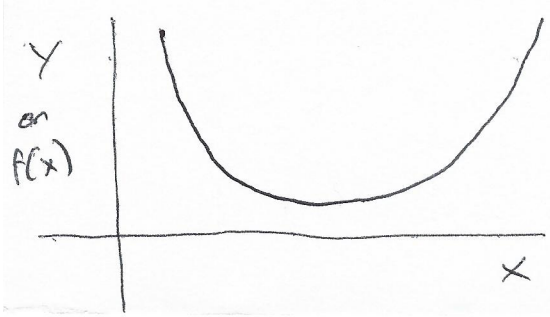
Cost function anatomy 101: Big picture categorization (1)

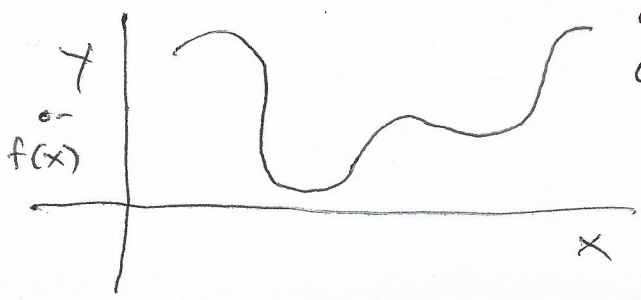
- The cost functions we looked at for examples of linear regression + classification were 3-d surfaces, but here we will examine 2-d cost functions (since they're easier to draw)!

- But no worries — everything discussed here generalizes easily to any dimension you want

- (cost) functions come in all shapes and sizes, but we can generally categorize them into two categories:

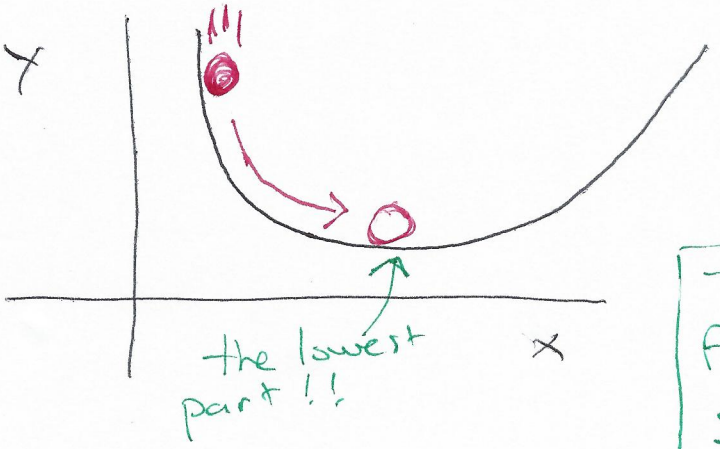


"convex"



"non convex"

- Convex functions look like bowls or half-pipes
  - If you rolled a marble down any side of a convex function it would eventually reach the lowest point

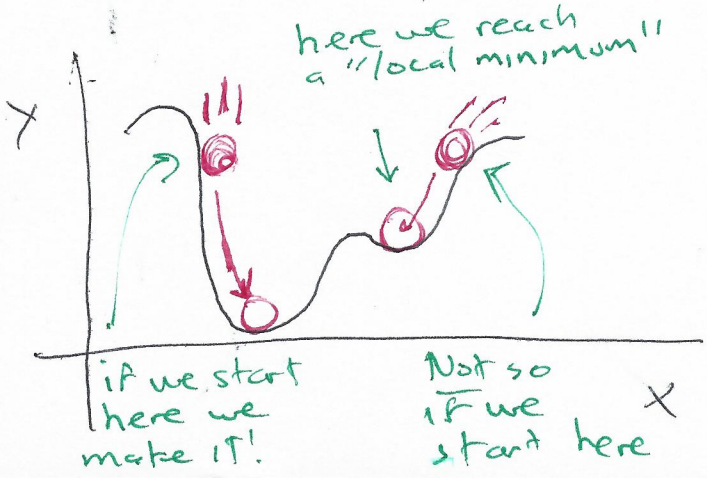


(as shown in our simulator over here)

The lowest point on a function is called it's global minimum

jargon time.

- A non-convex function is like a mountain range, it goes up and down a bunch, and has multiple hills and/or valleys

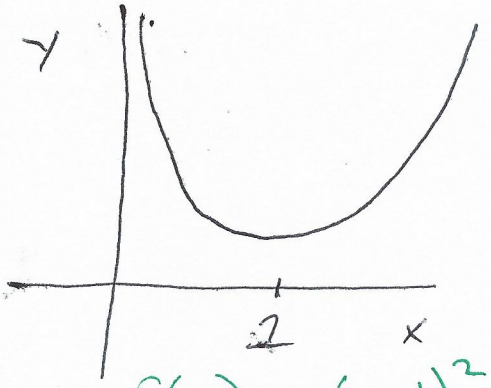


  - roll a marble down a side of a non-convex function and you might reach the global minimum - but you might not! Depends on where you start!

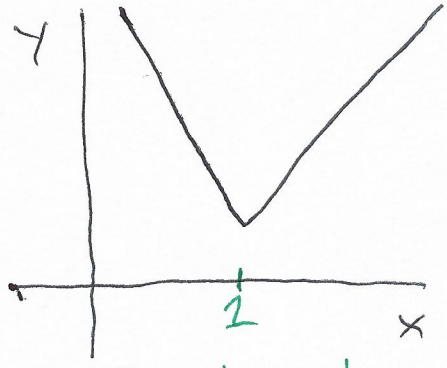
- Now don't go thinking that all convex functions look like the one I drew, they can be quite diverse looking. Examples

minima = many minimum

many global minima!



$f(x) = (x-1)^2$

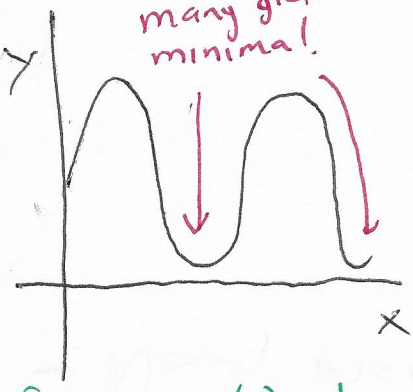


$f(x) = |x-1|$

$$f(x) = \begin{cases} -x+2 & x<1 \\ 1 & 1 \le x \le 3 \\ x-3 & x>3 \end{cases}$$

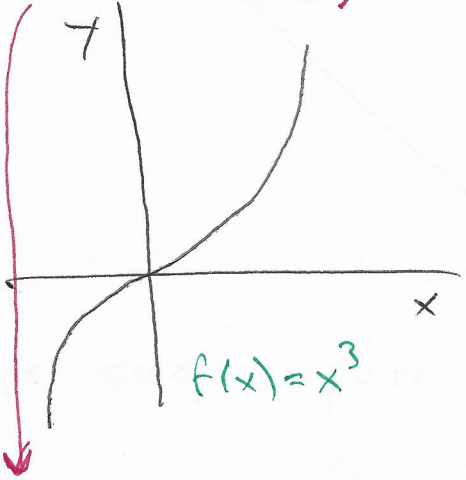
= Same goes for non-convex functions: they are even more diverse in shape

many global minima!



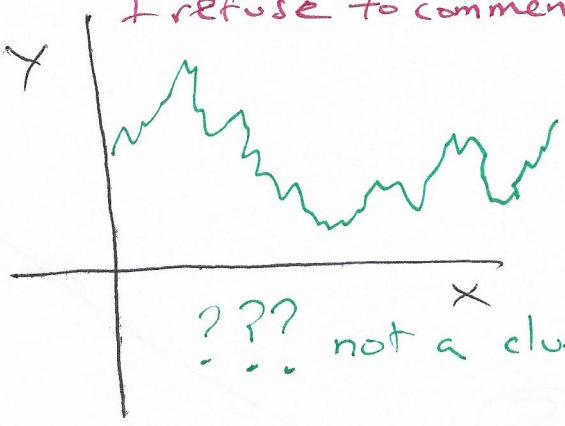
$f(x) = \sin(x)+1$

Where's the global minimum here?



$f(x) = x^3$

This is just crazy. I refuse to comment.

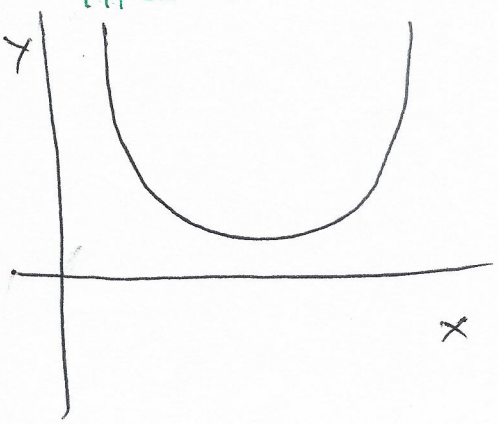


??? not a clue

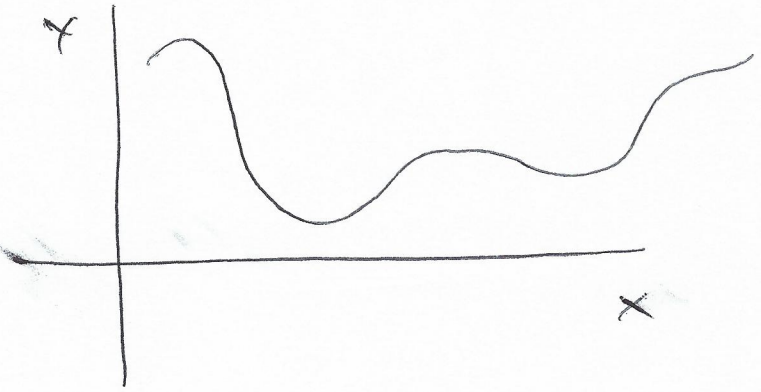
- However for machine learning we generally only see smoothly curving cost functions ← not always but very often with a fixed lower bound (i.e. where it's global minima are not $-\infty$)

- So from our lineup, we typically see

Convex function like this



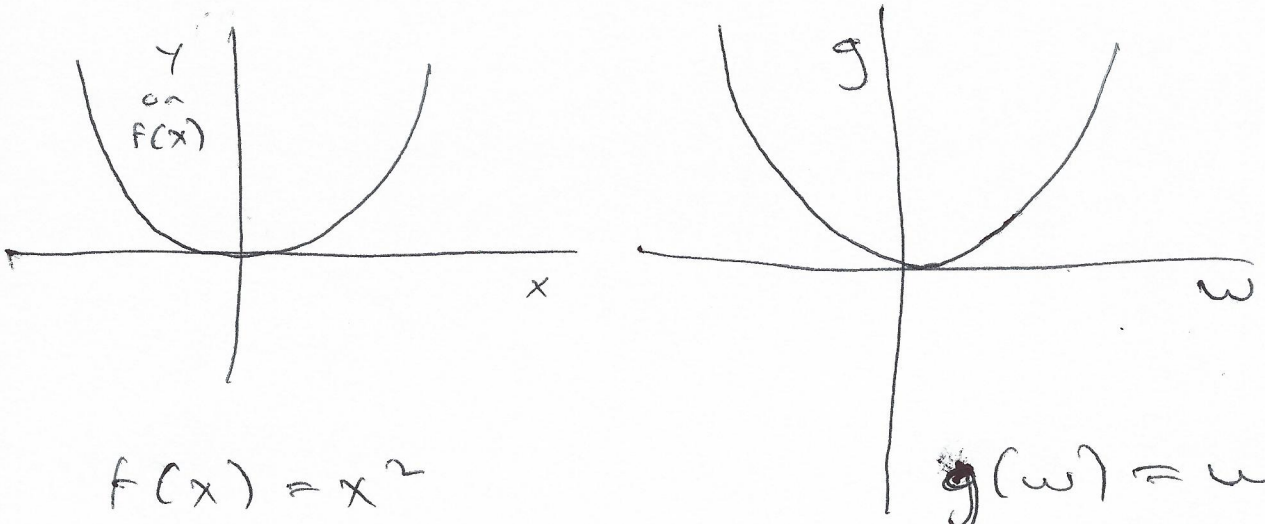
Non-convex function like this



Cost function anatomy 102: the derivative and linear approximation

- We're going to make a slight change of notation: since we are talking about cost functions of machine learning models that reflect a parameter or weight tuning, we will use w as our input from now on ("w" is short for "weight")

e.g.



