
Machine Learning

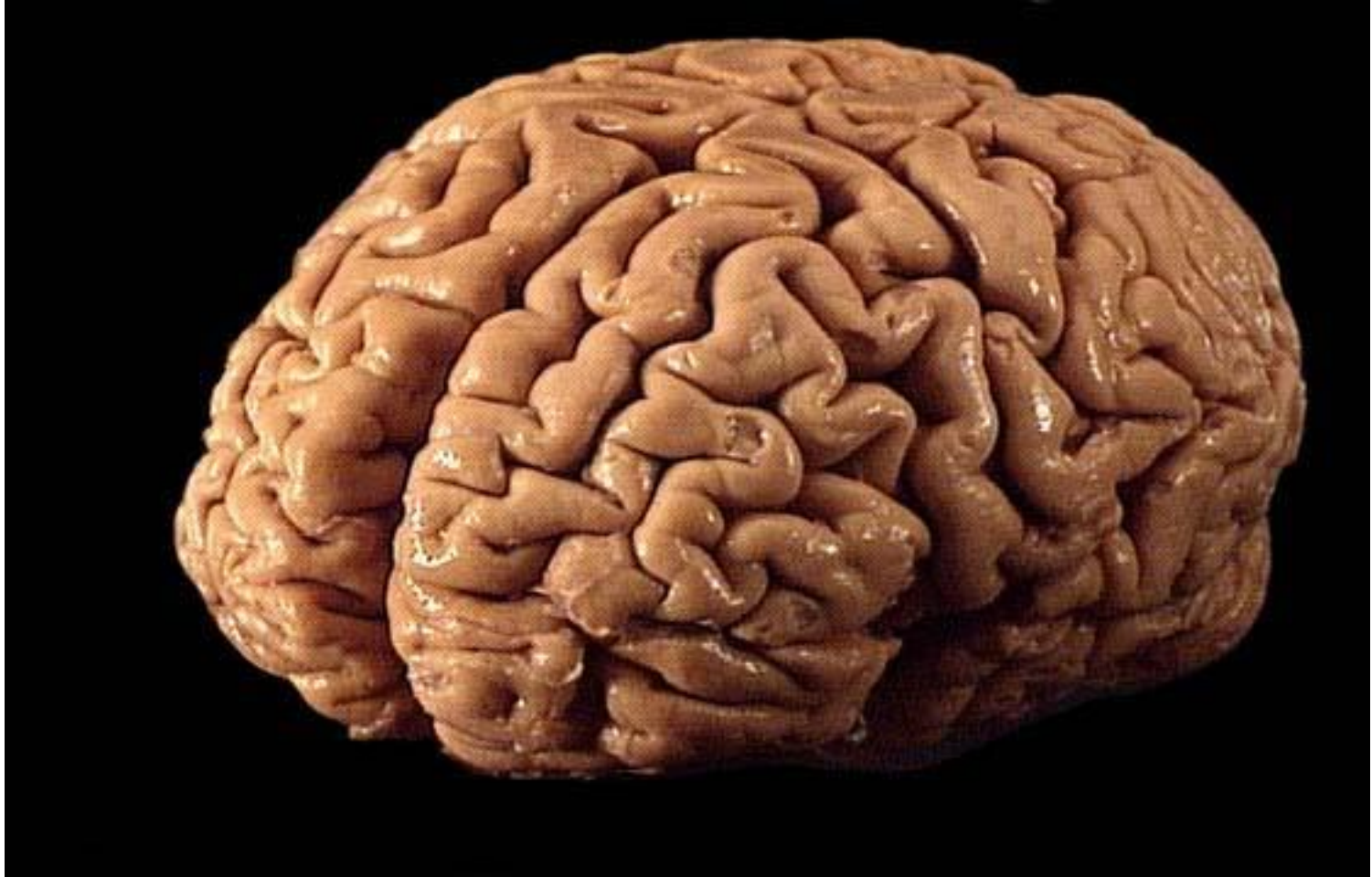
Neural Networks

(slides from Domingos, Pardo, others)

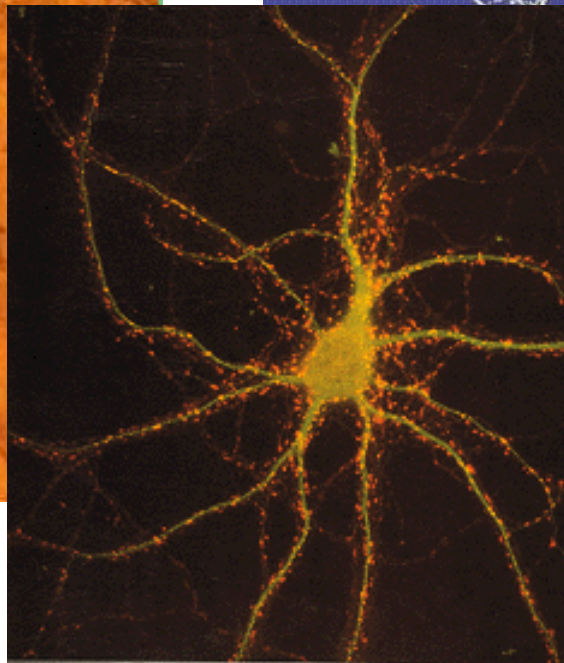
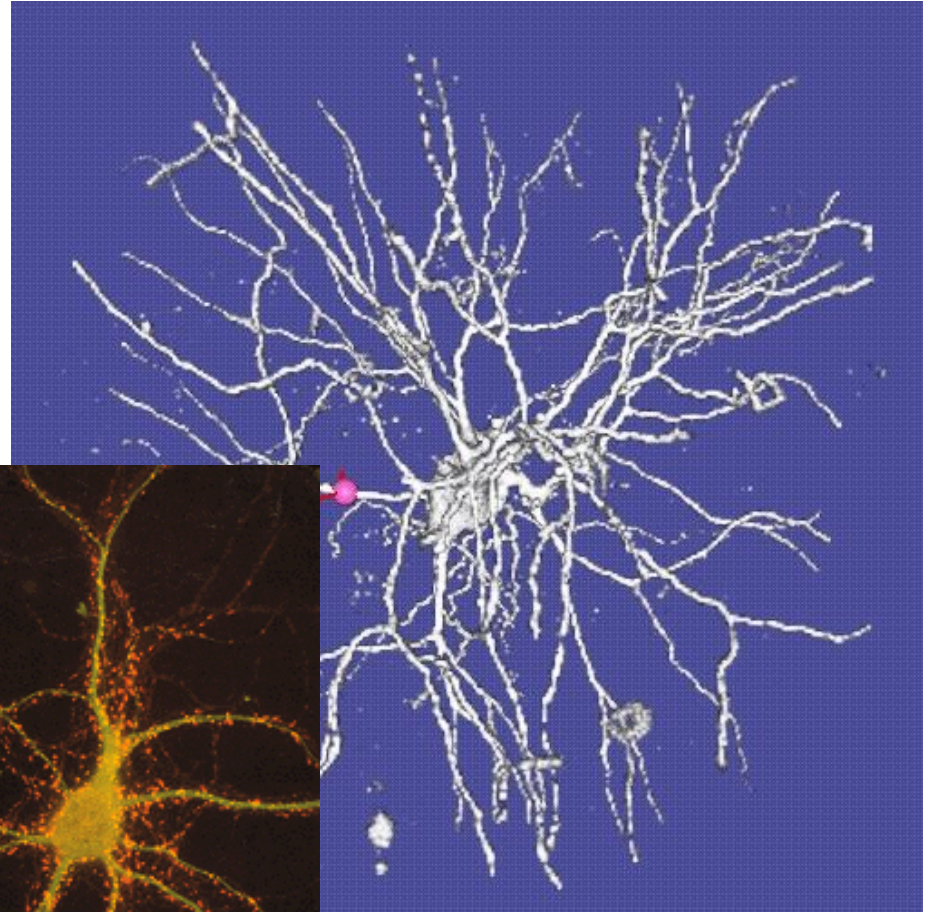
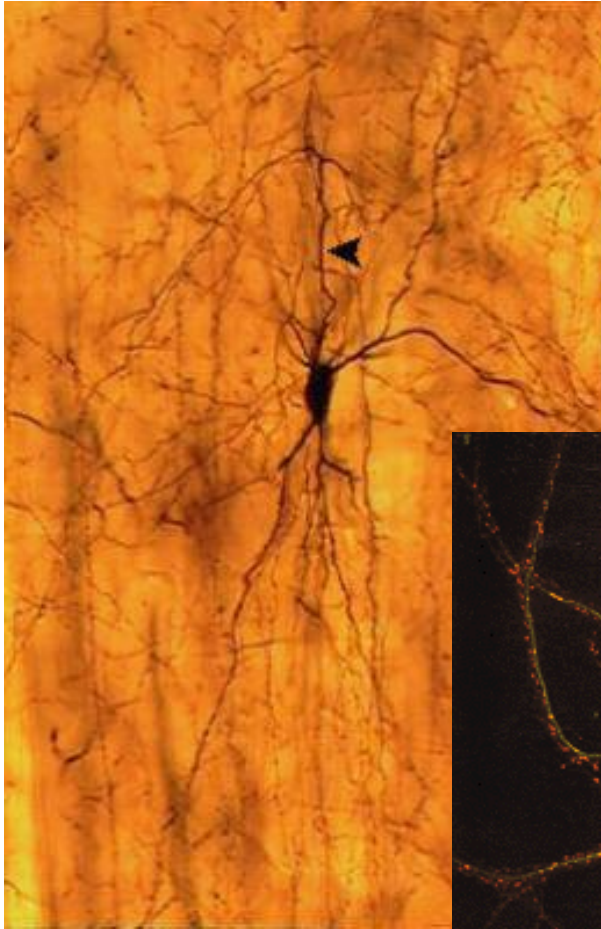
Reading

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

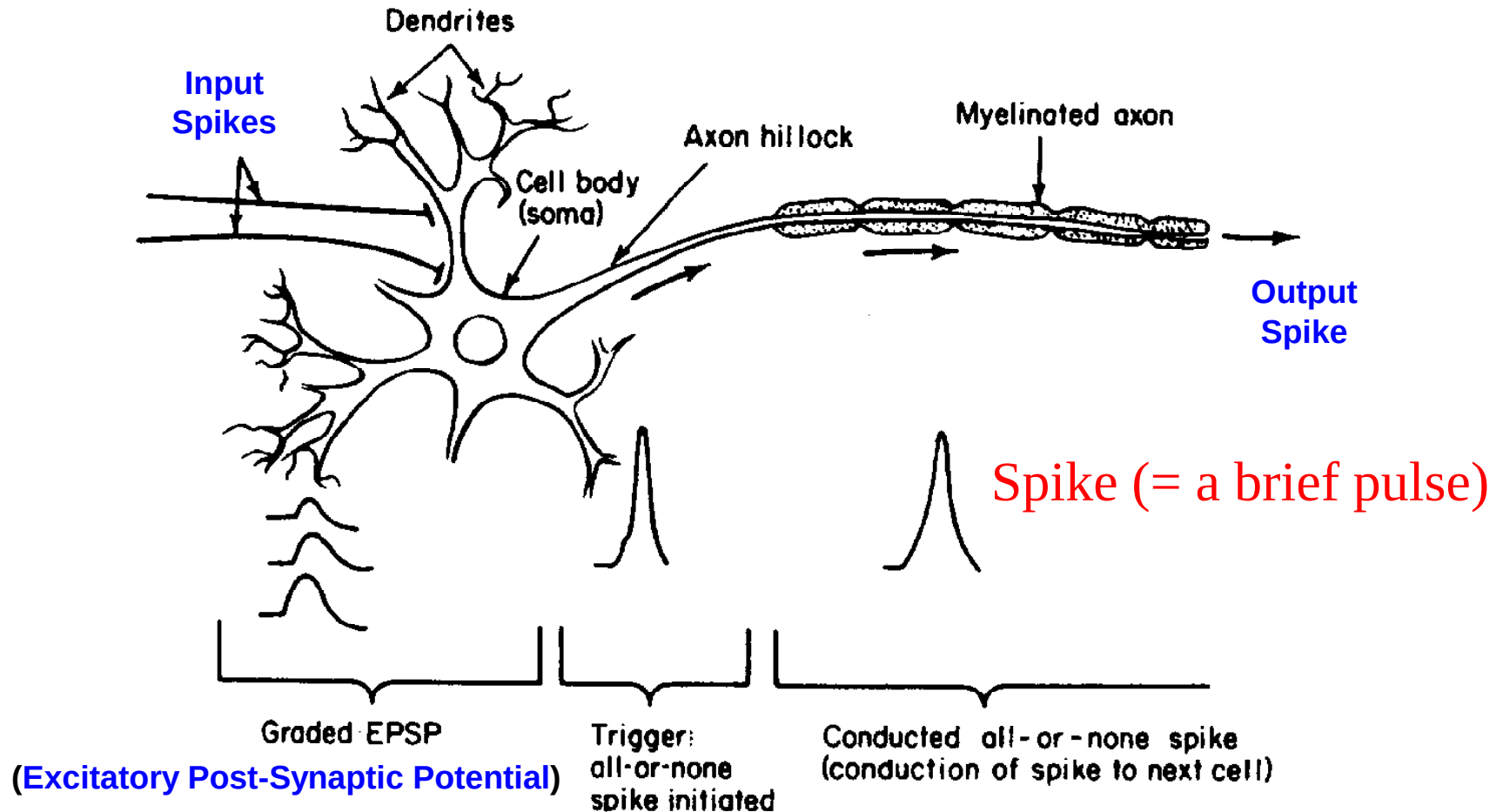
Human Brain



Neurons



Input-Output Transformation

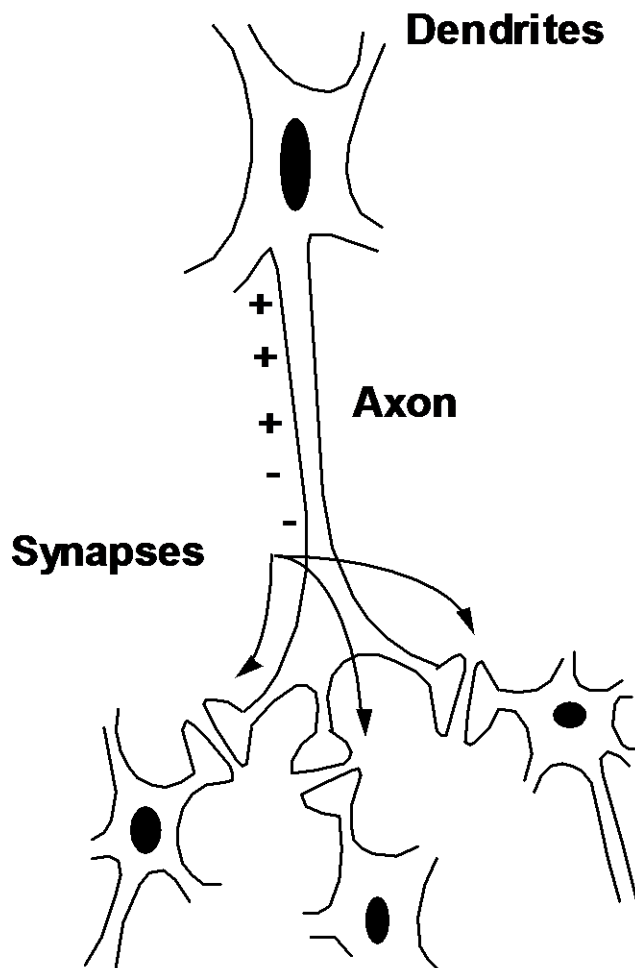


Human Learning

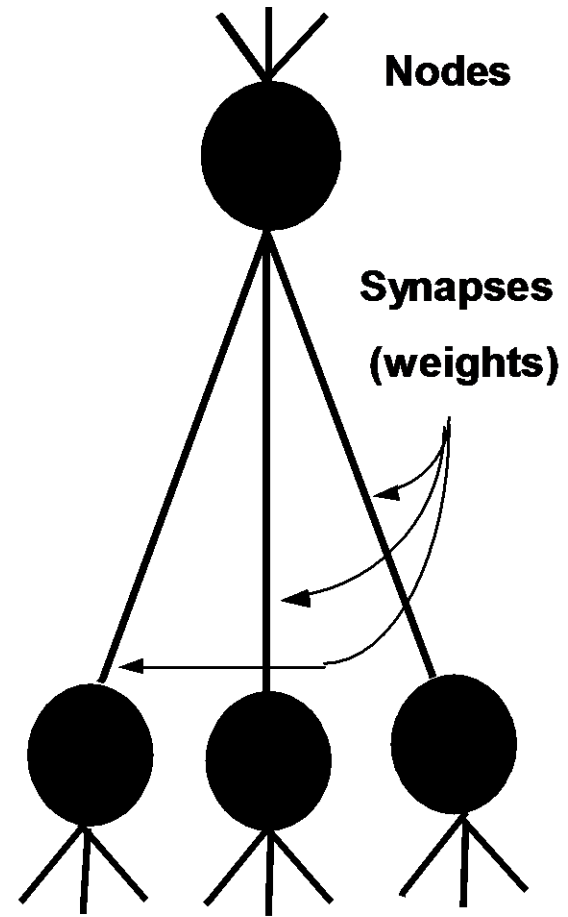
- Number of neurons: $\sim 10^{11}$
- Connections per neuron: $\sim 10^3$ to 10^5
- Neuron switching time: ~ 0.001 second
- Scene recognition time: ~ 0.1 second

