Neural Networks

Xiaojin Zhu

jerryzhu@cs.wisc.edu

Computer Sciences Department

University of Wisconsin, Madison

Terminator 2 (1991)

JOHN: Can you learn? So you can be... you know. More human. Not such a dork all the time.

TERMINATOR: My CPU is a neural-net processor... a learning computer. But **Skynet** presets the switch to "read-only" when we are sent out alone.

…

We'll learn how to **set** the neural net

TERMINATOR Basically. (starting the engine, backing out) The **Skynet** funding bill is passed. The system goes on-line August 4th, 1997. Human decisions are removed from strategic defense. **Skynet** begins to learn, at a geometric rate. It becomes **self-aware** at 2:14 a.m. eastern time, August 29. In a panic, they try to pull the plug.

SARAH: And **Skynet** fights back.

TERMINATOR: Yes. It launches its ICBMs against their targets in Russia.

SARAH: Why attack Russia?

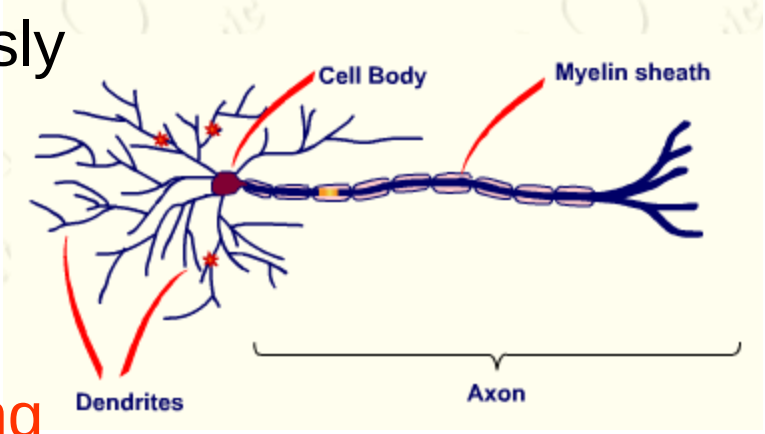
TERMINATOR: Because **Skynet** knows the Russian counter-strike will remove its enemies here.

Outline

- A single neuron
  - Linear perceptron
  - Non-linear perceptron
  - Learning of a single perceptron
  - The power of a single perceptron
- Neural network: a network of neurons
  - Layers, hidden units
  - Learning of neural network: backpropagation
  - The power of neural network
  - Issues
- Everything revolves around gradient descent

Biological neurons

- Human brain: 100, 000, 000, 000 neurons
- Each neuron receives input from 1,000 others
- Impulses arrive simultaneously
- Added together*
  - an impulse can either increase or decrease the possibility of nerve pulse firing



- If sufficiently strong, a nerve pulse is generated
- The pulse forms the input to other neurons.
- The interface of two neurons is called a synapse

http://www.bris.ac.uk/synaptic/public/brainbasic.html

Example: ALVINN

steering direction

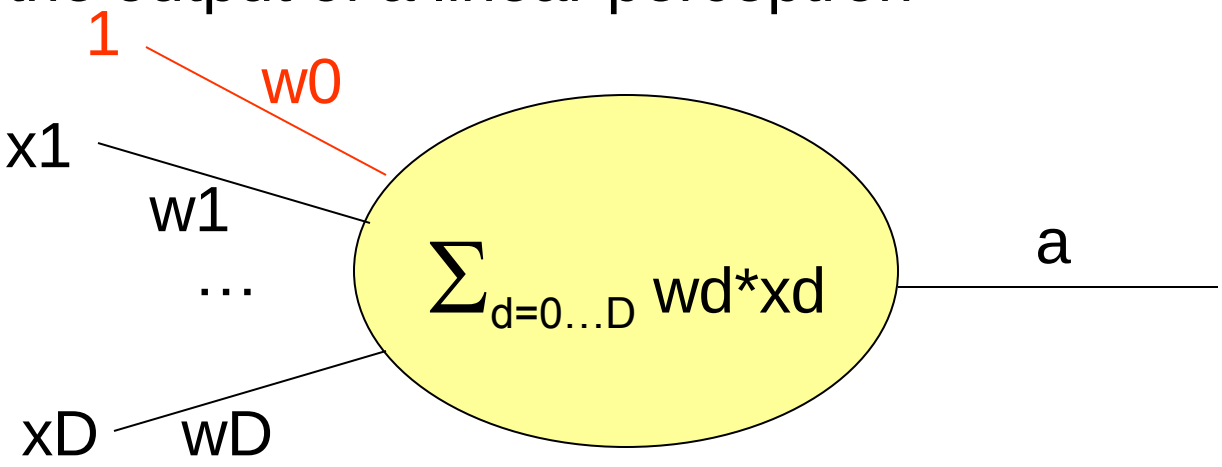
[Pomerleau, 1995]

Linear perceptron

- Perceptron = a math model for a single neuron
- Input: x1, …, xD (signal from other neurons)
- Weights: w1, …, wD (dendrites, can be negative)
- We sneak in a constant (bias term) x0=1, with some weight w0
- Activation function: linear (for the time being)

$$a = w0*x0 + w1*x1 + \ldots + wD*xD$$

- This is the output of a linear perceptron



Learning in linear perceptron

- Regression. Training data {(X1, y1), …, (XN, yN)}
- X1 is a vector: (x11, …, x1D), so are X2…XN
- y1 is a real-valued output

- Goal: learn the weights w0…wD, so that given input Xi, the output of the perceptron ai is close to yi
- Define "close":

$$E = \frac{1}{2} \sum_{i=1..N} (ai-yi)^2$$

- E is the "error". Given the training set, E is a function of w0…wD.
- Minimize E: unconstrained optimization. Variables w0…wD.

