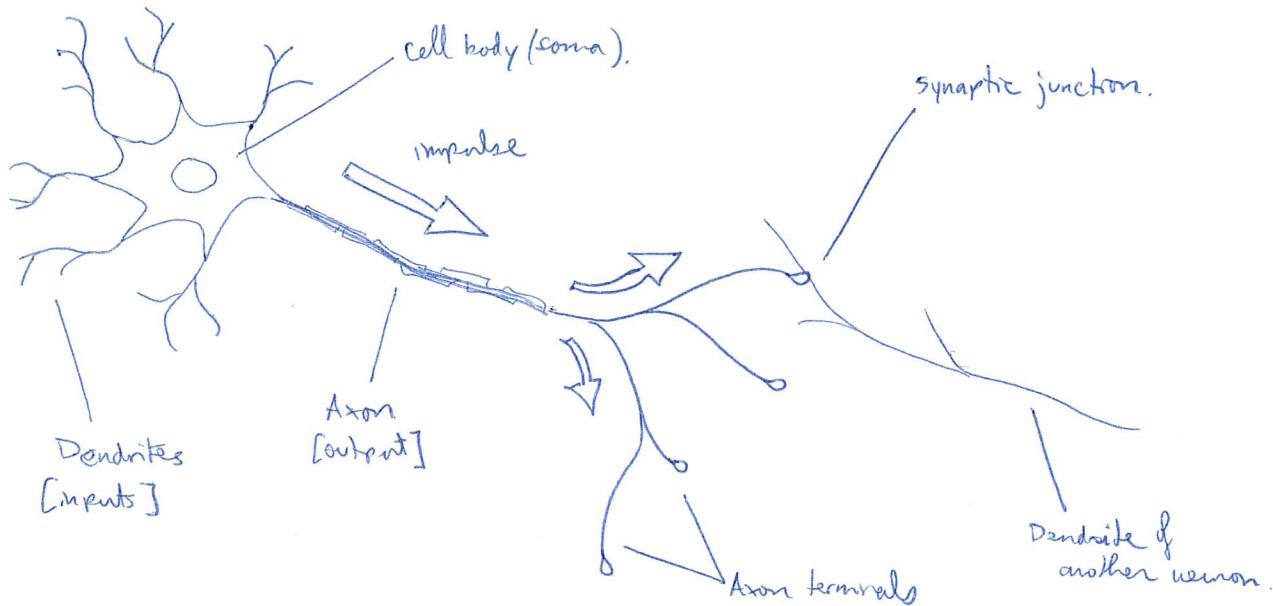


Neural networks

- inspired by the human brain!

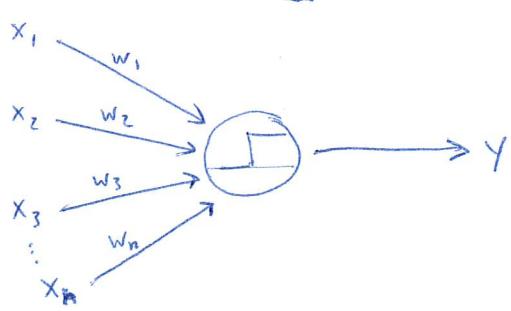
Everything has a different name!

1a



- signals arrive through the dendrites, change the resting potential of the cell.
- the changes are roughly additive. When they cross a threshold, the neuron "fires" and sends action potential through its axon to other neurons via synaptic junctions with other dendrites.

abstraction: (Perception).

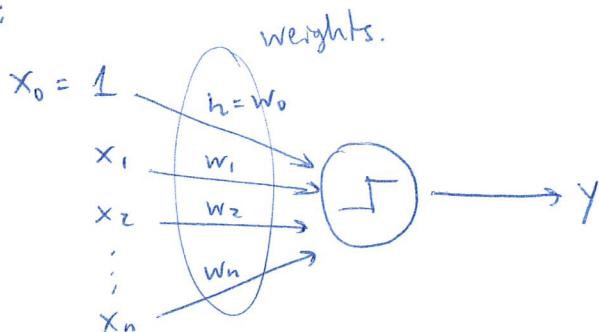


$$(i) \text{ compute sum: } s = \sum_{i=1}^n x_i w_i; \text{ weighted}$$

$$(ii) \text{ if } s > h, \text{ then } y = 1 \\ \text{if } s < h, \text{ then } y = 0.$$

h is the threshold.

