CSCI 497P/597P: Computer Vision

(Stochastic) Gradient Descent

Neural networks

Readings

with a great deal more detail…

- http://cs231n.github.io/optimization-1/
- http://cs231n.github.io/optimization-2/
- http://cs231n.github.io/neural-networks-1/
- http://cs231n.github.io/neural-networks-2/
- http://cs231n.github.io/neural-networks-3/

Goals

- Understand how to train a classifier by minimizing a loss function using gradient descent.
- Understand the intuition behind using Stochastic (Minibatch) Gradient Descent.
- Understand neural networks as a stack of linear classifiers with nonlinearities (activation functions) in between.
  - Understand why we need activation functions.
  - Understand the vanishing gradients problem.

Taking stock

- We have:
  - – ϕ = unravel(rgb2gray(img)), a feature extractor

  - – h(x) = W^T x, a multiclass linear classifier
  - – L = $\sum_{i=0}^{N} L_i$, a loss function

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

- We don't have:
  - – a way to find a W that results in a small L.

Minimizing the Loss

- Use **optimization** to find the W that *minimizes* the loss function.
    - Linear regression: solvable in closed form
    - Most of the time: no closed form.

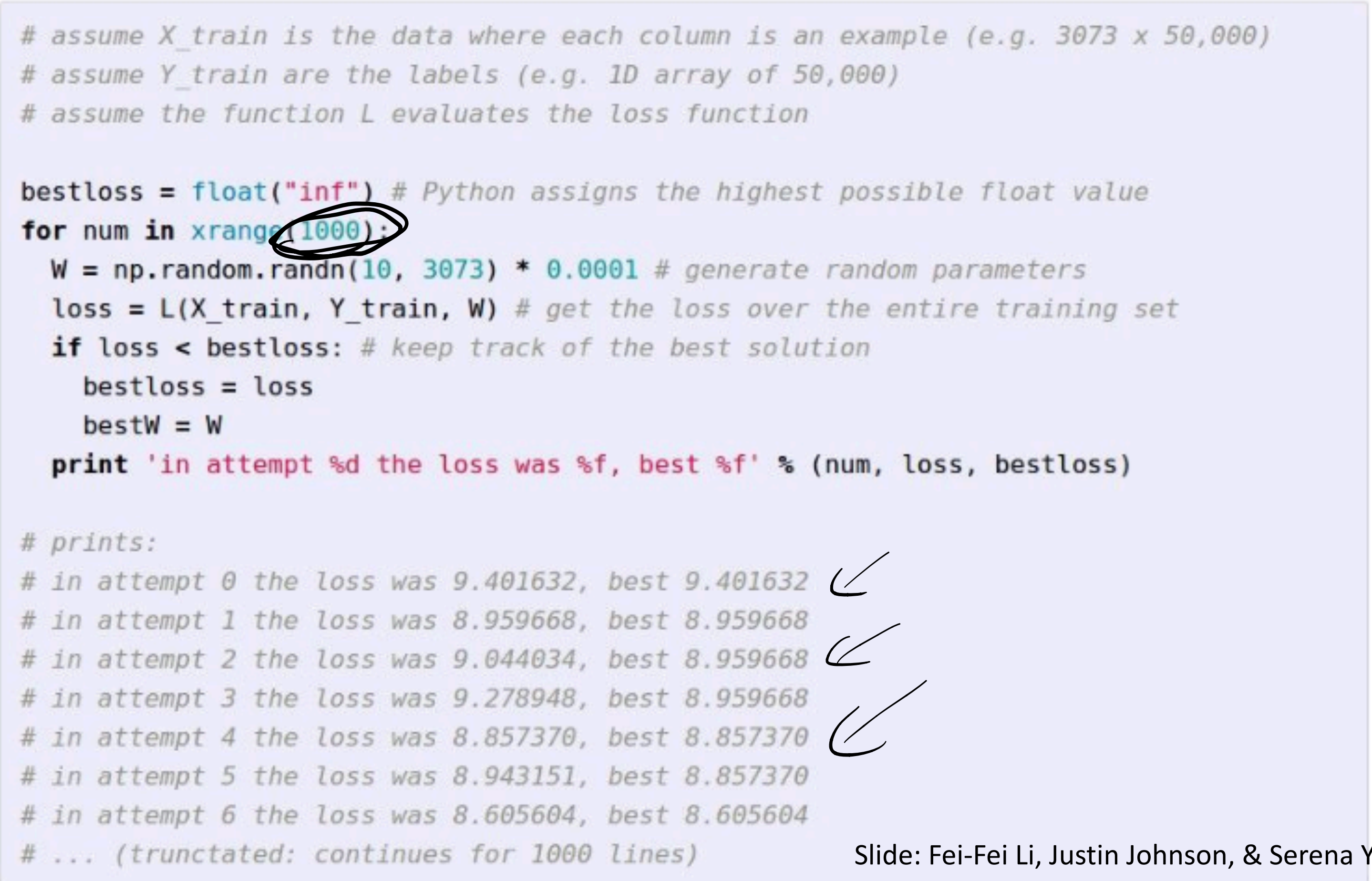
Optimization

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Walking man image is CC0 1.0 public domain

How do we find a W that minimizes L?

- Bad idea: Random search.

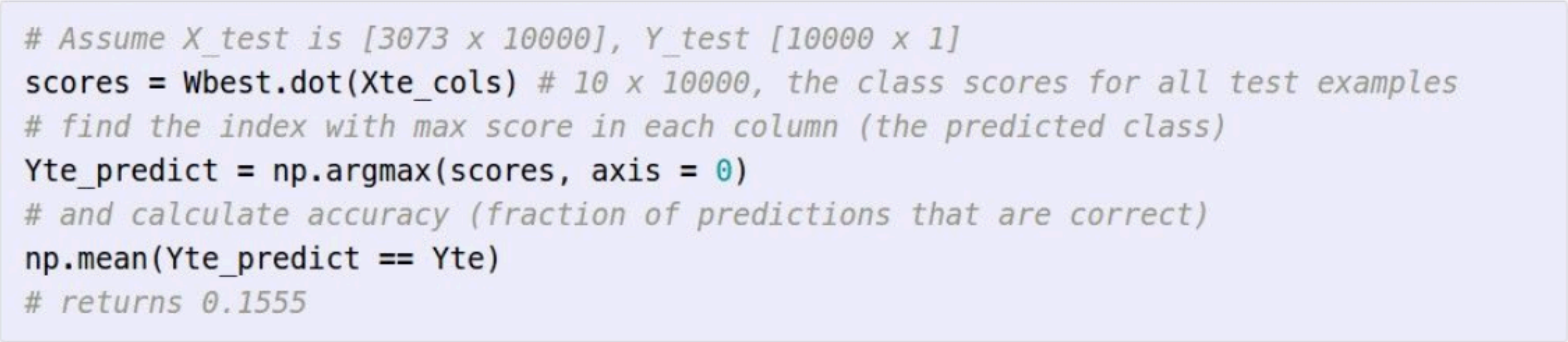


```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

How’d that go for you?

Lets see how well this works on the test set...



```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! ~~not~~ bad!
(SOTA is ~95%)

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Finding a W that minimizes L

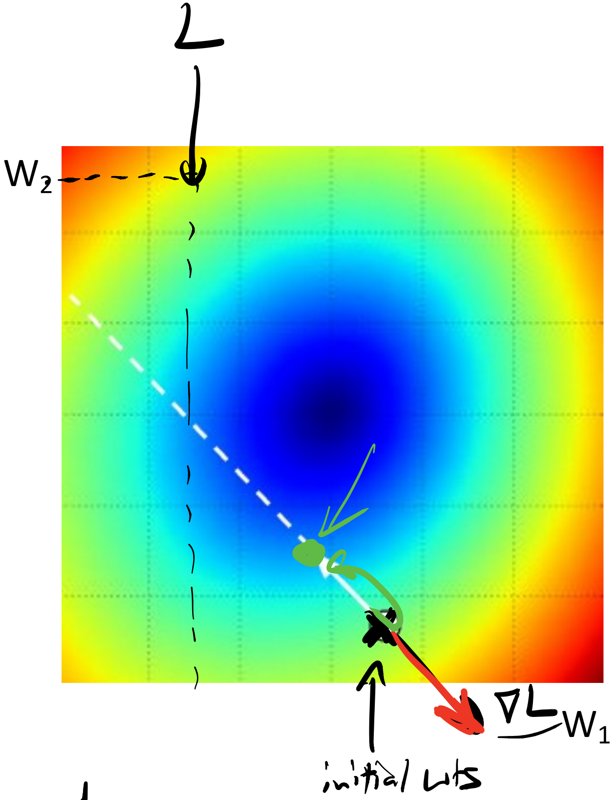
- Simple idea: walk downhill.

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Gradient Descent: Generally

- Gradient of the loss function with respect to the *weights* tells us how to change the weights to improve the loss.