Learning in linear perceptron

- Gradient descent: $W \leftarrow W - \alpha \nabla E(W)$
- α is a small constant, “learning rate” = step size
- The gradient descent rule:

$$E(W) = \tfrac{1}{2} \sum_{i=1..N} (ai-yi)^2$$

$$\partial E / \partial wd = \sum_{i=1..N} (ai-yi)\, xid$$

$$wd \leftarrow wd - \alpha \sum_{i=1..N} (ai-yi)\, xid$$

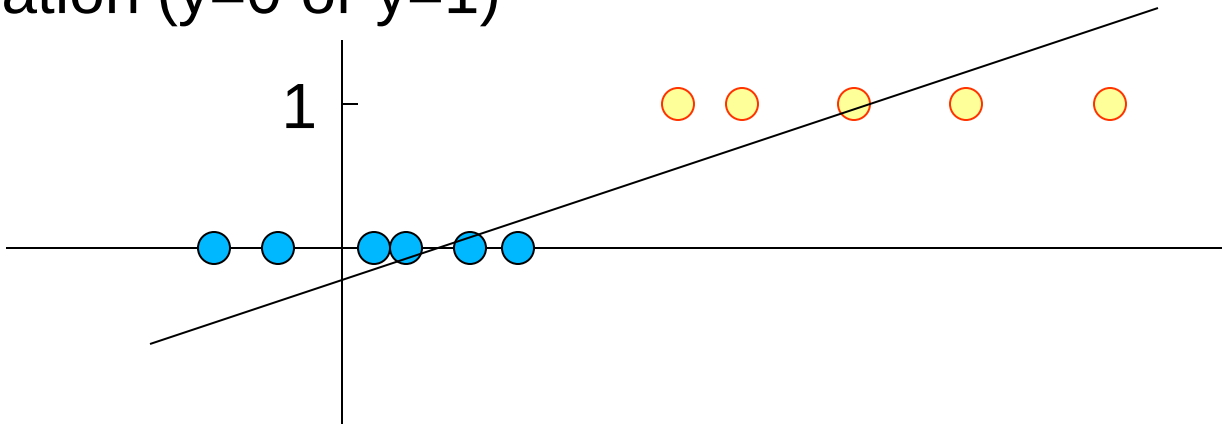
- Repeat until E converges.
- E is convex in W: there is a unique global minimum

The (limited) power of linear perceptron

- Linear perceptron is just

$$a = W'X$$

- where X is the input vector, augmented by x0=1
- It can represent any linear function in D+1 dimensional space… but that's it
- In particular, it won't be a nice fit to binary classification (y=0 or y=1)

