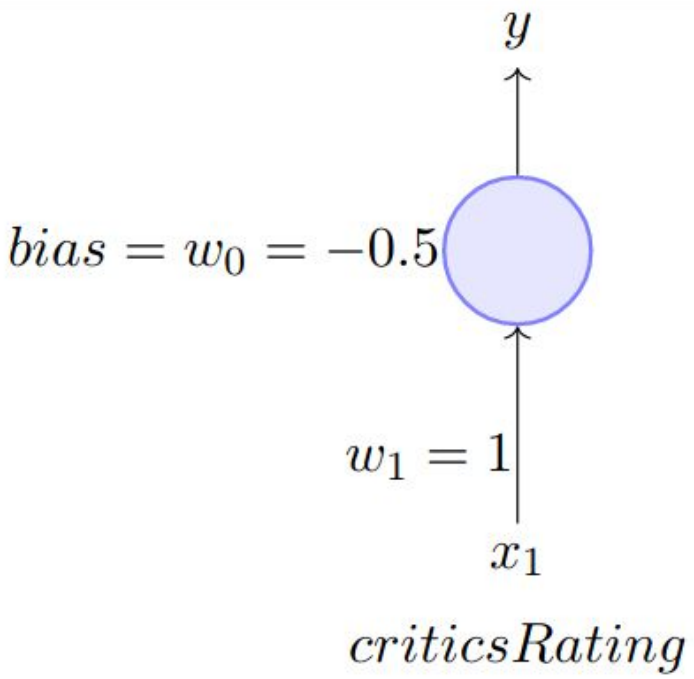
Neural Networks Part2

Aleksanyan Lida

From boolean functions to arbitrary

- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbb{R}^n$ (instead of $\{0,1\}^n$) and $y \in \mathbb{R}$ (instead of $\{0,1\}$) ?
- Can we have a network which can (approximately) represent such functions ?
- Before answering the above question we will have to first graduate from ***perceptrons*** to ***sigmoidal neurons*** ...

Example



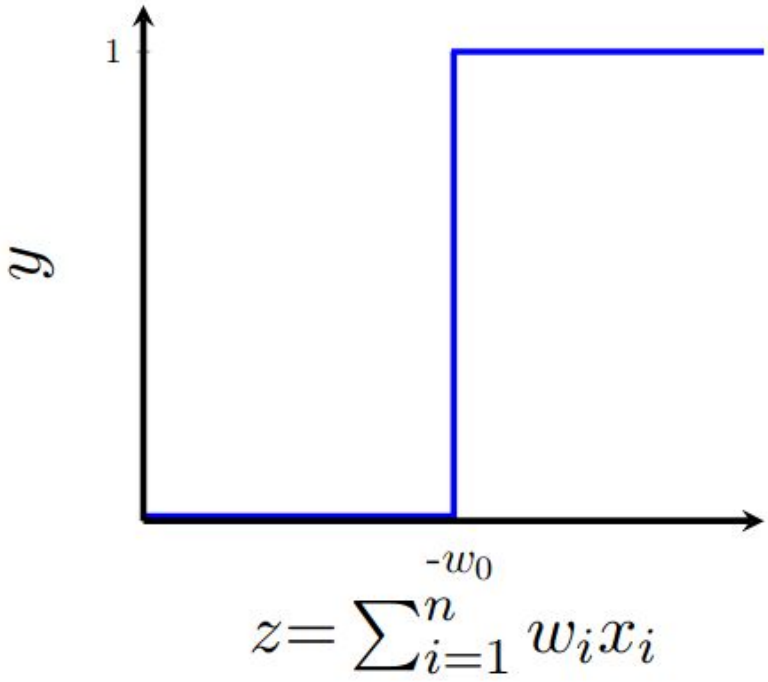
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) and $w_1 = 1$ then what would be the decision for a movie with $criticsRating = 0.51$? (like)
- What about a movie with $criticsRating = 0.49$? (dislike)

How does our Perceptron function look like?

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

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

Let’s return to our movie example, where we made sudden decisions



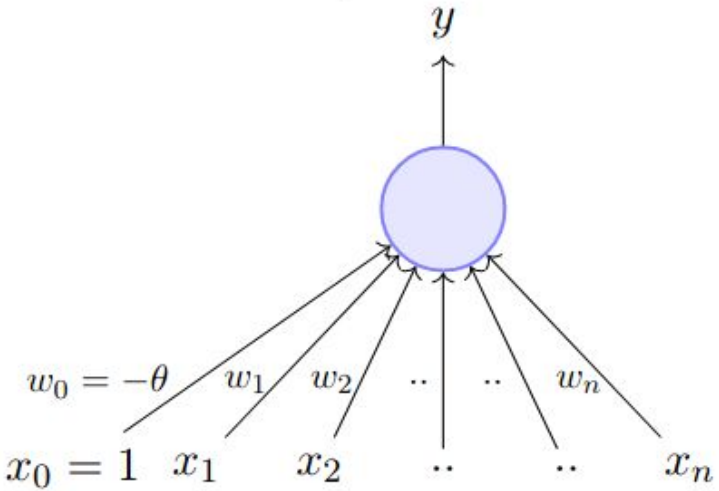
- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function
- There will always be this sudden change in the decision (from 0 to 1) when $\sum_{i=1}^{n}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

Sigmoid neurons

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

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

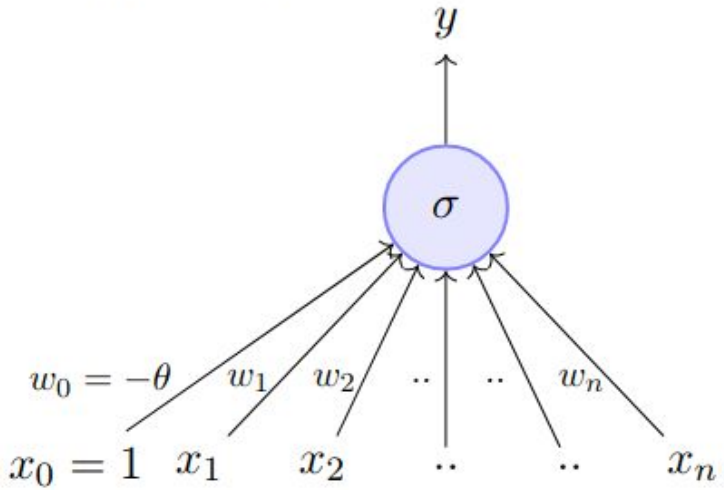
Perceptron



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

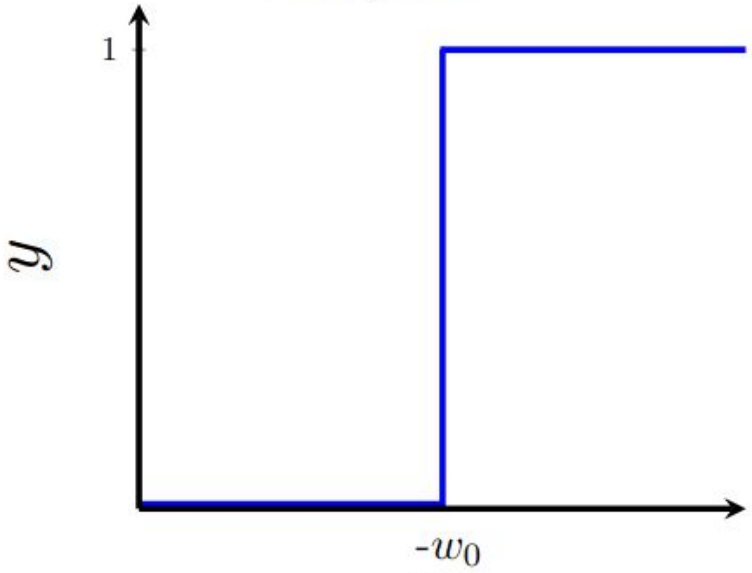
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Sigmoid (logistic) Neuron



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

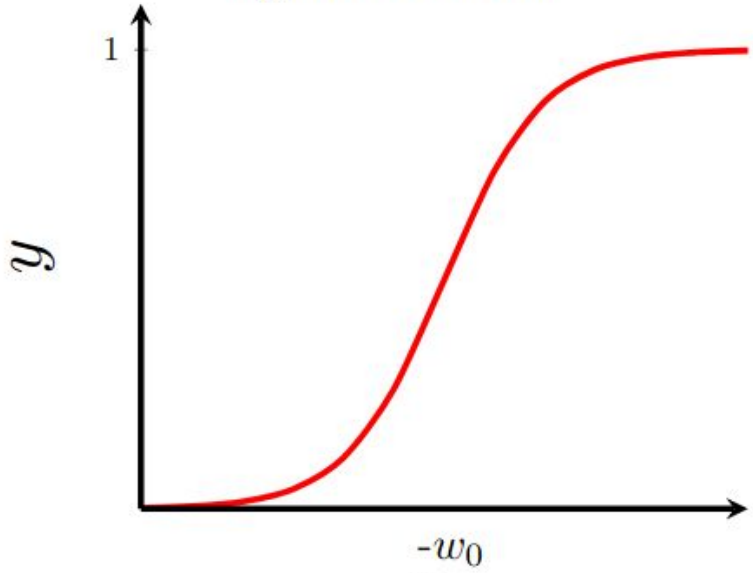
Perceptron



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

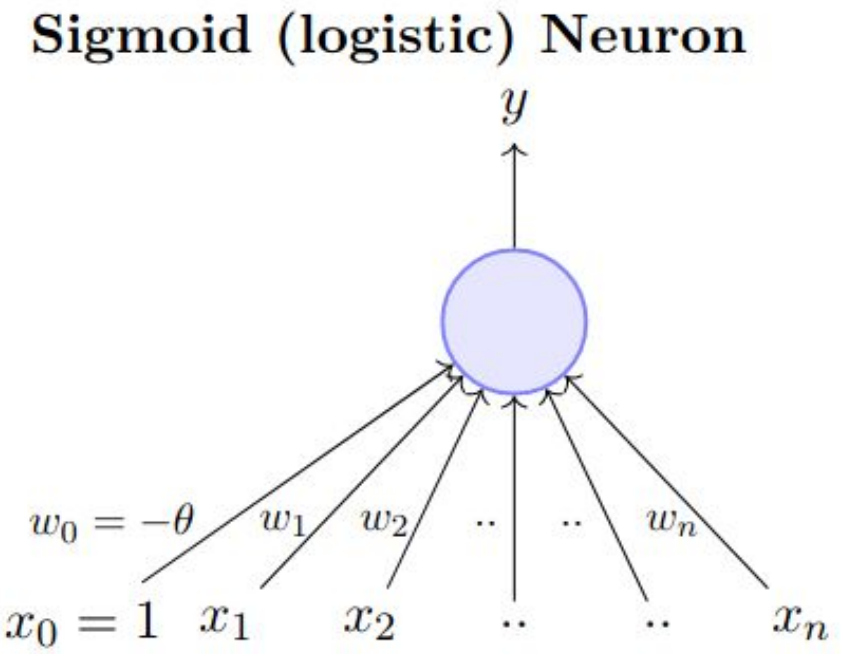
Not smooth, not continuous (at $w0$), **not differentiable**

Sigmoid Neuron



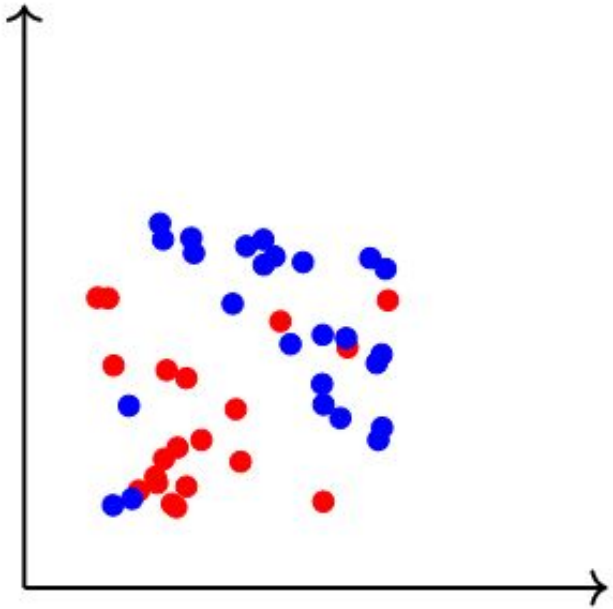
$$z = \sum_{i=1}^{n} w_i x_i$$

Smooth, continuous, **differentiable**



What's next?

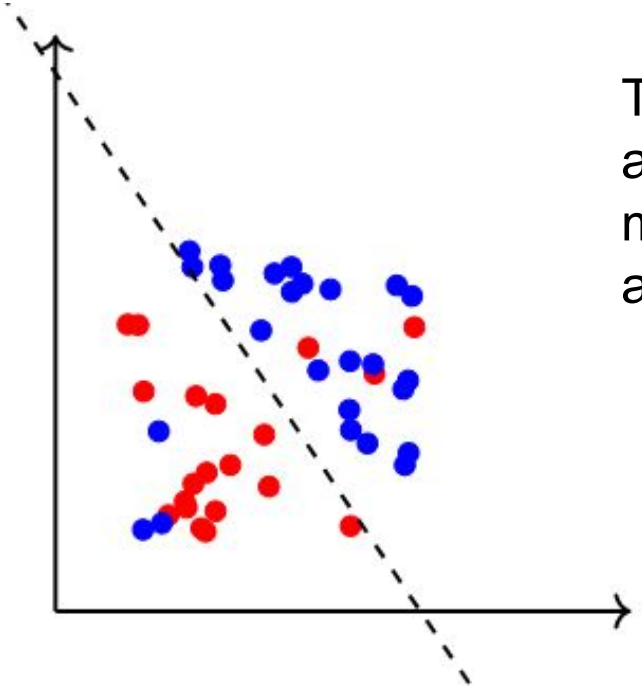
- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron.
- Before we see such an algorithm we will revisit the concept of error.



