Perceptron Learning II

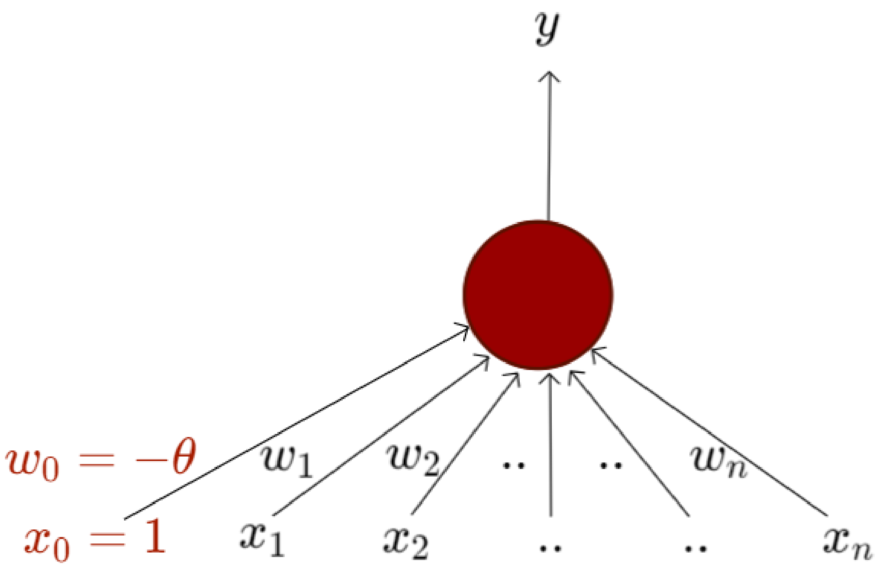
Last Lecture

- Perceptron Learning Algorithm

Today’s Topics

- Numerical Example
- Convergence theorem

Perceptron



A more accepted convention,

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

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

where, $x_0 = 1$ and $w_0 = -\theta$

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Perceptron Learning Algorithm

Algorithm: Perceptron Learning Algorithm

$P \leftarrow$ *inputs with label* 1;

$N \leftarrow$ *inputs with label* 0;

Initialize $\mathbf{w}$ randomly;

while $!convergence$ **do**

  Pick random $\mathbf{x} \in P \cup N$;

  if $\mathbf{x} \in \mathrm{P}$ *and* $\sum_{i=0}^{n} w_i * x_i < 0$ **then**

    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;

  end

  if $\mathbf{x} \in \mathrm{N}$ *and* $\sum_{i=0}^{n} w_i * x_i \geq 0$ **then**

    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;

  end

end

//the algorithm converges when all the inputs are classified correctly

- If the perceptron makes a **mistake (incorrect prediction)** and the **target = 1**, we **update weights as**: w=w+x
- If the perceptron makes a **mistake (incorrect prediction)** and the **target = 0**, we **update weights as**: w=w−x

Numerical Example: OR

Input (x1, x2)	Output (OR)
(0, 0)	0
(0, 1)	1
(1, 0)	1
(1, 1)	1

Parameters:

•Initial Weights:
$w1=0, w2=0, b=0$

•Threshold: 0

•Activation Function:
$y=1$ if $w1 \cdot x1+w2 \cdot x2+b)>=0$

$Y=0$ if $w1 \cdot x1+w2 \cdot x2+b) < 0$

- If the perceptron makes a **mistake (incorrect prediction)** and the **target = 1**, we **update weights as**:
  $w=w+x$
- If the perceptron makes a **mistake (incorrect prediction)** and the **target = 0**, we **update weights as**:
  $w=w-x$

Perceptron Example

Epoch 1:

1. Input (0, 0) -> Target = 0

- Net input = $0 \cdot 0 + 0 \cdot 0 + 0 = 0$
- Output = 1 (because net input ≥ 0)
- **Mistake** (output 1, target 0), so **update rule** $w = w - x$:
  - $w1 = 0 - 0 = 0$
  - $w2 = 0 - 0 = 0$
  - $b = 0 - 1 = -1$

2. Input (0, 1) -> Target = 1

- Net input = $0 \cdot 0 + 0 \cdot 1 + (-1) = -1$
- Output = 0 (because net input < 0)
- **Mistake** (output 0, target 1), so **update rule** $w = w + x$:
  - $w1 = 0 + 0 = 0$
  - $w2 = 0 + 1 = 1$
  - $b = -1 + 1 = 0$

Perceptron Example

Epoch 1:

Step	Input (x1, x2)	Target (t)	Net Input (w1*x1* + *w2*x2 + b)	Output (y)	Update Rule	Updated w1	Updated w2	Updated b
1	(0, 0)	0	$0 \cdot 0 + 0 \cdot 0 + 0 = 0$	1	$w = w - x$	0	0	-1
2	(0, 1)	1	$0 \cdot 0 + 0 \cdot 1 + (-1) = -1$	0	$w = w + x$	0	1	0
3	(1, 0)	1	$0 \cdot 1 + 1 \cdot 0 + 0 = 0$	1	No update	0	1	0
4	(1, 1)	1	$0 \cdot 1 + 1 \cdot 1 + 0 = 1$	1	No update	0	1	0

Perceptron Example

Epoch 2:

Step	Input (x1, x2)	Target (t)	Net Input (w1*x1* + *w2*x2 + b)	Output (y)	Update Rule	Updated w1	Updated w2	Updated b
1	(0, 0)	0	$0 \cdot 0 + 1 \cdot 0 + 0 = 0$	1	$w = w - x$	0	1	-1
2	(0, 1)	1	$0 \cdot 0 + 1 \cdot 1 + (-1) = 0$	1	No update	0	1	-1
3	(1, 0)	1	$0 \cdot 1 + 1 \cdot 0 + (-1) = -1$	0	$w = w + x$	1	1	0
4	(1, 1)	1	$1 \cdot 1 + 1 \cdot 1 + 0 = 2$	1	No update	1	1	0

Perceptron Example

Input (x1, x2)	Target (OR)	Net Input (w1 * x1 + w2 * x2 + b)	Output (y)	Correct?
(0, 0)	0	$1 \cdot 0 + 1 \cdot 0 + 0 = 0$	0	Yes
(0, 1)	1	$1 \cdot 0 + 1 \cdot 1 + 0 = 1$	1	Yes
(1, 0)	1	$1 \cdot 1 + 1 \cdot 0 + 0 = 1$	1	Yes
(1, 1)	1	$1 \cdot 1 + 1 \cdot 1 + 0 = 2$	1	Yes

Perceptron Learning on Iris Dataset (Binary Class)

- self.weights = np.zeros(n_features)
- self.bias = 0
- for _ in range(self.n_iters):
- for idx, x_i in enumerate(X):
    - linear_output = np.dot(x_i, self.weights) + self.bias
    - y_predicted = self.activation_function(linear_output)
    - update = self.learning_rate * (y[idx] - y_predicted)
    - self.weights += update * x_i
    - self.bias += update

Proof of Convergence

Proof of Convergence

Theorem

Definition: Two sets P and N of points in an n-dimensional space are called absolutely linearly separable if $n + 1$ real numbers $w_0, w_1, ..., w_n$ exist such that every point $(x_1, x_2, ..., x_n) \in P$ satisfies $\sum_{i=1}^{n} w_i * x_i > w_0$ and every point $(x_1, x_2, ..., x_n) \in N$ satisfies $\sum_{i=1}^{n} w_i * x_i < w_0$.

Proposition: If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector $\mathbf{w}_t$ a finite number of times. In other words: if the vectors in P and N are tested cyclically one after the other, a weight vector $\mathbf{w}_t$ is found after a finite number of steps t which can separate the two sets.

Proof of Convergence

- If $x \in N$ then -$x \in P$ ($\because w^T x < 0 \implies w^T(-x) \geq 0$)
- We can thus consider a single set $P' = P \cup N^-$ and for every element $p \in P'$ ensure that $w^T p \geq 0$
- Further we will normalize all the p's so that $||p|| = 1$ (notice that this does not affect the solution $\because if \quad w^T \frac{p}{||p||} \geq 0$ then $w^T p \geq 0$)

Algorithm: Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
N^- *contains negations of all points in* N;
$P' \leftarrow P \cup N^-;$
Initialize $\mathbf{w}$ randomly;
while !*convergence* **do**
    Pick random $\mathbf{p} \in P'$;
    $\mathbf{p} \leftarrow \frac{p}{||p||} \quad (so\ now, ||p|| = 1)$;
    if $\mathbf{w}.\mathbf{p} < 0$ **then**
        $\mathbf{w} = \mathbf{w} + \mathbf{p}$;
    end
end

Proof of Convergence

- w^* is some optimal solution which exists but we don't know what it is

- We make a correction only if $w^T \cdot p_i \leq 0$ at that time step

Proof of Convergence

Proof:

- Now suppose at time step t we inspected the point p_i and found that $w^T \cdot p_i \leq 0$
- We make a correction $w_{t+1} = w_t + p_i$
- Let β be the angle between w^* and w_{t+1}

$$cos\beta = \frac{w^* \cdot w_{t+1}}{||w_{t+1}||}$$

$$\begin{aligned} Numerator &= w^* \cdot w_{t+1} = w^* \cdot (w_t + p_i) \\ &= w^* \cdot w_t + w^* \cdot p_i \\ &\geq w^* \cdot w_t + \delta \quad (\delta = min\{w^* \cdot p_i | \forall i \\ &\geq w^* \cdot (w_{t-1} + p_j) + \delta \\ &\geq w^* \cdot w_{t-1} + w^* \cdot p_j + \delta \\ &\geq w^* \cdot w_{t-1} + 2\delta \\ &\geq w^* \cdot w_0 + (k)\delta \quad (By\ induction) \end{aligned}$$

- We make a correction only if $w^T \cdot p_i \leq 0$ at that time step
- So at time-step t we would have made only k ($\leq t$) corrections
- Every time we make a correction a quantity δ gets added to the numerator
- So by time-step t, a quantity $k\delta$ gets added to the numerator

Proof of Convergence

Proof (continued:)

$$So\ far\ we\ have, \quad w^T \cdot p_i \leq 0 \quad (and\ hence\ we\ made\ the\ correction)$$

$$\begin{aligned}
cos\beta &= \frac{w^* \cdot w_{t+1}}{||w_{t+1}||} \quad (by\ definition) \\
Numerator &\geq w^* \cdot w_0 + k\delta \quad (proved\ by\ induction) \\
Denominator^2 &= ||w_{t+1}||^2 \\
&= (w_t + p_i) \cdot (w_t + p_i) \\
&= ||w_t||^2 + 2w_t \cdot p_i + ||p_i||^2) \\
&\leq ||w_t||^2 + ||p_i||^2 \quad (\because w_t \cdot p_i \leq 0) \\
&\leq ||w_t||^2 + 1 \quad (\because ||p_i||^2 = 1) \\
&\leq (||w_{t-1}||^2 + 1) + 1 \\
&\leq ||w_{t-1}||^2 + 2 \\
&\leq ||w_0||^2 + (k) \quad (By\ same\ observation\ that\ we\ made\ about\ \delta)
\end{aligned}$$

