Gradient Descent

Objective Function

Some differentiable function $f : \mathbb{R}^n \to \mathbb{R}$.

Gradient Descent

```
Start with some $\mathbf{x}_0$, $i = 0$ and learning rate $\lambda$
repeat
  $\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \nabla f(\mathbf{x}_i)$
until $\|\nabla f(\mathbf{x}_{i+1})\| \leq \epsilon$
```

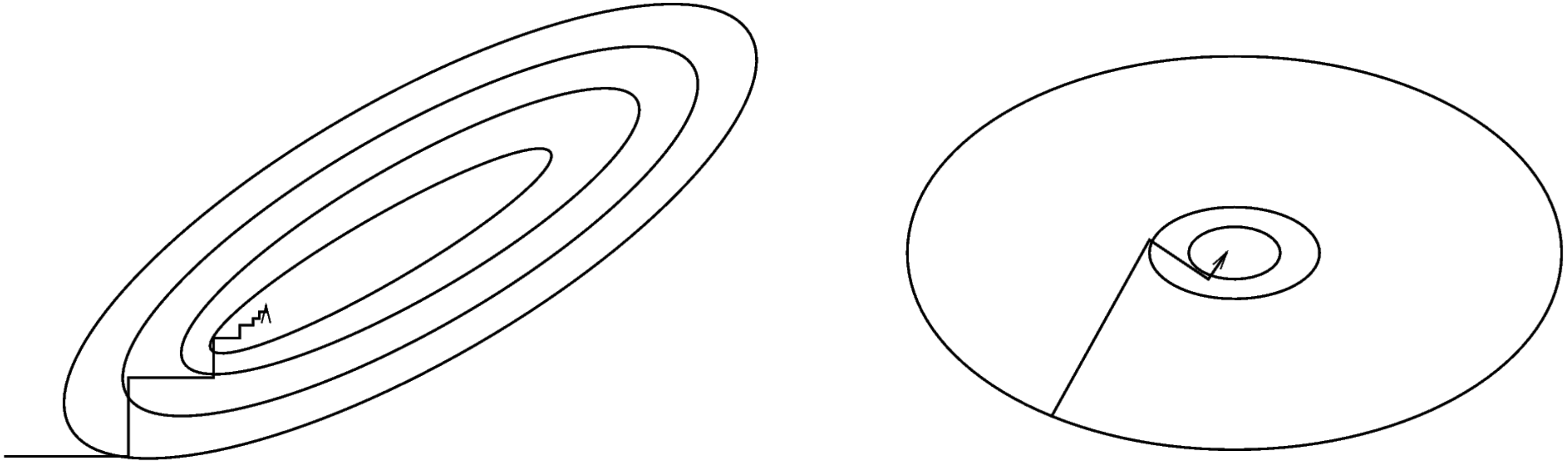
Line Search Variant

Replace $\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \nabla f(\mathbf{x}_i)$ by

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \hat{\lambda} \nabla f(\mathbf{x}_i) \text{ where } \hat{\lambda} = \underset{\lambda}{\operatorname{argmin}} f(\mathbf{x}_i - \lambda \nabla f(\mathbf{x}_i))$$

This ensures that we walk downhill. For fixed λ not even this may be the case.

Problems with Gradient Descent



Left: Gradient descent takes a long time to converge since the landscape of values of f forms a long and narrow valley, causing the algorithm to zig-zag along the walls of the valley.

Right: due to the homogeneous structure of the minimum the algorithm converges after very few iterations. Note that in both cases the next direction of descent is *orthogonal* to the previous one, since line search provides the optimal step length.

Stochastic Gradient Descent

Basic Idea

Sometimes the gradient is not reliable and we have only noisy estimates. Here we perform a descent step in the direction of the approximation of the gradient. Under many conditions this will still converge.

Trick

If we have a function $f : \mathfrak{X} \to \mathbb{R}$ made up of many individual terms $f_i : \mathfrak{X} \to \mathbb{R}$ via $f(x) = \frac{1}{m}\sum_{i=1}^{m} f_i(x)$ we may randomly select one f_j at a time and perform gradient with respect to f_i. This leads to

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \nabla f_j(\mathbf{x})$$

Advantage

This is much cheaper to compute than ∇f, in particular, if all f_i are somewhat similar. Later, the f_j will be the the performance of our estimator on individual observations.

Distorting the Space

Idea

We focus on the special case of convex quadratic functions. How easy it is to minimize

$$f(x) = \mathbf{c}^\top \mathbf{x} + \frac{1}{2}\mathbf{x}^\top K \mathbf{x}$$

via gradient descent depends on K. If K is diagonal, we get to the minimum in at most m steps.

Distort the metric such that $\|\cdot\|_K$ behaves as if K was diagonal.

K-Orthogonal Directions

Given a symmetric matrix $K \in \mathbb{R}^{m \times m}$, any two vectors $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^m$ are called K-orthogonal if $\langle \mathbf{x}, \mathbf{x}' \rangle_K = \mathbf{x}^\top K \mathbf{x}' = 0$.

Minimizer

Minimum at $\nabla f(\mathbf{x}) = \mathbf{c} + K\mathbf{x} = 0$.

Conjugate Gradient Descent

Basic Idea

In order to solve $\mathbf{c} + K\mathbf{x} = 0$ project $\mathbf{x}$ onto K and use orthogonal decomposition.

Decomposition Theorem

Denote by $\mathbf{v}_1, \ldots, \mathbf{v}_m$ a set of mutually K-orthogonal vectors for a strictly positive definite matrix $K \in \mathbb{R}^{m \times m}$. Then the following properties hold:

(i) The vectors $\mathbf{v}_1, \ldots, \mathbf{v}_m$ form a basis.

(ii) Any $\mathbf{x} \in \mathbb{R}^m$ can be expanded in terms of $\mathbf{v}_i$ by

$$\mathbf{x} = \sum_{i=1}^{m} \mathbf{v}_i \frac{\langle \mathbf{v}_i, \mathbf{x} \rangle_K}{\langle \mathbf{v}_i, \mathbf{v}_i \rangle_K} = \sum_{i=1}^{m} \mathbf{v}_i \frac{\mathbf{v}_i^\top K \mathbf{x}}{\mathbf{v}_i^\top K \mathbf{v}_i}.$$

In particular, for any $-\mathbf{c} = K\mathbf{x}$ we can find $\mathbf{x}$ by

$$\mathbf{x} = -\sum_{i=1}^{m} \mathbf{v}_i \frac{\mathbf{v}_i^\top \mathbf{c}}{\mathbf{v}_i^\top K \mathbf{v}_i}.$$

Conjugate Gradient Descent Algotihm

Generating Descent Directions

We use a Taylor expansion $f(\mathbf{x}_0 + \mathbf{w}) = f(x_o) + \nabla f(\mathbf{x})^\top \mathbf{w} + \frac{1}{2}\mathbf{w}^\top K \mathbf{w}$ where $K = f''(\mathbf{x})$. Hence $\mathbf{w}$ is given by $\nabla f(\mathbf{x}_0 + \mathbf{w}) = \nabla f(\mathbf{x}_0) + K\mathbf{w} = 0$. Apply the decomposition trick of the previous page to $\nabla f(\mathbf{x}_0)$.

```
Require: $\mathbf{x}_0$ and Set $i = 0$ and $\mathbf{v}_0 = \mathbf{g}_0 = f'(\mathbf{x}_0)$
  repeat
    $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{v}_i$ where $\alpha_i = -\frac{\mathbf{g}_i^\top \mathbf{v}_i}{\mathbf{v}_i^\top K \mathbf{v}_i}$ (project onto descent direction $\mathbf{v}_i$)
    $\mathbf{g}_{i+1} = f'(\mathbf{x}_{i+1})$ (get new descent direction)
    $\mathbf{v}_{i+1} = -\mathbf{g}_{i+1} + \beta_i \mathbf{v}_i$ where $\beta_i = \frac{\mathbf{g}_{i+1}^\top K \mathbf{v}_i}{\mathbf{v}_i^\top K \mathbf{v}_i}$. (make $\mathbf{v}_{i+1}$ orthogonal to $\mathbf{v}_i$).
    $i = i + 1$
  until $\mathbf{g}_i = 0$
Output: $\mathbf{x}_i$
```

This works since $\mathbf{g}_i$ contains the part of the gradient $\nabla f(\mathbf{x}_0)$ **not** spanned by $\mathbf{v}_0, \ldots, \mathbf{v}_{i-1}$.

Why Computers are Different

Accumulation of Errors

- Obviously, the following series will converge to ∞: $x_0 = 0$ and $x_{i+1} = x_i + 1$.
- On a computer (in MATLAB) this series will converge to $4.5036 \cdot 10^{15}$.

  Operations smaller than machine precision are ignored! This is a problem of the mantissa (number of digits) of the representation.

