1. 引言(Introduction)

1.1 Welcome

随着互联网数据不断累积，硬件不断升级迭代，在这个信息爆炸的时代，机器学习已被应用在各行各业中，可谓无处不在。

一些常见的机器学习的应用，例如：

- 手写识别
- 垃圾邮件分类
- 搜索引擎
- 图像处理
- …

使用到机器学习的一些案例：

- 数据挖掘

    - 网页点击流数据分析
- 人工无法处理的工作(量大)
    - 手写识别
    - 计算机视觉
- 个人定制
    - 推荐系统
- 研究大脑
- ……

1.2 什么是机器学习 (What is Machine Learning)

1. 机器学习定义 这里主要有两种定义：

- Arthur Samuel (1959). Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed.

  这个定义有点不正式但提出的时间最早，来自于一个懂得计算机编程的下棋菜鸟。他编写了一个程序，但没有显式地编程每一步该怎么走，而是让计算机自己和自己对弈，并不断地计算布局的好坏，来判断什么情况下获胜的概率高，从而积累经验，好似学习，最后，这个计算机程序成为了一个比他自己还厉害的棋手。

- Tom Mitchell (1998) Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some **task T** and some **performance measure P**, if its performance on T, as measured by P, improves with **experience E**.

  Tom Mitchell 的定义更为现代和正式。在过滤垃圾邮件这个例子中，电子邮件系统会根据用户对电子邮件的标记（是/不是垃圾邮件）不断学习，从而提升过滤垃圾邮件的准确率，定义中的三个字母分别代表：

    - T(Task): 过滤垃圾邮件任务。
    - P(Performance): 电子邮件系统过滤垃圾邮件的准确率。
    - E(Experience): 用户对电子邮件的标记。

2. 机器学习算法

   主要有两种机器学习的算法分类

    1. 监督学习
    2. 无监督学习

两者的区别为**是否需要人工参与数据结果的标注**。这两部分的内容占比很大，并且很重要，掌握好了可以在以后的应用中节省大把大把的时间~

还有一些算法也属于机器学习领域，诸如：

- 半监督学习: 介于监督学习于无监督学习之间
- 推荐算法: 没错，就是那些个买完某商品后还推荐同款的某购物网站所用的算法。
- 强化学习: 通过观察来学习如何做出动作，每个动作都会对环境有所影响，而环境的反馈又可以引导该学习算法。
- 迁移学习

1.3 监督学习(Supervised Learning)

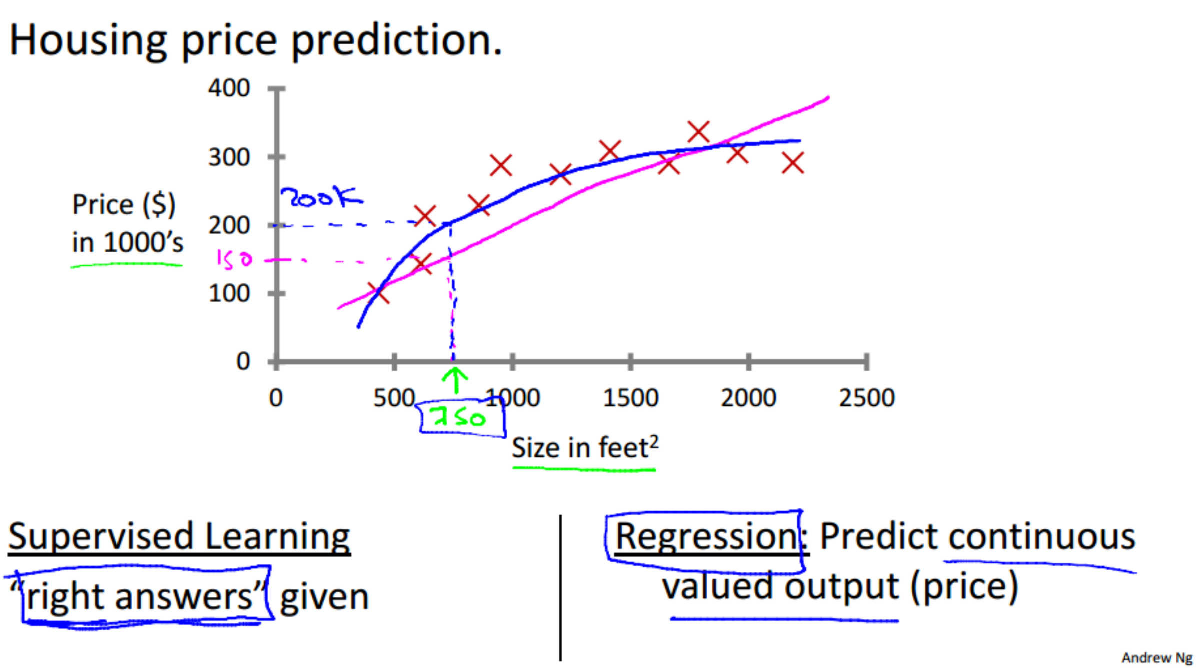
监督学习，即为教计算机如何去完成预测任务（有反馈），预先给一定数据量的输入**和对应的结果**即训练集，建模拟合，最后让计算机预测未知数据的结果。

监督学习一般有两种：

1. 回归问题(Regression)

   回归问题即为预测一系列的**连续值**。

   在房屋价格预测的例子中，给出了一系列的房屋面积数据，根据这些数据来预测任意面积的房屋价格。给出照片-年龄数据集，预测给定照片的年龄。

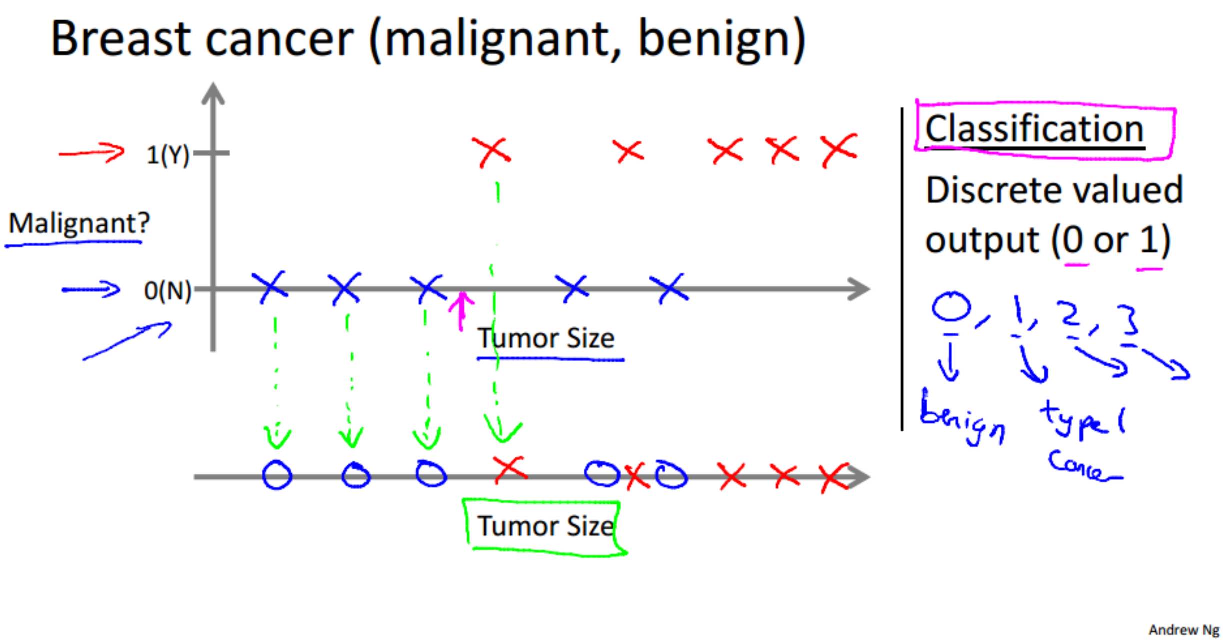


2. 分类问题(Classification)

   分类问题即为预测一系列的**离散值**。

即根据数据预测被预测对象属于哪个分类。

视频中举了癌症肿瘤这个例子，针对诊断结果，分别分类为良性或恶性。还例如垃圾邮件分类问题，也同样属于监督学习中的分类问题。



视频中提到**支持向量机**这个算法，旨在解决当特征量很大的时候(特征即如癌症例子中的肿块大小，颜色，气味等各种特征)，计算机内存一定会不够用的情况。**支持向量机能让计算机处理无限多个特征。**

1.4 无监督学习(Unsupervised Learning)

相对于监督学习，训练集不会有人为标注的结果（无反馈），我们**不会给出**结果或**无法得知**训练集的结果是什么样，而是单纯由计算机通过无监督学习算法自行分析，从而“得出结果”。计算机可能会把特定的数据集归为几个不同的簇，故叫做聚类算法。

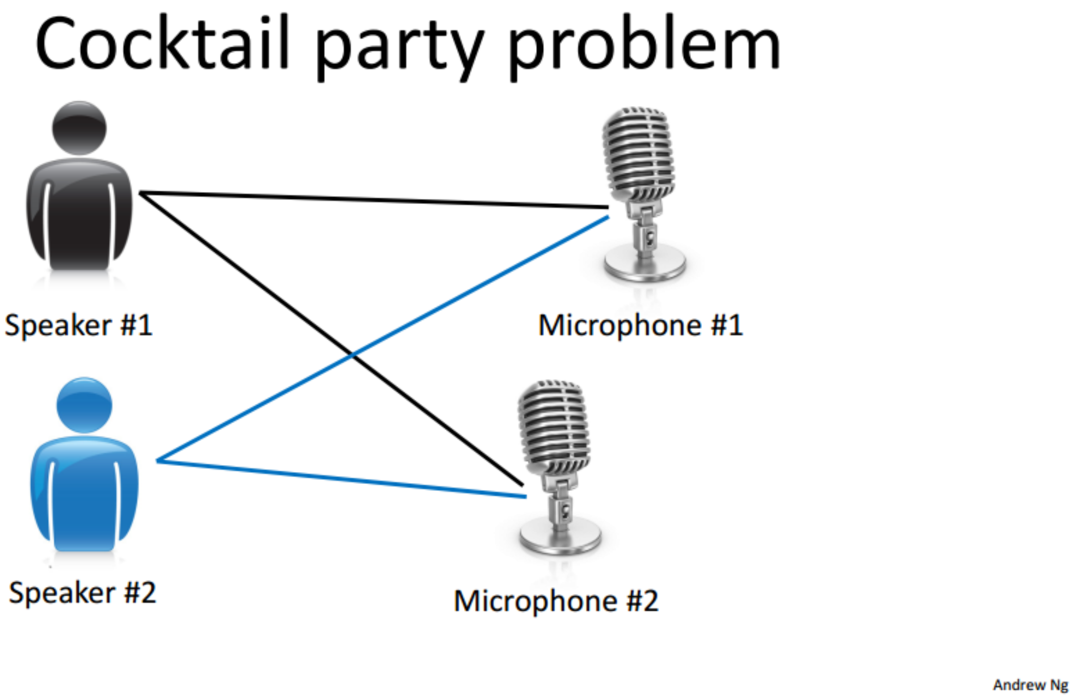
无监督学习一般分为两种：

1. 聚类(Clustering)
    - 新闻聚合
    - DNA 个体聚类
    - 天文数据分析
    - 市场细分
    - 社交网络分析
2. 非聚类(Non-clustering)
    - 鸡尾酒问题

新闻聚合

在例如谷歌新闻这样的网站中，每天后台都会收集成千上万的新闻，然后将这些新闻分组成一个个的新闻专题，这样一个又一个聚类，就是应用了无监督学习的结果。

鸡尾酒问题



在鸡尾酒会上，大家说话声音彼此重叠，几乎很难分辨出面前的人说了什么。我们很难对于这个问题进行数据标注，而这里的通过机器学习的无监督学习算法，就可以将说话者的声音同背景音乐分离出来，看视频，效果还不错呢~~。

嗯，这块是打打鸡血的，只需要一行代码就解决了问题，就是这么简单！当然，我没复现过 –……

神奇的一行代码： `[W,s,v] = svd((repmat(sum(x.*x,1),size(x,1),1).*x)*x');`

编程语言建议

在机器学习刚开始时，**推荐使用 Octave 类的工程计算编程软件**，因为在 C++ 或 Java 等编程语言中，编写对应的代码需要用到复杂的库以及要写大量的冗余代码，比较耗费时间，建议可以在学习过后再考虑使用其他语言来构建系统。 另外，在做**原型搭建**的时候也应该先考虑使用类似于 Octave 这种便于计算的编程软件，当其已经可以工作后，才将模型移植到其他的高级编程语言中。

> 注：Octave 与 MATLAB 语法相近，由于 MATLAB 为商业软件，课程中使用开源且免费的 Octave。
>
> 机器学习领域发展迅速，现在也可使用 Tensorflow 等开源机器学习框架编写机器学习代码，这些框架十分友好，易于编写及应用。

2 单变量线性回归(Linear Regression with One Variable)

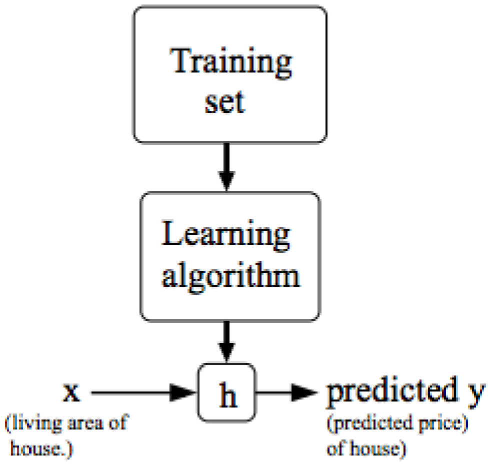
2.1 模型表示(Model Representation)

1. 房价预测训练集

Size in $feet^2$ (x)	Price ($) in 1000's($y$)
2104	460
1416	232
1534	315
852	178
...	...

房价预测训练集中，同时给出了输入 x 和输出结果 y，即给出了人为标注的”**正确结果**“，且预测的量是连续的，属于监督学习中的回归问题。

2. **问题解决模型**



其中 h 代表结果函数，也称为**假设(hypothesis)** 。假设函数根据输入(房屋的面积)，给出预测结果输出(房屋的价格)，即是一个 $X \to Y$ 的映射。

$h_\theta(x) = \theta_0 + \theta_1 x$，为解决房价问题的一种可行表达式。

> x: 特征/输入变量。

上式中，θ 为参数，θ 的变化才决定了输出结果，不同以往，这里的 x 被我们**视作已知**(不论是数据集还是预测时的输入)，所以怎样解得 θ 以更好地拟合数据，成了求解该问题的最终问题。

单变量，即只有一个特征(如例子中房屋的面积这个特征)。

2.2 代价函数(Cost Function)

> 李航《统计学习方法》一书中，损失函数与代价函数两者为**同一概念**，未作细分区别，全书没有和《深度学习》一书一样混用，而是统一使用**损失函数**来指代这类类似概念。
>
> 吴恩达(Andrew Ng)老师在其公开课中对两者做了细分。**如果要听他的课做作业，不细分这两个概念是会被打小手扣分的**！这也可能是因为老师发现了业内混用的乱象，想要治一治吧。
>
> **损失函数**(Loss/Error Function): 计算**单个**样本的误差。link
>
> **代价函数**(Cost Function): 计算整个训练集**所有损失函数之和的平均值**
>
> 综合考虑，本笔记对两者概念进行细分，若有所谬误，欢迎指正。

机器学习中的目标函数、损失函数、代价函数有什么区别？ - 知乎

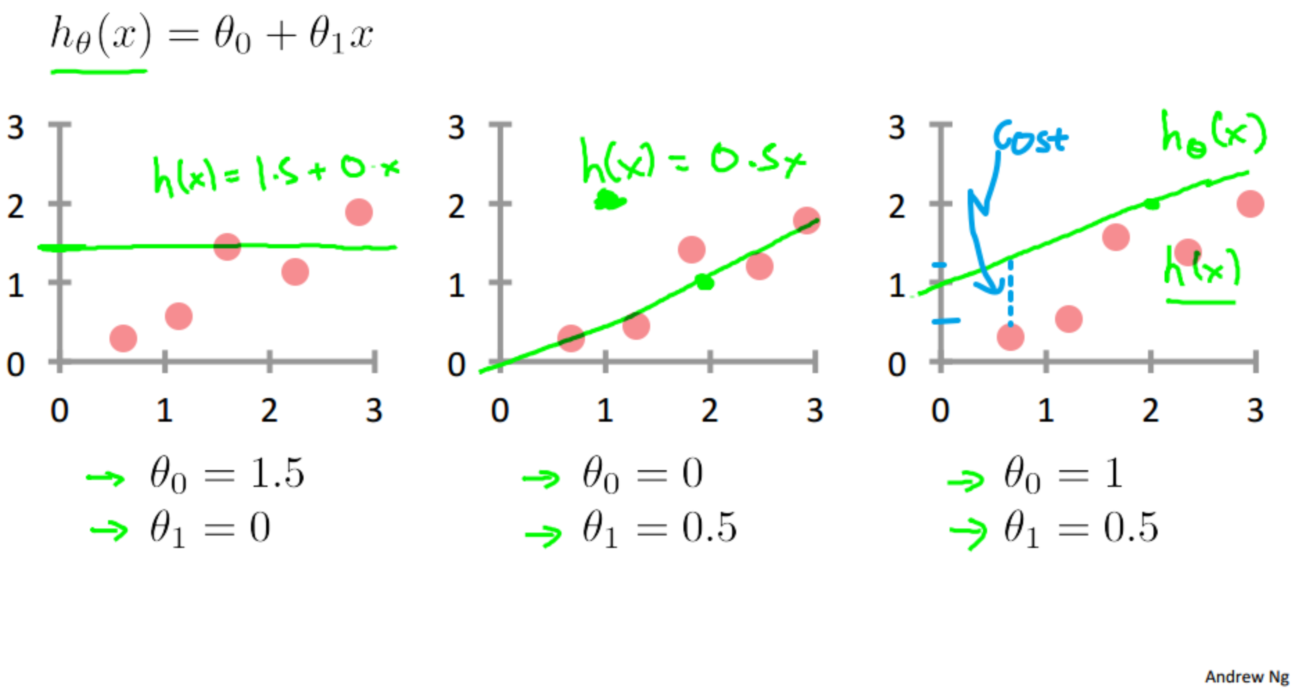
我们的目的在于求解预测结果 h 最接近于实际结果 y 时 θ 的取值，则问题可表达为**求解** $\sum_{i=0}^{m}(h_\theta(x^{(i)})-y^{(i)})$ **的最小值**。

m: 训练集中的样本总数

y: 目标变量/输出变量

(x,y): 训练集中的实例

$(x^{(i)},y^{(i)})$: 训练集中的第 i 个样本实例



上图展示了当 θ 取不同值时，$h_\theta(x)$ 对数据集的拟合情况，蓝色虚线部分代表**建模误差**（预测结果与实际结果之间的误差），我们的目标就是最小化所有误差之和。

为了求解最小值，引入代价函数(Cost Function)概念，用于度量建模误差。考虑到要计算最小值，应用二次函数对求和式建模，即应用统计学中的平方损失函数（最小二乘法）：

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

$\hat{y}$: y 的预测值

系数 $\frac{1}{2}$ 存在与否都不会影响结果，这里是为了在应用梯度下降时便于求解，平方的导数会抵消掉 $\frac{1}{2}$ 。

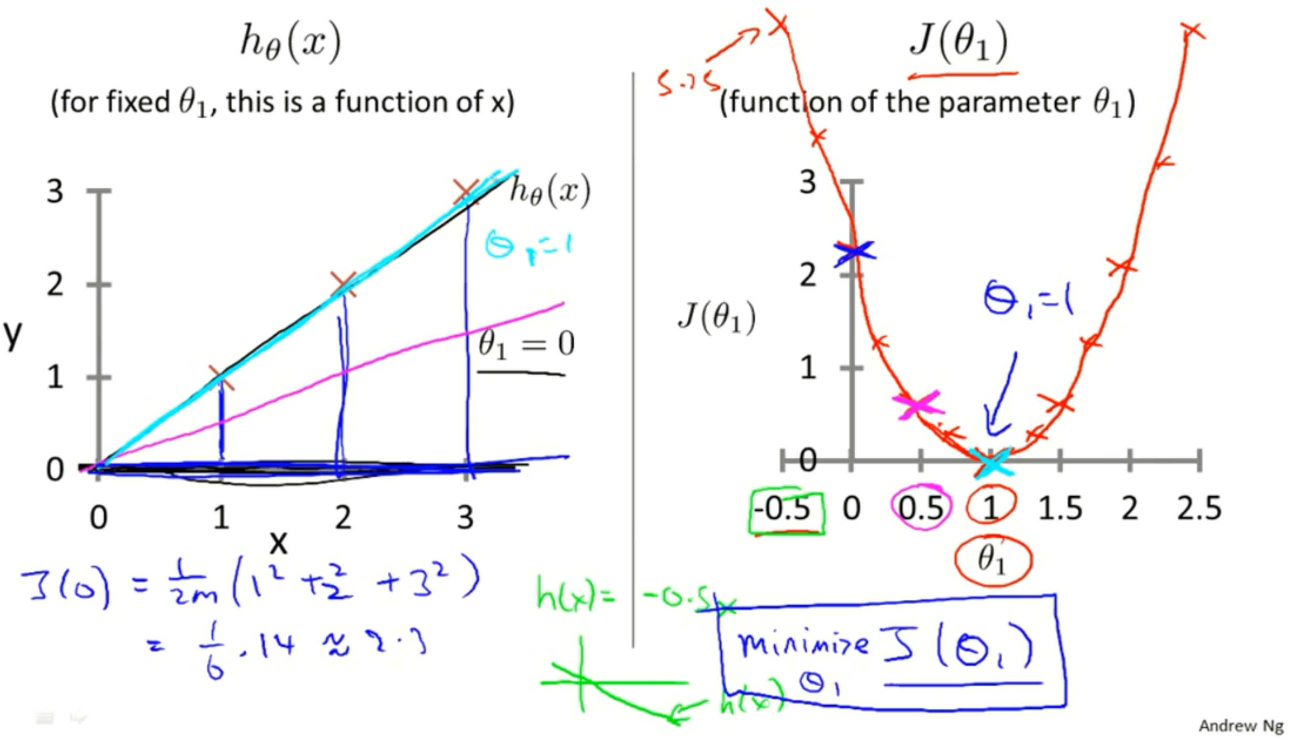
讨论到这里，我们的问题就转化成了**求解 $J\left(\theta_{0}, \theta_{1}\right)$ 的最小值**。

2.3 代价函数 - 直观理解1(Cost Function - Intuition I)

根据上节视频，列出如下定义：

- 假设函数(Hypothesis): $h_{\theta}(x)=\theta_{0}+\theta_{1} x$
- 参数(Parameters): θ_{0}, θ_{1}
- 代价函数(Cost Function): $J\left(\theta_{0}, \theta_{1}\right)=\frac{1}{2 m} \sum_{i=1}^{m}\left(h_{\theta}\left(x^{(i)}\right)-y^{(i)}\right)^{2}$
- 目标(Goal): $\underset{\theta_{0}, \theta_{1}}{\operatorname{minimize}} J\left(\theta_{0}, \theta_{1}\right)$

为了直观理解代价函数到底是在做什么，先假设 $\theta_{1}=0$，并假设训练集有三个数据，分别为$(1,1),(2,2),(3,3)$，这样在平面坐标系中绘制出 $h_{\theta}(x)$ ，并分析 $J\left(\theta_{0}, \theta_{1}\right)$ 的变化。

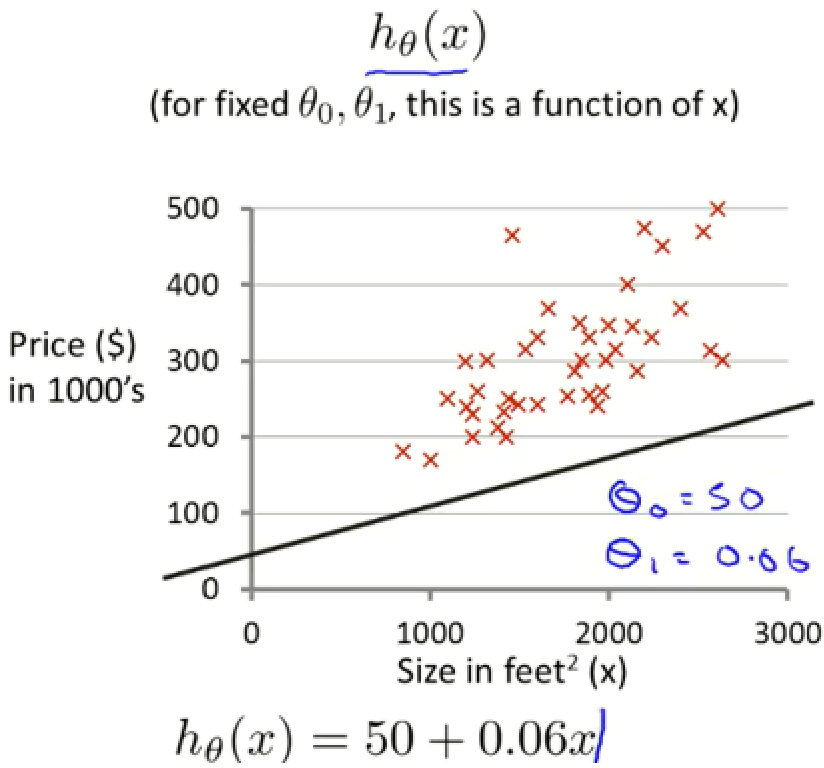


右图 $J(\theta_0, \theta_1)$ 随着 θ_1 的变化而变化，可见**当 $\theta_1 = 1$ 时，$J(\theta_0, \theta_1) = 0$，取得最小值，**对应于左图青色直线，即函数 h 拟合程度最好的情况。

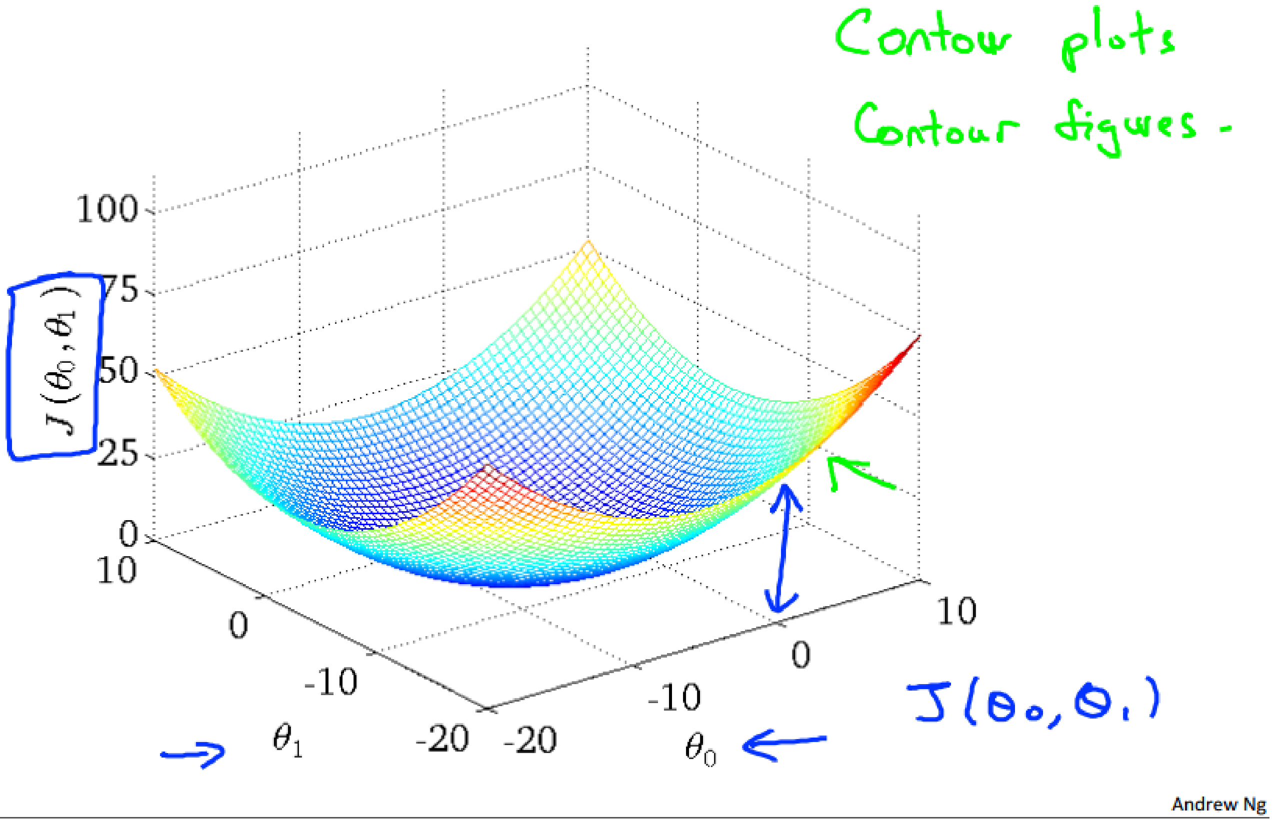
2.4 代价函数 - 直观理解2(Cost Function - Intuition II)

> 注：该部分由于涉及到了多变量成像，可能较难理解，要求只需要理解上节内容即可，该节如果不能较好理解可跳过。

给定数据集：

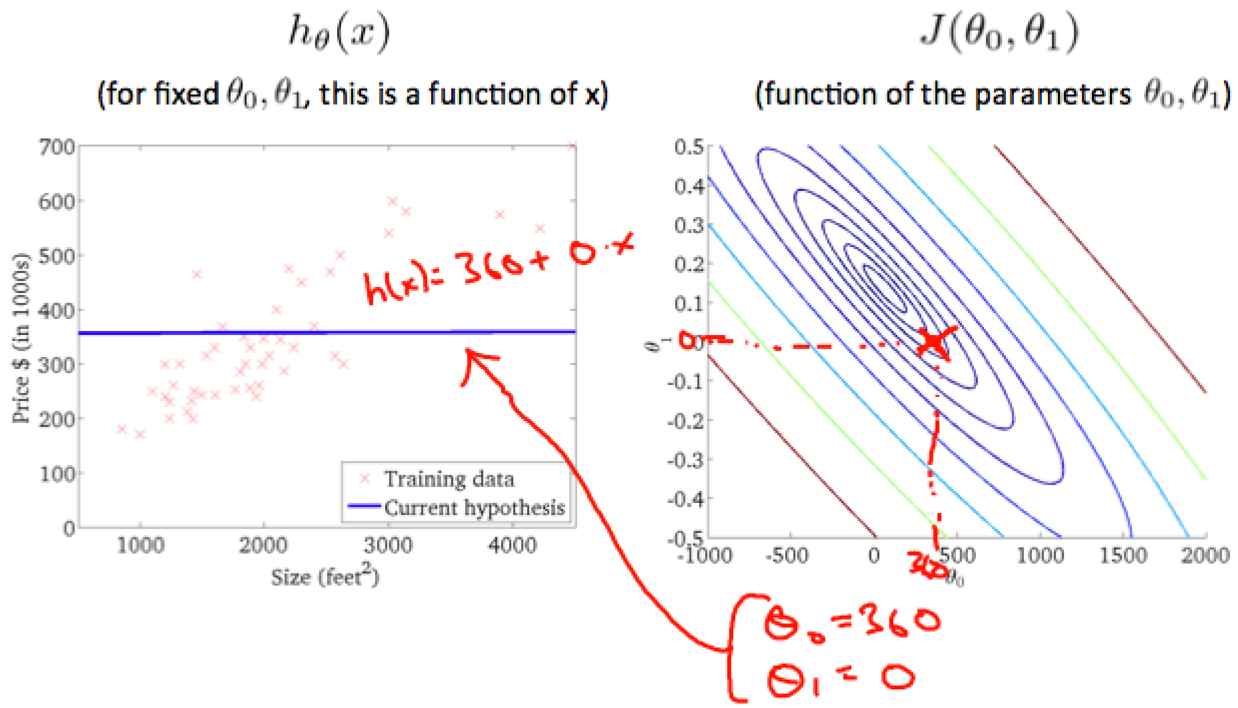


参数在 θ_0 不恒为 0 时代价函数 $J(\theta)$ 关于 θ_0, θ_1 的3-D图像，图像中的高度为代价函数的值。

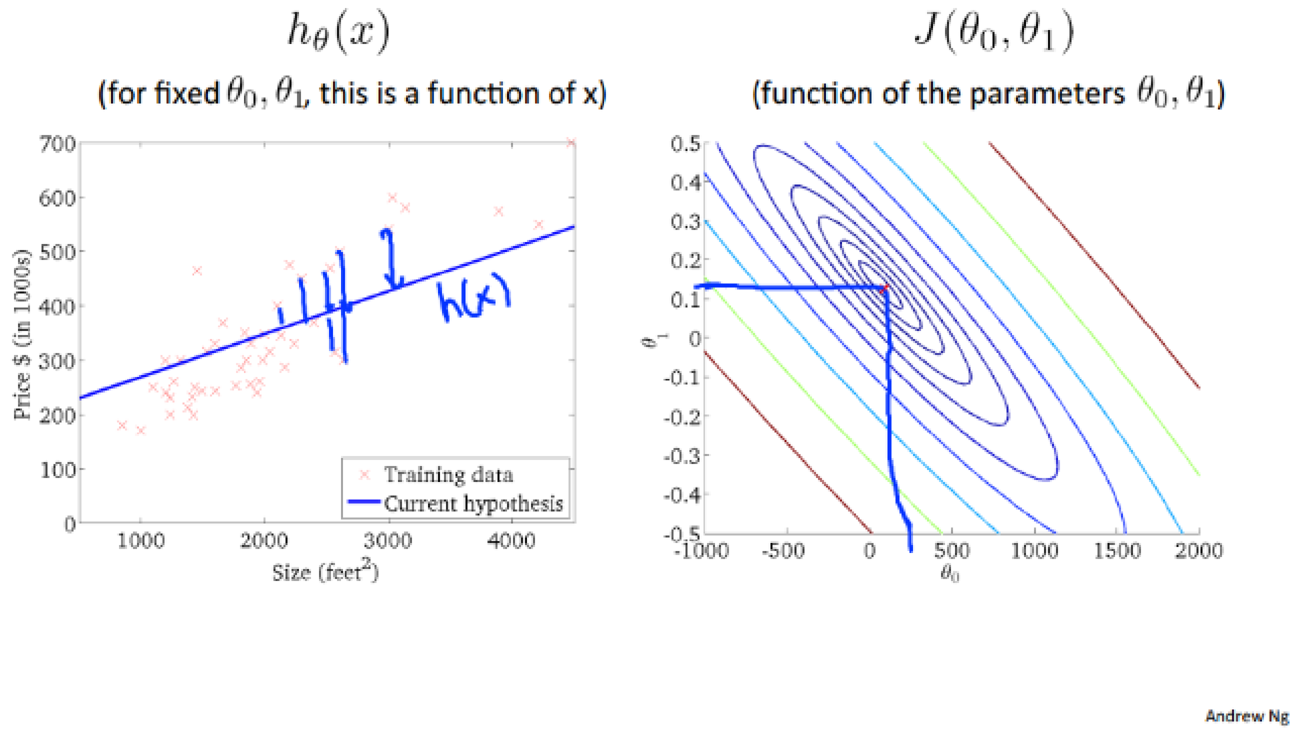


由于3-D图形不便于标注，所以将3-D图形转换为**轮廓图(contour plot)**，下面用轮廓图（下图中的右图）来作直观理解，其中相同颜色的一个圈代表着同一高度（同一 $J(\theta)$ 值）。

$\theta_0=360, \theta_1=0$ 时：



大概在 $\theta_0=0.12, \theta_1=250$ 时：



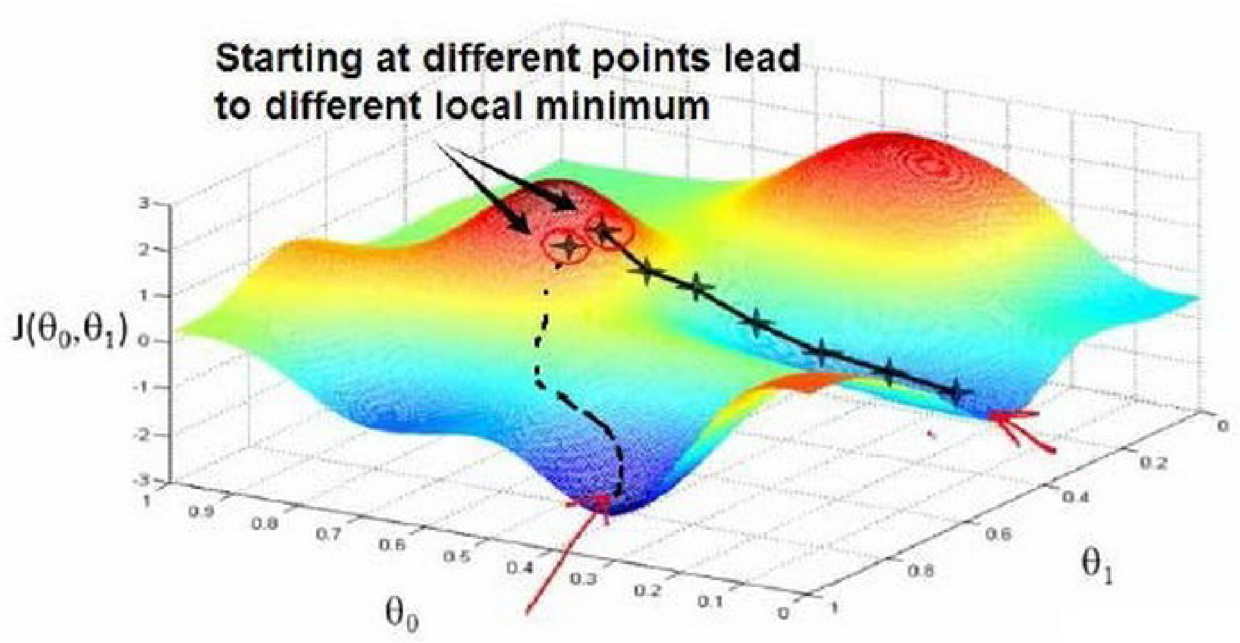
上图中最中心的点（红点），近乎为图像中的最低点，也即代价函数的最小值，此时对应 $h_\theta(x)$ 对数据的拟合情况如左图所示，嗯，一看就拟合的很不错，预测应该比较精准啦。

2.5 梯度下降(Gradient Descent)

在特征量很大的情况下，即便是借用计算机来生成图像，人工的方法也很难读出 $J(\theta)$ 的最小值，并且大多数情况无法进行可视化，故引入**梯度下降(Gradient Descent)方法，让计算机自动找出最小化代价函数时对应的 θ 值。**

梯度下降背后的思想是：开始时，我们随机选择一个参数组合$(\theta_0, \theta_1, \ldots\ldots, \theta_n)$即起始点，计算代价函数，然后寻找下一个能使得代价函数下降最多的参数组合。不断迭代，直到找到一个**局部最小值(local minimum)**，由于下降的情况只考虑当前参数组合周围的情况，所以无法确定当前的局部最小值是否就是**全局最小值(global minimum)**，不同的初始参数组合，可能会产生不同的局部最小值。

下图根据不同的起始点，产生了两个不同的局部最小值。



视频中举了下山的例子，即我们在山顶上的某个位置，为了下山，就不断地看一下周围**下一步往哪走**下山比较快，然后就**迈出那一步**，一直重复，直到我们到达山下的某一处**陆地**。

梯度下降公式：

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

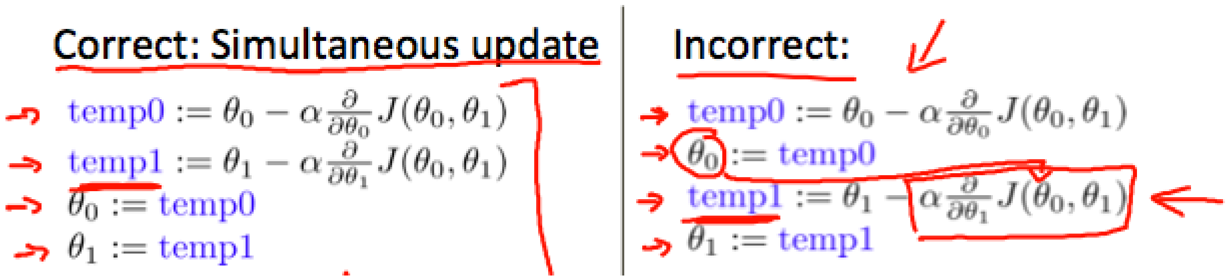
θ_j: 第 j 个特征参数

” := “: 赋值操作符

α: 学习速率(learning rate), $\alpha > 0$

$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$: $J(\theta_0, \theta_1)$ 的偏导

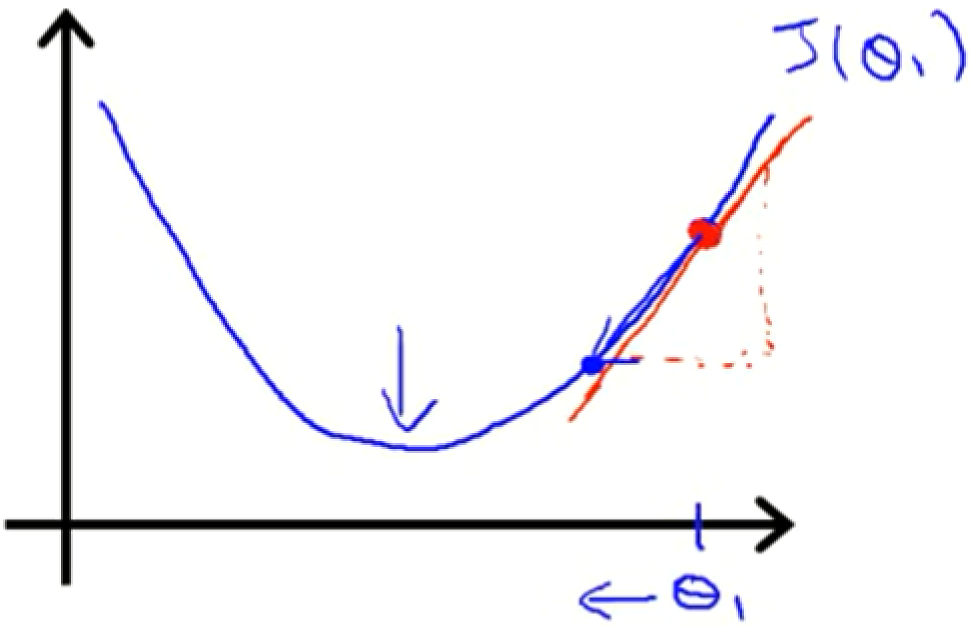
公式中，学习速率决定了参数值变化的速率即”**走多少距离**“，而偏导这部分决定了下降的方向即”**下一步往哪里**“走（当然实际上的走多少距离是由偏导值给出的，学习速率起到调整后决定的作用），收敛处的局部最小值又叫做极小值，即”**陆地**“。



注意，在计算时要**批量更新 θ 值**，即如上图中的左图所示，否则结果上会有所出入，原因不做细究。

2.6 梯度下降直观理解(Gradient Descent Intuition)

该节探讨 θ_1 的梯度下降更新过程，即 $\theta_1 := \theta_1 - \alpha\frac{d}{d\theta_1}J\left(\theta_1\right)$，此处为了数学定义上的精确性，用的是 $\frac{d}{d\theta_1}J\left(\theta_1\right)$，如果不熟悉微积分学，就把它视作之前的 $\frac{\partial}{\partial\theta}$ 即可。



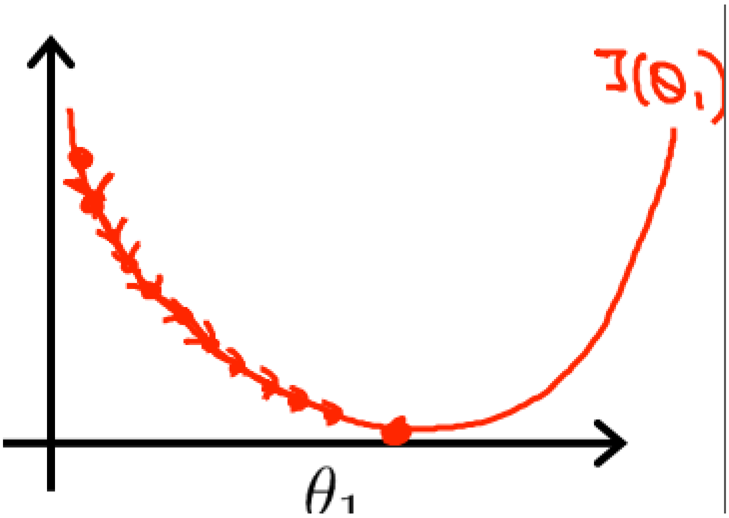
把红点定为初始点，切于初始点的红色直线的斜率，表示了函数 $J\left(\theta\right)$ 在初始点处有**正斜率**，也就是说它有**正导数**，则根据梯度下降公式，$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J\left(\theta_0, \theta_1\right)$ 右边的结果是一个正值，即 θ_1 会**向左边移动**。这样不断重复，直到收敛（达到局部最小值，即斜率为0）。

初始 θ 值（初始点）是任意的，若初始点恰好就在极小值点处，梯度下降算法将什么也不做（$\theta_1 := \theta_1 - \alpha * 0$）。

> 不熟悉斜率的话，就当斜率的值等于图中三角形的高度除以水平长度好啦，精确地求斜率的方法是求导。

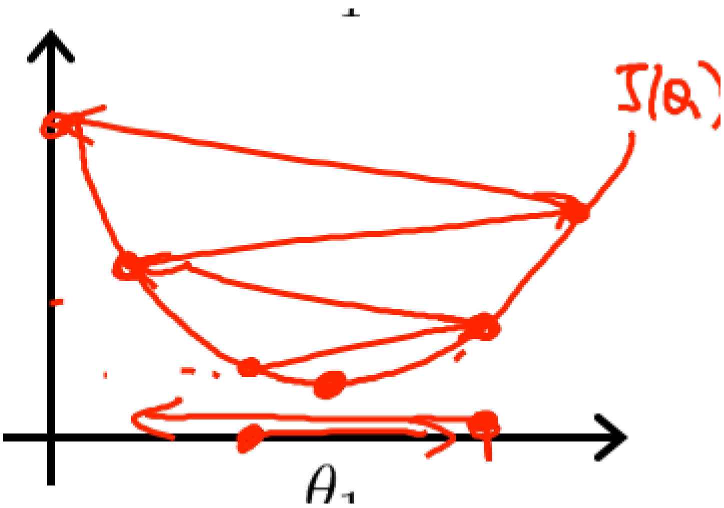
对于学习速率 α，需要选取一个合适的值才能使得梯度下降算法运行良好。

- 学习速率过小图示：



  收敛的太慢，需要更多次的迭代。

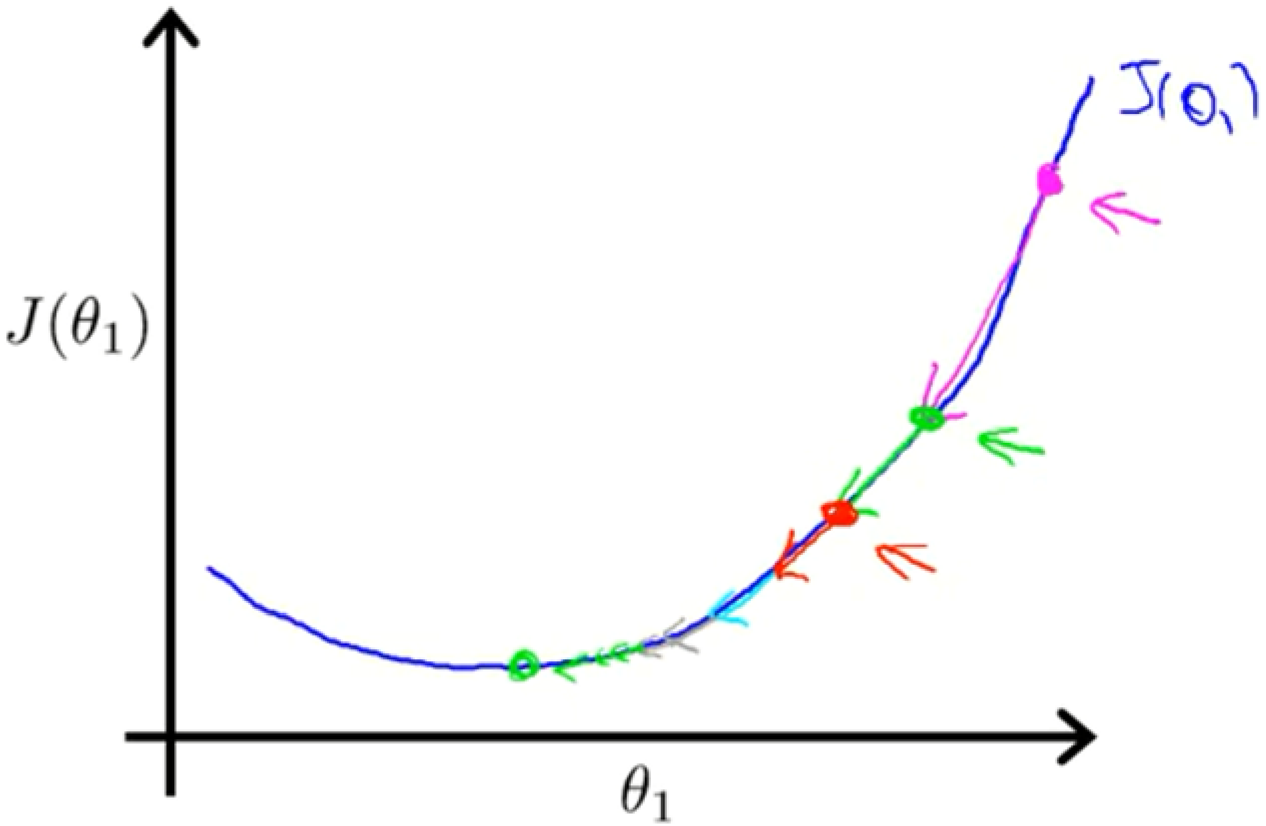
- 学习速率过大图示：



  可能越过最低点，甚至导致无法收敛。

学习速率只需选定即可，不需要在运行梯度下降算法的时候进行动态改变，随着斜率越来越接近于0，代价函数的变化幅度会越来越小，直到收敛到局部极小值。

如图，品红色点为初始点，代价函数随着迭代的进行，变化的幅度越来越小。



最后，梯度下降不止可以用于线性回归中的代价函数，还通用于最小化其他的代价函数。

2.7 线性回归中的梯度下降(Gradient Descent For Linear Regression)

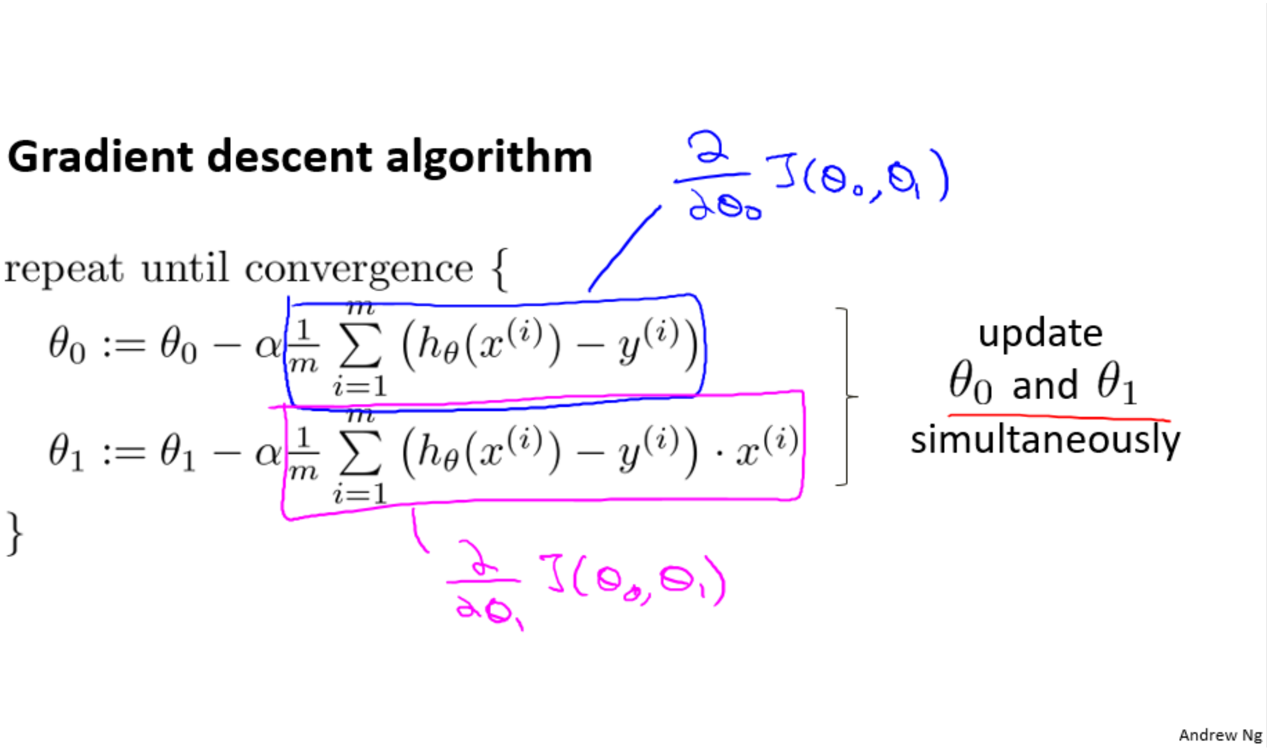
线性回归模型

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

梯度下降算法

$$
\begin{aligned}
&\text{Repeat until convergence: } \{ \\
&\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J\left(\theta_0, \theta_1\right) \\
\}
\end{aligned}
$$

直接将线性回归模型公式代入梯度下降公式可得出公式



当 $j=0, j=1$ 时，**线性回归中代价函数求导的推导过程：**

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta_1, \theta_2) &= \frac{\partial}{\partial \theta_j} \left(\frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right)^2 \right) \\
&= \left(\frac{1}{2m} * 2 \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) \right) * \frac{\partial}{\partial \theta_j} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) \\
&= \left(\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) \right) * \frac{\partial}{\partial \theta_j} \left(\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} - y^{(i)} \right)
\end{aligned}
$$

所以当 $j=0$ 时：

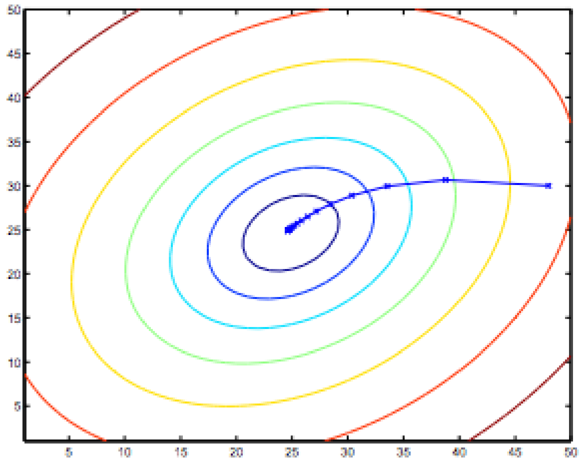
$$
\frac{\partial}{\partial \theta_0} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) * x_0^{(i)}
$$

所以当 $j=1$ 时：

$$
\frac{\partial}{\partial \theta_1} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) * x_1^{(i)}
$$

上文中所提到的梯度下降，都为批量梯度下降(Batch Gradient Descent)，即每次计算都使用**所有**的数据集 $\left(\sum_{i=1}^{m} \right)$ 更新。

由于线性回归函数呈现**碗状**，且**只有一个**全局的最优值，所以函数**一定总会**收敛到全局最小值（学习速率不可过大）。同时，函数 J 被称为**凸二次函数**，而线性回归函数求解最小值问题属于**凸函数优化问题**。



另外，使用循环求解，代码较为冗余，后面会讲到如何使用**向量化(Vectorization)**来简化代码并优化计算，使梯度下降运行的更快更好。

3 Linear Algebra Review

这部分，学过线性代数的可以复习一下，比较基础。笔记整理暂留。

3.1 Matrices and Vectors

Octave/Matlab 代码:

```
% The ; denotes we are going back to a new row.
A = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]

% Initialize a vector
v = [1;2;3]

% Get the dimension of the matrix A where m = rows and n = columns
[m,n] = size(A)

% You could also store it this way
dim_A = size(A)

% Get the dimension of the vector v
dim_v = size(v)
```

```
% Now let's index into the 2nd row 3rd column of matrix A
A_23 = A(2,3)
```

执行结果:

```
A =

    1    2    3
    4    5    6
    7    8    9
   10   11   12

v =

   1
   2
   3

m =  4
n =  3
dim_A =

   4   3

dim_v =

   3   1

A_23 =  6
```

3.2 Addition and Scalar Multiplication

Octave/Matlab 代码:

```
% Initialize matrix A and B
A = [1, 2, 4; 5, 3, 2]
```

```
B = [1, 3, 4; 1, 1, 1]

% Initialize constant s
s = 2

% See how element-wise addition works
add_AB = A + B

% See how element-wise subtraction works
sub_AB = A - B

% See how scalar multiplication works
mult_As = A * s

% Divide A by s
div_As = A / s

% What happens if we have a Matrix + scalar?
add_As = A + s
```

执行结果:

```
A =

   1   2   4
   5   3   2

B =

   1   3   4
   1   1   1

s =  2
add_AB =

   2   5   8
   6   4   3

sub_AB =
```

```
   0  -1   0
   4   2   1

mult_As =

    2    4    8
   10    6    4

div_As =

   0.50000   1.00000   2.00000
   2.50000   1.50000   1.00000

add_As =

   3   4   6
   7   5   4
```

3.3 Matrix Vector Multiplication

Octave/Matlab 代码:

```
% Initialize matrix A
A = [1, 2, 3; 4, 5, 6;7, 8, 9]

% Initialize vector v
v = [1; 1; 1]

% Multiply A * v
Av = A * v
```

执行结果:

```
```

```
A =

   1   2   3
   4   5   6
   7   8   9

v =

   1
   1
   1

Av =

    6
   15
   24
```

3.4 Matrix Matrix Multiplication

Octave/Matlab 代码:

```
% Initialize a 3 by 2 matrix
A = [1, 2; 3, 4;5, 6]

% Initialize a 2 by 1 matrix
B = [1; 2]

% We expect a resulting matrix of (3 by 2)*(2 by 1) = (3 by 1)
mult_AB = A*B

% Make sure you understand why we got that result
```

执行结果:

```
```

```
A =

   1   2
   3   4
   5   6

B =

   1
   2

mult_AB =

    5
   11
   17
```

3.5 Matrix Multiplication Properties

Octave/Matlab 代码:

```
% Initialize random matrices A and B
A = [1,2;4,5]
B = [1,1;0,2]

% Initialize a 2 by 2 identity matrix
I = eye(2)

% The above notation is the same as I = [1,0;0,1]

% What happens when we multiply I*A ?
IA = I*A

% How about A*I ?
AI = A*I

% Compute A*B
```

```
AB = A*B

% Is it equal to B*A?
BA = B*A

% Note that IA = AI but AB ≠ BA
```

执行结果:

```
A =

   1   2
   4   5

B =

   1   1
   0   2

I =

Diagonal Matrix

   1   0
   0   1

IA =

   1   2
   4   5

AI =

   1   2
   4   5

AB =

    1    5
```

```
    4   14

BA =

    5    7
    8   10
```

3.6 Inverse and Transpose

Octave/Matlab 代码:

```
% Initialize matrix A
A = [1,2,0;0,5,6;7,0,9]

% Transpose A
A_trans = A'

% Take the inverse of A
A_inv = inv(A)

% What is A^(-1)*A?
A_invA = inv(A)*A
```

执行结果:

```
A =

   1   2   0
   0   5   6
   7   0   9

A_trans =

   1   0   7
```

```
   2   5   0
   0   6   9

A_inv =

   0.348837  -0.139535   0.093023
   0.325581   0.069767  -0.046512
  -0.271318   0.108527   0.038760

A_invA =

   1.00000  -0.00000   0.00000
   0.00000   1.00000  -0.00000
  -0.00000   0.00000   1.00000
```