alternate:



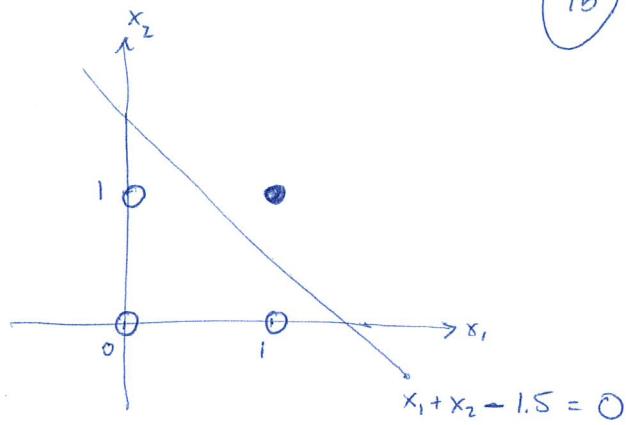
$$y = \text{sign} \left(\sum_{i=0}^n w_i x_i \right) \\ (\text{either } 0-1 \text{ or } \pm 1),$$

Sample example

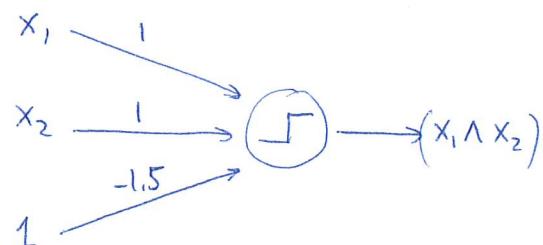
(1b)

AND:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

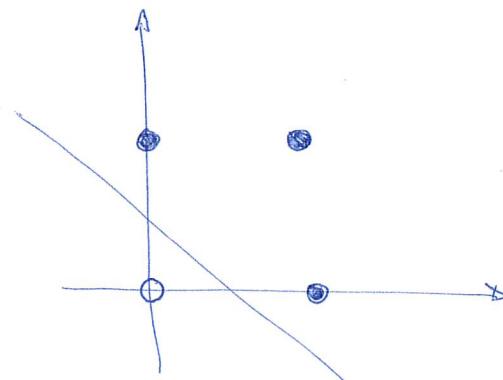


network:

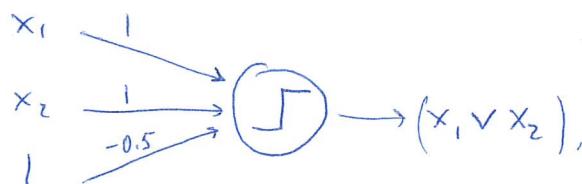


OR:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



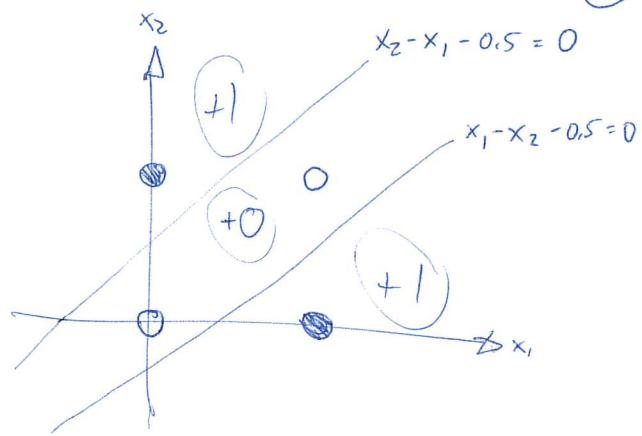
network:



1c

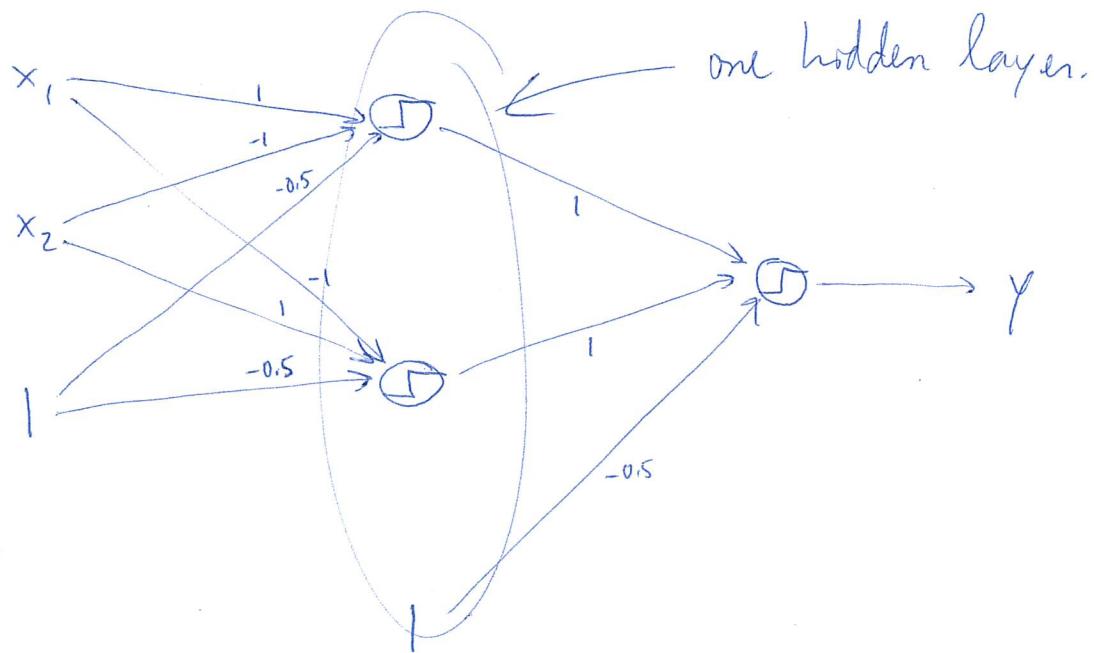
XOR:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



single layer won't do the job!

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND not } x_2) \text{ OR } (x_2 \text{ AND not } x_1).$$



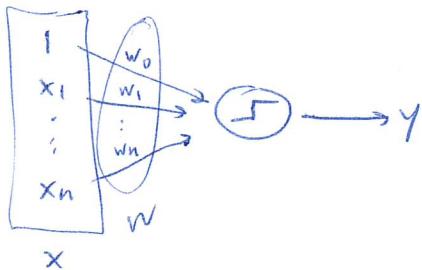
Universal approximation: an ANN with a single hidden layer can approximate any function!

Process is simple (in principle).

②

1. given training examples $\{x_j, y_j\}$, "learn" the weight w_i
2. once weights are learned, can use network to predict y for new x_i .

The "perceptron algorithm"



$$y = \text{sign}(x^T w).$$

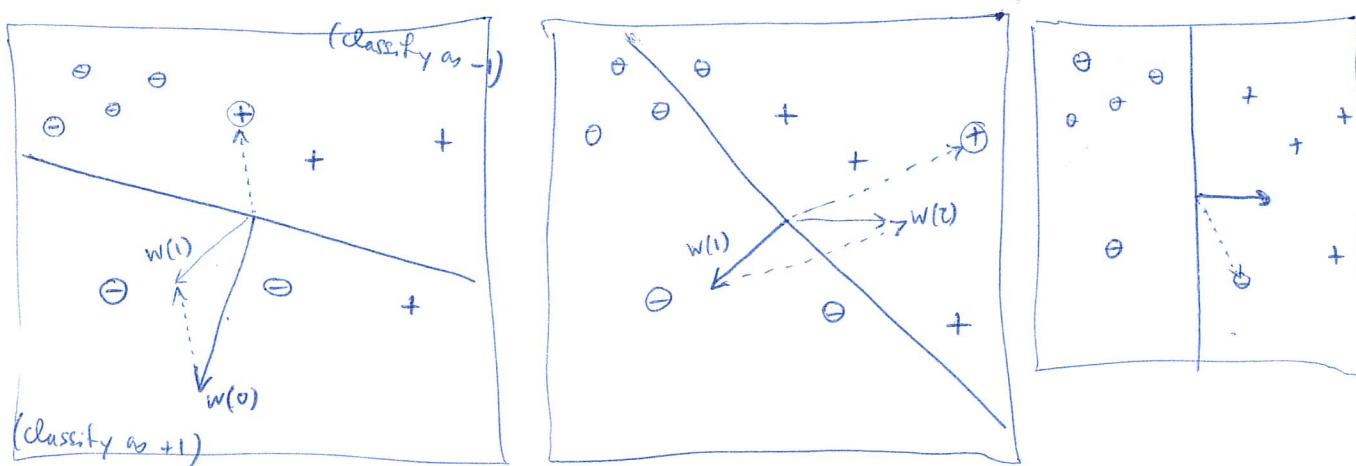
$$\text{For } i^{\text{th}} \text{ example: } y_i = \text{sign}(x_{i \cdot}^T w).$$

1. initialize $w(0)$ with random values.
2. For each j (# training example), calculate $\hat{y}_j = \text{sign}(x_j^T w(t))$.
3. update weights: $w(t+1) = w(t) + \eta (y_j - \hat{y}_j) x_j$
4. repeat steps 2-3 as required.

η is the learning rate

↑
actual label ↑
predicted label

interpretation: - if $y_j = \hat{y}_j$, do nothing;
- if $y_j \neq \hat{y}_j$, move toward correct point.



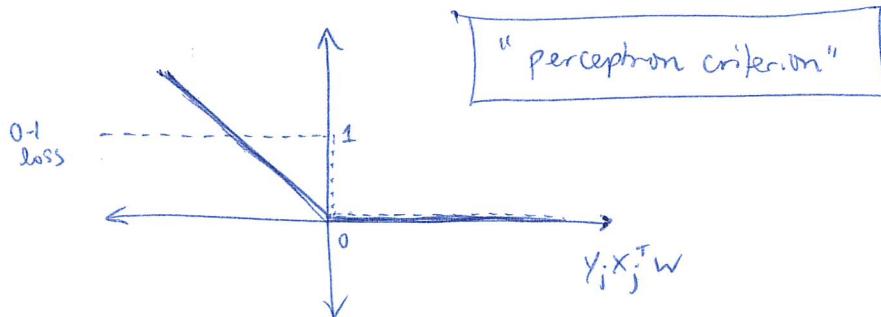
(3)

* Perceptron algorithm is actually gradient descent!
(sub)

$$\begin{aligned}
 w(t+1) &= w(t) + \eta (y_j - \hat{y}_j) x_j \\
 &= w(t) + \eta (y_j - \underbrace{\text{sign}(x_j^T w(t))}_{\begin{cases} 0 & \text{if } y_j = 1, x_j^T w(t) > 0 \\ 0 & \text{if } y_j = -1, x_j^T w(t) < 0 \\ -1 & \text{if } y_j = -1, x_j^T w(t) > 0 \\ +1 & \text{if } y_j = 1, x_j^T w(t) < 0 \end{cases}}) x_j \\
 &= w(t) + 2\eta (-y_j x_j^T w(t))_+ y_j x_j
 \end{aligned}$$

$$w(t+1) = \begin{cases} w(t) - 2\eta(-y_j x_j) & \text{if } y_j x_j^T w(t) < 0 \text{ (misclassified)} \\ w(t) & \text{if } y_j x_j^T w(t) > 0 \text{ (correctly classified)} \end{cases}$$

This is just SGD on the loss function $f_j(w) = (-y_j x_j^T w)_+$.

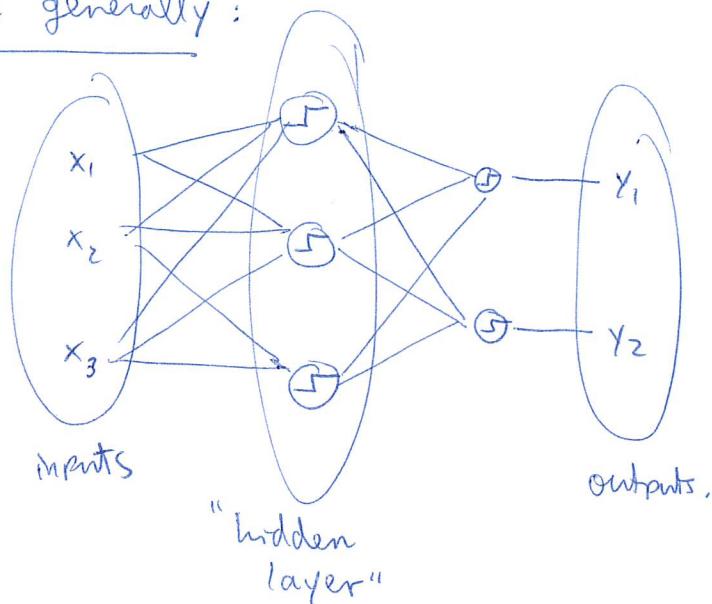


called the "online perceptron algorithm" (SGD)

if we train all data first before updating w , it's
called the batch perceptron algorithm (GD),

(4)

More generally:



"2-layer network."

(two sets of weights).

can have many layers

general neuron: $y = \sigma \left(\sum_{i=1}^m w_i x_i \right)$. where σ could be F or something else.

standard method: minimize "Mean Squared error" (MSE).

$$J(w) = \sum_p \frac{1}{2} (\hat{y}^{(p)} - y^{(p)})^2$$

$$\text{SGD: } w_i(t+1) = w_i(t) - \eta \frac{\partial J}{\partial w_i}$$

$$= w_i(t) - \eta \cdot \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_i}$$

$$= w_i(t) - \eta \cdot (\hat{y} - y) \cdot \sigma' \left(\sum_i w_i x_i \right) \cdot x_i$$

choose σ -function that is differentiable!

standard 0-1 choice: sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ $\sigma'(x) = \sigma(x)(1-\sigma(x))$ nic!

standard ±1 choice: $\sigma(x) = \tanh(x) = \frac{e^{-x} - e^x}{e^x + e^{-x}}$ $\sigma'(x) = 1 - \sigma(x)^2$ also nice!

(5)

So update is:

$$\begin{aligned} w_i(t+1) &= w_i(t) - \eta (\hat{y} - y) \cdot \sigma'(\sum_i w_i x_i) x_i \\ &= w_i(t) - \eta \underbrace{(\hat{y} - y)(1 - \hat{y}^2)}_{\delta_i} x_i \end{aligned}$$

$$w_i(t+1) = w_i(t) - \eta \delta_i x_i$$

"delta rule"

★ δ does not depend
on i !!!Back propagation (a.k.a. the chain rule).

$$\hat{y} = \sigma \left(\sum_i \alpha_i r_i \right) \quad \text{again, let}$$

$$J = \frac{1}{2} (\hat{y} - y)^2$$

$$= \sigma \left(\sum_i \alpha_i \sigma \left(\sum_j \beta_{ij} q_j \right) \right)$$

$$(\hat{y} - y)(1 - \hat{y}^2) = \delta$$

$$\text{Find } \alpha_i \text{ update: } \frac{\partial J}{\partial \alpha_i} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \alpha_i} = \overbrace{(\hat{y} - y) \sigma'(\sum_i \alpha_i r_i)}^{\delta} r_i = \delta r_i$$

~~update~~ therefore: $\alpha_i(t+1) = \alpha_i(t) - \eta \delta^{(p)} r_i^{(p)}$ $\forall i$ δ only depends on p , the p th training example.

