ENM 531: Data-driven Modeling and Probabilistic Scientific Computing

Lecture #5: Optimization

Paris Perdikaris
February 4, 2021

Maximum likelihood estimation

$$\theta_{\mathrm{MLE}} = \arg\max_{\theta \in \Theta} p(\mathcal{D}|\theta)$$

Objectives

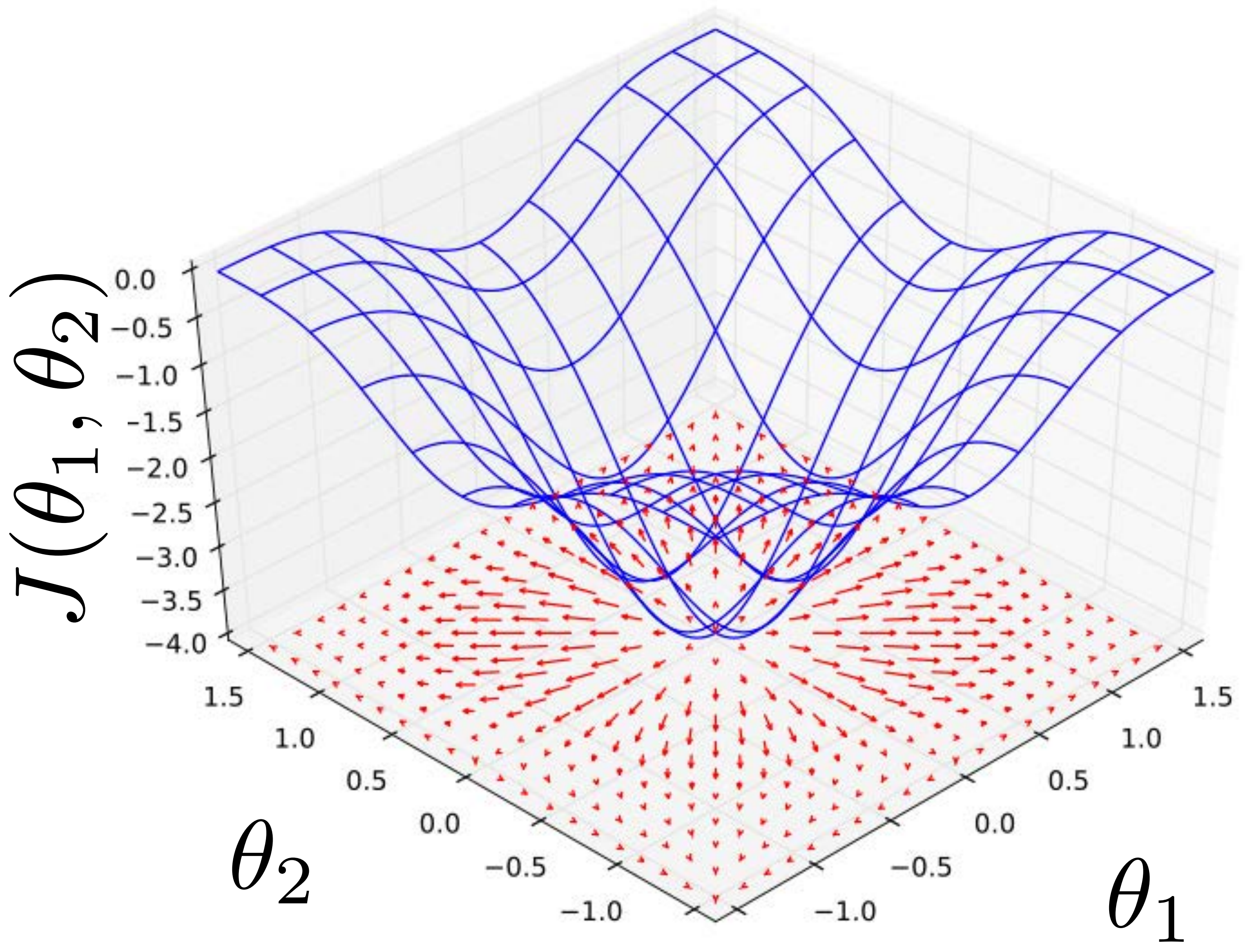
At its core, machine learning is all about integration (e.g., computing expectations, etc.) and ***optimization***. Today we'll revisit some basic concepts in optimization, and introduce them in the context of training machine learning algorithms.

Specifically, we'll cover:

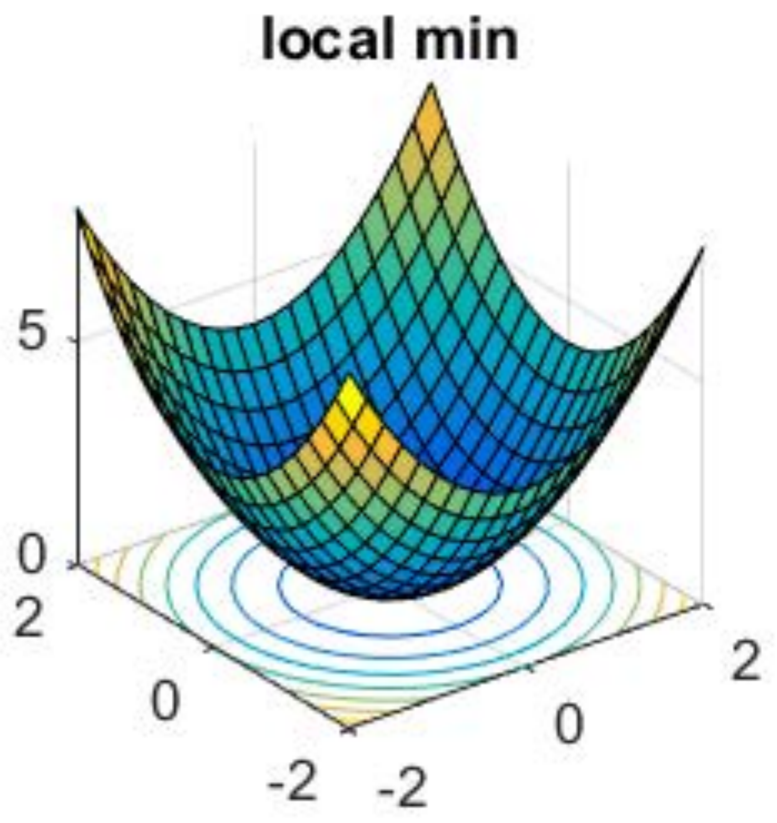
- The definition of gradients and Hessians.
- The gradient descent algorithm.
- Newton's algorithm.
- Applications to linear regression.
- Stochastic gradient descent.
- Modern variants of stochastic gradient descent.

