Lecture 6, Optimization Algorithms

36-462/662, Spring 2022

3 February 2022

Previously

- What we really want is the **risk-minimizing strategy**,

$$s^* \equiv \operatorname*{argmin}_{s \in S} \mathbb{E}\left[\ell(Y, s(X))\right] = \operatorname*{argmin}_{s \in S} r(s)$$

- We often settle instead for the **empirical risk minimizer**

$$\hat{s}_n \equiv \operatorname*{argmin}_{s \in S} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, s(x_i)) = \operatorname*{argmin}_{s \in S} \hat{r}(s)$$

- Finding the minimizers means doing optimization
  - Objective function M, variable being optimized θ
  - Local vs. global optima
  - Location of optimum θ^*, value of optimum $M(\theta^*)$ (argmin vs. min)
  - First-order condition: the function is flat at the minimum, $\frac{dM}{d\theta}(\theta^*) = 0$
  - Second-order condition: the function curves upwards at the minimum, $\frac{d^2M}{d\theta^2}(\theta^*) > 0$

Today

- What if θ has more than one dimension?
  - First order condition: "the gradient vanishes"
  - Second order condition: "the Hessian is positive-definite"
- What about actual algorithms for computing θ^*?
  - Solving the first-order condition equations
  - Using the first derivatives
  - Using the first and second derivatives
  - Adapting to big data
- How hard should we try to optimize anyway?
- "What do I type in R?"

What about more than one dimension?

- Usually θ is a vector of $p > 1$ dimensions
- We *usually* can't optimize each coordinate separately
- What should happen at an interior minimum θ^*?
- M should have *no slope* at θ^* in *every* direction
- M should increase as we move away from θ^* in *every* direction

No slope in any direction: the first-order condition

- Pick any direction $\vec{v}$, a vector of length 1, say $(v_1, v_2, \ldots v_p)$
- The slope of M in that direction, at θ, is (chain rule)

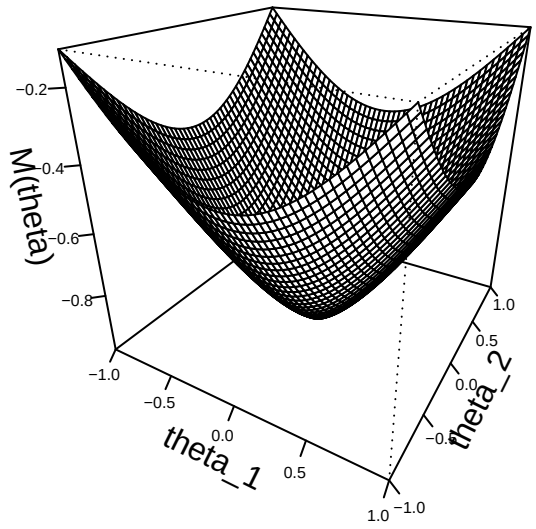
$$\sum_{i=1}^{p} v_i \frac{\partial M}{\partial \theta_i}(\theta) = \vec{v} \cdot \nabla M(\theta)$$

- Here $\nabla M(\theta)$ is the **gradient** of M at θ, the vector of partial derivatives

$$\nabla M(\theta) \equiv \left[\begin{array}{ccc} \frac{\partial M}{\partial \theta_1}(\theta) & \ldots & \frac{\partial M}{\partial \theta_p}(\theta) \end{array} \right]$$

- No slope in *any* direction at θ^* means: $\vec{v} \cdot \nabla M(\theta^*) = 0$ for all $\vec{v} \neq 0$
- And *that* means: $\nabla M(\theta^*) = 0$
- The **first-order condition** is: "the gradient vanishes at the optimum"

No slope in any direction



First-order condition or first-order *conditions*?

- We have one vector equation $\nabla M(\theta^*) = 0$
- This is the same as a *system* of p equations for the partial derivatives:

$$\begin{aligned} \frac{\partial M}{\partial \theta_1}(\theta^*) &= 0 \\ &\vdots \\ \frac{\partial M}{\partial \theta_p}(\theta^*) &= 0 \end{aligned}$$

- We also have p unknowns, $\theta^* = \left[\begin{array}{ccc} \theta_1^* & \ldots & \theta_p^* \end{array} \right]$
- p equations for p unknowns $\Rightarrow$ typically a solution
  - Typically a *unique* solution if all the equations are linear in θ^*
  - Often not unique because nonlinear in θ^*
  - But still, there are solutions!

