Optimization

Brooks Paige

Week 4

Optimization

We are able to find closed-form expressions for MAP estimates and posterior distributions for a linear regression model, with a Gaussian likelihood and Gaussian priors.

That will hardly ever happen again!

Optimization

We are able to find closed-form expressions for MAP estimates and posterior distributions for a linear regression model, with a Gaussian likelihood and Gaussian priors.

That will hardly ever happen again!

We will often need to find "best" values of parameters, perhaps ones which minimize some loss $\mathcal{L}(\boldsymbol{\theta})$, or maximize a joint probability.

In general it will **extremely difficult** to find an algorithm that will **guarantee** to find the optimal $\boldsymbol{\theta}$.

How can we minimize a function f?

. . . particularly if we don't know much about it?

The trick is to use a Taylor expansion. For a small value $\boldsymbol{\delta}$, we can approximate $f(\boldsymbol{\theta} + \boldsymbol{\delta})$ as

$$f(\boldsymbol{\theta} + \boldsymbol{\delta}) \approx f(\boldsymbol{\theta}) + \boldsymbol{\delta}^\top \nabla f(\boldsymbol{\theta}).$$

This is a **first-order** approximation, which uses the function value and its gradient at $\boldsymbol{\theta}$.

How can we minimize a function f?

. . . particularly if we don't know much about it?

The trick is to use a Taylor expansion. For a small value $\boldsymbol{\delta}$, we can approximate $f(\boldsymbol{\theta} + \boldsymbol{\delta})$ as

$$f(\boldsymbol{\theta} + \boldsymbol{\delta}) \approx f(\boldsymbol{\theta}) + \boldsymbol{\delta}^\top \nabla f(\boldsymbol{\theta}).$$

This is a **first-order** approximation, which uses the function value and its gradient at $\boldsymbol{\theta}$. A second-order approximation

$$f(\boldsymbol{\theta} + \boldsymbol{\delta}) \approx f(\boldsymbol{\theta}) + \boldsymbol{\delta}^\top \nabla f(\boldsymbol{\theta}) + \frac{1}{2}\boldsymbol{\delta}^\top H(\boldsymbol{\theta})\boldsymbol{\delta},$$

where $H(\boldsymbol{\theta}) = \nabla^2 f(\boldsymbol{\theta})$ is the Hessian of f, is **more accurate but also more expensive** to compute.

First order methods

Gradient descent

We wish to find $\boldsymbol{\theta}$ that minimizes $f(\boldsymbol{\theta})$. For general f there is no closed-form solution to this problem and we typically resort to iterative methods.

Gradient descent is a first-order method; it only uses the gradient. We assume our function is roughly linear around a starting point $\boldsymbol{\theta}$, and

$$f(\boldsymbol{\theta} + \mathbf{s}) \approx f(\boldsymbol{\theta}) + \mathbf{s}^\top \nabla f(\boldsymbol{\theta}).$$

Our goal is to choose a good step direction $\mathbf{s}$.

Gradient descent

As part of an iterative algorithm, for $\boldsymbol{\theta}_{t+1} \approx \boldsymbol{\theta}_t$, we have

$$f(\boldsymbol{\theta}_{t+1}) \approx f(\boldsymbol{\theta}_t) + \underbrace{(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t)^\top}_{\mathbf{s}} \nabla f(\boldsymbol{\theta}_t).$$

Gradient descent

As part of an iterative algorithm, for $\boldsymbol{\theta}_{t+1} \approx \boldsymbol{\theta}_t$, we have

$$f(\boldsymbol{\theta}_{t+1}) \approx f(\boldsymbol{\theta}_t) + \underbrace{(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t)^\top}_{\mathbf{s}} \nabla f(\boldsymbol{\theta}_t).$$

The simplest steepest descent approach is to choose a step direction $\mathbf{s}$ (or equivalently, a next location $\boldsymbol{\theta}_{t+1}$), as

$$\mathbf{s} \equiv \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\epsilon \nabla f(\boldsymbol{\theta}_t)$$

for small **learning rate** $\epsilon > 0$.

