DSE 3121 DEEP LEARNING [4 0 0 4]

Slide-2: Fundamentals of Neural Networks + Gradient Descent & Backpropagation

B.Tech Data Science & Engineering
Rohini R Rao & Abhilash Pai
Department of Data Science and Computer Applications
MIT Manipal

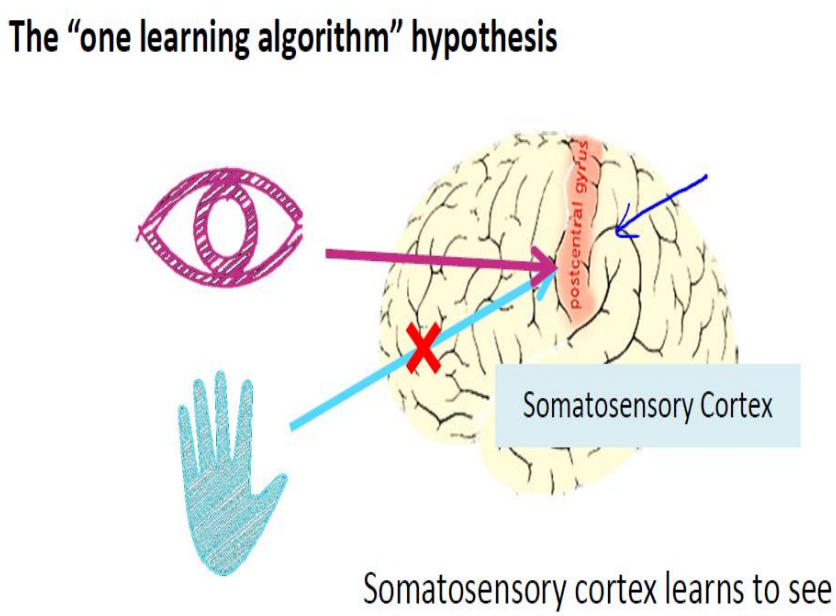
Credits

The contents for this slide set was adapted from:

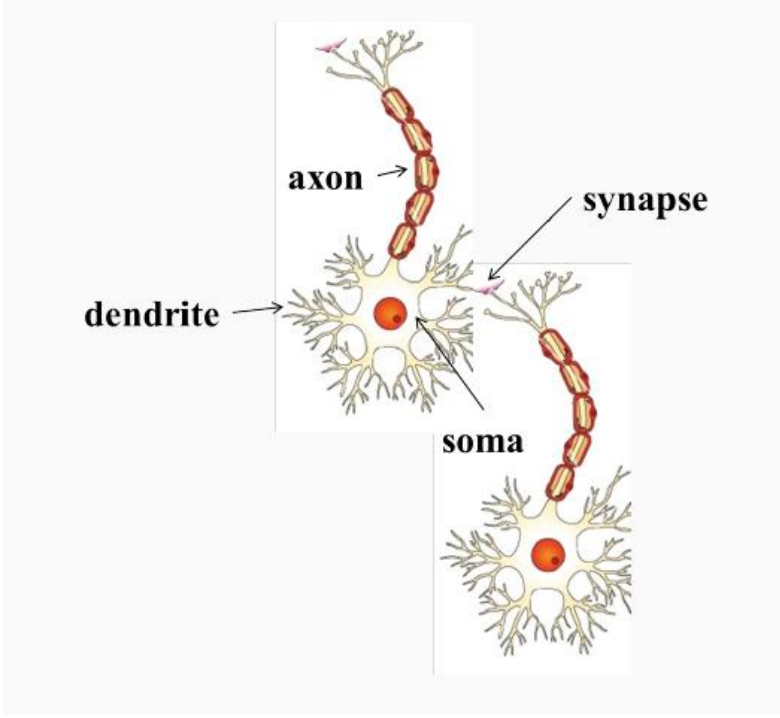
- Course Notes- Deep Learning IIT Madras, Prof Dr. Mithesh Khapra

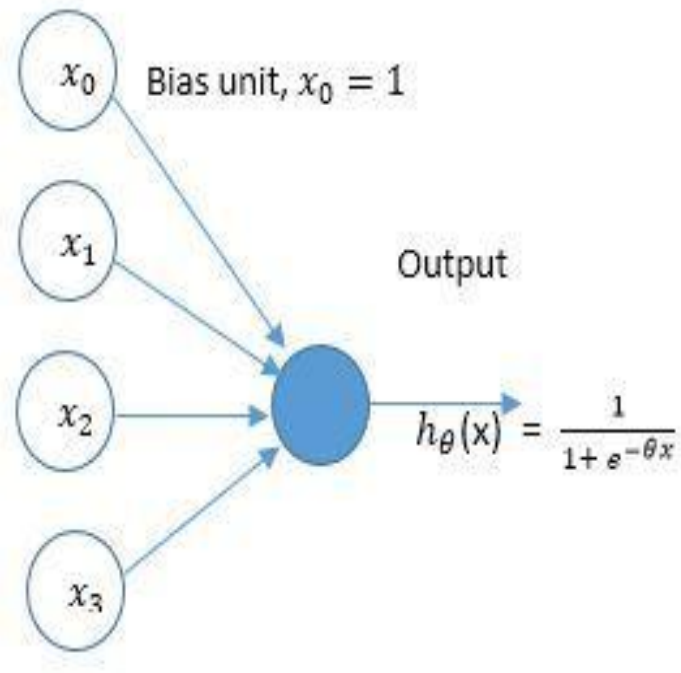
Neural Network

- is a set of connected input/output units in which each connection has
- a weight associated with it.
- During the learning phase, the network learns by adjusting
- the weights so as to be able to predict the correct class label of the input tuples.
- Also referred to as **Connectionist learning**

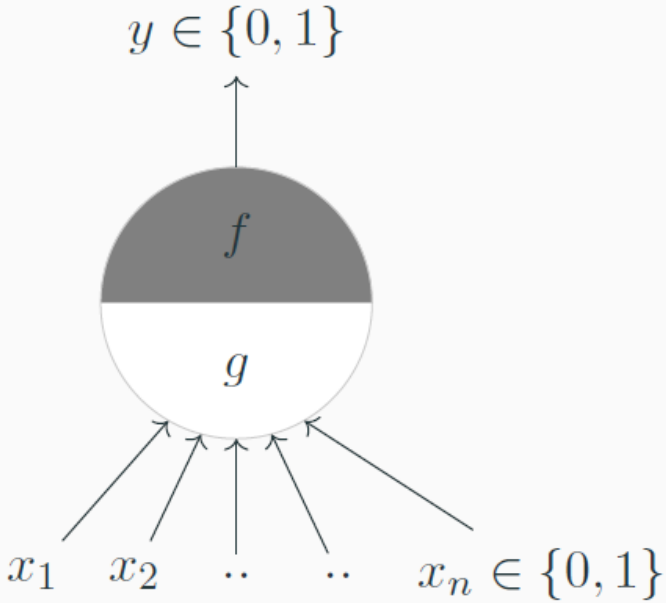


Neuron is a computational , logistic unit





McCulloch Pits (MP) Neuron

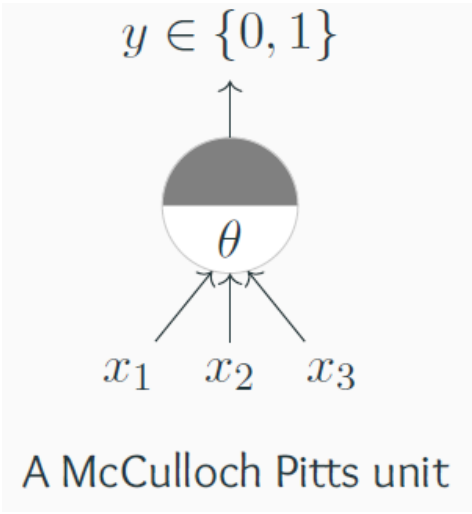


$$g(x_1, x_2, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

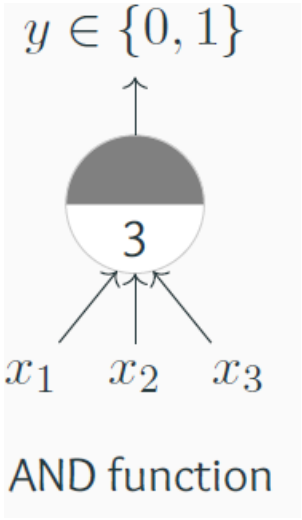
$$y = f(g(\mathbf{x})) = 1 \quad if \quad g(\mathbf{x}) \geq \theta$$
$$= 0 \quad if \quad g(\mathbf{x}) < \theta$$

- θ is called the thresholding parameter
- This is called Thresholding Logic

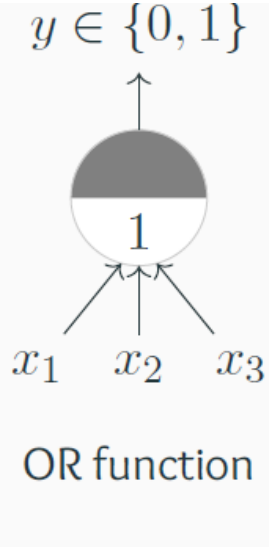
Implementing Boolean functions using MP Neuron



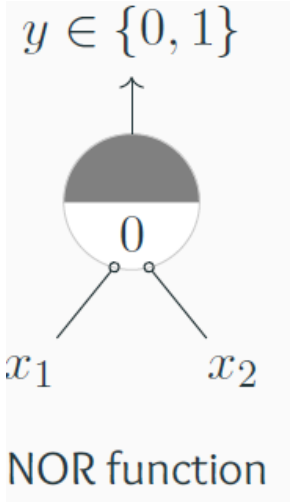
A McCulloch Pitts unit



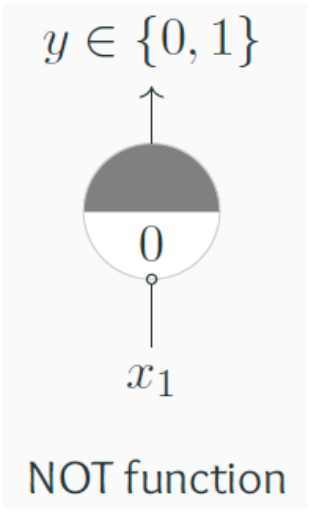
AND function



OR function



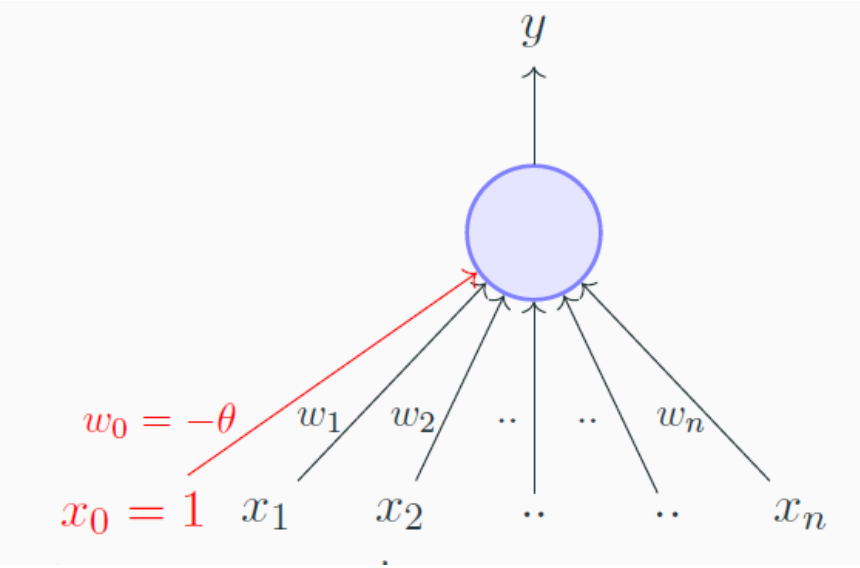
NOR function



NOT function

diff thresholds to activate the functions

Perceptron



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

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

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

Weights were introduced to given more importance to certain inputs/features

MP Neuron vs Perceptron

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

MP Neuron

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

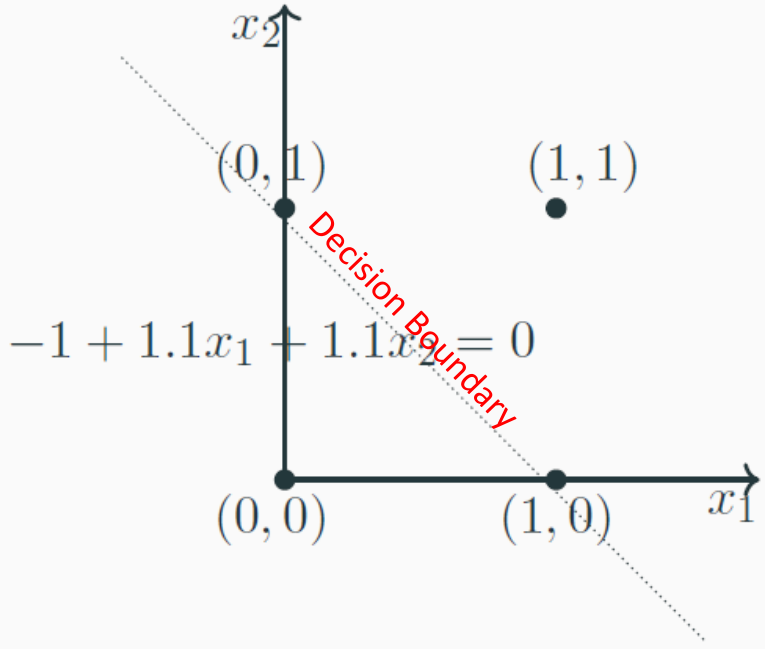
Perceptron

Boolean function using Perceptron : Example (OR)

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

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

System of Linear Inequalities

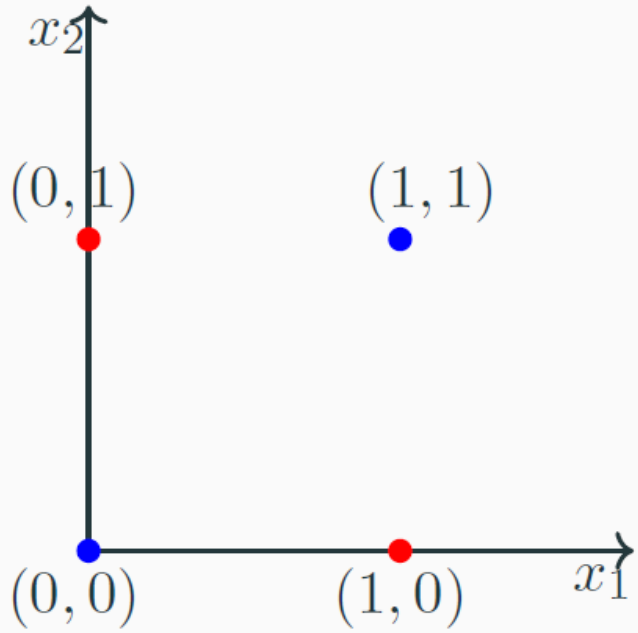


$$w_0 = -1, w_1 = 1.1, , w_2 = 1.1$$

One Possible Solution

The XOR Conundrum

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



Non-linear!

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

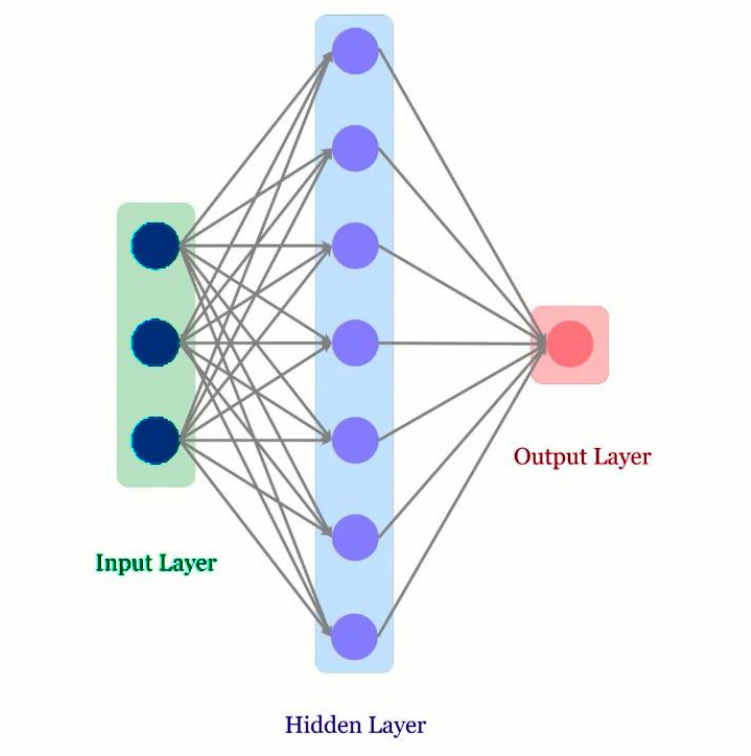
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 \geq -w_0$$

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

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

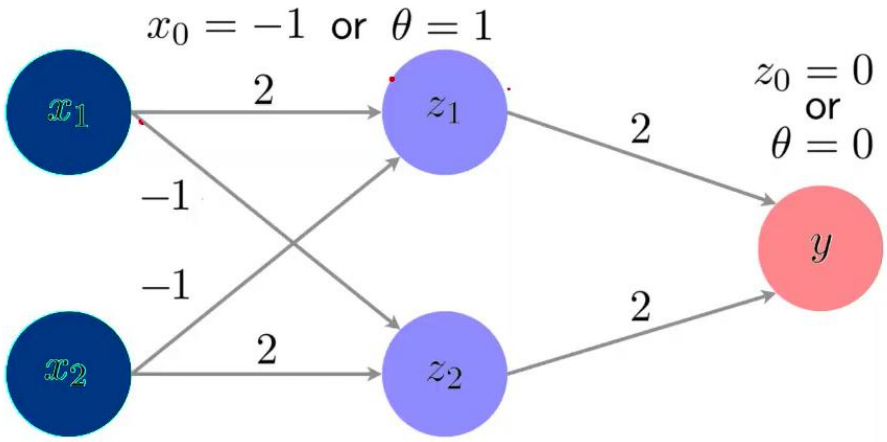
Contradicting equations, no possible solution!!

Solving XOR using Multi-Layer Perceptrons (MLP)



Example of MLP

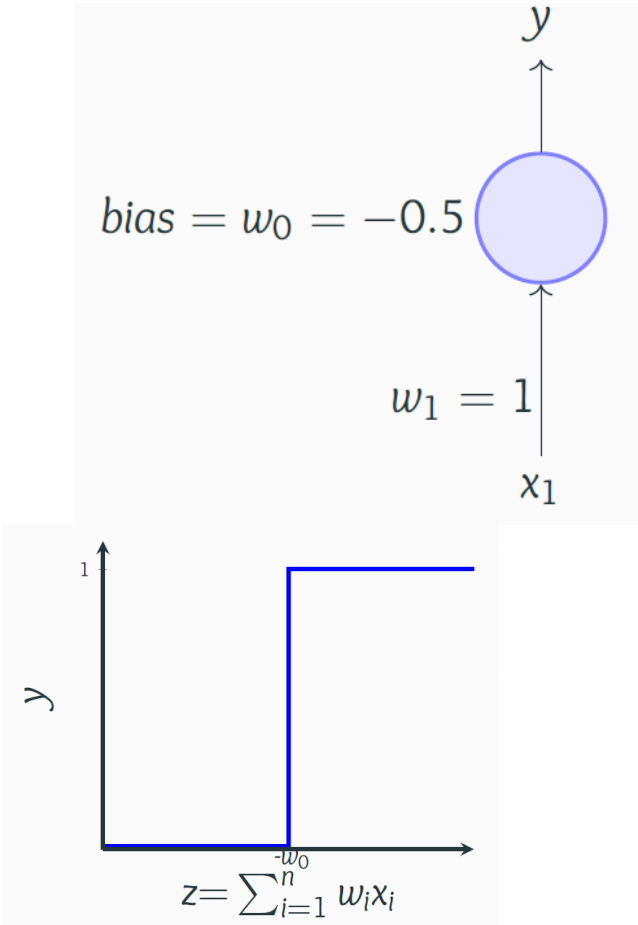
Solving XOR using Multi-Layer Perceptrons (MLP)



(x_1,x_2)	(z_1,z_2)	y
(0,0)	(0,0)	0
(0,1)	(0,0)	1
(1,0)	(1,0)	1
(1,1)	(0,0)	0

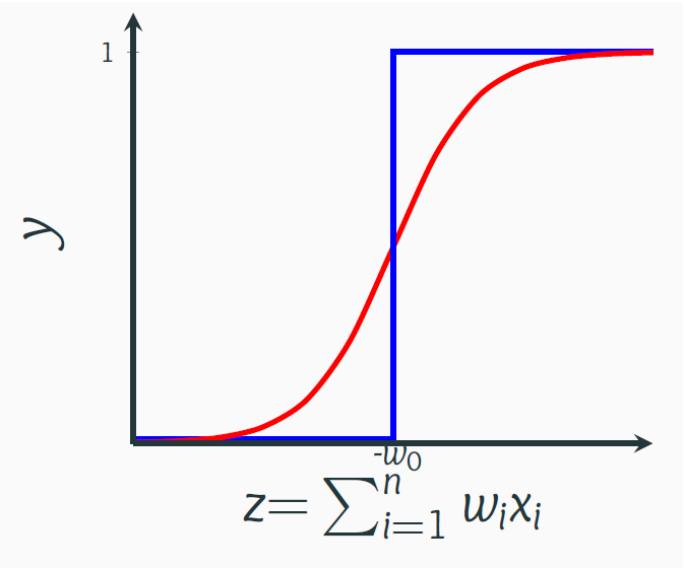
Theorem: Any boolean function of n inputs can be represented exactly by a network of perceptrons containing 1 hidden layer with 2^n perceptrons and one output layer containing 1 perceptron

Need for activation functions



- The thresholding logic used by a perceptron is very harsh !
- Eg: When $-w_0 = 0.5$, though the output values 0.49 and 0.51 are very close to each other, the perceptron would assign different values to them.
- This behavior is not a characteristic of the specific problem, the weight or threshold that we chose, it is a characteristic of the perceptron function itself which behaves like a step function

Sigmoid Neuron



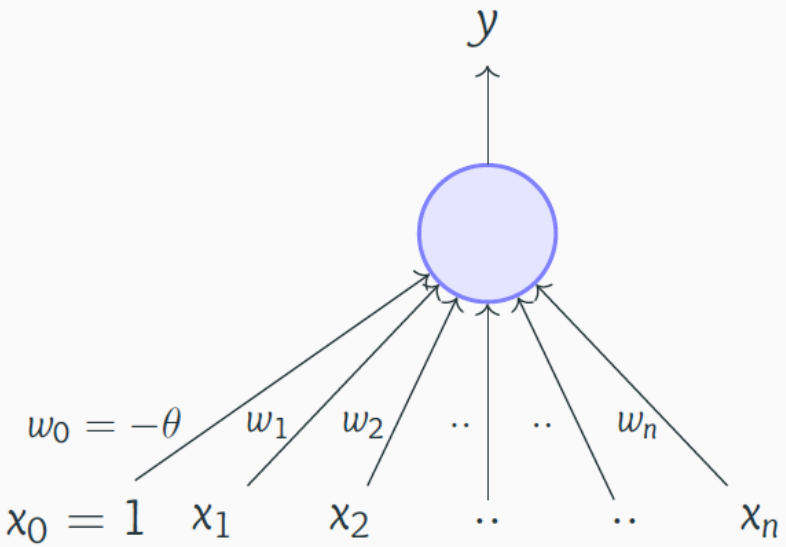
Here is one form of the sigmoid function called the logistic function

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

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

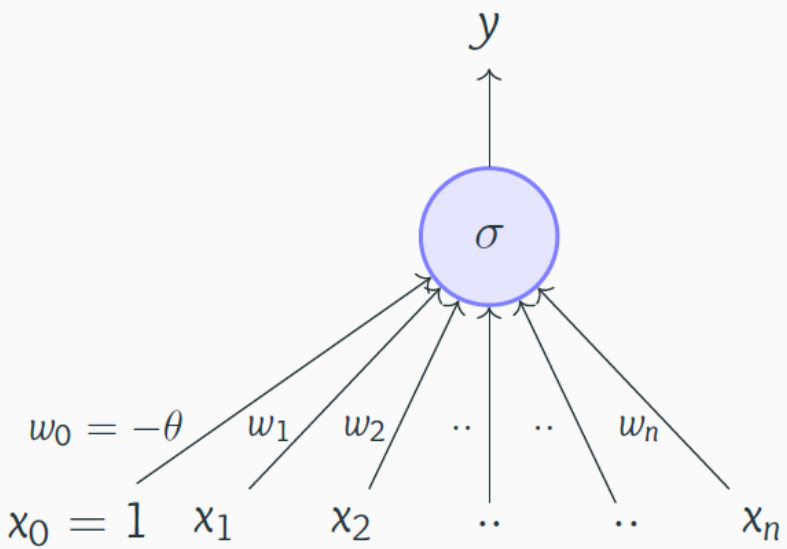
Perceptron



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

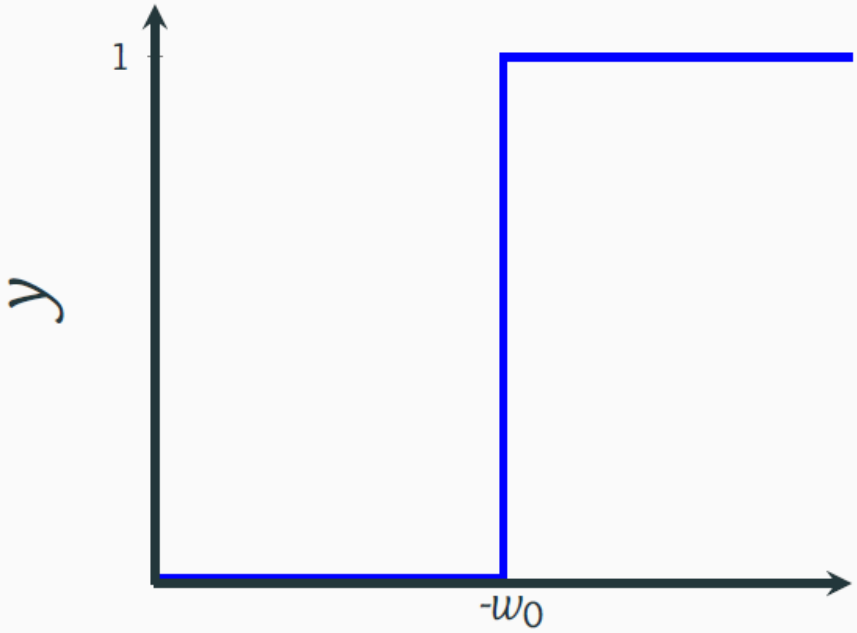
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Sigmoid (logistic) Neuron



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

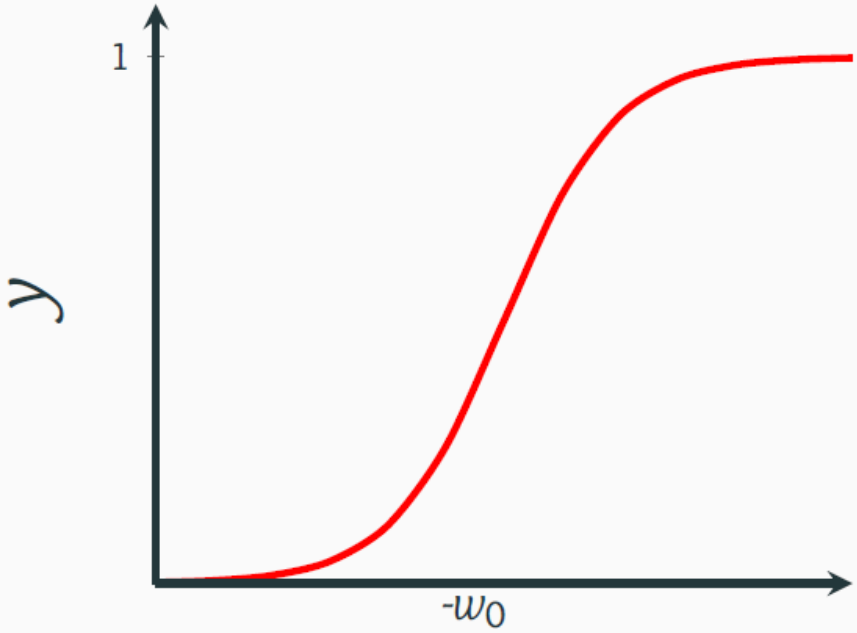
Perceptron



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

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

Sigmoid Neuron



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

Smooth, continuous, **differentiable**

Let's see how Neural Networks Learn..