The function increases in every direction: the second-order condition

- Second-order Taylor series for vectors:

$$M(\theta) \approx M(\theta^*) + (\theta - \theta^*) \cdot \nabla M(\theta^*) + \frac{1}{2}(\theta - \theta^*) \cdot (\nabla\nabla M(\theta^*))\,(\theta - \theta^*)$$

- Here $\nabla\nabla M(\theta^*)$ is the matrix of second partial derivatives, $\frac{\partial^2 M}{\partial \theta_i \partial \theta_j}$, a.k.a. the **Hessian**[1], or $\mathbf{h}$
- First-order condition says the gradient term is zero at θ^*, so

$$M(\theta) \approx M(\theta^*) + \frac{1}{2}(\theta - \theta^*) \cdot (\nabla\nabla M(\theta^*))\,(\theta - \theta^*)$$

  - "Typically, functions look quadratic near their minima"
- θ^* is a minimum means:

$$(\theta - \theta^*) \cdot (\nabla\nabla M(\theta^*))\,(\theta - \theta^*) > 0$$

Positive-definite matrices

- A square matrix $\mathbf{h}$ is **positive-definite** when, for any non-zero vector $\vec{v}$,

$$\vec{v} \cdot \mathbf{h}\vec{v} > 0$$

  - If we only have $\vec{v} \cdot \mathbf{h}\vec{v} \geq 0$ then $\mathbf{h}$ is only **non-negative-definite** (or **positive semi-definite**)
- *Not* the same as $\mathbf{h}$ only having positive entries!
  - E.g., $\mathbf{p} = \begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}$ is positive-definite
  - E.g., $\mathbf{n} = \begin{bmatrix} 0.5 & 1 \\ 1 & 0.5 \end{bmatrix}$ is not positive-definite
- We write this as $\mathbf{h} \succ 0$
  - Non-negative-definite is $\mathbf{h} \succeq 0$
- For symmetric matrices: $\mathbf{h}$ is positive definite $\Leftrightarrow$ all eigenvalues of $\mathbf{h}$ are > 0
  - The Hessian matrix $\nabla\nabla M$ is always symmetric (why?)
  - We'll do a refresher on eigenvalues in a few weeks before we really need them

The first- and second- order conditions for minima

For θ^* to be a local minimum,

- First-order condition: "The gradient must vanish", $\nabla M(\theta^*) = 0$
  - Necessary, except at a boundary
- Second-order condition: "The Hessian should be positive-definite", $\nabla\nabla M(\theta^*) \succ 0$
  - Sufficient; minima where it's violated are weird and a-typical
  - Necessary: $\nabla\nabla M(\theta^*) \succeq 0$, "the Hessian must be non-negative-definite"

Near a minimum, nice functions look quadratic

- Taylor approximation again: if θ^* is a local minimum, so $\nabla M(\theta^*) = 0$, then

$$M(\theta) \approx M(\theta^*) + \frac{1}{2}(\theta - \theta^*) \cdot (\nabla\nabla M(\theta^*))\,(\theta - \theta^*)$$

[1] After L. O. Hesse, 1811–1874

- Consequence: if we come *close* to the location of minimum, so $\|\theta - \theta^*\| = \delta \ll 1$, then

$$M(\theta) \approx M(\theta^*) + O(\delta^2)$$

- If we can get δ-close to the *location* of the optimum, we get $O(\delta^2)$-close to the *value* of the optimum (and $\delta^2 \ll \delta \ll 1$)
- To get within ϵ of the *value* of the optimum, we need to only get within $O(\sqrt{\epsilon})$ of the *location* of the optimum (and $\sqrt{\epsilon} \gg \epsilon$ if $\epsilon \ll 1$)

Minimizing risk vs. minimizing empirical risk

- We *want* to minimize risk,

$$\theta^* = \operatorname*{argmin}_{\theta \in \Theta} r(\theta) = \operatorname*{argmin}_{\theta \in \Theta} \mathbb{E}\left[\ell(Y, s(X))\right]$$

