Machine Learning

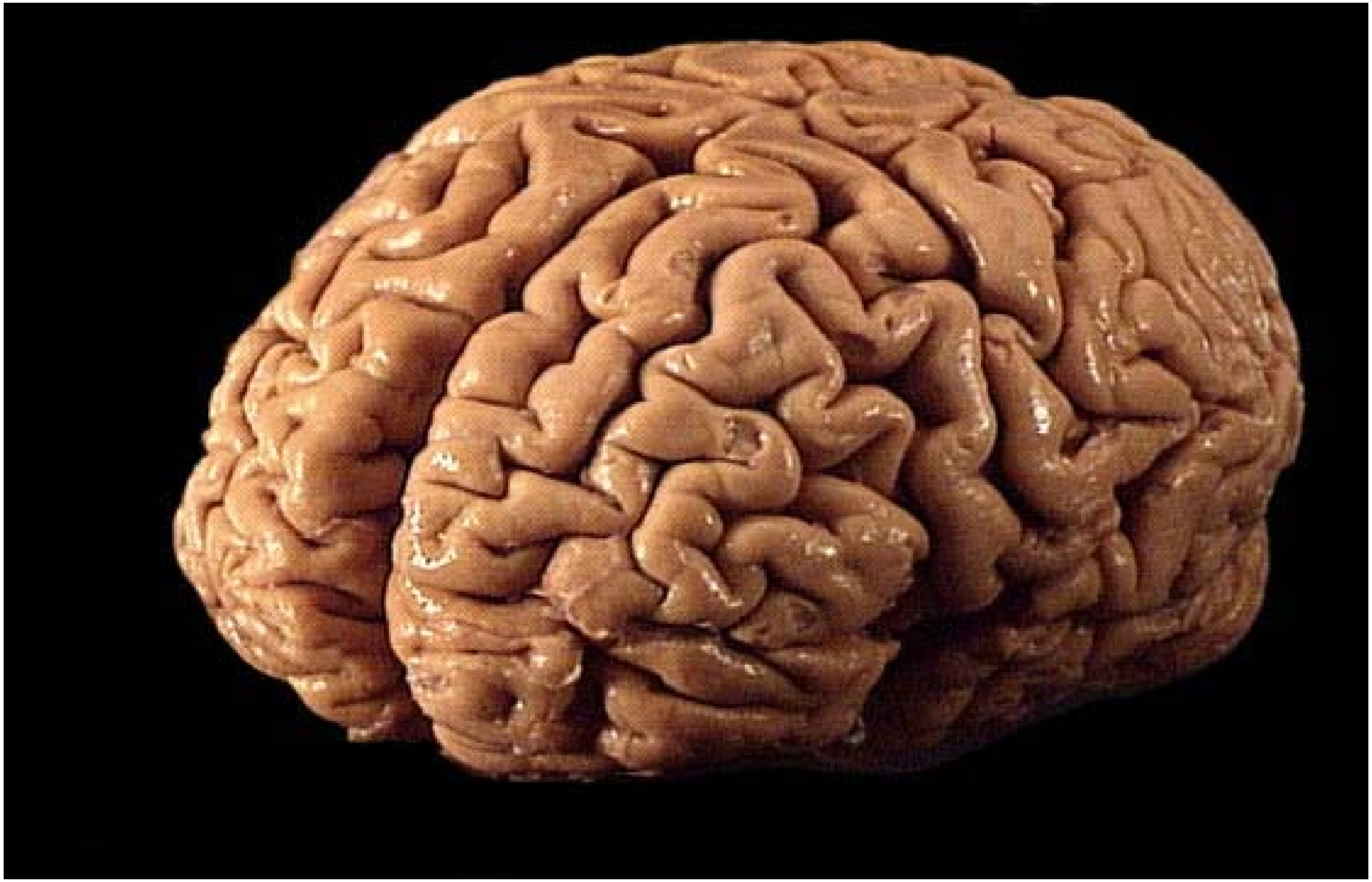
Neural Networks

(slides from Domingos, Pardo, others)

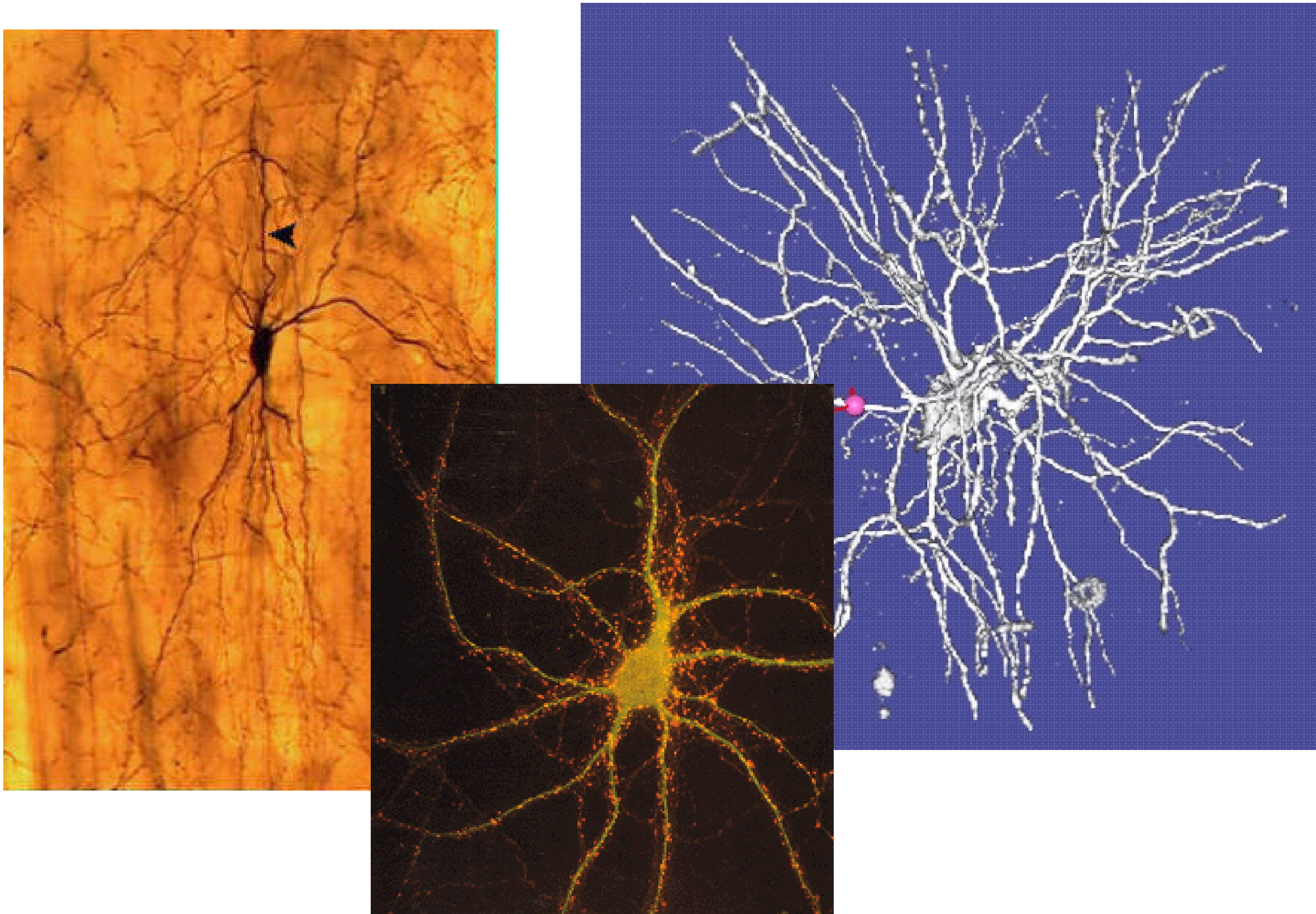
Reading

- For this week,
  - Chapter 4: Neural Networks (Mitchell, 1997)
  - See Canvas
- For subsequent weeks:
  - Scaling Learning Algorithms toward AI
  - Learning Deep Architectures for AI

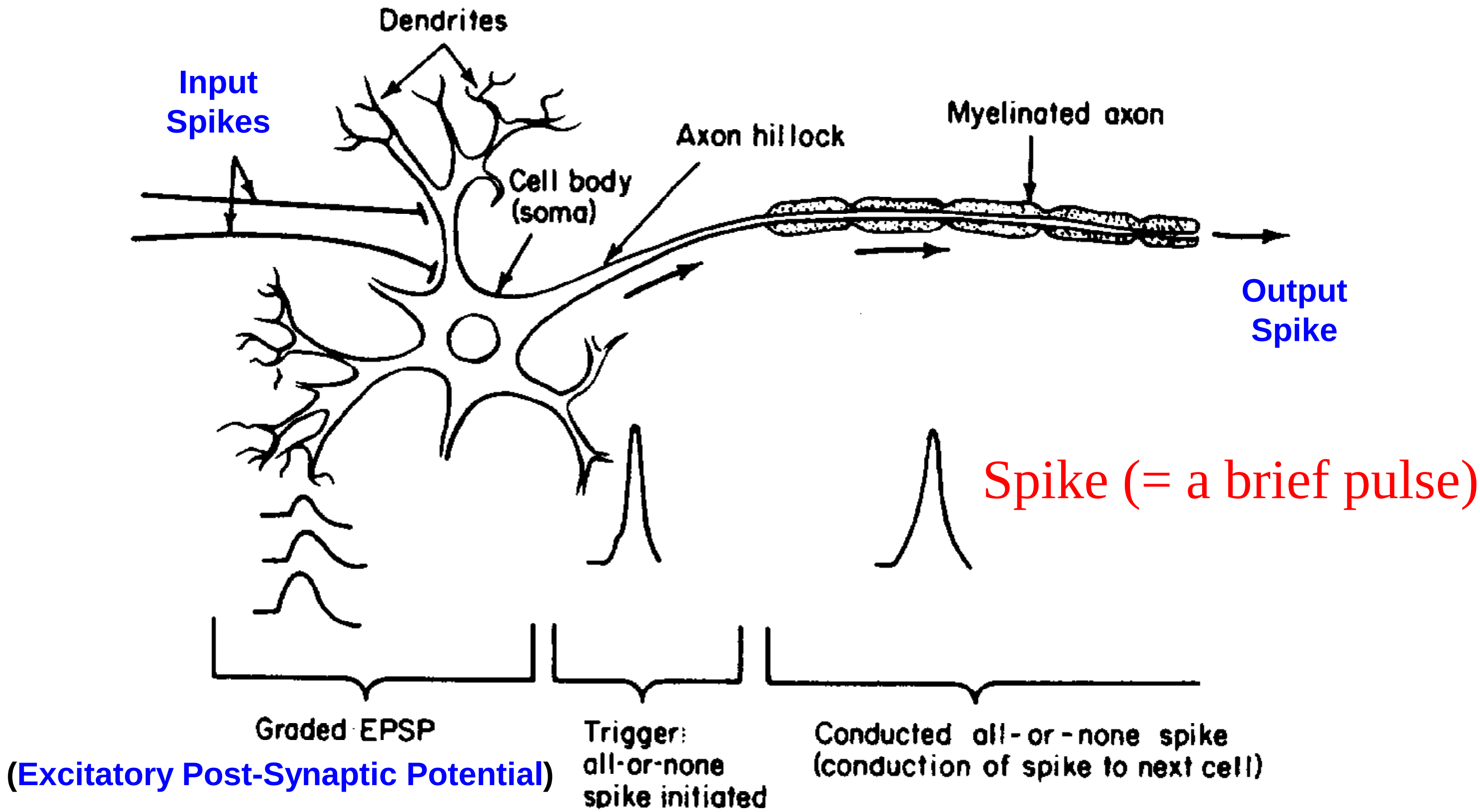
Human Brain



Neurons



Input-Output Transformation

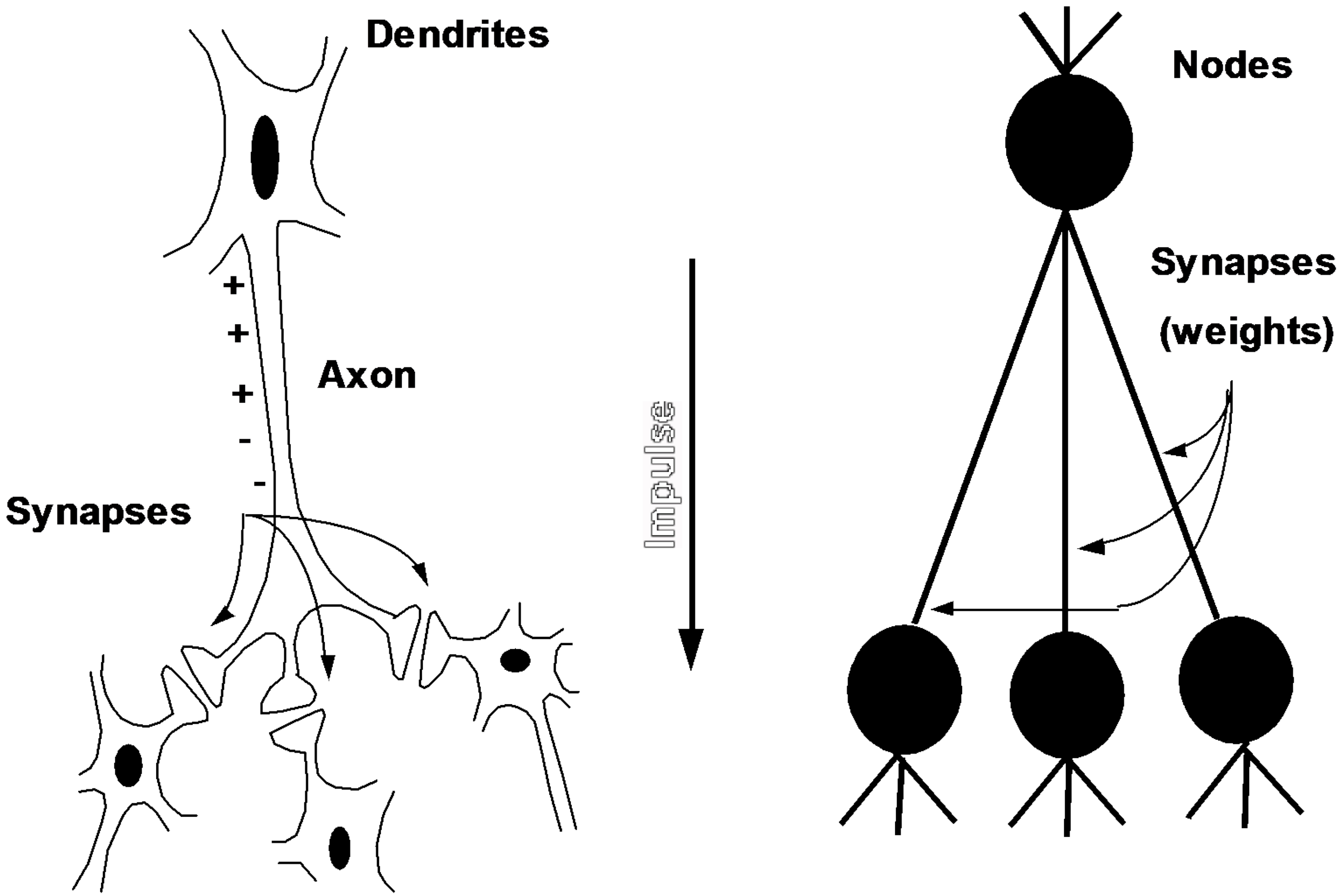


Human Learning

- Number of neurons: ~ 10^{11}
- Connections per neuron: ~ 10^3 to 10^5
- Neuron switching time: ~ 0.001 second
- Scene recognition time: ~ 0.1 second

