

# Review for exam #2

**Jason G. Fleischer, Ph.D.**

**Asst. Teaching Professor**

**Department of Cognitive Science, UC San Diego**

**jfleischer@ucsd.edu**

 **@jasongfleischer**

**<https://jgfleischer.com>**

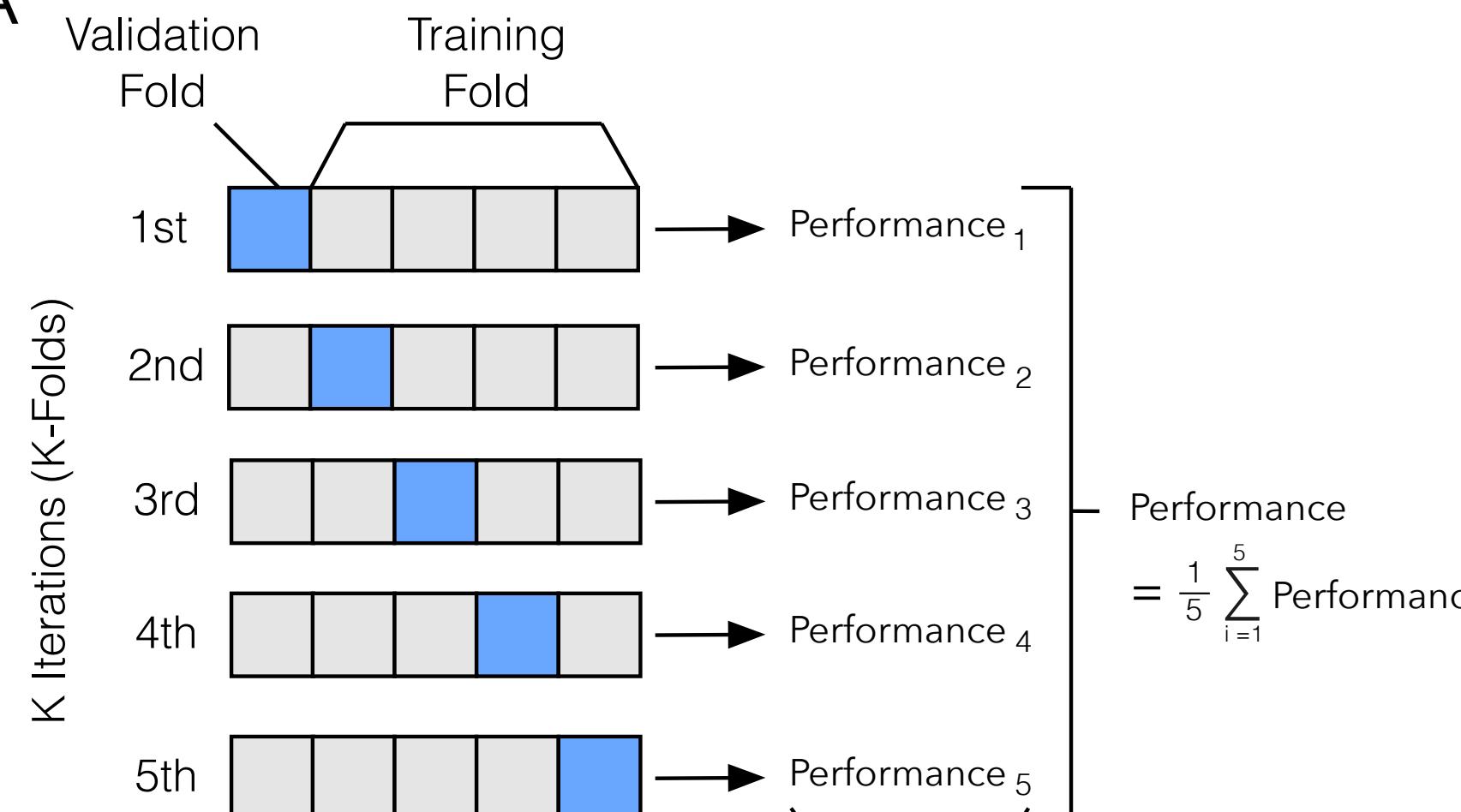
# NN links

- [https://pytorch.org/tutorials/beginner/deep learning 60min blitz.html](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)
- <https://huggingface.co/learn/nlp-course/chapter1/1>
- <https://course.fast.ai/>

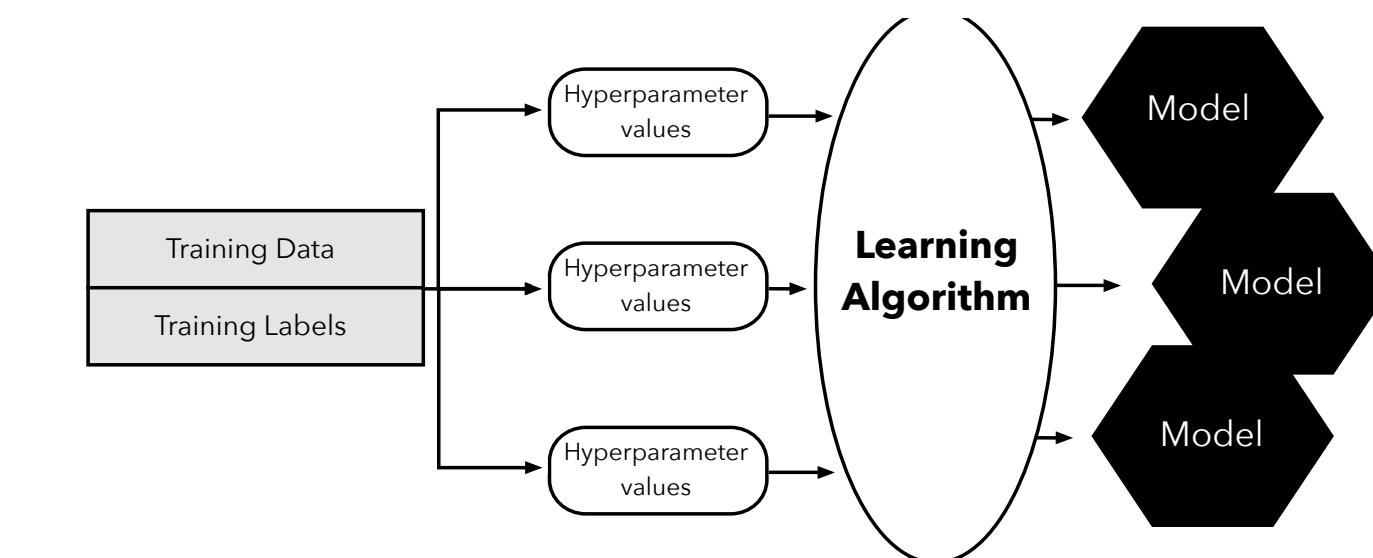
# Model selection

# k-Fold Cross-Validation

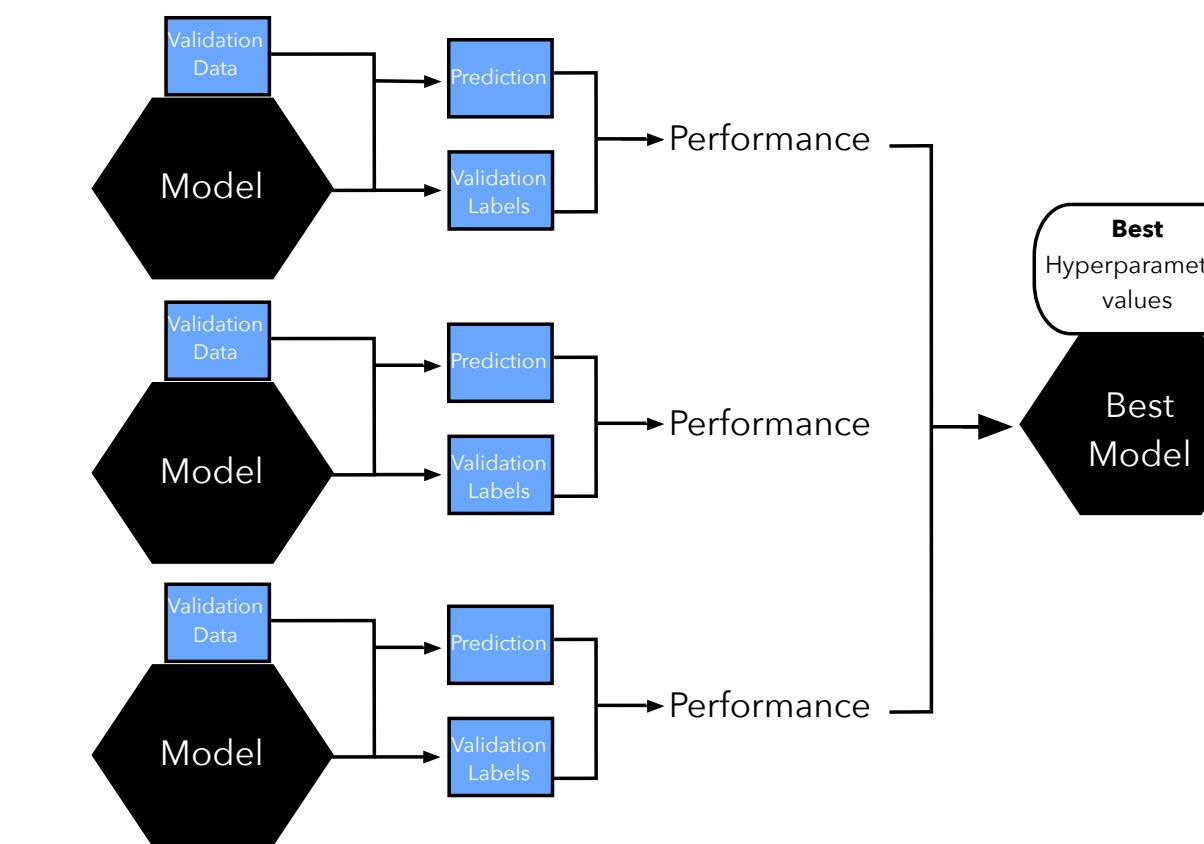
A



2



3

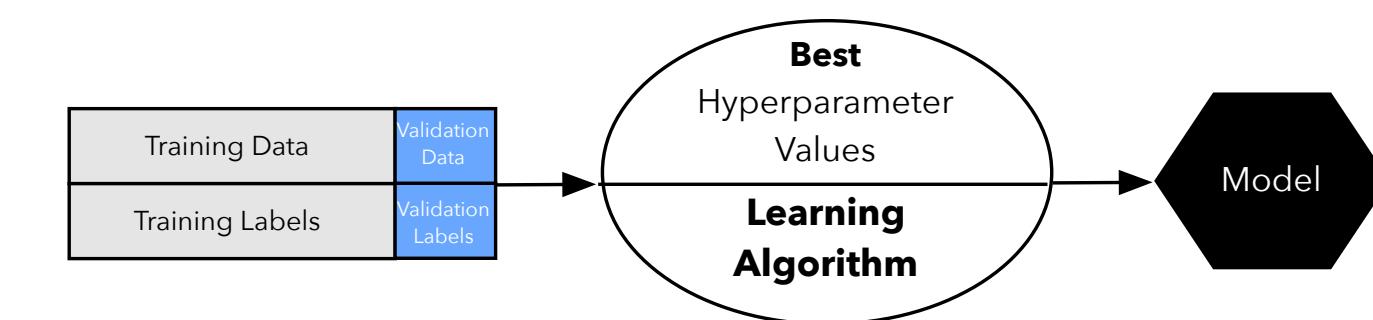


Default method for our project:

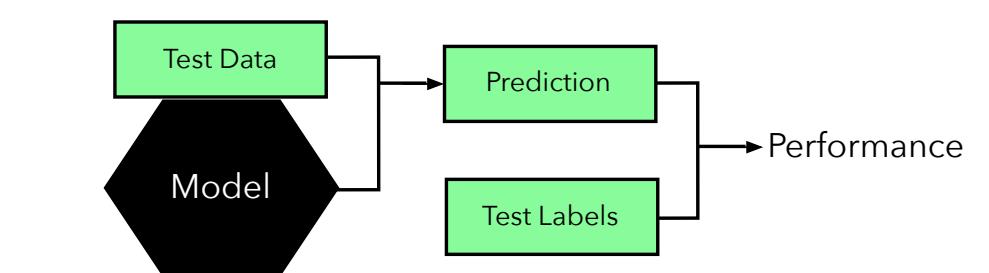
For each trial:

- training set ~ sample 5k with replacement from entire dataset
- Grid search of hyper parameters using 5-fold cross validation on the training set
- Select best model from grid, train on entire training set
- Evaluate best model on the test set (everything not sampled for training)

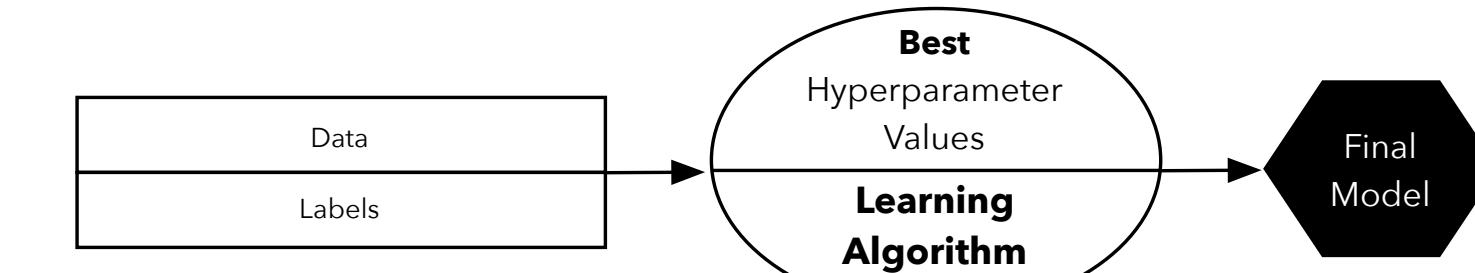
4



5



6



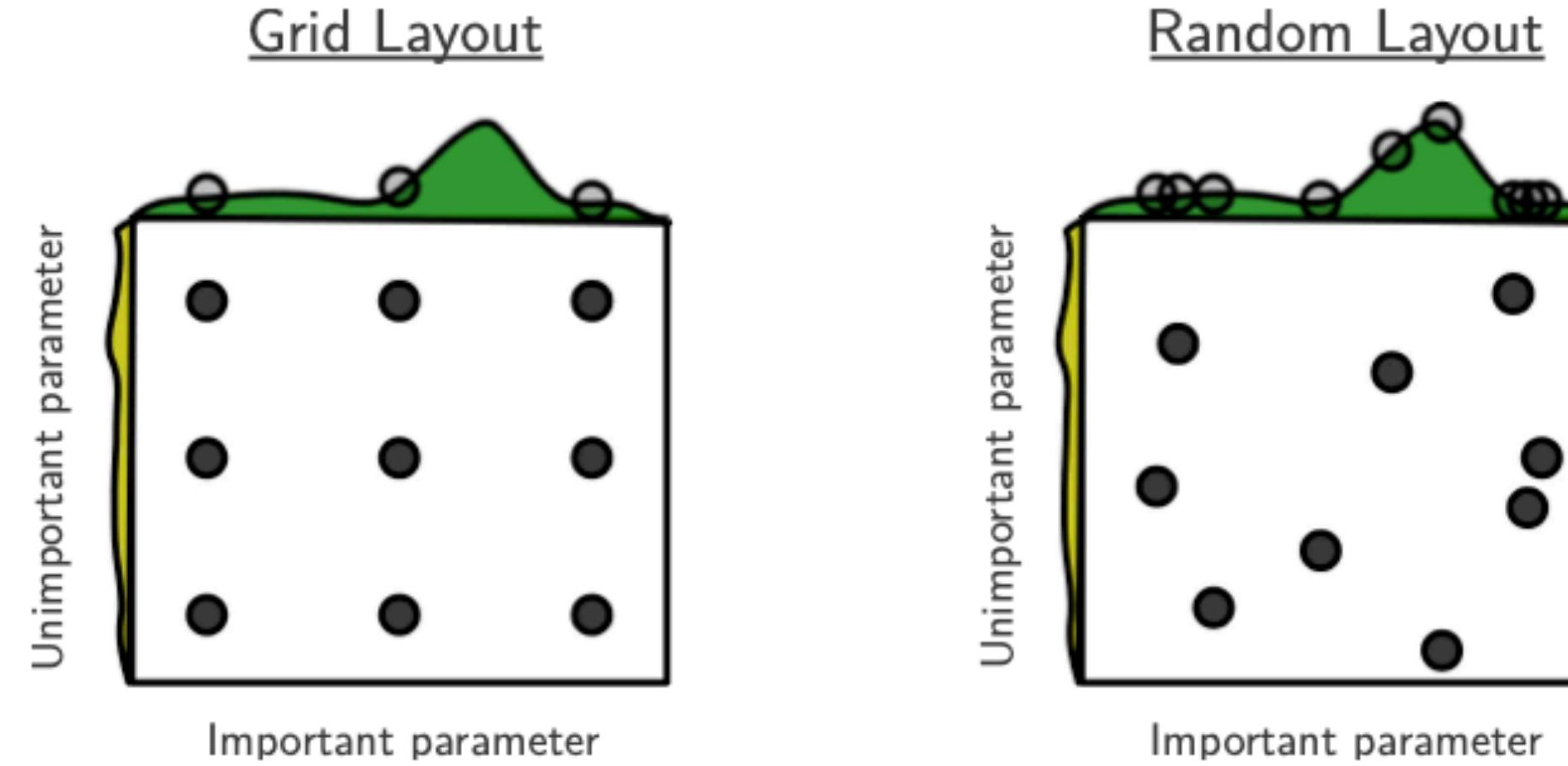
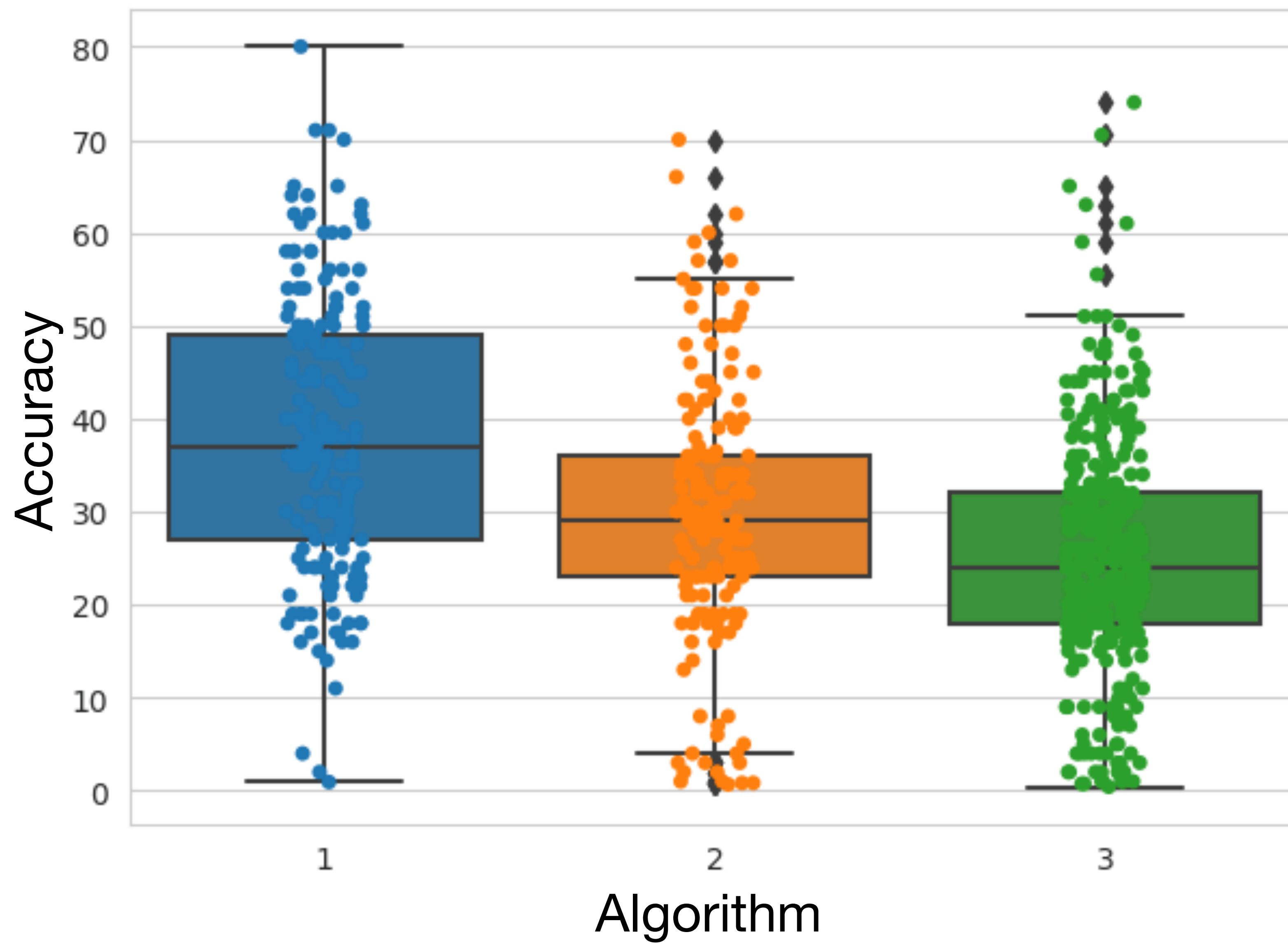


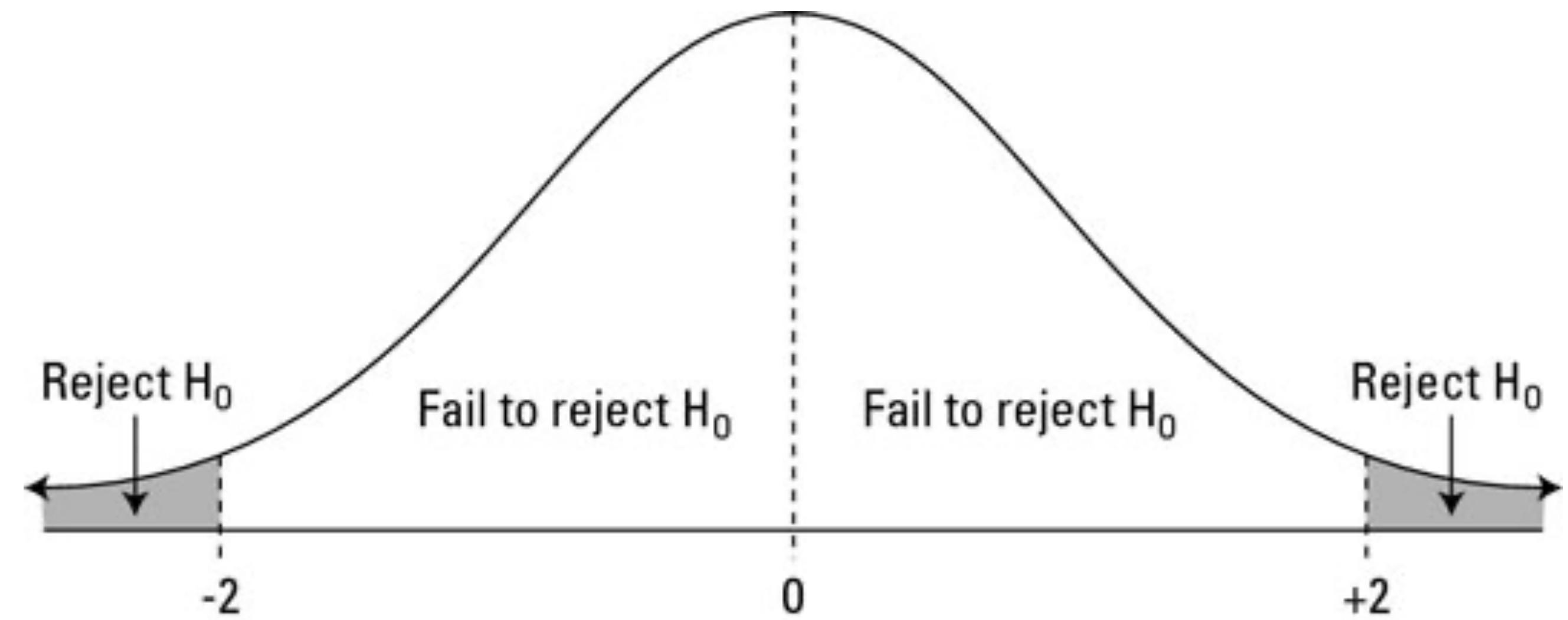
Figure 1: Grid and random search of nine trials for optimizing a function  $f(x,y) = g(x) + h(y) \approx g(x)$  with low effective dimensionality. Above each square  $g(x)$  is shown in green, and left of each square  $h(y)$  is shown in yellow. With grid search, nine trials only test  $g(x)$  in three distinct places. With random search, all nine trials explore distinct values of  $g$ . This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1), 281-305.



