Statistical (Machine) Learning

Lecture 10 — Optimization basics

Pierpaolo Brutti

Optimization for ML

Act 1 of 2: Uncontrained Optimization

Optimization basics

- The function we're trying to optimize is the **objective function** generically denoted by $M(\cdot)$
- The argument to $M(\cdot)$ is θ
  - Some people call this the **optimand**
- The possible values of θ is Θ, the **domain** or **feasible set**, whose dimension is p
- **Optimization** can be minimization or maximization, as we like; we'll stick with *minimizing*

Local vs Global minima

- θ is a **global minimum** when

$$\theta' \neq \theta \quad \Rightarrow \quad M(\theta') \geqslant M(\theta) \quad \text{not necessarily unique!}$$

- θ is a **local minimum** when $M(\theta) \leqslant M(\theta')$ whenever θ' is *close enough* to θ
  - Every global minimum is also a local minimum
  - If there's only one local minimum anywhere, it's the global minimum
- Lots of local minima tend to make it harder to find the global minimum

At "the" minimum: value vs location

- If $\theta^\star$ is a global minimum, then $M(\theta^\star)$ is the **value** of the minimum or minimal value, in symbols

$$\min_{\theta \in \Theta} M(\theta)$$

- But $\theta^\star$ itself is the **location** of the global minimum, in symbols

$$\operatorname{argmin}_{\theta \in \Theta} M(\theta)$$

- **Example**: the minimal value of $(x-1)^2$ is 0, but the location of the minimum is $x = 1$
- Both value and location can change with $\Theta \rightsquigarrow$ important when we look at **constraints** later

Finding the minimum: optimization algorithms

- An optimization algorithm starts from M and Θ, and possibly an initial guess $\theta^{(0)}$, and returns an **approximation** to $\operatorname{argmin}_{\theta \in \Theta} M(\theta)$, call it θ_{out}
- We usually measure the approximation by difference in the *value*, not the location: the algorithm gets ϵ-close when

$$M(\theta_{\text{out}}) \leqslant \epsilon + \min_{\theta \in \Theta} M(\theta)$$

- Often, the longer we let the algorithm run, the better the approximation
  - How many steps does the algorithm need to get ϵ-close to the optium?
    $O(1/\epsilon)$ or $O(\epsilon^{-d})$ would be polynomial, $O(\log 1/\epsilon)$ would be logarithmic, $\mathrm{e}^{O(1/\epsilon)}$ would be exponential (and bad!)
- Going forward, **early stopping** (an optimization algorithm) will be one of many ways to *regularize* our learning algorithms to avoid **overfitting**.

So how do we build an optimization algorithm anyway?

- Start with *calculus*, assume $M(\cdot)$ has as many derivatives with respect to θ as we need
- **1st-order condition**: at an (interior!) local minimum/maximum, or inflection point,

$$\nabla M(\theta) = 0 \quad \rightsquigarrow \quad M \text{ is flat at the minimum}$$

- **2nd-order condition**: at an (interior) local *minimum* only, for any direction $\boldsymbol{v} \neq 0$,

$$\boldsymbol{v} \left(\nabla \nabla M(\theta) \right) \boldsymbol{v} \geqslant 0 \quad (\text{strictly positive in at least some directions})$$

  - $\nabla \nabla M$ is the matrix of 2nd partial derivatives, or **Hessian**, so I will also write $\mathbf{h}(\theta)$
  - This condition is what's meant when we say the Hessian is **non-negative-definite**: $\mathbf{h}(\theta) \succeq 0$
  - **Positive-definite**: $\mathbf{h}(\theta) \succ 0$, means $\boldsymbol{v}\,\mathbf{h}\,\boldsymbol{v} > 0$ for all $\boldsymbol{v} \neq 0$; this implies θ is an *isolated* local minimum
  - **In words**: moving away from the local minimum in any direction can only increase the objective function

Optimizing by equation-solving

- One approach: use the first-order condition to get a system of equations

$$\nabla M(\theta) = 0$$

  - We have one equation per coordinate of θ
  - When M is empirical risk $\widehat{R}$, sometimes called the **estimating equations** or even **normal equations**
- Solve the system of equations for θ
- If there's more than one solution, check the second-order conditions
- This is what we did for estimating linear models by ordinary least squares, or even weighted least squares

Pros and cons of the solve-the-equations approach

- You need to *set up* the system of equations, and often finding ∇M would itself be a pain
  - However, **numerical differentiation** is quite widespread nowaday
- You need to solve a system of equations: good if there are good **solvers** for that type of system of equations, not so good otherwise
  - For **linear systems**, even very old-fashioned methods that go back to Gauss get ϵ approximations with $O(\log 1/\epsilon)$ iterations
  - General-purpose **nonlinear** equation-solving is still much *harder*
    - sometimes works by using Taylor expansion to linearize
    - sometimes works by turning the solve-the-equations into "*minimize the difference between the left and the right hand side of the equation*"

Computer differentiation: a few words (Nocedal and Wright, 2006)

1. Use a **computer algebra system**, such as Mathematica/Wolfram Alpha or Maxima.
   - Works well for relatively simple models $\rightsquigarrow$ the results quite often require "hand simplification"
2. Approximate derivative using **finite difference** $\rightsquigarrow$ always possible but less accurate than other methods

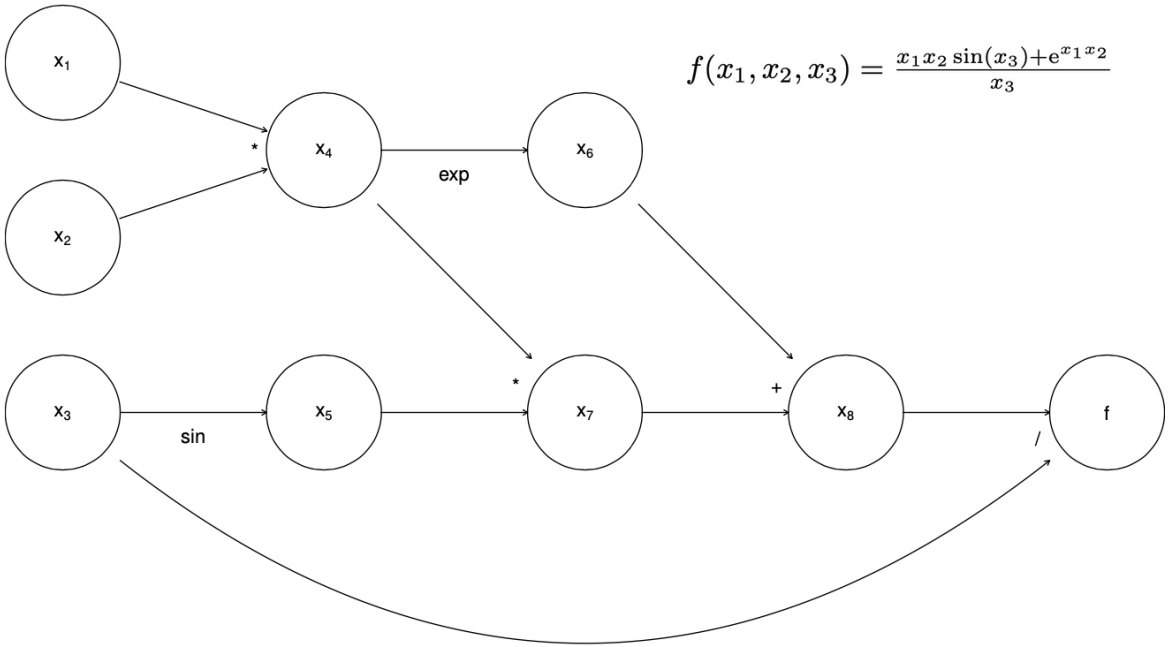
$$\frac{\partial M}{\partial \theta_j} \approx \frac{M(\boldsymbol{\theta} + \Delta \boldsymbol{e}_j) - M(\boldsymbol{\theta})}{\Delta} \quad \texttt{forward differencing}$$

$$\frac{\partial M}{\partial \theta_j} \approx \frac{M(\boldsymbol{\theta} + \Delta \boldsymbol{e}_j) - M(\boldsymbol{\theta} - \Delta \boldsymbol{e}_i)}{2\,\Delta} \quad \texttt{centred differencing}$$

   - *Centred differencing* is typically more accurate but also more costly than *forward differencing*.
   - *Higher order* derivatives can also be approximated similarly.
   - How big should Δ be? As *small* as possible? **Nope**! Because of your computer *finite precision*, to avoid *cancellation error* you have to trade-off!
3. Use **automatic differentiation** (AD), which computes numerically **exact** derivative directly from the code implementing the function to be differentiated, by automatic application of the **chain rule**.
   - As with *finite difference*, there are two main AD *modes*: a *forward*-mode and a *reverse*-mode. The latter comes with the potential for *big* computational savings but *heavy* storage requirements because of the "direction" of the derivative propagation.