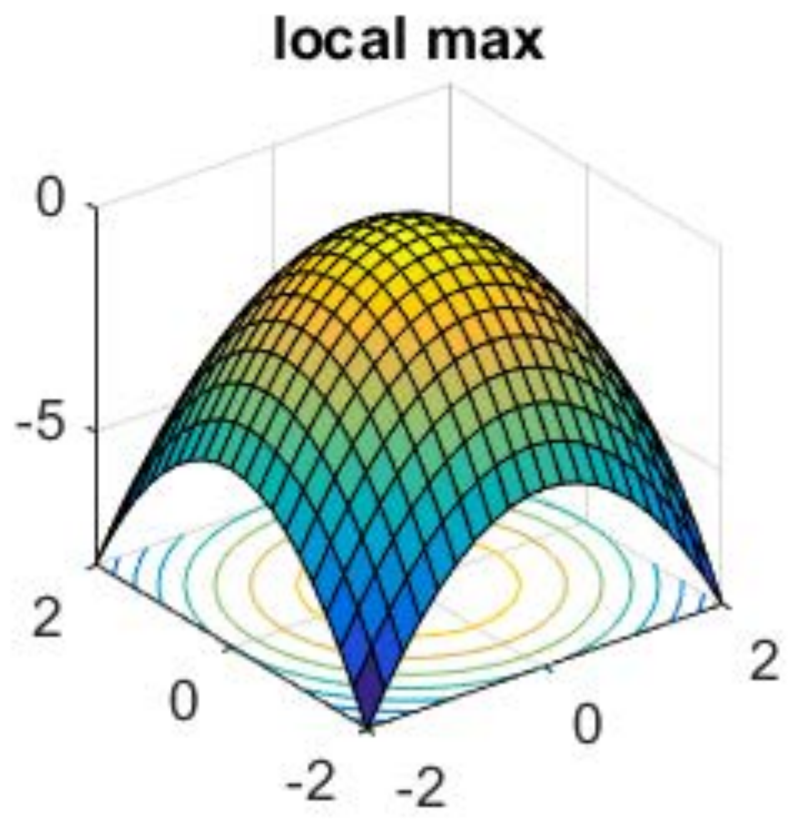
Gradients

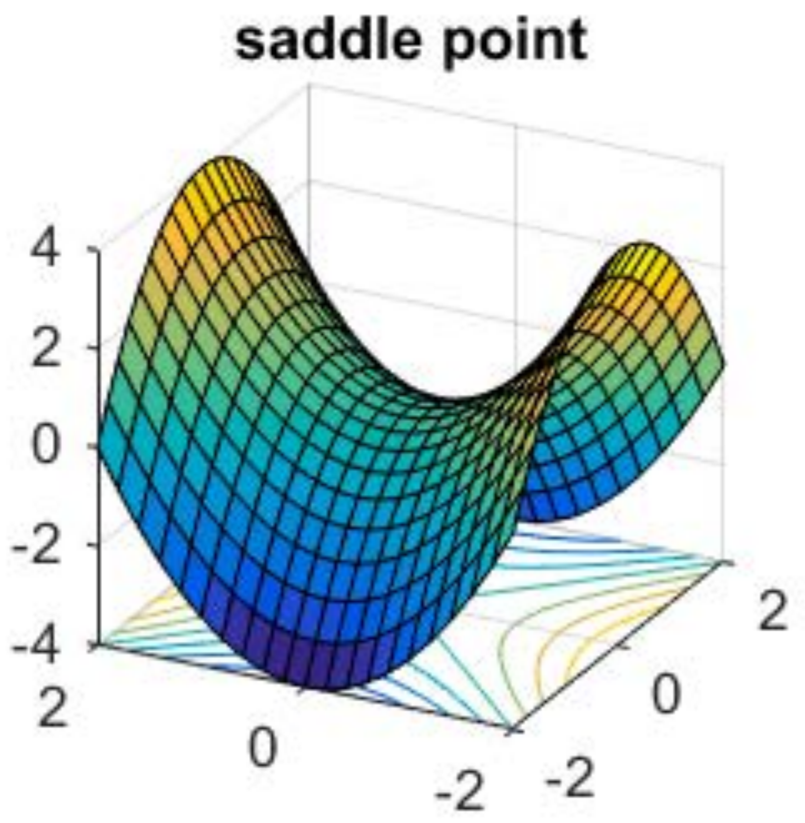
$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_1} \\ \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_n} \end{bmatrix}$$



Minima, maxima, and saddle points



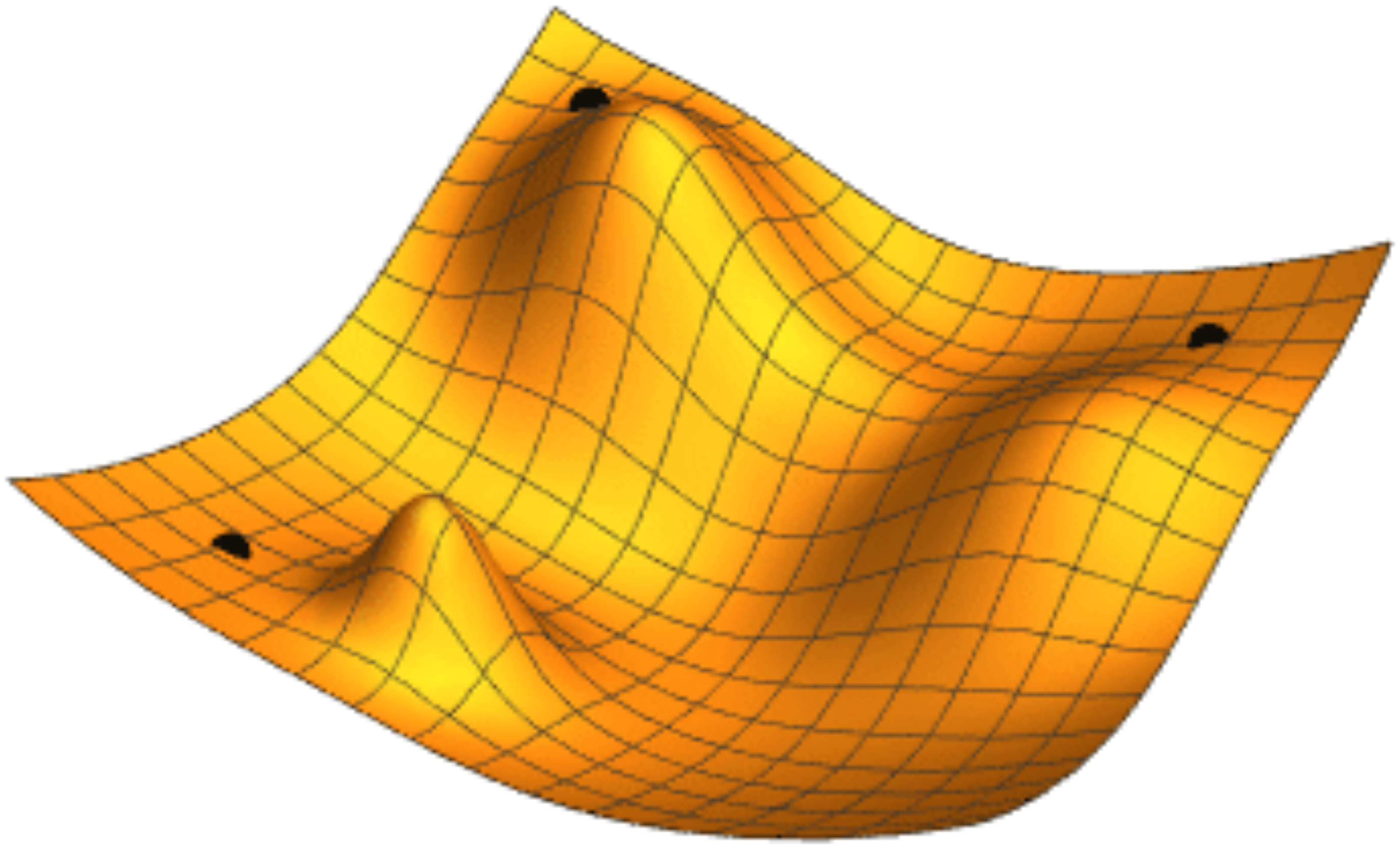




Gradient descent

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} J(\boldsymbol{\theta})$$

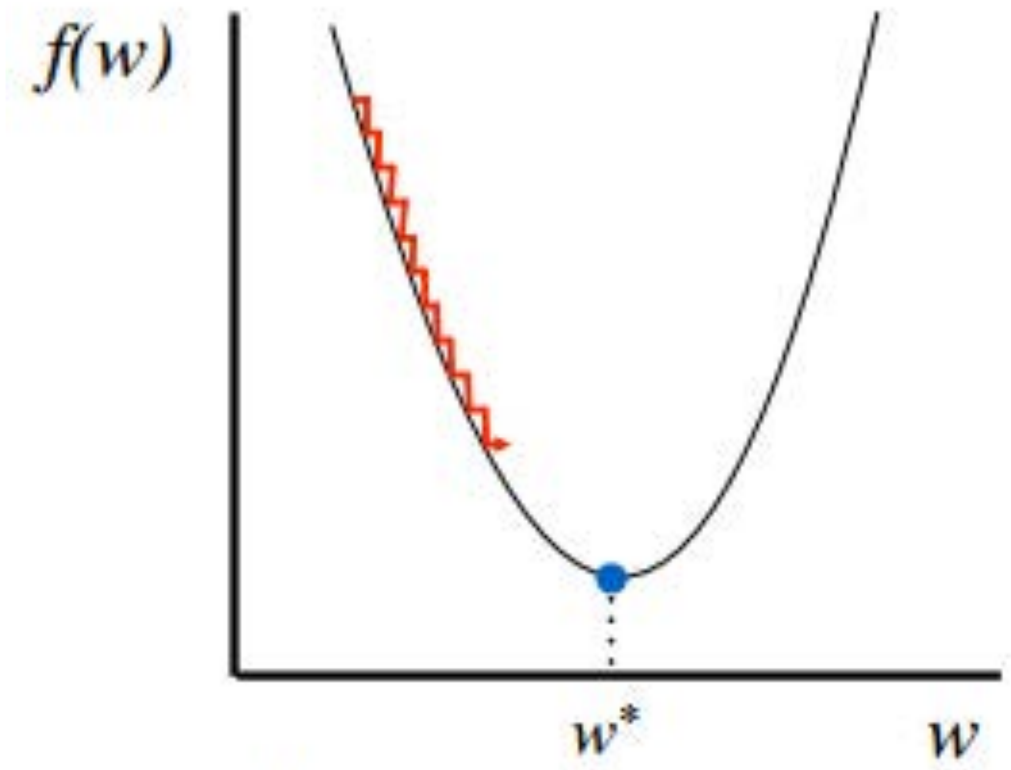
$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$