We know that a ***single*** perceptron cannot deal with this data because it is not linearly separable.

What would happen if we use a perceptron model to classify this data ?

We would probably end up with a line like this



This line misclassifies 3 blue points and 3 red points but it will surely make more mistakes in real world applications.

Let's look at a typical machine learning setup which has the following components:

This brings us to a typical machine learning setup which has the following components...

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

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

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

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

or just about any function

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

Example

As an illustration, consider our movie example

- **Data:** $\{x_i = movie, y_i = like/dislike\}_{i=1}^{n}$
- **Model:** Our approximation of the relation between $\mathbf{x}$ and y (the probability of liking a movie).

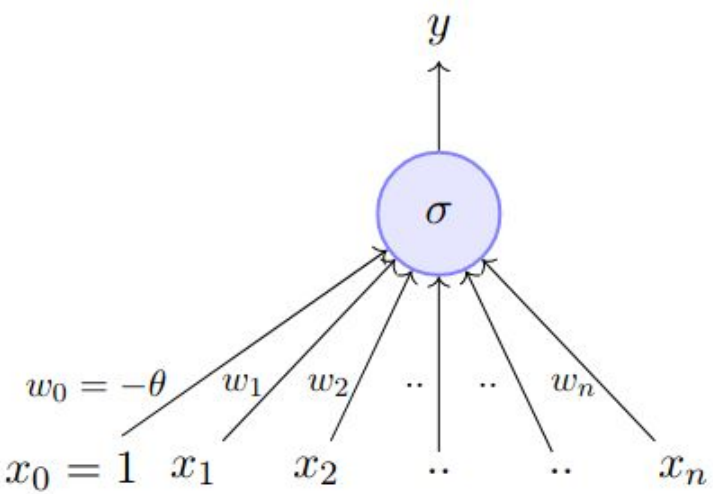
$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w^T x})}}$$

- **Parameter: w**
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error function:** One possibility is

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

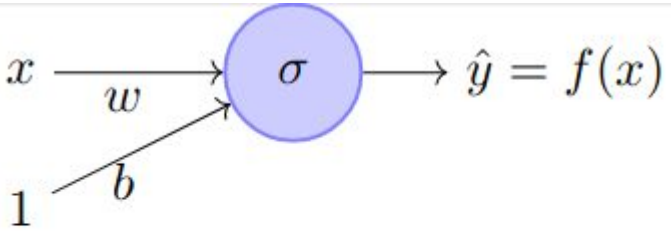
The learning algorithm should aim to find a w which minimizes the above function (squared error between y and $\hat{y}$)

Sigmoid neuron



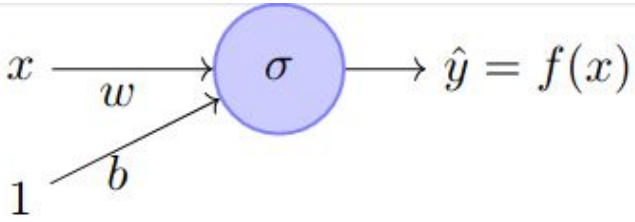
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**
- σ stands for the sigmoid function (logistic function in this case)



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

- For ease of explanation, we will consider a very simplified version of the model having just 1 input
- Further to be consistent with the literature, from now on, we will refer to w_0 as b (bias)



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

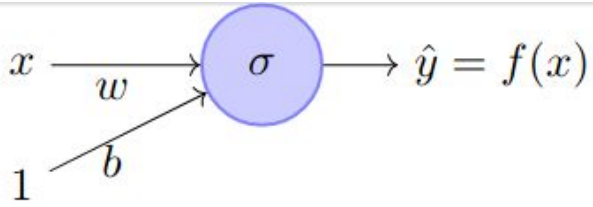
Input for training

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

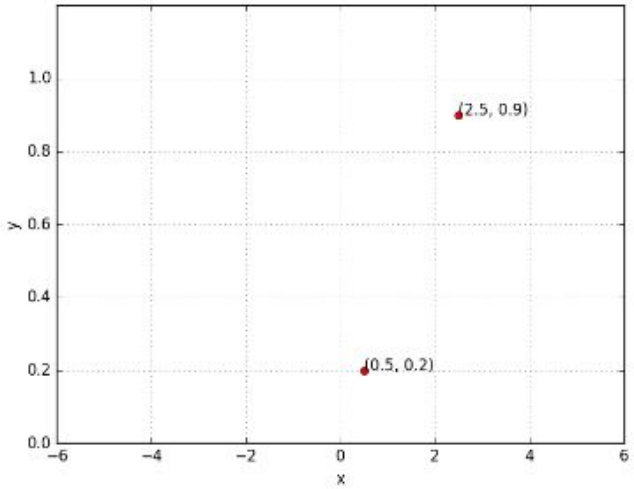
Training objective

Find w and b such that:

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

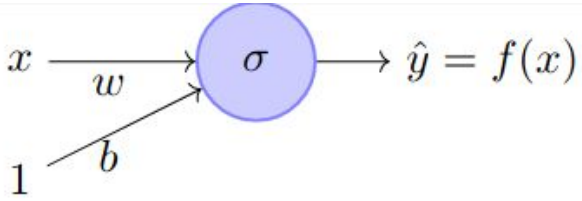


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

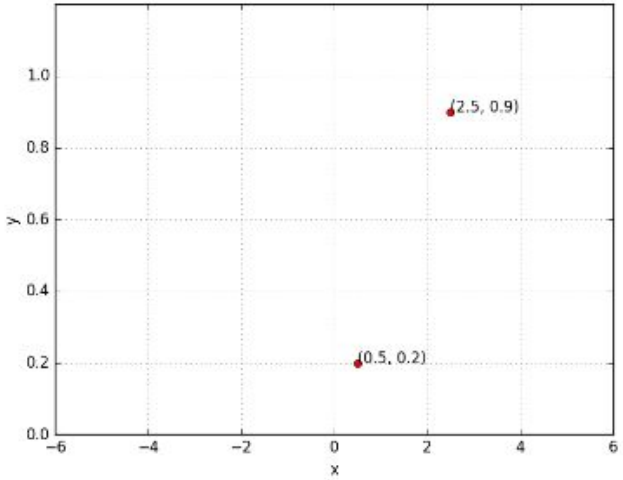


What does it mean to train the network?

- 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: **?**



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

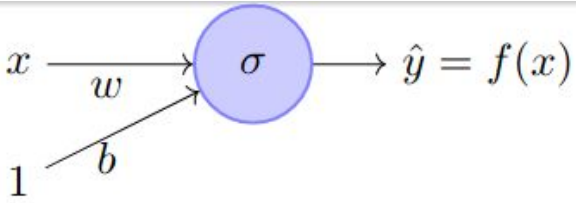


What does it mean to train the network?

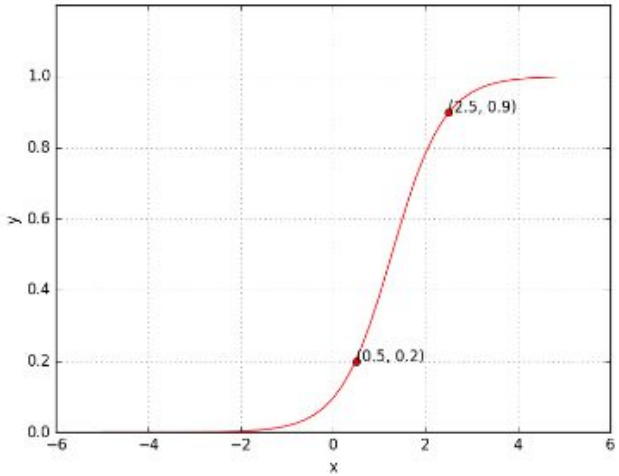
- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$
- At the end of training we expect to find w*, b* such that:
- $f(0.5) \to 0.2$ and $f(2.5) \to 0.9$

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



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



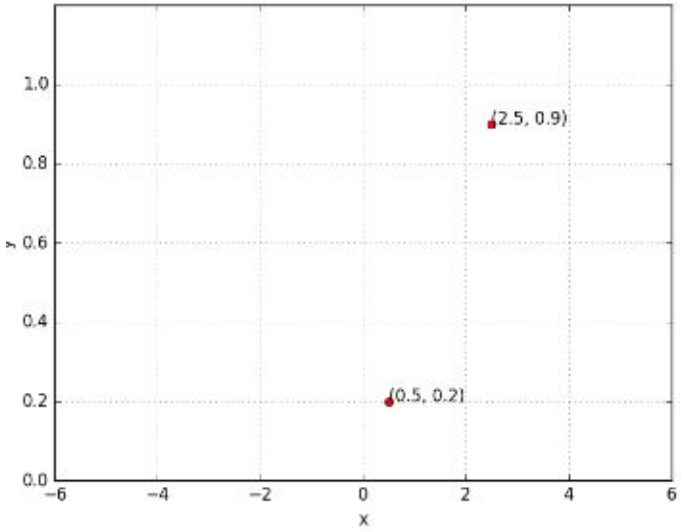
What does it mean to train the network?

- 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 $f(2.5) \to 0.9$

In other words...

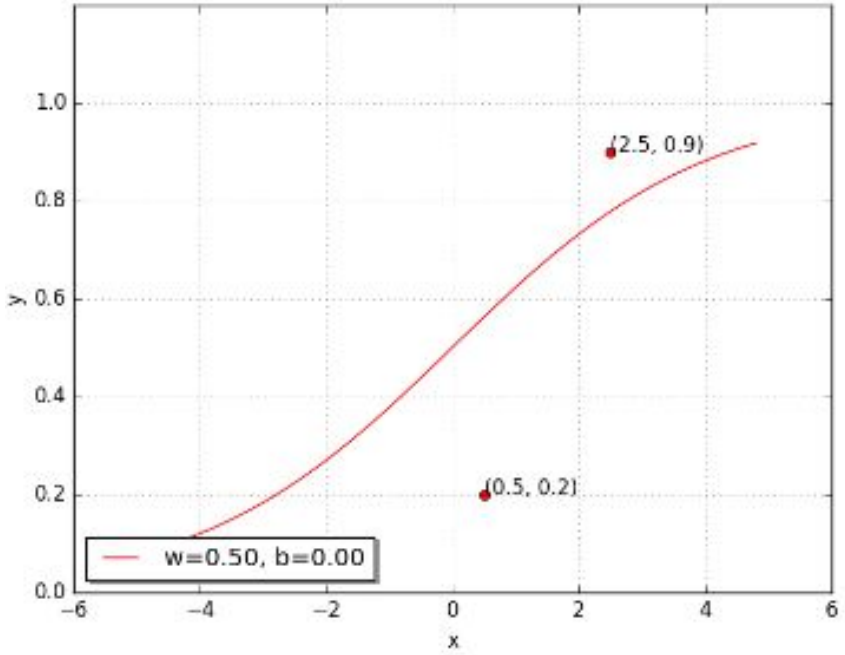
- We hope to find a sigmoid function such that $(0.5, 0.2)$ and $(2.5, 0.9)$ lie on this sigmoid

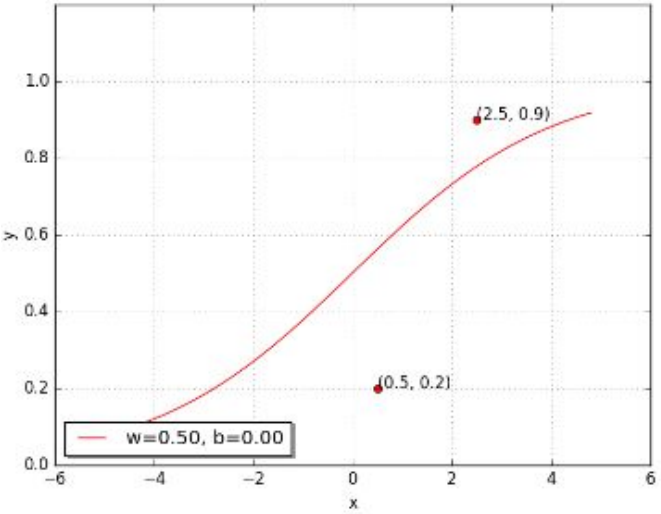
- Can we try to find such a w^*, b^* manually



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

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

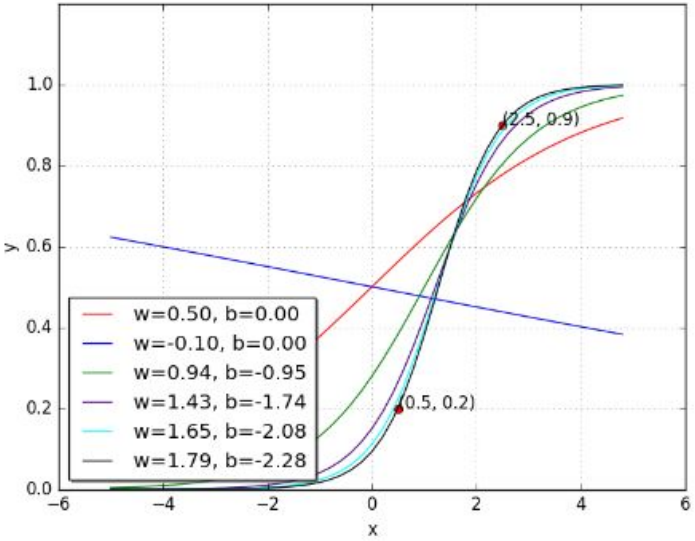




- Let us try a random guess.. (say, $w = 0.5, b = 0$)
- Clearly not good, but how bad is it ?
- Let us revisit $\mathscr{L}(w, b)$ to see how bad it is ...

