Practical Session on Convex Optimization: Differentiable Optimization

Mark Schmidt

INRIA/ENS

September 2011

Motviation: Parameter Estimation with Different Models

- We have a binary classification problem.
- We want to try: logistic regression, probit regression, weighted logistic regression, SVMs, neural nets, kernel regression, extreme-value regression, etc.
- We might have a software package with most of these, but what if an important one is missing?

- We have a binary classification problem.
- We want to try: logistic regression, probit regression, weighted logistic regression, SVMs, neural nets, kernel regression, extreme-value regression, etc.
- We might have a software package with most of these, but what if an important one is missing?
- One option is to use a plug-and-play gradient method.
- Gradient-based methods are for continuous, local optimization where we can *evaluate the function and gradient*
- This lecture: we will implement simple methods of this type.
- Why? Illustrate basic concepts and fundamental methods that are building blocks for more advanced methods.

Set Up

- If using Matlab, please download the supporting material available here:
  `http://www.di.ens.fr/~mschmidt/MLSS/`

Set Up

- If using Matlab, please download the supporting material available here: `http://www.di.ens.fr/~mschmidt/MLSS/`
- You may also download the sido0 data set, or you can generate/load your own data set.

Preparing a Data Set

- To load this data in Matlab:

```
>> load('sido0_train.mat');
>> y = load('sido_train.targets');
```

- To generate a random data set:

```
>> X = randn(10000,5000);
>> w = randn(5000,1);
>> y = sign(X*w);
>> flips = rand(10000,1) > .9;
>> y(flips) = -y(flips);
```

ℓ_2-Regularized Logistic Regression

- We focus on ℓ_2-regularized logistic regression (but the code allows an arbitrary differentiable loss)
- Objective:

$$f(w) = \frac{\lambda}{2}||w||^2 + \sum_{i=1}^{n} \log(1 + exp(-y_i(w^T x_i))).$$

- Gradient:

$$f'(w) = \lambda w + \sum_{i=1}^{n} \frac{-y_i}{1 + exp(-y_i(w^T x_i))} x_i.$$

This is implemented in the function regLogistic.m.

Derivative-Checking

- Before doing anything, check your derivative code!

Derivative-Checking

- Before doing anything, check your derivative code!
- We will check the first 25 partial derivatives:

```
>> w = randn(25,1);
>> lambda = 1;
>> [f,g] = regLogistic(w,X(:,1:25),y,lambda);
>> [f,g2] = autoGrad(w,@regLogistic,X(:,1:25),y,lambda);
>> maxDiff = norm(g-g2,'inf')
```

A Basic Gradient Method

- The function findMin0.m implements a basic gradient method:
  1. Evaluate the gradient, $g := f'(w)$.
  2. Take the step, $w = w - \alpha g$.

A Basic Gradient Method

```
function [w,f] = findMin(funObj,w,maxEvals,alpha)

[f,g] = funObj(w);
funEvals = 1;

while 1
     w = w - alpha*g;
     [f,g] = funObj(w);
     funEvals = funEvals+1;

     optCond = norm(g,'inf');
     fprintf('\%6d \%15.5e \%15.5e \%15.5e\n',funEvals,alpha,f,optCond);

     if optCond < 1e-2
          break;
     end

     if funEvals >= maxEvals
          break;
     end
end
```

A Basic Gradient Method

- Running the method:

```
>> [nSamples,nVars] = size(X);
>> w = zeros(nVars,1);
>> lambda = 1;
>> funObj = @(w)regLogistic(w,X,y,lambda);
>> findMin0(funObj,w,250,1);
```

A Basic Gradient Method

Result with $\alpha = 1$:

2	1.00000e+00	2.90492e+09	6.33850e+03
3	1.00000e+00	9.12062e+08	1.26770e+04
4	1.00000e+00	1.17077e+10	1.26770e+04
5	1.00000e+00	9.12062e+08	1.26770e+04
6	1.00000e+00	1.17077e+10	1.26770e+04
7	1.00000e+00	9.12062e+08	1.26770e+04
8	1.00000e+00	1.17077e+10	1.26770e+04
9	1.00000e+00	9.12062e+08	1.26770e+04
10	1.00000e+00	1.17077e+10	1.26770e+04

Step size is too large.