- We *can* minimize empirical risk,

$$\widehat{\theta} = \operatorname*{argmin}_{\theta \in \Theta} \hat{r}(\theta) = \operatorname*{argmin}_{\theta \in \Theta} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, s(x_i))$$

- We're going to see later that

$$\|\widehat{\theta} - \theta^*\| = O(\sqrt{1/n})$$

  - Basically: because of the law of large numbers
  - Assuming θ has finite dimension p not changing with n
- Consequence:

$$r(\widehat{\theta}) \approx r(\theta^*) + O(p/n)$$

  - Factor of p comes from the Hessian (basically)
- $\Rightarrow$ Minimizing the empirical risk comes closer and closer to minimizing the true risk

Finding the minimum: optimization algorithms

- An optimization algorithm starts from M and Θ, and (usually) a starting guess $\theta^{(0)}$, and finds an **approximation** to $\operatorname{argmin}_{\theta \in \Theta} M(\theta)$, say θ_{out}
- We care about approximating the *value*, not the location: the algorithm gets ϵ-close when

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

- Usually, 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 $O(\epsilon^{-d})$ steps is **polynomial** (tolerable, depending on d)
    - $O(\log 1/\epsilon)$ is **logarithmic** (very nice)
    - $\exp\left(O(1/\epsilon)\right)$ is **exponential** (bad)

How do we build an optimization algorithm?

- Remember our first and second order conditions:

$$\nabla M(\theta^*) = 0 \tag{1}$$
$$\nabla\nabla M(\theta^*) \succ 0 \tag{2}$$

- Two big approaches at this point:
  1. Solve the equations
  2. Keep moving until the gradient ∇M goes to 0

Optimizing by equation-solving

- Use the first-order condition to get a system of equations

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

  - One equation per coordinate of θ (as we saw)
  - When M is empirical risk $\hat{r}$, sometimes called the **estimting equations** or even **normal equations**
- Solve the system of equations for θ^*
- If there's more than one solution, check the second-order conditions
- We did this for ordinary least squares, weighted least squares...

Pros and cons of the solve-the-equations approach

- Con: You need to *set up* the system of equations, and often finding ∇M would itself be a pain
  - Pro: Numerical differentiation is a thing, however
- ?: 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
  - Pro: 200+ years of work have given us *very* good solvers for linear systems
    - Pro: For linear systems, even very old-fashioned methods that go back to Gauss around 1800 get ϵ approximations with $O(\log 1/\epsilon)$ iterations
  - Con: General-purpose nonlinear equation-solving is still much harder
    - ?: sometimes works by using Taylor expansion to linearize
    - Con: sometimes works by turning the solve-the-equations into "minimize the difference between the left and the right hand side of the equation"

Go back to the calculus

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

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

- Repeat:

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

- First-order condition means: a local optimum will be a fixed point!
- Issue: how big are the **step sizes** a_t?
  - (Sometimes called the **learning rate**, confusingly enough)

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 to bother with
  - θ isn't changing by too little to bother with
  - ∇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 $\theta^{(t-1)}$
- If M is nice, $\theta^{(t)}$ is an ϵ-approximation of the optimum after $t = O(\epsilon^{-2})$ iterations
  - i.e. at that point $M(\theta^{(t)}) \leq \epsilon + \min M(\theta)$
  - "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" plus *strictly* convex

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
  - e.g., "back-propagation" (Rumelhart, Hinton, and Williams 1986)
- It's the work-horse for large-scale industrial applications in modern machine learning
  - especially as **stochastic gradient descent**
- It's still a bit mysterious why it works so well for those applications, which actually have lots of local minima!

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
- $\Rightarrow$ We'd like to take bigger steps when the gradient doesn't 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

- Adaptively-chosen step size makes it harder to zig-zag, over-shoot, etc.
- Generally needs many fewer iterations than gradient descent
  - 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

Cons of Newton's method

- 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
- Alternative: solve $\mathbf{h}\theta^{(t+1)} = \mathbf{h}\theta^{(t)} - \nabla M(\theta^{(t)})$ for $\theta^{(t+1)}$ for the unknown $\theta^{(t+1)}$
  - (Take the basic update equation for Newton's method and multiply both sides by $\mathbf{h}$ from the left)
  - Solving a system of p linear equations for a *particular* RHS can be done faster than inverting a matrix (which'd give the solution for *any* RHS)
  - Lots of variants to use approximate Hessians rather than the full deal (BFGS, built in to R's `optim()`, is one of these)
- So each iteration is $O(p^2)$, much slower than gradient descent's $O(p)$
  - $O(p^2)$ to get Hessian and gradient plus $O(p^2)$ to solve for update $= O(p^2)$

