UVA CS 4774: Machine Learning

Lecture 4: More optimization for Linear Regression

Dr. Yanjun Qi

University of Virginia

Department of Computer Science

Rough Sectioning of this Course

- 1. Basic Supervised Regression + on Tabular Data
- 2. Basic Deep Learning + on 2D Imaging Data
- 3. Advanced Supervised learning + on Tabular Data
- 4. Generative and Deep + on 1D Sequence Text Data
- 5. Not Supervised + Mostly on Tabular Data

Today : GD and SGD for Multivariate Linear Regression

Data: X

↓

Task: y

↓

Representation: : x, f()

↓

Score Function: L()

↓

Search/Optimization : argmin()

↓

Models, Parameters

→

X: Tabular

↓

Regression: y continuous

↓ $\hat{y} = f(x) = \theta^T x$

Y = Weighted linear sum of Xs

↓

Sum of Squared Error (Least Squared)

↓

Normal Equation / GD / SGD ←

↓

Metrics, Implementation, Regression coefficients w, b

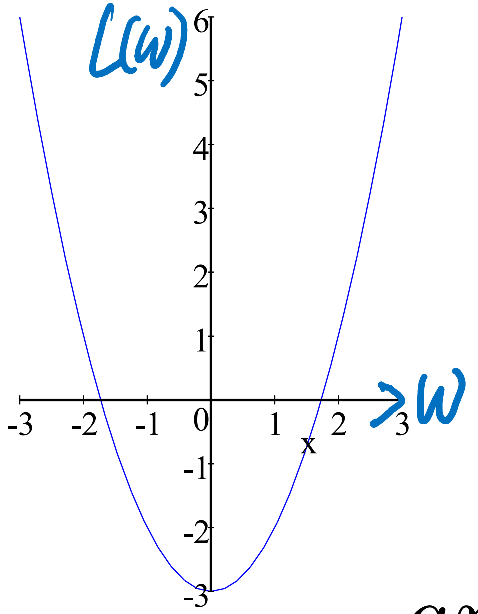
A little bit more about [Optimization]

- Objective function $F(x)$ $\rightarrow J(\theta)$
- Variables x $\rightarrow \theta$
- Constraints $\rightarrow \theta \in R^p$

To find values of the variables
that minimize or maximize the objective function
while satisfying the constraints

Method 1: Directly Optimize

Minimizing a Quadratic Function



$$L(w) = w^2 - 3$$

$$L'(w) = 2w$$

$$argmin_w L(w)$$

This quadratic (convex) function is minimized @ the unique point whose derivative (slope) is zero.

→ When we find zeros of the derivative of this function, we also find the minima (or maxima) of that function.

Method I: normal equations to solve for LR training

- Write the cost function in matrix form:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i{}^T\theta - y_i)^2$$

$$= \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

$$= \frac{1}{2}(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T \vec{y})$$

$$\mathbf{X}_{train} = \begin{bmatrix} -- & \mathbf{x}_1^T & -- \\ -- & \mathbf{x}_2^T & -- \\ \vdots & \vdots & \vdots \\ -- & \mathbf{x}_n^T & -- \end{bmatrix} \qquad \vec{y}_{train} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

To minimize J(θ), take its gradient and set to zero:

$$\nabla_\theta J(\theta) = 0 \Rightarrow \boxed{X^T X\theta = X^T\vec{y}}$$

The normal equations

$$\Downarrow$$

$$\theta^* = \left(X^T X\right)^{-1} X^T \vec{y}$$

$X^T X$ Gram matrix

↓

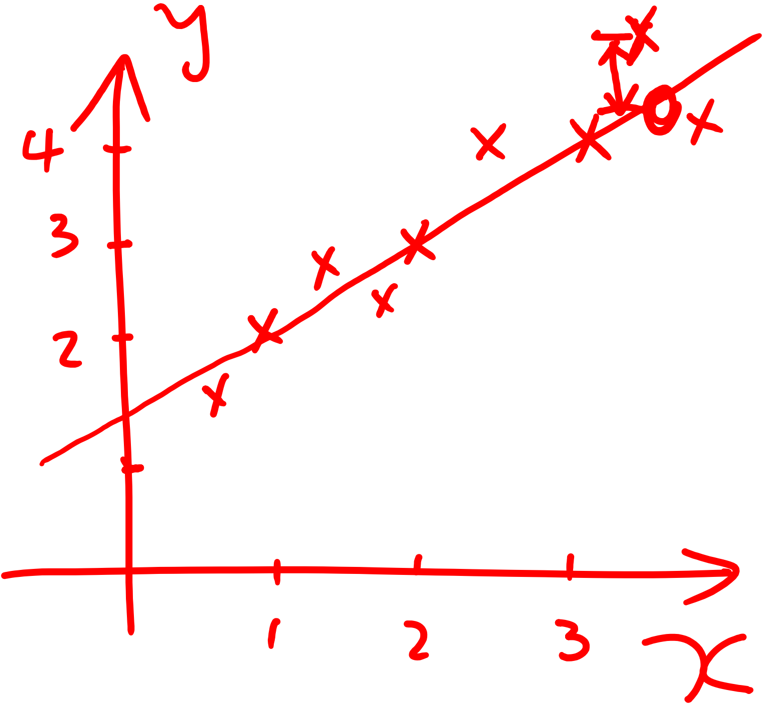
PSD

↓

$J(\theta)$ convex

↓

$\nabla J(\theta) = 0 \Rightarrow \theta^*$



x	y	$f(x) = wx + b$
1	2	$f(1) = w + b$
2	3	$f(2) = 2w + b$
3	4	$f(3) = 3w + b$

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i{}^T\theta - y_i)^2 =$$

$$\text{loss}(w, b) = (w+b-2)^2 + (2w+b-3)^2 + (3w+b-4)^2 \Rightarrow (w, b) = \underset{w,b}{\operatorname{argmin}}\ \text{loss}()$$

$$\text{loss}(w,b) = J(w,b) = J(\theta_1, \theta_0)$$

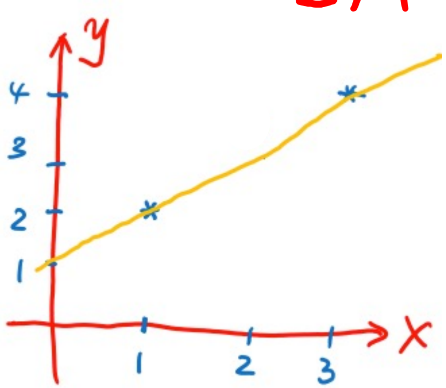
One concrete example

$$\Rightarrow \quad J(w,b) = \frac{1}{2}\left((w+b-2)^2 + (2w+b-3)^2\right)$$

$$\Rightarrow \begin{cases} \dfrac{\partial J(w,b)}{\partial w} = (w+b-2) + (2w+b-3)\cdot 2 = 0 \\[2ex] \dfrac{\partial J(w,b)}{\partial b} = w+b-2 + (2w+b-3) = 0 \end{cases}$$

2x1 Vector

$$\begin{cases} 5w + 3b - 8 = 0 \\ 3w + 2b - 5 = 0 \end{cases} \Rightarrow \begin{cases} 10w + 6b - 16 = 0 \\ 9w + 6b - 15 = 0 \end{cases}$$



$$\Rightarrow w = 1, \quad b = \frac{1}{2}(5 - 3w) = 1$$

we solve the matrix equation via Gaussian Elimination

Method 2: Iteratively Optimize via Gradient Descent

Gradient Descent (GD): An iterative Algorithm

$$F(x)$$

- Initialize k=0, (randomly or by prior) choose x_0

- While $k<k_{max}$

For the k-th epoch

$$x_k = x_{k-1} - \alpha \nabla_x F(x_{k-1})$$

Gradient Descent (GD): An iterative Algorithm

$$F(x)$$

- Initialize k=0, (randomly or by prior) choose x_0

- While $k<k_{max}$

For the k-th epoch

$$x_k = x_{k-1} - \alpha \nabla_x F(x_{k-1})$$

α: learning rate / defined by users

$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots \rightarrow x_T$

epoch

Review: Definitions of gradient (more in Algebra-note)

- Size of gradient vector is always the same as the size of the variable vector

$$\nabla_{\boldsymbol{x}} F(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial F(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial F(\boldsymbol{x})}{\partial x_2} \\ \vdots \\ \frac{\partial F(\boldsymbol{x})}{\partial x_p} \end{bmatrix} \in \mathbb{R}^p$$

if $\boldsymbol{x} \in \mathbb{R}^p$

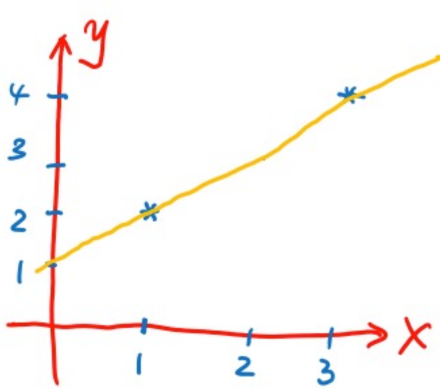
A vector whose entries, respectively, contain the p partial derivatives

Our concrete example

$$\Rightarrow \quad J(w,b) = \frac{1}{2}\left((w+b-2)^2 + (2w+b-3)^2\right)$$

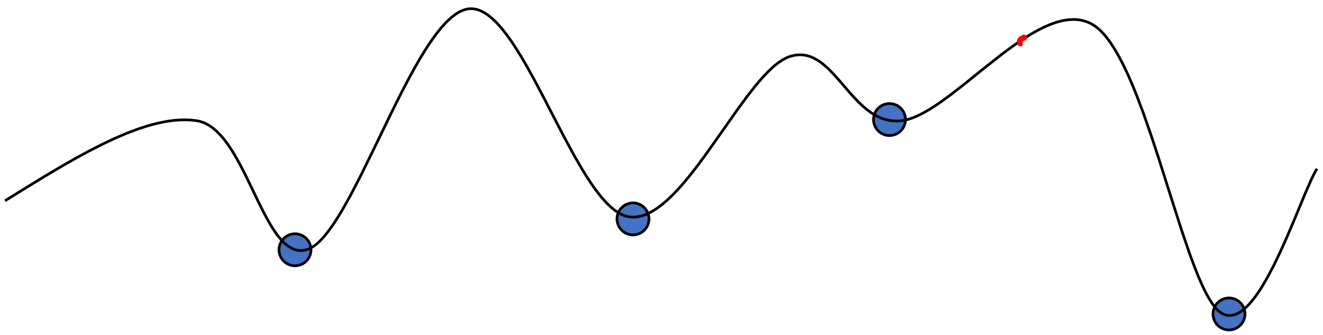
$$\Rightarrow \begin{cases} \dfrac{\partial J(w,b)}{\partial w} = (w+b-2) + (2w+b-3)\cdot 2 = 0 \\[2ex] \dfrac{\partial J(w,b)}{\partial b} = w+b-2 + (2w+b-3) = 0 \end{cases}$$

$$\begin{cases} \nabla_{\vec{\theta}} J(\vec{\theta}) = \begin{bmatrix} \dfrac{\partial J(\vec{\theta})}{\partial \theta_1} \\[2ex] \dfrac{\partial J(\vec{\theta})}{\partial \theta_0} \end{bmatrix} \\[4ex] \vec{\theta} = \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \end{cases}$$

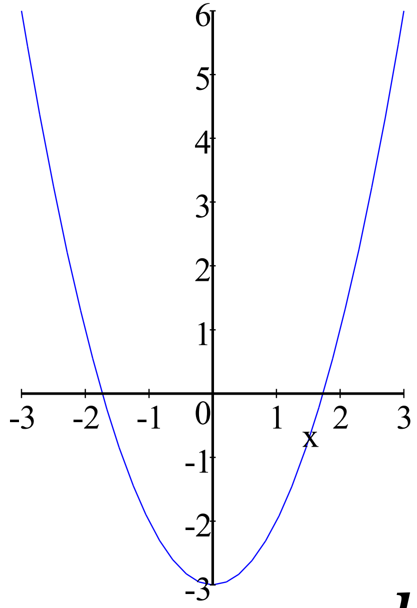


WHY ? Optimize through Gradient Descent (iterative) Algorithms

- Works on any objective function
  - as long as we can evaluate the gradient
  - this can be very useful for minimizing complex functions



Review: Definitions of derivative (single variable case)



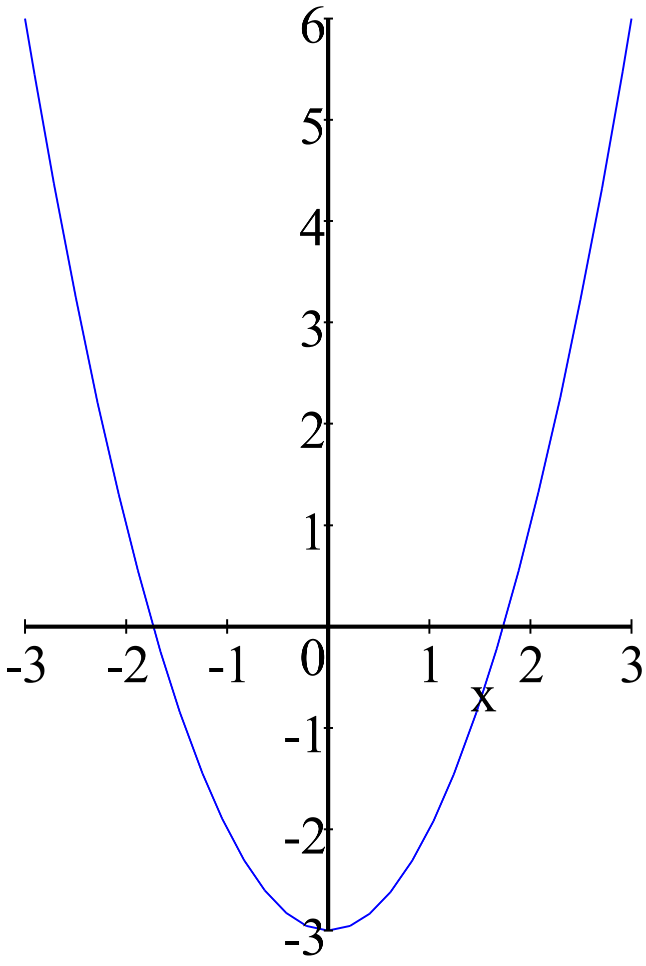
Review: Derivative of a Quadratic Function

$$l(w) = w^2 - 3$$

$$l'(w) = \lim_{h \to 0} \frac{(w+h)^2 - 3 - (w^2 - 3)}{h} = 2w$$

The derivative is often described as the "instantaneous rate of change",
➔ the ratio of the instantaneous change in F(x) to change in x

Our concrete example with Gradient Descent

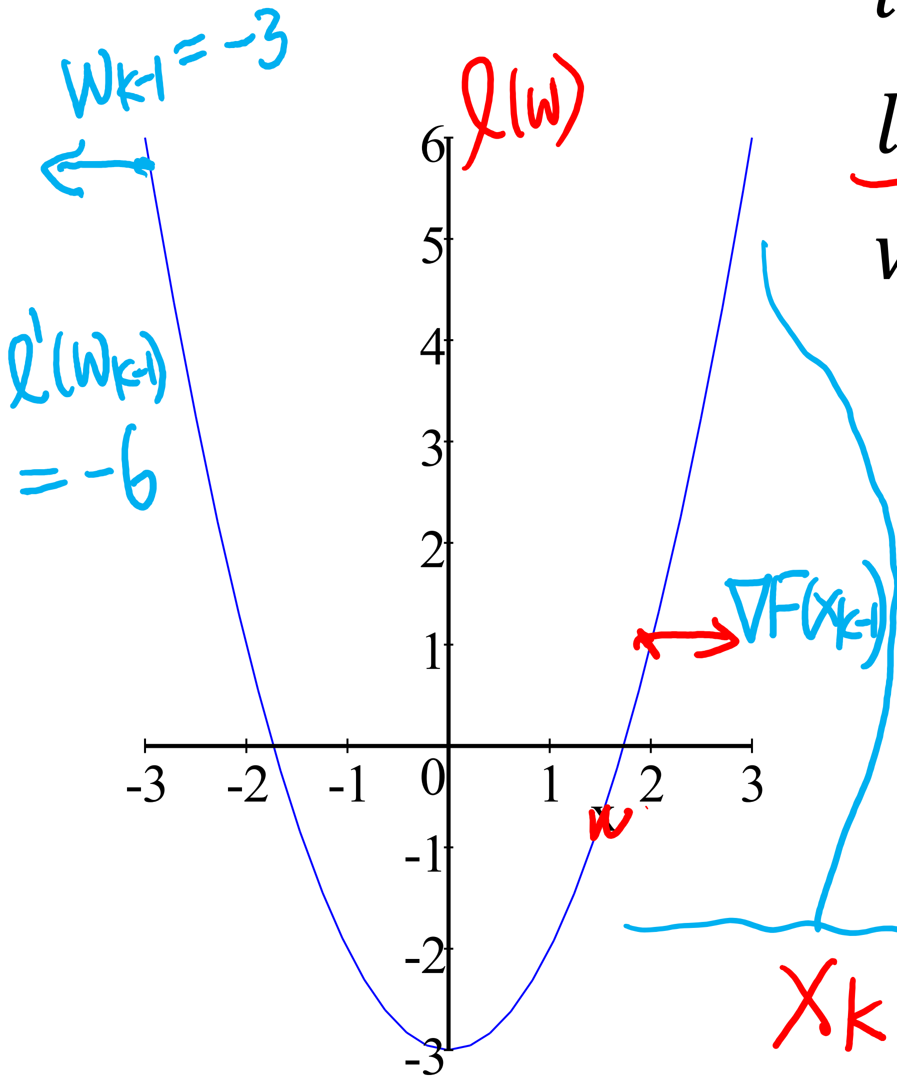


$$l(w) = w^2 - 3$$

$$l'(w) = 2w$$

$$w_k = w_{k-1} - 2 \propto w_{k-1}$$

Our concrete example with Gradient Descent



$$l(w) = w^2 - 3$$

$$l'(w) = 2w$$

$$w_k = w_{k-1} - 2 \propto w_{k-1}$$

$w_{k-1} = 2$, $\alpha = 0.1$

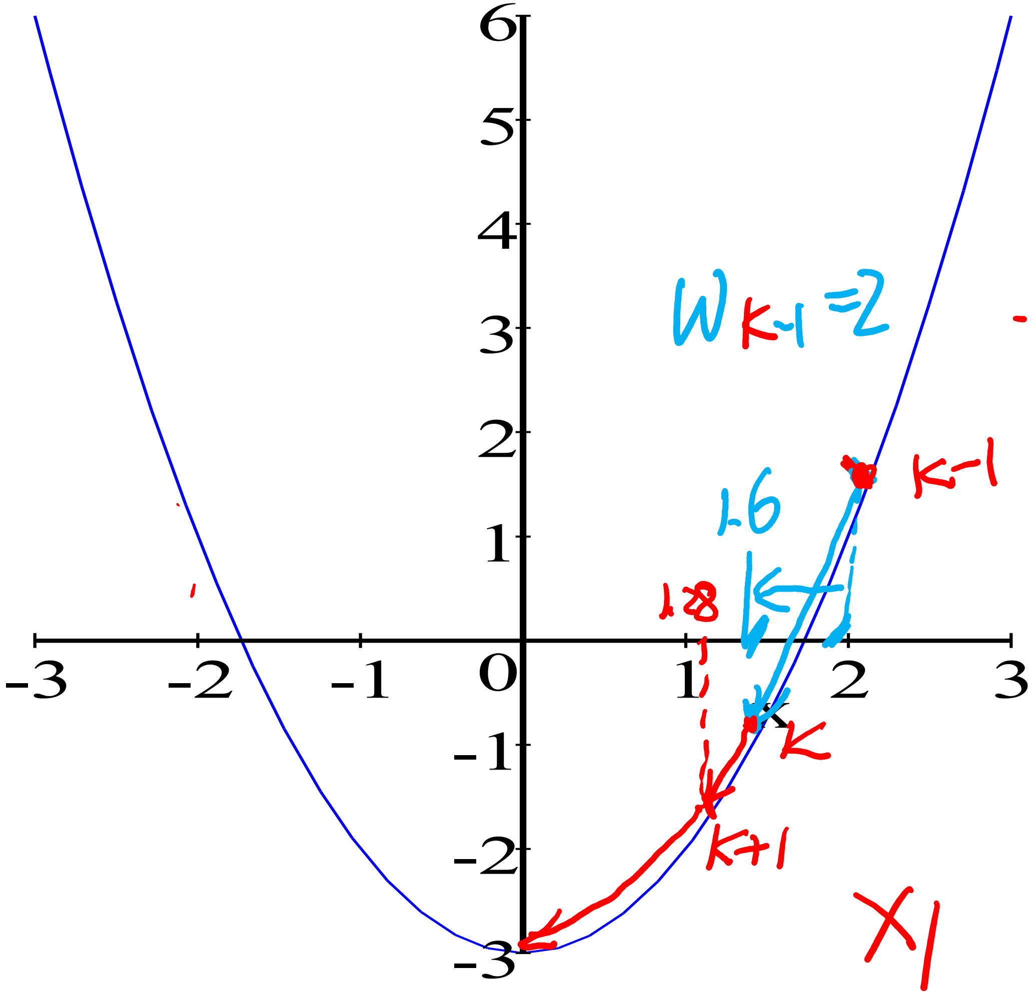
$l'(w_{k-1}) = 2w_{k-1} = 4$

$w_k = 2 - 2 \times (0.1) \times 2$

$= 1.6$

$X_k = X_{k-1} - \alpha \nabla_x F(X_{k-1})$

$$w_k = w_{k-1} - 2\alpha w_{k-1}$$



$W_{k-1} = 2$

$\alpha = 0.1$

$W_k = 2 - 2 \times 0.1 \times 2$

$= 1.6$

$W_{k+1} = 1.6 -$

$2 \times 0.1 \times 1.6$

$= 1.28$

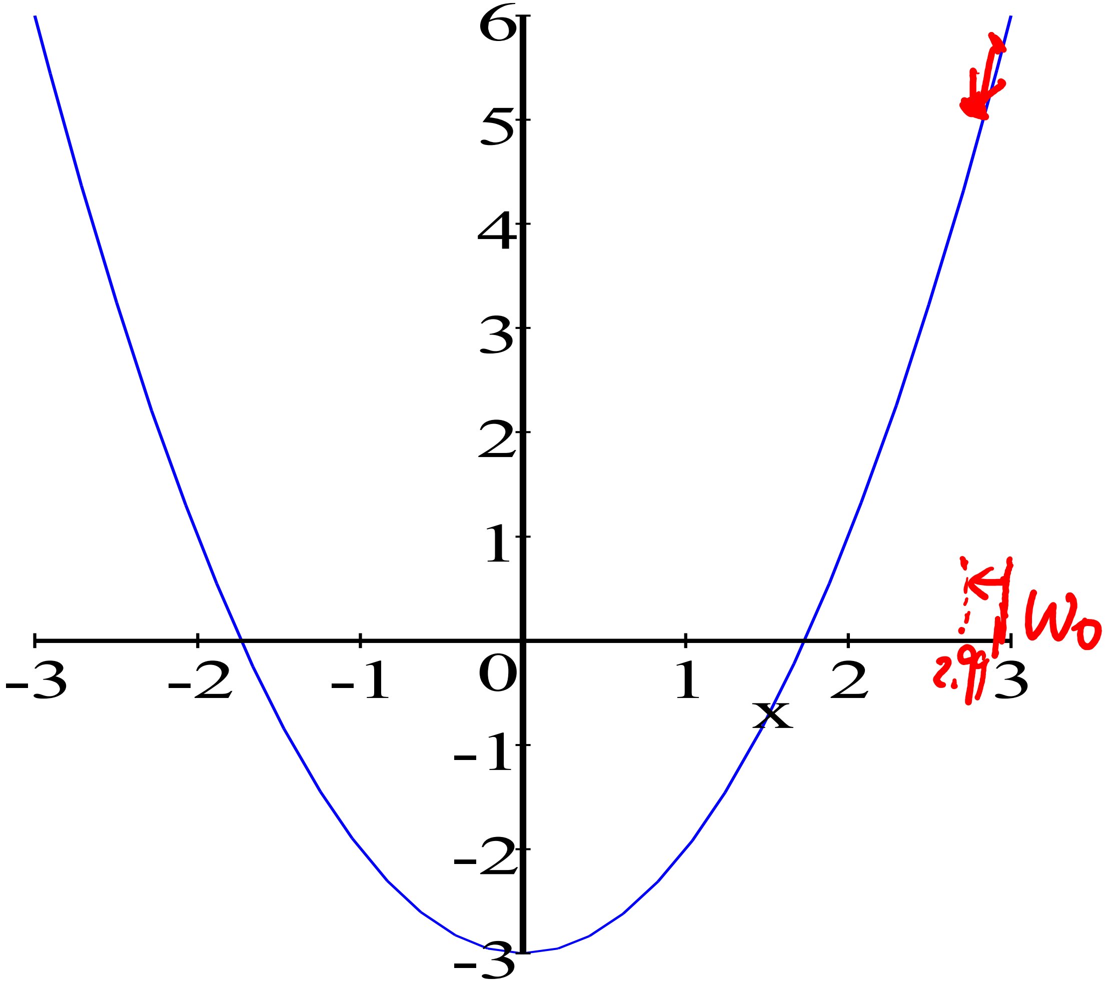
Gradient Descent (Iteratively Optimize)

- Learning Rate Matters

- Objective function matters

- Starting point matters

$$w_k = w_{k-1} - 2 \propto w_{k-1}$$



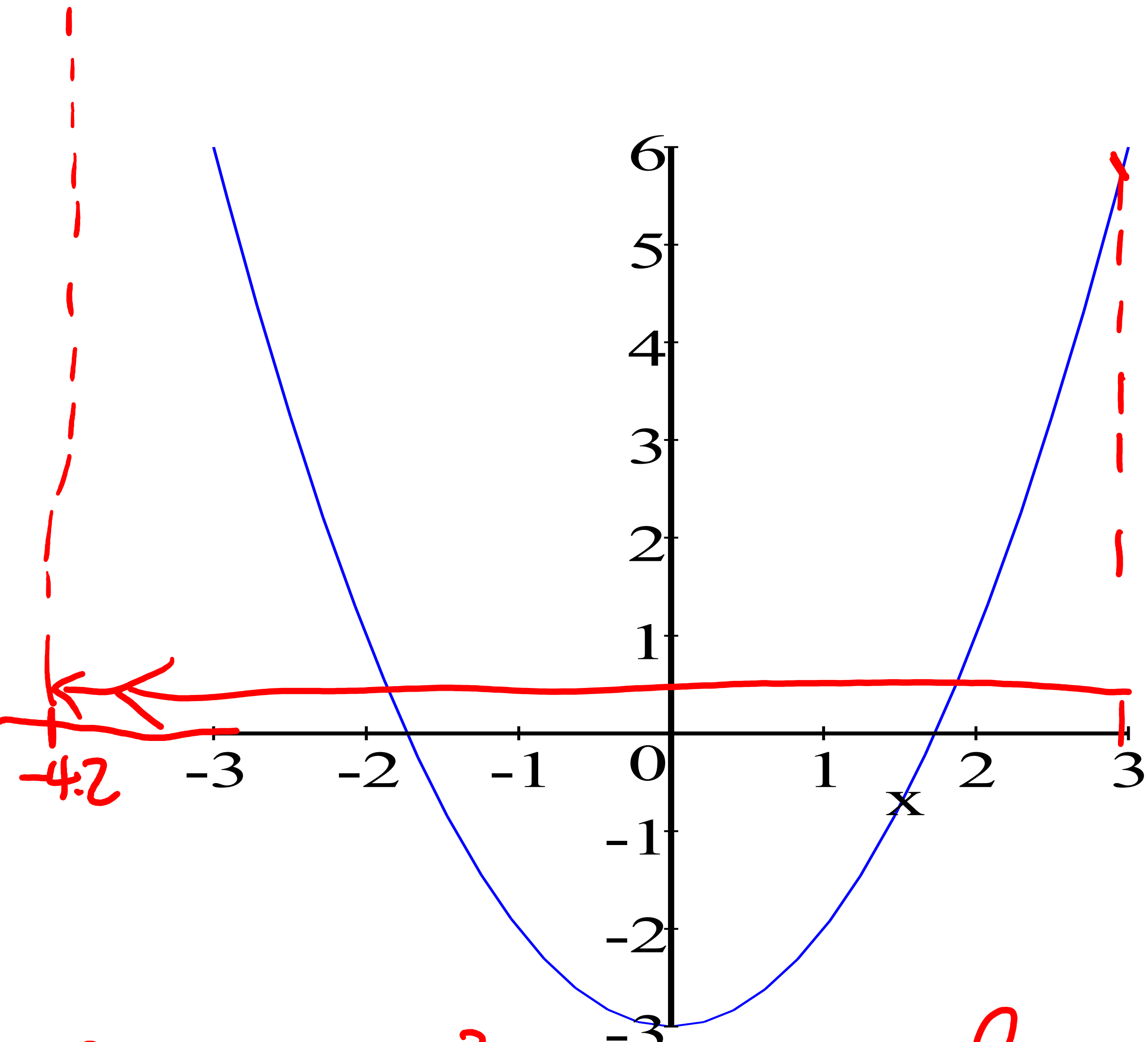
$W_0 = 3$

$\alpha = 0.001$

$W_1 = 3 - 2 \times 0.001 \times 3$

$= 2.994$

$$w_k = w_{k-1} - 2 \propto w_{k-1}$$



$W_0 = 3$

~~$\alpha = 0.1$~~

$\alpha = 1.2$

$W_1 = 3 - 2 \times 1.2 \times 3 = -4.2$

$l(w_1) = (-4.2)^2 - 3 > l(w_0) = 3^2 - 3$

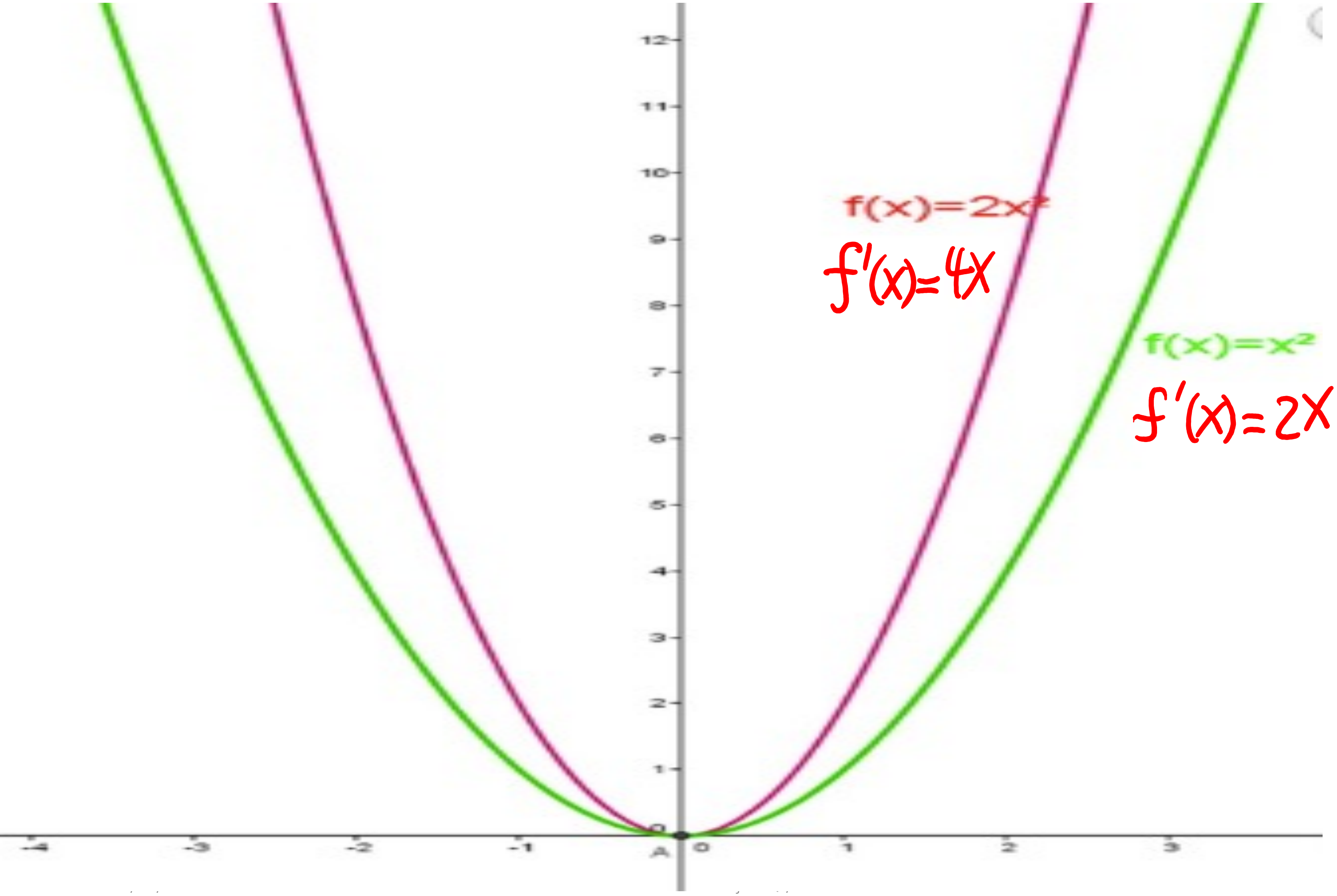
Gradient Descent (Iteratively Optimize)

- Learning Rate Matters

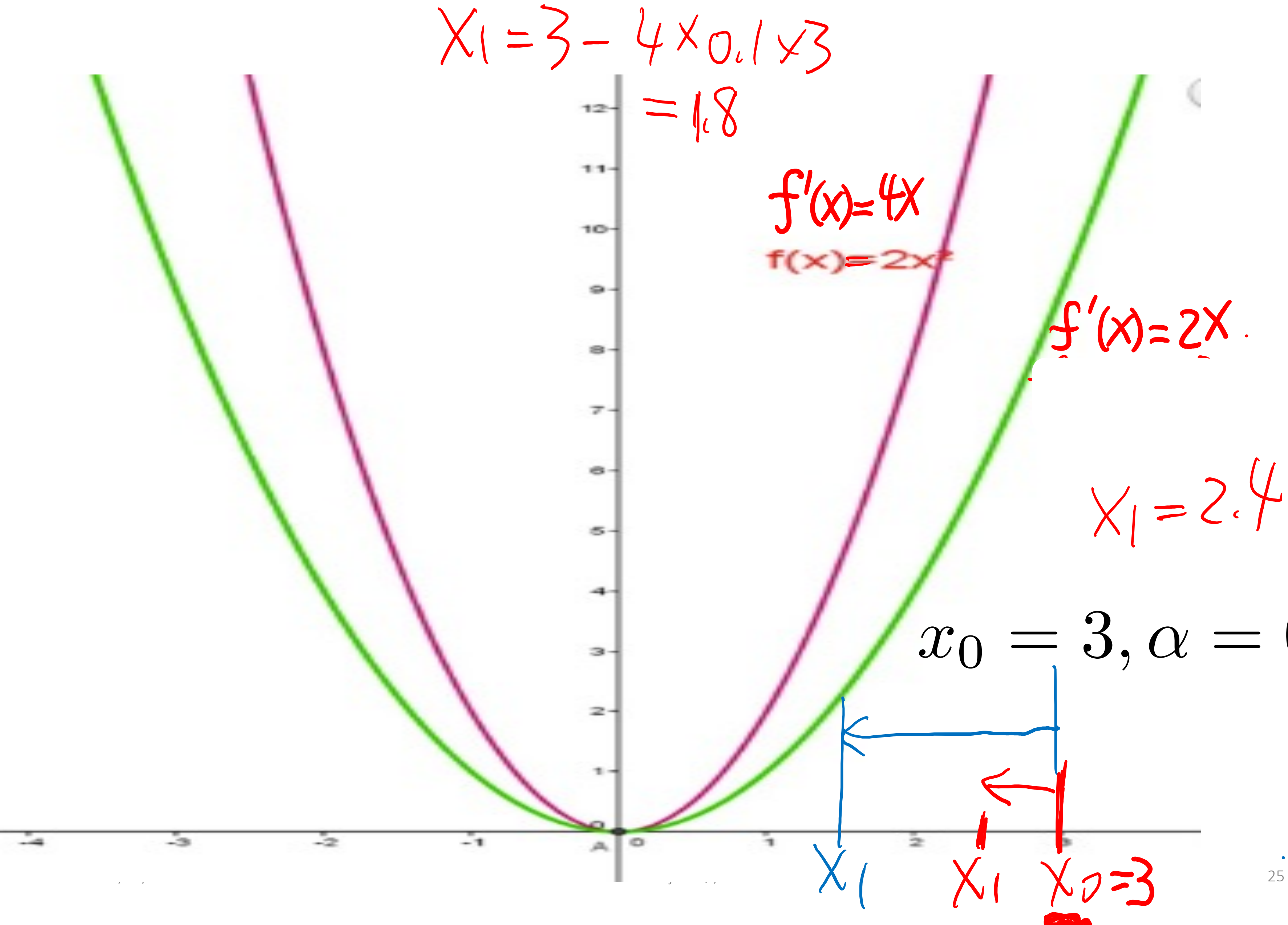
- Objective function matters

- Starting point matters

$$x_k = x_{k-1} - \alpha \nabla_x F(x_{k-1})$$



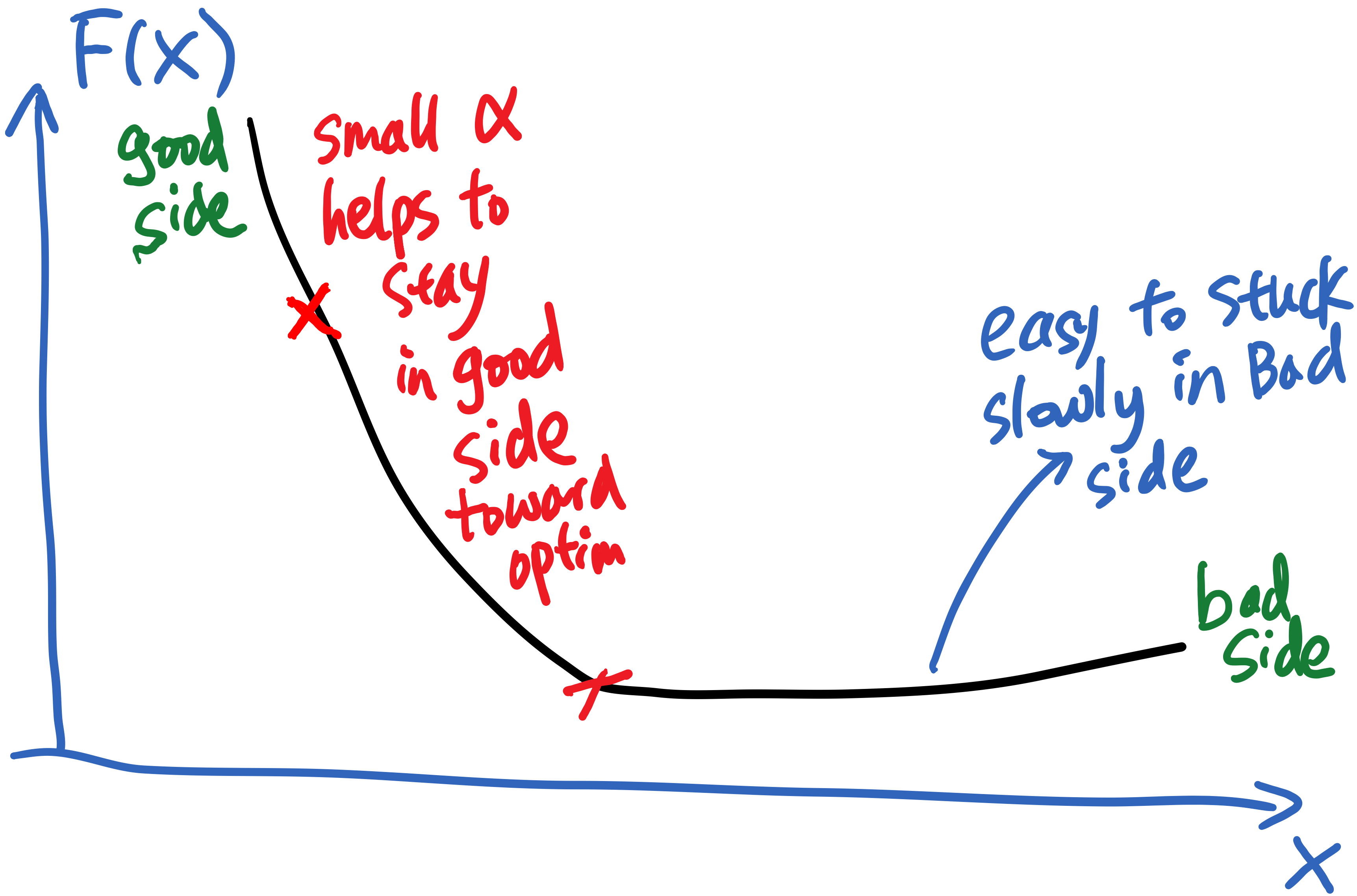
$$x_k = x_{k-1} - \alpha \nabla_x F(x_{k-1})$$

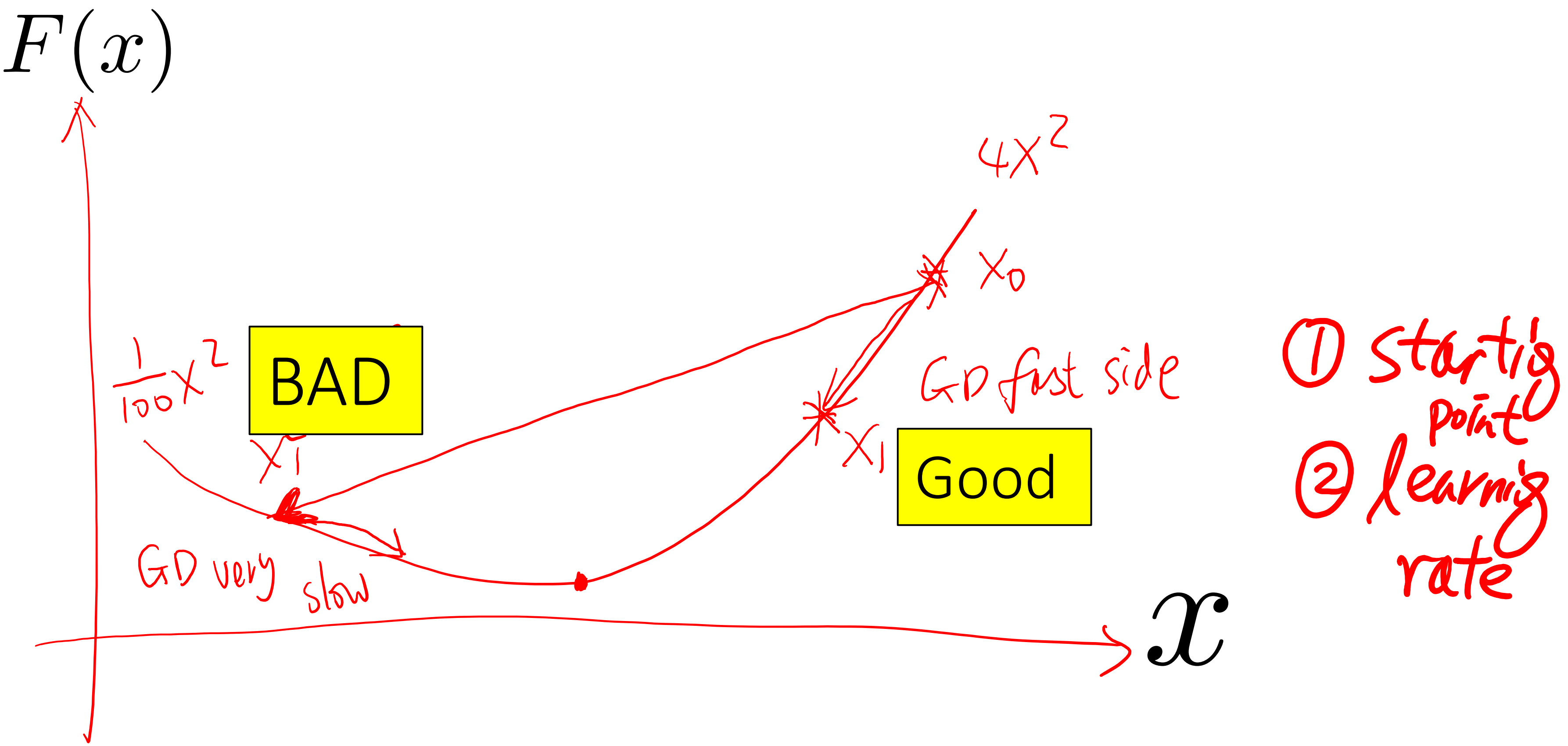


$$x_0 = 3, \alpha = 0.1$$

Gradient Descent (Iteratively Optimize)

- Learning Rate Matters
- Objective function matters
- Starting point matters

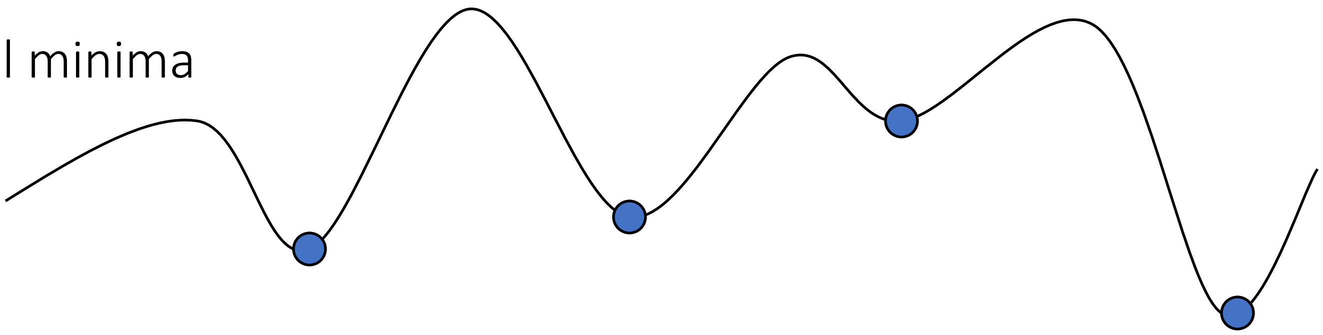

F(x)
good side
small α helps to stay in good side toward optim
easy to stuck slowly in Bad side
bad side
x



During optimization, We don't want to jump from the good side to the bad side

Comments on Gradient Descent Algorithm

- Works on any objective function F(x)
  - as long as we can evaluate the gradient
  - this can be very useful for minimizing complex functions
- Local minima



  - Can have multiple local minima
  - (note: for LR, its cost function only has a single global minimum, so this is not a problem)
  - If gradient descent goes to the closest local minimum:
    - solution: random restarts from multiple places in weight space


Thank You

UVA CS 4774 : Machine Learning

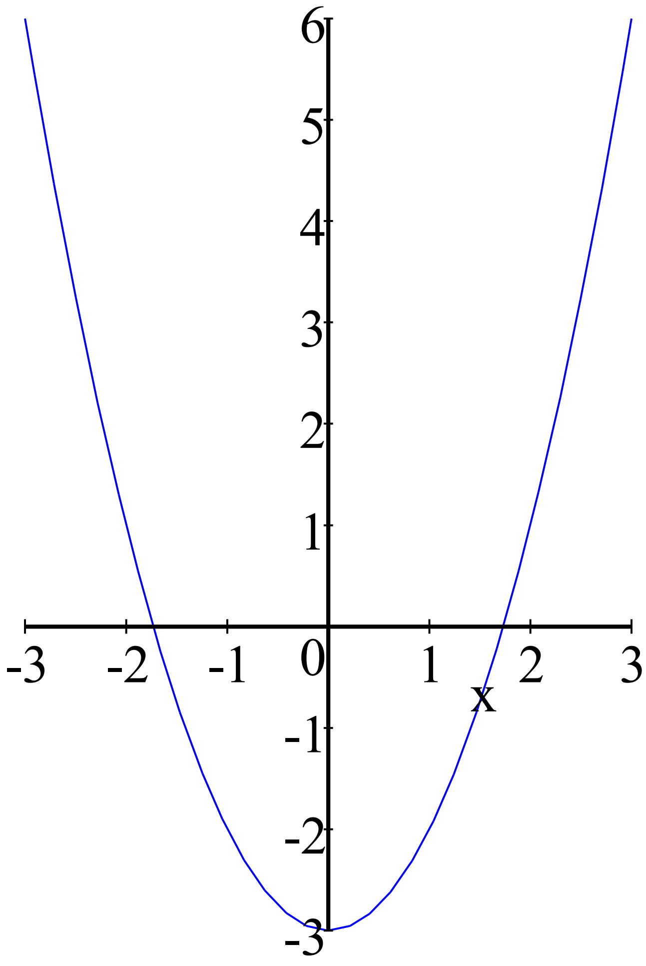
Lecture 4: More optimization for Linear Regression

Module 2

Dr. Yanjun Qi

University of Virginia
Department of Computer Science

Our concrete example with Gradient Descent

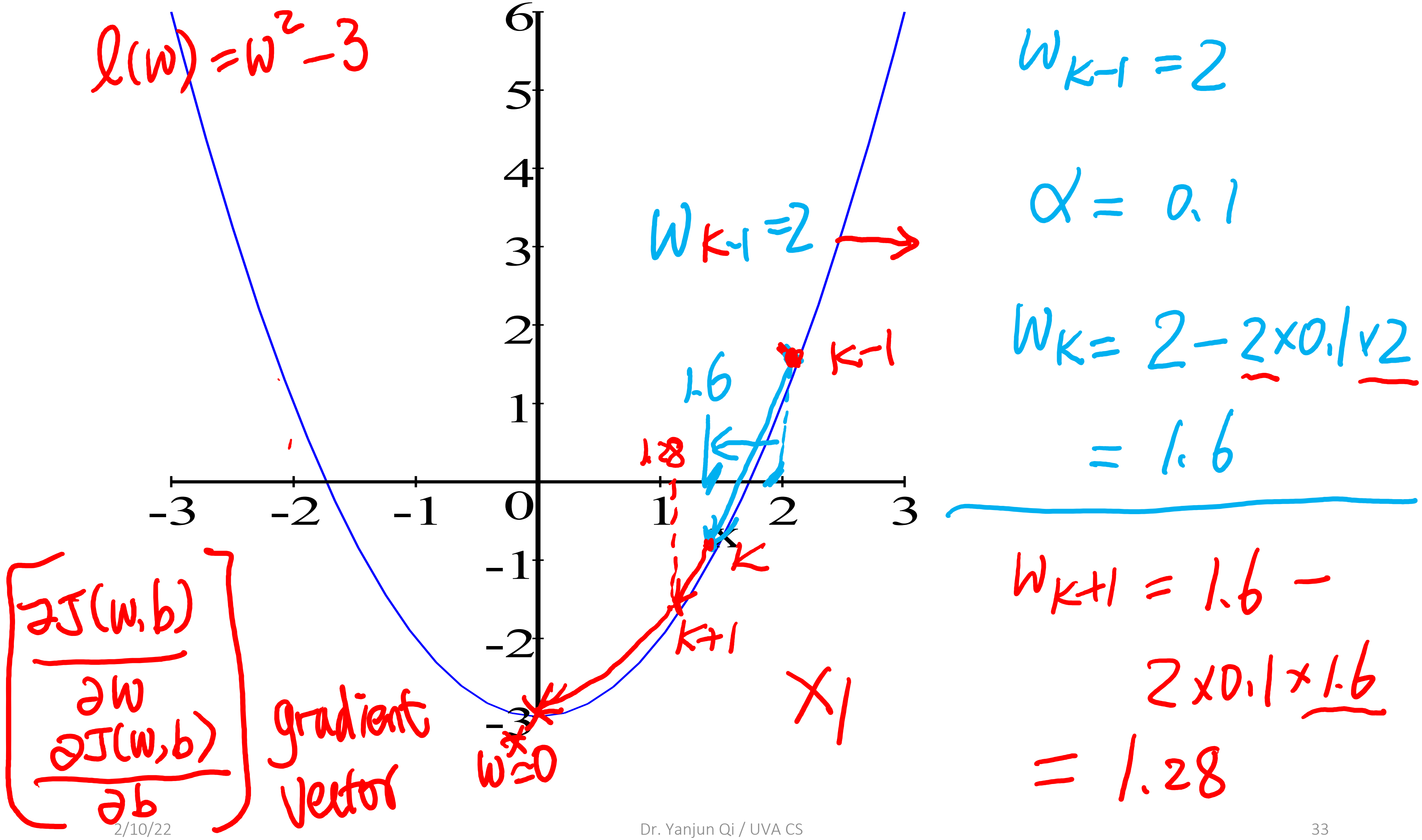


$$l(w) = w^2 - 3$$

$$l'(w) = 2w$$

$$w_k = w_{k-1} - 2 \propto w_{k-1}$$

$$w_k = w_{k-1} - 2 \propto w_{k-1}$$



Iteratively Optimize: Gradient Descent (GD) and Stochastic GD for Linear Regression Training

Review: Loss function of Least Square LR

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{n}(f(\mathbf{x}_i)-y_i)^2$$

$$=\frac{1}{2}\left(\theta^T X^T X\theta-\theta^T X^T \vec{y}-\vec{y}^T X\theta+\vec{y}^T \vec{y}\right)$$

Extra: more in note PDF

$\Rightarrow$ matrix calculus, partial deri $\Rightarrow$ Gradient

$$\nabla_\theta(\theta^T X^T X \theta) = 2X^T X \theta \quad (P24)$$

$$\nabla_\theta(-2\theta^T X^T y) = -2X^T Y \quad (P24)$$

$$\nabla_\theta(y^T y) = 0$$

$$\Rightarrow \nabla_\theta J(\theta) = X^T X \theta - X^T Y$$

$$\nabla_\theta J(\theta)$$

$$= X^T X\theta - X^T \vec{y}$$

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

Linear Regression Trained with batch GD

- A Batch **gradient descent** algorithm:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix}$$

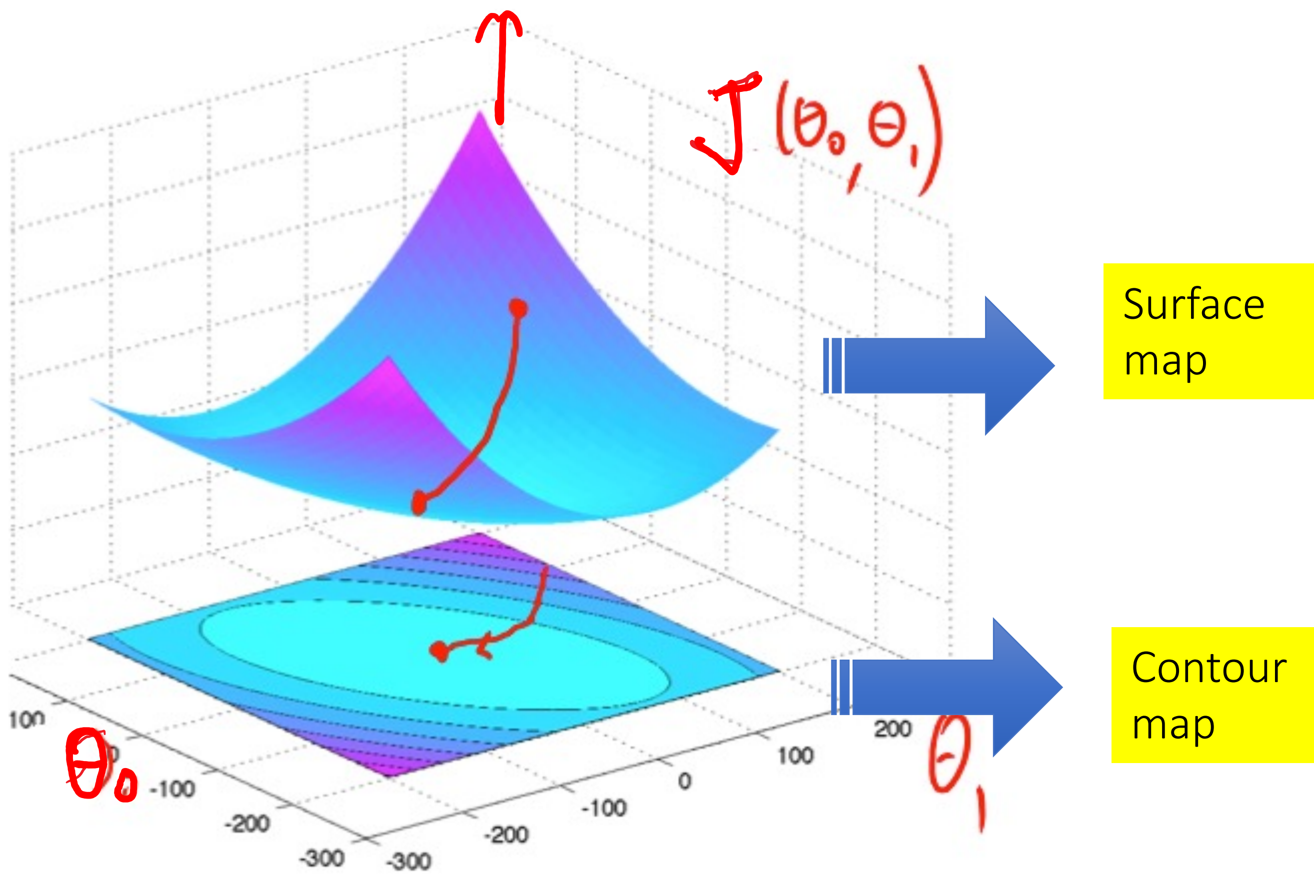
$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta J(\theta^t)$$

$$= \theta^t + \alpha X^T (\vec{y} - X\theta^t)$$

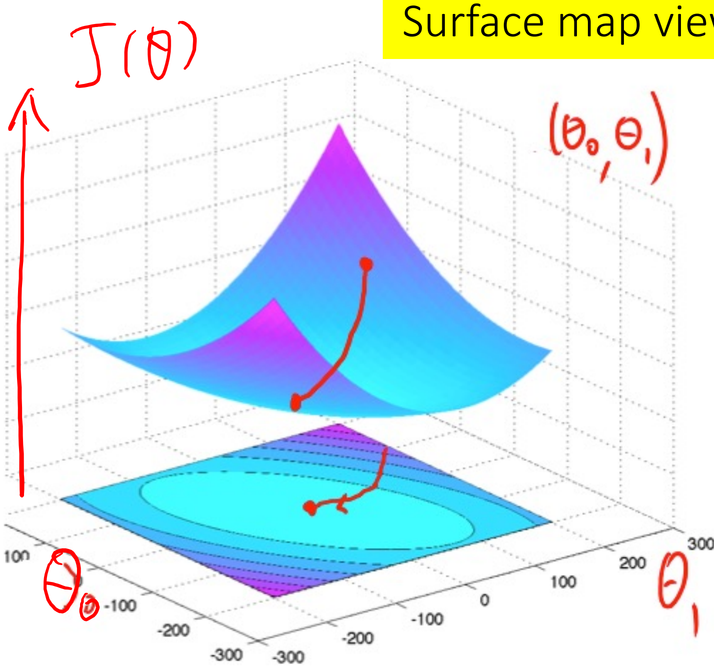
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} -x_1^T\theta- \\ -x_2^T\theta- \\ \vdots \\ -x_n^T\theta- \end{bmatrix}$$

$$GD: x_k = x_{k-1} - \alpha \nabla_x F(x_{k-1})$$

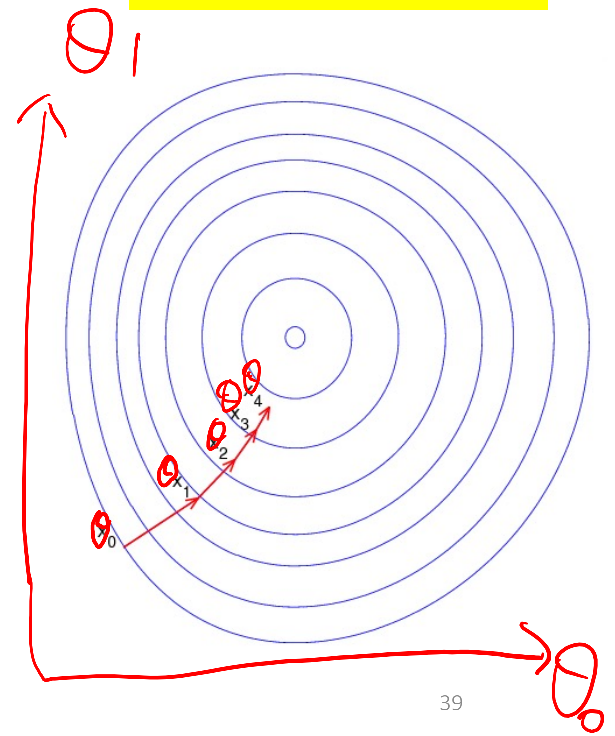
Review: two ways of Illustrating an Objective Function (for two variables case)



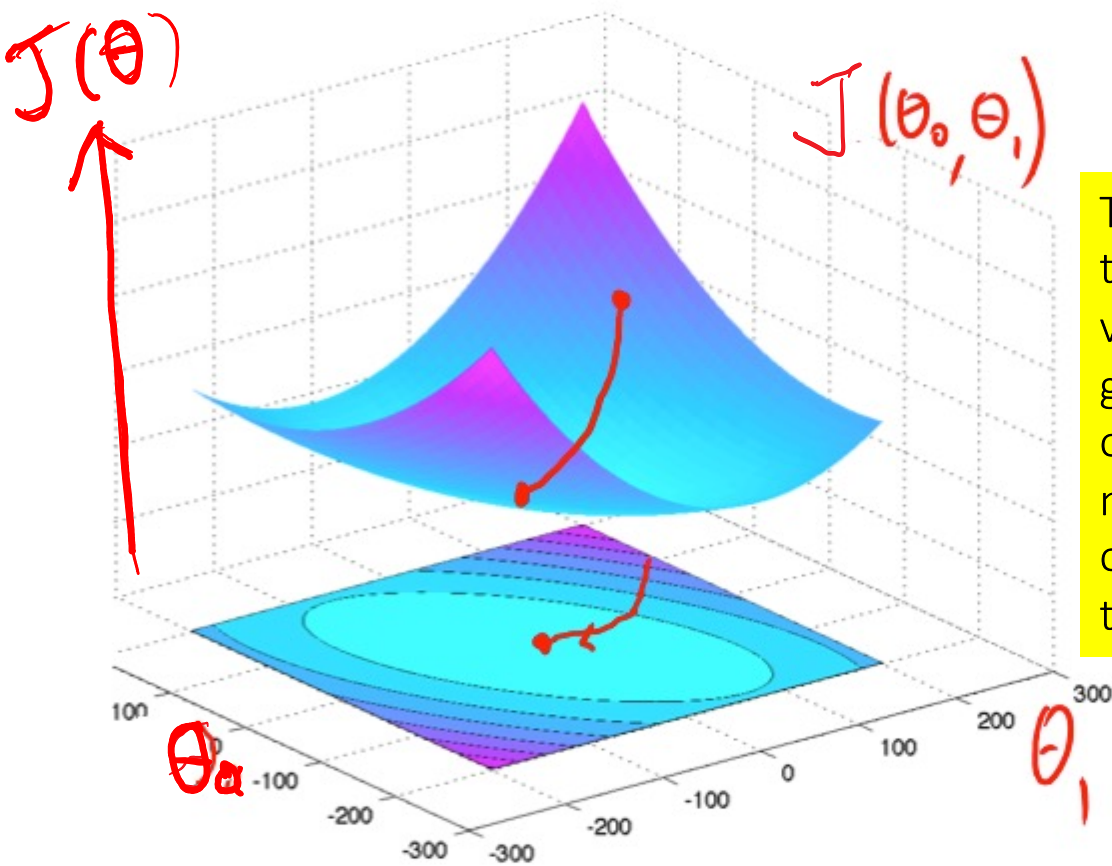
Surface map view



Contour map view



Review: two ways of Illustrating an Objective Function (for two variables case)