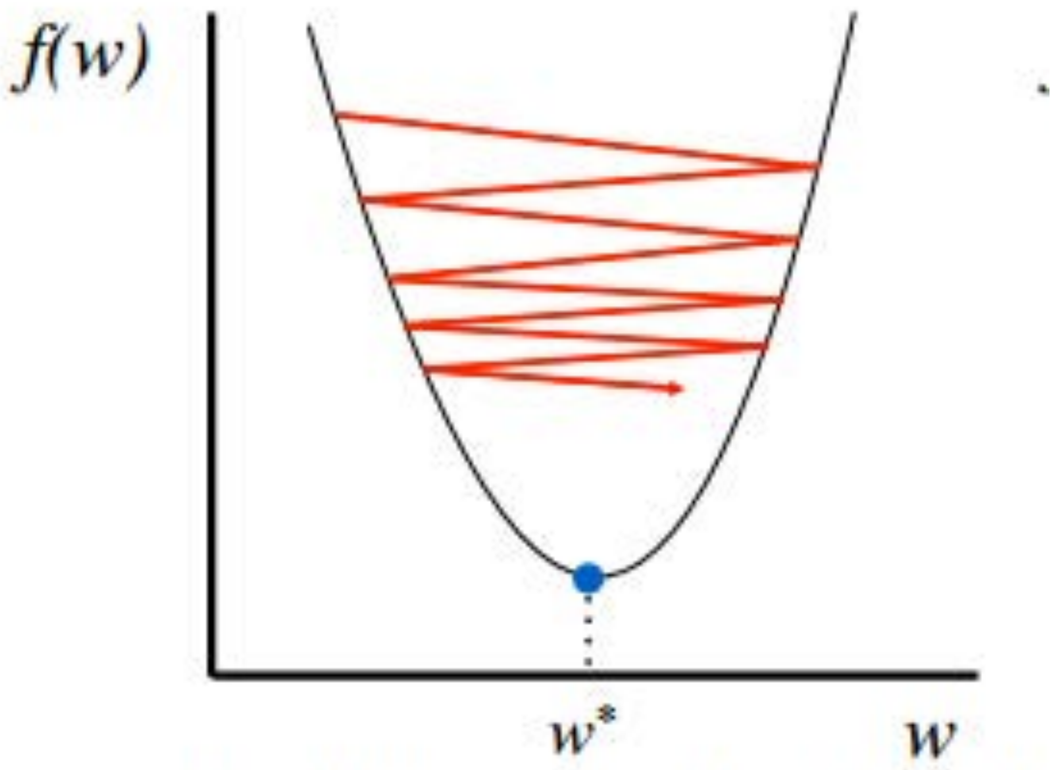
Gradient descent

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Effect of the learning rate



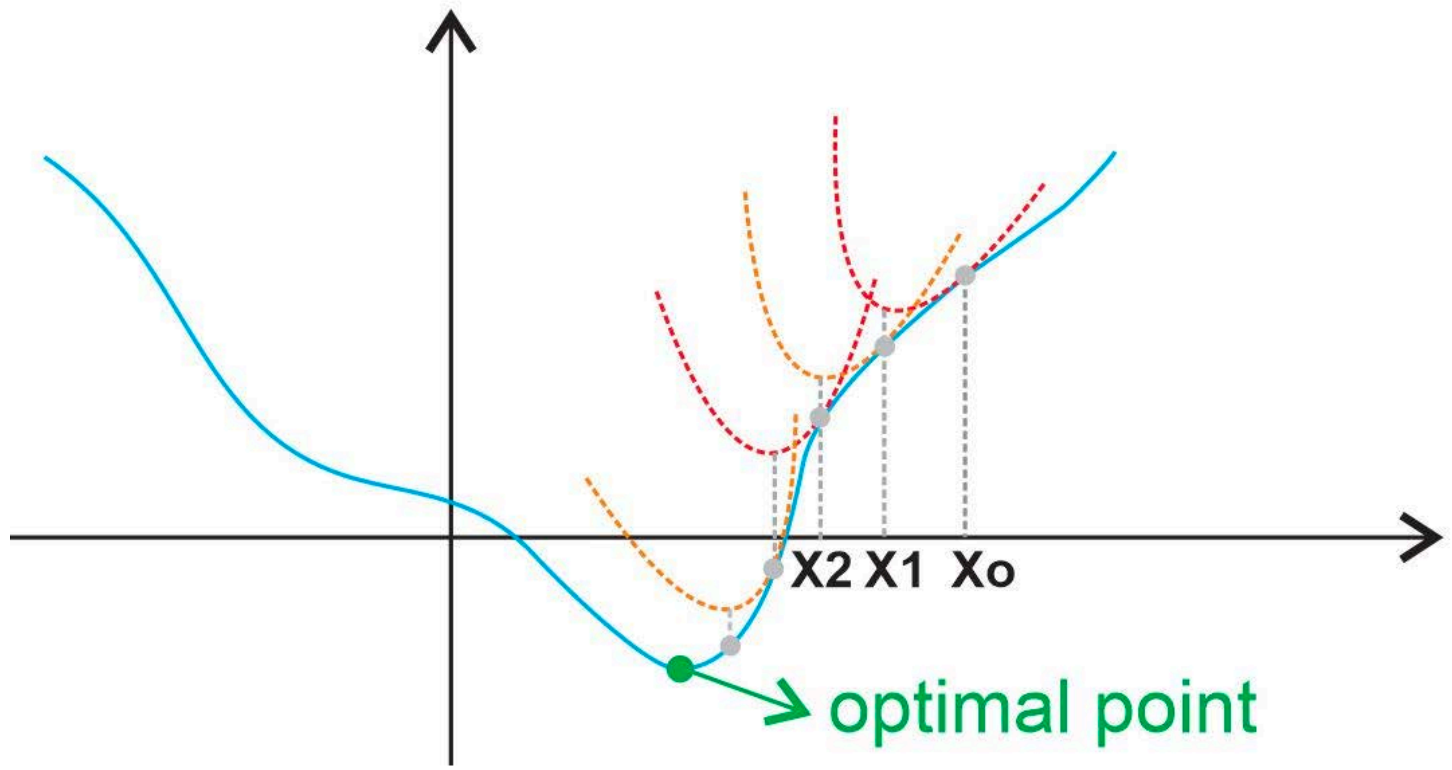
Too small: converge very slowly



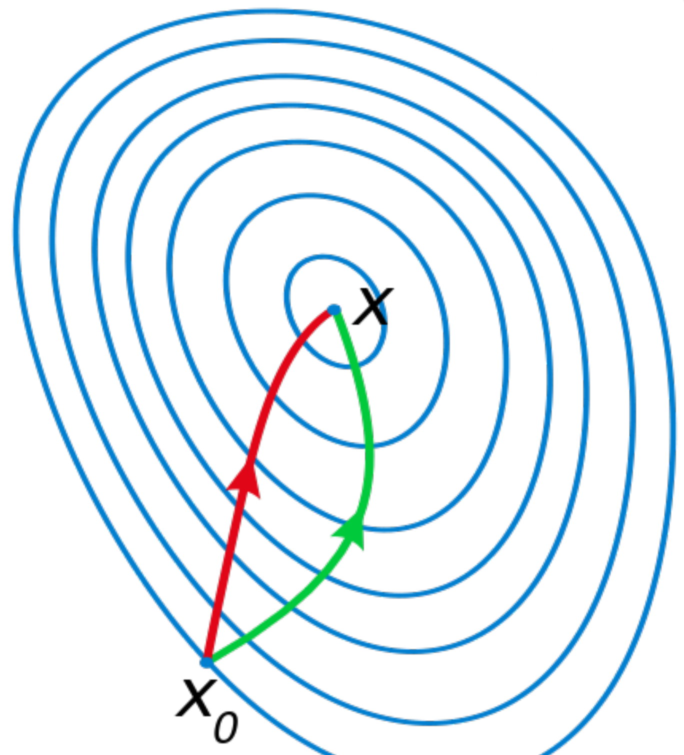
Too big: overshoot and even diverge

Hessian

$$\nabla^2_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1^2} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2^2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_2 \partial \theta_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_1} & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d \partial \theta_2} & \cdots & \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \theta_d^2} \end{bmatrix}$$

