FSAN/ELEG815: Statistical Learning

Gonzalo R. Arce

Department of Electrical and Computer Engineering
University of Delaware

5a: The Linear Model and Optimization

Linear Regression - Credit Example

Regression $\equiv$ Real-valued output

Classification: Credit approval (yes/no)
Regression: Credit line (dollar amount)

Input: $\mathbf{x} =$

age	23 years
gender	male
annual salary	\$30,000
years in residence	1 year
years in job	1 year
current debt	\$15,000
...	...

Linear regression output: $h(\mathbf{x}) = \sum_{i=0}^{d} w_i x_i = \mathbf{w}^T\mathbf{x}$

Credit Example Again - The data set

Input: $\mathbf{x} =$

age	23 years
gender	male
annual salary	\$30,000
years in residence	1 year
years in job	1 year
current debt	\$15,000
...	...

Output:

$$h(\mathbf{x}) = \sum_{i=0}^{d} w_i x_i = \mathbf{w}^T \mathbf{x}$$

Credit officers decide on credit lines:

$$(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_N, y_N)$$

$y_n \in \mathbb{R}$ is the credit for customer $\mathbf{x}_n$.

Linear regression wants to automate this task, trying to replicate human experts decisions.

E_{out} is unknown

Linear regression algorithm is based on minimizing the squared error:

$$E_{out}(h) = \mathbb{E}[(h(\mathbf{x}) - y)^2]$$

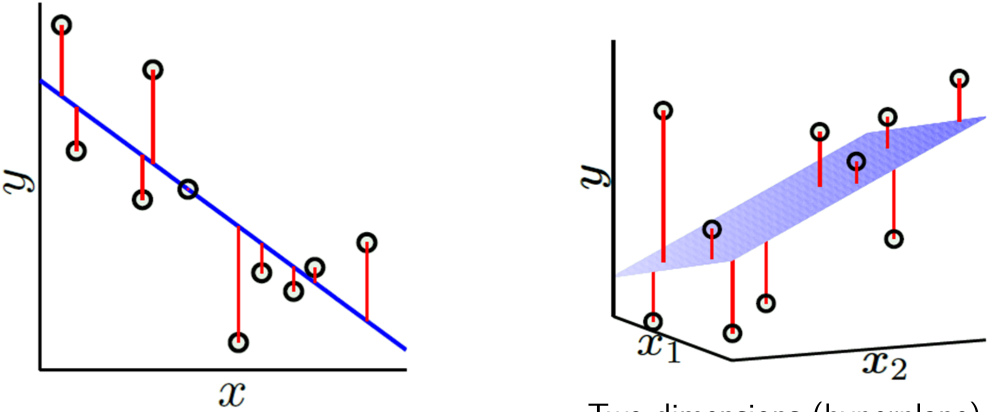
where $\mathbb{E}[\cdot]$ is taken with respect to $P(\mathbf{x}, y)$ that is unknown.
Thus, minimize the in-sample error:

$$E_{in}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x}_n) - y_n)^2$$

Find a hypothesis ($\mathbf{w}$) that achieves a small E_{in}.

Illustration of Linear Regression

The solution hypothesis (in blue) of the linear regression algorithm in one and two dimensions input. The sum of square error is minimized.



One dimension (line)

Two dimensions (hyperplane)

Linear Regression - The Expression for E_{in}

$$\mathbf{y} = w_0 + w_1\mathbf{x}_1 + w_2\mathbf{x}_2 + \ldots + w_p\mathbf{x}_d + \epsilon.$$

$$\underbrace{\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}}_{\mathbf{y}} = \underbrace{\begin{bmatrix} 1 & \mathrm{x}_{11} & \mathrm{x}_{12} & \cdots & \mathrm{x}_{1d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{x}_{N1} & \mathrm{x}_{N2} & \cdots & \mathrm{x}_{Nd} \end{bmatrix}}_{\mathbf{X}} \cdot \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}}_{\mathbf{w}} + \begin{bmatrix} \epsilon \\ \vdots \\ \epsilon \end{bmatrix}$$