$f(x) = x^2$

$g(w) = w^2$

↖ this says $g(w) = w^2$

Same thing, different notation

$x \longmapsto w$

$f \longmapsto g$

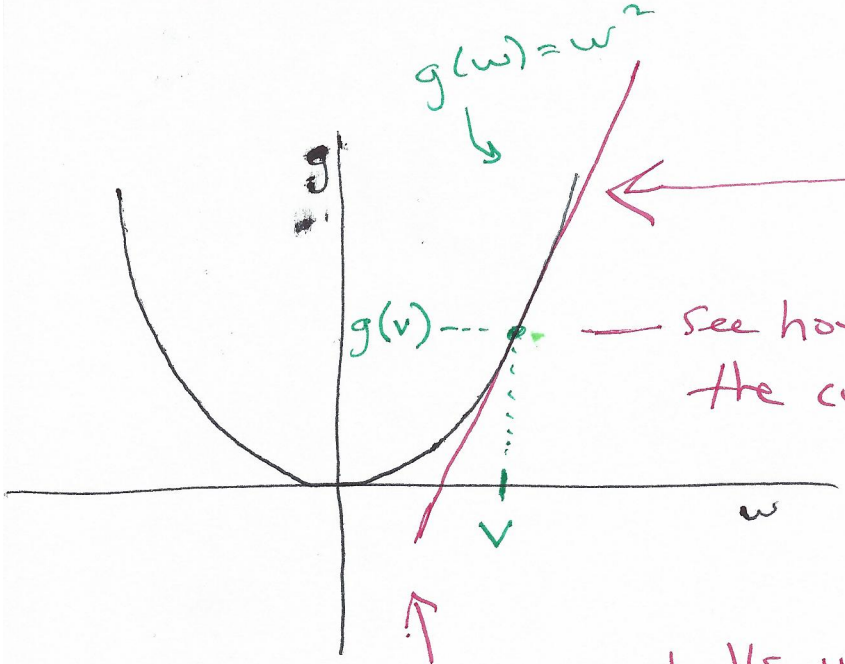
- Great, now with that out of the way we can talk about

The derivative

- What is it? The derivative of a ~~function~~ function $g(w)$ at a point v — denoted $g'(v)$ — (that's a "prime") (you say this "g prime v") is the slope of the line tangent to g at v

..... wha? Picture time.

Say this is g, $g'(v)$ is the slope of this line

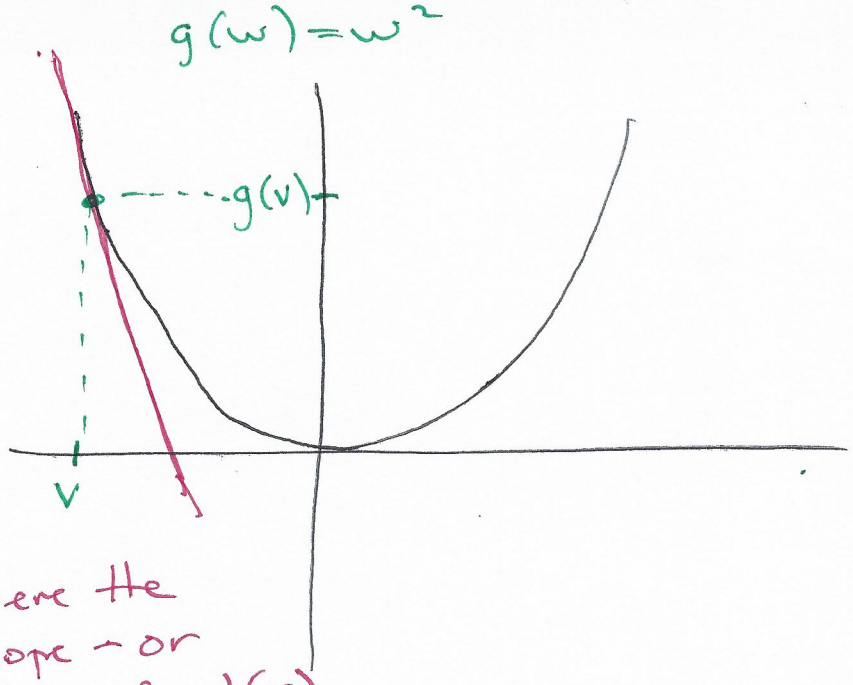


— see how that line is just touching the curve? That's how you know it's a tangent line! If it just touches.

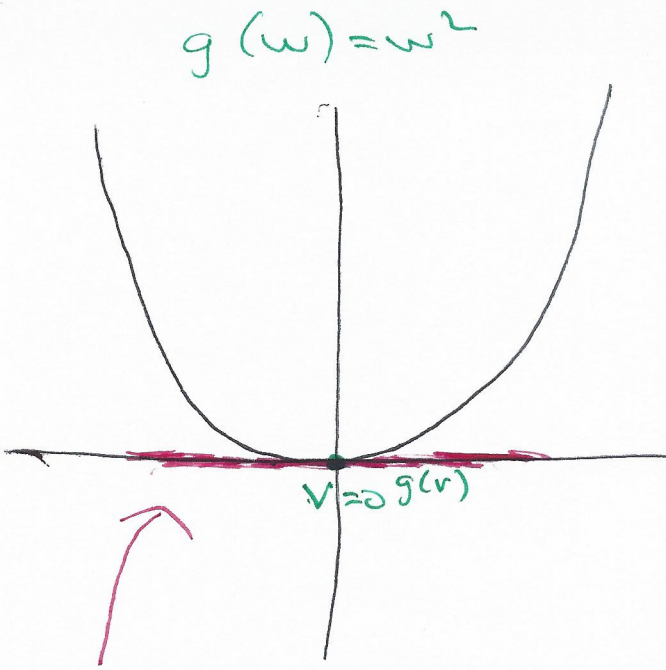
Using our eye-balls we can see the slope — or value of $g'(v)$ — is positive

That is $g'(v) > 0$

How about if v is here?



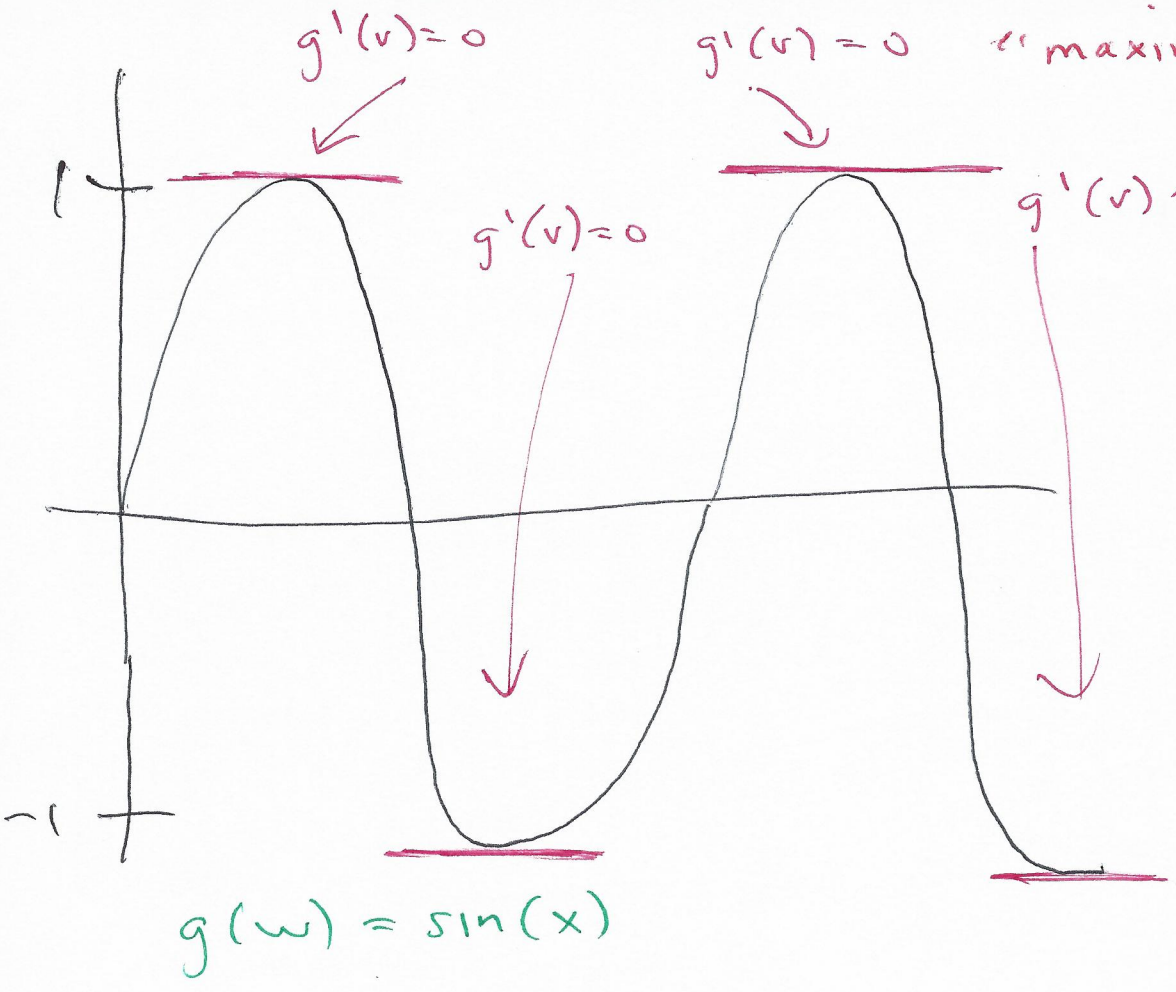
Here the slope - or value of g'(v) - is negative → so g'(v) < 0



Here g'(v) = 0

- Notice: g'(v) = 0 only when v was the global minimum of the function
  - Does that generalize? Is it true
    - for convex functions ✓✓✓ YOU BET!!
    - for non-convex functions ✗ NOPE

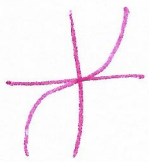
How come? Cause this....



