Lecture IV: Training predictors, Part I

Marina Meilă
mmp@stat.washington.edu

Department of Statistics
University of Washington

October, 2022

Analytic optimization

Optimization generalities

- Optimization glossary
- Descent methods zoo

Descent methods

- The steepest descent method
- Line minimization algorithms
- The Newton-Raphson method
- Examples: Logistic regression and Backpropagation

Reading HTF Ch.: Lasso 3.1,2,4, Logistic regression 4.4, Neural networks 11, Murphy Ch.: Ridge regression (including numerics) 7.5, Descent methods 8.3.2,3,5, Neural networks 16.5.1–4, Autoencoders 28, Bach Ch.: 5.

For more mathematical background, look e.g. at "Numerical recipes" chapter 10 or, for really advanced treatment Nocedal and Wrigth (Ch 3, 6).

The typical training problem

Given $\mathcal{D} = \{(x^1, y^1), \ldots, (x^n, y^n)\}$ data set (and implicitly a prediction task)
$\mathcal{F}$ class of predictors (e.g Linear $\{f(x) = \beta^T x\}$, quadratic $f(x) = x^T A x + \beta^T x + \gamma$, neural net)
and $J(f; \mathcal{D})$ **objective function**
find $f = \underset{\mathcal{F}}{\operatorname{argmin}}\, J(f; \mathcal{D})$

Remarks

- Typically objective functions J

$$J(f; \mathcal{D}) = \hat{L}(f) \quad \text{empirical loss} \tag{1}$$

OR

$$J_\lambda(f; \mathcal{D}) = \hat{L}(f) + \lambda R(f) \quad \textbf{regularized} \text{ objective} \tag{2}$$

$R(f)$ is called regularization (functional) and $\lambda \geq 0$ is the **regularization constant (or parameter)**
λ is fixed in the minimization of J. Setting λ can be thought as a form of model selection; as we shall see later λ plays the role of a smoothing parameter, or hyperparameter

- This lecture: algorithms for minimizing J's.

Linear Least Squares regression

- Problem $\mathcal{F} = \{f(x) = \beta^T x\}$, $L_{LS}(y, f(x)) = (y - f(x))^2$, $y \in \mathbb{R}$, $J = \hat{L}$
- Solution
  - Finding $f \in \mathcal{F}$ is equivalent to finding $\beta \in \mathbb{R}^d$ (or $\mathbb{R}^{d+1}$)

Linear Least Squares regression

- Problem $\mathcal{F} = \{f(x) = \beta^T x\}$, $L_{LS}(y, f(x)) = (y - f(x))^2$, $y \in \mathbb{R}$, $J = \hat{L}$
- Solution
  - Finding $f \in \mathcal{F}$ is equivalent to finding $\beta \in \mathbb{R}^d$ (or $\mathbb{R}^{d+1}$)
  - define **data matrix** or (transpose) **design matrix**

$$\mathbf{X} = \begin{bmatrix} (x^1)^T \\ (x^2)^T \\ \cdots \\ (x^i)^T \\ \cdots \\ (x^n)^T \end{bmatrix} \in \mathbb{R}^{n \times d} \quad \text{and} \quad Y = \begin{bmatrix} y^1 \\ y^2 \\ \cdots \\ y^n \end{bmatrix}, \quad E = \begin{bmatrix} \varepsilon^1 \\ \varepsilon^2 \\ \cdots \\ \varepsilon^n \end{bmatrix} \in \mathbb{R}^d$$

  - Then we can write

$$Y = X\beta + E$$

  - The solution $\hat{\beta}$ is chosen to minimize the sum of the squared errors

$$J(\beta) = \hat{L}_{LS} = \sum_{i=1}^{n} (y^i - \beta^T x_i)^2 = ||E||^2 \tag{3}$$

which gives Exercise Derive it

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y \tag{4}$$

Linear Least Squares regression

- Problem $\mathcal{F} = \{f(x) = \beta^T x\}$, $L_{LS}(y, f(x)) = (y - f(x))^2$, $y \in \mathbb{R}$, $J = \hat{L}$
- Solution
  - Finding $f \in \mathcal{F}$ is equivalent to finding $\beta \in \mathbb{R}^d$ (or $\mathbb{R}^{d+1}$)
  - define **data matrix** or (transpose) **design matrix**

$$\mathbf{X} = \begin{bmatrix} (x^1)^T \\ (x^2)^T \\ \dots \\ (x^i)^T \\ \dots \\ (x^n)^T \end{bmatrix} \in \mathbb{R}^{n \times d} \quad \text{and} \quad Y = \begin{bmatrix} y^1 \\ y^2 \\ \dots \\ y^n \end{bmatrix}, \quad E = \begin{bmatrix} \varepsilon^1 \\ \varepsilon^2 \\ \dots \\ \varepsilon^n \end{bmatrix} \in \mathbb{R}^d$$