$$\begin{aligned} E_{in} &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - y_n)^2 \qquad \mathbf{X} \in \mathbb{R}^{N\times(d+1)} \\ &= \frac{1}{N}||\mathbf{X}\mathbf{w}-\mathbf{y}||_2^2 = \frac{1}{N}(\mathbf{X}\mathbf{w}-\mathbf{y})^T(\mathbf{X}\mathbf{w}-\mathbf{y}) \\ &= \frac{1}{N}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{y}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{y}^T\mathbf{y}) \\ &= \frac{1}{N}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{y}^T\mathbf{y}) \end{aligned}$$

Learning Algorithm - Minimizing E_{in}

$$
\begin{aligned}
\hat{\mathbf{w}} &= \arg\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{N}||\mathbf{X}\mathbf{w}-\mathbf{y}||_2^2 \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{N}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}-2\mathbf{w}^T\mathbf{X}^T\mathbf{y}+\mathbf{y}^T\mathbf{y})
\end{aligned}
$$

Observation: The error is a quadratic function of $\mathbf{w}$

Consequences: The error is an $(d+1)$–dimensional bowl–shaped function of $\mathbf{w}$ with a unique minimum

Result: The optimal weight vector, $\hat{\mathbf{w}}$, is determined by differentiating $E_{in}(\mathbf{w})$ and setting the result to zero

$$\nabla_{\mathbf{w}} E_{in}(\mathbf{w}) = 0$$

- A closed form solution exists

Example

Consider a two dimensional case. Plot the error surface and error contours.

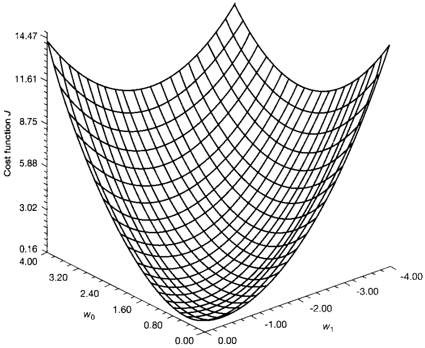


Figure 5.6 Error-performance surface of the two-tap transversal filter described in the numerical example.

Error Surface

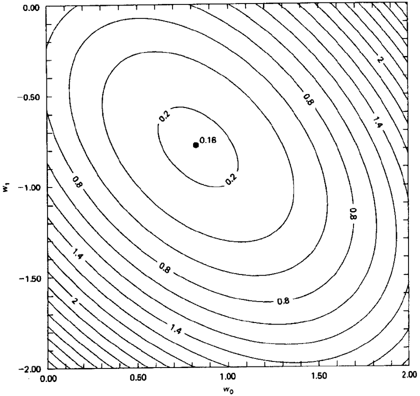


Figure 5.7 Contour plots of the error-performance surface depicted in Fig. 5.6.

Error Contours

Aside (Matrix Differentiation, Real Case):

Let $\mathbf{w} \in \mathbb{R}^{(d+1)}$ and let $f : \mathbb{R}^{(d+1)} \rightarrow \mathbb{R}$. The derivative of f (called gradient of f) with respect to $\mathbf{w}$ is:

$$\nabla_{\mathbf{w}}(f) = \frac{\partial f}{\partial \mathbf{w}} = \begin{bmatrix} \nabla_0(f) \\ \nabla_1(f) \\ \vdots \\ \nabla_d(f) \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial w_0} \\ \frac{\partial f}{\partial w_1} \\ \vdots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}$$

Thus,

$$\nabla_k(f) = \frac{\partial f}{\partial w_k}, \qquad k = 0, 1, \cdots, d$$

Example

Now suppose $f = \mathbf{c}^T\mathbf{w}$. Find $\nabla_{\mathbf{w}}(f)$

In this case,

$$f = \mathbf{c}^T\mathbf{w} = \sum_{k=0}^{d} w_k c_k$$

and

$$\nabla_k(f) = \frac{\partial f}{\partial w_k} = c_k, \qquad k = 0, 1, \cdots, d$$

Result: For $f = \mathbf{c}^T\mathbf{w}$

$$\nabla_{\mathbf{w}}(g) = \begin{bmatrix} \nabla_0(g) \\ \nabla_1(g) \\ \vdots \\ \nabla_{\text{d}}(g) \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{bmatrix} = \mathbf{c}$$

Same for $f = \mathbf{w}^T\mathbf{c}$.

