



Lecture 3

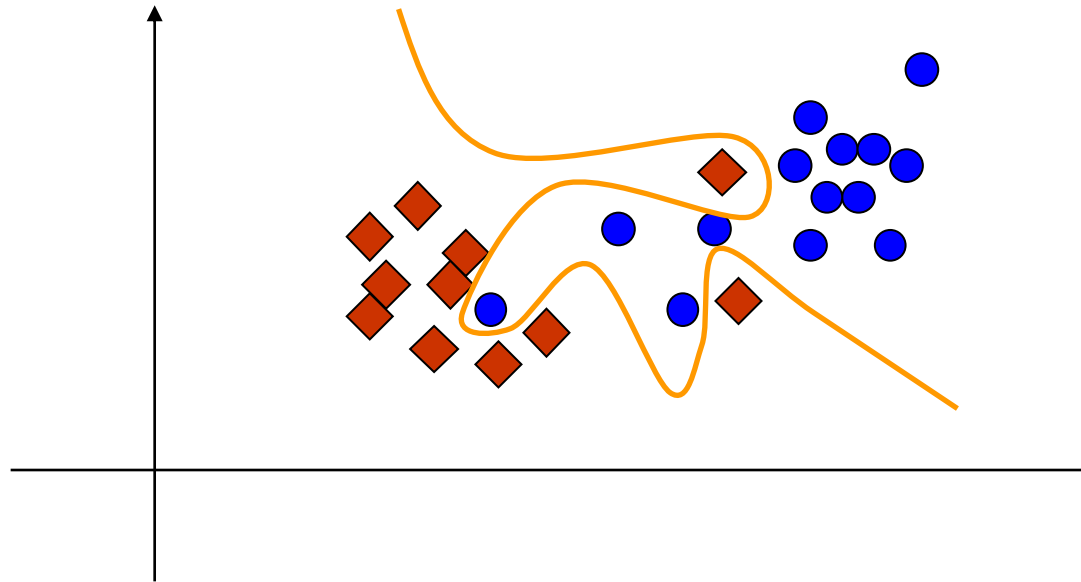
Optimisation and the Perceptron

Gabriele Pergola and Amina Asif
Machine Learning Algorithms and Practice
CS429

October 2026

Classification Approaches: A Linear Classifier?

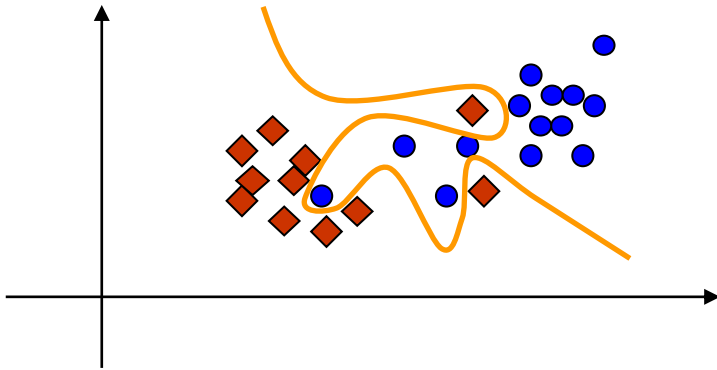
- Does k-NN classify with a straight line? (is it a linear classifier?)
- Nonlinear Classification boundary



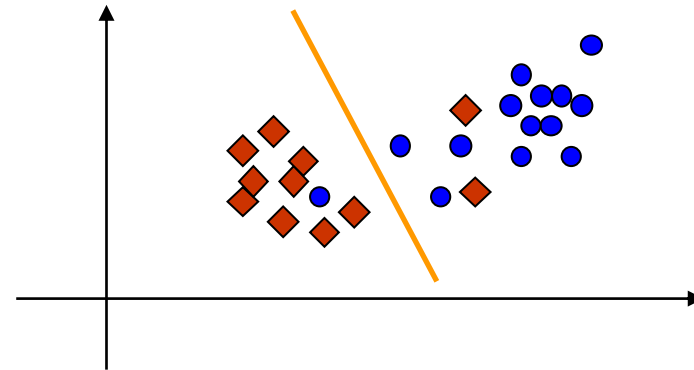
Classification Approaches: Linear Classifier

- **Generalization and Memorization**

- Remembering everything is not learning
- The true test of learning is handling similar but unseen cases, i.e., Generalization



Has great memorization but may generalize poorly (under certain assumptions)



Has lesser memorization but may generalize better (under certain assumptions)

Discriminant based Classification: From Class Scores to a Decision Boundary

A (set of) **discriminant function** f_m assigns a score to each class m :

$$f_1(x), f_2(x), \dots, f_M(x)$$

We pick the the class with the highest score:

$$\hat{y} = \operatorname{argmax}_{k=1,\dots,M} f_k(x)$$

where $f_k(x)$ is the **score** for **class** k .

For **binary classification** ($M = 2$): choose **class 1** if $f_1(x) \geq f_2(x)$

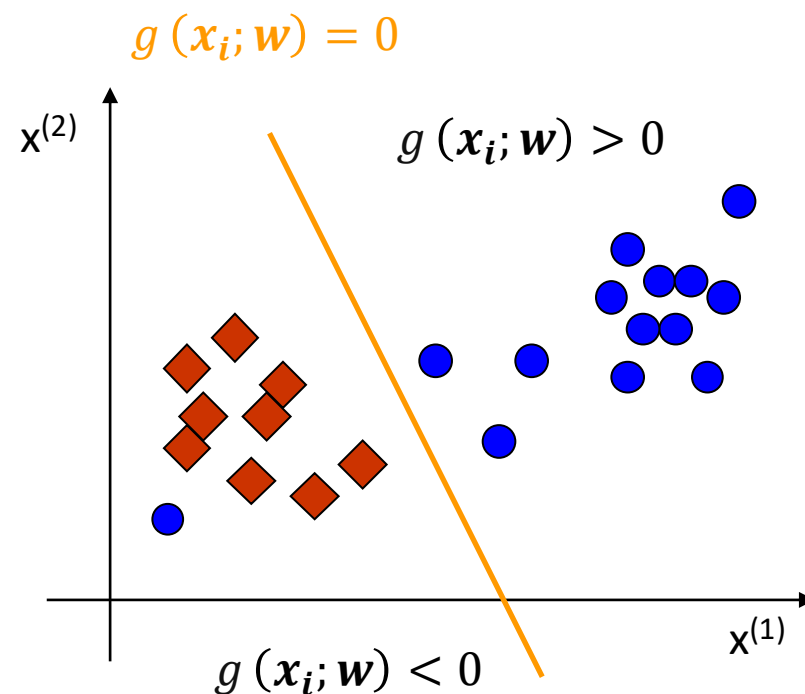
Then, we can define a single discriminant function:

$$g(x) = f_1(x) - f_2(x)$$

Thus, if $g(x) \geq 0 \Rightarrow$ Class: **+1**, $g(x) < 0 \Rightarrow$ Class: **-1**

