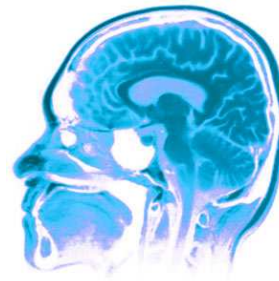




CPSC 540



Optimization and Online Learning for Logistic Regression and Neuron Models

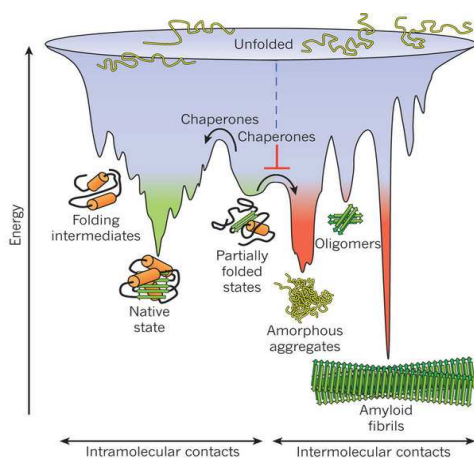


Nando de Freitas
October, 2011
University of British Columbia

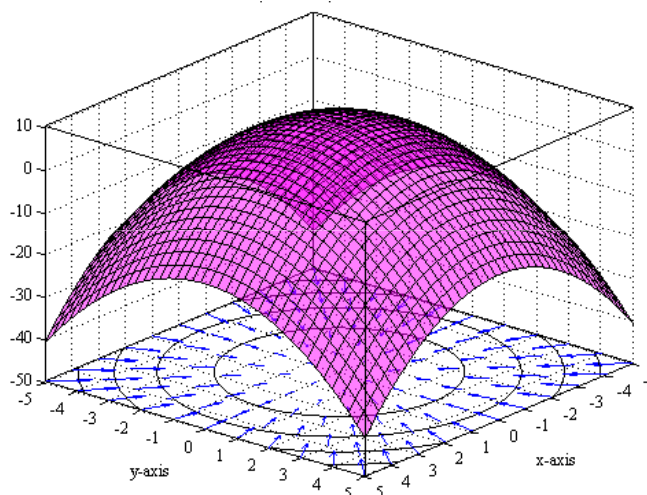
Gradient vector

- Let θ be an d -dimensional vector and $f(\theta)$ a scalar-valued function. The gradient vector of $f(\cdot)$ with respect to θ is:

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



[Nature 2011]



Hessian matrix

- The **Hessian** matrix of $f(\cdot)$ with respect to $\boldsymbol{\theta}$, written $\nabla_{\boldsymbol{\theta}}^2 f(\boldsymbol{\theta})$ or simply as \mathbf{H} , is the $d \times d$ matrix of partial derivatives,

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

Gradient descent in machine learning

- In **offline** learning, we have a **batch** of data $\mathbf{x}_{1:n} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$. We typically optimize cost functions of the form

$$f(\boldsymbol{\theta}) = f(\boldsymbol{\theta}, \mathbf{x}_{1:n}) = \frac{1}{n} \sum_{i=1}^n f(\boldsymbol{\theta}, \mathbf{x}_i)$$

- The corresponding gradient is

$$g(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}, \mathbf{x}_i)$$

- In some cases, we can solve $g(\boldsymbol{\theta}) = \mathbf{0}$ in closed form, but in general, we will have to use gradient-based optimizers.
- If we have **streaming data** or massive datasets, we opt for **online learning**. That is, we update our estimates as each new data point arrives.



