Gradient Descent

Learning Objectives

gradient descent = optimization algo.

- Describe what optimization is, ...
- ... and describe how gradient descent solves optimization problems
- Discuss what can go wrong when doing gradient descent
- Carry out gradient descent manually in Python

↳ learning rate
η (eta)

Optimization

Define: Optimization

The term **optimization** refers to computing either the minimum or maximum of an **objective function**, maybe subject to some **constraints**.

There are many ways of solving an optimization problem, and you may already be familiar with one of them: by computing the **derivative** of a function and setting it to zero.

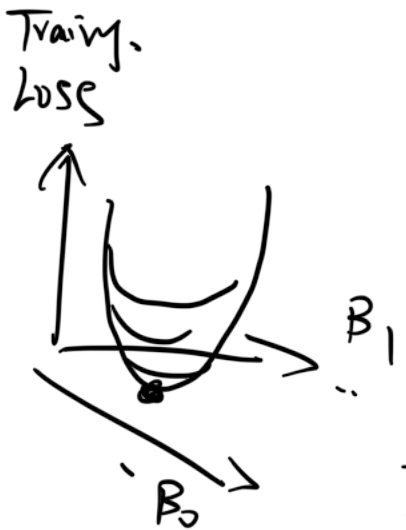
Define: Derivative

What is a derivative?

Define: Derivative

A **derivative** of a function at a point is the value of the slope of the tangent line at that point.

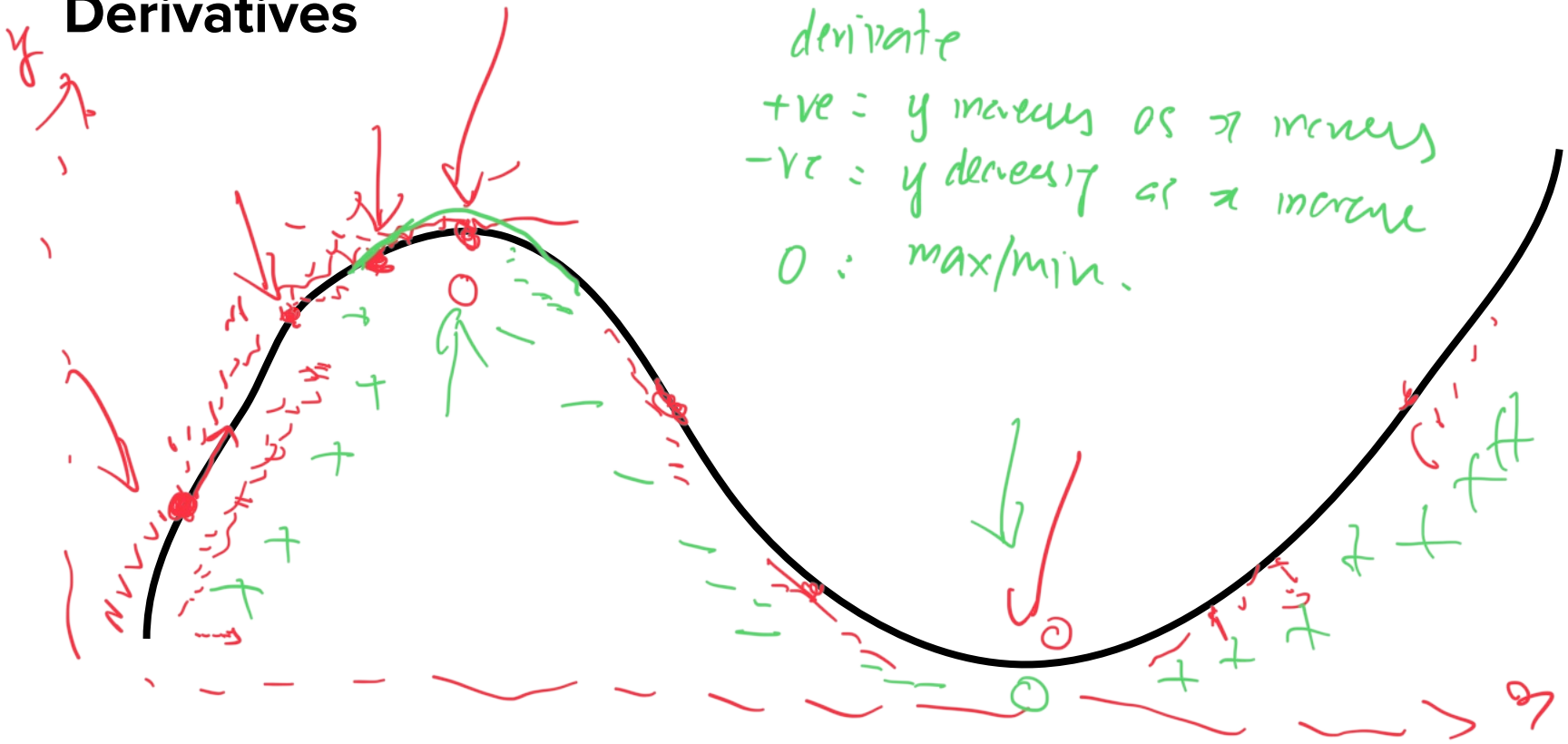
$\frac{dy}{dx}$



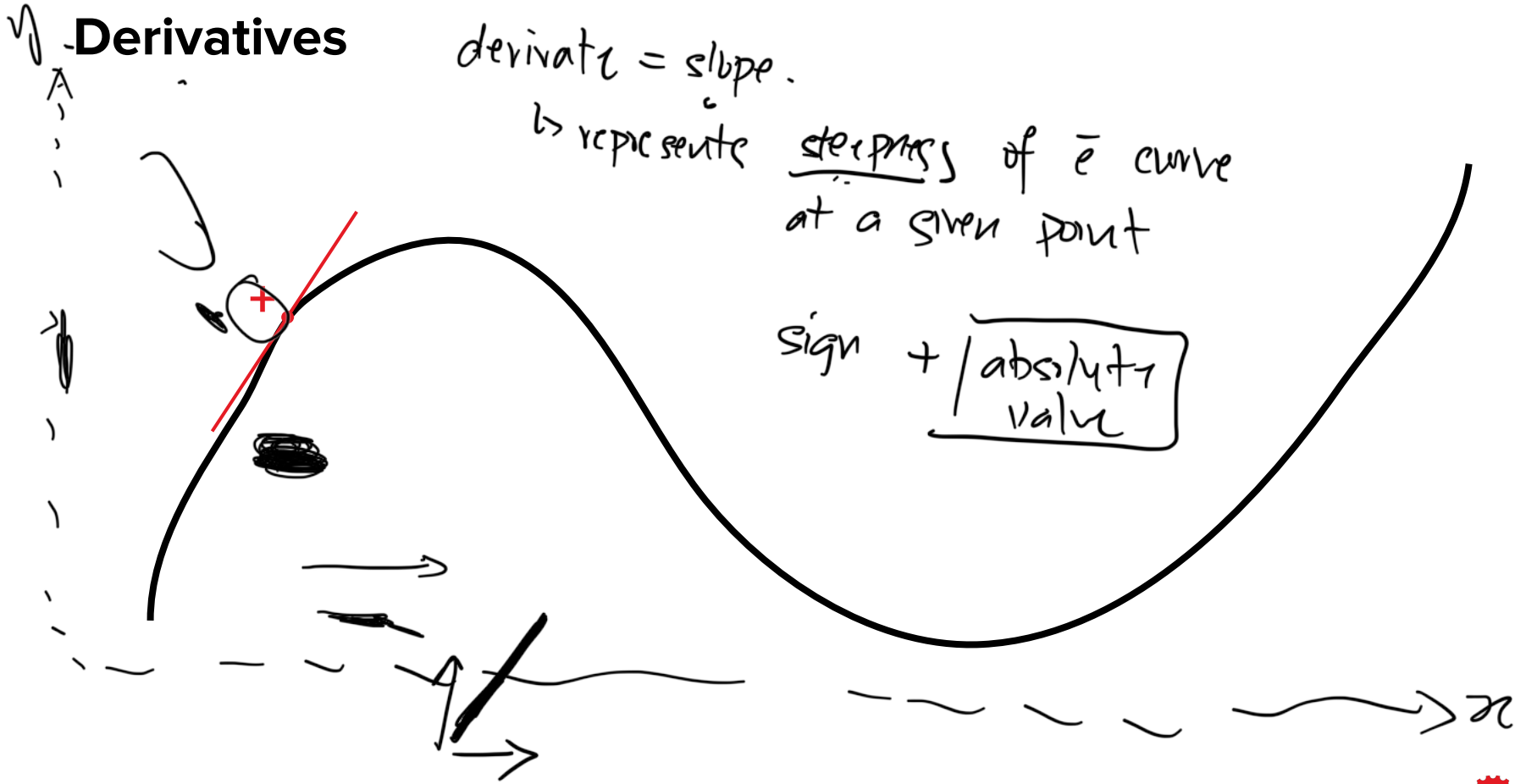
$y = B_0 x + B_1 x^2$

optimization algorithm that helps you find the optimal parameters for your model/ your loss fx I hope.