Proof of Convergence

Proof (continued:)

So far we have, $w^T \cdot p_i \leq 0$ *(and hence we made the correction)*

$$cos\beta = \frac{w^* \cdot w_{t+1}}{||w_{t+1}||} \quad (by\ definition)$$

$$Numerator \geq w^* \cdot w_0 + k\delta \quad (proved\ by\ induction)$$

$$Denominator^2 \leq ||w_0||^2 + k \quad (By\ same\ observation\ that\ we\ made\ about\ \delta)$$

$$cos\beta \geq \frac{w^* \cdot w_0 + k\delta}{\sqrt{||w_0||^2 + k}}$$

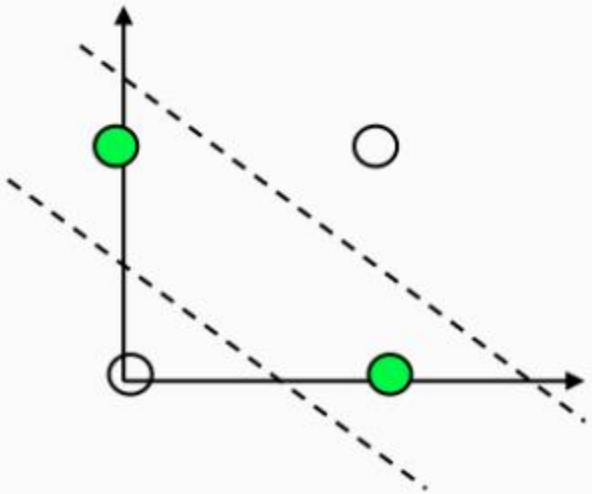
- $cos\beta$ thus grows proportional to $\sqrt{k}$
- As k (number of corrections) increases $cos\beta$ can become arbitrarily large
- But since $cos\beta \leq 1$, k must be bounded by a maximum number
- Thus, there can only be a finite number of corrections (k) to w and the algorithm will converge!

Multi-layer Perceptrons

Perceptron Limitations

For a linearly not-separable problem:

- Would it help if we use more layers of neurons?
- What could be the learning rule for each neuron?



Boolean XOR

Solution: Multilayer networks and the backpropagation learning algorithm

Boolean functions 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
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	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

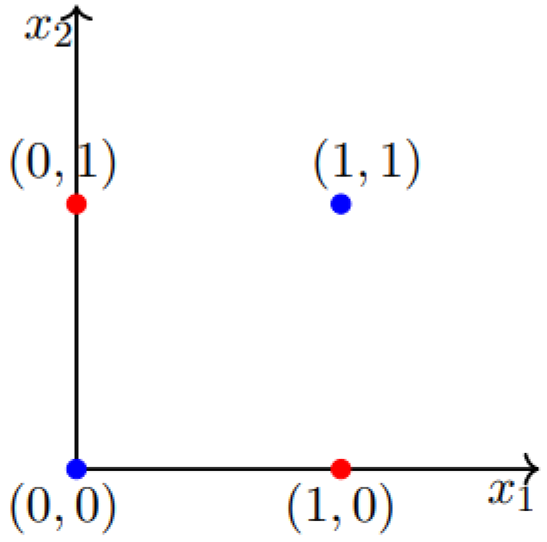
- Of these, how many are linearly separable ? (turns out all except XOR and !XOR - feel free to verify)

what do we do about functions which are not linearly separable ?

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

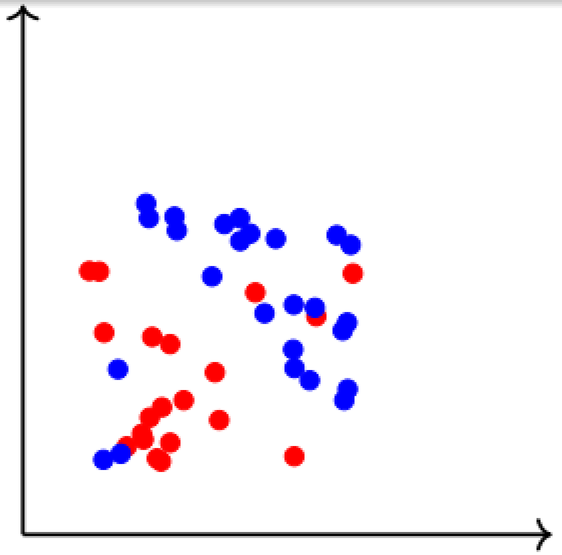
$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 \geq -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 \geq -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$

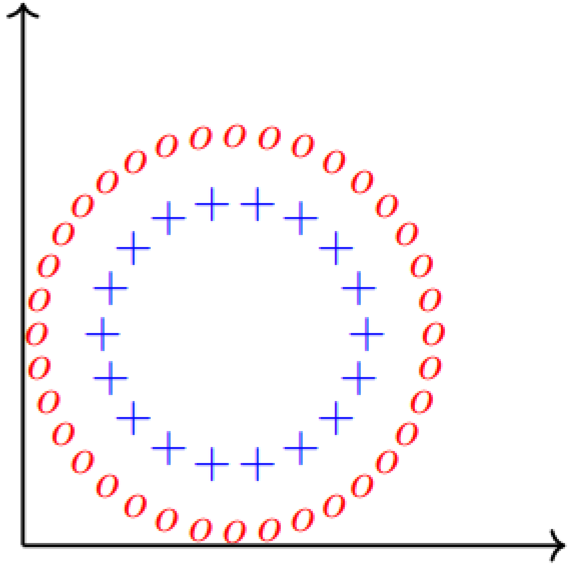
- The fourth condition contradicts conditions 2 and 3
- Hence we cannot have a solution to this set of inequalities



- And indeed you can see that it is impossible to draw a line which separates the red points from the blue points

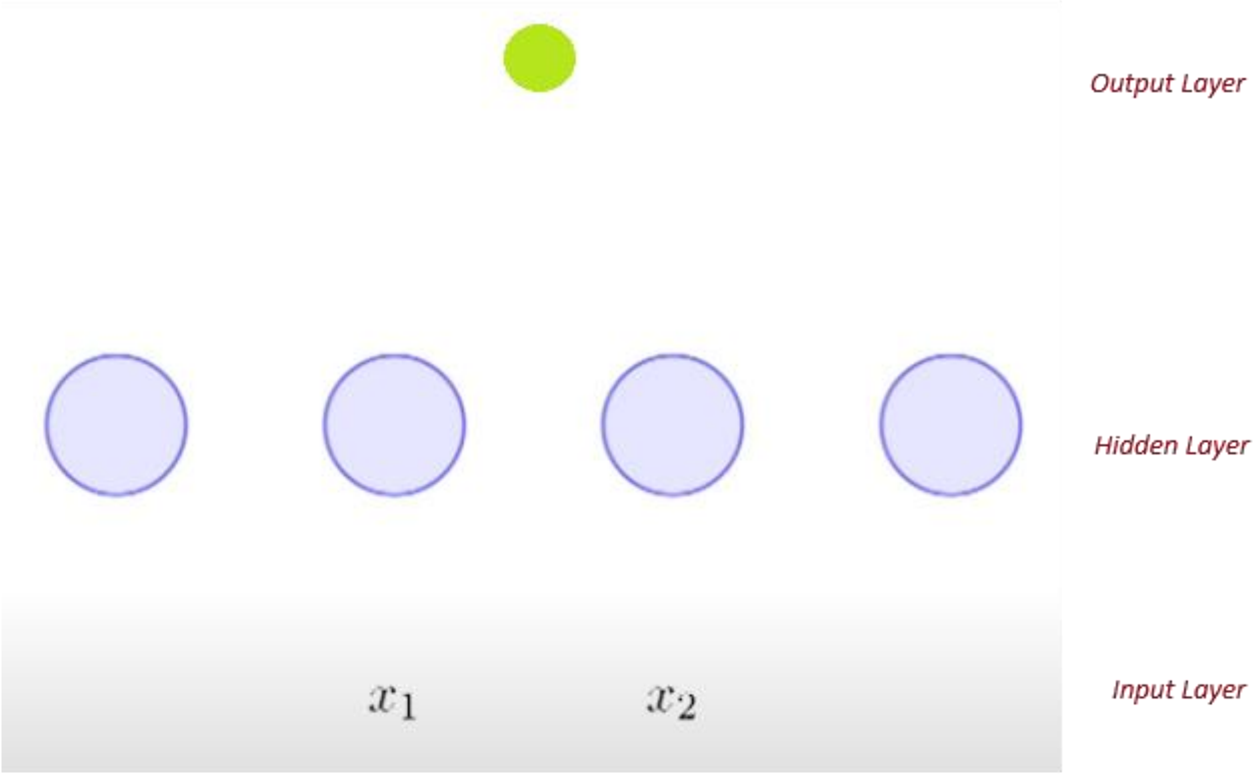
Non-linearly separable data





- Most real world data is not linearly separable and will always contain some outliers
- In fact, sometimes there may not be any outliers but still the data may not be linearly separable
- We need computational units (models) which can deal with such data
- While a single perceptron cannot deal with such data, we will show that a network of perceptrons can indeed deal with such data