A Basic Gradient Method

Result with $\alpha = 10^{-2}$:

2	1.00000e-02	4.36974e+06	5.10865e+02
3	1.00000e-02	3.69923e+06	5.05756e+02
4	1.00000e-02	3.04207e+06	5.00699e+02
5	1.00000e-02	2.39799e+06	4.95692e+02
6	1.00000e-02	1.76672e+06	4.90735e+02
7	1.00000e-02	1.14802e+06	4.85828e+02
8	1.00000e-02	5.59178e+05	3.74992e+02
9	1.00000e-02	6.03338e+05	7.88664e+02
10	1.00000e-02	1.00608e+06	4.84135e+02
...			
250	1.00000e-02	5.90354e+05	4.60089e+02

Actual progress with this step size, but not monotonic.

A Basic Gradient Method

Result with $\alpha = 10^{-5}$:

2	1.00000e-05	4.12469e+03	4.48073e+02
3	1.00000e-05	3.54435e+03	4.44394e+02
4	1.00000e-05	2.97095e+03	4.36504e+02
5	1.00000e-05	2.41459e+03	4.18184e+02
6	1.00000e-05	1.90241e+03	3.75860e+02
7	1.00000e-05	1.49796e+03	2.93738e+02
8	1.00000e-05	1.28333e+03	1.63623e+02
9	1.00000e-05	1.22771e+03	8.97186e+01
10	1.00000e-05	1.21347e+03	1.14682e+02
...			
250	1.00000e-05	8.34321e+02	2.28737e+01

This step size yields the best progress.

A Basic Gradient Method

Result with $\alpha = 10^{-7}$:

2	1.00000e-07	8.30078e+03	5.61680e+03
3	1.00000e-07	7.85817e+03	5.36067e+03
4	1.00000e-07	7.45565e+03	5.11815e+03
5	1.00000e-07	7.08930e+03	4.88905e+03
6	1.00000e-07	6.75553e+03	4.67301e+03
7	1.00000e-07	6.45106e+03	4.46955e+03
8	1.00000e-07	6.17292e+03	4.27812e+03
9	1.00000e-07	5.91844e+03	4.09810e+03
10	1.00000e-07	5.68525e+03	3.92885e+03
...			
250	1.00000e-07	1.72792e+03	3.97410e+02

This step size seems too small to make significant proress.

Armijo Backtracking Line-Search

- We don't want to tune the step size for every new problem.
- This is why we use a line search.

Armijo Backtracking Line-Search

- We don't want to tune the step size for every new problem.
- This is why we use a line search.
- A basic backtracking search:
  1. Start with a large value of α.
  2. Divided α in half if we don't satisfy the Armijo condition:

$$f(w - \alpha g) \leq f(w) - \gamma\alpha||g||^2.$$

Armijo Backtracking Line-Search

Basic backtracking line search:

```
wp = w - alpha*g;
[fp,gp] = funObj(wp);
funEvals = funEvals+1;

while fp > f - gamma*alpha*g'*g
    alpha = alpha/2;
    wp = w - alpha*g;
    [fp,gp] = funObj(wp);
    funEvals = funEvals+1;
end

w = wp;
f = fp;
g = gp;
```

Armijo Backtracking Line-Search

Result with $\alpha_0 = 1$ and $\gamma = 10^{-4}$:

18	1.52588e-05	6.28797e+03	4.52010e+02
19	1.52588e-05	5.39309e+03	4.51876e+02
20	1.52588e-05	4.49865e+03	4.51459e+02
21	1.52588e-05	3.60605e+03	4.49767e+02
22	1.52588e-05	2.72306e+03	4.41135e+02
23	1.52588e-05	1.89570e+03	3.93456e+02
24	1.52588e-05	1.32969e+03	2.14599e+02
25	1.52588e-05	1.22179e+03	8.68655e+01
26	1.52588e-05	1.20389e+03	8.76510e+01
...			
250	1.52588e-05	7.79165e+02	1.98571e+01

In this case, backtracking gives better performance than fixed α.