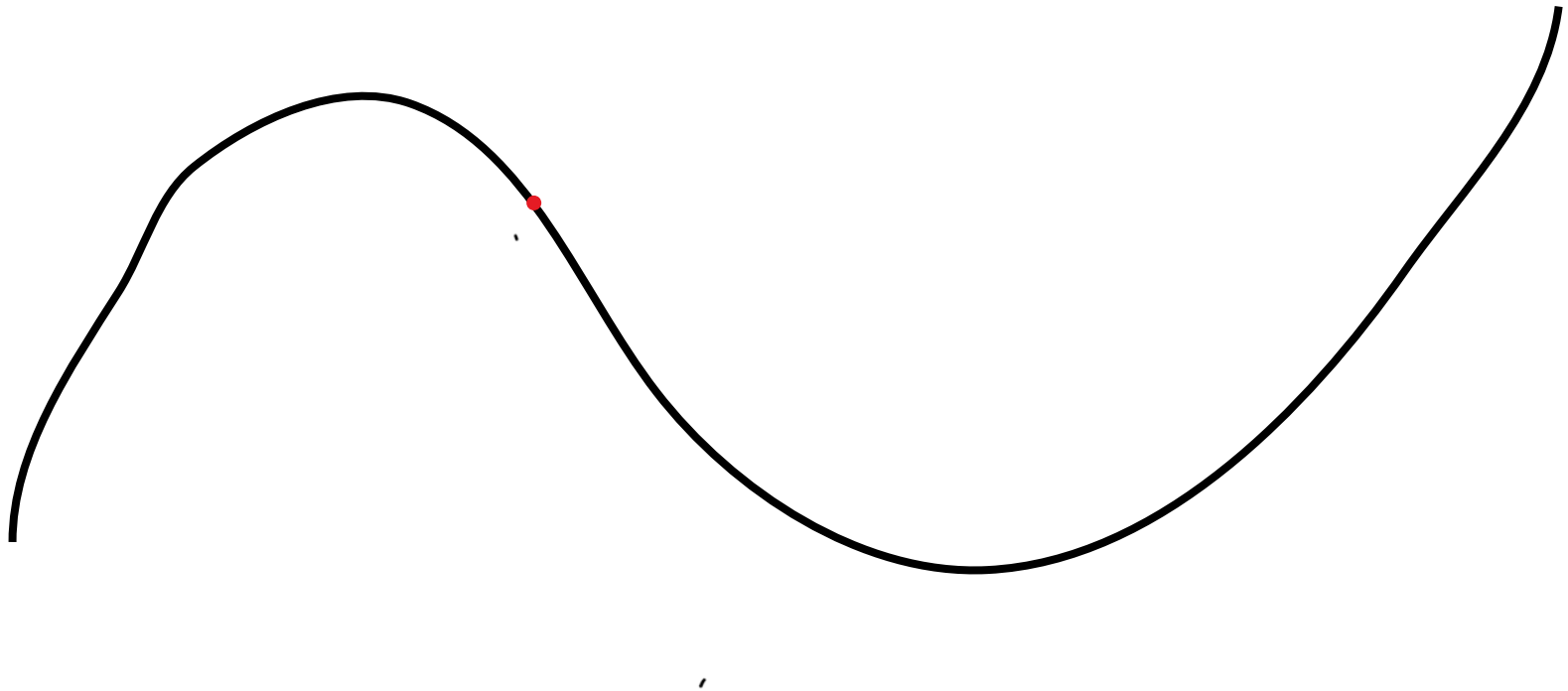
Derivatives



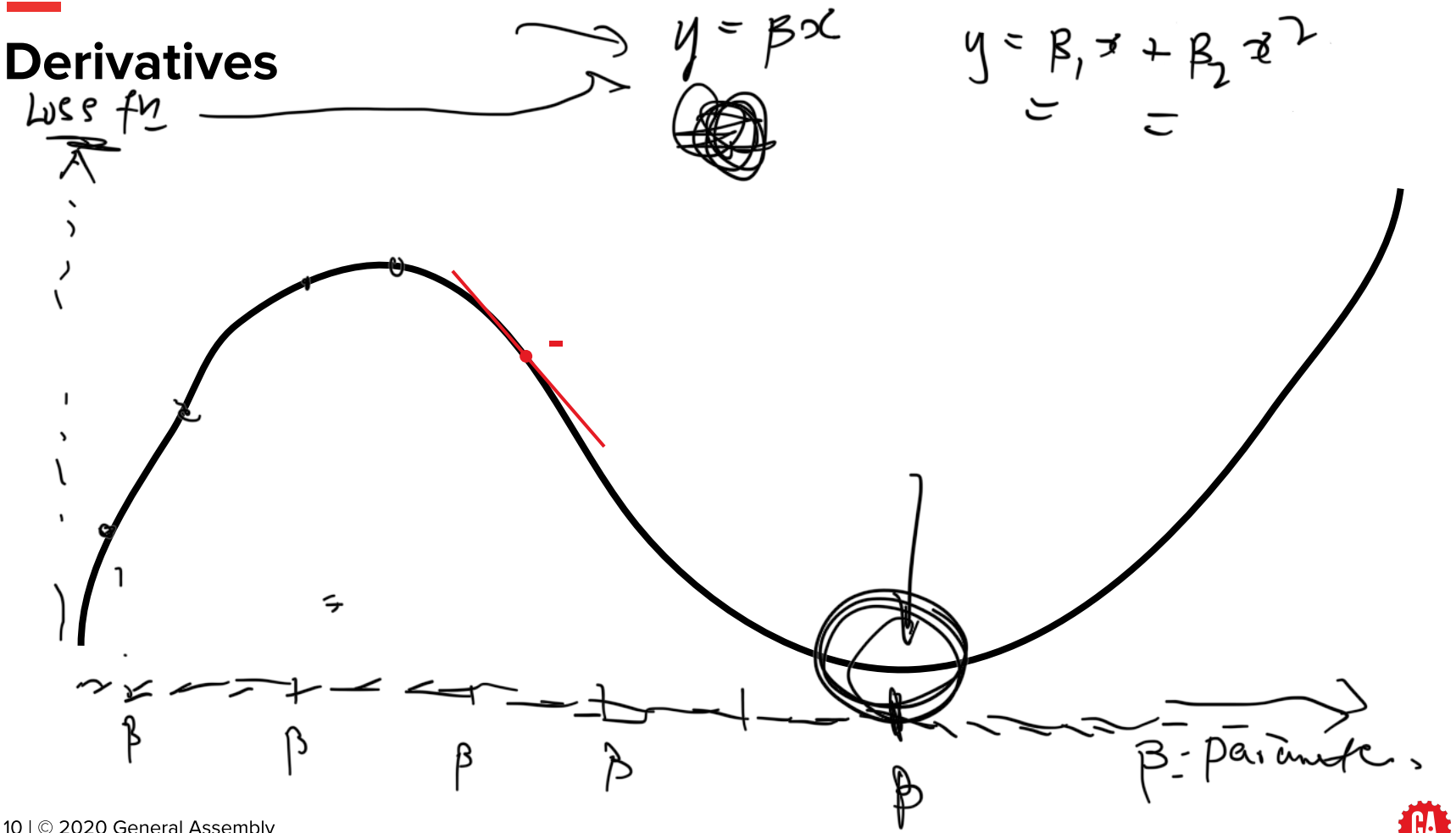
Derivatives



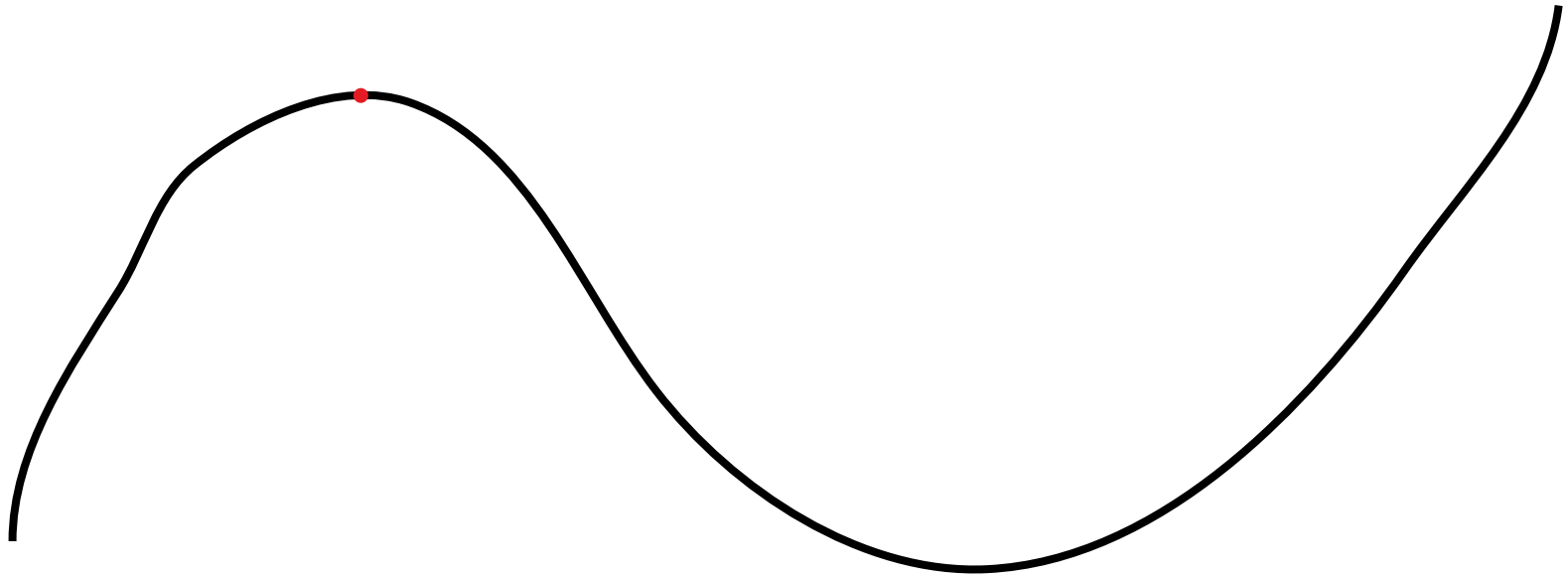
Derivatives



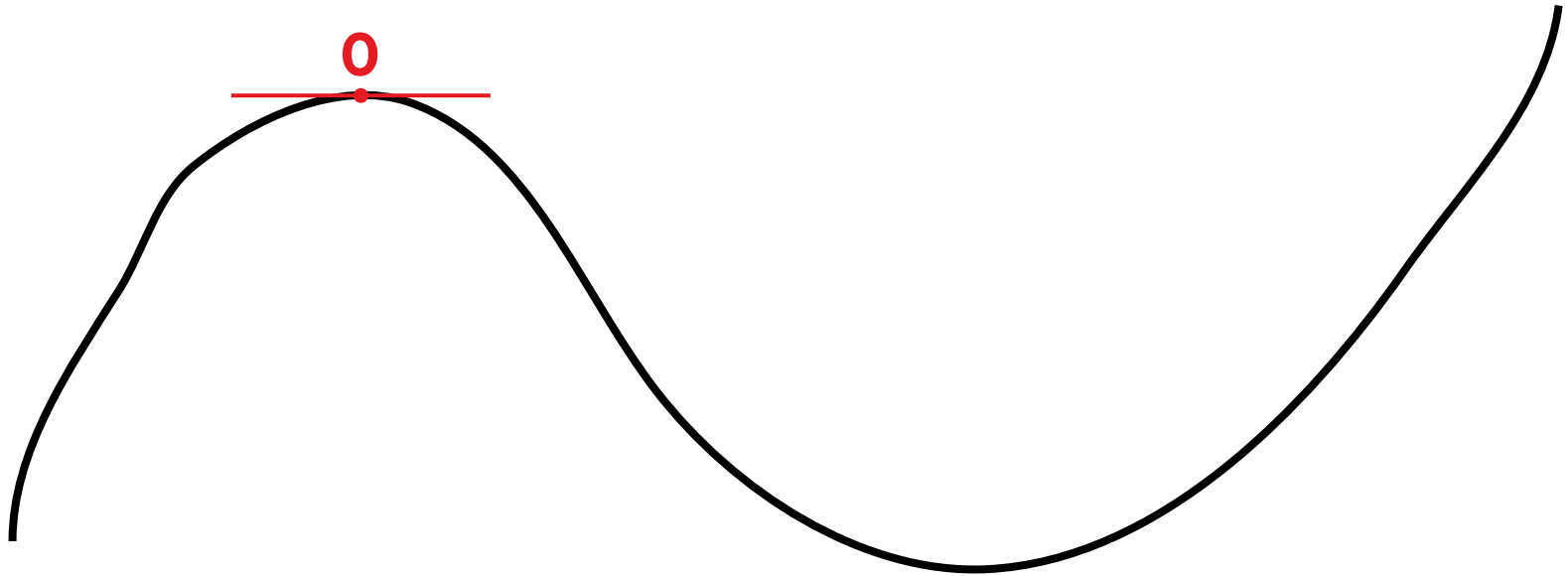
Derivatives



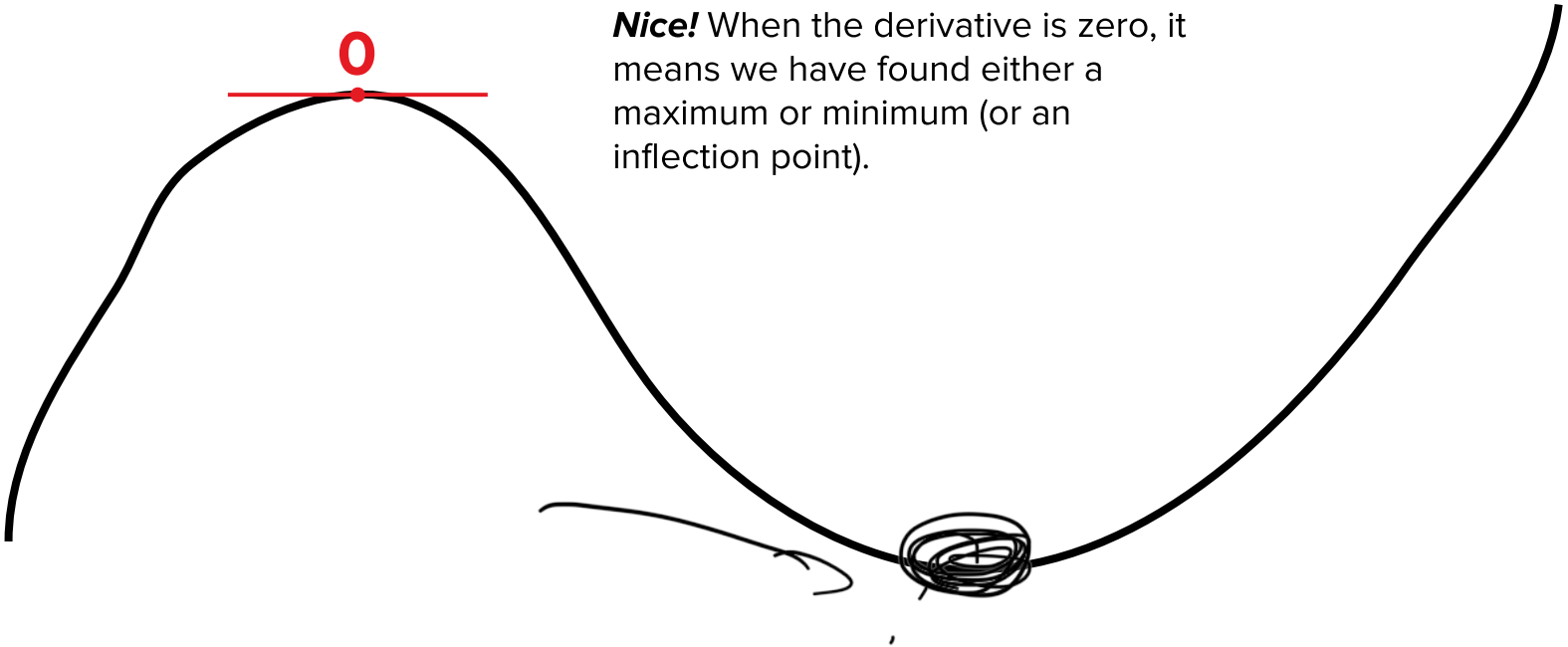
Derivatives



Derivatives



Derivatives



Nice! When the derivative is zero, it means we have found either a maximum or minimum (or an inflection point).

Optimizing the “old fashioned way”

In calculus class, you might have optimized a function by computing its derivative, setting it to zero, and then solving for *x*.

$$y$$

$$\frac{dy}{dx} = 0 \Rightarrow x\ ?$$

Optimizing the “old fashioned way”

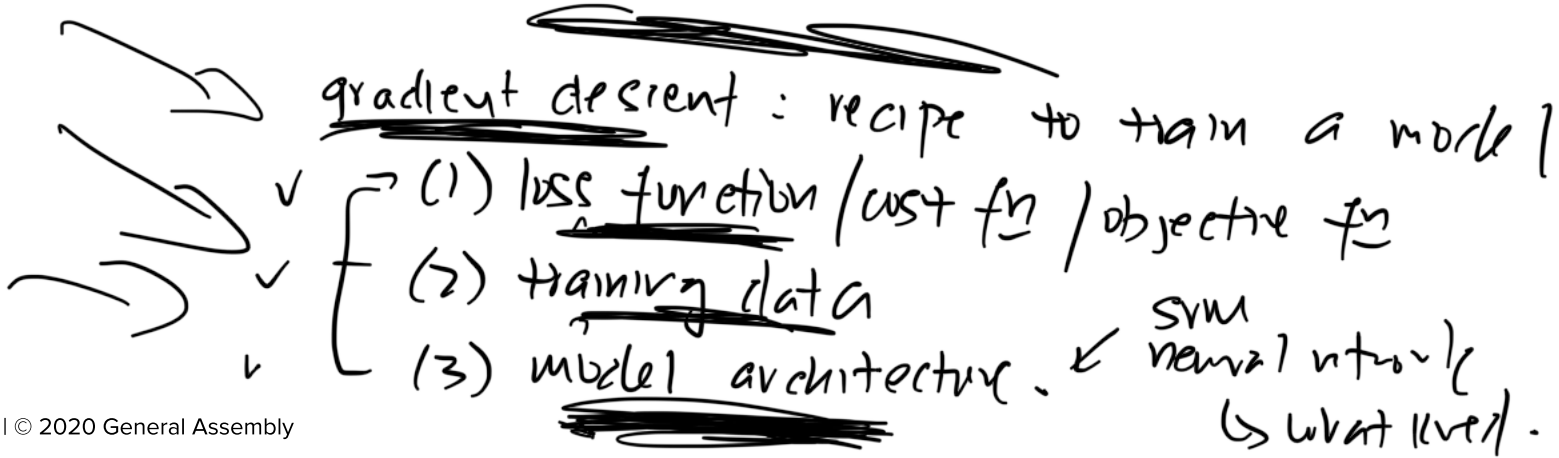
In calculus class, you might have optimized a function by computing its derivative, setting it to zero, and then solving for *x*.

Today’s lesson is about what happens when you can’t do that!