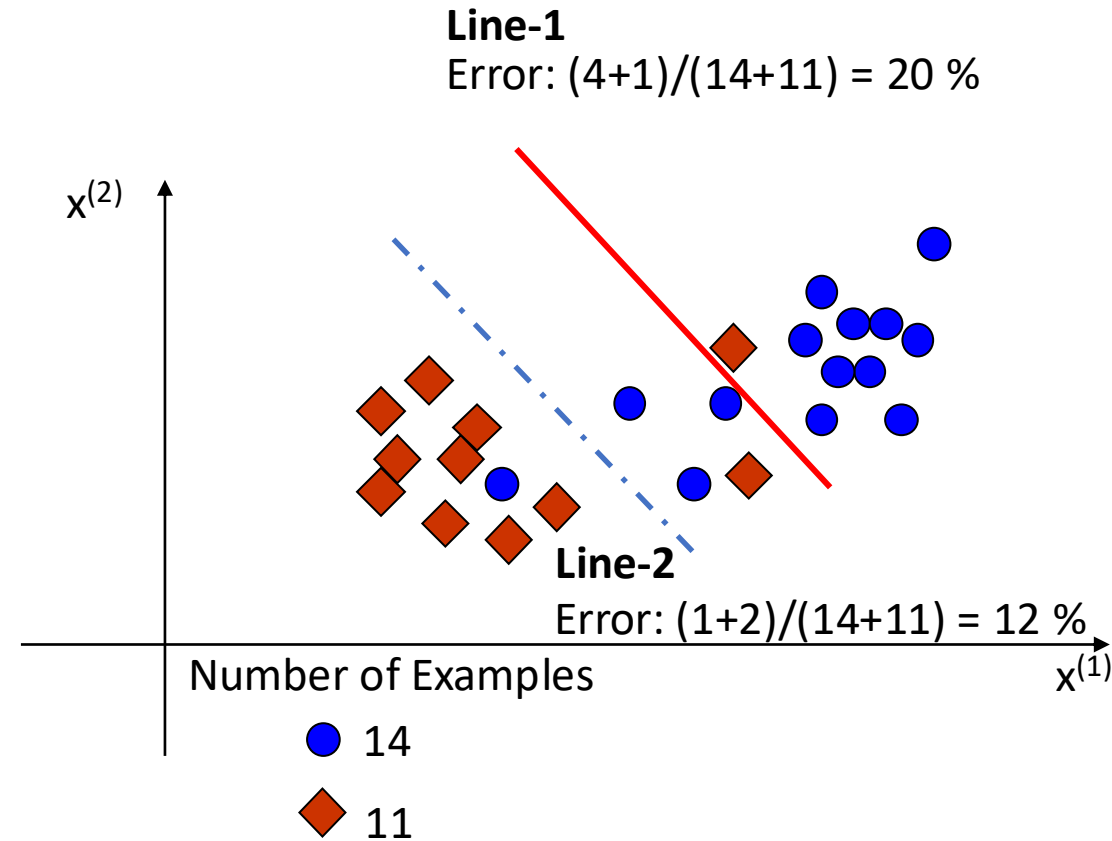
The points satisfying $g(x) = 0$ form the **decision boundary**.

If $g(x) = w^T x + b$, the decision boundary is linear, thus this is **linear classifier**.



REO: A general framework to look at Linear Classification

- We would like to minimize the number of errors a discriminant function $f(x)$ makes
- **Representation:**
Assume we look at only linear functions
$$f(x; w) = w_0 + w_1x^{(1)} + w_2x^{(2)} + \dots + w_dx^{(d)} = 0$$
- **Evaluation:**
We need to define error that a particular $f(x; w)$ makes
- **Optimization:**
We need to minimize the error by tuning w



Building Linear Discriminants

- Representation

- Features
- Linear Function

$$f(\mathbf{x}; \mathbf{w}) = w_1x^{(1)} + w_2x^{(2)} + \dots + w_dx^{(d)} + b = 0$$

- Evaluation

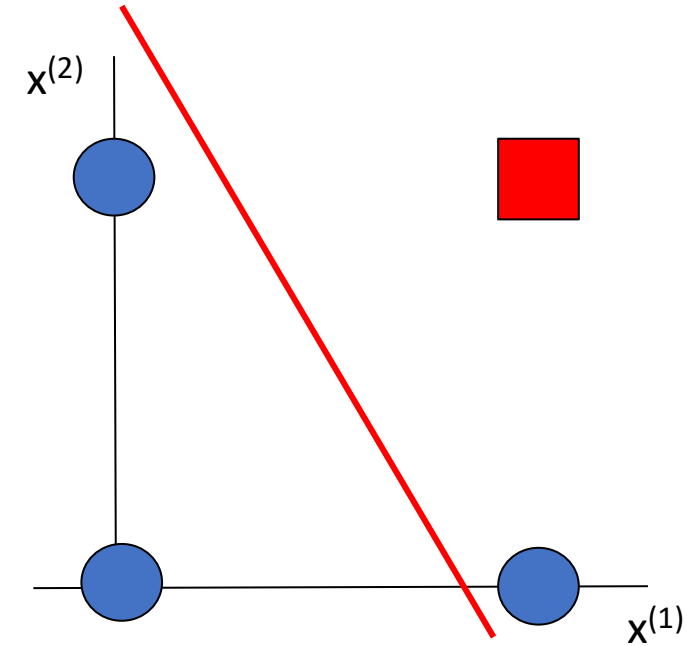
- Misclassification Error

?

- Optimization

- Find a line that minimizes misclassifications
- How done: Visual reckoning / Constraint Satisfaction

?



$$(1,1): w_1(1.0) + w_2(1.0) + b > 0$$

$$(1,0): w_1(1.0) + w_2(0.0) + b < 0$$

$$(0,1): w_1(0.0) + w_2(1.0) + b < 0$$

$$(0,0): w_1(0.0) + w_2(0.0) + b < 0$$

REO: Building Linear Discriminants

REO: **Building Linear Discriminants** **Representation**

Representation: A Linear Discriminant

A binary classifier can assign each example a real-valued score:

$$f(x) = w^T x + b$$

For d features:

$$f(x) = w_1 x_1 + w_2 x_2 + \dots + w_d x_d + b = 0$$

Classification rule

$$f(x) > 0 \Rightarrow \text{Class } +1$$

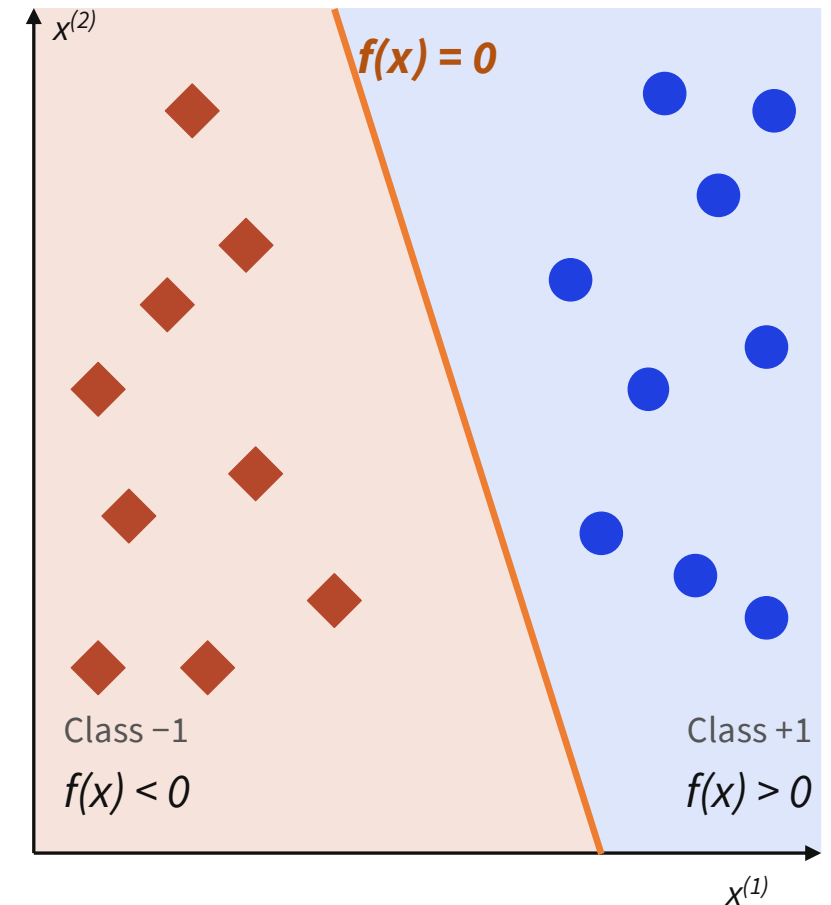
$$f(x) < 0 \Rightarrow \text{Class } -1$$

The points where $f(x) = 0$ form the **decision boundary**.