Gradient descent

As part of an iterative algorithm, for $\boldsymbol{\theta}_{t+1} \approx \boldsymbol{\theta}_t$, we have

$$f(\boldsymbol{\theta}_{t+1}) \approx f(\boldsymbol{\theta}_t) + \underbrace{(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t)^\top}_{\mathbf{s}} \nabla f(\boldsymbol{\theta}_t).$$

The simplest steepest descent approach is to choose a step direction $\mathbf{s}$ (or equivalently, a next location $\boldsymbol{\theta}_{t+1}$), as

$$\mathbf{s} \equiv \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\epsilon \nabla f(\boldsymbol{\theta}_t)$$

for small **learning rate** $\epsilon > 0$. Assuming our approximation holds, this decreases f, since

$$f(\boldsymbol{\theta}_{t+1}) \approx f(\boldsymbol{\theta}_t) - \underbrace{\epsilon \nabla f(\boldsymbol{\theta}_t)^\top \nabla f(\boldsymbol{\theta}_t)}_{\geq 0} \leq f(\boldsymbol{\theta}_t).$$

Gradient descent: general picture

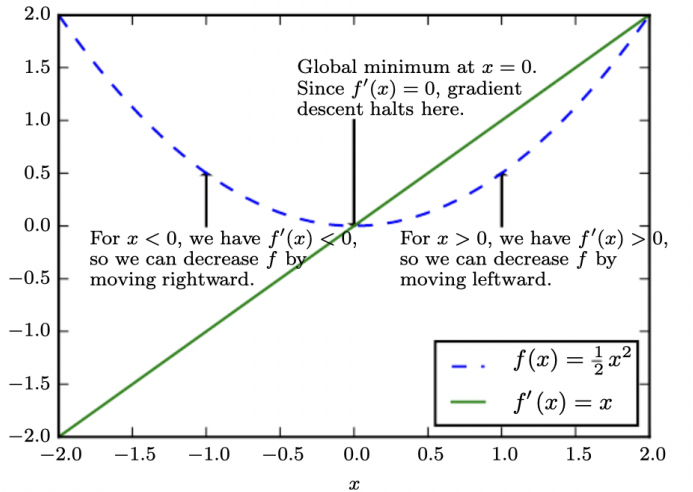
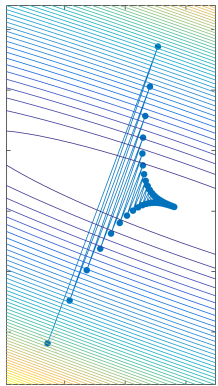


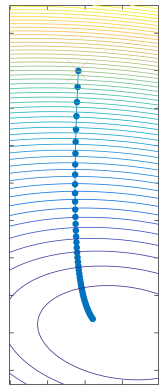
Figure: Goodfellow et al., Ch. 4

Learning rates for gradient descent

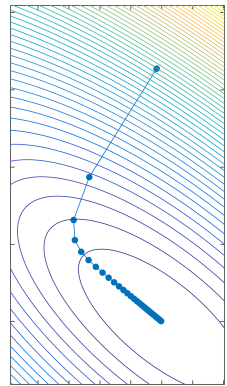
Note the difference between converging to the minimal function value, and the optimum parameters; we might have almost converged to the minimal value but still be a long way from the optimum parameters.



(a) Learning rate too large



(b) Learning rate too small



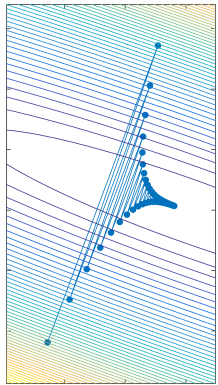
(c) Learning rate OK

Figures: Dmitry Adamskiy and David Barber

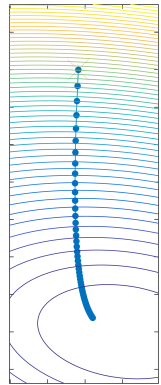
Learning rates for gradient descent

Common to consider adapting the learning rate.

