Numeric Optimization
Stochastic Gradient Descent

Nagiza F. Samatova, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

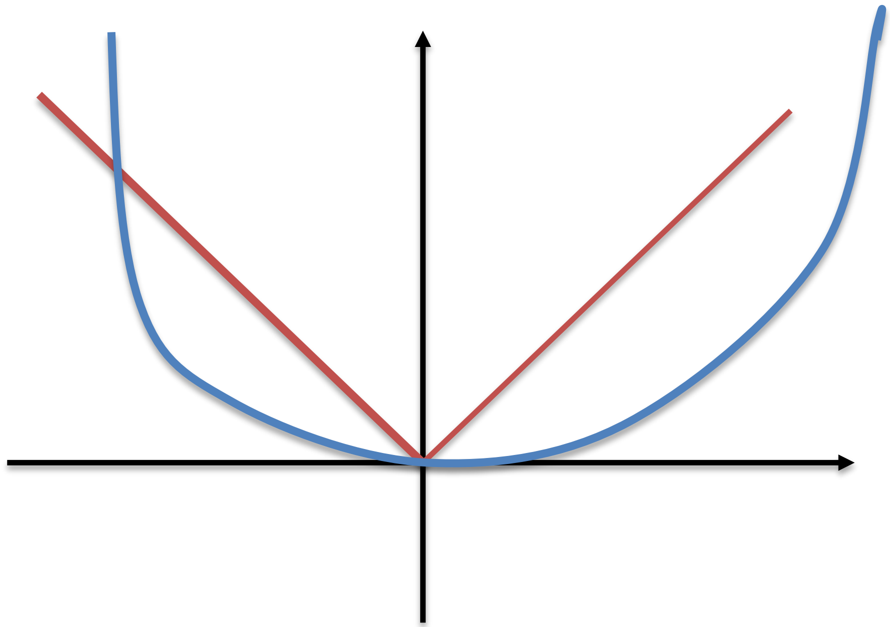
Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

- $\boldsymbol{f(z) = b + m \cdot z}$
- $\boldsymbol{z = g(x) = \; x^2}$
- $\boldsymbol{f\left(g\,(x)\right) = b + m \cdot x^2}$

- $\frac{df}{dx} =$

Function of residuals

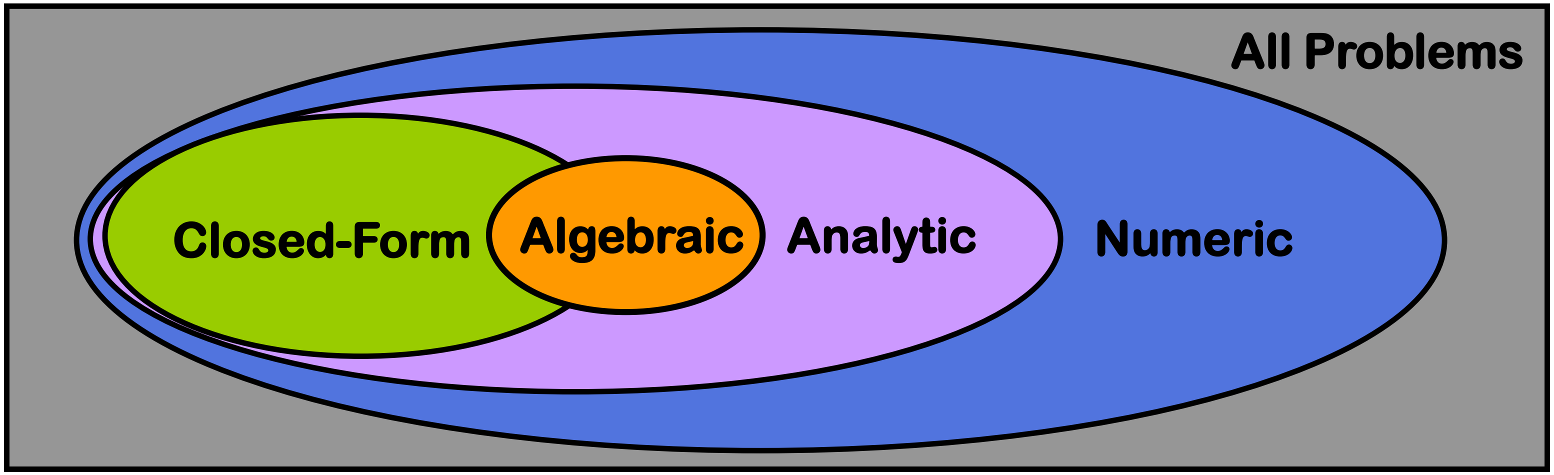
- $\sum_{i=1}^{n}\left(target^{(i)} - predicted^{(i)}\right)^2 \rightarrow minimize$



Optimization Problems

CLOSED-FORM, ALGEBRAIC, ANALYTIC, NUMERIC SOLUTIONS

Classes of Problems



Closed-Form Expression

- **A closed-form mathematical expression:**
  - Evaluated in a finite number of operations.
  - Expressed:
    - in terms of constants, variables, "well-known" operations (e.g., + − × ÷), and functions (e.g., n^{th} root, logs, exp, trigonometric functions, and inverse hyperbolic functions
    - but **NOT** in terms of **limits, integrals, infinite series**

- **Tractable Problems**:
  - Can be solved in terms of a closed-form expression
  - Example: $ax^2 + bx + c = 0$ is a tractable problem; its solution is in a closed-form

- **CDF:** Many cumulative distribution functions (CDF) can **NOT** be expressed in closed-form:
  - Ways around this issue: To consider special functions such as the error function or gamma function.

src: Wiki

Analytic Mathematical Expression

- **An analytic expression:**
  - Constructed using well-known operations that lend themselves readily to calculation
  - Expressed:
    - in terms of constants, variables, "well-known" operations (e.g., + − × ÷), and functions (e.g., n^{th} root, logs, exp, trigonometric functions, and inverse hyperbolic functions,
    - may include **infinite series**, **Gamma and Bessel functions**,
    - but **NOT** **limits, integrals**.

src: Wiki

Algebraic Expression

- **An algebraic expression is an analytic expression:**
  - Expressed only in terms of the algebraic operations (addition, subtraction, multiplication, division and exponentiation to a rational exponent) and rational constants

src: Wiki

Numeric Algorithms

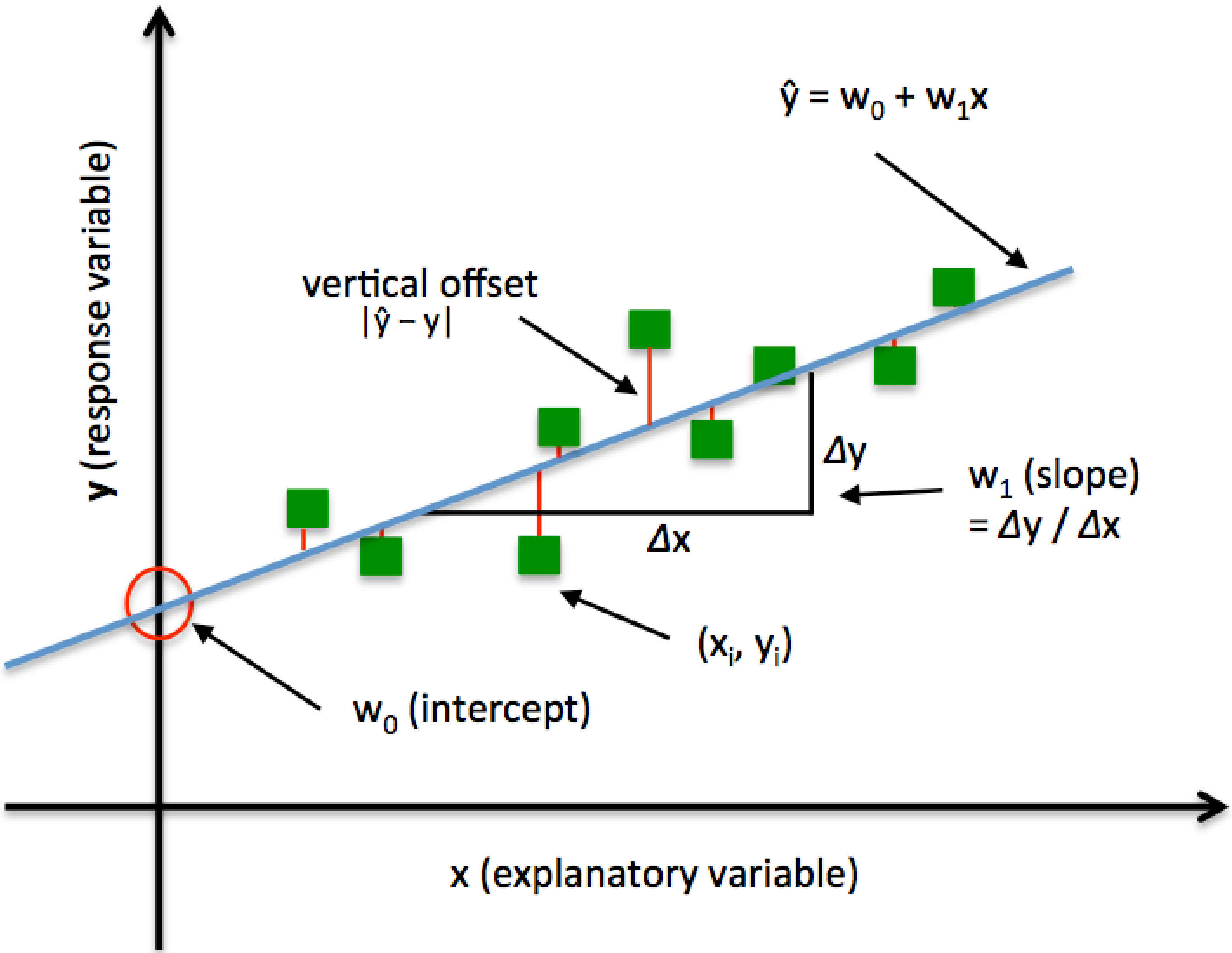
- **Numeric algorithms use numeric approximations:**
  - ***Discretization*** for numeric integration
  - ***Numerical differentiation***
  - ***Iterative*** methods (e.g., Newton's method) for optimization
  - **Numerical interpolation, extrapolation, smoothing**

