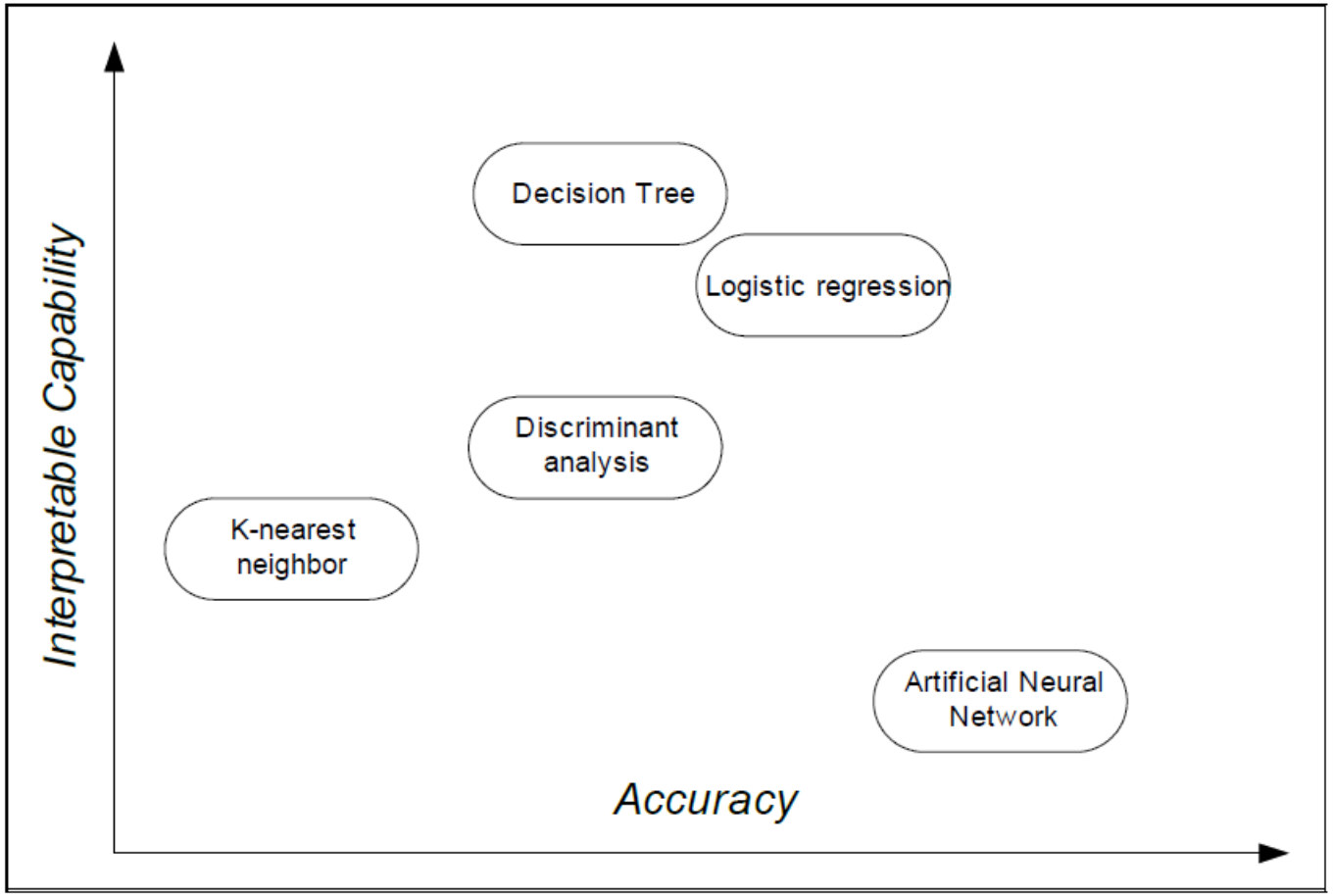
Machine Learning



Topic Covered

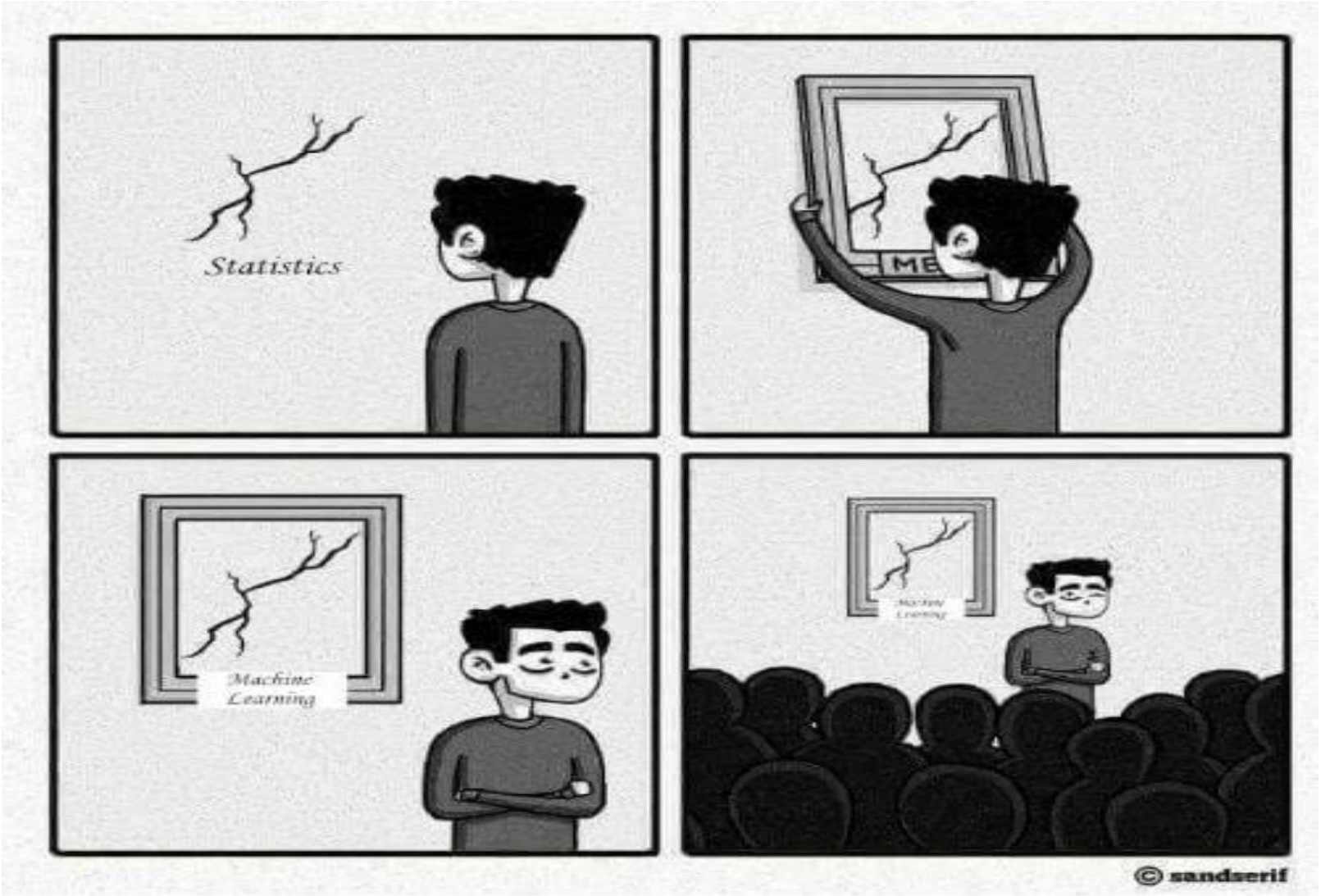
- Introduction to ML
- K-Nearest Neighbor
- Decision Tree
- Ensemble Methods
- Random Forest
- Artificial Neural Networks
- Support Vector Machines
- K-Means Clustering
- Reinforcement Learning

Supervised Learning: Tree-based Methods

Road Map

- **Basic concepts**
- K-nearest neighbor
- Decision tree induction
- Ensemble methods: Bagging and Boosting
- Summary

We start a little light….



Machine Learning

Machine Learning: An example application

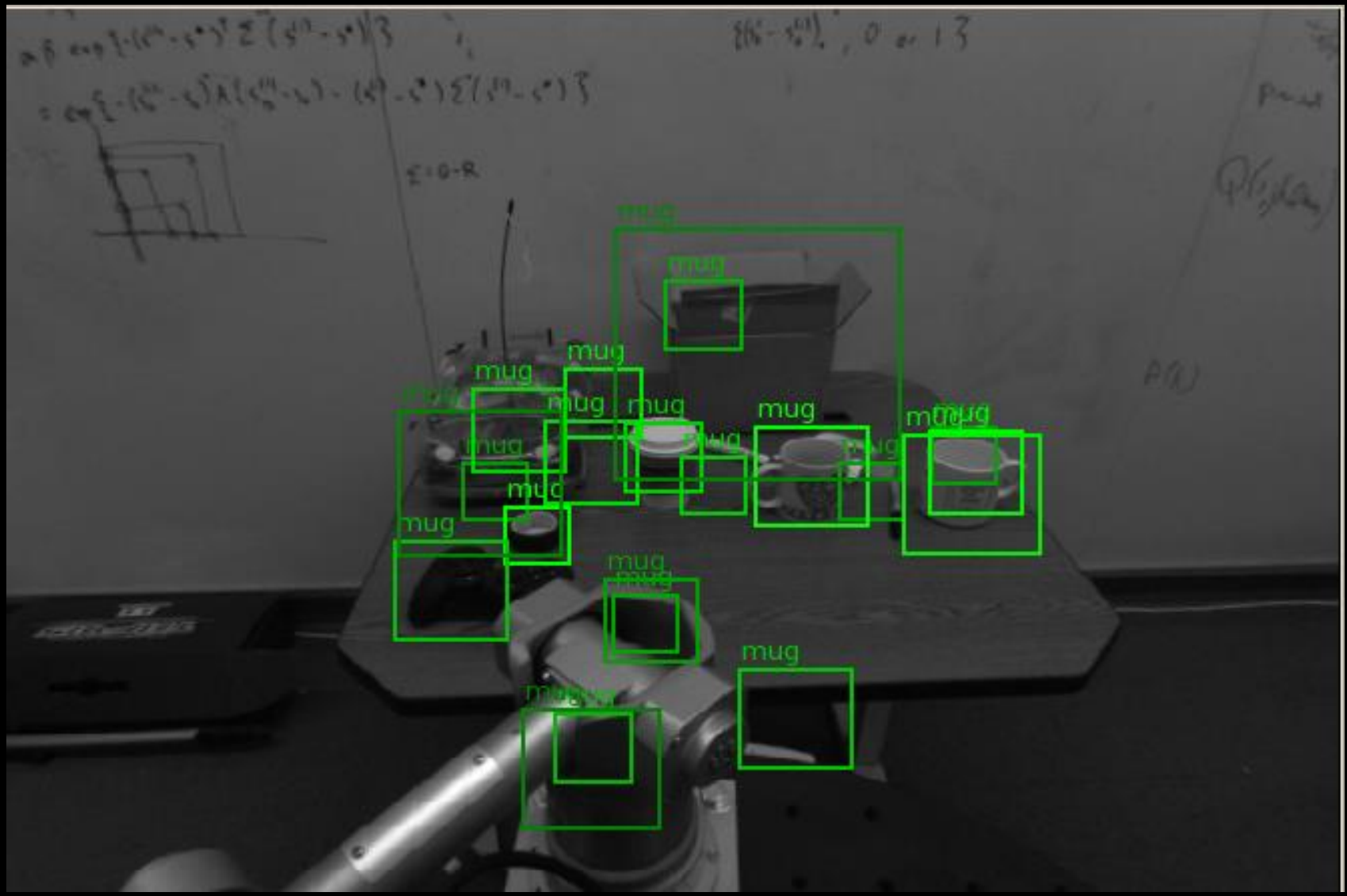
- An emergency room in a hospital measures 17 variables (e.g., blood pressure, age, etc) of newly admitted patients.
- A decision is needed: whether to put a new patient in an intensive-care unit.
- Due to the high cost of ICU, those patients who may survive less than a month are given higher priority.
- Problem: to predict high-risk patients and discriminate them from low-risk patients.

Another application

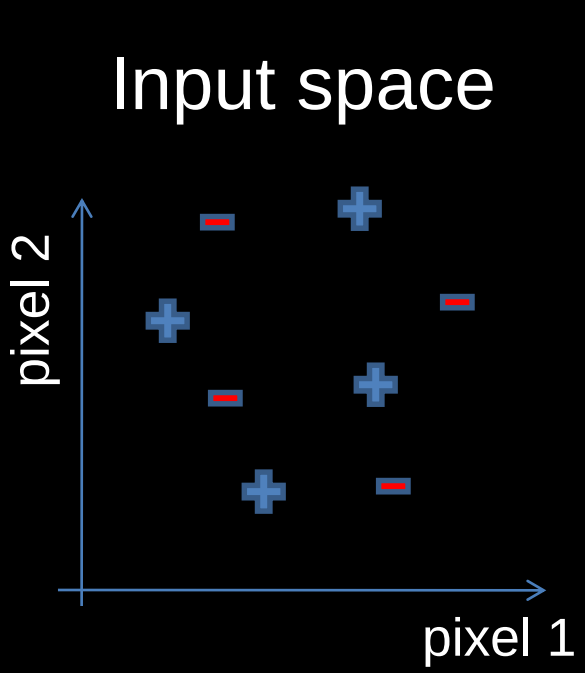
- A credit card company receives thousands of applications for new cards. Each application contains information about an applicant,
  - age
  - Marital status
  - annual salary
  - outstanding debts
  - credit rating
  - etc.
- Problem: to decide whether an application should approved, or to classify applications into two categories, approved and not approved.

Another Example

Computer vision is hard



Machine learning and feature representations



How is computer perception done?

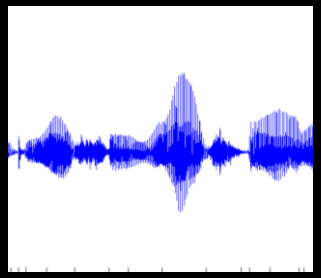
Object detection

Image

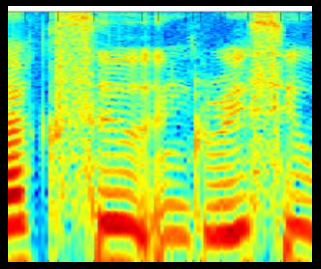
Low-level vision features

Recognition

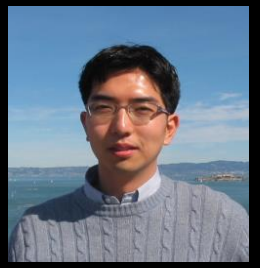
Audio classification



Audio



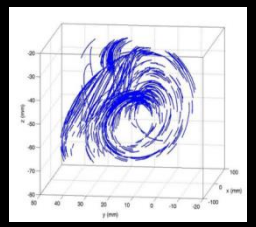
Low-level audio features



Speaker identification

Helicopter control

Helicopter



Low-level state features

Action

Machine learning and our focus

- Like human learning from past experiences.
- A computer does not have “experiences”.
- A computer system learns from data, which represent some “past experiences” of an application domain.
- Our focus: learn a target function that can be used to predict the values of a discrete class attribute, e.g., approve or not-approved, and high-risk or low risk.
- The task is commonly called: Supervised learning, classification, or inductive learning.

The data and the goal

- Data: A set of data records (also called examples, instances or cases) described by
  - *k* attributes: A_1, A_2, … A_k.
  - a class: Each example is labelled with a pre-defined class.
- Goal: To learn a classification model from the data that can be used to predict the classes of new (future, or test) cases/instances.

An example: data (loan application)

Approved or not

ID	Age	Has_Job	Own_House	Credit_Rating	Class
1	young	false	false	fair	**No**
2	young	false	false	good	**No**
3	young	true	false	good	**Yes**
4	young	true	true	fair	**Yes**
5	young	false	false	fair	**No**
6	middle	false	false	fair	**No**
7	middle	false	false	good	**No**
8	middle	true	true	good	**Yes**
9	middle	false	true	excellent	**Yes**
10	middle	false	true	excellent	**Yes**
11	old	false	true	excellent	**Yes**
12	old	false	true	good	**Yes**
13	old	true	false	good	**Yes**
14	old	true	false	excellent	**Yes**
15	old	false	false	fair	**No**

An example: the learning task

- Learn a classification model from the data
- Use the model to classify future loan applications into
  - Yes (approved) and
  - No (not approved)
- What is the class for following case/instance?

Age	Has_Job	Own_house	Credit-Rating	Class
young	false	false	good	?

Supervised vs. unsupervised Learning

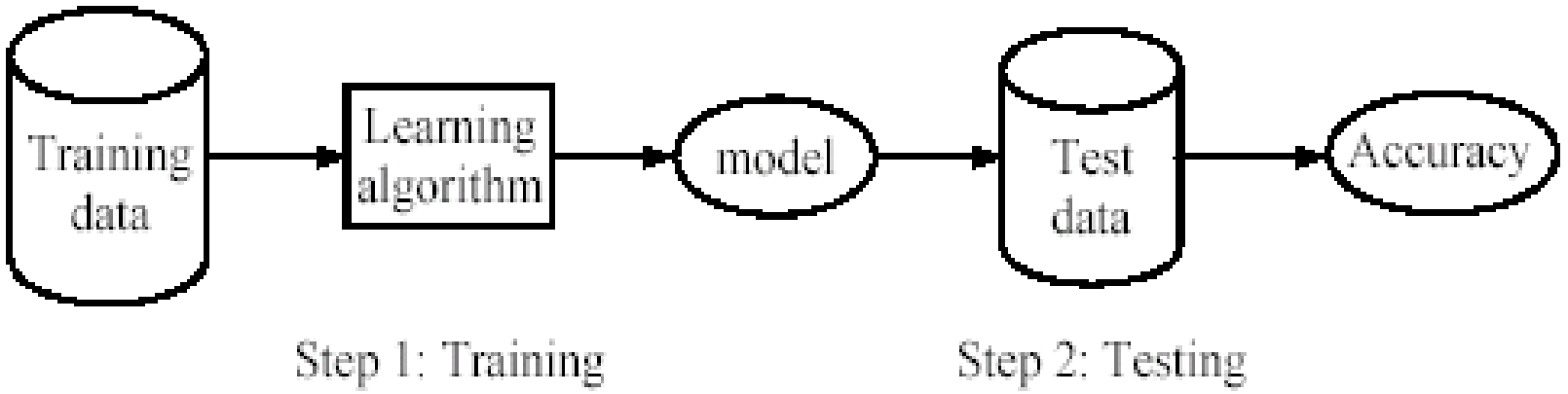
- Supervised learning: classification is seen as supervised learning from examples.
  - Supervision: The data (observations, measurements, etc.) are labeled with pre-defined classes. It is like that a “teacher” gives the classes (supervision).
  - Test data are classified into these classes too.
- Unsupervised learning (clustering)
  - Class labels of the data are unknown
  - Given a set of data, the task is to establish the existence of classes or clusters in the data

Supervised learning process: two steps

Learning (training): Learn a model using the training data

Testing: Test the model using unseen test data to assess the model accuracy

$$Accuracy = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}},$$



What do we mean by learning?

- Given
  - a data set *D*,
  - a task *T*, and
  - a performance measure *M*,

  a computer system is said to **learn** from *D* to perform the task *T* if after learning the system's performance on *T* improves as measured by *M*.
- In other words, the learned model helps the system to perform *T* better as compared to no learning.

An example

- Data: Loan application data
- Task: Predict whether a loan should be approved or not.
- Performance measure: accuracy.

No learning: classify all future applications (test data) to the majority class (i.e., Yes):

Accuracy = 9/15 = 60%.

- We can do better than 60% with learning.

Fundamental assumption of learning

Assumption: The distribution of training examples is identical to the distribution of test examples (including future unseen examples).

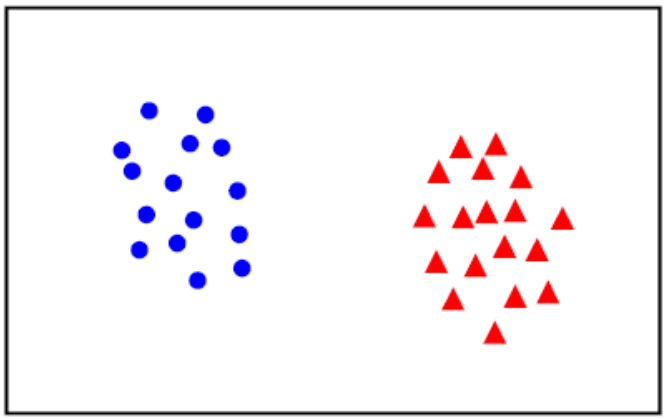
- In practice, this assumption is often violated to certain degree.
- Strong violations will clearly result in poor classification accuracy.
- To achieve good accuracy on the test data, training examples must be sufficiently representative of the test data.

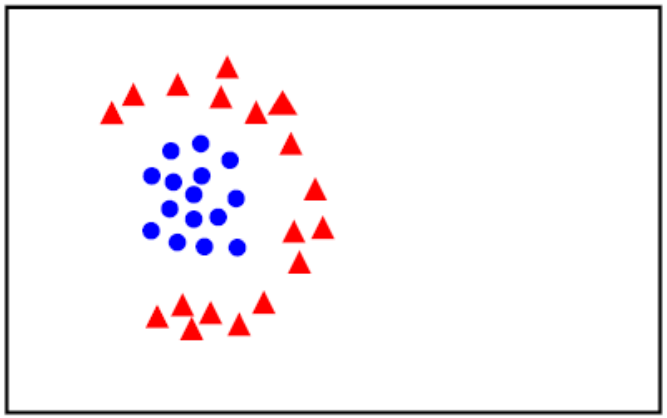
Binary Classification

Given training data $(\mathbf{x}_i, y_i)$ for $i = 1 \ldots N$, with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, learn a classifier $f(\mathbf{x})$ such that

$$f(\mathbf{x}_i) \begin{cases} \geq 0 & y_i = +1 \\ < 0 & y_i = -1 \end{cases}$$

i.e. $y_i f(\mathbf{x}_i) > 0$ for a correct classification.

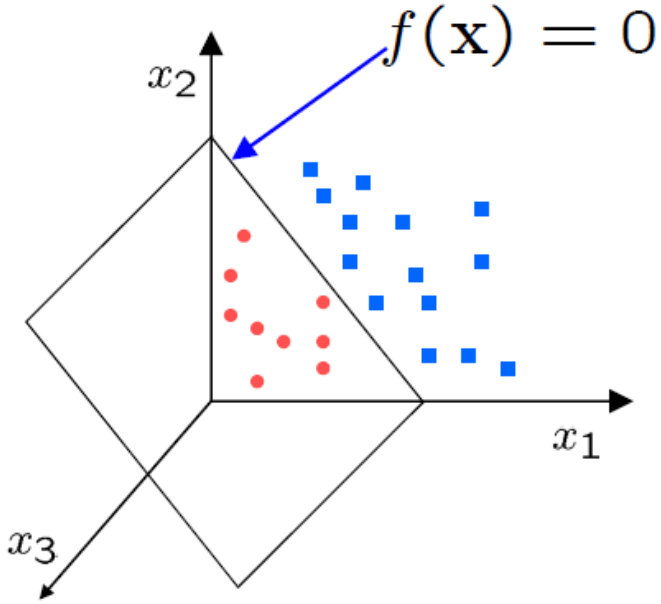




Linear classifiers

A linear classifier has the form

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$



- in 3D the discriminant is a plane, and in nD it is a hyperplane

For a K-NN classifier it was necessary to `carry' the training data

For a linear classifier, the training data is used to learn **w** and then discarded

Only **w** is needed for classifying new data

Road Map

- Basic concepts
- **K-nearest neighbor**
- Decision tree induction
- Ensemble methods: Bagging and Boosting
- Summary

k-Nearest Neighbor Classification (kNN)

- kNN does not build model from the training data.
- To classify a test instance *d*, define *k*-neighborhood *P* as *k* nearest neighbors of *d*
- Count number *n* of training instances in *P* that belong to class c_j
- Estimate Pr(c_j|*d*) as *n*/*k*
- No training is needed. Classification time is linear in training set size for each test case.

kNN Algorithm

Algorithm kNN(D, d, k)

1 Compute the distance between d and every example in D;

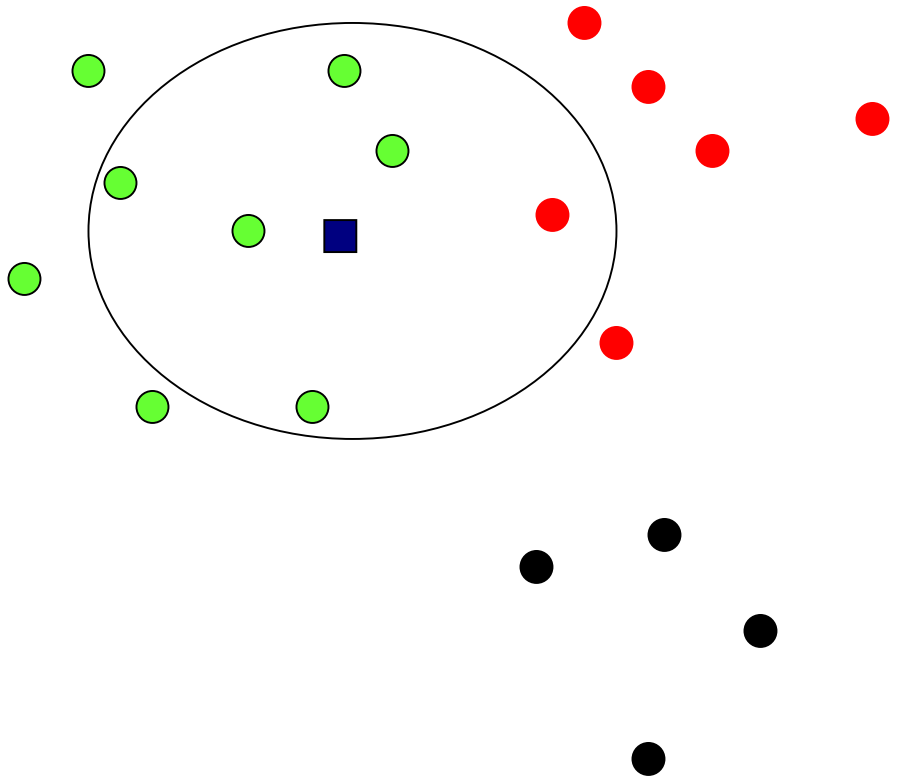
2 Choose the k examples in D that are nearest to d, denote the set by P ($\subseteq D$);

3 Assign d the class that is the most frequent class in P (or the majority class);

k is usually chosen empirically via a validation set or cross-validation by trying a range of *k* values.

Distance function is crucial, but depends on applications.

Example: k=6 (6NN)



- Government
- Science
- Arts

A new point
Pr(science|■)?

Discussions

- kNN can deal with complex and arbitrary decision boundaries.
- Despite its simplicity, researchers have shown that the classification accuracy of kNN can be quite strong and in many cases as accurate as other elaborated methods.
- kNN is slow at the classification time
- kNN does not produce an understandable model

Road Map

- Basic concepts
- K-nearest neighbor
- **Decision tree induction**
- Ensemble methods: Bagging and Boosting
- Summary

Introduction

- Decision tree learning is one of the most widely used techniques for classification.
  - Its classification accuracy is competitive with other methods, and
  - it is very efficient.
- The classification model is a tree, called decision tree.
- C4.5 by Ross Quinlan is perhaps the best known system and the codes are freelly available from internet.

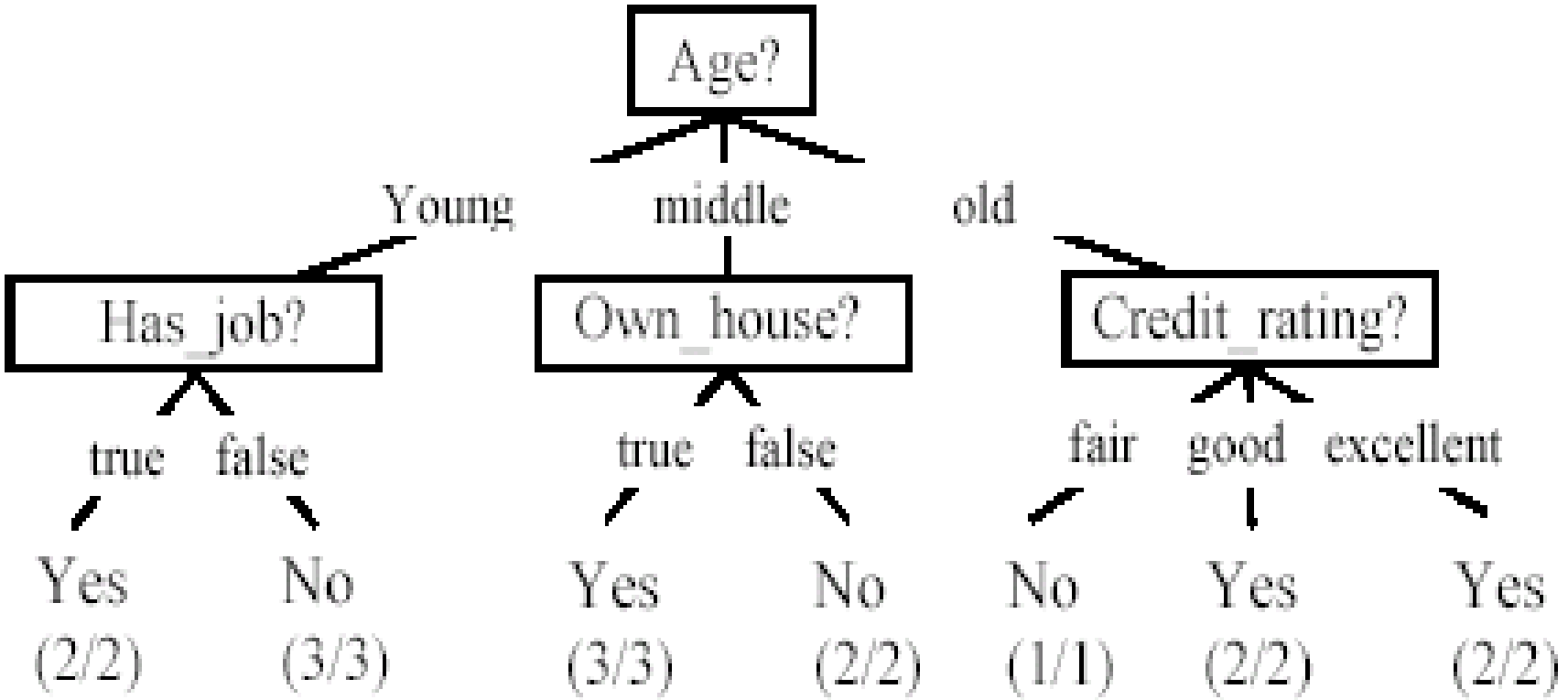
The loan data (reproduced)

Approved or not

ID	Age	Has_Job	Own_House	Credit_Rating	Class
1	young	false	false	fair	**No**
2	young	false	false	good	**No**
3	young	true	false	good	**Yes**
4	young	true	true	fair	**Yes**
5	young	false	false	fair	**No**
6	middle	false	false	fair	**No**
7	middle	false	false	good	**No**
8	middle	true	true	good	**Yes**
9	middle	false	true	excellent	**Yes**
10	middle	false	true	excellent	**Yes**
11	old	false	true	excellent	**Yes**
12	old	false	true	good	**Yes**
13	old	true	false	good	**Yes**
14	old	true	false	excellent	**Yes**
15	old	false	false	fair	**No**

A decision tree from the loan data

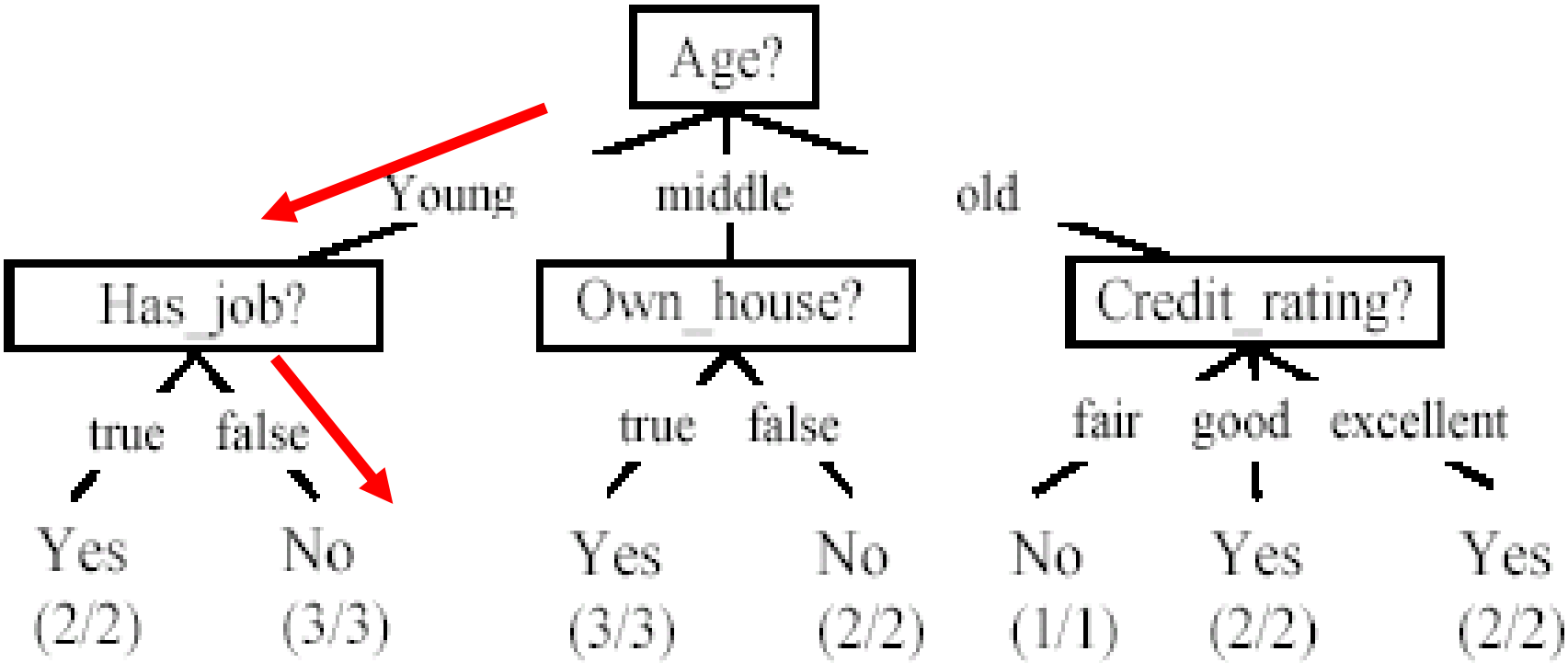
Decision nodes and leaf nodes (classes)



Use the decision tree

Age	Has_Job	Own_house	Credit-Rating	Class
young	false	false	good	?

No



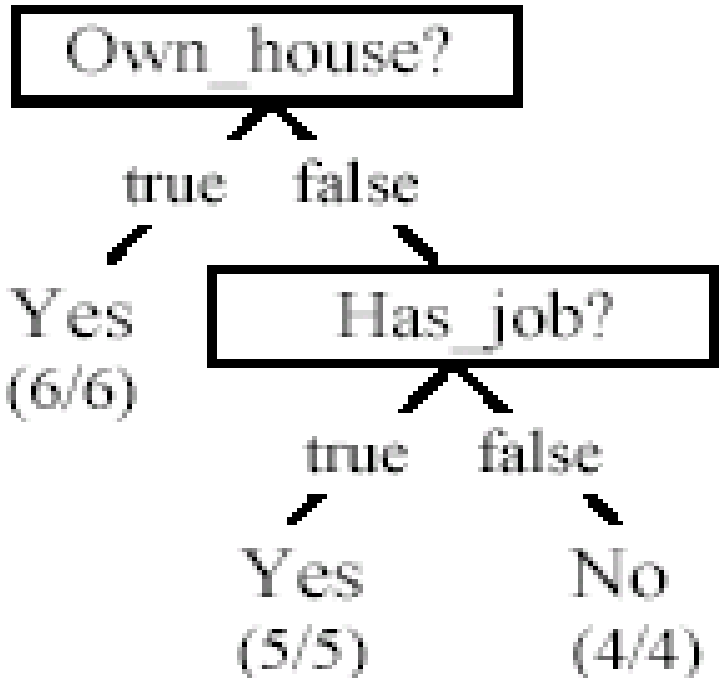
Is the decision tree unique?

No. Here is a simpler tree. We want smaller tree and accurate tree.

Easy to understand and perform better.

Finding the best tree is NP-hard.

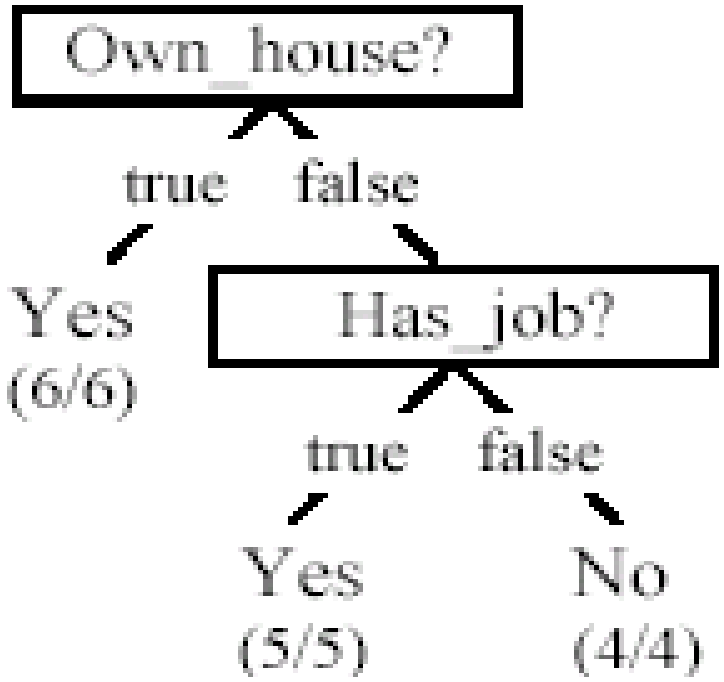
All current tree building algorithms are heuristic algorithms



From a decision tree to a set of rules

A decision tree can be converted to a set of rules

Each path from the root to a leaf is a rule.



Own_house = true → Class =Yes [sup=6/15, conf=6/6]

Own_house = false, Has_job = true → Class = Yes [sup=5/15, conf=5/5]

Own_house = false, Has_job = false → Class = No [sup=4/15, conf=4/4]

Algorithm for decision tree learning

- Basic algorithm (a greedy **divide-and-conquer** algorithm)
  - Assume attributes are categorical now (continuous attributes can be handled too)
  - Tree is constructed in a top-down recursive manner
  - At start, all the training examples are at the root
  - Examples are partitioned recursively based on selected attributes
  - Attributes are selected on the basis of an impurity function (e.g., information gain)
- Conditions for stopping partitioning
  - All examples for a given node belong to the same class
  - There are no remaining attributes for further partitioning – majority class is the leaf
  - There are no examples left

Decision tree learning algorithm

. **Algorithm** decisionTree(D, A, T)

```
1    if D contains only training examples of the same class c_j ∈ C then
2        make T a leaf node labeled with class c_j;
3    elseif A = ∅ then
4        make T a leaf node labeled with c_j, which is the most frequent class in D
5    else  // D contains examples belonging to a mixture of classes. We select a single
6          // attribute to partition D into subsets so that each subset is purer
7        p_0 = impurityEval-1(D);
8        for each attribute A_i ∈ {A_1, A_2, ..., A_k} do
9            p_i = impurityEval-2(A_i, D)
10       end
11       Select A_g ∈ {A_1, A_2, ..., A_k} that gives the biggest impurity reduction,
             computed using p_0 − p_i;
12       if p_0 − p_g < threshold then    // A_g does not significantly reduce impurity p_0
13         make T a leaf node labeled with c_j, the most frequent class in D.
14       else                             // A_g is able to reduce impurity p_0
15           Make T a decision node on A_g;
16           Let the possible values of A_g be v_1, v_2, ..., v_m. Partition D into m
                 disjoint subsets D_1, D_2, ..., D_m based on the m values of A_g.
17           for each D_j in {D_1, D_2, ..., D_m} do
18               if D_j ≠ ∅ then
19                   create a branch (edge) node T_j for v_j as a child node of T;
20                   decisionTree(D_j, A-{A_g}, T_j)// A_g is removed
21               end
22           end
23       end
24   end
```

Choose an attribute to partition data

- The *key* to building a decision tree - which attribute to choose in order to branch.
- The objective is to reduce impurity or uncertainty in data as much as possible.
  - A subset of data is pure if all instances belong to the same class.
- The *heuristic* in C4.5 is to choose the attribute with the maximum Information Gain or Gain Ratio based on information theory.

The loan data (reproduced)

Approved or not

ID	Age	Has_Job	Own_House	Credit_Rating	Class
1	young	false	false	fair	**No**
2	young	false	false	good	**No**
3	young	true	false	good	**Yes**
4	young	true	true	fair	**Yes**
5	young	false	false	fair	**No**
6	middle	false	false	fair	**No**
7	middle	false	false	good	**No**
8	middle	true	true	good	**Yes**
9	middle	false	true	excellent	**Yes**
10	middle	false	true	excellent	**Yes**
11	old	false	true	excellent	**Yes**
12	old	false	true	good	**Yes**
13	old	true	false	good	**Yes**
14	old	true	false	excellent	**Yes**
15	old	false	false	fair	**No**

Two possible roots, which is better?

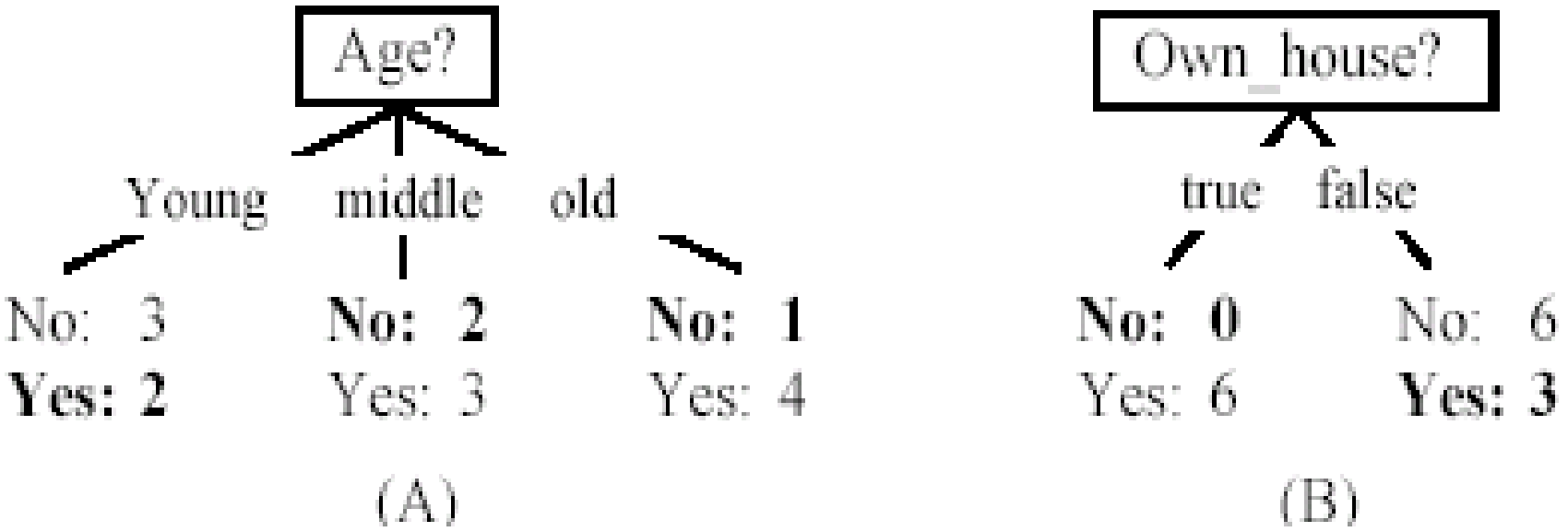


Fig. (B) seems to be better.

