NEURAL NETWORKS

1. The Perceptron

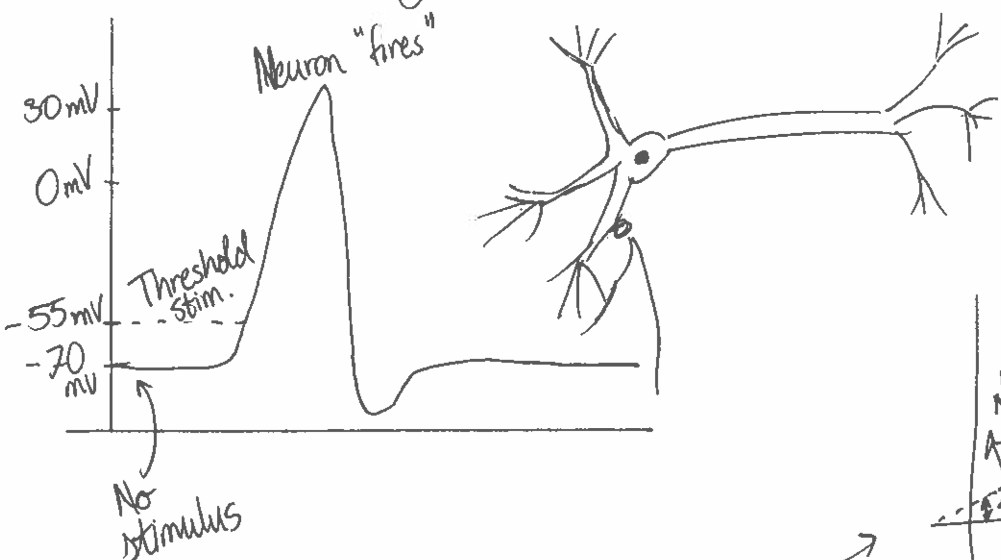
"The Man Who Tried to Redeem the World with Logic", Amanda Gefter.

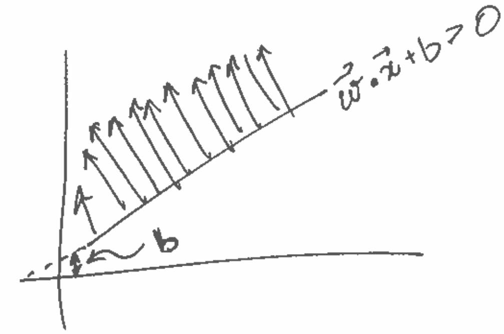
1943 - (Warren) McCulloch - (Walter) Pitts Neuron, A Logical Calculus of Ideas Immanent in Nervous Activity.

1958 - (Frank) Rosenblatt builds the "Mark 1 Perceptron" + Perceptron learning algorithm

400 photocells randomly connected to neurons (Image classification)

McCulloch & Pitts try to emulate human nerve cells:





Mathematically, $f(x) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0 \\ 0 & \text{otherwise.} \end{cases}$ "If input · weights + bias > 0, fire".

Shifts decision boundary.

Linear classifier!

$$\vec{w} \cdot \vec{x} = \sum_{i=1}^{n} w_i x_i, \quad n = \text{no. of inputs.}$$

This is great - but real trick is how we update the weights: perceptron learning algorithm.

Perceptron learning alg:

DEF: r = learning rate, $r \in [0,1]$. $\uparrow r \Rightarrow$ more drastic weight changes

$y = f(\vec{z}) \Rightarrow$ output for some input $\vec{z}$

$D = \{(x_1, d_1), (x_2, d_2), \dots (x_n, d_n)\}$ is our training set
(x_1: Input, d_1: Label.)

$x_{ji} \Rightarrow$ value of i^{th} feature of j^{th} input vector:

$x_{j0} = 1$.

(If no biases in dot prod).

$$X = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_j \\ \vdots \\ \vec{x}_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ & \vdots & & \\ x_{j1} & x_{j2} & \cdots x_{ji} & \cdots \\ & \vdots & & \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}$$

$n \times k$ matrix

$$= \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1i} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2i} & \cdots & x_{2k} \\ & & & \vdots & & & \\ 1 & x_{j1} & x_{j2} & \cdots & x_{ji} & \cdots & x_{jk} \\ & & & \vdots & & & \\ 1 & x_{n1} & x_{n2} & \cdots & x_{jn} & \cdots & x_{nk} \end{bmatrix}$$

If bias incorporated as first elem. of weight vec.

