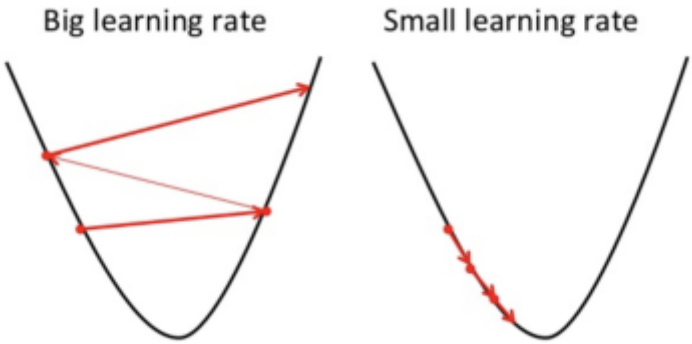
Linear Regression: Gradient Descent - Automating the Cost Function

Gradient Descent is an algorithm used to find the optimal or minimized $J(\theta_0, \theta_1)$. Sometimes it is referred to as *Batch Gradient Descent* because it is iterating over all training examples.

> $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$

> **Note:** The function uses an *assignment* operator $:=$ instead of an *equality* operator $=$ which means that θ_j is not *equal* to the right hand side of the equation but *set* to the result of the equation.

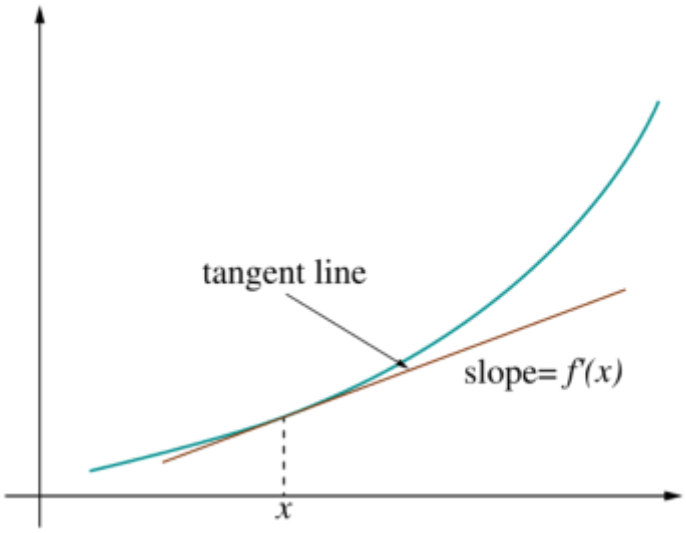
Here, α (alpha) is the *learning rate* or *step*. The smaller the step the slower the algorithm will run. The larger, the faster. However, if alpha is too large you may not get to the lowest theta values since the algorithm may overshoot the minimum.



The derivative expression, $\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$, measures the rate of change. The *derivative* just measures the *slope* of a line (rise over run or y over x) that is *tangent* to or next to a point within of a graphed function. Here, ∂ (partial derivative) is a just a mathematical term which means that it the function works with multiple variables contrasted to d (derivative) which is used with single variable functions.

Moving Toward Minimum

The derivative can be sloped in a positive direction on the x, y axis (going from lower left to upper right) which denotes a *positive derivative* as shown in the illustration below. This derivative (positive, negative, or zero) tells the Gradient Descent algorithm which way to move (step next) with respect to the graphed function (shown as the green line in the graph below) in order to seek the minimum of $J(\theta_0, \theta_1)$.



If the result of the derivative, $\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1)$, is a *positive number* then the equation looks like the below which moves the Gradient Decent algorithm to the *left* on the graph moving it closer to the minimum. Here θ_1 is on the x axis and the result would yield an x value to the left of the starting x value which moves closer to the minimum of our function $h_\theta(x)$.

> $\theta_1 := \theta_1 - \alpha(+\partial)$

As Gradient Descent approaches the minimum, the rate of change will become increasingly smaller. Hence, the algorithm will slow as it approaches the minimum and ultimately stop changing once it has reached the minimum point so there is no need to decrease the value of α.