100 inference steps doesn’t seem much

Machine Learning Abstraction



Artificial Neural Networks

- Typically, machine learning ANNs are very artificial, ignoring:
  - Time
  - Space
  - Biological learning processes
- More realistic neural models exist
  - Hodgkin & Huxley (1952) won a Nobel prize for theirs (in 1963)
- Nonetheless, very artificial ANNs have been useful in many ML applications

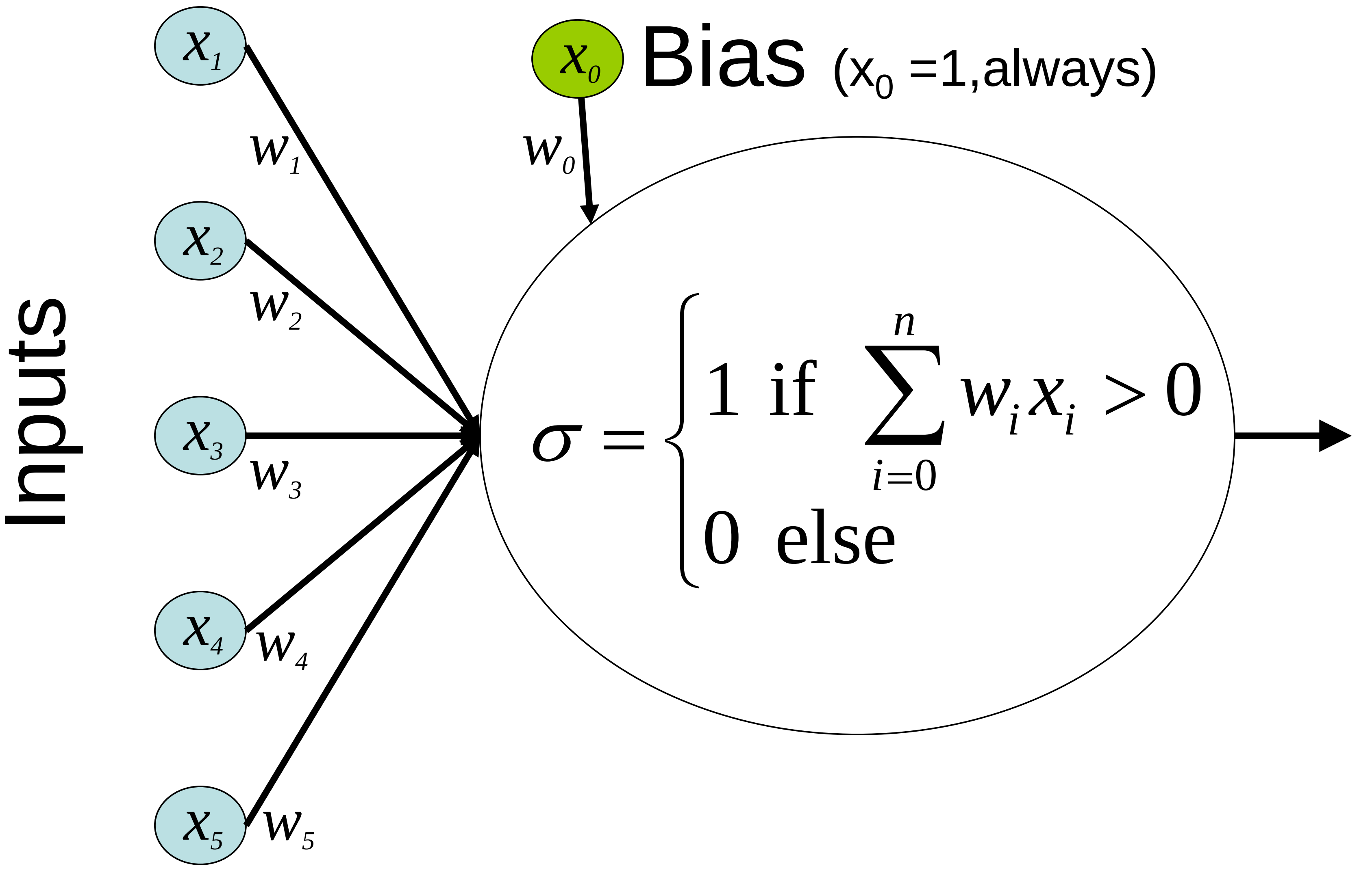
Perceptrons

- The “first wave” in neural networks
- Big in the 1960’s
  - McCulloch & Pitts (1943), Woodrow & Hoff (1960), Rosenblatt (1962)

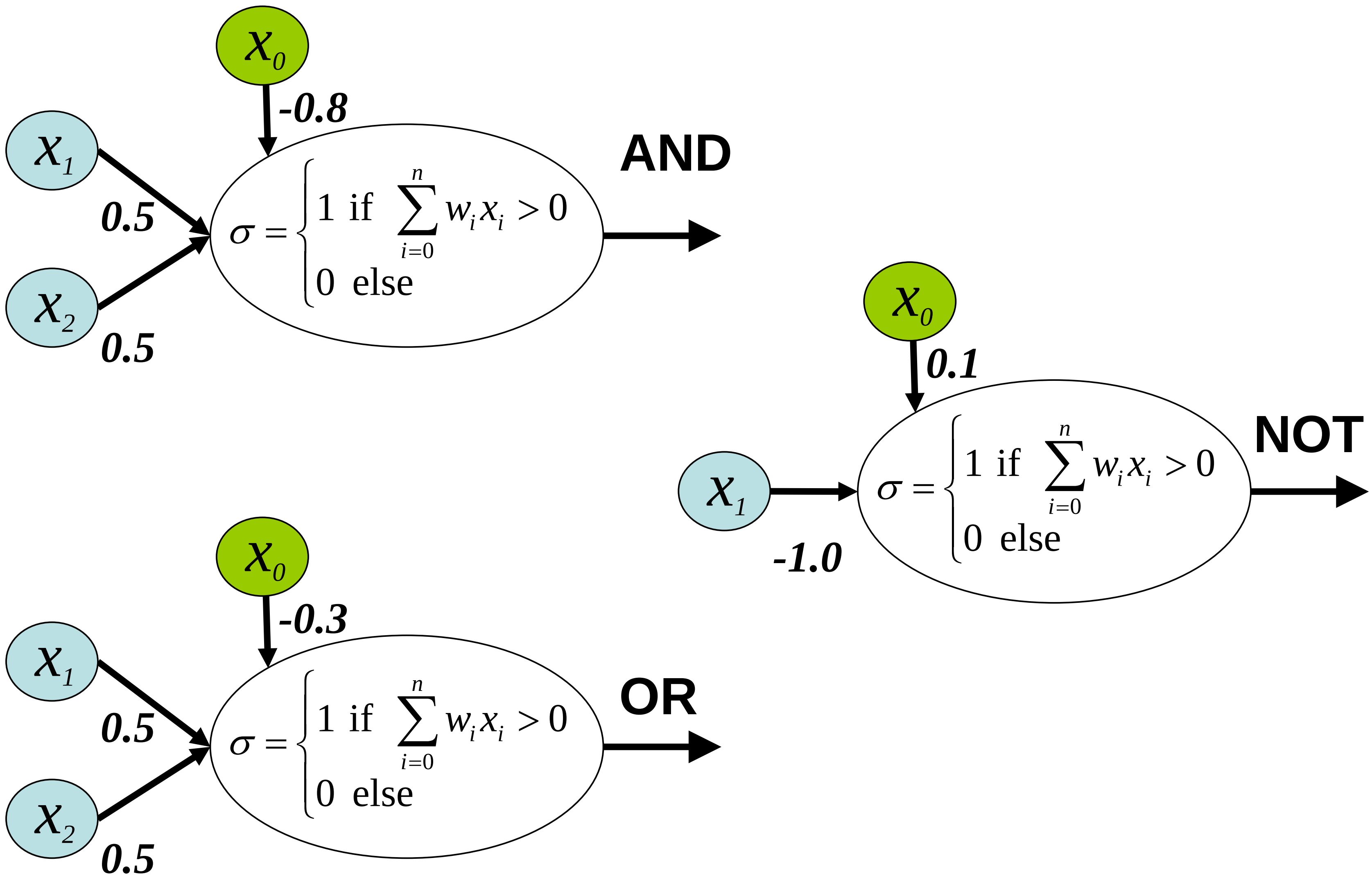
Perceptrons

- Problem def:
  - Let f be a target function from $X = <x_1, x_2, ...>$ where $x_i \in \{0, 1\}$ to $y \in \{0, 1\}$
  - Given training data $\{(X_1, y_1), (X_2, y_2)...\}$
    - Learn $h(X)$, an approximation of $f(X)$

A single perceptron

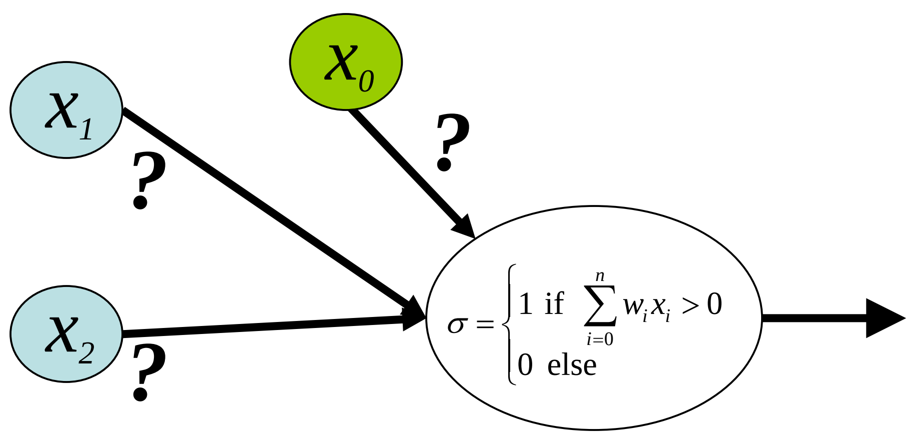


Logical Operators



Learning Weights

- Perceptron Training Rule
- Gradient Descent
- (other approaches: Genetic Algorithms)



Perceptron Training Rule

- Weights modified for each training example
- Update Rule:

$$w_i \leftarrow w_i + \Delta w_i$$

where

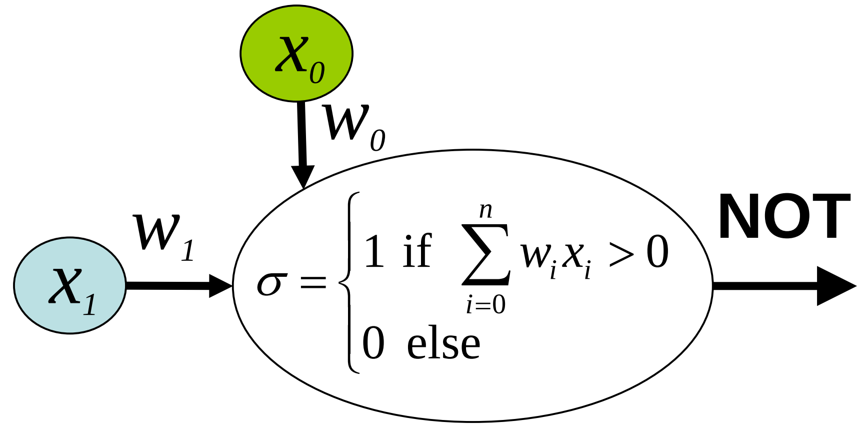
$$\Delta w_i = \eta(t - o)x_i$$

learning rate | target value | perceptron output | input value

Perception Training for NOT

Initialize:

$$w_0, w_1 = 0$$



$$w_i \leftarrow w_i + \Delta w_i$$

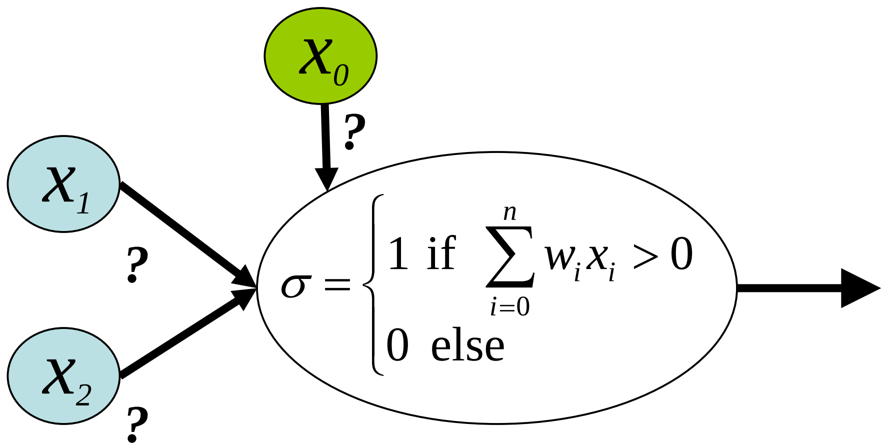
$$\Delta w_i = \eta(t - o)x_i$$

Work

Start

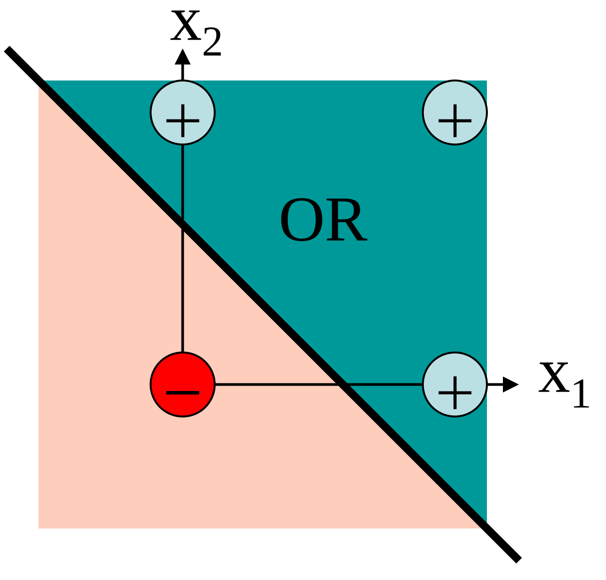
End

What weights make XOR?