Example

Lastly, suppose $f = \mathbf{w}^T \mathbf{Q} \mathbf{w}$. Where $\mathbf{Q} \in \mathbb{R}^{(d+1)\times(d+1)}$ and $\mathbf{w} \in \mathbb{R}^{d+1}$. Find $\nabla_{\mathbf{w}}(f)$

In this case, using the product rule:

$$
\begin{aligned}
\nabla_{\mathbf{w}} f &= \frac{\partial \mathbf{w}^T(\mathbf{Q}\bar{\mathbf{w}})}{\partial \mathbf{w}} + \frac{\partial(\bar{\mathbf{w}}^T \mathbf{Q})\mathbf{w}}{\partial \mathbf{w}} \\
&= \frac{\partial \mathbf{w}^T \mathbf{u}_1}{\partial \mathbf{w}} + \frac{\partial \mathbf{u}_2^T \mathbf{w}}{\partial \mathbf{w}}
\end{aligned}
$$

Using previous result, $\frac{\partial \mathbf{c}^T \mathbf{w}}{\partial \mathbf{w}} = \frac{\partial \mathbf{w}^T \mathbf{c}}{\partial \mathbf{w}} = \mathbf{c}$,

$$
\begin{aligned}
\nabla_{\mathbf{w}} f &= \mathbf{u}_1 + \mathbf{u}_2, \\
&= \mathbf{Q}\mathbf{w} + \mathbf{Q}^T \mathbf{w} = (\mathbf{Q} + \mathbf{Q}^T)\mathbf{w}, \qquad \text{if } \mathbf{Q} \text{ symmetric, } \mathbf{Q}^T = \mathbf{Q} \\
&= 2\mathbf{Q}\mathbf{w}
\end{aligned}
$$

Returning to the MSE performance criteria

$$E_{in}(\mathbf{w}) = \left[\frac{1}{N}(\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{y}^T\mathbf{y})\right]$$

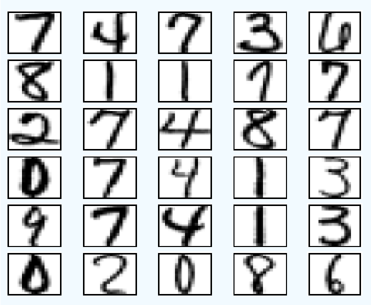
Differentiating with respect to $\mathbf{w}$ and setting equal to zero, we obtain,

$$\begin{aligned}\nabla E_{in}(\mathbf{w}) &= \frac{1}{N}(2\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{X}^T\mathbf{y} + 0) \\ &= \frac{2}{N}\mathbf{X}^T\mathbf{X}\mathbf{w} - \frac{2}{N}\mathbf{X}^T\mathbf{y} = 0\end{aligned}$$

$$\begin{aligned}\mathbf{X}^T\mathbf{X}\mathbf{w} &= \mathbf{X}^T\mathbf{y} \\ \hat{\mathbf{w}} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \\ &= \mathbf{X}^{\dagger}\mathbf{y}\end{aligned}$$

where $\mathbf{X}^{\dagger} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ is the pseudo-inverse of $\mathbf{X}$.

A real data set

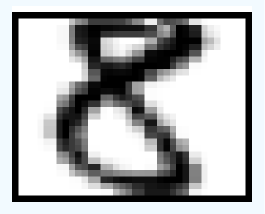


16x16 pixels gray-scale images of digits from the US Postal Service Zip Code Database. The goal is to recognize the digit in each image.

This is not a trivial task (even for a human). A typical human error E_{out} is about 2.5% due to common confusions between $\{4, 9\}$ and $\{2, 7\}$.

Machine Learning tries to achieve or beat this error.

Input Representation



Since the images are 16×16 pixels:

- 'raw' input
  $\mathbf{x}_r = (x_0, x_1, x_2, \cdots, x_{256})$
- Linear model:
  $(w_0, w_1, w_2, \cdots, w_{256})$

It has too many many parameters.
A better input representation makes it simpler.

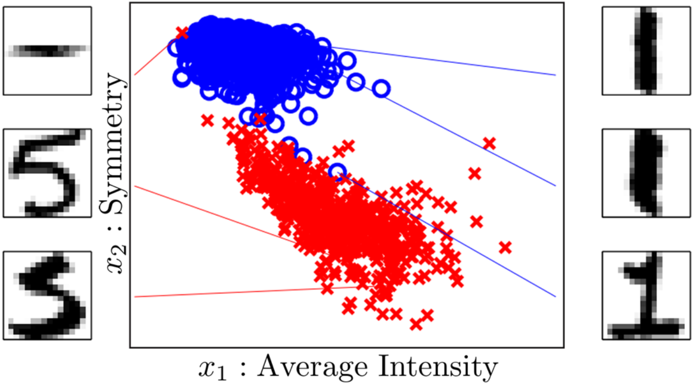
Features: Extract useful information, e.g.,

- Average intensity and symmetry
  $\mathbf{x} = (x_0, x_1, x_2)$
- Linear model: (w_0, w_1, w_2)

The descriptors must be representative of the data.

Illustration of Features

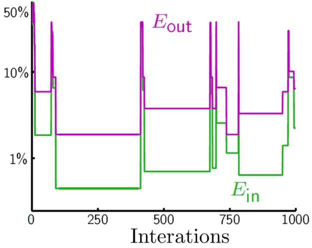
$$\mathbf{x} = (x_0, x_1, x_2) \quad x_0 = 1$$



It's almost linearly separable. However, it is impossible to have them all right.

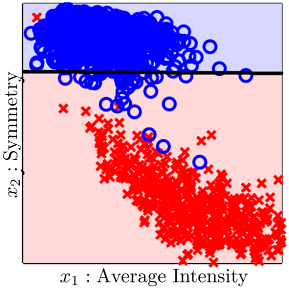
What Perceptron Learning Algorithm does?

Evolution of in-sample error E_{in} and out-of-sample error E_{out} as a function of iterations of PLA



- Assume we know E_{out} .
- E_{in} tracks E_{out}. PLA generalizes well!

- It would never converge (data not linearly separable).
- **Stopping criteria:** Max. number of iterations.



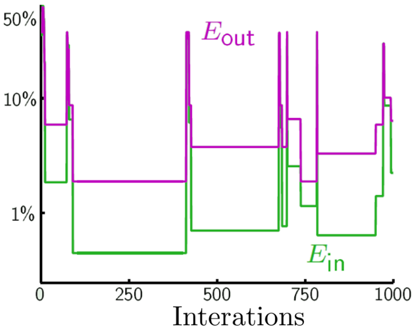
Final perceptron boundary

We can do better...

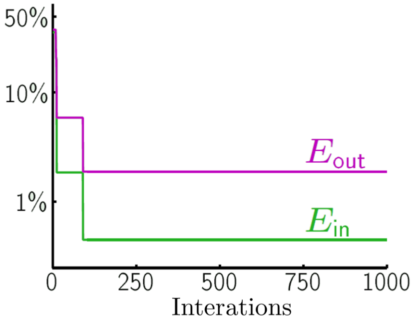
The 'pocket' algorithm

Keeps 'in its pocket' the best weight vector encountered up to the current iteration t in PLA.

PLA

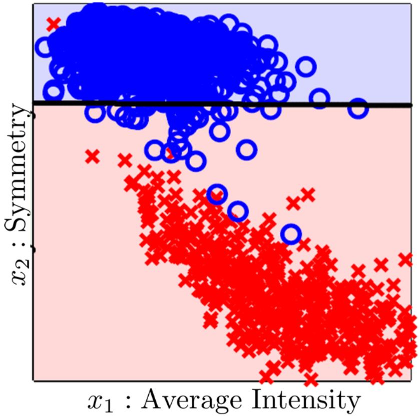


Pocket

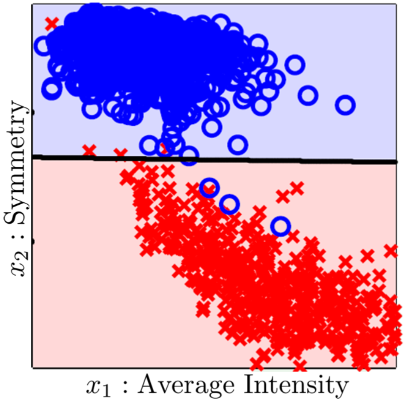


Classification boundary - PLA versus Poket

PLA



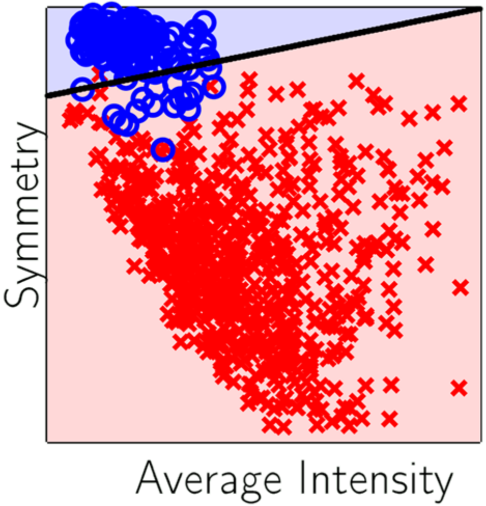
Pocket



Linear Regression for Classification

- Linear regression learns a real-valued function $y = f(\mathbf{x}) \in \mathbb{R}$
- Binary-valued functions are also real-valued! $\pm 1 \in \mathbb{R}$
- Use linear regression to get $\mathbf{w}$ where $\mathbf{w}^T \mathbf{x}_n \approx y_n = \pm 1$
- In this case, $\text{sign}(\mathbf{w}^T \mathbf{x}_n)$ is likely to agree with y_n
- Good initial weights for classification

Linear regression boundary



Definition (Steepest Descent (SD))

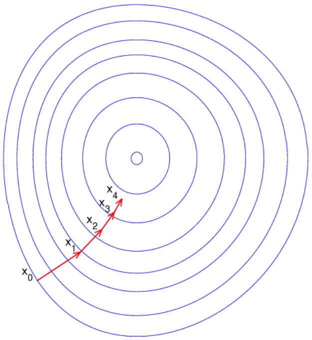
Steepest descent, also known as gradient descent, it is an iterative technique for finding the local minimum of a function.

Approach: Given an arbitrary starting point, the current location (value) is moved in steps proportional to the negatives of the gradient at the current point.

- SD is an old, deterministic method, that is the basis for stochastic gradient based methods
- SD is a feedback approach to finding local minimum of an error performance surface
- The error surface must be known *a priori*
- In the MSE case, SD converges converges to the optimal solution without inverting a matrix

Example

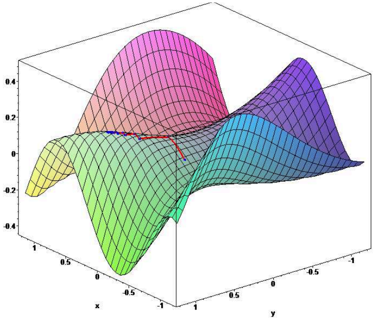
Consider a well structured cost function with a single minimum. The optimization proceeds as follows:



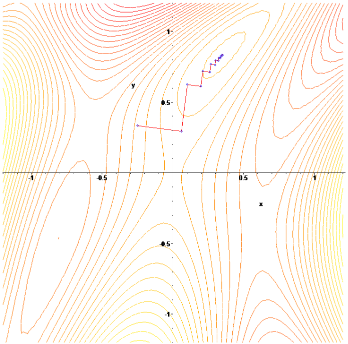
Contour plot showing that evolution of the optimization

Example

Consider a gradient ascent example in which there are multiple minima/maxima



Surface plot showing the multiple minima and maxima



Contour plot illustrating that the final result depends on starting value

More General - Gradient Descent

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} E_{in}(\mathbf{w})$$

- Use the method of **Gradient Descent (GD)** to minimize the in-sample error:

$$E_{in}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathrm{e}(f(\mathbf{x}_n, \mathbf{w}), y_n)$$

by iterative steps along $-\nabla E_{in}$:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta\nabla E_{in}(\mathbf{w}(t))$$

where η is the step size.

Gradient Descent (GD) and Stochastic Gradient Descent (SGD)

$$E_{in}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathrm{e}(f(\mathbf{x}_n,\mathbf{w}),y_n)$$

Gradient descent update:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta\nabla E_{in}(\mathbf{w}(t))$$

For $\mathrm{e}(h(\mathbf{x}_n,y_n)) = (\mathbf{w}^T\mathbf{x}_n - y_n)^2$ i.e. for the mean squared error:

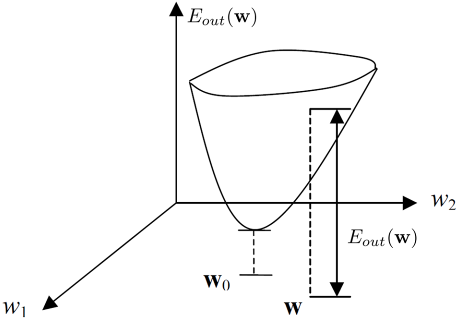
$$\nabla E_{in}(\mathbf{w}) = \frac{2}{N}(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y})$$

Note: ∇E_{in} is based on all examples $(\mathbf{x}_n,y_n)$

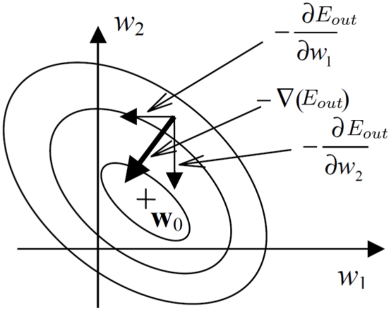
It is known as *batch* gradient descent.

Example

The MSE is a bowl–shaped surface, which is a function of the 2-D space weight vector $\mathbf{w}(n)$



Surface Plot

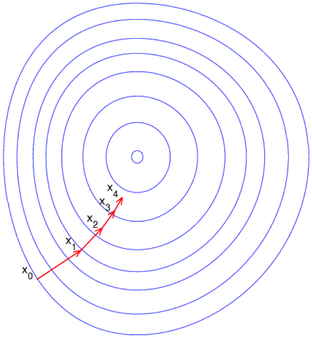


Contour Plot

Imagine dropping a marble at any point on the bowl-shaped surface.
The ball will reach the minimum point by going through the path of steepest descent.

Example

Consider a well structured cost function with a single minimum. The optimization proceeds as follows:



Contour plot showing that evolution of the optimization

Stochastic Gradient Descent (SGD)

Instead of considering the full *batch*, for each iteration, pick one training data point $(\mathbf{x}_n, y_n)$ at random and apply GD update to $\mathrm{e}(h(\mathbf{x}_n, y_n))$

The weight update of SGD is:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla \mathrm{e}_n(\mathbf{w}(t))$$

For $\mathrm{e}(h(\mathbf{x}_n, y_n)) = (\mathbf{w}^T\mathbf{x}_n - y_n)^2$ i.e. for the mean squared error:

$$\nabla \mathrm{e}_n(\mathbf{w}) = 2(\mathbf{x}_n\mathbf{w}^T\mathbf{x}_n - \mathbf{x}_n y_n)$$

Stochastic Gradient Descent (SGD)

Since n is picked at random, the expected weight change is:

$$\begin{aligned}\mathbb{E}_n\left[-\nabla \mathrm{e}(h(\mathbf{x}_n, y_n))\right] &= \frac{1}{N}\sum_{n=1}^{N} -\nabla \mathrm{e}(h(\mathbf{x}_n, y_n)) \\ &= -\nabla E_{in}\end{aligned}$$

Same as the *batch* gradient descent.

Result: On 'average' the minimization proceeds in the right direction (remember LMS).

Stochastic Gradient Descent (SGD)

Instead of considering the full *batch*, for each iteration, pick one training data point $(\mathbf{x}_n, y_n)$ at random and apply GD update to $\mathrm{e}(h(\mathbf{x}_n, y_n))$

The weight update of SGD is:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla \mathrm{e}_n(\mathbf{w}(t))$$

Since n is picked at random, the expected weight change is:

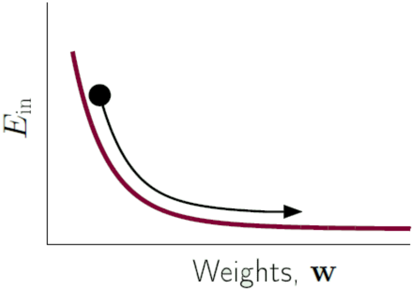
$$\begin{aligned} \mathbb{E}_n\left[-\nabla \mathrm{e}(h(\mathbf{x}_n, y_n))\right] &= \frac{1}{N}\sum_{n=1}^{N} -\nabla \mathrm{e}(h(\mathbf{x}_n, y_n)) \\ &= -\nabla E_{in} \end{aligned}$$

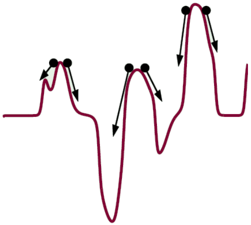
Same as the *batch* gradient descent.

Result: On 'average' the minimization proceeds in the right direction.

Benefits of SGD

1. Cheaper computation (by a factor of N compare to GD)
2. Randomization
3. Simple





Rule of thumb:

Start with:

$$\eta = 0.1 \quad \text{works!}$$

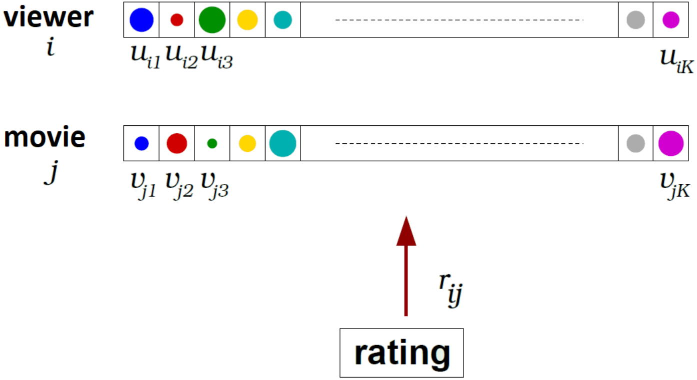
Randomization helps to avoid local minima and flat regions.

SGD is successful in practice!

SGD in Action

Remember movie ratings, we solved this using SVD:

- Describe the movie as an array of factors $\mathbf{v}_j$
- Describe each viewer using same factors $\mathbf{u}_i$
- Rating r_{ij} based on match/mismatch



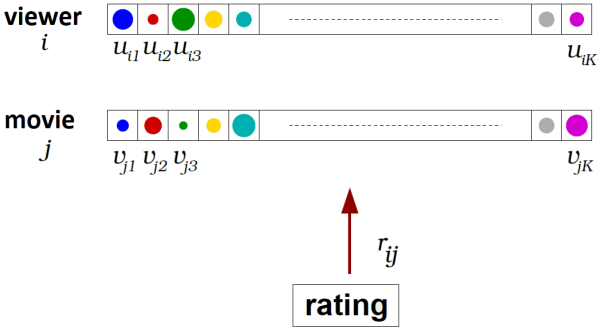
A model for movie rating

SVD is computationally expensive and requires care dealing with missing data, use **SGD** instead.

