CS546: Machine Learning in NLP (Spring 2018)

http://courses.engr.illinois.edu/cs546/

Lecture 5
Neural models for NLP

Julia Hockenmaier

juliahmr@illinois.edu

3324 Siebel Center

Office hours: Tue/Thu 2pm-3pm

Logistics

Schedule

Week 1—Week 4: Lectures

Paper presentations: Lectures 9-28

1. Word embeddings
2. Language models
3. More on RNNs for NLP
4. CNNs for NLP
5. Multitask learning for NLP
6. Syntactic parsing
7. Information extraction
8. Semantic parsing
9. Coreference resolution
10. Machine translation I
11. Machine translation II
12. Generation
13. Discourse
14. Dialog I
15. Dialog II
16. Multimodal NLP
17. Question answering
18. Entailment recognition
19. Reading comprehension
20. Knowledge graph modeling

Machine learning fundamentals

Learning scenarios

Supervised learning:

Learning to predict labels/structures
from correctly annotated data

Unsupervised learning:

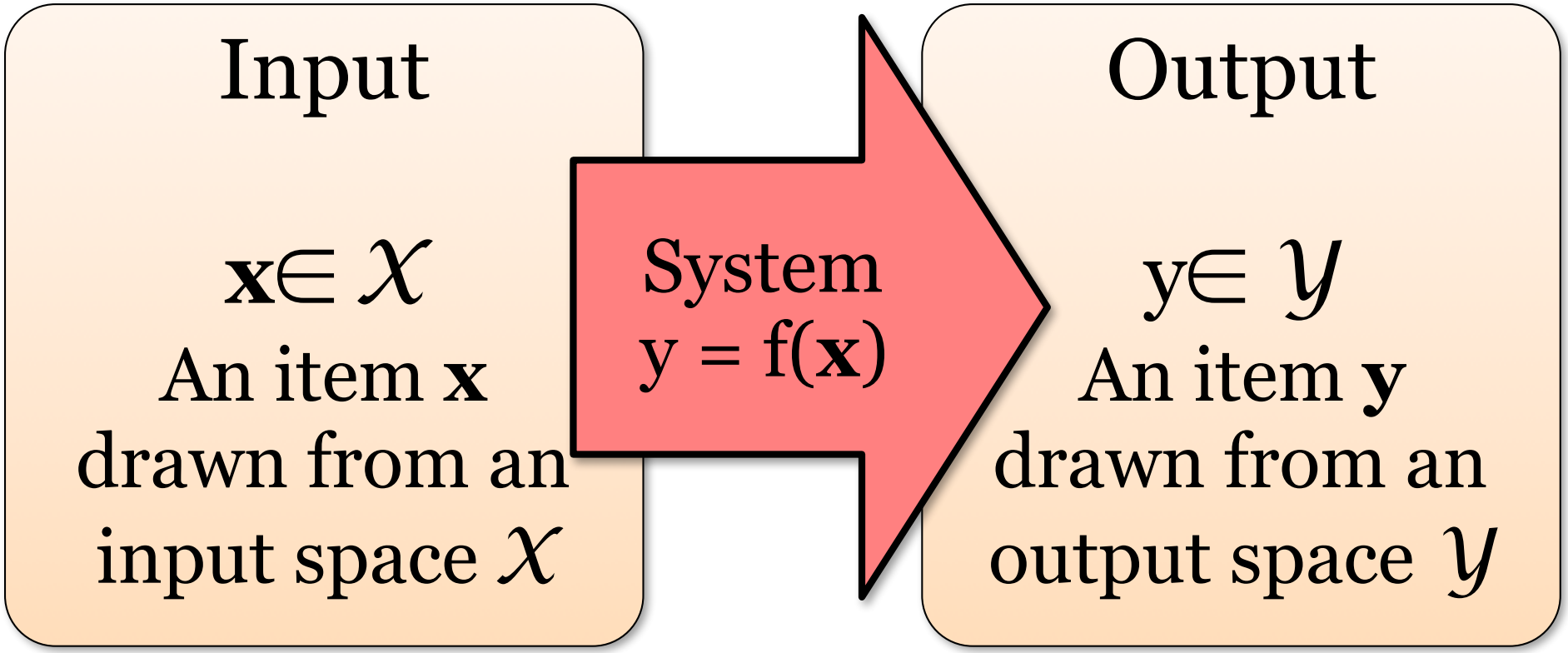
Learning to find hidden structure (e.g. clusters) in
[unannotated] input data

Semi-supervised learning:

Learning to predict labels/structures from (a little) annotated
and (a lot of) unannotated data

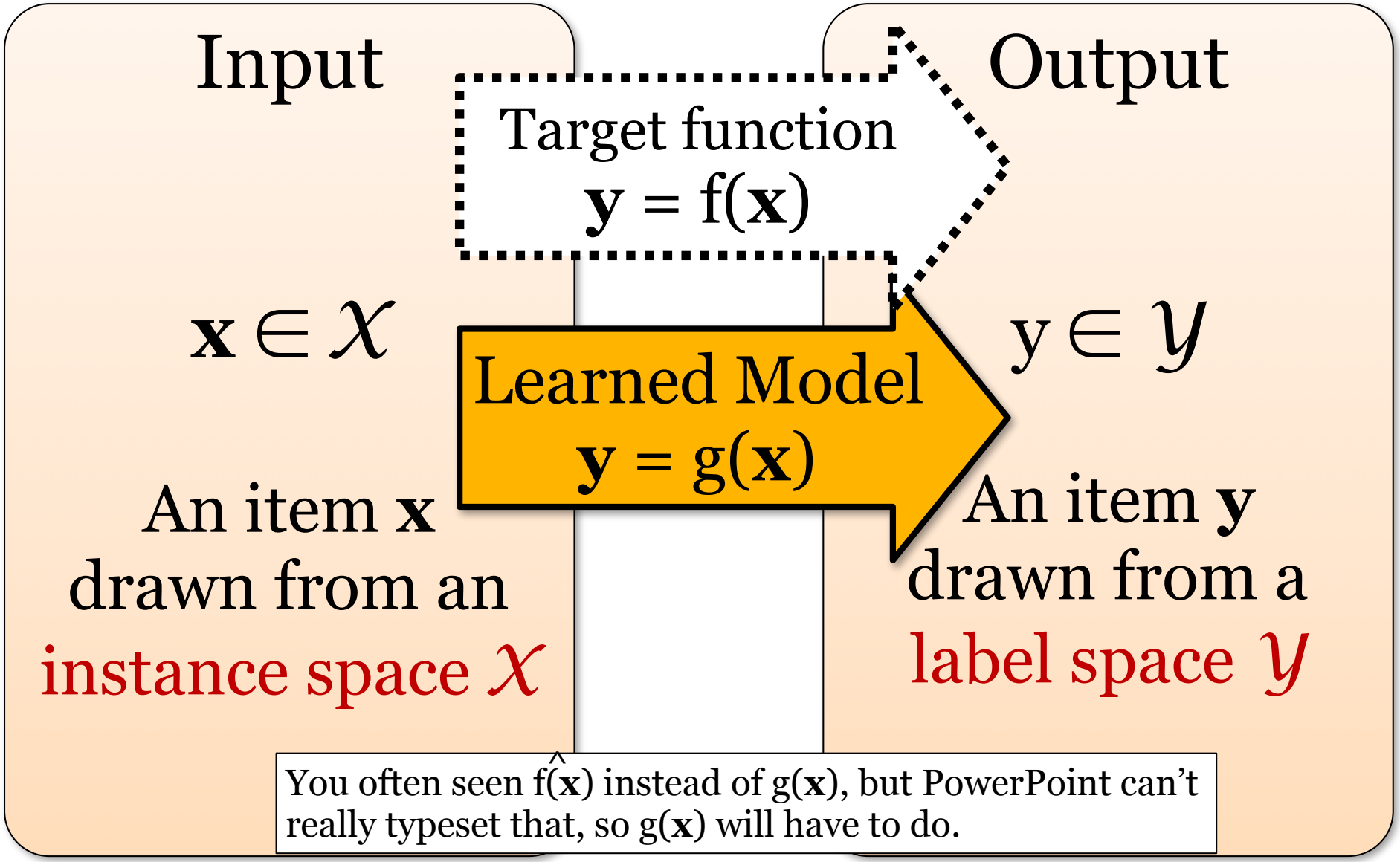
Reinforcement learning:

Learning to act through feedback for actions
(rewards/punishments) from the environment



In (supervised) machine learning, we deal with systems whose f(**x**) is learned from (labeled) examples.

