# Pattern Recognition

# ECSE 4410/6410 CAPA

# Fall 2022

## Gradient Descent

Course Instructor - Thirimachos Bourlai

Source: HERE;

ML-Mastery – ML Algorithms 2020 by J. Brownlee

Please Check the 2022 Syllabus

# OVERVIEW

**What we will learn – different perspective:**

- About the gradient descent optimization algorithm
- How gradient descent can be used in algorithms like linear regression
- How gradient descent can scale to very large datasets
- Tips for getting the most from gradient descent in practice.

SIMPLIFIED EXPLANATION

# Gradient Descent – Idea/Concept

**Agile → well-known term in the software development process**

The basic idea behind it is simple:

- build something quickly
- get it out there
- get some feedback
- make changes depending upon the feedback
- repeat the process

**GOAL**: Bring products close to the users and let them guide product improvement and optimization via feedback. The steps taken for improvement are small (rapid iterations) and user is constantly involved.

**Idea**:

- Start with a solution as soon as possible
- Measure and iterate as frequently as possible

is basically Gradient descent under the hood.

SIMPLIFIED EXPLANATION

# Gradient Descent

“Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost)”

**Gradient descent** is best used when the parameters:

1. Cannot be calculated analytically (e.g., using linear algebra) and
2. Must be searched for *by an optimization algorithm*

SIMPLIFIED EXPLANATION

# GD: Intuition

- Think of a large bowl like what you would eat cereal out of or store fruit in
- This bowl is a plot of the cost function (f)
- A **random position** on the surface of the bowl is the cost of the current values of the coefficients
- The **bottom of the bowl** is the cost of the best set of coefficients, the minimum of the function

Large Bowl
Photo by William Warby, some rights reserved.

SIMPLIFIED EXPLANATION

# GD: Intuition

**GOAL**:

- Continue to try different values for the coefficients
- Evaluate values cost, and
- Select new coefficients that have a slightly improved (lower) cost

- Repeating this process enough times leads to the bottom of the bowl

- At the end of the process, you **will know** the values of the coefficients that **result in the minimum cost**

SIMPLIFIED EXPLANATION

# Gradient Descent Procedure

- The procedure starts with initialization of function coefficients (e.g., 0, random numbers)

  COEF = 0.0

- Plug COEF into the function:

  cost = f(COEF)

- Calculating the cost:

  cost = evaluate(f(COEF))

SIMPLIFIED EXPLANATION

# Gradient Descent Procedure

- The derivative of the cost is calculated
  - The derivative → slope of the function at a given point
- Knowing the derivative

  = knowing the slope

  = knowing the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration

delta = derivative(cost)

SIMPLIFIED EXPLANATION

# Gradient Descent Procedure

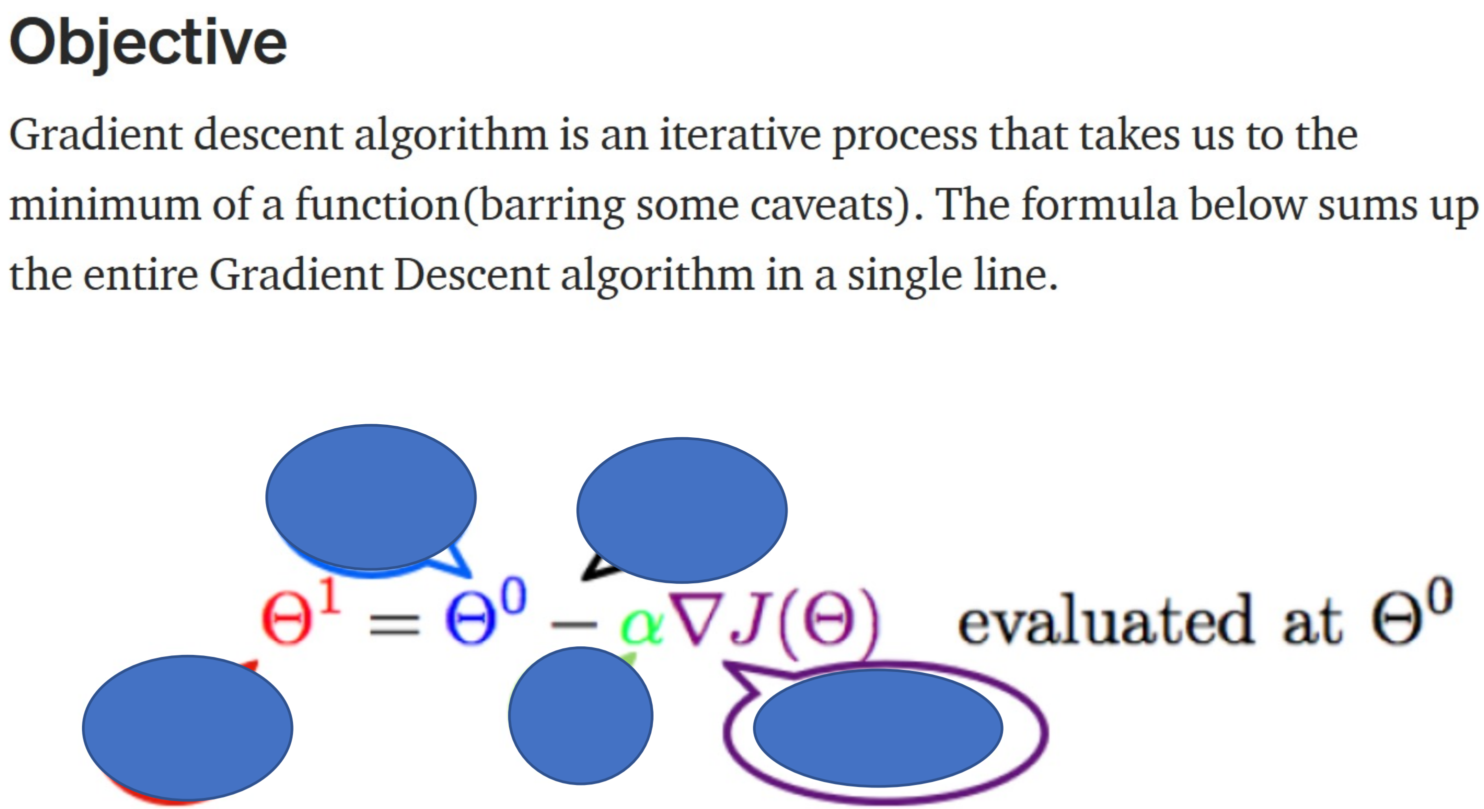
- From the derivative we learn which direction is downhill
- Next, we can update the coefficient values
- A **learning rate** parameter (**alpha**) is specified that controls how much the coefficients can change on each update.

COEF = COEF - (**alpha** × delta)

- This process is repeated **until** the
  - **Cost is 0.0** or
  - **No further improvements in cost can be achieved**

IN-DEPTH EXPLANATION

# Gradient Descent



## Objective

Gradient descent algorithm is an iterative process that takes us to the minimum of a function(barring some caveats). The formula below sums up the entire Gradient Descent algorithm in a single line.

$$\Theta^1 = \Theta^0 - \alpha \nabla J(\Theta) \quad \text{evaluated at } \Theta^0$$

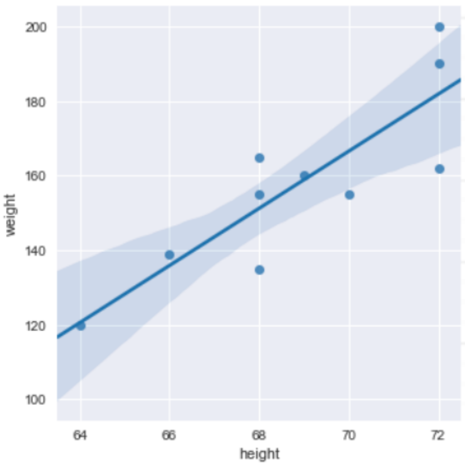
https://www.coursehero.com/file/27927651/Gradient-Descentpdf/

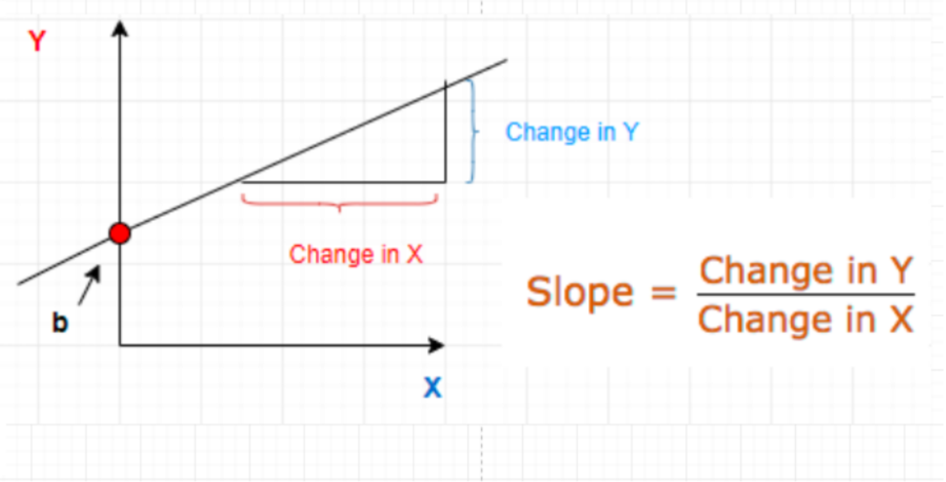
IN-DEPTH EXPLANATION

# A Machine Learning Model

- Consider a bunch of data points in a 2 D space. Assume that the data is related to the height and weight of a group of students. We are trying to predict some kind of relationship between these quantities so that we could predict the weight of some new students afterwards. This is essentially a simple example of a supervised Machine Learning technique.