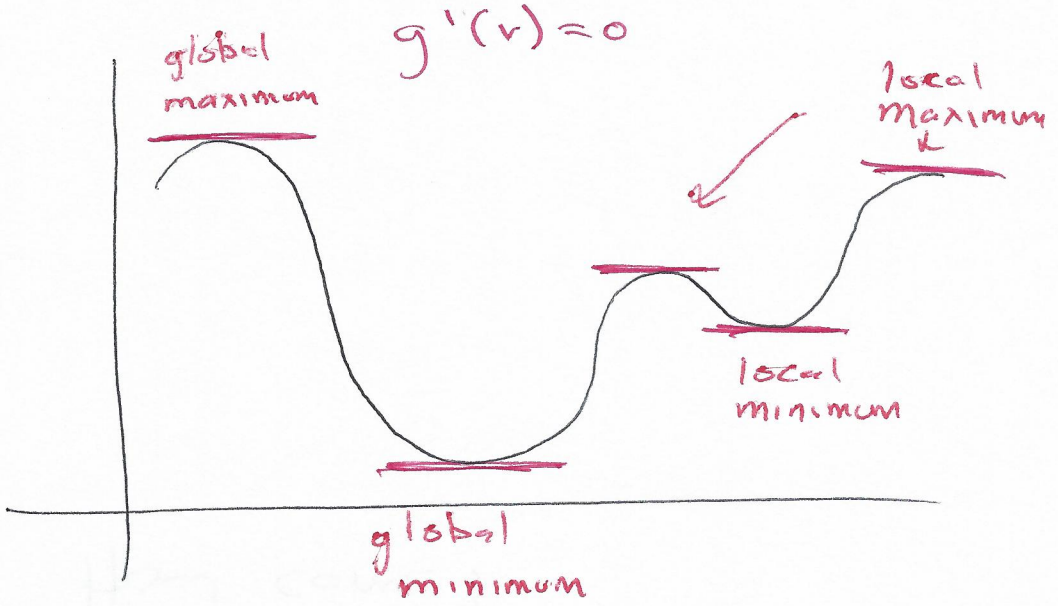
Aside

There are some other points to that satisfy $g'(v)=0$ that are not maxima or minima. These are called "saddle points"

E.g. $g(w)=w^3$ at $v=0$



and this



Please!

Make it stop!

So many $g'(v)=0$ that aren't global minima!!

Oh the humanity!;

- Look, chill out, its cool. Even though it is not the case for non-convex function that $g'(v) = 0$ only when v is a global minimum, this condition or test

$$g'(v) = 0$$

is extremely useful. Using this we can design an algorithm that can practically find the global minima of a generic convex or non-convex cost function

Jargon time
Any point satisfying this is called a "critical" or "stationary" point

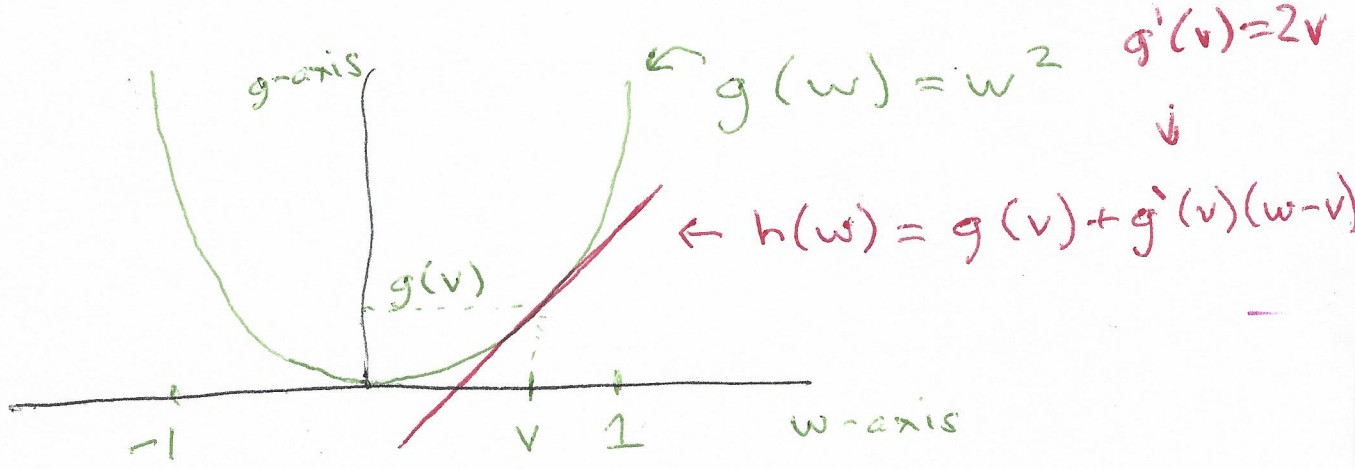
Reminder: we want to find the ~~g~~ global minima of machine learning cost functions since they correspond to best ~~fit~~ fitting regression / classification schemes.

To Do:

Write a short function in Python that produces the following items:

- A plot of the cost function $g(w) = w^2$ over the region of input $[-1, 1]$
- A plot of the linear approximation to g ~~for~~ at a point v where v can be any point in $(-1, 1)$

The following picture - minus the equations - is what you're after



- Why is $h(w) = g(v) + g'(v)(w-v)$ the equation of the tangent line at v?
- Drawing pictures is a great way to understand a concept → do whenever possible!!

To do:

(10.5)

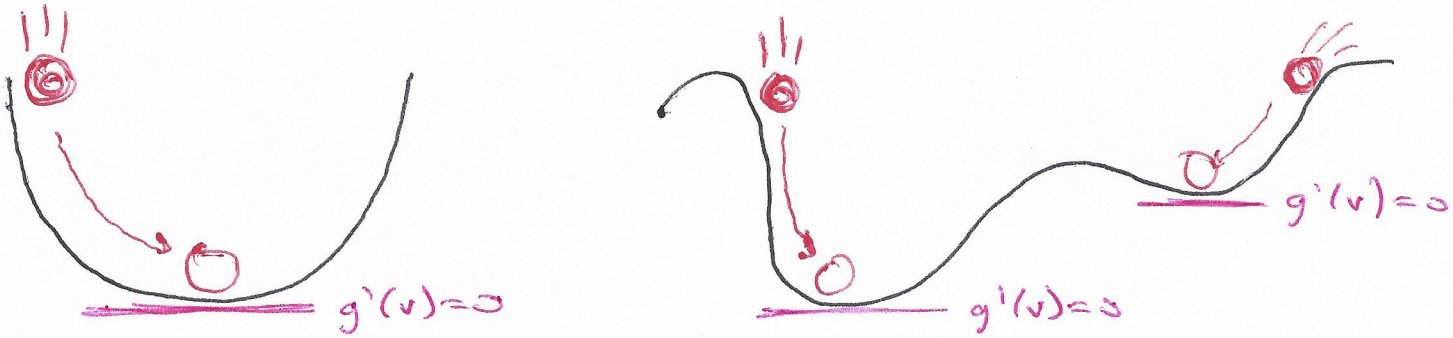
Write a short function in Python that produces the following items:

- A plot of the cost function $g(w) = \sin(w) + 0.1w^2$ over the region of input $[-10, 10]$
- A plot of the linear approximation to g at a point v where v can be any point in $[-10, 10]$
- note $g'(w) = \cos(w) + 0.2w$

So, how can we use this condition

$$g'(v) = 0$$

to find minima? Back to our marble simulator....



Mathematical optimization algorithms work like our marble simulator - you roll down hill until you reach a point where

$$g'(v) = 0$$ a stationary pt.!!

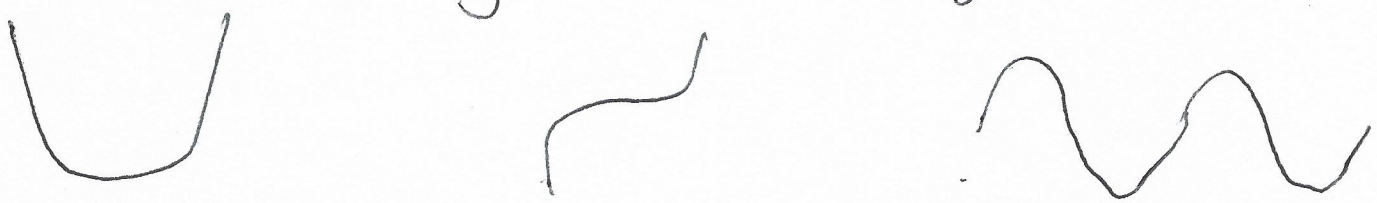
then you stop!!

- These algorithms sequentially decrease in cost function value on their way to a stationary point of g

Aside: can't we just calculate the solution to $g'(w) = 0$ by hand? (11 1/2)

- Sure for simple cases like

$g(w) = w^2$ $g(w^3) = w^3$ $g(w) = \sin(w)$

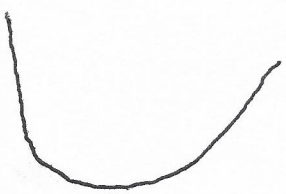


- If you remember / ever learned how to calculate derivatives you can find the stationary points of these examples

- the problem is that beyond these sorts of elementary functions its basically impossible to do e.g.

