Optimization Methods for Machine Learning

Sathiya Keerthi

Microsoft

Talks given at UC Santa Cruz
February 21-23, 2017

The slides for the talks will be made available at:
`http://www.keerthis.com/`

Introduction

Aim

To introduce optimization problems that arise in the solution of ML problems, briefly review relevant optimization algorithms, and point out which optimization algorithms are suited for these problems.

Range of ML problems

Classification (binary, multi-class), regression, ordinal regression, ranking, taxonomy learning, semi-supervised learning, unsupervised learning, structured outputs (e.g. sequence tagging)

Introduction

Aim

To introduce optimization problems that arise in the solution of ML problems, briefly review relevant optimization algorithms, and point out which optimization algorithms are suited for these problems.

Range of ML problems

Classification (binary, multi-class), regression, ordinal regression, ranking, taxonomy learning, semi-supervised learning, unsupervised learning, structured outputs (e.g. sequence tagging)

Classification of Optimization Algorithms

$$\min_{w \in C} E(w)$$

Nonlinear

- Unconstrained vs Constrained (Simple bounds, Linear constraints, General constraints)
- Differentiable vs Non-differentiable
- Convex vs Non-convex

Others

- Quadratic programming (E: convex quadratic function, C: linear constraints)
- Linear programming (E: linear function, C: linear constraints)
- Discrete optimization (w: discrete variables)

Classification of Optimization Algorithms

$$\min_{w \in C} E(w)$$

Nonlinear

- Unconstrained vs Constrained (Simple bounds, Linear constraints, General constraints)
- Differentiable vs Non-differentiable
- Convex vs Non-convex

Others

- Quadratic programming (E: convex quadratic function, C: linear constraints)
- Linear programming (E: linear function, C: linear constraints)
- Discrete optimization (w: discrete variables)

Classification of Optimization Algorithms

$$\min_{w \in C} E(w)$$

Nonlinear

- Unconstrained vs Constrained (Simple bounds, Linear constraints, General constraints)
- Differentiable vs Non-differentiable
- Convex vs Non-convex

Others

- Quadratic programming (E: convex quadratic function, C: linear constraints)
- Linear programming (E: linear function, C: linear constraints)
- Discrete optimization (w: discrete variables)

Unconstrained Nonlinear Optimization

$$\min_{w \in R^m} E(w)$$

Gradient

$$g(w) = \nabla E(w) = [\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_m}]^T \quad T = \text{transpose}$$

Hessian

$$H(w) = m \times m \quad \text{matrix} \quad \textit{with} \quad \frac{\partial^2 E}{\partial w_i \partial w_j} \quad \text{as} \quad \text{elements}$$

Before we go into algorithms let us look at an ML model where unconstrained nonlinear optimization problems arise.

Unconstrained Nonlinear Optimization

$$\min_{w \in R^m} E(w)$$

Gradient

$$g(w) = \nabla E(w) = [\frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_m}]^T \quad T = \text{transpose}$$

Hessian

$$H(w) = m \times m \quad \text{matrix} \quad \textit{with} \quad \frac{\partial^2 E}{\partial w_i \partial w_j} \quad \text{as} \quad \text{elements}$$

Before we go into algorithms let us look at an ML model where unconstrained nonlinear optimization problems arise.

Unconstrained Nonlinear Optimization

$$\min_{w \in R^m} E(w)$$

Gradient

$$g(w) = \nabla E(w) = [\frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_m}]^T \quad T = \text{transpose}$$

Hessian

$$H(w) = m \times m \quad \text{matrix} \quad \textit{with} \quad \frac{\partial^2 E}{\partial w_i \partial w_j} \quad \text{as} \quad \text{elements}$$

Before we go into algorithms let us look at an ML model where unconstrained nonlinear optimization problems arise.

Unconstrained Nonlinear Optimization

$$\min_{w \in R^m} E(w)$$

Gradient

$$g(w) = \nabla E(w) = [\frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_m}]^T \quad T = \text{transpose}$$

Hessian

$$H(w) = m \times m \quad \text{matrix} \quad \textit{with} \quad \frac{\partial^2 E}{\partial w_i \partial w_j} \quad \text{as} \quad \text{elements}$$

Before we go into algorithms let us look at an ML model where unconstrained nonlinear optimization problems arise.

Regularized ML Models

Training data: $\{(\mathbf{x}_i, t_i)\}_{i=1}^{nex}$

$x_i \in R^m$ is the i-th input vector

t_i is the target for x_i

e.g. binary classification: $t_i = 1 \Rightarrow$ Class 1 and $-1 \Rightarrow$ Class 2

The aim is to form a decision function $y(x, w)$

e.g. Linear classifier: $y(x, w) = \sum_i w_i x_i = w^T x$.

Loss function

$L(y(x_i, w), t_i)$ expresses the loss due to y not yielding the desired t_i

The form of L depends on the problem and model used.

Empirical error

$$\mathcal{L} = \sum_i L(y(x_i, w), t_i)$$

Regularized ML Models

Training data: $\{(\mathbf{x}_i, t_i)\}_{i=1}^{nex}$

$x_i \in R^m$ is the i-th input vector

t_i is the target for x_i

e.g. binary classification: $t_i = 1 \Rightarrow$ Class 1 and $-1 \Rightarrow$ Class 2

The aim is to form a decision function $y(x, w)$

e.g. Linear classifier: $y(x, w) = \sum_i w_i x_i = w^T x$.

Loss function

$L(y(x_i, w), t_i)$ expresses the loss due to y not yielding the desired t_i

The form of L depends on the problem and model used.

Empirical error

$$\mathcal{L} = \sum_i L(y(x_i, w), t_i)$$

Regularized ML Models

Training data: $\{(\mathbf{x}_i, t_i)\}_{i=1}^{nex}$

$x_i \in R^m$ is the i-th input vector

t_i is the target for x_i

e.g. binary classification: $t_i = 1 \Rightarrow$ Class 1 and $-1 \Rightarrow$ Class 2

The aim is to form a decision function $y(x, w)$

e.g. Linear classifier: $y(x, w) = \sum_i w_i x_i = w^T x$.

Loss function

$L(y(x_i, w), t_i)$ expresses the loss due to y not yielding the desired t_i

The form of L depends on the problem and model used.

Empirical error

$$\mathcal{L} = \sum_i L(y(x_i, w), t_i)$$

The Optimization Problem

Regularizer

Minimizing only $\mathcal{L}$ can lead to overfitting on the training data. The regularizer function $\mathcal{R}$ prefers simpler models and helps prevent overfitting. E.g. $\mathcal{R} = \|w\|^2$.

Training problem

w, the parameter vector which defines the model is obtained by solving the following optimization problem: $\min_w E = \mathcal{R} + C\mathcal{L}$

Regularization parameter

The parameter C helps to establish a trade-off between $\mathcal{R}$ and $\mathcal{L}$. C is a *hyperparameter*. All hyperparameters need to be tuned at a higher level than the training stage, e.g. by doing cross-validation.

The Optimization Problem

Regularizer

Minimizing only $\mathcal{L}$ can lead to overfitting on the training data. The regularizer function $\mathcal{R}$ prefers simpler models and helps prevent overfitting. E.g. $\mathcal{R} = \|w\|^2$.

Training problem

w, the parameter vector which defines the model is obtained by solving the following optimization problem: $\min_w E = \mathcal{R} + C\mathcal{L}$

Regularization parameter

The parameter C helps to establish a trade-off between $\mathcal{R}$ and $\mathcal{L}$. C is a *hyperparameter*. All hyperparameters need to be tuned at a higher level than the training stage, e.g. by doing cross-validation.

The Optimization Problem

Regularizer

Minimizing only $\mathcal{L}$ can lead to overfitting on the training data. The regularizer function $\mathcal{R}$ prefers simpler models and helps prevent overfitting. E.g. $\mathcal{R} = \|w\|^2$.

Training problem

w, the parameter vector which defines the model is obtained by solving the following optimization problem: $\min_w E = \mathcal{R} + C\mathcal{L}$

Regularization parameter

The parameter C helps to establish a trade-off between $\mathcal{R}$ and $\mathcal{L}$. C is a *hyperparameter*. All hyperparameters need to be tuned at a higher level than the training stage, e.g. by doing cross-validation.

Binary Classification: loss functions

Decision: $y(x, w) > 0 \Rightarrow$ Class 1, else Class 2.

Logistic Regression

Logistic loss: $L(y, t) = \log(1 + \exp(-ty))$
It is the negative-log-likelihood of the probability of t:
$1/(1 + \exp(-ty))$.

Support Vector Machines (SVMs)

Hinge loss: $l(y, t) = 1 - ty$ if $ty < 1$; 0 otherwise.
Squared Hinge loss: $l(y, t) = (1 - ty)^2/2$ if $ty < 1$; 0 otherwise.
Modified Huber loss: $l(y, t)$ is: 0 if $\xi \geq 0$; $\xi^2/2$ if $0 < \xi < 2$; and $2(\xi - 1)$ if $\xi \geq 2$, where $\xi = 1 - ty$.

Binary Classification: loss functions

Decision: $y(x, w) > 0 \Rightarrow$ Class 1, else Class 2.

Logistic Regression

Logistic loss: $L(y, t) = \log(1 + \exp(-ty))$
It is the negative-log-likelihood of the probability of t:
$1/(1 + \exp(-ty))$.

Support Vector Machines (SVMs)

Hinge loss: $l(y, t) = 1 - ty$ if $ty < 1$; 0 otherwise.
Squared Hinge loss: $l(y, t) = (1 - ty)^2/2$ if $ty < 1$; 0 otherwise.
Modified Huber loss: $l(y, t)$ is: 0 if $\xi \geq 0$; $\xi^2/2$ if $0 < \xi < 2$; and $2(\xi - 1)$ if $\xi \geq 2$, where $\xi = 1 - ty$.

Binary Classification: loss functions

Decision: $y(x, w) > 0 \Rightarrow$ Class 1, else Class 2.

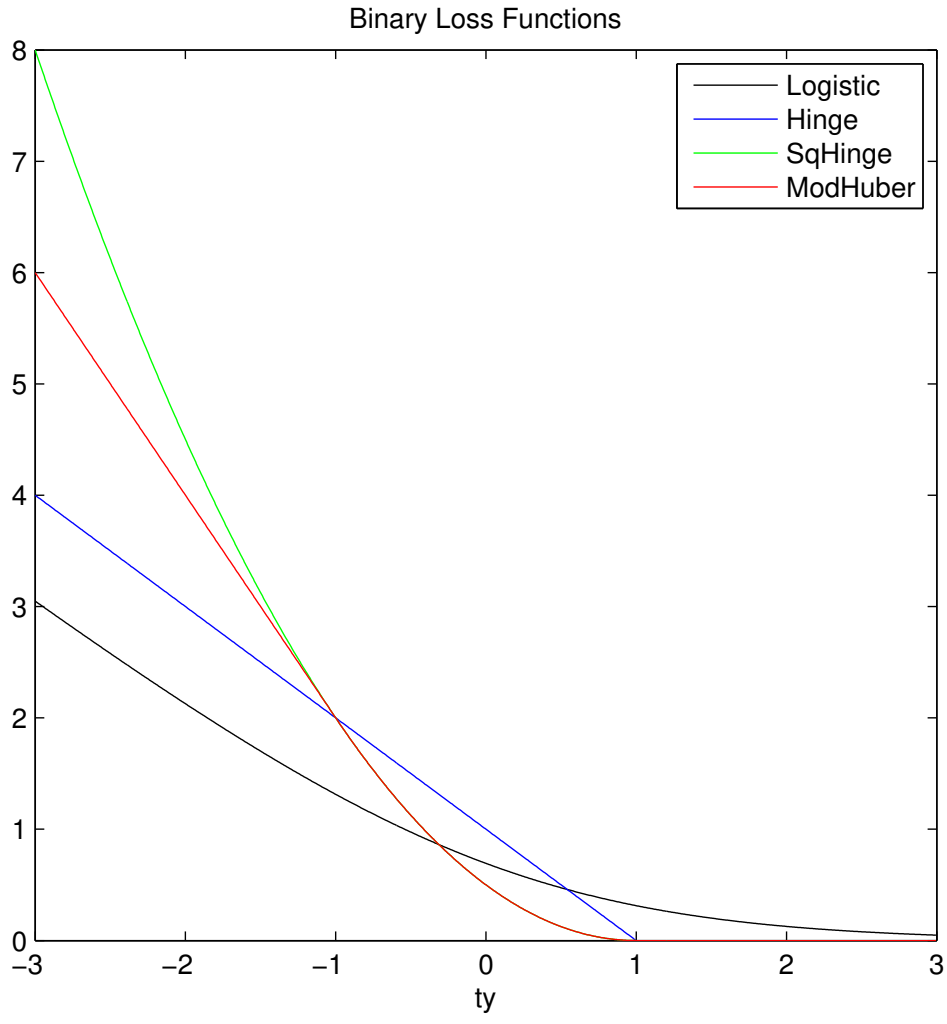
Logistic Regression

Logistic loss: $L(y, t) = \log(1 + \exp(-ty))$
It is the negative-log-likelihood of the probability of t:
$1/(1 + \exp(-ty))$.

