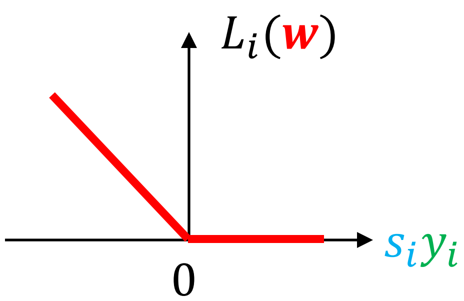
Loss function for perceptron

- "Training": finds weights to minimise some loss. Which?
- Our task is binary classification. Encode one class as $+1$ and the other as -1. So each training example is now $(\boldsymbol{x}_i, y_i)$, where y_i is either $+1$ or -1
- Recall that, in a perceptron, $s_i = \boldsymbol{w}'\boldsymbol{x}_i = \sum_{j=0}^{m} w_j x_{ij}$, and the sign of s_i determines the predicted class: $+1$ if $s_i > 0$, and -1 if $s_i < 0$
- Consider a single training example.
  * If y_i and s_i have same sign then the example is classified correctly.
  * If y_i and s_i have different signs, the example is misclassified

Loss function for perceptron

- The perceptron uses a loss function in which there is no penalty for correctly classified examples, while the penalty (loss) is equal to s_i for misclassified examples*
- Formally:
  - $L_i(\boldsymbol{w}) = 0$ if both s_i, y_i have the same sign
  - $L_i(\boldsymbol{w}) = |s_i|$ if both s_i, y_i have different signs
- This can be re-written as $L_i(\boldsymbol{w}) = \max(0, -s_i y_i)$



$s_i = \boldsymbol{w}'\boldsymbol{x}_i$

* This is similar, but not identical to the SVM's ***hinge*** loss

Stochastic gradient descent

- Randomly shuffle/split all training examples in B batches
- Choose initial $\boldsymbol{\theta}^{(1)}$
- For t from 1 to T
- For b from 1 to B
- Do gradient descent update using data from batch b

Iterations over the entire dataset are called *epochs*

- Advantage of such an approach: computational feasibility for large datasets

Perceptron training algorithm

Choose initial guess $\boldsymbol{w}^{(0)}$, $k = 0$

For t from 1 to T (epochs)

For i from 1 to N (training examples)

Consider example $(\boldsymbol{x}_i, y_i)$

Update*: $\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta \boldsymbol{\nabla} L_i\left(\boldsymbol{w}^{(k)}\right)$

$k = k + 1$

$L_i(\boldsymbol{w}) = \max(0, -s_i y_i)$

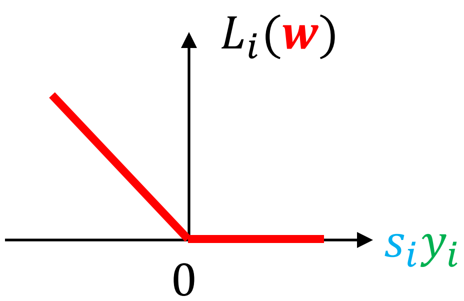
$s_i = \boldsymbol{w}'\boldsymbol{x}_i$

η is learning rate

*There is no derivative when $s_i = 0$, but this case is handled explicitly in the algorithm, see next slides

Perceptron training rule

- We have $\nabla L_i(\boldsymbol{w}) = \mathbf{0}$ when $s_i y_i > 0$
  - We don't need to do update when sample i is correctly classified
- What is $\nabla L_i(\boldsymbol{w})$ when $s_i y_i < 0$?
  - We need to update when sample i is misclassified
  - We have $\nabla L_i(\boldsymbol{w}) = \boldsymbol{x}_i$ when $y_i = -1$ and $s_i > 0$
  - We have $\nabla L_i(\boldsymbol{w}) = -\boldsymbol{x}_i$ when $y_i = 1$ and $s_i < 0$
  - Thus $\nabla L_i(\boldsymbol{w}) = -y_i \boldsymbol{x}_i$



$s_i = \boldsymbol{w}'\boldsymbol{x}_i$

- $L_i(\boldsymbol{w}) = \max(0, -s_i y_i)$

Perceptron training algorithm

Choose initial guess $\boldsymbol{w}^{(0)}$, $k = 0$

For t from 1 to T (epochs)

For i from 1 to N (training examples)

Compute $s_i = \left(\boldsymbol{w}^{(k)}\right)' \boldsymbol{x}_i$

If $s_i y_i \leq 0$: (sample i misclassified)

