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

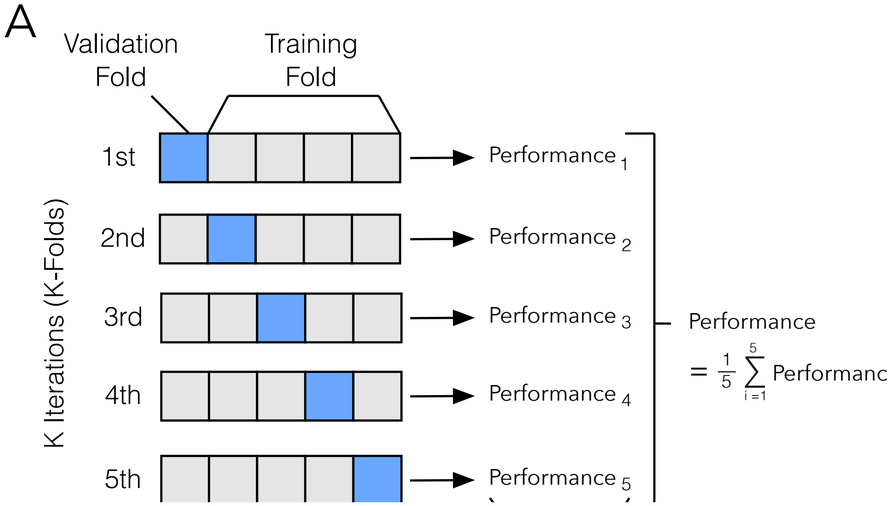
Slides in this presentation are from material kindly provided by Zhuowen Tu and others credited at those slides

NN links

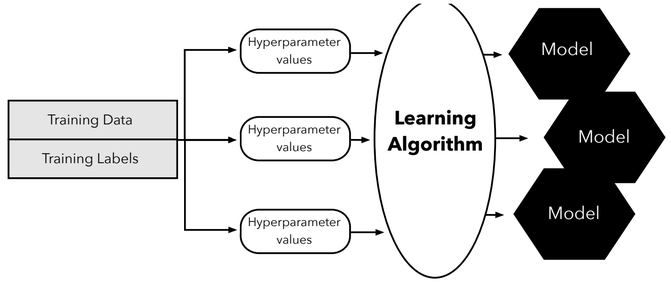
- https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- https://huggingface.co/learn/nlp-course/chapter1/1
- https://course.fast.ai/

Model selection

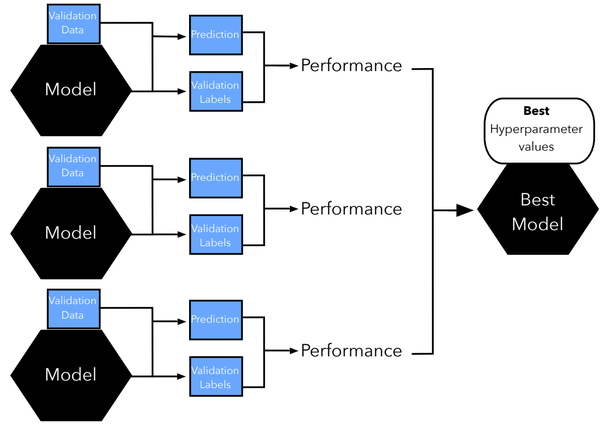
k-Fold Cross-Validation



2



3

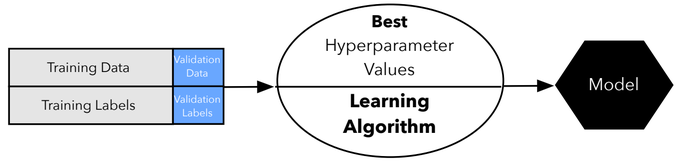


Default method for our project:

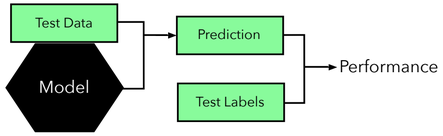
For each trial:

- 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)

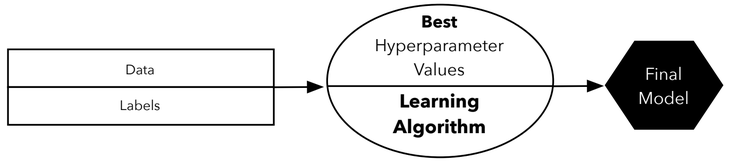
4



5



6



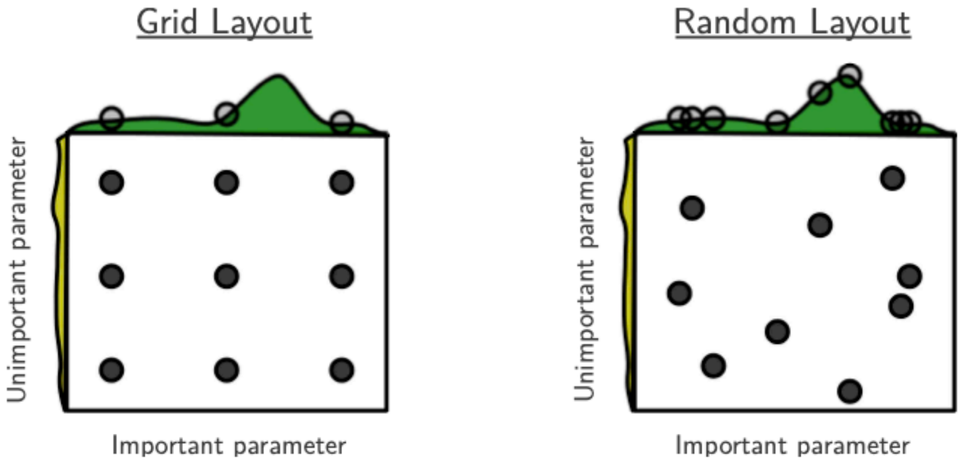
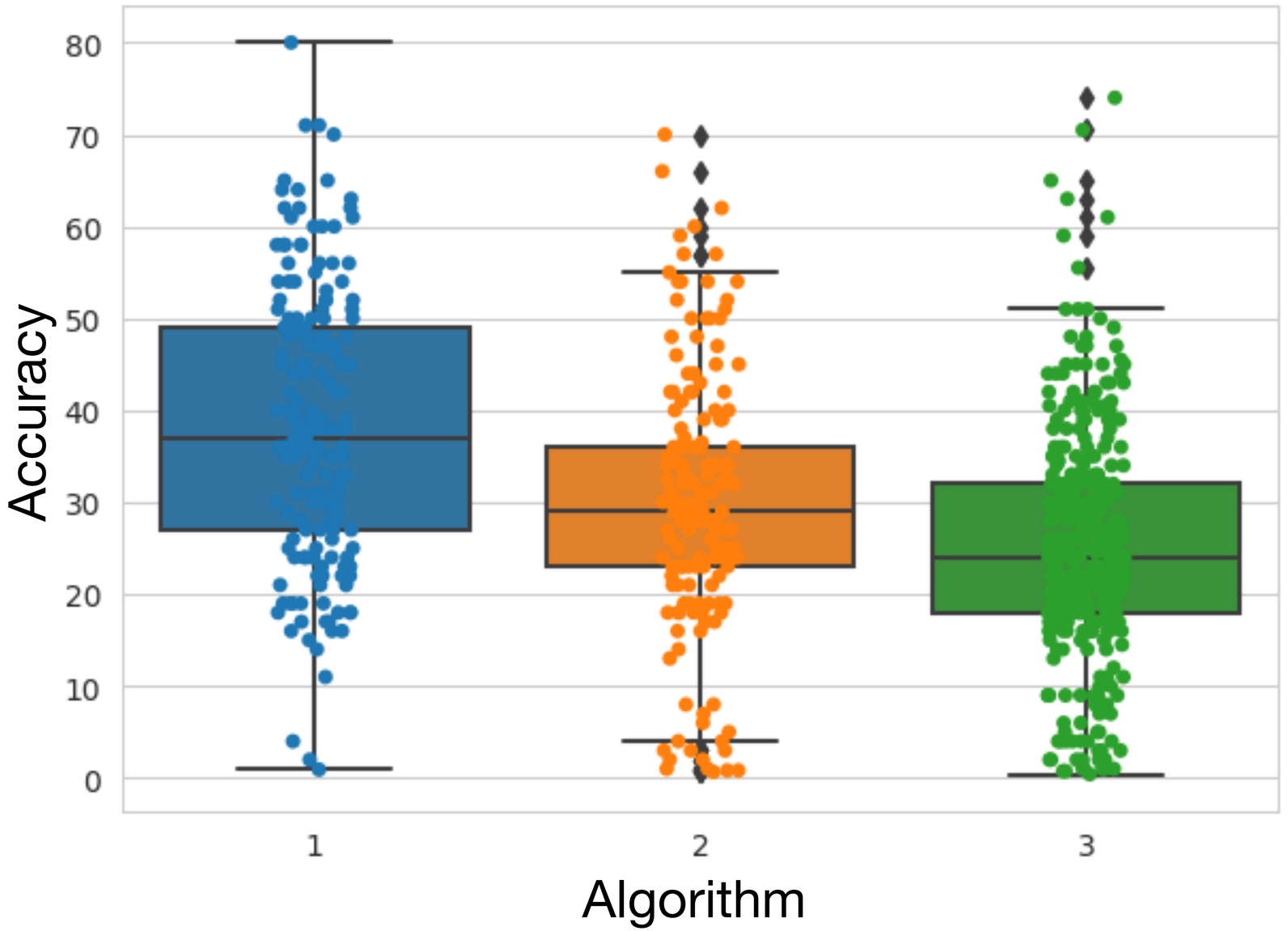
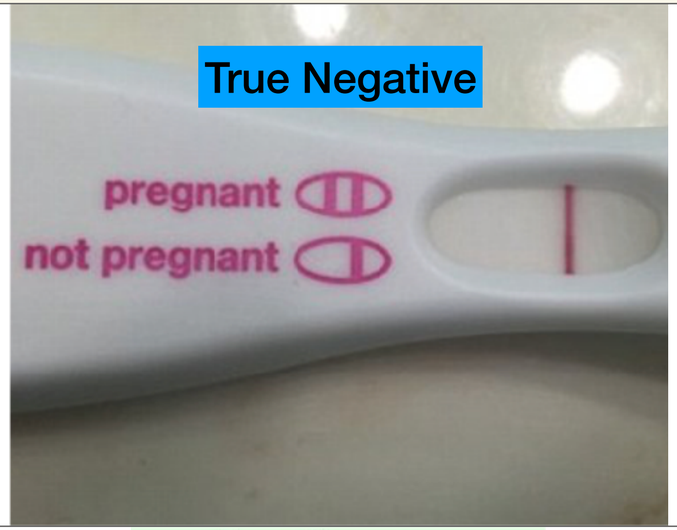
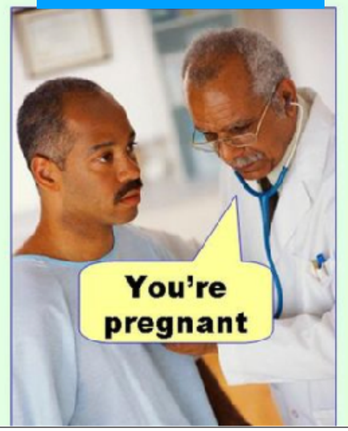
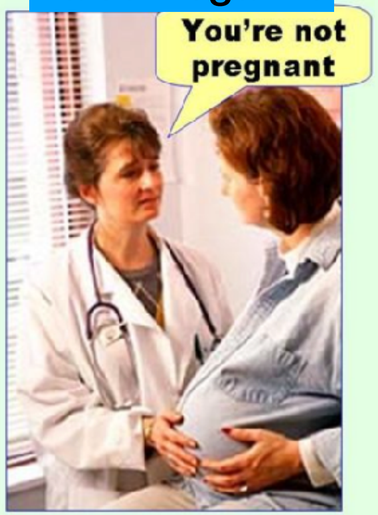
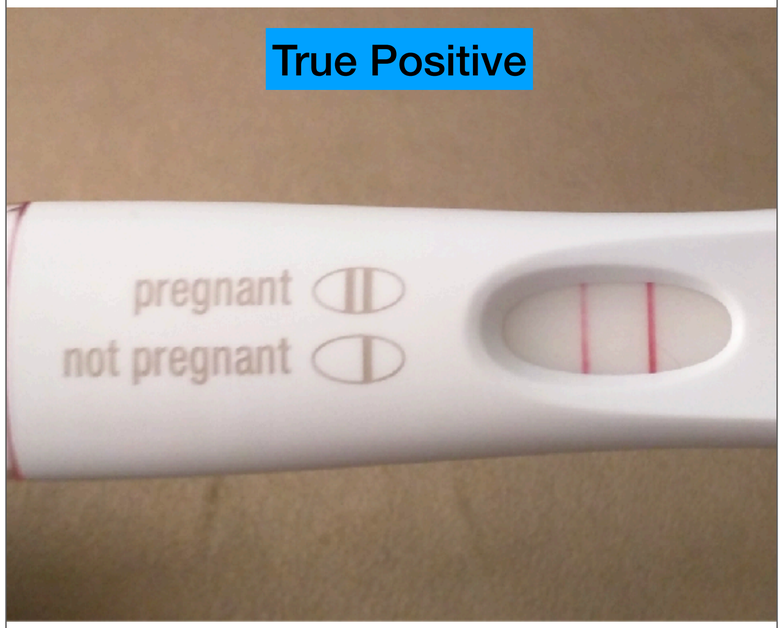


Figure 1: Grid and random search of nine trials for optimizing a function $f(x, y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct 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.


