DS 4400

Machine Learning and Data Mining I

Alina Oprea
Associate Professor
Khoury College of Computer Science
Northeastern University

September 29 2020

Outline

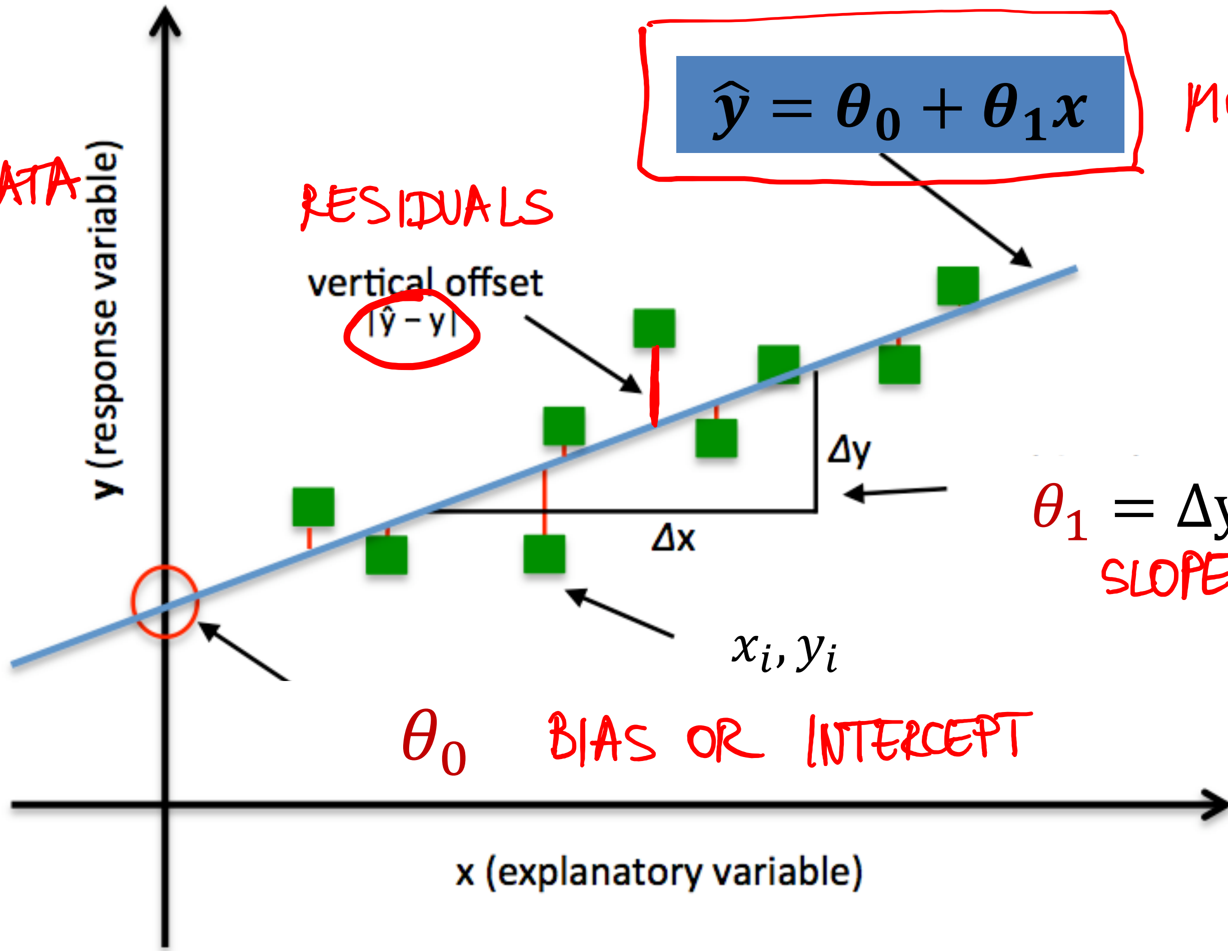
- Multiple linear regression
  - Lab in Python
  - Feature standardization
  - Outliers
- Gradient descent optimization
  - General algorithm
  - Instantiation for linear regression

Linear Regression

TRAINING DATA

x_i, y_i

$i = \overline{1, N}$



MODEL

$\theta_1 = \Delta y / \Delta x$

SLOPE

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

Min $\quad MSE = \frac{1}{N} \sum_{i=1}^{n} \left[h_\theta(x_i) - y_i \right]^2$

Solution for simple linear regression

- Dataset $x_i \in R, y_i \in R, h_\theta(x) = \theta_0 + \theta_1 x$
- $J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\theta_0 + \theta_1 x_i - y_i)^2$ MSE / Loss

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{2}{N}\sum_{i=1N}(\theta_0 + \theta_1 x_i - y_i) = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{2}{N}\sum_{i=1}^{N} x_i(\theta_0 + \theta_1 x_i - y_i) = 0$$

- Solution of min loss

$- \theta_0 = \bar{y} - \theta_1 \bar{x}$

$- \theta_1 = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} = \frac{Cov(x, y)}{Var(x)}$

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}$$

$$\bar{y} = \frac{\sum_{i=1}^{N} y_i}{N}$$

Multiple Linear Regression

Feature

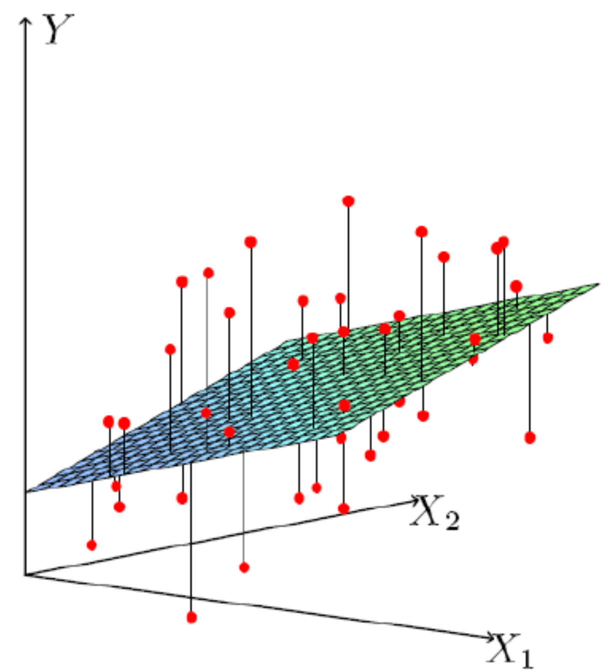
Matrix: $X = \begin{bmatrix} 1 & \\ 1 & \\ \vdots & \\ 1 & \end{bmatrix}$

Training example

- Dataset: $x_i \in R^d, y_i \in R$
- Hypothesis $h_\theta(x) = \theta^T x$
- MSE $= \frac{1}{N} \sum (\theta^T x_i - y_i)^2$ Loss / cost
- MSE is convex
- Unique minimum

$$\boxed{\theta = (X^\top X)^{-1} X^\top y}$$

Closed-form optimum solution for linear regression



Vectorization

- Two options for operations on training data
  - Matrix operations
  - For loops to update individual entries
- Most software packages are highly optimized for matrix operations
  - Python numpy
  - Preferred method!
- See Matthew's tutorial

- Matrix operations are much faster than loops!

Closed-form solution

- Can obtain $\boldsymbol{\theta}$ by simply plugging $\boldsymbol{X}$ and $\boldsymbol{y}$ into

$$\boldsymbol{\theta} = \boxed{(\boldsymbol{X}^\top \boldsymbol{X})^{-1}} \boldsymbol{X}^\top \boldsymbol{y}$$

$$X = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{i1} & \cdots & x_{id} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{Nd} \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_N \end{bmatrix}$$

X: size $n \times (d+1)$

$X^T . X$: size $(d+1) \times (d+1)$

- If $\boldsymbol{X}^\top \boldsymbol{X}$ is not invertible (i.e., singular), may need to:
  - Use pseudo-inverse instead of the inverse
    - In python, `numpy.linalg.pinv(a)`
  - Remove redundant (not linearly independent) features
  - Remove extra features to ensure that $d \leq n$

Def: Pseudo-inverse G:

$$\boxed{AGA = A}$$

$X^T X$ invertible when $\text{rank}(X^T X) = d+1$

If A is invertible $\Rightarrow$ $G = A^{-1}$

Multiple LR Lab

RESPONSE

```
# Multiple LR

#X_multi = pd.DataFrame(np.c_[boston['LSTAT'], boston['RM']], columns = ['LSTAT','RM'])
X_multi = boston.loc[:, boston.columns != 'MEDV']
Y = boston['MEDV']

X_m_train, X_m_test, Y_m_train, Y_m_test = train_test_split(X_multi, Y, test_size = 0.2, random_state=5)
print(X_m_train.shape)
print(X_m_test.shape)
print(Y_m_train.shape)
print(Y_m_test.shape)
```

```
(404, 13)
(102, 13)
(404,)
(102,)
```

13 FEATURES

```
mlr = LinearRegression()
mlr.fit(X_m_train, Y_m_train)
```

MLR :

```
LinearRegression()
```

Multiple LR Lab

```
coeff_df = pd.DataFrame(mlr.coef_, X_m_train.columns, columns=['Coefficient'])
coeff_df
```

	Coefficient
CRIM	-0.130800
ZN	0.049403
INDUS	0.001095
CHAS	2.705366
NOX	-15.957050
RM	3.413973
AGE	0.001119
DIS	-1.493081
RAD	0.364422
TAX	-0.013172
PTRATIO	-0.952370
B	0.011749
LSTAT	-0.594076

θ_1

θ_2

θ_d

$h(x) = \theta_0 + \sum \theta_i x_i$

INTERPRETABLE!

Simple vs Multiple LR

```
print(slr.intercept_)
print(slr.coef_)
```

```
-32.839129906011266
[8.82345634]
```

```
Y_train_predict = slr.predict(X_train)
mse = mean_squared_error(Y_train, Y_train_predict)

print("The model performance for training set")
print('MSE is {}'.format(mse))
```

```
The model performance for training set
MSE is 48.612648648611334
```

```
Y_m_train_predict = mlr.predict(X_m_train)
mse = mean_squared_error(Y_m_train, Y_m_train_predict)

print("The model performance for training set")
print("--------------------------------------")
print('MSE is {}'.format(mse))
print("\n")
```

```
The model performance for training set
--------------------------------------
MSE is 22.477090408387635
```

Simple vs Multiple LR

" RM " (1 FEATURE)

```
df_m = pd.DataFrame({'Actual': Y_train, 'Predicted simple': Y_train_predict, 'Predicted multi': Y_m_train_predict})
df_m.head(10)
```

	Actual	Predicted simple	Predicted multi
33	13.1	17.463395	13.828770
283	50.0	37.069115	44.528528
418	8.8	19.722200	3.915991
502	20.6	21.160423	22.377959
402	12.1	23.666285	18.235923
368	50.0	11.013448	25.523748
201	24.1	21.531008	29.439747
310	16.1	11.039918	18.694533
343	23.9	26.242734	27.856463
230	24.3	19.933962	24.644734

SIMPLE MULTIPLE

MULTIPLE LR FITS BETTER

Supervised Learning: Regression

Training

Data → Pre-processing → Feature extraction → Learning model

Labeled

$x_i, y_i \in R$

Normalization
Standardization

Feature
Selection

Regression

$f(x)$

Testing

New data → Learning model → Predictions

$y' = f(x') \in R$

Unlabeled

x'

$f(x)$

Response variable

Regression

Feature Standardization

$$X = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1j} & \cdots & x_{1d} \\ 1 & & & & & \\ \vdots & & & x_{ij} & & \\ 1 & x_{N1} & \cdots & x_{Nj} & \cdots & x_{Nd} \end{bmatrix}$$

FEATURE j

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{ij}$$

$$s_j = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_{ij} - \mu_j)^2}$$

EMPIRICAL ESTIMATION OF MEAN AND ST. DEV

$$x_{ij} \leftarrow \frac{x_{ij} - \mu_j}{s_j} \Rightarrow$$

MEAN ON EACH FEATURE 0

ST DEV 1

FEATURE SCALING

Feature Standardization

- Rescales features to have zero mean and unit variance
  - Let μ_j be the mean of feature j:
  - Replace each value with:

$$\boxed{x_{ij} \leftarrow \frac{x_{ij} - \mu_j}{s_j}}$$

for j = 1...d
(not x_0!)

    - s_j is the standard deviation of feature j

- μ_j & s_j COMPUTED ON TRAINING
- TESTING: USE μ_j & s_j FROM TRAINING

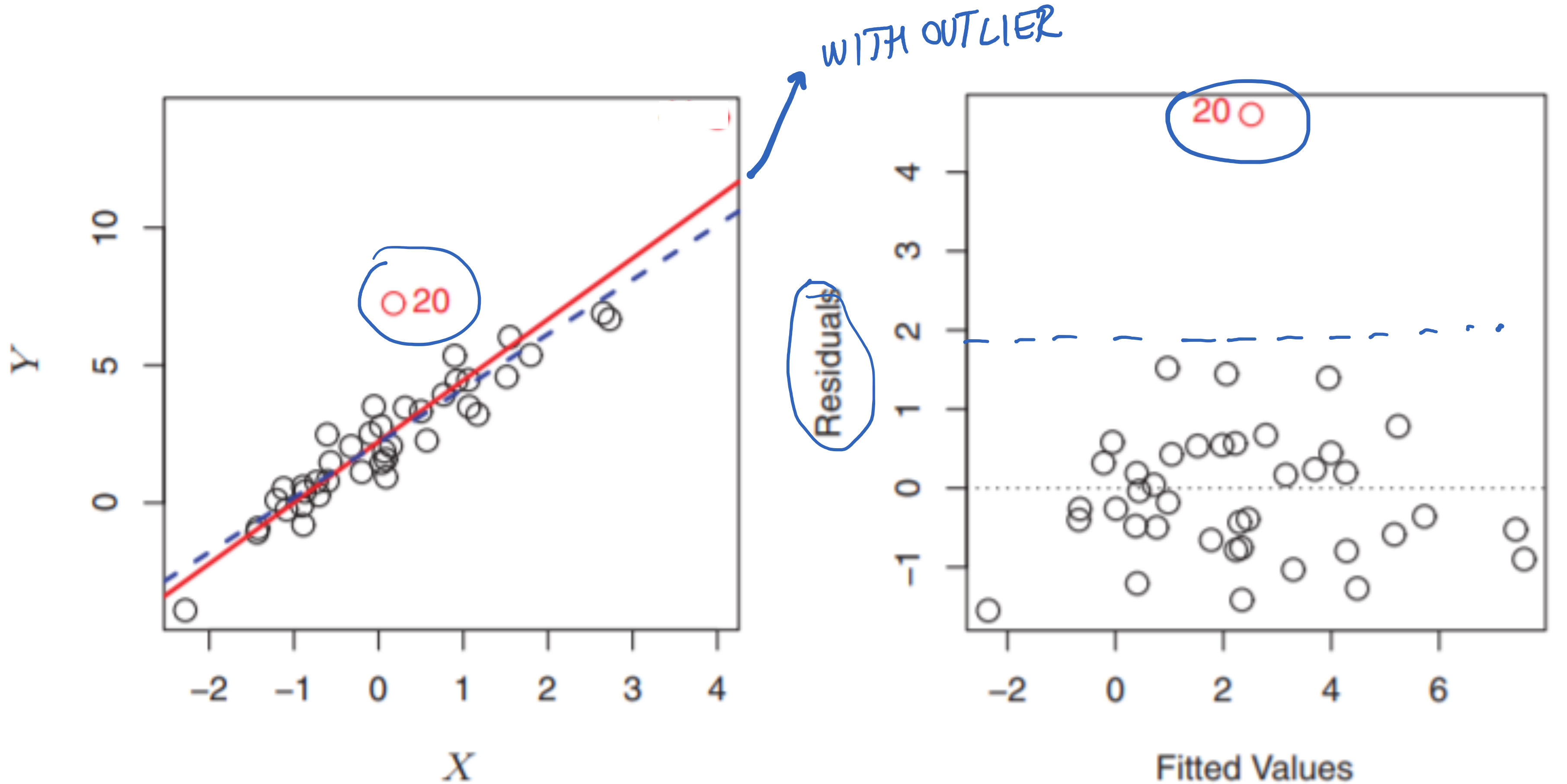
Other feature normalization

- Min-Max rescaling
  - $\boxed{x_{ij} \leftarrow \frac{x_{ij} - min_j}{max_j - min_j}}$ $\in [0,1]$
  - min_j and max_j: min and max value of feature j
- Mean normalization
  - $x_{ij} \leftarrow \frac{x_{ij} - \mu_j}{max_j - min_j}$ $\rightarrow$ Mean 0

Feature standardization/normalization

- Goal is to have individual features on the same scale
- Is a pre-processing step in most learning algorithms
- Necessary for linear models and Gradient Descent
- Different options:
  - Feature standardization
  - Feature min-max rescaling
  - Mean normalization

Practical issues: Outliers



- Dashed model is without outlier point
- Linear regression is not resilient to outliers!
- Outliers can be eliminated based on residual value
- Other methods to eliminate outliers (anomaly detection)

Categorical variables

- Predict credit card balance
  - Age
  - Income
  - Number of cards
  - Credit limit
  - Credit rating
- Categorical variables
  - Student (Yes/No)
  - State (50 different levels)

How to generate numerical representations of these?

Indicator Variables

- One-hot encoding
- Binary (two-level) variable
  - Add new feature $x_j = 1$ if student and 0 otherwise
- Multi-level variable
  - State: 50 values
  - $x_{MA} = 1$ if State $=$ MA and $0,$ otherwise
  - $x_{NY} = 1$ if State $=$ NY and $0,$ otherwise
  - …
  - How many indicator variables are needed?
- Disadvantages: data becomes too sparse for large number of levels
  - Will discuss feature selection later in class

How to optimize loss functions?

- Dataset: $x_i \in R^d, y_i \in R$
- Hypothesis $h_\theta(x) = \theta^T x$
- MSE $J(\theta) = \frac{1}{N}\sum (\theta^T x_i - y_i)^2$ Loss / cost
  - Strictly convex function (unique minimum)
- General method to optimize a multi-variate function
  - Practical (low asymptotic complexity)
  - Convergence guarantees to global minimum

GRADIENT DESCENT

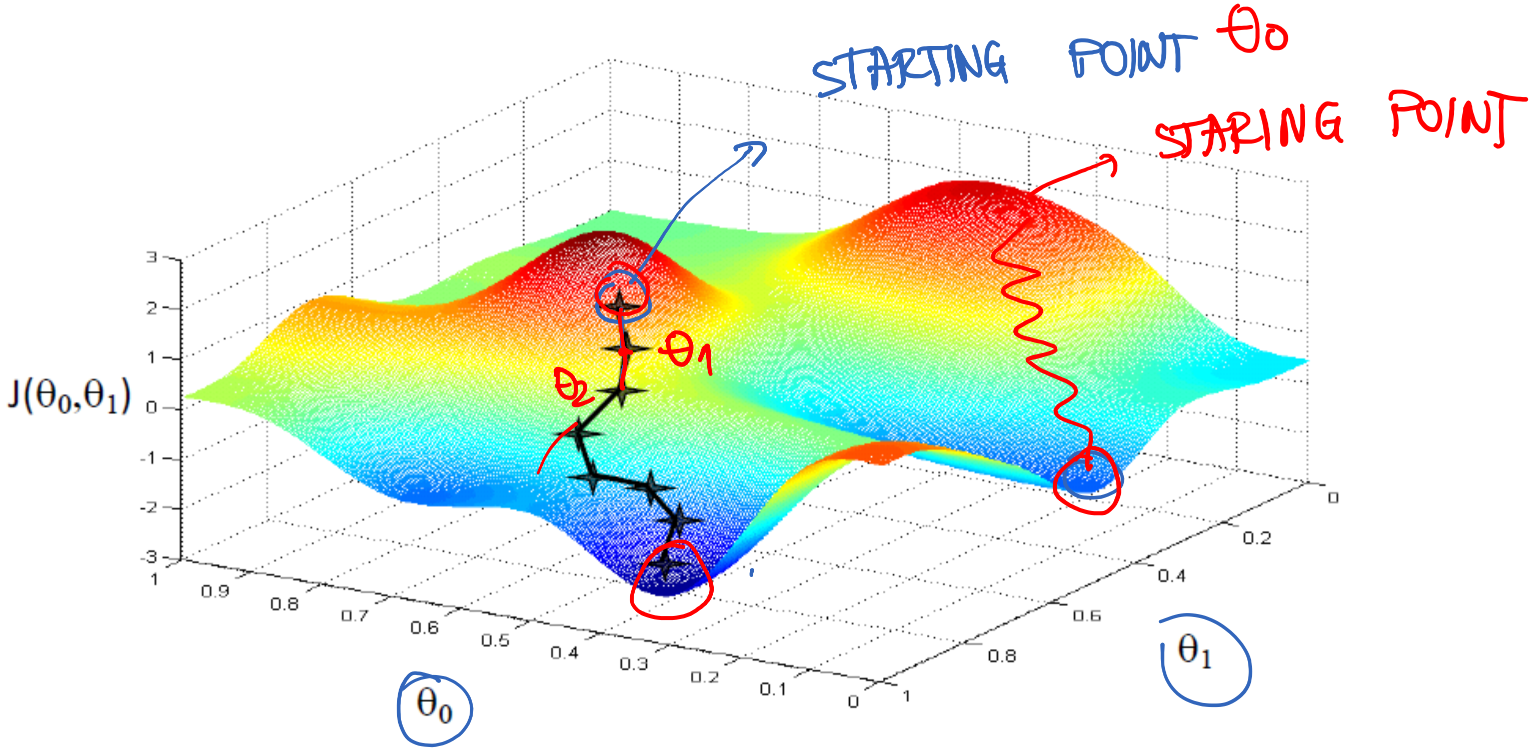
What Strategy to Use?

Follow the Slope

Follow the direction of steepest descent!

How to optimize $J(\theta)$?

- Choose initial value for θ
- Until we reach a minimum:
  - Choose a new value for θ to reduce $J(\theta)$



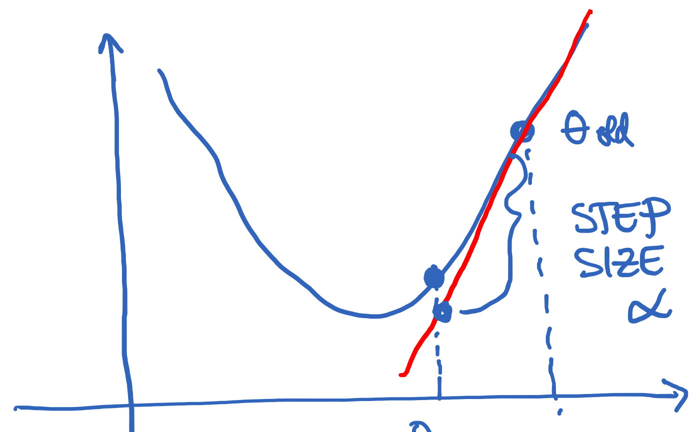
Gradient Descent

Goal: Min $J(\theta)$

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

- Initialize θ // RANDOM
- REPEAT UNTIL STOPPING CONDITION MET:

$$\theta_j \leftarrow \theta_j - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta_j} \; ; \; j = 0, 1, \dots, d$$



α: LEARNING RATE (STEP SIZE)

DIR. OF STEEPEST DESCENT: SLOPE OF TANGENT

GRADIENT OF FUNCTION $J(\theta)$ W.RESPECT TO θ

Gradient Descent

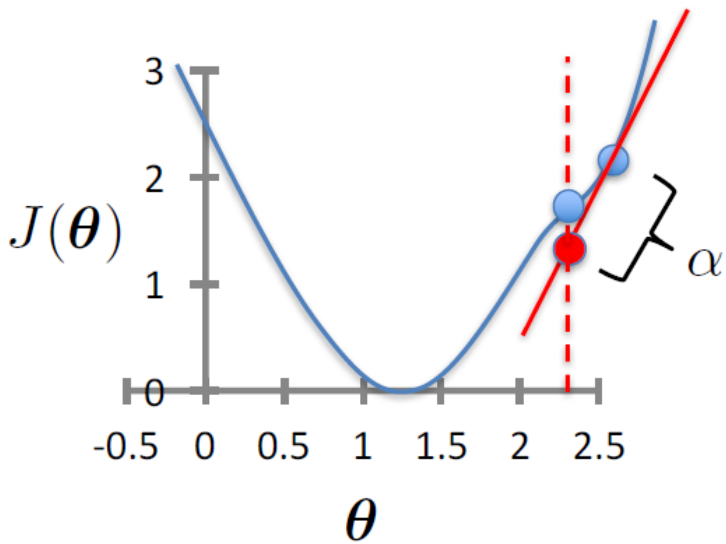
- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

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

simultaneous update
for j = 0 ... d

learning rate (small)
e.g., α = 0.05

$\alpha > 0$



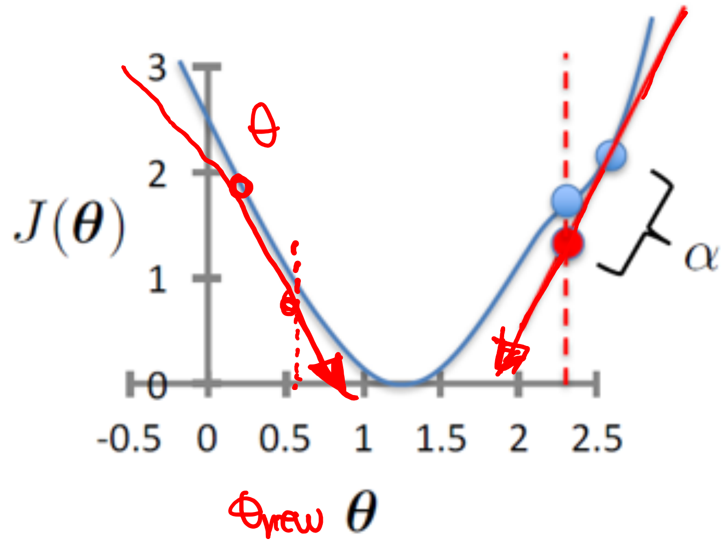
$$\boxed{\theta \leftarrow \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}}$$

VECTOR UPDATE RULE

Gradient Descent

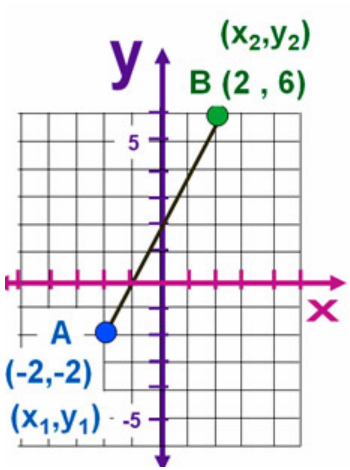
slope < 0

θ increases



slope > 0

θ decreases



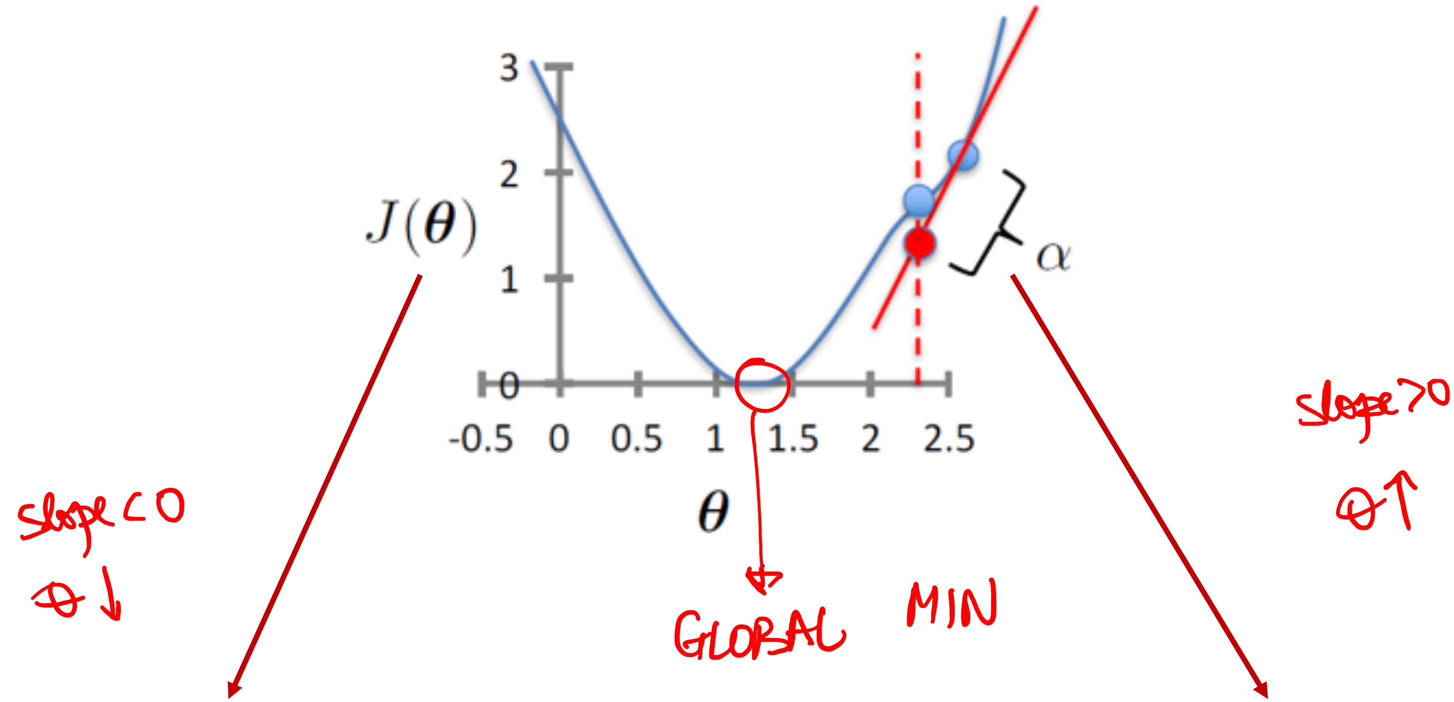
The Gradient "m" is:

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta Y}{\Delta X}$$

$$m = \frac{6 - {}^{-}2}{2 - {}^{-}2}$$

$$m = 8 / 4 = 2$$ ✓

Gradient Descent



In both cases θ gets closer to minimum

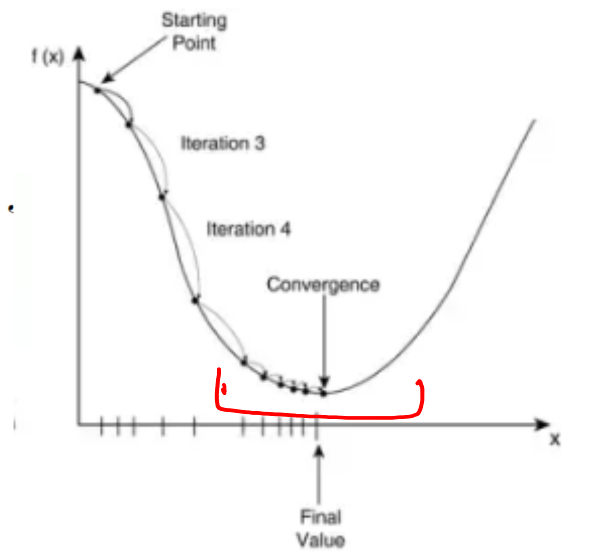
Gradient Descent

- 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 ... d

learning rate (small)
e.g., α = 0.05



- As you approach the minimum, the slope gets smaller, and GD will take smaller steps
- It converges to local minimum (which is global minimum for convex functions)!

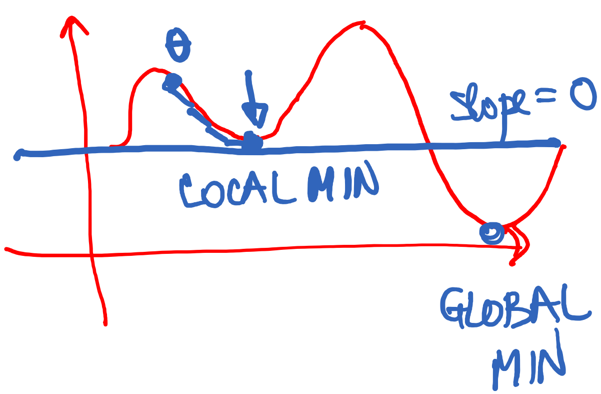
Gradient Descent

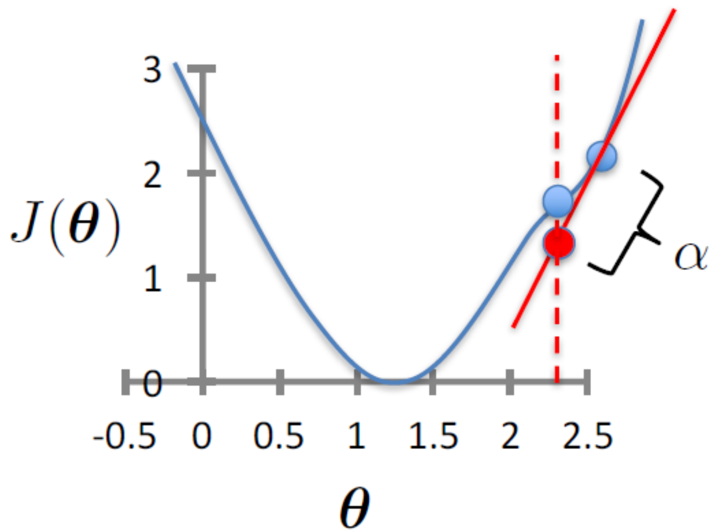
- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

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

simultaneous update
for j = 0 ... d

learning rate (small)
e.g., α = 0.05





- MSE CONVEX
  ⇓
  SINGLE MIN
  ⇓
  GD WILL CONVERGE

- What happens when θ reaches a local minimum?

CAN GET STUCK IN LOCAL MIN.!

