perceptron training rule and delta rule

perceptron training rule:

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

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

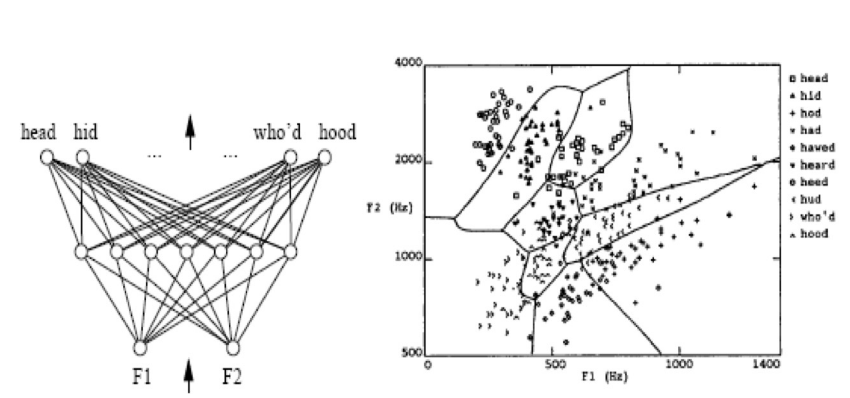
Delta rule:

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

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

The definition of output ***o*** is different!
Perceptron rule updates weights based on the error in the ***thresholded*** perceptron output, whereas delta rule updates weights based on the error in the ***unthresholded linear combination of inputs***

Is linear decision surface enough?



Multilayer network to represent highly nonlinear decision surface

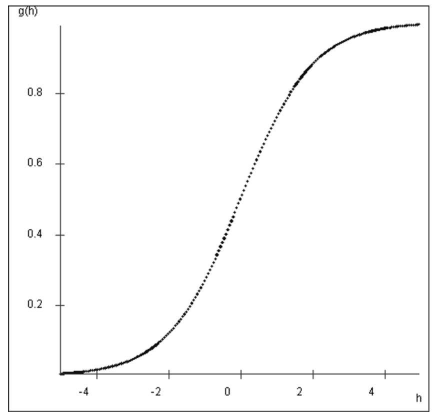
The first question: a differentiable threshold unit

$$g(h) = \frac{1}{1+\exp(-h)}$$

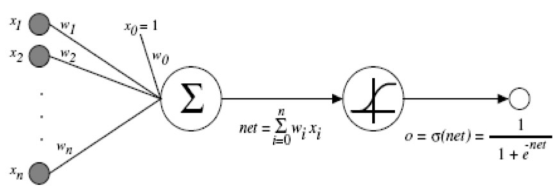
sigmoid unit (logistic function, squashing function)

Note that if you rotate this curve through 180° centered on *(0,1/2)* you get the same curve.

i.e. *g(h)=1-g(-h)*



Differentiable threshold unit



$\sigma(x)$ is the sigmoid function

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

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Error gradient for a sigmoid function

$$
\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\,\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\left(-\frac{\partial o_d}{\partial w_i}\right) \\
&= -\sum_d (t_d - o_d)\,\frac{\partial o_d}{\partial net_d}\,\frac{\partial net_d}{\partial w_i}
\end{aligned}
$$

But we know:

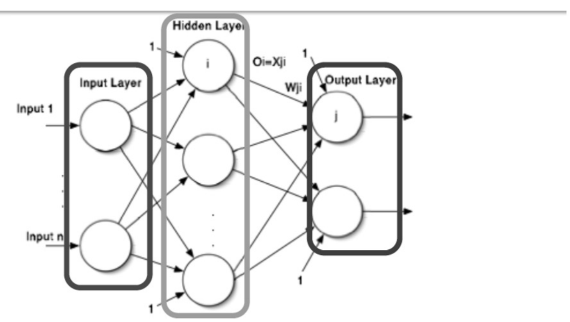
$$\frac{\partial o_d}{\partial net_d} = \frac{\partial \sigma(net_d)}{\partial net_d} = o_d(1 - o_d)$$

$$\frac{\partial net_d}{\partial w_i} = \frac{\partial(\vec{w}\cdot\vec{x}_d)}{\partial w_i} = x_{i,d}$$

So:

$$\frac{\partial E}{\partial w_i} = -\sum_{d\in D}(t_d - o_d)o_d(1 - o_d)x_{i,d}$$

Feed forward neural networks



- A collection of neurons with sigmoid activation, arranged in layers.
- Layer 0 is the **input layer**, its units just copy the input.
- Last layer (layer K) is the **output layer**, its units provide the output.
- Layers *1, .., K-1* are **hidden layers**, cannot be detected outside of network.