- Let us now draw an arbitrary line in space that passes through some of these data points. The equation of this straight line would be `Y = mX + b` where **m** is the slope and **b** is its intercept on the Y-axis.

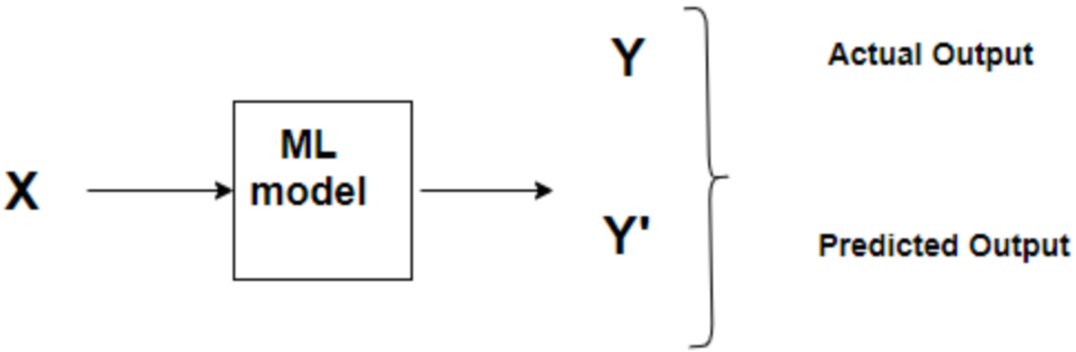




IN-DEPTH EXPLANATION

# Predictions

Given a known set of inputs and their corresponding outputs, A machine learning model tries to make some predictions for a new set of inputs.



ML process

The Error would be the difference between the two predictions.

**Error = Y'(Predicted) - Y(Actual)**

This relates to the idea of a **Cost function or Loss function.**

IN-DEPTH EXPLANATION

# Cost Function

A **Cost Function/Loss Function** evaluates the performance of our Machine Learning Algorithm. The **Loss function** computes the error for a single training example while the **Cost function** is the average of the loss functions for all the training examples. Henceforth, I shall be using both the terms interchangeably.

> A Cost function basically tells us ‘ how good’ our model is at making predictions for a given value of m and b.

Let’s say, there are a total of ’N’ points in the dataset and for all those ’N’ data points we want to minimize the error. So the Cost function would be the total squared error i.e

$$Cost = \frac{1}{N}\sum_{i=1}^{N}(Y' - Y)^2$$

IN-DEPTH EXPLANATION

# Cost Function

$$Cost = \frac{1}{N}\sum_{i=1}^{N}(Y' - Y)^2$$

Why do we take

the squared differences and

**not** the absolute differences?

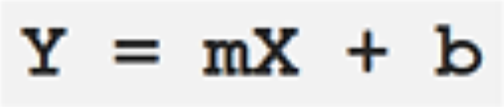
IN-DEPTH EXPLANATION

# Minimizing the Cost Function

*The goal of any ML Algorithm is to minimize the Cost Function.*

The lower the error between the **actual** and the **predicted** values → the algorithm has done a good job in learning

Since we want the **lowest error value**, we want those **‘m’** and **‘b’** values that give the smallest possible error.



Y = mX + b

IN-DEPTH EXPLANATION

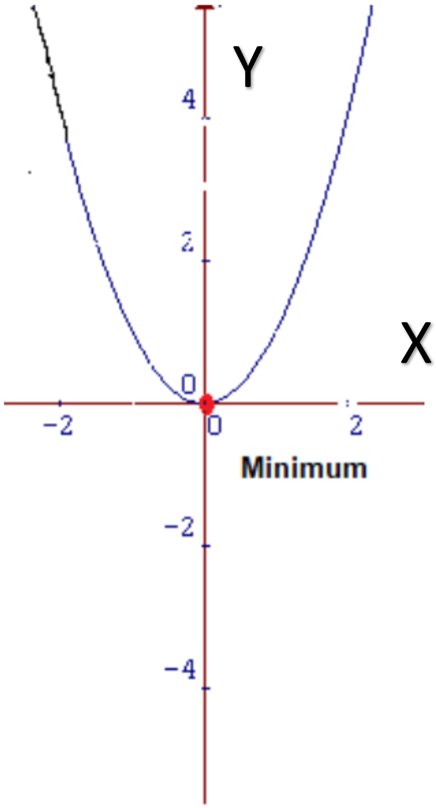
# How do we minimize any function?

- Our **Cost Function** is of the form **Y = $X^2$**. In a Cartesian coordinate system, this is an equation for a parabola:

- Y is min at the red dot, where X is min

- In practice, this may not always be the case (especially in case of higher dimensions)

- For those cases, we need an algorithm to locate the local minimum → Gradient Descent



Parabola

IN-DEPTH EXPLANATION

# GD -- Taking Actions

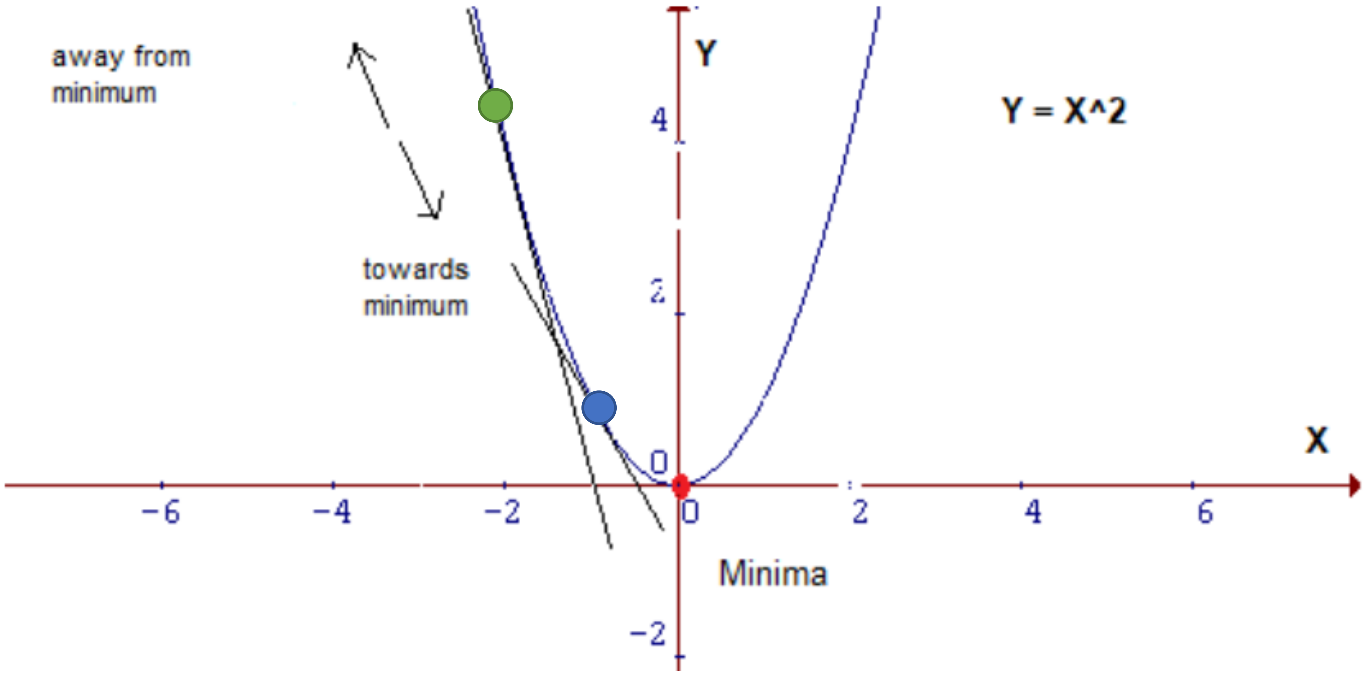
- After initialization → possible steps would be to go upward or downward, and then, to either take a bigger step or a little step to reach your destination.