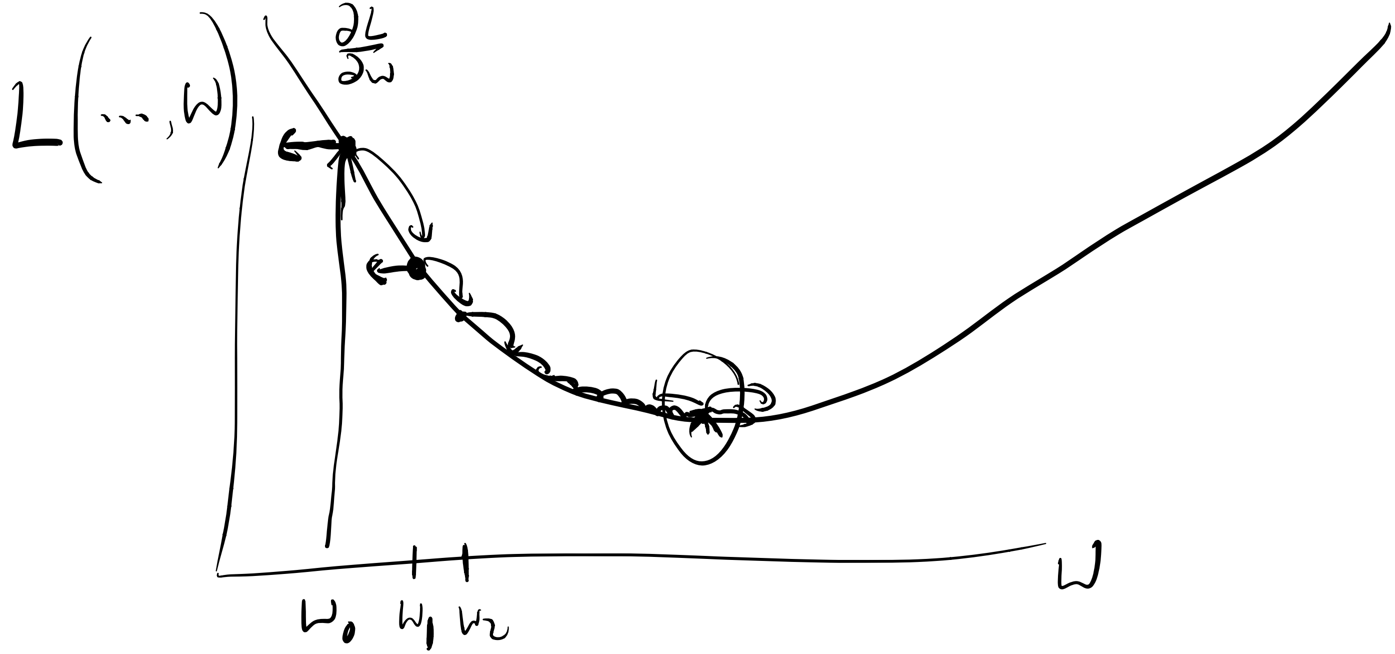
$$\downarrow L(\underset{\substack{\uparrow \\ \text{all training data}}}{X}, \underset{\substack{\uparrow \\ \text{all training labels}}}{Y}, \overbrace{W_1, W_2}^{\text{classifier wts}})$$



$$\begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix} = \nabla L$$

$$W \leftarrow W - \nabla L$$

Gradient Descent: Intuition



$$w_1 \leftarrow w_0 - \alpha \frac{\partial L}{\partial w}(w_0)$$

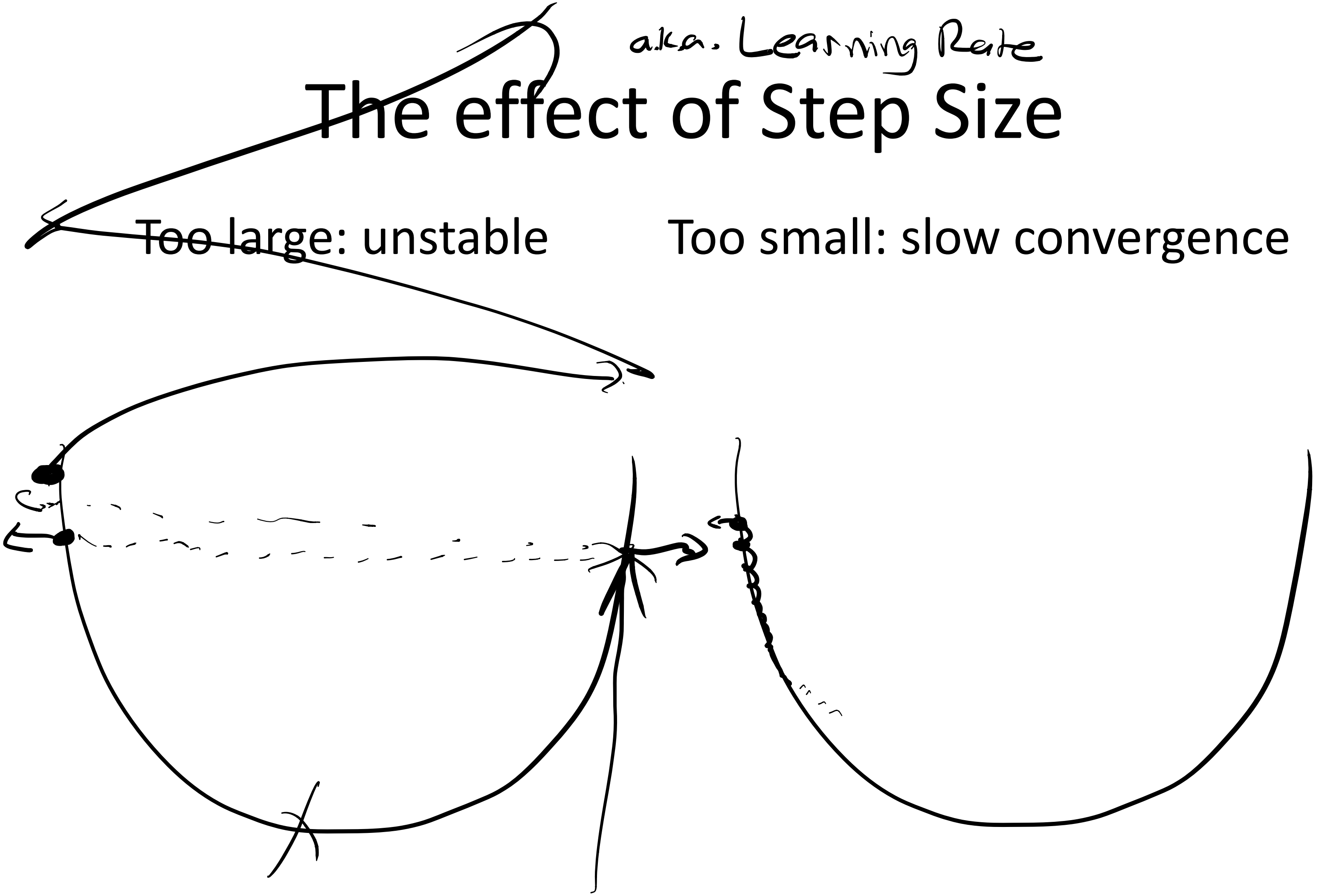
$$w_2 \leftarrow w_1 - \alpha \underset{\uparrow}{\frac{\partial L}{\partial w}}(w_1)$$