100 inference steps doesn't seem much

Machine Learning Abstraction



Impulse



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

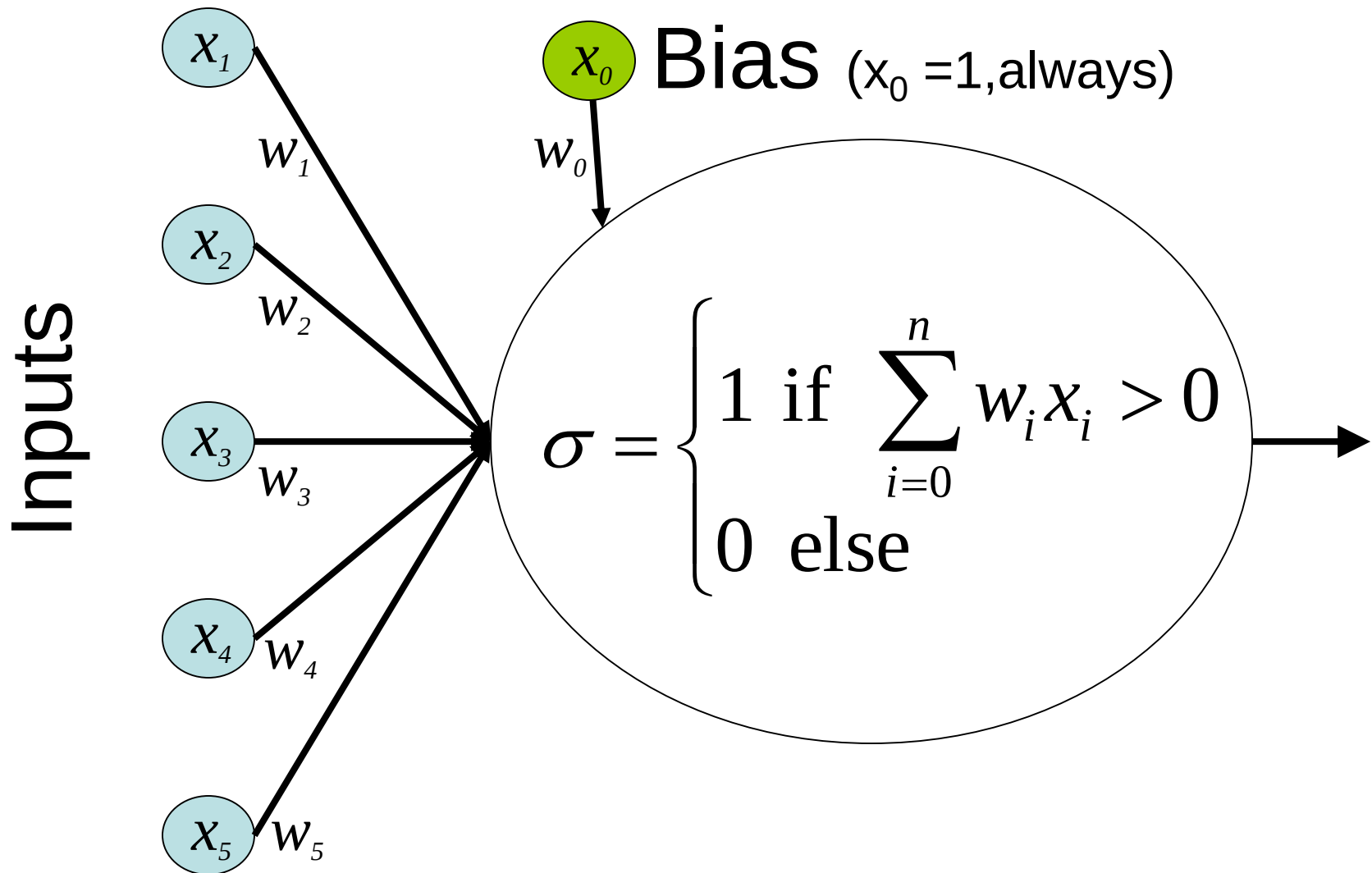
Perceptrons

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

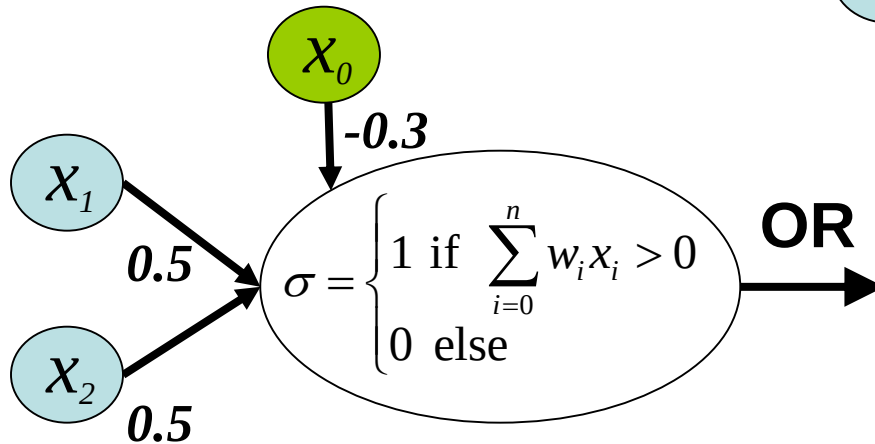
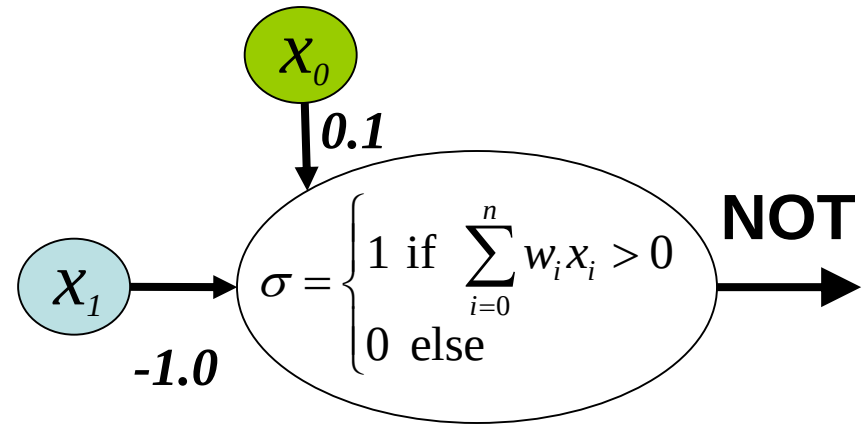
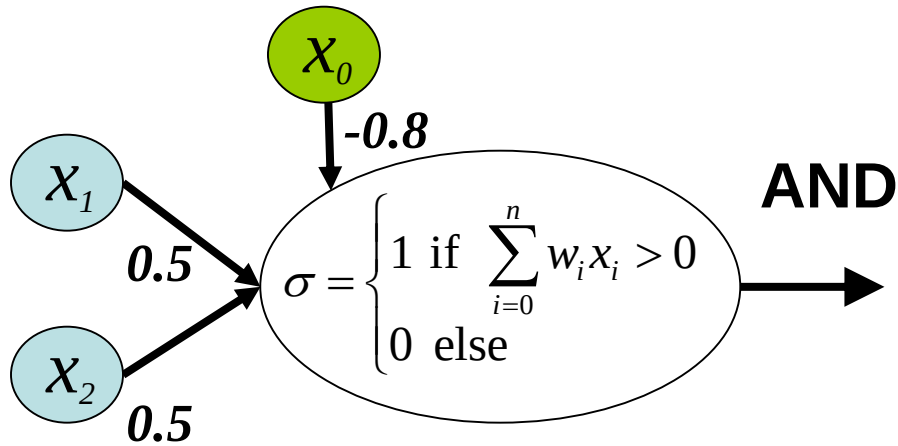
Perceptrons

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

A single perceptron

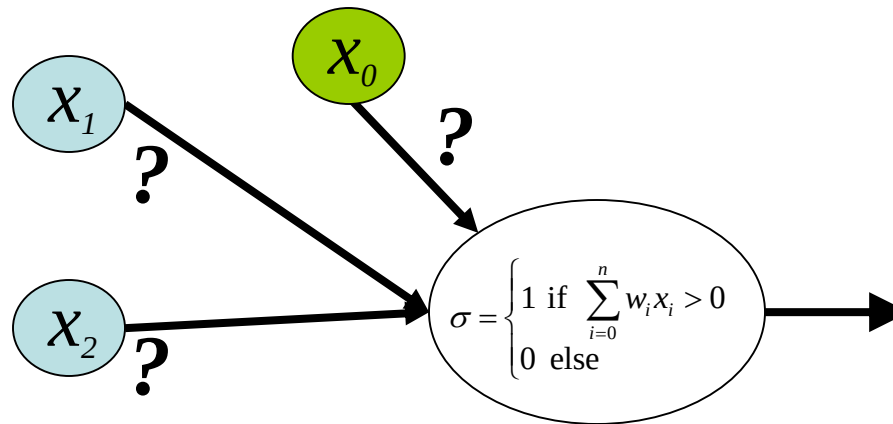


Logical Operators



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

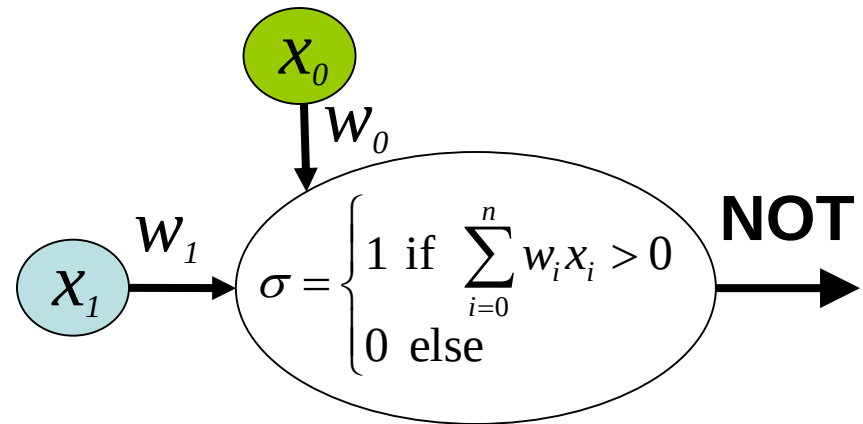
$$\Delta w_i = \eta(t - o)x_i$$

A diagram with four labels at the bottom: 'learning rate', 'target value', 'perceptron output', and 'input value'. Four arrows point from these labels to the corresponding parts of the equation $\Delta w_i = \eta(t - o)x_i$ above. The first arrow points from 'learning rate' to η . The second arrow points from 'target value' to t . The third arrow points from 'perceptron output' to o . The fourth arrow points from 'input value' to x_i .

learning rate target value perceptron output input value

Perception Training for NOT

Initialize:
 $w_0, w_1 = 0$

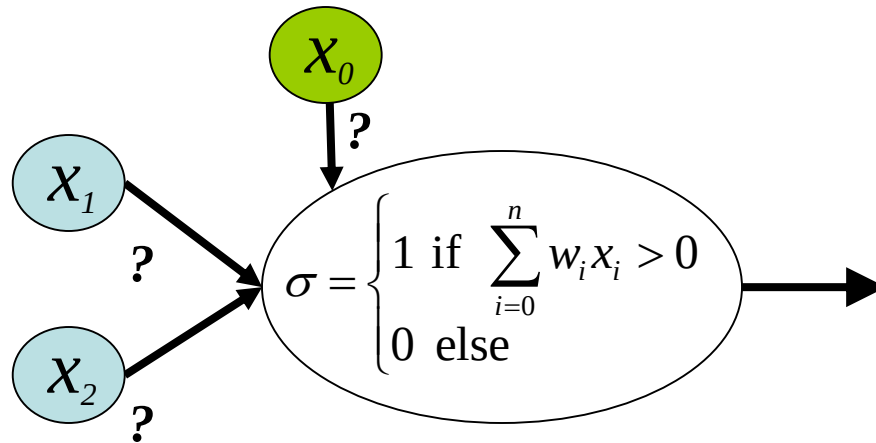


$$w_i \leftarrow w_i + \Delta w_i$$
$$\Delta w_i = \eta(t - o)x_i$$

| Work
Start

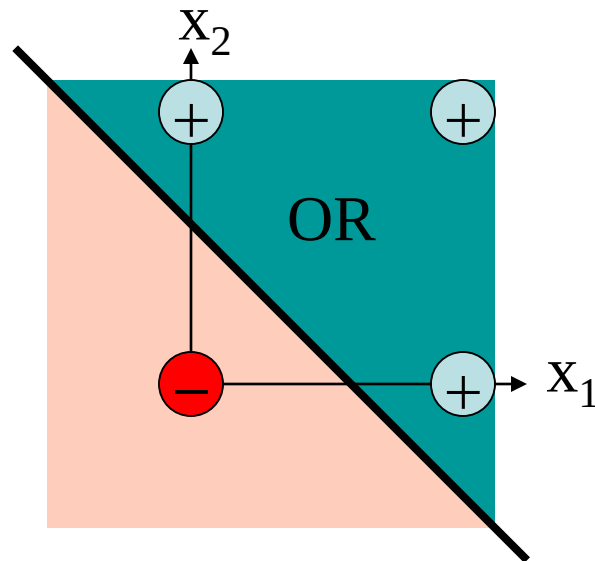
| End

What weights make XOR?

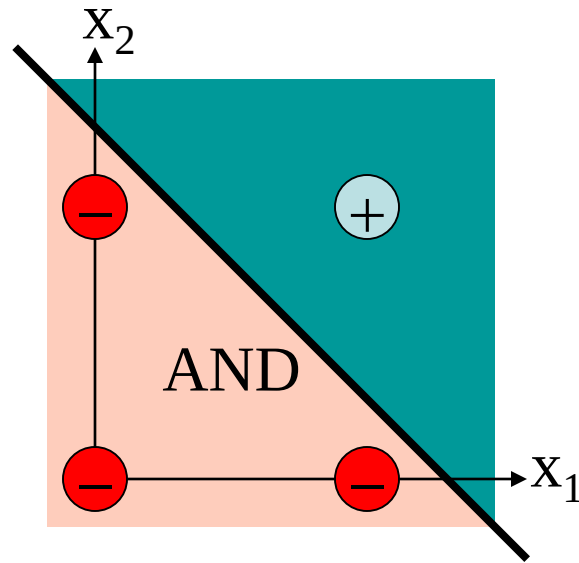


- No combination of weights works
- Perceptrons can only represent linearly separable functions

