



*CS545 – Machine Learning for Signal Processing*

# From linear to non-linear classifiers

28 September 2023

# Today's lecture

- The theme:
  - From linear to non-linear classifiers
- Neural Nets
  - From the perceptron to multiple layers
- Support Vector Machines
  - From linear to kernel operations

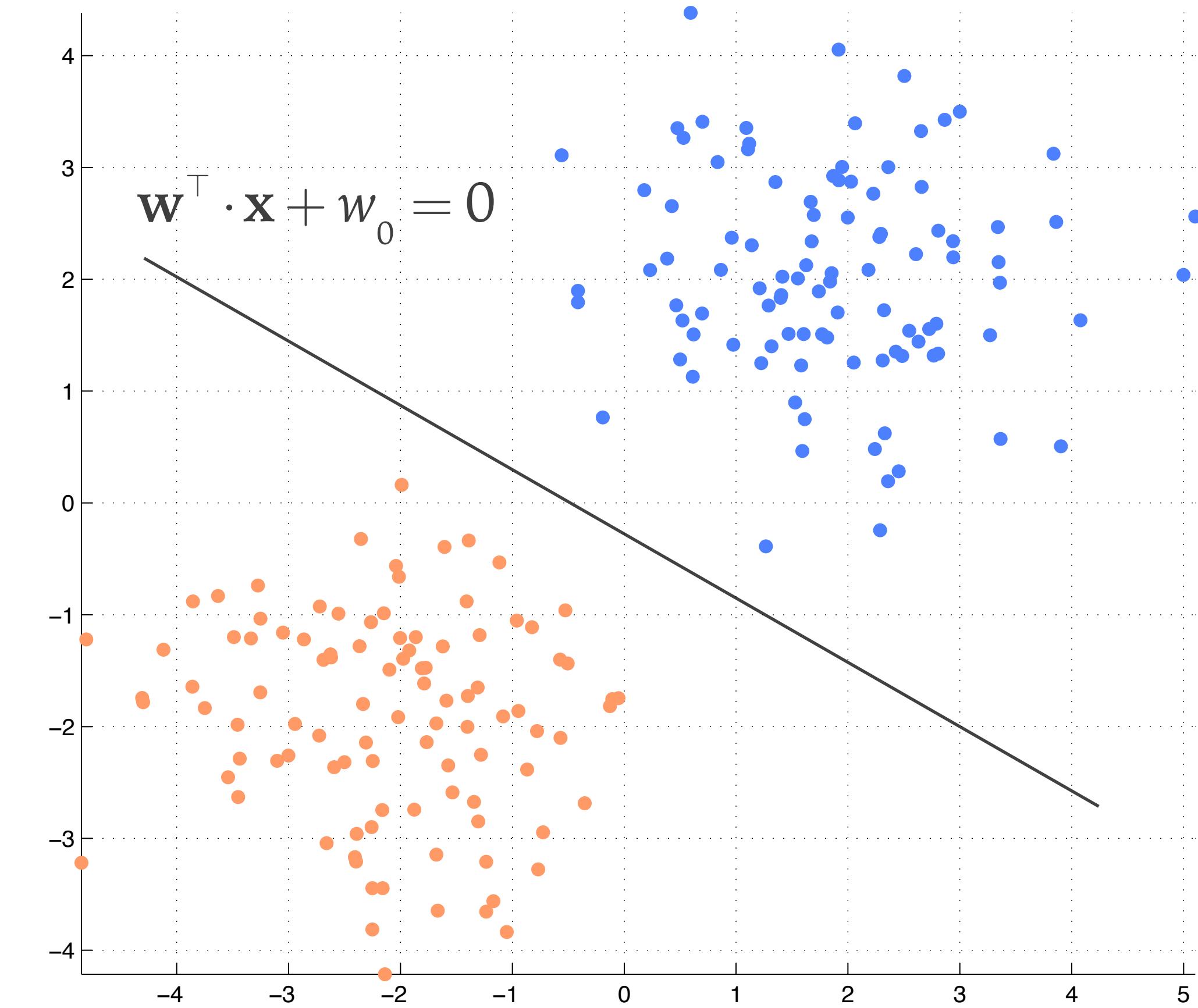
# The Perceptron again

- Defines the boundary that results in the least misclassifications

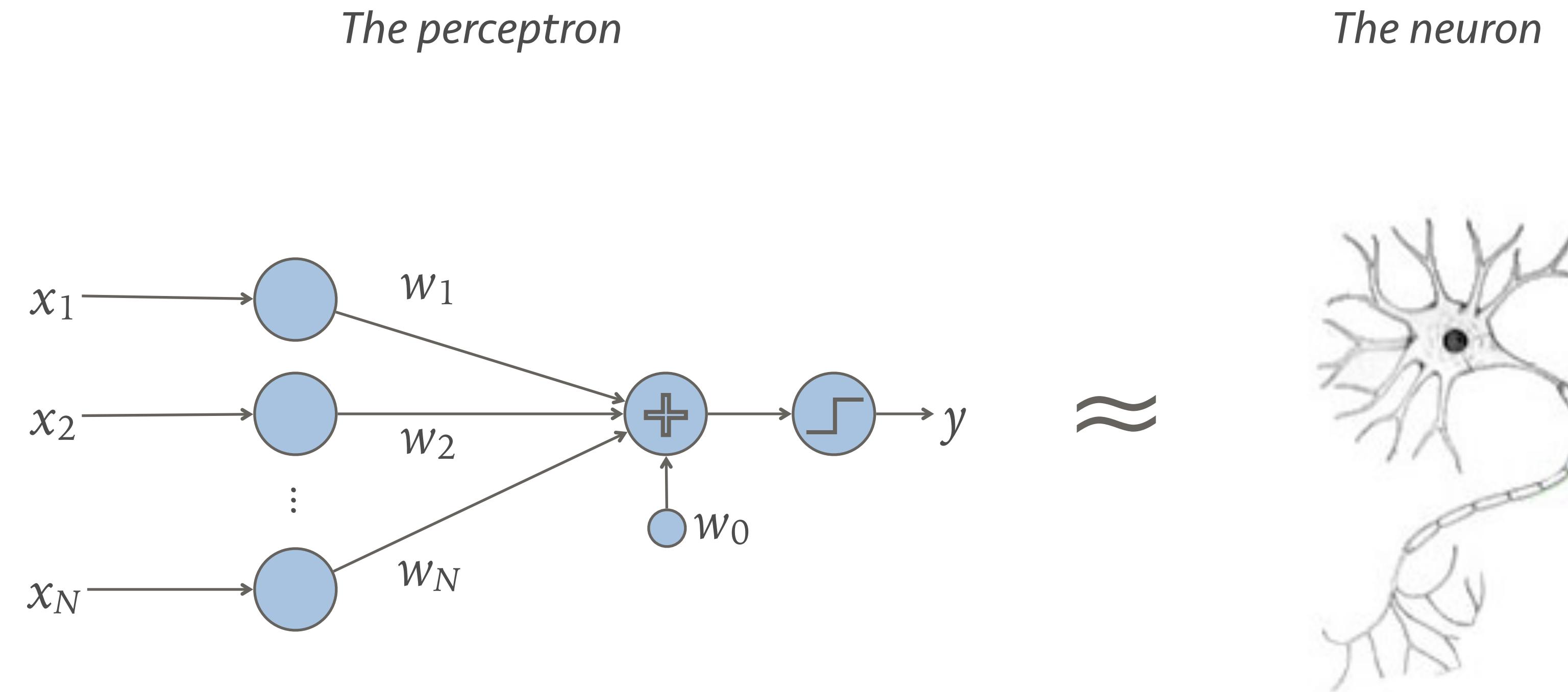
$$y_i = \mathbf{w}^\top \cdot \mathbf{x}_i + w_0 > 0$$

- Learn iteratively:

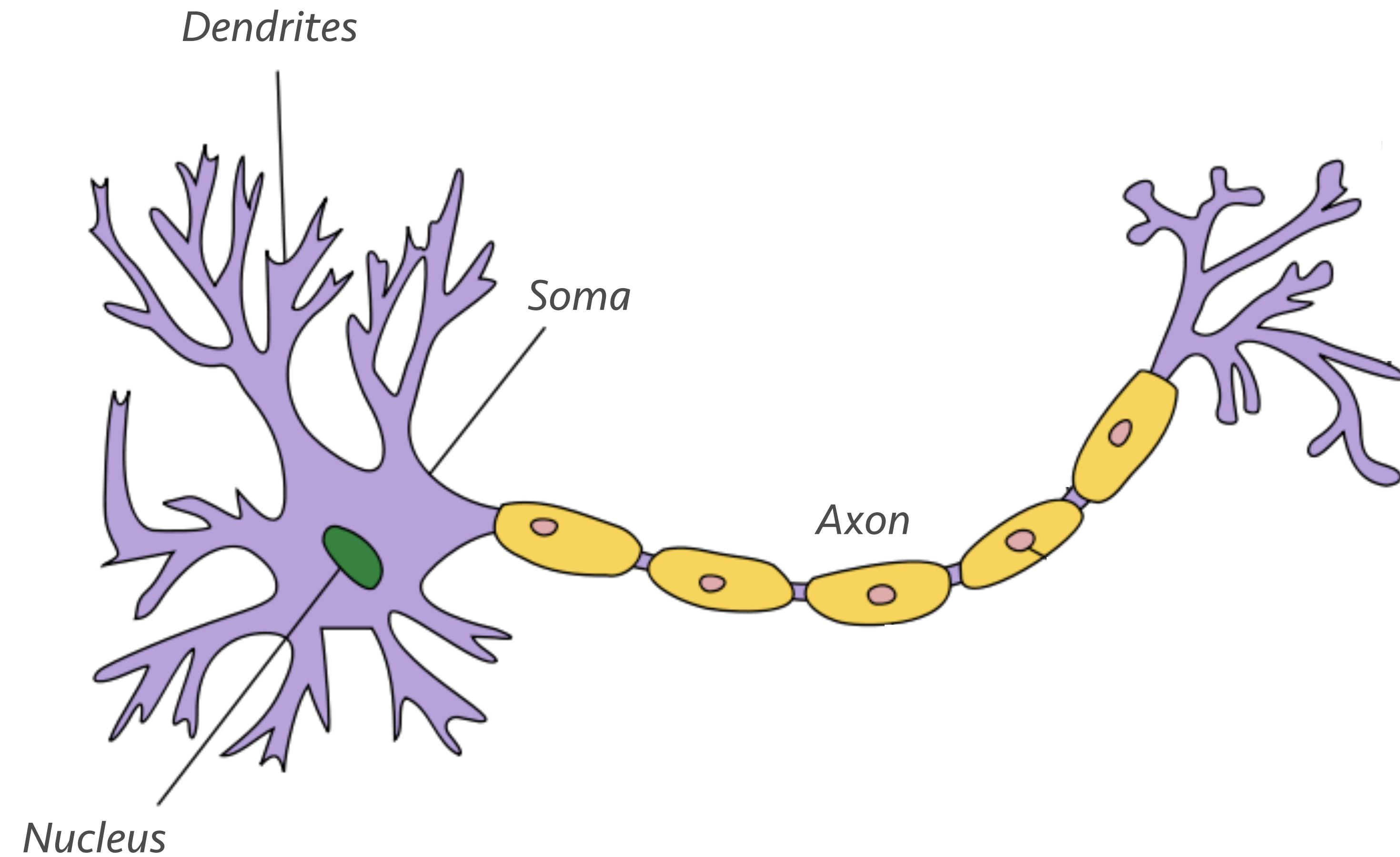
$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$$



# The biological inspiration



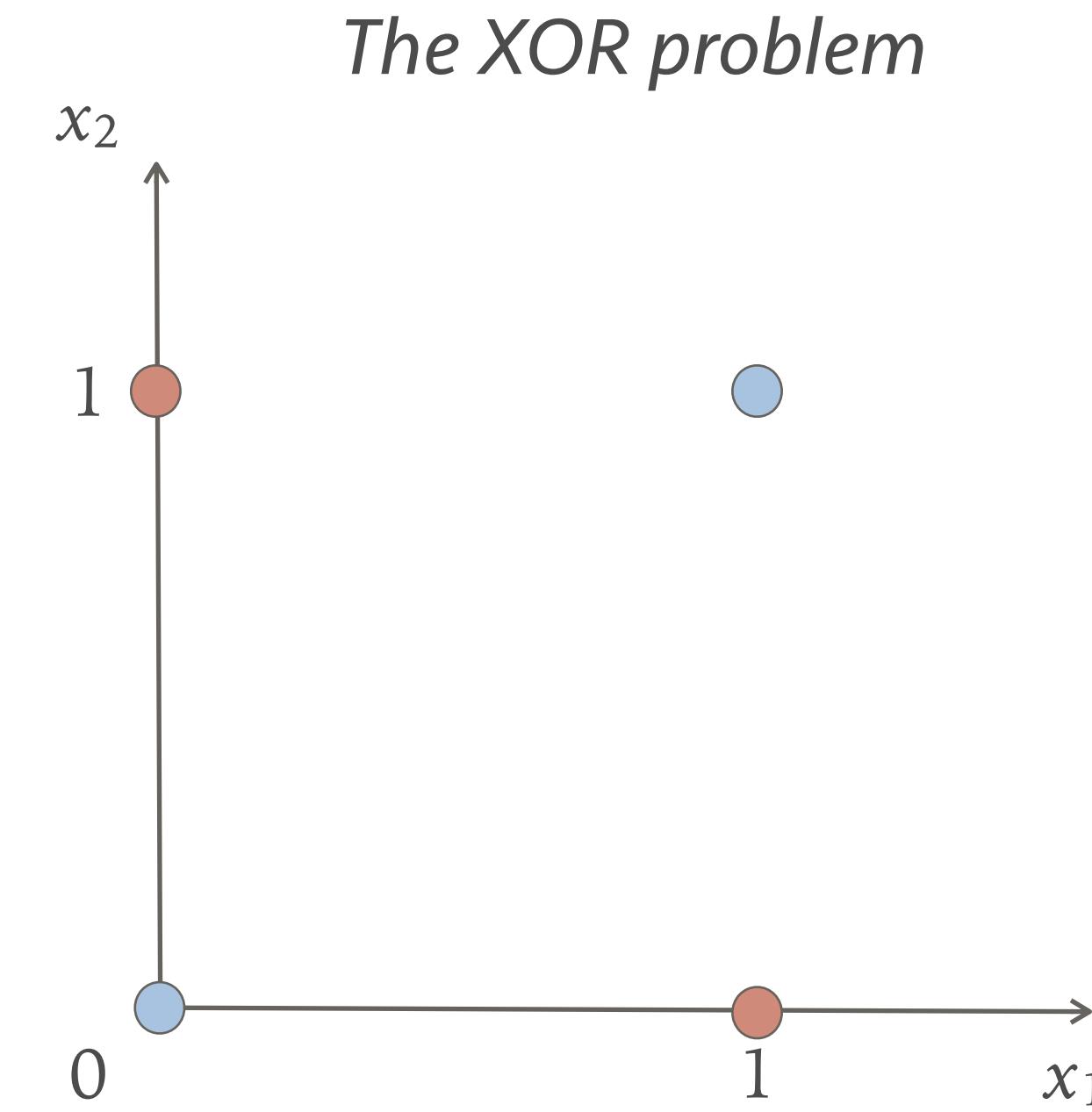
# The Neuron



# The problem case

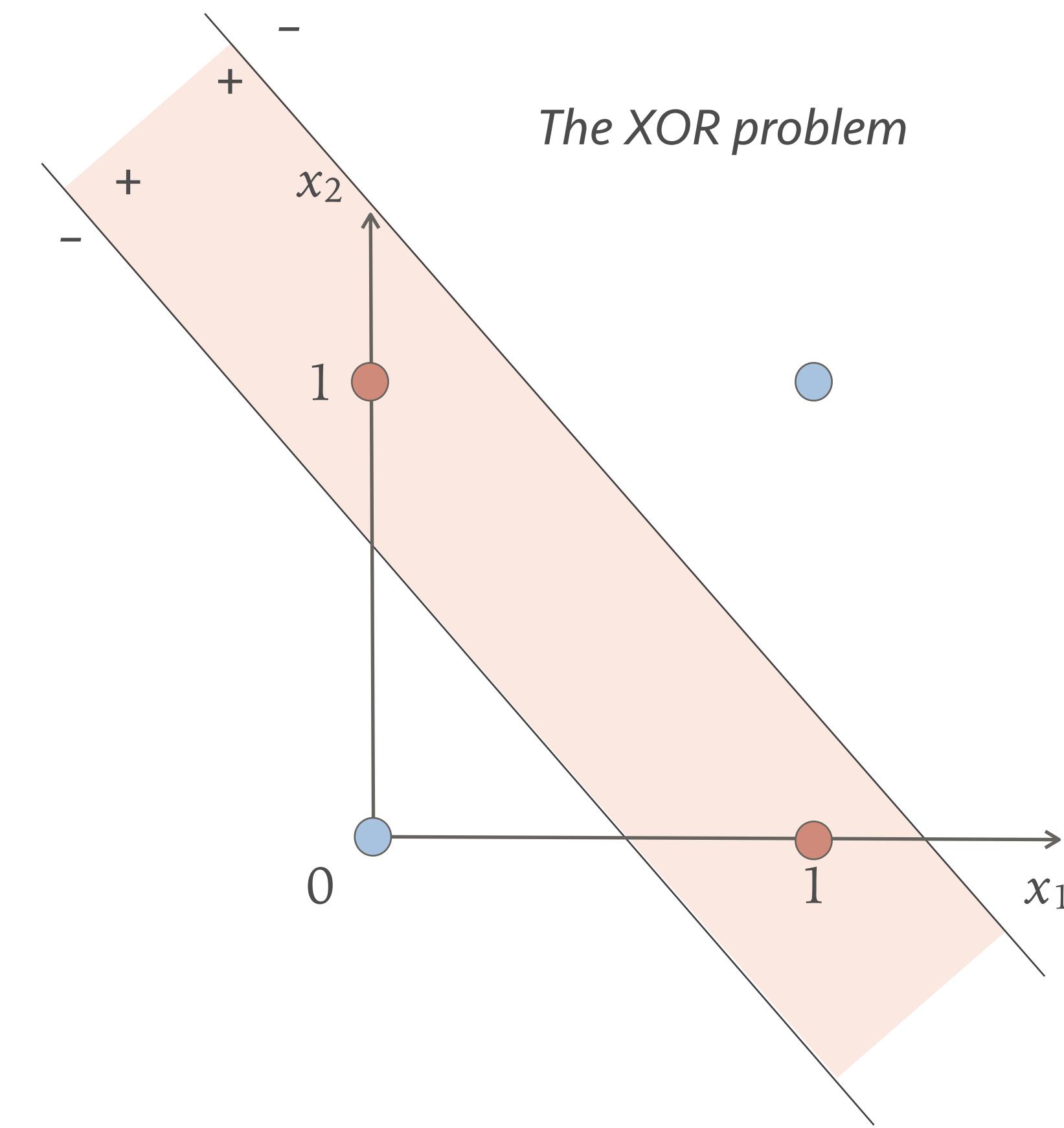
- The XOR case
  - Linearly non-separable case