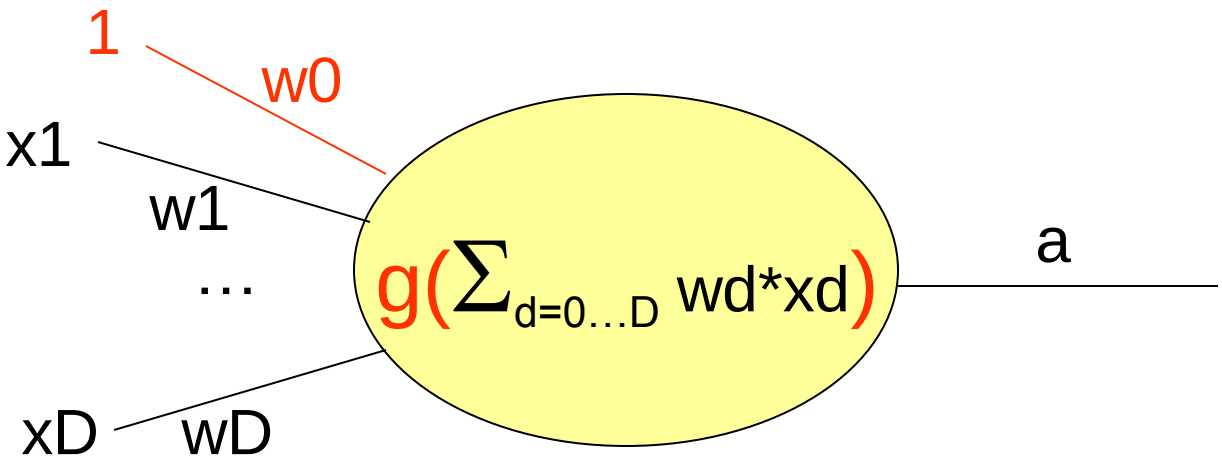


Non-linear perceptron

- Change the activation function: use a step function

  $$a = g(w0*x0 + w1*x1 + \ldots + wD*xD)$$

- g(h)=0, if h < 0; g(h)=1 if h≥0



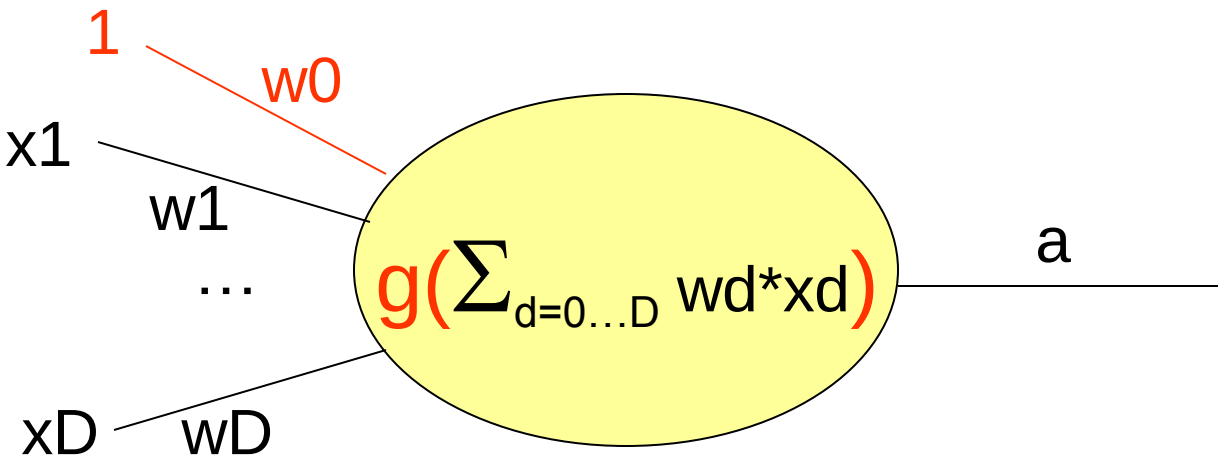
- Can you see how to make logic AND, OR, NOT with such a perceptron?

Linear Threshold Unit (LTU): Our first non-linear perceptron

- Change the activation function: use a step function

$$a = g(w0*x0 + w1*x1 + \ldots + wD*xD)$$

- g(h)=0, if h < 0; g(h)=1 if h≥0



- AND: w1=w2=1, w0= -1.5
- OR: w1=w2=1, w0= -0.5
- NOT: w1= -1, w0= 0.5

Now we see the reason for bias terms

Sigmod activation function: Our second non-linear perceptron

- The problem with LTU: step function is discontinuous, cannot use gradient descent
- Change the activation function (again): use a sigmoid function

$$g(h) = 1 / (1 + \exp(-h))$$

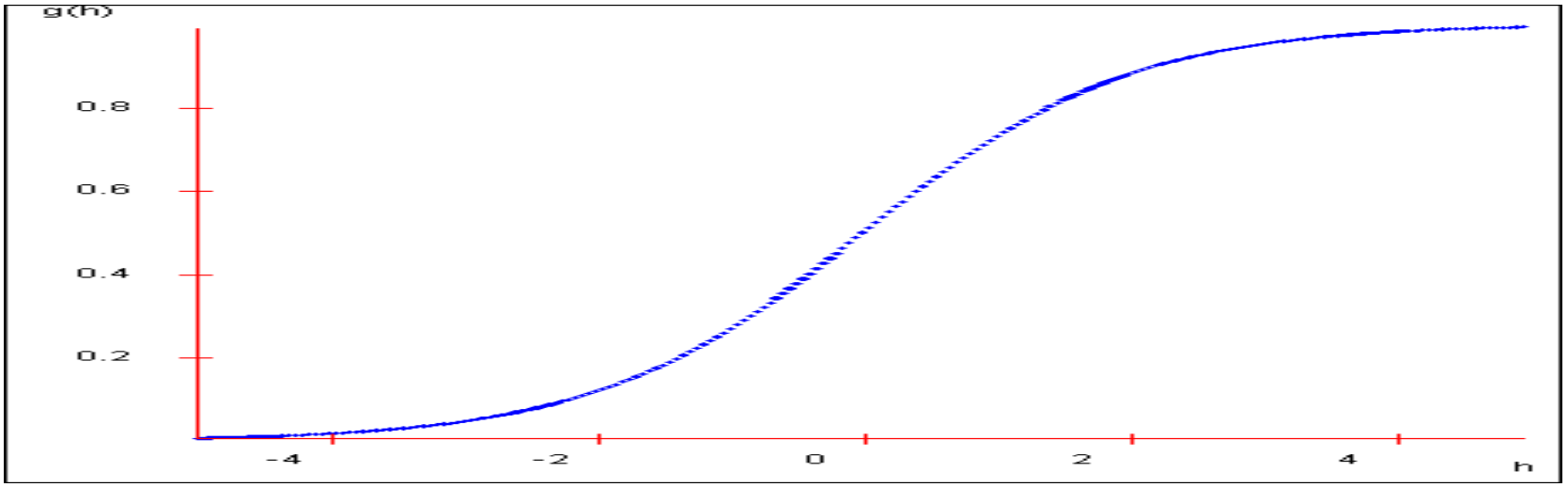
- Exercise: g'(h)=?

Sigmod activation function: Our second non-linear perceptron

- The problem with LTU: step function is discontinuous, cannot use gradient descent
- Change the activation function (again): use a sigmoid function

$$g(h) = 1 / (1 + \exp(-h))$$

- Exercise: g'(h)= g(h) (1-g(h))



Learning in non-linear perceptron

- Again we will minimize the error:

$$E(W) = \tfrac{1}{2} \sum_{i=1..N} (ai-yi)^2$$

- Now $ai = g(\Sigma_d\ wd*xid)$

$$\partial E/\partial wd = \sum_{i=1..N} (ai-yi)\ ai\ (1-ai)\ xid$$

- The sigmoid perceptron update rule

$$wd \leftarrow wd - \alpha \sum_{i=1..N} (ai-yi)\ ai\ (1-ai)\ xid$$

- α is a small constant, “learning rate” = step size
- Repeat until E converges

The (limited) power of non-linear perceptron

- Even with a non-linear sigmoid function, the decision boundary a perceptron can produce is still linear