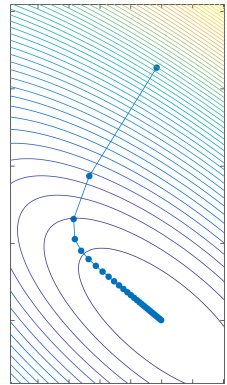
This is usually determined by experimenting with different values or learning schedules.



(a) Learning rate too large



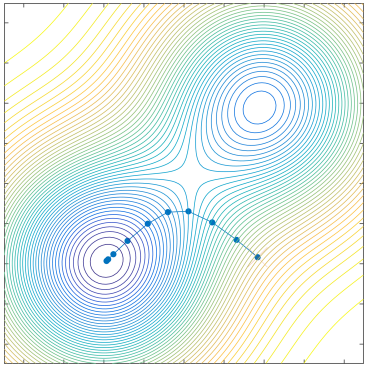
(b) Learning rate too small



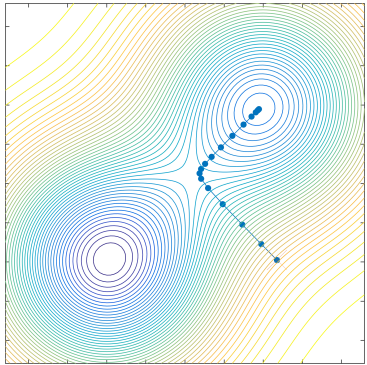
(c) Learning rate OK

Figures: Dmitry Adamskiy and David Barber

Gradient descent: local minima



(a) Found global minimum



(b) Found local minimum

For non-convex functions, depending on the initial point, even for the same function and small learning rate, we can converge to different solutions.

Figures: Dmitry Adamskiy and David Barber

Stochastic gradient descent

- In machine learning, the loss function is often the sum (average) of losses for individual examples:

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L_i(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \ell(f(x_i|\boldsymbol{\theta}), y_i)$$

- If the dataset is big, one step of gradient descent could be very expensive.
- Instead, we can use **stochastic gradient descent**, iterating over the training set examples:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \nabla L_i(\boldsymbol{\theta})$$

- We can make progress even before we see all the data!

Mini-batch stochastic gradient descent

Problem: Gradient estimates from one datapoint are noisy.

- Common solution is to use mini-batches of training examples and compute the gradient estimate based on those. For $j = 1, \ldots, M$ subsampled from $i = 1, \ldots, N$,

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}) \approx \frac{1}{M} \sum_{j=1}^{M} \nabla_{\boldsymbol{\theta}} L_j(\boldsymbol{\theta}).$$

Mini-batch stochastic gradient descent

Problem: Gradient estimates from one datapoint are noisy.

- Common solution is to use mini-batches of training examples and compute the gradient estimate based on those. For $j = 1, \ldots, M$ subsampled from $i = 1, \ldots, N$,

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}) \approx \frac{1}{M} \sum_{j=1}^{M} \nabla_{\boldsymbol{\theta}} L_j(\boldsymbol{\theta}).$$

- Trade-offs for different batch sizes:
  - Large batch size: less noisy estimates, convergence might be faster (in number of iterations)
  - Smaller batch size: faster parameter updates.
  - GPUs usually allow for larger batch sizes.

Fluctuations

- Stochastic gradient descent introduces noise into the training process. Each update does not necessarily point toward the direction of true gradient.
- It may even increase the loss!
- Reducing learning rate reduces fluctuations (but may slow down convergence) — a trade-off between learning rate and batch size.

Momentum

One simple idea to limit the "zig-zag" behavior is to choose a step direction which is an average of the previous updates.

