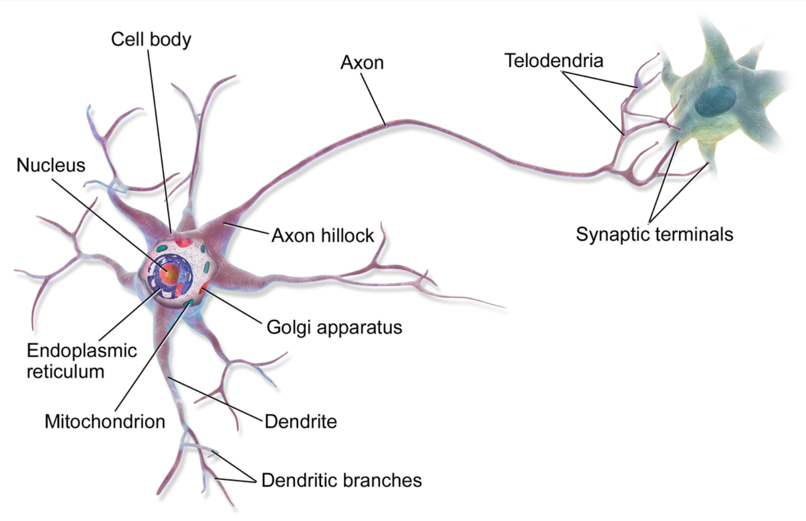
Supervised learning

CM5 : Perceptron

Not about


Cell body
Axon
Telodendria
Nucleus
Axon hillock
Synaptic terminals
Golgi apparatus
Endoplasmic
reticulum
Mitochondrion
Dendrite
Dendritic branches

Section 1

Background

Supervised learning

We are given a **training set** of N examples input-output pairs

$$(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$$

where each pair was generated by an unknown function f:

$$y = f(x)$$

Objective: discover a function h, the **hypothesis** that approximates the true function f.

Running example

Let say that I want to sell my apartment in the center of Toulouse.

My apartment[1] has:

- 165 m^2
- 4 rooms
- on the 6th floor

Question: what is the market price for such an apartment?

[1]complety fictional, I am just an associate professor

Dataset

I look at several apartments on sell in the neighborhood and come up with the following dataset:[2]

X			Y
m^2	Num Rooms	Floor	Price (€)
24	1	4	102 000
46	3	2	140 000
50	3	6	353 600
211	5	3	892 000
74	3	1	198 000

[2]Source: leboncoin.fr

Vocabulary

Each example i has 3 **features** (m^2, #rooms, floor)

$$x_i = [x_{i,1}, x_{i_2}, x_{i,3}]$$

and associated **ground truth** y_i.

The *unknown* function f associates each example with its ground truth:

$$f(x_i) = y_i$$

For instance: $f([24, 1, 4]) = 102\text{k€}$

I want to know the market price for my apartment: $f([165, 4, 6]) = \;??$

Hypothesis space

I want to **learn** a function h that closely approximates f.

$$\text{i.e.} \quad h(x) \approx f(x), \forall x \qquad \text{(more complex in practice)}$$

Among the set of all possible functions $\mathcal{H}$, I want to choose the one that has the least different behavior from f:

$$h^* = \arg\min_{h \in \mathcal{H}} \; \textit{diff}(h, f)$$

$\mathcal{H}$, the set of all possible functions, is called the **hypothesis space**.

What's a suitable hypothesis space?

The set of possible python functions:

```python
def h(sqm, nroom, floor):
    price = 4000 * sqm + 10000 * nrooms
    if floor == 1:
        price -= 30000
    return price
```

The set of linear functions:

$$h(sqm, nrooms, floor) = 4000 * sqm + 10000 * nrooms + 10000 * floor$$

. . . or the set of possible decision trees.

What's a suitable hypothesis space?