Gradient methods with big data

$$\hat{r}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, s(x_i;\theta))$$

- Getting a value of $\hat{r}$ at a particular θ is $O(n)$, getting $\nabla\hat{r}$ 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 when $n = 100$ or $n = 10^4$, but with $n = 10^9$ or $n = 10^{12}$, we really don't know which way to move

A way out: sampling is an unbiased estimate

- Pick *one* data point I at random, uniform on $1:n$
- $\ell(y_I, s(x_I;\theta))$ is random, but
$$\mathbb{E}\left[\ell(y_I, s(x_I;\theta))\right] = \hat{r}(\theta)$$
- Re-brand $\ell(y_I, s(x_I;\theta))$ as $\hat{r}_I(\theta)$

$$\mathbb{E}\left[\hat{r}_I(\theta)\right] = \hat{r}(\theta) \quad (3)$$
$$\mathbb{E}\left[\nabla\hat{r}_I(\theta)\right] = \nabla\hat{r}(\theta) \quad (4)$$
$$\mathbb{E}\left[\nabla\nabla\hat{r}_I(\theta)\right] = \mathbf{h}(\theta) \quad (5)$$

- $\Rightarrow$ Don't 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:

0. Start with initial guess $\theta^{(0)}$, adjustment rate a
1. While (not too tired) and (making adequate progress))
   a. At t^{th} iteration, pick random I uniformly on $1:n$
   b. Set $\theta^{(t+1)} \leftarrow \theta^{(t)} - \frac{a}{t}\nabla\hat{r}_I(\theta^{(t)})$
2. Return final θ

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

Stochastic gradient descent (2)

- 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 is (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 optimzation 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")

Estimation error vs. optimization error

- Remember our approximation error vs. estimation error decomposition:

$$\begin{aligned} r(\hat{s}) &= r(\sigma) + (r(s^*) - r(\sigma)) + (r(\hat{s}) - r(s^*)) && (6)\\ &= (\text{true minimum risk}) + (\text{approximation error from limited strategy set}) && (7)\\ &\quad +(\text{estimation error from not knowing the best-in-class set}) && (8) \end{aligned}$$

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

$$\begin{aligned} r(\hat{s}_{\text{out}}) &= r(\sigma) + (r(s^*) - r(\sigma)) + (r(\hat{s}) - r(s^*)) + (r(\hat{s}_{\text{out}}) - r(\hat{s})) && (9)\\ &= (\text{optimal risk}) + (\text{approximation error}) + (\text{estimation error}) + (\text{optimization error}) && (10) \end{aligned}$$

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

Estimation error vs. optimization error (2)

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

- 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
- 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
- So: don't try to make the optimization error much smaller than $O(p/n)$

  Don't bother optimizing more precisely than the noise in the data will support

What do we do in R?

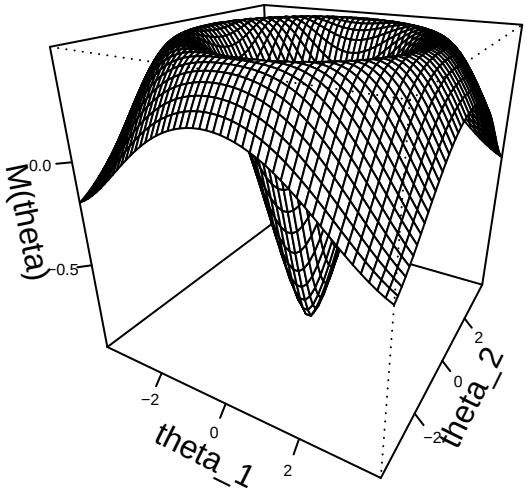
- The basic function for optimization in R is `optim()`

```
optim(par, fn, gr, method, ...)
```