Accuracy
80
70
60
50
40
30
20
10
0
1
2
3
Algorithm

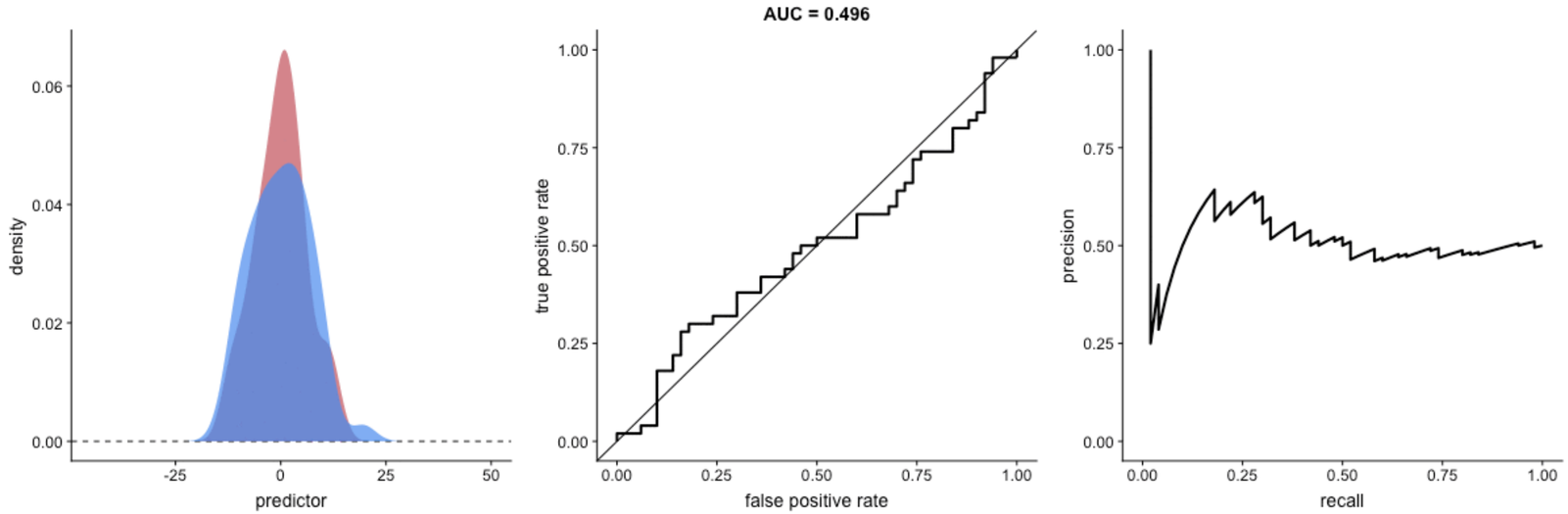
Metrics

		Our algorithm	
		-	+
Actual class	-	True Negative 	Type I error False Positive 
	+	Type II error False Negative 	True Positive 

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

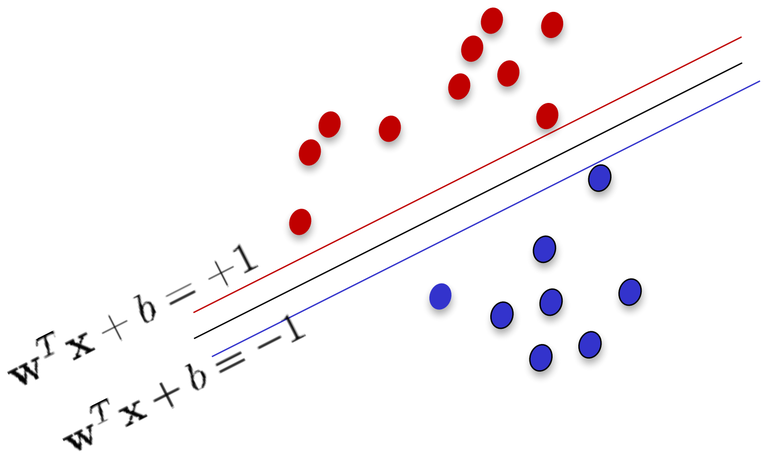
ROC and PR curves

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



SVM

SVM



$$\text{margin } M = \frac{2}{\sqrt{\mathbf{w}^T\mathbf{w}}}$$

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

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

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

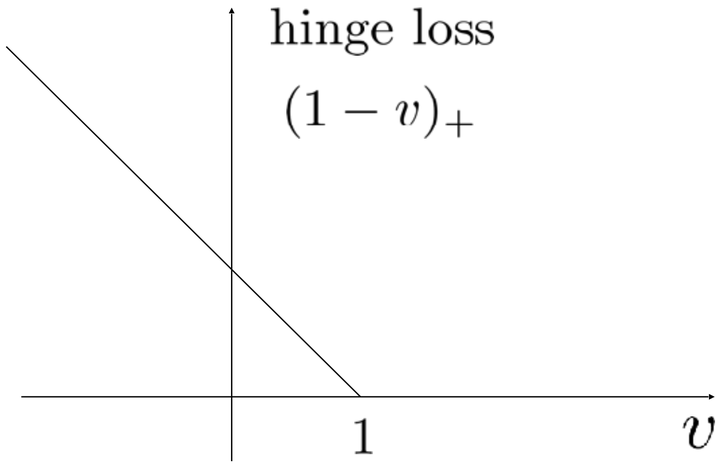
Find: $\arg\min_{\mathbf{w}} \frac{1}{2}||\mathbf{w}||^2$ subject to $y_i(\mathbf{w}^T\mathbf{x} + b) - 1 \geq 0$

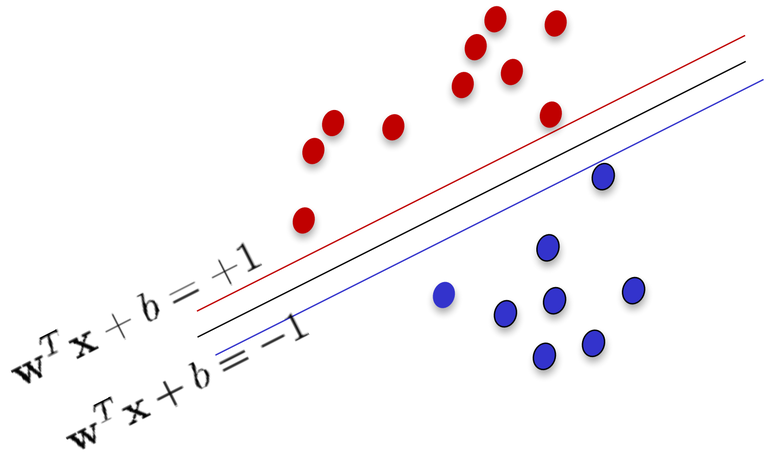
Hinge Loss

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

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

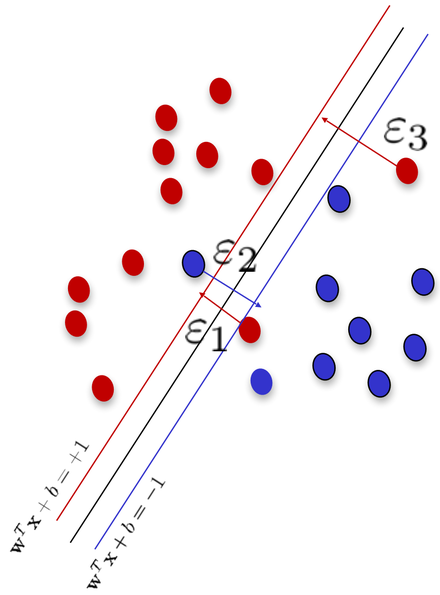
Hinge: $(1 - v)_+ = max(0, 1 - v)$



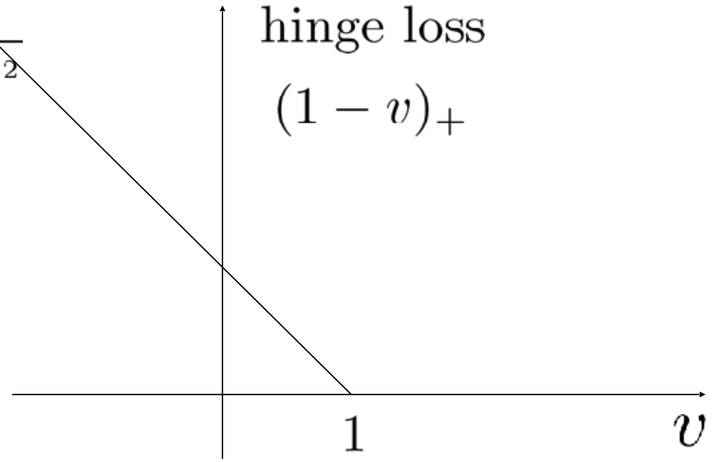


Find: $\arg\min_{\mathbf{w}} \frac{1}{2}||\mathbf{w}||^2 + C \times \sum_{i=1}^{n}(1 - y_i \times (\mathbf{w}^T\mathbf{x}_i + b))_+$

SVM with non-separable data



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



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

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

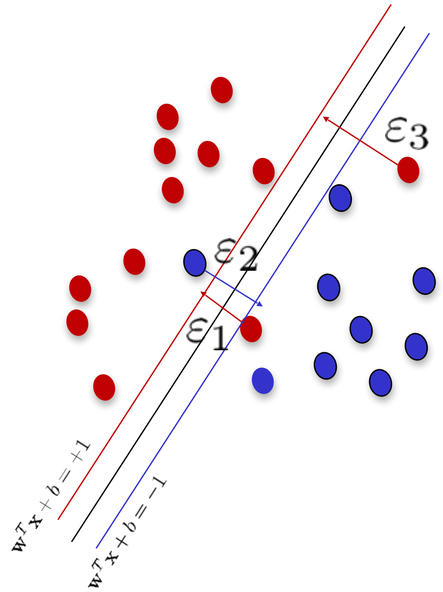
subject to: $y_i \times (\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \varepsilon_i$

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

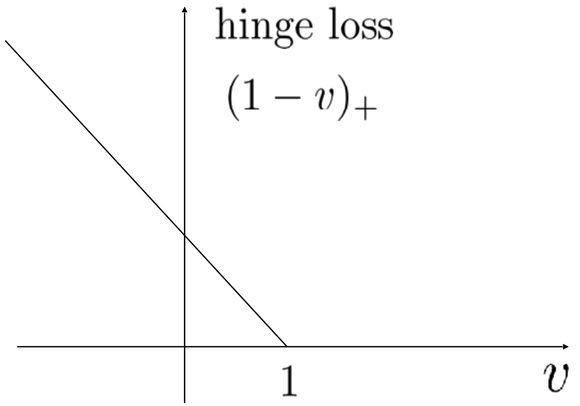
Find: $\arg\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C \times \sum_{i=1}^{n} max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$

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



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

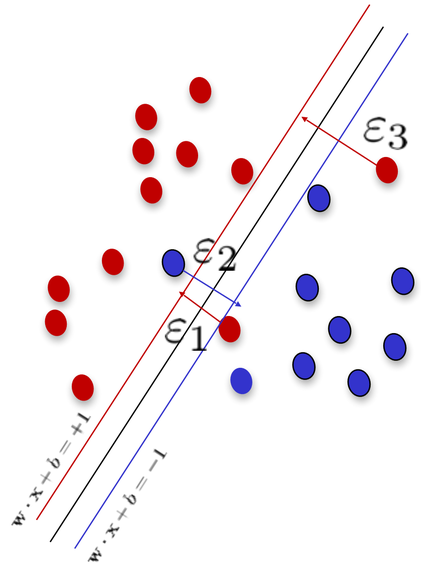


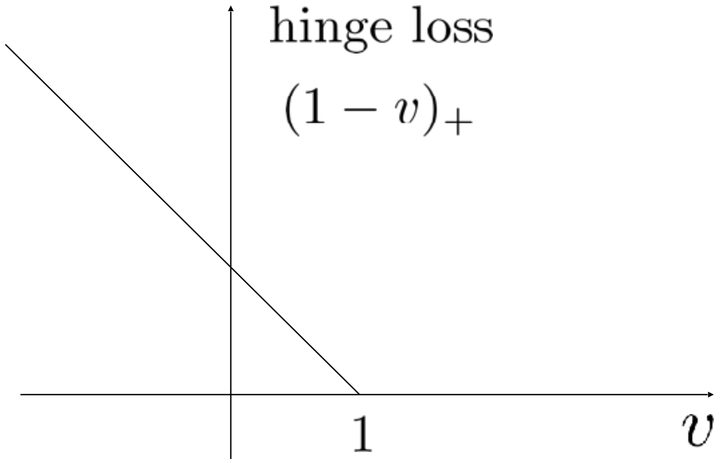
$$\text{Minimize } \mathcal{L}(\mathbf{w},b) = \tfrac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}(1-y_i(\mathbf{w}^T\mathbf{x}_i+b))_+$$

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

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

Convex?