(do this for each i).

$$\Rightarrow \boxed{\vec{\alpha}(t+1) = \vec{\alpha}(t) - \eta \delta^{(p)} \vec{r}^{(p)}}$$

(6)

find β_{ij} update

$$\begin{aligned}\frac{\partial J}{\partial \beta_{ij}} &= \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial r_i} \frac{\partial r_i}{\partial \beta_{ij}} = \underbrace{(\hat{y} - y) \sigma'(\sum_i \alpha_i r_i) \alpha_i}_{(\hat{y} - y)(1 - r_i^2)} \cdot \underbrace{\sigma'(\sum_j \beta_{ij} q_j) q_j}_{(1 - r_i^2)} \\ &= \delta \alpha_i (1 - r_i^2) q_j \\ &= \bar{\delta}_i q_j\end{aligned}$$

) $\delta J / \bar{\delta}_i = \delta \alpha_i (1 - r_i^2)$,
(does not depend on j !).

Therefore:

$$\begin{aligned}\beta_{ij}(t+1) &= \beta_{ij}(t) - \eta \bar{\delta}_i^{(p)} q_j^{(p)} \quad \forall i, j. \\ \Rightarrow \boxed{\beta(t+1) = \beta(t) - \eta \bar{\delta}^{(p)} \bar{q}^{(p)}} \quad &\text{(matrix equation),}\end{aligned}$$

find γ_{jk} update

$$\begin{aligned}\frac{\partial J}{\partial \gamma_{jk}} &= \frac{\partial J}{\partial \hat{y}} \sum_i \frac{\partial \hat{y}}{\partial r_i} \frac{\partial r_i}{\partial \beta_{ij}} \frac{\partial \beta_{ij}}{\partial \gamma_{jk}} = (\hat{y} - y) \sum_i \sigma'(\sum_i \alpha_i r_i) \alpha_i \sigma'(\sum_j \beta_{ij} q_j) \beta_{ij} \sigma'(\sum_k \gamma_{jk} x_k) x_k \\ &= (\hat{y} - y)(1 - \hat{y}^2) \sum_i \alpha_i (1 - r_i^2) \beta_{ij} (1 - q_j^2) x_k \\ &= \delta \sum_i \alpha_i (1 - r_i^2) \beta_{ij} (1 - q_j^2) x_k \\ &= \bar{\delta}_j \beta_{ij} (1 - q_j^2) x_k\end{aligned}$$

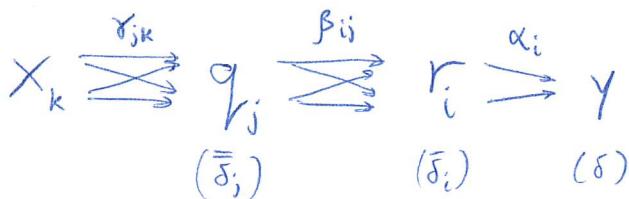
) $\delta J / \bar{\delta}_j = \sum_i \bar{\delta}_i \beta_{ij} (1 - q_j^2)$
(does not depend on k !).

Therefore:

$$\begin{aligned}\gamma_{jk}(t+1) &= \gamma_{jk}(t) - \eta \bar{\delta}_j^{(p)} x_k^{(p)} \\ \Rightarrow \boxed{\gamma(t+1) = \gamma(t) - \eta \bar{\delta}^{(p)} x^{(p)T}} \quad &\text{(matrix equation).}\end{aligned}$$

(7)

once α , β , γ have been updated, move to the next training example.



recap:

- 1) compute \hat{y} (and subsequently also q_j , r_i) by feeding x_k into the network and calculating the corresponding y .

- 2) use \hat{y} to calculate δ .

\hookrightarrow update α_i using (δ, r_i)

- 3) use (δ, α_i, r_i) to calculate $\bar{\delta}_i$

\hookrightarrow update β_{ij} using $(\bar{\delta}_i, q_j)$

- 4) use $(\bar{\delta}_i, \beta_{ij}, q_j)$ to calculate $\bar{\delta}_j$

\hookrightarrow update γ_{jk} using $(\bar{\delta}_j, x_k)$

repeat for
each training
sample.

what is actually going on?

$$\text{if } \text{sign}(w^T x_i) \approx \tanh(w^T x_i)$$

on backprop, we
actually do

$$\text{then } \frac{1}{2}(1 - \text{sign}(y_i x_i^T w)) \approx \frac{1}{2}(1 - \tanh(y_i x_i^T w)).$$

$$\frac{1}{4}(1 - \tanh(y_i x_i^T w))^2$$