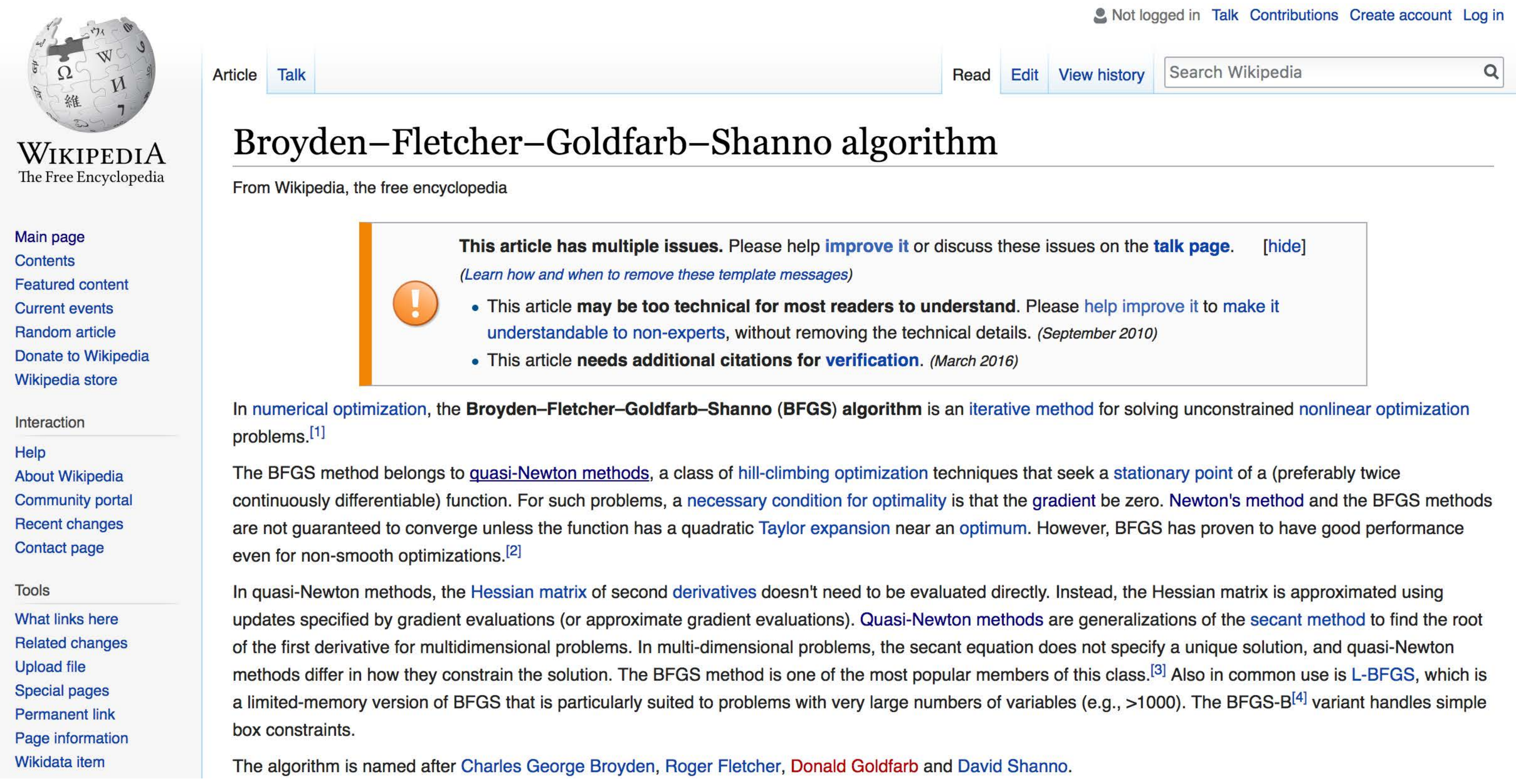


Local quadratic approximation
of the loss == Newton's method



Gradient descent vs Newton

BFGS



Broyden–Fletcher–Goldfarb–Shanno algorithm

From Wikipedia, the free encyclopedia

This article has multiple issues. Please help **improve it** or discuss these issues on the **talk page**. [hide]

(Learn how and when to remove these template messages)

- This article **may be too technical for most readers to understand**. Please help improve it to make it understandable to non-experts, without removing the technical details. *(September 2010)*
- This article **needs additional citations for verification**. *(March 2016)*

In numerical optimization, the **Broyden–Fletcher–Goldfarb–Shanno** (**BFGS**) **algorithm** is an iterative method for solving unconstrained nonlinear optimization problems.[1]

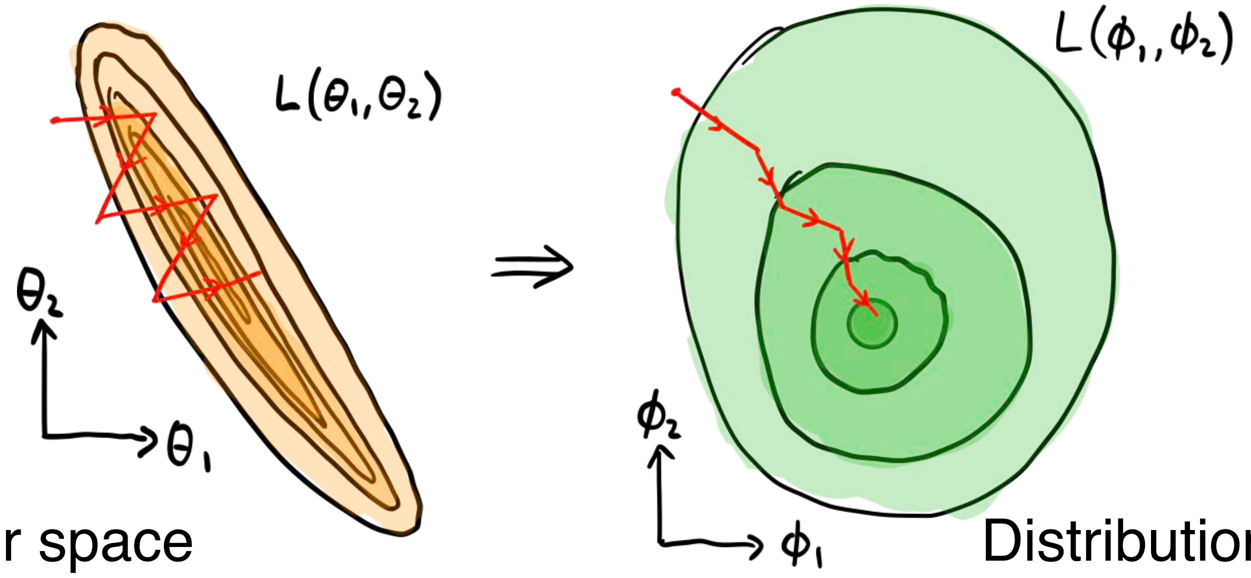
The BFGS method belongs to quasi-Newton methods, a class of hill-climbing optimization techniques that seek a stationary point of a (preferably twice continuously differentiable) function. For such problems, a necessary condition for optimality is that the gradient be zero. Newton's method and the BFGS methods are not guaranteed to converge unless the function has a quadratic Taylor expansion near an optimum. However, BFGS has proven to have good performance even for non-smooth optimizations.[2]

In quasi-Newton methods, the Hessian matrix of second derivatives doesn't need to be evaluated directly. Instead, the Hessian matrix is approximated using updates specified by gradient evaluations (or approximate gradient evaluations). Quasi-Newton methods are generalizations of the secant method to find the root of the first derivative for multidimensional problems. In multi-dimensional problems, the secant equation does not specify a unique solution, and quasi-Newton methods differ in how they constrain the solution. The BFGS method is one of the most popular members of this class.[3] Also in common use is L-BFGS, which is a limited-memory version of BFGS that is particularly suited to problems with very large numbers of variables (e.g., >1000). The BFGS-B[4] variant handles simple box constraints.

The algorithm is named after Charles George Broyden, Roger Fletcher, Donald Goldfarb and David Shanno.

Gradient descent vs natural gradient descent

Motivation: If our objective is to minimize the loss function (maximizing the likelihood), then it is natural that we taking step in the space of all possible likelihoods, realizable by the parameters θ. As the likelihood function itself is a probability distribution, we call this space distribution space. Thus it makes sense to take the steepest descent direction in this distribution space instead of parameter space.



