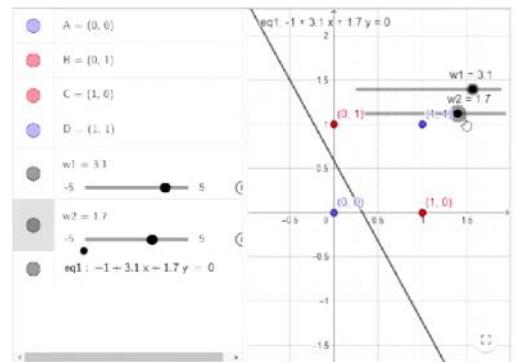


Linearly Separable Boolean Functions

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$
0	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
1	0	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$



$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \Rightarrow w_0 < 0$$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \Rightarrow w_2 \geq -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \Rightarrow w_1 \geq -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \Rightarrow w_1 + w_2 < -w_0$$

2 & 3 conditions contradict condition

∴ it's not linearly separable

Most real world data will not be linearly separable & will always contain outliers!

A single perceptron cannot deal with such data.

However, a network of perceptrons can indeed!

How many boolean functions can you design from 2 inputs?

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Only 2 functions f_7 & f_{10} are not linearly separable
 $(XOR)(\bar{XOR})$

So for n inputs we can have: 2^{2^n} boolean functions

How many are not linearly separable?

It's an unsolved problem!

Representation power of a network of perceptrons

We will see how to implement any boolean function using a network of perceptrons...

We will consider 2 inputs & 4 perceptrons False = -1

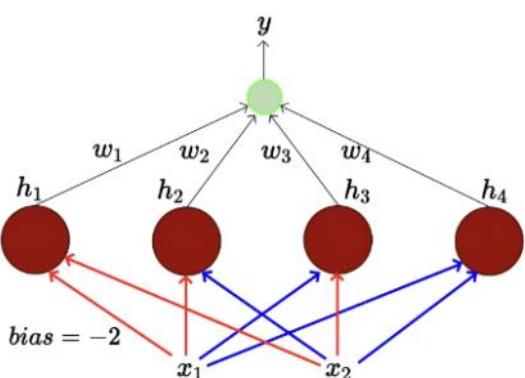
Each input is connected to all 4 perceptrons with specific weights

$$\text{red } w = -1, \quad \text{bias}(w_0) = -2$$

blue w = + :

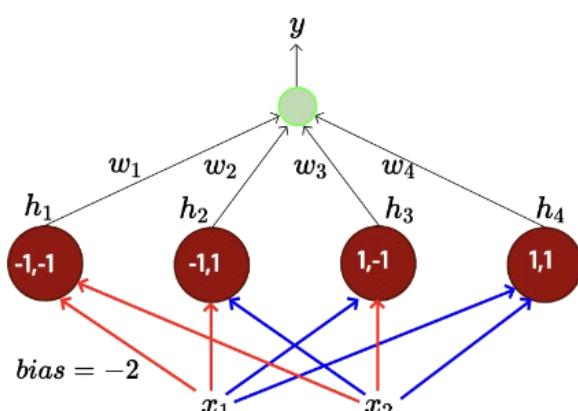
\therefore perceptron will only fire if weighted sum of input will be ≥ 2

Each perception is connected to an output perception by weights (which needs to be learned)



Terminology

- This network has 3 layers
 - The layer containing (x_1, x_2) is called the **input layer**
 - The middle layer containing the 4 perceptrons is called the **hidden layer**
 - The final layer is called the **output layer**
 - The outputs of the 4 perceptrons are denoted by h_1, h_2, h_3, h_4
 - Red & blue weights are called layer 1 weights
 w_1, w_2, w_3, w_4 are layer 2 weights



We claim that this network can be used to implement any boolean function (linearly separable or not)

In other words : we can find w_1, w_2, w_3 & w_4 such that the truth table can be represented by this network.

Each perceptron in the middle layer is gonna fire for a specific input.