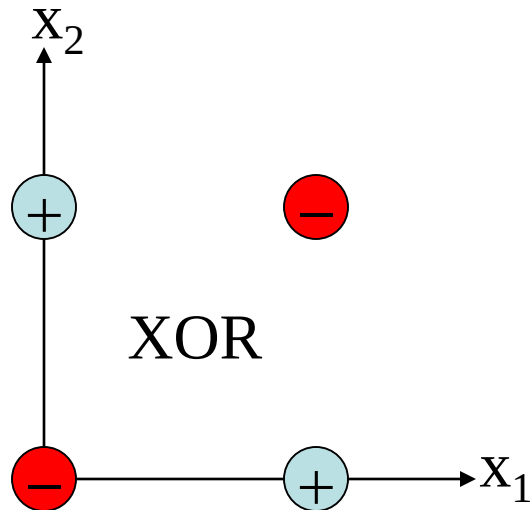
Linear Separability



Linear Separability



Linear Separability

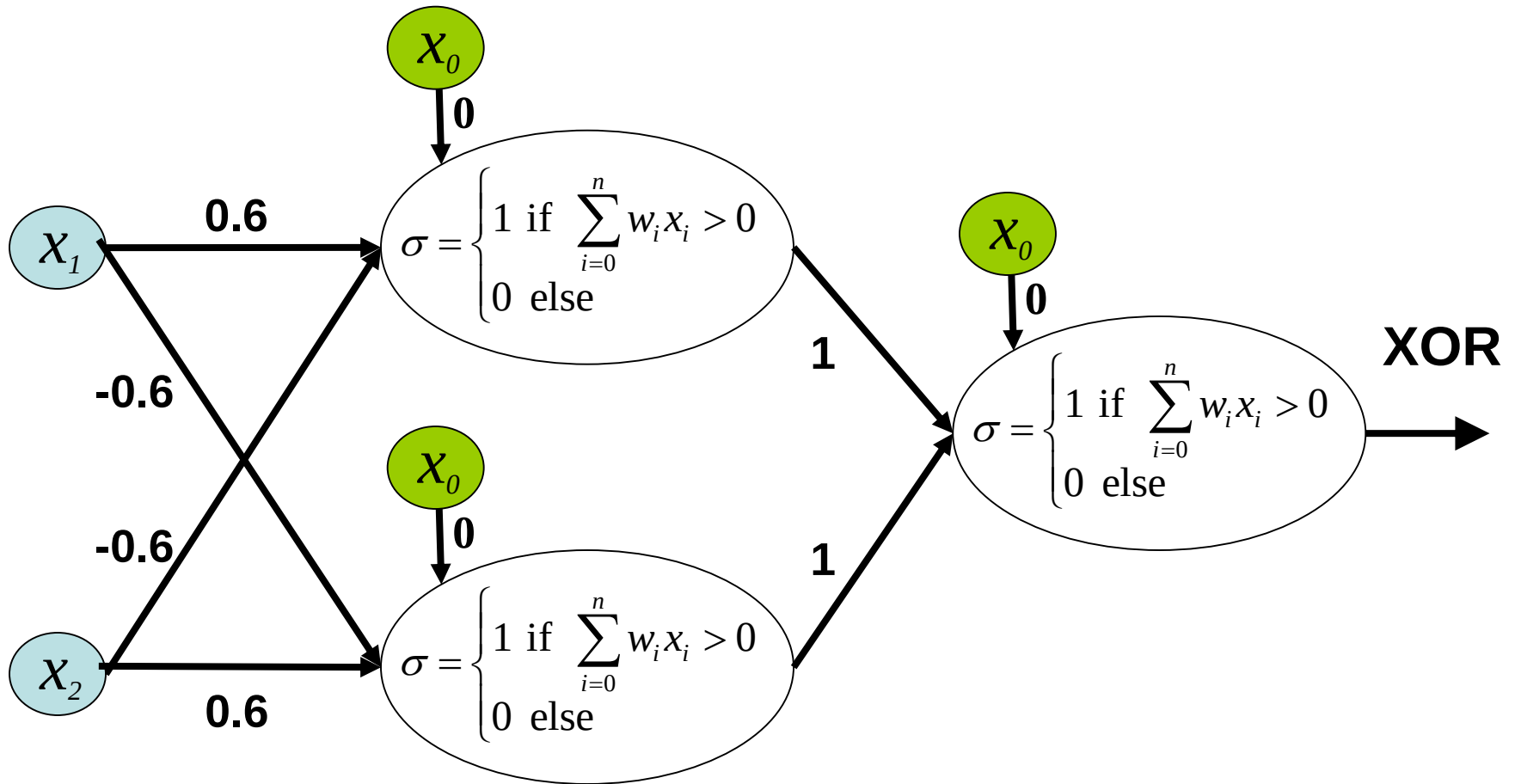


Perceptron Training Rule

- 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

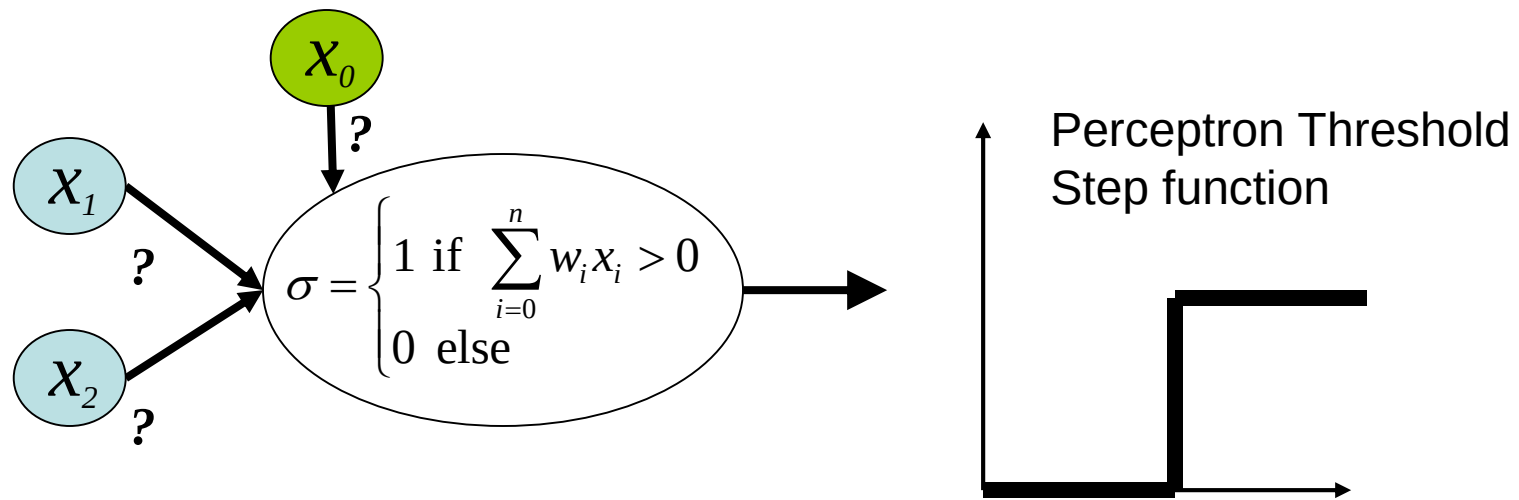
XOR



What's wrong with perceptrons?

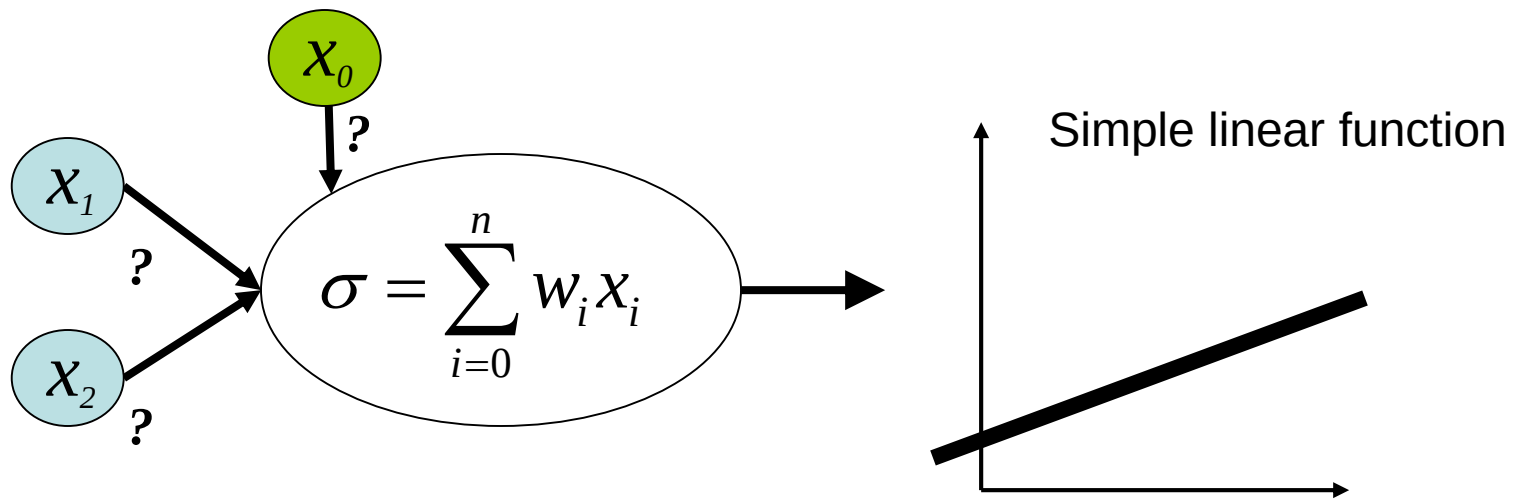
- You can always plug multiple perceptrons together to calculate any function.
- BUT...who decides what the weights are?
 - Assignment of error to parental inputs becomes a problem....

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:

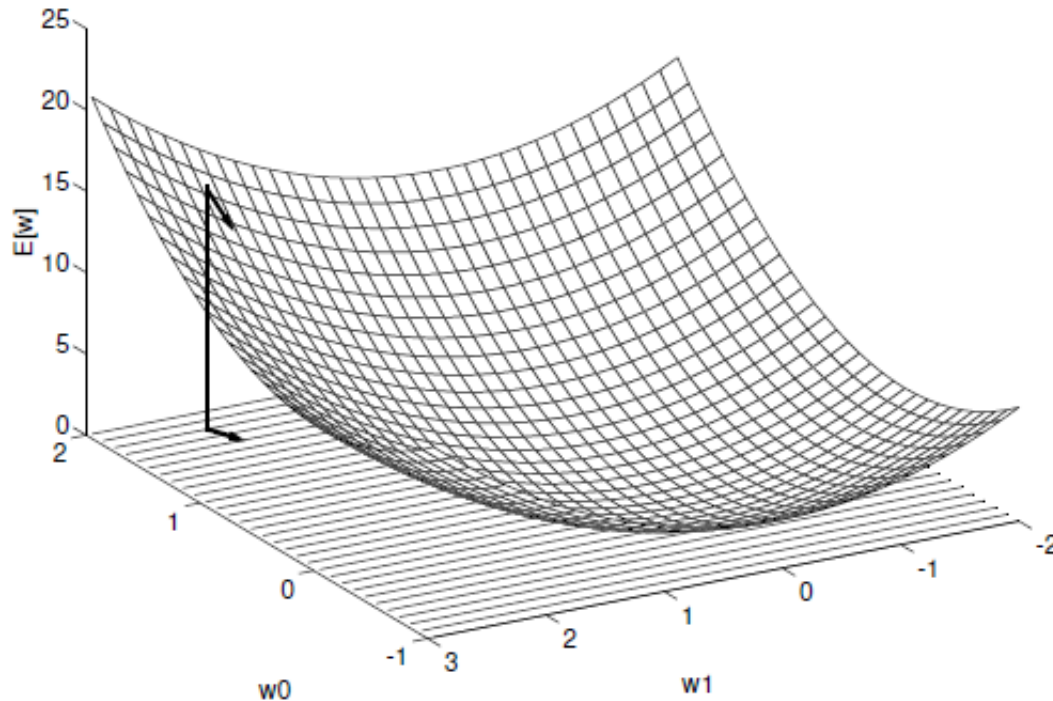
$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

data

target
value

linear unit
output

Gradient Descent



Gradient:

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

Training rule:

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

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

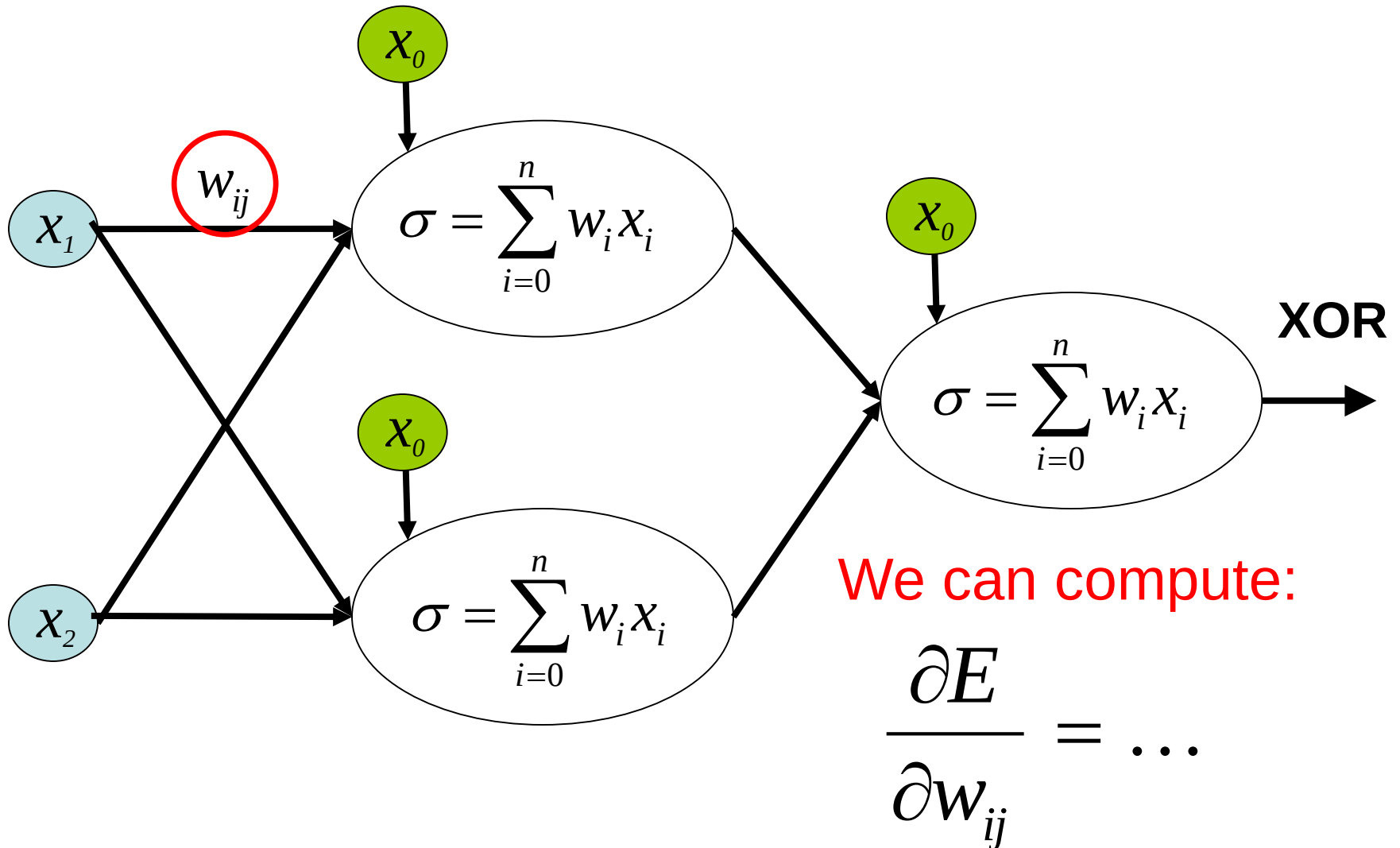
Gradient Descent Rule

$$\begin{aligned}\frac{\partial E}{\partial w_i} &\equiv \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \sum_{d \in D} (t_d - o_d)(-x_{i,d})\end{aligned}$$

Update Rule:

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

Gradient Descent for Multiple Layers



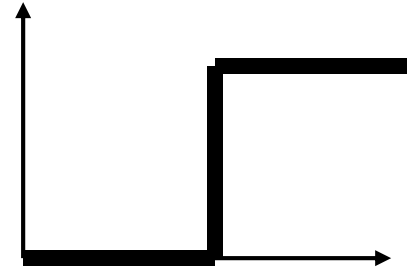
Gradient Descent vs. Perceptrons

- 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...

A compromise function

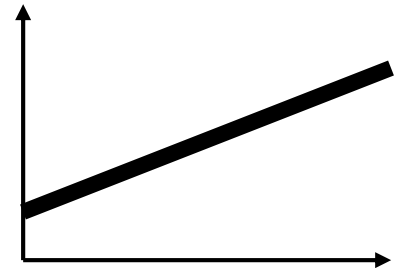
- Perceptron

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



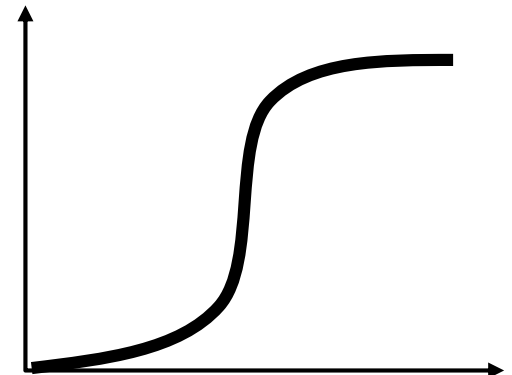
- Linear

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



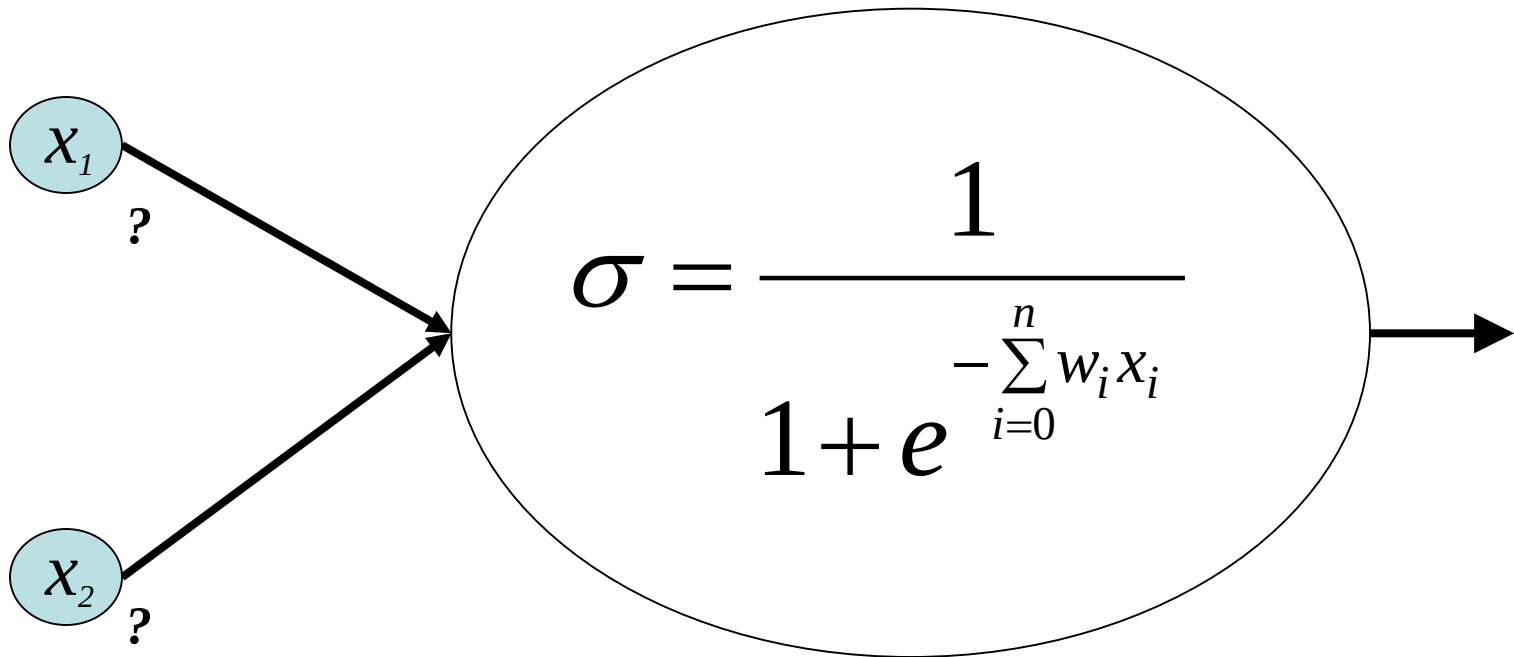
- Sigmoid (Logistic)