Supervised learning



Supervised learning

Regression: $\mathcal{Y}$ is continuous

Classification: $\mathcal{Y}$ is discrete (and finite)

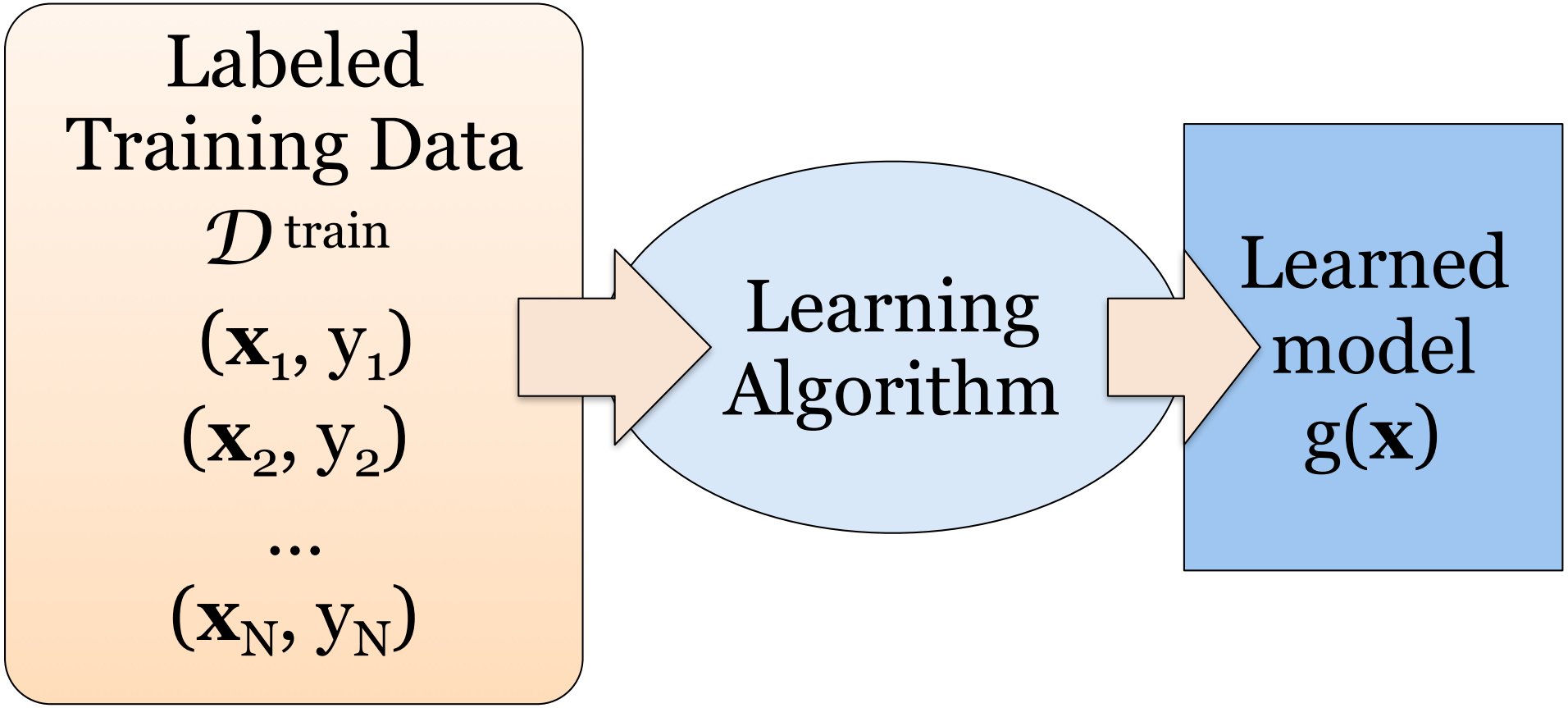
Binary classification: $\mathcal{Y} = \{0,1\}$ or $\{+1, -1\}$

Multiclass classification: $\mathcal{Y} = \{1,\ldots,K\}$ (with $K>2$)

Structured prediction: $\mathcal{Y}$ consists of structured objects

$\mathcal{Y}$ often has some sort of compositional structure and may be infinite

Supervised learning: Training



Give the learner examples in $\mathcal{D}$ train

The learner returns a model g(**x**)

Supervised learning: Testing

Labeled
Test Data
$\mathcal{D}$ test

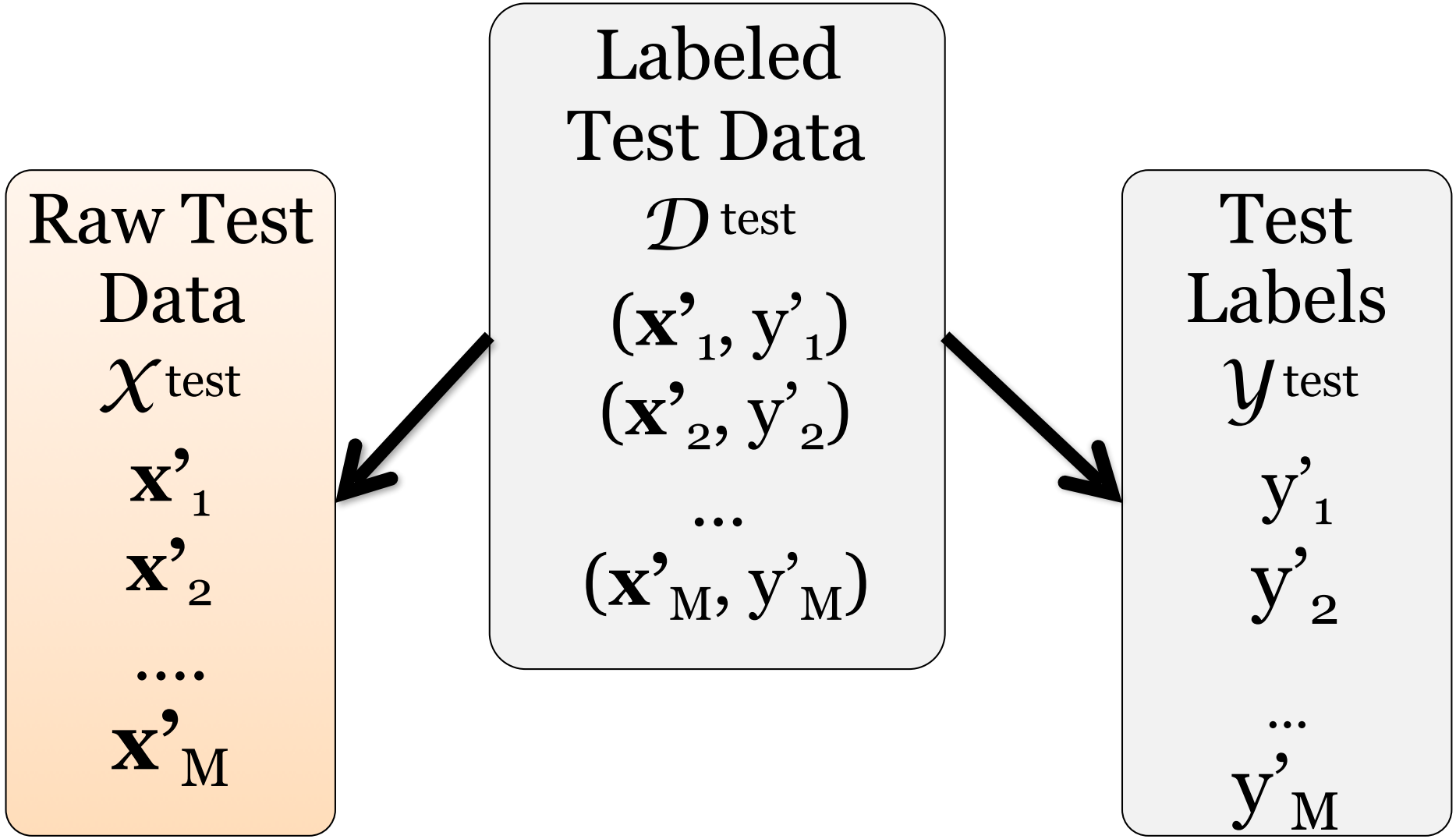
$(\mathbf{x'}_1, y'_1)$
$(\mathbf{x'}_2, y'_2)$