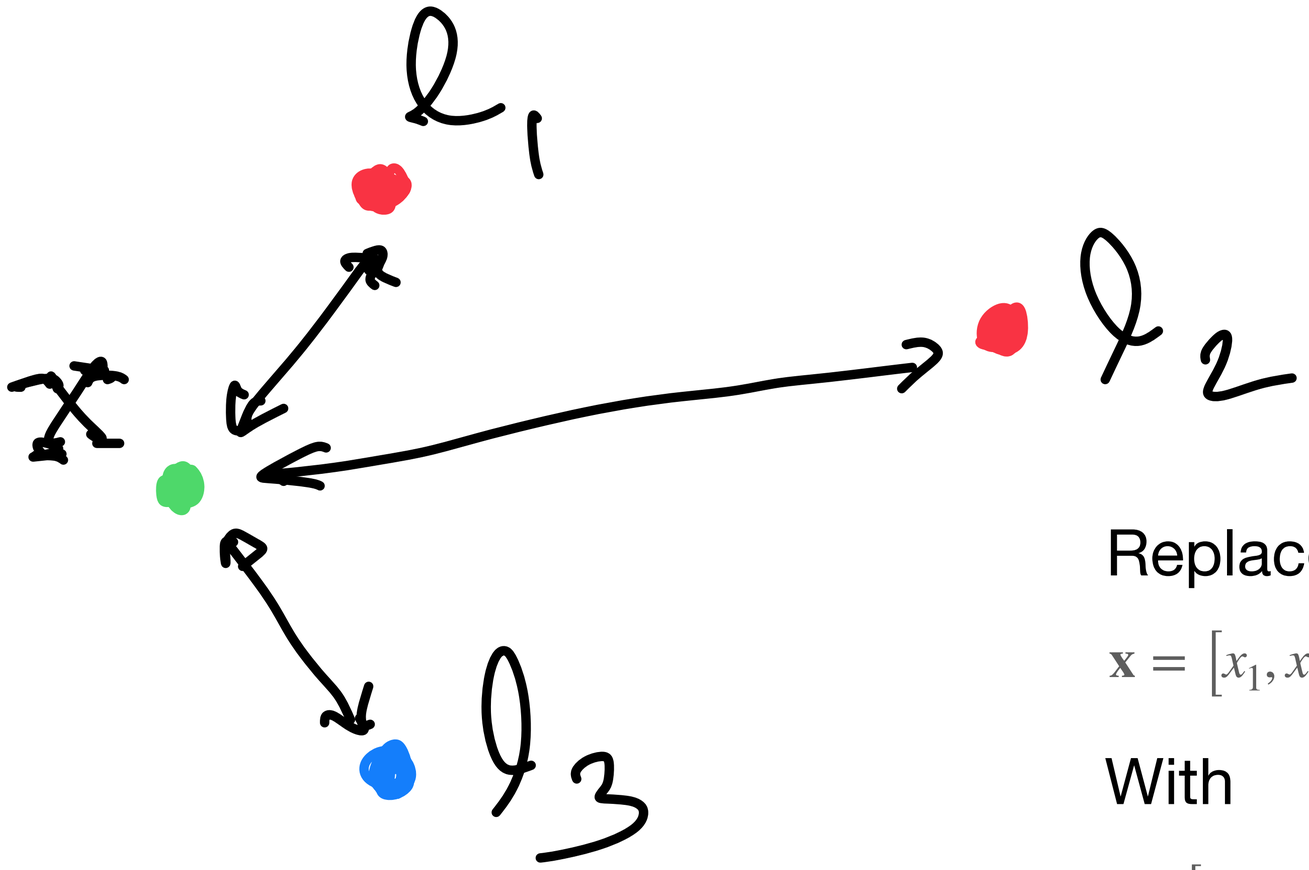


Find: $\arg\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C \times \sum_{i=1}^{n} max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b))$

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

The summation of convex functions is also convex.

The kernel trick



Replace

$$\mathbf{x} = \left[x_1, x_2, \ \ldots \ x_m\right]^T$$

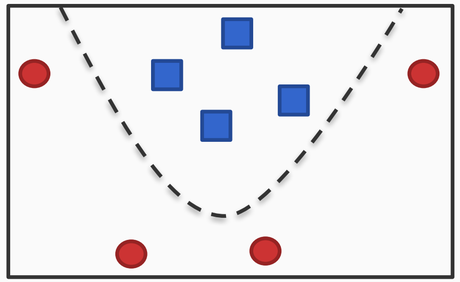
With

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

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

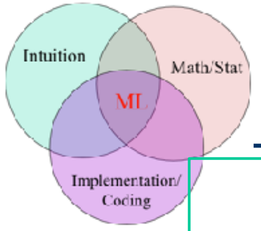
$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

Inner product: output is a scalar!

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

- not the input space

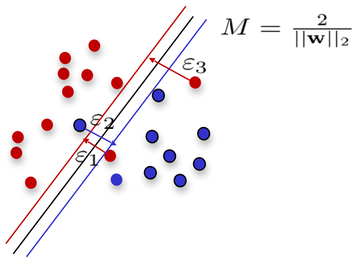
- not in the higher dimensional space the kernel transforms into



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

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

Small number of hyper parameters

Popular because they are often good enough

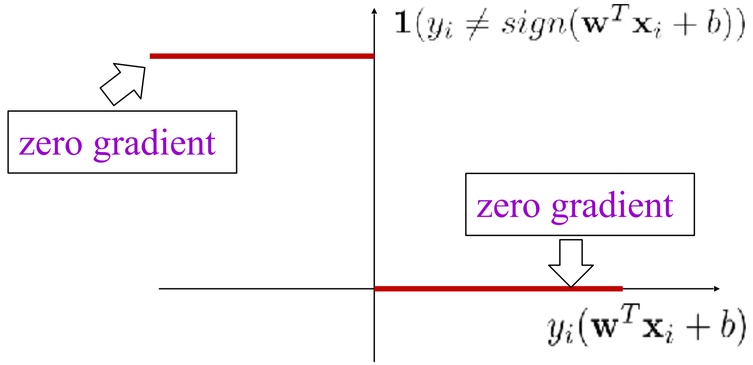
Cheap computationally if setup well, can scale to internet sized data when done right

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

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

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



Main motivation

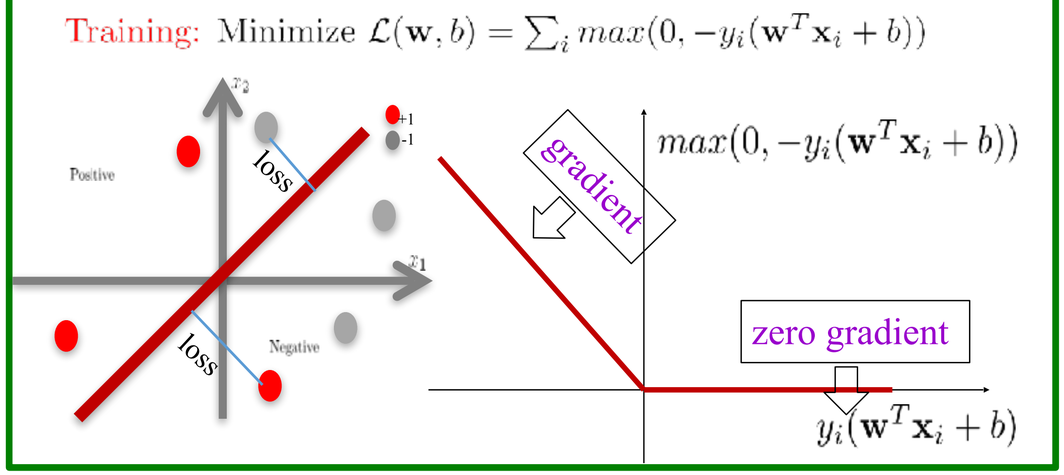
Hard->Half-hard->Soft Error

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

Zero gradient everywhere!

Half-hard loss (error) function

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.

Main motivation

Hard->Half-hard->Soft Error

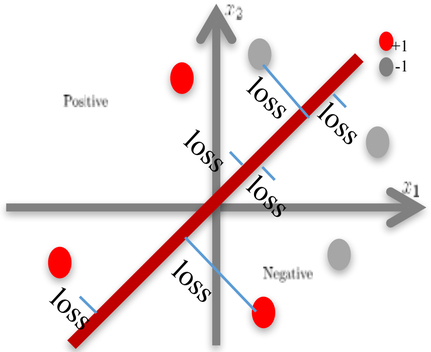
Soft loss (error) function

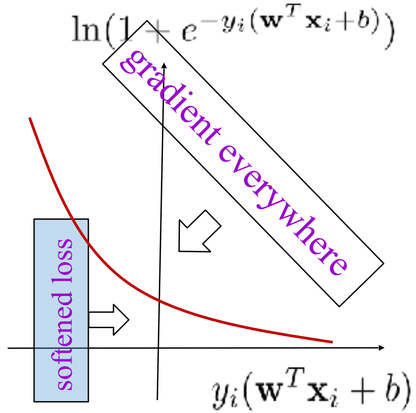
Main motivation

Hard->Half-hard->Soft Error

Loss used in logistic regression.

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





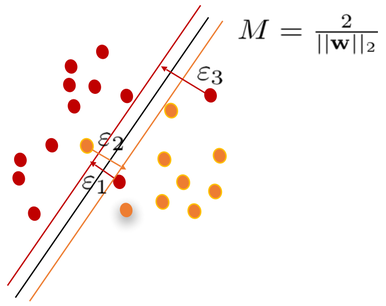
Every data point receives a loss (gradient everywhere).

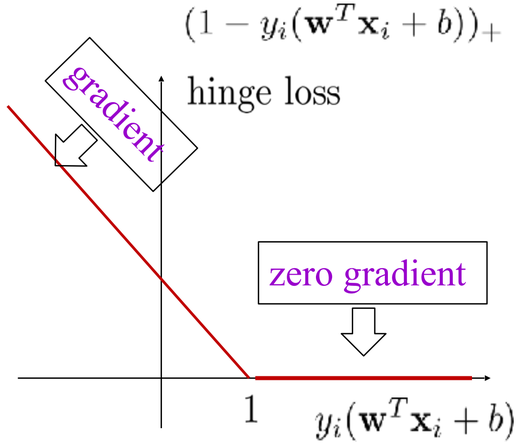
A loss based on the distance to the decision boundary for wrong classification (has a gradient).

Used in logistic regression classifier.

Loss in SVM

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





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).