$x_1$	$x_2$	XOR	Class
0	0	0	-1
0	1	1	+1
1	0	1	+1
1	1	0	-1

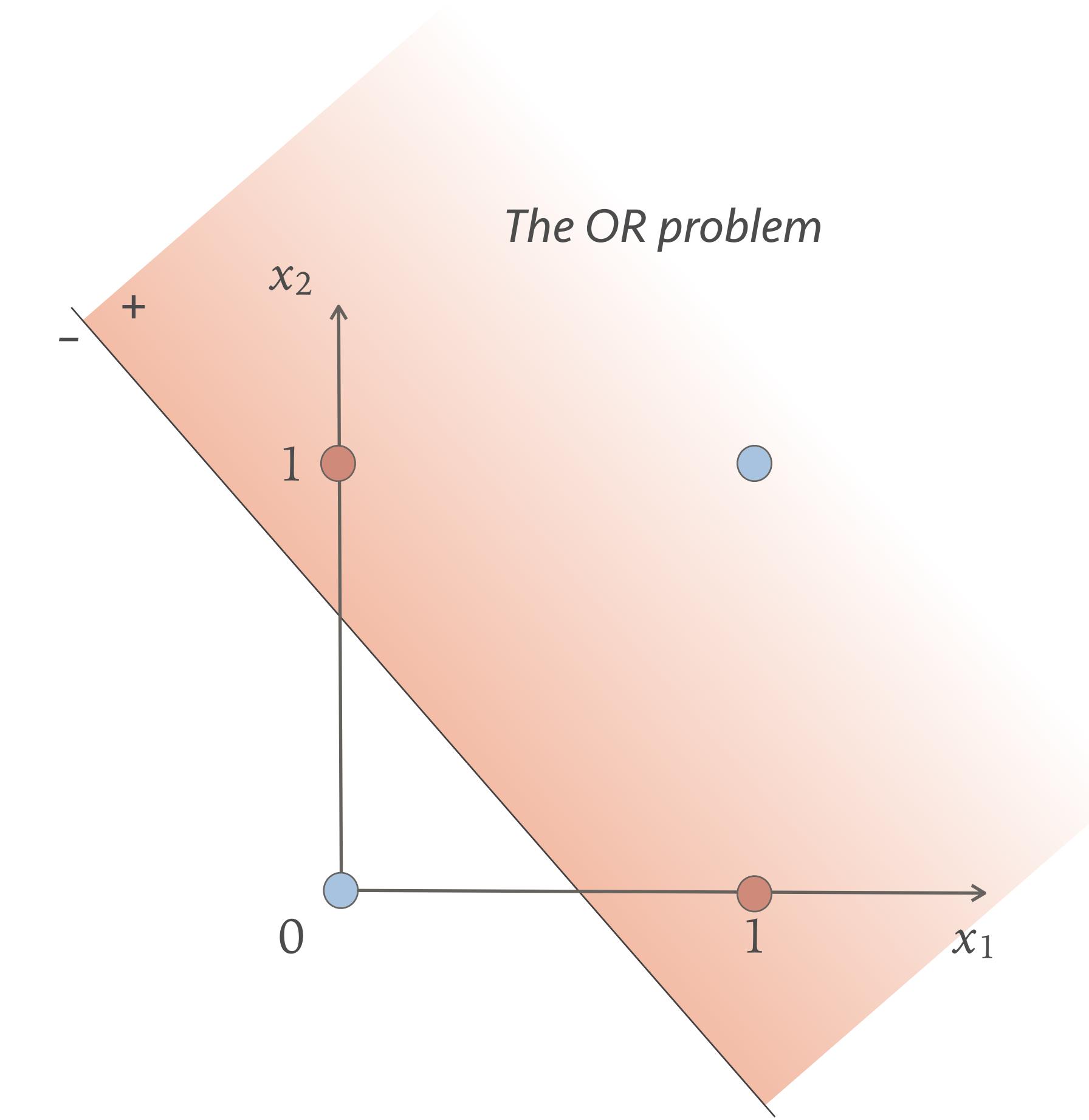
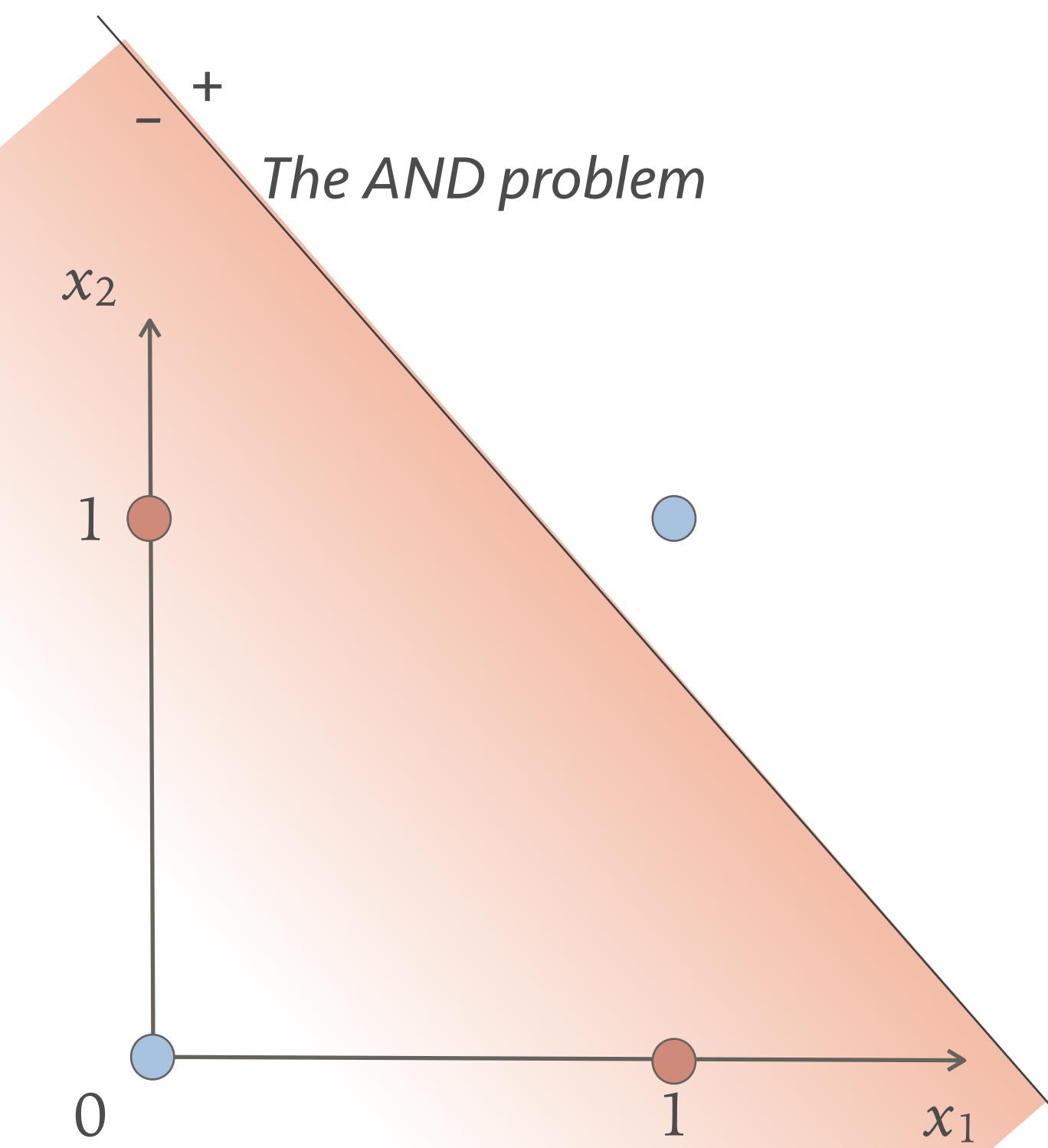


# An ideal solution

- Use two decision lines!
- How do we do this?



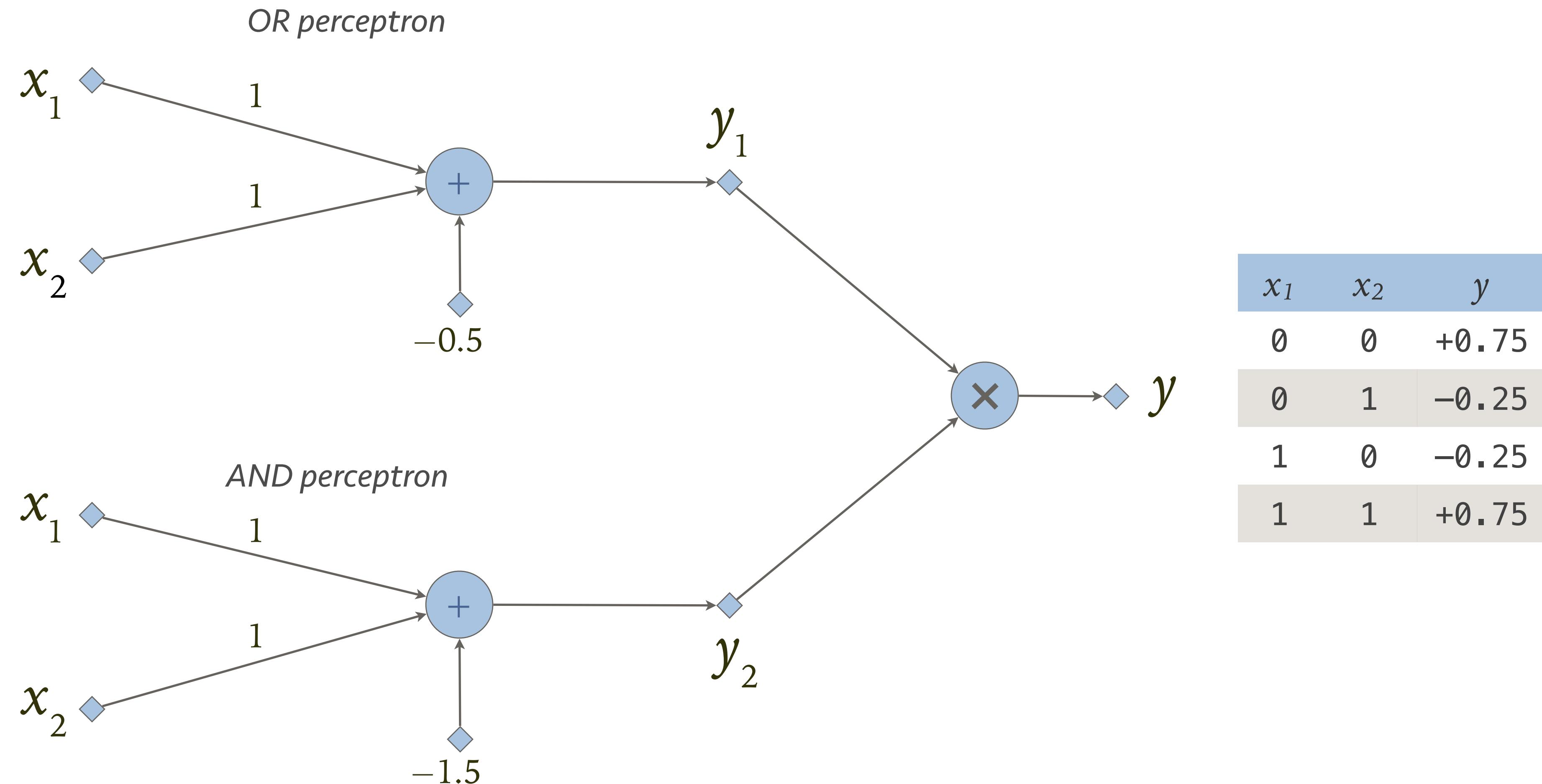
# An AND and an OR



# Two parallel perceptrons

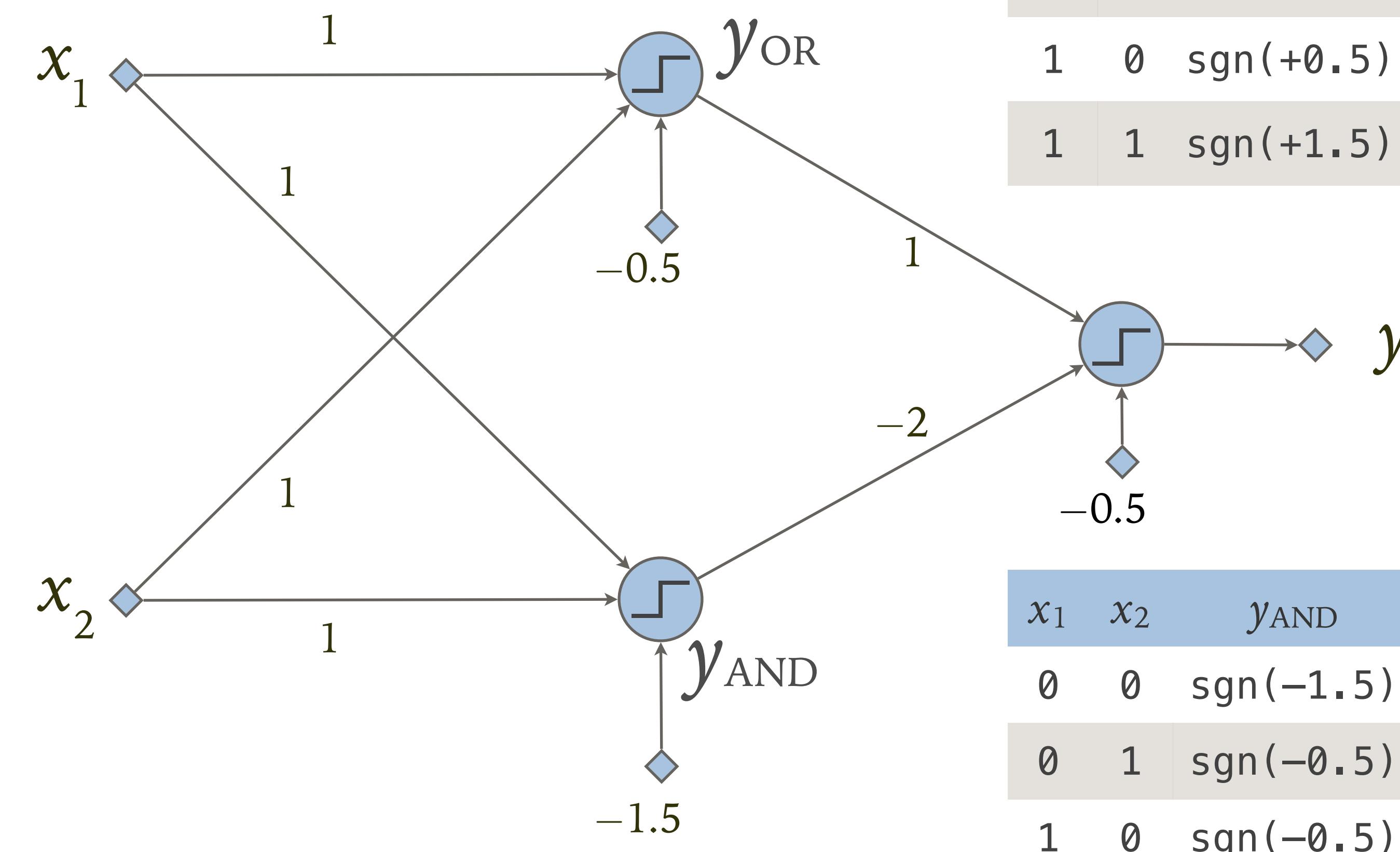
$x_1$	$x_2$	$y_1$
0	0	-0.5
0	1	+0.5
1	0	+0.5
1	1	+1.5

$x_1$	$x_2$	$y_2$
0	0	-1.5
0	1	-0.5
1	0	-0.5
1	1	+0.5



# Or more compactly

- As one 2-layer perceptron



$x_1$	$x_2$	$y_{\text{OR}}$
0	0	$\text{sgn}(-0.5)$
0	1	$\text{sgn}(+0.5)$
1	0	$\text{sgn}(+0.5)$
1	1	$\text{sgn}(+1.5)$

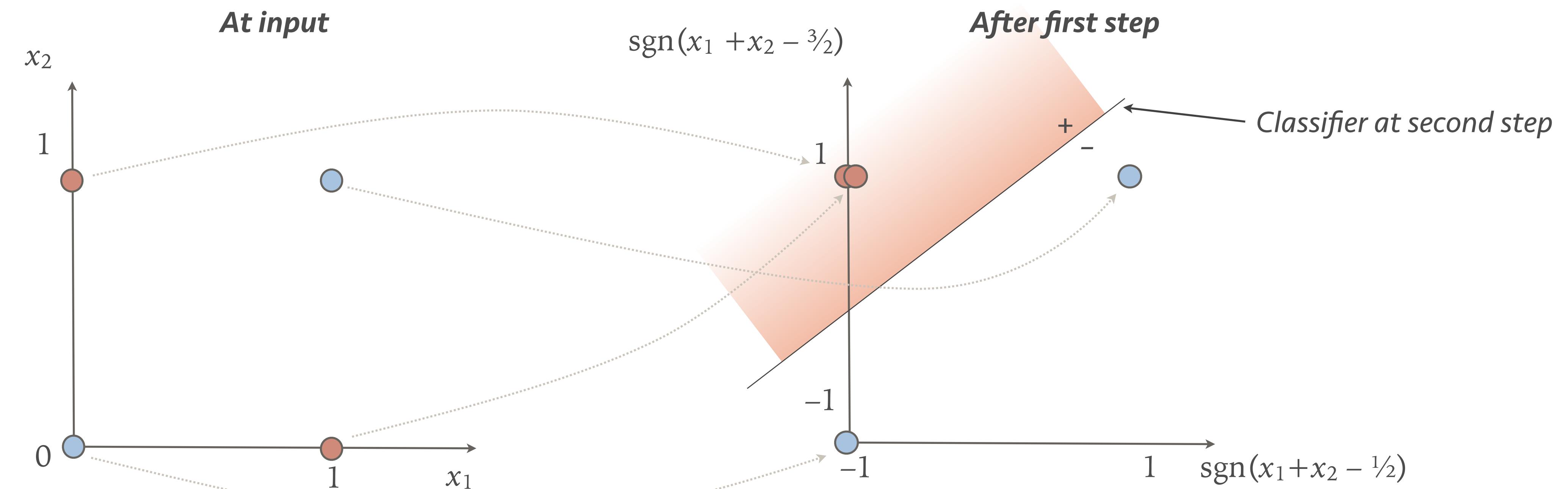
$x_1$	$x_2$	$y$	$\text{sgn}(y)$
0	0	-0.5	-1
0	1	+1.5	+1
1	0	+1.5	+1
1	1	-2.5	-1

$x_1$	$x_2$	$y_{\text{AND}}$
0	0	$\text{sgn}(-1.5)$
0	1	$\text{sgn}(-0.5)$
1	0	$\text{sgn}(-0.5)$
1	1	$\text{sgn}(+0.5)$

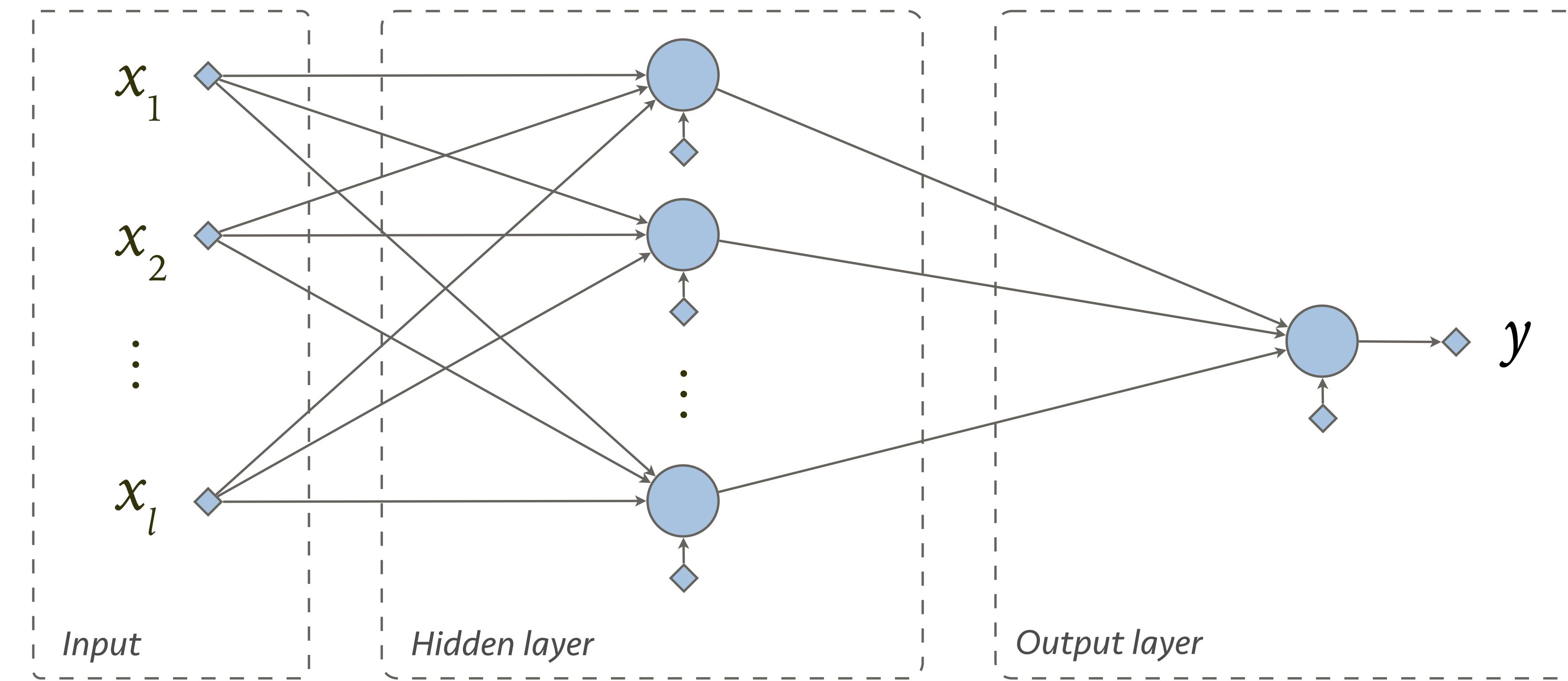
# A different interpretation

- First set of operations transforms the data to a space where the solution is linearly-separable
  - Second operation finishes the job!



