CS598PS – Machine Learning for Signal Processing

From linear to non-linear classifiers

29 September 2017

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 kernelized

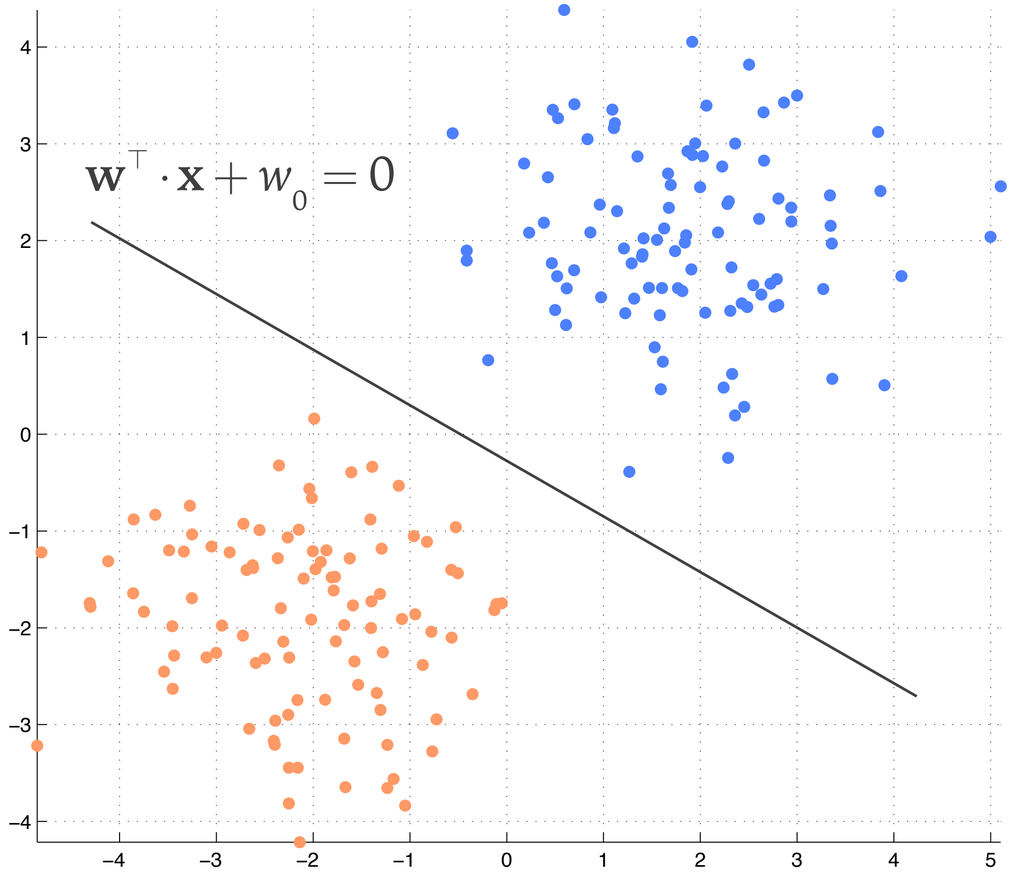
The Perceptron again

- Defines the boundary that results in the least misclassifications

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

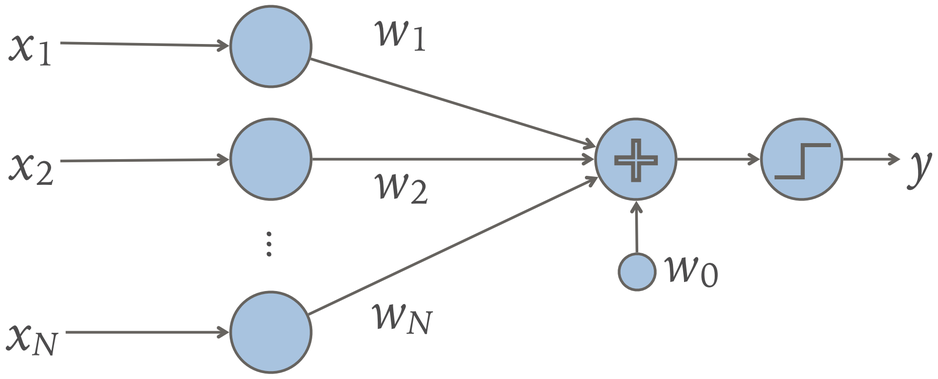
- Learn iteratively:

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



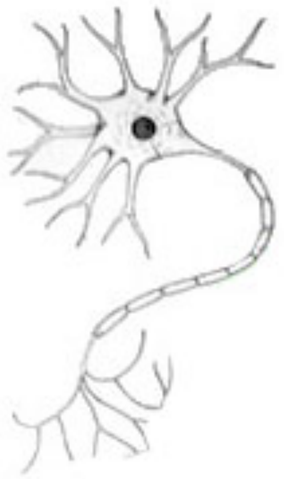
The biological interpretation

The perceptron

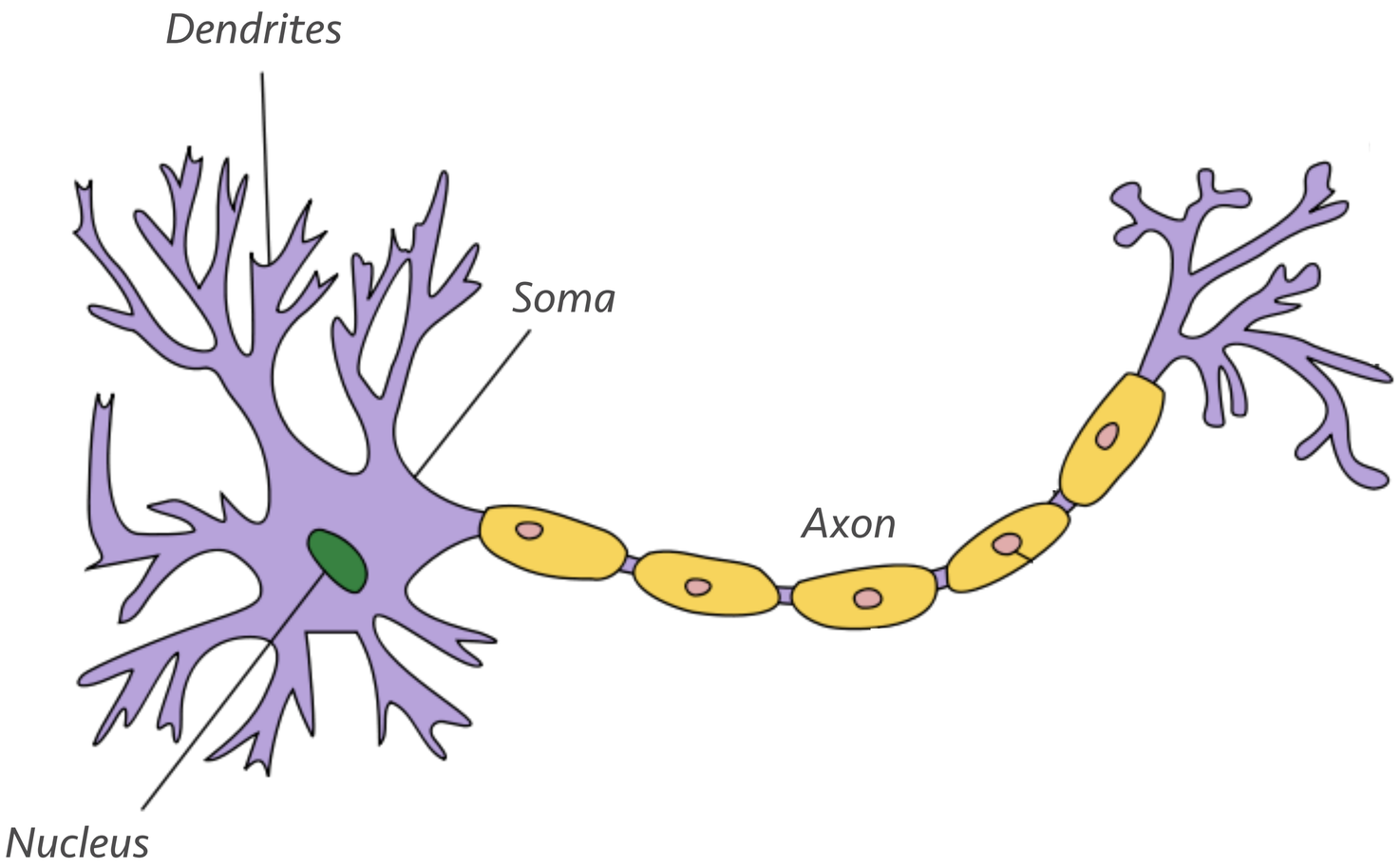


$\approx$

The neuron



The Neuron

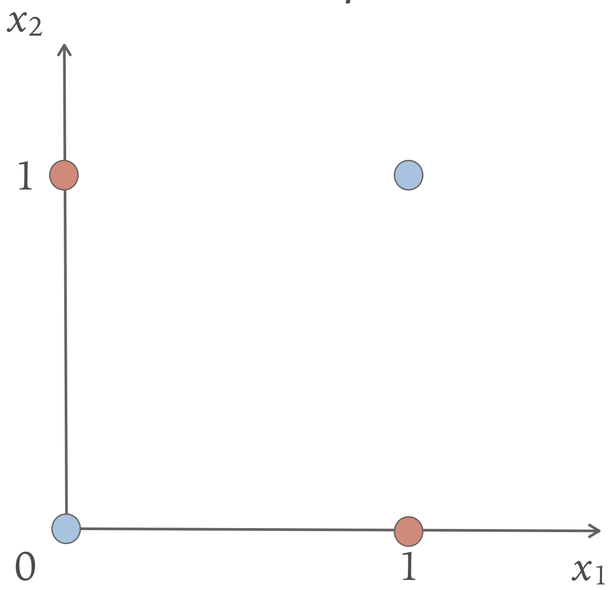


The problem case

- The XOR case
  - Linearly non-separable case

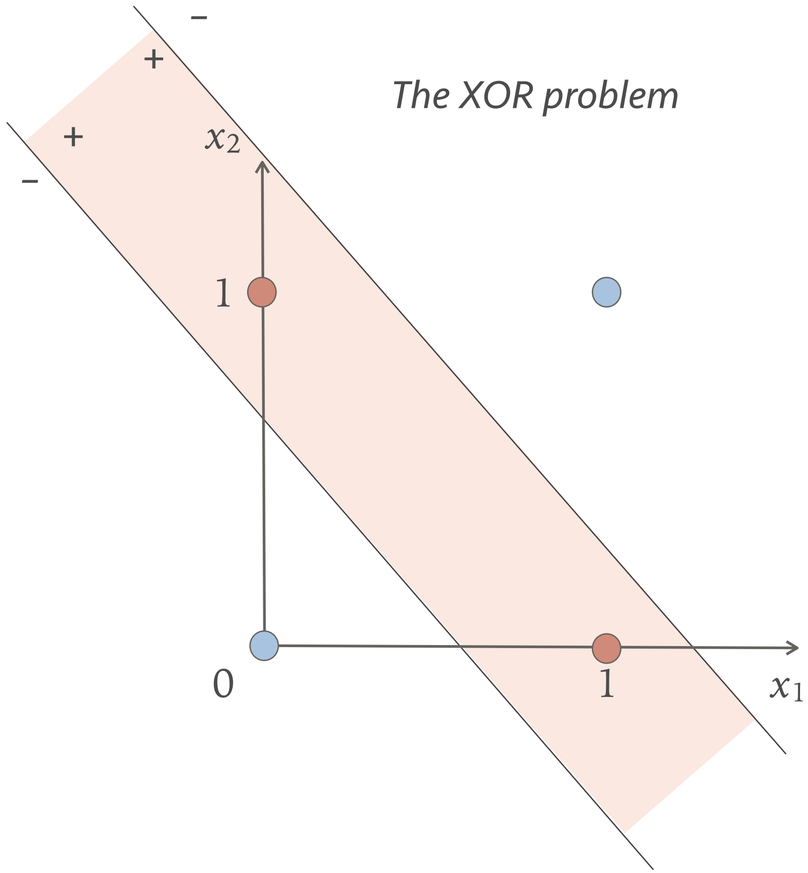
x_1	x_2	XOR	Class
0	0	0	−1
0	1	1	+1
1	0	1	+1
1	1	0	−1