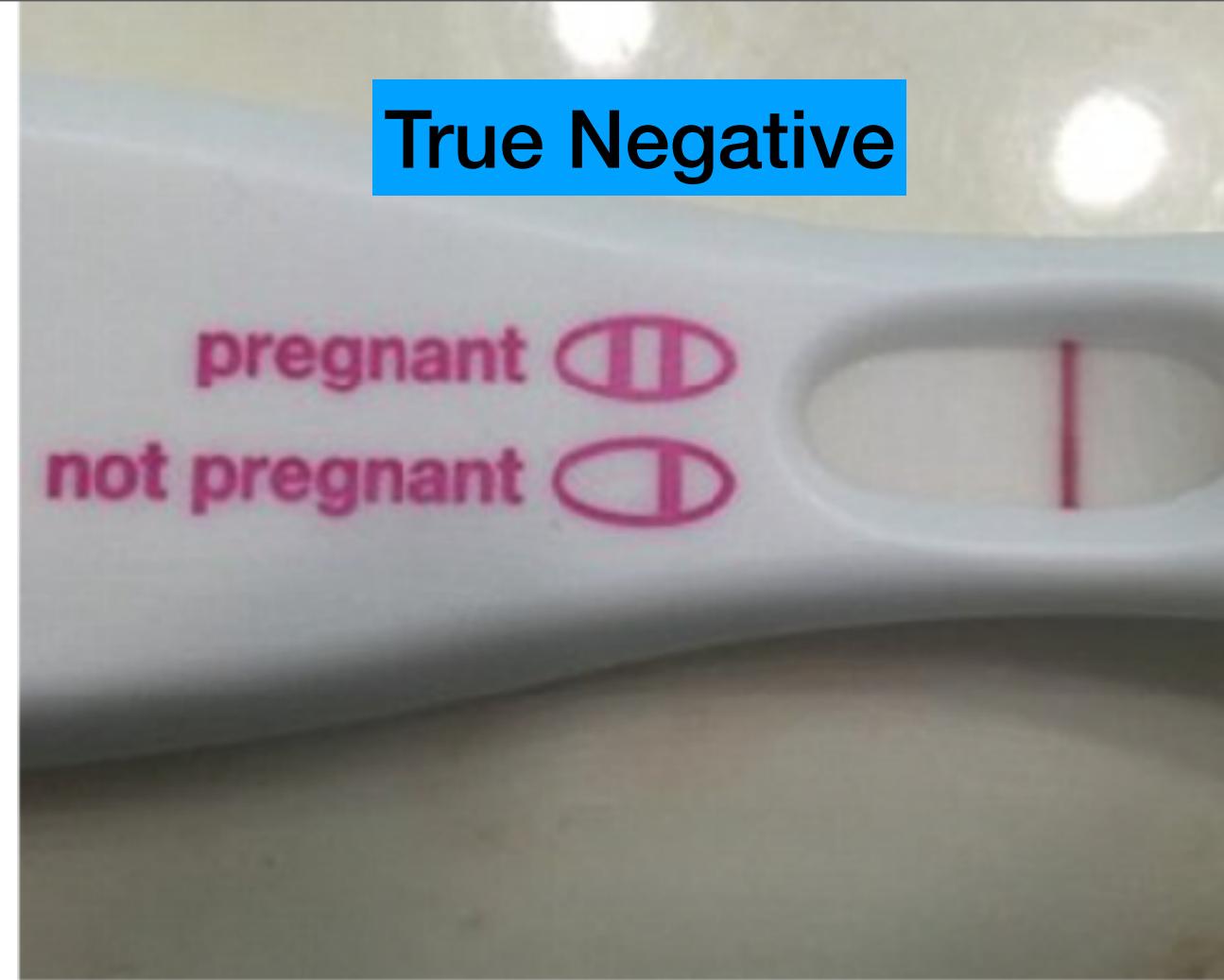
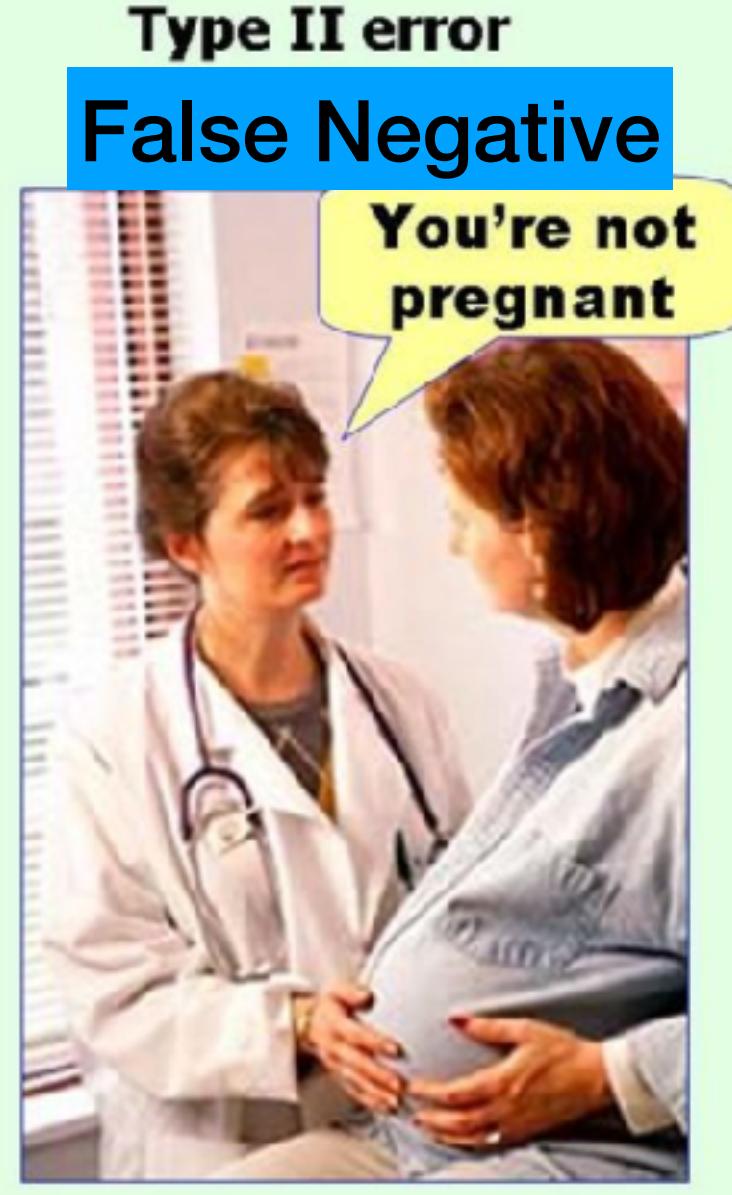
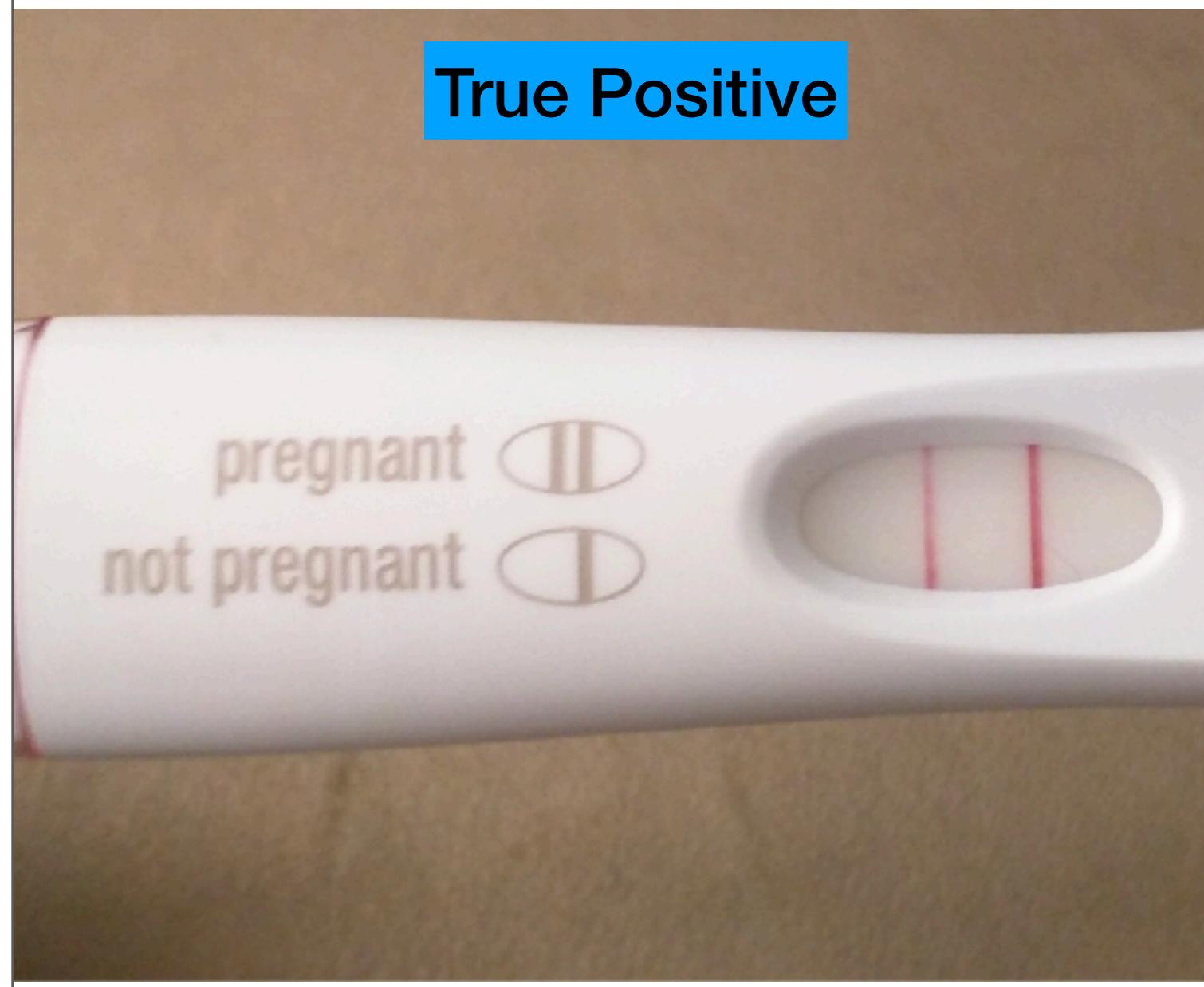
# Null hypothesis statistical testing

- Smaller p-values support for alternative hypothesis
- Larger is inconclusive
- Ignores effect size!!!@!!! Is the difference practically important?
- Many statistical tests, details matter
- Don't take it too seriously!



# Metrics

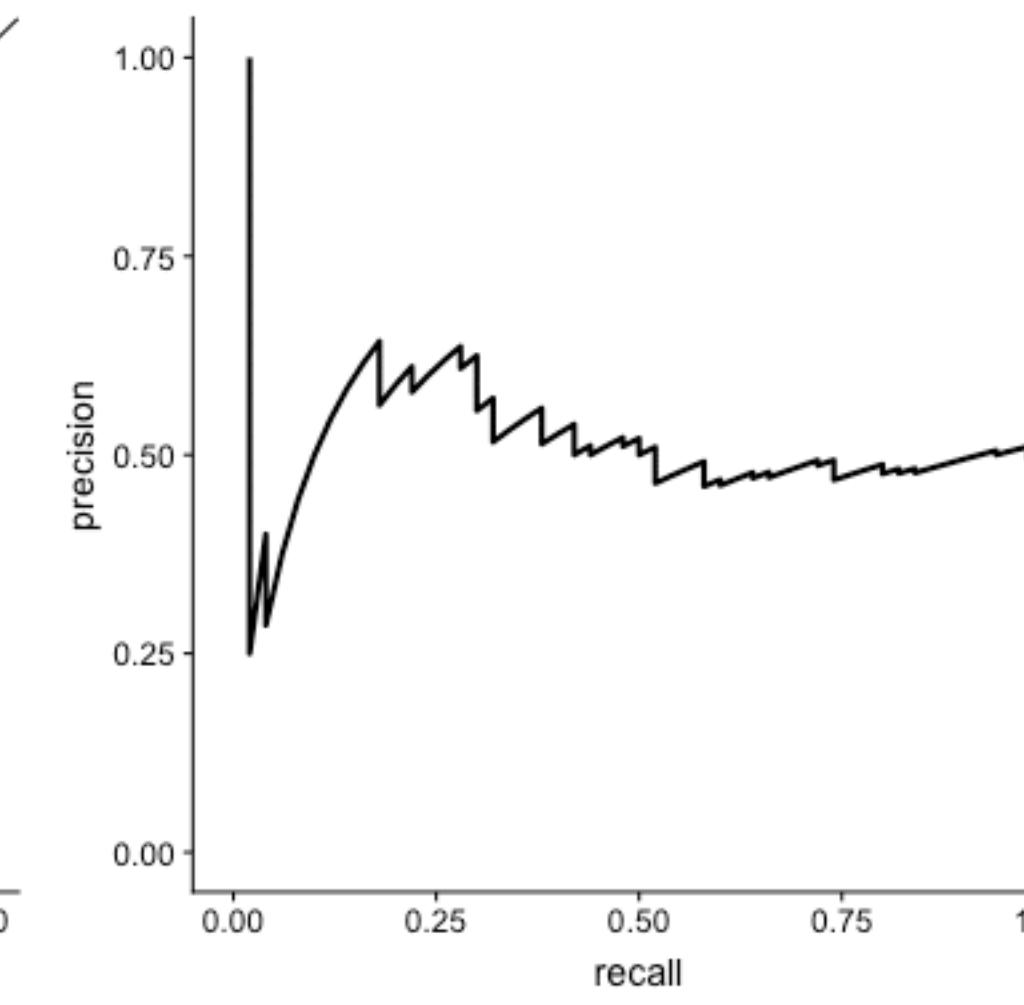
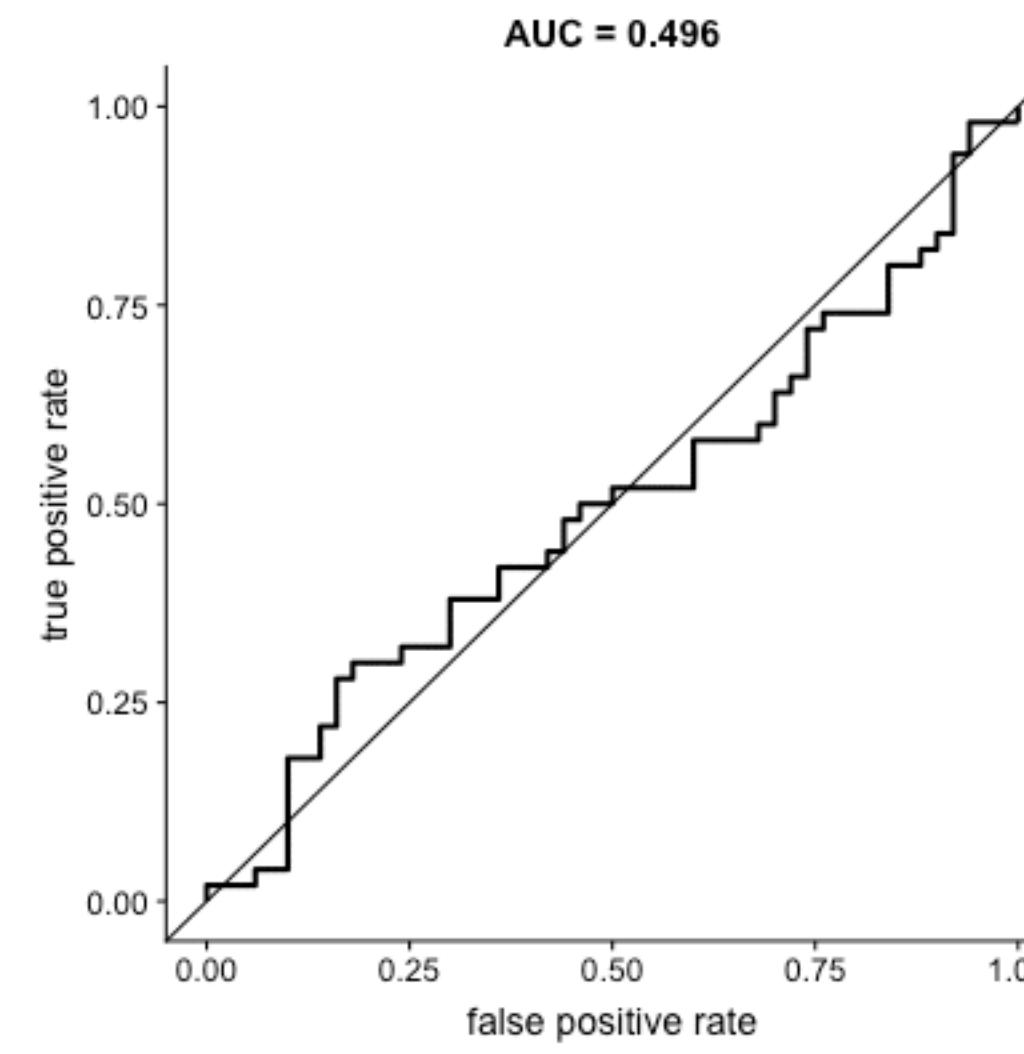
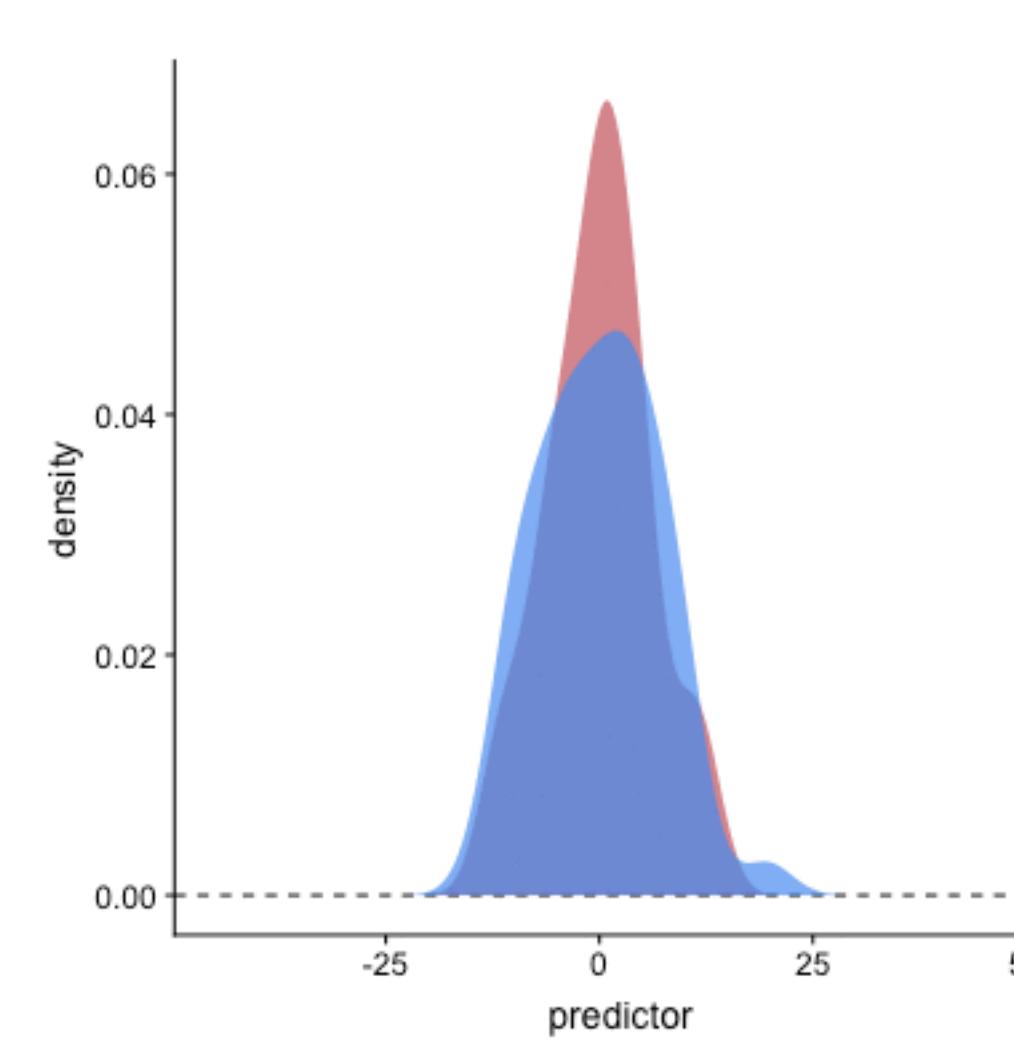
## Our algorithm