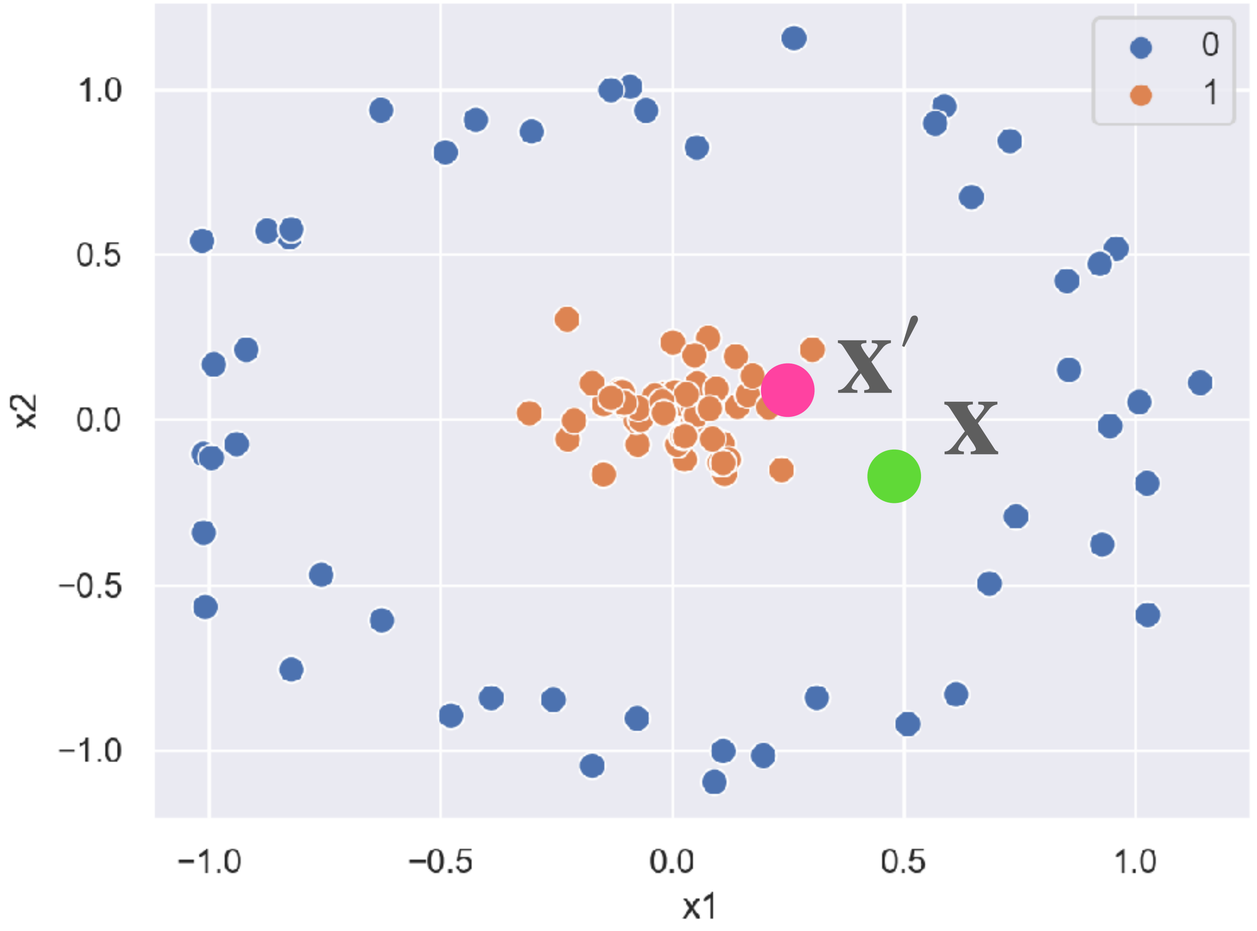
Main motivation

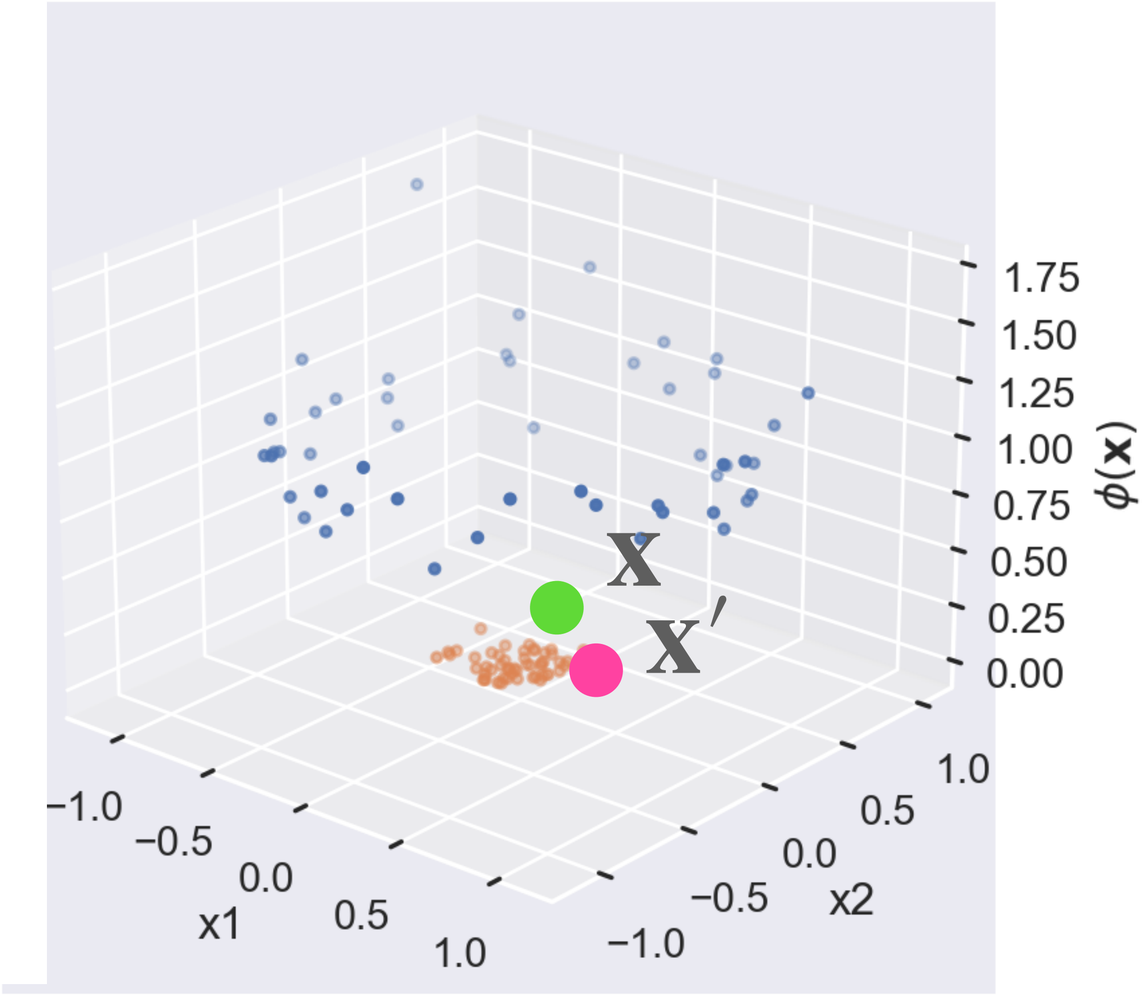
Hard->Half-hard->Soft Error

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

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

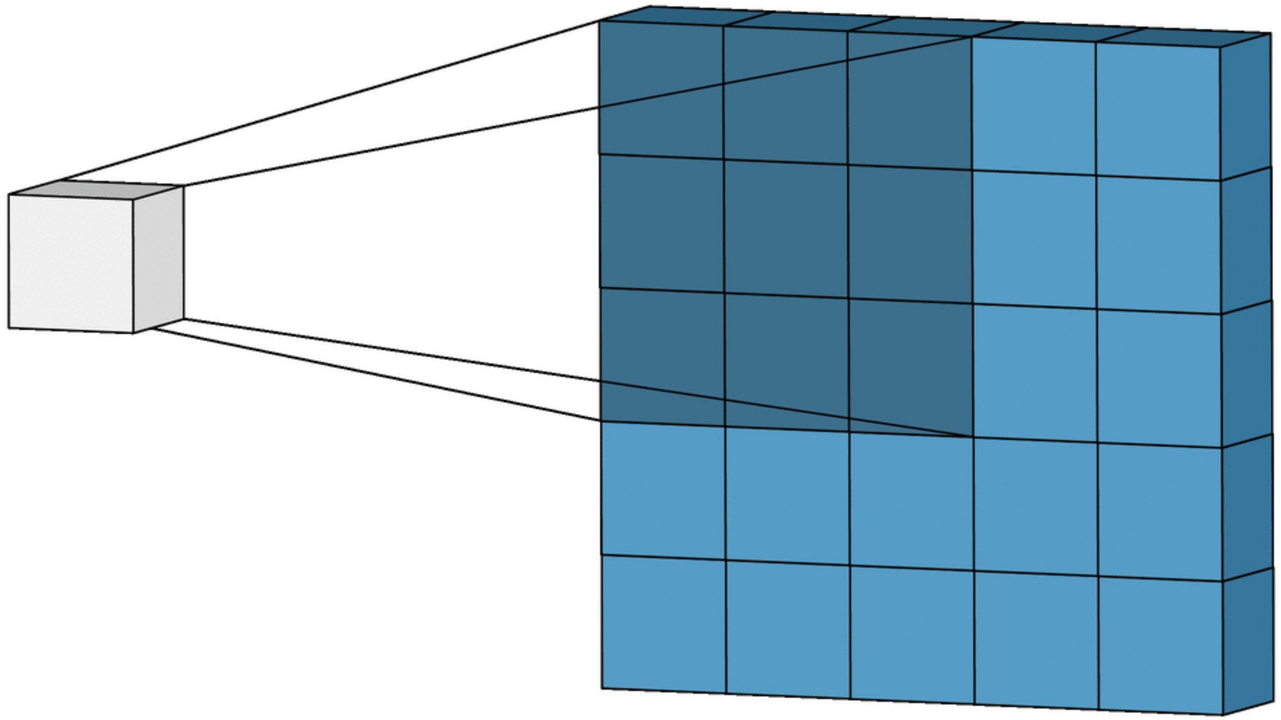
its in terms of a scalar dot product of vectors in $\mathcal{V}$!





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

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

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

Inner product: output is a scalar!

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

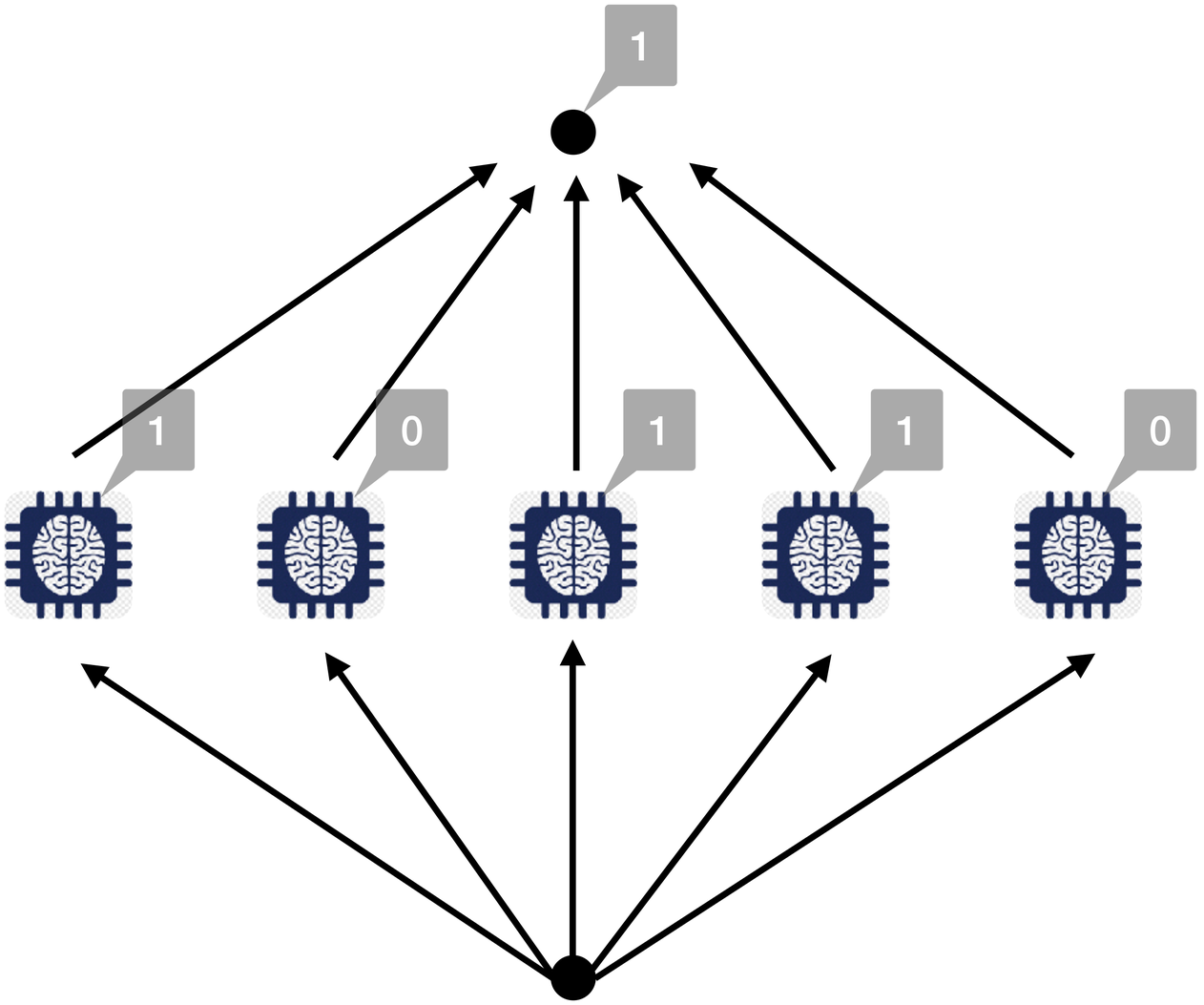
- not the input space

- 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!