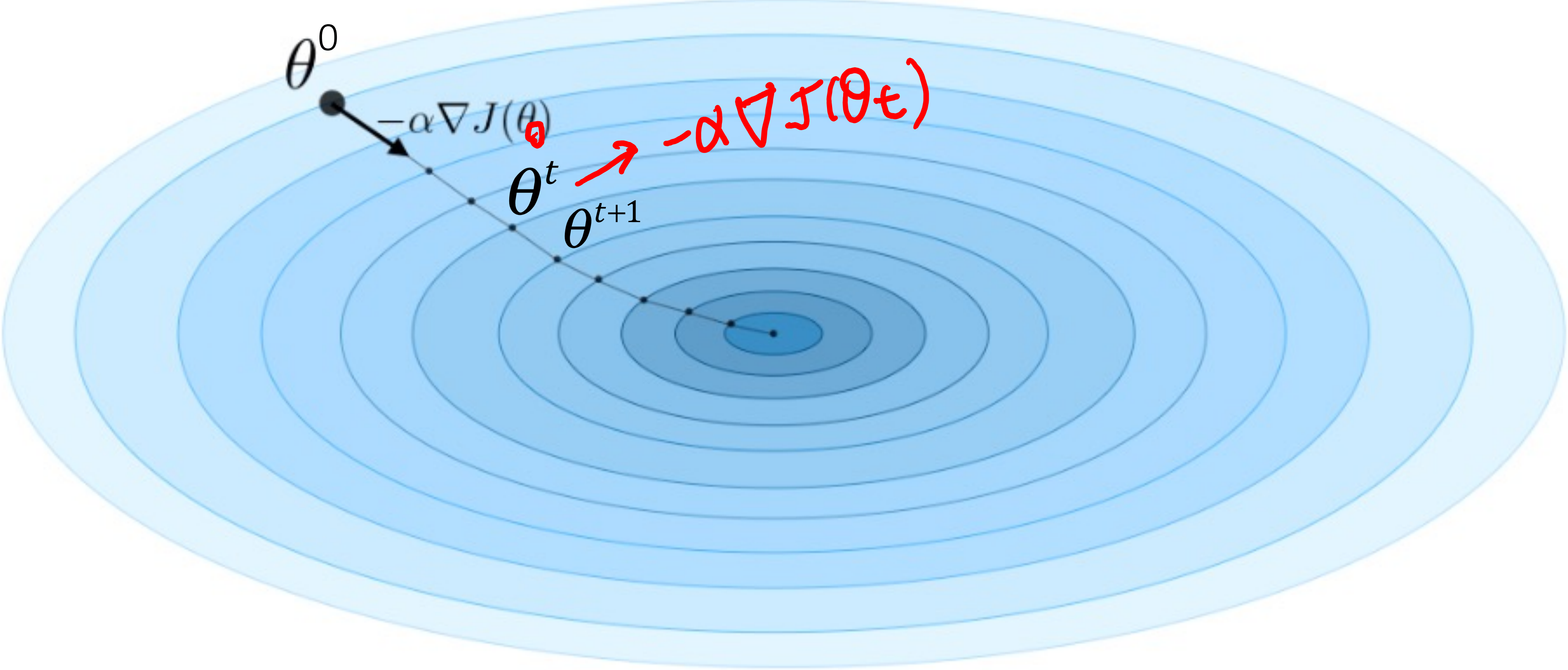


Vector

The gradient points in the direction (in the θ variable space) of the greatest rate of increase of the function and its magnitude is the slope of the surface graph in that direction

Gradient Descent (Steepest Descent)

$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta J(\theta^t)$$
$$= \theta^t + \alpha X^T (\vec{y} - X\theta^t)$$



To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient of the function at the current point.

Contour map view

$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta J(\theta^t)$$
$$= \theta^t + \alpha X^T (\vec{y} - X\theta^t)$$
$$= \theta^t + \alpha \sum_{i=1}^{n} (y_i - \vec{\mathbf{x}}_i^T \theta^t)\vec{\mathbf{x}}_i$$

https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf

a vector of errors on each point @ $\vec{\theta}_t$

$$\vec{\theta}^{t+1} = \theta^t + \alpha X^T (\vec{y} - X\vec{\theta}^t)$$

$p \times 1 \quad p \times 1 \quad 1 \times 1 \quad p \times n \quad n \times 1 \quad n \times p \quad p \times 1$

$n \times 1$

$n \times 1$

$p \times 1$

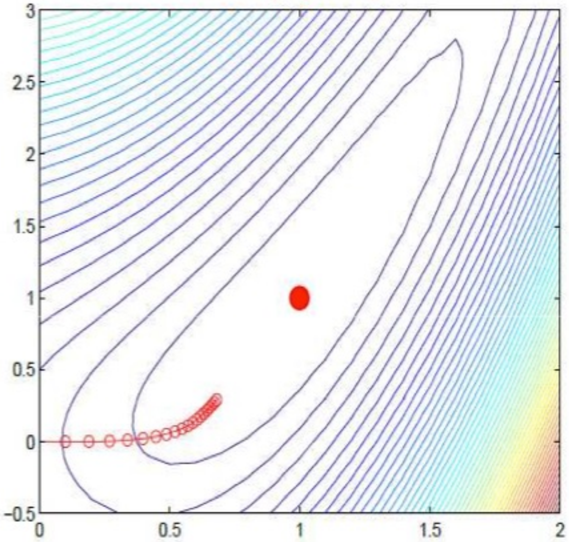
$p \times 1$

https://numpy.org/doc/stable/reference/routines.linalg.html

You can use panda: pandas is a thin abstraction layer on top of numpy

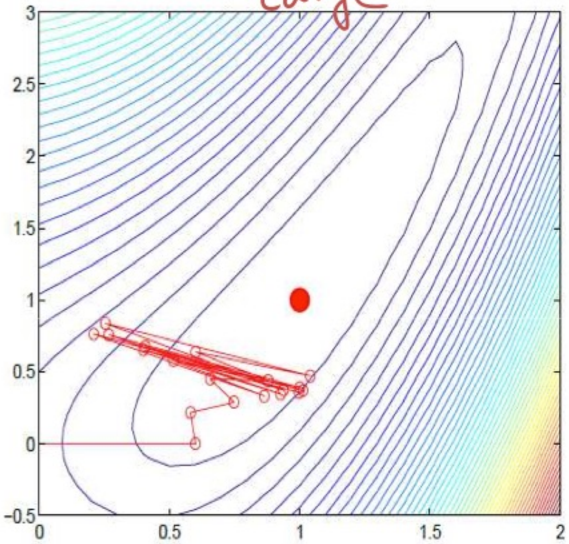
Choosing the Right Learning-Rate is critical

Too small



$\alpha = 0.1.$

Too Large



$\alpha = 0.6.$

Training with batch GD

- Gradient of Cost Function:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i^T\theta - y_i)^2$$

- Consider a gradient descent algorithm and reformulate:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix}$$

$$\theta^{t+1} = \theta^t - \alpha \nabla_\theta J(\theta^t)$$

$$= \theta^t + \alpha X^T(\vec{y} - X\theta^t)$$

$$= \theta^t + \alpha \sum_{i=1}^{n}(y_i - \vec{\mathbf{x}}_i^T\theta^t)\vec{\mathbf{x}}_i$$

Review: Gradient Vector of Linear Regression Loss

- The Cost Function:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(\mathbf{x}_i^T\theta - y_i)^2$$

- Consider a gradient descent algorithm and reformulate:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_p \end{bmatrix}$$

$$\theta^{t+1} = \theta^t - \alpha\nabla_\theta J(\theta^t)$$

$$= \theta^t + \alpha X^T(\vec{y} - X\theta^t)$$

$$= \theta^t + \alpha\sum_{i=1}^{n}(y_i - \vec{\mathbf{x}}_i^T\theta^t)\vec{\mathbf{x}}_i$$

LR with Stochastic GD ➔

n ~ 2M points
p ~ 3k features

$\theta^t + \alpha X^T (y - X\theta^t)$

p×n n×1

- Batch GD rule:

$$\theta^{t+1} = \theta^t + \alpha \sum_{i=1}^{n} (y_i - \vec{\mathbf{x}}_i^T \theta^t)\vec{\mathbf{x}}_i$$

- For a single training point (i-th), we have:

$$\theta^{t+1} = \theta^t + \alpha (y_i - \vec{\mathbf{x}}_i^T \theta^t)\vec{\mathbf{x}}_i$$

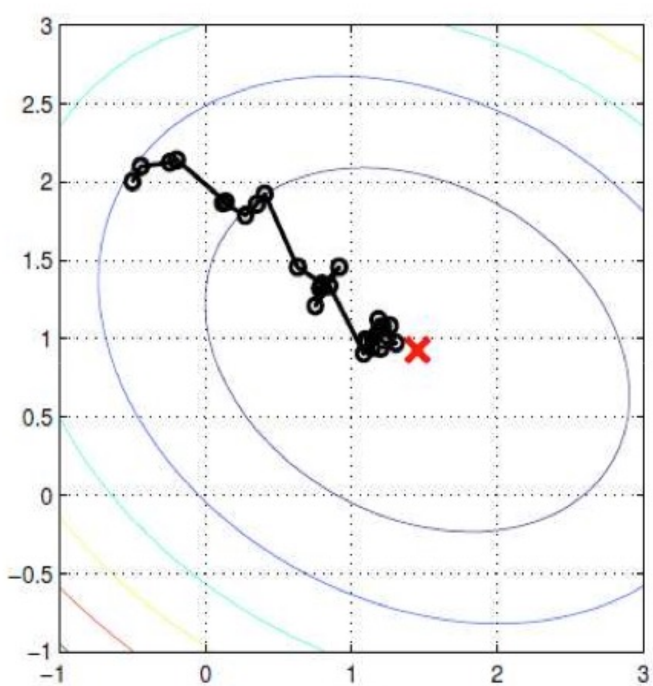
$\theta^0 \longrightarrow \theta^1 \longrightarrow \theta^2$

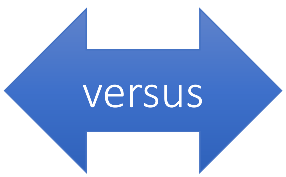
- A "stochastic" descent algorithm
- Can be used as an on-line algorithm

Stochastic gradient descent

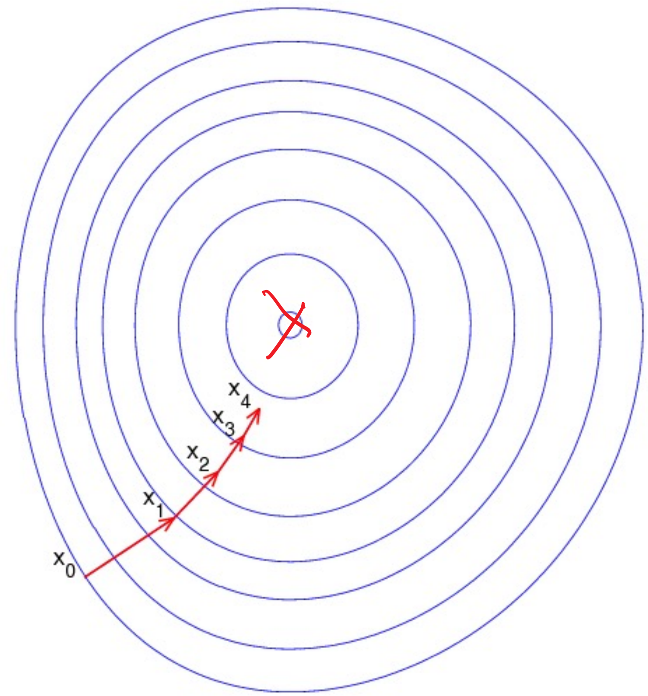
VS. Gradient Descent

SGD





GD



Stochastic gradient descent : More variations

- Single-sample:

$$\theta^{t+1} = \theta^t + \alpha(y_i - \vec{\mathbf{x}}_i^{\,T}\theta^t)\vec{\mathbf{x}}_i$$

- Mini-batch:

$$\theta^{t+1} = \theta^t + \alpha \sum_{j=1}^{B} \left(y_{Ij} - \vec{\mathbf{x}}_{Ij}^{\,T}\theta^t\right) \vec{\mathbf{x}}_{Ij}$$

e.g. B= 15

$$= \theta^t + \alpha X_B^T(\vec{y} - X_B\theta^t)$$

$$J^t_{train\_MSE} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i{}^T\theta^{\mathrm{t}} - y_i)^2$$

OK Implementation for HW1

$$\theta^{\mathrm{t+1}} = \theta^{\mathrm{t}} + \alpha X^T(\boldsymbol{y} - X\theta^{\mathrm{t}})$$

$$\theta^{t+1} = \theta^{t} + \alpha(y_i - \vec{\mathbf{x}}_i{}^T\theta^{t})\vec{\mathbf{x}}_i$$

$$\theta^{\mathrm{t+1}} = \theta^{\mathrm{t}} + \alpha X_B^T(\boldsymbol{y} - X_B\,\theta^{\mathrm{t}})$$

BETTER Practice for HW1

$$\theta^{\mathrm{t+1}} = \theta^{\mathrm{t}} + \frac{1}{n}\alpha X^T(\boldsymbol{y} - X\theta^{\mathrm{t}})$$

$$\theta^{t+1} = \theta^{t} + \alpha(y_i - \vec{\mathbf{x}}_i{}^T\theta^{t})\vec{\mathbf{x}}_i$$

$$\theta^{\mathrm{t+1}} = \theta^{\mathrm{t}} + \frac{1}{B}\alpha X_B^T(\boldsymbol{y} - X_B\,\theta^{\mathrm{t}})$$

Stochastic gradient descent (more)

- Very useful when training with massive datasets , e.g. not fit in main memory
- Very useful when training data arrives online (e.g. streaming)..
- SGD can be used for offline training, by repeated cycling through the whole data
  - Each such pass over the whole data → an epoch !

- In offline case, often better to use mini-batch SGD
  - E.g. B=64
  - B=1 standard SGD
  - B=N standard batch GD

Mini-batch: (stochastic gradient descent)

- Compute the gradient on a small mini-batch of samples (e.g. B=32/64/……)
- Much faster computationally than single point SGD (better use of computer architecture like GPU)

Epoch / cover all examples

GD : one update

SGD : n_tr updates

miniSGD : n_tr/B updates

Low per-step cost, fast convergence and perhaps less prone to local optimum

(Stochastic) Gradient Descent (Iteratively Optimize)

- Learning Rate Matters
- Starting point matters
- Objective function matters
- Stop criterion matters!

- Train MSE Error to observe:

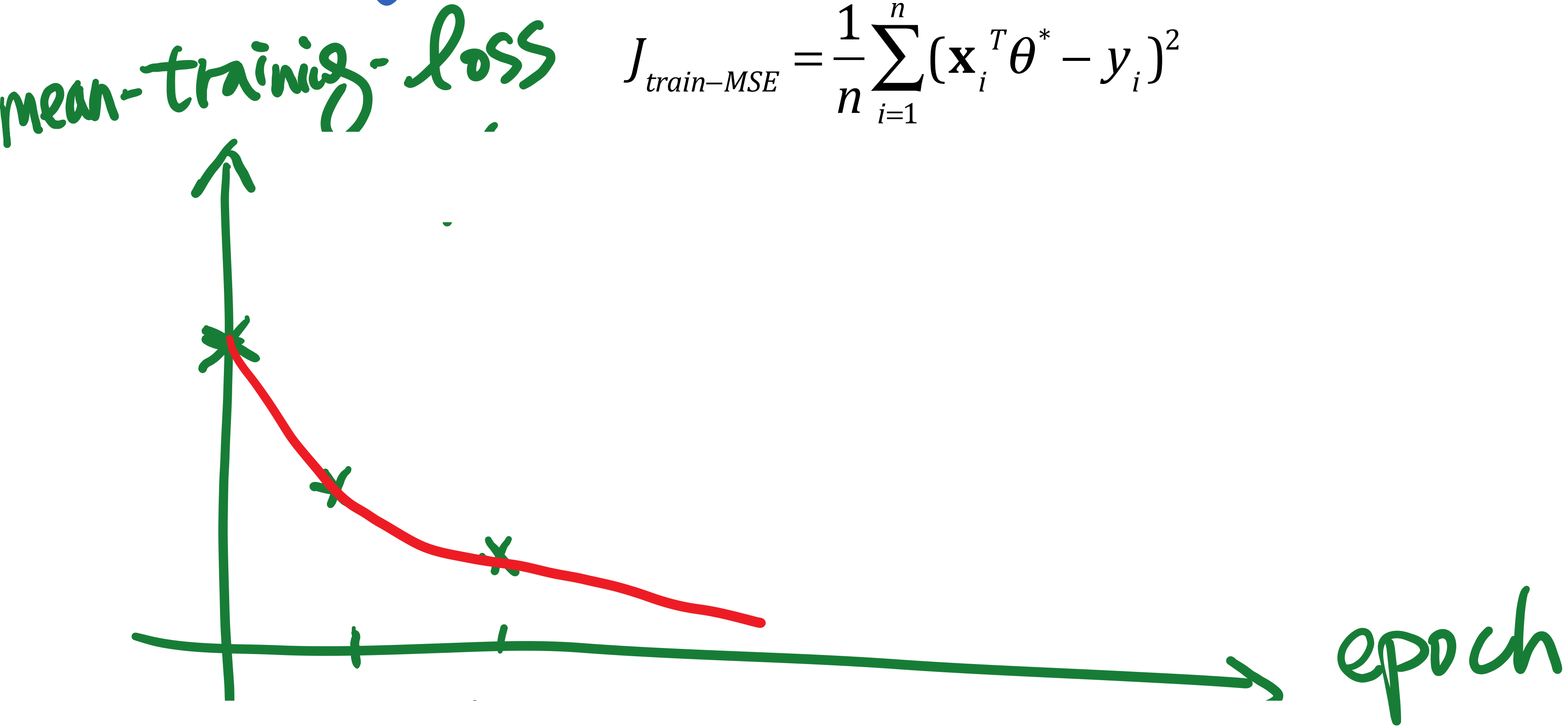
$$J^{t}{}_{train\_MSE} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i{}^T\theta^{\mathrm{t}} - y_i)^2$$

In many situations, visualizing Train-MSE can be helpful to understand the behavior of your method, e.g., how it decreases with epochs, …

In Homework, when we ask for plots of training error, we ask for the MSE per-sample train errors; Because it is comparable to test MSE error (later to cover).

One good plot for GD / SGD

$$J_{train-MSE} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i^T\theta^* - y_i)^2$$



Each pass of SGD repeated cycling through all samples in the whole train ➔ an epoch !

When to stop (S)GD ?

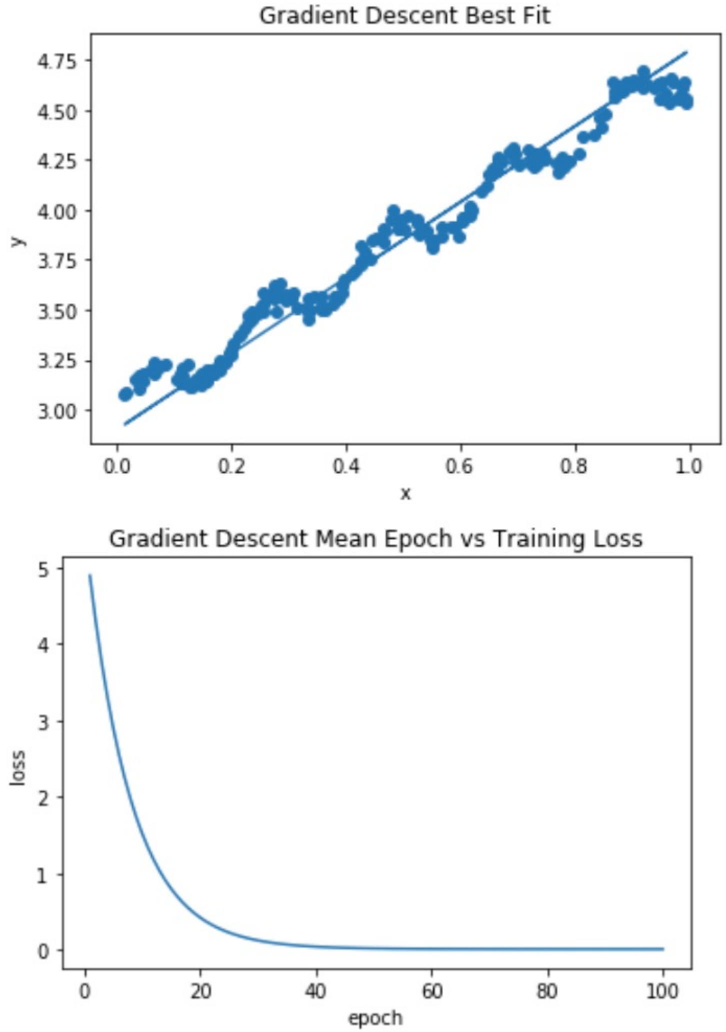
- Lots of stopping rules in the literature,
- there are advantages and disadvantages to each, depending on context

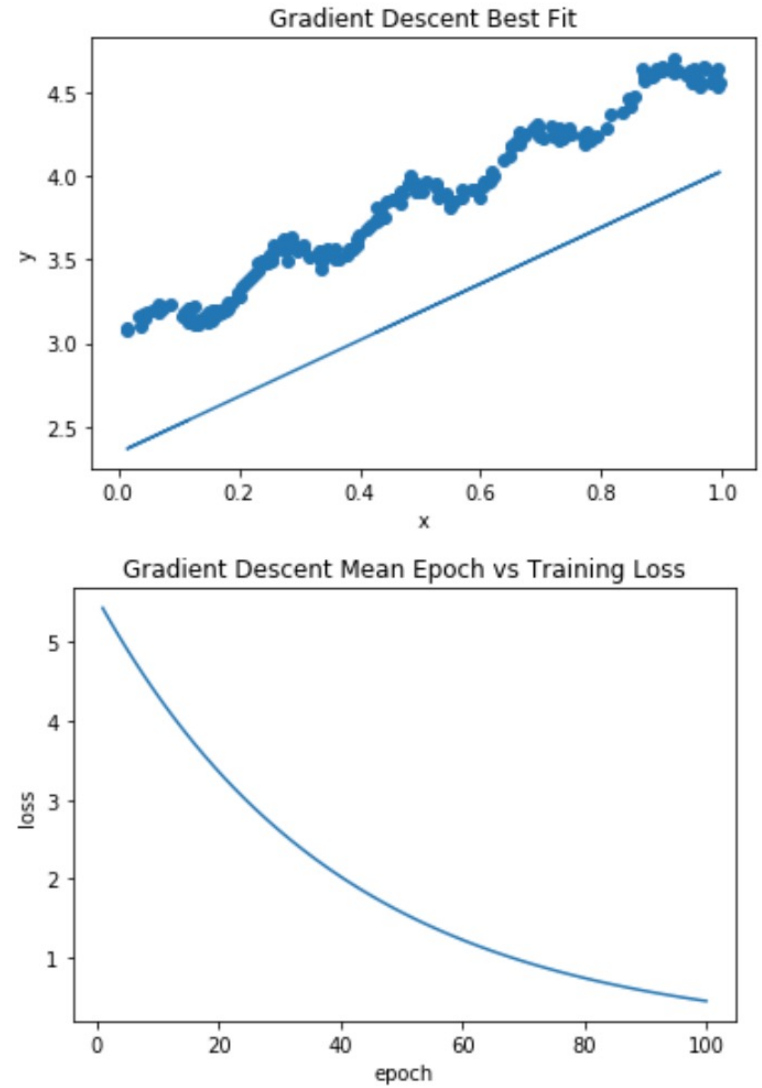
- E.g., a predetermined maximum number of iterations
- E.g., stop when the improvement drops below a threshold
- ….

e.g. HW1 discussions: Stopping and Learning Rates

```
thetas = gradient_descent(X, Y, 0.05, 100)
plotPredict(X, Y, thetas[-1], "Gradient Descent Best Fit")
plot_training_errors(X, Y, thetas, "Gradient Descent Mean Ep
```

```
thetas = gradient_descent(X, Y, 0.01, 100)
plotPredict(X, Y, thetas[-1], "Gradient Descent Best Fit'
plot_training_errors(X, Y, thetas, "Gradient Descent Mear
```





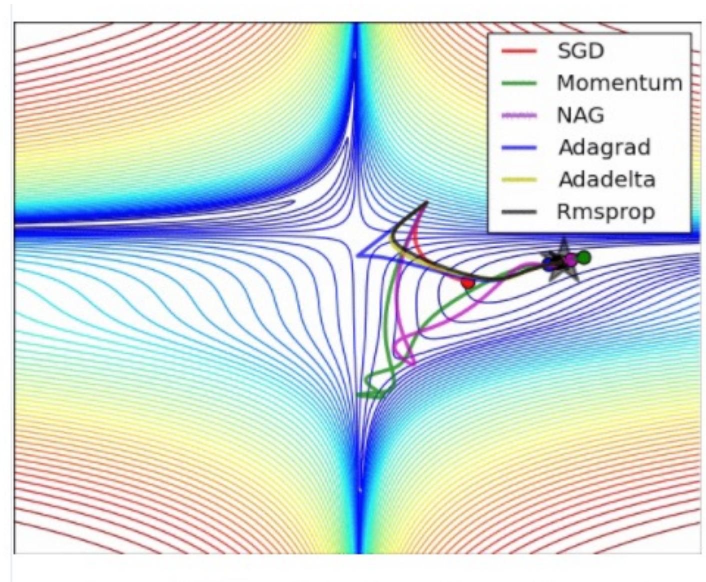
Summary so far: Four ways to learn LR

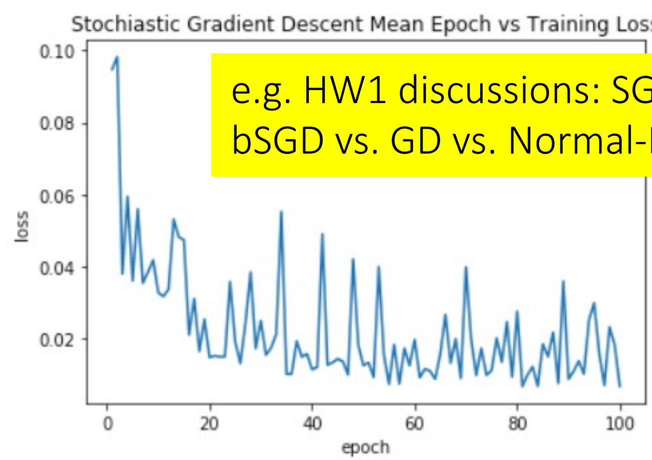
- Normal equations $$\theta^* = \left(X^T X\right)^{-1} X^T \vec{y}$$
  - Pros: a single-shot algorithm! Easiest to implement.
  - Cons: need to compute pseudo-inverse $(X^TX)^{-1}$, expensive, numerical issues (e.g., matrix is singular ..), although there are ways to get around this …
- GD $$\theta^{t+1} == \theta^t + \alpha X^T(\vec{y} - X\theta) = \theta^t + \alpha \sum_{i=1}^{n} (y_i - \mathbf{x}_i^{\ T}\theta^t)\mathbf{x}_i$$
  - Pros: easy to implement, conceptually clean, guaranteed convergence
  - Cons: batch, often slow converging
- Stochastic GD and miniBatch $$\theta^{t+1} = \theta^t + \alpha(y_i - \mathbf{x}_i^T\theta^t)\mathbf{x}_i$$ $$\theta^{t+1} = \theta^t + \alpha \sum_{j=1}^{B} \left(y_{Ij} - \vec{\mathbf{x}}_{Ij}^{\ T}\theta^t\right) \vec{\mathbf{x}}_{Ij}$$
  - Pros: on-line, low per-step cost, fast convergence and perhaps less prone to local optimum
  - Cons: convergence to optimum not always guaranteed

frequent
→ N updates
noise

- Challenges
- Gradient descent optimization algorithms
  - Momentum
  - Nesterov accelerated gradient
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam
  - AdaMax
  - Nadam
  - AMSGrad
  - Other recent optimizers
  - Visualization of algorithms
  - Which optimizer to use?
- Parallelizing and distributing SGD
  - Hogwild!
  - Downpour SGD
  - Delay-tolerant Algorithms for SGD
  - TensorFlow
  - Elastic Averaging SGD
- Additional strategies for optimizing SGD
  - Shuffling and Curriculum Learning
  - Batch normalization
  - Early Stopping
  - Gradient noise

- More about SGD:
  - Popular optimization for NOW
  - Many advanced variations
  - https://ruder.io/optimizing-gradient-descent/
  - https://arxiv.org/abs/1609.04747



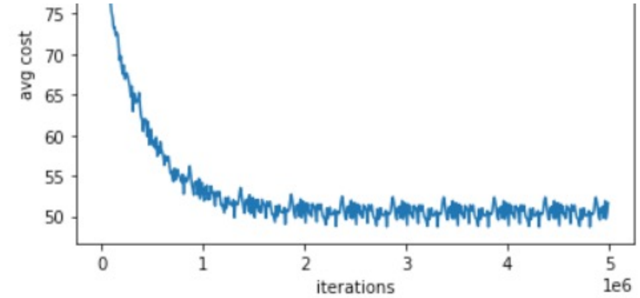


e.g. HW1 discussions: SGD vs. bSGD vs. GD vs. Normal-Equation

GD and SGD code cell run @
https://colab.research.google.com/drive/1Gchka0n69mTwRvZUEpPBKK1BgUq94qPk?usp=sharing

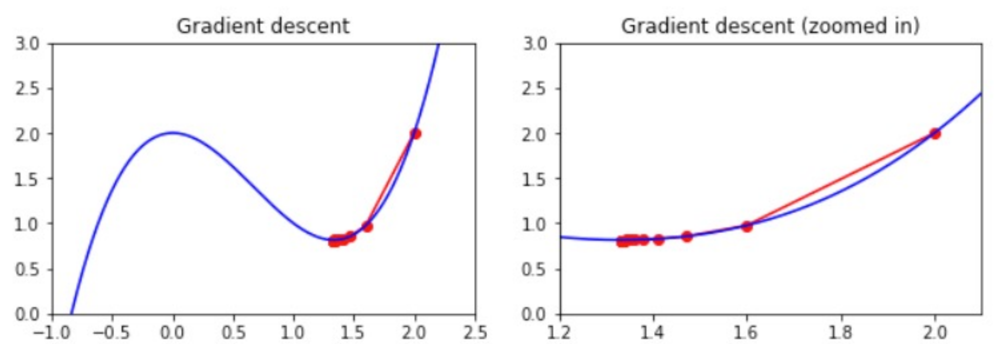
Revised from:
http://cs229.stanford.edu/
https://github.com/dtnewman/stochastic_gradient_descent

18]



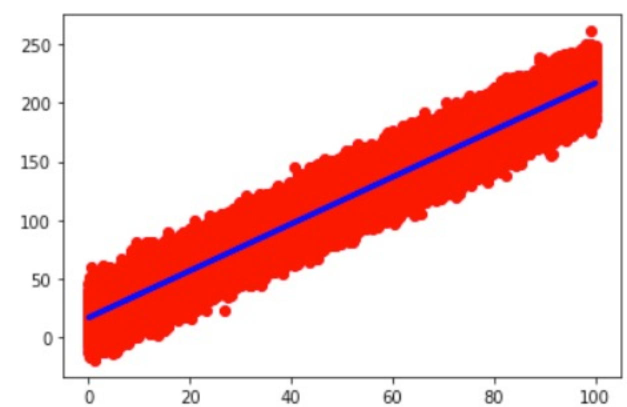

```
# use scipy fmin function to find ideal step size.
n_k = fmin(f2,0.1,(x_old,s_k), full_output = False, disp = False)
```

```
plt.title("Gradient descent (zoomed in)")
plt.show()
```




```
f = lambda x: x*2+17+np.random.randn(len(x))*10

x = np.random.random(500000)*100
y = f(x)
hy = theta_new[0] + theta_new[1]*x
plt.plot(x,hy,c="b", linewidth=3)
plt.scatter(x,y,c="r")
plt.show()
```



Recap : GD and SGD for Multivariate Linear Regression

Data: X

↓

Task: y

↓

Representation: : x, f()

↓

Score Function: L()

↓

Search/Optimization : argmin()

↓

Models, Parameters

→

X: Tabular

↓

Regression: y continuous

↓ $\hat{y} = f(x) = \theta^T x$

Y = Weighted linear sum of Xs

↓

Sum of Squared Error (Least Squared)

↓

Normal Equation / GD / SGD ←

↓

Metrics, Implementation, Regression coefficients w, b

Thank You

EXTRA I

In Case you are interested in more advanced details!