$w_i \Rightarrow i^{th}$ value of weight vector

$w_0 \Rightarrow$ bias.

$w_i(t) = i^{th}$ value ... at time (t)

1. Initialize weights (orig. 0, but pract. use some small no.)

~~2. For each sample in the training set:~~

2. While not converged:

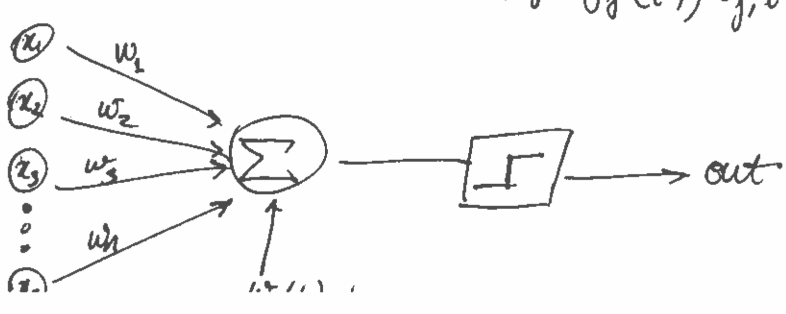
for $j \in D$, i) calculate actual output:

$$y_j(t) = f[\vec{w}(t) \cdot x_j].$$
$$= f[w_0(t) x_{j,0} + w_1(t) x_{j,1} + \dots w_k(t) x_{j,k}].$$

ii) update weights based on error:

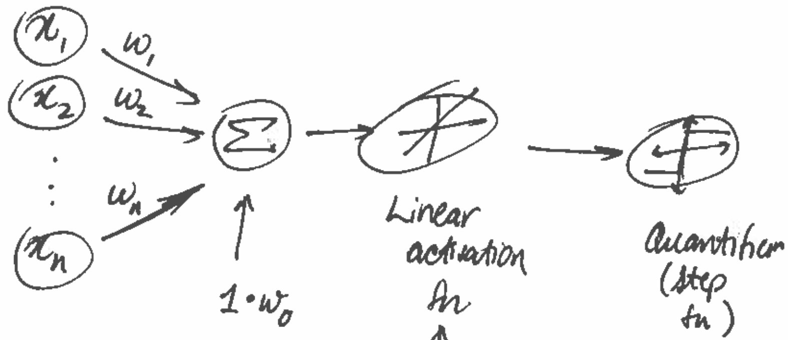
$$w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t)) x_{j,i} \quad \forall i \in k$$

~~Gradient~~ ~~descent~~ Update one at a time!



A better way ~~to~~ calc. errors & update weights: gradient descent!

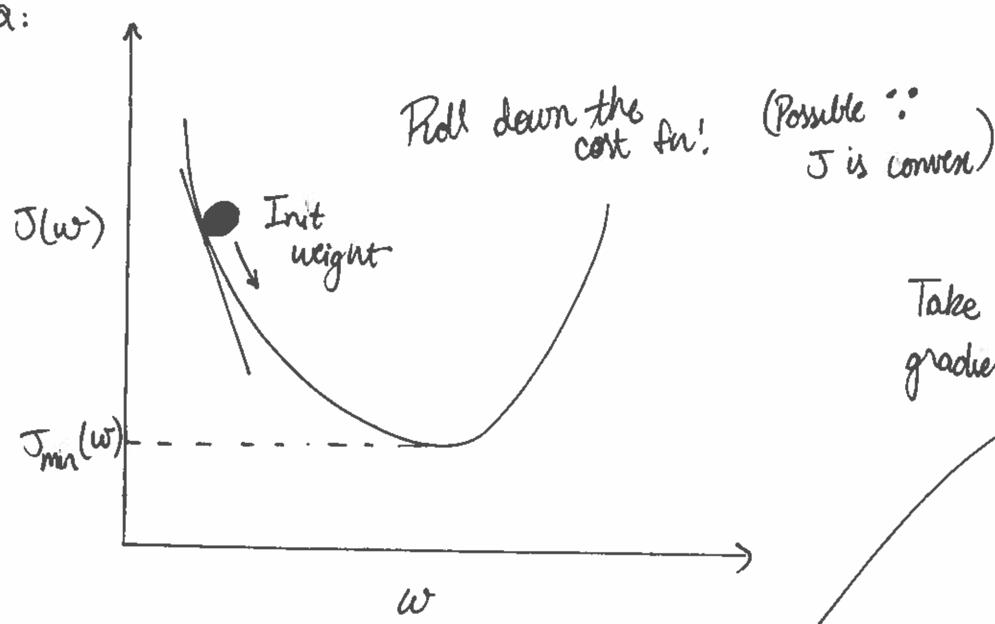
1960 - Widrow & Hoff propose Adaline (Adaptive Linear Neuron)