The XOR problem

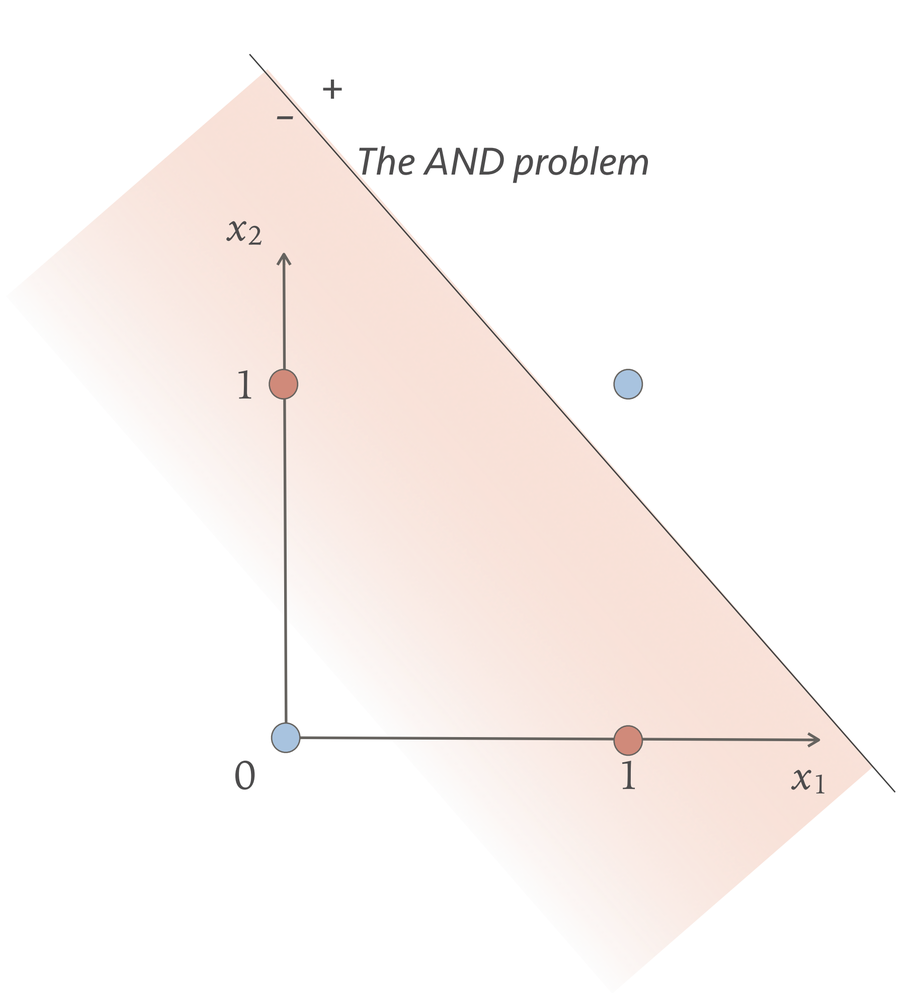


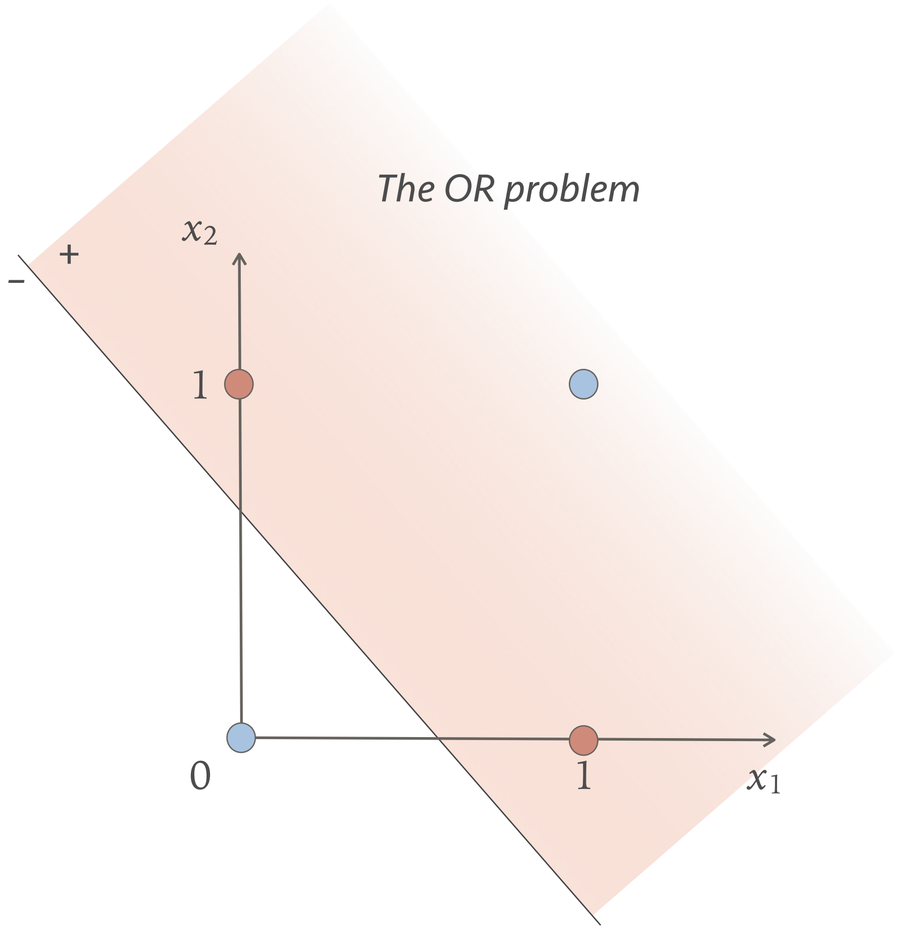
An ideal solution

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



An AND and an OR

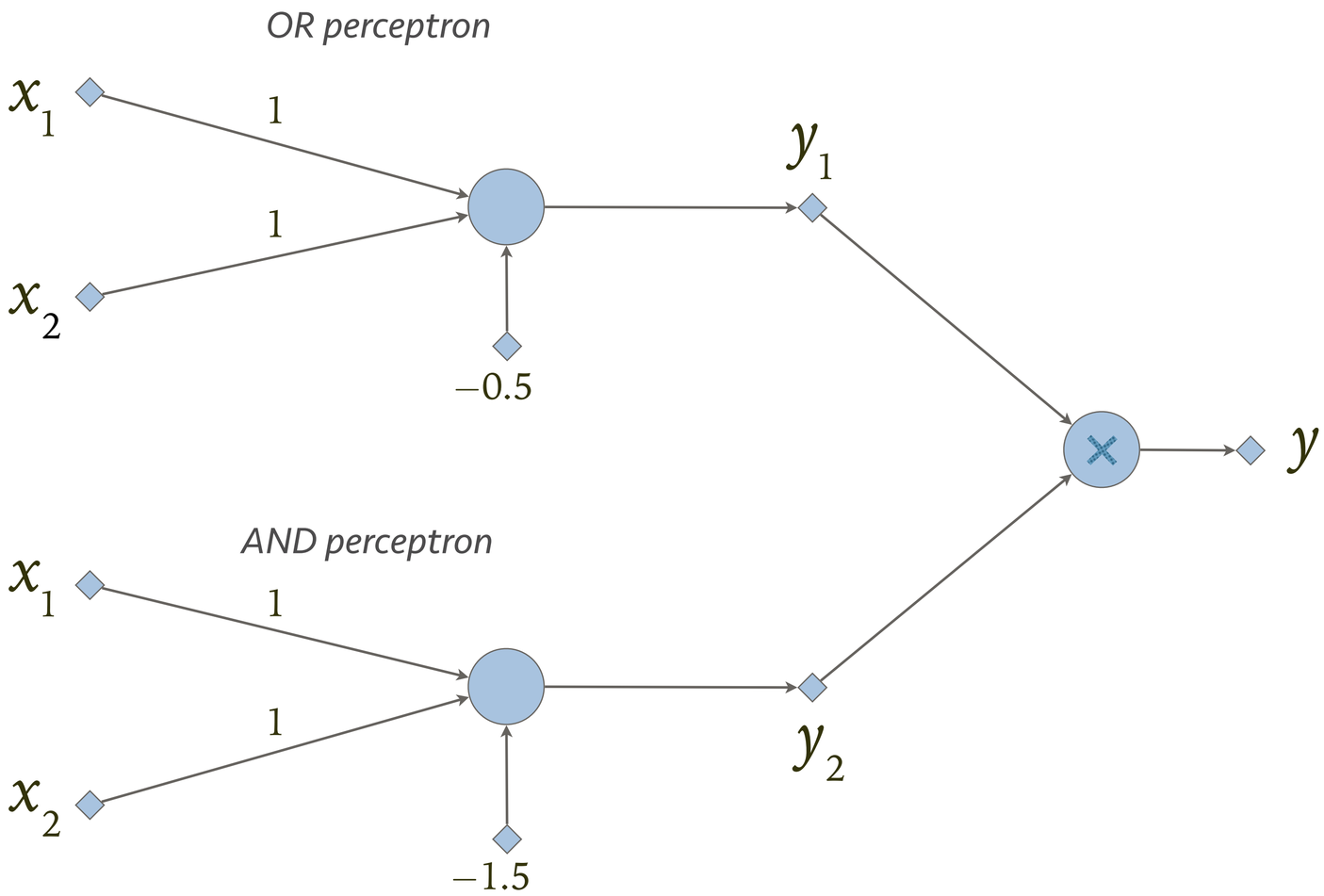




Two parallel perceptrons

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

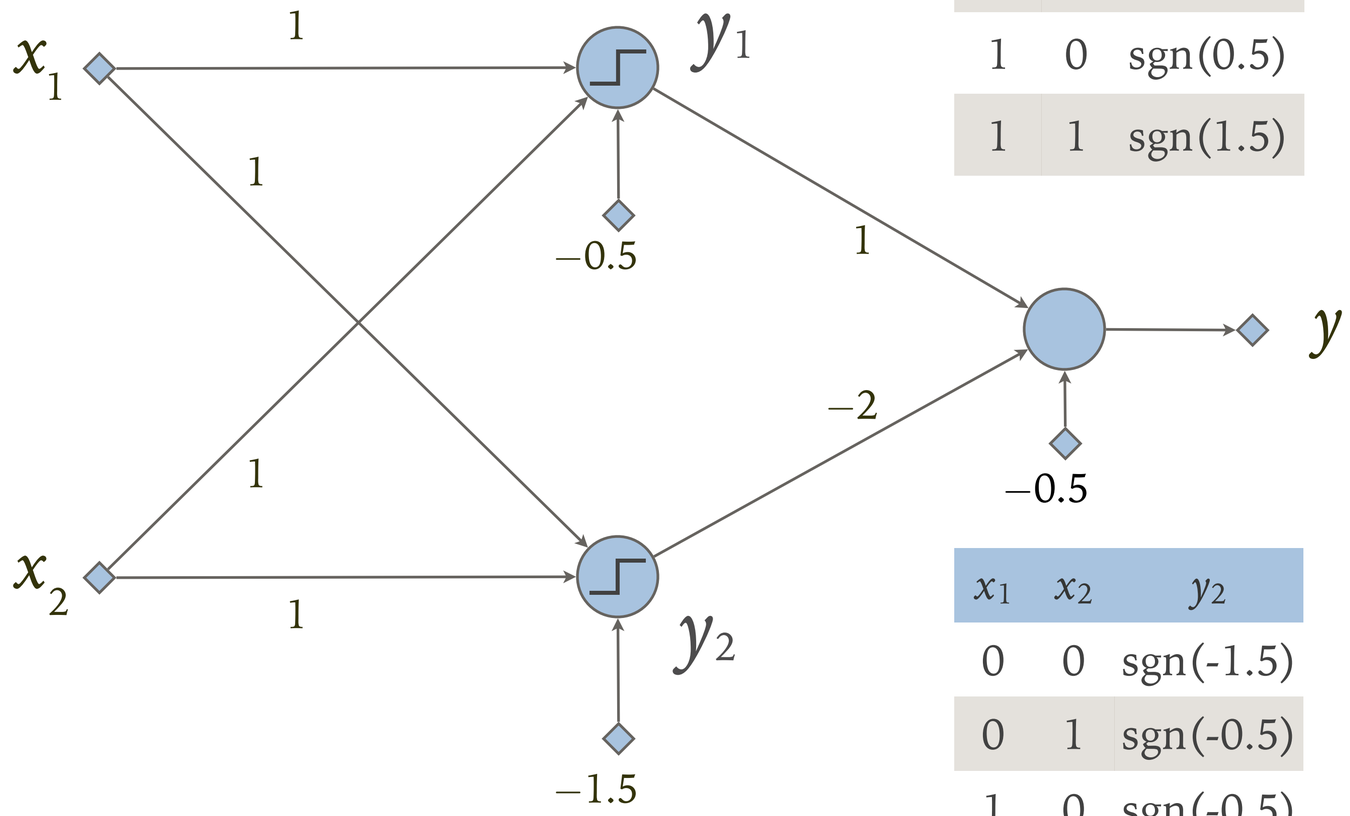
x_1	x_2	y_2
0	0	-1.5
0	1	-0.5
1	0	-0.5
1	1	0.5



x_1	x_2	y
0	0	0.75
0	1	-0.25
1	0	-0.25
1	1	0.75

Or more compactly

- As one 2-layer perceptron



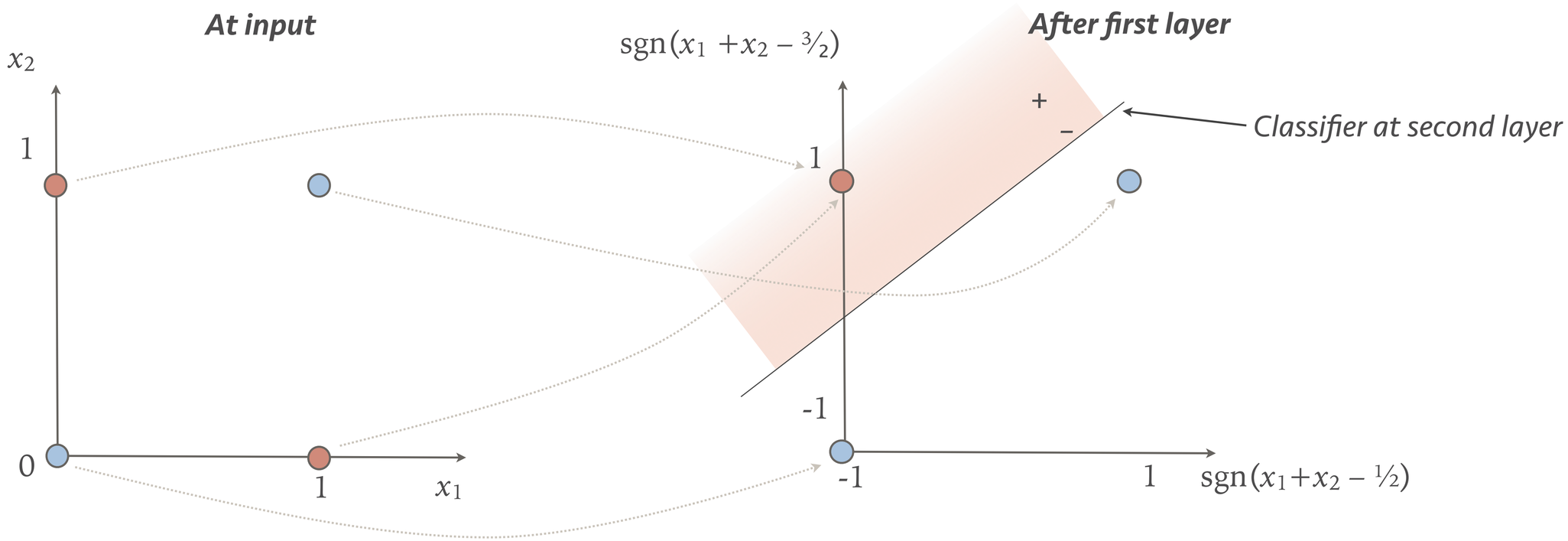
x_1	x_2	y_1
0	0	sgn(-0.5)
0	1	sgn(0.5)
1	0	sgn(0.5)
1	1	sgn(1.5)