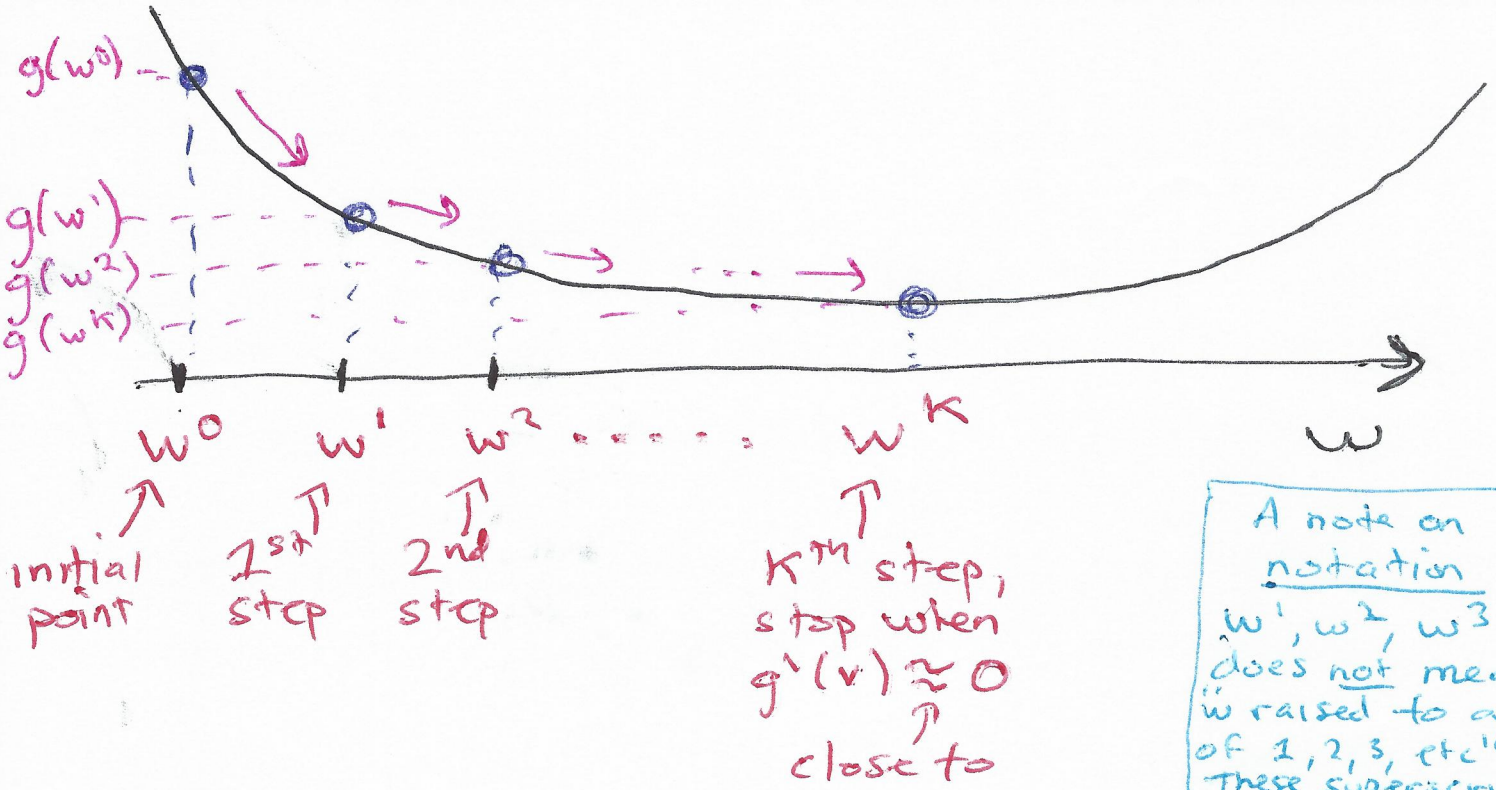
$g(w) = w^4 + w^2 + 10w$ ← looks simple, no?

do it, I dare you!



- And ridiculously hard / impossible for vector input functions, which is all we deal with in ML

Looks like this on a convex cost



A note on notation

$w^1, w^2, w^3 \ldots$ does not mean "w raised to a power of 1, 2, 3, etc" These superscripts are indicies!!

- You get down sequentially i.e. by a sequence of steps

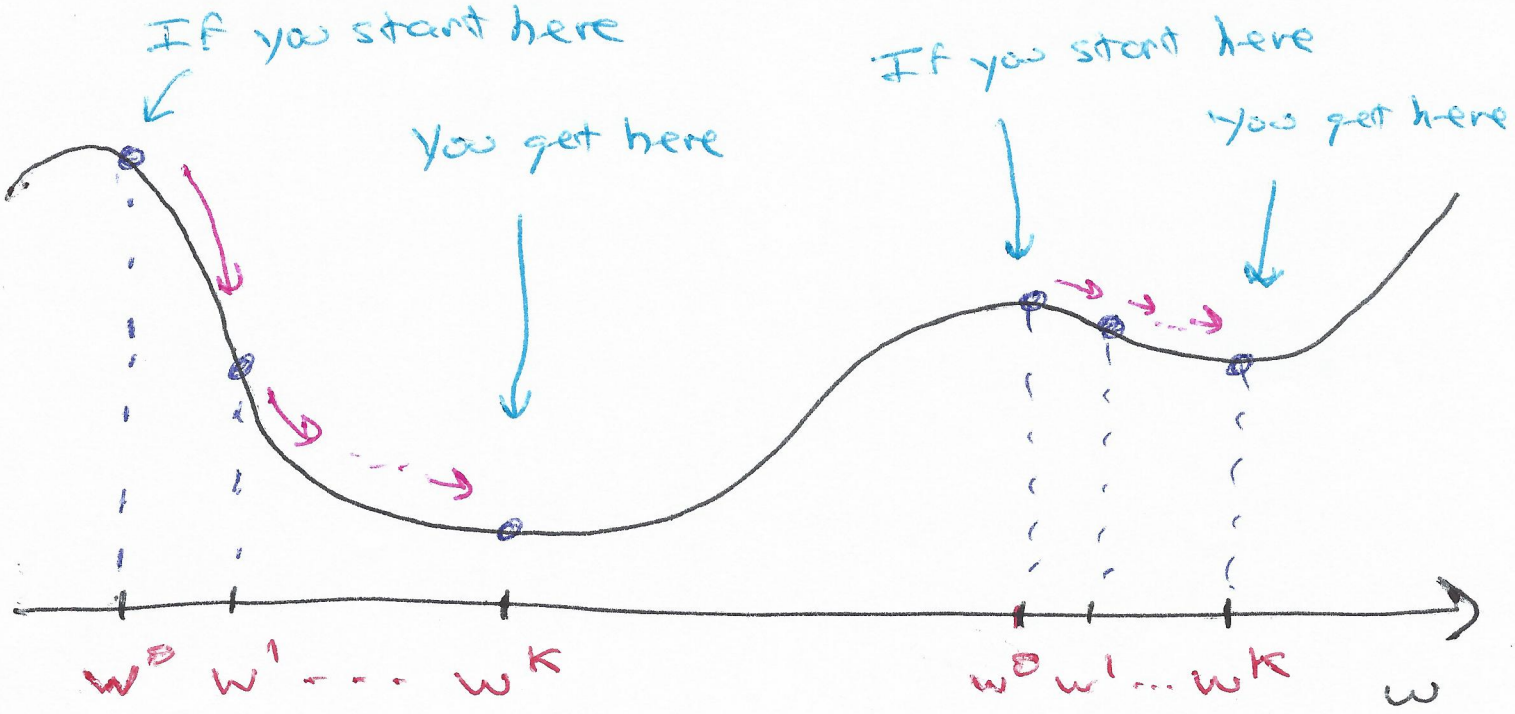
$$w^0, w^1, w^2, \ldots, w^K$$

and at each step you (typically) are decreasing the cost function value

$$g(w^0) > g(w^1) > g(w^2) > \ldots > g(w^K)$$

until at some w^K $g(w^K) \approx 0$

Same sort of deal with non-convex costs, but where we end up depends on where we begin. E.g.,



- Ok, fine, but how does it work? How can we program an algo to do this?

we need this!!

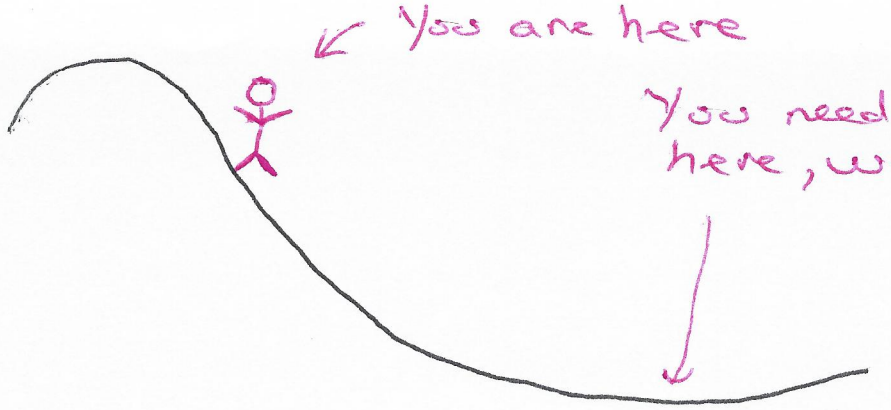
Remember: in our pictures these cost functions have only 1 weight/parameter. So we can see everything that's going on. In practice machine learning costs can have hundreds, thousands, or even millions of weights! These costs are REALLY high dimensional!! You can't visualize them!

- How do you know which way is down on a cost function that lives in 10, 100, or 10^6 dimensional space?

- Remember - you need to imagine you're completely in the dark - you can't "see" the function

- It's like if you were somewhere near the top of a mountain [redacted]. It's pitch-black out. No moon, no stars, no light whatsoever. You need to get to the bottom of the mountain. How do you do it?

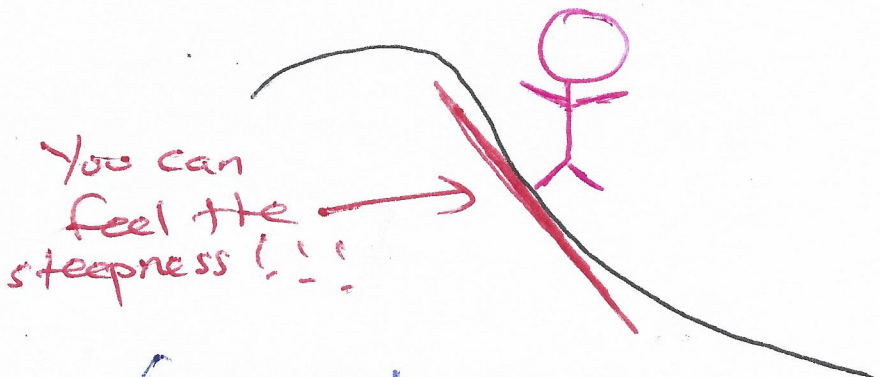
- You get no fancy equipment (no ropes, grapling hooks, sleds etc). You have to walk: step-by-step



- Remember you can't <u>see</u> anything...

  but you can <u>feel</u> something.

- You can feel the <u>steepness</u> of the ground beneath your feet!!