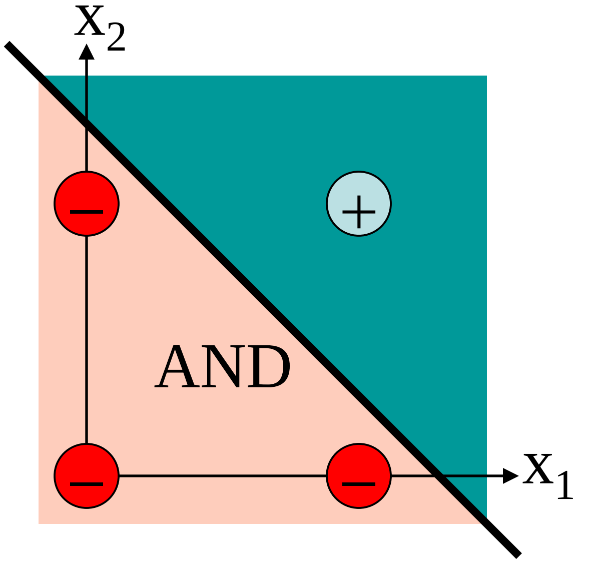


- No combination of weights works
- Perceptrons can only represent linearly separable functions

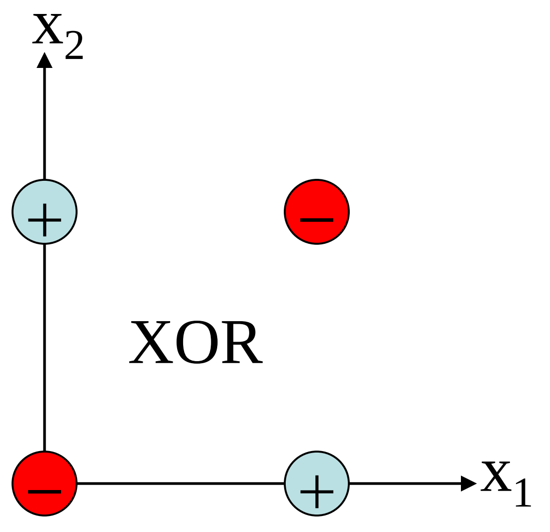
Linear Separability



Linear Separability



Linear Separability

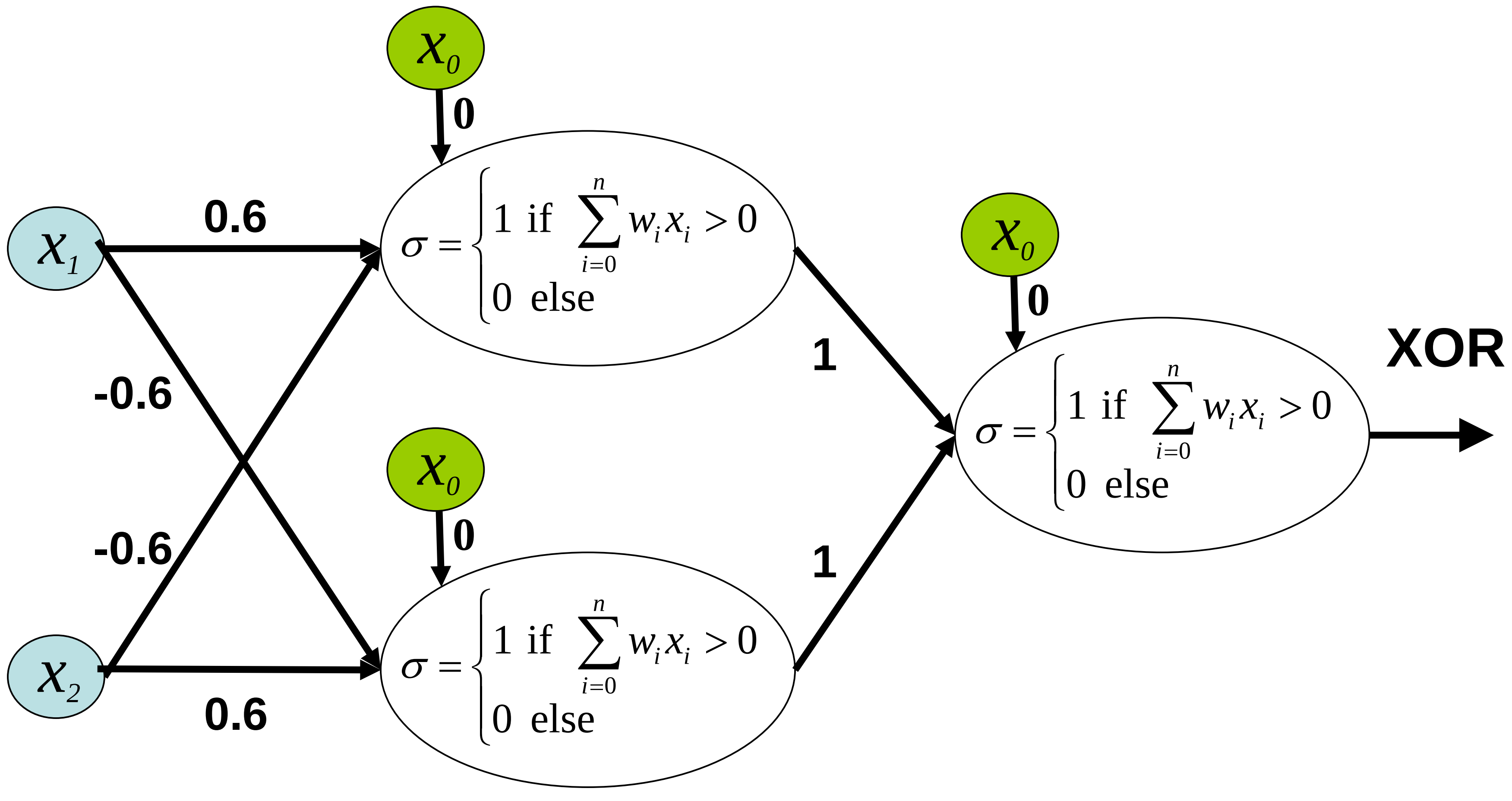


Perceptron Training Rule

- Converges to the correct classification IF
  - Cases are linearly separable
  - Learning rate is slow enough
  - Proved by Minsky and Papert in 1969

Killed widespread interest in perceptrons till the 80's

XOR

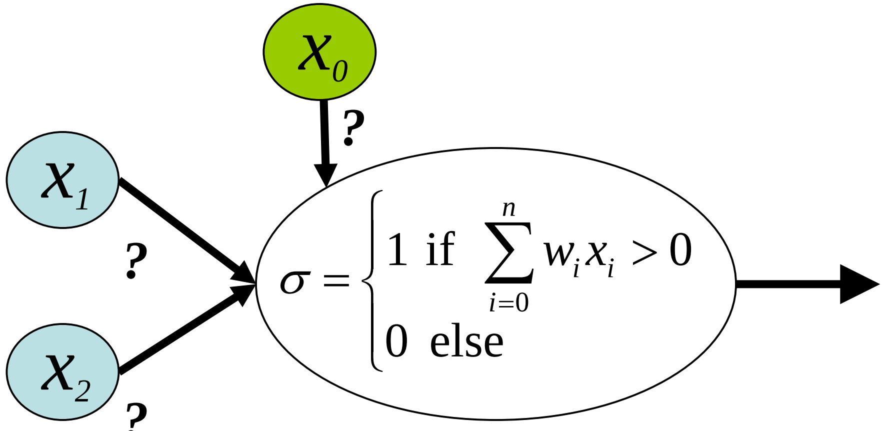


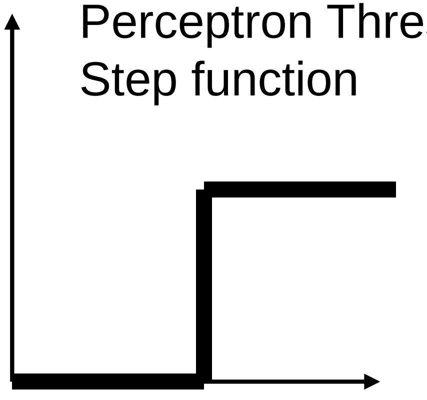
x_0
0
x_0
0
x_0
0
x_1
x_2
0.6
-0.6
-0.6
0.6
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$
$\sigma = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$
1
1
XOR

What’s wrong with perceptrons?

- You can always plug multiple perceptrons together to calculate any function.
- BUT...who decides what the weights are?
  - Assignment of error to parental inputs becomes a problem....

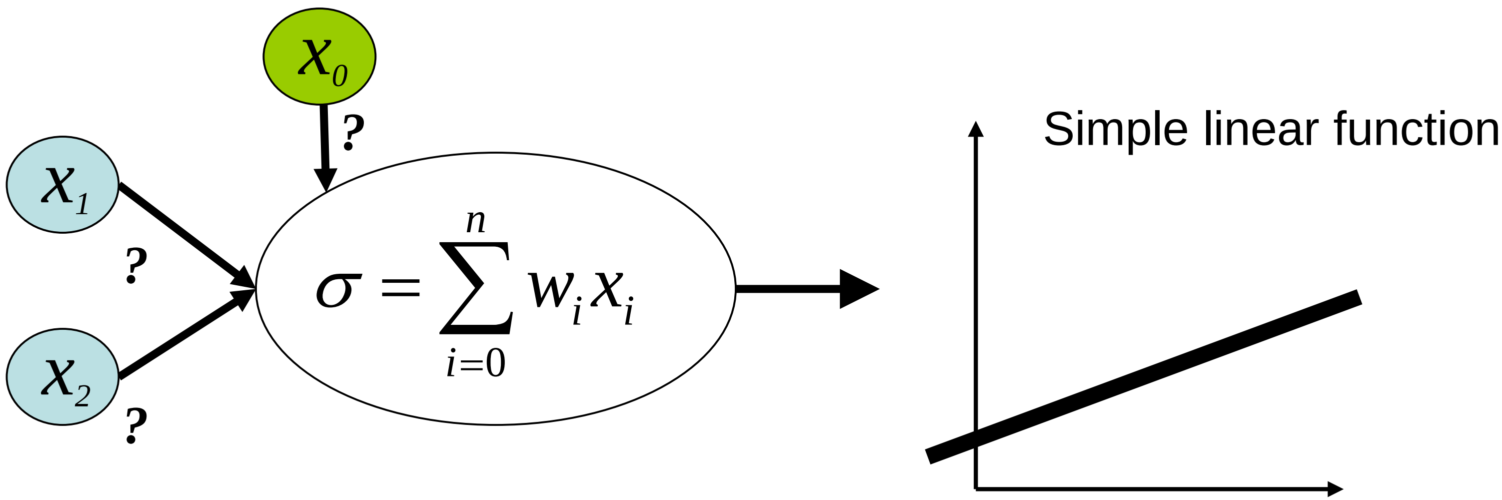
Perceptrons use a step function





- Small changes in inputs -> either no change or large change in output.

Solution: Differentiable Function



- Varying any input a little creates a perceptible change in the output
- We can now characterize how *error* changes w_i even in multi-layer case

Measuring error for linear units

- Output Function

$$\sigma(\vec{x}) = \vec{w} \cdot \vec{x}$$

- Error Measure:

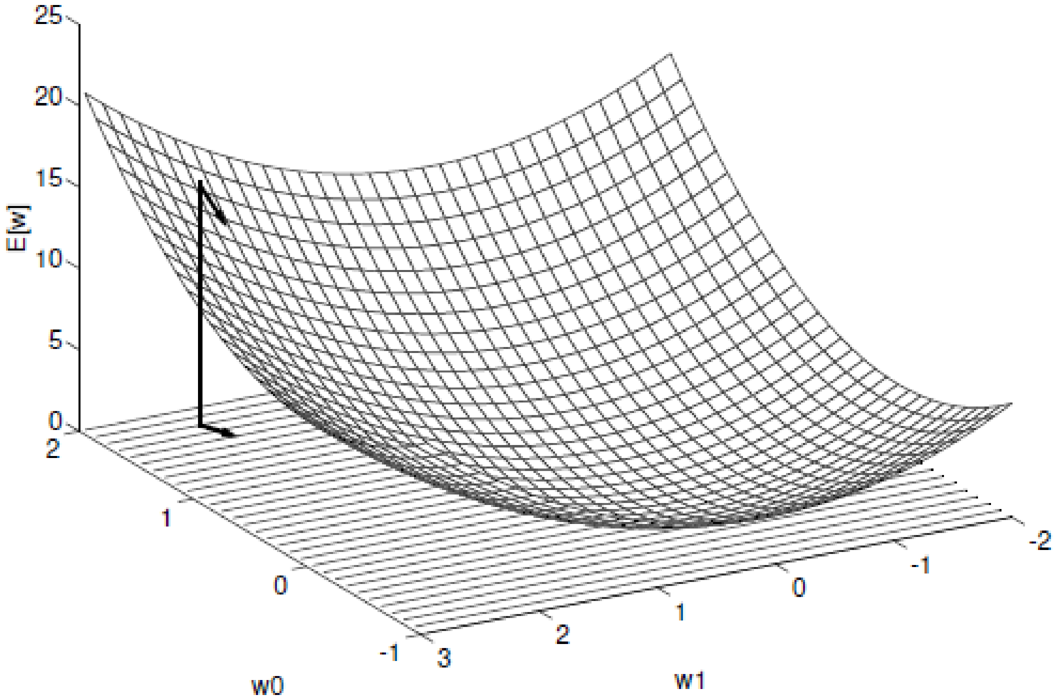
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

data

target value

linear unit output

Gradient Descent



Training rule:

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient:

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n} \right]$$

Gradient Descent Rule

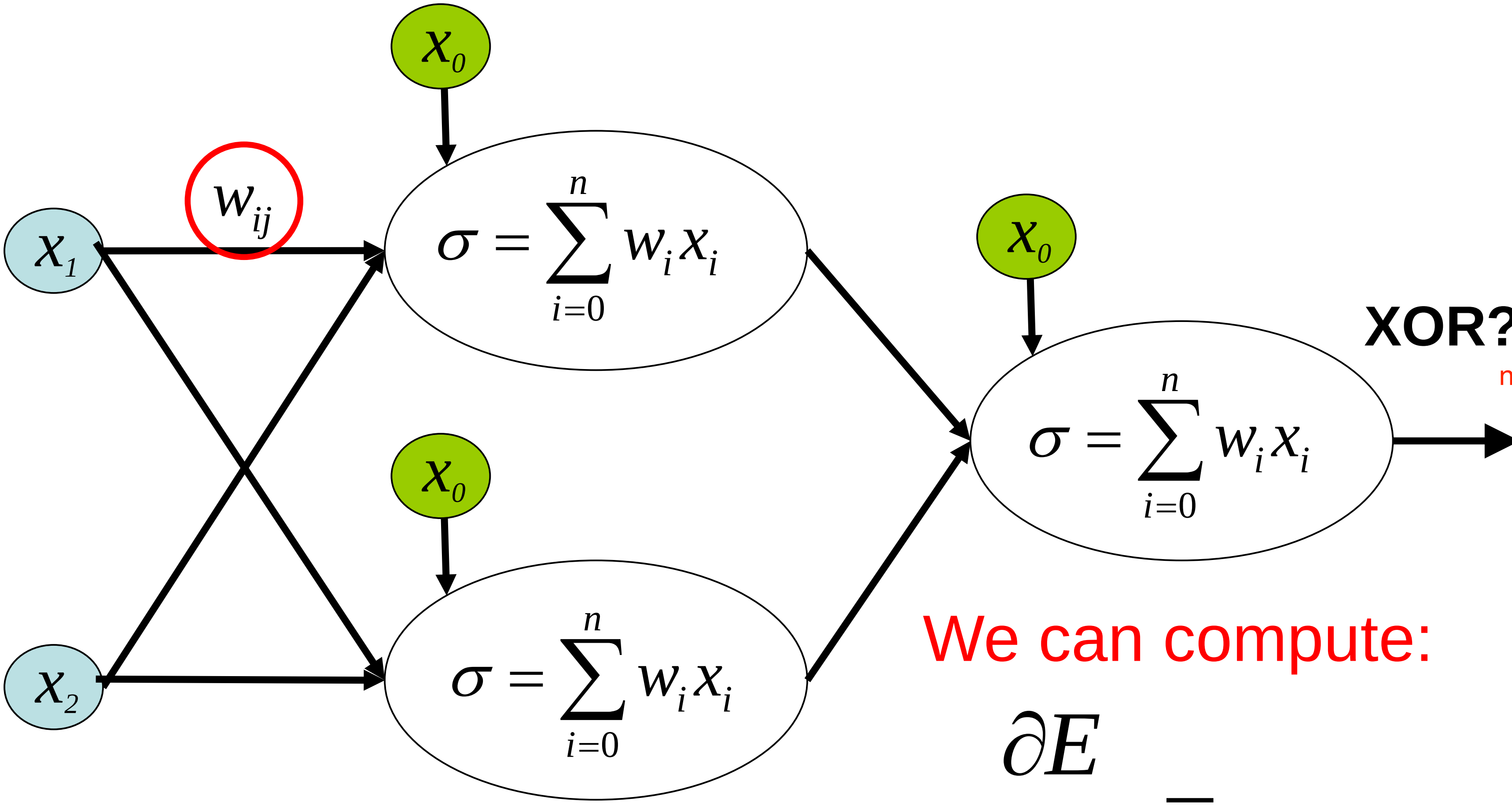
$$\frac{\partial E}{\partial w_i} \equiv \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \sum_{d \in D} (t_d - o_d)(-x_{i,d})$$