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \eta y_i \boldsymbol{x}_i$$

$$k = k + 1$$

($\eta > 0$ is called *learning rate)*

Perceptron training algorithm

Choose initial guess $\boldsymbol{w}^{(0)}$, $k = 0$

For t from 1 to T (epochs)

For i from 1 to N (training examples)

Compute $s_i = \left(\boldsymbol{w}^{(k)}\right)' \boldsymbol{x}_i$

If $s_i y_i \leq 0$: (sample i misclassified)

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \eta y_i \boldsymbol{x}_i$$

$$k = k + 1$$

Strictly speaking it should be $s_i y_i < 0$ but $\leq$ allows handling the case $\boldsymbol{w}^{(k)} = \boldsymbol{0}$

$\boldsymbol{w}^{(k)}$ represents the value of $\boldsymbol{w}$ after k updates (useful for theory). If you implement this, just write: $\boldsymbol{w} = \boldsymbol{w} + \eta y_i \boldsymbol{x}_i$

Perceptron training algorithm

Choose initial guess $\boldsymbol{w}^{(0)}$, $k = 0$

For t from 1 to T (epochs)

For i from 1 to N (training examples)

Compute $s_i = \left(\boldsymbol{w}^{(k)}\right)' \boldsymbol{x}_i$

If $s_i y_i \leq 0$: (sample i misclassified)

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \eta y_i \boldsymbol{x}_i$$

$$k = k + 1$$

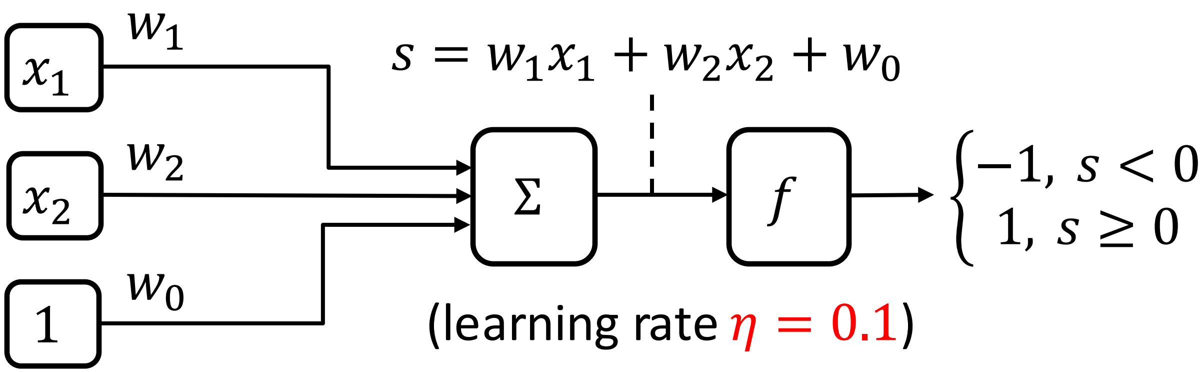
Convergence Theorem: if the training data is linearly separable, the algorithm is guaranteed to converge to a solution. That is, there exist a finite k such that $L\left(\boldsymbol{w}^{(k)}\right) = 0$

Pros and cons of perceptron learning

- If the data is linearly separable, the perceptron training algorithm will converge to a correct solution
  - There is a formal proof ← good!
  - It will converge to some solution (separating boundary), one of infinitely many possible ← bad!
- However, if the data is not linearly separable, the training will fail completely rather than give some approximate solution
  - Ugly ☹