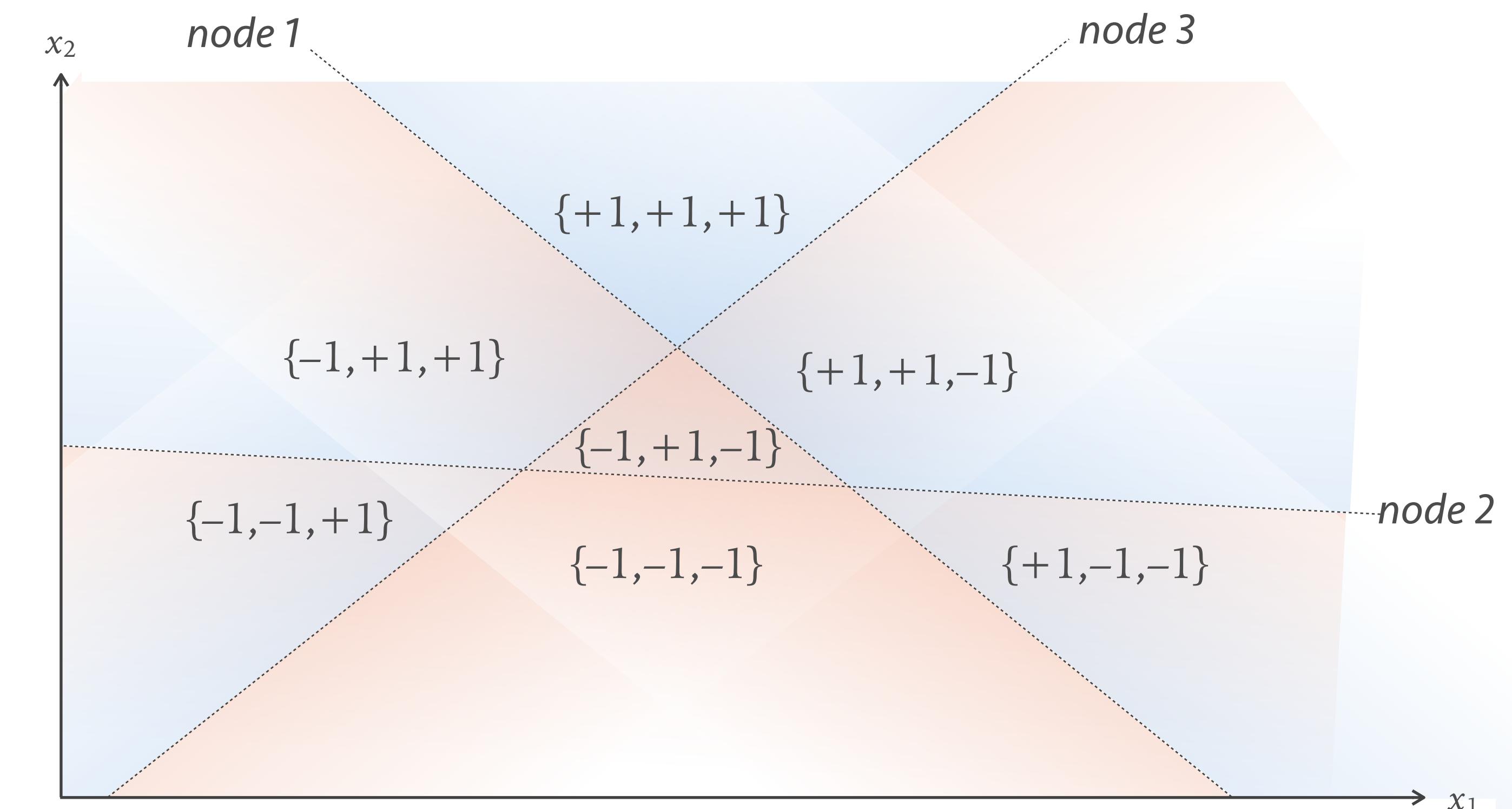
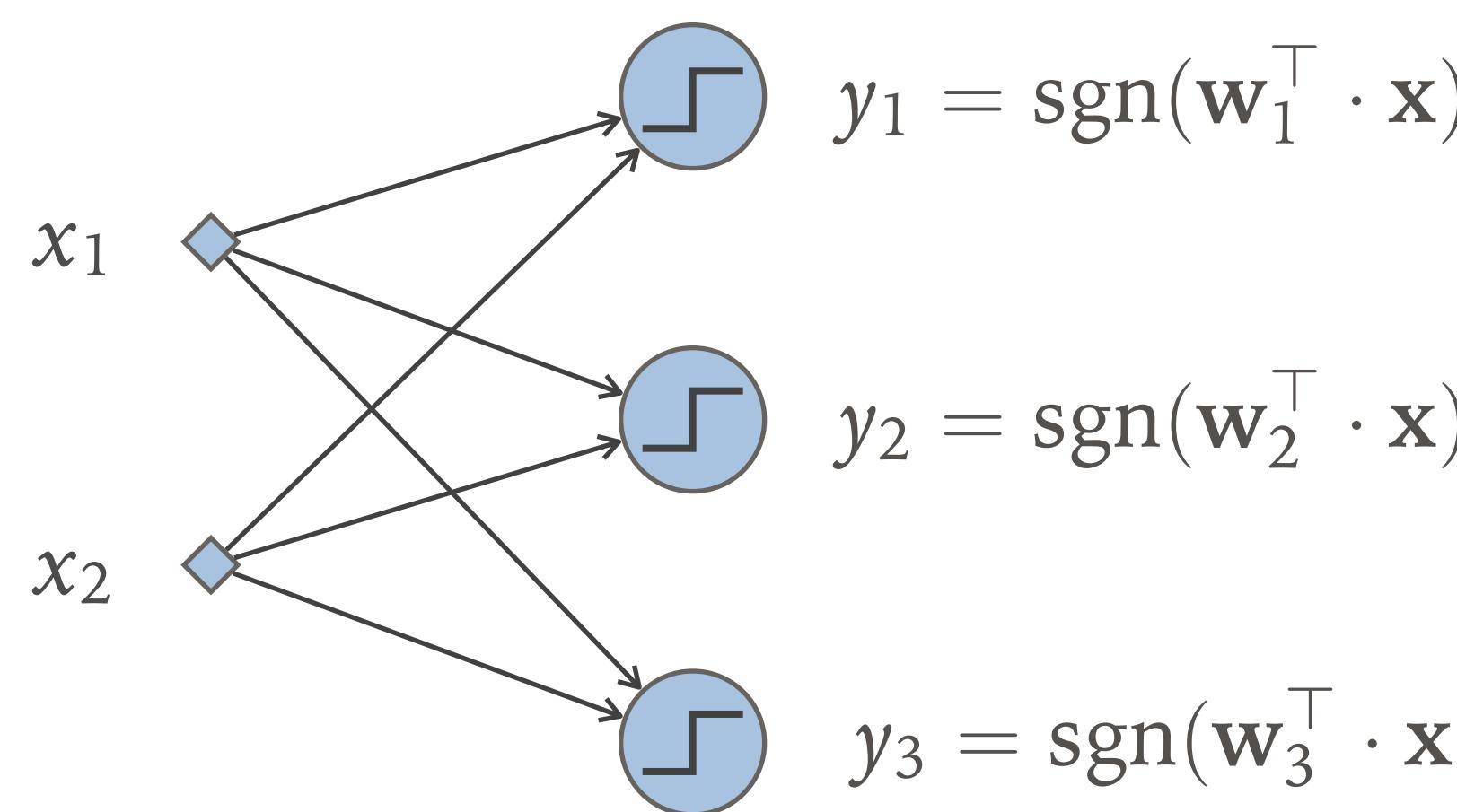
# Terminology

- Two-layer feed-forward neural net



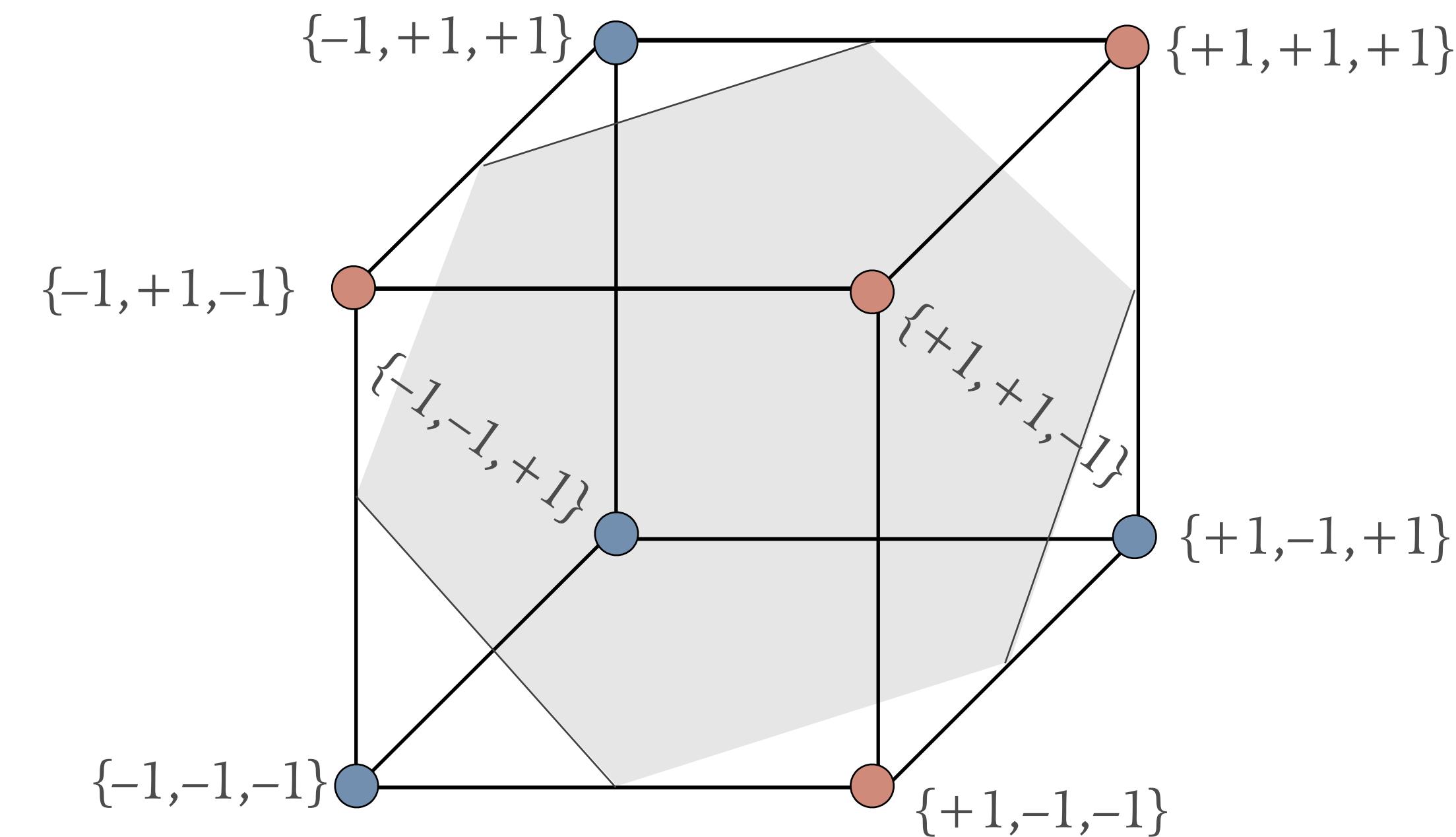
# The hidden layer

- Each node is a decision boundary
  - Maps all inputs onto a binary code



# The output layer

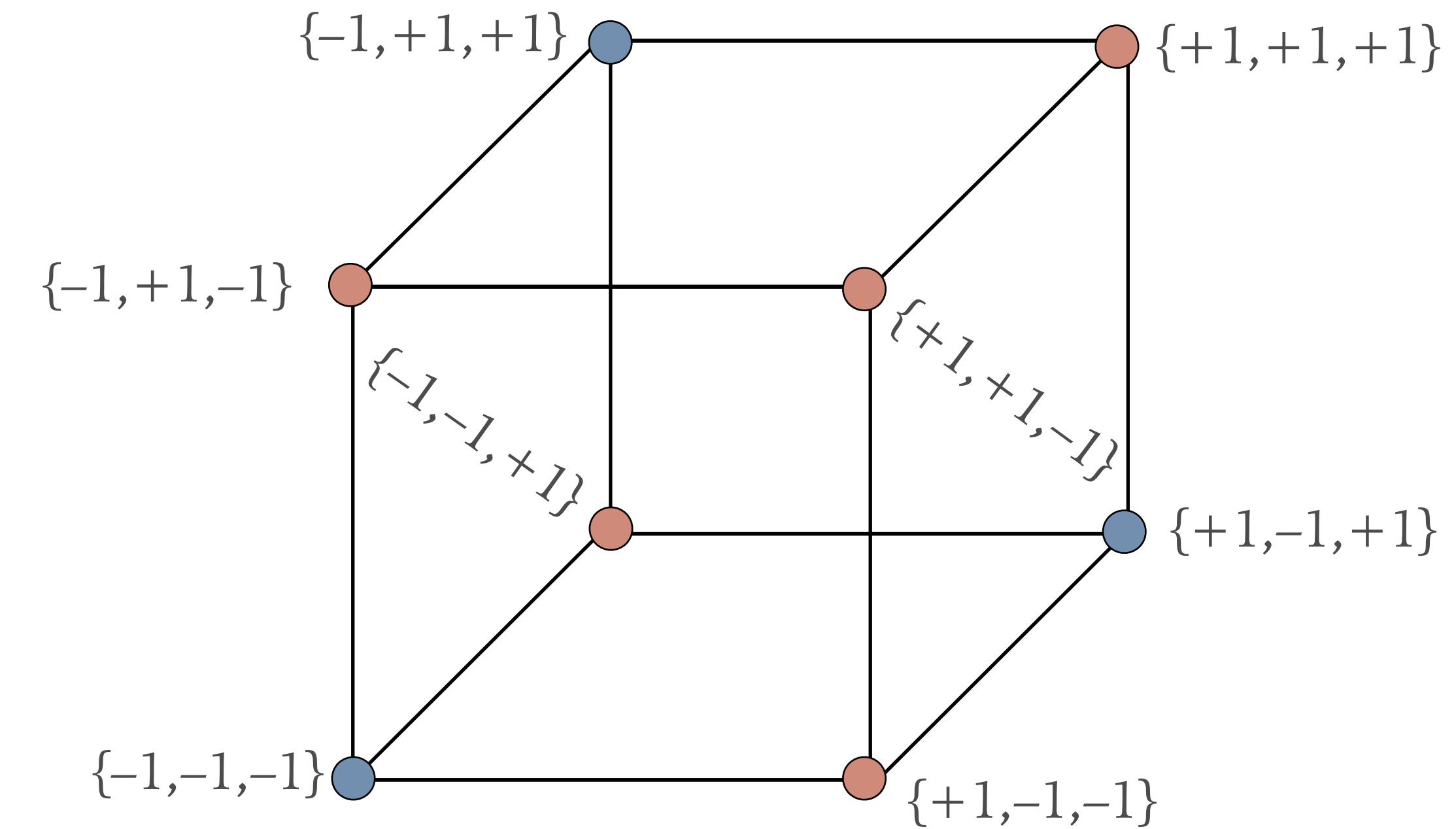
- In the case of a 1-dimensional output layer
  - The output layer node defines a hyperplane
  - Cuts the hypercube to define two classes



*From three hidden-layer nodes to one output*

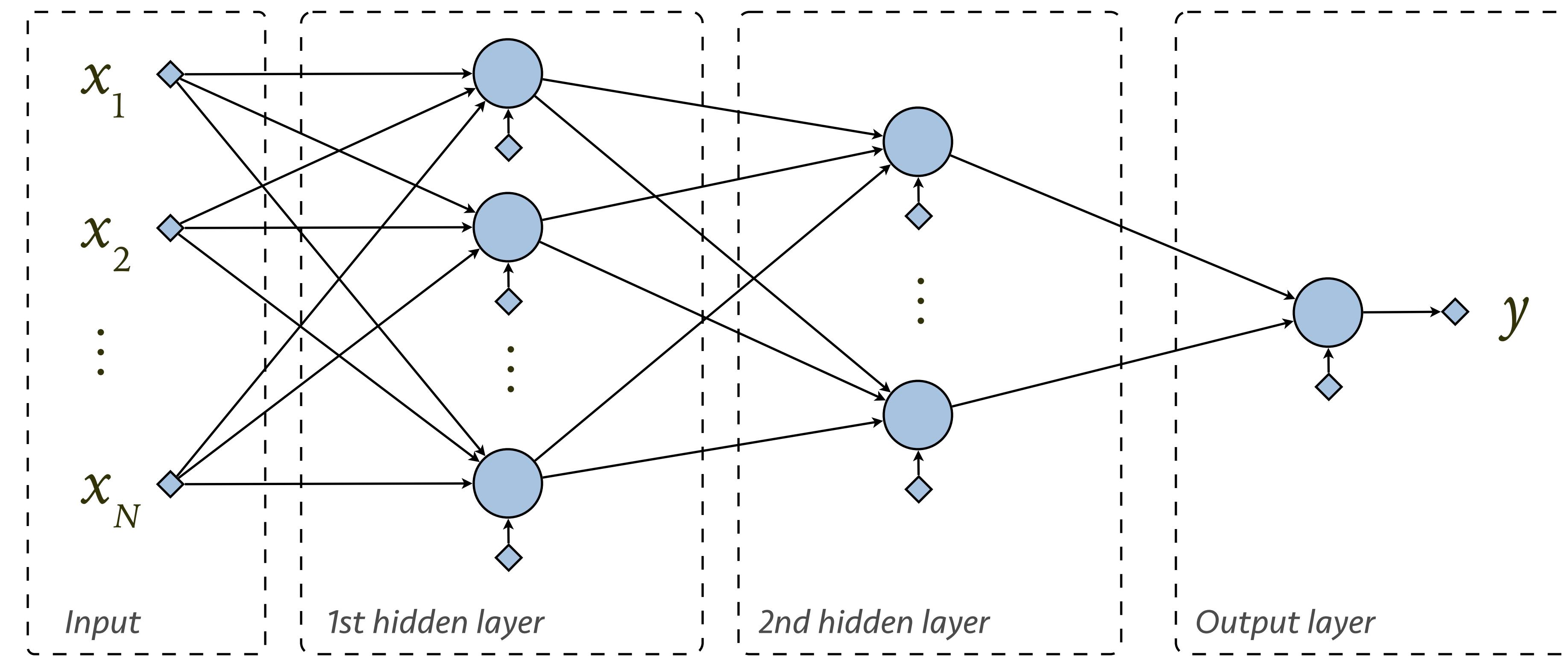
# A new problem

- We can still have a non-separable case
    - This time after the hidden layer



# Piling them on ...

- The three-layer feed-forward network



# Three-layer nets

- Three layers can be sufficient
  - First layer maps data on a hypercube
  - If non-separable, second layer takes care of it
  - Final layer makes an easy linear decision
    - Any more layers are redundant

# That's nice, but ...

- We have a system that can solve hard problems
  - But we now have a new problem!
- How do we get the desired weight values?
  - This is not a simple perceptron
  - This is a highly non-linear system

# Defining the process

- Each layer has:
  - 1. A weighted sum with an offset term (bias)
  - 2. An *activation* function applied on it
- Layer process is described by:

$$\mathbf{y}_l = g\left(\mathbf{W}_l \cdot \mathbf{y}_{l-1} + \mathbf{w}_l\right)$$

# Neural network learning

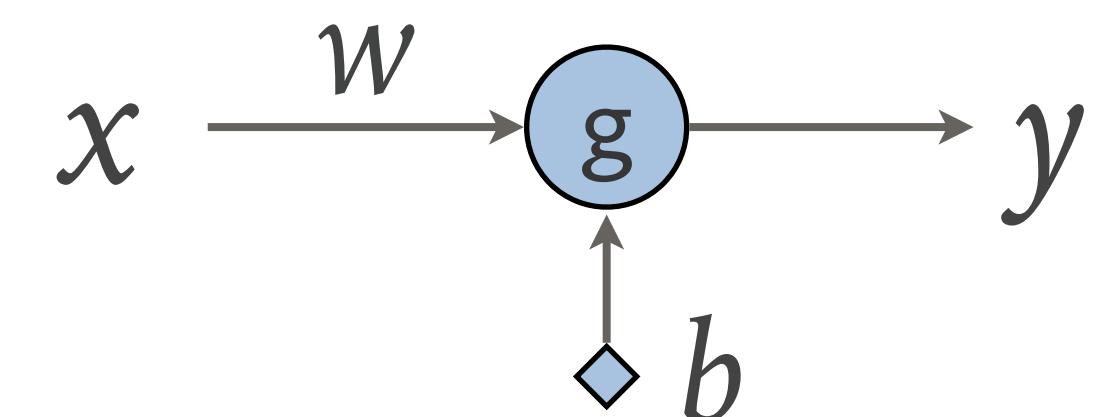
- Define inputs/outputs

$$\mathbf{x}_i \rightarrow \mathbf{t}_i$$

- e.g., data samples and class labels
- Iterative training
  - Pass data through network
  - Measure error at output
  - Adjust weights to minimize error

# Starting simple

- 1-layer, 1-input, 1-output:  $y = g(w \cdot x + b)$
- For a target  $t$  the error is:  $\frac{1}{2}(y - t)^2$
- Start with random  $w$  and  $b$  and use gradient descent updates to revise them:

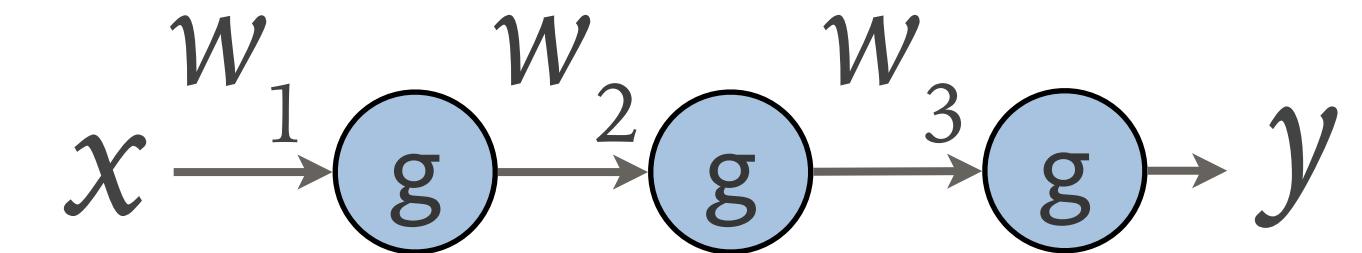


$$\Delta w \propto x \left( g(w \cdot x + b) - t \right) g'(w \cdot x + b)$$

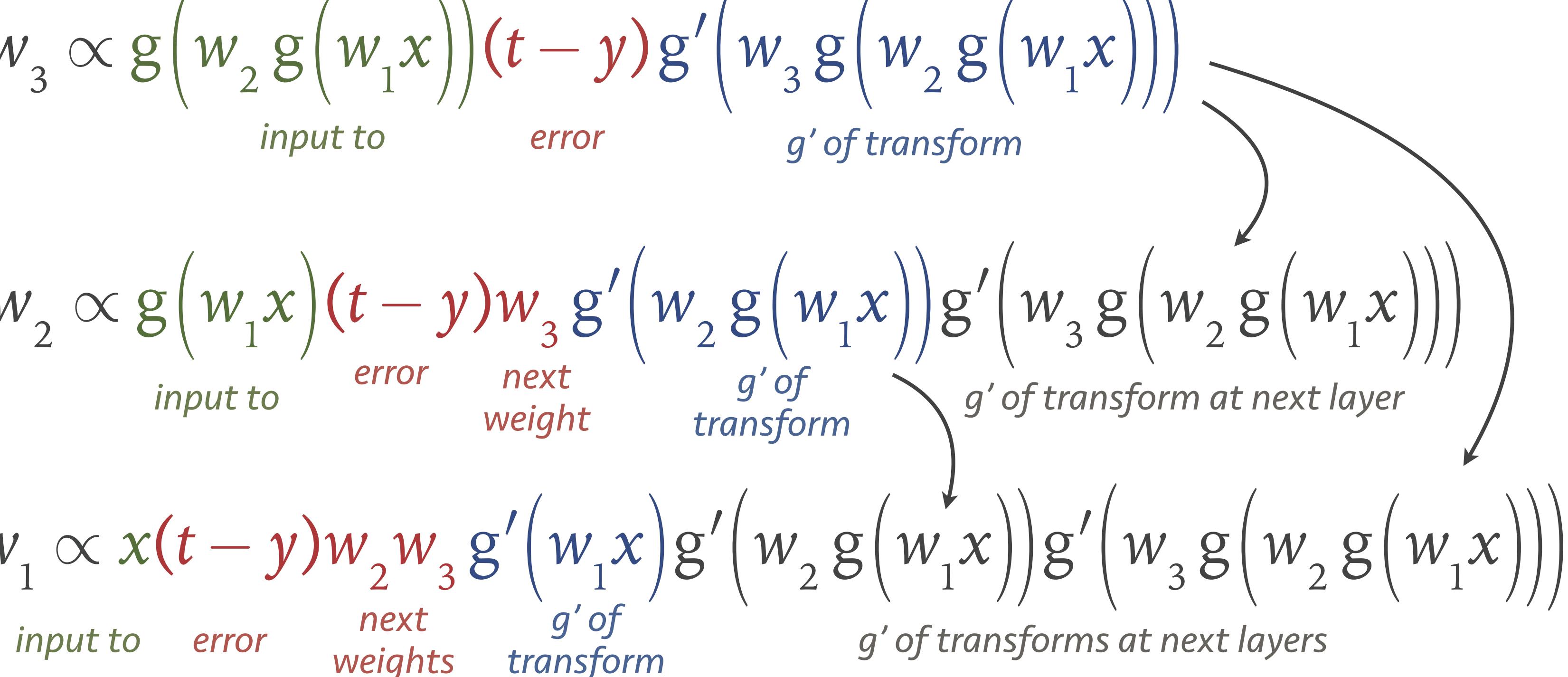
$$\Delta b \propto \left( g(w \cdot x + b) - t \right) g'(w \cdot x + b)$$