…

$(\mathbf{x'}_M, y'_M)$

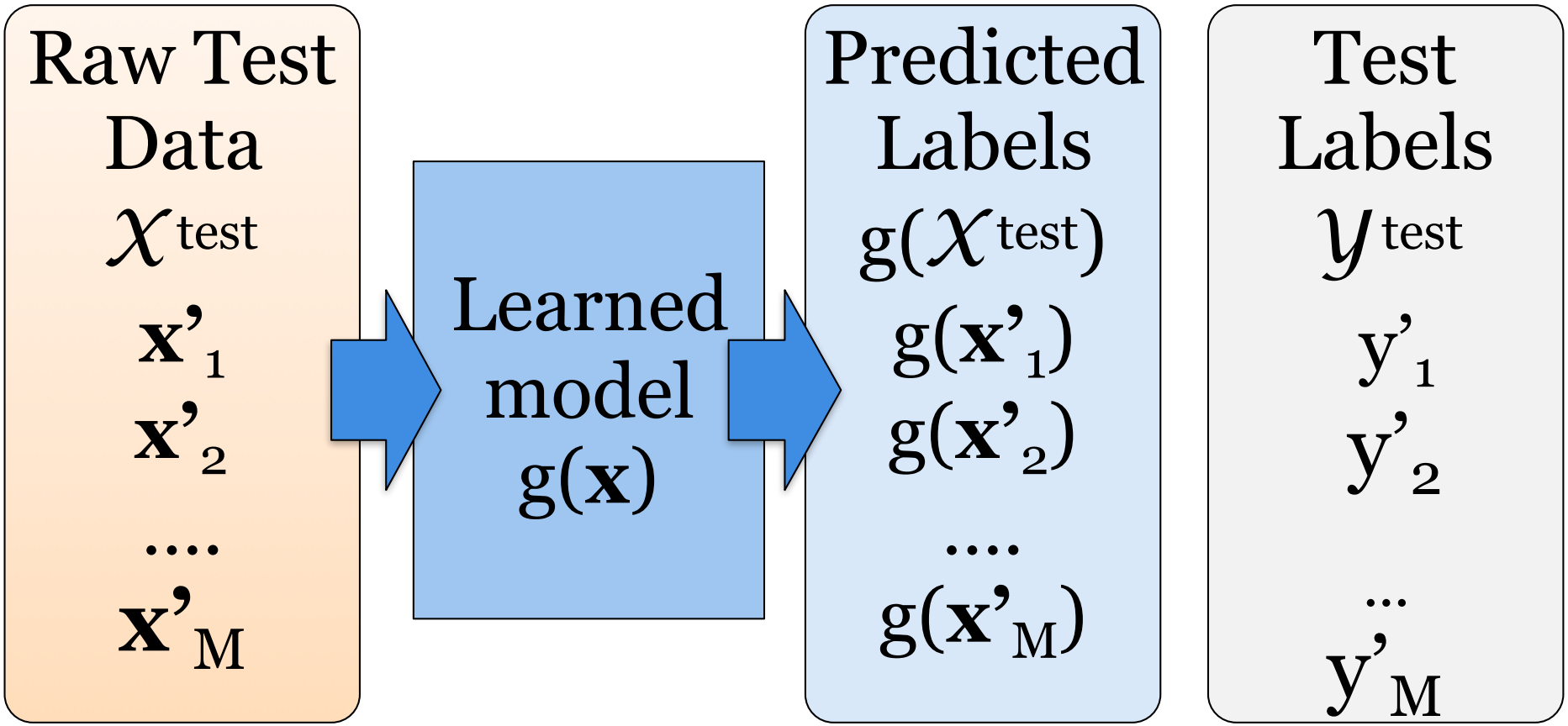
Reserve some labeled data for testing

Supervised learning: Testing



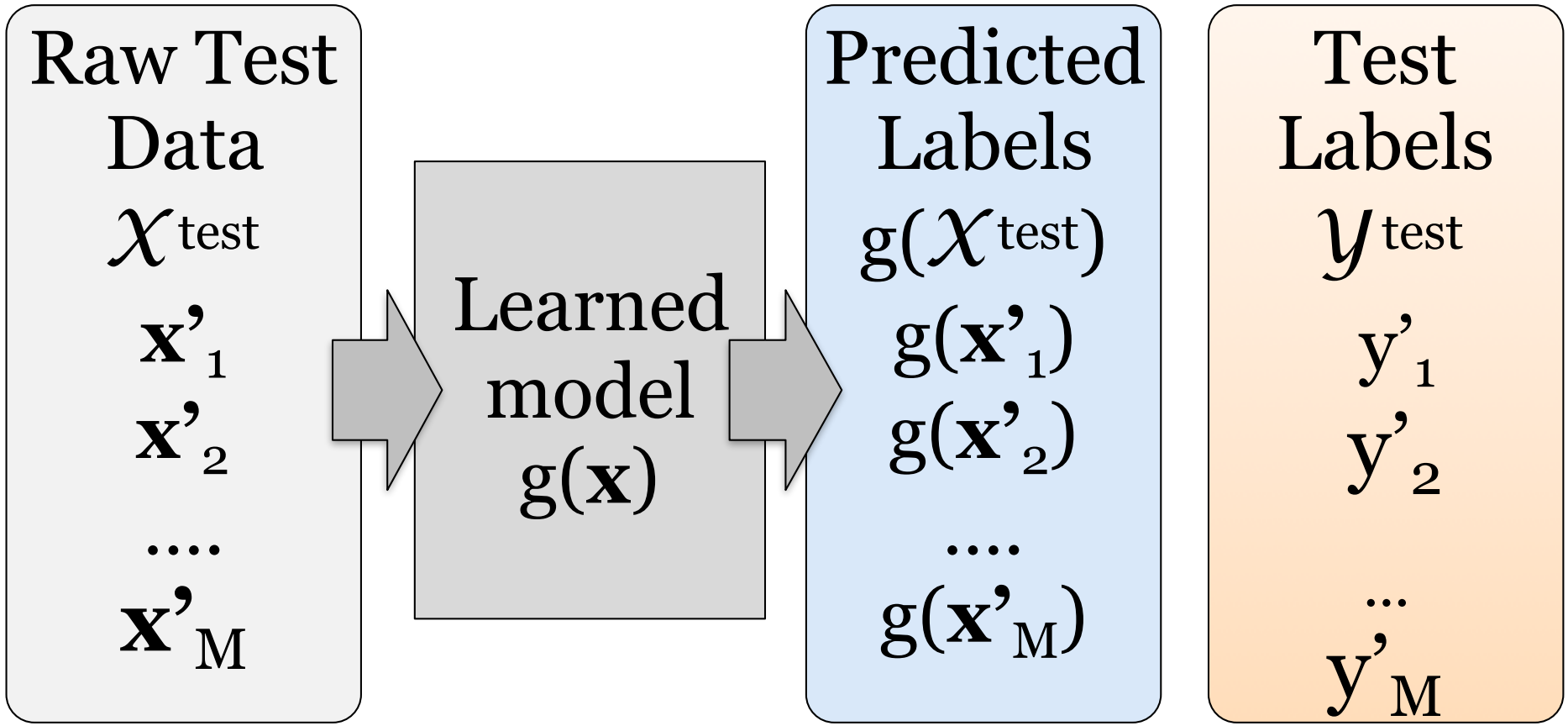
Supervised learning: Testing

Apply the model to the raw test data



Supervised learning: Testing

Evaluate the model by comparing the predicted labels against the test labels



Design decisions

What **data** do you use to train/test your system?

Do you have enough training data? How noisy is it?

What **evaluation metrics** do you use to test your system?

Do they correlate with what you want to measure?

What **features** do you use to represent your data $\mathcal{X}$?