Essentially, there are two things that you should know to reach the minima, i.e., which way to go and how big a step to take.

- **GD** helps us to **make these decisions** → using **derivatives**
- **Derivative** → calculated as **the slope** of the graph at a particular point
- **Slope**: described by drawing a tangent line to the graph at the point
  - By computing this tangent line → we compute the desired direction to reach the minima

IN-DEPTH EXPLANATION

# The minimum value



The slope at the blue point is less steep than that at the green point which means it will take much smaller **steps** to reach the minimum from the blue point than from the green point.

IN-DEPTH EXPLANATION

# Math Interpretation of Cost Function

**Parameters with small changes:**

$$m = m - \delta m$$
$$b = b - \delta b$$

**Given Cost Function for 'N' no of samples**

$$Cost = \frac{1}{N}\sum_{i=1}^{N}(Y_i' - Y_i)^2$$

**Cost function is denoted by J where J is a function of m and b**

$$J_{m,b} = \frac{1}{N}\sum_{i=1}^{N}(Y_i' - Y_i)^2$$

**Substituting the term Y'-Y with error for simplicity**

$$J_{m,b} = \frac{1}{N}\sum_{i=1}^{N}(Error_i)^2$$

**IDEA:** By computing the derivative/slope of the function → we find the minimum of a function.

IN-DEPTH EXPLANATION

# Learning Rate

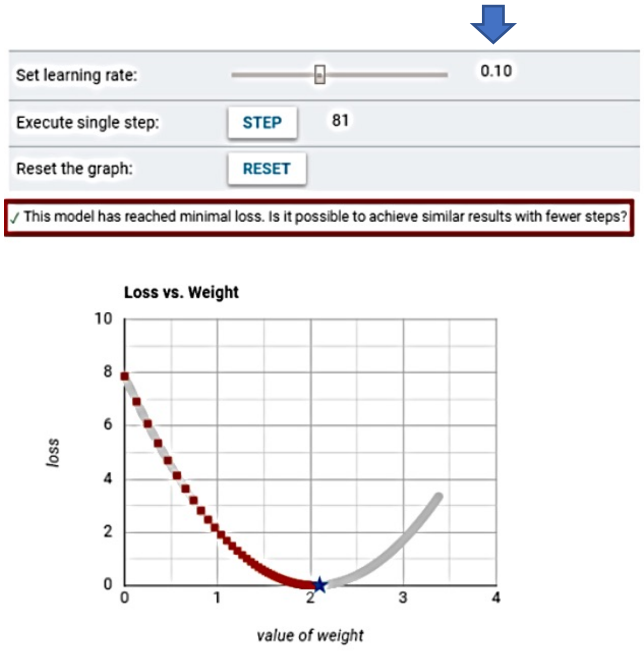
**Learning Rate:** size of steps taken to reach the minimum or bottom

- Higher learning rate (via larger steps) → **risk of overshooting the minima**.
- Small steps/smaller learning rates → **consume a lot of time to reach minima**.

See visualizations in next slide

# Learning Rate

- **Figure 1 vs Figure 3**: we reach the minimum point from TOO many steps to the minimum number of steps.
- **Figure 3**: the optimum learning rate for this problem.

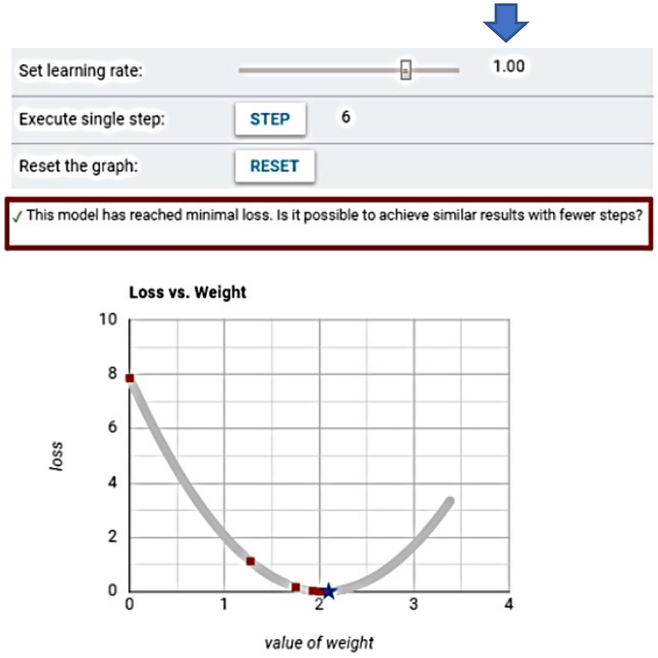


Set learning rate: 0.10

Execute single step: STEP 81

Reset the graph: RESET

✓ This model has reached minimal loss. Is it possible to achieve similar results with fewer steps?

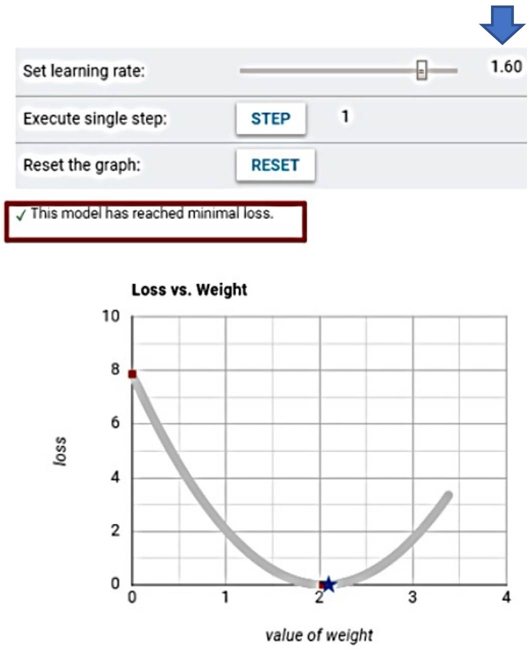


Set learning rate: 1.00

Execute single step: STEP 6

Reset the graph: RESET

✓ This model has reached minimal loss. Is it possible to achieve similar results with fewer steps?



Set learning rate: 1.60

Execute single step: STEP 1

Reset the graph: RESET

✓ This model has reached minimal loss.

https://developers.google.com/machine-learning/crash-course/fitter/graph

IN-DEPTH EXPLANATION

# Learning Rate

LIVE DEMO

HERE – Live GD

IN-DEPTH EXPLANATION

# Derivatives

- Optimization algorithms like gradient descent use derivates to decide whether to increase or decrease the weights in order to increase or decrease any objective function.

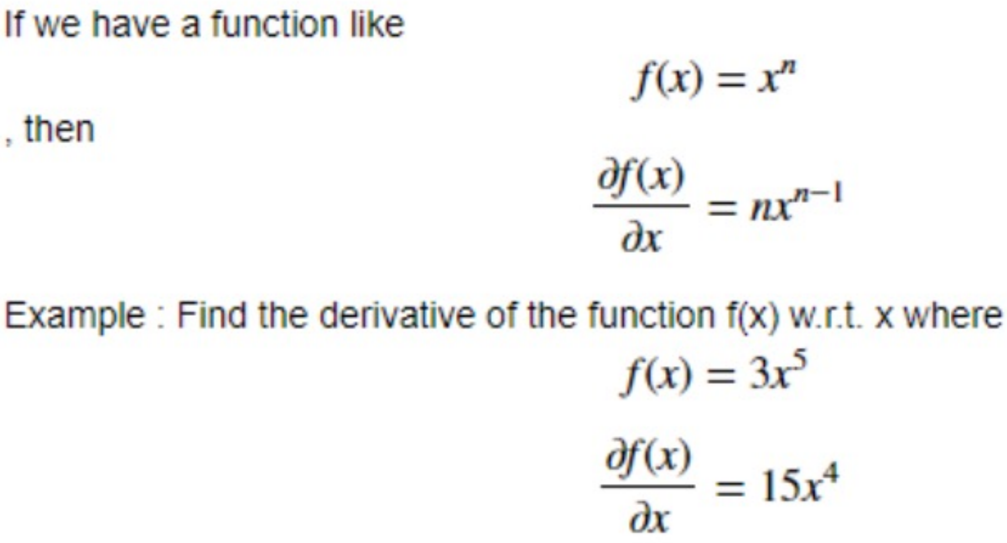
- By computing the **derivative of a function**, we **know in which direction** to proceed to minimize it.

- Primarily we shall be dealing with two concepts from calculus :
  - **Power Rule**
  - **Chain Rule**

IN-DEPTH EXPLANATION

# Derivatives

Power rule calculates the derivative of a variable raised to a power.



If we have a function like

$$f(x) = x^n$$

, then

$$\frac{\partial f(x)}{\partial x} = nx^{n-1}$$

Example : Find the derivative of the function f(x) w.r.t. x where

$$f(x) = 3x^5$$

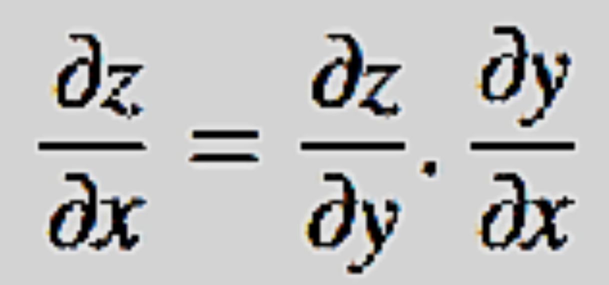
$$\frac{\partial f(x)}{\partial x} = 15x^4$$

IN-DEPTH EXPLANATION

# Derivatives

“The **chain rule** is used for calculating the derivative of composite functions. The chain rule can also be expressed in Leibniz’s notation as follows:

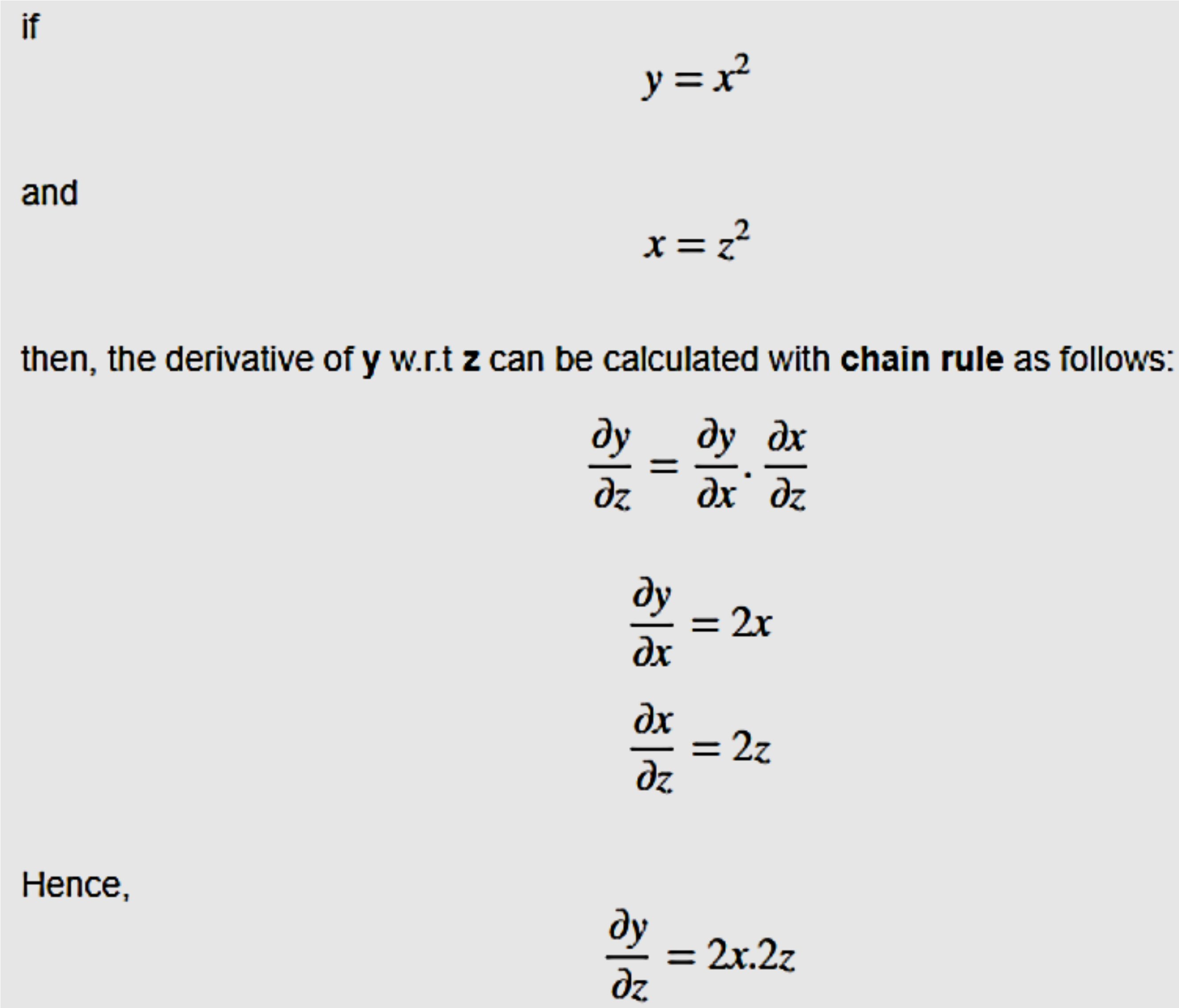
- If a variable ***z*** depends on the variable ***y***, which itself depends on the variable ***x***, so that *y* and *z* are dependent variables, then *z*, via the intermediate variable of *y*, depends on *x* as well.
- This is called the chain rule and is mathematically written as:”



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

IN-DEPTH EXPLANATION

# Derivatives



if

$$y = x^2$$

and

$$x = z^2$$

then, the derivative of **y** w.r.t **z** can be calculated with **chain rule** as follows:

$$\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x} . \frac{\partial x}{\partial z}$$

$$\frac{\partial y}{\partial x} = 2x$$

$$\frac{\partial x}{\partial z} = 2z$$

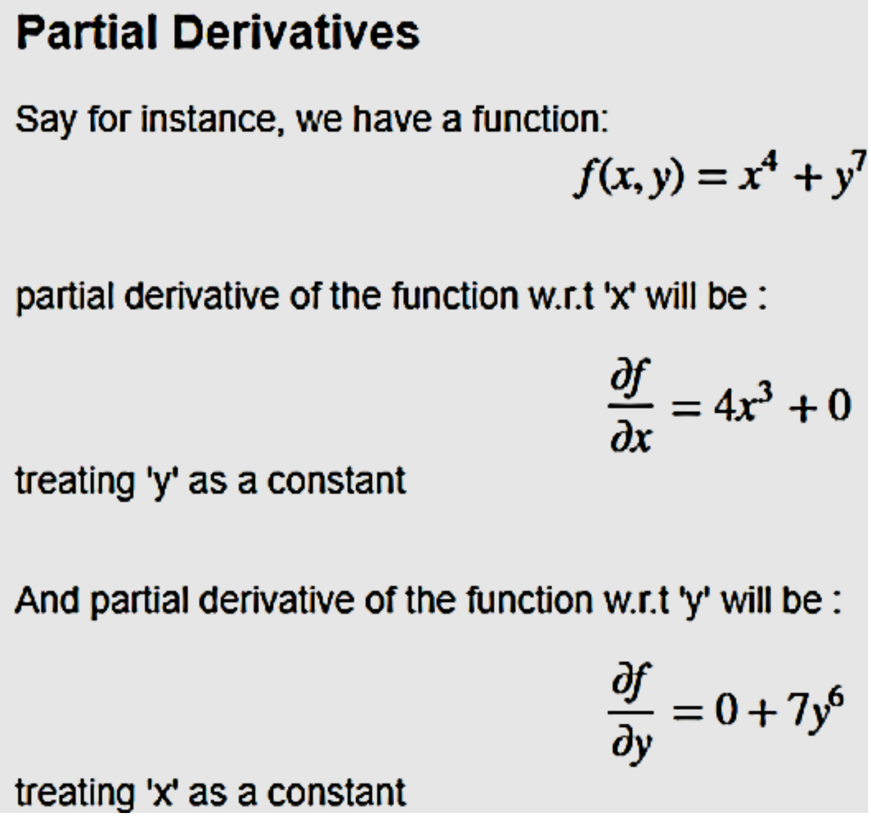
Hence,

$$\frac{\partial y}{\partial z} = 2x.2z$$

IN-DEPTH EXPLANATION

# Derivatives

- Using the **Power** and **Chain** Rule for derivatives, let’s calculate how Cost function changes relative to “m” and “b” --- Y= m · X + b
- This deals with the concept of **partial derivatives**.
- Let’s see this example:



**Partial Derivatives**

Say for instance, we have a function:

$$f(x,y) = x^4 + y^7$$

partial derivative of the function w.r.t 'x' will be :

$$\frac{\partial f}{\partial x} = 4x^3 + 0$$

treating 'y' as a constant

And partial derivative of the function w.r.t 'y' will be :

$$\frac{\partial f}{\partial y} = 0 + 7y^6$$

treating 'x' as a constant

IN-DEPTH EXPLANATION

# Calculating Gradient Descent

- Let us now apply the knowledge of these rules of calculus in our original equation and find the derivative of the Cost Function w.r.t to both **‘m’** and **‘b’**.
- Revising the Cost Function equation :

$$J_{m,b} = \frac{1}{N}\sum_{i=1}^{N}(Error_i)^2$$

To keep things simple, we will assume that we are looking at each error one at a time.

$$\frac{\partial J}{\partial m} = 2.Error.\frac{\partial}{\partial m}Error$$

$$\frac{\partial J}{\partial b} = 2\cdot Error.\frac{\partial}{\partial b}Error$$

Let’s calculate the **gradient of Error** w.r.t to both m and b :

$$\frac{\partial}{\partial m}\mathrm{Error} = \frac{\partial}{\partial m}(\mathrm{Y}' - \mathrm{Y})$$

$$\frac{\partial}{\partial m}\mathrm{Error} = \frac{\partial}{\partial m}(\mathrm{mX} + \mathrm{b} - \mathrm{Y})$$

constants

$$\frac{\partial}{\partial m}\mathrm{Error} = \mathbf{x}$$

$$\frac{\partial}{\partial b}\mathrm{Error} = \frac{\partial}{\partial b}(\mathrm{Y}' - \mathrm{Y})$$

$$\frac{\partial}{\partial b}\mathrm{Error} = \frac{\partial}{\partial b}(\mathrm{mX} + \mathrm{b} - \mathrm{Y})$$

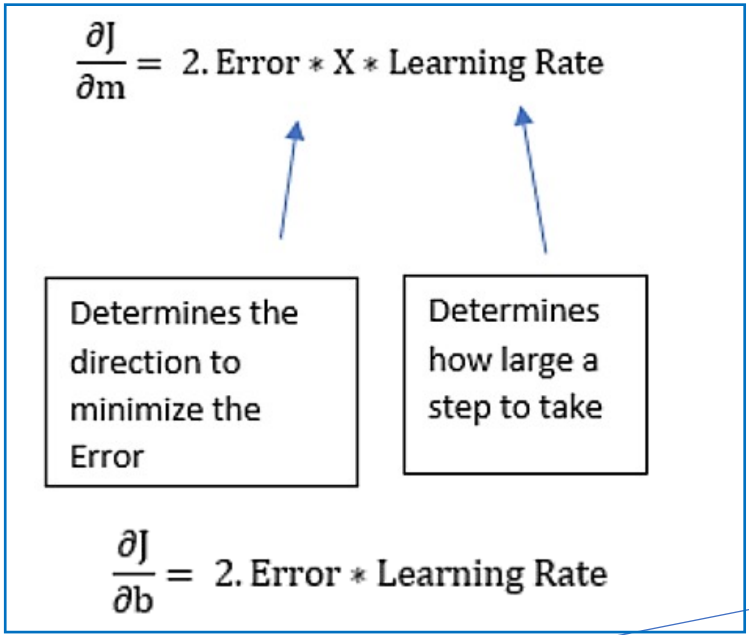
constants

$$\frac{\partial}{\partial b}\mathrm{Error} = 1$$

IN-DEPTH EXPLANATION

# Calculating Gradient Descent

Plugging the values back in the cost function and multiplying it with the learning rate:



$$\frac{\partial J}{\partial m} = 2.\,\text{Error} * X * \text{Learning Rate}$$

Determines the direction to minimize the Error
Determines how large a step to take

$$\frac{\partial J}{\partial b} = 2.\,\text{Error} * \text{Learning Rate}$$

This **2** in this equation isn't that significant since it just says that we have a learning rate twice as big or half as big. **Let's just get rid of it too.**

$$\frac{\partial J}{\partial m} = \text{Error} * X * \text{Learning Rate}$$

$$\textit{Since } m = m - \delta m$$

$$m^1 = m^0 - \text{Error} * X * \text{Learning Rate}$$

$$\frac{\partial J}{\partial b} = \text{Error} * \text{Learning Rate}$$

$$\textit{Since } b = b - \delta b$$

$$b^1 = b^0 - \text{Error} * \text{Learning Rate}$$

- $m^1$,$b^1$ = **next position** parameters
- $m^o$,$b^o$ = **current position** parameters

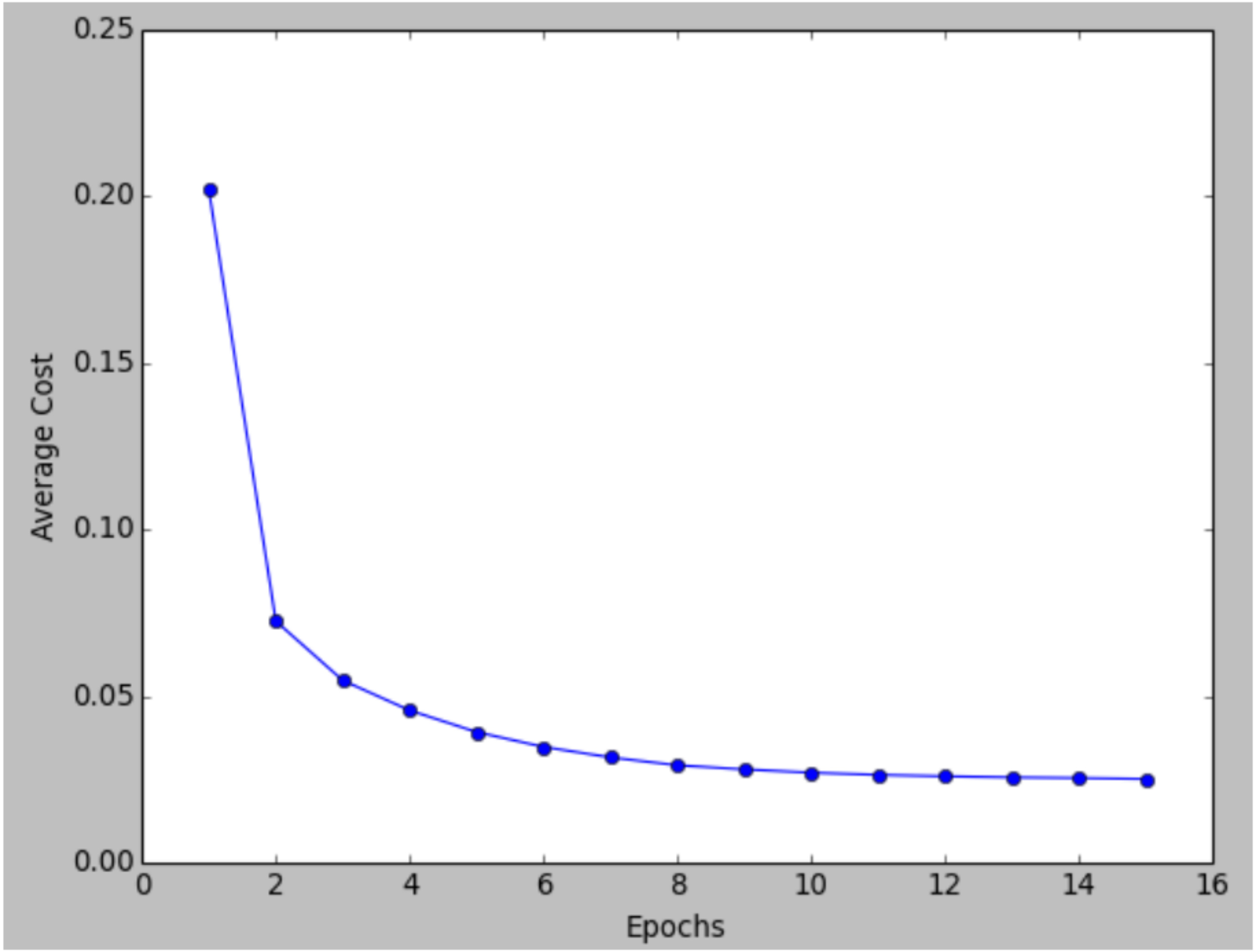
# Batch Gradient Descent

- **All the training data is taken into consideration** to take a single step in the optimization process
- **Process**:
  - We take the **average** of the gradients of all the training examples.
  - Then, we use that **mean gradient** to update our parameters

    → that's just one step of gradient descent in one epoch.
- It's **ideal** for convex or relatively smooth error manifolds, where the movement is somewhat directly towards an optimum solution

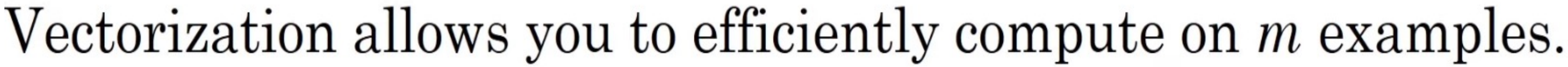
IN-DEPTH EXPLANATION

# Batch Gradient Descent



Cost vs Epochs (Source: https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php)

# Batch Gradient Descent



Vectorization allows you to efficiently compute on $m$ examples.

i.e., it allows you to use the whole training set explicitly without any for loop.

Say you have **m training examples**:

$X = [x^{(1)}, x^{(2)}, x^{(3)}, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ x^{(m)}]$

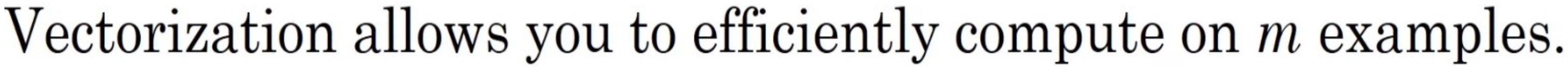
- Dimension of X = ($n_x$ , m), m examples, $n_x$ the features per example

$Y = [y^{(1)}, y^{(2)}, y^{(3)}, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ y^{(m)}]$

- Dimension of Y = (1, m)

Vectorization allows us to process all examples relatively quickly

# Batch Gradient Descent



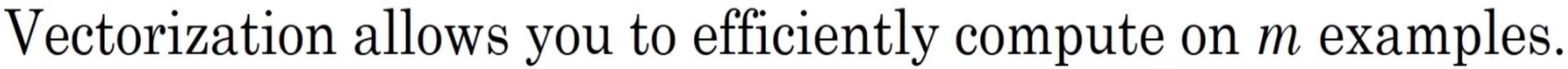
Vectorization allows you to efficiently compute on $m$ examples.

## But can be also slow! How?

Say, m = 5million, 10million or more:

1. Then, we will have to process the entire training set **before taking 1 step on Gradient Descent**
2. Next, we repeat processing the entire training set again and we keep doing that for each GD step we need to take

# Batch Gradient Descent



Vectorization allows you to efficiently compute on $m$ examples.

But can be also slow! How?

So, the question is:

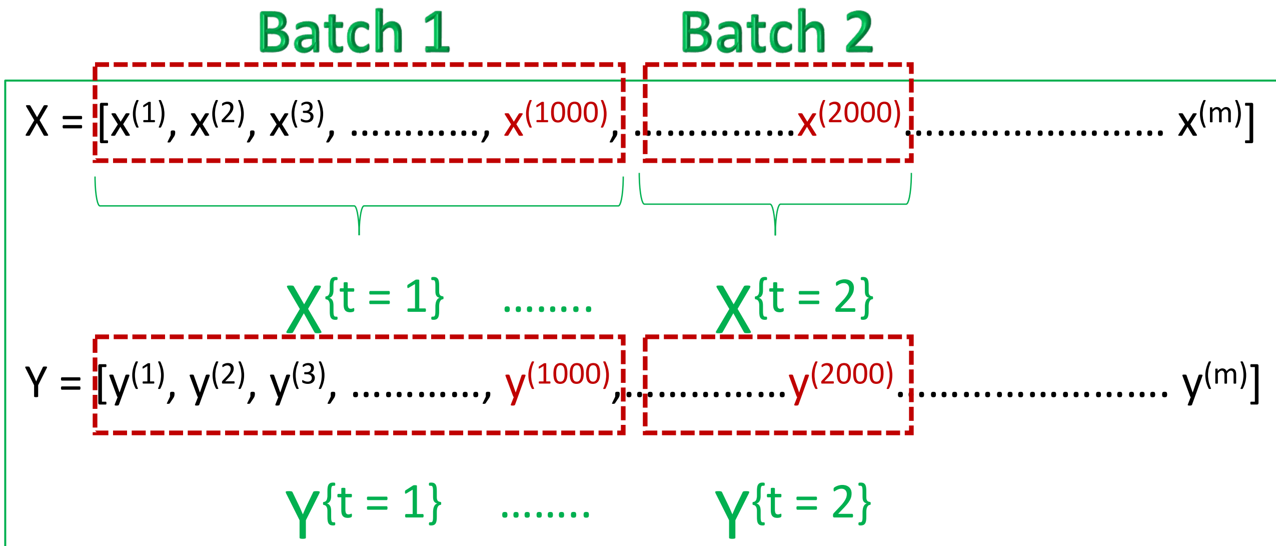
- Can we take a step without having to PROCESS THE WHOLE TRAINING SET, every single time?

# Mini-Batch Gradient Descent

So, the question is:

- Can we take a step without having to PROCESS THE WHOLE TRAINING SET, every single time?

**What if we split the 5 million training samples to 5,000 mini-batches of 1000 examples?**



Batch 1 Batch 2

$X = [x^{(1)}, x^{(2)}, x^{(3)}, \ldots\ldots\ldots\ldots, x^{(1000)}, \ldots\ldots\ldots\ldots x^{(2000)} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots x^{(m)}]$

$X^{\{t = 1\}}$ …….. $X^{\{t = 2\}}$

$Y = [y^{(1)}, y^{(2)}, y^{(3)}, \ldots\ldots\ldots\ldots, y^{(1000)}, \ldots\ldots\ldots\ldots y^{(2000)} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots y^{(m)}]$

$Y^{\{t = 1\}}$ …….. $Y^{\{t = 2\}}$

- Mini-batch → (t) index, $X^{\{t\}}$, $Y^{\{t\}}$, is the mini batch number

>>> (i) index in the training set: x(i), is the ith training example

>>> (l) index, lth layer of the Neural Network → if we are working with NNs

# Batch vs. Mini-Batch Gradient Descent

- Batch GD:
  - When you process the **entire training set** all at the same time, the entire batch of training examples all at the same time

- Mini-Batch GD:
  - Refers to the algorithm that processes a single mini-batch [$X^{\{t\}}$, $Y^{\{t\}}$], at the same time --- rather than processing the ENTIRE training set at the same time ---

Mini-Batch RUNS **MUCH FASTER THAN** Batch-GD

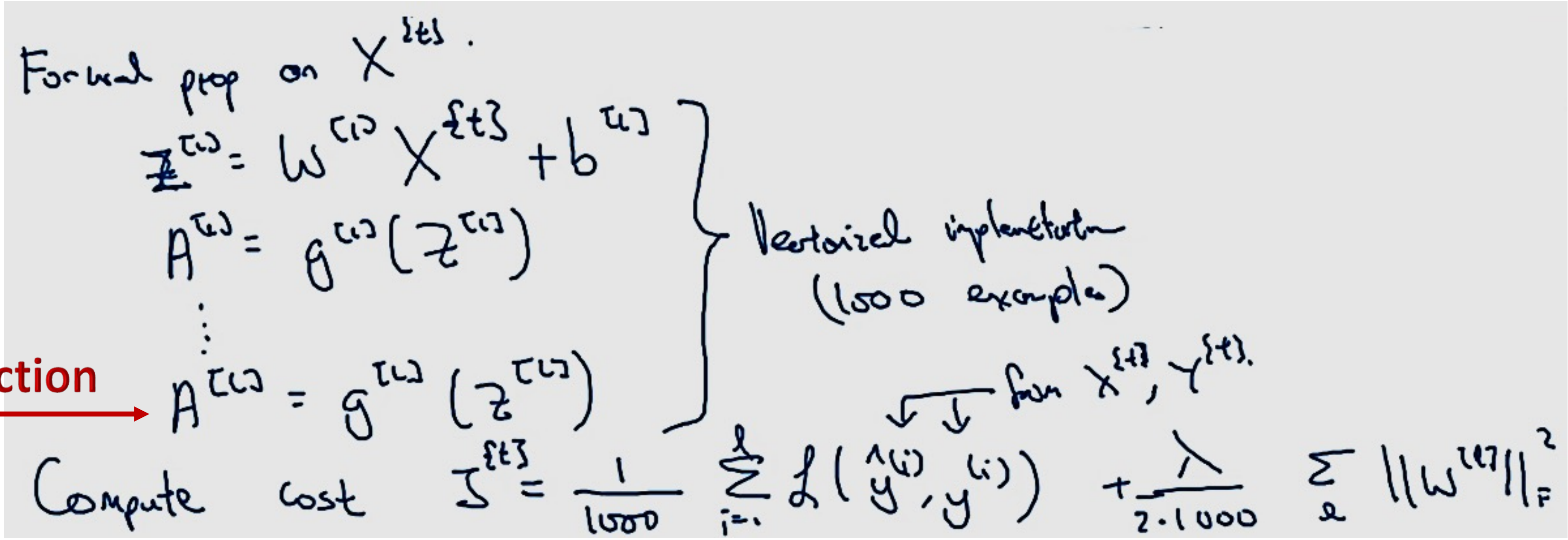
# Mini-Batch Gradient Descent

Mini-Batch GD – we have a **FOR LOOP**

For t=1, 2, …., 5000

// (i.e., 5000 batches that we decided we have)

{ We run 1 step of GR, $[X^{\{t\}}, Y^{\{t\}}]$, for the tth step (t=1) of 1000 examples



Forward prop on $X^{\{t\}}$.

$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

⋮

$A^{[L]} = g^{[L]}(Z^{[L]})$

Vectorized implementation (1000 examples)

from $X^{\{t\}}, Y^{\{t\}}$.

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$

Prediction

// where , L= loss, J= Cost Function

:

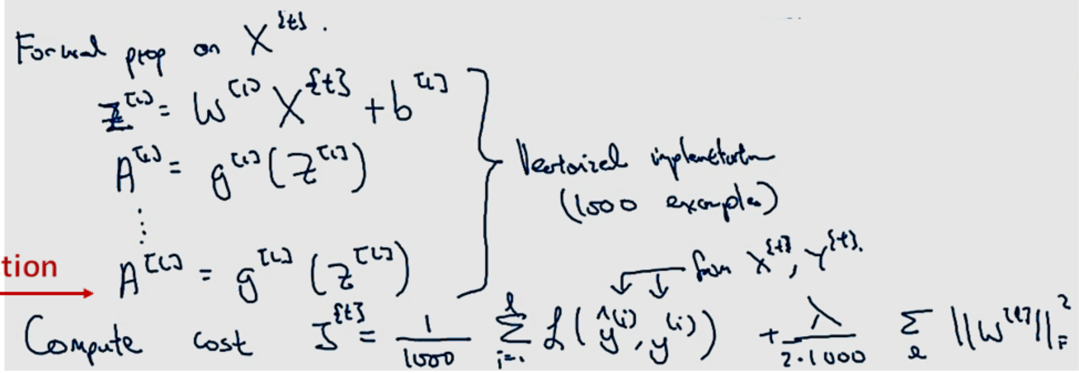
# Mini-Batch Gradient Descent (Loop)

{

For t=1, 2, …., 5000

// (i.e., 5000 batches that we decided we have)

{ We run 1 step of GR, [X{t}, Y{t}], for the tth step (t=1) of 1000 examples



Forward prop on $X^{\{t\}}$.

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$\vdots$$

$$A^{[L]} = g^{[L]}(Z^{[L]})$$

Vectorized implementation (1000 examples)

Prediction

from $X^{\{t\}}, Y^{\{t\}}$.

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2\cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$

// where , L= loss, J= Cost Function

- Next, we perform Backpropagation, to compute the gradients with respect to J{t} , using [X{t}, Y{t}]
- Next, we update the weights

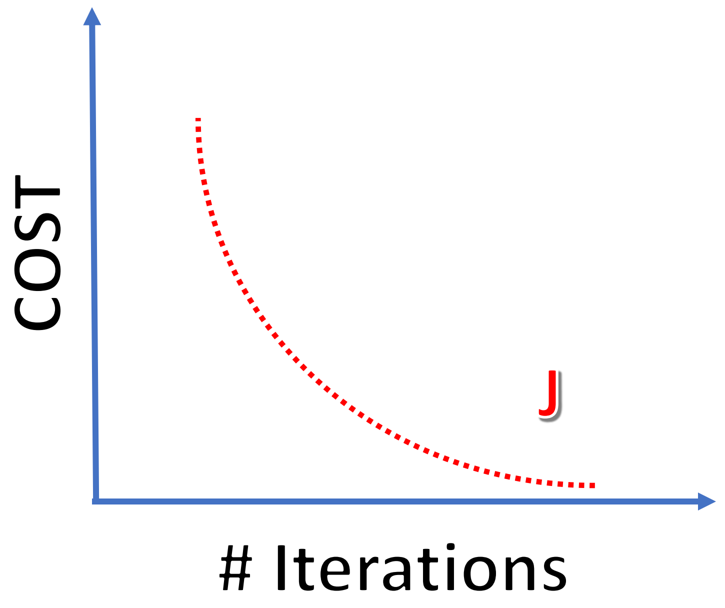
$$W^{\{l\}} := W^{\{l\}} - a\cdot\mathbf{d}(W^{\{l\}}),\ b^{\{l\}} := b^{\{l\}} - a\cdot\mathbf{d}(b^{\{l\}})$$

} // 1 **EPOCH** OF TRAINING, i.e. THIS IS A **SINGLE PASS THROUGH OUR TRAINING SET** using mini-batch GD

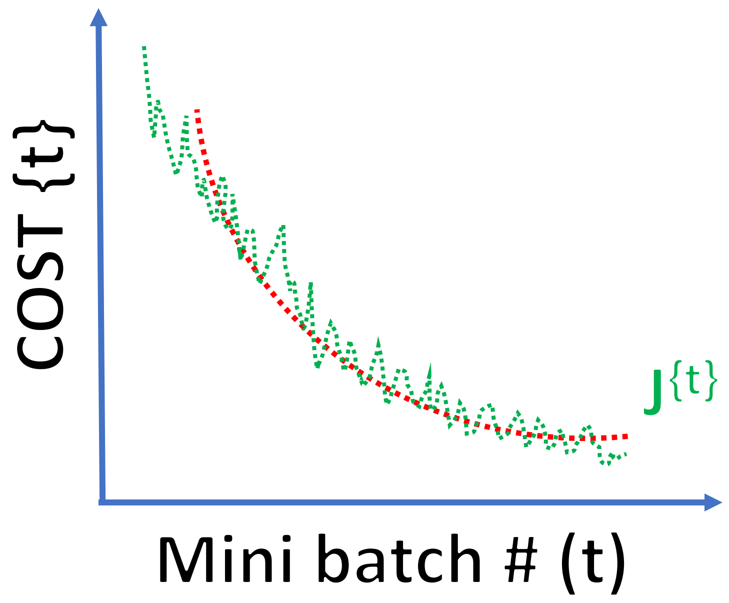
} // MULTIPLE PASSES UNTIL WE **CONVERGE**

# Training with Mini-Batch Gradient Descent

## Batch GD



## Mini-Batch GD



**Q: Why does it go up and down**?

**A:** Maybe the X{1}Y{1} is a GOOD mini batch and it goes down, vs. the X{2}Y{2} a BAD minibatch (not good data; mislabeled examples) and the J goes up!