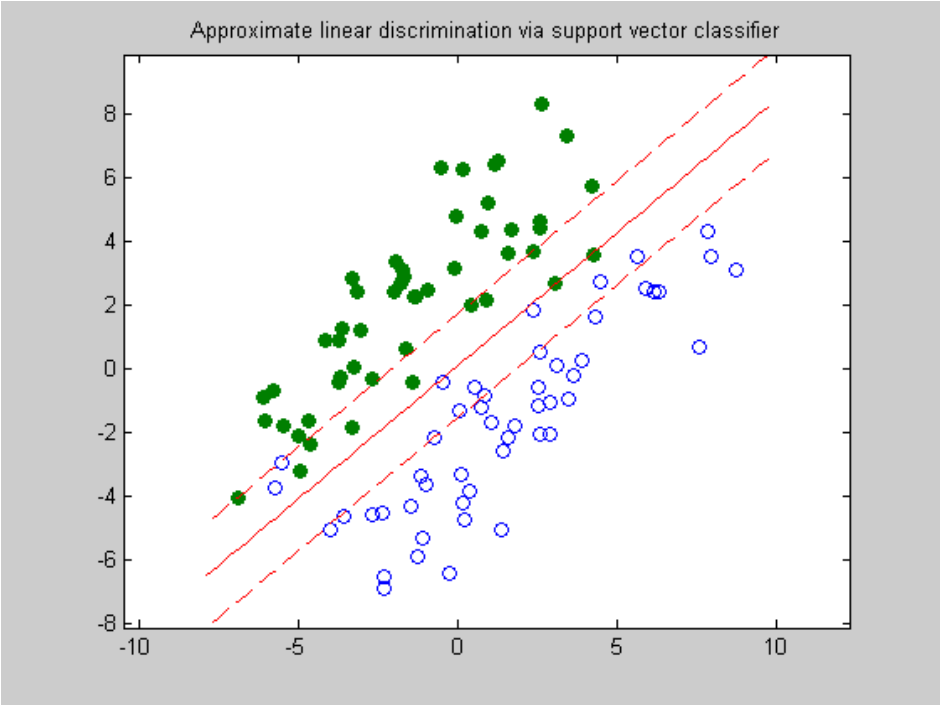
Support Vector Machines (SVMs)

Hinge loss: $l(y, t) = 1 - ty$ if $ty < 1$; 0 otherwise.
Squared Hinge loss: $l(y, t) = (1 - ty)^2/2$ if $ty < 1$; 0 otherwise.
Modified Huber loss: $l(y, t)$ is: 0 if $\xi \geq 0$; $\xi^2/2$ if $0 < \xi < 2$; and $2(\xi - 1)$ if $\xi \geq 2$, where $\xi = 1 - ty$.

Binary Loss functions



SVMs and Margin Maximization



The margin between the planes defined by $y = \pm 1$ is $2/\|w\|$.
Making margin big is equivalent to making $\mathcal{R} = \|w\|^2$ small.

Unconstrained optimization: Optimality conditions

At a minimum we have stationarity: $\nabla E = 0$
Non-negative curvature: H is positive semi-definite

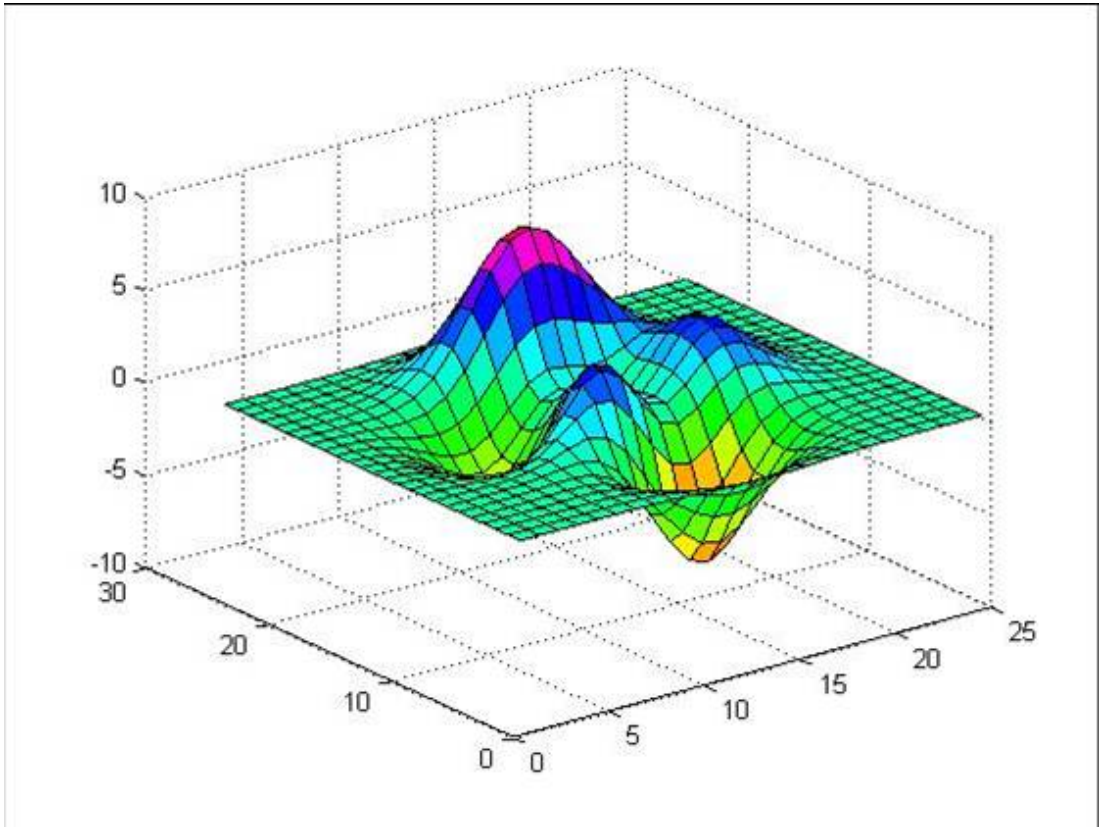
E convex $\Rightarrow$ local minimum is a global minimum.

Unconstrained optimization: Optimality conditions

At a minimum we have stationarity: $\nabla E = 0$
Non-negative curvature: H is positive semi-definite

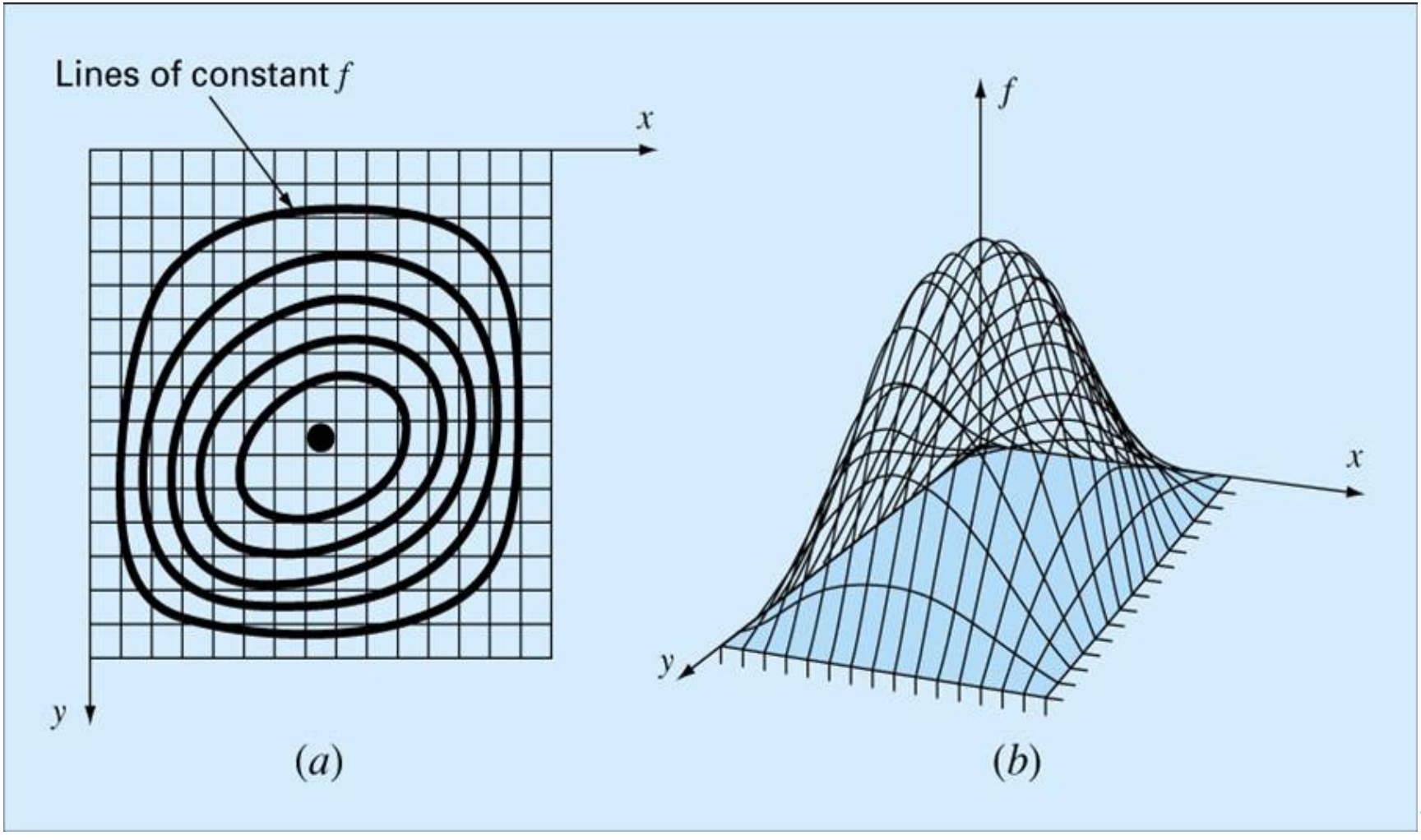
E convex $\Rightarrow$ local minimum is a global minimum.

Non-convex functions have local minima



Representation of functions by contours

$$w = (x, y) \qquad E = f$$



Geometry of descent

$$\nabla E(\theta^{\text{now}})^T d < 0; \qquad \text{Here : } \theta \text{ is } w$$

Tangent plane: E =constant is approximately
$E(\theta^{\text{now}}) + \nabla E(\theta^{\text{now}})^T(\theta - \theta^{\text{now}})$ =constant $\Leftrightarrow$
$\nabla E(\theta^{\text{now}})^T(\theta - \theta^{\text{now}}) = 0$

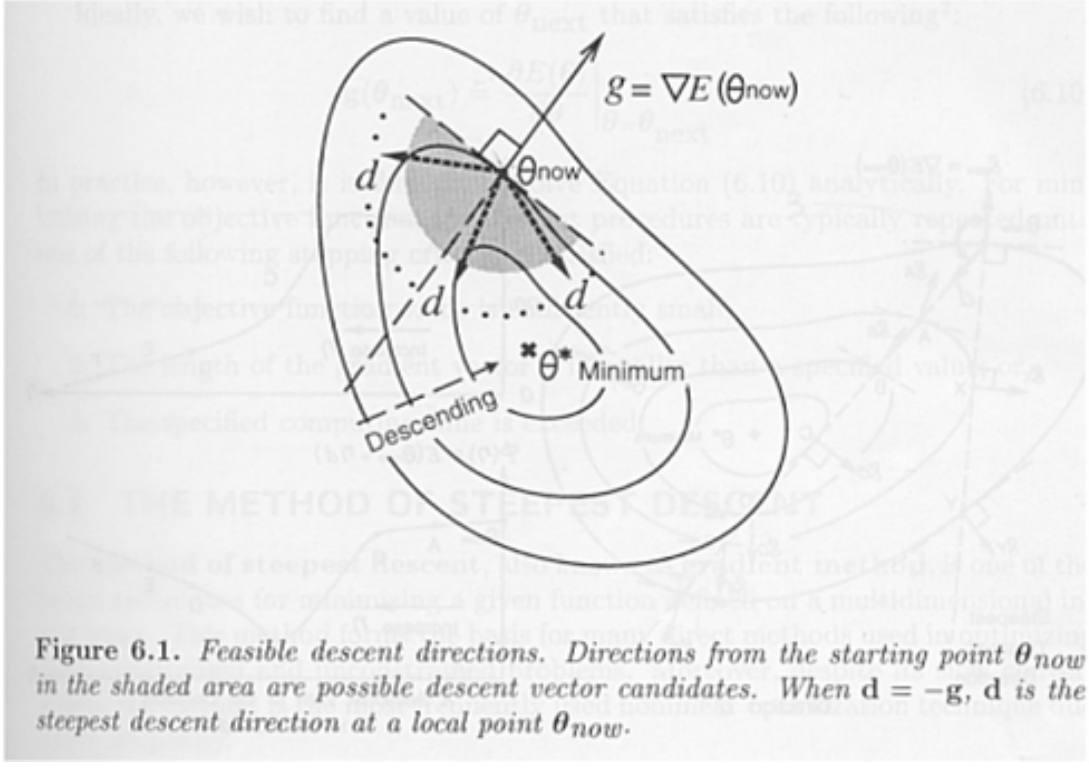


Figure 6.1. *Feasible descent directions. Directions from the starting point $\boldsymbol{\theta}_{now}$ in the shaded area are possible descent vector candidates. When $\mathbf{d} = -\mathbf{g}$, $\mathbf{d}$ is the steepest descent direction at a local point $\boldsymbol{\theta}_{now}$.*