Feature engineering used to be really important

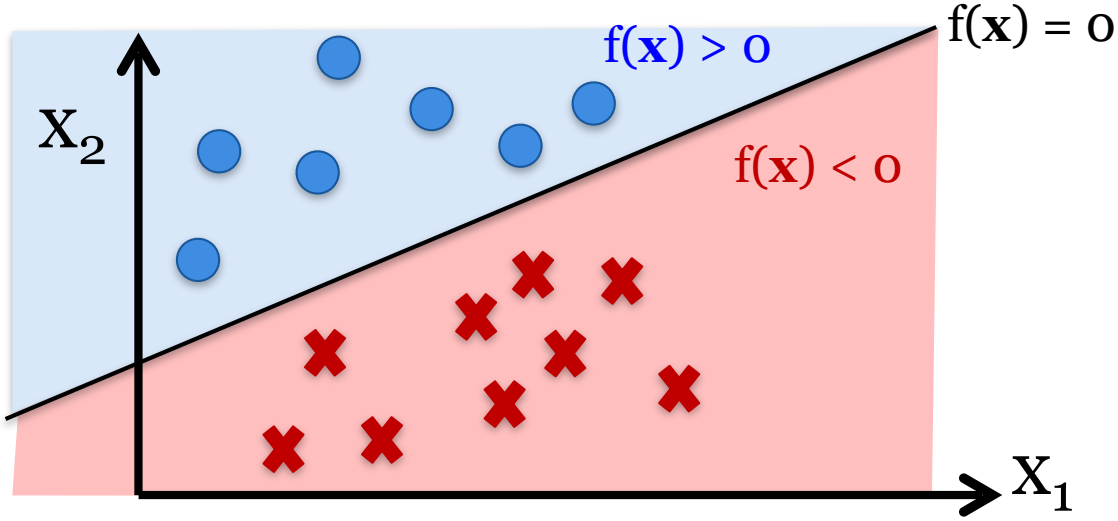
What **kind of a model** do you want to use?

What network architecture do you want to use?

What **learning algorithm** do you use to train your system?

How do you set the hyperparameters of the algorithm?

Linear classifiers: $f(\mathbf{x}) = w_0 + \mathbf{wx}$



Linear classifiers are defined over vector spaces

Every hypothesis f(**x**) is a hyperplane:

$$f(\mathbf{x}) = w_0 + \mathbf{wx}$$

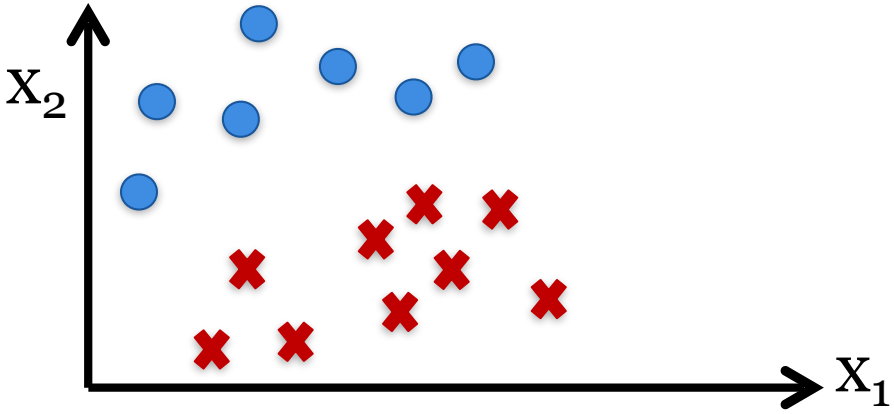
f(**x**) is also called the decision boundary

– Assign $\hat{y} = 1$ to all **x** where $f(\mathbf{x}) > 0$

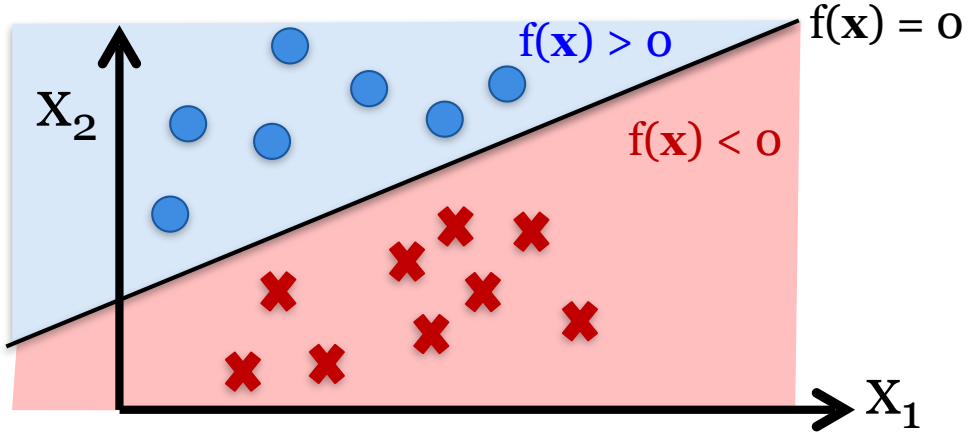
– Assign $\hat{y} = -1$ to all **x** where $f(\mathbf{x}) < 0$

$$\hat{y} = \text{sgn}(f(\mathbf{x}))$$

Learning a linear classifier

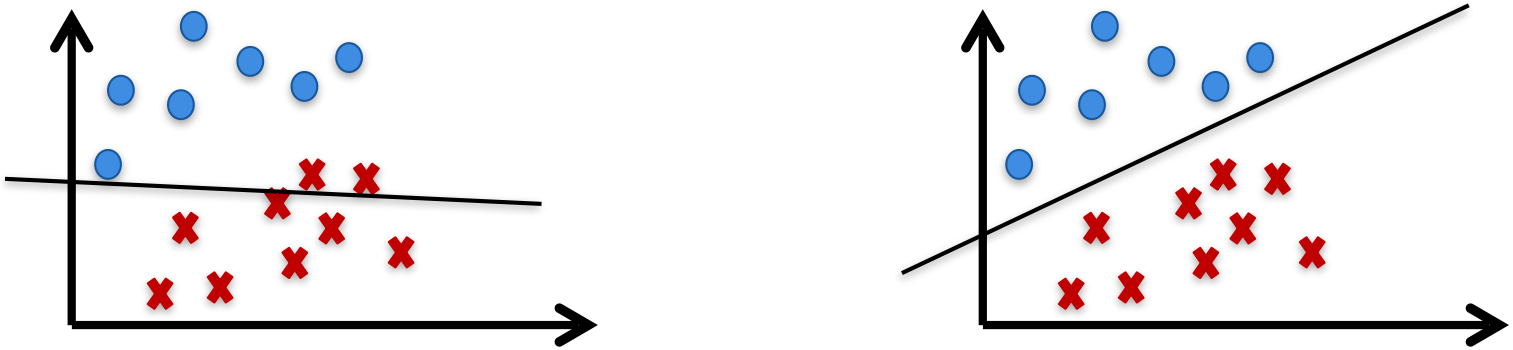


Input: Labeled training data
$\mathcal{D} = \{(\mathbf{x}^1, y^1),\ldots,(\mathbf{x}^D, y^D)\}$
plotted in the sample space $\mathcal{X} = \mathbf{R}^2$
with ●: $y^i = +1$, ✖: $y^i = -1$



Output: A decision boundary $f(\mathbf{x}) = 0$ that separates the training data
$y^i \cdot f(\mathbf{x}^i) > 0$

Which model should we pick?

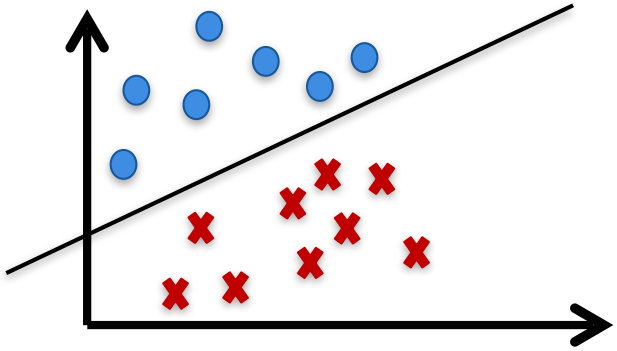


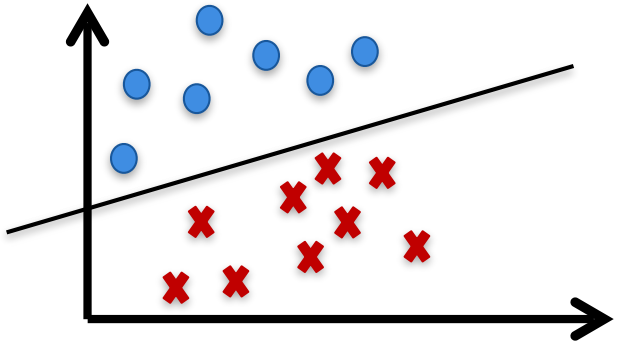
We need a metric (aka an objective function)

We would like to minimize the probability of misclassifying *unseen* examples, but we can't measure that probability.