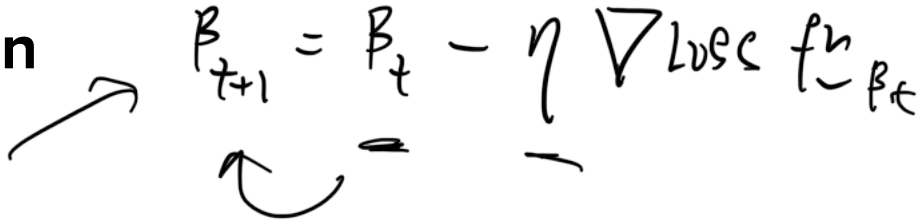
Optimizing for data science

(Almost) all machine learning algorithms involve some sort of optimization problem. With one notable exception (OLS), they are all unsolvable the "old fashioned way".

Instead, we'll need to use **gradient descent** to optimize **loss functions**.



Define: Loss function

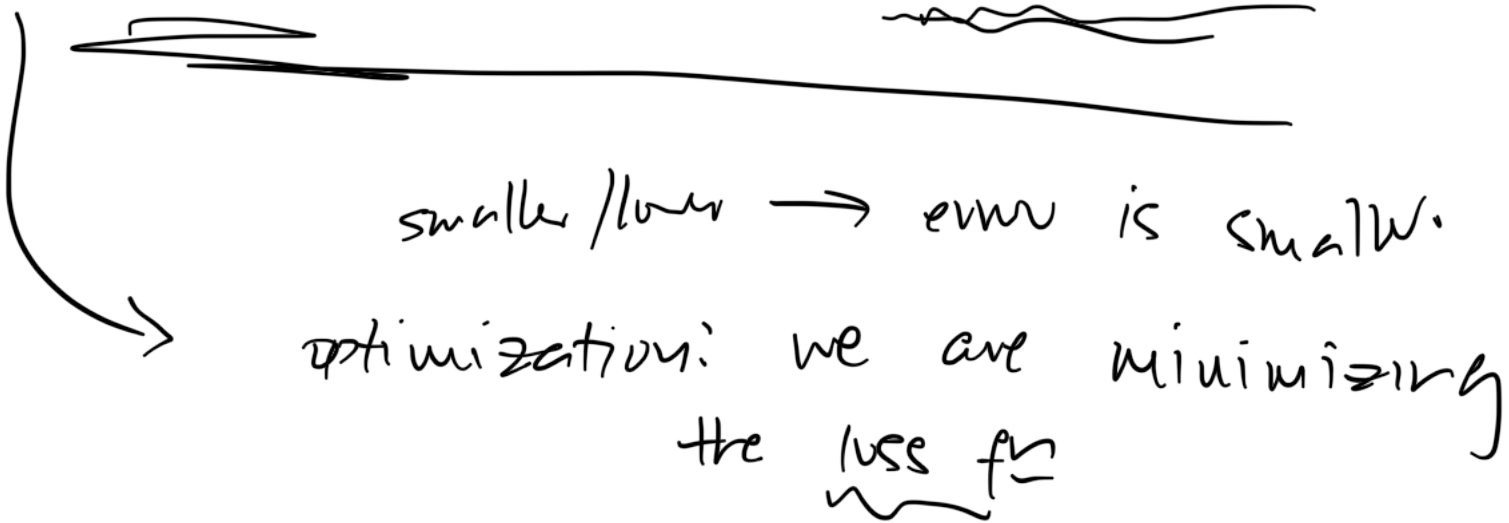


In your own words, what is a loss function?

Name all the loss functions you know.

Define: Loss function

A **loss function** is a function that describes the difference between y and $\hat{y}$. In order for something to be a loss function, **lower values must be better**.



Define: Loss function

A **loss function** is a function that describes the difference between y and $\hat{y}$. In order for something to be a loss function, **lower values must be better**.

This means that R^2 (for regression) and accuracy (for classification) don't count as loss functions!

OLS: Minimizing the MSE loss function

The goal of OLS is to find the value of b that minimizes the MSE:

$$\text{minimize} \quad \sum \left(y_i - \mathbf{x}_i^T \beta\right)^2$$

$$(y - \beta x)^2$$

Minimizing the MSE loss function

The goal of OLS is to find the value of b that minimizes the MSE:

$$\text{minimize} \quad \sum \left(y_i - \mathbf{x}_i^T \beta\right)^2$$

This is one of the only problems interesting to data scientists that we can solve the “old fashioned way” in **closed form**:

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Minimizing the LASSO loss function

The LASSO, on the other hand, is harder. We can't solve this with simple calculus:

$$\text{minimize} \quad \sum \left(y_i - \mathbf{x}_i^T \beta\right)^2 + \lambda \sum |\beta_j|$$

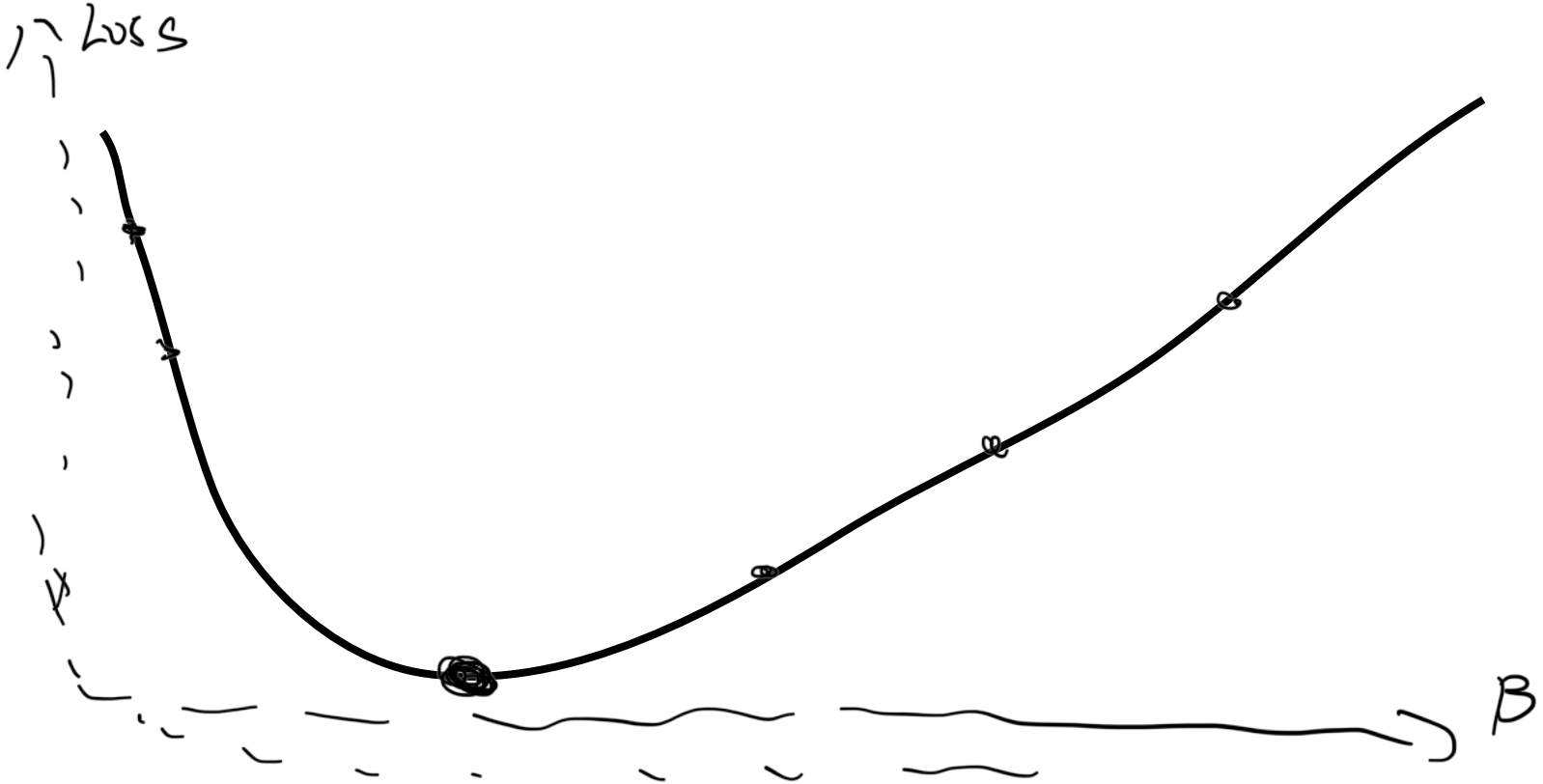
Minimizing the log-loss function

Still more difficult is the **log-loss function** (also called **binary crossentropy**) for finding the optimal betas for logistic regression:

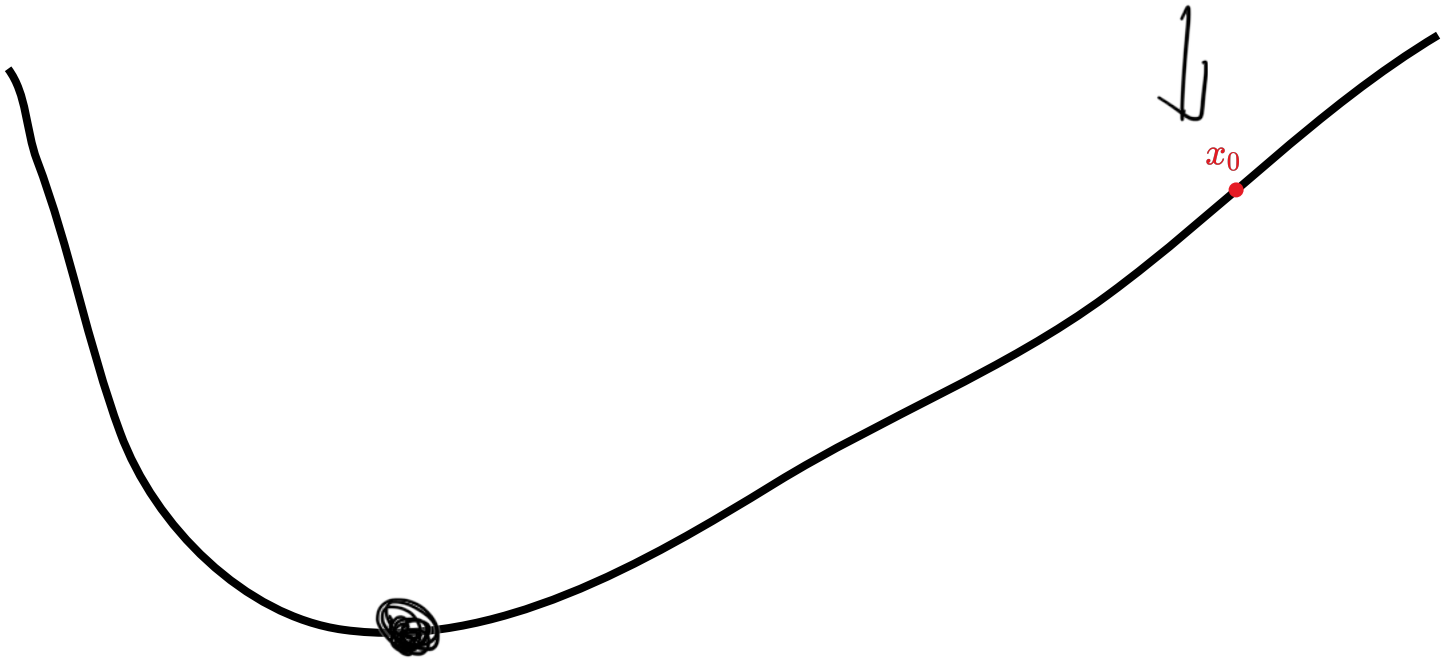
$$\text{minimize} \quad \sum \left[-y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i) \right]$$