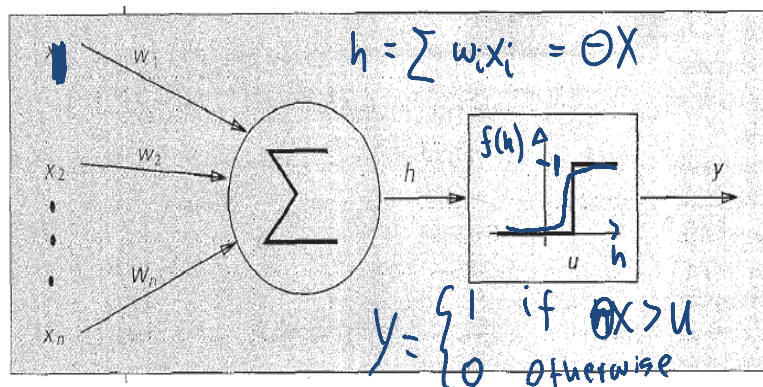
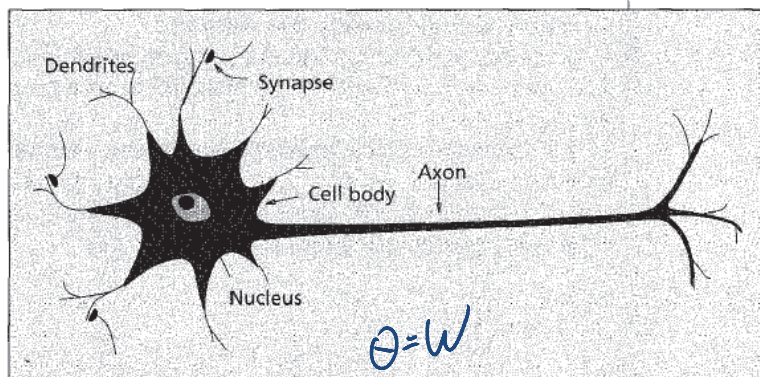
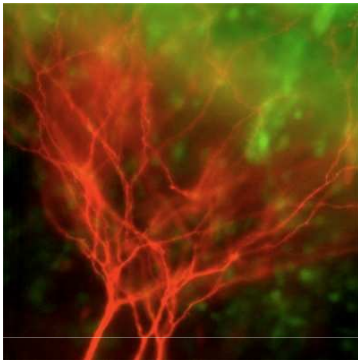
- For linear regression with training data $\{\mathbf{x}_i, y_i\}_{i=1}^n$, we have the quadratic cost

$$f(\boldsymbol{\theta}) = f(\boldsymbol{\theta}, \mathbf{X}, \mathbf{y}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) = \sum_{i=1}^n (y_i - \mathbf{x}_i \boldsymbol{\theta})^2$$

- The gradient and Hessian are:

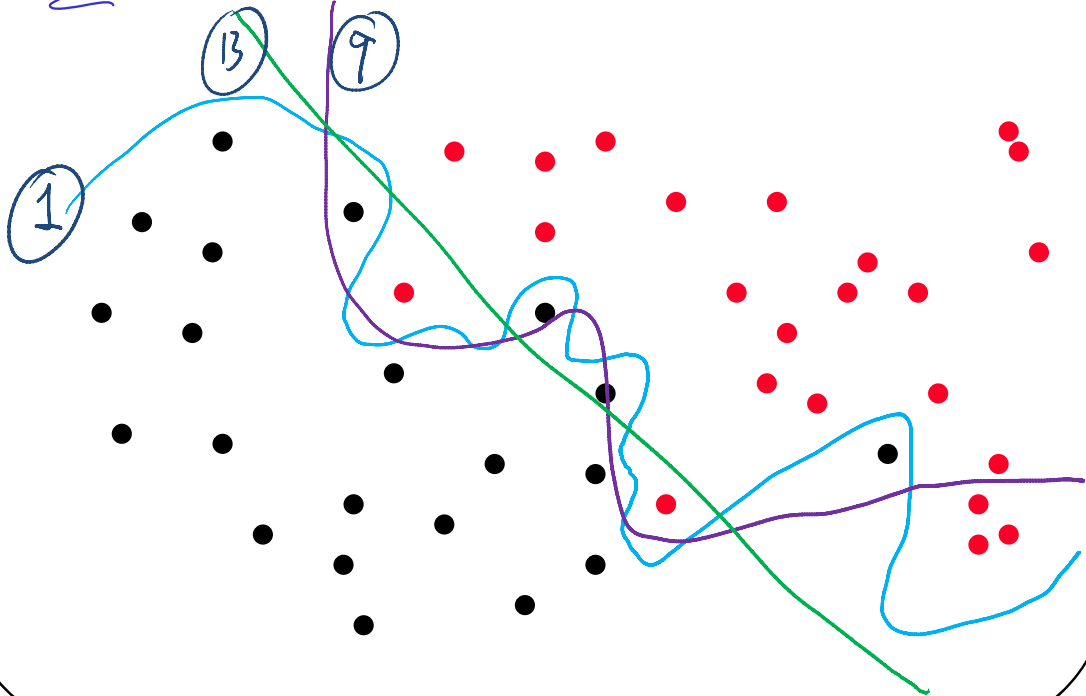
[e.]