Instead: minimize the number of misclassified training examples

Which model should we pick?

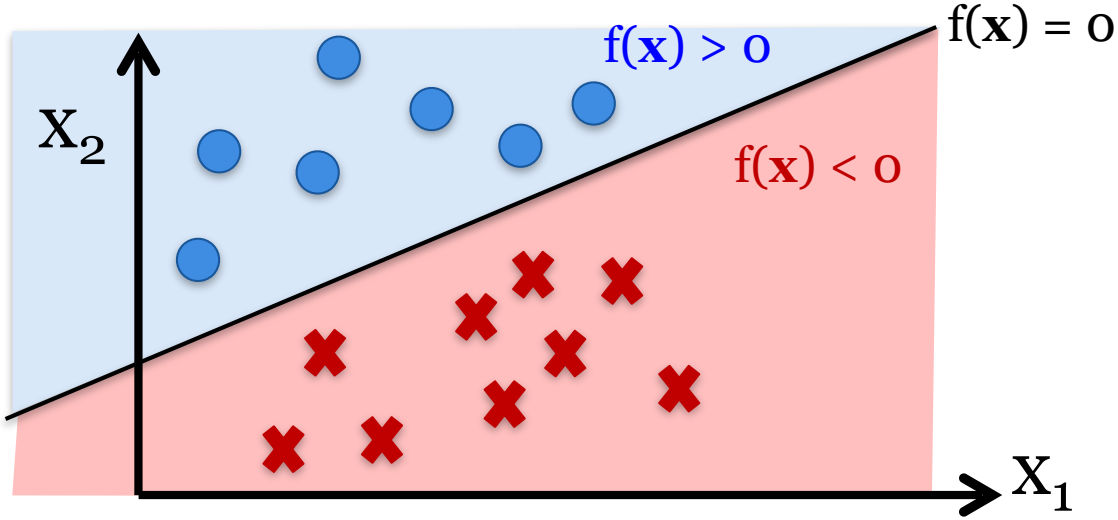




We need a more specific metric:
There may be many models that are consistent with the training data.

Loss functions provide such metrics.

$y \cdot f(\mathbf{x}) > 0$: Correct classification



An example ($\mathbf{x}$, y) is correctly classified by $f(\mathbf{x})$ if and only if $y \cdot f(\mathbf{x}) > 0$:

Case 1 ($y = +1 = \hat{y}$): $f(\mathbf{x}) > 0 \Rightarrow y \cdot f(\mathbf{x}) > 0$

Case 2 ($y = -1 = \hat{y}$): $f(\mathbf{x}) < 0 \Rightarrow y \cdot f(\mathbf{x}) > 0$

Case 3 ($y = +1 \neq \hat{y} = -1$): $f(\mathbf{x}) > 0 \Rightarrow y \cdot f(\mathbf{x}) < 0$

Case 4 ($y = -1 \neq \hat{y} = +1$): $f(\mathbf{x}) < 0 \Rightarrow y \cdot f(\mathbf{x}) < 0$

Loss functions for classification

Loss = What penalty do we incur if we misclassify $\mathbf{x}$?

$L(y, f(\mathbf{x}))$ is the loss (aka cost) of classifier f on example $\mathbf{x}$ when the true label of $\mathbf{x}$ is y.

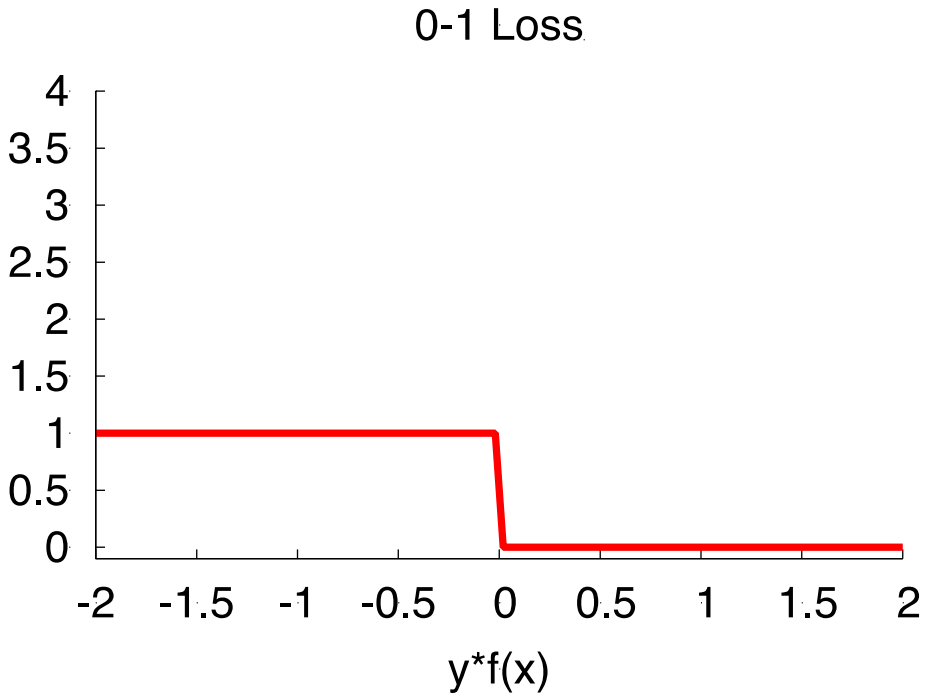
We assign label $\hat{y} = \text{sgn}(f(\mathbf{x}))$ to $\mathbf{x}$

Plots of $L(y, f(\mathbf{x}))$: x-axis is typically $y \cdot f(\mathbf{x})$

Today: 0-1 loss and square loss

(more loss functions later)