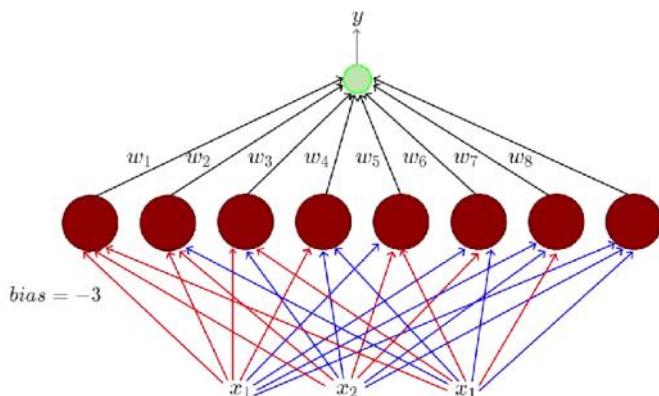
Let w_0 be the bias output of the neuron (i.e., it will fire if $\sum_{i=1}^4 w_i h_i \geq w_0$)

x_1	x_2	XOR	h_1	h_2	h_3	h_4	$\sum_{i=1}^4 w_i h_i$
0	0	0	1	0	0	0	w_1
0	1	1	0	1	0	0	w_2
1	0	1	0	0	1	0	w_3
1	1	0	0	0	1	1	w_4

This results in the following four conditions to implement XOR: $w_1 < w_0, w_2 \geq w_0, w_3 \geq w_0, w_4 < w_0$

These conditions are not contradicting so we can choose proper values for them!

For 3 inputs :



what about n inputs? 2^n perceptrons containing 1 hidden layer

A network of $2^n + 1$ perceptrons is not necessary but sufficient

Networks of the form are called Multilayered Perceptrons

[MLP]

... form we cannot implement recursive
[MLP]

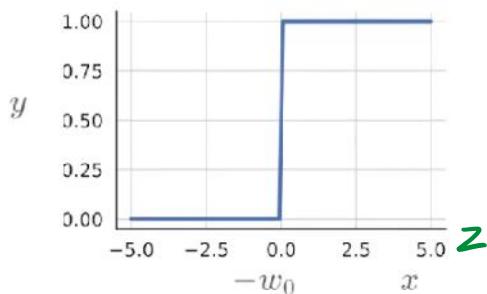
Sigmoid Neurons

We saw enough boolean functions!

What about arbitrary functions of the form: $y \in f(x)$
where $x \in \mathbb{R}^n$ & $y \in \mathbb{R}$?

Can we have a network which can (approximately) represent such functions?

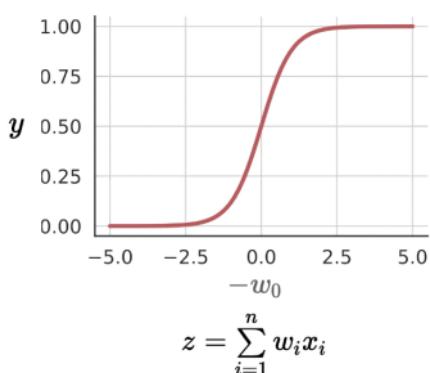
The threshold rule of perceptron is very harsh! It behaves like a step function



There is a sudden change in the decision (y) from 0 to 1 when we cross the threshold.

$$z = \sum_{i=1}^n w_i x_i$$

Introducing Sigmoid neurons where the output function is Smoother



Here is one form of sigmoid function called logistic function:

$$y = \frac{1}{1 + \exp(-(w_0 + \sum_{i=1}^n w_i x_i))}$$

As y is no longer binary but is a real value between 0 & 1.

As $w^T x \rightarrow \infty$

$$y = \frac{1}{1 + e^{-\infty}} = \frac{1}{1 + \frac{1}{e^{\infty}}} = \frac{1}{1+0} = 1$$

binary but is a real value b/w 0 & 1,
it can be interpreted as probability

$$y = \frac{1}{1+e^{-\infty}} = \frac{1}{1+\frac{1}{e^{\infty}}} = 1+0$$

As $w^T x \rightarrow -\infty$

$$y = \frac{1}{1+e^{\infty}} = \frac{1}{\infty} = 0$$

As $w^T x \rightarrow 0$

$$y = \frac{1}{1+e^0} = 0.5$$

Piecewise

$$y = 1 \text{ if } \sum_{i=0}^n w_i x_i \geq 0 \\ 0 \text{ if } \sum_{i=0}^n w_i x_i < 0$$