a.k.a. Learning Rate

The effect of Step Size

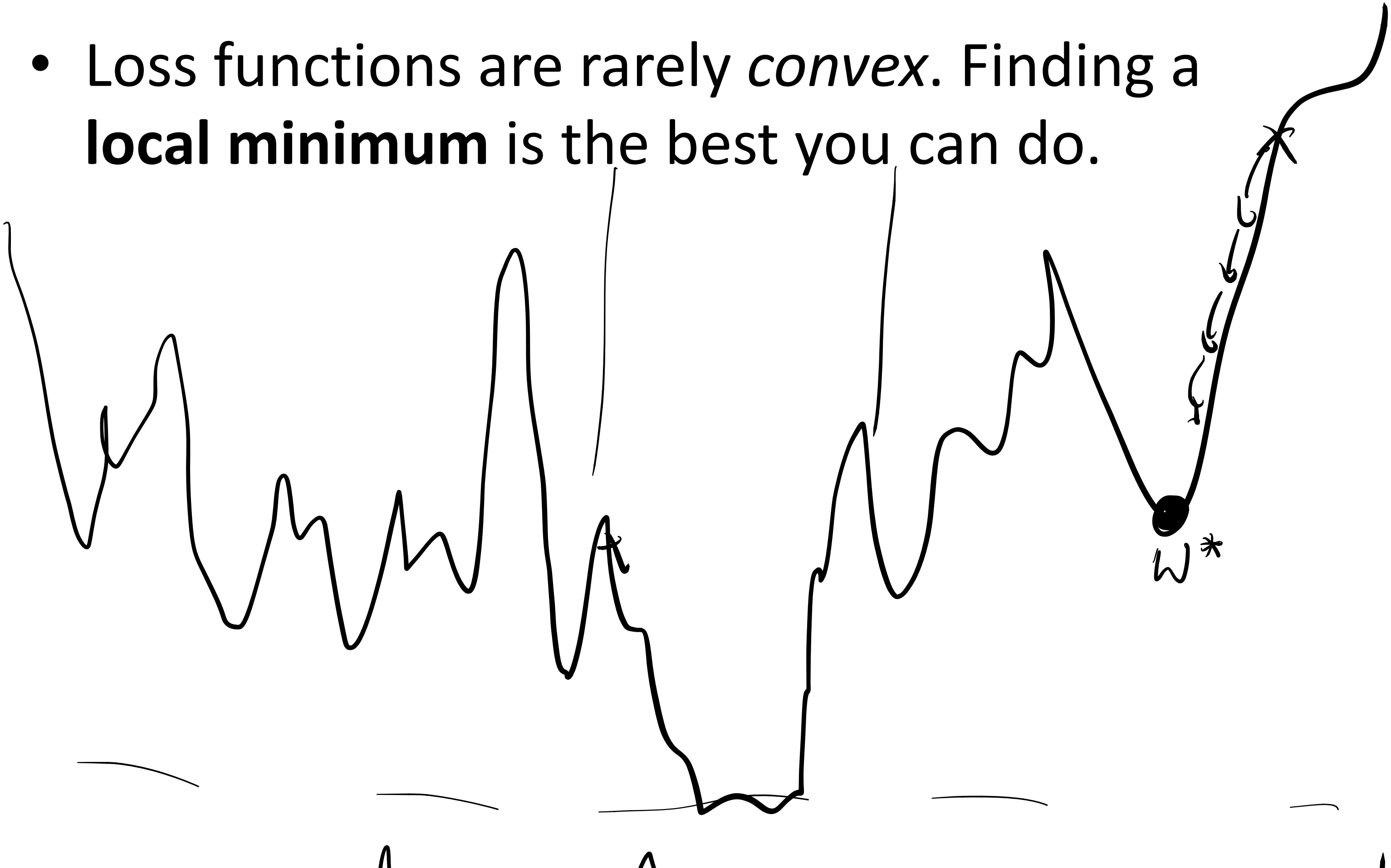
Too large: unstable

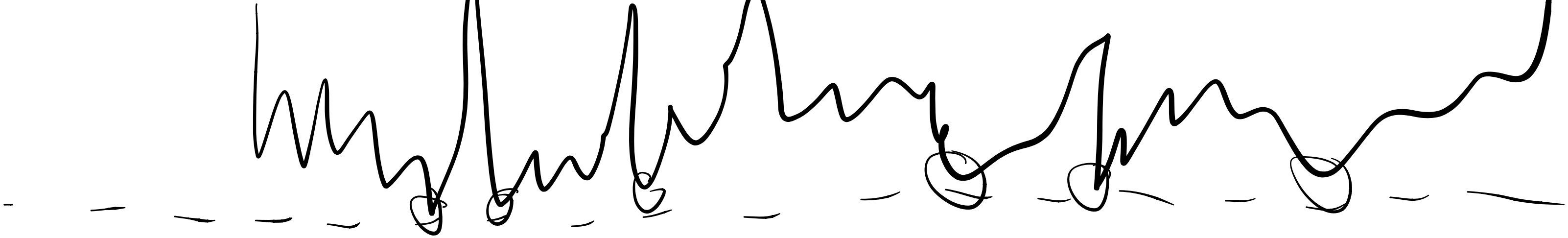
Too small: slow convergence



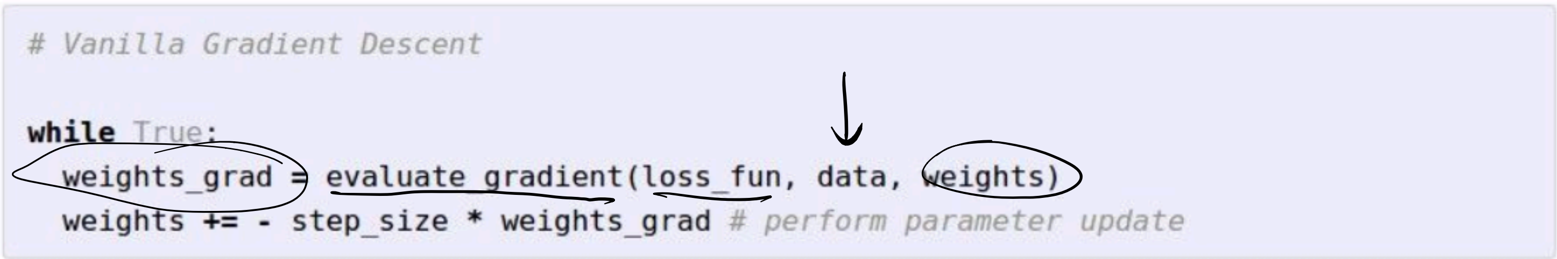
Reality isn't quite so pretty

- Loss functions are rarely *convex*. Finding a **local minimum** is the best you can do.





Gradient Descent

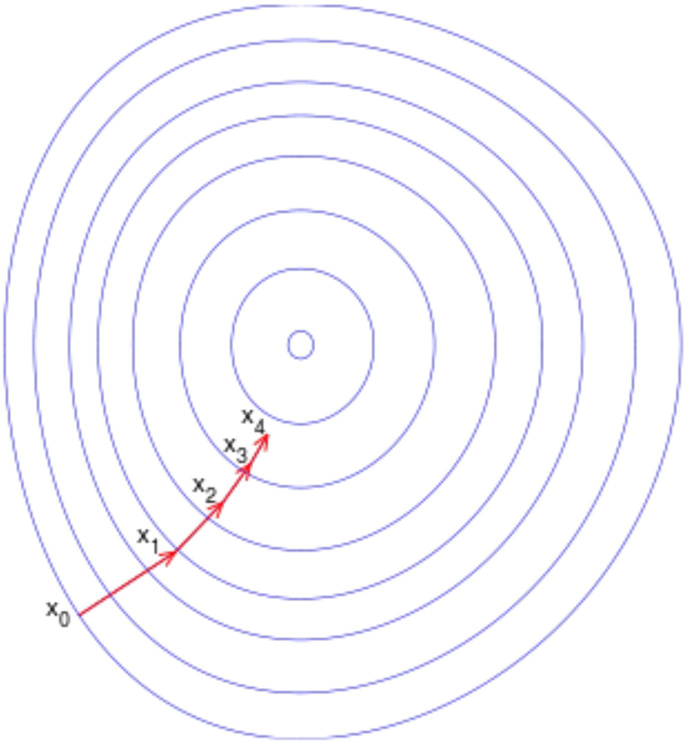


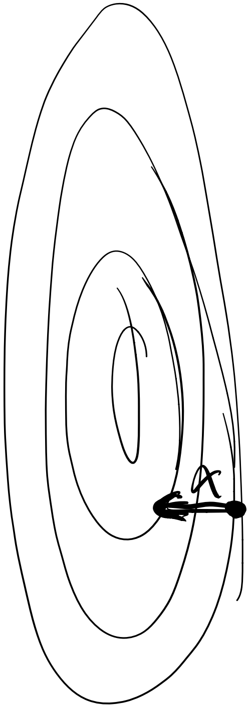
```
# Vanilla Gradient Descent

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

$- \alpha * \nabla L$

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Gradient Descent: Intuition

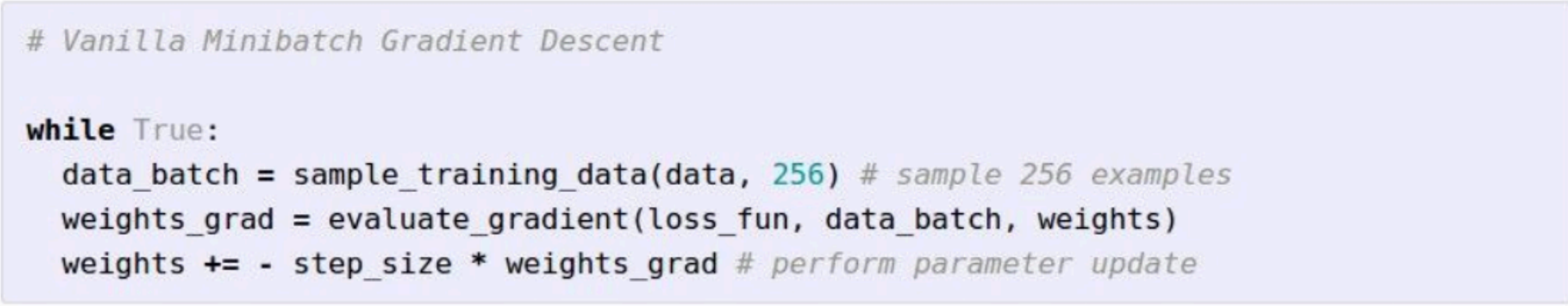




Gradient Descent: Demo

- http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/
  - select “Softmax” radio button at the bottom

Stochastic Gradient Descent