Ensemble

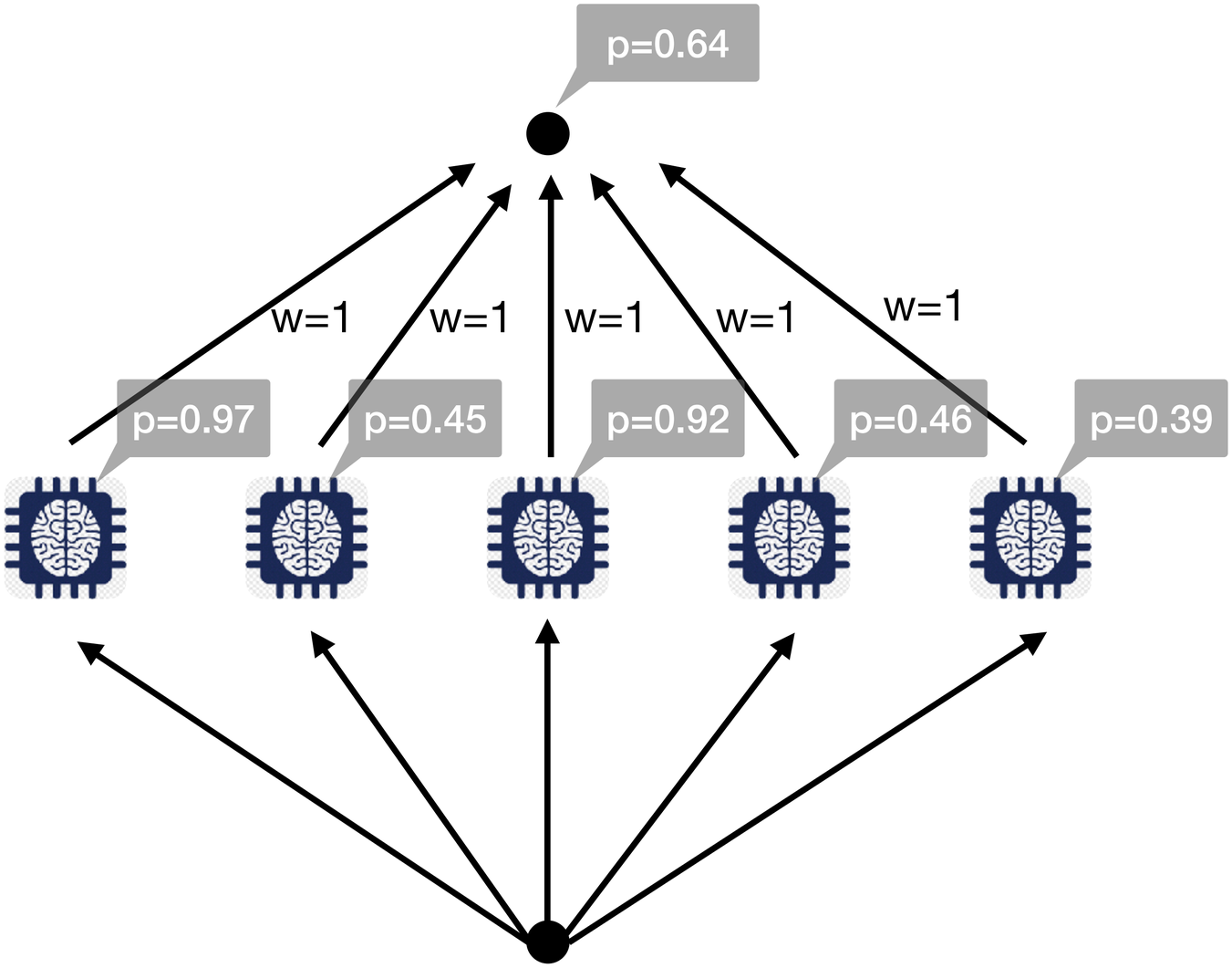


Majority vote

Diverse learners

(fully trained)

Test set
data point



Soft vote

Weights per classifier (optional)

Diverse learners

(fully trained)

Test set data point

Bagging
(Bootstrap Aggregating)

Algorithm 1 Bagging

1: Let n be the number of bootstrap samples
2:
3: **for** i=1 to n **do**
4: Draw bootstrap sample of size m, $\mathcal{D}_i$
5: Train base classifier h_i on $\mathcal{D}_i$
6: $\hat{y} = mode\{h_1(\mathbf{x}), ..., h_n(\mathbf{x})\}$

Out Of Bag error - a built-in generalization estimate

Note the upward bias!

For each observation $(x_i, y_i) \in S$, construct a predictor by averaging only those $h(x_n)$ created from bootstrap samples missing (x_i, y_i)

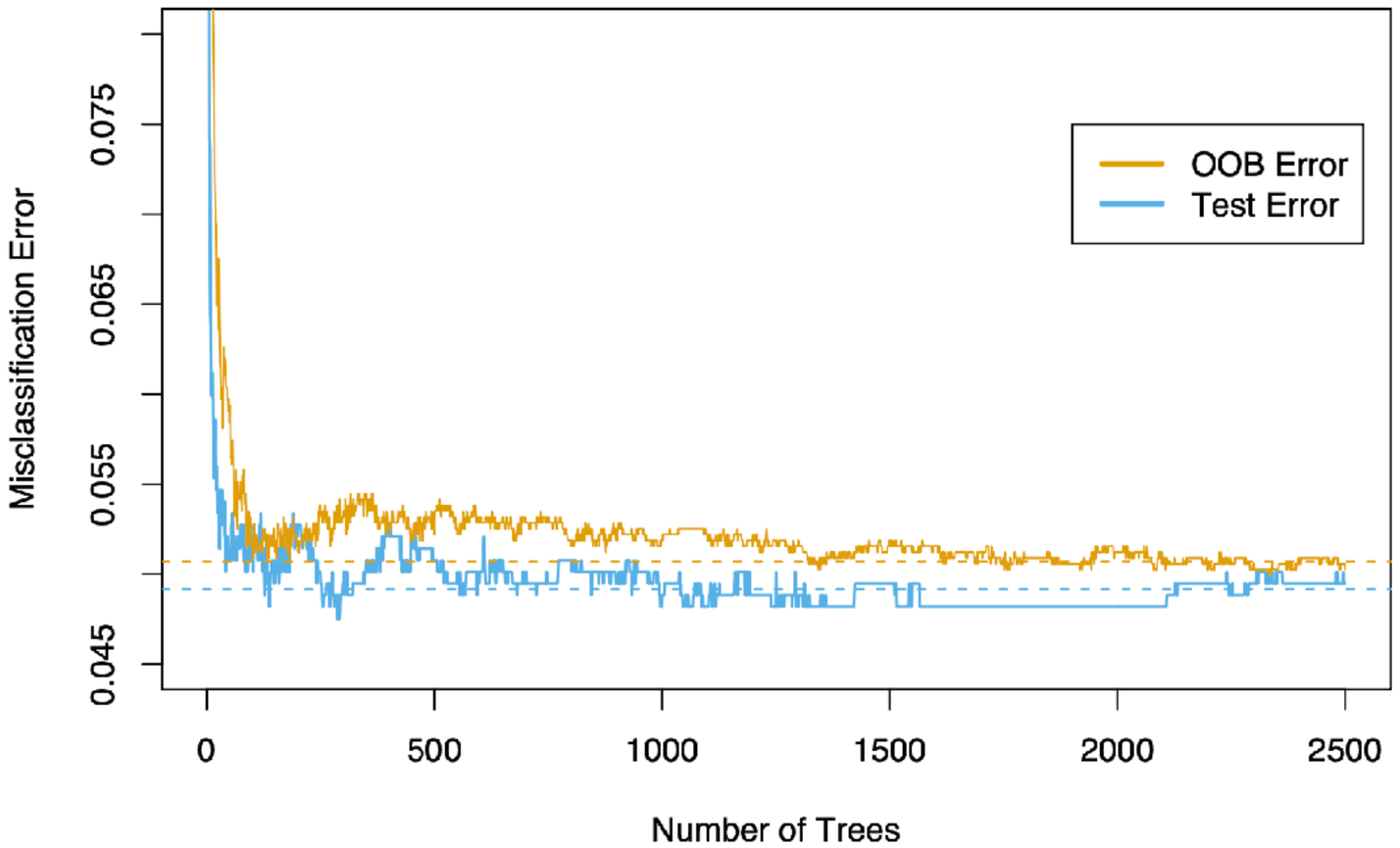


FIGURE 15.4. OOB *error computed on the* spam *training data, compared to the test error computed on the test set.*

Hastie et al, Elements of Statistical Learning

OOB is pessimistic

- 0.632 estimate: because on average only 63.2% of the unique samples get into the bag

$$ACC_{.632} = \frac{1}{b}\sum_{i=1}^{b} 0.632\, ACC_{OOB_i} + 0.368\, ACC_{train_i}$$

- .632+ estimate: because the .632 estimate can be optimistic if the model tends to overfit

$$ACC_{.632+} = \frac{1}{b}\sum_{i=1}^{b} \left(\omega * ACC_{OOB_i} + (1-\omega) * ACC_{\text{train}_i}\right)$$

$$\omega = \frac{.632}{(1-.368)R},\ R = -\frac{ACC_{OOB_i} - ACC_{train_i}}{\gamma - (1 - ACC_{OOB_i})},$$ where γ is a constant calculated empirically on the dataset (the no information rate, related to class priors)

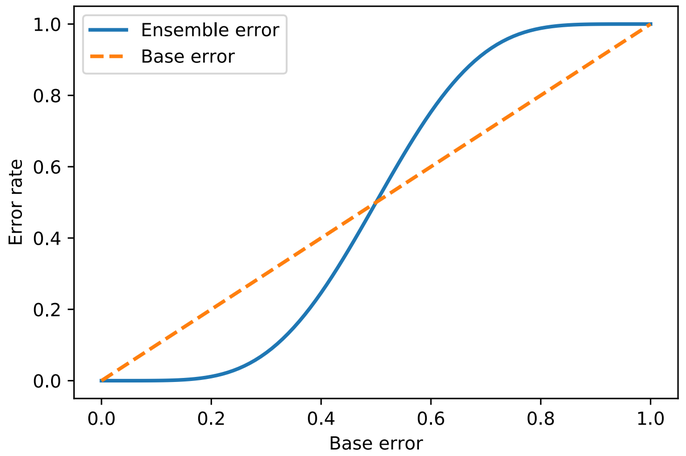
Ensemble summaries

- A necessary and sufficient condition for an ensemble to be more accurate than any of its individual classifiers is that those classifiers are accurate and diverse [Hansen & Salamon, *IEEE Trans Pattern Anal Mach Intell* 1990]

- An accurate classifier has a lower error rate than uniform random guessing

- Diverse classifiers make different errors when generalizing to new data.

$$\epsilon_{ens} = \sum_{k}^{n} \binom{n}{k} \epsilon^k (1-\epsilon)^{n-k}$$



Ensemble summaries

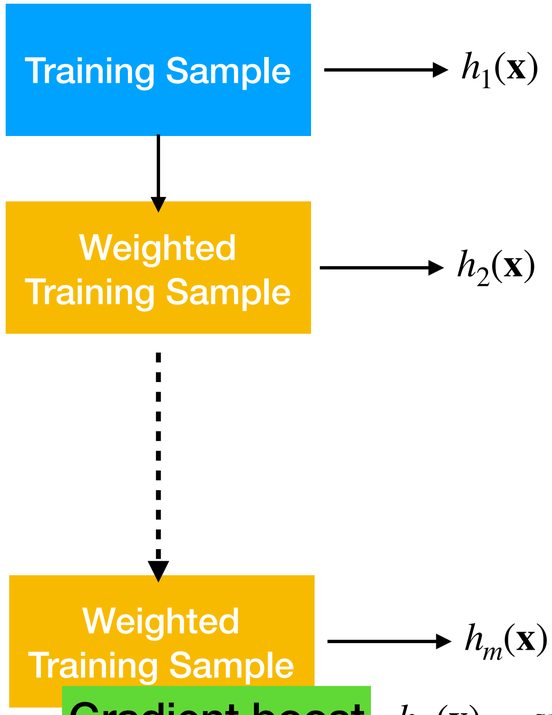
Classifier diversity can be achieved by injecting randomness via:

- Subsetting the training examples
- Subsetting the input features
- Non-deterministic algorithms

Ensemble summaries

- Ensembles often provide a boost in accuracy over base learner
- Increase runtime over base learner, but compute cycles are usually much cheaper than training instances
- Some ensemble approaches (e.g. bagging, random forests) are easily parallelized
- Prediction contests (e.g. Kaggle, Netflix Prize) usually won by ensemble solutions
- Ensemble models are usually low on the comprehensibility scale, although see work by [Craven & Shavlik, NIPS 1996] [Domingos, Intelligent Data Analysis 1998] [Van Assche & Blockeel, ECML 2007]

General Boosting



Gradient boost $h_m(\mathbf{x}) = sign\left(\sum_{j=1}^{m} w_j \, h_j(\mathbf{x})\right)$ for $h(\mathbf{x}) \in \{-1, 1\}$

AdaBoost $h_m(\mathbf{x}) = \arg\max_i \left(\sum_{j=1}^{m} w_j \, \mathbf{1}[h_j(\mathbf{x}) = i]\right)$ for $h(\mathbf{x}) = i, \quad i \in \{1, \ldots, n\}$

General boosting idea

- Initialize sample weighting with equal weights
- Loop until we hit boosting end conditions
  - Create a weak learner to learn a dataset where the misclassification error being minimized is sample weighted
  - Calculate classifier weighting based on the total misclassification error
  - Increase sample weighting for misclassified samples as a function of the classifier weighting
- Ensemble predicts a classifier weighted majority vote of the output of each classifier

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.
Initialize: $D_1(i) = 1/m$ for $i = 1, \ldots, m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : \mathcal{X} \to \{-1, +1\}$.
- Aim: select h_t with low weighted error:

$$\varepsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$.
- Update, for $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if } y_i \neq h_t(x_i) \end{cases}$$

  where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \operatorname{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right).$$

Fig. 1 The boosting algorithm AdaBoost.