Moving average: Consider a set of numbers $x_1, \ldots, x_t$. Then the average a_t is given by

$$a_t = \frac{1}{t}\sum_{\tau=1}^{t} x_\tau = \frac{1}{t}\left(x_t + (t-1)a_{t-1}\right) = \epsilon_t x_t + \mu_t a_{t-1}$$

for suitably chosen ϵ_t and $0 \leq \mu_t \leq 1$. If μ_t is small then the more recent x contribute more strongly to the moving average.

Momentum

The idea of momentum is to use a form of moving average to the updates:

$$\tilde{\mathbf{g}}_{t+1} = \mu_t \tilde{\mathbf{g}}_t - \epsilon \nabla f(\boldsymbol{\theta}_t)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \tilde{\mathbf{g}}_{t+1}$$

Instead of using the simple update rule $-\epsilon \nabla f(\boldsymbol{\theta})$, we use the moving average $\tilde{\mathbf{g}}_{t+1}$ to define our step direction.

Momentum

- Momentum can increase the speed of convergence since, for smooth objectives, as we get close to the minimum the gradient decreases and standard gradient descent starts to slow down.
- If the learning rate is too large, standard gradient descent may oscillate, but momentum may reduce oscillations by going in the average direction.
- However, the momentum parameter μ may need to be reduced with the iteration count to ensure convergence.
- Particularly useful when the gradient is noisy. By averaging over previous gradients, the noise 'averages' out and the moving average direction can be much less noisy.
- Momentum is also useful to avoid saddles (a point where the gradient is zero, but the objective function is not a minimum, such as the function x^3 at the origin) since typically the momentum will carry you over the saddle.

Saddle points and local optima

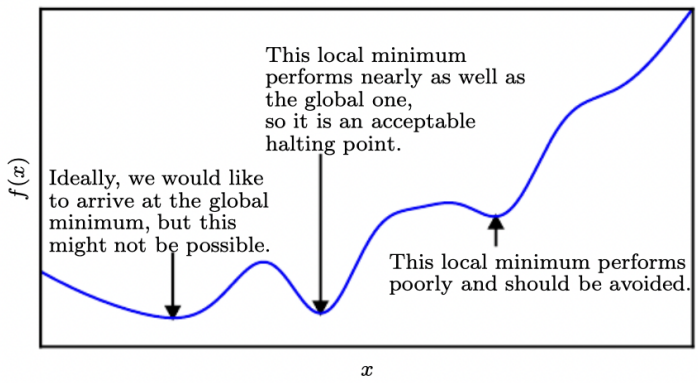


Figure: Goodfellow et al., Ch. 4

Adaptive learning rates

Adagrad

Often we have settings where the objective function is very **sensitive** to small changes in some directions, but very **insensitive** in others. Adagrad sets a different learning rate in every direction:

$$g_{t,i} = (\nabla L(\boldsymbol{\theta}^t))_i \qquad G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} g_{t,i}$$

Adagrad

Often we have settings where the objective function is very **sensitive** to small changes in some directions, but very **insensitive** in others. Adagrad sets a different learning rate in every direction:

$$g_{t,i} = (\nabla L(\boldsymbol{\theta}^t))_i \qquad G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} g_{t,i}$$

Good: Per-parameter learning rates scale inversely to squared magnitude of gradient. Using a default choice of $\eta = 1.0$ or maybe $\eta = 0.1$ works pretty often.

Adagrad

Often we have settings where the objective function is very **sensitive** to small changes in some directions, but very **insensitive** in others. Adagrad sets a different learning rate in every direction:

$$g_{t,i} = (\nabla L(\boldsymbol{\theta}^t))_i \qquad G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} g_{t,i}$$

Good: Per-parameter learning rates scale inversely to squared magnitude of gradient. Using a default choice of $\eta = 1.0$ or maybe $\eta = 0.1$ works pretty often.