Computer differentiation: a few words (Nocedal and Wright, 2006)

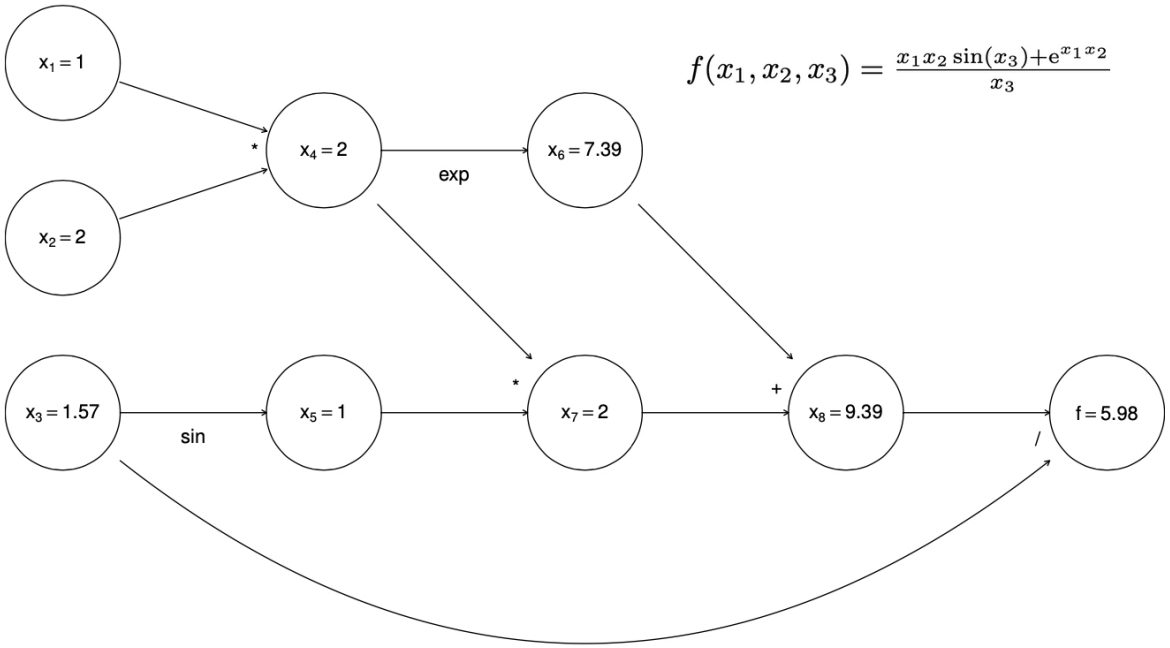
- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - The nodes x_4 and x_8 are intermediate quantities to get to the final answer.
  - The arrows run from *parent* to *child* nodes: **no** child can be evaluated before any of its parents.

Computer differentiation: a few words (Nocedal and Wright, 2006)

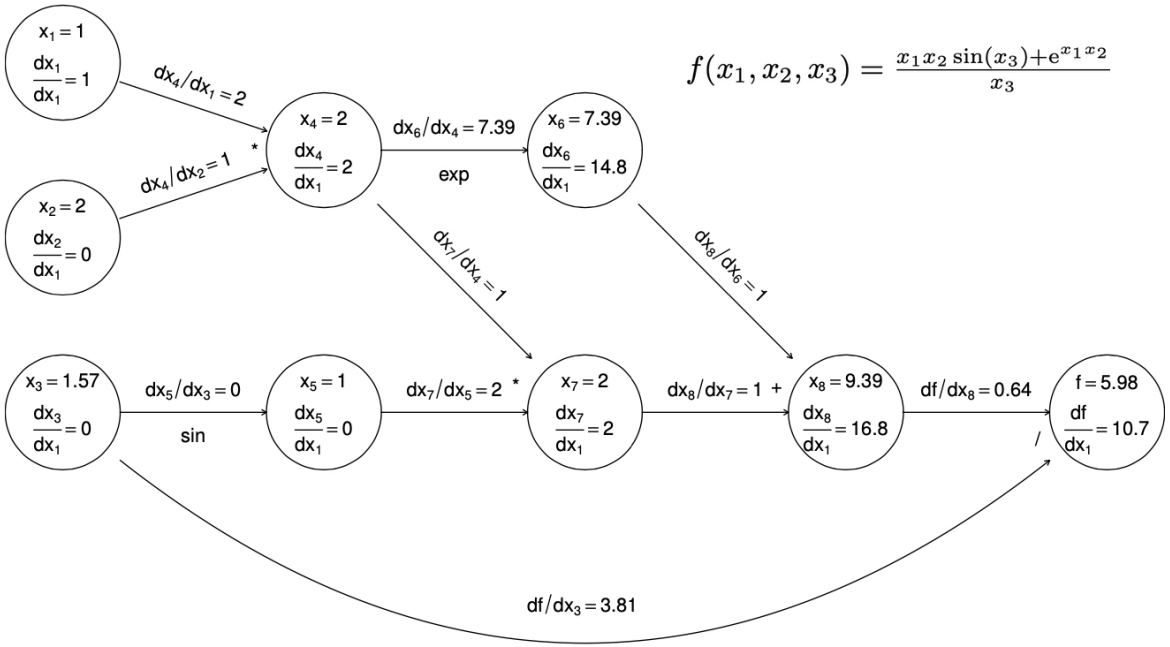
- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - The arrows run from *parent* to *child* nodes: **no** child can be evaluated before any of its parents.
  - An example of simple *left-to-right* evaluation

Computer differentiation: a few words (Nocedal and Wright, 2006)

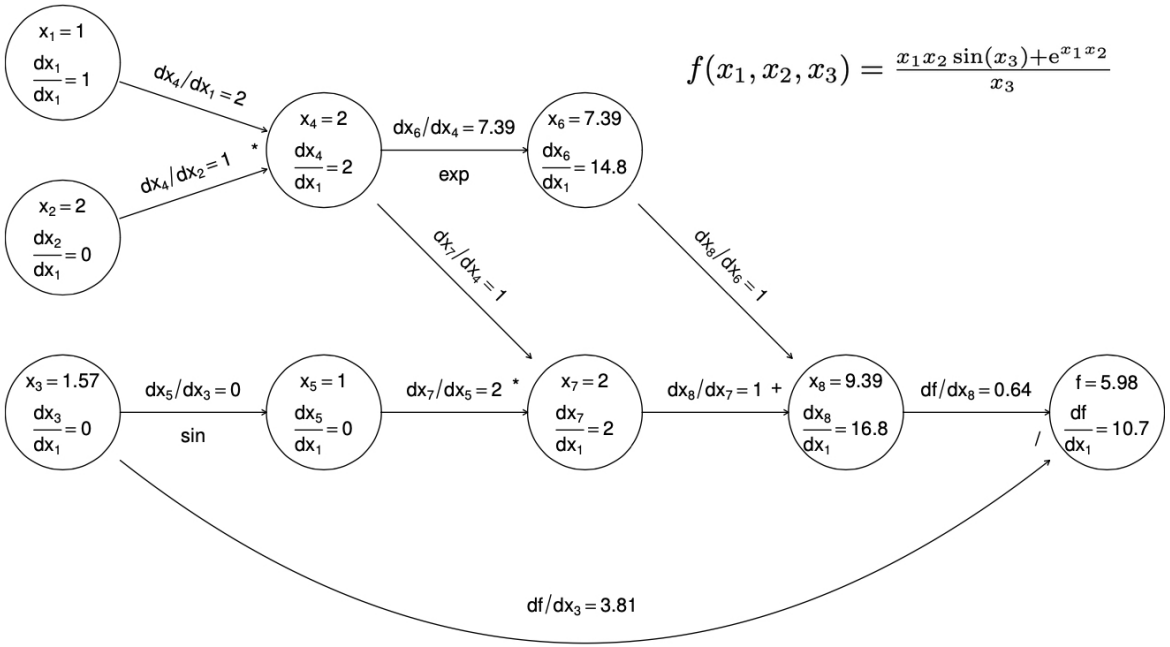
- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - *Forward-mode* carries derivatives forward through the graph alongside values.
  - For example, the derivative of a node w.r.t. x_1 is computed using: $\frac{\partial x_k}{\partial x_1} = \sum_{\{j \text{ parent of } k\}} \frac{\partial x_k}{\partial x_j} \frac{\partial x_j}{\partial x_1}$

Computer differentiation: a few words (Nocedal and Wright, 2006)