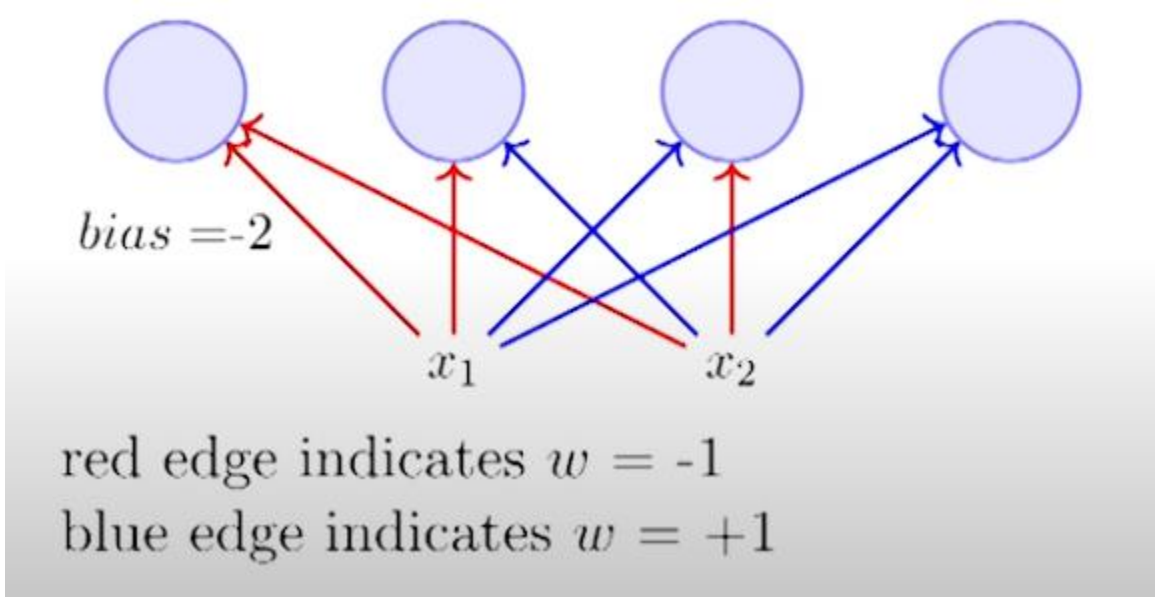
Network of Perceptrons

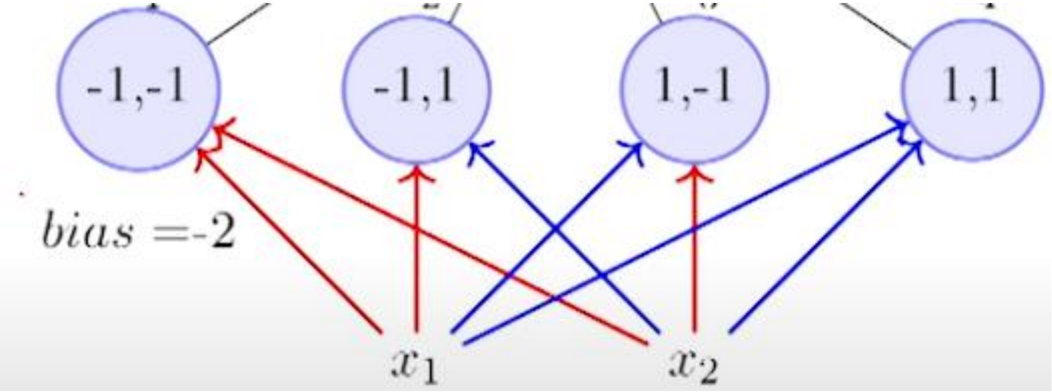


- This network contains 3 layers
- The layer containing the inputs (x_1, x_2) is called the **input layer**
- The middle layer containing the 4 perceptrons is called the **hidden layer**
- The final layer containing one output neuron is called the **output layer**

Source: Prof. Mithesh Khapra Deep Learning Course

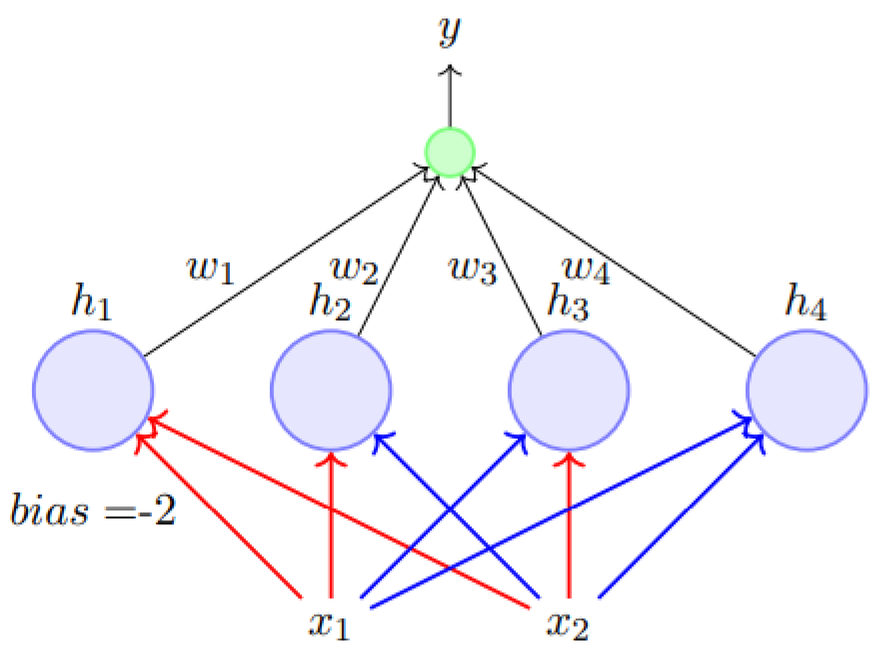
Network of Perceptrons





Source: Prof. Mithesh Khapra Deep Learning Course

Network of Perceptrons

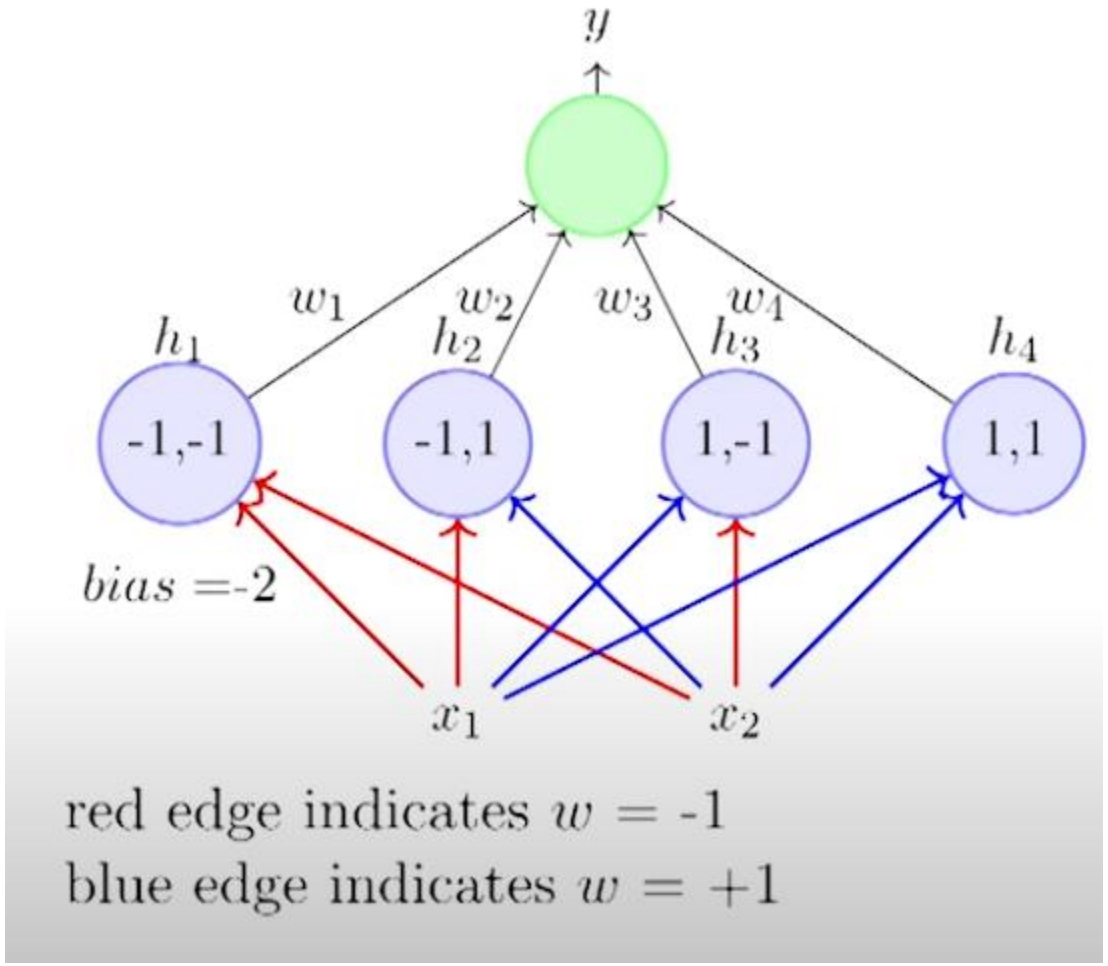


red edge indicates w = -1
blue edge indicates $w = +1$

- This network contains 3 layers
- The layer containing the inputs (x_1, x_2) is called the **input layer**
- The middle layer containing the 4 perceptrons is called the **hidden layer**
- The final layer containing one output neuron is called the **output layer**
- The outputs of the 4 perceptrons in the hidden layer are denoted by h_1, h_2, h_3, h_4
- The red and blue edges are called layer 1 weights
- w_1, w_2, w_3, w_4 are called layer 2 weights

Source: Prof. Mithesh Khapra Deep Learning Course

Network of Perceptrons

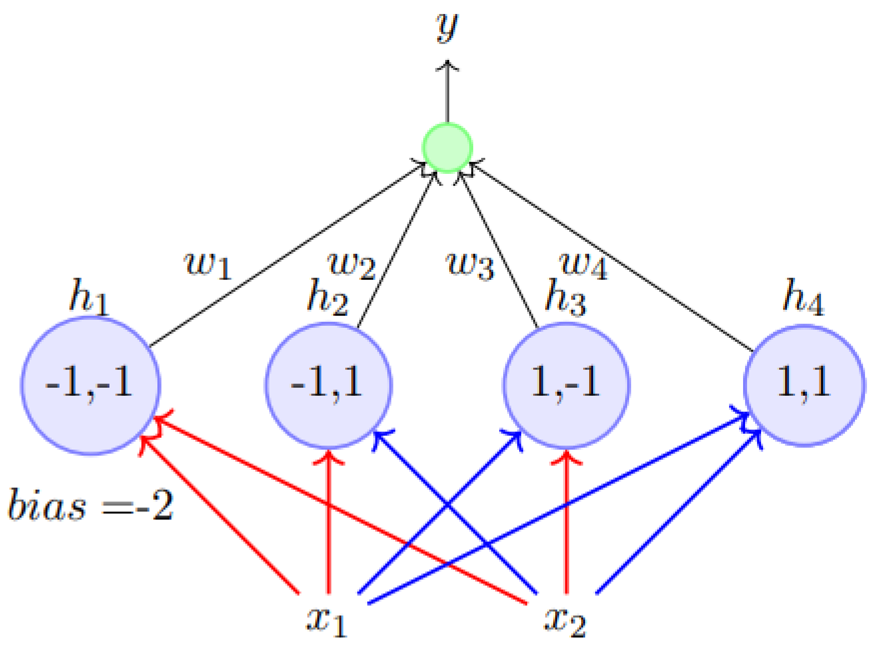


- Let w_0 be the bias output of the neuron (*i.e.*, it will fire if $\sum_{i=1}^{4} w_i h_i \geq w_0$)