Neural Network learning as optimization

- cannot calculate the perfect weights for a neural network since there are too many unknowns.
- Instead, the problem of learning is as a search or optimization problem
- An algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions.
- **Objective of optimizer** is to minimize the loss function or the error term.
- loss function
  - gives the difference between observed value from the predicted value.
  - must be **continuous and differentiable at each point**.
  - To minimize the loss generated from any model compute
    - The magnitude that is by how much amount to decrease or increase, and
    - direction in which to move

A Typical Machine Learning Set-up

Data: $\{x_i, y_i\}_{i=1}^{n}$

Model: Our approximation of the relation between $\mathbf{x}$ and y. For example,

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

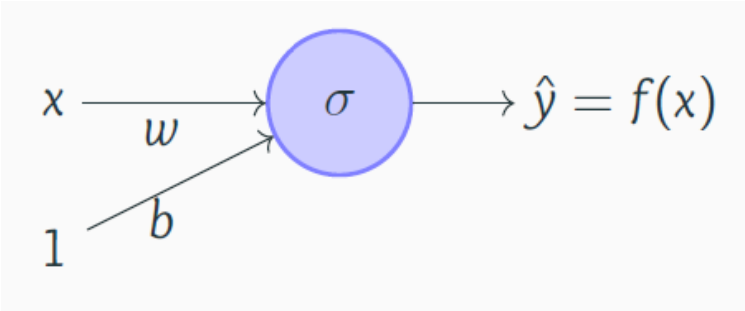
Parameters: In all the above cases, w is a parameter which needs to be learned from the data

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

Training a Neural Network

- Let us understand what it means to train a network, by considering the following perceptron model:



- Consider that we are going to train the network to behave like a sigmoid function.

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

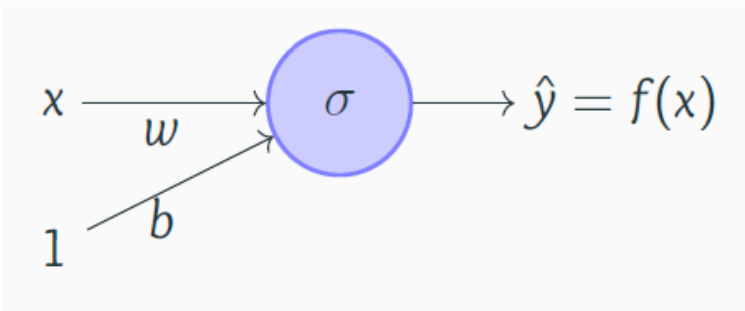
What does it mean to train the network?

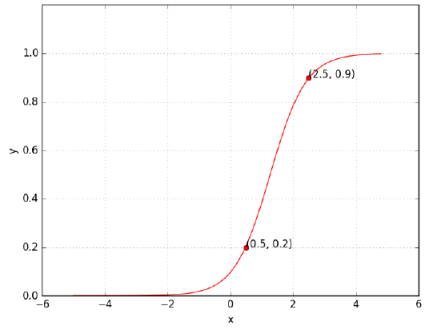
Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$

At the end of training we expect to find w*, b* such that: $f(0.5) \to 0.2$ and $f(2.5) \to 0.9$

Training a Neural Network

- Let us understand what it means to train a network, by considering the following perceptron model:





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

- Consider that we are going to train the network to behave like a sigmoid function.

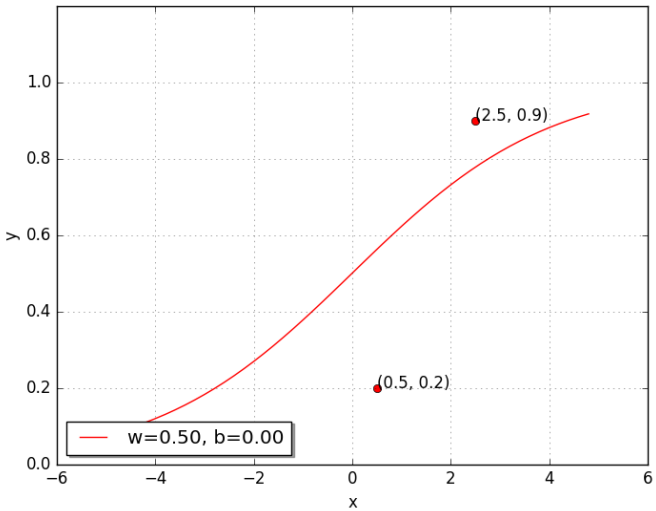
What does it mean to train the network?

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: $f(0.5) \to 0.2$ and $f(2.5) \to 0.9$

In other words...

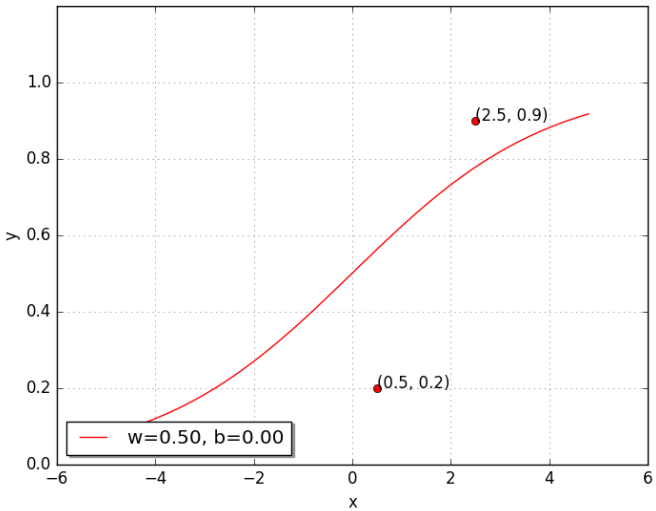
We hope to find a sigmoid function such that $(0.5, 0.2)$ and $(2.5, 0.9)$ lie on this sigmoid



Can we try to find such a w^*, b^* manually

Let us try a random guess.. (say, $w = 0.5$, $b = 0$)

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

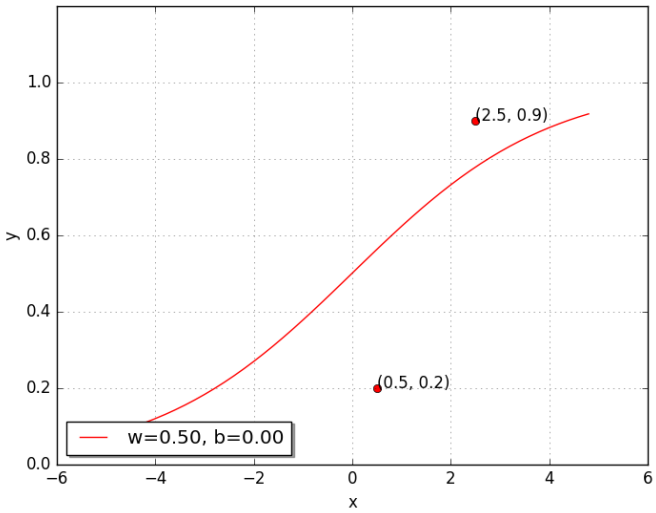


Can we try to find such a w^*, b^* manually

Let us try a random guess.. (say, $w = 0.5$, $b = 0$)

Clearly not good, but how bad is it ?

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



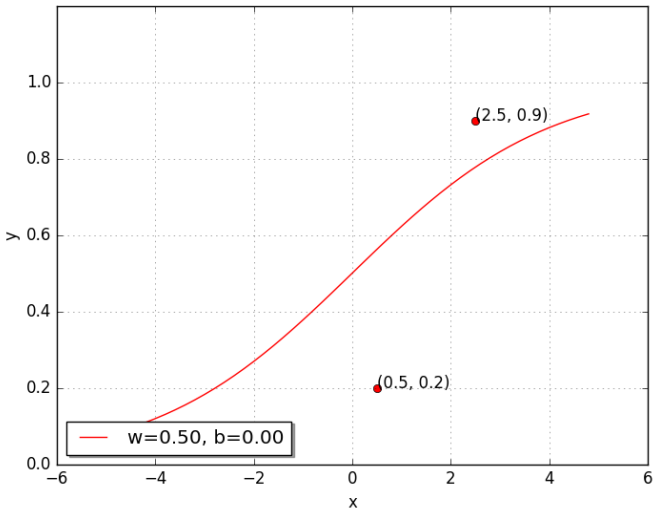
Can we try to find such a w^*, b^* manually

Let us try a random guess.. (say, $w = 0.5$, $b = 0$)

Clearly not good, but how bad is it ?

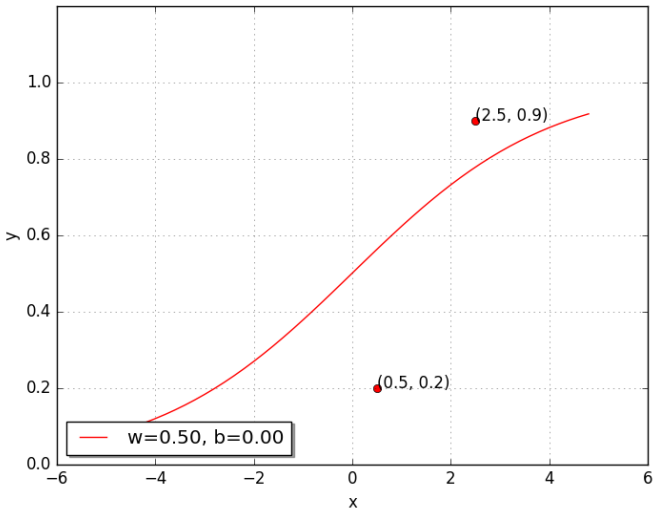
Let us revisit $L(w, b)$ to see how bad it is ...

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



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

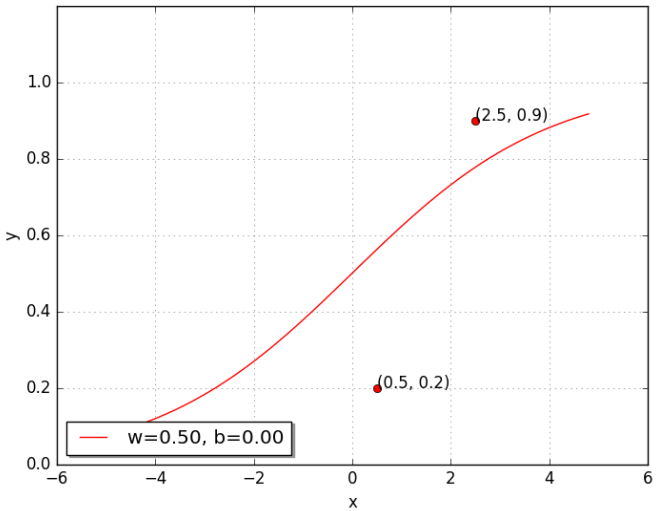
$$\sigma(x) = \frac{1}{1 + e^{-}\ (wx+b)}$$



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

$$= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2$$

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

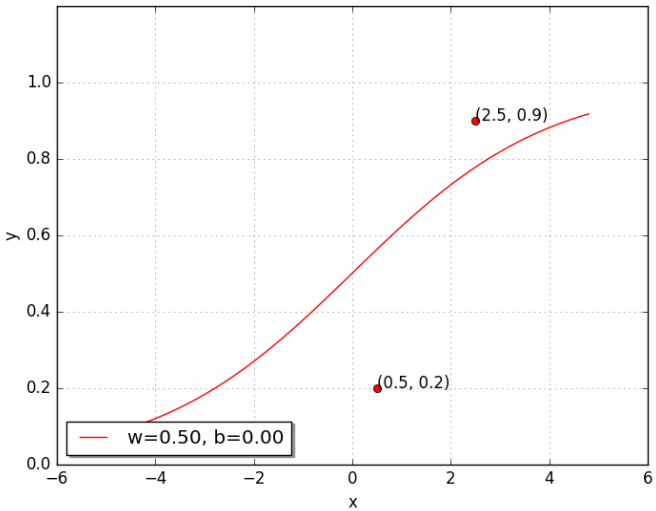


$$L\ (w, b) = \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2$$

$$= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2$$

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



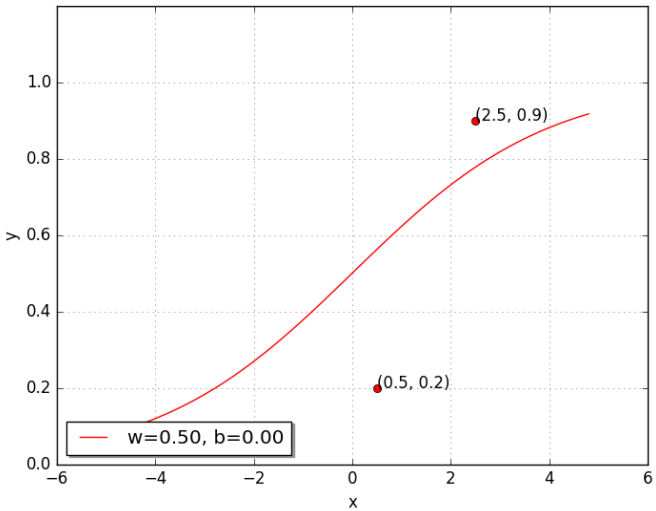
$$L\ (w, b) = \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2$$

$$= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2$$

$$= 0.073$$

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



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

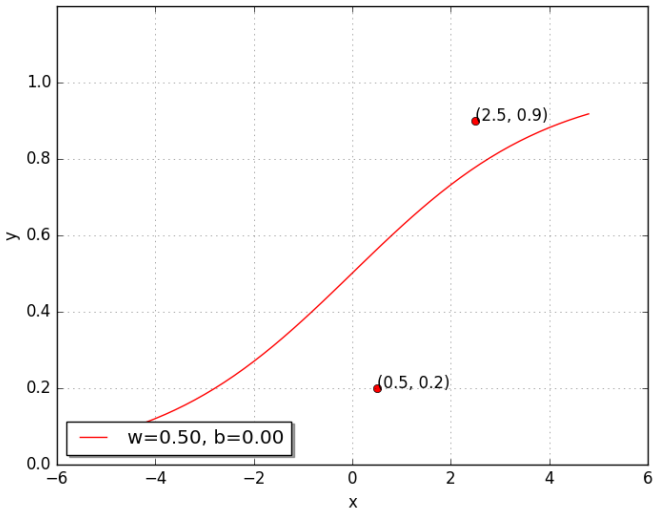
$$= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2$$

$$= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2$$

$$= 0.073$$

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

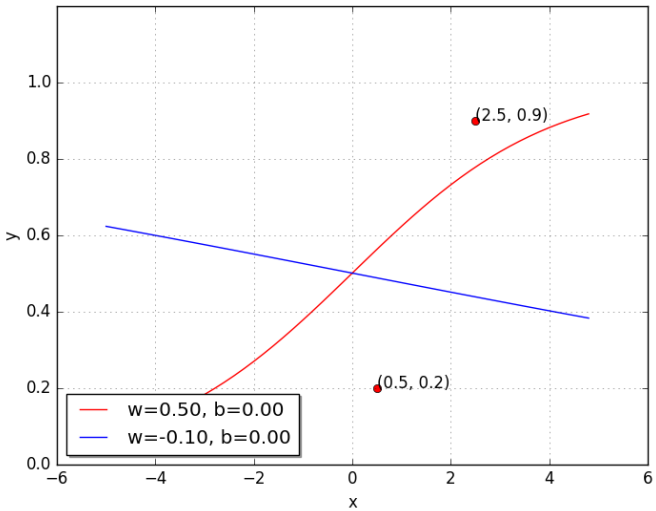
We want $L\ (w, b)$ to be as close to 0 as possible



Let us try some other values of w, b

w	b	$L\ (w, b)$
0.50	0.00	0.0730

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

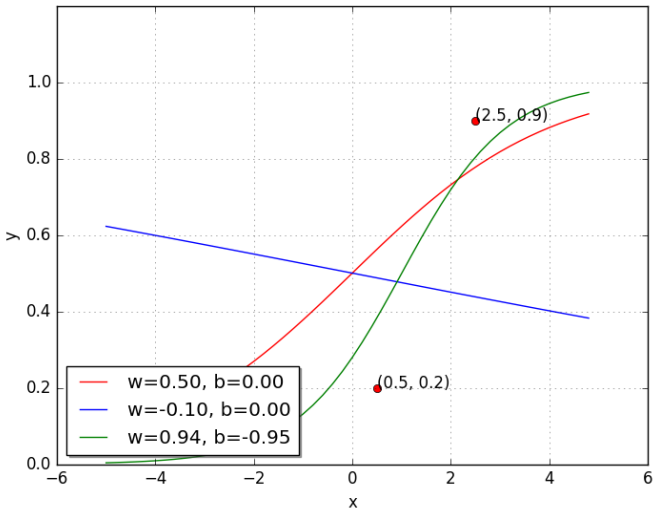


Let us try some other values of w, b

w	b	$L(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481

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

Oops!! this made things even worse...

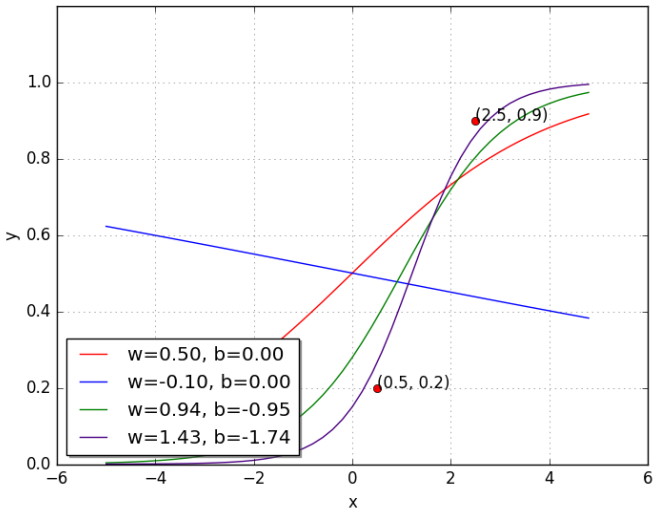


Let us try some other values of w, b

w	b	$L(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214

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

Perhaps it would help to push w and b in the other direction...

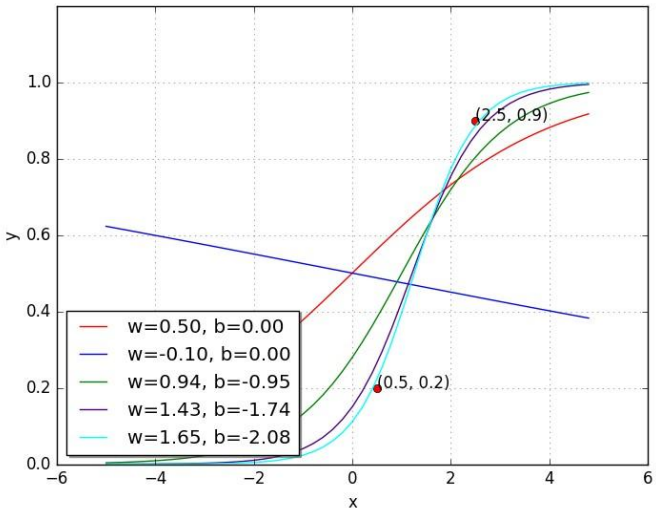


Let us try some other values of w, b

w	b	$L(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028

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

Let us keep going in this direction, *i.e.*, increase w and decrease b

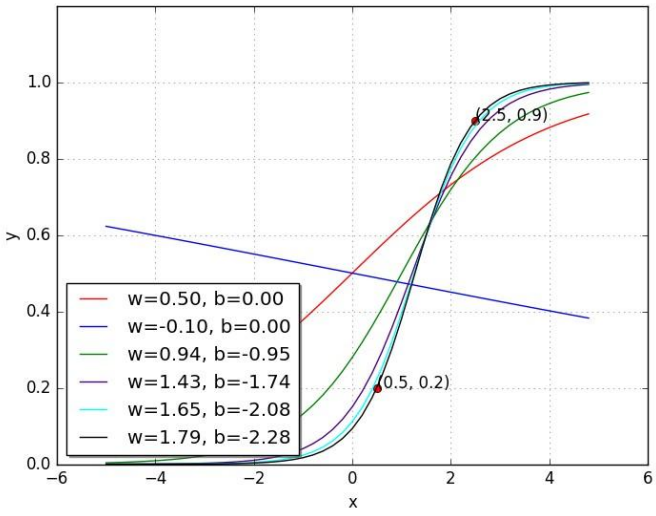


Let us try some other values of w, b

w	b	$L(w, b)$
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

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

Let us keep going in this direction, *i.e.*, increase w and decrease b



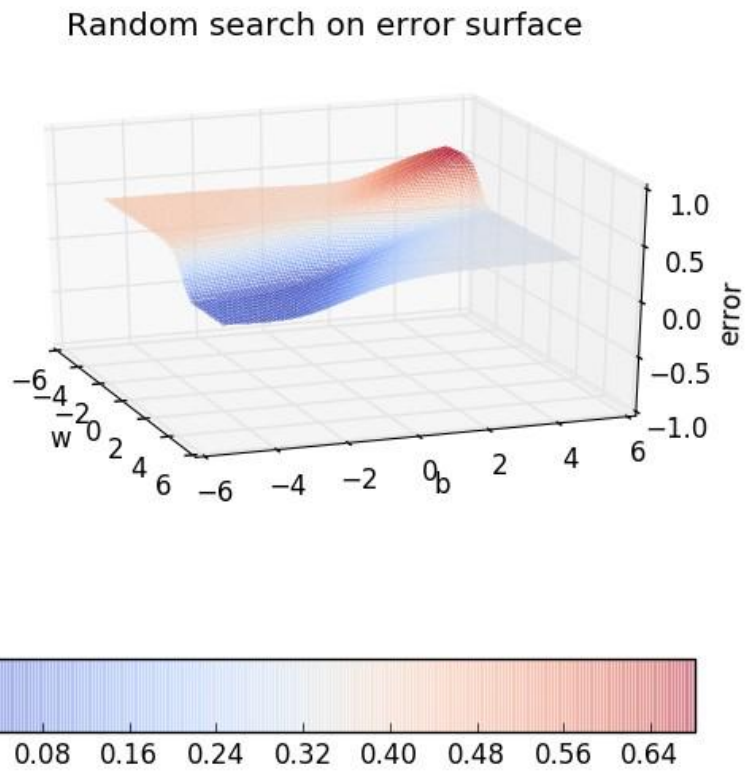
Let us try some other values of w, b

w	b	$L(w, b)$
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

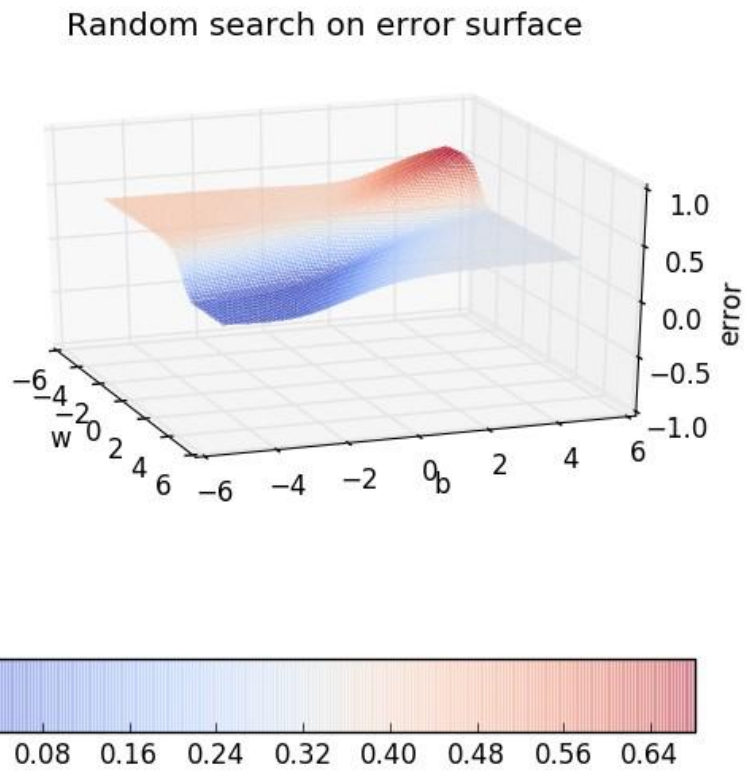
$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

With some guess work and intuition we were able to find the right values for w and b

Let us look at something better than our "guess work" algorithm....

