

Departamento de Eletrónica, Telecomunicações e  
Informática

## **LECTURE 2: LINEAR REGRESSION**

**Petia Georgieva**  
**([petia@ua.pt](mailto:petia@ua.pt))**

# LINEAR REGRESSION - outline

## 1. Univariate linear regression

- Cost (loss) function - Mean Squared Error (MSE)
- Cost function convergence
- Gradient descent algorithm

## 2. Multivariate linear regression

- Overfitting problem

## 3. Regularization => way to deal with overfitting

# Supervised Learning - CLASSIFICATION vs REGRESSION

**Classification**- the label is an integer number.  
(e.g. 0, 1 for binary classification)

**Regression** - the label is a real number.

## **Examples of regression problems:**

- Weather forecast
- Predicting wind velocity from temperature, humidity, air pressure
- Time series prediction of stock market indices
- Predicting sales amounts of new product based on advertising expenditure

# Standard Notations in this course

$x$  – input vector of features, attributes

$y$  – output vector of labels, ground truth, target

$m$  - number of training examples

$n$  – number of features

$h_{\theta}(x)$  - model (hypothesis)

$\theta$  - vector of model parameters

Training set: data matrix  $X$  ( $m$  rows,  $n$  columns)

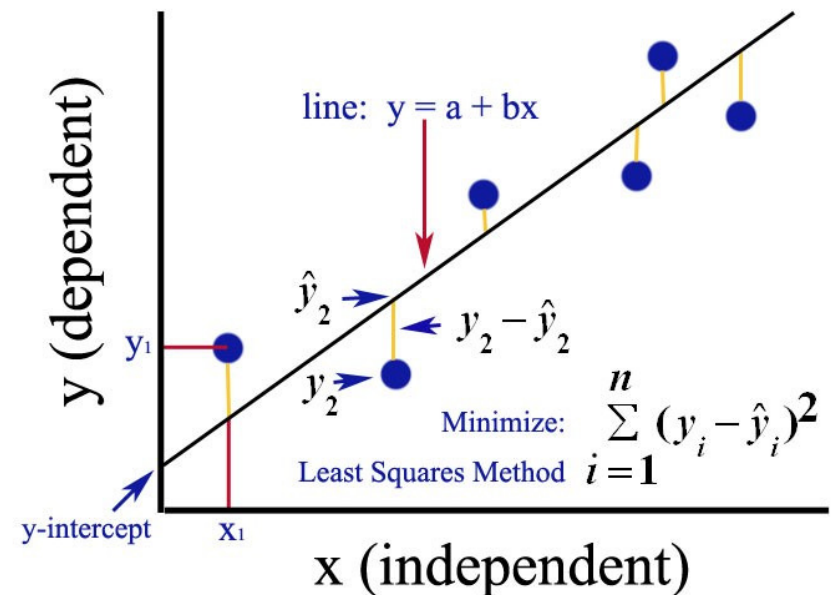
	feature $x_1$	feature $x_2$	.....	feature $x_n$	output(label) $y$
Example 1	$x_1^{(1)}$			$x_n^{(1)}$	$y^{(1)}$
Example 2	$x_1^{(2)}$			$x_n^{(2)}$	$y^{(2)}$
...					
Example $i$	$x_1^{(i)}$			$x_n^{(i)}$	$y^{(i)}$
...					
...					
Example $m$	$x_1^{(m)}$			$x_n^{(m)}$	$y^{(m)}$

# Supervised Learning – univariate regression

**Problem:** Learning to predict the housing prices (output, predicted variable) as a function of the living area (input, feature, predictor)

**observation – prediction = Error (“residual”)**

Living area (feet <sup>2</sup> )	Price (1000\$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮



# Supervised Learning – univariate regression

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 = \begin{bmatrix} 1 & x_1 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \vec{x}^T \vec{\theta}$$

=> in Python => np.dot(X, Theta)

$$\vec{x} = \begin{bmatrix} x_0 = 1 \\ x_1 \end{bmatrix}$$

	$X_0$ (extra column)	feature $x_1$ (living area)	output(label) $y$ (price)
Example 1=>	1	$x^{(1)}$	$y^{(1)}$
Example 2=>	1	$x^{(2)}$	$y^{(2)}$
	1		
Example m=>	1	$x^{(m)}$	$y^{(m)}$

# Mean Square Error (MSE)

**Linear Model (hypothesis)** =>

$$h_{\theta}(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

**Cost (loss) function** =>  
(Mean Square Error)

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

**m** – number of training examples

Goal =>

$$\min_{\theta} J(\theta)$$

**Gradient descent algorithm** =>  
iterative algorithm; at each  
iteration all parameters (theta)  
are updated simultaneously

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

**alpha** – learning rate > 0

# Linear Regression

## (computing the gradient)

**Cost function** =>

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

**Cost function gradients** =>

vector with partial derivatives of  $J$  with respect to each parameter for one example ( $m=1$ )

$$\begin{aligned} \frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_{\theta}(x) - y)^2 \\ &= 2 \cdot \frac{1}{2} (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_{\theta}(x) - y) \\ &= (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left( \sum_{i=0}^n \theta_i x_i - y \right) \\ &= (h_{\theta}(x) - y) x_j \end{aligned}$$

$$h_{\theta}(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

**Cost function gradients** =>

for  $m$  examples

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



# Linear Regression – iterative gradient descent algorithm (summary)

**Initialize model parameters** (e.g.  $\theta = 0$ )

**Repeat until J converge {**

**Compute Linear Regression Model =>**

$$h_{\theta}(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

**Compute cost function =>**

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

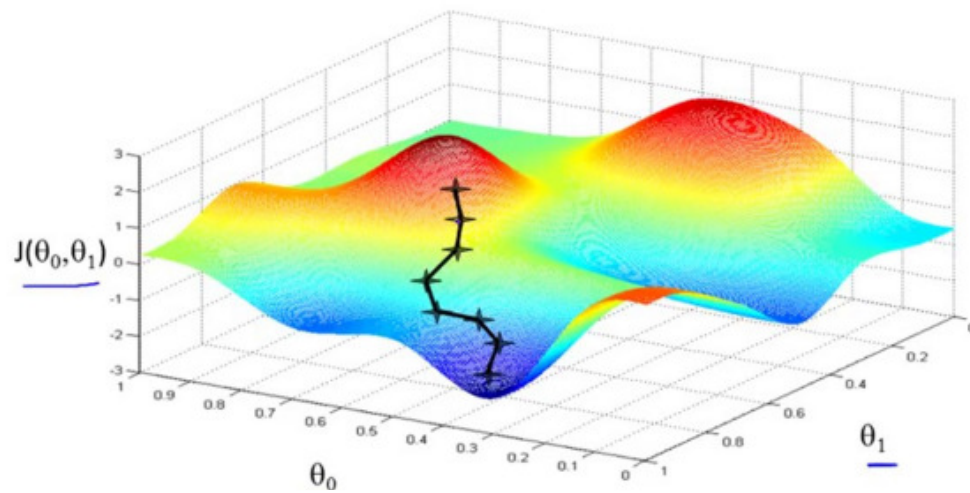
**Compute cost function gradients =>**

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

**Update parameters =>**

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}



# Batch/mini batch/stochastic gradient descent for parameter update

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

**Batch learning** (classical approach):

update parameters after all training examples have been processed, repeat several iterations until convergence

**Mini batch learning** (if big training data):

divide training data into small batches update parameters after each mini batch has been processed, repeat until convergence

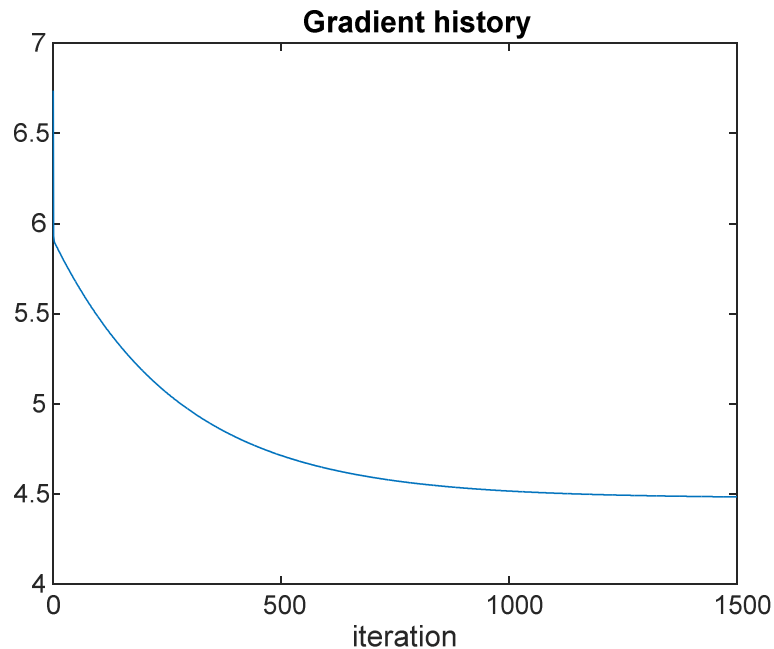
**Stochastic (incremental) learning** (large-scale ML problems):

update parameters after every single training example has been processed.

**Stochastic Gradient Descent (SGD):** the true gradient is approximated by a gradient at a single example. Adaptive learning rate.

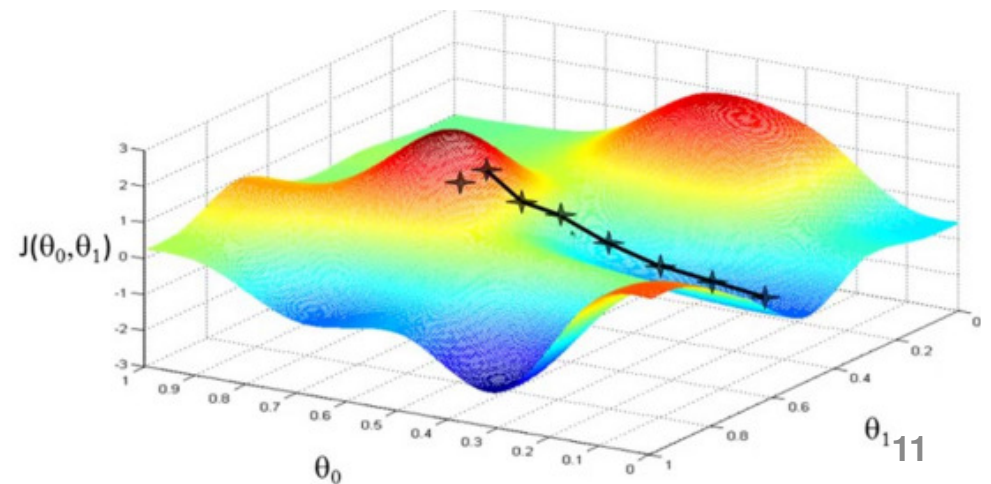
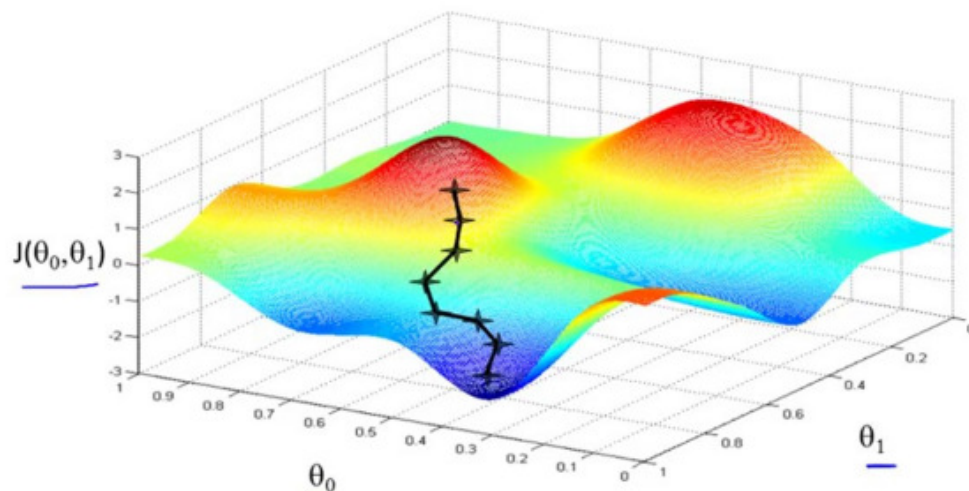
# Cost function convergence

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



## Linear Regression (LR):

starting from different initial values of the parameters the cost function  $J$  should always converge (**maybe to a local minimum !!!**) if LR works properly.

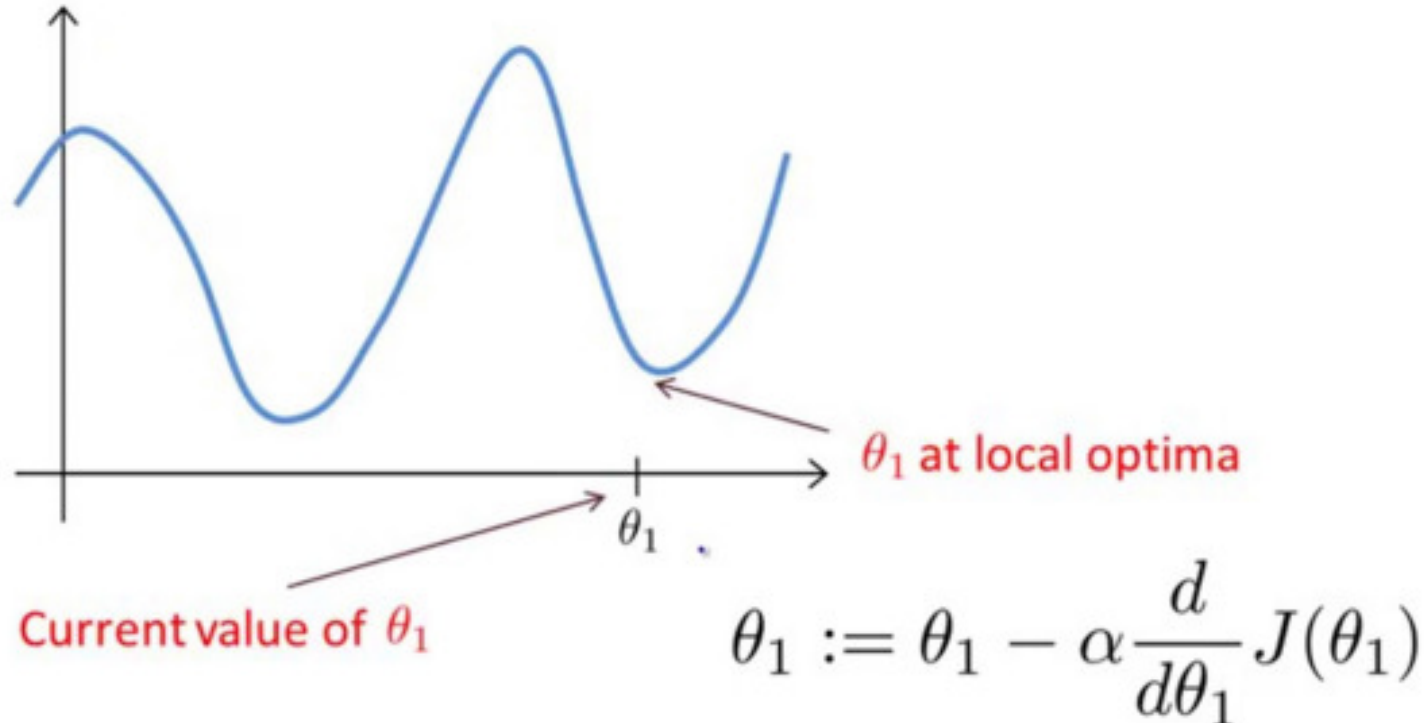


# Lin Reg Cost function – local minimum

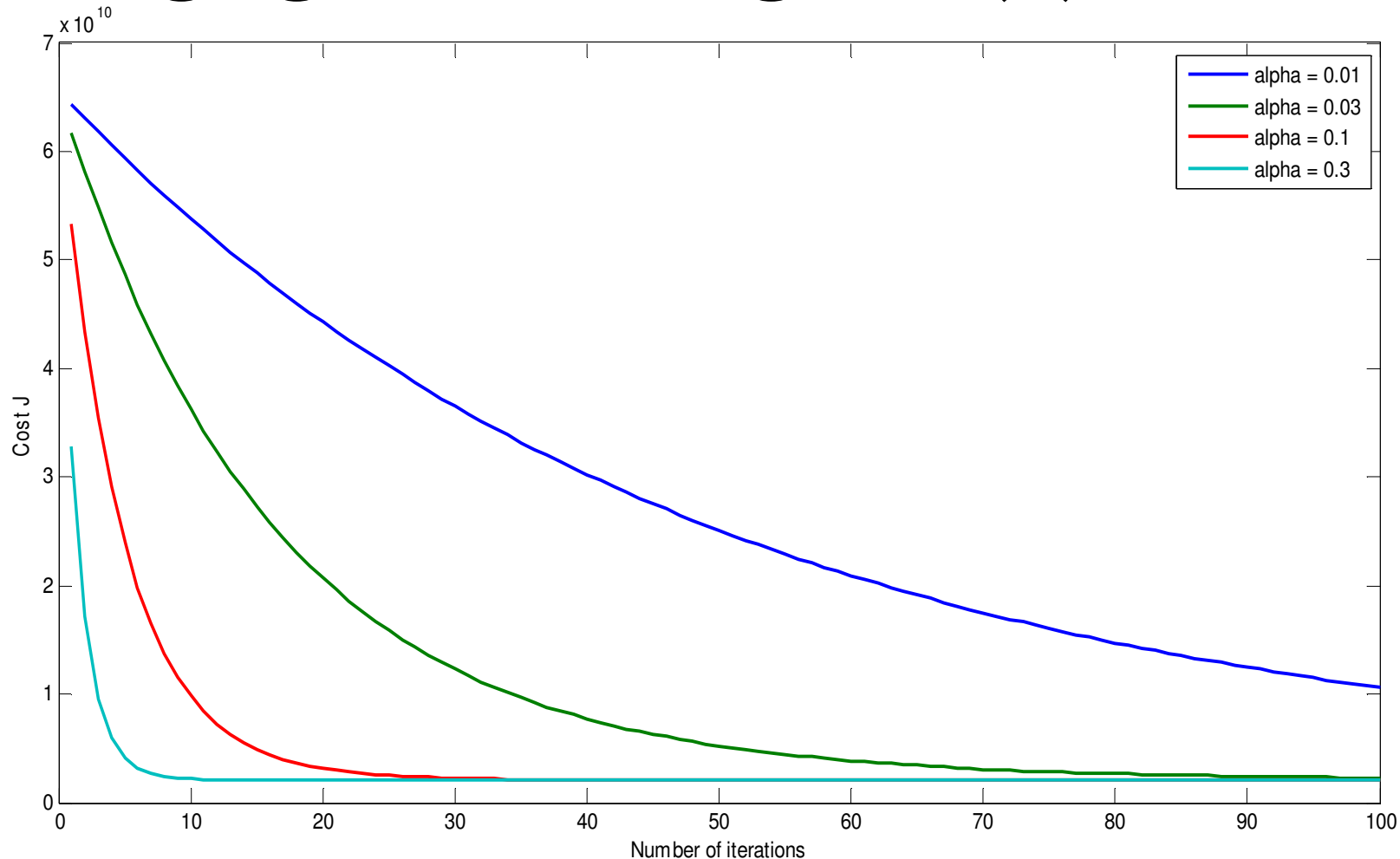
Suppose  $\theta_1$  is at a local optima as shown in the figure.

**What will one step of Gradient Descent do ?**

- 1) Leave  $\theta_1$  unchanged
- 2) Change  $\theta_1$  in a random direction
- 3) Decrease  $\theta_1$
- 4) Move  $\theta_1$  in direction to the global minimum of J



# Cost function convergence changing the learning rate ( $\alpha$ ) - 100 iter.

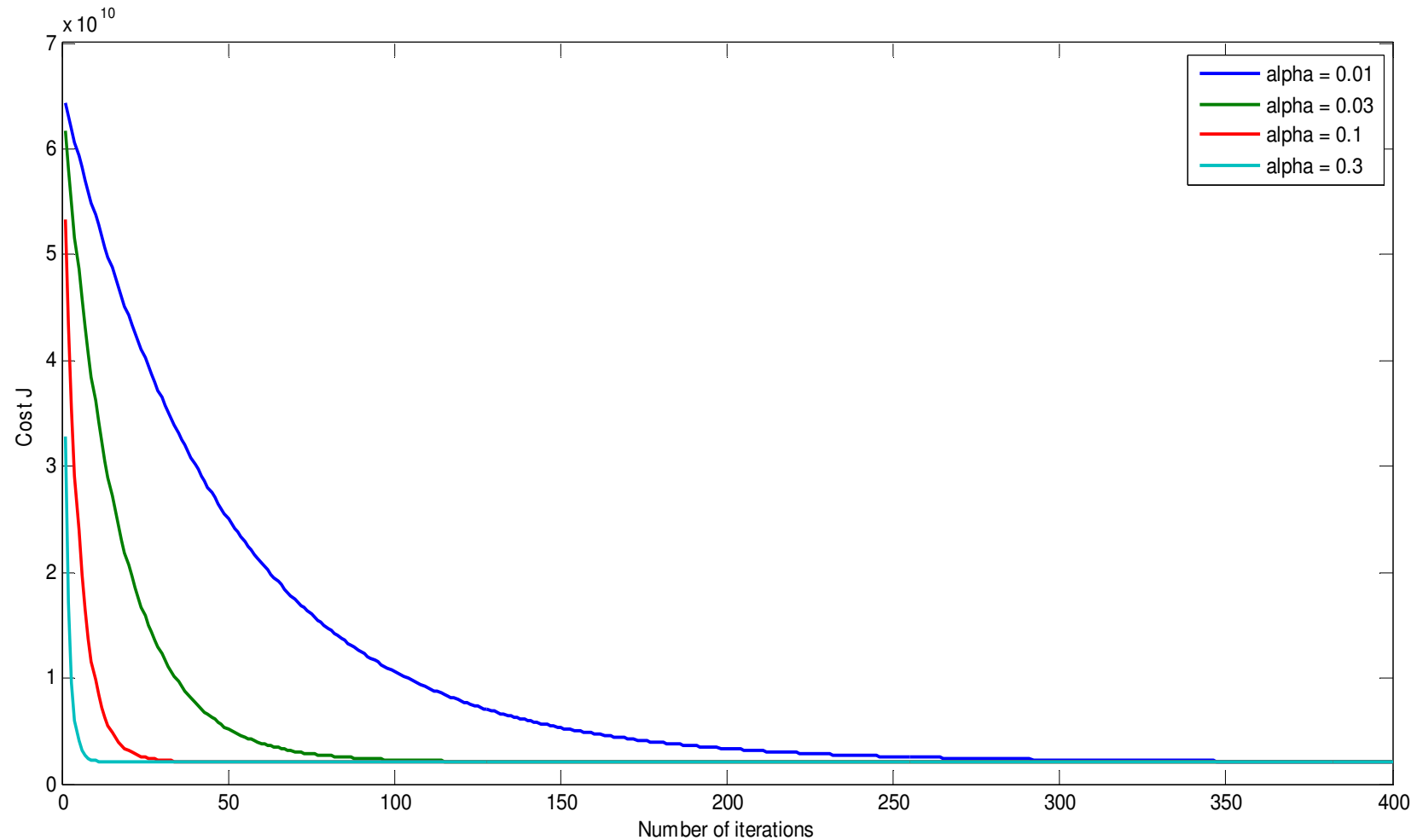


$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

**If  $\alpha$  too small :** slow convergence of the cost function  $J$  (the Gradient Descent optimization can be slow)

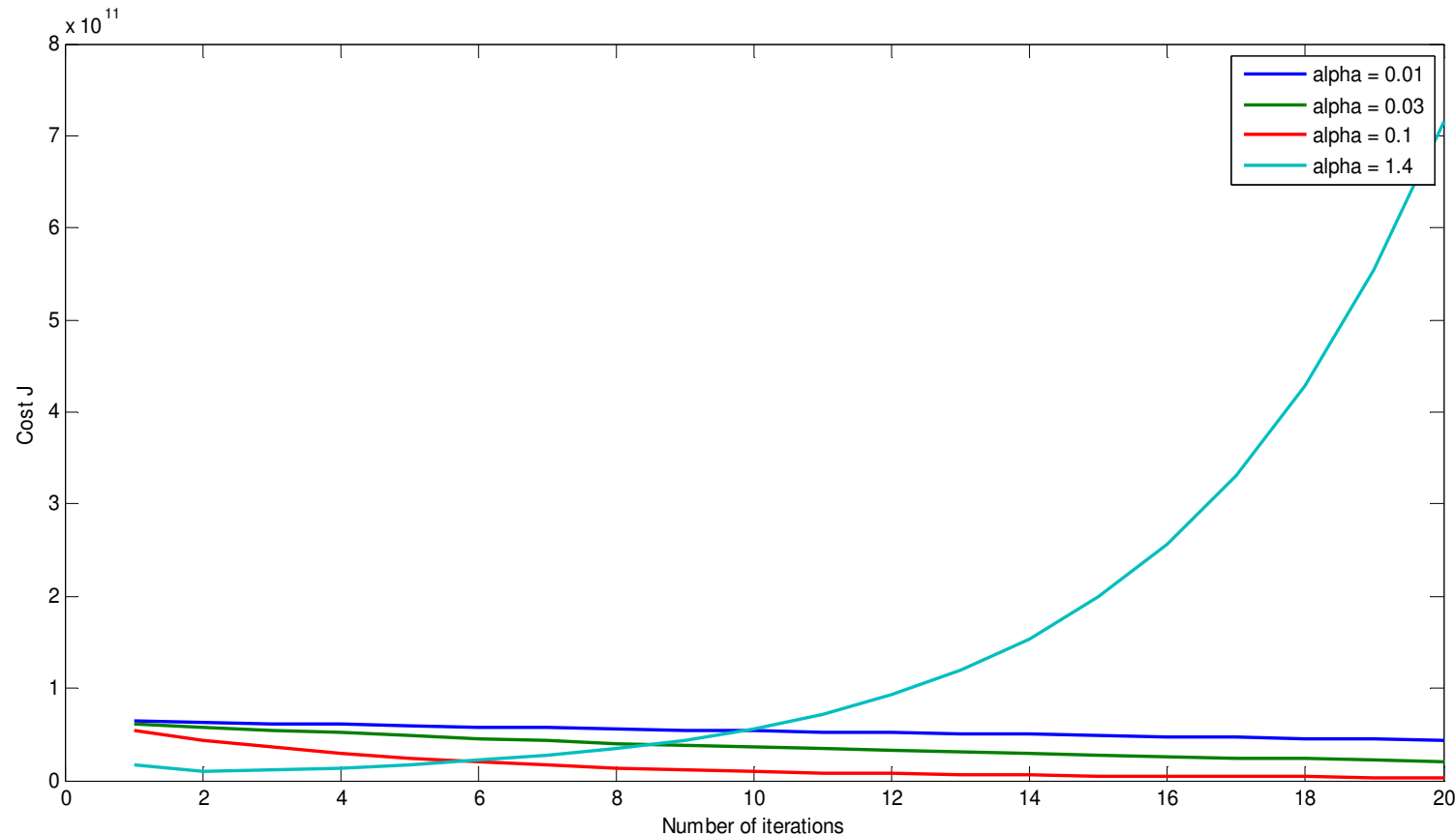
# Cost function convergence

## changing the learning rate ( $\alpha$ ) -400 iter.



$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

# LinReg Cost function convergence - learning rate variation ( $\alpha$ )

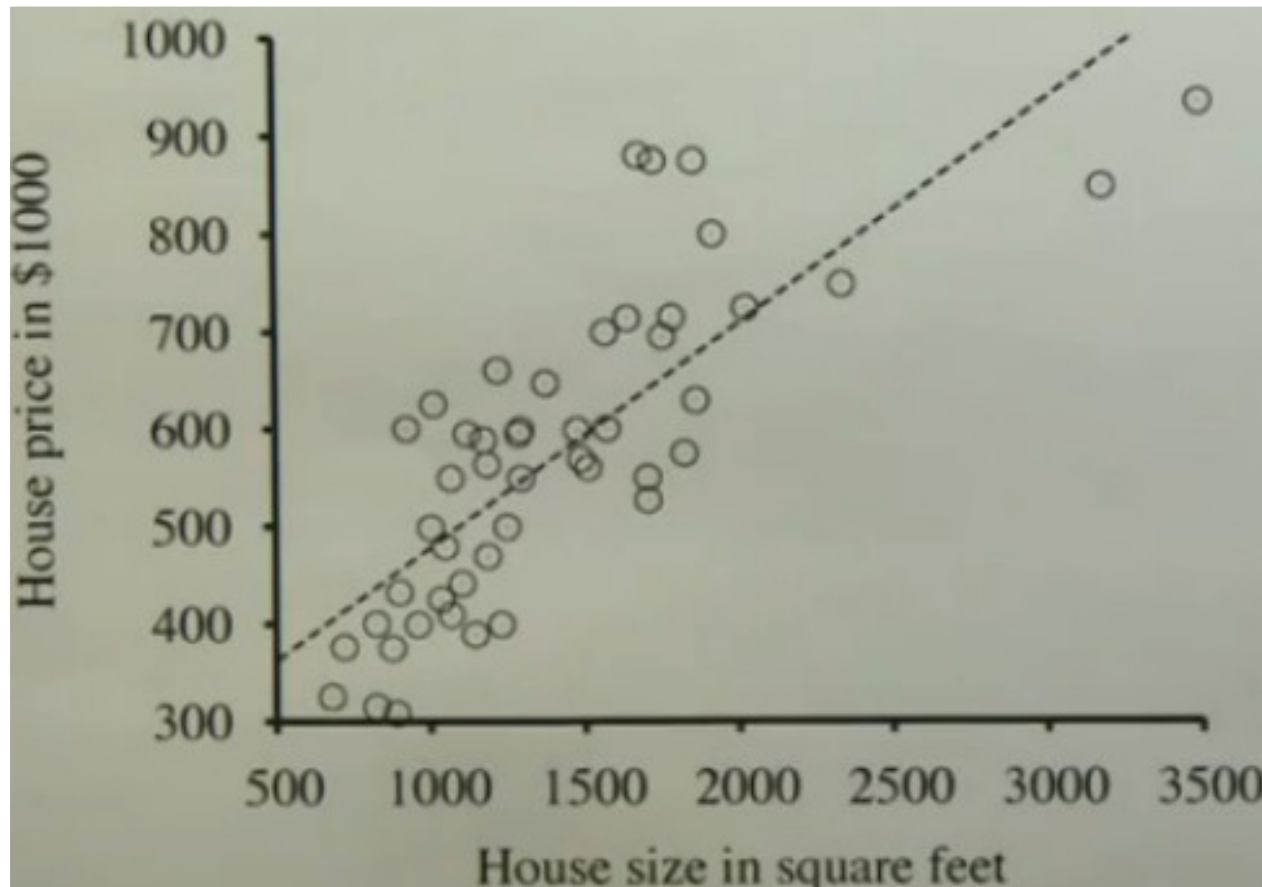


**If  $\alpha$  too large:** the cost function  $J$  may no converge (decrease at each iteration). It may diverge !

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

# Univariate Regression

$$h_{\theta}(x) = \theta^T x = \theta_0 + \theta_1 x_1$$



Given the house area, what is the most likely house price?  
If univariate linear regression model is not sufficiently good model,  
add more data (ex. # bedrooms).



# Multivariate Regression

**Problem: Learning to predict the housing price as a function of living area & number of bedrooms.**

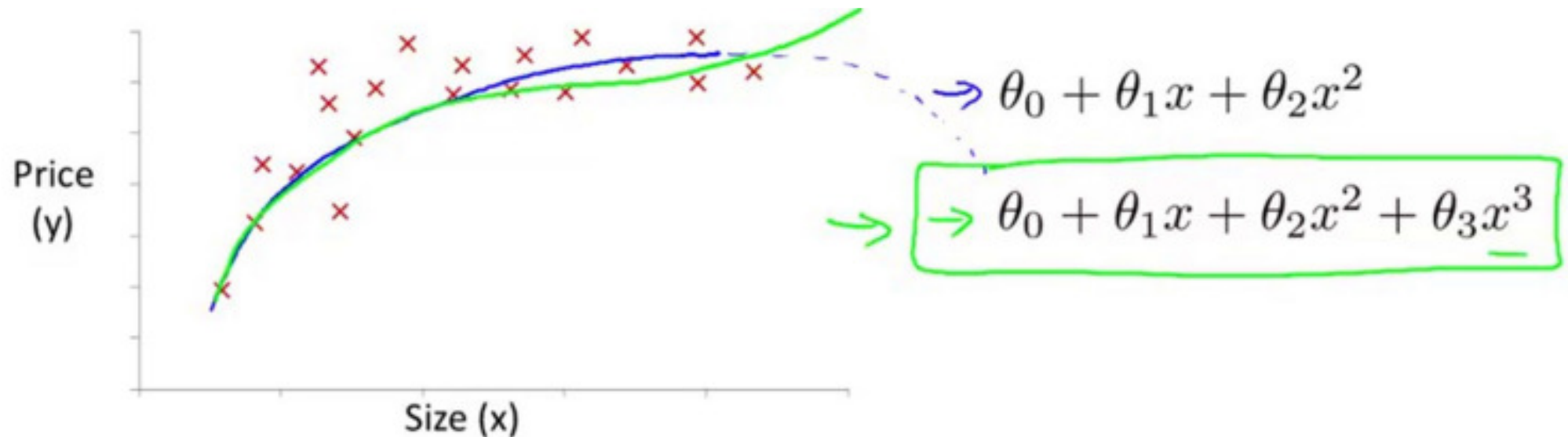
Living area (feet <sup>2</sup> )	#bedrooms	Price (1000\$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
$\vdots$	$\vdots$	$\vdots$

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = [\theta_0 \quad \theta_1 \quad \theta_2] \begin{bmatrix} x_0 = 1 \\ x_1 \\ x_2 \end{bmatrix} = \vec{\theta}^T \vec{x}$$

# Polynomial Regression

If univariate linear regression model is not a good model, try polynomial model.

Univariate ( $x_1 = \text{size}$ ) housing price problem transformed into multivariate (still linear !!!) regression model  $x = [x_1 = \text{size}, x_2 = \text{size}^2, x_3 = \text{size}^3]$



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

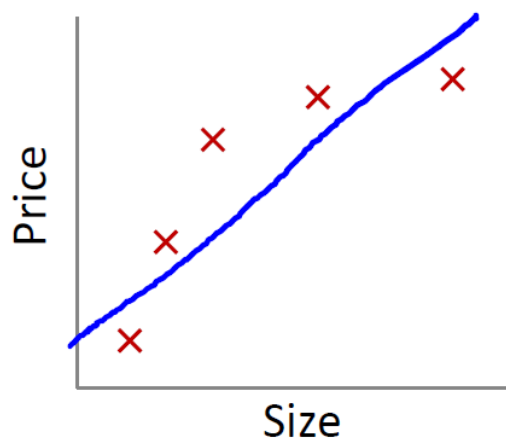
$$x_1 = (\text{size})$$

$$x_2 = (\text{size})^2$$

$$x_3 = (\text{size})^3$$

# Overfitting problem

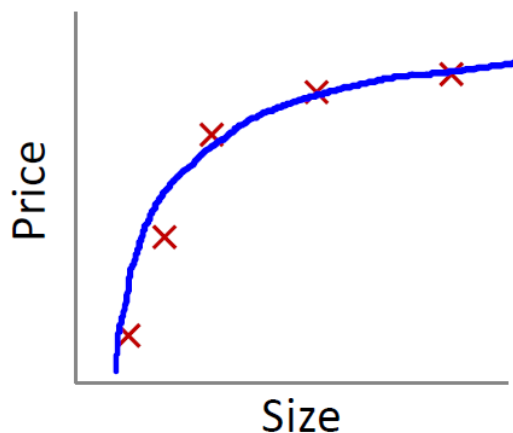
Overfitting: If we have too many features ( e.g. high order polynomial model), the learned hypothesis may fit the training set very well but fail to generalize to new examples (predict prices on new examples).



**underfit**

(1st order polin. model)

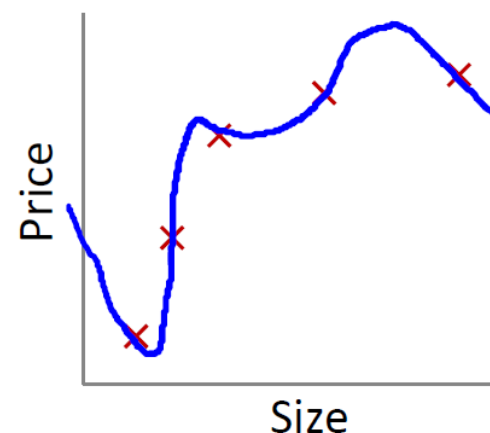
$$h_{\theta}(x) = \theta_0 + \theta_1 x$$



**just right**

(3rd order polinom. model)

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$



**overfit**

(higher ord. polinom. Model)

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \dots + \theta_{16} x^n$$

# Overfitting problem

Overfitting: If we have too many features ( $x_1, \dots, x_{100}$ ) the learned model may fit the training data very well but fails to generalize to new examples.

$x_1$  = size of house

$x_2$  = no. of bedrooms

$x_3$  = no. of floors

$x_4$  = age of house

$x_5$  = average income in neighborhood

$x_6$  = kitchen size

$\vdots$

$x_{100}$

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n = \vec{\theta}^T \vec{x}$$

# How to deal with overfitting problem ?

## 1. Reduce number of features.

- Manually select which features to keep.
- Algorithm to select the best model complexity.

## 2. Regularization (add extra term in cost function)

Regularization methods shrink model parameters  $\theta$  towards zero to prevent overfitting by reducing the variance of the model.

### 2.1 Ridge Regression (L2 norm)

- Reduce magnitude of  $\theta$  (but never make them =0)  $\Rightarrow$  keep all features
- Works well when all features contributes a bit to the output  $y$ .

### 2.2 Lasso Regression (L1 norm)

- May shrink some of the elements of vector  $\theta$  to become 0.
- Eliminate some of the features  $\Rightarrow$  Serve as feature selection

# Regularized Linear Regression (cost function)

Unregularized cost function =>

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

**Regularized cost function**  
(add extra regularization term  
don't regularize  $\theta_0$ )

Ridge Regression

$$J(\theta) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$$\min_{\theta} J(\theta)$$

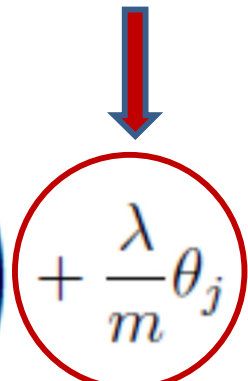
# Regularized Linear Regression (cost function gradient)

**Unregularized cost  
function gradients =>**

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

**Regularized cost  
function gradients =>**

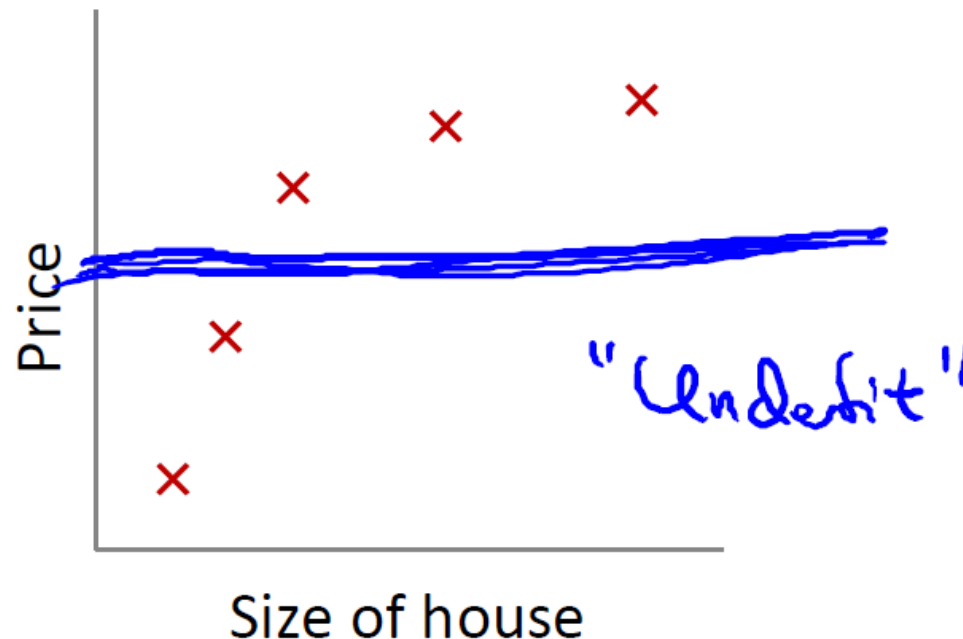
$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{for } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left( \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \quad \text{for } j \geq 1$$


# Regularized Linear Regression

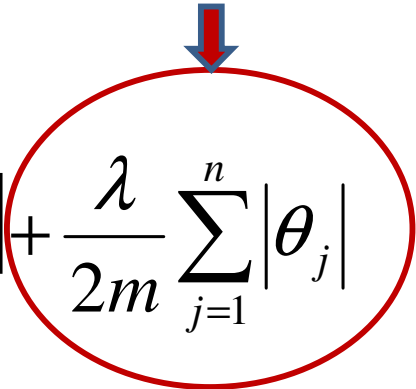
What if lambda is set to an extremely large value ?

- Algorithm fails to eliminate overfitting.
- Algorithm results in under-fitting. (Fails to fit even training data well).
- Gradient descent will fail to converge.





# Regularization: Lasso Regression

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \left[ -y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n |\theta_j|$$


Ridge Regression shrinks  $\theta$  towards zero, but never equal to zero => all features are included in the model no matter how small are the coefficients.

Lasso Regression is able to shrink coefficients to exactly zero => reduces the number of features. This makes Lasso Regression useful in cases with high dimension.

Lasso Regression involves absolute values (not differentiable) => computing is difficult => relevant algorithms available in sklearn Python library.