Optimizing loss functions with gradient descent

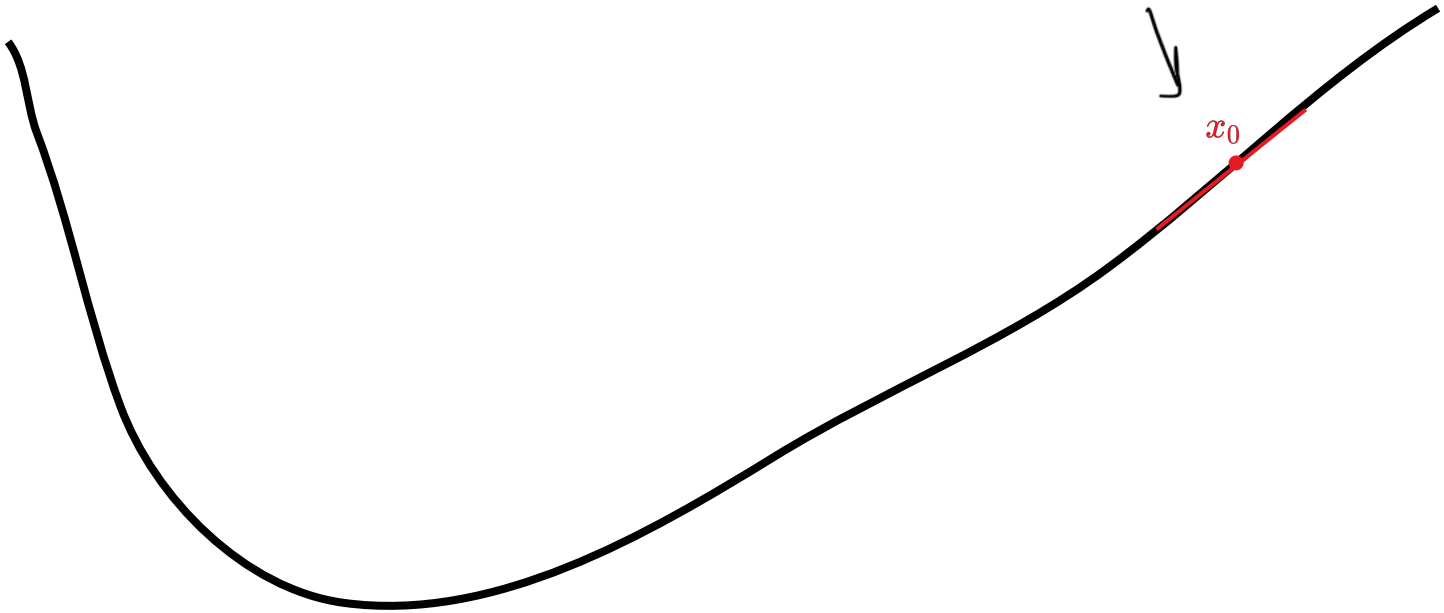
Gradient Descent



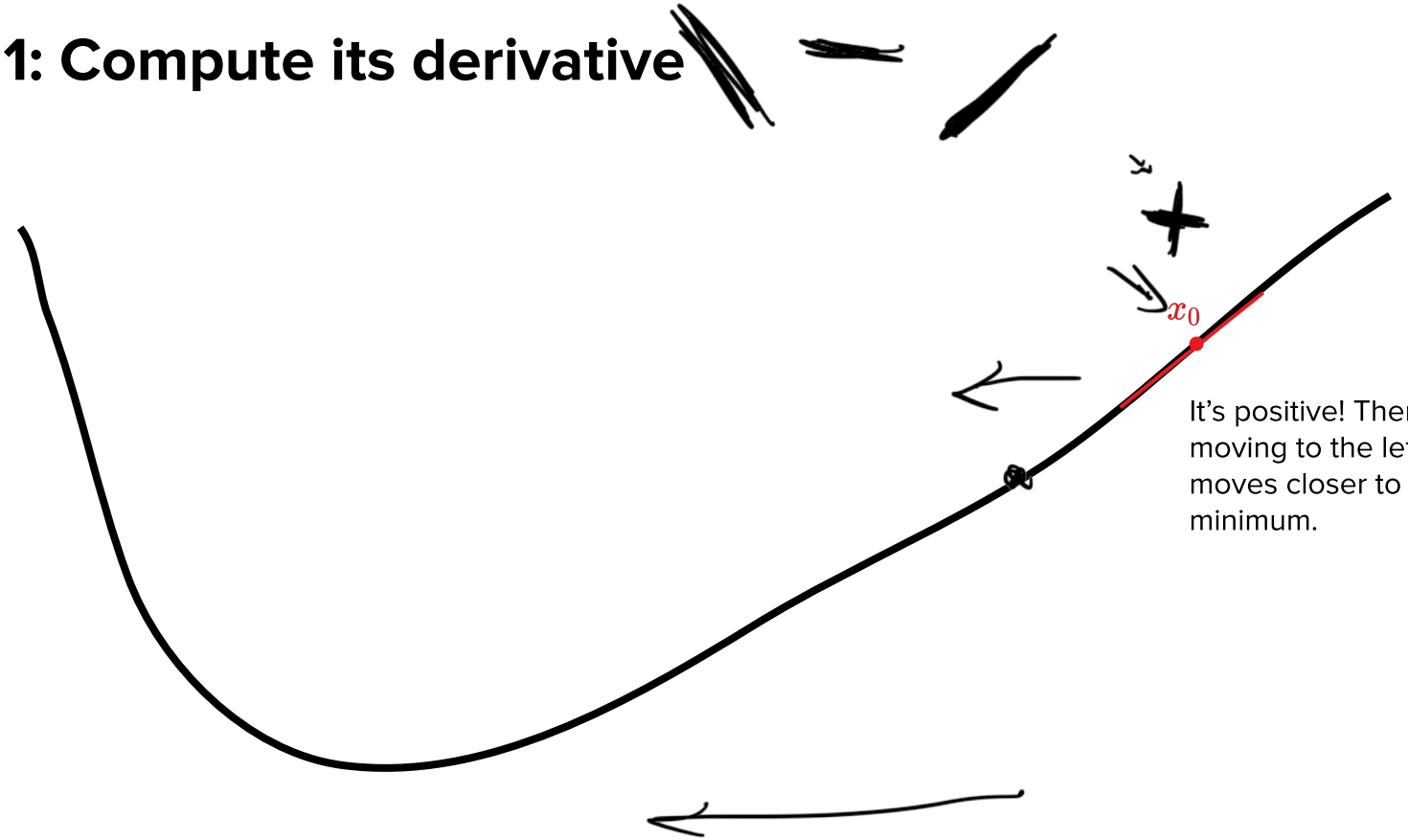
Step 0: Start at a random point



Step 1: Compute its derivative

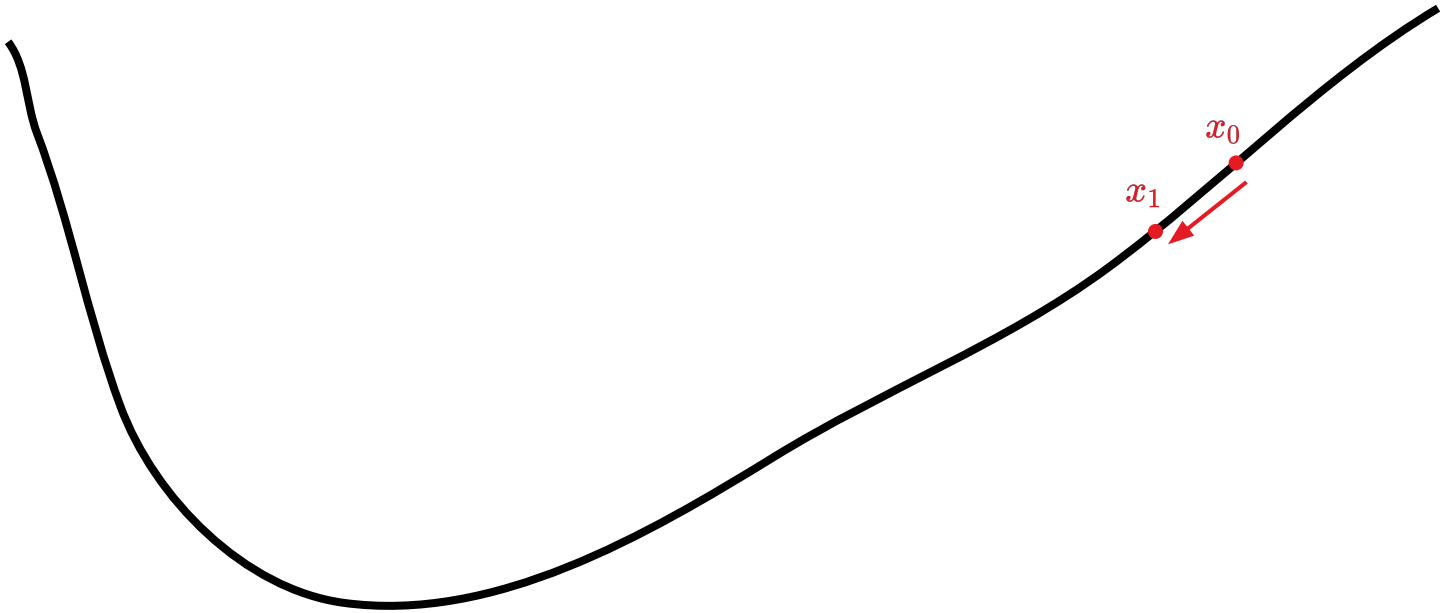


Step 1: Compute its derivative

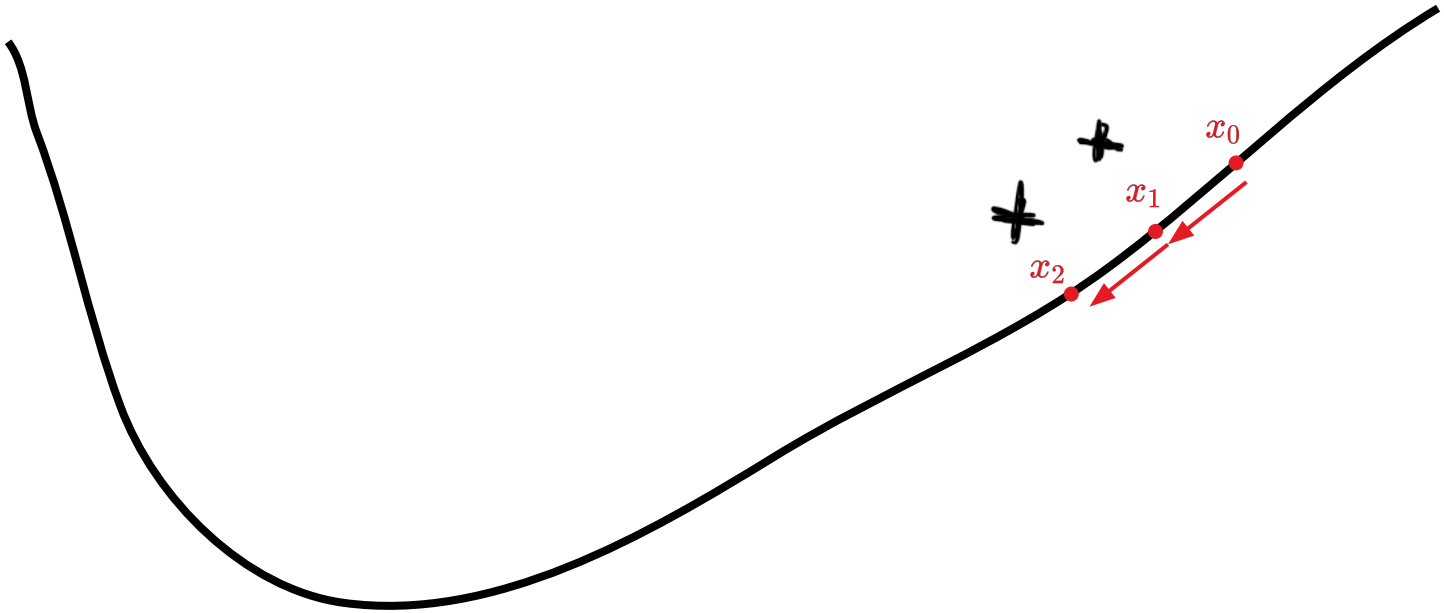


It's positive! Therefore, moving to the left moves closer to the minimum.

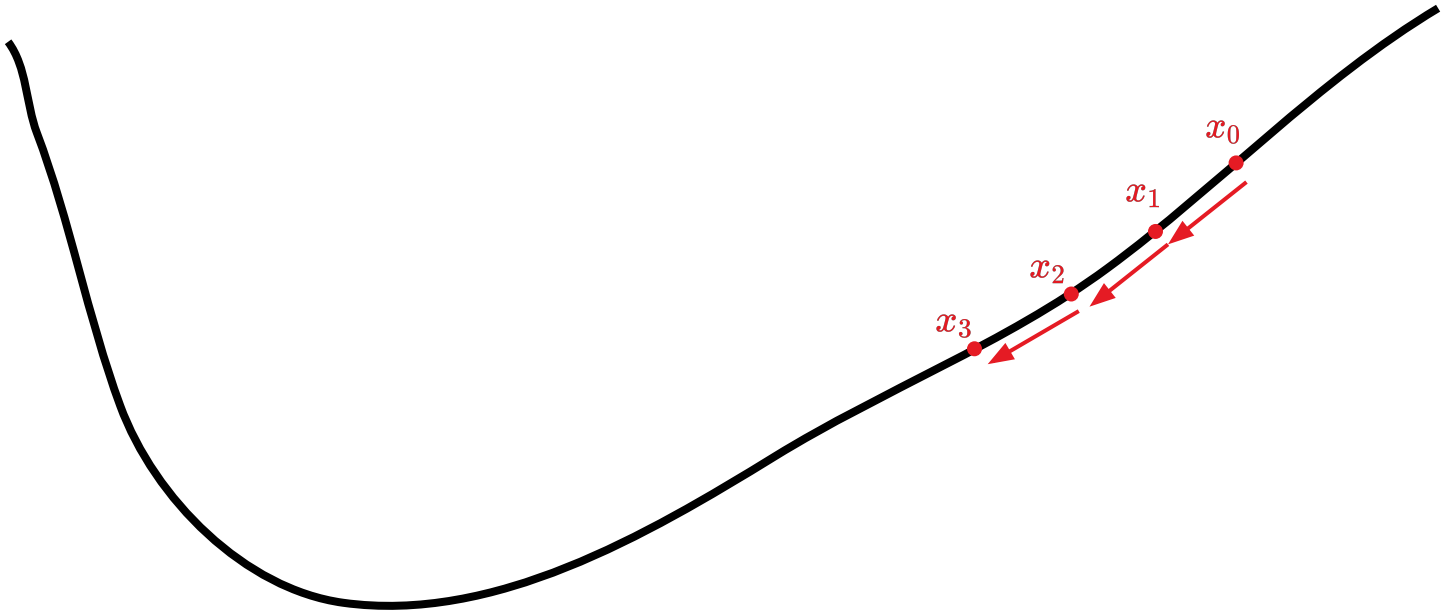
Step 2: Move in the appropriate direction