Geometry:

For two features, $w_1 x_1 + w_2 x_2 + b = 0$ is a line.

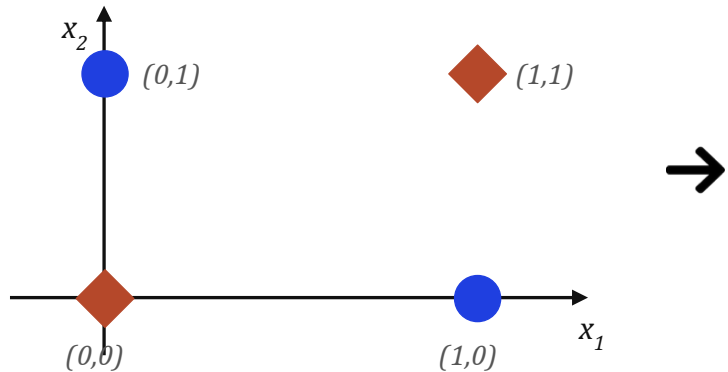
More generally, in d dimensions it is a **hyperplane**.



A linear classifier partitions the feature space using a hyperplane.

When a Linear Boundary Is Not Enough

What if the classes cannot be separated by a straight line in the current feature space?

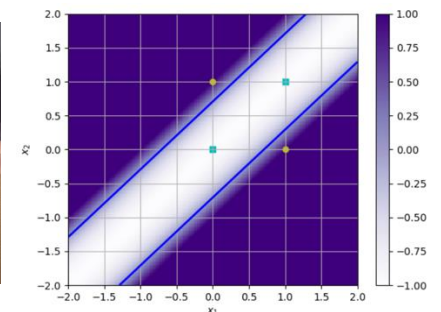


Fold-and-Cut Theorem: a complex separation in the original space may become a simple straight cut after the space is transformed.

- Notice: folding changes the distance between points
- How to achieve such folding?
 - One way is to **transform** the data

Thus, it is theoretically possible to **partition any space into regions containing positive and negative training examples** no matter how complex such a boundary is by simply folding the feature space appropriately and using a linear classifier (single straight cut).

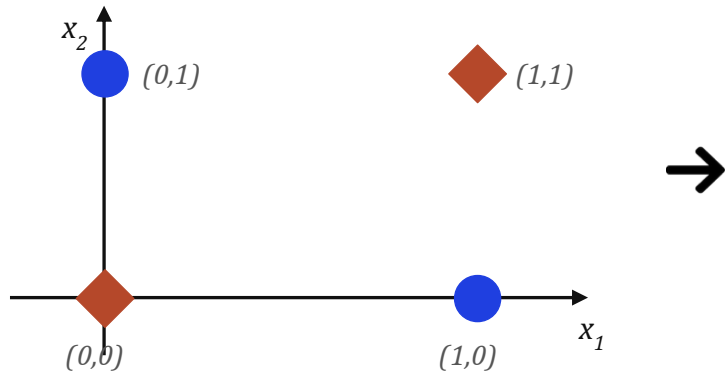
https://en.wikipedia.org/wiki/Fold-and-cut_theorem



<https://www.youtube.com/watch?v=ZREp1mAPKTM>

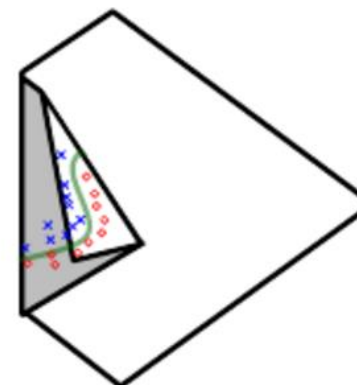
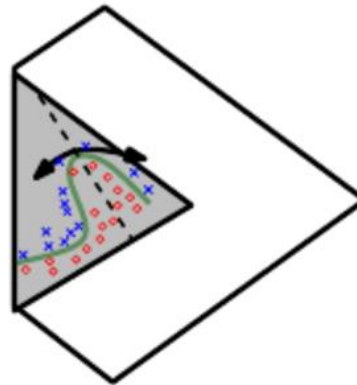
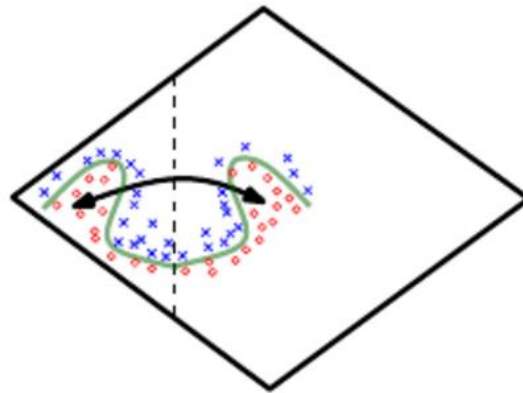
When a Linear Boundary Is Not Enough

What if the classes cannot be separated by a straight line in the current feature space?



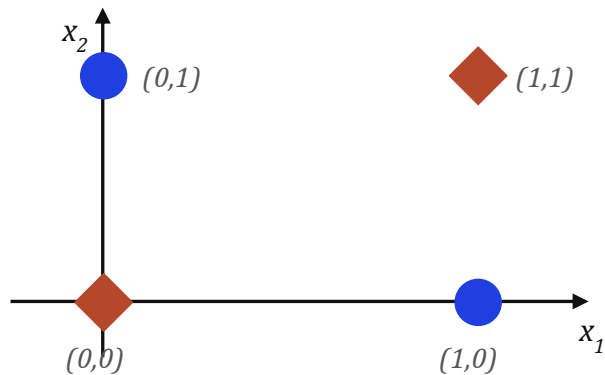
Fold-and-Cut Theorem: a complex separation in the original space may become a simple straight cut after the space is transformed.

- Notice: folding changes the distance between points
- How to achieve such folding?
 - One way is to **transform** the data



Transforming

What if the classes cannot be separated by a straight line in the current feature space?



Transformation

$$X \longrightarrow \varphi(X)$$

$$\varphi(x) = (x_1 + x_2, x_1 x_2)$$

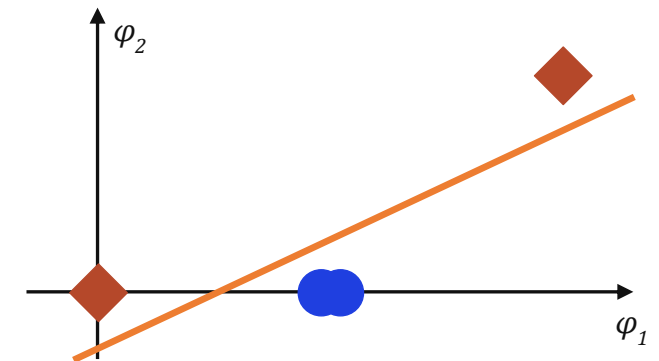
Transformed feature space:

$$f(x) = w^T \varphi(x) + b$$

later: kernels



Linear separation in transformed space



A simple classifier can become powerful if the representation is appropriate.