```
# 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
```

- L(X, Y; W) depends on
  - All data points $x_1..x_n$
  - Ground truth labels $y_1..y_n$
  - Weights W
- Very expensive to evaluate if you have a lot of data.

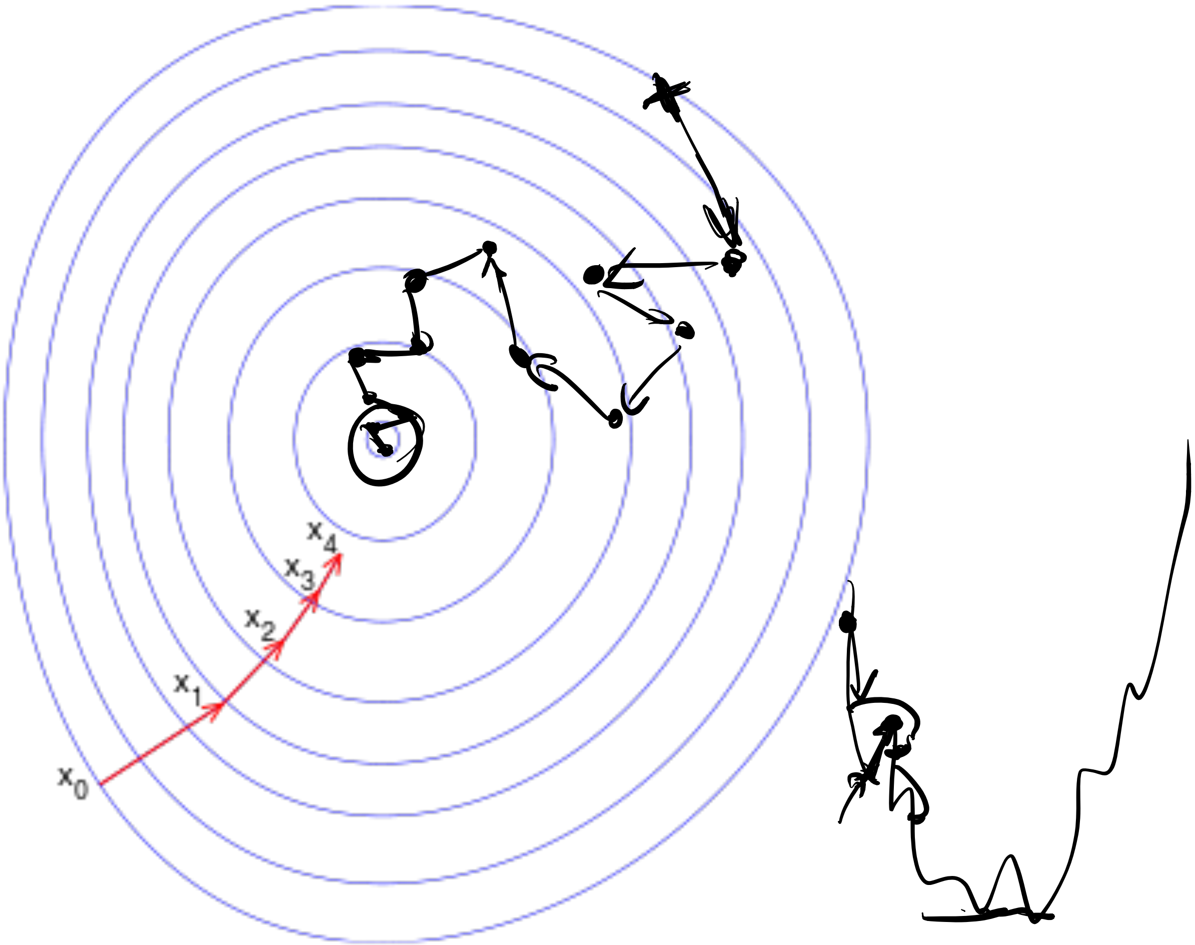
$$L(X_{i..j}\ Y_{i..j},\ W)$$

$$L = \sum_i \ell(x_i, y_i, w)$$

Stochastic Gradient Descent

- Idea: consider only a few data points at a time.
- Loss is now computed using only a small batch (minibatch) of data points.
- Update weights the same way using the gradient of L wrt the weights.

Stochastic Gradient Descent: Intuition



Taking stock

- We have:
  - ϕ = unravel(rgb2gray(img)), a feature extractor
  - h(x) = W^T x, a multiclass linear classifier
  - L = $\sum_{i=0}^{N} L_i$, a loss function

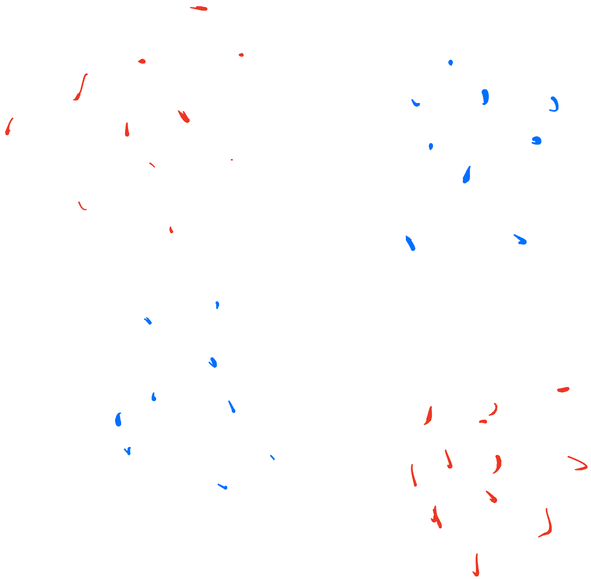
$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

↓

  - A way too adjust W until we can't make L any smaller.

So about that linearly separable assumption...

- Ideas:
  - – ϕ = unravel(rgb2gray(img)), a feature extractor
    - use a fancier ϕ?
  - – Learn ϕ too.



Neural Networks

Neural Network

Linear classifiers

This image is CC0 1.0 public domain