src: Wiki

Linear Regression: Analytic Solution

MODEL FITTING VIA CLOSED-FORM EQUATIONS

Ordinary Least Squares (OLS) Linear Regression



OLS Objective (Minimization) Function

Goal: To find the line (or hyperplane) that minimizes the vertical offsets

- **In other words, to find the "best-fitting line"—the line that minimizes:**
  - **The sum of squared errors (SSE) or**
  - **The mean squared error (MSE)**

between the target variable (y) and predicted output over all samples in the data set of size n

$$\boxed{SSE = \sum_{i=1}^{n} \left(target^{(i)} - output^{(i)}\right)^2 \rightarrow minimize}$$

$$\boxed{MSE = \frac{1}{n} \times SSE \rightarrow minimize}$$

Closed-Form Algebraic Solution

Linear regression model

for the response variable y and an d-dimensional predictor vector $x \in \mathbb{R}^d$:

$$\boxed{y = w_0x_0 + w_1x_1 + \cdots + w_dx_d}$$

- where w_0 is the y-axis intercept
- and therefore, $x_0 = 1$

Vector form:

$$\boxed{y = \vec{w}^T\vec{x}} \longleftarrow$$ scalar product of the weight and predictor vectors

- **<u>Known</u>: The i.i.d. sample of size n:**
  - **Response vector : $\vec{y} = (y_1, y_2, \cdots, y_n)$**
  - **Matrix : $X_{n\times(d+1)}$, each row is an m-dimensional predictor vector**
- **<u>UN-known</u>: the weight vector $\vec{w}$ for the "best-fitting" model**

$$\boxed{\vec{y} = X \times \vec{w}}$$

Can you solve this system of linear algebraic equations?

Closed-Form Algebraic Solution

- **<u>Known</u>: The i.i.d. sample of size n:**
  - **Response vector :** $\vec{y} = (y_1, y_2, \cdots, y_n)$
  - **Matrix :** $X_{n \times d}$**, each row is an** m**-dimensional predictor vector**

- **<u>UNknown</u>: the weight vector** $\vec{w}$ **for the "best-fitting" model**

$$\boxed{\vec{y} = X \times \vec{w}}$$

- **Algebraic Solution via Matrix Inverse**

$$\boxed{X^T \times \vec{y} = X^T \times X \times \vec{w}}$$

1. Multiply both sides by X^T

$$\boxed{\left(X^T \times X\right)^{-1} \times X^T \times \vec{y} = \vec{w}}$$

2. Multiply both sides by $(X^T \times X)^{-1}$

3. Using the closed-form (normal equation) solution, compute the weights of the model as follows:

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

When to Use Algebraic (Closed-Form) Solution?

Using the closed-form (normal equation) solution, compute the weights of the model as follows:

$$\boxed{\vec{w} = \left(X^T X\right)^{-1} X^T \, \vec{y}}$$

- **The closed-form solution may (should) be preferred for "smaller" datasets – if computing (a "costly") matrix inverse is not a concern.**

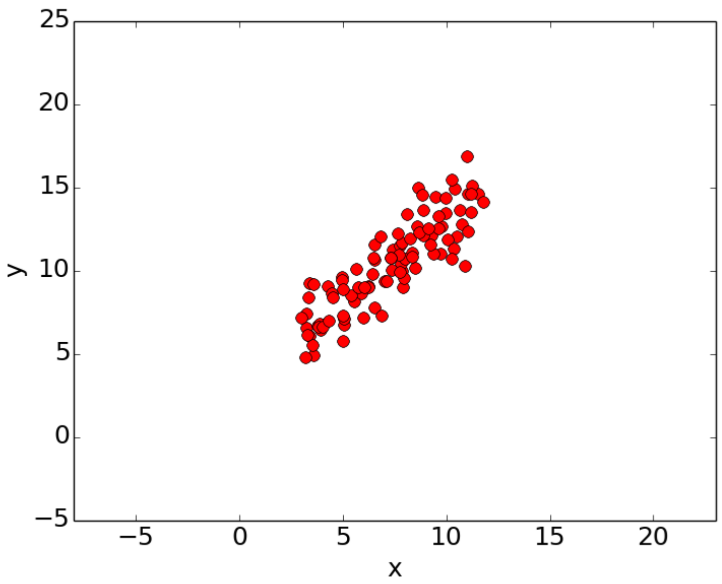
- **For very large datasets or datasets where the inverse of $X^T X$ may not exist (the matrix is non-invertible or singular, e.g., in case of perfect multicollinearity), different approaches such as Gradient Descent or Stochastic Gradient Descent are to be preferred.**

Numeric Approximation

GD: GRADIENT DESCENT

Simple Example: Linear Regression

Input:



- Goal: To model this set of points with a line.

- Use the standard $\boldsymbol{y} = \boldsymbol{m} \cdot \boldsymbol{x} + \boldsymbol{b}$ line equation
  - where $w_1 = m$ is the line's slope and
  - $w_0 = b$ is the line's y-intercept

- To find the best line for this data:
  - to find the **best pair of slope** $\boldsymbol{m}$ and $\boldsymbol{y}$**-intercept** $\boldsymbol{b}$ values

Standard Approach

- To solve this type of problem:
  - Step 1: Define an **error function** (also called a **cost function**) that measures how "good" a given line is:
    - Input: an $(\mathrm{w}_0, \mathrm{w}_1) = (m, b)$ pair for a given line
    - Output: an error value based on how well the line fits our data
  - Step 2: **Compute this error** for the given line by
    - ***iterating*** through each (x, y) point in the input data set and
    - ***computing MSE***: summing the square distances between each point's y value and the candidate line's y value (computed at $m \cdot x + b$)
    - Why to square this distance? To ensure that it is positive and to make the error function differentiable (e.g., unlike absolute value function, $abs()$).

$$\boxed{\boldsymbol{Error(m, b) = \frac{1}{n}\sum_{j=1}^{n}\left(y_j - \left(m \cdot x_j + b\right)\right)^2}}$$