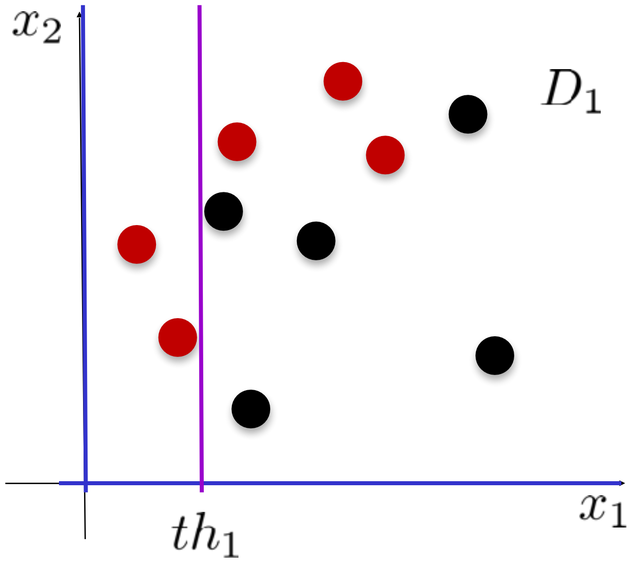
Toy Example

$$S_{training} = \{(\mathbf{x}_i, D_1(i), y_i), i = 1..n\} \qquad \mathbf{x} \in \mathbb{R}^2 \qquad y \in \{-1, +1\}$$

$y = +1$

$y = -1$

Minimize: $e_t = \sum_i D_t(i) \times \mathbf{1}(y_i \neq h_t(\mathbf{x}_i))$

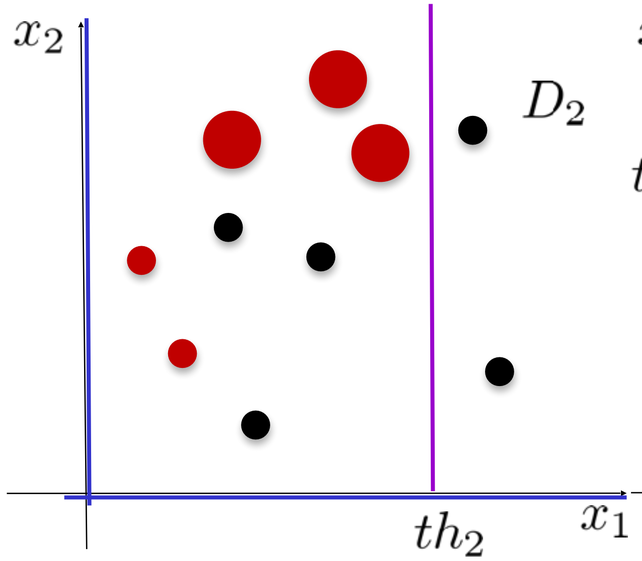


$$h_1(\mathbf{x}) = sign(x_1 \leq th_1)$$

$$\epsilon_1 = 3/9$$

$$\alpha_1 = \frac{1}{2}\ln(\frac{1-e_1}{e_1}) = 0.42$$

$$D_2(i) \propto D_1(i) \times \begin{cases} e^{-\alpha_1} & if\ y_i = h_1(\mathbf{x}_i) \\ e^{\alpha_1} & if\ y_i \neq h_1(\mathbf{x}_i) \end{cases}$$

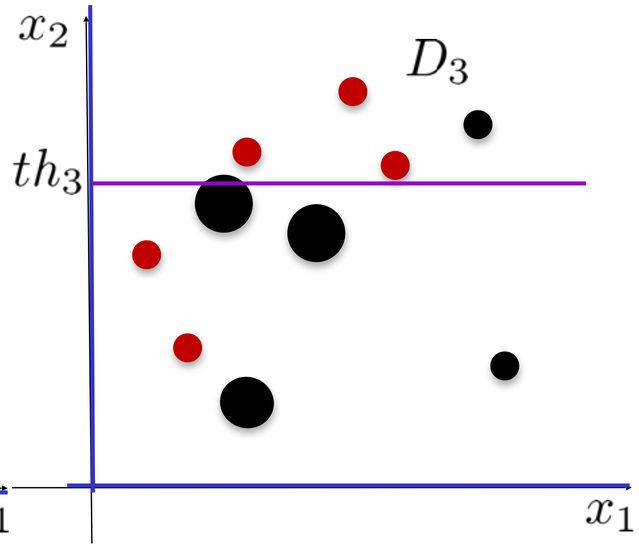


$$h_2(\mathbf{x}) = sign(x_1 \leq th_2)$$

$$\epsilon_2 = 0.215$$

$$\alpha_2 = \frac{1}{2}\ln(\frac{1-e_2}{e_2}) = 0.65$$

$$D_3(i) \propto D_2(i) \times \begin{cases} e^{-\alpha_2} & if\ y_i = h_2(\mathbf{x}_i) \\ e^{\alpha_2} & if\ y_i \neq h_2(\mathbf{x}_i) \end{cases}$$



$$h_3(\mathbf{x}) = sign(x_2 \geq th_3)$$

$$\epsilon_3 = 0.137$$

$$\alpha_3 = \frac{1}{2}\ln(\frac{1-e_3}{e_3}) = 0.92$$

$$D_4(i) \propto D_3(i) \times \begin{cases} e^{-\alpha_3} & if\ y_i = h_3(\mathbf{x}_i) \\ e^{\alpha_3} & if\ y_i \neq h_3(\mathbf{x}_i) \end{cases}$$

The final classifier: $H(\mathbf{x}) = sign(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}))$

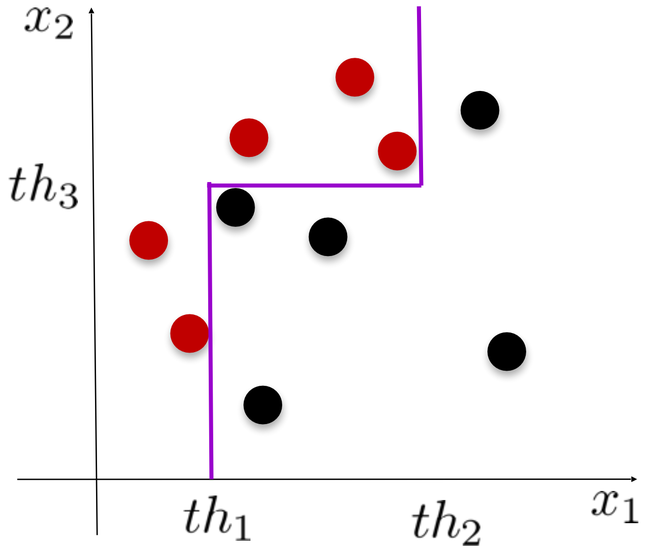
AdaBoost Example

$$S_{training} = \{(\mathbf{x}_i, y_i), i = 1..n\} \qquad \mathbf{x} \in \mathbb{R}^2 \qquad y \in \{-1, +1\}$$

$y = +1$

$y = -1$

Weak classifier: h decision stump.

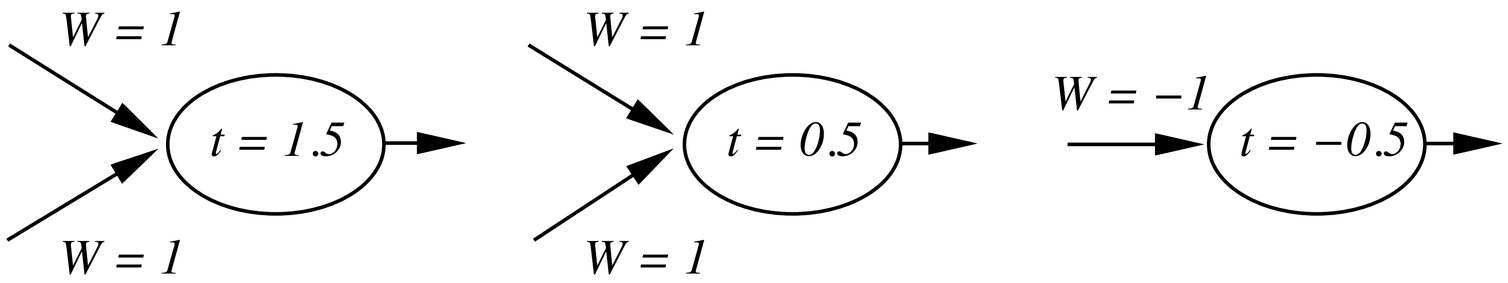


$$H(\mathbf{x}) = 0.42 \times h_1(\mathbf{x}) + 0.65 \times h_2(\mathbf{x}) + 0.92 \times h_3(\mathbf{x})$$

$$= 0.42 \times sign(x_1 \leq th_1) + 0.65 \times sign(x_1 \leq th_2) + 0.92 \times sign(x_2 \geq th_3)$$