Neural networks: without the brain stuff

(**Before**) Linear score function: $\underset{\underset{\text{Scores}}{\uparrow}}{f} = \underset{\underset{\text{classifier}}{\nearrow}}{\underset{\rightarrow}{W}}\,\underset{\underset{\text{data}}{\uparrow}}{x}$

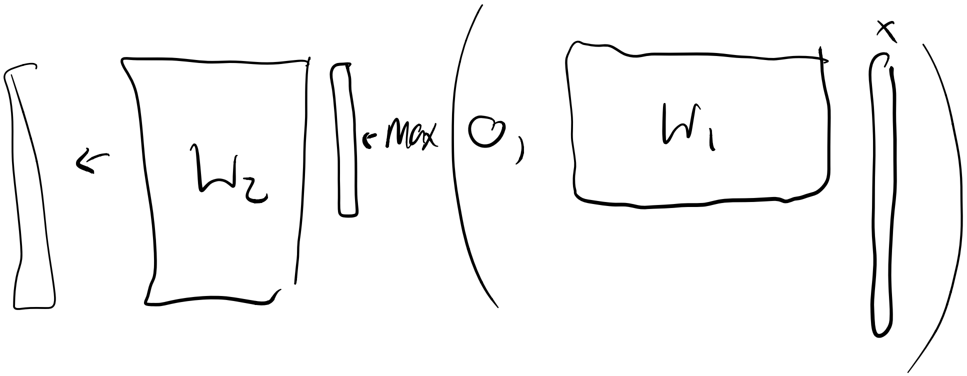
Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Neural networks: without the brain stuff

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

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

?

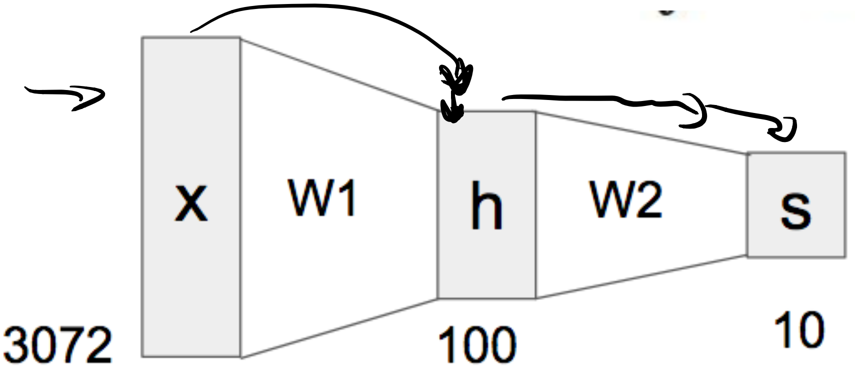


Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

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: Fei-Fei Li, Justin Johnson, & Serena Yeung

Neural networks: without the brain stuff

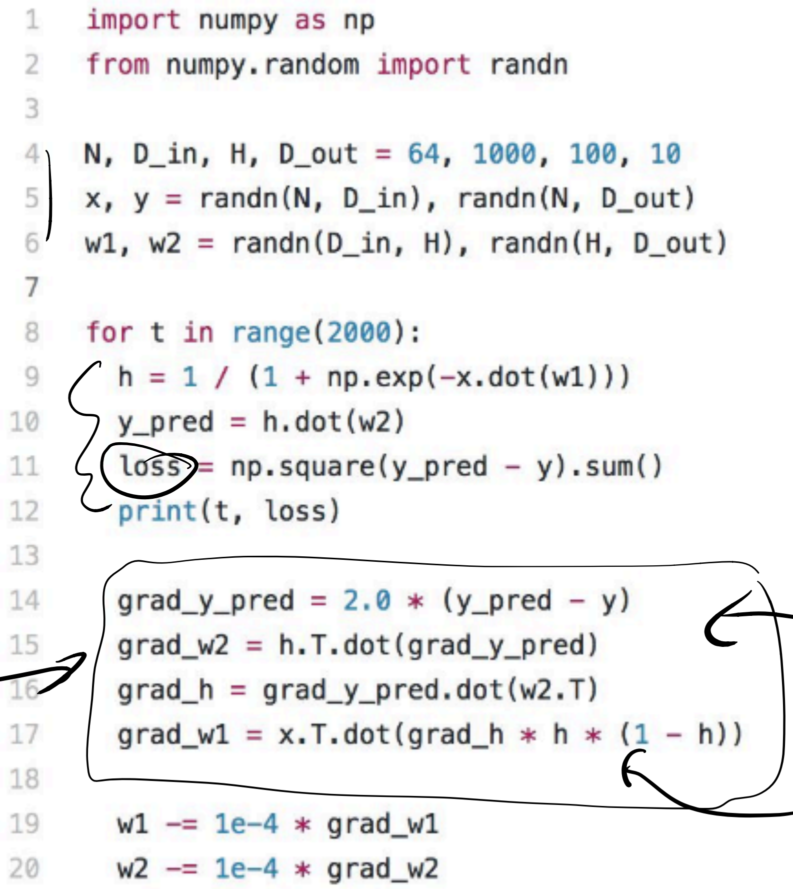
(**Before**) Linear score function: $f = Wx$

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

$$f = \underbrace{W_3}_{} \max(0, \underbrace{W_2}_{} \max(0, \underbrace{W_1 x}_{}))$$

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Training a 2 layer neural network in 20 lines of python



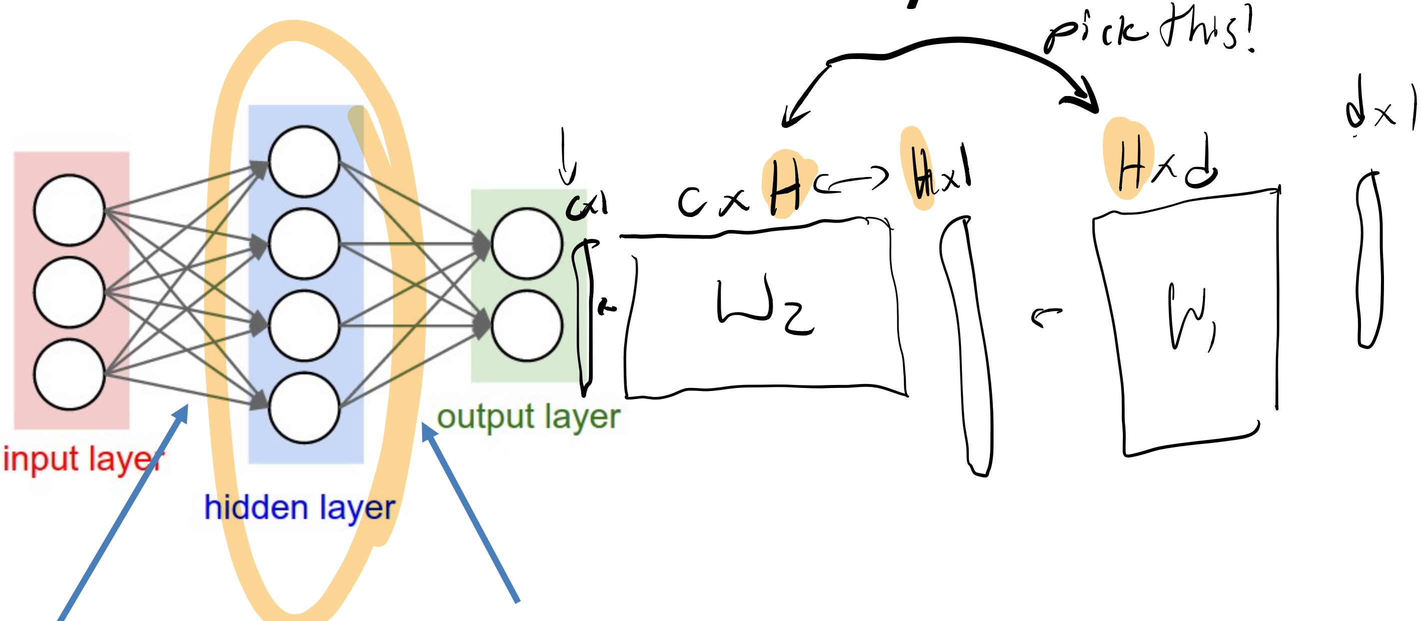
```
import numpy as np
from numpy.random import randn