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

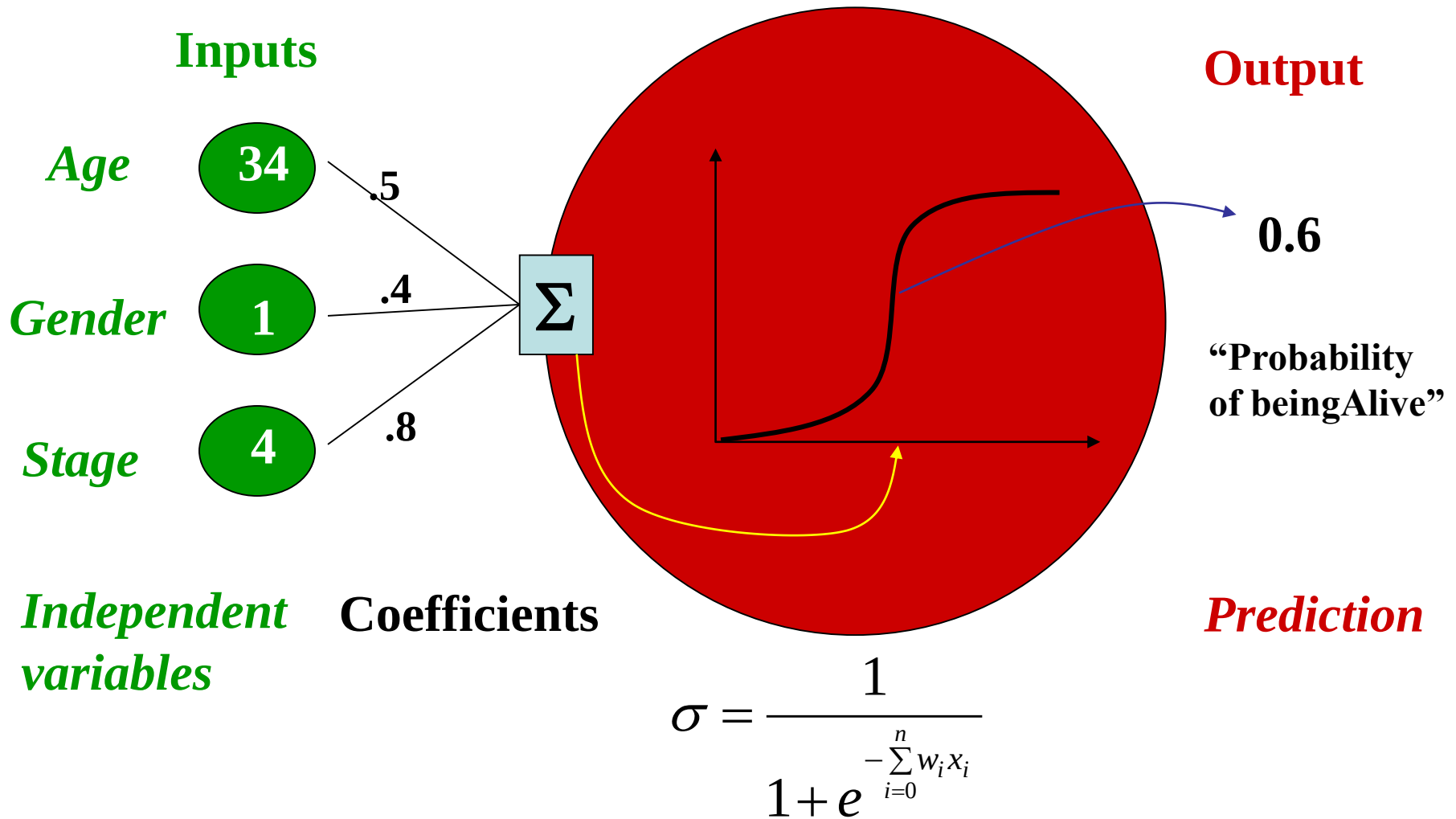


The sigmoid (logistic) unit

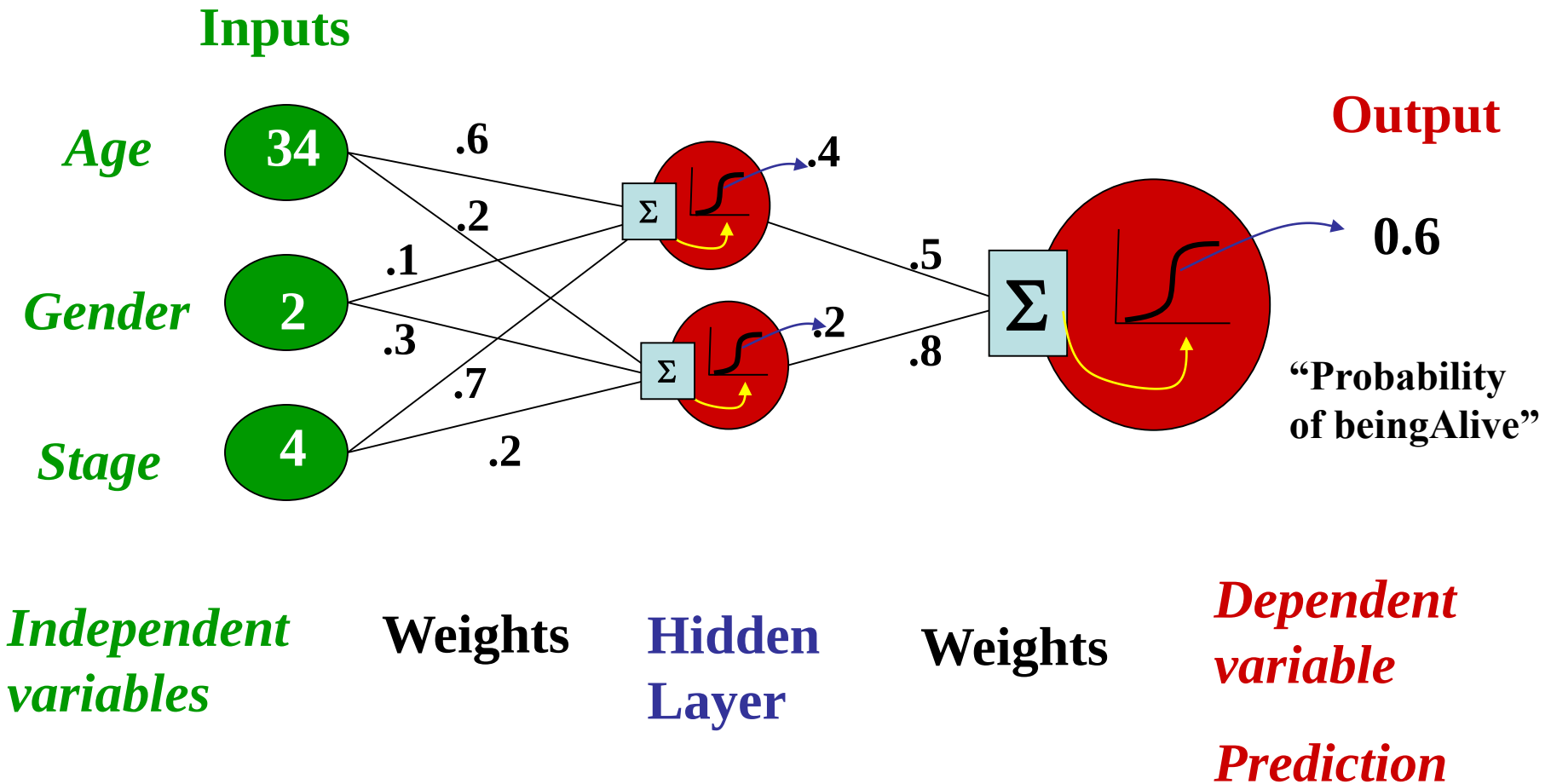
- Has differentiable function
 - Allows gradient descent
- Can be used to learn non-linear functions