Armijo Backtracking Line-Search

- A danger with the simple backtracking is that α_k *may become too small to make substantial progress*.
- We can reset α_k on each iteration:

```
if funEvals > 1
     alpha = 1;
end
```

Armijo Backtracking Line-Search

Result with resetting to $\alpha = 1$:

18	1.52588e-05	6.28797e+03	4.52010e+02
32	1.22070e-04	3.15106e+03	1.08930e+03
49	1.52588e-05	1.38857e+03	2.74485e+02
65	3.05176e-05	1.36762e+03	3.92855e+02
82	1.52588e-05	1.19098e+03	1.63615e+02
99	1.52588e-05	1.14223e+03	7.11060e+01
113	1.22070e-04	1.10533e+03	1.46868e+02
130	1.52588e-05	1.06385e+03	5.81590e+01
144	1.22070e-04	1.04816e+03	1.20607e+02
...			
254	1.52588e-05	9.57028e+02	3.52162e+01

Each iteration *makes more progress*, but requires more evaluations.

Hermite Polynomial Interpolation

- Step size halving ignores information collected during the line-search.
- We can reduce the number of evaluations per iteration using Hermite polynomial interpolation.
- E.g., we can minimize the quadratic passing through $f(w)$, $f'(w)$, and $f(w - \alpha g)$:

Hermite Polynomial Interpolation

- Step size halving ignores information collected during the line-search.
- We can reduce the number of evaluations per iteration using Hermite polynomial interpolation.
- E.g., we can minimize the quadratic passing through $f(w)$, $f'(w)$, and $f(w - \alpha g)$:

```
alpha = alpha^2*g'*g/(2*(fp + g'*g*alpha - f));
```

Hermite Polynomial Interpolation

Result with resetting to $\alpha = 1$ quadratic interpolation:

16	1.22706e-05	5.05711e+03	4.51353e+02
20	3.45637e-05	3.03913e+03	4.43463e+02
24	3.24136e-05	1.37769e+03	2.43325e+02
28	1.08171e-05	1.22260e+03	1.33995e+02
32	1.47454e-05	1.19365e+03	1.08309e+02
38	1.01415e-04	1.14996e+03	1.52492e+02
45	5.18193e-05	1.09565e+03	1.36125e+02
51	1.65099e-05	1.08090e+03	7.84479e+01
54	3.26453e-05	1.07401e+03	1.14560e+02
...			
251	1.23991e-05	7.87318e+02	2.33304e+01

Significantly reduces the number of evaluations per iteration.

Hermite Polynomial Interpolation

- Setting $\alpha_k = 1$ is typically too large.

Hermite Polynomial Interpolation

- Setting $\alpha_k = 1$ is typically too large.
- On the first iteration, we can use some heuristic like:

```
alpha = 1/||g||
```

Hermite Polynomial Interpolation

- Setting $\alpha_k = 1$ is typically too large.
- On the first iteration, we can use some heuristic like:

```
alpha = 1/||g||
```

- On subsequent iterations, we can initialize α_k with polynomial interpolation:

```
alpha = min(1,2*(f_old-f)/(g'*g));
```

Hermite Polynomial Interpolation

Result with quadratic initialization and quadratic interpolation:

2	1.41623e-05	5.83619e+03	4.51905e+02
3	1.00669e-04	1.65409e+03	5.95919e+02
6	1.22437e-05	1.22162e+03	1.58192e+02
8	1.33724e-05	1.18060e+03	7.74888e+01
9	8.93858e-05	1.11182e+03	8.70173e+01
11	2.88742e-05	1.09897e+03	9.47907e+01
12	2.54553e-05	1.08877e+03	9.32675e+01
13	1.88667e-05	1.07807e+03	7.66592e+01
14	3.75443e-05	1.06820e+03	9.11023e+01
...			
250	9.74511e-06	6.20440e+02	2.09519e+01

There is now *very little backtracking*.