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

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



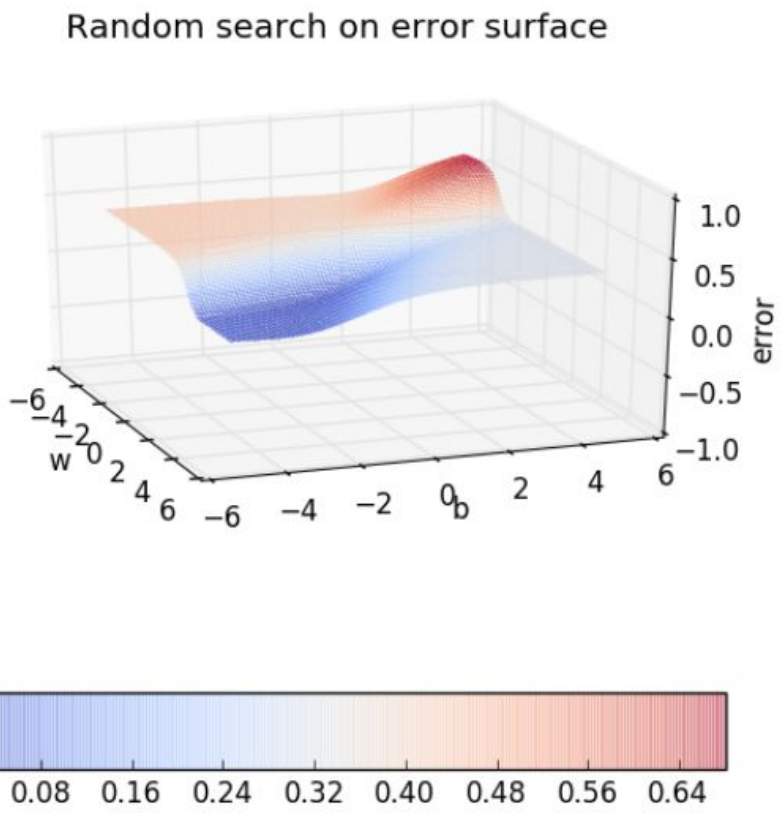
Let us try some other values of w, b

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

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

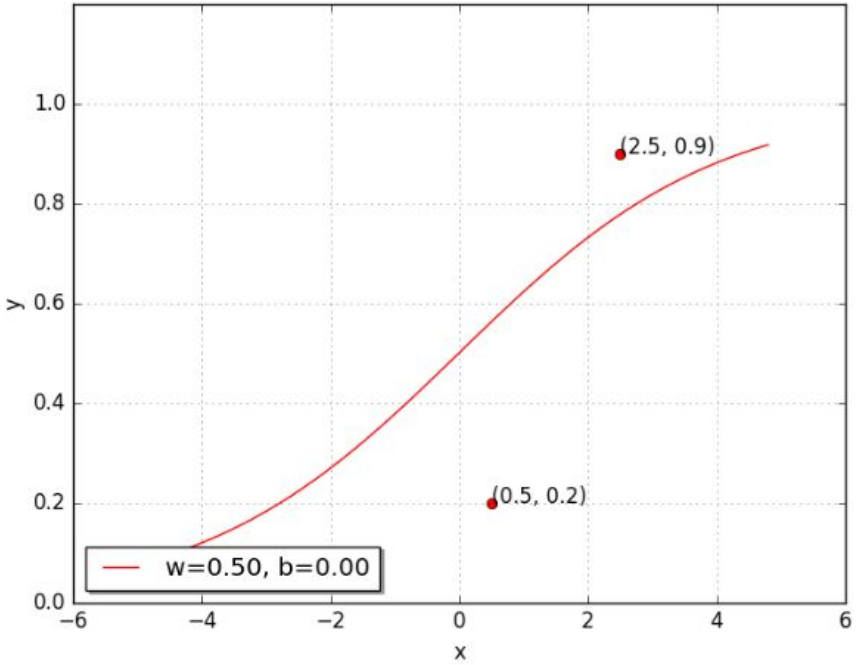
With some guess work and intuition we were able to find the right values for w and b

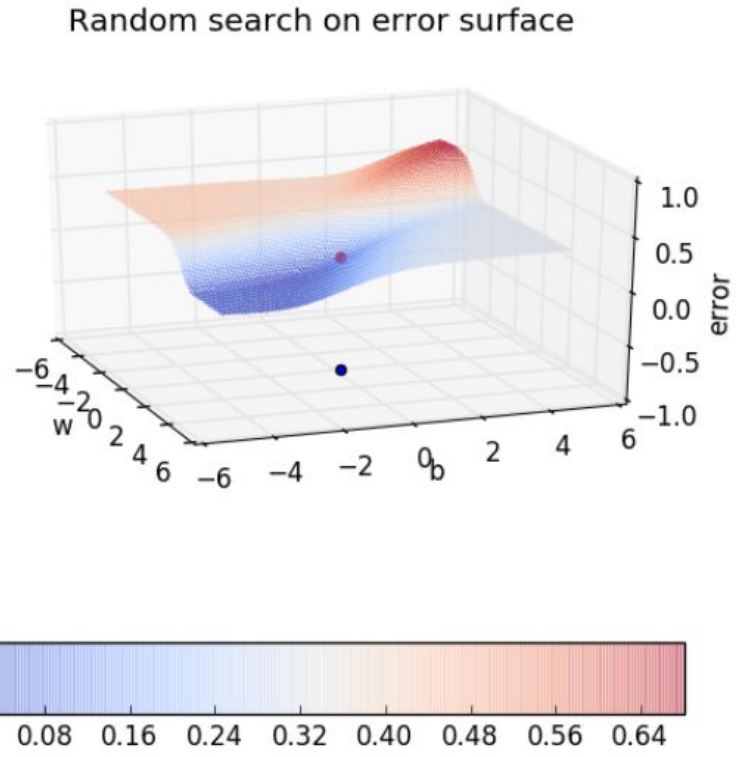
Let us look at something better than our "guess work" algorithm:

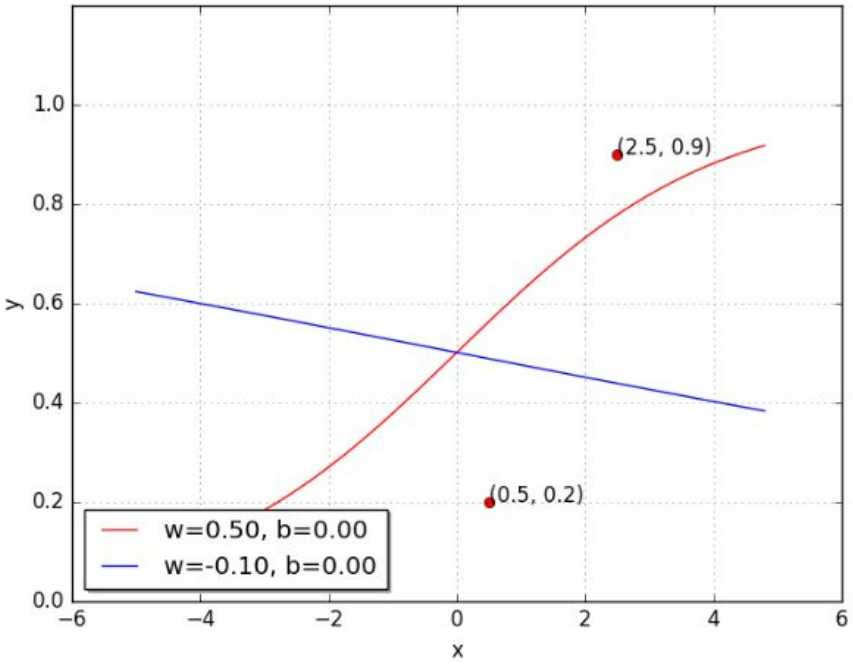


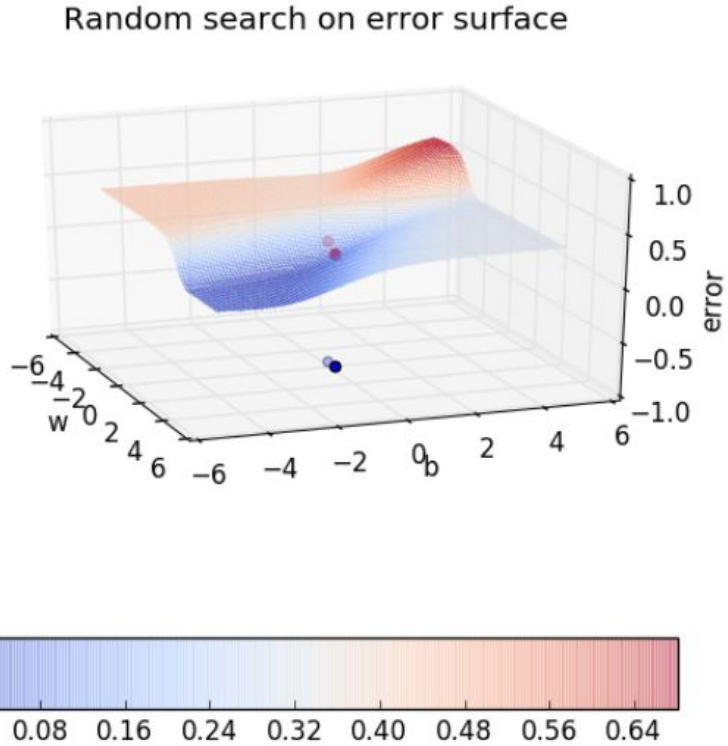
- Since we have only 2 points and 2 parameters (w, b) we can easily plot $\mathscr{L}(w, b)$ for different values of (w, b) and pick the one where $\mathscr{L}(w, b)$ is minimum
- But of course this becomes intractable once you have many more data points and many more parameters !!
- Further, even here we have plotted the error surface only for a small range of (w, b) [from $(-6, 6)$ and not from $(-\inf, \inf)$]

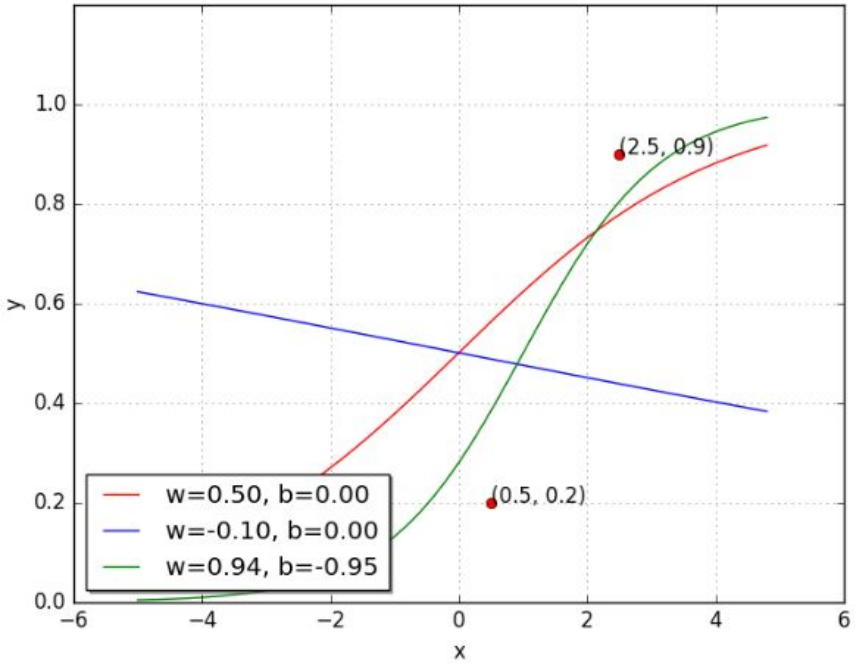
Let us look at the geometric interpretation of our “guess work” algorithm in terms of this error surface

