Locally Linear Embedding

Locally linear embedding (LLE), is an unsupervised learning algorithm that computes low-dimensional neighborhood preserving embeddings.

- Given N real-valued vectors $\mathbf{X}_i$, each of dimensionality D.
- Suppose data point and its neighbors lie on or close to a locally *linear* patch.
- Characterize the local geometry of these patches by linear coefficients that reconstruct each data point from its neighbors.

Reconstruction errors are measured by cost function

$$E(\mathbf{W}) = \sum_i (\mathbf{X}_i - \sum_j W_{ij}\mathbf{X}_j)^2.$$

Locally Linear Embedding (cont.)

- Cost function adds up squared distances between all the data points and their reconstructions.
- Weights W_{ij} summarize the contribution of the jth data point to the ith reconstruction.
- For determining the weights W_{ij} the cost function is minimized subject to two constraints.
  1. Data point $\mathbf{X}_i$ is reconstructed *only* from its neighbors (enforcing $W_{ij} = 0$ if $\mathbf{X}_j$ does not belong to the set of neighbors of $\mathbf{X}_i$.
  2. Rows in the weight matrix sum to 1: $\sum_j W_{ij} = 1$.
- Optimal weights W_{ij} subject to these constraints are found by solving a least-squares problem.

Locally Linear Embedding (cont.)

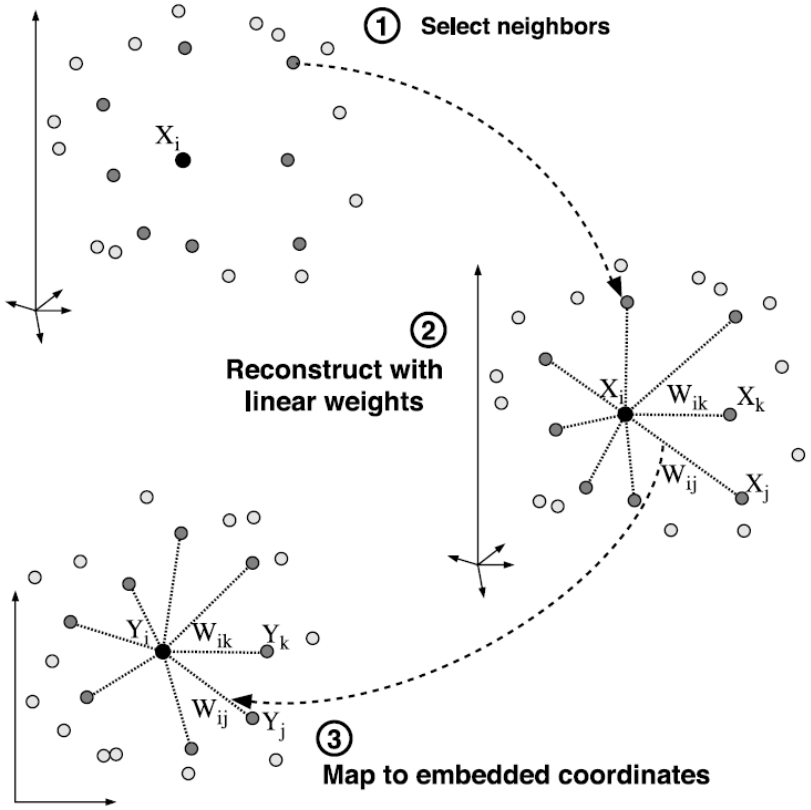
- Constrained weights that minimize the reconstruction errors obey an important symmetry
  - For any particular data point they are invariant to rotations, rescalings, and translations of that data point and its neighbors.

In the final step of LLE, high-dimensional observation $\mathbf{X}_i$ is mapped to a low-dimensional vector $\mathbf{Y}_i$ by choosing d-dimensional coordinates $\mathbf{Y}_i$ ($d \ll D$) such that the embedding cost function

$$\Phi(\mathbf{Y}) = \sum_i (\mathbf{Y}_i - \sum_j W_{ij} \mathbf{Y}_j)^2.$$

is minimized. However, now we fix the weights W_{ij} while optimizing the coordinates $\mathbf{Y}_i$.

Locally Linear Embedding (cont.)



Nonlinear dimensionality reduction by locally linear embedding.
Sam Roweis & Lawrence Saul. Science v.290 no.5500,
Dec.22, 2000. pp.2323–2326.

Method of Steepest Descent

Let $E(\mathbf{w})$ be a continuously differentiable function of some unknown (weight) vector $\mathbf{w}$.

Find an optimal solution $\mathbf{w}^\star$ that satisfies the condition

$$E(\mathbf{w}^\star) \leq E(\mathbf{w}).$$

The necessary condition for optimality is

$$\nabla E(\mathbf{w}^\star) = \mathbf{0}.$$

Let us consider the following *iterative* descent:

Start with an initial guess $\mathbf{w}^{(0)}$ and generate sequence of weight vectors $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots$ such that

$$E(\mathbf{w}^{(i+1)}) \leq E(\mathbf{w}^{(i)}).$$

Steepest Descent Algorithm

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta \nabla E(\mathbf{w}^{(i)})$$

where η is a positive constant called learning rate.

At each iteration step the algorithm applies the correction

$$\begin{aligned} \Delta \mathbf{w}^{(i)} &= \mathbf{w}^{(i+1)} - \mathbf{w}^{(i)} \\ &= -\eta \nabla E(\mathbf{w}^{(i)}) \end{aligned}$$

Steepest descent algorithm satisfies:

$$E(\mathbf{w}^{(i+1)}) \leq E(\mathbf{w}^{(i)}),$$

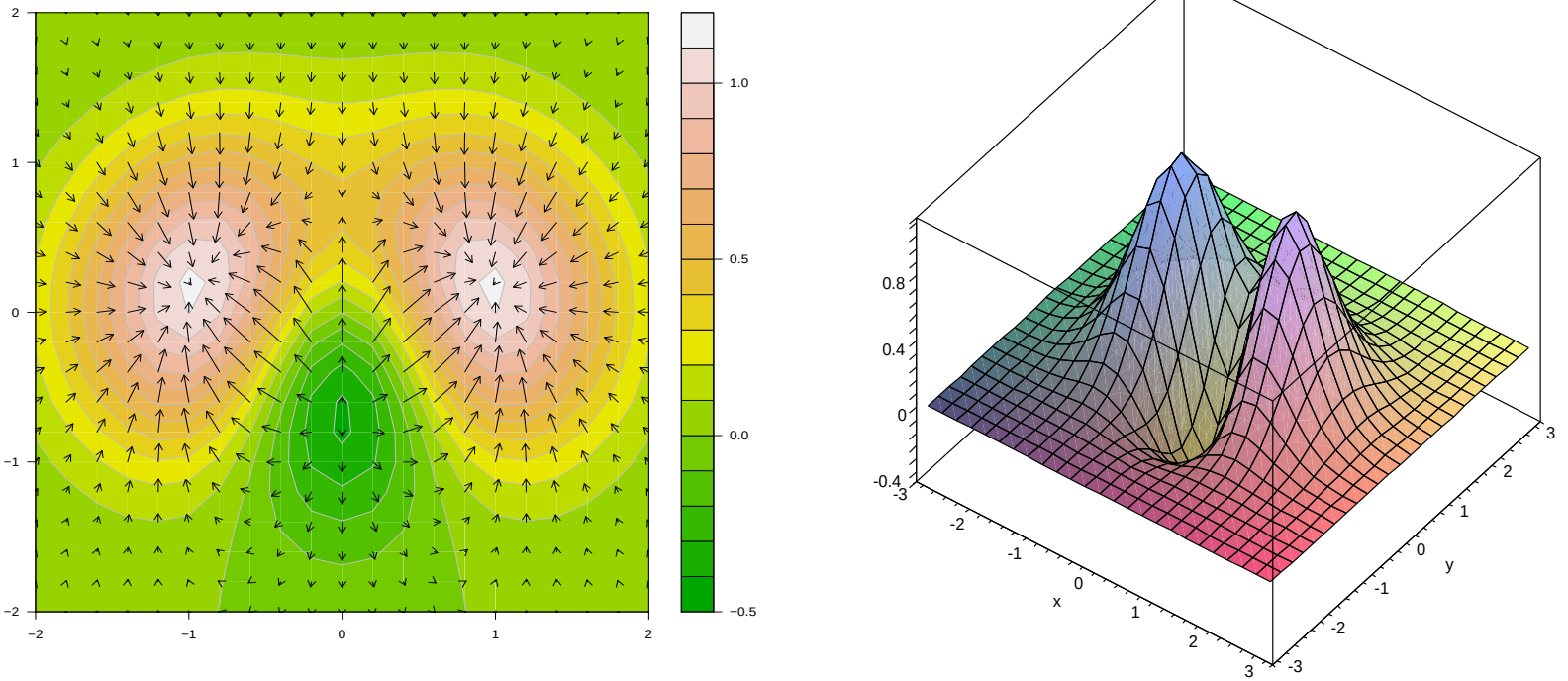
to see this, use first-order Taylor expansion around $\mathbf{w}^{(i)}$ to approximate $E(\mathbf{w}^{(i+1)})$ as $E(\mathbf{w}^{(i)}) + (\nabla E(\mathbf{w}^{(i)}))^T \Delta \mathbf{w}^{(i)}$.

Steepest Descent Algorithm (cont.)

$$\begin{aligned} E(\mathbf{w}^{(i+1)}) &\approx E(\mathbf{w}^{(i)}) + (\nabla E(\mathbf{w}^{(i)}))^T \Delta \mathbf{w}^{(i)} \\ &= E(\mathbf{w}^{(i)}) - \eta \|\nabla E(\mathbf{w}^{(i)})\|^2 \end{aligned}$$

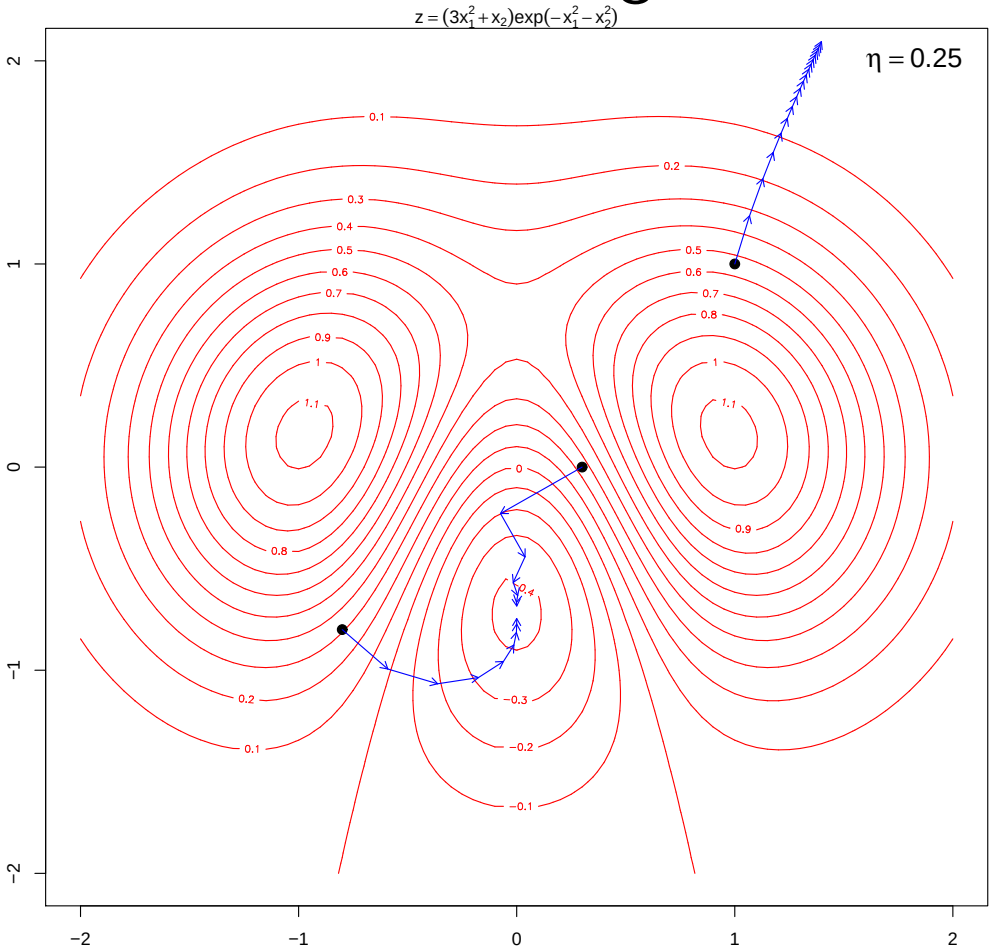
For positive learning rate η, $E(\mathbf{w}^{(i)})$ decreases in each iteration step (for small enough learning rates).