Hermite Polynomial Interpolation

Practical implementations take these ideas further:

- Interpolation based on cubic or higher-order interpolation.
- Line-search based on Wolfe conditions.

Nesterov Extrapolation

- One way to enhance the performance of gradient methods is with Nesterov's exploration step between gradient updates.
- Also known as accelerated or optimal gradient methods.
- Extrapolation set to achieve an optimal convergence rate for convex optimization.

Nesterov Extrapolation

- One way to enhance the performance of gradient methods is with Nesterov's exploration step between gradient updates.
- Also known as accelerated or optimal gradient methods.
- Extrapolation set to achieve an optimal convergence rate for convex optimization.
- Based on the simple recursion:

$$w_{k+1} = y_k - \alpha f'(y_k),$$

$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2}$$

$$y_{k+1} = w_k + \frac{t_k - 1}{t_{k+1}}(w_{k+1} - w_k).$$

Nesterov Extrapolation

```
t = 1;
y = w;
while 1
    if funEvals > 1
         tp = (1 + sqrt(1+4*t^2))/2;
         y = w + ((t-1)/tp)*(w-w_old);
         t = tp;
         [f,g] = funObj(y);
         funEvals = funEvals+1;
    end
    w_old = w;

    wp = y - alpha*g;
    [fp,gp] = funObj(wp);
    funEvals = funEvals+1;

    while fp > f - gamma*alpha*g'*g
         alpha = alpha^2*g'*g/(2*(fp + g'*g*alpha - f));
         wp = y - alpha*g;
         [fp,gp] = funObj(wp);
         funEvals = funEvals+1;
    end

    w = wp;
```

Nesterov Extrapolation

Result with Nesterov's extrapolation scheme:

2	1.41623e-05	8.78772e+03	5.88650e+03
4	1.41623e-05	5.83619e+03	4.51905e+02
6	1.41623e-05	4.77196e+03	4.51507e+02
8	1.41623e-05	3.48256e+03	4.48841e+02
10	1.41623e-05	2.01470e+03	4.06706e+02
12	1.41623e-05	1.28744e+03	2.67356e+02
14	1.41623e-05	1.19696e+03	1.66685e+02
16	1.41623e-05	1.14429e+03	7.45916e+01
18	1.41623e-05	1.11691e+03	8.06874e+01
...			
250	1.41623e-05	3.20896e+02	5.60646e+00

Final function value is nearly cut in half, despite two evaluations per iteration (various tricks can make this work better).

Newton's Method

- The other classical differentiable optimization method is Newton's method.

Newton's Method

- The other classical differentiable optimization method is Newton's method.
- Uses the update

$$w_{k+1} = w_k - \alpha_k d_k,$$

where d_k is a solution to the system

$$f''(w_k)d_k = -f'(w_k).$$

Newton's Method

- The other classical differentiable optimization method is Newton's method.
- Uses the update
$$w_{k+1} = w_k - \alpha_k d_k,$$
where d_k is a solution to the system
$$f''(w_k)d_k = -f'(w_k).$$
- We modify the Armijo condition to
$$f(w_{k+1}) \leq f(w_k) + \gamma\alpha_k f'(w_k)^T d_k.$$
- Has a natural step length of $\alpha_k = 1$.

Newton's Method

- The other classical differentiable optimization method is Newton's method.
- Uses the update

$$w_{k+1} = w_k - \alpha_k d_k,$$

where d_k is a solution to the system

$$f''(w_k)d_k = -f'(w_k).$$

- We modify the Armijo condition to

$$f(w_{k+1}) \leq f(w_k) + \gamma\alpha_k f'(w_k)^T d_k.$$

- Has a natural step length of $\alpha_k = 1$.
- Simple implementation in findMinNewton.m (do not run this).

Newton's Method

Running findMinNewton.m:

2	1.00000e+00	1.91912e+03	1.40881e+03
3	1.00000e+00	7.96491e+02	5.06968e+02
4	1.00000e+00	3.85229e+02	1.92354e+02
5	1.00000e+00	2.20943e+02	7.45484e+01
6	1.00000e+00	1.51860e+02	2.96743e+01
7	1.00000e+00	1.22474e+02	1.20043e+01
8	1.00000e+00	1.11158e+02	4.60128e+00
9	1.00000e+00	1.08039e+02	1.38955e+00
10	1.00000e+00	1.07643e+02	2.17531e-01

Our previous methods are *still far from the solution*.

Newton's Method

- Newton's method often converges very fast, but with very expensive iterations.
- Typically, you need to modify the Hessian to be positive-definite.

Newton's Method

- Newton's method often converges very fast, but with very expensive iterations.
- Typically, you need to modify the Hessian to be positive-definite.
- Is it possible to get fast convergence like Newton's method, without the cost?

Diagonally-Scaled Steepest Descent

- First Newton approximation: use the diagonal of $f''(w_k)$.
- Use regLogisticDiag.m and in findMinNewton.m use:

```
d = g./H;
```

Diagonally-Scaled Steepest Descent

Diagonally-Scaled Steepest Descent:

8	1.02548e-02	5.08497e+03	4.46796e+02
12	8.41245e-05	2.40090e+03	3.93547e+02
16	3.20810e-04	1.44074e+03	2.28581e+02
19	8.09445e-04	1.24985e+03	1.11657e+02
22	2.36797e-03	1.17424e+03	1.79855e+02
26	9.45584e-03	1.14728e+03	1.83633e+02
29	7.95545e-04	1.05724e+03	9.75465e+01
32	1.79066e-03	1.02717e+03	8.32227e+01
36	1.97395e-02	9.74572e+02	1.40304e+02
...			
251	2.35414e-03	1.45367e+02	1.28719e+01

Substantially more progress than gradient methods, but more expensive iterations and natural step length typically not accepted.

Barzilai-Borwein

- Second Newton approximation:
  - Use the Barzilai-Borwein step length in a gradient method.
- Use the following search direction in findMinScaled.m:

```
if funEvals > 1
     g_diff = g-g_old;
     alpha = -alpha*(g_old'*g_diff)/(g_diff'*g_diff);
end
```

Barzilai-Borwein

The Barzilai-Borwein method:

2	1.41623e-05	5.83619e+03	4.51905e+02
3	1.30265e-05	5.07243e+03	4.51675e+02
6	3.77621e-05	2.86695e+03	4.43147e+02
9	2.57551e-05	1.51890e+03	3.12312e+02
11	1.39417e-05	1.23060e+03	1.36640e+02
12	1.89086e-05	1.20056e+03	1.14600e+02
13	1.21666e-05	1.18450e+03	8.24373e+01
14	1.18475e-05	1.17338e+03	7.81654e+01
15	6.85312e-05	1.11907e+03	6.82128e+01
...			
251	3.16440e-04	1.08771e+02	3.52277e-01

Performance is typically improved using the non-monotonic Armijo condition.