	-	+
-	 <p>True Negative</p>	 <p>Type I error False Positive</p>
+	 <p>Type II error False Negative You're not pregnant</p>	 <p>True Positive</p>
Actual class		

	True condition				
Total population	Condition positive	Condition negative	Prevalence = $\frac{\sum \text{Condition positive}}{\sum \text{Total population}}$	Accuracy (ACC) = $\frac{\sum \text{True positive} + \sum \text{True negative}}{\sum \text{Total population}}$	
Predicted condition	Predicted condition positive	Predicted condition negative	Positive predictive value (PPV), Precision = $\frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\sum \text{False positive}}{\sum \text{Predicted condition positive}}$	
Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\sum \text{False negative}}{\sum \text{Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$	
	True positive rate (TPR), Recall, Sensitivity, probability of detection, Power $= \frac{\sum \text{True positive}}{\sum \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm $= \frac{\sum \text{False positive}}{\sum \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) $= \frac{\text{LR+}}{\text{LR-}}$	$F_1 \text{ score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
	False negative rate (FNR), Miss rate $= \frac{\sum \text{False negative}}{\sum \text{Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\sum \text{True negative}}{\sum \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$		

# ROC and PR curves

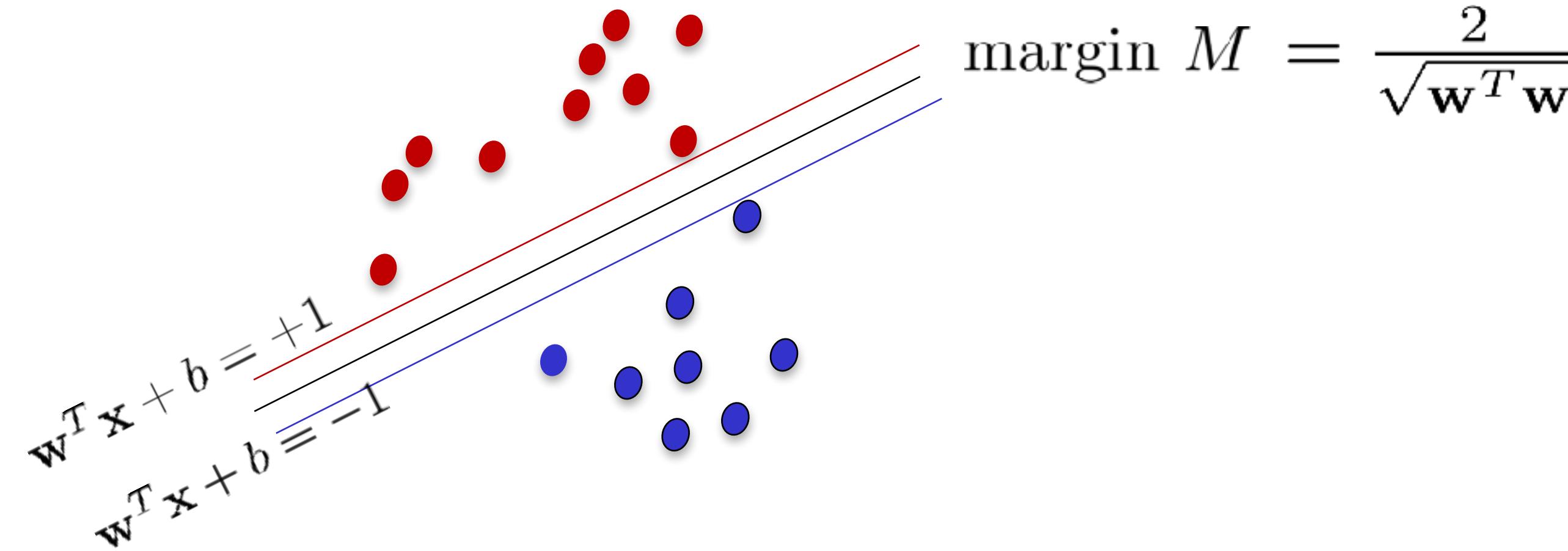
Both are ways to look at the classifier performance for TP/TN/FP/FN



# **SVM**

# SVM

---



Separable case: all positive and negative points are perfectly separable.

Maximizing  $\frac{2}{\sqrt{\mathbf{w}^T \mathbf{w}}}$  is equivalent to minimizing  $\mathbf{w}^T \mathbf{w} = \|\mathbf{w}\|^2$

$$S_{training} = \{(\mathbf{x}_i, y_i), i = 1..n\}$$

Find:  $\arg \min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2$     subject to  $y_i(\mathbf{w}^T \mathbf{x} + b) - 1 \geq 0$

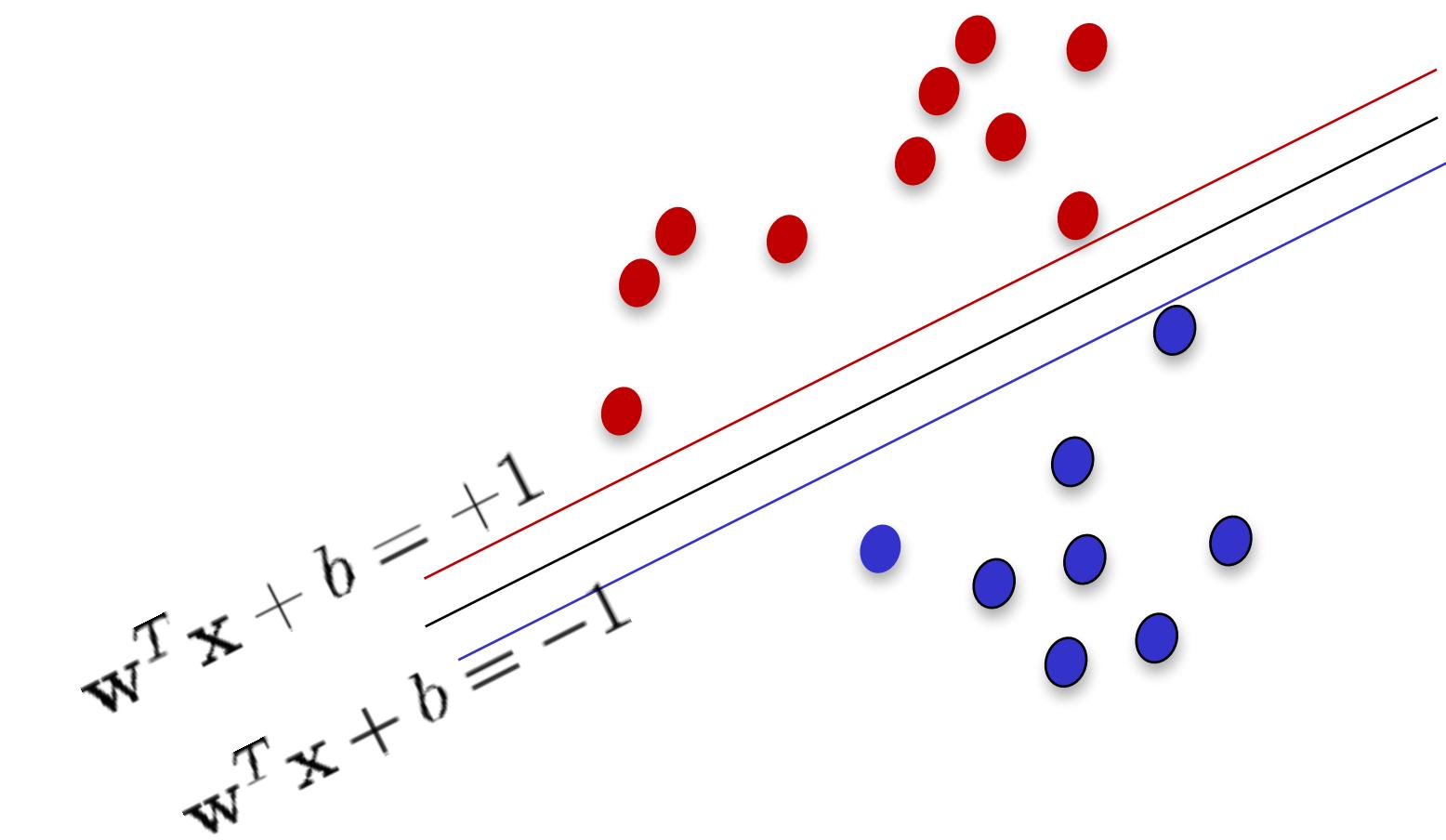
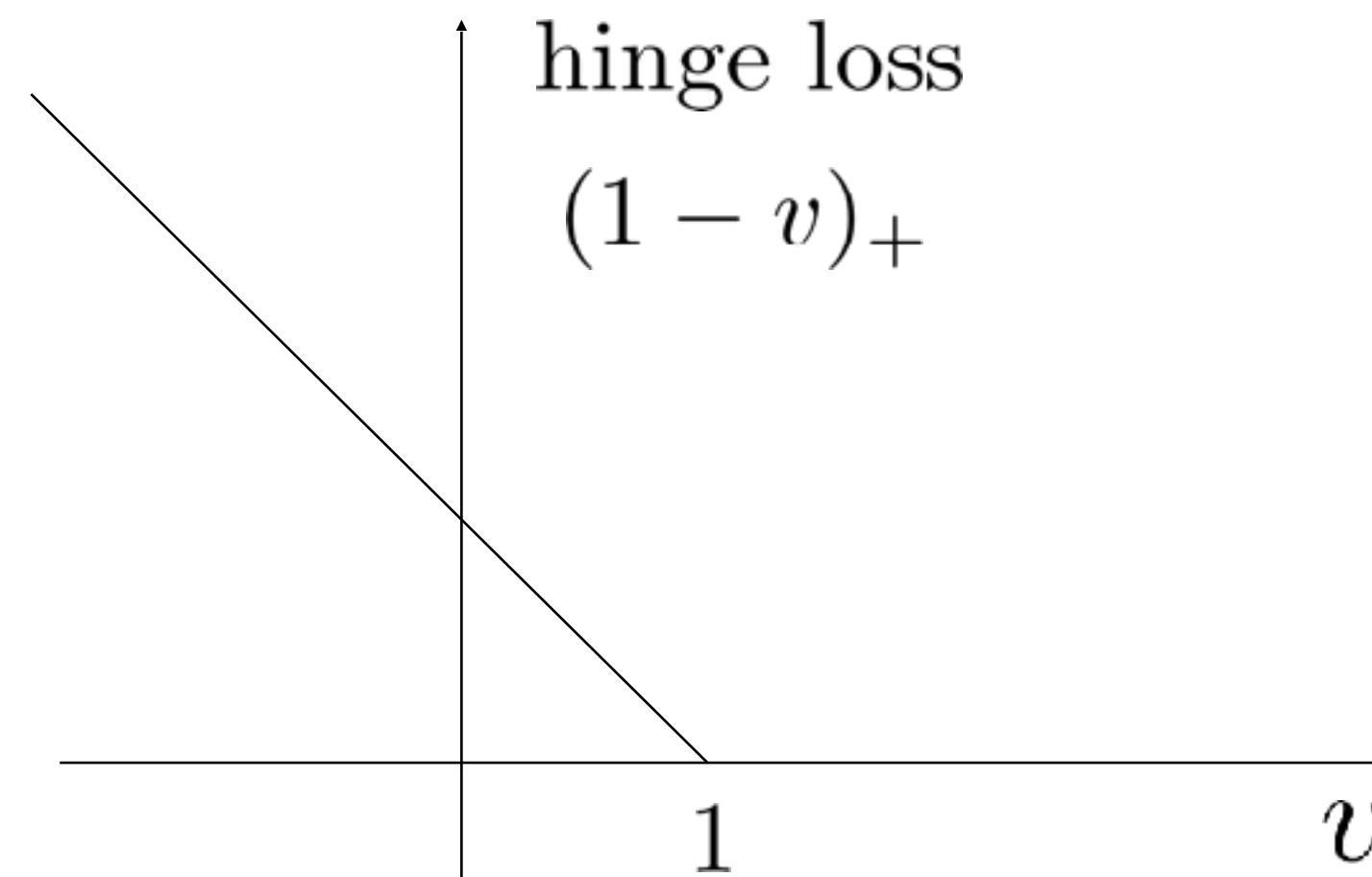
# Hinge Loss

---

Find:  $\arg \min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2$

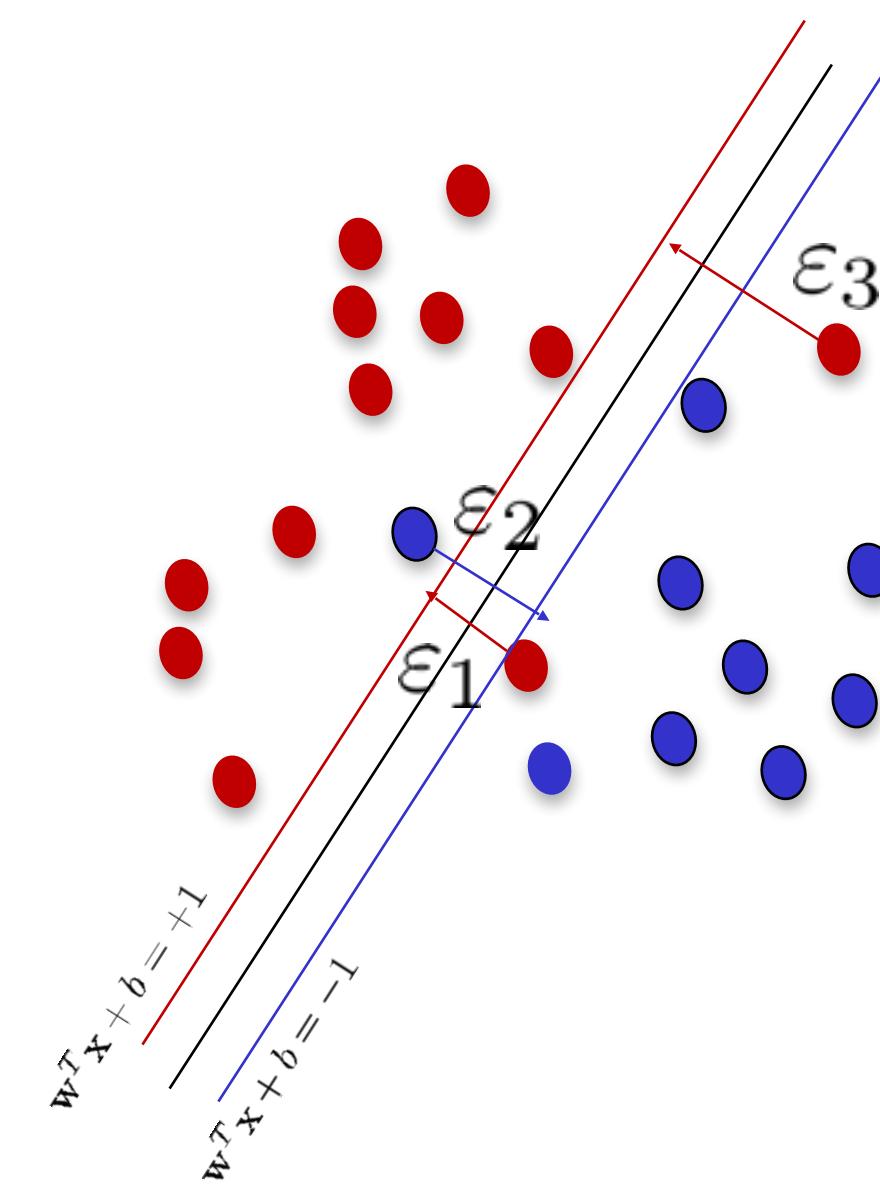
subject to  $y_i(\mathbf{w}^T \mathbf{x} + b) - 1 \geq 0$

Hinge:  $(1 - v)_+ = \max(0, 1 - v)$

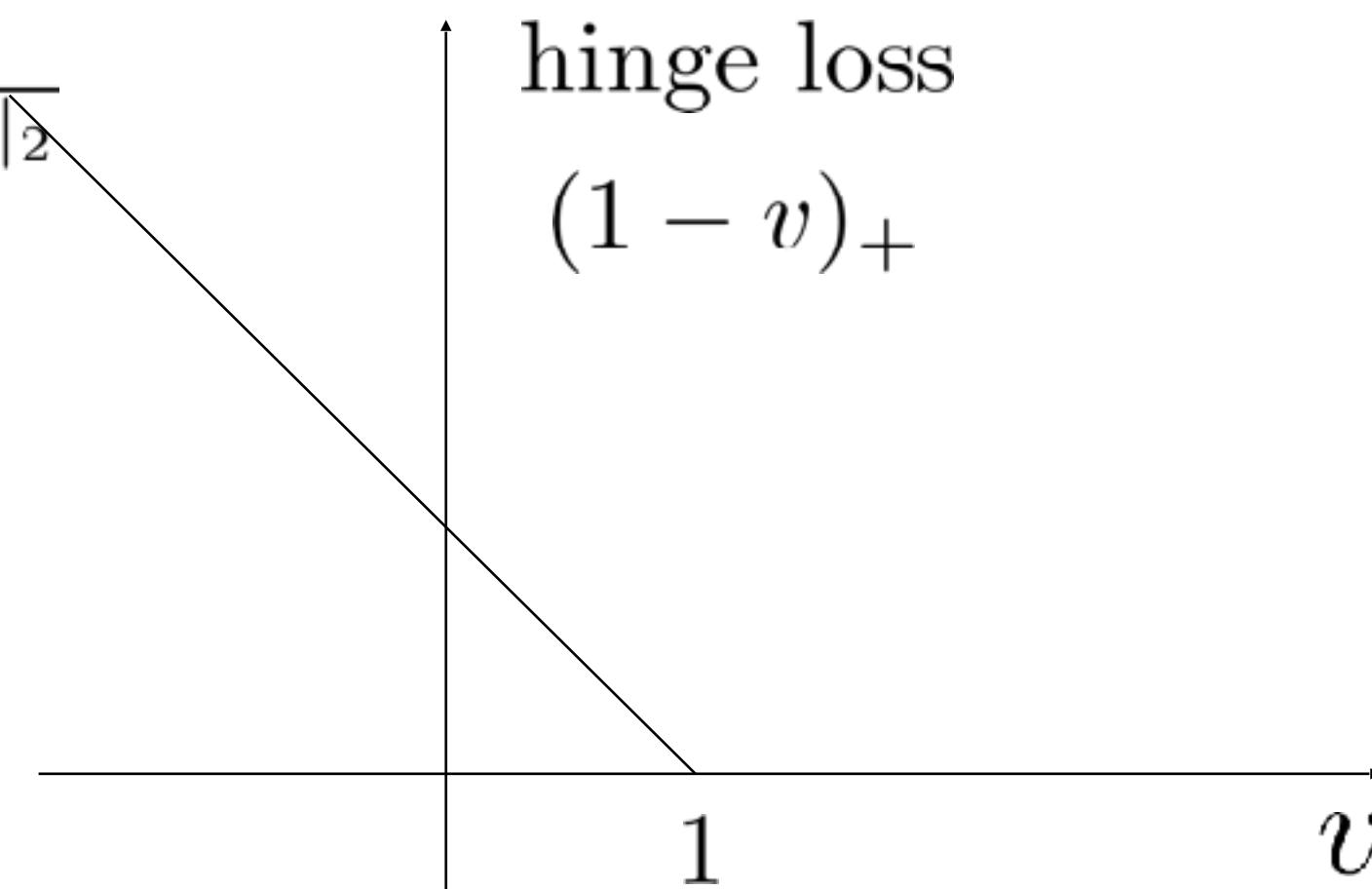


Find:  $\arg \min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2 + C \times \sum_{i=1}^n (1 - y_i \times (\mathbf{w}^T \mathbf{x}_i + b))_+$

# SVM with non-separable data



$$M = \frac{2}{\|\mathbf{w}\|_2}$$



$$\varepsilon_i \geq 0, \forall i$$

Find:  $\arg \min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|^2 + C \times \sum_{i=1}^n \varepsilon_i$

subject to:  $y_i \times (\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \varepsilon_i$