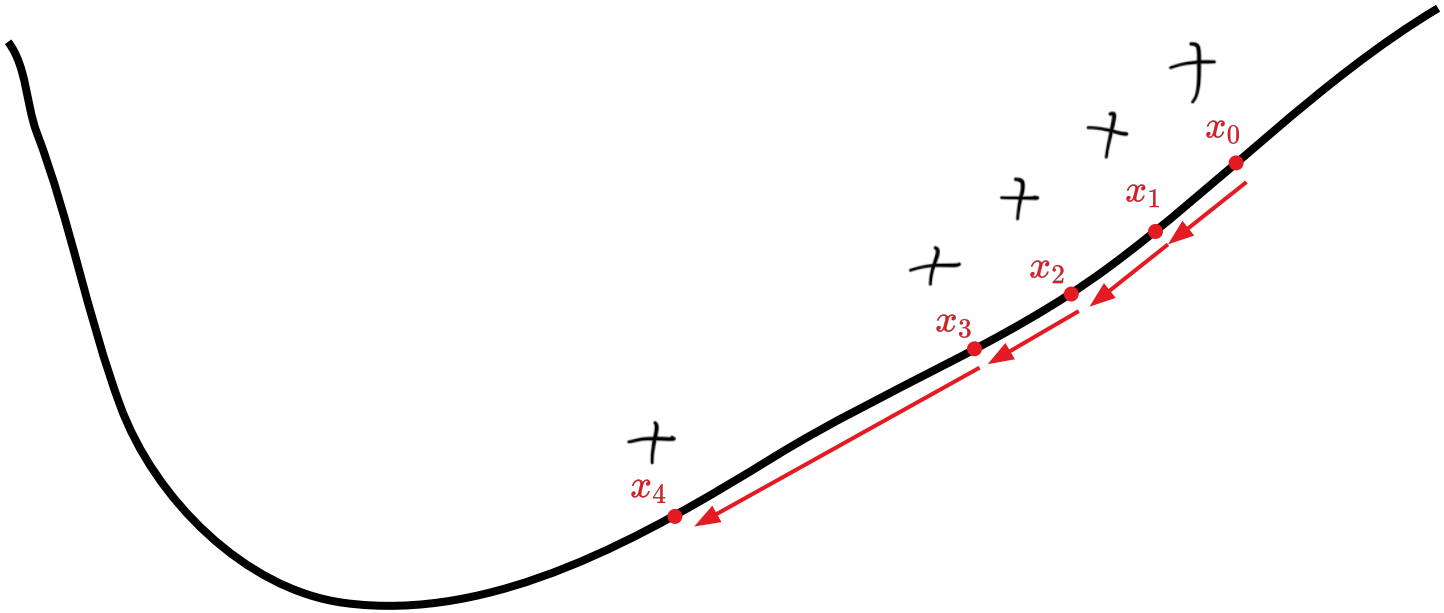
Step 3: Keep repeating steps 1 and 2



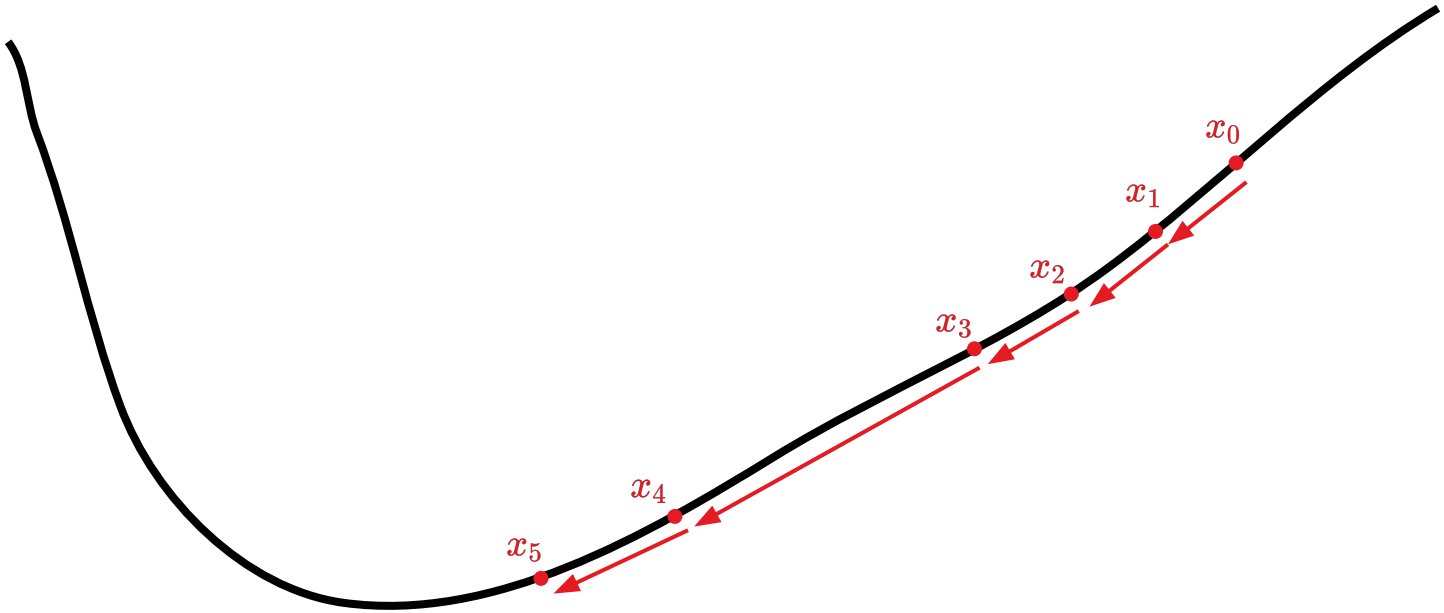
Step 3: Keep repeating steps 1 and 2



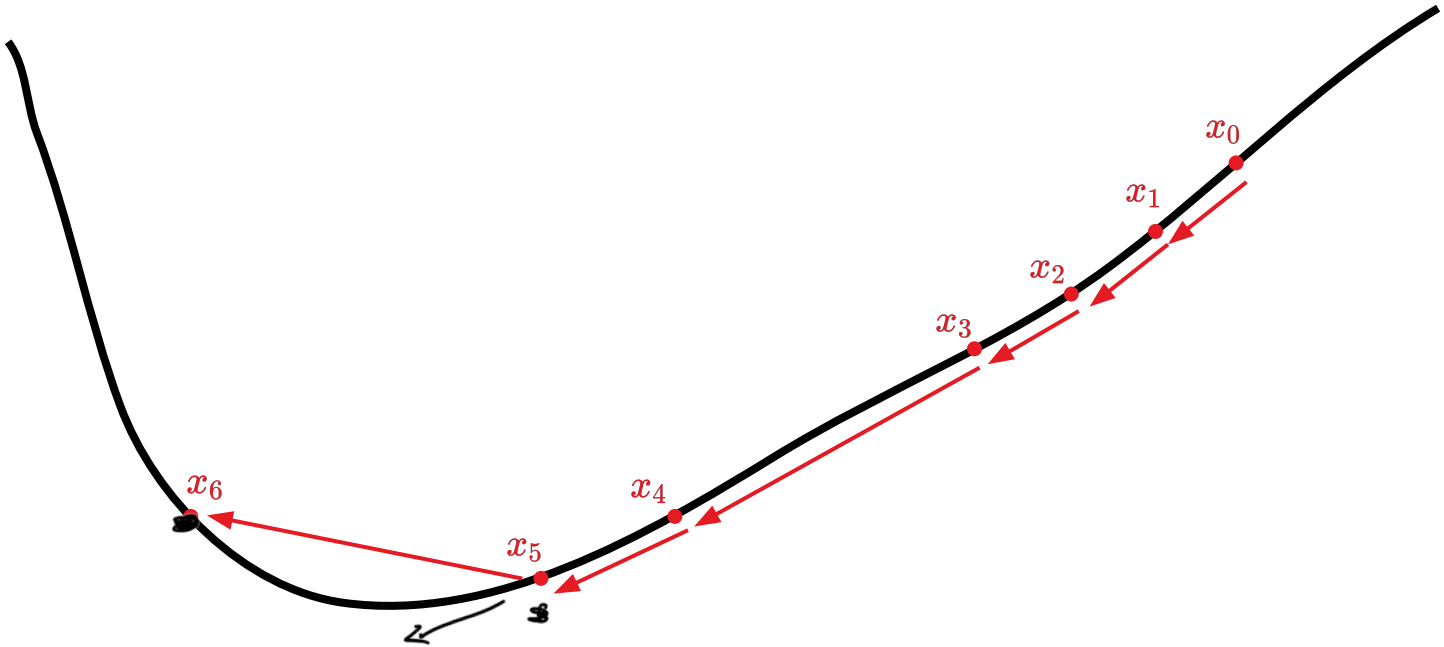
Step 3: Keep repeating steps 1 and 2



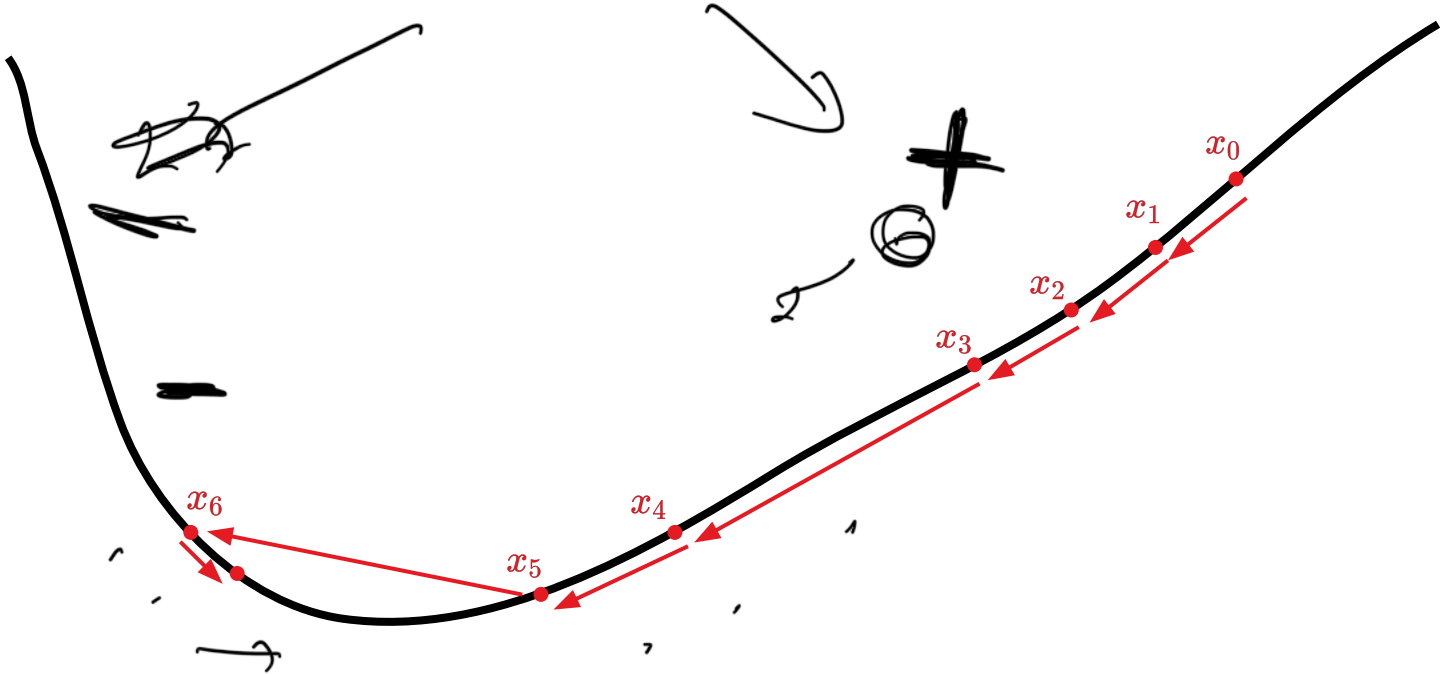
Step 3: Keep repeating steps 1 and 2



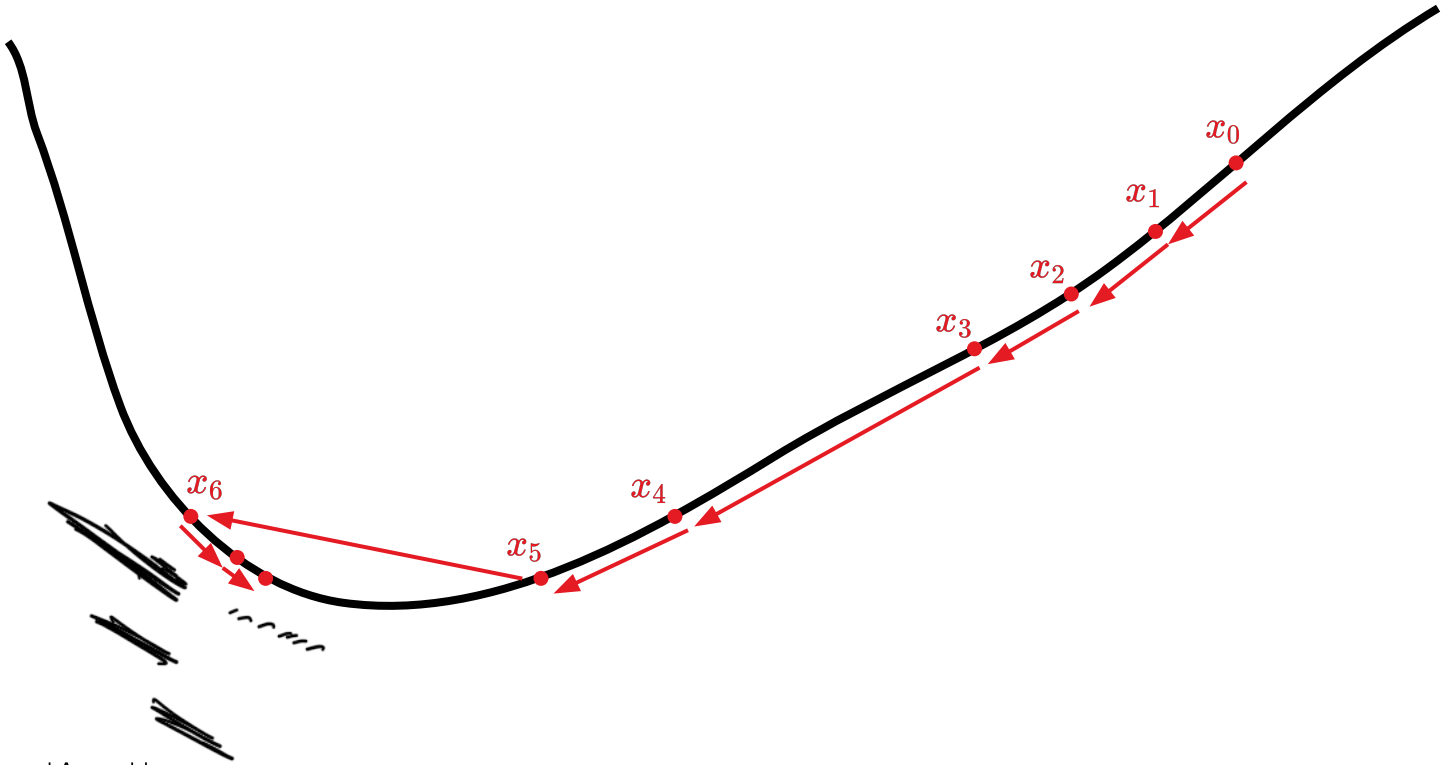
Step 3: Keep repeating steps 1 and 2



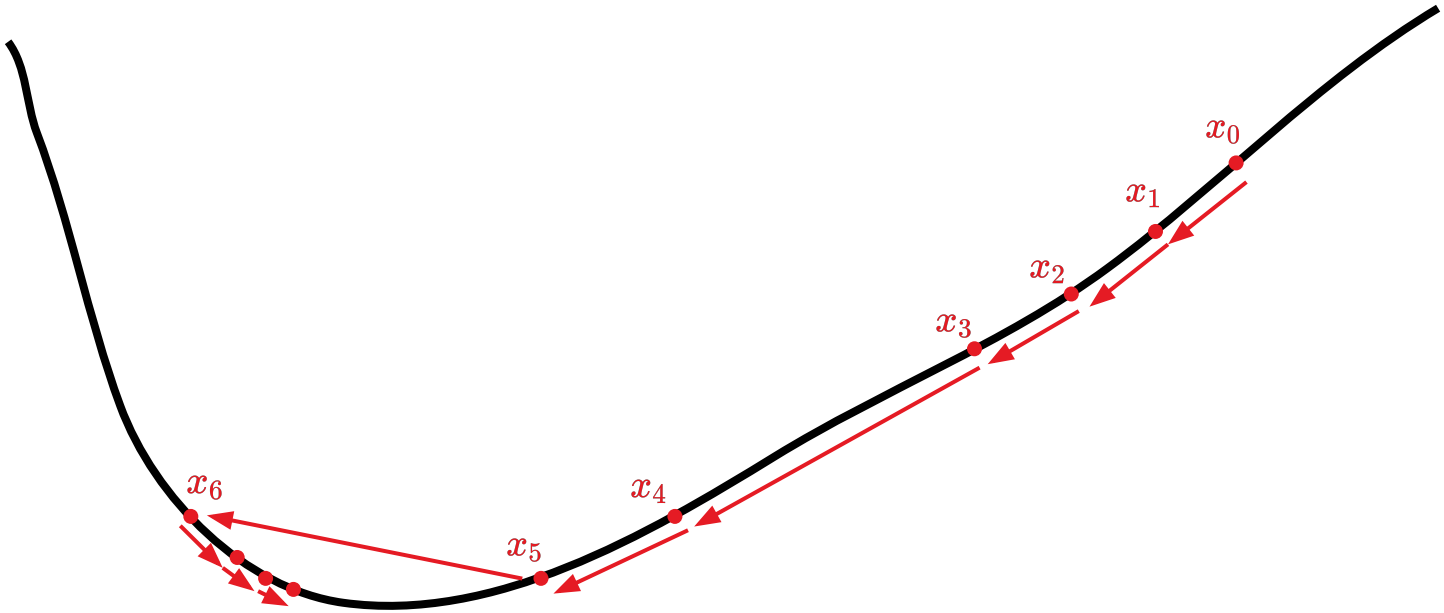
Step 3: Keep repeating steps 1 and 2



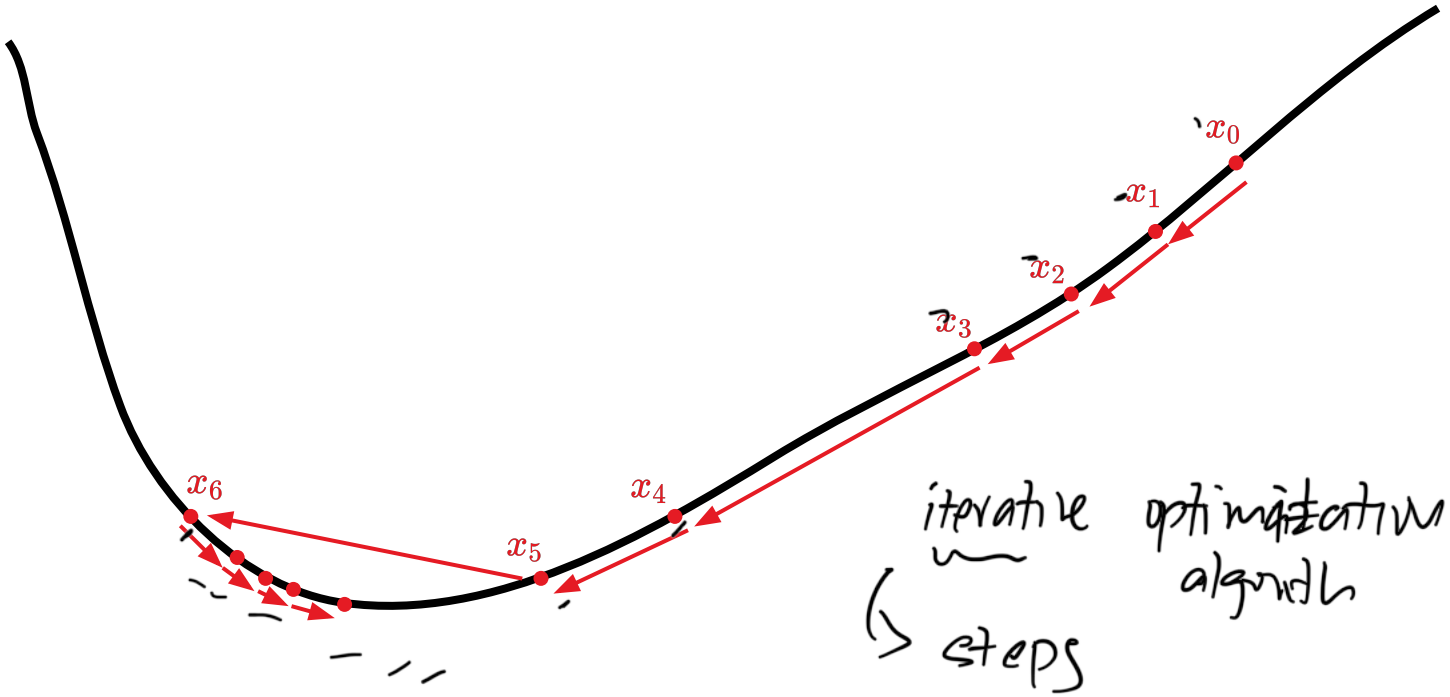
Step 3: Keep repeating steps 1 and 2