Information theory

- Information theory provides a mathematical basis for measuring the information content.
- To understand the notion of information, think about it as providing the answer to a question, for example, whether a coin will come up heads.
  - If one already has a good guess about the answer, then the actual answer is less informative.
  - If one already knows that the coin is rigged so that it will come with heads with probability 0.99, then a message (advanced information) about the actual outcome of a flip is worth less than it would be for a honest coin (50-50).

Information theory (cont …)

- For a fair (honest) coin, you have no information, and you are willing to pay more (say in terms of $) for advanced information - less you know, the more valuable the information.
- Information theory uses this same intuition, but instead of measuring the value for information in dollars, it measures information contents in **bits**.
- One bit of information is enough to answer a yes/no question about which one has no idea, such as the flip of a fair coin

Information theory: Entropy measure

- The entropy formula,

$$entropy(D) = -\sum_{j=1}^{|C|} \Pr(c_j) \log_2 \Pr(c_j)$$

$$\sum_{j=1}^{|C|} \Pr(c_j) = 1,$$

- $\Pr(c_j)$ is the probability of class c_j in data set D
- We use entropy as a measure of impurity or disorder of data set D. (Or, a measure of information in a tree)

Entropy measure: let us get a feeling

1. The data set D has 50% positive examples ($\Pr(positive) = 0.5$) and 50% negative examples ($\Pr(negative) = 0.5$).

$$entropy(D) = -0.5 \times \log_2 0.5 - 0.5 \times \log_2 0.5 = 1$$

2. The data set D has 20% positive examples ($\Pr(positive) = 0.2$) and 80% negative examples ($\Pr(negative) = 0.8$).

$$entropy(D) = -0.2 \times \log_2 0.2 - 0.8 \times \log_2 0.8 = 0.722$$

3. The data set D has 100% positive examples ($\Pr(positive) = 1$) and no negative examples, ($\Pr(negative) = 0$).

$$entropy(D) = -1 \times \log_2 1 - 0 \times \log_2 0 = 0$$

As the data become purer and purer, the entropy value becomes smaller and smaller. This is useful to us!

Information gain

- Given a set of examples *D*, we first compute its entropy:

$$entropy(D) = -\sum_{j=1}^{|C|} \Pr(c_j) \log_2 \Pr(c_j)$$

- If we make attribute A_i, with v values, the root of the current tree, this will partition *D* into v subsets D_1, D_2 …, D_v . The expected entropy if A_i is used as the current root:

$$entropy_{A_i}(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times entropy(D_j)$$

Information gain (cont …)

- Information gained by selecting attribute A_i to branch or to partition the data is

$$gain(D, A_i) = entropy(D) - entropy_{A_i}(D)$$

- We choose the attribute with the highest gain to branch/split the current tree.

An example

$$entropy(D) = -\frac{6}{15} \times \log_2 \frac{6}{15} - \frac{9}{15} \times \log_2 \frac{9}{15} = 0.971$$

$$entropy_{Own\_house}(D) = -\frac{6}{15} \times entropy(D_1) - \frac{9}{15} \times entropy(D_2)$$
$$= \frac{6}{15} \times 0 + \frac{9}{15} \times 0.918$$
$$= 0.551$$

ID	Age	Has_Job	Own_House	Credit_Rating	Class
1	young	false	false	fair	No
2	young	false	false	excellent	No
3	young	true	false	good	Yes
4	young	true	true	good	Yes
5	young	false	false	fair	No
6	middle	false	false	fair	No
7	middle	false	false	good	No
8	middle	true	true	good	Yes
9	middle	false	true	excellent	Yes
10	middle	false	true	excellent	Yes
11	old	false	true	excellent	Yes
12	old	false	true	good	Yes
13	old	true	false	good	Yes
14	old	true	false	excellent	Yes
15	old	false	false	fair	No

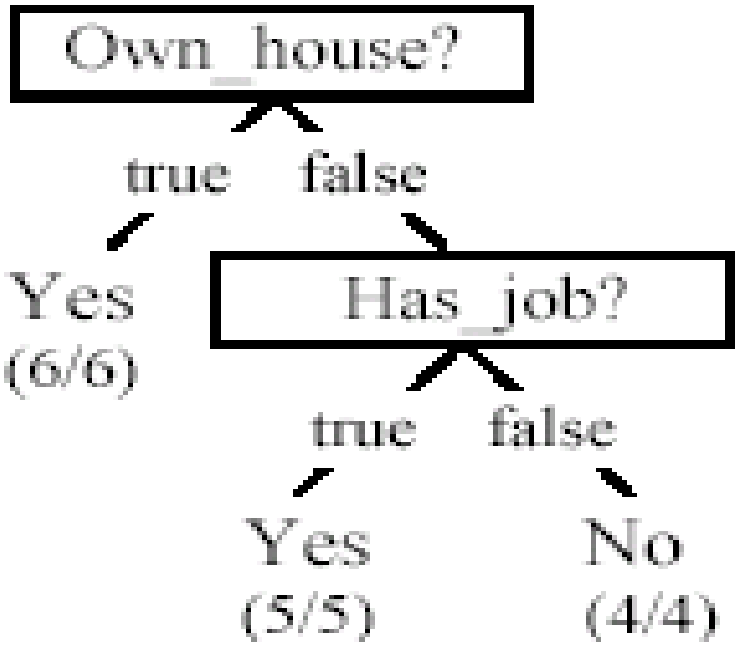
$$entropy_{Age}(D) = -\frac{5}{15} \times entropy(D_1) - \frac{5}{15} \times entropy(D_2) - \frac{5}{15} \times entropy(D_3)$$
$$= \frac{5}{15} \times 0.971 + \frac{5}{15} \times 0.971 + \frac{5}{15} \times 0.722$$
$$= 0.888$$

Age	Yes	No	entropy(Di)
young	2	3	0.971
middle	3	2	0.971
old	4	1	0.722

Own_house is the best choice for the root.

$gain(D, \text{Age}) = 0.971 - 0.888 = 0.083$
$gain(D, \text{Own\_house}) = 0.971 - 0.551 = 0.420$
$gain(D, \text{Has\_Job}) = 0.971 - 0.647 = 0.324$
$gain(D, \text{Credit\_Rating}) = 0.971 - 0.608 = 0.363$

We build the final tree

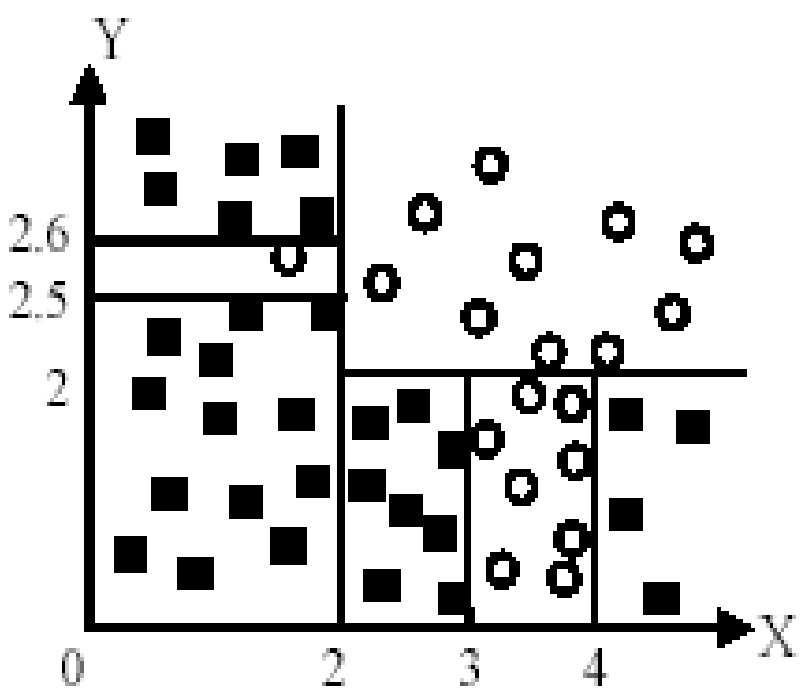


We can use information gain ratio to evaluate the impurity as well

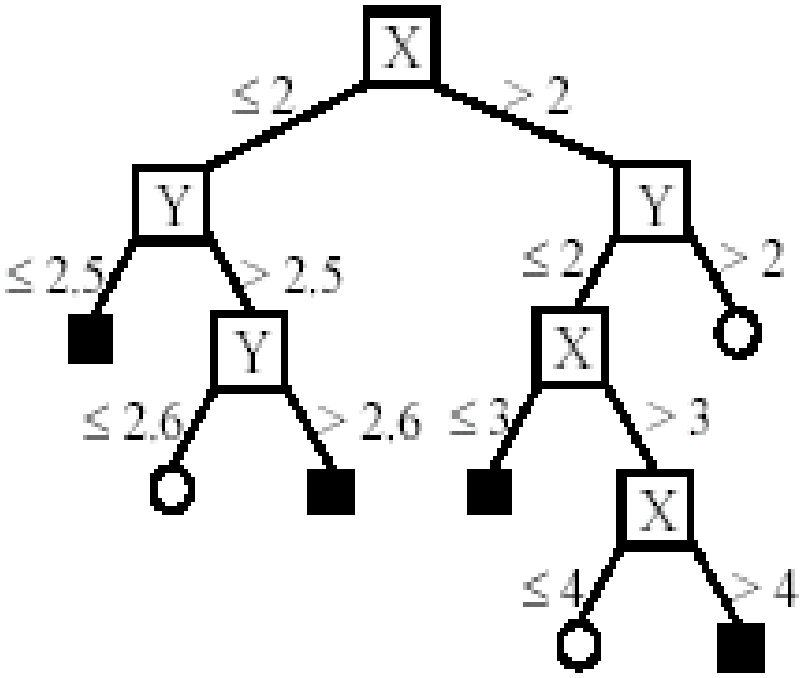
Handling continuous attributes

- Handle continuous attribute by splitting into two intervals (can be more) at each node.
- How to find the best threshold to divide?
  - Use information gain or gain ratio again
  - Sort all the values of an continuous attribute in increasing order $\{v_1, v_2, \ldots, v_r\}$,
  - One possible threshold between two adjacent values v_i and v_{i+1}. Try all possible thresholds and find the one that maximizes the gain (or gain ratio).

An example in a continuous space



(A) A partition of the data space



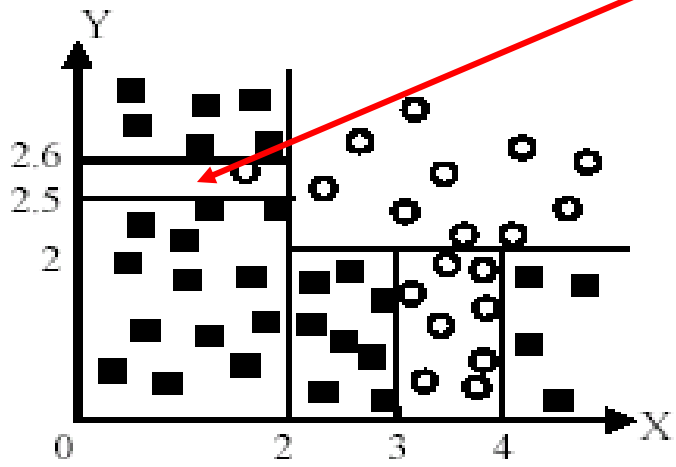
(B). The decision tree

Avoid overfitting in classification

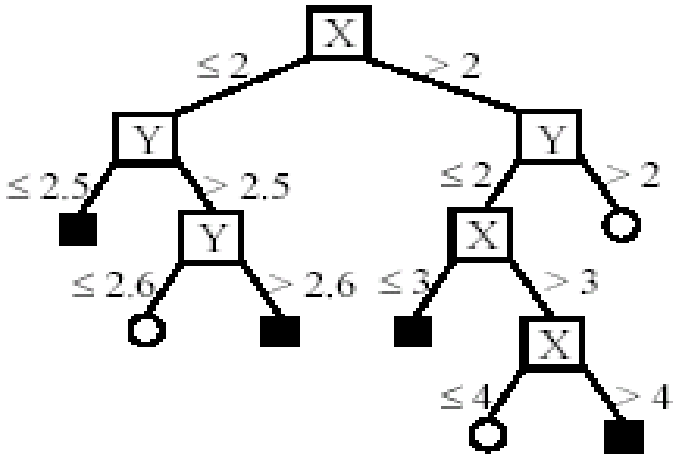
- Overfitting: A tree may overfit the training data
  - Good accuracy on training data but poor on test data
  - Symptoms: tree too deep and too many branches, some may reflect anomalies due to noise or outliers
- Two approaches to avoid overfitting
  - Pre-pruning: Halt tree construction early
    - Difficult to decide because we do not know what may happen subsequently if we keep growing the tree.
  - Post-pruning: Remove branches or sub-trees from a "fully grown" tree.
    - This method is commonly used. C4.5 uses a statistical method to estimates the errors at each node for pruning.
    - A validation set may be used for pruning as well.

An example

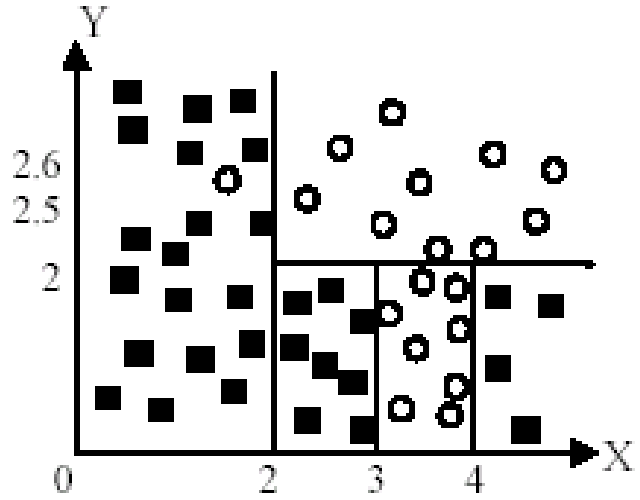
Likely to overfit the data

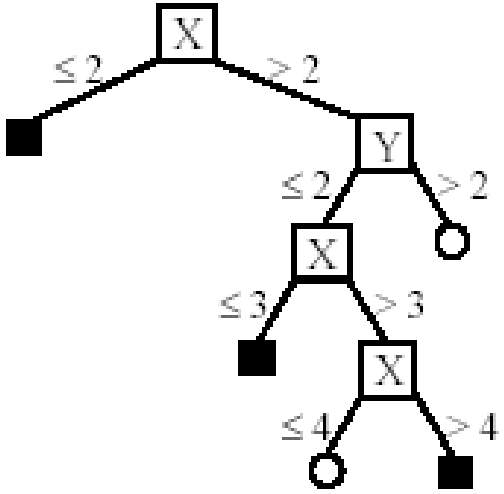


(A) A partition of the data space



(B). The decision tree





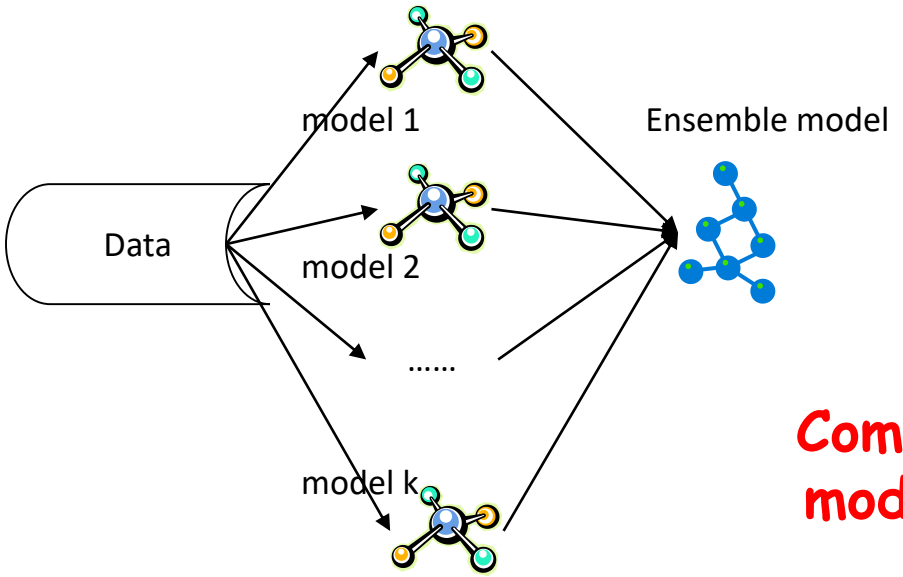
Other issues in decision tree learning

- From tree to rules, and rule pruning
- Handling of missing values
- Handing skewed distributions
- Handling attributes and classes with different costs.
- Attribute construction
- Etc.

Road Map

- Basic concepts
- K-nearest neighbor
- Decision tree induction
- **Ensemble methods: Bagging and Boosting**
- Summary

Ensemble



Combine multiple models into one!

Applications: classification, clustering, collaborative filtering, anomaly detection……

http://ews.uiuc.edu/~jinggao3/sdm10ensemble.htm

Motivations

- Motivations of ensemble methods
  - Ensemble model improves accuracy and robustness over single model methods
  - Applications:
    - distributed computing
    - privacy-preserving applications
    - large-scale data with reusable models
    - multiple sources of data
  - Efficiency: a complex problem can be decomposed into multiple sub-problems that are easier to understand and solve (divide-and-conquer approach)

Relationship with Related Studies

- Multi-task learning
  - Learn **multiple** tasks simultaneously
  - Ensemble methods: use multiple models to learn **one** task

- Data integration
  - Integrate raw data
  - Ensemble methods: integrate information at the **model** level

Relationship with Related Studies (2)

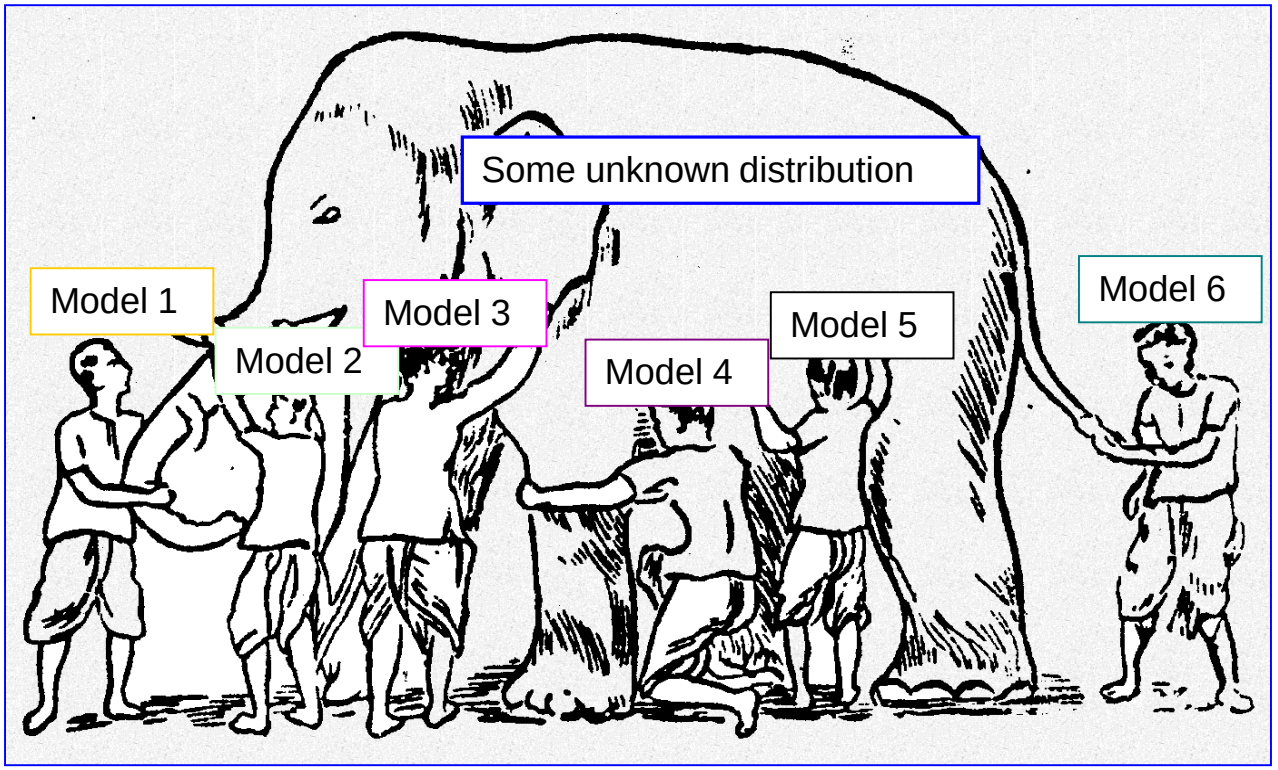
- Meta learning
  - **Learn** on meta-data (include base model output)
  - Ensemble methods: besides learn a joint model based on model output, we can also combine the output by **consensus**
- Non-redundant clustering
  - Give **multiple** non-redundant clustering solutions to users
  - Ensemble methods: give **one** solution to users which represents the consensus among all the base models

Why Ensemble Works?

- Intuition
  - combining diverse, independent opinions in human decision-making as a protective mechanism (e.g. stock portfolio)
- Uncorrelated error reduction
  - Suppose we have 5 completely independent classifiers for majority voting
  - If accuracy is 70% for each
    - 10 (.7^3)(.3^2)+5(.7^4)(.3)+(.7^5)
    - **83.7% majority vote accuracy**
  - 101 such classifiers
    - **99.9% majority vote accuracy**

from T. Holloway, Introduction to Ensemble Learning, 2007.

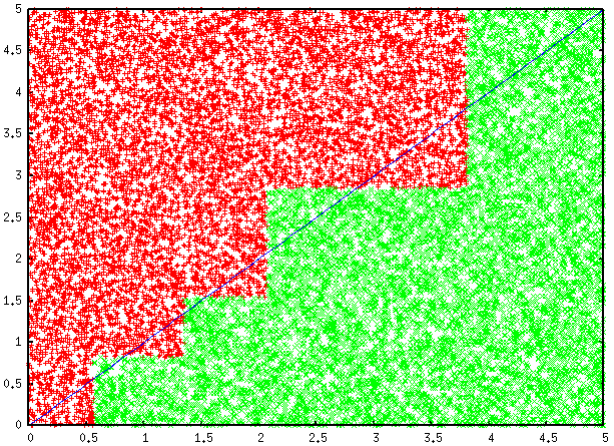
Why Ensemble Works? (2)



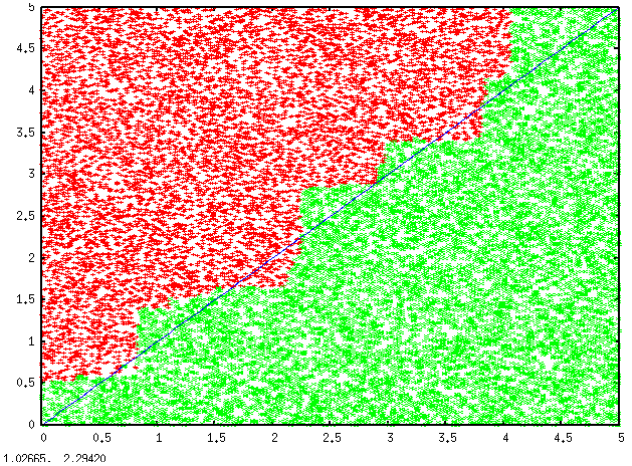
Ensemble gives the global picture!

Why Ensemble Works?

- Overcome limitations of single hypothesis
  - The target function may not be implementable with individual classifiers, but may be approximated by model averaging

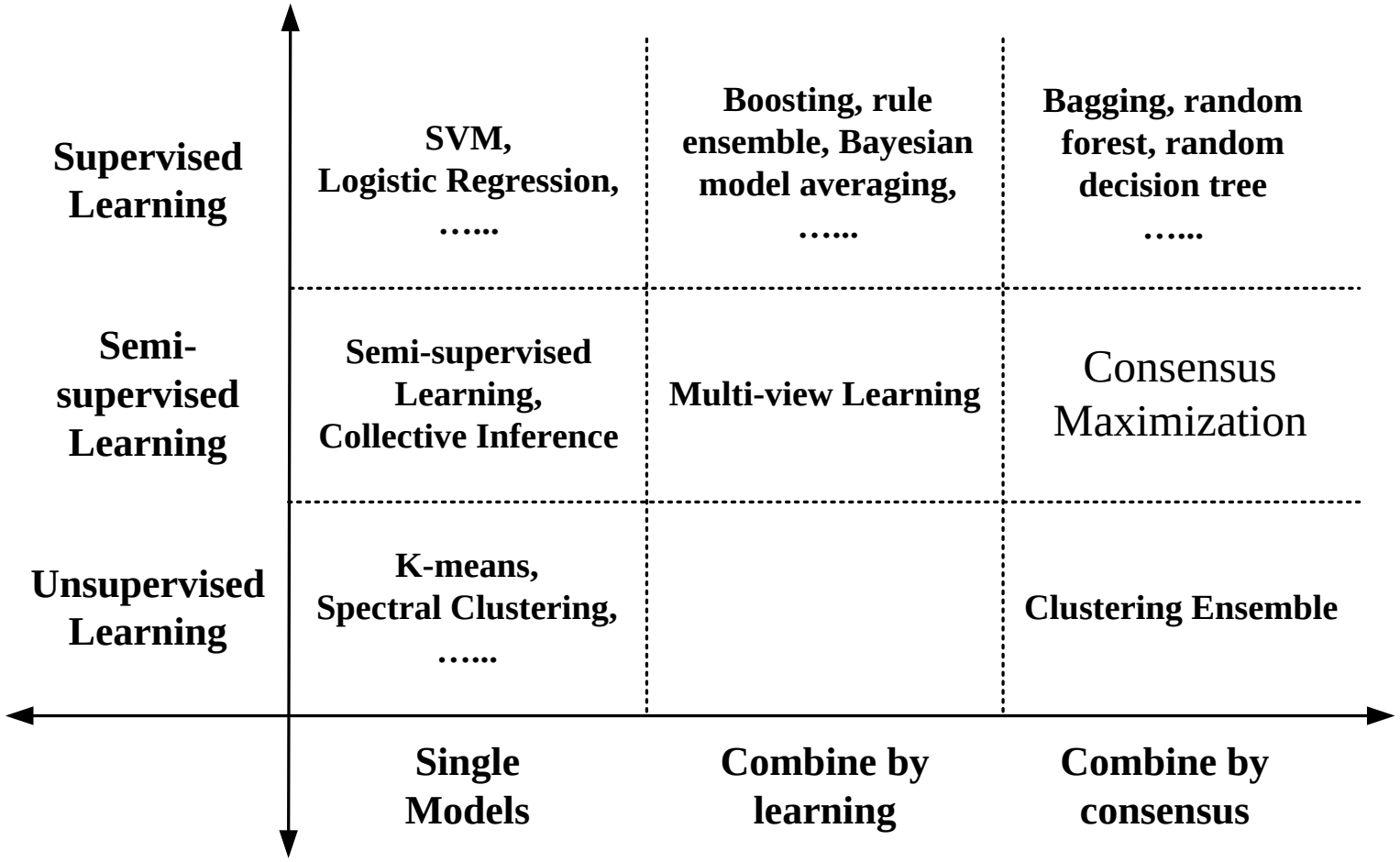


Decision Tree

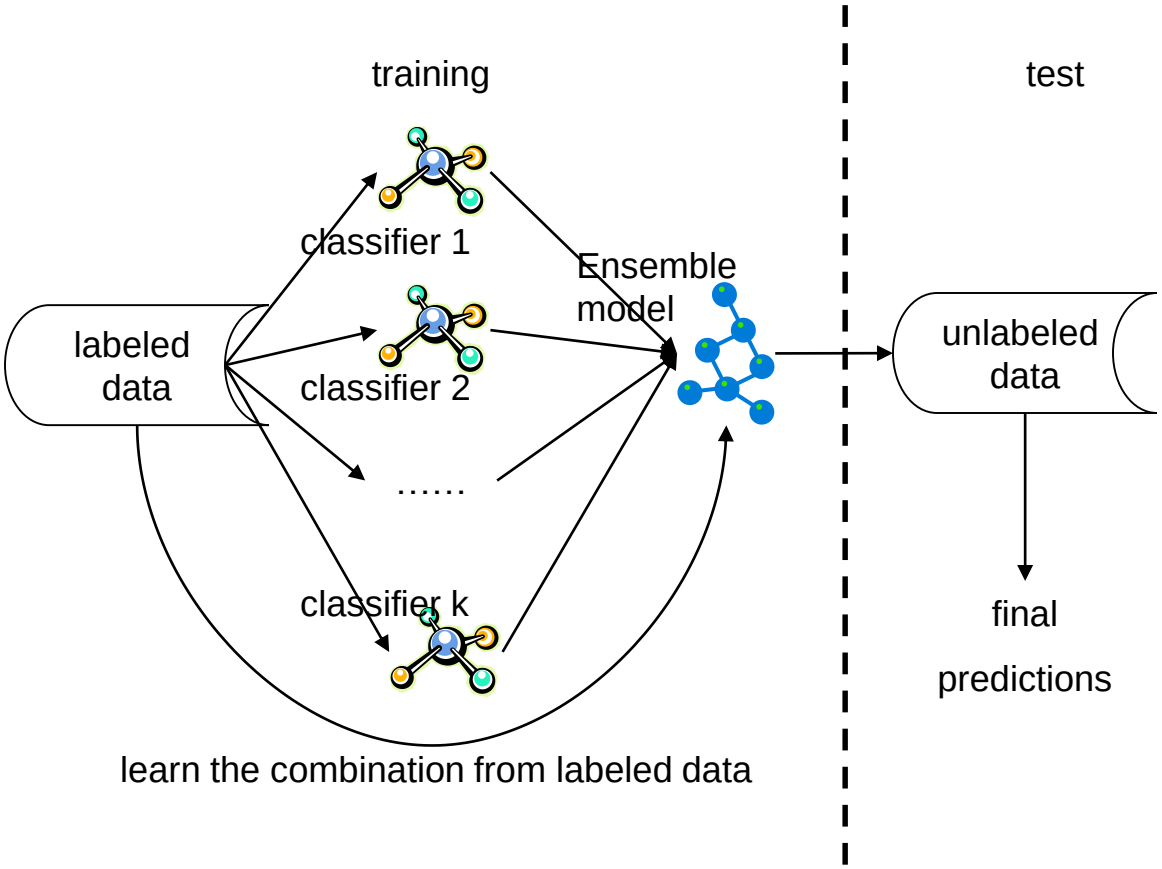


Model Averaging

Summary

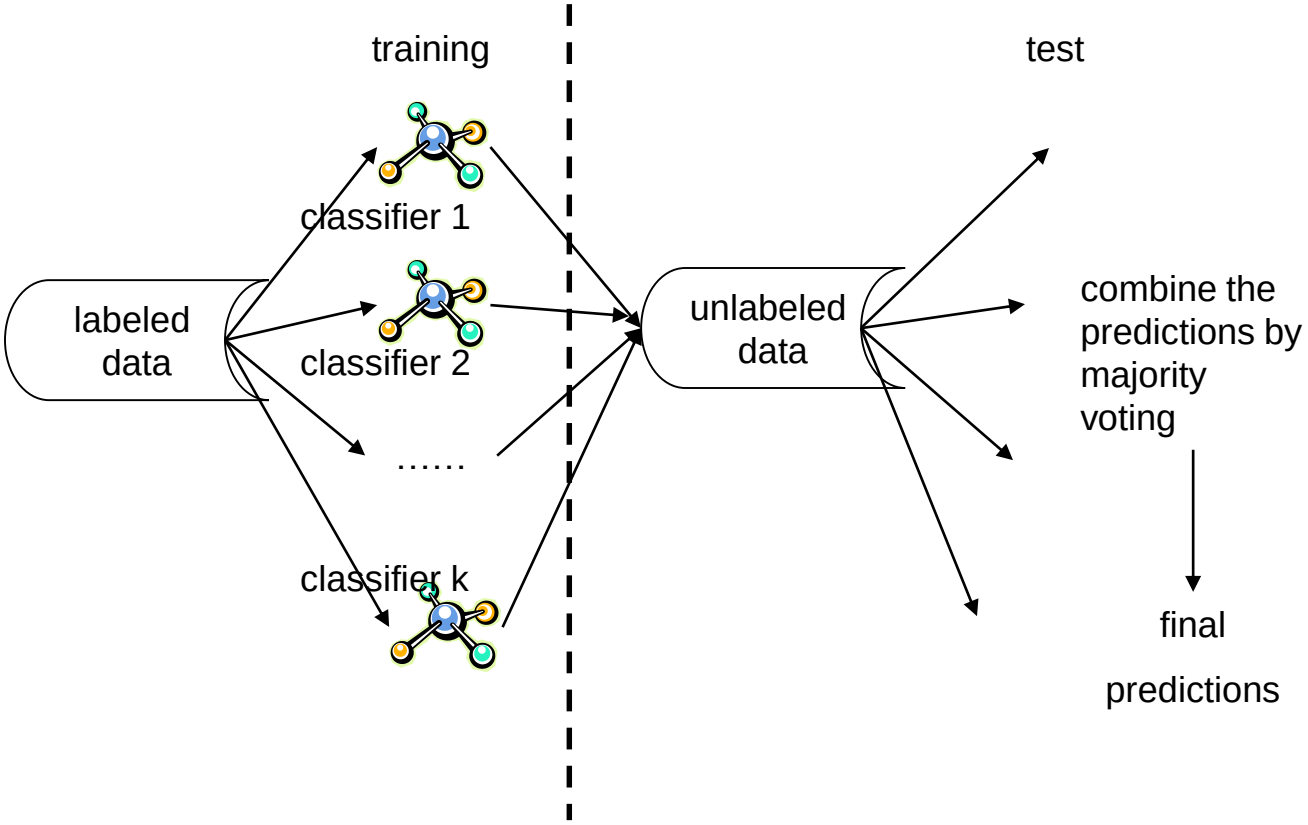

Supervised Learning
Semi-supervised Learning
Unsupervised Learning
SVM, Logistic Regression, …...
Boosting, rule ensemble, Bayesian model averaging, …...
Bagging, random forest, random decision tree …...
Semi-supervised Learning, Collective Inference
Multi-view Learning
Consensus Maximization
K-means, Spectral Clustering, …...
Clustering Ensemble
Single Models
Combine by learning
Combine by consensus

Ensemble of Classifiers—Learn to Combine



Algorithms: boosting, stacked generalization, rule ensemble,
Bayesian model averaging……

Ensemble of Classifiers—Consensus



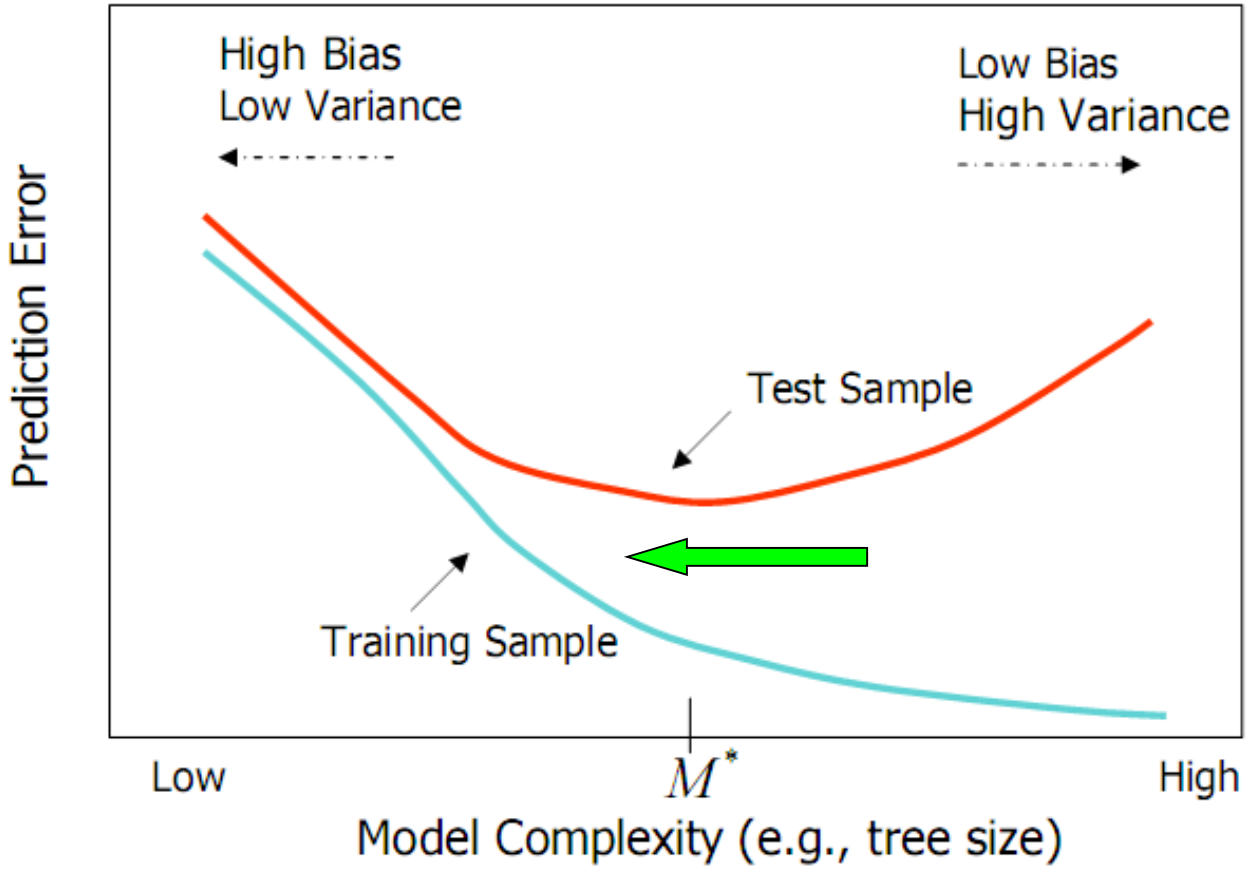
Algorithms: bagging, random forest, random decision tree, model averaging of probabilities……

Supervised Ensemble Methods

- Problem
  - Given a data set $D=\{x_1,x_2,\ldots,x_n\}$ and their corresponding labels $L=\{l_1,l_2,\ldots,l_n\}$
  - An ensemble approach computes:
    - A set of classifiers $\{f_1,f_2,\ldots,f_k\}$, each of which maps data to a class label: $f_j(x)=l$
    - A combination of classifiers f^* which minimizes generalization error: $f^*(x)= w_1f_1(x)+ w_2f_2(x)+\ldots+ w_kf_k(x)$

Bias and Variance

- Ensemble methods
  - Combine learners to reduce variance



from Elder, John. From Trees to Forests and Rule Sets - A Unified Overview of Ensemble Methods. 2007.

Generating Base Classifiers

- Sampling training examples
  - Train k classifiers on k subsets drawn from the training set
- Using different learning models
  - Use all the training examples, but apply different learning algorithms
- Sampling features
  - Train k classifiers on k subsets of features drawn from the feature space
- Learning “randomly”
  - Introduce randomness into learning procedures

Bagging

Bagging or bootstrap aggregation averages a given procedure over many samples, to reduce its variance — a poor man's Bayes. See

pp 246.

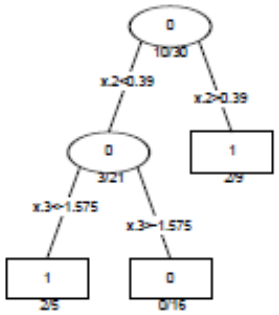
Suppose $C(\mathcal{S}, x)$ is a classifier, such as a tree, based on our training data $\mathcal{S}$, producing a predicted class label at input point x.

To bag C, we draw bootstrap samples $\mathcal{S}^{*1}, \ldots \mathcal{S}^{*B}$ each of size N with replacement from the training data. Then

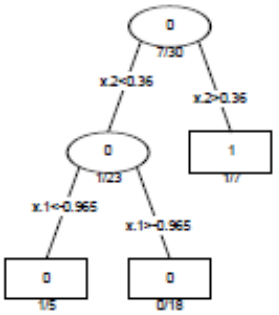
$$\hat{C}_{bag}(x) = \text{Majority Vote } \{C(\mathcal{S}^{*b}, x)\}_{b=1}^{B}.$$

Bagging can dramatically reduce the variance of unstable procedures (like trees), leading to improved prediction. However any simple structure in C (e.g a tree) is lost.

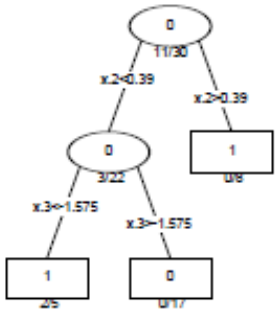
Original Tree



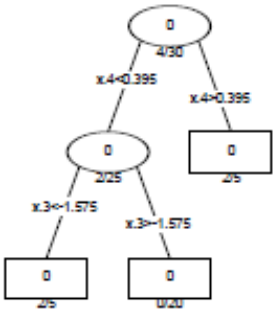
Bootstrap Tree 1



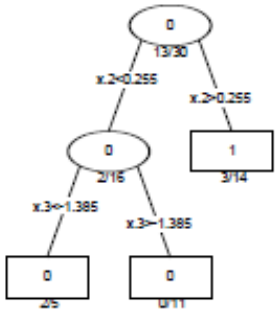
Bootstrap Tree 2



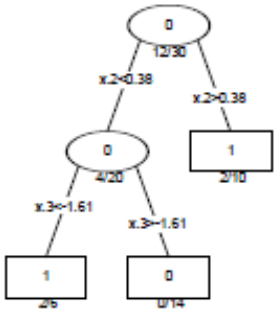
Bootstrap Tree 3



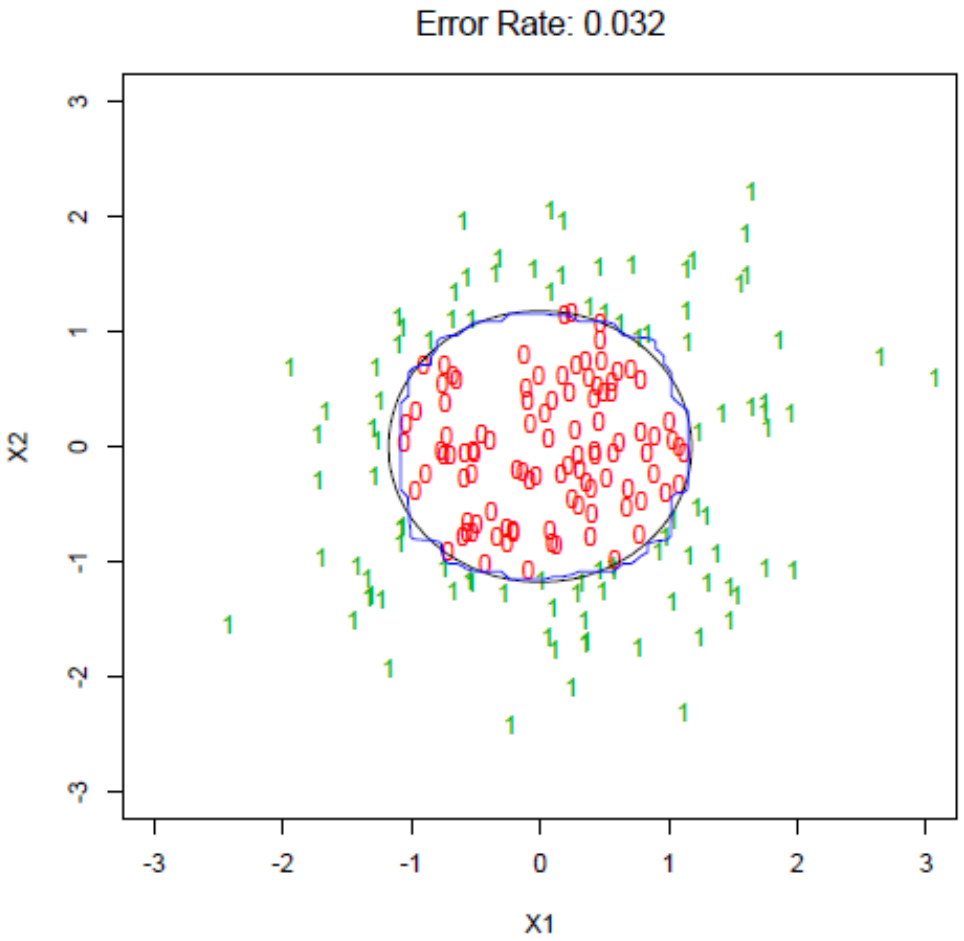
Bootstrap Tree 4



Bootstrap Tree 5



Decision Boundary: Bagging



Bagging averages many trees, and produces smoother decision boundaries.