# Adding a couple of layers

- We now have:  $y = g\left(w_3 g\left(w_2 g\left(w_1 x\right)\right)\right)$
- We skip the bias terms for sanity's sake*



- Update for  $w_3$ :  $\Delta w_3 \propto g\left(w_2 g\left(w_1 x\right)\right)(t - y)g'\left(w_3 g\left(w_2 g\left(w_1 x\right)\right)\right)$
- Update for  $w_2$ :  $\Delta w_2 \propto g\left(w_1 x\right)(t - y)w_3 g'\left(w_2 g\left(w_1 x\right)\right)g'\left(w_3 g\left(w_2 g\left(w_1 x\right)\right)\right)$
- Update for  $w_1$ :  $\Delta w_1 \propto x(t - y)w_2 w_3 g'\left(w_1 x\right)g'\left(w_2 g\left(w_1 x\right)\right)g'\left(w_3 g\left(w_2 g\left(w_1 x\right)\right)\right)$



# The backpropagation algorithm

- Initialize first layer:

$$\mathbf{y}_0 = \mathbf{x}$$

- Pass data through network:

$$\mathbf{y}_l = g(\mathbf{W}_l \cdot \mathbf{y}_{l-1} + \mathbf{b}_l)$$

- Calculate error in output:

$$\delta_L = \mathbf{t} - \mathbf{y}_L$$

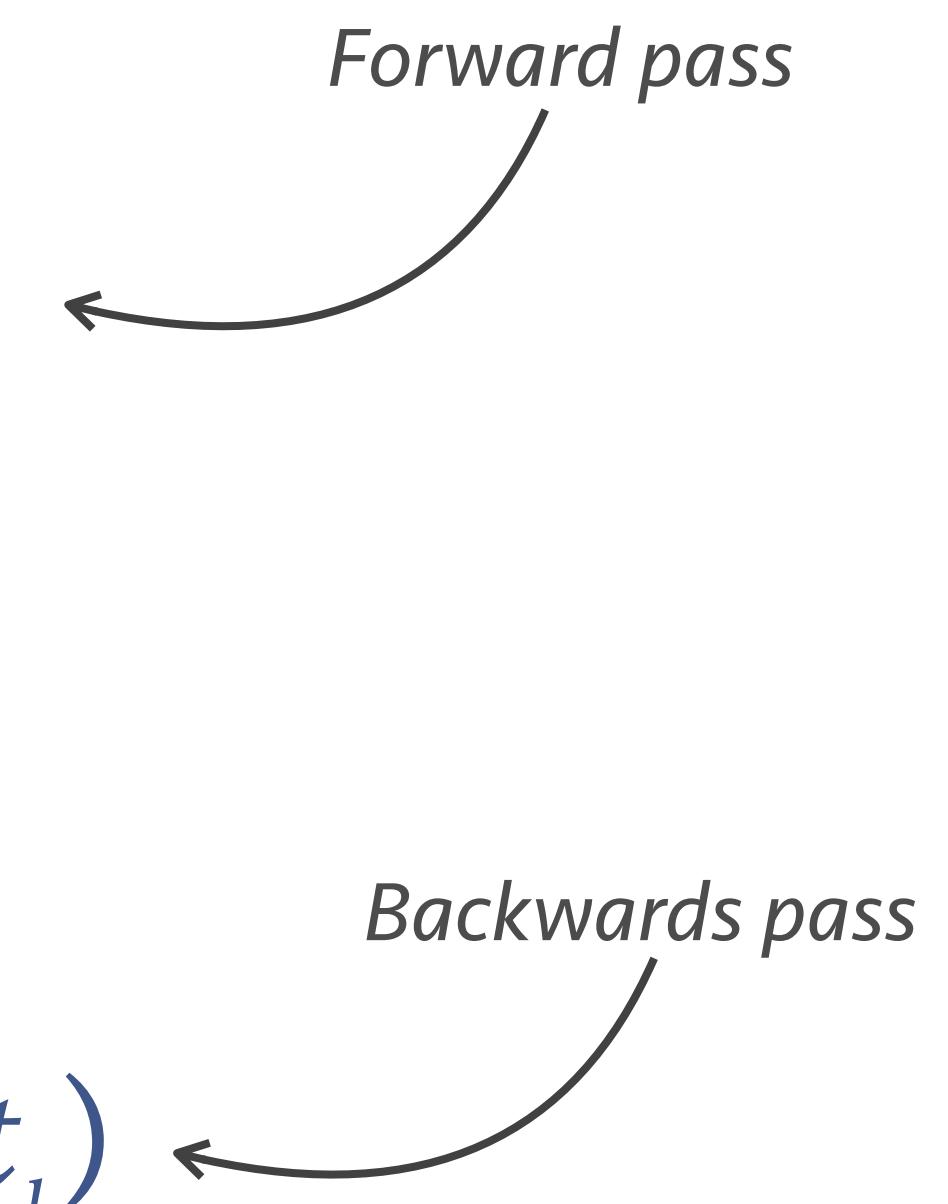
- *Backpropagate* error derivative:

$$\delta_l = (\mathbf{W}_{l+1}^\top \cdot \delta_{l+1}) g'(\text{net}_l)$$

- Adjust weights/biases:

$$\Delta \mathbf{W}_l = \delta_l \cdot \mathbf{y}_{l-1}^\top$$

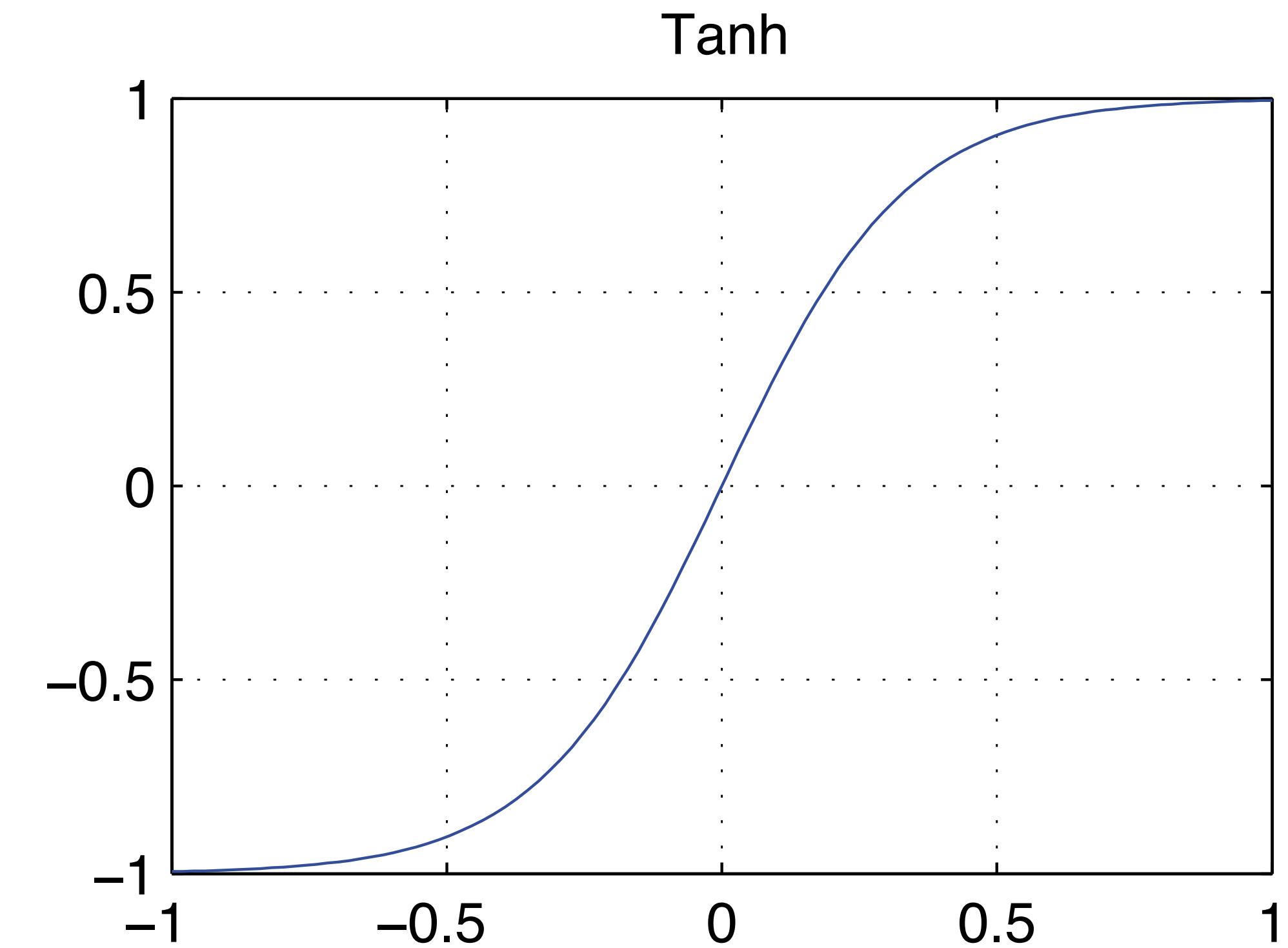
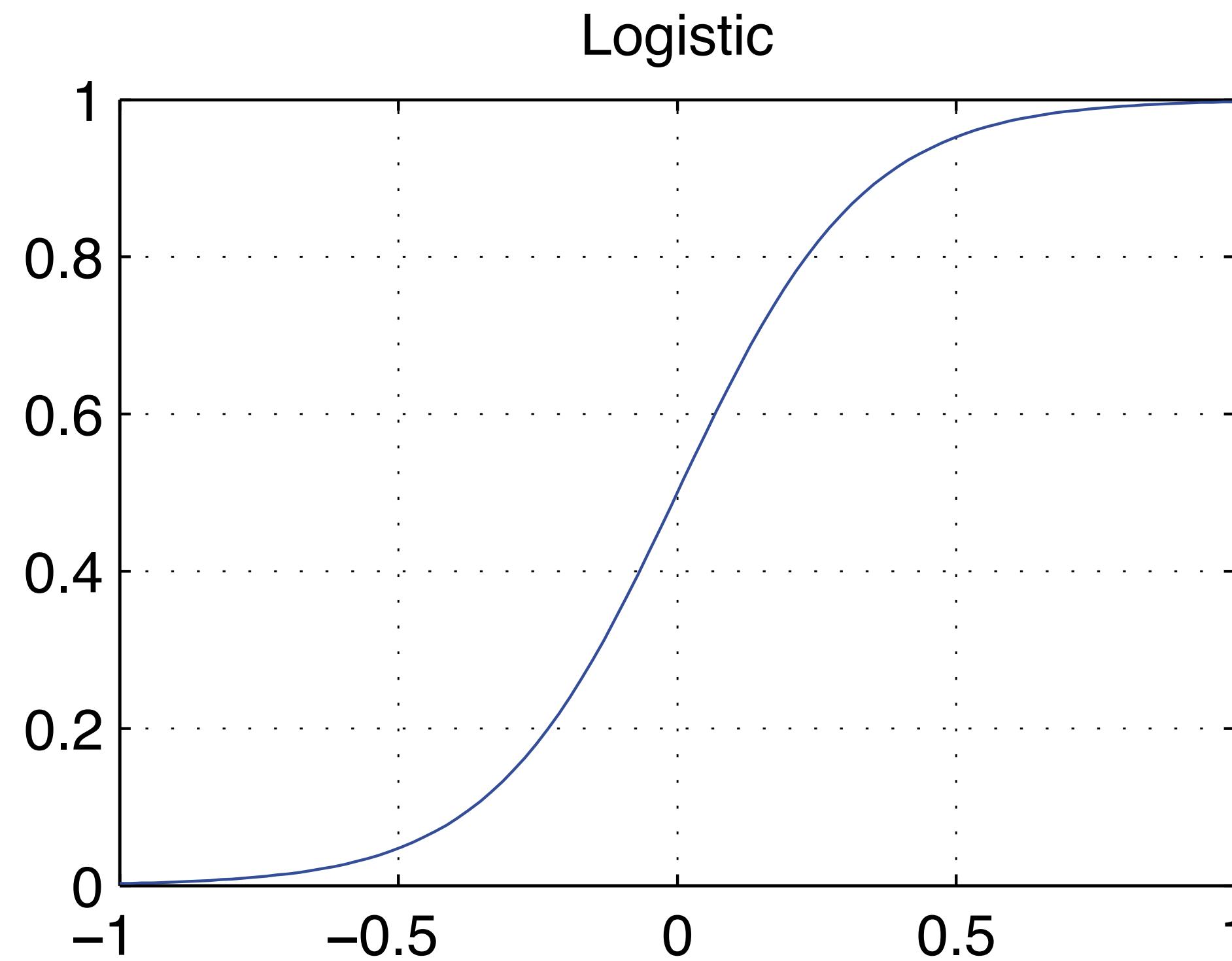
$$\Delta \mathbf{b}_l = \delta_l$$



# One minor issue

- We need a differentiable  $g()$ 
  - The sign function is not going to work here
  - We instead “soften” it and use a sigmoid function
- Original traditional choices are:
  - The logistic:  $g(x) = \left(1 + e^{-ax}\right)^{-1}$
  - The hyperbolic tangent:  $g(x) = \tanh(ax)$

# Sigmoids (beware of the output range!)

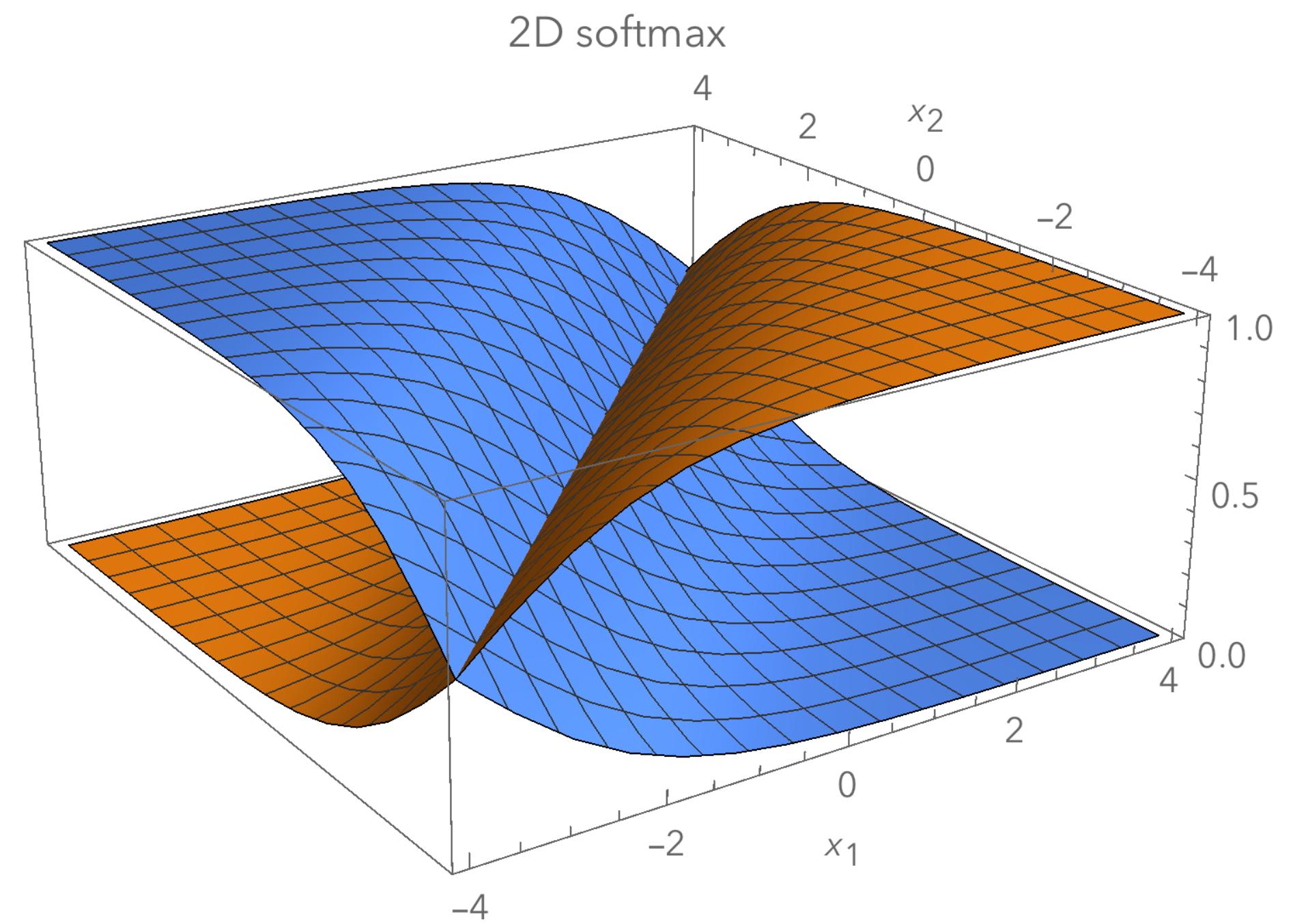


# The softmax activation

- If you want the outputs to be class “probabilities”
  - i.e. should sum to 1 and be inside [0,1]

$$g(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum e^{x_i}}$$

- Good activation choice for classification problems



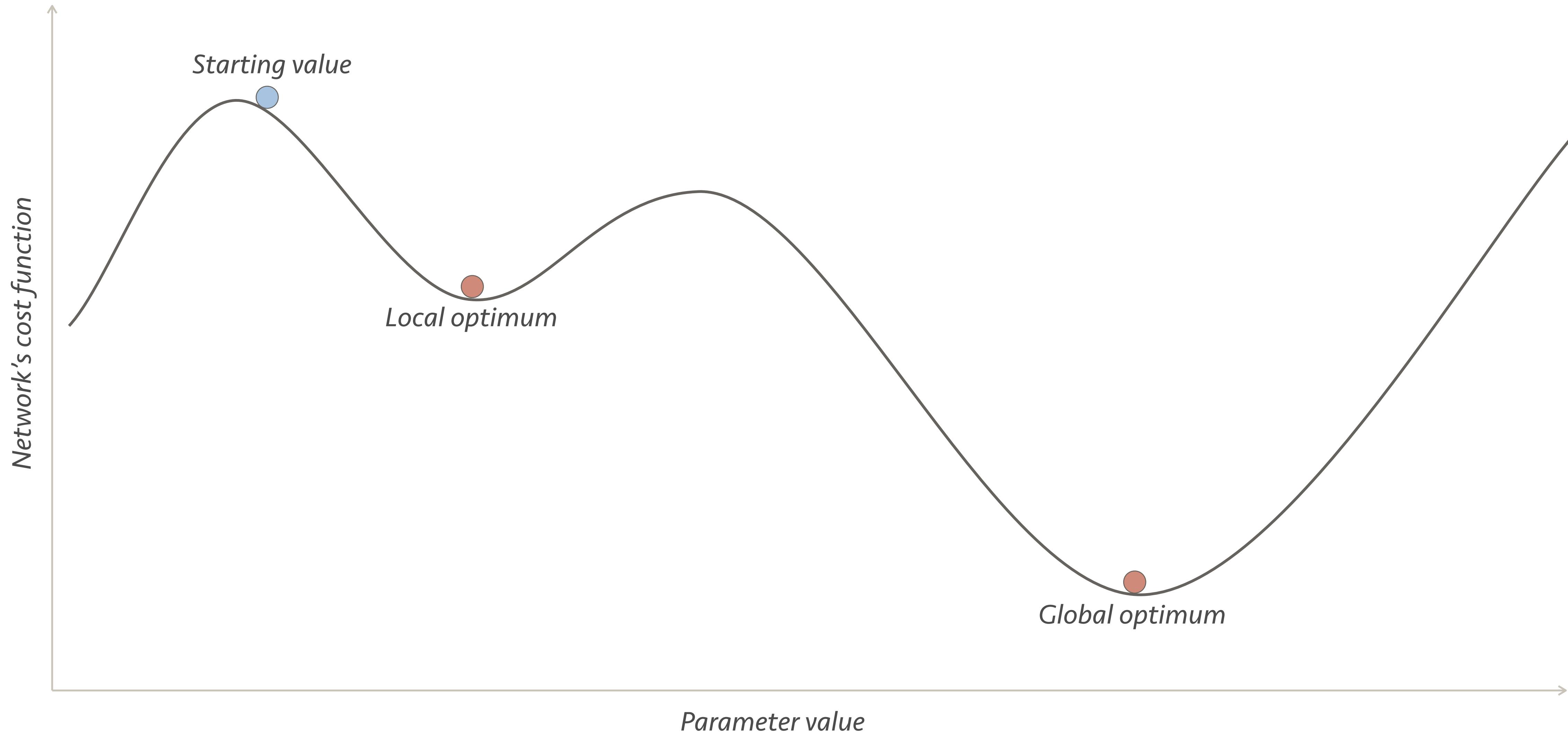
# Practical matters

- We don't just want to use the gradient as is
- We use a *learning rate*

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \Delta \mathbf{W}_t$$

- Often we also use a *momentum* term
  - we can adapt the learning rate
    - we can perform a line search
    - ...
- Eventually training becomes an art ...