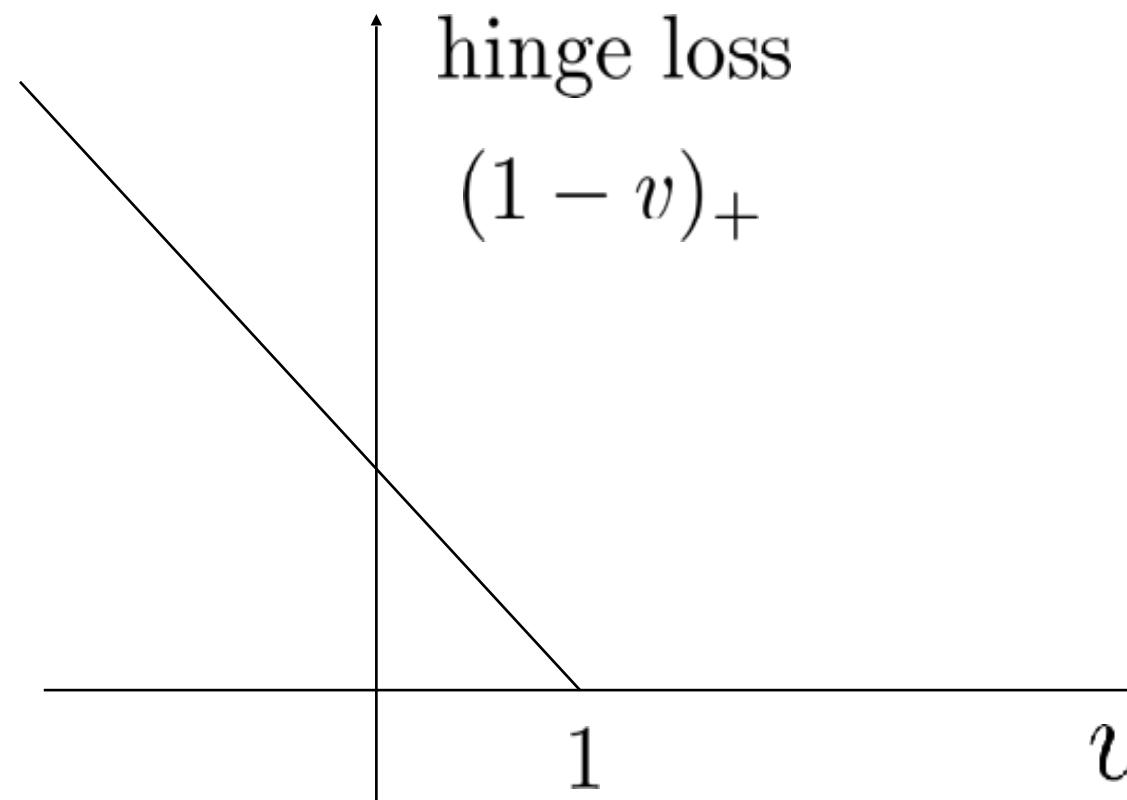
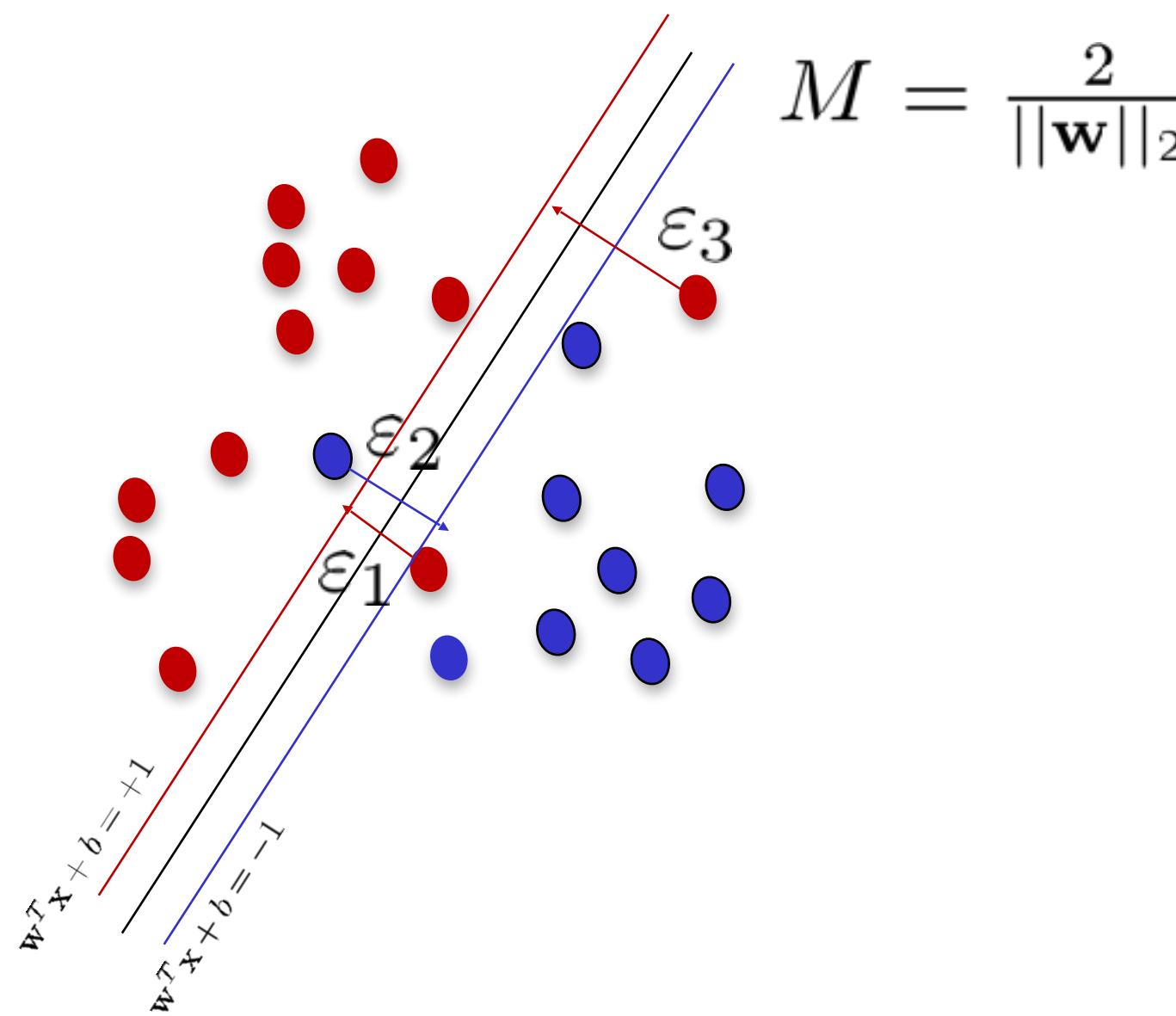
$$\xi_i = \max(1 - y_i(\mathbf{w}^T \mathbf{x}_i + b), 0)$$

Find:  $\arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 + C \times \sum_{i=1}^n \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))$

Minimize  $\mathcal{L}(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n (1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))_+$

# SVM: non-separable

---

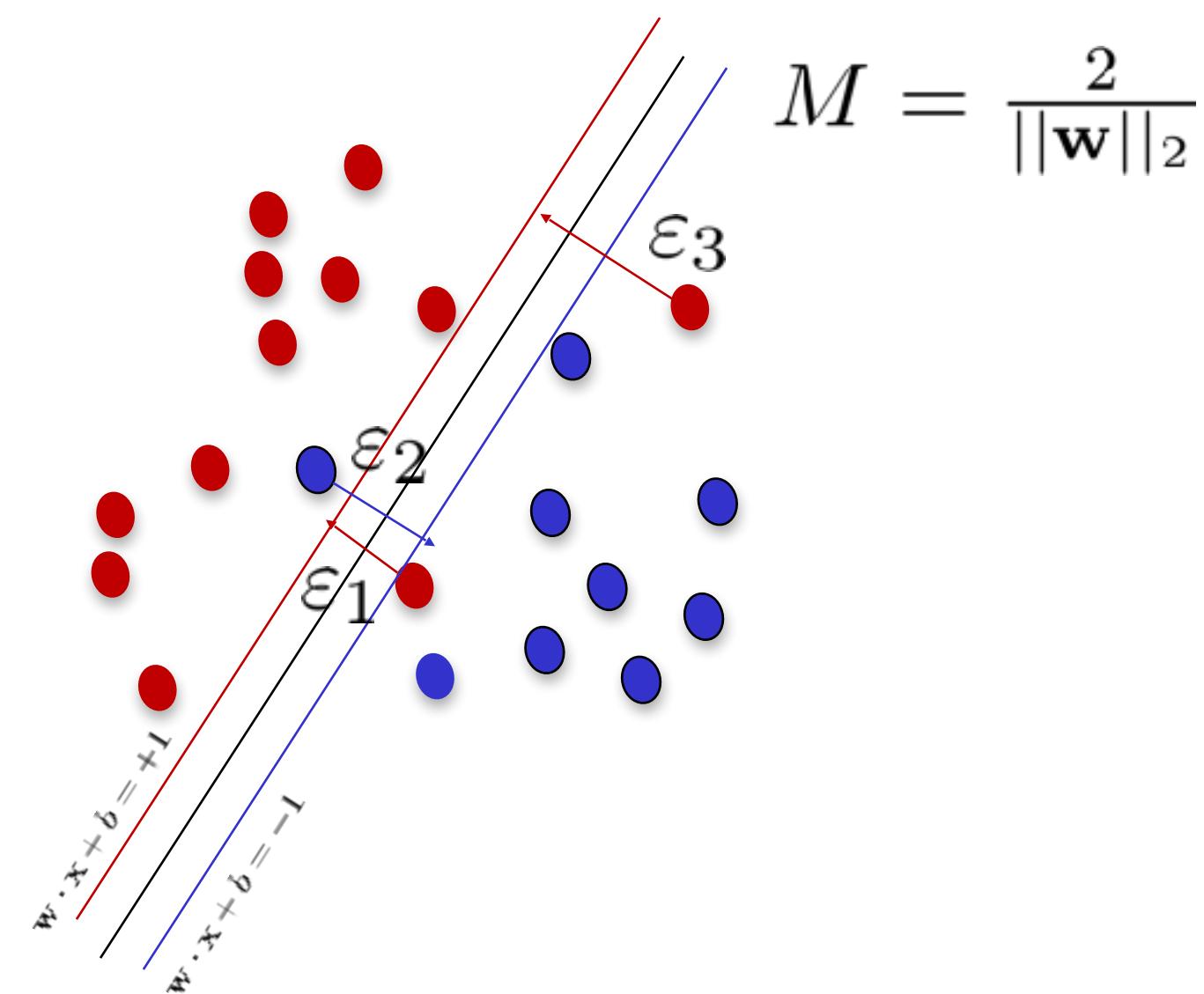


$$\text{Minimize } \mathcal{L}(\mathbf{w}, b) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n (1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))_+$$

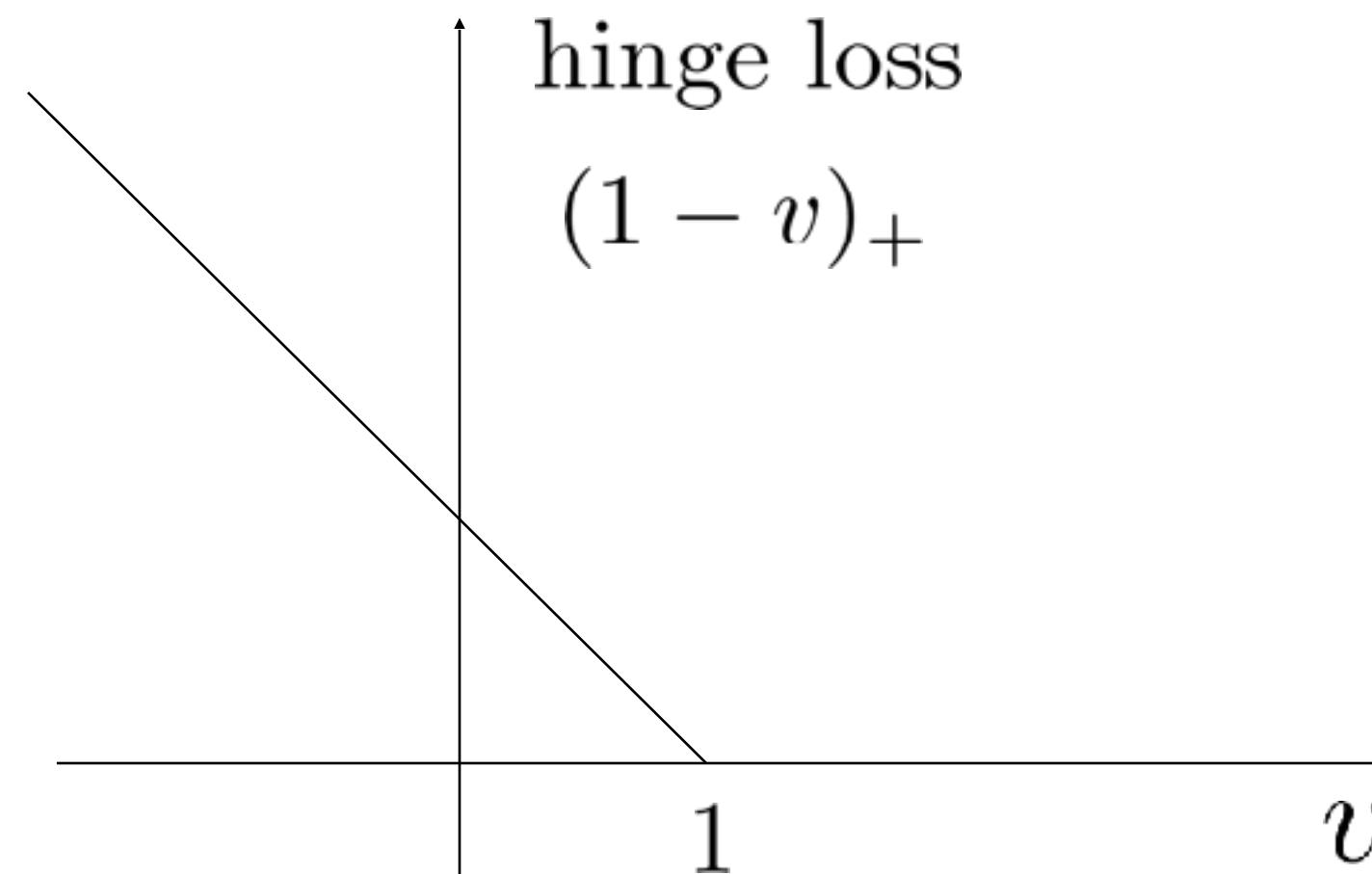
$$\frac{\partial \mathcal{L}(\mathbf{w}, b)}{\partial \mathbf{w}} = \mathbf{w} + C \sum_{i=1}^n \begin{cases} 0 & \text{if } y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \\ -y_i \mathbf{x}_i & \text{otherwise} \end{cases}$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, b)}{\partial b} = C \sum_{i=1}^n \begin{cases} 0 & \text{if } y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \\ -y_i & \text{otherwise} \end{cases}$$

# Convex?



$$M = \frac{2}{\|\mathbf{w}\|_2}$$

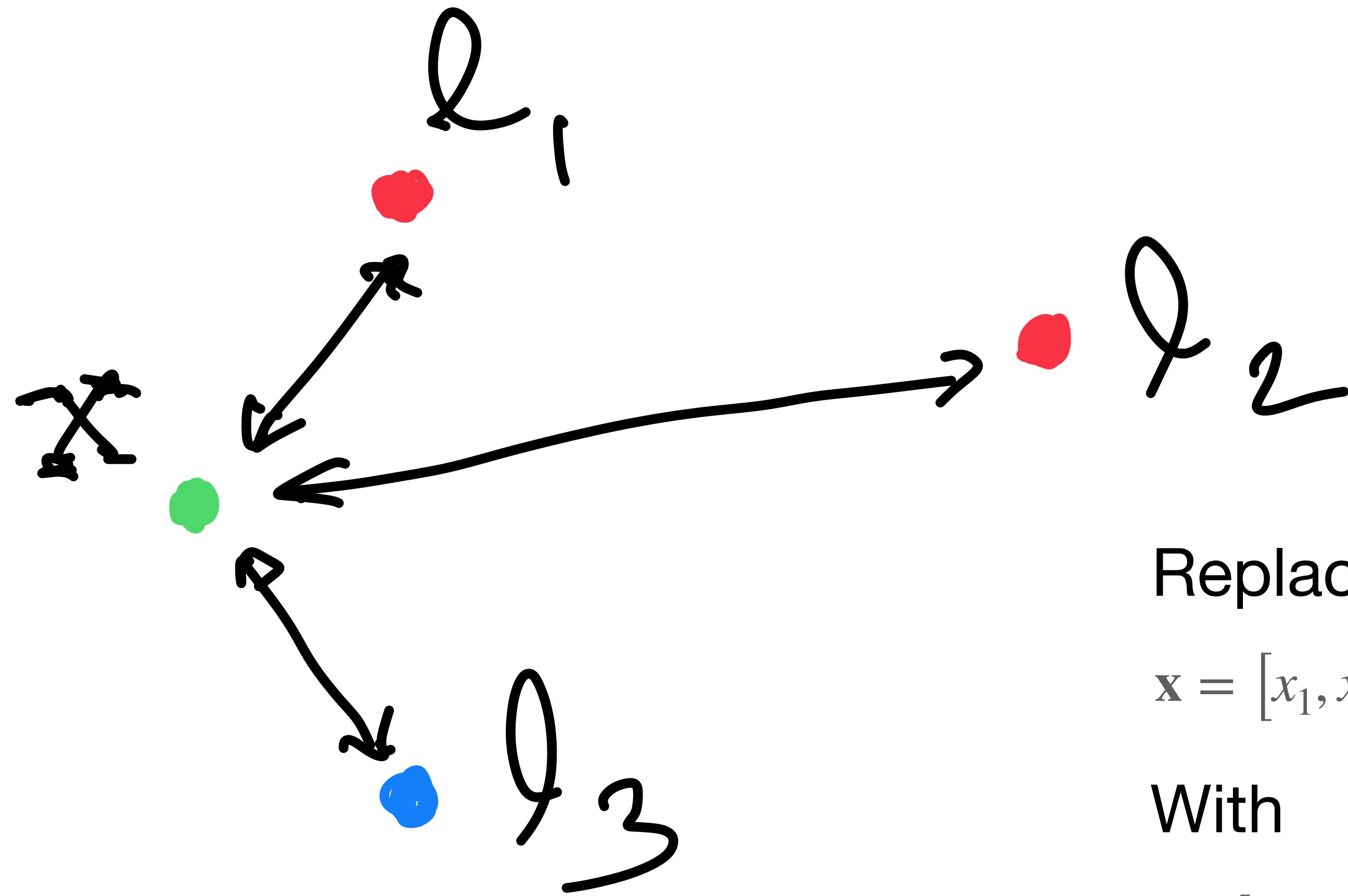


Find:  $\arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 + C \times \sum_{i=1}^n \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))$

- 😊 A. Convex  
B. Concave  
C. No-convex  
D. It depends

The summation of convex functions is also convex.

# The kernel trick



Replace

$$\mathbf{x} = [x_1, x_2, \dots x_m]^T$$

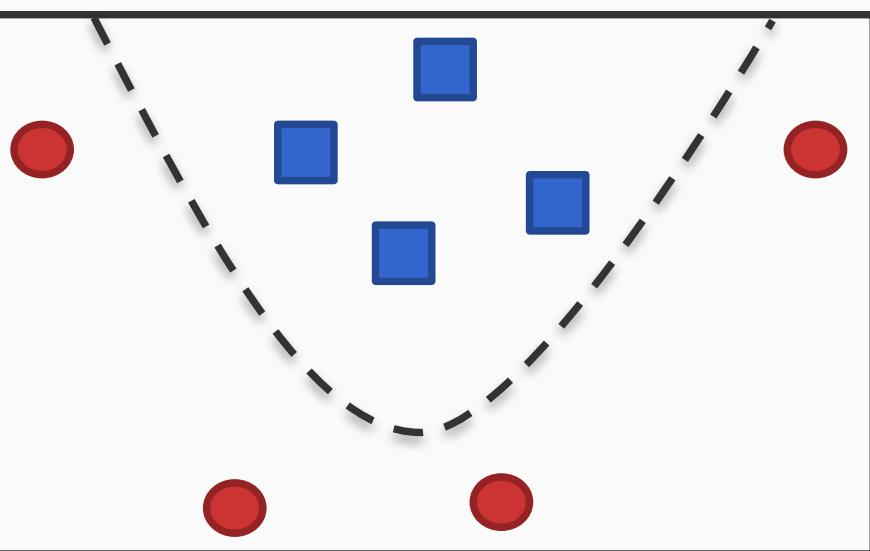
With

$$\mathbf{f} = [\text{similarity}(x, l_1), \text{similarity}(x, l_2), \dots \text{similarity}(x, l_n)]$$

# One way to learn non-linear models

Explicitly introduce non-linearity into the feature space

If the true separator is quadratic



## Dot products in high dimensional spaces

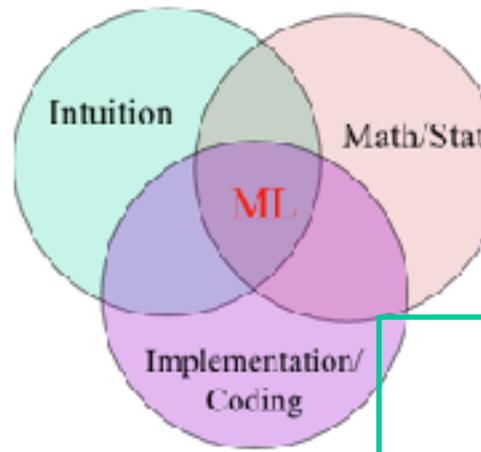
Let us define a dot product in the high dimensional space

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

Inner product: output is a scalar!

This is the kernel trick; we are doing the math in 1-D space

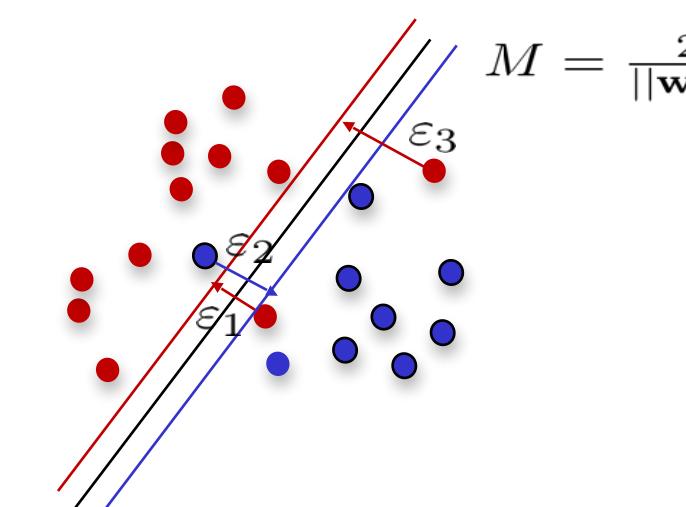
- not the input space
- not in the higher dimensional space the kernel transforms into



# Recap: Support Vector Machine

**Intuition:** It explicitly introduces a “regularization” (margin) into the objective function to combine with a classification error (restricted using a hinge loss) term.

- It achieves unprecedented robustness when training a linear classifier due to the use of margin term in training.
- The learned model is based on a balance between classification error and margin. The balancing term  $C$  is typically attained using cross-validation.
- Kernel based SVM makes non-separable samples feasible to classify by projecting the data onto higher dimensional spaces.
- The features defined under kernels don't need to be computed explicitly.
- The learned weights  $\mathbf{w}$  is carried in the weights for the samples and those samples with non-zero weights are called support vectors.



