analytic gradient: fast :), exact :), error-prone :(

In practice: Derive analytic gradient, check your implementation with numerical gradient.
This is called a **gradient check.**

How do we compute gradients?

- Analytic or "Manual" Differentiation
- Symbolic Differentiation
- Numerical Differentiation
- Automatic Differentiation
  - Forward mode AD
  - Reverse mode AD
    - aka "backprop"

Vector/Matrix Derivatives Notation

$$\begin{array}{c|ccc} & S & V & M \\ \hline S & \frac{\partial y}{\partial x} & \frac{\partial y}{\partial \vec{x}} & \frac{\partial y}{\partial X} \\ V & \frac{\partial \vec{y}}{\partial x} & \frac{\partial \vec{y}}{\partial \vec{x}} & \\ M & \frac{\partial Y}{\partial x} & & \end{array}$$

tensor

First = num

$$\frac{\partial \vec{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \vdots \\ \frac{\partial y_c}{\partial x} \end{bmatrix}_{c \times 1}$$

$x, y \in \mathbb{R}^1$

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

$\vec{x} \in \mathbb{R}^d \quad \vec{y} \in \mathbb{R}^c$

$X, Y \in \mathbb{R}^{m \times n}$

num = dim1 = col

$$\frac{\partial y}{\partial \vec{x}} = \left[\frac{\partial y}{\partial x_1} \; \cdots \; \frac{\partial y}{\partial x_d}\right]$$

[Gradient]

Vector/Matrix Derivatives Notation

$$\frac{\partial \vec{y}}{\partial \vec{x}} = \begin{array}{cc} & \qquad\qquad\quad j \\ i & \begin{bmatrix} \partial & \frac{\partial y_i}{\partial x_j} \\ \frac{\partial y_1}{\partial x_1} \cdots & \frac{\partial y_i}{\partial x_j} \end{bmatrix}_{c \times d} \end{array}$$

Vector Derivative Example

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x \\ x^2 \end{bmatrix} \qquad \frac{\partial \vec{y}}{\partial x} = \begin{bmatrix} 1 \\ 2x \end{bmatrix}$$

$$y = \vec{w}^T \vec{x} = \sum_{i=1}^{d} w_i x_i \qquad \frac{\partial y}{\partial \vec{x}} = \left[\frac{\partial y}{\partial x_1} \cdots \frac{\partial y}{\partial x_d} \right]$$

$$\frac{\partial \left(\sum w_i x_i\right)}{\partial x_1}$$

$$\cancel{\frac{\partial (\vec{w}^T \vec{x})}{\partial \vec{x}}} \qquad [w_1 \cdots w_d] = \vec{w}^T$$

Vector Derivative Example

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

$y_i = \sum_j a_{ij} x_j$

$\vec{x} \in \mathbb{R}^d$

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

$A \in \mathbb{R}^{c \times d}$

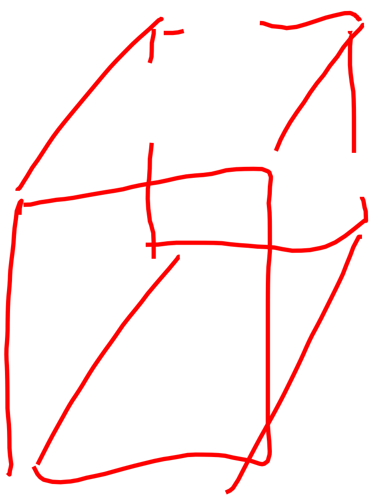
$\vec{y} = A\vec{x}$

$$\left[\frac{\partial \vec{y}}{\partial \vec{x}}\right] = \left[\quad A \quad\right] \qquad i \qquad \overset{j}{\left[\frac{\partial y_i}{\partial x_j} = a_{ij}\right]}$$

Extension to Tensors

$X \in \mathbb{R}^{d_1 \times \dots \times d_m}$

$Y \in \mathbb{R}^{c_1 \times \dots \times c_n}$



y-vec = Y(:)

x-vec = X(:)

$$\frac{\partial Y[i_1 \dots i_n]}{\partial X[j_1 \dots j_m]}$$

$$\frac{\partial \, \text{y-vec}}{\partial \, \text{x-vec}} = \begin{bmatrix} \; \end{bmatrix}$$

Chain Rule: Composite Functions

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

$$L(w) = g_l(g_{l-1}(g_{l-2} \cdots (g_1(w))))$$

$$= (g_l \circ \cdots g_1)(\cancel{x}w)$$

$$\frac{\partial L}{\partial w}$$