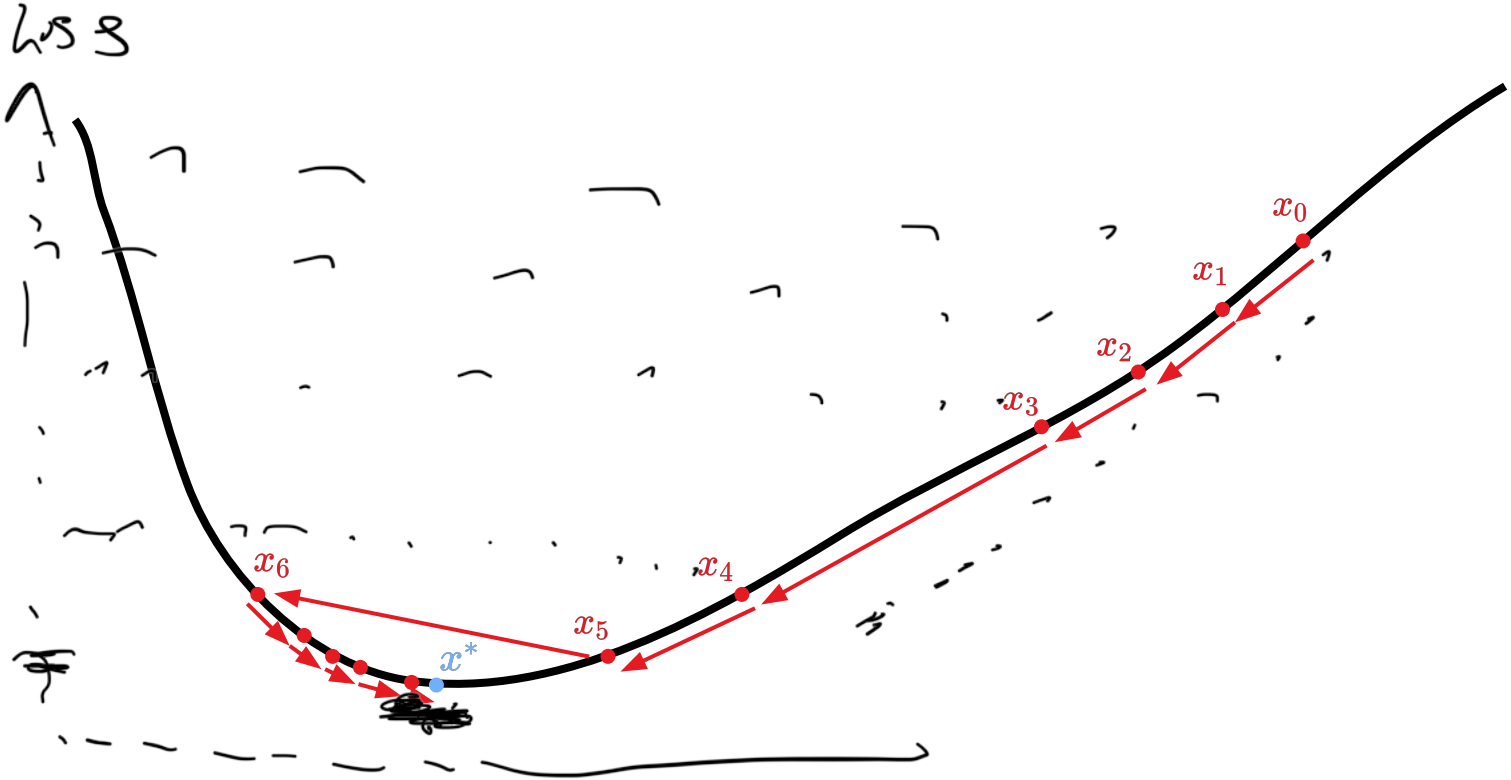
Step 3: Keep repeating steps 1 and 2



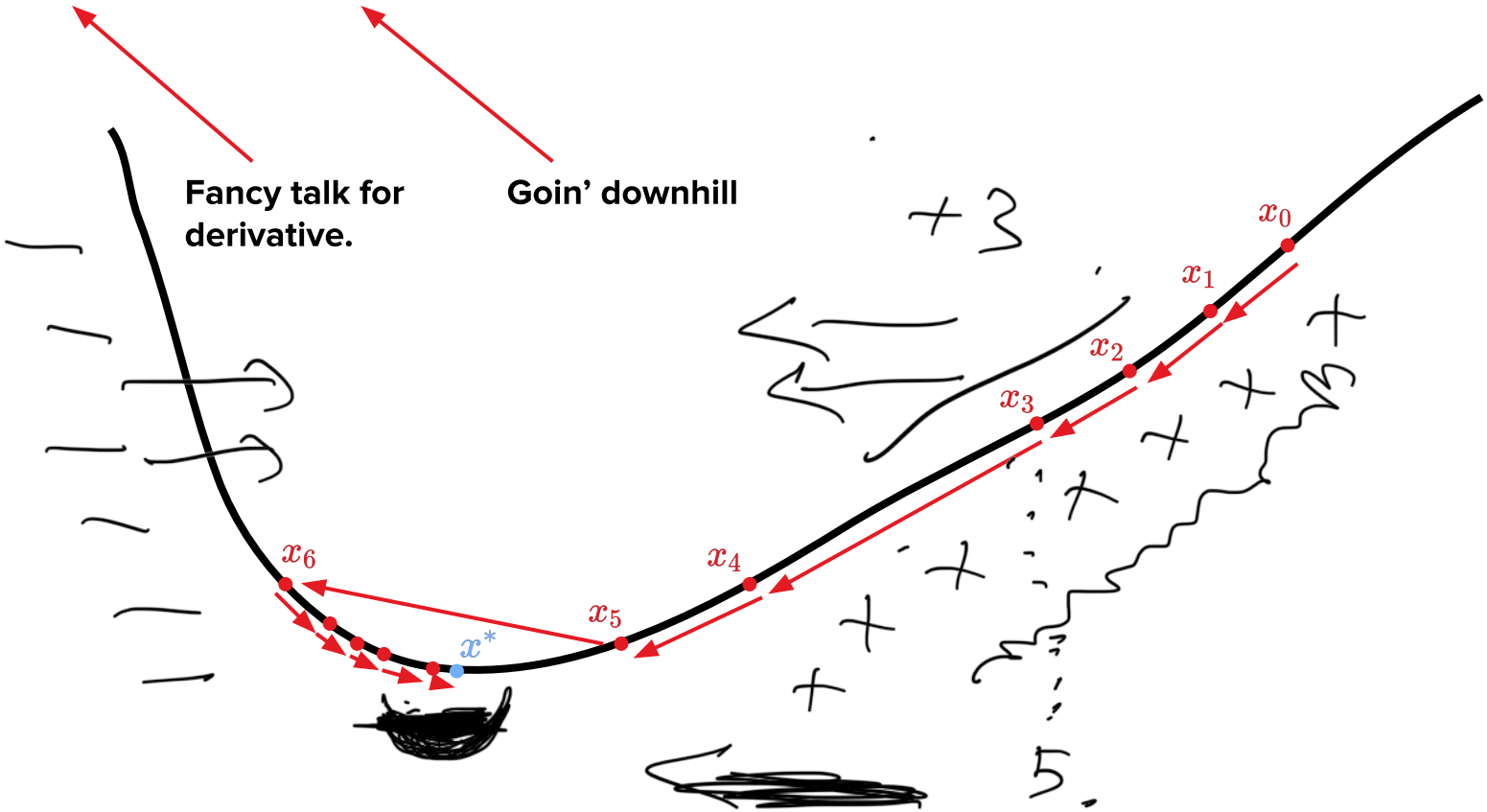
Step 3: Keep repeating steps 1 and 2



Step 4: Stop when your move is small enough



Gradient Descent



Gradient Descent

So how does it work, mathematically? There are a few considerations:

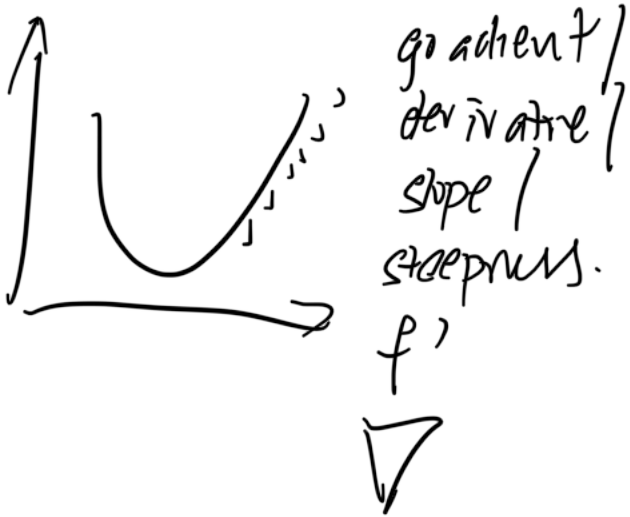
- We'll be "updating" our current position over and over again
- We need to move in the opposite direction of the derivative
- We need to decide how far to move at each step