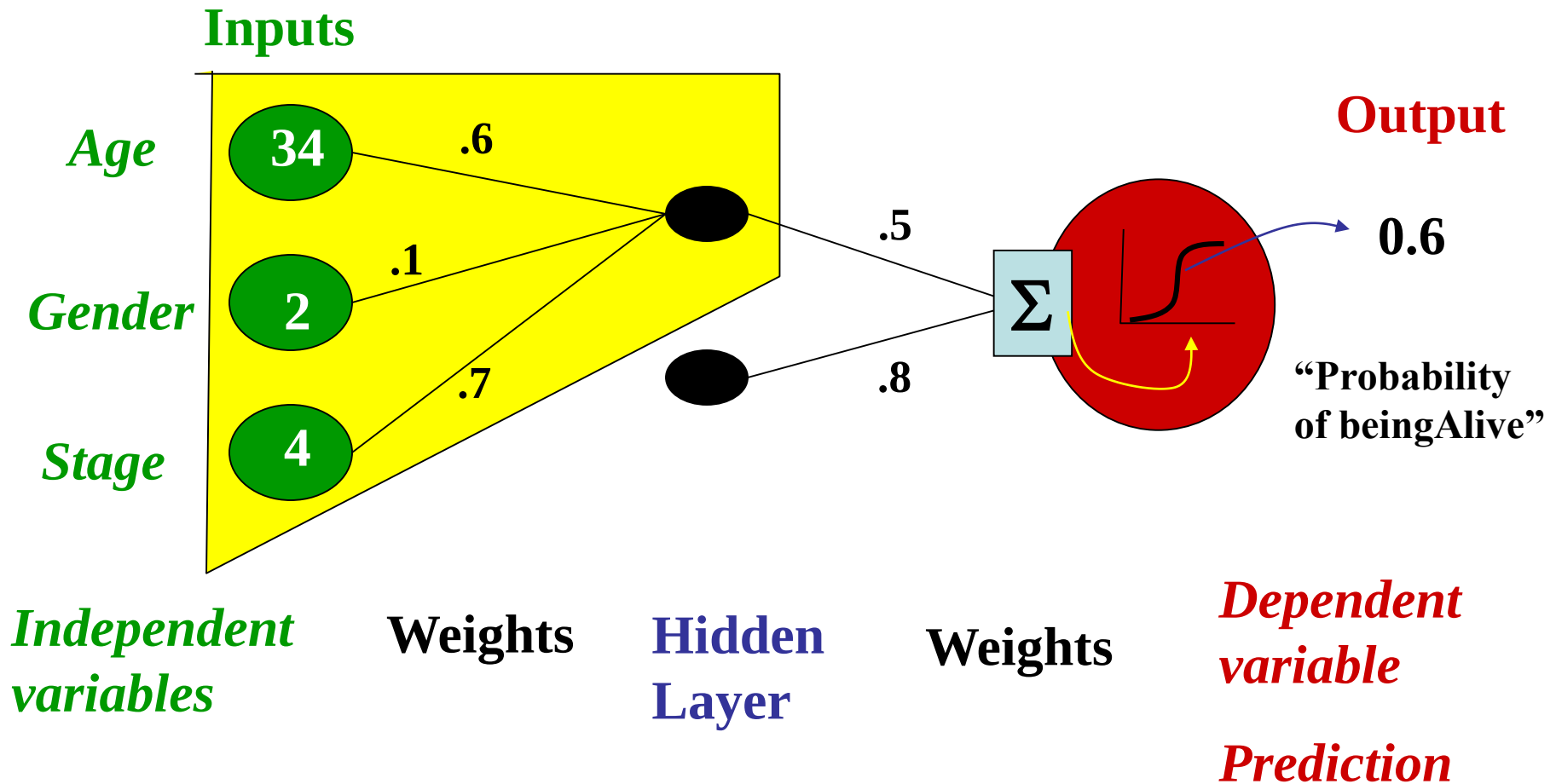
Logistic function



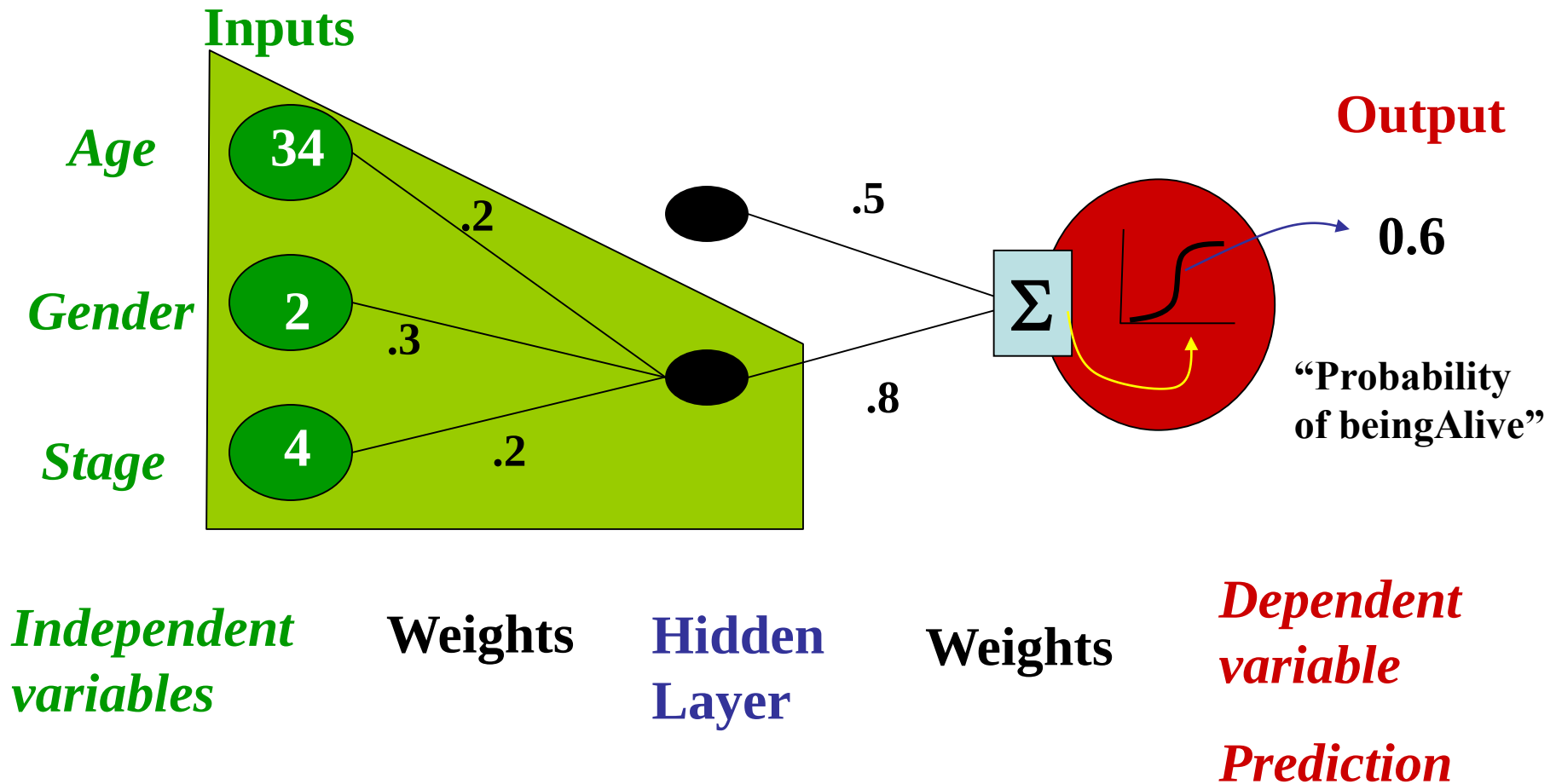
Neural Network Model



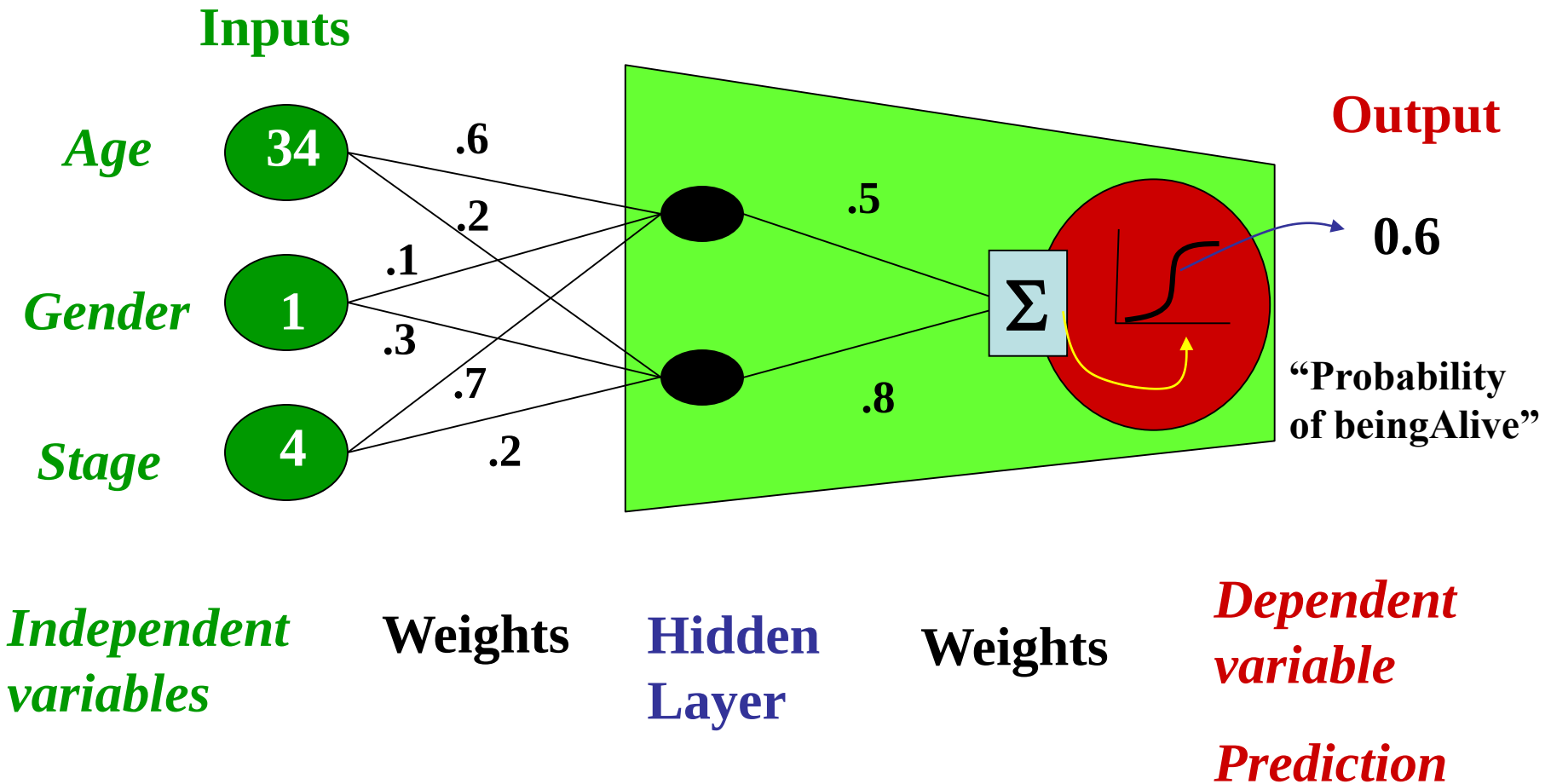
Getting an answer from a NN



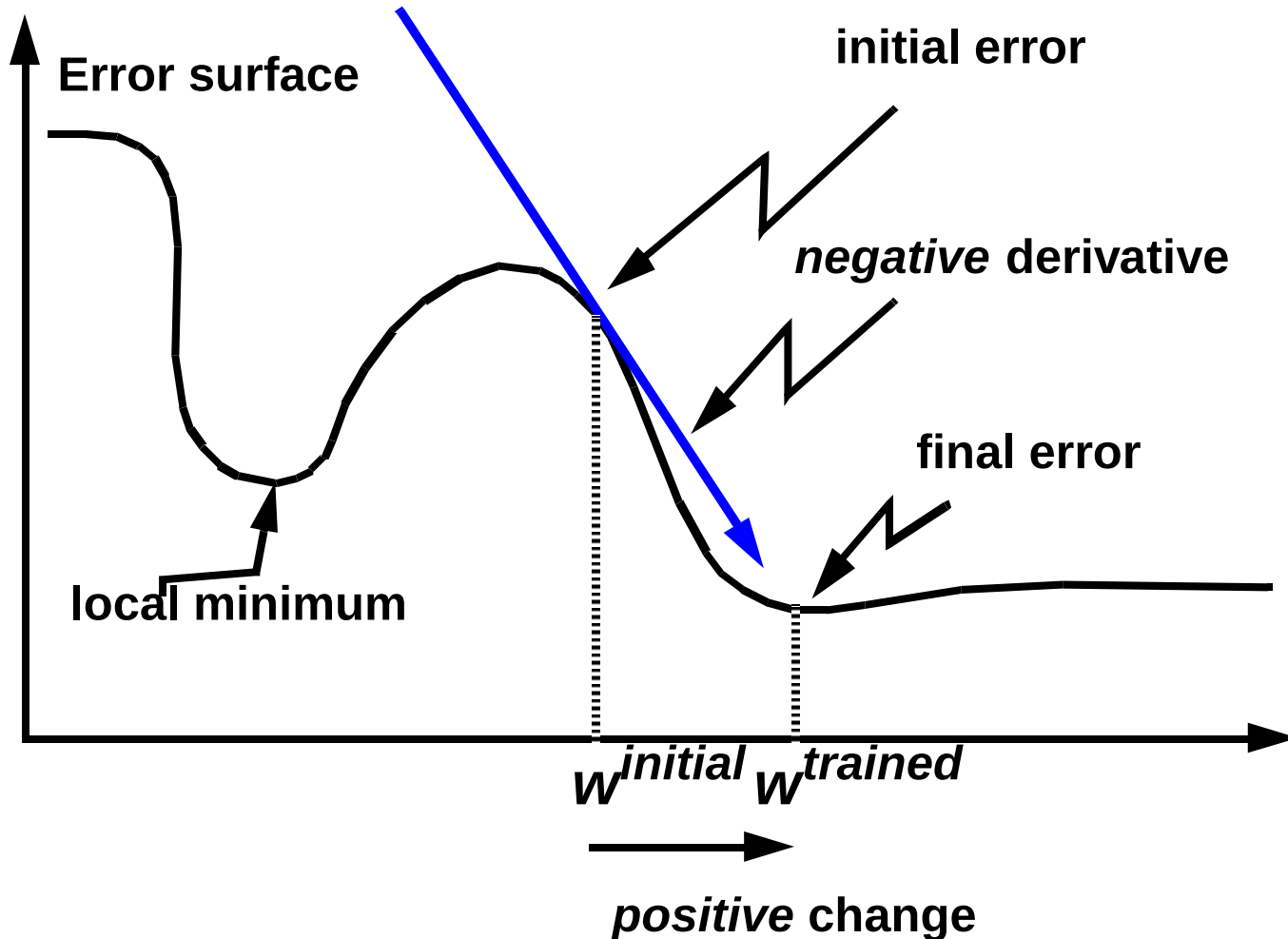
Getting an answer from a NN



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, $\langle \vec{x}, \vec{t} \rangle$

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

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

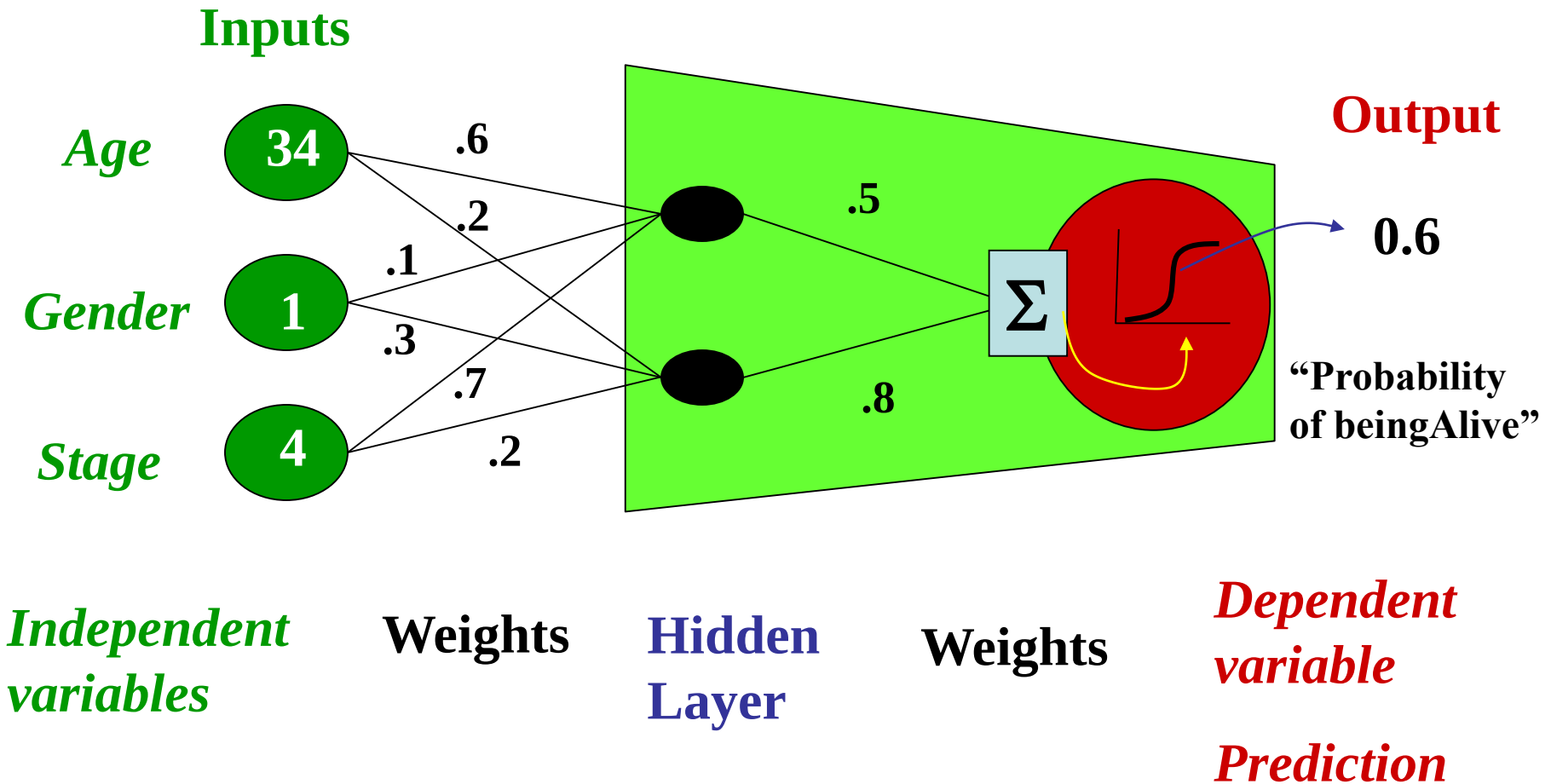
3. For each hidden unit h , calculate its error term δ_h

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

4. Update each network weight w_{ji}

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

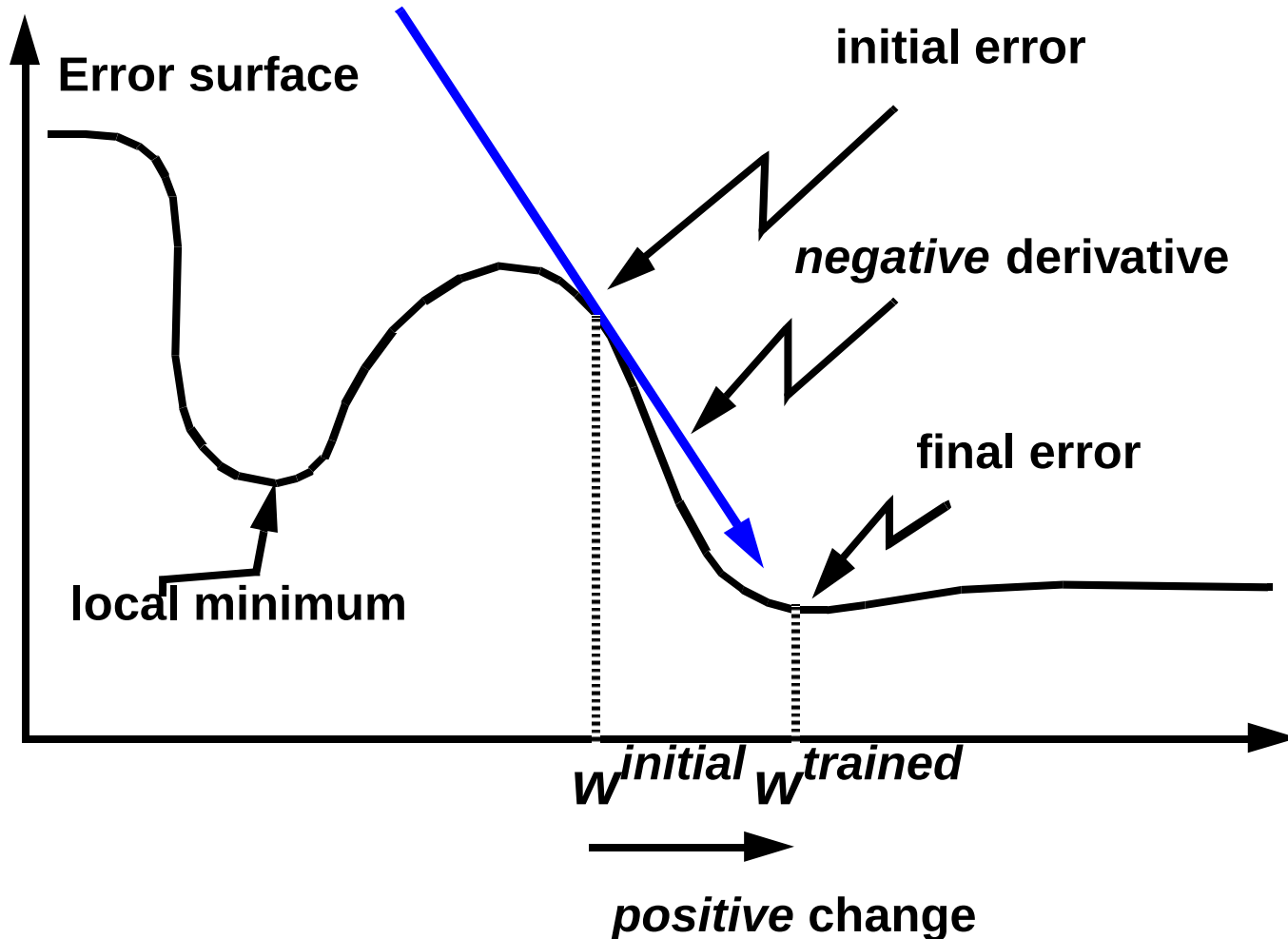
Learning Weights



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

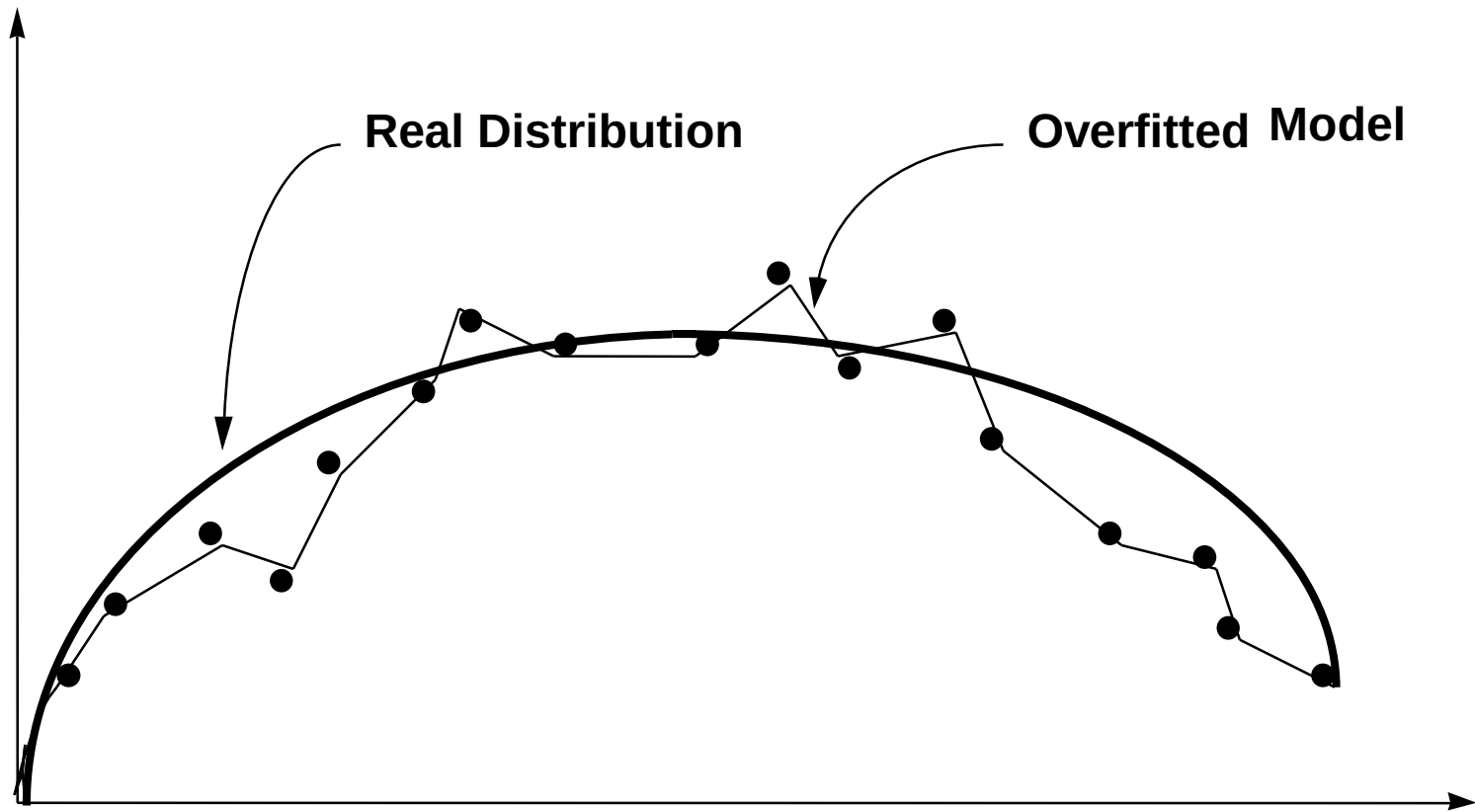
Minimizing the Error



Expressive Power of ANNs

- Universal Function Approximator:
 - Given enough hidden units, can approximate *any* continuous function f
- Need 2+ hidden units to learn XOR
- Why not use millions of hidden units?
 - Efficiency (training is slow)
 - Overfitting