# Cartoon case to keep in mind



# Many more extensions

- Various cost functions
  - Mean squared error is not always best
- Various optimization schemes
  - Many variants of backpropagation
  - More advanced local searches
- More later when we talk about deep learning

# The bigger picture

- Neural nets are just a nonlinear regression
  - $y_i = g(\mathbf{x}_i)$ 
    - $g()$  is arbitrarily defined from the layers/activations
  - So we effectively extended the linear regression classifier
- That means that we can also approximate nonlinear functions, not just perform classification (more later)

# Some criticism that killed neural nets

- Neural nets can be a black box
  - What do the weights mean?
  - How big should the network be?
- Training is cumbersome
  - Too many tricks to make things work
  - How long do you train for?
  - Not trivial for huge data sets

# The revival

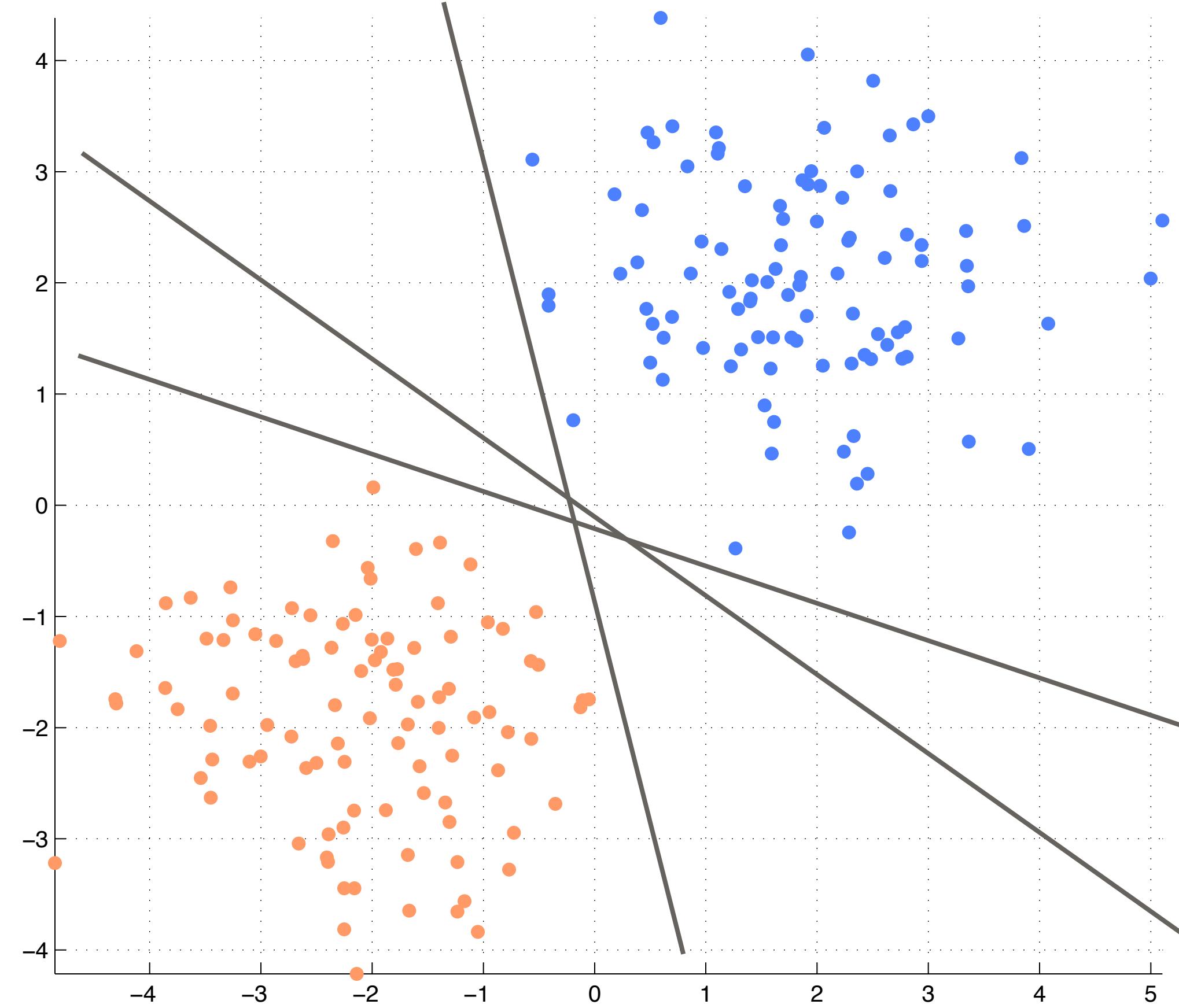
- “Deep learning” architectures
  - Smarter local pre-training that improves accuracy
  - Better training strategies, activations, and topologies
- Relatively tractable training
  - Thanks NVIDIA!
- Superb results in a range of applications
  - More in later lectures on deep learning

# A different approach

- Support vector machines
  - Start from an algebraic perspective
  - Generalize to non-linear classification
- Formidable competition to earlier neural nets
  - Eventually displaced them as a tool of choice in the late 90's
    - Only to be later displaced by neural nets later on ...
      - Pay attention, they might strike back!

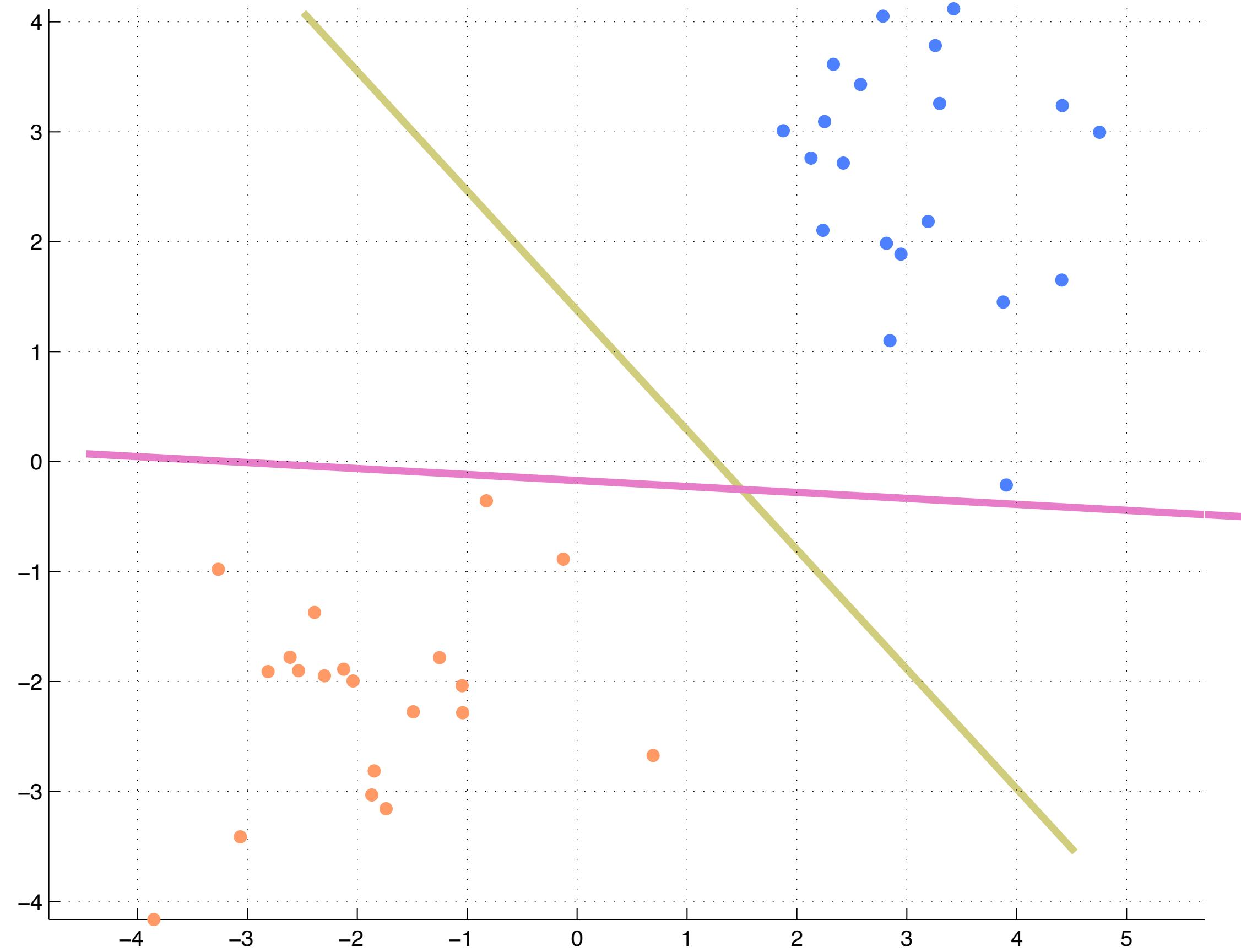
# Which line is best?

- Linear classifiers were not very well defined so far
  - Many possibilities
  - Which one is best?
  - We need a better definition!



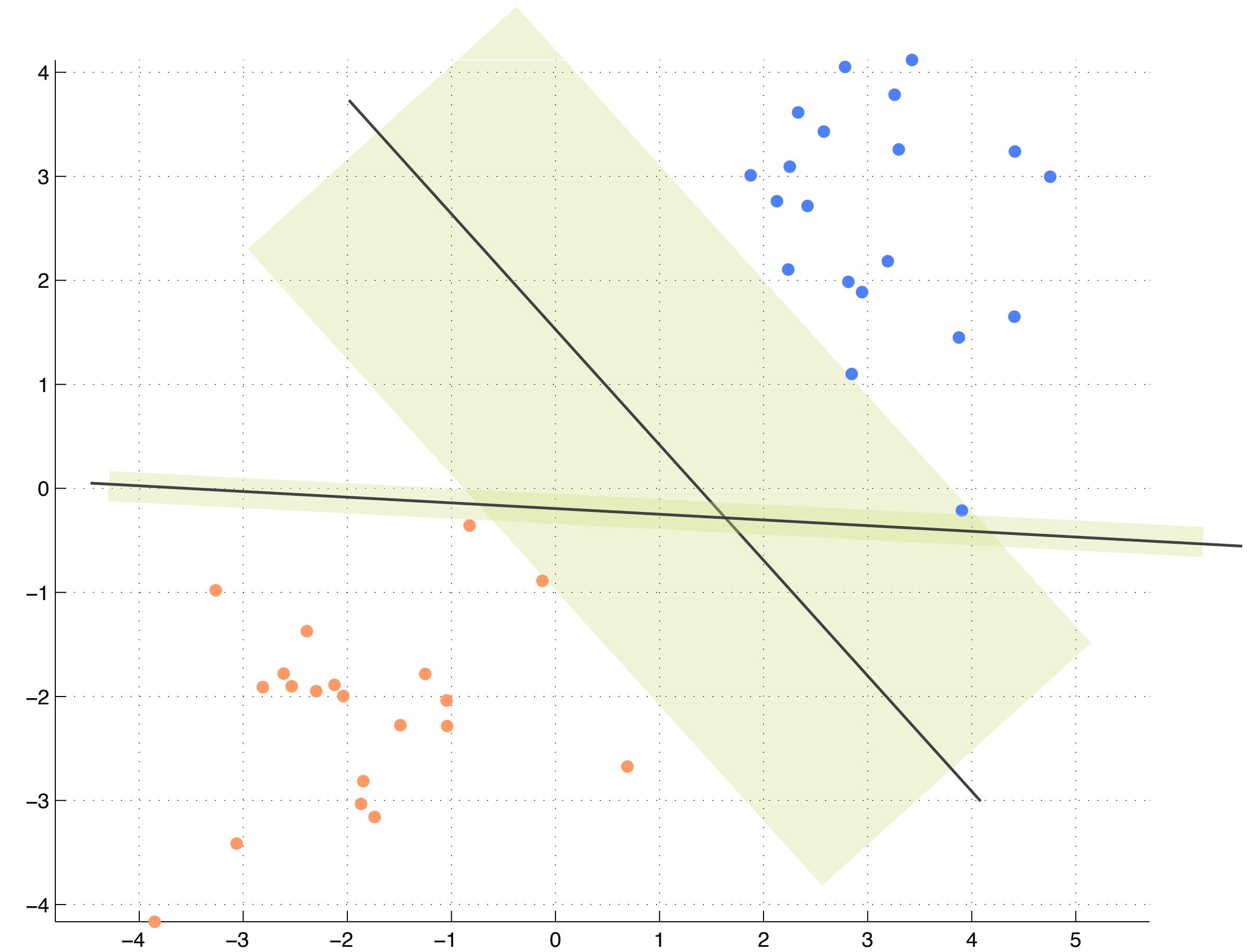
# Which one is best?

- And why?



# Maximum margin classifiers

- Maximize distance between decision line and nearest training points
- Provides maximum slack for new data to classify



# Defining the margin

- Our decision hyperplane is:

$$g(\mathbf{x}) = \mathbf{w}^\top \cdot \mathbf{x} + w_0 = 0$$

- The distance of a point  $\mathbf{x}$  from the decision boundary is:

$$z = \frac{|g(\mathbf{x})|}{\|\mathbf{w}\|}$$

- We want this to be maximized for the  $\mathbf{x}$ 's that are nearest to the decision boundary

# Redefining the classifier

- Assume a linearly separable case
- We now want the constraint:

$$\mathbf{w}^\top \cdot \mathbf{x} + w_0 \geq +1, \quad \forall \mathbf{x} \in \omega_1$$

$$\mathbf{w}^\top \cdot \mathbf{x} + w_0 \leq -1, \quad \forall \mathbf{x} \in \omega_2$$

- In addition to having a maximally large margin
  - Which is proportional to  $1/\|\mathbf{w}\|$

# Defining the problem

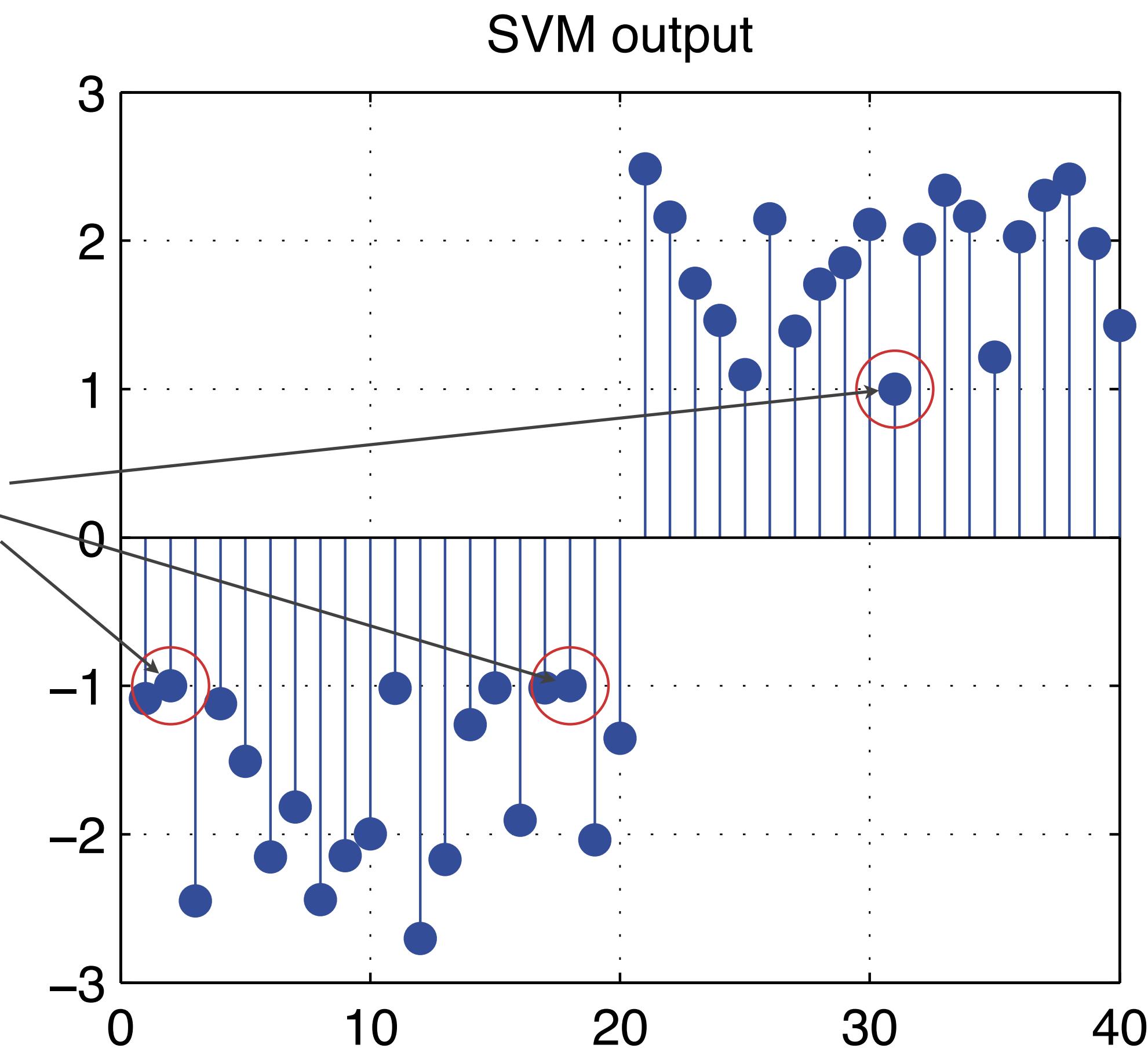
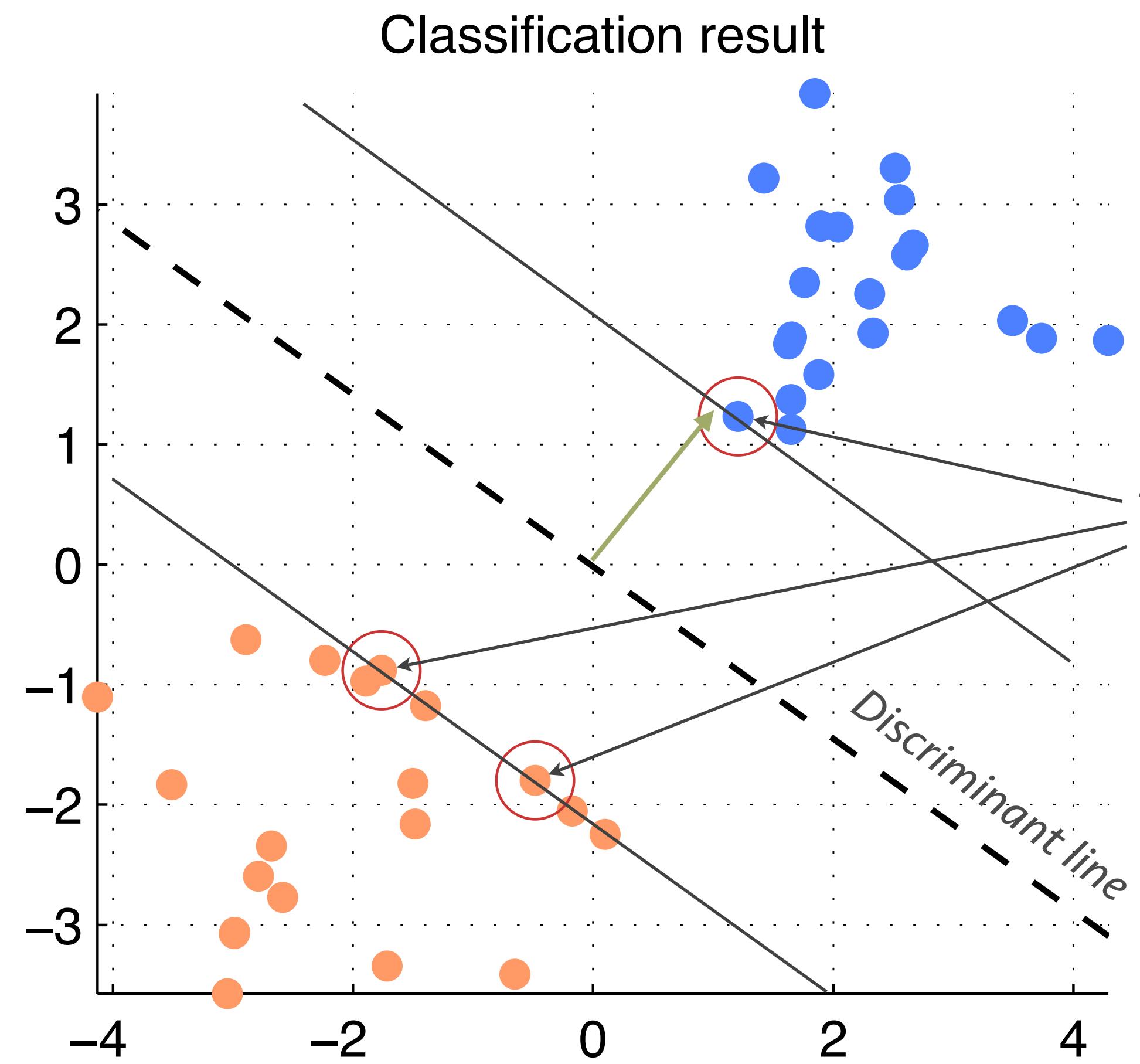
- Quadratic optimization

$$\text{minimize } J(\mathbf{w}, w_0) = \frac{1}{2} \|\mathbf{w}\|^2 \quad \text{i.e. maximize margin}$$

$$\text{subject to } y_i (\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) \geq 1 \quad \text{i.e. classify correctly}$$