A sketch of a descent algorithm

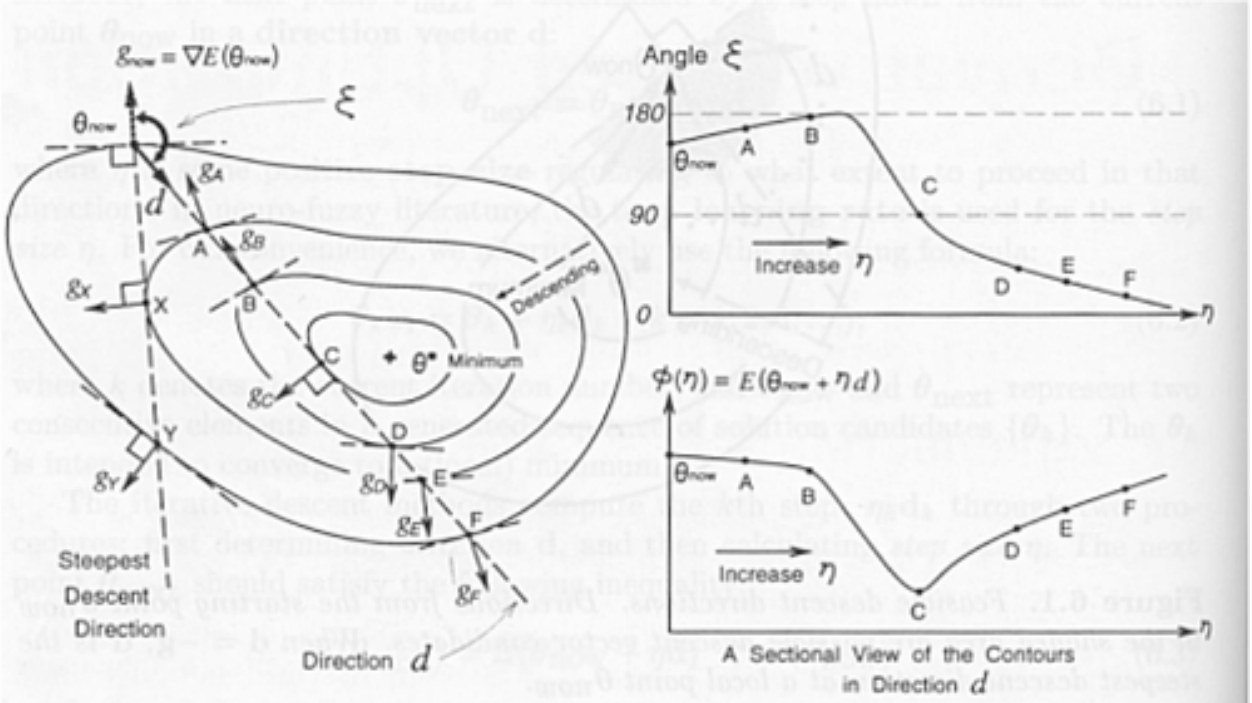


Figure 6.2. *Angle ξ between gradient directions* $\mathbf{g}$ *and a descent direction* $\mathbf{d}$*, which is determined by a certain algorithm at the current point* $\boldsymbol{\theta}_{now}$*. Let* N *be the set of all possible next points;* $N \supset \{A, B, C, D, E, F, X, Y\}$*. In the one-way downhill direction* $\mathbf{d}$*, the next point* $\boldsymbol{\theta}_{next}$ *may be one of six points—*A*,* B*,* C*,* D*,* E*, or* F*—or be in the vicinity of them, depending on step sizes. By comparison, in the steepest descent direction,* $\boldsymbol{\theta}_{next}$*, may be either* X *or* Y*, or close to them.*

Steps of a Descent Algorithm

1. Input w_0.
2. For $k \geq 0$, choose a descent direction d_k at w_k:

$$\nabla E(w_k)^T d_k < 0$$

3. Compute a step size η by line search on $E(w_k + \eta d_k)$.
4. Set $w_{k+1} = w_k + \eta d_k$.
5. Continue with next k until some termination criterion (e.g. $\|\nabla E\| \leq \epsilon$) is satisfied.

Most optimization methods/codes will ask for the functions, $E(w)$ and $\nabla E(w)$ to be made available. (Some also need H^{-1} or H times a vector d operation to be available.)

Gradient/Hessian of $E = \mathcal{R} + C\mathcal{L}$

Classifier outputs

$y_i = w^T x_i = x_i^T w$, written combined for all i as: $y = Xw$
X is $nex \times m$ matrix with x_i^T as the i-th row.

Gradient structure

$$\nabla E = 2w + C \sum_i a(y_i, t) x_i = 2w + CX^T a$$

where a is a nex dimensional vector containing the $a(y_i, t)$ values.

Hessian structure

$$H = 2I + CX^T DX, \quad D \text{ is diagonal}$$

In large scale problems (e.g text classification) X turns out to be sparse and $Hd = 2d + CX^T(D(Xd))$ calculation for any given vector d is cheap to compute.

Gradient/Hessian of $E = \mathcal{R} + C\mathcal{L}$

Classifier outputs

$y_i = w^T x_i = x_i^T w$, written combined for all i as: $y = Xw$
X is $nex \times m$ matrix with x_i^T as the i-th row.

Gradient structure

$$\nabla E = 2w + C \sum_i a(y_i, t) x_i = 2w + CX^T a$$

where a is a nex dimensional vector containing the $a(y_i, t)$ values.

Hessian structure

$$H = 2I + CX^T DX, \quad D \text{ is diagonal}$$

In large scale problems (e.g text classification) X turns out to be sparse and $Hd = 2d + CX^T(D(Xd))$ calculation for any given vector d is cheap to compute.

Gradient/Hessian of $E = \mathcal{R} + C\mathcal{L}$

Classifier outputs

$y_i = w^T x_i = x_i^T w$, written combined for all i as: $y = Xw$
X is $nex \times m$ matrix with x_i^T as the i-th row.

Gradient structure

$$\nabla E = 2w + C \sum_i a(y_i, t) x_i = 2w + CX^T a$$

where a is a nex dimensional vector containing the $a(y_i, t)$ values.

Hessian structure

$$H = 2I + CX^T DX, \quad D \text{ is diagonal}$$

In large scale problems (e.g text classification) X turns out to be sparse and $Hd = 2d + CX^T(D(Xd))$ calculation for any given vector d is cheap to compute.

Exact line search along a direction d

$$\eta^\star = \min_\eta \phi(\eta) = E(w + \eta d)$$

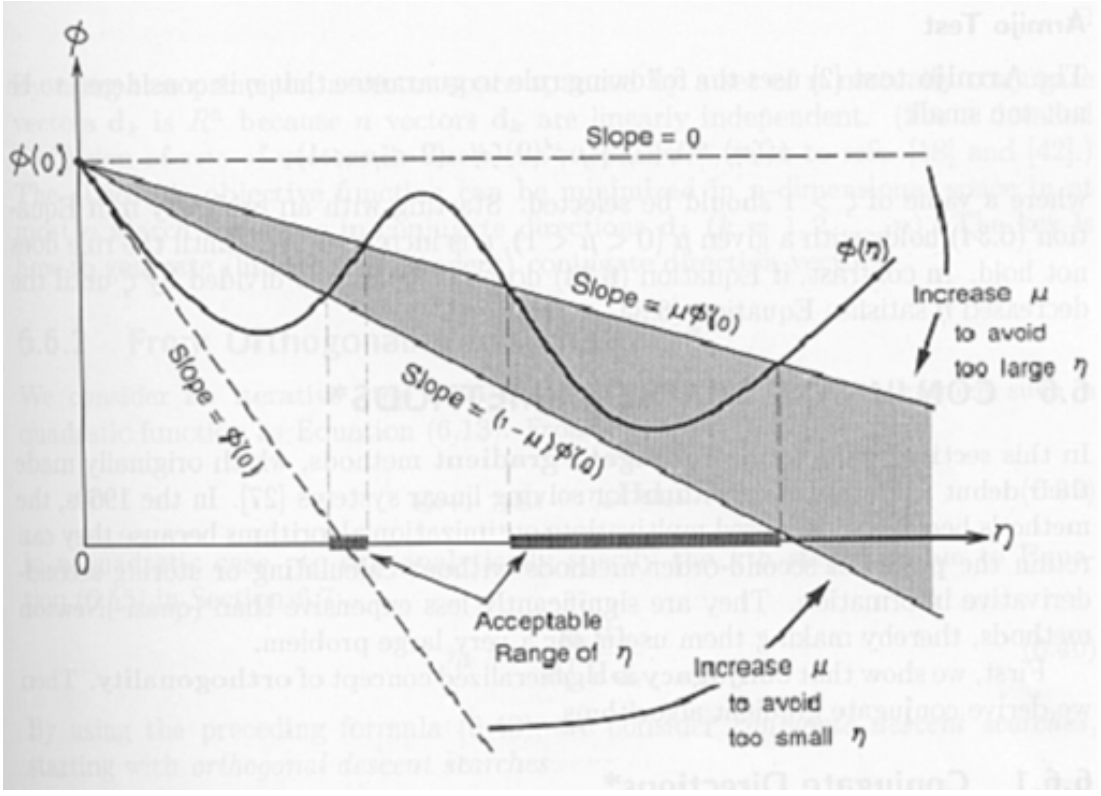
Hard to do unless E has simple form such as a quadratic form.

Exact line search along a direction d

$$\eta^\star = \min_\eta \phi(\eta) = E(w + \eta d)$$

Hard to do unless E has simple form such as a quadratic form.

Inexact line search: Armijo condition



Global convergence theorem

E is Lipschitz continuous

Sufficient angle of descent condition: For some fixed $\delta > 0$,

$$-\nabla E(w_k)^T d_k \geq \delta \|\nabla E(w_k)\| \|d_k\|$$

Armijo line search condition: For some fixed $\mu_1 \geq \mu_2 > 0$

$$-\mu_1 \eta \nabla E(w_k)^T d_k \geq E(w_k) - E(w_k + \eta d_k) \geq -\mu_2 \eta \nabla E(w_k)^T d_k$$

Then, either $E \to -\infty$ or w_k converges to a stationary point $w^\star$: $\nabla E(w^\star) = 0$.

Global convergence theorem

E is Lipschitz continuous

Sufficient angle of descent condition: For some fixed $\delta > 0$,

$$-\nabla E(w_k)^T d_k \geq \delta \|\nabla E(w_k)\| \|d_k\|$$

Armijo line search condition: For some fixed $\mu_1 \geq \mu_2 > 0$

$$-\mu_1 \eta \nabla E(w_k)^T d_k \geq E(w_k) - E(w_k + \eta d_k) \geq -\mu_2 \eta \nabla E(w_k)^T d_k$$

Then, either $E \to -\infty$ or w_k converges to a stationary point $w^\star$: $\nabla E(w^\star) = 0$.

Global convergence theorem

E is Lipschitz continuous

Sufficient angle of descent condition: For some fixed $\delta > 0$,

$$-\nabla E(w_k)^T d_k \geq \delta \|\nabla E(w_k)\| \|d_k\|$$

Armijo line search condition: For some fixed $\mu_1 \geq \mu_2 > 0$

$$-\mu_1 \eta \nabla E(w_k)^T d_k \geq E(w_k) - E(w_k + \eta d_k) \geq -\mu_2 \eta \nabla E(w_k)^T d_k$$

Then, either $E \to -\infty$ or w_k converges to a stationary point $w^\star$: $\nabla E(w^\star) = 0$.

Global convergence theorem

E is Lipschitz continuous

Sufficient angle of descent condition: For some fixed $\delta > 0$,

$$-\nabla E(w_k)^T d_k \geq \delta \|\nabla E(w_k)\| \|d_k\|$$

Armijo line search condition: For some fixed $\mu_1 \geq \mu_2 > 0$

$$-\mu_1 \eta \nabla E(w_k)^T d_k \geq E(w_k) - E(w_k + \eta d_k) \geq -\mu_2 \eta \nabla E(w_k)^T d_k$$

Then, either $E \to -\infty$ or w_k converges to a stationary point $w^\star$: $\nabla E(w^\star) = 0$.

Rate of convergence

$$\epsilon^k = E(w_k) - E(w_{k+1})$$

$|\epsilon_{k+1}| = \rho|\epsilon_k|^r$ in limit as $k \to \infty$

$r =$ rate of convergence,
a key factor for speed of convergence of optimization algorithms

Linear convergence $(r = 1)$ is quite a bit slower than
quadratic convergence $(r = 2)$.