0-1 Loss

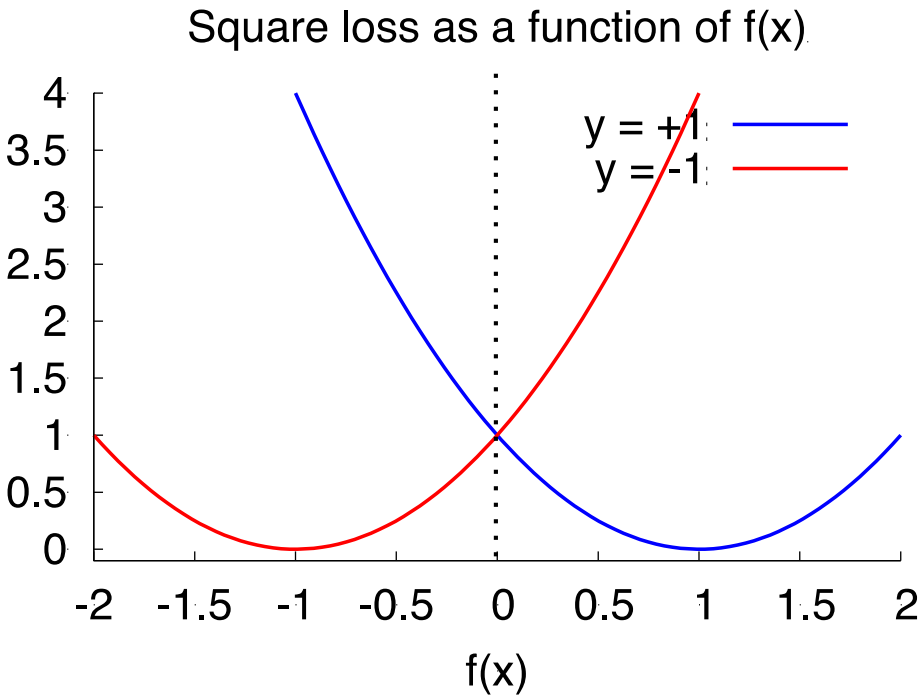


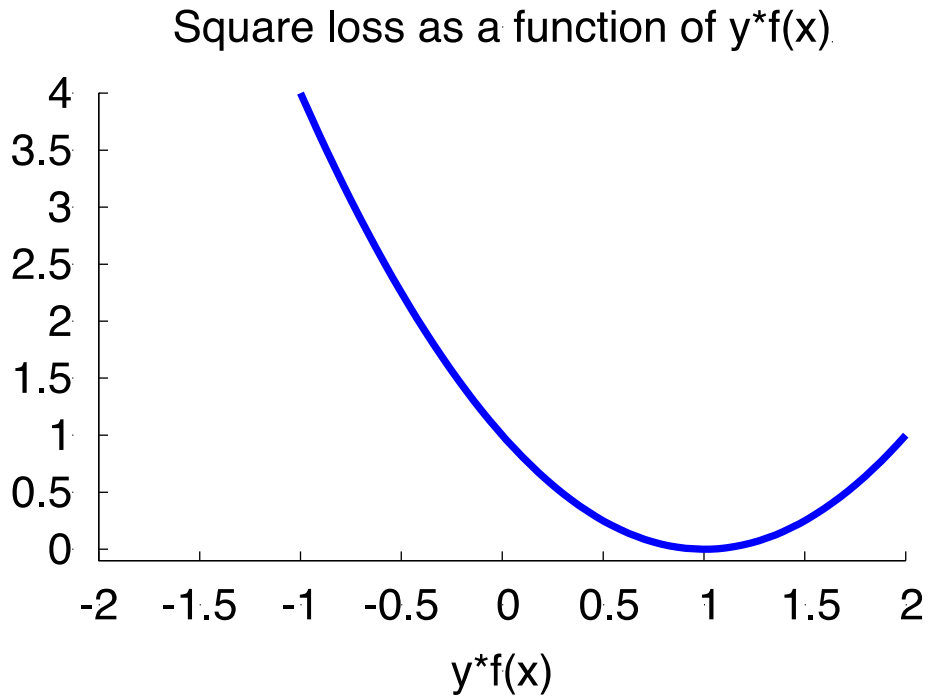
$$L(y, f(\mathbf{x})) = 0 \quad \text{iff} \quad y = \hat{y}$$
$$= 1 \quad \text{iff} \quad y \neq \hat{y}$$

$L(\ y \cdot f(\mathbf{x})\ ) = 0$ iff $y \cdot f(\mathbf{x}) > 0$ (correctly classified)

$= 1$ iff $y \cdot f(\mathbf{x}) < 0$ (misclassified)

Square Loss: $(y - f(\mathbf{x}))^2$

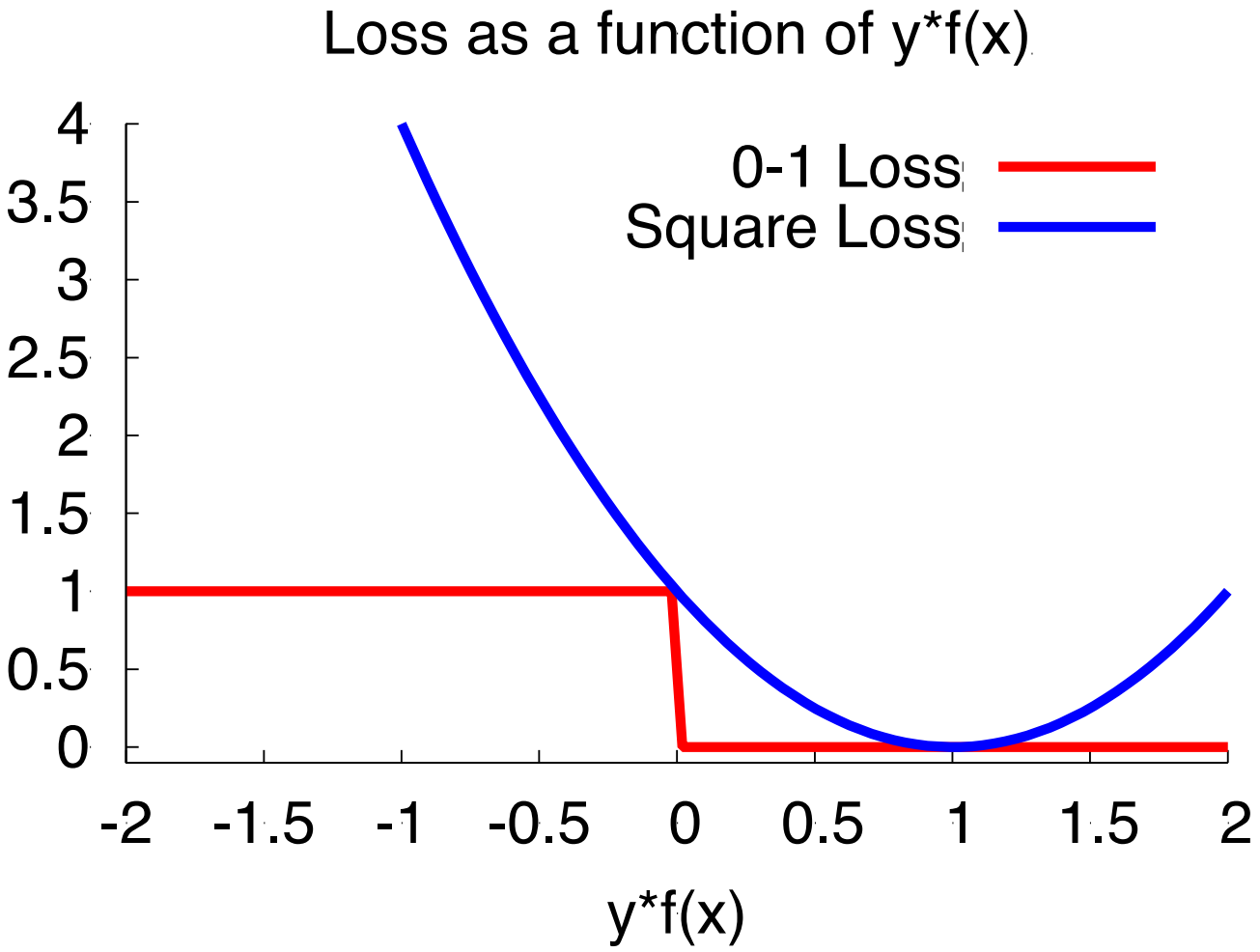




$$L(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$$

Note: $L(-1, f(\mathbf{x})) = (-1 - f(\mathbf{x}))^2 = (1 + f(\mathbf{x}))^2 = L(1, -f(\mathbf{x}))$
(the loss when y=-1 [red] is the mirror of the loss when y=+1 [blue])

The square loss is a **convex upper bound** on 0-1 Loss



Batch learning: Gradient Descent for Least Mean Squares (LMS)

Gradient Descent

Iterative batch learning algorithm:

– Learner updates the hypothesis based on the entire training data

– Learner has to go multiple times over the training data

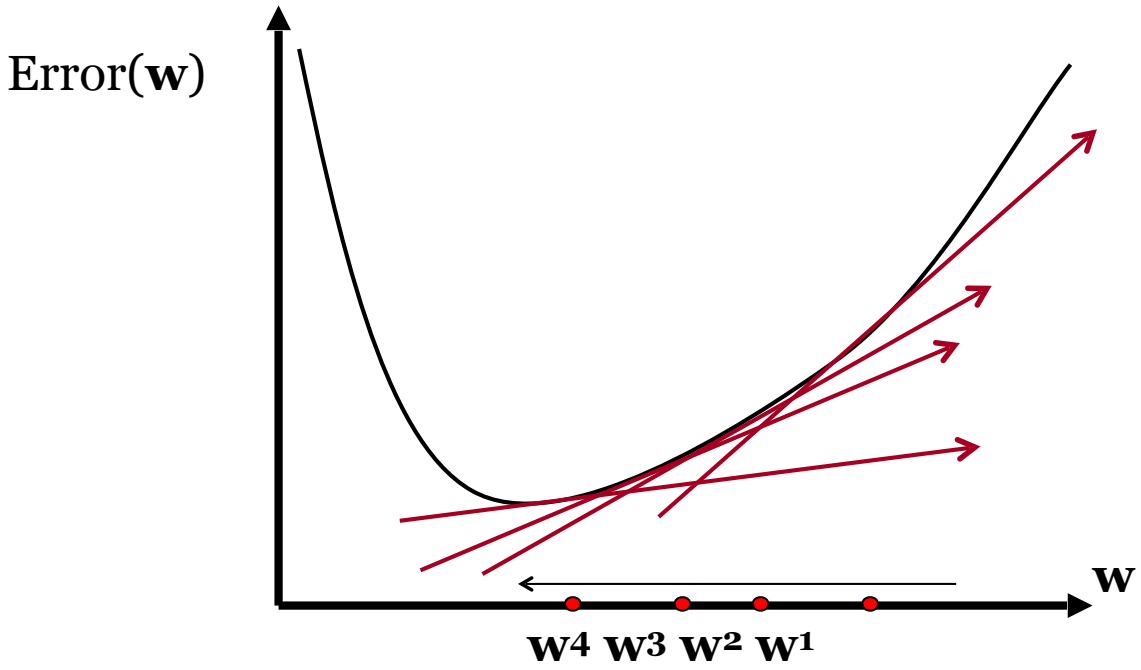
Goal: Minimize training error/loss

– At each step: move **w** in the direction of *steepest descent* along the error/loss surface