Not smooth,
not differentiable at $(-w_0)$

Sigmoid Neuron

$$y = \frac{1}{1 + \exp(-(\omega_0 + \sum_{i=1}^n w_i x_i))}$$

Smooth,
Differentiable

A typical Supervised Machine Learning Setup

Data: $\{x_i, y_i\}_{i=1}^n$ $n \left\{ \begin{bmatrix} & \\ & \end{bmatrix} \right| \underbrace{x \in \mathbb{R}^m}_{y \in \mathbb{R}}$

Model: Our approximation of the relation b/w x & y
eq: $y = \frac{1}{1 + e^{-w^T x}}$ or $\hat{y} = w^T x$ or $\hat{y} = x^T w$
 $\therefore y = \hat{f}(x)$

Parameters: In the above examples w is the parameters

Learning Algorithm: An algorithm for learning the parameters w of the model

Objective / Loss / Error Function: To guide the learning algorithm to minimize the loss function

Example: Movie Going

Data: $\{x_i = \text{movie}, y_i = (\text{like/dislike})\}_{i=1}^n$

Model: Our approximation relation b/w x & y (probability of liking a movie)

$$\hat{y} = \frac{1}{1 + e^{-w^T x}} \quad \hat{y} \rightarrow \text{approximation of } y$$

Parameters: w

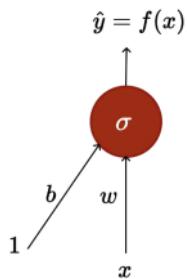
Learning Algorithm: Gradient descent

$$\text{Objective: } L(w) = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

$$\text{Objective: } L(\omega) = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

The learning algorithm should aim to find a ω which minimizes the above function

Learning Parameters: (Infeasible) guess work



This is the supervised learning setup we'll follow for learning parameters.

σ stands for sigmoid function
(logistic function in this case)

$$f(x) = \frac{1}{1+e^{-(wx+b)}}$$

There is only 1 input
b is bias (w_0)

Input: $\{(x_i, y_i)\}_{i=1}^N \rightarrow N$ pairs of (x, y)

Objective: Find w & b such that

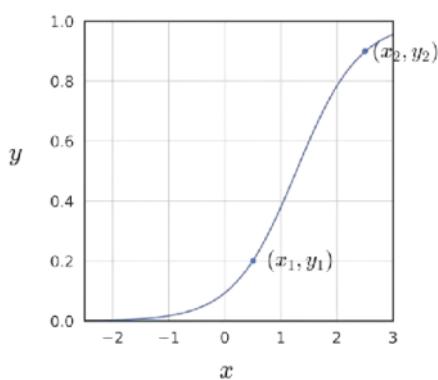
$$\underset{(w,b \in \mathbb{R})}{\text{minimize}} \quad L(w, b) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2$$

what does it mean to train the network?

Suppose we train with $(x, y) = (0.5, 0.2) \& (2.5, 0.9)$

At the end we are expected to find w^*, b^*
such that $f(0.5) \rightarrow 0.2$ & $f(2.5) \rightarrow 0.9$

$$f(x) = \frac{1}{1+e^{-(wx+b)}}$$

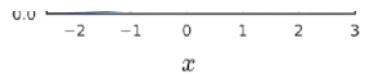


$$\text{Loss : } L(w, b) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2$$

$$= \frac{1}{2} \left[\{0.2 - f(0.5)\}^2 + \{0.9 - f(2.5)\}^2 \right]$$

For different values of w & b ,

w	b	Loss
0.50	0.00	0.0730
-0.10	0.00	0.14
0.94	-0.94	0.0214



w	b	Loss
0.50	0.00	0.0730
-0.10	0.00	0.14
0.94	-0.94	0.0214
1.42	-1.72	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