Many optimization algorithms have
superlinear convergence $(1 < r < 2)$ which is pretty good.

Rate of convergence

$$\epsilon^k = E(w_k) - E(w_{k+1})$$

$|\epsilon_{k+1}| = \rho|\epsilon_k|^r$ in limit as $k \to \infty$

$r =$ rate of convergence,
a key factor for speed of convergence of optimization algorithms

Linear convergence $(r = 1)$ is quite a bit slower than
quadratic convergence $(r = 2)$.

Many optimization algorithms have
superlinear convergence $(1 < r < 2)$ which is pretty good.

Rate of convergence

$\epsilon^k = E(w_k) - E(w_{k+1})$

$|\epsilon_{k+1}| = \rho|\epsilon_k|^r$ in limit as $k \to \infty$

$r =$ rate of convergence,
a key factor for speed of convergence of optimization algorithms

Linear convergence $(r = 1)$ is quite a bit slower than
quadratic convergence $(r = 2)$.

Many optimization algorithms have
superlinear convergence $(1 < r < 2)$ which is pretty good.

Rate of convergence

$\epsilon^k = E(w_k) - E(w_{k+1})$

$|\epsilon_{k+1}| = \rho|\epsilon_k|^r$ in limit as $k \to \infty$

$r =$ rate of convergence,
a key factor for speed of convergence of optimization algorithms

Linear convergence $(r = 1)$ is quite a bit slower than
quadratic convergence $(r = 2)$.

Many optimization algorithms have
superlinear convergence $(1 < r < 2)$ which is pretty good.

Rate of convergence

$\epsilon^k = E(w_k) - E(w_{k+1})$

$|\epsilon_{k+1}| = \rho|\epsilon_k|^r$ in limit as $k \to \infty$

$r =$ rate of convergence,
a key factor for speed of convergence of optimization algorithms

Linear convergence $(r = 1)$ is quite a bit slower than
quadratic convergence $(r = 2)$.

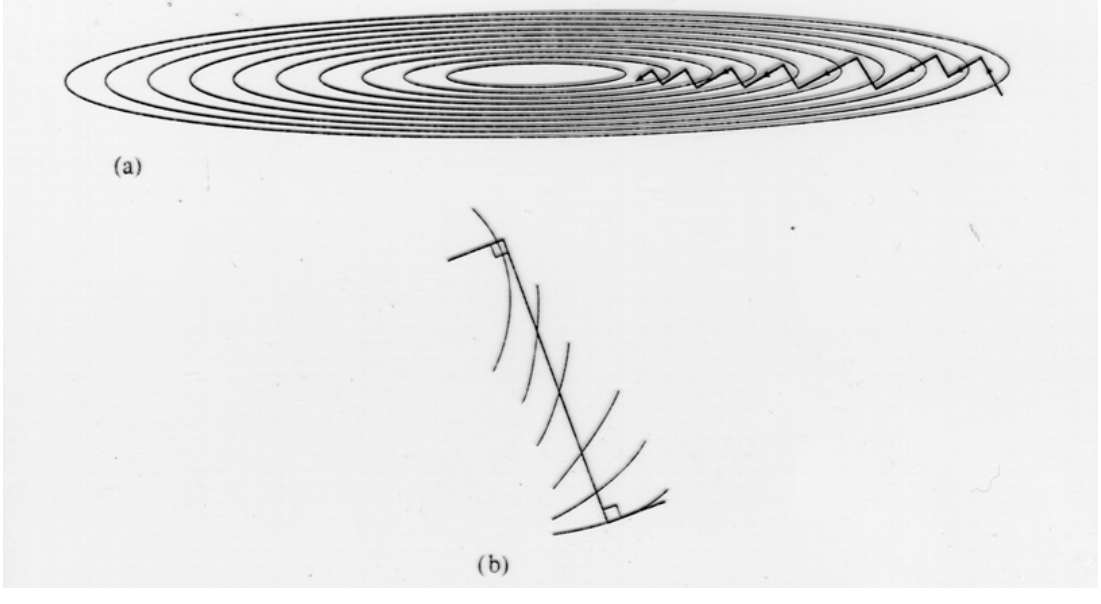
Many optimization algorithms have
superlinear convergence $(1 < r < 2)$ which is pretty good.

(Batch) Gradient descent method

$$d = -\nabla E$$

Linear convergence

Very simple; locally good; but often very slow; rarely used in practice.
`http://en.wikipedia.org/wiki/Steepest_descent`



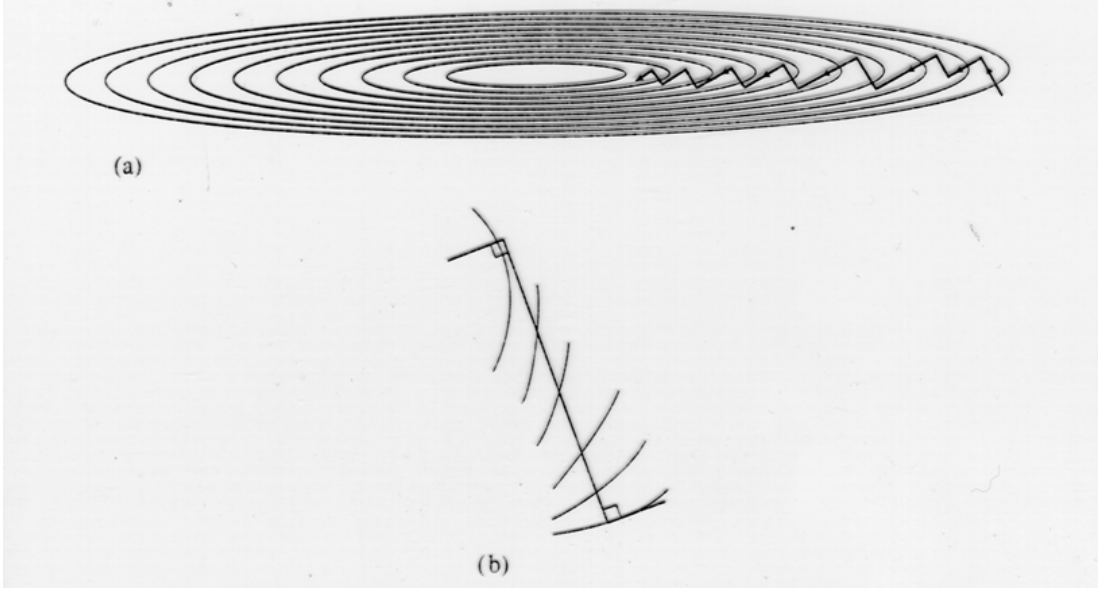
(Batch) Gradient descent method

$$d = -\nabla E$$

Linear convergence

Very simple; locally good; but often very slow; rarely used in practice.

`http://en.wikipedia.org/wiki/Steepest_descent`

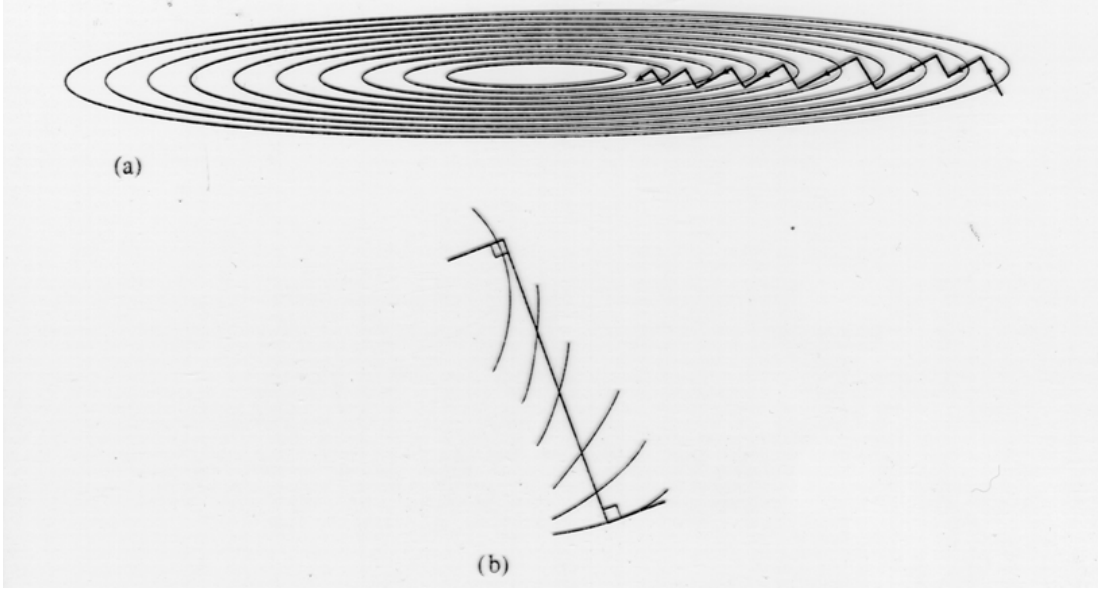


(Batch) Gradient descent method

$d = -\nabla E$

Linear convergence

Very simple; locally good; but often very slow; rarely used in practice.
`http://en.wikipedia.org/wiki/Steepest_descent`



Conjugate Gradient (CG) Method

Motivation

Accelerate slow convergence of steepest descent, but keep its simplicity: use only ∇E and avoid operations involving Hessian.

Conjugate gradient methods can be regarded as somewhat in between steepest descent and Newton's method (discussed below), having the positive features of both of them.

Conjugate gradient methods originally invented and solved for the quadratic problem:
$min\ E = w^T Q w - b^T w \quad \Leftrightarrow \quad \text{solving} \ \ 2Qw = b$
Solution of $2Qw = b$ this way is referred as: *Linear Conjugate Gradient*

http://en.wikipedia.org/wiki/Conjugate_gradient_method

Conjugate Gradient (CG) Method

Motivation

Accelerate slow convergence of steepest descent, but keep its simplicity: use only ∇E and avoid operations involving Hessian.

Conjugate gradient methods can be regarded as somewhat in between steepest descent and Newton's method (discussed below), having the positive features of both of them.

Conjugate gradient methods originally invented and solved for the quadratic problem:
$min\ E = w^T Qw - b^T w \quad \Leftrightarrow \quad \text{solving}\ \ 2Qw = b$
Solution of $2Qw = b$ this way is referred as: *Linear Conjugate Gradient*

http://en.wikipedia.org/wiki/Conjugate_gradient_method

Conjugate Gradient (CG) Method

Motivation

Accelerate slow convergence of steepest descent, but keep its simplicity: use only ∇E and avoid operations involving Hessian.

Conjugate gradient methods can be regarded as somewhat in between steepest descent and Newton's method (discussed below), having the positive features of both of them.

Conjugate gradient methods originally invented and solved for the quadratic problem:
$min\ E = w^T Q w - b^T w \quad \Leftrightarrow \quad \text{solving}\ \ 2Qw = b$
Solution of $2Qw = b$ this way is referred as: *Linear Conjugate Gradient*

`http://en.wikipedia.org/wiki/Conjugate_gradient_method`

Basic Principle

Given a symmetric pd matrix Q, two vectors d_1 and d_2 are said to be Q *conjugate* if $d_1^T Q d_2 = 0$.

Given a full set of independent Q conjugate vectors $\{d_i\}$, the minimizer of the quadratic E can be written as

$$w^\star = \eta_1 d_1 + \ldots + \eta_m d_m \tag{1}$$

Using $2Qw^\star = b$, pre-multiplying (1) by $2Q$ and by taking the scalar product with d_i we can easily solve for η_i:

$$d_i^T b = d_i^T 2Q w^\star = 0 + \cdots + \eta_i d_i^T 2Q d_i + \cdots + 0$$

Key computation: Q times d operations.

Basic Principle

Given a symmetric pd matrix Q, two vectors d_1 and d_2 are said to be Q *conjugate* if $d_1^T Q d_2 = 0$.

Given a full set of independent Q conjugate vectors $\{d_i\}$, the minimizer of the quadratic E can be written as

$$w^\star = \eta_1 d_1 + ... + \eta_m d_m \tag{1}$$

Using $2Qw^* = b$, pre-multiplying (1) by $2Q$ and by taking the scalar product with d_i we can easily solve for η_i:

$$d_i^T b = d_i^T 2Q w^\star = 0 + \cdots + \eta_i d_i^T 2Q d_i + \cdots + 0$$

Key computation: Q times d operations.

Basic Principle

Given a symmetric pd matrix Q, two vectors d_1 and d_2 are said to be Q *conjugate* if $d_1^T Q d_2 = 0$.

Given a full set of independent Q conjugate vectors $\{d_i\}$, the minimizer of the quadratic E can be written as

$$w^\star = \eta_1 d_1 + ... + \eta_m d_m \tag{1}$$

Using $2Qw^\star = b$, pre-multiplying (1) by $2Q$ and by taking the scalar product with d_i we can easily solve for η_i:

$$d_i^T b = d_i^T 2Q w^\star = 0 + \cdots + \eta_i d_i^T 2Q d_i + \cdots + 0$$

Key computation: Q times d operations.