x_1	x_2	y_2
0	0	sgn(-1.5)
0	1	sgn(-0.5)
1	0	sgn(-0.5)
1	1	sgn(0.5)

x_1	x_2	y	sgn(y)
0	0	-0.5	-1
0	1	1.5	1
1	0	1.5	1
1	1	-2.5	-1

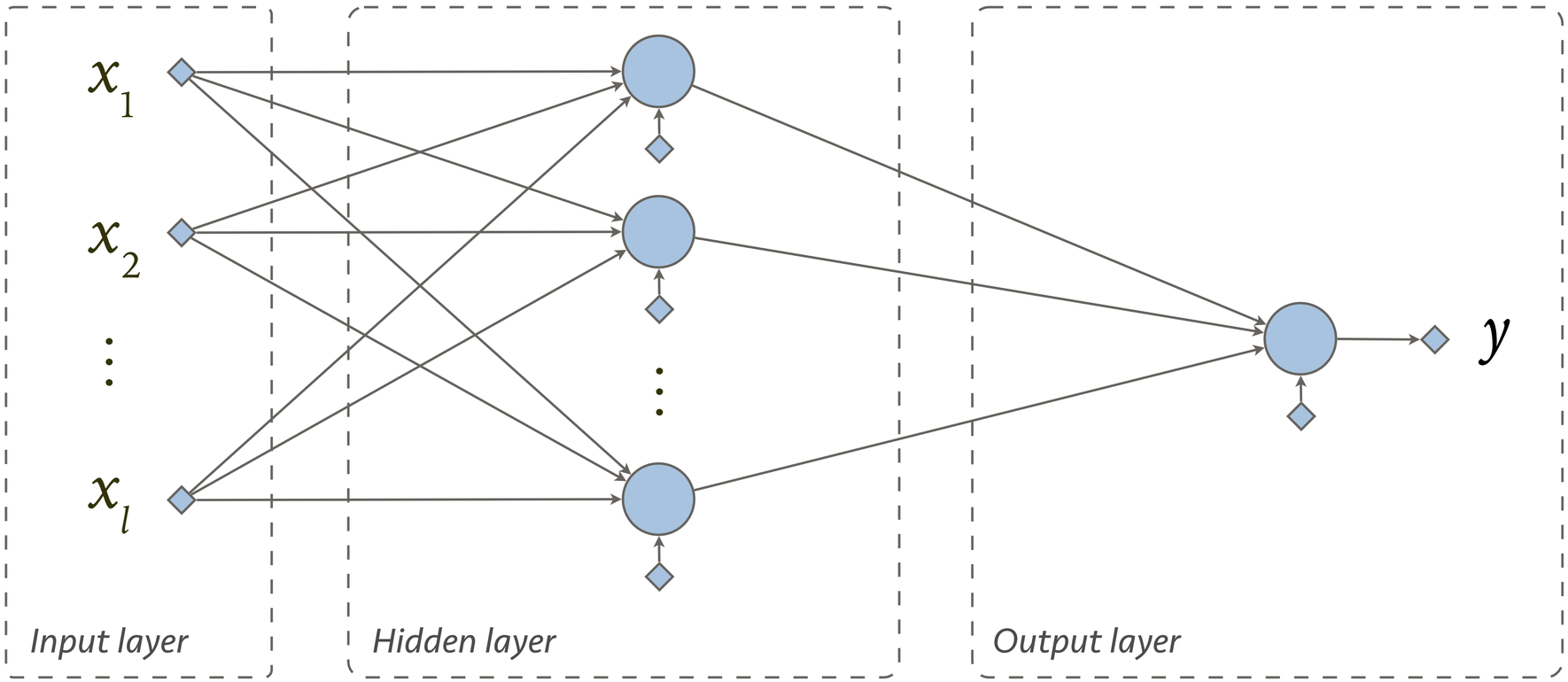
A different interpretation

- First layer transforms the data to a space where the solution is linearly-separable
  - Second layer finishes the job!



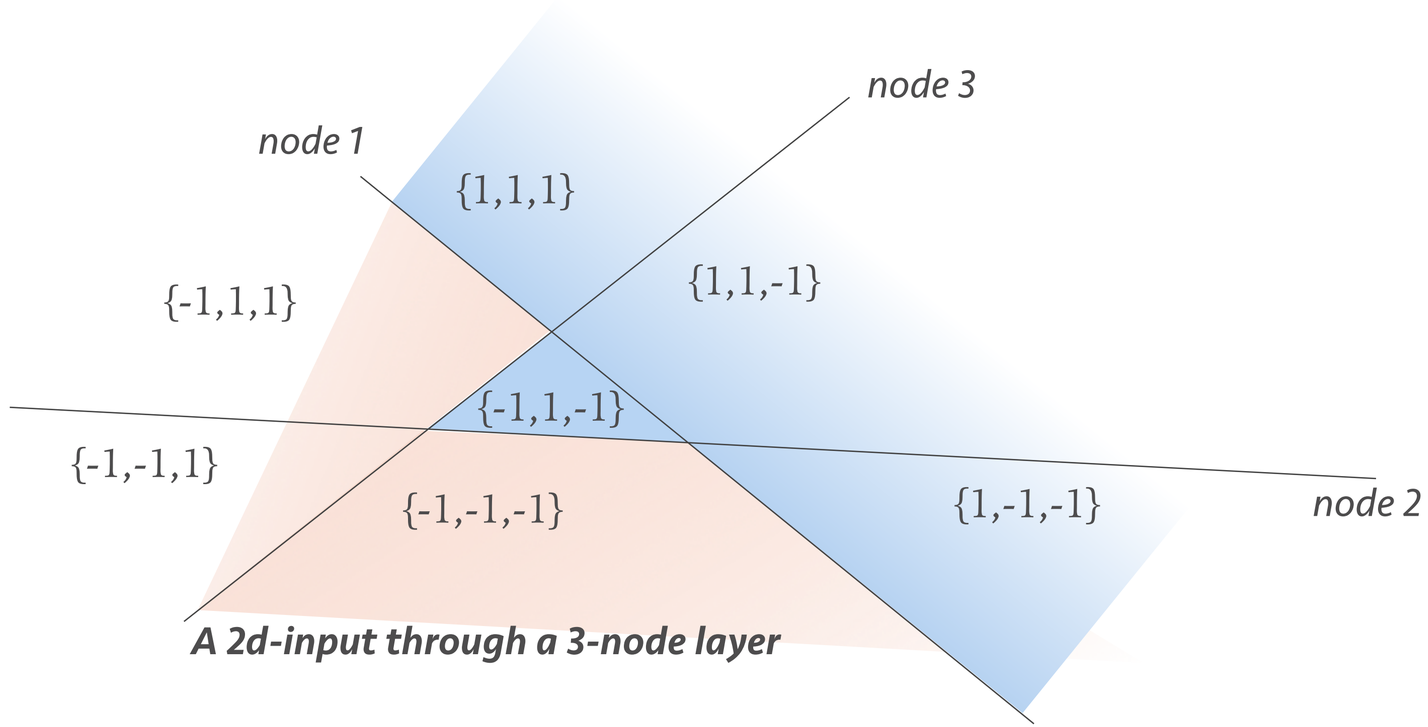
Terminology

- Two-layer feed-forward neural net



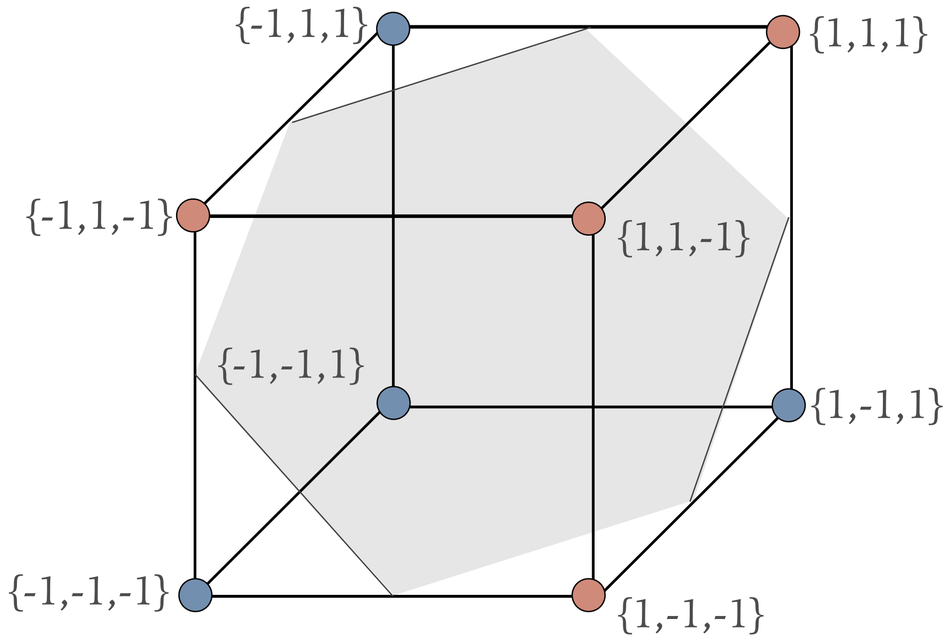
The hidden layer

- Maps all inputs onto a hypercube's vertices
  - Each node is a decision boundary
  - Hypercube in *p*-space, for *p*-nodes



The output layer

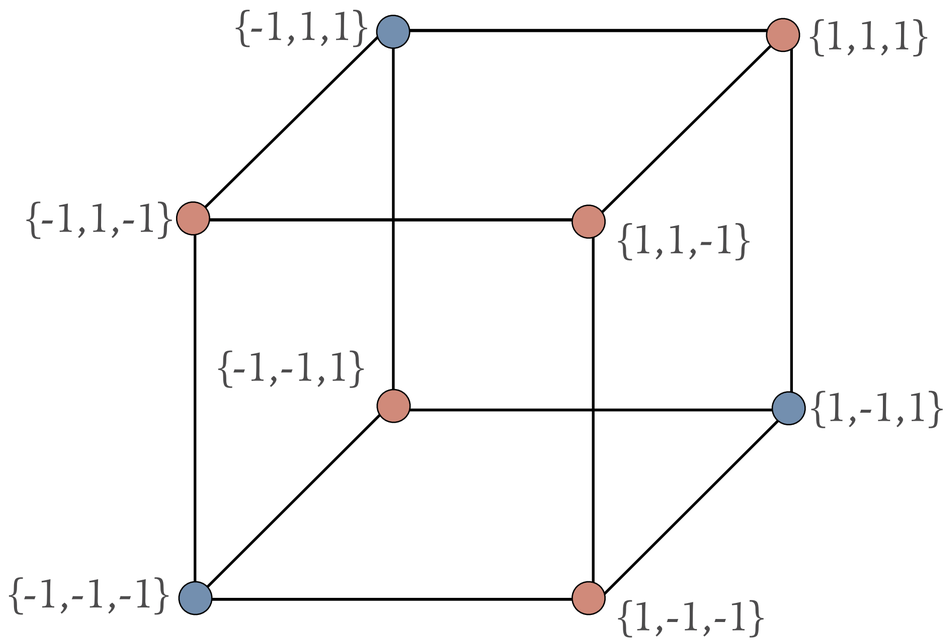
- Assuming one output
  - The single node defines a hyperplane
  - Cuts the hypercube to define two classes



From three hidden-layer nodes to one output

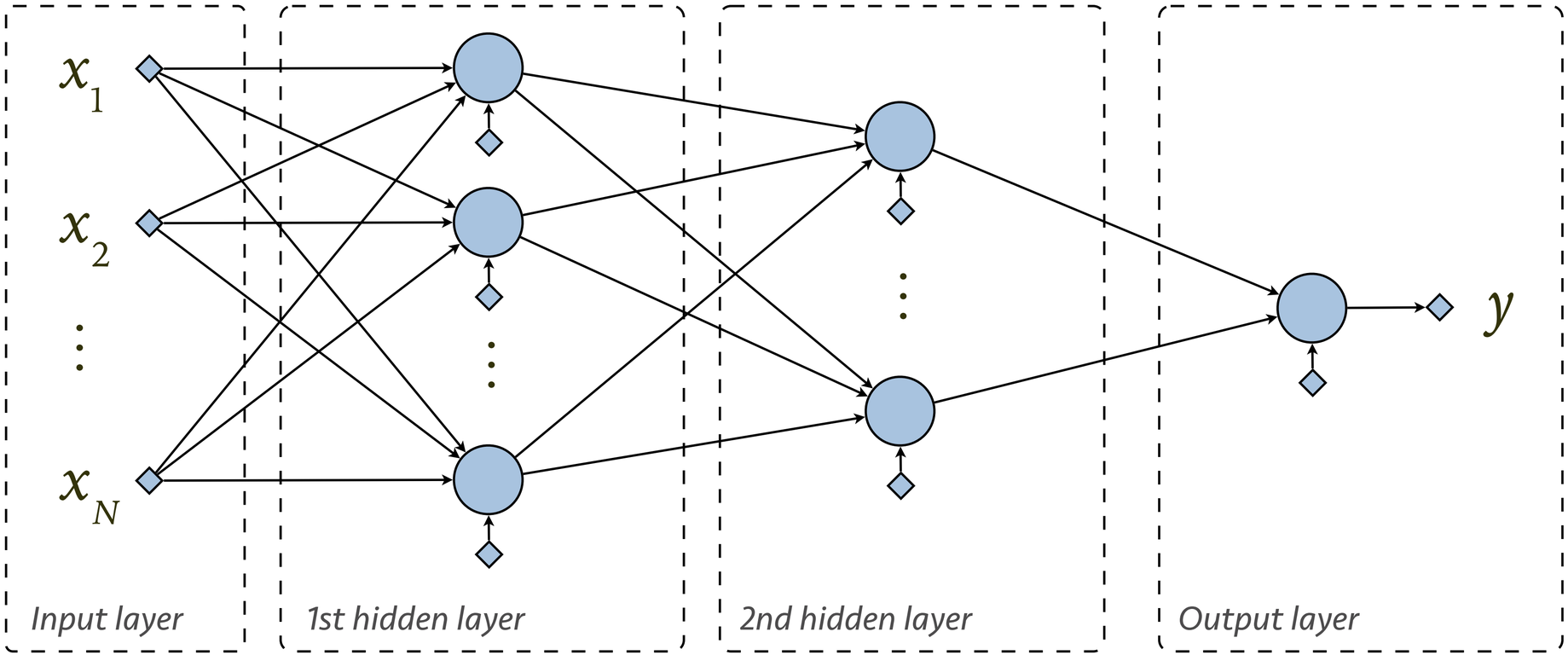
A new problem

- 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 are sufficient
  - First layer hyperplanes map 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 problems
  - But we have a new problem!