Why this name?

- In **feed-forward networks** the output of units in layer *k* become input to the units in layers *k+1, k+2, …, K*.
- No cross-connection between units in the same layer.
- No backward ("recurrent") connections from layers downstream.
- Typically, units in layer *k* provide input to units in layer *k+1* only.
- In **fully-connected networks**, all units in layer *k* provide input to all units in layer *k+1*.

Computing the output of the network

- Suppose we want network to make prediction about instance <***x***,*y=?*>.

Run a **forward pass** through the network.

For layer *k* = 1 … *K*

1. Compute the output of all neurons in layer *k:*

$$o_j = \sigma(\mathbf{w_j} \cdot \mathbf{x_j}), \forall j \in \text{Layer } k$$

2. Copy this output as the input to the next layer:

$$x_{j,i} = o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k+1$$

The output of the last layer is the predicted output *y*.

Learning in feed forward neural networks

- Assume the network structure (units+connections) is given.
- The learning problem is finding a good set of weights to minimize the error at the output of the network.
- Approach: **gradient descent**, because the form of the hypothesis formed by the network, $h_{\mathbf{w}}$ is:
  - **Differentiable**! Because of the choice of sigmoid units.
  - **Very complex**! Hence direct computation of the optimal weights is not possible.

Gradient-descent preliminaries for NN

- Assume we have a fully connected network:
  - *N* input units (indexed *1, …, N*)
  - *H* hidden units in a single layer (indexed *N+1, …, N+H*)
  - one output unit (indexed *N+H+1*)
- Suppose you want to compute the weight update after seeing instance <***x***, *y*>.
- Let o_i, $i = 1, \ldots, H+N+1$ be the outputs of all units in the network for the given input ***x***.
- The sum-squared error function is:

$$J(\mathbf{w}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2 = \frac{1}{2}(y - o_{N+H+1})^2$$

Gradient-descent update for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

$\partial J(w)/\partial \sigma = -(y-o_{N+H+1})$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

Gradient-descent update for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

$\partial J(w)/\partial \sigma = -(y-o_{N+H+1})$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y-o_{N+H+1})o_{N+H+1}(1-o_{N+H+1})x_{N+H+1,j}$$

Gradient-descent update for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -\boxed{(y-o_{N+H+1})o_{N+H+1}(1-o_{N+H+1})}x_{N+H+1,j}$$

- Hence, we can write: $$\frac{\partial J}{\partial w_{N+H+1,j}} = \boxed{-\delta_{N+H+1}}x_{N+H+1,j}$$

where:
$$\delta_{N+H+1} = (y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})$$

Gradient-descent update for **hidden** node

- The derivative wrt the other weights, $w_{l,j}$ where $j = 1, \ldots, N$ and $l = N+1, \ldots, N+H$ can be computed using chain rule:

$$\begin{aligned}\frac{\partial J}{\partial w_{l,j}} &= -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1}) \\ &\quad \cdot \frac{\partial}{\partial w_{l,j}}(\mathbf{w}_{N+H+1} \cdot \mathbf{x}_{N+H+1}) \\ &= -\delta_{N+H+1}w_{N+H+1,l}\frac{\partial}{\partial w_{l,j}}x_{N+H+1,l}\end{aligned}$$

- Recall that $x_{N+H+1,l} = o_l$. Hence we have:

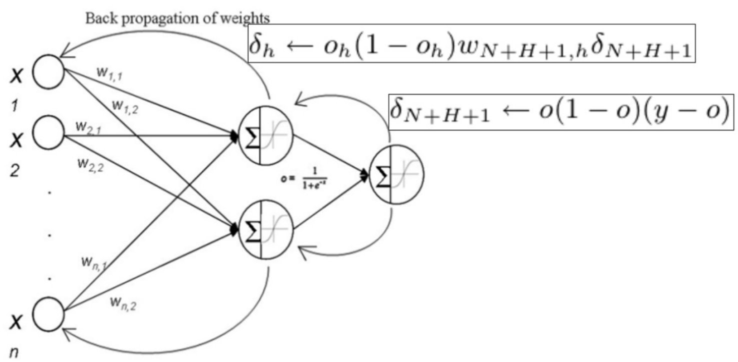
$$\frac{\partial}{\partial w_{l,j}}x_{N+H+1,l} = o_l(1 - o_l)x_{l,j}$$

- Putting these together and using similar notation as before:

$$\frac{\partial J}{\partial w_{l,j}} = -o_l(1 - o_l)\delta_{N+H+1}w_{N+H+1,l}x_{l,j} = -\delta_l x_{l,j}$$

Gradient-descent update for **hidden** node

- The derivative wrt the other weights, $w_{k,j}$ where $j = 1, \ldots, N$ and $k = N+1, \ldots, N+H$ can be computed again using chain rule.



Stochastic gradient descent

- Initialize all weights to small random numbers.
- Repeat until convergence:
  - Pick a training example.
  - Feed example through network to compute output $o = o_{N+H+1}$.
  - For the output unit, compute the correction:

$$\delta_{N+H+1} \leftarrow o(1-o)(y-o)$$

  - For each hidden unit h, compute its share of the correction:

$$\delta_h \leftarrow o_h(1-o_h)w_{N+H+1,h}\delta_{N+H+1}$$

  - Update each network weight:

$$w_{h,i} \leftarrow w_{h,i} + \alpha_{h,i}\delta_h x_{h,i}$$

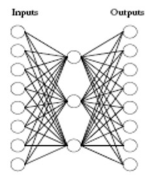
Initialization

Forward pass

Backpropagation

Gradient descent

Learning Hidden layer representation



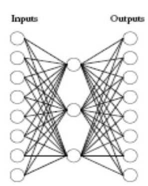
A target function:

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

Can this be learned??

Learning Hidden layer representation

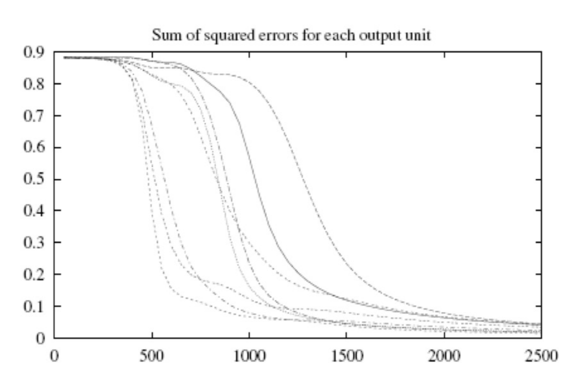
A network:



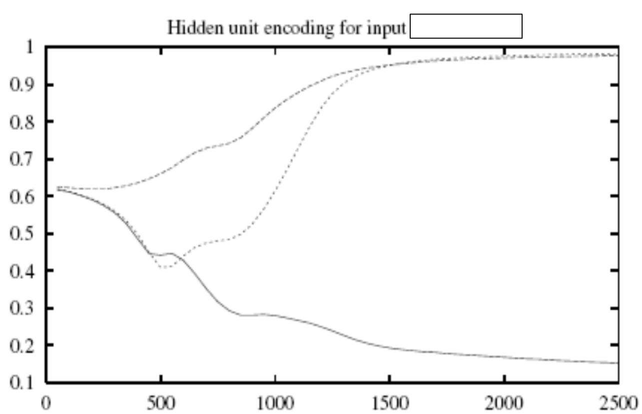
Learned hidden layer representation:

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

Sum of squared errors for each output unit

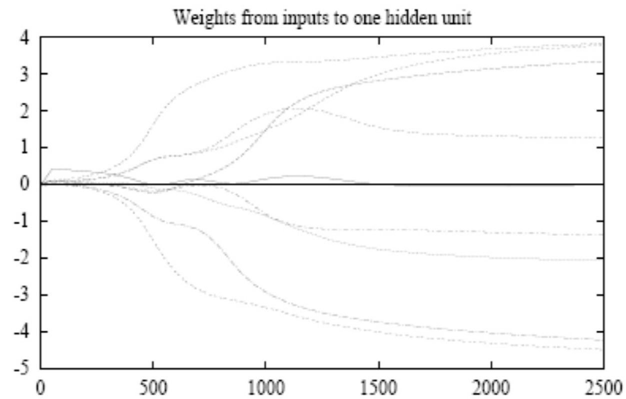


Evolution of the hidden layer representation



Three hidden unit values for one of the possible inputs

Weights from inputs to one hidden unit



Hidden Unit Representations

- Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.
- On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..
- However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.

Source: Raymond J. Mooney, University of Texas at Austin, CS 391L: Machine Learning Neural Networks

Convergence of backpropagation

Gradient descent to some local minimum

- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different inital weights

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Expressive capabilities of ANNs

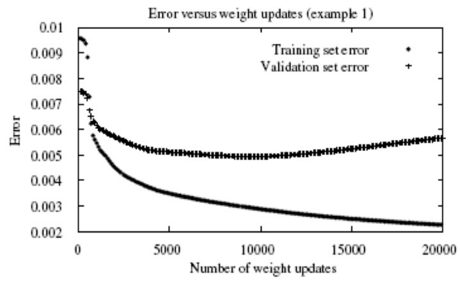
Boolean functions:

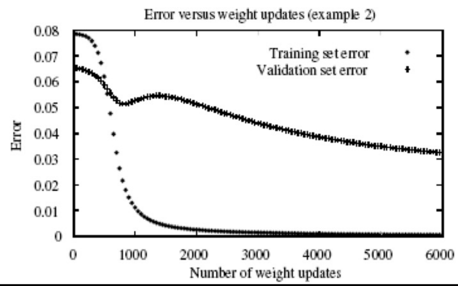
- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:

- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

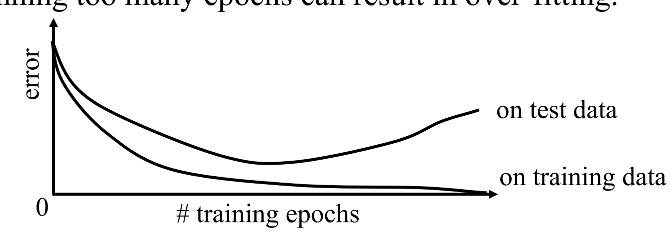
Overfitting of ANNs





Over-Training Prevention

- Running too many epochs can result in over-fitting.

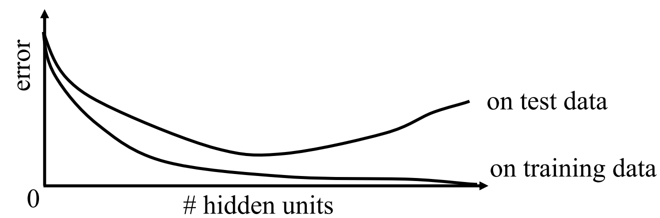


- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.
- To avoid losing training data for validation:
  - Use internal 10-fold cross-validation on the training set to compute the average number of epochs that maximizes generalization accuracy.
  - Train final network on complete training set for this many epochs.

Source: Raymond J. Mooney, University of Texas at Austin, CS 391L: Machine Learning Neural Networks

Determining the Best Number of Hidden Units

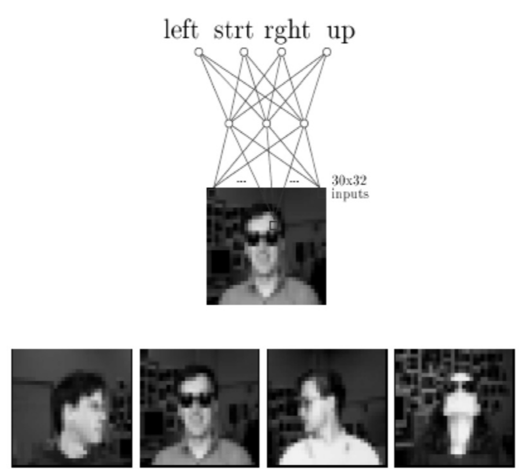
- Too few hidden units prevent the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.



- Use internal cross-validation to empirically determine an optimal number of hidden units.

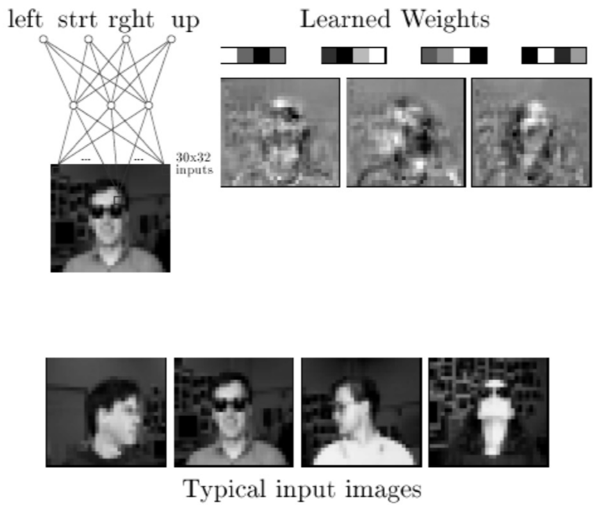
Source: Raymond J. Mooney, University of Texas at Austin, CS 391L: Machine Learning Neural Networks