Non-Linear Conjugate Gradient

- Third Newton approximation: non-linear conjugate gradient.
- Use the following search direction in findMinScaled.m:

```
if funEvals > 1
     alpha = min(1,2*(f_old-f)/(g'*g));
     beta = (g'*g)/(g_old'*g_old);
     d = g + beta*d;
else
     d = g;
end
```

Non-Linear Conjugate Gradient:

2	1.41623e-05	5.83619e+03	4.51905e+02
3	1.00669e-04	1.28081e+03	2.21334e+02
6	4.03711e-05	1.24034e+03	1.88791e+02
7	8.70114e-06	1.19428e+03	1.58569e+02
8	1.70587e-05	1.15260e+03	1.47617e+02
9	1.96374e-05	1.13544e+03	1.79580e+02
10	4.62645e-06	1.10123e+03	1.35990e+02
11	1.88296e-05	1.08414e+03	1.37598e+02
13	-4.61296e-06	1.08268e+03	1.24459e+02
...			
251	1.56501e-04	1.08880e+02	4.33766e-01

Note that d is not necessarily a descent direction and typically you need to implement a check for this. Performance is improved by *preconditioning* and/or using a *more accurate line-search*.

Quasi-Newton Methods

- Fourth Newton approximation: quasi-Newton methods.
- The L-BFGS approximation:

```
if funEvals > 1
     g_diff = g-g_old;
     [S,Y] = lbfgsUpdate(S,Y,-alpha*d,g_diff,memory);
     H0 = -alpha*(d'*g_diff)/(g_diff'*g_diff);

     d = lbfgs(g,S,Y,H0);
     alpha = 1;
else
     S = zeros(length(w),0);
     Y = zeros(length(w),0);
     H0 = 1;
     d = g;
end
```

- Typically, you also need update skipping/damping to preserve positive-definiteness of the approximation.

Quasi-Newton Methods:

```
  2      1.41623e-05      5.83619e+03      4.51905e+02
  3      1.00000e+00      5.11759e+03      4.51692e+02
  6      1.52249e-03      1.45874e+03      4.45557e+02
  8      3.46536e-01      1.24791e+03      1.77168e+02
  9      1.00000e+00      1.20050e+03      9.55587e+01
 11      3.13685e-01      1.18137e+03      9.50644e+01
 12      1.00000e+00      1.14498e+03      7.55459e+01
 13      1.00000e+00      1.07778e+03      1.48323e+02
...
 76      1.00000e+00      1.07632e+02      4.80271e-03
```

Solution found.

Hessian-Free Newton

- Fifth Newton approximation: Hessian-Free Newton.

```
cgMaxIter = min(maxEvals-funEvals);
cgForce = min(0.5,sqrt(norm(g)))*norm(g);
HvFunc = @(v)autoHv(v,w,g,funObj);

[d,cgIters,cgRes] = conjGrad(HvFunc,g,cgForce,cgMaxIter);
funEvals = funEvals + cgIters;
alpha = 1;
```

Quasi-Newton Methods:

3	1.00000e+00	2.81156e+03	1.38607e+03
5	1.00000e+00	2.06682e+03	5.85128e+02
8	1.00000e+00	1.76603e+03	8.21305e+02
10	1.00000e+00	1.19297e+03	1.52940e+02
14	1.00000e+00	9.70365e+02	8.09050e+01
17	1.00000e+00	9.20963e+02	2.69460e+01
30	1.00000e+00	4.74937e+02	2.61594e+02
32	1.00000e+00	3.61888e+02	7.77862e+01
35	1.00000e+00	2.92318e+02	4.26920e+01
...			
141	1.00000e+00	1.07633e+02	9.65971e-03

Performance is substantially improved by preconditioning (i.e. diagonal or use L-BFGS).

Extensions:

- Use another data set.
- Use another loss function (i.e. smooth SVMs).
- More accurate linesearch (cubic interpolation, Wolfe line-search).
- Variants of Nesterov's method.
- Non-monotonic Armijo condition.
- Check that non-linear CG gives a descent direction.
- Update skipping/damping in L-BFGS.
- Preconditioning in Hessian-free Newton.

References

Most of this lecture is based on material from Nocedal and Wright's very good "Numerical Optimization" book.