Deep Learning

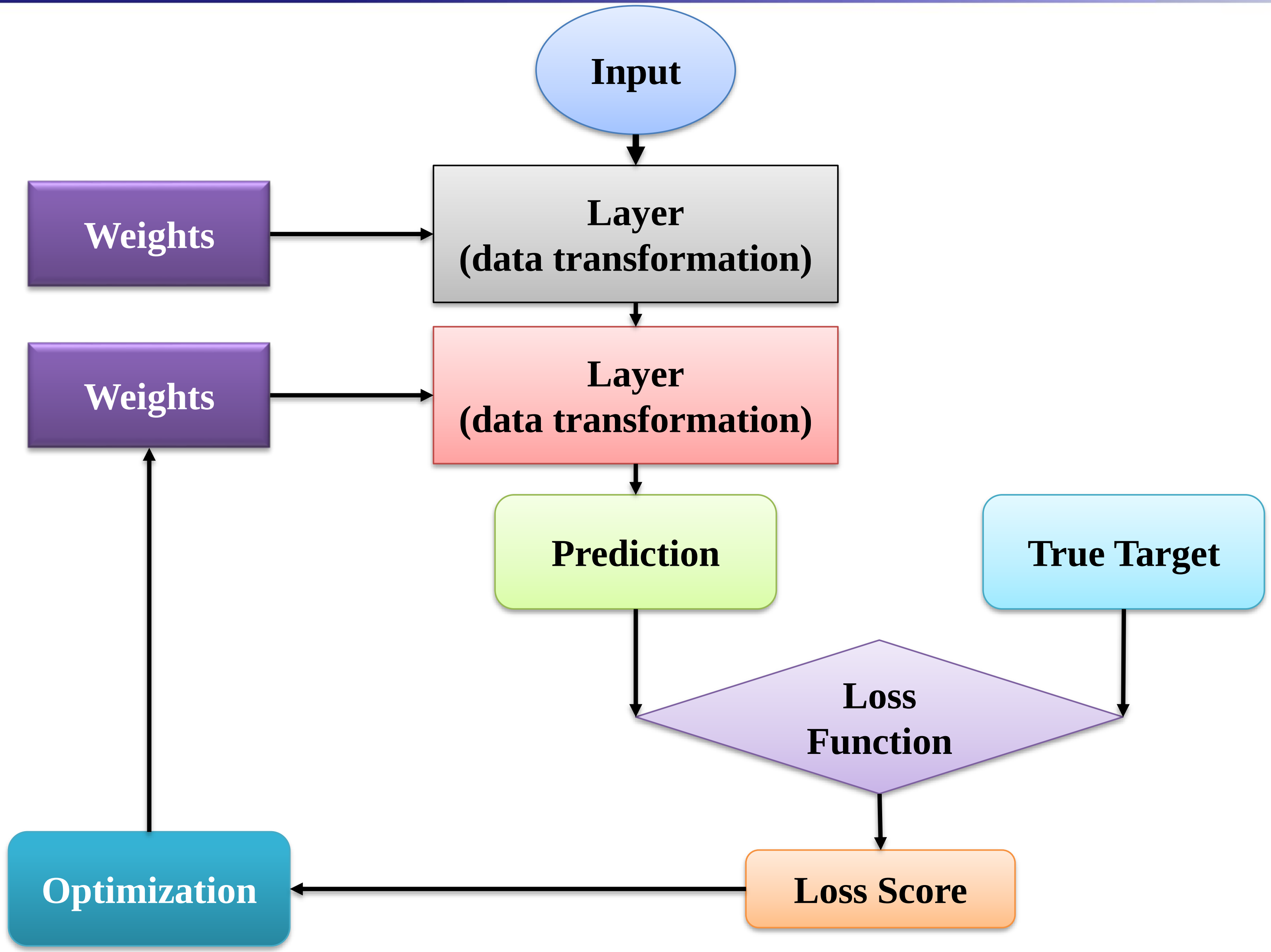
Md. Jalil Piran, PhD
Asst. Professor
Computer Science and Engineering
Sejong University
Spring, 2021

Outline

- Introduction to Deep Learning (DL)
- The History of DL
- Programming Tools
- Artificial Neural Networks
- Convolutional Neural networks
- **Optimization in DL**

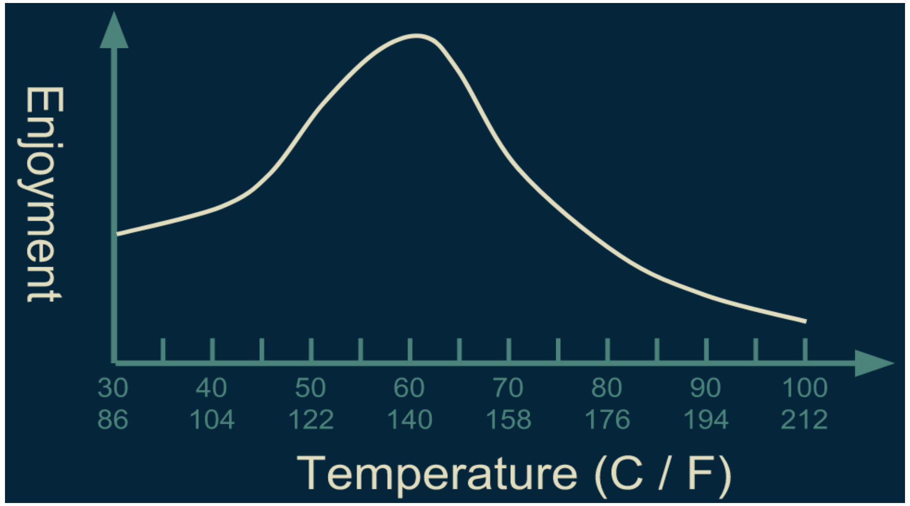
OPTIMIZATION IN DEEP LEARNING

Anatomy of a ML



$$ML = Data + Model + Optimization$$

- **Learning** = optimization over data
- **Optimization**: the problem of finding a set of inputs to an objective function that results in a maximum or minimum function evaluation.
- Example: tea; if it is too hot, cannot drink, if it is too cold, you don't like it



Backpropagation Representation

The Algorithm

- **Training set:** $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$
- **Set** $\Delta_{ij}^{(l)} = 0 \quad \forall\, l, i, j$
- **For** $i = 1$ to m do
  - Set $a^{(1)} = x^{(i)}$ *#the input*
  - **Perform forward propagation to compute** $a^{(l)}$ for $l = 1, 3, \dots, L$
  - Using $y^{(1)}$ compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ *#there is no $\delta^{(1)}$ because there is no error in the input layer*
  - $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
- **Find** the partial derivative:
  - $\begin{cases} j \neq 0 \Rightarrow D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} \\ j = 0 \Rightarrow D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} \end{cases}$

$\Rightarrow$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

 # partial derivative of the cost function
(Can be used in an optimization algorithm)

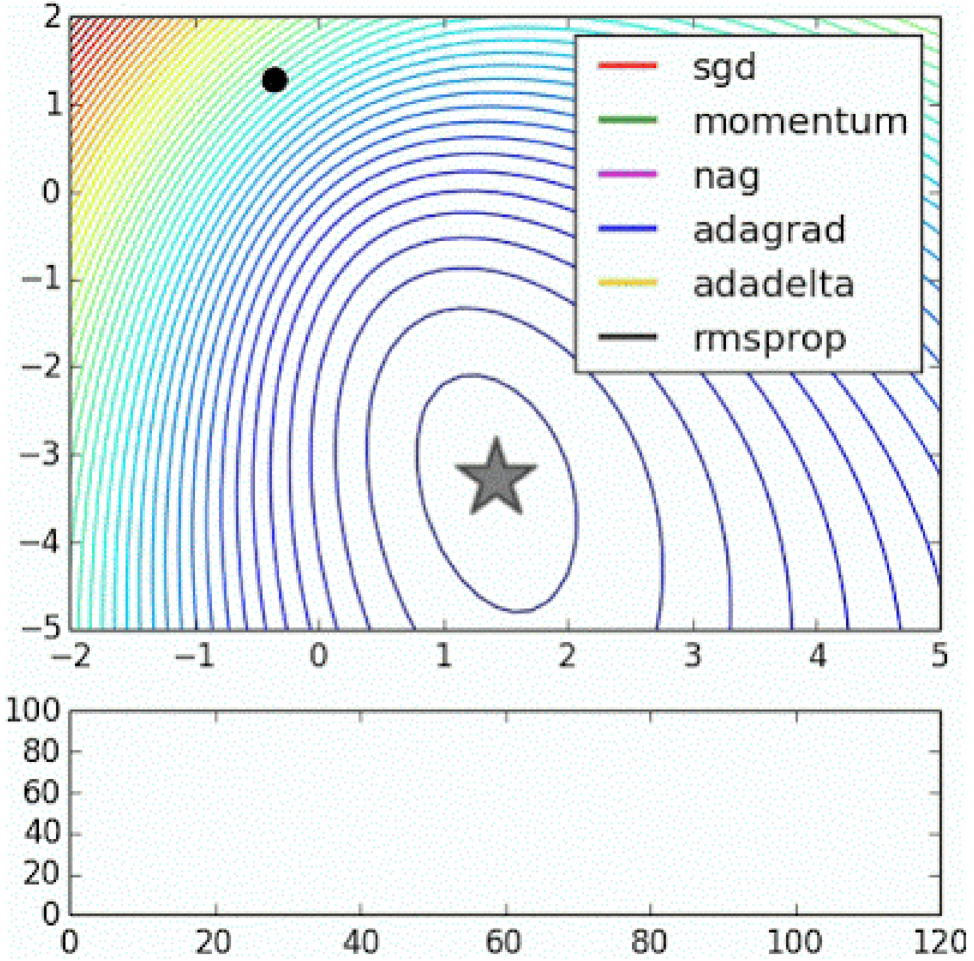
Parameter

- **Parameters Initialization**
  - The parameters need to be initialized prior to the network training.
  - Initialize the weights randomly
  - Initialize the biases to zero.

- **Parameters Updates**
  - e.g. optimization in ANNs, how to update the weights based on the loss function?
  - Learning rate, α

Optimizers

- Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Stochastic Gradient Descent
- Momentum
- Adagrad
- RMSProp
- Adadelta
- Adam
- …



https://imgur.com/s25RsOr

Gradient Descent

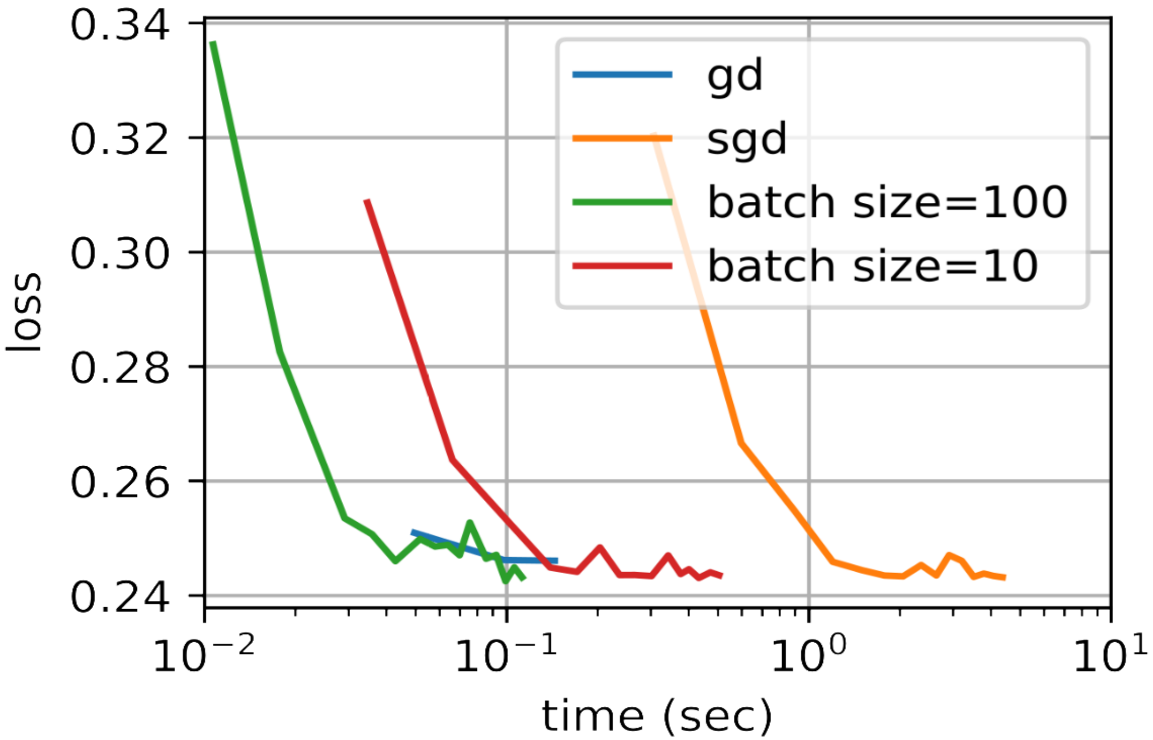
- By far the most common way to **optimize ANNs**.
- To **minimize** an objective function $J(\theta)$.
- By **updating the parameters** in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)\ w.r.t$ to the parameters.

$$\boldsymbol{\theta = \theta - \alpha . \nabla J(\theta)}$$

- **Learning rate**: the size of the steps to reach a (local) minima.
- **Advantages**:
  - Easy computation
  - Easy to implement
  - Easy to understand
- **Disadvantages**:
  - May trap at local minima
  - Changes in weights after gradient computation
  - Requires large memory

Gradient Descent

- Variants in terms of how much data need to compute the gradient of the objective function:
  - **Batch gradient descent**
  - **Stochastic gradient descent**
  - **Mini-batch gradient descent**



Gradient Descent

- **Batch Gradient descent**:
  - Inject all data at once.
  - A high risk of getting stuck
  - For ANNs, it is better to have an input with some randomness.

Gradient Descent

- **Stochastic gradient descent**:
  - A single random sample is introduced on each iteration.
  - The gradient is calculated for that specific sample only.
  - Implying the introduction of the desired randomness
  - Making more difficult the possibility of getting stuck

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\alpha}.\boldsymbol{\nabla J}\big(\boldsymbol{\theta}; \boldsymbol{x}(\boldsymbol{i}); \boldsymbol{y}(\boldsymbol{i})\big)$$

  - $\boldsymbol{x}(\boldsymbol{i})$,$\boldsymbol{y}(\boldsymbol{i})$ training samples

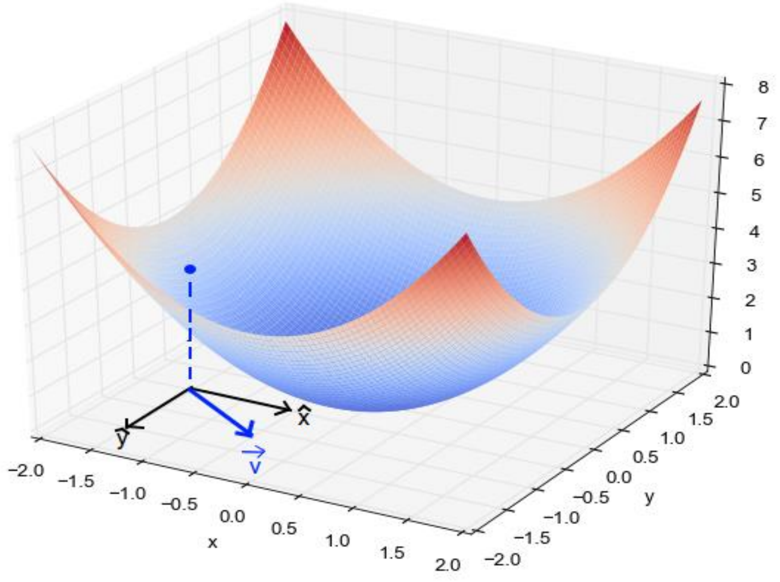
Gradient Descent

- **Mini-batch gradient descent**:
  - N random items are introduced on each iteration.
  - Getting faster training due to the parallelization of operations.
  - Cost function is calculated for each mini-batch.
  - Calculate the gradient as the multi-variable derivative of the cost function with respect to all the network parameters.
    - e.g. the slope of the tangent line to the cost function at a given point.

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\alpha} . \nabla \boldsymbol{J}\left(\boldsymbol{\theta}; \boldsymbol{B}(\boldsymbol{i})\right)$$

  - $\boldsymbol{B}(\boldsymbol{i})$ the batches of training samples

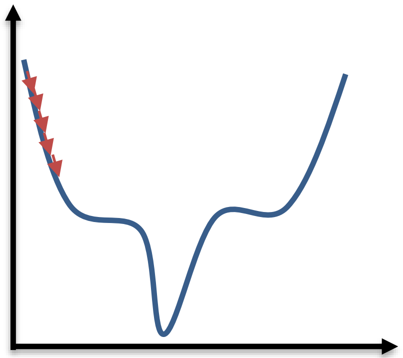
Gradient Descent



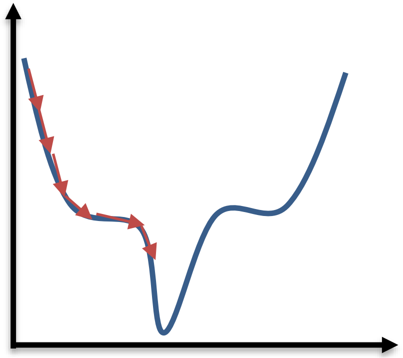
- Example: Consider a network with 2 parameters
- The cost function is in 3D
  - X: parameter 1
  - Y: parameter 2
  - Z: cost/loss value
- **Gradient**: a vector with 2 components on X and Y
- Update network parameters by subtracting the corresponding gradient value from their current value, multiplied by a learning rate
- Learning rate is to adjust the magnitude of the steps.
- Repeat all these steps as long as the loss value and the output metrics don't start to steadily worsen.

Gradient Descent

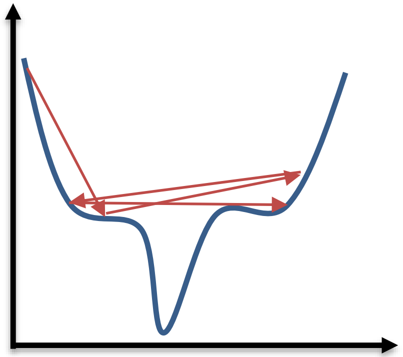
- **Learning rate**
- $\alpha \in [0, \infty)$



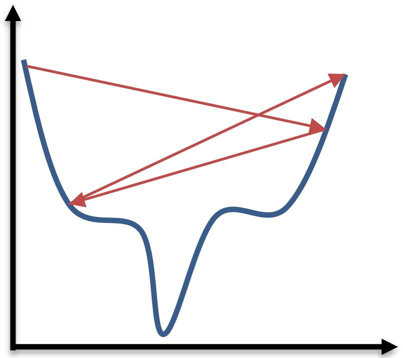
too small α



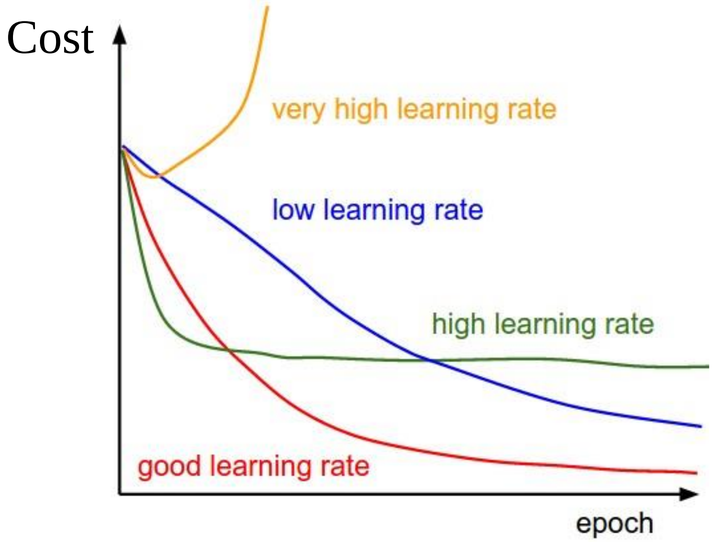
Good α



Large α

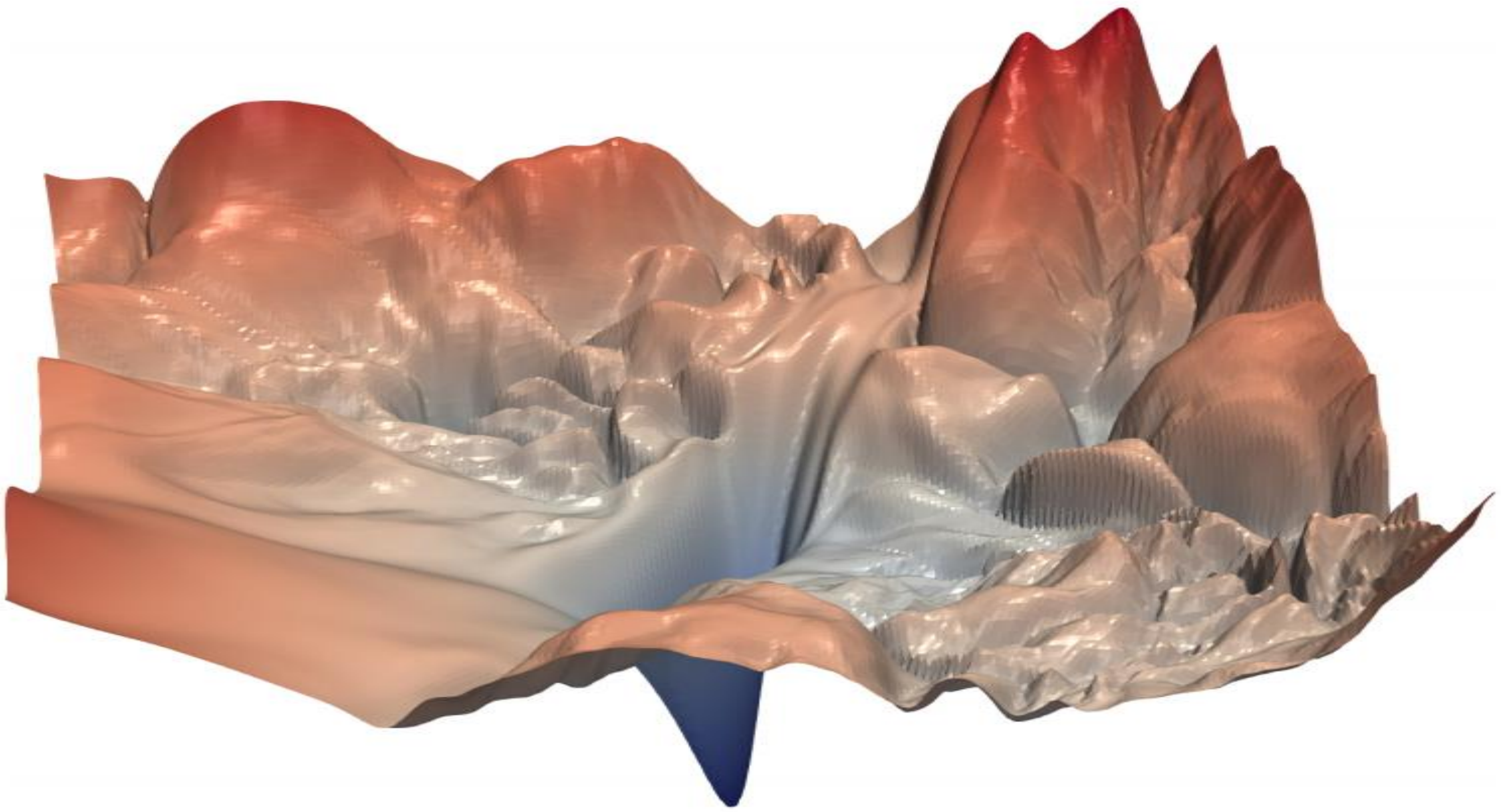


Too Large α



Gradient Descent

- However, it is not always two parameters!