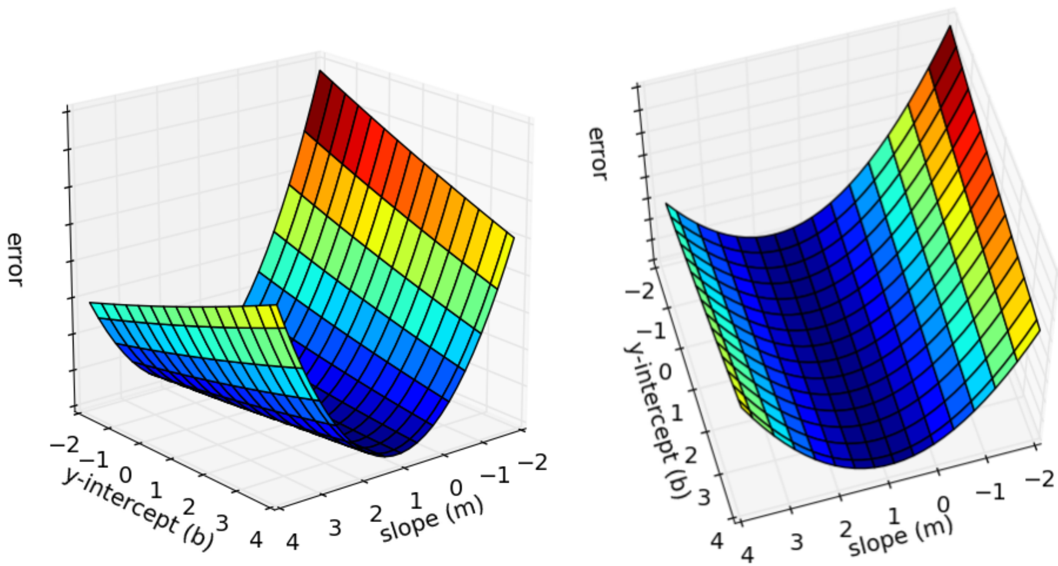
Error for a Given Line: Python Code

```python
# y = mx + b
# m is slope, b is y-intercept
def computeErrorForLineGivenPoints(b, m, points):
    totalError = 0
    for i in range(0, len(points)):
        totalError += (points[i].y - (m * points[i].x + b)) ** 2
    return totalError / float(len(points))
```

Minimization of the Error / Cost Function

Lines that fit the data better (where "better" is defined by the error function) will result in lower error values.

$$Error(m, b) = \frac{1}{n}\sum_{j=1}^{n}\left(y_j - (m \cdot x_j + b)\right)^2 \rightarrow minimize$$

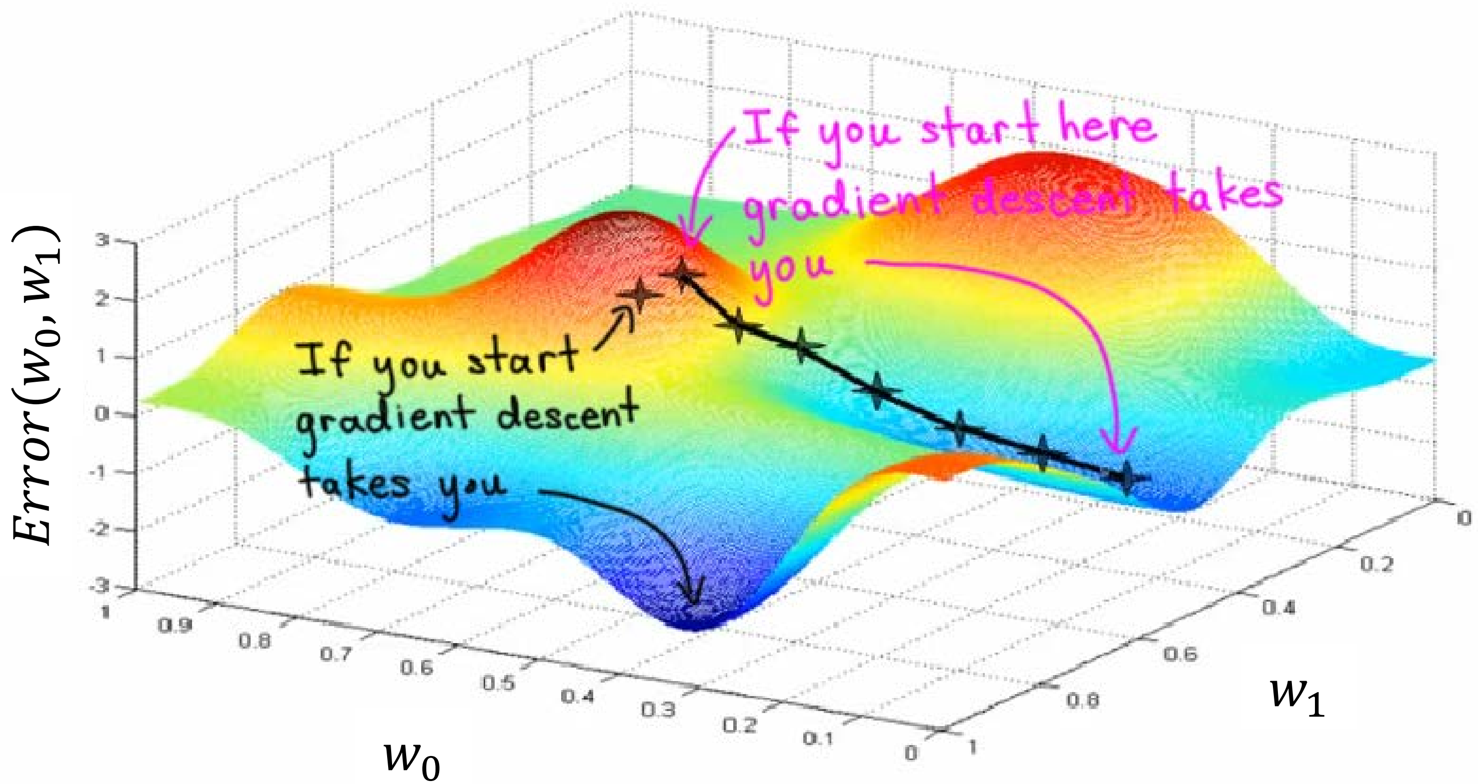


- *Each point* in this two dimensional space represents *a line*
- The *height* of the function at each point is the *error value* for that line
- Some lines yield *smaller error values* than others (*fit data better*)

Figure: Visualization of the error function consisting of two parameters (m and b) as a two-dimensional surface.

Gradient Descent Search for the Minimum

- **<u>Main Idea</u>:**
  - **Start from some location on this surface and**
  - **Move downhill to find the line with the lowest error.**



The Gradient of the Error Function

- **To minimize the error function via gradient descent search:**
  - **Compute the gradient of the error function:**
    - The gradient will act like a compass and always point us downhill
    - **To compute the gradient ≡ to compute *partial derivative* for each parameter**, (m and b)

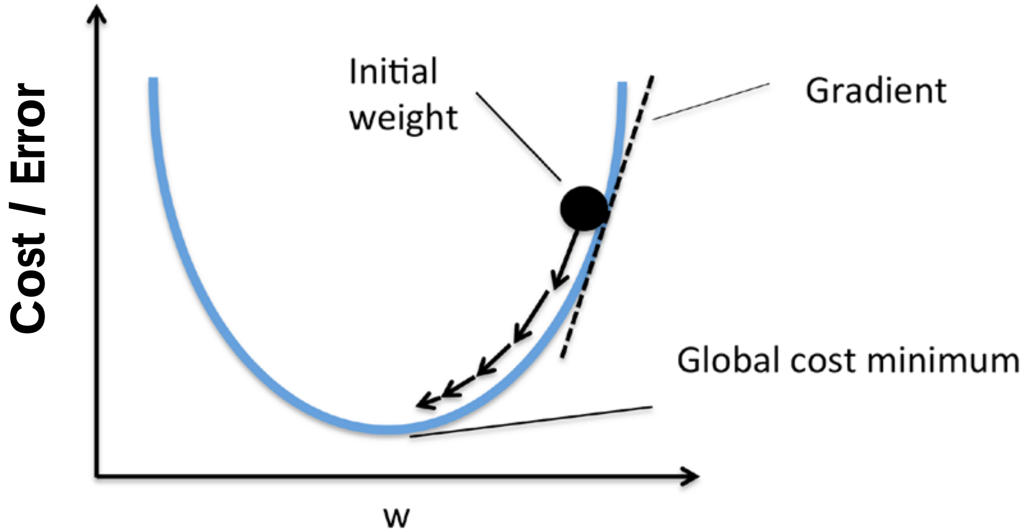
$$\boxed{\boldsymbol{Error(m,b) = \frac{1}{n}\sum_{j=1}^{n}\left(y_j - (m \cdot x_j + b)\right)^2}}$$

$$\frac{\partial Error(w_0,w_1)}{\partial w_0} = \frac{\boldsymbol{\partial Error(m,b)}}{\boldsymbol{\partial b}} = \frac{2}{n}\sum_{j=1}^{n} -\left(y_j - (m \cdot x_j + b)\right)$$

$$\frac{\partial Error(w_0,w_1)}{\partial w_1} = \frac{\boldsymbol{\partial Error(m,b)}}{\boldsymbol{\partial m}} = \frac{2}{n}\sum_{j=1}^{n} (-x_j) \cdot \left(y_j - (m \cdot x_j + b)\right)$$

Gradient Descent Search: The Mechanics

- **To minimize the error function via gradient descent search:**
  - <u>Step 1</u>: Initialize the search to start at any pair of m and b values (any line)
  - <u>Step 2</u>: Let the gradient descent algorithm march downhill on the error function towards the best line
    - For each iteration, update m and b to a line that yields slightly lower error than the previous iteration:
    - The direction to move in for each iteration is calculated using the two partial derivatives from above



GD Search: Python Code

```
def stepGradient(b_current, m_current, points, learningRate):
    b_gradient = 0
    m_gradient = 0
    N = float(len(points))
    for i in range(0, len(points)):
        b_gradient += -(2/N) * (points[i].y - ((m_current*points[i].x) + b_current
        m_gradient += -(2/N) * points[i].x * (points[i].y - ((m_current * points[i
    new_b = b_current - (learningRate * b_gradient)
    new_m = m_current - (learningRate * m_gradient)
    return [new_b, new_m]
```

gradient

update

b_gradient and m_gradient:

$$\frac{\partial Error(m,b)}{\partial b} = \frac{2}{n}\sum_{j=1}^{n} -\left(y_j - \left(m \cdot x_j + b\right)\right)$$

$$\frac{\partial Error(m,b)}{\partial m} = \frac{2}{n}\sum_{j=1}^{n} -x_j \cdot \left(y_j - \left(m \cdot x_j + b\right)\right)$$

new_b and new_m:

$$b_{new} = b_{current} - \gamma \cdot b_{gradient}$$

$$m_{new} = m_{current} - \gamma \cdot m_{gradient}$$

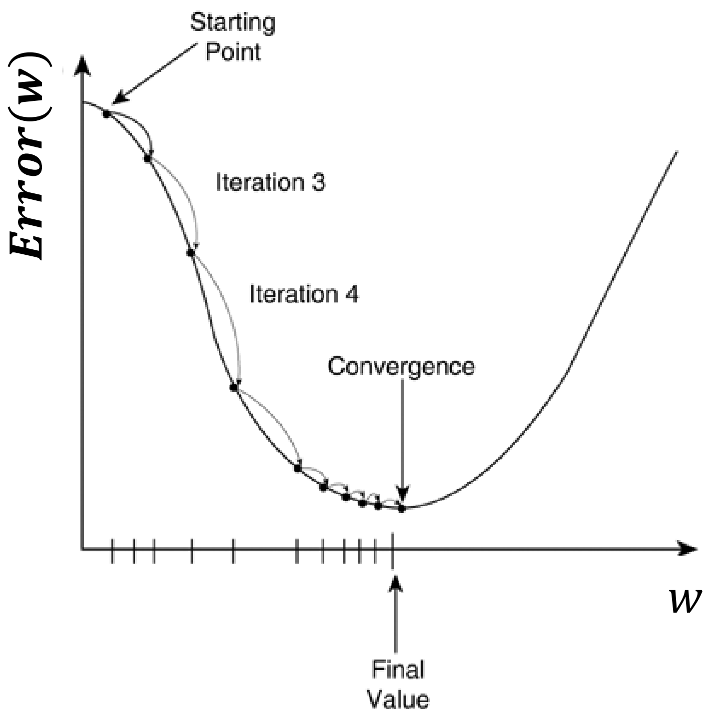
learning rate

The Learning Rate, γ

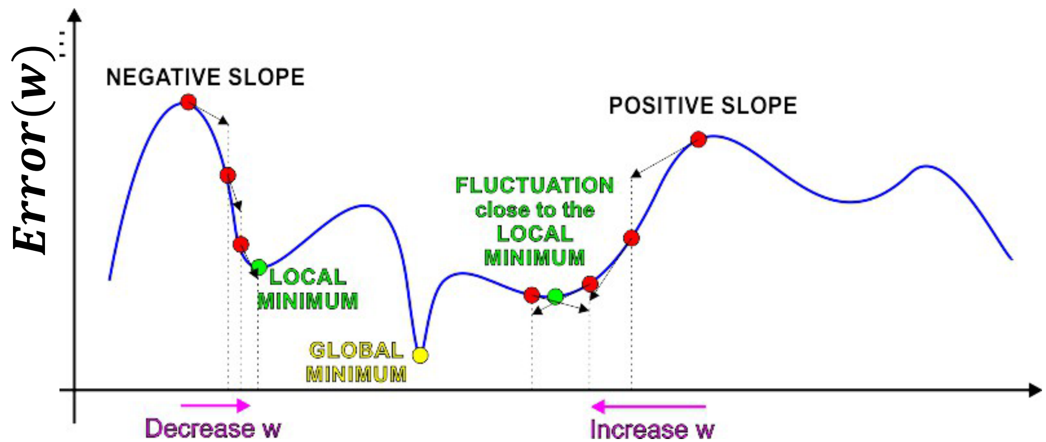
$$\boldsymbol{w_{new}} = \boldsymbol{w_{current}} - \boldsymbol{\gamma} \cdot \boldsymbol{w_{gradient}}$$

learning rate

- Learning rate (or step size) may change as a function of iteration

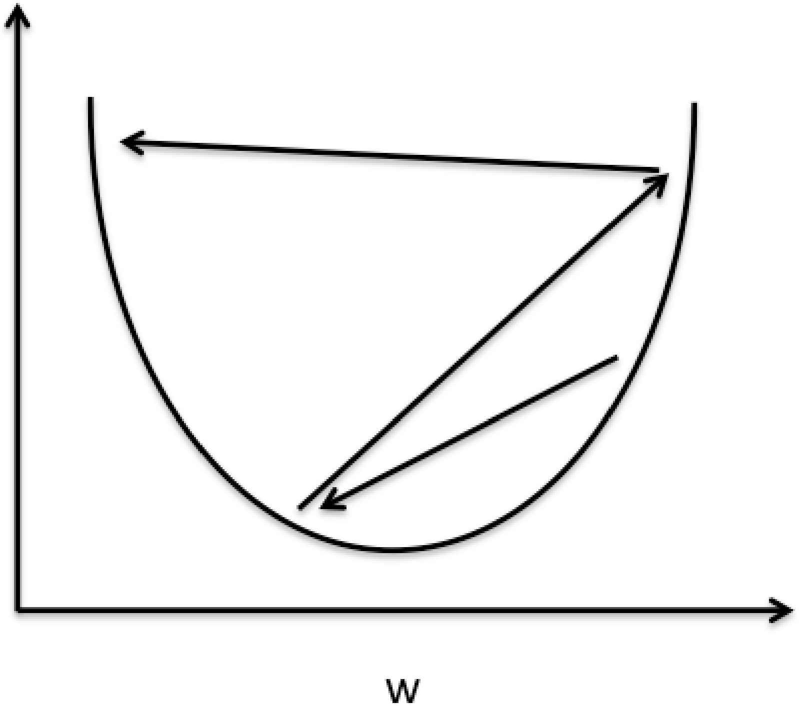


- The ***learning rate*** variable (γ) controls **how large of a step to take downhill** during each iteration:
  - If a step is too large, we may step over the minimum
  - However, if we take small steps, it will require many iterations to arrive at the minimum
  - Example: $\gamma = 0.0005$



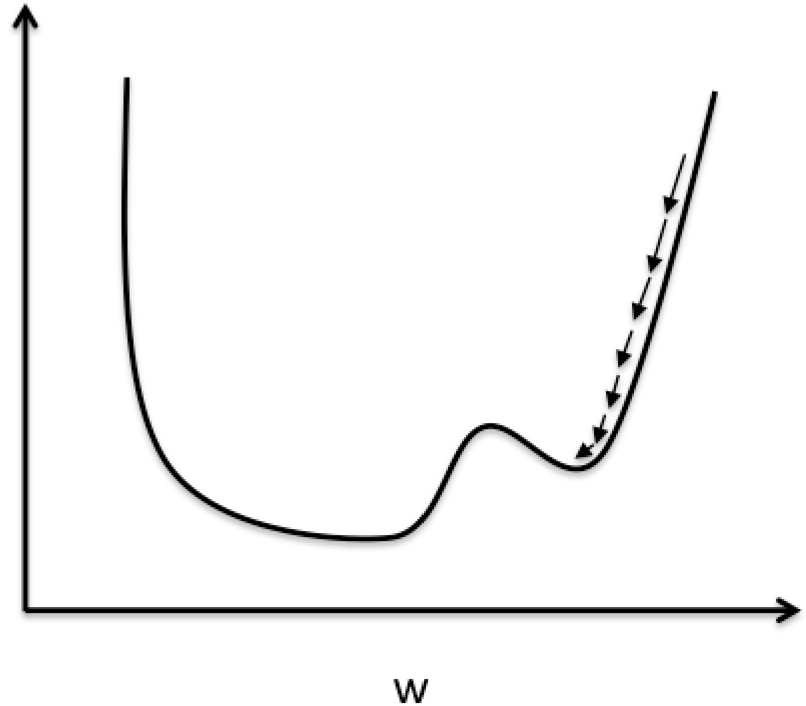
Small vs. Large Learning Rate

- zig-zag for large learning rate



Large learning rate: Overshooting.

- long convergence



Small learning rate: Many iterations until convergence and trapping in local minima.

The Weight Update

- The magnitude and direction of each weight update is computed by taking a **step in the opposite direction of the cost / error gradient**:

$$\Delta w_j = -\gamma \cdot \frac{\partial Error(w_0, w_1, \cdots, w_j, \cdots, w_d)}{\partial w_j}$$

$$\Delta w_j = \gamma \cdot \sum_{i=1}^{n} \left(target^{(i)} - output^{(i)}\right) \cdot x_j^{(i)}$$

note the summation over the entire training set

$$w_j := w_j + \Delta w_j$$

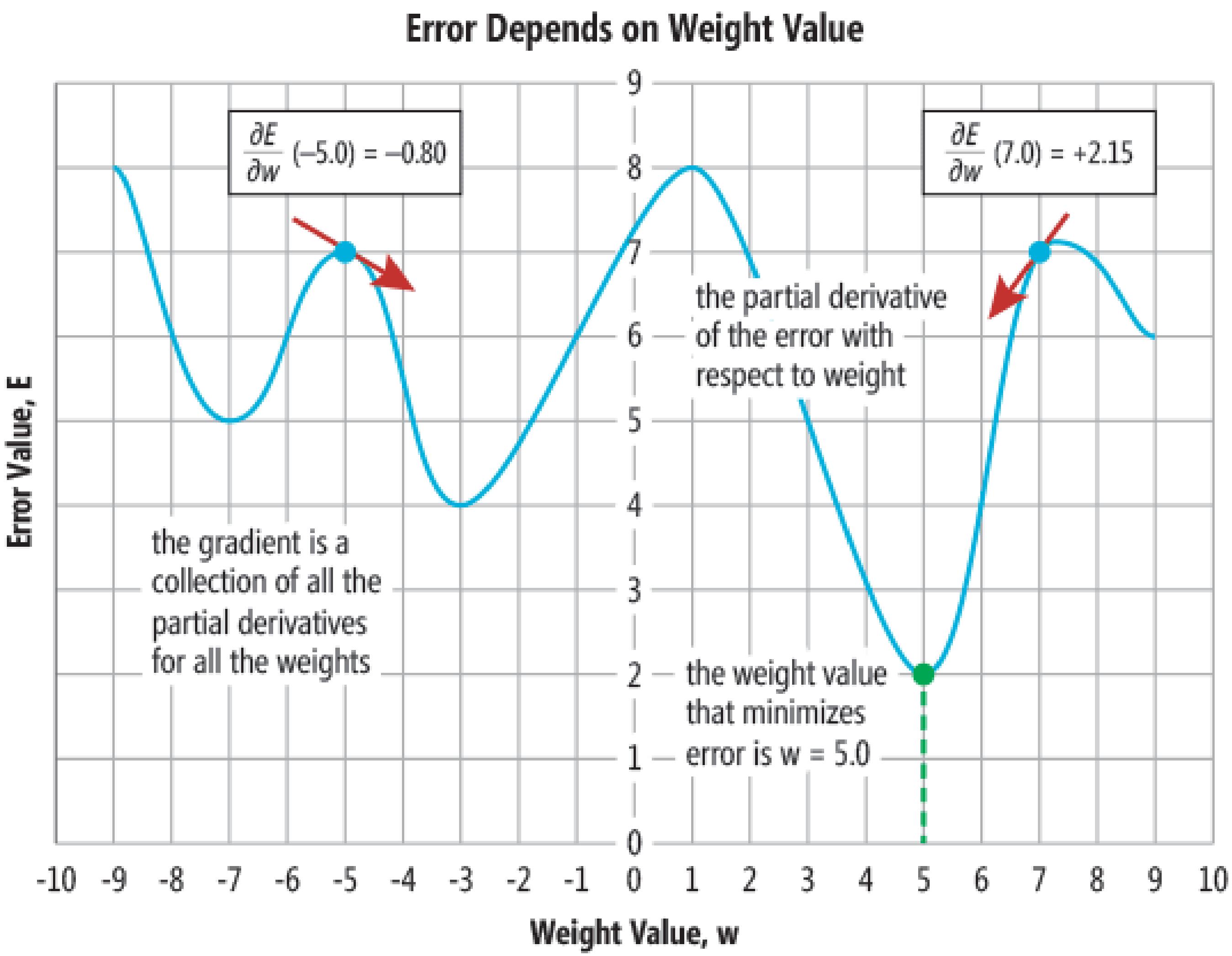
update

$$\vec{w} := \vec{w} + \overrightarrow{\Delta w}$$

$\overrightarrow{\Delta w}$ is a vector that contains the weight updates of each weight coefficient w_j

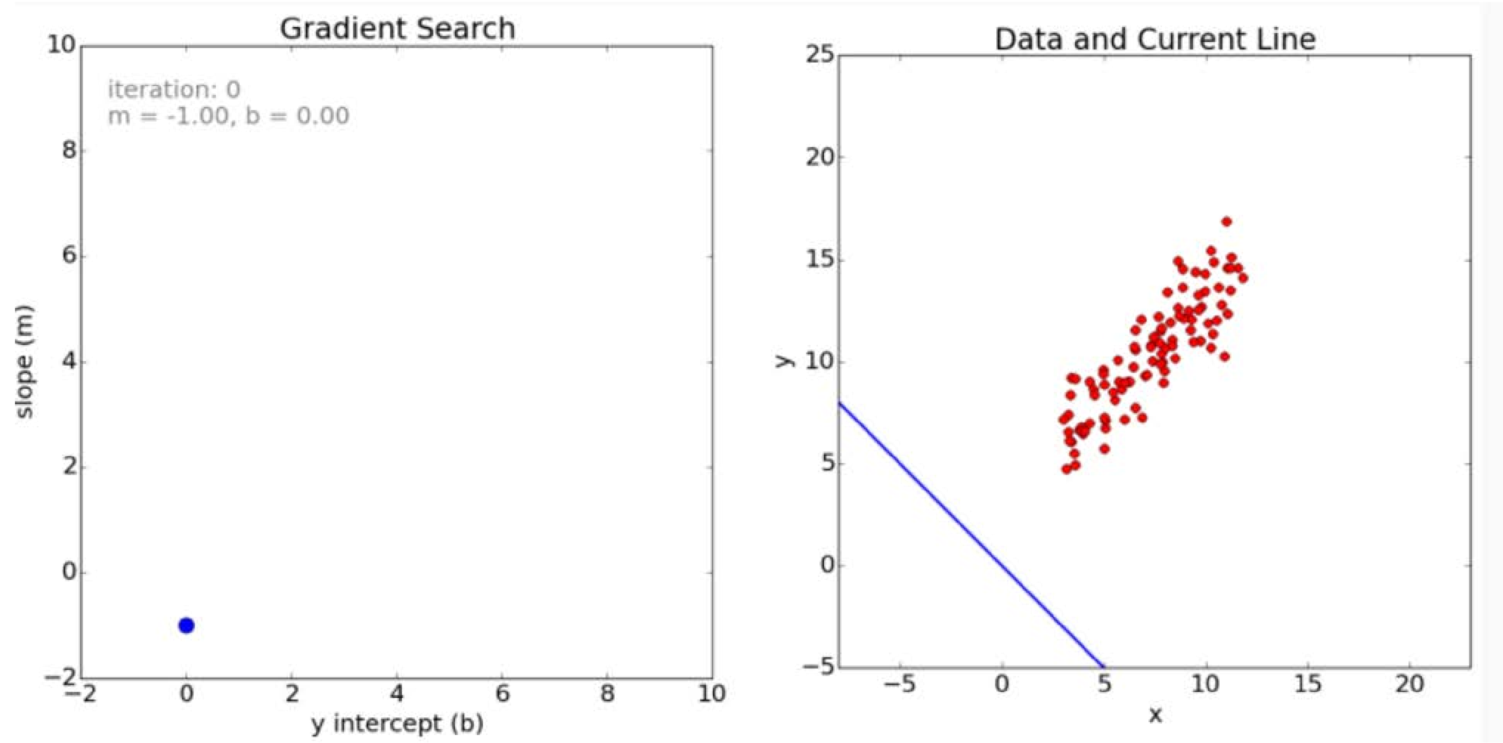
*Essentially, we can picture GD optimization as a hiker (**the weight coefficient**) who wants to climb down a mountain (**cost function**) into a valley (**cost minimum**), and each step is determined by the steepness of the slope (**gradient**) and the leg length of the hiker (**learning rate**)*

Error Depends on the Weight Value



Example: Iteration 0

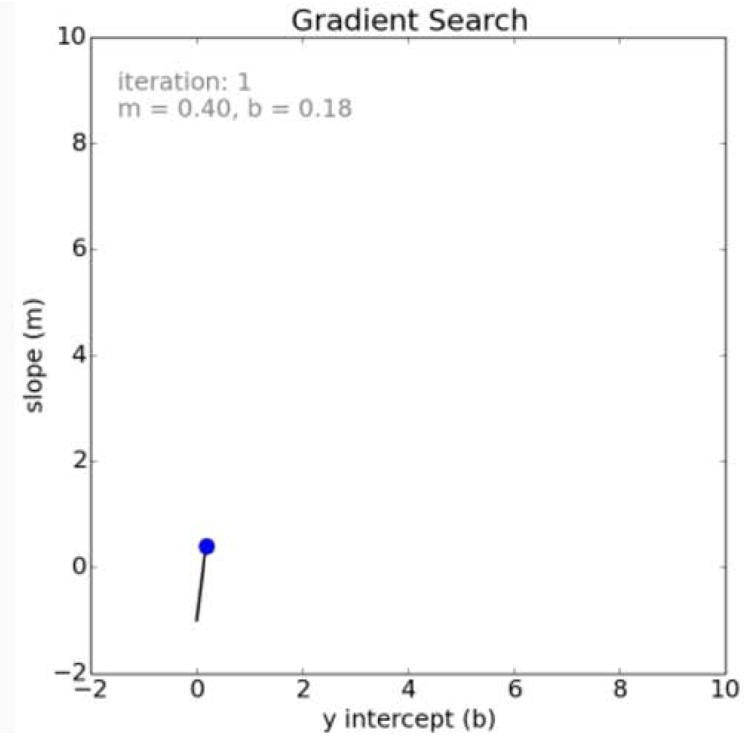
- Start out at point m = -1 and b = 0

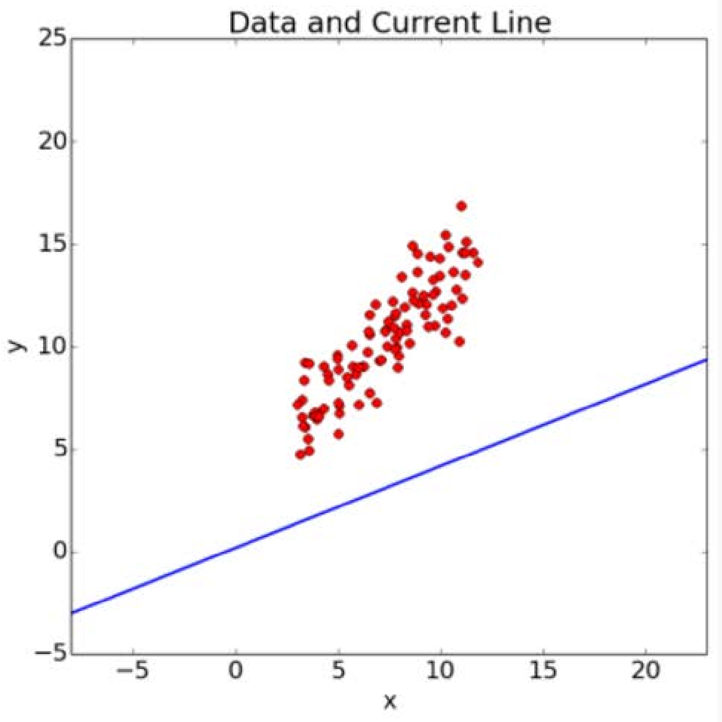


- The left plot displays **the current location of the** gradient descent search (blue dot) and the path taken to get there (black line)
- The right plot displays the corresponding line for the current search location.

Example: Iteration 1

- Each iteration m and b are updated to values that yield slightly lower error than the previous iteration.

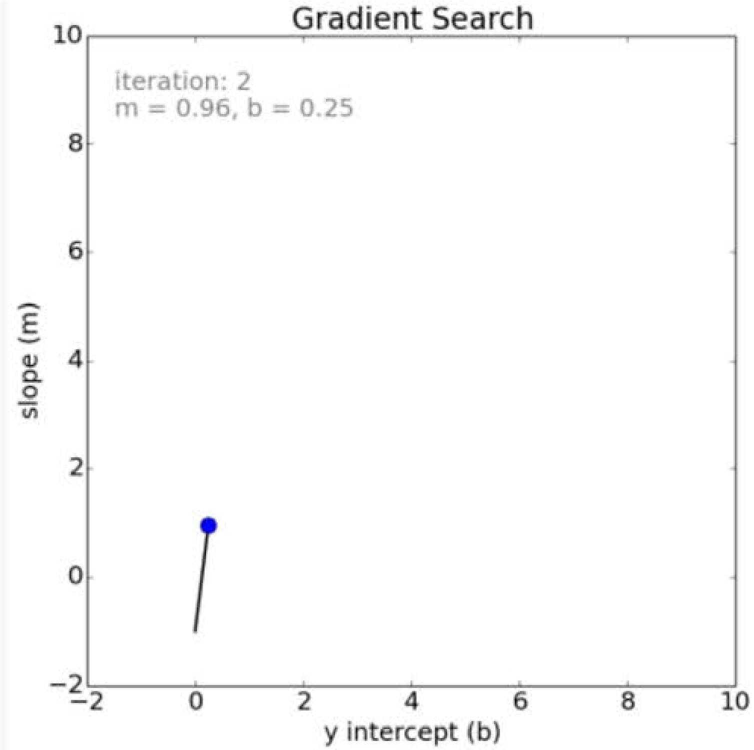


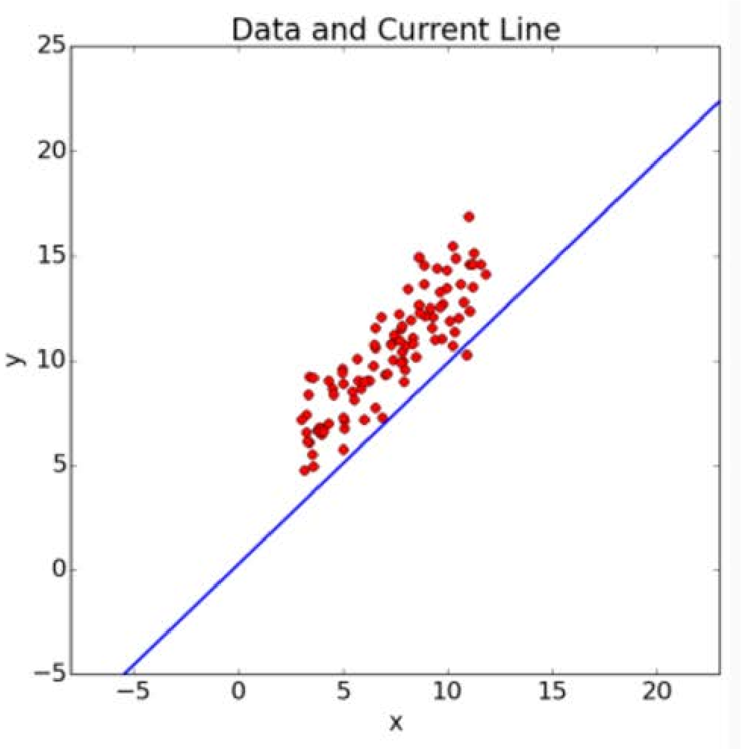


- The left plot displays **the current location of the** gradient descent search (blue dot) and the path taken to get there (black line)
- The right plot displays the corresponding line for the current search location.

Example: Iteration 2

- Each iteration m and b are updated to values that yield slightly lower error than the previous iteration.

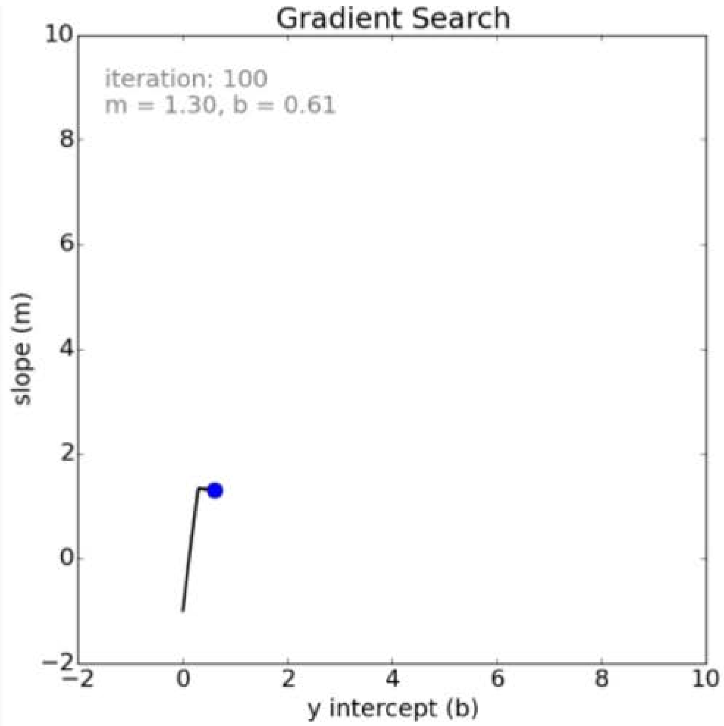


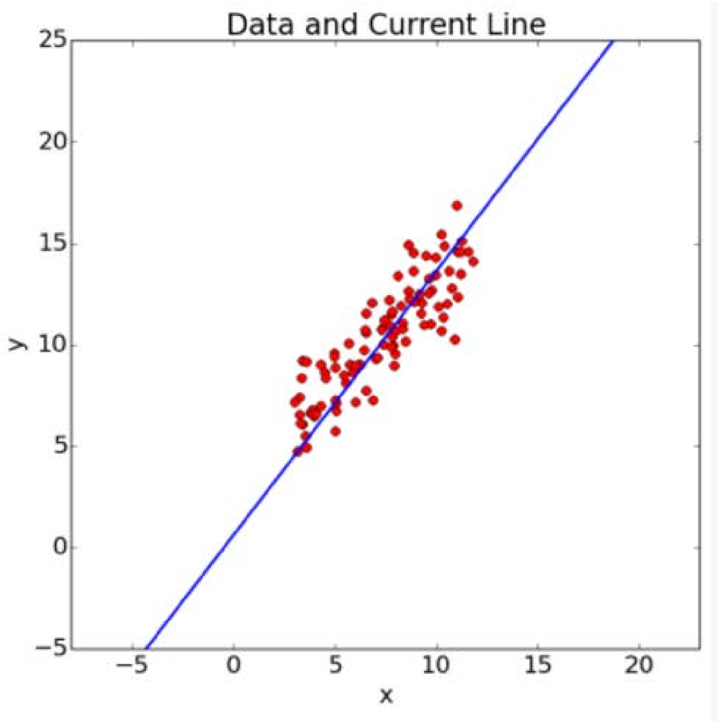


- The left plot displays **the current location of the** gradient descent search (blue dot) and the path taken to get there (black line)
- The right plot displays the corresponding line for the current search location.

Example: Iteration 100

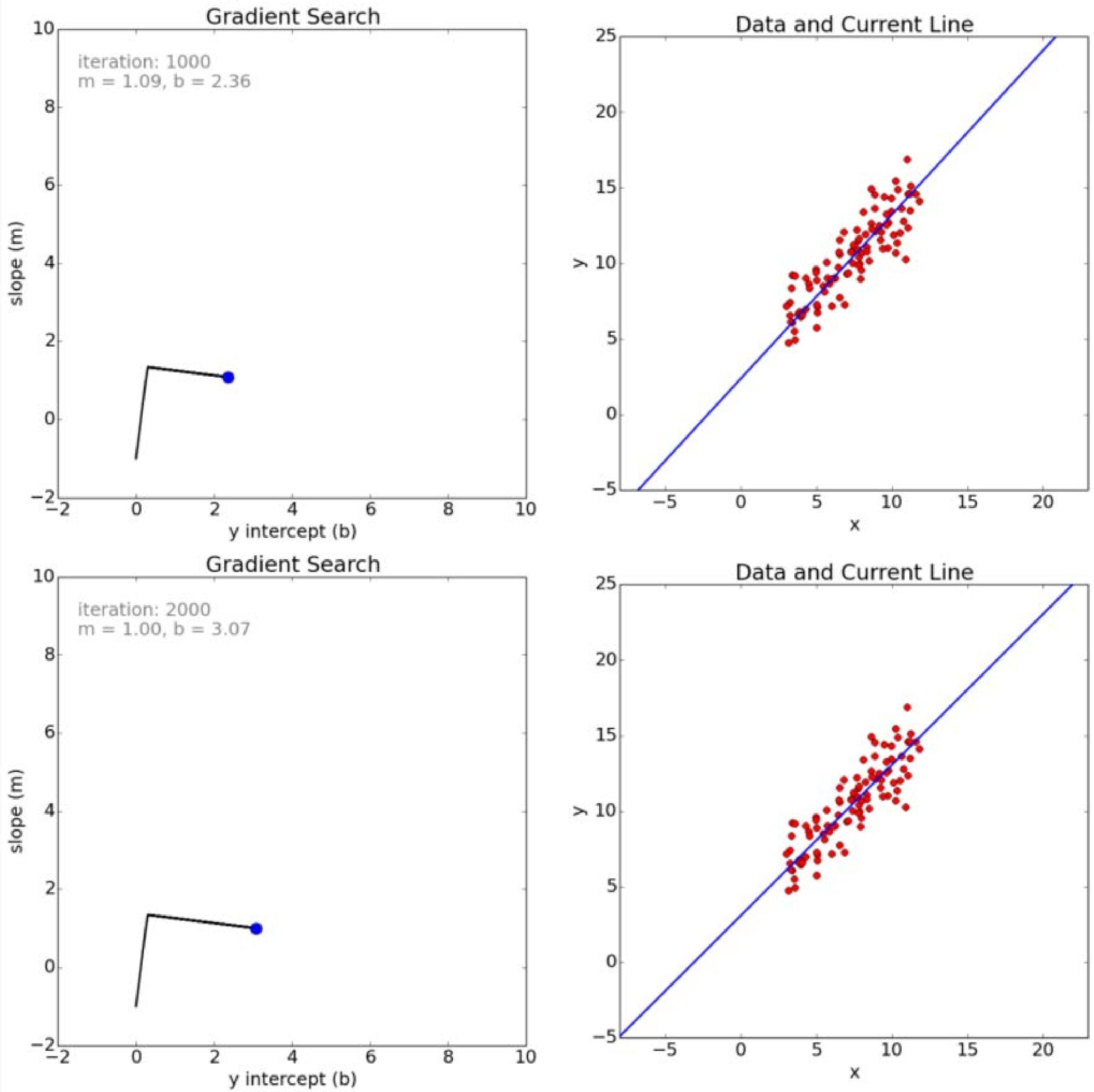
- Each iteration m and b are updated to values that yield slightly lower error than the previous iteration.



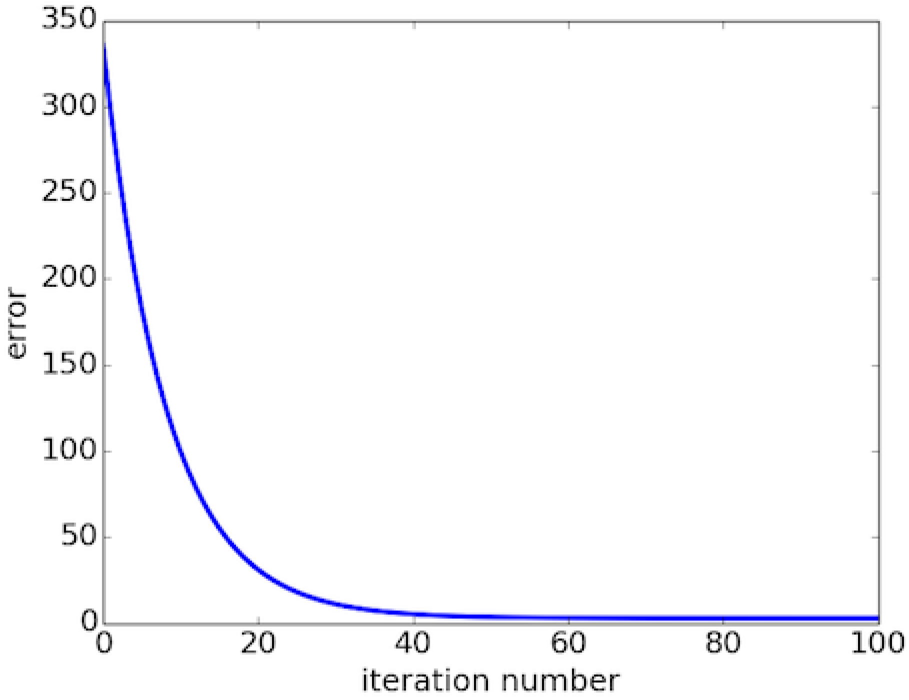


- The left plot displays **the current location of the** gradient descent search (blue dot) and the path taken to get there (black line)
- The right plot displays the corresponding line for the current search location.

Iterations 1,000 and 2,000



Error Change with Each Iteration



- Observe how the error changes as we move toward the minimum.
- A good way to ensure that gradient descent is working correctly is to make sure that the error decreases for each iteration.

Core Concept: Convexity

- In our linear regression problem, there was **only one minimum**.
- Our error surface was convex:
  - Regardless of where we started, we would eventually arrive at the absolute minimum.

- In general, this need not be the case:
  - It's possible to have a problem with local minima that a gradient search can get stuck in.
  - There are several approaches to mitigate this (e.g., stochastic gradient search).

Core Concept: Performance

- We used vanilla gradient descent with a learning rate of 0.0005 in the above example, and ran it for 2000 iterations.
- There are approaches such as a **line search**, that can reduce the number of iterations required.
  - For the above example, line search reduces the number of iterations to arrive at a reasonable solution from several thousand to around 50.

Core Concept: Convergence

- We didn't talk about how to determine when the search finds a solution.
  - This is typically done by looking for small changes in error iteration-to-iteration (e.g., where the gradient is near zero).

Numeric Approximation

SGD: STOCHASTIC GRADIENT DESCENT

- Scikit Learn: http://scikit-learn.org/stable/modules/sgd.html

SGD: Motivation

- In GD optimization, the cost gradient is computed based on the **complete training set**, or **batch GD**.
- In case of very large datasets, using GD is costly:
  - only taking a *single step* for *one pass* over the *entire training set*
  - the larger the training set, the slower the GD algorithm updates the weights and the longer it may take until it converges to the **global cost minimum** (note that the ***SSE cost function is convex***).

```
• for one or more iteration
    • for each weight j
        • w_j := w_j + Δw_j
        • where Δw_j = γ · Σ_{i=1}^{n} (target^(i) − output^(i)) · x_j^(i)
```

- **for one or more iteration**
  - **for each weight** j
    - $w_j := w_j + \Delta w_j$
    - **where** $\Delta w_j = \gamma \cdot \sum_{i=1}^{n} \left(target^{(i)} - output^{(i)}\right) \cdot x_j^{(i)}$

note the summation over
the entire training set

SGD: Iterative or On-Line GD

In Stochastic Gradient Descent (SGD; sometimes also referred to as **iterative** or **on-line GD**), we **don't accumulate the weight updates** as in GD:

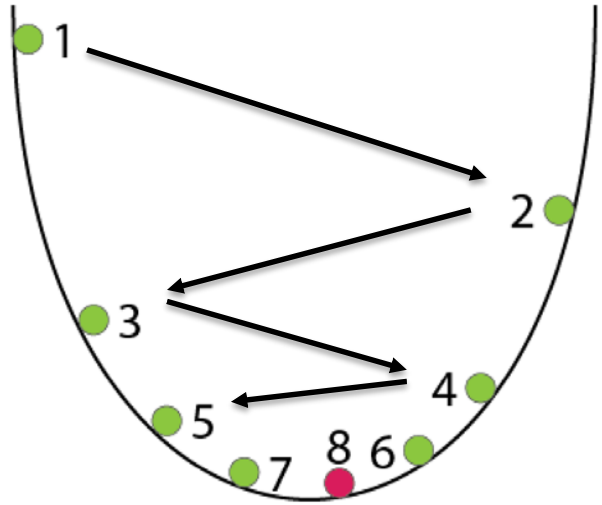
- Instead, we **update the weights after each training sample**:

- **for one or more iteration, or until approximate cost minimum is reached:**
  - **for training sample i**
    - **for each weight j**
      - $w_j := w_j + \Delta w_j$
      - **where** $\Delta w_j = \gamma \cdot \left(target^{(i)} - output^{(i)} \right) \cdot x_j^{(i)}$

note there is NO summation over the entire training set

Why "stochastic" in SGD?

- The term "stochastic" comes from the fact that the **gradient based on a single training sample is a "stochastic approximation" of the "true" cost gradient**
- Due to its stochastic nature, **the path towards the global cost minimum is not "direct" as in GD, but may go "zig-zag"**
- However, it has been shown that SGD almost surely converges to the global cost minimum if the cost function is convex (or pseudo-convex)
- Furthermore, there are different tricks to improve the GD-based learning such as adaptive learning rate



Adaptive Learning Rate

- **Adaptive learning rate, γ to improve GD-based learning**
  - Choosing a decreasing constant α that shrinks the learning rate over time

$$\boldsymbol{\gamma(t+1)} \coloneqq \frac{\boldsymbol{\gamma(t)}}{\boldsymbol{1 + t \times \alpha}}$$

Shuffling: Flavors of SGD

- randomly shuffle samples in the training set
    - for one or more iteration, or until approximate cost minimum is reached:
        - for training sample i
            - compute gradients and perform weight update

- for one or more iteration, or until approximate cost minimum is reached:
    - randomly shuffle samples in the training set
        - for training sample i
            - compute gradients and perform weight update

- for one or more iteration, or until approximate cost minimum is reached:
    - draw random sample from the training set
        - compute gradients and perform weight update

Shuffling: Flavors of SGD

(A)

- randomly shuffle samples in the training set
  - for t iterations, or until approx. cost min is reached:
    - for training sample i
      - compute gradients and perform weight update

(B)

- for t iterations, or until approx. cost min is reached:
  - randomly shuffle samples in the training set
    - for training sample i
      - compute gradients and perform weight update

(C)

- for t iterations, or until approx. cost min is reached:
  - draw random sample from the training set
    - compute gradients and perform weight update

- (A): shuffle the training set only one time in the beginning;
- (B): shuffle the training set after each epoch to prevent repeating update cycles.
- (A) and (B): each training sample is only used once per epoch to update the model weights.
- (C): draw the training samples randomly *with replacement* from the training set. If the number of iterations t is equal to the number of training samples, we learn the model based on a *bootstrap sample* of the training set.

MB-GD: Mini-Batch Gradient Descent

- **Mini-Batch Gradient Descent** (MB-GD) a compromise between batch GD and SGD
- MB-GD updates the model based on ***smaller groups of training samples***:
  - instead of computing the gradient from 1 sample (SGD) or all n training samples (GD), we **compute the gradient from 1 < k < n training samples** (a common mini-batch size is k=50).
- MB-GD converges in ***fewer iterations*** than GD because we update the weights more frequently
- MB-GD utilizes ***vectorized operation***, which typically results in a computational performance gain over SGD

Stochastic Gradient Descent Tricks from Microsoft

http://research.microsoft.com/pubs/192769/tricks-2012.pdf

- Use stochastic gradient descent when training time is the bottleneck
- Prepare the data by random shuffling
- Use preconditioning techniques (see Sections 1.4.3 and 1.5.3)
- Monitoring and debugging: Monitor both the training cost and the validation error
- Check the gradients using finite differences
- Experiment with the learning rates $\gamma(t)$ using a small sample of the training set

More topics to be covered later...

- Negative Sampling
- Stochastic Gradient Descent (SGD) for Logistic Regression
- Gradient Descent-based Techniques with L_2-regularization
- SGD with back propagation

References

- Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations". Online Learning and Neural Networks. Cambridge University Press. ISBN 978-0-521-65263-6
- Bottou, Léon. "Large-scale machine learning with SGD." Proceedings of COMPSTAT'2010. Physica-Verlag HD, 2010. 177-186.
- Bottou, Léon. "SGD tricks." Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. 421-436.
- https://en.wikipedia.org/wiki/Gradient_descent
- http://research.microsoft.com/pubs/192769/tricks-2012.pdf
- https://msdn.microsoft.com/en-us/magazine/dn913188.aspx
- http://sebastianraschka.com/faq/docs/closed-form-vs-gd.html
- https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/