Random forest classifier

- Random forest classifier, an extension to bagging which uses *de-correlated* trees.

Random Forest Classifier

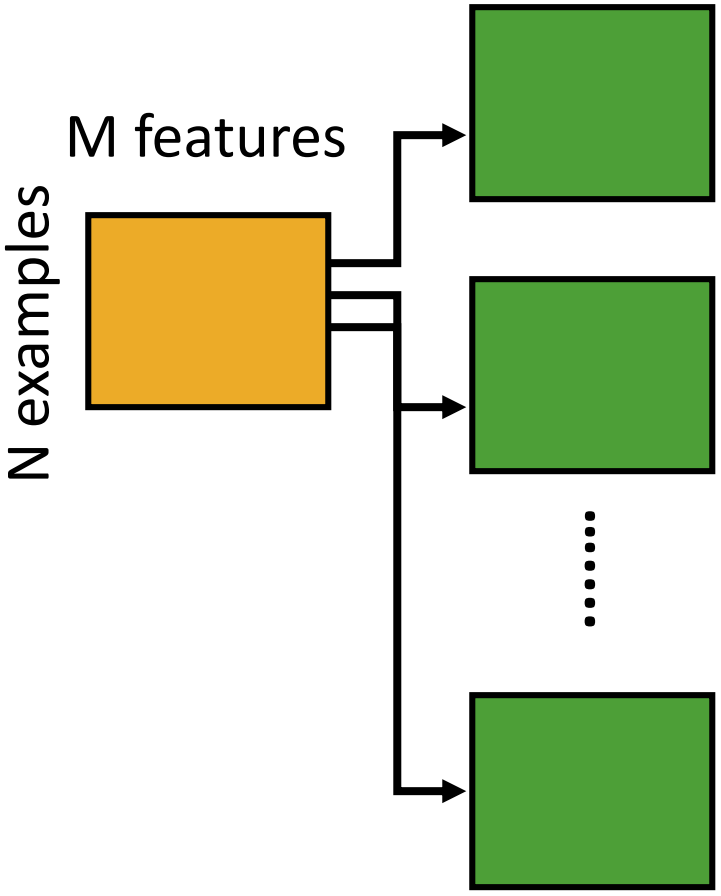
Training Data

M features

N examples

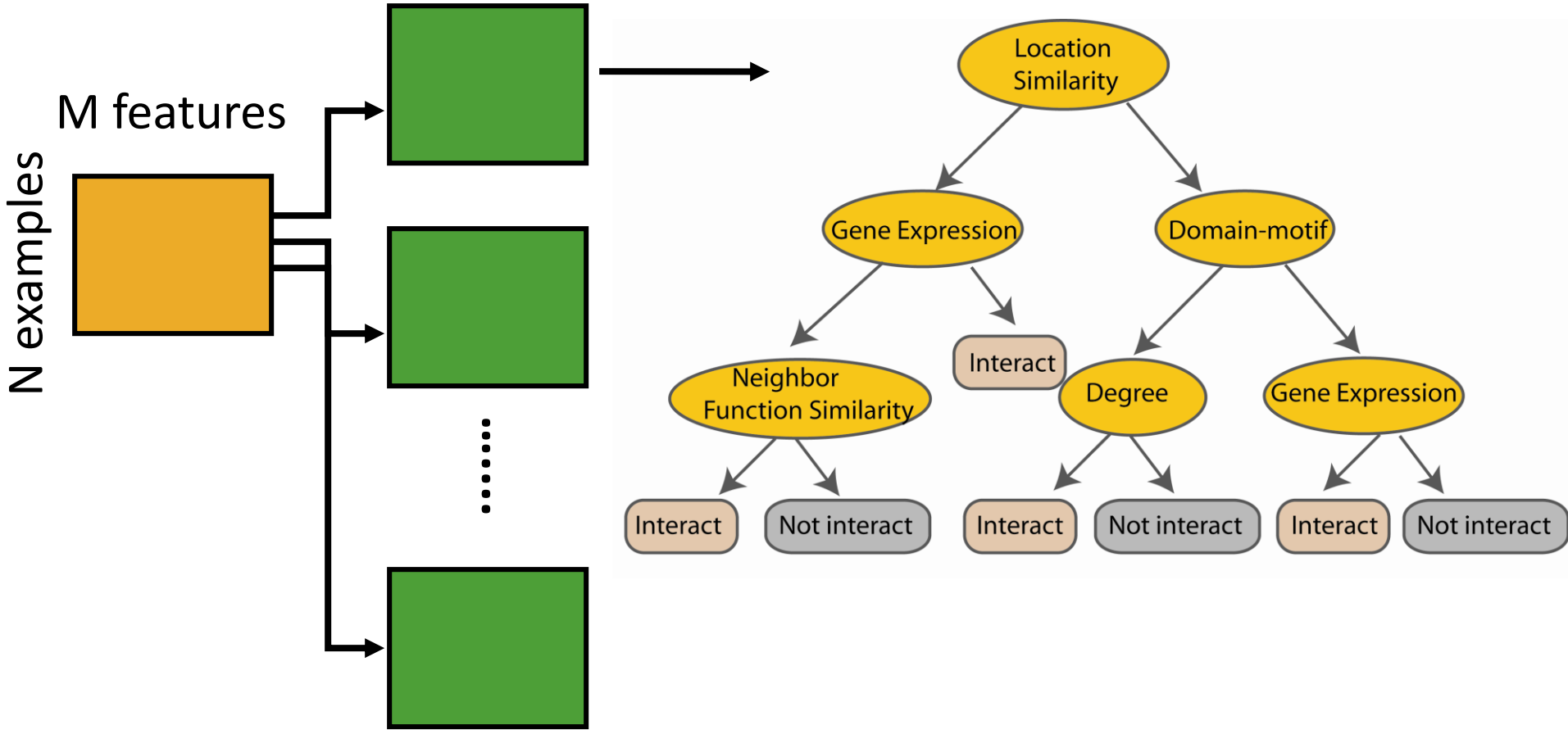
Random Forest Classifier

Create bootstrap samples
from the training data



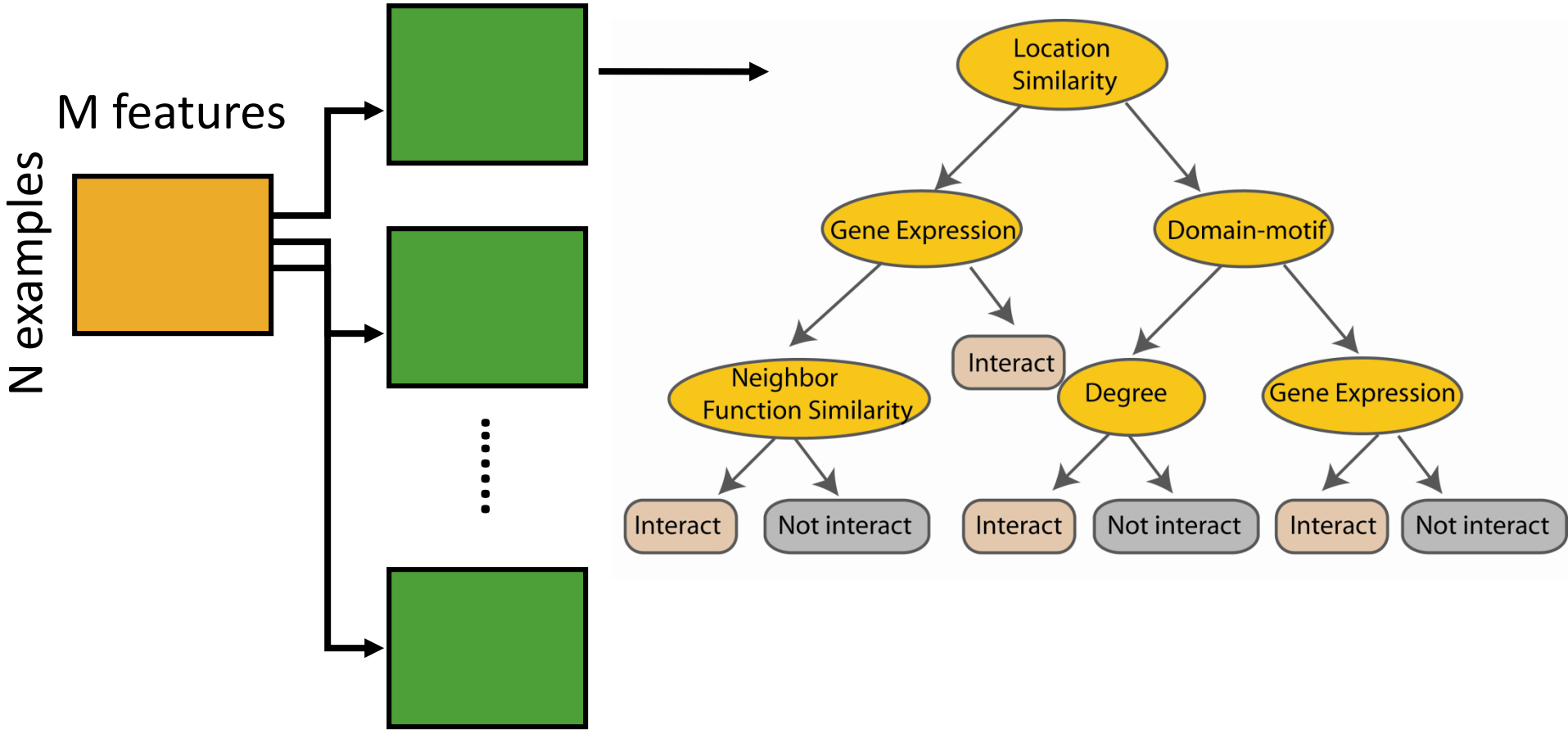
Random Forest Classifier

Construct a decision tree

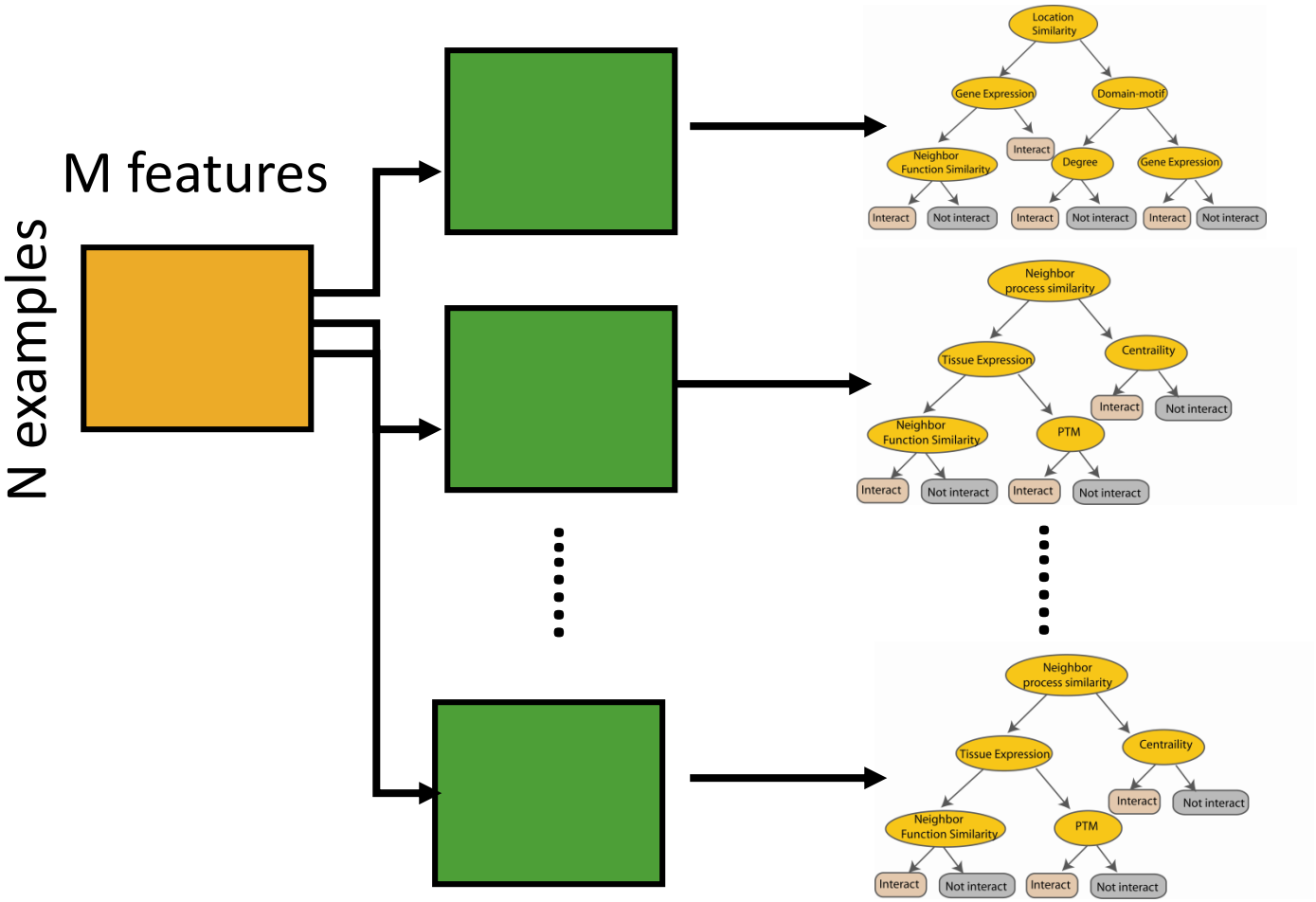


Random Forest Classifier

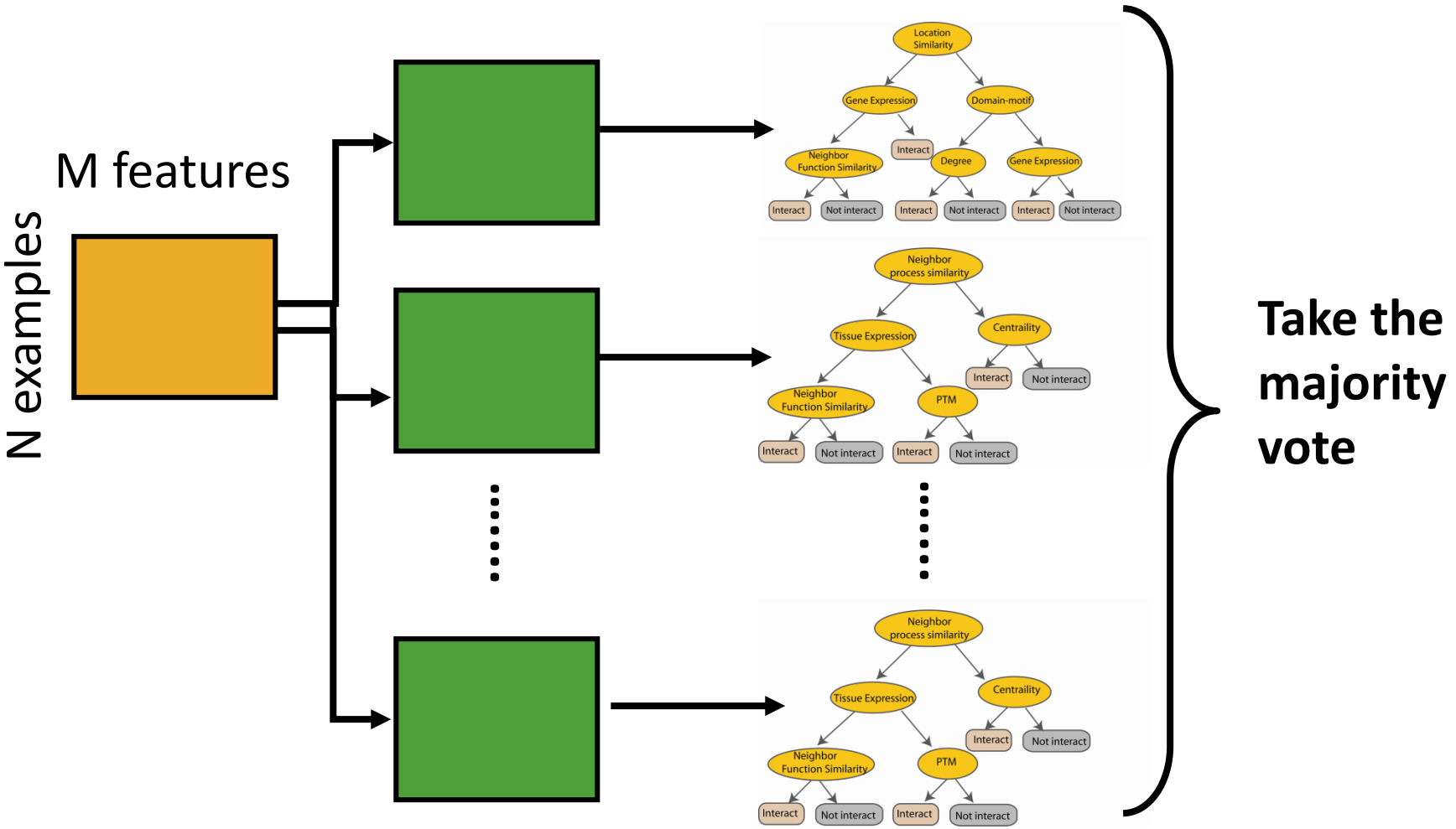
At each node in choosing the split feature choose only among *m*<*M* features



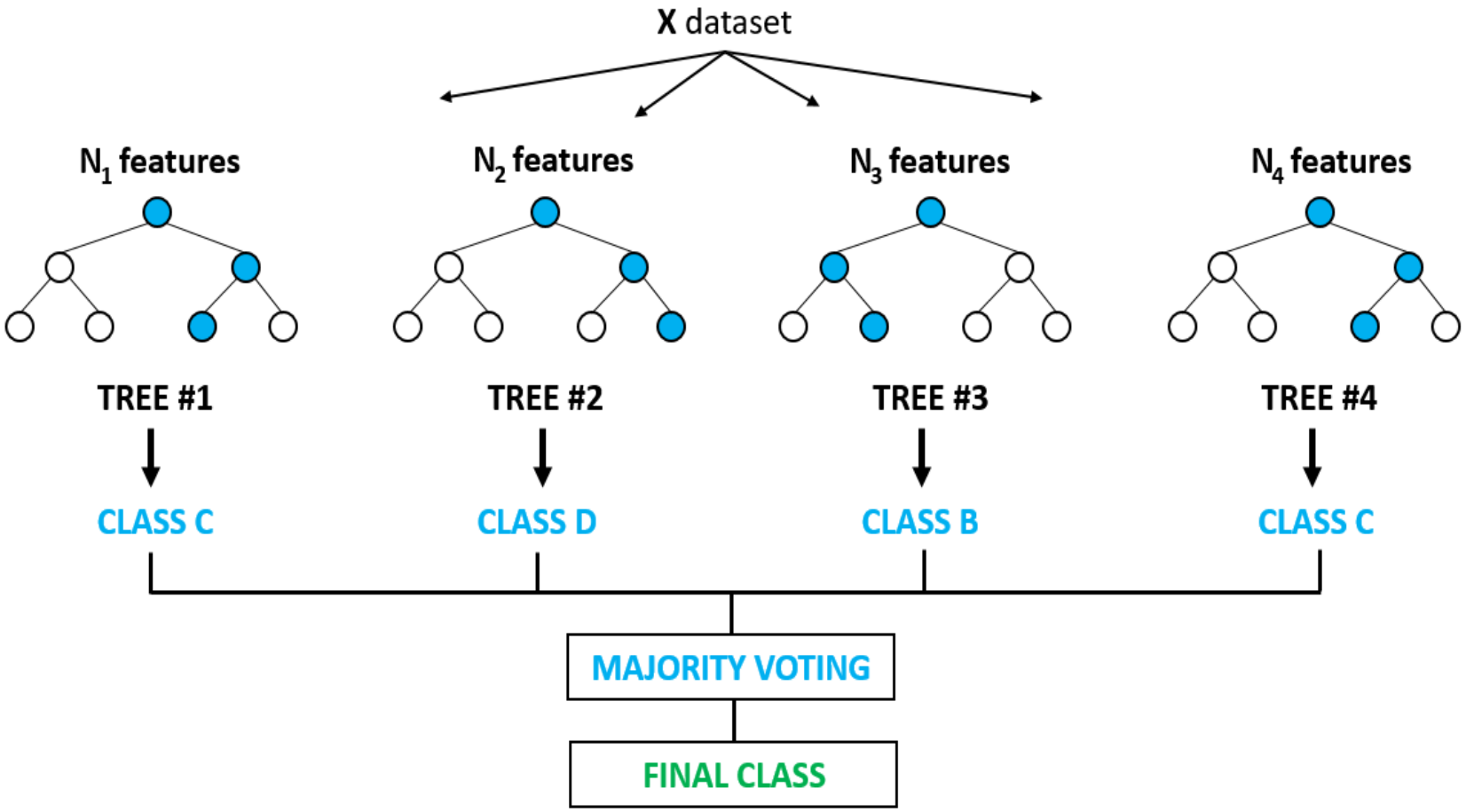
Random Forest Classifier



Random Forest Classifier



Thus,…..


X dataset
N1 features
N2 features
N3 features
N4 features
TREE #1
TREE #2
TREE #3
TREE #4
CLASS C
CLASS D
CLASS B
CLASS C
MAJORITY VOTING
FINAL CLASS

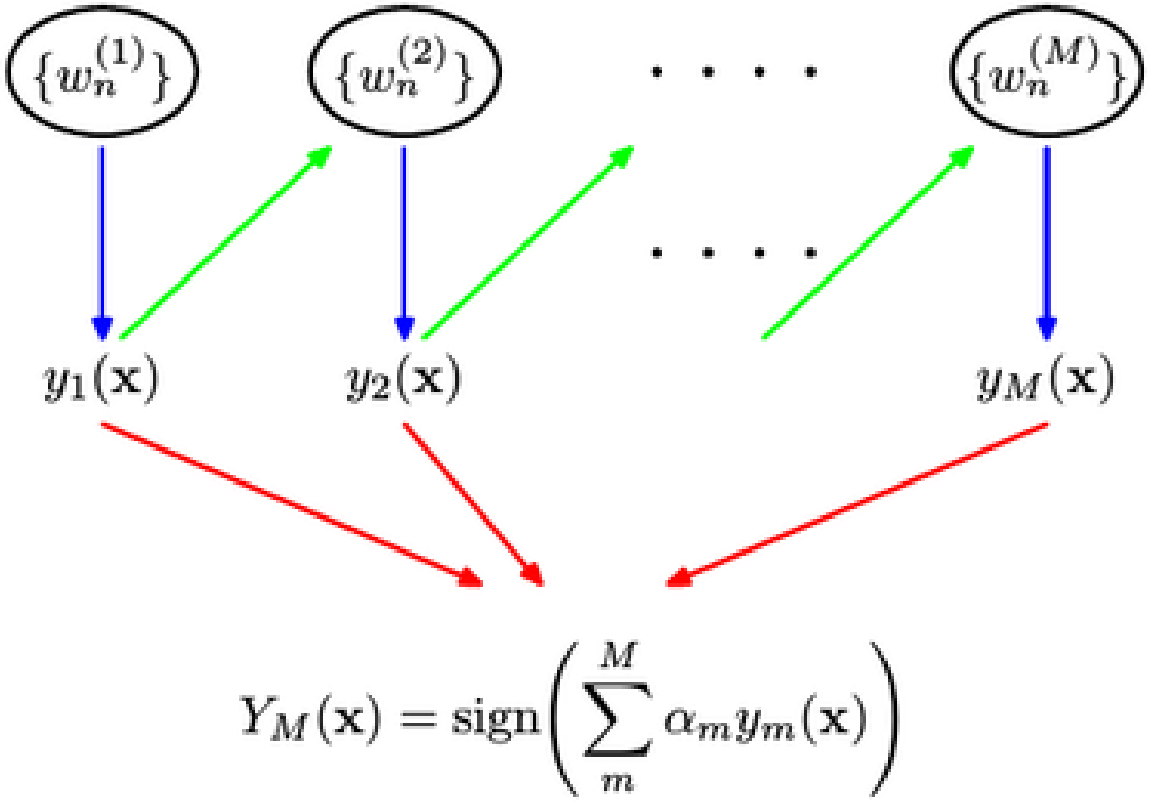
Random forest

- Available package:
- http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

- To read more:
- http://www-stat.stanford.edu/~hastie/Papers/ESLII.pdf

Boosting

Schematic illustration of the boosting framework. Each base classifier $y_m(\mathbf{x})$ is trained on a weighted form of the training set (blue arrows) in which the weights $w_n^{(m)}$ depend on the performance of the previous base classifier $y_{m-1}(\mathbf{x})$ (green arrows). Once all base classifiers have been trained, they are combined to give the final classifier $Y_M(\mathbf{x})$ (red arrows).



Boosting (1)

- Principles
  - Boost a set of weak learners to a strong learner
  - Make records currently misclassified more important

- Example
  - Record 4 is hard to classify
  - Its weight is increased, therefore it is more likely to be chosen again in subsequent rounds

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

*[FrSc97]

from P. Tan et al. Introduction to Data Mining.

Boosting (2)

- **AdaBoost**
  - Initially, set uniform weights on all the records
  - At each round
    - Create a bootstrap sample based on the weights
    - Train a classifier on the sample and apply it on the original training set
    - Records that are wrongly classified will have their weights increased
    - Records that are classified correctly will have their weights decreased
    - If the error rate is higher than 50%, start over
  - Final prediction is weighted average of all the classifiers with weight representing the training accuracy

Boosting (3)

- **Determine the weight**
  - **For classifier *i*, its error is** $\varepsilon_i = \frac{\sum_{j=1}^{N} w_j \delta(C_i(x_j) \neq y_j)}{\sum_{j=1}^{N} w_j}$
  - **The classifier's importance is represented as:**

$$\alpha_i = \frac{1}{2} \ln\left(\frac{1-\varepsilon_i}{\varepsilon_i}\right)$$

  - **The weight of each record is updated as:**

$$w_j^{(i+1)} = \frac{w_j^{(i)} \exp\left(-\alpha_i y_j C_i(x_j)\right)}{Z^{(i)}}$$

  - **Final combination:**

$$C^*(x) = \arg\max_y \sum_{i=1}^{K} \alpha_i \delta\left(C_i(x) = y\right)$$

In a nutshell….

Model Averaging

Classification trees can be simple, but often produce noisy (bushy) or weak (stunted) classifiers.

- Bagging (Breiman, 1996): Fit many large trees to bootstrap-resampled versions of the training data, and classify by majority vote.
- Boosting (Freund & Shapire, 1996): Fit many large or small trees to reweighted versions of the training data. Classify by weighted majority vote.
- Random Forests (Breiman 1999): Fancier version of bagging.

In general Boosting $\succ$ Random Forests $\succ$ Bagging $\succ$ Single Tree.


thank you

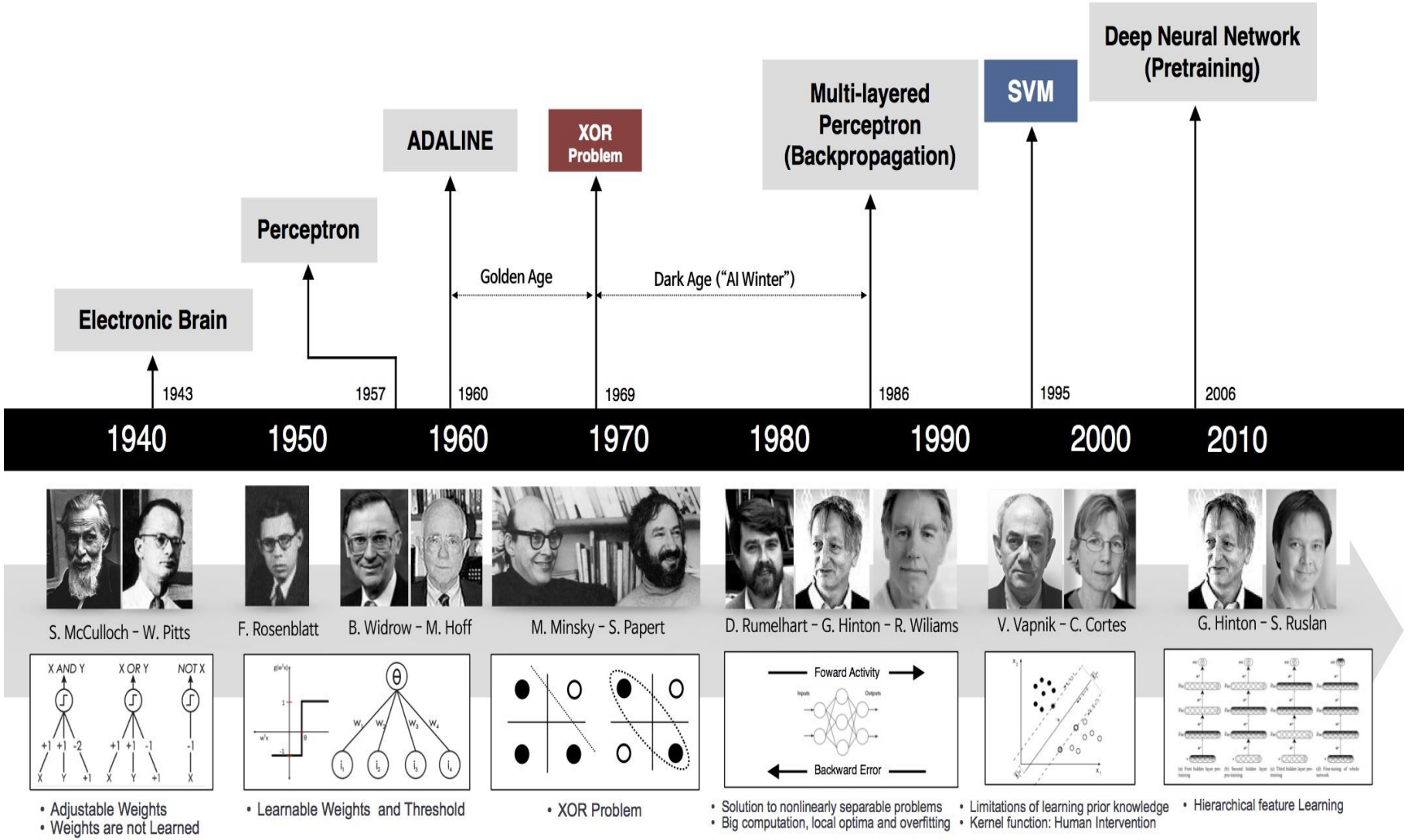
Supervised Learning: Artificial Neural Networks

What are connectionist neural networks?

- Connectionism refers to a computer modeling approach to computation that is loosely based upon the architecture of the brain.

- Many different models, but all include:
  - Multiple, individual “nodes” or “units” that operate at the same time (in parallel)
  - A network that connects the nodes together
  - Information is stored in a distributed fashion among the links that connect the nodes
  - Learning can occur with gradual changes in connection strength

Developments in Neural Learning Systems



The 2012 Breakthrough….

- A Krizhevsky, I Sutskever, and GE Hinton, "**Imagenet classification with deep convolutional neural networks**", Advances in neural information processing systems (***NIPS 2012***), 1097-1105

Has 18,000+ citations by now!!

Reduced the error rate on ***imagenet*** under LSVRC to 16%

Later evolved into ***Alexnet***.

Deep Learning: Acknowledging Some Resources used here…

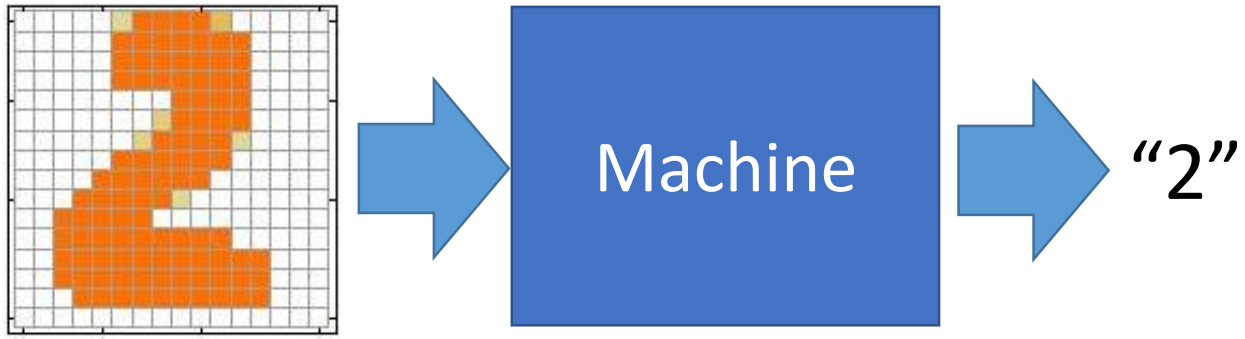
- "Neural Networks and Deep Learning"
  - written by Michael Nielsen
  - http://neuralnetworksanddeeplearning.com/
- "Deep Learning"
  - Written by Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville
  - http://www.iro.umontreal.ca/~bengioy/dlbook/
    - Deep Learning Tutorial. Hung-yi Lee, NTU.

Introduction to Neural Learning

What people already knew since 1950s

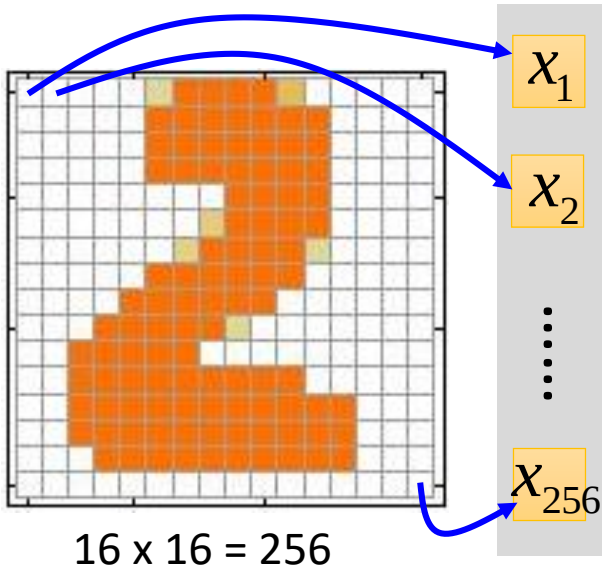
Example Application

- Handwriting Digit Recognition



Handwriting Digit Recognition

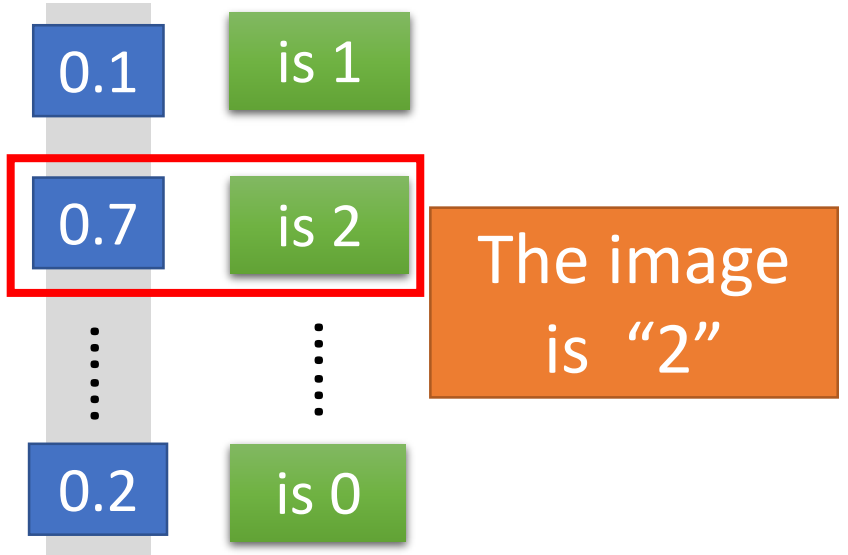
Input



Ink → 1

No ink → 0

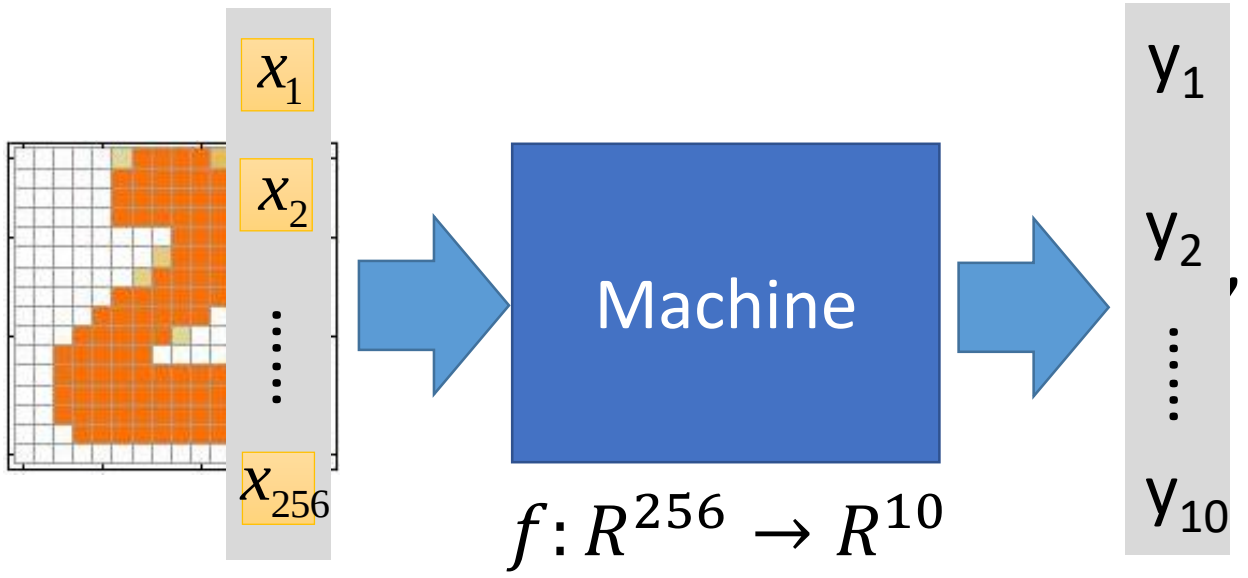
Output



Each dimension represents the confidence of a digit.