Parameter space

Distribution space

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

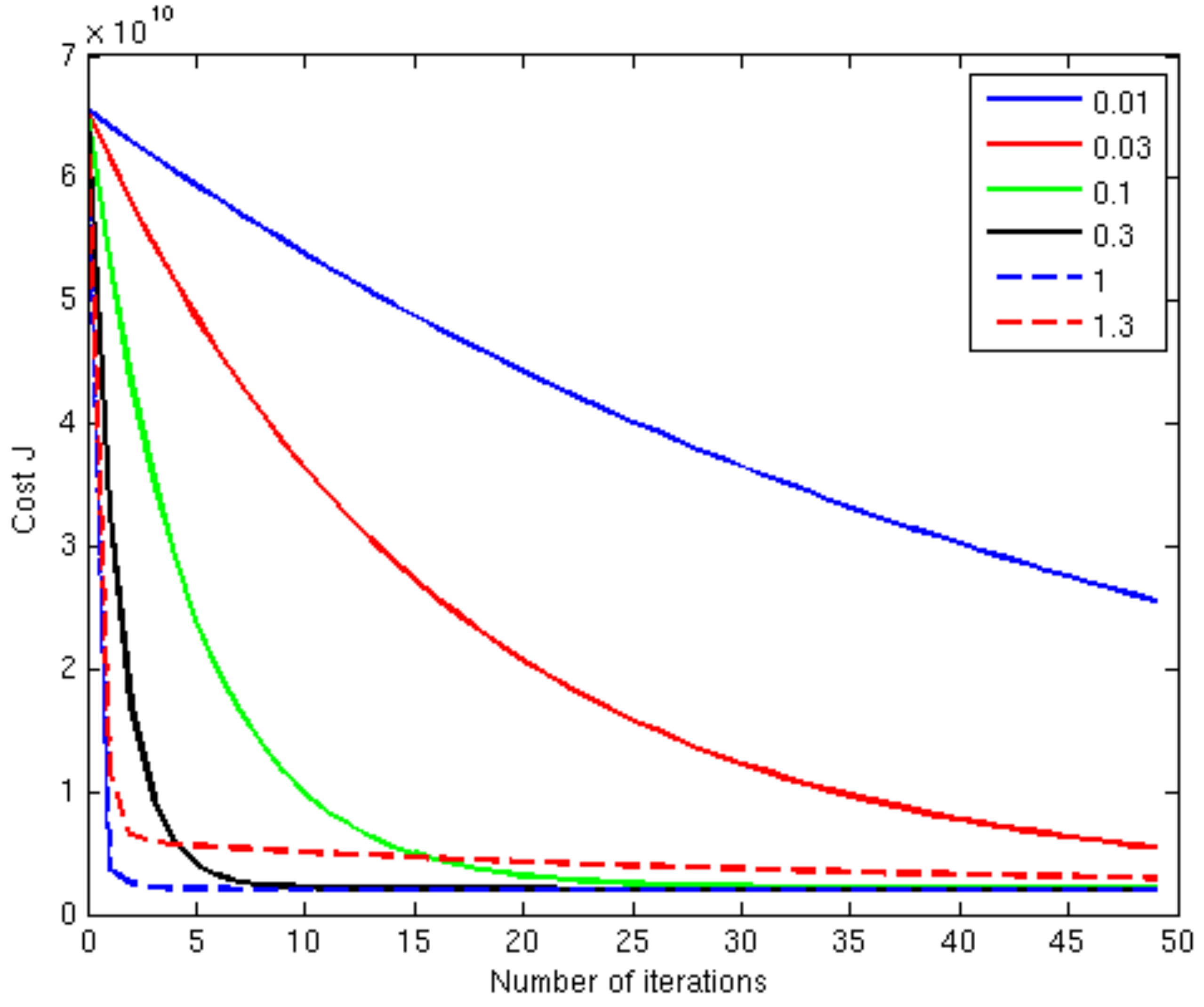
$$\theta_{n+1} = \theta_n - \eta F^{-1} \nabla \mathscr{L}(\theta)$$

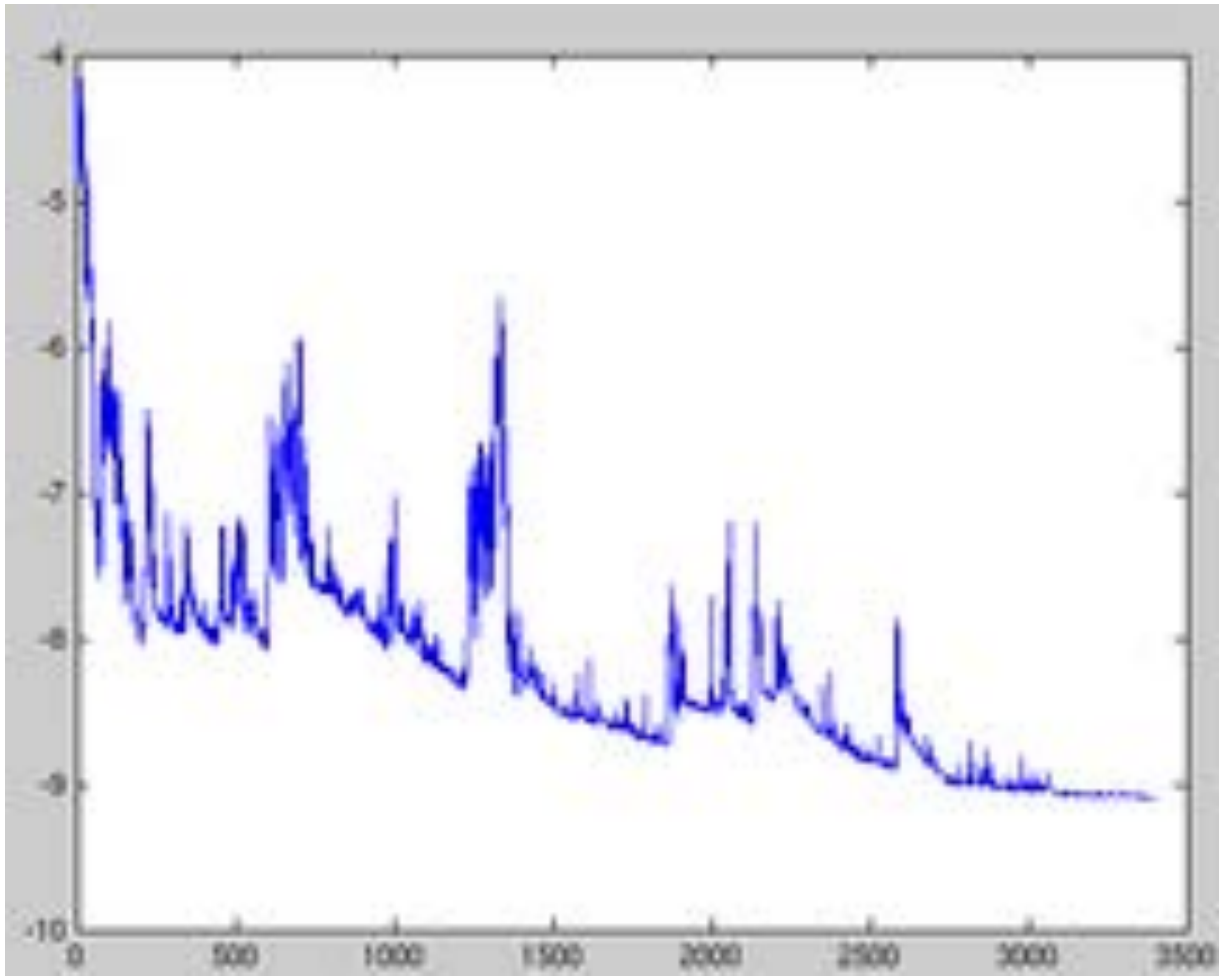
Cramer-Rao bound: The inverse of the Fisher information is a lower bound on the variance of any unbiased estimator of θ