NN for face recognition



Typical input images

Learned hidden unit weights



http://www.cs.cmu.edu/~tom/faces.html

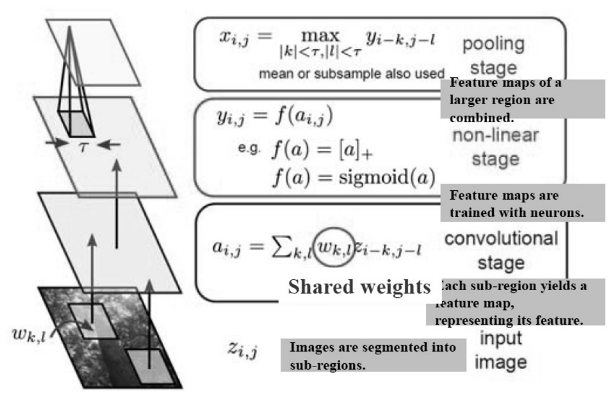
Convolutional Neural Network (CNN)

- Convolutional Neural Networks are inspired by mammalian visual cortex.
  - The visual cortex contains a complex arrangement of cells, which are sensitive to small sub-regions of the visual field, called a receptive field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.
  - Two basic cell types:
    - Simple cells respond maximally to specific edge-like patterns within their receptive field.
    - Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

CNN Structure

- Intuition: Neural network with specialized connectivity structure,
  - Stacking multiple layers of feature extractors
  - Low-level layers extract local features.
  - High-level layers extract learn global patterns.
- A CNN is a list of layers that transform the input data into an output class/prediction.
- There are a few distinct types of layers:
  - Convolutional layer
  - Non-linear layer
  - Pooling layer

Building-blocks for CNN



Summary

- Threshold units
- Gradient descent
- Multilayer networks
- Backpropagation
- Hidden layer representations
- Example: Face Recognition

Summary

- Deep learning = Learning Hierarchical Representations
- Deep learning is thriving in big data analytics, including *image processing, speech recognition,* and *natural language processing*.
- Deep learning has matured and is very promising as an artificial intelligence method.
- Still has room for improvement:
  - Scaling computation
  - Optimization
  - Bypass intractable marginalization
  - More disentangled abstractions
  - Reasoning from incrementally added facts

Reference

The lecture notes in this lecture are adopted and based on the following information:

- T. M. Mitchell, Machine Learning, McGraw Hill, 1997. ISBN: 978-0-07-042807-2
- Learning Systems (course CD5720), Department of Computer Science and Electronics, Mälardalen University. [Online], available: http://www.idt.mdh.se/kurser/cd5720/rjn/2006lp1/
- Statistical Data Mining Tutorials, Dr. Andrew Moore, [Online], available: Andrew' s tutorials: http://www.cs.cmu.edu/~awm/tutorials
- Dr. Qiang Yang, Decision tree, [online], available: http://www.cs.ust.hk/~qyang/521/PPT/dtrees2.ppt#294,1,Classification with Decision Trees II
- Ian H. Witten and Eibe Frank, "Data mining: practical machine learning tools and techniques", Morgan Kaufmann series. 2005
- C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006, ISBN: 978-0-387-31073-2.
- E. Alpaydin, Introduction to Machine Learning, MIT Press, 2004, ISBN 0-262-01211-1
- J. Hawkins, S. Blakeslee, "On Intelligence," Times Books, 2004;
- S. Haykin, "Neural Networks: A Comprehensive Foundation," Prentice Hall, 2nd edition, 1999, ISBN: 0-13-273350-1
- R. Pfeifer, C. Scheier, "Understanding Intelligence," The MIT Press, 2001.
- R. S. Sutton, A. G. Barto, "Reinforcement Learning: An Introduction," MIT Press, 1998.
- The AI lectures from Tokyo: http://tokyolectures.org/
- https://www.cs.mcgill.ca/~jpineau/comp551/Lectures/14NeuralNets.pdf