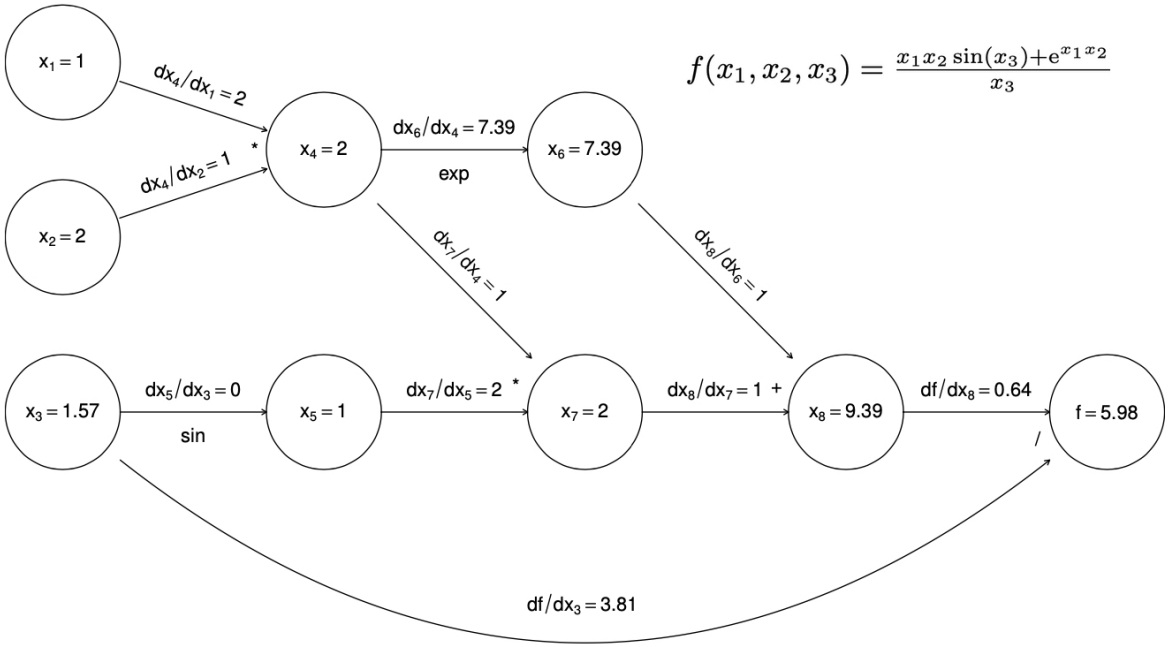
- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - **Pro**: we could discard the values and derivatives of a node *as soon as* all its children are evaluated
  - **Con**: if many derivatives are required, the theoretical computational cost is similar to finite difference (as many operations are required for *each* derivative as are required for function evaluation)

Computer differentiation: a few words (Nocedal and Wright, 2006)

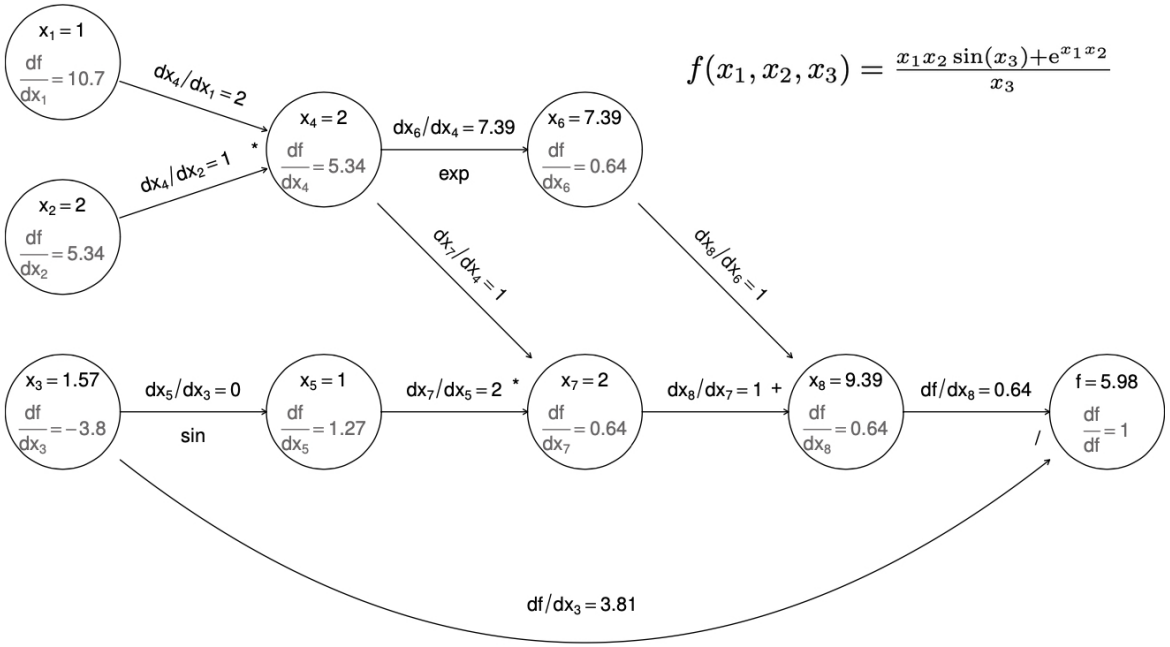
- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - *Reverse-mode* first executed a *forward sweep* evaluating the function and all the derivatives of nodes w.r.t. parents.

Computer differentiation: a few words (Nocedal and Wright, 2006)

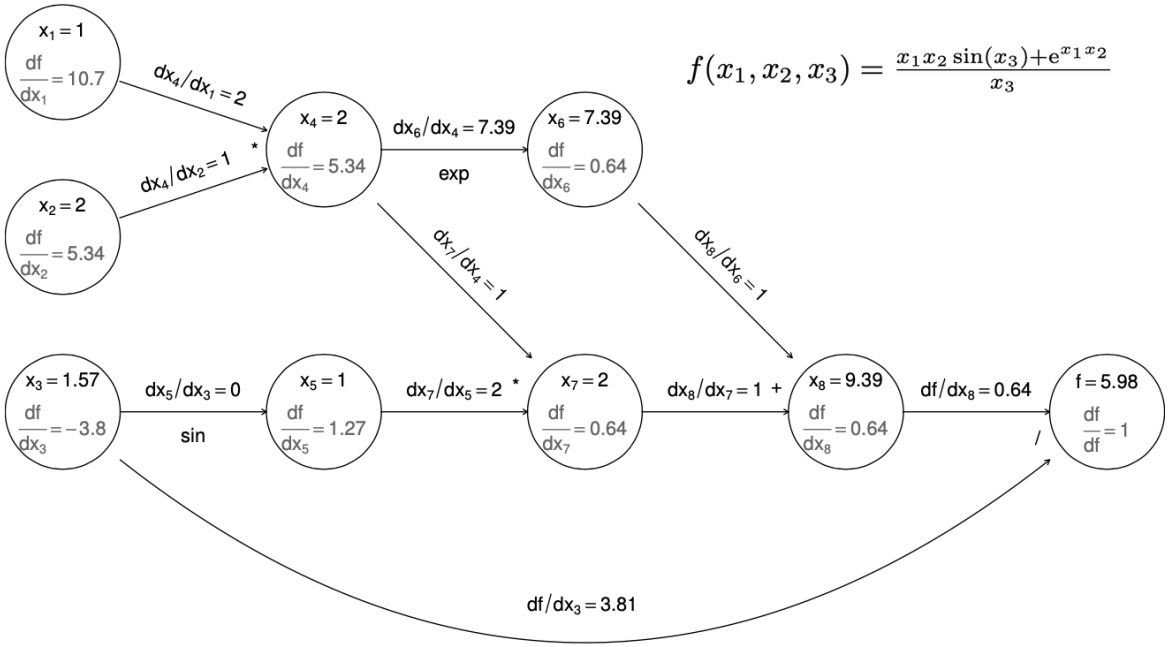
- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - *Reverse-mode* then executed a *reverse sweep* that works backward from the terminal node, where $\partial f / \partial f = 1$, evaluating the derivative of f w.r.t. each node (in gray) using: $\frac{\partial f}{\partial x_k} = \sum_{\{j \text{ is child of } k\}} \frac{\partial x_j}{\partial x_k} \frac{\partial f}{\partial x_j}$

Computer differentiation: a few words (Nocedal and Wright, 2006)

- To better understand what I mean by "*direction of the derivative propagation*", we need to introduce the concept of **computational graph** that details how your computer breaks down the computation into a sequence of elementary operations.



  - **Pro**: now there is only one derivative to be evaluated at each node, but in the end we know the derivative of f w.r.t. every input variable.
  - **Con**: The values of *all nodes* and the evaluated derivatives associated with *every connection* have to be stored during the *forward sweep* in order to be used in the *reverse sweep*.