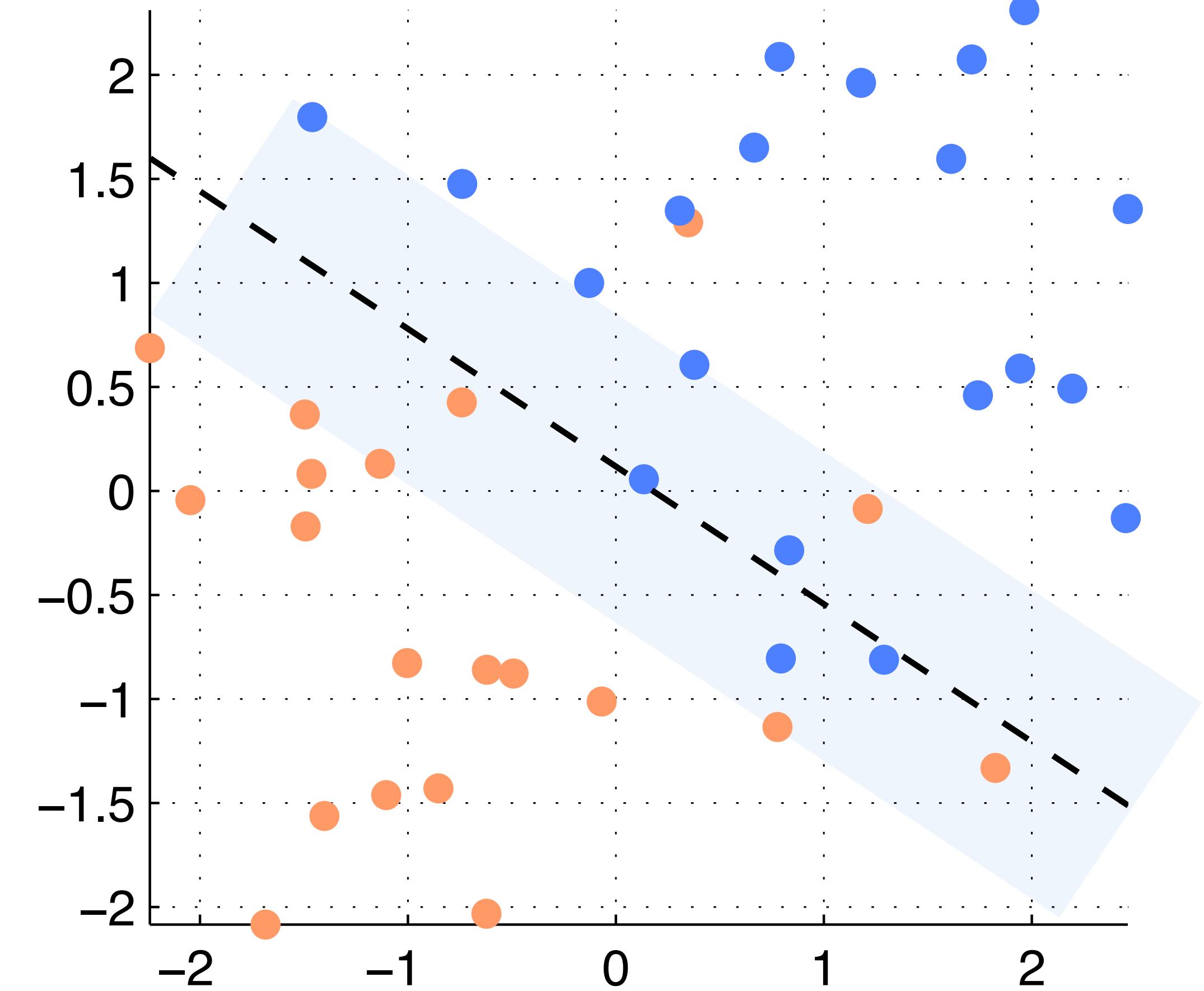
- We can solve this with quadratic solvers
  - e.g., MATLAB's or Python's quadprog

# Example run



# Non-Separable case

- Constraints might be infeasible
- Three kinds of points
  - well classified  $y_i(\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) \geq 1$
  - within margin  $0 < y_i(\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) < 1$
  - misclassified  $y_i(\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) < 0$
- All cases can be written as:
 
$$y_i(\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) \geq 1 - \xi_i$$
  - $\xi_i \geq 0$ , are called *slack variables*



# New optimization problem

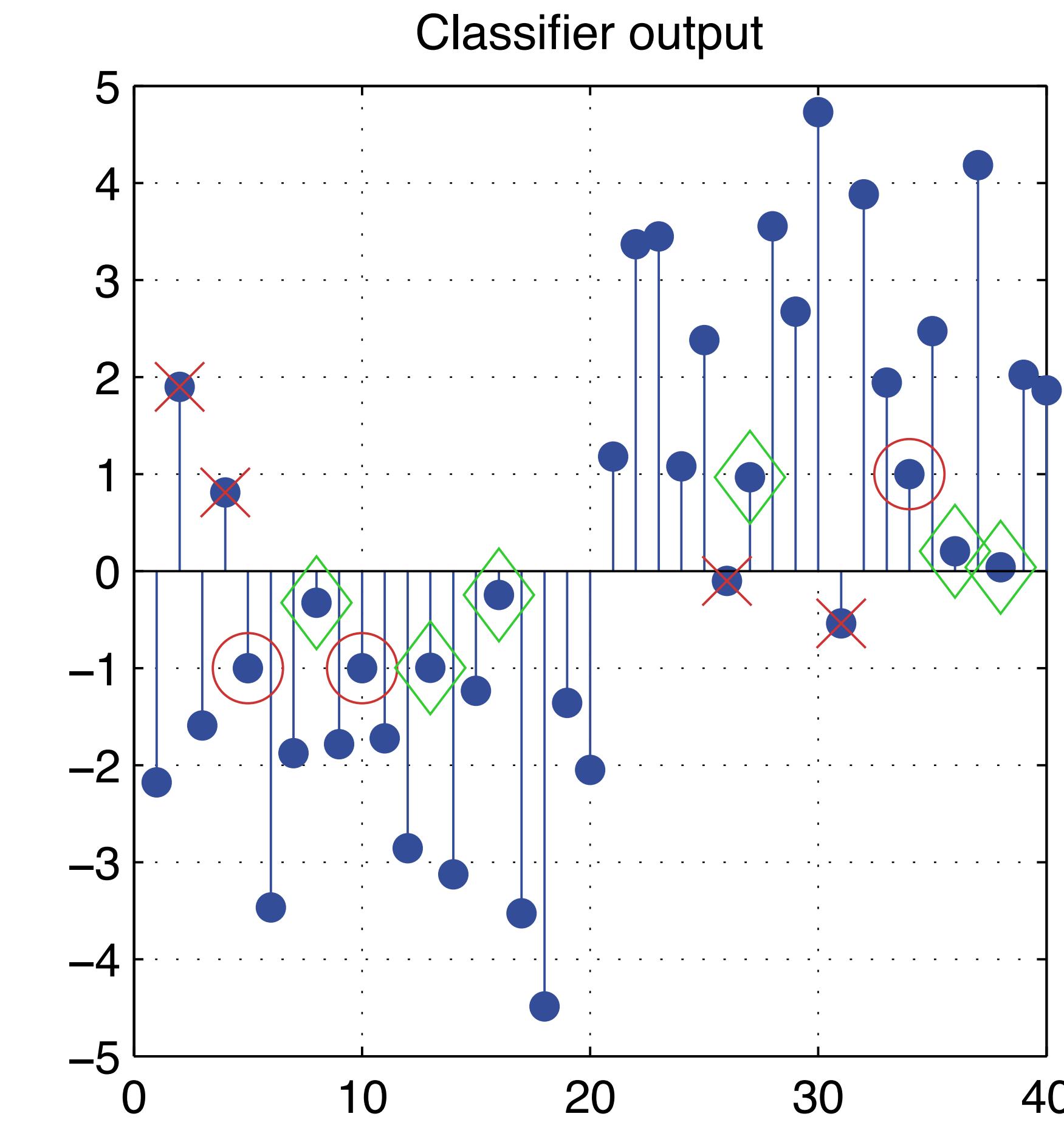
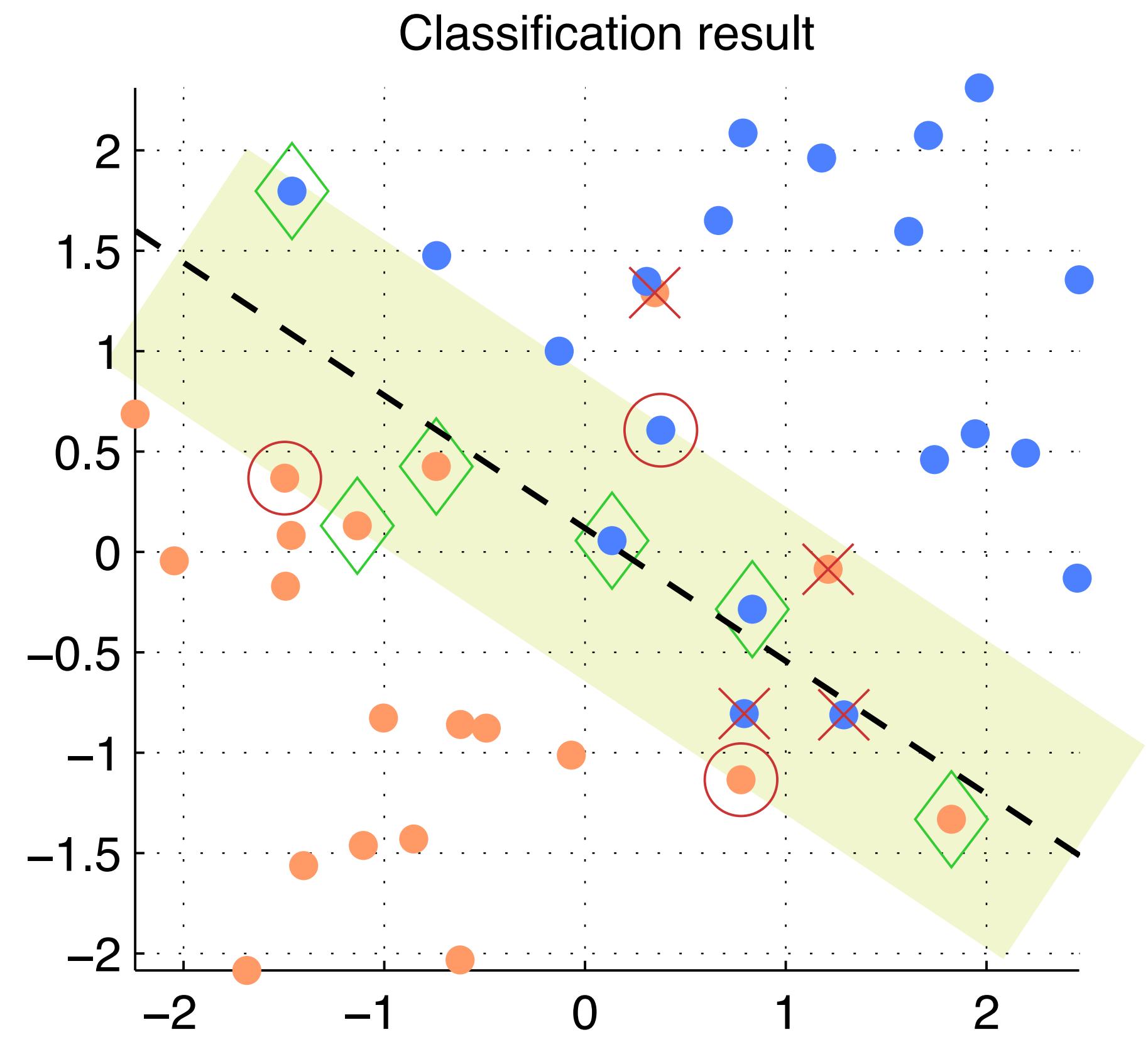
- Maximize margin and minimize slack
- Once again a quadratic program

$$\text{minimize } J(\mathbf{w}, w_0, \xi) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum \xi_i$$

$$\text{subject to } y_i (\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

# Example run

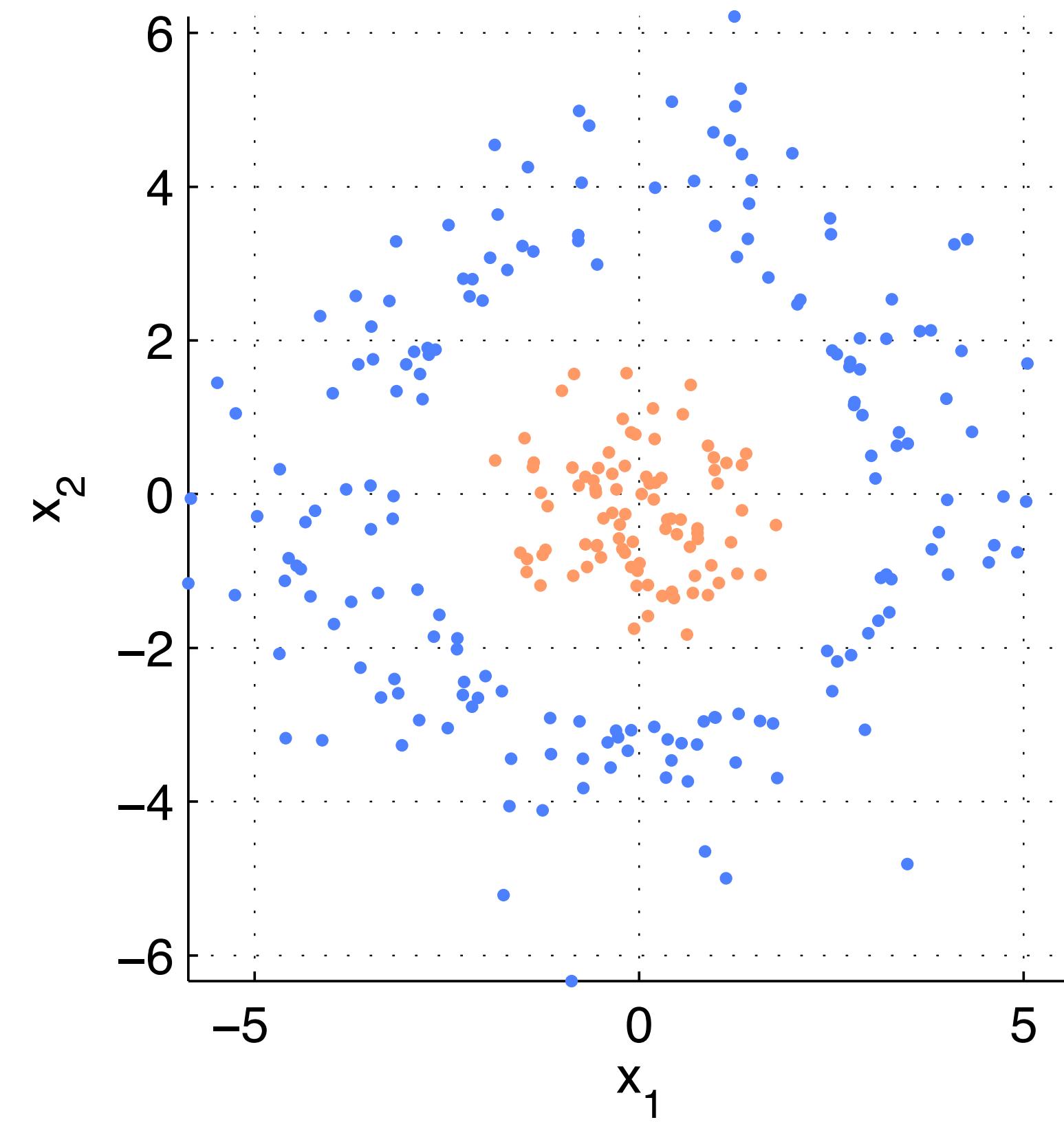


# This can be slow to solve

- There are also fast alternatives
  - E.g. when you have a ton of data
- PEGASOS algorithm:
  - 1) Pick  $k$  samples at random
  - 2) Get failure cases where  $y_i(\mathbf{w}^\top \cdot \mathbf{x}_i) < 1$
  - 3) Tweak weights  $\mathbf{w} \leftarrow (1 - h\lambda)\mathbf{w} + (h/k)\mathbf{x}_i y_i^\top$ 
$$\mathbf{w} \leftarrow \min \left\{ 1, \frac{1/\lambda}{\|\mathbf{w}\|} \right\} \mathbf{w}$$
- Essentially a sub-gradient approach

# But we should do better than that

- Some non-linearly separable problems look “separable”
  - We need non-linear discriminant functions

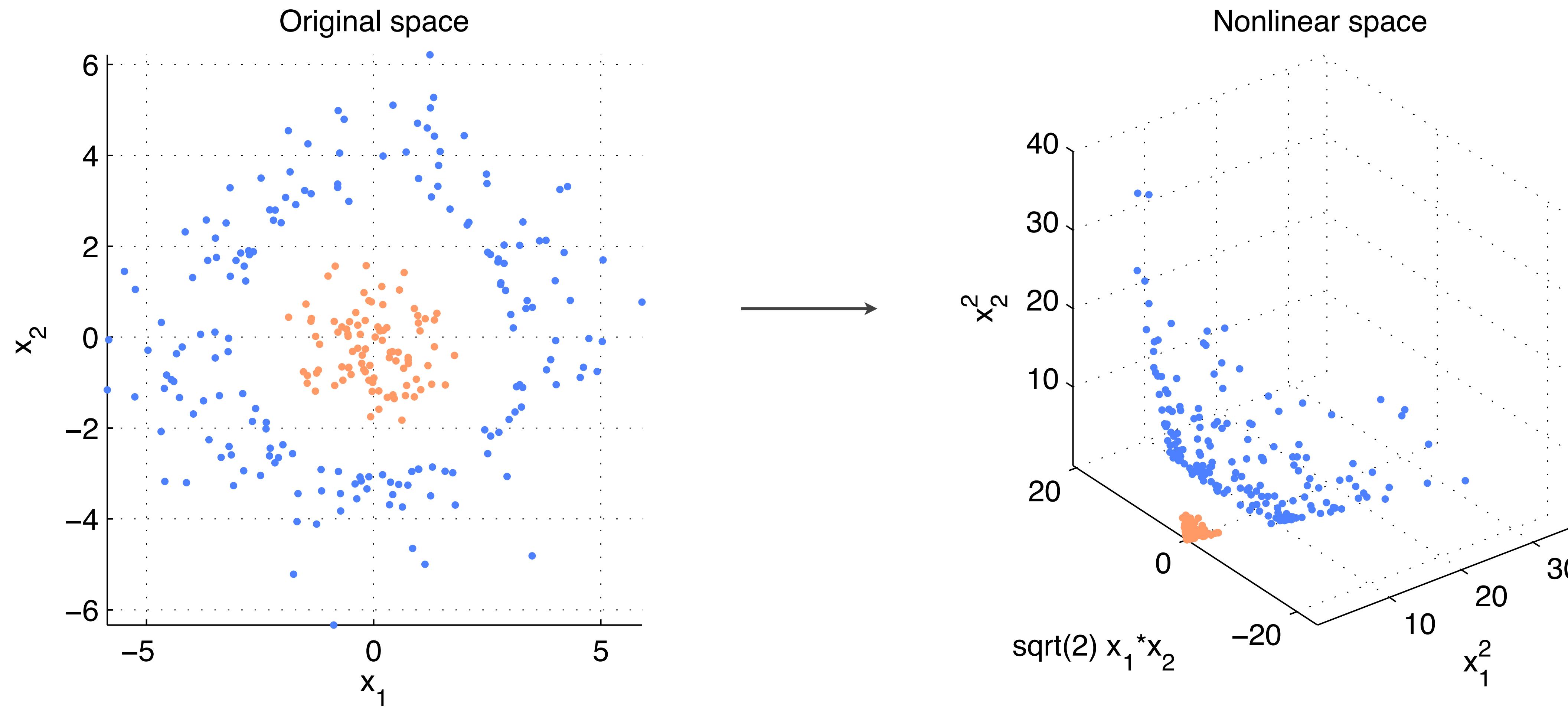


# Kernel SVMs

- Remember Kernel PCA?
  - We non-linearly map our variables to a higher dimensional space to simplify the problem
- We will do the same thing for classification
  - Example case:

$$\mathbf{x} \in \mathbb{R}^2 \rightarrow \mathbf{z} = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

# What does this map do?



# But there's a problem ...

- The projected data will lie in a high-D space
  - That will require more computations
  - And potentially result in ill-defined estimates

$$\text{minimize } J(\mathbf{w}, w_0, \xi) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum \xi_i$$

$$\text{subject to } y_i (\mathbf{w}^\top \cdot \mathbf{z}_i + w_0) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

*This can be very high dimensional!*

# Avoiding $w$

- We want to avoid using  $w$  because it is high dimensional
  - Instead we use it implicitly
- Note that  $w$  can be defined from the training data

$$w = \sum_i y_i a_i z_i$$

- We can now write the classification process as:

$$g(z_j) = w^\top \cdot z_j + w_0 = \left( \sum_i y_i a_i z_i \right)^\top \cdot z_j + w_0 = \sum_i y_i a_i (z_i^\top \cdot z_j) + w_0$$

# A new formulation of the problem

- The margin we minimized  $\|\mathbf{w}\|^2$ , is now:

$$\|\mathbf{w}\|^2 = \left( \sum_i y_i a_i \mathbf{z}_i \right)^\top \cdot \left( \sum_j y_j a_j \mathbf{z}_j \right) = \sum_{i,j} y_i y_j a_i a_j (\mathbf{z}_i^\top \cdot \mathbf{z}_j)$$

- And the classification loss is now:

$$y_i g(\mathbf{z}_j) = y_j \sum_i y_i a_i (\mathbf{z}_i^\top \cdot \mathbf{z}_j) + w_0 \geq 1, \forall j$$

- So now we seek to find the optimal  $a$ 's, not a  $\mathbf{w}$

# The dual problem

- A formulation without the  $w$  vector, still a quadratic program:

$$\max_a \left[ \sum a_i - \frac{1}{2} \sum a_i a_j y_i y_j \mathbf{z}_i^\top \cdot \mathbf{z}_j \right]$$

subject to  $\sum a_i y_i = 0$

$$a_i \geq 0$$

- We only need to estimate the  $a$ 's
  - For  $N$  samples  $\mathbf{z} \in \mathbb{R}^d$  we learn  $N a_i$ 's (as opposed to  $d w_i$ 's)