- `par` = Initial guess at the "parameters" = a vector, our $\theta^{(0)}$
- `fn` = Function to be minimized, our $M(\theta)$
  - Should take a *single* vector as input and return a single numeric value
  - R lets functions be arguments to other functions without any fuss
- `gr` = Function to calculate the gradient, our $\nabla M(\theta)$
  - Should take a vector and return a vector of the same length
  - Optional, not used by all methods, if missing R will try numerical differentiation
    - Numerical differentiation can be very slow so your doing some math to work this out can be very useful
- `method` = Which optimization algorithm?
  - Default is `Nelder-Mead` a.k.a. simplex method, doesn't use derivatives, can be good for discontinuous functions but inefficient for smooth ones
  - `BFGS` is a Newton-type method, but with clever tricks to not spend quite so much time computing and inverting Hessians
- `...`: lots of extra settings, including things like the "tolerance" (how small an improvement in `fn` / M to bother with)

No, really, what do we do in R?

```
my.fn <- function(t) {
    - exp(-0.25*sqrt(t[1]^2+t[2]^2))*cos(sqrt(t[1]^2+t[2]^2))
}
```



No, really, what do we do in R?

```
my.fit <- optim(par=c(1,1), fn=my.fn, method="BFGS") # Starting here is dumb!
str(my.fit)
```

```
## List of 5
##  $ par        : num [1:2] 4.96e-10 4.96e-10
##  $ value      : num -1
##  $ counts     : Named int [1:2] 41 13
##   ..- attr(*, "names")= chr [1:2] "function" "gradient"
##  $ convergence: int 0
##  $ message    : NULL
```

```
my.fit$par    # Location of the minimum
```

```
## [1] 4.956205e-10 4.956205e-10
```

```
my.fit$value  # Value at the minimum
```

```
## [1] -1
```

What if `optim()` isn't enough?

- We'll look briefly at doing *constrained* optimization in R next time
- [https://cran.r-project.org/web/views/Optimization.html] is your friend

Summing up

- When optimizing multiple variables at once, we still want all the derivatives to be zero (first-order condition, $\nabla M(\theta^*) = 0$), and the second derivative to be positive in every direction (second order condition, $\nabla\nabla M(\theta^*) \succ 0$)
- 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 one algorithm is best for all problems

Backup: More about second-order conditions

- I've been writing $\nabla\nabla M \succ 0$, which is a *sufficient* condition for a local minimum (if the first-order condition also holds)
- $\nabla\nabla M \succeq 0$ is a *necessary* condition for a local minimum
  - There can't be any directions in which the function curves *down*
- Again, think of θ^4 in one dimension
- Again, the typical local minimum of a smooth function has a positive-definite Hessian, cases where it's only non-negative-definite are fragile
  - See backup slides to lecture 5 for more, including an illustration

Backup: Representation vs. Reality

- Optimization algorithms don't *really* start with M, Θ and $\theta^{(0)}$
- They start with *digital representations* of all these things
- Different representations can be easier or harder to work with
- *Digital* representations of *continuous* things always have limited detail, which can lead to extra error
- The late Joseph Traub, of our CS department, developed an interesting theory about *how much* detail the representations *had* to have, to achieve a certain accuracy
  - See Traub and Werschulz (1998) if that sounds interesting

Backup: Why are there so many different optimization algorithms?

- "Come up with a new algorithm" is a way to make a mark...
- 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, shown by the **no free lunch theorem** of Wolpert and Macready (1997)
  - In fact: For every problem where your favorite algorithm does better than mine, I can design a new problem where my algorithm leads yours by just as much (Culberson 1998)
- We need to know *something* about the problem to select a good optimizer

References

Culberson, Joseph C. 1998. "On the Futility of Blind Search: An Algorithmic View of 'No Free Lunch'." *Evolutionary Computation* 6:109–27. http://www.cs.ualberta.ca/~joe/Abstracts/TR96-18.html.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323:533–36. https://doi.org/10.1038/323533a0.

Traub, J. F., and A. G. Werschulz. 1998. *Complexity and Information*. Lezioni Lincee. Cambridge, England: Cambridge University Press.

Wolpert, David H., and William G. Macready. 1997. "No Free Lunch Theorems for Optimization." *IEEE Transactions on Evolutionary Computation* 1:67–82. https://doi.org/10.1109/4235.585893.