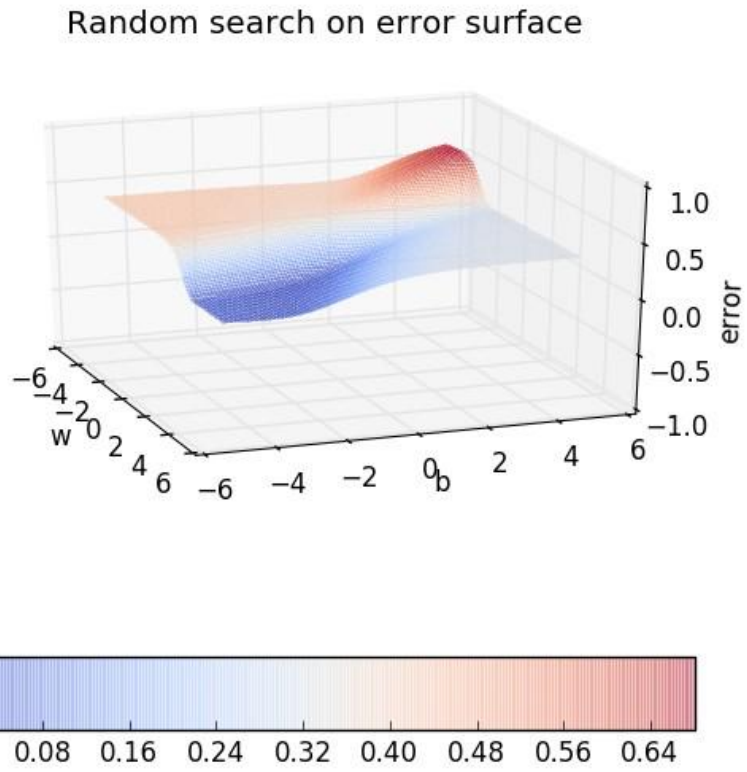


Since we have only 2 points and 2 parameters (w, b) we can easily plot $L\ (w, b)$ for different values of (w, b) and pick the one where $L\ (w, b)$ is minimum



Since we have only 2 points and 2 parameters (w, b) we can easily plot $L\ (w, b)$ for different values of (w, b) and pick the one where $L\ (w, b)$ is minimum

But of course this becomes intractable once you have many more data points and many more parameters !!

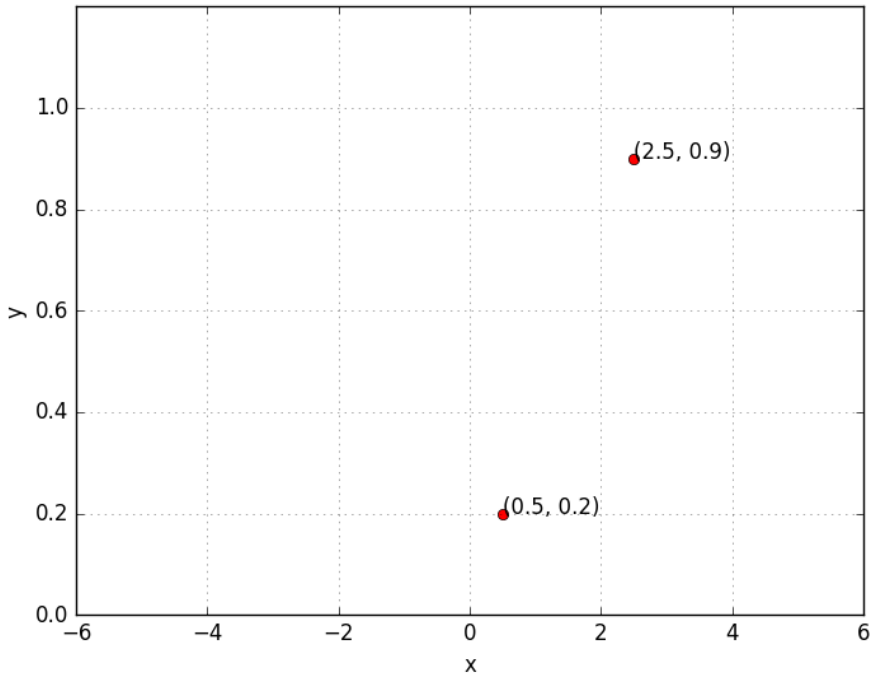


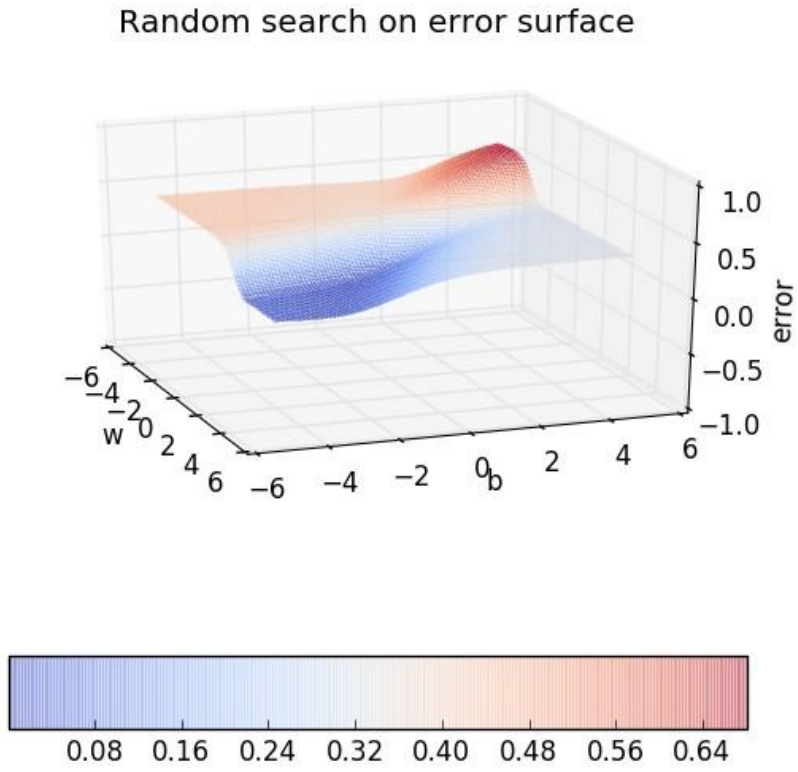
Since we have only 2 points and 2 parameters (w, b) we can easily plot $L(w, b)$ for different values of (w, b) and pick the one where $L(w, b)$ is minimum

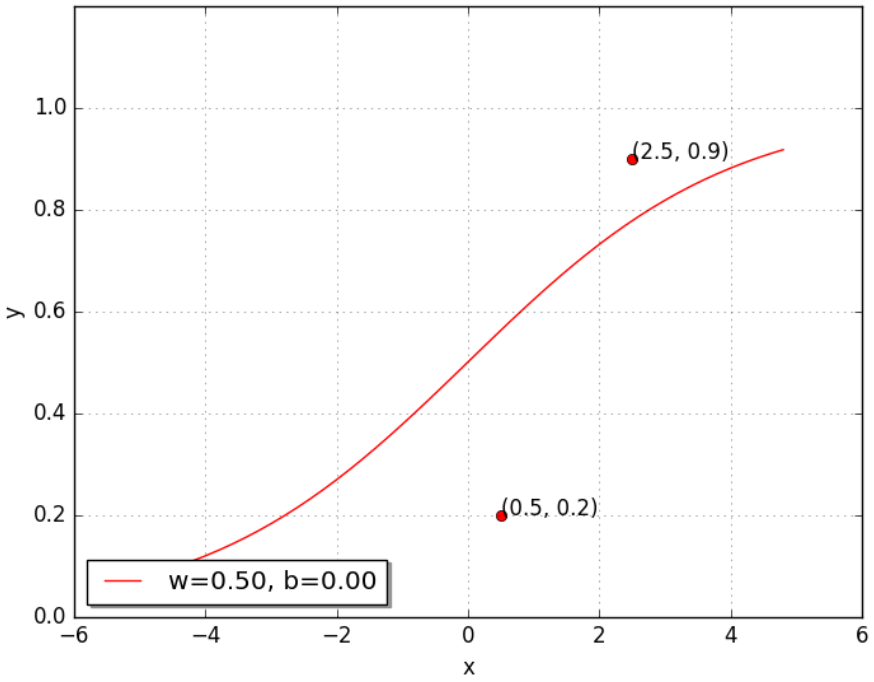
But of course this becomes intractable once you have many more data points and many more parameters !!

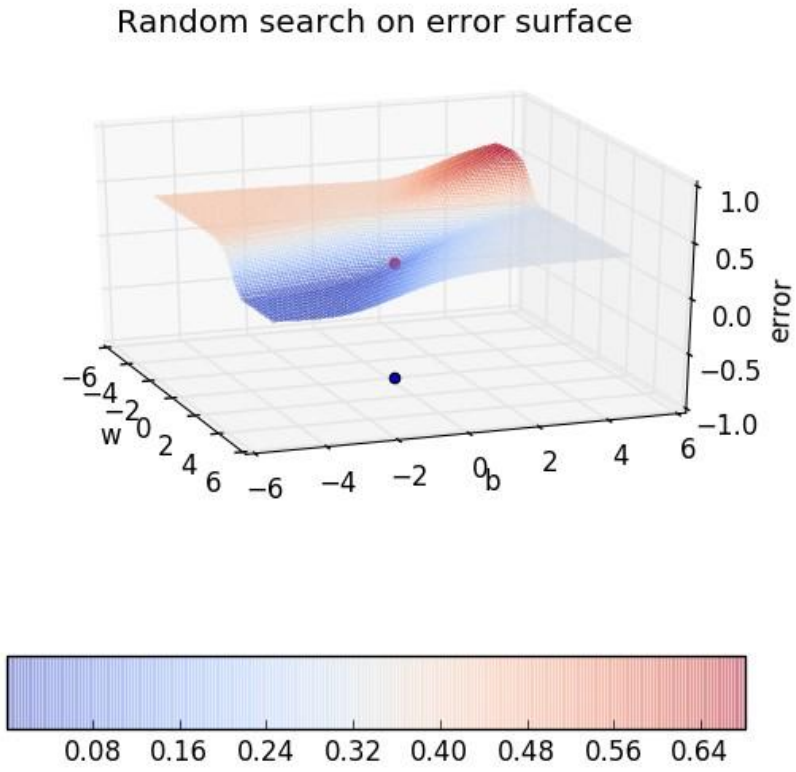
Further, even here we have plotted the error surface only for a small range of (w, b) [from $(-6, 6)$ and not from $(-\text{inf}, \text{inf})$]

Let us look at the geometric interpretation of our "guess work" algorithm in terms of this error surface

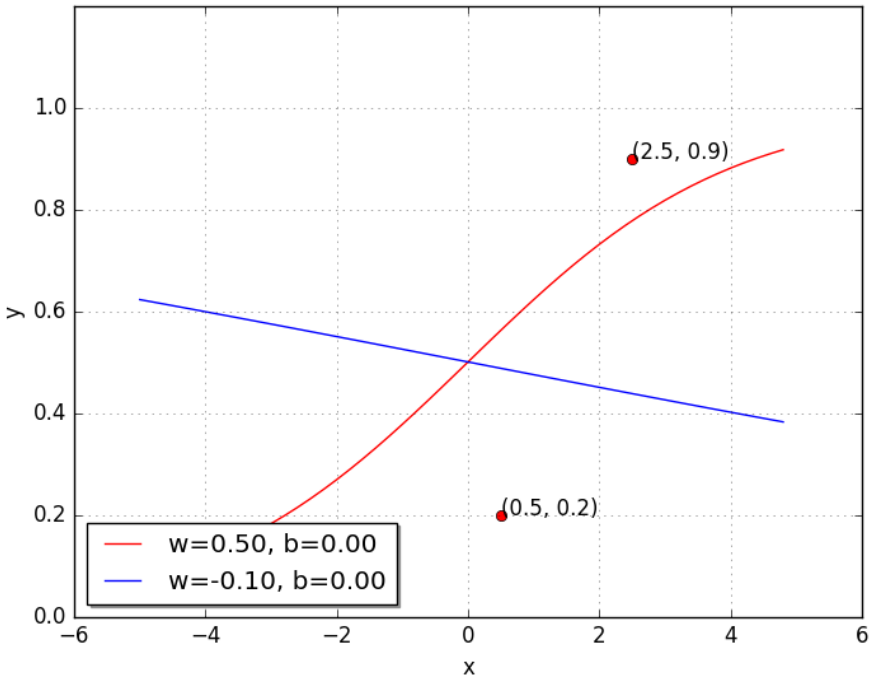

(2.5, 0.9)
(0.5, 0.2)
y
x

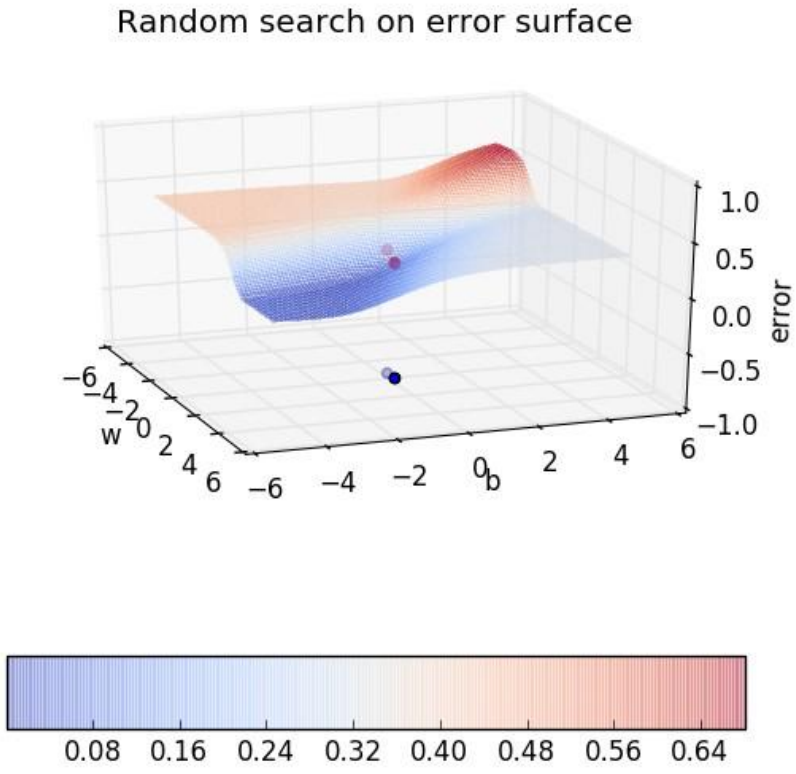

Random search on error surface
error
w
b

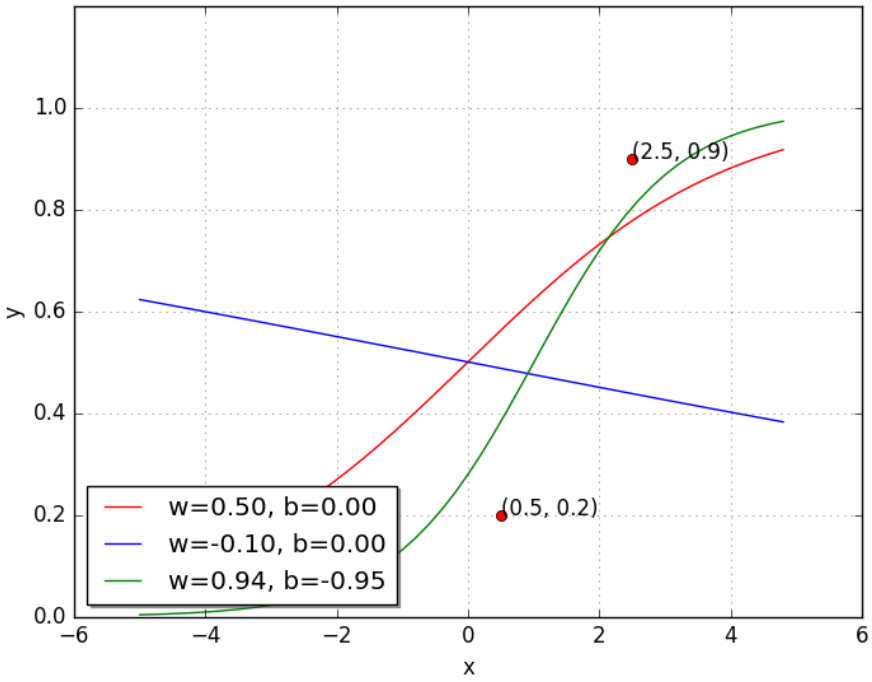


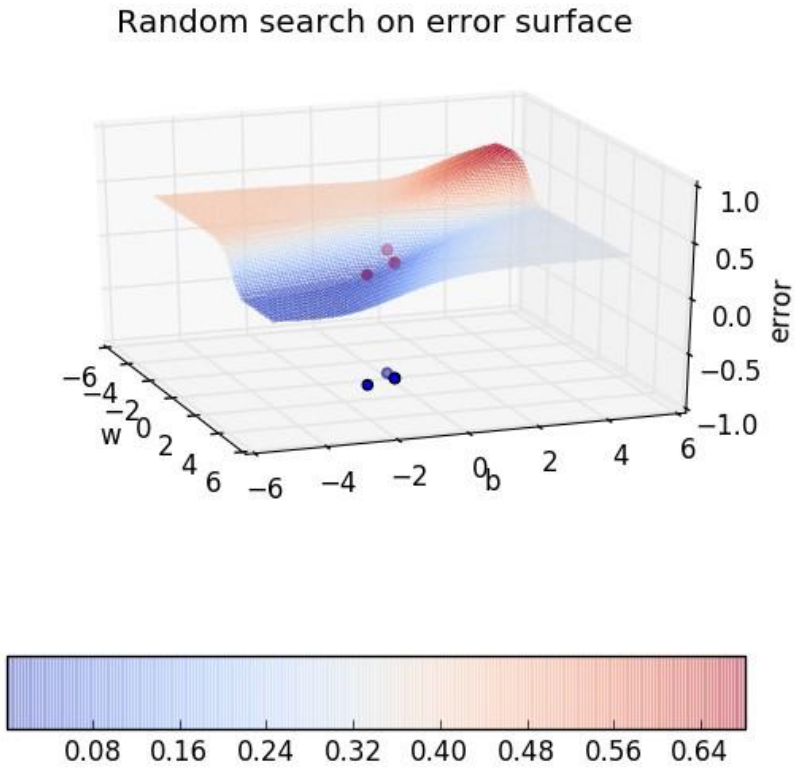


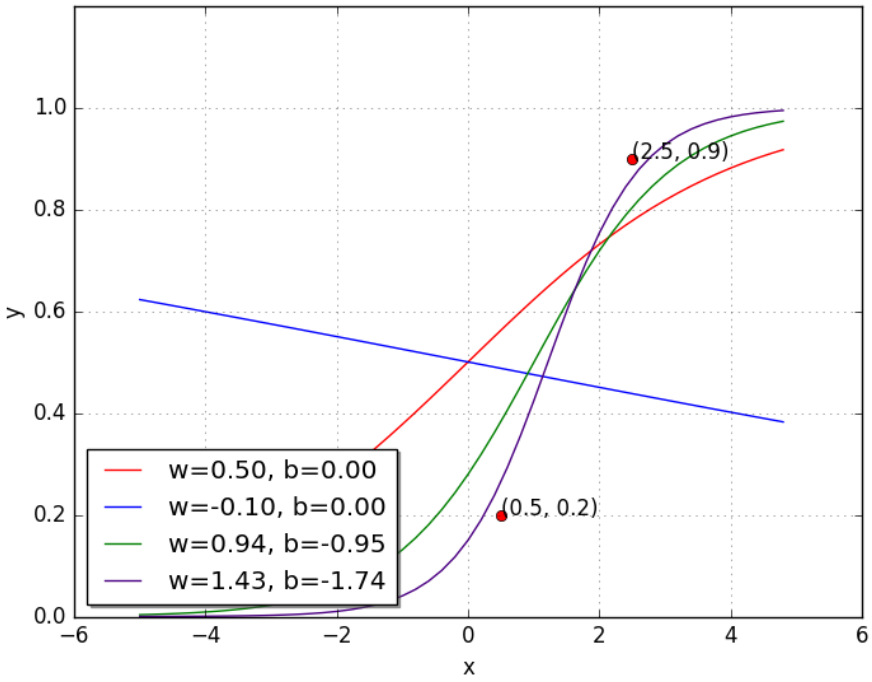
basically w and b on x and y and then the value of loss as z direction we are doing for our example of sigmoid function with neuron training

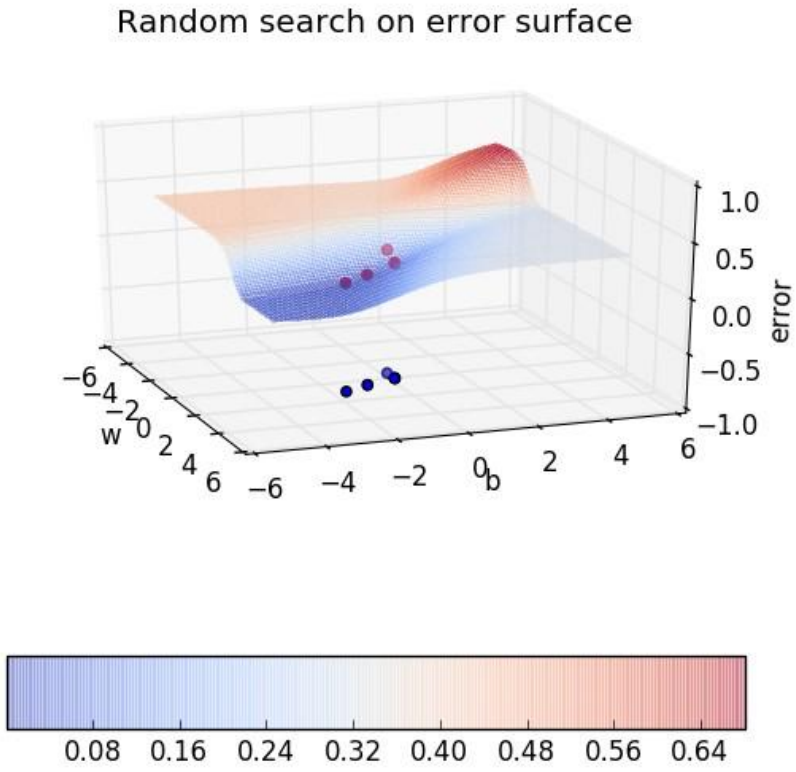

1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00

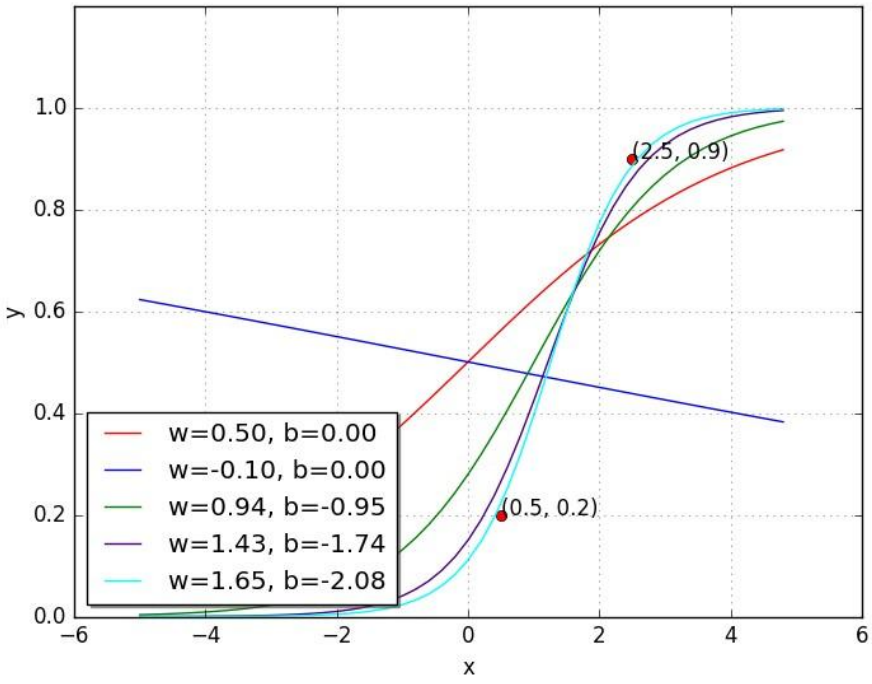

Random search on error surface
1.0
0.5
0.0
−0.5
−1.0
error
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64

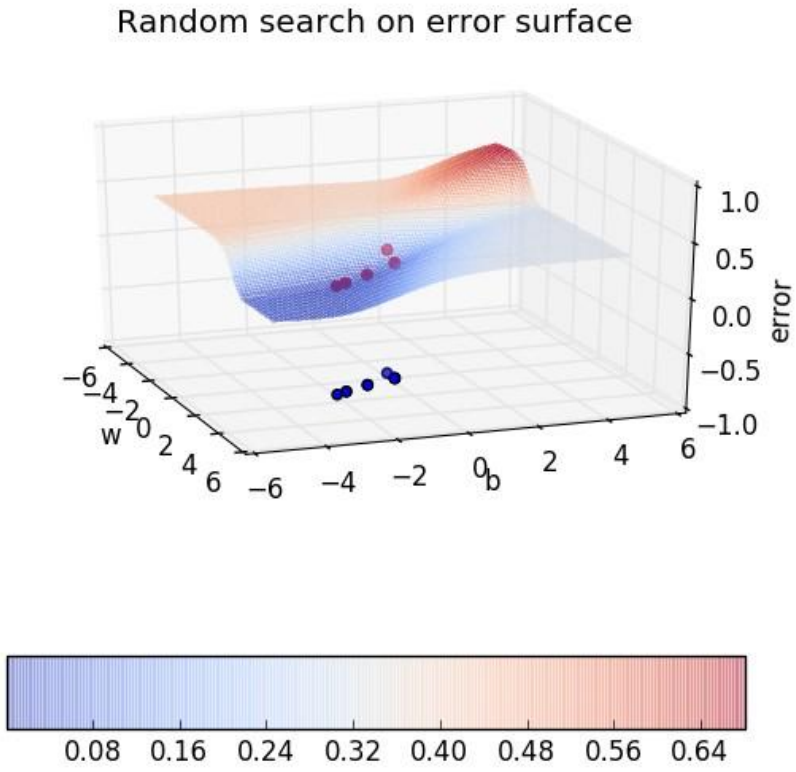

1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95

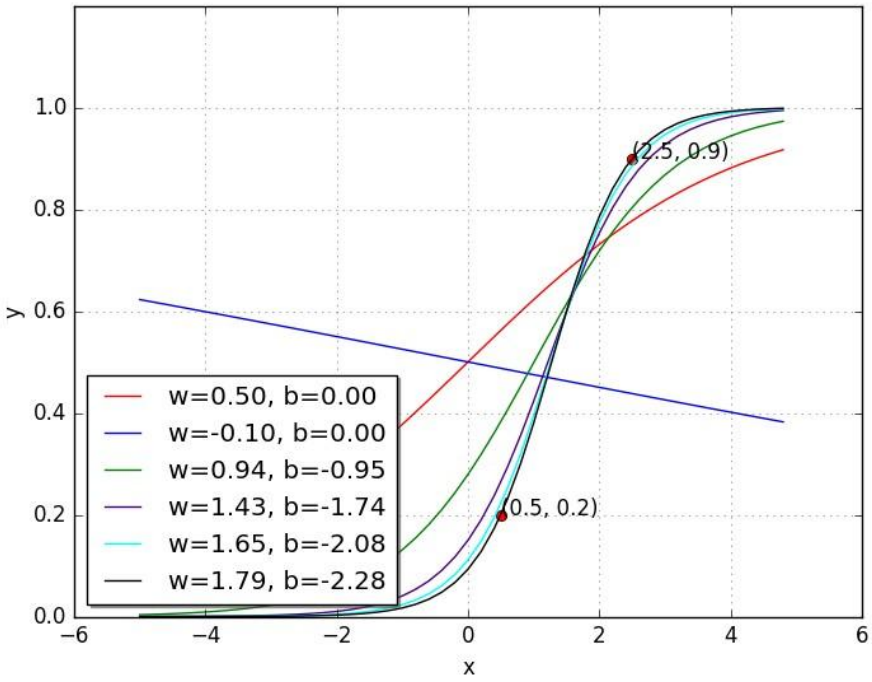

Random search on error surface
1.0
0.5
0.0
−0.5
−1.0
error
−6
−4
−2
0
2
4
6
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64