Update Rule:

$$w_i \leftarrow w_i + \eta \sum_{d \in D} (t_d - o_d) x_{i,d}$$

Gradient Descent for Multiple Layers



We can compute:

$$\frac{\partial E}{\partial w_{ij}} = \ldots$$

Gradient Descent vs. Perceptrons

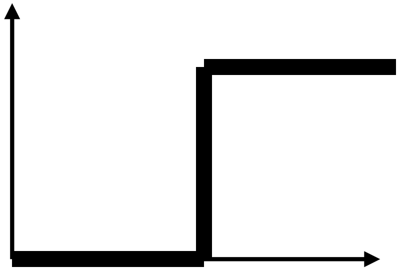
- Perceptron Rule & Threshold Units
  - Learner converges on an answer ONLY IF data is linearly separable
  - Can't assign proper error to parent nodes
- Gradient Descent
  - (locally) Minimizes error even if examples are not linearly separable
  - Works for multi-layer networks
    - But...linear units only make linear decision surfaces (can't learn XOR even with many layers)
  - And the step function isn't differentiable...

need a different activation function,
in particular one that is nonlinear AND differentiable

A compromise function

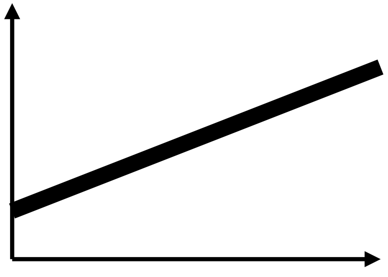
- Perceptron

$$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 \text{ else} \end{cases}$$



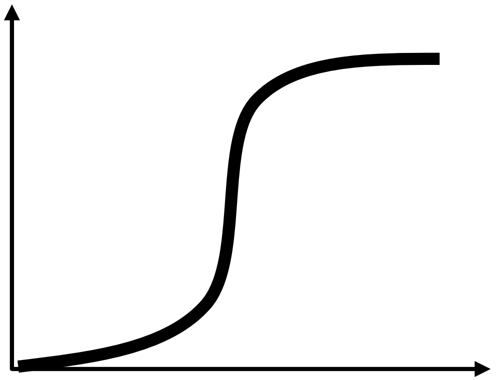
- Linear

$$output = net = \sum_{i=0}^{n} w_i x_i$$



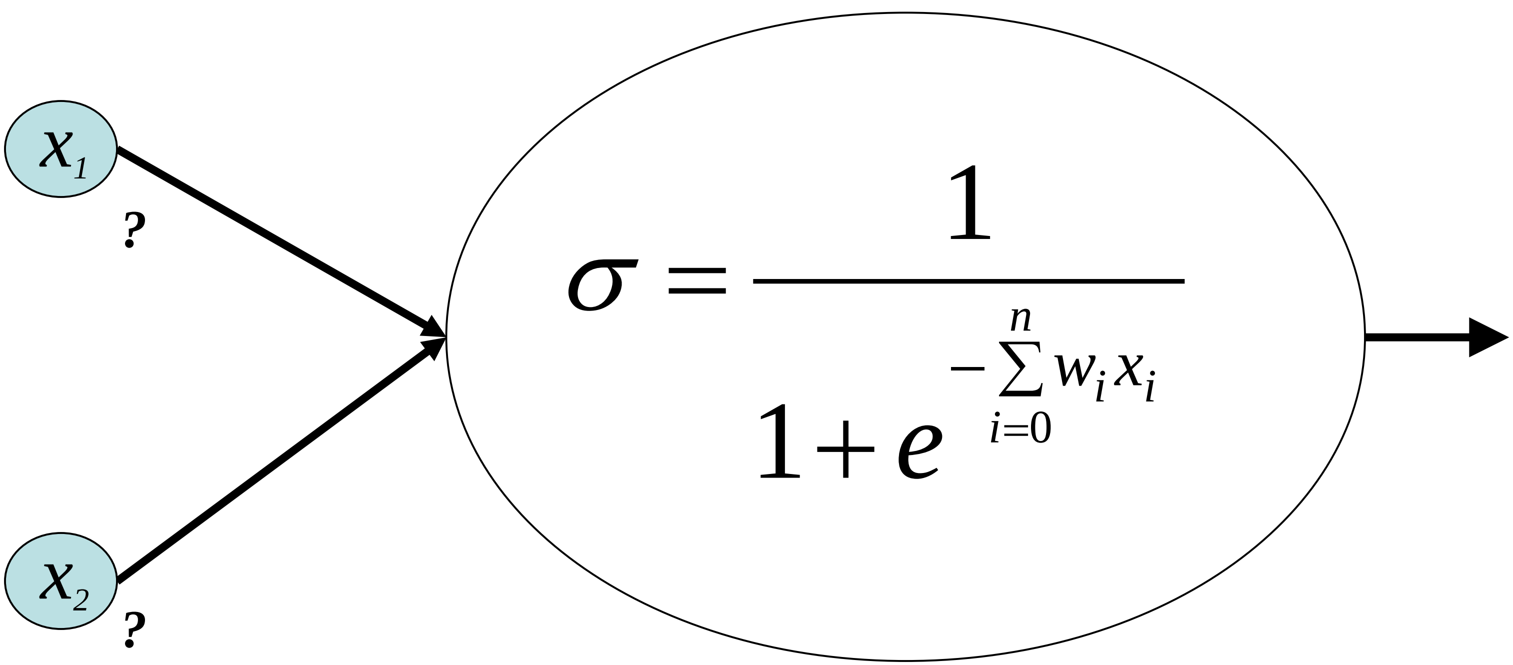
- Sigmoid (Logistic)

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

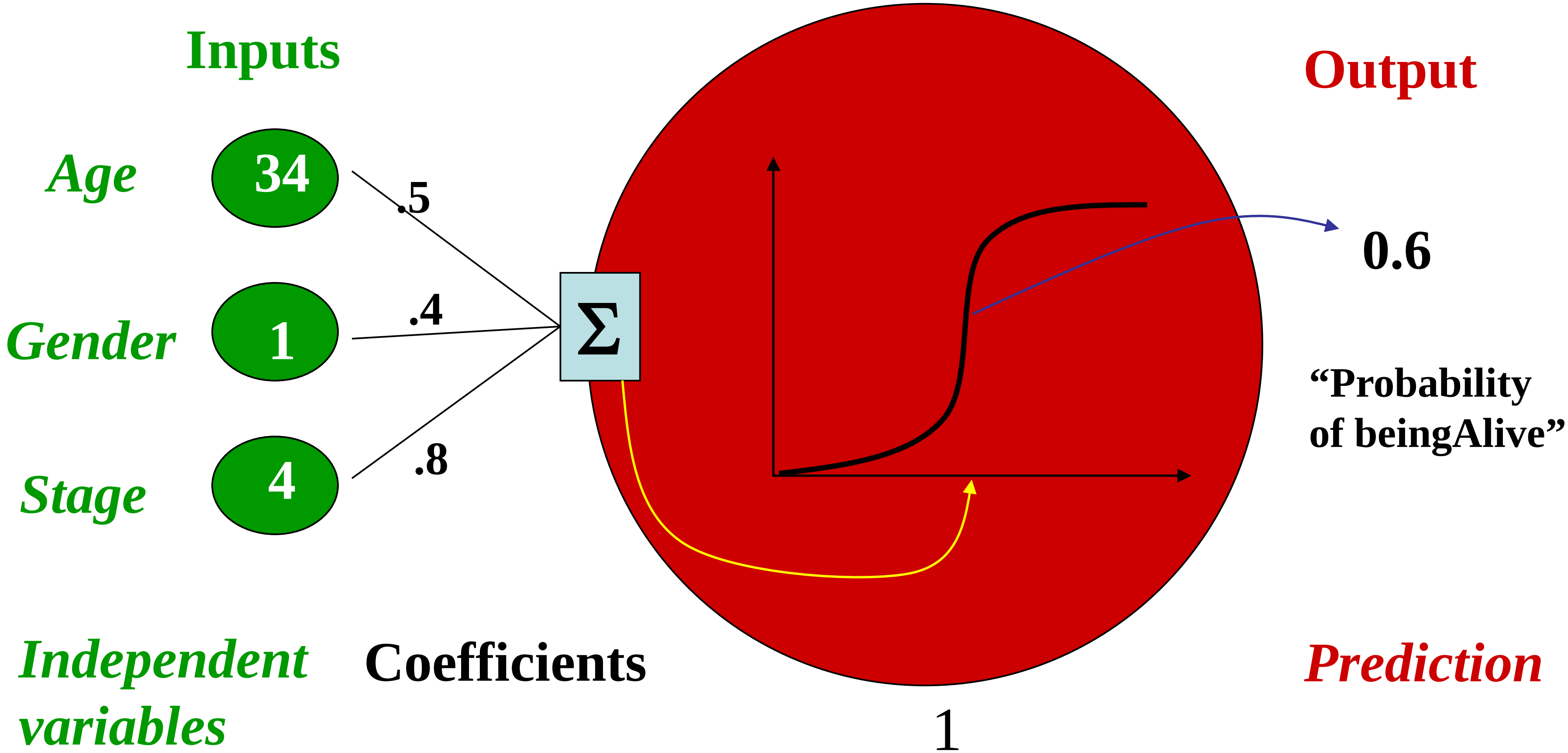


The sigmoid (logistic) unit

- Has differentiable function
  - Allows gradient descent
- Can be used to learn non-linear functions