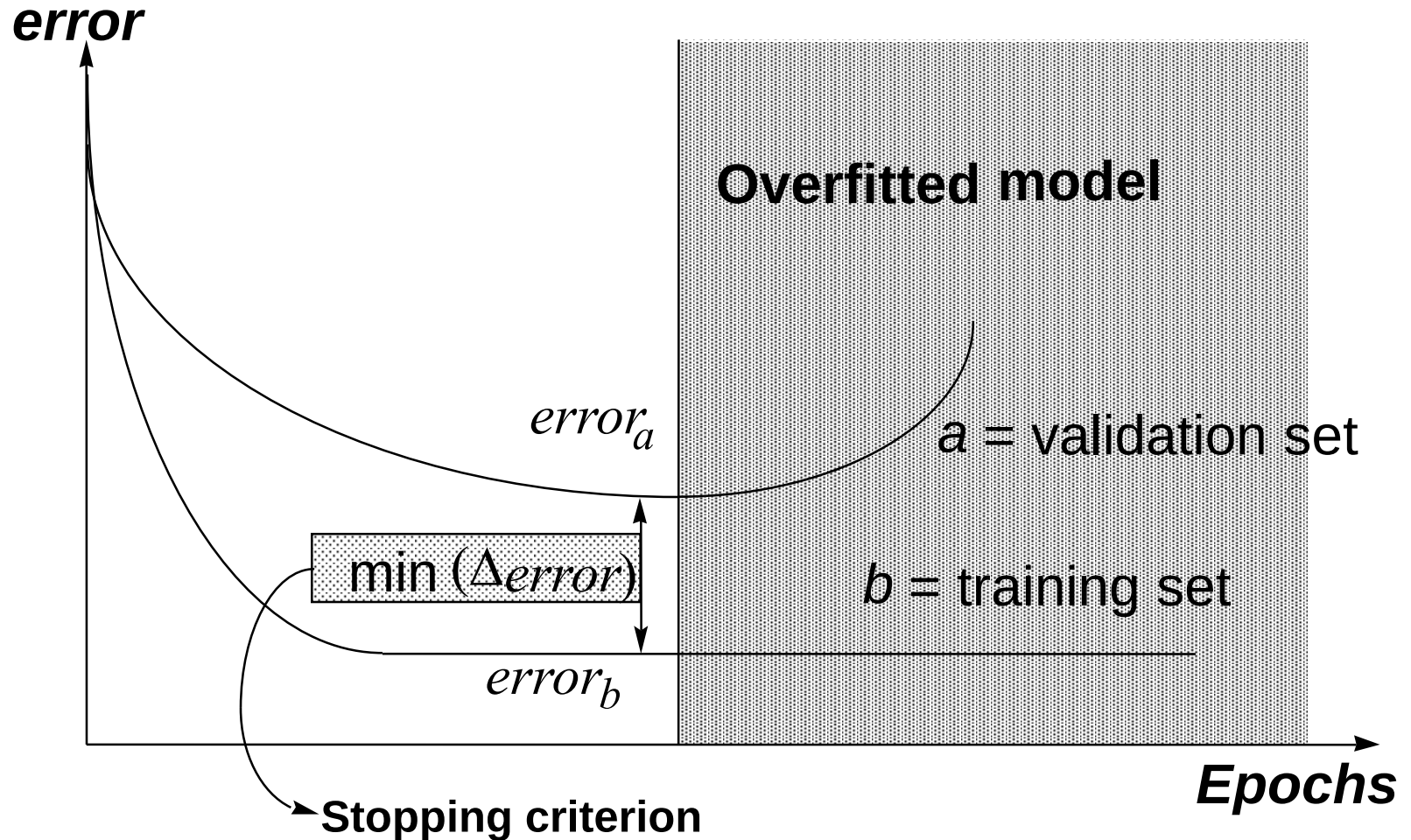
Overfitting



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?

Advanced Topics in Neural Nets

- Batch Move vs. stochastic
- Auto-Encoders
- Deep Belief Nets (briefly)
- Neural Networks on Silicon
- Neural Network language models

Stochastic vs. Batch Mode

Stochastic Gradient Descent

Do until satisfied

- For each training example d in D
 1. Compute the gradient $\nabla E_d[\vec{w}]$
 2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

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: $\Delta \vec{w}_D = \sum_{d \in D} \Delta \vec{w}_d$
- Then setting $\vec{w} \leftarrow \vec{w} + \Delta \vec{w}_D$

Advanced Topics in Neural Nets

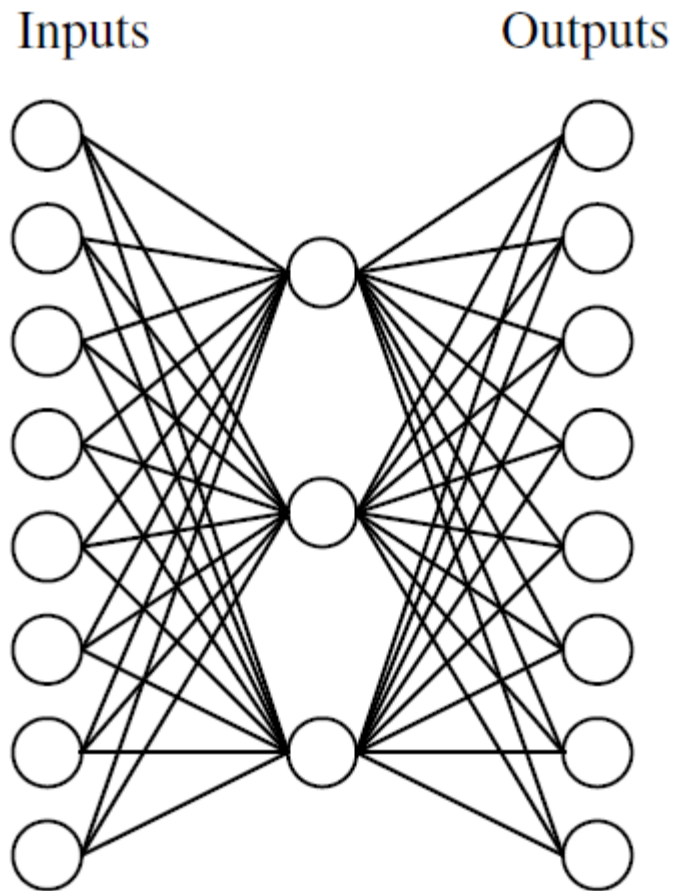
- Batch Move vs. incremental
- **Auto-Encoders**
- Deep Belief Nets (briefly)
- Neural Networks on Silicon
- Neural Network language models

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
 - $\langle x_1, \dots, x_n \rangle \rightarrow \langle x'_1, \dots, x'_m \rangle$ where $m < n$
 - Useful for visualization, problem simplification, data compression, etc.

Dimensionality Reduction

Model:



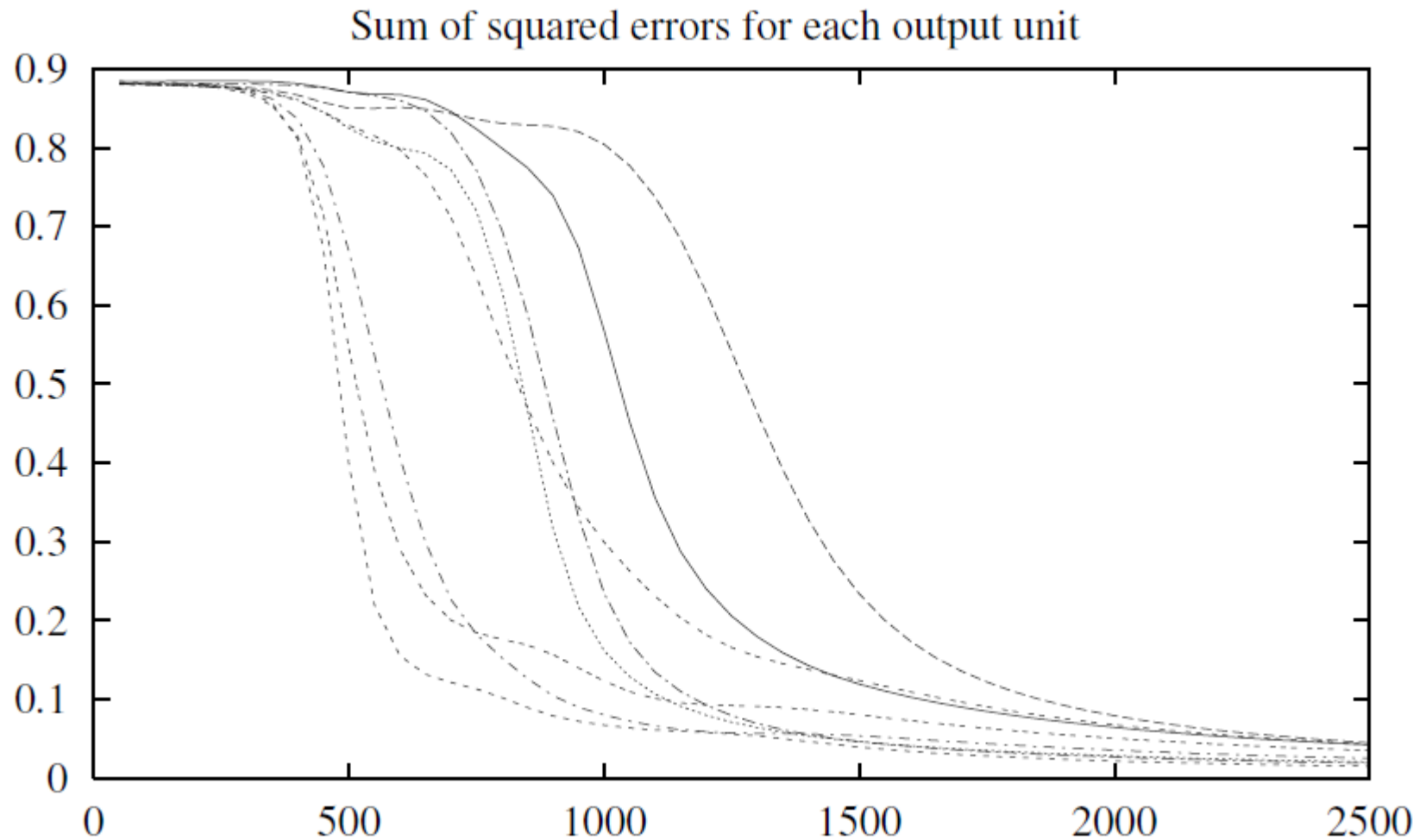
Function to learn:

Input	Output
10000000	→ 10000000
01000000	→ 01000000
00100000	→ 00100000
00010000	→ 00010000
00001000	→ 00001000
00000100	→ 00000100
00000010	→ 00000010
00000001	→ 00000001