NN

Units as Logic Gates

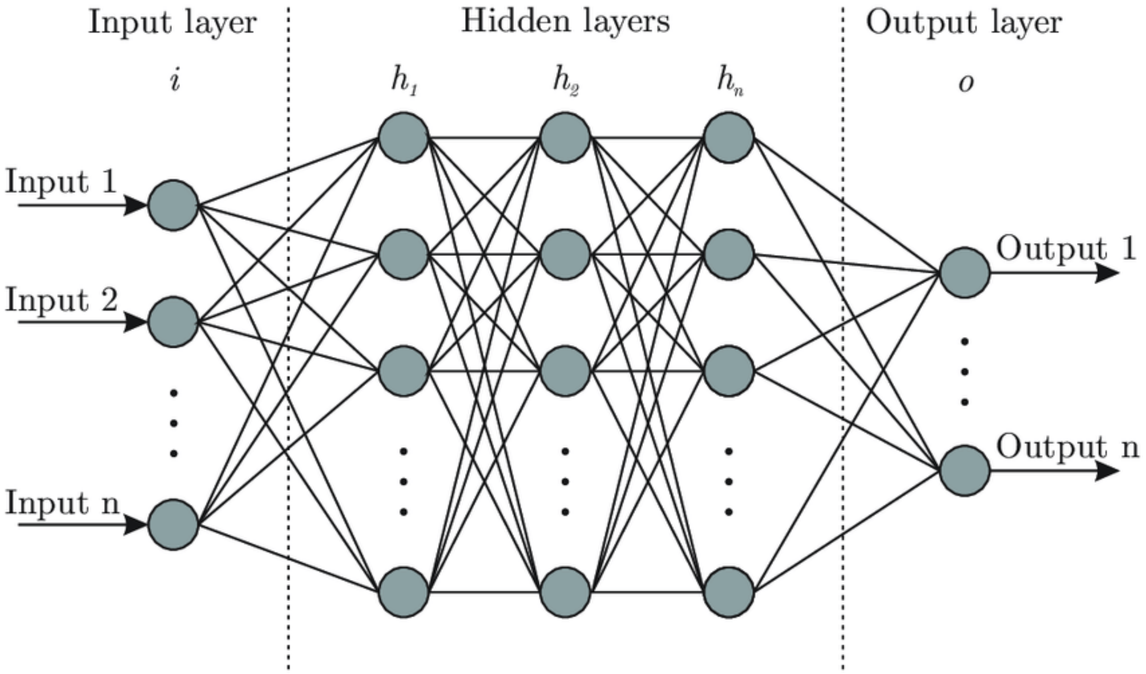


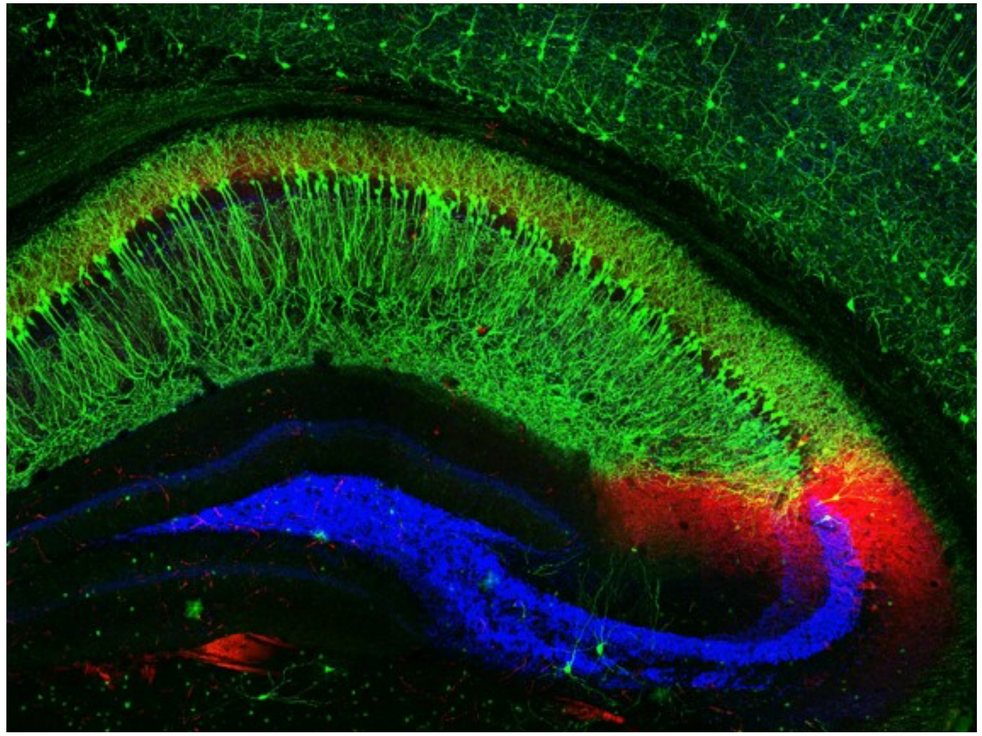
Activation function: $step_t$

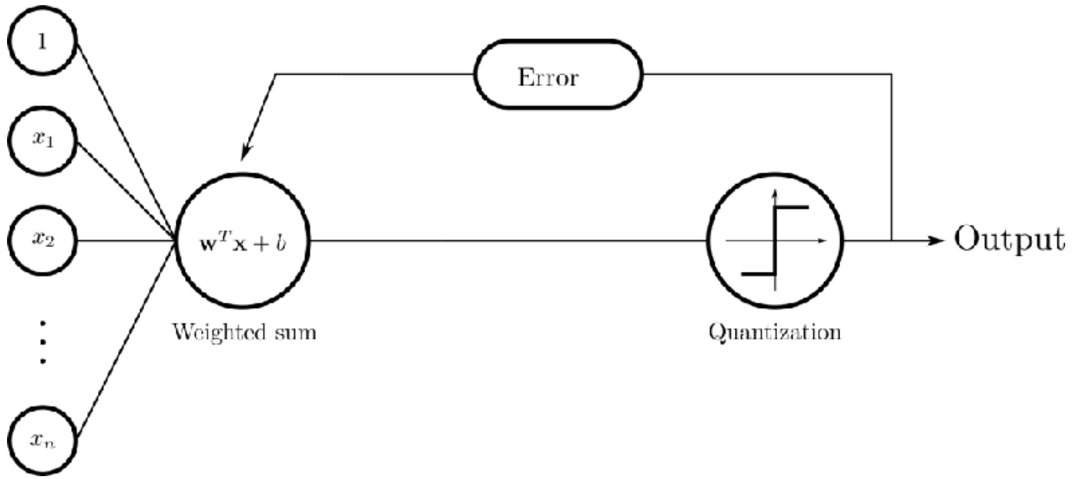
Since units can implement the $\wedge, \vee, \neg$ boolean operators, neural nets are **Turing-complete**: they can implement *any* computable function.

Diagram from Russell & Norvig

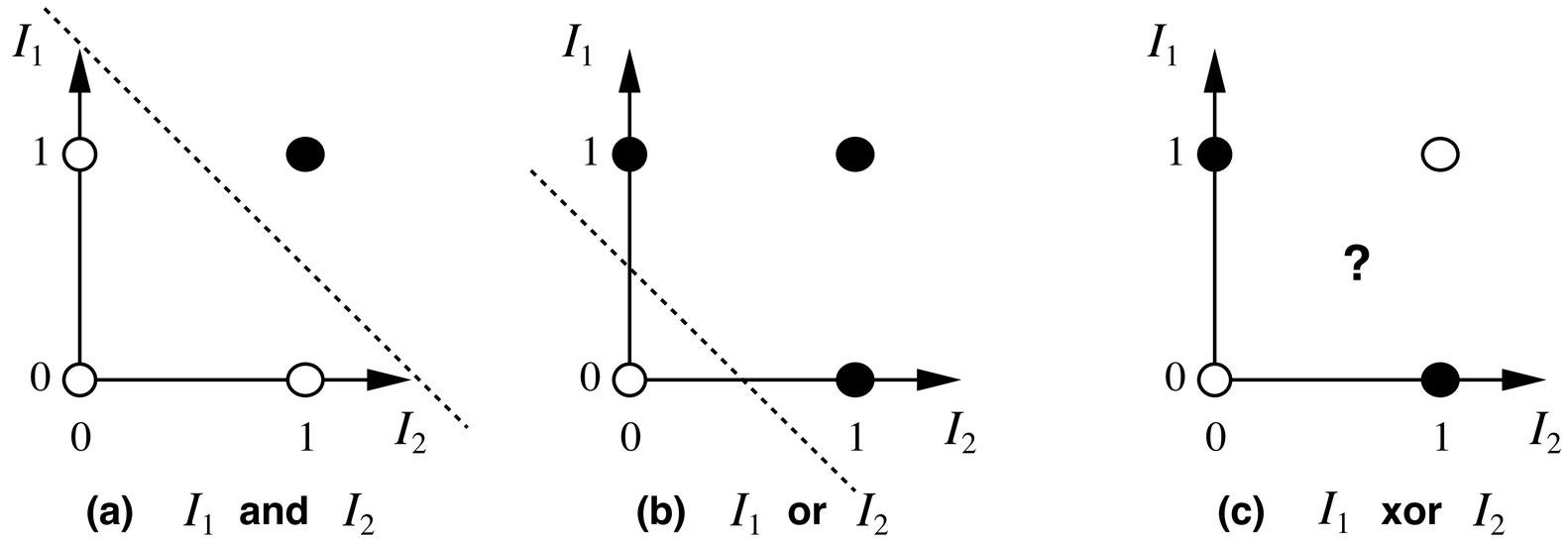
Artificial Neural Networks







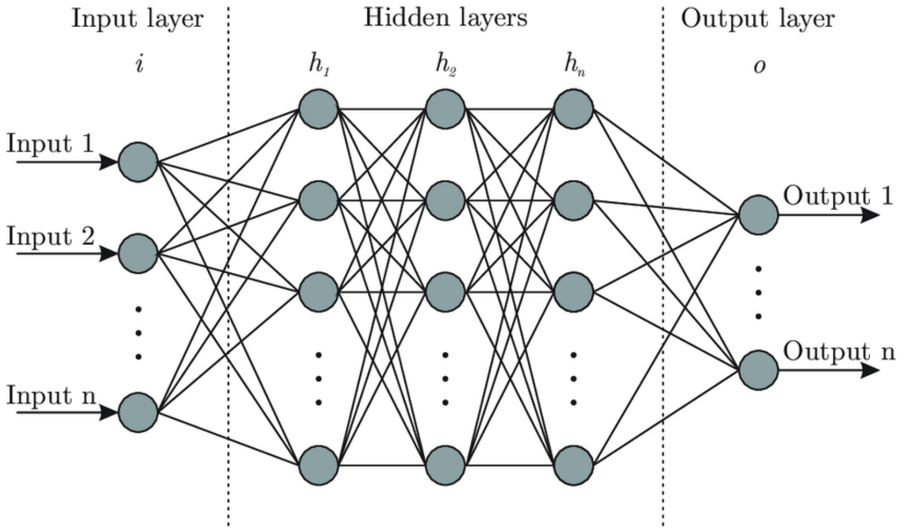
Linearly Separable Functions on a 2-dimensional Space

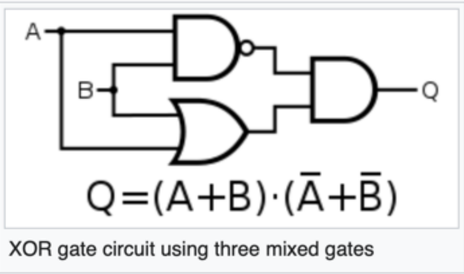


A black dot corresponds to an output value of 1. An empty dot corresponds to an output value of 0.

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”





XOR gate circuit using three mixed gates

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

$$\mathbf{w}_{t+1} = \mathbf{w}_t + (target_i - output_i)\mathbf{x}_i$$
$$b_{t+1} = b_t + (target_i - output_i)$$

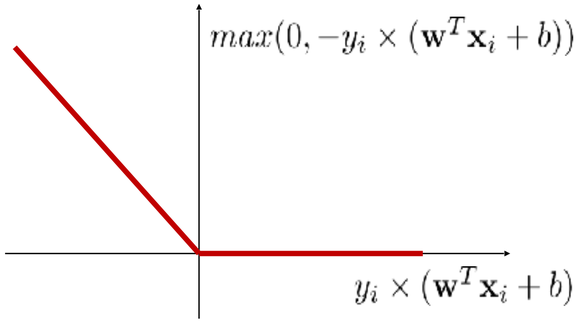
$$\mathbf{w}_{t+1} = \mathbf{w}_t + \lambda(target_i - output_i)\mathbf{x}_i$$
$$b_{t+1} = b_t + \lambda(target_i - output_i)$$

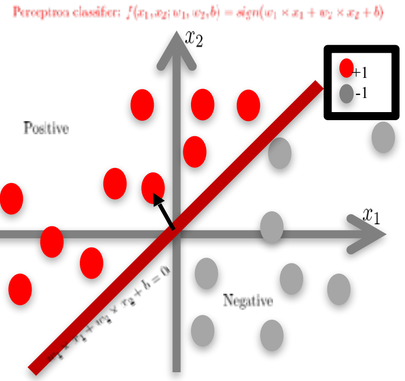
- Step 5: Go to the next data point. If you have gone through them all, you have found the solution!

Swapping loss functions

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

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





- Perceptron is a linear classifier.
- It replaces the 0/1 loss by a relaxed loss.
- It updates the model parameters (**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

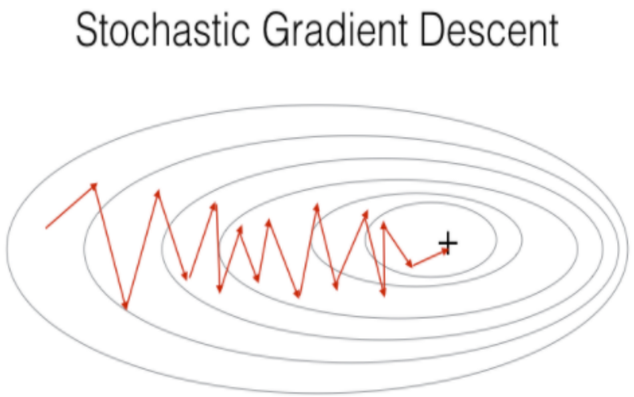
Perceptron:

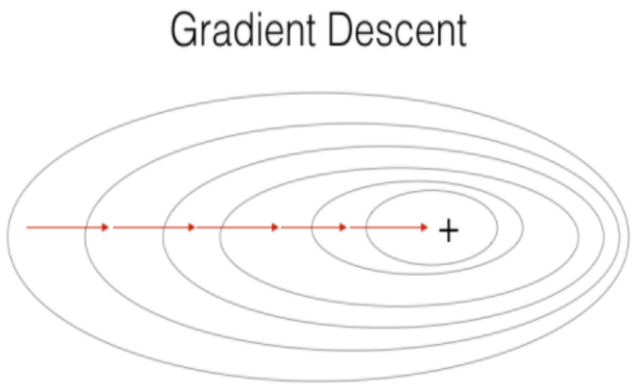
Note that the learning process is not strictly gradient descent.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + (target_i - output_i)\mathbf{x}_i$$
$$b_{t+1} = b_t + (target_i - output_i)$$

It's a stochastic gradient descent algorithm!