Example Application

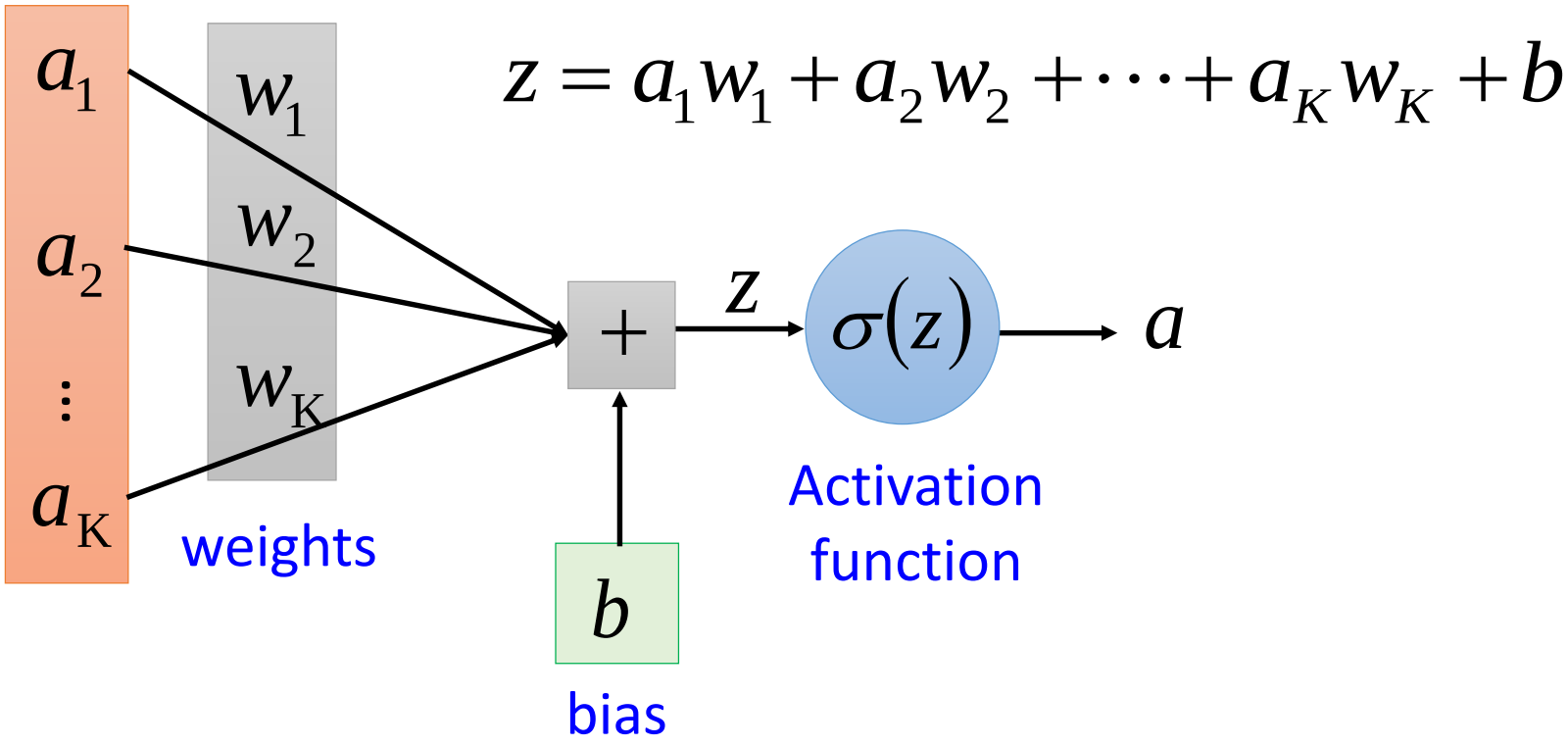
- Handwriting Digit Recognition



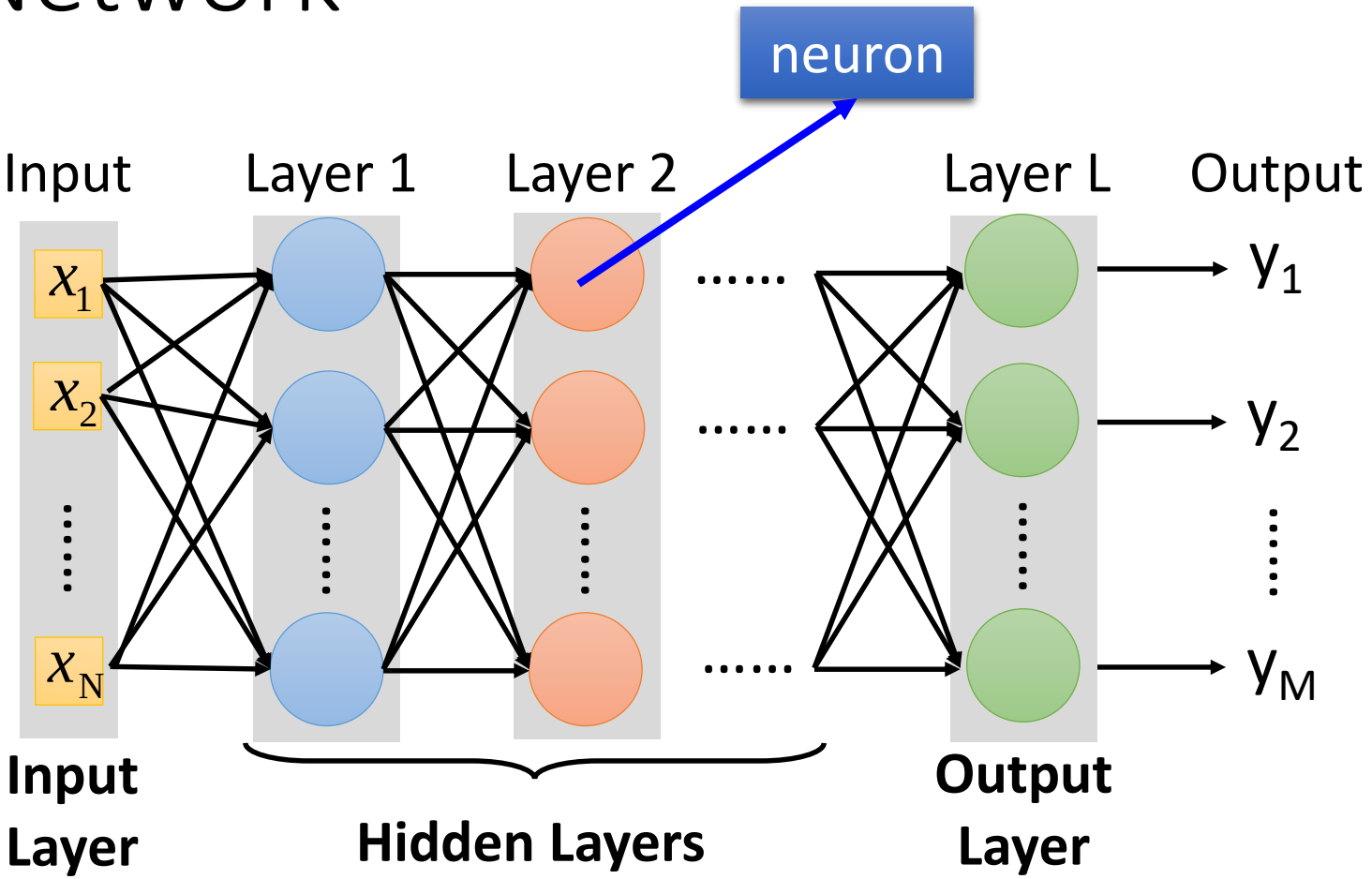
In deep learning, the function f is represented by neural network

Element of Neural Network

Neuron $f: R^K \rightarrow R$

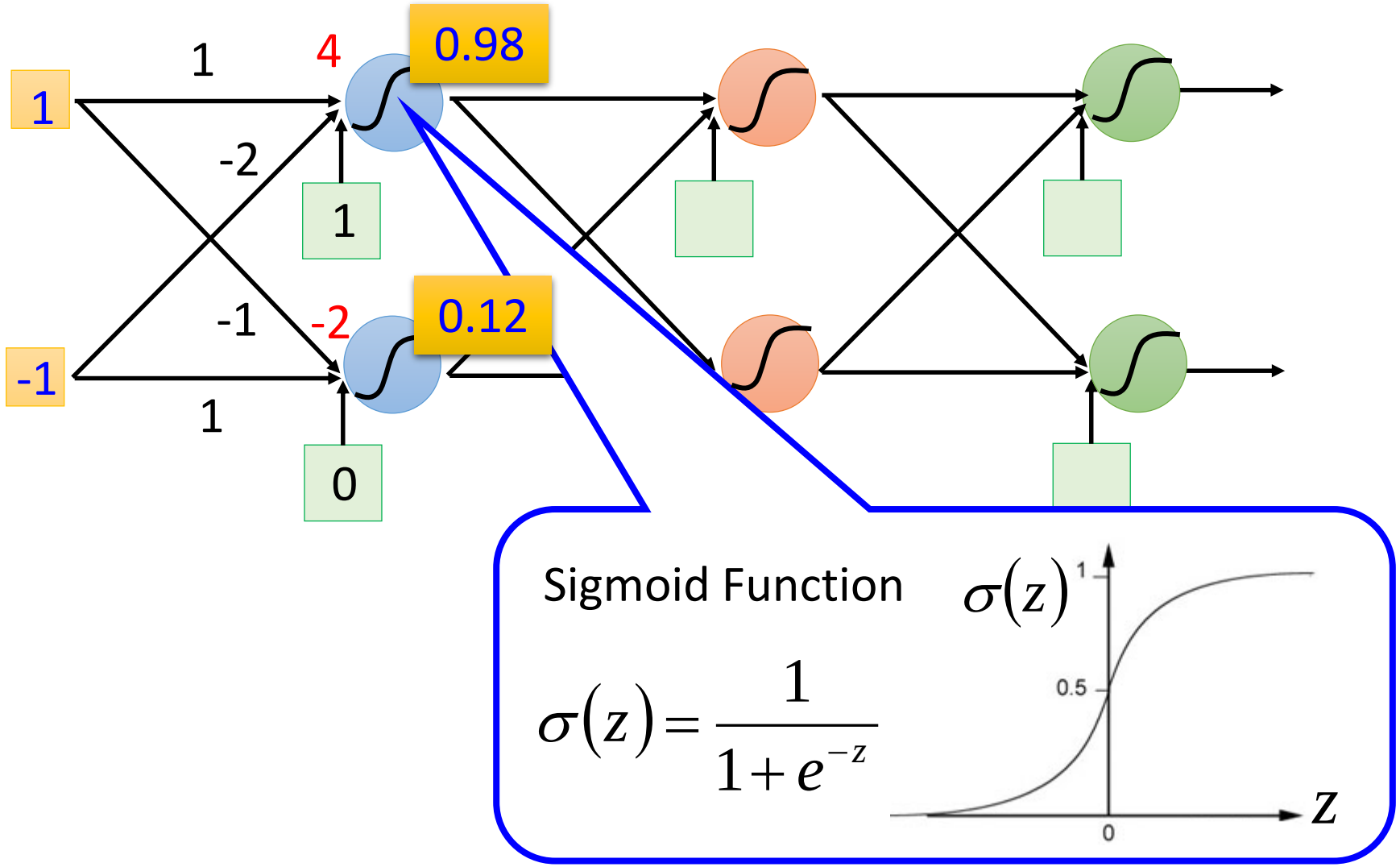


Neural Network

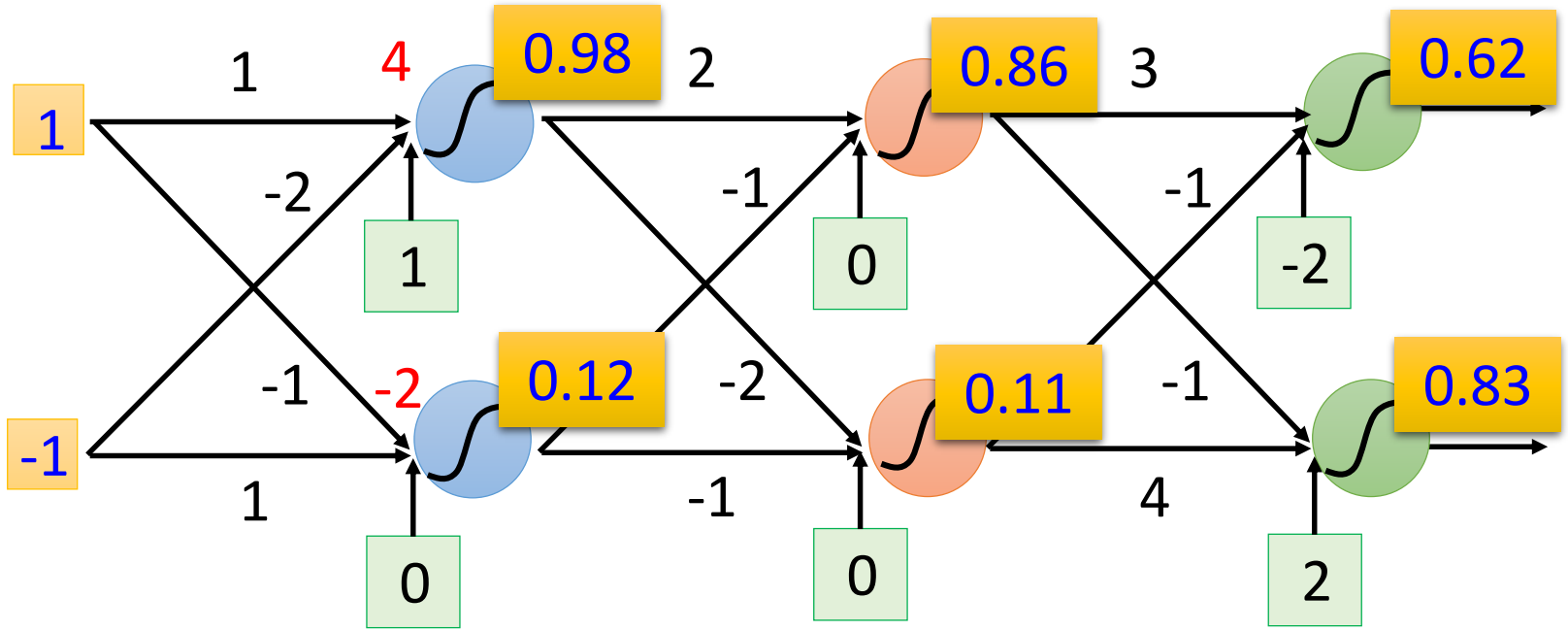


Deep means many hidden layers

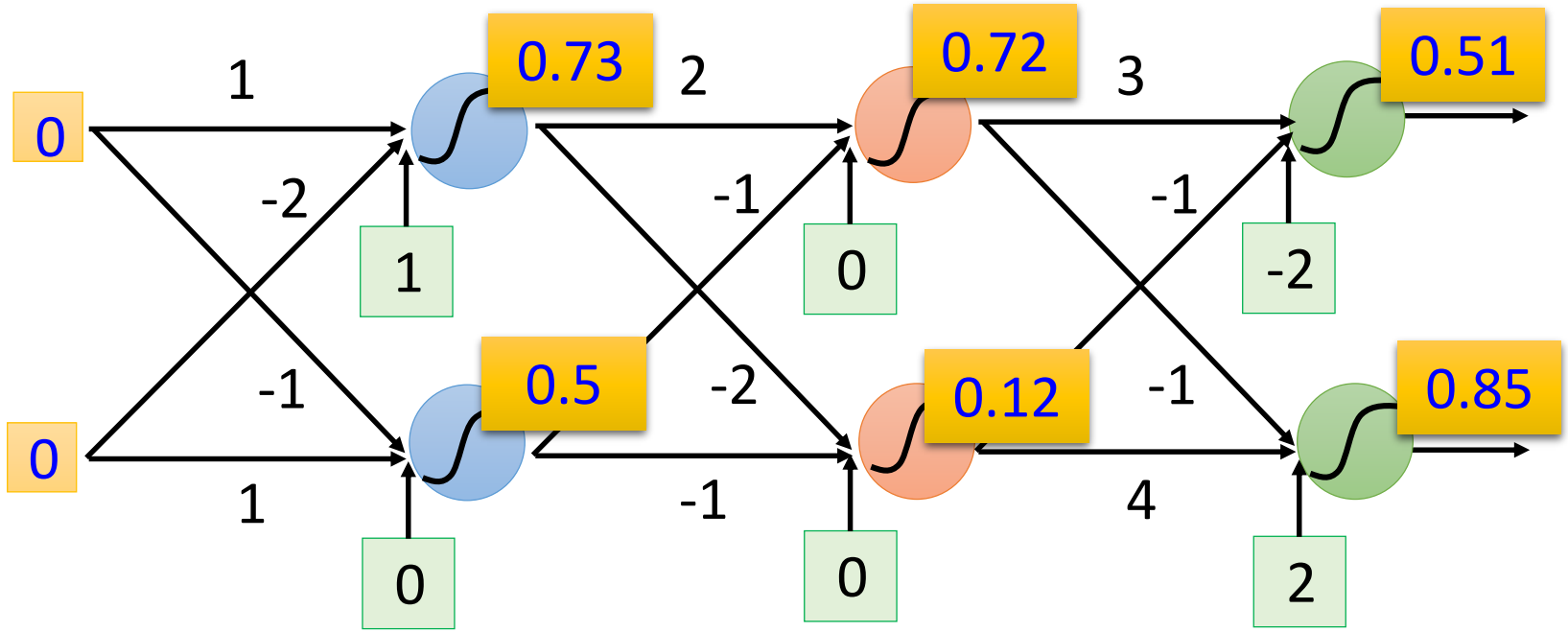
Example of Neural Network



Example of Neural Network



Example of Neural Network

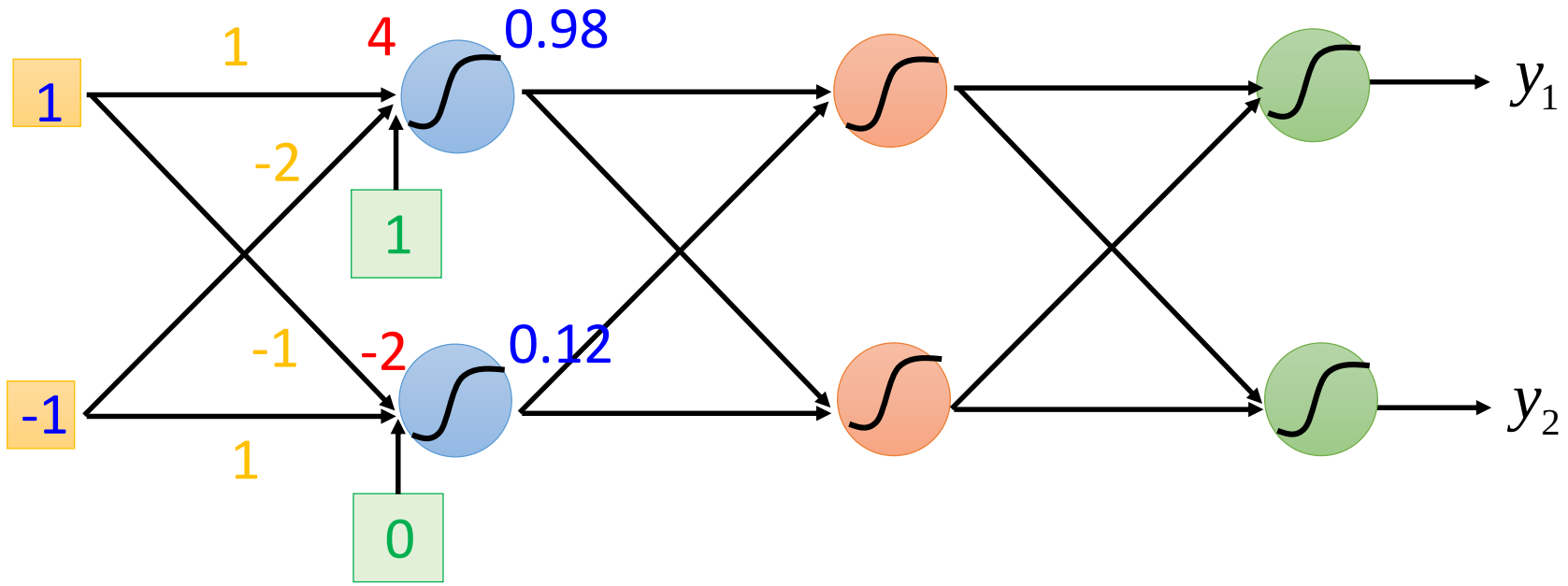


$f: R^2 \rightarrow R^2$

$$f\left(\begin{bmatrix}1\\-1\end{bmatrix}\right) = \begin{bmatrix}0.62\\0.83\end{bmatrix} \quad f\left(\begin{bmatrix}0\\0\end{bmatrix}\right) = \begin{bmatrix}0.51\\0.85\end{bmatrix}$$

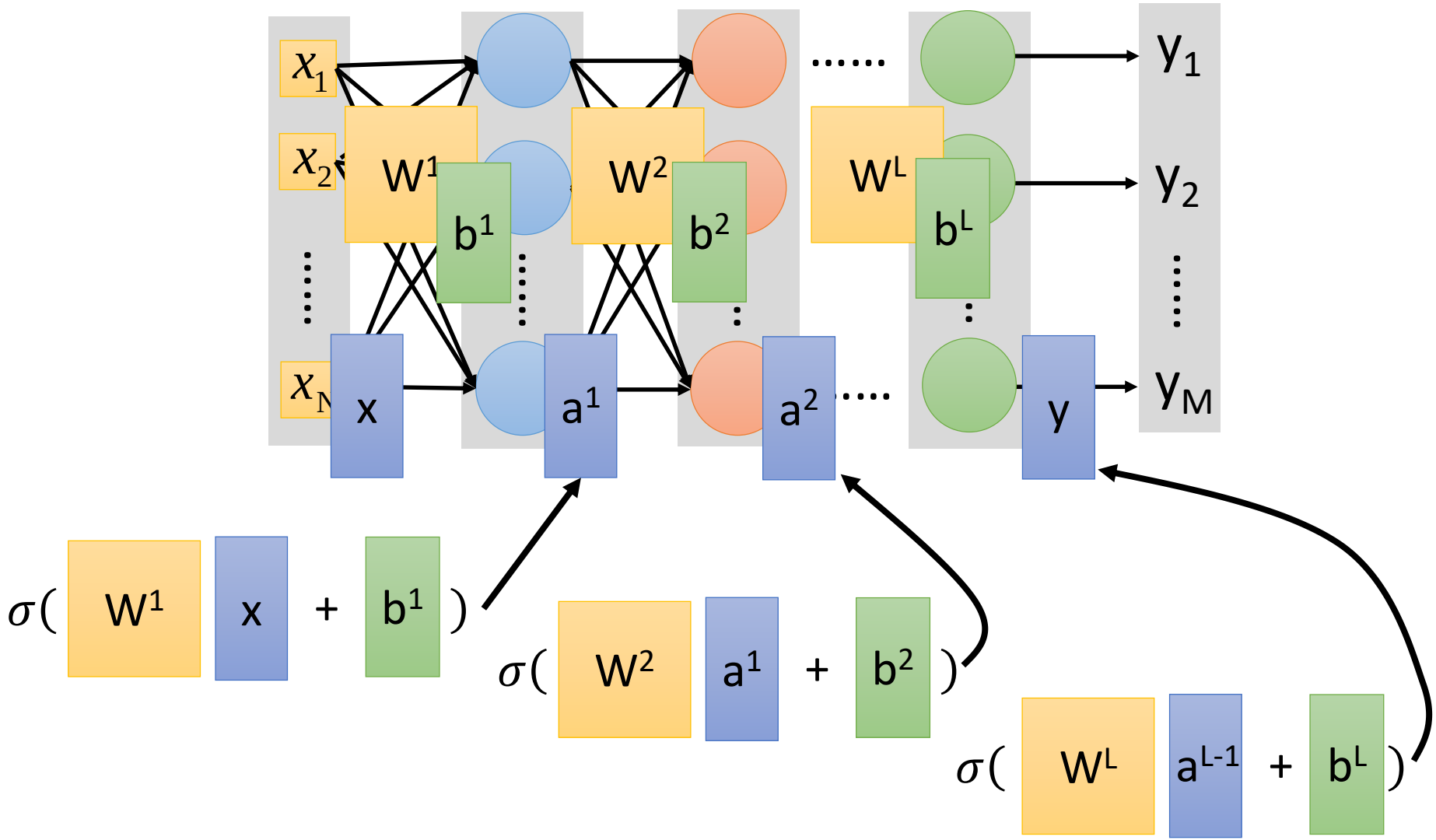
Different parameters define different function

Matrix Operation

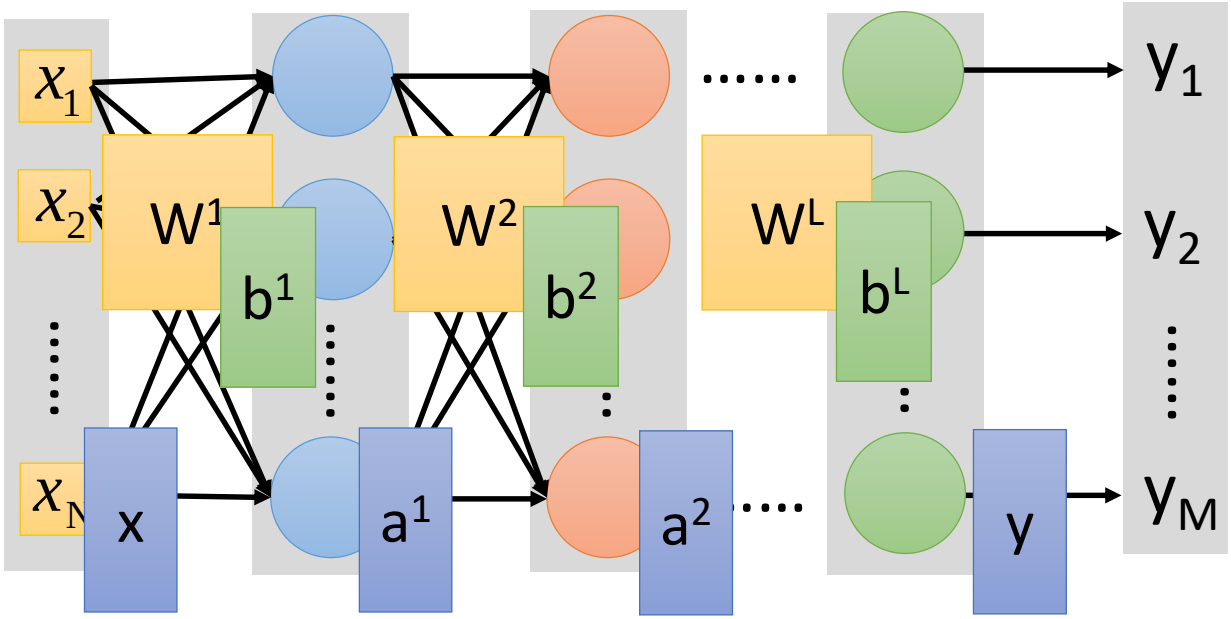


$$\sigma\left(\underbrace{\begin{bmatrix}1 & -2\\ -1 & 1\end{bmatrix}\begin{bmatrix}1\\ -1\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix}}_{\begin{bmatrix}4\\ -2\end{bmatrix}}\right)=\begin{bmatrix}0.98\\ 0.12\end{bmatrix}$$

Neural Network



Neural Network



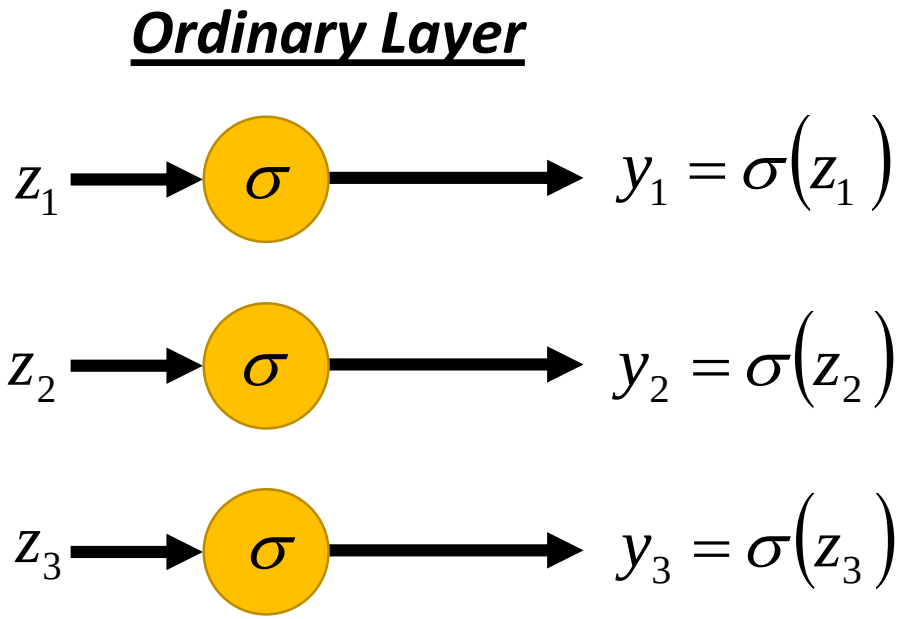
$$\mathbf{y} = f(\mathbf{x})$$

Using parallel computing techniques to speed up matrix operation

$$= \sigma(\mathbf{W}^L \cdots \sigma(\mathbf{W}^2 \sigma(\mathbf{W}^1 \mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2) \cdots + \mathbf{b}^L)$$

Softmax

- Softmax layer as the output layer



In general, the output of network can be any value.

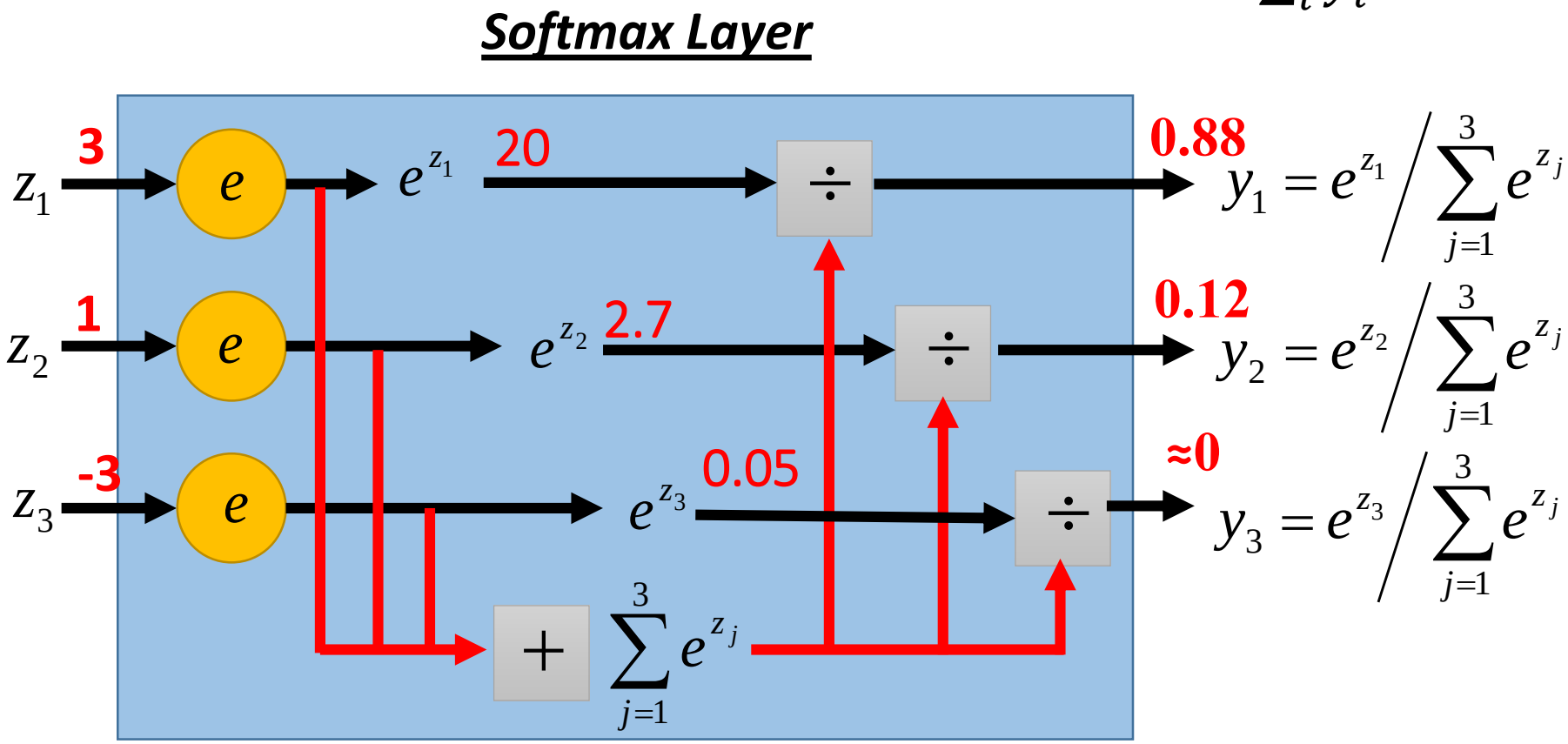
May not be easy to interpret

Softmax

- Softmax layer as the output layer

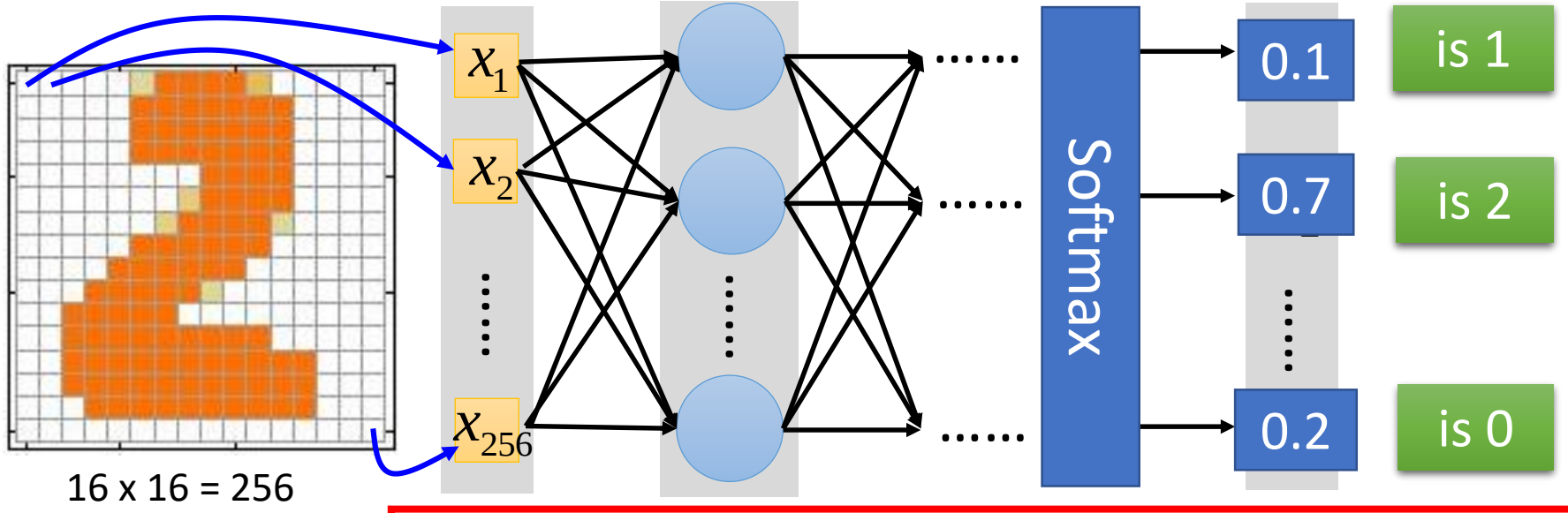
Probability:

- $1 > y_i > 0$
- $\sum_i y_i = 1$



How to set network parameters

$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$



16 x 16 = 256

Ink → 1

No ink → 0

Set the network parameters θ such that .....

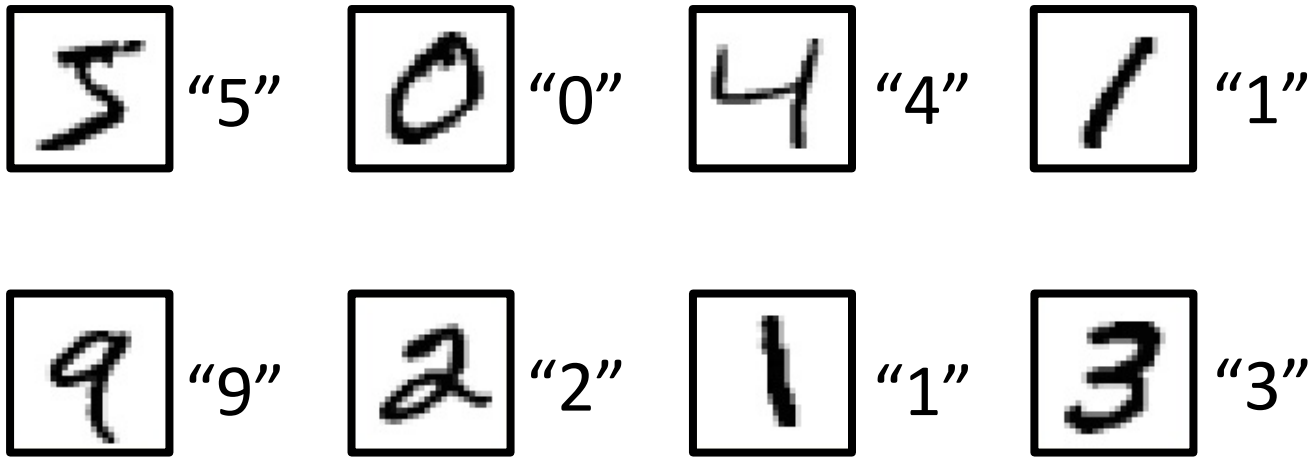
Input: m value

Input: y_2 has the maximum value

How to let the neural network achieve this

Training Data

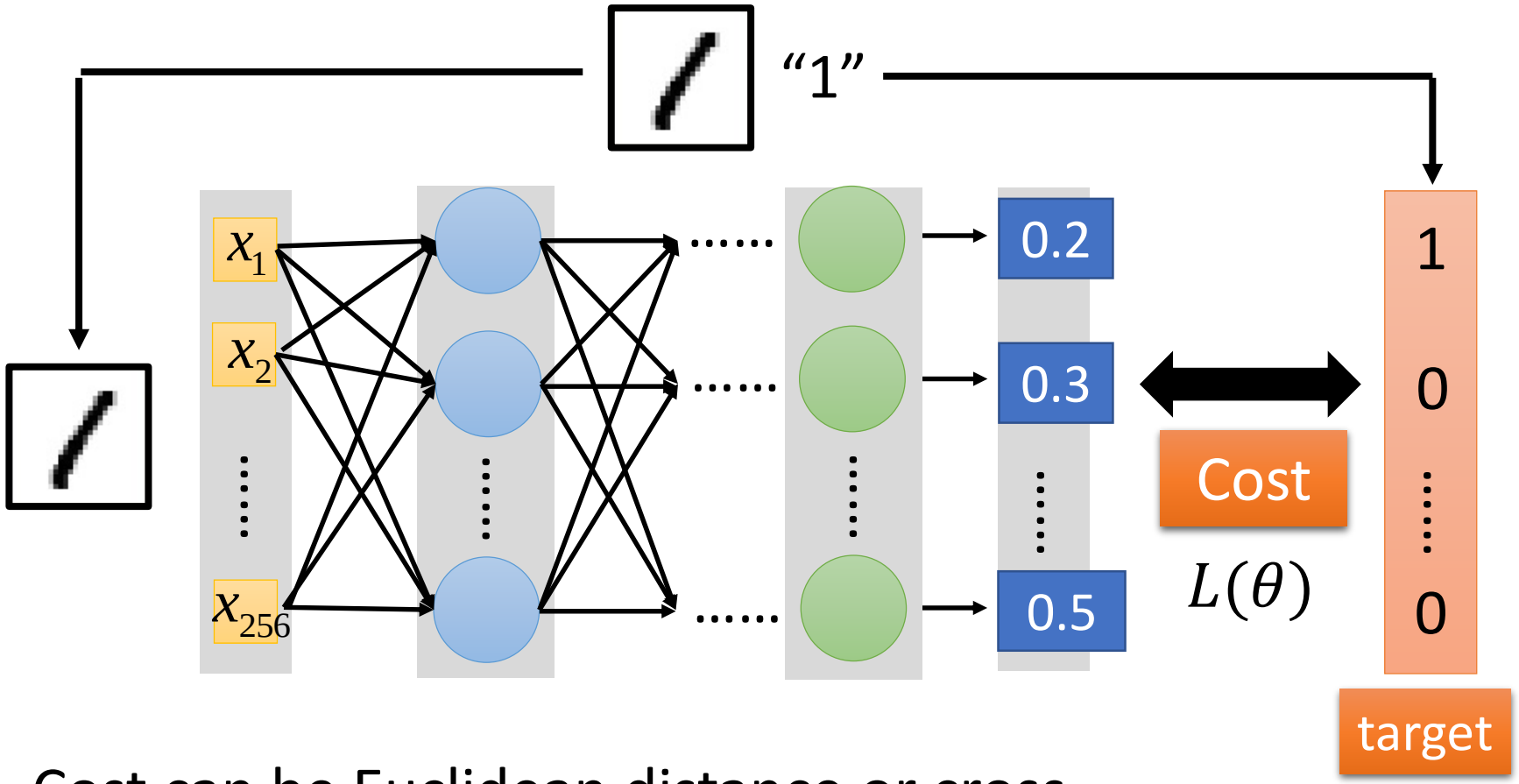
- Preparing training data: images and their labels



Using the training data to find the network parameters.

Cost

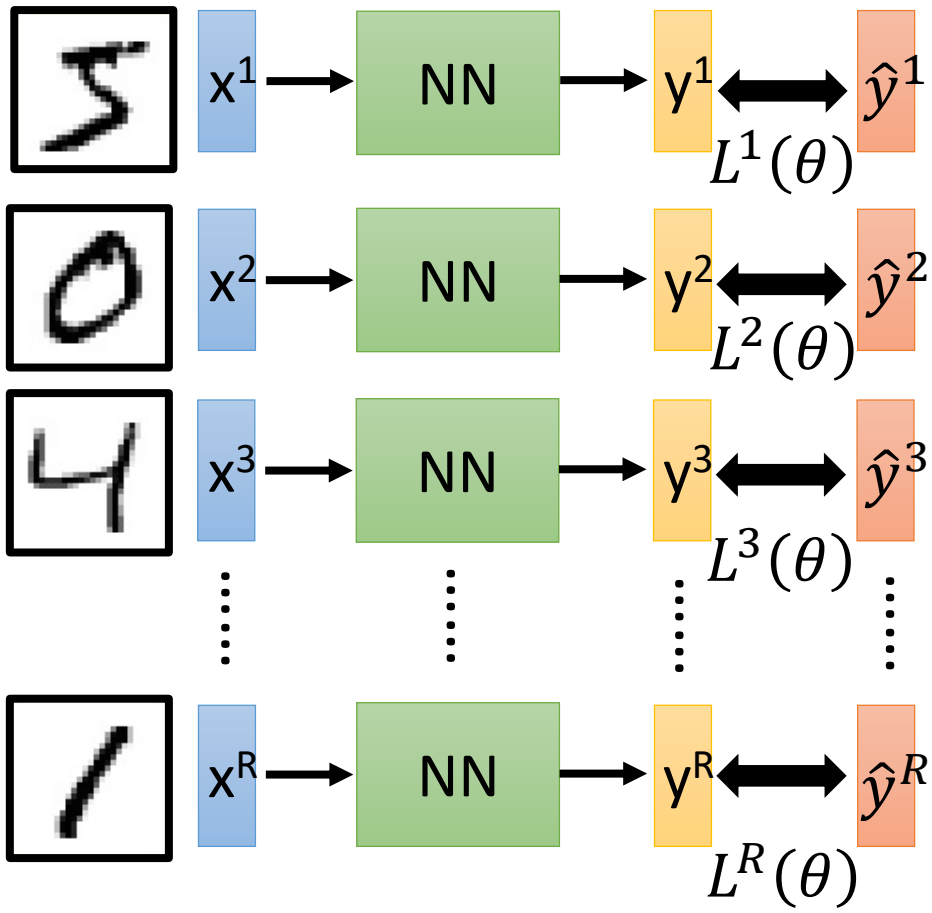
Given a set of network parameters θ, each example has a cost value.