Fisher Information Matrix == negative expected Hessian of log likelihood

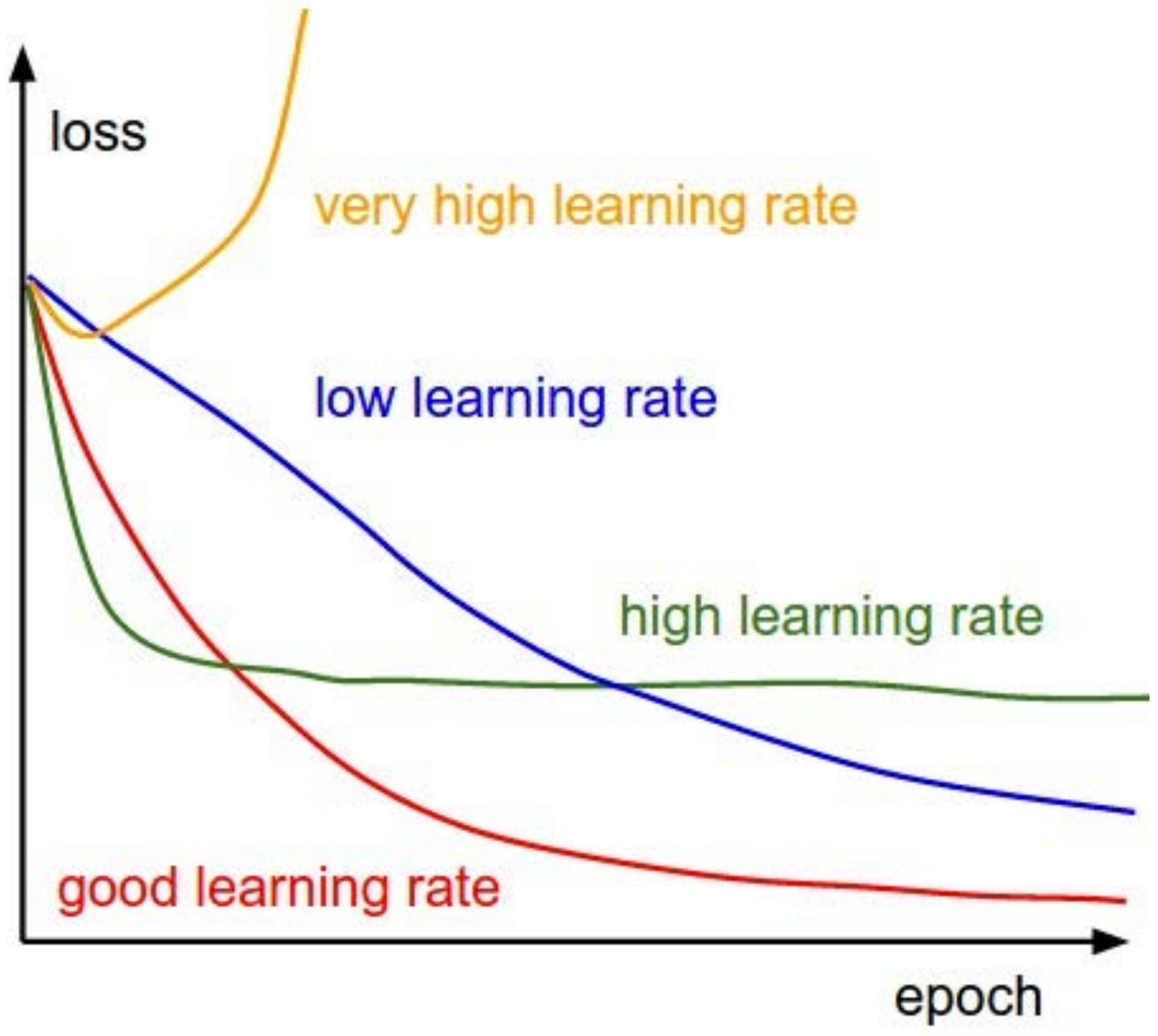
$$\begin{aligned}
\mathrm{H}_{\mathrm{KL}[p(x|\theta)\,\|\,p(x|\theta')]} &= -\int p(x|\theta)\ \nabla^2_{\theta'} \log p(x|\theta')\big|_{\theta'=\theta}\, \mathrm{d}x \\
&= -\int p(x|\theta)\, \mathrm{H}_{\log p(x|\theta)}\, \mathrm{d}x \\
&= -\mathop{\mathbb{E}}_{p(x|\theta)} [\mathrm{H}_{\log p(x|\theta)}] \\
&= \mathbf{F}\,.
\end{aligned}$$

Gradient descent vs SGD





Gradient descent vs SGD



Remarks

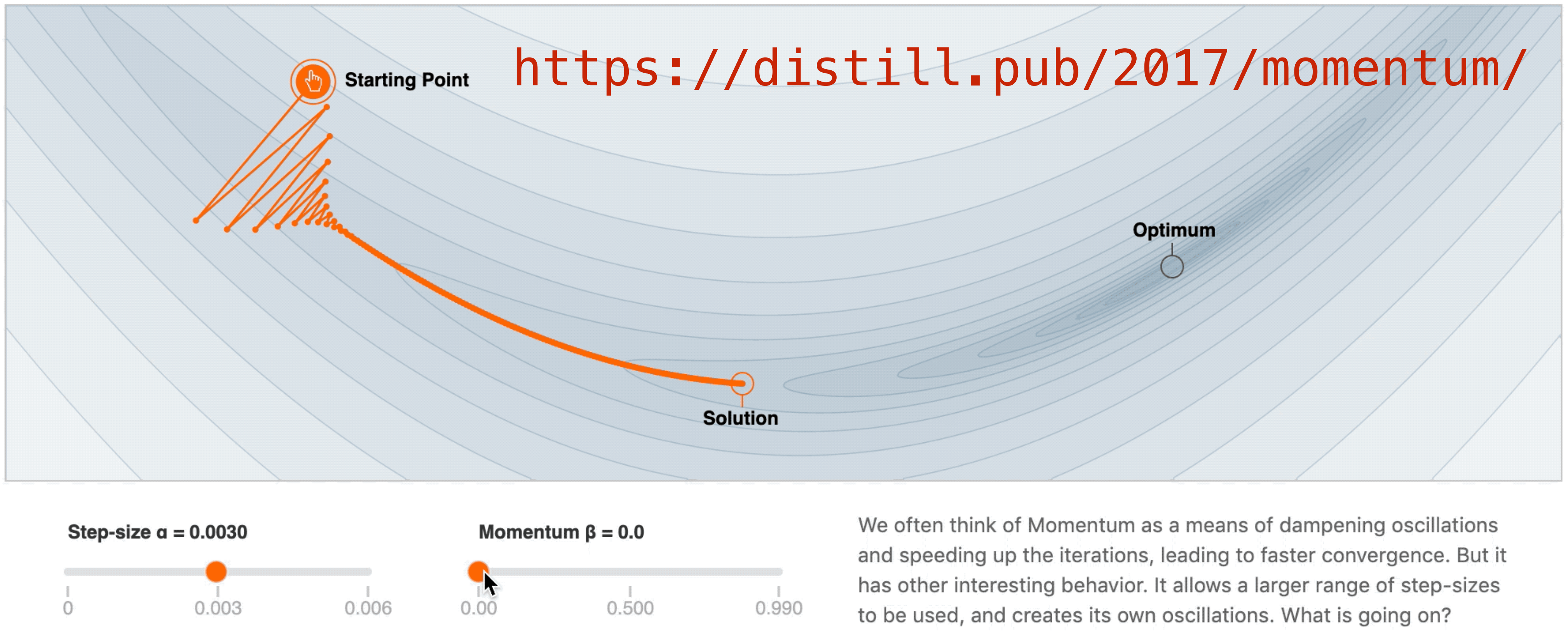
Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

- Choosing a proper learning rate can be difficult.
- Learning rate schedules (i.e., adjusting the learning rate during training) has to be defined in advance and it is thus unable to adapt to a dataset's characteristics.
- The same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin et al. argue that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

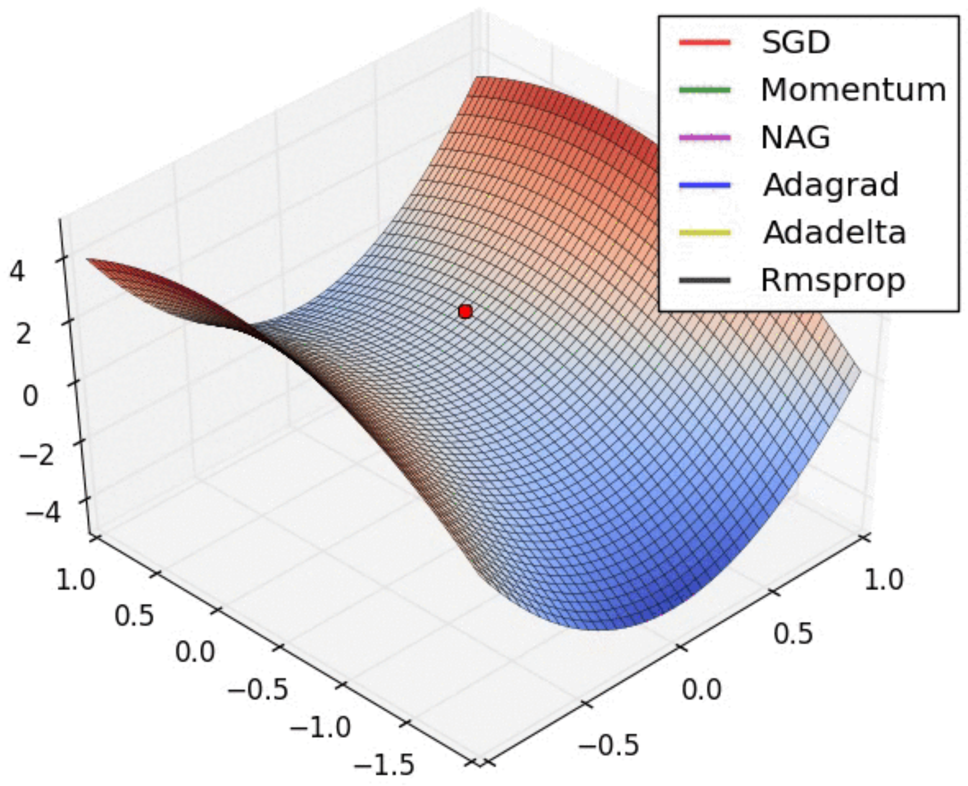
Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