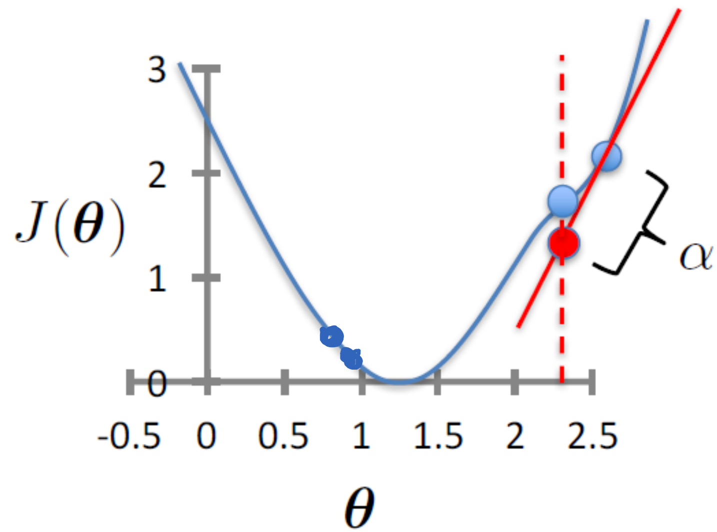
Stopping Condition

- 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 ... d

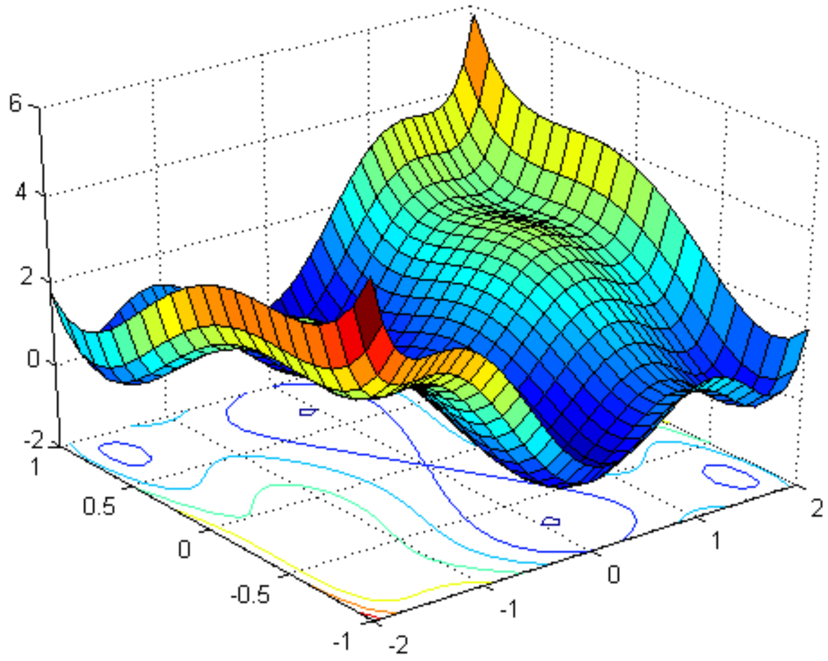
learning rate (small)
e.g., α = 0.05



- When should the algorithm stop?

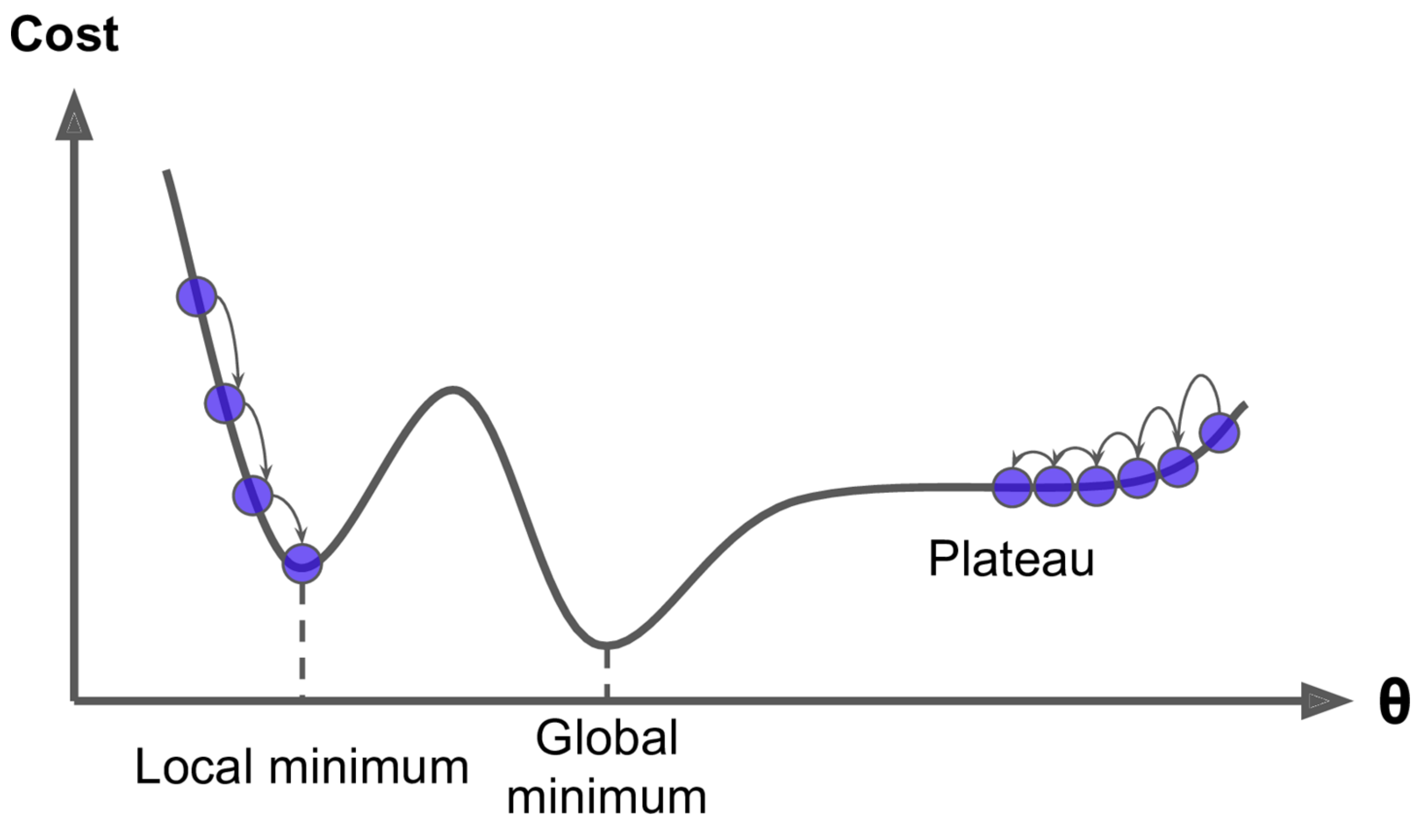
1) FIX NUMBER OF ITERATIONS
2) $\|\theta_{new} - \theta_{old}\| < \varepsilon$

Complex loss function

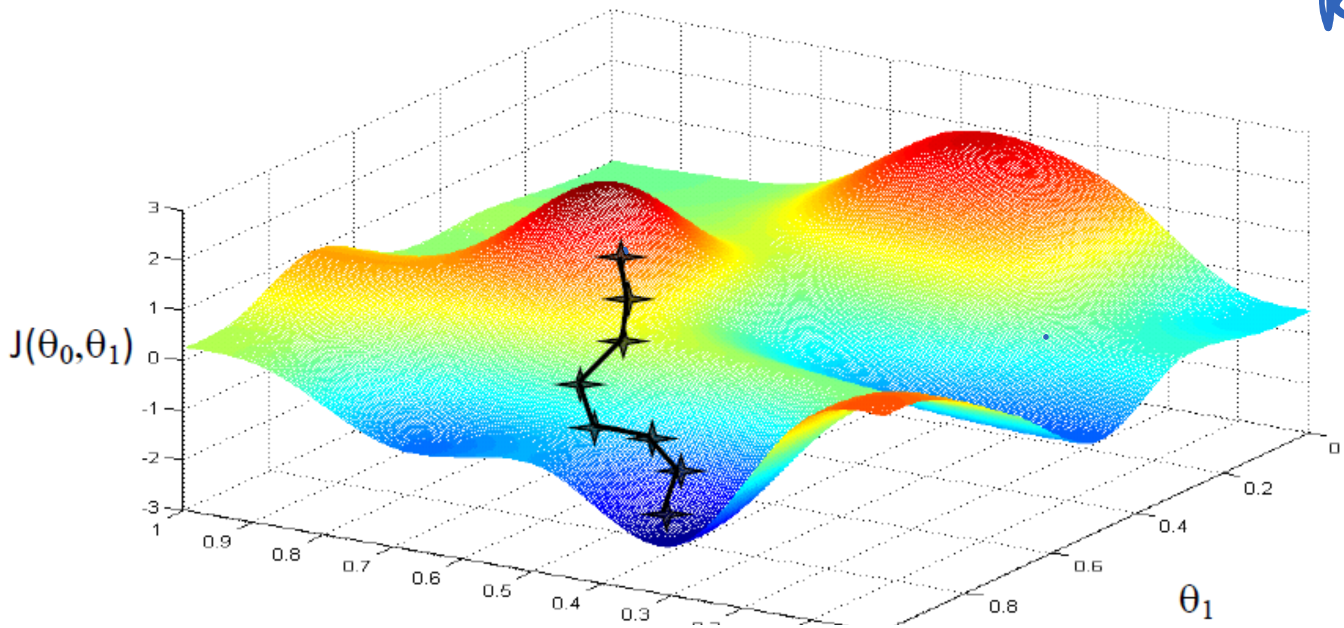


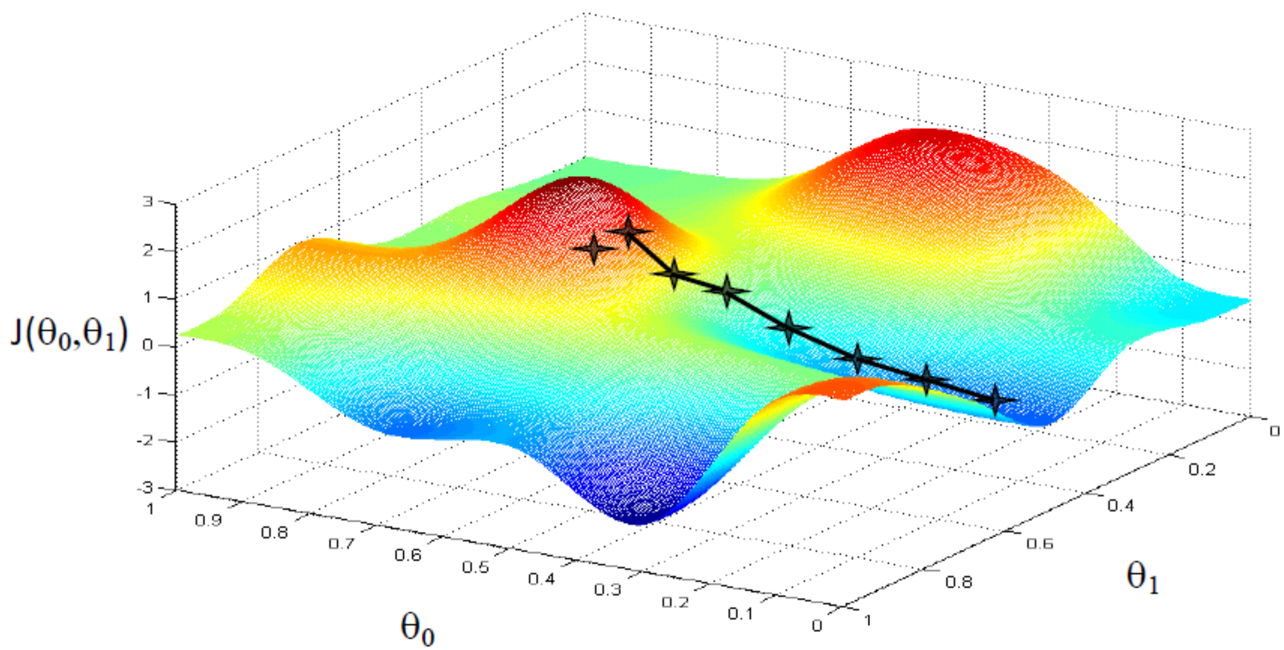
- Complex loss functions are more difficult to optimize

GD Convergence Issues



- Local minimum: Gradient descent stops
- Plateau: Almost flat region where slope is small





RUN MULTIPLE TIMES FROM RANDOM POINTS

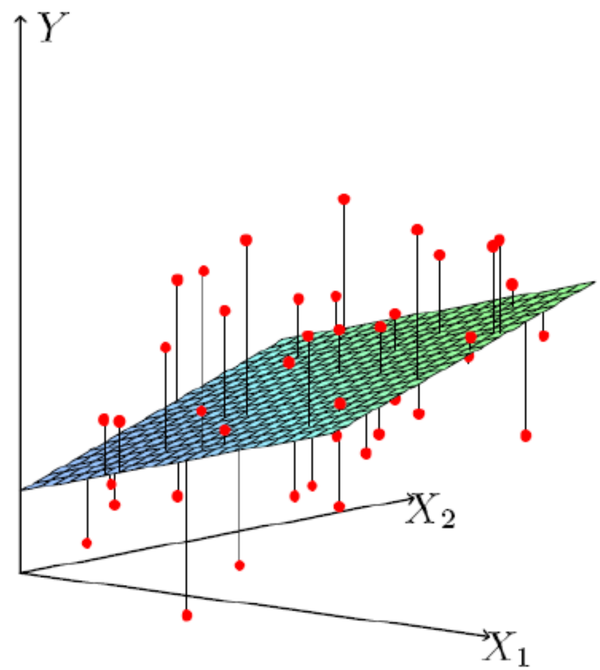
Multiple Linear Regression

- Dataset: $x_i \in R^d, y_i \in R$
- Hypothesis $h_\theta(x) = \theta^T x$
- MSE $= \frac{1}{N}\sum \underbrace{(\theta^T x_i - y_i)^2}$ Loss / cost

$J(\theta) =$

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

MSE is a strictly convex function and has unique minimum



GD for Multiple Linear Regression

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

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

simultaneous update
for j = 0 ... d

α IS THE SAME FOR ALL FEATURES

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} [h_\theta(x_i) - y_i]^2 \quad \text{MSE}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{N} \sum_{i=1}^{N} 2[h_\theta(x_i) - y_i] \cdot \boxed{\frac{\partial h_\theta(x_i)}{\partial \theta_j}} \quad x_{ij}$$

$$h_\theta(x_i) = \theta_0 + \theta_1 x_{i1} + \dots + \theta_d x_{id} \quad \boxed{\theta_j \cdot x_{ij}}$$

$$\frac{\partial h_\theta(x_i)}{\partial \theta_j} = x_{ij}$$

GD for Linear Regression

- Initialize θ
- Repeat until convergence

ALL TRAING EX
FEATURE j

ASSIGNMENT $\theta_j \leftarrow \theta_j - \alpha \boxed{\frac{2}{N}\sum_{i=1}^{N} \underbrace{(h_\theta(x_i) - y_i)}_{\text{Residual}} x_{ij}}$

simultaneous update for j = 0 ... d

Feature j

- To achieve simultaneous update
  - At the start of each GD iteration, compute $\boxed{h_\theta(x_i)}$ COMPUTE θold.
  - Use this stored value in the update step loop

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

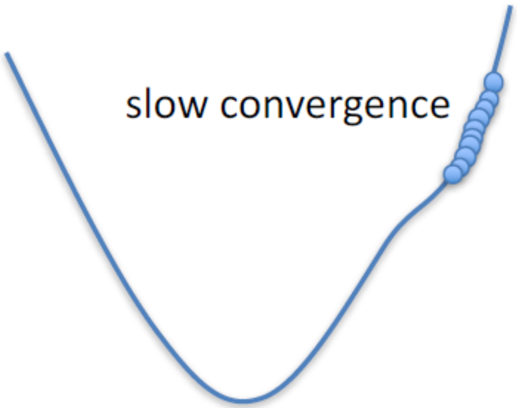
L$_2$ norm: $\|\boldsymbol{v}\|_2 = \sqrt{\sum_i v_i^2} = \sqrt{v_1^2 + v_2^2 + \ldots + v_{|v|}^2}$