Cost can be Euclidean distance or cross entropy of the network output and target

Total Cost

For all training data …



Total Cost:

$$C(\theta) = \sum_{r=1}^{R} L^r(\theta)$$

How bad the network parameters θ is on this task

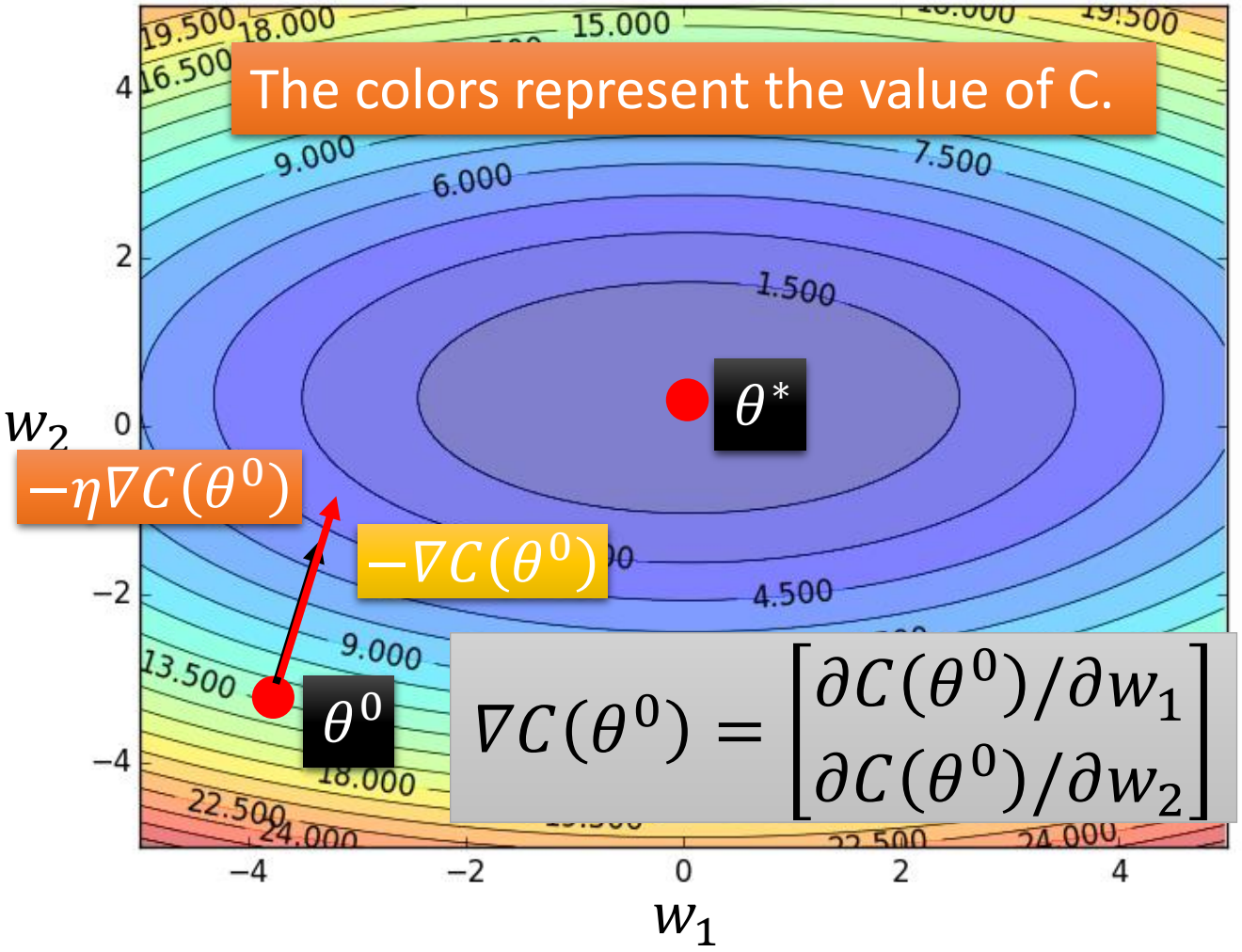
Find the network parameters θ^* that minimize this value

Gradient Descent

Assume there are only two parameters w_1 and w_2 in a network.

$$\theta = \{w_1, w_2\}$$

Error Surface



Randomly pick a starting point θ^0

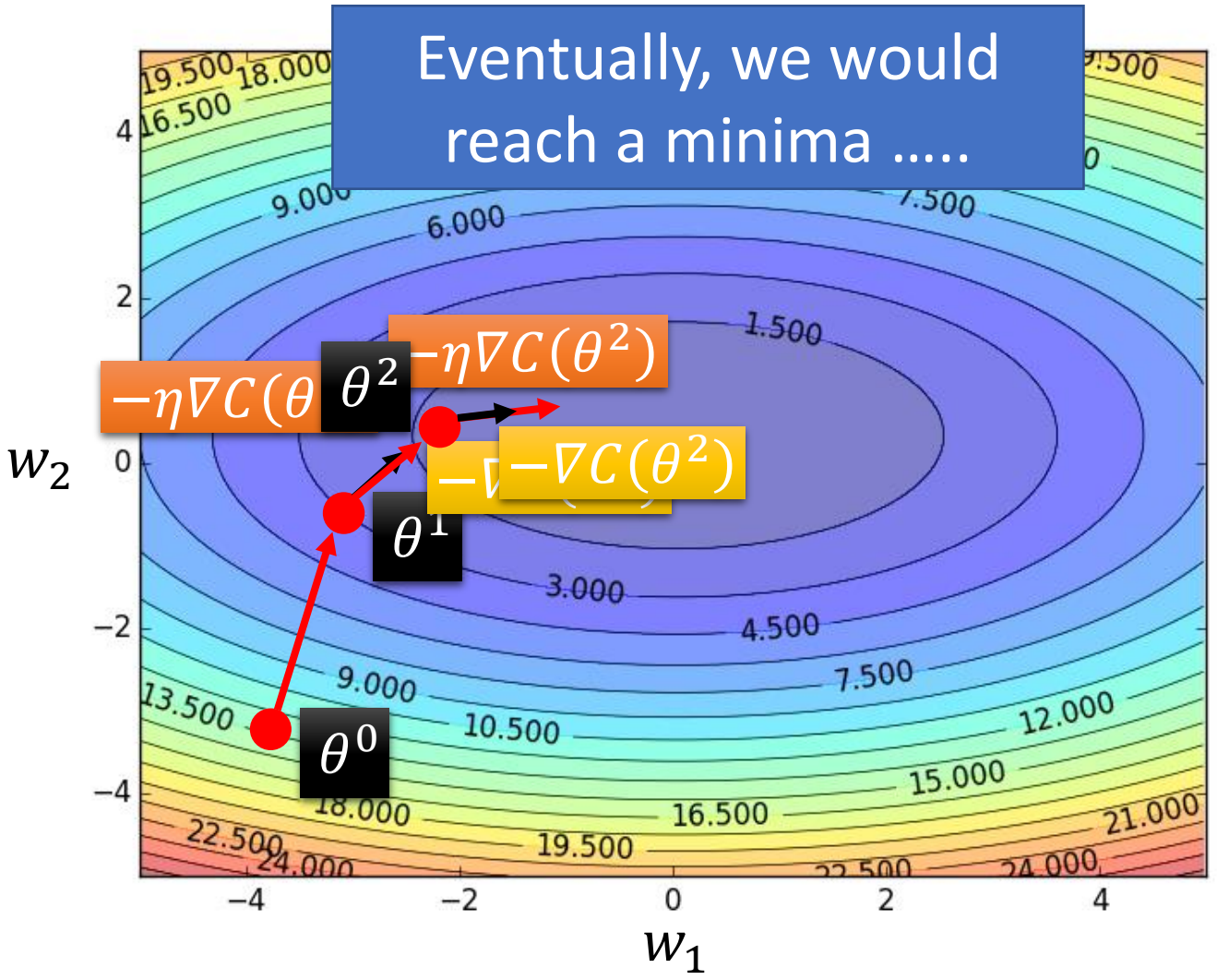
Compute the negative gradient at θ^0

$\rightarrow -\nabla C(\theta^0)$

Times the learning rate η

$\rightarrow -\eta\nabla C(\theta^0)$

Gradient Descent



Randomly pick a starting point θ^0

Compute the negative gradient at θ^0

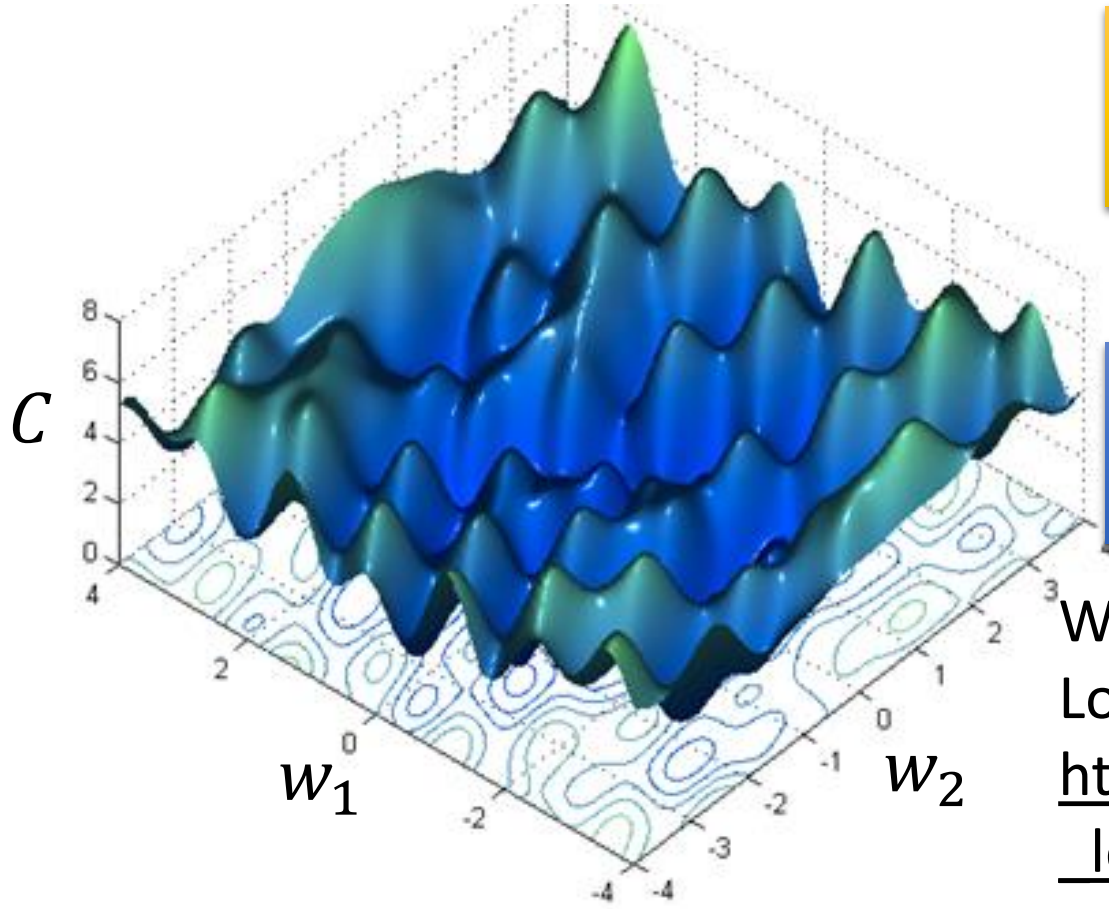
$\rightarrow -\nabla C(\theta^0)$

Times the learning rate η

$\rightarrow -\eta\nabla C(\theta^0)$

Local Minima

- Gradient descent never guarantee global minima

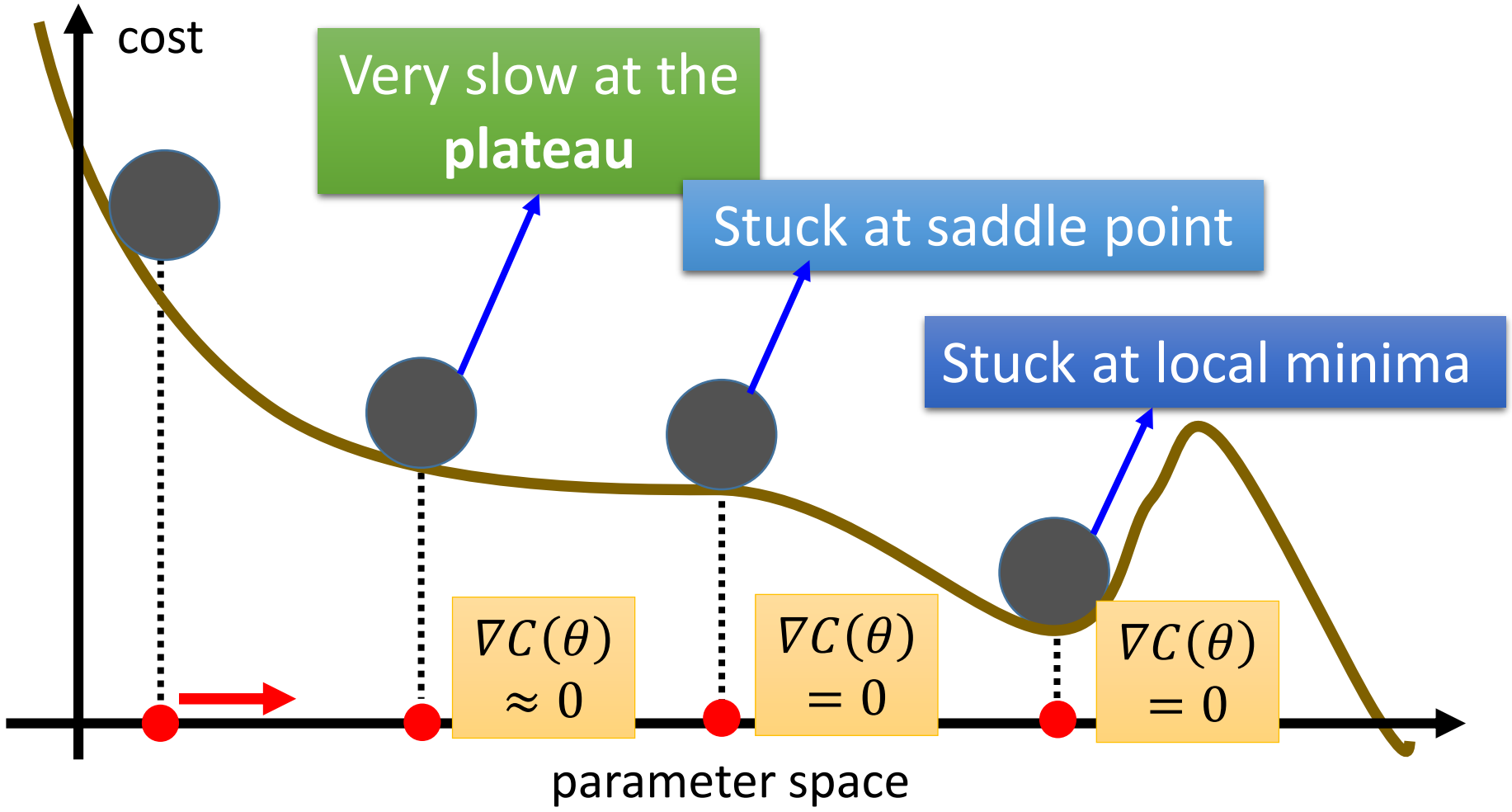


Different initial point θ^0

Reach different minima, so different results

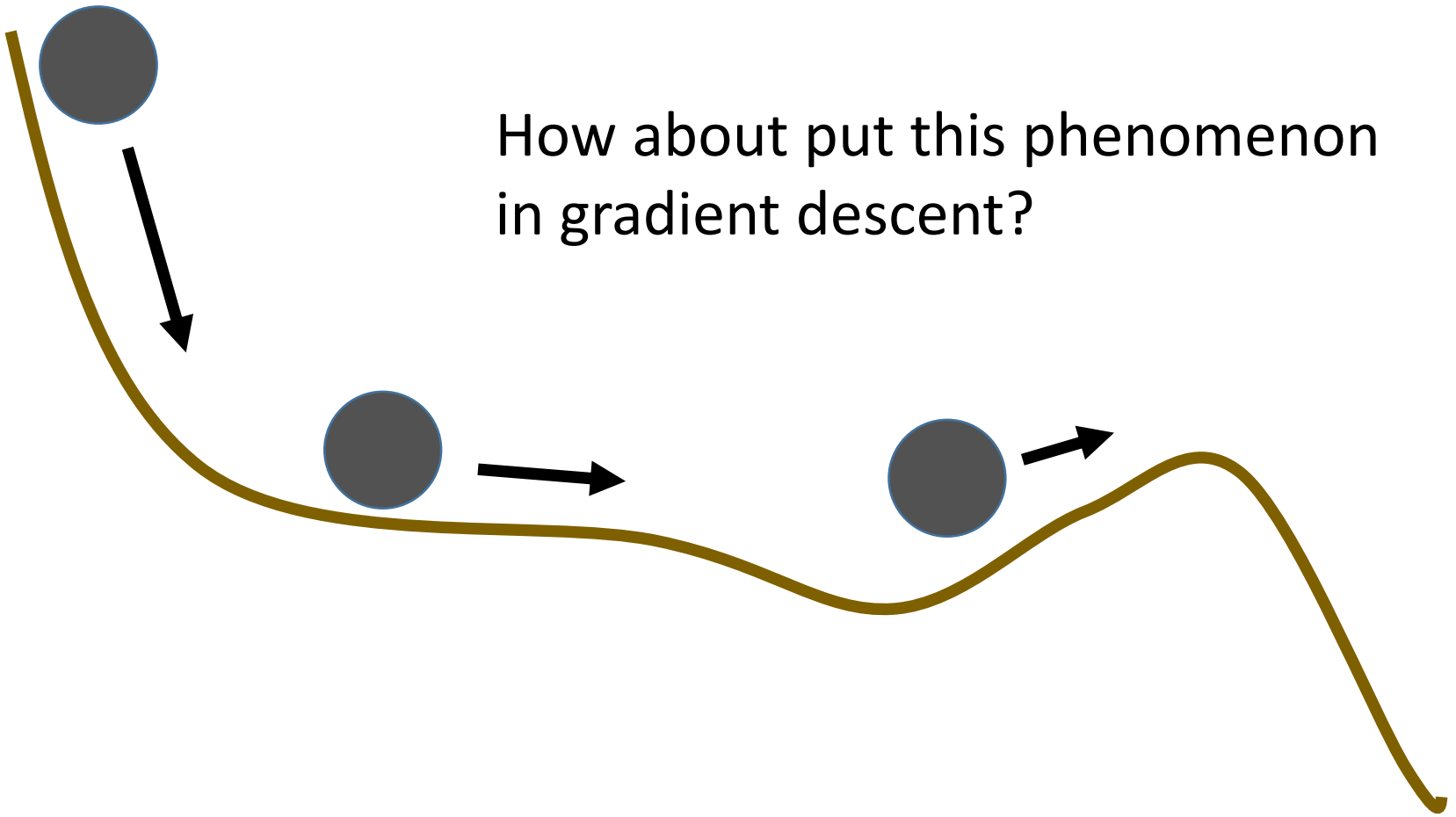
Who is Afraid of Non-Convex Loss Functions?
http://videolectures.net/eml07_lecun_wia/

Besides local minima ……



In physical world ……

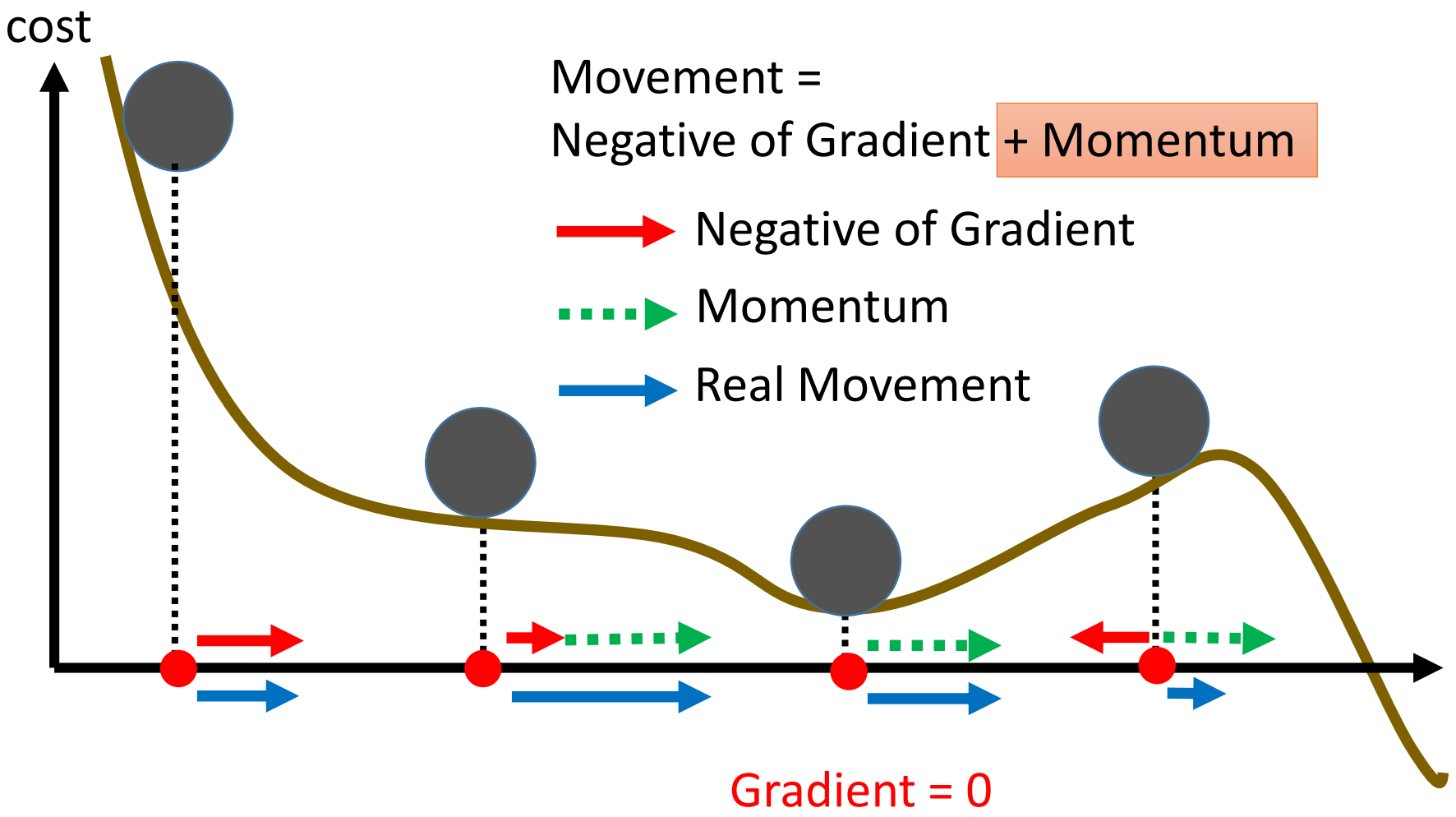
- Momentum



How about put this phenomenon in gradient descent?

Momentum

Still not guarantee reaching global minima, but give some hope ……



Improving Simple Gradient Descent

Momentum

Don't just change weights according to the current data point.

Re-use changes from earlier iterations.

Let $\Delta\mathbf{w}(t)$ = weight changes at time t.

Let $-\eta \frac{\partial \mathrm{E}}{\partial \mathbf{w}}$ be the change we would make with regular gradient descent.

Instead we use

$$\Delta\mathbf{w}(t+1) = -\eta \frac{\partial \mathrm{E}}{\partial \mathbf{w}} + \alpha\Delta\mathbf{w}(t)$$

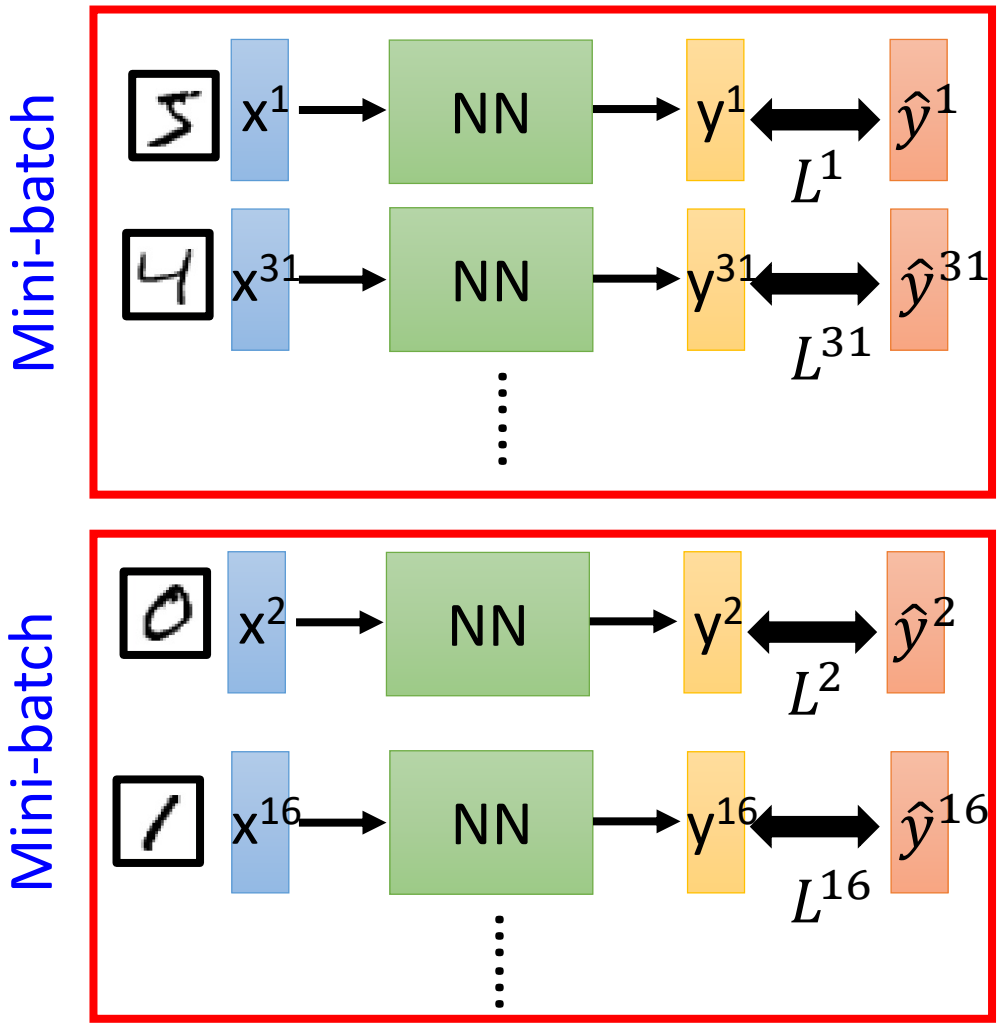
$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t)$$

momentum parameter

Momentum damps oscillations.

A hack? Well, maybe.

Mini-batch



- Randomly initialize θ^0
- Pick the 1st batch

$$C = L^1 + L^{31} + \cdots$$

$$\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$$

- Pick the 2nd batch

$$C = L^2 + L^{16} + \cdots$$

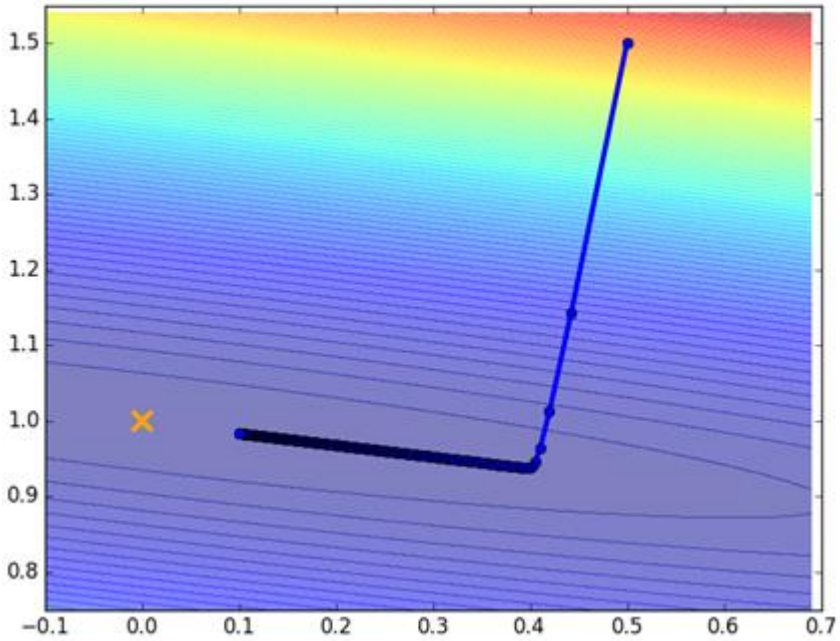
$$\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$$

$\vdots$

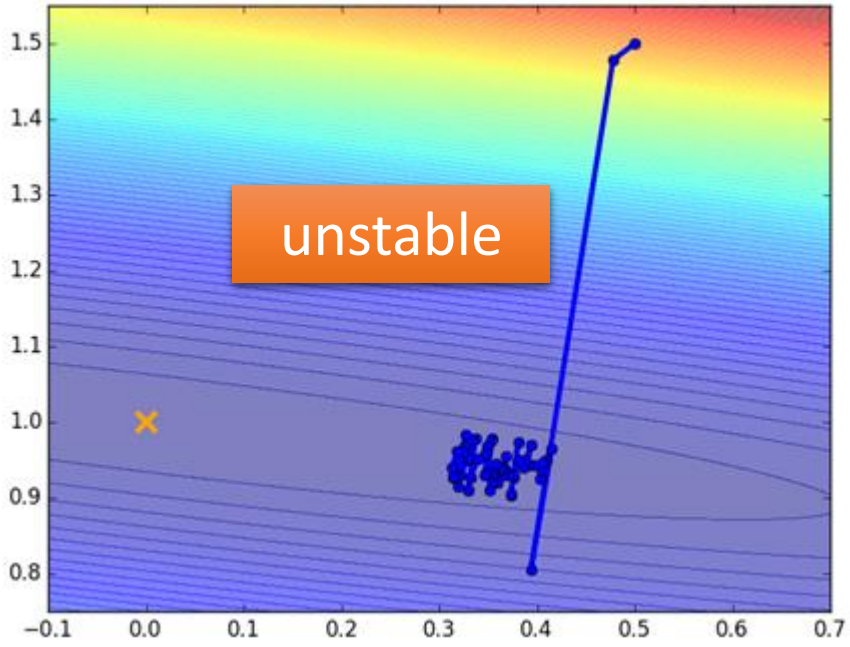
C is different each time when we update parameters!

Mini-batch

Original Gradient Descent



With Mini-batch

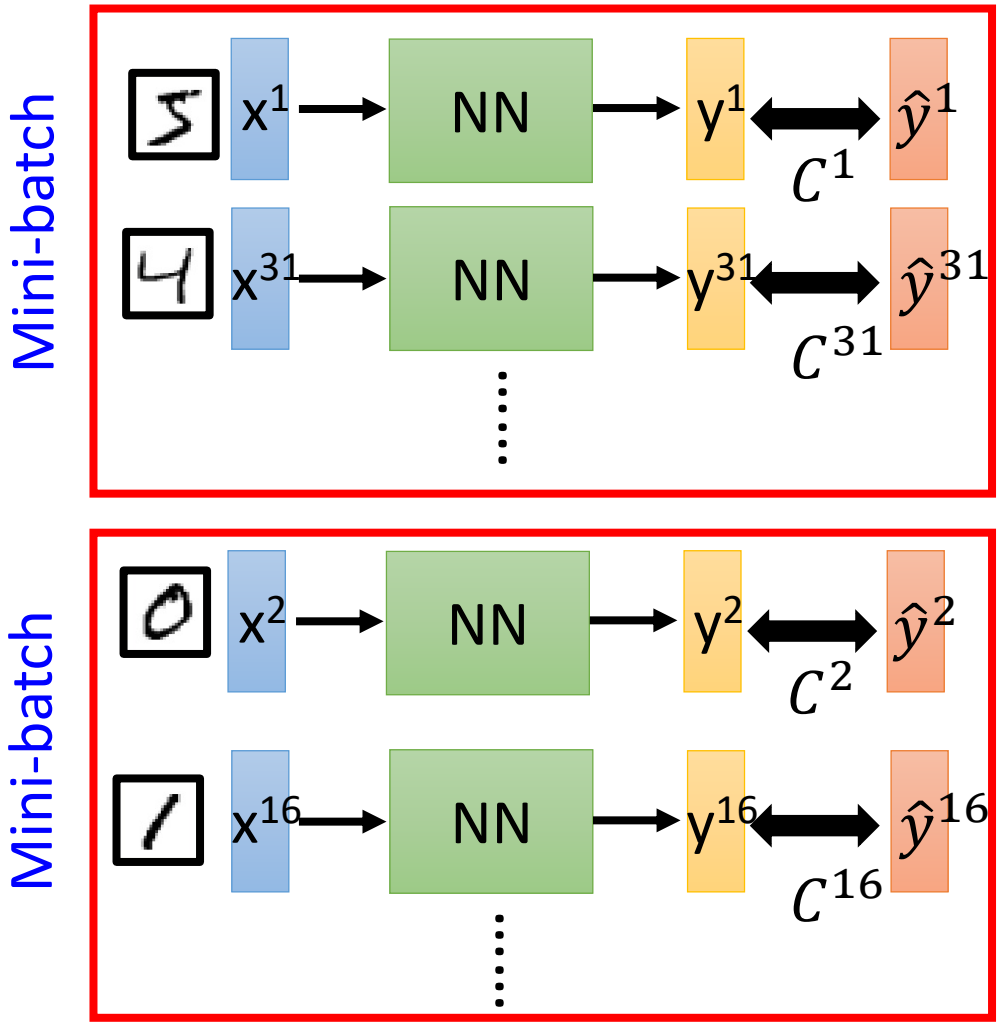


The colors represent the total C on all training data.

Mini-batch

Faster

Better!



- Randomly initialize θ^0
- Pick the 1st batch

  $C = C^1 + C^{31} + \cdots$

  $\theta^1 \leftarrow \theta^0 - \eta \nabla C(\theta^0)$
- Pick the 2nd batch

  $C = C^2 + C^{16} + \cdots$

  $\theta^2 \leftarrow \theta^1 - \eta \nabla C(\theta^1)$

  ⋮
- Until all mini-batches have been picked

one epoch

Repeat the above process

Backpropagation

- A network can have millions of parameters.
  - Backpropagation is the way to compute the gradients efficiently (not today)
  - Ref: http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20backprop.ecm.mp4/index.html
- Many toolkits can compute the gradients automatically

theano

TensorFlow

Ref: http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/Theano%20DNN.ecm.mp4/index.html

Linear Perceptrons

They are multivariate linear models:

$$\text{Out}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$$

And "training" consists of minimizing sum-of-squared residuals by gradient descent.

$$\begin{aligned} E &= \sum_k (\text{Out}(\mathbf{x}_k) - y_k)^2 \\ &= \sum_k (\mathbf{w}^T\mathbf{x}_k - y_k)^2 \end{aligned}$$

QUESTION: Derive the perceptron training rule.

Linear Perceptron Training Rule

$$E = \sum_{k=1}^{R} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

So what's $\frac{\partial E}{\partial w_j}$?

Linear Perceptron Training Rule

$$E = \sum_{k=1}^{R} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

So what's $\frac{\partial E}{\partial w_j}$?

$$\frac{\partial E}{\partial w_j} = \sum_{k=1}^{R} \frac{\partial}{\partial w_j} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

$$= \sum_{k=1}^{R} 2(y_k - \mathbf{w}^T \mathbf{x}_k) \frac{\partial}{\partial w_j} (y_k - \mathbf{w}^T \mathbf{x}_k)$$

$$= -2 \sum_{k=1}^{R} \delta_k \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}_k$$

...where...

$$\delta_k = y_k - \mathbf{w}^T \mathbf{x}_k$$

$$= -2 \sum_{k=1}^{R} \delta_k \frac{\partial}{\partial w_j} \sum_{i=1}^{m} w_i x_{ki}$$

$$= -2 \sum_{k=1}^{R} \delta_k x_{kj}$$

Linear Perceptron Training Rule

$$E = \sum_{k=1}^{R} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

...where...

$$\frac{\partial E}{\partial w_j} = -2 \sum_{k=1}^{R} \delta_k x_{kj}$$

$$w_j \leftarrow w_j + 2\eta \sum_{k=1}^{R} \delta_k x_{kj}$$

We frequently neglect the 2 (meaning we halve the learning rate)

The "Batch" perceptron algorithm

1) Randomly initialize weights $w_1\ w_2\ \ldots\ w_m$

2) Get your dataset (append 1's to the inputs if you don't want to go through the origin).

3) for $i = 1$ to R

$$\delta_i := y_i - \mathbf{w}^{\mathrm{T}}\mathbf{x}_i$$

4) for $j = 1$ to m

$$w_j \leftarrow w_j + \eta \sum_{i=1}^{R} \delta_i x_{ij}$$

5) if $\sum \delta_i^{\,2}$ stops improving then stop. Else loop back to 3.

$$\delta_i \leftarrow y_i - \mathbf{w}^{\mathrm{T}}\mathbf{x}_i$$

$$w_j \leftarrow w_j + \eta\delta_i x_{ij}$$

A RULE KNOWN BY MANY NAMES

The LMS Rule

The delta rule

The Widrow Hoff rule

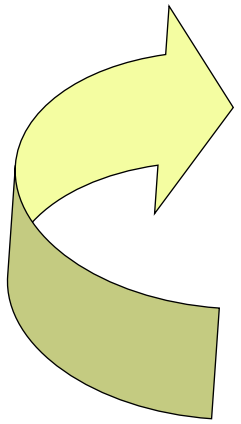
The adaline rule

Classical conditioning

If data is voluminous and arrives fast

Input-output pairs (***x***,*y*) come streaming in very quickly. THEN
Don't bother remembering old ones.
Just keep using new ones.

observe (**x**,*y*)



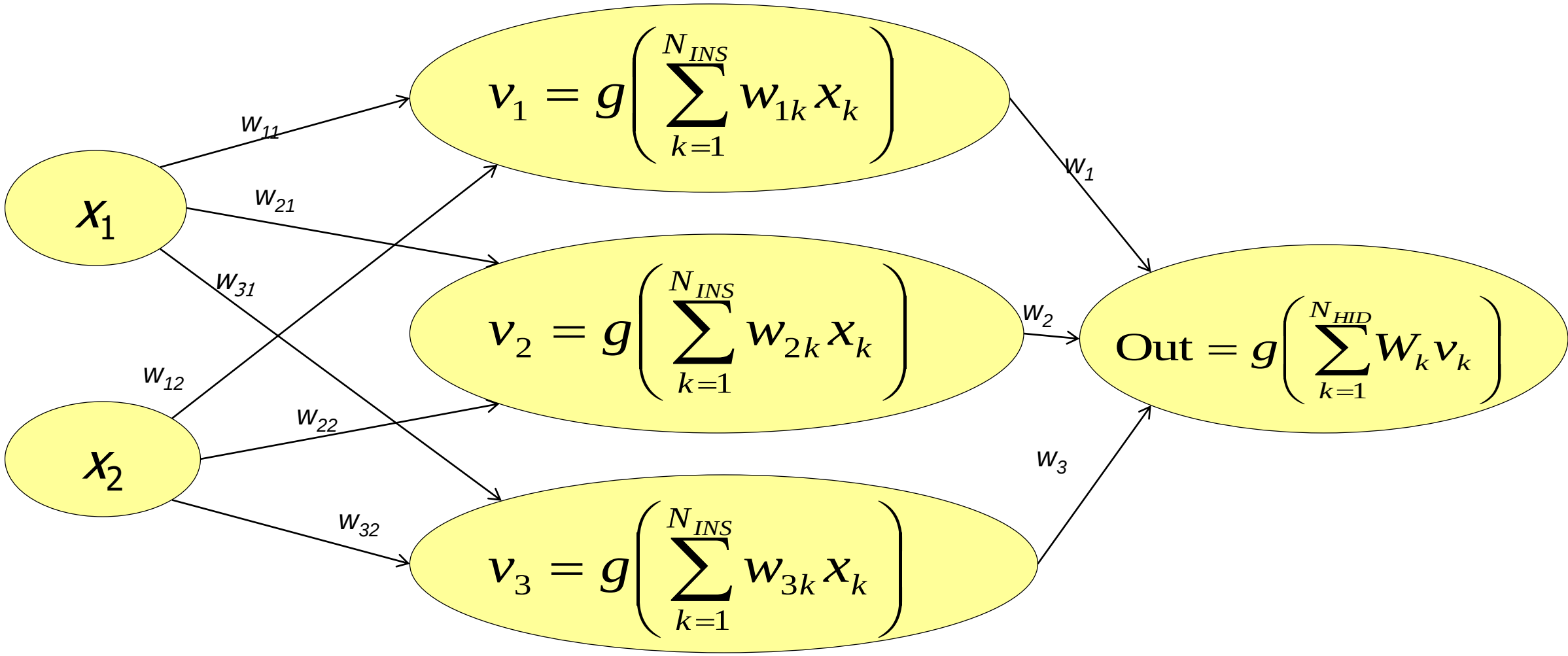
$$\delta \leftarrow y - \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

$$\forall j \;\; w_j \leftarrow w_j + \eta\,\delta\, x_j$$

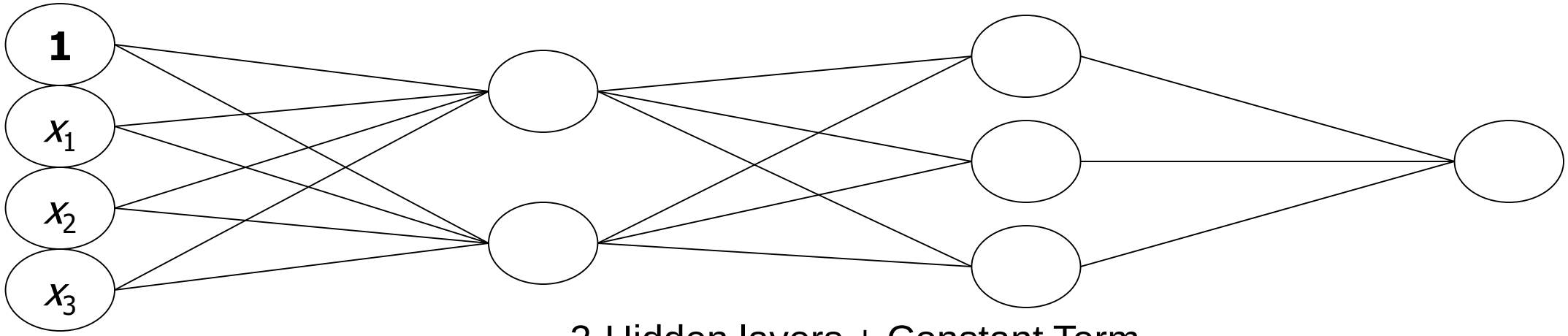
A 1-HIDDEN LAYER NET

$N_{INPUTS} = 2$

$N_{HIDDEN} = 3$

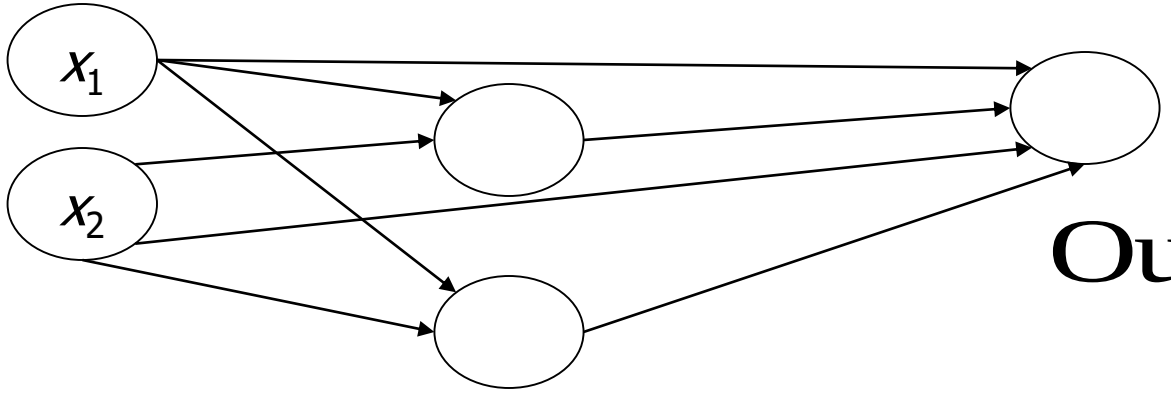


OTHER NEURAL NETS



2-Hidden layers + Constant Term

"JUMP" CONNECTIONS

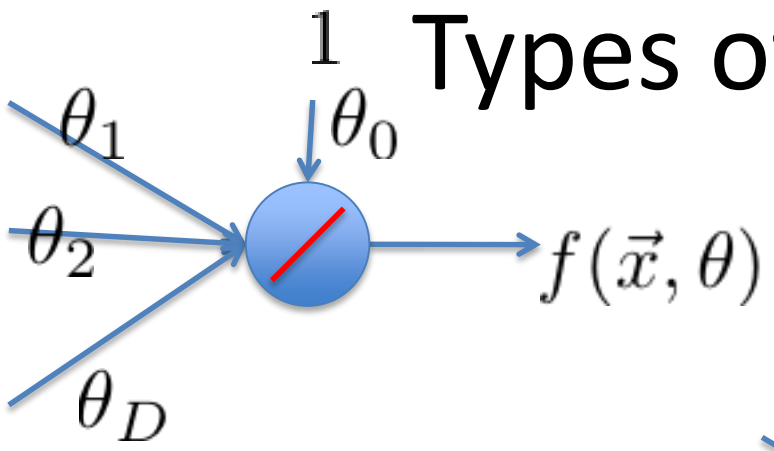


$$\text{Out} = g\left(\sum_{k=1}^{N_{INS}} w_{0k} x_k + \sum_{k=1}^{N_{HID}} W_k v_k\right)$$

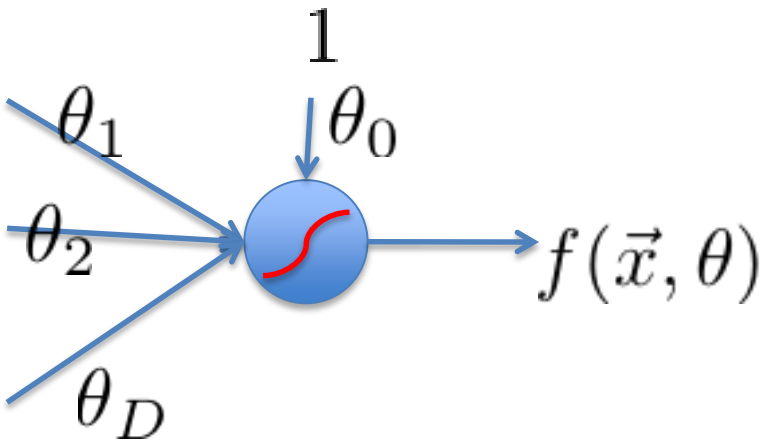
Backprop

- Very powerful - can learn any function, given enough hidden units! With enough hidden units, we can generate any function.
- Have the same problems of Generalization vs. Memorization. With too many units, we will tend to memorize the input and not generalize well. Some schemes exist to “prune” the neural network.
- Networks require extensive training, many parameters to fiddle with. Can be extremely slow to train. May also fall into local minima.
- Inherently parallel algorithm, ideal for multiprocessor hardware.
- Despite the cons, a very powerful algorithm that has seen widespread successful deployment.