Stochastic vs Batch Gradient Descent

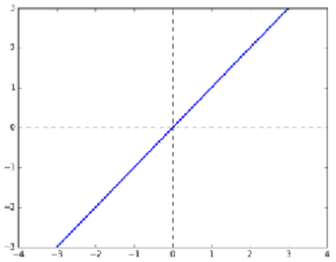




- Random order of updates means you can take different paths with same data
- Online approximation to batch
- Subsample training set for faster progress

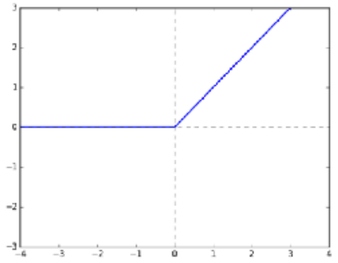
- Always the same path for the same data, order irrelevant
- Offline

Some activation functions:



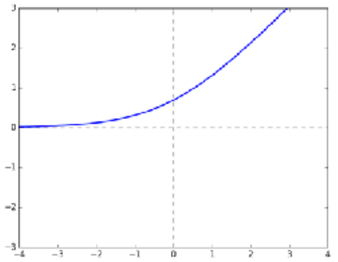
Linear

$y = z$



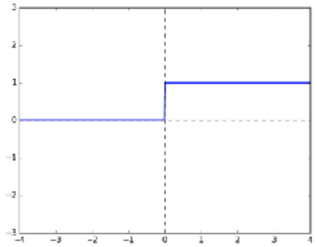
Rectified Linear Unit (ReLU)

$y = \max(0, z)$



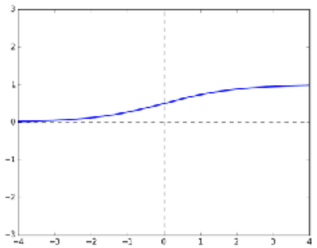
Soft ReLU

$y = \log 1 + e^z$



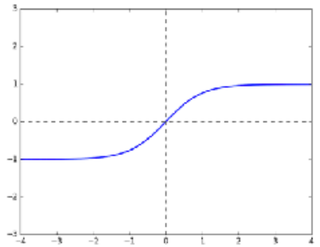
Hard Threshold

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



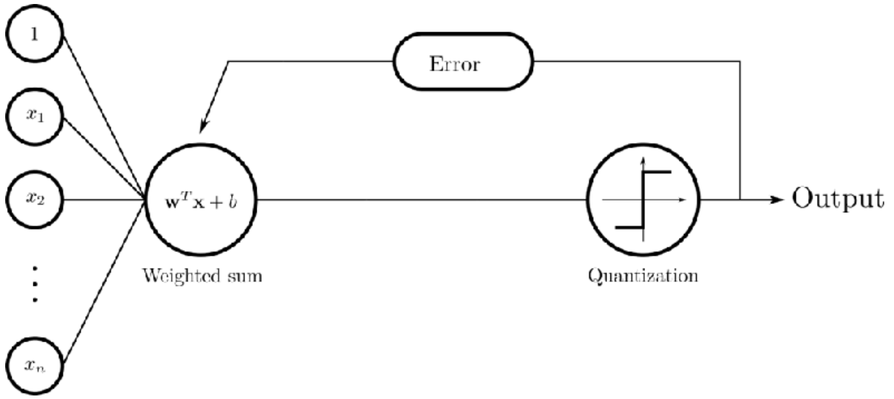
Logistic

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



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

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

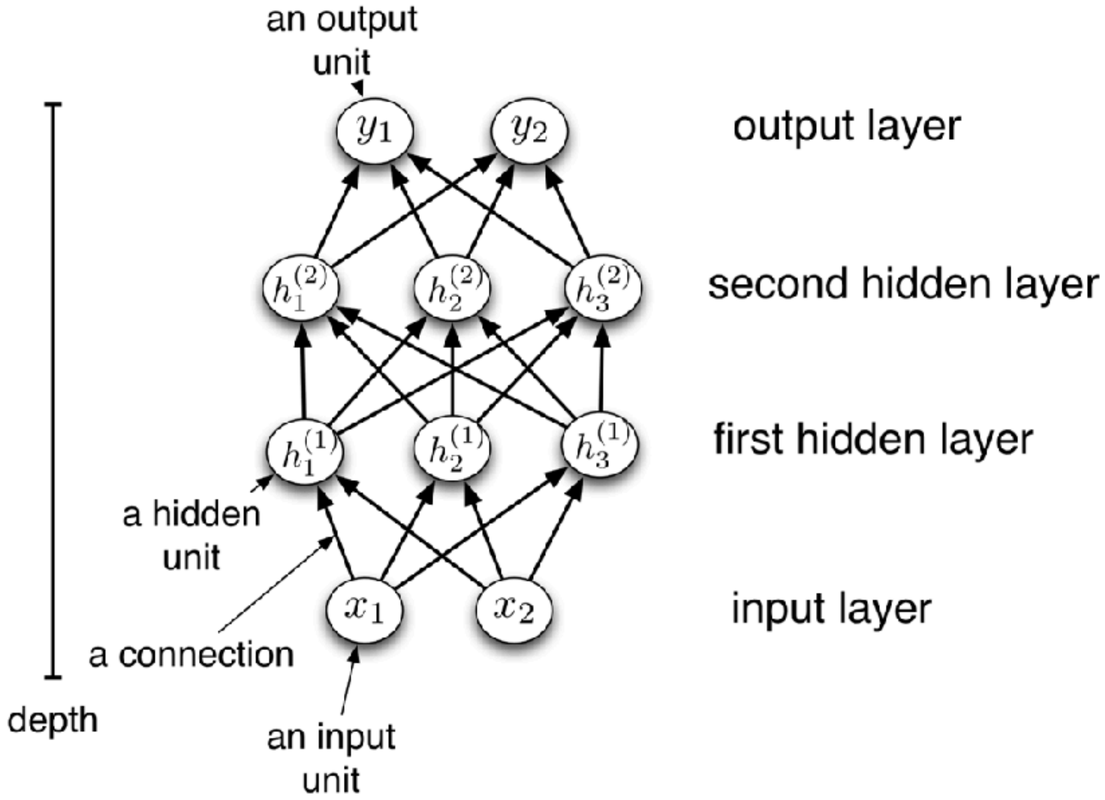
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)})$$

$$\dots$$

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

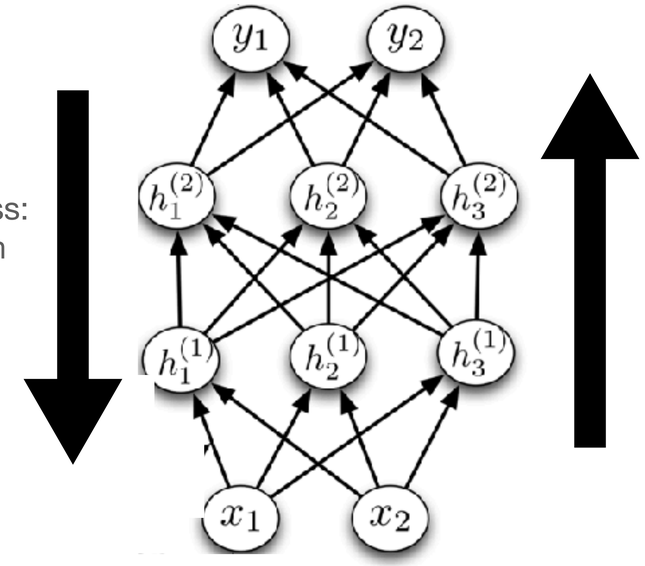
- Or more simply it is a modular composition of the functions of each layer

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



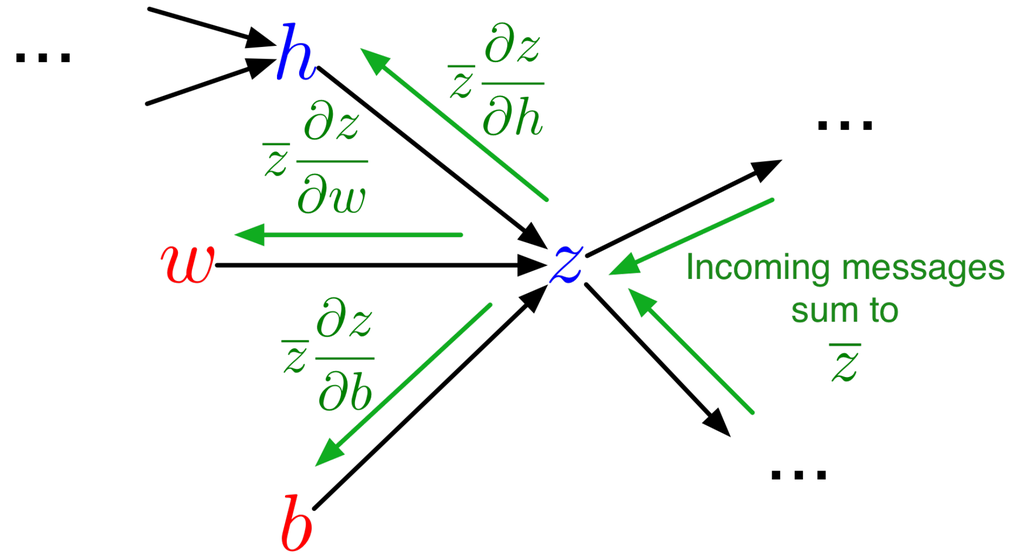
Backpropogation

- 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

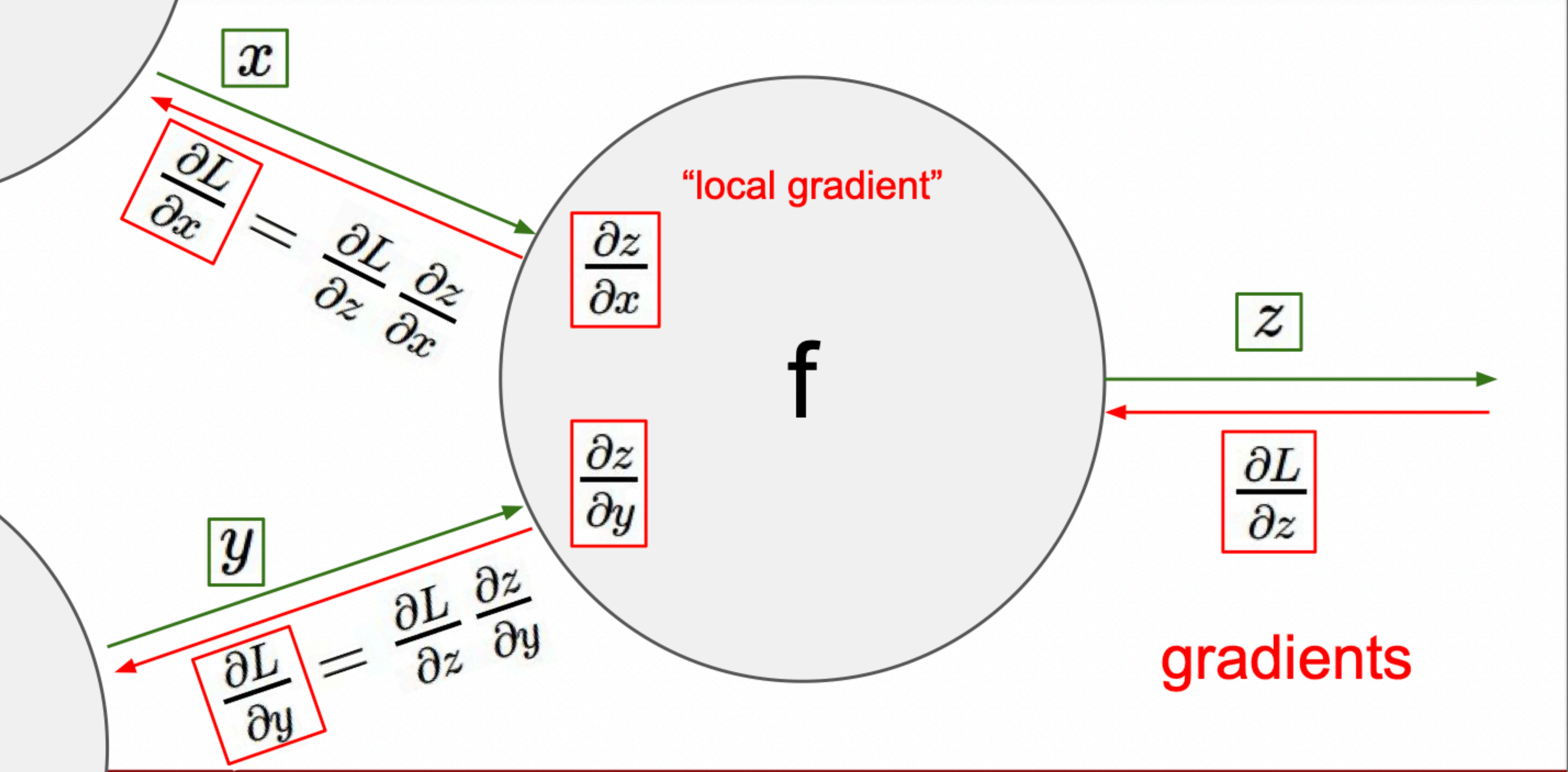


Backpropagation

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.


x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients