# Multivariate Calculus for Machine Learning

Francesco Pugliese, PhD

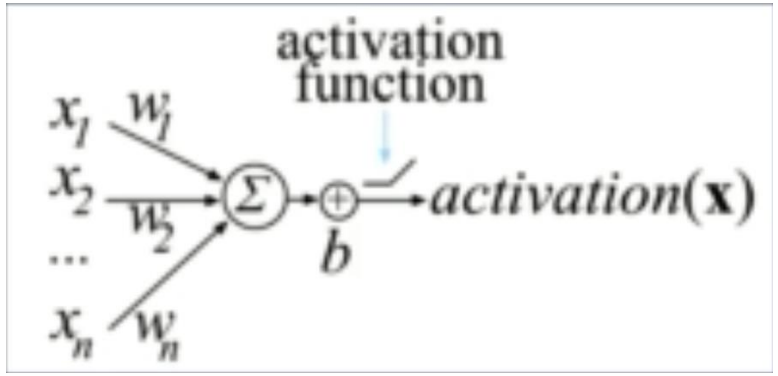
neural1977@gmail.com

# Table of contents

# Table of contents(cont..)

# Why calculus ?

- **Typical artificial neuron** –




- x1,x2….xn are **inputs**, w1,w2….wn are **weights** and b is the **bias** and applying an activation function we get an **output**. So this is what goes in an artificial neural network (no matter how complex network might be structured)
- To optimize a neural network, we have to update the weights w1, w2… wn after each iteration to **improve** the performance and **reduce** the loss of the network and that is accomplished using **gradient descent**.
- Gradients are calculated using multi variate calculus which has derivatives, gradients, integrals etc.., so in other words we need calculus to improve our neural networks

# Derivative

- Definition of a derivative:
  - Geometry: The derivative is the slope of the curve
  - Physical: The derivative is the rate of change.
- In other words, the derivative is rate of change of any given function so as to understand how fast or in what direction and by how much an equation is changing
- The derivative operator is written as d/dx

$$\text{If } y = f(x), \text{ then } \frac{dy}{dx} = \frac{df(x)}{dx} = \frac{d}{dx} f(x)$$

# Scalar derivative rules

- **Constant**: Derivative of a constant is 0.

  **Example**: $\frac{d}{dx}$ 6 = 0

- **Power rule**: $\frac{d}{dx}\, x^n$ = n$x^{n-1}$

  **Example**: $\frac{d}{dx}\, x^3$ = 3$x^2$

- **Multiplication**: $\frac{d}{dx}$ c$x^n$ = cn$x^{n-1}$

  **Example**: $\frac{d}{dx}$ 4$x^3$ = 12$x^2$

- **Sum rule**: $\frac{d}{dx}$( f(x) + g(x) ) = $\frac{d}{dx}$ f(x) + $\frac{d}{dx}$ g(x)

  **Example**: $\frac{d}{dx}$( 4x + 2$x^2$) = $4\frac{d}{dx}\, x + 2\frac{d}{dx} x^2$ = 4 + 4x

- **Product rule**: $\frac{d}{dx}$( f(x) g(x) ) = g(x)$\frac{d}{dx}$ f(x) + f($x$) $\frac{d}{dx}$ g(x)

  **Example**: $\frac{d}{dx}$( x$x^2$) = $x^2 \frac{d}{dx}\, x + x\frac{d}{dx} x^2$ = $x^2$(1) + x(2x) = $x^2$ + 2 $x^2$ = 3 $x^2$