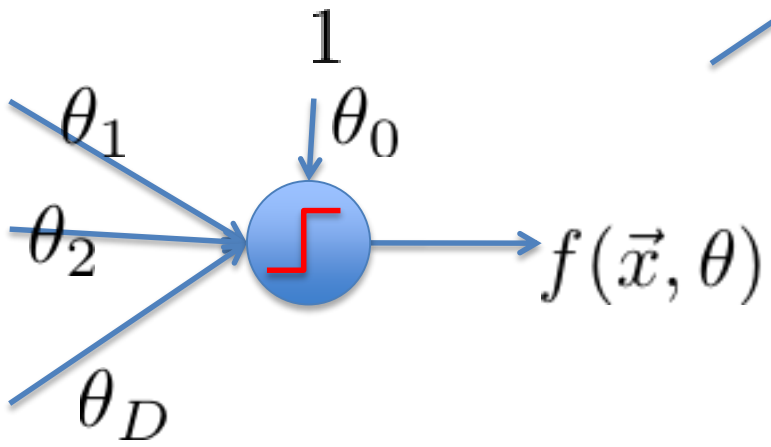
Types of Neurons



Linear Neuron



Logistic Neuron

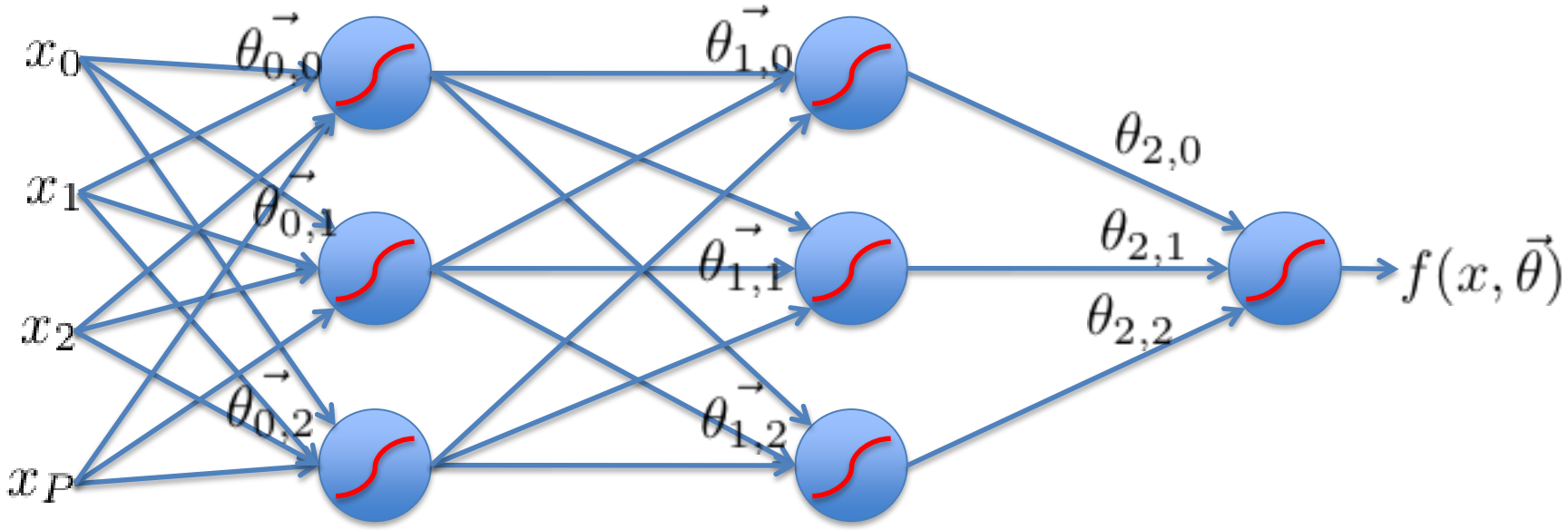


Perceptron

Potentially more. Require a convex loss function for gradient descent training.

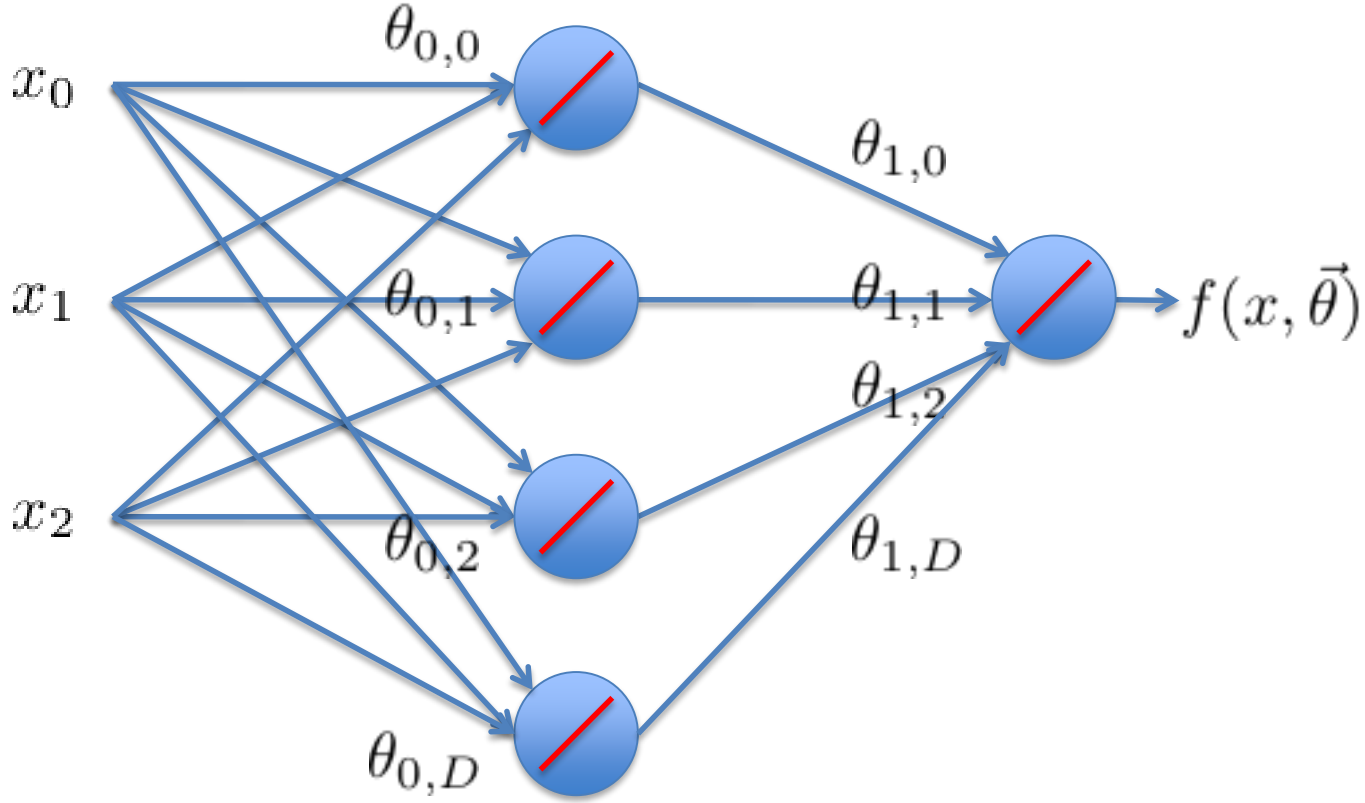
Multilayer Networks

- Cascade Neurons together
- The output from one layer is the input to the next
- Each Layer has its own sets of weights



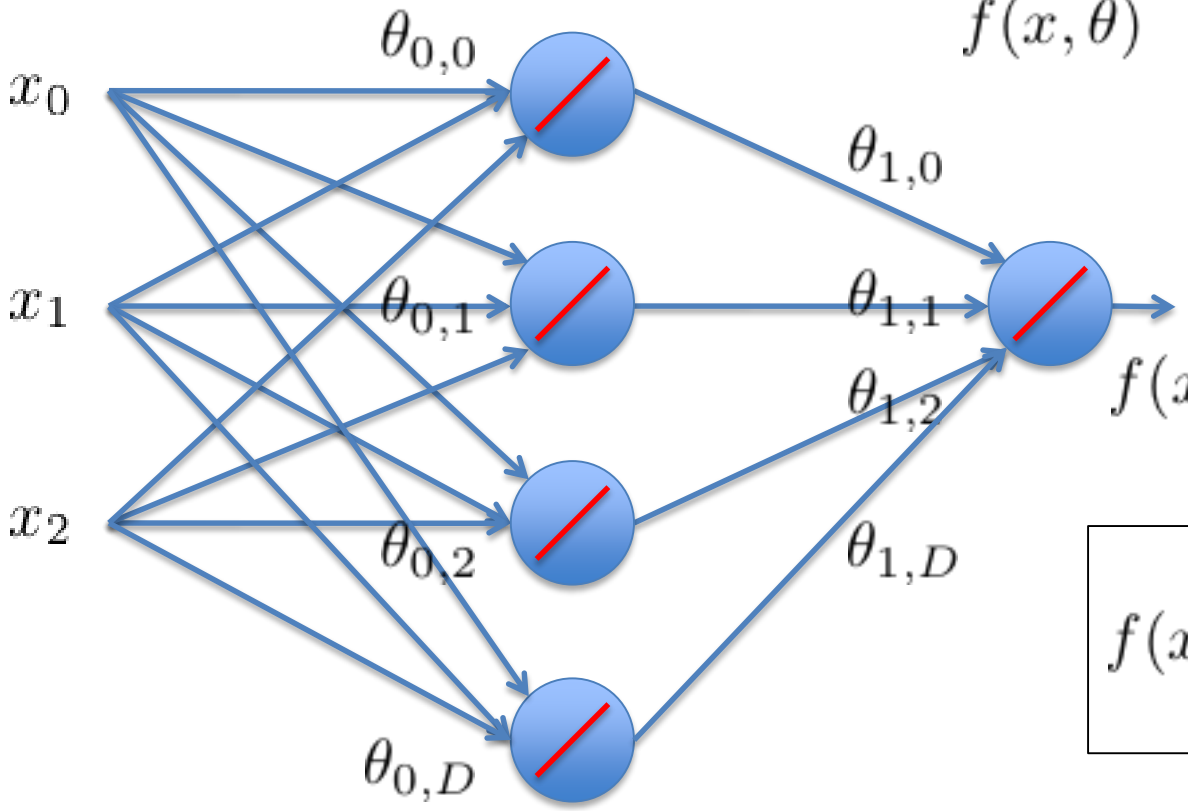
Linear Regression Neural Networks

- What happens when we arrange **linear neurons** in a multilayer network?



Linear Regression Neural Networks

- Nothing special happens.
  - The product of two linear transformations is itself a linear transformation.



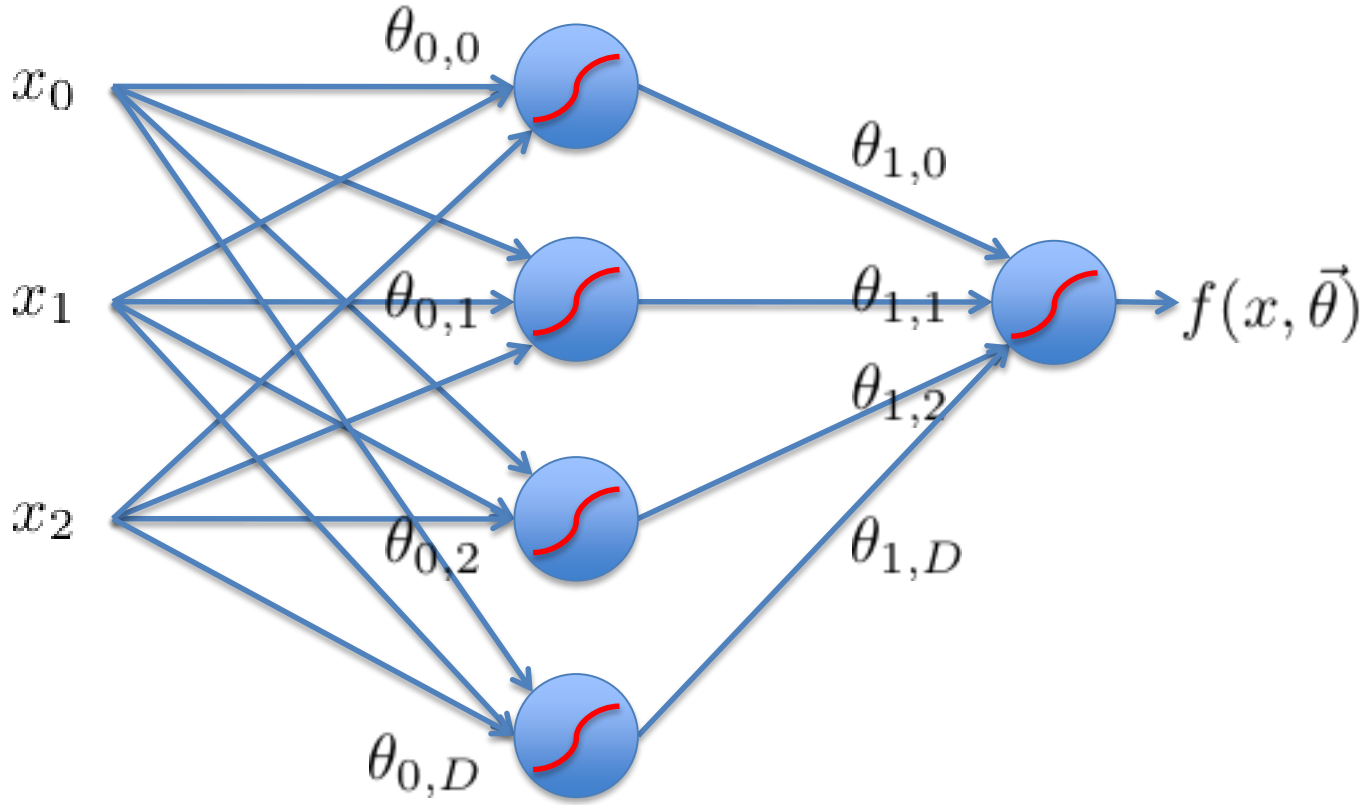
$$f(x,\vec{\theta}) = \sum_{i=0}^{D} \theta_{1,i} \sum_{n=0}^{N-1} \theta_{0,i,n} x_n$$

$$f(x,\vec{\theta}) = \sum_{i=0}^{D} \theta_{1,i} [\theta_{0,i}^T \vec{x}]$$

$$\boxed{f(x,\vec{\theta}) = \sum_{i=0}^{D} [\hat{\theta}_i^T \vec{x}]}$$

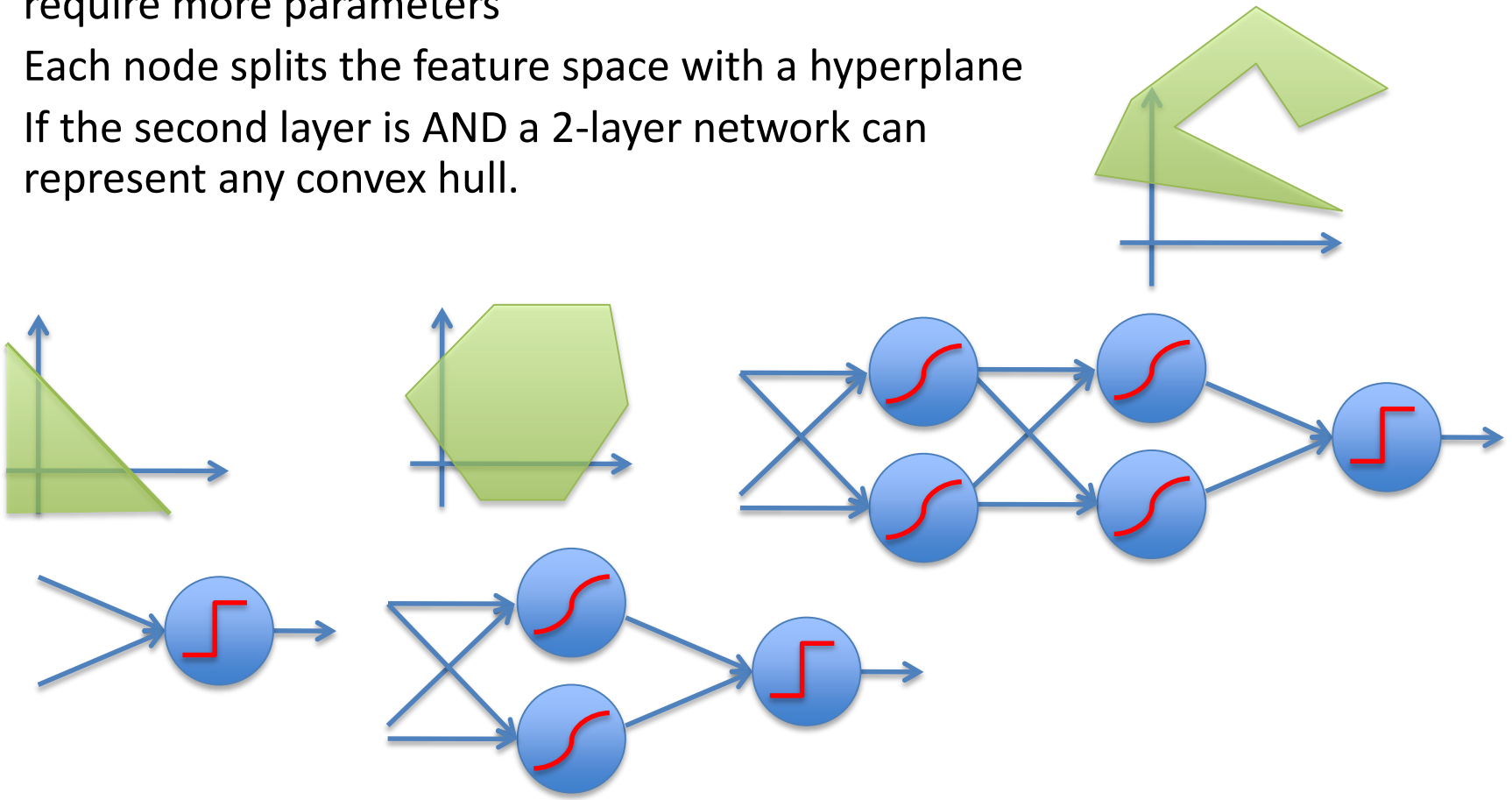
Neural Networks

- We want to introduce non-linearities to the network.
  - Non-linearities allow a network to identify complex regions in space



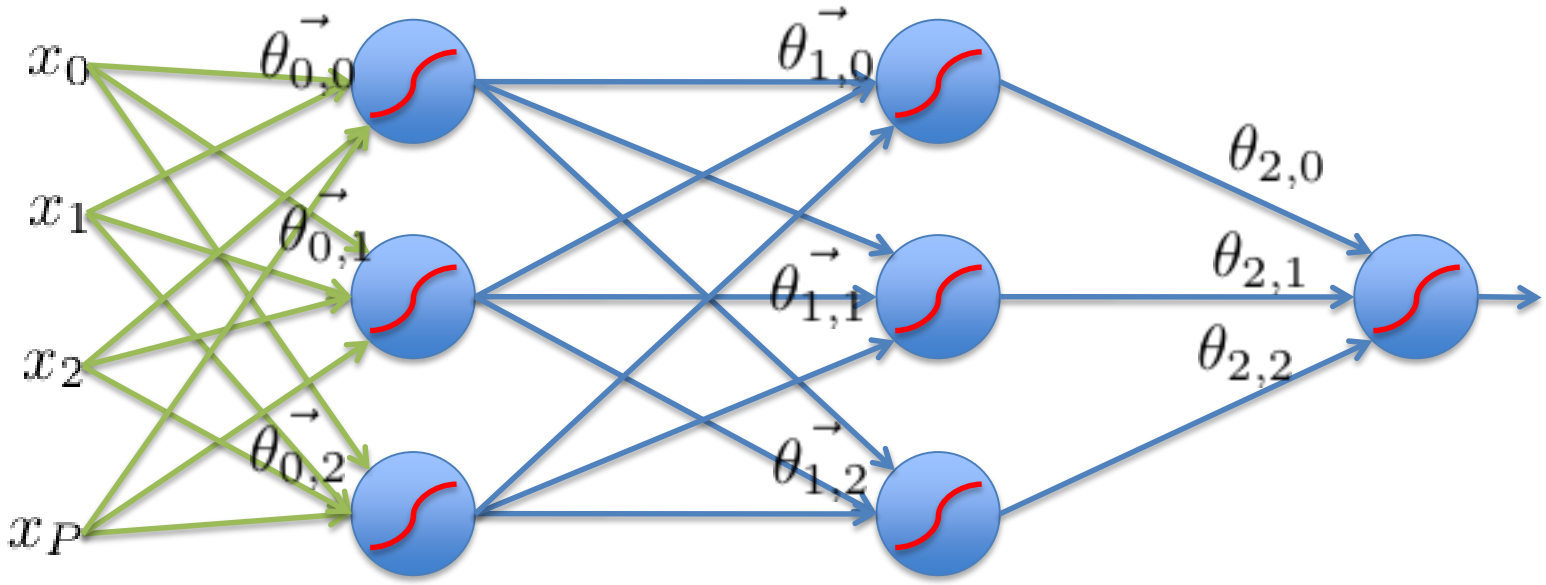
Linear Separability

- 1-layer cannot handle XOR
- More layers can handle more complicated spaces – but require more parameters
- Each node splits the feature space with a hyperplane
- If the second layer is AND a 2-layer network can represent any convex hull.



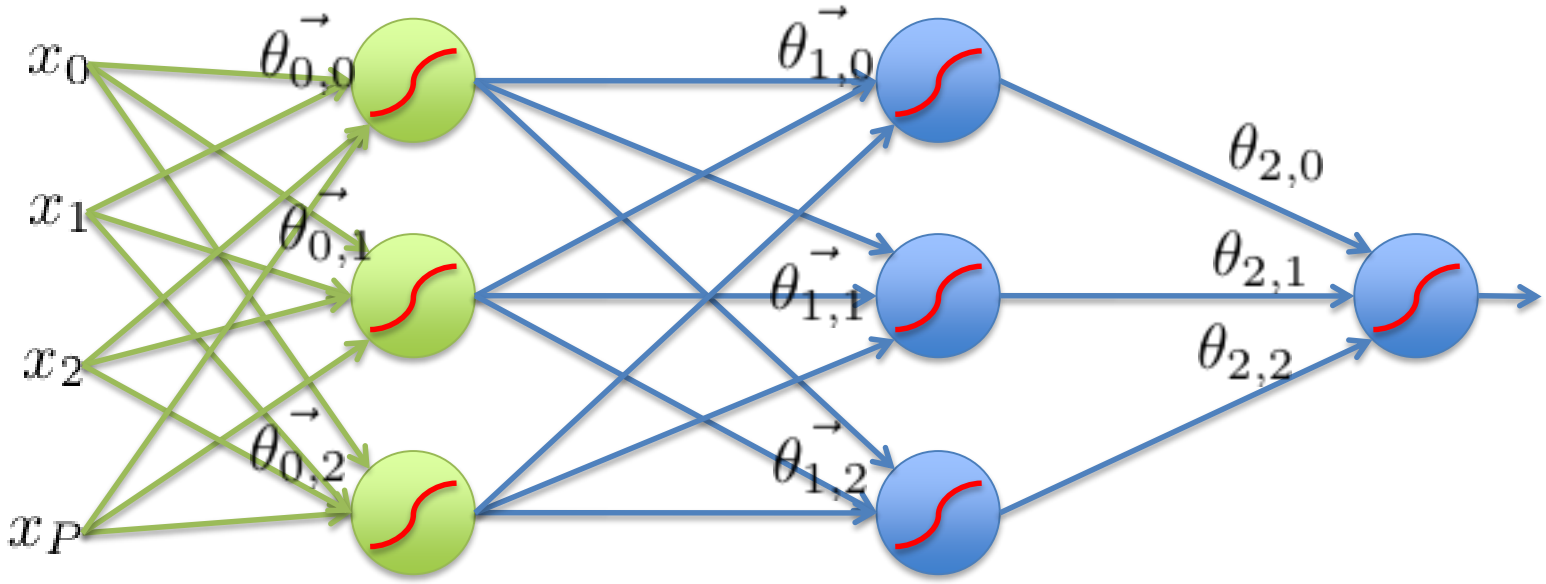
Feed-Forward Networks

- Predictions are fed forward through the network to classify



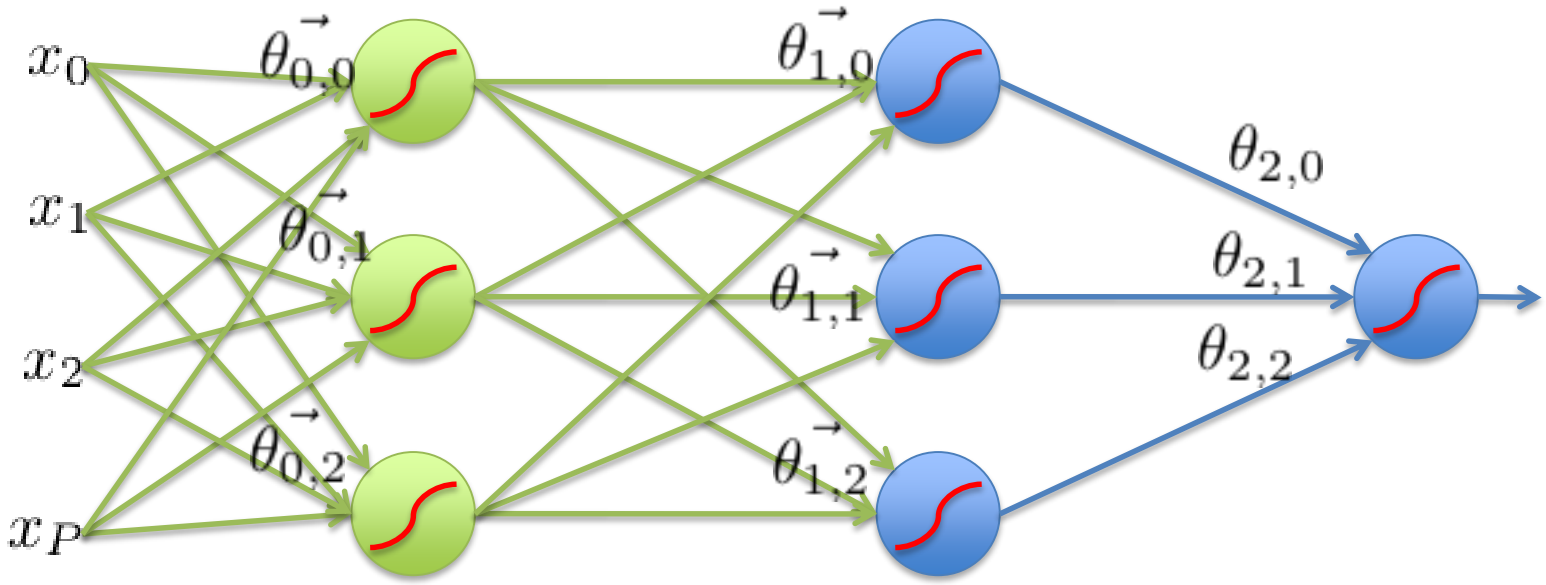
Feed-Forward Networks

- Predictions are fed forward through the network to classify



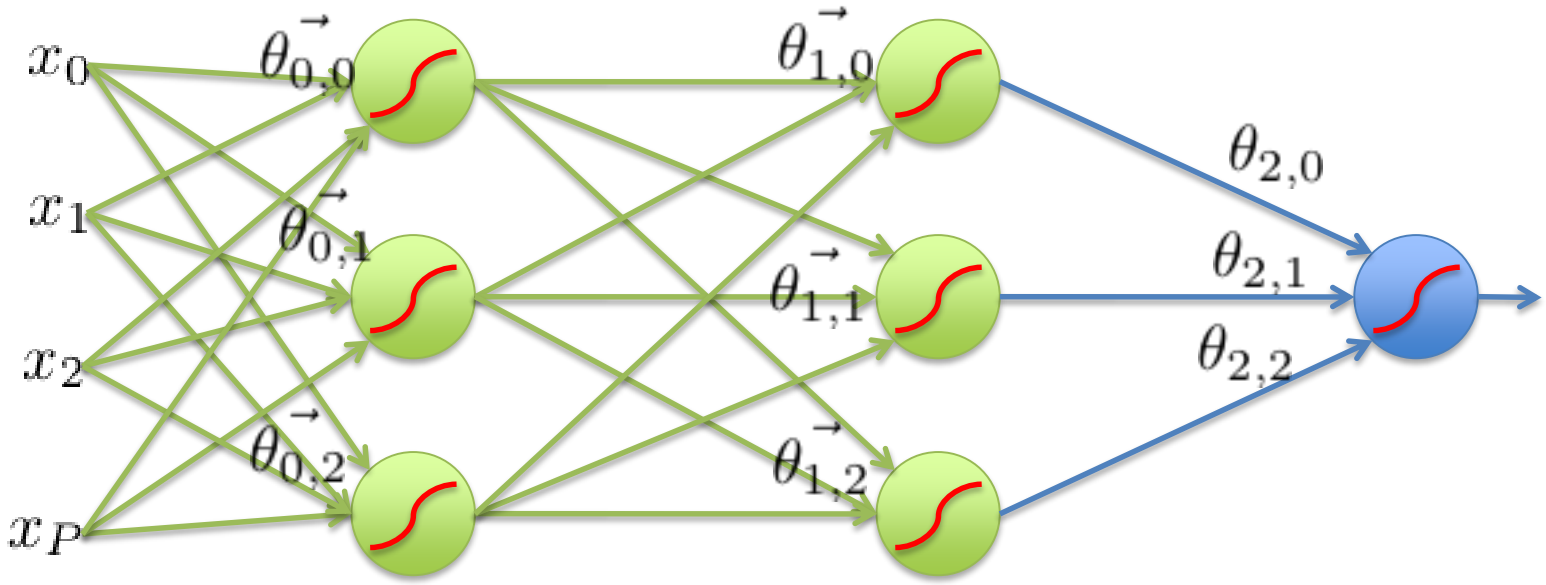
Feed-Forward Networks

- Predictions are fed forward through the network to classify



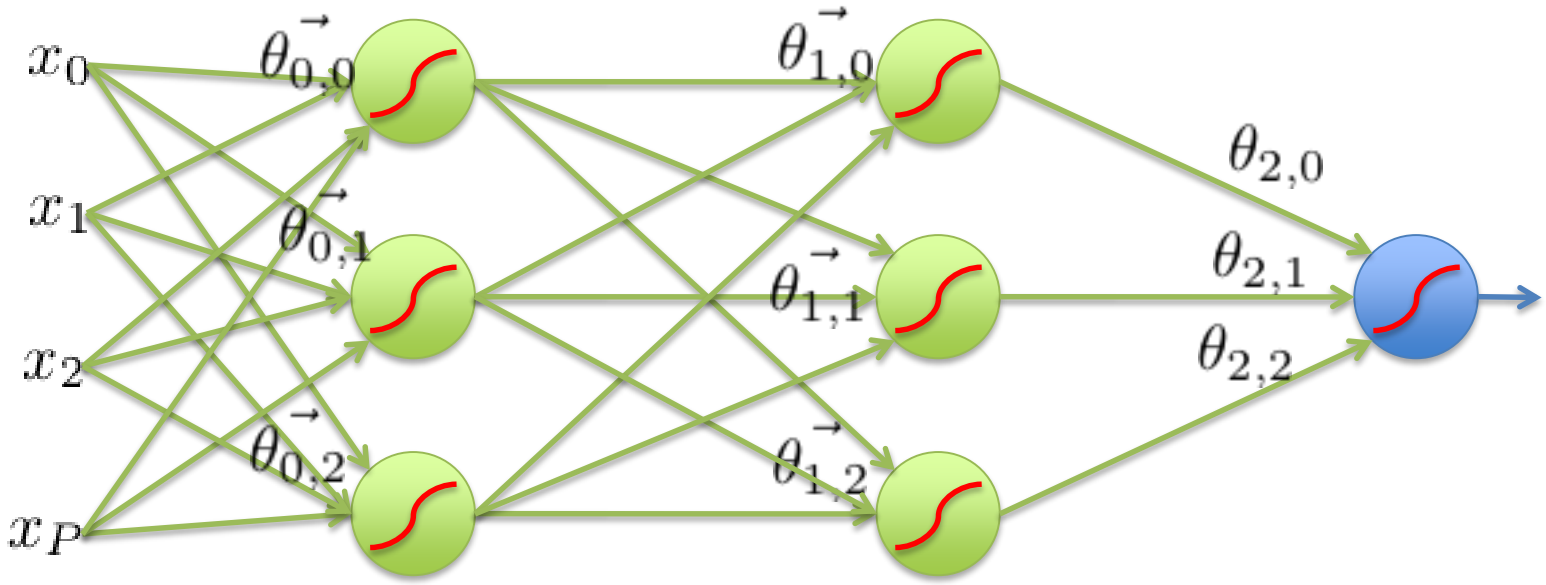
Feed-Forward Networks

- Predictions are fed forward through the network to classify



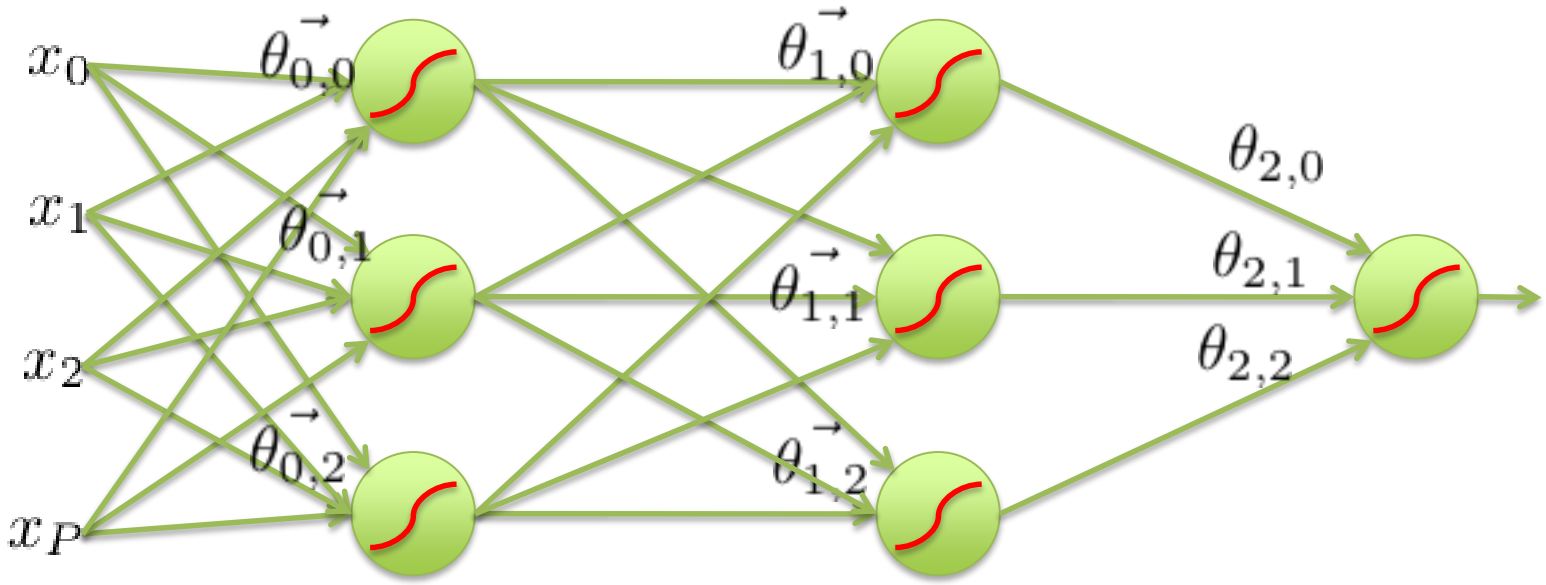
Feed-Forward Networks

- Predictions are fed forward through the network to classify



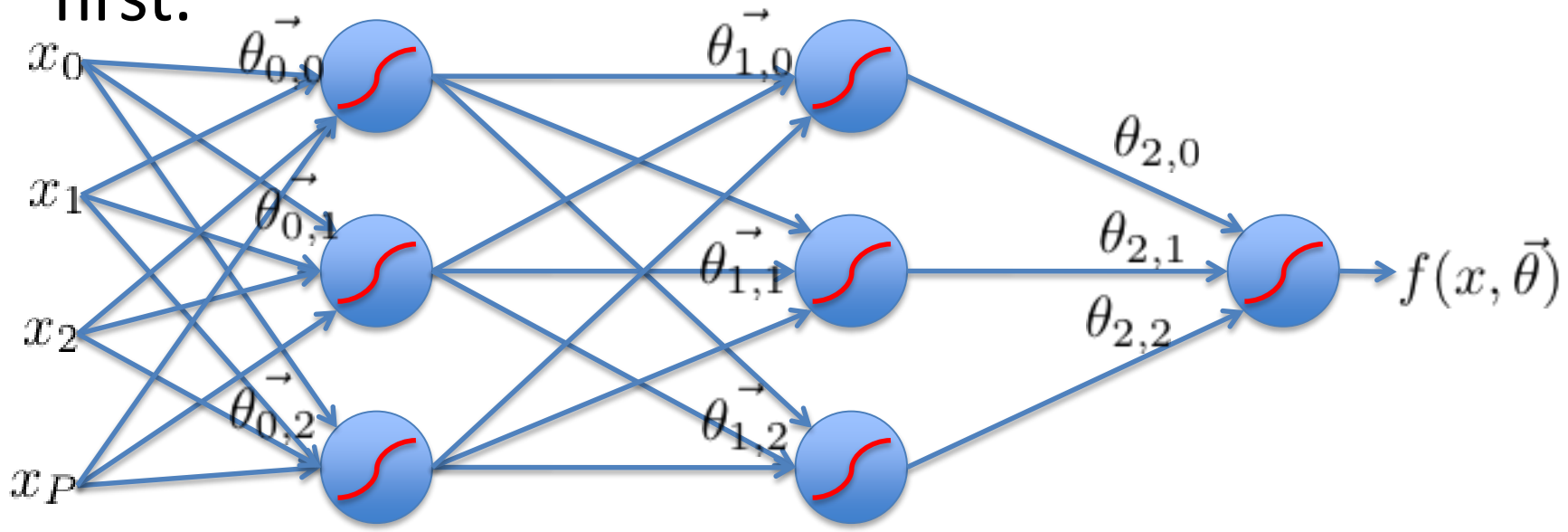
Feed-Forward Networks

- Predictions are fed forward through the network to classify



Error Backpropagation

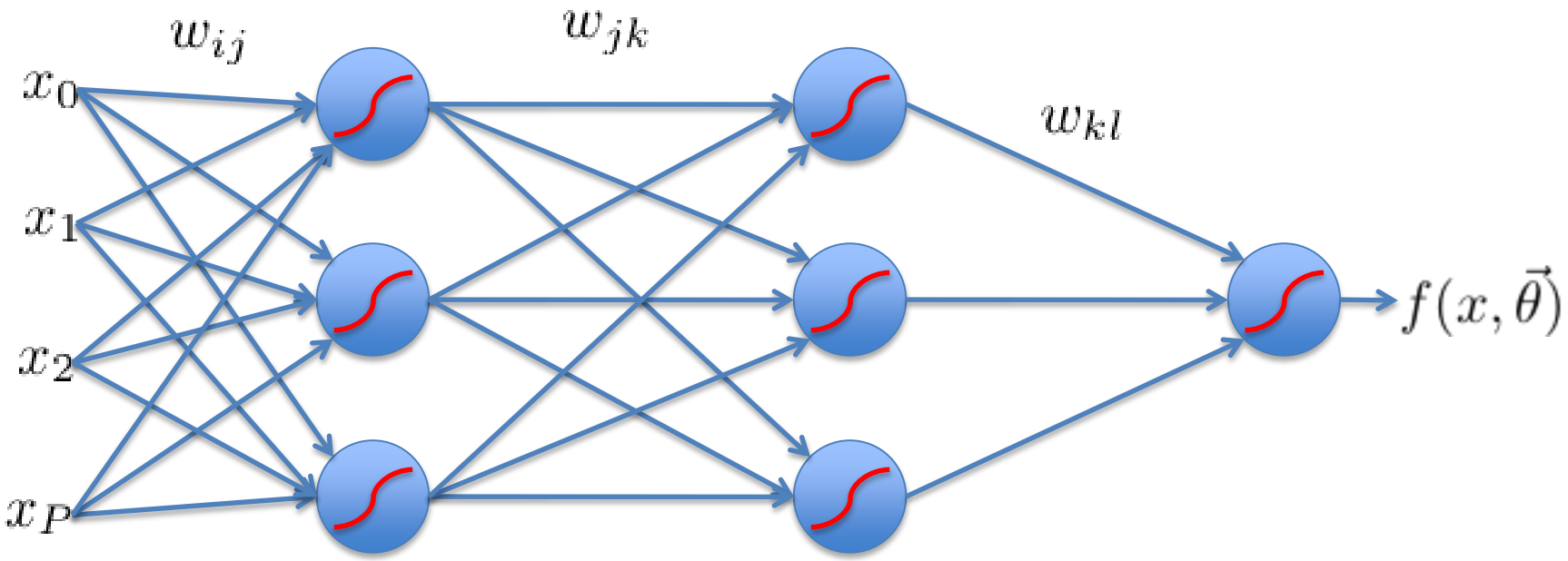
- We will do gradient descent on the whole network.
- Training will proceed from the last layer to the first.