Computer differentiation: a few packages

$$M(a, x) = \frac{1}{x^2 \sin(a\ x)}$$

```
dx <- D( expression( 1/( x^2*sin(a*x)) ), "x"); dx
```

```
-((2 * x * sin(a * x) + x^2 * (cos(a * x) * a))/(x^2 * sin(a *
    x))^2)
```

$$M(x_1, x_2, x_3) = \frac{x_1 x_2 \sin(x_3) + e^{x_1 x_2}}{x_3}$$

```
# For not overly complex function, we may try <deriv>
M  <- expression( (x1*x2*sin(x3) + exp(x1*x2))/x3)
gr <- deriv(M, c("x1", "x2", "x3"), function.arg = c("x1", "x2", "x3"))
gr(1, 2, pi/2)
```

```
[1] 5.977259
attr(,"gradient")
           x1       x2        x3
[1,] 10.68128 5.340639 -3.805241
```

Go back to the calculus

Gradient Descent

1. Start with a guess $\theta^{(0)}$
2. Find $\nabla M(\theta^{(0)})$
3. Move in the **opposite** direction:

$$\theta^{(1)} = \theta^{(0)} - a_0 \nabla M(\theta^{(0)})$$

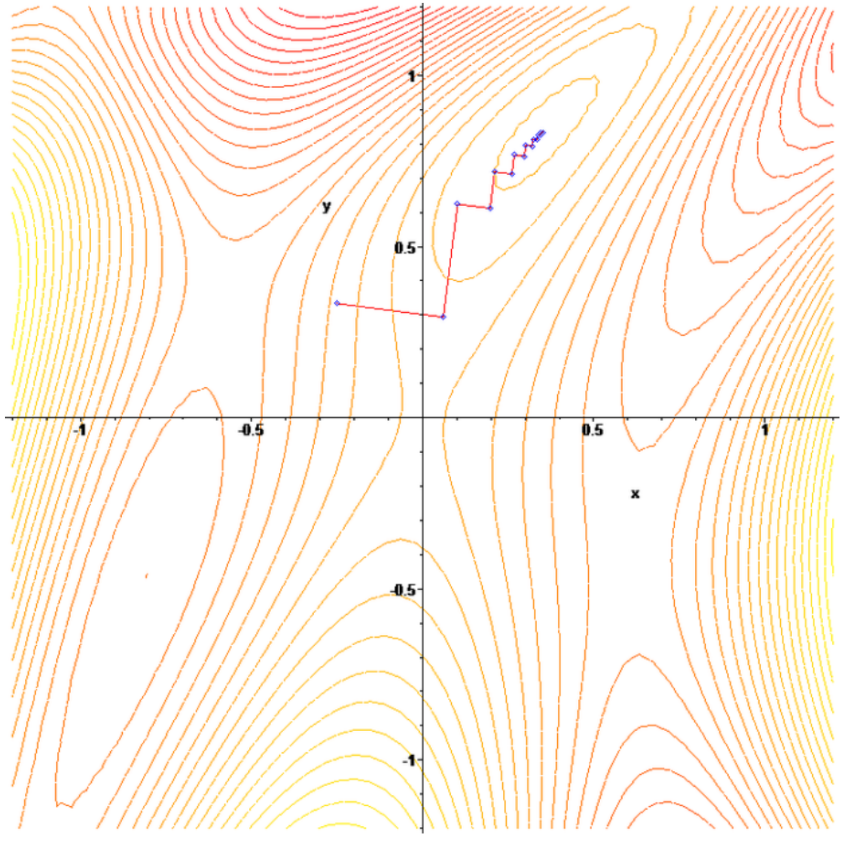
4. Repeat:

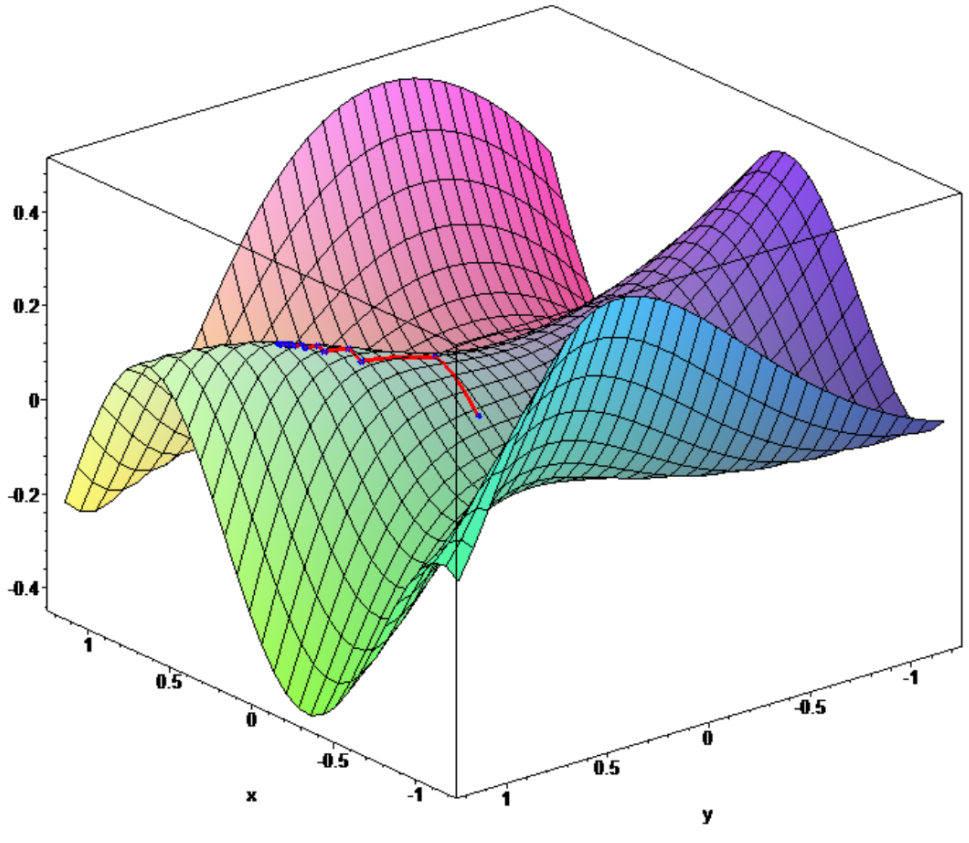
$$\theta^{(t+1)} = \theta^{(t)} - a_t \nabla M(\theta^{(t)})$$

- **Remark**: a *local* optimum will be a **fixed point**
- **Issue**: how big are the **step sizes** (a.k.a. *learning rate*) a_t?

Go back to the calculus

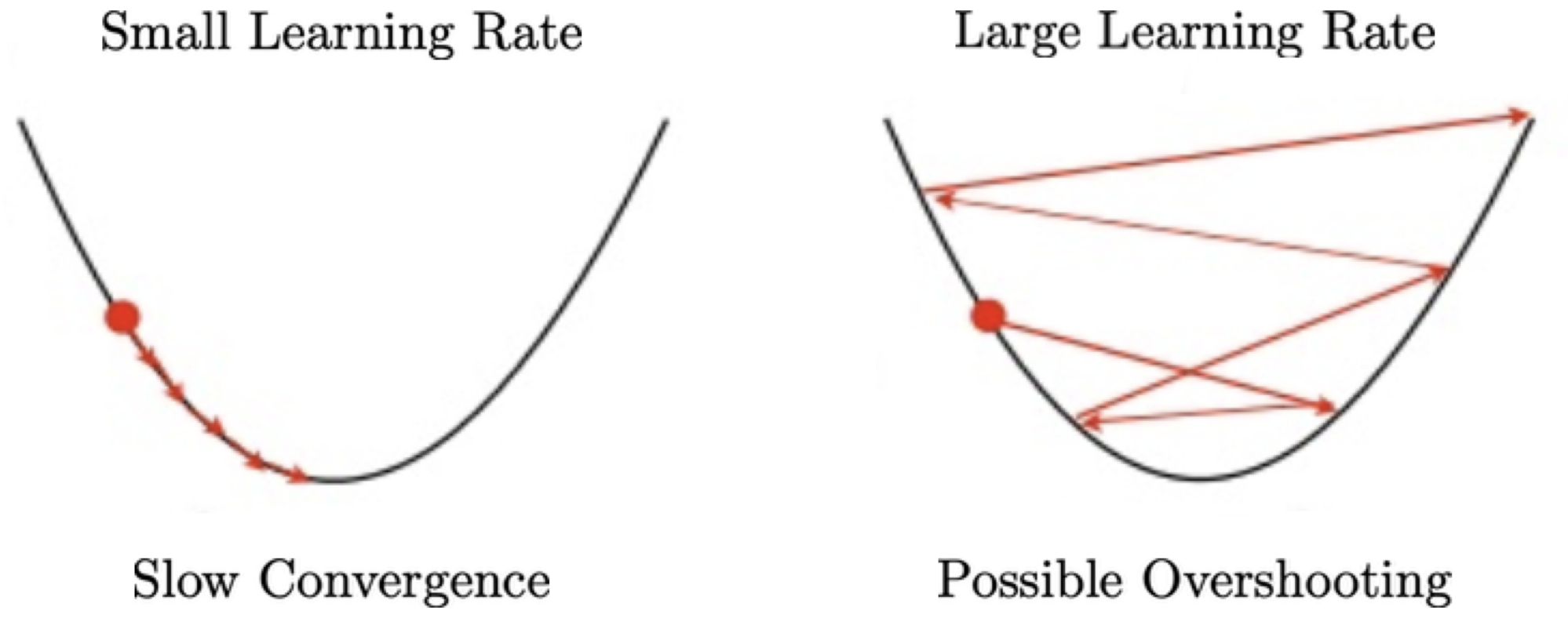
Gradient Descent





Go back to the calculus

Gradient Descent



Constant-step-size gradient descent

- **Inputs**: objective function M, step size a, initial guess $\theta^{(0)}$
- **While**: (*not too tired*) and (*making adequate progress*)
  - **Find**: $\nabla M(\theta^{(t)})$
  - **Set**: $\theta^{(t+1)} \leftarrow \theta^{(t)} - a\nabla M(\theta^{(t)})$
- Return final θ