$\theta_j^{new} = \theta_j^{old} - \ldots\ldots$

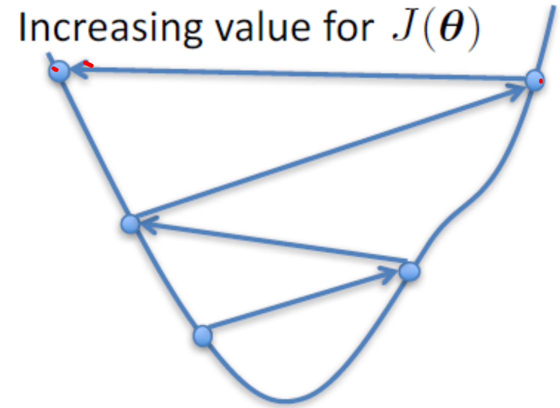
$\theta_j = \theta_j - \ldots\ldots$

Choosing learning rate

α too small



α too large

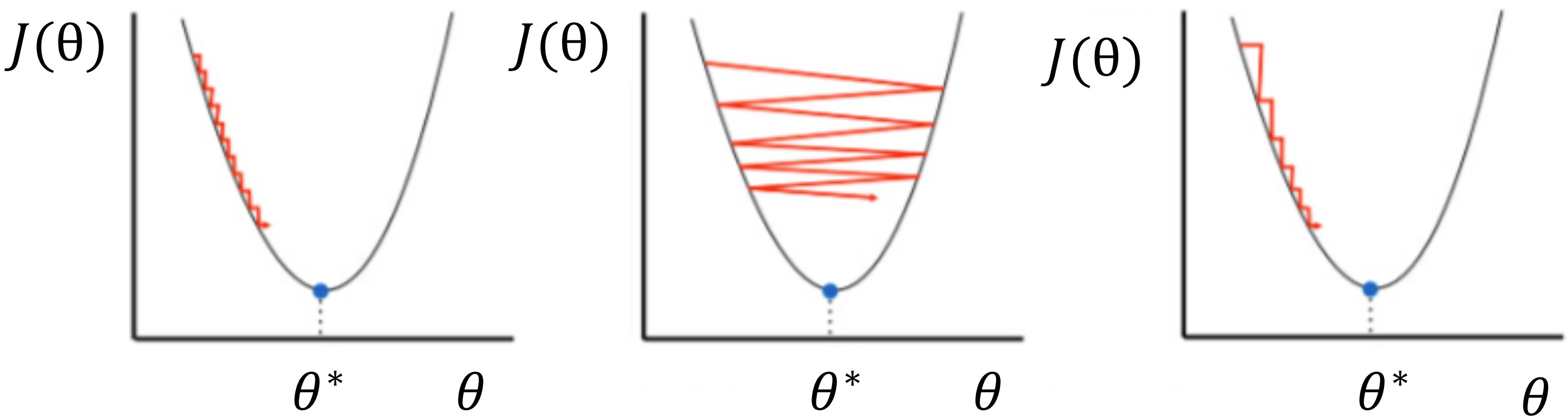


- May overshoot the minimum
- May fail to converge
- May even diverge

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

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

Adaptive step size

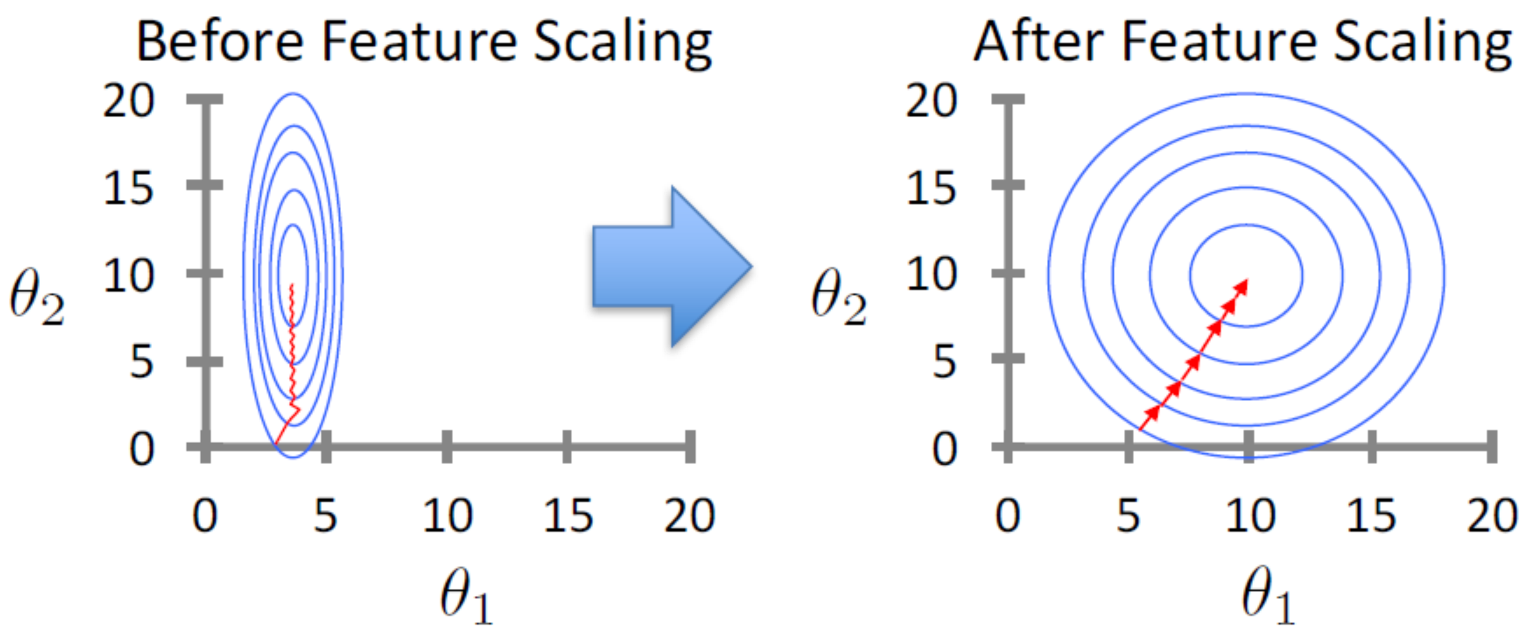


(a) Step-size too small (b) Step-size too big (c) Adaptive step-size

- Start with large step size and reduce over time, adaptively
- Line search method
- Measure how objective decreases

Feature Scaling

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



- Makes gradient descent converge *much* faster

Gradient Descent in Practice

- Asymptotic complexity
  - N is size of training data, d is feature dimension, and T is number of iterations
- Most popular optimization algorithm in use today
- At the basis of training
  - Linear Regression
  - Logistic regression
  - SVM
  - Neural networks and Deep learning
  - Stochastic Gradient Descent variants

BACK-PROPAGATION

Review Gradient Descent

- Gradient descent is an efficient algorithm for optimization and training ML models
  - The most widely used algorithm in ML!
  - Much faster than using closed-form solution for linear regression
  - Main issues with Gradient Descent is convergence and getting stuck in local optima (for neural networks)
- Gradient descent is guaranteed to converge to optimum for strictly convex functions if run long enough