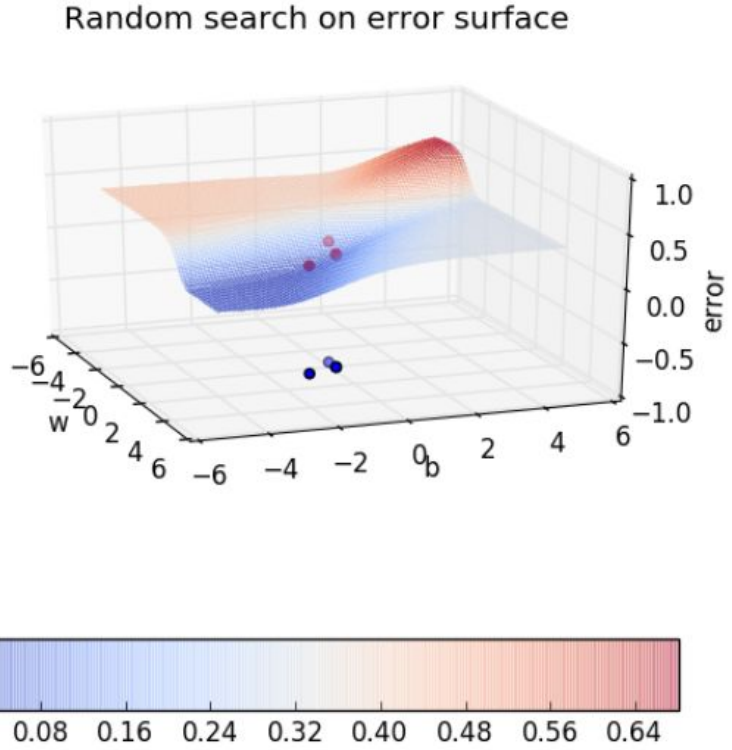

(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
x
y

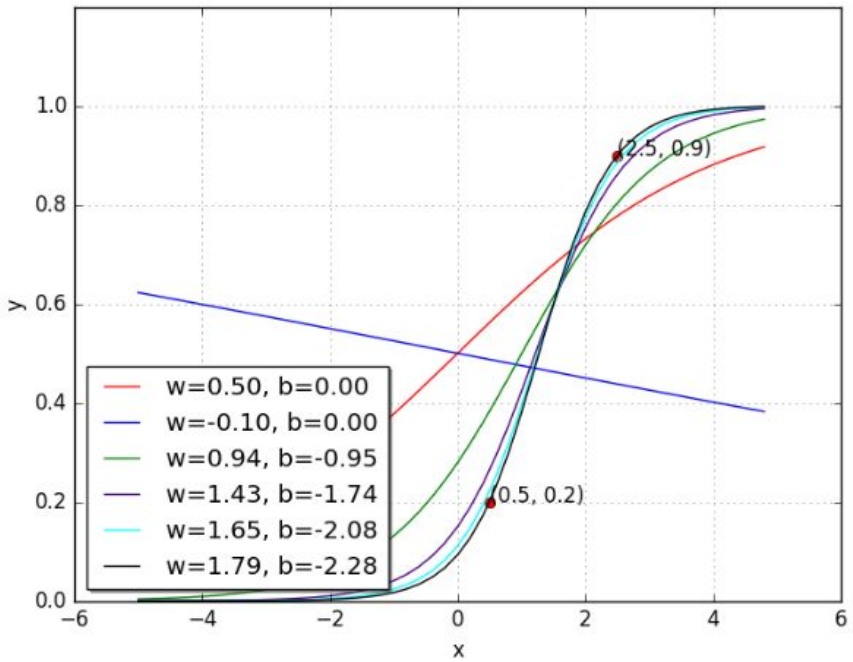

Random search on error surface
w
b
error

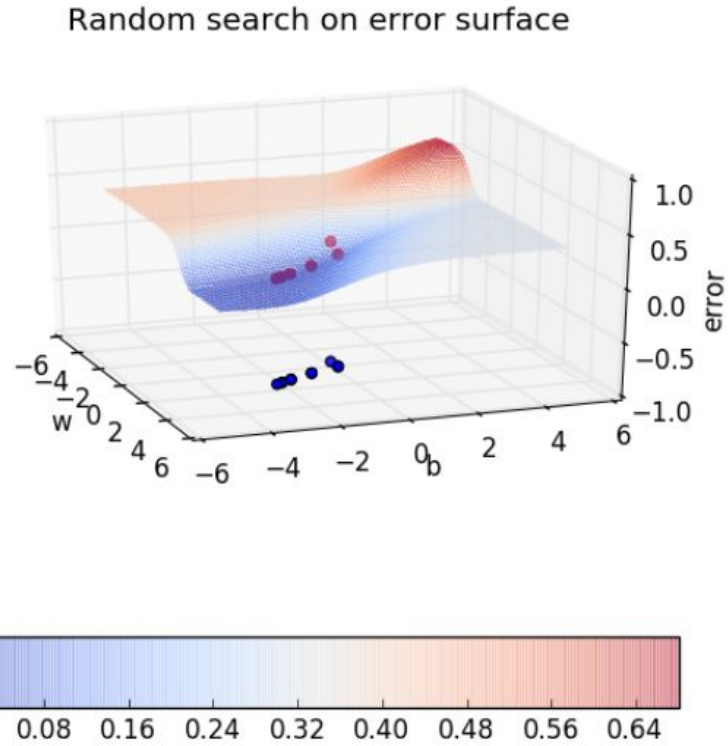

(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
x
y


Random search on error surface
w
b
error


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
w
b
error
1.0
0.5
0.0
−0.5
−1.0
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
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
w
b
error

Learning Parameters : Gradient Descent

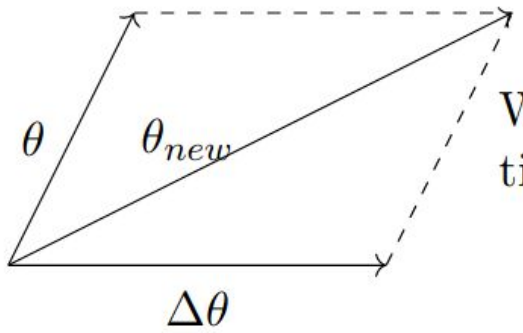
Goal: Find a better way of traversing the error surface so that we can reach the minimum value quickly without resorting to brute force search.

vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b



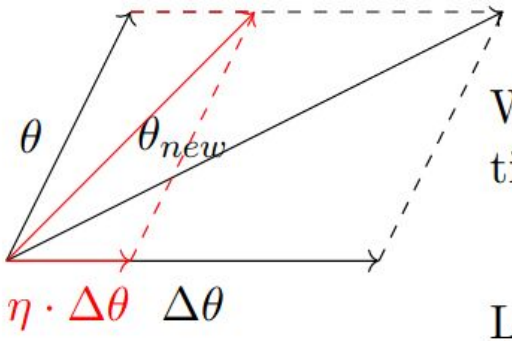
We moved in the direction of $\Delta\theta$

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 $\Delta\theta$

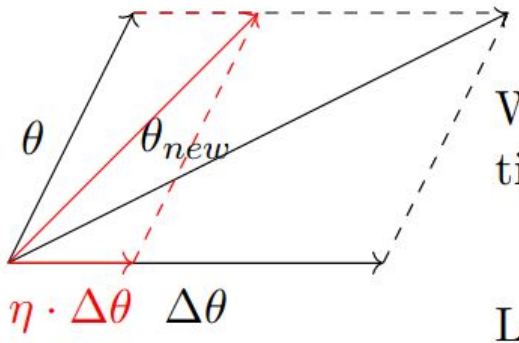
Let us be a bit conservative: move only by a small amount η

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

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

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

ФОРМУЛА ТЕЙЛОРА

Пусть имеем функцию многочлен $P_n(x)$ степени n:

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n .$$

Преобразуем этот многочлен также в многочлен степени n относительно разности $x - x_0$, где x_0 – произвольное число, т.е. представим $P_n(x)$ в виде

$$P_n(x) = A_0 + A_1(x - x_0) + A_2(x - x_0)^2 + \ldots + A_n(x - x_0)^n . \qquad (*)$$

Для нахождения коэффициентов $A_0, A_1, A_2, \ldots, A_n$ продифференцируем n раз равенство (*):

$$P_n'(x) = A_1 + 2A_2(x - x_0) + 3A_3(x - x_0)^2 + \ldots + nA_n(x - x_0)^{n-1},$$

$$P_n''(x) = 2A_2 + 2 \cdot 3A_3(x - x_0) + \ldots + n(n-1)A_n(x - x_0)^{n-2},$$

$$P_n'''(x) = 2 \cdot 3A_3 + 2 \cdot 3 \cdot 4A_4(x - x_0) + \ldots + n(n-1)(n-2)A_n(x - x_0)^{n-3},$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$P_n^{(n)}(x) = n(n-1)(n-2)\ldots 2 \cdot 1 A_n .$$

Подставляя $x = x_0$ в полученные равенства и равенство (*), имеем:

$P_n(x_0) = A_0$, т.е. $A_0 = P_n(x_0)$,

$P_n'(x_0) = A_1$, т.е. $A_1 = \frac{P_n'(x_0)}{1!}$,

$P_n''(x_0) = 2A_2$, т.е. $A_2 = \frac{P_n''(x_0)}{2!}$,

$P_n'''(x_0) = 2 \cdot 3A_3$, т.е. $A_3 = \frac{P_n'''(x_0)}{3!}$,

.....

$P_n^{(n)}(x_0) = n(n-1)(n-2)\ldots 2 \cdot 1A_n$, т.е. $A_n = \frac{P_n^{(n)}(x_0)}{n!}$.

Подставляя найденные значения A_0, A_1, A_2, ..., A_n в равенство (*), получим:

$$P_n(x) = P_n(x_0) + \frac{P_n'(x_0)}{1!}(x - x_0) + \frac{P_n''(x_0)}{2!}(x - x_0)^2 + \ldots + \frac{P_n^{(n)}(x_0)}{n!}(x - x_0)^n, \qquad (**)$$

т.е. коэффициент многочлена определяется через производную в точке x_0.

Формула (**) называется формулой Тейлора для многочлена $P_n(x)$ степени n.

Рассмотрим функцию $y = f(x)$. Пусть для функции $f(x)$, существуют производные n - ого порядка. Любой такой функции $f(x)$ сопоставим многочлен n - ого порядка.

$f(x) \sim f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + ... + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$ и обозначим через

$$r_n(x) = f(x) - \left(f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + ... + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \right)$$

Свойства $r_n(x)$:

1. $r_n(x)$ имеет производную до n - ого порядка включительно.
2. $r_n(x_0) = 0,\ r_n'(x_0) = 0,\ r_n''(x_0) = 0,\ ....,\ r_n^{(n)}(x_0) = 0.$

Лемма. Если функция $\varphi(x)$ имеет производную n - ого порядка в окрестности точки x_0 и $\varphi(x_0) = \varphi'(x_0) = ... = \varphi^{(n)}(x_0) = 0$, то $\varphi(x) = o(x - x_0)^n$.

Так как функция $r_n(x)$ удовлетворяет всем условиям леммы, то можно сказать, что $r_n(x) = o(x - x_0)^n$. Таким образом для функции имеем следующее представление

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + o(x - x_0)^n \qquad (1)$$

Формула (1) называется формулой Тейлора с остаточным членом в форме Пеано. Формула Пеано имеет недостаток, т.к. в явном виде мы не знаем вид остаточного члена.

Лагранж получил другой вид для остаточного члена. Он показал, что если существует производная до n-ого порядка, то $r_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)^{n+1}$, где c некоторая точка из интервала $(x_0; x)$, тем самым получил другую запись формулы Тейлора

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + \frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)^{n+1} \qquad (2)$$

Формула (3) называется формулой Тейлора с остаточным членом в форме Лагранжа.

При $x_0 = 0$ получается частный случай формулы Тейлора – формула Маклорана:

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^{(n)}(0)}{n!}x^n + \frac{f^{(n+1)}(c)}{(n+1)!}x^{n+1}, \qquad (3)$$

Разложение по формуле Тейлора для элементарных функций.

Запишем формулу Маклорана для функции $f(x)=e^x$. Находим производные этой функции: $f'(x)=e^x$, $f''(x)=e^x$, … , $f^{(n+1)}(x)=e^x$. Так как $f(0)=e^0=1$, $f'(0)=e^0=1$, $f''(x)=e^0=1$, … , $f^{(n)}(0)=e^0=1$, $f^{(n+1)}(c)=e^c$, то по формуле (3) имеем:

$$e^x=1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots+\frac{x^n}{n!}+\frac{e^c}{(n+1)!}x^{n+1},$$

при $x=1$, $0<c<2$, $e^c<9$, оценим число e.

$$e-\left(2+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{n!}\right)<\frac{9}{(n+1)!},$$

Ранее (при изучении числа e) мы получили более точную оценку $e=2+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{n!}+\ldots+\frac{\Theta_n}{n!n}$, где $0<\Theta_n<1 \Rightarrow \quad e\approx 2+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{n!}$.

Приведем разложения по формуле Маклорана некоторых других элементарных функций:

$$f(x)=\sin x,\quad f^{(n)}(x)=\sin\left(x+\frac{\pi n}{2}\right),\quad f^{(n)}(0)=\sin\frac{\pi n}{2}=\begin{cases}-1, & n=4m+3,\\ 0, & n=2m,\\ 1, & n=1+4m.\end{cases}$$

$$\sin x=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\ldots+(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}+o\left(x^{2n}\right);$$

$$f(x)=\cos x,\quad f^{(n)}(x)=\cos\left(x+\frac{\pi n}{2}\right),\quad f^{(n)}(0)=\cos\frac{\pi n}{2}=\begin{cases}-1, & n=4m+2,\\ 0, & n=2m+1,\\ 1, & n=4m.\end{cases}$$

$$\cos x=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots+(-1)^{n-1}\frac{x^{2n}}{(2n)!}+o\left(x^{2n}\right);$$

$$f(x)=\ln(x+1),$$

$$\ln(1+x)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\ldots+(-1)^{n-1}\frac{x^n}{n}+o(x^n),$$

при $x=1$, $\ln 2\approx 1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\ldots+(-1)^{n-1}\frac{1}{n}$;

$$f(x)=(1+x)^{\alpha},$$

$$(1+x)^{\alpha}=1+\alpha x+\frac{\alpha(\alpha-1)}{2!}x^2+\ldots+\frac{\alpha(\alpha-1)\ldots(\alpha-n+1)}{n!}x^n+o\left(x^n\right);$$

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,

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

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

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

This implies,

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

Okay, so we have,

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

But, what is the range of $u^T \nabla_\theta \mathscr{L}(\theta)$? Let us see....
Let β be the angle between u and $\nabla_\theta \mathscr{L}(\theta)$, then we know that,

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

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

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

Thus, $\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) = u^T \nabla_\theta \mathscr{L}(\theta) = k * cos(\beta)$ 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

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

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

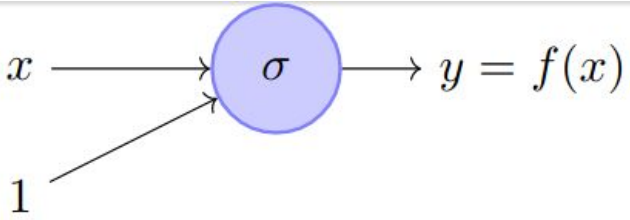
$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t, b=b_t}, \nabla b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w=w_t, b=b_t}$$

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

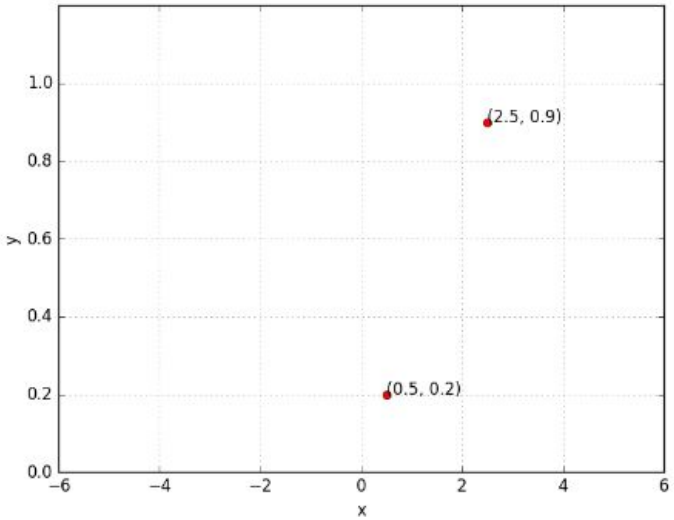
- Let us create an algorithm from this rule ...