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
```

$\frac{\partial L}{\partial W_2}$

$\frac{\partial L}{\partial W_1}$

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

“Hidden Layers”

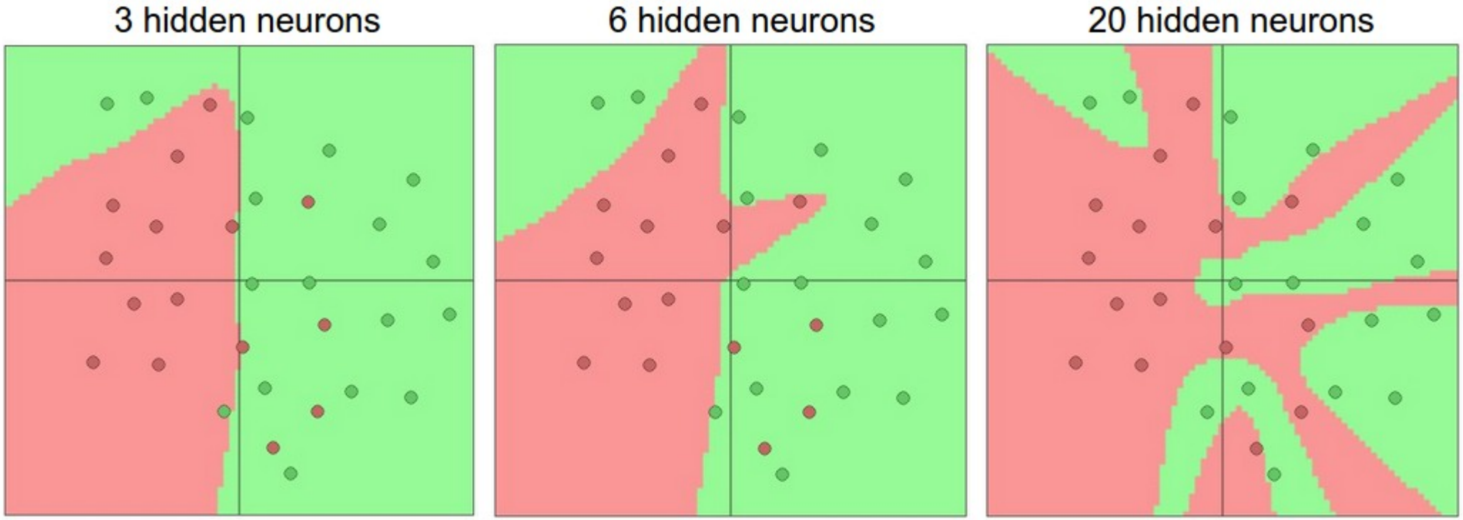


W_1, a 3x4 matrix converts input into hidden layer activations

W_2, a 4x2 matrix transforms hidden layer activations to output scores

Figures: Fei-Fei Li, Justin Johnson, & Serena Yeung

Neural Networks: Nonlinear Classifiers built from Linear Classifiers



Figures: Fei-Fei Li, Justin Johnson, & Serena Yeung

Neural networks: without the brain stuff

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

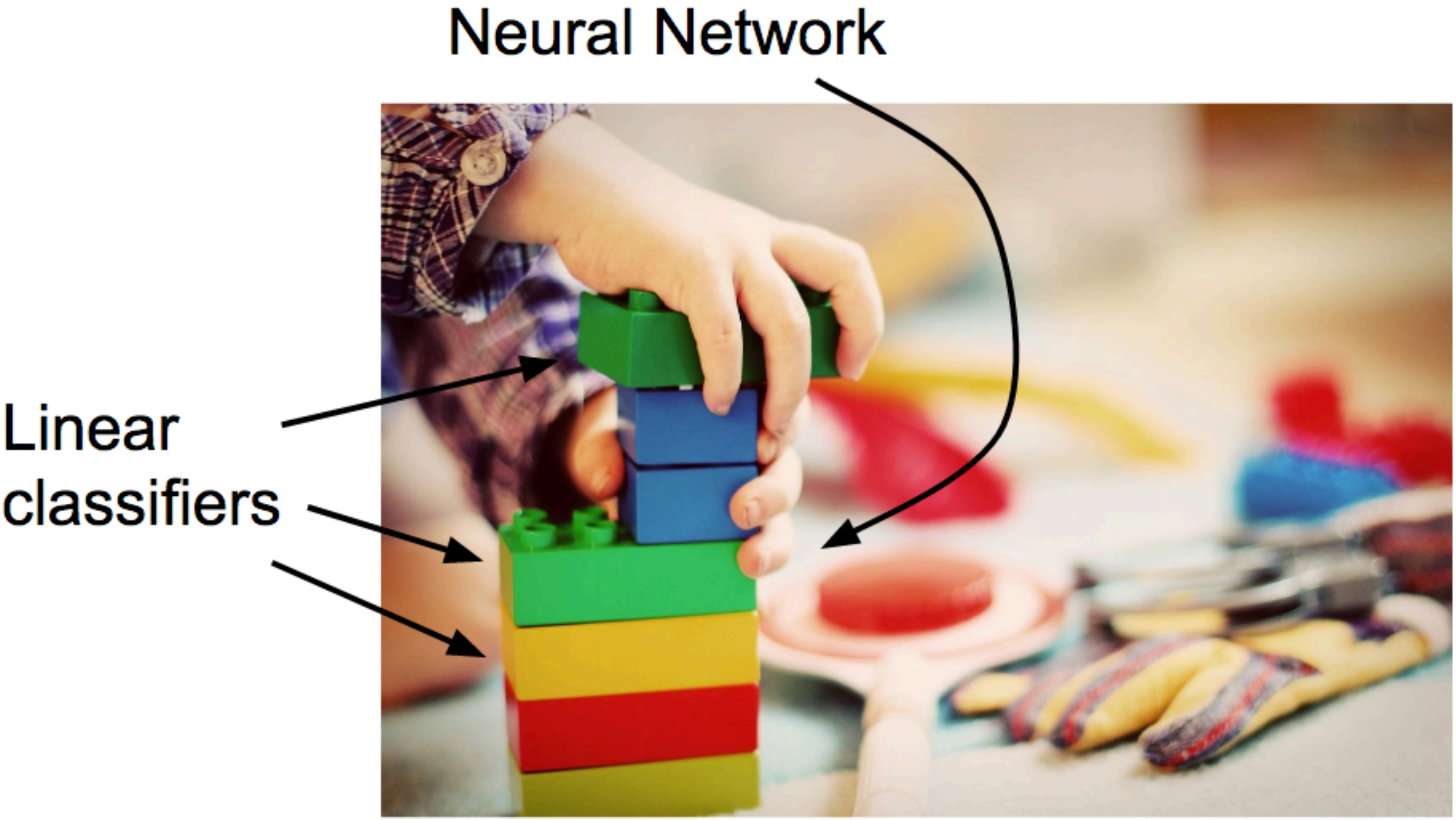
(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$
or 3-layer Neural Network

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

↑ ??? ↑

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

Neural Networks



Neural Network

Linear classifiers

This image is CC0 1.0 public domain

Slide: Fei-Fei Li, Justin Johnson, & Serena Young

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

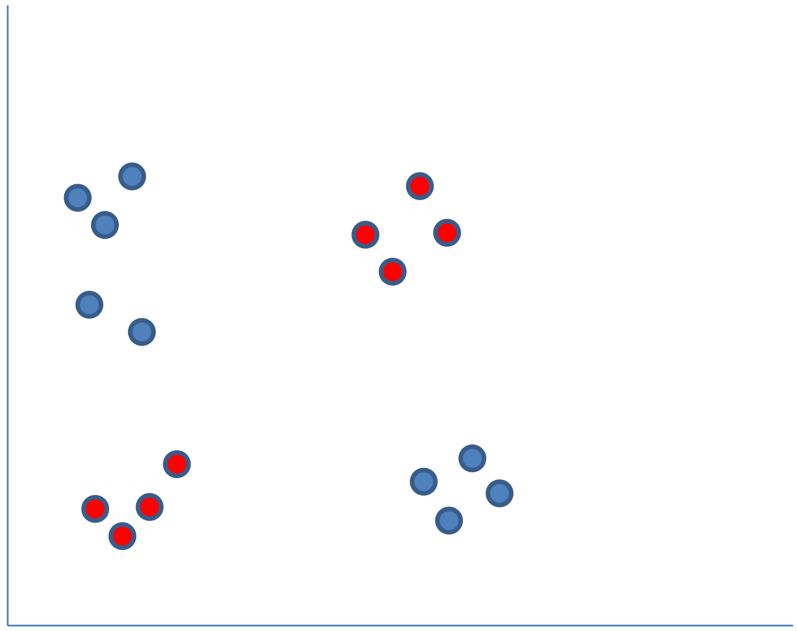
$$f = W_3 \boxed{\max(0,} W_2 \boxed{\max(0,} W_1 x))$$