- AND, OR, NOT revisited

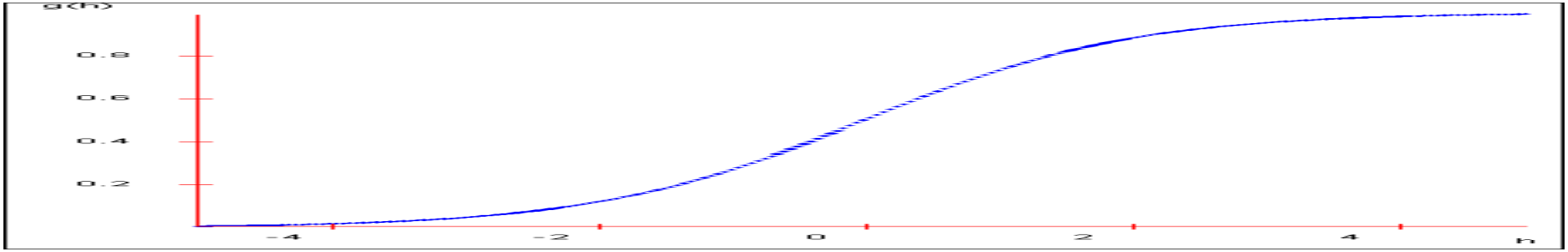
- How about XOR?

The (limited) power of non-linear perceptron

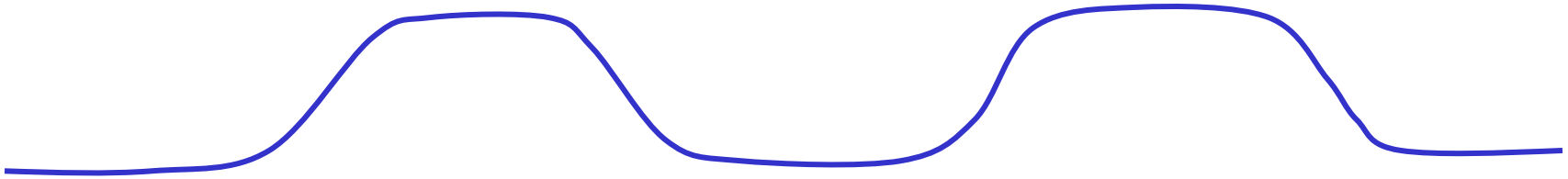
- Even with a non-linear sigmoid function, the decision boundary a perceptron can produce is still linear
- AND, OR, NOT revisited
- How about XOR?
- This contributed to the first AI winter

(Multi-layer) neural network

- Given sigmoid perceptrons



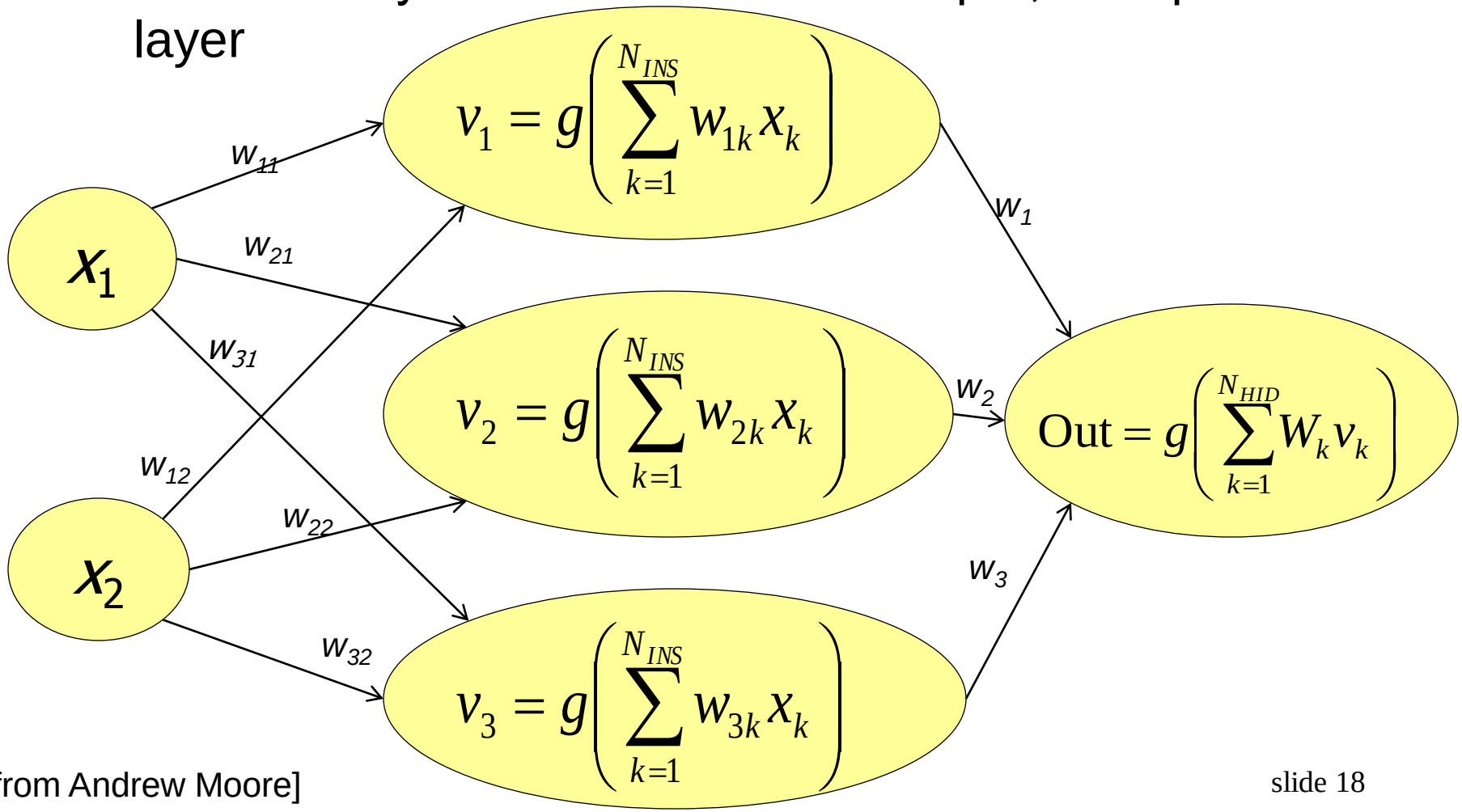
- Can you produce output like



- which had non-linear decision boundarys

Multi-layer neural network

- There are many ways to connect perceptrons into a network. One standard way is multi-layer neural nets
- 1 Hidden layer: we can't see the output; 1 output layer



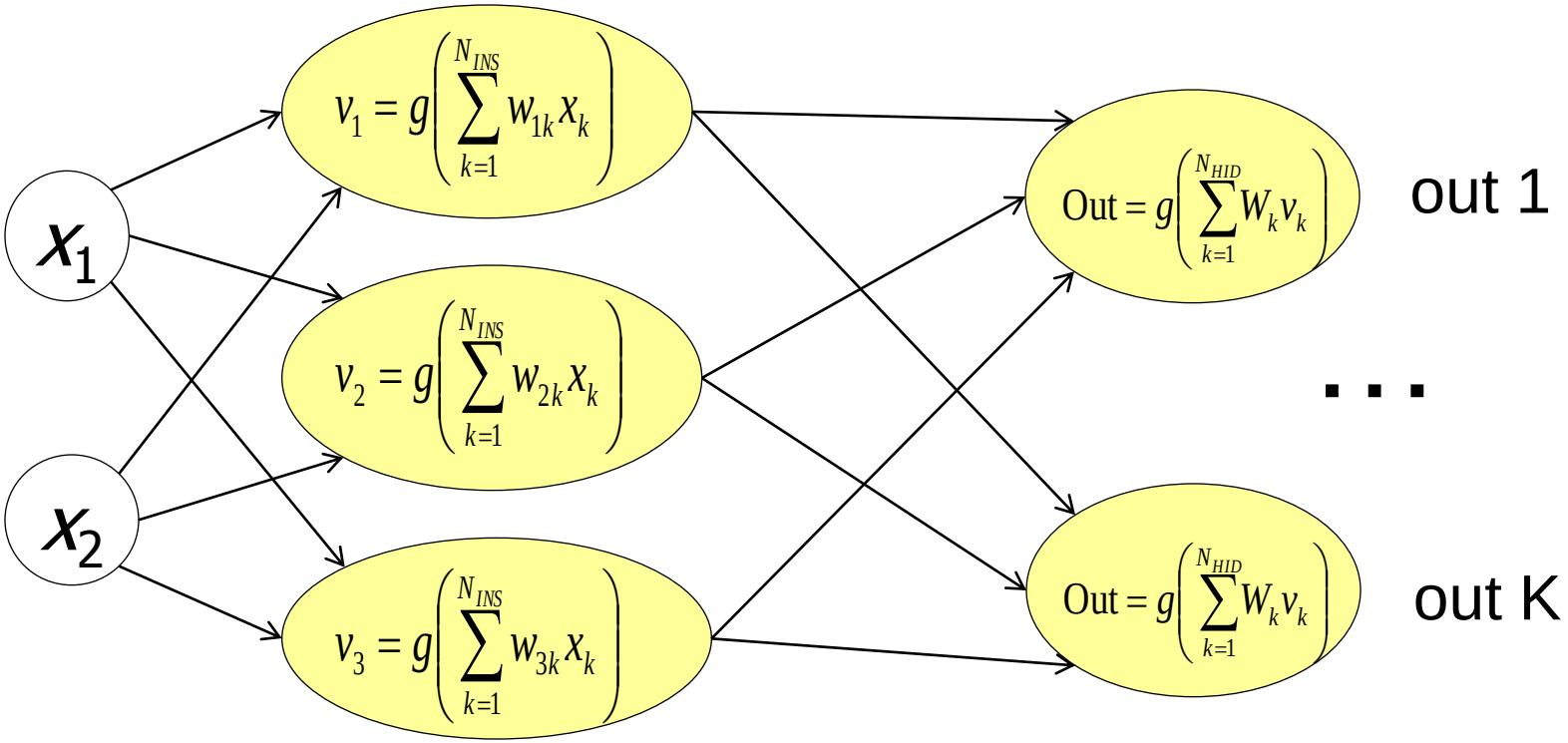
[from Andrew Moore]

The (unlimited) power of neural network

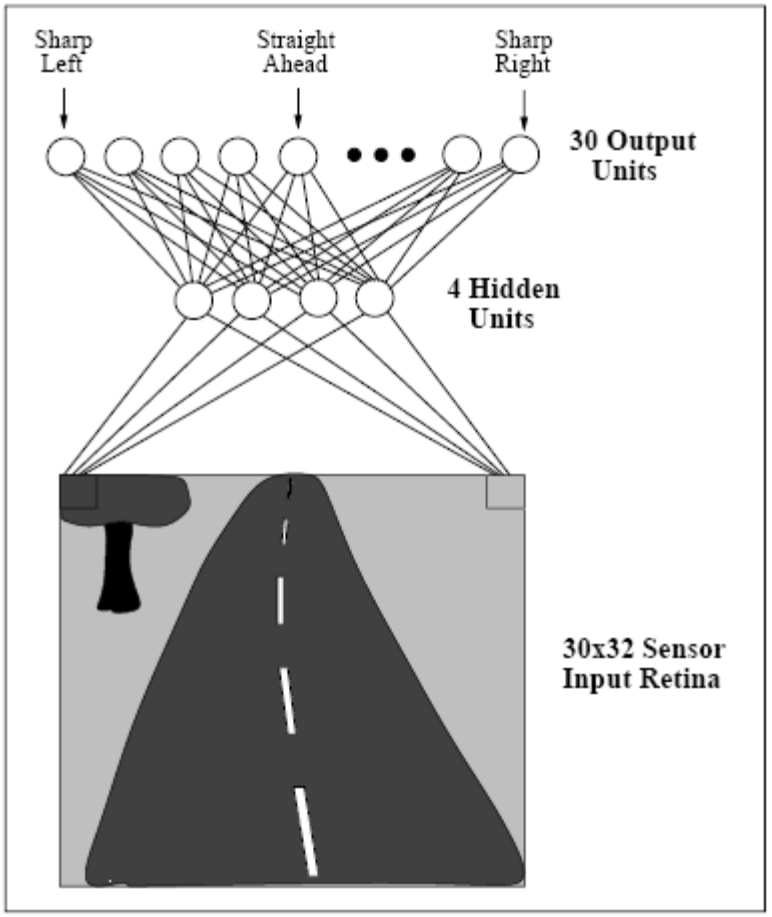
- In theory
  - we don’t need too many layers:
  - 1-hidden-layer net with enough hidden units can represent any continuous function of the inputs with arbitrary accuracy
  - 2-hidden-layer net can even represent discontinuous functions

Neural net for K-way classification

- Use K output units. During training, encode a label *y* by an indicator vector with K entries:
  - class1=(1,0,0,…,0), class2=(0,1,0,…,0) etc.
- During test (decoding), choose the class corresponding to the largest output unit

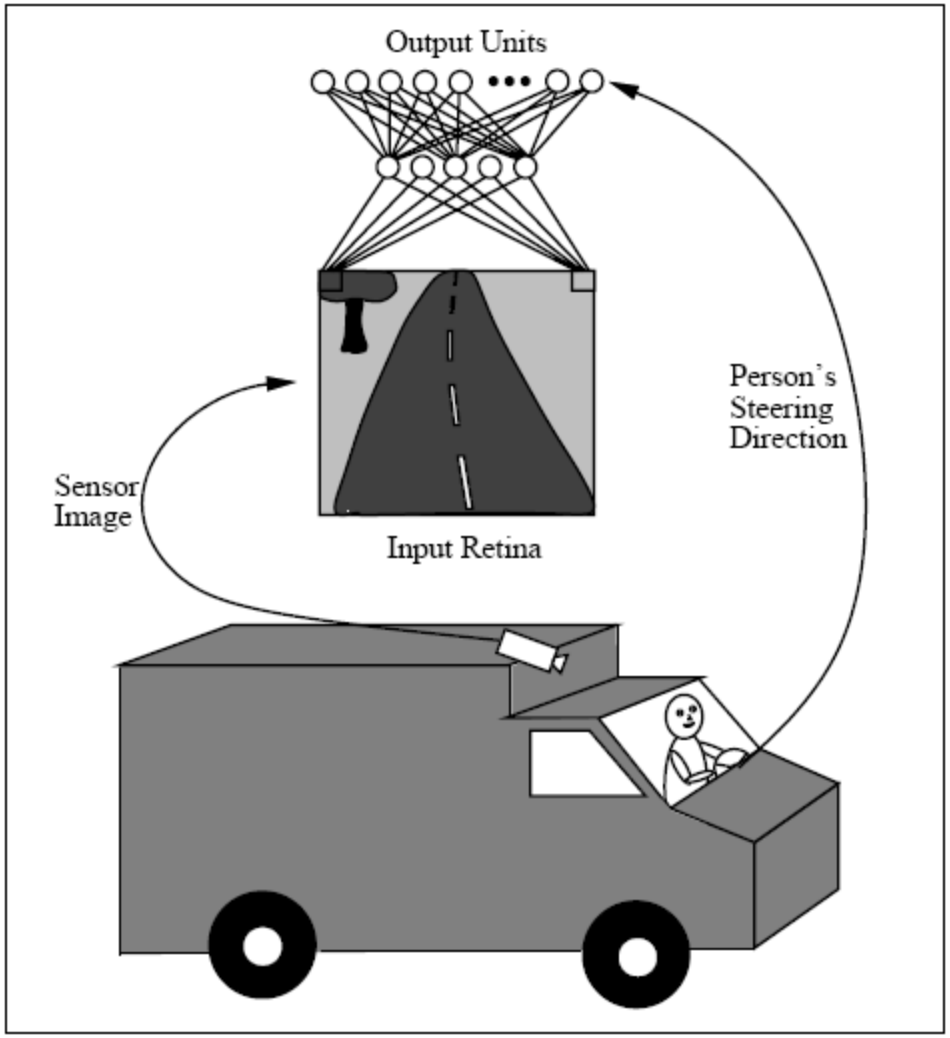


Example Y encoding



[Pomerleau, 1995]

Obtaining training data



[Pomerleau, 1995]

Learning in neural network

- Again we will minimize the error (K outputs):

$$E(W) = \tfrac{1}{2} \sum_{i=1..N} \sum_{c=1..K} (o_{ic} - Y_{ic})^2$$

- i: the i-th training point
- o_{ic}: the c-th output for the i-th training point
- Y_{ic}: the c-th element of the i-th label indicator vector
- Our variables are all the weights w on all the edges
  - Apparent difficulty: we don't know the 'correct' output of hidden units
  - It turns out to be OK: we can still do gradient descent. The trick you need is the chain rule
  - The algorithm is known as **back-propagation**

Backpropagation algorithm (page 1)

BACKPROPAGATION(training set, α, D, n_{hidden}, K)

- Training set: {(X1, Y1), …, (Xn, Yn)}, Xi is a feature vector of size D, Yi is an output vector of size K, α is the learning rate (step size in gradient descent), n_{hidden} is the number of hidden units

- Create a neural network with D inputs, n_{hidden} hidden units, and K outputs. Connect each layer.
- Initialize all weights to some small random numbers (e.g. between –0.05 and 0.05)
- Repeat next page until the termination condition is met…

Backpropagation algorithm (page 2)

For each training example (X, Y):

- Propagate the input forward through the network

  Input X to the network, compute output o_u for every unit *u* in the network

- Propagate the errors backward through the network
  - for each output unit *c*, compute its error term δ_c

$$\delta_c \leftarrow (o_c - y_c) o_c (1 - o_c)$$

  - for each hidden unit *h*, compute its error term δ_h

$$\delta_h \leftarrow \left(\sum_{i \in succ(h)} w_{ih} \delta_i \right) o_h (1 - o_h)$$

  - update each weight w_{ji}

$$w_{ji} \leftarrow w_{ji} - \alpha \delta_j x_{ji}$$

    - where x_{ji} is the input from unit *i* into unit *j* (o_i if *i* is a hidden unit; X_i if *i* is an input)
    - w_{ji} is the weight from unit i to unit j

Derivation of backpropagation

- For simplicity we assume online learning (as oppose to batch learning): 1-step gradient descent after seeing each training example (X,Y)
- For each (X,Y), the error is

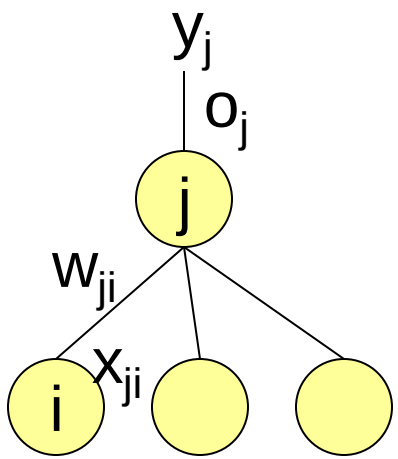
$$E(W) = \frac{1}{2} \sum_{c=1..K} (o_c - Y_c)^2$$

  - o_c: the c-th output unit (when input is X)
  - Y_c: the c-th element of the label indicator vector
- Use gradient descent to change all the weights w_{ji} to minimize the error. Separate two cases:
  - Case 1: w_{ji} when j is an output unit
  - Case 2: w_{ji} when j is a hidden unit

Case 1: weights of an output unit

$$\frac{\partial Error}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(o_j - y_j)^2}{\partial w_{ji}} = \frac{\partial \frac{1}{2}(g(\sum_m w_{jm}x_{jm}) - y_j)^2}{\partial w_{ji}}$$

$$= (o_j - y_j)o_j(1-o_j)x_{ji}$$



- o_c: the c-th output unit (when input is X)
- Y_c: the c-th element of the label indicator vector

- gradient descent: to minimize error, run away from the partial derivative

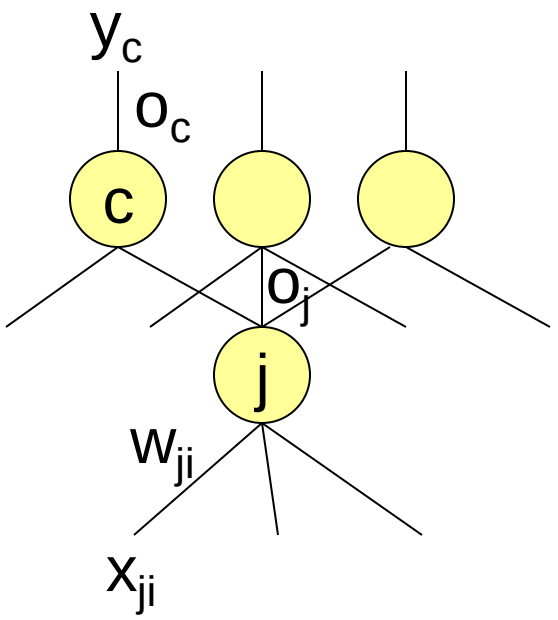
$$w_{ji} \leftarrow w_{ji} - \alpha \frac{\partial Error}{\partial w_{ji}} = w_{ji} - \alpha(o_j - y_j)o_j(1-o_j)x_{ji}$$

Case 2: weights of a hidden unit

$$\frac{\partial Error}{\partial w_{ji}} = \sum_{c=succ(j)} \frac{\partial E_c}{\partial o_c} \cdot \frac{\partial o_c}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ji}}$$

$$= \sum_{c=succ(j)} (o_c - y_c) \cdot \frac{\partial g(\sum_m w_{cm} x_{cm})}{\partial x_{cm}} \cdot \frac{\partial g(\sum_n w_{jn} x_{jn})}{\partial w_{ji}}$$

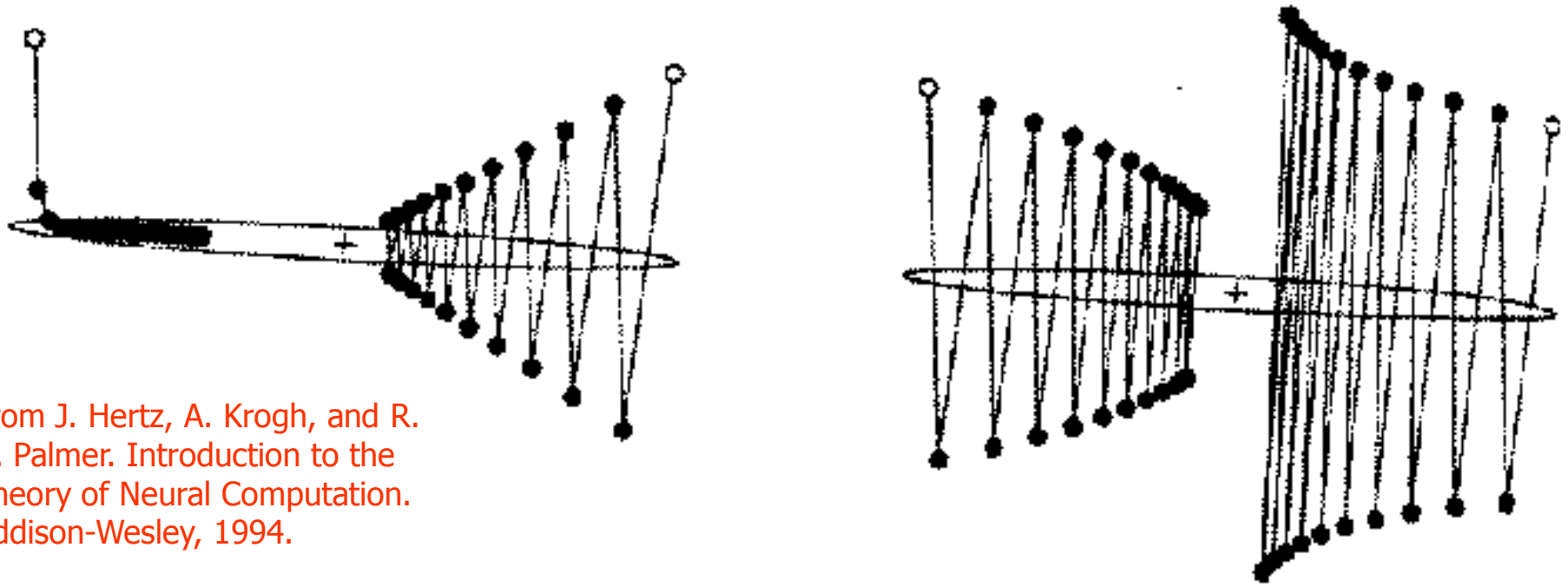
$$= \sum_{c=succ(j)} (o_c - y_c) \cdot o_c(1-o_c) w_{cj} \cdot o_j(1-o_j) x_{ji}$$



Neural network weight learning issues

- When to terminate backpropagation? Overfitting and early stopping
  - After fixed number of iterations (ok)
  - When training error less than a threshold (wrong)
  - When holdout set error starts to go up (ok)
- Local optima
  - The weights will converge to a local minimum
- Learning rate
  - Convergence sensitive to learning rate
  - Weight learning can be rather slow

Sensitivity to learning rate



From J. Hertz, A. Krogh, and R. G. Palmer. Introduction to the Theory of Neural Computation. Addison-Wesley, 1994.

FIGURE 5.10 Gradient descent on a simple quadratic surface (the left and right parts are copies of the same surface). Four trajectories are shown, each for 20 steps from the open circle. The minimum is at the + and the ellipse shows a constant error contour. The only significant difference between the trajectories is the value of η, which was 0.02, 0.0476, 0.049, and 0.0505 from left to right.

Neural network weight learning issues

- Use 'momentum' (a heuristic?) to dampen gradient descent

$$\Delta w^{(t-1)} = \text{last time's change to } w$$

$$\Delta w^{(t)} = -\alpha \, \partial E(W) / \partial w + \beta \, \Delta w^{(t-1)}$$

$$w \leftarrow w + \Delta w^{(t)}$$

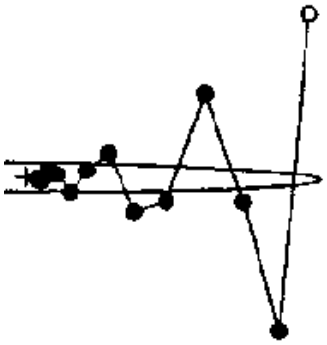


FIGURE 6.3 Gradient descent on the simple quadratic surface of Fig. 5.10. Both trajectories are for 12 steps with $\eta = 0.0476$, the best value in the absence of momentum. On the left there is no momentum ($\alpha = 0$), while $\alpha = 0.5$ on the right.

- Alternatives to gradient descent: Newton-Raphson, Conjugate gradient

Neural network structure learning issues

- How many hidden units?
- How many layers?
- How to connect units?

- Cross validation