Each hypothesis space has its own characteristics:

- **bias:** tendency to underfit the data.
  - linear function strongly limit the possible hypotheses which could result in a failure to fit the data
- **variance**: tendency to overfit the data
  - python is turing complete and could be made to produce exactly the ground truth for each example

Rule of thumb:

- simple model:
  - high-bias, low variance / poor-fit but generalizes well
- complex model:
  - low-bias, high variance / great fit but generalizes poorly
- (Deep) neural network: complex model that (sometimes) generalizes well

What's a good hypothesis in $\mathcal{H}$?

Given a prediction

$$\hat{y} = h(x)$$

The **loss function** measures how bad it is to have the prediction $\hat{y}$ instead of the true value y for the example x.

$$L(x, y, \hat{y})$$

It is often stated independently of x: $L(y, \hat{y})$

Some common loss functions

Absolute-value loss	$L_1(y, \hat{y}) = \lvert y - \hat{y} \rvert$
Squared-error loss	$L_2(y, \hat{y}) = (y - \hat{y})^2$
0/1 loss	$L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1

Note that for any (well-formed) loss function:

$$L(y, y) = 0$$

i.e., nothing is lost if the prediction is perfect.

Evaluating a hypothesis: the perfect measure

An agent should choose the hypothesis that minimizes the expected loss over all input-output pairs it **will** see.

Let $\mathcal{E}$ be the set of all possible examples and $P(X, Y)$ be a probability distribution over examples.

We can define the **generalization loss** for a hypothesis h and a loss function L:

$$GenLoss_L(h) = \sum_{(x,y)\in\mathcal{E}} L(y, h(x)) \times P(x, y)$$

The best hypothesis is the one with the minimum expected generalization loss:

$$h^* = \arg\min_{h\in\mathcal{H}} GenLoss_l(h)$$

Evaluating a hypothesis: the empirical measure

Problem: $P(X, Y)$ is usually unknown. Instead we only have a set of examples E.

The **empirical loss** is an estimate of the generalization loss on a set of examples E:

$$EmpLoss_{L,E}(h) = \sum_{(x,y)\in E} L(y, h(x)) \times \frac{1}{|E|}$$

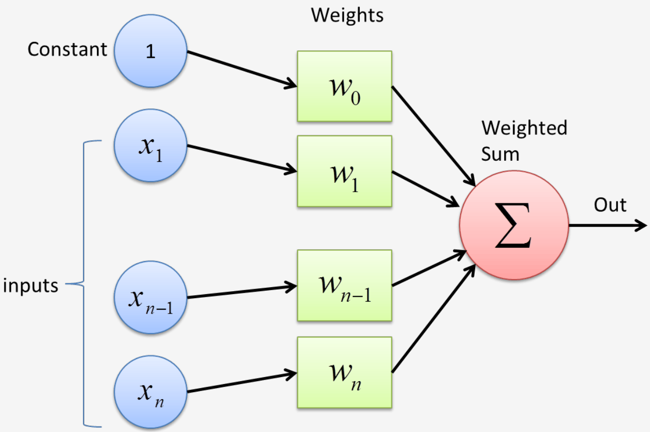
The estimated best hypothesis $\hat{h}^*$ is the one with the minimum empirical loss:

$$\hat{h}^* = \arg\min_{h\in\mathcal{H}} EmpLoss_{L,E}(h)$$

Section 2

The perceptron (regression)

Where we see a first neuron



$$h_w(x) = w_0 + w_1 \times x_1 + w_2 \times x_2 + \cdots + w_n \times x_n$$

Characterisation

The elements of the **w** vector are called the weights of the perceptron.

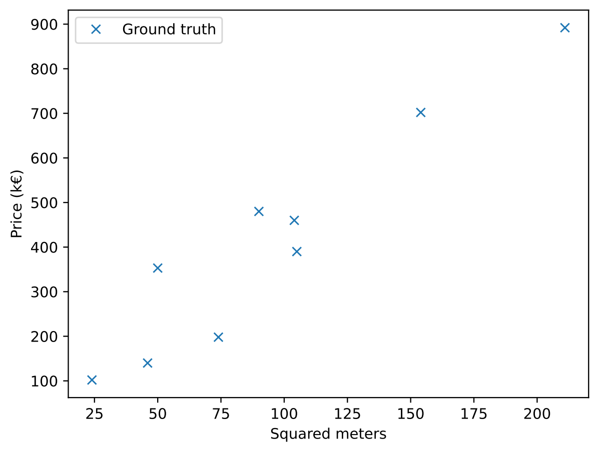
Given an input vector x, the output of the perceptron is a linear combination of its input.

$$h_w(x) = w_0 + w_1 \times x_1 + w_2 \times x_2 + \cdots + w_n \times x_n$$

The set of functions representable by a perceptron is the set of linear functions. That's our hypothesis space $\mathcal{H}_{perc}$.

An even simpler dataset

X	Y
m^2	Price (€)
24	102 000
46	140 000
50	353 600
211	892 000
74	198 000



Perceptron for predicting the price

We have a single feature (x_1: squared meters), thus a function representable by a perceptron would have the form:

$$h_w(sqm) = w_0 + w_1 \times x_1$$

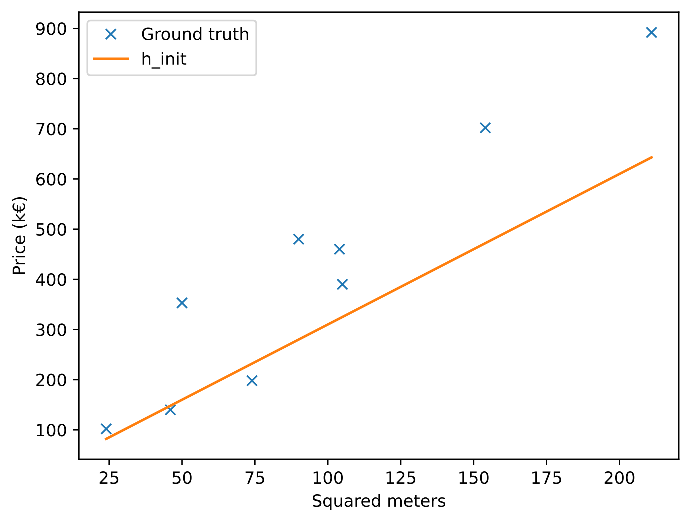
where we can interpret:

- w_0 as the base price
- w_1 as the price per squared meters

Perceptron for predicting the price

Making an educated guess we could set:

$$w_0 = 10000 \quad (€)$$
$$w_1 = 3000 \quad (€/m^2)$$

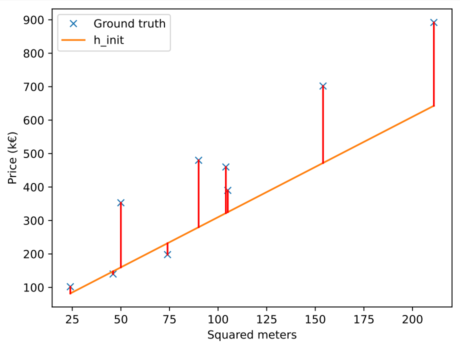


Perceptron: prediction error

For an example (x, y) and predictor h_w

prediction error $= |y - h_w(x)|$ (length of red segments)

L_2 loss: $L_2(y, \hat{y}) = (y - h_w(x))^2$ (squared error)



This leads to the empirical loss (for L_2) over the entire dataset E:

$$EmpLoss_{L_2,E}(h_w) = \sum_{(x,y)\in E} \frac{(y - h_w(x))^2}{|E|}$$

Finding the best hypothesis

I want the hypothesis $\hat{h}^*$, with the minimum empirical loss:

$$\hat{h}^* = \underset{h_w \in \mathcal{H}_{perc}}{\arg\min}\, EmpLoss_{L_2,E}(h_w)$$

For the perceptron, this means finding the best weights $\hat{w}^*$ in the weight space.

$$\hat{w}^* = \underset{w}{\arg\min}\, EmpLoss_{L_2,E}(h_w)$$

Posing $Loss(w) = EmpLoss_{L_2,E}(h_w)$, we obtain:

$$\hat{w}^* = \underset{w}{\arg\min}\, Loss(w)$$

Gradient: direction of steepest ascent

In any point w of the function, the gradient defines the direction of steepest ascent:

$$\vec{\nabla} g(w)$$

It can be computed from the partial derivatives:

$$\vec{\nabla} g(w) = \begin{bmatrix} \frac{\delta}{\delta w_0} g(w) \\ \frac{\delta}{\delta w_1} g(w) \\ \vdots \\ \frac{\delta}{\delta w_m} g(w) \end{bmatrix}$$

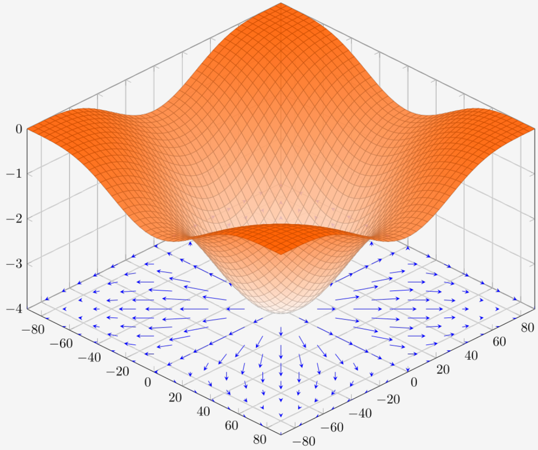


Figure: Gradient of $f(x,y) = -(cos^2(x) + cos^2(y))^2$

Gradient descent

From a point w, compute a new candidate w' by following the direction of steepest descent (opposite of the gradient).

$$w' = w - \alpha \times \vec{\nabla} g(w)$$

The distance traveled is parameterized by the **step size** α.

Since the function is decreasing in this direction, there is a good chance that

$$g(w') < g(w)$$

Gradient descent

Applying this repeatedly, we get the gradient descent algorithm:[3]

$w \leftarrow$ any value in the parameter space
while not converged **do**
  $w \leftarrow w - \alpha \times \vec{\nabla} Loss(w)$
end while

Typical convergence criteria: stop when the update did not provide an improvement for the last k iterations (e.g. $k = 5$).

[3] Recal that $Loss(w)$ is shortcut for $EmpLoss_{L,E}(h_w)$

Computing the gradient (L_2 loss, single example)

Partial derivative of the L_2 loss for a single example (x, y):

$$\begin{aligned}\frac{\delta}{\delta w_i} Loss(w) &= \frac{\delta}{\delta w_i}(y - h_w(x))^2 \\ &= 2(y - h(_w(x)) \times \frac{\delta}{\delta w_i}(y - h_w(x))\end{aligned}$$

Applied to our system with a single feature (x_1) we obtain:

$$\begin{aligned}\frac{\delta}{\delta w_0} Loss(w) &= -2(y - h_w(x)) \\ \frac{\delta}{\delta w_1} Loss(w) &= -2(y - h_w(x)) \times x_1\end{aligned}$$

Update rules

Updating the weights based on a single example:[4]

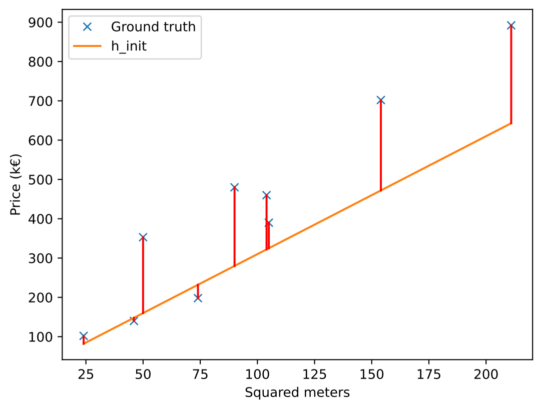
$$w_0 \leftarrow w_0 + \alpha \times (y - h_w(x))$$
$$w_1 \leftarrow w_1 + \alpha \times (y - h_w(x)) \times x_1$$

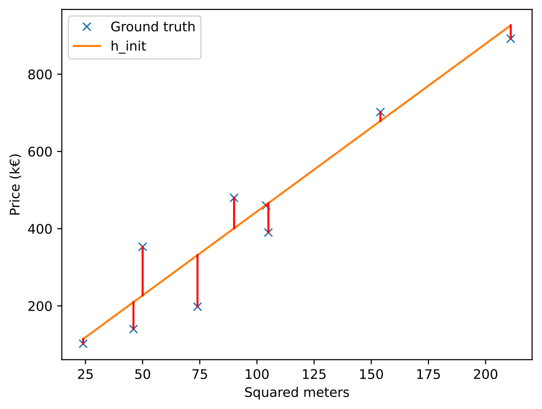
Updating the weights based on the entire training set E:

$$w_0 \leftarrow w_0 + \alpha \times \sum_{(x,y) \in E} (y - h_w(x))$$
$$w_1 \leftarrow w_1 + \alpha \times \sum_{(x,y) \in E} (y - h_w(x)) \times x_1$$

[4] Note that the -2 factor from the previous equation is included in α term.

Updating our initial guess[5]





$$w = [10,000, 3,000]$$

$$w' = [10010.53, 4343.82]$$

[5] With $alpha = 10^{-5}$

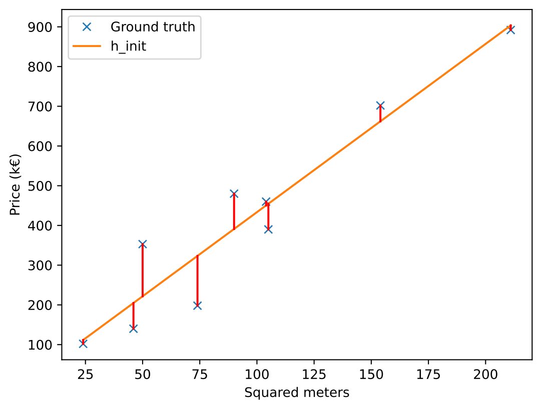
Result of the gradient descent

Repeating the gradient descent step, we eventually converge to our best solution.

$$\hat{w}^* = [9947.29, 4235.02]$$

I should sell my apartment for:

$$9947.29 + 4235.02 \times 165 = 708725€$$



Gradient descent (applied to a perceptron with L_2 loss)

$w \leftarrow$ any value in the parameter space
while not converged **do**
    $w_0 \leftarrow w_0 + \alpha \times \sum_{(x,y)\in E}(y - h_w(x))$
    for $i \in 1 \ldots n$ **do**
        $w_i \leftarrow w_i + \alpha \times \sum_{(x,y)\in E}(y - h_w(x)) \times x_i$
    end for
end while

Representation trick

In the previous slides, we always had to deal with the w_0 weight specially because it has no corresponding feature.

We can define an artificial feature x_0 that always has the value 1. And reformulate h_w:

$$h_w(x) = \sum_{i \in [0,m]} w_i \times x_i$$

$$h_w(x) = w \cdot x \qquad \text{(dot product)}$$

and the update rule (for L_2 loss):

$$w_i \leftarrow w_i + \alpha \times \sum_{(x,y) \in E} (y - h_w(x)) \times x_i$$

Section 3

A perceptron for classification

A classification problem

I now want to buy a new apartment to replace the one I just sold. To be reactive I built an automated system that sends me any new announce of an apartment for sale.

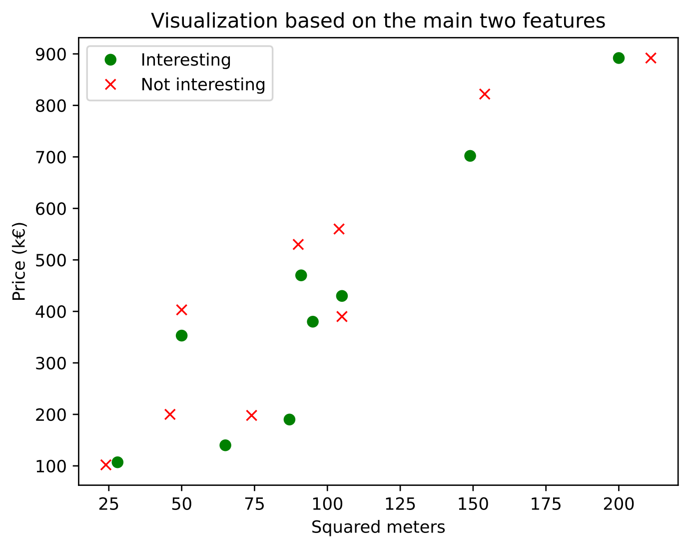
- Problem: there are dozens of announces every day and I don't have time to look at them all.
- Solution: build an AI system that will predict whether I will be interested in a particular apartment based on a few of its features. If it predicts that I am not interested, it will discard the announce.

A classification problem: dataset

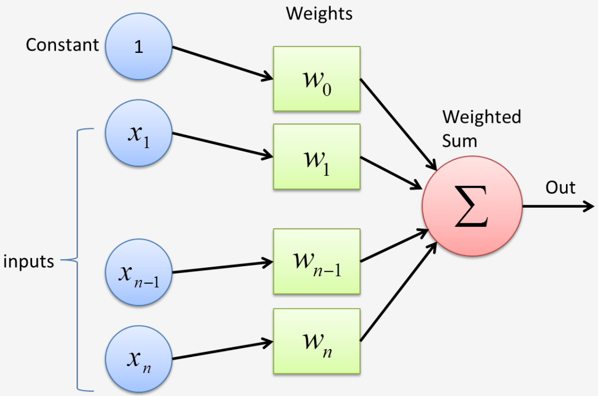
So far, I collect the following information stating whether an announce that I previously saw was interesting.

X				Y
m^2	Num Rooms	Floor	Price (€)	Interesting
24	1	4	102 000	true
46	3	2	140 000	false
50	3	6	353 600	false
211	5	3	892 000	true
74	3	1	198 000	true

A classification problem: dataset visualization



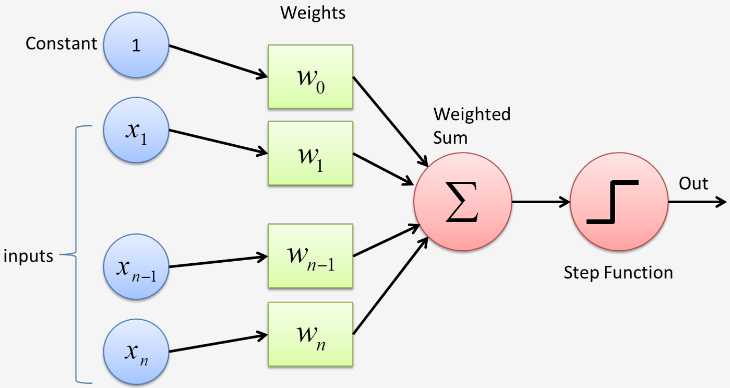
Our previous perceptron



$$h_w(x) = w_0 + w_1 \times x_1 + w_2 \times x_2 + \cdots + w_n \times x_n$$

$$h_w(x) = w \cdot x$$

Perceptron for classification

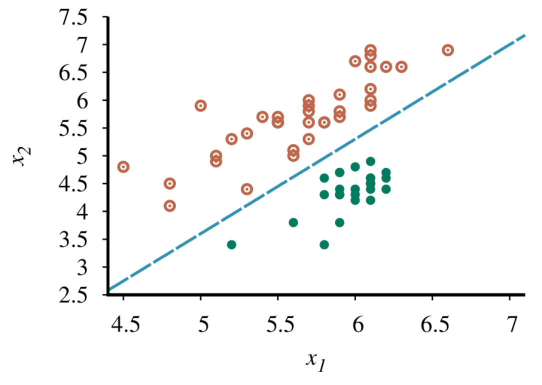


$$h_w(x) = Step(w \cdot x)$$

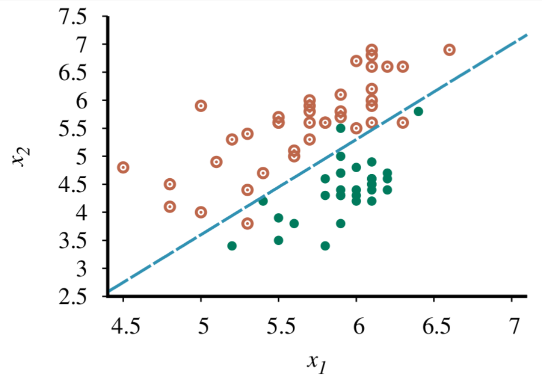
where $Step(z) = 1$ if $z \geq 0$ and 0 otherwise

The perceptron as a linear classifier

The perceptron defines a **decision boundary** that separates two classes.



Linearly separable
Perfectly classifiable by a perceptron



Not linearly separable
Not perfectly classifiable by a perceptron

The step function

$$h_w(x) = Step(w \cdot x)$$

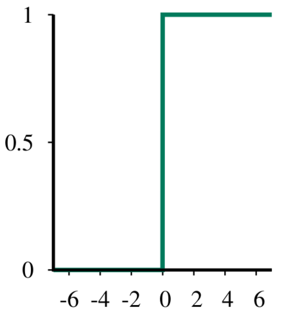
where $Step(z) = 1$ if $z \geq 0$ and 0 otherwise

We now have a function that we could train in order to output:

- 1 if the example is in the class (interesting)
- 0 otherwise (not interesting)

Problem: the function is:

- non-differentiable in 0
- the gradient is 0 everywhere else



$Step(z)$

The perceptron learning rule

Nevertheless, an rule was proposed the **perceptron update rule** (here for a single example (x, y)):

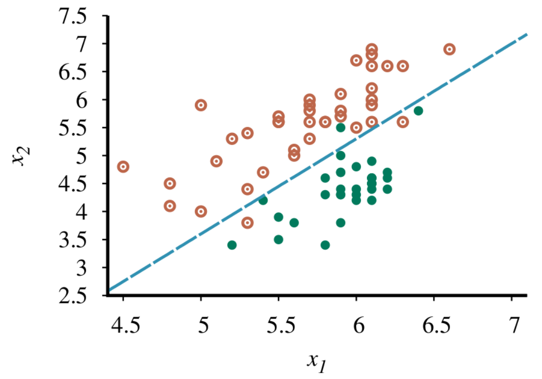
$$w_i \leftarrow w_i + \alpha \times (y - h_w(x)) \times x_i$$

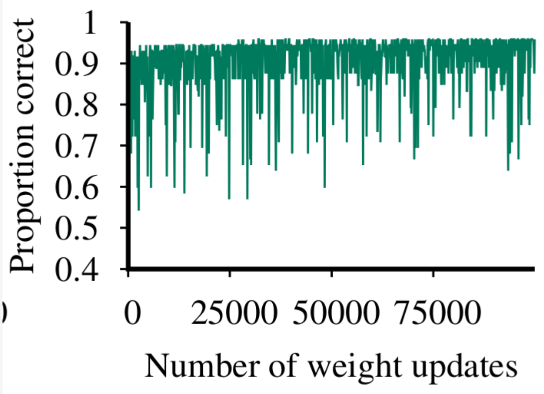
which is identical to the update rule for linear regression (for L_2).

The rule is show to converge to a solution when the data is linearly separable.

The perceptron learning rule (under non separable data)

However the perceptron learning rule is unstable when the data is not linearly separable:





Replacing the step function

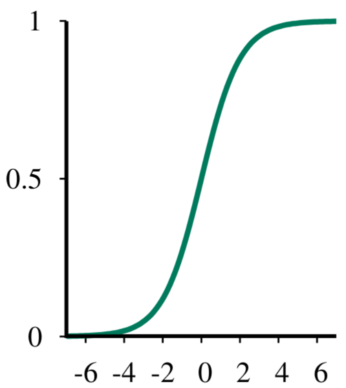
Turns out we can replace the step function with one with nicer properties.

$$Logistic(z) = \frac{1}{1 + e^{-z}}$$

and redefine our hypothesis function:

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

Often called the **logistic regression**.



Back on track

This allows us to reuse gradient descent for training:

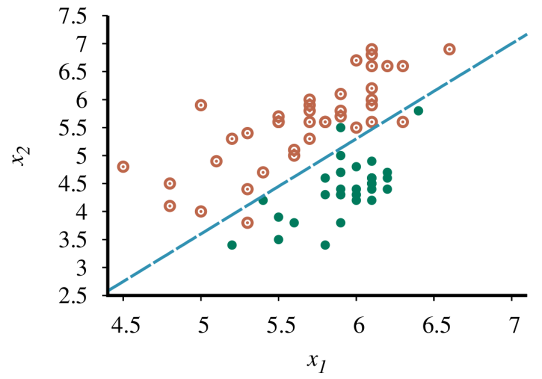
$$w \leftarrow w - \alpha \times \vec{\nabla} Loss(w)$$

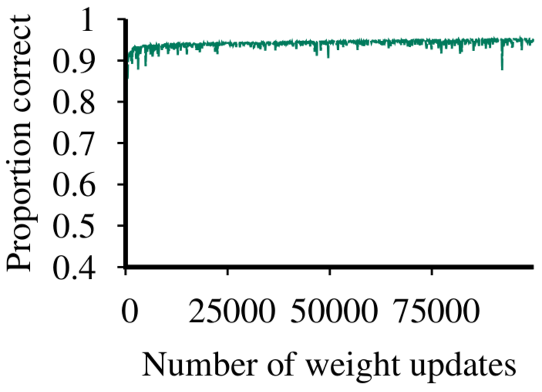
For an L_2 loss we obtain the update rule:

$$w_i \leftarrow w_i + \alpha(y - h_w(x)) \times h_w(x) \times (1 - h_w(x)) \times x_i$$

Training the logistic regression (under non separable data)

The logistic regression tends to converge more quickly an reliable in the presence of noisy and non-separable data.





Section 4

Synthesis

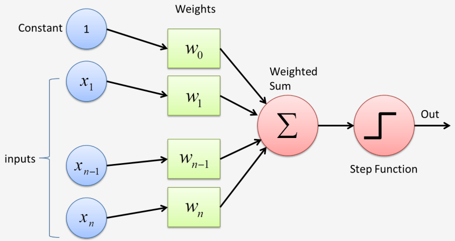
Synthesis

We saw to classes of perceptrons:

- linear regressor
- linear classifier

Both can be trained with **gradient descent** in attempt to **minimize the loss**.

In the next course, the perceptron will be a **neural unit** in a **neural network**.



Exercises

1. I am selling my apartment and I really want it to be sold quickly even if its means loosing some money in the process. Propose a loss function that would help a learning system come up with a reasonable price for selling.
2. What's the update formula of a regression perceptron using the L_1 loss?
3. You have N examples in your dataset, each with M features. Give an estimate of the computational cost of a single update step. Does it scale to large-scale datasets (e.g. $N = 10^5, M = 10^4$)
4. For the linear regression (regression perceptron), are we guaranteed to find the optimal weights with gradient descent?