McCulloch-Pitts model of a neuron

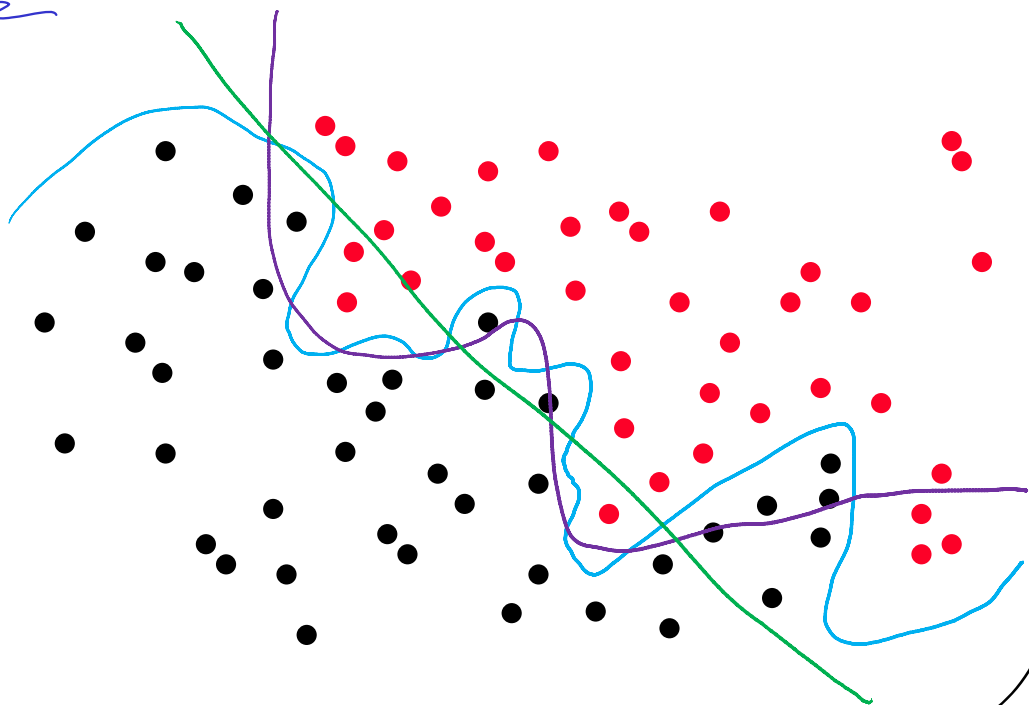




What discriminant curve best separates these two classes best?