# And with slack variables

- If we add slack variables the dual problem becomes:

$$\max_a \left[ \sum a_i - \frac{1}{2} \sum a_i a_j y_i y_j z_i^\top \cdot z_j \right]$$

subject to  $\sum a_i y_i = 0$

$$0 \leq a_i \leq C$$

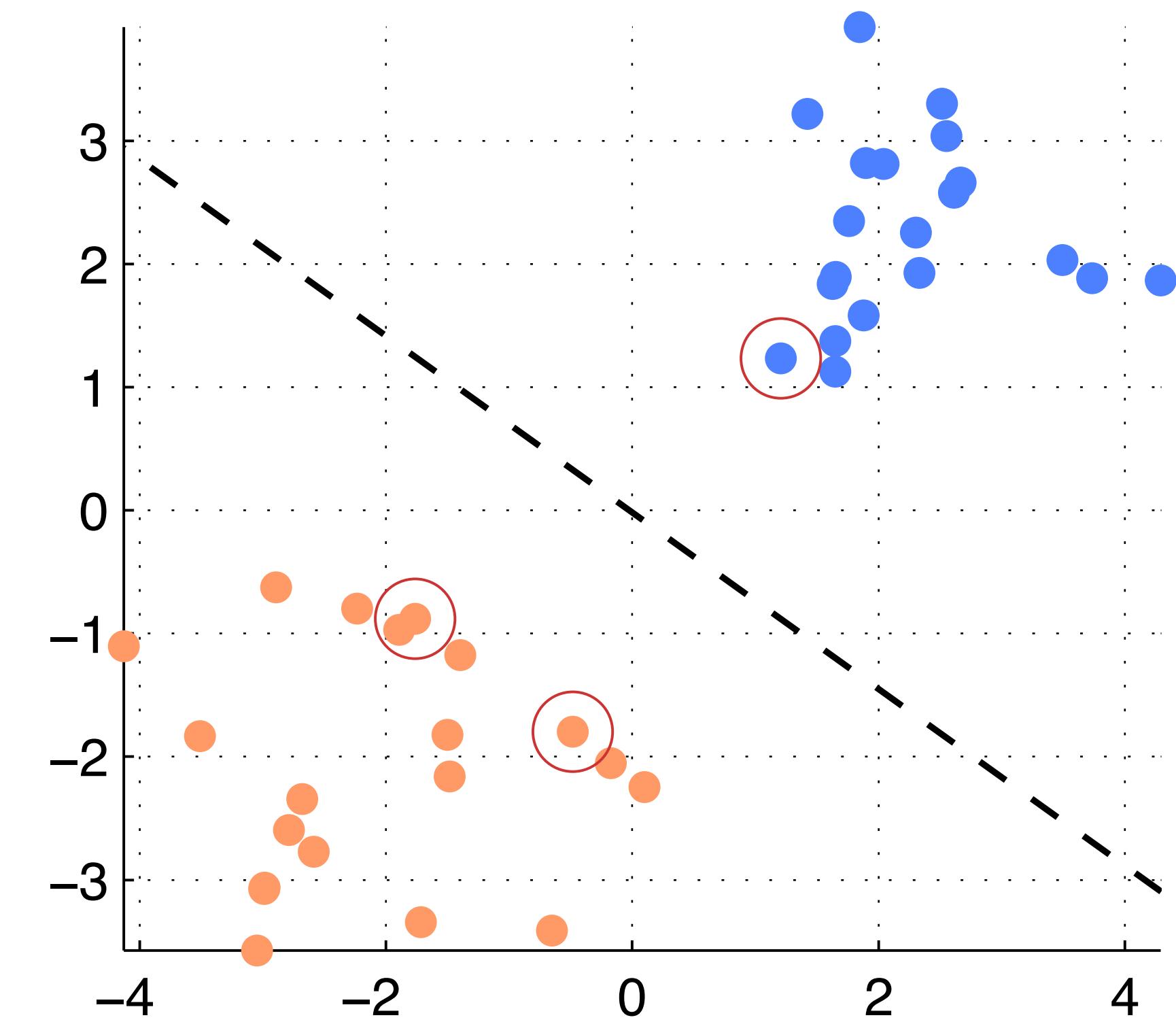
- Slack variables are gone!
  - Upper-bounding  $a_i$  with  $C$  has the same effect
  - Easier optimization, same tradeoffs

# That's not very space efficient!

- Keeping all the  $z_i$ 's takes space!

$$g(z) = \sum_i y_i a_i (z_i^\top \cdot z) + w_0$$

- But we don't need them all, we only need the *support vectors*!
  - $a_i$ 's should be mostly zero
  - Non-zero values select the support vectors that define the discriminant



# But it isn't compute efficient now!

- From the dual formulation we have:

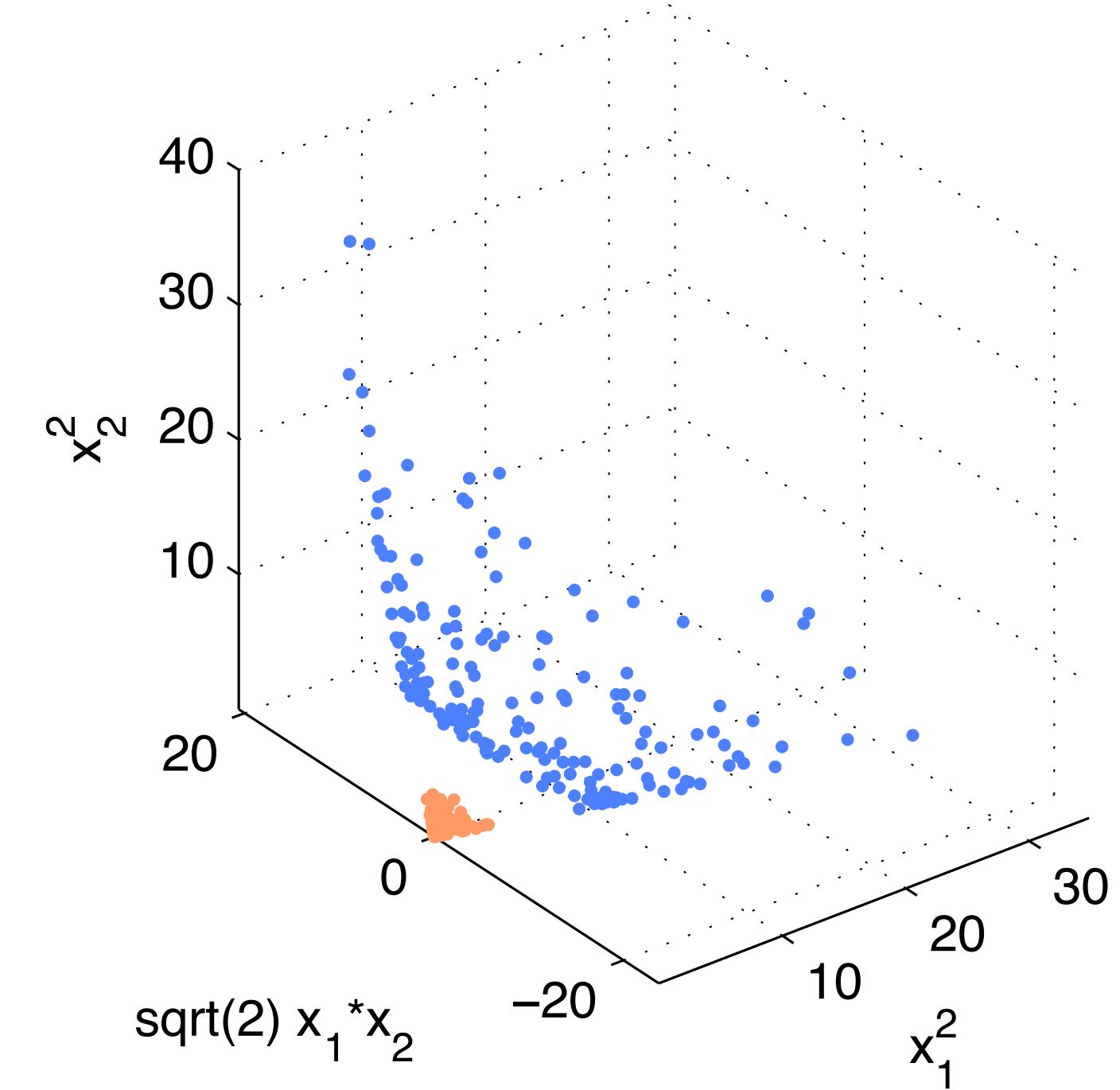
$$g(\mathbf{z}_j) = \mathbf{w}^\top \cdot \mathbf{z}_j + w_0 = \sum a_i y_i (\mathbf{z}_i^\top \cdot \mathbf{z}_j) + w_0$$

- We can easily show that, e.g., for:

$$\mathbf{z} = \begin{bmatrix} x_1^2, & \sqrt{2}x_1 x_2, & x_2^2 \end{bmatrix}^\top$$

$$\mathbf{z}_i^\top \cdot \mathbf{z}_j = (\mathbf{x}_i^\top \cdot \mathbf{x}_j)^2$$

- i.e., we don't have to work in high dims!



# Similar such maps

- Polynomial:

$$K(\mathbf{x}, \mathbf{v}) = (\mathbf{x}^\top \cdot \mathbf{v} + 1)^q, \quad q > 0$$

- Radial basis functions:

$$K(\mathbf{x}, \mathbf{v}) = e^{-\frac{\|\mathbf{x} - \mathbf{v}\|^2}{\sigma^2}}$$

- Hyperbolic tangent:

$$K(\mathbf{x}, \mathbf{v}) = \tanh(\beta \mathbf{x}^\top \cdot \mathbf{v} + \gamma)$$

# Kernel SVMs

- Use the dual form to optimize:

$$\max_a \left[ \sum a_i - \frac{1}{2} \sum a_i a_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \right]$$

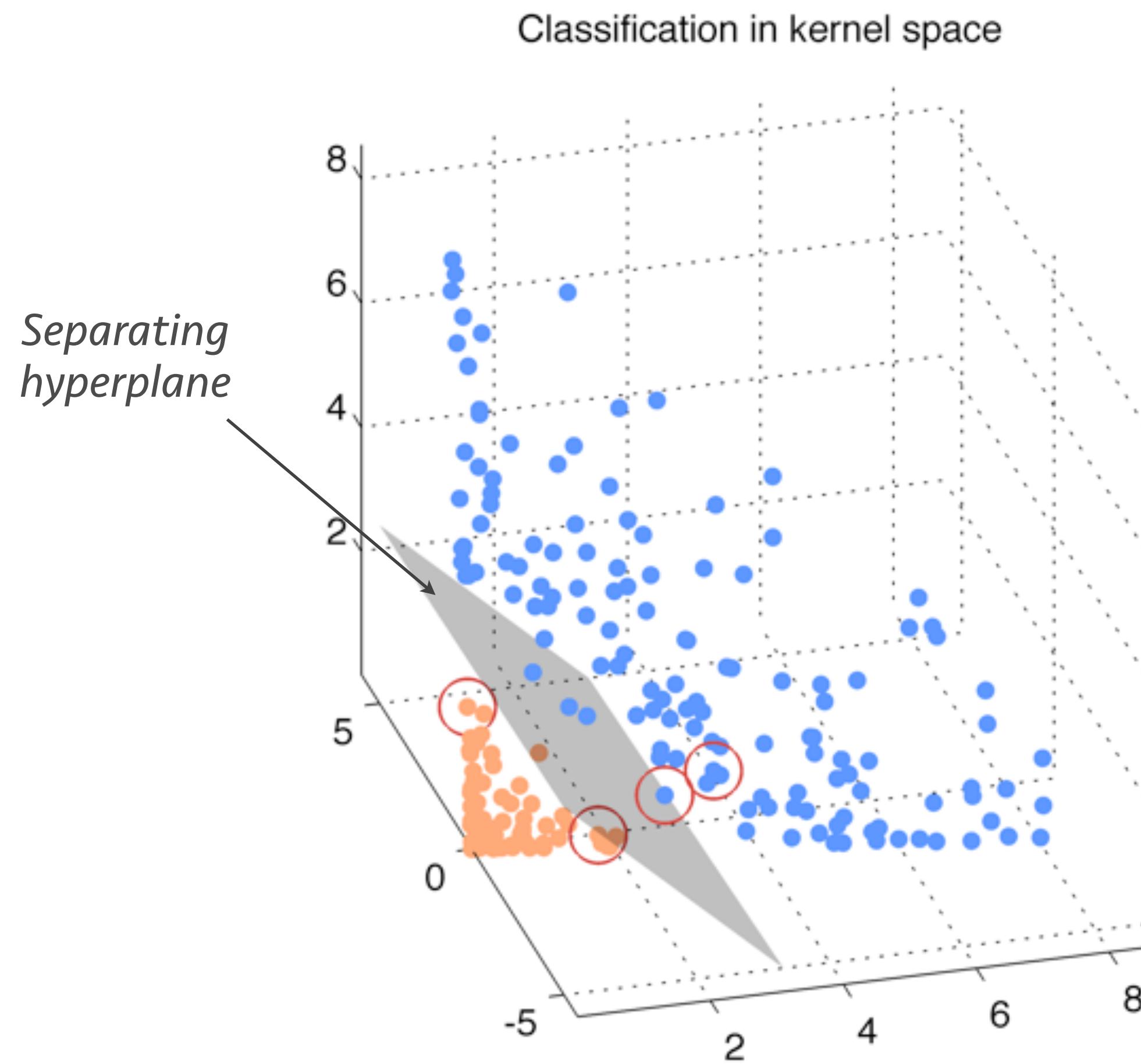
*subject to*  $\sum a_i y_i = 0$

$$0 \leq a_i \leq C$$

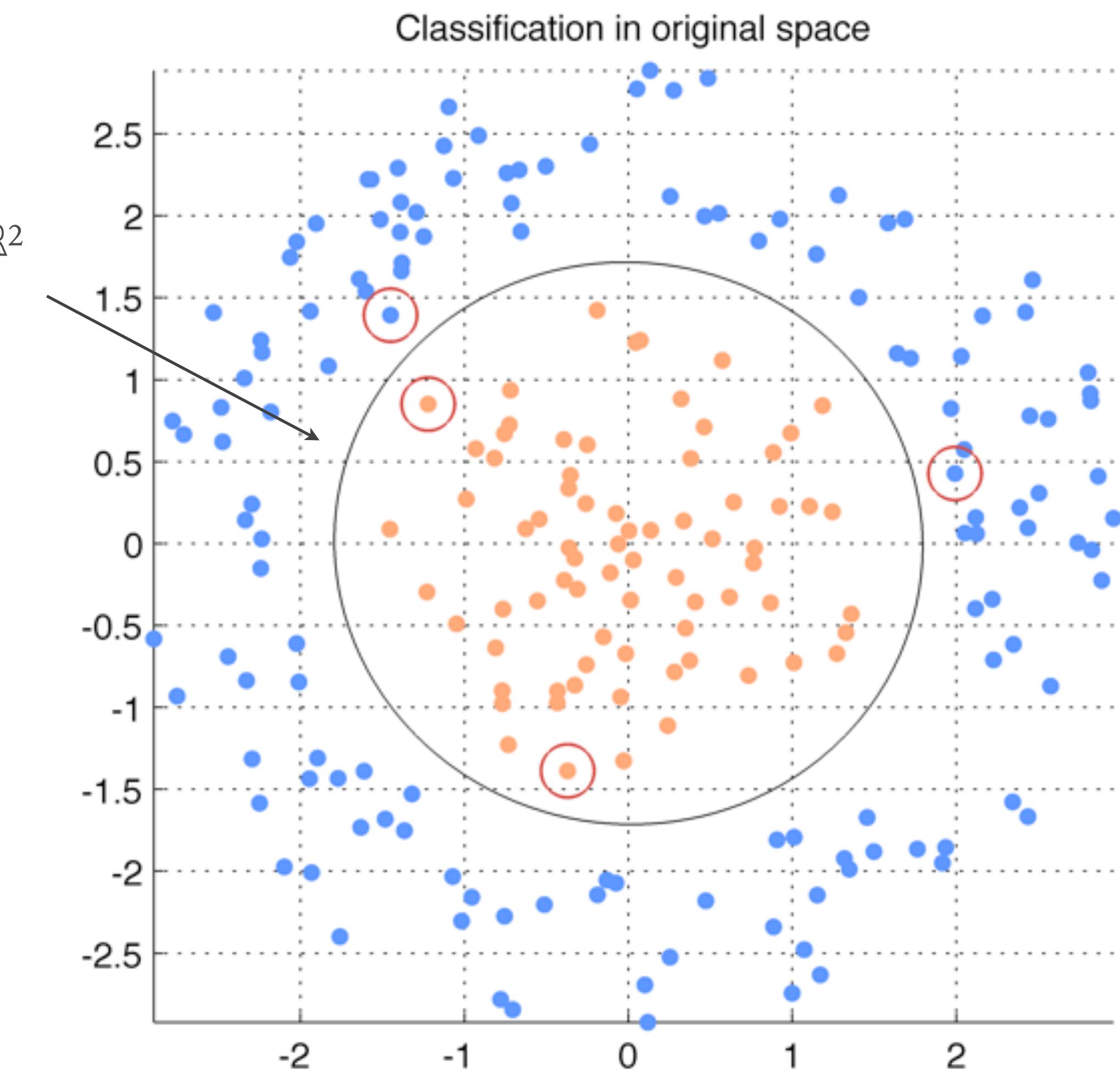
- And classify with:

$$g(\mathbf{x}) = \sum a_i y_i K(\mathbf{x}_i, \mathbf{x}) + w_0$$

# Example case



*Separating  
hyperplane  
projected in  $\mathbb{R}^2$*



# Lots of ways to SVM

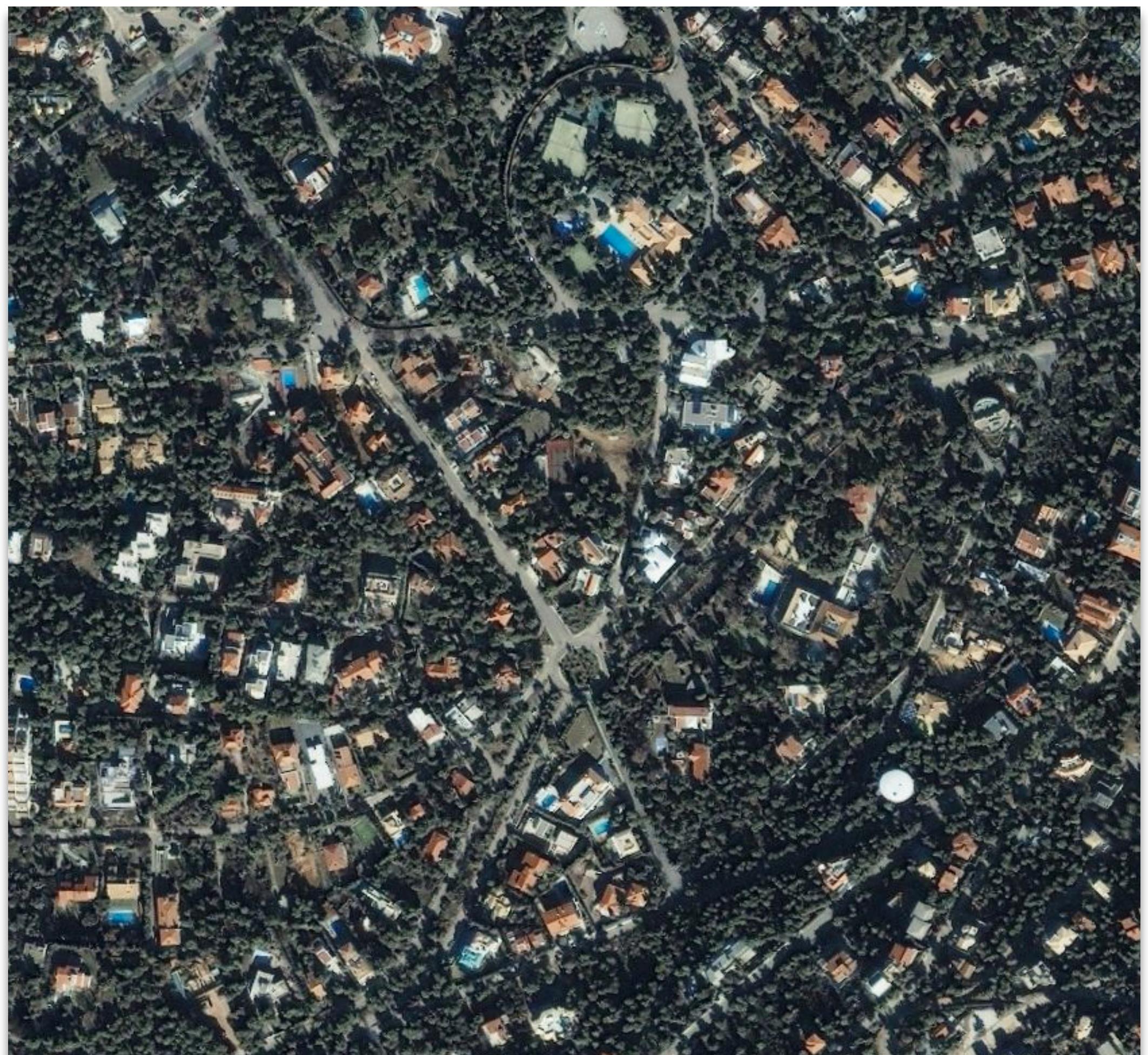
- Quadratic programs are one way
  - The SMO algorithm is faster
  - Can also use Interior Point solvers
  - Or subsample the gradient (PEGASOS)
- Specialized solvers
  - SVM-light, libSVM, ...