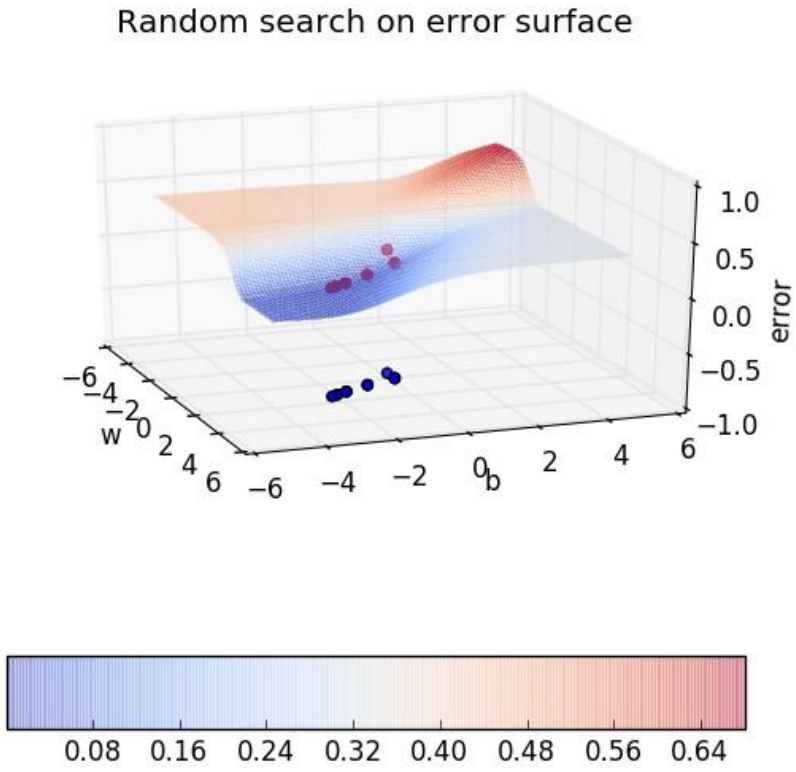

w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
(2.5, 0.9)
(0.5, 0.2)
x
y


Random search on error surface
error
w
b


w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08
(2.5, 0.9)
(0.5, 0.2)
x
y


Random search on error surface
w
b
error


w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08
w=1.79, b=-2.28
(2.5, 0.9)
(0.5, 0.2)
x
y


Random search on error surface
error
w
b

Now let us see if there is a more eficient and principled way of doing this

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!

vector of parameters,
say, randomly initialized

$\theta = [w, b]$

vector of parameters,
say, randomly initialized

$\theta = [w, b]$

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

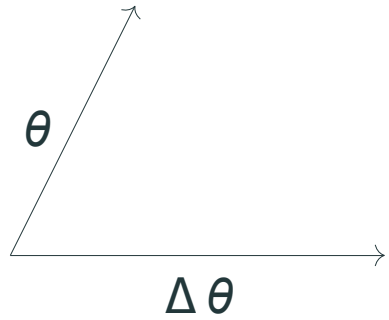
change in the
values of w, b

vector of parameters,
say, randomly initialized

$\theta = [w, b]$

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

change in the
values of w, b

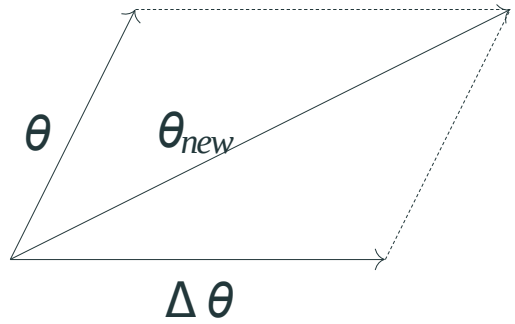


vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b

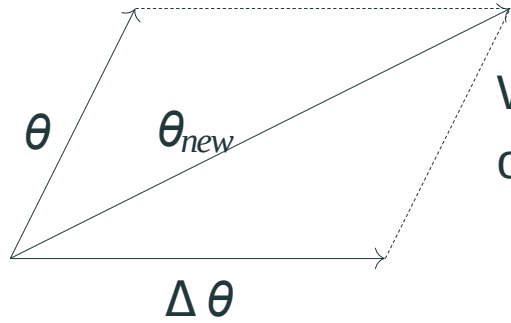


vector of parameters,
say, randomly initialized

$\theta = [w, b]$

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

change in the
values of w, b



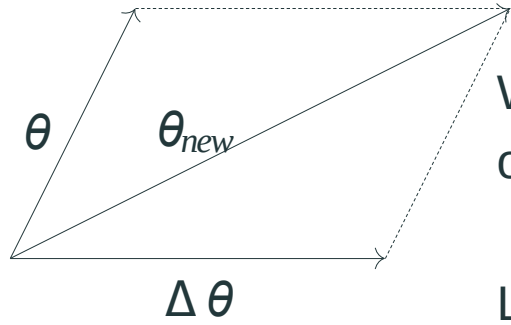
We moved in the direction
of $\Delta\theta$

vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b



We moved in the direction of $\Delta\theta$

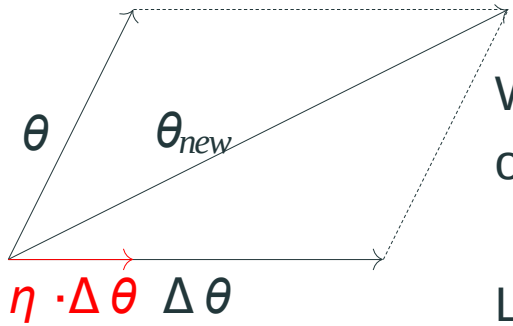
Let us be a bit conservative: move only by a small amount η

vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b



We moved in the direction of $\Delta\theta$

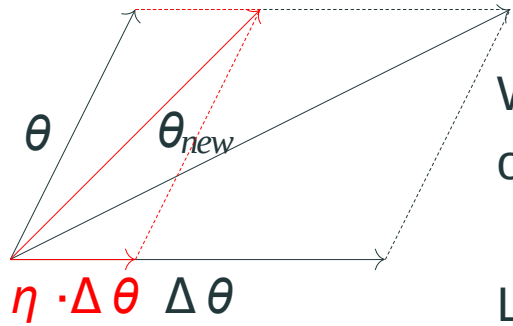
Let us be a bit conservative: move only by a small amount η

vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b



We moved in the direction of $\Delta\theta$

Let us be a bit conservative: move only by a small amount η

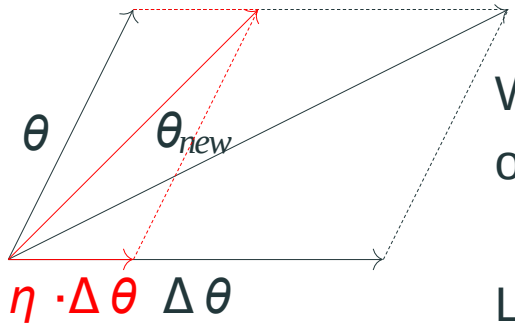
vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b

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



We moved in the direction of $\Delta\theta$

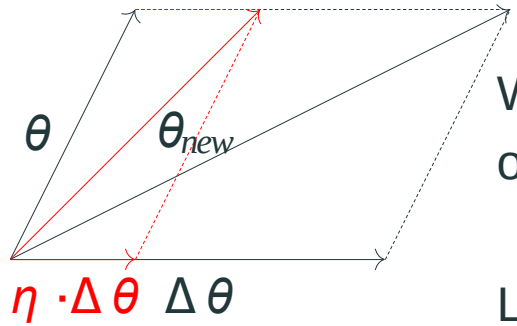
Let us be a bit conservative: move only by a small amount η

vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b



We moved in the direction of $\Delta\theta$

Let us be a bit conservative: move only by a small amount η

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

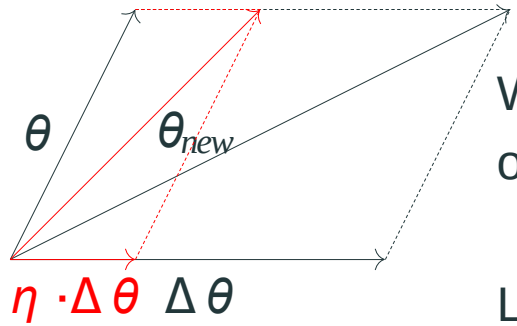
Question: What is the right $\Delta\theta$ to use ?

vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b



We moved in the direction of $\Delta\theta$

Let us be a bit conservative: move only by a small amount η

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

Question: What is the right $\Delta\theta$ to use ?

The answer comes from Taylor series

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

$$L\ (\theta + \eta u) = L\ (\theta) + \eta * u^T \nabla_\theta L\ (\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L\ (\theta) u + \frac{\eta^3}{3!} * ... + \frac{\eta^4}{4!} * ...$$

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

$$L\ (\theta + \eta u) = L\ (\theta) + \eta * u^T \nabla_\theta L\ (\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L\ (\theta) u + \frac{\eta^3}{3!} * ... + \frac{\eta^4}{4!} * ...$$

$$= L\ (\theta) + \eta * u^T \nabla_\theta L\ (\theta)\ [\eta \textit{ is typically small, so } \eta^2, \eta^3, .. \rightarrow 0]$$

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

$$L\ (\theta + \eta u) = L\ (\theta) + \eta * u^T \nabla_\theta L\ (\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L\ (\theta) u + \frac{\eta^3}{3!} * \ldots + \frac{\eta^4}{4!} * \ldots$$
$$= L\ (\theta) + \eta * u^T \nabla_\theta L\ (\theta)\ [\eta \textit{ is typically small, so } \eta^2, \eta^3, .. \rightarrow 0]$$

Note that the move (ηu) would be favorable only if,

$$L\ (\theta + \eta u) - L\ (\theta) < 0\ [\textit{i.e., if the new loss is less than the previous loss}]$$

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

$$L(\theta + \eta u) = L(\theta) + \eta * u^T \nabla_\theta L(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L(\theta) u + \frac{\eta^3}{3!} * \ldots + \frac{\eta^4}{4!} * \ldots$$
$$= L(\theta) + \eta * u^T \nabla_\theta L(\theta) \; [\eta \text{ is typically small, so } \eta^2, \eta^3, .. \rightarrow 0]$$

Note that the move (ηu) would be favorable only if,

$$L(\theta + \eta u) - L(\theta) < 0 \; [\textit{i.e., if the new loss is less than the previous loss}]$$

This implies,

$$u^T \nabla_\theta L(\theta) < 0$$

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$?

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$? Let us see....

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$? Let us see....

Let β be the angle between u and $\nabla_\theta L\ (\theta)$, then we know that,

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$? Let us see....

Let β be the angle between u and $\nabla_\theta L\ (\theta)$, then we know that,

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

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$? Let us see....

Let β be the angle between u and $\nabla_\theta L\ (\theta)$, then we know that,

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

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$? Let us see....

Let β be the angle between u and $\nabla_\theta L\ (\theta)$, then we know that,

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

multiply throughout by $k = ||u|| * ||\nabla_\theta L\ (\theta)||$

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

Okay, so we have,

$$u^T \nabla_\theta L\ (\theta) < 0$$

But, what is the range of $u^T \nabla_\theta L\ (\theta)$? Let us see....

Let β be the angle between u and $\nabla_\theta L\ (\theta)$, then we know that,

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

multiply throughout by $k = ||u|| * ||\nabla_\theta L\ (\theta)||$

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

Thus, $L\ (\theta + \eta u) - L\ (\theta) = u^T \nabla_\theta L\ (\theta) = k * cos(\beta)$ will be most negative when $cos(\beta) = -1$ *i.e.*, when β is 180°

Gradient Descent Rule

The direction u that we intend to move in should be at 180° w.r.t. the gradient

In other words, move in a direction opposite to the gradient

Parameter Update Equations

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

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

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

Gradient Descent Rule

The direction u that we intend to move in should be at 180° w.r.t. the gradient

In other words, move in a direction opposite to the gradient

Parameter Update Equations

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

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

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

So we now have a more principled way of moving in the w-b plane than our "guess work" algorithm

Let us create an algorithm from this rule ...

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

Let us create an algorithm from this rule ...

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

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

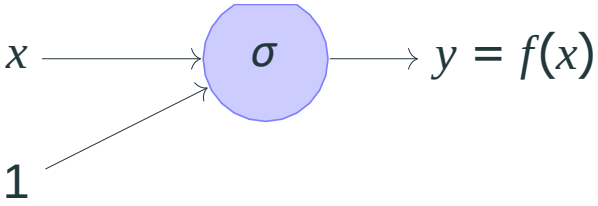
    $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;

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

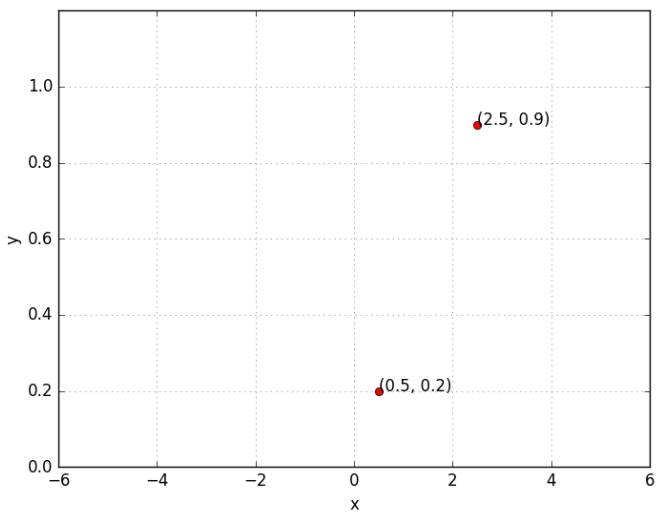
    $t \leftarrow t + 1$;

end

∇w and ∇b *are found using BACKPROPAGATION*

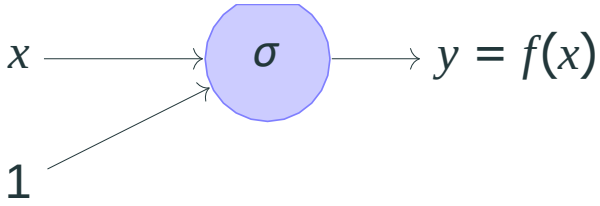


$$f(x) = \frac{1}{1+e^{-(w \cdot x+b)}}$$

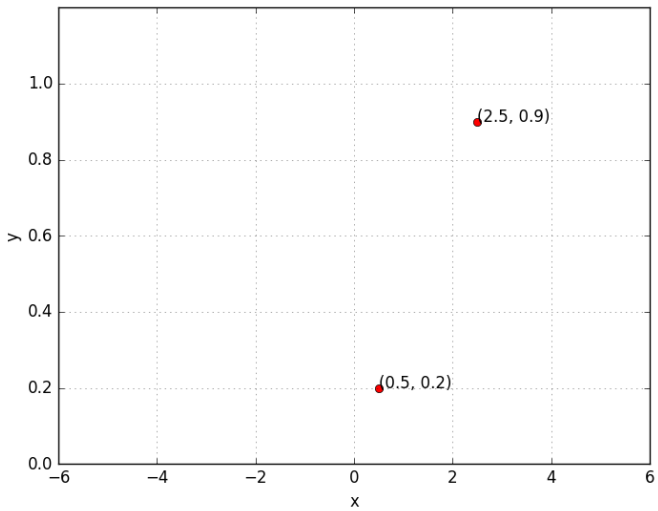


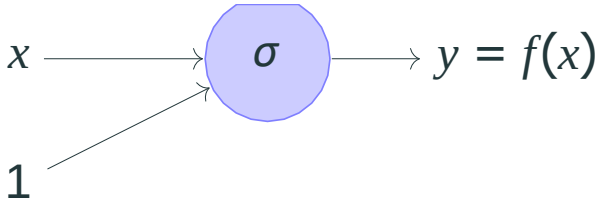
Let's assume there is only 1 point to fit (x, y)

$$L\ (w, b) = \frac{1}{2} * (f(x) - y)^2$$



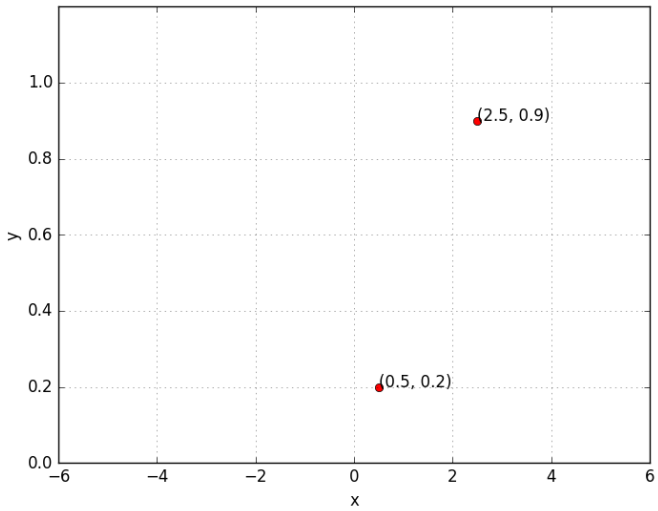
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

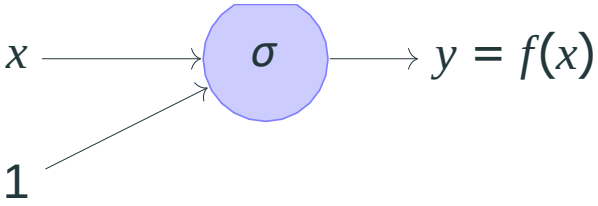




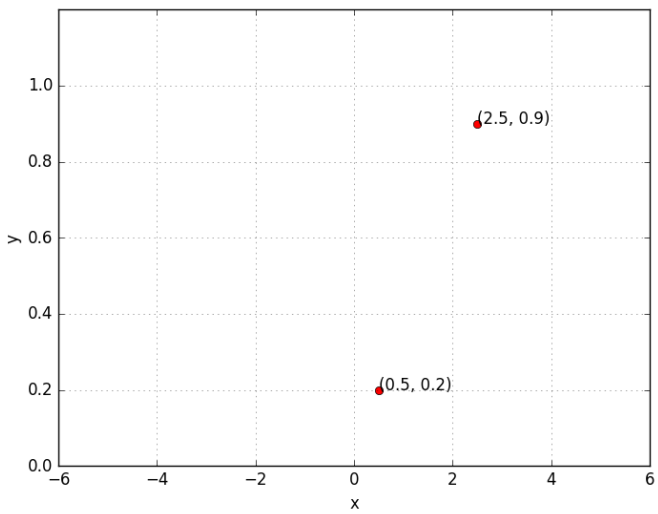
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

Let's assume there is only 1 point to fit (x, y)





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



Let's assume there is only 1 point to fit (x, y)

$$L\ (w, b) = \frac{1}{2} * (f(x) - y)^2$$

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

$$\nabla w = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$
$$= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)]$$
$$= (f(x) - y) * \frac{\partial}{\partial w}(f(x))$$
$$= (f(x) - y) * \frac{\partial}{\partial w} \frac{1}{1 + e^{-(wx+b)}}$$
$$= (f(x) - y) * f(x) * (1 - f(x)) * x$$

$$\frac{\partial}{\partial w} \frac{1}{1 + e^{-(wx+b)}}$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2} \frac{\partial}{\partial w}(e^{-(wx+b)}))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)}) \frac{\partial}{\partial w}(-(wx + b)))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x)$$
$$= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x)$$
$$= f(x) * (1 - f(x)) * x$$

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err
```

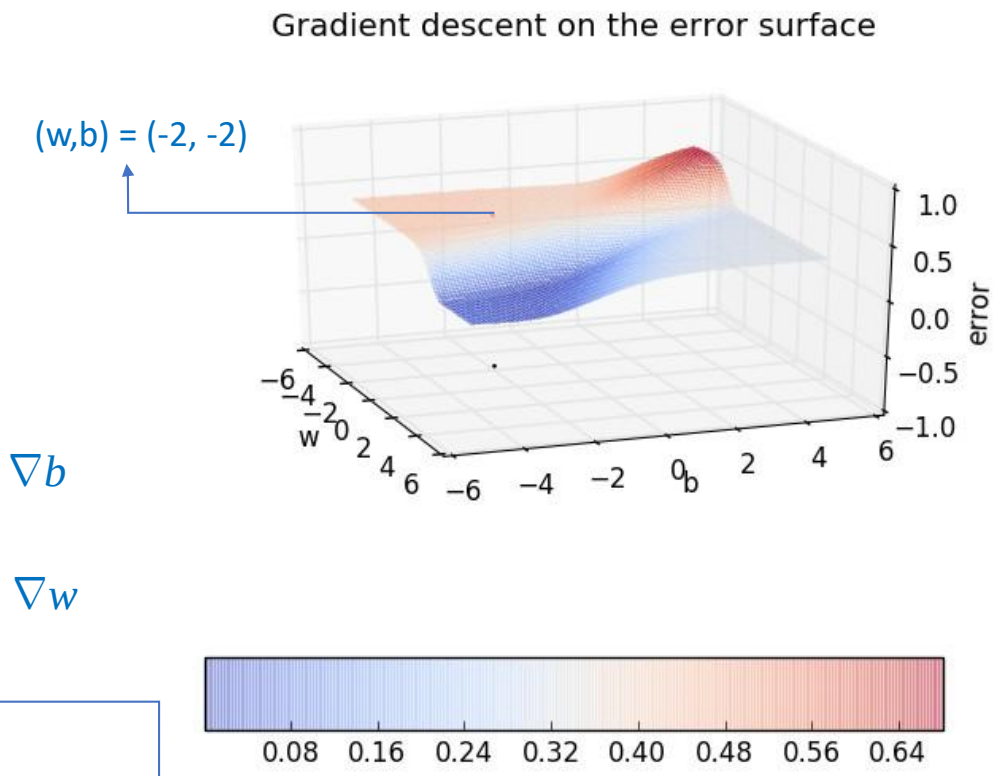
```
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)
```

∇b

```
def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x
```

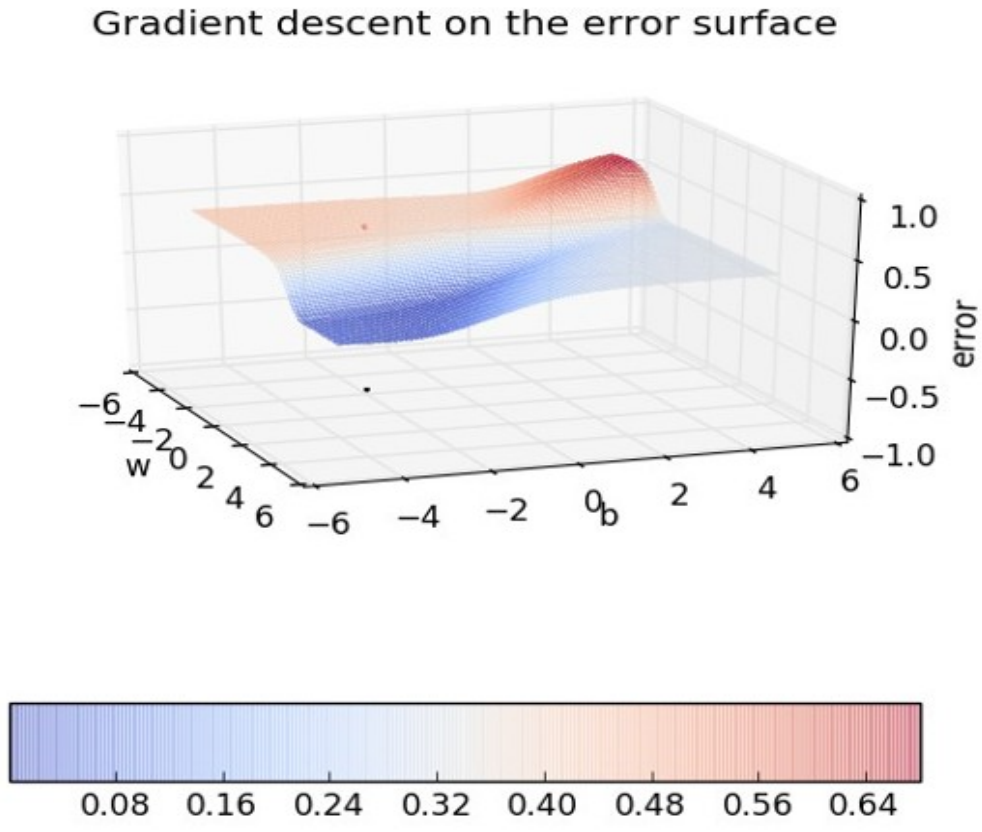
∇w

```
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```



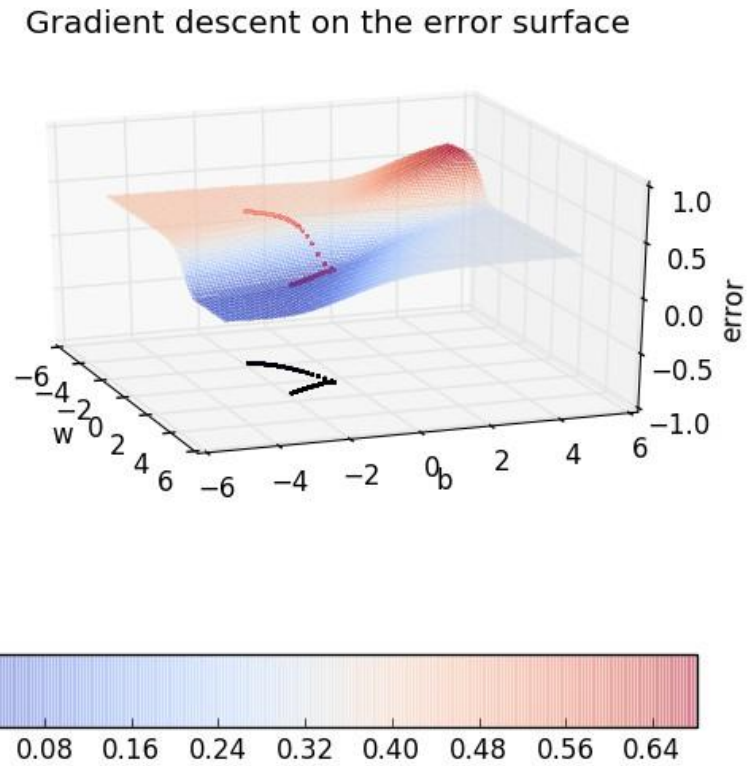
Let us start with w = b= -2 and learning rate = 1

Let's run the algorithm and see the movement in the error surface



Play the video and notice how the values of w and b automatically update for each iteration.
Also observe that the updates are slow in the flat regions and quick at the slopes
(a point of discussion with respect to optimizers at a later point of our class).