???

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung

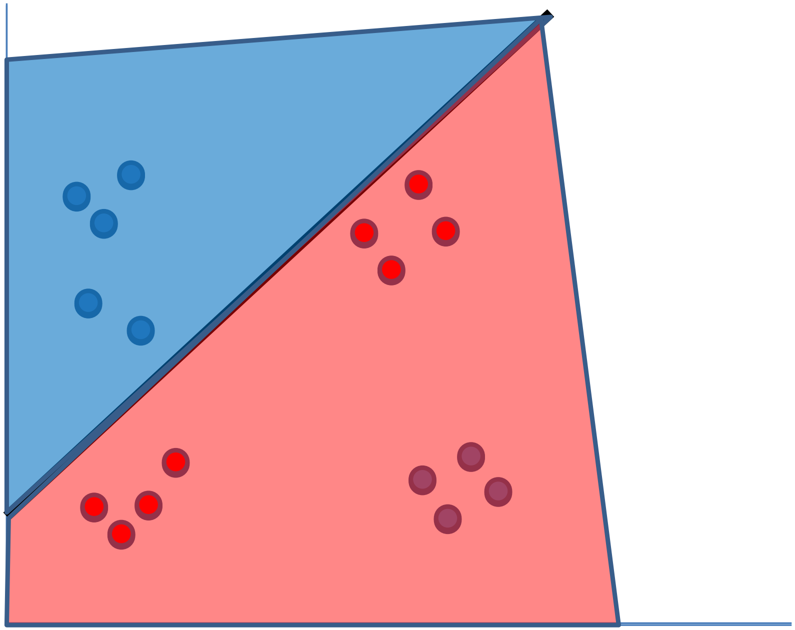
Activation Functions

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



Activation Functions

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

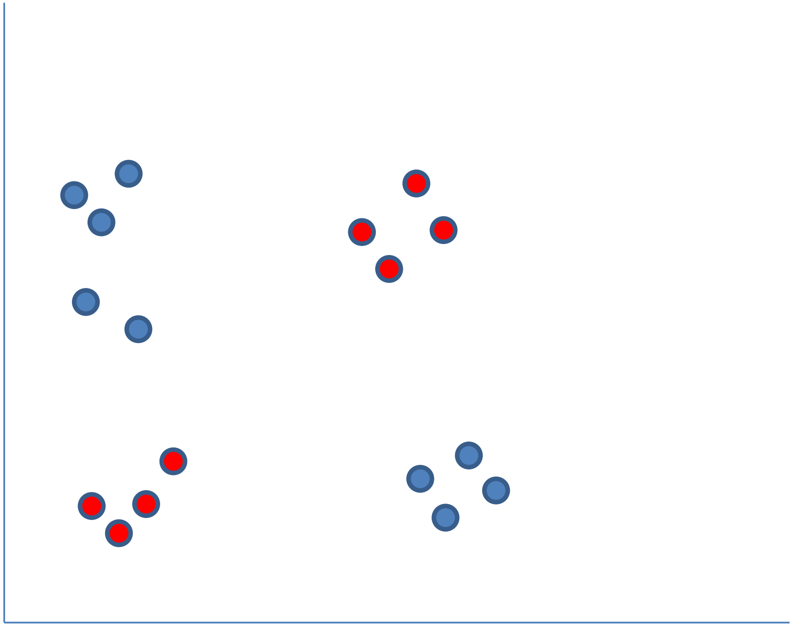


A linear classifier can only do so well…

Activation Functions

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

$$f(x, W_1, W_2) = W_1(W_2x)$$



Let’s try stacking two linear classifiers together

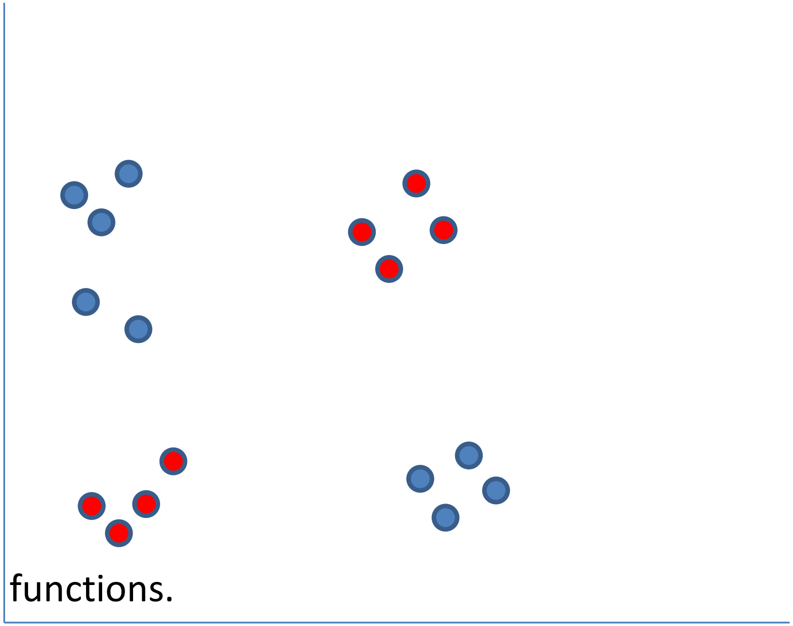
Activation Functions

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

$$f(x, W_1, W_2) = W_1(W_2 x)$$

$$W \leftarrow W_1 W_2$$

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



Uh oh – linear functions compose to linear functions.

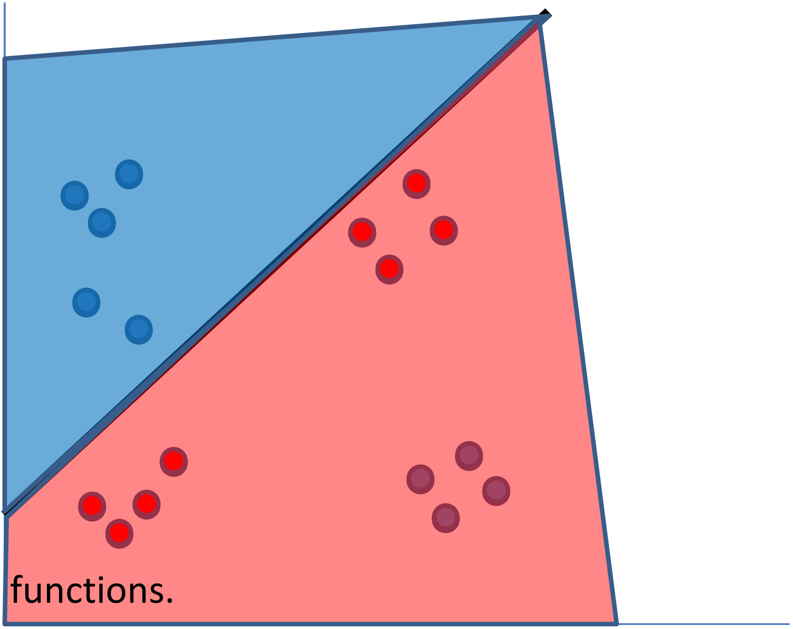
Activation Functions

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

$$f(x, W_1, W_2) = W_1(W_2 x)$$

$$W \leftarrow W_1 W_2$$

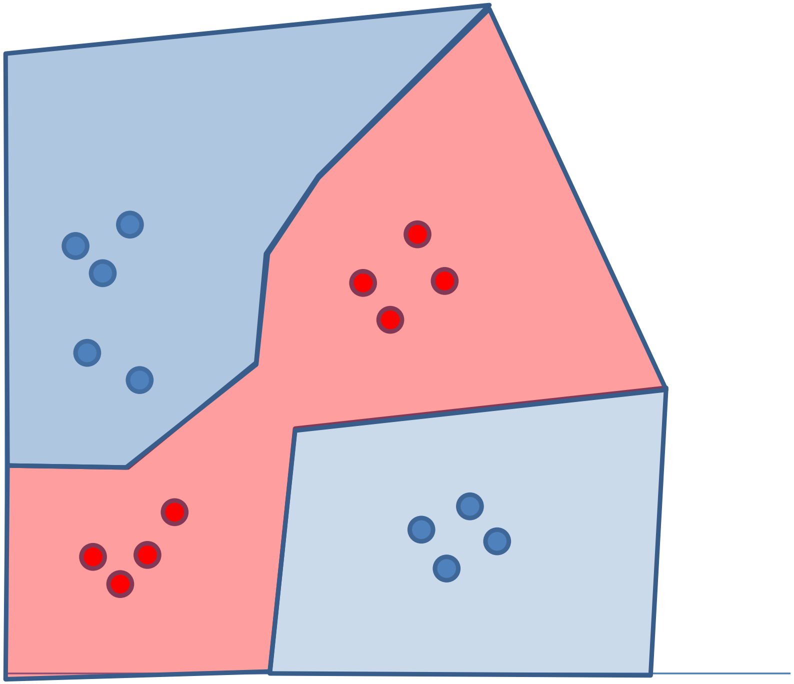
$$f(x, W) = Wx$$



Uh oh – linear functions compose to linear functions.

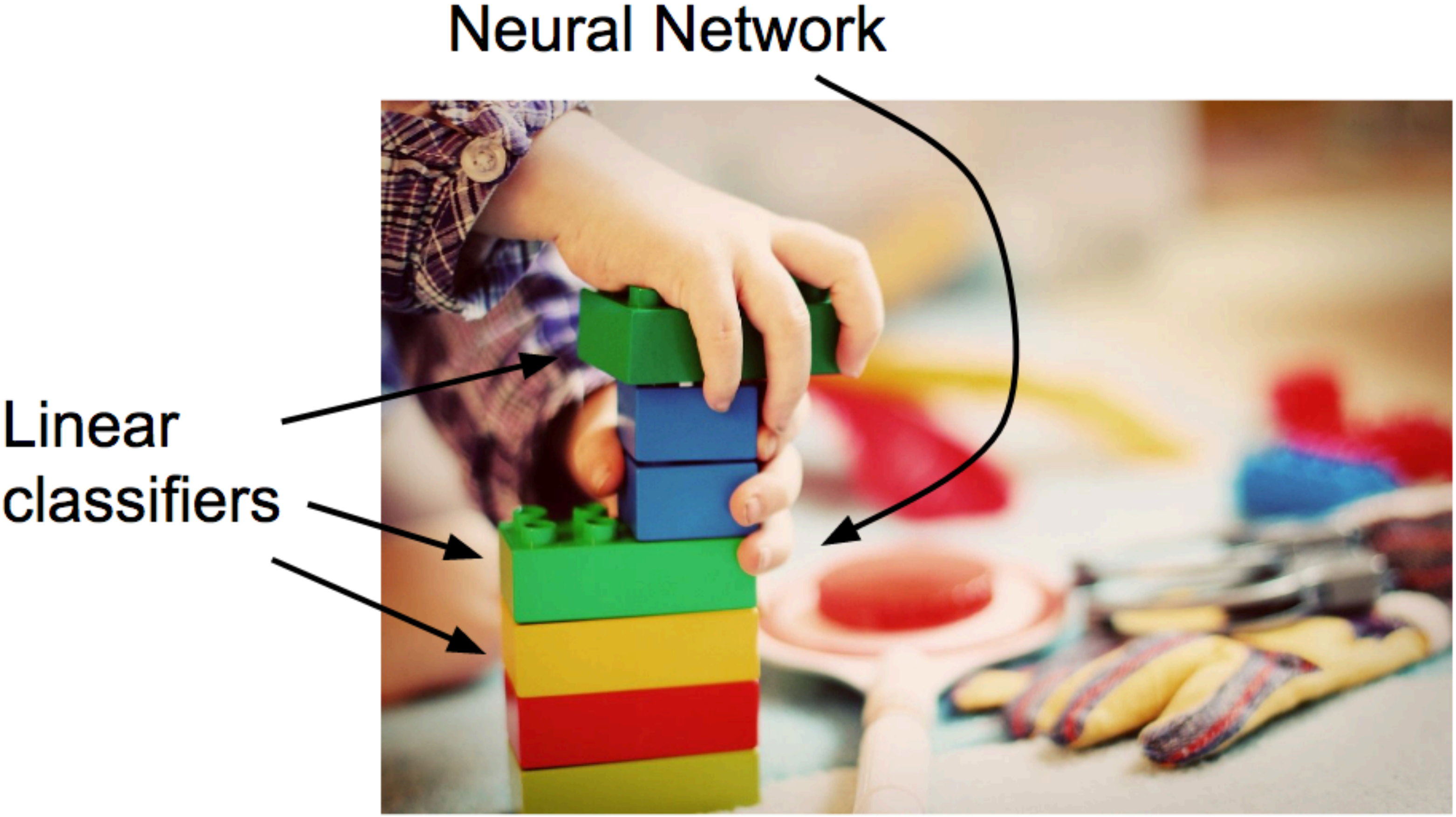
Activation Functions