Continuous!
I.e., differentiable!

Define a cost fn. that we want to min.: $J(w) = \frac{1}{2} \sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$ ← Convenient when using grad!

Idea:



Take a step in opposite dir. of gradient: $\Delta w = -r \nabla J(\vec{w})$. ← η commonly used in lit

(Weight update)

$\nabla \Rightarrow$ gradient operator

In QM, $\nabla \cdot \nabla = \nabla^2 = \Delta$ ← Laplace operator.

(grad wrt. Cart coord.)

To do this: need to find $\Delta w_j = -r \dfrac{\partial J(\vec{w})}{\partial w_j}$

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\left(\frac{1}{2}\sum_i (d^{(i)} - y^{(i)})^2\right) = \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(d^{(i)} - y^{(i)})^2$$

$$= \frac{1}{2}\sum_i 2(d^{(i)} - y^{(i)}) \cdot \frac{\partial}{\partial w_j}(d^{(i)} - y^{(i)}) = \sum_i (d^{(i)} - y^{(i)}) \frac{\partial}{\partial w_j}\Big(\underbrace{d^{(i)}}_{\text{const}} - \sum_j w_j x_j^{(i)}\Big)$$

$$= \sum_i (d^{(i)} - y^{(i)})(-x_j^{(i)})$$

$$\Rightarrow \Delta w_j = -r \sum_i (d^{(i)} - y^{(i)})(-x_j^{(i)})$$

$$= r \sum_i (d^{(i)} - y^{(i)}) x_j^{(i)}$$

real number, not a class label!

$$\vec{w} := \vec{w} + \Delta \vec{w}$$

Update is performed for all samples at once!

Convergence:

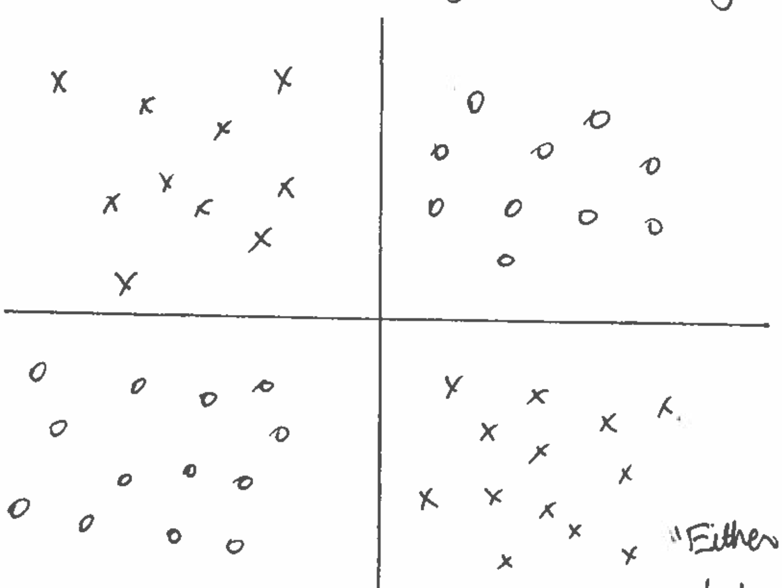
If training set D is not linearly separable $\Rightarrow$ perception <u>never</u> converges, and training fails <u>completely</u>!