# Kernels everywhere!

- If you have a linear product you can kernelize it
  - E.g. The perceptron, PCA, etc
- Kernels are a great way to non-linearize approaches that are inherently linear
  - You can even do it for DSP algorithms

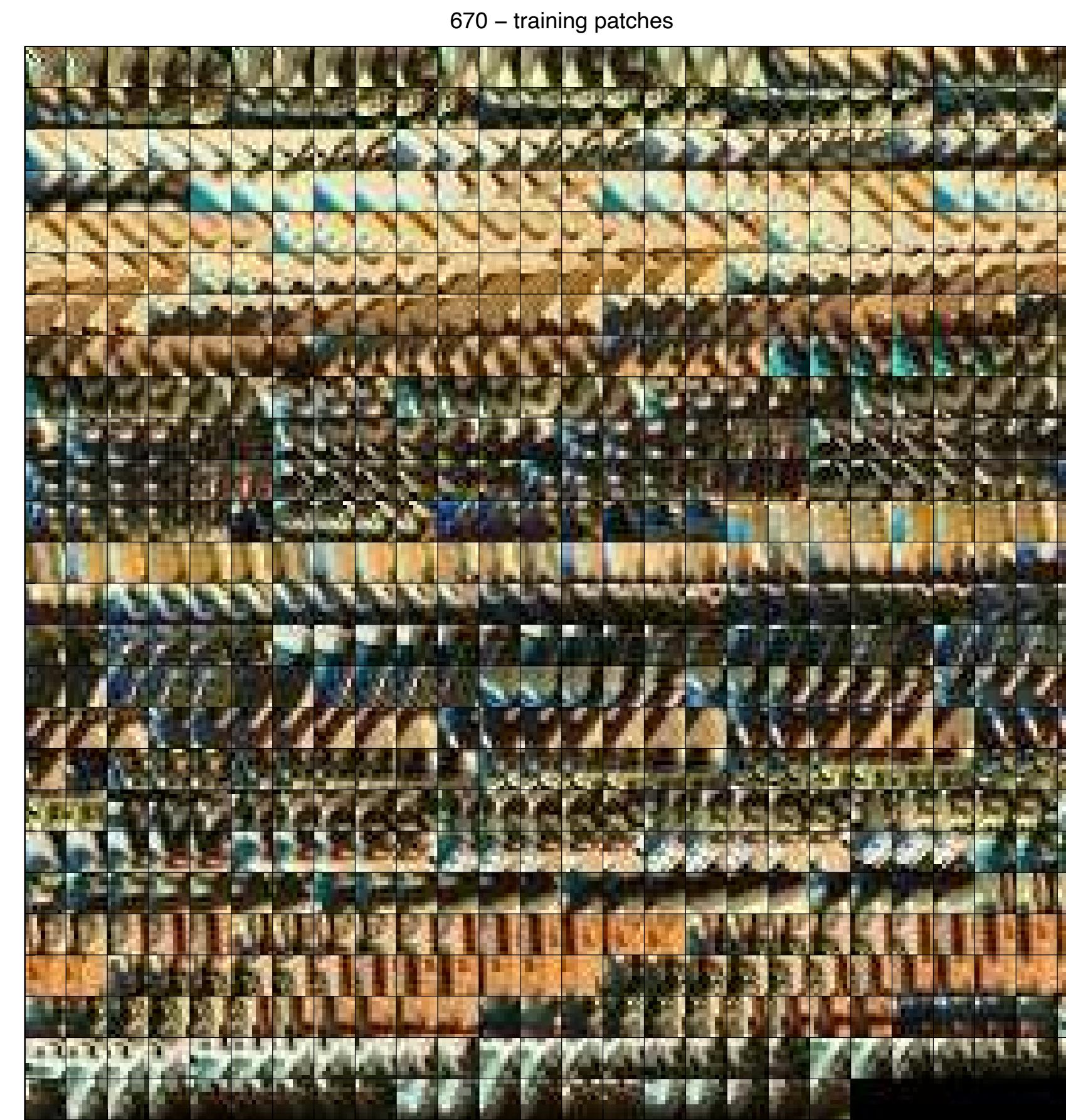
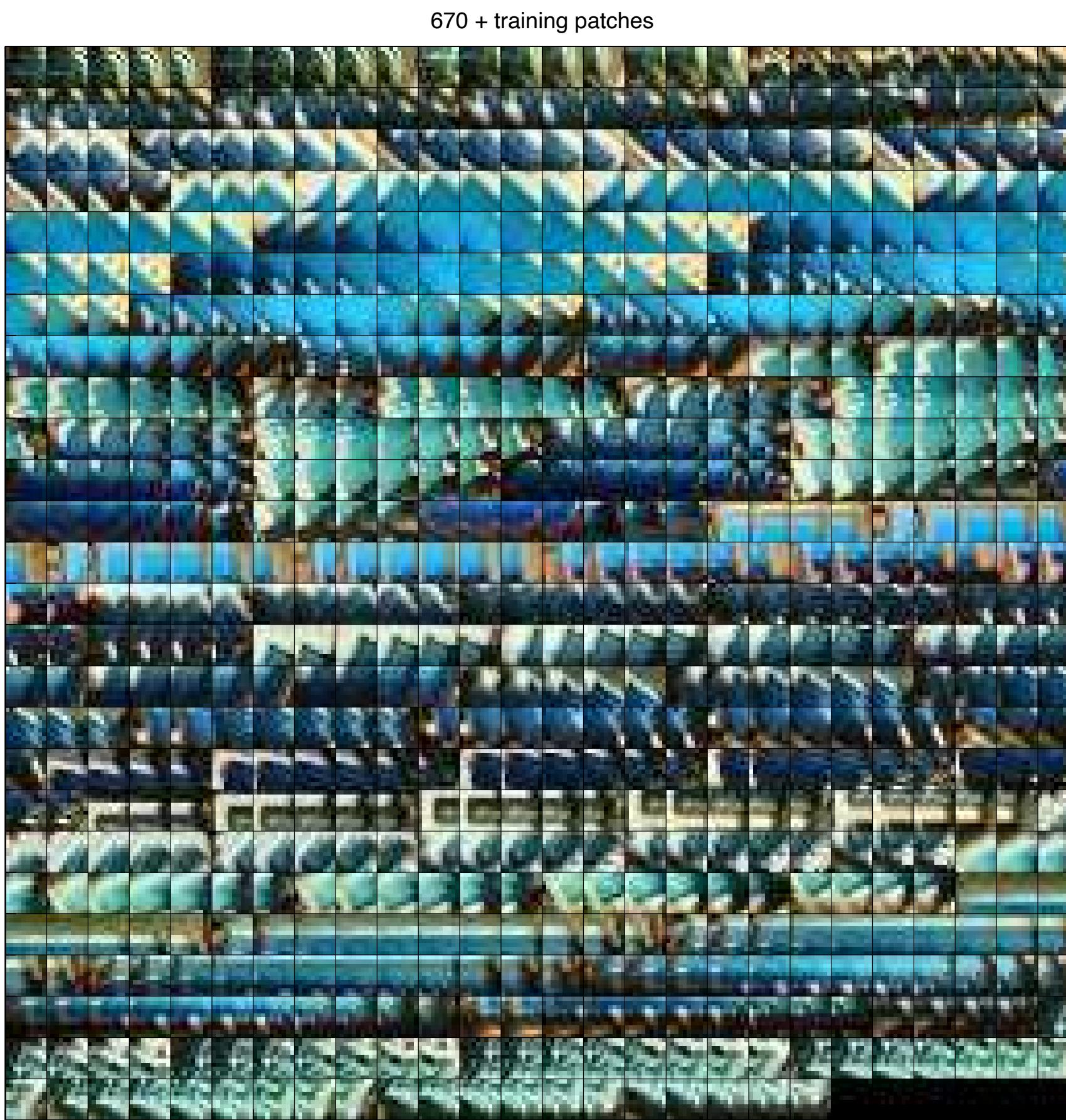
# Non-linear classifiers at work

- Object detection from satellite images
- Break image in tiny patches and classify them
  - We did the same with correlation and matched filters
- This time we get a binary answer instead  $\{-1, +1\}$

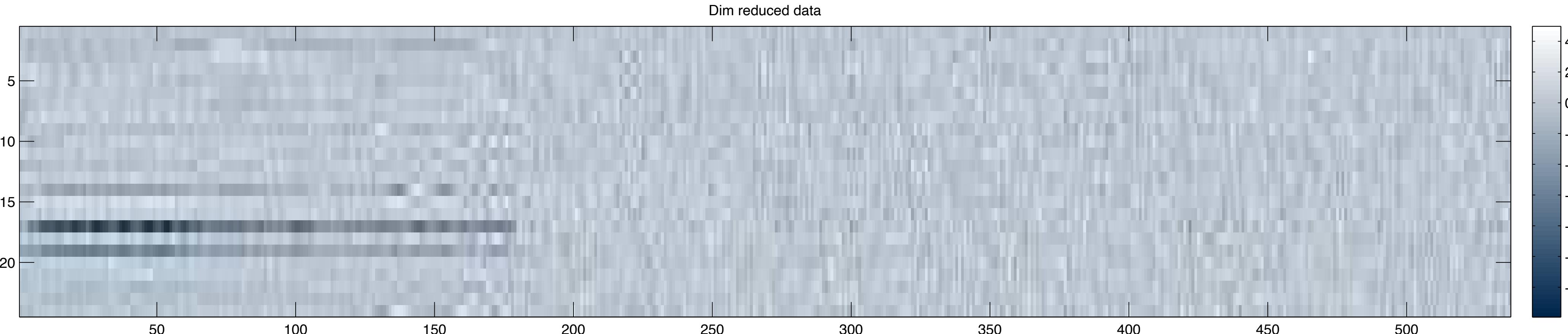
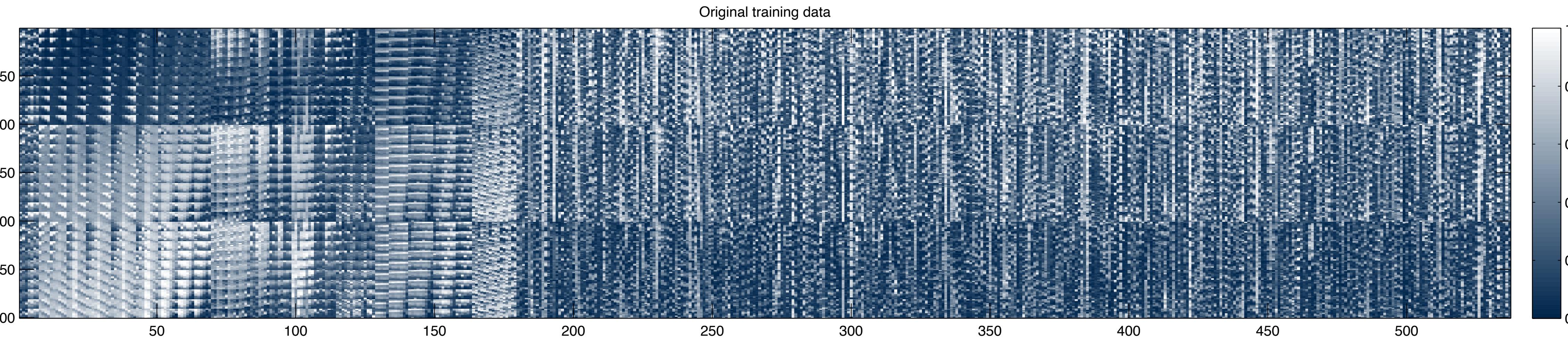


# Step 1: Make training data

- Collect examples of positives and falses



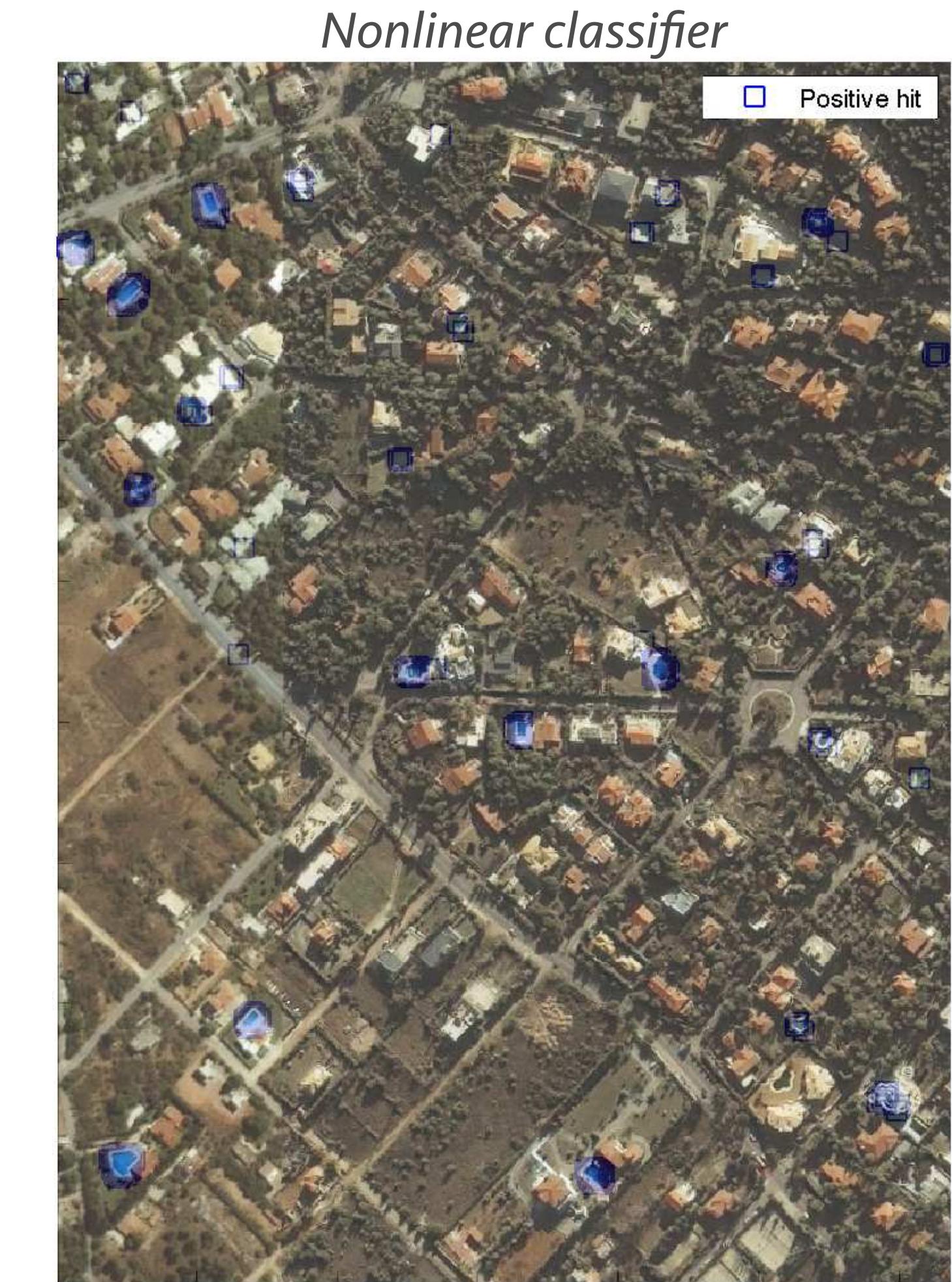
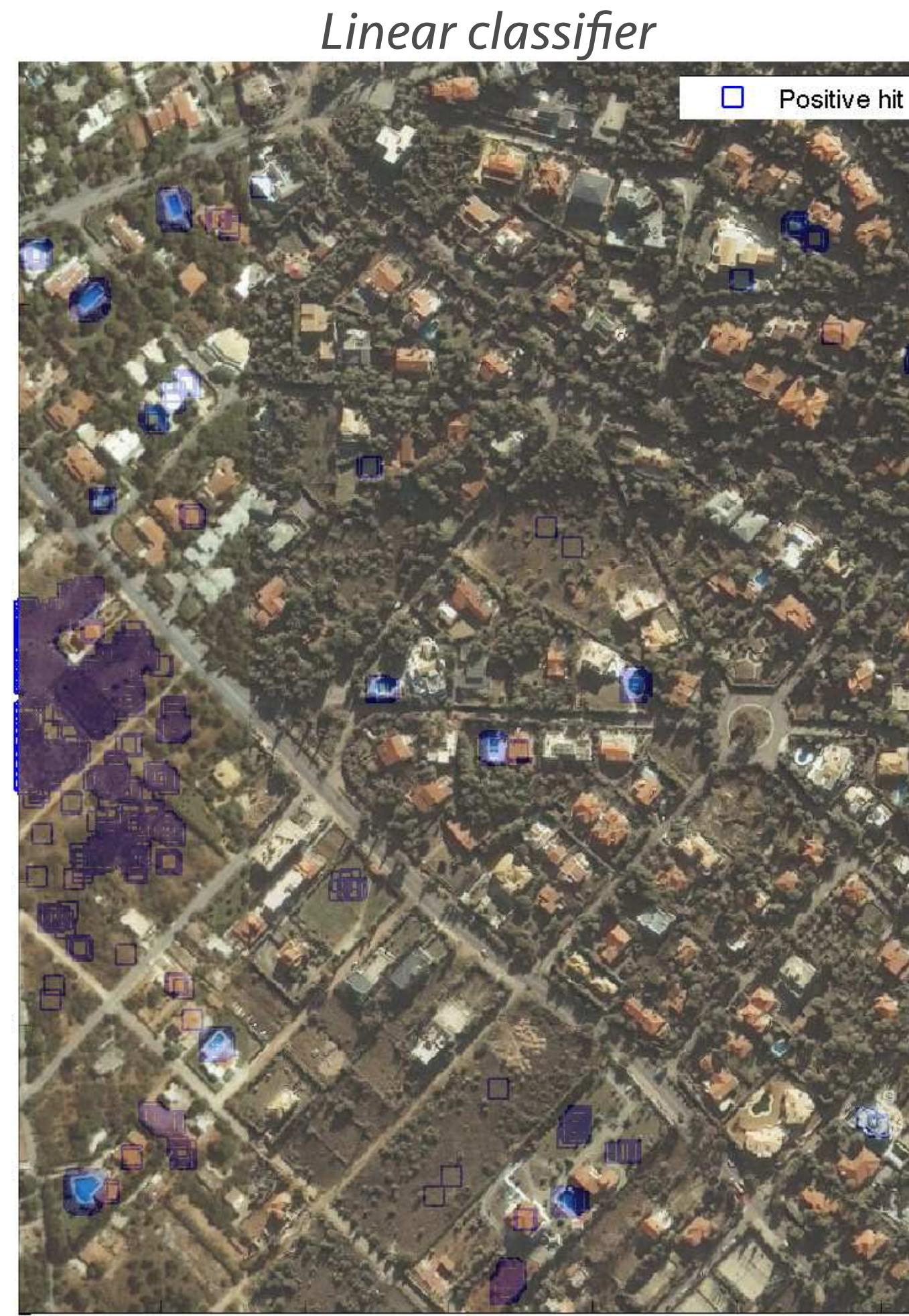
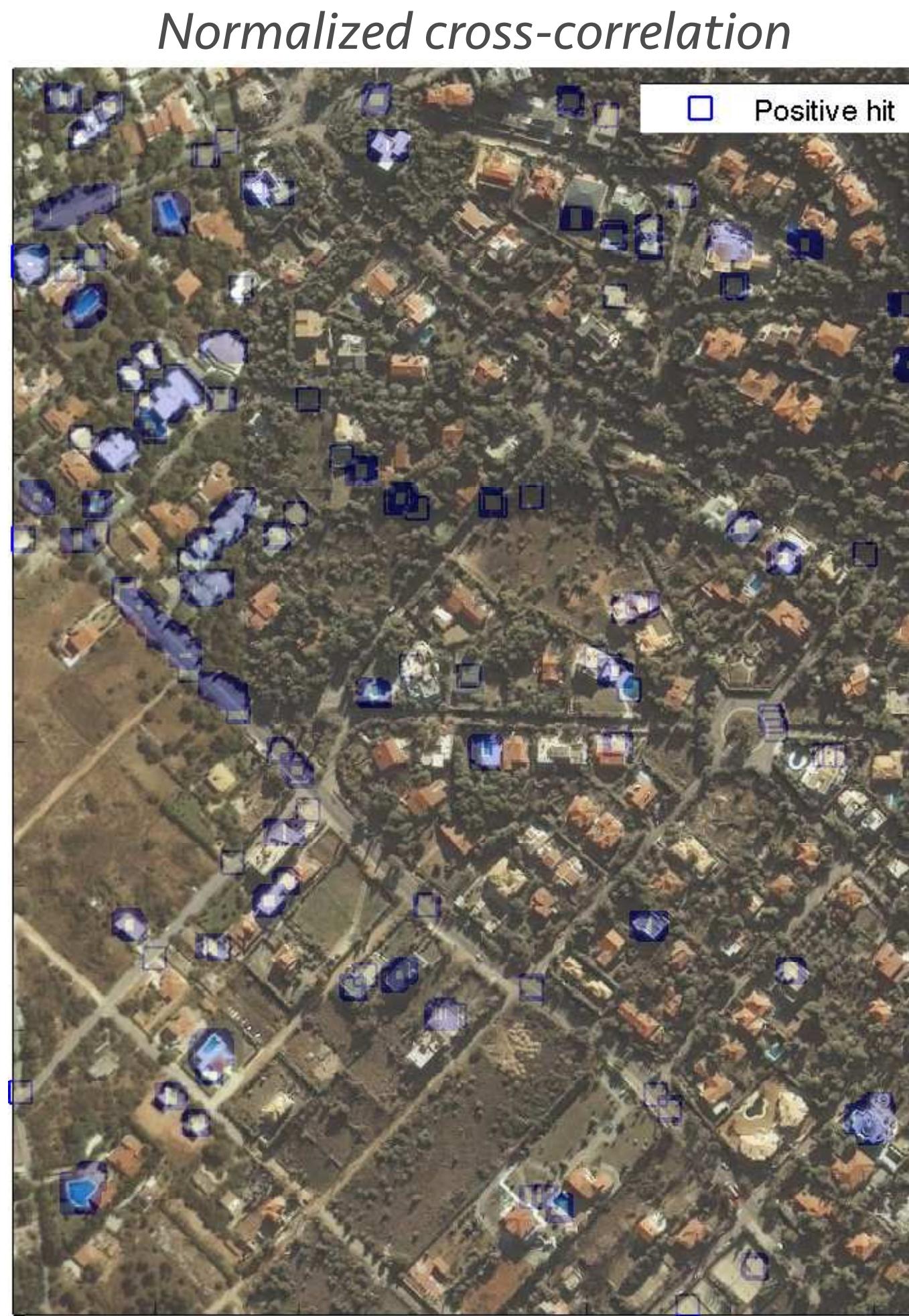
# Step 2: Drop the dimensions (optional)



# Step 3: Learn various classifiers

- As you should know by now, there is no “best” option
  - Start with simplest classifier and move to more complex if it fails
- Number of feature dimensions is also crucial
  - You need to play around with that
- Classifier parameters are also important
  - Which kernel? How many hidden layers? Diagonal covariance?

# Step 4: Evaluate on unseen data



# Recap

- Neural Nets
  - Perceptron to multilayered neural nets
- Support Vector Machines
  - Maximum margins to kernel classifiers

# Next lecture

- More classification odds and ends
- Combining classifiers
- Real-life use and experiment setups
- Tradeoffs in classification

# Reading

- Textbook: Finish up chapter 3 and 4.1-4.20
  - We haven't covered all of that work, treat the uncovered material as extra reading