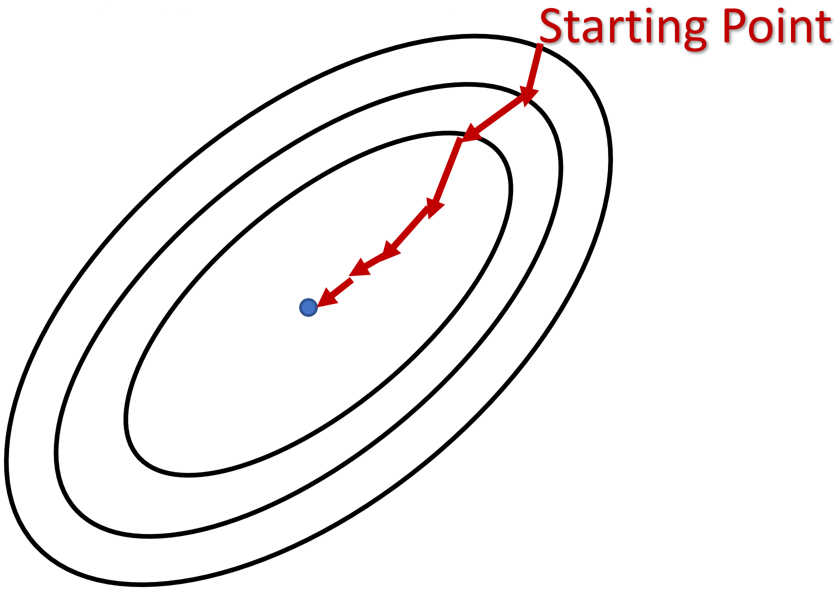
# Choosing the GD Mini-Batch Size

## We need to choose the SIZE of the mini-batch

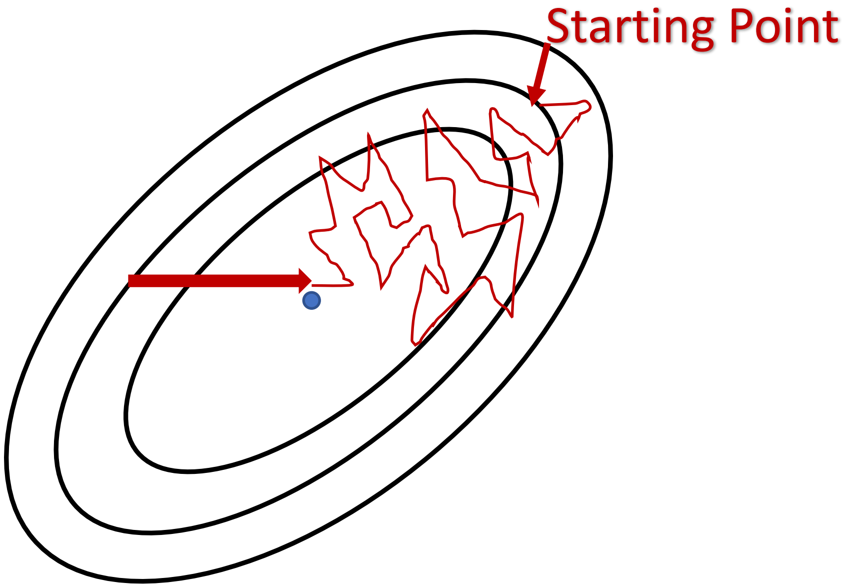
- If **Mini-batch Size == m**
  - We are using all examples (m)→ we end up with the **BATCH Gradient Descent algorithm**, which means we are using the entire training set (X, Y)
- If **Mini-batch Size == 1**
  - We end up using an algorithm called: **Stochastic Gradient Descent**
  - In SGD, every example **m=1**, is its own Mini-Batch
    - $X^{\{t\}}Y^{\{t\}} = (x^{(1)}, y^{(1)}); (x^{(2)}, y^{(2)})$ ……………… $(x^{(m)}, y^{(m)})$

# What those 2 extremes do when optimizing the Cost Function?

Batch GD
(all examples)



Stochastic GD
(1 example)



**These are the CONTOURS of the Cost Function we are trying to minimize (min is at the CENTER of the contour)**

# IN PRACTICE - Comparison

- The Mini Batch Size is between 1 and m (max # of examples)
- 1 = TOO low and m = TOO high

❑ **Batch GD – BATCH SIZE=m:**

- If the training set is **not TOO** large, *then we can use m*, i.e. it will NOT take TOO long to train.
- If the training set is **TOO large**, then it will take too long so picking m is **not** a good idea

❑ **MINI – BATCH --- Ideal size: 1< Batch Size <m:**

- Not too big and not too small size

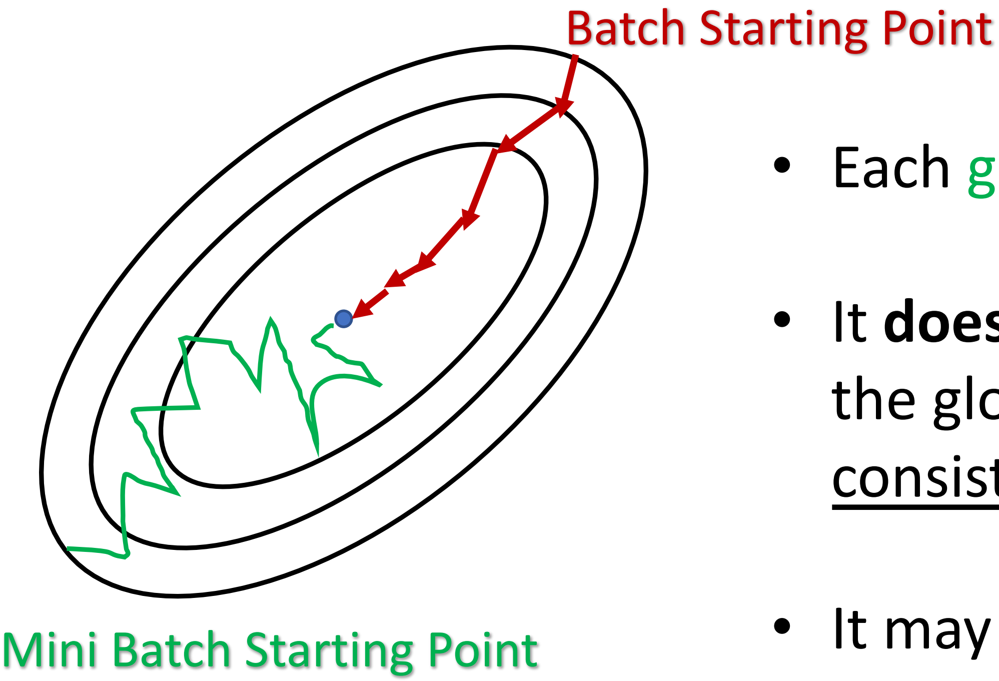
+ Offers the **fastest learning**: vectorization helps, e.g. processing 1000 examples at a time vs. 1 at a time

+ *No need to wait to process the entire training set to* ***make progress***

❑ **Stochastic GD – BATCH SIZE=1**:

- We can make a progress after even 1 example BUT its inefficient
- We lose almost ALL the speed up from vectorization

# IN PRACTICE - Comparison

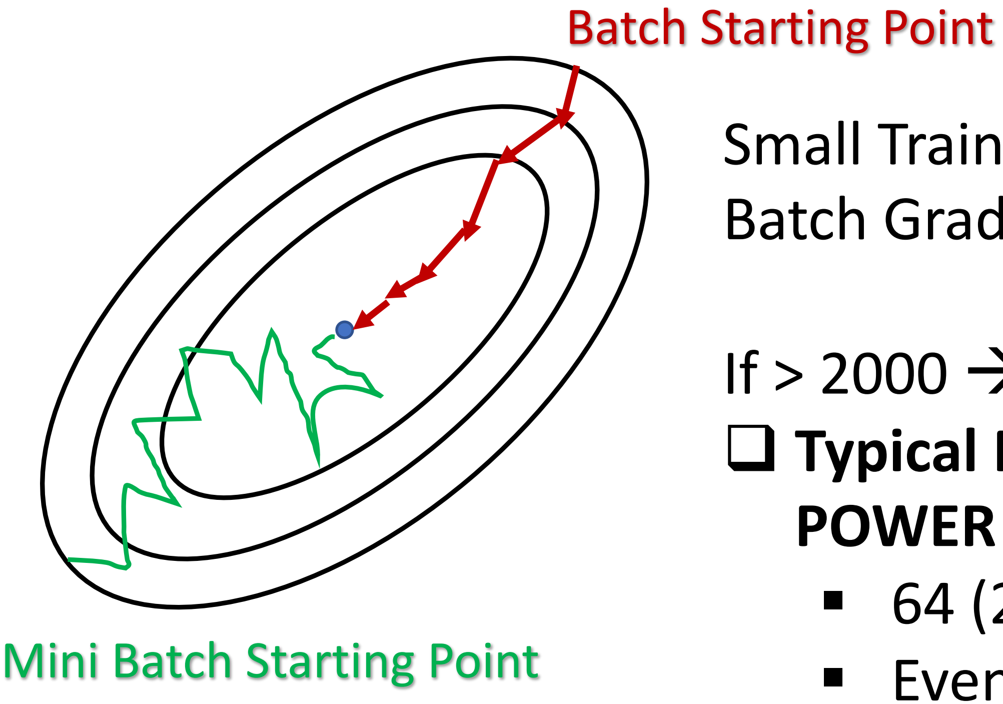


Batch Starting Point

Mini Batch Starting Point

- Each green step, is an actual iteration
- It **does not have to** always end up at the global minimum, but it does go consistently toward it
- It may oscillate close to the minimum for a while → in this case, we need to test smaller learning rates, by reducing the learning rate slowly
- Question: so **HOW** do we **really choose** the **actual mini batch size**

# Choosing the Mini Batch Size - Guidelines



Small Training Set (<2000 samples) → use Batch Gradient Descent

If > 2000 → use Mini Batch GD

❑ **Typical Mini Batch Sizes can better be POWER of 2**

- 64 ($2^6$), 128 ($2^7$), 256 ($2^7$),….
- Even though we stated that we can use a 1000 mini batch size, **ideally**, we should better use a **1024 size**
- **64 – 512 a more common mini batch size range**
- **Make sure the mini batch size fits the CPU/GPU memory**

# Questions?

THANK YOU!

More resources: https://towardsdatascience.com/an-introduction-to-gradient-descent-c9cca5739307