- How do we get the desired weight values?
  - This is not a simple perceptron
  - This is a highly non-linear process

Defining the process

- Each layer has:
  - 1. A weighted sum with a bias
  - 2. A function applied on the above
- Layer process is described by:

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

Neural network learning

- Define inputs/outputs

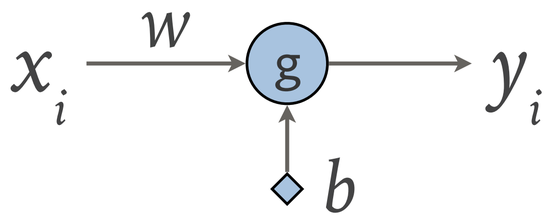
$$\mathbf{x}_i \longrightarrow \mathbf{t}_i$$

  - e.g., data samples and class labels

- Iterative training
  - Pass data through network
  - Measure error at output
  - Adjust weights to minimize error

Starting simple

- Single layer 1-input / 1-output: $y = \mathrm{g}\left(w \cdot x + b\right)$
- For a target t the error is: $\frac{1}{2}\left(y - t\right)^2$



- Start with random w and b and use gradient descent updates to revise them:

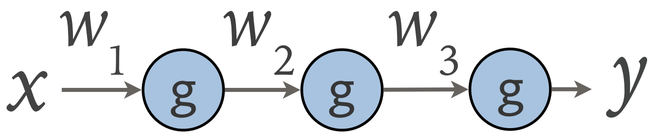
$$\Delta w \propto x\left(\mathrm{g}\left(w \cdot x + b\right) - t\right)\mathrm{g}'\left(w \cdot x + b\right)$$

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

Adding a couple of layers

- We now have: $y = \mathrm{g}\left(w_3\,\mathrm{g}\left(w_2\,\mathrm{g}\left(w_1 x\right)\right)\right)$

We skip the bias terms for sanity's sake



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

input to *error* *g' of input*

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

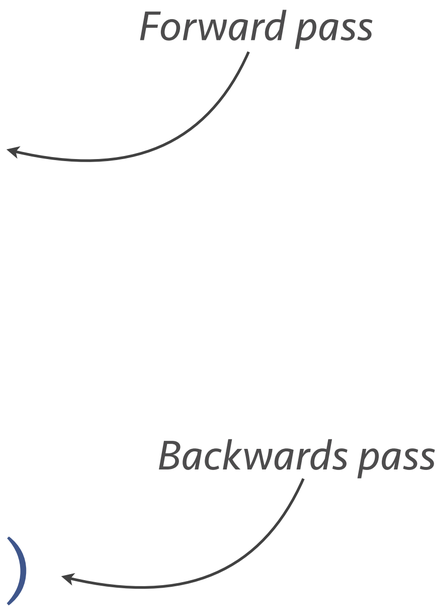
input to *error* *next weight* *g' of input* *g' of input at next layer*

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

input to *error* *next weights* *g' of input* *g' of input at next layers*

The backpropagation algorithm

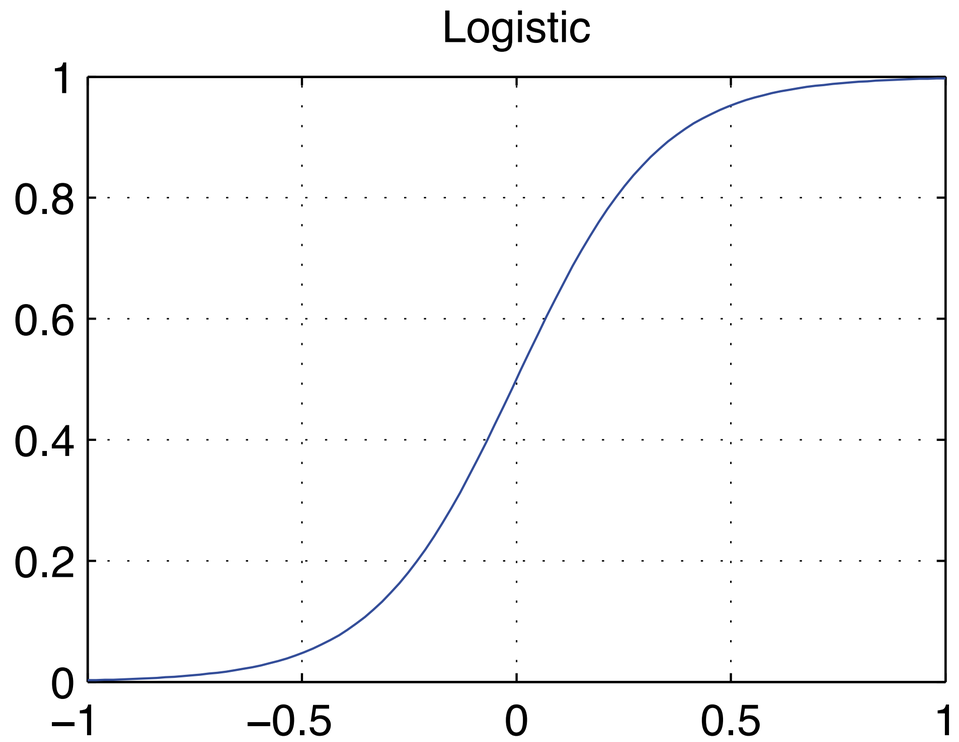
- Initialize first layer: $\mathbf{y}_0 = \mathbf{x}$
- Pass data through network: $\mathbf{y}_l = \mathrm{g}(\mathbf{W}_l \cdot \mathbf{y}_{l-1} + \mathbf{b}_l)$
- Calculate error in output: $\boldsymbol{\delta}_L = \mathbf{t} - \mathbf{y}_L$
- Backpropagate error derivative: $\boldsymbol{\delta}_l = (\mathbf{W}_{l+1}^\top \cdot \boldsymbol{\delta}_{l+1})\mathrm{g}'(net_l)$
- Adjust weights/biases: $\Delta\mathbf{W}_l = \boldsymbol{\delta}_l \cdot \mathbf{y}_{l-1}^\top \qquad \Delta\mathbf{b}_l = \boldsymbol{\delta}_l$

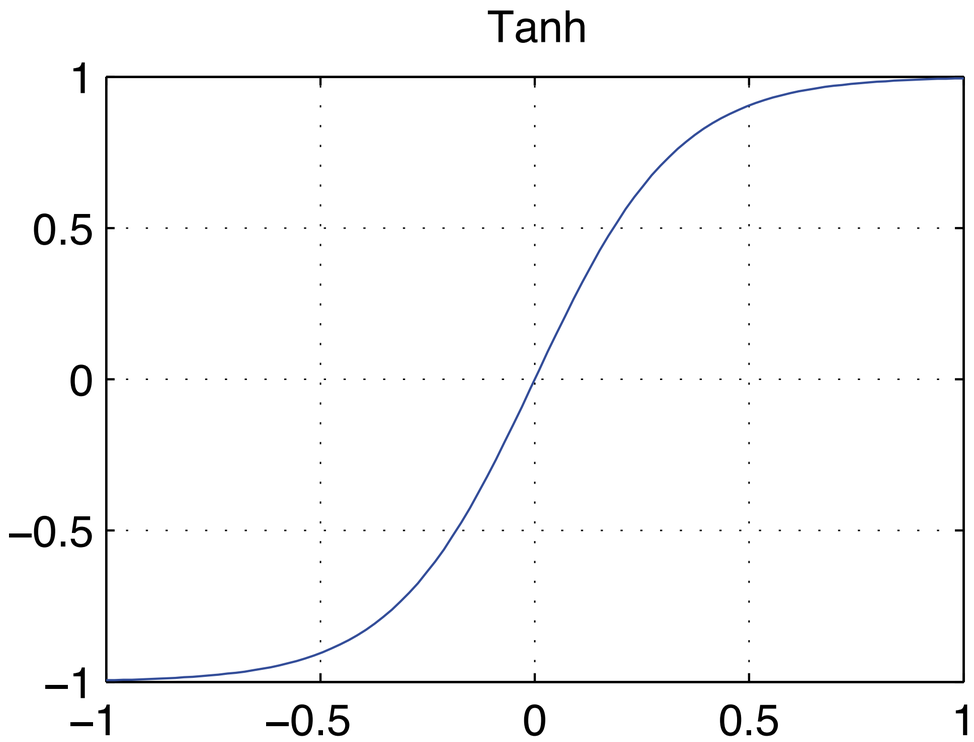


One minor issue

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

Sigmoids (beware of the output range!)



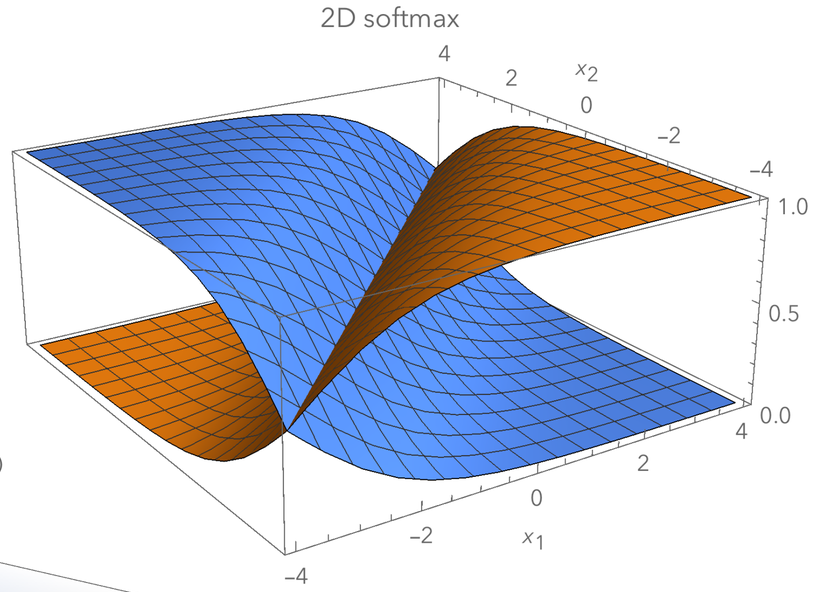


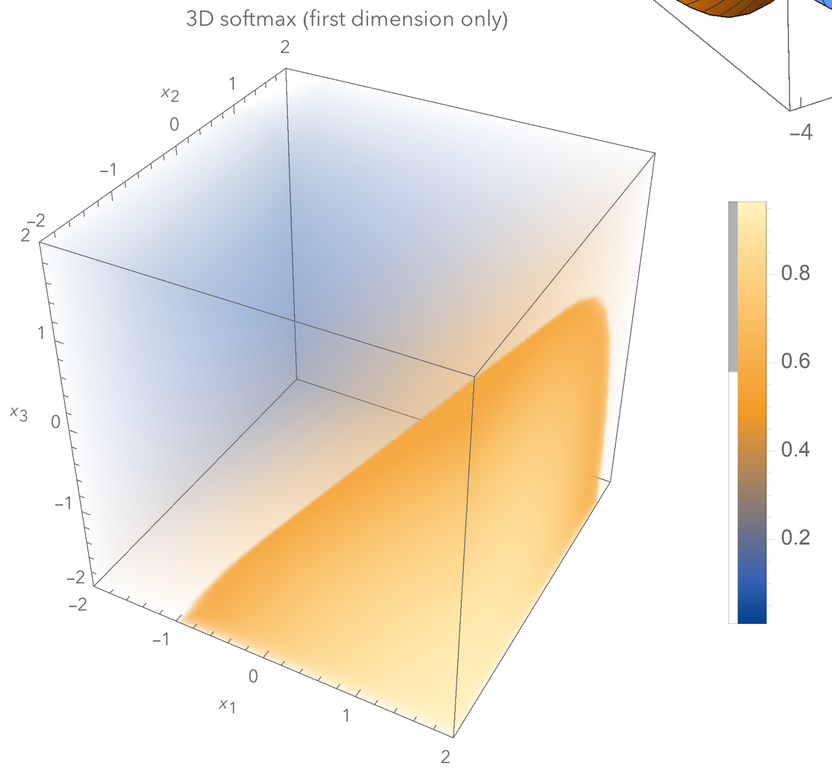
The softmax activation

- 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





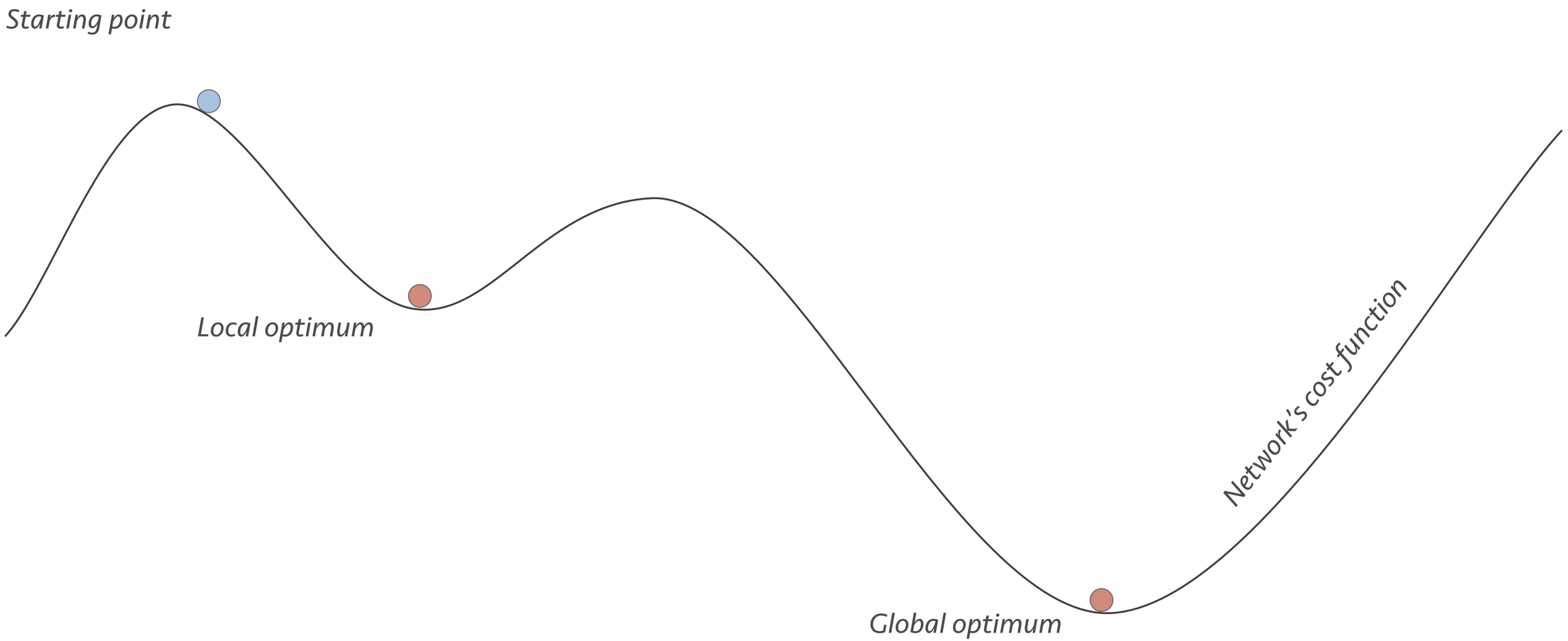
Practical matters

- 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 ...