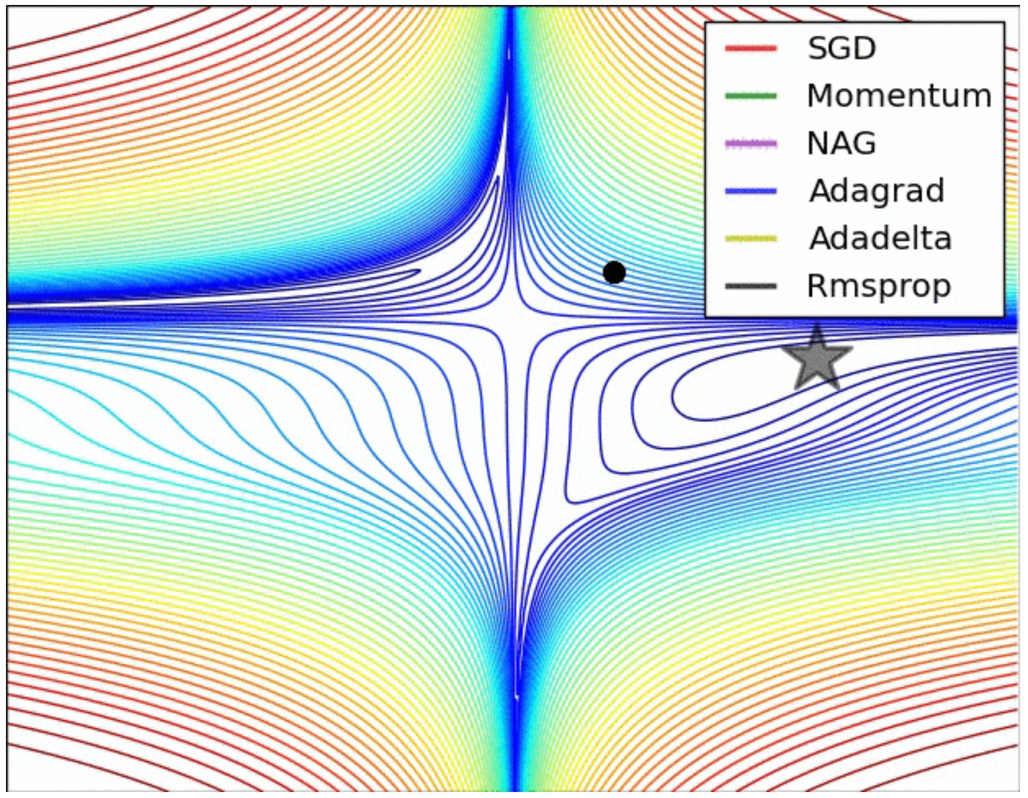
Dauphin, Y., et. al. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1–14.

Modern SGD variants



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?





http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/

http://ruder.io/optimizing-gradient-descent/

*animation credit: Alec Redford

Automatic differentiation

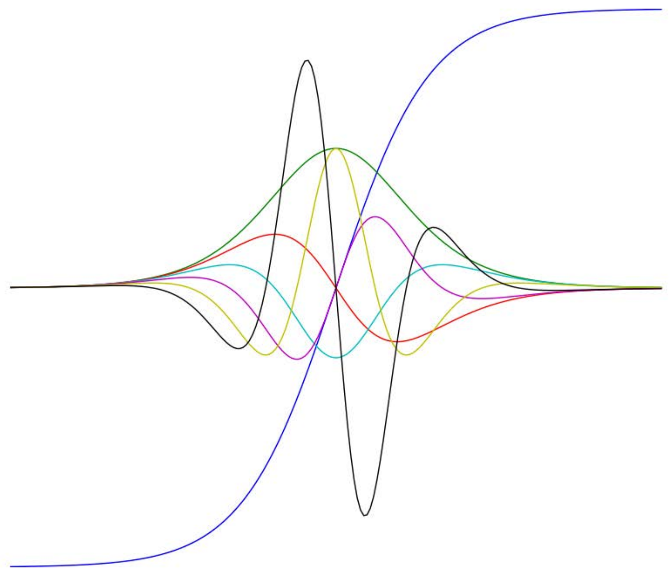
It is one of the most useful, elegant, and perhaps under-utilized tools in modern scientific computing!

Applications:

- real-parameter optimization (many good methods are gradient-based)
- sensitivity analysis (local sensitivity = ∂(result)/∂(input))
- physical modeling (forces are derivatives of potentials; equations of motion are derivatives of Lagrangians and Hamiltonians; etc.)
- probabilistic inference (e.g., Hamiltonian Monte Carlo)
- machine learning
- and who knows how many other scientific computing applications.

```
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt

x = np.linspace(-7, 7, 200)
plt.plot(x, np.tanh(x),
         x, grad(np.tanh)(x),
         x, grad(grad(np.tanh))(x),
         x, grad(grad(grad(np.tanh)))(x),
         x, grad(grad(grad(grad(np.tanh))))(x),
         x, grad(grad(grad(grad(grad(np.tanh)))))(x),
         x, grad(grad(grad(grad(grad(grad(np.tanh))))))(x))
```



Automatic differentiation

The chain rule, forward and reverse accumulation [edit]

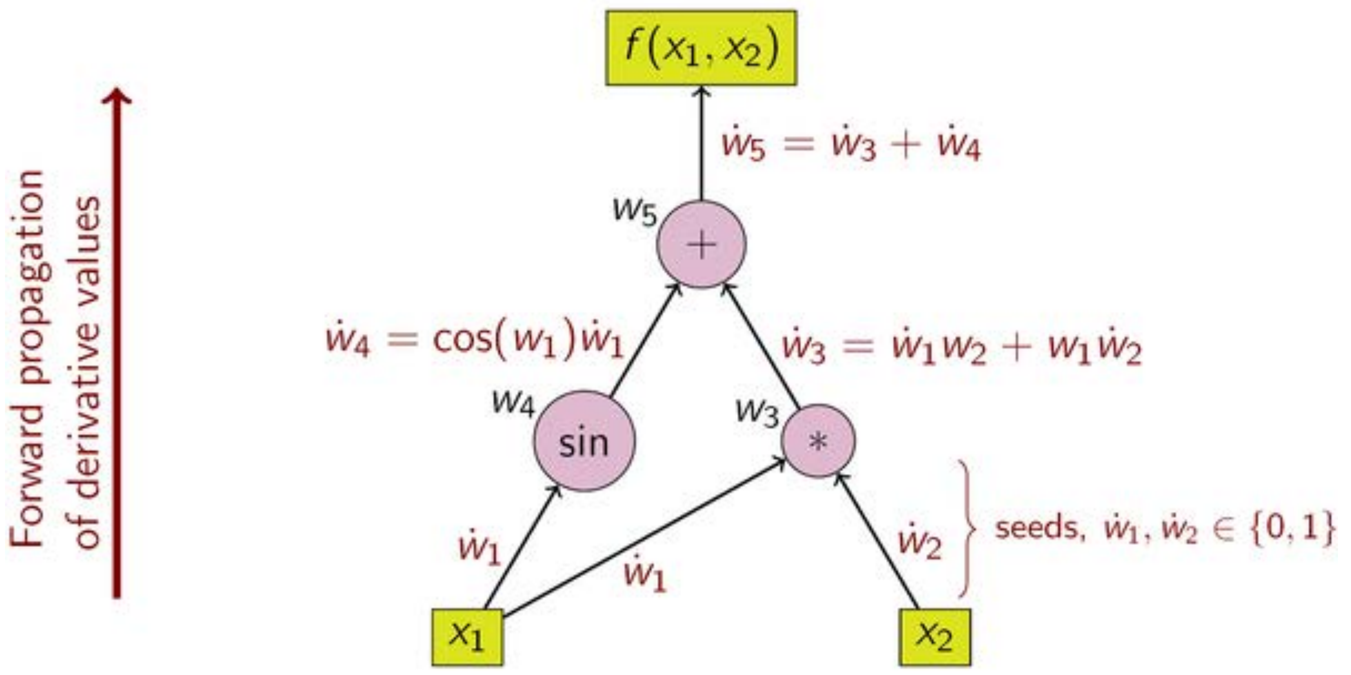
Fundamental to AD is the decomposition of differentials provided by the chain rule. For the simple composition $y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$ the chain rule gives