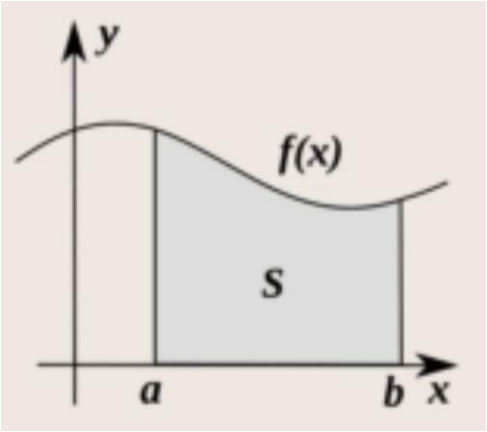
# Chain rule and Partial derivatives

- **Chain rule:** $\frac{d}{dx}$( f(g(x)) ) = $f'\big(g(x)\big)\, g'(x)$
  - **Example**: $\frac{d}{dx}$ sin($x^2$) = cos($x^2$) * 2x = 2x cos($x^2$)
- This is very important in Machine learning because when you have **multiple** layers in your artificial neural network, you have to calculate the derivative of **earlier or previous** layers
- **Partial derivatives**: Used for equations with more than one variable
  - **Example**: f(x,y) = 3$x^2$y, $\frac{d}{dx}$ f(x,y) = $\frac{d}{dx}$ 3$x^2$y = 3y $\frac{d}{dx}$ $x^2$ = 3y(2x) = 6xy

    $\frac{d}{dy}$ f(x,y) = $\frac{d}{dy}$ 3$x^2$y = 3$x^2$ $\frac{d}{dy}$ y = 3$x^2$(1) = 3$x^2$

# Integration

- Integration is used to find the **area under the curve** (displacement)




- Integration is the reverse of differentiation(derivatives), as a result, the integral is often called the anti-derivative

# Types of Integrals

- **Indefinite**:
  - Integrals with no limits
  - ∫ f(x) dx
- **Definite**:
  - Integrals with limits
  - $\int_a^b f(x)dx$
- Integration **reverses** differentiation:
  - ∫ $f'$(x) dx = f(x) + C
  - Here, $f'$(x) is the derivative of *f(*x) and C is the constant of integration

# Rules of integration

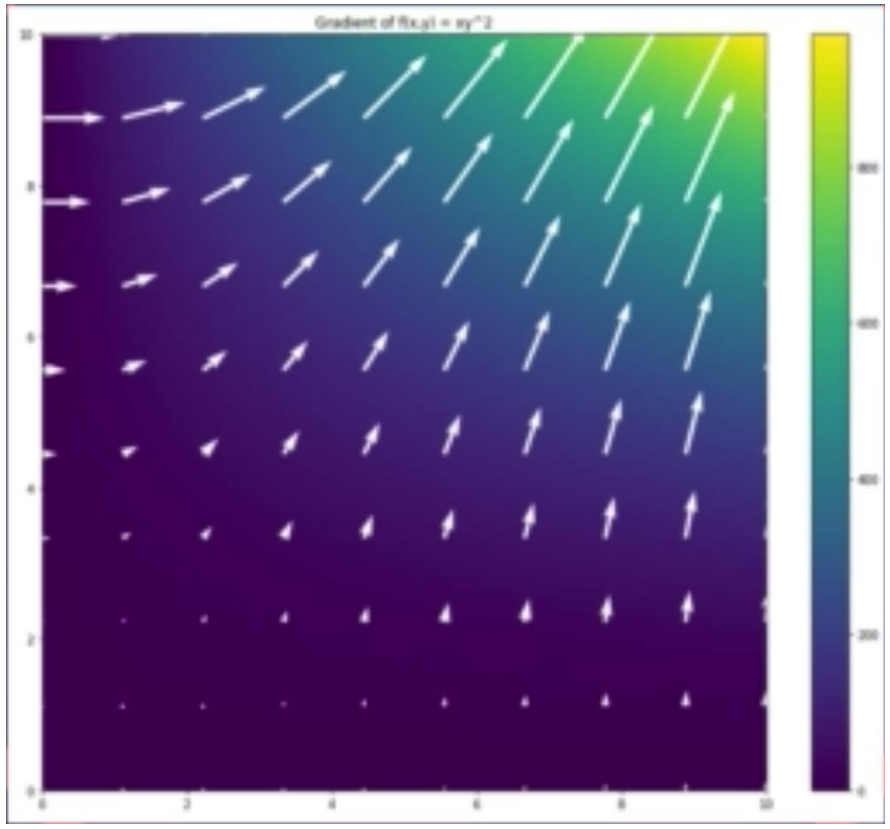
- **Power rule:** $\int x^n$ dx = $\frac{x^{n+1}}{n+1}$ + C
  - **Example**: $\int x^2$ dx = $\frac{x^3}{3}$ + C
- **Constants:** $\int$ kdx = kx + C
  - Example: $\int$ 4dx = 4x + C
- **Evaluating definite integrals** (Area under the curve):
  - $\int_a^b f'(x)dx$ = f(x) + C
  - ( value of f(x) + C at x=b ) - ( value of f(x) + C at x=a )
  - f(b) – f(a)
  - Example: $\int_0^2 3x^2dx$ = 3 $\int_0^2 x^2dx$ = 3 $(\frac{x^3}{3})$ | $\frac{2}{0}$ = $2^3$ - $0^3$ = 8

# Gradients aka Vector calculus

- Gradients are **pivotal** to machine learning. In machine learning the application of gradients is referred to as **convex** optimization. It gives a mechanism to update the weights of a neural network to minimize the loss function.
- Gradient is nothing but partial derivatives organized in to a vector – [ $\frac{d}{dx}$ , $\frac{d}{dy}$ ]
- In other words, Gradient is nothing but partial derivative of every single variable in the equation of interest.
- The gradient for f(x,y) -
  - $\nabla$ f(x,y) = [ $\frac{df(x,y)}{dx}$ , $\frac{df(x,y)}{dy}$ ]
  - **Example**: f(x,y) = $3x^2$y, $\nabla$ $3x^2$y = [ $\frac{\mathrm{d}3x^2\mathrm{y}}{dx}$ , $\frac{\mathrm{d}3x^2\mathrm{y}}{dy}$ ] = [6xy, $3x^2$]
- Like the derivative, the gradient represents the slope of a function or rate of change.
- The gradient points in the direction of the **greatest** rate of increase of function, and its **magnitude** is the **slope** in that direction. This is invaluable because using gradient we can determine the direction of change for our variable that has **greatest** impact
- In the concept of Machine learning we are actually going in negative gradient direction.

# Graphical representation of gradients

Gradient representation for, f(x,y) = $xy^2$



# Gradient vectors

- Gradients are not restricted to two dimensions, they can easily be expanded,
  - $\nabla$ f(x,y,z) = [ $\frac{df(x,y,z)}{dx}$ , $\frac{df(x,y,z)}{dy}$ , $\frac{df(x,y,z)}{dz}$]
- In machine learning we rarely stop at 3 dimensions and we rarely have one equation ie.., we have more complex equations and more dimensions.
- Gradient vectors organize the partial derivatives for a scalar function. If we have **multiple functions**, we use the Jacobian to represent **vector** of gradients.
- **Jacobian matrix**, J = $\begin{bmatrix} \nabla \text{ f(x,y)} \\ \nabla \text{ g(x,y)} \end{bmatrix}$ = $\begin{bmatrix} \frac{df(x,y)}{dx} & \frac{df(x,y)}{dy} \\ \frac{dg(x,y)}{dx} & \frac{dg(x,y)}{dy} \end{bmatrix}$ Here, we have 2 functions f, g with 2 variables each
- In machine learning we are often working in high dimensional spaces, we need to use these derivatives and gradients in high dimensional spaces to understand how we can update the neural network weights so as to get a better correlation or accuracy for a problem at hand.
- So in a nutshell, Gradients are vector of partial derivatives and Jacobian is a vector of gradients.