Not good: The accumulated gradients eventually make learning very slow and the learning rate never recovers.

RMSprop

RMSprop fixes the problem with Adagrad vanishing learning rates by using moving average of past squared gradients instead (like momentum is doing for the gradients themselves):

$$s_{t,i} = \gamma s_{t-1,i} + (1-\gamma) g_{t,i}^2$$
$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{s_{t,i} + \varepsilon}} g_{t,i}$$

The algorithm was first presented in Geoff Hinton Coursera course (Lecture 6e).

(Note: You also might see this quoted as using a fixed "rolling window" average of the past few gradient updates, rather than a moving average.)

Adam

Adam is a very popular optimize, specifically for deep learning models. Essentially, it combines RMSprop with momentum:

$$m_{t,i} = \beta m_{t-1,i} + (1-\beta) g_{t,i} \qquad \hat{m}_{t,i} = m_{t,i}/(1-\beta^t)$$
$$s_{t,i} = \gamma s_{t-1,i} + (1-\gamma) g_{t,i}^2 \qquad \hat{s}_{t,i} = s_{t,i}/(1-\gamma^t)$$
$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{\hat{s}_{t,i} + \varepsilon}} \hat{m}_{t,i}$$

The $\hat{m}$ and $\hat{s}$ are "bias-corrected", accounting for the fact that we typically initialize with $\mathbf{m}_0 = \mathbf{s}_0 = \mathbf{0}$, which is intended to help in the early iterations.

All of these algorithms are implemented in common ML and DL libraries.

Higher order methods

Second-order methods

Instead of a first-order Taylor expansion, we can use a second-order one, incorporating the Hessian. This is like assuming the function is roughly quadratic around the current $\boldsymbol{\theta} \in \mathbb{R}^D$, with

$$f(\boldsymbol{\theta} + \mathbf{s}) \approx f(\boldsymbol{\theta}) + \mathbf{s}^\top \nabla f(\boldsymbol{\theta}) + \frac{1}{2}\mathbf{s}^\top \mathbf{H}(\boldsymbol{\theta})\mathbf{s}$$

where $\mathbf{H}(\boldsymbol{\theta})$ is the matrix of all partial derivatives of $f(\boldsymbol{\theta})$, i.e.

$$\mathbf{H}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f}{\partial \theta_1^2} & \cdots & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial \theta_D \partial \theta_1} & \cdots & \frac{\partial^2 f}{\partial \theta_D^2} \end{bmatrix}.$$

Newton's method

For **Newton's method**, we consider

$$f(\boldsymbol{\theta} + \mathbf{s}) \approx f(\boldsymbol{\theta}) + \mathbf{s}^\top \nabla f(\boldsymbol{\theta}) + \frac{1}{2}\mathbf{s}^\top \mathbf{H}(\boldsymbol{\theta})\mathbf{s}$$

and find the step direction by minimizing it with respect to $\mathbf{s}$. Differentiating the right hand side yields

$$\nabla_{\mathbf{s}} f(\boldsymbol{\theta} + \mathbf{s}) \approx \nabla f(\boldsymbol{\theta}) + \mathbf{H}(\boldsymbol{\theta})\mathbf{s};$$

setting equal to zero and solving for $\mathbf{s}$, we find the minimum to be at

$$\mathbf{s} = -\mathbf{H}(\boldsymbol{\theta})^{-1} \nabla f(\boldsymbol{\theta}).$$

Often, a learning rate $\epsilon \in (0, 1)$ is included to dampen the update,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \mathbf{H}^{-1} \nabla f(\boldsymbol{\theta}_t).$$

Newton's method

When we ran gradient descent, we made a linear approximation to $f(\boldsymbol{\theta})$ and then stepped a short distance ϵ.