Gradient Descent

- To compute the gradient:
  - **Numerical gradient**
    - Slow,
    - Not accurate e.g. returns an approximate,
    - Easy
    - Just use to be validate the results of the other ways
  - **Analytic gradient**
    - Fast,
    - Exact,
    - More error-prone
    - Used in practice

Gradient Descent

- **Numerical gradient**:
  - In a 1D space
  - e.g. the slope
  - $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$

- In a multi dimension, the gradient is a vector of partial derivatives along each dimension.
- Example) For a function: $f(u_1, u_2, u_3)$
- Gradient in the curvilinear coordinates: $\nabla\phi = \frac{1}{h_1}\frac{\partial\phi}{\partial \mathrm{u}_1}\hat{u}_1 + \frac{1}{h_2}\frac{\partial\phi}{\partial \mathrm{u}_2}\hat{u}_2 + \frac{1}{h_3}\frac{\partial\phi}{\partial \mathrm{u}_3}\hat{u}_3$
- Gradient in Cartesian coordinate: $\nabla f = \frac{\partial f}{\partial \mathrm{x}} i + \frac{\partial f}{\partial \mathrm{y}} j + \frac{\partial f}{\partial \mathrm{z}} k$

Gradient Descent

- **Numerical gradient**:
  - Example:

$\boldsymbol{\theta_1}$	$\boldsymbol{\theta_1 + h}$	**Gradient Vector** $\boldsymbol{d(\theta_1)}$
0.34	0.34+0.0001	-2.5
-1.11	-1.11	
0.78	0.78	
0.12	0.12	
0.55	0.55	
2.81	2.81	
-3.10	-3.10	
-1.50	-1.50	
0.33	0.33	
…	…	
Cost: 1.25347	**Cost**: 1.25322	

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$\frac{(1.25322 - 1.25347)}{0.0001} = -2.5$$

Gradient Descent

- **Numerical gradient**:
  - Example:

$\boldsymbol{\theta_1}$	$\boldsymbol{\theta_1 + h}$	Gradient Vector $\boldsymbol{d(\theta_1)}$
0.34	0.34	-2.5
-1.11	-1.11+0.0001	-0.60
0.78	0.78	
0.12	0.12	
0.55	0.55	
2.81	2.81	
-3.10	-3.10	
-1.50	-1.50	
0.33	0.33	
…	…	
Loss: 1.25347	Loss: 1.25353	

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$\frac{(1.25353 - 1.25347)}{0.0001} = -0.6$$

- Repeat for the other records.

Gradient Descent

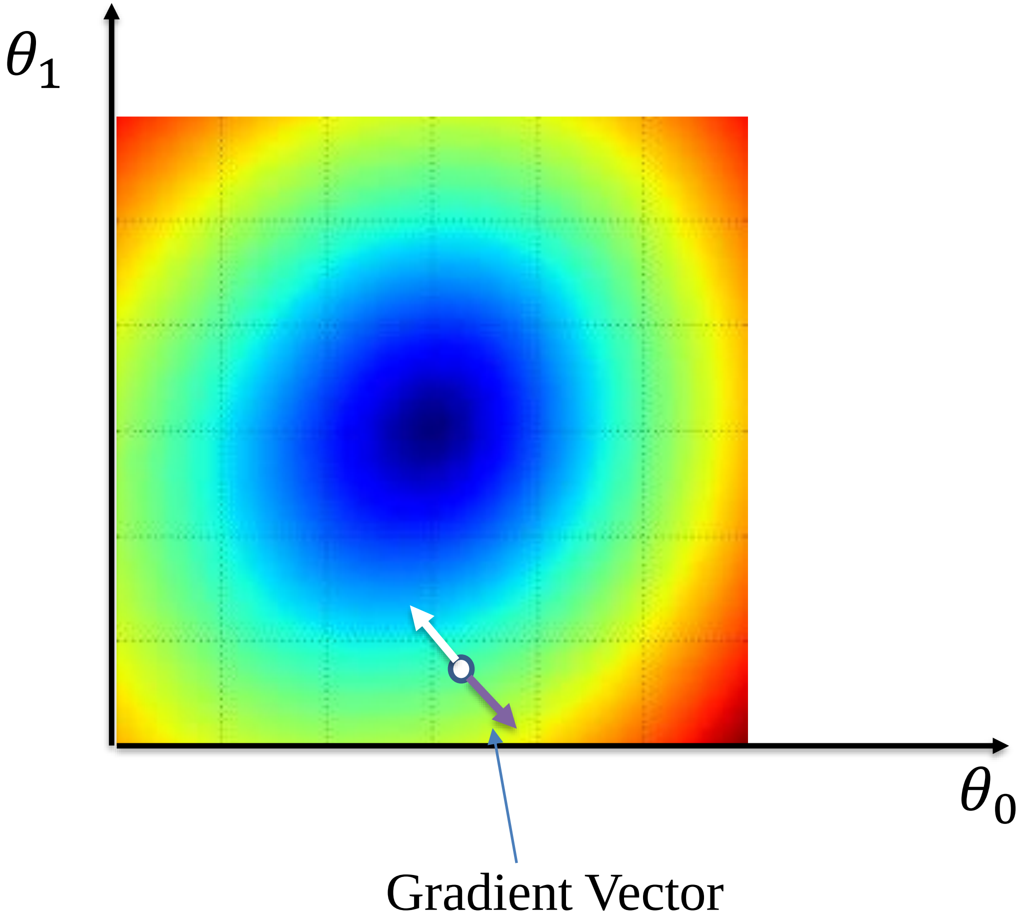
- **Analytic gradient**:
  - Using Calculus
  - By deriving a direct formula for the gradient (no approximations)
  - Compute the gradient of the loss function:

- $J(\theta) = \frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\theta(x^{(i)}))_k + \left(1 - y_k^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)_k\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\theta_{ji}^{(l)}\right)^2$
- $J(\theta_1) = \ldots$
- $\nabla_\theta J(\theta) = \cdots$

$$
\begin{aligned}
&while\ \textbf{True}:\\
&\qquad gradient = evaluat\_gradient(cost\_fun, data, weithts)\\
&\qquad weights \mathrel{+}= -(step\_size\ * gradient) \quad \#paramter\ update
\end{aligned}
$$

