Today

- Gradient descent for minimization of functions of real variables.
- Multi-dimensional scaling
- Self-organizing maps

Gradient Descent

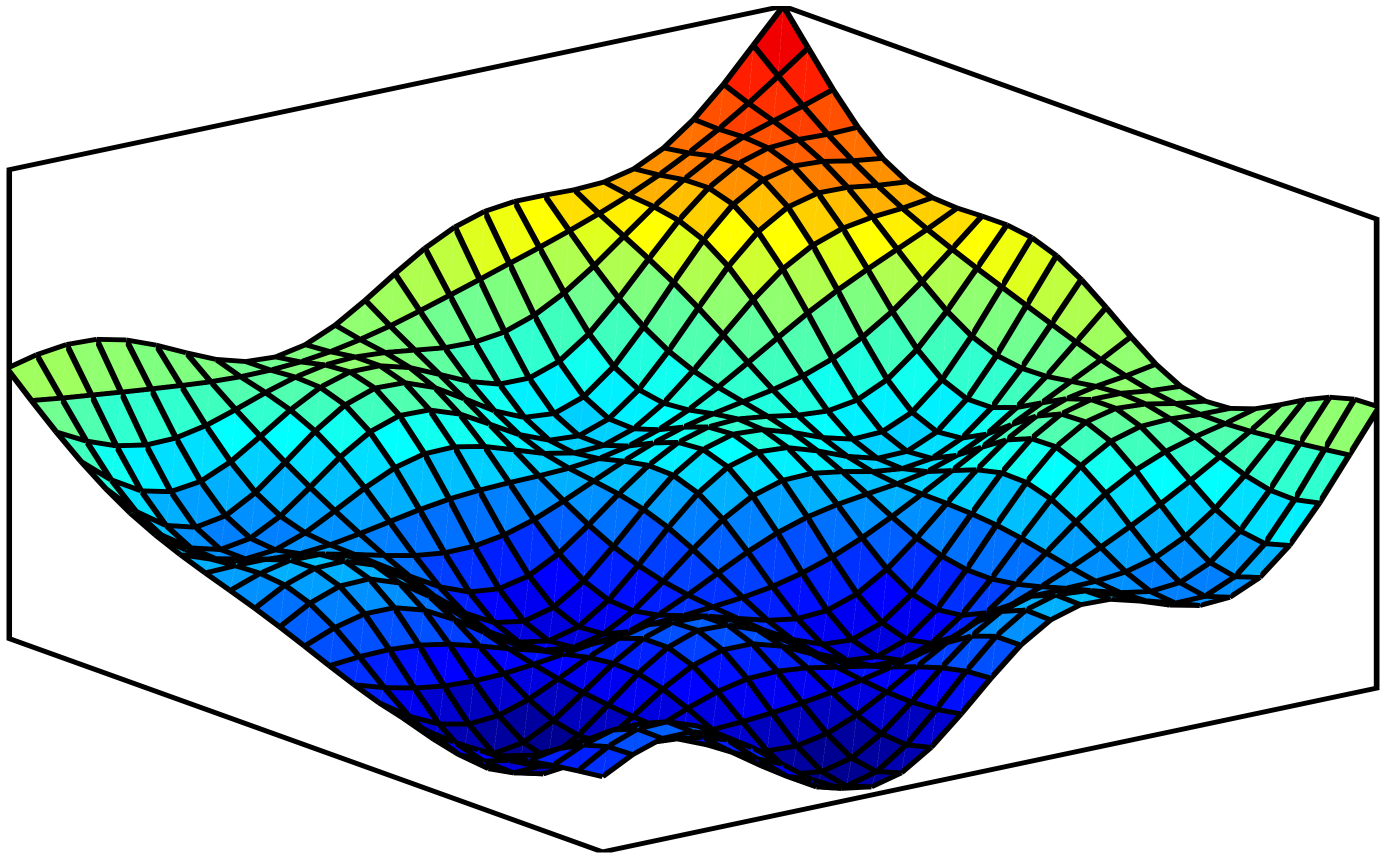
Derivatives

- Consider function $f(x) : \Re \mapsto \Re$.
- The derivative w.r.t. x is another function $f'(x)$ or $\frac{d}{dx} f(x) : \Re \mapsto \Re$, giving the slope of the tangent to f.
- If f' is continuous, then f is *continuously differentiable*, sometimes written $f \in C^1$.
- If $f'(x) > 0$ then $f(x + \epsilon) > f(x)$ for all $\epsilon > 0$ that are sufficiently small.

Gradients

- Consider function $f(x_1, x_2, \ldots, x_n) : \Re^n \mapsto \Re$.
- The *partial derivative* w.r.t. x_i is denoted
  $\frac{\partial}{\partial x_i} f(x_1, x_2, \ldots, x_n) : \Re^n \mapsto \Re$.
  The partial derivative is the derivative along the x_i axis, keeping all other variables fixed.
- The *gradient* $\nabla f(x_1, x_2, \ldots, x_n) : \Re^n \mapsto \Re^n$ is a function which outputs a vector containing the partial derivatives.
  That is, $\nabla f =< \frac{\partial}{\partial x_1} f, \frac{\partial}{\partial x_2} f, \ldots, \frac{\partial}{\partial x_n} f >$.

- The gradient of f at a point $< x_1, x_2, \ldots, x_n >$ can be thought of as a vector indicating which way is “uphill”.
- If $f \in C^1$, then for all sufficiently small ϵ, $f(x + \epsilon * \nabla f) > f(x)$.



Gradient descent

- In many applications, gradient descent is used to minimize an "error" function when explicit solution is not possible.

- The basic algorithm assumes that ∇f readily computed, and produces a sequence of vectors $\mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3, \ldots$ with the aim that:
  - $f(\mathbf{x}^1) > f(\mathbf{x}^2) > f(\mathbf{x}^3) > \ldots$
  - $\lim_{i \to \infty} \mathbf{x}^i = \mathbf{x}$, locally optimal.

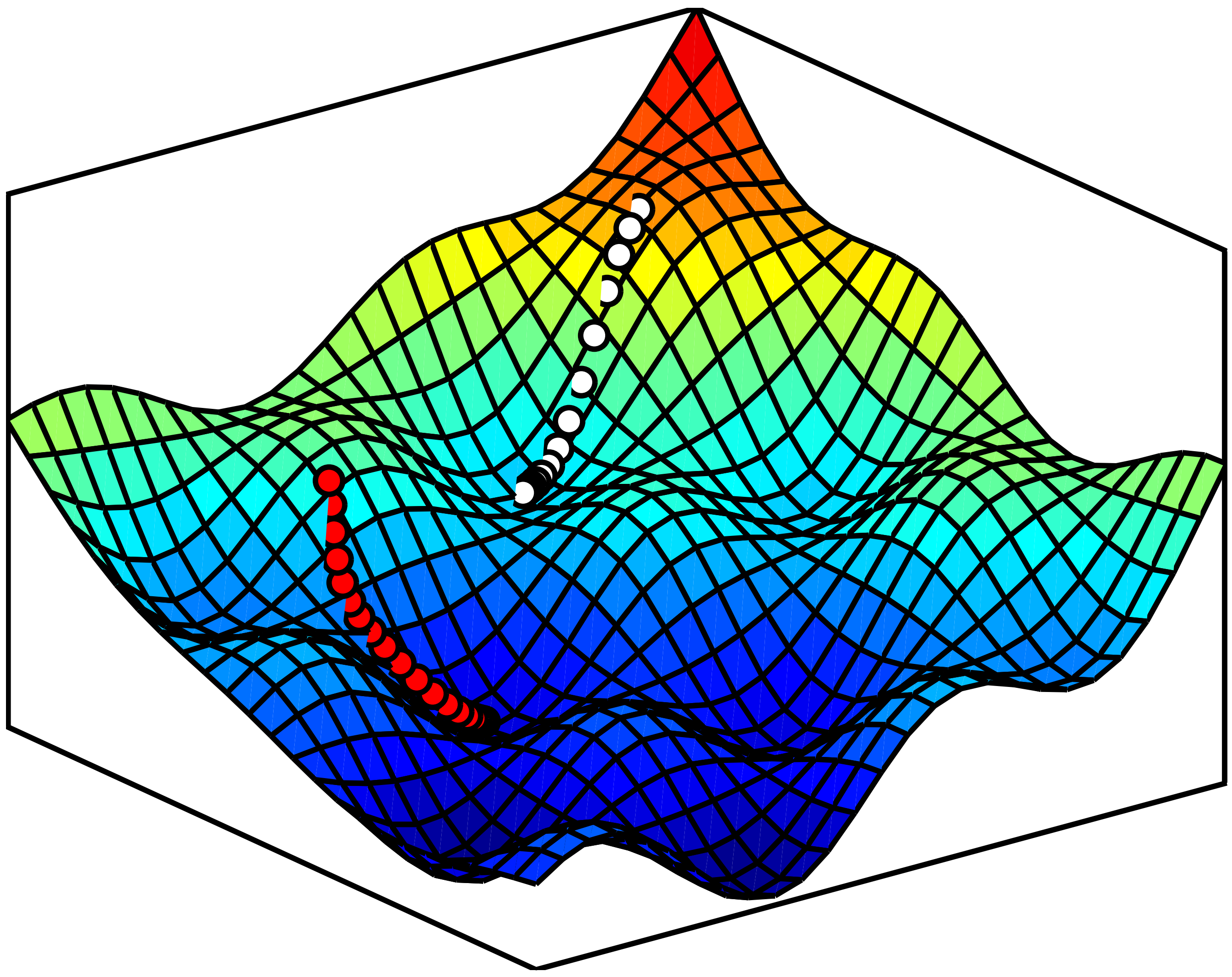
- The algorithm: Given $\mathbf{x}^0$, do for $i = 0, 1, 2, \ldots$

$$\mathbf{x}^{i+1} = \mathbf{x}^i - \alpha_i \nabla f(\mathbf{x}^i) \ ,$$

  where $\alpha_i > 0$ is the "step-size" for iteration i.

- Conditions for convergence? Termination?

Example gradient descent traces



Step-size conditions

- Convergence depends in part on the α_i.
- If they are too large (such as constant) oscillation or “bubbling” may occur.
  (This suggests the α_i should tend to zero as $i \longrightarrow \infty$.)
- If they are too small, the $\mathbf{x}^i$ may not move far enough to reach a local minimum.

Robbins-Monroe Conditions

- The α_i are a Robbins-Monroe sequence if:
  - $\sum_{i=0}^{\infty} \alpha_i = +\infty$
  - $\sum_{i=0}^{\infty} \alpha_i^2 < \infty$
- These conditions, along with appropriate conditions on f are sufficient to ensure that convergence of the $\mathbf{x}^i$.
- Many variants are possible. For example, it is allowed for $\nabla f(\mathbf{x}^i)$ to by a random vector with mean $\nabla f(\mathbf{x}^i)$; this is stochastic gradient descent.

“Batch” versus “On-line” optimization

- Often in machine learning our error function, $\mathcal{E}$, is a sum of errors attributed to each data objects.
  ($\mathcal{E} = \mathcal{E}_1 + \mathcal{E}_2 + \ldots + \mathcal{E}_m$.)

- In “batch” mode gradient descent, the true gradient is computed at each step:

$$\nabla\mathcal{E} = \nabla\mathcal{E}_1 + \nabla\mathcal{E}_2 + \ldots \nabla\mathcal{E}_m.$$

- In “on-line” gradient descent, at each iteration one data object, $j \in \{1, \ldots, m\}$, is chosen at random and only $\nabla\mathcal{E}_j$ is used in the update.

- Why prefer one or the other?

“Batch” versus “On-line” optimization

Why prefer one or the other?

- Batch is simple, repeatable.
- On-line:
  - Requires less computation per step.
  - Randomization may help the procedure escape poor local minima.
  - Allows streams of data objects rather than a static set (hence “on-line”).

Termination

There are many heuristics for deciding when to stop gradient descent.

1. Run until $||\nabla f||$ is smaller than some threshold.
2. Run it for as long as you can stand.
3. Run it for a short time from 100 different starting points, see which one is doing best, goto 2.
4. . . .

Applications in dimensionality reduction

Dimensionality reduction

- Recall: the task of dimensionality reduction is to assign data objects to points in a low-dimensional Euclidean space ($\Re, \Re^2, \Re^3$) such that pairwise distances are preserved as much as possible.

Multi-dimensional scaling

Multi-dimensional scaling does this in the most direct manner possible.

- Input:
  - A dissimilarity matrix $\mathcal{DS}$ for m data objects, where $\mathcal{DS}(i, j)$ is the distance between objects i and j.
  - Desired dimension d of the embedding.
- Output:
  - Coordinates $z_i \in \Re^d$ for each data object i which, as much as possible, minimize a "stress" function which quantifies the mismatch between distances in $\mathcal{DS}$ and distances of data objects' coordinates in $\Re^d$.

Stress functions

Common stress functions include:

- The least-squares or Kruskal-Shephard criterion:

$$\sum_{i=1}^{m} \sum_{i' \neq i} (\mathcal{DS}(i,i') - \|z_i - z_{i'}\|)^2$$

- The Sammon mapping:

$$\sum_{i=1}^{m} \sum_{i' \neq i} \frac{(\mathcal{DS}(i,i') - \|z_i - z_{i'}\|)^2}{\mathcal{DS}(i,i')} ,$$

  which emphasizes getting small distances correct.

For minimizing least-squares or Sammon criteria, one usually resorts to a gradient-based optimization to find the z_i.

Self-organizing maps

- SOMs can also be viewed as a performing something like gradient descent.
- Recall the simple 1D SOM algorithm we discussed earlier:
  - Initialize grid-point coordinates $z_j \in \Re$, $j \in \{1, \ldots, p\}$.
  - For $l = 0, 1, 2, \ldots$
    - Choose a data object x_i at random.
    - Find the nearest grid-point $j \in \arg\min_j \|x_i - z_j\|$.
    - Find the neighborhood of j: $N = \{j' : |j - j'| < r\}$.
    - Update the grid-point coordinates for all $j' \in N$:

$$z_{j'} \leftarrow (1 - \alpha_l) z_{j'} + \alpha_l x_i$$

- What is being minimized?