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



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



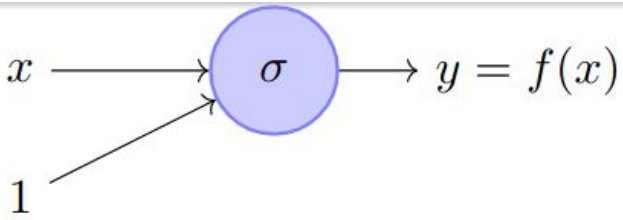
Let's assume there is only 1 point to fit (x, y)

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

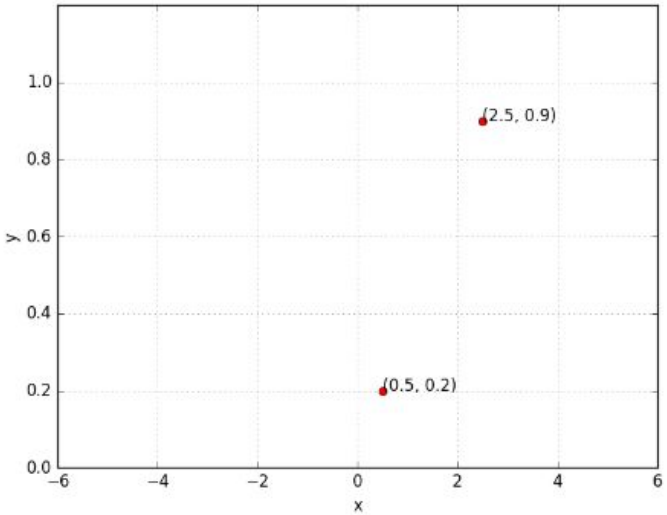
$$\nabla w = \frac{\partial \mathscr{L}(w, b)}{\partial w} = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$

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

$$
\begin{aligned}
&\frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)})) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b))) \\
&= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x) \\
&= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x) \\
&= f(x) * (1 - f(x)) * x
\end{aligned}
$$



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



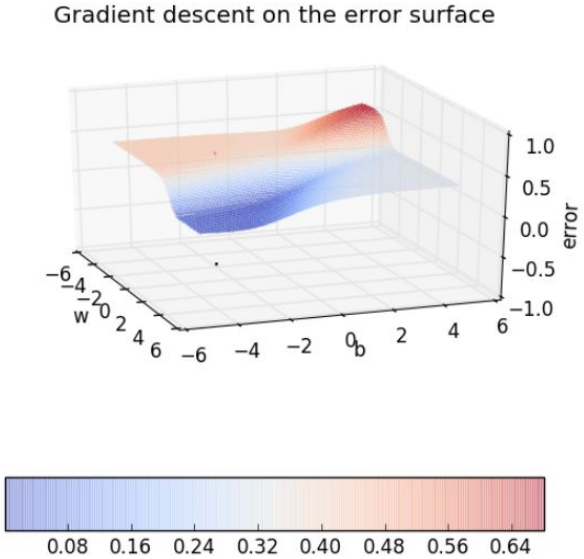
So if there is only 1 point (x, y), we have,

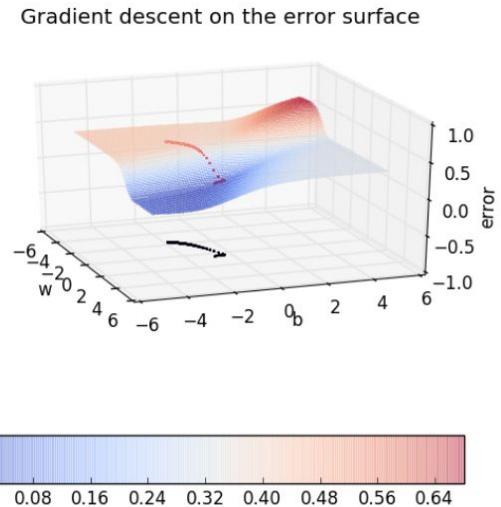
$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

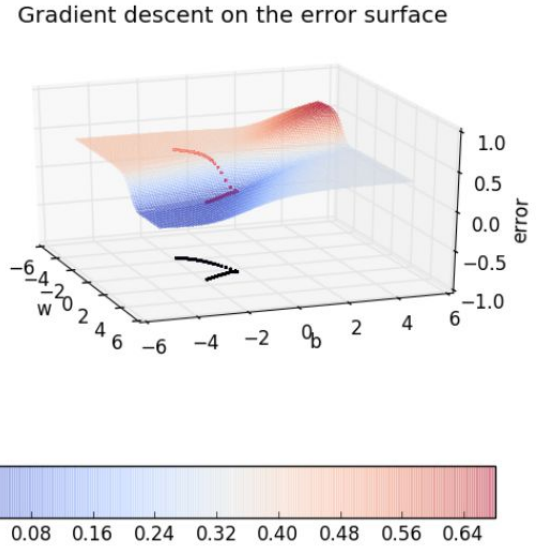
For two points,

$$\nabla w = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$$

$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$


Gradient descent on the 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


Gradient descent on the 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


Gradient descent on the 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

Perceptrons vs Sigmoid neurons

Representation power of a multilayer network of perceptrons

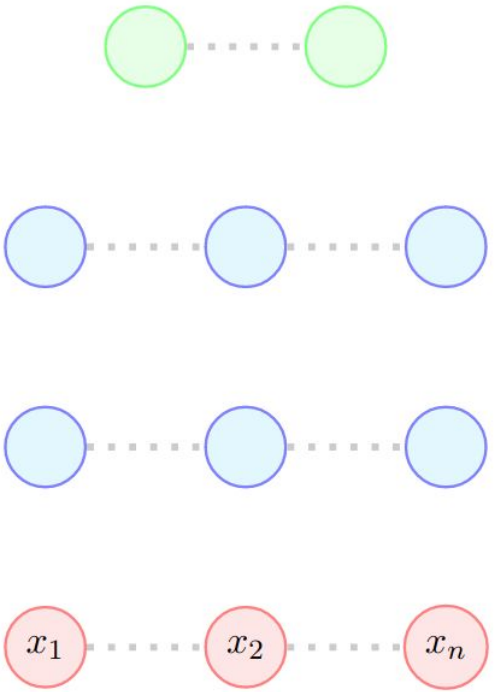
A multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors)

Representation power of a multilayer network of sigmoid neurons

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

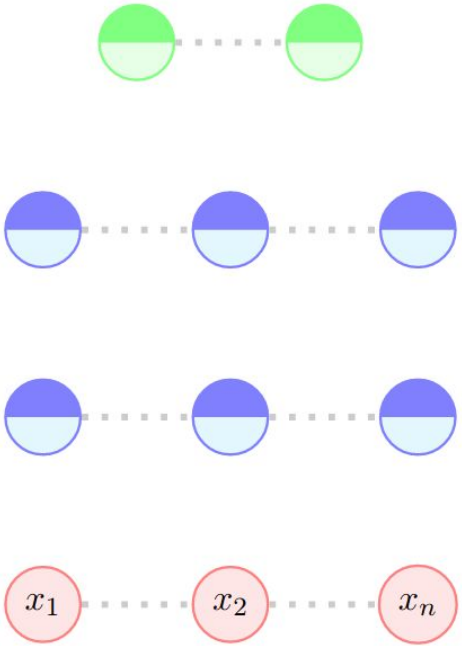
In other words, there is a guarantee that for any function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, we can always find a neural network (with 1 hidden layer containing enough neurons) whose output g(x) satisfies $|g(x) - f(x)| < \epsilon$!!

Feedforward Neural Networks (a.k.a.multilayered network of neurons)

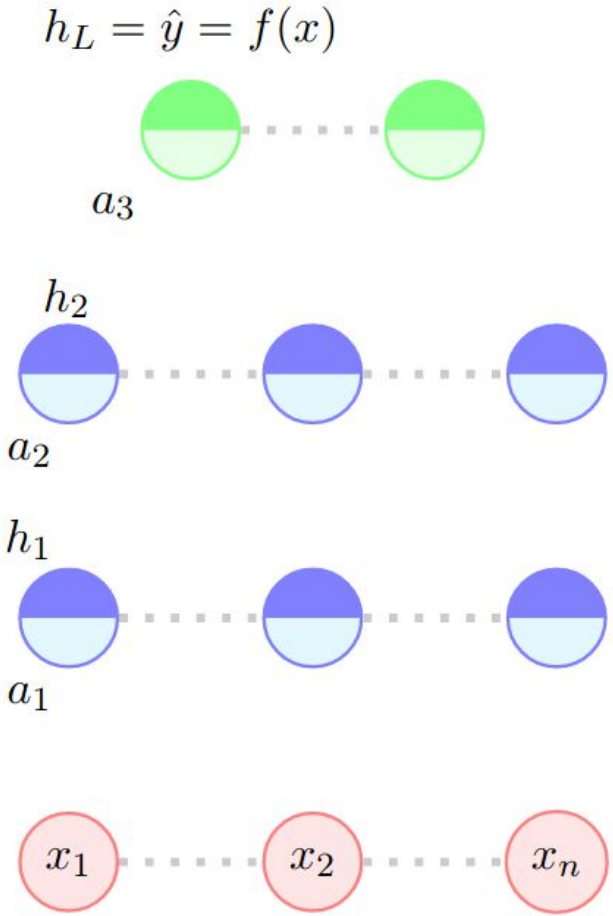


- The input to the network is an **n**-dimensional vector
- The network contains $\mathbf{L-1}$ hidden layers (2, in this case) having **n** neurons each
- Finally, there is one output layer containing **k** neurons (say, corresponding to **k** classes)

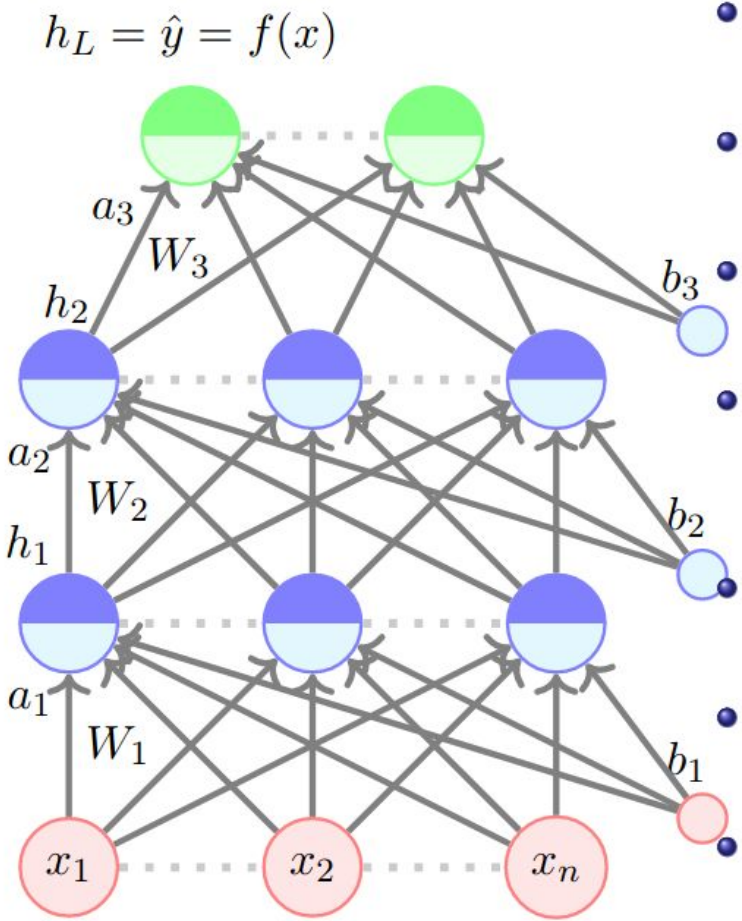
Feedforward Neural Networks



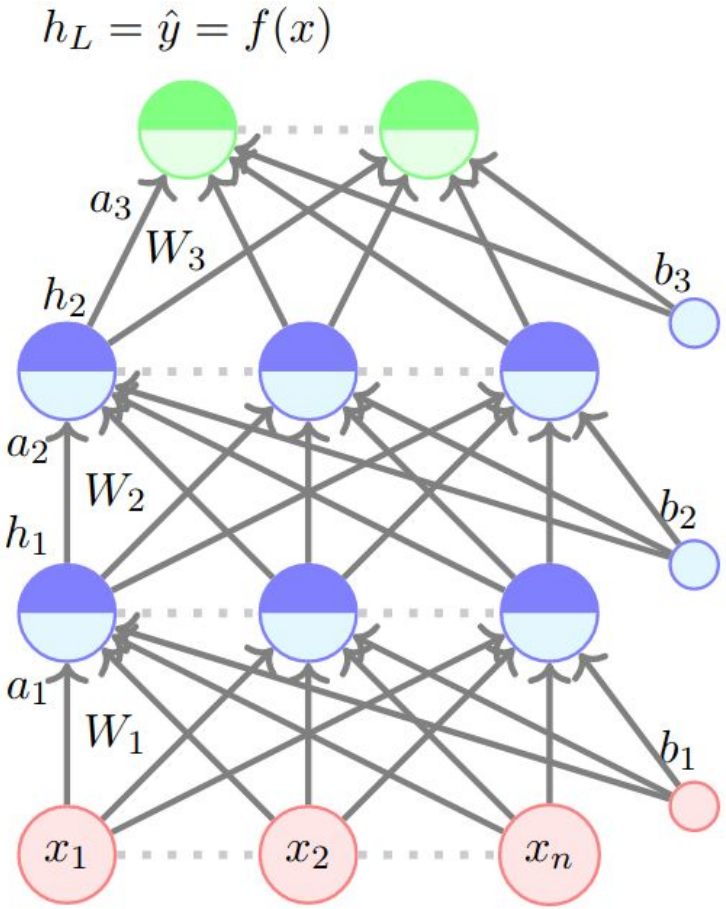
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation



- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation (a_i and h_i are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the (L)-th layer



- The input to the network is an **n**-dimensional vector
- The network contains $\mathbf{L-1}$ hidden layers (2, in this case) having **n** neurons each
- Finally, there is one output layer containing **k** neurons (say, corresponding to **k** classes)
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation (a_i and h_i are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the (L)-th layer
- $W_i \in \mathbb{R}^{n \times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and i $(0 < i < L)$
- $W_L \in \mathbb{R}^{n \times k}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer $(L = 3$ in this case)



- 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

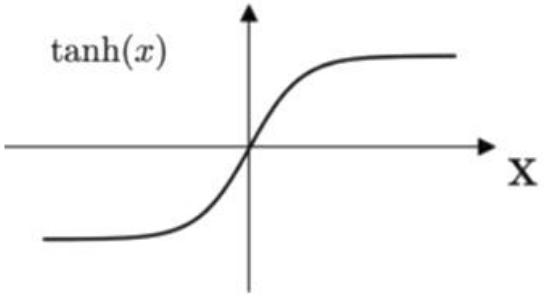
$$f(x) = h_L(x) = O(a_L(x))$$

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

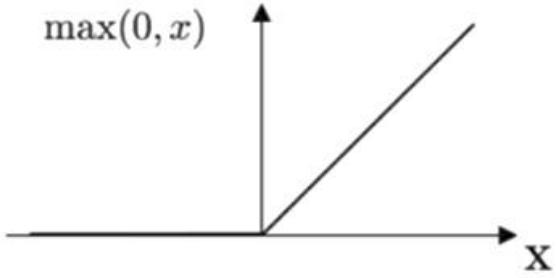
- To simplify notation we will refer to $a_i(x)$ as a_i and $h_i(x)$ as h_i

Some activation functions

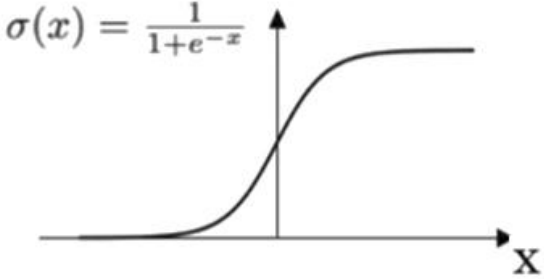
Tanh



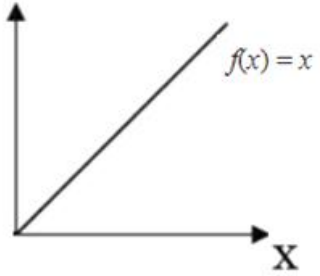
ReLU

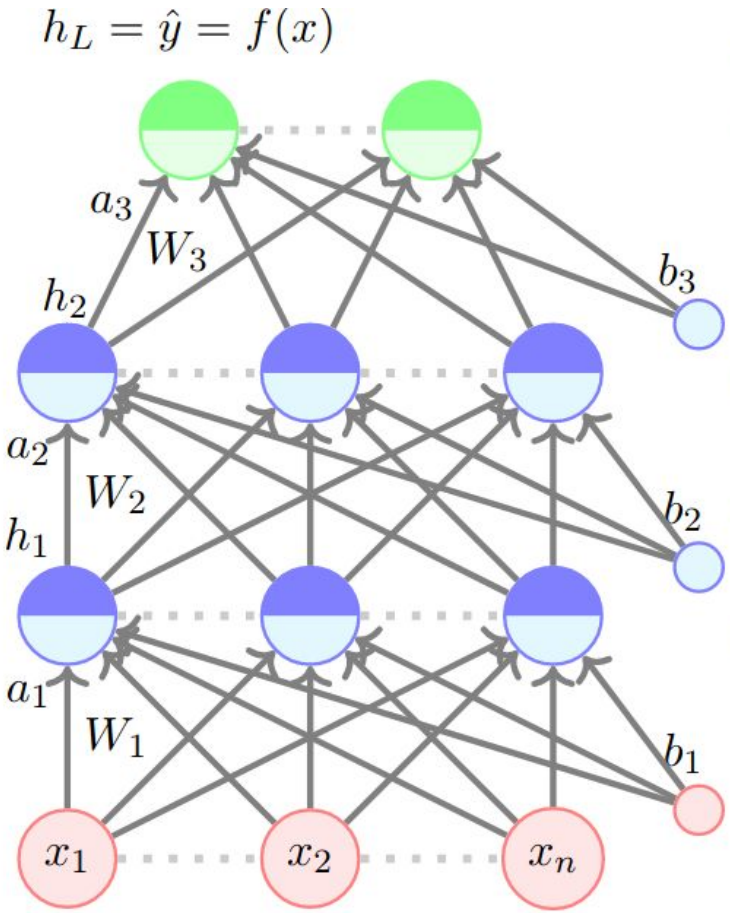


Sigmoid



Linear





- **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 (we will see soon)
- **Objective/Loss/Error function:** Say,

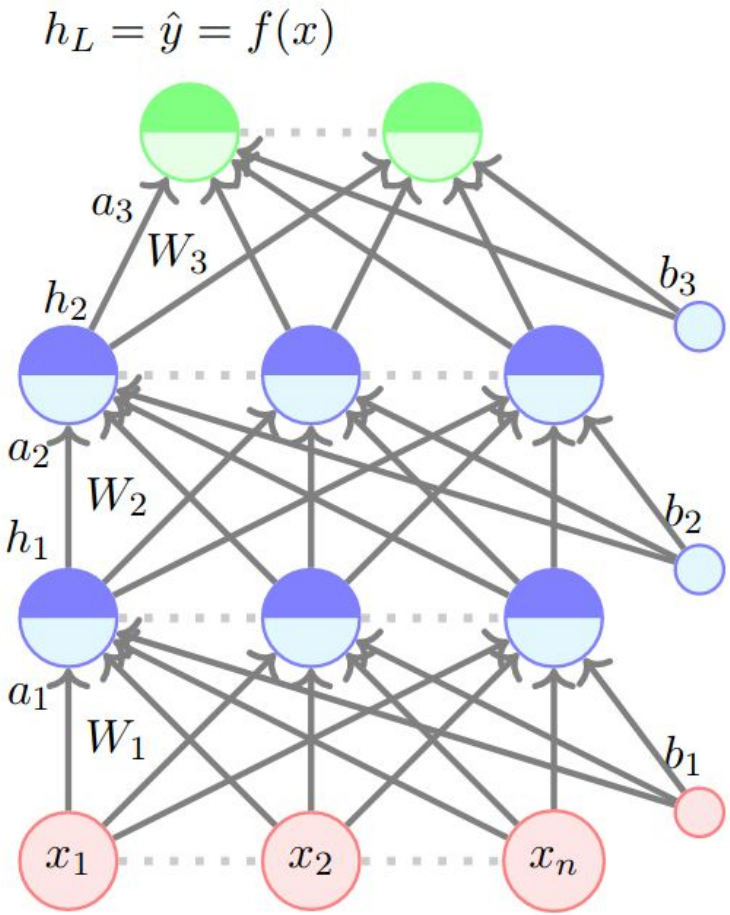
$$min \ \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} (\hat{y}_{ij} - y_{ij})^2$$

In general, $min \ \mathscr{L}(\theta)$

where $\mathscr{L}(\theta)$ is some function of the parameters

Learning Parameters of Feedforward Neural Networks

- We have introduced feedforward neural networks
- We are now interested in finding an algorithm for learning the parameters of this model

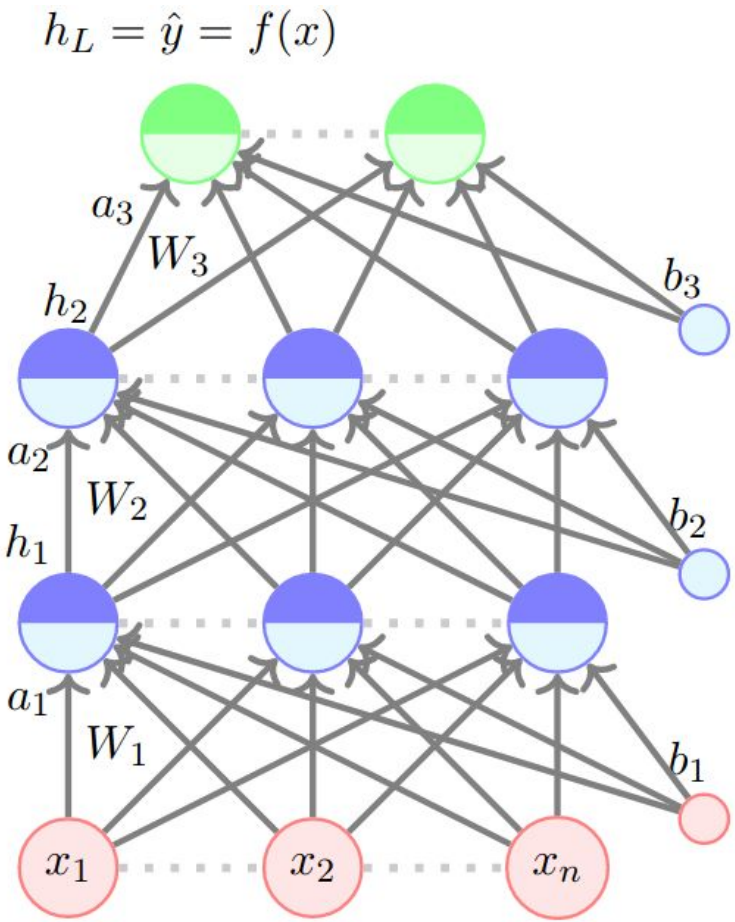


- Recall our gradient descent algorithm
- We can write it more concisely as

Algorithm: gradient_descent()

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

- where $\nabla \theta_t = \left[\frac{\partial \mathscr{L}(\theta)}{\partial w_t}, \frac{\partial \mathscr{L}(\theta)}{\partial b_t}\right]^T$
- Now, in this feedforward neural network, instead of $\theta = [w, b]$ we have $\theta = [W_1, W_2, .., W_L, b_1, b_2, .., b_L]$
- We can still use the same algorithm for learning the parameters of our model



- Recall our gradient descent algorithm
- We can write it more concisely as

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 $\nabla \theta_t = [\frac{\partial \mathscr{L}(\theta)}{\partial W_{1,t}}, .., \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,t}}, \frac{\partial \mathscr{L}(\theta)}{\partial b_{1,t}}, .., \frac{\partial \mathscr{L}(\theta)}{\partial b_{L,t}}]^T$
- Now, in this feedforward neural network, instead of $\theta = [w, b]$ we have $\theta = [W_1, W_2, .., W_L, b_1, b_2, .., b_L]$
- We can still use the same algorithm for learning the parameters of our model

$\nabla\theta$ now looks like this:

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

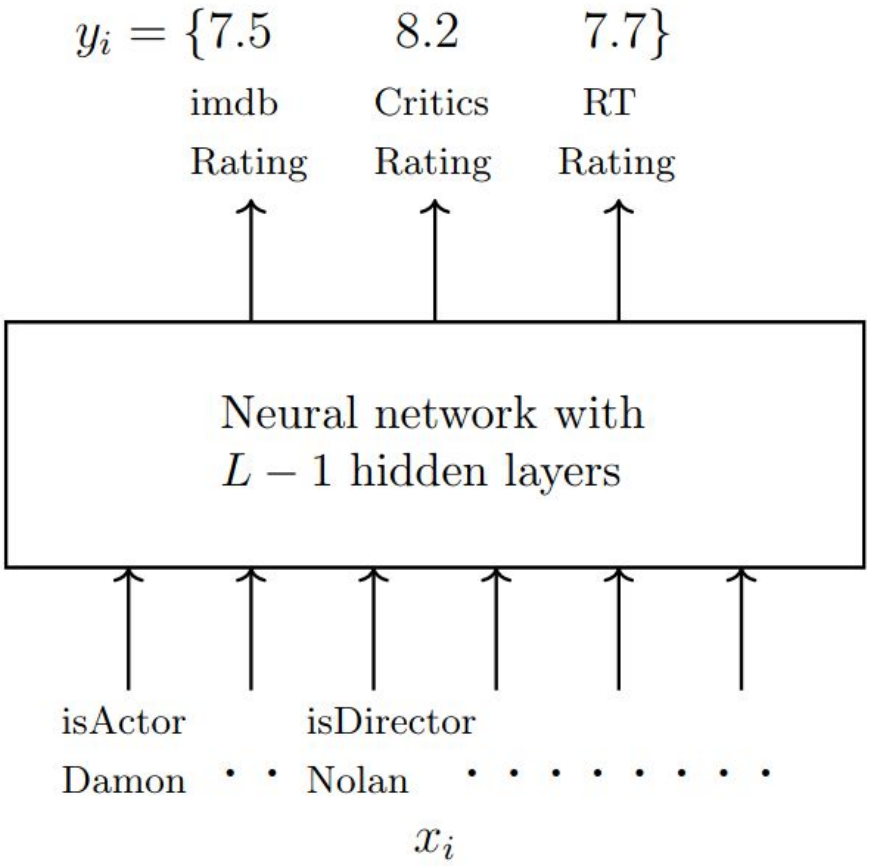
- $\nabla\theta$ is thus composed of $\nabla W_1, \nabla W_2, ...\nabla W_{L-1} \in \mathbb{R}^{n\times n}, \nabla W_L \in \mathbb{R}^{n\times k}$, $\nabla b_1, \nabla b_2, ..., \nabla b_{L-1} \in \mathbb{R}^n$ and $\nabla b_L \in \mathbb{R}^k$

We need to answer two questions

- How to choose the loss function $\mathscr{L}(\theta)$?
- How to compute $\nabla\theta$ which is 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 $\nabla b_L \in \mathbb{R}^k$?

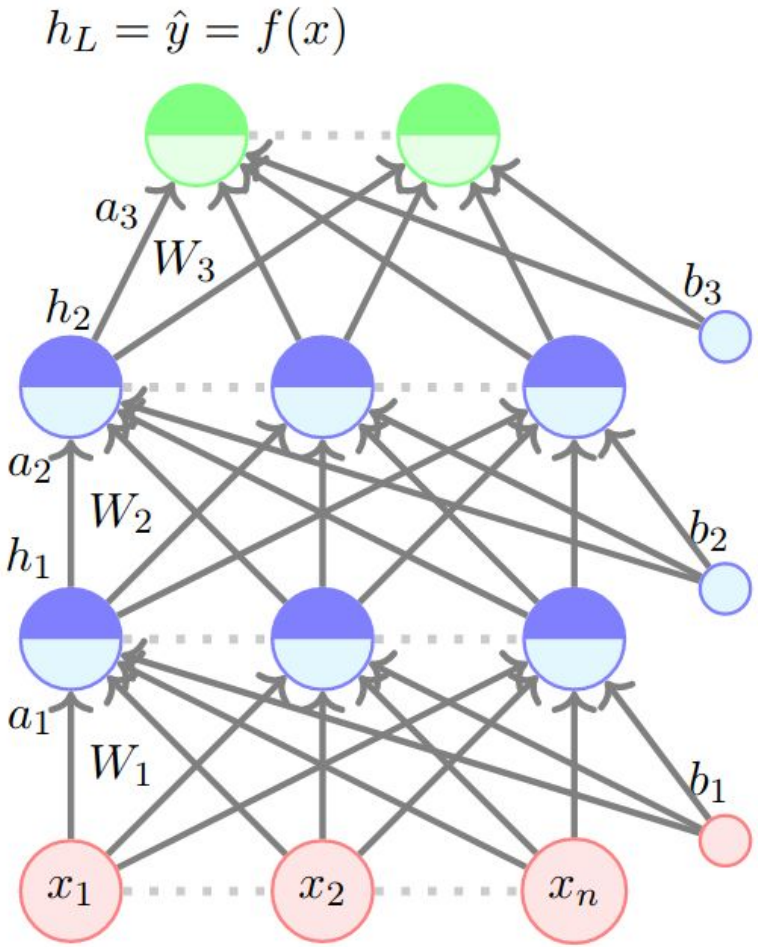
Output Functions and Loss Functions

How to choose the loss function?



- The choice of loss function depends on the problem at hand
- We will illustrate this with the help of two examples
- Consider our movie example again but this time we are interested in predicting ratings
- Here $y_i \in \mathbb{R}^3$
- The loss function should capture how much $\hat{y}_i$ deviates from y_i
- If $y_i \in \mathbb{R}^n$ then the squared error loss can capture this deviation

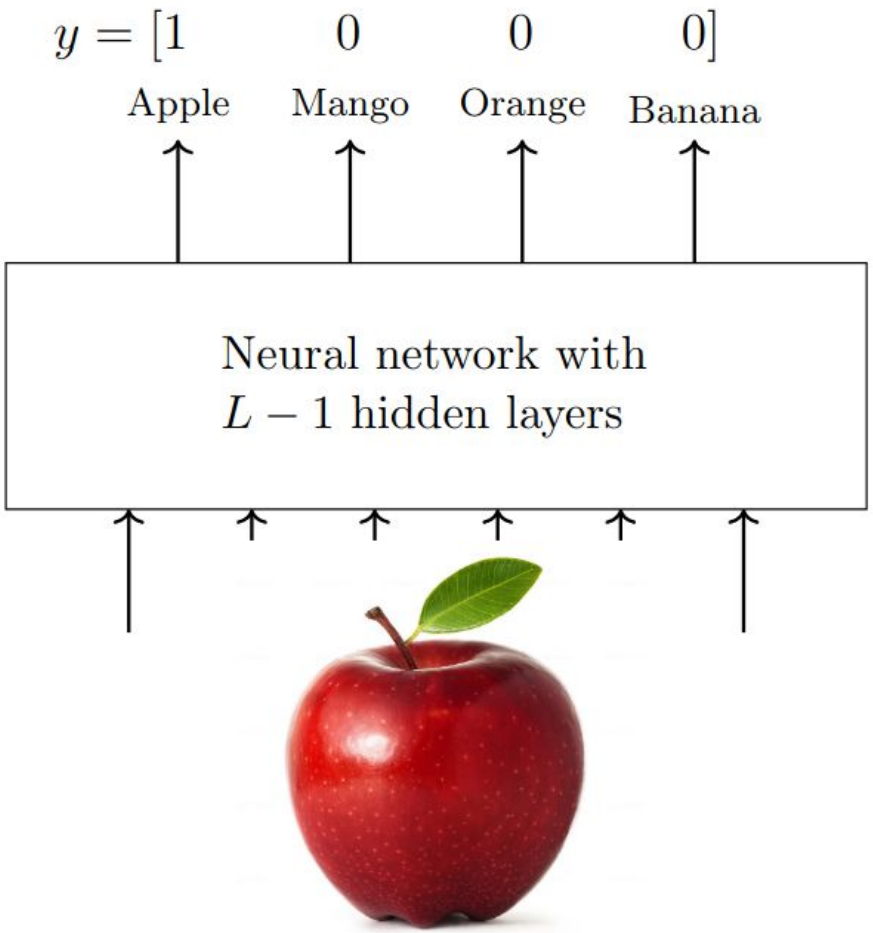
$$\mathscr{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{3}(\hat{y}_{ij} - y_{ij})^2$$



- A related question: What should the output function 'O' be if $y_i \in \mathbb{R}$?
- More specifically, can it be the logistic function?
- No, because it restricts $\hat{y}_i$ to a value between 0 & 1 but we want $\hat{y}_i \in \mathbb{R}$
- So, in such cases it makes sense to have 'O' as linear function

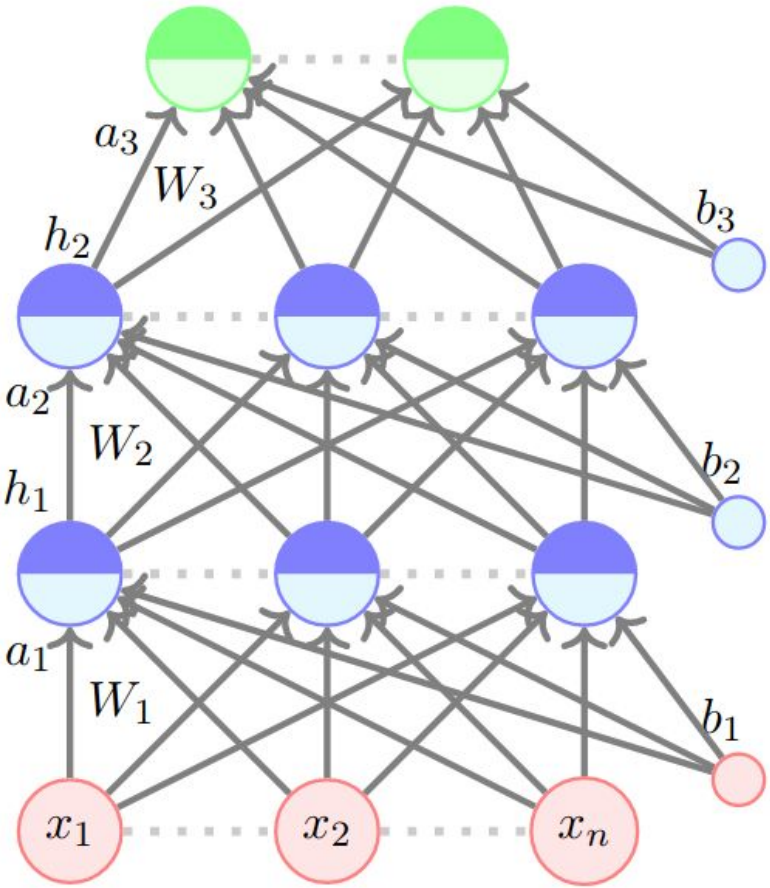
$$\begin{aligned} f(x) = h_L &= O(a_L) \\ &= W_O a_L + b_O \end{aligned}$$

- $\hat{y}_i = f(x_i)$ is no longer bounded between 0 and 1



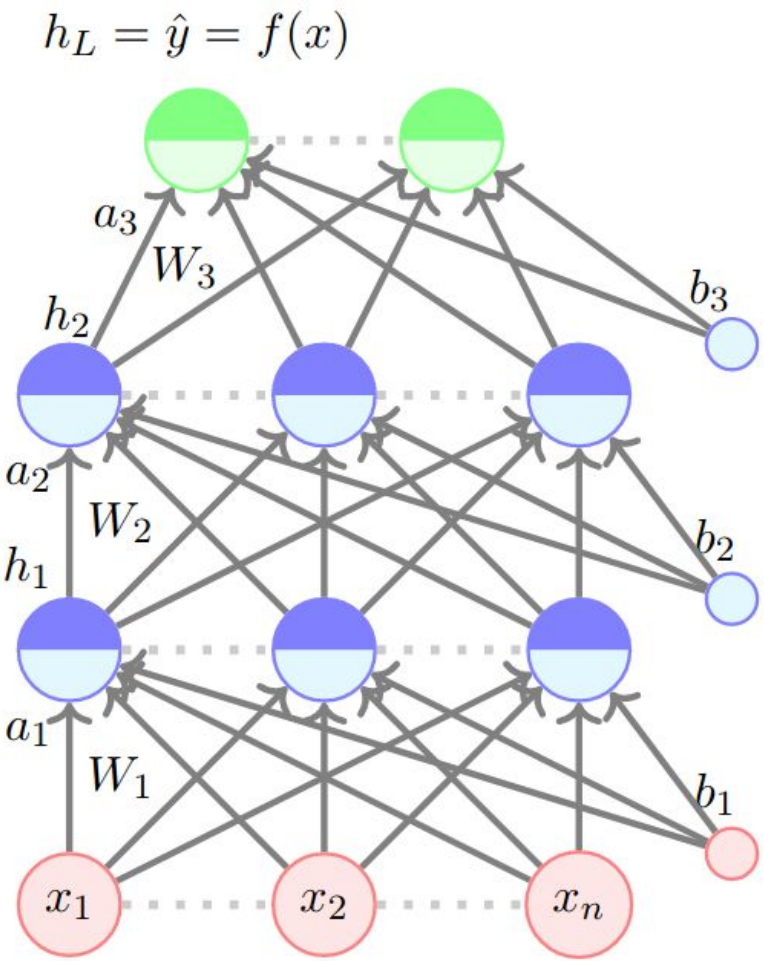
- Now let us consider another problem for which a different loss function would be appropriate
- Suppose we want to classify an image into 1 of k classes
- Here again we could use the squared error loss to capture the deviation
- But can you think of a better function?

$h_L = \hat{y} = f(x)$



- Notice that y is a probability distribution
- Therefore we should also ensure that $\hat{y}$ is a probability distribution
- What choice of the output activation 'O' will ensure this ?

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



- Notice that y is a probability distribution
- Therefore we should also ensure that $\hat{y}$ is a probability distribution
- What choice of the output activation 'O' will ensure this ?

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

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

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

- This function is called the *softmax* function

$$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

where,

y	is an input vector to a softmax function, S. It consist of n elements for n classes (possible outcomes)
y_i	the i-th element of the input vector. It can take any value between -inf and +inf
$exp(y_i)$	standard exponential function applied on y_i. The result is a small value (close to 0 but never 0) if $y_i < 0$ and a large value if y_i is large. eg • exp(55) = 7.69e+23 (A very large value) • exp(−55) = 1.30e-24 (A very small value close to 0) **Note**: exp(∗) is just $\mathbf{e}^*$ where $\mathbf{e} = 2.718$, the Euler's number.
$\sum_{j=1}^{n} \exp(y_j)$	A normalization term. It ensures that the values of output vector $S(y)_i$ sums to 1 for i-th class and each of them and each of them is in the range 0 and 1 which makes up a valid probability distribution.
n	Number of classes (possible outcomes)

- Now that we have ensured that both y & $\hat{y}$ are probability distributions can you think of a function which captures the difference between them?

Entropy

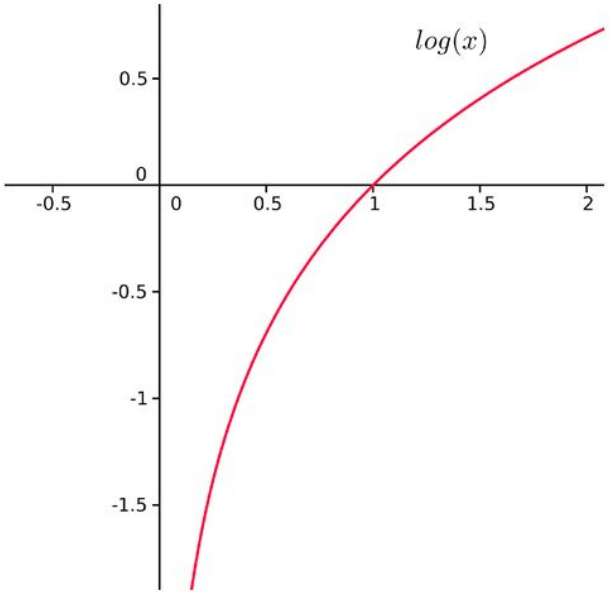
Entropy of a random variable X is the level of uncertainty inherent in the variables possible outcome.