Illustrative case



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 actually nonlinear regression
  - $\mathbf{y}_i = g(\mathbf{x}_i)$
    - $g()$ 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

- 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 next generation

- Deep learning architectures
  - Smarter local pre-training that improves accuracy
  - Better training strategies, activations, and topologies
- Relatively efficient training
  - Biological plausibility (-ish)
- Superb results in a range of applications
  - More in later lectures on deep learning

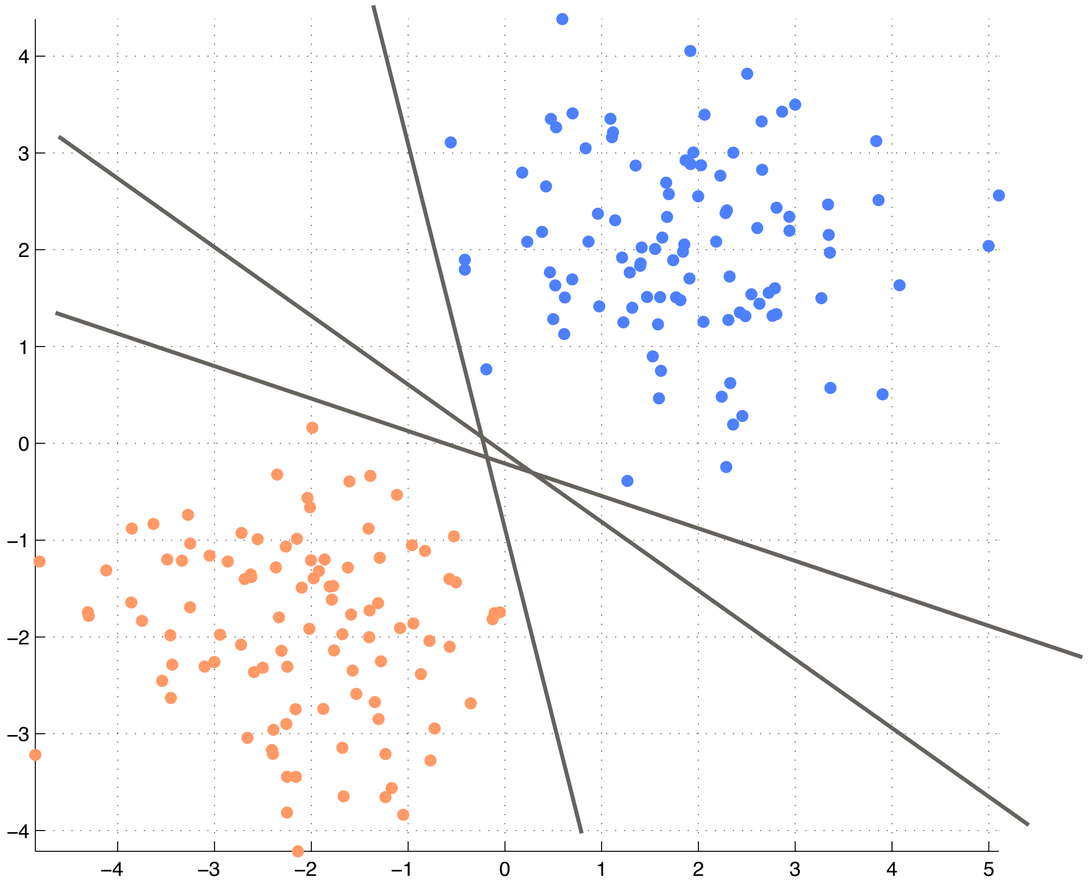
A different approach

- Support vector machines
  - Start from an algebraic perspective
  - Generalize to non-linear classification

- Formidable competition to 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 ...

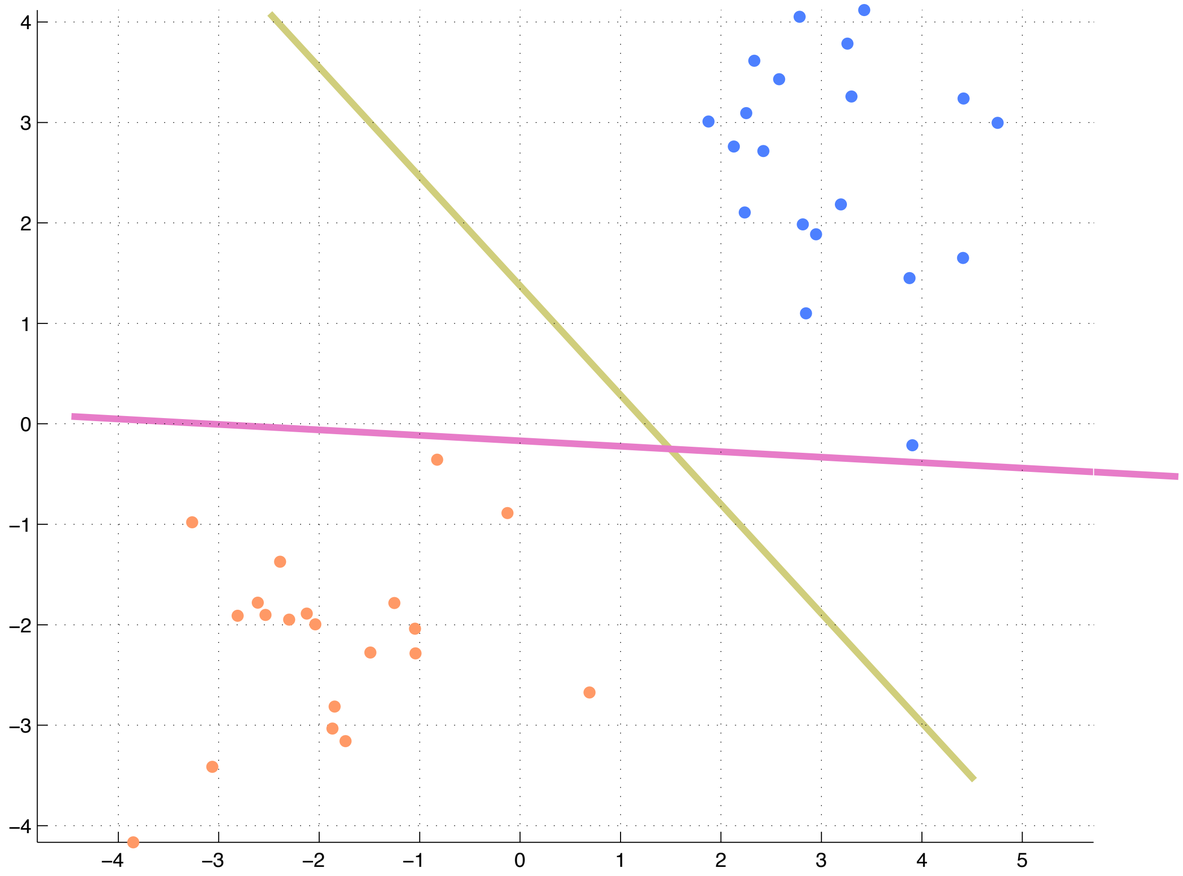
Which line is best?

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



Which one is best?

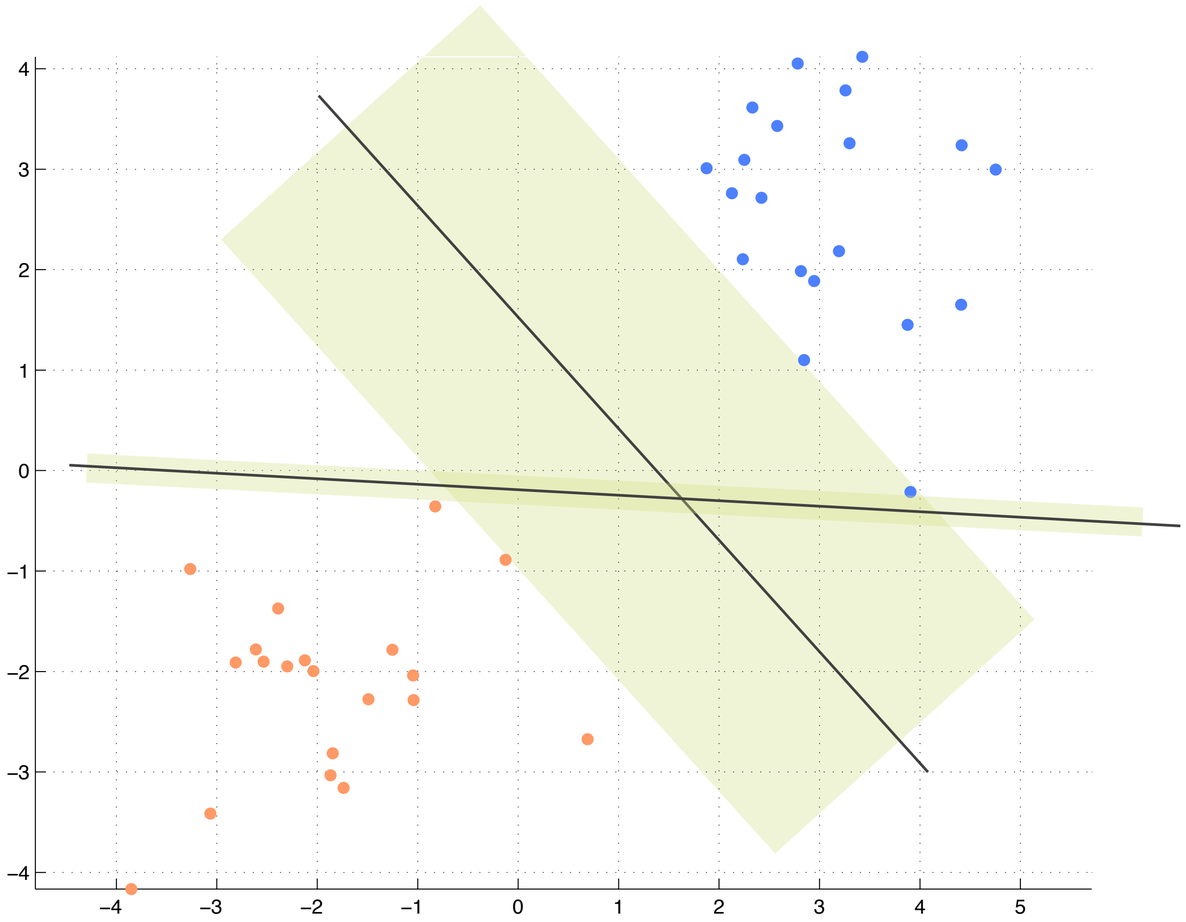
- And why?



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:

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

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

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

  - 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:

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

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

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

Defining the problem

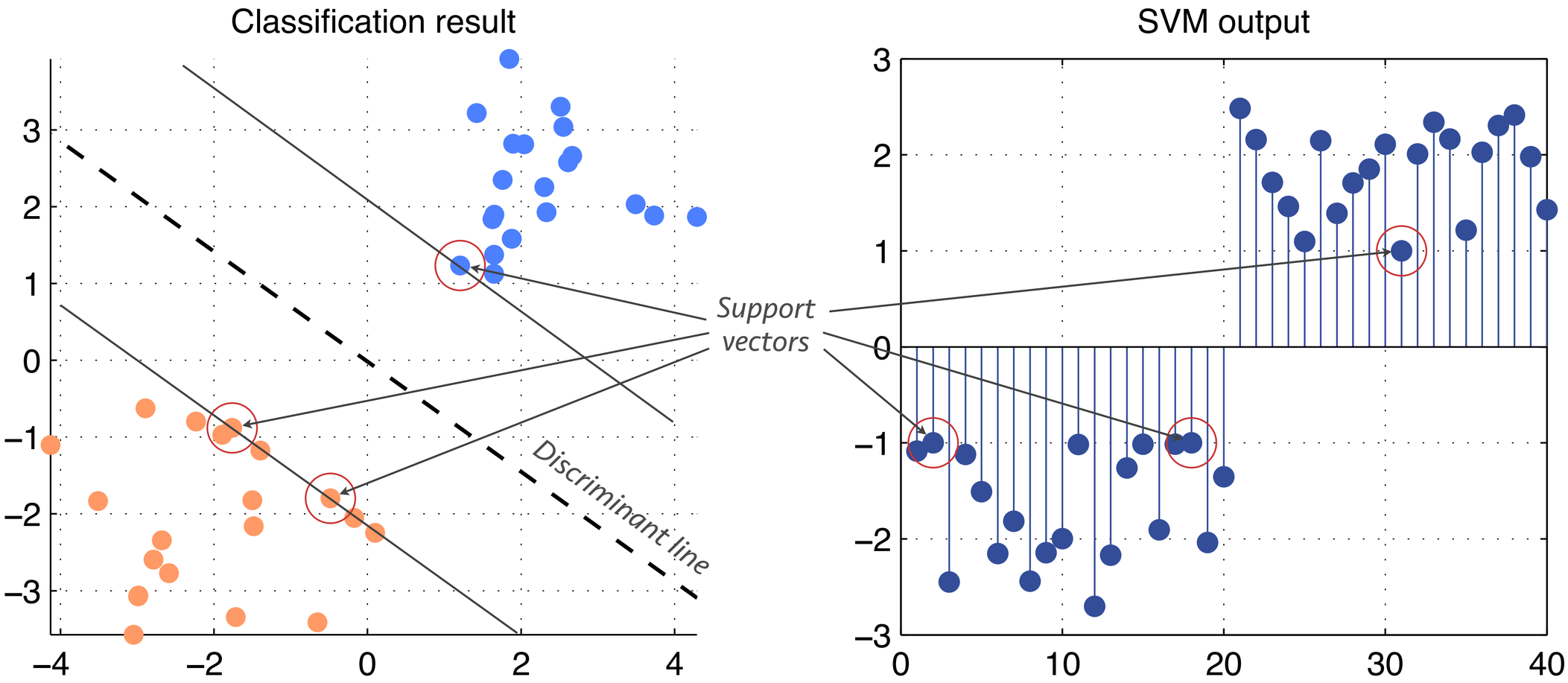
- Quadratic optimization

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

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

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

Example run

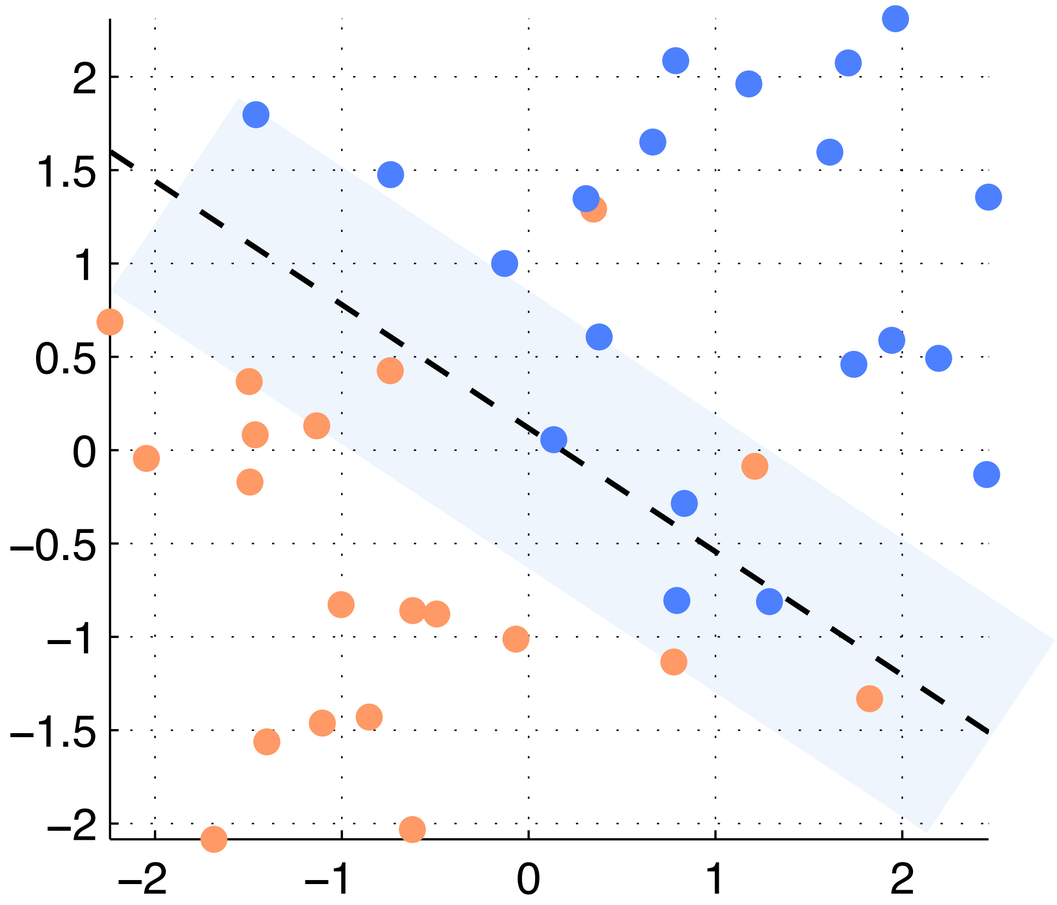

Classification result
SVM output
Support vectors
Discriminant line
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
10
20
30
40

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

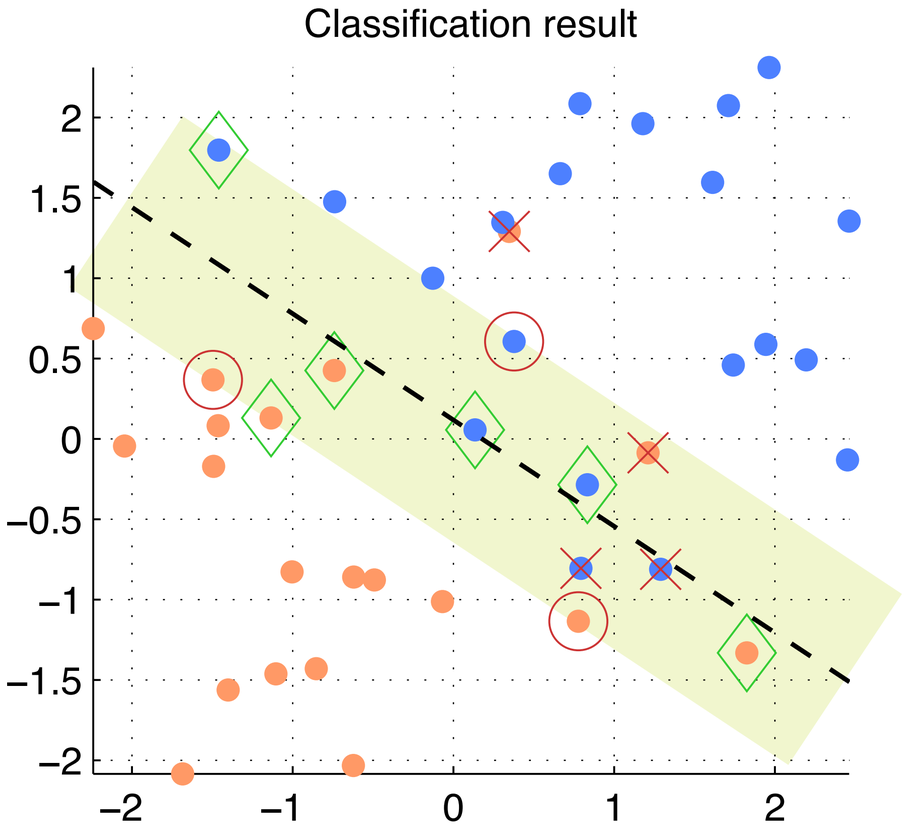


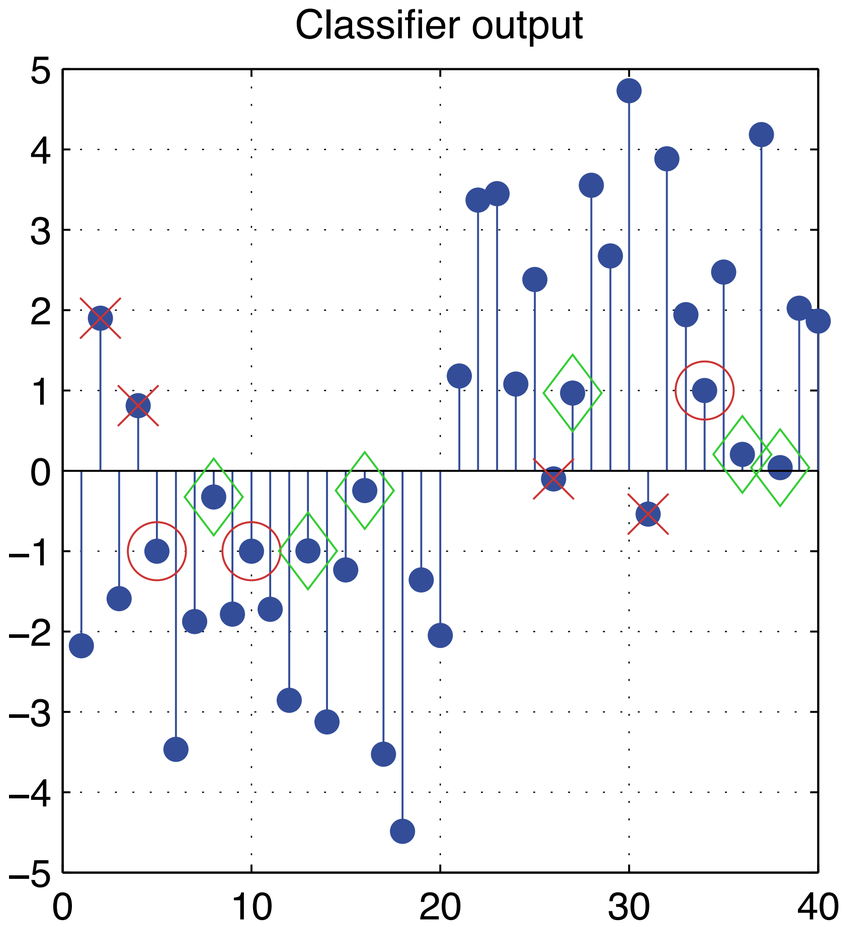
New optimization problem

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

$$\begin{aligned} &\textit{minimize} \quad J(\mathbf{w}, w_0, \xi) = \frac{1}{2}\left\|\mathbf{w}\right\|^2 + C\sum \xi_i \\ &\textit{subject to} \quad y_i(\mathbf{w}^\top \cdot \mathbf{x}_i + w_0) \geq 1 - \xi_i \\ &\qquad\qquad\quad \xi_i \geq 0 \end{aligned}$$

Example run


Classification result
2
1.5
1
0.5
0
−0.5
−1
−1.5
−2
−2
−1
0
1
2


Classifier output
5
4
3
2
1
0
−1
−2
−3
−4
−5
0
10
20
30
40

This can be slow to solve

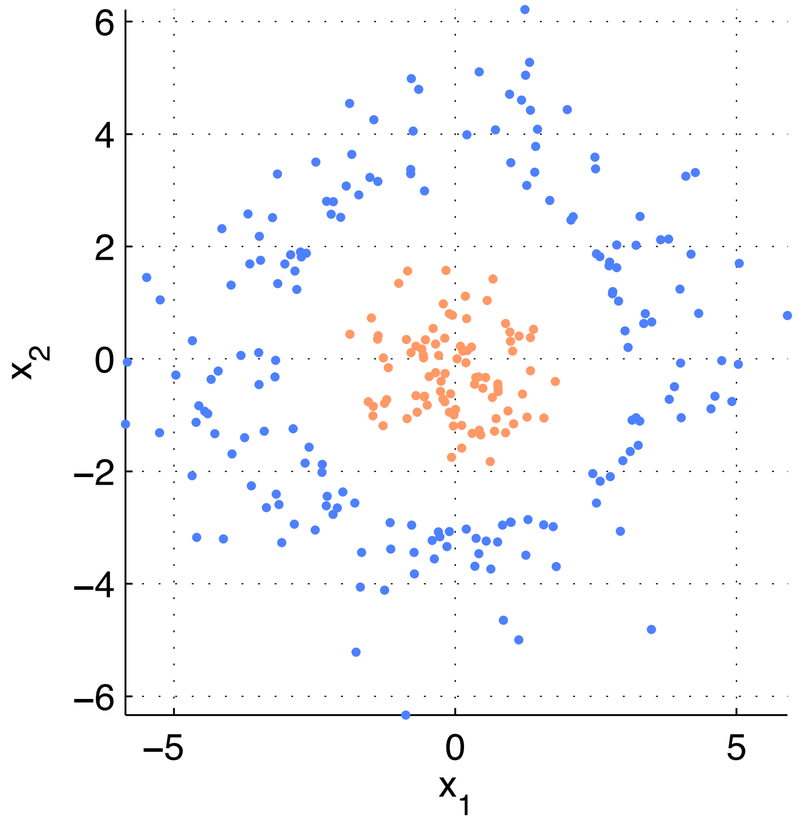
- There are also fast alternatives
  - E.g. when you have a ton of data
- PEGASOS algorithm:
  - 1) Pick k samples at random
  - 2) Get failure cases where $y_i(\mathbf{w}^\top \cdot \mathbf{x}_i) < 1$
  - 3) Tweak weights $\mathbf{w} \leftarrow (1 - h\lambda)\mathbf{w} + (h / k)\mathbf{x}_i y_i^\top$

$$\mathbf{w} \leftarrow \min\left\{1, \frac{1/\lambda}{\|\mathbf{w}\|}\right\}\mathbf{w}$$

  - 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

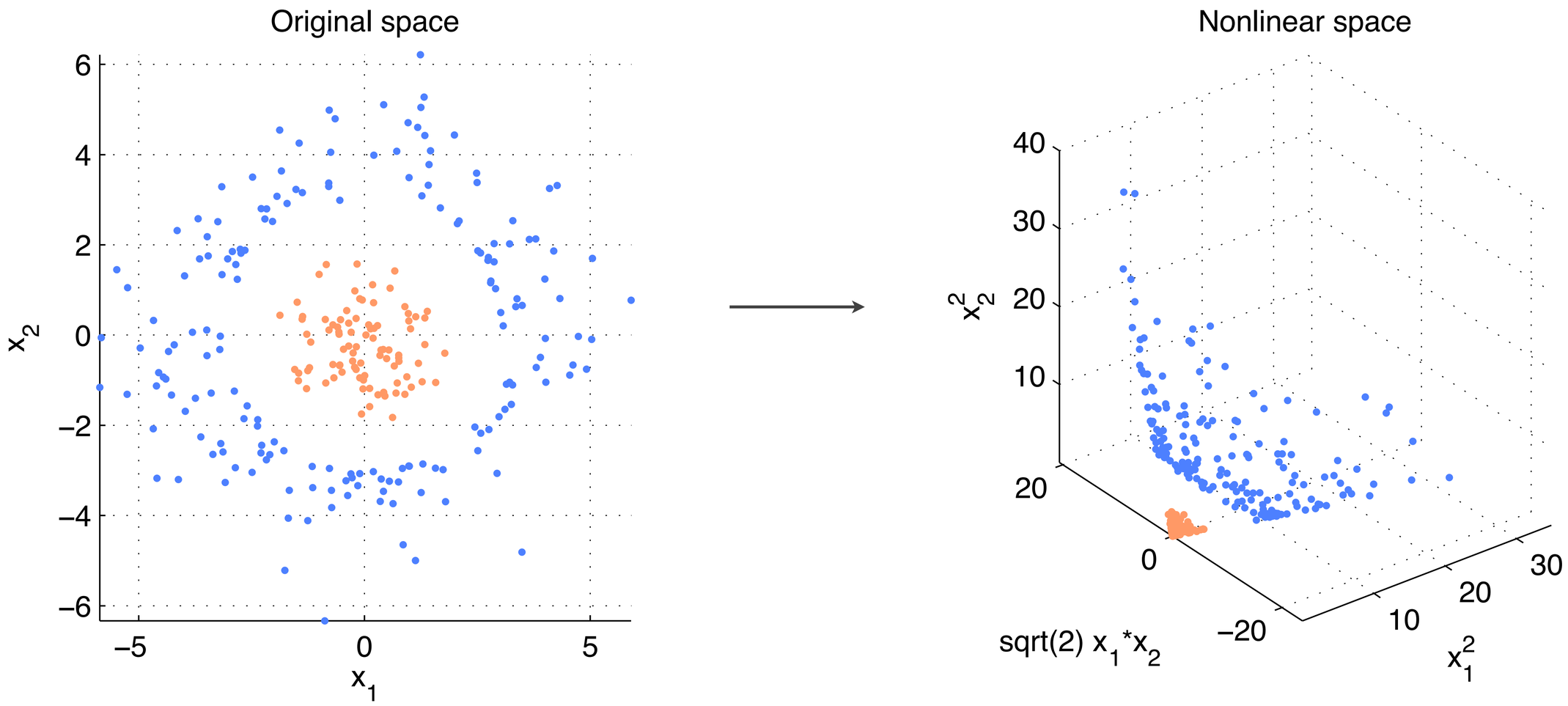


Kernel SVMs

- 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 glitch ...