  - Then we can write

$$Y = X\beta + E$$

  - The solution $\hat{\beta}$ is chosen to minimize the sum of the squared errors

$$J(\beta) = \hat{L}_{LS} = \sum_{i=1}^{n} (y^i - \beta^T x_i)^2 = ||E||^2 \tag{3}$$

    which gives Exercise Derive it

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y \tag{4}$$

- Underlying statistical model $y = \beta^T x + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2)$ (and $(x^{1:n}, y^{1:n})$ sampled i.i.d., of course)
  - Define the negative log-likelihood $L_{logl} = -\ln P(y|x, \beta)$
  - Then, $(y - \beta^T x)^2 = \sum_{i=1}^{n} (\varepsilon^i)^2 = -2\sigma^2 \ln P(y^i|x^i, \beta) = -2\sigma^2 \hat{L}_{logl}$
  - Hence, $\hat{\beta}$ from (4) is the Maximum Likelihood (ML) estimator of the parameter β and the minimizer of $\hat{L}_{logl}$
- This is an example where the minimizer of $J(f)$ has an analytical formula. A few more examples of this kind follow.

Regularized Linear Regression

- L2 regularized (linear) LS regression (**Ridge regression**)

$$J(f;\mathcal{D}) = \sum_{i=1}^{n}(y^i - \beta^T x_i)^2 + \lambda||\beta||^2 \tag{5}$$

Solution $\hat{\beta} = (\mathbf{X}^T\mathbf{X} + \lambda I_n)^{-1}\mathbf{X}^T Y$ Exercise Derive it.

- L1 regularized (linear) regression (**LASSO**)

$$J(f;\mathcal{D}) = \sum_{i=1}^{n}(y^i - \beta^T x_i)^2 + \lambda||\beta||_1 \tag{6}$$

By contrast with the previos problem(s), this one does not have an analytic solution.

Regularized Linear Regression

Generative models for classification

- If each class conditional distribution $P_{X|Y=y}$ can be estimated by an analytic formula, then the generative classifier can be estimated analytically
- Examples LDA, QDA

Example (Naive Bayes for text classification)

The **bag of words** model of text. Let $D = \{\text{words}\}$ be a dictonary. For simplicity, we shall assume $D = \{$ and, average, batting, score, variance$\}$. We are given a **corpus** $\mathcal{D}$ containing N documents of two classes $\{$ sports$= -1$, statistics$= 1\}$. For each document in $\mathcal{D}$, we form a vector $x \in \{0,1\}^{|D|}$ by setting $x_w = 1$ if the document contains word w and 0 otherwise. For example, the document "Reddick's batting average is 0.50" has $x = [0\,1\,1\,0\,xs0]$. The x vectors are the data to which we fit a Naive Bayes model. Assume our toy corpus is now the table on the left, and the estimated class conditional distributions are on the right.

x					y
0	1	1	0	0	-1
1	1	0	1	0	-1
1	0	1	1	0	-1
1	0	1	0	0	-1
1	0	1	0	0	-1
1	1	0	0	1	1
0	0	0	1	0	1
1	1	0	0	1	1
1	1	0	0	0	1
1	0	0	0	1	1

$P_{X_j=1\|Y=-1}$					y
.8	.4	.8	.4	0	-1
.8	.6	0	.2	.6	1

$P_Y(1) = 0.5 = p$

Example (Naive Bayes for text (continued))

Before we go further, we will "adjust" the 0 and 1 estimated probabilities for "variance" and "batting", setting them to $1/M$, respectively $1 - 1/M$ for some "large" $M = 10$ Exercise Do you think this problem occurs often in real applications? Find a statistical framework to justify the adjustment. The probability of a new document, e.g $x = [10010]$ in class 1, respectively -1 is

$$P(x|y=-1) = 0.8 \times (1-0.4) \times (1-0.8) \times 0.4 \times 0.9 \quad (7)$$

$$= 0.8^{x_1}(1-0.8)^{1-x_1} \times 0.4^{x_2}(1-0.4)^{1-x_2} \ldots \quad (8)$$

$$P(x|y=1) = 0.8 \times (1-0.6) \times (1-0.1) \times 0.2 \times (1-0.6) \quad (9)$$