# SVM yeah!

Robust in that they are determined only by support vectors... noise matters less

Explicit bias-variance tradeoff; Softer margins allow for more generalization  
(regularization)

Small number of hyper parameters

Popular because they are often good enough

Cheap computationally if setup well, can scale to internet sized data when done right

Kernel trick means you get the benefits of high D with the computational complexity of low D (in variables, still have high D in sample size)

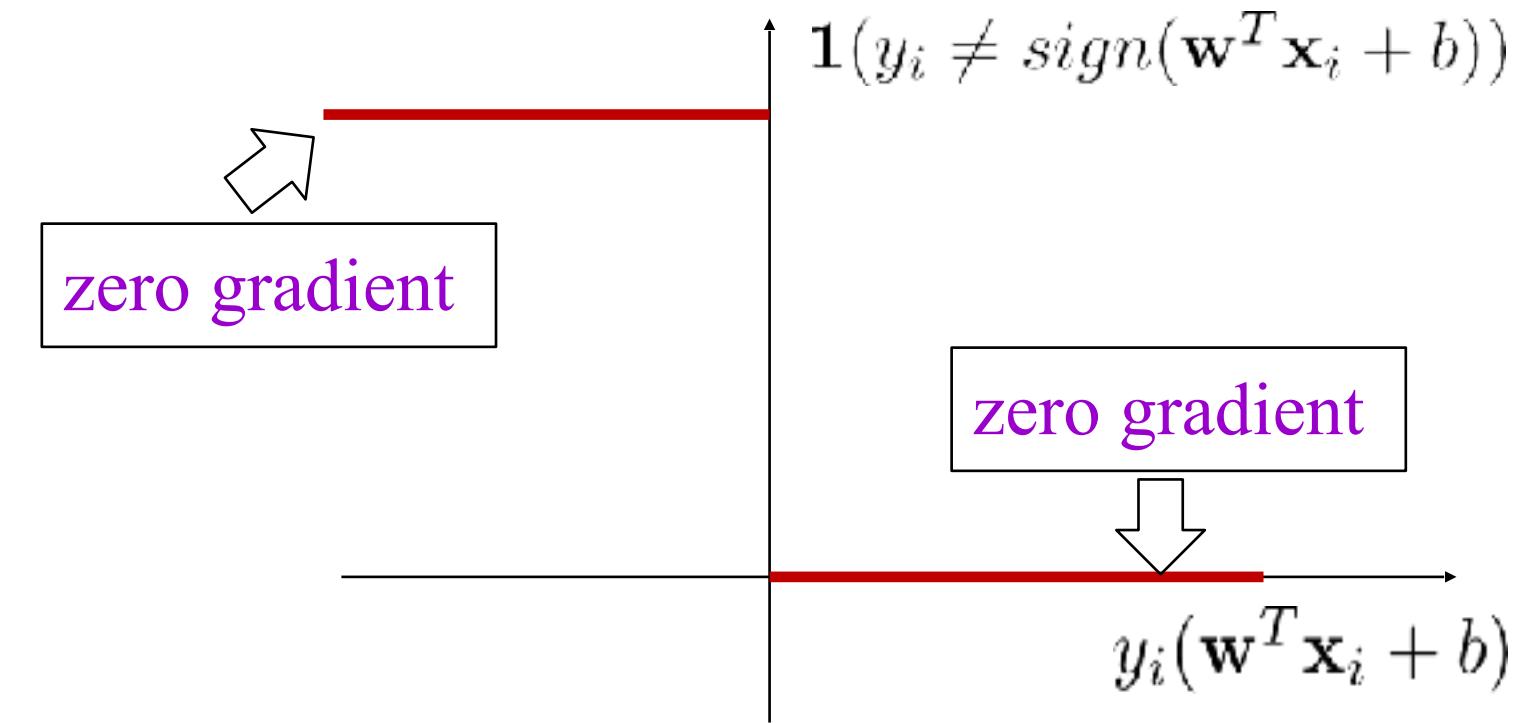
## Standard loss (error) function

Main motivation

Hard->Half-hard->Soft Error

Standard 0/1 loss (gradient 0 nearly everywhere,  
**no gradient feedback**):

Training: Minimize  $\mathcal{L}(\mathbf{w}, b) = \sum_i \mathbf{1}(y_i \neq \text{sign}(\mathbf{w}^T \mathbf{x}_i + b))$



It is the most **direct** loss, but is also  
the **hardest** to minimize.

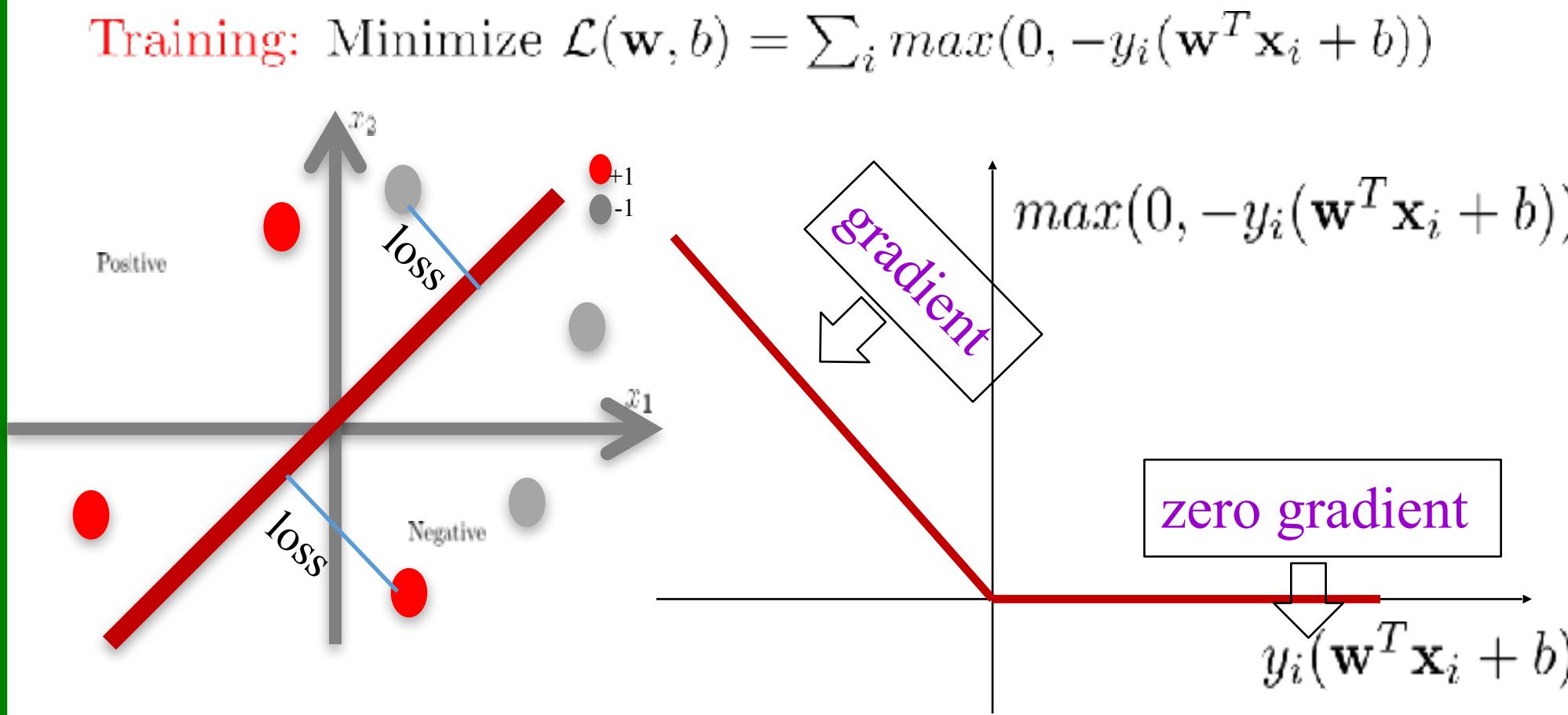
Zero gradient everywhere!

# Half-hard loss (error) function

Main motivation

Hard->Half-hard->Soft Error

Loss implicitly used in the perceptron algorithm: with **gradient feedback** when the target (ground-truth label) and the output (classification) are different).



Zero loss for correct classification (**no gradient**).

A loss based on the **distance** to the decision boundary for **misclassification** (**with gradient**).

Used in the **perceptron** training.

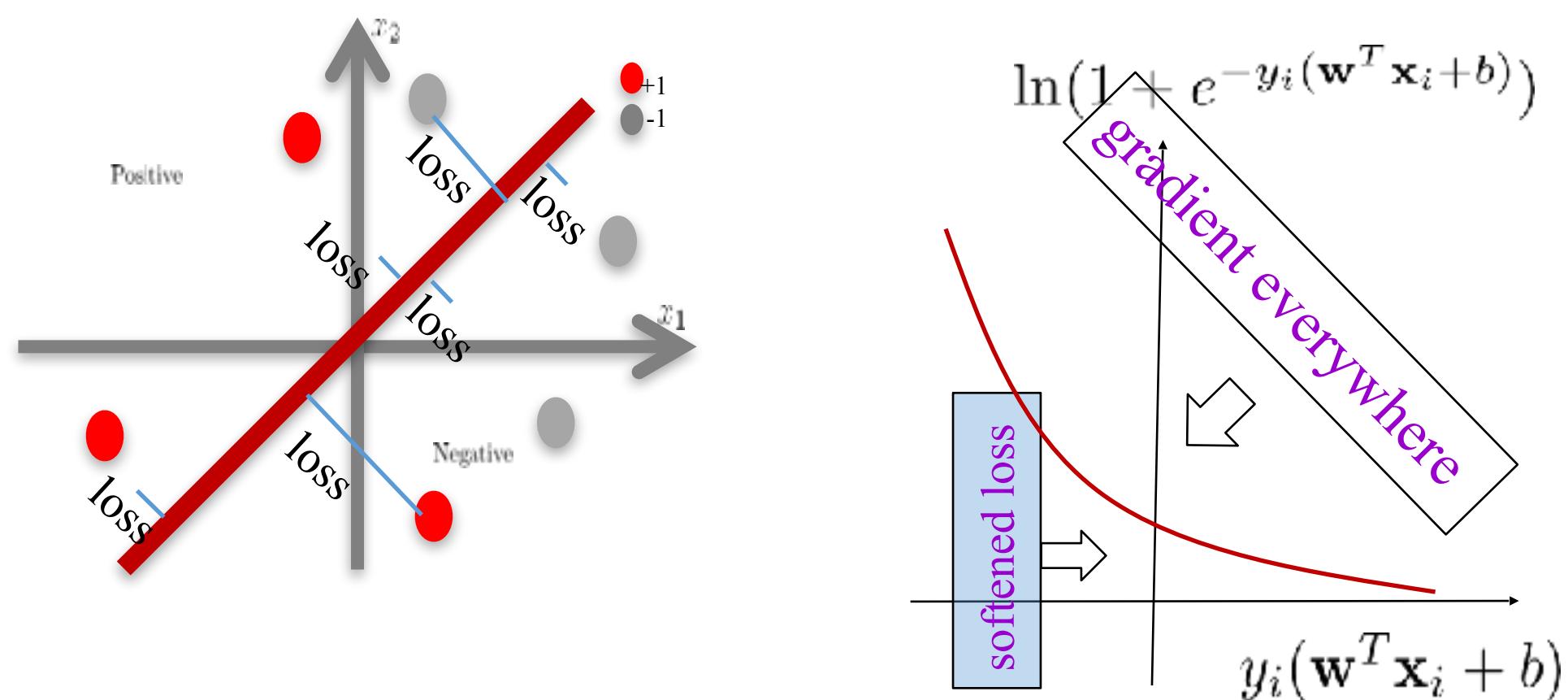
# Soft loss (error) function

Main motivation

Hard->Half-hard->**Soft** Error

Loss used in logistic regression.

**Training:** minimize  $\mathcal{L}(\mathbf{w}, b) = \sum_{i=1}^n \ln(1 + e^{-y_i(\mathbf{w}^T \mathbf{x}_i + b)})$



Every data point receives a loss (gradient everywhere).

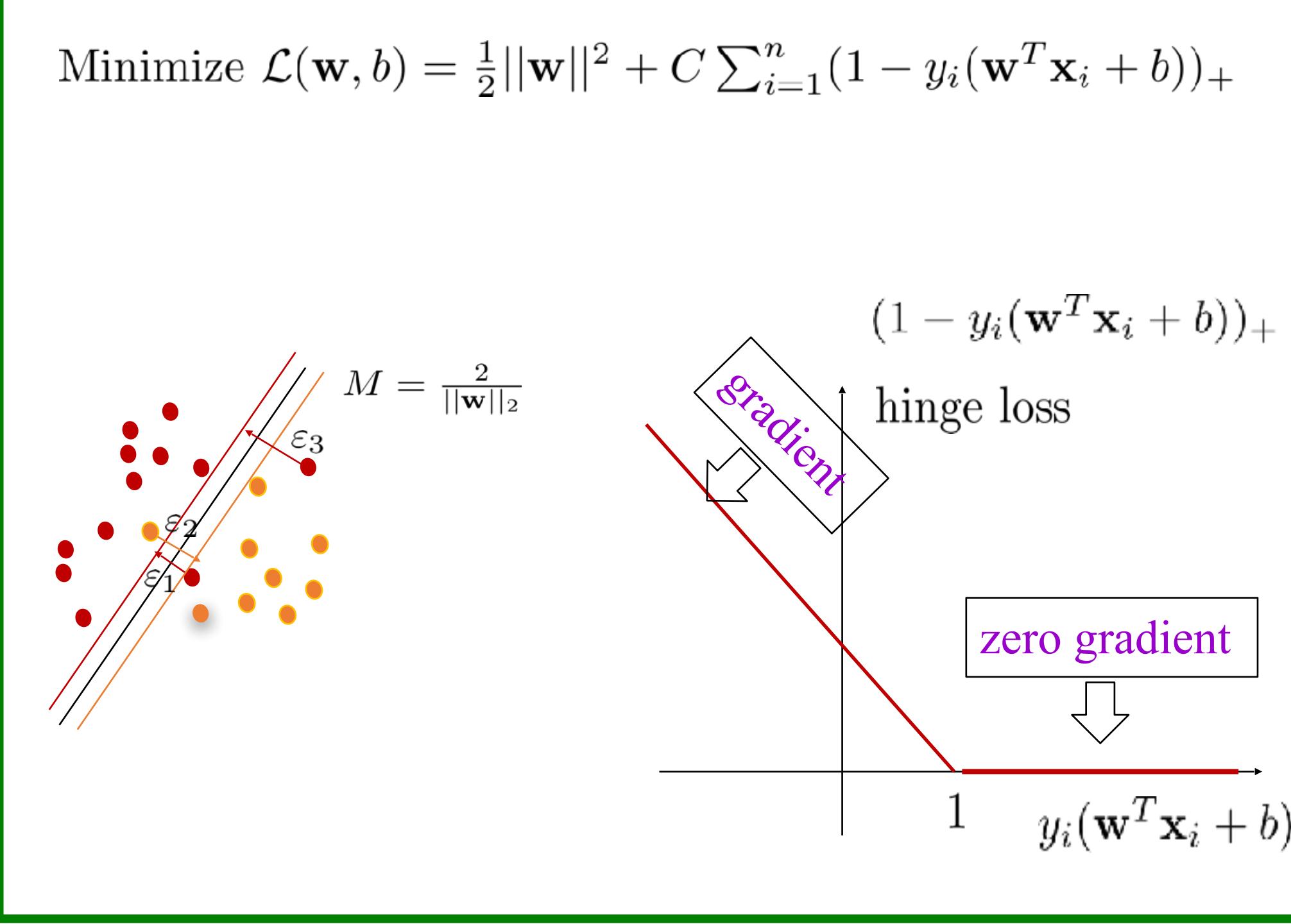
A loss based on the **distance to the decision boundary** for wrong classification (has a gradient).

Used in **logistic regression** classifier.

# Loss in SVM

Main motivation

Hard->Half-hard->Soft Error



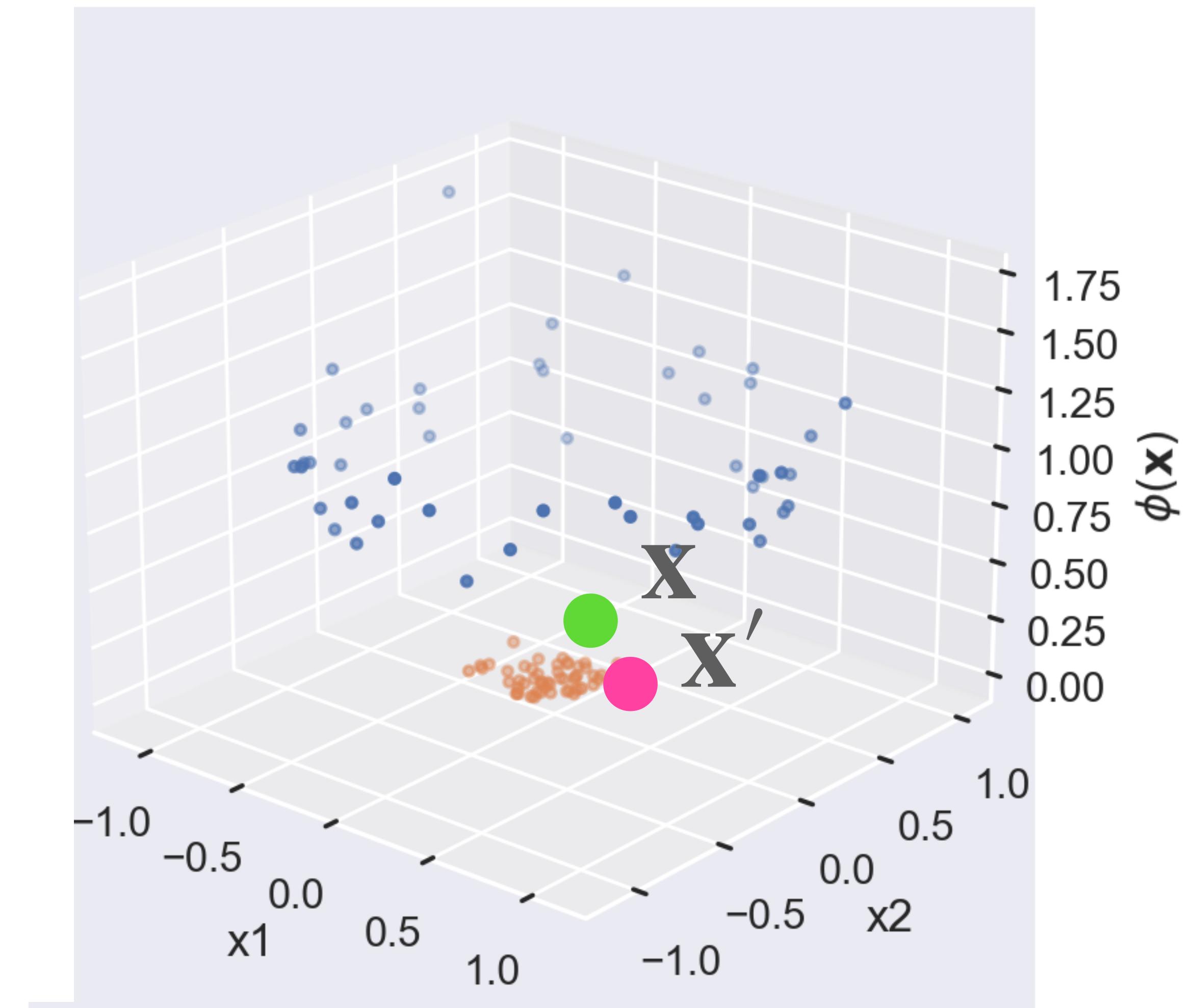
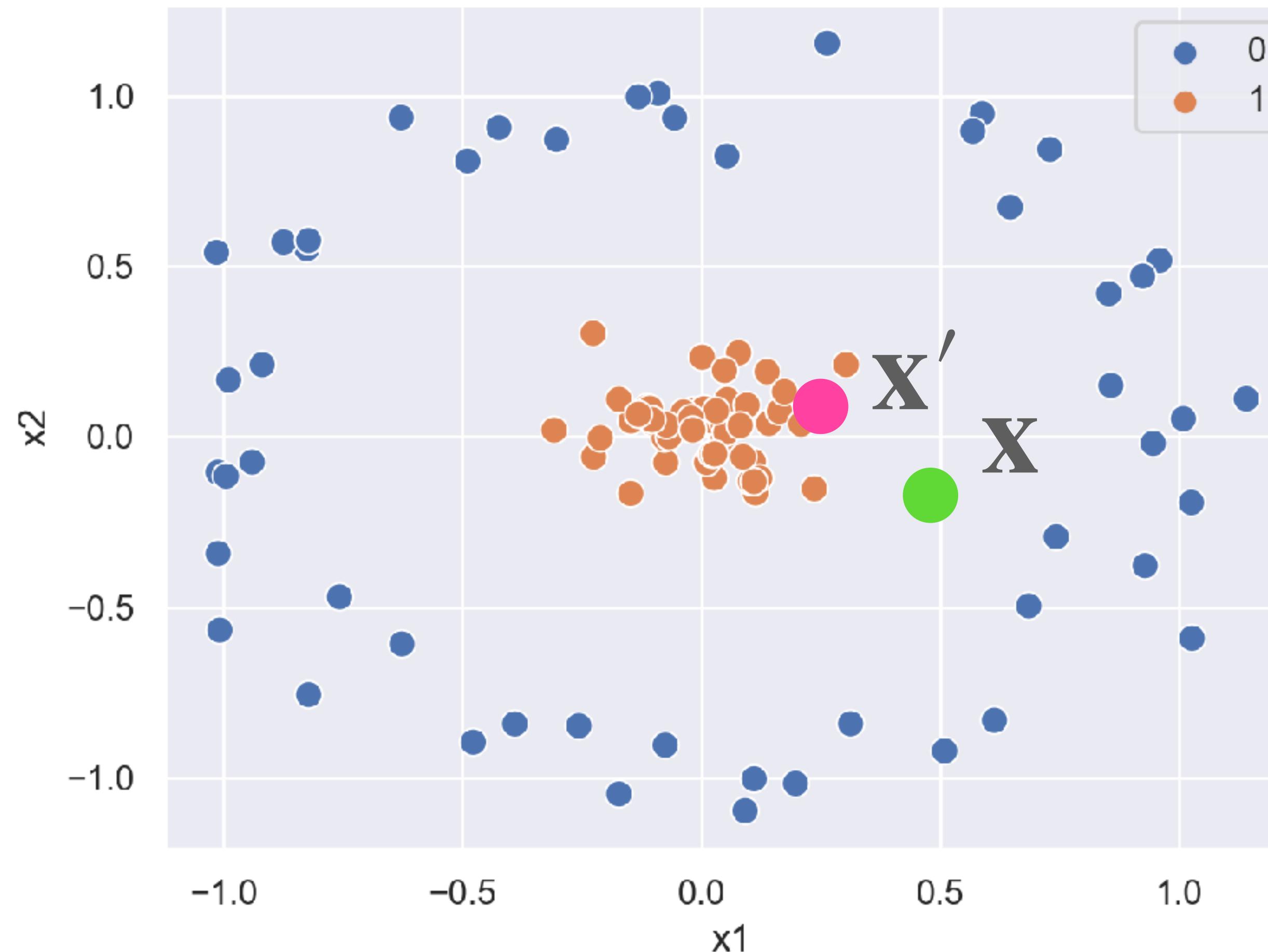
Zero loss for correct classification beyond the margin (**no gradient**).

A loss based on the **distance** to the decision boundary for **misclassification** or **within the margin (with gradient)**.