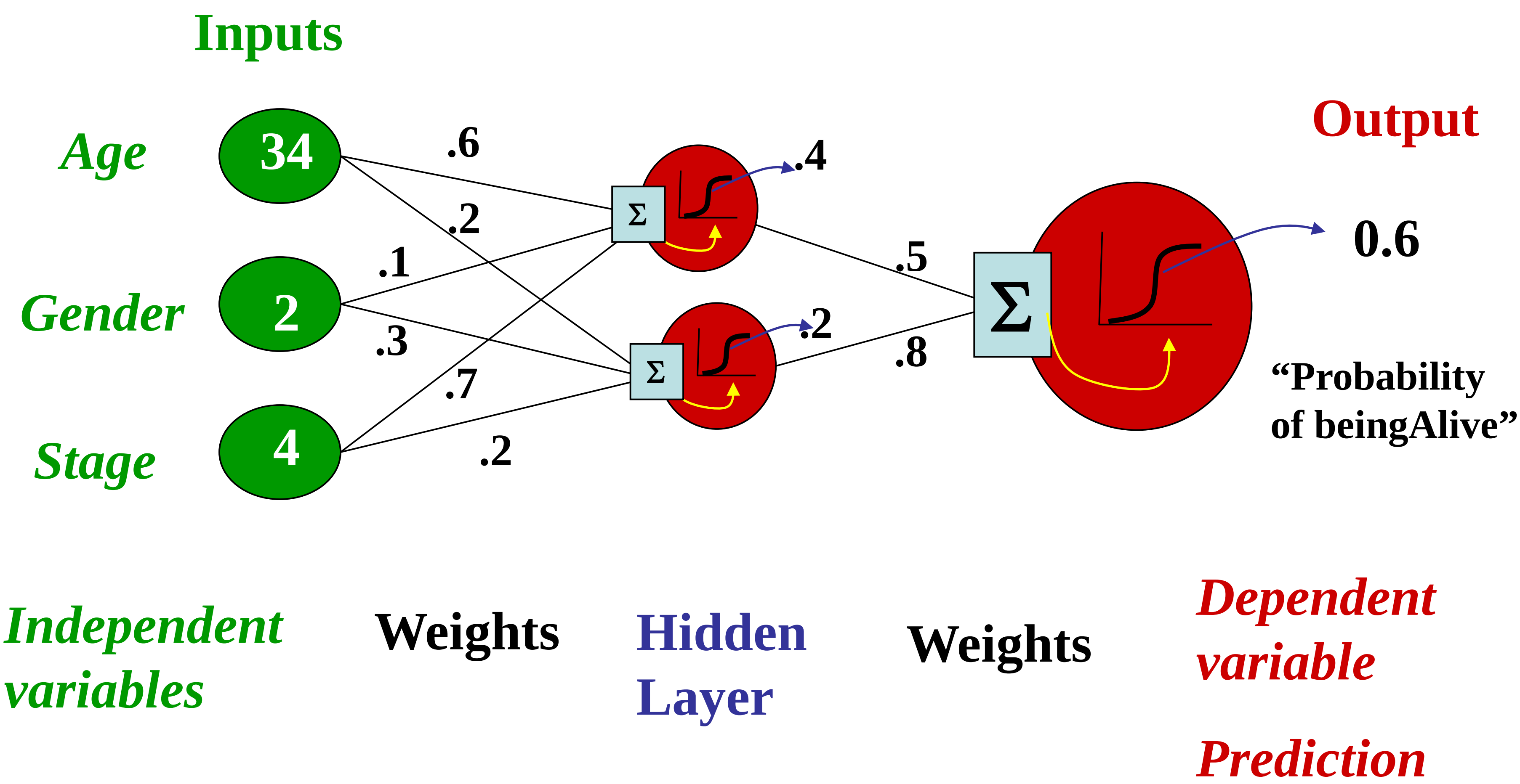


Logistic function

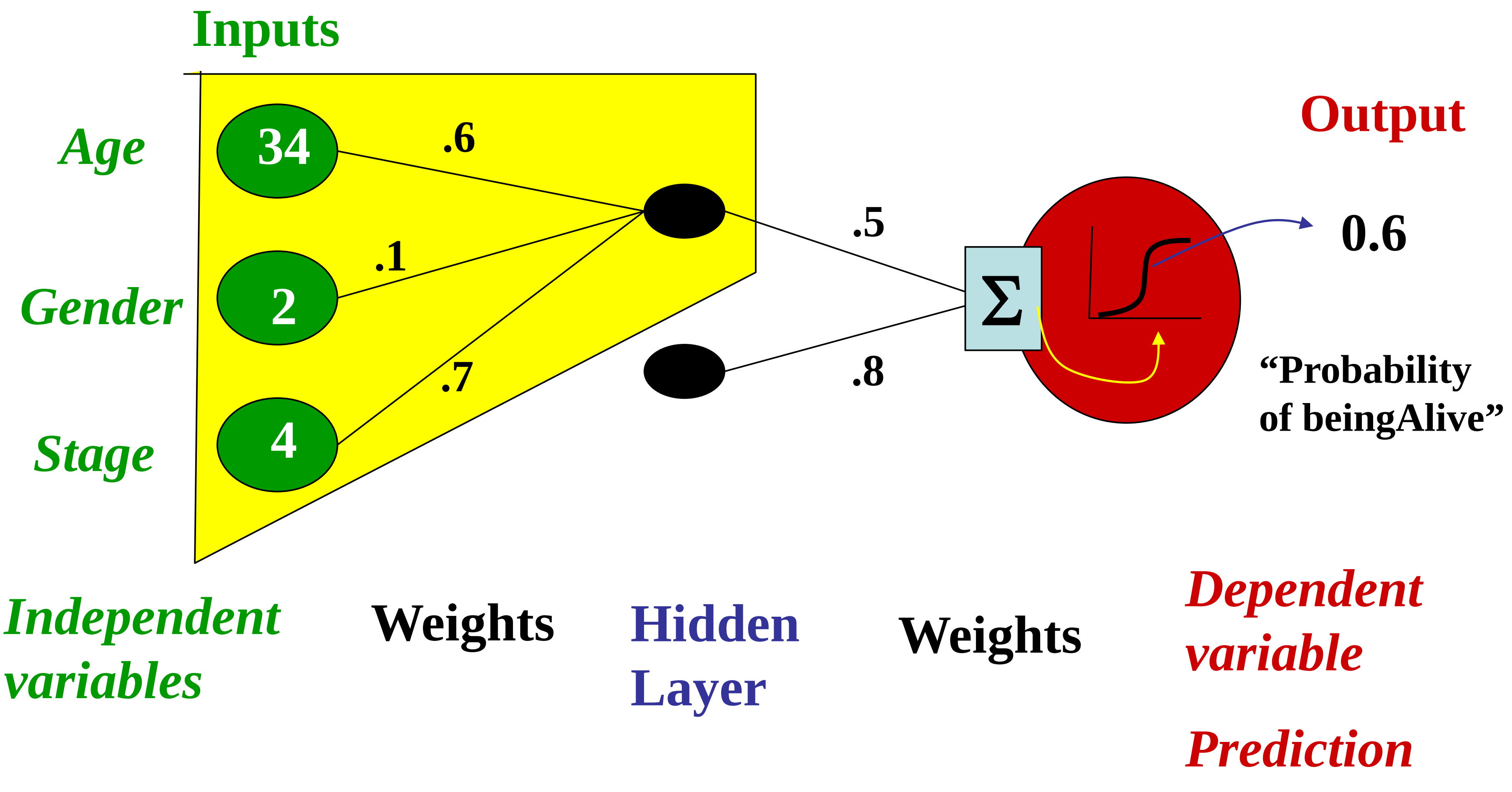


$$\sigma = \frac{1}{1 + e^{-\sum_{i=0}^{n} w_i x_i}}$$

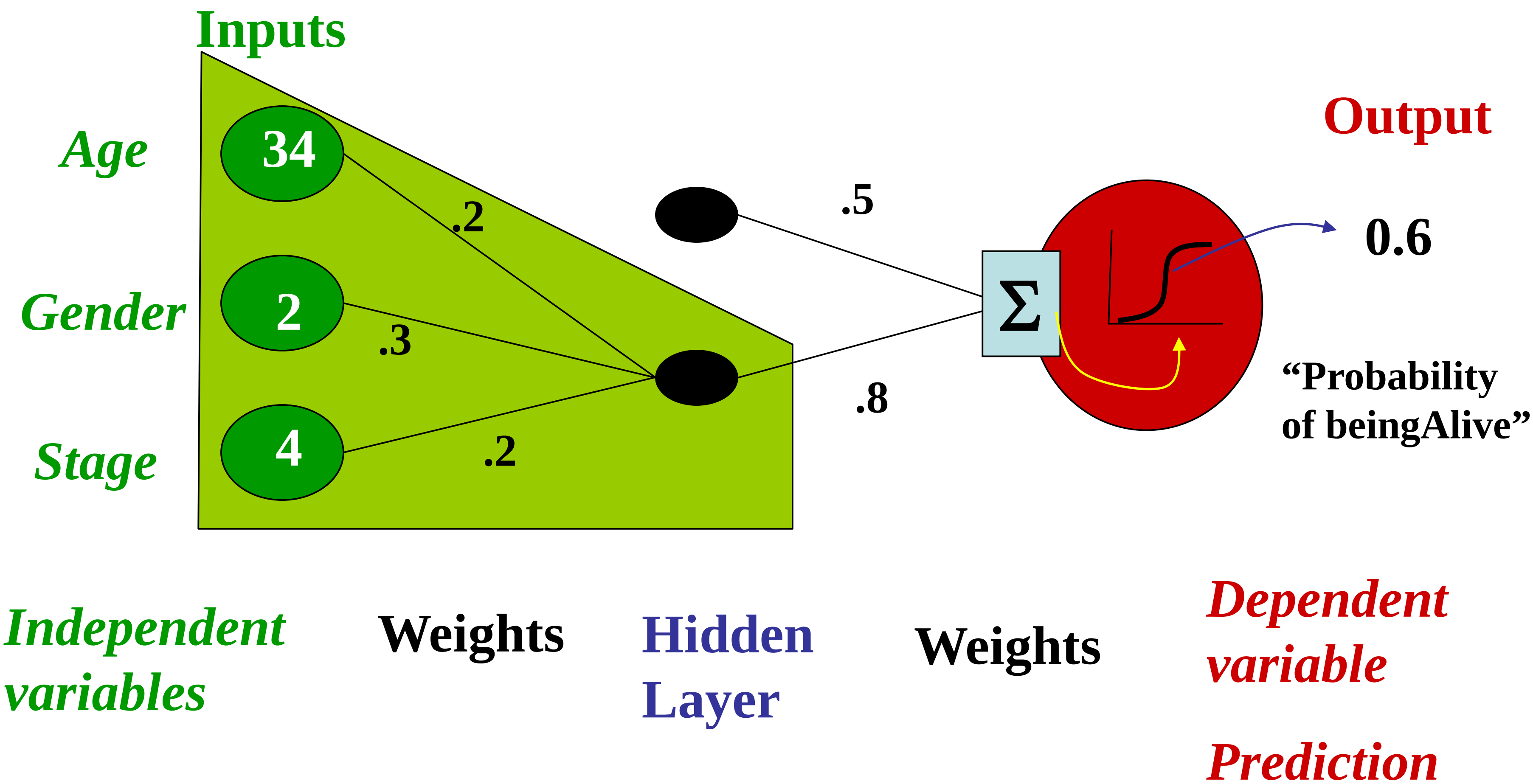
Neural Network Model



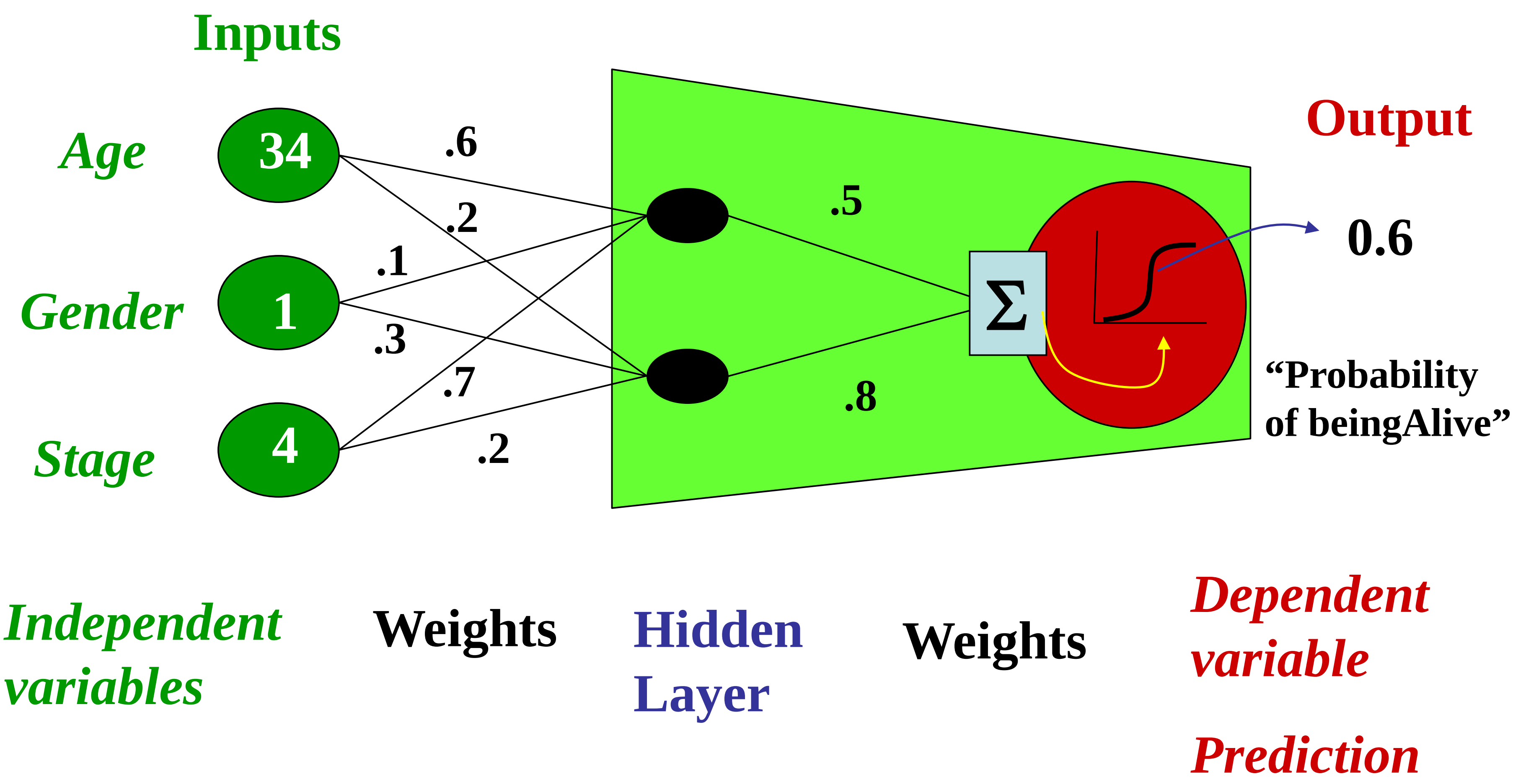
Getting an answer from a NN



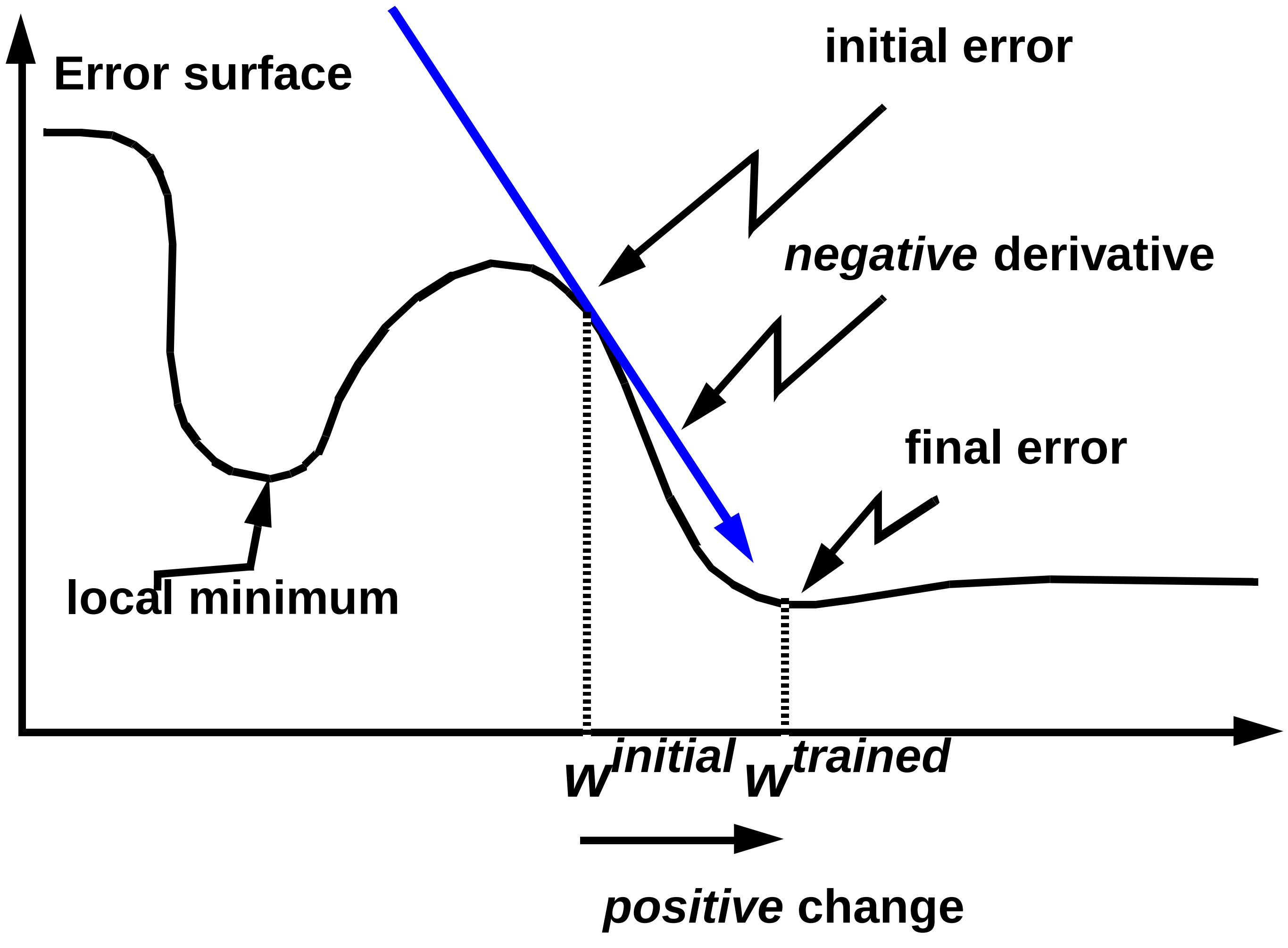
Getting an answer from a NN



Getting an answer from a NN



Minimizing the Error



Differentiability is key!

- Sigmoid is easy to differentiate

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

- For gradient descent on multiple layers, a little dynamic programming can help:
  - Compute errors at each output node
  - Use these to compute errors at each hidden node
  - Use these to compute weight gradient

The Backpropagation Algorithm

For each input training example, $\left\langle \vec{x}, \vec{t} \right\rangle$

1. Input instance $\vec{x}$ to the network and compute the output o_u for every unit u in the network

2. For each output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

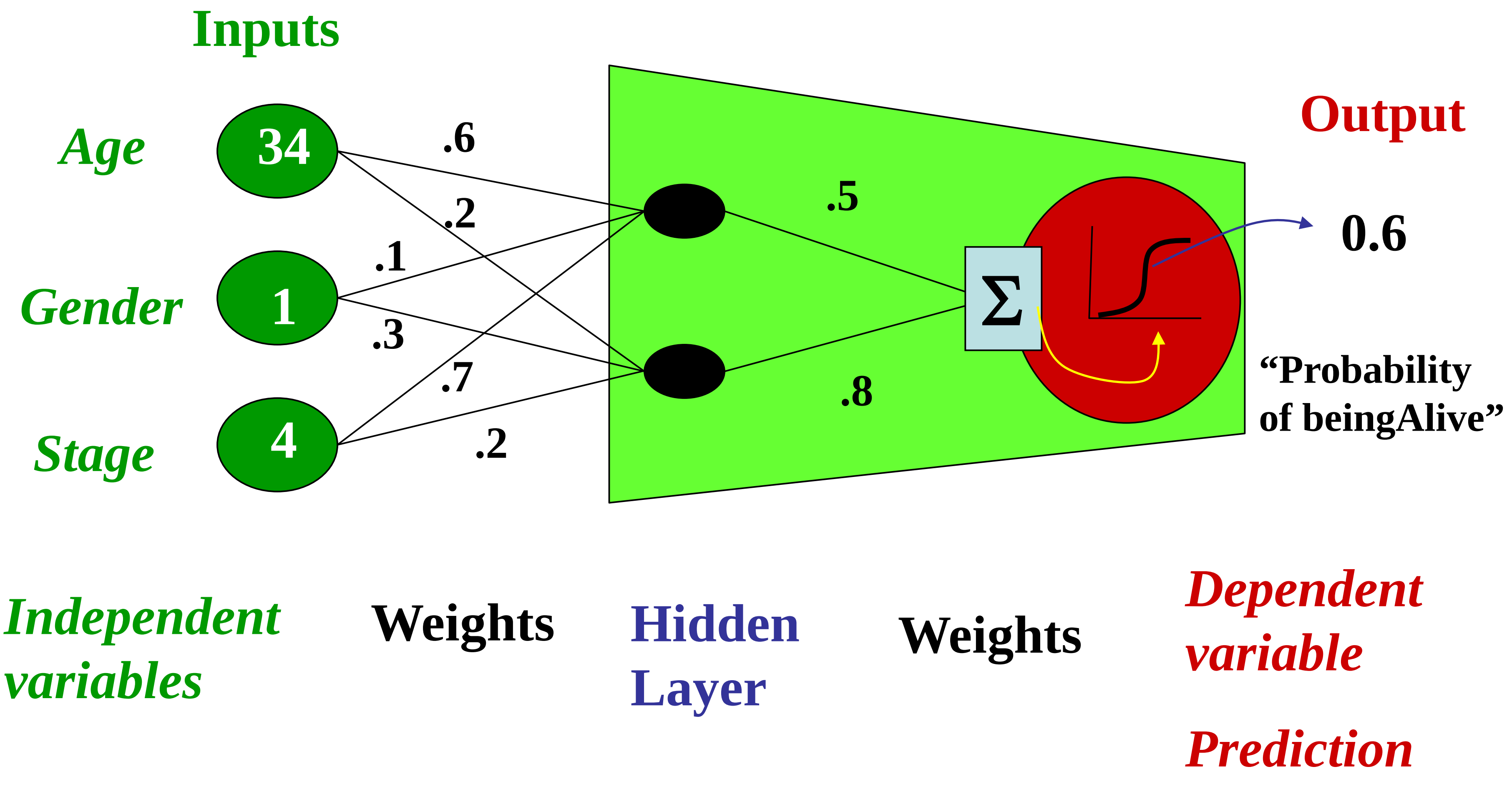
3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{hk}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \eta\delta_i x_{ji}$$

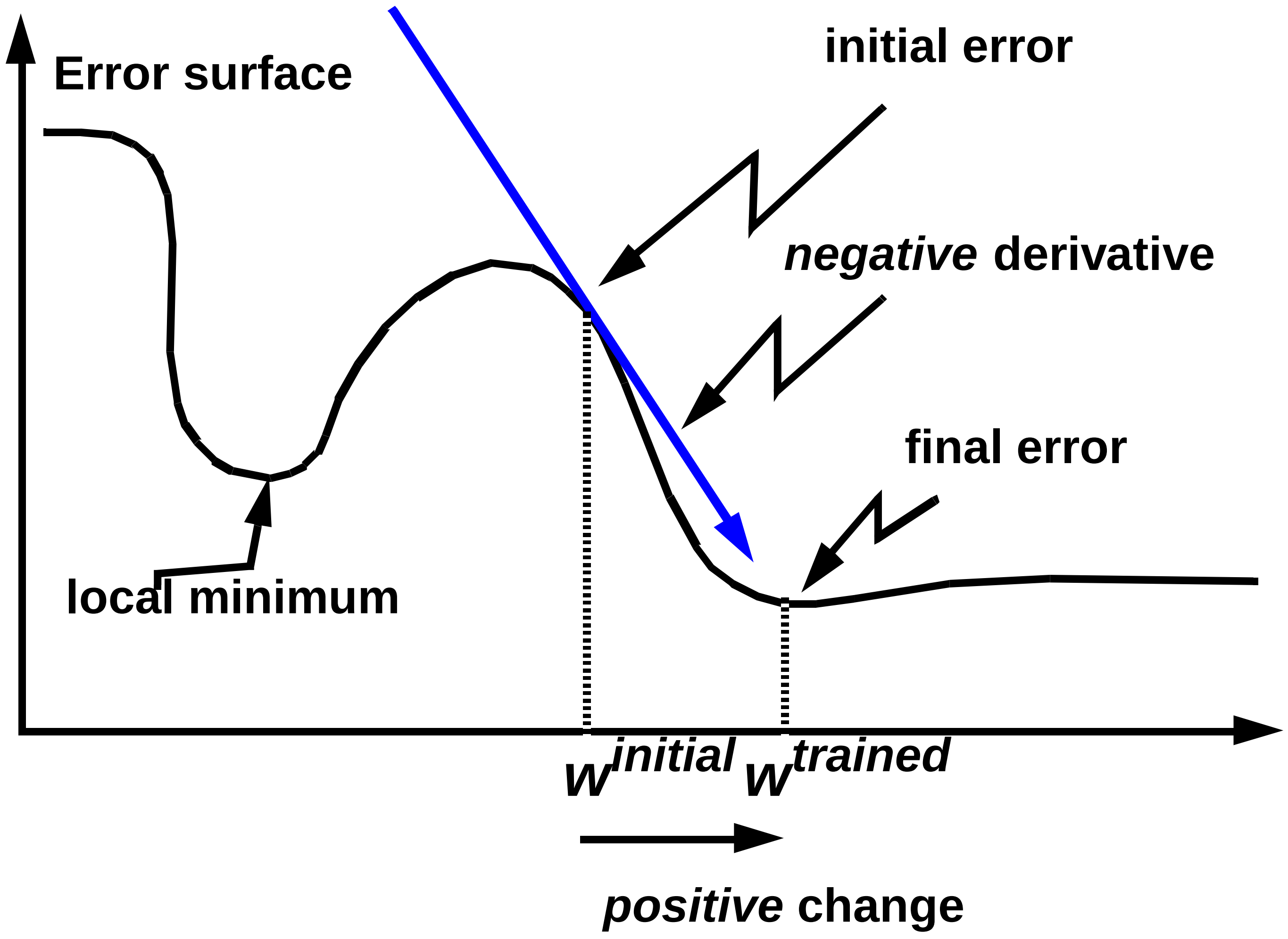
Learning Weights



The fine print

- Don't implement back-propagation
  - Use a package
  - Second-order or variable step-size optimization techniques exist
- Feature normalization
  - Typical to normalize inputs to lie in [0,1]
    - (and outputs must be normalized)
- Problems with NN training:
  - Slow training times (though, getting better)
  - ~~Local minima~~

Minimizing the Error



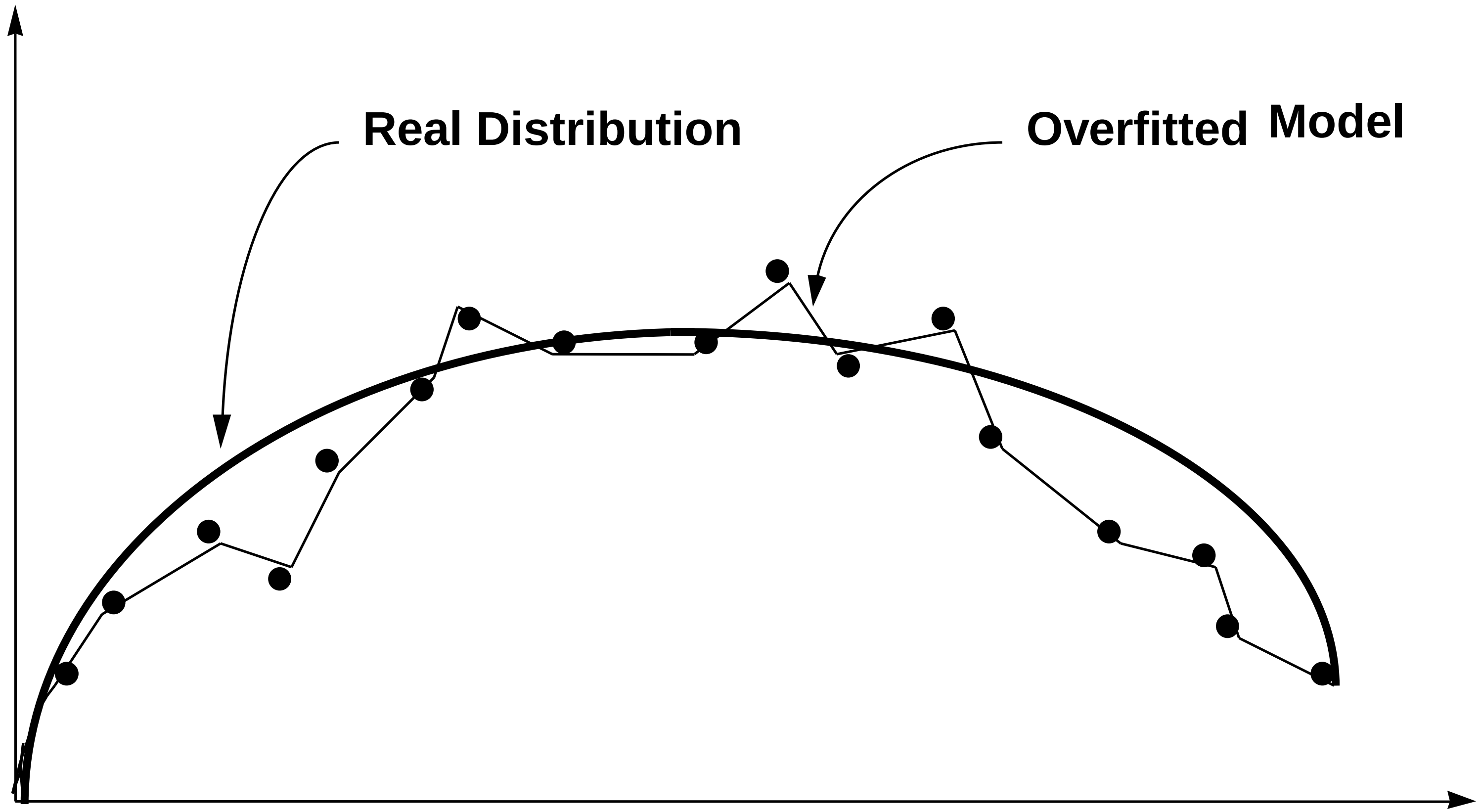
Expressive Power of ANNs

- Universal Function Approximator:
  - Given enough hidden units, can approximate *any* continuous function *f*
- Need 2+ hidden units to learn XOR

computation time & overfitting can be a problem with lots and lots of hidden units

- Why not use millions of hidden units?
  - Efficiency (training is slow)
  - Overfitting

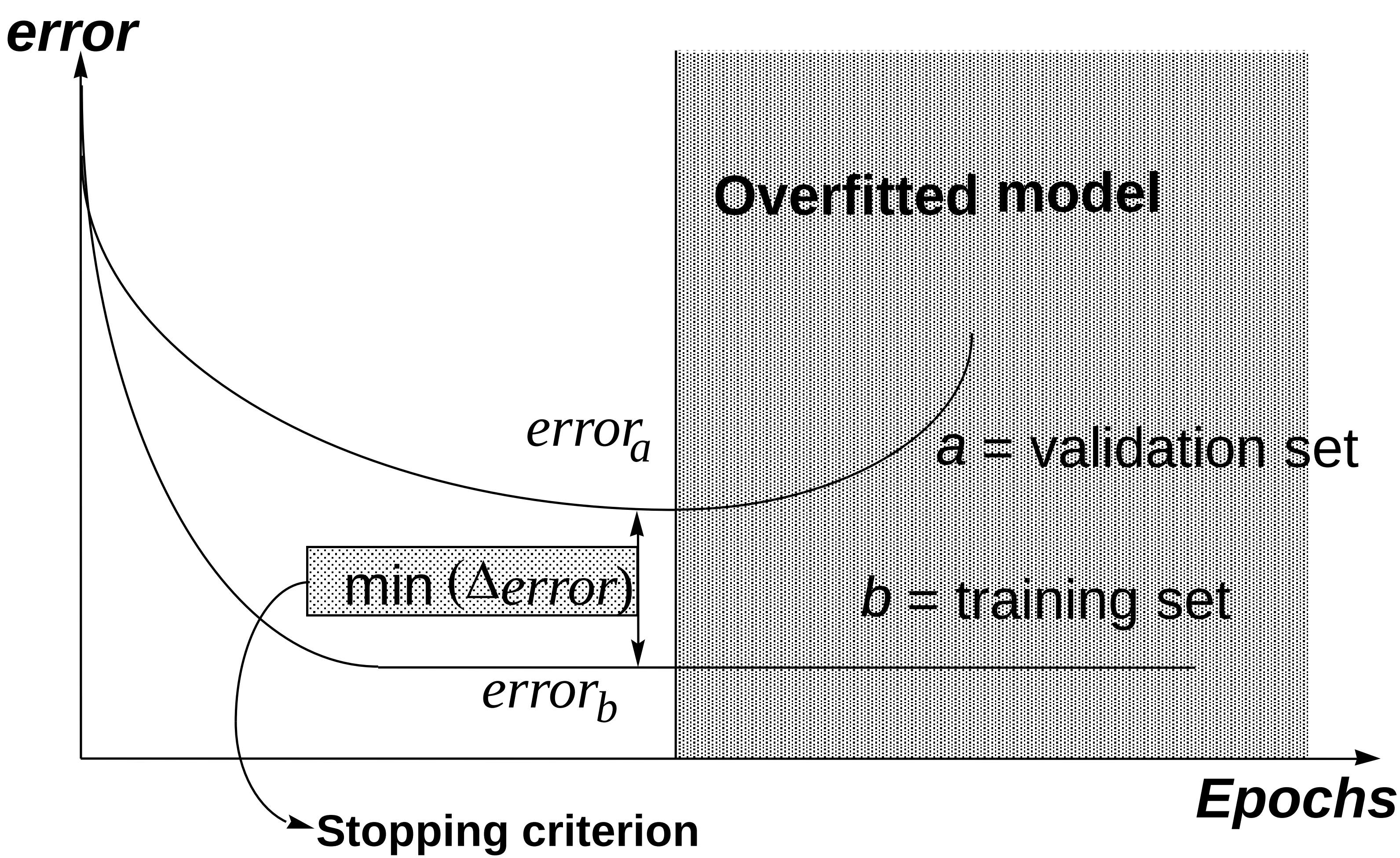
Overfitting



Combating Overfitting in Neural Nets

- Many techniques
- Two popular ones:
  - Early Stopping (most popular)
    - Use “a lot” of hidden units
    - Just don’t over-train
  - Cross-validation
    - Test different architectures to choose “right” number of hidden units

Early Stopping



Learning Rate?