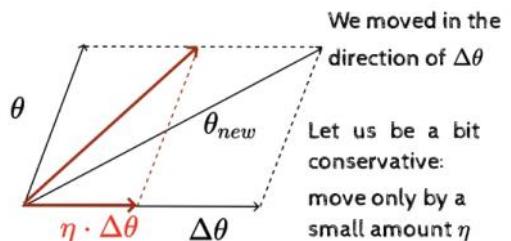
Taylor Series

Goal: Finding minimum error w/o brute force

Parameters: $\theta = [w, b]$

$$\Delta\theta = [\Delta w, \Delta b]$$

Change in parameters w, b



$$\theta_{new} = \theta + \eta \cdot \Delta\theta$$

what is the right $\Delta\theta$?
Answer comes from

Taylor Series

Taylor series is a way of approximating any continuous differentiable function $L(w)$ using polynomials of degree n . The higher the degree, the better the approximation.

$$L(w) = L(w_0) + \frac{L'(w_0)}{1!} (w - w_0) + \frac{L''(w_0)}{2!} (w - w_0)^2 + \frac{L'''(w_0)}{3!} (w - w_0)^3 + \dots$$

Linear approximation at $n=1$

$$L(w) = L(w_0) + \frac{L'(w_0)}{1!} (w - w_0)$$

Linear approximation at n=2

$$L(w) = L(w_0) + \frac{L'(w_0)}{1!} (w-w_0) + \frac{L''(w_0)}{2!} (w-w_0)^2$$

Gradient Descent

For ease of notation, let $\Delta\theta = u$, then from Taylor Series we have the following:

$$L(\theta + \eta u) = L(\theta) + \eta * u^T \nabla_{\theta} L(\theta) + \frac{\eta^2}{2!} * u^T \nabla_{\theta}^2 L(\theta) u + \dots$$

Gradient \rightarrow collection of partial derivatives

e.g.:- $\theta = w^2 + b^2$ Gradient of Gradient is Hessian Matrix

$$\begin{bmatrix} \frac{\partial \theta}{\partial w} \\ \frac{\partial \theta}{\partial b} \end{bmatrix} = \begin{bmatrix} 2w \\ 2b \end{bmatrix} \rightarrow \nabla_{\theta} L(\theta)$$

$$L(\theta + \eta u) = L(\theta) + \eta * u^T \nabla_{\theta} L(\theta)$$

η is typically small,
so $\eta^2, \eta^3 \rightarrow 0$

The move ηu would be favourable only if,

$$L(\theta) > L(\theta + \eta u)$$

or, $L(\theta + \eta u) - L(\theta) < 0$ [new loss is smaller]

implying: $u^T \nabla_{\theta} L(\theta) < 0$

what is it's range?

Let β be the angle b/w u^T & $\nabla_{\theta} L(\theta)$,

we know, $-1 \leq \cos(\beta) = \frac{u^T \cdot \nabla_{\theta} L(\theta)}{\|u^T\| * \|\nabla_{\theta} L(\theta)\|} \leq 1$

$\|u^T\| * \|\nabla_{\theta} L(\theta)\| \rightarrow k$ (let)

$$-k \leq k \cos \beta = u^T \cdot \nabla_{\theta} L(\theta) < 1$$

$$-k \leq k \cos \beta = u^T \nabla_{\theta} L(\theta) \leq k$$

Thus $L(\theta + \eta u) - L(\theta) = u^T \nabla_{\theta} L(\theta) = k * \cos(\beta)$
is the most negative when $\cos(\beta) = -1$
 $\therefore \beta = 180^\circ$

In other words, move to the opposite of the gradient

Parameter Updation Rule

$$w_{t+1} = w_t - \gamma \nabla w_t$$

$$b_{t+1} = b_t - \gamma \nabla b_t$$

$$\nabla w_t = \frac{\partial L(w, b)}{\partial L(w)} \text{ at } w=w_t, b=b_t$$

$$\nabla b_t = \frac{\partial L(w, b)}{\partial L(b)} \text{ at } w=w_t, b=b_t$$

Gradient Descent: Weight Update Rule

$$w_{t+1} = w_t - \eta \nabla w_t$$

$$b_{t+1} = b_t - \eta \nabla b_t$$

$$\nabla w_t = \frac{\partial L(w, b)}{\partial L(w)} \text{ at } w=w_t, b=b_t$$

$$\nabla b_t = \frac{\partial L(w, b)}{\partial L(b)} \text{ at } w=w_t, b=b_t$$

Algorithm

$t \leftarrow 0$

$\text{max_iterations} \leftarrow 1000$

$w, b \leftarrow \text{initialize randomly}$

while $t < \text{max_iterations}$ do

$$w_{t+1} \leftarrow w_t - \eta \nabla w_t$$

$$b_{t+1} \leftarrow b_t - \eta \nabla b_t$$

$$t \leftarrow t + 1$$

end

what exactly is ∇w_t ?
How to compute it?

For just one point,

$$\nabla w = \frac{\partial}{\partial w} \left[\frac{1}{2} * (f(x) - y)^2 \right]$$

$$= \frac{1}{2} \left[2 * (f(x) - y) * \frac{\partial}{\partial w} (f(x) - y) \right]$$

$$= (f(x) - y) * \frac{\partial}{\partial w} f(x)$$

for ... , \rightarrow

$$f(x) = \frac{1}{1 + \exp(-wx+b)}$$

y is a constant
& its partial derivative will be 0

$$= (f(x) - y) \times \frac{\partial}{\partial w} \left[\frac{1}{1 + e^{-(wx+b)}} \right] \quad \left. \begin{array}{l} \text{& it's partial} \\ \text{derivative will be 0} \end{array} \right\} \textcircled{1}$$

$$\frac{\partial}{\partial w} \left[\frac{1}{1 + e^{-(wx+b)}} \right] = \frac{1}{1 + e^{-(wx+b)}} \frac{\partial}{\partial w} (e^{-(wx+b)})$$

$$= \frac{-1}{1 + e^{-(wx+b)}} * e^{-(wx+b)} * \frac{\partial}{\partial w}(-(wx+b))$$

$$= \frac{-1}{1 + e^{-(wx+b)}} * \frac{e^{-(wx+b)}}{1 + e^{-(wx+b)}} * -x$$

$$= \frac{1}{1 + e^{-(wx+b)}} * \frac{e^{-(wx+b)}}{1 + e^{-(wx+b)}} * x$$

$$= f(x)(1-f(x))x$$

Substituting this in eqn ① we get,

$$\nabla w = (f(x) - y) * f(x) * (1-f(x)) * x$$

for one point.

For n points,

$$\nabla w = \sum_{i=1}^n [(f(x_i) - y_i) * f(x_i) * (1-f(x_i)) * x_i]$$

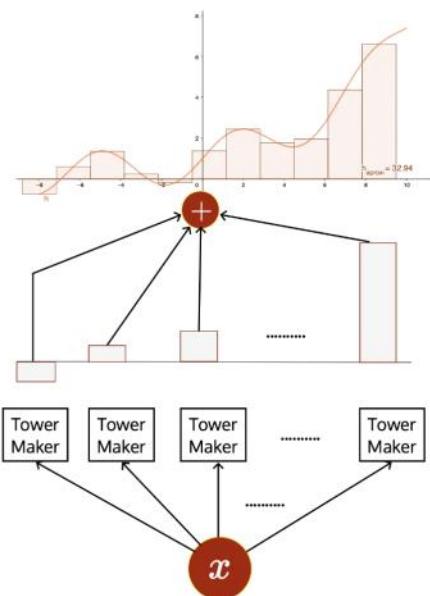
Similarly,

$$\nabla b = \sum_{i=1}^n [(f(x_i) - y_i) * f(x_i) * (1-f(x_i))]$$

Representation of Power of a Multilayer Network of Sigmoid Neurons

A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.

In other words, there is a guarantee that for any function $f(x): \mathbb{R}^n \rightarrow \mathbb{R}^m$, we can always find a neural network (with 1 hidden layer containing enough neurons) whose output $g(x)$ satisfies $|g(x) - f(x)| < \epsilon$



Any function can be represented by summing shifted and scaled versions of tower functions

Suppose there is a black box which takes the original input (x) and constructs these tower functions

We can then have a simple network which can just add them up to approximate the function

Our job now is to figure out what is inside this blackbox