B → Varying the value B In

$$\theta^{t+1} = \theta^{t} + \alpha \sum_{j=1}^{B} \left(y_{Ij} - \vec{\mathbf{x}}_{Ij}^{\,T}\theta \right) \vec{\mathbf{x}}_{Ij}$$

1	1<B<n	n
SGD	miniB-SGD	GD
very noisy GD update	a bit noisy GD update	precise GD update
low memory cost	middle memory cost	high memory cost
O(one gradient cacl cost)	if multi-parallal B threads O(one gradient cal cost)	very costly gradient calculation

Extra: Direct (normal equation) vs. Iterative (GD, SGD,) methods

- Direct methods: we can achieve the solution in a single step by solving the normal equation
  - Using Gaussian elimination or QR decomposition, we converge in a finite number of steps
  - It can be infeasible when data are streaming in in real time, or of very large amount

- Iterative methods: stochastic GD or GD
  - Converging in a limiting sense
  - But more attractive in large practical problems
  - Caution is needed for deciding the learning rate

Stochastic gradient descent (Pros)

- Efficiency: Good approximation of Gradient:
  - Intuitively fairly good estimation of the gradient by looking at just a few examples
  - Carefully evaluating precise gradient using large set of examples is often a waste of time (because need to calculate the gradient of the next t any way)
  - Better to get a noisy estimate and move rapidly in the parameter space

- SGD is often less prone to stuck in shallow local minima
  - Because of the certain "noise",
  - popular for nonconvex optimization cases

Extra: Convergence rate

- **Theorem**: the steepest descent / GD equation algorithm converge to the minimum of the cost characterized by normal equation:

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

If the learning rate parameter satisfy ➔

$$0 < \alpha < 2/\lambda_{\max}[X^T X]$$

- A formal analysis of GD-LR need more math; in practice, one can use a small a, or gradually decrease a.

$\alpha_0 = 0.05$

Extra: Computational Cost (Naïve..)

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

(X^T: $p \times n$, X: $n \times p$)

$X^T X$: $O(p^2 n)$

$(X^T X)^{-1}$: $O(p^3)$

$O(np^2 + p^3)$

when $n \gg p$, matrix multi slower than inversion

mostly about Memory Cost →

Interesting discussion in: https://stackoverflow.com/questions/10326853/why-does-lm-run-out-of-memory-while-matrix-multiplication-works-fine-for-coeffic

$$\nabla_\theta J(\theta) = X^T X \theta - X^T Y$$

$$= X^T (X\theta - Y)$$

$$\mathbf{X} = \begin{bmatrix} -- & \mathbf{x}_1^T & -- \\ -- & \mathbf{x}_2^T & -- \\ \vdots & \vdots & \vdots \\ -- & \mathbf{x}_n^T & -- \end{bmatrix}$$

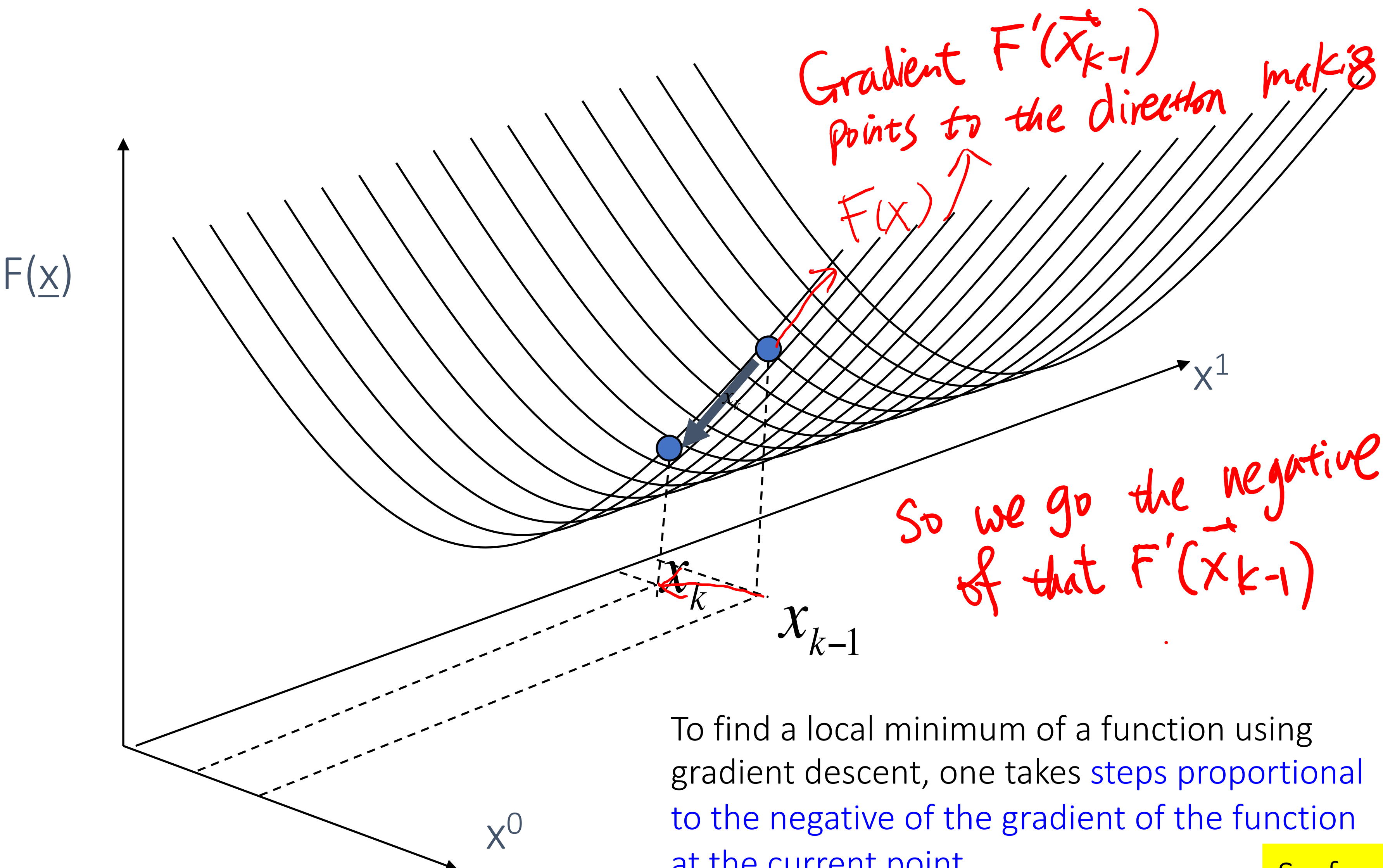
$$= X^T \left(\underset{n \times p}{\begin{bmatrix} - x_1^T - \\ - x_2^T - \\ \cdots \\ - x_n^T - \end{bmatrix}} \underset{p \times 1}{\theta} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \right).$$

$$= \underset{p \times n}{X^T} \underset{n \times 1}{\begin{bmatrix} x_1^T \theta - y_1 \\ x_2^T \theta - y_2 \\ \cdots \\ x_n^T \theta - y_n \end{bmatrix}} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x_1^T \theta - y_1 \\ \vdots \\ x_n^T \theta - y_n \end{bmatrix}.$$

$$= \sum_{i=1}^{n} x_i \cdots (x_i^T \theta - y_i)$$

1*1

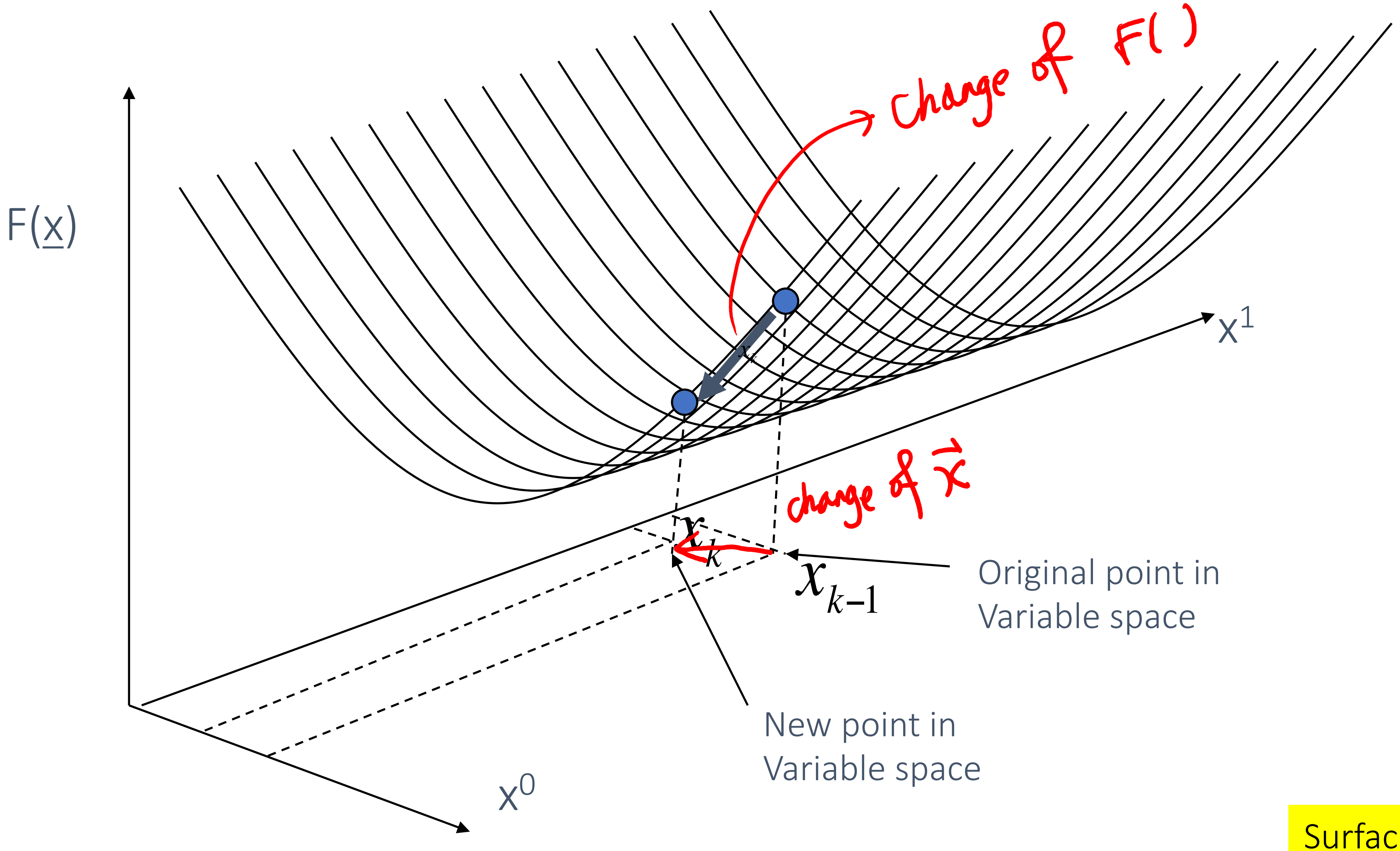
Illustration of Gradient Descent (2D case)



To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient of the function at the current point.

Surface map view

Illustration of Gradient Descent (2D case)



Surface map view

LR with batch GD / Per Feature View

- Note that:

$$\nabla_\theta J = \left[\frac{\partial}{\partial \theta_1} J, \ldots, \frac{\partial}{\partial \theta_k} J \right]^T$$

- For its j-th variable:

$$\theta^{t+1} = \theta^t + \alpha \sum_{i=1}^{n} (y_i - \mathbf{x}_i^T \theta^t) \mathbf{x}_i$$

$$\theta_j^{\,t+1} = \theta_j^{\,t} + \alpha \sum_{i=1}^{n} (y_i - \mathbf{x}_i^{\,T} \theta^t) x_{i,j}$$

Update Rule Per Feature
(Variable-Wise)

LR with Stochastic GD / Per Feature View

- For a single training point (i-th), we have:
  - For its j-th variable:

$$\theta^{t+1} = \theta^{t} + \alpha(y_i - \vec{\mathbf{x}}_i{}^T\theta^{t})\vec{\mathbf{x}}_i$$

$$\theta_j{}^{t+1} = \theta_j{}^{t} + \alpha(y_i - \vec{\mathbf{x}}_i{}^T\theta^{t})x_{i,j}$$

Update Rule Per Feature (Variable-Wise)

EXTRA II

In Case you are interested in more advanced details!

Extra: Newton's Method and

Connecting to Normal Equation

Review: Single Var-Func to Multivariate

Single **Var-Function**	Multivariate Calculus
Derivative Second-order derivative	Partial **Derivative** Gradient Directional Partial Derivative Vector Field Contour map of a function Surface map of a function Hessian matrix Jacobian matrix (vector in / vector out)

$$\vec{x} \xrightarrow{F(\vec{x})} \vec{y}$$

$p \times 1$ $\quad\quad$ $c \times 1$

$p=1$	$c=1$	derivative	2nd-order derivative
$p>1$	$c=1$	gradient vector	Hessian matrix
$p>1$	$c>1$	Jacobian matrix	

Newton's method for optimization

- The most basic second-order optimization algorithm
- Updating parameter with

GD: $\theta_{k+1} = \theta_k - \alpha g_k$

Newton: $$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{H}_K^{-1}\mathbf{g}_k$$

$p \times p$ $p \times 1$

$p \times 1$

Review: Hessian Matrix / n==2 case

Singlevariate → multivariate

$$f(x,y)$$

- 1st derivative to gradient,

$$g = \nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

- 2nd derivative to Hessian

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

Review: Hessian Matrix

Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is a function that takes a vector in $\mathbb{R}^n$ and returns a real number. Then the ***Hessian*** matrix with respect to x, written $\nabla_x^2 f(x)$ or simply as H is the $n \times n$ matrix of partial derivatives,

$$\nabla_x^2 f(x) \in \mathbb{R}^{n \times n} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}.$$

Newton's method for optimization

- Making a quadratic/second-order Taylor series approximation

$$\hat{f}_{quad}(\boldsymbol{\theta}) = f(\boldsymbol{\theta}_k) + \mathbf{g}_k^T(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \mathbf{H}_k(\boldsymbol{\theta} - \boldsymbol{\theta}_k)$$

Finding the minimum solution of the above right quadratic approximation (quadratic function minimization is easy !)

$$\hat{f}(\theta) = f(\theta_k) + g_k^T(\theta - \theta_k) + \frac{1}{2}(\theta - \theta_k)^T H_k (\theta - \theta_k)$$

$$\Downarrow$$

$$\frac{1}{2}\left(\theta^T H_k \theta - 2\theta^T H_k \theta_k + \theta_k^T H_k \theta_k\right)$$

$$\frac{\partial \hat{f}(\theta)}{\partial \theta} = 0 + g_k + \frac{2}{2} H_k \theta - \frac{2}{2} H_k \theta_k := 0$$