# Gradients for machine learning

Gradient and Jacobian matrix for a function with n variables x1,x2….xn

$$f(x, y, z) \rightarrow f(\mathbf{x}) \text{ where } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

And Jacobian matrix for m such functions f1, f2….fn with n variables each is given by,

$$\mathbf{J} = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \cdots \\ \nabla f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{d}{dx_1} f_1(\mathbf{x}) & \frac{d}{dx_2} f_1(\mathbf{x}) & \cdots & \frac{d}{dx_n} f_1(\mathbf{x}) \\ \frac{d}{dx_1} f_2(\mathbf{x}) & \frac{d}{dx_2} f_2(\mathbf{x}) & \cdots & \frac{d}{dx_n} f_2(\mathbf{x}) \\ \cdots & \cdots & \cdots & \cdots \\ \frac{d}{dx_1} f_m(\mathbf{x}) & \frac{d}{dx_2} f_m(\mathbf{x}) & \cdots & \frac{d}{dx_n} f_m(\mathbf{x}) \end{bmatrix}$$

This is how we calculate gradients for larger array of equations and this is important in Machine learning where we often work in high dimensional spaces, we use these gradients to minimize our objective function.

# Numerical computation

- Machine learning algorithms usually require a **high** amount of numerical **computation**.
- This typically refers to algorithms that solve mathematical problems by methods that update **estimates** of the solution via an **iterative** process.
- Common operations include **optimization** (finding the value of an **argument** that minimizes or maximizes a function) and solving systems of linear equations.
- Even just evaluating a mathematical function on a digital computer can be difficult when the function involves real numbers, which **cannot** be represented precisely using a **finite** amount of memory

# Overflow and Underflow

- The fundamental difficulty in performing continuous math on a digital computer is that we need to **represent** infinitely many real numbers with a finite number of **bit** patterns.
- This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer.
- In many cases, this is just rounding error.
- Rounding error is problematic, especially when it compounds across many operations, and can cause algorithms that work in theory to fail in practice if they are not designed to minimize the accumulation of rounding error.
- One form of rounding error that is particularly devastating is underflow.

# Overflow and Underflow

- Underflow occurs when numbers near zero are rounded to zero.
- Many functions behave qualitatively differently when their argument is zero rather than a small positive number.
- For example, we usually want to avoid division by zero (some software environments will raise exceptions when this occurs, others will return a result with a placeholder not-a-number value) or taking the logarithm of zero (this is usually treated as $-\infty$, which then becomes not-a-number if it is used for many further arithmetic operations).
- Another highly damaging form of numerical error is overflow.
- Overflow occurs when numbers with large magnitude are approximated as $\infty$or $-\infty$.
- Further arithmetic will usually change these infinite values into not-a-number values.
- One example of a function that must be stabilized against underflow and overflow is the softmax function.

# Optimization

- Optimization is essential for artificial intelligence and neural networks specifically deep neural networks
- This is the process through which we will make the machine learning algorithms **learn**.
- This is how we are going to update the parameters in our neural network so as to minimize the loss function and improve the performance of our machine learning model.
- We will be using derivates and gradients that we discussed to do the optimization.
- The function we want to minimize or maximize is called the **objective function**, or criterion. When we are minimizing it, we may also call it the **cost** function, **loss** function, or **error** function. All these names can be used interchangeably.