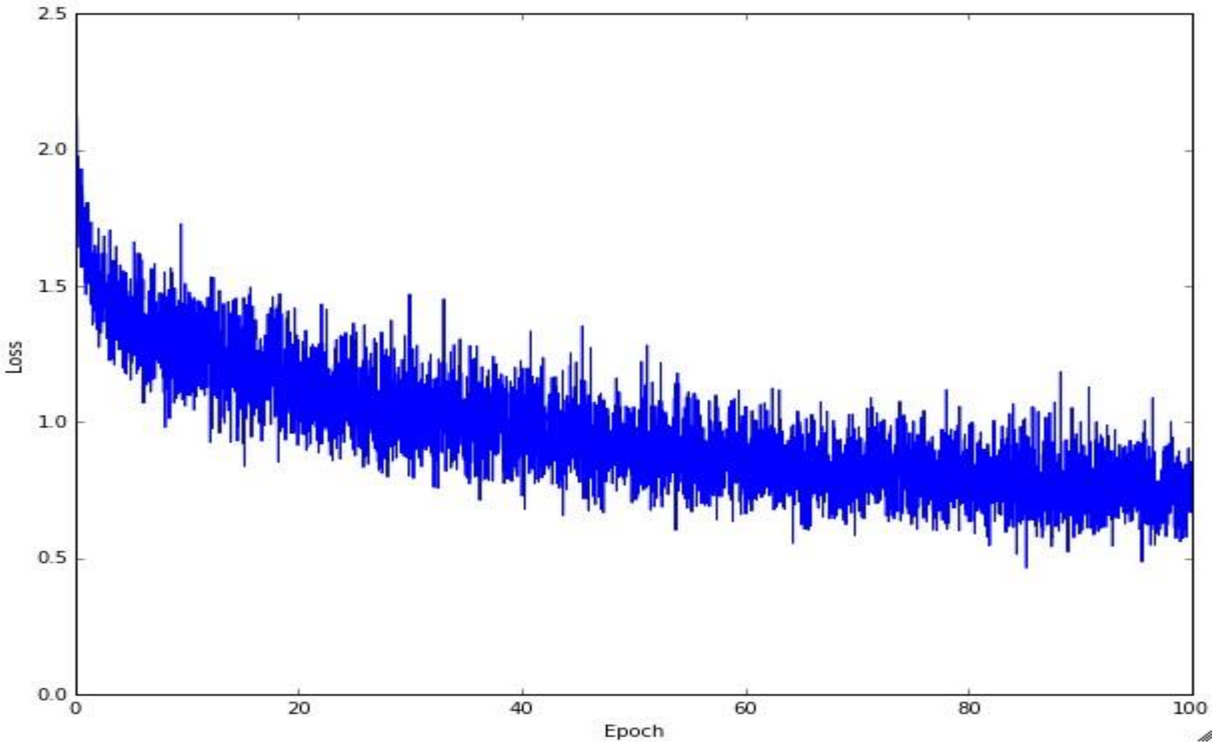
Gradient Descent

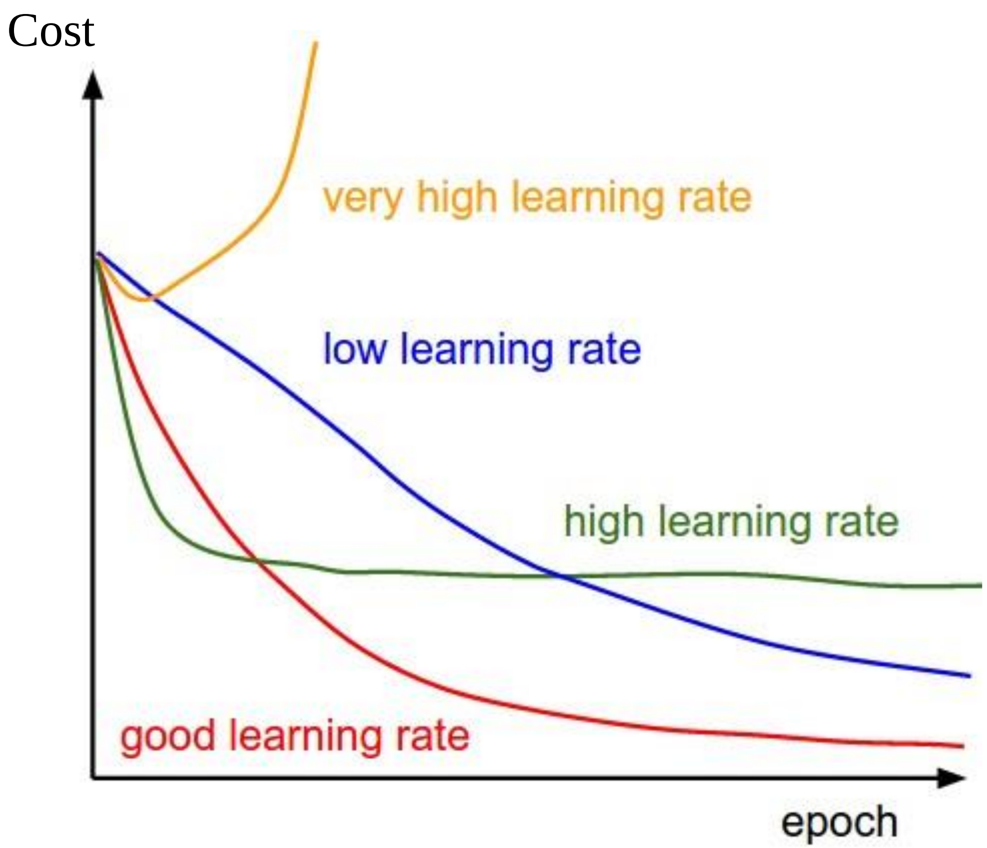
- Analytic gradient:
  - Move in opposite direction of the gradient vector



Gradient Descent

- **Mini-batch GD**:
  - Improves the optimization of the weights
  - Cost decreases over the time





Gradient Descent

- **Stochastic Mini-batch GD**:
  - SGD: rather than computing the loss and gradient over the entire training set, instead at every iteration, we sample small set of training examples [mini-batch]
  - Then, use those mini-batches to compute of the full sum and estimate the true gradient
  - Batch size: 32, 64, 128, 256

$while\ \boldsymbol{True}:$

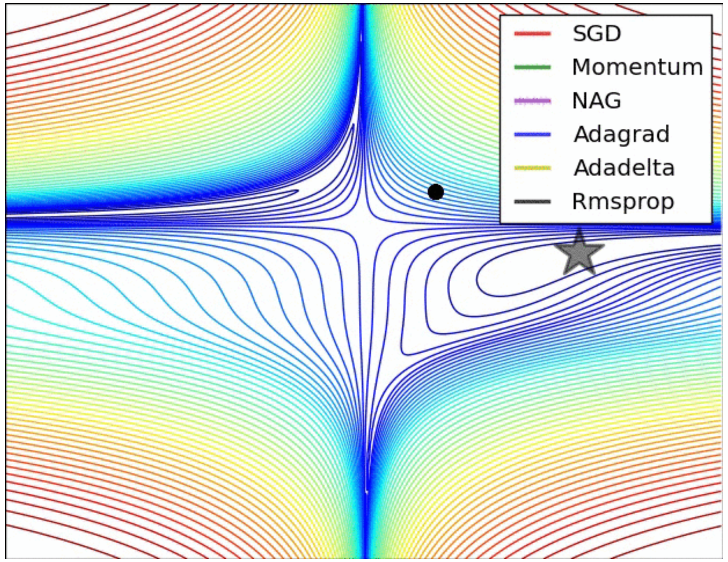
$$data\_batch = sample\_training\_data(data, 256)$$

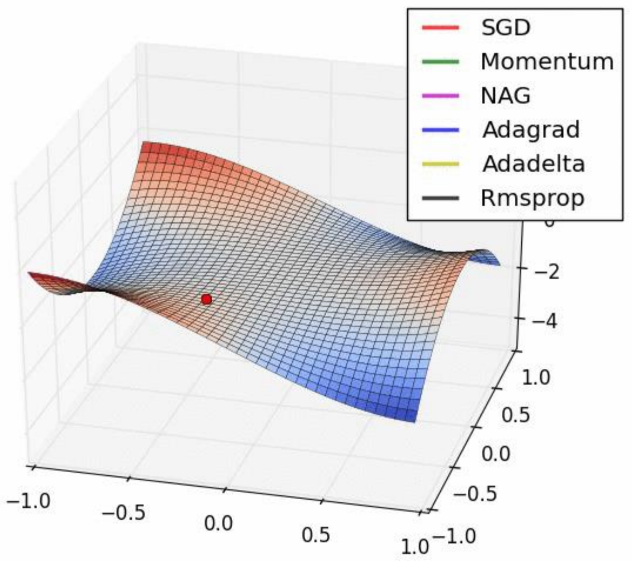
$$gradient = evaluat\_gradient(cost\_fun, data, weithts)$$

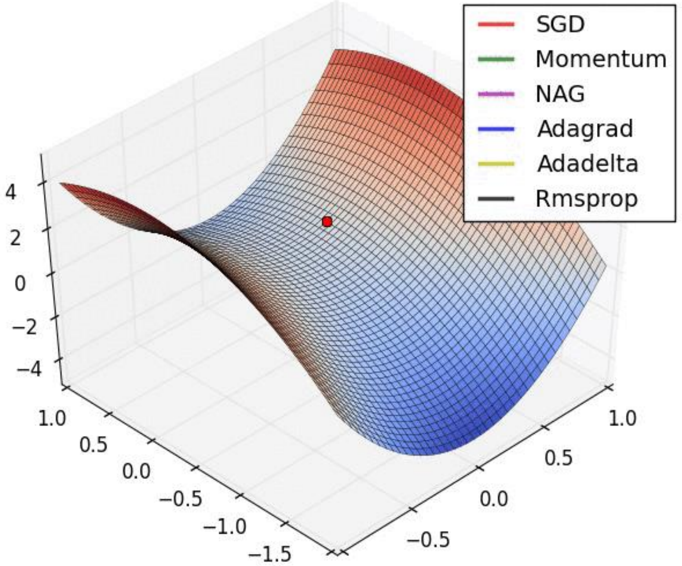
$$weights \mathrel{+}= -(step\_size\ * gradient) \quad \#update\ weights$$

Gradient Descent

- GD algorithms:
  - Momentum
  - Nesterov accelerated gradient
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam
  - Adamax
  - Nadam
  - AMSGrad







Gradient Descent

- **Momentum**
  - It gives a kind of 'inertia' to the process of moving through GD.
  - Updating the network parameters by adding an extra term
  - The term considers the value of the last iteration update,
  - So the previous gradients will be taken into account in addition to the current one.
  - When a motion vector at time t, the motion is:

$$v_t = \gamma . \Delta v_{t-1} - \eta_t . \nabla_v J(\theta)$$

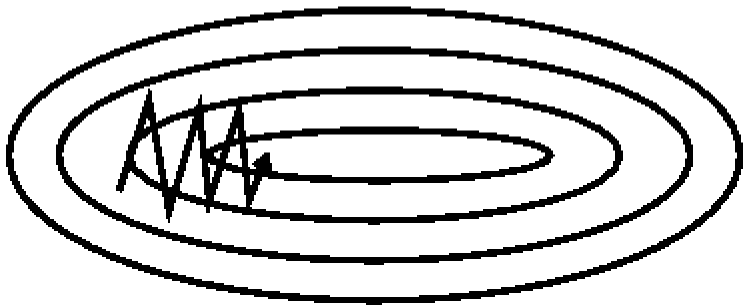
$$\theta = \theta - v_t \quad \text{\#weight update}$$

γ: the momentum term, 0.9

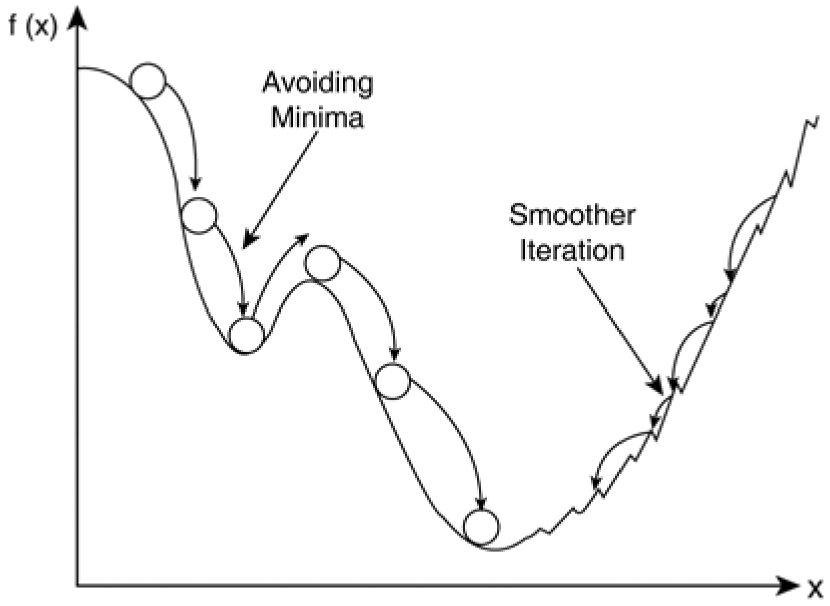
v: the moving term, e.g. how much it moved in the past

Gradient Descent

- **Momentum**
  - Solve the issue when SGD undergoes oscillation.
  - Momentum applies inertia in the direction of the frequent movement.