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

Source: Prof. Mithesh Khapra Deep Learning Course

Network of Perceptrons



red edge indicates w = -1
blue edge indicates w = +1

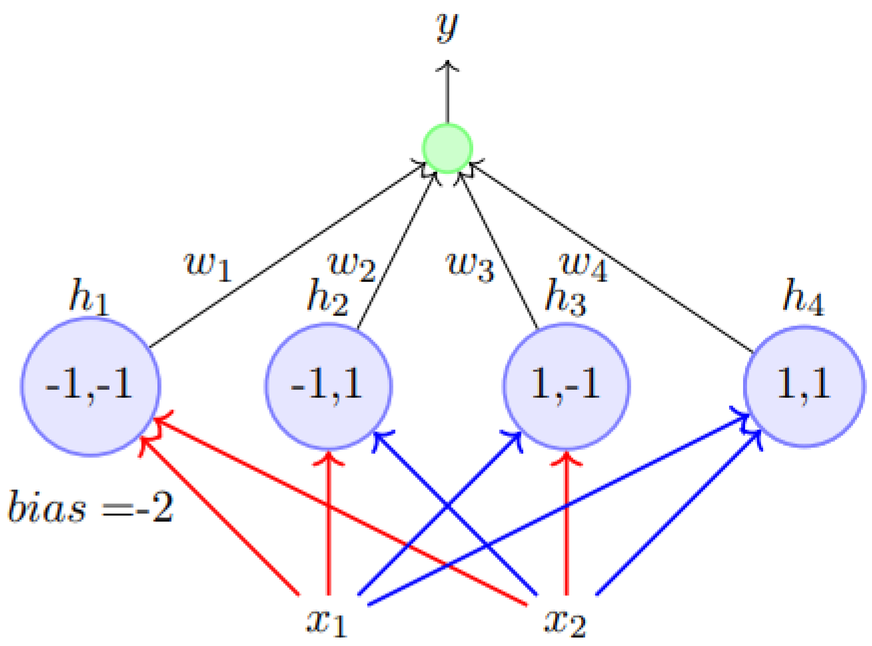
- Let w_0 be the bias output of the neuron (*i.e.*, it will fire if $\sum_{i=1}^{4} w_i h_i \geq w_0$)

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	0	1	w_4

- This results in the following four conditions to implement XOR: $w_1 < w_0, w_2 \geq w_0, w_3 \geq w_0, w_4 < w_0$
- Unlike before, there are no contradictions now and the system of inequalities can be satisfied
- Essentially each w_i is now responsible for one of the 4 possible inputs and can be adjusted to get the desired output for that input

Source: Prof. Mithesh Khapra Deep Learning Course

Network of Perceptrons



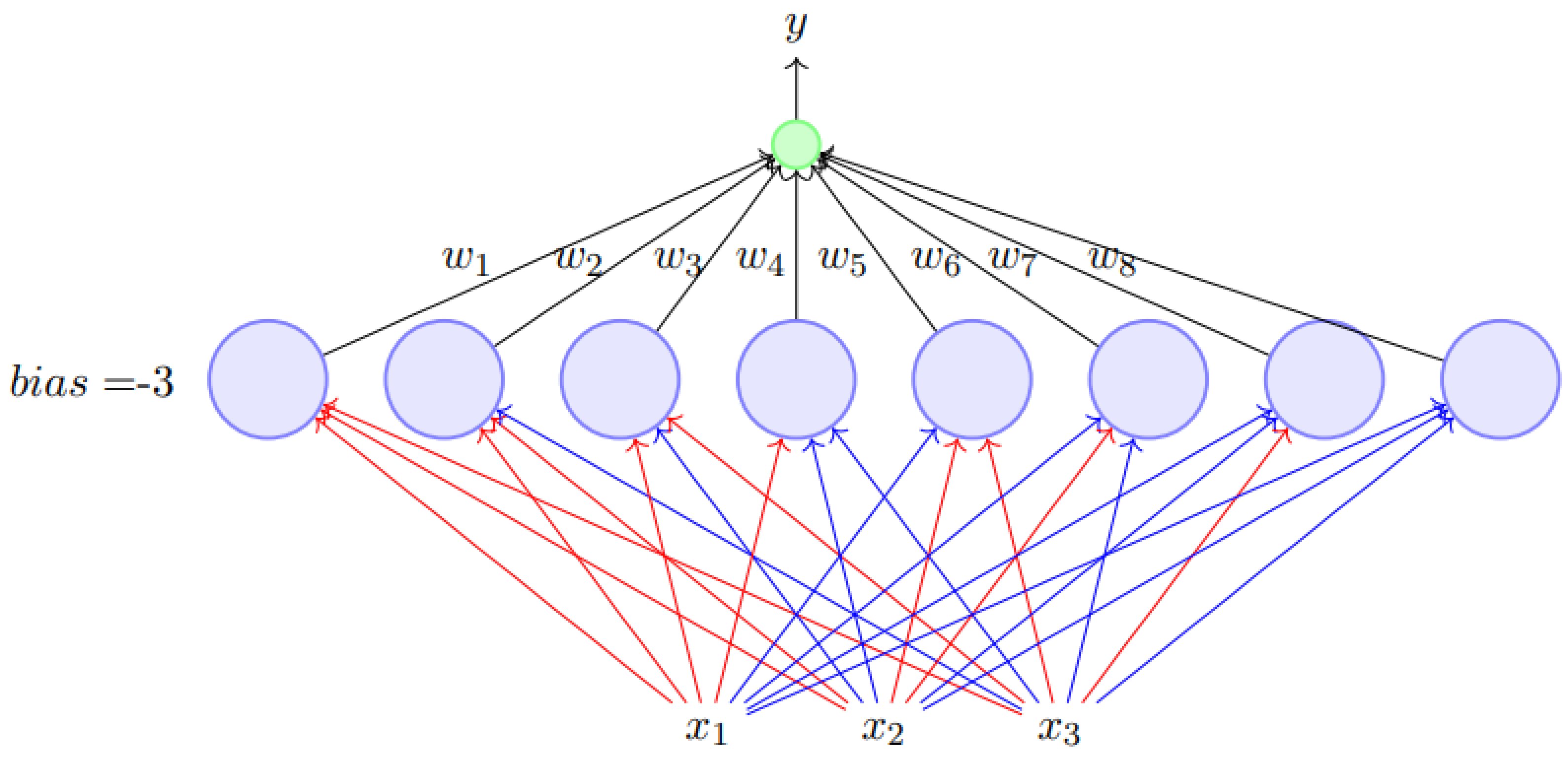
red edge indicates w = -1
blue edge indicates $w = +1$

- It should be clear that the same network can be used to represent the remaining 15 boolean functions also
- Each boolean function will result in a different set of non-contradicting inequalities which can be satisfied by appropriately setting w_1, w_2, w_3, w_4

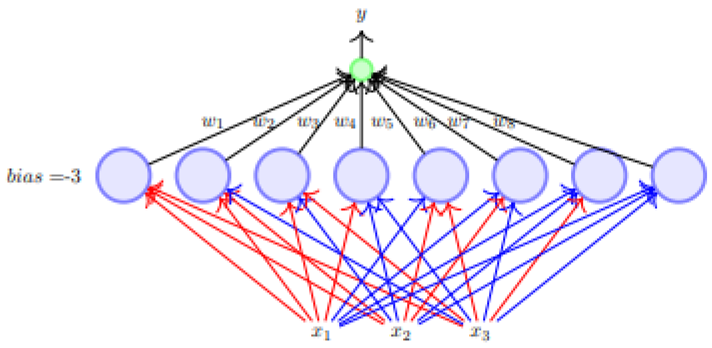
Source: Prof. Mithesh Khapra Deep Learning Course

more than 3 inputs

- Again each of the 8 perceptorns will fire only for one of the 8 inputs
- Each of the 8 weights in the second layer is responsible for one of the 8 inputs and can be adjusted to produce the desired output for that input



How to solve real world problem?



$$
\begin{array}{c}
p_1 \\ p_2 \\ \vdots \\ n_1 \\ n_2 \\ \vdots
\end{array}
\begin{bmatrix}
x_{11} & x_{12} & \dots & x_{1n} & y_1 = 1 \\
x_{21} & x_{22} & \dots & x_{2n} & y_2 = 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
x_{k1} & x_{k2} & \dots & x_{kn} & y_i = 0 \\
x_{j1} & x_{j2} & \dots & x_{jn} & y_j = 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots
\end{bmatrix}
$$

n inputs

Theorem

Any boolean function of n inputs can be represented exactly by a network of perceptrons containing 1 hidden layer with 2^n perceptrons and one output layer containing 1 perceptron

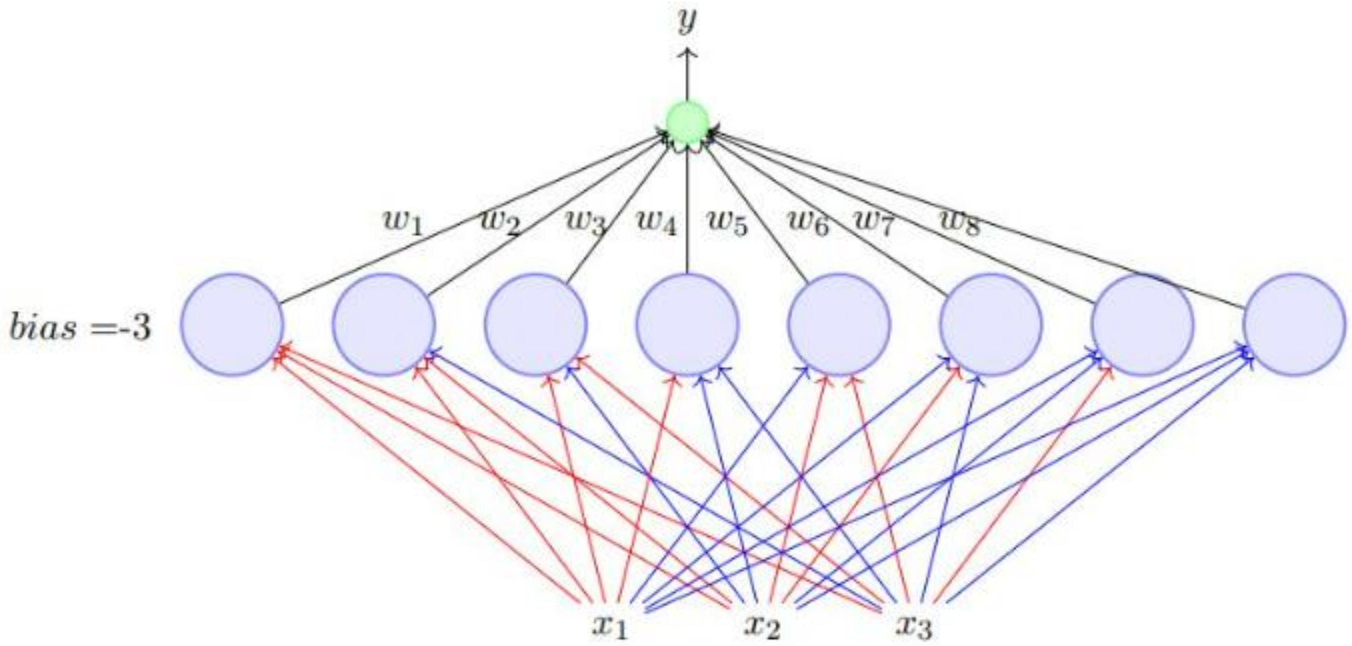
Proof (informal:) We just saw how to construct such a network