$k(\mathbf{x}, \mathbf{x}') = \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$  means you never have to calculate

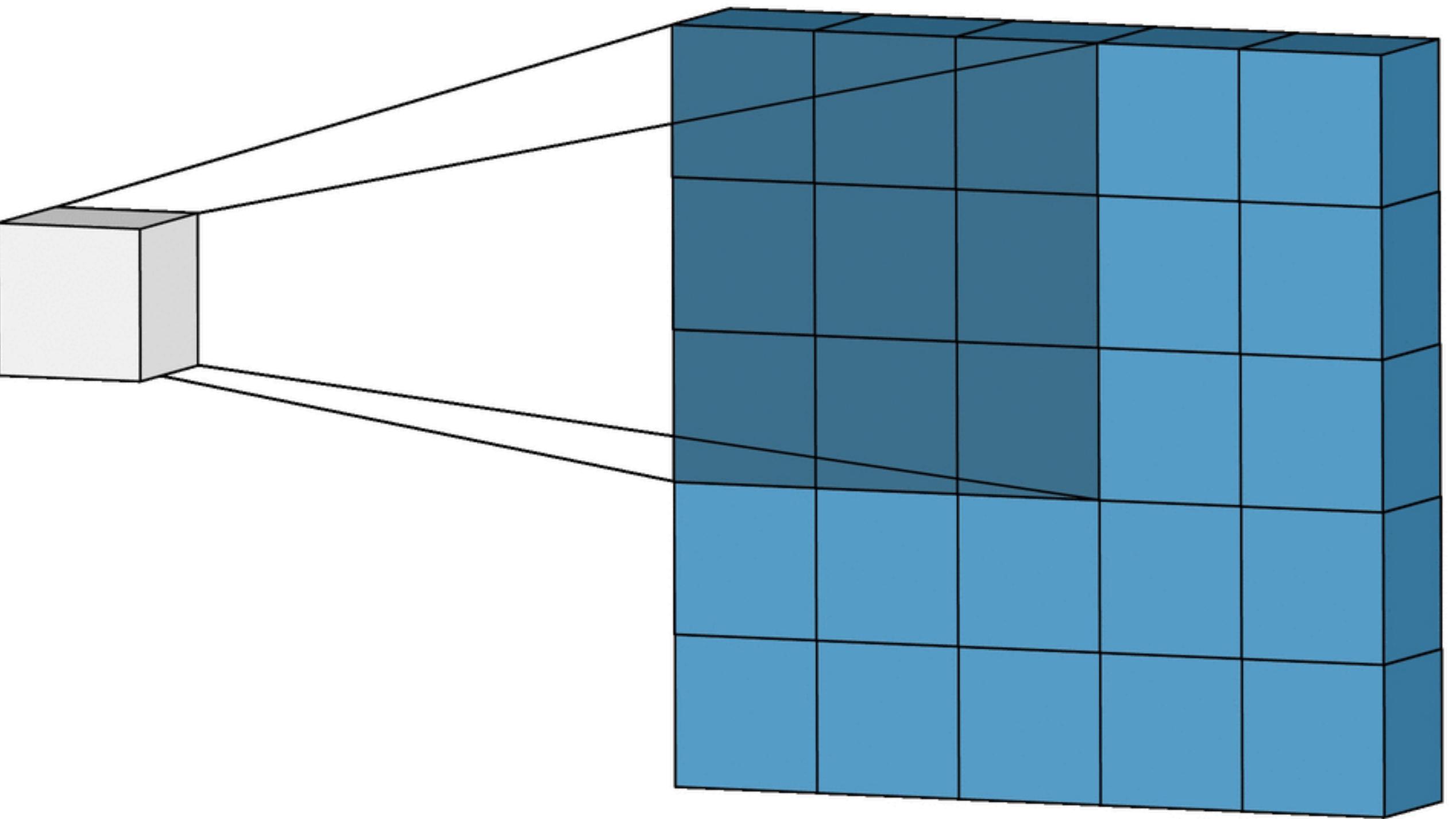
the decision boundary in either  $\mathcal{X}$  or  $\mathcal{V}$ ...

its in terms of a scalar dot product of vectors in  $\mathcal{V}$ !



# Common kinds of kernels

- Moving average window
- Polynomial of order  $d$
- Radial basis / Gaussian
- Sigmoid / tanh



## Dot products in high dimensional spaces

Let us define a dot product in the high dimensional space

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

So prediction with this *high dimensional lifting map* is

$$\text{sgn}(\mathbf{w}^T \phi(\mathbf{x})) = \text{sgn} \left( \sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) \right)$$

because  $\mathbf{w}^T \phi(\mathbf{x}) = \sum_i \alpha_i y_i \phi(\mathbf{x}_i)^T \phi(\mathbf{x})$

26

Inner product: output is a scalar!

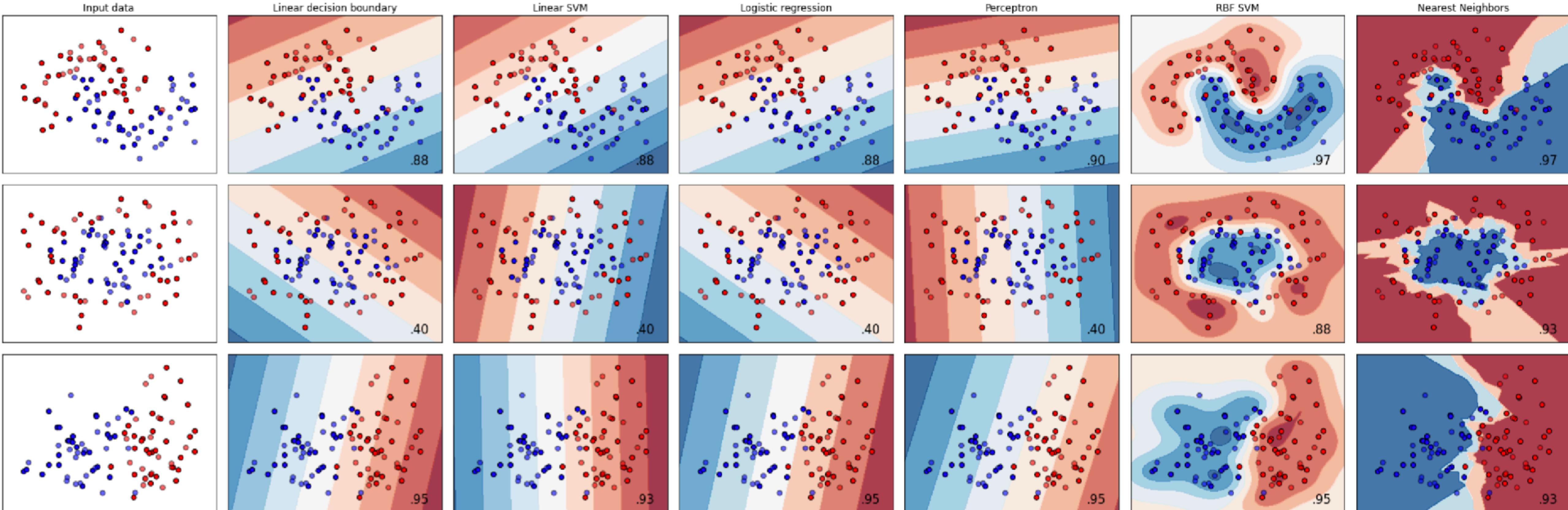
This is the kernel trick; we are doing the math in 1-D space

- not the input space

- not in the higher dimensional space the kernel transforms into

**The trick:  
Replace a high dimensional projection with pairwise  
similarities between data samples.**

**Still roughly  $n^2$  but that's better than the high-D case!**

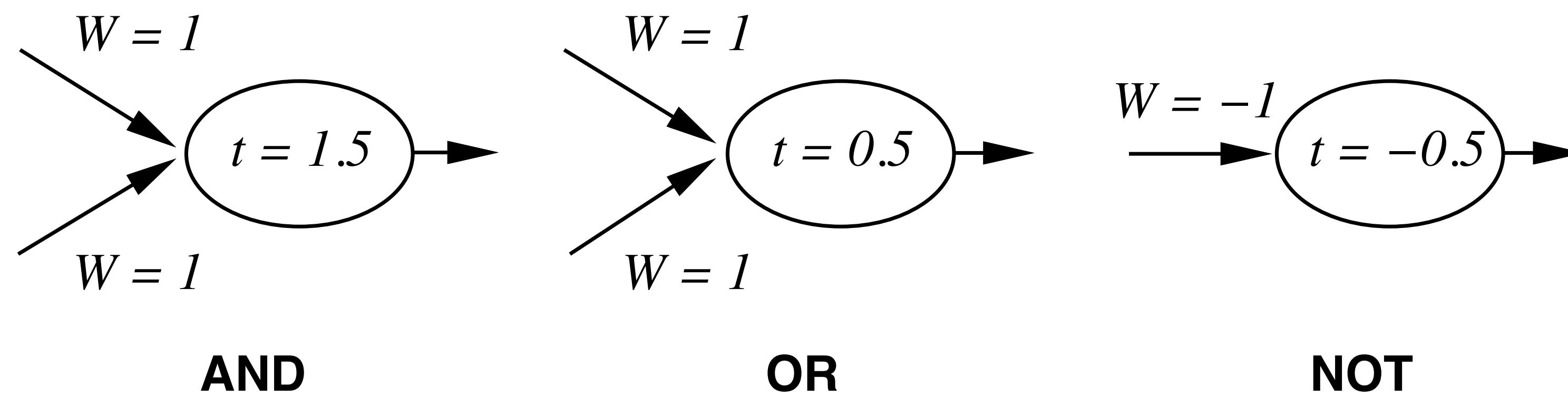


# Choosing between classifiers

- Want probabilities/interpretability? Logistic regression
- Want to fit non-linear decision boundaries? k-NN or kernel SVM
- Want something that is more likely to generalize well? SVM
- High dimensionality? Avoid k-NN, prefer SVM with low C.
- Answer is almost never perceptron, but perceptron -> MLP -> deep learning which is our new AI overlord.

NN

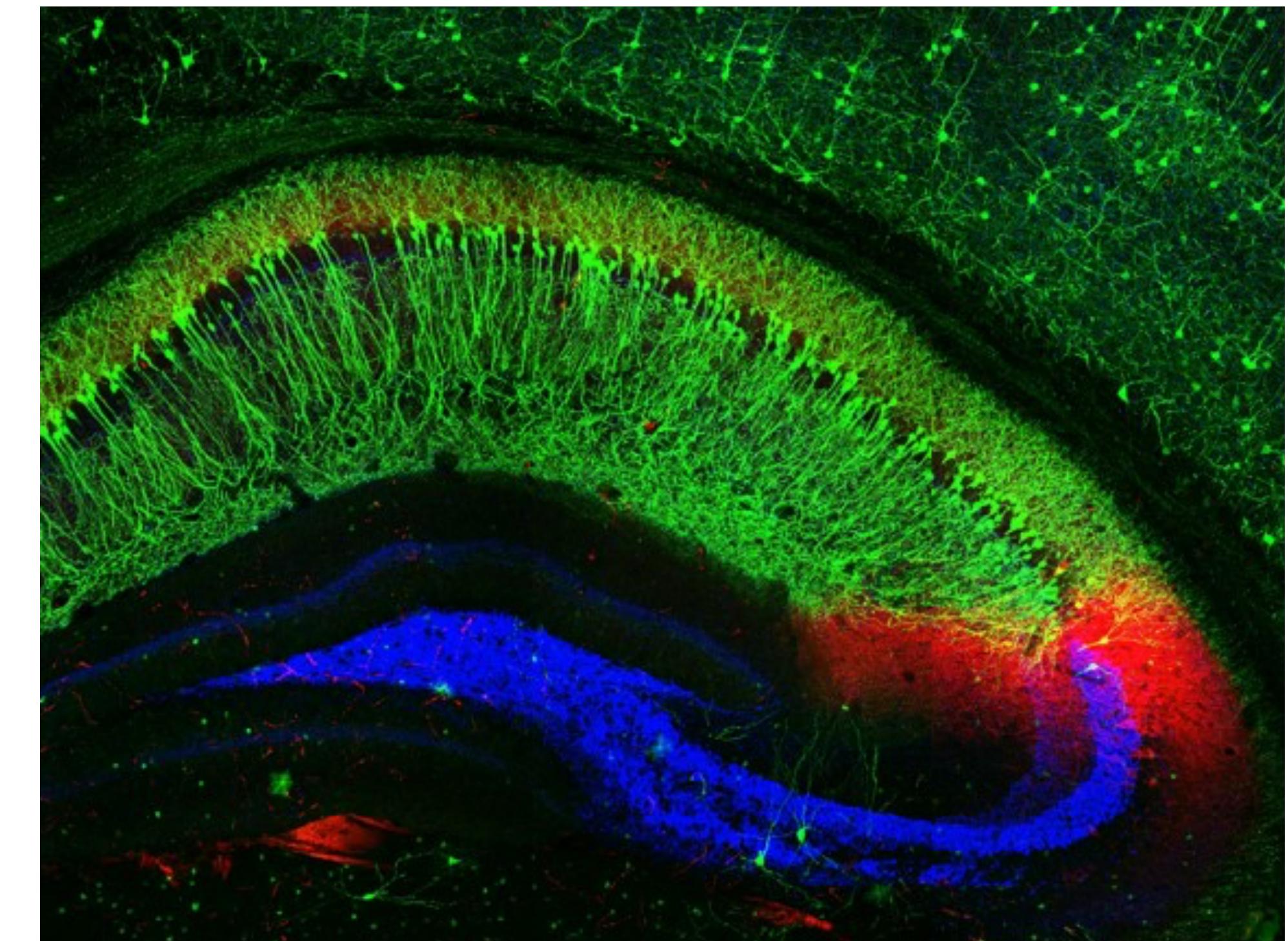
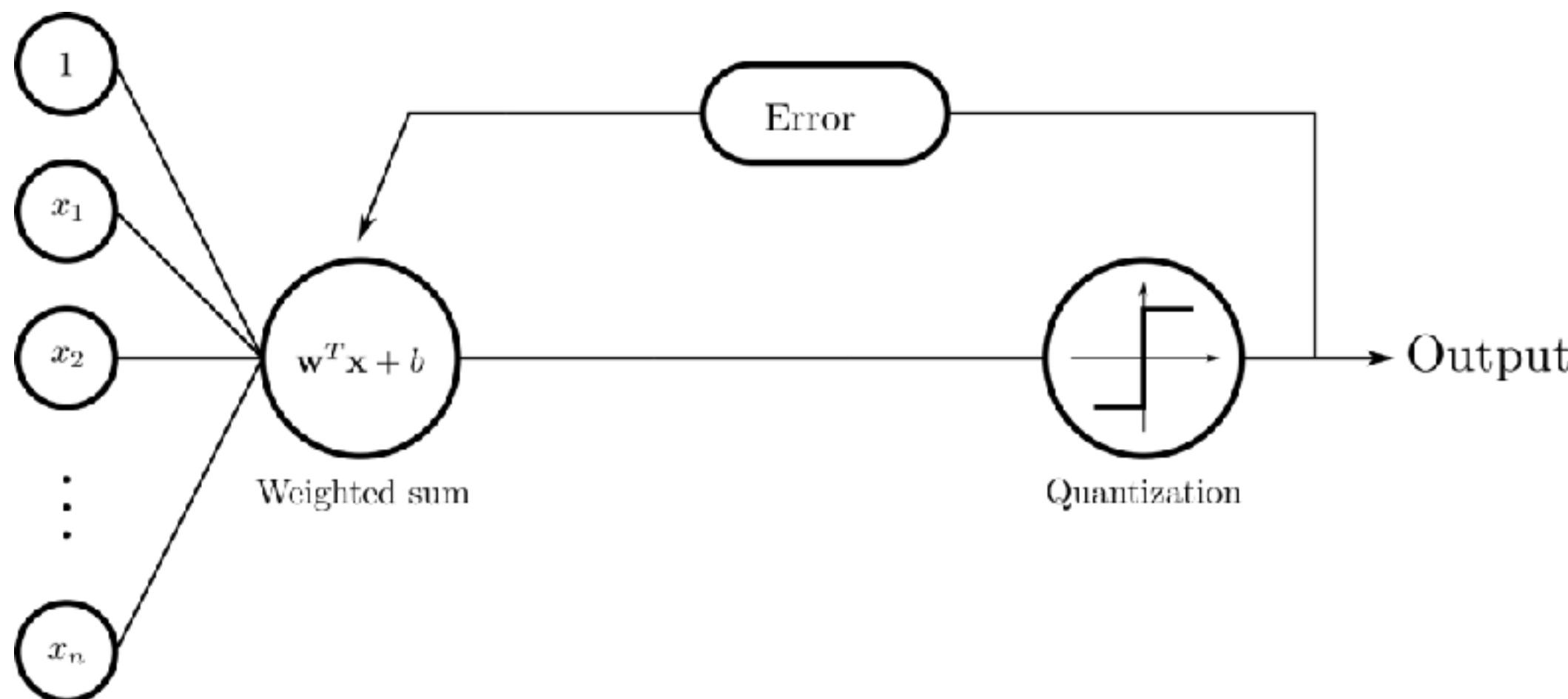
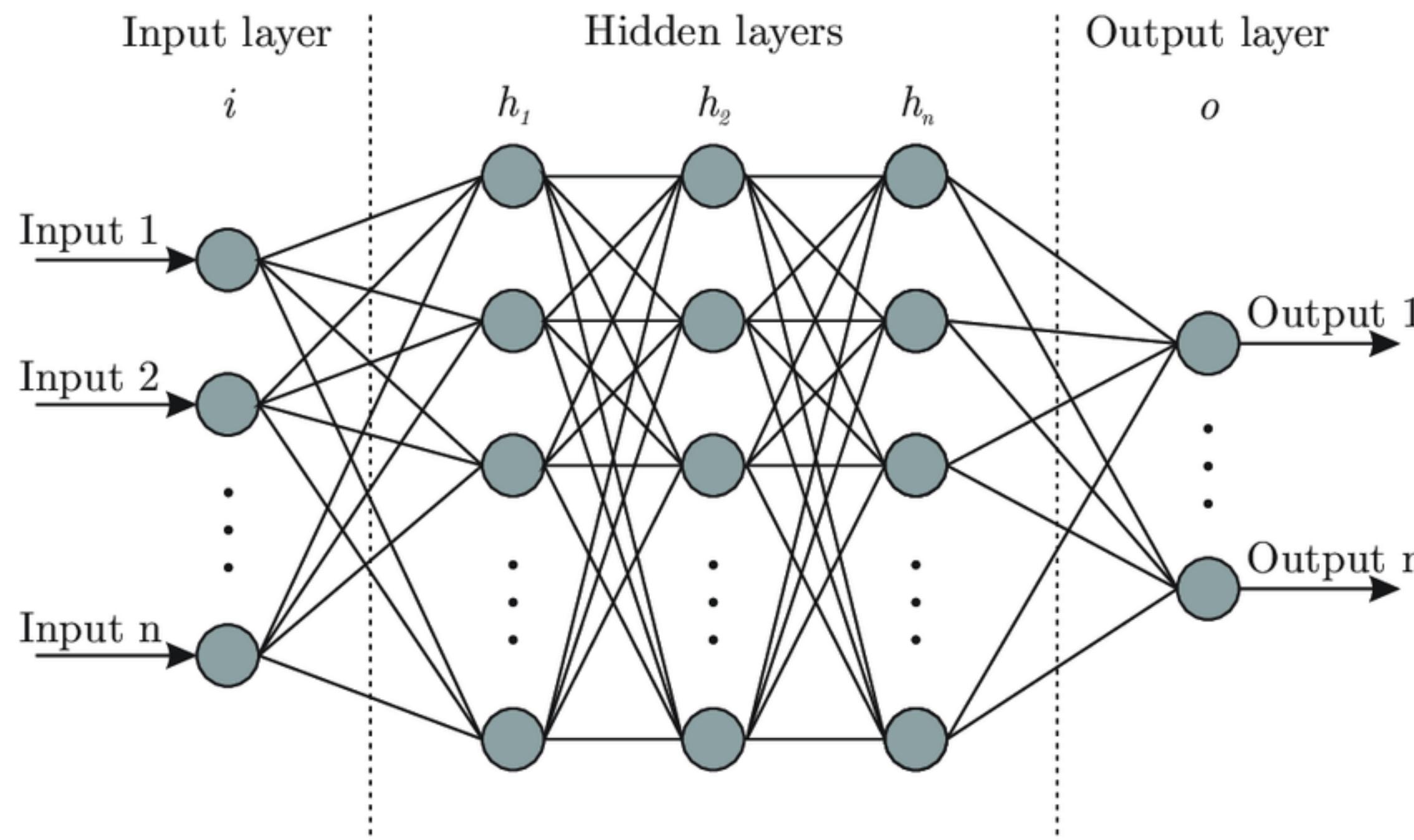
## Units as Logic Gates



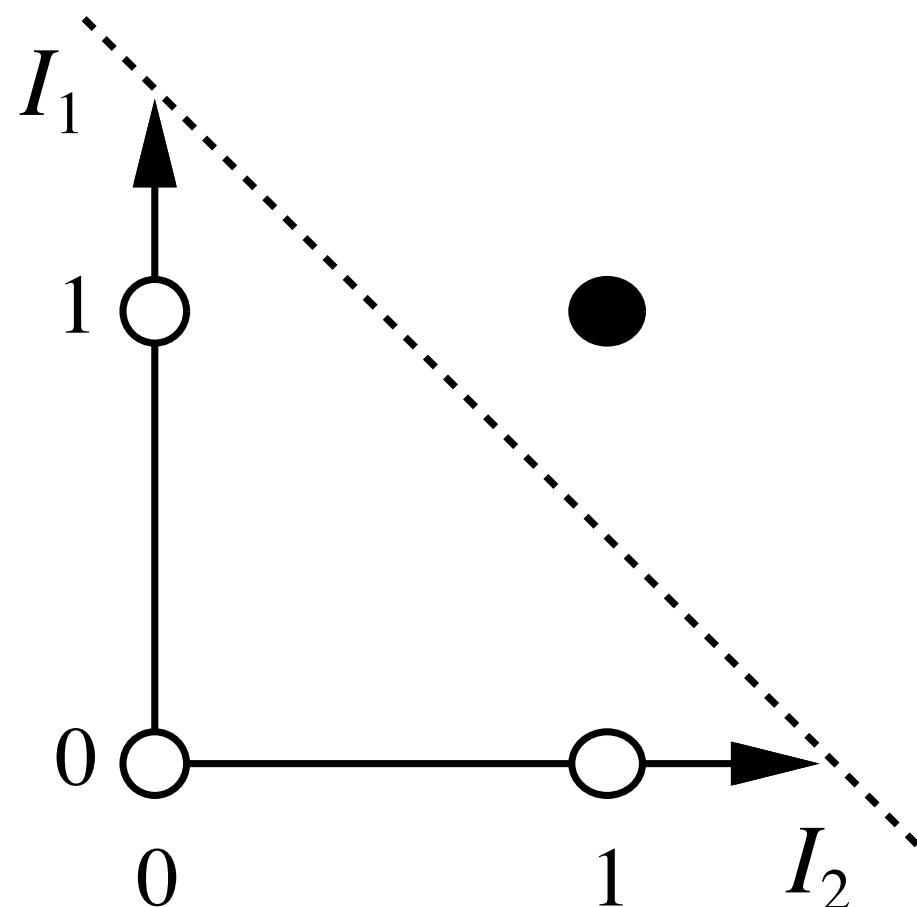
Activation function:  $\text{step}_t$

Since units can implement the  $\wedge$ ,  $\vee$ ,  $\neg$  boolean operators, neural nets are **Turing-complete**: they can implement *any* computable function.

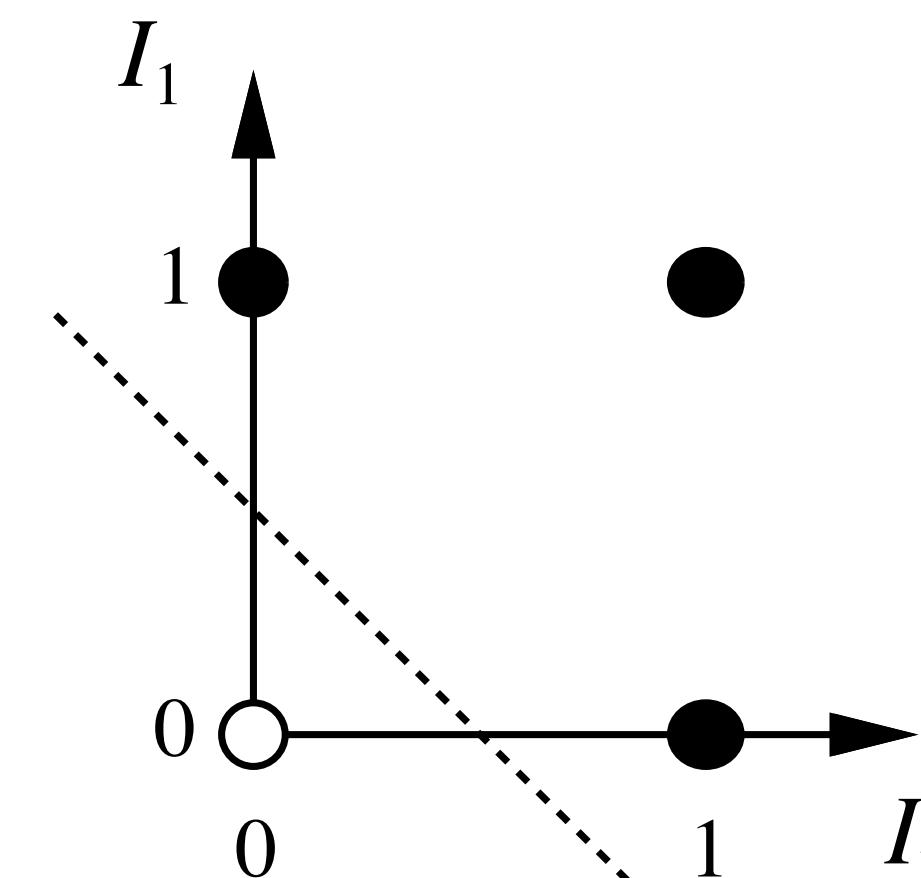
# Artificial Neural Networks



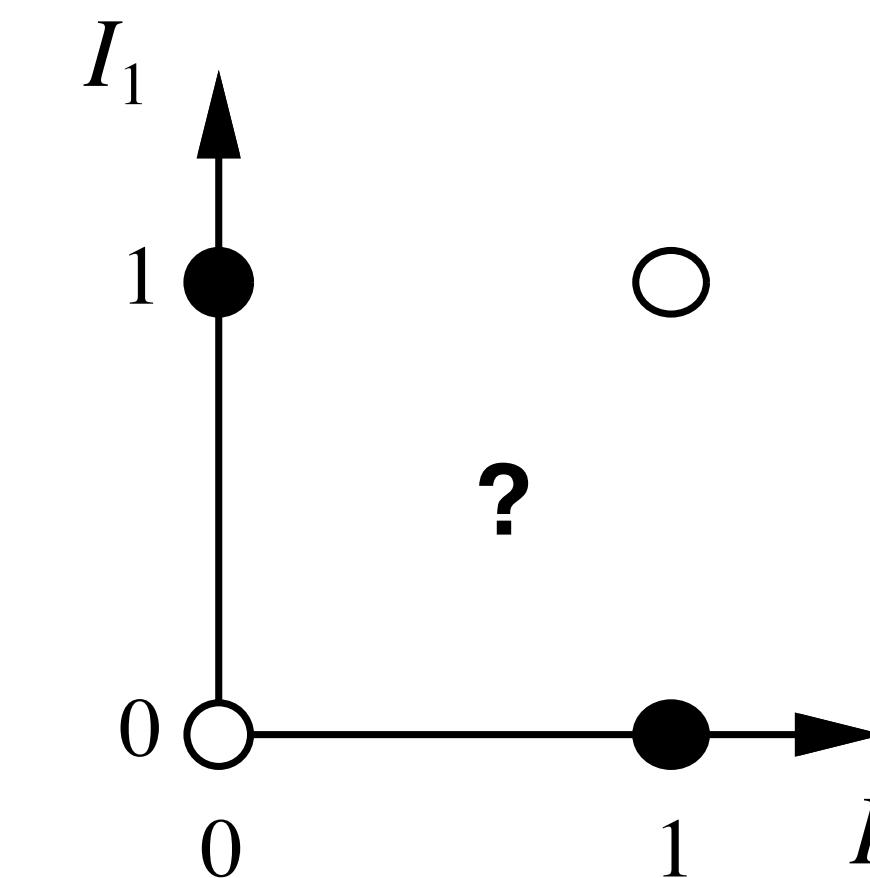
# Linearly Separable Functions on a 2-dimensional Space



**(a)**  $I_1$  **and**  $I_2$



**(b)**  $I_1$  **or**  $I_2$

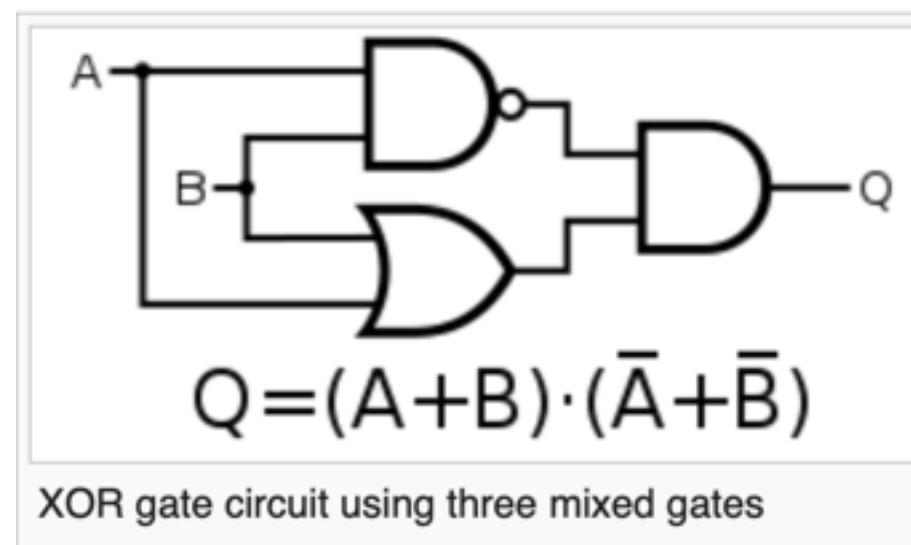
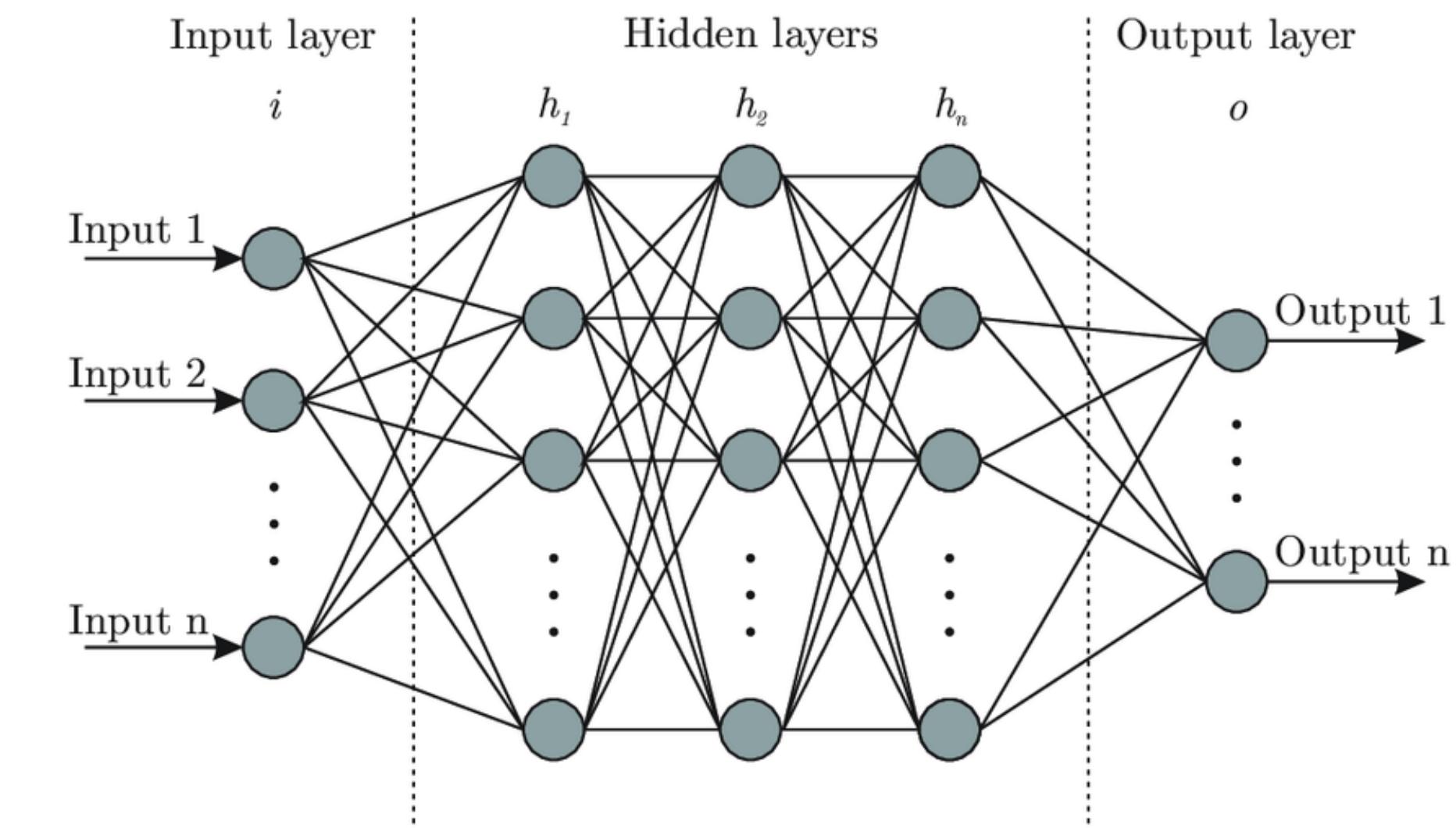


**(c)**  $I_1$  **xor**  $I_2$

A black dot corresponds to an output value of 1. An empty dot corresponds to an output value of 0.

# Multi-layer perceptrons and even deeper

- Solve the XOR, and in fact are general function approximations
- Need a special kind of gradient descent “back propagation”



# Perceptron Learning Algorithm

---

- Initialize the weights (however you choose)
  - $w_1x_1 + w_2x_2 + b$  (initialize  $w_1$ ,  $w_2$ , and  $b$ )
- Step 1: Choose a data point.
- Step 2: Compute the model output for the datapoint.
- Step 3: Compare model output to the target output.
  - If correct classification, go to Step 5!
  - If not, go to Step 4.
- Step 4: Update weights using perceptron learning rule. Start over on Step 1 with the first data point.

Or

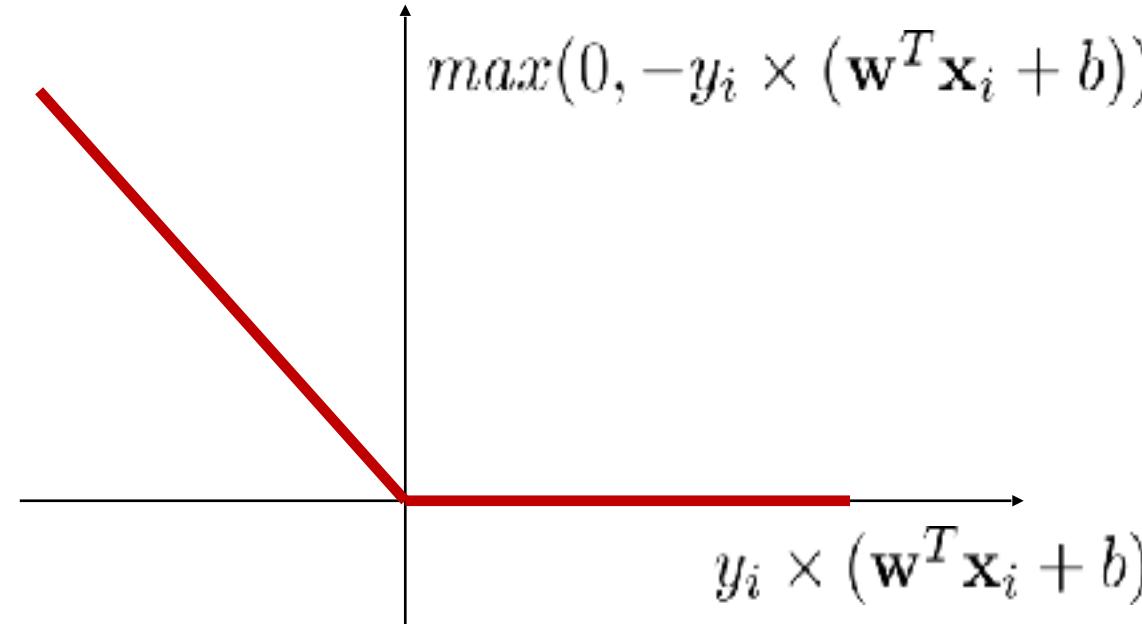
$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + (\text{target}_i - \text{output}_i)\mathbf{x}_i \\ b_{t+1} &= b_t + (\text{target}_i - \text{output}_i) \end{aligned}$$

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \lambda(\text{target}_i - \text{output}_i)\mathbf{x}_i \\ b_{t+1} &= b_t + \lambda(\text{target}_i - \text{output}_i) \end{aligned}$$

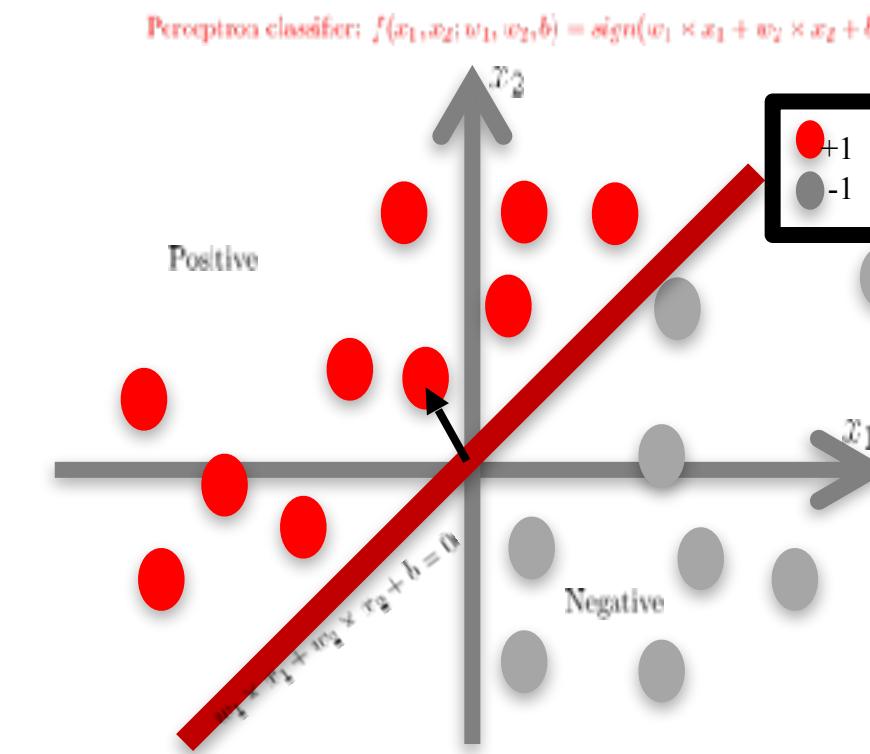
- Step 5: Go to the next data point. If you have gone through them all, you have found the solution!

Hinge loss: **gradient feedback** when the target (ground-truth label) and the output (classification) are different.

Training: Minimize  $\mathcal{L}(\mathbf{w}, b) = \sum_i \max(0, -y_i \times (\mathbf{w}^T \mathbf{x}_i + b))$



## Swapping loss functions



- Perceptron is a **linear classifier**.
- It replaces the 0/1 loss by a **relaxed** loss.
- It updates the model parameters  $(\mathbf{w}, b)$  based on a **single sample**, whereas standard gradient descent algorithm computes the gradient by taking **ALL the training samples** into account.

# Perceptron Learning

---

Perceptron:

Note that the learning process is **not strictly** gradient descent.

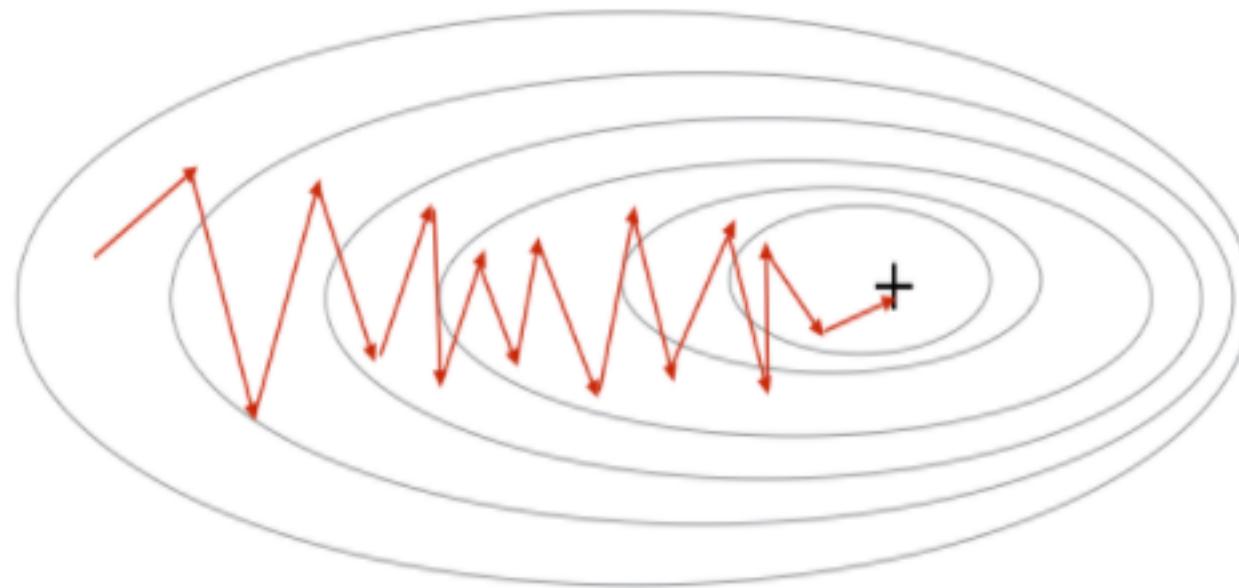
$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + (\text{target}_i - \text{output}_i) \mathbf{x}_i \\ b_{t+1} &= b_t + (\text{target}_i - \text{output}_i)\end{aligned}$$

It's a **stochastic** gradient descent algorithm!