- The kernelized data will lie on a high-D space
  - That means many more numerical operations
  - And ill-defined estimates

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

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

$$\xi_i \geq 0$$

This can be very high dimensional!

Avoiding $\mathbf{w}$

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

$$\mathbf{w} = \sum_i y_i \alpha_i \mathbf{x}_i$$

- We can now write the classification process as:

$$g(\mathbf{x}_j) = \mathbf{w}^\top \cdot \mathbf{x}_j + w_0 = \left(\sum_i y_i \alpha_i \mathbf{x}_i \right)^\top \cdot \mathbf{x}_j + w_0 = \sum_i y_i \alpha_i \left(\mathbf{x}_i^\top \cdot \mathbf{x}_j \right) + w_0$$

A new formulation of the problem

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

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

- Subject to the correct classification:

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

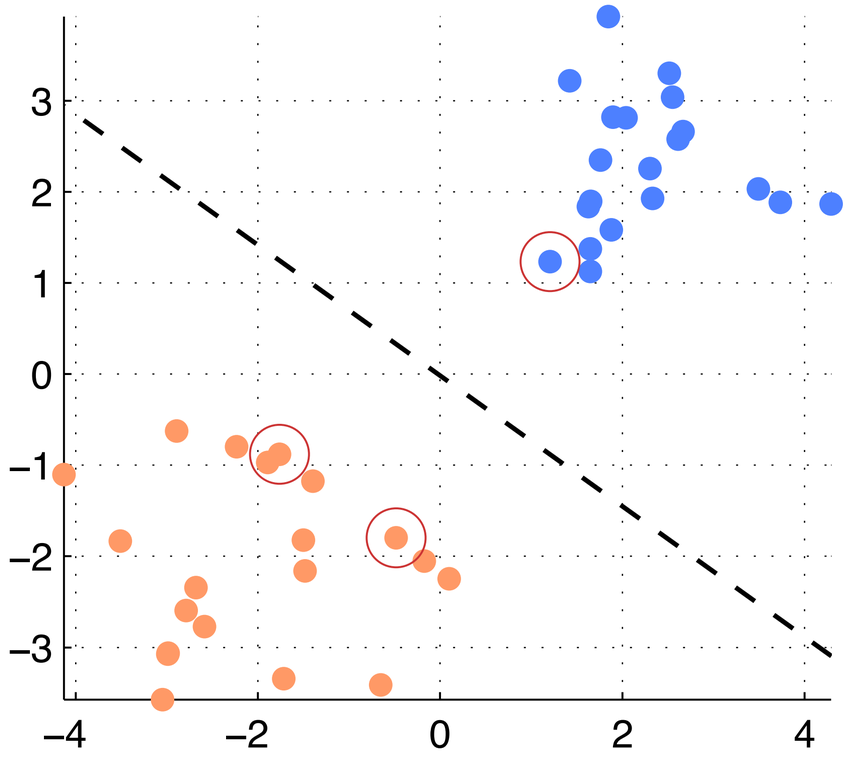
- We now seek to find the optimal α's, not a $\mathbf{w}$

That's not very space efficient!

- Keeping all the $\mathbf{x}_i$'s takes space!

$$g(\mathbf{x}) = \sum_i y_i \alpha_i \left(\mathbf{x}_i^\top \cdot \mathbf{x} \right) + w_0$$

- But we don't need them all, we only need the support vectors!
  - α_i's are mostly zero
  - Non-zero values select the support vectors that define the discriminant



The dual problem

- A formulation without the $\mathbf{w}$ vector, still a quadratic program:

$$\max_{\alpha} \left[\sum \alpha_i - \frac{1}{2} \sum \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \cdot \mathbf{x}_j \right]$$

$$\textit{subject to} \quad \sum \alpha_i y_i = 0$$

$$\alpha_i \geq 0$$

- We only need to estimate the α's
  - For N samples $\mathbf{x} \in \mathbb{R}^d$ we learn N parameters (as opposed to d)
    - But if $N \ll d$, the dual is more efficient

And with slack variables

- If we add slack variables the dual problem becomes:

$$\max_{\alpha}\left[\sum \alpha_i - \frac{1}{2}\sum \alpha_i \alpha_j y_i y_j \mathbf{x}_i^{\top} \cdot \mathbf{x}_j\right]$$

$$\textit{subject to} \quad \sum \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C$$

  - Slack variables are gone!
    - Easier optimization, same tradeoffs

Back to the problem at hand

- From the dual formulation we have:

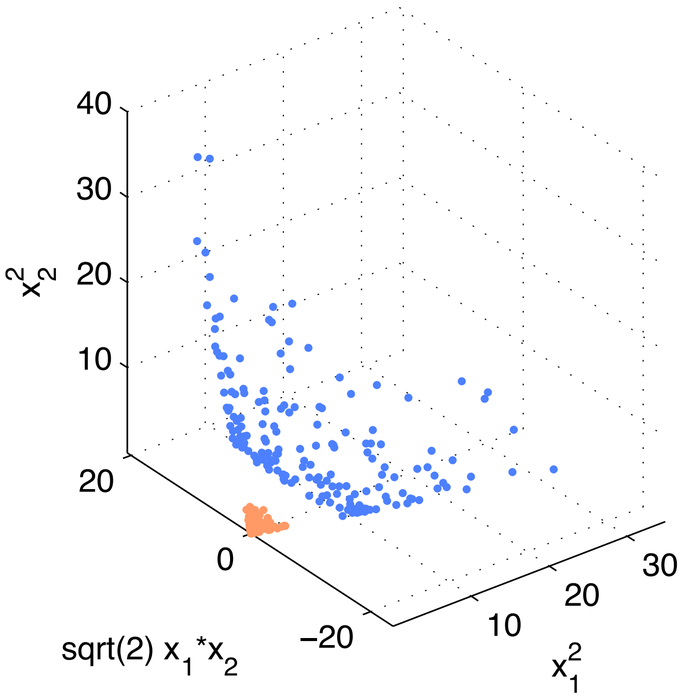
$$g(\mathbf{z}_j) = \mathbf{w}^\top \cdot \mathbf{z}_j + w_0 = \sum \alpha_i y_i \left(\mathbf{z}_i^\top \cdot \mathbf{z}_j \right) + w_0$$

- We can easily show that for:

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

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

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



Similar such maps

- Polynomial:

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

- Radial basis functions:

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

- Hyperbolic tangent:

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

Kernel SVMs

- Use the dual form to optimize:

$$\max_{\alpha}\left[\sum \alpha_i - \frac{1}{2}\sum \alpha_i \alpha_j y_i y_j K\left(\mathbf{x}_i, \mathbf{x}_j\right)\right]$$

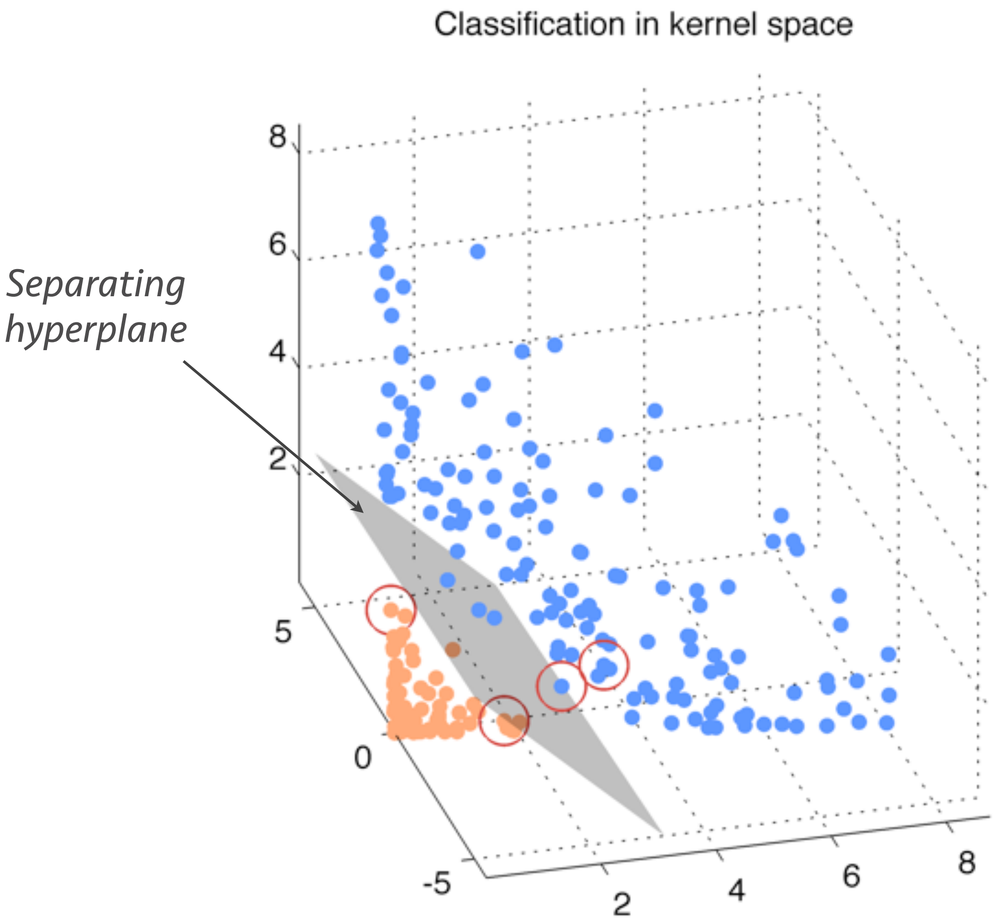
$$\textit{subject to} \quad \sum \alpha_i y_i = 0$$

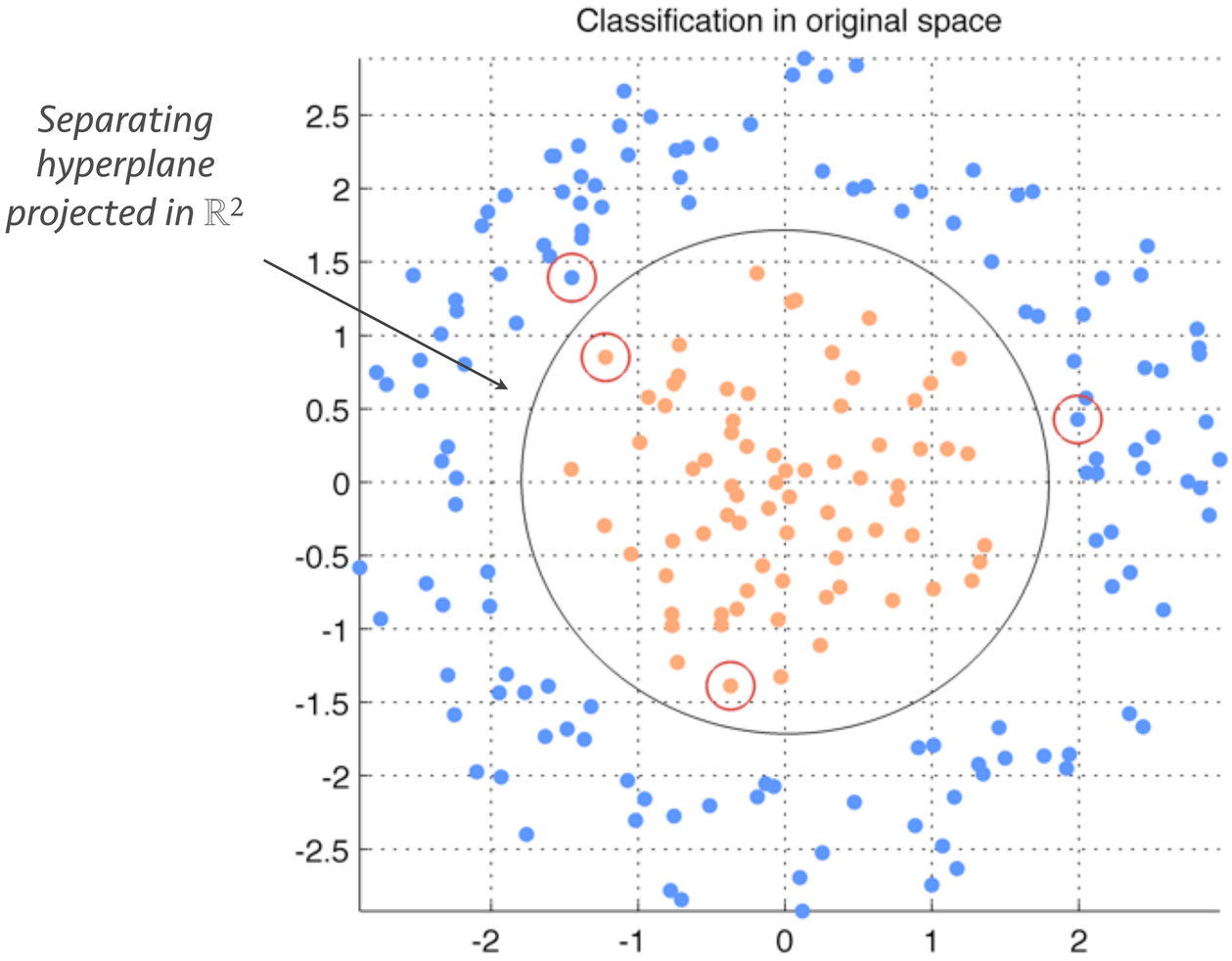
$$0 \le \alpha_i \le C$$

- And classify with:

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

Example case





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

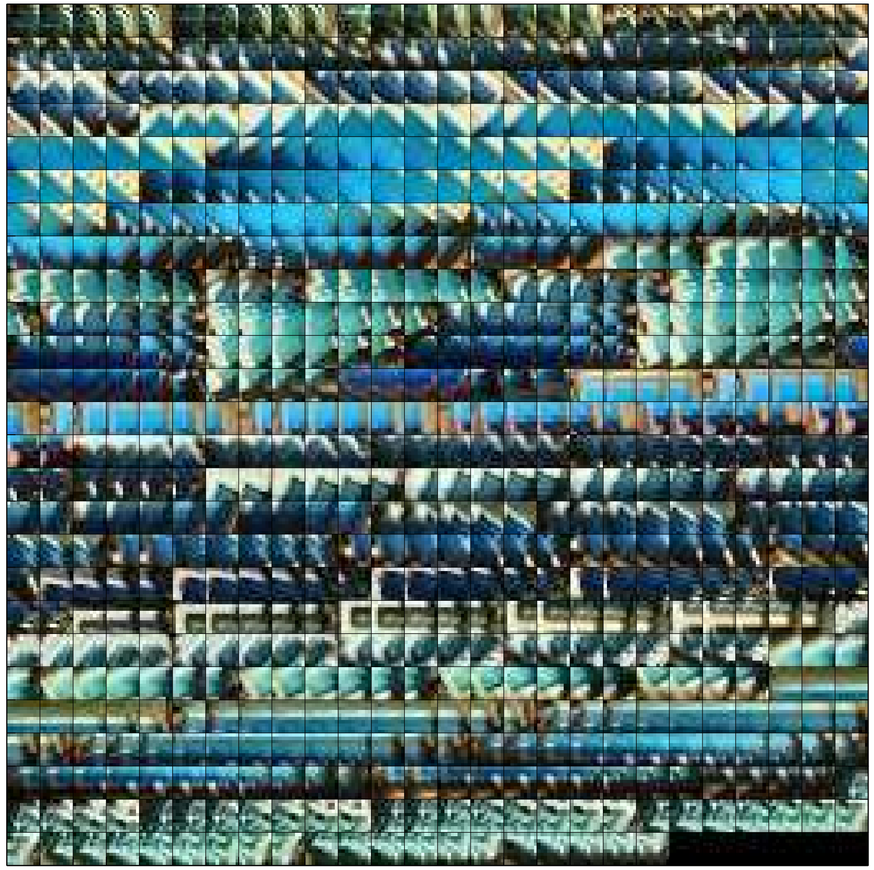
Non-linear classifiers at work

- 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 $\{-1,1\}$

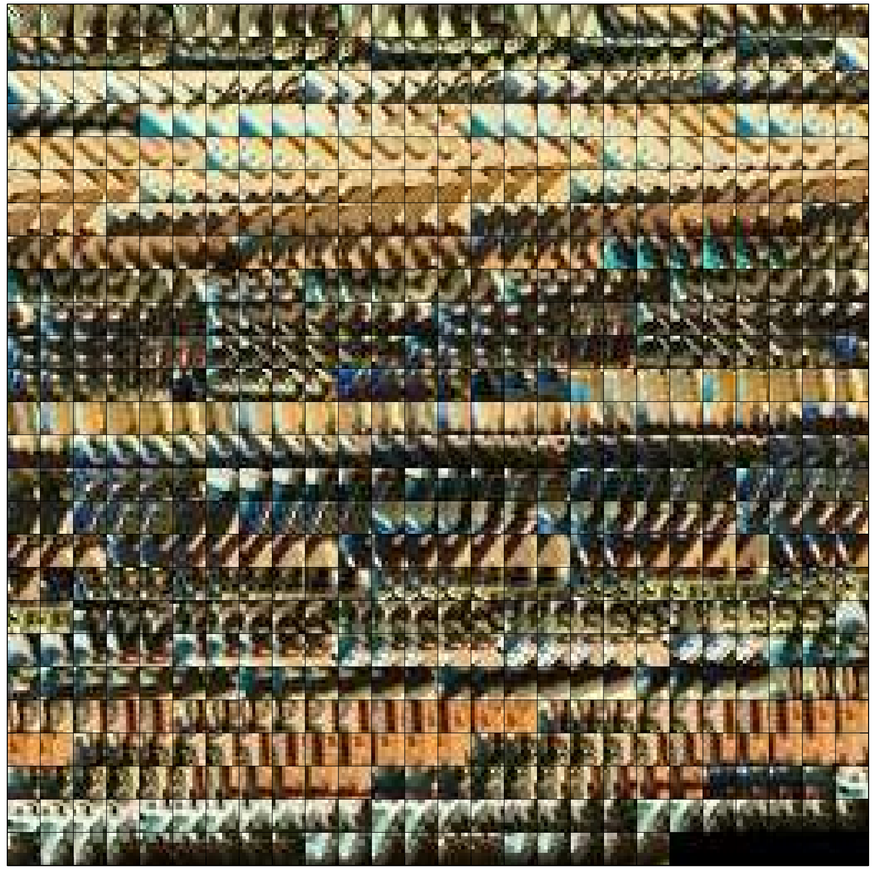
Step 1: Make training data

- Collect examples of positives and falses

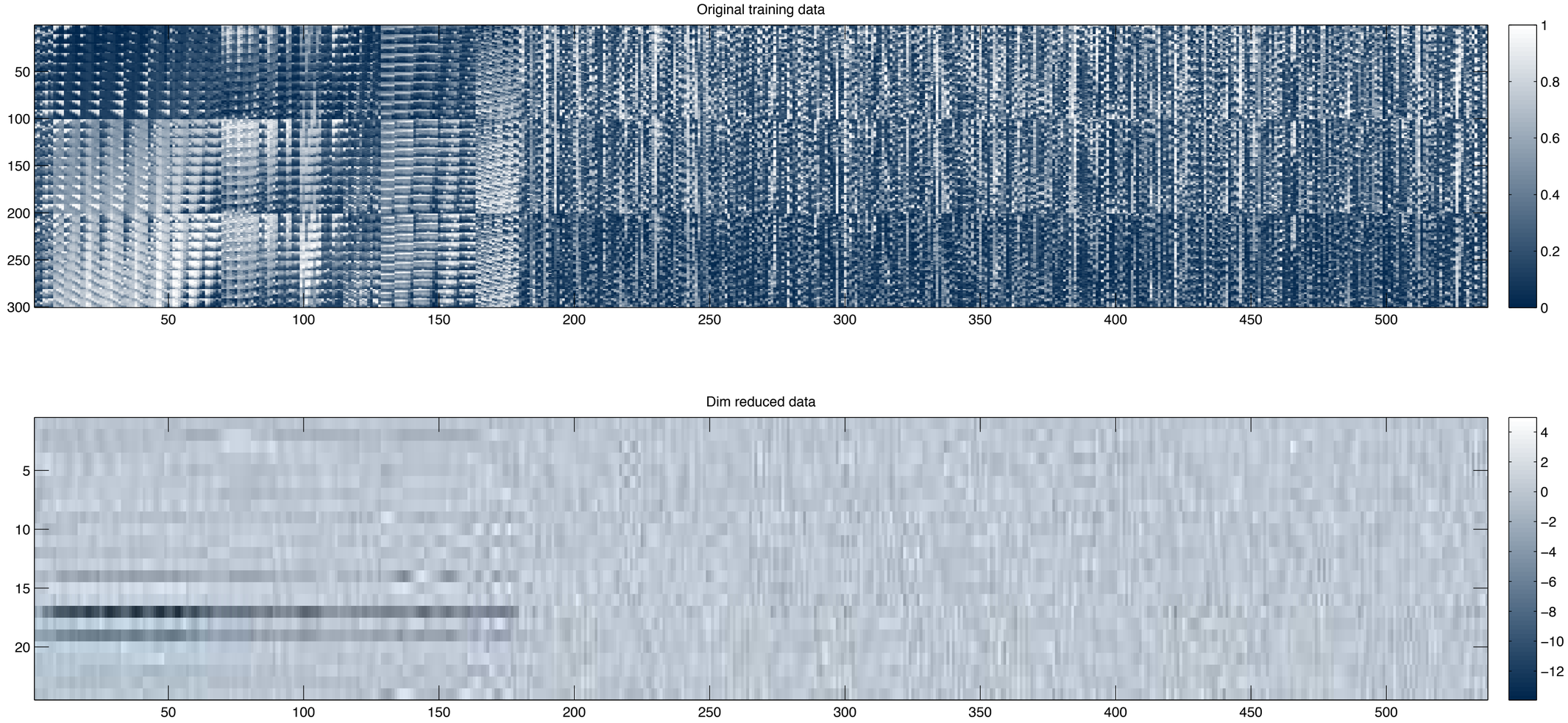
670 + training patches



670 – training patches



Step 2: Drop the dimensions (optional)



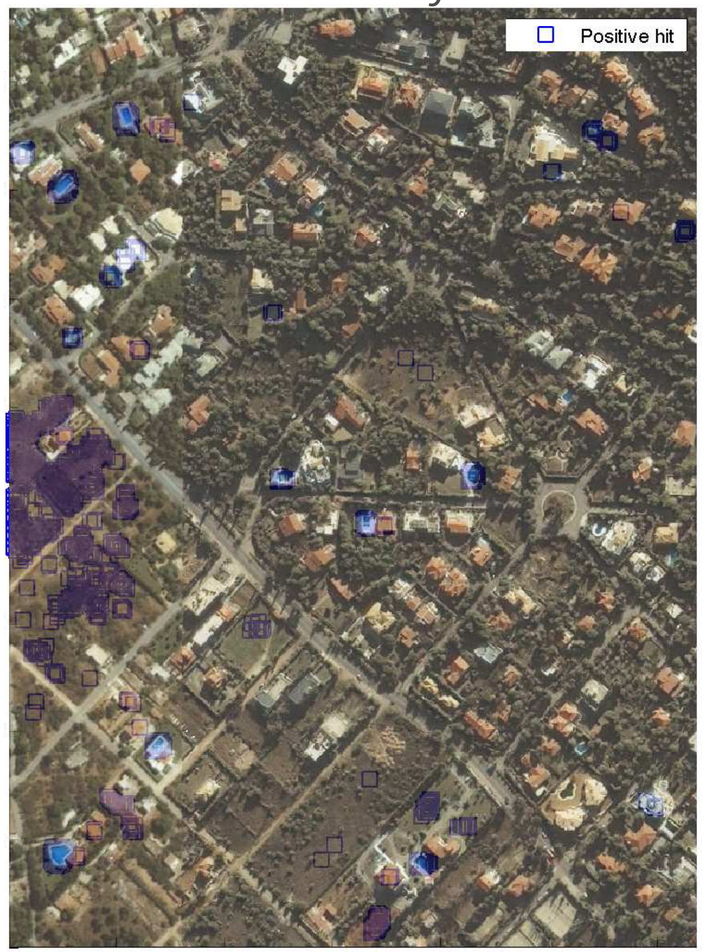
Step 3: Learn various classifiers

- As you 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?

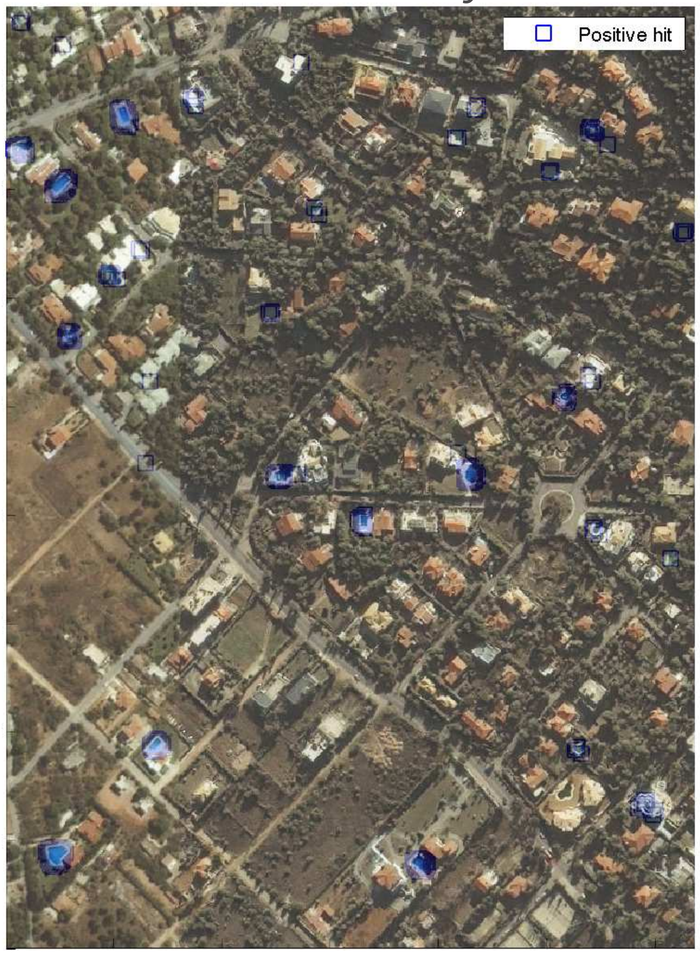
Step 4: Evaluate on unseen data

Normalized cross-correlation

Linear classifier



Nonlinear classifier



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

Final projects

- Start thinking about them
  - If unsure come talk to me or the TAs
  - In the next homework we will ask for a project proposal

- Make friends!
  - Aiming for ~25 slots, which means groups of 2 or 3