SGD



Momentum

Gradient Descent

- **Nestrov Accelerated Gradient (NAG)**
  - Based on the Momentum method
  - Considers the momentum step first,
  - If so, then moves the gradient step by obtaining the gradient at that location

$$v_t = \gamma v_{t-1} \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

  - Moves more effectively than the momentum

Gradient Descent

- **Adaotive Gradient (Adagrad)**
  - Moving by setting the step size differently for each variable when update variables.
  - Increases the step size for variables that have not changed much so far
  - Reduce the step size for variables that have changes much so far

$$G_t = G_{t-1} + \left(\nabla_\theta J(\theta_t)\right)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta J(\theta_t)$$

Gradient Descent

- **RMSProp: root mean square propagation**
  - To solve the shortcoming of Adagrad
  - Obtained by the squared value of the gradient in Adagrads equation, G_t
  - The learning rate is adapted for each parameter.
  - Improves the latter by including the exponential moving average of the squared gradient

$$G = \gamma G + (1 - \gamma)\left(\nabla_\theta J(\theta_t)\right)^2$$

$$\theta = \theta - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta J(\theta_t)$$

- **Adaptive Moment Estimation (Adam)**
  - A combination of RMSprop with Momentum
  - Fast performance
  - Like the Momentum; stores the exponential average of the slope calculated so far
  - Like RMSProp: stores the exponential average of the square value of the gradient

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta \mathrm{J}(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left(\nabla_\theta J(\theta)\right)^2$$

  - $m_t$and v_t initially zero

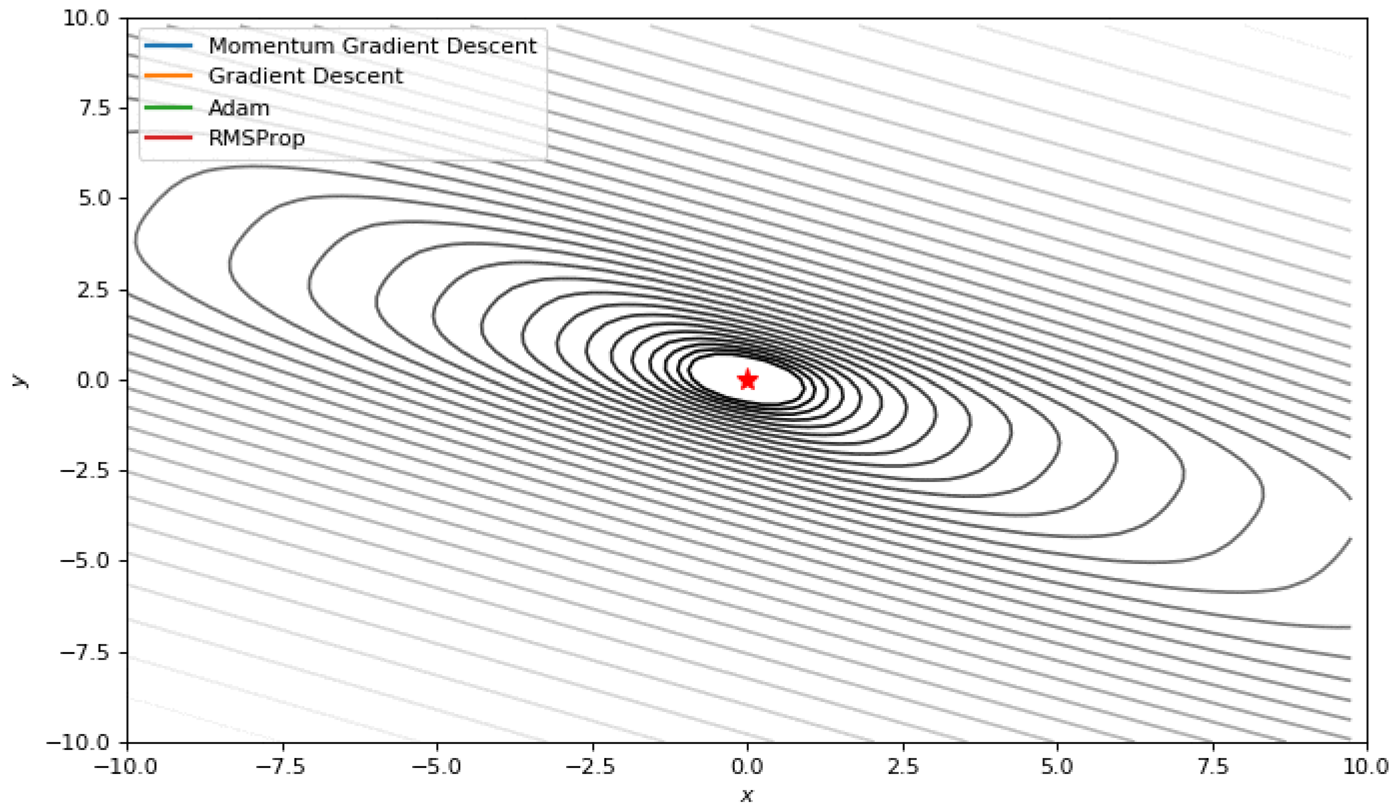
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

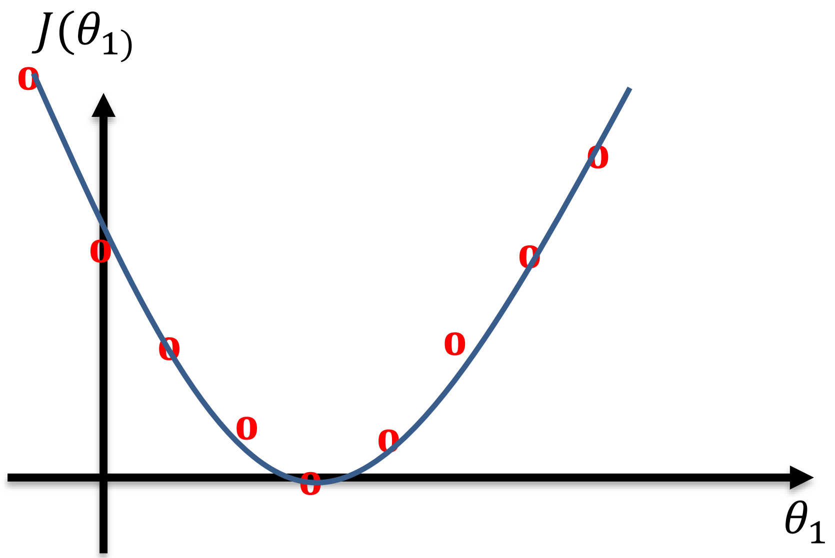
Gradient Descent

- **Comparison:**



Cost Function

- **Visualizing the cost function**
  - If there is only one parameter, then easy to draw
  - But not practical for high-dimensional spaces
  - e.g. in CIFAR-10 has 30,730 parameters

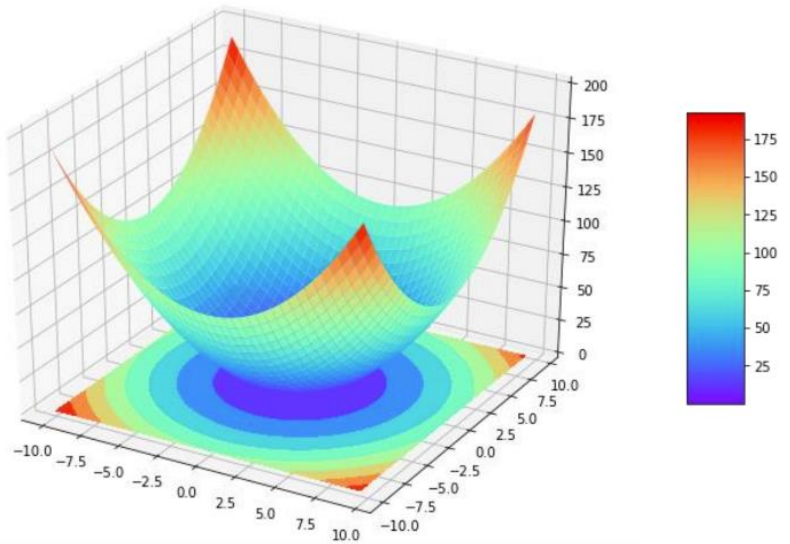


$\boldsymbol{\theta_1}$	$\boldsymbol{J(\theta_1)}$
0	2.3
1	0
0.5	0.58
…	…

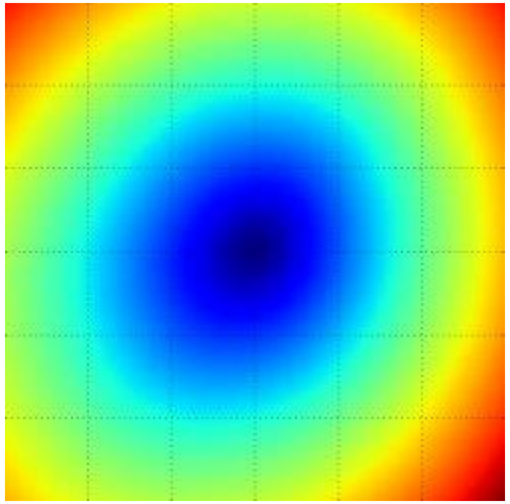
Cost Function

- **Visualizing the cost function**
  - For high-dimensional space:
    - Generate a random weight matrix θ
    - θ corresponds to a single point in the space
    - Then, generate a random directions θ_1
    - Compute the loss along this direction by evaluating $J(\theta + a\theta_1)$ for different values of a

1D loss by only varying $\boldsymbol{a}$



3D loss slice



2D countour

ANN cost function:

$$J(\Theta) = \frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\theta(x^{(i)}))_k + \left(1 - y_k^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)_k\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

Goal: to **minimize** the **cost function,** $\min_{\boldsymbol{\Theta}} \boldsymbol{J}(\boldsymbol{\Theta})$

- We need code to compute:
  - Cost function: $J(\Theta)$
  - Partial derivative terms: $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$

Gradient Descent

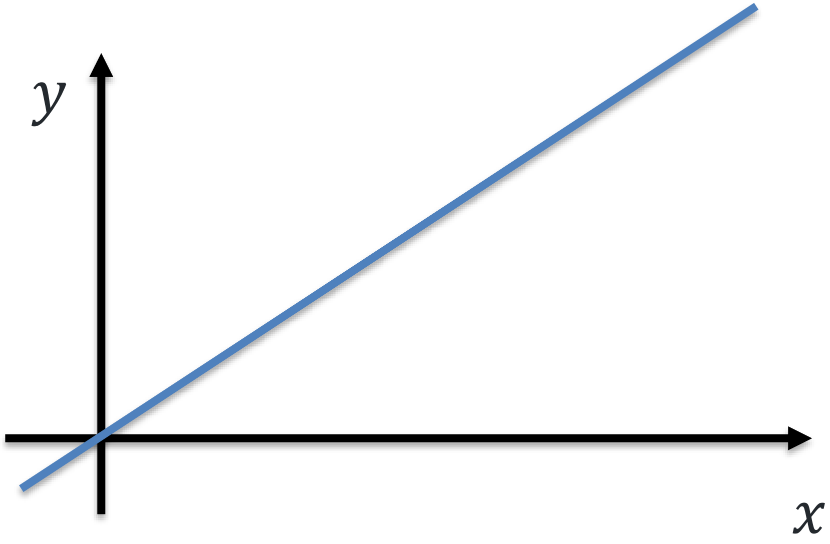
- For simplicity at this time just consider one parameter, e.g. θ_0=0:

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

Parameters: θ_1

Cost function: $$J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(\underbrace{h_\theta\left(x^{(i)}\right)}_{\theta_1 x} - y^{(i)}\right)^2$$

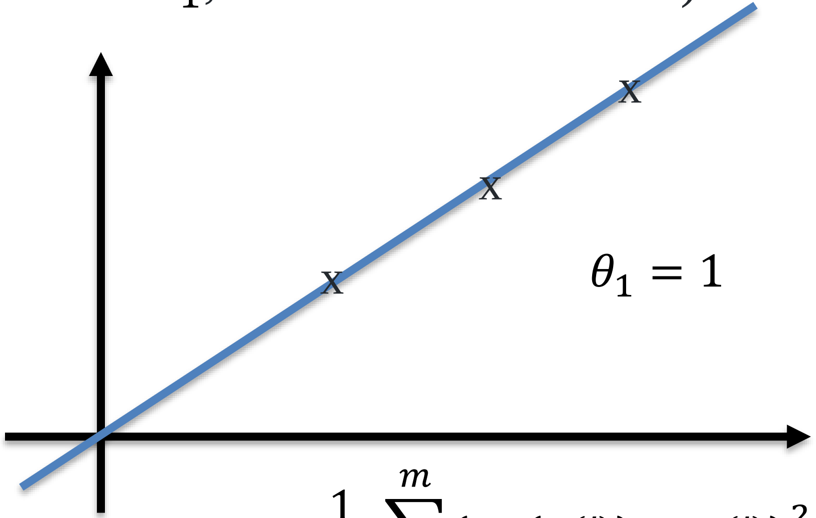
Goal: $\min_{\theta_0,\theta_1} J(\theta_1)$



Hypothesis

$h_\theta(x)$

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



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

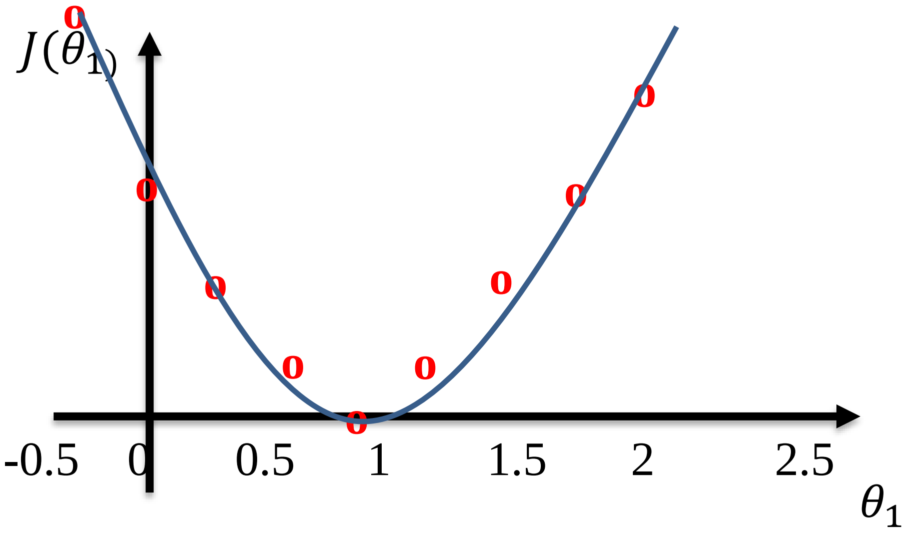
$$= \frac{1}{2m}\sum_{i=1}^{m}\left(\theta_1\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$= \frac{1}{2m}(0^2 + 0^2 + 0^2) = 0$$

Cost Function

$J(\theta_1)$

(Function of the parameter θ_1)

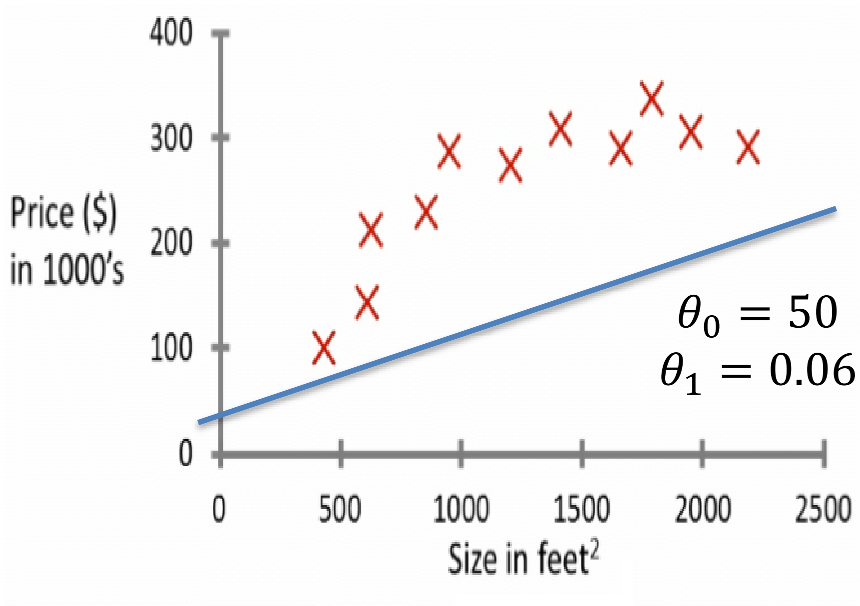


θ_1	$J(\theta_1)$
0	2.3
1	0
0.5	0.58
…	…

Hypothesis

$h_\theta(x)$

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

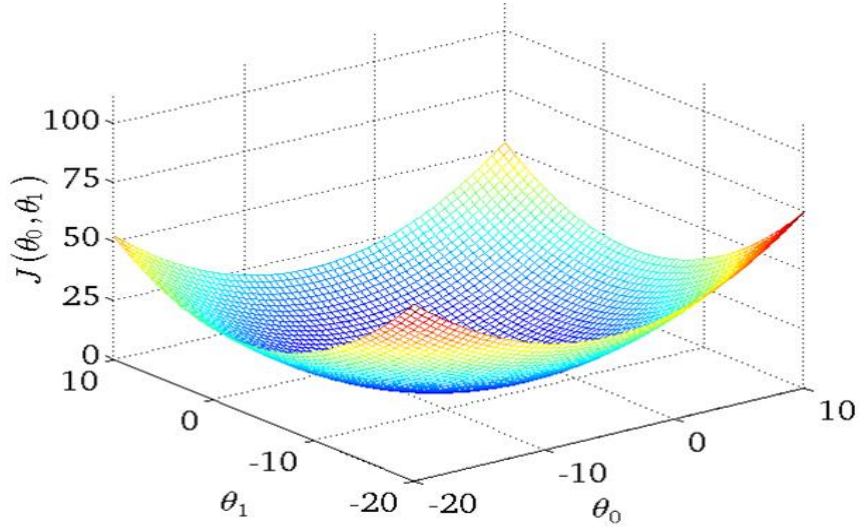


$$h_\theta(x) = 50 + 0.06x$$

Cost Function

$J(\theta_0, \theta_1)$

(Function of the parameters θ_0 and θ_1)

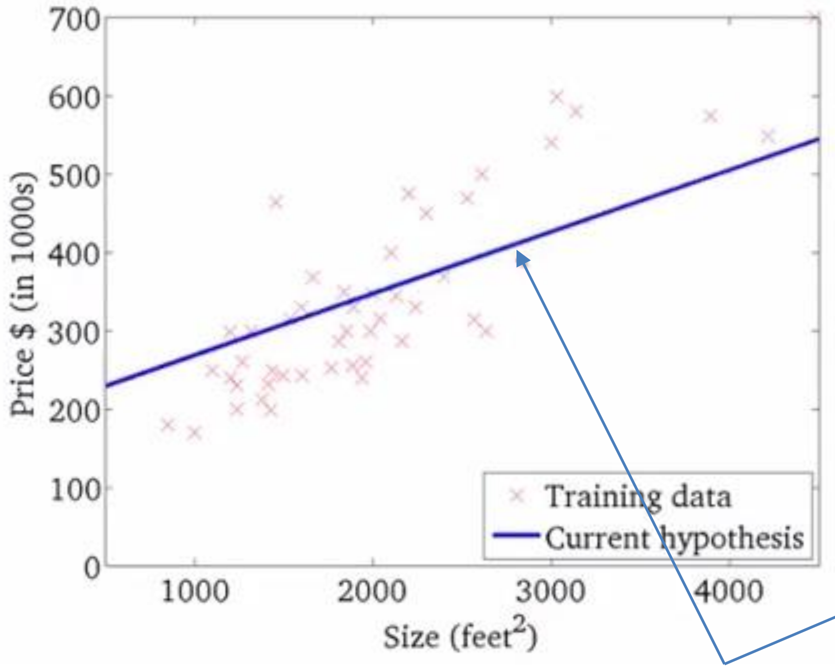


Cost Function

Hypothesis

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

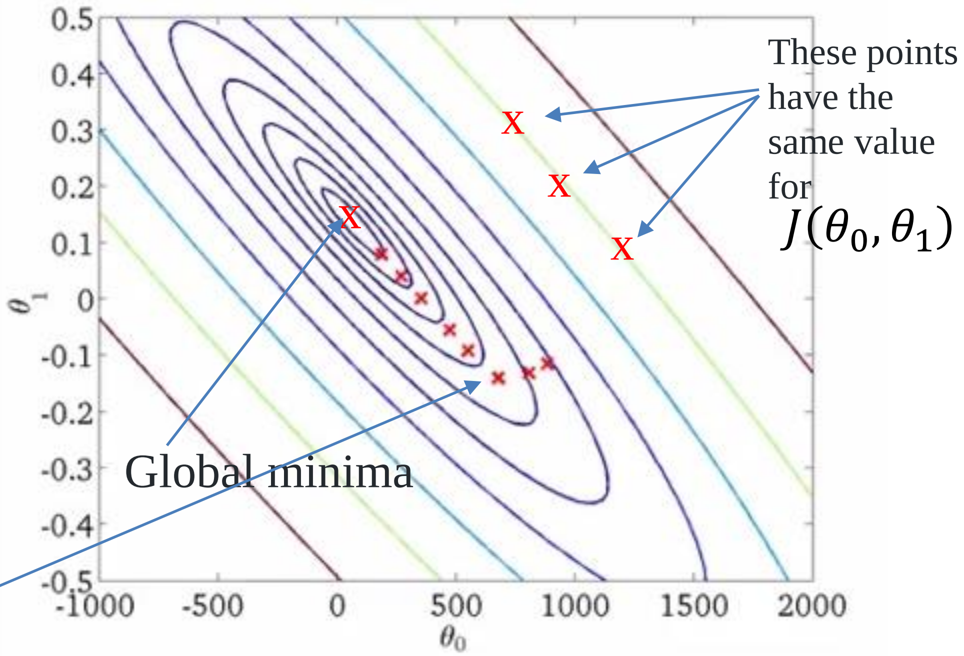
(for fixed θ_0 and θ_1, this is a function of x)



Cost Function

$$J(\theta_0, \theta_1)$$

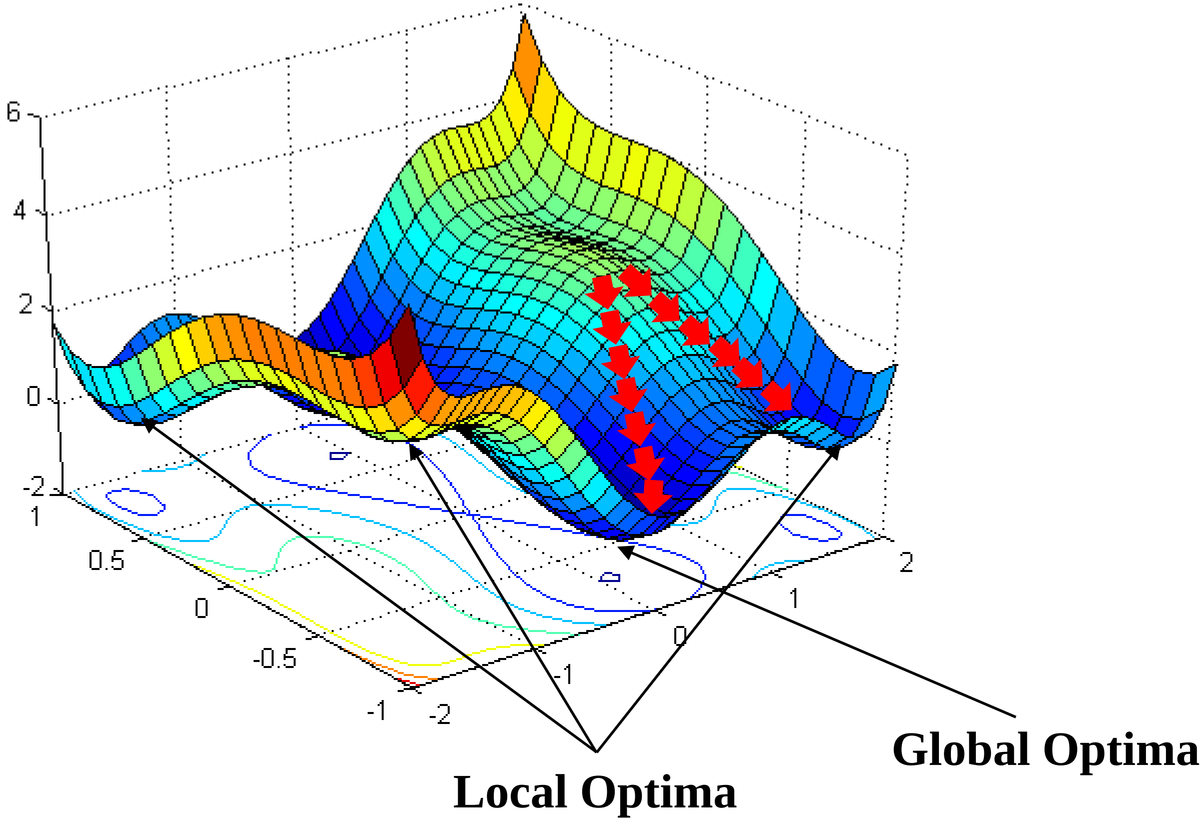
(Function of the parameters θ_0 and θ_1)



By adjusting the regression line we move toward the global minima point

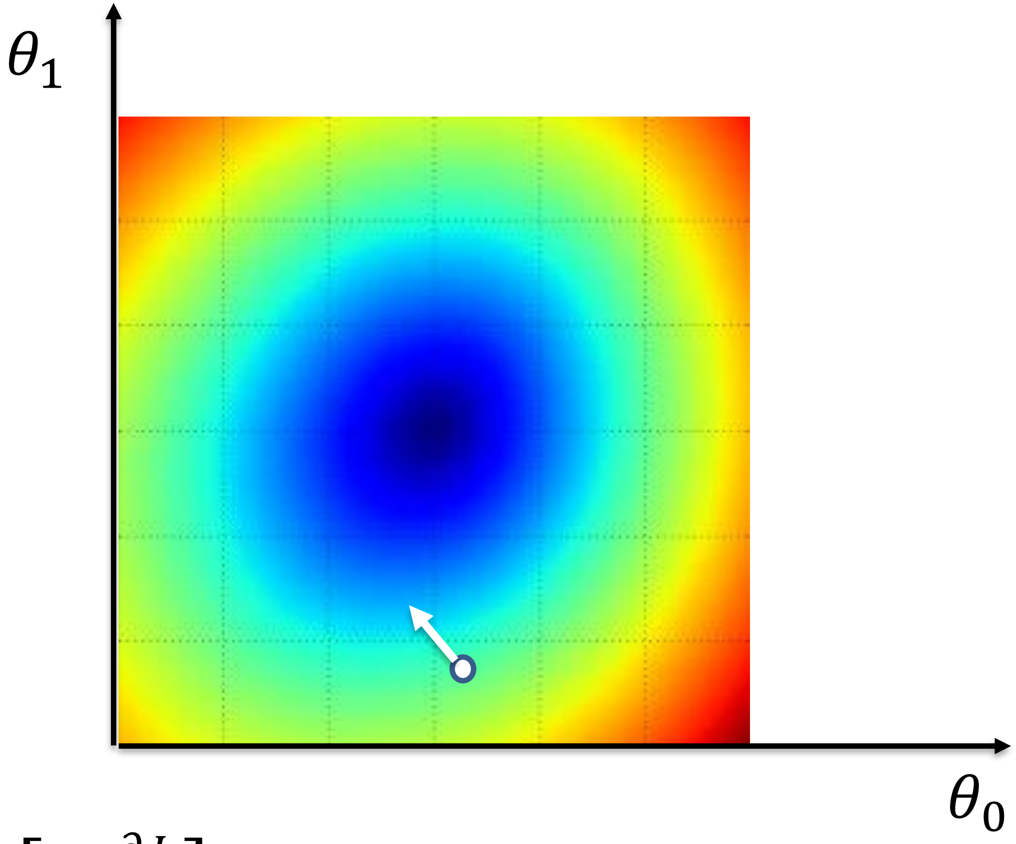
Gradient Descent

- We start by some random value for θ_0, θ_1
- We take some step towards minimum points.



Gradient Descent

- $J(\theta_0, \theta_1)$
- Define a vector $\boldsymbol{\Theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$
- Transpose of $\boldsymbol{\Theta}^{\mathbf{T}} = [\theta_0 \quad \theta_1]$
- $\nabla J(\theta_0, \theta_1) = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \end{bmatrix}$
- -$\nabla J(\theta_0, \theta_1) = \begin{bmatrix} -\frac{\partial J}{\partial \theta_0} \\ -\frac{\partial J}{\partial \theta_1} \end{bmatrix}$
- $\boldsymbol{\Theta}_{\mathbf{j}} = \boldsymbol{\Theta}_{\mathbf{j}} + \alpha(-\nabla \mathrm{J}(\theta_0, \theta_1)) = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} + \alpha \begin{bmatrix} -\frac{\partial J}{\partial \theta_0} \\ -\frac{\partial J}{\partial \theta_1} \end{bmatrix}$



Gradient Descent

The Algorithm

$$repeat\ until\ convergence\ \{$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad for\ j = 0, j = 1$$

$$\}$$

- Simultaneous update θ_0, θ_1

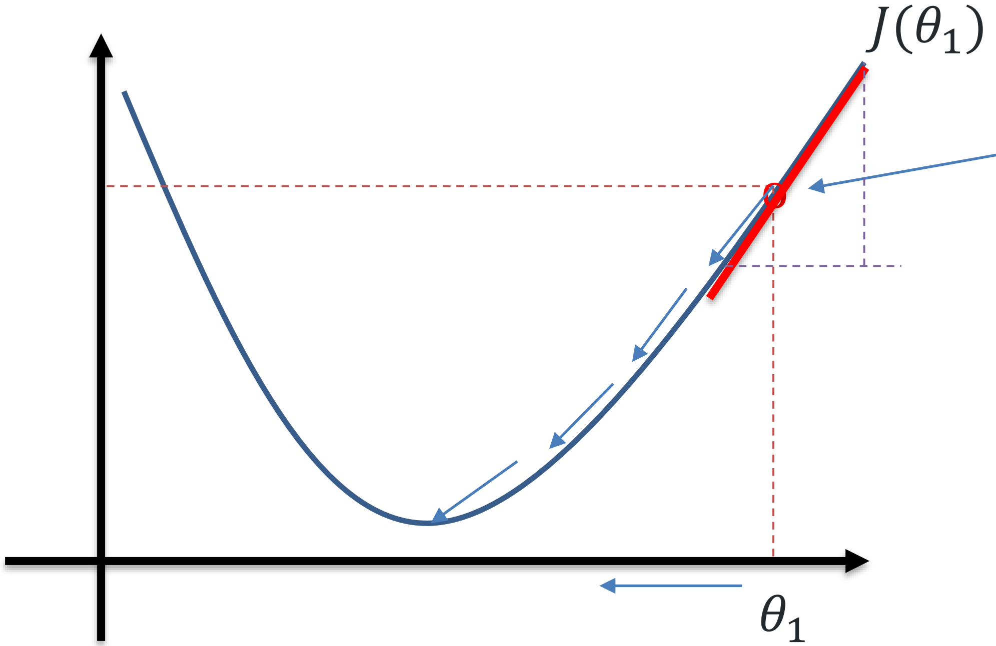
$$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \qquad temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

$$\theta_0 := temp0 \qquad \theta_1 := temp1$$

Gradient Descent

- Suppose we have one parameter

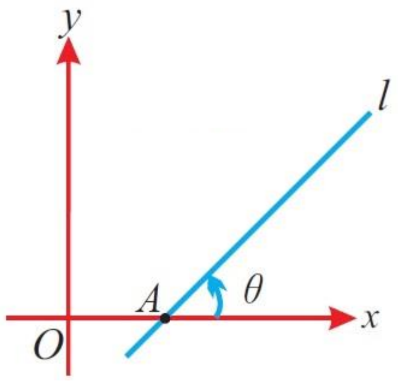
$$\min_{\theta_1} J(\theta_1) \qquad \theta_1 \in \mathbb{R}$$



$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

$$\frac{\partial}{\partial \theta_1} J(\theta_1) \geq 0$$

$$\theta_1 := \theta_1 - \alpha(positive\ number)$$



$slope$:
$m = \tan(\theta)$

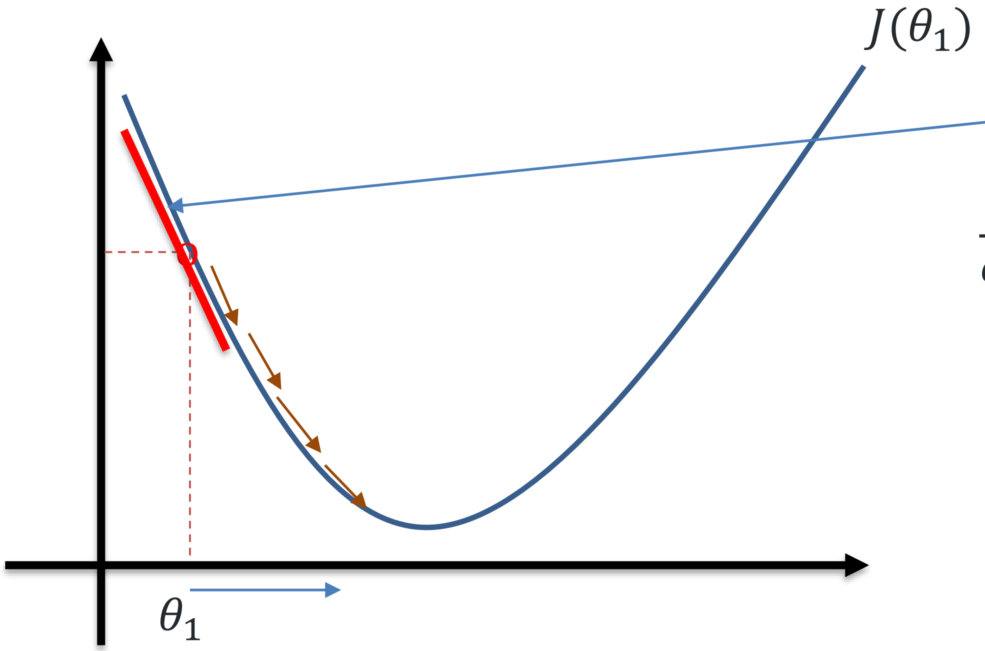
Gradient Descent

- Lets try another example:

- Suppose we have one parameter

$$\min_{\theta_1} J(\theta_1) \qquad\qquad \theta_1 \in \mathbb{R}$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$



$$\frac{\partial}{\partial \theta_1} J(\theta_1) \leq 0$$

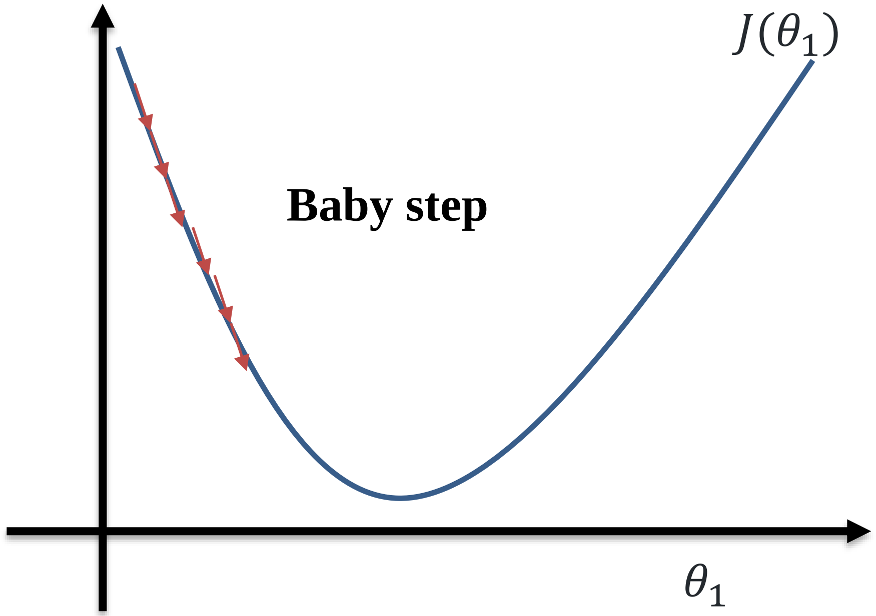
$$\theta_1 := \theta_1 - \alpha(negative\ number)$$

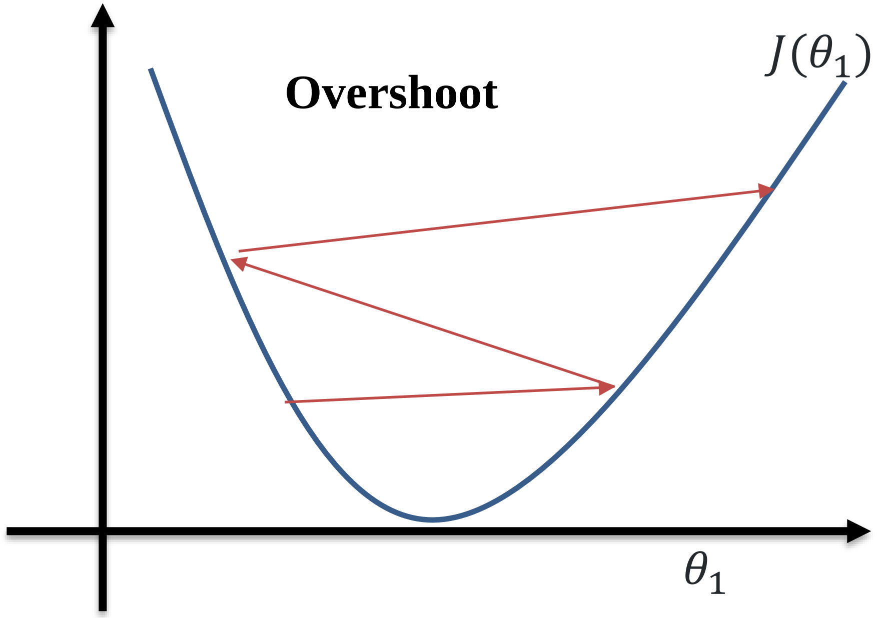
Gradient Descent

- **Learning rate**:

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

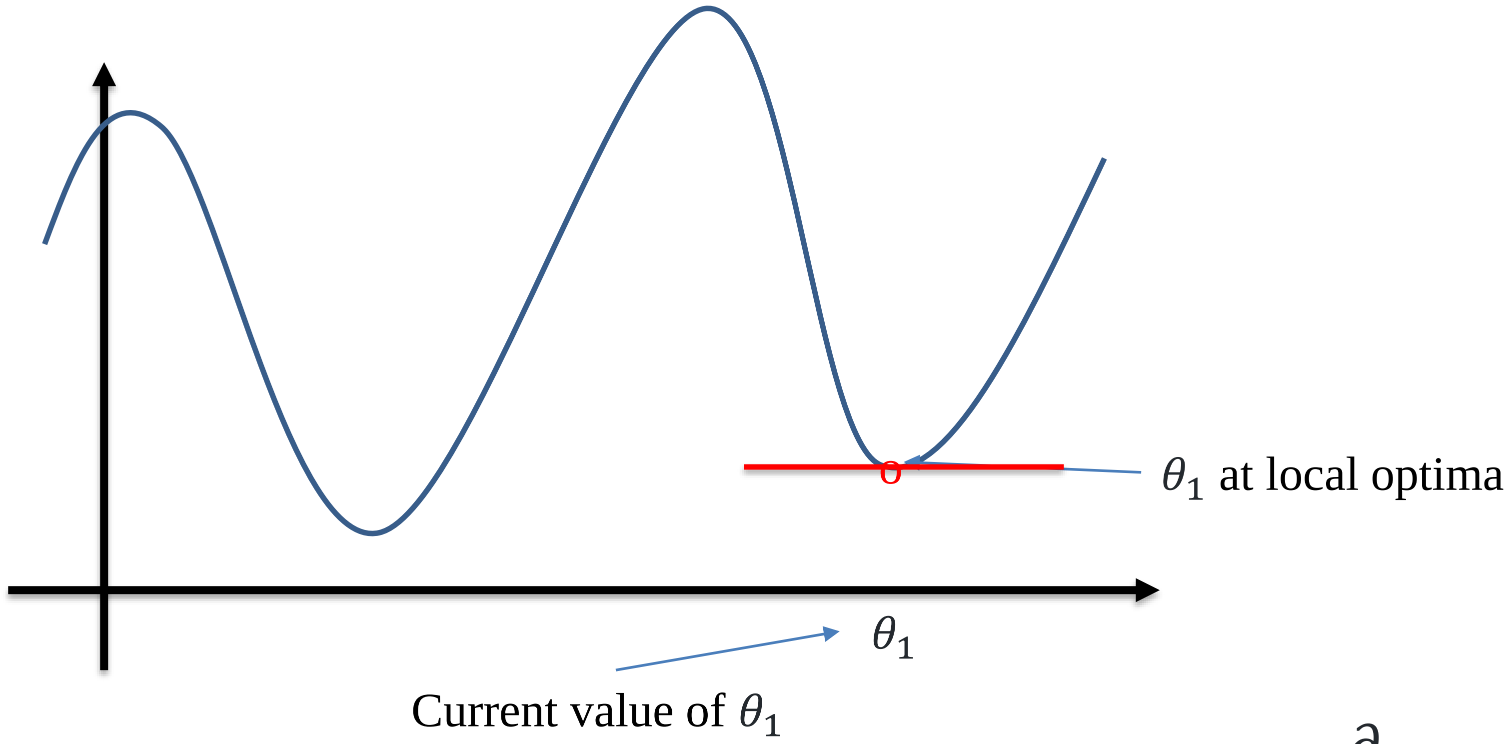
- **Small** α: gradient descent can be **slow**.
- **Large** α:, **overshoot** the minimum.
    - e.g. may fail to converge, or even diverge.





Gradient Descent

- What if the starting point is at **local optima**?



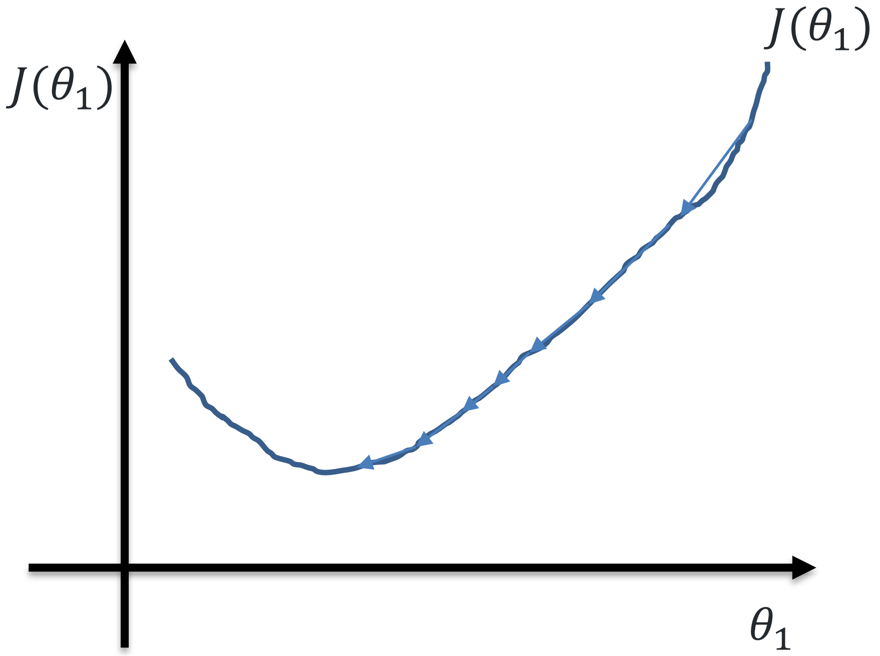
$$\theta_1 := \theta_1 - \alpha \cancelto{0}{\frac{\partial}{\partial \theta_1}} J(\theta_1)$$

$$\theta_1 := \theta_1 - \alpha(0)$$
$$\theta_1 := \theta_1$$

Gradient Descent

- Gradient descent can converge to a local minimum, even with the learning α **fixed**.

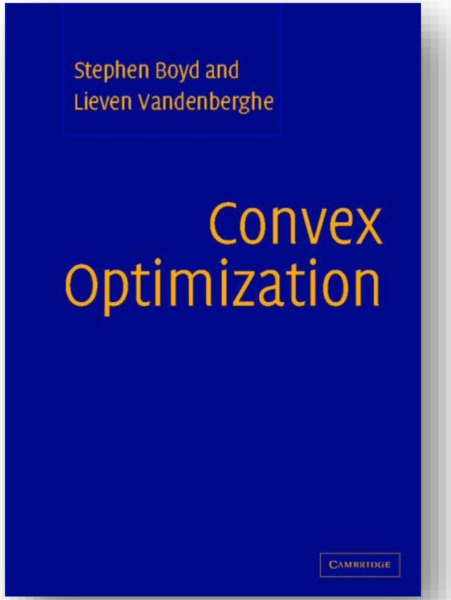
$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$



- As we approached a **local minimum**, gradient descent will automatically take **smaller** steps.
- So, no need to decrease α over time.

Sources

- Algorithms for Optimization, 2019.
- Essentials of Metaheuristics, 2011.
- Computational Intelligence: An Introduction, 2007.
- Introduction to Stochastic Search and Optimization, 2003.



Convex Optimization
Stephen Boyd and Lieven Vandenberghe
Cambridge University Press

Office: #432, Dayang AI Center, **Phone**: 010-8999-8586, **email**: piran@sejong.ac.kr