The NB classifier we obtain is

$$f(x) = \underbrace{\ln \frac{p}{1-p}}_{1} + x_1 \underbrace{\ln \frac{P_{and|1}}{P_{and|-1}}}_{1} + x_2 \underbrace{\ln \frac{P_{average|1}}{P_{average|-1}}}_{2/3} + x_3 \underbrace{\ln \frac{P_{batting|1}}{P_{batting|-1}}}_{+\infty} + x_4 \underbrace{\ln \frac{P_{score|1}}{P_{score|-1}}}_{2.} + x_5 \underbrace{\ln \frac{P_{variance|1}}{P_{variance|-1}}}_{0}$$

$$+(1-x_1)\ln\frac{1-P_{and|1}}{1-P_{and|-1}} + (1-x_2)\ln\frac{1-P_{average|1}}{1-P_{average|-1}} + (1-x_3)\ln\frac{1-P_{batting|1}}{1-P_{batting|-1}} + (1-x_4)\ln$$

which evaluates to $f([1\,0\,0\,1\,0]) = \ln 1 + \ln\frac{3}{2} + \ln\frac{2}{9} + \ln 2 + \ln\frac{9}{4} = 0.405 > 0$ Notes As you saw above, we are not required to include all possible words in the dictionary Exercise Find some reasons why. A common preprocessing step stemming which aims to map words in the same word family to a single w; e.g "batting" $\rightarrow$ "bat". Sometimes stop words like "the", "and" are removed.

Most predictors can't be estimated analytically

- Unfortunately, minimizing J analytically is possible only in a handful of examples.
- In all other cases, we find $f = \underset{\mathcal{F}}{\operatorname{argmin}} J$ by numerical/iterative methods also called search (or training/learning, of course). For example
  - CART algorithm Exercise What J is the CART algorithm minimizing?
  - Perceptron algorithm (will be revisited this lecture)
- Therefore now we study generic algorithms for finding minima of functions of n variables. This is called **(numerical) optimization**.

Optimization glossary

Change in notation!

Here, f is a function to be minimized, and x the variable in the domain of the function. In a learning task, f will be replaced by an objective J like $\hat{L}_{logl}$ and x by the parameters of the predictor, e.g. $w, \beta, \theta, \ldots$.

The methods in this lecture belong to the class of **unconstrained optimization** methods.
Problem Find $\min_x f(x)$ for $x \in \mathbb{R}^d$ or $x \in D$ the **domain** of f. We assume that f is a twice differentiable function with continuous second derivatives.
Notation The **gradient** of f is the column vector

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_i}(x)\right]_{i=1}^{n} \tag{11}$$

and the **Hessian** of f is the square symmetric matrix of second partial derivatives of f

$$\nabla^2 f(x) = \left[\frac{\partial^2 f}{\partial x_i x_j}(x)\right]_{i,j=1}^{n} \tag{12}$$

Optimization basics

Local and global minima

- A **local minimum** for f is point x^* for which

$$f(x^*) \leq f(x) \quad \text{whenever } ||x - x^*|| < \epsilon$$

- A **global minimum** for f is point x^* for which

$$f(x^*) \leq f(x) \quad \text{for all } x \text{ in the domain of } f$$

  We say x^* is a **strict local/global minimum** when the above inequalities are strict for $x \neq x^*$.
- A minimum is **isolated** if it is the only local minimum in an ϵ-ball around itself.
- A **stationary point** for f is a point x^* for which $\nabla f(x^*) = 0$.

Theorem

If f has continuous second derivative everywhere in D, and $x^* \in D$ is a point for which $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*) \geq 0$ ($\nabla^2 f(x^*) > 0$) then x^* is a (nonsingular) local minimum for f.

In what follows, we will deal only with non-singular local minima. A non-singular local minimum is strict and isolated.

Descent methods

Many unconstrained optimization methods for finding a local minimum are of the form:

$$x^{k+1} = x^k + \eta^k d^k \tag{13}$$

where $d^k \in \mathbb{R}^d$ represents an **(unnormalized) direction** and $\eta^k > 0$ is a scalar called the **step size**.

Direction choice

- gradient based $d^k = -D^k \nabla f(x^k)$ with $D^k \in \mathbb{R}^{n \times n}$
  - steepest descent $D^k = I$
  - stochastic gradient (more about it later)
  - Newton-Raphson $D^k = \nabla^2 f(x^k)^{-1}$
  - conjugate gradient – implicity multistep rescaling of the axes "equivalent" to $D^k = \nabla^2 f(x^k)^{-1}$
  - quasi-Newton – implicit multistep approximation of $D^k = \nabla^2 f(x^k)^{-1}$
- non-gradient based
  - coordinate descent $d^k =$ one of the basis vectors in $\mathbb{R}^d$

Step size choice

- line minimization $\eta^k = \min_\eta f(x^k + \eta d^k)$
- Armijo rule (also called Backtracking) = search but not minimization
- constant step size $\eta^k = s$
- diminishing step size $\eta^k \to 0$; $\sum_k \eta^k = \infty$

Steepest descent

Algorithm STEEPEST-DESCENT

Input x^0 initial point

For $k = 0, 1, \ldots$

1. calculate $d^k = \nabla f(x^k)$
2. find η^k by line minimization
3. $x^{k+1} \rightarrow x^k - \eta^k d^k$

until stopping condition satisfied

Output x^{k+1}

How to evaluate an optimization method? [Optional]

- Does it converge to a minimum?

As we shall see, all the methods described here converge to a minimum, but some of the require the function f to have additional "good" properties. A method which converges for larger classes of f's is more **robust**.

- How fast?

Answer is usually in terms of **rates of convergence**
Let $e^k = x^k - x^*$ or $e^k = f(x^k) - f(x^*)$ denote the "error" at step k. Then, an algorithm has a **rate of convergence of order** p if

$$||e^{k+1}|| \leq \beta(||e^k||)^p \quad \text{for some } 0 < \beta < 1 \tag{14}$$

In the above, $p > 0$ but not necessarily an integer.
Most common cases are $p = 1$ (**linear**[1]) and $p = 2$ (quadratic). A rate of $p < 1$ is possible, and relevant for machine learning (see Part II). Superlinear scales are desirable – and often achievable.
Some modern machine learning algorithms have **sublinear** rates, e.g

$$||e^k|| \leq \frac{\beta}{k}||e^0|| \tag{15}$$

This is considered a slow convergence rate in classical optimization. Exercise Why may we like this rates in statistics/machine learning?

- Practical issues: Is it easy to implement or tune? Available software?

[1] The use of the term "linear" here is inconsistent with its use in e.g complexity theory. If an optimization algorithm is *linear*, that means that the error decreases **exponentially** with k, as $||e^k|| \leq \beta^k||e^0||$.

Computational complexity – theory and practice

In optimization problems, there are various ways of expressing the computational complexity of an algorithm:

- number of flops (floating point operations) per iteration, usually as a function of n the dimension of the problem
- *number of function* or gradient evaluations per iteration
- *number of iterations*; this latter quantity is given implicitly, by the rate of convergence.
- *memory requirements*
- With the increased complexity and variation of computer systems, the above mentioned number of operations is becoming obsolete. Algorithms are increasingly judged by other, system-related qualities, like: type of memory access (do they access memory in blocks or randomly), cache misses, etc. These criteria are beyond the scope of this course, but what you need to remember is that the texbook properties on an algorithm alone do not always predict its performance on the system you are going to run it. You may need to experiment with parameters and with algorithms to determine which algorithm is better suited for your data and system.

What's in a function evaluation?

In classical optimization, the computation of $f(x)$ or $\nabla f(x)$ is considered one unit of computation.
For machine learning, let us look inside the box. Take for example the LASSO objective function

$$J(\beta) = \sum_{i=1}^{n}(y^i - \beta^T x_i)^2 + \lambda||\beta||_1.$$

We notice that

- J is a sum with $n+1$ terms
- each $x^i \in \mathbb{R}^d$, $\beta \in \mathbb{R}^d$ so each term takes $\mathcal{O}(n)$ operations (additions and multiplications) to compute
- Total number of operations to compute $J(\beta)$ once is $\mathcal{O}(nd)$
- Do this exercise Exercise The gradient ∇J is in $\mathbb{R}^d$ and is also a sum of $n+1$ terms. How many operations to compute it?

Transitory and asymptotic regimes

There are two regimes for each algorithm:

- the **transitory**, or approach regime, when x^k is far away from x^*
- the **asymptotic** regime, near x^* – most classic results are about this regime

The theoretical (and practical) behavior of optimization algorithms will strongly depend on the properties of the function f to be optimized around its minimum x^*.

Not all f's are twice differentiable

By default we will assume f is twice differentiable and that its Hessian $\nabla^2 f$ is continuous and strictly positive definite around x^*. But other, weaker, conditions on f also indicate an f that is "easy" to minimize. Here are some of the most common ones.

- f is B-**Lipschitz** if there is $B > 0$ so that

$$|f(y) - f(x)| \leq B||y - x|| \quad \text{for all } x, y$$

  A Lipschitz function, behaves "almost" like a differentiable function with bounded gradient.
- The problem $\min_x f$ is a **smooth** minimization problem if f it is upper bounded by a quadratic function around x^*, i.e. iff there exists $M > 0$ so that

$$f(x) - f(x^*) \leq \frac{1}{2} M||x - x^*||^2$$

  on a neighborhood of x^*. This property indicates that f, even though it may not be differentiable, behaves "almost like a quadratic", in the sense that local quantities (gradient, Hessian) are informative w.r.t the minimum. An example of a non-smooth minimization problem is $\min_x |x|$. The gradient ∇f is ± 1 everywhere but in 0, so it gives us information on which side of x the minimum lies, but its size does not tell us how far we are from x^*.
- A function f that is lower bounded by a quadratic function around x^* is called **strongly convex** (more about this later).

$$f(y) - f(x) - \nabla f(x)^T (y - x) \geq \frac{1}{2} m||x - y||^2$$

  If f is strongly convex, then the minimum x^* is well localized.

Examples from Machine Learning

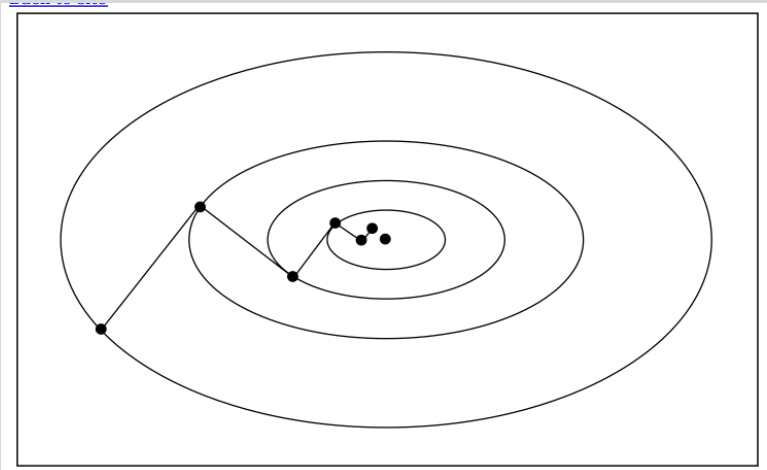
- $f(x) = |x|$ is 1-Lipschitz
- If ∇f exists and $||\nabla f(x)|| \leq B$ for all x, then f is B-Lipschitz
- A C^2 function (continuous Hessian everywhere on the domain) with $\nabla^2 f(x^*) \prec 0$ is both strongly convex and smooth
- $\hat{L}_{01}$ the misclassification cost is piecewise constant. Hence, not differentiable everywhere, not strongly convex, but smooth. Very bad J to optimize, though.
- The LASSO cost function is not differentiable everywhere, it is not smooth but strongly convex. This is a very frequent in machine learning problems.

Steepest (gradient) descent

- The steepest descent method follows the direction of the gradient.
- Rate of convergence: with line minimization or Armijo (see next slide), linear rate. For other line search methods, including constant step size, the rate of convergence is no larger.
- Ill-conditioning: The convergence coefficient β of equation (14) can get very close to 1 (very slow convergence) if the Hessian is ill conditioned.

Let M, m denote respectively the largest and the smallest eigenvalue of $\nabla^2 f(x^*)$. By continuity, we can assume that the Hessian around x^* is approximately the same. If $M >> m$ then the function will have a "long, narrow valley" with an almost flat "bottom" around x^*, oriented along the smallest eigenvector. The gradient will be almost perpendicular to the valley, and the algorithm, even with the optimal line minimization, will advance very slowly. Hence, many optimization methods (but not stochastic gradient) can be seen as "applying some coordinate transformation" that will turn the elongated ellipses into circles, so that steepest descent in this new coordinates can move rapidly towards the optimum. Equivalently, having such a transformation (which is represented by the Hessian matrix), one can apply the "inverse transformation" to the descent direction, which is precisely what the Newton-Raphson method does.

The rate of steepest descent



reproduced from Nocedal & Wright

In the ideal case of a quadratic function

$$\beta = \left(\frac{M-m}{M+m}\right)$$

This holds approximately in a neighborhood of the optimum. Hence, when $M >> m$, $\beta \to 1$ and the optimum is approached very slowly. When $M = m$, then the convergence is **super-linear** (i.e. $\lim_{k\to\infty} e^{k+1}/e^k = 0$)

Line minimization algorithms

Bracketing the minimum is optional but recommended. It means to find an interval $[0, s]$ that contains the desired η.
A more general definition of a bracketed minimum (not assuming that the function is decreasing at x^k in the direction of d^k) is to find a triplet of points on the line $a = x^k$, b, c with b between a and c and $f(b) < f(a), f(c)$. NR gives a method of finding such a triplet by starting with an initial value for b and iteratively expanding the candidate triplet.

The Armijo (Backtracking) Rule for line minimizaton

Intuition Assume that we found a bracketing interval [0, s] for η^k. We start with $\eta^k = s$ and decrease it exponentially until we find that the function f has decreased "enough". What is "enough"? In an infinitesimal interval near x^k along the direction of descent, the function will decrease linearly, hence

$$f(x^k) - f(x^k + \eta d^k) \approx \eta(-\nabla f(x^k)^T d^k) \tag{16}$$

For a finite interval, we will ask for a decrease in f that is at least $\sigma < 1$ smaller than the above. Note that for a sufficiently small η such a decrease can always be attained.

Armijo Line Search

1. Start with $\eta^k = s$, $\beta < 1$, $\sigma < 1$
2. If $f(x^k) - f(x^k + \eta^k d^k) > \sigma\eta^k(-\nabla f(x^k)^T d^k)$
   - then STOP
   - else $\eta^k \longleftarrow \beta\eta^k$ and repeat

In practice, $\beta \approx 0.5$ and $\sigma << 1$ e.g 0.1, 0.01 or even 0; $s = 1$ if no bracketing is done.

The Newton-Raphson method

Assume that our function f is quadratic, i.e

$$f(x) = \frac{1}{2}x^T Ax + b^T x + c \text{ with } A \succ 0. \tag{17}$$

Then,

$$\nabla f(x) = Ax + b \tag{18}$$
$$\nabla^2 f(x) = A \tag{19}$$

and the minimum can be computed analytically as the solution of $Ax + b = 0$, namely $x^* = -A^{-1}b$. Equivalently, for any x

$$x^* - x = -A^{-1}b - A^{-1}Ax = -A^{-1}(Ax + b) = -\nabla^2 f(x)^{-1}\nabla f(x) \tag{20}$$

Hence, if f is quadratic, from any point x we can move in one step equal to $-\nabla^2 f(x)^{-1}\nabla f(x)$ to the minimum. Therefore, the Newton-Raphson method takes $D^k = \nabla^2 f(x)^{-1}$ as if the function was quadratic. Usually one also does a line search method, i.e $\eta^k \neq 1$ in practice.

Performance of Newton-Raphson

Convergence rate:

Newton-Raphson is practically and theoretically very fast once we are in the vicinity of the optimum. However, its behavior far away from the optimum must be monitored carefully. Note for example that this is not a descent method, in the sense that it's not guaranteed that $f(x^{k+1}) < f(x^k)$ unless some form of line minimization is used. Also, the method is attracted by local maxima just as much as by local minima, so attention must be paid any time the Hessian is not positive definite.

Computation complexity

- The Hessian requires $\mathcal{O}(n^2)$ memory and $\mathcal{O}(n^2)$ operations ($\mathcal{O}(n^3 N)$ for machine learning).
- Computing the direction is done by solving the linear system $Hd = g$ where H=Hessian, g=gradient, d=direction of descent. Solving a linear system takes $\mathcal{O}(n^3)$ operations. Computation makes Newton-Raphson prohibitive in high dimensions. Much work has been done to find faster approximations to the Newton step.

NEVER INVERT A MATRIX and other rules of the numerically savvy

. . . unless the inverse matrix is the last result you need from your computations. You may not think of it much, but data in computers has finite precision, and all our results have numerical errors. Here are a few simple rules to get the same result faster and often with a smaller numerical error

- To calculate $z = A^{-1}b$ call a linear system solver for $Az = b$. This is twice as fast for general A.
- If A is symmetric (in a linear system, eigenvalue problem, etc), tell your solver. You save time (another factor of 3) and the results are much more accurate.
- (An advanced one) Solve all large linear systems iteratively.
- Computing the product $ABCd$ where A, B, C are matrices and d is a vector: $((AB)C)d$ takes $2n^3 + n^2$ operations, $A(B(Cd))$ takes $3n^2$.
- Adding small numbers to large numbers: in a computer, $1. + \varepsilon = 1.$ if $\varepsilon < 1e - 16$ or so. Use the log-sum-exp trick (Murphy) or don't waste computer time doing it (more examples of the former later).
- Ask me about: numerical precision vs convergence precision vs statistical precision

Faster approximate Newton iterations

Remember the original Newton step is $\mathcal{O}(n^3)$ (in general, not in machine learning)

- **diagonal scaling:** compute only the diagonal of the Hessian $D^k = \text{diag}\left\{\frac{\partial^2 f}{\partial x_i^2}\right\}$ A quick fix, not too effective. This method is obviously $\mathcal{O}(n)$ in storage and number of operations but it tends to underestimate the ratio M/m. Diagonal scaling amounts to rescaling each variable separately, and it is effective in those cases when the variables have very different ranges because of "imbalanced" measurement units (e.g in one direction the unit is miles, in the other one it is millimeters).
  It is useful to make also the general observation that the Newton method is "scale free", i.e it is unaffected by linear coordinate changes.
- **Conjugate directions methods** compute one exact Newton step in n conjugate steps, each taking order n operations and memory. Beautiful, but not robust.
- **Quasi-Newton (variable metric) methods** takes only order n^2 computing per iteration (or less, for the limited memory variants). See Murphy for more on Quasi-Newton. Recommended, look for the LBFGS function or option in your favorite language.

The main disadvantage of the quasi-Newton methods compared to the conjugate directions is their large computational complexity (n^2 versus n). The advantage is that the method is robust to inexact line minimization, and that it does not need restarts. In addition, near the optimum, the directions quasi-Newton generates are conjugate and so it is equivalent to conjugate gradients and Newton-Raphson and thus converges very fast.

Example: Logistic regression

Training = Estimating the parameters by Max Likelihood

Problem setup

- Denote $y_* = (1-y)/2 \in \{0,1\}$
- The likelihood of a data point is $P_{Y|X}(y|x) = \frac{e^{y_* f(x)}}{1+e^{f(x)}}$
- The log-likelihood is $l(\beta; x, y) = y_* f(x) - \ln(1+e^{f(x)})$
- Log-likelihood of the data set $\mathcal{D}$

$$l(\beta; \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} l(\beta; (x^i, y^i)) \tag{21}$$

- Define the loss function

$$L_{logl}(\beta) = -l(\beta) \tag{22}$$

- and the optmization criterion

$$J(\beta) = \hat{L}_{logl} = \frac{1}{n}\sum_{i=1}^{n} -l(\beta; x^i, y^i) \tag{23}$$

Minimizing J is maximizing $l(\beta; \mathcal{D})$

Logistic regression

- loss function L_{logl}
- optimization criterion $J(\beta)$

Calculating the gradient

- $\frac{\partial l}{\partial f} = y_* - \frac{1}{1+e^f}$

  This is a scalar, and $\operatorname{sgn} \frac{\partial l}{\partial f} = y$
- We have also $\frac{\partial f(x)}{\partial \beta} = x$
- Now, the gradient of l w.r.t the parameter vector β is

$$\frac{\partial l}{\partial \beta} = \frac{\partial l}{\partial f}\frac{\partial f}{\partial \beta} = \left(y_* - \frac{1}{1+e^{-f(x)}}\right)x \tag{24}$$

Interpretation: The infinitezimal change of β to increase log-likelihood for a single data point is along the direction of x, with the sign of y Exercise Prove that (23) has a unique local optimum.

Calculating the gradient

- $\frac{\partial l}{\partial f} = y_* - \frac{1}{1+e^f}$
  This is a scalar, and $\operatorname{sgn} \frac{\partial l}{\partial f} = y$
- We have also $\frac{\partial f(x)}{\partial \beta} = x$
- Now, the gradient of l w.r.t the parameter vector β is

$$\frac{\partial l}{\partial \beta} = \frac{\partial l}{\partial f}\frac{\partial f}{\partial \beta} = \left(y_* - \frac{1}{1+e^{-f(x)}}\right) x \tag{24}$$

Interpretation: The infinitezimal change of β to increase log-likelihood for a single data point is along the direction of x, with the sign of y Exercise Prove that (23) has a unique local optimum.

Algorithm Steepest-Descent for Logistic Regression

Input $\beta^0 \in \mathbb{R}^d$ initial point
For $k = 0, 1, \ldots$

1. calculate $d^k = \frac{1}{n}\sum_{i=1}^n \left(y_*^i - \frac{1}{1+e^{-f(x^i)}}\right) x^i$
2. find η^k by line minimization
3. $\beta^{k+1} \rightarrow \beta^k - \eta^k d^k$

until stopping condition satisfied
Output β^{k+1}

Example: Backpropagation

- The **Backpropagation** algorithm is steepest descent for neural networks
- Consider a two layer neural network

$$f(x) = \sum_{j=1}^{m} \beta_j z_j = \sum_{j=1}^{m} \beta_j \phi(\sum_{k=1}^{n} w_{kj} x_k) \tag{25}$$

  The parameters are β and $W = [w_{kj}]_{j=1:m,k=1:n}$
- Let the loss be L_{LS} the Least Squares loss, $J(\beta, W) = \hat{L}_{LS}(\beta, W)$

Derivation of the gradient Exercise Derive this

$$\frac{\partial J}{\partial \beta_j} = \frac{1}{n} \sum_i \frac{\partial (y^i - f(x^i))^2}{\partial \beta_j} = \frac{1}{n} \sum_i 2(y^i - f(x^i)) z_j(x^i) \tag{26}$$

$$\frac{\partial J}{\partial w_{kj}} = \frac{1}{n} \sum_i \frac{\partial L_{LS}(y^i, f(x^i))}{\partial z_j(x^i)} \frac{\partial z_j(x^i)}{\partial w_{kj}} = \frac{1}{n} \sum_i \Big(2\beta_k (y^i - f(x^i))\Big) \underbrace{z_j(x^i)(1 - z_j(x^i))}_{\phi'} x_k^i$$

$$= \frac{\beta_k}{n} \sum_i x_k^i (y^i - f(x^i)) \nabla(\text{logistic regressor}) \tag{27}$$

In the above we used the identity $\phi' = \phi(1 - \phi)$ Exercise Prove it

Computational savings

- when $f(x^i)$ is computed, $z_j(x^i)$ are too; they should be "cached" and re-used
- the derivative of ϕ is easily obtained from the ϕ value
- Exercise The above gradient formulas can be easilty written in matrix-vector form

Backpropagation extends recursively to multi-layer networks. Exercise Derive it. Exercise Calculate the gradient for the 2 layer neural network with logistic output.

Practical properties of backpropagation

- Unlike in logistic regression, J has many local optima even for two layers and simple problems.
- Hence, initialization is important, and there are no general rules for a good initialization. Even if the neural network works well, we do not know if we are at the optimum.
- Saturation If $\tilde{z}_j = w_j^T x$ is large in magnitude, then $z_j = \phi(\tilde{z}_j)$ is near 0 or 1. In either case, $\phi'(\tilde{z}_j) = z_j(1 - z_j) \approx 0$. We say that that this sigmoid is **saturated**; z_j will be virtually insensitive to changes in w_j[2]
  To avoid saturation at the beginning of the training, one initializes W with "small" (w.r.t $\max ||x^i||$, random values. Exercise Why random and not exactly 0?
- To speed up training, it is useful to standardize the input data[3] $x^{1:N}$ as a preprocessing step. Exercise Note that theoretically shifting and rescaling the data should NOT have any effect.
- J can have **plateaus**, i.e. regions where $\nabla J \approx 0$ but that do not contain a local minimum. Exercise What can cause plateaus? Exercise And what is bad about them?

- In conclusion, training neural networks by backpropagation is an art: requires experience with the algorithm, careful tuning, repeated restarts, and a long time.

[2] or to changes in previous layers, if this is a multilayer network.

[3] Standardization should NOT include the dummy coordinate $x^0 \equiv 1$.

Training autoencoders

- The **autoencoder** is a neural network with one (or more) hidden layers, whose output y is identical with the input x.
- Let $x \in \mathbb{R}^d$, denote by $z \in \mathbb{R}^m$ the variables in the hidden layer, and by $\tilde{x} \in \mathbb{R}^d$ the output variables. Then,

$$x_{1:n} \xrightarrow{W} z_{1:m} \xrightarrow{\tilde{W}} \tilde{x}_{1:n}$$

$$z_j = \phi(w_j^T x), j = 1:m \qquad \tilde{x}_k = \phi(\tilde{w}_k^T z), k = 1:n \tag{28}$$

  where $W = [w_{ij}]_{i=1:n, j=1:m}$ and $\tilde{W} = [\tilde{w}_{kj}]_{j=1:n, k=1:m}$ are the parameters (or weights) to be learned.
  Note that this is a neural network with multiple outputs
- The "labels" are the inputs $x_{1:n}$, and the cost is the least squares cost.

$$L(x, \tilde{x}) = ||x - \tilde{x}||^2 = \sum_{i=1}^{n} (x_i - \tilde{x}_i)^2 \tag{29}$$

  If the variables x are binary, then the output layer will have a sigmoid, and the cost will be the logistic regression cost L_{logl}
- The training proceeds by backpropagation.

What is an autoencoder good for?

- Note that if $m \geq n$, we could set $W = \alpha I, \tilde{W} = \frac{1}{\alpha} I$; then z could be a (scaled) copy of x, and no training would be necessary to reproduce the input. Exercise Why do we need α at all?
  Thus, interesting autoencoders set $m < n$. If x can be reconstructed well from z, then we have succeeded to compress x, and we have learned in the process a set of descriptors, or a representation for x.
  If we want to have $m > n$, then we must use a sparsity inducing regularization (e.g L_1) to obtain an interesting representation.
- Autoencoders are the winning ingredient in Deep neural networks, and are a general method automatically find features for prediction.
  If the real problem is to predict another variable y from x, one can do as follows:
  1. Train an autoencoder for x, learn $W, \tilde{W}$.
  2. "remove" the top layer $\tilde{W}, \tilde{x}$, i.e discard all but W
  3. Construct a predictor $\hat{y} = f(z)$, with $z = \phi(Wx)$, where W are the autoencoder weights.

  Optional Do backpropagation to fine tune W.

  If f is linear or logistic, we obtain a two layer neural net $x \rightarrow z \rightarrow y$.
- The above can be applied recursively. Given $z^{1:N}$ the representations of the inputs $x^{1:N}$, one can now train an autoencoder $z_{1:m} \rightarrow u_{1:p} \rightarrow \tilde{z}_{1:m}$, perhaps with $p < m$.
  The interpretation is that $u_{1:p}$ are representing x at a higher level of abstraction than z.
- By using autoencoders, one can train multilayer neural networks, i.e. deep networks, while avoiding the plateaus that plague backpropagation. Hence "deep learning" is training with autoencoders.