Simultaneous Compute

What *Gradient Descent* algorithm does is *simultaneously* compute values for θ_0 and θ_1. What is meant by *simultaneously* is represented in the pseudo code below where θ_0 and θ_1 are assigned new values at the same time. In other words, if θ_0 was set ($\theta_0 :=$ *temp0*) *before* temp1 was set (*temp1* $:= \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_0,\theta_1)$) then it would affect the results of temp1 and yield incorrect results. We want to repeat this series of steps until we reach *convergence* or θ_0 and θ_1 are at their minimum.

> *repeat until convergence {*
>
> $temp0 := \theta_0 - \alpha\frac{\partial}{\partial\theta_0}J(\theta_0,\theta_1)$
>
> $temp1 := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_0,\theta_1)$
>
> $\theta_0 := temp0$
>
> $\theta_1 := temp1$
>
> *}*

Finding the partial derivatives of the expression $\alpha\frac{\partial}{\partial\theta_1}J(\theta_0,\theta_1)$ with respect to θ_0 and θ_1 yields the below where m is the number of examples in our training set and $\alpha\frac{1}{m}$ is the *learning rate*:

> *repeat until convergence {*
>
> $$temp0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\cdot x_0^{(i)}$$
>
> $$temp1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\cdot x_1^{(i)}$$

$$\theta_0 := temp0$$

$$\theta_1 := temp1$$

$$\}$$

Advanced Alternatives

There are more advanced alternatives to Gradient Descent which negate the need to pick a learning rate and often times faster than Gradient Descent. Below are some alternatives. In Python we would use *SciPy optimize* set of functions.

SciPy optimize provides functions for minimizing (or maximizing) objective functions, possibly subject to constraints. It includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programing, constrained and nonlinear least-squares, root finding and curve fitting.

Conjugate Gradent

The *Conjugate Gradient* algorithm is build on the *Truncated Newton* algorithm. In Python we can use the scipy.optimize.fmin_tnc function. Below is a Python example where we plug in our cost and gradient compute functions:

```
# Minimize a function with variables subject to bounds,
# using gradient information using truncated Newton or Newton Conjugate-
Gradient.
result = opt.fmin_tnc(
  # Initial guess.
  x0 = theta,
  # Objective function to be minimized.
  func = compute_cost,
  # Gradient of func.
  fprime = compute_gradient,
  # Extra arguments passed to f and fprime.
  args = ( X, y ))

theta_min = result[0]
iterations = result[1]
cost = compute_cost(theta_min, X, y)

print(f'Minimized theta values: {theta_min}')
print(f'Number of iterations: {iterations}')
print(f'Cost: {cost}')
```

BFGS

The *Broyden, Fletcher, Goldfarb, and Shanno* (BFGS) algorithm is based on the quasi-Netwon method. In Python we can use the scipy.optimize.fmin_bfgs function.

The BFGS method accumulates all gradients since the given initial guess. There is two problems with this method. First, the memory can increase indefinitely. Second, for nonlinear problems, the Hessian (theta matrix) at the initial guess is often not representative of the Hessian at the solution. The approximated Hessian will thus be biased until enough gradients are accumulated close to the solution. This can slow down convergence, but should still converge with good line search algorithms that have a single local minimum.

L-BFGS

The *Limited memory BFGS* (L-BFGS) algorithm is based on the quasi-Netwon method and approximates BFGS algorithm using a limited amount of memory. In Python we can use the scipy.optmize.l-bfgs-b function which is the same a L-BFGS but with bounds.

L-BFGS is the same as BFGS but with a limited-memory, which means that after some time, old gradients are discarded to leave more space for freshly computed gradients. This solves the problem of the memory, and it avoids the bias of the initial gradient. However, depending on the number of gradients kept in memory, the Hessian (theta matrix) might never be precisely estimated, and can be another source of bias. This can slow down convergence, but should still converge with good line search algorithms that have a single local minimum.

L-BFGS-B is the same as L-BFGS but with bound constraints on the input variables. L-BFGS-B will stop optimizing variables that are on the boundary of the domain.