- "*not too tired*" = $\{$set a maximum number of iterations$\}$
- "*making adequate progress*" = $\{M$ isn't changing by too little$\}$ and/or $\{\theta$ isn't changing by too little$\}$ and/or $\{\nabla M$ isn't too close to zero$\}$

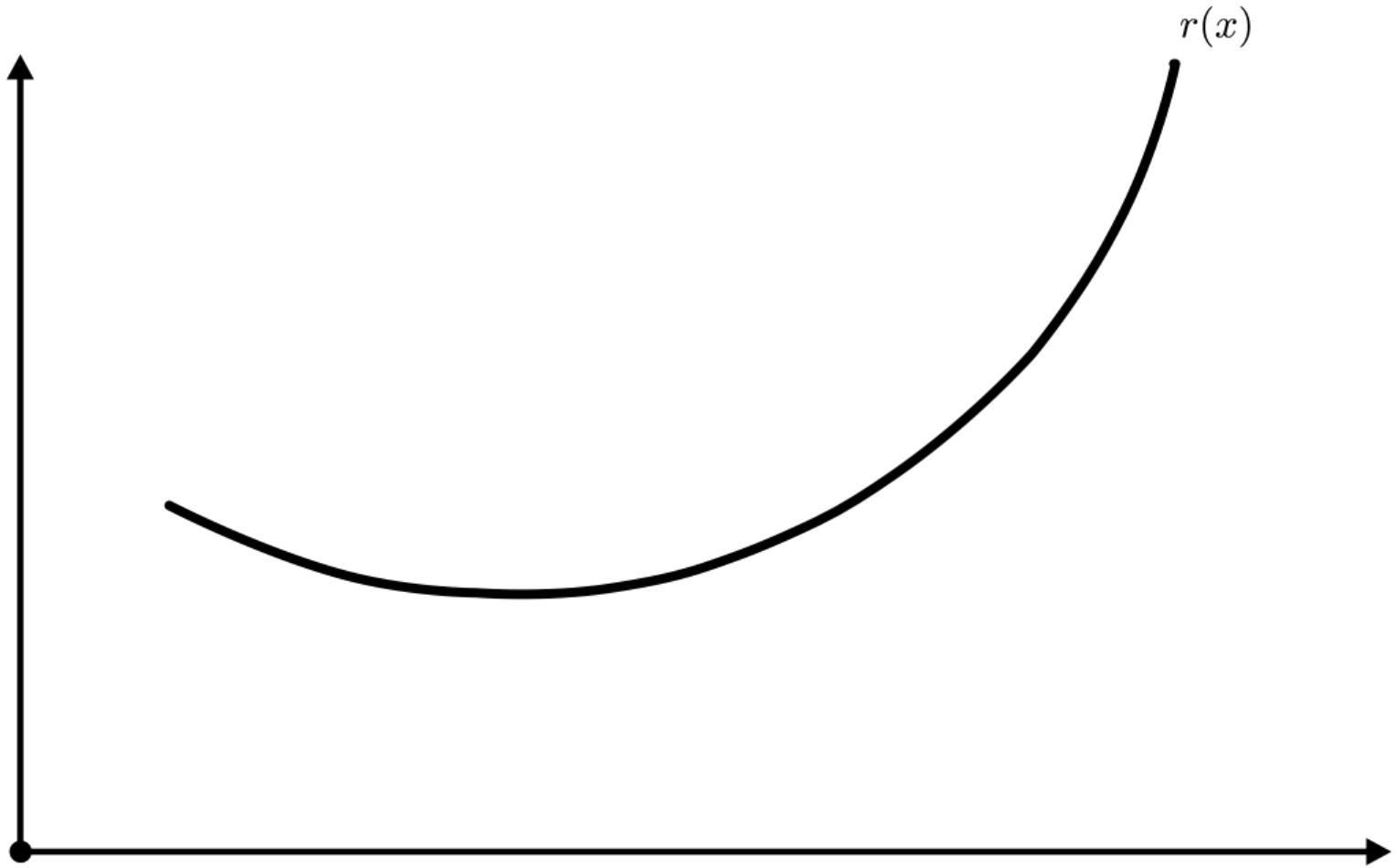
Constant-step-size gradient descent

- Pick an $a > 0$ that's small and use it at each step
- Each iteration of gradient descent takes $O(p)$ operations
    - Find p derivatives, multiply by a, add to $\theta^{(t-1)}$

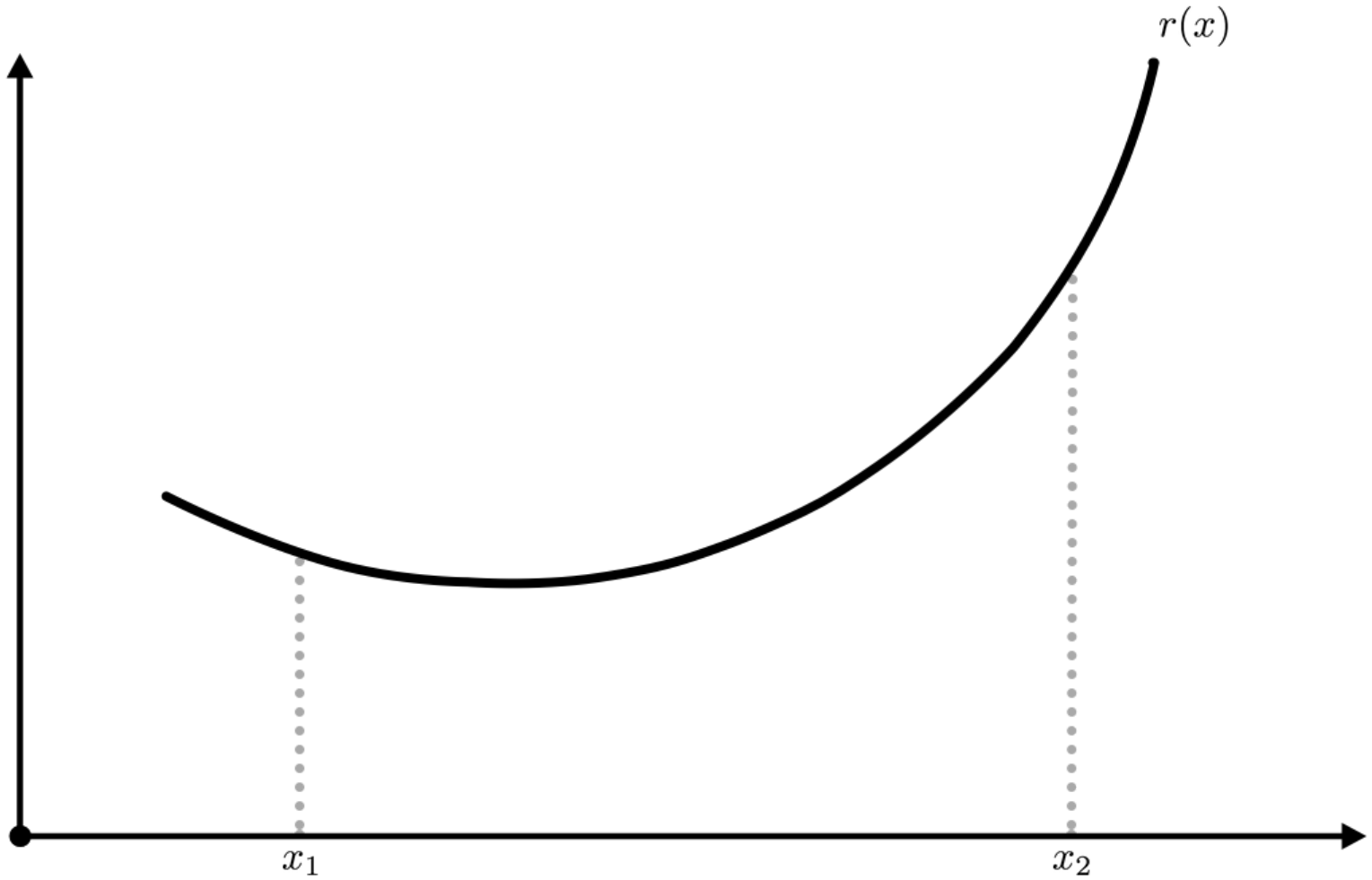
Constant-step-size gradient descent

- Pick an $a > 0$ that's small and use it at each step
- Each iteration of gradient descent takes $O(p)$ operations
  - Find p derivatives, multiply by a, add to $\theta^{(t-1)}$
- **IF** M is *nice*, $\theta^{(t)}$ is an ϵ-approximation of the optimum after $t = O(\epsilon^{-2})$ iterations
  - In other words, at that point $M(\theta^{(t)}) \leqslant \epsilon + \min M(\theta)$
  - "*Nice*" here means: **convex** and *second-differentiable*
- **IF** M is **very** *nice*, $\theta^{(t)}$ is an ϵ-approximation after only $t = O(\log 1/\epsilon)$ iterations
  - "*Nice*" + **strictly** convex