Conjugate Gradient Method

The conjugate gradient method starts with gradient descent direction as the first direction and selects the successive conjugate directions on the fly.

- Start with $d_0 = -g(w_0)$, where $g = \nabla E$.
- Simple formula to determine the new Q-conjugate direction:

$$d_{k+1} = -g(w_{k+1}) + \beta_k d_k$$

Only slightly more complicated than steepest descent.

Fletcher-Reeves formula: $\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$

Polak-Ribierre formula: $\beta_k = \frac{g_{k+1}^T (g_{k+1} - g_k)}{g_k^T g_k}$

Conjugate Gradient Method

The conjugate gradient method starts with gradient descent direction as the first direction and selects the successive conjugate directions on the fly.

- Start with $d_0 = -g(w_0)$, where $g = \nabla E$.
- Simple formula to determine the new Q-conjugate direction:

$$d_{k+1} = -g(w_{k+1}) + \beta_k d_k$$

Only slightly more complicated than steepest descent.

Fletcher-Reeves formula: $\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$

Polak-Ribierre formula: $\beta_k = \frac{g_{k+1}^T (g_{k+1} - g_k)}{g_k^T g_k}$

Conjugate Gradient Method

The conjugate gradient method starts with gradient descent direction as the first direction and selects the successive conjugate directions on the fly.

- Start with $d_0 = -g(w_0)$, where $g = \nabla E$.
- Simple formula to determine the new Q-conjugate direction:

$$d_{k+1} = -g(w_{k+1}) + \beta_k d_k$$

Only slightly more complicated than steepest descent.

Fletcher-Reeves formula: $\beta_k = \frac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$

Polak-Ribierre formula: $\beta_k = \frac{g_{k+1}^T (g_{k+1} - g_k)}{g_k^T g_k}$

Extending CG to Nonlinear Minimization

There is no proper theory since there is no specific Q matrix.

Still, simply extend CG by:

- using FR or PR formulas for choosing the directions
- obtaining step sizes η_i by line search

The resulting method has good convergence when implemented with good line search.

`http://en.wikipedia.org/wiki/Nonlinear_conjugate_gradient`

Extending CG to Nonlinear Minimization

There is no proper theory since there is no specific Q matrix.

Still, simply extend CG by:

- using FR or PR formulas for choosing the directions
- obtaining step sizes η_i by line search

The resulting method has good convergence when implemented with good line search.

http://en.wikipedia.org/wiki/Nonlinear_conjugate_gradient

Extending CG to Nonlinear Minimization

There is no proper theory since there is no specific Q matrix.

Still, simply extend CG by:

- using FR or PR formulas for choosing the directions
- obtaining step sizes η_i by line search

The resulting method has good convergence when implemented with good line search.

`http://en.wikipedia.org/wiki/Nonlinear_conjugate_gradient`

Newton method

$$d = -H^{-1}\nabla E, \qquad \eta = 1$$

$w + d$ minimizes second order approximation

$$\hat{E}(w + d) = E(w) + \nabla E(w)^T d + \frac{1}{2} d^T H(w) d$$

$w + d$ solves linearized optimality condition

$$\nabla E(w + d) \approx \nabla \hat{E}(w + d) = \nabla E(w) + H(w)d = 0$$

Quadratic rate of convergence

http://en.wikipedia.org/wiki/Newton's_method_in_optimization

Newton method

$$d = -H^{-1}\nabla E, \qquad \eta = 1$$

$w + d$ minimizes second order approximation

$$\hat{E}(w + d) = E(w) + \nabla E(w)^T d + \frac{1}{2} d^T H(w) d$$

$w + d$ solves linearized optimality condition

$$\nabla E(w + d) \approx \nabla \hat{E}(w + d) = \nabla E(w) + H(w)d = 0$$

Quadratic rate of convergence

http://en.wikipedia.org/wiki/Newton's_method_in_optimization

Newton method

$$d = -H^{-1}\nabla E, \qquad \eta = 1$$

$w + d$ **minimizes second order approximation**

$$\hat{E}(w + d) = E(w) + \nabla E(w)^T d + \frac{1}{2} d^T H(w) d$$

$w + d$ **solves linearized optimality condition**

$$\nabla E(w + d) \approx \nabla \hat{E}(w + d) = \nabla E(w) + H(w)d = 0$$

Quadratic rate of convergence

http://en.wikipedia.org/wiki/Newton's_method_in_optimization

Newton method

$$d = -H^{-1}\nabla E, \qquad \eta = 1$$

$w + d$ minimizes second order approximation

$$\hat{E}(w + d) = E(w) + \nabla E(w)^T d + \frac{1}{2} d^T H(w) d$$

$w + d$ solves linearized optimality condition

$$\nabla E(w + d) \approx \nabla \hat{E}(w + d) = \nabla E(w) + H(w)d = 0$$

Quadratic rate of convergence

http://en.wikipedia.org/wiki/Newton's_method_in_optimization

Newton method: Comments

Compute $H(w)$, $g = \nabla E(w)$ solve $Hd = -g$ and set $w := w + d$. When number of variables is large do the linear system solution approximately by iterative methods, e.g. linear CG discussed earlier. http://en.wikipedia.org/wiki/Conjugate_gradient_method

Newton method may not converge (or worse, If H is not positive definite, Newton method may not even be properly defined when started far from a minimum $\Rightarrow$ d may not even be descent)

Convex E: H is positive definite, so d is a descent direction. Still, an added step, *line search*, is needed to ensure convergence.

If E is piecewise quadratic, differentiable and convex (e.g. SVM training with squared hinge loss) then the Newton-type method converges in a finite number of steps.

Newton method: Comments

Compute $H(w)$, $g = \nabla E(w)$ solve $Hd = -g$ and set $w := w + d$. When number of variables is large do the linear system solution approximately by iterative methods, e.g. linear CG discussed earlier. http://en.wikipedia.org/wiki/Conjugate_gradient_method

Newton method may not converge (or worse, If H is not positive definite, Newton method may not even be properly defined when started far from a minimum $\Rightarrow$ d may not even be descent)

Convex E: H is positive definite, so d is a descent direction. Still, an added step, *line search*, is needed to ensure convergence.

If E is piecewise quadratic, differentiable and convex (e.g. SVM training with squared hinge loss) then the Newton-type method converges in a finite number of steps.

Newton method: Comments

Compute $H(w)$, $g = \nabla E(w)$ solve $Hd = -g$ and set $w := w + d$. When number of variables is large do the linear system solution approximately by iterative methods, e.g. linear CG discussed earlier. http://en.wikipedia.org/wiki/Conjugate_gradient_method

Newton method may not converge (or worse, If H is not positive definite, Newton method may not even be properly defined when started far from a minimum $\Rightarrow$ d may not even be descent)

Convex E: H is positive definite, so d is a descent direction. Still, an added step, *line search*, is needed to ensure convergence.

If E is piecewise quadratic, differentiable and convex (e.g. SVM training with squared hinge loss) then the Newton-type method converges in a finite number of steps.

Newton method: Comments

Compute $H(w)$, $g = \nabla E(w)$ solve $Hd = -g$ and set $w := w + d$. When number of variables is large do the linear system solution approximately by iterative methods, e.g. linear CG discussed earlier. http://en.wikipedia.org/wiki/Conjugate_gradient_method

Newton method may not converge (or worse, If H is not positive definite, Newton method may not even be properly defined when started far from a minimum $\Rightarrow$ d may not even be descent)

Convex E: H is positive definite, so d is a descent direction. Still, an added step, *line search*, is needed to ensure convergence.

If E is piecewise quadratic, differentiable and convex (e.g. SVM training with squared hinge loss) then the Newton-type method converges in a finite number of steps.

Trust Region Newton method

Define *trust region* at the current point: $T = \{w : \|w - w_k\| \leq r\}$ a region where you think $\hat{E}$, the quadratic used in the derivation of Newton method approximates E well.

Optimize the Newton quadratic $\hat{E}$ only within T. In the case of solving large scale systems via linear CG iterations, simply terminate when the iterations hit the boundary of T.

After each iteration, observe the ratio of decrements in E and $\hat{E}$. Compare this ratio with 1 to decide whether to expand or shrink T.

In large scale problems, when far away from $w^\star$ (the minimizer of E) T is hit in just a few CG iterations. When near $w^\star$ many CG iterations are used to zoom in on $w^\star$ quickly.

http://www.machinelearning.org/proceedings/icml2007/papers/114.pdf

Trust Region Newton method

Define *trust region* at the current point: $T = \{w : \|w - w_k\| \leq r\}$ a region where you think $\hat{E}$, the quadratic used in the derivation of Newton method approximates E well.

Optimize the Newton quadratic $\hat{E}$ only within T. In the case of solving large scale systems via linear CG iterations, simply terminate when the iterations hit the boundary of T.

After each iteration, observe the ratio of decrements in E and $\hat{E}$. Compare this ratio with 1 to decide whether to expand or shrink T.

In large scale problems, when far away from $w^\star$ (the minimizer of E) T is hit in just a few CG iterations. When near $w^\star$ many CG iterations are used to zoom in on $w^\star$ quickly.

http://www.machinelearning.org/proceedings/icml2007/papers/114.pdf

Trust Region Newton method

Define *trust region* at the current point: $T = \{w : \|w - w_k\| \leq r\}$ a region where you think $\hat{E}$, the quadratic used in the derivation of Newton method approximates E well.

Optimize the Newton quadratic $\hat{E}$ only within T. In the case of solving large scale systems via linear CG iterations, simply terminate when the iterations hit the boundary of T.

After each iteration, observe the ratio of decrements in E and $\hat{E}$. Compare this ratio with 1 to decide whether to expand or shrink T.

In large scale problems, when far away from $w^\star$ (the minimizer of E) T is hit in just a few CG iterations. When near $w^\star$ many CG iterations are used to zoom in on $w^\star$ quickly.

http://www.machinelearning.org/proceedings/icml2007/papers/114.pdf

Trust Region Newton method

Define *trust region* at the current point: $T = \{w : \|w - w_k\| \leq r\}$ a region where you think $\hat{E}$, the quadratic used in the derivation of Newton method approximates E well.

Optimize the Newton quadratic $\hat{E}$ only within T. In the case of solving large scale systems via linear CG iterations, simply terminate when the iterations hit the boundary of T.

After each iteration, observe the ratio of decrements in E and $\hat{E}$. Compare this ratio with 1 to decide whether to expand or shrink T.

In large scale problems, when far away from $w^\star$ (the minimizer of E) T is hit in just a few CG iterations. When near $w^\star$ many CG iterations are used to zoom in on $w^\star$ quickly.

`http://www.machinelearning.org/proceedings/icml2007/papers/114.pdf`

Quasi-Newton Methods

Instead of the true Hessian, an initial matrix H_0 is chosen (usually $H_0 = I$) which is subsequently modified by an update formula:

$$H_{k+1} = H_k + H_k^u$$

where H_k^u is the update matrix.

This updating can also be done with the inverse of the Hessian $B = H^{-1}$ as follows:

$$B_{k+1} = B_k + B_k^u$$

This is better since Newton direction is: $-H^{-1}g = -Bg$.

Quasi-Newton Methods

Instead of the true Hessian, an initial matrix H_0 is chosen (usually $H_0 = I$) which is subsequently modified by an update formula:

$$H_{k+1} = H_k + H_k^u$$

where H_k^u is the update matrix.

This updating can also be done with the inverse of the Hessian $B = H^{-1}$ as follows:

$$B_{k+1} = B_k + B_k^u$$

This is better since Newton direction is: $-H^{-1}g = -Bg$.

Hessian Matrix Updates

Given two points w_k and w_{k+1}, define $g_k = \nabla E(w_k)$ and $g_{k+1} = \nabla E(w_{k+1})$. Further, let $p_k = w_{k+1} - w_k$, then

$$g_{k+1} - g_k \approx H(w_k)p_k$$

If the Hessian is constant, then $g_{k+1} - g_k = H_{k+1}p_k$.

Define $q_k = g_{k+1} - g_k$. Rewrite this condition as

$$H_{k+1}^{-1}q_k = p_k$$

This is called the quasi-Newton condition.

Hessian Matrix Updates

Given two points w_k and w_{k+1}, define $g_k = \nabla E(w_k)$ and $g_{k+1} = \nabla E(w_{k+1})$. Further, let $p_k = w_{k+1} - w_k$, then

$$g_{k+1} - g_k \approx H(w_k)p_k$$

If the Hessian is constant, then $g_{k+1} - g_k = H_{k+1}p_k$.

Define $q_k = g_{k+1} - g_k$. Rewrite this condition as

$$H_{k+1}^{-1}q_k = p_k$$

This is called the quasi-Newton condition.

Hessian Matrix Updates

Given two points w_k and w_{k+1}, define $g_k = \nabla E(w_k)$ and $g_{k+1} = \nabla E(w_{k+1})$. Further, let $p_k = w_{k+1} - w_k$, then

$$g_{k+1} - g_k \approx H(w_k)p_k$$

If the Hessian is constant, then $g_{k+1} - g_k = H_{k+1}p_k$.