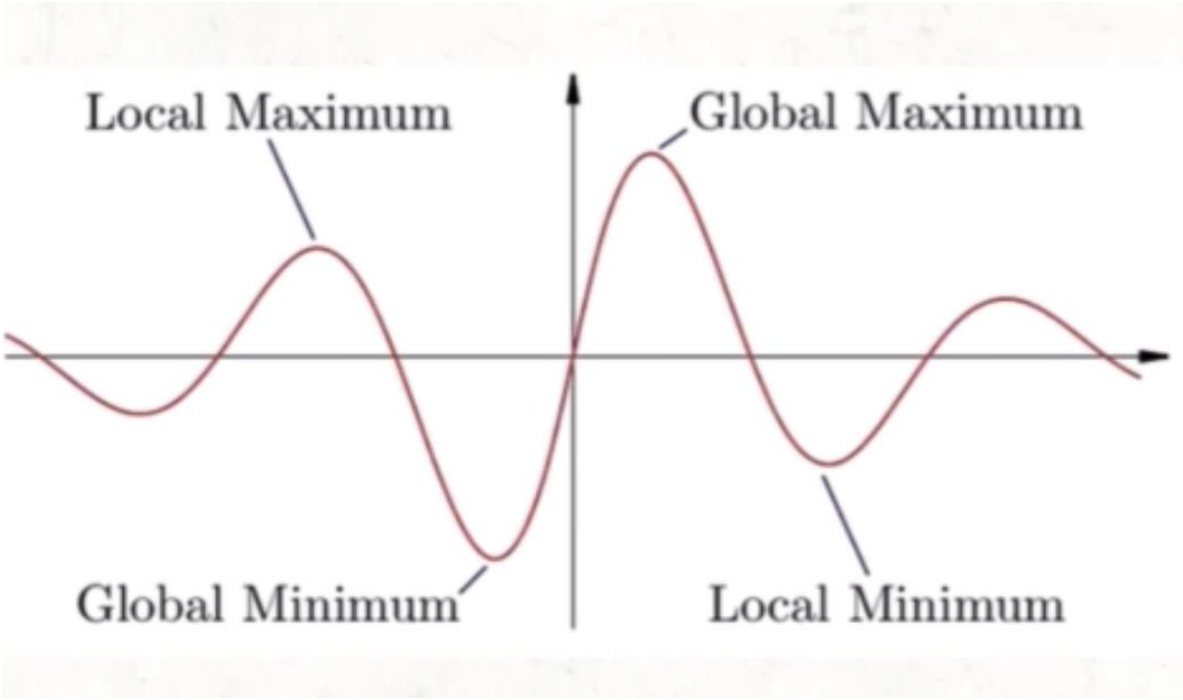
# The Optimization problem

With optimization, we minimize a function for instance, F(x) with respective to certain constraints.

Optimization seeks to **find** the global minimum of an objective function, subject to constraints.

Slope at local minimum/maximum and global minimum/maximum is **0**

Local minimum is important because when we are optimizing our algorithms we can often get stuck in local minimum even through there are further minimum point in the graph or data distribution, we will talk about how to mitigate this situation.




Local maximum/minimum – maximum/minimum value in their local area

Global maximum/minimum – Absolute smallest or largest point in the graph

# Why Optimization is important ?

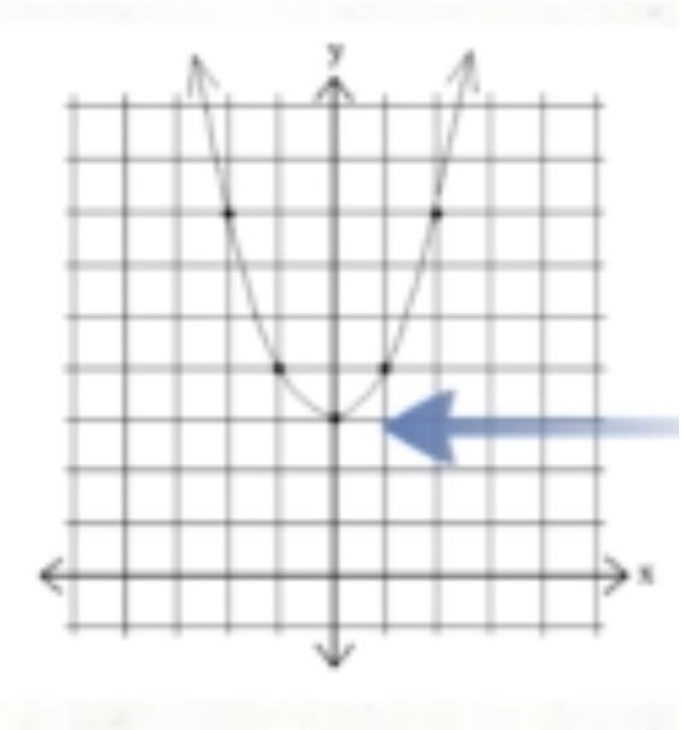
In every machine learning algorithm we have a function that we want to minimize, as mentioned below:

| | |
|---|---|
| Classification | $\underset{w}{\text{minimize}} \sum_{i=1}^{n} \log\left(1+\exp(-y_i x_i^T w)\right)$ |
| K-Means | $\underset{\mu_1,\ldots,\mu_k}{\text{minimize}} \; J(\mu) = \sum_{j=1}^{k} \sum_{i \in C_j} \|x_i - \mu_j\|^2$ |
| Logistic Regression | $\underset{w}{\text{minimize}} \; \|Xw - y\|^2$ |

So its important to understand the fundamental math behind all these optimizations

# Example problem

$Y = x^2 + 3$




Derivative is 0 at local min/max and global min/max

# Derivative tests

- Calculus can be used to find the extrema of an objective function: uses first and second derivates.
- **First derivative test**:

  Y = $x^2$ + 3

  $Y' = 2x\ (\frac{dY}{dx})$

  0 = 2x (Set derivative to 0 since the slope is 0 at global maximum and minimum)

  x = 0, this means that we have a global minimum at x = 0

- **Second derivative test**: (take the derivative again)

  $Y' = 2x$

  $Y'' = 2$

  **Note**: if second derivative is positive then its minimum

  if second derivative is negative then its maximum

# Optimization for Machine learning

- In the context of Machine learning, optimization is an iterative process i.e.., initially we are going to make a **guess** about the minimum and take the **gradients** and we are going to use those gradients to **update** our parameters to make a slightly **better** guess.
- We are going to do this over and over again until our guess no longer improves our results.
- The gradient tells us which way to change our parameters since in general gradient tells us the direction of maximum increase
- So what we need in the context of Machine learning is direction of maximum decrease which is nothing but negative gradient.

# Optimization for Machine learning

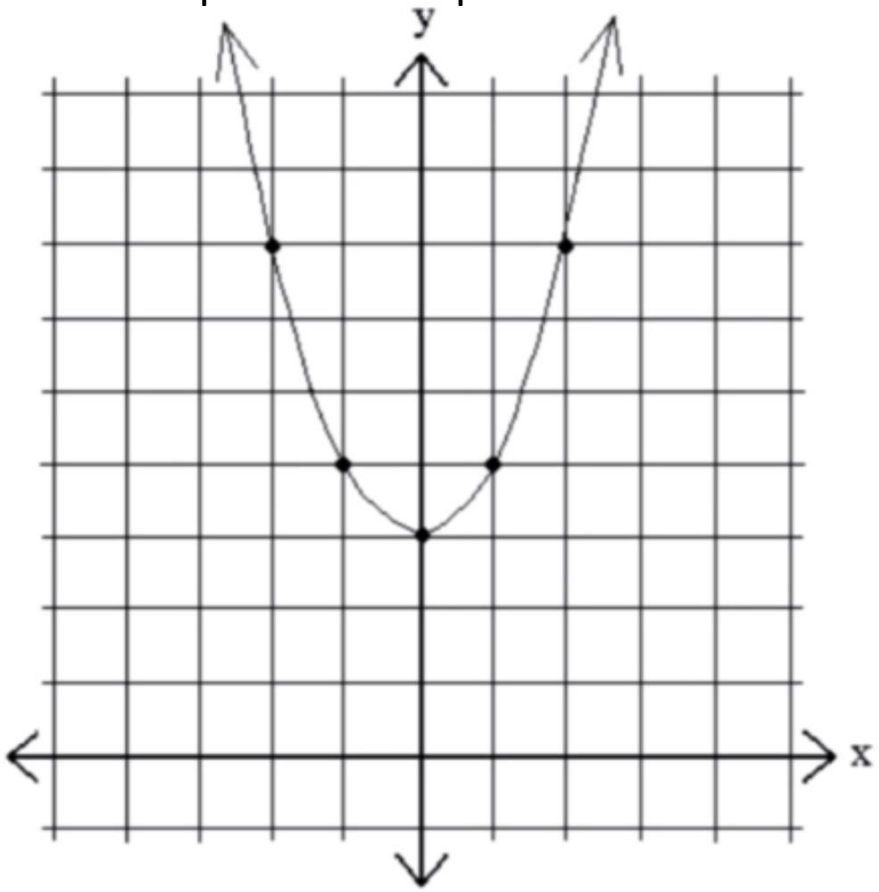
Here we will see how we use gradients to optimize a simple function with one variable, Y = $x^2$ + 3