Note: A network of $2^n + 1$ perceptrons is not necessary but sufficient. For example, we already saw how to represent AND function with just 1 perceptron

Catch: As n increases the number of perceptrons in the hidden layers obviously increases exponentially

Last Lecture

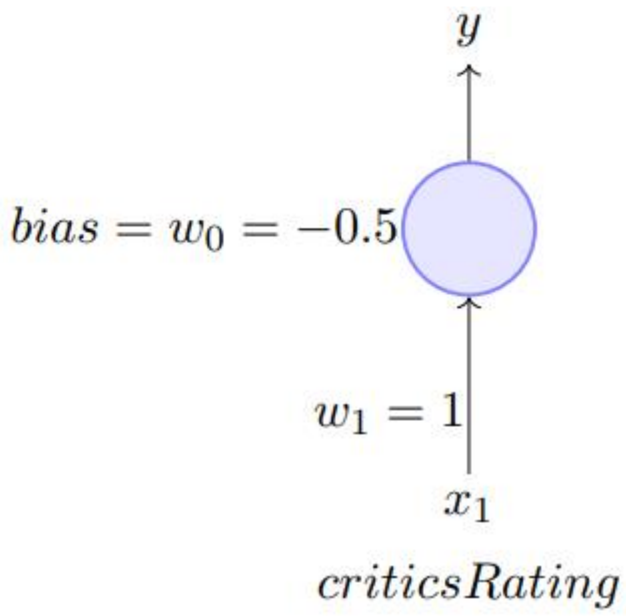
- MLP
- Any boolean function can be represented using an MLP



Sigmoid Neuron

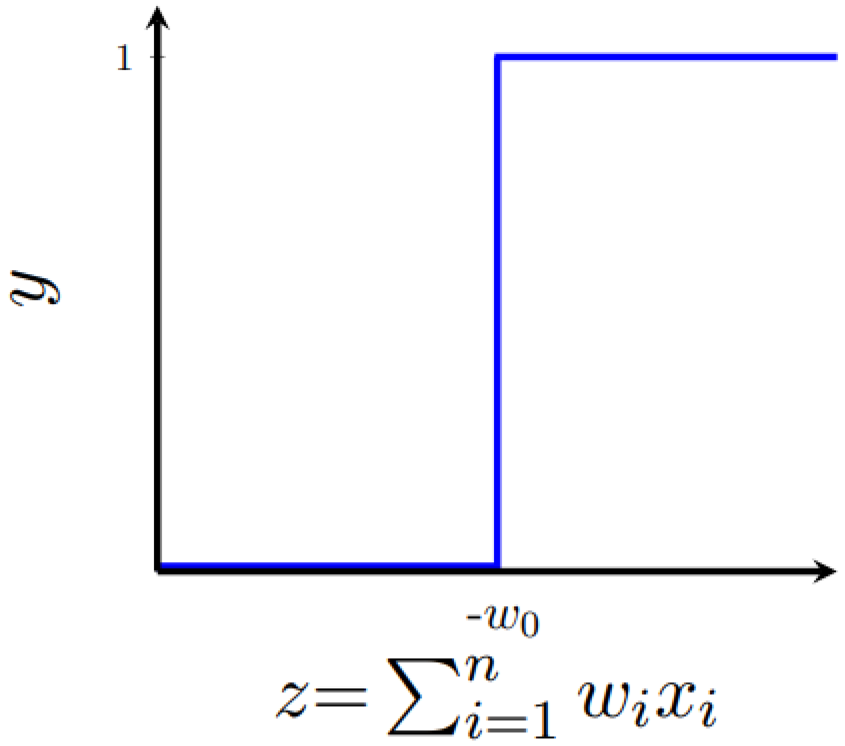
- Negatives of thresholding logic in perceptron
- How sigmoid neuron overcomes this limitation?
- Learning algorithm of sigmoid neuron
- How sigmoid neurons represents arbitrary functions?

Thresholding Logic of Perceptron



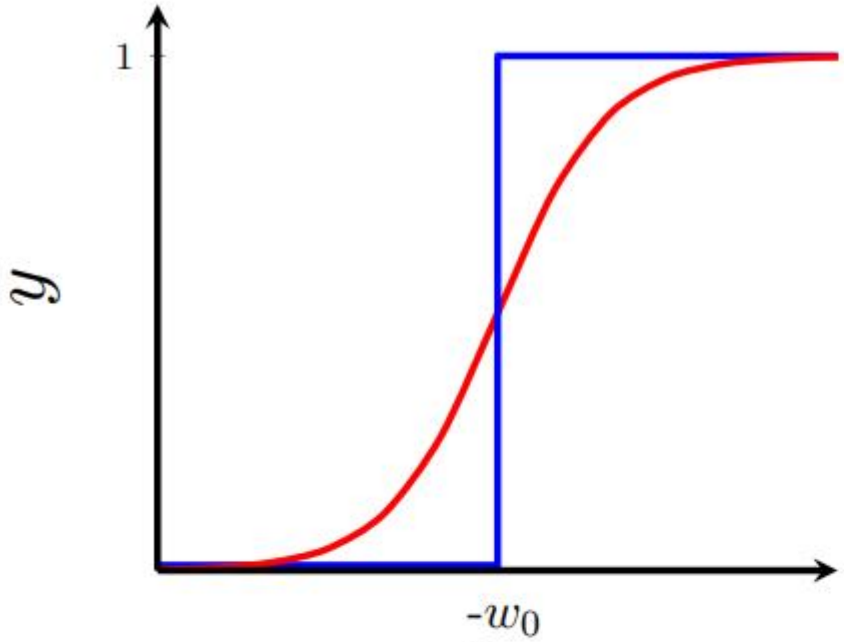
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) and $w_1 = 1$ then what would be the decision for a movie with $criticsRating = 0.51$? (like)
- What about a movie with $criticsRating = 0.49$? (dislike)
- It seems harsh that we would like a movie with rating 0.51 but not one with a rating of 0.49

Thresholding Logic of Perceptron



- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function
- There will always be this sudden change in the decision (from 0 to 1) when $\sum_{i=1}^{n} w_i x_i$ crosses the threshold ($-w_0$)
- For most real world applications we would expect a smoother decision function which gradually changes from 0 to 1

Sigmoid Neuron



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

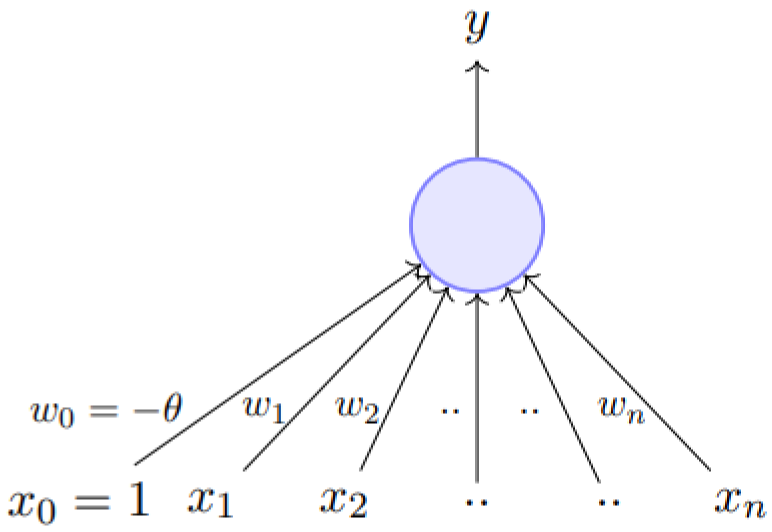
- Introducing sigmoid neurons where the output function is much smoother than the step function
- Here is one form of the sigmoid function called the logistic function

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

- We no longer see a sharp transition around the threshold -w_0
- Also the output y is no longer binary but a real value between 0 and 1 which can be interpreted as a probability
- Instead of a like/dislike decision we get the probability of liking the movie