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



Nonlinearities prevent the composed linear functions from collapsing into a single one.

Neural Networks



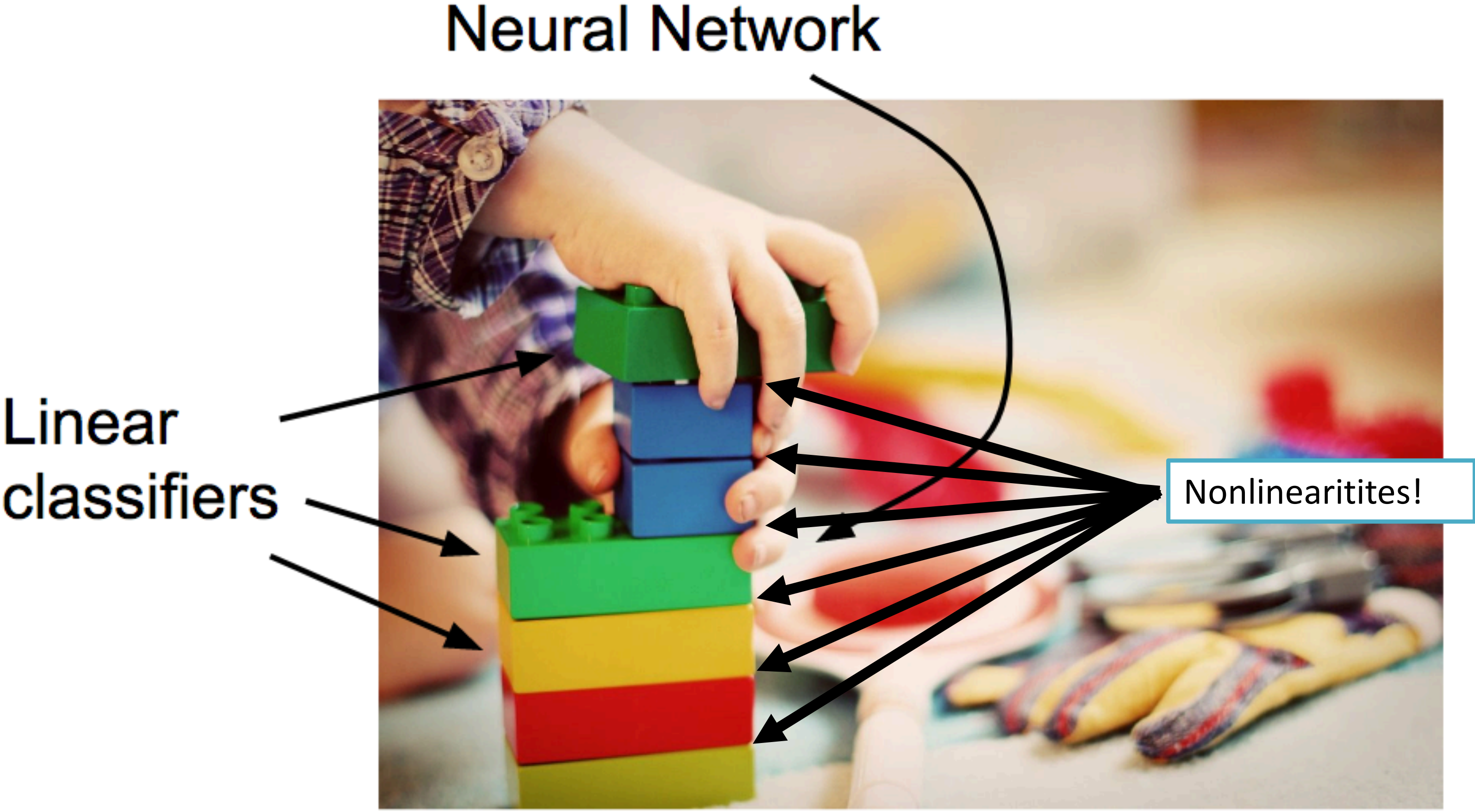
Neural Network

Linear classifiers

This image is CC0 1.0 public domain

Slide: Fei-Fei Li, Justin Johnson, & Serena Young

Neural Networks



Neural Network

Linear classifiers

Nonlinearitites!

This image is CC0 1.0 public domain

Slide: Fei-Fei Li, Justin Johnson, & Serena Young

Neural Networks: Nonlinear Classifiers built from Linear Classifiers

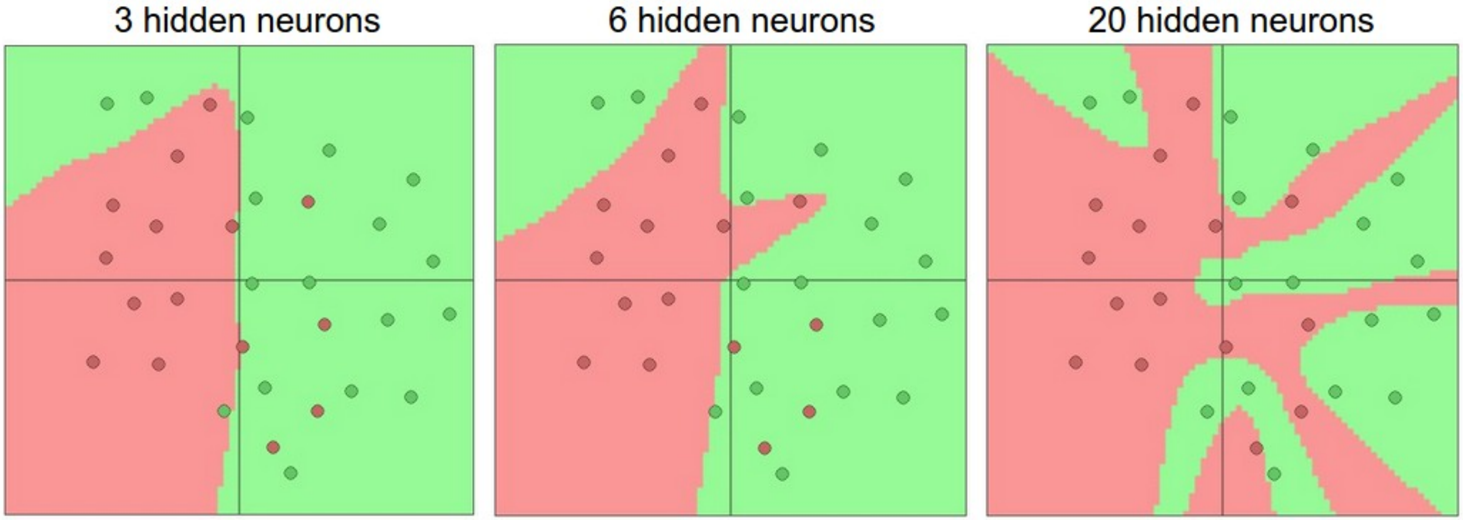
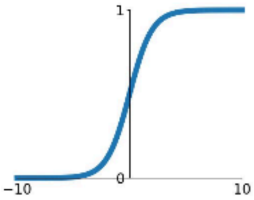


Figure: Fei-Fei Li, Justin Johnson, & Serena Yeung

Activation Functions

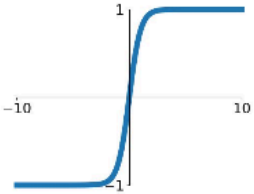
Sigmoid

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



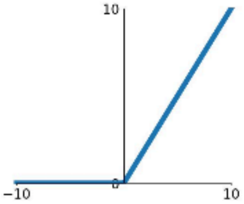
tanh

$\tanh(x)$



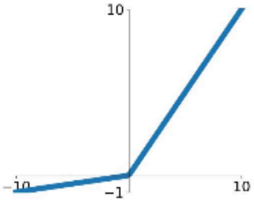
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

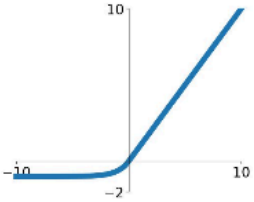


Maxout

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

ELU

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



Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung