Lecture 2 [Linear Regression and Gradient Descent]

The process of supervised learning is that you have a [Training Set] that you feed to the learning algorithm whose job is to output a function to make predictions (a hypothesis)

When designing a learning algorithm, the first question you must ask is "how do you represent the hypothesis, H,

Linear Regression:

$H(x) = O_0 + O_1x1 + O_2x_2 + \ldots$

The hypothesis is going to have input of size x and outputs a number as a linear function of the size x

$$h(x) = \sum_{j=0}^{n} \theta_j x_j$$

where $x_0 = 1$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \quad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

θ = parameters $\{n+1$ dimensions$\}$

m = # of training examples
(# of rows in table above)

x = "inputs" / features $\{n+1$ dimensions$\}$

y = "output" / target variable

(x, y) = training example

$(x^{(i)}, y^{(i)})$ = i^{th} training example

n = # of features

How do you choose the parameters?

The learning algorithms job is to choose values for a parameter's data(theta?) so that it can output a hypothesis

In the Linear Regression algorithm, also called ordinary least squares, you'll want to minimize the squared difference between what the hypothesis outputs

Choose Θ s.t. $h(x) \approx y$ for training examples

$h_\Theta(x) = h(x)$

Linear Regression: minimize Θ

cost function

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

Batch / Stochastic Gradient Descent:

You can implement an algorithm that find the value of Θ that minimizes the cost function J(Θ)

You're going to use an algorithm called Gradient Descent

Gradient Descent

Start with some θ (say $\theta = \vec{0}$ (vector of 0's))

Keep changing θ to reduce $J(\theta)$

Gradient descent will take the steps that lead to the lowest possible value of J(Θ) (local optima)

When you run Gradient Descent on Linear Regression, it turns out that there will not be local optimum

Each step in Gradient Descent is implemented as follows:

Gradient Descent

Start with some θ (say $\theta = \vec{0}$ (vector of 0's))

Keep changing θ to reduce $J(\theta)$

$:=$ denotes assignment

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

α is the learning rate

Derivative

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_\theta(x) - y)^2$$

$$= 2 \cdot \frac{1}{2} (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x) - y)$$

Partial derivative

$$= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n - y)$$

$$= (h_\theta(x) - y) \cdot x_j$$

One step of Gradient Descent:

(Repeat until convergence)

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

(for $j = 0, 1, \ldots, n$)

Where n is the # of features

When you plot the cost function of J(Θ) for a Linear Regression model, J(Θ) will be a quadratic function (3D graphically looks like a bowl)

If you think about the contours of the function, it turns out that the direction of steepest descent is always at 90 degrees, it's always orthogonal to the contour direction

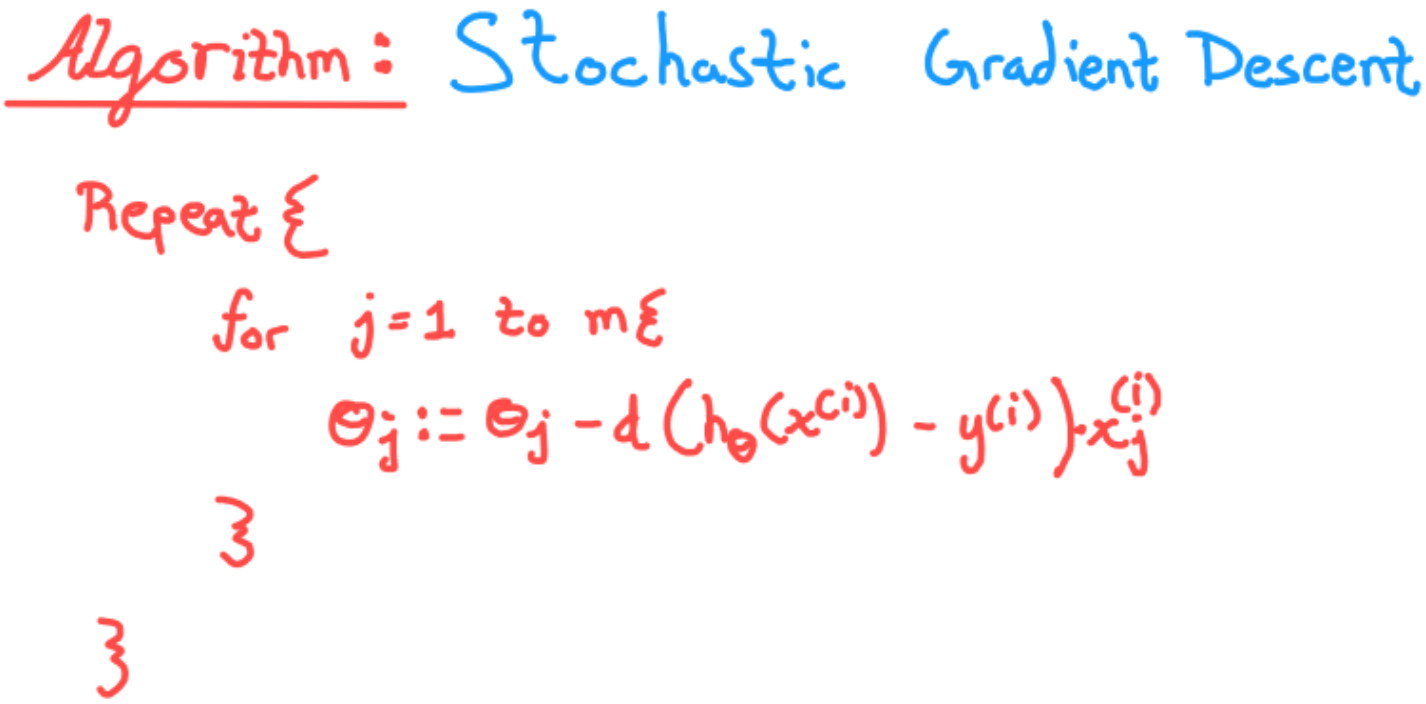
This algorithm on the cost function of J(Θ) has only one global minimum and will always converge to the global minimum

Is an algorithm for fitting linear regression models

The Gradient Descent algorithm above calculates the derivate by summing over your entire training set, m, and sometimes this version of Gradient Descent has another name which is **Batch Gradient Descent**

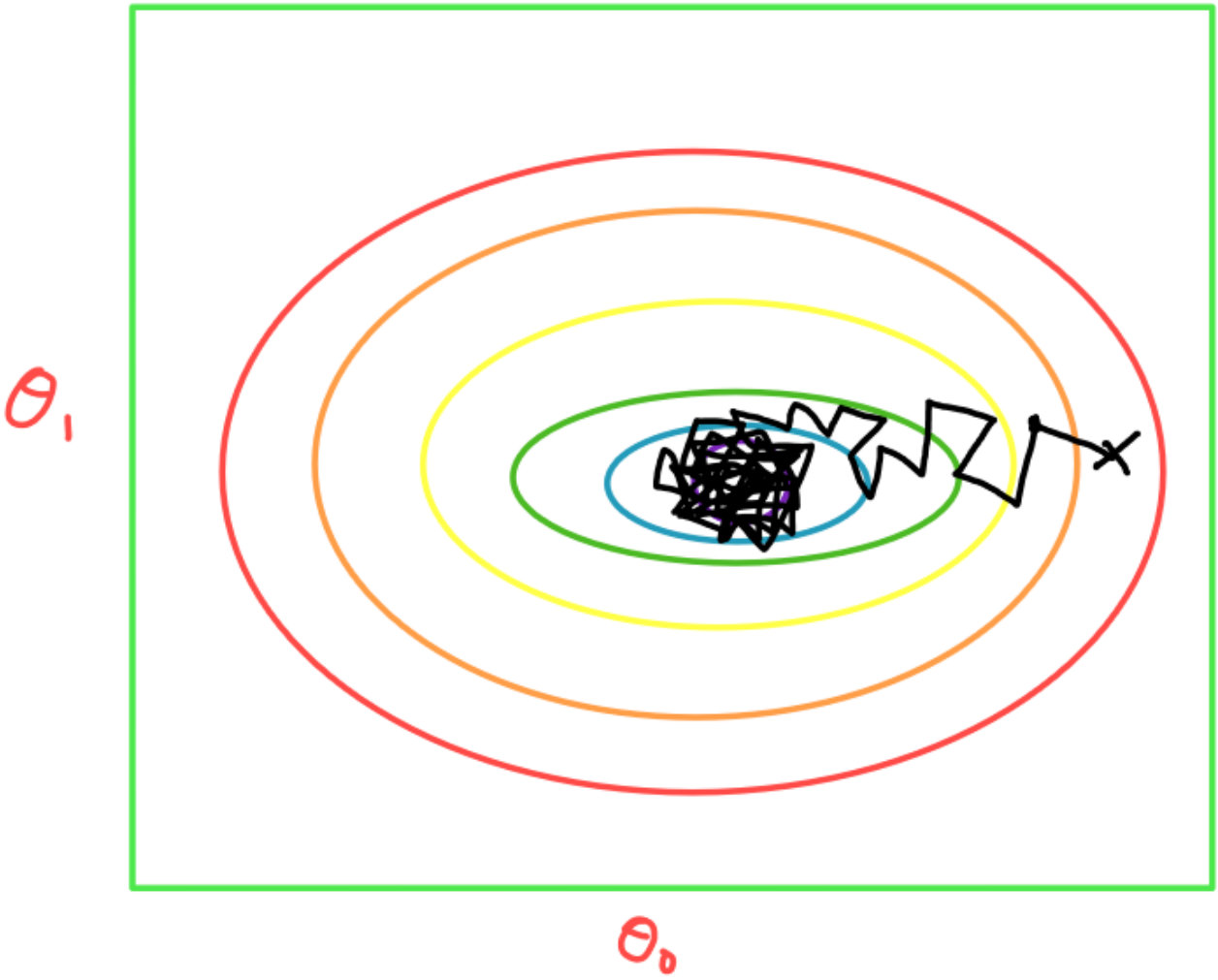
The disadvantage of Batch Gradient Descent is that if you have a giant dataset, in order to make one update to your parameters, in order to take even a single step of Gradient Descent, you need to scan through your entire dataset and calculate the sum

There's an alternative to Batch Gradient Descent:



Instead of scanning through all examples before you update the parameters theta even a little bit, in **Stochastic Gradient Descent**, instead in the inner loop of the algorithm, you loop through j = 1 to m of taking a gradient descent step using the derivative of just one

single example



As you run Stochastic Gradient Descent, it takes a slightly noisy, slightly random path, but on average is headed toward the global minimum. It will never quite converge

When you have a very large dataset, it allows your implementation/algorithm to make much faster progress

When you have very large datasets, stochastic gradient descent is used much more in practice than batch gradient descent

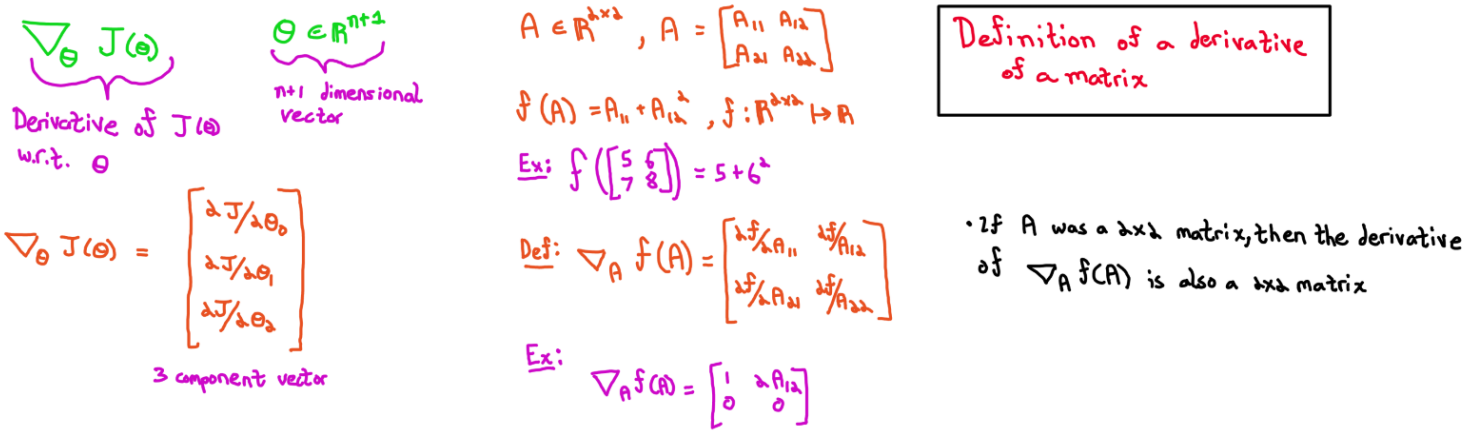
Gradient Descent is an iterative algorithm

With Linear Regression specifically, there's a way to solve for the optimal value of the parameter's theta to just jump in one step to the global optimum without needing to use an iterative algorithm

Normal Equation:

It works only for Linear Regression

Matrix Derivation definition:



Derivation of the Normal Equation

Cost function: $J(\theta)$

$\nabla_\theta J(\theta) \overset{\text{set}}{=} \vec{0}$

minimize function

- then solve for θ

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

Design Matrix

$$X\theta = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}\theta \quad \text{(column vector)} \quad = \begin{bmatrix} x^{(1)T}\theta \\ x^{(2)T}\theta \\ \vdots \\ x^{(m)T}\theta \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

$$= \begin{bmatrix} h_\theta(x^{(1)}) \\ \vdots \\ h_\theta(x^{(m)}) \end{bmatrix} \quad \text{vector of all predictions of the algorithm}$$

$\leftarrow X\theta$

$$X\theta - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

contains all the errors (difference between predictions and actual labels) that your learning algorithm makes under m examples

If A is square matrix ($A \in \mathbb{R}^{n\times n}$)

$\text{tr}\, A$ = sum of diagonal entries = $\sum_i A_{ii}$

"$\text{tr}(A)$" "trace of A"

$\text{tr}\, A = \text{tr}\, A^T$

useful properties

- If, $f(A) = \text{tr}\, AB$, B is a fixed matrix

Then, $\nabla_A f(A) = B^T$

$\text{tr}\, AB = \text{tr}\, BA$

- $\text{tr}\, ABC = \text{tr}\, CAB$

- $\nabla_A \text{tr}\, AA^TC = CA + C^TA$

$\approx \frac{d}{da} a^2c = 2ac$

$$\nabla_\theta J(\theta) = \nabla_\theta \frac{1}{2}(X\theta - y)^T(X\theta - y) = \frac{1}{2}\nabla_\theta\left(\theta^TX^T - y^T\right)(X\theta - y)$$

$$= \frac{1}{2}\nabla_\theta\left[\theta^TX^TX\theta - \theta^TX^Ty - y^TX\theta + y^Ty\right]$$

- $(ax-b)(ax-b) = a^2x^2 - axb - bax + b^2$

$$= \frac{1}{2}\left[X^TX\theta + X^TX\theta - X^Ty - X^Ty\right]$$

$$= X^TX\theta - X^Ty \overset{\text{set}}{=} \vec{0}$$

$$X^TX\theta = X^Ty \qquad \text{"Normal equation"}$$

$$\theta = (X^TX)^{-1}X^Ty$$

What if X is non-invertable?

That usually means that you have redundant features, that your features are linearly dependent, but if you use something called a pseudo-inverse you can kind of get the right answer. If you have linearly dependent features, it probably means you have the same feature repeated twice