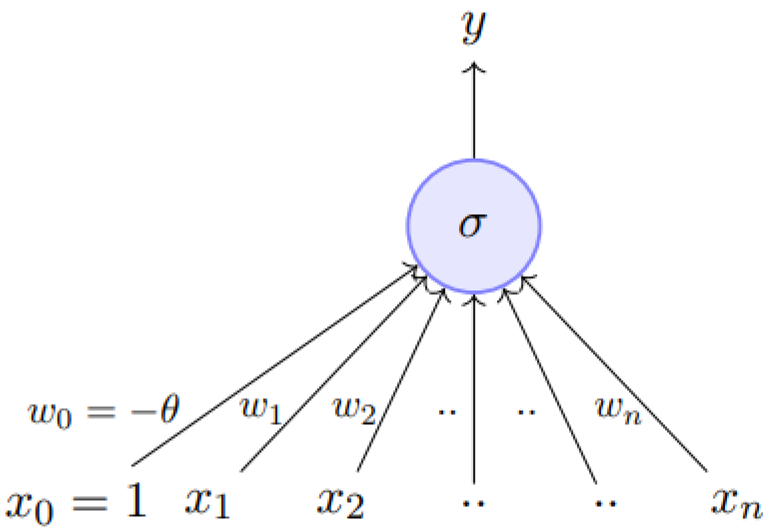
Perceptron Vs Sigmoid

Perceptron



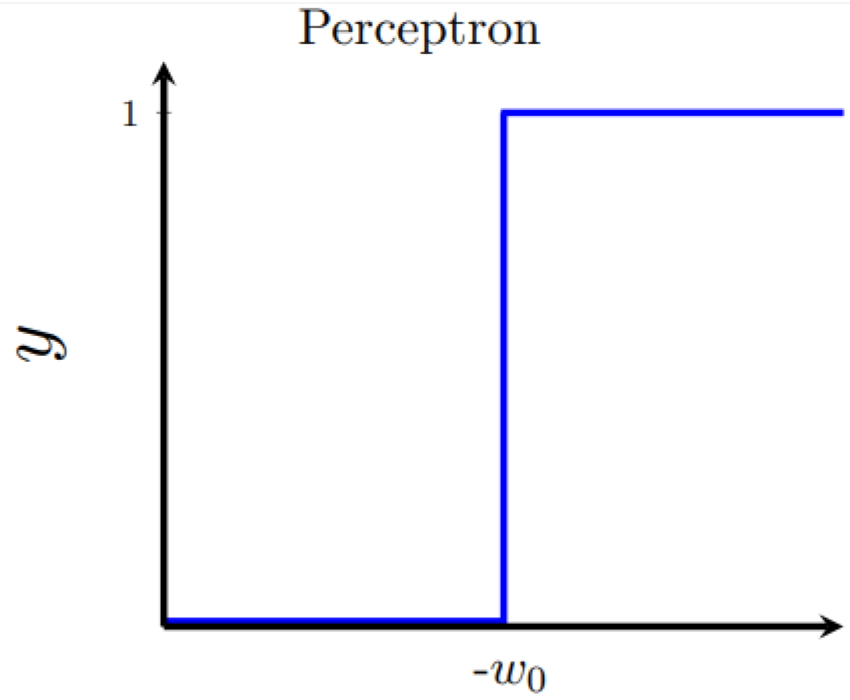
$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Sigmoid (logistic) Neuron



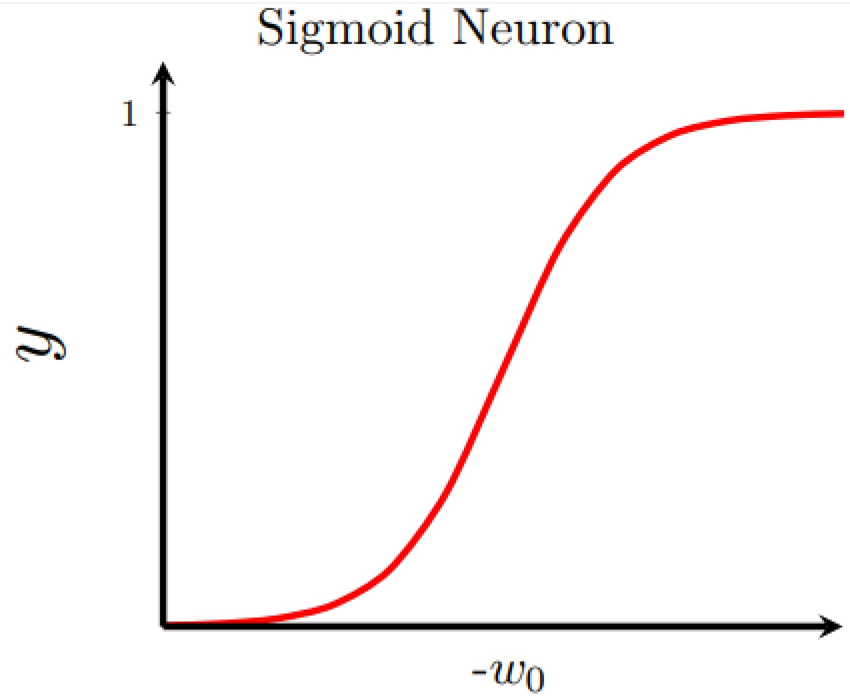
$$y = \frac{1}{1 + e^{-(\sum_{i=0}^{n} w_i x_i)}}$$

Perceptron Vs Sigmoid



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

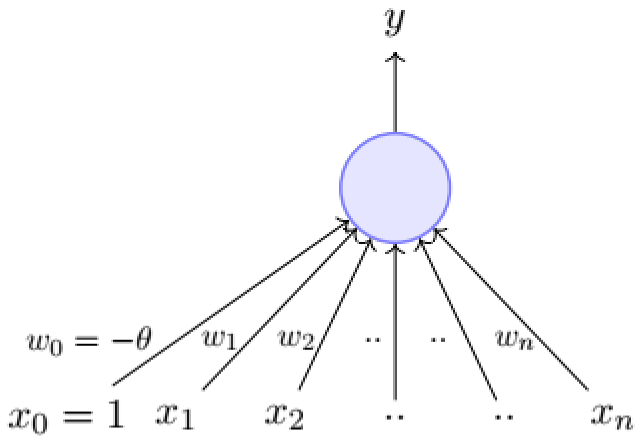
Not smooth, not continuous (at $w0$), **not differentiable**



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

Smooth, continuous, **differentiable**

How do we learn weights of Sigmoid Neuron?



Learning Setup

- **Data:** $\{x_i, y_i\}_{i=1}^{n}$
- **Model:** Our approximation of the relation between $\mathbf{x}$ and y. For example,

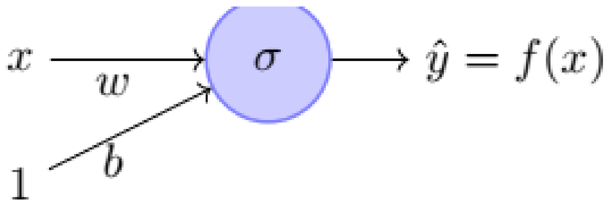
$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w^T x})}}$$

$$or \quad \hat{y} = \mathbf{w^T x}$$

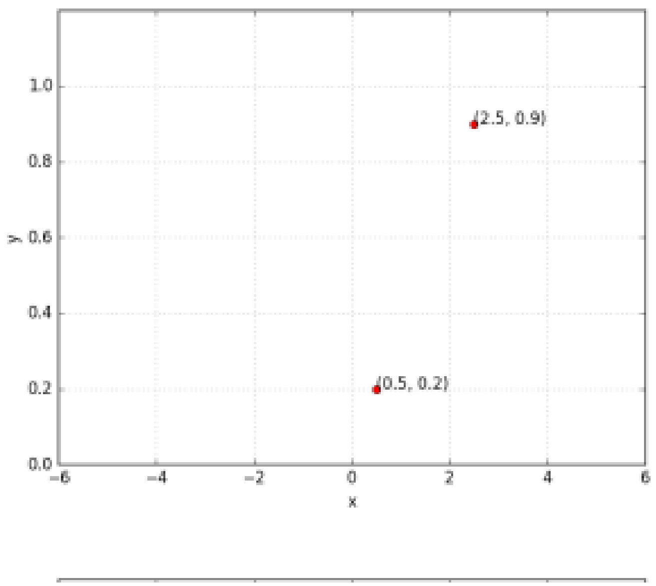
$$or \quad \hat{y} = \mathbf{x^T W x}$$

  or just about any function
- **Parameters:** In all the above cases, w is a parameter which needs to be learned from the data
- **Learning algorithm:** An algorithm for learning the parameters (w) of the model (for example, perceptron learning algorithm, gradient descent, etc.)
- **Objective/Loss/Error function:** To guide the learning algorithm - the learning algorithm should aim to minimize the loss function

Example



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



Input for training

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

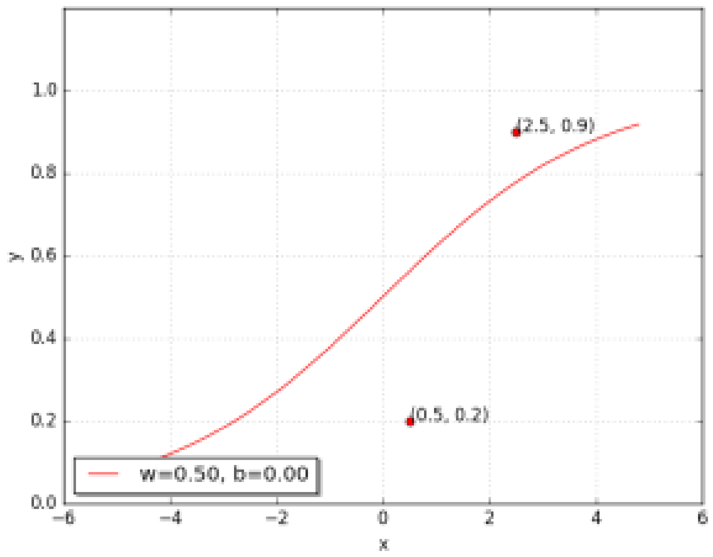
Training objective

Find w and b such that:

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

What does it mean to train the network?

- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$
- At the end of training we expect to find w*, b* such that:
- $f(0.5) \rightarrow 0.2$ and $f(2.5) \rightarrow 0.9$

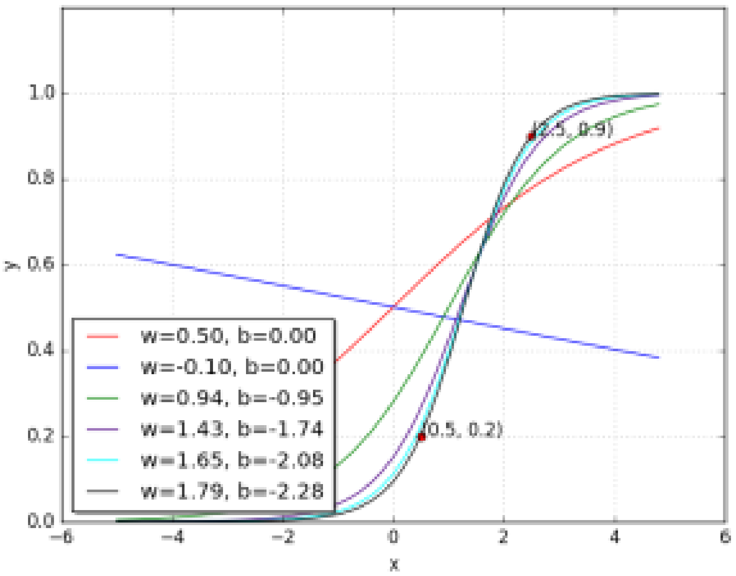


$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

- Can we try to find such a w^*, b^* manually
- Let us try a random guess.. (say, $w = 0.5, b = 0$)
- Clearly not good, but how bad is it ?
- Let us revisit $\mathscr{L}(w, b)$ to see how bad it is ...

$$\begin{aligned}
\mathscr{L}(w, b) &= \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2 \\
&= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 \\
&= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2 \\
&= 0.073
\end{aligned}$$

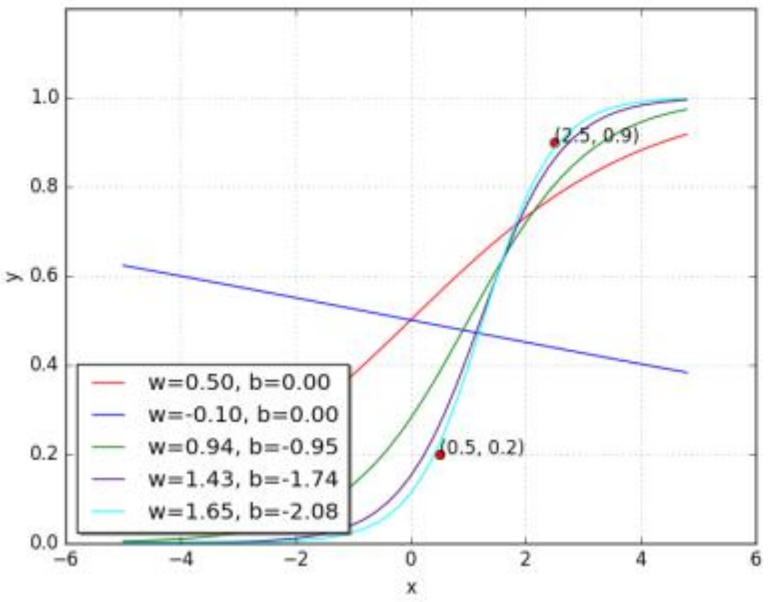
We want $\mathscr{L}(w, b)$ to be as close to 0 as possible

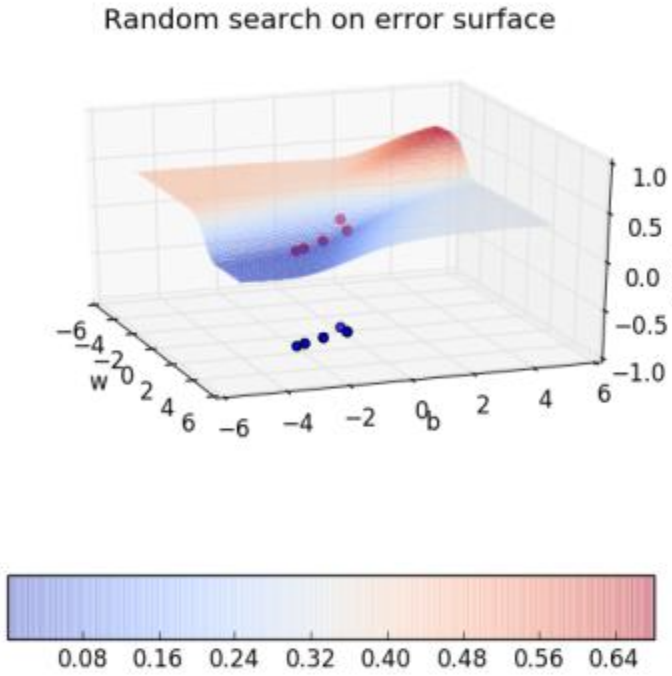


Let us try some other values of w, b

w	b	$\mathscr{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

Error Surface





Gradient Descent

Goal

Find a better way of traversing the error surface so that we can reach the minimum value quickly without resorting to brute force search!

Algorithm: gradient_descent()

$t \leftarrow 0;$

$max\_iterations \leftarrow 1000;$

while $t < max\_iterations$ **do**

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

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

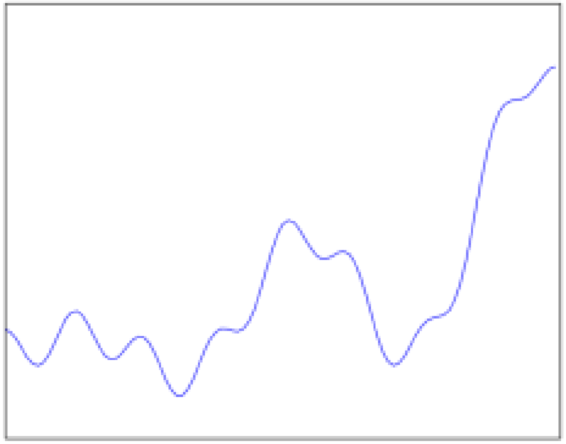
  $t \leftarrow t + 1;$

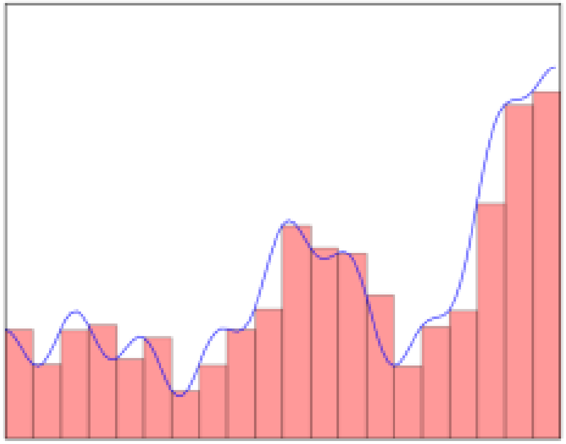
end

Multi Layer Network of Sigmoid Neurons

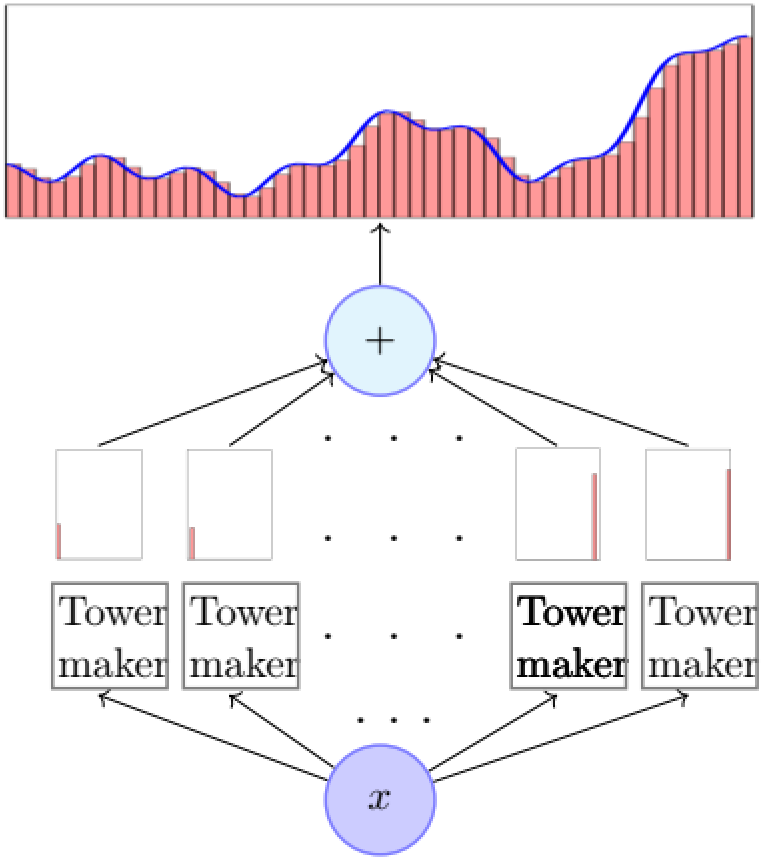
Representation power of a multilayer network of perceptrons	**Representation power of a multilayer network of sigmoid neurons**
A multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors)	A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision

How to represent an arbitrary function?

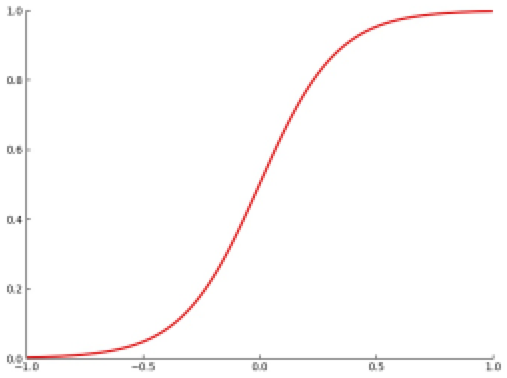


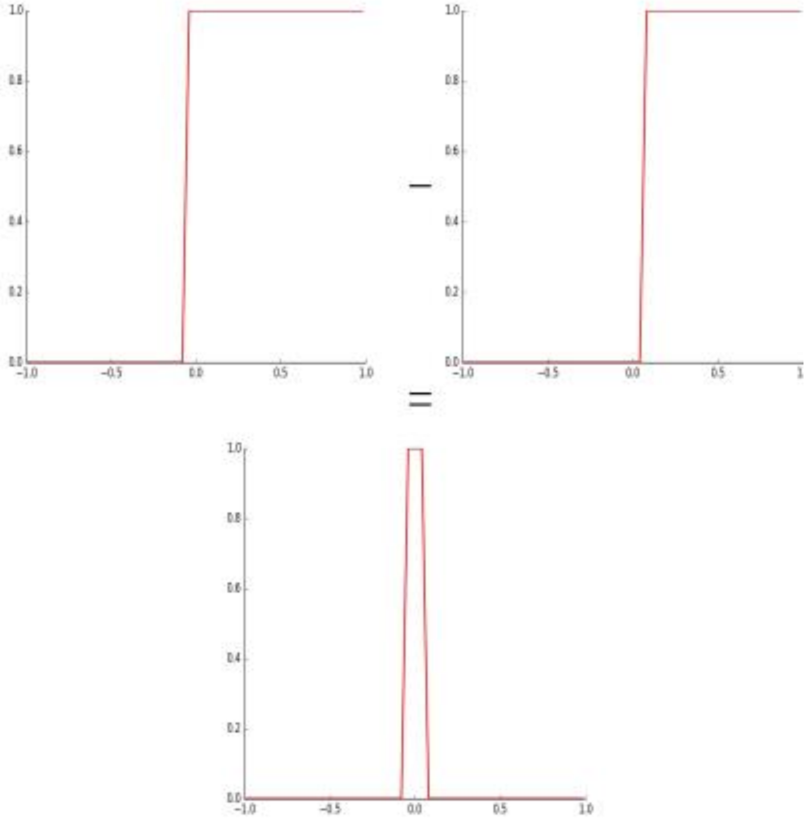


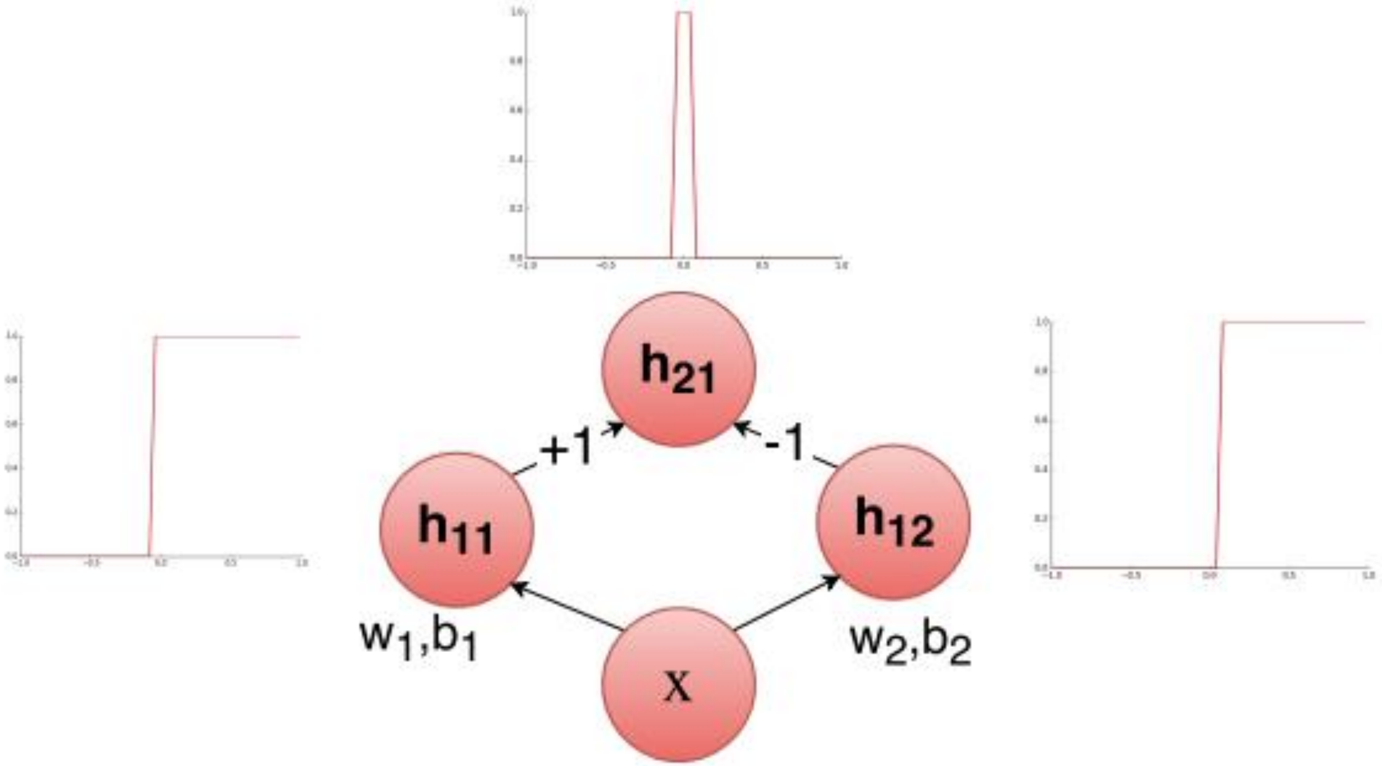
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several "tower" functions
- More the number of such "tower" functions, better the approximation
- To be more precise, we can approximate any arbitrary function by a sum of such "tower" functions


+
Tower maker
Tower maker
Tower maker
Tower maker
x

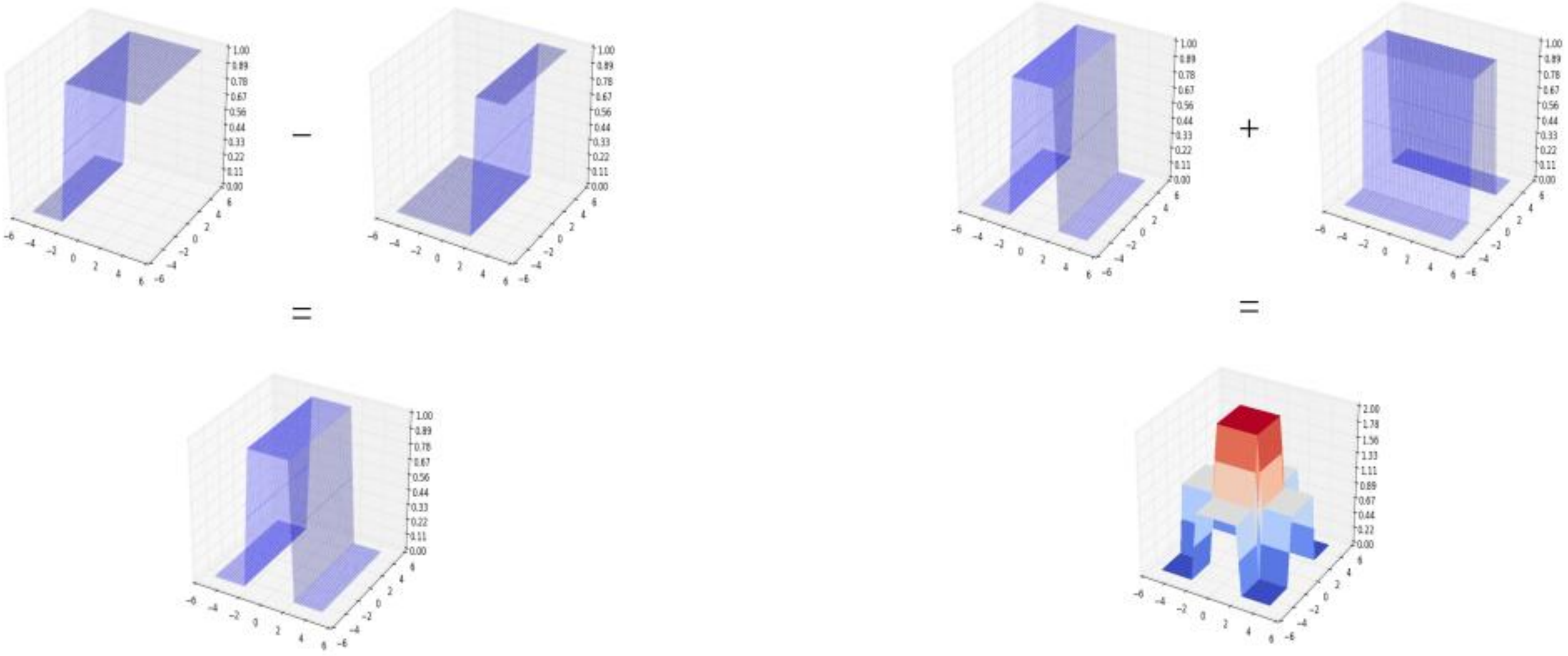
How do we generate tower functions using sigmoid neuron?

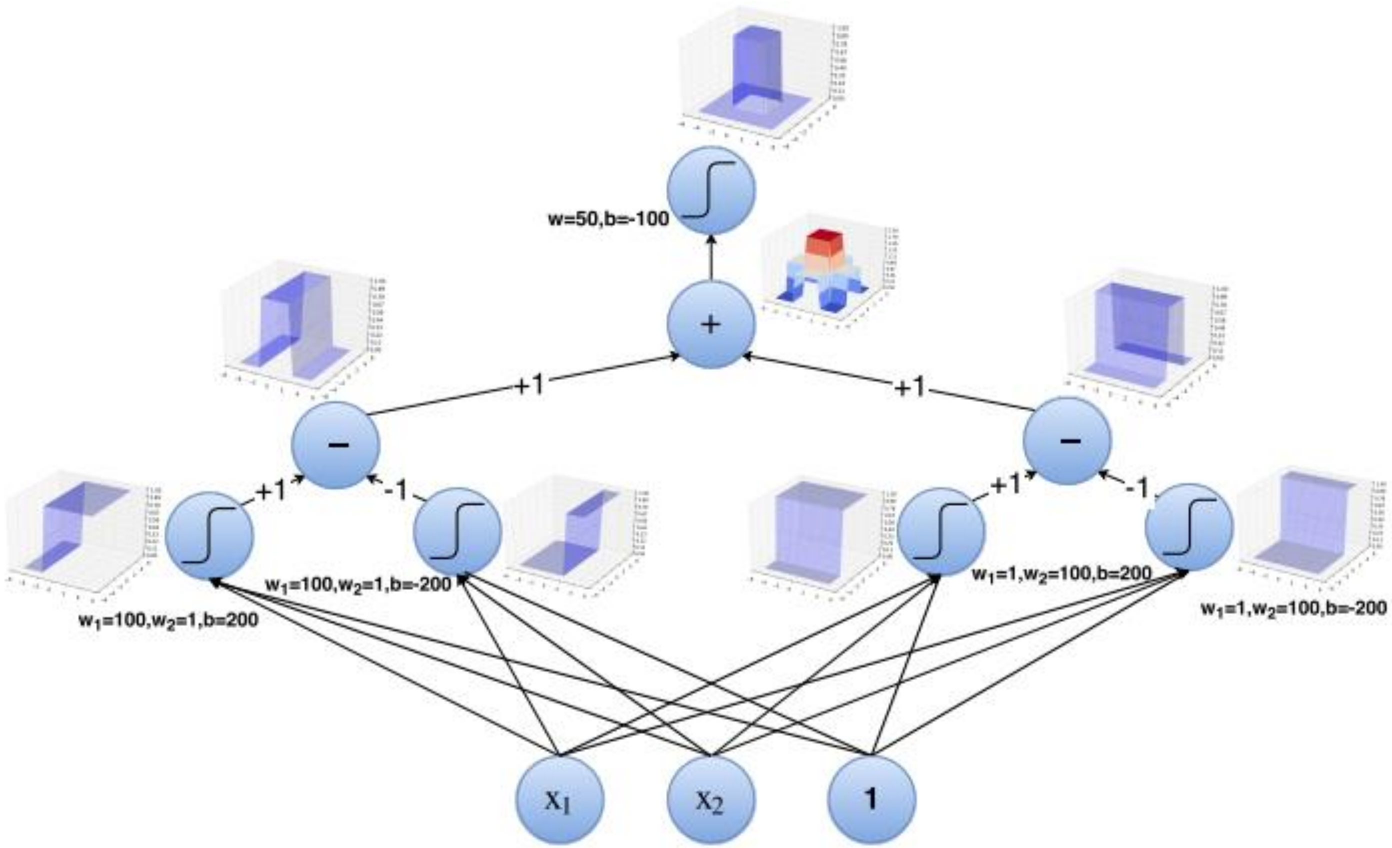





h21
+1
-1
h11
h12
w1,b1
w2,b2
x

2 inputs





w=50,b=-100
+
+1
+1
–
–
+1
-1
+1
-1
w1=100,w2=1,b=-200
w1=100,w2=1,b=200
w1=1,w2=100,b=200
w1=1,w2=100,b=-200
x1
x2
1