If ~~it~~ D is linearly sep. $\Rightarrow$ guaranteed convergence! (Novikoff 1962)
Not only that, we can put an upper bound on the number of weight updates!.

$\Rightarrow$ Convergence guaranteed after $O\left(\frac{R^2}{\gamma^2}\right)$!

Note: convergence guaranteed for <u>some</u> separating hyperplane!
Best $\Rightarrow$ SVM! "Maximum margin hyperplane"
$\equiv$ "perception of optimal stability" Krauth & Mezard, 1987.

Famous example of non-convergence: Marvin Minsky & Seymour Papert, "Perceptrons", an Introduction to Computational Geometry" (1969)



"Either one, but not both / none"

Truth Table:

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$p \nLeftrightarrow q = (p \vee q) \wedge \neg (p \wedge q)$

In addition,

"It has many features to attract attention: its linearity, its intriguing learning theory, its clear pragmatic simplicity as a kind of parallel computation. Nevertheless, we consider it to be an important research problem to elucidate or reject our intuitive judgement that the extension to multilayer systems is sterile." – Minsky & Papert

Ie., MLP's suffer from same fate...

But, Rosenblatt had been experimenting w/ chaining together Perceptrons ⇒ Multi-layer perceptrons (since 1962!)

↳ ~~Sup~~ Surprising lack of citations: even Le Cunn et. al 2015 Nature doesn't cite.

1) No formal mathematical proof

2) Very, very hard to train (update weights) for MLPs!

Leads to AI Winter of 70's, 80's!

70's, 80's: Self Organizing Maps, Associate Memories

~~[illegible]~~

Breakthroughs:

1982: ~~Generalized~~ Hopfield Networks: explain associative memory using Stat Mech! → Later generalized to multiple layers

↳ Mathematical formalism!

1983: Kirkpatrick, Gelatt, Vechi: Simulated annealing

↳ Ackley, Hinton, Sejnowsky develop Boltzmann Machines by adapting SA ⇒ first successful realization of Multilayered N.N's!

1983: Barto, Sutton, Anderson popularize reinforcement learning.

↳ Source: Minsky's 1954 PhD dissertation!

1986: $\boxed{\text{Rumelhart, Hinton, Williams}}$

Backprop! "Backwards propagation of errors"

Generalization of Widrow & Hoff's Least Mean Square alg.

Use chain rule to propagate error through layers!

Evaluate expression for the derivative of the cost function as a product of derivatives between each layer from right to left ("back") ⇒ gradient of weights is modification of partial products.

For an input / output pair (x, y) & loss fn. L, for a MLP w/ l layers & activation fn. f:

$$\text{Loss} = L\left(y,\ f^l(W^l \cdot f^{l-1}(W^{l-1} \cdot f^{l-2}(W^{l-2} \ldots f^2(W^2 f^1(W^1 \cdot x)))\ldots)\right)$$

Start w/ x, then work forward: weighted input (result of dot prod) at each layer: z^l; activation produced: a^l

Need to store this + derivatives at z^l (i.e. $f^{l\prime}(z^l)$).

$\Rightarrow$ derivative of loss fn. wrt. inputs:

$$\frac{dL}{da^l} \cdot \frac{da^l}{dz^l} \cdot \frac{dz^l}{da^{l-1}} \cdot \frac{da^{l-1}}{dz^{l-1}} \cdots \frac{da^1}{dz^1} \cdot \frac{dz^1}{dx}$$

diagonal matrix

Note that these are not partial!

Hadamard product!

$$= \frac{dL}{a^l} \circ (f^l)' \cdot W^l \circ (f^{l-1})' \cdot W^{l-1} \circ \ldots \circ (f^1)' \cdot W^{l-1}$$

$\therefore$ Gradient is transpose of derivative of output in terms of input,

$$\nabla_x L = (W^1)^T \cdot (f^1)' \circ \ldots \circ (W^{l-1})^T \cdot (f^{l-1})' \circ (W^l)^T \cdot (f^l)' \circ \nabla_{a^l} L$$

By starting at $\nabla_{a^l} L$. $\Rightarrow$ multiply a vector by weight matrix & der. of activation fns. Forward op. $\Rightarrow$ matrix x matrix!