How do we decide whether a particular boundary is good?

What do w and b control?

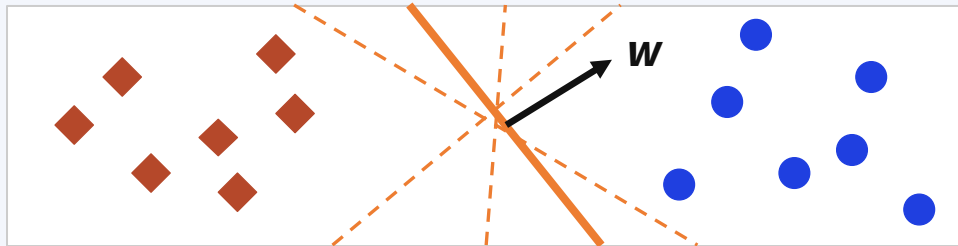
$$f(x) = w^T x + b$$

The weight vector w

w controls the **orientation** of the boundary.

$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ is perpendicular to the decision boundary.

Changing the relative values of w_1 and w_2 rotates the line.

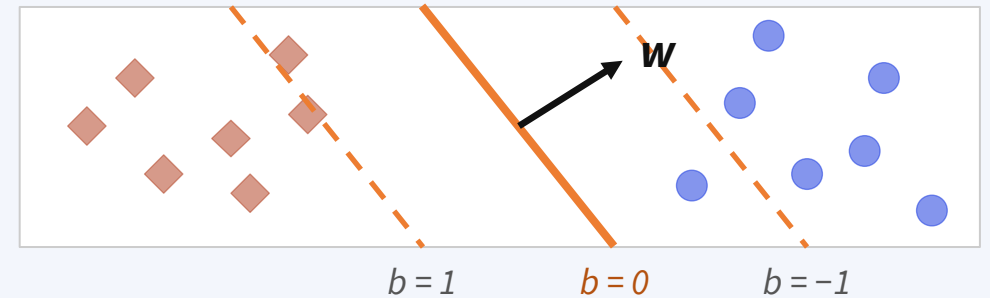


$w \rightarrow$ orientation

The bias b

b controls the **position** of the boundary.

Changing b moves the line parallel to itself, without changing its orientation.



$b \rightarrow$ position

Which values of w and b should we choose?

We need an **evaluation** criterion to choose/learn them.

REO: **Building Linear Discriminants** **Evaluation**

Evaluation: When is a Classification Correct?

For each training example (x_i, y_i) , where $y_i \in \{-1, +1\}$:
the classifier produces the **score** $f(x_i) = w^T x_i + b$

Correct classification

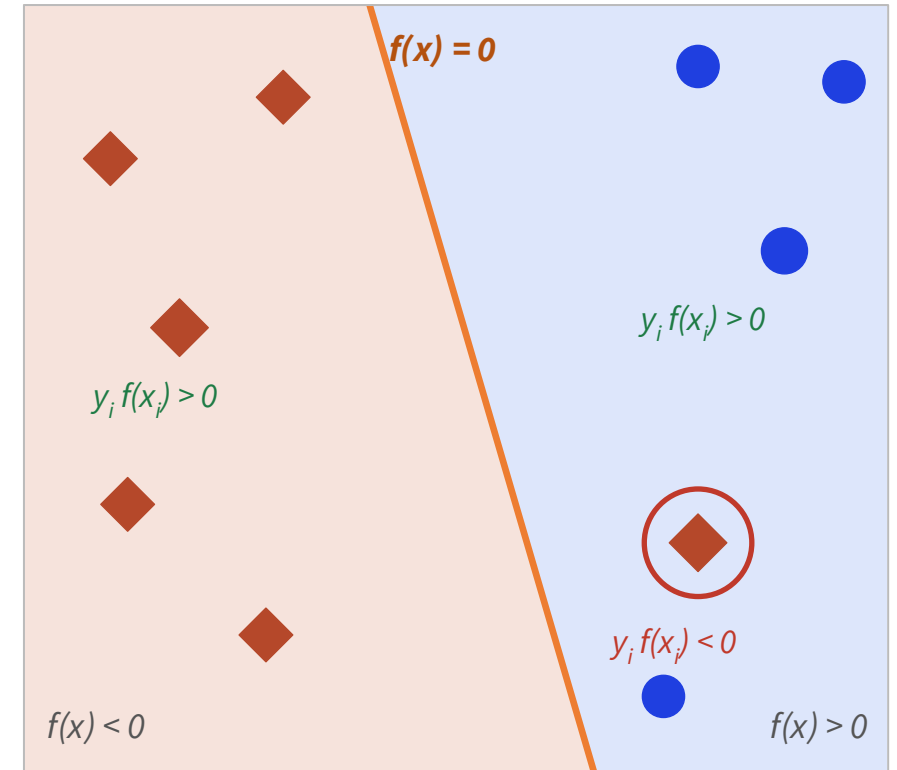
$y_i = +1 \Rightarrow f(x_i) > 0$ (positive example)

$y_i = -1 \Rightarrow f(x_i) < 0$ (negative example)

Combining both cases into one expression:

$$y_i f(x_i) > 0$$

Value of $y_i f(x_i)$	Interpretation
> 0	correct side of the boundary
< 0	wrong side of the boundary
$= 0$	exactly on the boundary



How do we turn “correct” or “incorrect” into a **quantity** that we can minimise?

Evaluation: 0 – 1 Misclassification Loss

For training example (x_i, y_i) , with $y_i \in \{-1, +1\}$, we know:

$y_i f(x_i) > 0 \Rightarrow$ correct classification

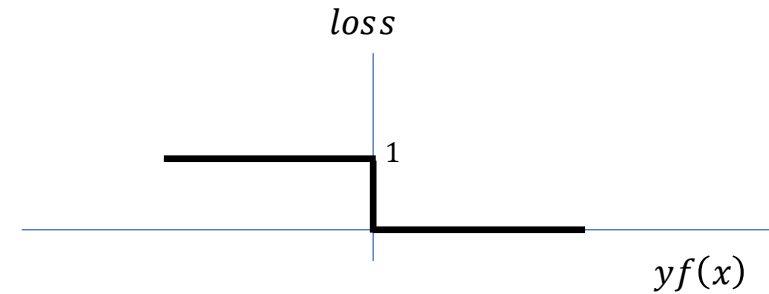
$y_i f(x_i) \leq 0 \Rightarrow$ misclassification

Definition for one example

$$l(f(x_i), y_i) = \begin{cases} 0, & y_i f(x_i) > 0 \\ 1, & y_i f(x_i) \leq 0 \end{cases}$$

$l = 0$: correct prediction $l = 1$: wrong prediction

- $y_i f(x_i) > 0$ *only when* the prediction has the same sign as the true label



From one example to the training set

$$L(w, b) = (1/N) \sum_{i=1 \dots N} \mathbb{I}[y_i(w^T x_i + b) \leq 0]$$

This is simply the proportion of training examples classified incorrectly.

0-1 Loss in Practice

$$l(f(x_i), y_i) = \begin{cases} 0, & y_i f(x_i) > 0 \\ 1, & y_i f(x_i) \leq 0 \end{cases}$$

Suppose the current linear classifier is

$$f(\mathbf{x}) = 2x_1 - x_2 + 0.4 \quad \text{so } \mathbf{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix} \quad b = 0.4$$

First example

$$\mathbf{x}_1 = \begin{bmatrix} 1.2 \\ 0.5 \end{bmatrix}, \quad y_1 = +1$$

The classifier score

$$\begin{aligned} f(x_1) &= \mathbf{w}^T \mathbf{x}_1 + b \\ &= 2(1.2) - 1(0.5) + 0.4 \\ &= 2.4 - 0.5 + 0.4 \\ &= 2.3 \end{aligned}$$

$$\text{Then: } y_1 f(x_1) = (+1)(2.3) =$$

$$+2.3 > 0$$

correct classification $\Rightarrow l = 0$

0–1 Loss in Practice

$$l(f(x_i), y_i) = \begin{cases} 0, & y_i f(x_i) > 0 \\ 1, & y_i f(x_i) \leq 0 \end{cases}$$

Suppose the current linear classifier is

$$f(\mathbf{x}) = 2x_1 - x_2 + 0.4 \quad \text{so } \mathbf{w} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \quad \mathbf{b} = 0.4$$

First example

$$\mathbf{x}_1 = \begin{bmatrix} 1.2 \\ 0.5 \end{bmatrix}, \quad y_1 = +1$$

The classifier score

$$\begin{aligned} f(x_1) &= \mathbf{w}^T \mathbf{x}_1 + b \\ &= 2(1.2) - 1(0.5) + 0.4 \\ &= 2.4 - 0.5 + 0.4 \\ &= \mathbf{2.3} \end{aligned}$$

$$\text{Then: } y_1 f(x_1) = (+1)(+2.3) =$$

$$\mathbf{+2.3} > 0$$

correct classification $\Rightarrow l = 0$

The same classifier applied to four labelled examples

$\mathbf{x}_i = (x_1, x_2)$	y_i	$f(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i + b$	$y_i f(\mathbf{x}_i)$	Correct?	l
(1.2, 0.5)	+1	$2(1.2) - 1(0.5) + 0.4 = 2.3$	$(+1)(+2.3) = +2.3$	Yes	0
(0.2, 1.5)	+1	$2(0.2) - 1(1.5) + 0.4 = -0.7$	$(+1)(-0.7) = -0.7$	No	1
(0.4, 3.0)	-1	$2(0.4) - 1(3.0) + 0.4 = -1.8$	$(-1)(-1.8) = +1.8$	Yes	0
(0.5, 0.9)	-1	$2(0.5) - 1(0.9) + 0.4 = 0.5$	$(-1)(+0.5) = -0.5$	No	1

Why Can't We Just Minimise 0-1 Loss?

We now have exactly the objective we care about, so why not optimise it directly?

$$l_{0/1} = (1/N) \sum_i \mathbb{1}[y_i f(x_i) \leq 0]$$

All mistakes receive exactly the same loss.

Flat almost everywhere

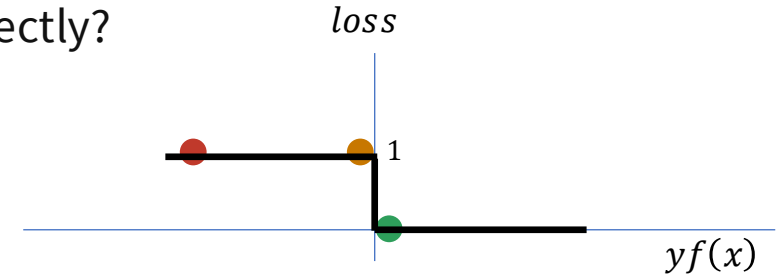
Small changes to w usually leave the loss unchanged.

Jump at the decision boundary

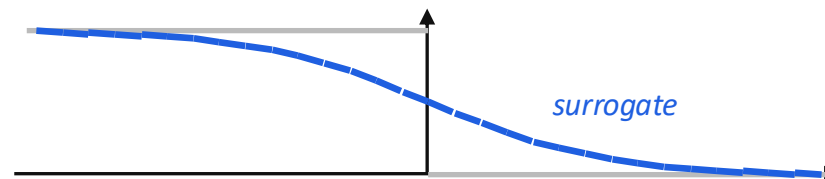
At $y_i f(x_i) = 0$ the loss jumps from 1 to 0, so it is **not differentiable** there.

No useful direction for gradient descent

Gradient descent does not know which small change reduces the loss.



- $yf(x) = -3$: badly wrong, $l = 1$
- $yf(x) = -0.1$: barely wrong, $l = 1$
- $yf(x) = +0.1$: barely correct, $l = 0$



Solution: use a **surrogate loss**.

It should give **larger penalties** to worse predictions, change **smoothly**, and be **easier** to optimise.

A Possible Surrogate: The Perceptron Criterion

Main idea

Instead of giving every mistake the same loss of 1, penalise a misclassified example according to how far it is on the wrong side:

$$l_{\text{perc}}(y_i, f(x_i)) = \max(0, -y_i f(x_i))$$

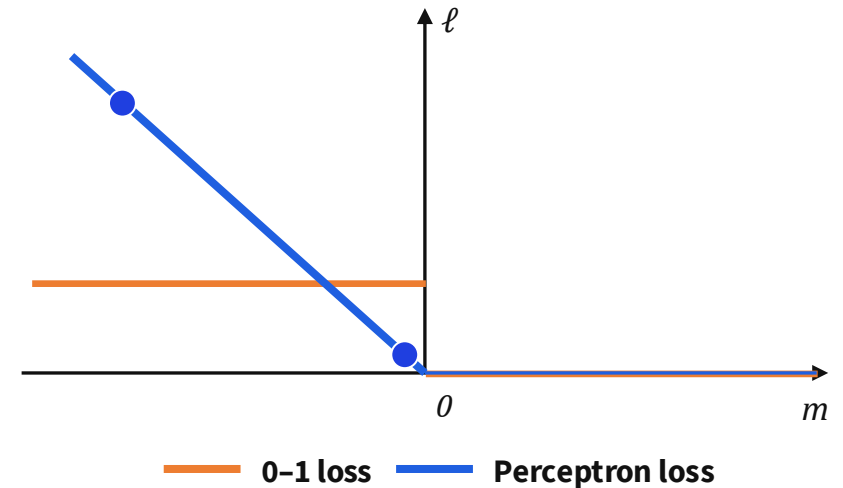
$$y_i f(x_i) > 0 \Rightarrow l_{\text{perc}} = 0$$

correct side: no penalty

$$y_i f(x_i) < 0 \Rightarrow l_{\text{perc}} = -y_i f(x_i) > 0$$

wrong side: larger mistakes receive larger penalties

Still: once $y_i f(x_i) > 0$, its loss is constantly zero. **Are all correct predictions equally good?**



Loss values for the four examples

$\mathbf{x}_i = (x_1, x_2)$	y_i	$f(x_i)$	$y_i f(x_i)$	Correct?	$\ell_{0/1}$	$\ell_{\text{perc}} = \max(0, -y_i f(x_i))$
(1.2, 0.5)	+1	+2.3	+2.3	Yes	0	$\max(0, -2.3) = 0$
(0.2, 1.5)	+1	-0.7	-0.7	No	1	$\max(0, +0.7) = 0.7$
(0.4, 3.0)	-1	-1.8	+1.8	Yes	0	$\max(0, -1.8) = 0$
(0.5, 0.9)	-1	+0.5	-0.5	No	1	$\max(0, +0.5) = 0.5$

A Possible Surrogate: The Perceptron Criterion

Main idea

Instead of giving every mistake the same loss of 1, penalise a misclassified example according to how far it is on the wrong side:

$$l_{\text{perc}}(y_i, f(x_i)) = \max(0, -y_i f(x_i))$$

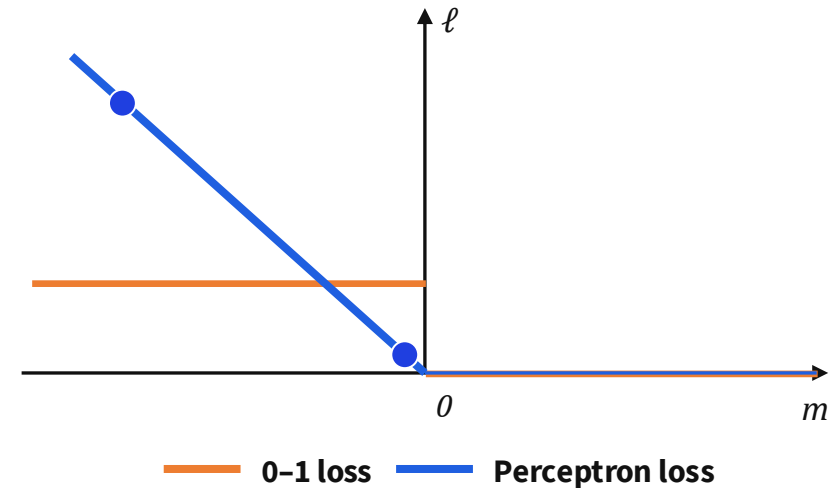
$$y_i f(x_i) > 0 \Rightarrow l_{\text{perc}} = 0$$

correct side: no penalty

$$y_i f(x_i) < 0 \Rightarrow l_{\text{perc}} = -y_i f(x_i) > 0$$

wrong side: larger mistakes receive larger penalties

Still: once $y_i f(x_i) > 0$, its loss is constantly zero. **Are all correct predictions equally good?**



Now the objective is not constant and depends on w and b . We can try to minimize it!

Representation $f(x) = w^T x + b$



Evaluation $l_{\text{perc}} = \max(0, -yf)$



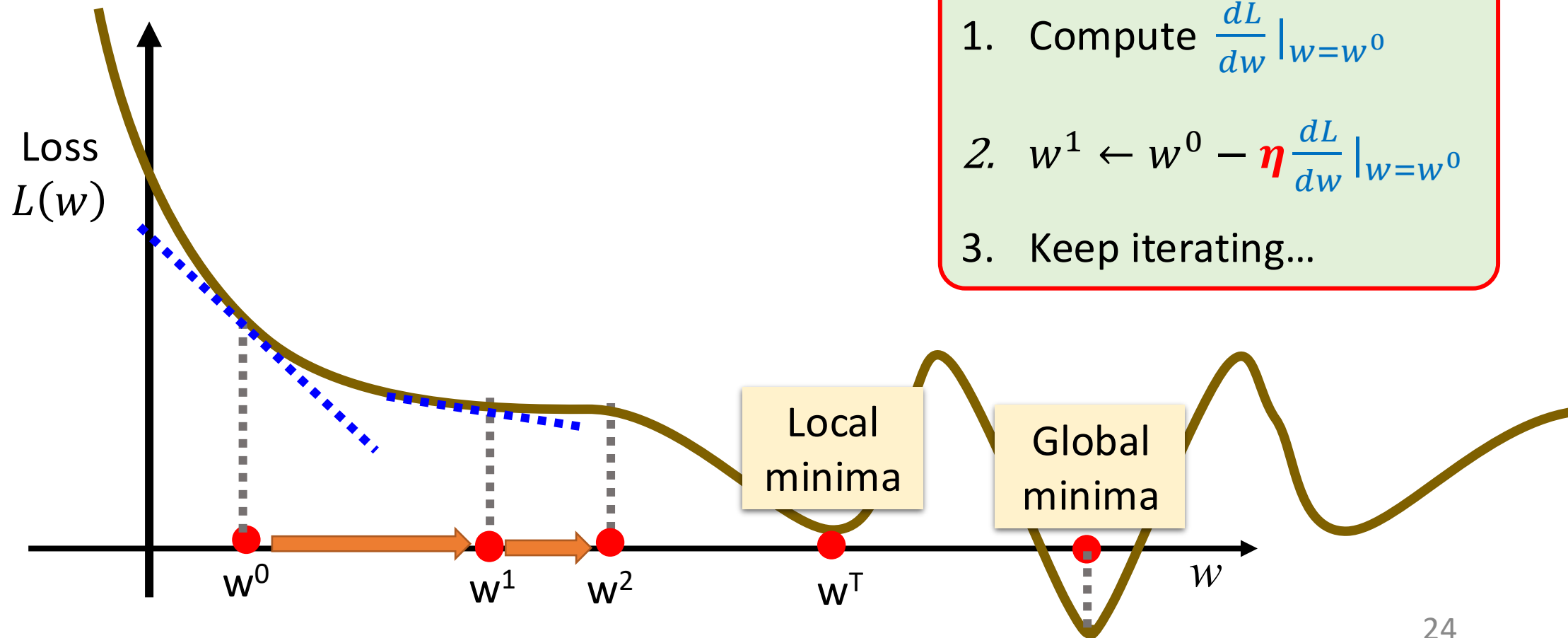
Optimization ?

How should we change w and b to reduce this loss?

REO: **Building Linear Discriminants** **Optimisation**

Find the best solution – Optimising via Gradient Descent

- Consider **loss function** $L(w)$ with one parameter w : $w^* = \arg \min_w L(w)$