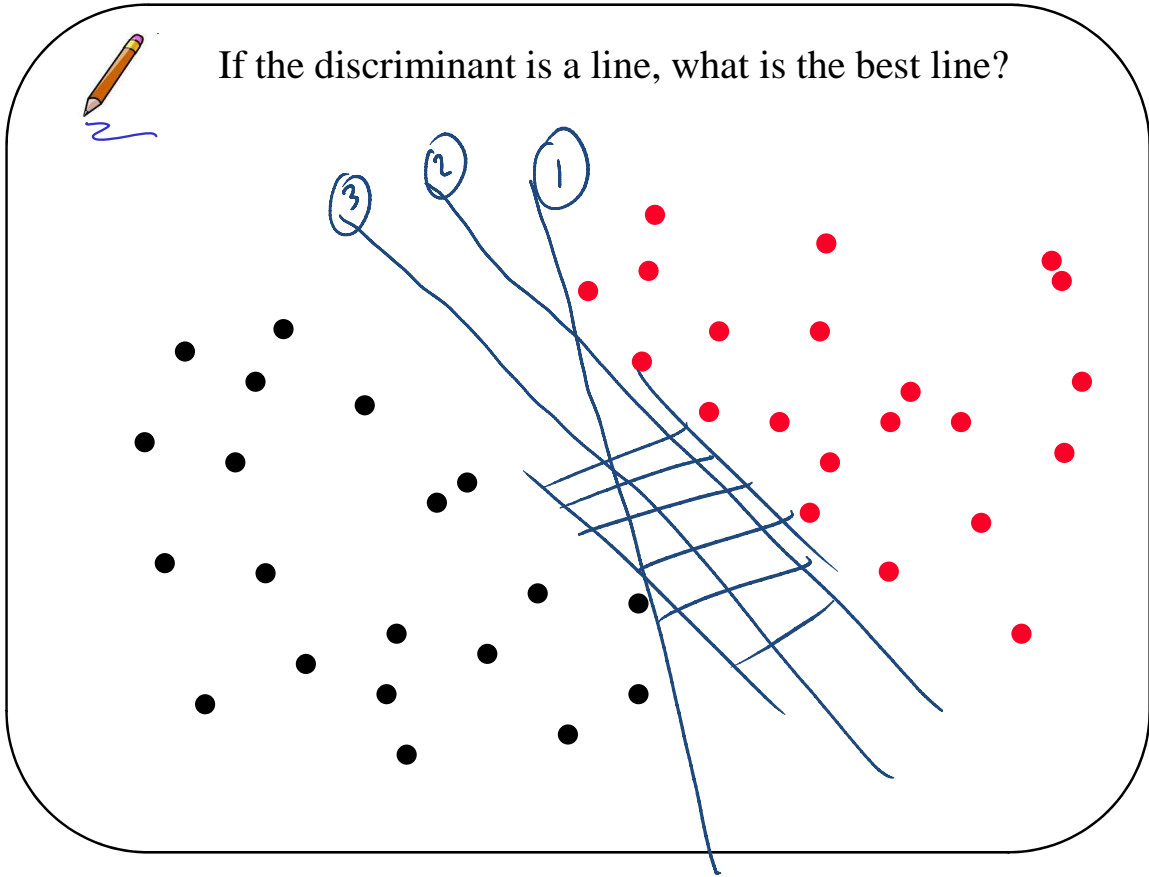


Next day, we get more data and a surprise!





If the discriminant is a line, what is the best line?



Logistic regression

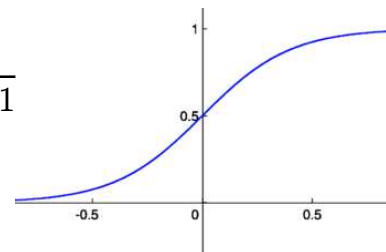
- The logistic regression model specifies the probability of a binary output $y_i \in \{0, 1\}$ given the input \mathbf{x}_i as follows:

$$\begin{aligned} p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) &= \prod_{i=1}^n \text{Ber}(y_i | \text{sigm}(\mathbf{x}_i \boldsymbol{\theta})) \\ &= \prod_{i=1}^n \left[\frac{1}{1 + e^{-\mathbf{x}_i \boldsymbol{\theta}}} \right]^{y_i} \left[1 - \frac{1}{1 + e^{-\mathbf{x}_i \boldsymbol{\theta}}} \right]^{1-y_i} \end{aligned}$$

where $\mathbf{x}_i \boldsymbol{\theta} = \theta_0 + \sum_{j=1}^d \theta_j x_{ij}$

- $\text{sigm}(\eta)$ refers to the **sigmoid** function, also known as the **logistic** or **logit** function:

$$\text{sigm}(\eta) = \frac{1}{1 + e^{-\eta}} = \frac{e^{\eta}}{e^{\eta} + 1}$$



Gradient and Hessian of binary logistic regression

- The gradient and Hessian of the negative loglikelihood, $J(\boldsymbol{\theta}) = -\log p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$, are given by:

$$\mathbf{g}(\boldsymbol{\theta}) = \frac{d}{d\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{i=1}^n \mathbf{x}_i^T (\pi_i - y_i) = \mathbf{X}^T (\boldsymbol{\pi} - \mathbf{y})$$

$$\mathbf{H} = \frac{d}{d\boldsymbol{\theta}} \mathbf{g}(\boldsymbol{\theta})^T = \sum_i \pi_i (1 - \pi_i) \mathbf{x}_i \mathbf{x}_i^T = \mathbf{X}^T \text{diag}(\pi_i (1 - \pi_i)) \mathbf{X}$$

where $\pi_i = \text{sigm}(\mathbf{x}_i \boldsymbol{\theta})$

- One can show that \mathbf{H} is positive definite; hence the NLL is **convex** and has a unique global minimum.
- To find this minimum, we turn to batch optimization.

Steepest gradient descent

- One of the simplest optimization algorithms is called **gradient descent** or **steepest descent**. This can be written as follows:

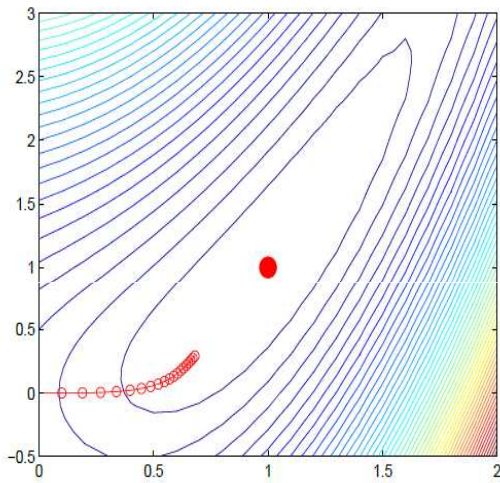
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \mathbf{g}_k = \boldsymbol{\theta}_k - \eta_k \nabla f(\boldsymbol{\theta}_k)$$

where k indexes steps of the algorithm, $\mathbf{g}_k = \mathbf{g}(\boldsymbol{\theta}_k)$ is the gradient at step k , and $\eta_k > 0$ is called the **learning rate** or **step size**.