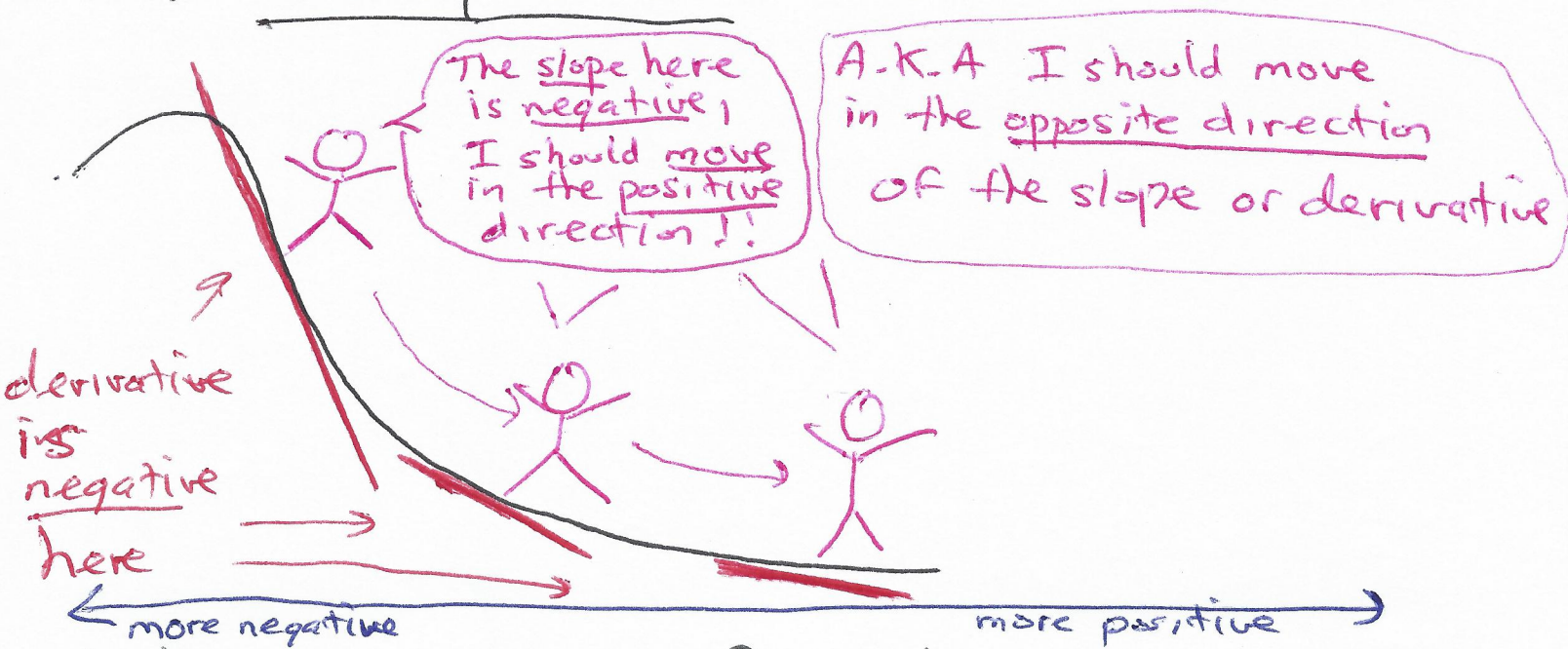
(Zoomed version of you on a mountain)

- In other words - you can feel the <u>slope</u> of the line tangent to where you are standing

  i.e. the <u>derivative there</u>

- Imagine that - you can <u>feel</u> the steepness under your feet. You can feel the slope of the tangent line — the derivative.

- Using that knowledge you can easily take <u>a step down</u> the mountain

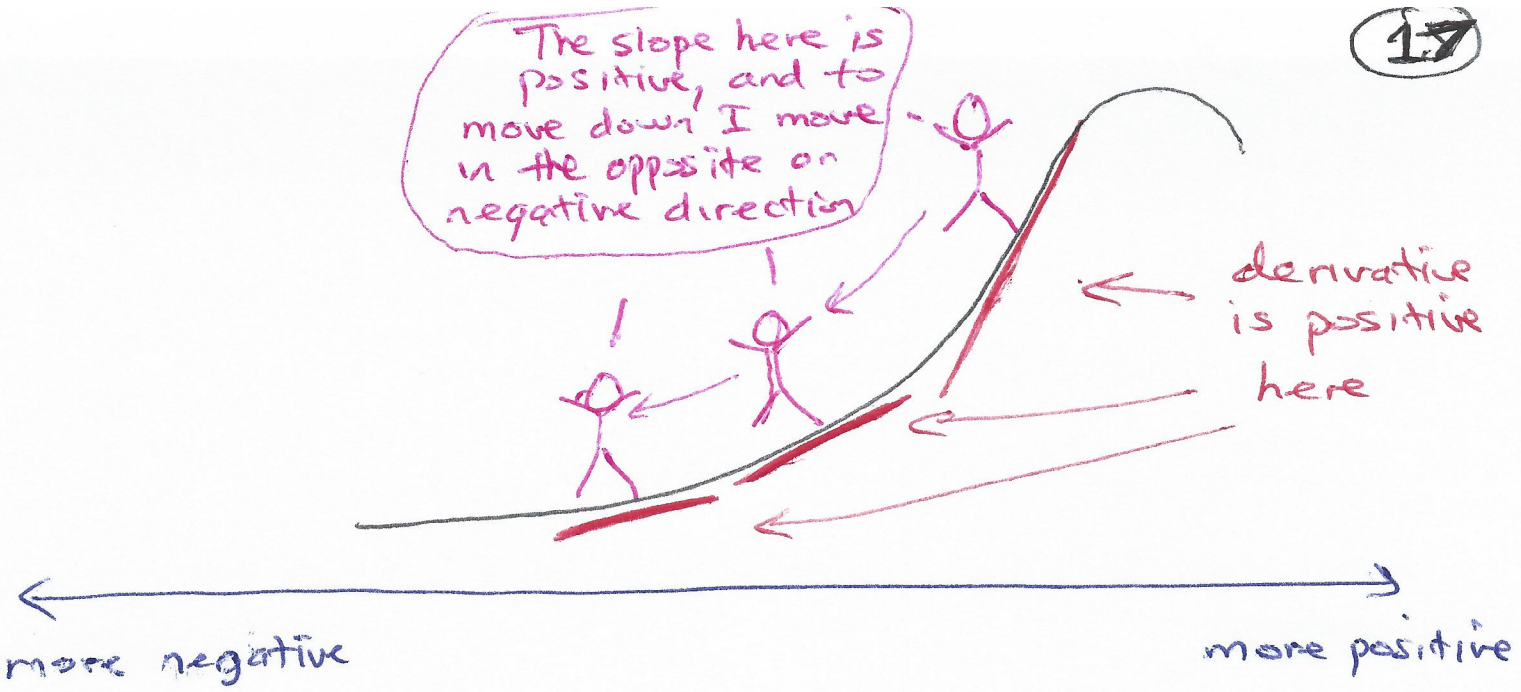


- What do you do? You take a step in the <u>opposite direction</u> <u>of the derivative</u>

or negative

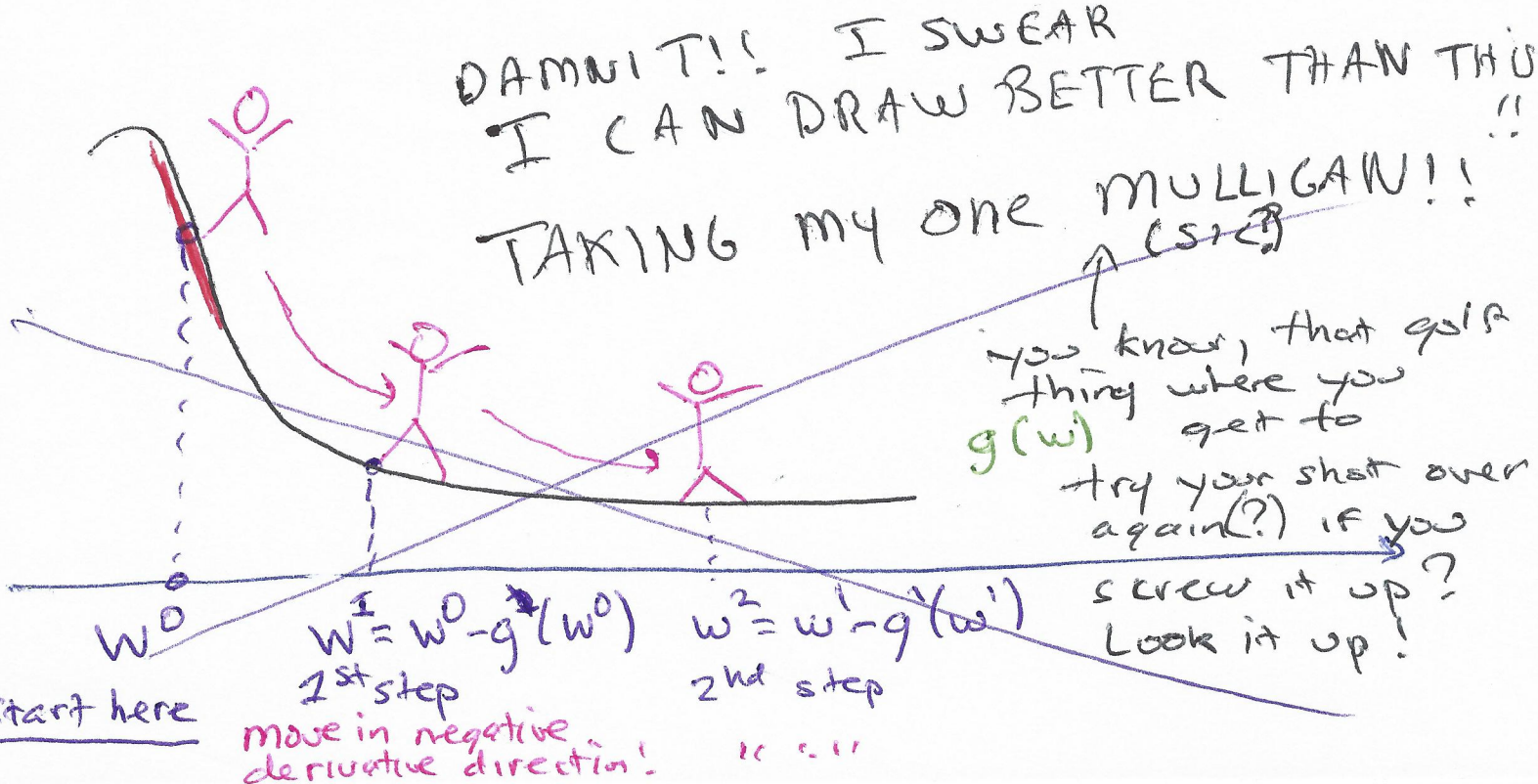
opposite of derivitive at v $= -g'(v)$

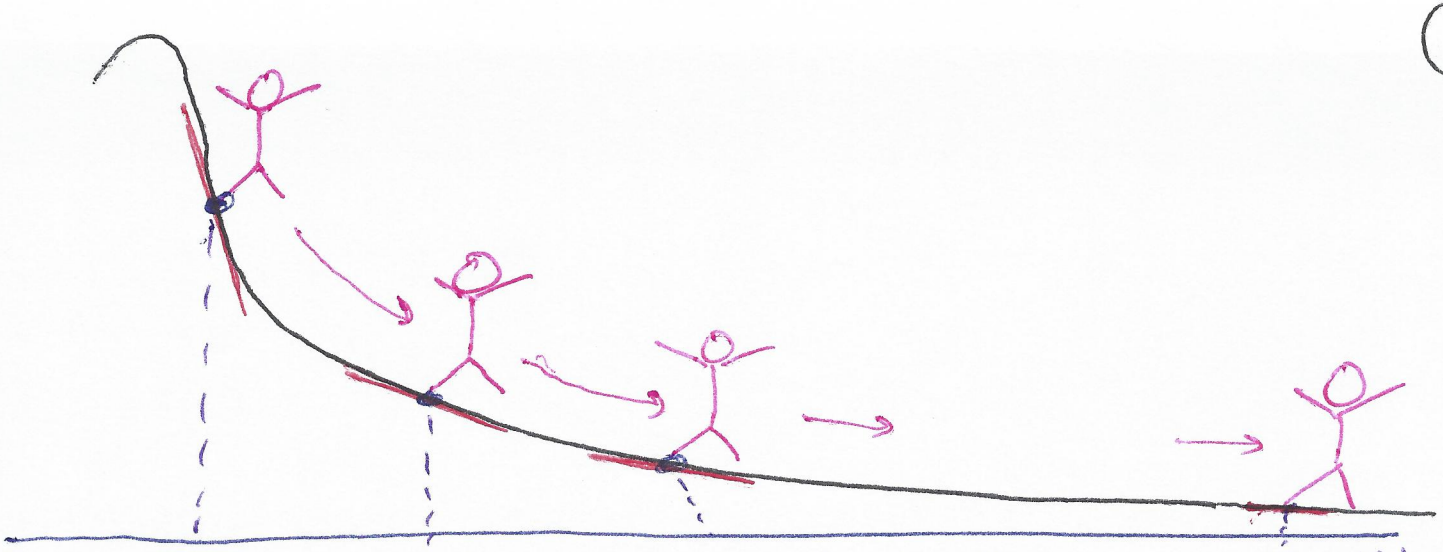
- Notice how this works if you start on the right side instead of the left.