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

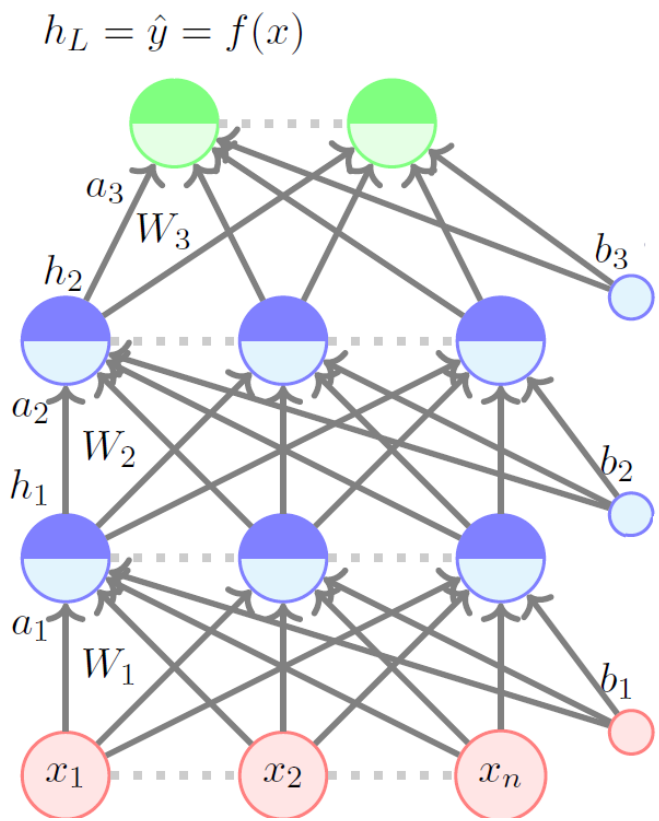


So we now have a more principled way of moving in the *w*-*b* plane than our “guess work” algorithm

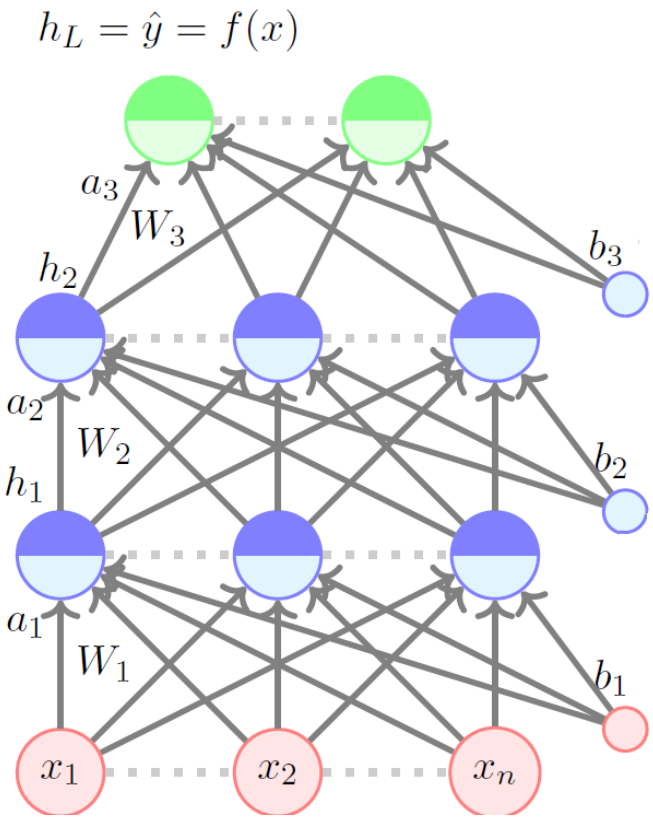
The Gradient Descent we saw was for single perceptron!

Let's now see how Gradient Descent works in a Multi Layered Network of Perceptron (or Feed Forward Neural Network) . . .

Feed Forward Neural Networks: Representation



Feed Forward Neural Networks: Representation



- The pre-activation at layer i is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

- The activation at layer i is given by

$$h_i(x) = g(a_i(x))$$

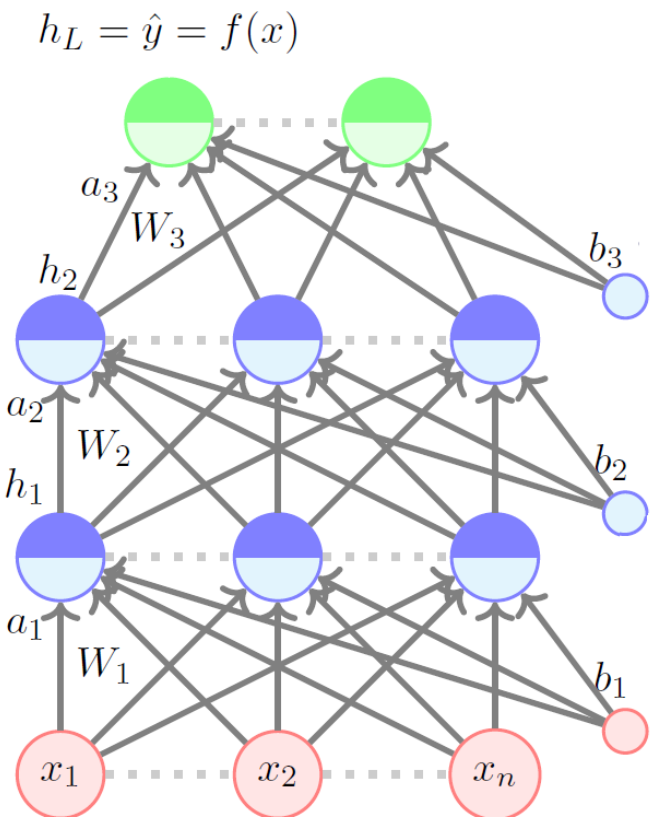
where g is called the activation function (for example, logistic, tanh, linear, *etc.*)

- The activation at the output layer is given by

$$f(x) = h_L(x) = O(a_L(x))$$

where O is the output activation function (for example, softmax, linear, *etc.*)

Feed Forward Neural Networks: Representation (Matrix Form)



Assume that:

- Input layer has n features.
- Hidden layer 1 has m_1 neurons.
- Hidden layer 2 has m_2 neurons.
- Output layer has k neurons.

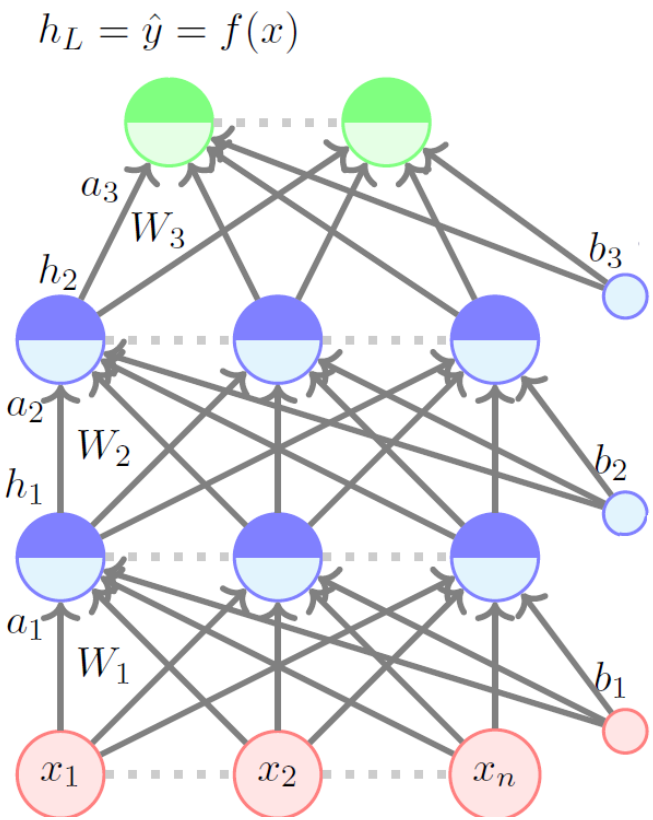
The equation for pre-activation at the hidden layer 1 is : $a_1 = b_1 + W_1 x$

The full-matrix representation of this equation is as follows:

$$\begin{pmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{1m_1} \end{pmatrix} = \begin{pmatrix} b_{11} \\ b_{12} \\ \vdots \\ b_{1m_1} \end{pmatrix} + \begin{pmatrix} W_{111} & W_{112} & \cdots & W_{11n} \\ W_{121} & W_{122} & \cdots & W_{12n} \\ \vdots & \vdots & \ddots & \vdots \\ W_{1m_11} & W_{1m_12} & \cdots & W_{1m_1n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

W_{ijk} - i is the layer number, j is the input neuron, and k is the output neuron.

Feed Forward Neural Networks: Representation (Matrix Form)



Assume that:

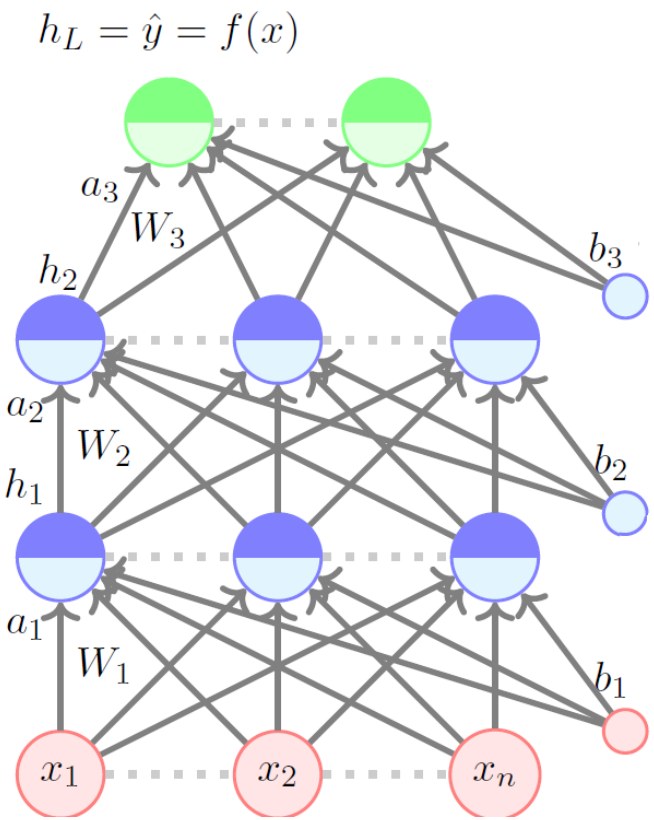
- Input layer has n features.
- Hidden layer 1 has m_1 neurons.
- Hidden layer 2 has m_2 neurons.
- Output layer has k neurons.

The equation for activation at the hidden layer 1 is : $h_1 = g(a_1)$

The full-matrix representation of this equation is as follows:

$$\begin{pmatrix} h_{11} \\ h_{12} \\ \vdots \\ h_{1m_1} \end{pmatrix} = g\left(\begin{pmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{1m_1} \end{pmatrix}\right)$$

Feed Forward Neural Networks: Representation (Matrix Form)



Assume that:

- Input layer has n features.
- Hidden layer 1 has m_1 neurons.
- Hidden layer 2 has m_2 neurons.
- Output layer has k neurons.

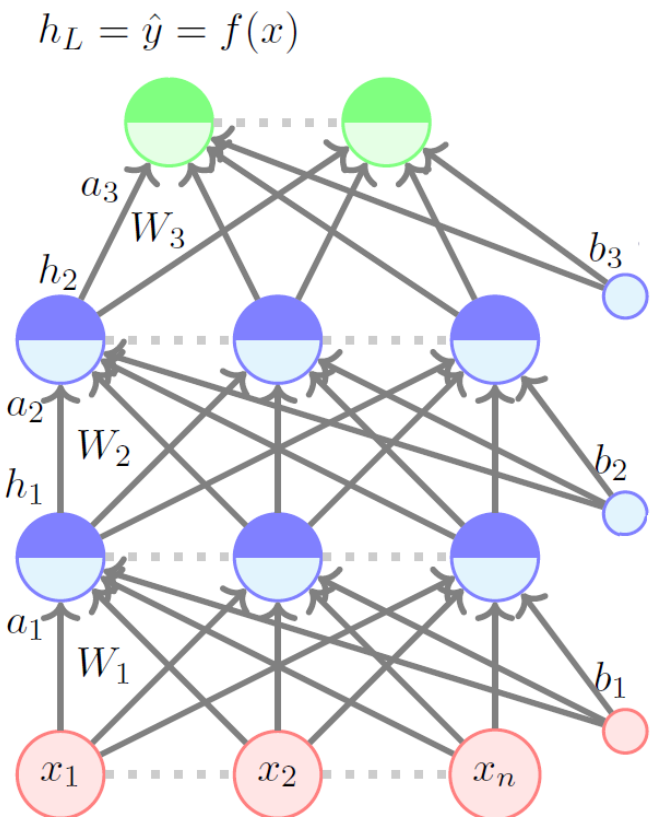
The equation for pre-activation at the hidden layer 2 is : $a_2 = b_2 + W_2 h_1$

The full-matrix representation of this equation is as follows:

$$\begin{pmatrix} a_{21} \\ a_{22} \\ \vdots \\ a_{2m_2} \end{pmatrix} = \begin{pmatrix} b_{21} \\ b_{22} \\ \vdots \\ b_{2m_2} \end{pmatrix} + \begin{pmatrix} W_{211} & W_{212} & \cdots & W_{21m_1} \\ W_{221} & W_{222} & \cdots & W_{22m_1} \\ \vdots & \vdots & \ddots & \vdots \\ W_{2m_21} & W_{2m_22} & \cdots & W_{2m_2m_1} \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ \vdots \\ h_{1m_1} \end{pmatrix}$$

W_{ijk} - i is the layer number, j is the input neuron, and k is the output neuron.

Feed Forward Neural Networks: Representation (Matrix Form)



Assume that:

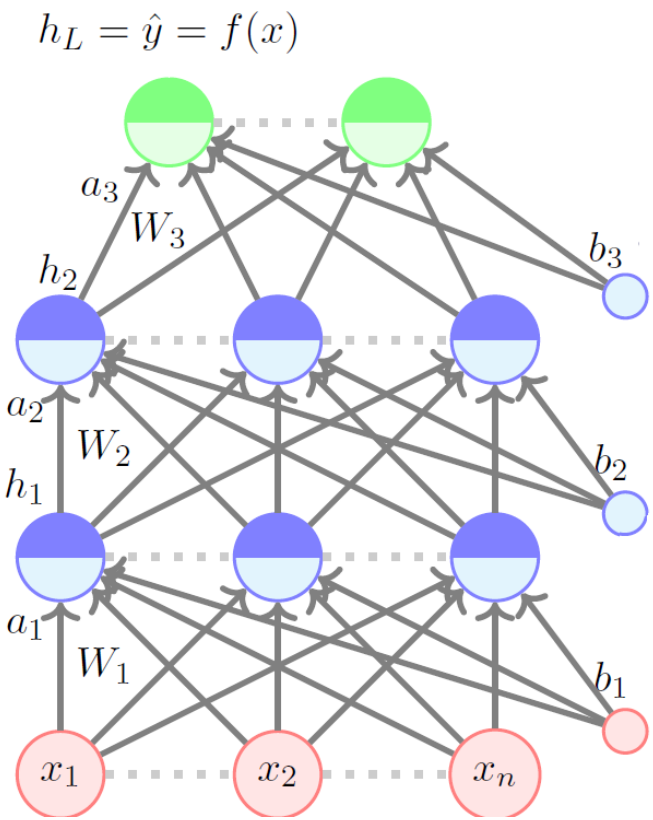
- Input layer has n features.
- Hidden layer 1 has m_1 neurons.
- Hidden layer 2 has m_2 neurons.
- Output layer has k neurons.

The equation for activation at the hidden layer 2 is : $h_2 = g(a_2)$

The full-matrix representation of this equation is as follows:

$$\begin{pmatrix} h_{21} \\ h_{22} \\ \vdots \\ h_{2m_2} \end{pmatrix} = g\left(\begin{pmatrix} a_{21} \\ a_{22} \\ \vdots \\ a_{2m_2} \end{pmatrix}\right)$$

Feed Forward Neural Networks: Representation (Matrix Form)



Assume that:

- Input layer has n features.
- Hidden layer 1 has m_1 neurons.
- Hidden layer 2 has m_2 neurons.
- Output layer has k neurons.

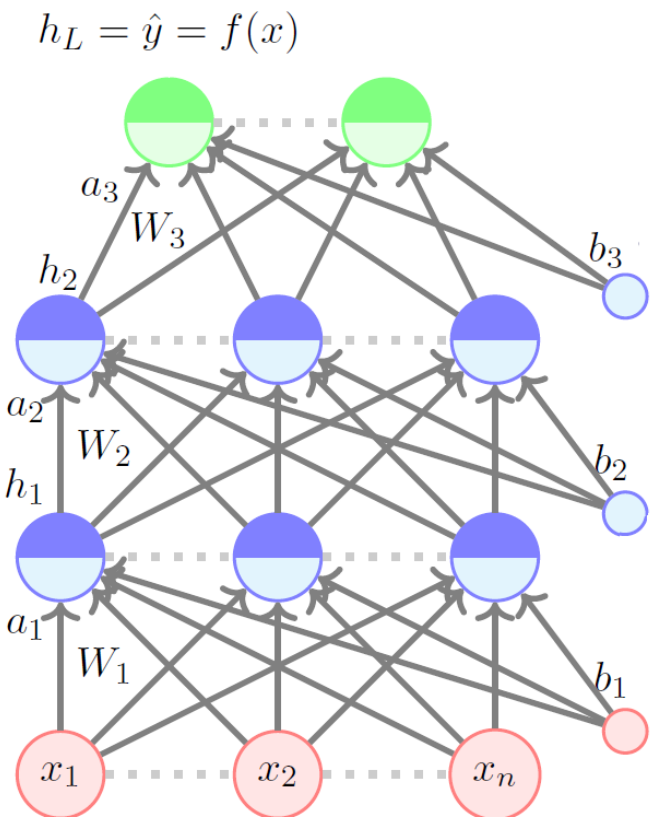
The equation for pre-activation at the output layer is : $a_3 = b_3 + W_3 h_2$

The full-matrix representation of this equation is as follows:

$$\begin{pmatrix} a_{31} \\ a_{32} \\ \vdots \\ a_{3k} \end{pmatrix} = \begin{pmatrix} b_{31} \\ b_{32} \\ \vdots \\ b_{3k} \end{pmatrix} + \begin{pmatrix} W_{311} & W_{312} & \cdots & W_{31m_2} \\ W_{321} & W_{322} & \cdots & W_{32m_2} \\ \vdots & \vdots & \ddots & \vdots \\ W_{3k1} & W_{3k2} & \cdots & W_{3km_2} \end{pmatrix} \begin{pmatrix} h_{21} \\ h_{22} \\ \vdots \\ h_{2m_2} \end{pmatrix}$$

W_{ijk} - i is the layer number, j is the input neuron, and k is the output neuron.

Feed Forward Neural Networks: Representation (Matrix Form)



Assume that:

- Input layer has n features.
- Hidden layer 1 has m_1 neurons.
- Hidden layer 2 has m_2 neurons.
- Output layer has k neurons.

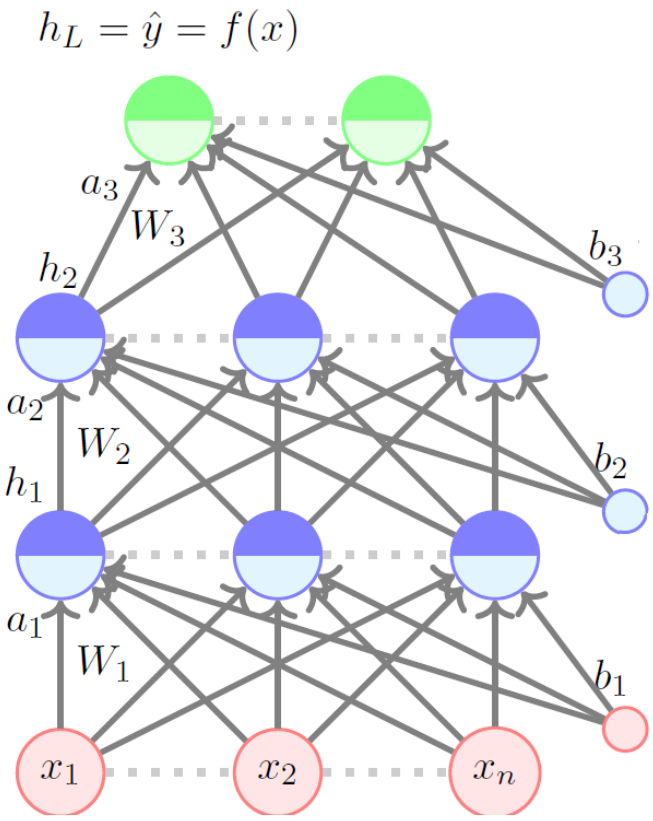
The equation for activation at the output layer is :

$$h_3 = O(a_3)$$

The full-matrix representation of this equation is as follows:

$$\begin{pmatrix} h_{31} \\ h_{32} \\ \vdots \\ h_{3k} \end{pmatrix} = O\left(\begin{pmatrix} a_{31} \\ a_{32} \\ \vdots \\ a_{3k} \end{pmatrix}\right)$$

Feed Forward Neural Networks: Representation



- **Data:** $\{x_i, y_i\}_{i=1}^{N}$
- **Model:**

$$\hat{y}_i = f(x_i) = O(W_3 g(W_2 g(W_1 x + b_1) + b_2) + b_3)$$

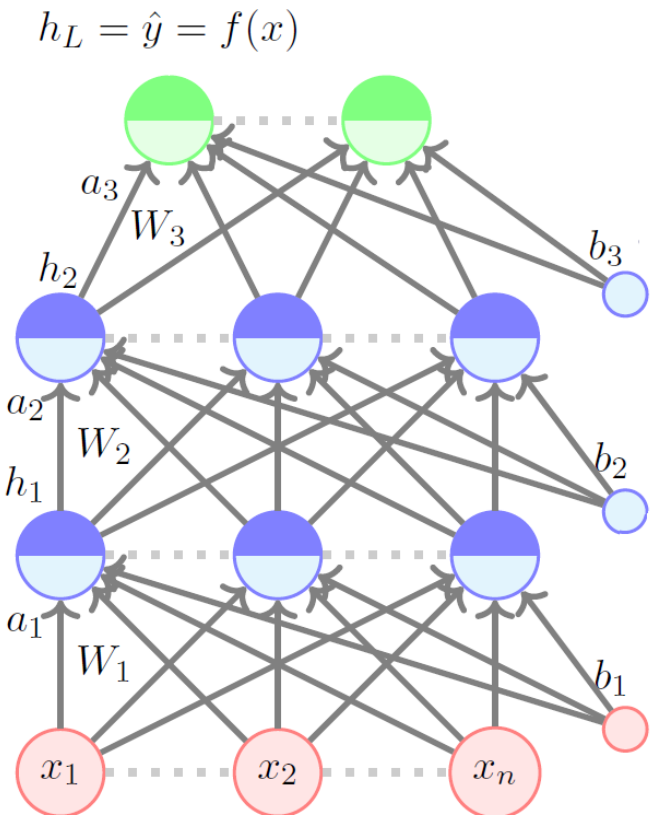
- **Parameters:**

$$\theta = W_1, .., W_L, b_1, b_2, ..., b_L (L = 3)$$

- **Algorithm:** Gradient Descent with Back-propagation
- **Objective/Loss/Error function:** Say,

$$min \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} (\hat{y}_{ij} - y_{ij})^2$$

Feed Forward Neural Networks: Gradient Descent



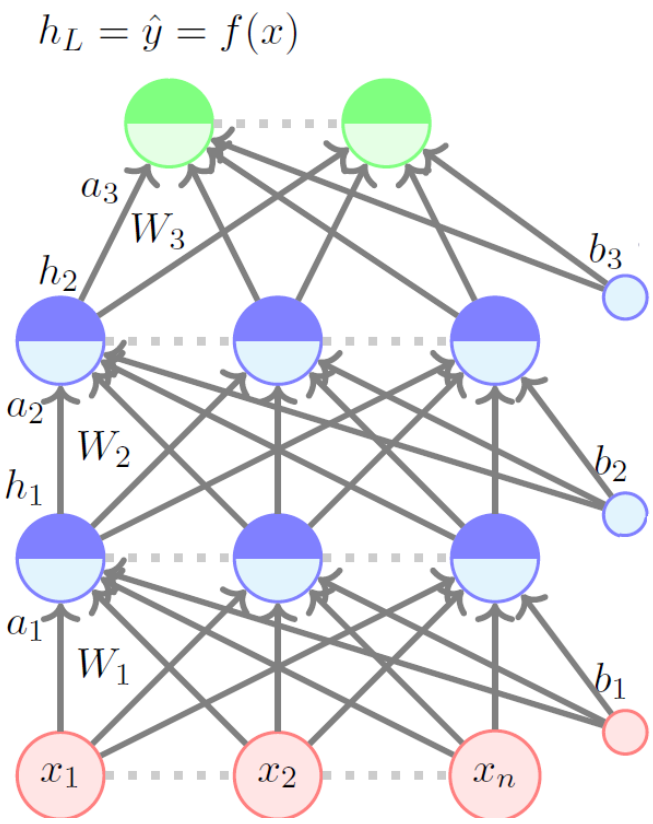
Our old Gradient Descent Algorithm with one neuron

Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
$Initialize$ $\theta_0 = [w_0, b_0]$;
while t++ $< max\_iterations$ **do**
  $\theta_{t+1} \leftarrow \theta_t - \eta \nabla \theta_t$;
end

where $\nabla \theta_t = \left[\frac{\partial \mathscr{L}(\theta)}{\partial w_t}, \frac{\partial \mathscr{L}(\theta)}{\partial b_t}\right]^T$

Feed Forward Neural Networks: Gradient Descent



Gradient Descent Algorithm for the MLP Model

Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;
while t++ $< max\_iterations$ **do**
  $\theta_{t+1} \leftarrow \theta_t - \eta \nabla \theta_t$;
end

where $\nabla \theta_t = \left[\frac{\partial \mathscr{L}(\theta)}{\partial W_{1,t}}, .., \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,t}}, \frac{\partial \mathscr{L}(\theta)}{\partial b_{1,t}}, .., \frac{\partial \mathscr{L}(\theta)}{\partial b_{L,t}}\right]^T$

Feed Forward Neural Networks: Gradient Descent

$$\begin{bmatrix} \frac{\partial \mathscr{L}(\theta)}{\partial W_{111}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{11n}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{211}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{21n}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,11}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,1k}} & \frac{\partial \mathscr{L}(\theta)}{\partial b_{11}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial b_{L1}} \\ \frac{\partial \mathscr{L}(\theta)}{\partial W_{121}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{12n}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{221}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{22n}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,21}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,2k}} & \frac{\partial \mathscr{L}(\theta)}{\partial b_{12}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial b_{L2}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathscr{L}(\theta)}{\partial W_{1n1}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{1nn}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{2n1}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{2nn}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,n1}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,nk}} & \frac{\partial \mathscr{L}(\theta)}{\partial b_{1n}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial b_{Lk}} \end{bmatrix}$$

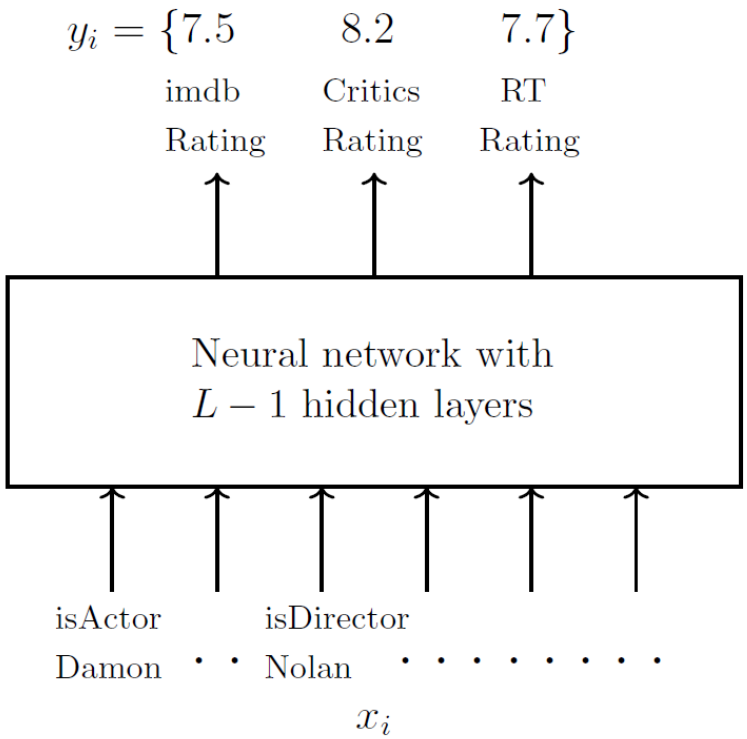
$\nabla\theta$ is thus composed of
$\nabla W_1, \nabla W_2, ...\nabla W_{L-1} \in \mathbb{R}^{n\times n}, \nabla W_L \in \mathbb{R}^{n\times k}$,
$\nabla b_1, \nabla b_2, ..., \nabla b_{L-1} \in \mathbb{R}^n$ and $\nabla b_L \in \mathbb{R}^k$

We need to answer two questions:

- How to choose the loss function?
- How to compute $\nabla\theta$?