Define $q_k = g_{k+1} - g_k$. Rewrite this condition as

$$H_{k+1}^{-1}q_k = p_k$$

This is called the quasi-Newton condition.

Rank One and Rank Two Updates

Substitute the updating formula $B_{k+1} = B_k + B_k^u$ and the condition becomes

$$B_k q_k + B_k^u q_k = p_k \tag{1}$$

(remember: $p_k = w_{k+1} - w_k$ and $q_k = g_{k+1} - g_k$)

There is no unique solution to finding the update matrix B_k^u.

A general form is

$$B_k^u = auu^T + bvv^T$$

where a and b are scalars and u and v are vectors.
auu^T and bvv^T are rank one matrices.
Quasi-Newton methods that take $b = 0$: rank one updates.
Quasi-Newton methods that take $b \neq 0$: rank two updates.

Rank One and Rank Two Updates

Substitute the updating formula $B_{k+1} = B_k + B_k^u$ and the condition becomes

$$B_k q_k + B_k^u q_k = p_k \tag{1}$$

(remember: $p_k = w_{k+1} - w_k$ and $q_k = g_{k+1} - g_k$)

There is no unique solution to finding the update matrix B_k^u.

A general form is

$$B_k^u = auu^T + bvv^T$$

where a and b are scalars and u and v are vectors.
auu^T and bvv^T are rank one matrices.
Quasi-Newton methods that take $b = 0$: rank one updates.
Quasi-Newton methods that take $b \neq 0$: rank two updates.

Rank One and Rank Two Updates

Substitute the updating formula $B_{k+1} = B_k + B_k^u$ and the condition becomes

$$B_k q_k + B_k^u q_k = p_k \tag{1}$$

(remember: $p_k = w_{k+1} - w_k$ and $q_k = g_{k+1} - g_k$)

There is no unique solution to finding the update matrix B_k^u.

A general form is

$$B_k^u = auu^T + bvv^T$$

where a and b are scalars and u and v are vectors.
auu^T and bvv^T are rank one matrices.
Quasi-Newton methods that take $b = 0$: rank one updates.
Quasi-Newton methods that take $b \neq 0$: rank two updates.

Rank one updates are simple, but have limitations. Rank two updates are the most widely used schemes. The rationale can be quite complicated.

The following two update formulas have received wide acceptance:
Davidon -Fletcher-Powell (DFP) formula
Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula.

Numerical experiments have shown that BFGS formula's performance is superior over DFP formula. Hence, BFGS is often preferred over DFP.
`http://en.wikipedia.org/wiki/BFGS_method`

Rank one updates are simple, but have limitations. Rank two updates are the most widely used schemes. The rationale can be quite complicated.

The following two update formulas have received wide acceptance:
Davidon -Fletcher-Powell (DFP) formula
Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula.

Numerical experiments have shown that BFGS formula's performance is superior over DFP formula. Hence, BFGS is often preferred over DFP.
http://en.wikipedia.org/wiki/BFGS_method

Rank one updates are simple, but have limitations. Rank two updates are the most widely used schemes. The rationale can be quite complicated.

The following two update formulas have received wide acceptance:
Davidon -Fletcher-Powell (DFP) formula
Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula.

Numerical experiments have shown that BFGS formula's performance is superior over DFP formula. Hence, BFGS is often preferred over DFP.
`http://en.wikipedia.org/wiki/BFGS_method`

Quasi-Newton Algorithm

1. Input w_0, $B_0 = I$.
2. For $k \geq 0$, set $d_k = -B_k g_k$.
3. Compute a step size η (e.g., by line search on $E(w_k + \eta d_k)$) and set $w_{k+1} = w_k + \eta d_k$.
4. Compute the update matrix B_k^u according to a given formula (say, DFP or BFGS) using the values $q_k = g_{k+1} - g_k$, $p_k = w_{k+1} - w_k$, and B_k.
5. Set $B_{k+1} = B_k + B_k^u$.
6. Continue with next k until termination criteria are satisfied.

Limited Memory Quasi Newton

When the number of variables is large, even maintaining and using B is expensive.

Limited memory quasi Newton methods which use a low rank updating of B using only the (p_k, q_k) vectors from the past L steps (L small, say 5-15) work well in such large scale settings.

L-BFGS is very popular. In particular see Nocedal's code (`http://en.wikipedia.org/wiki/L-BFGS`).

Limited Memory Quasi Newton

When the number of variables is large, even maintaining and using B is expensive.

Limited memory quasi Newton methods which use a low rank updating of B using only the (p_k, q_k) vectors from the past L steps (L small, say 5-15) work well in such large scale settings.

L-BFGS is very popular. In particular see Nocedal's code (`http://en.wikipedia.org/wiki/L-BFGS`).

Limited Memory Quasi Newton

When the number of variables is large, even maintaining and using B is expensive.

Limited memory quasi Newton methods which use a low rank updating of B using only the (p_k, q_k) vectors from the past L steps (L small, say 5-15) work well in such large scale settings.

L-BFGS is very popular. In particular see Nocedal's code (`http://en.wikipedia.org/wiki/L-BFGS`).

Overall comparison of the methods

Gradient descent method is slow and should be avoided as much as possible.

Conjugate gradient methods are simple, need little memory, and, if implemented carefully, are very much faster.

Quasi-Newton methods are robust. But, they require $O(m^2)$ memory space (m is number of variables) to store the approximate Hessian inverse, so they are not suited for large scale problems. Limited Memory Quasi-Newton methods use $O(m)$ memory (like CG and gradient descent) and they are suited for large scale problems.

In many problems Hd evaluation is fast and, for them Newton-type methods are excellent candidates, e.g. Trust region Newton.

Overall comparison of the methods

Gradient descent method is slow and should be avoided as much as possible.

Conjugate gradient methods are simple, need little memory, and, if implemented carefully, are very much faster.

Quasi-Newton methods are robust. But, they require $O(m^2)$ memory space (m is number of variables) to store the approximate Hessian inverse, so they are not suited for large scale problems. Limited Memory Quasi-Newton methods use $O(m)$ memory (like CG and gradient descent) and they are suited for large scale problems.

In many problems Hd evaluation is fast and, for them Newton-type methods are excellent candidates, e.g. Trust region Newton.

Overall comparison of the methods

Gradient descent method is slow and should be avoided as much as possible.

Conjugate gradient methods are simple, need little memory, and, if implemented carefully, are very much faster.

Quasi-Newton methods are robust. But, they require $O(m^2)$ memory space (m is number of variables) to store the approximate Hessian inverse, so they are not suited for large scale problems. Limited Memory Quasi-Newton methods use $O(m)$ memory (like CG and gradient descent) and they are suited for large scale problems.

In many problems Hd evaluation is fast and, for them Newton-type methods are excellent candidates, e.g. Trust region Newton.

Overall comparison of the methods

Gradient descent method is slow and should be avoided as much as possible.

Conjugate gradient methods are simple, need little memory, and, if implemented carefully, are very much faster.

Quasi-Newton methods are robust. But, they require $O(m^2)$ memory space (m is number of variables) to store the approximate Hessian inverse, so they are not suited for large scale problems. Limited Memory Quasi-Newton methods use $O(m)$ memory (like CG and gradient descent) and they are suited for large scale problems.

In many problems Hd evaluation is fast and, for them Newton-type methods are excellent candidates, e.g. Trust region Newton.

Simple Bound Constraints

Example. L_1 regularization: $\min_w \sum_j |w^j| + C\mathcal{L}(w)$ Compared to using $\|w\|^2 = \sum_j (w^j)^2$ the use of the L_1 norm causes all irrelevant w^j variables to go to zero. Thus feature selection is neatly achieved.

Problem: L_1 norm is non-differentiable.
Take care of this by introducing two variables $w_p^j \geq 0$, $w_n^j \geq 0$, setting $w^j = w_p^j - w_n^j$ and $|w^j| = w_p^j + w_n^j$ so that we have

$$\min_{w_p \geq 0, w_n \geq 0} \sum_j (w_p^j + w_n^j) + C\mathcal{L}(w_p - w_n)$$

Most algorithms (Newton, BFGS etc) have modified versions that can tackle the simple constrained problem.

Simple Bound Constraints

Example. L_1 regularization: $\min_w \sum_j |w^j| + C\mathcal{L}(w)$ Compared to using $\|w\|^2 = \sum_j (w^j)^2$ the use of the L_1 norm causes all irrelevant w^j variables to go to zero. Thus feature selection is neatly achieved.

Problem: L_1 norm is non-differentiable.
Take care of this by introducing two variables $w_p^j \geq 0$, $w_n^j \geq 0$, setting $w^j = w_p^j - w_n^j$ and $|w^j| = w_p^j + w_n^j$ so that we have

$$\min_{w_p \geq 0, w_n \geq 0} \sum_j (w_p^j + w_n^j) + C\mathcal{L}(w_p - w_n)$$

Most algorithms (Newton, BFGS etc) have modified versions that can tackle the simple constrained problem.

Simple Bound Constraints

Example. L_1 regularization: $\min_w \sum_j |w^j| + C\mathcal{L}(w)$ Compared to using $\|w\|^2 = \sum_j (w^j)^2$ the use of the L_1 norm causes all irrelevant w^j variables to go to zero. Thus feature selection is neatly achieved.

Problem: L_1 norm is non-differentiable.
Take care of this by introducing two variables $w_p^j \geq 0$, $w_n^j \geq 0$, setting $w^j = w_p^j - w_n^j$ and $|w^j| = w_p^j + w_n^j$ so that we have

$$\min_{w_p \geq 0, w_n \geq 0} \sum_j (w_p^j + w_n^j) + C\mathcal{L}(w_p - w_n)$$

Most algorithms (Newton, BFGS etc) have modified versions that can tackle the simple constrained problem.

Stochastic methods

Deterministic methods

The methods we have covered till now are based on using the "full" gradient of the training objective function. They are *deterministic* in the sense that, from the same starting w_0, these methods will produce exactly the same sequence of weight updates each time they are run.

Stochastic methods

These methods are based on partial gradients with randomness in-built; they are also a very effective class of methods.

Stochastic methods

Deterministic methods

The methods we have covered till now are based on using the "full" gradient of the training objective function. They are *deterministic* in the sense that, from the same starting w_0, these methods will produce exactly the same sequence of weight updates each time they are run.

Stochastic methods

These methods are based on partial gradients with randomness in-built; they are also a very effective class of methods.

Objective function as an expectation

The original objective

$$E = \|w\|^2 + C \sum_{i=1}^{nex} L_i(w)$$

Multiply through by $\lambda = 1/(C * nex)$

$$\tilde{E} = \lambda \|w\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} L_i(w) = \frac{1}{nex} \sum_{i=1}^{nex} \tilde{L}_i(w) = Exp \;\; \tilde{L}_i(w)$$

where $\tilde{L}_i(w) = \lambda \|w\|^2 + L_i$ and *Exp* denotes *Expectation over examples*. Gradient: $\nabla \tilde{E}(w) = Exp \;\; \nabla \tilde{L}_i(w)$

Stochastic methods

Update w using a sample of examples, e.g., just one example.

Objective function as an expectation

The original objective

$$E = \|w\|^2 + C \sum_{i=1}^{nex} L_i(w)$$

Multiply through by $\lambda = 1/(C * nex)$

$$\tilde{E} = \lambda\|w\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} L_i(w) = \frac{1}{nex} \sum_{i=1}^{nex} \tilde{L}_i(w) = Exp \;\; \tilde{L}_i(w)$$

where $\tilde{L}_i(w) = \lambda\|w\|^2 + L_i$ and *Exp* denotes *Expectation over examples*. Gradient: $\nabla\tilde{E}(w) = Exp \;\; \nabla\tilde{L}_i(w)$

Stochastic methods

Update w using a sample of examples, e.g., just one example.

Objective function as an expectation

The original objective

$$E = \|w\|^2 + C \sum_{i=1}^{nex} L_i(w)$$

Multiply through by $\lambda = 1/(C * nex)$

$$\tilde{E} = \lambda\|w\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} L_i(w) = \frac{1}{nex} \sum_{i=1}^{nex} \tilde{L}_i(w) = Exp \;\; \tilde{L}_i(w)$$

where $\tilde{L}_i(w) = \lambda\|w\|^2 + L_i$ and *Exp* denotes *Expectation over examples*. Gradient: $\nabla\tilde{E}(w) = Exp \;\; \nabla\tilde{L}_i(w)$

Stochastic methods

Update w using a sample of examples, e.g., just one example.

Stochastic Gradient Descent (SGD)

Steps of SGD

1. Repeat the following steps for many epochs.
2. In each epoch, randomly shuffle the dataset
3. Repeat, for each i: $w \leftarrow w - \eta \nabla \tilde{L}_i(w)$

Mini-batch SGD

- In step 2, form random sets of mini-batches
- In step 3, do for each mini-batch set MB: $w \leftarrow w - \eta \frac{1}{m} \sum_{i \in MB} \nabla \tilde{L}_i(w)$

Need for random shuffling in step 2

Any systematic ordering of examples will lead to poor or slow convergence.