- Still happening: to go down hill take steps in opposite (or negative) derivative direction

- Adding back in the math notation we have





w^0 — start here

$w^1 = w^0 - g'(w^0)$ — move away from w^0 in negative derivative direction

$w^2 = w^1 - g'(w^1)$ — " " " " " "

$\cdots$

$w^K = w^{K-1} - g'(w^{K-1})$ — " " " " " "

- So to go down hill we take steps of the form

$$w^j = w^{j-1} - g'(w^{j-1})$$

- One caveat: how big should each step be? We can control the stride-length by multiplying the direction $-g'(w^{j-1})$ by a tunable parameter α → we set this either once at the beginning of our journey or at each step.

- So our strategy for step taking becomes

$$w^j = w^{j-1} - \alpha\, g'(w^{j-1})$$

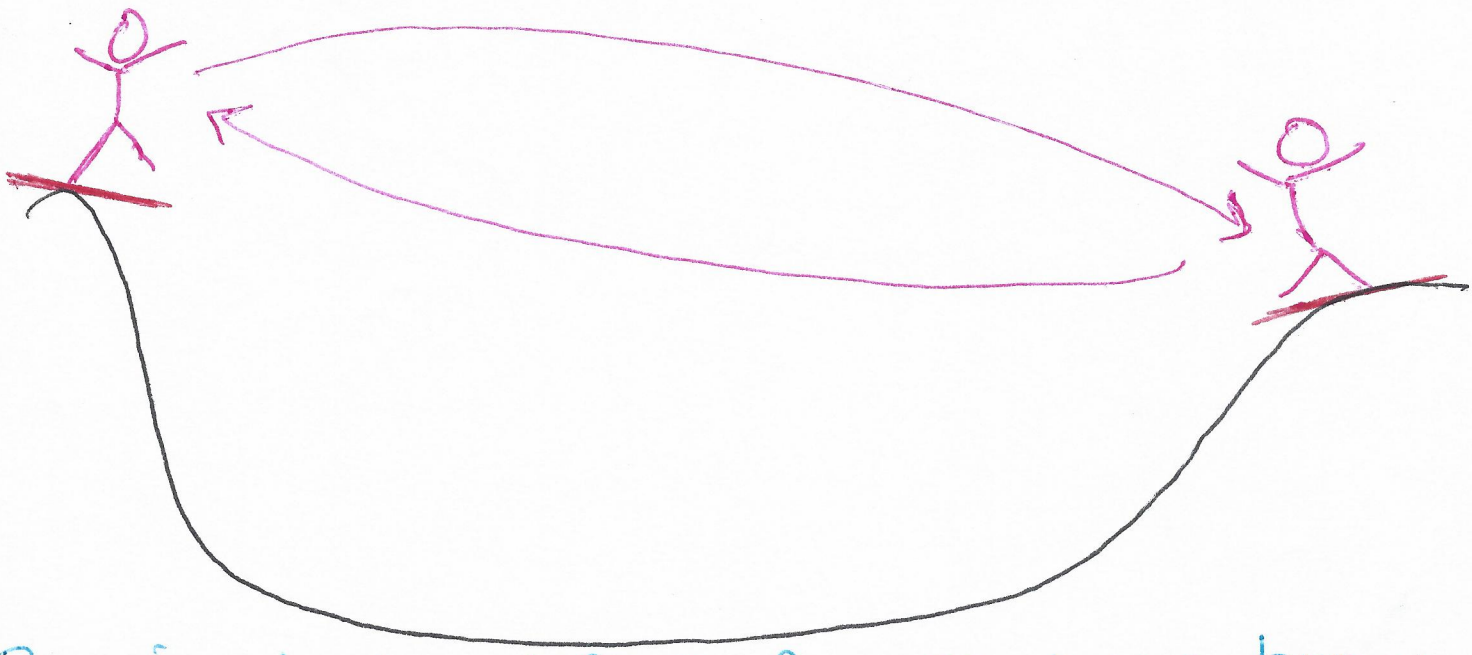
- Why add this <u>step-length</u> adjusting parameter? <u>So we don't overshoot our objective</u> as $-g'(w^{j-1})$ gives us the <u>direction</u> to travel, but doesn't tell us how big each step should be.

- Could be the case that just doing

$$w^j = w^{j-1} - g'(w^{j-1})$$

this is like if $\alpha = 1$

we might take too big of steps. So we need to control their magnitude



Bouncing back and forth forever → we know the right direction at each step, but haven't made the <u>step length</u> small enough.

- So we need that step length parameter, so our scheme can always be made to descend if we make it (step length) small enough

$$w^j = w^{j-1} - \alpha\, g'(w^{j-1})$$

- This is called the Gradient Descent Algorithm

- Jargon: α is sometimes called the "learning rate", often called the step length - which makes more sense and is less hocus-pocus-y

- Why called "Gradient Descent"?

The gradient is just the derivative of a multivariable input function.

We have 1 input in our examples so far.

So we could just call it Derivative Descent, which makes sense: we are descending in the negative direction of the derivative!!

- So, the whole Gradient Descent scheme pseudo-code

20 1/2

Gradient Descent pseudo-code

Input: step length α, initial point w^0

for: $j = 1, \ldots, K$

$$w^j = w^{j-1} - \alpha g'(w^{j-1})$$

- THATS IT, thats how kernel and neural network code you played with in level 1 parameter-tuned! And trees work on a completely similar principle.

- This algo - and its extensions (20 3/4)
(i.e. cousins ~~cousins~~) - powers much of supervised learning (regression + classification)

- There are other great algos out there, but this is the BIG ONE.

- It's like this, say you want to be Batman, and to be Batman you need to learn how to use all his tools (e.g. bat-a-rang, grappling hooks, etc.), but you only have time to learn how to use one tool

- You want to be the most like Batman you can be - so which one to learn?

- THE JET FIGHTER, or the ROCKET CAR. Learn one of these and you are 99% BATMAN! ~~Greatest Dessert~~

20 4/5

- Gradient Descent is like Batman's Jet Fighter:

learn how to use it and you might not be Batman but you're 99% as bad-ass (the thing has got MISSLES people!)

TO-DO:

- Code up gradient descent to minimize $g(w) = w^2$ on the interval $[-1, 1]$
- Use a maximum of 100 steps
- Play around with α the step size. Make it small enough to force the steps to descend!
- To debug it is helpful to print out the cost function value at each step-wise. $g(w^j)$, or store them all on a run and plot them after!

20 + ∗
4/5 < x < 5/6

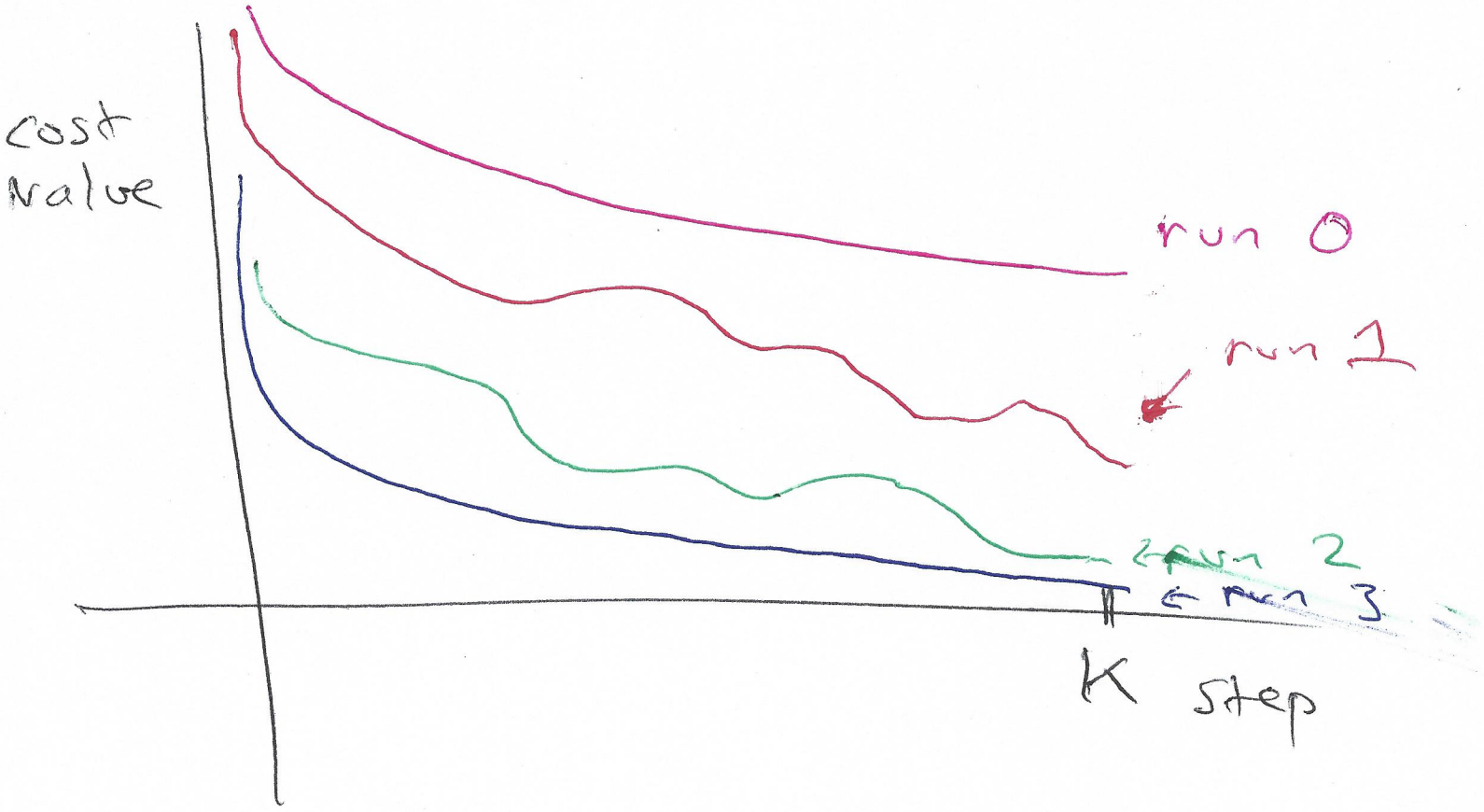
To do:

- Code up gradient descent to minimize $g(w) = \sin(w) + 0.1w$ on the input interval $[-10, 10]$
- Note here $g'(w) = \cos(w) + 0.2w$
- Play around with α the step size. Make it small enough for the algs to converge!
- Record the cost function value at each step $g(w^j)$ for a fixed number of steps and <u>return</u> the history $[g(w^0)\ g(w^1) \cdots g(w^K)]$ <u>as well as</u> w^0
- Run your algorithm with a random initial w^0 in $[-10, 10]$ for <u>several times</u>

Plot all of the cost histories together on the same plot to see which initial point led to the best solution

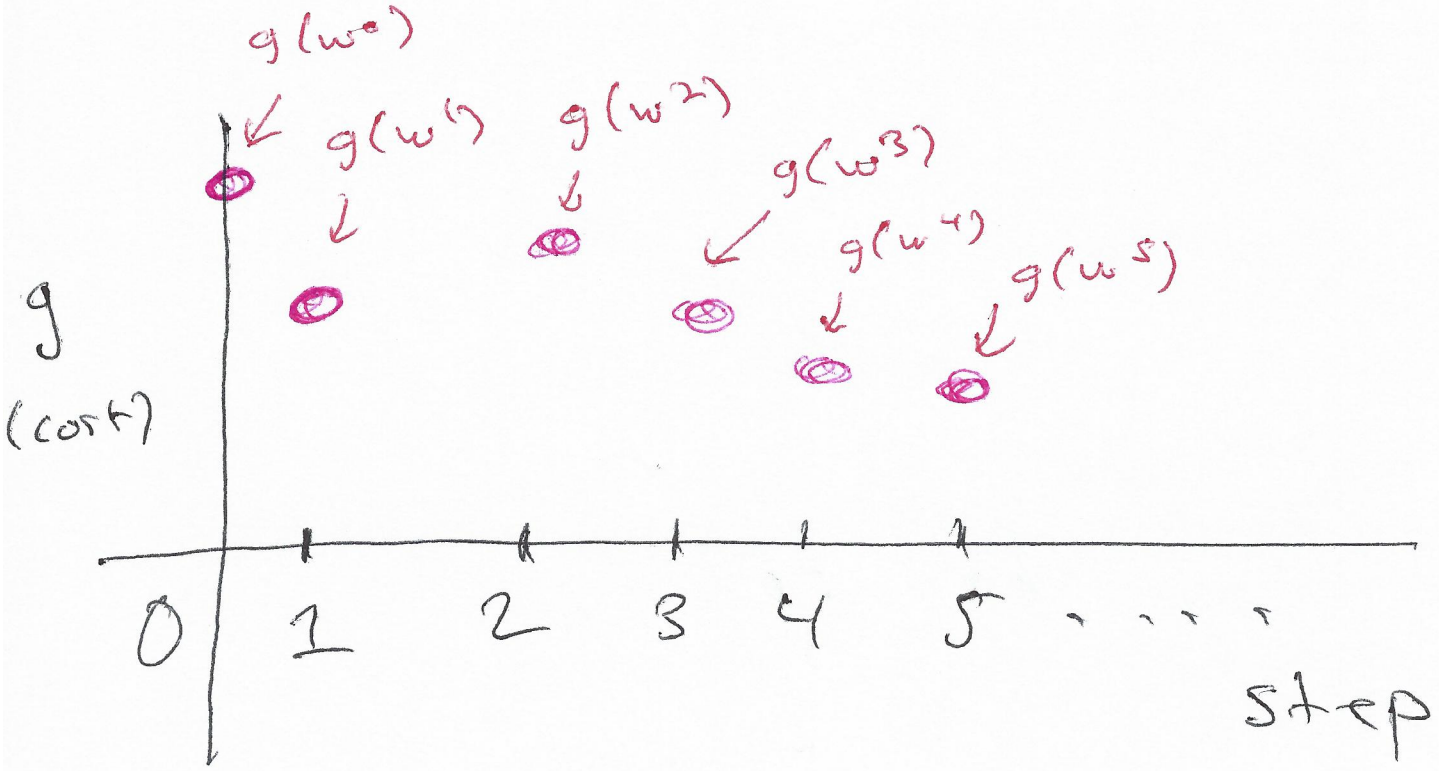
20+?

You should see something like this when you plot the various histories



20 5/6

What does that plot of the cost values look like?



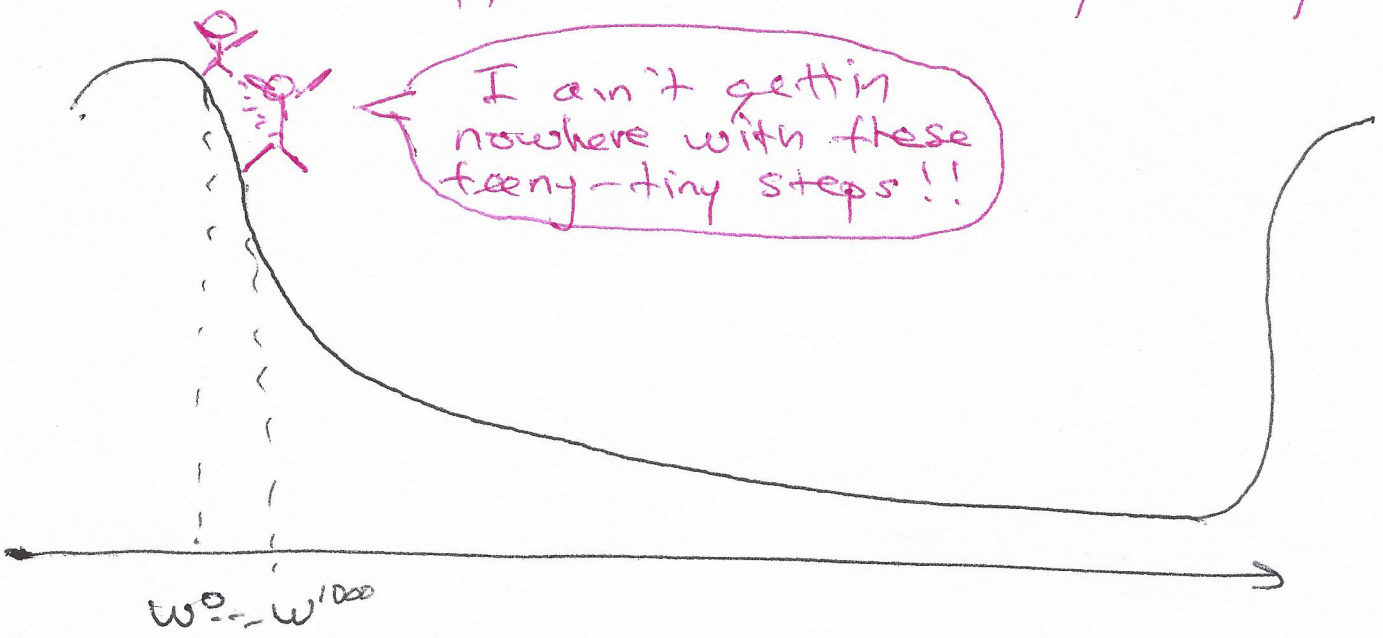
- If your algorithm + step size is chosen correctly these values should decrease (to 0 in this case) eventually!

- Make your code so that you can initialize anywhere in the interval $[-1, 1)$ and you reach the minimum at $w = 0$ (approximately)

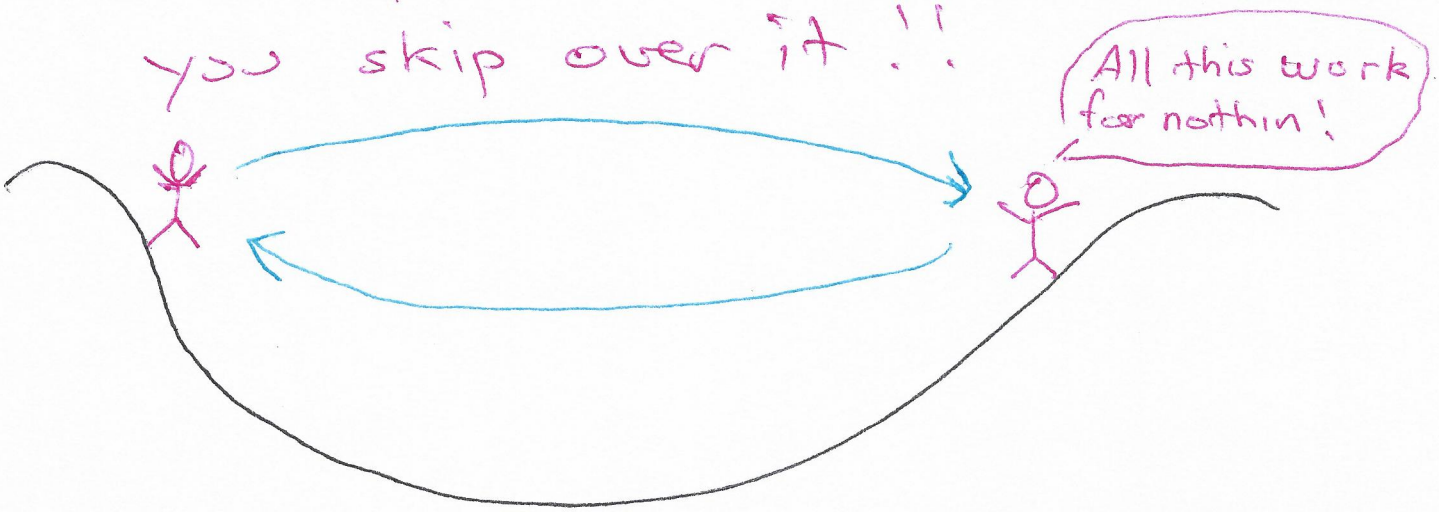
Q & A Time!

- How to choose the right value of step length α?

→ if too small we travel downward, at each step, but we don't get anywhere



→ If too big → bounce around! never get to the bottom cause you skip over it!!



- Before we get an answer to this question.... lets play!

  → download the mlrefined repo at

https://github.com/jermwatt/mlrefined

and check out the first 2 demos in the

Basic_optimization_demos

notebook.

- These let you play around with gradient descent on two real 1-dimensional functions

- Try messing around with the step length value of each and run the demos several times

- What did you learn from playing with / breaking these demos?

- Ok, so how to choose α step length in general?

  - The naive way is extremely common: choose a wide range of values for α e.g. $[10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}; 1]$ and run Gradient Descent for each. Then pick / keep the run that gives you the best (i.e. lowest cost function value) result.