Gradient Descent

Error($\mathbf{w}$): Error of $\mathbf{w}$ on training data

$\mathbf{w}^i$: Weight at iteration i



Least Mean Square Error

$$\mathrm{Err}(\mathbf{w}) = \frac{1}{2}\sum_{d \in D}(y_d - \hat{y}_d)^2$$

LMS Error:
Sum of square loss over all training items (multiplied by 0.5 for convenience)

D is fixed, so no need to divide by its size

Goal of learning: Find $\mathbf{w}^* = \mathrm{argmin}(\mathrm{Err}(\mathbf{w}))$

Iterative batch learning

Initialization:

Initialize $\mathbf{w}^0$ (the initial weight vector)

For each iteration:

for i = 0…T:

Determine by how much to change w based on the entire data set $\mathcal{D}$

$\Delta\mathbf{w}$ = computeDelta($\mathcal{D}$, $\mathbf{w}^i$)

Update w:

$\mathbf{w}^{i+1}$ = update($\mathbf{w}^i$, $\Delta\mathbf{w}$)

Gradient Descent: Update

1. Compute $\nabla Err(\mathbf{w}^i)$, the gradient of the training error at $\mathbf{w}^i$

   This requires going over the entire training data

$$\nabla Err(\mathbf{w}) = \left(\frac{\partial Err(\mathbf{w})}{\partial w_0}, \frac{\partial Err(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial Err(\mathbf{w})}{\partial w_N} \right)^T$$

2. Update $\mathbf{w}$:

   $\mathbf{w}^{i+1} = \mathbf{w}^i - \alpha \nabla Err(\mathbf{w}^i)$

   $\alpha > 0$ is the learning rate

What's a gradient?

$$\nabla \mathrm{Err}(\mathbf{w}) = \left(\frac{\partial \mathrm{Err}(\mathbf{w})}{\partial w_0}, \frac{\partial \mathrm{Err}(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial \mathrm{Err}(\mathbf{w})}{\partial w_N} \right)^T$$

The gradient is a vector of partial derivatives

It indicates the direction of steepest *increase* in Err($\mathbf{w}$)

Hence the *minus* in the upgrade rule: $\mathbf{w}^i - \alpha \nabla \mathrm{Err}(\mathbf{w}^i)$

Computing $\nabla Err(\mathbf{w}^i)$

$$\frac{\partial Err(\mathbf{w})}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(y_d - f(\mathbf{x}_d))^2$$

$$= \frac{1}{2}\frac{\partial}{\partial w_i}\sum_{d\in D}(y_d - f(\mathbf{x}_d))^2$$

$$= \frac{1}{2}\sum_{d\in D} 2(y_d - f(\mathbf{x}_d))\frac{\partial}{\partial w_i}(y_d - \mathbf{w}\cdot\mathbf{x}_d)$$

$$= -\sum_{d\in D}(y_d - f(\mathbf{x}_d))x_{di}$$

$$Err(\mathbf{w}^{(j)}) = \frac{1}{2}\sum_{d\in D}(y_d - f(\mathbf{x})_d)^2$$

Gradient descent (batch)

Initialize $\mathbf{w}^0$ randomly

for i = 0…T:

  $\Delta\mathbf{w}$ = (0, …., 0)

  for every training item d = 1…D:

    $f(\mathbf{x_d}) = \mathbf{w}^i \cdot \mathbf{x_d}$

    for every component of $\mathbf{w}$ j = 0…N:

      $\Delta w_j \mathrel{+}= \alpha(y_d - f(\mathbf{x_d})) \cdot x_{dj}$

  $\mathbf{w}^{i+1} = \mathbf{w}^i + \Delta\mathbf{w}$

  return $\mathbf{w}^{i+1}$ when it has converged

The batch update rule for each component of **w**

$$\Delta w_i = \alpha \sum_{d=1}^{D} (y_d - \mathbf{w}^j \cdot \mathbf{x}_d) x_{di}$$

Implementing gradient descent:
As you go through the training data, you can just accumulate the change in each component w_i of **w**

Learning rate and convergence

The learning rate is also called the *step size*.

More sophisticated algorithms (Conjugate Gradient) choose the step size automatically and converge faster.

– When the learning rate is too small, convergence is very slow
– When the learning rate is too large, we may oscillate (overshoot the global minimum)
– You have to experiment to find the right learning rate for your task

Online learning with Stochastic Gradient Descent

Stochastic Gradient Descent

Online learning algorithm:

– Learner updates the hypothesis with each training example

– No assumption that we will see the same training examples again

– Like batch gradient descent, except we update after seeing each example

Why online learning?

Too much training data:

– Can't afford to iterate over everything

Streaming scenario:

– New data will keep coming

– You can't assume you have seen everything

– Useful also for adaptation (e.g. user-specific spam detectors)

Stochastic Gradient descent (online)

Initialize $\mathbf{w^o}$ randomly

for m = o…M:

$f(\mathbf{x_m}) = \mathbf{w^i} \cdot \mathbf{x_m}$

$\Delta w_j = \alpha(y_d - f(\mathbf{x_m})) \cdot x_{mj}$

$\mathbf{w^{i+1}} = \mathbf{w^i} + \Delta \mathbf{w}$

return $\mathbf{w^{i+1}}$ when it has converged

Online perceptron

Assumptions: class labels $y \in \{+1, -1\}$;
learning rate $\alpha > 0$

Initial weight vector $\mathbf{w}^0 := (0,\ldots,0)$

$i = 0$

for $m = 0 \ldots M$:

if $y_m \cdot f(\mathbf{x_m}) = y_m \cdot \mathbf{w}^i \cdot \mathbf{x_m} < 0$:

($\mathbf{x_m}$ is misclassified – add $\alpha \cdot y_m \cdot \mathbf{x_m}$ to $\mathbf{w}$!)

$\mathbf{w}^{i+1} := \mathbf{w}^i + \alpha \cdot y_m \cdot \mathbf{x_m}$

$i := i+1$

Perceptron rule

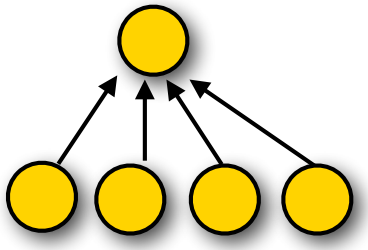
return $\mathbf{w}^{i+1}$ when all examples correctly classified

Neural Networks

What are neural nets?

Simplest variant: single-layer feedforward net

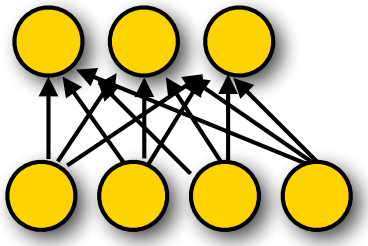
For binary classification tasks:
Single output unit
Return 1 if $y > 0.5$
Return 0 otherwise



Output unit: scalar *y*

Input layer: vector **x**

For multiclass classification tasks:
K output units (a vector)
Each output unit
y_i = class i
Return $\text{argmax}_i(y_i)$



Output layer: vector **y**

Input layer: vector **x**

Multiclass models: softmax(y_i)

Multiclass classification = predict one of K classes.
Return the class i with the highest score: $\text{argmax}_i(y_i)$

In neural networks, this is typically done by using the **softmax** function, which maps real-valued vectors in R^N into a distribution over the N outputs
For a vector $\mathbf{z} = (z_0 \ldots z_K)$: $P(i) = \text{softmax}(z_i) = \exp(z_i) \; / \; \sum_{k=0..K} \exp(z_k)$
(NB: This is just logistic regression)

Single-layer feedforward networks

Single-layer (linear) feedforward network

$y = \mathbf{wx} + b$ (binary classification)

w is a weight vector, b is a bias term (a scalar)

This is just a linear classifier (aka Perceptron)
(the output y is a linear function of the input **x**)

Single-layer non-linear feedforward networks:

Pass **wx** + b through a non-linear activation function,
e.g. $y = \tanh(\mathbf{wx} + b)$

Nonlinear activation functions

Sigmoid (logistic function): $\sigma(x) = 1/(1 + e^{-x})$

Useful for output units (probabilities) [0,1] range

Hyperbolic tangent: $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$

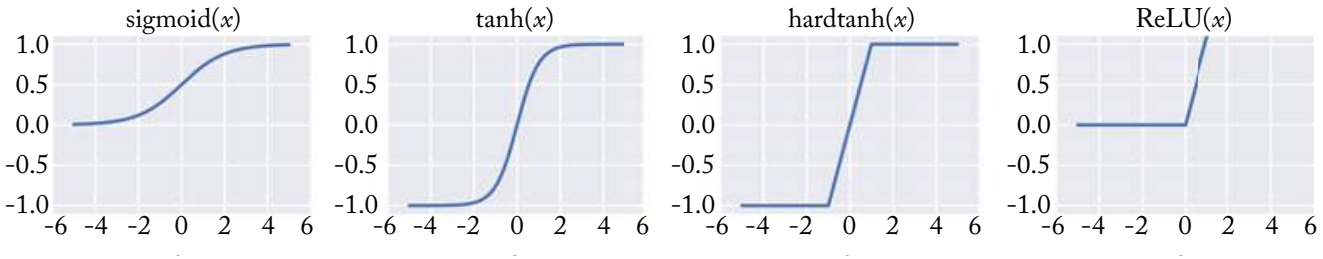
Useful for internal units: [-1,1] range

Hard tanh (approximates tanh)

$$\text{htanh}(x) = -1 \text{ for } x < -1,\ 1 \text{ for } x > 1,\ x \text{ otherwise}$$

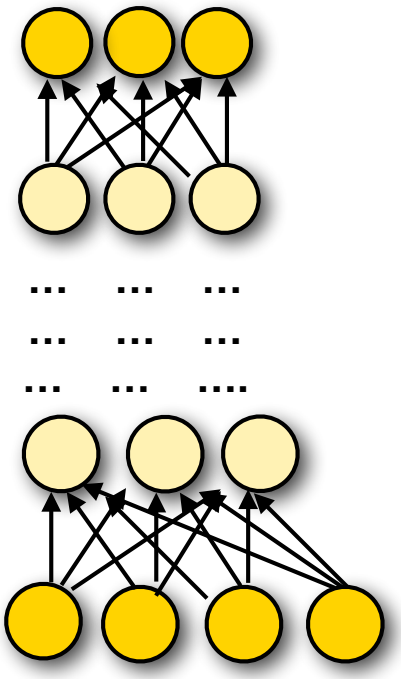
Rectified Linear Unit: $\text{ReLU}(x) = \max(0, x)$

Useful for internal units



Multi-layer feedforward networks

We can generalize this to multi-layer feedforward nets



Output layer: vector **y**

Hidden layer: vector $\mathbf{h_n}$

Hidden layer: vector $\mathbf{h_1}$

Input layer: vector **x**

Why neural approaches to NLP?

Motivation for neural approaches to NLP: Features can be brittle

Word-based features:

How do we handle unseen/rare words?

Many features are produced by other NLP systems (POS tags, dependencies, NER output, etc.)
These systems are often trained on labeled data.

- Producing labeled data can be very expensive.
- We typically don't have enough labeled data from the domain of interest.
- We might not get accurate features for our domain of interest.

Features in neural approaches

Many of the current successful neural approaches to NLP do not use traditional discrete features.

Words in the input are often represented as dense vectors (aka. word embeddings, e.g. word2vec)

- Traditional approaches: each word in the vocabulary is a separate feature. No generalization across words that have similar meanings.
- Neural approaches: Words with similar meanings have similar vectors. Models generalize across words with similar meanings

Other kinds of features (POS tags, dependencies, etc.) are often ignored.

Motivation for neural approaches to NLP: Markov assumptions

Traditional sequence models (n-gram language models, HMMs, MEMMs, CRFs) make rigid Markov assumptions (bigram/trigram/n-gram).

Recurrent neural nets (RNNs, LSTMs) can capture arbitrary-length histories without requiring more parameters.

Neural approaches to NLP

Challenges in using NNs for NLP

Our input and output variables are discrete: words, labels, structures.

NNs work best with continuous vectors.

- We typically want to learn a mapping (embedding) from discrete words (input) to dense vectors.
- We can do this with (simple) neural nets and related methods.

The input to a NN is (traditionally) a fixed-length vector. How do you represent a variable-length sequence as a vector?

- Use recurrent neural nets: read in one word at the time to predict a vector, use that vector and the next word to predict a new vector, etc.

How does NLP use NNs?

Word embeddings (word2vec, Glove, etc.)

- Train a NN to predict a word from its context (or the context from a word).
- This gives a dense vector representation of each word

Neural language models:

- Use recurrent neural networks (RNNs) to predict word sequences
- More advanced: use LSTMs (special case of RNNs)

Sequence-to-sequence (seq2seq) models:

- From machine translation: use one RNN to encode source string, and another RNN to decode this into a target string.
- Also used for automatic image captioning, etc.

Recursive neural networks:

- Used for parsing

Neural Language Models

LMs define a distribution over strings: $P(w_1 \ldots w_k)$

LMs factor $P(w_1 \ldots w_k)$ into the probability of each word:

$P(w_1 \ldots w_k) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1 w_2) \cdot \ldots \cdot P(w_k \mid w_1 \ldots w_{k-1})$

A neural LM needs to define a distribution over the V words in the vocabulary, conditioned on the preceding words.

Output layer: V units (one per word in the vocabulary) with softmax to get a distribution

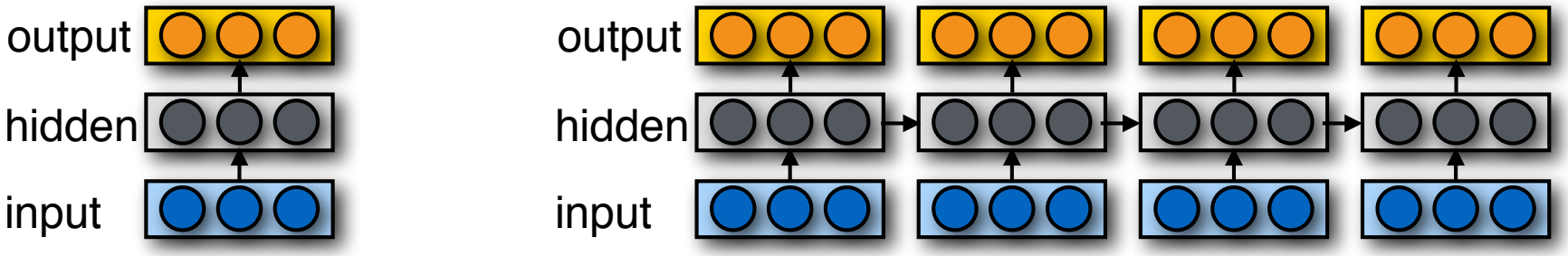
Input: Represent each preceding word by its d-dimensional embedding.

- Fixed-length history (n-gram): use preceding n–1 words
- Variable-length history: use a recurrent **neural net**

Recurrent neural networks (RNNs)

Basic RNN: Modify the standard feedforward architecture (which predicts a string $w_0 \ldots w_n$ one word at a time) such that the output of the current step (w_i) is given as additional input to the next time step (when predicting the output for w_{i+1}).

"Output" — typically (the last) hidden layer.



Feedforward Net **Recurrent Net**

Word Embeddings (e.g. word2vec)

Main idea:
If you use a feedforward network to predict the probability of words that appear in the context of (near) an input word, the hidden layer of that network provides a dense vector representation of the input word.

Words that appear in similar contexts (that have high distributional similarity) wils have very similar vector representations.

These models can be trained on large amounts of raw text (and pretrained embeddings can be downloaded)

Sequence-to-sequence (seq2seq) models

Task (e.g. machine translation):

Given one variable length sequence as input, return another variable length sequence as output

Main idea:

Use one RNN to encode the input sequence (“encoder”)

Feed the last hidden state as input to a second RNN (“decoder”) that then generates the output sequence.