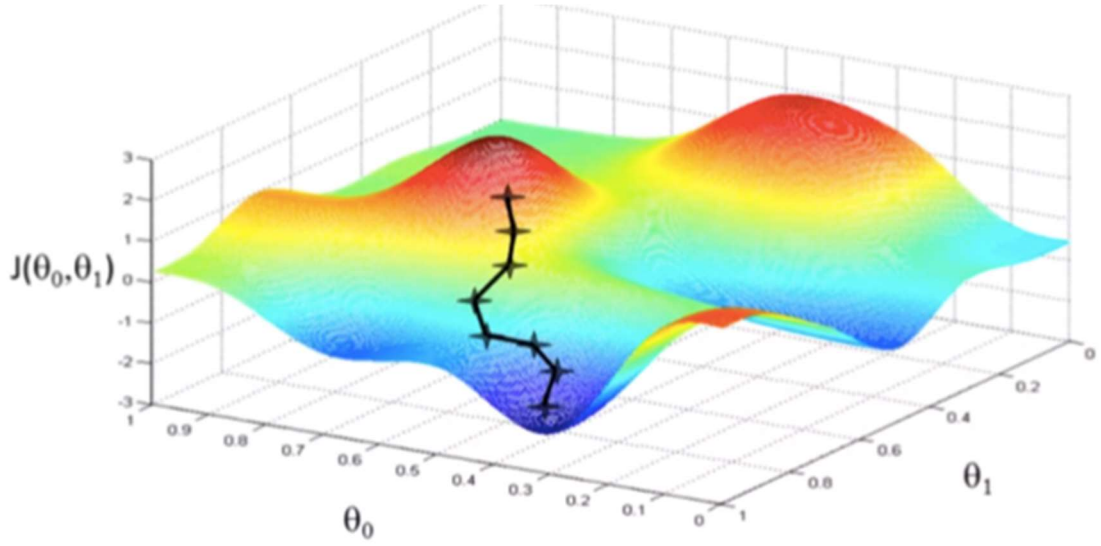
03- PARAMETER LEARNING: GRADIENT DESCENT

Gradient descent

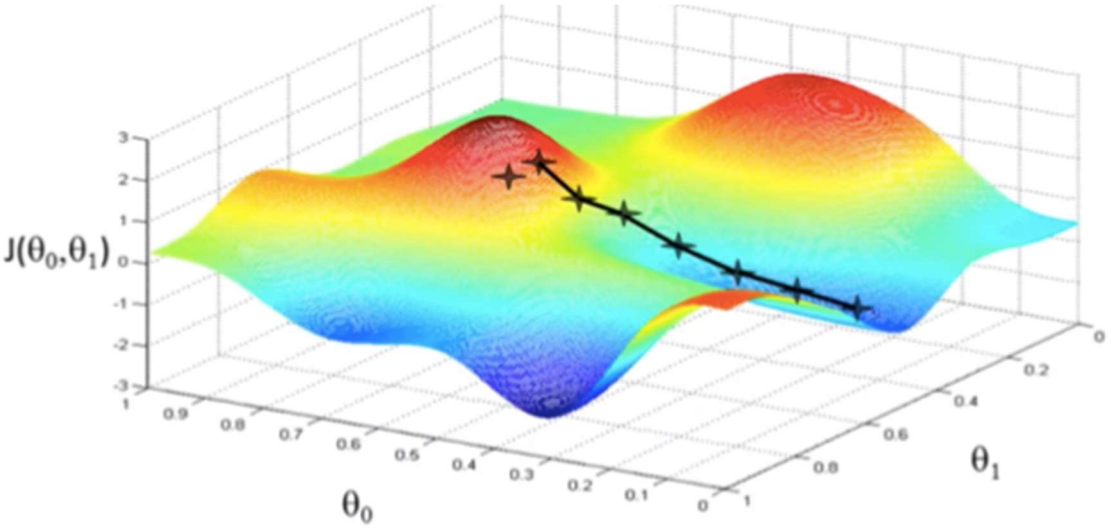
È un algoritmo generale che si applica nel ML. Lo si usa per **minimizzare la funzione di costo**.
Si parte con qualche valore di θ_0, θ_1 che sia zero o dei valori ragionati e si comincia a cambiarli facendo in modo che $J(\theta_0, \theta_1)$ raggiunga un minimo.

Non sappiamo se J abbia un solo minimo o diversi minimi locali.



Per ogni passo vado a scegliere la direzione in cui si ha la pendenza più ripida.

Una caratteristica sgradita del gradient descent è che spostando di poco la configurazione di inizializzazione, la discesa potrebbe andare verso un minimo locale e non in un minimo globale.



Devo dunque calcolare il punto in cui la pendenza è più ripida -> calcolare la derivata.

Formula del gradient descent algorithm:

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0,\theta_1) \quad \text{(for j=0 and j=1)}$$

}

Quindi per determinare il valore del parametro θ_j prendo il valore precedente a cui sottraggo la discesa calcolata in base alla derivata parziale rispetto allo stesso parametro della funzione di costo, moltiplicata per α che è il learning rate. Il learning rate determina la dimensione di ogni step (quanto velocemente scenderò)

Si parla di una simultaneous update in quanto si aggiornano contemporaneamente entrambi i parametri

Simultaneous update implementation

$$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0,\theta_1)$$

$$temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0,\theta_1)$$

$$\theta_0 := temp0$$

$$\theta_1 := temp1$$

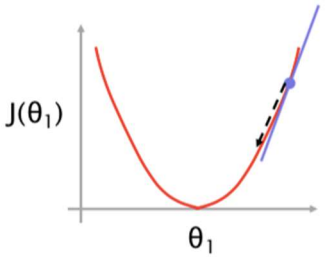
:= è la Simultaneous update implementation dunque aggiornamento simultaneo dei θ .

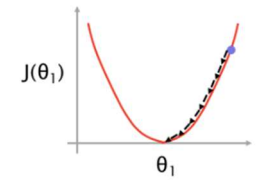
Immaginiamo di avere un caso più semplice con il solo parametro θ_1

repeat until convergence {

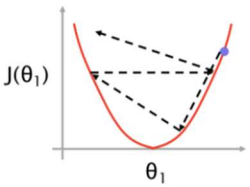
$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

}

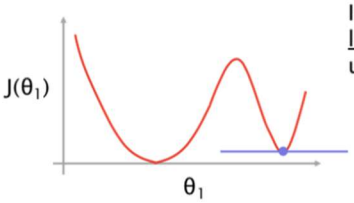




If α is too small, gradient descent can be slow



If α is too large, gradient descent can overshoot the minimum (it may fail to converge, or even diverge)



If θ_1 is already at a local mimimum, no update

Dunque:

- La macchina deve trovare θ_0 e θ_1
- α definito dal programmatore → dunque è un ulteriore iperparametro in quanto anche lui è definito dal programmatore come:
  - modello
  - scelta lineare
  - funzione di costo
  - α
- se la funzione di costo ha più minimi locali, può essere problematico in quanto potrei fermarmi in un minimo locale

L'algoritmo di gradient descent con il **learning rate fisso** va in automatico a modulare il passo facendo passi sempre più piccoli quando va a convergenza perché il valore della derivata diminuisce.

- **Il learning rate non viene cambiato** durante l'addestramento di una rete neurale, il passo diventa più piccolo perché man mano che si va iverso il minimo. diminuisce la derivata

Modello di regressione lineare usato con l'algoritno di gradient descent che abbiamo utilizzato per ottimizzare il problema. → Training linear regression

$$h(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\min_{\theta_0\,\theta_1} J(\theta_0,\theta_1)$$

Linear regression model

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0,\theta_1)$$

} (for j=0 and j=1)

Gradient descent algorithm

Modello della regressione lineare e l'algoritmo di gradient descent

Ora bisogna calcolare la derivata della funzione di costo. La scriviamo per entrambi i parametri

$$\frac{\partial}{\partial \theta_j} J(\theta_0,\theta_1) = \frac{\partial}{\partial \theta_j} \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$= \frac{\partial}{\partial \theta_j} \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

$$j=0: \frac{\partial}{\partial \theta_0} J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$$

$$j=1: \frac{\partial}{\partial \theta_1} J(\theta_0,\theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\, x^{(i)}$$

La differenza sta che per θ_1 c'è anche la derivata composta e quindi una x in più.
L'algoritmo diventa quindi:

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$$

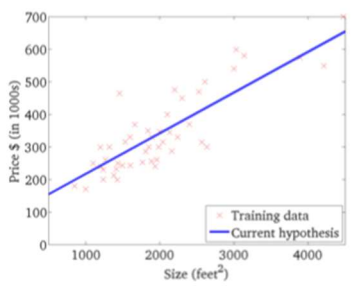
$$\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\, x^{(i)}$$

}

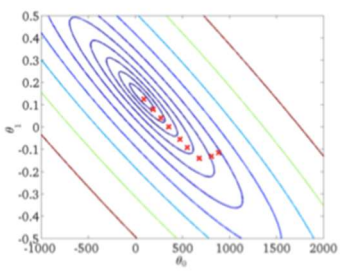
Questo algoritmo viene chiamato **batch gradient descent** perché è un algoritmo per passi e ogni passo usa tutti i training examples → molto lento ma meno costoso

La funzione di costo per la linear regression è caratterizzata per come è stata scritta nell'avere un solo minimo locale ***→ funziona convessa → ci si può avvicinare usando il gradient descent algorithm.***

$h_\theta(x)$
(for fixed θ_1, θ_2, this is a function of x)



$J(\theta_1,\theta_2)$
(function of parameters θ_1,θ_2)



L'obiettivo in generale è quello di scegliere delle funzioni di costo che permettano di avere un unico minimo globale.

Multivariate linear regression

Ci sono più features/caratteristiche.

Notazione:

- m: numero di training examples → numero di righe
- n: numero di features/caratteristiche → numero di colonne
- $x^{(i)}$: input features dell'i-esimo training example → i-esima riga
- $\mathbf{x_j^{(i)}}$**:** valore della feature j dell'i-esimo training example → valore singolo della matrice
- **y:** valore reale da predire

	Size feet² x_1	Number of bedroom x_2	Number of floors x_3	Age of home (years) x_4	Price ($) y
	2104	5	1	45	460
$x^{(2)}$	1416	3	2	40	$y^{(2)}$ 232
	1534	3	2	30	315
	852	2	1	36	178
	..	...	...	...	...

- Hypothesis

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

- Parameters

$$\theta_0, \theta_1, \theta_2, \theta_3, \theta_4$$

- For convenience of notation, an additional variable x_0 is often added to the feature vector, and set to 1 for all the examples in the training set ($x_0^{(i)}=1$)

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

L'ipotesi si può scrivere anche in forma matriciale ottenendo una **visione vettorizzata**.

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix} \quad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

$$h_\theta(x) = \theta^T x$$

$$\theta^T = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \dots & \theta_n \end{bmatrix} \quad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

$$h_\theta(x) = \theta^T x$$

- Cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

- Gradient descent

repeat until convergence {

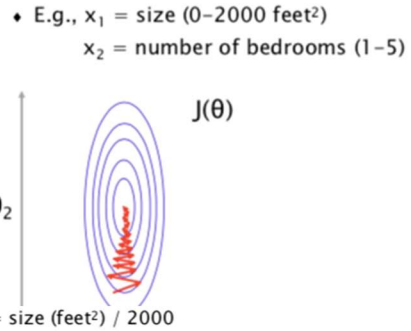
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad \text{(for j=0, ..., n)}$$

→ per ogni θ

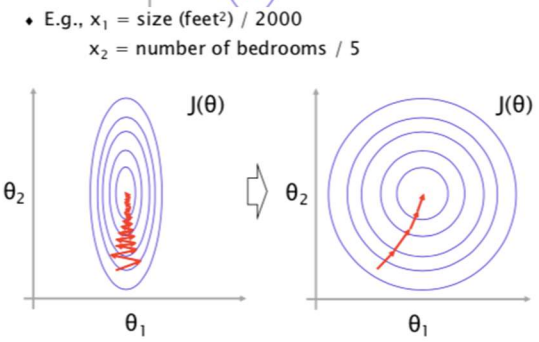
}

Feature scaling

Nel caso in cui le features sono su scale differenti, siccome vengono semplicemente sommate, si rischia di avere dei passi più grandi del dovuto.



L'ambizione è avere ogni feature in un range **[-1, 1]**. Questo si ottiene dividendo la feature per il valore massimo presente all'interno dell'interno training set.



Mean normalization

Si va a sostituire al valore della feature in modo tale da avere un media vicina allo zero:
$(x_i - \mu_i) / s_i$ dove:

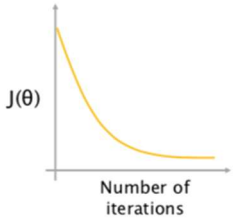
- μ_i è il valore medio di x_i per l'intero training set
- s_i è il range over max-min sempre sull'interno training set

➔ si usa questo valore nella colonna $$x_i := \frac{x_i - \mu_i}{s_i}$$

Debugging gradient descent

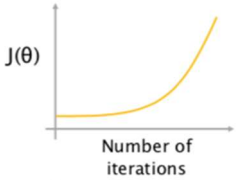
Come si fa a capire che sta funzionando correttamente?

- Disegno grafico

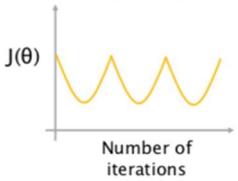


J(θ) should decrease after every iteration

Automatic convergence test: declare convergence of J(θ) decreased by, e.g., less than 10^{-3} in one iteration (arbitrary threshold)



Not working
Use smaller α



Not working
Use smaller α

Dovremmo vedere la funzione di costo scendere ad ogni iterazione,

Summary:

- Se α troppo grande → funzione di costo potrebbe non decresce ad ogni iterazione o non convergere mai
- Se α sufficientemente piccolo → funzione di costo dovrebbe descrescere ad ogni interazione
- Se α troppo piccolo → la funzione di costo potrebbe impiegare troppo tempo per poter convergere
    - Invece di aspettare troppo tempo, conviene provare con dei valori di learning rate su una scala logaritmica finchè non si trova quello corretto

Alternatives to gradient descent

Ci possono essere anche delle alternative al gradient descent, per esempio nel caso specifico della linear regression c'è la tecnica della normal equation o del sistema di equazioni normali che da in un colpo la soluzione:

$$\theta = (X^T X)^{-1} X^T y$$

Bisogna scrivere la X come un vettore di esempi

$$X = \begin{bmatrix} \text{---} & x^{(1)} & \text{---} \\ \text{---} & x^{(2)} & \text{---} \\ & \vdots & \\ \text{---} & x^{(m)} & \text{---} \end{bmatrix}$$

Il problema è che questo tipo di tecnica non funziona per qualsiasi problema, mentre il gradient descent funziona per tutti.

Gradient descent	Normal equation
Need to choose alfa Needs many iterations	No need to choose alfa Don't need to iterate No need for feature scaling
Work well even when n is large	Need to compute $(X^TX)^{-1}$ Slow if n is large

La normal equation può essere efficace con poche feature.

Optimization algorithms

Per poter ottimizzare la funzione di costo, mi serve la stessa e le derivate parziali.

- Given θ, we have code that can compute

  $J(\theta)$

  $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, 2, \dots, n$)

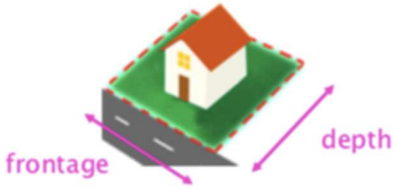
- Optimization algorithms
  - Gradient descent
  - Conjugate gradient
  - BFGS
  - L-BFGS
  - ...

  Advantages:
  No need to manually pick alfa
  Often faster than gradient descent

  Disadvantages
  More complex
  But we do not need to implement them

Polynomial regression

Immaginiamo di voler predire il costo delle case avendo queste informazioni



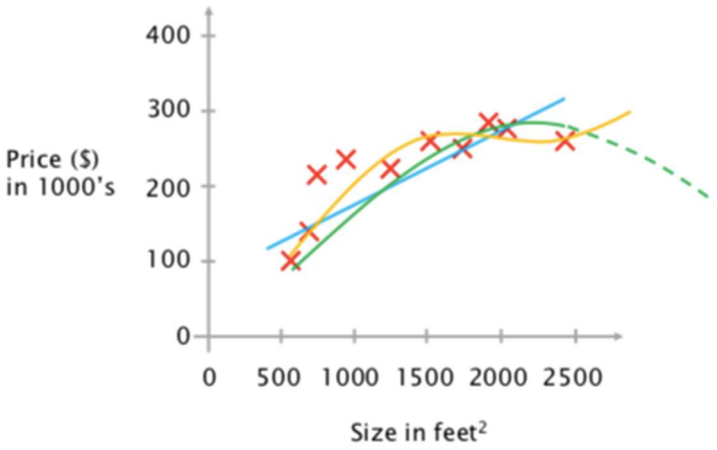
$$h_\theta(x) = \theta_0 + \theta_1 \times \underbrace{\text{frontage}}_{x_1} + \theta_2 \times \underbrace{\text{depth}}_{x_2}$$

$$\text{area} = \text{frontage} \times \text{depth} = x$$

$$h_\theta(x) = \theta_0 + \theta_1 \times x$$

Per tutte le features del mio nuovo dataset mi creo una **nuova feature che è data dalla relazione lineare delle features originali** ottenendo un parametro quadratico. L'algoritmo non si accorge che sia diventato quadratico e quindi continua ad affrontare in maniera lineare ma con delle features quadratiche.
In questo modo si possono gestire problemi complessi.

Ad esempio per la distribuzione vista del dataset potrebbe aver senso avere non una retta ma forme polinomiale quadratiche, cubiche, etc



Si possono quindi creare delle **hand-crafted features** per modellare al meglio il problema.

- Third order polynomial function

$$h_\theta(x) = \theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 + \theta_3 \times x_3$$
$$= \theta_0 + \theta_1 \times (\text{size}) + \theta_2 \times (\text{size})^2 + \theta_3 \times (\text{size})^3$$

$x_1 = (\text{size})$
$x_2 = (\text{size})^2$
$x_3 = (\text{size})^3$
} Newly designed features

In situazioni come queste diventa fondamentale il **feature scaling**;
La creazione di **hand-crafted features** ci serve per introdurre delle conoscenze del dominio.

Le reti neurali profonde sono brave ad imparare da sole le features rilevanti, successivamente noi addestriamo le reti neurali fondendo degli esempi e le addestriamo insegnandole come trovare i valori dei parametri ϑ .