Overflow and Underflow

- We can only store numbers up to a limit, determined by the exponent of the representation. In MATLAB this is $1.7977 \cdot 10^{308}$ and $2.2251 \cdot 10^{-308}$.

Rule of Thumb

- Do not mix additions and subtractions if possible. Combine all additions and all subtractions separately first. Use a tree-based scheme.
- Balance off multiplications and divisions to avoid over- or underflow.

More Rules of Thumb

Function Calls: Invoke a function $f(x)$

Call by Value

Passing data to a function is only a good thing if we are talking about one or two numbers. Otherwise avoid.

Cannot be avoided in MATLAB. It tries to be clever, though.

Call by Reference

Passing a pointer to data to a function is much cheaper. However, it costs an indirection to look up. Use for large amounts of data.

Cannot be done in MATLAB. Best workaround (ugly) via global variables.

Divisions and Multiplications

Multiplications are cheaper. Hence, pool divisions if possible and use them once.

The Cache and Missing It

Many Levels of Memory

- Registers: very few on CPU (less than 100), extremely fast access (1ns)
- Level 1 Cache: typically around 64kB, very fast access (2ns)
- Level 2 Cache: typically around 512KB, fast access (5ns)
- Main Memory: often around 256MB, reasonably fast access (10ns)
- Hard Disk (swap space): in the order of 20GB, slow access (10ms)

Fast and Slow Index

When computing the sum $\sum_{i,j} M_{ij}$ it makes a big difference whether we use

$$\sum_{i=1}^{m}\sum_{j=1}^{n} M_{ij} \text{ or } \sum_{j=1}^{n}\sum_{i=1}^{m} M_{ij}$$

What's Happening

In once case, we get continuous cache misses and have to wait for the Main Memory (like a 100MHz CPU), in the other case we can use the $L1/L2$ cache an get performance closer to $500MHz$.

Do It NOT Yourself

Consequence

If possible, avoid coding up matrix operations yourself. Careful cache tuning needed. Use ready-made libraries instead.

LINPACK/EISPACK

LINear PACKage: old and reliable. Not very cache optimized. MATLAB until version 5.x runs on it.

BLAS

Basic Linear Algebra Subroutines: new standard. You can get optimized implementations from Sun, Intel, Compaq, ... Aberdeen and Baxter won the Gordon-Bell prize by coding up one.

LAPACK

Linear Algebra Package: runs on top of BLAS, more sophisticated. Usually, you can get BLAS/LAPACK in a self optimizing version, called ATLAS.

Inverting Matrices

An Innocuous Example

$M = O^\top \Lambda O$ with some small and some reasonably large λ_i. Then $M^{-1}\mathbf{x}$ may change a lot depending on small changes in $\mathbf{x}$.

Why Things go Wrong

Assume $\mathbf{x}$ corresponds to the smallest eigenvalue $\lambda_{\min}$ of M. Then $M^{-1}\mathbf{x} = \lambda_{\min}^{-1}\mathbf{x}$. This can be huge.

Matrix Inversion in Eigensystem Notation

Denote by $\mathbf{v}_i, \lambda_i$ the eigensystem of M. Then for $\mathbf{x} = \sum_i \alpha_i \mathbf{v}_i$ we have

$$M^{-1}\mathbf{x} = M^{-1} \sum_{i=1}^{m} \alpha_i \mathbf{v}_i = \sum_{i=1}^{m} \frac{\alpha_i}{\lambda_i} \mathbf{v}_i$$

Small changes in α_i may translate into big changes in $M^{-1}\mathbf{x}$.

Condition of a Matrix

Definition The condition of a matrix is given by

$$\operatorname{cond}(M) = \|M\| \|M^{-1}\| = \left|\frac{\lambda_{\max}}{\lambda_{\min}}\right|$$

The smaller $\operatorname{cond}(M)$, the better behaved M is.

Regularization

Quite often we do not know $\mathbf{x}$ exactly in $M^{-1}\mathbf{x}$ anyway. Can we find a $\tilde{M}$ such that M^{-1} is well behaved?

Trick: add a small term of size ε on the main diagonal. This idea will come back later ...

Doing It Again (now the condition is $\frac{\lambda_{\max} + \varepsilon}{\lambda_{\min} + \varepsilon}$)

$$(M + \varepsilon\mathbf{1})^{-1}\mathbf{x} = (M + \varepsilon\mathbf{1})^{-1} \sum_{i=1}^{m} \alpha_i \mathbf{v}_i = \sum_{i=1}^{m} \frac{\alpha_i}{\lambda_i + \varepsilon} \mathbf{v}_i$$

Small changes in α_i may translate into bounded changes in $(M + \varepsilon\mathbf{1})^{-1}\mathbf{x}$.

Triangular Matrices

THE AUSTRALIAN NATIONAL UNIVERSITY

Definition

A matrix $T \in \mathbb{R}^{m \times m}$ where $T_{ij} \neq 0$ only if $j \geq i$ (upper triangular matrix).

$$T = \begin{bmatrix} T_{11} & \dots & T_{1m} \\ 0 & T_{ii} & T_{im} \\ 0 & 0 & T_{mm} \end{bmatrix}$$

Elimination Method

To solve $T\mathbf{x} = \mathbf{y}$ we use an iterative method.

- For m we have $T_{mm}x_m = y_m$ and thus $x_m = T_{mm}^{-1}y_m$
- For $i < m$ we have $T_{ii}x_i + \sum_{j=1}^{m} T_{ij}x_j = y_i$ which allows us to solve for x_i.
- Each x_i costs us roughly $(m - i)$ operations. This yields a total of $O(m^2)$ operations to invert a triangular matrix
- For lower triangular matrices we start from 1 rather than m.

Factorization Methods

LU Decomposition

We decompose $M \in \mathbb{R}^{m \times m}$ into $M = LU$ where L is a lower triangular matrix and U is an upper triangular one.

Cholesky Decomposition

We decompose M into $L^\top L$ where L is a lower triangular matrix. This works only for positive definite matrices. Method of choice.

Bunch-Kaufmann Decomposition

Modification which of the Cholesky decomposition which works also for indefinite matrices (positive and negative eigenvalues). Part of LAPACK.

SVD

Decomposition of M into $M = U \Lambda O$ where U and O are orthogonal matrices. This also allows for fast inversion via $U^\top U = O O^\top = \mathbf{1}$. Expensive to compute, though.

Rule of Thumb: Use Numerical Recipes

Cholesky Decomposition

Factorization for $M \in \mathbb{R}^{1\times 1}$

$M = L^\top L$ and hence $L_{11} = \sqrt{M_{11}}$

Iteration Step

To go from $M \in \mathbb{R}^{m\times m}$ with $M = LL^\top$ to $M \in \mathbb{R}^{(m+1)\times(m+1)}$ we posit

$$\begin{bmatrix} M & \mathbf{m}^\top \\ \mathbf{m} & \mu \end{bmatrix} = \begin{bmatrix} L & 0 \\ \mathbf{l} & \lambda \end{bmatrix} \begin{bmatrix} L^\top & \mathbf{l}^\top \\ 0 & \lambda \end{bmatrix}$$

Writing out the equations this means

- $M = LL^\top$ (satisfied by induction assumption)
- $\mathbf{m}^\top = L\mathbf{l}^\top$ (satisfied by solving for $\mathbf{l}^\top = L^{-1}\mathbf{m}^\top$).

$$m_j = \sum_{k=1}^{j} L_{jk} l_k \text{ and thus } l_j = \frac{1}{L_{jj}}\left(m_j - \sum_{k=1}^{j-1} L_{jk} l_k\right) \text{ for all } j < m$$

- $\mu = \mathbf{l}^\top \mathbf{l} + \lambda^2$ (satisfied by solving $\lambda = \sqrt{\mu - \mathbf{l}^\top \mathbf{l}}$)

Cholesky Decomposition

```
for $i = 1$ until $m$ do
  for $j = i$ until $m$ do
    $\mathrm{tmp} = M_{ij}$
    for $k = i - 1$ step $-1$ until $1$ do
      $\mathrm{tmp} -= M_{jk} M_{ik}$
    end for
    if $i = j$ then
      $p_i = \frac{1}{\sqrt{\mathrm{tmp}}}$ (the $(L_{ii})^{-1}$ entry, fail if $\mathrm{tmp} \leq 0$)
    else
      $M_{ji} = \mathrm{tmp} \cdot p_i$
    end if
  end for
end for
```

Cholesky Decomposition

In Place Factorization

The Cholesky decomposition uses only the upper diagonal matrix, and writes its results into the lower diagonal entries.

Pivoting

Sometimes we will find some small μ for which subsequently λ is very small. This causes error propagation. We avoid this by choosing the next large μ instead.

Incomplete Factorizations

We stop if we cannot find any $\lambda \geq 0$ any more.

More Info

Numerical Recipes (www.nr.com), Luenberger (Introduction to Linear and Nonlinear Programming), Mangasarian (Nonlinear Programming)