Error Backpropagation

- Introduce variables over the neural network

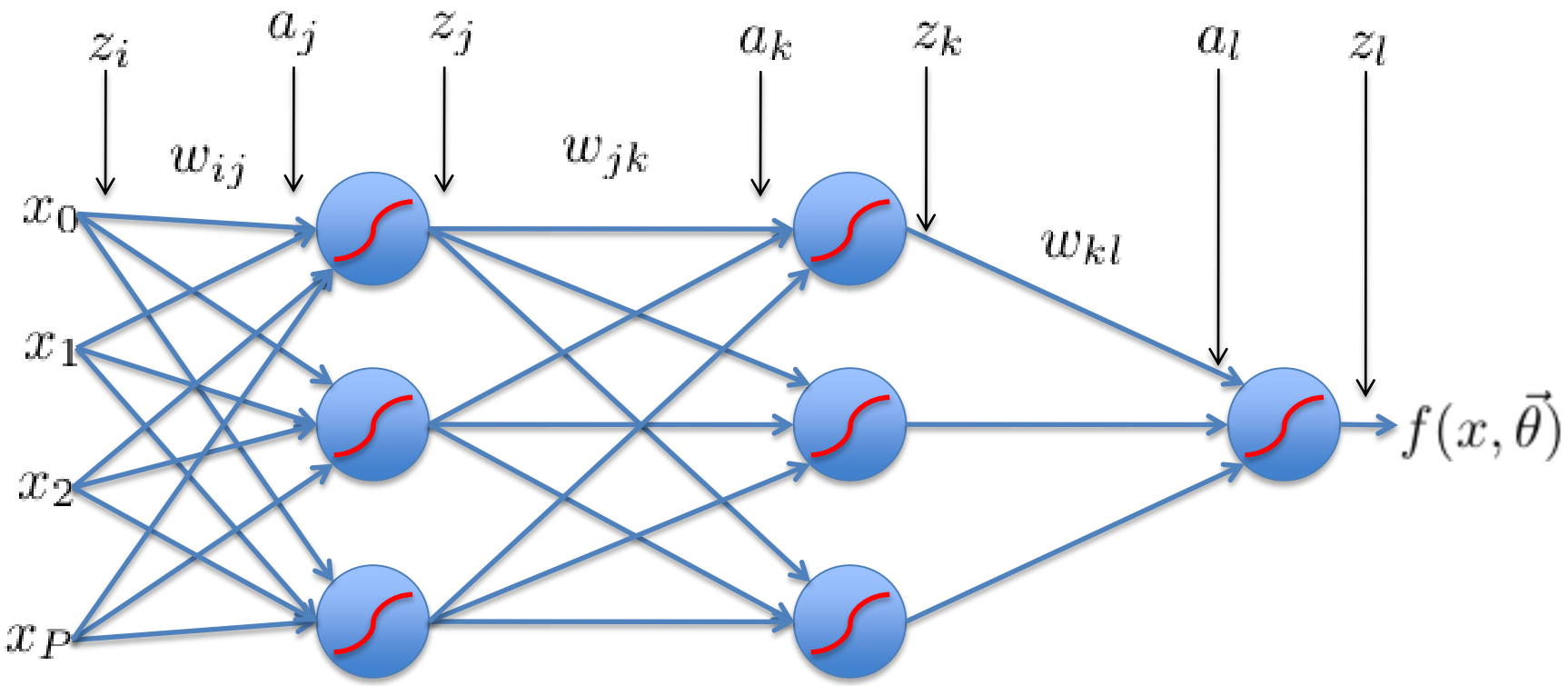
$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$



Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

- Introduce variables over the neural network
  - Distinguish the input and output of each node

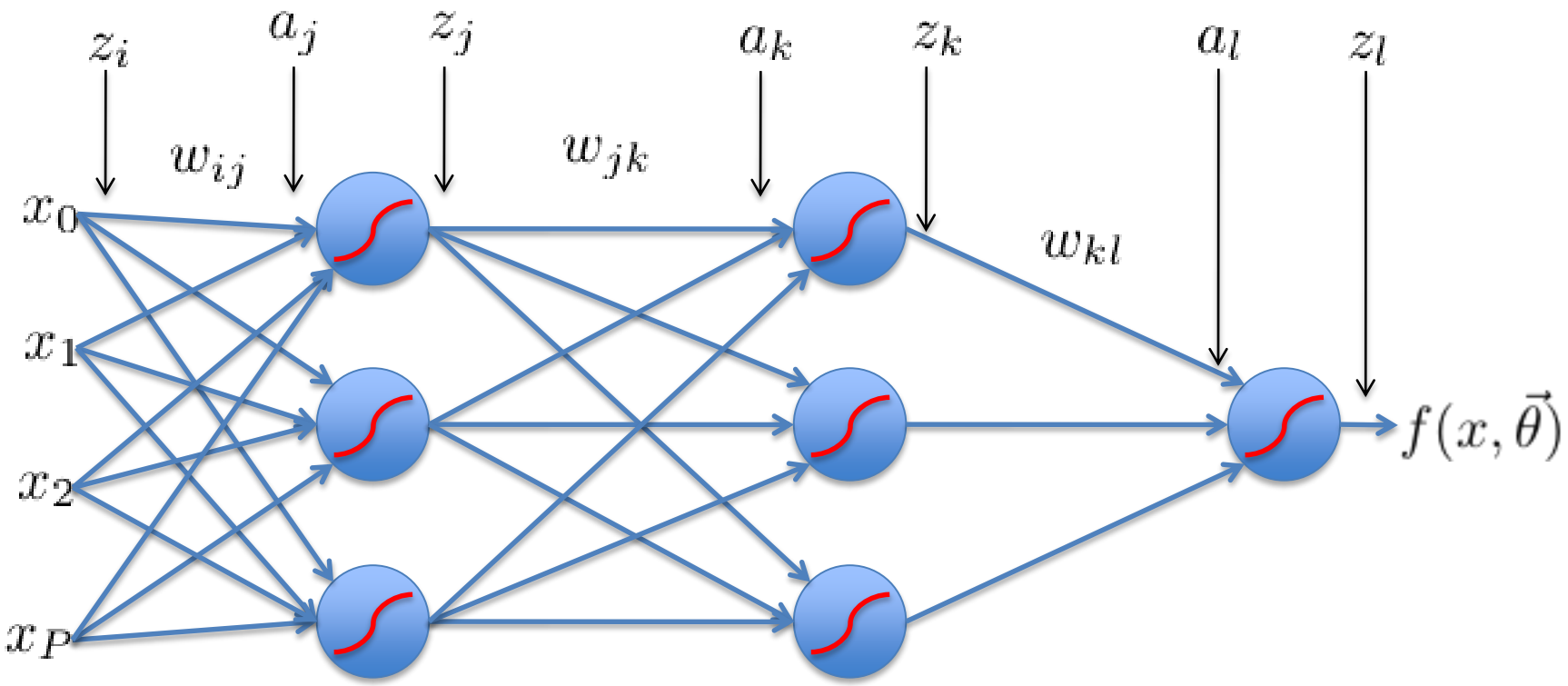


Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

$$a_j = \sum_i w_{ij} z_i \qquad a_k = \sum_j w_{jk} z_j \qquad a_l = \sum_k w_{kl} z_k$$

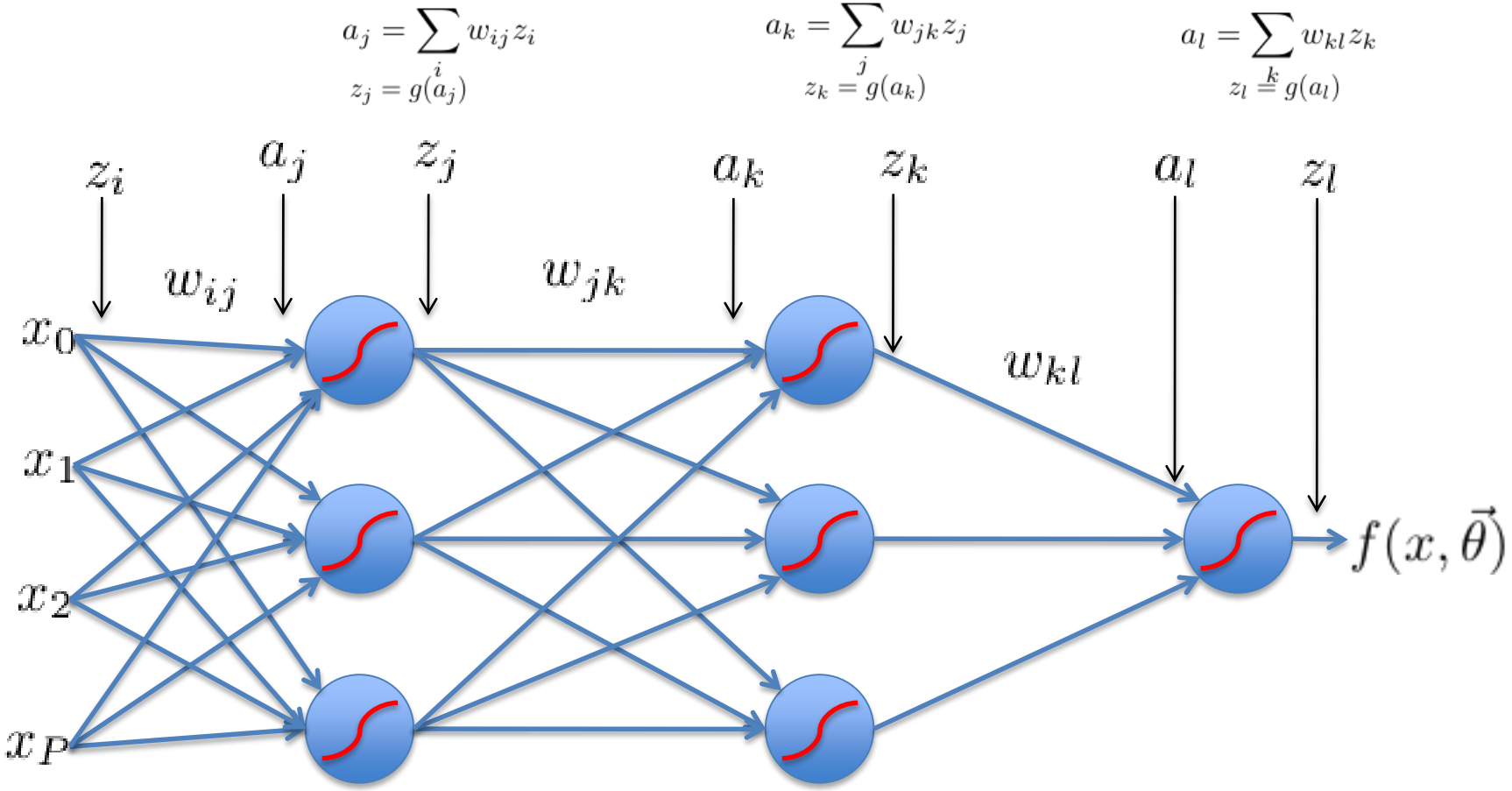
$$z_j = g(a_j) \qquad z_k = g(a_k) \qquad z_l = g(a_l)$$



Error Backpropagation

$$\vec{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$$

Training: Take the gradient of the last component and iterate backwards



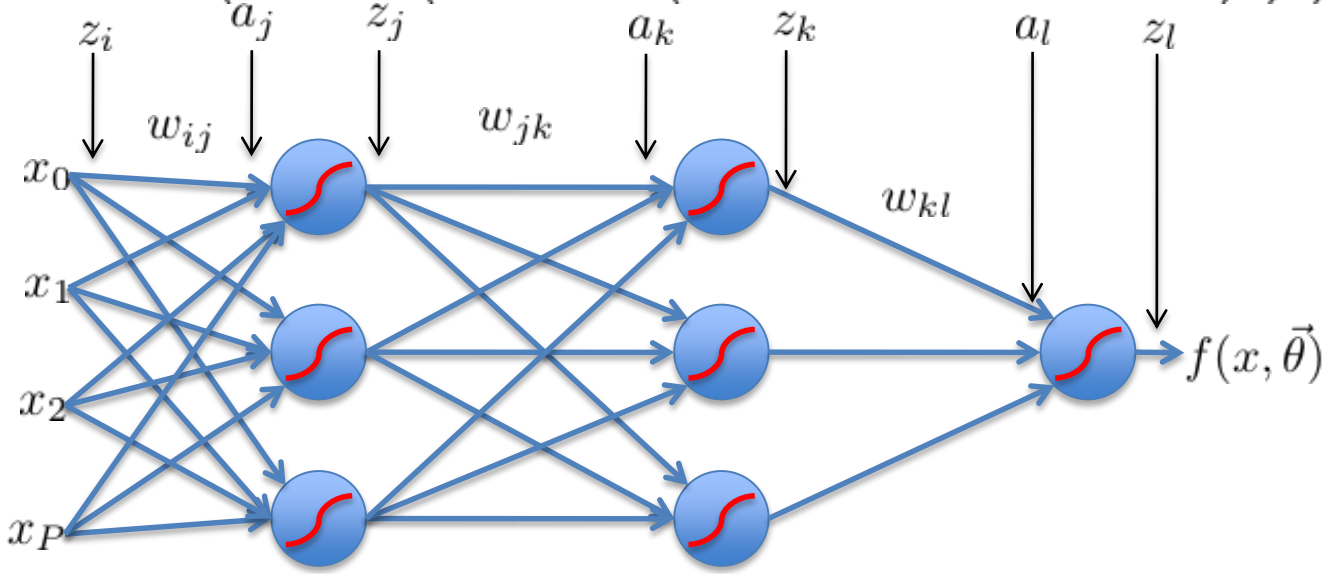
Error Backpropagation

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$

Empirical Risk Function

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl} g\left(\sum_j w_{jk} g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$



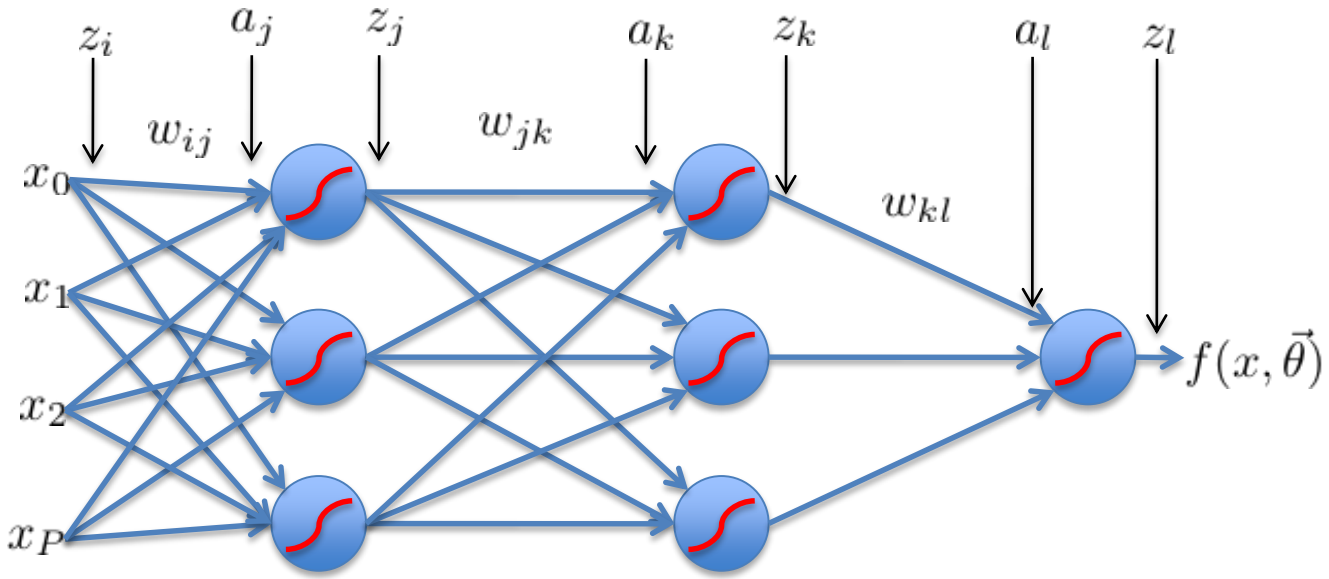
Error Backpropagation

Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule



Error Backpropagation

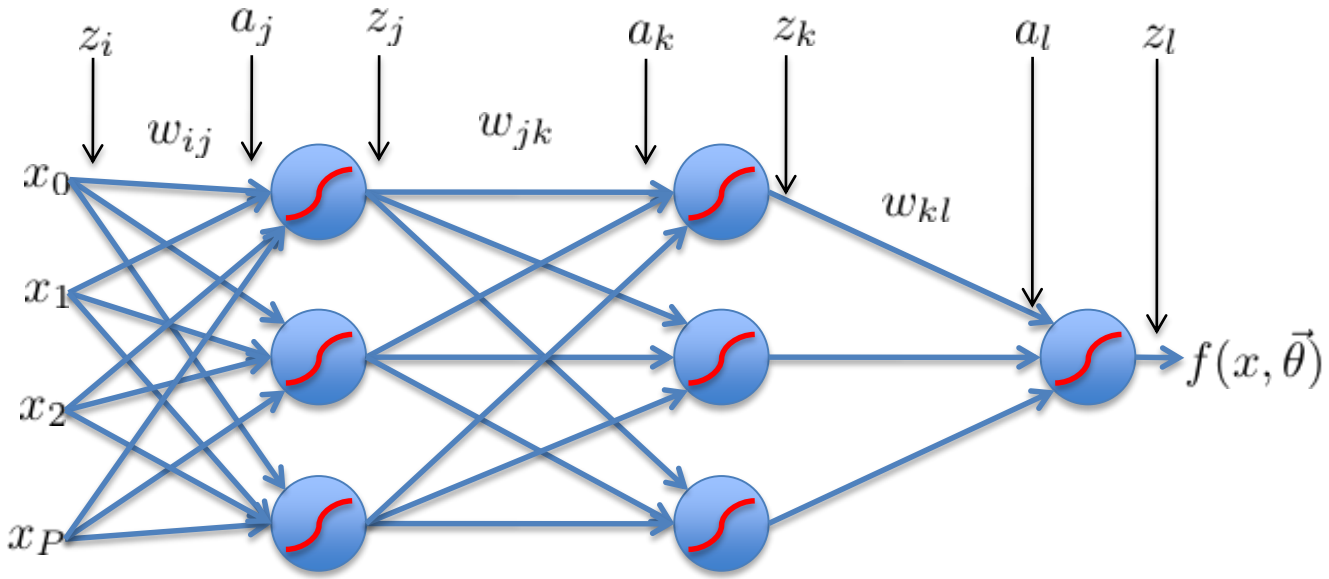
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$



Error Backpropagation

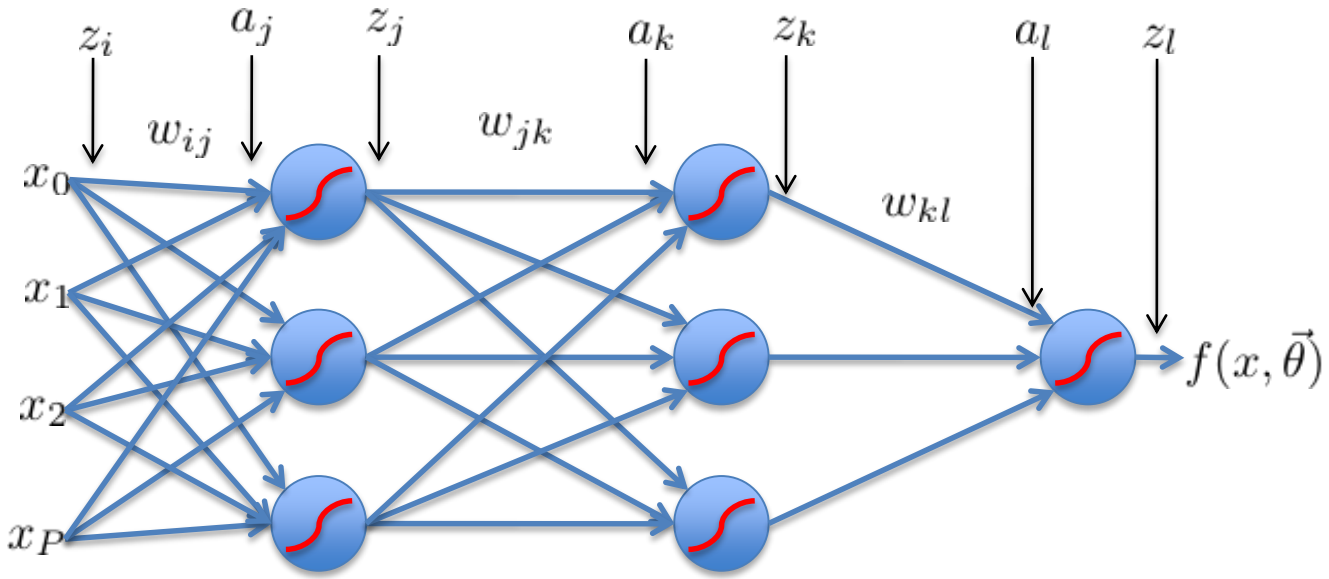
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right]$$



Error Backpropagation

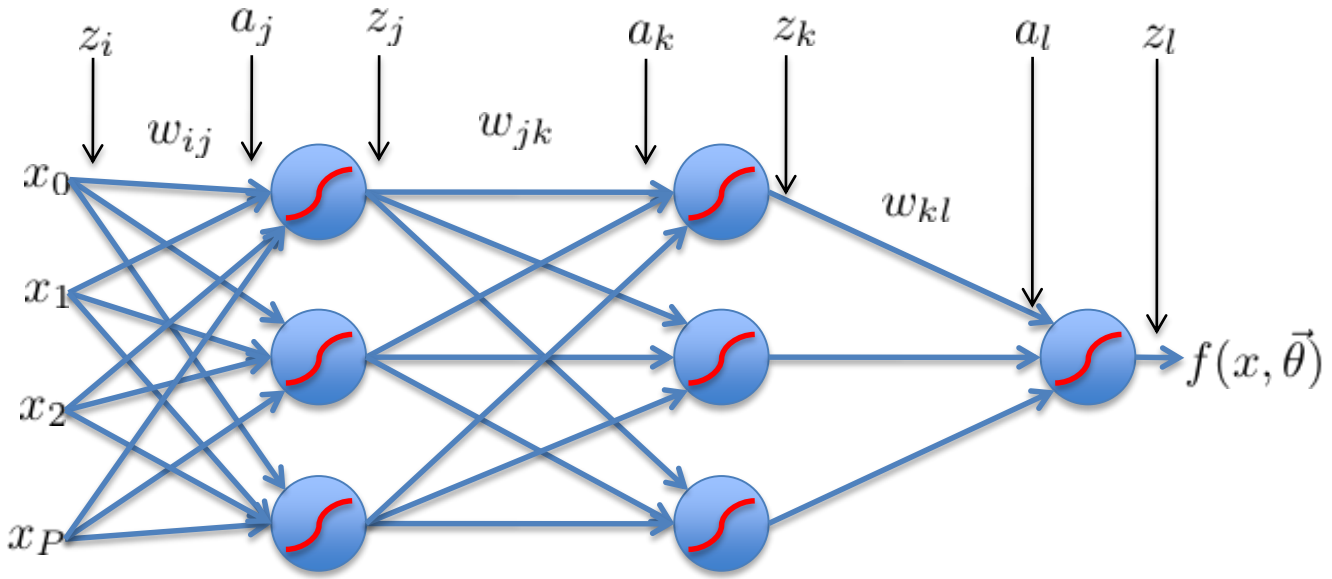
Optimize last layer weights w_{kl}

$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$



Error Backpropagation

Optimize last layer weights w_{kl}

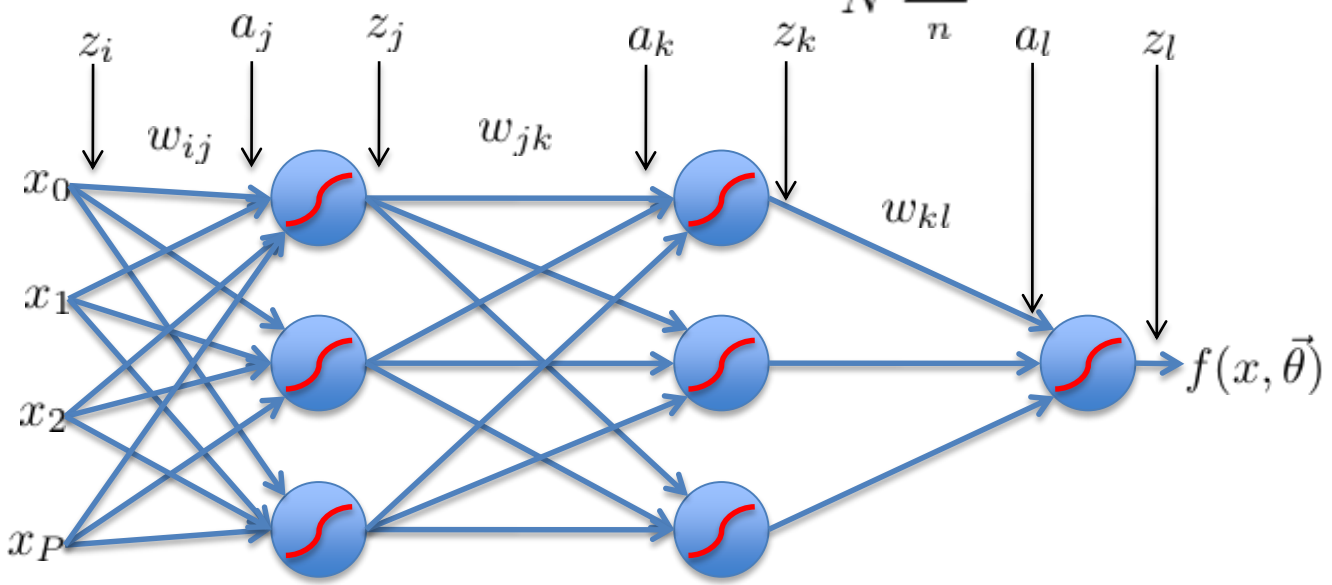
$$L_n = \frac{1}{2}\left(y_n - f(x_n)\right)^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial \frac{1}{2}(y_n - g(a_{l,n}))^2}{\partial a_{l,n}}\right]\left[\frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n}$$

$$= \frac{1}{N}\sum_n \delta_{l,n} n z_{k,n}$$

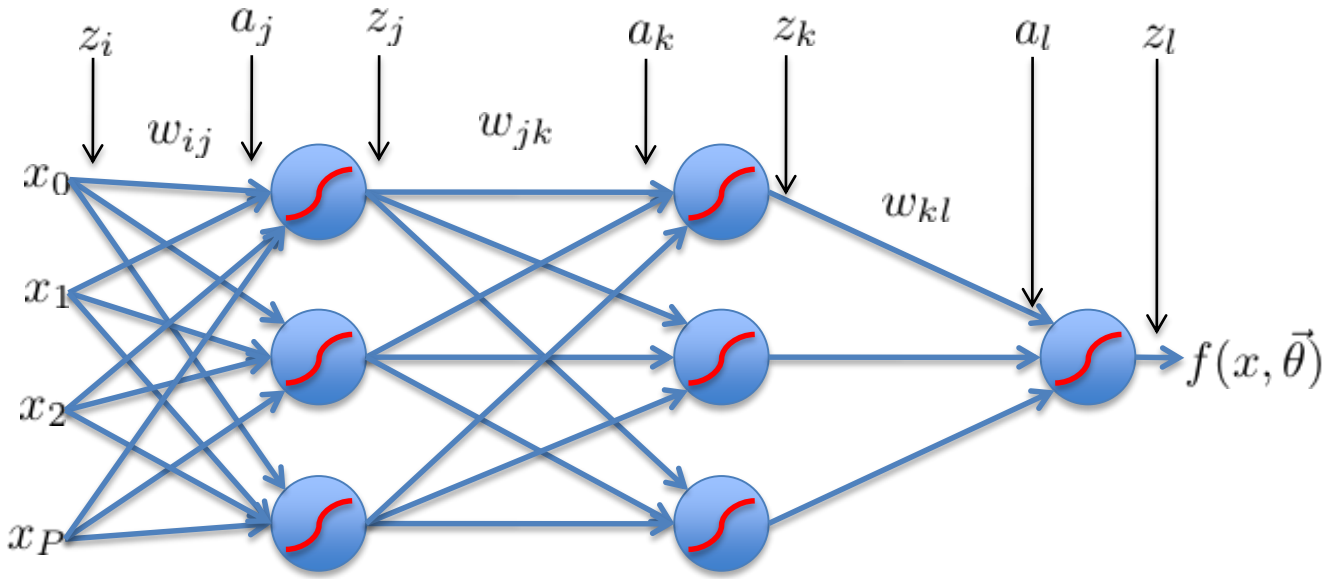


Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$



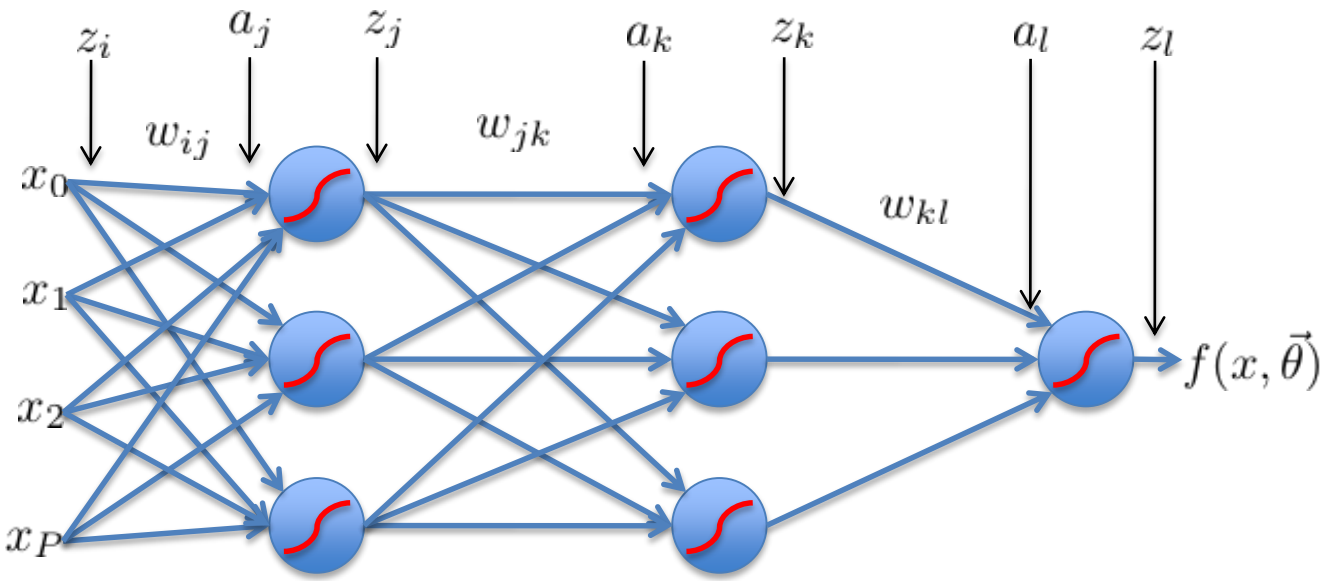
Error Backpropagation

Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}}\frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule



Error Backpropagation

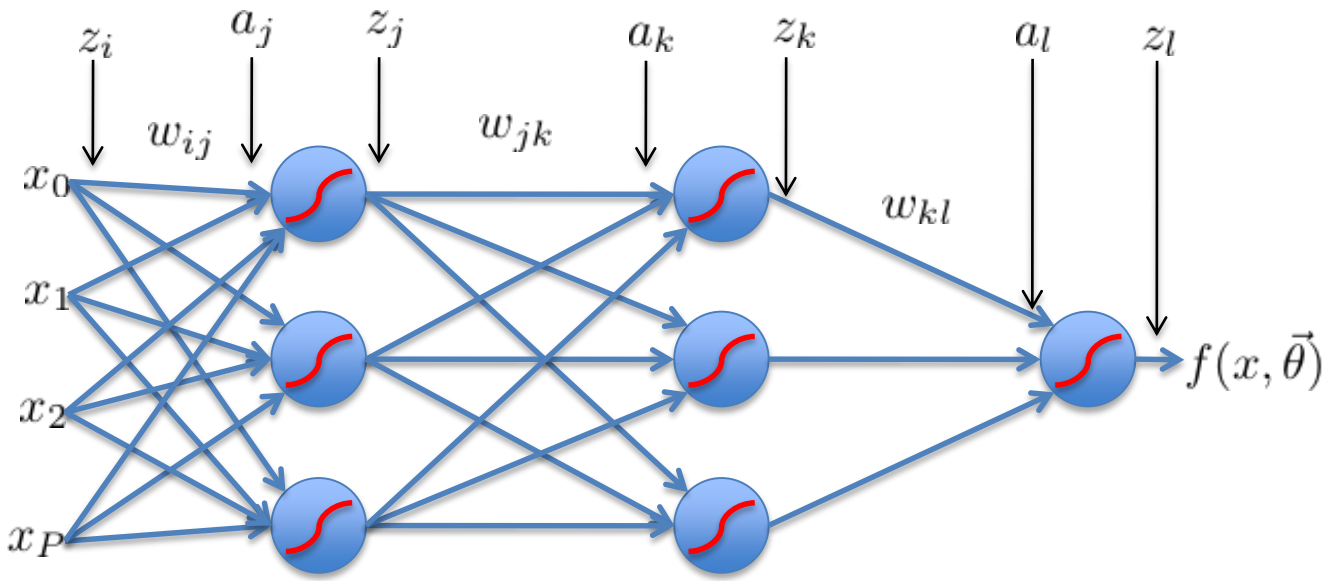
Optimize last hidden weights w_{jk}

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] [z_{j,n}]$$



Error Backpropagation

Optimize last hidden weights w_{jk}

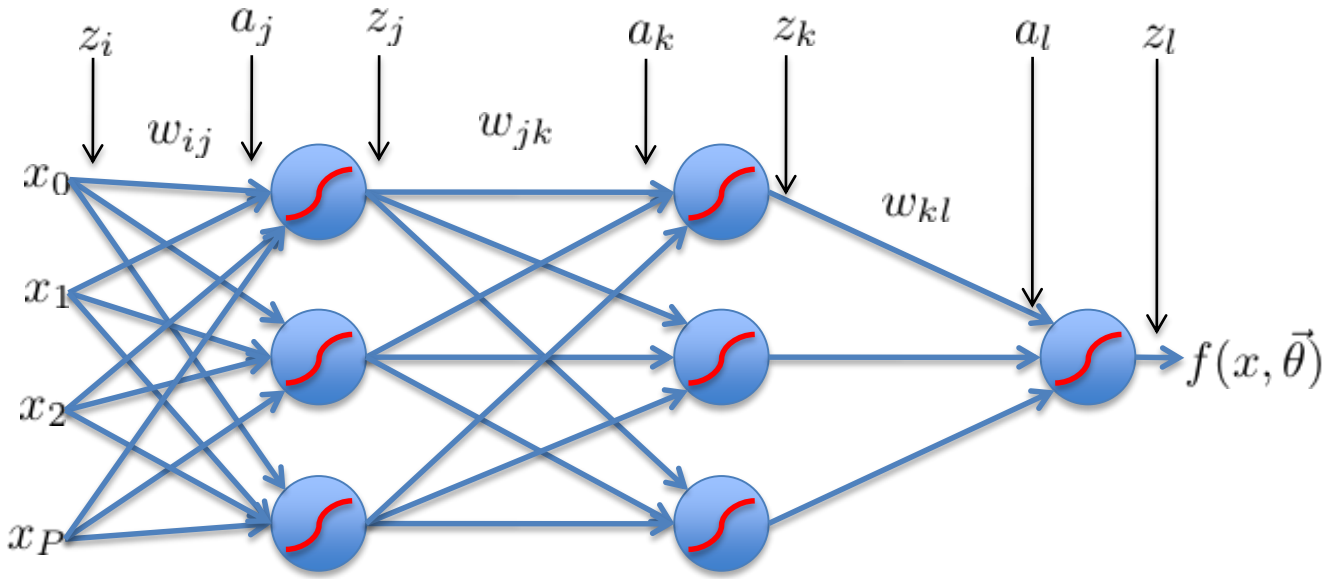
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}} \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] \left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l \frac{\partial a_{l,n}}{\partial a_{k,n}}\right] [z_{j,n}]$$

$$a_l = \sum_k w_{kl} g(a_k)$$



Error Backpropagation

Optimize last hidden weights w_{jk}

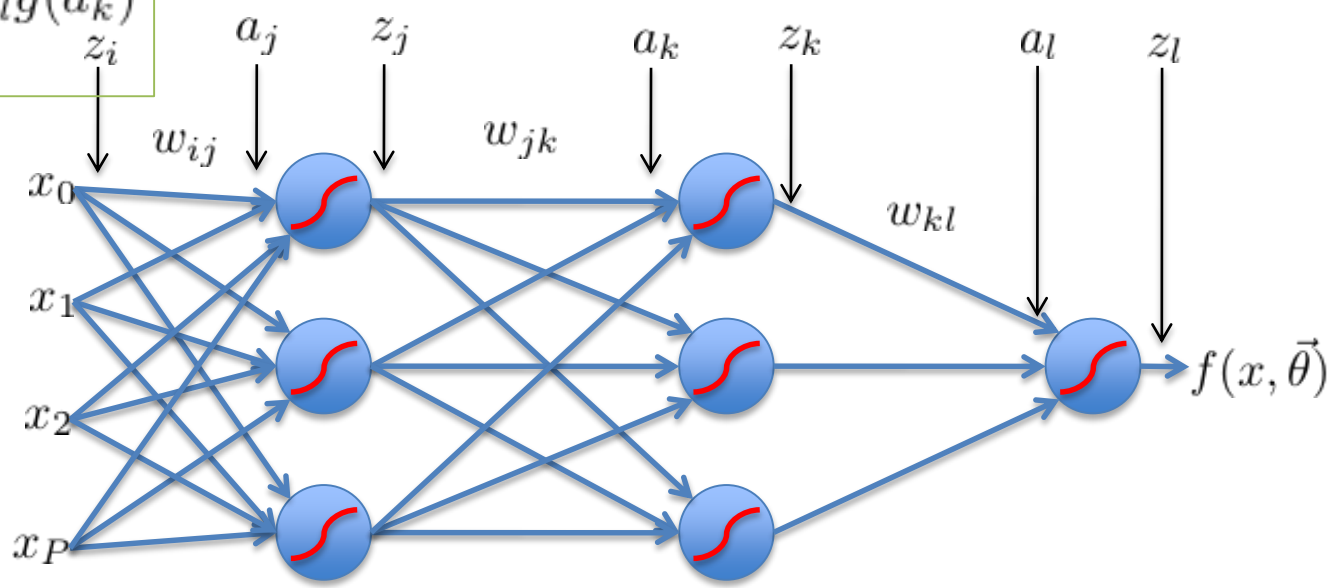
$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \frac{\partial L_n}{\partial a_{l,n}}\frac{\partial a_{l,n}}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right]$$

Multivariate chain rule

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\sum_l \delta_l w_{kl} g'(a_{k,n})\right][z_{j,n}] = \frac{1}{N}\sum_n [\delta_{k,n}][z_{j,n}]$$

$$a_l = \sum_k w_{kl} g(a_k)$$



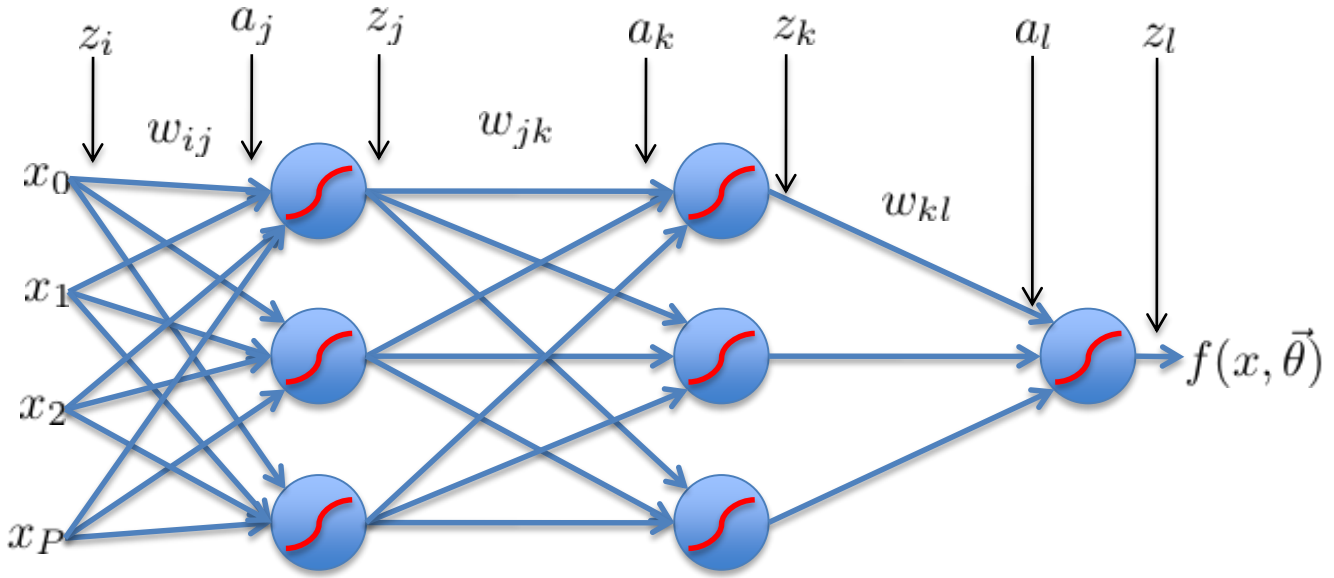
Error Backpropagation

Repeat for all previous layers

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{l,n}}\right]\left[\frac{\partial a_{l,n}}{\partial w_{kl}}\right] = \frac{1}{N}\sum_n \left[-(y_n - z_{l,n})g'(a_{l,n})\right] z_{k,n} = \frac{1}{N}\sum_n \delta_{l,n} z_{k,n}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{k,n}}\right]\left[\frac{\partial a_{k,n}}{\partial w_{jk}}\right] = \frac{1}{N}\sum_n \left[\sum_l \delta_{l,n} w_{kl} g'(a_{k,n})\right] z_{j,n} = \frac{1}{N}\sum_n \delta_{k,n} z_{j,n}$$

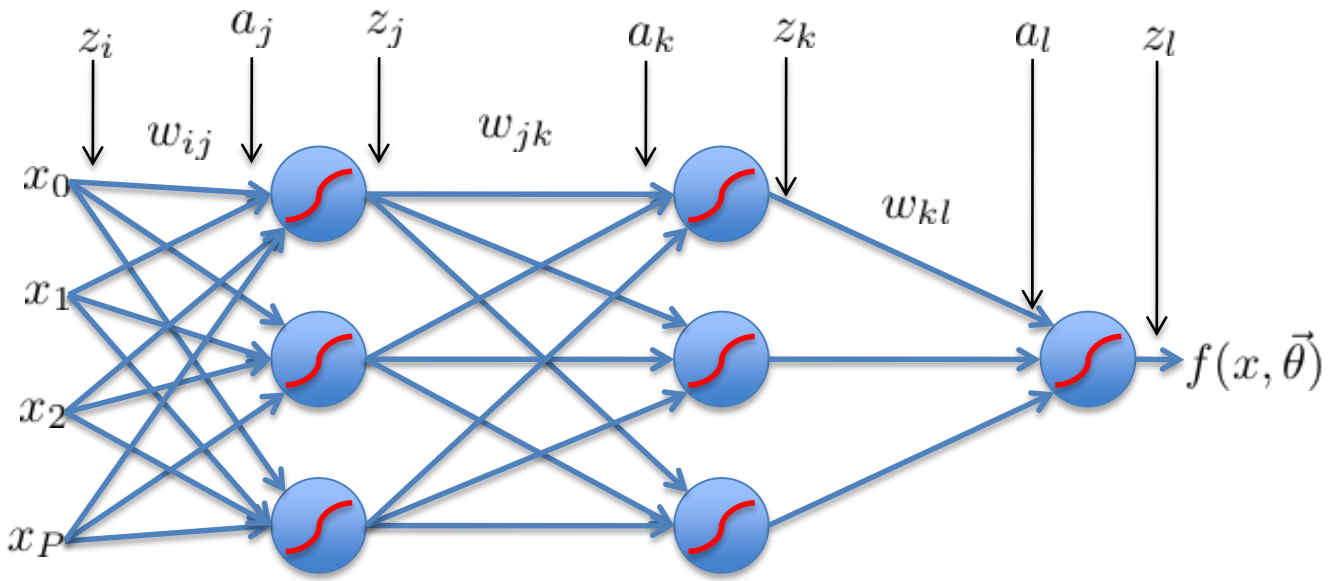
$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left[\frac{\partial L_n}{\partial a_{j,n}}\right]\left[\frac{\partial a_{j,n}}{\partial w_{ij}}\right] = \frac{1}{N}\sum_n \left[\sum_k \delta_{k,n} w_{jk} g'(a_{j,n})\right] z_{i,n} = \frac{1}{N}\sum_n \delta_{j,n} z_{i,n}$$



Error Backpropagation

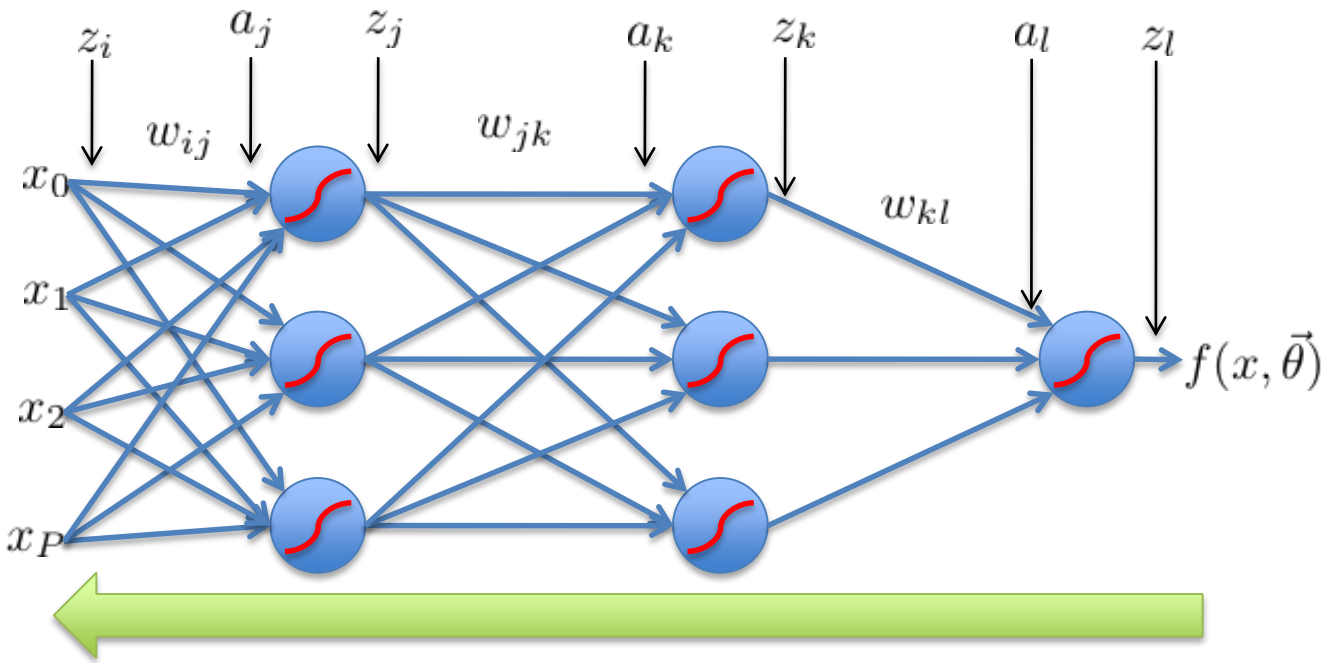
Now that we have well defined gradients for each parameter, update using Gradient Descent

$$
\begin{aligned}
w_{ij}^{t+1} &= w_{ij}^{t} - \eta \frac{\partial R}{w_{ij}} \\
w_{jk}^{t+1} &= w_{jk}^{t} - \eta \frac{\partial R}{w_{kl}} \\
w_{kl}^{t+1} &= w_{kl}^{t} - \eta \frac{\partial R}{w_{kl}}
\end{aligned}
$$



Error Back-propagation

- Error backprop unravels the multivariate chain rule and solvesthe gradient for each partial component separately.
- The target values for each layer come from the next layer.
- This feeds the errors back along the network.