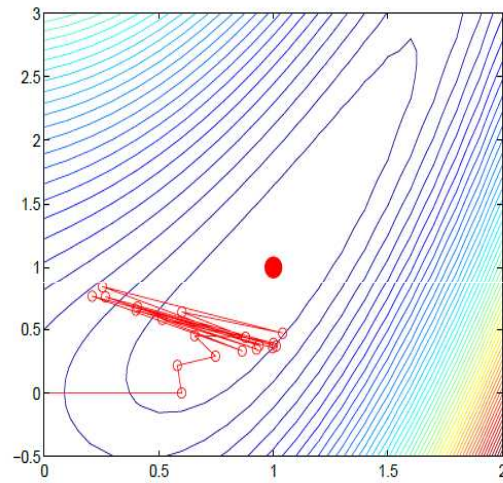
- **Taylor's theorem** illustrates why this is a sensible thing to do:

$$f(\boldsymbol{\theta}_{k+1}) \approx f(\boldsymbol{\theta}_k) + \nabla f(\boldsymbol{\theta}_k) (\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k) = f(\boldsymbol{\theta}_k) - \|\nabla f(\boldsymbol{\theta}_k)\|^2 \eta_k$$

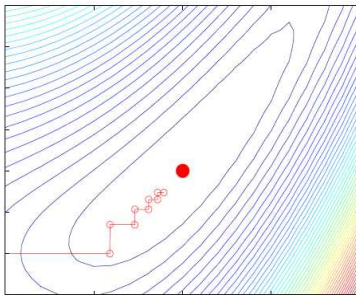
Step size choice



$\eta = 0.1.$



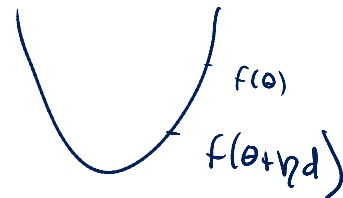
$\eta = 0.6.$



Line search

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k \mathbf{d}_k$$

$$f(\boldsymbol{\theta} + \eta \mathbf{d}) \approx f(\boldsymbol{\theta}) + \eta \mathbf{g}^T \mathbf{d}$$



- Pick η to minimize

$$\phi(\eta) = f(\boldsymbol{\theta}_k + \eta \mathbf{d}_k)$$

subject to the constraint that the resulting direction is a descent direction (the so-called **Wolfe conditions**).

- This optimization of η can be costly. Alternatively, choose an initial step size η . If the step size doesn't decrease in the objective function, then reduce the step size and repeat. This is the intuition behind the **Armijo rule**.

Newton's algorithm

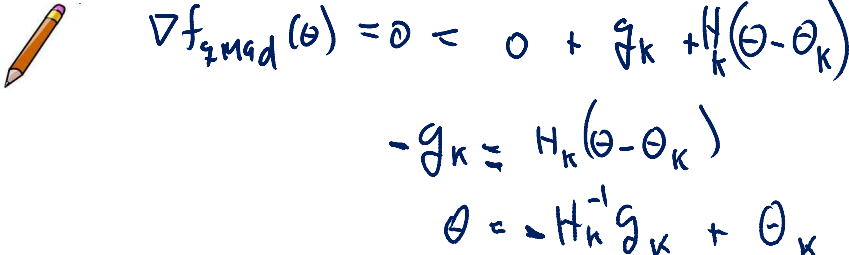
- The most basic second-order optimization algorithm is **Newton's algorithm**, which consists of updates of the form

$$\theta_{k+1} = \theta_k - \mathbf{H}_k^{-1} \mathbf{g}_k$$

- This algorithm is derived by making a second-order Taylor series approximation of $f(\theta)$ around θ_k :

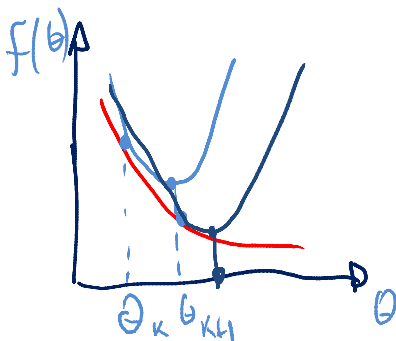
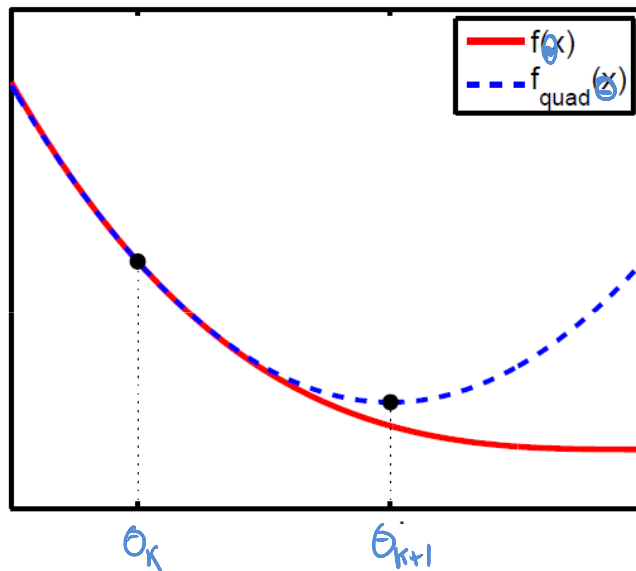
$$f_{quad}(\theta) = f(\theta_k) + \mathbf{g}_k^T (\theta - \theta_k) + \frac{1}{2} (\theta - \theta_k)^T \mathbf{H}_k (\theta - \theta_k) \leftarrow$$