- or - even simpler - choose one small value and live with the consequences

- For ML cost functions there are elegant mathematical ways of determining very reasonable step length values, but still more common to see the "guess and check" method.

Next Question: how do we know how many steps to take?

- Same sort of answer. You can stop when the derivative in magnitude is "small enough", but in practice people pick a number of steps and just run! If the result isn't good enough, run it some more (unless

the cost function value isn't changing any more, indicating that you have reached a stationary point)

- How do I know that the algorithm stops at a stationary point if I have chosen the step length correctly & taken a bunch of steps?

  - the form of the step is

$$w^j = w^{j-1} - \alpha g'(w^{j-1})$$

if we aren't moving around any more then

$$w^j \approx w^{j-1}$$

and so

$$g'(w^{j-1}) \approx 0$$

i.e. a stationary point!

Q:

- Aren't there theoretical (mathematical) ways of doing this stuff? e.g. This all seems messy → choosing the step length α by trial & error, just setting the number of steps to something large and going with the flow....

- A: Yes, there are ways of deriving working step lengths, maximum # of steps, etc by using math

For those who are really curious I would suggest you look at chapters 2 and 8 of my book, which goes into these details

Nonetheless, while there are certainly topics I recommend you dive

into if you're curious as they definitely help you understand Gradient Descent at a much deeper level, and that knowledge can improve your work performance if used appropriately, it is still common place to just do the guess and check approach to step length and max steps in practice. ~~So~~ ~~operating~~ If you open up a lot of commonly used libraries for ML - e.g., scikit and Tensorflow - you'll see that this is precisely how they've been setup

27 1/2

- Q: How do we compute the derivative g'' in general?

- A: There are just a few calculus based rules for computing derivatives, and using them in combination we can compute the derivative of any valid function. These rules include
  - the chain rule
  - the product rule
  - the sum rule
  - the power rule
  - rules for differentiating elementary functions like log, exp, sin, etc.

Nonetheless, computing the derivative of complicated functions e.g. compositions of elementary functions

as well as vector input functions is a difficult task that one needs to be careful when performing by hand, to avoid careless errors

(e.g. if you have some familiarity with the basic rules, go ahead and think about how messy the computation of

$$g(w) = \tanh\left(\sum_{j=1}^{N} \sin(jw)\cdot\cos(jw)\right)$$

$$\cos(\sin(\cos(w)))$$

is. Lots of repeated use of chain rule here!!)

So, with that said, how skilled do you need to be at computing derivatives?

my advice

- Get as comfortable computing derivatives as you are at computing multiplication of 2 numbers
- You know the basic rules of multiplication, so become familiar with the basic rules (listed earlier) of computing derivatives
- Be able to do simple examples using elementary functions e.g.

$$g(w) = \sin(w)$$

$$q(w) = \log(w)\, e^{w}$$

Akin to being able to do simple multiplication by hand. E.g.

123×4

27×56

- But just like you turn to a calculator when multiplication gets

tedious, e.g.

$157,479,230 \times 562,457$

do as you do with multiplication:

use a table or calculator

- There are plenty of resources / tables that list the derivatives of complicated cost functions
- there are also calculators!! these are sometimes called automatic differentiators.
- So, in short, with derivatives the #1 most important thing is to understand the concept, and to be able to compute simple examples (as you can do with multiplication), otherwise use a table or calculator to compute!

So, our Gradient Descent Algorithm pseudo code, fully commented, looks like

either use small value or try over range of values several orders of magnitude e.g., $[10^{-3}, 10^{-4}, \ldots 10^{-1}, 1]$

Input = step length α, initial point w^0, maximum number of iterations K

for $j = 1 \ldots K$

$$w^j = w^{j-1} - \alpha g'(w^{j-1})$$

w^0: random

K: as large as you can handle

g': use table or auto-differentiator to calculate g'

optional but very useful

- Record cost function value $g(w^j)$ and return history $[g(w^0), g(w^1) \ldots, g(w^K)]$ for analysis and return
- Record weight history $[w^0 w^1 \ldots w^K]$ and return
- OR return best weight / cost function value from the run. i.e. if

$$k^* = \arg\min_j g(w^j)$$ return

$w^{k^*}, g(w^{k^*})$ ← best weight / value pair

Going up, up, up (dimension-wise, that is)

- With machine learning, cost functions almost always have more than 1 parameter to tune. In fact they can have 100s, 1,000s, or even 100,000s of parameters that need tuning
- So how does what we discussed previously for ~~Gradient descent~~, gradients, Gradient Descent, etc. generalize for cost functions of many inputs?
- All of it generalizes directly with few caveats
- Lets go through and talk first about notation, then see how our concepts generalize
- Please be sure to complete the vector/matrix jupyter notebook - to familiarize yourself with necessary

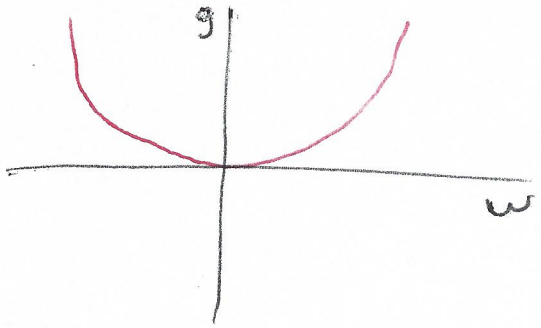
Concepts before proceeding

Scalar input (1-d)

Variable: w

Cost: $g(w)$

e.g. $g(w) = w^2$

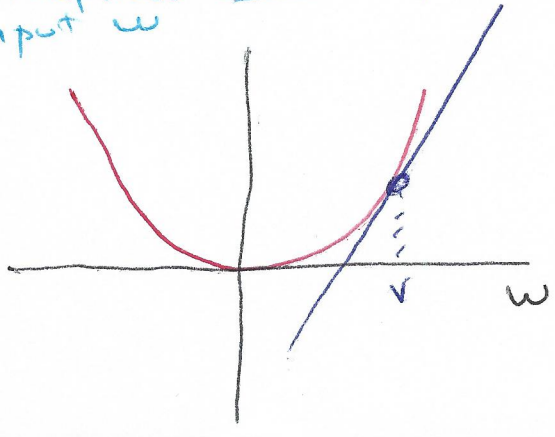


Derivative: measure of steepness of tangent line at a point v, denoted

$g'(v)$ or $\frac{\partial}{\partial w} g(v)$

↑ new equivalent notation

Steepness with respect to input w



Vector input (N-d)

Variable: $\vec{w} = [w_0, w_1, \ldots w_N]$

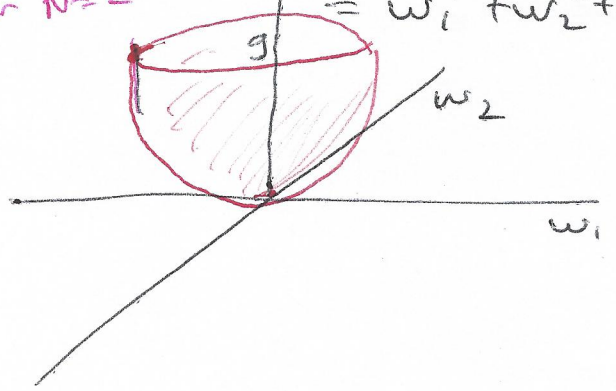
↑ N entries!

Cost: $g(\vec{w})$

e.g. $g(\vec{w}) = \vec{w}^T\vec{w}$

$= w_1^2 + w_2^2 + \ldots + w_N^2$

for $N=2$

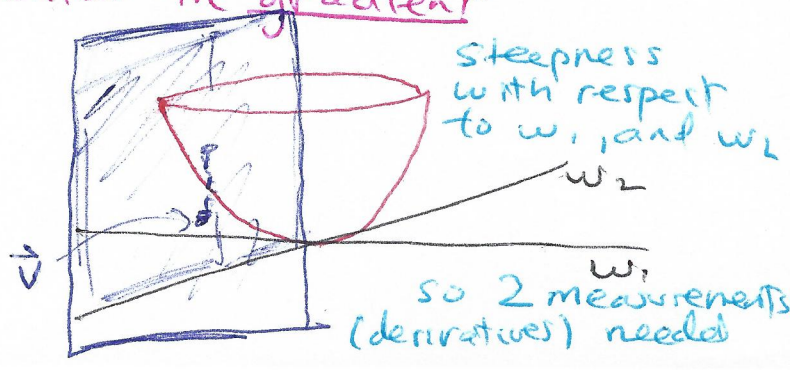


Derivative: N measures of steepness of tangent hyperplane at a point $\vec{v}$, denoted

$\left[\frac{\partial}{\partial w_1} g(\vec{v}), \frac{\partial}{\partial w_2} g(\vec{v}), \ldots, \frac{\partial}{\partial w_N} g(\vec{v})\right]$

or $\nabla g(\vec{v})$ ← an N-length vector of (partial) derivatives

↑ generalized derivative called the gradient



Gradient Descent

Input: w^0, α, K

for $j = 1 \ldots K$

$w^j = w^{j-1} - \alpha g'(w^{j-1})$

optional:
- return cost history and/or $[g(w^0) \ldots g(w^K)]$
- return parameter history or $[w^0, \ldots w^K]$
- $w^{k^*}, g(w^{k^*})$ where $k^* = \underset{j}{\text{argmin}}\, g(w^j)$

Gradient Descent (31)

Input $\vec{w}^0, \alpha, K$ (Note all of these are N-dim vectors!)

for $j = 1 \ldots K$

$\vec{w}^j = \vec{w}^{j-1} - \alpha \nabla g(\vec{w}^{j-1})$

optional
- return cost history and/or $[g(\vec{w}^0) \ldots g(\vec{w}^K)]$
- return parameter history/or $[\vec{w}^0 \ldots \vec{w}^K]$
- $\vec{w}^{k^*}, g(\vec{w}^{k^*})$ where $k^* = \underset{j}{\text{argmin}}\, g(\vec{w}^j)$

To-do:

- Perform gradient descent to minimize the convex function $g(\vec{w}) = \vec{w}^T \vec{w}$ where $\vec{w} = [w_1, w_2]$, i.e. $N = 2$
- Pick your initialization at random over the interval $[-1, 1] \times [-1, 1]$
- try several values of α, choices of K, record and return the cost history $[g(\vec{w}^0) \ldots g(\vec{w}^K)]$ with each run and plot to see which values tend to work, and which ones don't

- note the true minimum is at

$\vec{v} = [0,0]$ and here $g(\vec{v}) = 0$

- note that $\nabla g(\vec{w}) = \begin{bmatrix} 2w_1 \\ 2w_2 \end{bmatrix}$

To-do:

- repeat the previous exercise, except now

$$g(\vec{w}) = \sin(\vec{w}^T \vec{1}) + 0.01\, \vec{w}^T \vec{w}$$

where $\vec{w} = [w_1\ w_2]$ and $\vec{1} = [1\ 1]$

- Here ~~collect~~ initialize at random $\vec{w}^0$ in $[-10,10] \times [-10,10]$

- Run several times with random initializations and compare cost histories to find the lowest minimum on the cost function that you possibly can