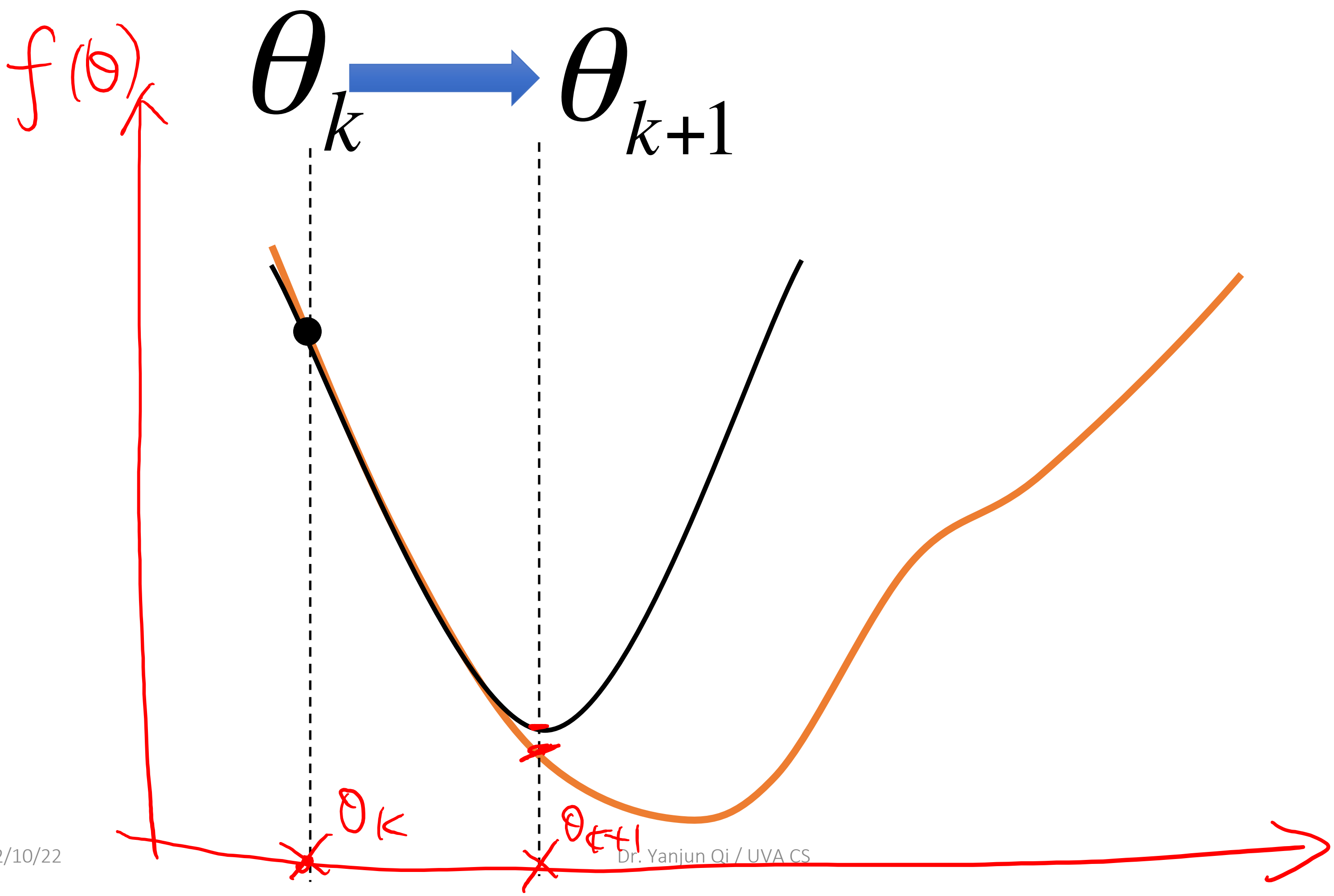
see p24 handout

$$g_k + H_k(\theta - \theta_k) = 0$$

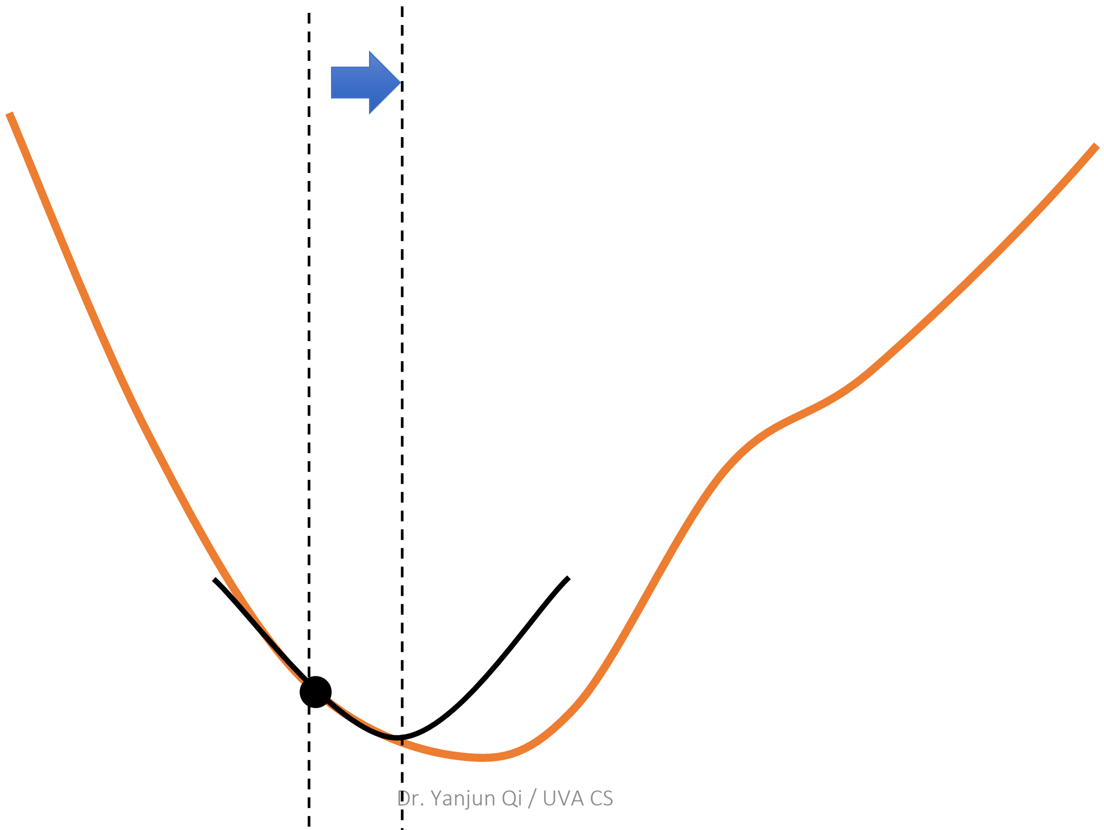
$$\Rightarrow \quad \theta = \theta_k - H_k^{-1} g_k$$

where $\begin{cases} H_k \in \mathbb{R}^{p \times p} \\ g_k \in \mathbb{R}^{p} \end{cases}$

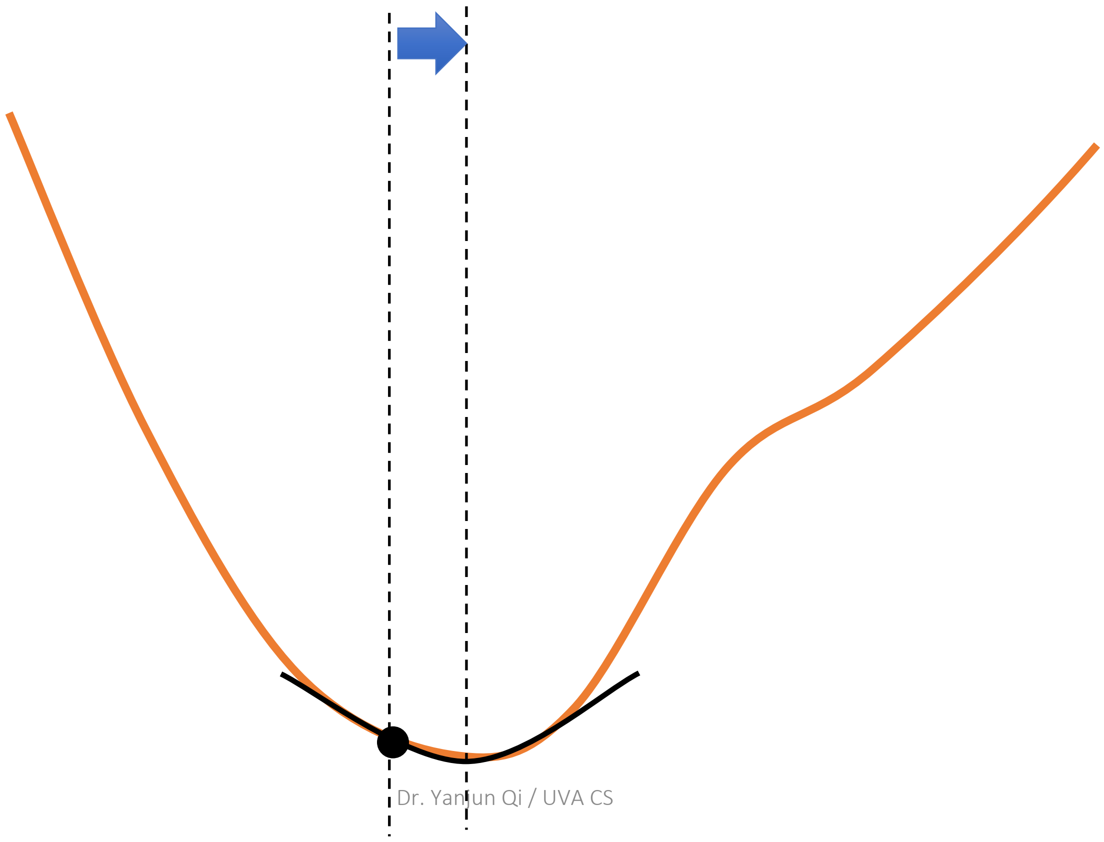
Newton's Method / second-order Taylor series approximation



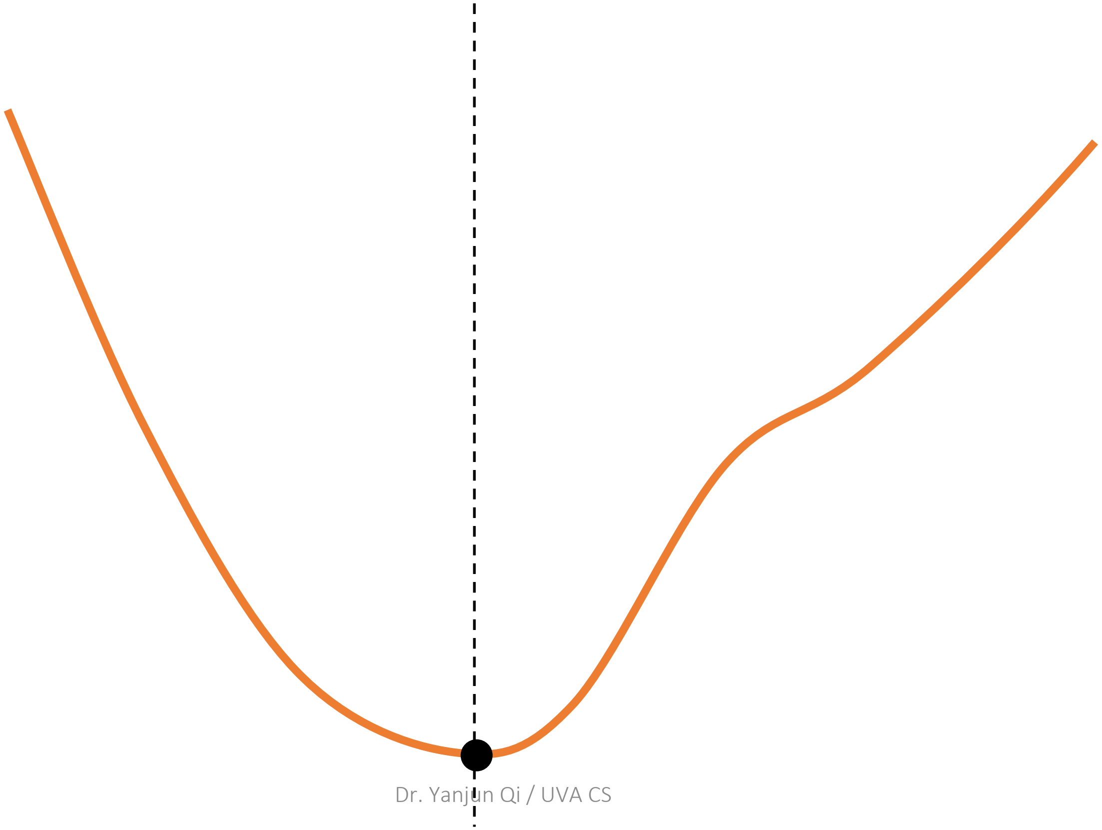
Newton's Method / second-order Taylor series approximation



Newton's Method / second-order Taylor series approximation



Newton's Method / second-order Taylor series approximation



Newton's Method

- At each step:

$$\theta_{k+1} = \theta_k - \frac{f'(\theta_k)}{f''(\theta_k)}$$

$$\theta_{k+1} = \theta_k - H^{-1}(\theta_k)\nabla f(\theta_k)$$

- Requires 1st and 2nd derivatives
- Quadratic convergence
- ➔ However, finding the inverse of the Hessian matrix is often expensive

Newton vs. GD for optimization

- Newton: a quadratic/second-order Taylor series approximation

$$\theta_{k+1} = \theta_k - \frac{1}{H(\theta_k)} g(\theta_k)$$

$$\hat{f}_{quad}(\boldsymbol{\theta}) = f(\boldsymbol{\theta}_k) + \mathbf{g}_k^T(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \mathbf{H}_k (\boldsymbol{\theta} - \boldsymbol{\theta}_k)$$

Finding the minimum solution of the above right quadratic approximation (quadratic function minimization is easy !)

$$\hat{f}_{quad}(\boldsymbol{\theta}) = f(\boldsymbol{\theta}_k) + \mathbf{g}_k^T(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \frac{1}{\alpha} (\boldsymbol{\theta} - \boldsymbol{\theta}_k)$$

$$\Downarrow$$

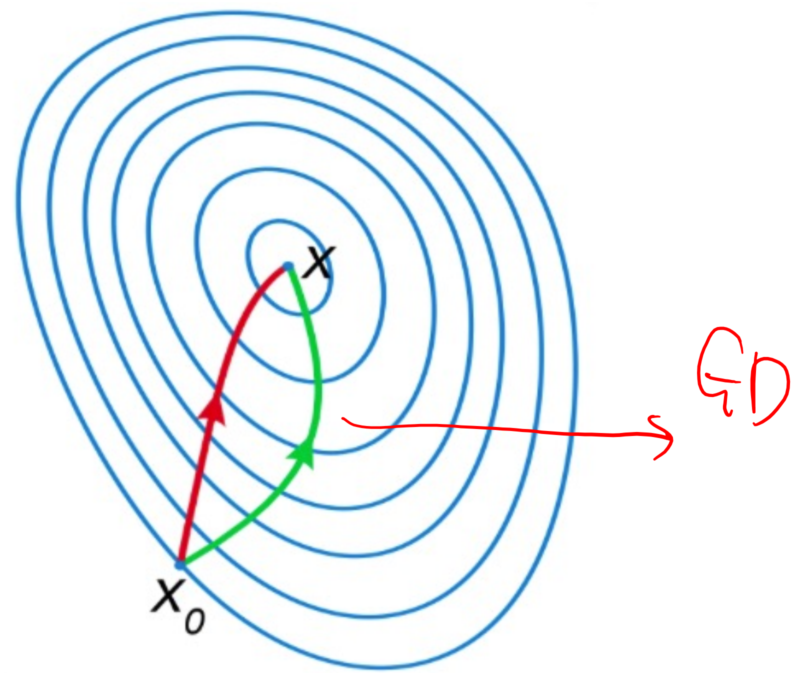
$$\theta_{k+1} = \theta_k - \alpha\, g(\theta_k)$$

Comparison

- Newton’s method vs. Gradient descent

A comparison of gradient descent (green) and Newton's method (red) for minimizing a function (with small step sizes).

Newton’s method uses curvature information to get a more direct route ...



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

$$\nabla_\theta J(\theta) = X^T X \theta - X^T \vec{y}$$

$$H = \nabla_\theta^2 J(\theta) = X^T X$$

$$\Rightarrow \theta^t = \theta^{t-1} - H^{-1} \nabla J(\theta^{t-1})$$

Newton

$$= \theta^{t-1} - (X^T X)^{-1} \left[X^T X \theta^{t-1} - X^T \vec{y} \right]$$

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

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

??? Normal Equation?

Newton's method for Linear Regression

References

- Big thanks to Prof. Eric Xing @ CMU for allowing me to reuse some of his slides
- http://en.wikipedia.org/wiki/Matrix_calculus
- Prof. Nando de Freitas's tutorial slide
- An overview of gradient descent optimization algorithms, https://arxiv.org/abs/1609.04747