1. Compute $\frac{dL}{dw} \big|_{w=w^0}$
2. $w^1 \leftarrow w^0 - \eta \frac{dL}{dw} \big|_{w=w^0}$
3. Keep iterating...

Recap: Loss Functions

Parameters

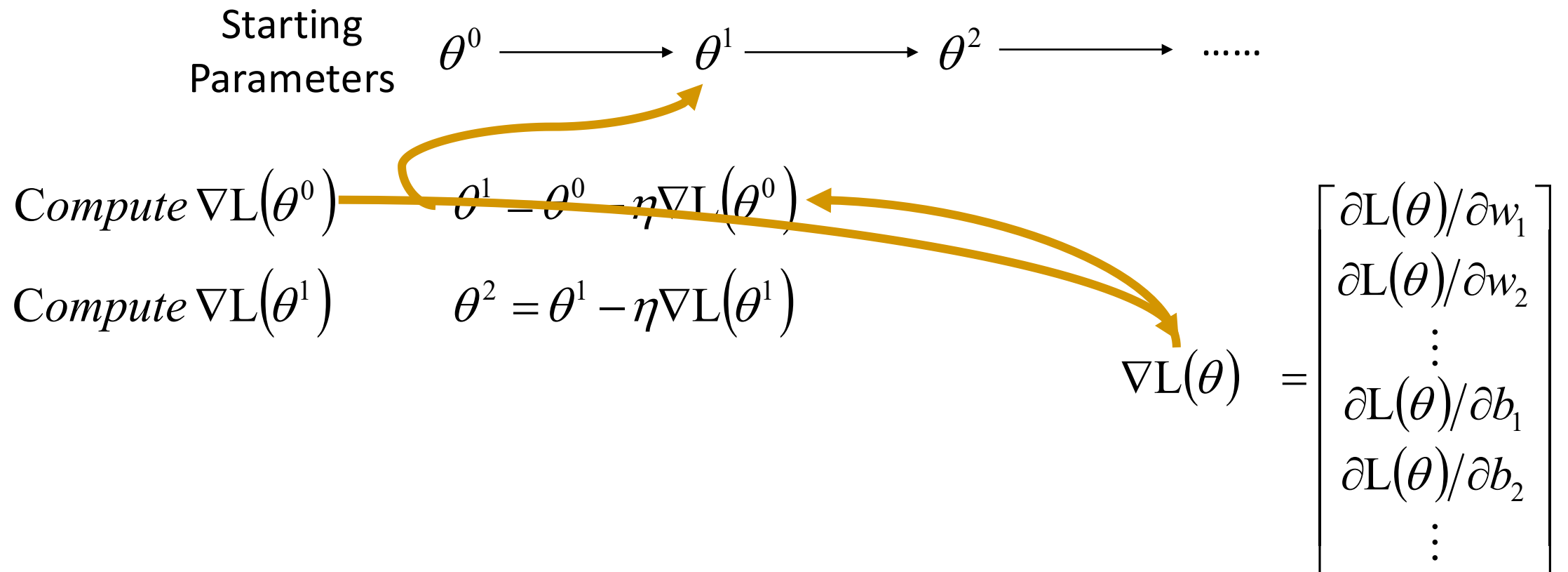
$$\theta = \{W, b\}$$

Optimise network parameters using
gradient descent:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} \mathcal{L}(\theta)$$

Gradient Descent

- Network parameters: $\theta = \{w_1, w_2, \dots, b_1, b_2, \dots\}$



Building Linear Discriminants

- Representation

- Features
- Linear Function

$$f(\mathbf{x}; \mathbf{w}) = w_1 x^{(1)} + w_2 x^{(2)} + \dots + w_d x^{(d)} + b = 0$$

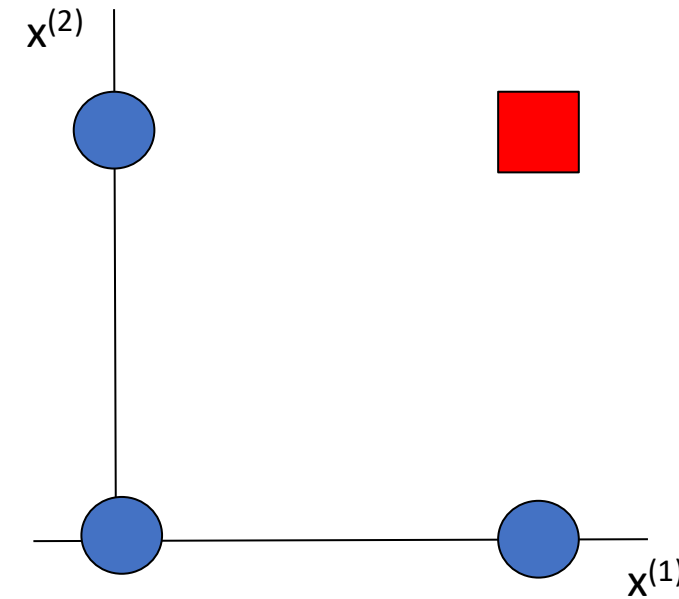
- Evaluation

- Surrogate Loss (from the 0/1 loss):

$$l_i = \max(0, -y_i f(x_i)) = \max(0, -y_i (w^T x_i + b))$$

- Optimization

- Find a line that minimizes misclassifications
- How done: Visual reckoning / Constraint Satisfaction



Building Linear Discriminants

- **Representation**

- Features
- Linear Function

$$f(\mathbf{x}; \mathbf{w}) = w_1 x^{(1)} + w_2 x^{(2)} + \dots + w_d x^{(d)} + b = 0$$

- **Evaluation**

- Surrogate Loss (from the 0/1 loss):

$$l_i = \max(0, -y_i f(x_i)) = \max(0, -y_i (w^T x_i + b))$$

- **Optimization**

- Gradient Descent

$$\nabla L = \frac{1}{N} \sum_{i=1}^N \nabla_w l(f(\mathbf{x}_i; \mathbf{w}), y_i)$$

If $y_i f(x_i) \leq 0$: $l_i = -y_i (w^T x_i + b)$

Gradients:

$$\nabla_w l_i = -y_i x_i \quad \partial l_i / \partial b = -y_i$$

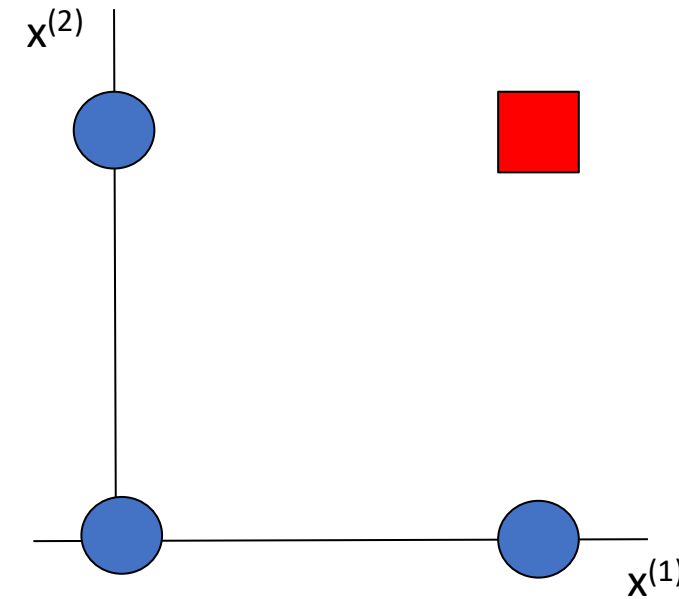
Gradient-descent update:

$$w \leftarrow w - \eta \nabla_w l_i = w + \eta y_i x_i$$

$$b \leftarrow b - \eta \partial l_i / \partial b = b + \eta y_i$$

Perceptron learning rule

$$\text{if } y_i f(x_i) \leq 0: \quad w \leftarrow w + \eta y_i x_i \quad b \leftarrow b + \eta y_i$$



Algorithm

- Given:
 - Training examples: $\{(\mathbf{x}_i, y_i) \mid i = 1 \dots N\}$, $y_i \in \{-1, +1\}$
 - Learning rate (step size): $\eta > 0$
- Initialise $\mathbf{w}^{(0)} = \mathbf{0}$ and $b^{(0)} = 0$
- Until convergence ($k = 1 \dots K$ epochs, or no mistakes occur)
 - For $i = 1 \dots N$
 - Pick example $\mathbf{x}_i$ with label y_i
 - Compute $f(\mathbf{x}_i) = \mathbf{w}^{(k-1)\top} \mathbf{x}_i + b^{(k-1)}$
 - If $y_i f(\mathbf{x}_i) \leq 0$ (a mistake) then update $\mathbf{w}$ and b by gradient descent

$$\mathbf{w}^{(k)} = \mathbf{w}^{(k-1)} - \eta \nabla_{\mathbf{w}} \ell_i(\mathbf{w}^{(k-1)}) = \mathbf{w}^{(k-1)} - \eta(-y_i \mathbf{x}_i) = \mathbf{w}^{(k-1)} + \eta y_i \mathbf{x}_i$$

$$b^{(k)} = b^{(k-1)} - \eta \partial \ell_i / \partial b = b^{(k-1)} - \eta(-y_i) = b^{(k-1)} + \eta y_i$$

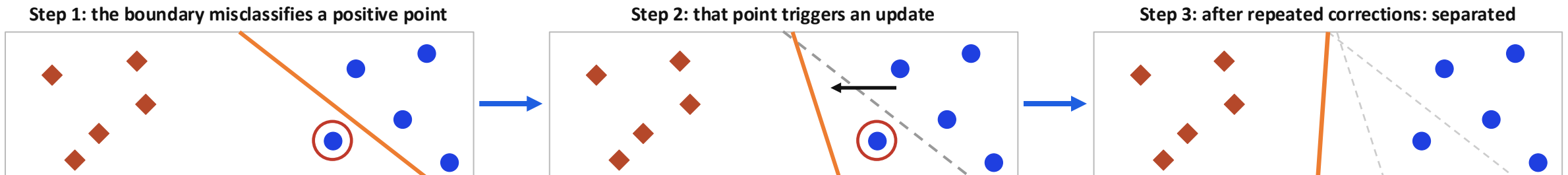
- Check for convergence to stop (an epoch with no mistakes)

$$\nabla_{\mathbf{w}} \max\{0, -y(\mathbf{w}^\top \mathbf{x} + b)\} = \begin{cases} \mathbf{0}, & y f(\mathbf{x}) > 0 \\ -y \mathbf{x}, & y f(\mathbf{x}) \leq 0 \end{cases}$$

(a valid subgradient at $y f(\mathbf{x}) = 0$)

Algorithm

- Given:
 - Training examples: $\{(x_i, y_i) \mid i = 1 \dots N\}$, $y_i \in \{-1, +1\}$
 - Learning rate (step size): $\eta > 0$
- Initialise $\mathbf{w}^{(0)} = \mathbf{0}$ and $b^{(0)} = 0$
- Until convergence ($k = 1 \dots K$ epochs, or no mistakes occur)
 - For $i = 1 \dots N$
 - Pick example $\mathbf{x}_i$ with label y_i
 - Compute $f(\mathbf{x}_i) = \mathbf{w}^{(k-1)\top} \mathbf{x}_i + b^{(k-1)}$
 - If $y_i f(\mathbf{x}_i) \leq 0$ (a mistake) then update $\mathbf{w}$ and b by gradient descent
$$\mathbf{w}^{(k)} = \mathbf{w}^{(k-1)} - \eta \nabla_{\mathbf{w}} \ell_i(\mathbf{w}^{(k-1)}) = \mathbf{w}^{(k-1)} - \eta(-y_i \mathbf{x}_i) = \mathbf{w}^{(k-1)} + \eta y_i \mathbf{x}_i$$
$$b^{(k)} = b^{(k-1)} - \eta \partial \ell_i / \partial b = b^{(k-1)} - \eta(-y_i) = b^{(k-1)} + \eta y_i$$
 - Check for convergence to stop (an epoch with no mistakes)



The Perceptron Finds a Separator: but which one?

Suppose the training data are linearly separable. There may be many decision boundaries for which the training error is zero.

$$f_A(x) = 0, \quad f_B(x) = 0, \quad f_C(x) = 0$$

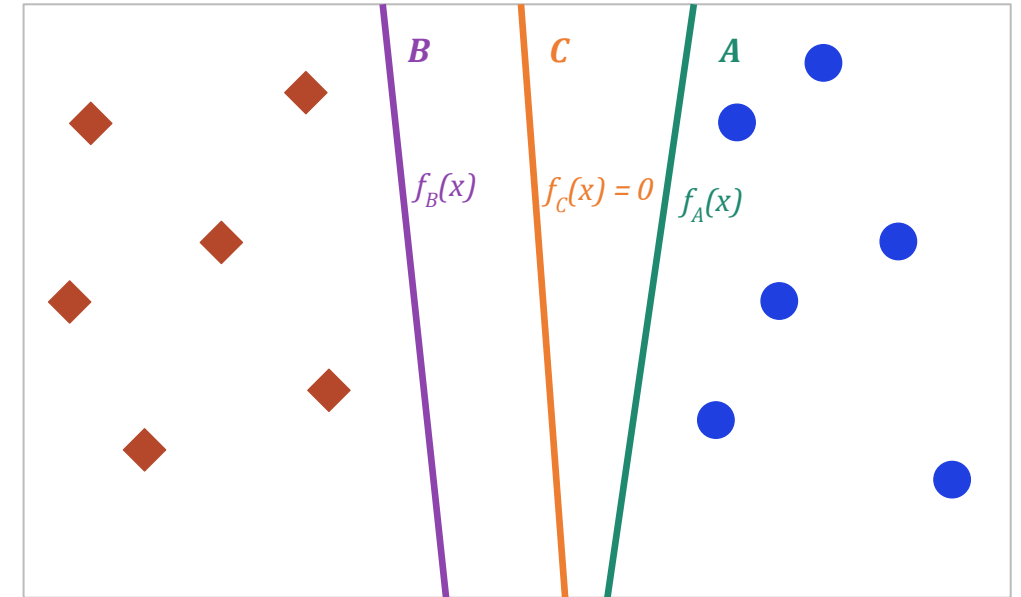
All three correctly classify every training example.

If they all make zero training errors, are they equally good?

- 0–1 error cannot distinguish between them.

Perceptron Convergence Theorem

If the training data are linearly separable, the perceptron [will find a separating hyperplane](#) after a finite number of updates.



The perceptron convergence theorem tells us:

a separating hyperplane will be found

but it does not tell us:

which separating hyperplane should be preferred

Can we prefer a separator that not only classifies the training data correctly, but also stays as far as possible from the training examples?

Margin and Support Vector Machine
(SVM)

Coding Exercise

```
import numpy as np
import matplotlib.pyplot as plt
import itertools

class Perceptron:

    def __init__(self, alpha = 0.1, epochs = 200):
        self.alpha = alpha
        self.epochs = epochs
        self.W = np.array([0])
        self.bias = np.random.randn()
        self.Lambda = 0.5
    def fit(self, Xtr, Ytr):
        d = Xtr.shape[1]
        self.W = np.random.randn(d)
        for e in range(self.epochs):
            finished = True
            for i, x in enumerate(Xtr):
                if Ytr[i]*self.predict(np.atleast_2d(x))<1:
                    finished = False
                    self.W += self.alpha*Ytr[i]*x
                    self.bias += self.alpha*Ytr[i]
            if finished: break

    def score(self, x):
        return np.dot(x, self.W) + self.bias

    def predict(self, x):
        return np.sign(self.score(x))
```

```
if __name__ == '__main__':
    from plotit import plotit
    Xtr = np.array([[ -1, 0], [0, 1], [4, 4], [2, 3]])
    ytr = np.array([ -1, -1, +1, +1])
    clf = Perceptron()
    clf.fit(Xtr, ytr)
    z = clf.score(Xtr)
    print("Prediction Scores:", z)
    y = clf.predict(Xtr)
    print("Prediction Labels:", y)
    plotit(Xtr, ytr, clf=clf.score, conts=[0],
           extent = [-5, +5, -5, +5])
```