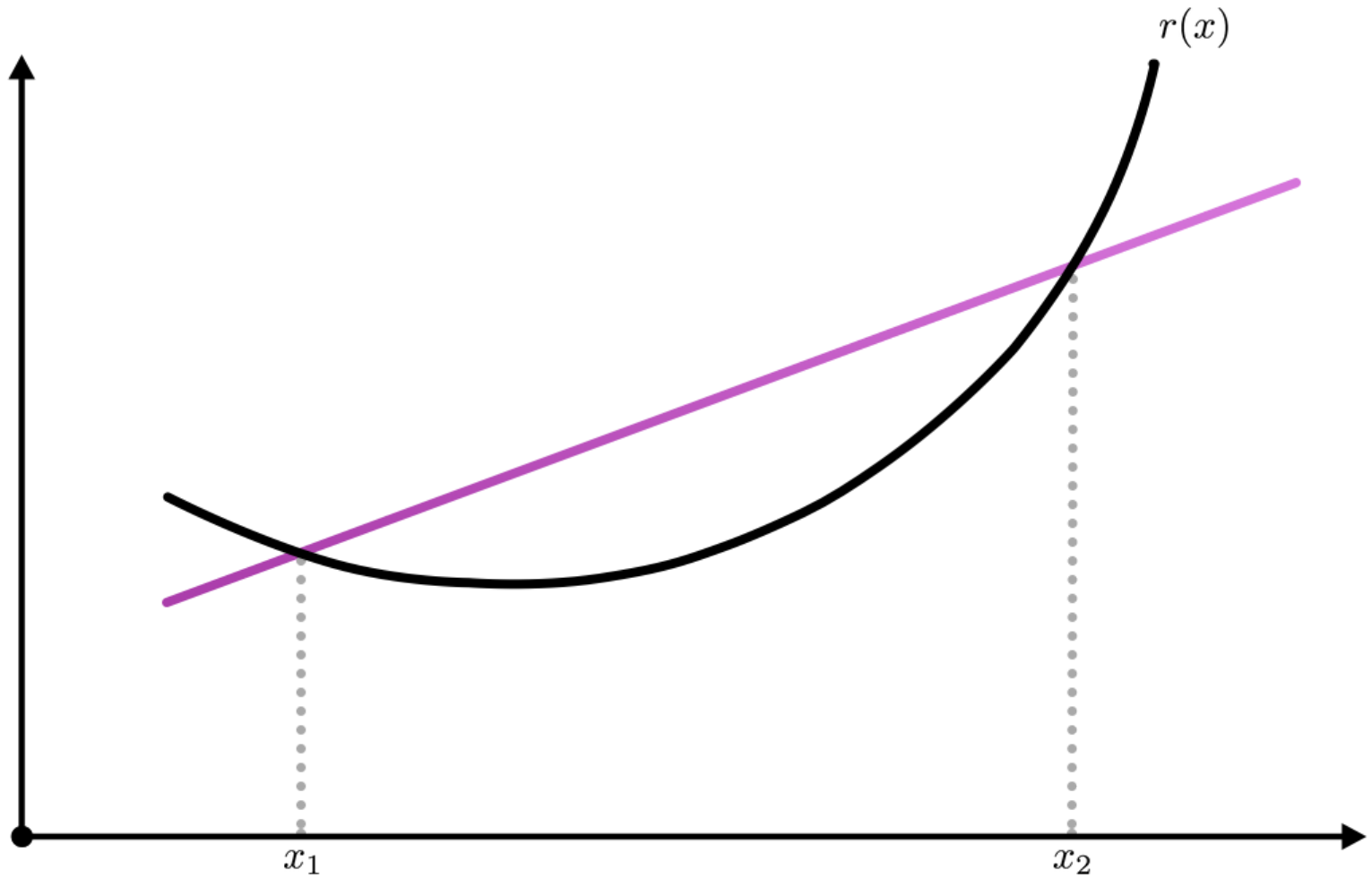
Convexity is and will be crucial



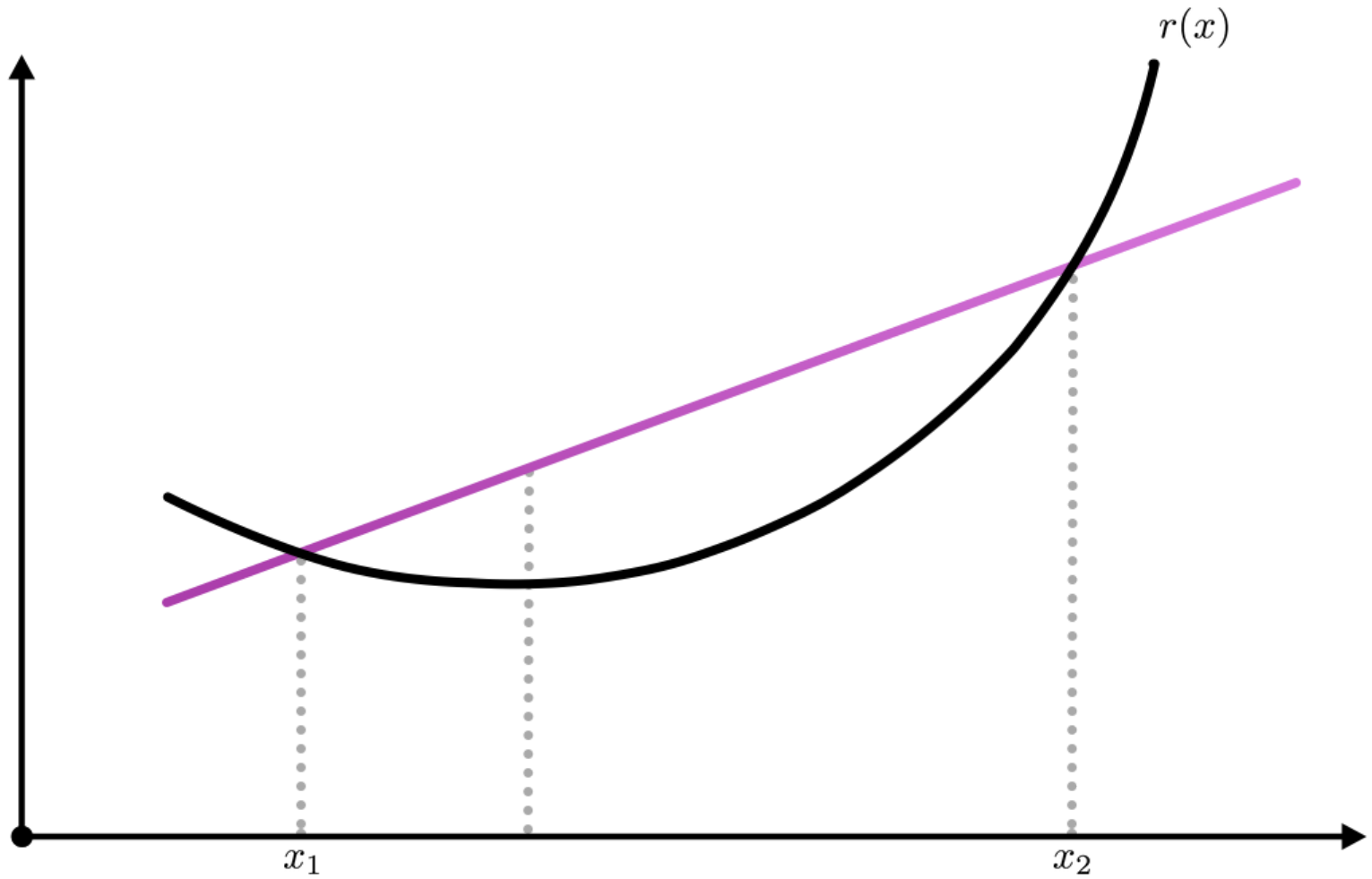
Convexity is and will be crucial



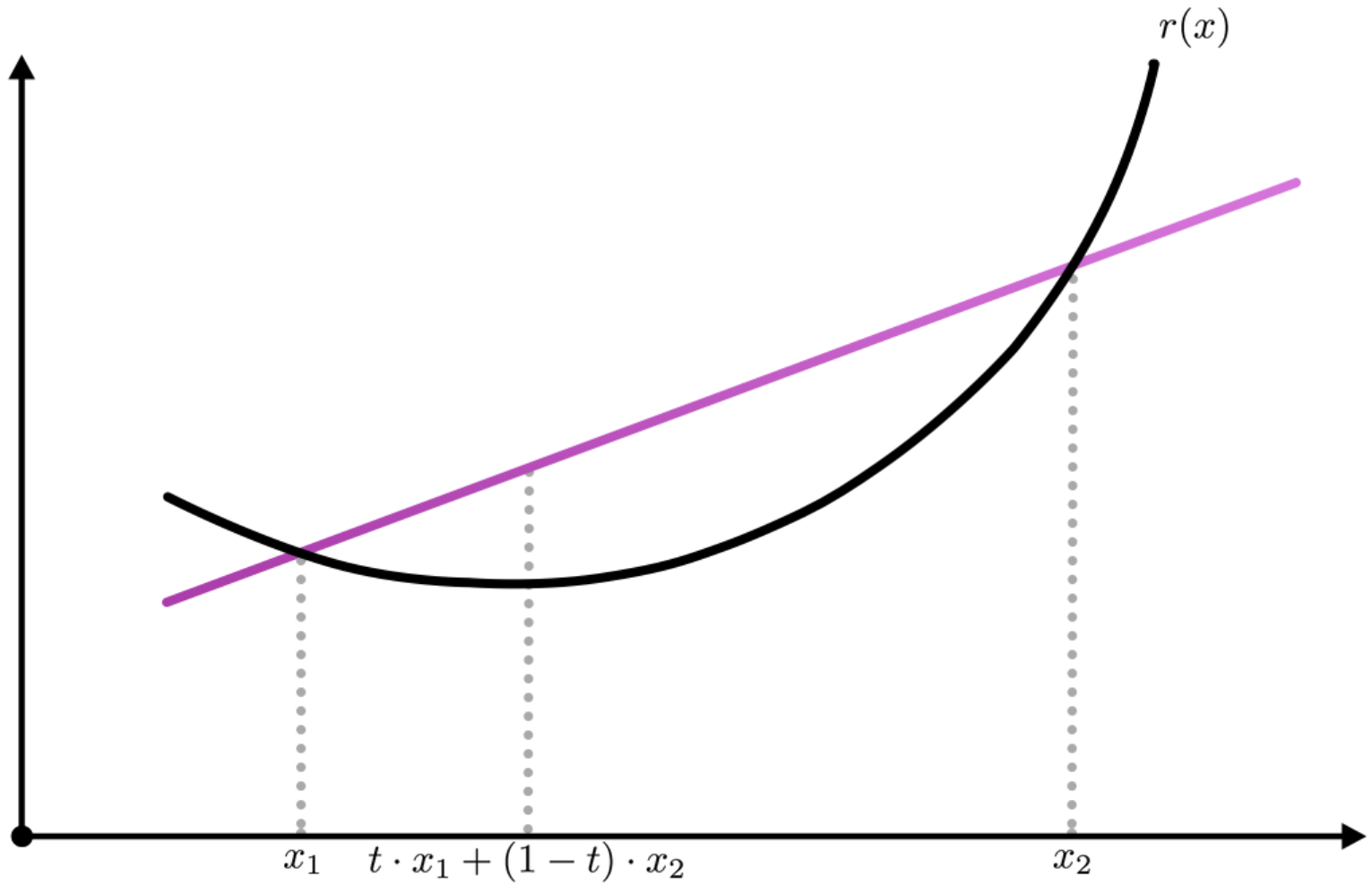
Convexity is and will be crucial



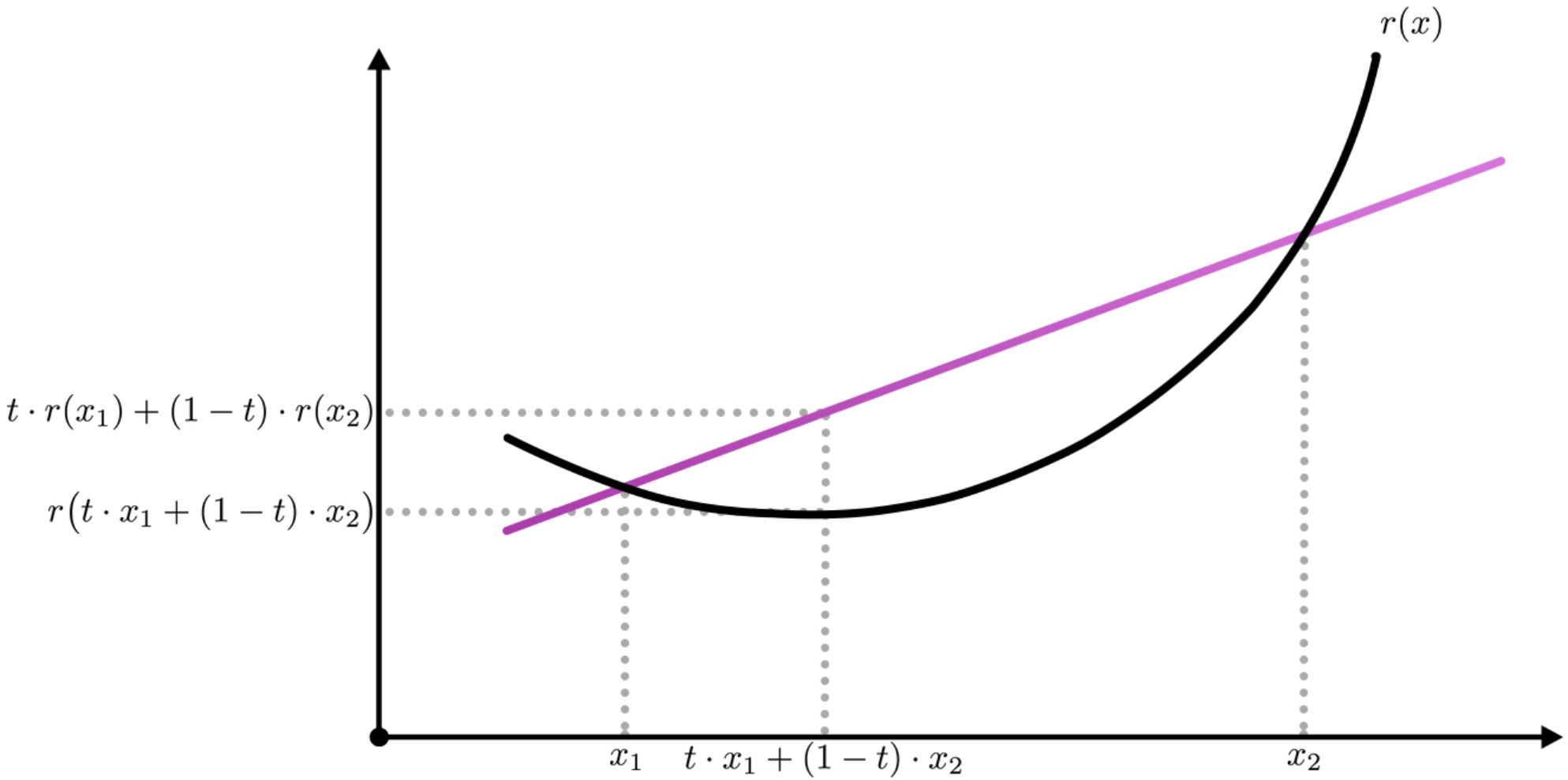
Convexity is and will be crucial



Convexity is and will be crucial



Convexity is and will be crucial



Convexity is and will be crucial

Gradient descent is basic, but powerful

- Gradient descent works well when there's a single global minimum, no flat parts to the function, and the step size is small enough to not over-shoot or zig-zag
- It's actually been re-invented a number of times under different names
  - For example, "*back-propagation*" (see Rumehlart et al., 1996)
- It's the work-horse for **large-scale** industrial applications in modern machine learning
  - Especially as **stochastic gradient descent** (see Robbins and Monro, 1951)
- It's still a bit *mysterious* why it works so well for those applications, which actually have lots of local minima!