Here, we instead make a quadratic approximation to $f(\boldsymbol{\theta})$, and move *directly to its minimum*. The rough idea (Newton's method):

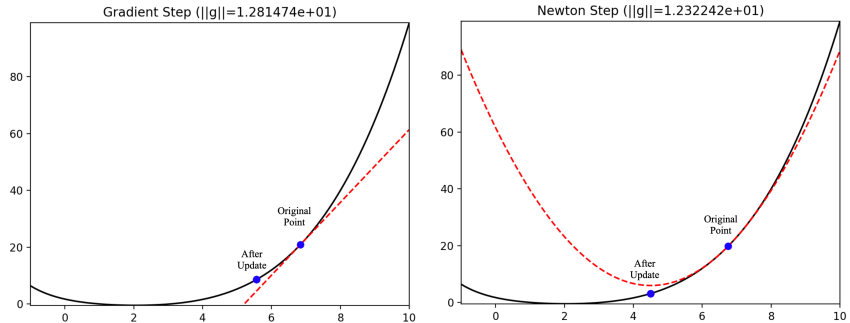


Figure: Kilian Weinberger

Comments on Newton's method

- Storing the Hessian and solving the linear system $\mathbf{H}^{-1}\nabla f$ is very expensive. One way to make it cheaper can be to use only a diagonal approximation to the Hessian.
- Newton's method is not guaranteed to produce a downhill step!
- This only happens (for sure) if ϵ is small and $\mathbf{H}$ is positive definite.
- If $\mathbf{H}$ is positive definite, one can use a line search in the direction $\mathbf{H}^{-1}\nabla f$ to ensure we go downhill and make a non-trivial jump.
- Not very useful in general for stochastic objectives.

Quasi-Newton methods

Quasi-Newton methods avoid computing the Hessian, using an approximation instead, finding a matrix $\mathbf{B}$ which satisfies

$$\nabla f(\boldsymbol{\theta}_{t+1}) = \nabla f(\boldsymbol{\theta}_t) + \mathbf{B}(\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t),$$

corresponding to the Taylor expansion of the gradient. This leads to an optimization step

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \mathbf{B}_t^{-1} \nabla f(\boldsymbol{\theta}_t).$$

- In Broyden-Fletcher-Goldfarb-Shannon (BFGS), an approximate inverse Hessian $\mathbf{B}_t^{-1}$ is formed iteratively.
- L-BFGS is a popular practical method that limits the memory requirement.
- This update sort of looks like the Adagrad / RMSProp updates. . . !

Outer product approximations

Suppose we are optimizing a function that decomposes into a sum of squared errors, e.g.

$$L(\boldsymbol{\theta}) = \frac{\lambda}{2} \sum_{n=1}^{N} (y_n - f(\boldsymbol{\theta}, \mathbf{x}_n))^2.$$

The Hessian of this overall loss function has entries

$$\frac{\partial L}{\partial \theta_i \partial \theta_j} = \lambda \sum_{n=1}^{N} \left((f(\boldsymbol{\theta}, \mathbf{x}_n) - y_n) \frac{\partial f_n}{\partial \theta_i \partial \theta_j} + \frac{\partial f_n}{\partial \theta_i} \frac{\partial f_n}{\partial \theta_j} \right).$$

Near the minimum, the first term will be small, as $f(\boldsymbol{\theta}, \mathbf{x}_n) \approx y_n$. This suggests that we can construct a Hessian approximation by using the outer product of the gradients,

$$\mathbf{B} = \lambda \sum_{i=1}^{N} \nabla f_n(\boldsymbol{\theta}) \nabla f_n(\boldsymbol{\theta})^\top.$$

Summary

- Gradient descent: general-purpose optimization of real-valued scalar functions
- Stochastic / mini-batch methods can speed up runtime by fast parallel estimation of an approximate step direction
- Momentum and adaptive learning rates can speed up rate of optimization (and will be essential for training deep networks with millions of parameters!)
- Second order methods are memory-intensive and not always "worth it" on stochastic objectives, but can converge in far fewer iterations. There's some current work on approximating Hessians for deep networks that we may discuss later.