Feed Forward Neural Networks: Introduction

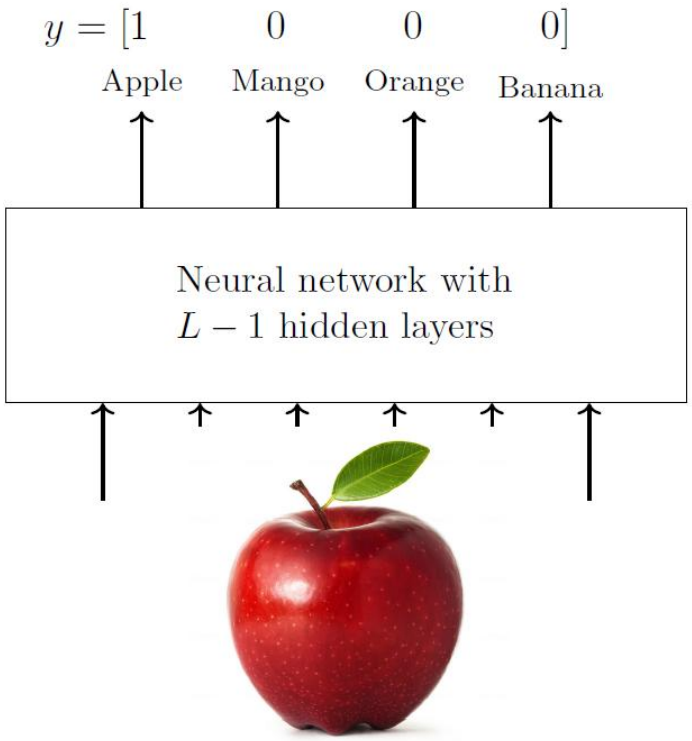
The choice of the loss function depends on the problem at hand



Squared Error Loss

$$\mathscr{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{3} (\hat{y}_{ij} - y_{ij})^2$$

Feed Forward Neural Networks: Loss and Activation Functions



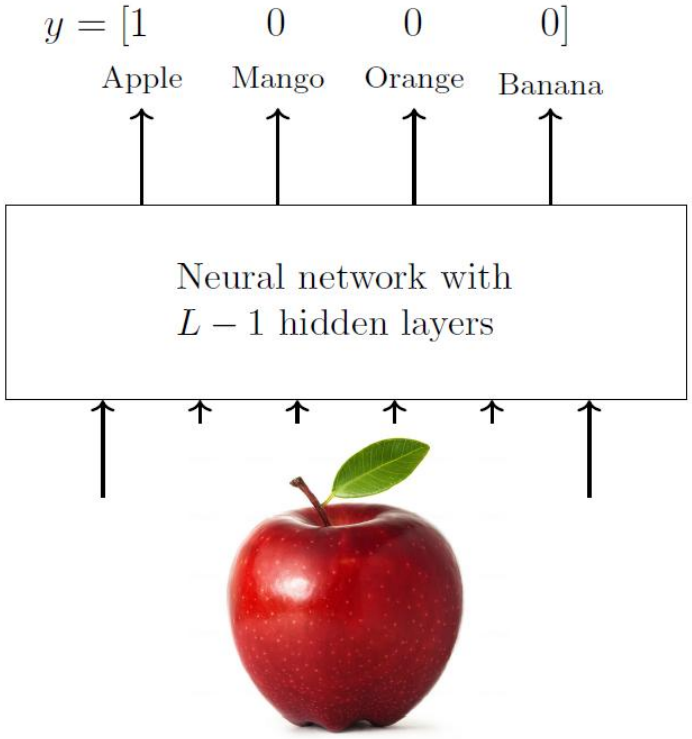
Softmax Activation

$$a_L = W_L h_{L-1} + b_L$$

$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{L,j}}}{\sum_{i=1}^{k} e^{a_{L,i}}}$$

$O(a_L)_j$ is the j^{th} element of $\hat{y}$ and $a_{L,j}$ is the j^{th} element of the vector a_L.

Feed Forward Neural Networks: Loss and Activation Functions



Cross-entropy

$$\mathscr{L}(\theta) = -\sum_{c=1}^{k} y_c \log \hat{y}_c$$

Notice that

$$\begin{aligned} y_c &= 1 \quad \text{if } c = \ell \text{ (the true class label)} \\ &= 0 \quad \text{otherwise} \\ \therefore \quad \mathscr{L}(\theta) &= -\log \hat{y}_\ell \end{aligned}$$

Feed Forward Neural Networks: Loss and Activation Functions

	Outputs	
	Real Values	Probabilities
Output Activation	Linear	Softmax
Loss Function	Squared Error	Cross Entropy

Loss Functions

- **Regression Loss Functions**
  - model predicts a corresponding output value
  - **Output Layer Configuration**: One node with a linear activation unit.
  - **Loss Function**: Mean Squared Error (MSE).
- **Classification Loss Functions**
  - neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging
  - **Binary Classification Problem**
    - A problem where you classify an example as belonging to one of two classes.
    - The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.
    - **Output Layer Configuration**: One node with a sigmoid activation unit.
    - **Loss Function**: Cross-Entropy, also referred to as Logarithmic loss.
  - **Multi-Class Classification Problem**
    - A problem where you classify an example as belonging to one of more than two classes.
    - The problem is framed as predicting the likelihood of an example belonging to each class.
    - **Output Layer Configuration**: One node for each class using the softmax activation function.
    - **Loss Function**: Cross-Entropy, also referred to as Logarithmic loss.

Regression Loss Functions

- **Mean Squared Error (MSE)**
  - values with a large error are penalized.
  - is a convex function with a clearly defined global minimum
  - Can be used in **gradient descent optimization** to set the weight values
  - Very sensitive to outliers , will significantly increase the loss.
- **Mean Absolute Error (MAE)**
  - used in cases when the training data has a large number of outliers
    as the average distance approaches 0, gradient descent optimization will not work
- **Huber Loss**
  - Based on absolute difference between the actual and predicted value and threshold value, δ
  - Is quadratic when error is smaller than δ but linear when error is larger than δ

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|$$

$$Huber\ Loss = \begin{cases} \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 & |y^{(i)} - \hat{y}^{(i)}| \le \delta \\ \frac{1}{n}\sum_{i=1}^{n}\delta(|y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2}\delta) & |y^{(i)} - \hat{y}^{(i)}| > \delta \end{cases}$$

Classification Loss Functions

- **Binary Cross-Entropy/Log Loss**
  - Compares the actual value (0 or 1) with the probability that the input aligns with that category
    - *p(i)* = probability that the category is 1
    - 1 — *p(i)* = probability that the category is 0)

$$CE\ Loss = \frac{1}{n}\sum_{i=1}^{N} -(y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)$$

- **Categorical Cross-Entropy Loss**
  - In cases where the number of classes is greater than two

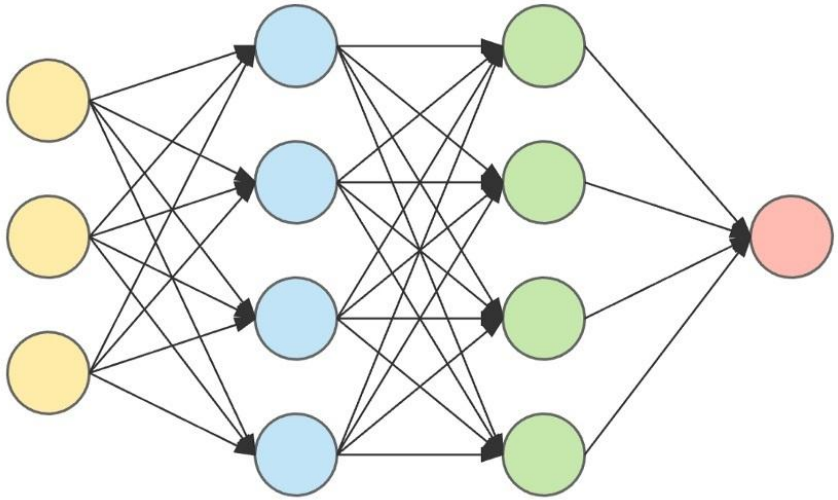
$$CE\ Loss = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} \cdot log(p_{ij})$$

Feed Forward Neural Networks

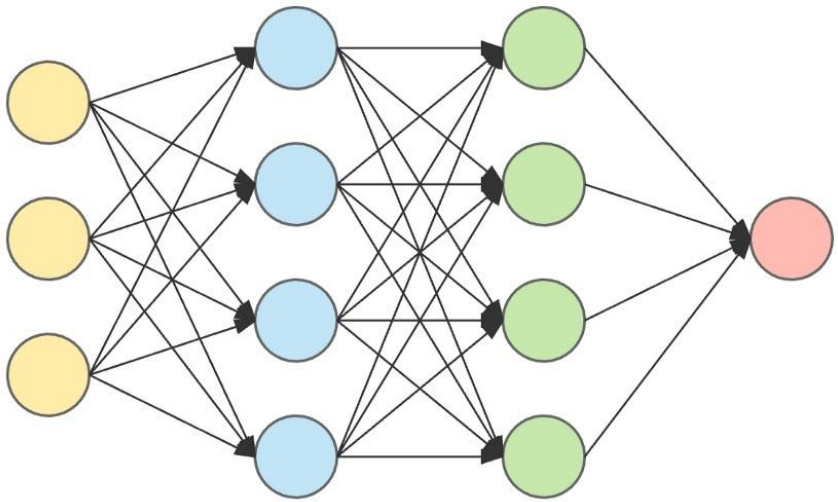
We need to answer two questions:

- How to choose the loss function?
- How to compute $\nabla\theta$? (Backpropagation)

Backpropagation : Intuition

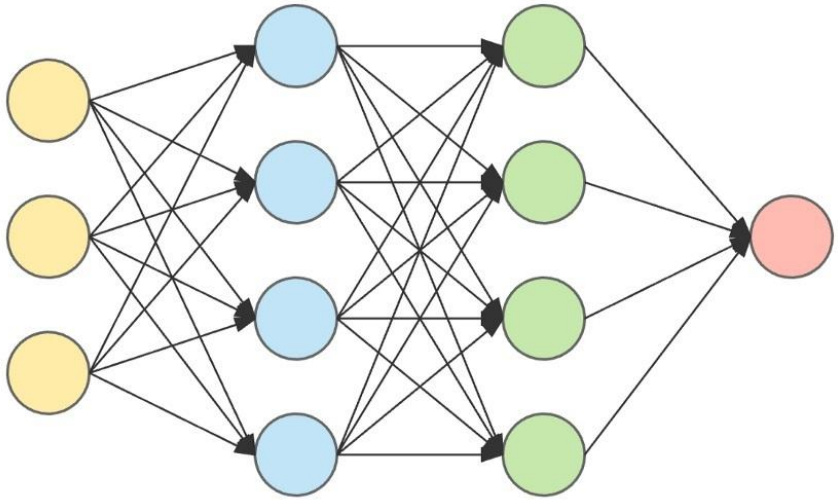


Backpropagation : Intuition



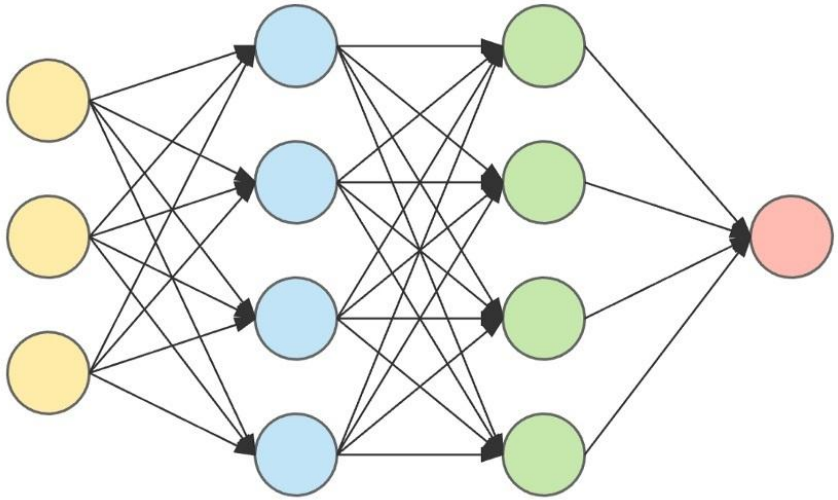
For simplicity, lets represent the above network as follows:

Backpropagation : Intuition

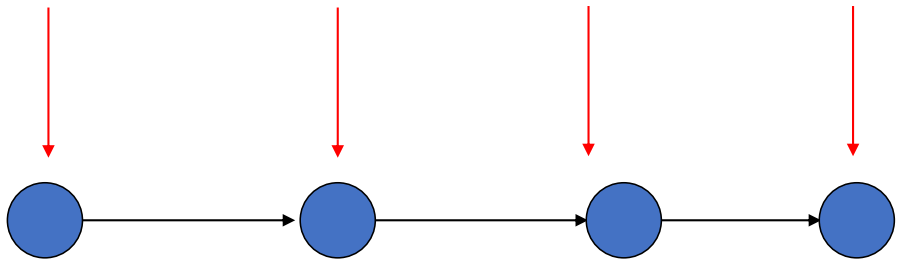


For simplicity, lets represent the above network as follows:

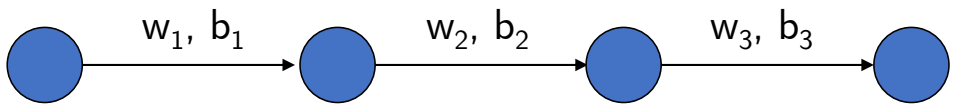
Backpropagation : Intuition



For simplicity, lets represent the above network as follows:

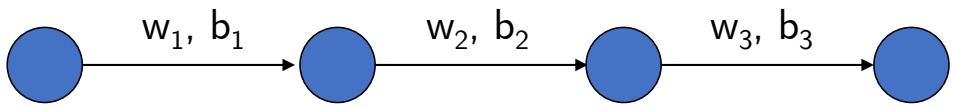


Backpropagation : Intuition



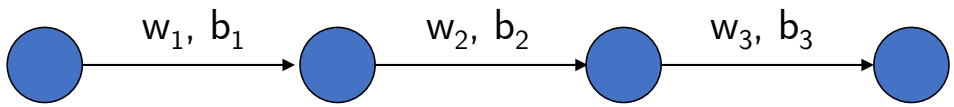
Backpropagation : Intuition

The Loss function: $L(w_1, b_1, w_2, b_2, w_3, b_3)$



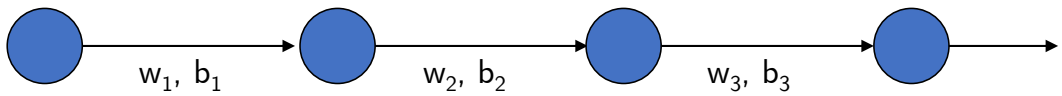
Backpropagation : Intuition

The Loss function: $L(w_1, b_1, w_2, b_2, w_3, b_3)$

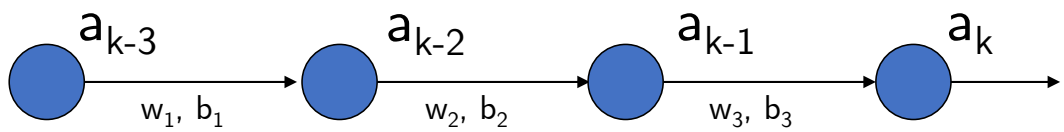


By how much should the parameters be changed to make an efficient decrease in the loss L ?

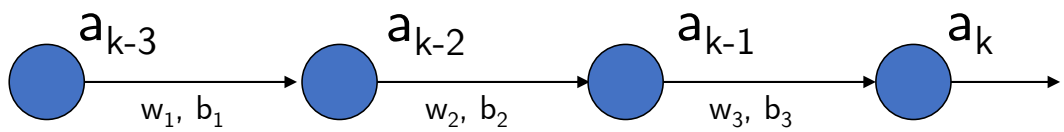
Backpropagation : Intuition



Backpropagation : Intuition

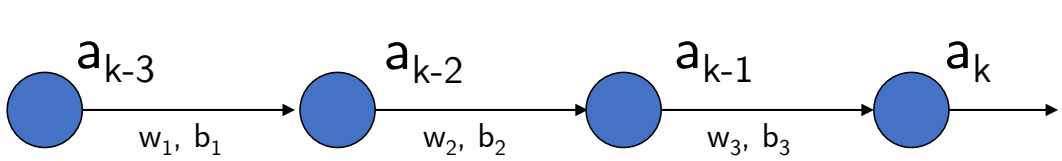


Backpropagation : Intuition



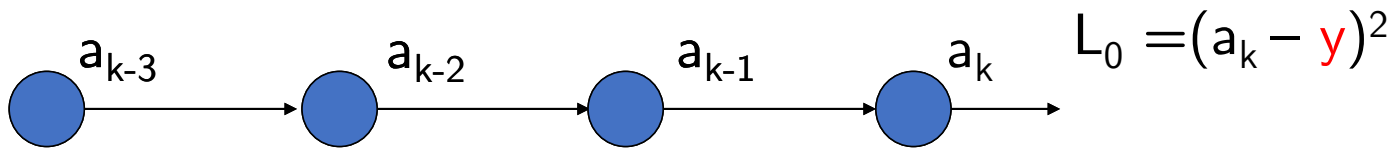
$$a_k = \sigma(w_k\, a_{k-1} + b_k)$$

Backpropagation : Intuition



$$a_k = \sigma(w_k\, a_{k-1} + b_k)$$

Backpropagation : Intuition

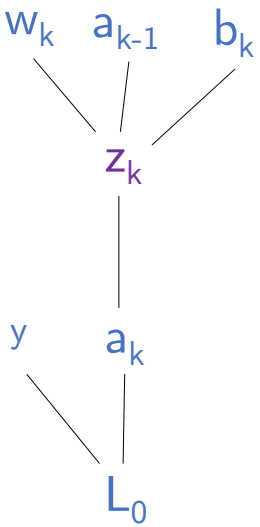


$$a_k = \sigma(w_k\, a_{k-1} + b_k)$$

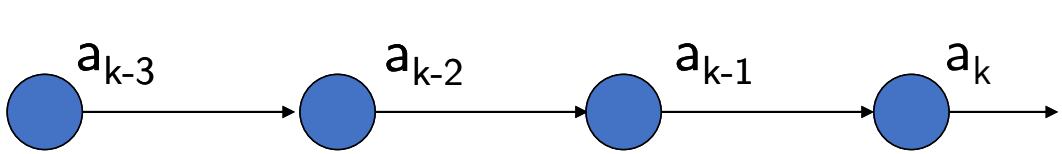
Now, if $z_k = w_k\, a_{k-1} + b_k$

Then, $a_k = \sigma(z_k)$

Backpropagation : Intuition



Dependency graph



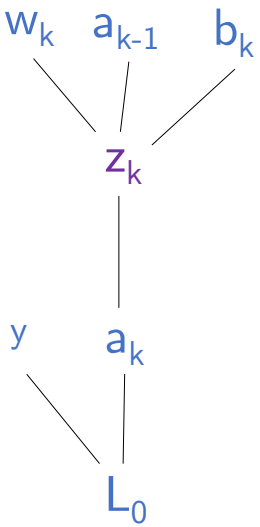
$$a_k = \sigma(w_k\, a_{k-1} + b_k)$$

Now, if $z_k = w_k\, a_{k-1} + b_k$

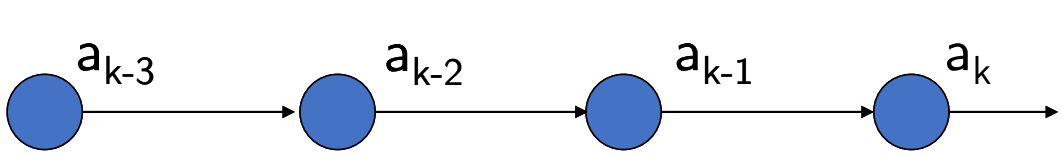
Then, $a_k = \sigma(z_k)$

Backpropagation : Intuition

Aim is to compute : $\frac{\partial L_0}{\partial w_k}$



Dependency graph



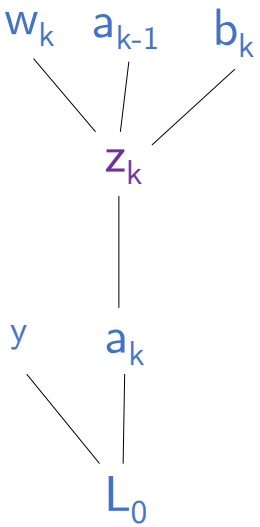
$a_k = \sigma(w_k\, a_{k-1} + b_k)$

Now, if $z_k = w_k\, a_{k-1} + b_k$

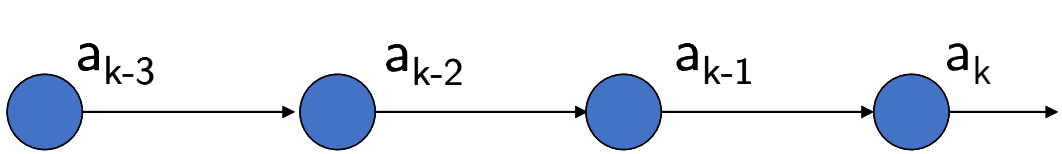
Then, $a_k = \sigma(z_k)$

Backpropagation : Intuition

As there is a dependency, we need to apply chain rule ➡ $\frac{\partial L_0}{\partial w_k} = \frac{\partial z_k}{\partial w_k}\frac{\partial a_k}{\partial z_k}\frac{\partial L_0}{\partial a_k}$



Dependency graph



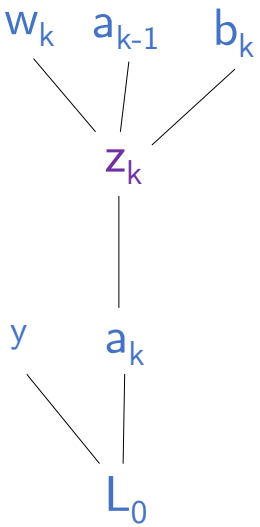
$a_k = \sigma(w_k\, a_{k-1} + b_k)$

Now, if $z_k = w_k\, a_{k-1} + b_k$

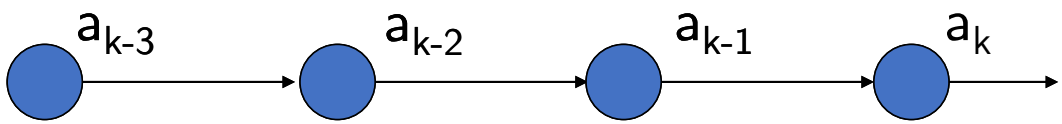
Then, $a_k = \sigma(z_k)$

Backpropagation : Intuition

$$\frac{\partial L_0}{\partial w_k} = \frac{\partial z_k}{\partial w_k}\frac{\partial a_k}{\partial z_k}\frac{\partial L_0}{\partial a_k} = a_{k-1}\,\sigma'(z_k) * 2*(a_k - y)$$