differentiating and equating to zero to solve for θ_{k+1} .



$\nabla f_{quad}(\theta) = 0 = 0 + \mathbf{g}_k + \mathbf{H}_k(\theta - \theta_k)$
 $-\mathbf{g}_k = \mathbf{H}_k(\theta - \theta_k)$
 $\theta = -\mathbf{H}_k^{-1} \mathbf{g}_k + \theta_k$

Newton's as bound optimization



Newton CG algorithm

- Rather than computing $\mathbf{d}_k = -\mathbf{H}_k^{-1}\mathbf{g}_k$ directly, we can solve the linear system of equations $\mathbf{H}_k\mathbf{d}_k = -\mathbf{g}_k$ for \mathbf{d}_k .
- One efficient and popular way to do this, especially if \mathbf{H} is sparse, is to use a conjugate gradient method to solve the linear system.

```
1 Initialize  $\boldsymbol{\theta}_0$ 
2 for  $k = 1, 2, \dots$  until convergence do
3   Evaluate  $\mathbf{g}_k = \nabla f(\boldsymbol{\theta}_k)$ 
4   Evaluate  $\mathbf{H}_k = \nabla^2 f(\boldsymbol{\theta}_k)$ 
5   Solve  $\mathbf{H}_k\mathbf{d}_k = -\mathbf{g}_k$  for  $\mathbf{d}_k$ 
6   Use line search to find stepsize  $\eta_k$  along  $\mathbf{d}_k$ 
7    $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \eta_k\mathbf{d}_k$ 
```

Iteratively reweighted least squares (IRLS)

- For binary logistic regression, recall that the gradient and Hessian of the negative log-likelihood are given by

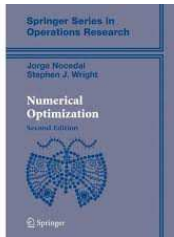
$$\begin{aligned}\mathbf{g}_k &= \mathbf{X}^T(\boldsymbol{\pi}_k - \mathbf{y}) \\ \mathbf{H}_k &= \mathbf{X}^T \mathbf{S}_k \mathbf{X} \\ \mathbf{S}_k &:= \text{diag}(\pi_{1k}(1 - \pi_{1k}), \dots, \pi_{nk}(1 - \pi_{nk})) \\ \pi_{ik} &= \text{sigm}(\mathbf{x}_i \boldsymbol{\theta}_k)\end{aligned}$$

- The Newton update at iteration $k + 1$ for this model is as follows (using $\eta_k = 1$, since the Hessian is exact):

$$\begin{aligned}\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \mathbf{H}^{-1}\mathbf{g}_k \\ &= \boldsymbol{\theta}_k + (\mathbf{X}^T \mathbf{S}_k \mathbf{X})^{-1} \mathbf{X}^T (\mathbf{y} - \boldsymbol{\pi}_k) \\ &= (\mathbf{X}^T \mathbf{S}_k \mathbf{X})^{-1} [(\mathbf{X}^T \mathbf{S}_k \mathbf{X})\boldsymbol{\theta}_k + \mathbf{X}^T (\mathbf{y} - \boldsymbol{\pi}_k)] \\ &= (\mathbf{X}^T \mathbf{S}_k \mathbf{X})^{-1} \mathbf{X}^T [\mathbf{S}_k \mathbf{X} \boldsymbol{\theta}_k + \mathbf{y} - \boldsymbol{\pi}_k]\end{aligned}$$