- A “knob” you twist empirically
  - Important
- One popular option: look for validation set acc to decrease/stabilize, then halve learning rate

Modern Neural Networks (Deep Nets)

Local minima in large networks is less of an issue

Early stopping is useful, but so is initializing at zero and training until almost zero training error

count on stochastic gradient descent to perform “implicit regularization”

Also: Dropout

Many layers are now common

And specific structure: convolution, max pooling

Summary of Neural Networks

When are Neural Networks useful?

- Instances represented by attribute-value pairs
  - Particularly when attributes are real valued
- The target function is
  - Discrete-valued
  - Real-valued
  - Vector-valued
- Training examples may contain errors
- Fast evaluation times are necessary

When not?

- Fast training times are necessary
- Understandability of the function is required

Summary of Neural Networks

Non-linear regression technique that is trained with gradient descent.

Question: How important is the biological metaphor?

Other Topics in Neural Nets

- Batch Move vs. stochastic
- Auto-Encoders
- Neural Networks on Silicon

Stochastic vs. Batch Mode

stochastic is more common:
-has a randomness that will prevent
us from getting stuck in minima

Stochastic Gradient Descent

Do until satisfied

- For each training example d in D

  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

stochastic is faster:
-say 1mil examples, train on 1mil/min, train on an hour:
stochastic mode : 60 million, but
batch mode : 60 weight updates (60 gradients)

Batch mode Gradient Descent:

Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Incremental vs. Batch Mode

- In Batch Mode we minimize:

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- Same as computing: $\Delta w_D = \sum_{d \in D} \Delta w_d$

- Then setting $w = w + \Delta w_D$

Advanced Topics in Neural Nets

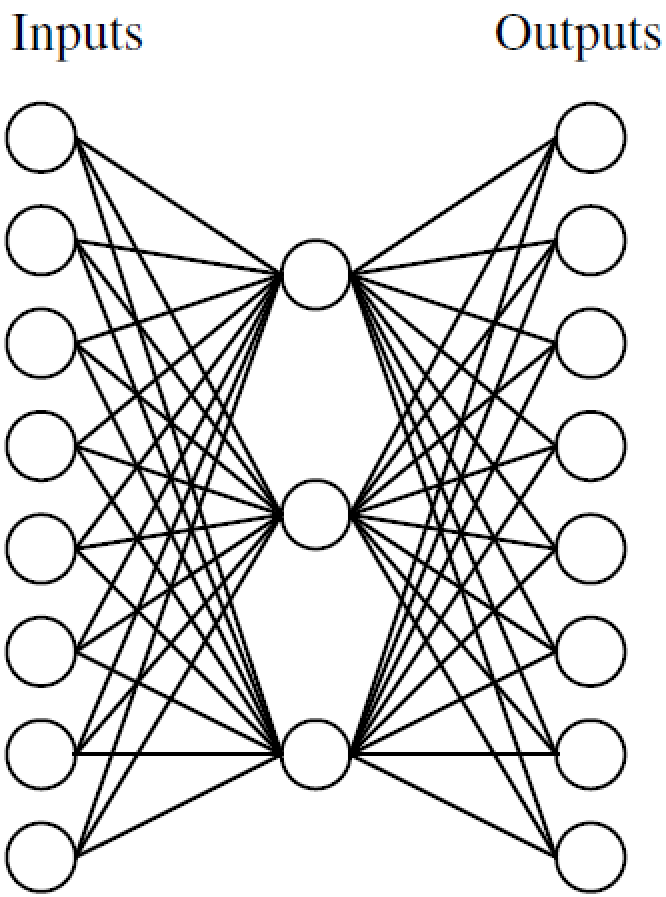
- Batch Move vs. incremental
- Auto-Encoders
- Neural Networks on Silicon

Hidden Layer Representations

- Input->Hidden Layer mapping:
  - representation of input vectors tailored to the task
- Can also be exploited for *dimensionality reduction*
  - Form of unsupervised learning in which we output a “more compact” representation of input vectors
  - $<x_1, \ldots, x_n> \rightarrow <x'_1, \ldots, x'_m>$ where $m < n$
  - Useful for visualization, problem simplification, data compression, etc.

Dimensionality Reduction

Model:



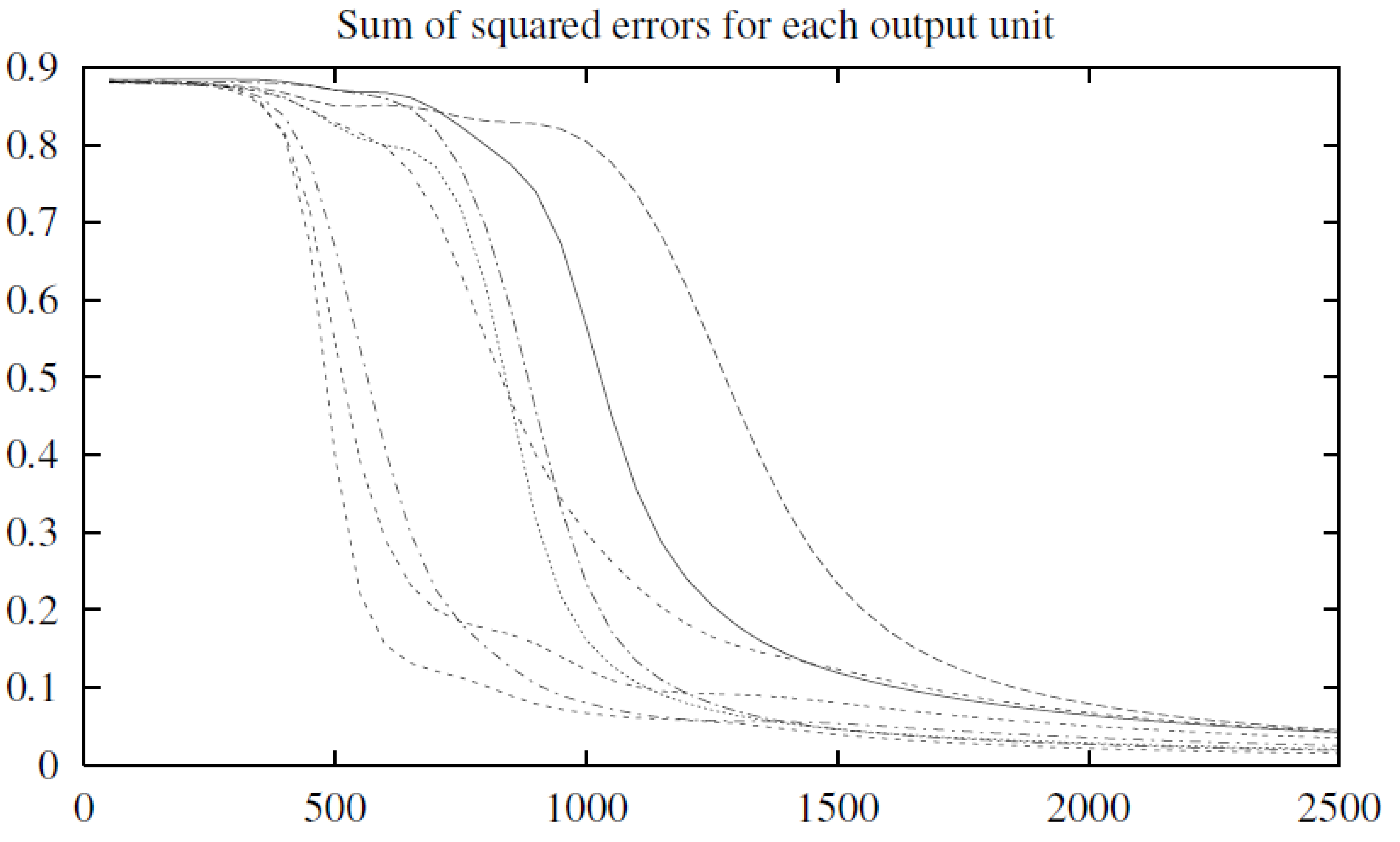
Function to learn:

Input		Output
10000000	$\rightarrow$	10000000
01000000	$\rightarrow$	01000000
00100000	$\rightarrow$	00100000
00010000	$\rightarrow$	00010000
00001000	$\rightarrow$	00001000
00000100	$\rightarrow$	00000100
00000010	$\rightarrow$	00000010
00000001	$\rightarrow$	00000001

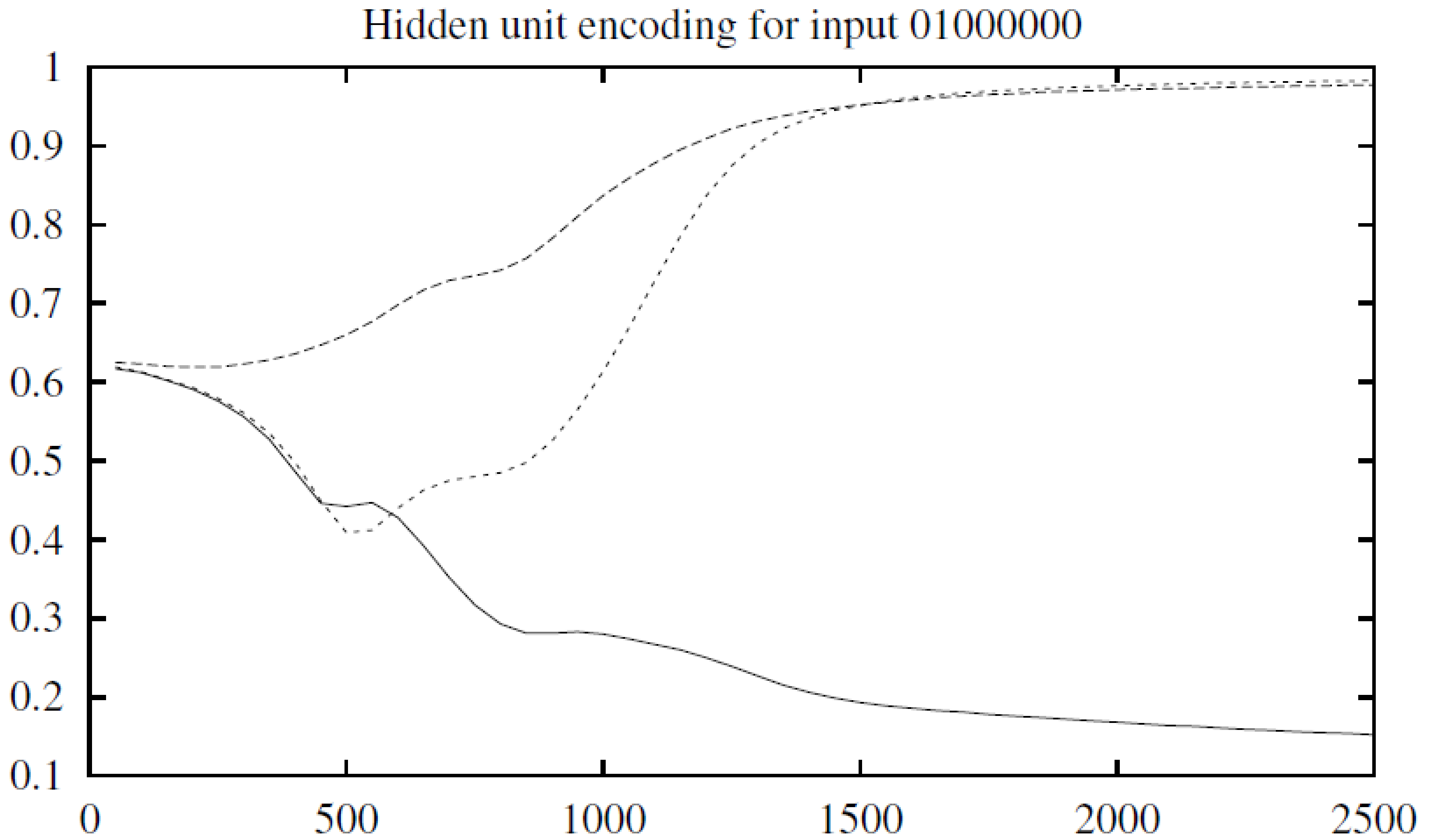
Dimensionality Reduction: Example

Input		Hidden Values				Output
10000000	$\rightarrow$	.89	.04	.08	$\rightarrow$	10000000
01000000	$\rightarrow$	.01	.11	.88	$\rightarrow$	01000000
00100000	$\rightarrow$	.01	.97	.27	$\rightarrow$	00100000
00010000	$\rightarrow$	.99	.97	.71	$\rightarrow$	00010000
00001000	$\rightarrow$	.03	.05	.02	$\rightarrow$	00001000
00000100	$\rightarrow$	.22	.99	.99	$\rightarrow$	00000100
00000010	$\rightarrow$	.80	.01	.98	$\rightarrow$	00000010
00000001	$\rightarrow$	.60	.94	.01	$\rightarrow$	00000001

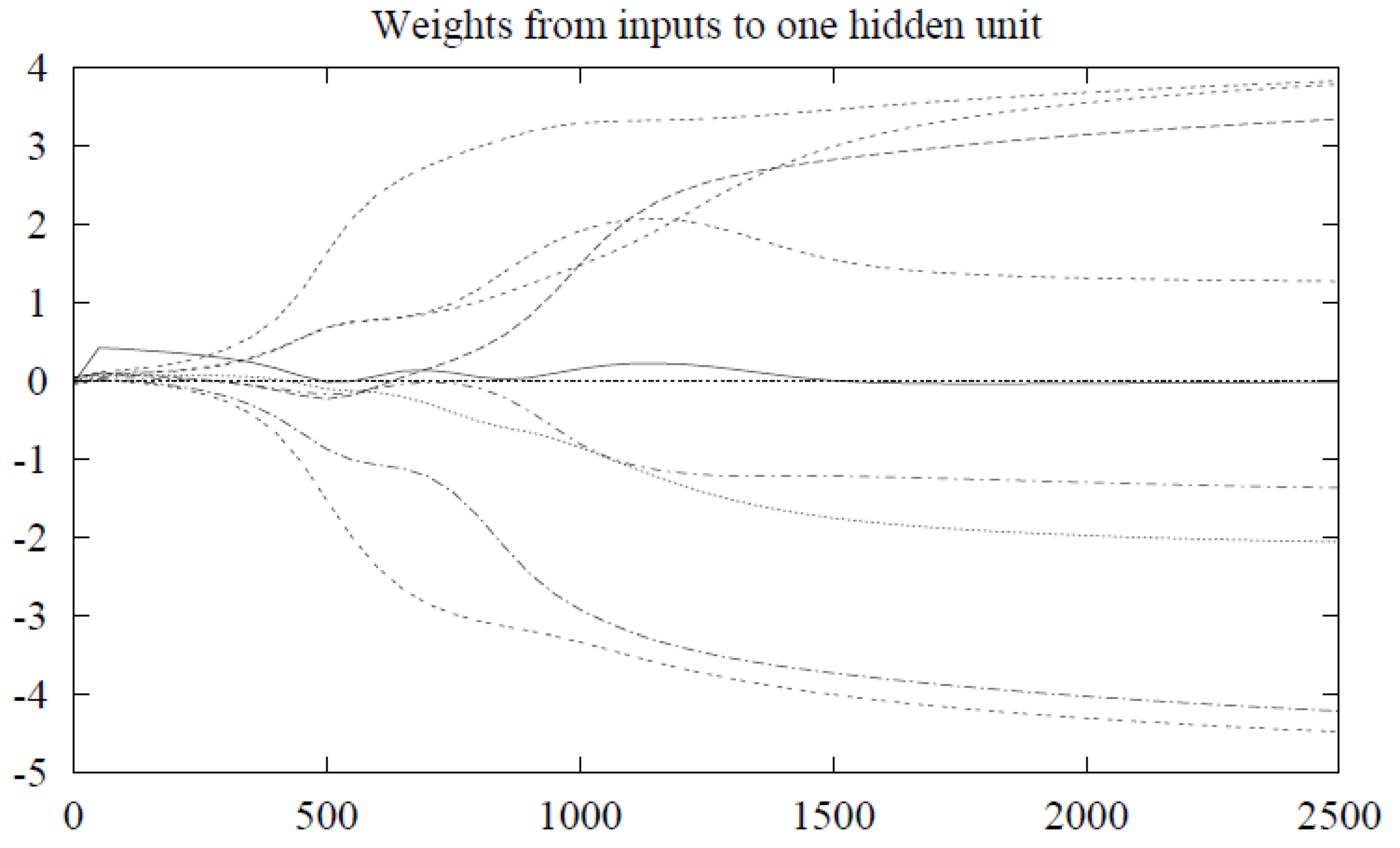
Dimensionality Reduction: Example



Dimensionality Reduction: Example



Dimensionality Reduction: Example

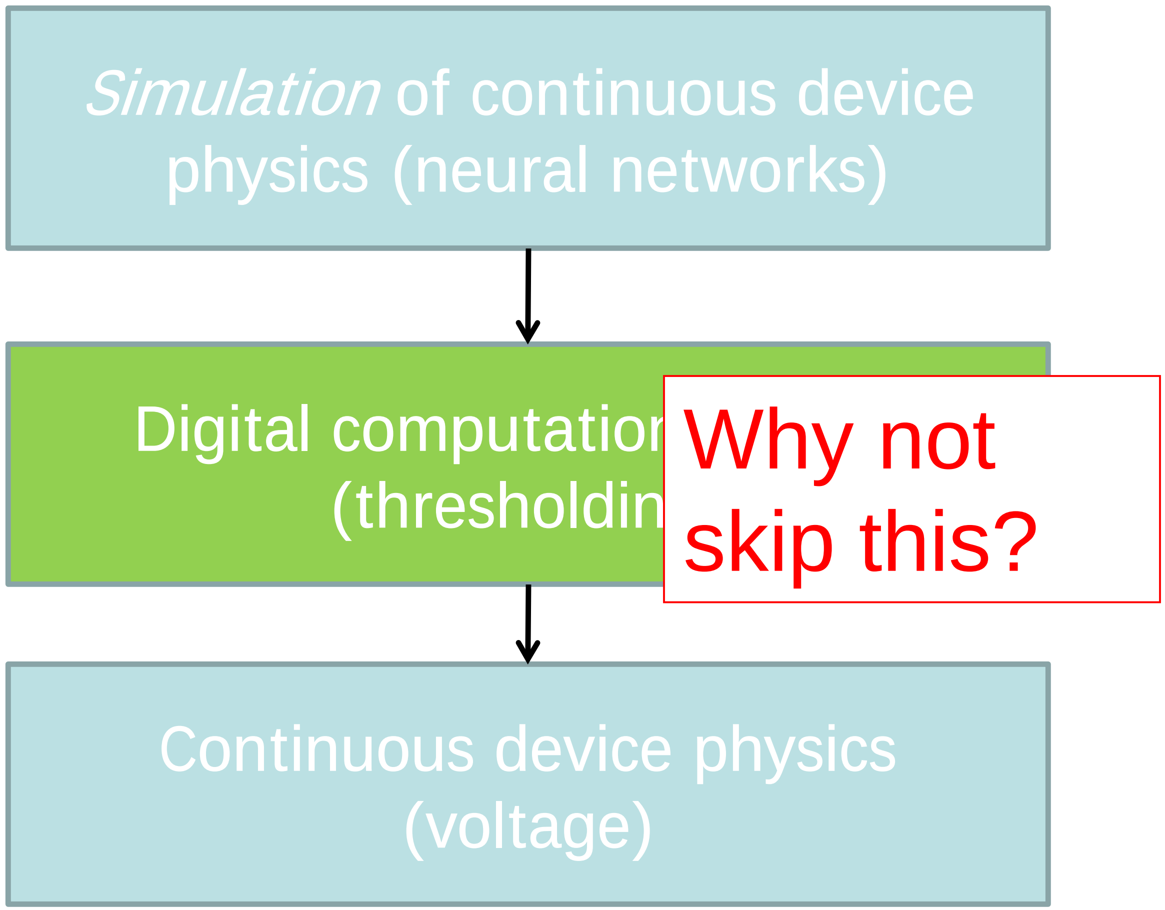


Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-encoders
- Neural Networks on Silicon

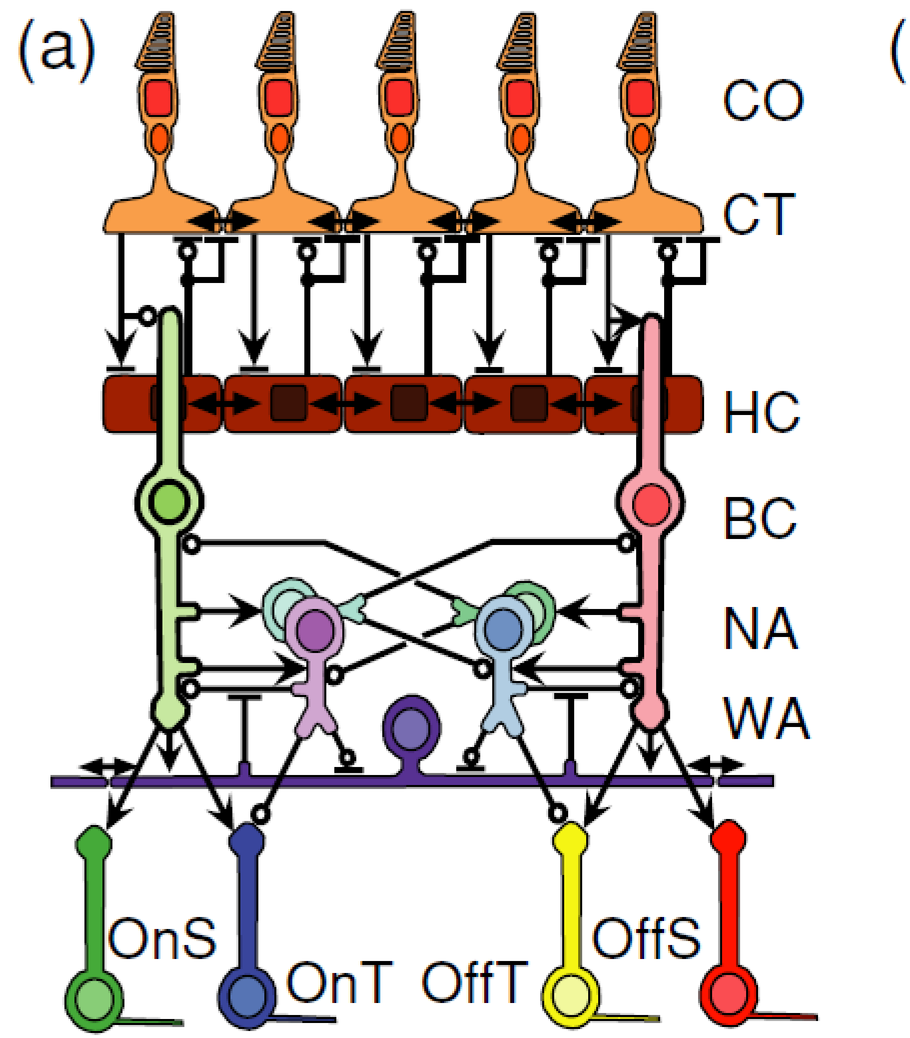
Neural Networks on Silicon

- Currently:



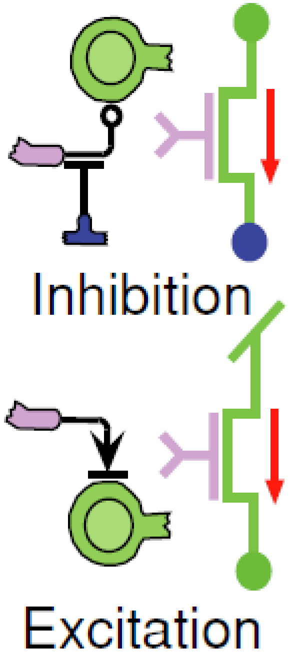
Example: Silicon Retina

- Simulates function of biological retina
- Single-transistor synapses adapt to luminance, temporal contrast
- Modeling retina directly on chip => requires 100x less power!

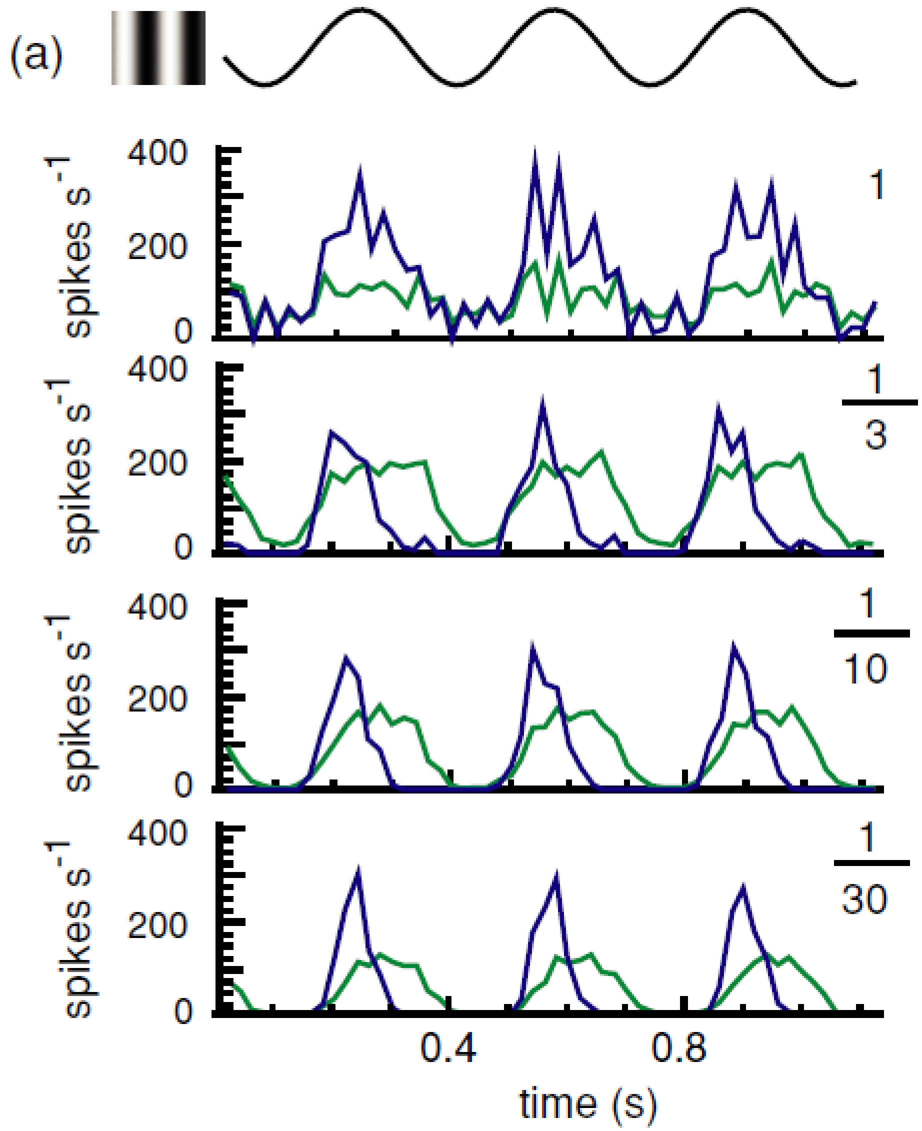


Example: Silicon Retina

- Synapses modeled with single transistors

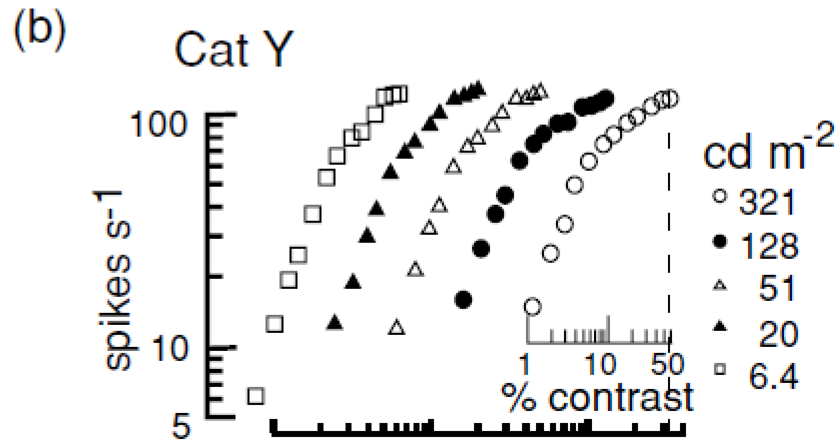


Luminance Adaptation

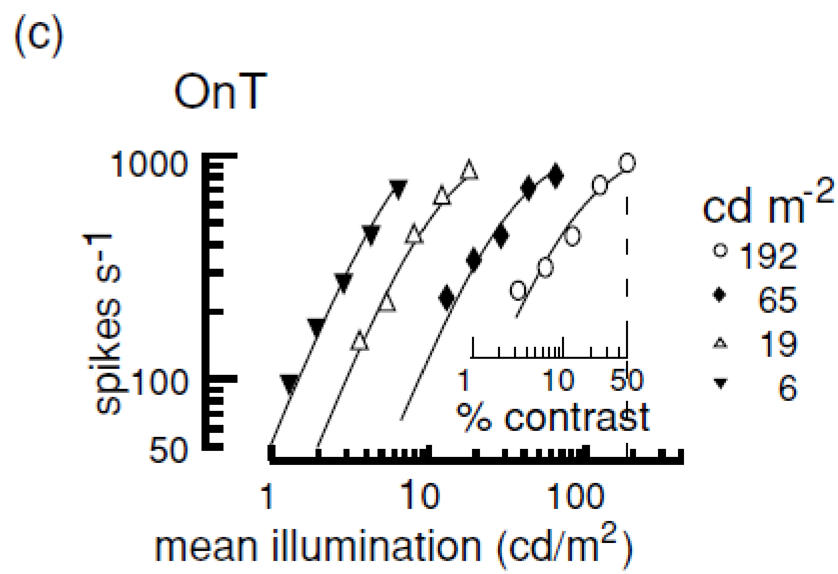


Comparison with Mammal Data

- Real:



- Artificial:



- Graphics and results taken from:

INSTITUTE OF PHYSICS PUBLISHING JOURNAL OF NEURAL ENGINEERING

J. Neural Eng. **3** (2006) 257–267 doi:10.1088/1741-2560/3/4/002

A silicon retina that reproduces signals in the optic nerve

Kareem A Zaghloul[1] and Kwabena Boahen[2,3]

General NN learning in silicon?

- People seem more excited about / satisfied with GPUs
- But, that could change