Linear Regression

Problem Setup

Training set of housing prices (Portland, OR)

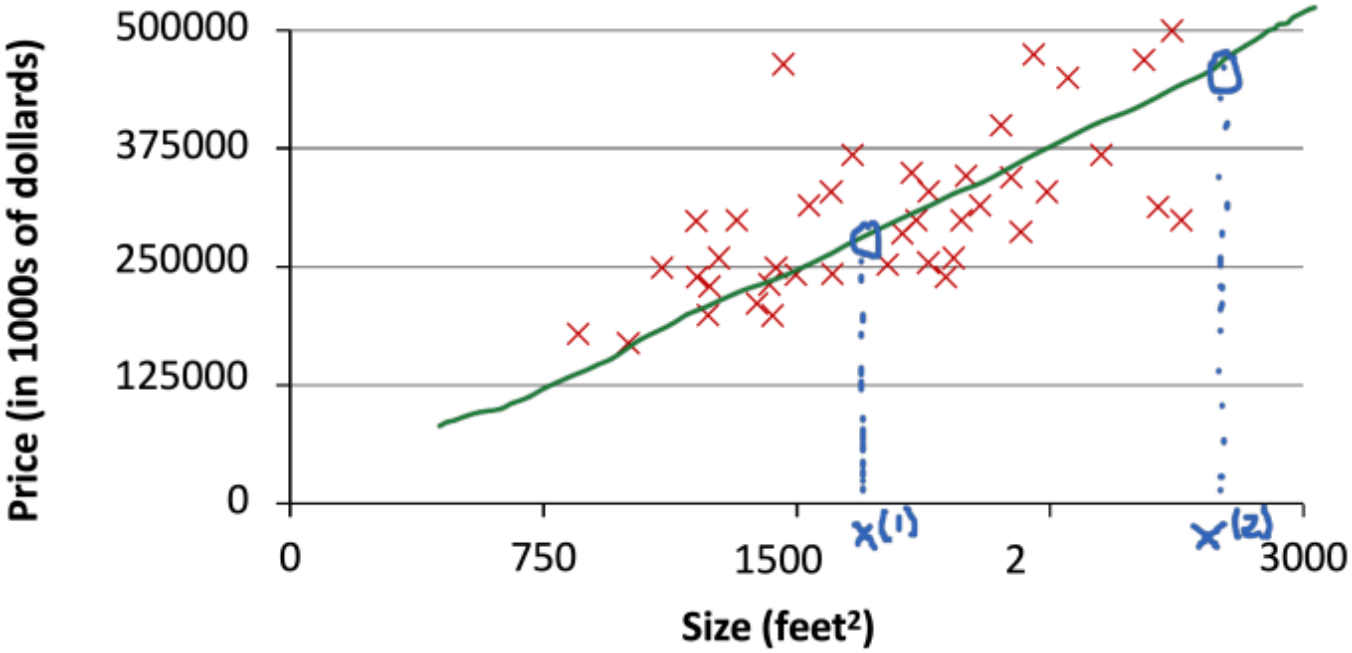
Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
…	…

Notation:

- **m** = Number of training examples
- **x**'s = "input" variable / features
- **y**'s = "output" variable / "target" variable

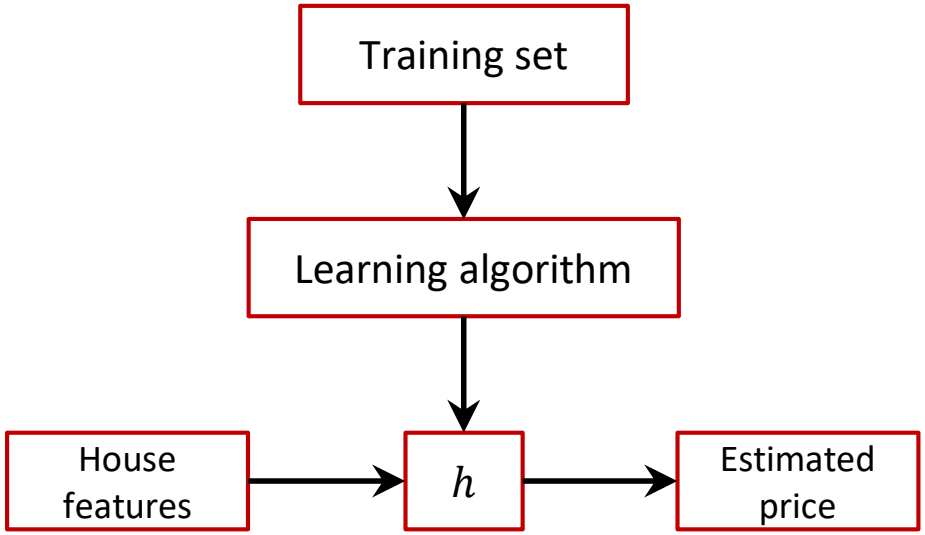
Problem Setup

Housing Prices (Portland, OR)



Supervised Learning: Given the "right answer" for each example in the data.
Regression problem: Predict real-valued output

Problem Setup



How do we represent h?

- We represent hypothesis about the data using parameters $\theta = (\theta_0, \theta_1)$.
- If the data is correctly predicted according to the hypothesis h_θ, then $y \approx h_\theta(x) = \theta_0 + \theta_1 x$
- The learning algorithm finds the best hypothesis h_θ for the training set.
- We can then estimate the values of y for the test set using that h_θ
- If $h_\theta(x)$ is a linear function of a real number x of , this procedure is called linear regression.

Problem Setup

Training Set

Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
…	…

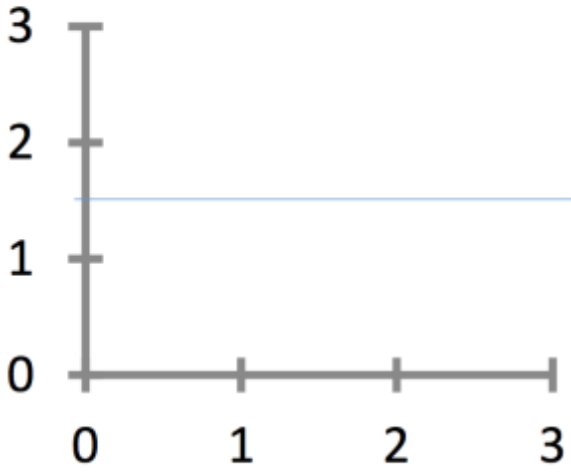
Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

where θ_0, θ_1 are learnable parameters.

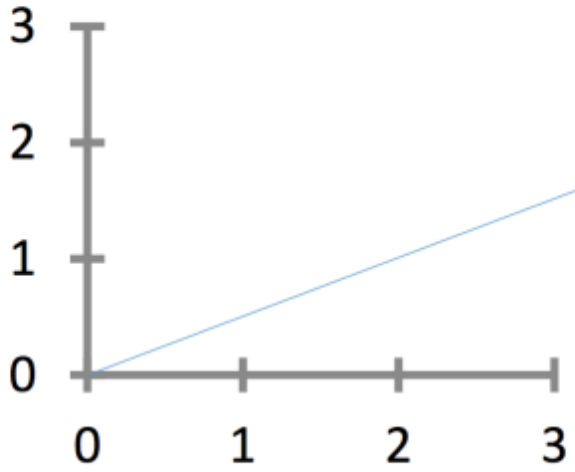
Question: how to choose

Observation

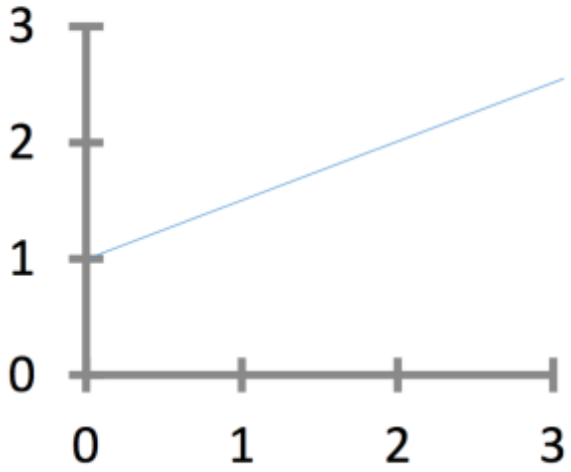
$$h_\theta(x) = \theta_0 + \theta_1 x$$



$\theta_0 = 1.5$
$\theta_1 = 0$

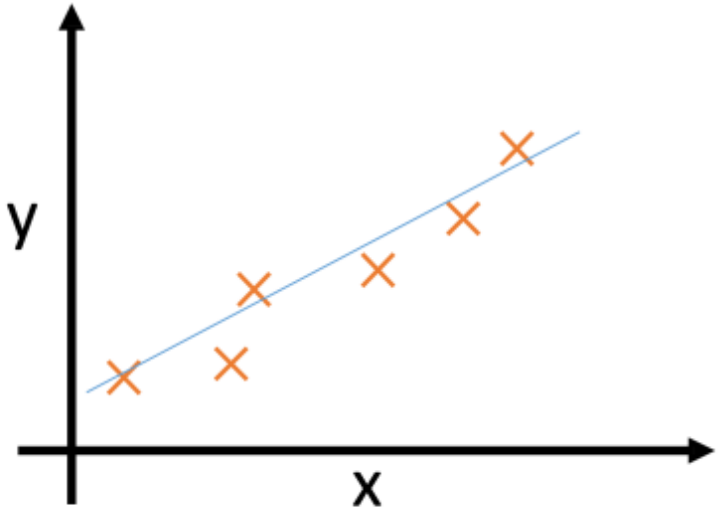


$\theta_0 = 0$
$\theta_1 = 0.5$



$\theta_0 = 1$
$\theta_1 = 0.5$

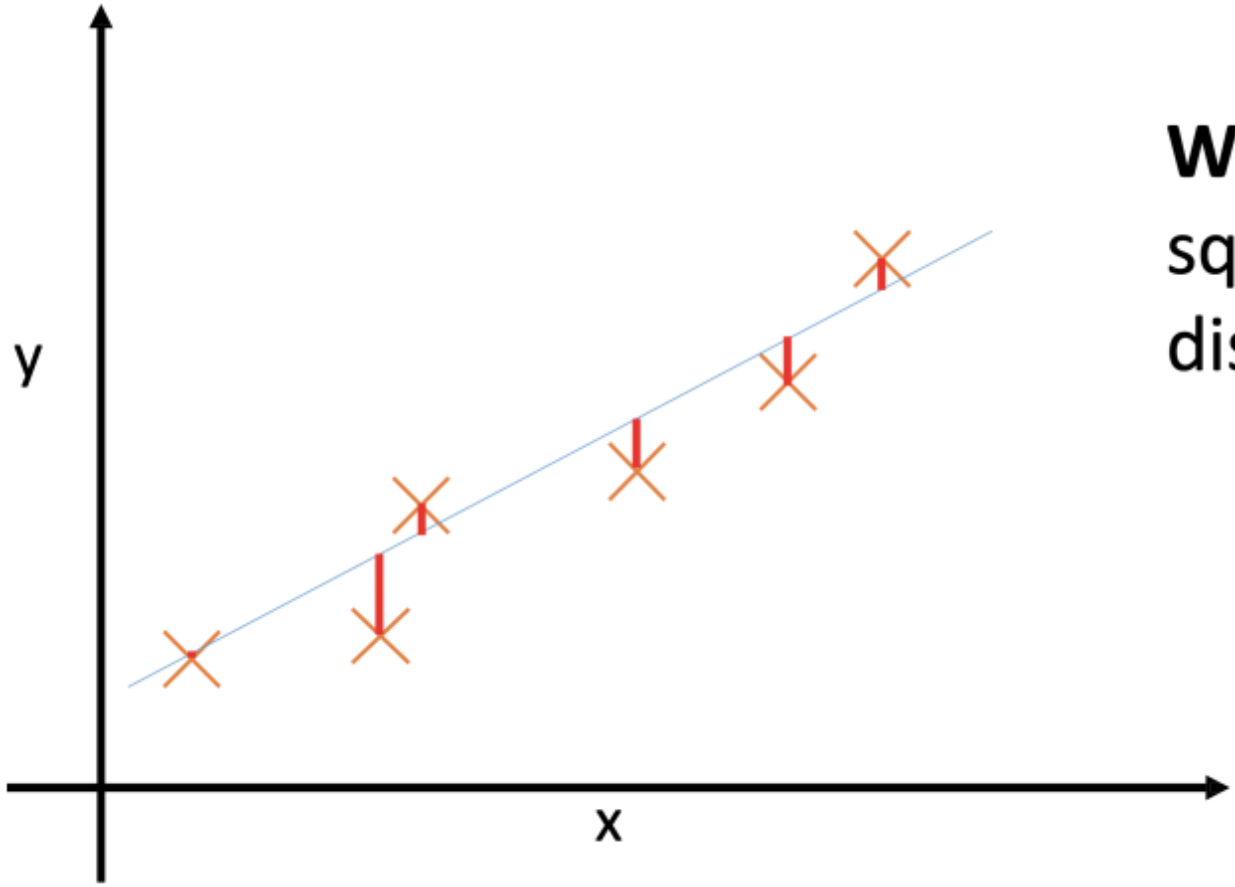
Observation



But what does "close" mean?

Idea: Choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y)

Observation



We choose:
squared vertical distance

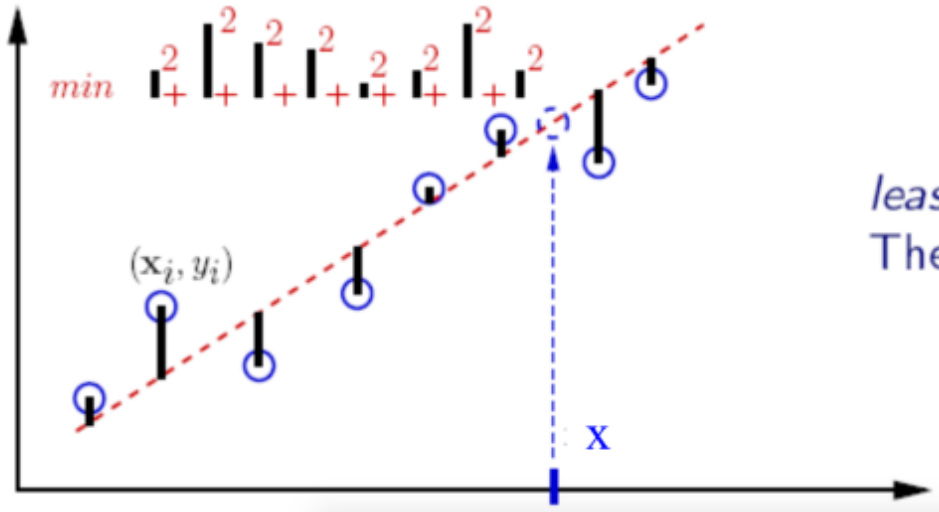
Linear Regression

- Hypothesis:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d = \sum_{j=0}^{d} \theta_j x_j$$

Assume $x_0 = 1$

- Fit model by minimizing sum of squared errors



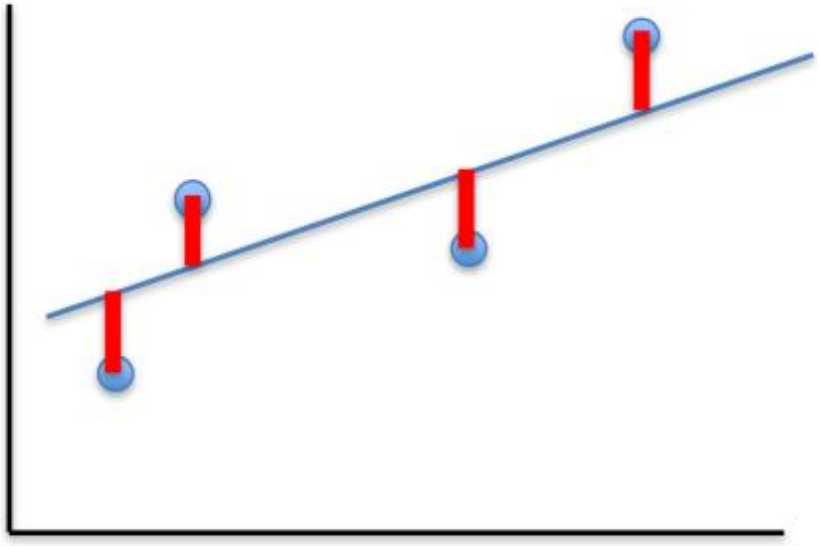
least squares (LSQ)
The fitted line is used as a predictor

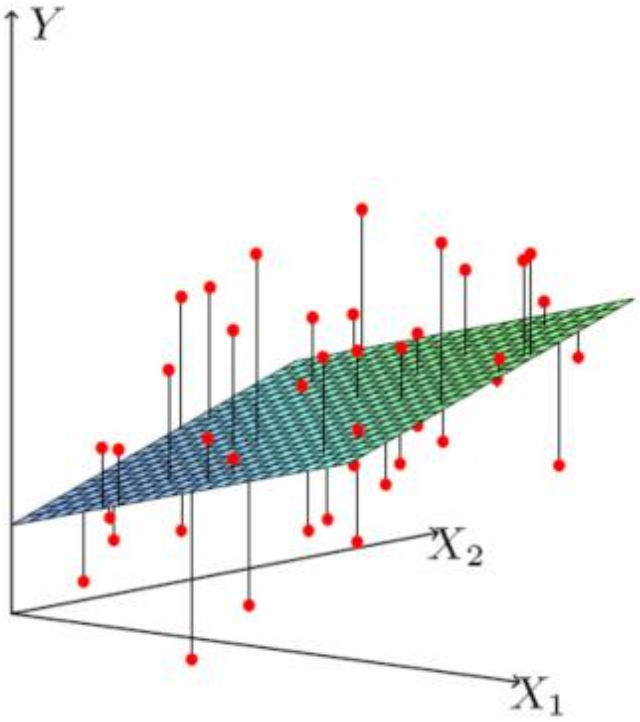
Least Squares Linear Regression

- Cost Function

$$J(\boldsymbol{\theta}) = \frac{1}{2n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right)^2$$

- Fit by solving $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$





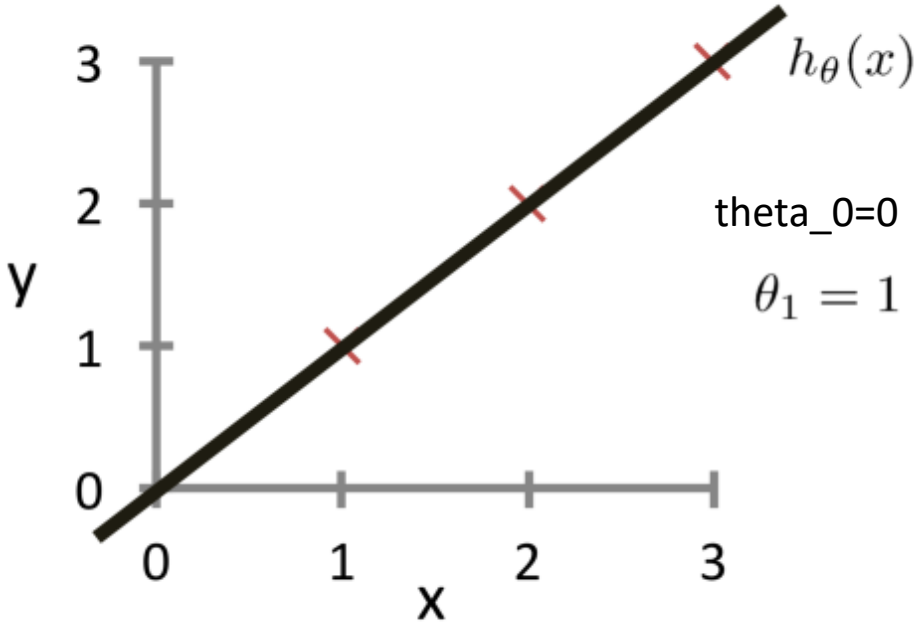
Intuition Behind Cost Function

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

For insight on J(), let's assume $x \in \mathbb{R}$ so $\boldsymbol{\theta} = [\theta_0, \theta_1]$

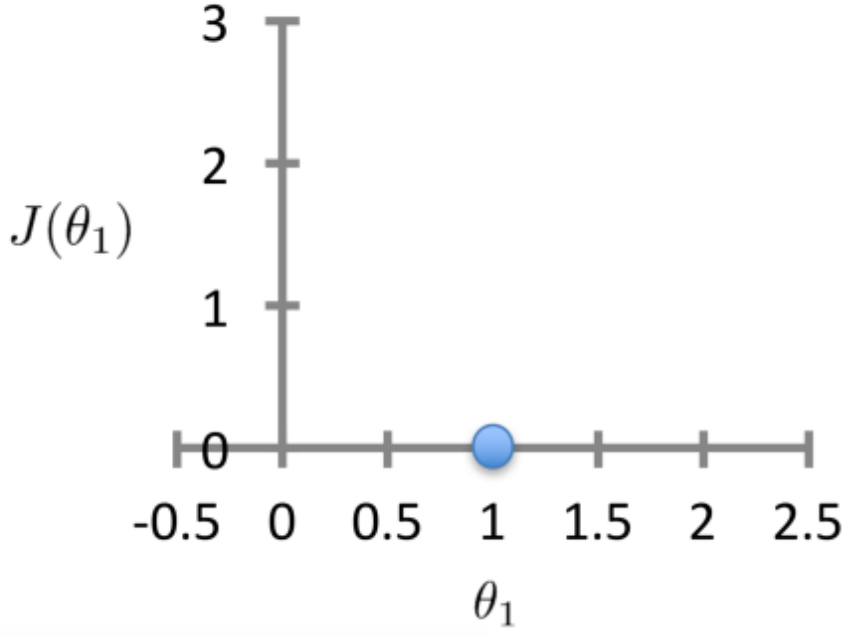
$h_\theta(x)$

(for fixed θ_1, this is a function of x)



$J(\theta_1)$

(function of the parameter θ_1)



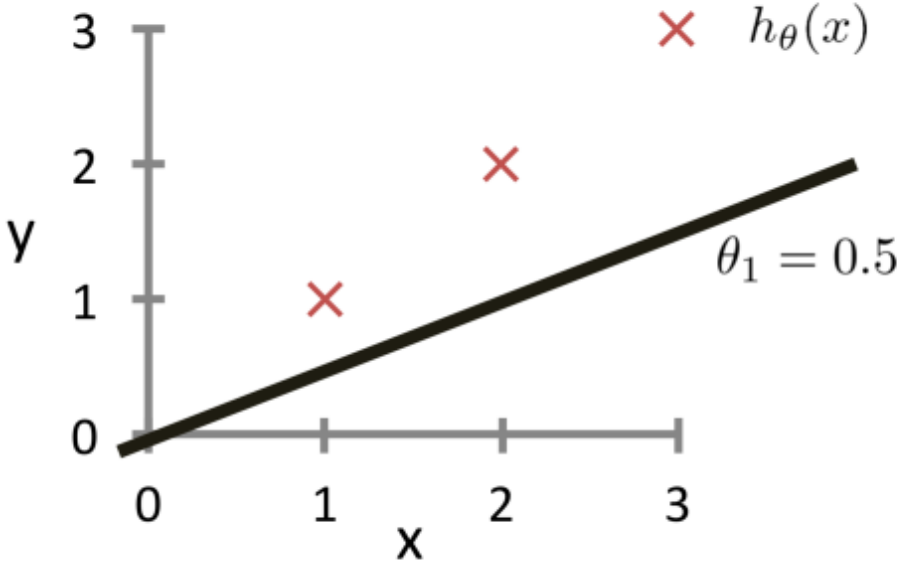
Intuition Behind Cost Function

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

For insight on J(), let's assume $x \in \mathbb{R}$ so $\boldsymbol{\theta} = [\theta_0, \theta_1]$

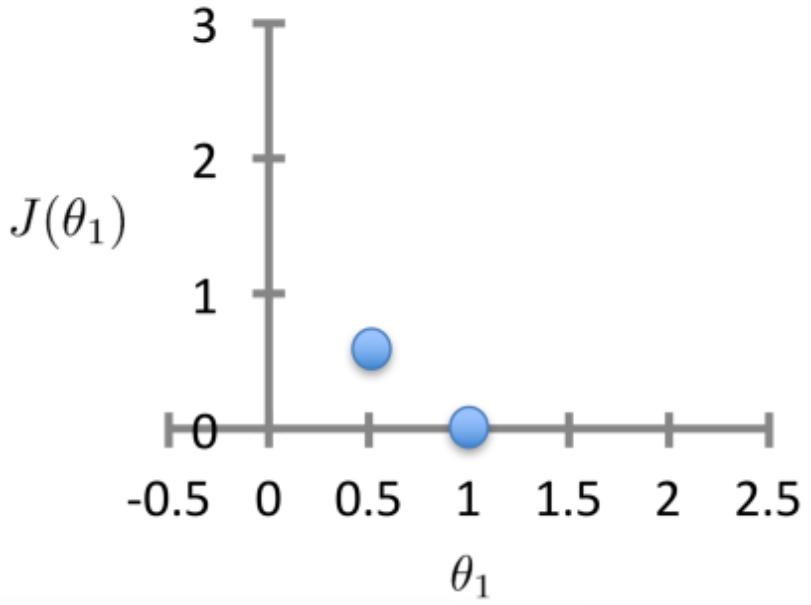
$h_\theta(x)$

(for fixed θ_1, this is a function of x)



$J(\theta_1)$

(function of the parameter θ_1)



$$J([0, 0.5]) = \frac{1}{2 \times 3}\left[(0.5-1)^2 + (1-2)^2 + (1.5-3)^2\right] \approx 0.58$$

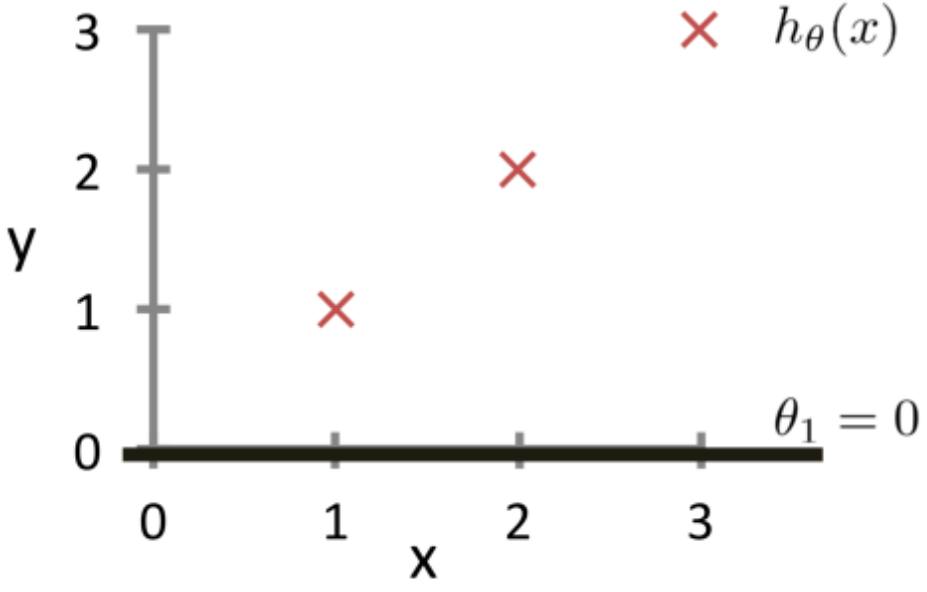
Intuition Behind Cost Function

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

For insight on J(), let's assume $x \in \mathbb{R}$ so $\boldsymbol{\theta} = [\theta_0, \theta_1]$

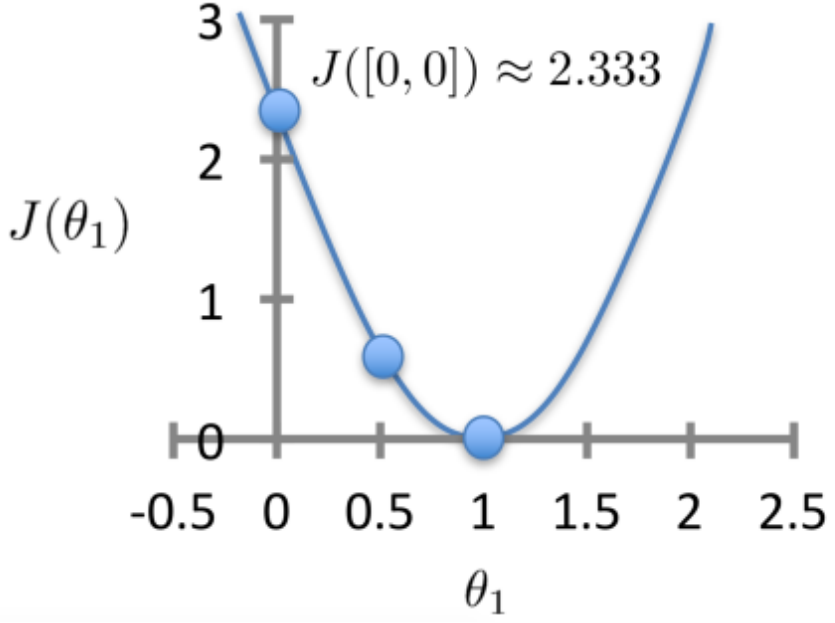
$h_\theta(x)$

(for fixed θ_1, this is a function of x)

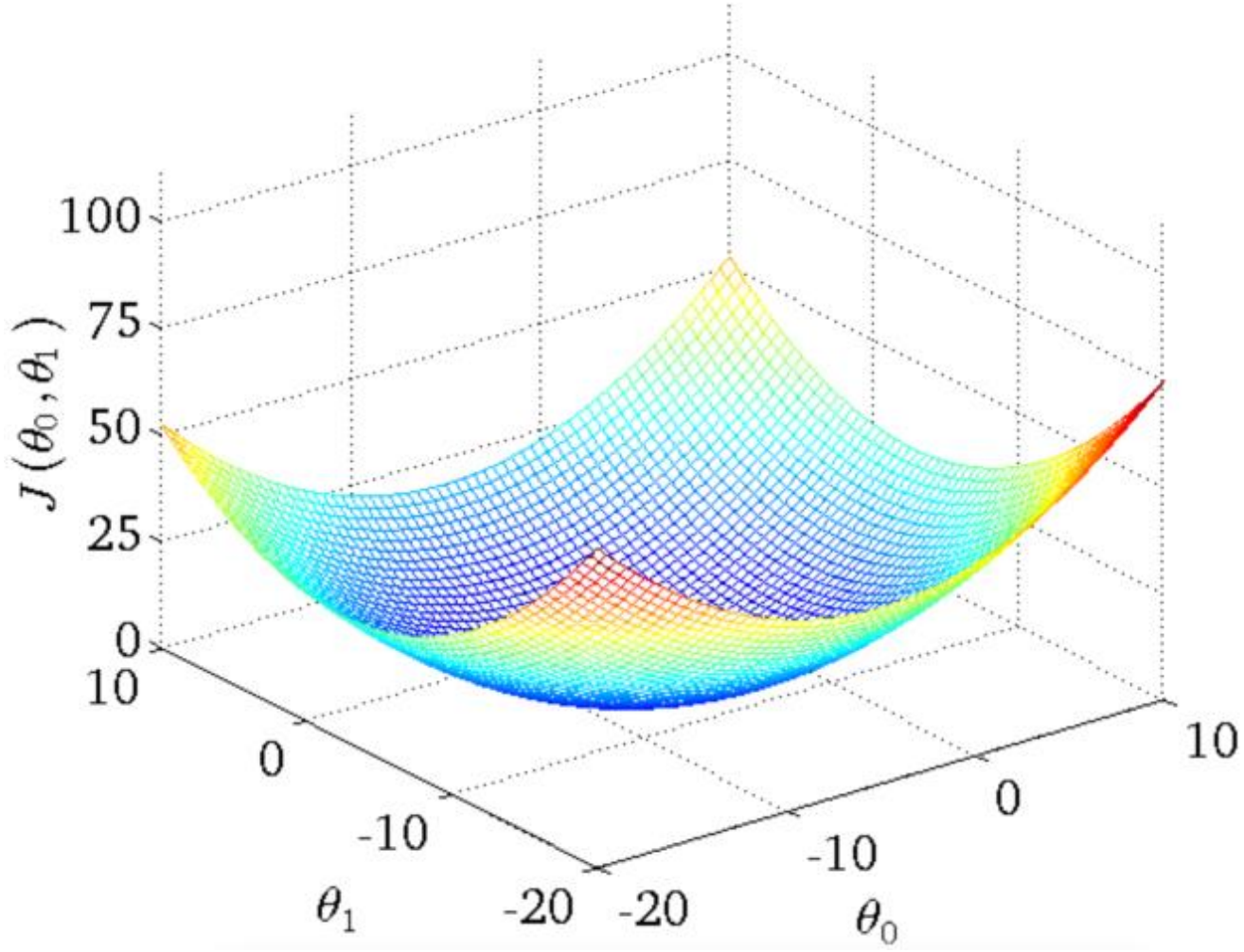


$J(\theta_1)$

(function of the parameter θ_1)



Intuition Behind Cost Function



Formulation

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

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

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

Formulation

Have some function $J(\theta_0, \theta_1)$

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

Outline:

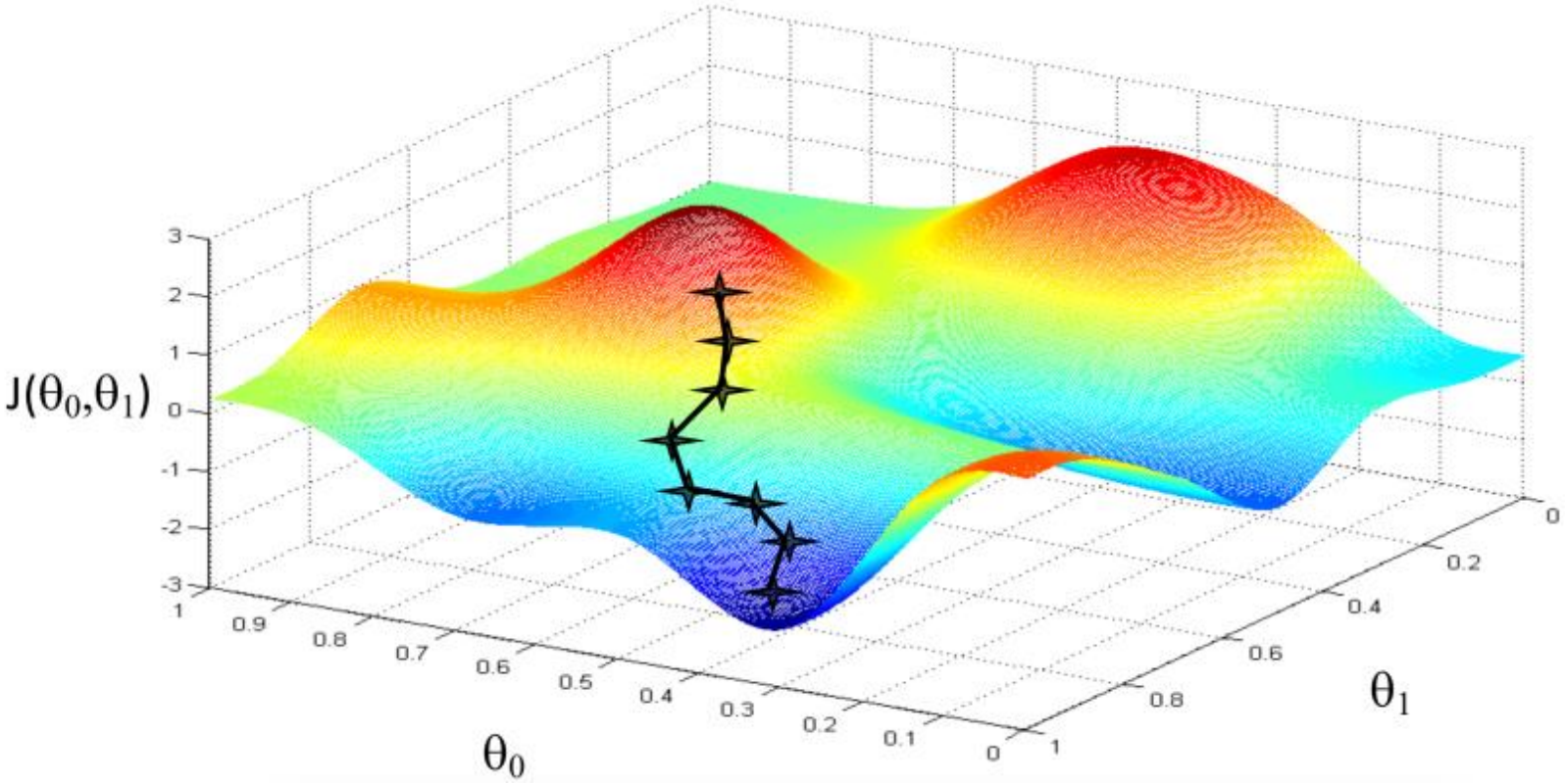
- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce

  until we hopefully end up at a minimum $J(\theta_0, \theta_1)$

Gradient descent

- Choose initial value for θ.
- Until we reach the minimum, update θ to reduce $J(\theta)$:

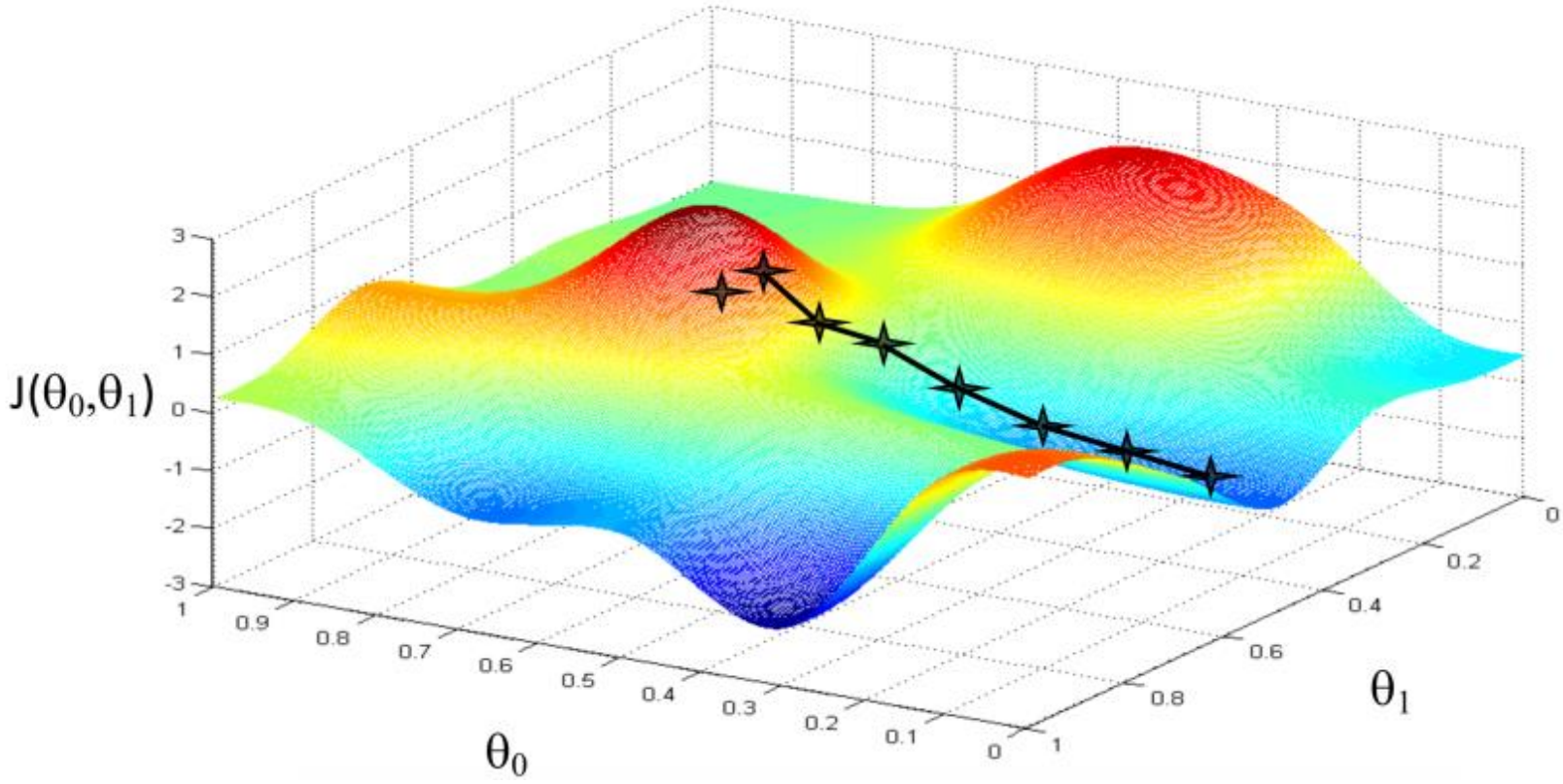
$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$



Gradient descent

- Choose initial value for θ.
- Until we reach the minimum, update θ to reduce $J(\theta)$:

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



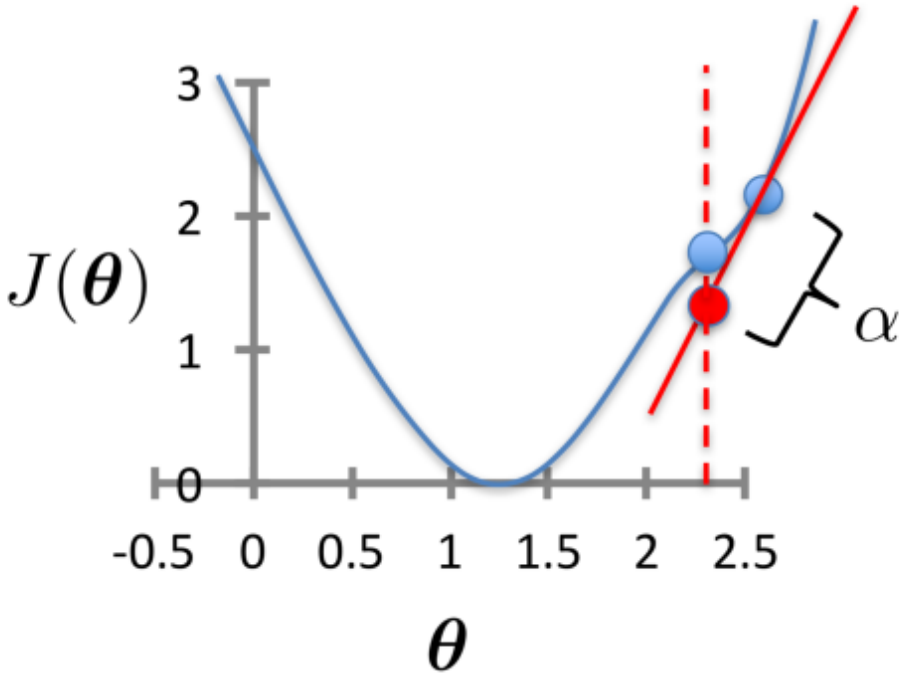
Gradient Descent

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

simultaneous update
for j = 0 1 … d

learning rate (small)
e.g., α = 0.05



Gradient Descent

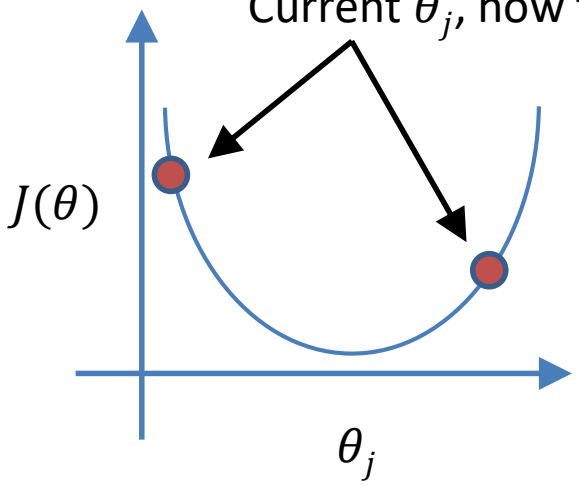
- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

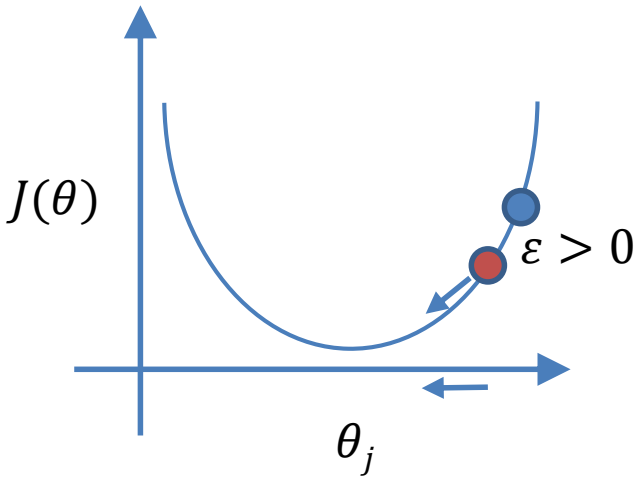
simultaneous update
for j = 0, 1, ... d

Why minus?

Current θ_j, how to update?

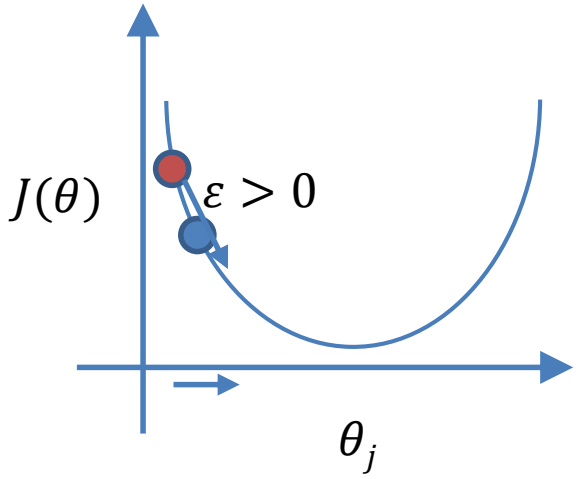


Gradient Descent



$$\frac{\partial J(\theta)}{\partial \theta_j} = \lim_{\varepsilon \to 0} \frac{J(\theta|\theta_j + \varepsilon) - J(\theta|\theta_j)}{\varepsilon} > 0$$

→θ_j moves to the **negative direction** → **counter-direction** w.r.t. the partial derivative.



$$\frac{\partial J(\theta)}{\partial \theta_j} = \lim_{\varepsilon \to 0} \frac{J(\theta|\theta_j + \varepsilon) - J(\theta|\theta_j)}{\varepsilon} < 0$$

→θ_j moves to the **positive direction** → **counter-direction** w.r.t. the partial derivative.

Gradient Descent

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

simultaneous update
for j = 0, 1 ... d

For Linear Regression:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) &= \frac{\partial}{\partial \theta_j} \frac{1}{2n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right)^2 \\
&= \frac{\partial}{\partial \theta_j} \frac{1}{2n} \sum_{i=1}^{n} \left(\sum_{k=0}^{d} \theta_k x_k^{(i)} - y^{(i)} \right)^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{k=0}^{d} \theta_k x_k^{(i)} - y^{(i)} \right) \times \frac{\partial}{\partial \theta_j} \left(\sum_{k=0}^{d} \theta_k x_k^{(i)} - y^{(i)} \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{k=0}^{d} \theta_k x_k^{(i)} - y^{(i)} \right) x_j^{(i)}
\end{aligned}$$

Gradient Descent

What happens with $\frac{\partial}{\partial \theta_0} J(\theta)$???

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{k=0}^{d} \theta_k x_k^{(i)} - y^{(i)} \right)$$

$$j = 1 : \frac{\partial}{\partial \theta_1} J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{k=0}^{d} \theta_k x_k^{(i)} - y^{(i)} \right) x_j^{(i)}$$

Gradient Descent

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{n} \sum_{i=1}^{n} \left(h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right) - y^{(i)} \right) x_j^{(i)}$$

simultaneous update for j = 1 ... d

- To achieve simultaneous update
  - At the start of each GD iteration, compute $h_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)} \right)$
  - Use this stored value in the update step loop

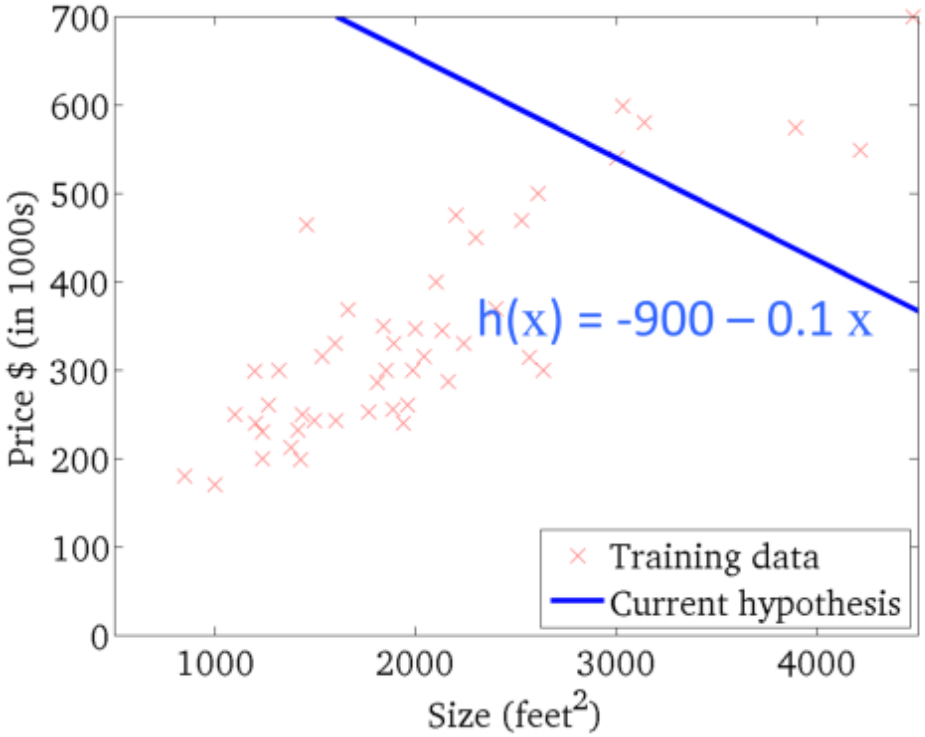
- Assume convergence when $\|\boldsymbol{\theta}_{new} - \boldsymbol{\theta}_{old}\|_2 < \epsilon$

L_2 norm: $\|\boldsymbol{v}\|_2 = \sqrt{\sum_i v_i^2} = \sqrt{v_1^2 + v_2^2 + \ldots + v_{|v|}^2}$

Gradient Descent

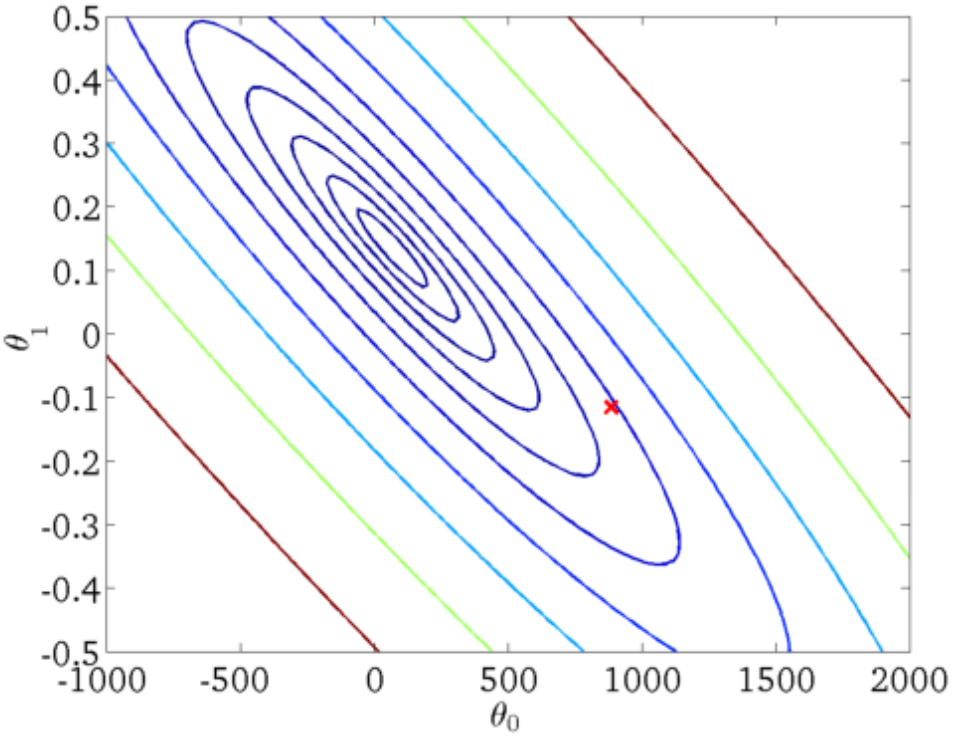
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

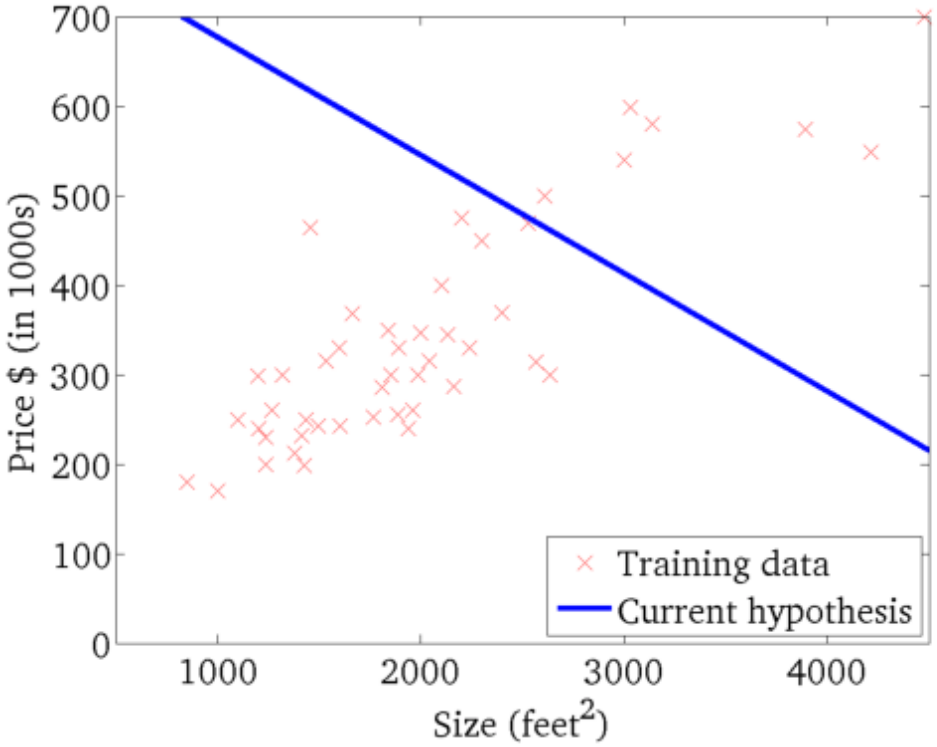
(function of the parameters θ_0, θ_1)



Gradient Descent

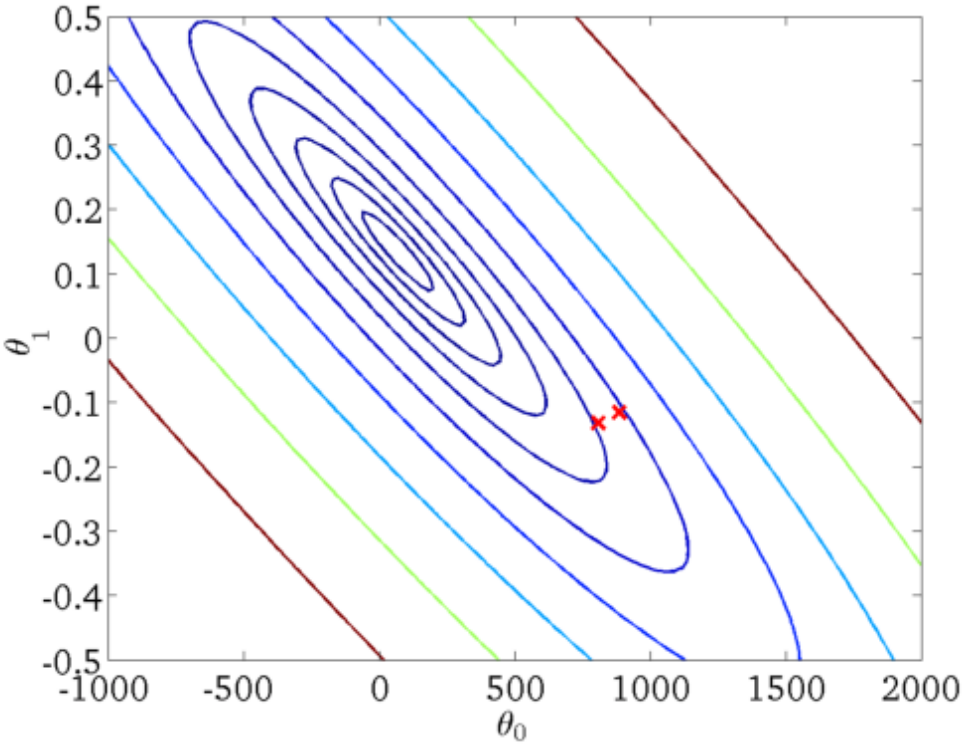
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

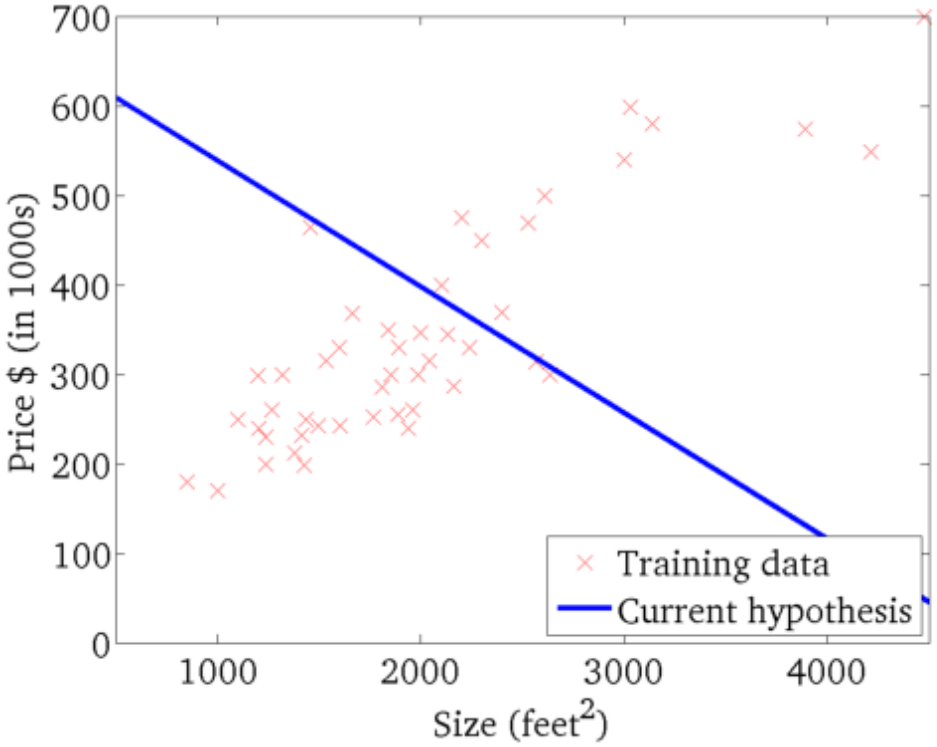
(function of the parameters θ_0, θ_1)



Gradient Descent

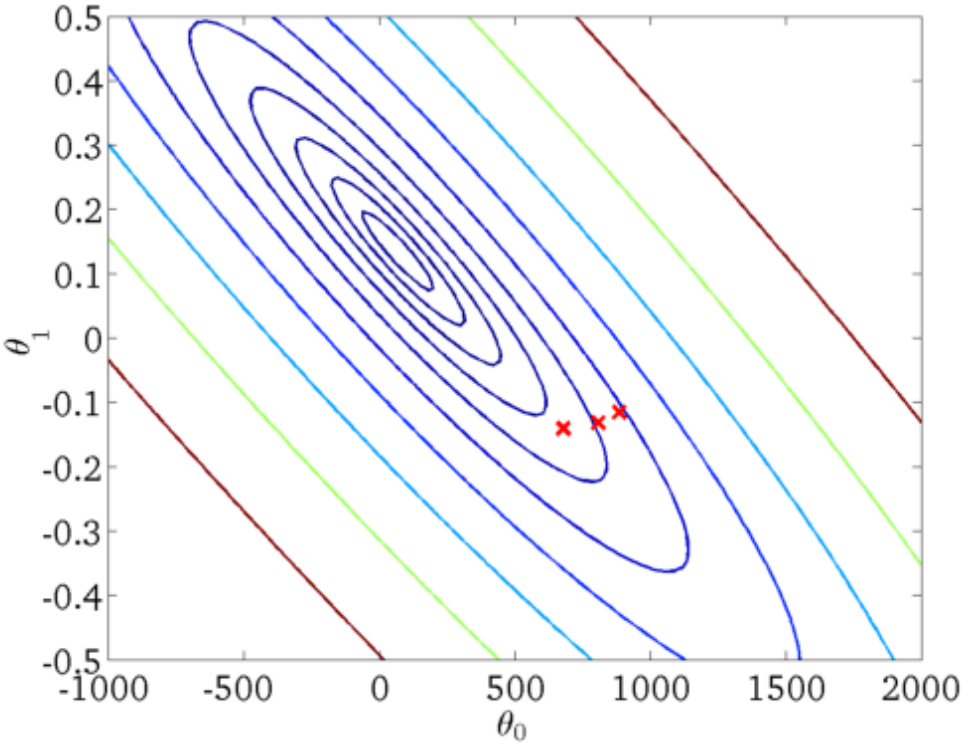
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

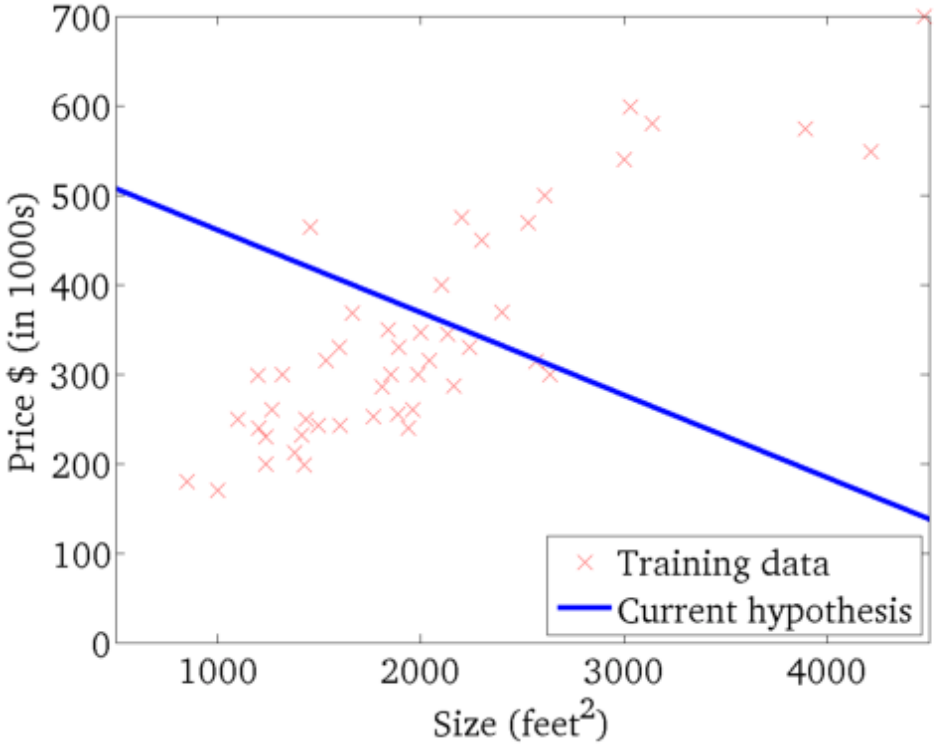
(function of the parameters θ_0, θ_1)



Gradient Descent

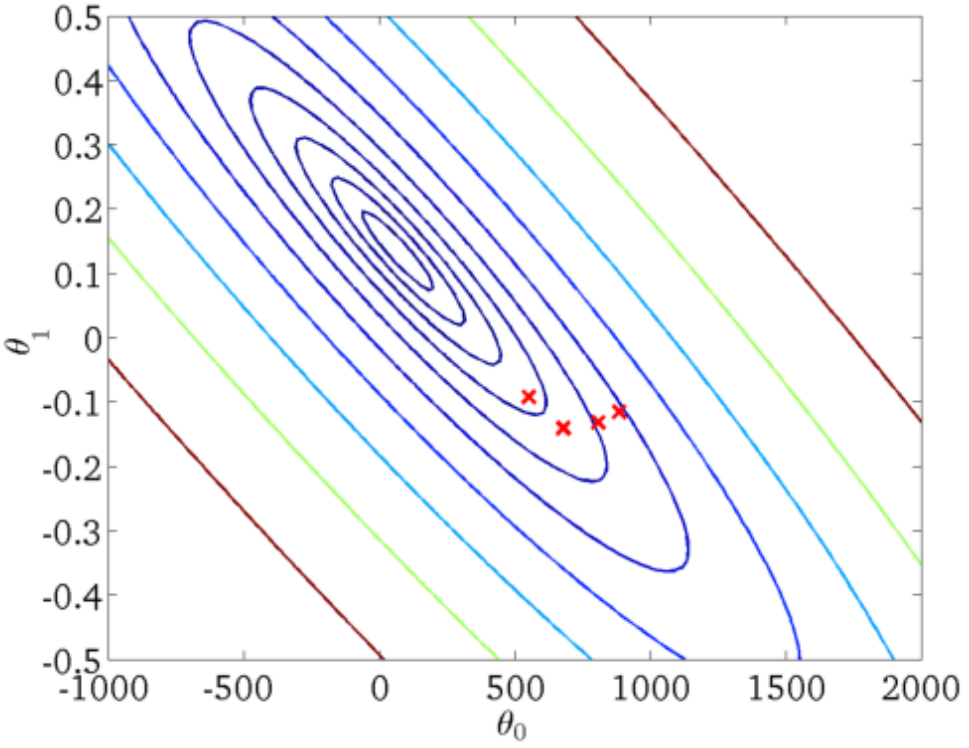
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

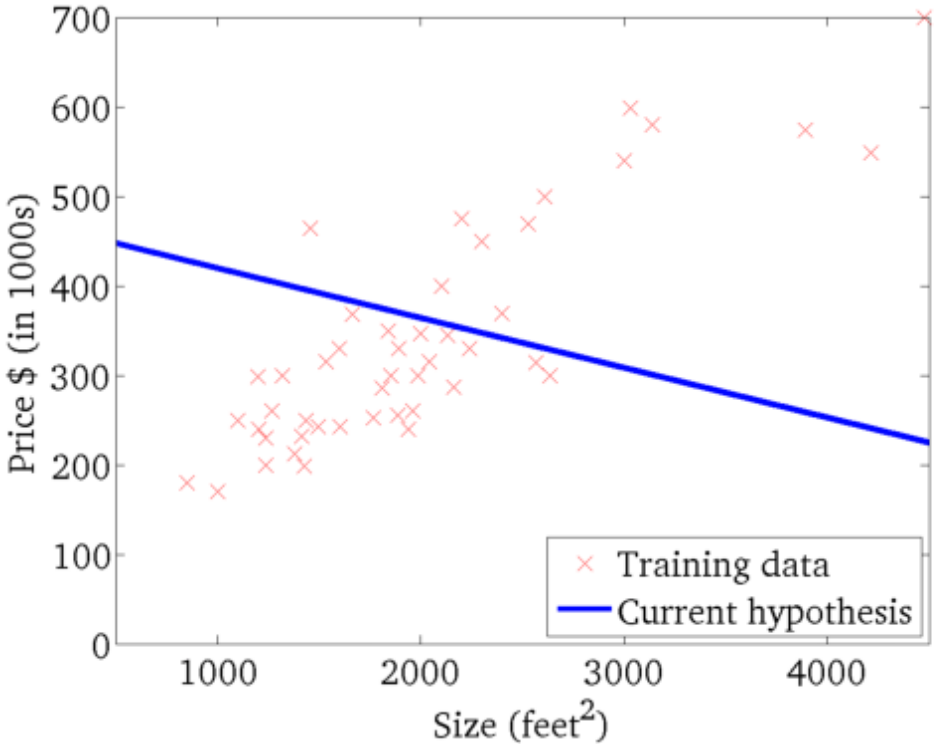
(function of the parameters θ_0, θ_1)



Gradient Descent

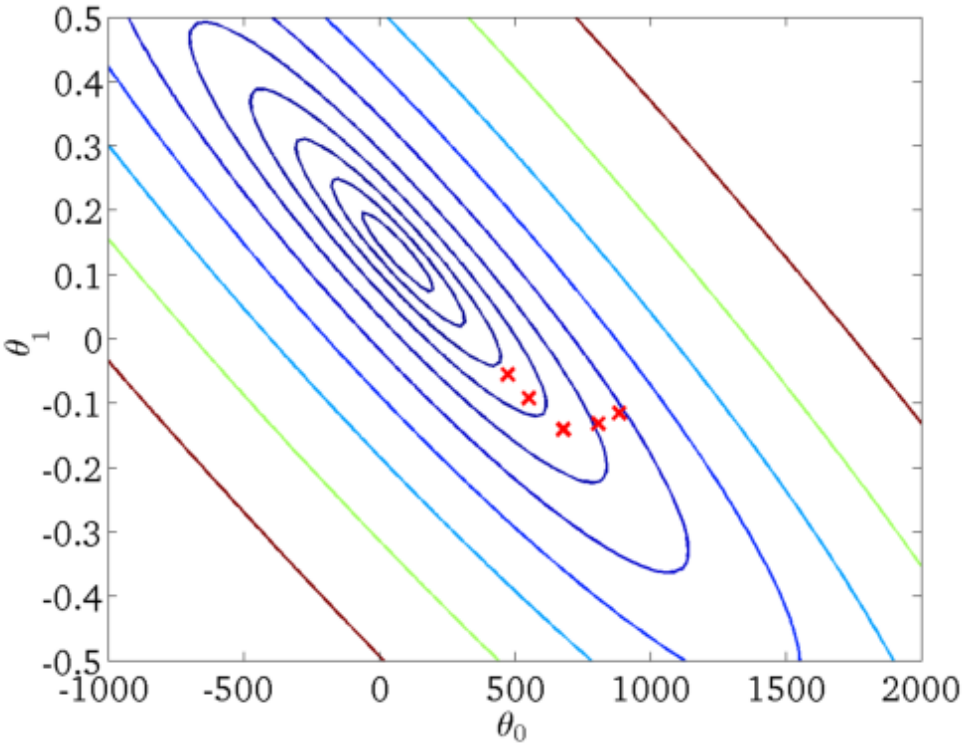
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

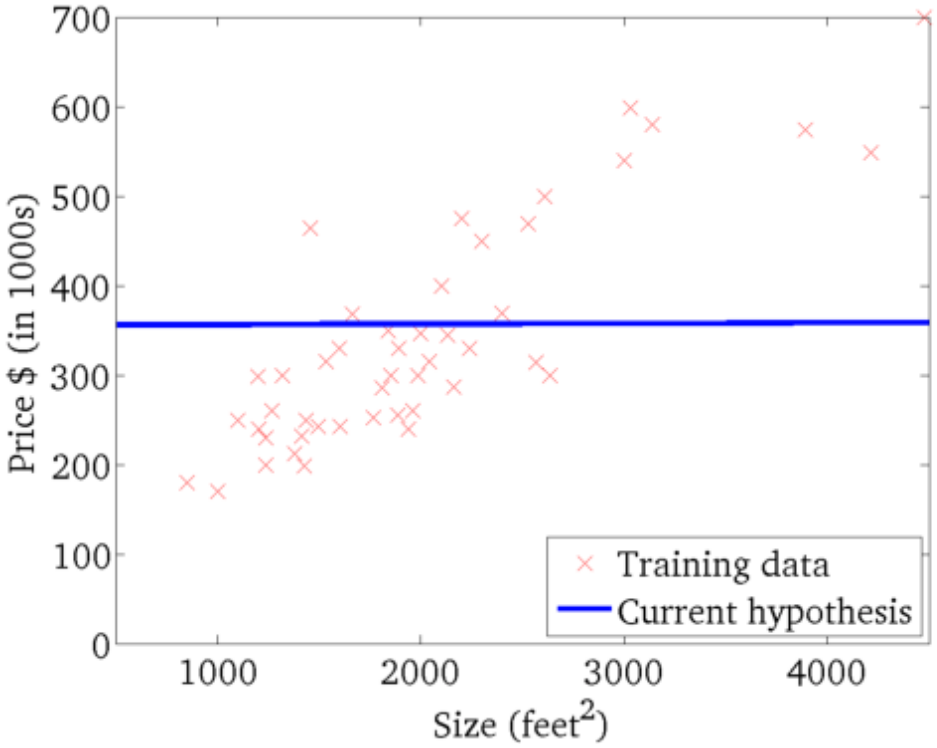
(function of the parameters θ_0, θ_1)



Gradient Descent

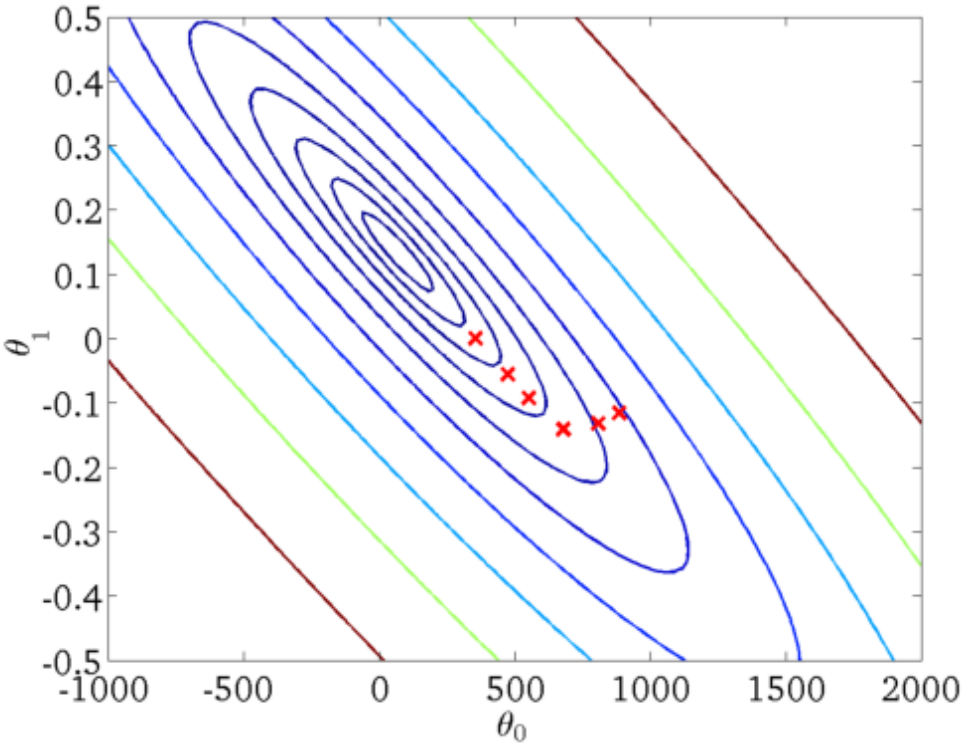
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

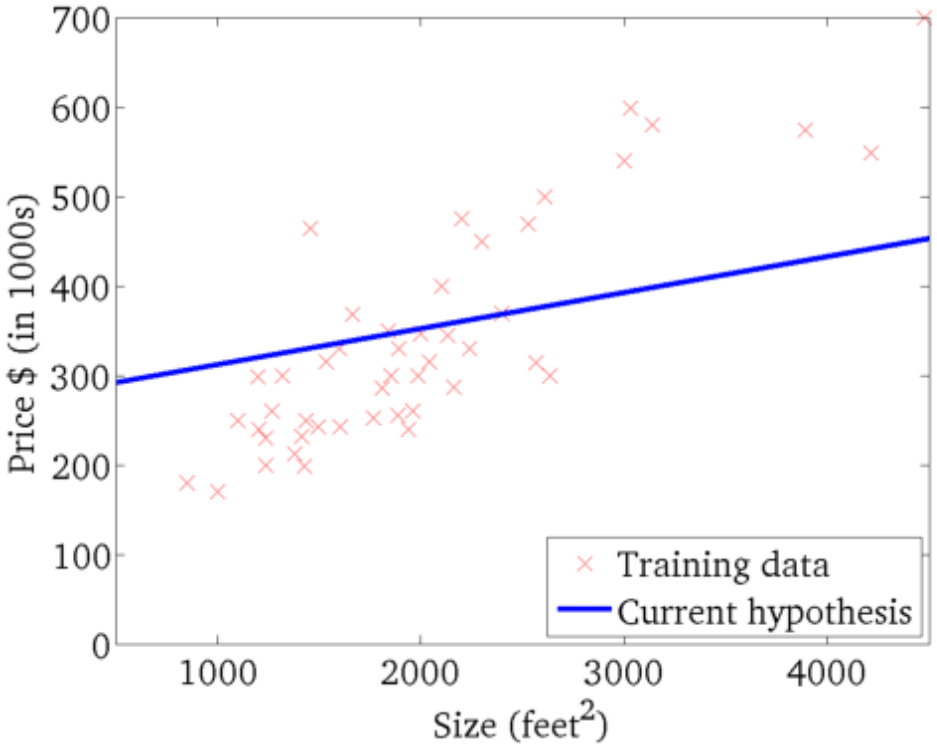
(function of the parameters θ_0, θ_1)



Gradient Descent

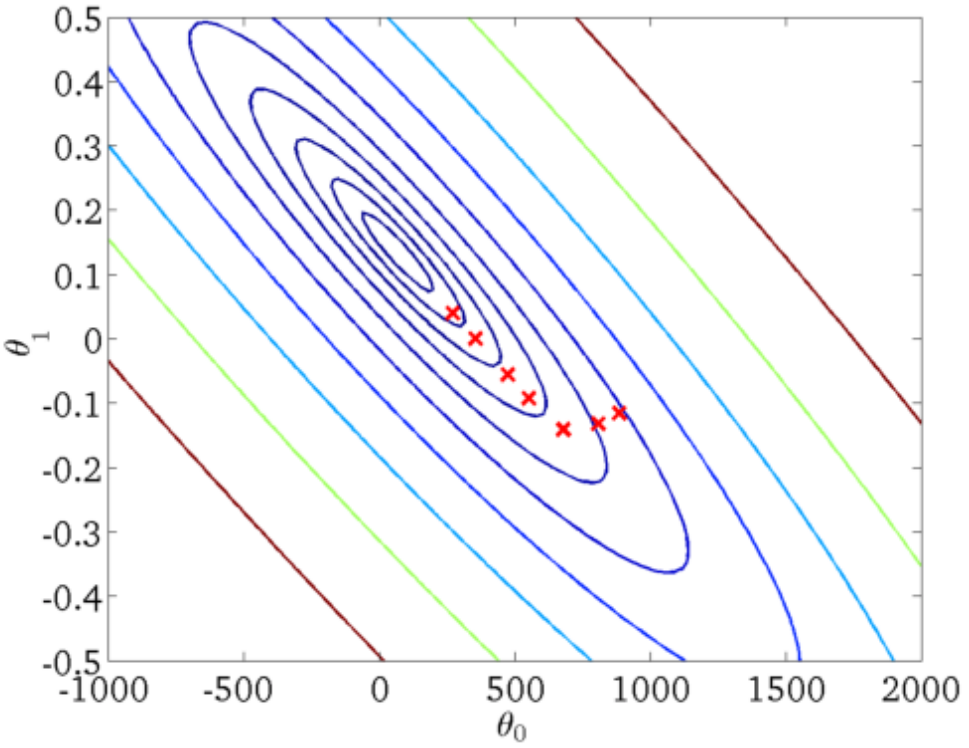
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

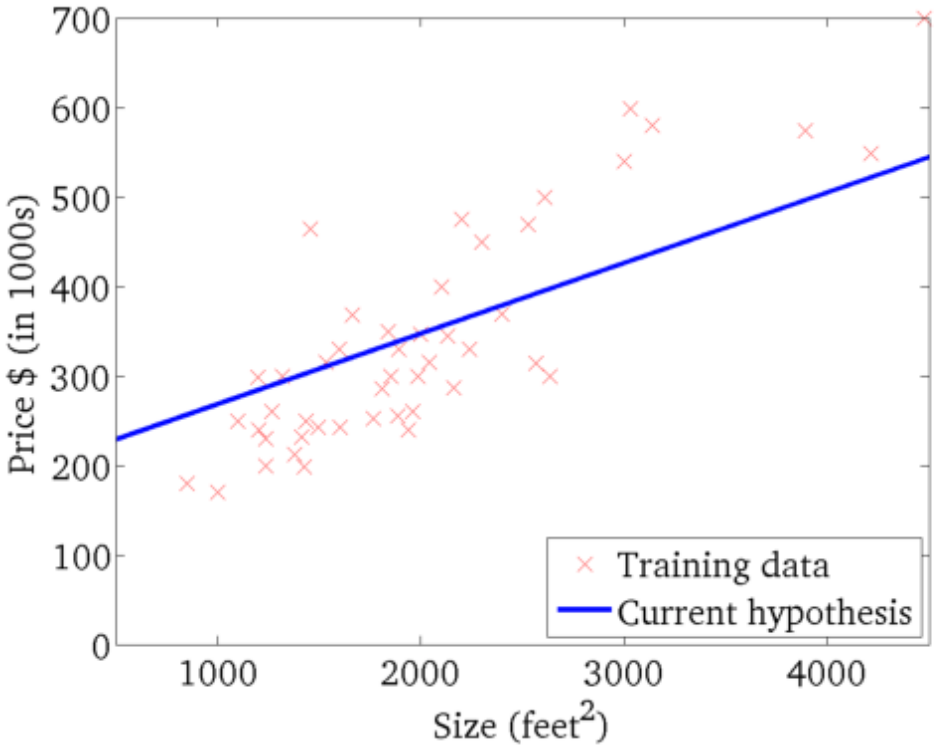
(function of the parameters θ_0, θ_1)



Gradient Descent

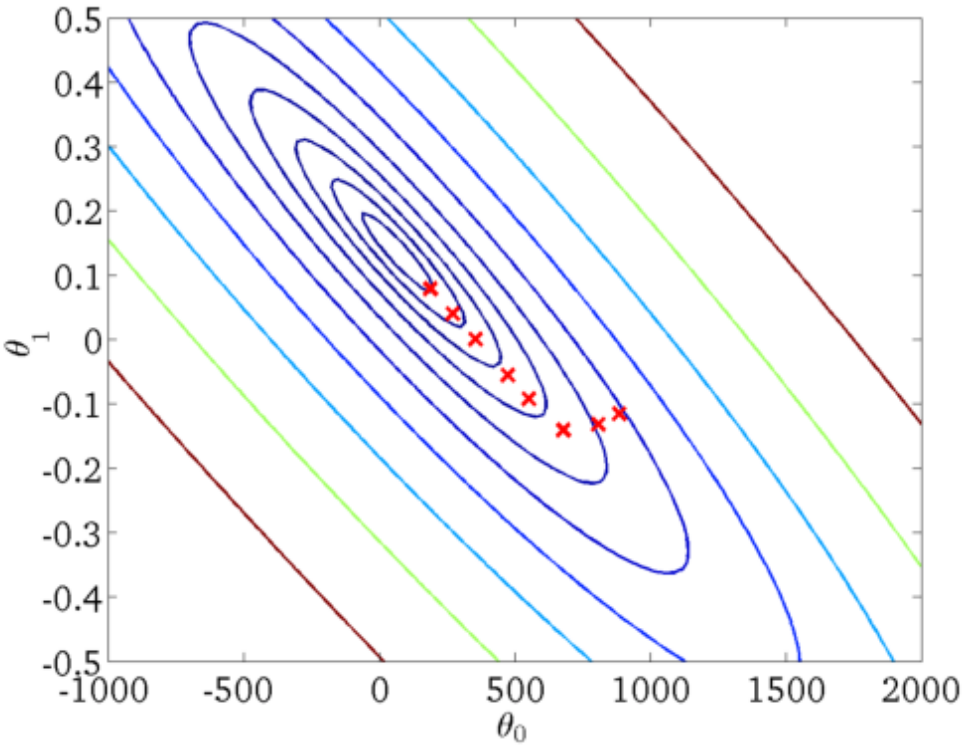
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

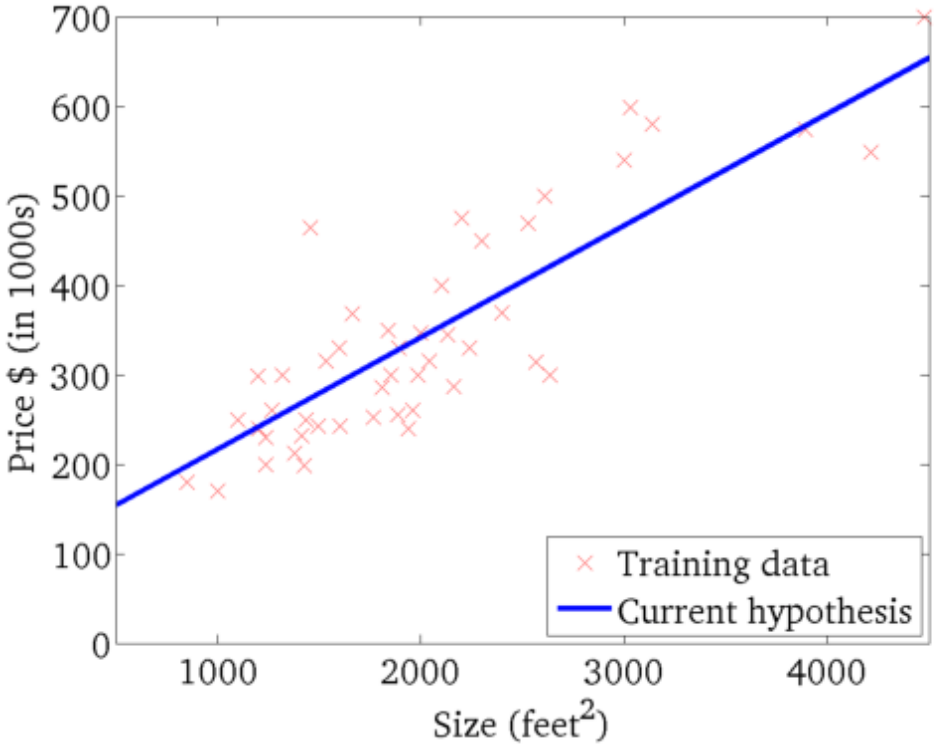
(function of the parameters θ_0, θ_1)



Gradient Descent

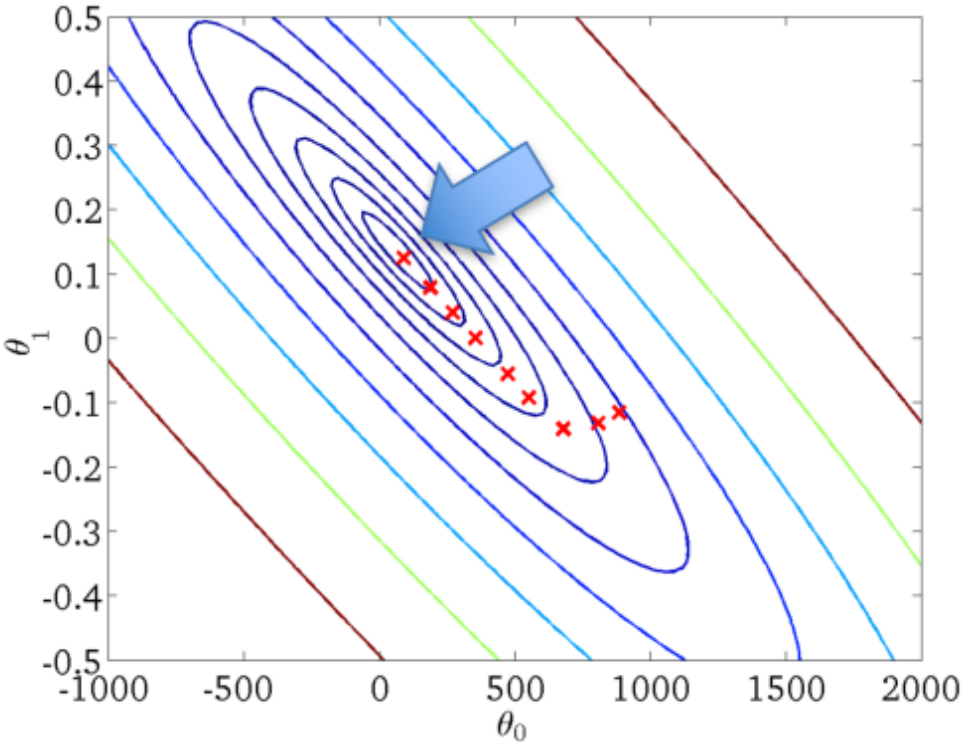
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



Gradient Descent Variants

There are three variants of gradient descent.

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

The difference of these algorithms is the **amount of data.**

$$\theta_j \leftarrow \theta_j - \alpha \underbrace{\frac{\partial}{\partial \theta_j} J(\theta)}$$

This term is different with each method

Batch gradient descent

This method computes the gradient of the cost function with the **entire training dataset**.

We need to calculate the gradients for the whole dataset to perform **just one update**.

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

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Batch gradient descent

Advantage

- It is guaranteed to converge to the **global minimum for convex error surfaces** and to a **local minimum for nonconvex surfaces**.

Disadvantages

- It can be **very slow**.
- It is intractable for datasets that **do not fit in memory**.
- It **does not allow** us to update our model **online**.

Stochastic gradient descent

- This method performs a parameter update for **each training example** $x^{(i)}$ and label $y^{(i)}$.
- We need to calculate the gradient for **each training example** to perform **one update**.
- **Note**: we **shuffle** the training data at every epoch

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta, x^{(i)}, y^{(i)})$$

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

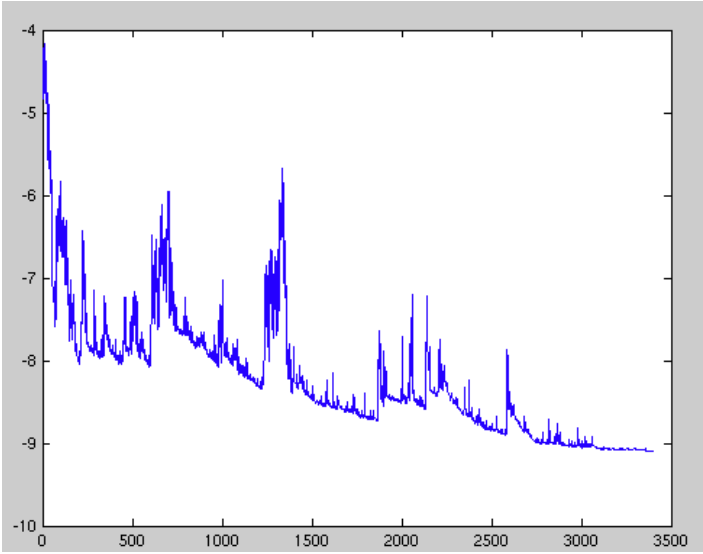
Stochastic gradient descent

Advantage

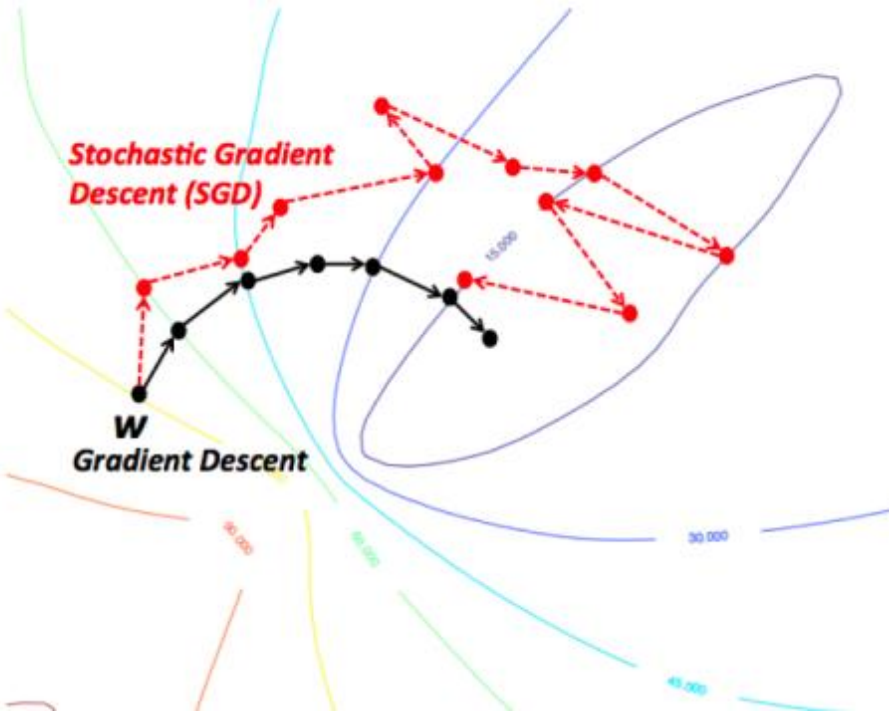
- It is usually much faster than batch gradient descent.
- It can be used to learn online.

Disadvantage

- It performs frequent updates with a high variance that cause the objective function to fluctuate heavily.



The fluctuation : Batch vs SGD



Batch gradient descent converges to the minimum of the basin the parameters are placed in and the **fluctuation is small**.

SGD's **fluctuation is large** but it enables to jump to new and potentially better local minima.

However, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting.

Learning rate of SGD

When we **slowly decrease the learning rate**, SGD shows the same convergence behavior as batch gradient descent

It **almost certainly converging** to a local or the global minimum for non-convex and convex optimization respectively.

Mini-batch gradient descent

This method takes the best of both **batch** and **SGD**, and performs an update for every **mini-batch of *k*** training examples.

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta, x^{(i:i+k)}, y^{(i:i+k)})$$

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size = 50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

Mini-batch gradient descent

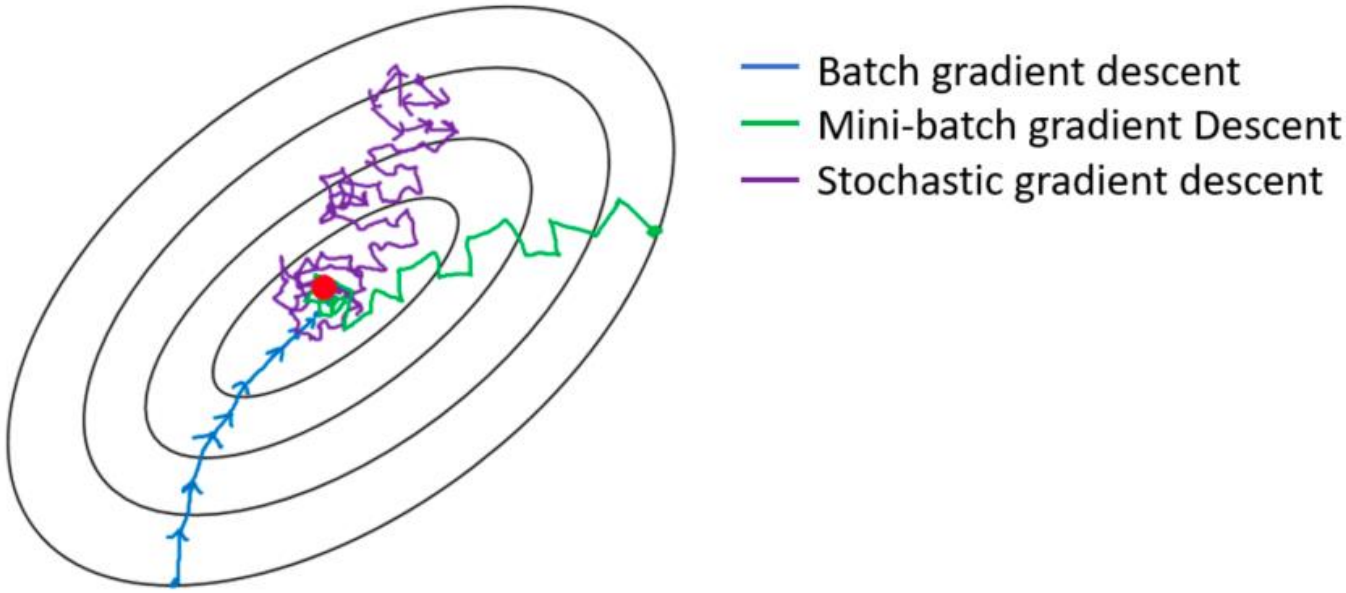
Advantage

- It **reduces the variance** of the parameter updates.
    - This can lead to more stable convergence.
- It can make use of highly optimized matrix optimizations common to deep learning libraries that make computing the gradient very efficiently.

Disadvantage

- We have to set mini-batch size.
    - Common mini-batch sizes range between 50 and 256, but can vary for different applications.

Gradient Descent Variants



Gradient descent variants' trajectory towards minimum

Trade-off

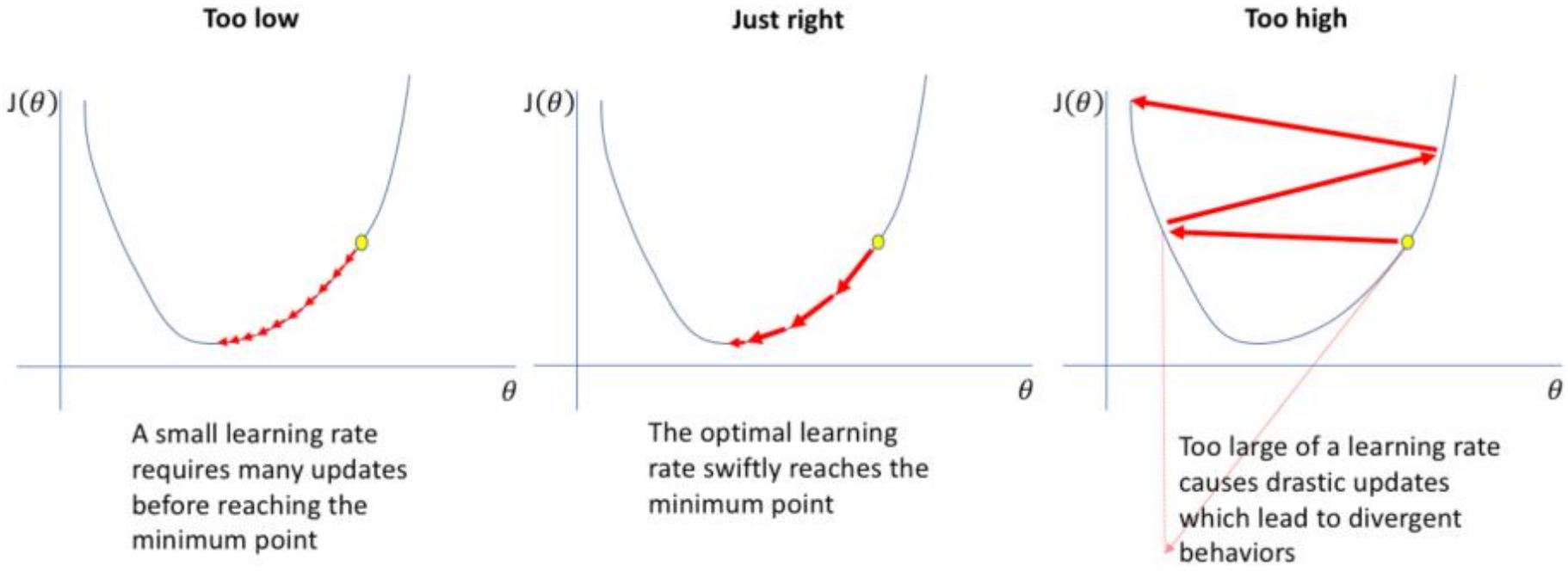
Depending on the amount of data, they make a trade-off:

- The **accuracy** of the parameter update
- The **time** it takes to perform an update.
- The **memory** it takes to perform an update.
- Capability of **online learning**.

Method	Accuracy	Time	Memory Usage	Online Learning
Batch gradient descent	○	Slow	High	×
Stochastic gradient descent	△	High	Low	○
Mini-batch gradient descent	○	Medium	Medium	○

Choosing Learning Rate

- Choosing a **proper learning rate** is difficult.
- The settings of **learning rate schedule** is difficult.
- **Changing learning rate for each parameter** is difficult.
- Avoiding getting trapped in highly non-convex error functions' numerous suboptimal local minima is difficult.



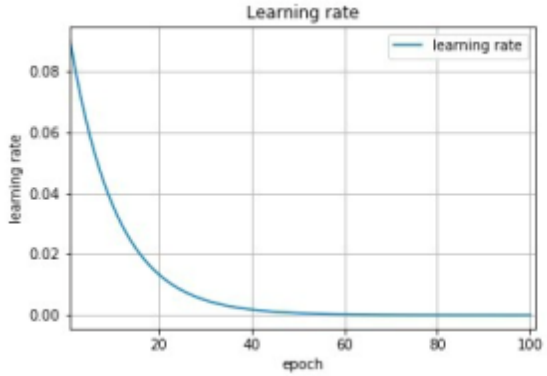
Learning Rate with Exponential Decay

- When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies an exponential decay function to a provided initial learning rate

```
decayed_learning_rate = learning_rate *
                        decay_rate ^ (global_step / decay_steps)
```

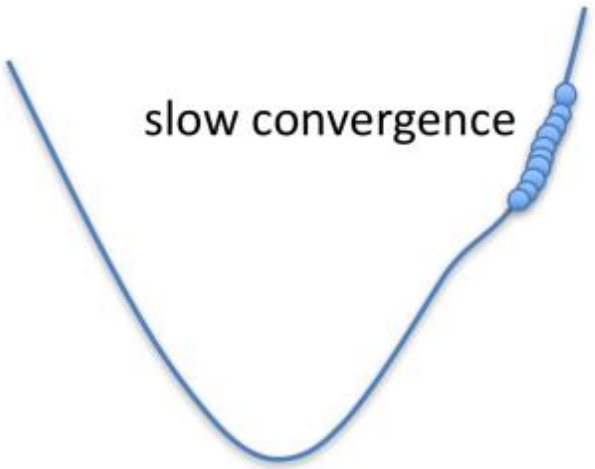
```
global_step = tf.Variable(0, trainable=False)
starter_learning_rate = 0.1
learning_rate = tf.train.exponential_decay(starter_learning_rate, global_step,
                                           100000, 0.96, staircase=True)
# Passing global_step to minimize() will increment it at each step.
learning_step = (
    tf.train.GradientDescentOptimizer(learning_rate)
    .minimize(...my loss..., global_step=global_step)
)
```

```
tf.train.exponential_decay(
    learning_rate,
    global_step,
    decay_steps,
    decay_rate,
    staircase=False,
    name=None
)
```

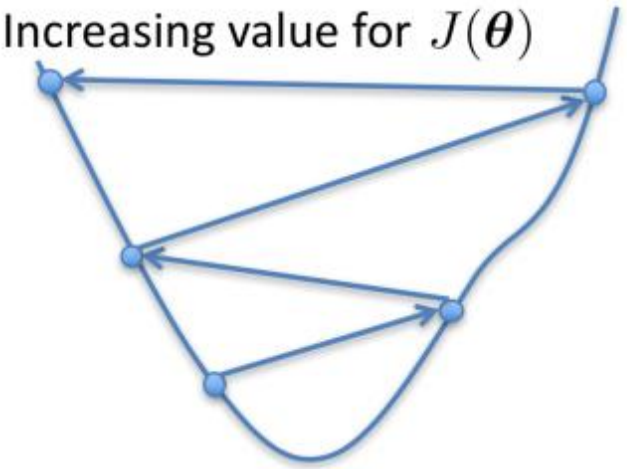


Choosing α

α too small



α too large



- May overshoot the minimum
- May fail to converge
- May even diverge

To see if gradient descent is working, print out $J(\theta)$ each iteration

- The value should decrease at each iteration
- If it doesn't, adjust α

Extending Linear Regression to More Complex Models

- The inputs **X** for linear regression can be:
  - Original quantitative inputs
  - Transformation of quantitative inputs
    - e.g. log, exp, square root, square, etc.
  - Polynomial transformation **a.k.a. Polynomial (linear) regression!**
    - example: $y = \beta_0 + \beta_1 \cdot x + \beta_2 \cdot x^2 + \beta_3 \cdot x^3$
  - Basis expansions
  - Dummy coding of categorical inputs
  - Interactions between variables
    - example: $x_3 = x_1 \cdot x_2$

This allows use of linear regression techniques to fit non-linear datasets.

Linear Basis Function Models

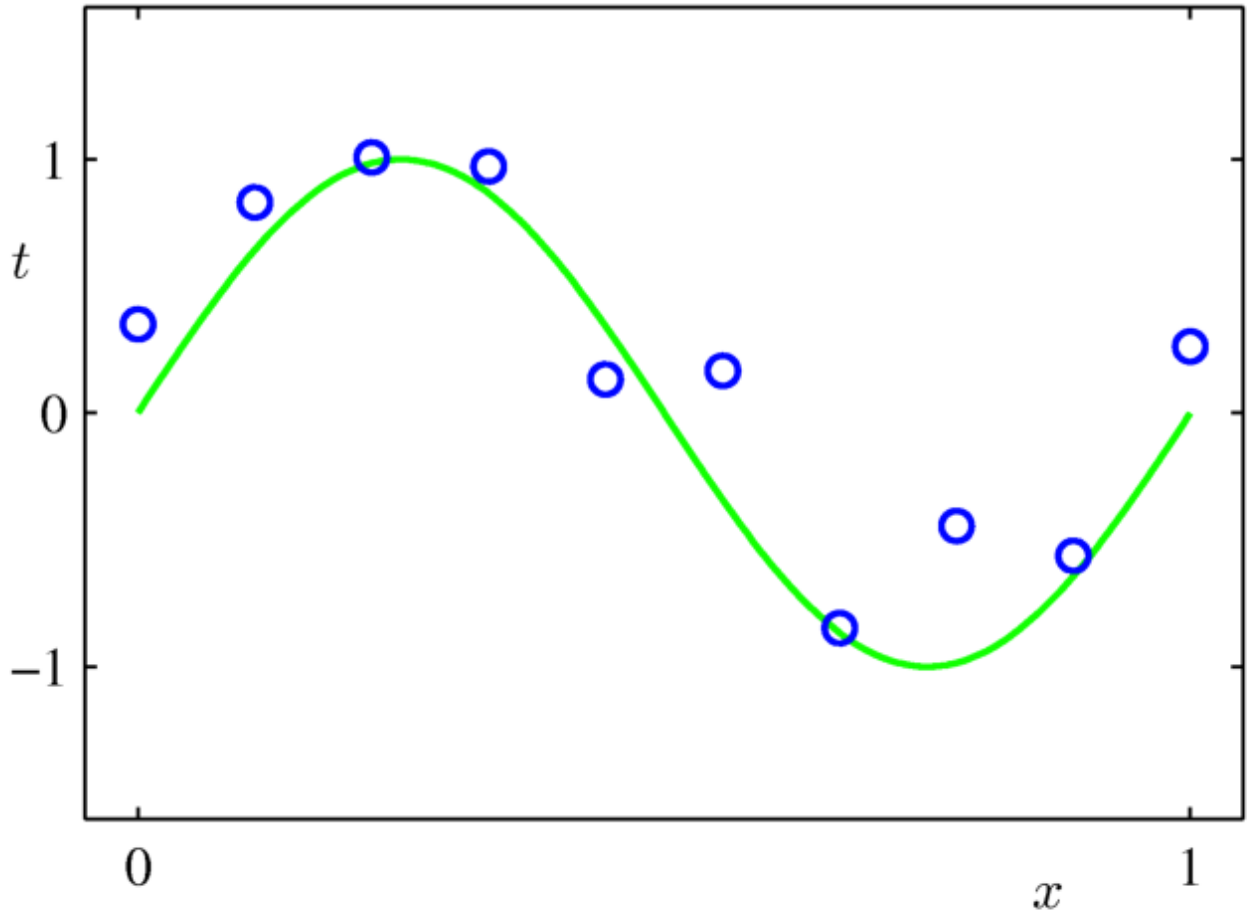
- Generally,

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=0}^{d} \theta_j \underbrace{\phi_j(\boldsymbol{x})}_{\text{basis function}}$$

- Typically, $\phi_0(\boldsymbol{x}) = 1$ so that θ_0 acts as a bias
- In the simplest case, we use linear basis functions :

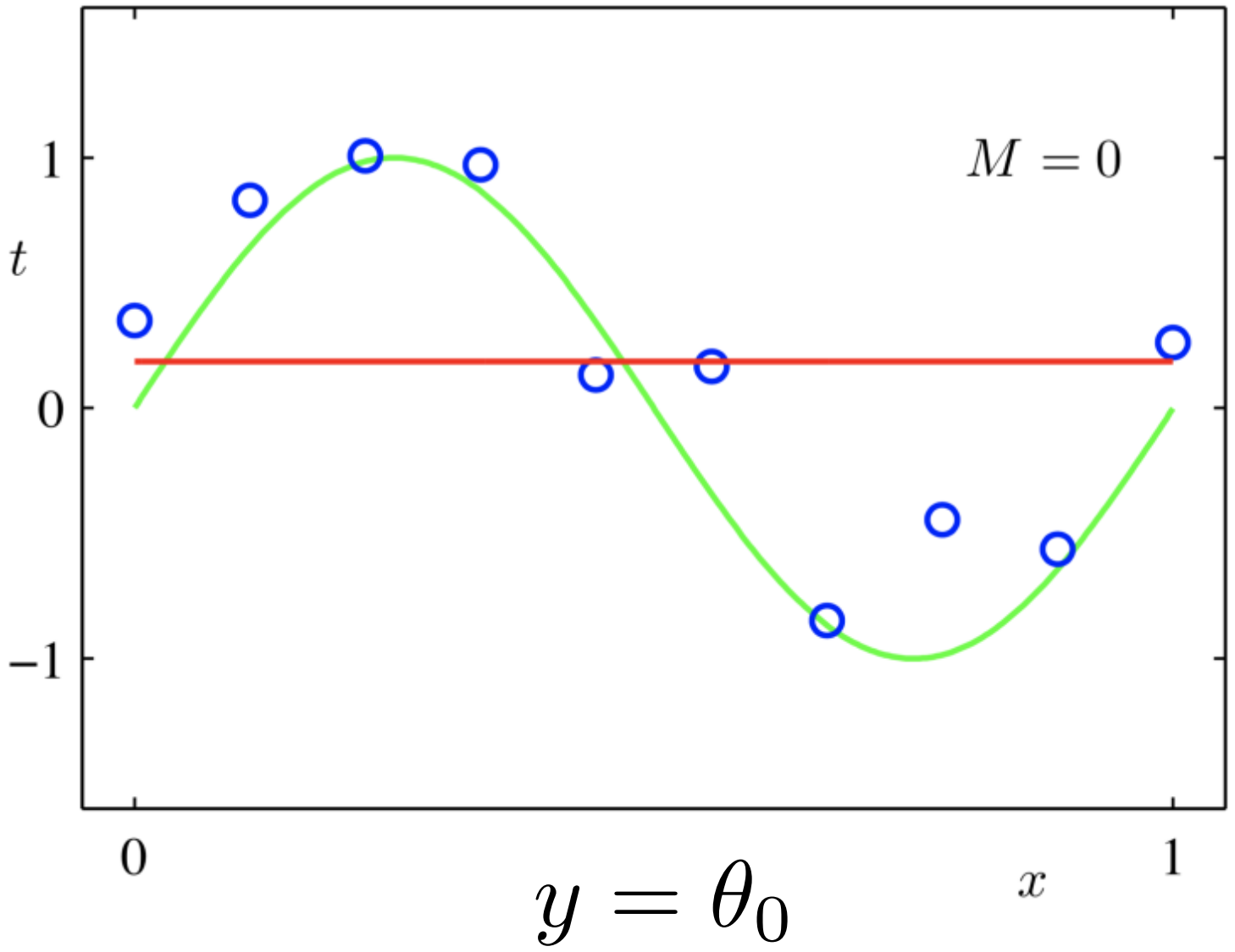
$$\phi_j(\boldsymbol{x}) = x_j$$

Example of Fitting a Polynomial Curve with a Linear Model

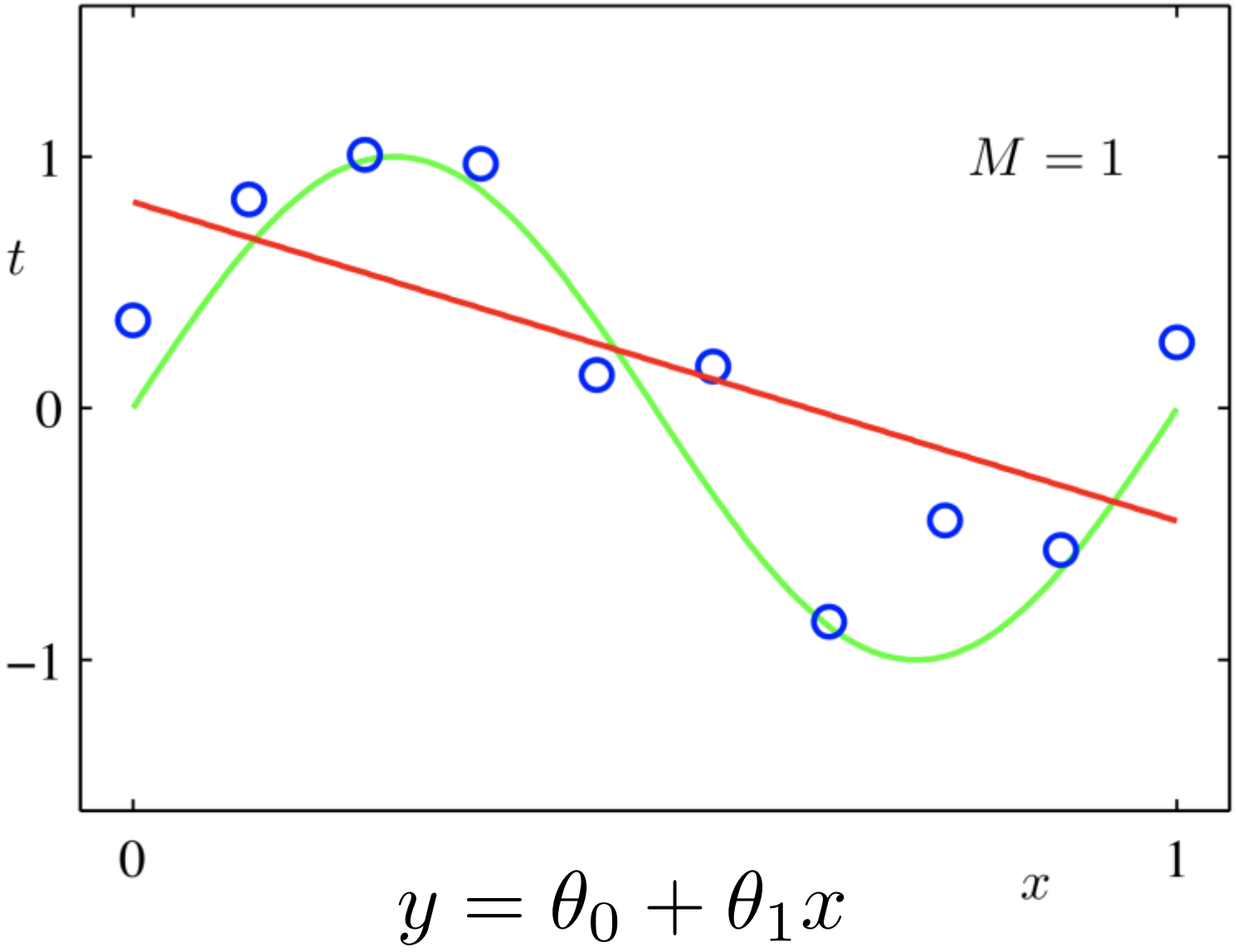


$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_p x^p = \sum_{j=0}^{p} \theta_j x^j$$

Fitting polynomials

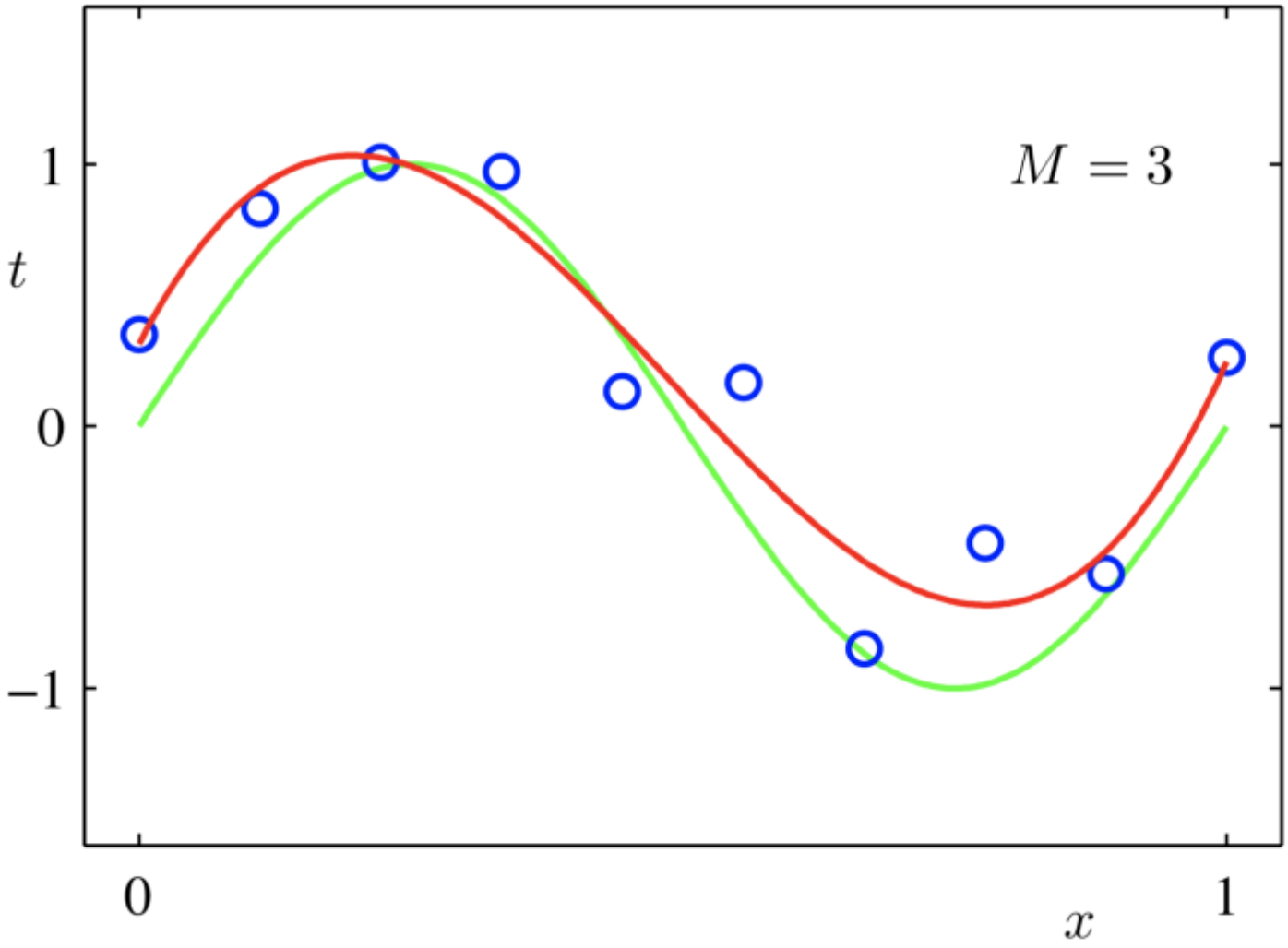


Fitting polynomials



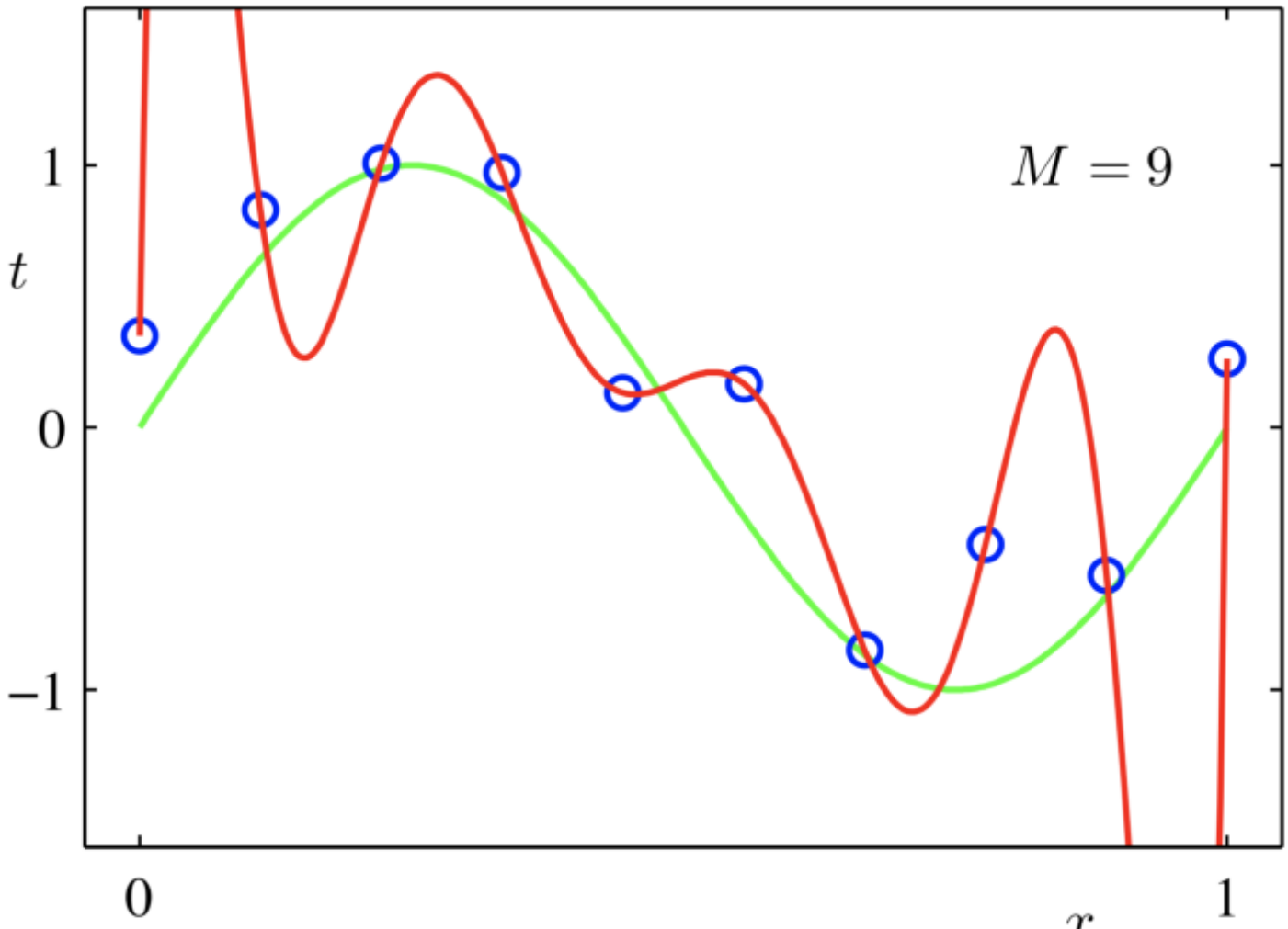
$$y = \theta_0 + \theta_1 x$$

Fitting polynomials



$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

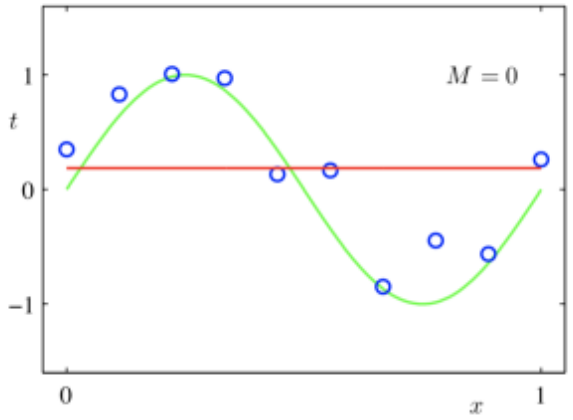
Fitting polynomials



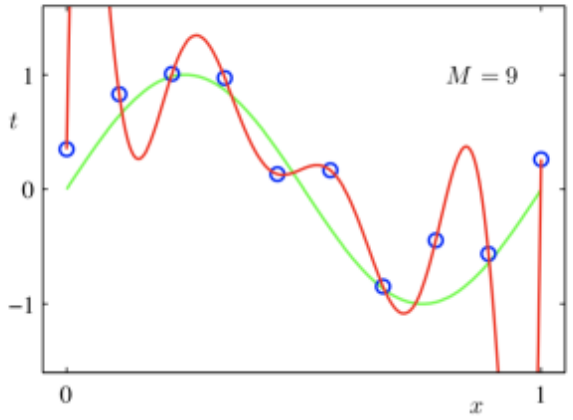
$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \ldots + \theta_9 x^9$$

Generalization

Underfitting : The model is too simple - does not fit the data.

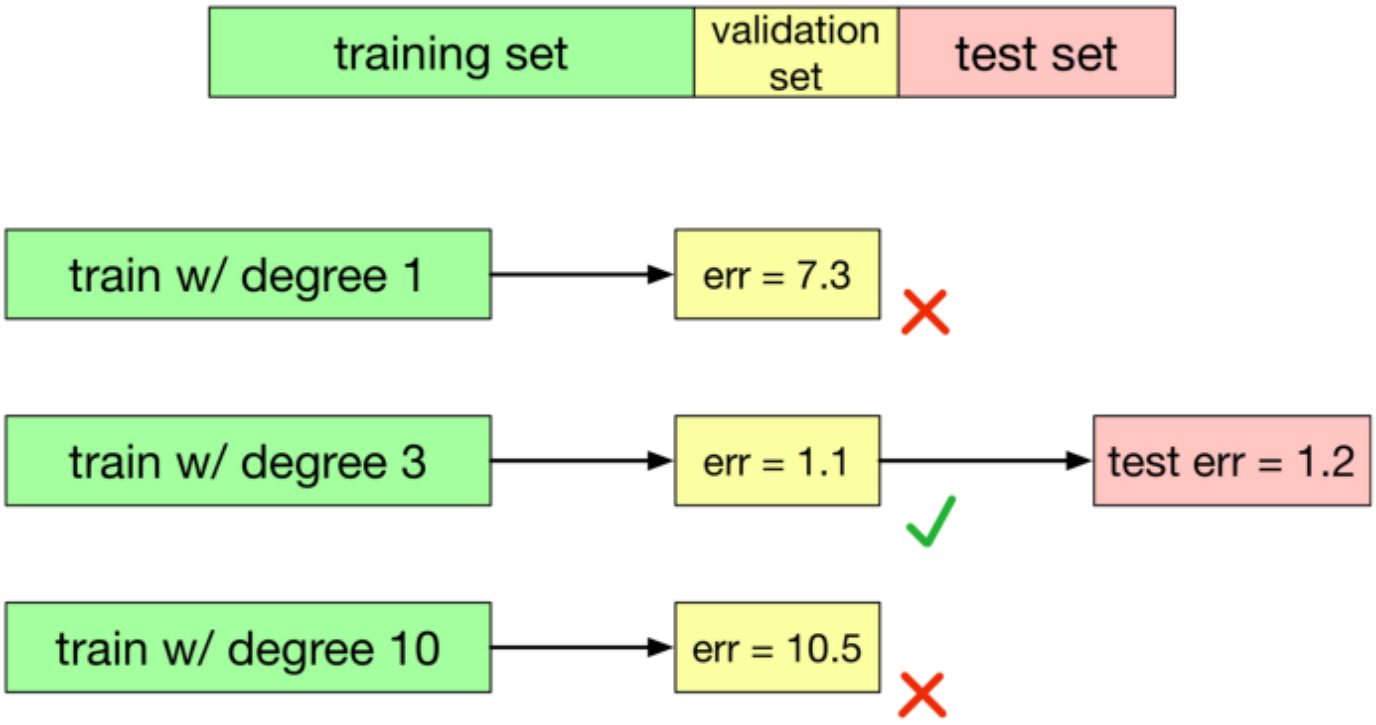


Overfitting : The model is too complex - fits perfectly, does not generalize.



Generalization

- We would like our models to generalize to data they haven't seen before
- The degree of the polynomial is an example of a hyperparameter, something we can't include in the training procedure itself
- We can tune hyperparameters using a validation set:



Foreshadowing

Feature maps aren't a silver bullet:

- It's not always easy to pick good features.
- In high dimensions, polynomial expansions can get very large!

Until the last few years, a large fraction of the effort of building a good machine learning system was feature engineering

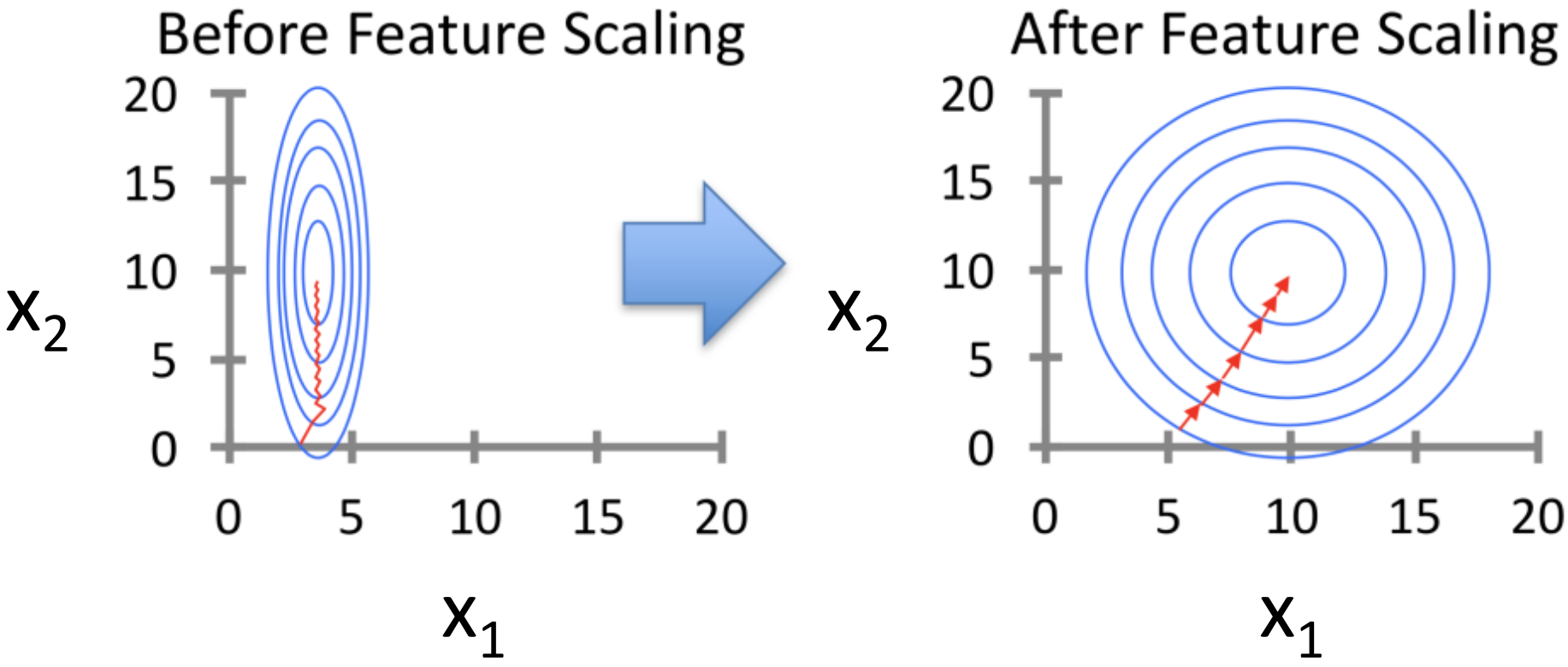
We'll see that neural networks are able to learn nonlinear functions directly, avoiding hand-engineering of features

Linear Basis Function Models

- Basic Linear Model: $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=0}^{d} \theta_j x_j$
- Generalized Linear Model: $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{j=0}^{d} \theta_j \phi_j(\boldsymbol{x})$
- Once we have replaced the data by the outputs of the basis functions, fitting the generalized model is exactly the same problem as fitting the basic model

Improving Learning: Feature Scaling

- **Idea:** Ensure that feature have similar scales



- Makes gradient descent converge *much* faster

Standardization

- Rescales features to have zero mean and unit variance
  - Let μ_j be the mean of feature j: $\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)}$
  - Replace each value with:

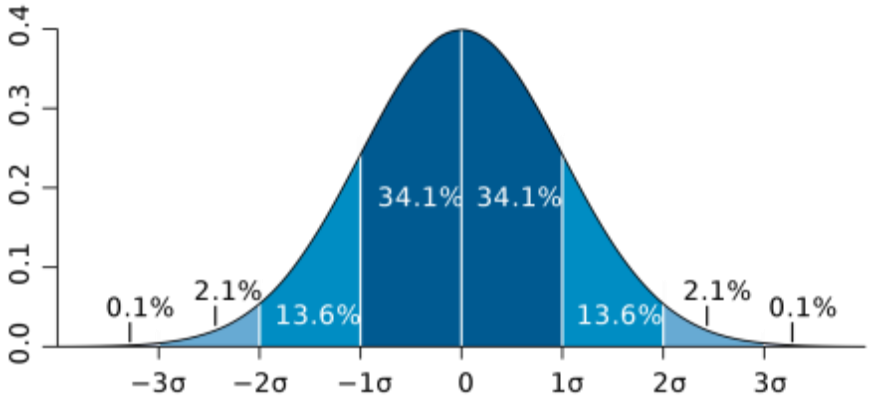
$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j} \qquad \text{for j = 1...d (not } x_0\text{!)}$$

    - s_j is the standard deviation of feature j
    - Could also use the range of feature j ($max_j - min_j$) for s_j
- Must apply the same transformation to instances for both training and prediction
- Outliers can cause problems

This redistributes the features with their mean **μ=0** and standard deviation **σ=1**. *sklearn.preprocessing.scale* helps us implementing standardization in python.

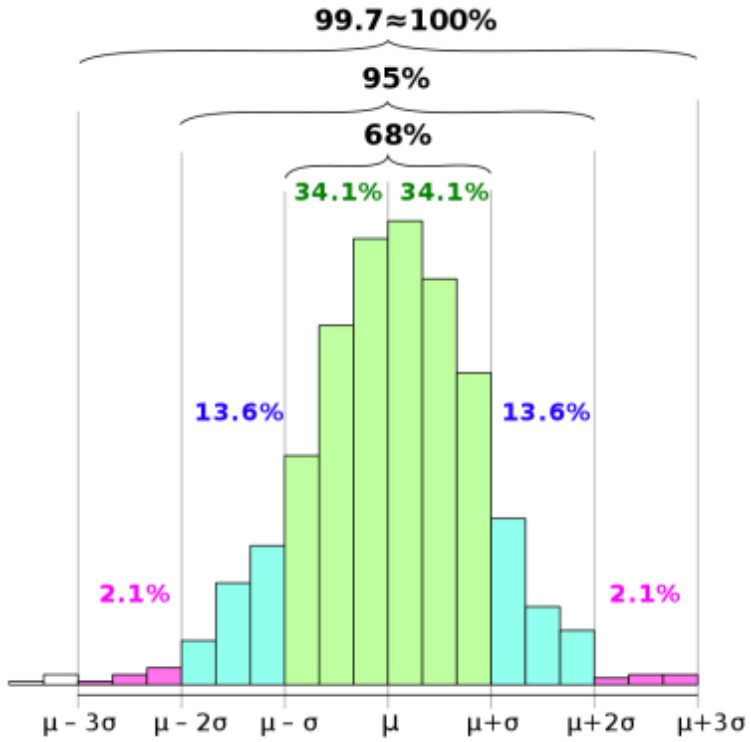
Improving Learning: Feature Scaling

Standardization



Standard deviation

$$s_j = \sqrt{\frac{\sum_{i=1}^{n} |x_j^{(i)} - \mu_j|}{n}}$$



68–95–99.7 rule

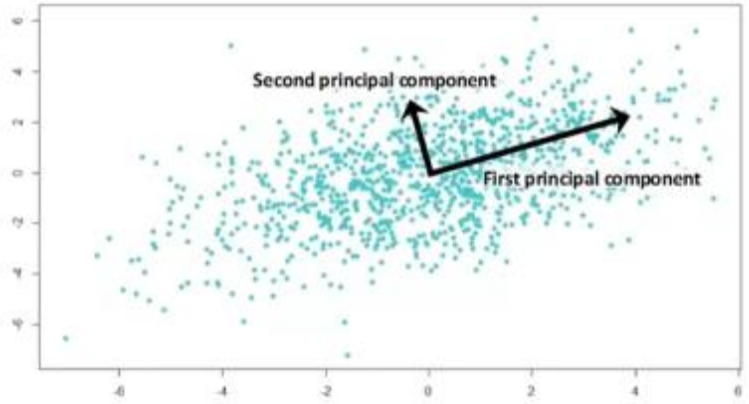
$$\Pr(\mu - 1\sigma \leq X \leq \mu + 1\sigma) \approx 0.6827$$
$$\Pr(\mu - 2\sigma \leq X \leq \mu + 2\sigma) \approx 0.9545$$
$$\Pr(\mu - 3\sigma \leq X \leq \mu + 3\sigma) \approx 0.9973$$

Mean Normalization

$$x' = \frac{x - mean(x)}{max(x) - min(x)}$$

This distribution will have values between **[-1,1]** with **μ=0**.

Standardization and **Mean Normalization** can be used for algorithms that assumes zero centric data like **Principal Component Analysis (PCA)**.



Min-Max Scaling

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

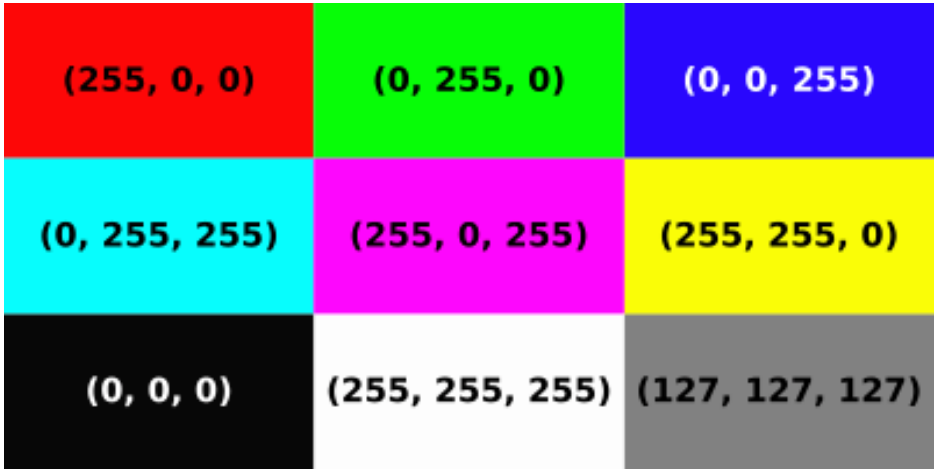
This scaling brings the value in **[0,1]**.

Unit Vector

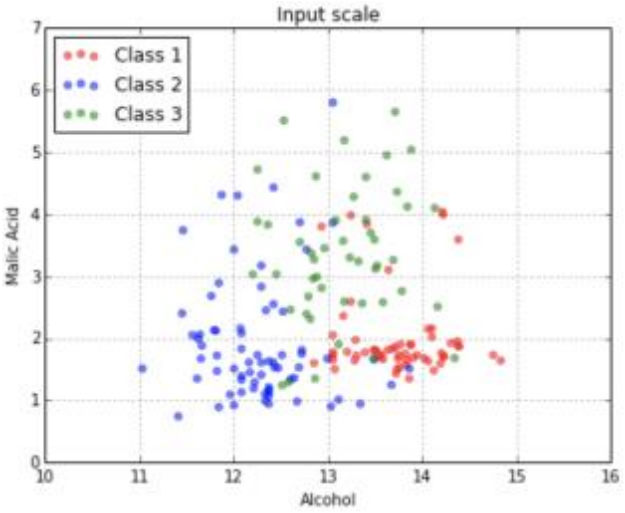
$$x' = \frac{x}{||x||}$$

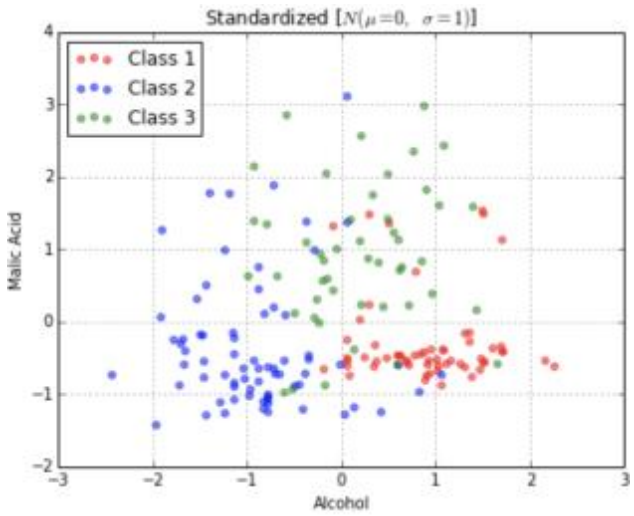
Scaling is done considering the whole feature vector to be of unit length.

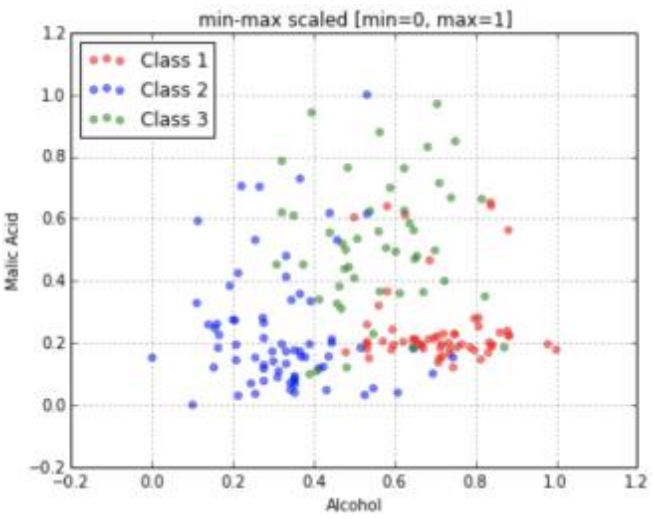
Min-Max Scaling and **Unit Vector** techniques produces values of range **[0,1]**. When dealing with features with hard boundaries this is quite useful. For example, when dealing with image data, the colors can range from only 0 to 255.



Improving Learning: Feature Scaling



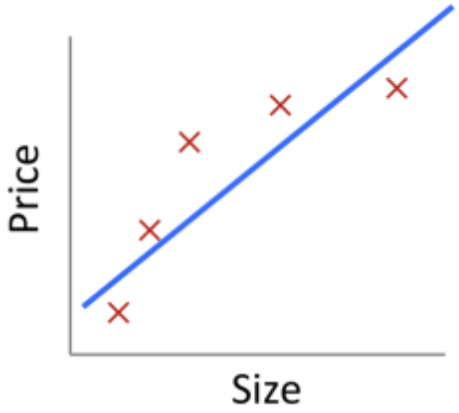




$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$$

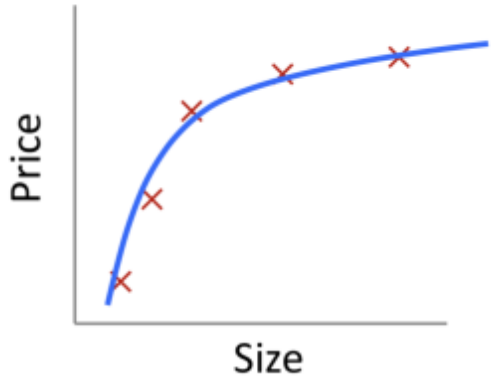
$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

Quality of Fit



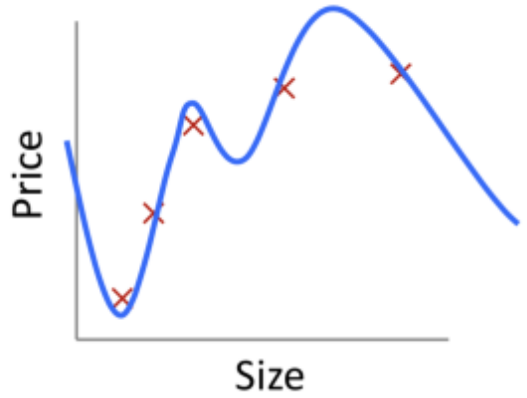
$\theta_0 + \theta_1 x$

Underfitting
(high bias)



$\theta_0 + \theta_1 x + \theta_2 x^2$

Correct fit



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

Overfitting
(high variance)

Overfitting:

- The learned hypothesis may fit the training set very well ($J(\boldsymbol{\theta}) \approx 0$)
- ...but fails to generalize to new examples

Regularization

• A method for automatically controlling the complexity of the learned hypothesis.

• **Idea**: penalize for large values of θ_j
  - Can incorporate into the cost function
  - Works well when we have a lot of features, each that contributes a bit to predicting the label.

• Can also address overfitting by eliminating features (either manually or via model selection).

Regularization

Linear regression objective function

$$J(\boldsymbol{\theta}) = \underbrace{\frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2}_{\text{model fit to data}} + \underbrace{\frac{\lambda}{2}\sum_{j=1}^{d}\theta_j^2}_{\text{regularization}}$$

- λ is the regularization parameter ($\lambda \geq 0$)
- No regularization on θ_0!

Understanding Regularization

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

- Note that $\sum_{j=1}^{d}\theta_j^2 = \|\boldsymbol{\theta}_{1:d}\|_2^2$
  - This is the magnitude of the feature coefficient vector!

- We can also think of this as:

$$\sum_{j=1}^{d}(\theta_j - 0)^2 = \|\boldsymbol{\theta}_{1:d} - \vec{\mathbf{0}}\|_2^2$$

  - L_2 regularization pulls coefficients toward 0

Understanding Regularization

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

- What happens if we set λ to be huge (e.g., 10^{10})?

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

0 0 0 0

Regularized Linear Regression

- Cost Function

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

- Fit by solving $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

- Gradient update:

$\frac{\partial}{\partial\theta_0}J(\theta)$

$$\theta_0 \leftarrow \theta_0 - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)$$

$\frac{\partial}{\partial\theta_j}J(\theta)$

$$\theta_j \leftarrow \theta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)} - \lambda\theta_j$$

regularization

Regularized Linear Regression

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

$$\theta_0 \leftarrow \theta_0 - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)$$

$$\theta_j \leftarrow \theta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)} - \lambda\theta_j$$

- We can rewrite the gradient step as:

$$\theta_j \leftarrow \theta_j\left(1 - \alpha\lambda\right) - \alpha\frac{1}{n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Summary

Linear regression

Cost function

Gradient descent

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent

Learning Rate

- With Exponential Decay

Underfitting & Overfitting

Generalization using Training, Validation and Test set

Feature scaling

- Standardization
- Mean Normalization
- Min-Max scaling
- Unit Vector

Regularization

Q&A

Thank you