Dependency graph



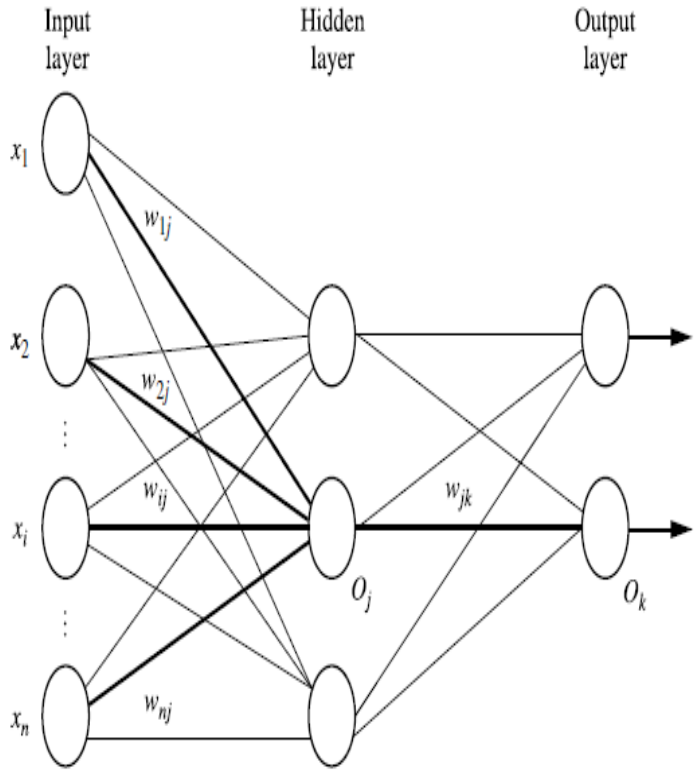
$$C_0 = (a_k - y)^2$$

$$a_k = \sigma(w_k\, a_{k-1} + b_k)$$

Now, if $z_k = w_k\, a_{k-1} + b_k$

Then, $a_k = \sigma(z_k)$

Back Propagation Algorithm : Simple MLP Example



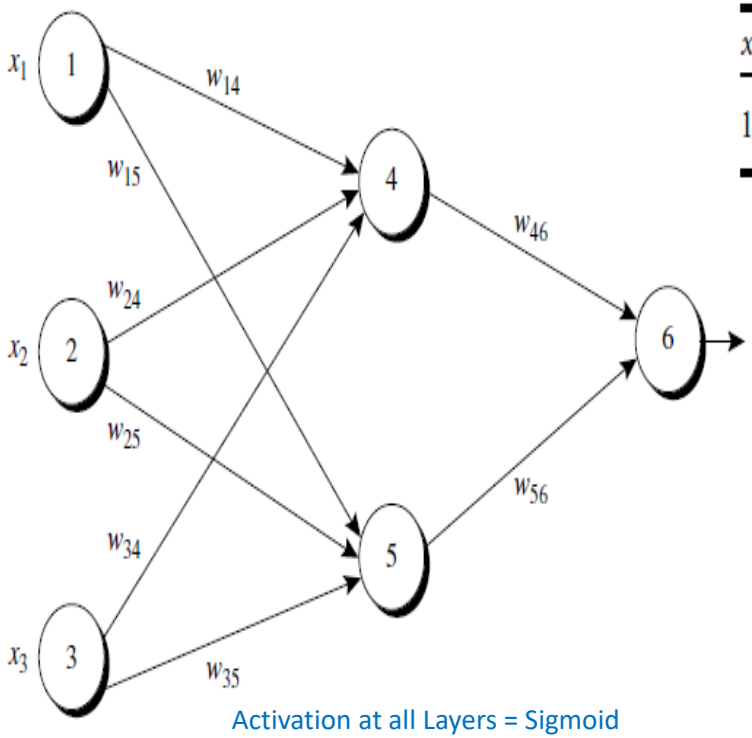
Back Propagation Algorithm : Simple MLP Example

D: dataset consisting f training tuples
l: learning rate

(1) Initialize all weights and biases in *network*;
(2) while terminating condition is not satisfied {
(3) for each training tuple X in D {
(4) // Propagate the inputs forward:
(5) for each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) for each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) for each unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) for each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15) for each weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) for each bias θ_j in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }

Jiawei Han and Micheline Kamber, "Data Mining Concepts And Techniques", 3rd Edition, Morgan Kauffmann

Back Propagation Algorithm : Simple MLP Example



Activation at all Layers = Sigmoid

Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

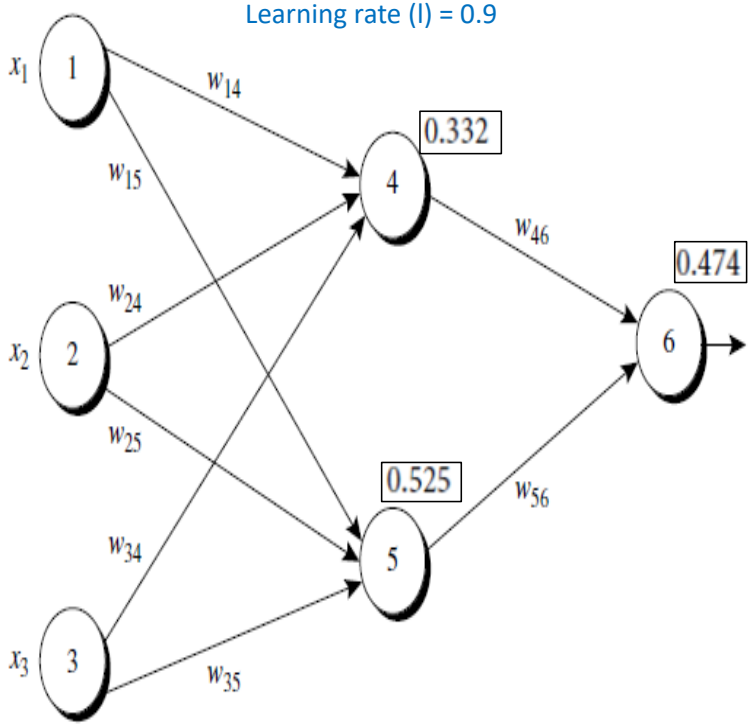
Net Input and Output Calculations

Unit, j	*Net Input, I_j*	*Output, O_j*
4		
5		
6		

Target value (T_j) = 1

Learning rate (l) = 0.9

Output neuron $Err_j = O_j(1 - O_j)(T_j - O_j),$ hidden neuron $Err_j = O_j(1 - O_j)\sum_k Err_k w_{jk}$



Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Calculation of the Error at Each Node

Unit, j	*Err_j*
6	
5	
4	

$$\Delta w_{ij} = (l) Err_j O_i.$$

$$w_{ij} = w_{ij} + \Delta w_{ij}.$$

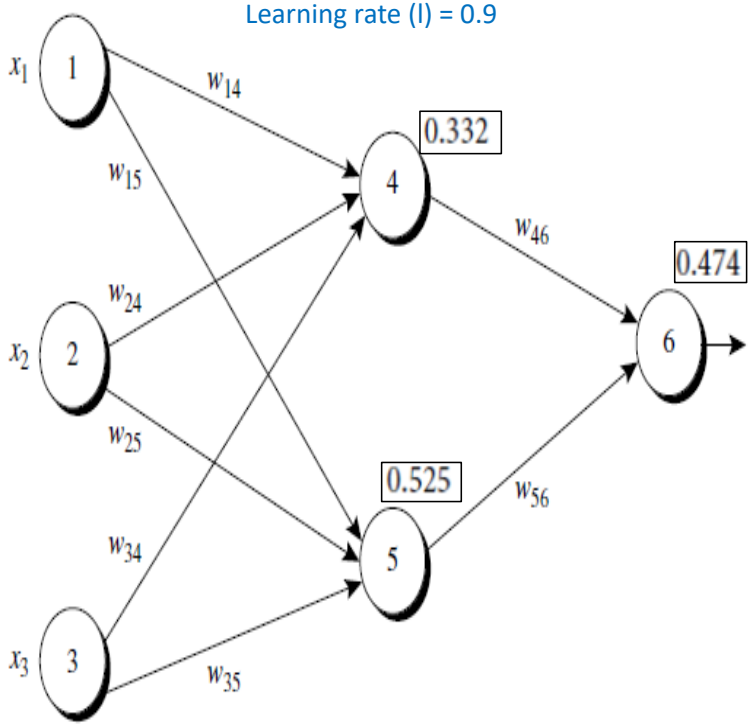
Calculations for Weight and Bias Updating

Weight or Bias	*New Value*
w_{46}	
w_{56}	
w_{14}	
w_{15}	
w_{24}	
w_{25}	
w_{34}	
w_{35}	
θ_6	
θ_5	
θ_4	

Target value (T_j) = 1

Learning rate (l) = 0.9

Output neuron $Err_j = O_j(1 - O_j)(T_j - O_j),$ hidden neuron $Err_j = O_j(1 - O_j)\sum_k Err_k w_{jk}$



Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Calculation of the Error at Each Node

Unit, j	*Err_j*
6	$(0.474)(1 - 0.474)(1 - 0.474) = 0.1311$
5	$(0.525)(1 - 0.525)(0.1311)(-0.2) = -0.0065$
4	$(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$

$$\Delta w_{ij} = (l) Err_j O_i.$$

$$w_{ij} = w_{ij} + \Delta w_{ij}.$$

Calculations for Weight and Bias Updating

Weight or Bias	*New Value*
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

Algorithm: Gradient Descent + Backpropagation

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;

Algorithm: Gradient Descent + Backpropagation

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;

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

end

Algorithm: Gradient Descent + Backpropagation

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;

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

  $h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, \hat{y} = forward\_propagation(\theta_t)$;

end

Algorithm: Gradient Descent + Backpropagation

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;

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

$h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, \hat{y} = forward\_propagation(\theta_t)$;

$\nabla\theta_t = backward\_propagation(h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, y, \hat{y})$;

end

Algorithm: Gradient Descent + Backpropagation

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;

while $t{+}{+} < max\_iterations$ **do**

$h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, \hat{y} = forward\_propagation(\theta_t)$;

$\nabla\theta_t = backward\_propagation(h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, y, \hat{y})$;

$\theta_{t+1} \leftarrow \theta_t - \eta\nabla\theta_t$;

end

Let's look at some activation functions (used at the hidden layers) . .

Activation Functions

- To make the network robust use of **Activation or Transfer functions**.
- Activations functions introduce non-linear properties in the neural networks.
- A good Activation function has the following properties:
- **Monotonic Function:**
  - should be either entirely non-increasing or non-decreasing.
  - If not monotonic then increasing the neuron's weight might cause it to have less influence on reducing the error of the cost function.
- **Differential:**
  - mathematically means the change in y with respect to change in x.
  - should be differential because we want to calculate the change in error with respect to given weights at the time of gradient descent.
- **Quickly Converging:**
  - Should reach its desired value fast.

Popular activation functions

Considering $z = \sum_{i=0}^{n} w_i x_i$

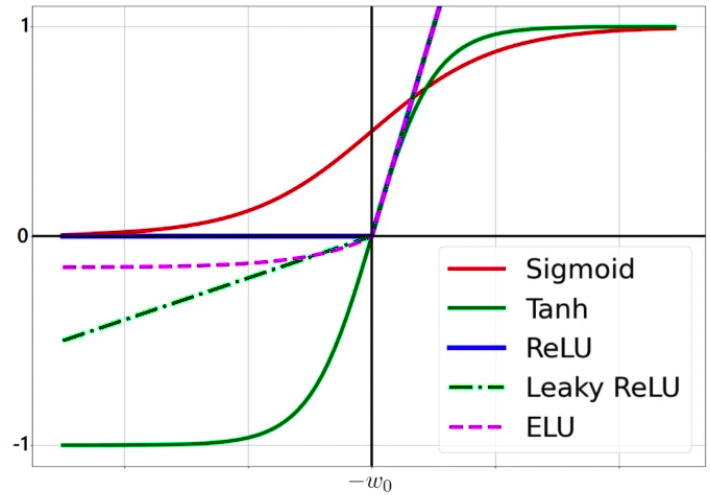
Sigmoid: $y = \frac{1}{1+e^{-z}}$

Tanh: $y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Rectified Linear Unit (ReLU): $y = \max(0, z)$

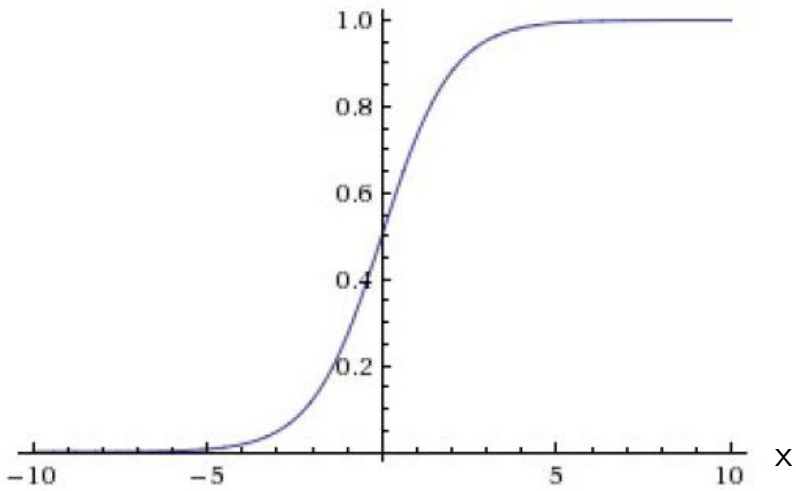
Leaky ReLU: $y = \max(\alpha z, z), \alpha \in (0, 1)$

Exponential Linear Unit (ELU):
$y = \max(\alpha(e^z - 1), z)$, where $\alpha > 0$



Activation functions : Sigmoid

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



Sigmoid

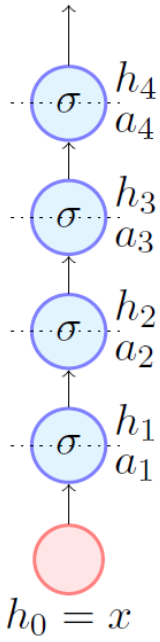
Gradient of sigmoid function:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Problems with Sigmoid:

- Cause the gradients to vanish
- Not zero-centered
- Computationally expensive (because of exp (x))

Activation functions : Sigmoid



$$a_3 = w_2 h_2$$
$$h_3 = \sigma(a_3)$$

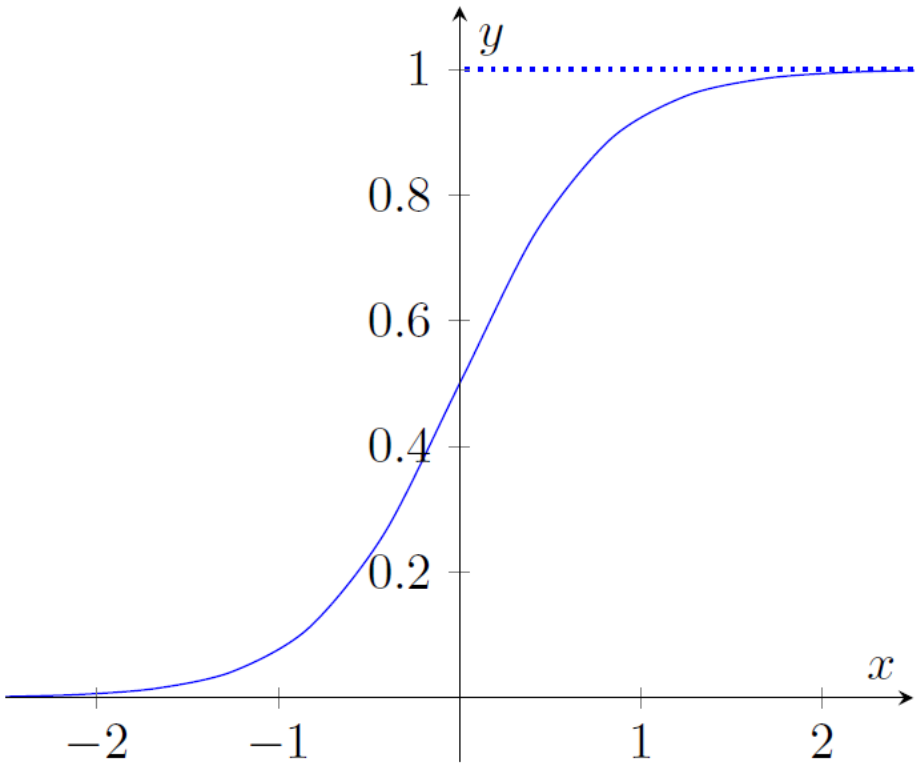
While calculating ∇w_2 at some point in the chain rule we will encounter

$$\frac{\partial h_3}{\partial a_3} = \frac{\partial \sigma(a_3)}{\partial a_3} = \sigma(a_3)(1 - \sigma(a_3))$$

What is the consequence of this ?

To answer this question let us first understand the concept of saturated neuron ?

Activation functions : Sigmoid



- A sigmoid neuron is said to have saturated when:

$$\sigma(x) = 1$$

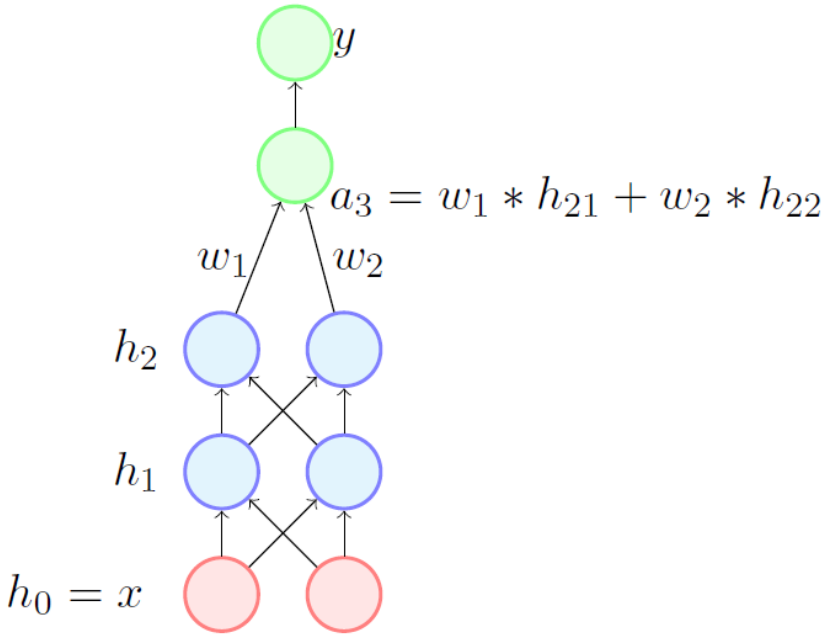
or

$$\sigma(x) = 0$$

- The gradient at the saturation would be equal to 0.

- Therefore, saturated neurons causes the gradient to vanish.

Activation functions : Sigmoid



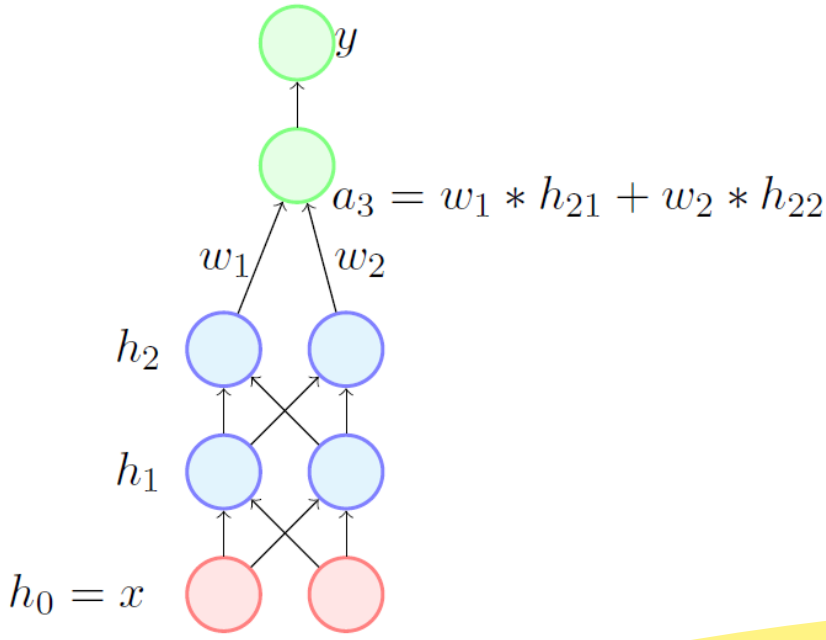
Sigmoid neurons are not zero-centered.

Consider the gradients w.r.t. w_1 and w_2

$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_1}$$

$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \frac{\partial a_3}{\partial w_2}$$

Activation functions : Sigmoid



Sigmoid neurons are not zero-centered.

Consider the gradients w.r.t. w_1 and w_2

$$\nabla w_1 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{21}$$

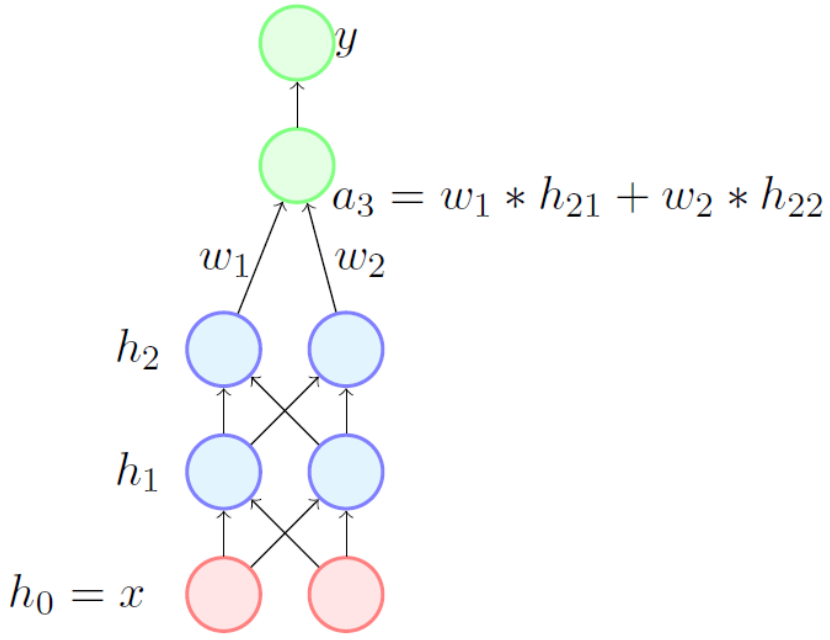
$$\nabla w_2 = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \frac{\partial y}{h_3} \frac{\partial h_3}{\partial a_3} \; h_{22}$$

Note that h_{21} and h_{22} are between $[0, 1]$ (*i.e.*, they are both positive)

So if the first common term (in red) is positive (negative) then both ∇w_1 and ∇w_2 are positive (negative)

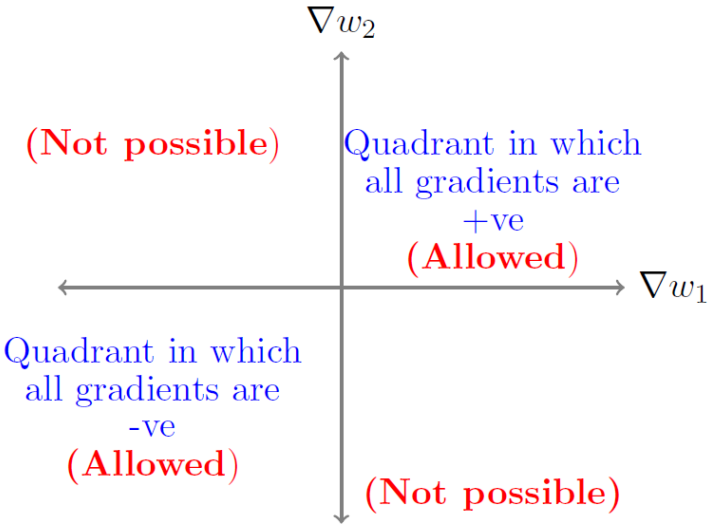
Essentially, either all the gradients at a layer are positive or all the gradients at a layer are negative

Activation functions : Sigmoid

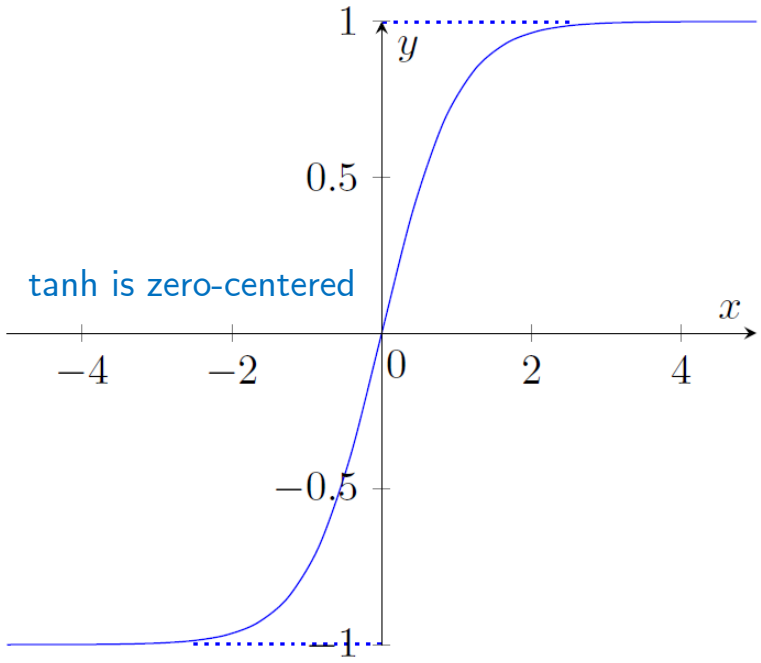


Sigmoid neurons are not zero-centered.

- This restricts the possible update directions



Activation functions : tanh



$$f(x) = \tanh(x)$$

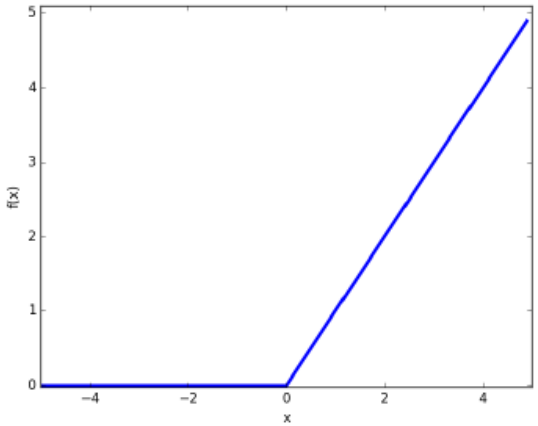
Gradient of tanh function:

$$\frac{\partial tanh(x)}{\partial x} = (1 - tanh^2(x))$$

Problems:

- The gradient still vanishes at saturation.
- Also, computationally expensive.

Activation functions : ReLU

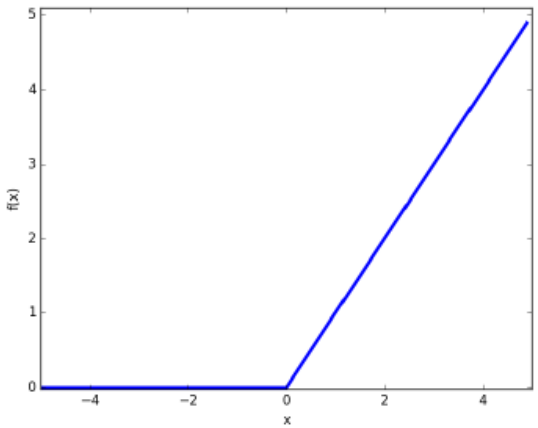


$$f(x) = max(0, x)$$

Advantages of ReLU:

- Does not saturate in the positive region
- Computationally efficient
- In practice converges much faster than sigmoid/tanh

Activation functions : ReLU

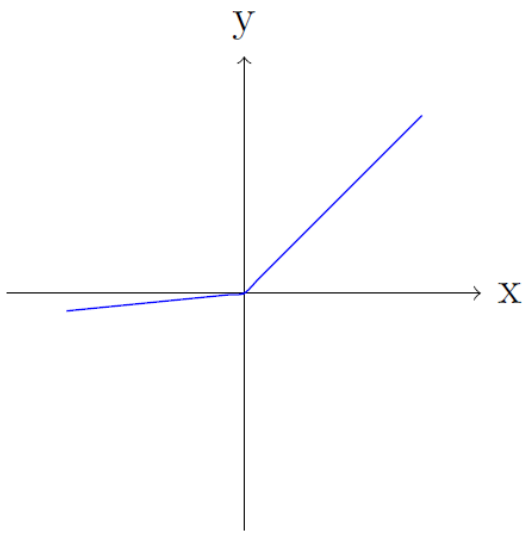


$$f(x) = max(0, x)$$

Gradient of ReLU function:

$$\frac{\partial ReLU(x)}{\partial x} = 0 \quad if \quad x < 0$$
$$= 1 \quad if \quad x > 0$$

Activation functions : Leaky ReLU

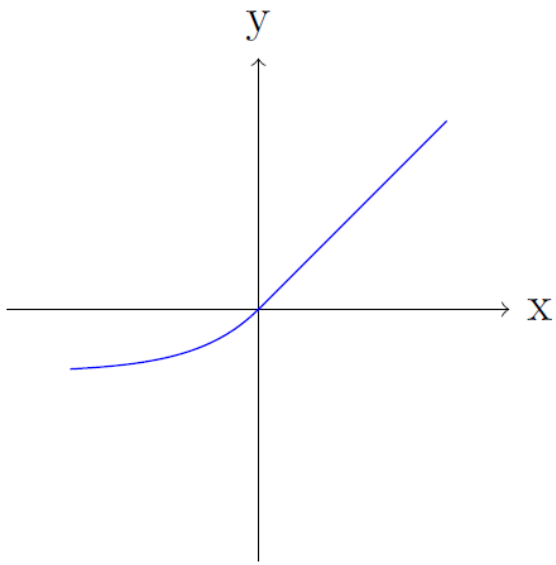


$$f(x) = \max(0.01x, x)$$

- In ReLU, a neuron would die if the input is negative or bias gets updated with a large negative value.
- To solve this issue, leaky ReLU was introduced, where 0.01x ensures that at least a small gradient will flow through.
- It is Computationally efficient
- Close to zero centered outputs
- No saturation.
- In general, we have : **Parametric ReLU**

$$f(x) = \max(\alpha x, x)$$

Activation functions : Exponential LU



$$f(x) = x \quad if \quad x > 0$$
$$= ae^x - 1 \quad if \quad x \leq 0$$

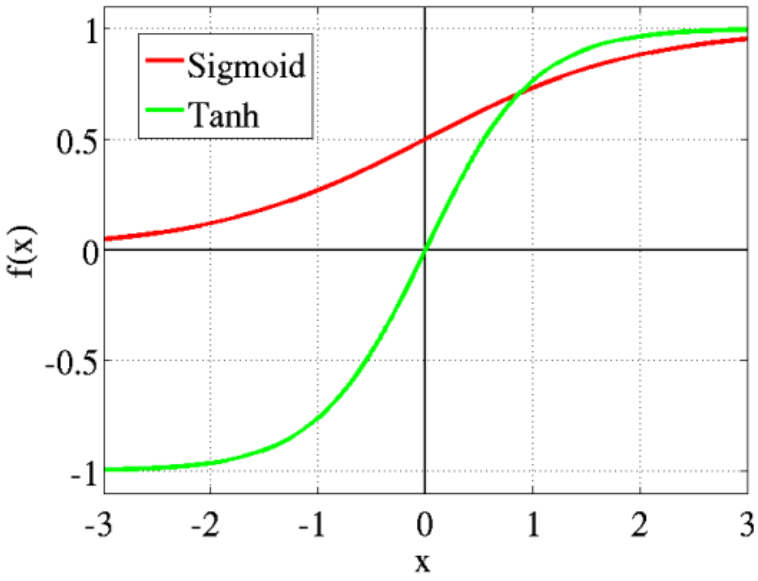
- All benefits of ReLU
- $(ae^x + 1)$ ensures that at least a small gradient will flow through
- Close to zero centered outputs
- Expensive (requires computation of exp(x))

Activation functions : Maxout

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

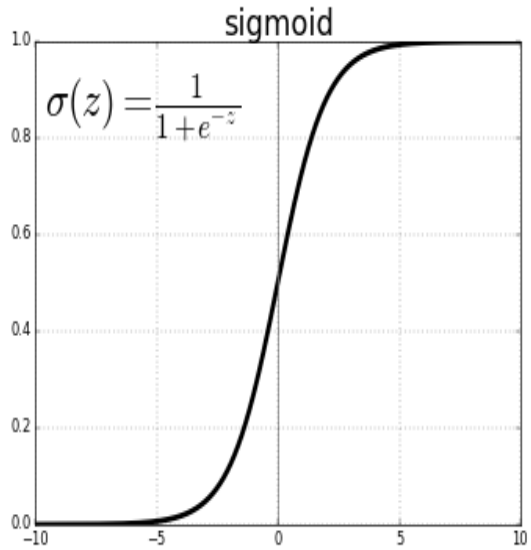
- Generalizes ReLU and Leaky ReLU
- No saturation! No death!
- But, doubles the number of parameters

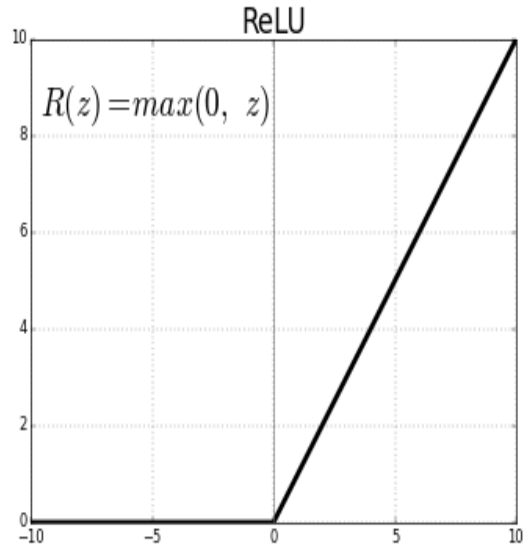
Activation Functions – Sigmoid vs Tanh



- Sigmoid
  - to **predict the probability**
  - between the range of **0 and 1,** sigmoid is the right choice.
  - is **differentiable**-, we can find the slope of the sigmoid curve at any two points.
  - The function is **monotonic** but function's derivative is not.
  - can cause a neural network to get stuck at the training time.
- Tanh
  - Range is from (-1 to 1)
  - Centering data – mean = 0
  - is also sigmoidal (s - shaped)
  - Advantage is that the -ve inputs will be mapped strongly negative
- Disadvantage – If x is very small or very large slope or gradient becomes 0 which slows down gradient descent

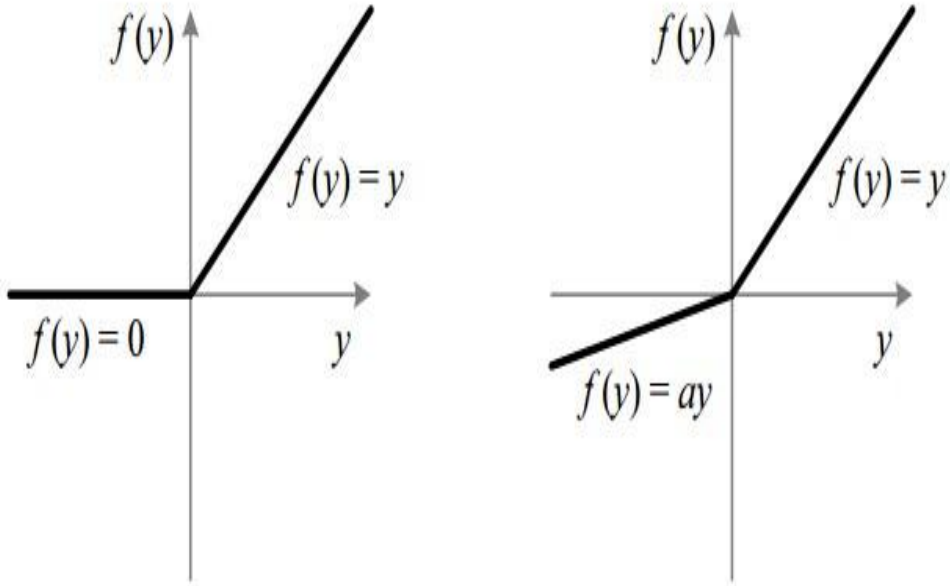
Activation Functions – Sigmoid vs ReLU





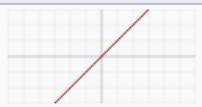
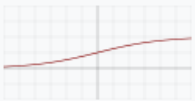
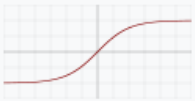
- ReLU is half rectified
- f(z) is 0 when z is < 0 and f(z) is equal to z when z is >= zero.
- Derivative =1 when z is +ve and 0 when z is 0-ve
- The function and its derivative **both are monotonic**
- **Alternate to ReLU is softplus activation function**
  - **Softplus(z) = log(1+exp(z)), Close to 0 when z is –ve and close to z when z is +ve**

Activation Functions- ReLU vs Leaky ReLU



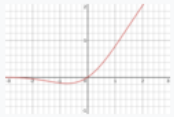
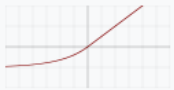
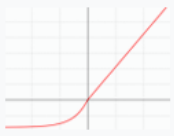
- The leak helps to increase the range of the ReLU function.
- Usually, the value of **a** is 0.01.
- When **a is not 0.01** then it is called **Randomized ReLU**.
- **range** of the Leaky ReLU is (-infinity to infinity)

Activation Function Cheat Sheet

Name	Plot	Function, $g(x)$	Derivative of g, $g'(x)$	Range	Order of continuity
Identity		x	1	$(-\infty, \infty)$	C^∞
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	0	$\{0, 1\}$	C^{-1}
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1+e^{-x}}$	$g(x)(1-g(x))$	$(0, 1)$	C^∞
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - g(x)^2$	$(-1, 1)$	C^∞
Soboleva modified hyperbolic tangent (smht)		$\operatorname{smht}(x) \doteq \frac{e^{ax} - e^{-bx}}{e^{cx} + e^{-dx}}$		$(-1, 1)$	C^∞
Rectified linear unit (ReLU)[10]		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max(0, x) = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$	$[0, \infty)$	C^0

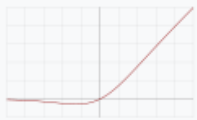
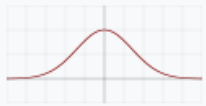
Activation function - Wikipedia

Activation Function Cheat Sheet

Gaussian Error Linear Unit (GELU)[2]		$\frac{1}{2}x\left(1+\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $= x\Phi(x)$	$\Phi(x)+x\phi(x)$	$(-0.17\ldots,\infty)$	C^∞
Softplus[11]		$\ln(1+e^x)$	$\frac{1}{1+e^{-x}}$	$(0,\infty)$	C^∞
Exponential linear unit (ELU)[12]		$\begin{cases}\alpha\left(e^x-1\right) & \text{if } x\le 0\\ x & \text{if } x>0\end{cases}$ with parameter α	$\begin{cases}\alpha e^x & \text{if } x<0\\ 1 & \text{if } x>0\end{cases}$	$(-\alpha,\infty)$	$\begin{cases}C^1 & \text{if } \alpha=1\\ C^0 & \text{otherwise}\end{cases}$
Scaled exponential linear unit (SELU)[13]		$\lambda\begin{cases}\alpha(e^x-1) & \text{if } x<0\\ x & \text{if } x\ge 0\end{cases}$ with parameters $\lambda=1.0507$ and $\alpha=1.67326$	$\lambda\begin{cases}\alpha e^x & \text{if } x<0\\ 1 & \text{if } x\ge 0\end{cases}$	$(-\lambda\alpha,\infty)$	C^0
Leaky rectified linear unit (Leaky ReLU)[14]		$\begin{cases}0.01x & \text{if } x\le 0\\ x & \text{if } x>0\end{cases}$	$\begin{cases}0.01 & \text{if } x<0\\ 1 & \text{if } x>0\end{cases}$	$(-\infty,\infty)$	C^0
Parametric rectified linear unit (PReLU)[15]		$\begin{cases}\alpha x & \text{if } x<0\\ x & \text{if } x\ge 0\end{cases}$ with parameter α	$\begin{cases}\alpha & \text{if } x<0\\ 1 & \text{if } x\ge 0\end{cases}$	$(-\infty,\infty)$	C^0

Activation function - Wikipedia

Activation Function Cheat Sheet

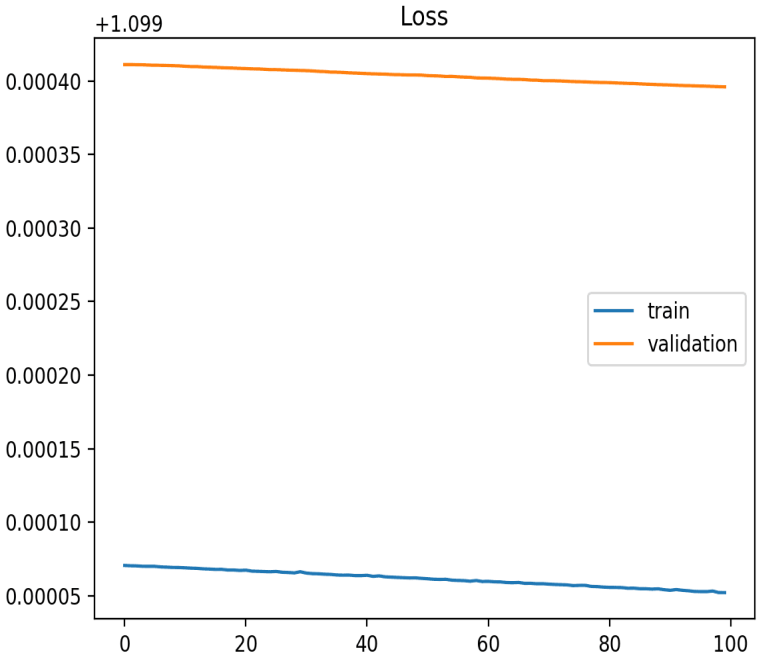
Sigmoid linear unit (SiLU,[2] Sigmoid shrinkage,[16] SiL,[17] or Swish-1[18])		$\frac{x}{1+e^{-x}}$	$\frac{1+e^{-x}+xe^{-x}}{(1+e^{-x})^2}$	$[-0.278\ldots,\infty)$	C^∞
Gaussian		e^{-x^2}	$-2xe^{-x^2}$	$(0,1]$	C^∞

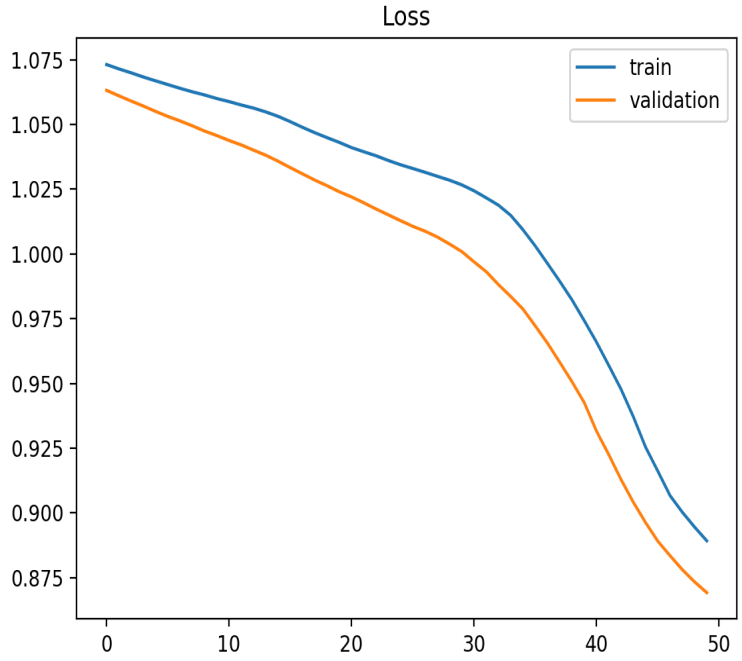
Activation function - Wikipedia

Learning Curves

- Line plot of learning (y-axis) over experience (x-axis)
- The metric used to evaluate learning could be
- **Optimization Learning Curves**:
  - calculated on the metric by which the parameters of the model are being optimized, e.g. loss.
  - Minimizing, such as loss or error
- **Performance Learning Curves**:
  - calculated on the metric by which the model will be evaluated and selected, e.g. accuracy.
  - Maximizing metric , such as classification accuracy
- **Train Learning Curve**:
  - calculated from the training dataset that gives an idea of how well the model is learning.
- **Validation Learning Curve**:
  - calculated from a hold-out validation dataset that gives an idea of how well the model is generalizing.

Underfitting



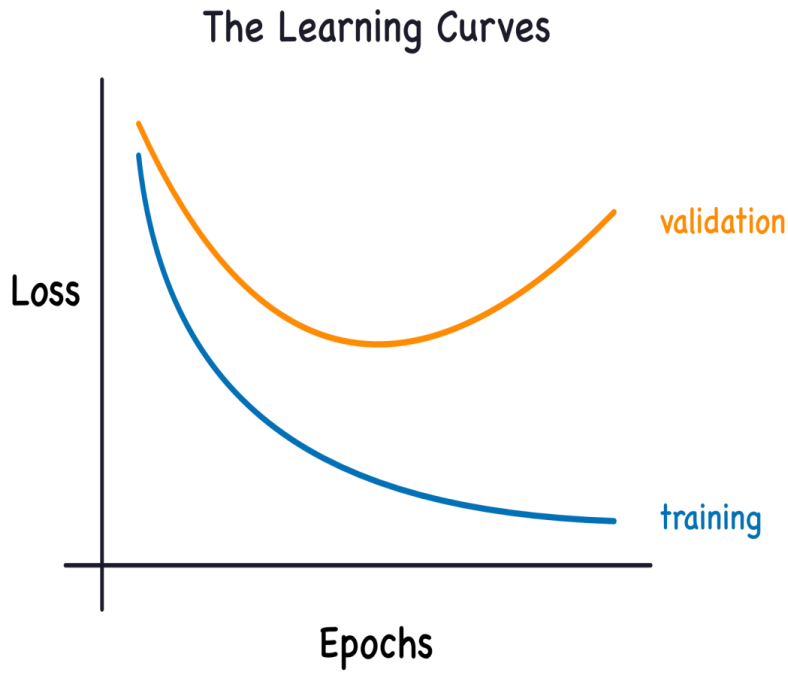


Underfitting refers to a model that cannot learn the training dataset.
A plot of learning curves shows underfitting if:

- The training loss remains flat regardless of training.
- The training loss continues to decrease until the end of training

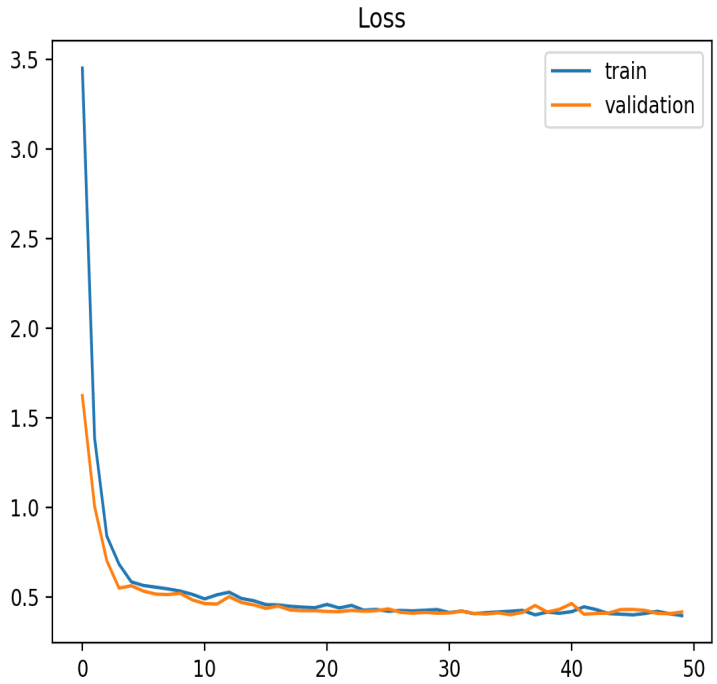
Overfitting

- Overfitting
  - Model specialized on training data, it is not able to generalize to new data
  - Results in increase in generalization error.
  - generalization error can be measured by the performance of the model on the validation dataset.
- A plot of learning curves shows overfitting if:
  - The plot of training loss continues to decrease with experience.
  - The plot of validation loss decreases to a point and begins increasing again.
  - The inflection point in validation loss may be the point at which training could be halted as experience after that point shows the dynamics of overfitting.



Good Fit Learning Curves

- A good fit is the goal of the learning algorithm and exists between an overfit and underfit model.
- A plot of learning curves shows a good fit if:
  - Plot of training loss decreases to a point of stability
  - Plot of validation loss decreases to a point of stability and has a small gap with the training loss.
- Loss of the model will almost always be lower on the training than the validation dataset.
- We should expect some gap between the train and validation loss learning curves.
- This gap is referred to as the "generalization gap."



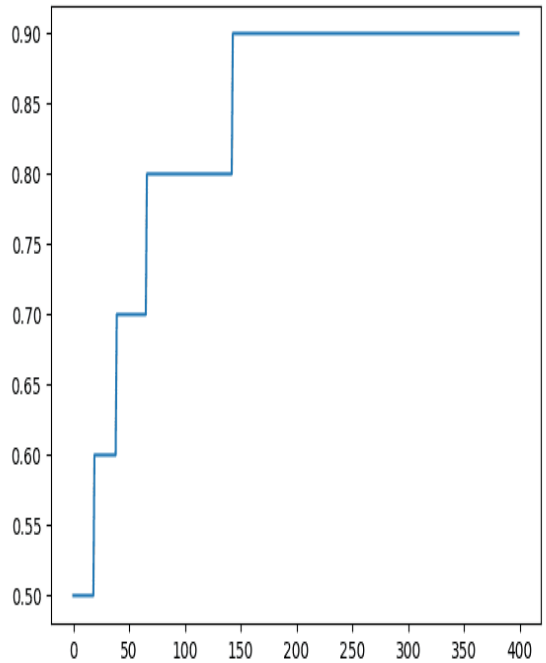
Keras Metrics

- model.compile(..., metrics=['mse'])

- Metric values are recorded at the end of each epoch on the training dataset.
- If a validation dataset is also provided, then is also calculated for the validation dataset.
- All metrics are reported in verbose output and in the history object returned from calling the fit() function.

Keras Metrics

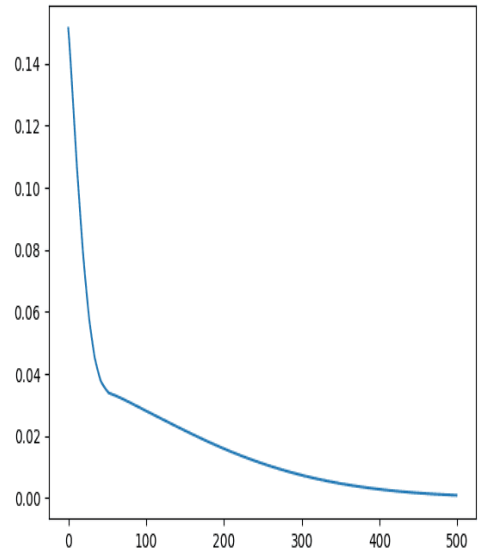
- **Accuracy metrics**
  - Accuracy
    - Calculates how often predictions equal labels.
  - Binary Accuracy
    - Calculates how often predictions match binary labels.
  - Categorical Accuracy
    - Calculates how often predictions match one-hot labels
  - Sparse Categorical Accuracy
    - Calculates how often predictions match integer labels.
  - TopK Categorical Accuracy
    - calculates the percentage of records for which the targets are in the top K predictions
      - rank the predictions in the descending order of probability values.
      - If the rank of the yPred is less than or equal to K, it is considered accurate.
  - Sparse TopK Categorical Accuracy class
    - Computes how often integer targets are in the top K predictions.



Aurelien Geron, "Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

Keras Metrics

- **Regression metrics**
  - Mean Squared Error
    - Computes the mean squared error between y_true and y_pred
  - Root Mean Squared Error
    - Computes root mean SE metric between y_true and y_pred
  - Mean Absolute Error
    - Computes the mean absolute error between the labels and predictions
  - Mean Absolute Percentage Error
    - **MAPE** = (1/n) * Σ(|actual – prediction| / |actual|) * 100
    - Average difference between the predicted and the actual in %
  - Mean Squared Logarithmic Error
    - measure of the ratio between the true and predicted values.



Keras Metrics

- AUC
- Precision
- Recall
- TruePositives
- TrueNegatives
- FalsePositives
- FalseNegatives
- PrecisionAtRecall
  - Computes best precision where recall is >= specified value
- SensitivityAtSpecificity
  - Computes best sensitivity where specificity is >= specified value
- SpecificityAtSensitivity
  - Computes best specificity where sensitivity is >= specified value

- **Probabilistic metrics**
  - Binary Crossentropy
  - Categorical Crossentropy
  - Sparse Categorical Crossentropy
  - KLDivergence
    - a measure of how two probability distributions are different from each other
  - Poisson
    - if dataset comes from a Poisson distribution