$$(3x^2 + y)\exp(-x^2 - y^2)$$



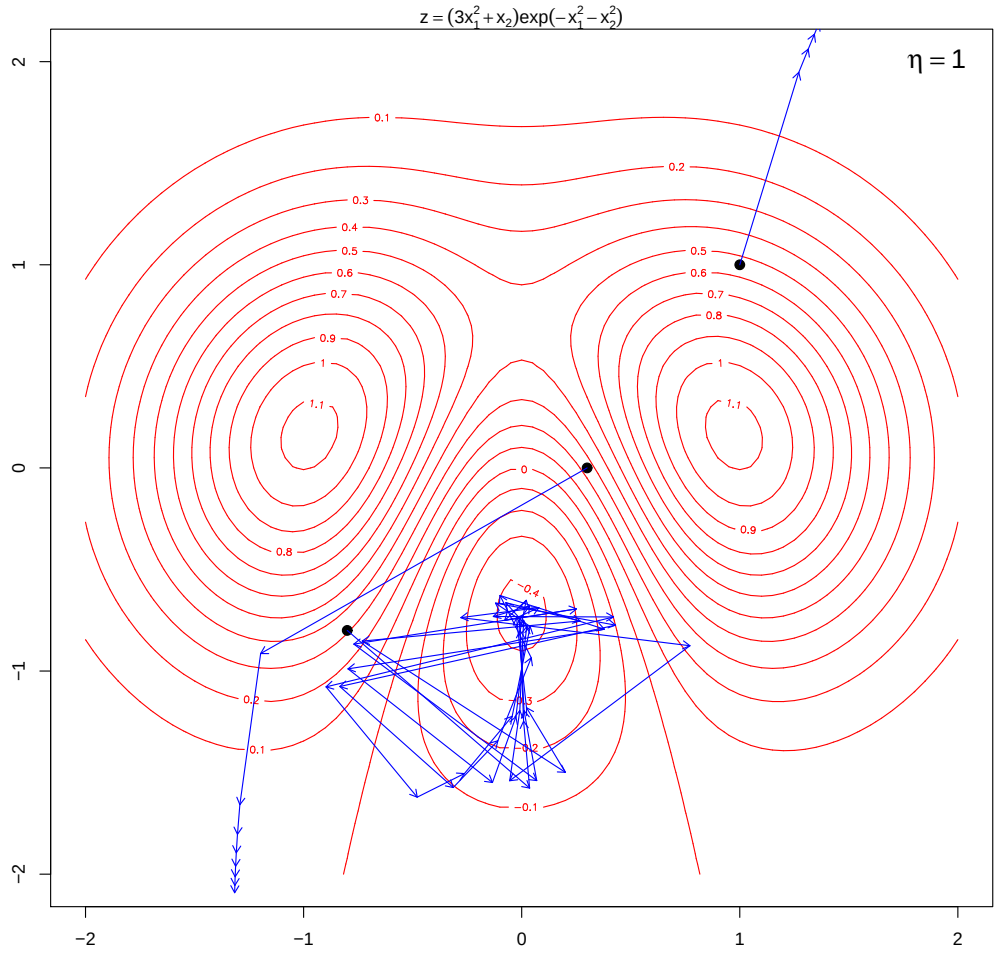
Steepest Descent Algorithm Example

Black points denote different starting values. Learning rate η is properly chosen, however for starting value $(1, 1)$, algorithm converges not to the global minimum. It follows steepest descent in the "wrong direction", in other words, gradient based algorithms are local search algorithms.



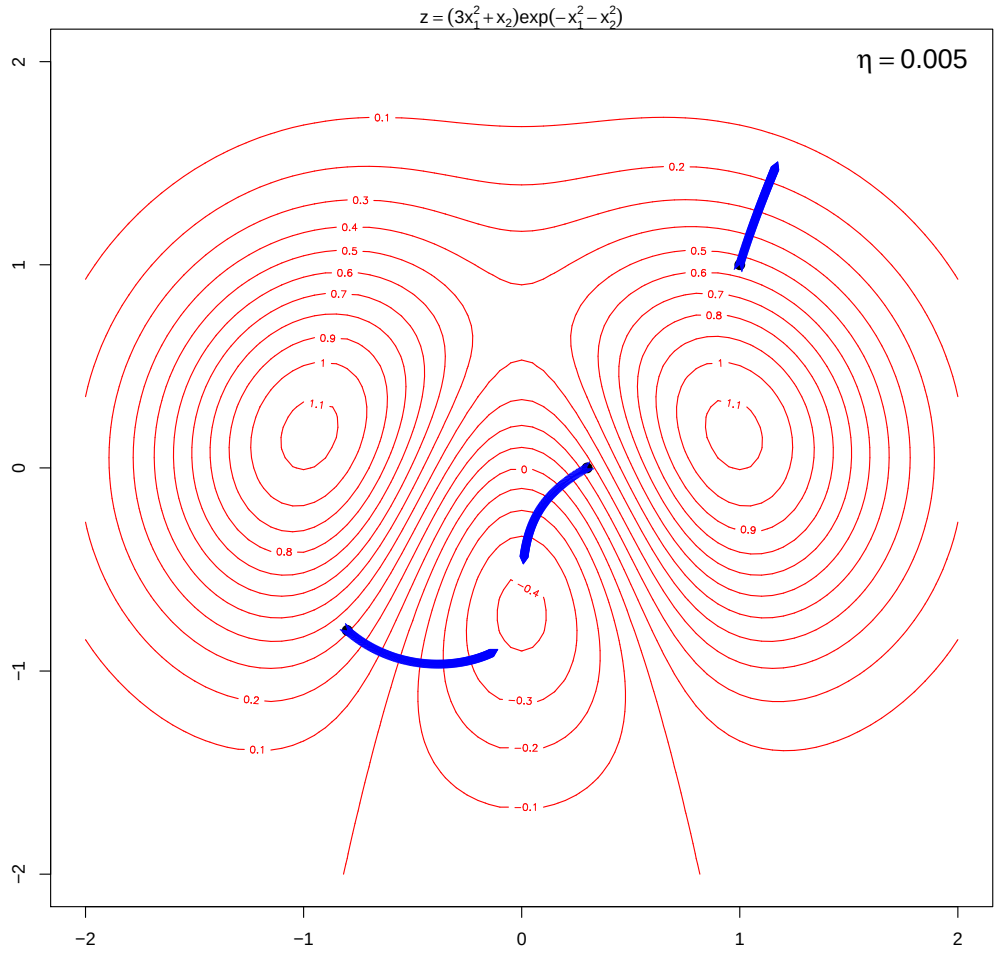
Steepest Descent Algorithm Example (cont.)

Learning rate $\eta = 1.0$ is too large, algorithm oscillates in a "*zig-zag*" manner or "*overleap*" the global minimum.

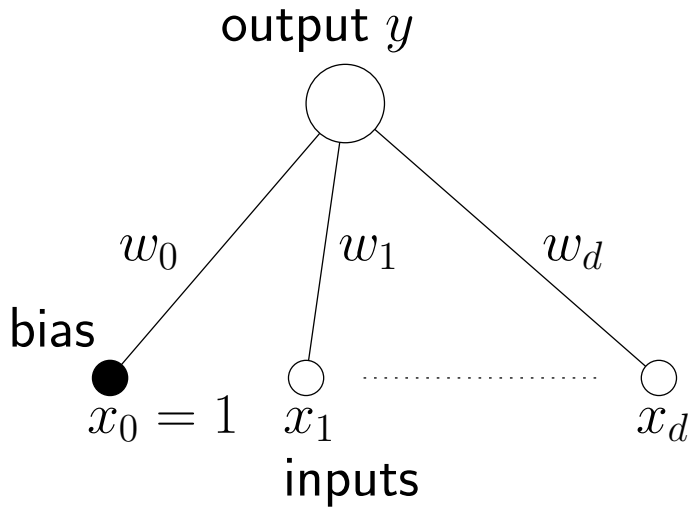


Steepest Descent Algorithm Example (cont.)

Learning rate $\eta = 0.005$ is too small, algorithm converges "very slowly".



Single-Layer Network



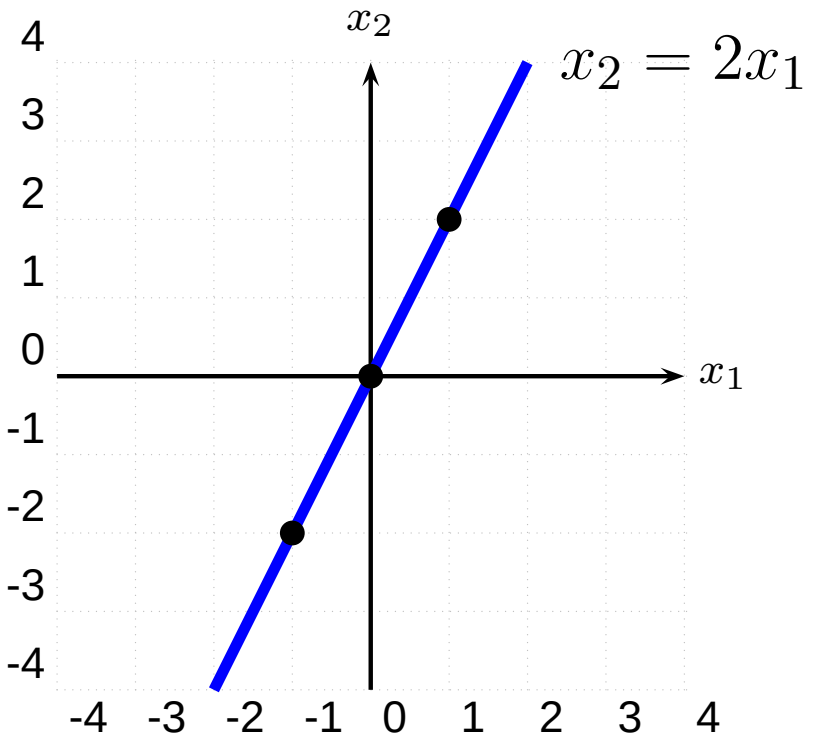
Equivalent notation:

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

where $\widetilde{\mathbf{w}} = (w_0, \mathbf{w})$ and $\widetilde{\mathbf{x}} = (1, \mathbf{x})$.

Linear Classifier

- Linear classifiers are single layer neural networks.



Observe, that $x_2 = 2x_1$ can also be expressed as

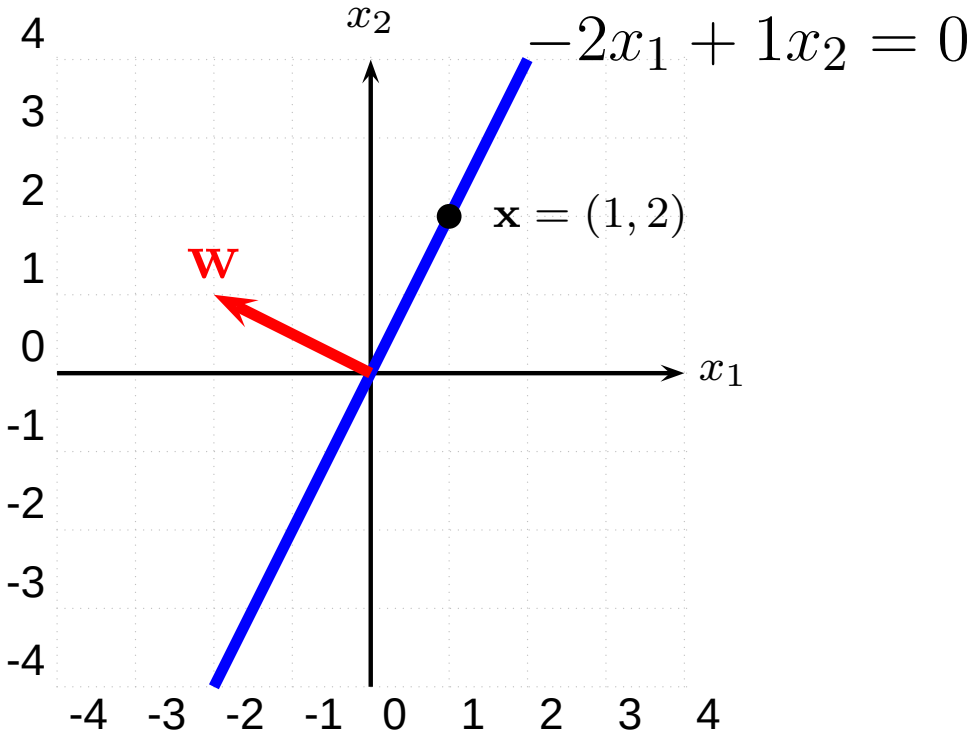
$$w_1 x_1 + w_2 x_2 = 0 \Leftrightarrow x_2 = -\frac{w_1}{w_2} x_1,$$

where for instance

$$w_1 = -2,\ w_2 = 1.$$

Furthermore, observe that all points lying on the line $x_2 = 2x_1$ satisfy $w_1 x_1 + w_2 x_2 = -2x_1 + 1x_2 = 0$.

Linear Classifier & Dot Product



- What about the vector $\mathbf{w} = (w_1, w_2) = (-2, 1)$?
- Vector $\mathbf{w}$ is perpendicular to the line $-2x_1 + 1x_2 = 0$.
- Let us calculate the dot product of $\mathbf{w}$ and $\mathbf{x}$.

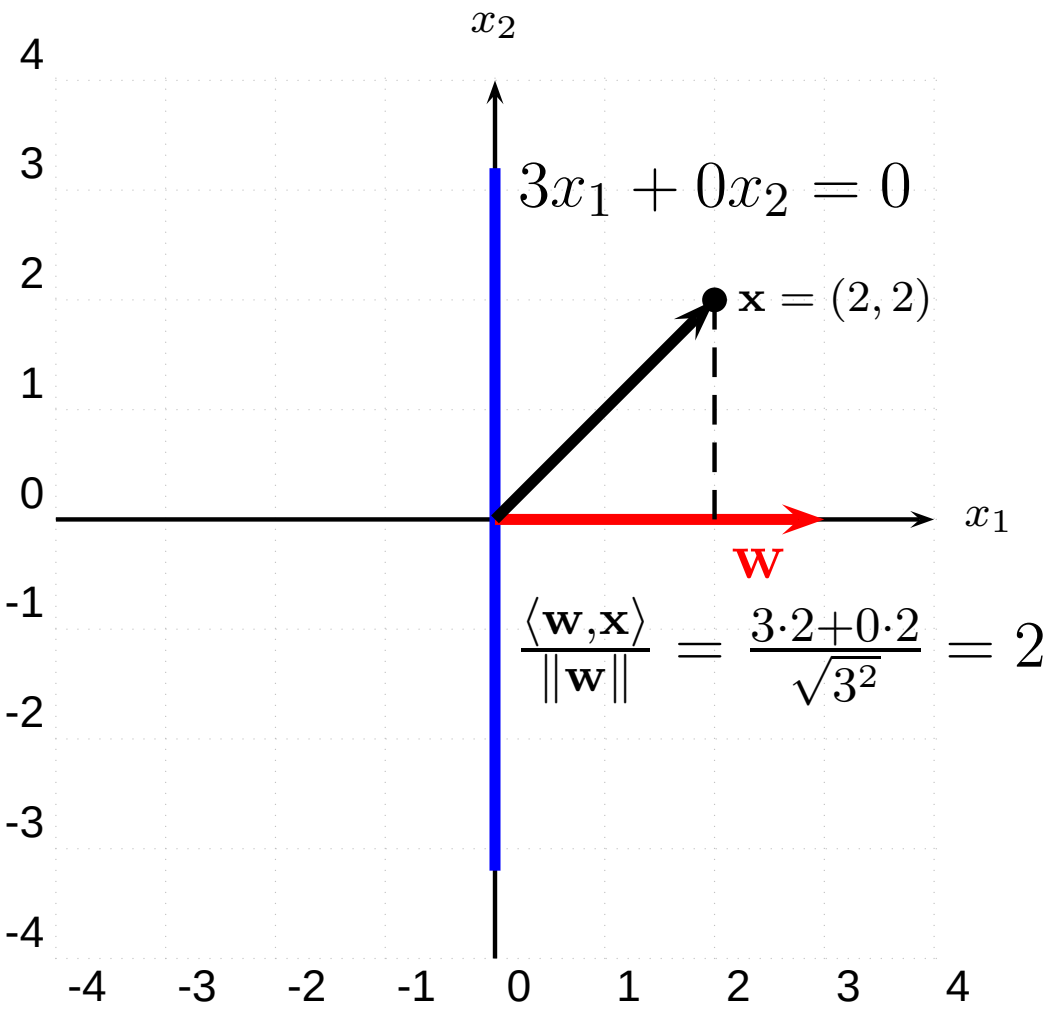
The dot product is defined as

$w_1x_1 + w_2x_2 + \ldots + w_dx_d \stackrel{def}{=} \langle \mathbf{w}, \mathbf{x} \rangle$, for some $d \in \mathbb{N}$.

In our example $d = 2$ and we obtain $-2 \cdot 1 + 1 \cdot 2 = 0$.

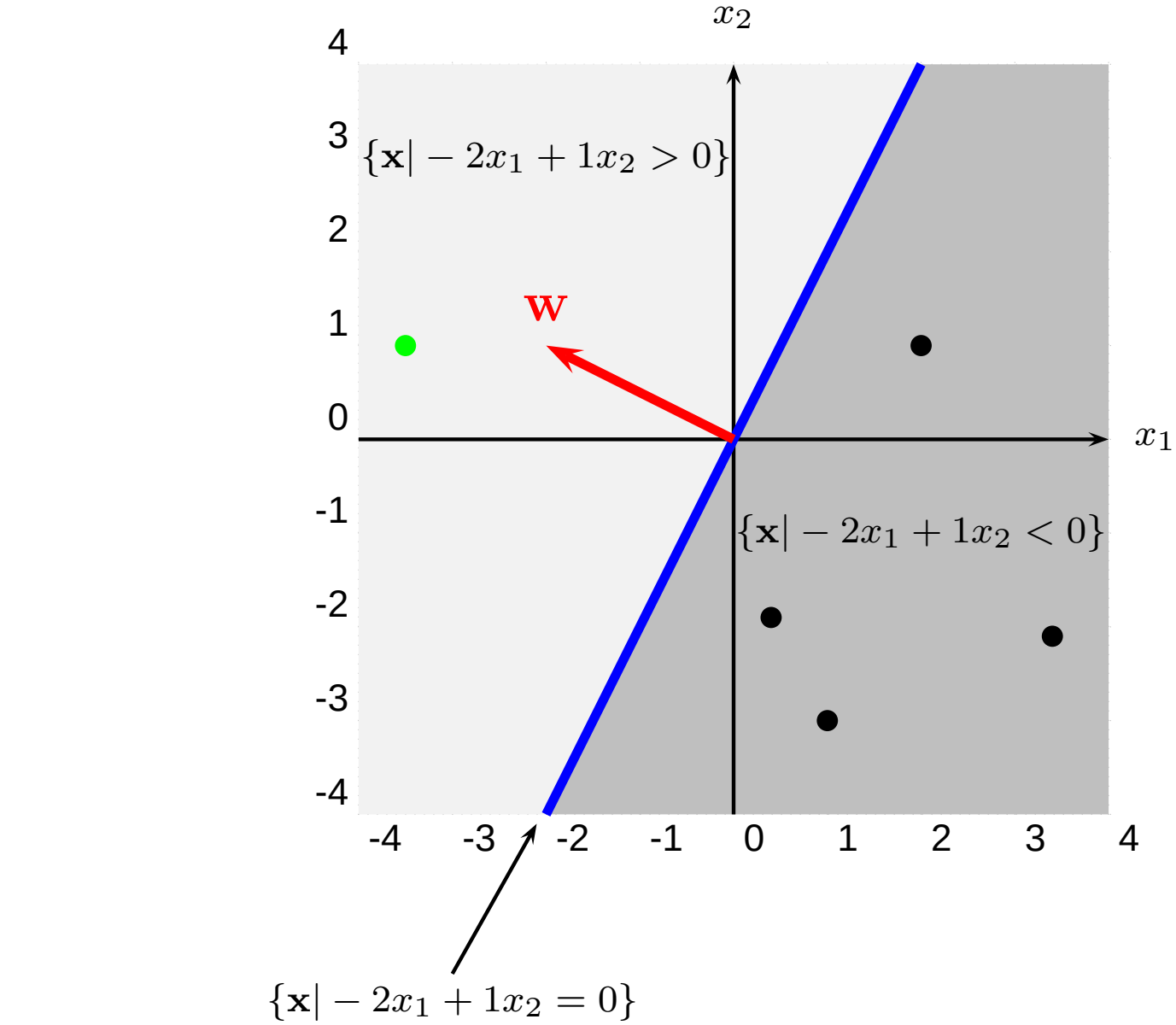
Linear Classifier & Dot Product (cont.)

Let us consider the *weight* vector $\mathbf{w} = (3, 0)$ and vector $\mathbf{x} = (2, 2)$.



Geometric interpretation of the dot product: Length of the projection of $\mathbf{x}$ onto the unit vector $\mathbf{w}/\|\mathbf{w}\|$.

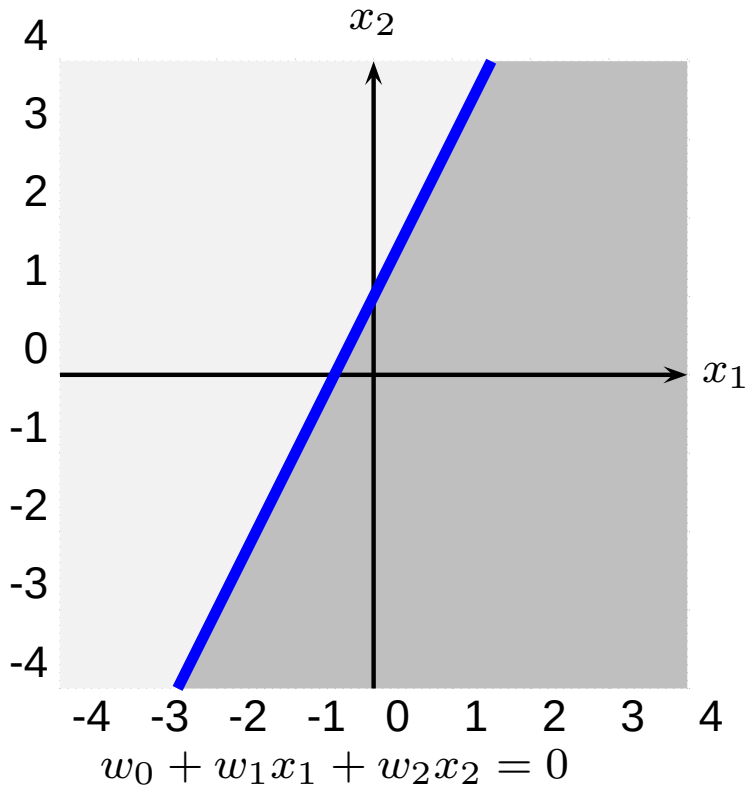
Linear Classifier & Two Half-Spaces



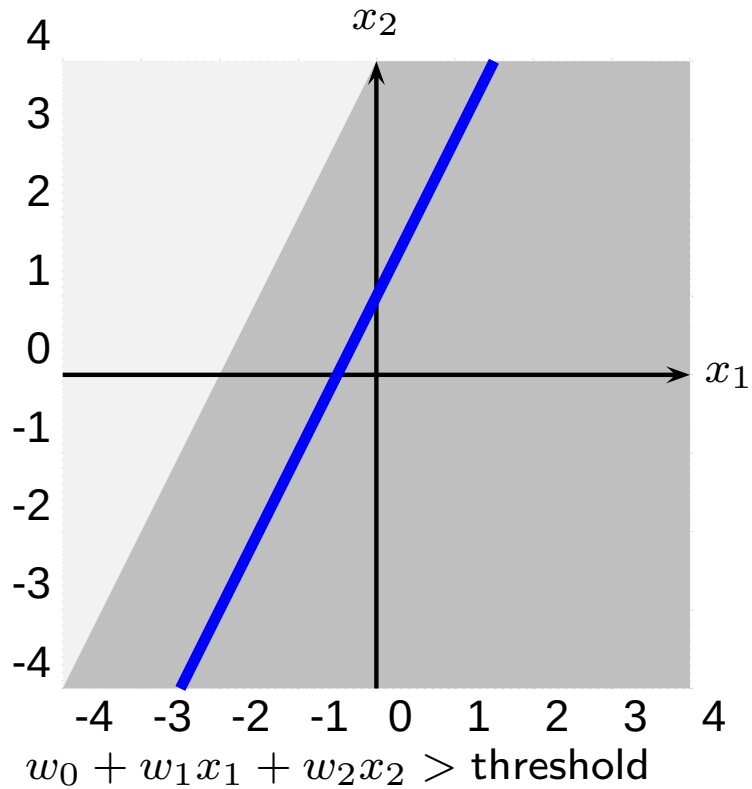
The $\mathbf{x}$-space is separated in two half-spaces.

Linear Classifier & Dot Product (cont.)

- Observe, that $w_1 x_1 + w_2 x_2 = 0$ implies, that the separating line always goes through the origin.
- By adding an offset (bias), that is $w_0 + w_1 x_1 + w_2 x_2 = 0 \Leftrightarrow x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2} \equiv y = mx + b$, one can shift the line arbitrary.

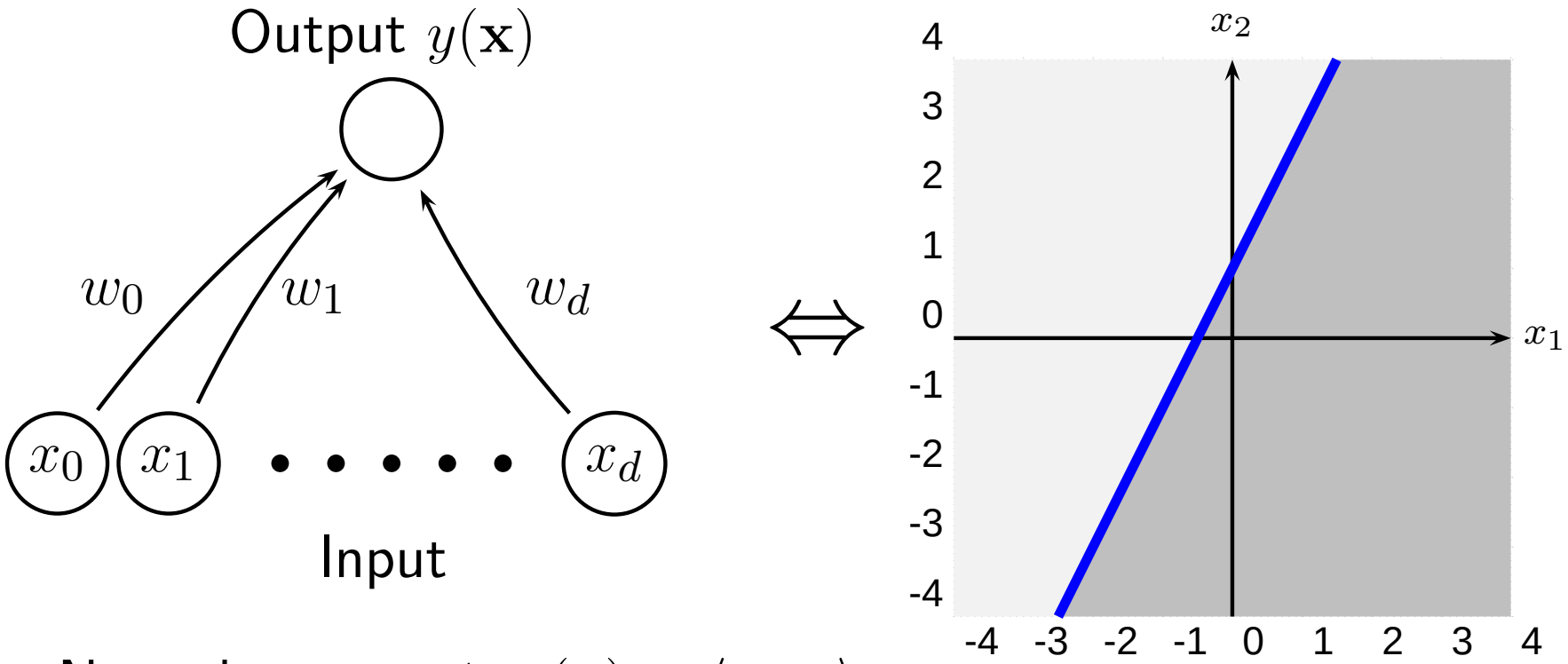


$w_0 + w_1 x_1 + w_2 x_2 = 0$



$w_0 + w_1 x_1 + w_2 x_2 >$ threshold

Linear Classifier & Single Layer NN



Note that $x_0 = 1$, $y(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + w_0$.

Given data which we want to separate, that is, a sample $\mathcal{X} = \{(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), \ldots, (\mathbf{x}_N, t_N)\} \in \mathbb{R}^d \times \{-1, +1\}$.

How to determine the proper values of $\mathbf{w}$ such that the "minus" and "plus" points are separated by $y(\mathbf{x})$? Infer the values of $\mathbf{w}$ from the data by some learning algorithm.

Perceptron

Note, so far we have not seen a method for finding the weight vector $\mathbf{w}$ to obtain a linearly separation of the training set.

Let $g(a)$ be (sign) activation function

$$g(a) = \begin{cases} -1 & \text{if } a < 0 \\ +1 & \text{if } a \geq 0 \end{cases}$$

and decision function

$$g(\langle \mathbf{w}, \mathbf{x} \rangle) = g\left(\sum_{i=0}^{d} w_i x_i \right).$$

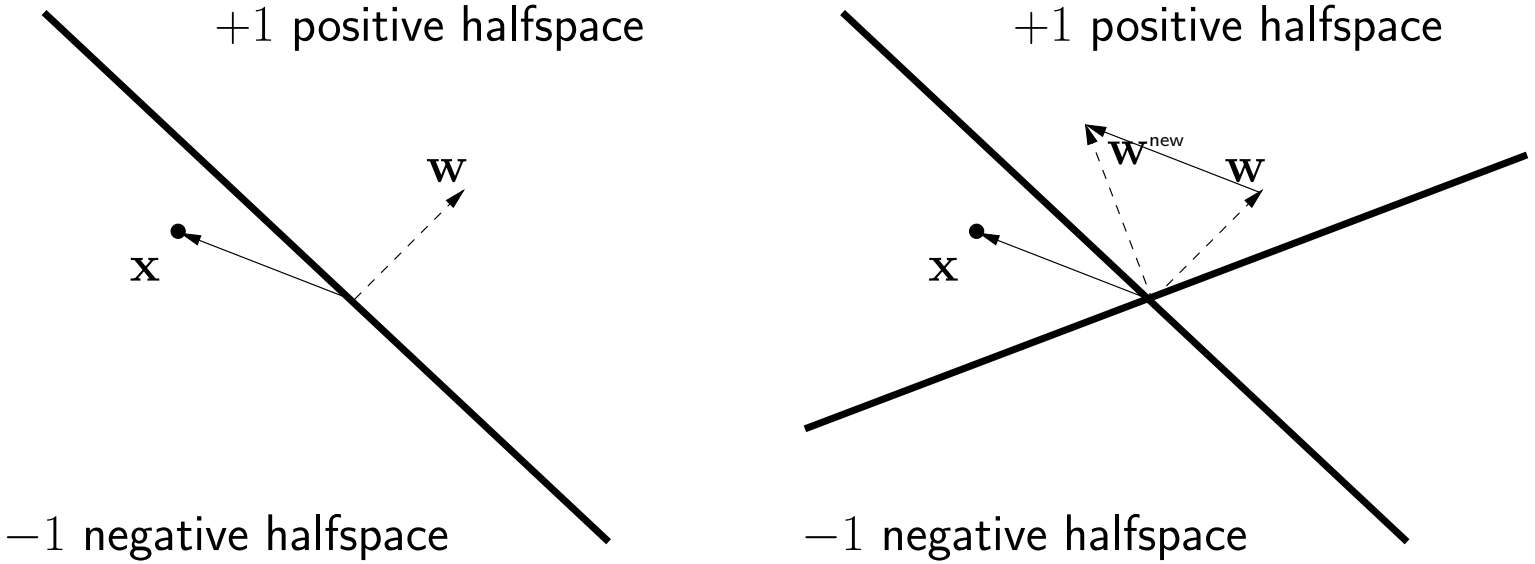
Note: x_0 is set to $+1$, that is, $\mathbf{x} = (1, x_1, \ldots, x_d)$. Training pattern consists of $(\mathbf{x}, t) \in \mathbb{R}^{d+1} \times \{-1, +1\}$

Perceptron Learning Algorithm

input : $(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N) \in \mathbb{R}^{d+1} \times \{-1, +1\}, \eta \in \mathbb{R}_+$, max.epoch $\in \mathbb{N}$
output: $\mathbf{w}$
begin
  Randomly initialize $\mathbf{w}$
  epoch $\leftarrow 0$
  repeat
    for $i \leftarrow 1$ **to** N **do**
      if $t_i \langle \mathbf{w}, \mathbf{x}_i \rangle \leq 0$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \eta \mathbf{x}_i \, t_i$
    epoch $\leftarrow$ epoch $+ 1$
  until (*epoch* = *max.epoch*) *or* (*no change in* $\mathbf{w}$)
  return $\mathbf{w}$
end

Training the Perceptron (cont.)

Geometrical explanation: If $\mathbf{x}$ belongs to $\{+1\}$ and $\langle \mathbf{w}, \mathbf{x} \rangle < 0 \Rightarrow$ angle between $\mathbf{x}$ and $\mathbf{w}$ is greater than $90°$, rotate $\mathbf{w}$ in direction of $\mathbf{x}$ to bring missclassified $\mathbf{x}$ into the positive half space defined by $\mathbf{w}$. Same idea if $\mathbf{x}$ belongs to $\{-1\}$ and $\langle \mathbf{w}, \mathbf{x} \rangle \geq 0$.



Perceptron Error Reduction

Recall: missclassifcation results in:

$$\mathbf{w}^{\text{new}} = \mathbf{w} + \eta \mathbf{x}\, t,$$

this reduces the error since

$$\begin{aligned} -\mathbf{w}^{\text{new}}(\mathbf{x}\, t)^T &= -\mathbf{w}(\mathbf{x}\, t)^T - \underbrace{\eta}_{>0} \underbrace{(\mathbf{x}\, t)(\mathbf{x}\, t)^T}_{\|\mathbf{x}t\|^2>0} \\ &< -\mathbf{w}^T \mathbf{x} t \end{aligned}$$

How often one has to cycle through the patterns in the training set?

- A finite number of steps?

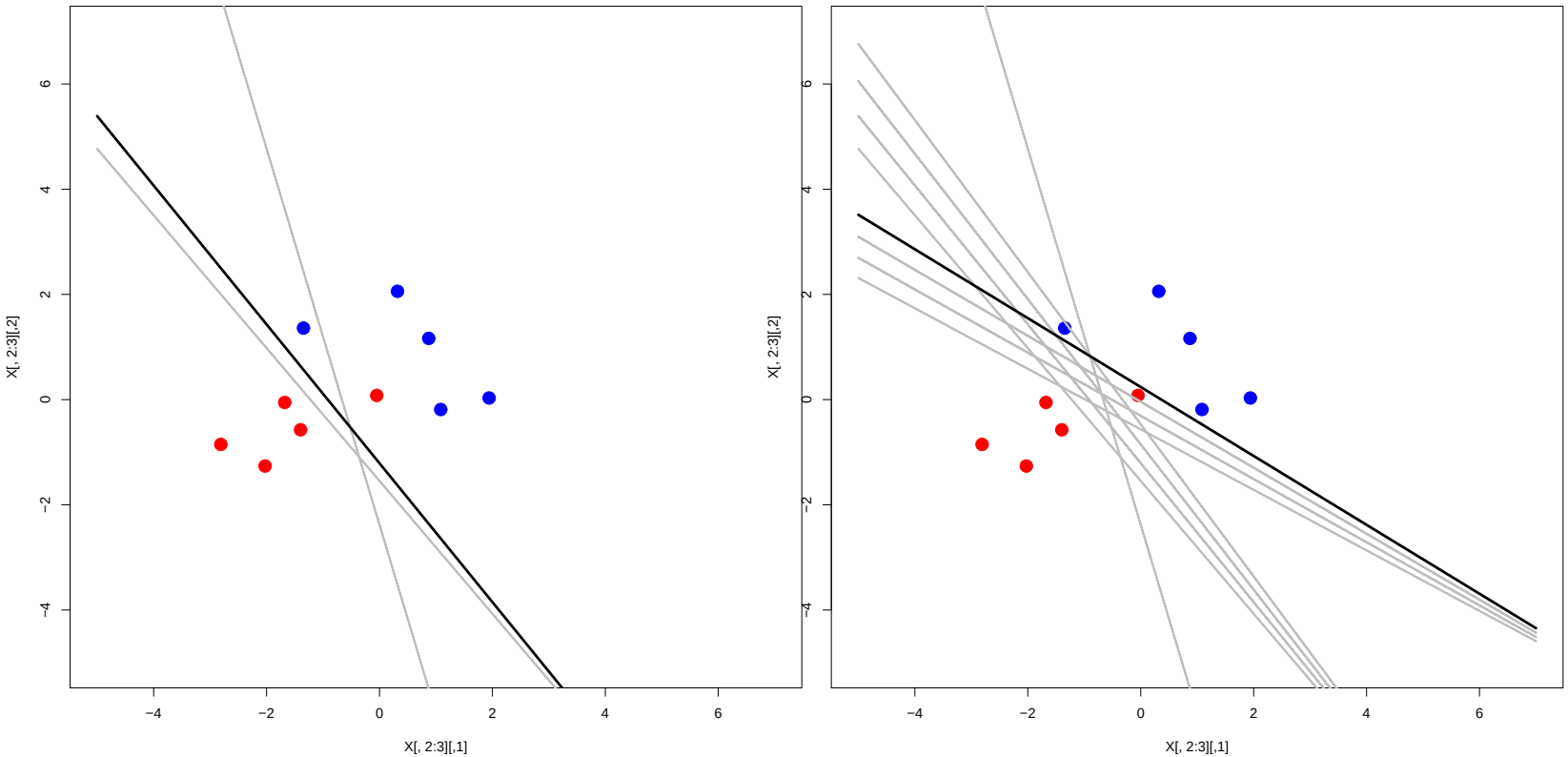
Perceptron Convergence Theorem

Proposition 1 *Given a finite and linearly separable training set. The perceptron converges after some finite steps [Rosenblatt, 1962].*

Perceptron Algorithm (R-code)

```
#####################################################
perceptron <- function(w,X,t,eta,max.epoch) {
#####################################################
  N <- nrow(X)/2;
  epoch <- 0;
  repeat {
    w.old <- w;
    for (i in 1:(2*N)) {
      if ( t[i]*y(X[i,],w) <= 0 )
        w <- w + eta * t[i] * X[i,];
    }
    epoch <- epoch + 1;
    if ( identical(w.old,w) || epoch = max.epoch ) {
      break; # terminate if no change in weights or max.epoch reached
    }
  }
  return (w);
}
```

Perceptron Algorithm Visualization



One epoch

terminate if no change in $\mathbf{w}$