Gradient Descent

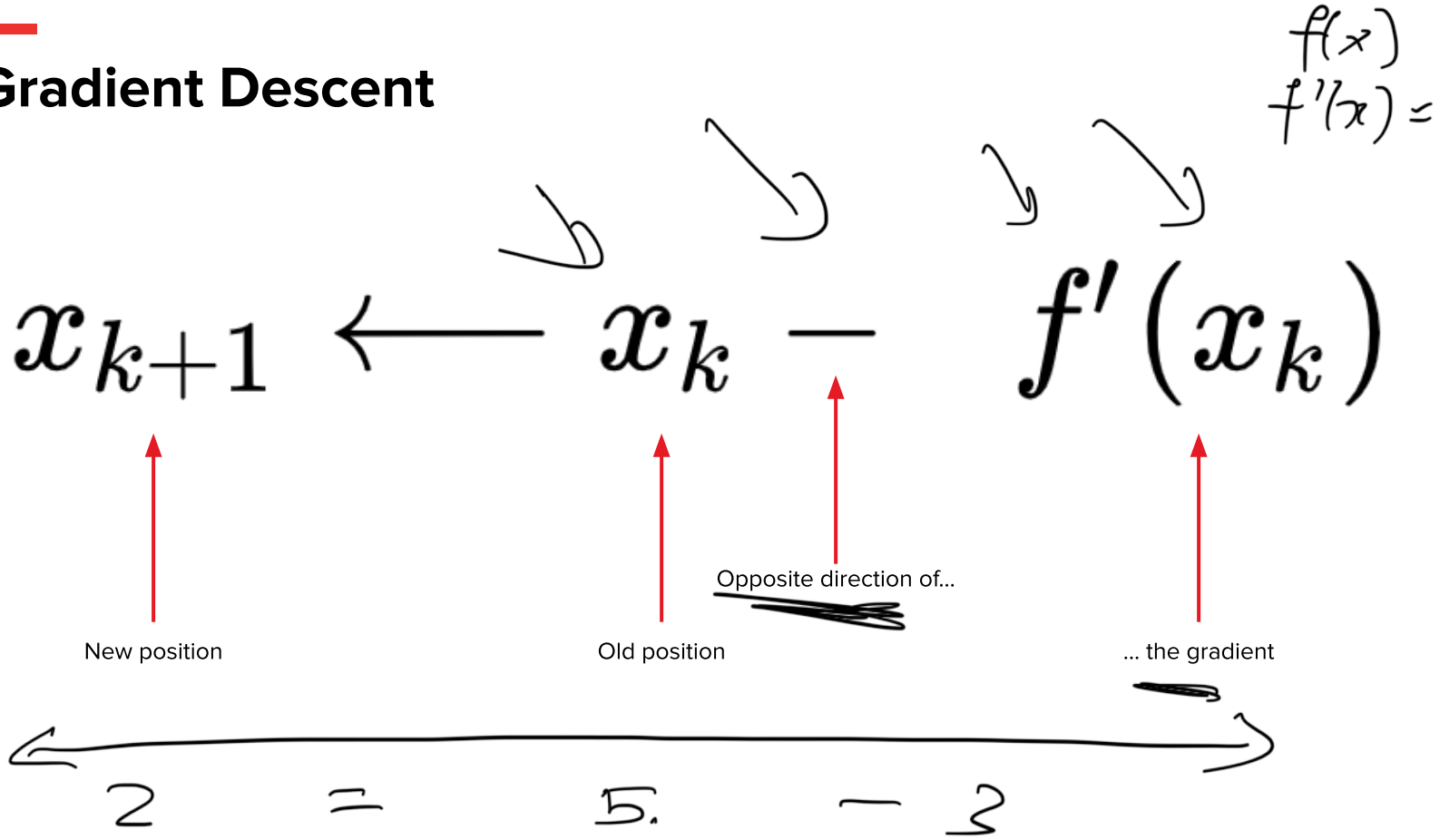
$$x_{k+1} \leftarrow x_k$$

New position

Old position



Gradient Descent



Gradient Descent

$$x_{k+1} \longleftarrow x_k = \alpha f'(x_k)$$

Step size

New position

Old position

Opposite direction of...

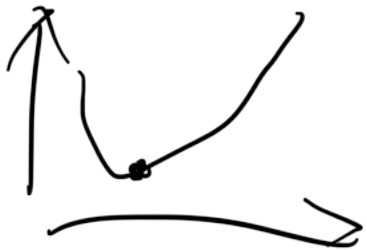
... the gradient

What can go wrong?

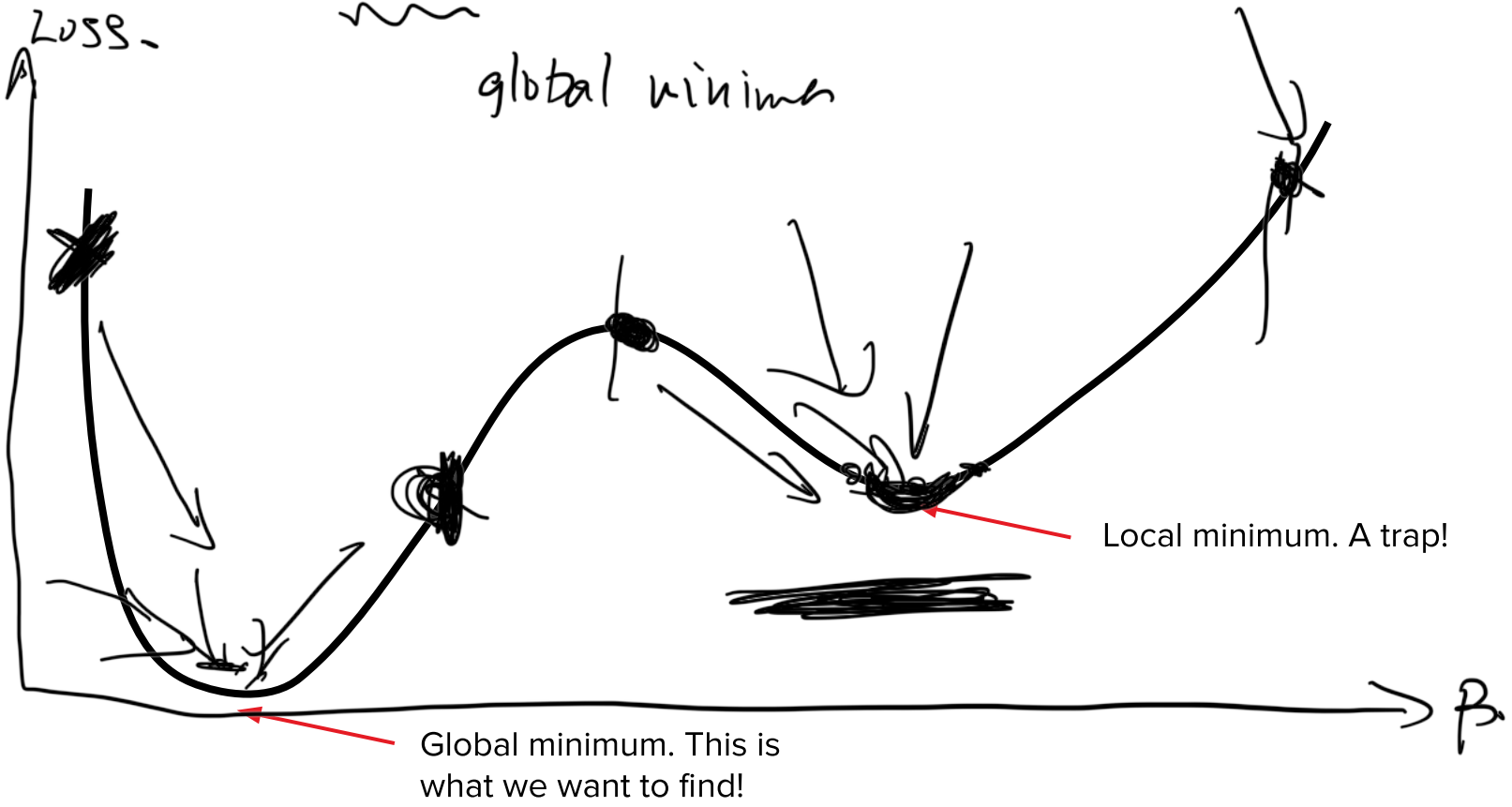
What can go wrong?

While it's difficult to grasp at first, gradient descent is actually a very simple algorithm. There's really only one component that's difficult to calculate, and the rest is easy.

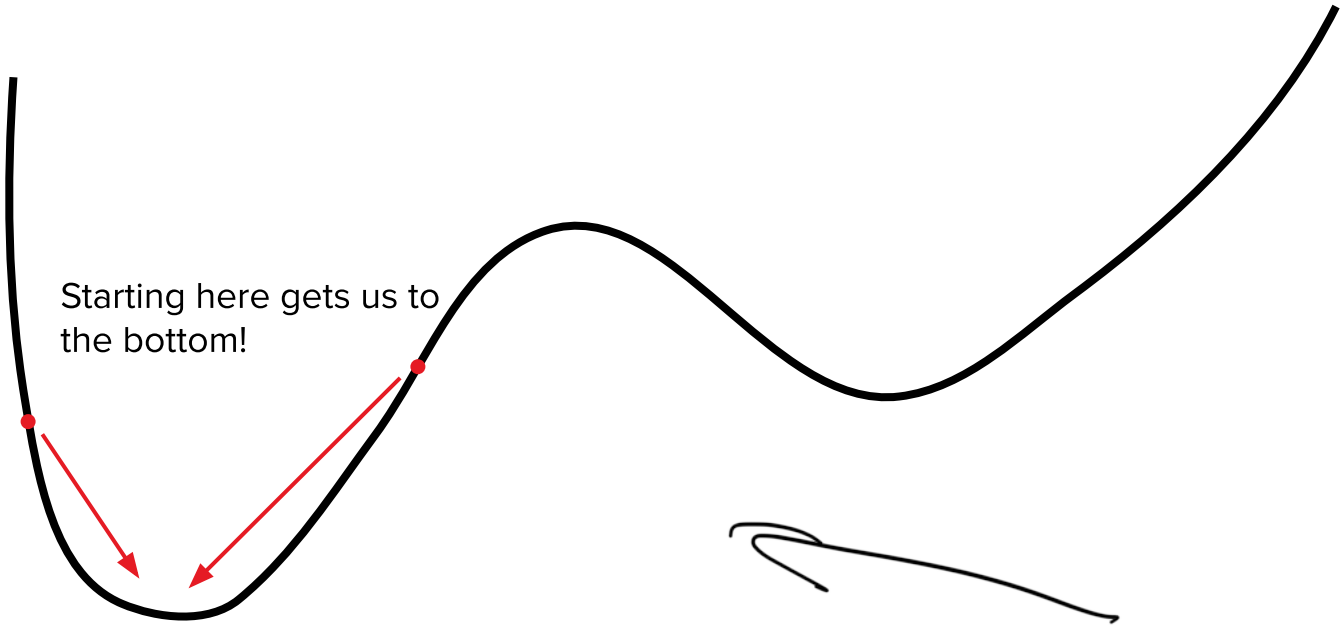
But if it were *really* easy, why make such a big fuss?



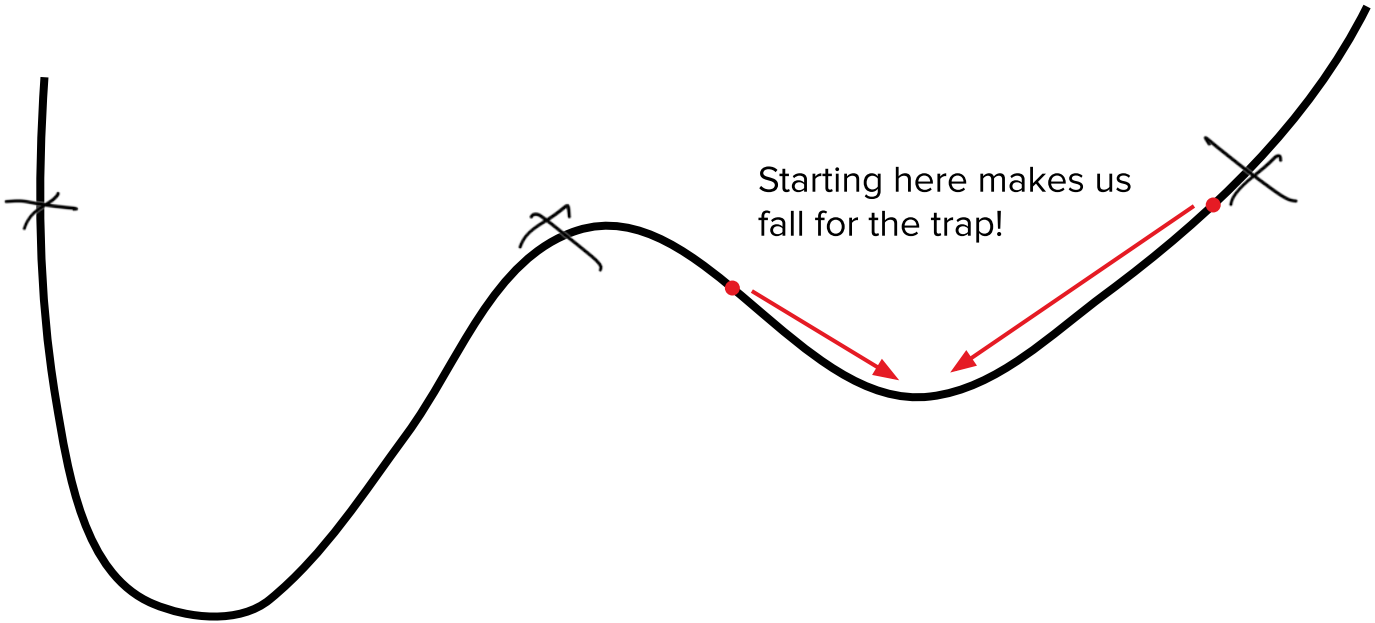
Problem #1: Local Minima



Problem #1: Local Minima



Problem #1: Local Minima



Problem #1: Local Minima

The solution to this problem is to verify that our objective function is convex.

A function is **convex** if it is perfectly U-shaped.