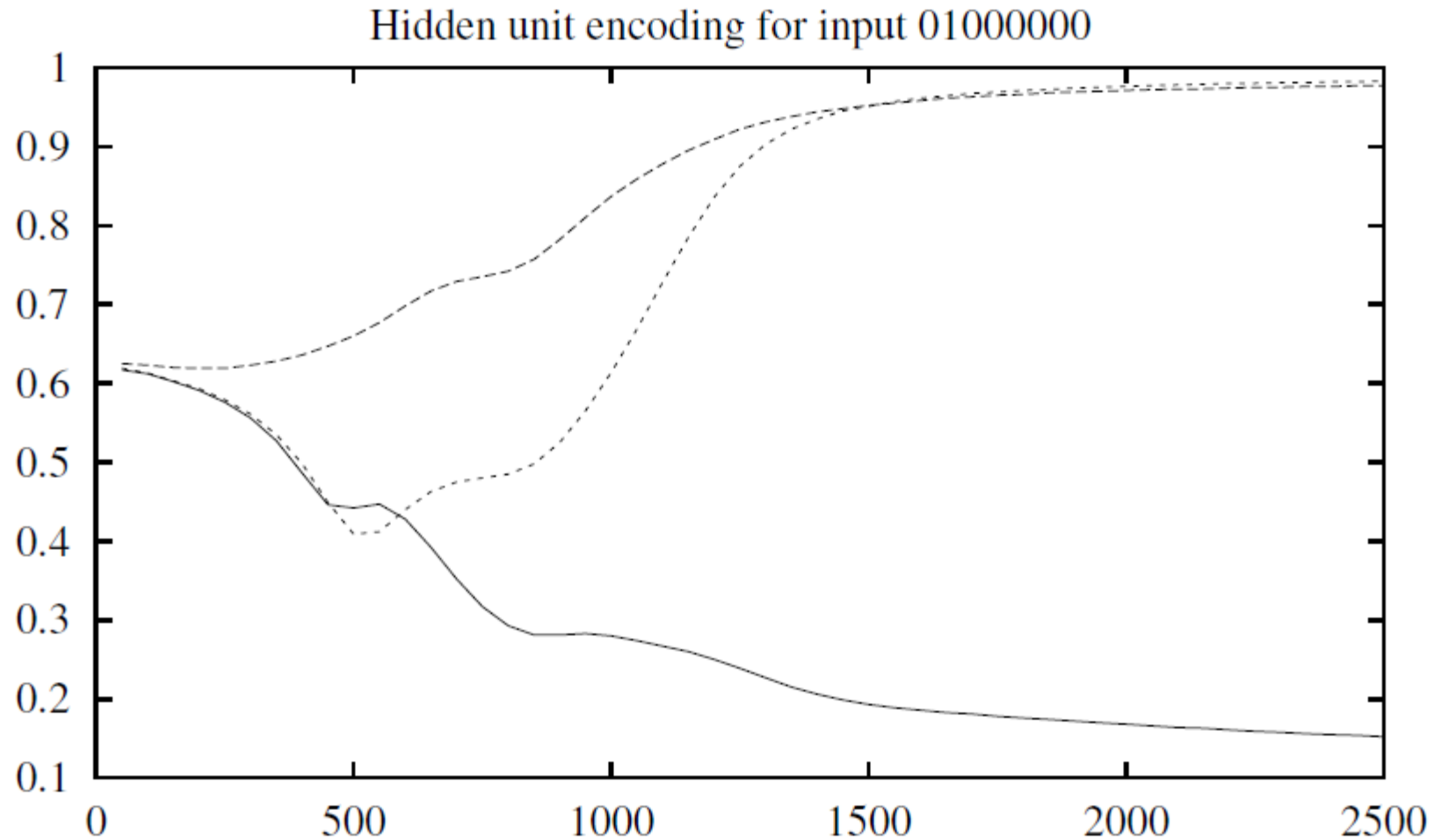
Dimensionality Reduction: Example

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

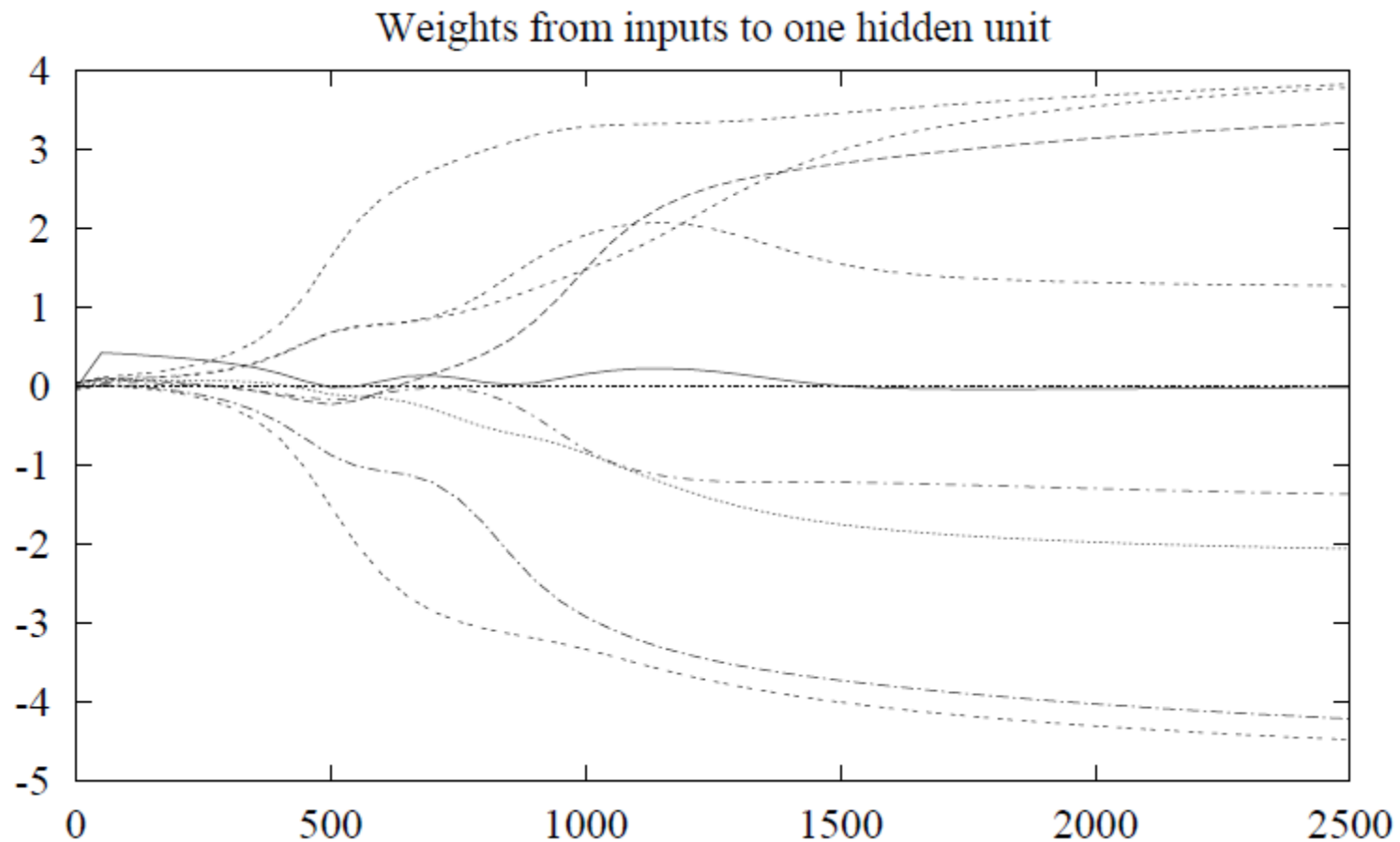
Dimensionality Reduction: Example



Dimensionality Reduction: Example



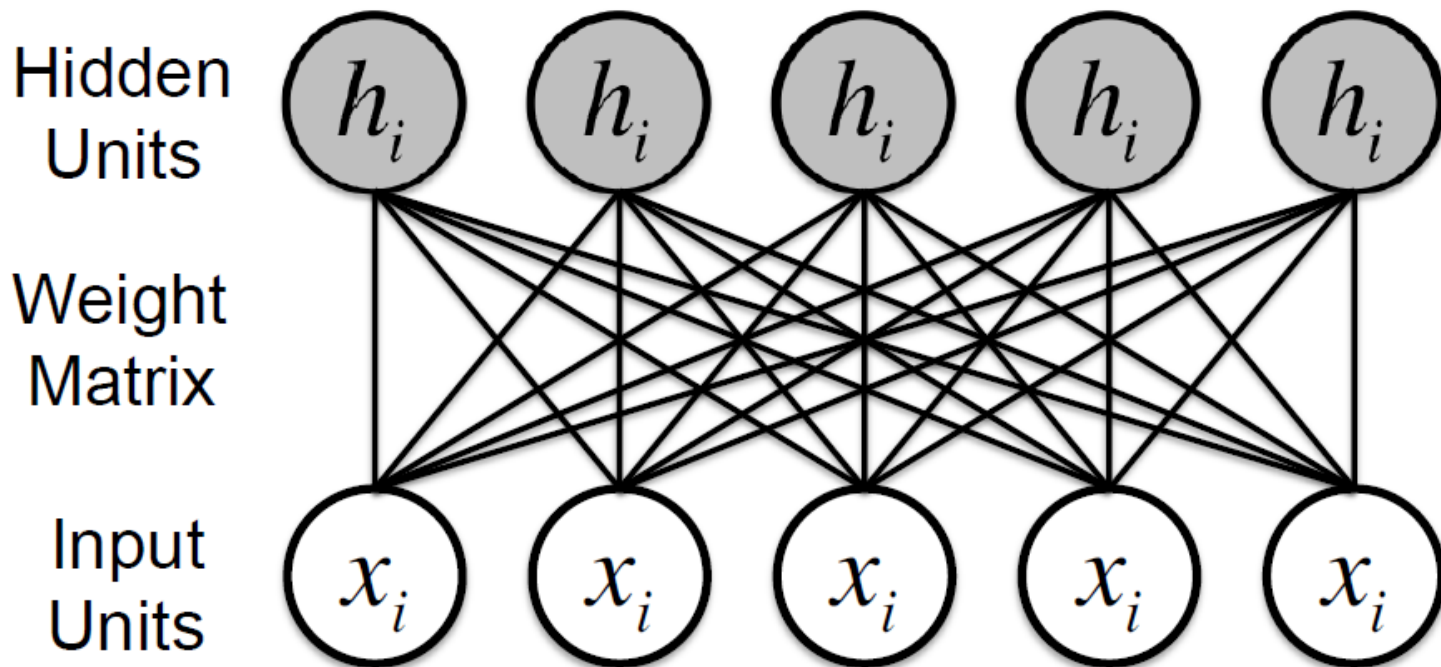
Dimensionality Reduction: Example



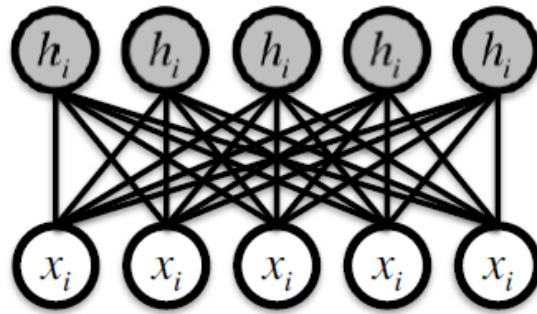
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-encoders
- **Deep Belief Nets (briefly)**
- Neural Networks on Silicon
- Neural Network language models

Restricted Boltzman Machine



2 layers (hidden & input) of Boolean nodes
Nodes only connected to the other layer



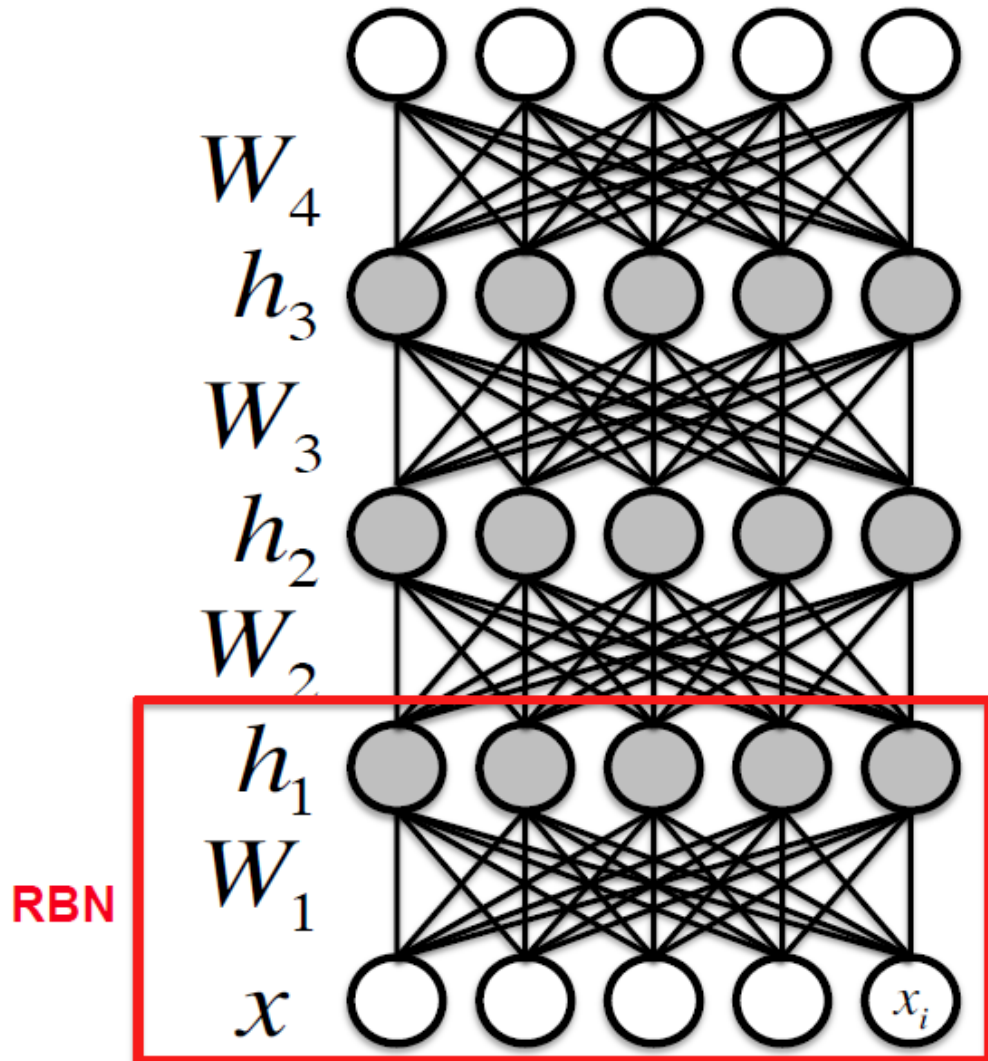
- Setting the hidden nodes to a vector of values updates the visible nodes...and vice versa

Auto-encoders vs. RBMs?

- Similar
- Auto-encoder (AE) goal is to reconstruct input in two steps, input->hidden->output
- RBM defines a probability distribution over $P(\mathbf{x})$
 - Goal is to assign high likelihood to the observed training examples
 - Determining likelihood of a given $\mathbf{x}$ actually requires summing over all possible settings of hidden nodes, rather than just computing a single activation as in AE
 - Take EECS 395/495 Probabilistic Graphical Models to learn more

Deep Belief Nets

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)

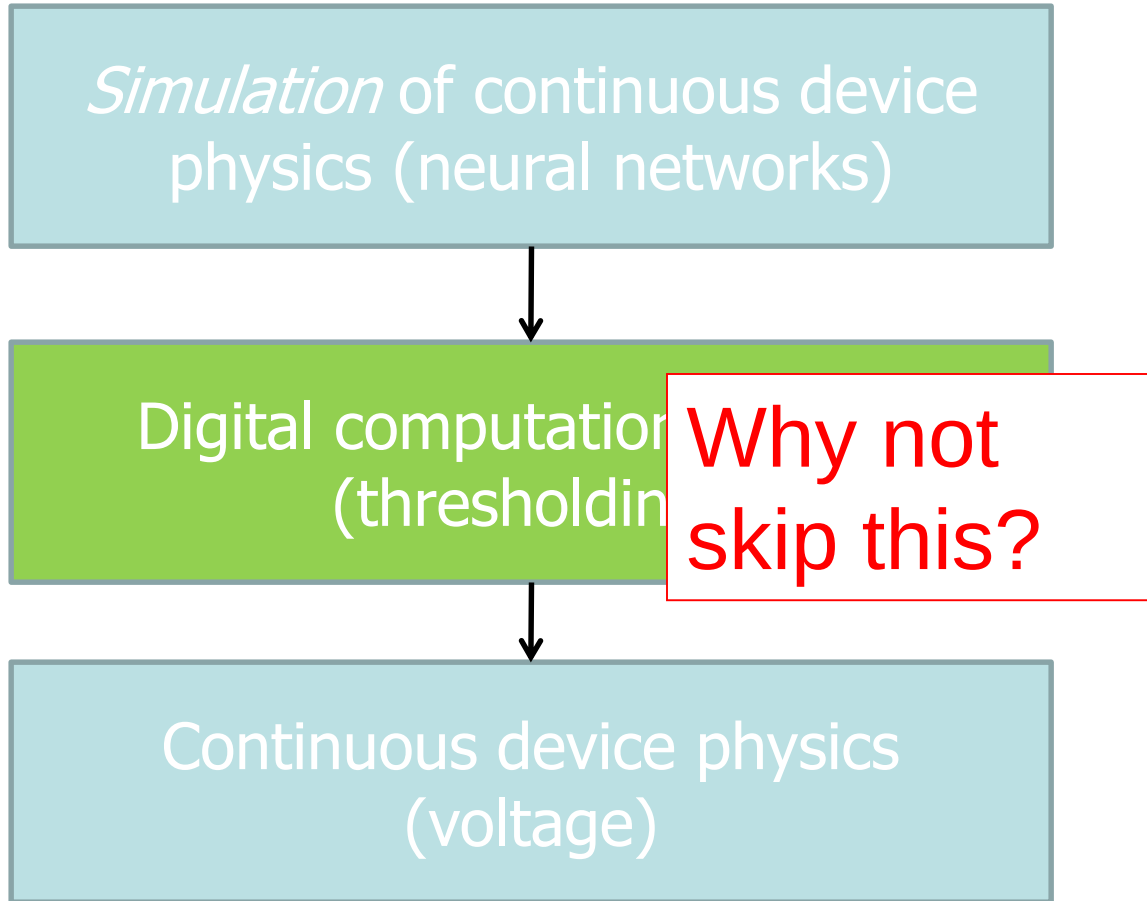


Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-Encoders
- Deep Belief Nets (briefly)
- **Neural Networks on Silicon**
- Neural Network language models

Neural Networks on Silicon

- Currently:

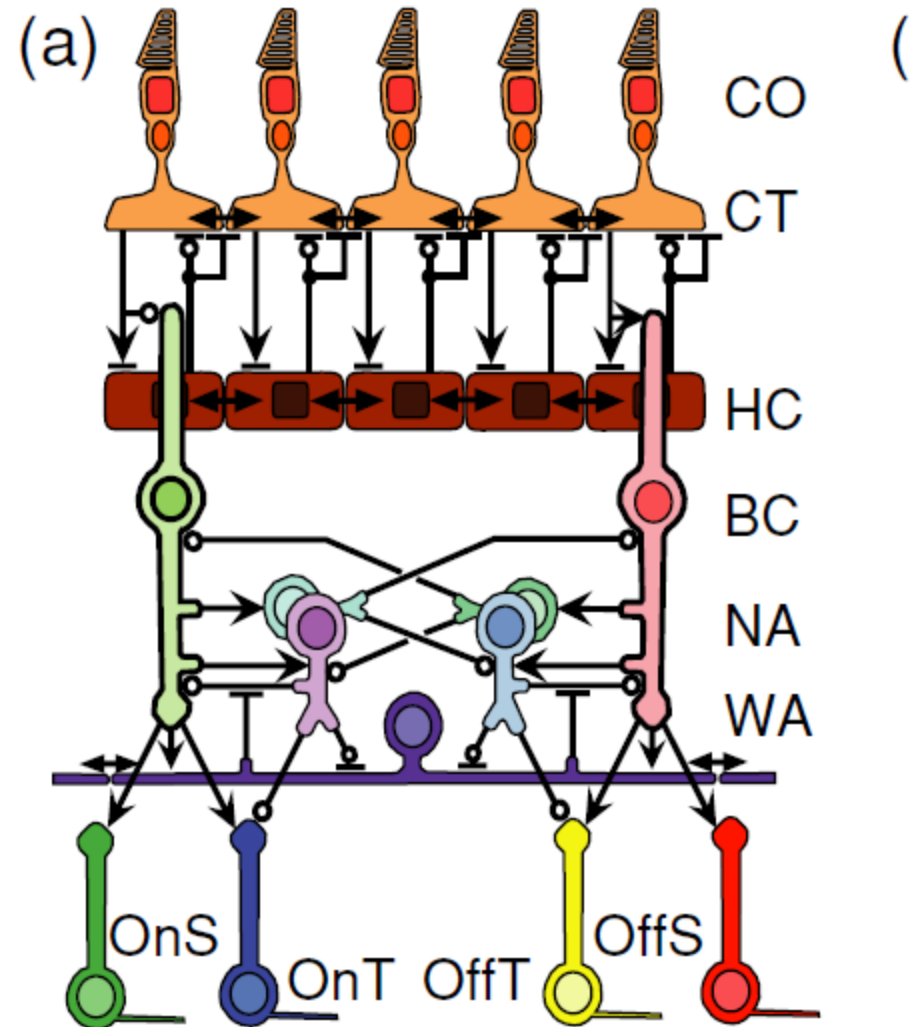


Example: Silicon Retina

Simulates function
of biological retina

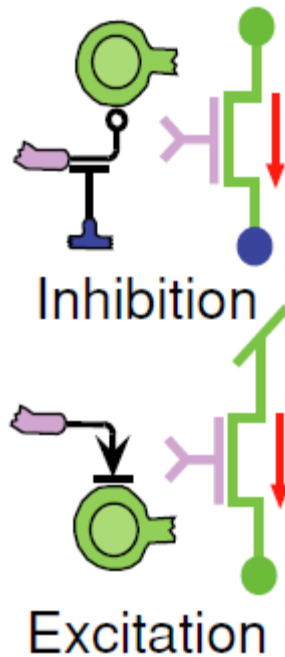
Single-transistor
synapses adapt to
luminance,
temporal contrast

Modeling retina
directly on chip
=> requires 100x
less power!