Quasi-Newton methods

- The Newton direction $\mathbf{d}_k = -\mathbf{H}_k^{-1} \mathbf{g}_k$ has two drawbacks. The first is that it is not necessarily a descent direction unless \mathbf{H}_k is positive definite.
- The second is that it may be too expensive to compute \mathbf{H} explicitly.
- **Quasi-Newton** methods iteratively build up an approximation to the Hessian using information gleaned from the gradient vector at each step.
- **BFGS** (named after its inventors, Broyden, Fletcher, Goldfarb and Shanno), updates the approximation to the Hessian $\mathbf{B}_k \approx \mathbf{H}_k$ as follows:



$$\begin{aligned}\mathbf{B}_{k+1} &= \mathbf{B}_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{\mathbf{B}_k^T \mathbf{s}_k \mathbf{s}_k^T \mathbf{B}_k}{\mathbf{s}_k^T \mathbf{B}_k \mathbf{s}_k} \\ \mathbf{s}_k &= \boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1} \\ \mathbf{y}_k &= \mathbf{g}_k - \mathbf{g}_{k-1}\end{aligned}$$

This is a rank-two update to the matrix, and ensures that the matrix remains positive definite.

L-BFGS

- Since storing the Hessian takes $O(d^2)$ space, for very large problems, one can use **limited memory BFGS**, or **L-BFGS**, where a low rank approximation to \mathbf{H}_k is stored implicitly. In particular, the product $\mathbf{H}_k^{-1} \mathbf{g}_k$ can be obtained by performing a sequence of inner products with \mathbf{s}_k and \mathbf{y}_k , using only the m most recent $(\mathbf{s}_k, \mathbf{y}_k)$ pairs, and ignoring older information. Typically $m \sim 20$ suffices for good performance.
- L-BFGS and Newton CG are often the methods of choice for most unconstrained optimization problems that arise in machine learning, e.g., fitting logistic regression, conditional random fields (CRFs), neural nets, etc.



Online learning

- The online gradient at iteration k is given by

$$\mathbf{g}_k := \mathbf{g}(\boldsymbol{\theta}_k) \approx (\mathbf{x}_i \boldsymbol{\theta} - y_i) \mathbf{x}_i$$

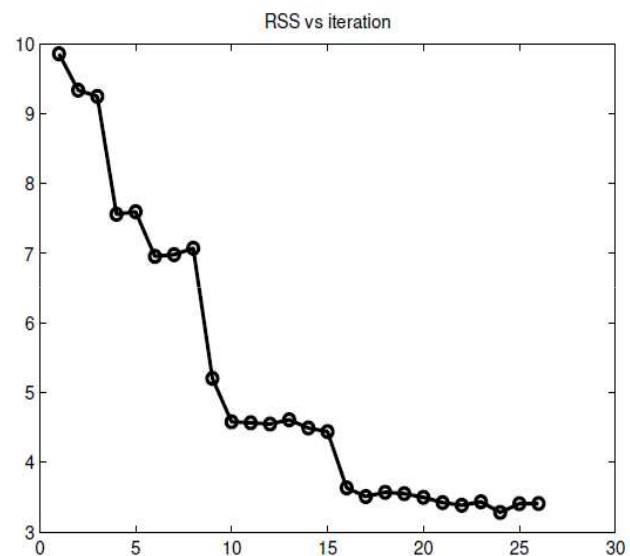
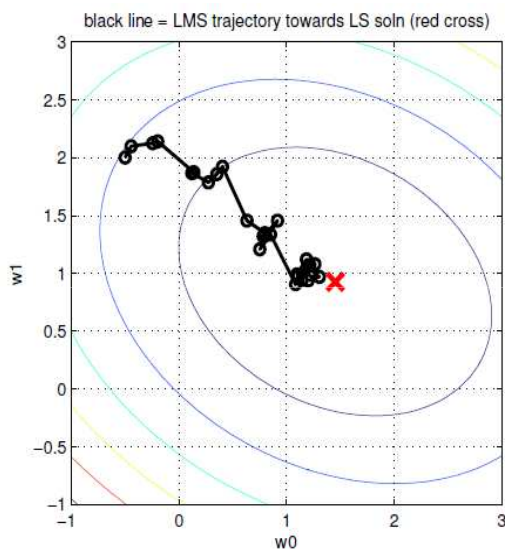
where $i = k \bmod n$ is the training example to use at iteration k .

- The feature vector \mathbf{x}_i is weighted by the difference between what we predicted, $\hat{y}_i = \pi_i = \mathbf{x}_i \boldsymbol{\theta}_k$, and the true response, y_i .
- After computing the gradient, we take a step along it as follows:

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta_k \mathbf{g}_k = \boldsymbol{\theta}_{k-1} - \eta_k (\pi_i - y_i) \mathbf{x}_i$$

This algorithm is called the **least mean squares (LMS)** and is also known as the **delta rule**, or the **Widrow-Hoff rule**.

The LMS algorithm

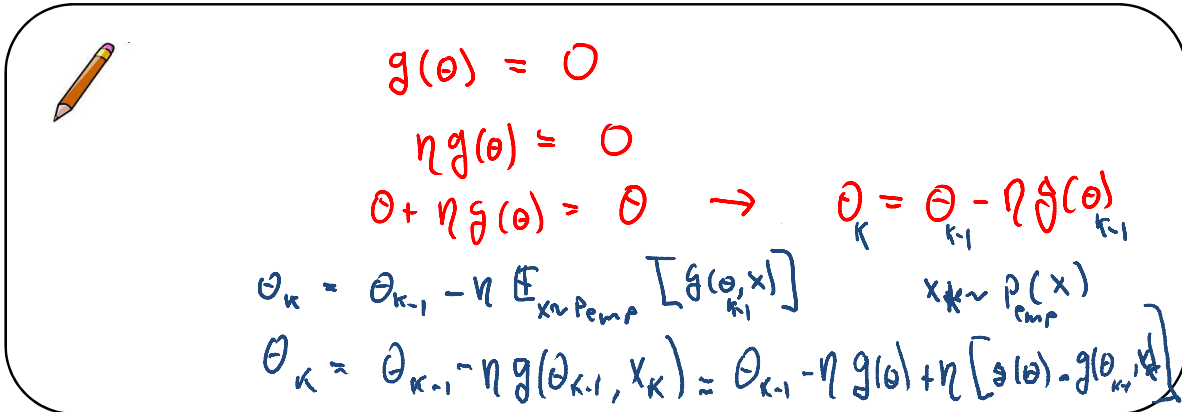


- In **stochastic gradient descent (SGD)**, we write the objective and its gradient as an expectation with respect to the empirical distribution of the data $p_{\text{emp}}(\mathbf{x})$:

$$f(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x} \sim p_{\text{emp}}} [f(\boldsymbol{\theta}, \mathbf{x})] = \int f(\boldsymbol{\theta}, \mathbf{x}) p_{\text{emp}}(\mathbf{x}) d\mathbf{x}, \quad g(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x} \sim p_{\text{emp}}} [\nabla f(\boldsymbol{\theta}, \mathbf{x})]$$

We approximate the gradient with a single sample, \mathbf{x}_k , corresponding to the most recent observation:

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta_k g(\boldsymbol{\theta}_{k-1}, \mathbf{x}_k)$$



Handwritten notes illustrating the SGD update rule:

$$g(\boldsymbol{\theta}) = 0$$

$$\eta g(\boldsymbol{\theta}) = 0$$

$$\boldsymbol{\theta} + \eta g(\boldsymbol{\theta}) = \boldsymbol{\theta} \rightarrow \boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta g(\boldsymbol{\theta}_{k-1}, \mathbf{x}_k)$$

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta \mathbb{E}_{\mathbf{x} \sim p_{\text{emp}}} [g(\boldsymbol{\theta}_{k-1}, \mathbf{x})]$$

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta g(\boldsymbol{\theta}_{k-1}, \mathbf{x}_k) = \boldsymbol{\theta}_{k-1} - \eta g(\boldsymbol{\theta}) + \eta [g(\boldsymbol{\theta}) - g(\boldsymbol{\theta}_{k-1}, \mathbf{x}_k)]$$

Stochastic gradient descent

- SGD can also be used for offline learning, by repeatedly cycling through the data; each such pass over the whole dataset is called an **epoch**. This is useful if we have **massive datasets** that will not fit in main memory. In this offline case, it is often better to compute the gradient of a **mini-batch** of B data cases. If $B = 1$, this is standard SGD, and if $B = N$, this is standard steepest descent. Typically $B \sim 100$ is used.
- Intuitively, one can get a fairly good estimate of the gradient by looking at just a few examples. Carefully evaluating precise gradients using large datasets is often a waste of time, since the algorithm will have to recompute the gradient again anyway at the next step. It is often a better use of computer time to have a noisy estimate and to move rapidly through parameter space.
- SGD is often less prone to getting stuck in shallow local minima, because it adds a certain amount of “noise”. Consequently it is quite popular in the machine learning community for fitting models such as neural networks and deep belief networks with non-convex objectives.

Momentum

- One simple heuristic to reduce the effect of zig-zagging is to add a **momentum** term, as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + (1 - \mu_k)\eta_k \mathbf{g}_k + \mu_k(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$$

where $0 \leq \mu_k \leq 1$ is the amount of momentum.

- This technique is widely used to train neural networks and other nonlinear models, such as deep belief nets.
- Hinton recommends starting with $\mu_k = 0.5$ and then slowly increasing this to $\mu_k = 0.9$. See also results of Kevin Swersky.