Want a more formal definition?

Convexity

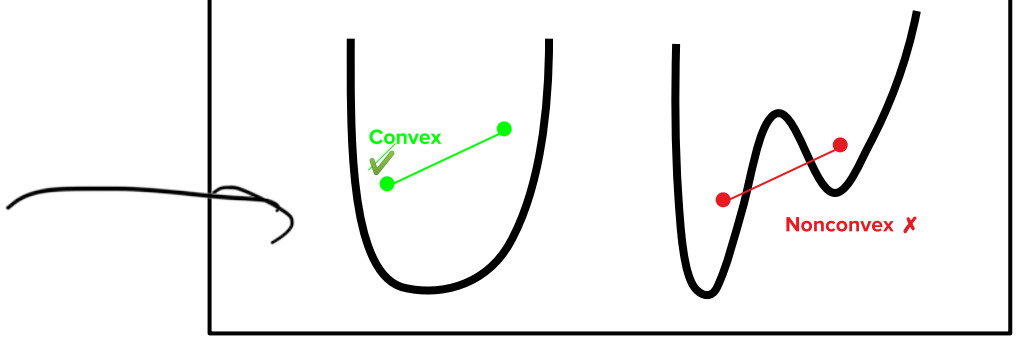
A function is **convex** if the two equivalent definitions are true:

Definition #1

$$f''(x) \geq 0 \text{ for all } x$$

Definition #2

The straight line between any two points above a function are entirely inside that function.

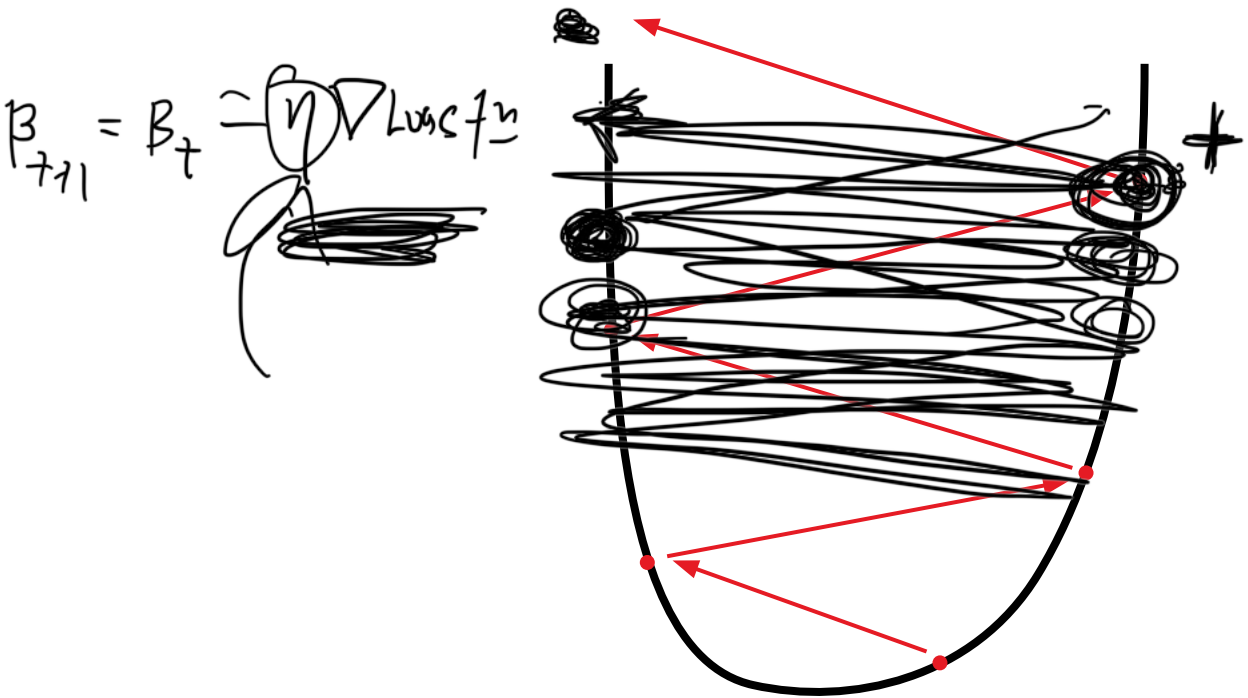


Convexity

Thankfully, all* loss functions in our course are convex!

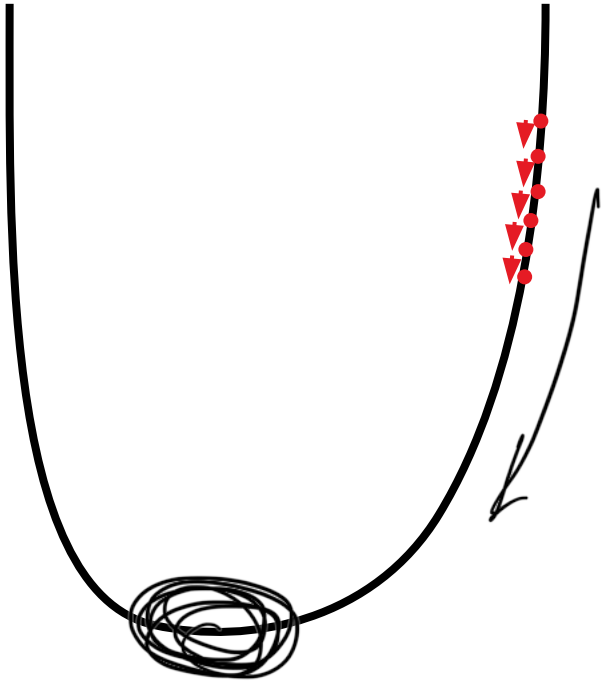
*except the loss functions of neural networks. Always the exception...

Problem #2: α too large

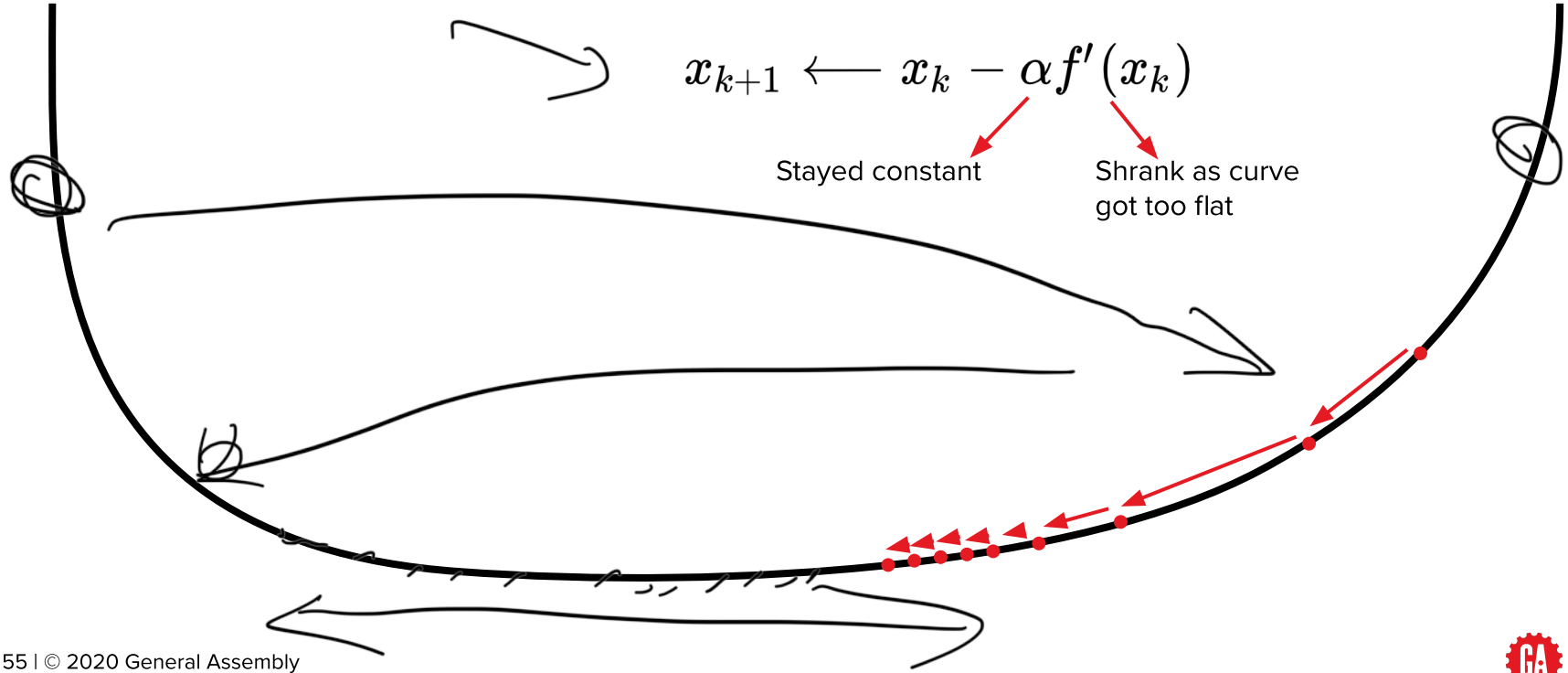


Problem #3: α too small

convergence



Problem #4: Curve is too flat



So then, how do we choose α?

Choice of α

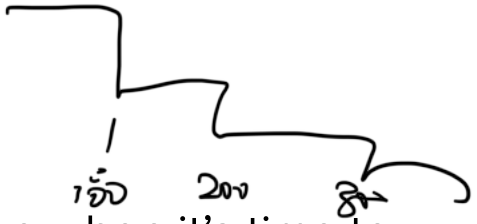
Choosing the best value for our step size α is probably the most difficult problem to solve. Here are a few ways:

Method #1: Manually

We can choose α manually, and then using guess-and-check to iterate the process until we get an α that converges.

It might not sound scientific, but it's what we'll do today. It's also what we'll do for neural networks, since other methods will save us from our folly.

Method #2: Let α shrink over time



This is sensible, since we don't want α being too large when it's time to converge. The following choices are common:

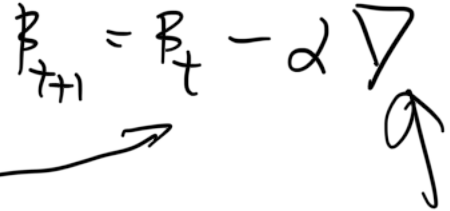
$$\alpha_k = \frac{c}{k} \text{ or } \frac{c}{\sqrt{k}}$$

While it's a good try, this rarely works in practice and adjusting c just amounts to doing guess-and-check again.

Adam

RMSprop

Method #3: Use a fancier algorithm



The version of gradient descent we've learned today is overly simple for today's world, but it still gets to the gist of what we're trying to learn. In practice, more advanced methods are used, for example:

- Iteratively Reweighted Least Squares (IRLS)
- Adaptive Momentum (Adam)
- RMS Propagation (RMSProp)

Method #3: Use a fancier algorithm

The version of gradient descent we've learned today is overly simple for today's world, but it still gets to the gist of what we're trying to learn. In practice, more advanced methods are used, for example:

- Iteratively Reweighted Least Squares (IRLS) ← **GLMs use this!**
- Adaptive Momentum (Adam)
- RMS Propagation (RMSProp)

Neural networks often use these!

Method #4: Second-order methods

On that note, what we've been doing is actually **first-order gradient descent** since it only uses the first derivative. Second-order gradient descent methods exist, the most common of which is known as **Newton's method** (yes, that Newton!)

Newton's Method

Instead of using a step size α, we'll use:

$$x_{k+1} \longleftarrow x_k - \frac{f'(x_k)}{f''(x_k)}$$

Newton's Method

Instead of using a step size α, we'll use:

$$x_{k+1} \longleftarrow x_k - \frac{f'(x_k)}{f''(x_k)}$$

Not only does this fix all of our problems, but under most conditions, it's *guaranteed to work, and to work faster than first order methods!*

... so what's the catch?

Newton's Method

$$x_{k+1} \longleftarrow x_k - \frac{f'(x_k)}{f''(x_k)}$$

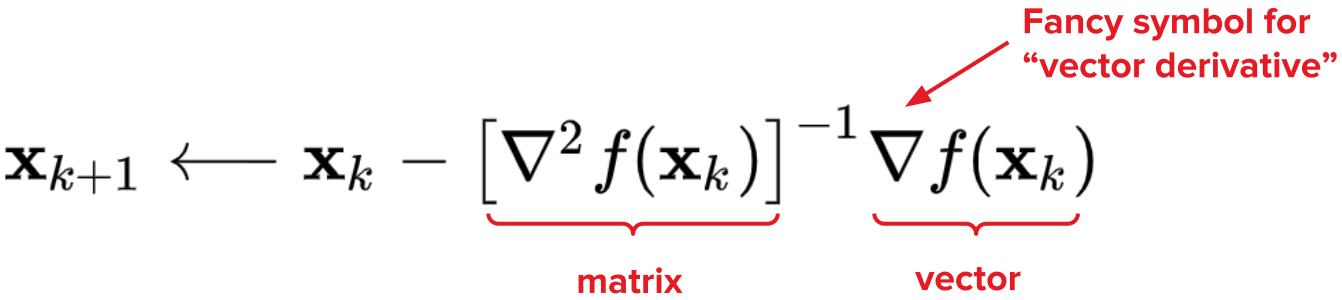
Turns out, second derivatives are **really** hard to calculate. Everything we've done today looks simple, but most problems are way harder to write out. Derivatives need to be estimated by computers - a whole other field of study!

Furthermore, most optimization problems work with vector-valued *x*s, which makes this orders of magnitude more complicated!

Vector-valued Newton's method

$$\mathbf{x}_{k+1} \longleftarrow \mathbf{x}_k - \left[\nabla^2 f(\mathbf{x}_k)\right]^{-1} \nabla f(\mathbf{x}_k)$$

Vector-valued Newton's method



That inverse is the tricky part. It's almost always computationally infeasible to find it. You're better off with method #3 at that point!

Food for thought during our break

There are always students who don't believe me that there are simple one-variable problems that can't be optimized using calculus. If you are in that boat, convince yourself with this exercise! I'll even give you the derivative. We'll solve it with Python after the break.

$$f(x) = -\frac{\log x}{1+x}$$

$$f'(x) = -\frac{1/x+1-\log x}{(1+x)^2}$$