Estimation error vs Optimization error

- For a generic ERM $\hat{s}$, remember our **approximation error** vs **estimation error** decomposition:

$$\begin{aligned} R(\hat{s}) &= R(s_{\text{opt}}) + (R(s_0) - R(s_{\text{opt}})) + (R(\hat{s}) - R(s_0)) \\ &= (\text{true minimum risk}) + (\text{approximation error from limited strategy set}) \\ &\quad + (\text{estimation error from not knowing the best-in-class set}) \end{aligned}$$

- Now we don't **even** have $\hat{s} = \operatorname{argmin} \widehat{R}(s)$, we have $\hat{s}_{\text{out}}$, the output of some algorithm

$$\begin{aligned} R(\hat{s}_{\text{out}}) &= R(s_{\text{opt}}) + (R(s_0) - R(s_{\text{opt}})) + (R(\hat{s}) - R(s_0)) + (R(\hat{s}_{\text{out}}) - R(\hat{s})) \\ &= (\text{optimal risk}) + (\text{approximation error}) + (\text{estimation error}) + (\text{optimization error}) \end{aligned}$$

- Optimization error $\approx$ what I've been calling ϵ
  - only $\approx$ because of R vs $\widehat{R}$ issue

Estimation error vs Optimization error

$$\text{risk} = \text{minimal risk} + \text{approximation error} + \text{estimation error} + \text{optimization error}$$

- Minimal risk and approximation error don't change with n or with how we optimize
- Estimation error shrinks with n: for large n, typically $O(p/n)$
  - Possibly more slowly converging in n for some families, or if p grows with n
  - *VC theory* will tell us that the *estimation error* can be $O\left(\frac{\log n/d}{n/d}\right)$
- Optimization error shrinks as we do more computational work
- There's no point to making the optimization error much smaller than the estimation error
  - More exactly: lots of work for little real benefit

Moral

Do **not** try to make the optimization error much smaller than *O(p/n)*

Do **not** bother optimizing more precisely than the noise level in the data will support

Beyond gradient descent: Newton's method

- Needing to pick the step-size a_t is annoying
- We'd like to take **big** steps, but ∇M is a local quantity and might be mis-leading far away
- **Idea**: We'd like to take **bigger** steps when the (non-zero) gradient does **not** change much
- This is **Newton's method**:

$$\theta^{(t+1)} = \theta^{(t)} - \left(\mathbf{h}(\theta^{(t)})\right)^{-1} \nabla M(\theta^{(t)})$$

  - One route to this: pretend M is quadratic, as justified by a Taylor expansion around the true minimum
- This is like gradient descent, but using the inverse Hessian to give the step size
  - And possibly a bit of rotation away from the gradient

Pros of Newton's method

$$\theta^{(t+1)} = \theta^{(t)} - \left(\mathbf{h}(\theta^{(t)})\right)^{-1} \nabla M(\theta^{(t)})$$

- **Adaptively-chosen** step size means harder to get zig-zags, over-shooting, etc.
- Need $O(\epsilon^{-2})$ steps to get an ϵ approximation to the minimum for "*nice*" functions
- For "*very nice*"functions, only need $O(\log(\log(1/\epsilon)))$ iterations
- Generally needs many **fewer** iterations than gradient descent
- Extremely useful for (approximate) **statistical inference**
  - $M(\theta) = \text{Likelihood Function}$: the *Hessian* (i.e. its curvature) is related to the *Fisher Information matrix* (it's inverse) and, consequently, to the (asymptotic) *standard error* of MLE's.
  - Hence, BFGS for example, will return the maximum likelihood estimates plus an *approximation* to their standard errors to build (asymptotic) confidence sets.

Cons of Newton's method

$$\theta^{(t+1)} = \theta^{(t)} - \left(\mathbf{h}(\theta^{(t)})\right)^{-1} \nabla M(\theta^{(t)})$$

- Hopeless if the Hessian doesn't exist or isn't invertible
- Need to take $O(p^2)$ second derivatives *and* p first derivatives, total $O(p^2)$
- Need to find $\theta^{(t+1)}$
  - Seems straightforward, it's $\theta^{(t+1)} = \theta^{(t)} - \left(\mathbf{h}(\theta^{(t)})\right)^{-1} \nabla M(\theta^{(t)})$
  - But inverting a $[p \times p]$ matrix takes $O(p^3)$ operations in general, so this would be an $O(p^3)$ step
  - Lots of variants to use approximate Hessians rather than the full deal (BFGS, built in to R's `optim()`, is one of these)

Gradient methods with big data

$$\theta^{(t+1)} = \theta^{(t)} - a_t \nabla M(\theta^{(t)})$$

$$\theta^{(t+1)} = \theta^{(t)} - \left(\mathbf{h}(\theta^{(t)})\right)^{-1} \nabla M(\theta^{(t)})$$

$$\widehat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(Y_i, s(X_i; \theta))$$

- Getting a value of $\widehat{R}$ at a particular θ is $O(n)$, getting $\nabla \widehat{R}$ is $O(np)$, getting $\mathbf{h}$ is $O(np^2)$
  - And that's assuming calculating $s(x_i; \theta)$ doesn't slow down with n
- Maybe OK if $n = 100$ or $n = 10^4$, but with $n = 10^9$ or $n = 10^{12}$, we really don't know which way to move

The Curses and Blessings of Dimensionality
(...in both, sampling information and model complexity...)

(...more examples to come...)

A way out: sampling is an unbiased estimate

- Pick *one* data point I at random, uniform on $1:n$
- $L\big(Y_I, s(X_I;\theta)\big)$ is random, but

$$\mathbb{E}_I\, L\big(Y_I, s(X_I;\theta)\big) = \widehat{R}(\theta)$$

- Re-brand $L\big(Y_I, s(X_I;\theta)\big)$ as $\widehat{R}_I(\theta)$

$$\begin{aligned}\mathbb{E}\widehat{R}_I(\theta) &= \widehat{R}(\theta)\\ \mathbb{E}\nabla\widehat{R}_I(\theta) &= \nabla\widehat{R}(\theta)\\ \mathbb{E}\nabla\nabla\widehat{R}_I(\theta) &= \mathbf{h}(\theta)\end{aligned}$$

Moral
Do **not** optimize with *all* the data, optimize with *random* samples

---

Stochastic gradient descent

Draw *lots* of random one-point samples and let their noise cancel out

---

1. Start with initial guess $\theta^{(0)}$, adjustment rate a
2. **While**: (*not too tired*) and (*making adequate progress*)

   a. $t^{\texttt{th}}$ iteration: pick random I uniformly on $1:n$

   b. **Set**:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \frac{a}{t} \nabla \widehat{R}_I(\theta^{(t)})$$

3. Return final θ

- **Remark**: Shrinking step-sizes by $1/t$ ensures noise in each gradient dies down

Stochastic gradient descent

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \frac{a}{t} \nabla \widehat{R}_I(\theta^{(t)})$$

- Tons of variants:
  - Put the data points $1:n$ in a random order and then cycle through them
  - Don't check the "*making adequate progress*" condition too often
  - Adjust the $1/t$ step-size to some other function
  - **Stochastic Newton's method**: use the sample to also calculate the Hessian and take a Newton's method step
  - **Mini-batch**: sample a *few* of random data points at once
  - Mini-batch stochastic Newton's method, etc.

Pros and cons of stochastic gradient methods

- Pro: Each iteration constant (or at least constant in n)
- Pro: Never need to hold all the data in memory at once
- Pro: Does converge eventually (at least if the non-stochastic method would)
- Cons: sampling noise increases optimization error
    - That is: more iterations to come within the same ϵ of the optimum as non-stochastic GD or Newton
- Over-all pro: often low computational cost to make the optimization error small compared to the estimation error

More optimization algorithms

- Ones which play more tricks with derivatives than just gradient descent and Newton ("*conjugate gradient*", etc., etc.)
- Ones which avoid derivatives ("*simplex*" or "*Nelder-Mead*")
- Ones which avoid derivatives and try random changes ("*simulated annealing*")
- Ones which use natural-selection-with-random-variation to evolve a whole population of approximate optima ("*genetic algorithms*")

Why are there so many different optimization algorithms?

- No one algorithm works well on *every* problem
  - Sometimes obvious: don't use Newton's method if Θ is discrete
- Fundamental limit: no algorithm is *universally* better than others on *every* problem (**no free lunch theorem** of Wolpert et al., 1997)
  - For every problem where your favorite algorithm does better than mine, I can design a new problem where my algorithm leads by just as much (see Culberson, 1998)
- We need to know *something* about the problem to select a good optimizer

Summing up

- With real data and real computers, finding the empirical-risk-minimizer means using an algorithm to solve an optimization problem
- These algorithms almost never give the **exact** optimum but just an *approximation*
- Usually, the longer an algorithm is allowed to work, the closer it can get to the true optimum
- This adds **optimization error** on to *estimation error*
- For *many* statistical learning problems, gradient descent and Newton's method work really well
    - With **sampling** to make them more computationally efficient for big data
- Don't bother reducing the optimization error much *beyond* the estimation error
- **No** algorithm is best for all problems
- **Coming up**:

    1. More on Convexity
    2. *Constrained* Optimization $\rightsquigarrow$ Lagrange Multipliers + Duality