1st Guess -
x = 3 is the minimum

Evaluate f(x) -
$y = (3)^2 + 3 = 12$

Evaluate derivative -
$y' = 2(3) = 6$

Update guess -
x = -1 is the minimum




2nd Guess -
x = -1 is the minimum

Evaluate f(x) -
$y = (-1)^2 + 3 = 4$

Evaluate derivative -
$y' = 2(-1) = -2$

Update guess -
x = 0 is the minimum

# Gradient descent

- Gradient descent is a way to minimize an objective function J(θ),
  - $\theta \in R^d$ : **model** parameters
  - η: **learning** rate
  - $\nabla_\theta J(\theta)$: **gradient** of the objective function with regard to the parameters
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$
- Updates parameters in opposite direction of gradient.
- Gradient descent variants:
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini batch gradient descent
- All the above 3 variants differ on the amount of data used for update

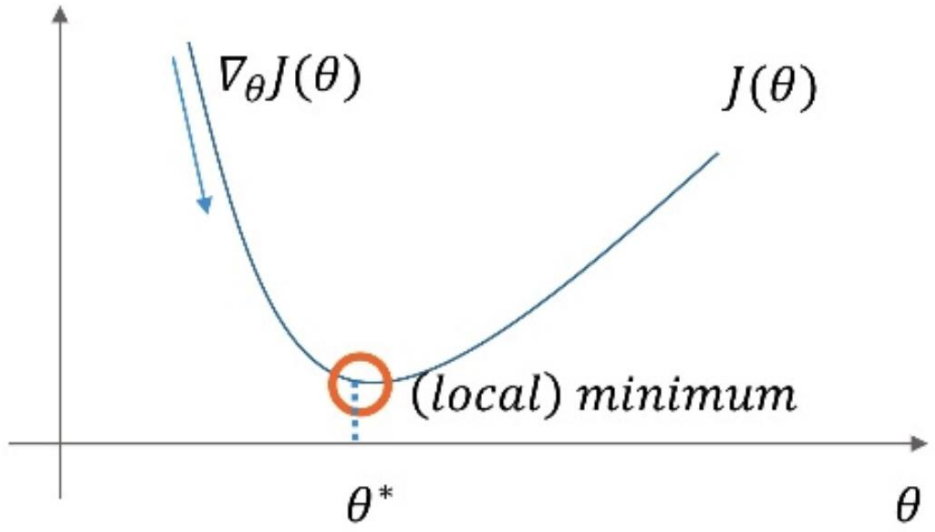
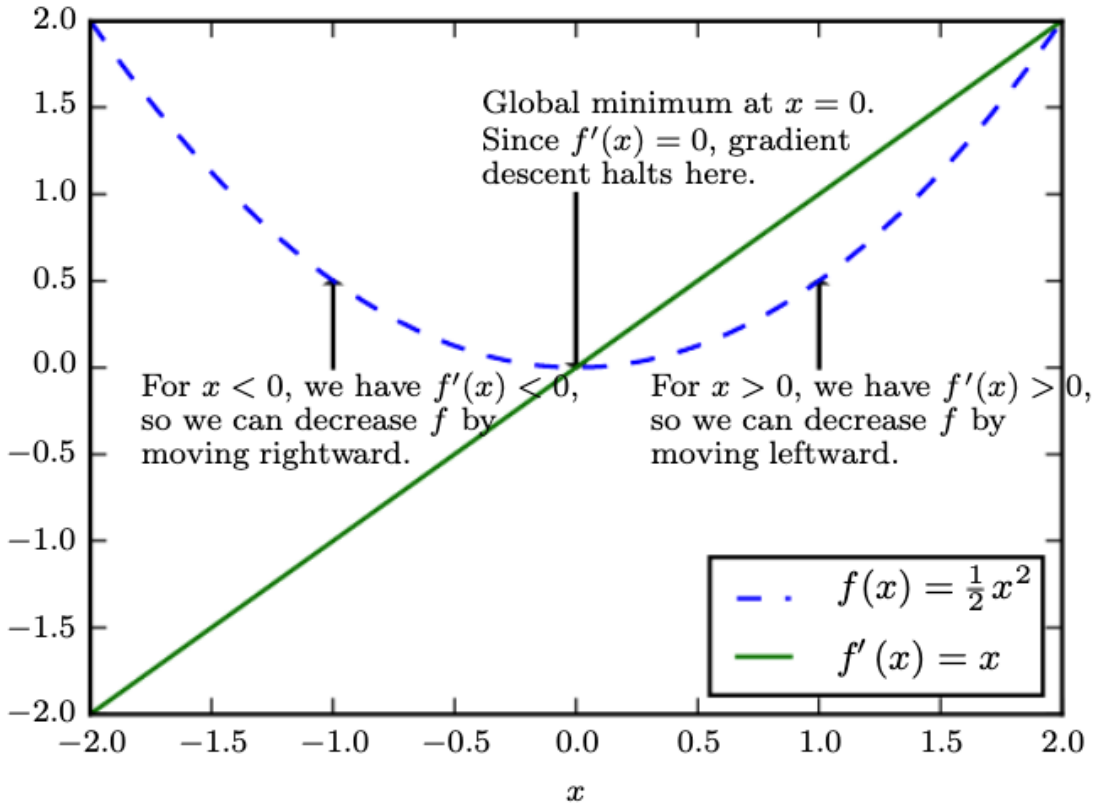



Fig: Optimization with gradient descent

# Gradient descent




An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

# Comparison of Gradient descent algorithms

| Method | Accuracy | Update Speed | Memory Usage | Online Learning |
|---|---|---|---|---|
| **Batch** gradient descent | Good | Slow | High | No |
| **Stochastic** gradient descent | Good (with annealing) | High | Low | Yes |
| **Mini-batch** gradient descent | Good | Medium | Medium | Yes |

Fig: Comparison of trade-off's of Gradient descent algorithms

# Gradient descent algorithms for Machine learning

- Some common variants of Gradient descent for Machine learning are as follows,
  - Stochastic gradient descent
  - SGD with Momentum
  - AdaGrad
  - RMSprop
  - Adadelta
  - NAG
  - Adam
  - AdaMax
  - Nadam
  - AMSGrad
- Gradient descent is one of the most **popular** algorithms to perform **optimization** and by far the most common way to optimize neural networks.
- At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent.
- These algorithms, however, are often used as **black-box** optimizers.

Fig: Optimization with gradient descent

# References and further reading


- https://www.slideshare.net/SebastianRuder/optimization-for-deep-learning
- https://ruder.io/optimizing-gradient-descent/
- http://www.deeplearningbook.org/contents/numerical.html
- https://www.udemy.com/course/mathematical-foundation-for-machine-learning-and-ai

Francesco **Pugliese**