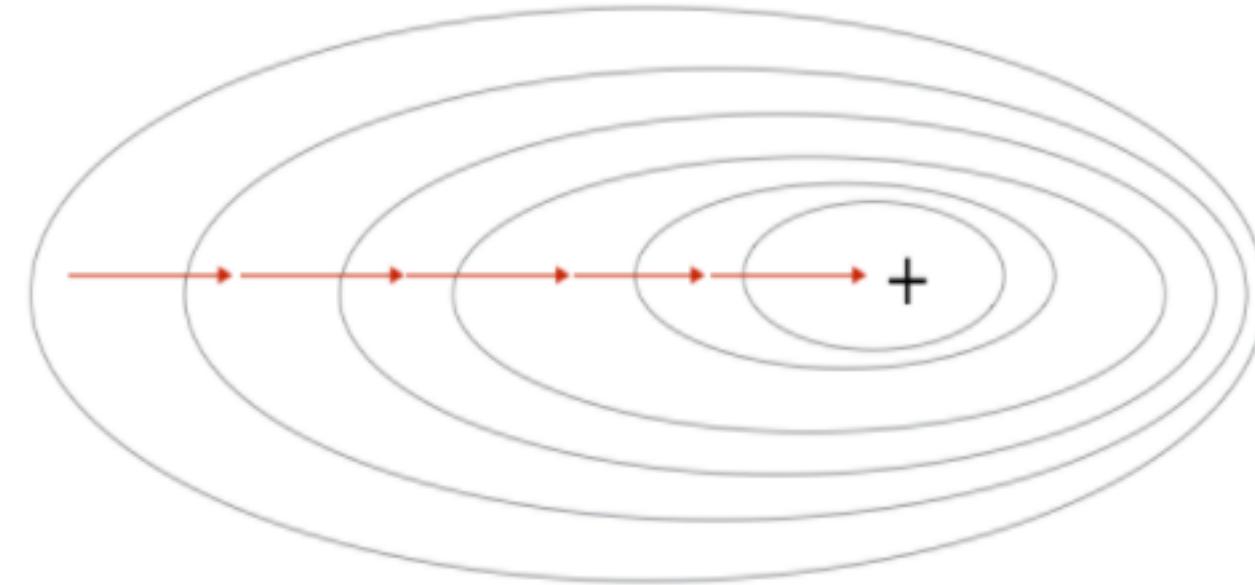
---

# Stochastic vs Batch Gradient Descent

Stochastic Gradient Descent

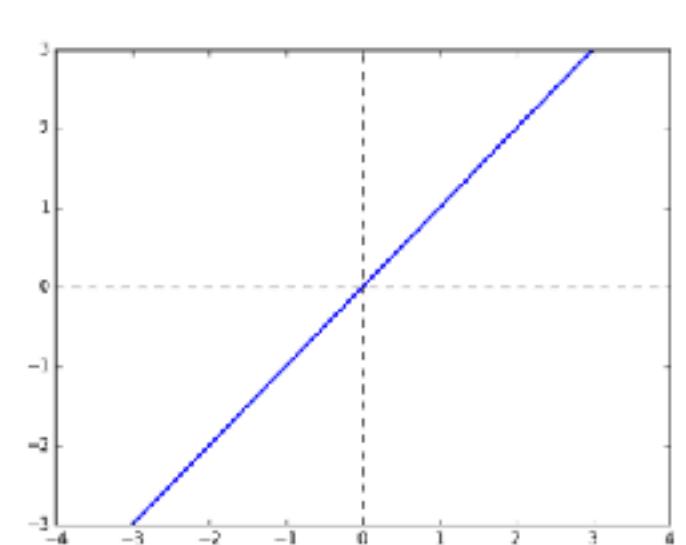


Gradient Descent



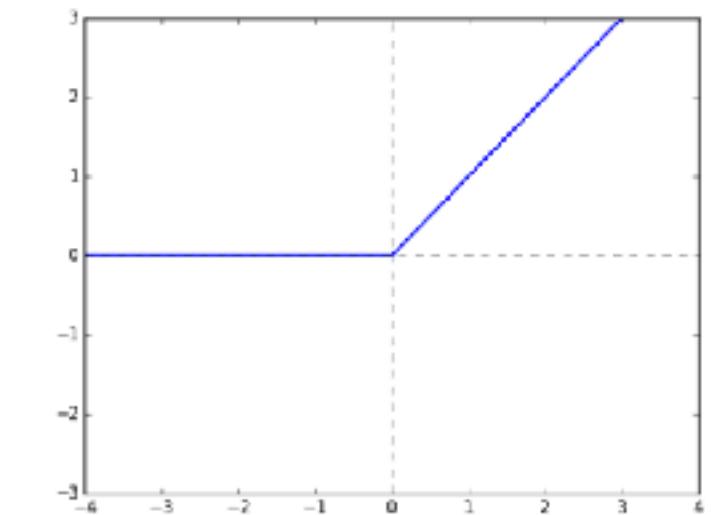
- Random order of updates means you can take different paths with same data
- Online approximation to batch
- Subsample training set for faster progress
- Always the same path for the same data, order irrelevant
- Offline

## Some activation functions:



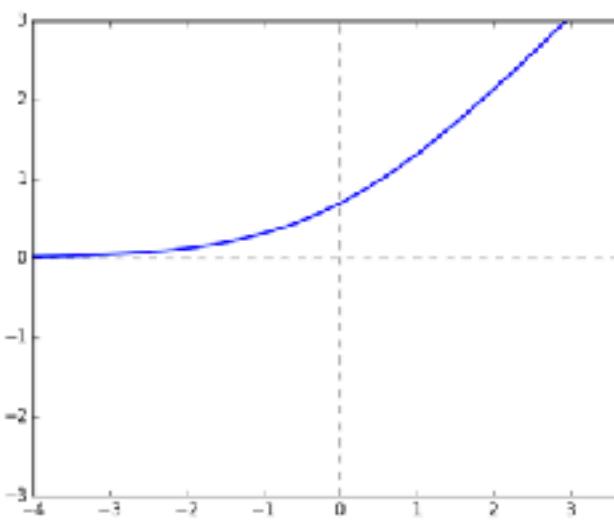
**Linear**

$$y = z$$



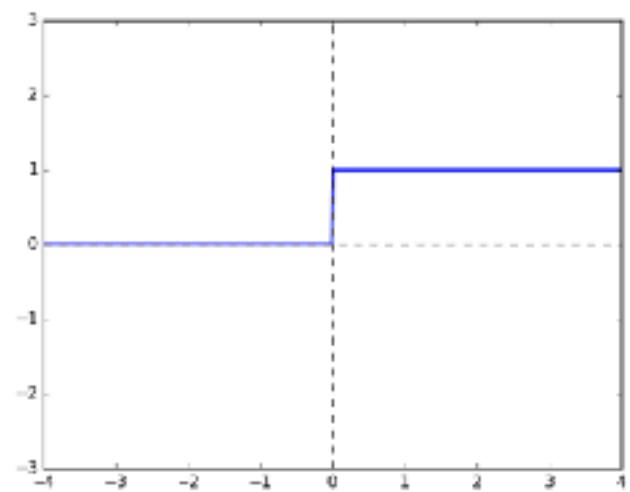
**Rectified Linear Unit  
(ReLU)**

$$y = \max(0, z)$$



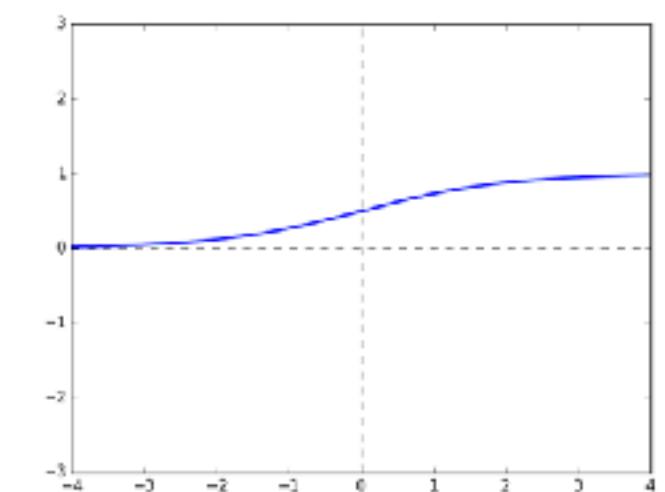
**Soft ReLU**

$$y = \log 1 + e^z$$



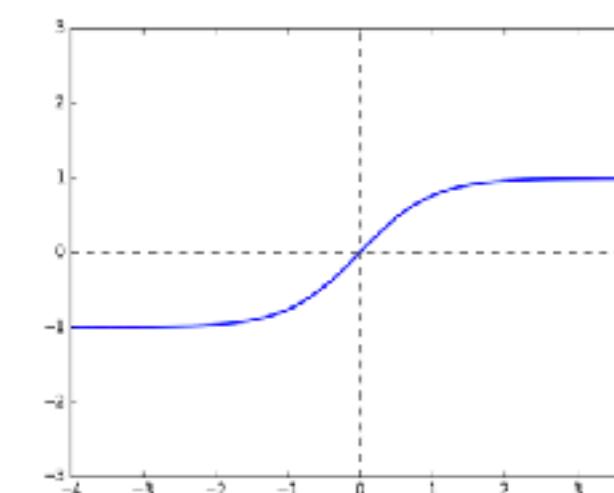
**Hard Threshold**

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



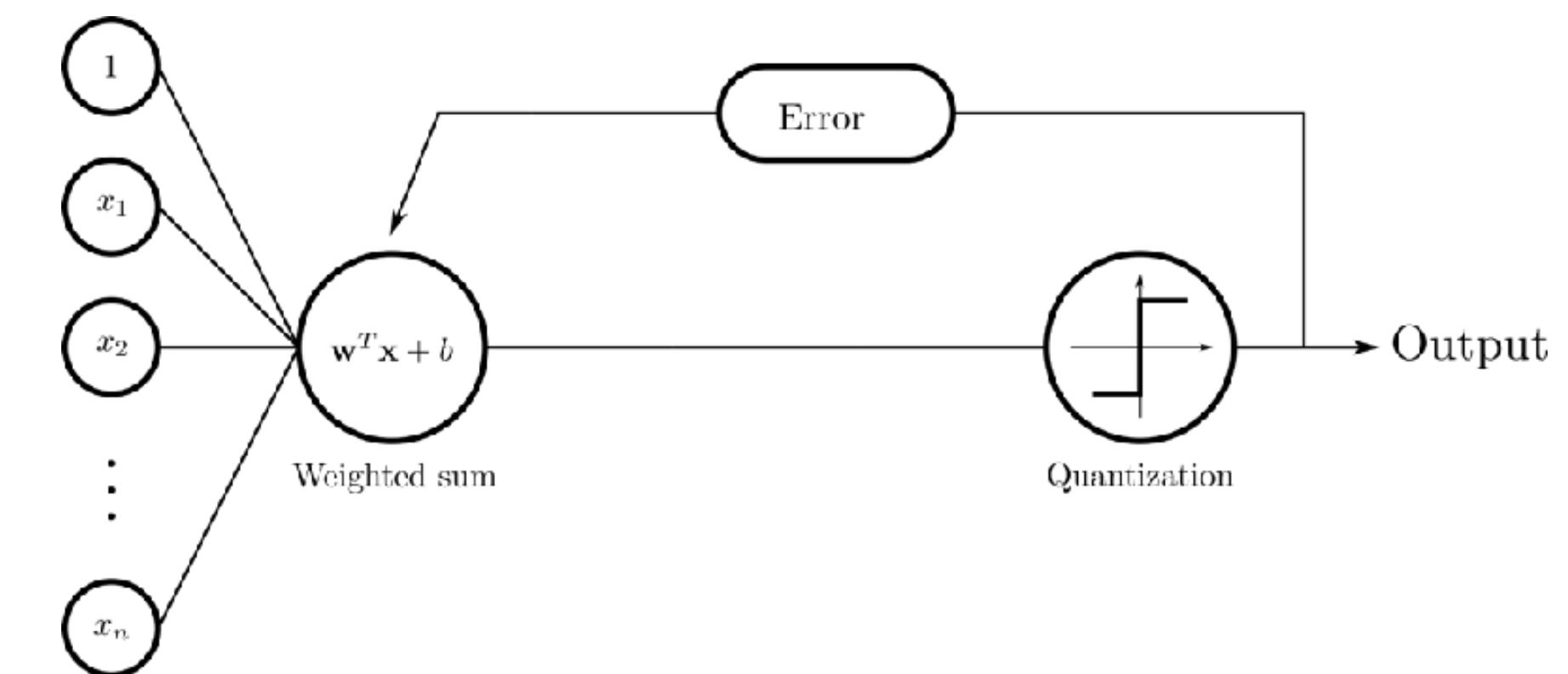
**Logistic**

$$y = \frac{1}{1 + e^{-z}}$$



**Hyperbolic Tangent  
(tanh)**

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



# Multilayer NNs

## A simple example architecture

- Each layer is a function of the one below it

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x})$$

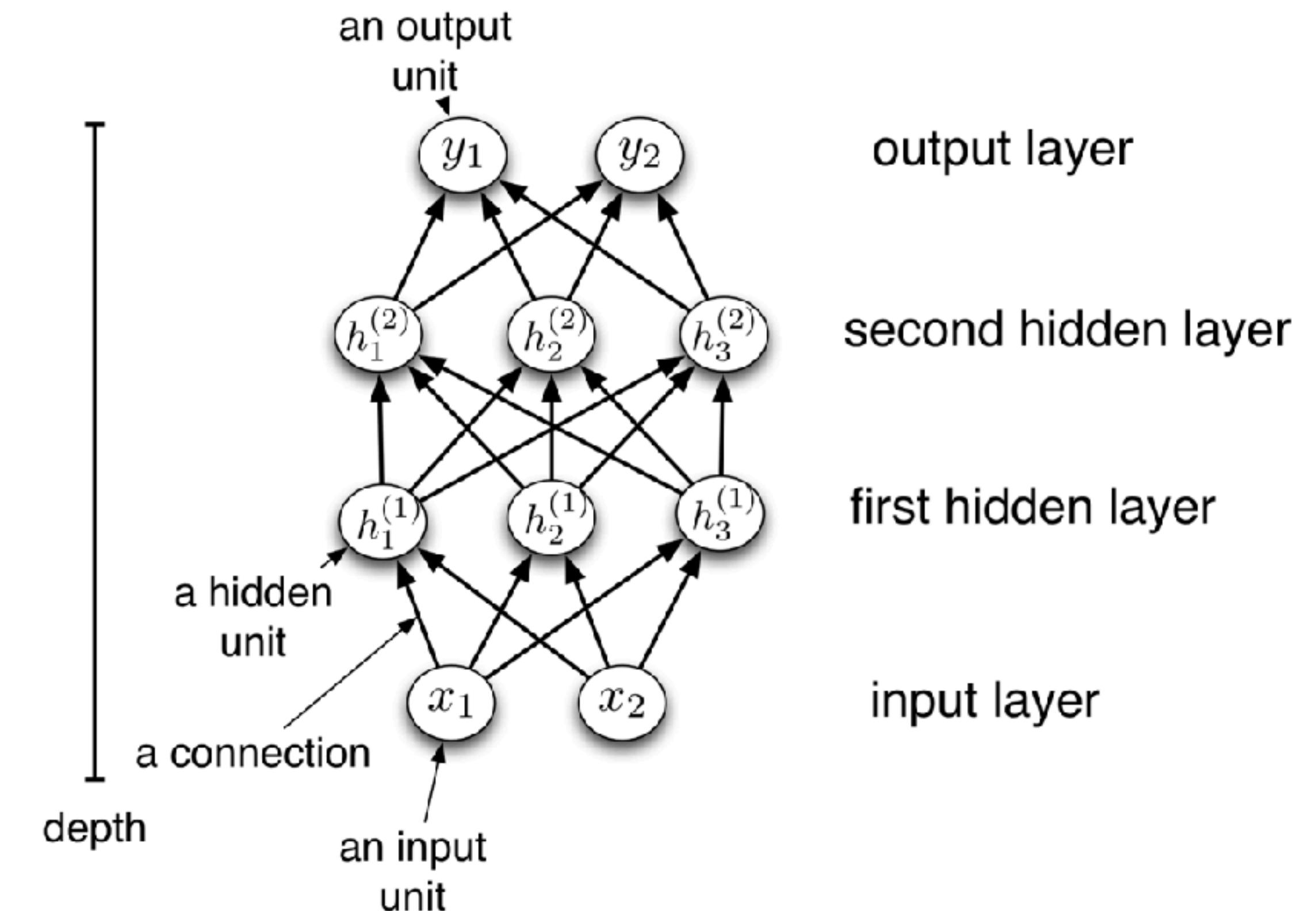
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)})$$

...

$$\mathbf{y} = f^{(D)}(\mathbf{h}^{(D)})$$

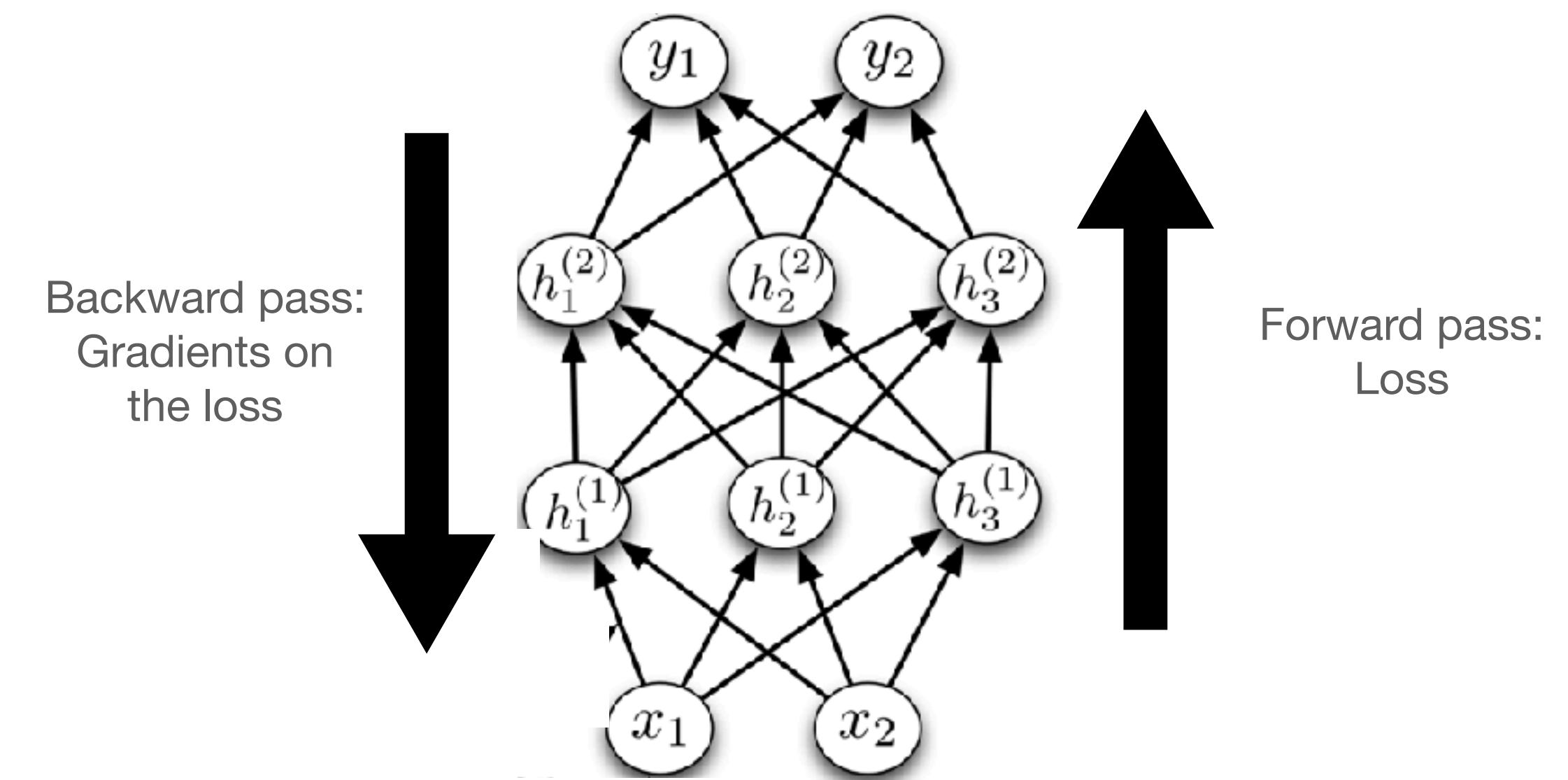
- Or more simply it is a modular composition of the functions of each layer

$$\mathbf{y} = f^{(1)} \circ f^{(2)} \dots \circ f^{(D)}$$



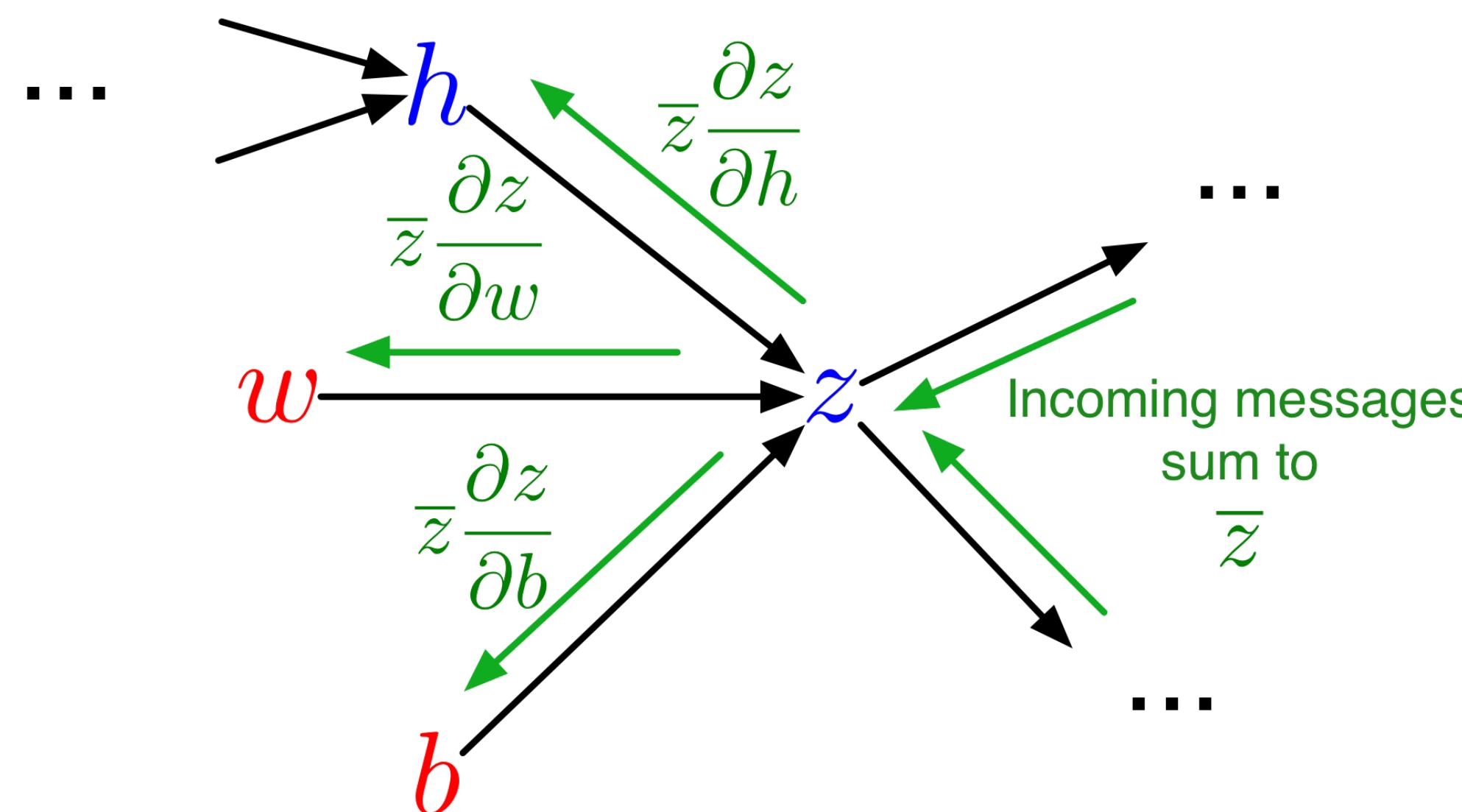
# Backpropagation

- Forward pass: calculate the outputs... and therefore the loss
- Backward pass: calculate the gradients and adjust the weights
  - At each layer using the gradient from the layer above



# Backpropagation

## Backprop as message passing:



- Each node receives a bunch of messages from its children, which it aggregates to get its error signal. It then passes messages to its parents.
- This provides modularity, since each node only has to know how to compute derivatives with respect to its arguments, and doesn't have to know anything about the rest of the graph.