Example 2: Perceptron learning

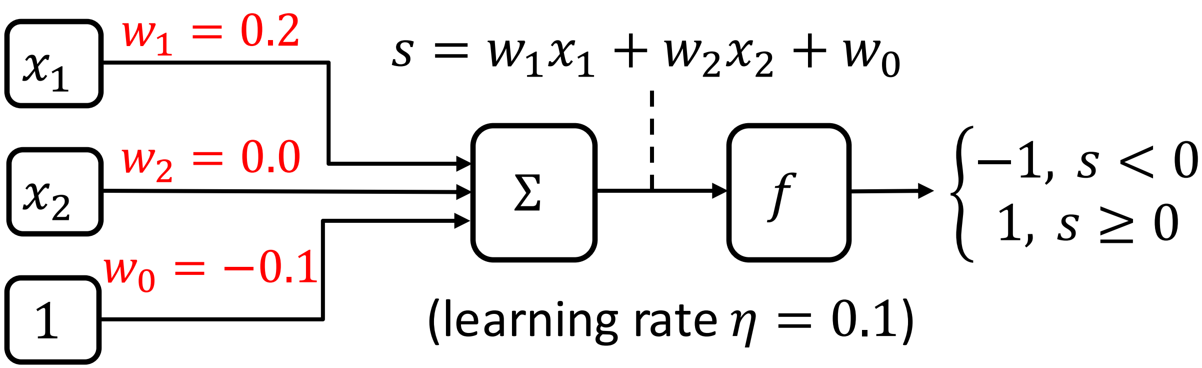
Basic setup

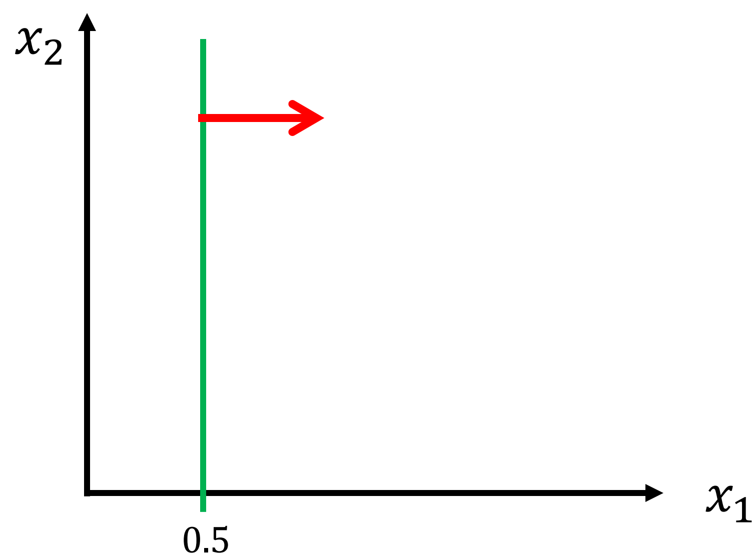


* We drop the sample index i to have a simpler notation.

Example 2: Perceptron learning

Start with random weights

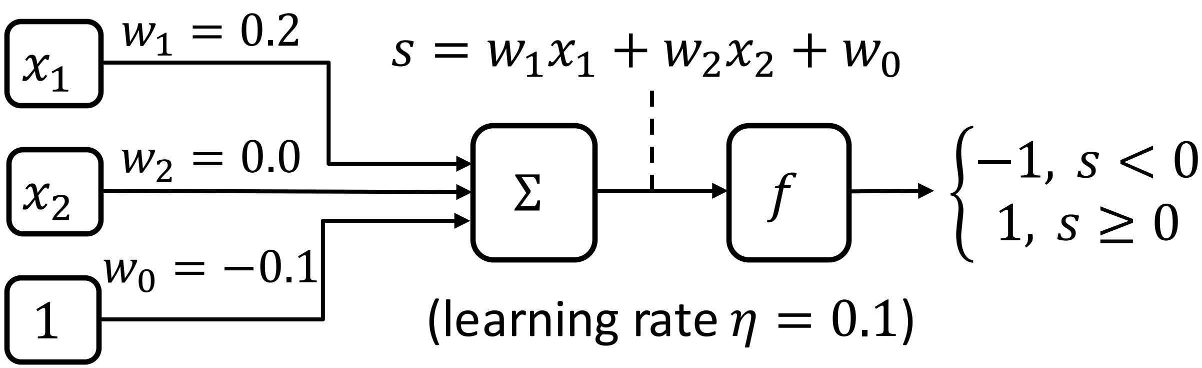




* We drop the sample index i to have a simpler notation.

Example 2: Perceptron learning

Consider training example 1

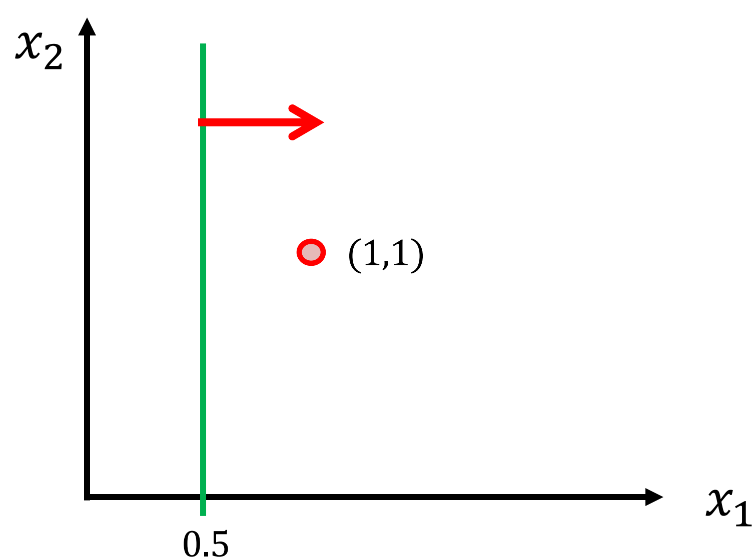


class -1

class 1

$$y(0.2x_1 + 0.0x_2 - 0.1) = -0.1 \leq 0$$

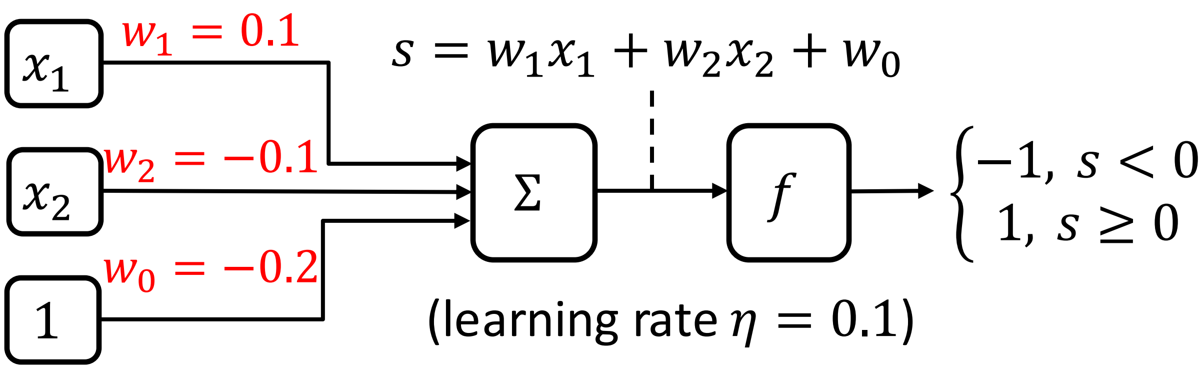
$$\begin{aligned} w_1 &\leftarrow w_1 - \eta x_1 = 0.1 \\ w_2 &\leftarrow w_2 - \eta x_2 = -0.1 \\ w_0 &\leftarrow w_0 - \eta \quad = -0.2 \end{aligned}$$



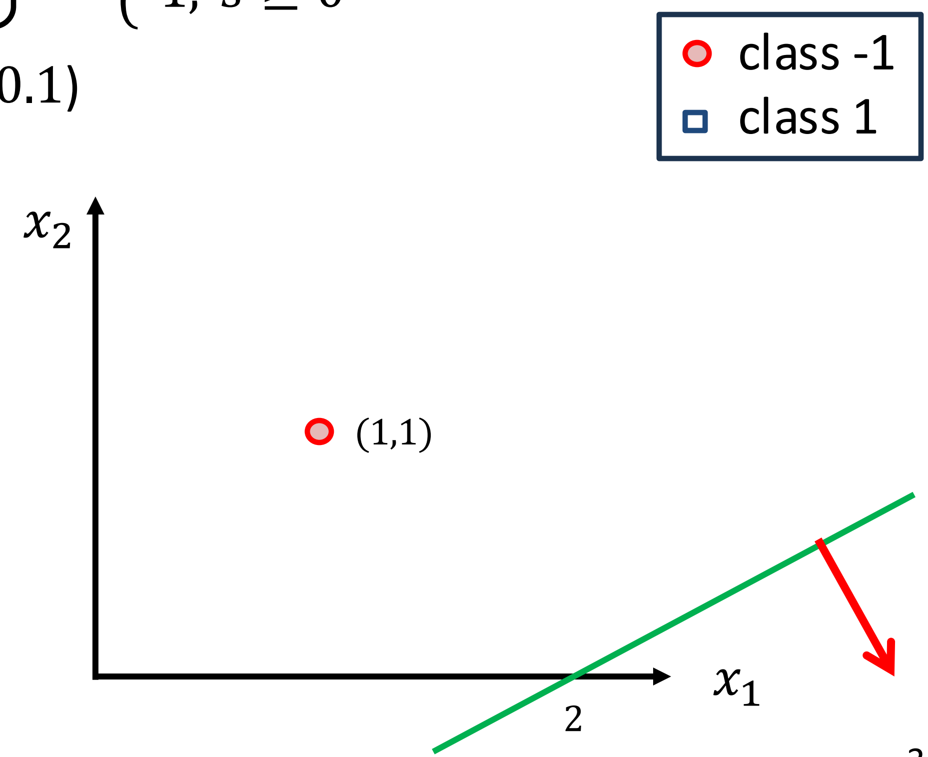
* We drop the sample index i to have a simpler notation.

Example 2: Perceptron learning

Update weights



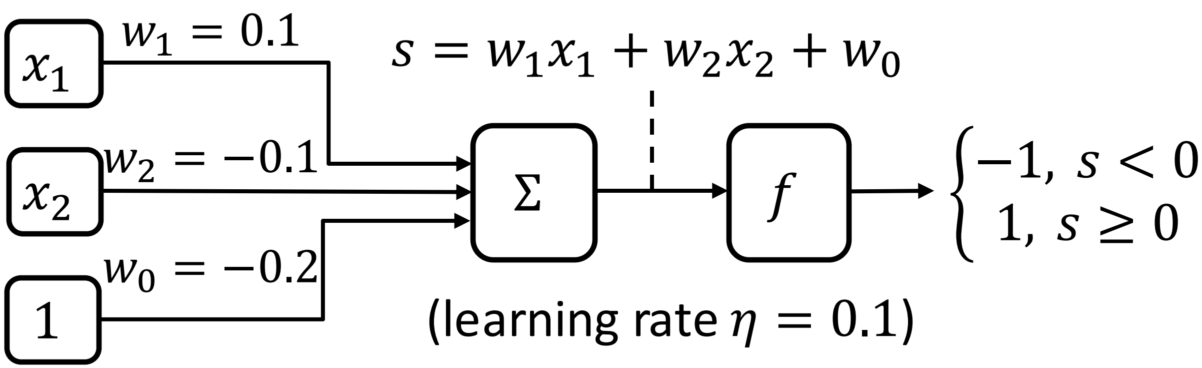
(learning rate $\eta = 0.1$)



* We drop the sample index i to have a simpler notation.

Example 2: Perceptron learning

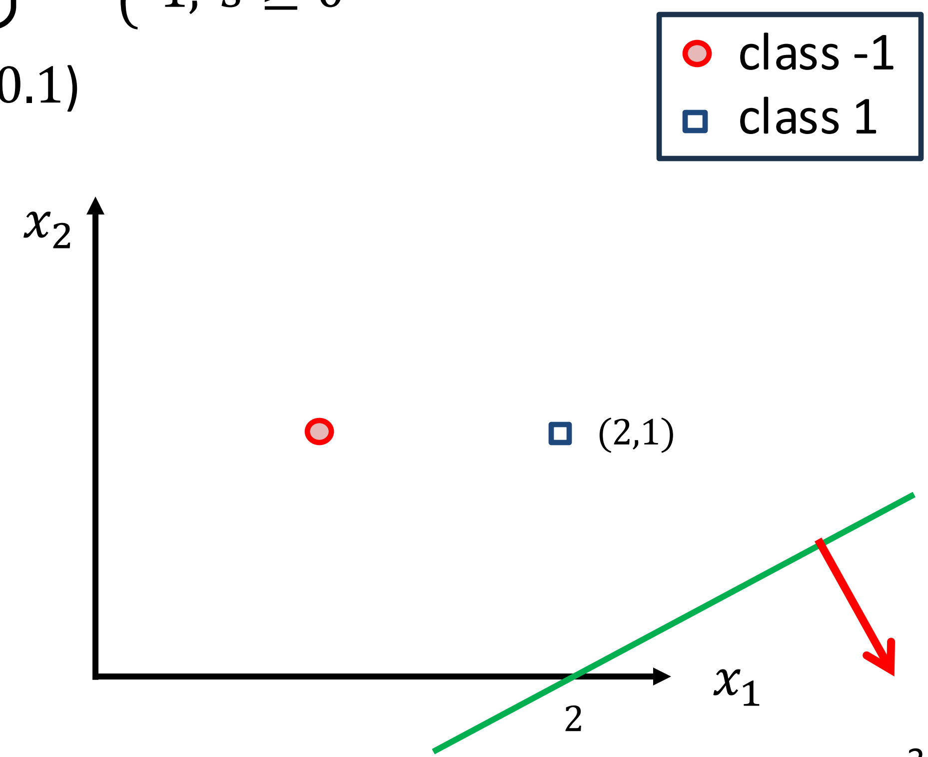
Consider training example 2



(learning rate $\eta = 0.1$)

$$y(0.1x_1 - 0.1x_2 - 0.2) = -0.1 \leq 0$$

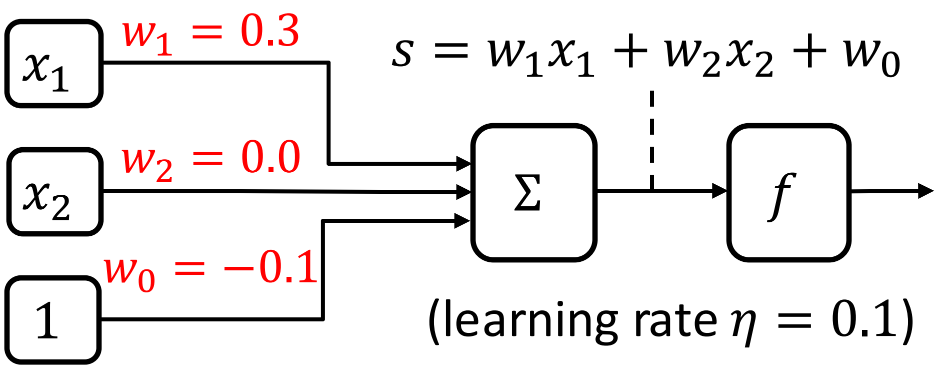
$$\begin{aligned} w_1 &\leftarrow w_1 + \eta x_1 = 0.3 \\ w_2 &\leftarrow w_2 + \eta x_2 = 0.0 \\ w_0 &\leftarrow w_0 + \eta \quad = -0.1 \end{aligned}$$



* We drop the sample index i to have a simpler notation.

Example 2: Perceptron learning

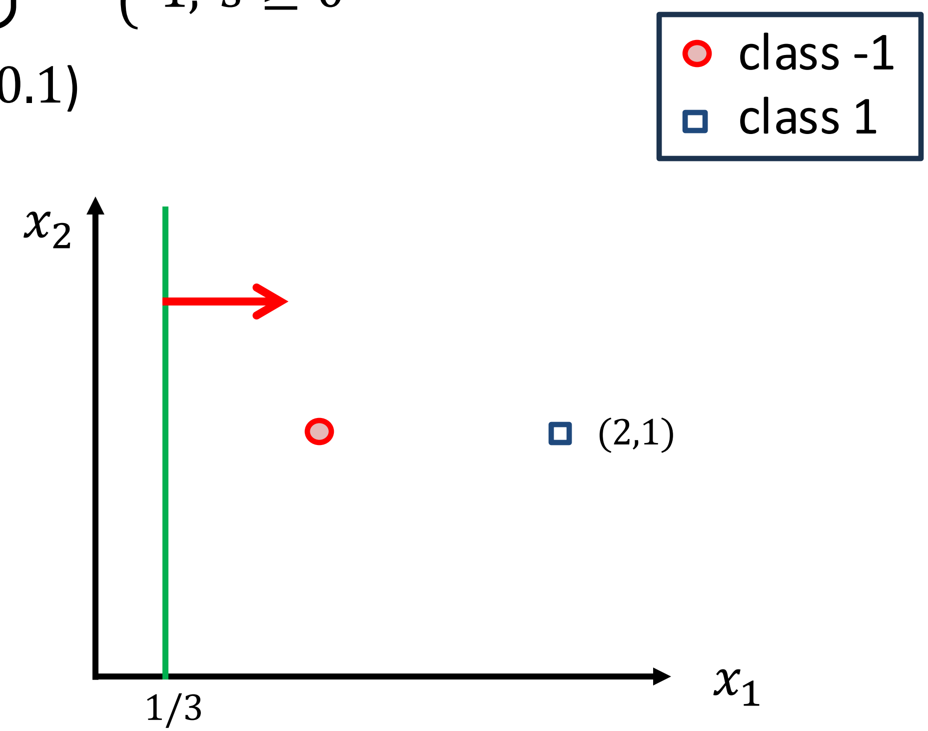
Update weights



$$s = w_1 x_1 + w_2 x_2 + w_0$$

$$\begin{cases} -1, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

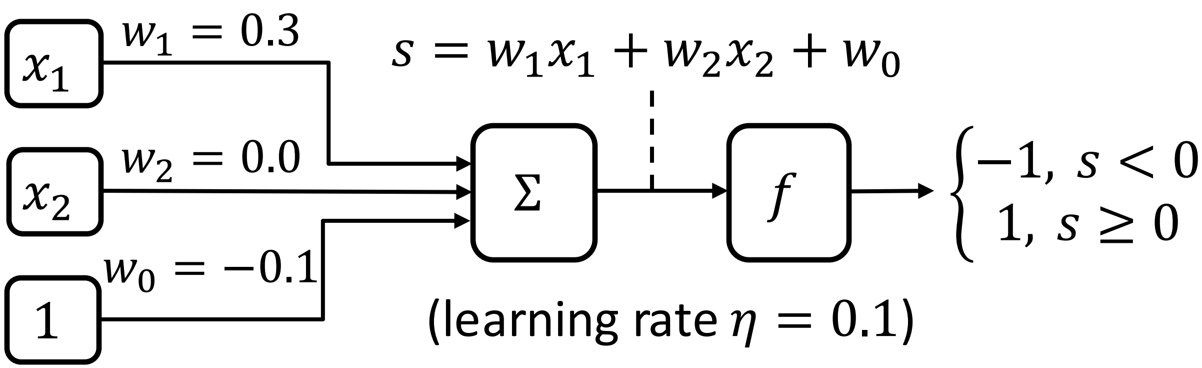
(learning rate $\eta = 0.1$)

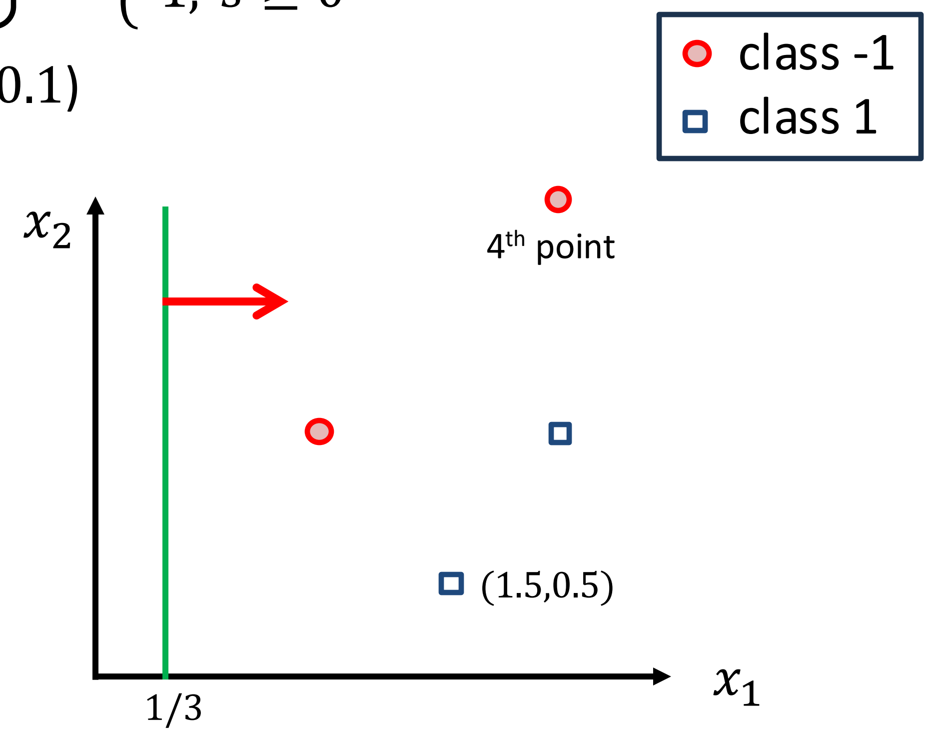


* We drop the sample index i to have a simpler notation.

Example 2: Perceptron learning

Further examples





$$y(0.3x_1 - 0.0x_2 - 0.1) = 0.35 > 0$$

3^{rd} point: correctly classified

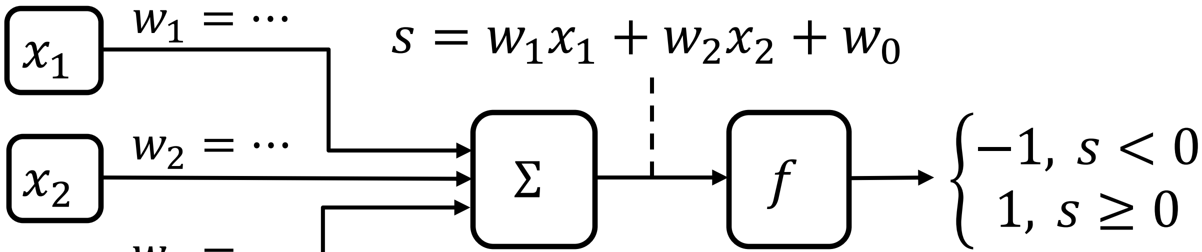
4^{th} point: incorrect, update

etc.

* We drop the sample index i to have a simpler notation.

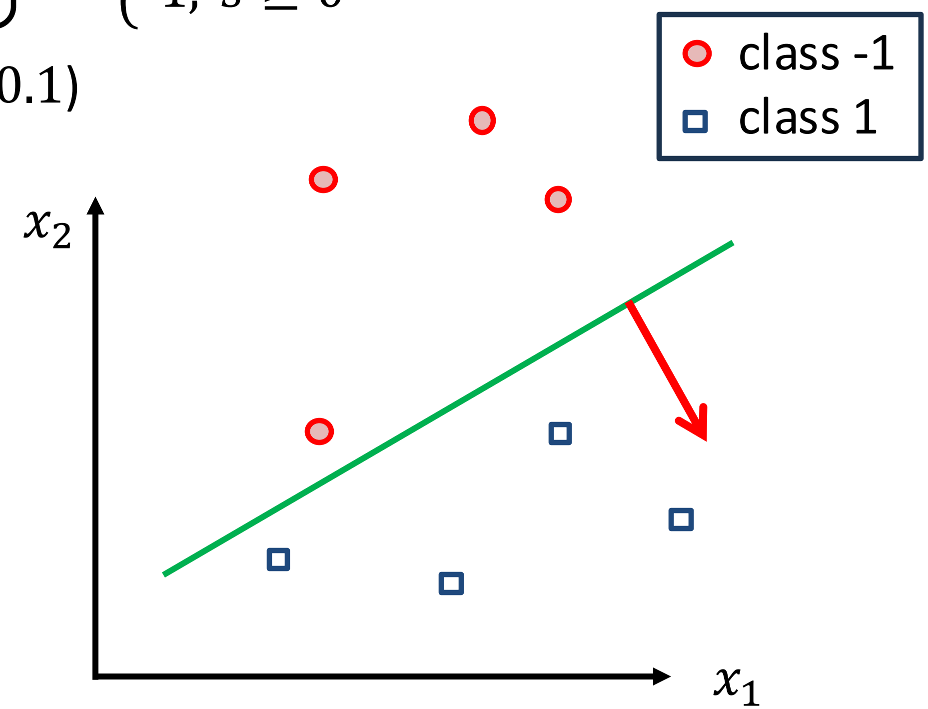
Example 2: Perceptron learning

Further examples



(learning rate $\eta = 0.1$)

Eventually, all the data will be correctly classified (provided it is linearly separable)



* We drop the sample index i to have a simpler notation.

Kernel Perceptron

Another example of a kernelizable learning algorithm (like the SVM).

Perceptron training rule: Recap

Compute $s_i = \left(\boldsymbol{w}^{(k)}\right)' \boldsymbol{x}_i$

If $s_i y_i \leq 0$: (sample i misclassified)

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \eta y_i \boldsymbol{x}_i$$

$$k = k + 1$$

Suppose weights are initially set to $\boldsymbol{w}^{(0)} = \mathbf{0}$

Suppose the algorithm misclassifies sample 1, 7, 29, and 1 again

First update: $\boldsymbol{w}^{(1)} = \eta y_1 \boldsymbol{x}_1$

Second update: $\boldsymbol{w}^{(2)} = \eta y_1 \boldsymbol{x}_1 + \eta y_7 \boldsymbol{x}_7$

Third update: $\boldsymbol{w}^{(3)} = \eta y_1 \boldsymbol{x}_1 + \eta y_7 \boldsymbol{x}_7 + \eta y_{29} \boldsymbol{x}_{29}$

Third update: $\boldsymbol{w}^{(4)} = 2\eta y_1 \boldsymbol{x}_1 + \eta y_7 \boldsymbol{x}_7 + \eta y_{29} \boldsymbol{x}_{29}$ etc.

Accumulating updates: Data enters via dot products

- Weights always take the form $\boldsymbol{w} = \sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j$, where $\boldsymbol{\alpha}$ some coefficients
- Perceptron weights always linear comb. of data!
- Recall that prediction for a new point $\boldsymbol{x}$ is based on sign of $\boldsymbol{w}'\boldsymbol{x}$
- Substituting $\boldsymbol{w}$ we get $\boldsymbol{w}'\boldsymbol{x} = \sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j'\boldsymbol{x}$
- The dot product $\boldsymbol{x}_j'\boldsymbol{x}$ can be replaced with a kernel

Kernelised perceptron training rule

Set $\boldsymbol{\alpha} = \mathbf{0}$

For t from 1 to T (epochs)

For i from 1 to N (training examples)

Compute $s_i = \sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j' \boldsymbol{x}_i$

If $s_i y_i \leq 0$: (sample i misclassified)

$\alpha_i \leftarrow \alpha_i + \eta$

($\eta > 0$ is called *learning rate)*

Kernelised perceptron training rule

Set $\boldsymbol{\alpha} = \mathbf{0}$

For t from 1 to T (epochs)

For i from 1 to N (training examples)

Compute $s_i = \sum_{j=1}^{N} \alpha_j y_j K(\boldsymbol{x}_j, \boldsymbol{x}_i)$

If $s_i y_i \leq 0$: (sample i misclassified)

$\alpha_i \leftarrow \alpha_i + \eta$