The Learning Approach

The learning algorithm does **reverse-engineering** (estimates factors from a given rating).

- Starts with random (meaningless) factors
- Tunes factors to be aligned with a previous rating.
- Does the same for millions of ratings, cycling over and over.
- Eventually the factors are meaningful (complete).

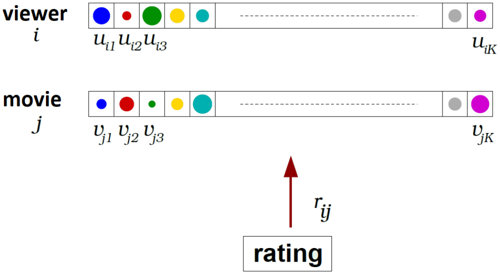


Let's use **Stochastic Gradient Descent**

SGD in Action

Consider the error for each data point r_{ij} as

$$e_{i,j} = \left(r_{ij} - \sum_{k=1}^{K} u_{ik} v_{jk} \right)^2 = (r_{ij} - \mathbf{u}_i^T \mathbf{v}_j)^2$$



Regularized Minimization problem:

$$\min_{\mathbf{u}^*, \mathbf{v}^*} \sum_{(l,m)\in\mathcal{K}} (r_{lm} - \mathbf{u}_l^T \mathbf{v}_m)^2 + \lambda(||\mathbf{u}_l||^2 + ||\mathbf{v}_m||^2)$$

r_{lm} with $(l,m) \in \mathcal{K}$ is the set of all known ratings. Apply SGD to compute $\mathbf{u}^*$ and $\mathbf{v}^*$:

$$\begin{aligned} \mathbf{u}_l(t+1) &= \mathbf{u}_l(t) - \eta \nabla e_{lm}(\mathbf{u}_l(t)) \\ \mathbf{v}_m(t+1) &= \mathbf{v}_m(t) - \eta \nabla e_{lm}(\mathbf{v}_m(t)) \end{aligned}$$

SGD in Action

For each known rating, compute the gradient:

$$
\begin{aligned}
\nabla \mathsf{e}_{lm}(\mathbf{u}_l) &= -2\mathbf{v}_m(r_{lm} - \mathbf{u}_l^T\mathbf{v}_m) + 2\lambda\mathbf{u}_l \\
\nabla \mathsf{e}_{lm}(\mathbf{v}_m) &= -2\mathbf{u}_l(r_{lm} - \mathbf{u}_l^T\mathbf{v}_m) + 2\lambda\mathbf{v}_m
\end{aligned}
$$

Thus, the parameters (factors) are updated according to:

$$
\begin{aligned}
\mathbf{u}_l(t+1) &= \mathbf{u}_l(t) - 2\eta(-\mathbf{v}_m(t)e_{lm}(t) + \lambda\mathbf{u}_l(t)) \\
\mathbf{v}_m(t+1) &= \mathbf{v}_m(t) - 2\eta(-\mathbf{u}_l(t)e_{lm}(t) + \lambda\mathbf{v}_m(t))
\end{aligned}
$$

Rearranging and setting $\gamma = 2\eta$:

$$
\begin{aligned}
\mathbf{u}_l(t+1) &= \mathbf{u}_l(t) + \gamma(e_{lm}(t)\mathbf{v}_m(t) - \lambda\mathbf{u}_l(t)) \\
\mathbf{v}_m(t+1) &= \mathbf{v}_m(t) + \gamma(e_{lm}(t)\mathbf{u}_l(t) - \lambda\mathbf{v}_m(t))
\end{aligned}
$$

where $e_{lm} = r_{lm} - \mathbf{u}_l^T\mathbf{v}_m$, γ is the learning rate parameter and λ a regularization parameter.