Stochastic Gradient Descent (SGD)

Steps of SGD

1. Repeat the following steps for many epochs.
2. In each epoch, randomly shuffle the dataset
3. Repeat, for each i: $w \leftarrow w - \eta \nabla \tilde{L}_i(w)$

Mini-batch SGD

- In step 2, form random sets of mini-batches
- In step 3, do for each mini-batch set MB: $w \leftarrow w - \eta \frac{1}{m} \sum_{i \in MB} \nabla \tilde{L}_i(w)$

Need for random shuffling in step 2

Any systematic ordering of examples will lead to poor or slow convergence.

Stochastic Gradient Descent (SGD)

Steps of SGD

1. Repeat the following steps for many epochs.
2. In each epoch, randomly shuffle the dataset
3. Repeat, for each i: $w \leftarrow w - \eta \nabla \tilde{L}_i(w)$

Mini-batch SGD

- In step 2, form random sets of mini-batches
- In step 3, do for each mini-batch set MB: $w \leftarrow w - \eta \frac{1}{m} \sum_{i \in MB} \nabla \tilde{L}_i(w)$

Need for random shuffling in step 2

Any systematic ordering of examples will lead to poor or slow convergence.

Pros and Cons

Speed

Unlike (batch) gradient methods they don't have to wait for a full round (epoch) over all examples to do an update.

Variance

Since each update uses a small sample of examples, the behavior will be jumpy.

Jumpiness even at optimality

$\nabla \tilde{E}(w) = Exp\ \nabla \tilde{L}_i(w) = 0$ does not mean $\nabla \tilde{L}_i(w)$ or a mean over a small sample will be zero.

Pros and Cons

Speed

Unlike (batch) gradient methods they don't have to wait for a full round (epoch) over all examples to do an update.

Variance

Since each update uses a small sample of examples, the behavior will be jumpy.

Jumpiness even at optimality

$\nabla \tilde{E}(w) = Exp\ \ \nabla \tilde{L}_i(w) = 0$ does not mean $\nabla \tilde{L}_i(w)$ or a mean over a small sample will be zero.

Pros and Cons

Speed

Unlike (batch) gradient methods they don't have to wait for a full round (epoch) over all examples to do an update.

Variance

Since each update uses a small sample of examples, the behavior will be jumpy.

Jumpiness even at optimality

$\nabla \tilde{E}(w) = Exp\ \ \nabla \tilde{L}_i(w) = 0$ does not mean $\nabla \tilde{L}_i(w)$ or a mean over a small sample will be zero.

SGD Improvements

Greatly researched topic over many years

- Momentum, Nesterov accelerated gradient are examples;
- Make learning rate adaptive.
- There is rich theory.
- There are methods which reduce variance + improve convergence.

(Deep) Neural Networks

- Mini-batch SGD variants are the most popular.
- Need to deal with non-convex objective functions
- Objective functions also have special ill-conditionings
- Need for separate adaptive learning rates for each weight
- Methods - Adagrad, Adadelta, RMSprop, Adam (currently the most popular)

SGD Improvements

Greatly researched topic over many years

- Momentum, Nesterov accelerated gradient are examples;
- Make learning rate adaptive.
- There is rich theory.
- There are methods which reduce variance + improve convergence.

(Deep) Neural Networks

- Mini-batch SGD variants are the most popular.
- Need to deal with non-convex objective functions
- Objective functions also have special ill-conditionings
- Need for separate adaptive learning rates for each weight
- Methods - Adagrad, Adadelta, RMSprop, Adam (currently the most popular)

Dual methods

Primal

$$\min_w \tilde{E} = \lambda \|w\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} L_i(w)$$

Dual

$$\max_\alpha D(\alpha) = \lambda \|w(\alpha)\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} -\phi(\alpha_i)$$

- α has dimension nex - there is one α_i for each example i
- $\phi(\cdot)$ is the conjugate of the loss function
- $w(\alpha) = \frac{2}{nex} \sum_{i=1}^{nex} \alpha_i x_i$
- Always $E(w) \geq D(\alpha)$; At optimality, $E(w) = D(\alpha)$ and $w^* = w(\alpha^*)$.

Dual methods

Primal

$$\min_w \tilde{E} = \lambda \|w\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} L_i(w)$$

Dual

$$\max_\alpha D(\alpha) = \lambda \|w(\alpha)\|^2 + \frac{1}{nex} \sum_{i=1}^{nex} -\phi(\alpha_i)$$

- α has dimension *nex* - there is one α_i for each example i
- $\phi(\cdot)$ is the conjugate of the loss function
- $w(\alpha) = \frac{2}{nex} \sum_{i=1}^{nex} \alpha_i x_i$
- Always $E(w) \geq D(\alpha)$; At optimality, $E(w) = D(\alpha)$ and $w^* = w(\alpha^*)$.

Dual Coordinate Ascent Methods

Dual Coordinate Ascent (DCA)

Epochs-wise updating

1. Repeat the following steps for many epochs.
2. In each epoch, randomly shuffle the dataset.
3. Repeat, for each i: maximize D with respect to α_i only, keeping all other α variables fixed.

Stochastic Dual Coordinate Ascent (SDCA)

There is no epochs-wise updating.

1. Repeat the following steps.
2. Choose a random example i with uniform distribution.
3. maximize D with respect to α_i only, keeping all other α variables fixed.

Dual Coordinate Ascent Methods

Dual Coordinate Ascent (DCA)

Epochs-wise updating

1. Repeat the following steps for many epochs.
2. In each epoch, randomly shuffle the dataset.
3. Repeat, for each i: maximize D with respect to α_i only, keeping all other α variables fixed.

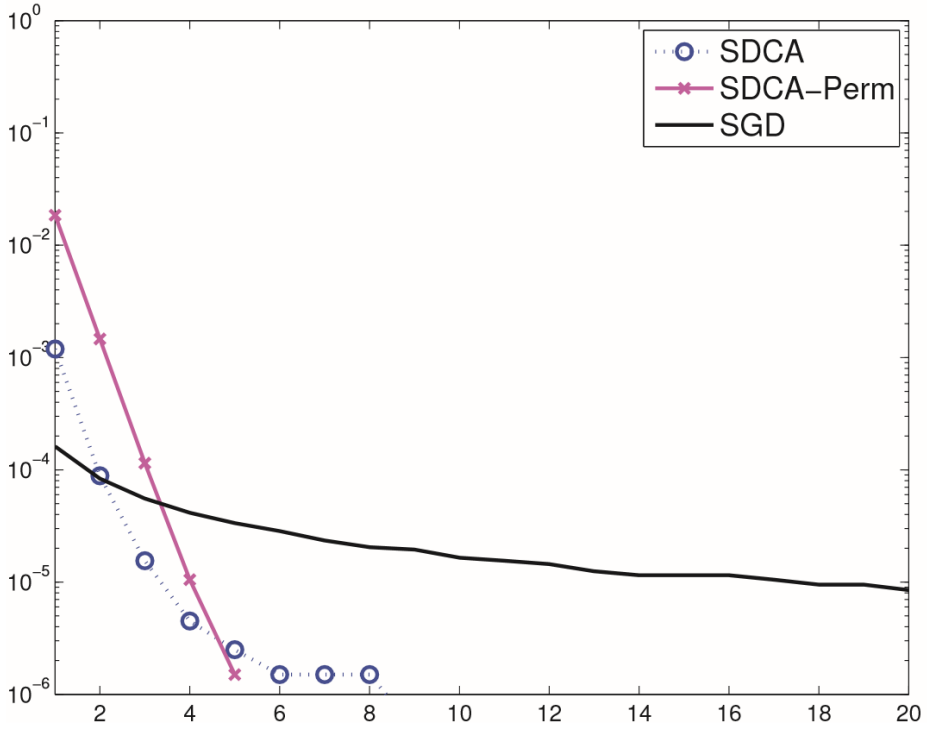
Stochastic Dual Coordinate Ascent (SDCA)

There is no epochs-wise updating.

1. Repeat the following steps.
2. Choose a random example i with uniform distribution.
3. maximize D with respect to α_i only, keeping all other α variables fixed.

Convergence

DCA (SDCA-Perm) and SDCA methods enjoy linear convergence.



References for Stochastic Methods

SGD

http://sebastianruder.com/optimizing-gradient-descent/

http://cs231n.github.io/neural-networks-3/#update

http://en.wikipedia.org/wiki/Stochastic_gradient_descent

DCA, SDCA

https://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf

http://www.jmlr.org/papers/volume14/shalev-shwartz13a/shalev-shwartz13a.pdf

References for Stochastic Methods

SGD

http://sebastianruder.com/optimizing-gradient-descent/

http://cs231n.github.io/neural-networks-3/#update

http://en.wikipedia.org/wiki/Stochastic_gradient_descent

DCA, SDCA

https://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf

http://www.jmlr.org/papers/volume14/shalev-shwartz13a/shalev-shwartz13a.pdf

Least Squares loss for Classification

$$L(y, t) = (y - t)^2$$

with $t \in \{1, -1\}$ as above for logistic and SVM losses.

Although one may have doubting questions such as:
"Why should we penalize $(y - t)$ *when, say,* $t = 1$ *and* $y > 1$*?"*,
the method works surprisingly well in practice!

Least Squares loss for Classification

$$L(y, t) = (y - t)^2$$

with $t \in \{1, -1\}$ as above for logistic and SVM losses.

Although one may have doubting questions such as:
"Why should we penalize $(y - t)$ *when, say,* $t = 1$ *and* $y > 1$*?"*,
the method works surprisingly well in practice!

Multi-Class Models

Decision functions

One weight vector w_c for each class c. $w_c^T x$ is the score for class c. Classifier chooses $\arg\max_c w_c^T x$

Logistic loss (differentiable)

Negative log-likelihood of target class t:

$$p(t|x) = \frac{\exp(w_t^T x)}{Z}, \qquad Z = \sum_c \exp(w_c^T x)$$

Multi-class SVM loss (non-differentiable)

$$L = \max_c [w_c^T x - w_t^T x + \Delta(c, t)]$$

$\Delta(c, t)$ is penalty for classifying t as c.

Multi-Class Models

Decision functions

One weight vector w_c for each class c. $w_c^T x$ is the score for class c. Classifier chooses $\arg\max_c w_c^T x$

Logistic loss (differentiable)

Negative log-likelihood of target class t:

$$p(t|x) = \frac{\exp(w_t^T x)}{Z}, \qquad Z = \sum_c \exp(w_c^T x)$$

Multi-class SVM loss (non-differentiable)

$$L = \max_c [w_c^T x - w_t^T x + \Delta(c, t)]$$

$\Delta(c, t)$ is penalty for classifying t as c.

Multi-Class Models

Decision functions

One weight vector w_c for each class c. $w_c^T x$ is the score for class c. Classifier chooses $\arg\max_c w_c^T x$

Logistic loss (differentiable)

Negative log-likelihood of target class t:

$$p(t|x) = \frac{\exp(w_t^T x)}{Z}, \qquad Z = \sum_c \exp(w_c^T x)$$

Multi-class SVM loss (non-differentiable)

$$L = \max_c [w_c^T x - w_t^T x + \Delta(c, t)]$$

$\Delta(c, t)$ is penalty for classifying t as c.

Multi-Class: One Versus Rest Approach

For each c develop a binary classifier $w_c^T x$ that helps differentiate class c from all other classes.

Then apply the usual multi-class decision function for inference: $\arg\max_c w_c^T x$

This simple approach works very well in practice.
Given the decoupled nature of the optimization, the approach also turns out to be very efficient in training.

Multi-Class: One Versus Rest Approach

For each c develop a binary classifier $w_c^T x$ that helps differentiate class c from all other classes.

Then apply the usual multi-class decision function for inference: $\arg\max_c w_c^T x$

This simple approach works very well in practice.
Given the decoupled nature of the optimization, the approach also turns out to be very efficient in training.

Multi-Class: One Versus Rest Approach

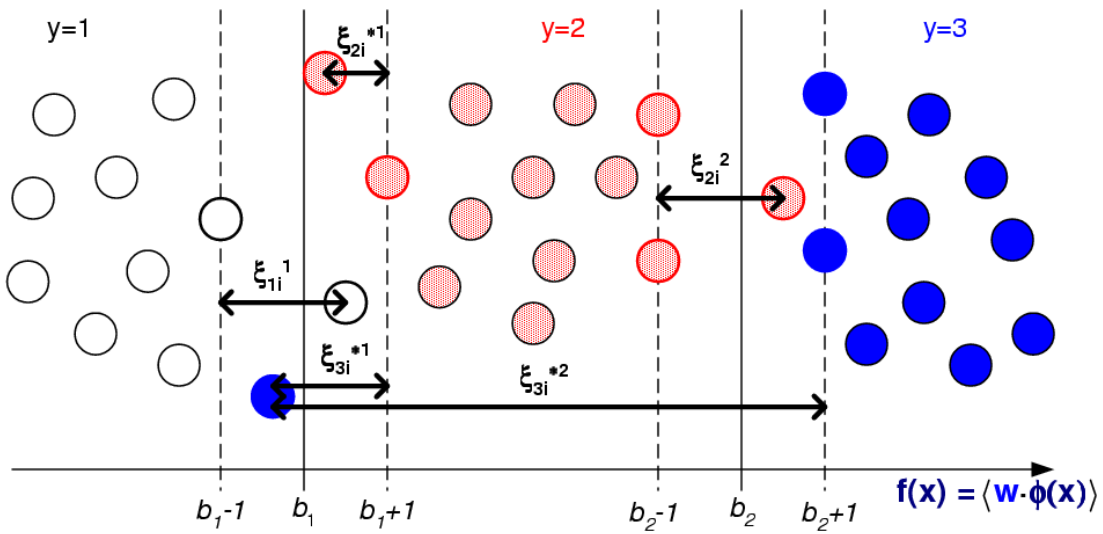
For each c develop a binary classifier $w_c^T x$ that helps differentiate class c from all other classes.

Then apply the usual multi-class decision function for inference: $\arg\max_c w_c^T x$

This simple approach works very well in practice.
Given the decoupled nature of the optimization, the approach also turns out to be very efficient in training.

Ordinal Regression

Only difference from *multi-class*: Same scoring function $w^T x$ for all classes, but different thresholds, which form additional parameters that can be included in w.



Collaborative Prediction via Max-Margin Factorization

Applications: Predicting users ratings for movies, music

Low dimensional factor model

U $(n \times k)$: Representation of n users by the k factors
V $(d \times k)$: Representation of d items by the k factors
Rating matrix: $Y = UV^T$

Known target ratings

T $(n \times d)$: True user ratings of items.
S $(n \times d)$: Sparse indicator matrix of combinations for which ratings are available for training.

`http://people.csail.mit.edu/jrennie/matlab/`

Collaborative Prediction via Max-Margin Factorization

Applications: Predicting users ratings for movies, music

Low dimensional factor model

U $(n \times k)$: Representation of n users by the k factors
V $(d \times k)$: Representation of d items by the k factors
Rating matrix: $Y = UV^T$

Known target ratings

T $(n \times d)$: True user ratings of items.
S $(n \times d)$: Sparse indicator matrix of combinations for which ratings are available for training.

`http://people.csail.mit.edu/jrennie/matlab/`

Collaborative Prediction via Max-Margin Factorization

Applications: Predicting users ratings for movies, music

Low dimensional factor model

U $(n \times k)$: Representation of n users by the k factors
V $(d \times k)$: Representation of d items by the k factors
Rating matrix: $Y = UV^T$

Known target ratings

T $(n \times d)$: True user ratings of items.
S $(n \times d)$: Sparse indicator matrix of combinations for which ratings are available for training.

http://people.csail.mit.edu/jrennie/matlab/

Collaborative Prediction via Max-Margin Factorization

Applications: Predicting users ratings for movies, music

Low dimensional factor model

U $(n \times k)$: Representation of n users by the k factors
V $(d \times k)$: Representation of d items by the k factors
Rating matrix: $Y = UV^T$

Known target ratings

T $(n \times d)$: True user ratings of items.
S $(n \times d)$: Sparse indicator matrix of combinations for which ratings are available for training.

`http://people.csail.mit.edu/jrennie/matlab/`

Collaborative Prediction: Optimization

Optimization: $\min_{U,V} E = \mathcal{R} + C\mathcal{L}$

Regularizer $\mathcal{R} = \|U\|_F^2 + \|V\|_F^2$ (F is Frobenius)
Loss $\mathcal{L} = \sum_{(i,j)\in S} L(Y_{ij}, T_{ij})$ where L is a suitable loss (e.g. from ordinal regression)

Gradient evaluations and Hessian times vector operations can be efficiently done.

Collaborative Prediction: Optimization

Optimization: $\min_{U,V} E = \mathcal{R} + C\mathcal{L}$

Regularizer $\mathcal{R} = \|U\|_F^2 + \|V\|_F^2$ (F is Frobenius)
Loss $\mathcal{L} = \sum_{(i,j)\in S} L(Y_{ij}, T_{ij})$ where L is a suitable loss (e.g. from ordinal regression)

Gradient evaluations and Hessian times vector operations can be efficiently done.

Complex Outputs: e.g. Sequence Tagging

One example: Input and Target

$x = \{x_j\}$ is sequence of tokens (e.g. properties of word in sentence)
$t = \{t_j\}$ is a sequence of tags (e.g. part of speech)

Basic Token Weights

Tags (classes) $\in C$. For each c have a weight vector w_c to compute $w_c^T x_j$ (view it as the base score for class c for word x_j).

Transition Weights

For each $c, \tilde{c} \in C$, have a weight $w_{c\tilde{c}}^{\text{transition}}$: the strength of transiting from tag c at $j-1$ to tag $\tilde{c}$ at the next sequence point j.

Note: If the transition weights are absent then the problem is a pure multi-class problem with the individual tokens acting as examples.

Complex Outputs: e.g. Sequence Tagging

One example: Input and Target

$x = \{x_j\}$ is sequence of tokens (e.g. properties of word in sentence)
$t = \{t_j\}$ is a sequence of tags (e.g. part of speech)

Basic Token Weights

Tags (classes) $\in C$. For each c have a weight vector w_c to compute $w_c^T x_j$ (view it as the base score for class c for word x_j).

Transition Weights

For each $c, \tilde{c} \in C$, have a weight $w_{c\tilde{c}}^{\text{transition}}$: the strength of transiting from tag c at $j-1$ to tag $\tilde{c}$ at the next sequence point j.

Note: If the transition weights are absent then the problem is a pure multi-class problem with the individual tokens acting as examples.

Complex Outputs: e.g. Sequence Tagging

One example: Input and Target

$x = \{x_j\}$ is sequence of tokens (e.g. properties of word in sentence)
$t = \{t_j\}$ is a sequence of tags (e.g. part of speech)

Basic Token Weights

Tags (classes) $\in C$. For each c have a weight vector w_c to compute $w_c^T x_j$ (view it as the base score for class c for word x_j).

Transition Weights

For each $c, \tilde{c} \in C$, have a weight $w_{c\tilde{c}}^{\text{transition}}$: the strength of transiting from tag c at $j-1$ to tag $\tilde{c}$ at the next sequence point j.

Note: If the transition weights are absent then the problem is a pure multi-class problem with the individual tokens acting as examples.

Complex Outputs: e.g. Sequence Tagging

One example: Input and Target

$x = \{x_j\}$ is sequence of tokens (e.g. properties of word in sentence)
$t = \{t_j\}$ is a sequence of tags (e.g. part of speech)

Basic Token Weights

Tags (classes) $\in C$. For each c have a weight vector w_c to compute $w_c^T x_j$ (view it as the base score for class c for word x_j).

Transition Weights

For each $c, \tilde{c} \in C$, have a weight $w_{c\tilde{c}}^{\text{transition}}$: the strength of transiting from tag c at $j-1$ to tag $\tilde{c}$ at the next sequence point j.

Note: If the transition weights are absent then the problem is a pure multi-class problem with the individual tokens acting as examples.

Complex Outputs: Decision function

$$\arg \max_{y=\{y_j\}} f(y) = \sum_j [w_{y_j}^T x_j + w_{y_{j-1}y_j}^{\text{transition}}]$$

Note: y_0 can be taken as the special tag denoting the beginning of a sentence.
Decision function efficiently evaluated using the Viterbi algorithm.

Models

Conditional Random Fields (CRFs) (involves differentiable nonlinear optimization)
SVMs for structured outputs (involves non-differentiable optimization)

Complex Outputs: Decision function

$$\arg \max_{y=\{y_j\}} f(y) = \sum_j [w_{y_j}^T x_j + w_{y_{j-1}y_j}^{\text{transition}}]$$

Note: y_0 can be taken as the special tag denoting the beginning of a sentence.
Decision function efficiently evaluated using the Viterbi algorithm.

Models

Conditional Random Fields (CRFs) (involves differentiable nonlinear optimization)
SVMs for structured outputs (involves non-differentiable optimization)

CRFs

A good tutorial: `https://people.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf`

Probability of $t = \{t_j\}$: $p(t) = \exp(f(t))/Z \quad Z = \sum_y \exp(f(y))$
Z is called the partition function. Note its complexity: it involves summation over all possible $y = \{y_j\}$.

Computation of Z as well as the gradient of $E = \mathcal{R} + C\mathcal{L}$ (as in logistic models, $\mathcal{L}$ is the negative log-likelihood of all examples) can be efficiently done using forward-backward recursions.
Hd computation by using complex arithmetic:
$\nabla E(w + i\epsilon d) = \nabla E(w) + i\epsilon Hd + O(\epsilon^2)$

See eqn (12) of `http://www.cs.ubc.ca/~murphyk/Papers/icml06_camera.pdf`.

CRFs

A good tutorial: `https://people.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf`

Probability of $t = \{t_j\}$: $p(t) = \exp(f(t))/Z \quad Z = \sum_y \exp(f(y))$
Z is called the partition function. Note its complexity: it involves summation over all possible $y = \{y_j\}$.

Computation of Z as well as the gradient of $E = \mathcal{R} + C\mathcal{L}$ (as in logistic models, $\mathcal{L}$ is the negative log-likelihood of all examples) can be efficiently done using forward-backward recursions.
Hd computation by using complex arithmetic:
$\nabla E(w + i\epsilon d) = \nabla E(w) + i\epsilon Hd + O(\epsilon^2)$

See eqn (12) of `http://www.cs.ubc.ca/~murphyk/Papers/icml06_camera.pdf`.

CRFs

A good tutorial: `https://people.cs.umass.edu/~mccallum/papers/crf-tutorial.pdf`

Probability of $t = \{t_j\}$: $p(t) = \exp(f(t))/Z \quad Z = \sum_y \exp(f(y))$
Z is called the partition function. Note its complexity: it involves summation over all possible $y = \{y_j\}$.

Computation of Z as well as the gradient of $E = \mathcal{R} + C\mathcal{L}$ (as in logistic models, $\mathcal{L}$ is the negative log-likelihood of all examples) can be efficiently done using forward-backward recursions.
Hd computation by using complex arithmetic:
$\nabla E(w + i\epsilon d) = \nabla E(w) + i\epsilon Hd + O(\epsilon^2)$

See eqn (12) of `http://www.cs.ubc.ca/~murphyk/Papers/icml06_camera.pdf`.

Other models that use Nonlinear optimization

Neural networks

Training of weights of multi-layer perceptrons and RBF networks. Gradient evaluation efficiently done by backpropagation. Efficient Hessian operations can also be done.

Hyperparameter tuning

In SVM, Logistic and Gaussian Process models (particularly in their nonlinear versions) there can be many hyperparameters present (e.g. individual feature weighting parameters) which are usually tuned by optimizing a differentiable validation function.

Semi-supervised learning

Make use of unlabeled examples to improve classification. Involves an interesting set of nonlinear optimization problems. `http://twiki.corp.yahoo.com/view/YResearch/SemisupervisedLearning`

Other models that use Nonlinear optimization

Neural networks

Training of weights of multi-layer perceptrons and RBF networks. Gradient evaluation efficiently done by backpropagation. Efficient Hessian operations can also be done.

Hyperparameter tuning

In SVM, Logistic and Gaussian Process models (particularly in their nonlinear versions) there can be many hyperparameters present (e.g. individual feature weighting parameters) which are usually tuned by optimizing a differentiable validation function.

Semi-supervised learning

Make use of unlabeled examples to improve classification.
Involves an interesting set of nonlinear optimization problems.
`http://twiki.corp.yahoo.com/view/YResearch/SemisupervisedLearning`

Other models that use Nonlinear optimization

Neural networks

Training of weights of multi-layer perceptrons and RBF networks. Gradient evaluation efficiently done by backpropagation. Efficient Hessian operations can also be done.

Hyperparameter tuning

In SVM, Logistic and Gaussian Process models (particularly in their nonlinear versions) there can be many hyperparameters present (e.g. individual feature weighting parameters) which are usually tuned by optimizing a differentiable validation function.

Semi-supervised learning

Make use of unlabeled examples to improve classification. Involves an interesting set of nonlinear optimization problems.
`http://twiki.corp.yahoo.com/view/YResearch/SemisupervisedLearning`

Some Concluding Remarks

Optimization plays a key role in the design of ML models. A good knowledge comes in quite handy.

Only differentiable nonlinear optimization methods were covered.

Quadratic programming, Linear programming and Non-differentiable optimization methods also find use in several ML situations.

Some Concluding Remarks

Optimization plays a key role in the design of ML models. A good knowledge comes in quite handy.

Only differentiable nonlinear optimization methods were covered.

Quadratic programming, Linear programming and Non-differentiable optimization methods also find use in several ML situations.

Some Concluding Remarks

Optimization plays a key role in the design of ML models. A good knowledge comes in quite handy.

Only differentiable nonlinear optimization methods were covered.

Quadratic programming, Linear programming and Non-differentiable optimization methods also find use in several ML situations.