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

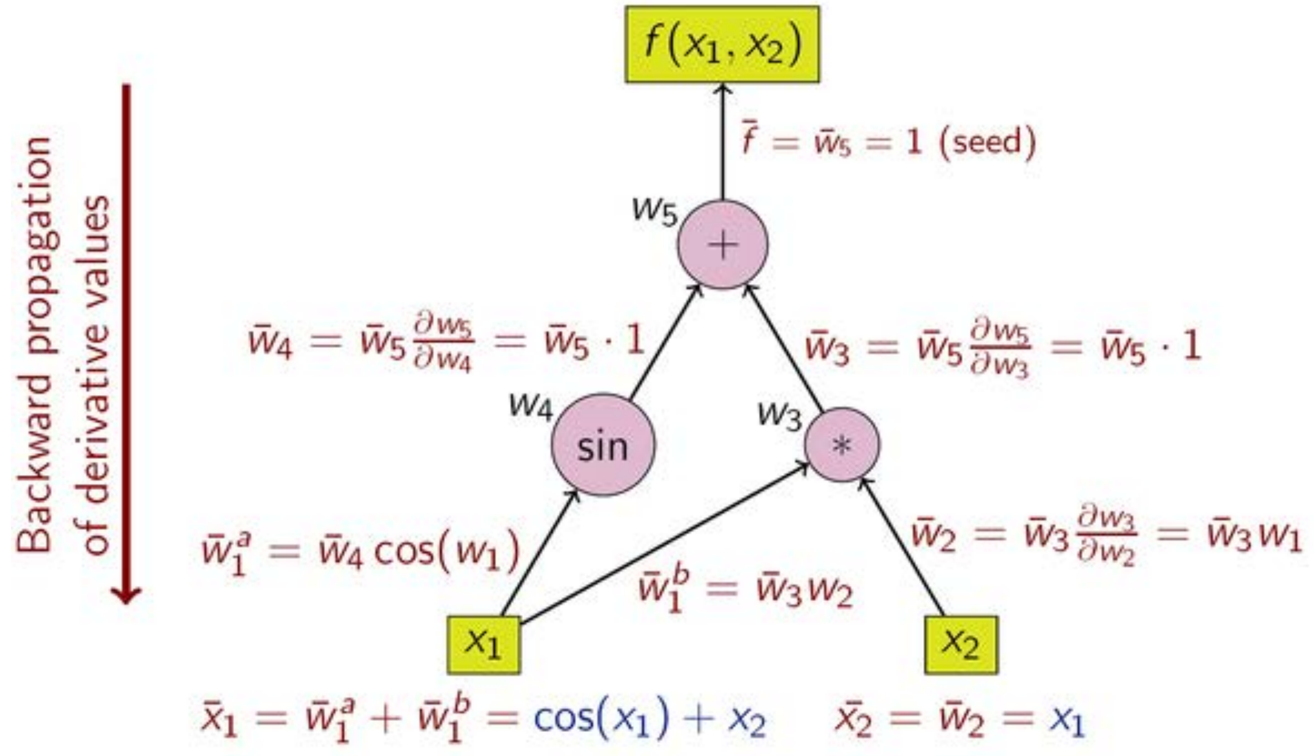
Usually, two distinct modes of AD are presented, **forward accumulation** (or **forward mode**) and **reverse accumulation** (or **reverse mode**). Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first compute dw_1/dx and then dw_2/dx and at last dy/dx), while reverse accumulation has the traversal from outside to inside (first compute dy/dw_2 and then dy/dw_1 and at last dy/dx). More succinctly,

1. **forward accumulation** computes the recursive relation: $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}$ with $w_3 = y$, and,
2. **reverse accumulation** computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$ with $w_0 = x$.

Example: $z = f(x_1, x_2) = x_1 x_2 + \sin x_1$



Forward propagation of derivative values
$f(x_1, x_2)$
$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$
w_5
+
$\dot{w}_4 = \cos(w_1)\dot{w}_1$
$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2$
w_4
sin
w_3
*
$\dot{w}_1$
$\dot{w}_1$
$\dot{w}_2$
seeds, $\dot{w}_1, \dot{w}_2 \in \{0, 1\}$
x_1
x_2



Backward propagation of derivative values
$f(x_1, x_2)$
$\bar{f} = \bar{w}_5 = 1$ (seed)
w_5
+
$\bar{w}_4 = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = \bar{w}_5 \cdot 1$
$\bar{w}_3 = \bar{w}_5 \frac{\partial w_5}{\partial w_3} = \bar{w}_5 \cdot 1$
w_4
sin
w_3
*
$\bar{w}_1^a = \bar{w}_4 \cos(w_1)$
$\bar{w}_1^b = \bar{w}_3 w_2$
$\bar{w}_2 = \bar{w}_3 \frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1$
x_1
x_2
$\bar{x}_1 = \bar{w}_1^a + \bar{w}_1^b = \cos(x_1) + x_2$
$\bar{x}_2 = \bar{w}_2 = x_1$

Automatic differentiation

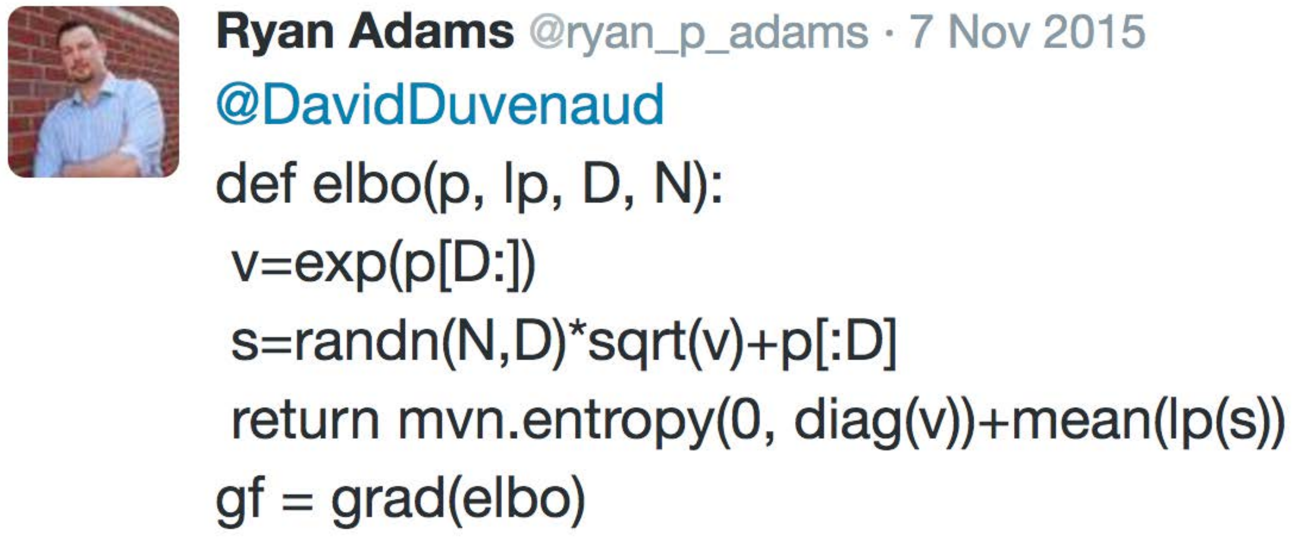
- We live in an awesome new world: JAX, Tensorflow, PyTorch, Stan, Theano
- We only need to specify our forward model and evaluate its loss
- Autodiff + optimization will then do the inference for you!

- loops? branching? recursion? closures? data structures? No problem!

- github.com/hips/autograd
  - differentiates native Python code
  - handles most of Numpy + Scipy
  - loops, branching, recursion, closures
  - arrays, tuples, lists, dicts, classes, …
  - derivatives of derivatives
  - a one-function API
  - small and easy to extend

Dougal Maclaurin
Harvard/Google Brain

- Nowadays you can do inference in a Tweet:


Ryan Adams @ryan_p_adams · 7 Nov 2015
@DavidDuvenaud

```
def elbo(p, lp, D, N):
 v=exp(p[D:])
 s=randn(N,D)*sqrt(v)+p[:D]
 return mvn.entropy(0, diag(v))+mean(lp(s))
gf = grad(elbo)
```