For `p(x)` — probability distribution and a random variable X, entropy is defined as follows

$$H(X) = \begin{cases} -\int_x p(x) \log p(x), & \text{if } X \text{ is continous} \\ -\sum_x p(x) \log p(x), & \text{if } X \text{ is discrete} \end{cases}$$

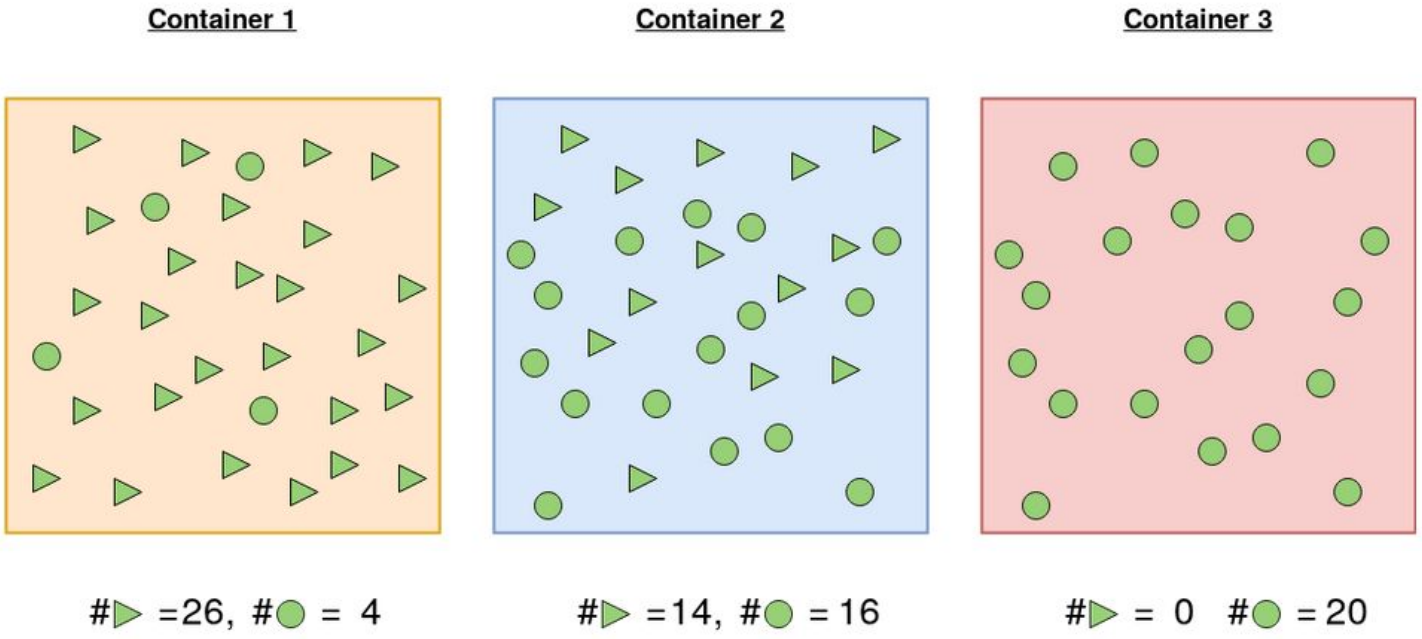
Reason for negative sign

- **log**(p(x))<0 for all p(x) in (0,1)
- p(x) is a probability distribution and therefore the values must range between 0 and 1.



The greater the value of entropy H(x), the greater the uncertainty for probability distribution, the smaller the value the less the uncertainty.

Example



What's the probability of picking a triangle from first container?

Container 1: The probability of picking a triangle is 26/30 and the probability of picking a circle is 4/30. For this reason, the probability of picking one shape and/or not picking another is more certain.

Container 2: Probability of picking the a triangular shape is 14/30 and 16/30 otherwise. There is almost 50–50 chance of picking any particular shape. Less certainty of picking a given shape than in 1.

Container 3: A shape picked from container 3 is surely a circle. Probability of picking a circle is 1 and the probability of picking a triangle is 0. It is perfectly certain than the shape picked will be circle.

Entropy for container 1

$$
\begin{aligned}
H(X) &= -\sum_{x} p(x)\log(p(x)) \\
&= -[p(x_1)\log_2(p(x_1)) + p(x_2)\log_2(p(x_2))] \\
&= -\left[\frac{26}{30}\log_2\left(\frac{26}{30}\right) + \frac{4}{30}\log_2\left(\frac{4}{30}\right)\right] \\
&= 0.5665
\end{aligned}
$$

Entropy for container 2

$$
\begin{aligned}
H(X) &= -\sum_{x} p(x)\log(p(x)) \\
&= -[p(x_1)\log_2(p(x_1)) + p(x_2)\log_2(p(x_2))] \\
&= -\left[\frac{14}{30}\log_2\left(\frac{14}{30}\right) + \frac{16}{30}\log_2\left(\frac{16}{30}\right)\right] \\
&= 0.9968
\end{aligned}
$$

Entropy for container 3

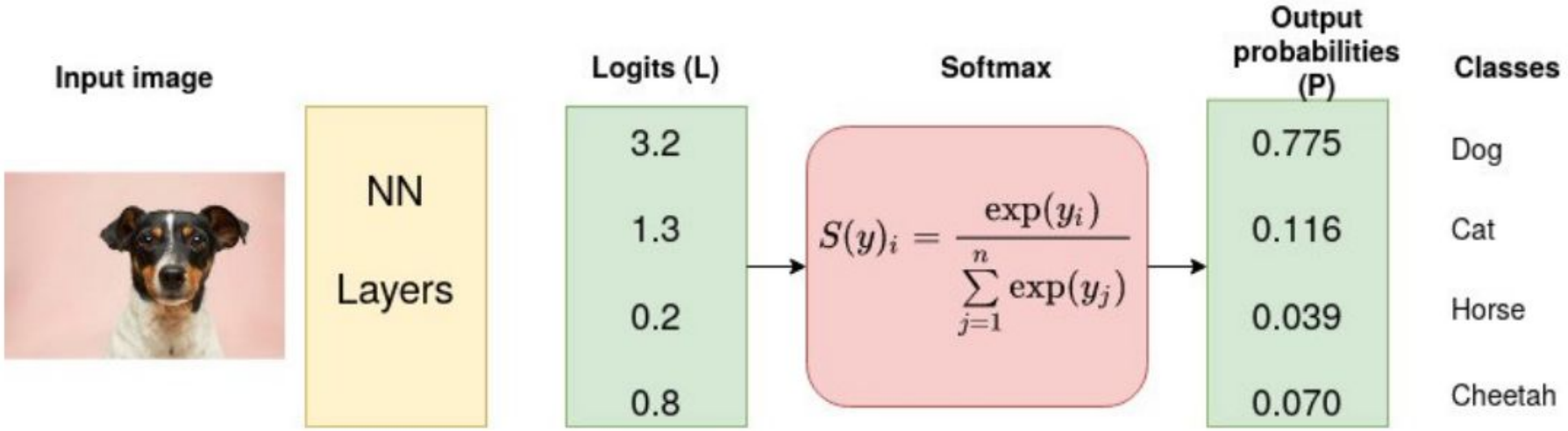
$$
\begin{aligned}
H(X) &= -\sum_{x} p(x)\log(p(x)) \\
&= -[p(x_1)\log_2(p(x_1)) + p(x_2)\log_2(p(x_2))] \\
&= -\left[\frac{20}{20}\log_2\left(\frac{20}{20}\right) + \frac{20}{20}\log_2\left(\frac{20}{20}\right)\right] \\
&= 0
\end{aligned}
$$

Cross-entropy is defined as

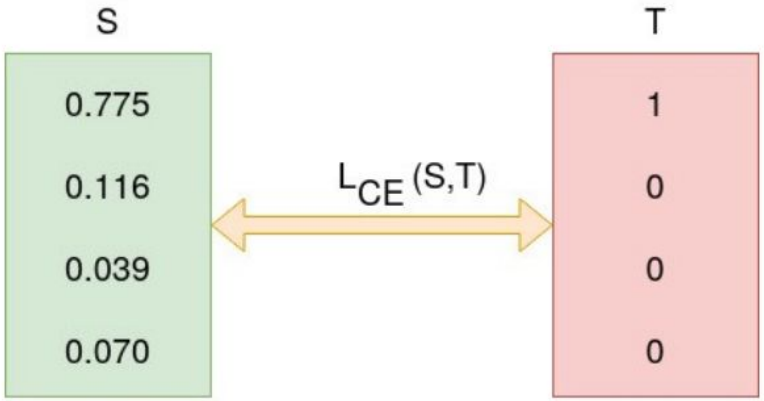
$$L_{\text{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes,}$$

where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

Example


Input image
NN
Layers
Logits (L)
3.2
1.3
0.2
0.8
Softmax
$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$
Output probabilities (P)
0.775
0.116
0.039
0.070
Classes
Dog
Cat
Horse
Cheetah

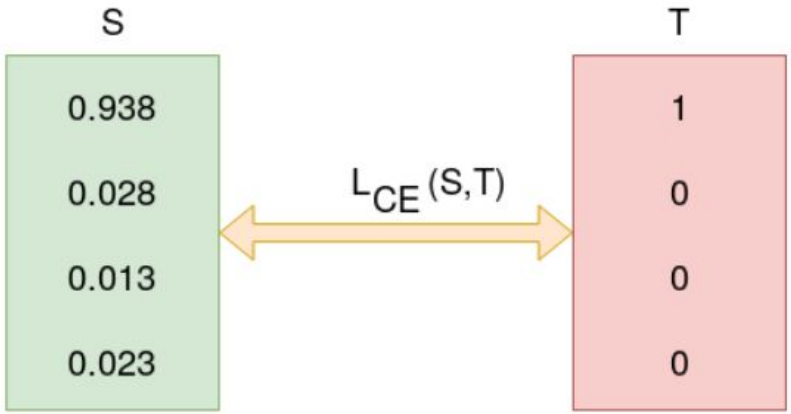
Example



The categorical cross-entropy is computed as follows

$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

Assume that after some iterations of model training the model outputs the following vector of logits



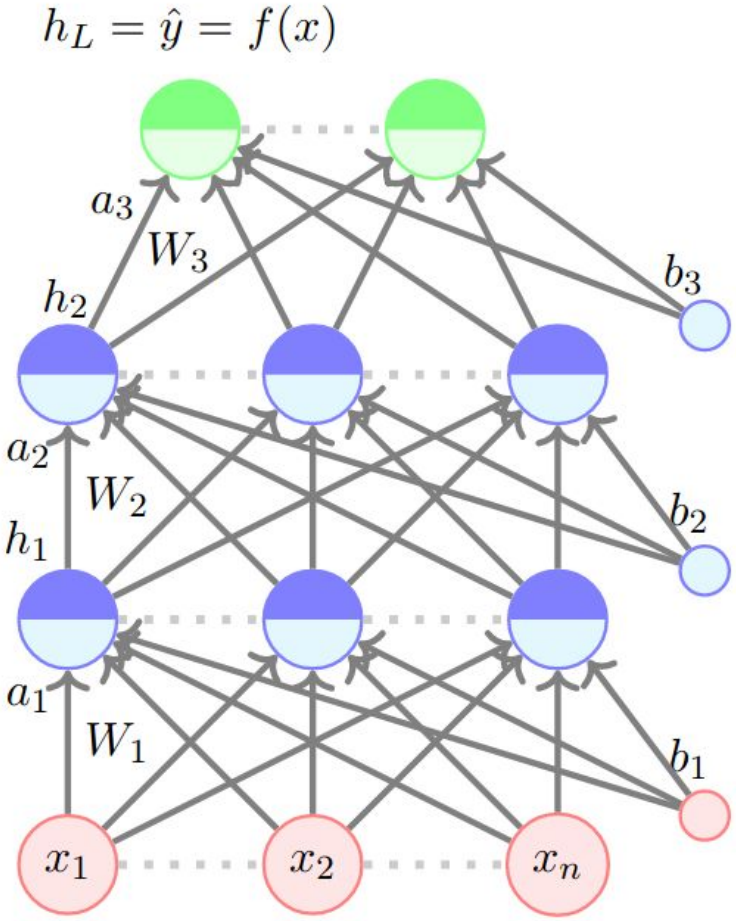
$$L_{CE} = -1 \log_2(0.936) + 0 + 0 + 0$$
$$= 0.095$$

- Cross-entropy

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

- Notice that

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



- So, for classification problem (where you have to choose 1 of K classes), we use the following objective function

$$\underset{\theta}{\text{minimize}} \quad \mathscr{L}(\theta) = -\log \hat{y}_\ell$$

$$\text{or} \quad \underset{\theta}{\text{maximize}} \quad -\mathscr{L}(\theta) = \log \hat{y}_\ell$$

- But wait!
  Is $\hat{y}_\ell$ a function of $\theta = [W_1, W_2, ., W_L, b_1, b_2, ., b_L]$?
- Yes, it is indeed a function of θ

$$\hat{y}_\ell = [O(W_3 g(W_2 g(W_1 x + b_1) + b_2) + b_3)]_\ell$$

- What does $\hat{y}_\ell$ encode?
- It is the probability that x belongs to the ℓ^{th} class (bring it as close to 1).
- $\log \hat{y}_\ell$ is called the *log-likelihood* of the data.

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

Thank you