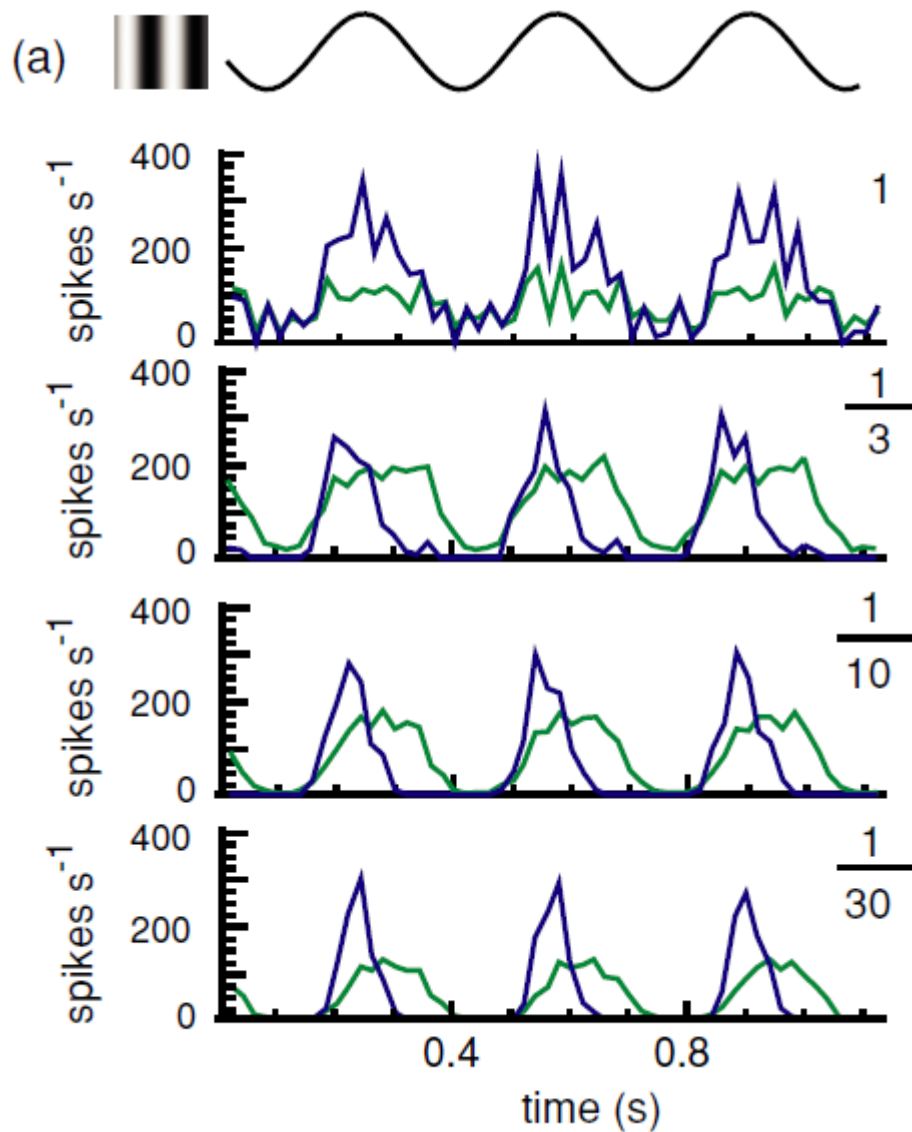


Example: Silicon Retina

- Synapses modeled with single transistors

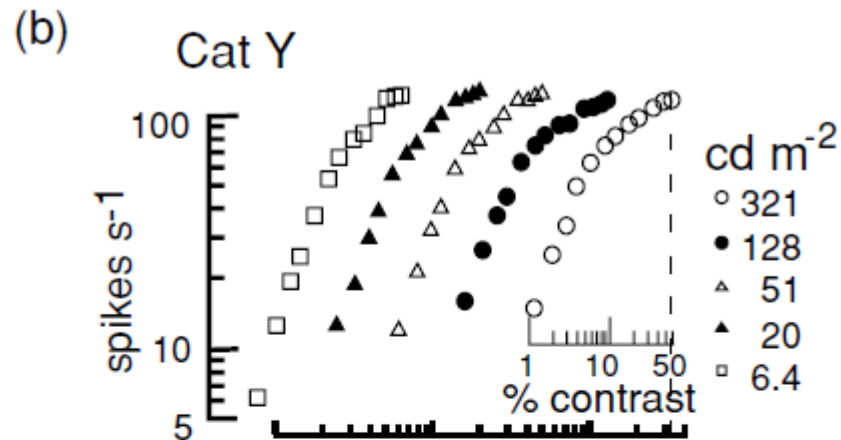


Luminance Adaptation

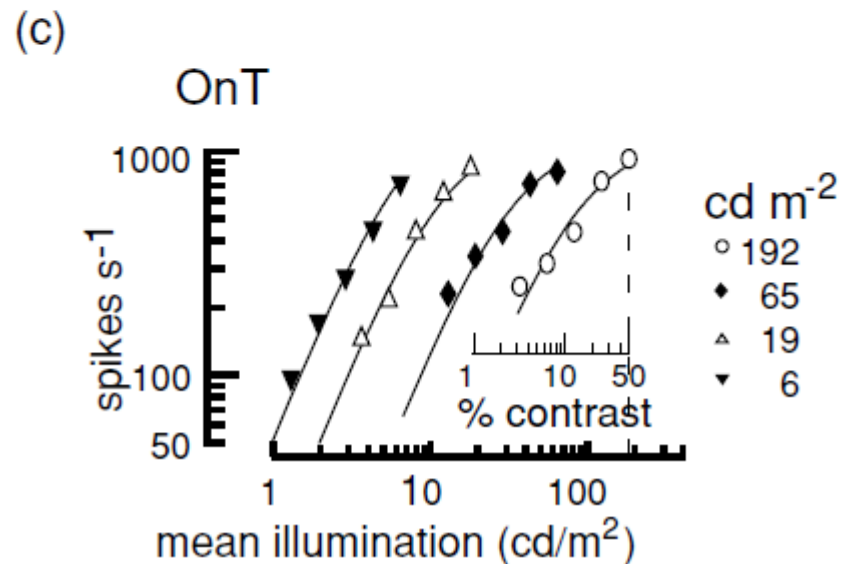


Comparison with Mammal Data

- Real:



- Artificial:



-
- Graphics and results taken from:

INSTITUTE OF PHYSICS PUBLISHING

J. Neural Eng. 3 (2006) 257–267

JOURNAL OF NEURAL ENGINEERING

[doi:10.1088/1741-2560/3/4/002](https://doi.org/10.1088/1741-2560/3/4/002)

A silicon retina that reproduces signals in the optic nerve

Kareem A Zaghoul¹ and Kwabena Boahen^{2,3}

General NN learning in silicon?

- People seem more excited about / satisfied with GPUs

Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Hidden Layer Representations
- Hopfield Nets
- Neural Networks on Silicon
- **Neural Network language models**

Neural Network Language Models

- Statistical Language Modeling:
 - Predict probability of next word in sequence

I was headed to Madrid , _____

$P(\text{_____} = \text{"Spain"}) = 0.5,$

$P(\text{_____} = \text{"but"}) = 0.2, \text{ etc.}$

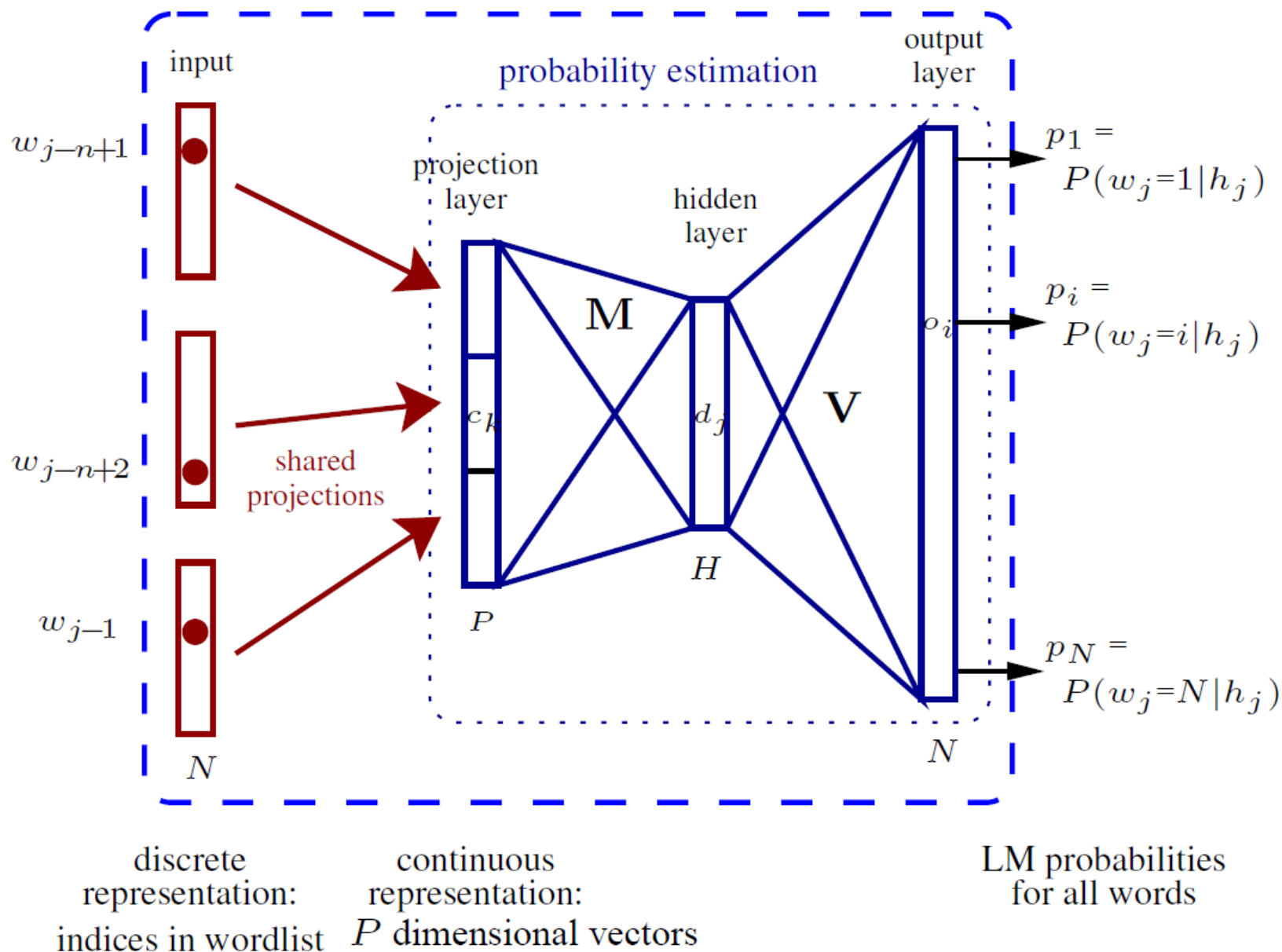
- Used in speech recognition, machine translation, (recently) information extraction

Formally

- Estimate:

$$P(w_j \mid w_{j-1}, w_{j-2}, \dots, w_{j-n+1})$$
$$= P(w_j \mid h_j)$$

Neural Network



Optimizations

- Key idea – learn simultaneously:
 - vector representations of each word (here 120 dim)
 - predictor of next word. based on previous vectors
- Short-lists
 - Much complexity in hidden->output layer
 - Number of possible next words is large
 - Only predict a *subset* of words
 - Use a standard probabilistic model for the rest
 - Can also bin words into fixed **classes**

Design Decisions (1)

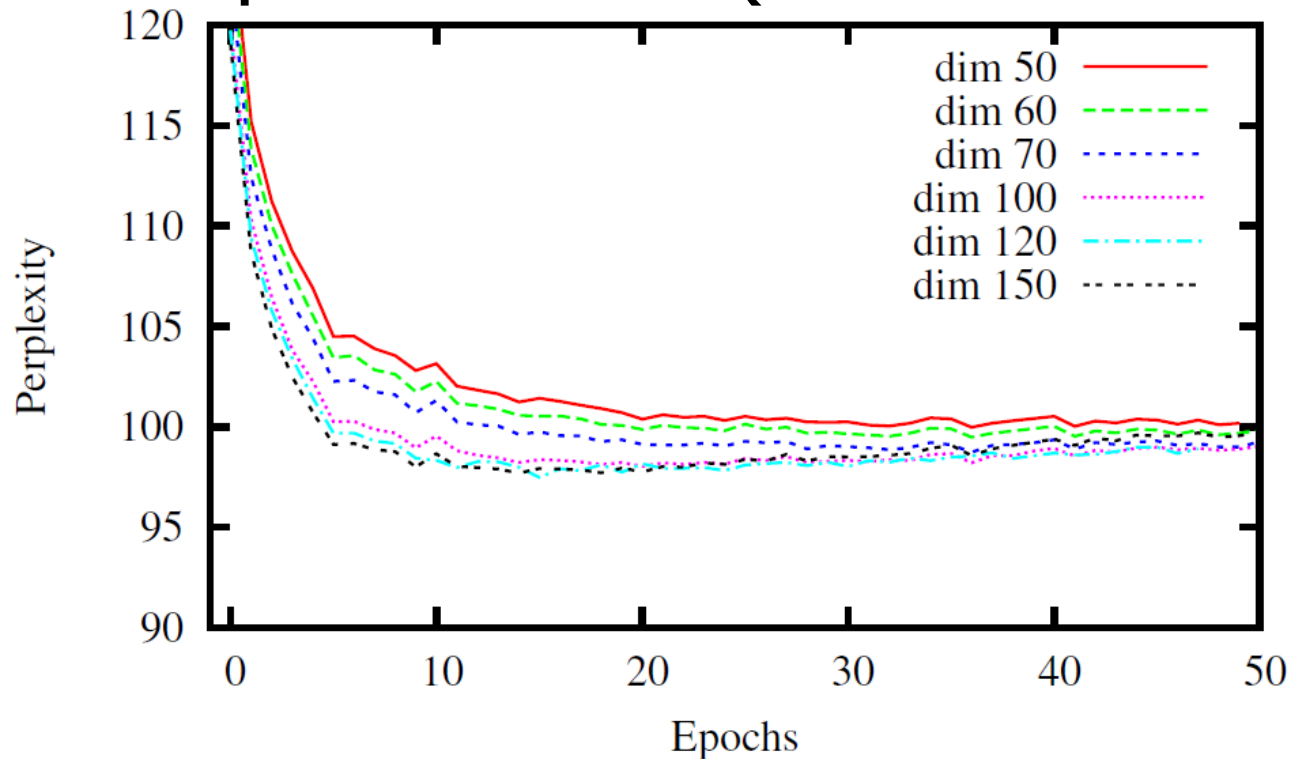
- Number of hidden units

size	400	500	600	1000*
Tr. time	11h20	13h50	16h15	11+16h
Px alone	100.5	100.1	99.5	94.5
interpol.	68.3	68.3	68.2	68.0
Werr	13.99%	13.97%	13.96%	13.92%

* Interpolation of networks with 400 and 600 hidden units.

Design Decisions (2)

- Word representation (# of dimensions)



- They chose 120 – more recent work uses up to 1000s

Comparison vs. state of the art

- Circa 2005

	Back-off LM	Neural Network LM				
Training data [#words]	600M	4M	22M	92.5M*	600M*	
Training time [h/epoch]	-	2h40	14h	9h40	12h	3 × 12h
Perplexity (NN LM alone)	-	103.0	97.5	84.0	80.0	76.5
Perplexity (interpolated LMs)	70.2	67.6	67.9	66.7	66.5	65.9
Word error rate (interpolated LMs)	14.24%	14.02%	13.88%	13.81%	13.75%	13.61%

* By resampling different random parts at the beginning of each epoch.

Schwenk, Holger, and Jean-Luc Gauvain. "Training neural network language models on very large corpora." *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2005.

More recent results

Model	Num. Params [billions]	Training Time		Perplexity
		[hours]	[CPUs]	
Interpolated KN 5-gram, 1.1B n-grams (KN)	1.76	3	100	67.6
Katz 5-gram, 1.1B n-grams	1.74	2	100	79.9
Stupid Backoff 5-gram (SBO)	1.13	0.4	200	87.9
Interpolated KN 5-gram, 15M n-grams	0.03	3	100	243.2
Katz 5-gram, 15M n-grams	0.03	2	100	127.5
Binary MaxEnt 5-gram (n-gram features)	1.13	1	5000	115.4
Binary MaxEnt 5-gram (n-gram + skip-1 features)	1.8	1.25	5000	107.1
Hierarchical Softmax MaxEnt 4-gram (HME)	6	3	1	101.3
Recurrent NN-256 + MaxEnt 9-gram	20	60	24	58.3
Recurrent NN-512 + MaxEnt 9-gram	20	120	24	54.5
Recurrent NN-1024 + MaxEnt 9-gram	20	240	24	51.3

Chelba, Ciprian, et al. "One billion word benchmark for measuring progress in statistical language modeling." *arXiv preprint arXiv:1312.3005* (2013).