Problems with Neural Networks

- Interpretation of Hidden Layers
- Overfitting

Interpretation of Hidden Layers

- What are the hidden layers doing?!
- Feature Extraction
- The non-linearities in the feature extraction can make interpretation of the hidden layers very difficult.
- This leads to Neural Networks being treated as **black boxes**.

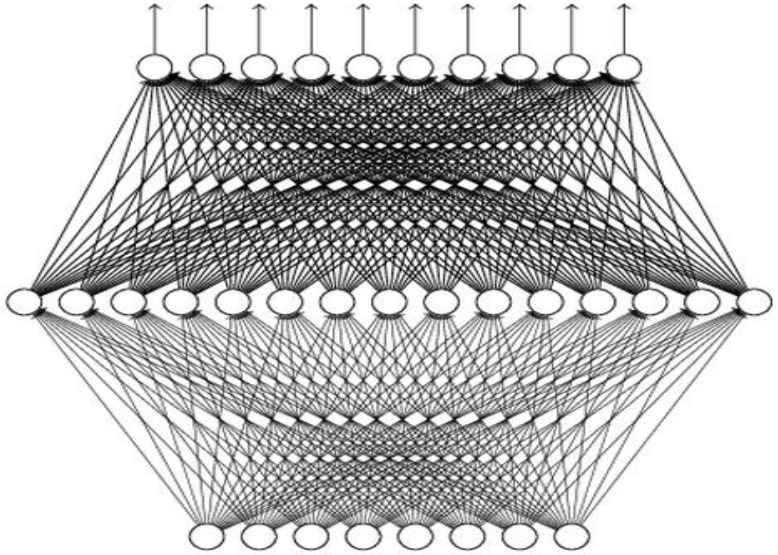
Universality Theorem

Any continuous function f

$$f : R^N \rightarrow R^M$$

Can be realized by a network with one hidden layer

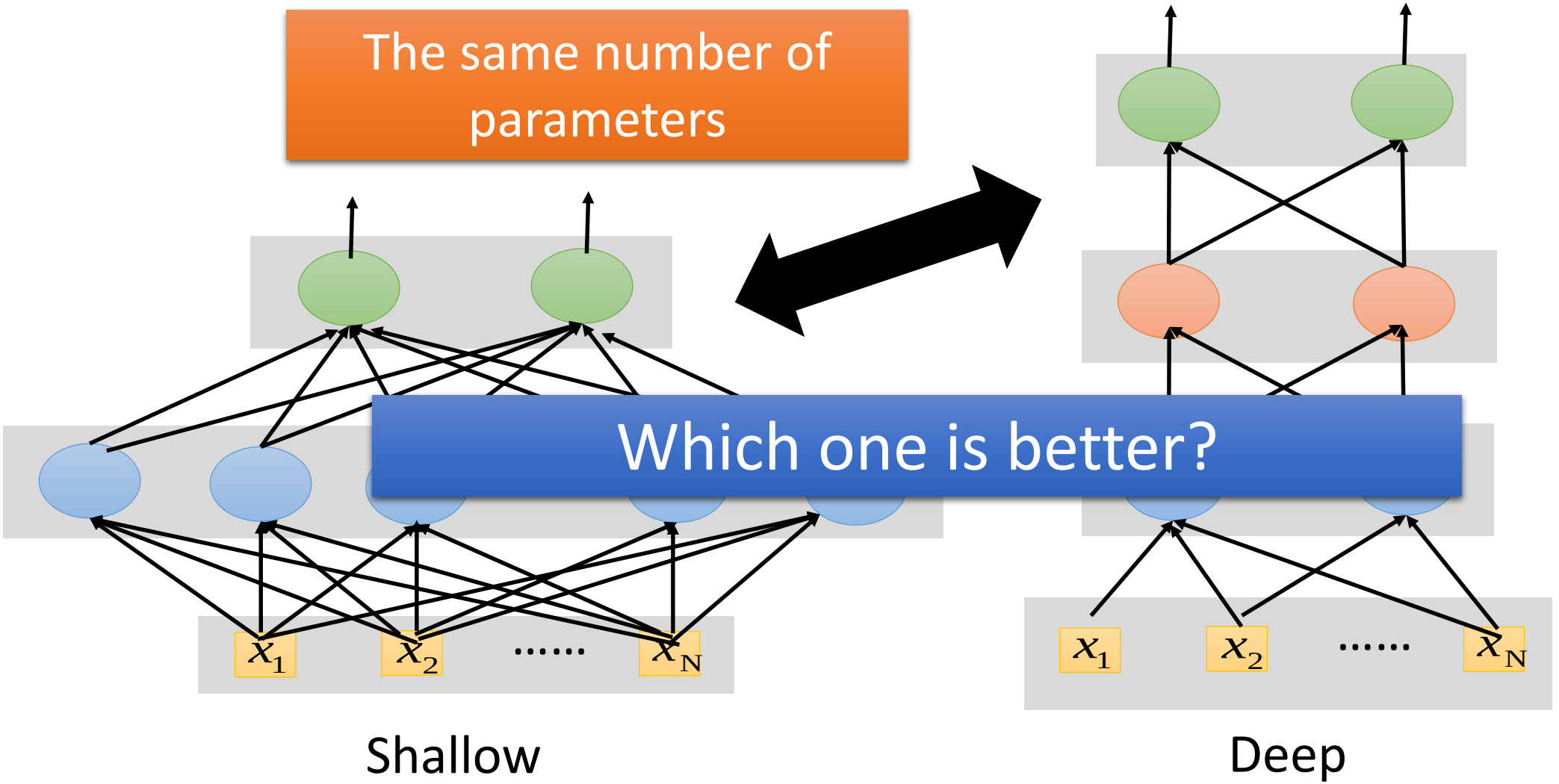
(given **enough** hidden neurons)



Reference for the reason: http://neuralnetworksanddeeplearning.com/chap4.html

"Deep" neural network vs. "Fat" neural network?

Fat + Short v.s. Thin + Tall



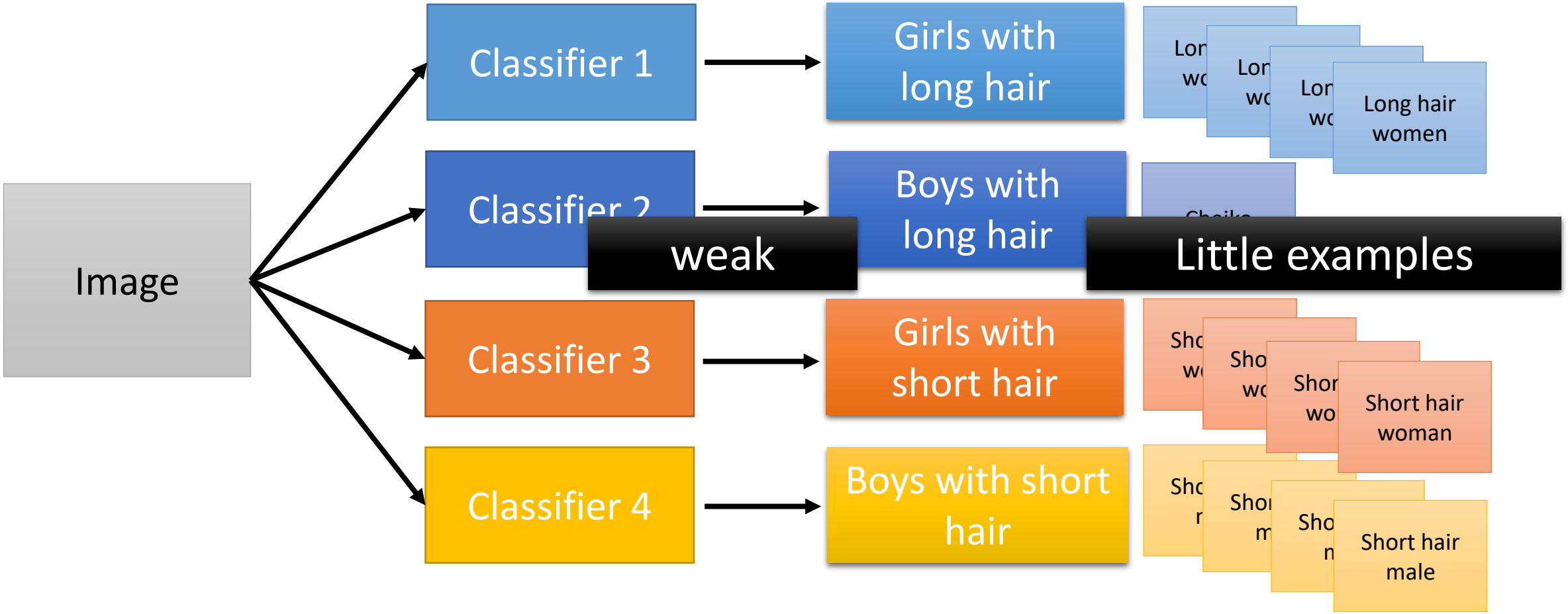
Fat + Short v.s. Thin + Tall

Layer X Size	Word Error Rate (%)	Layer X Size	Word Error Rate (%)
1 X 2k	24.2		
2 X 2k	20.4		
3 X 2k	18.4		
4 X 2k	17.8		
5 X 2k	17.2	1 X 3772	22.5
7 X 2k	17.1	1 X 4634	22.6
		1 X 16k	22.1

Seide, Frank, Gang Li, and Dong Yu. "Conversational Speech Transcription Using Context-Dependent Deep Neural Networks." *Interspeech*. 2011.

Why Deep?

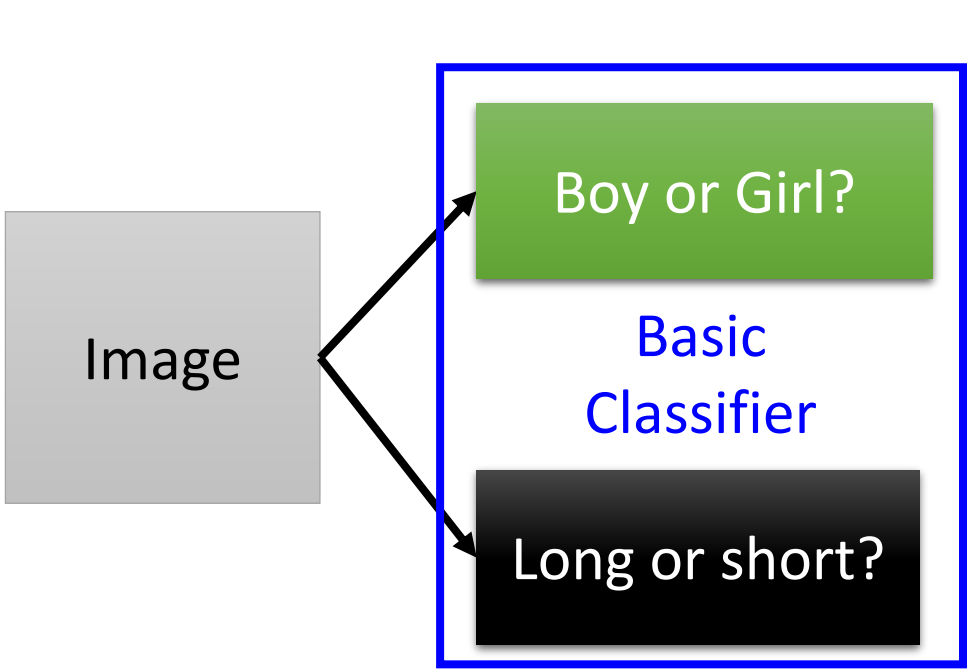
- Deep → Modularization



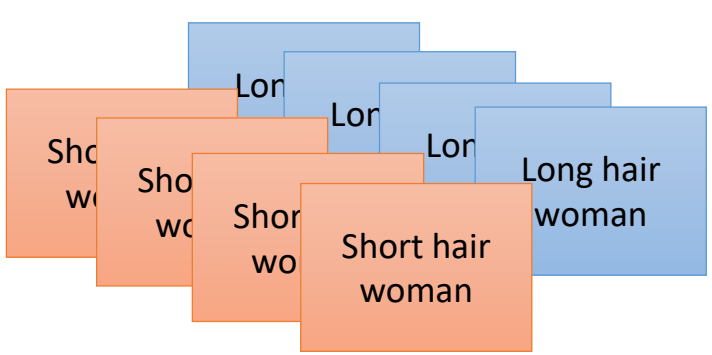
Why Deep?

Each basic classifier can have sufficient training examples.

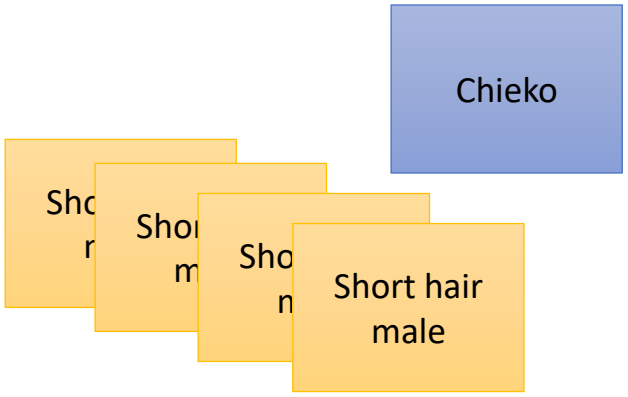
- Deep → Modularization

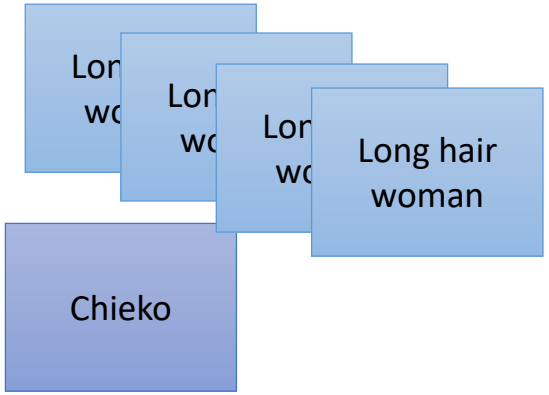


Classifiers for the attributes

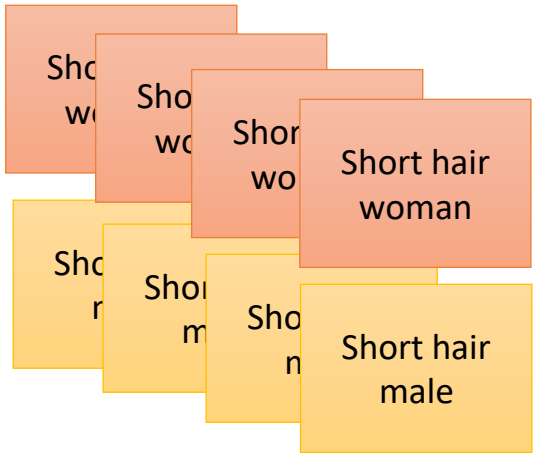


V.S.



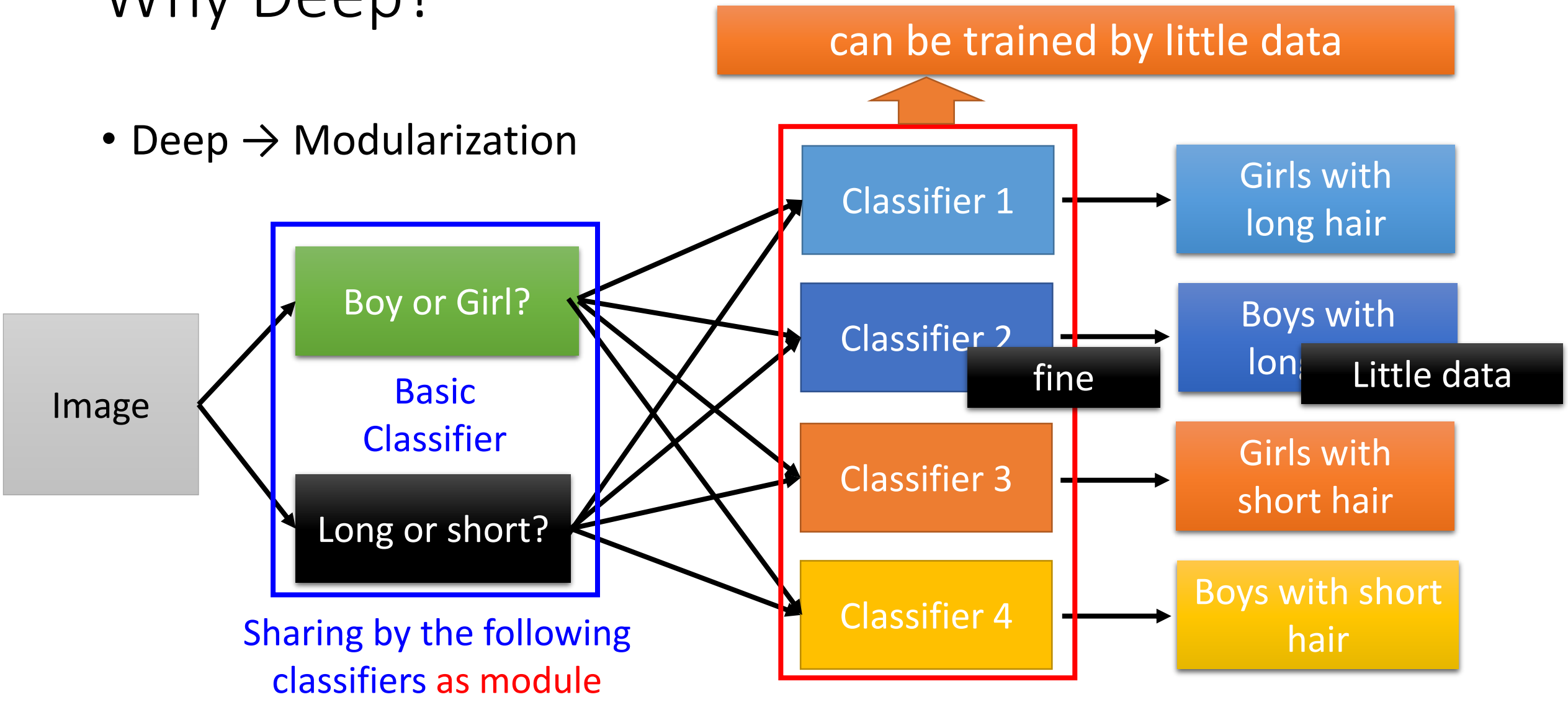


V.S.



Why Deep?

- Deep → Modularization

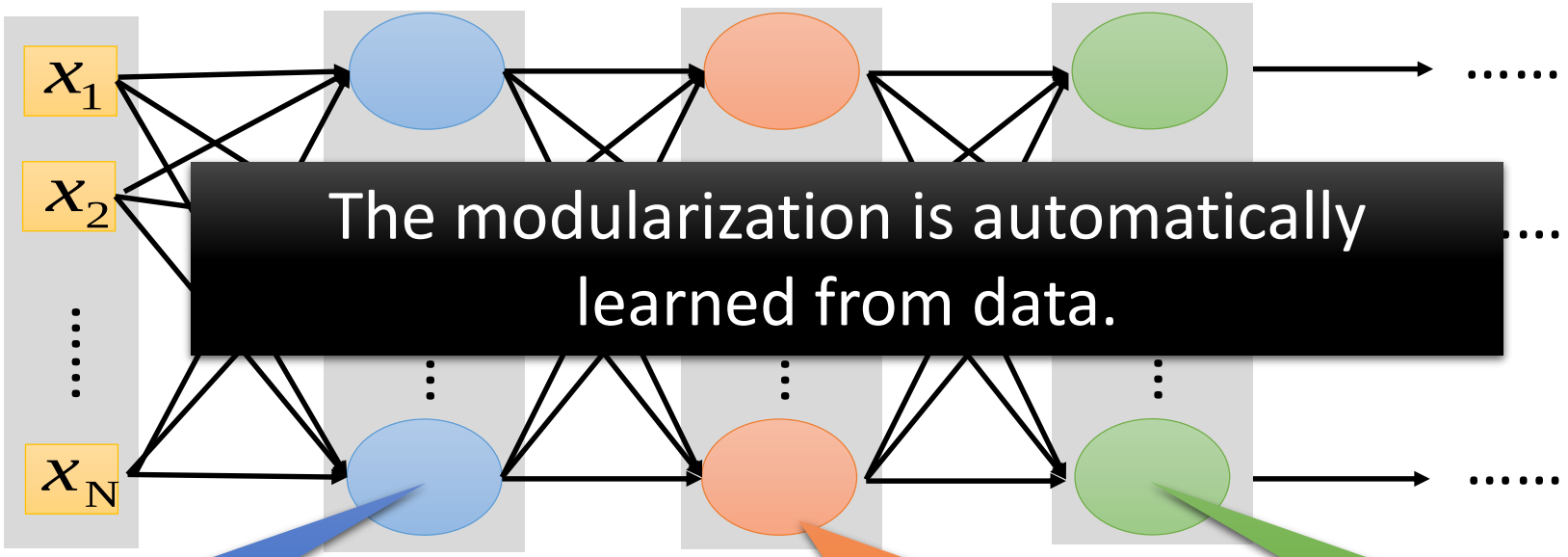


Why Deep?

Deep Learning also works on small data set like TIMIT.

- Deep → Modularization

→ Less training data?



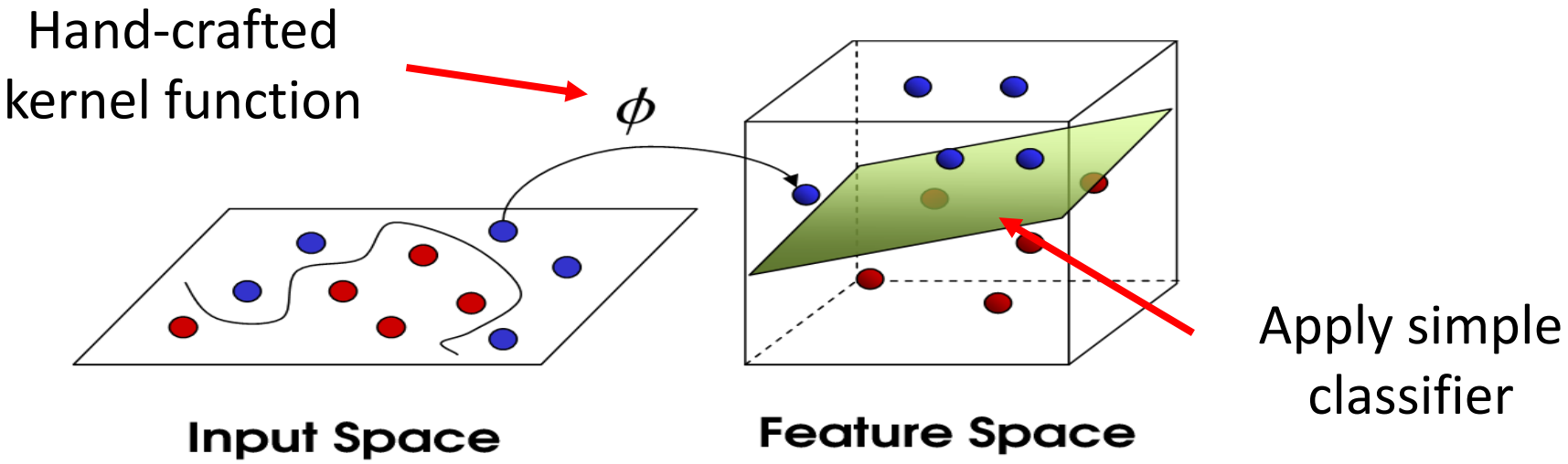
The modularization is automatically learned from data.

The most basic classifiers

Use 1st layer as module to build classifiers

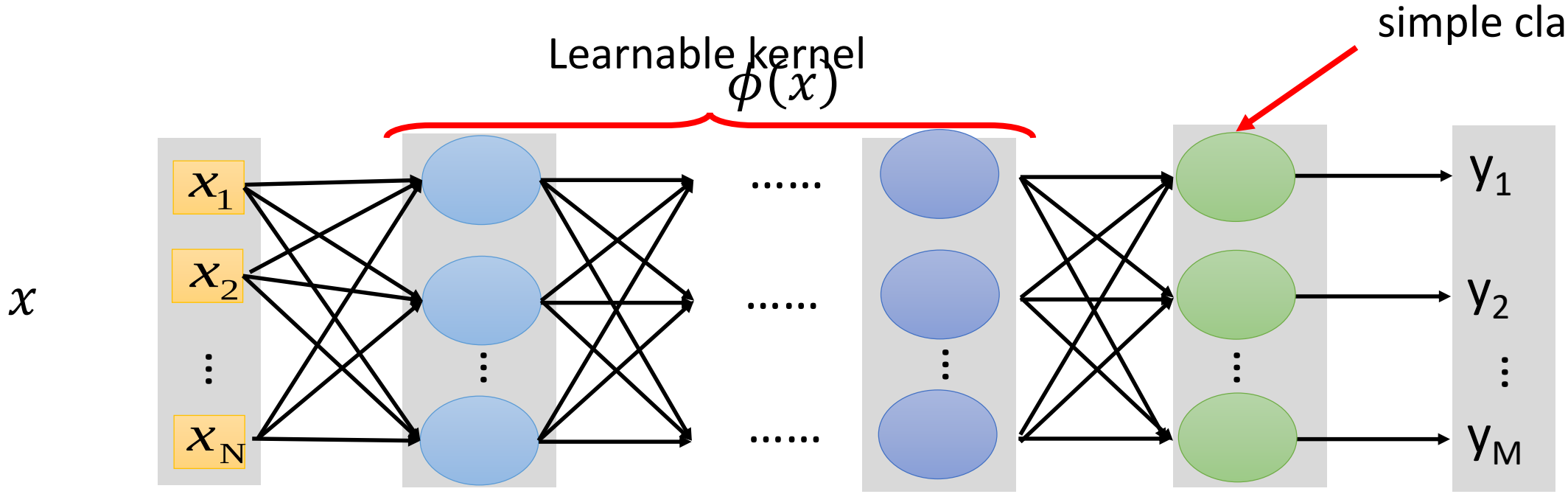
Use 2nd layer as module ……

SVM

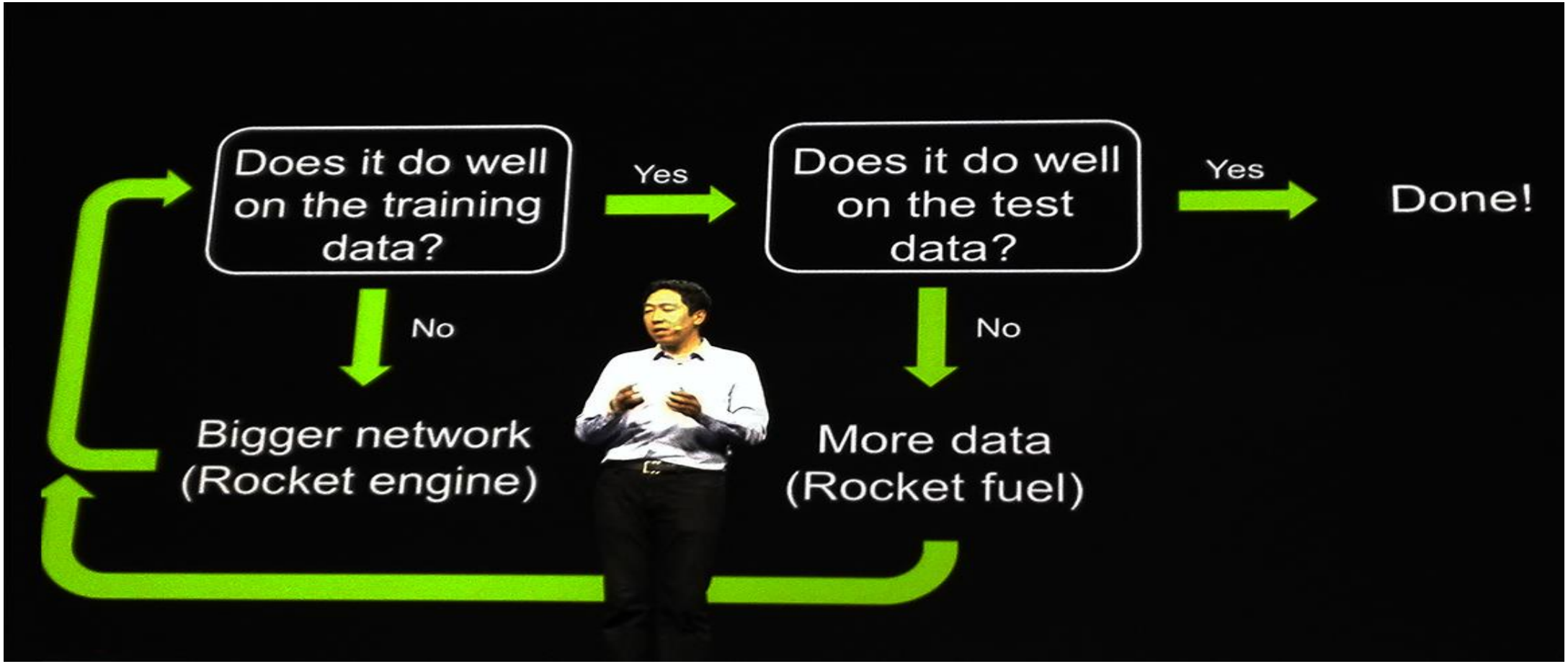


Source of image: http://www.gipsa-lab.grenoble-inp.fr/transfert/seminaire/455_Kadri2013Gipsa-lab.pdf

Deep Learning

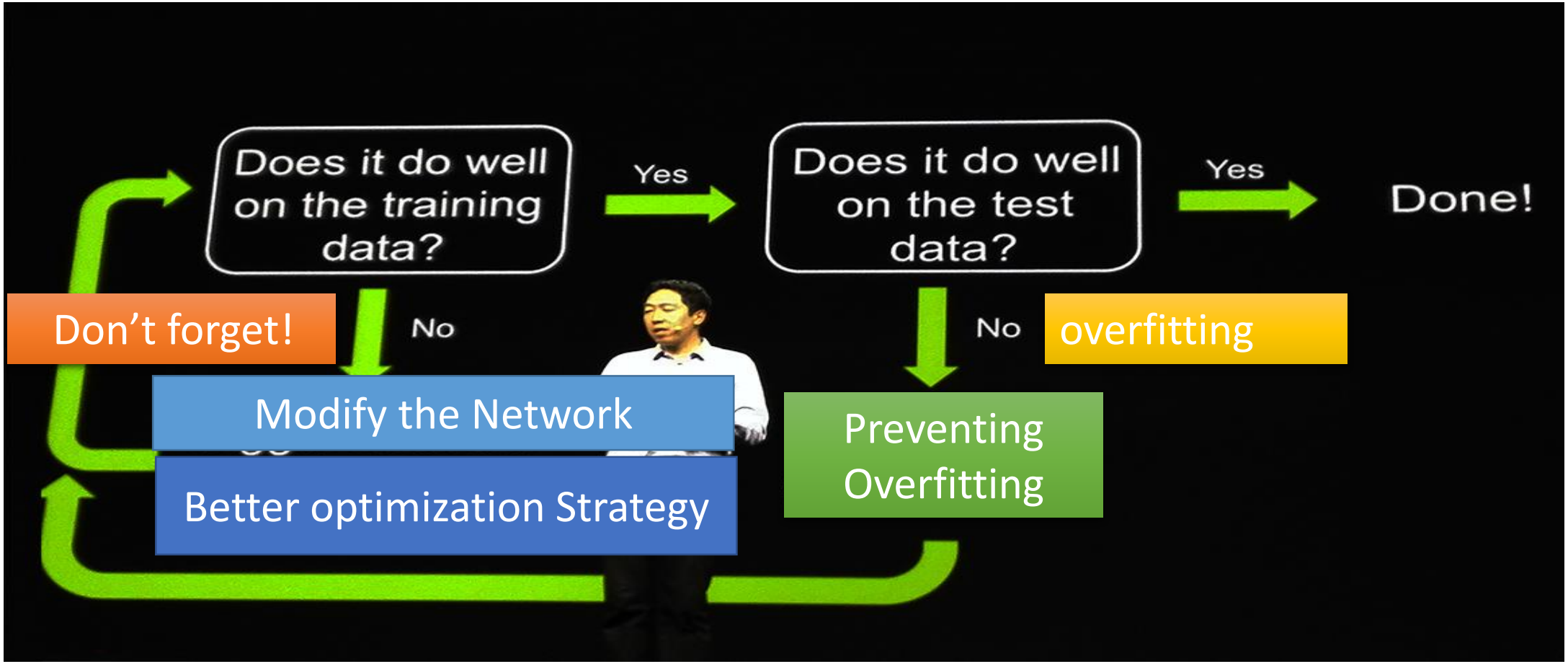


Recipe for Learning



http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

Recipe for Learning



http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

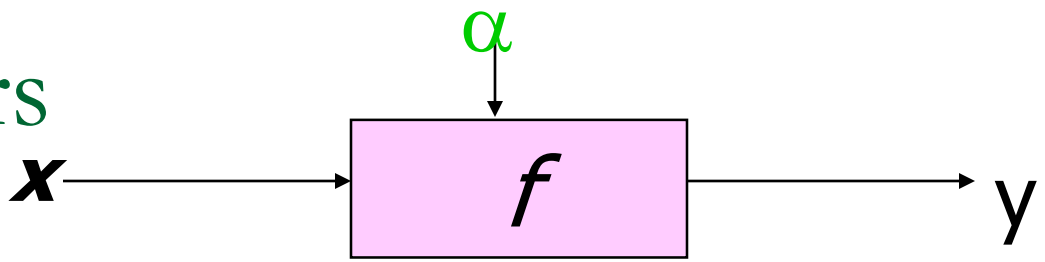

thank you

Support Vector Machines, Clustering, and more…

Introduction

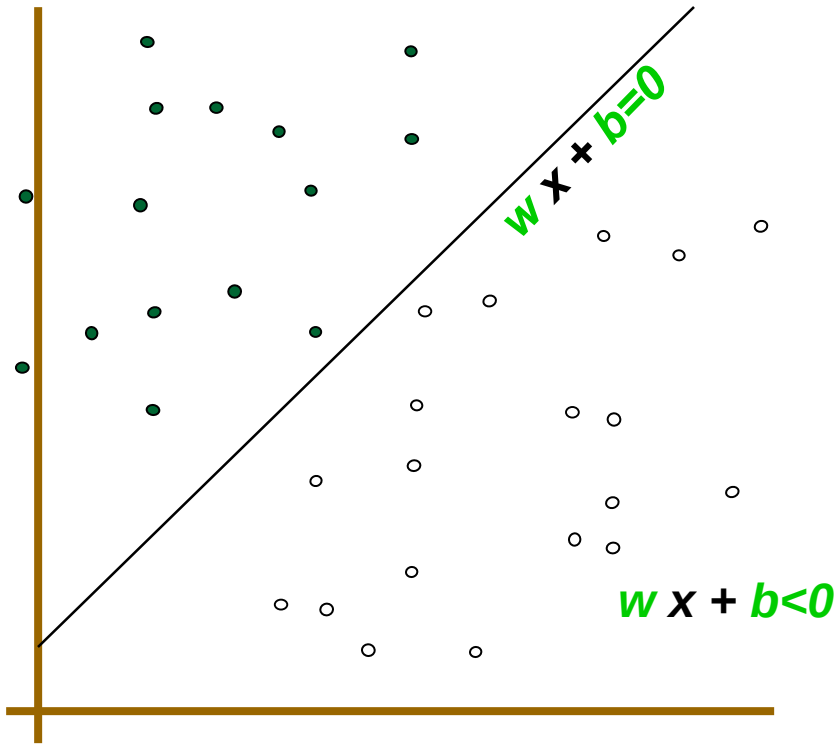
- Support vector machines were invented by V. Vapnik and his co-workers in 1970s in Russia and became known to the West in 1992.
- SVMs are linear classifiers that find a hyperplane to separate two class of data, positive and negative.
- Kernel functions are used for nonlinear separation.
- SVM not only has a rigorous theoretical foundation, but also performs classification more accurately than most other methods in applications, especially for high dimensional data.
- It is perhaps the best classifier for text classification.

Linear Classifiers



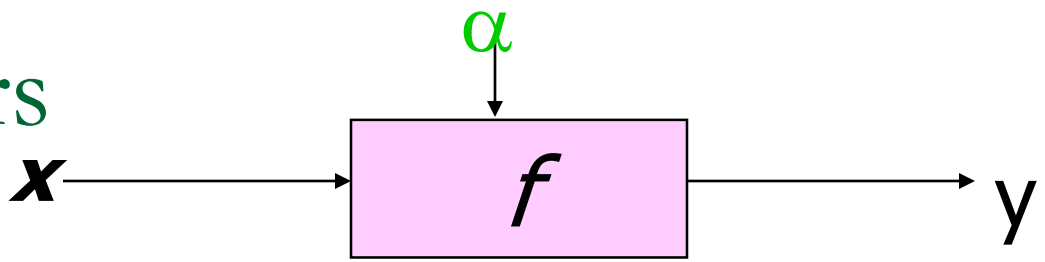
- denotes +1
- denotes -1

$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$



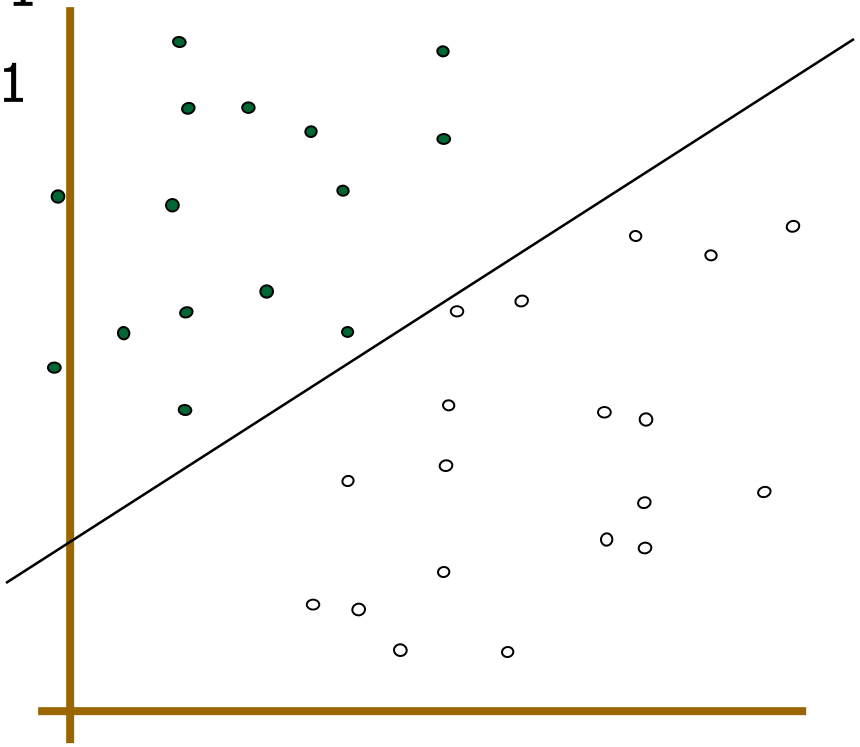
How would you classify this data?

Linear Classifiers



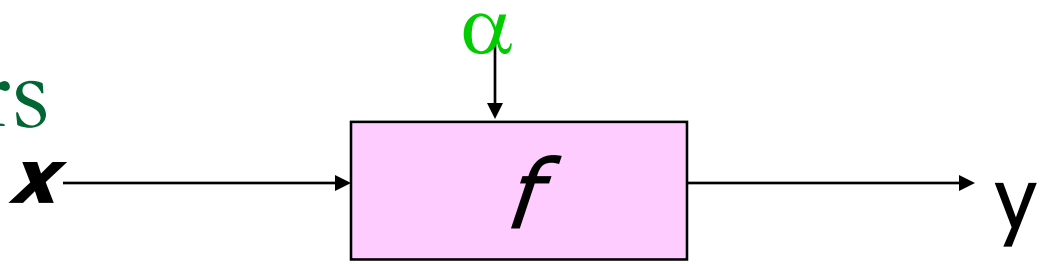
$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$

- denotes +1
- denotes -1



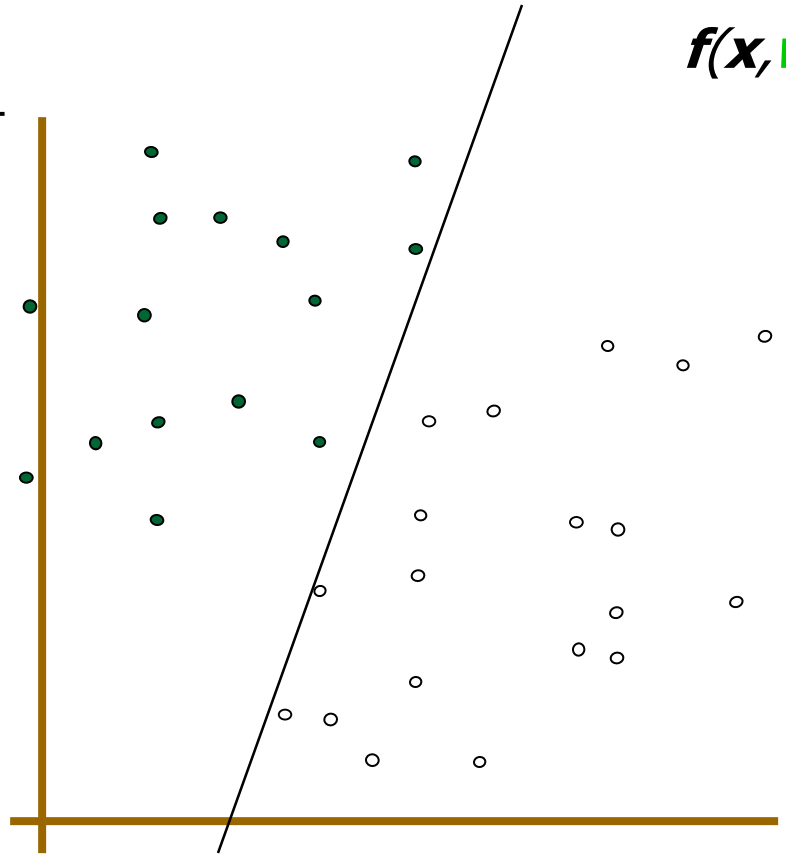
How would you classify this data?

Linear Classifiers



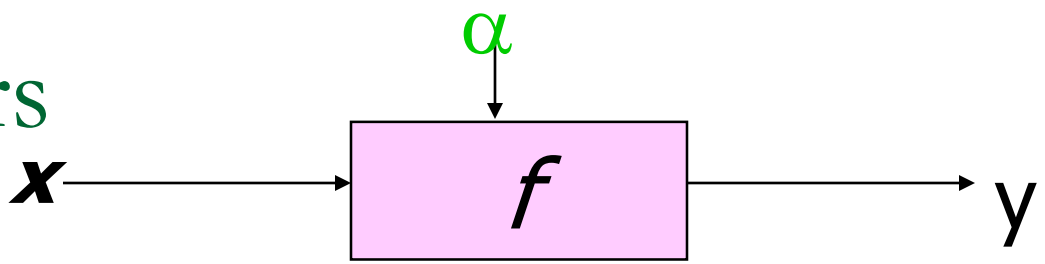
$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$

- denotes +1
- denotes -1



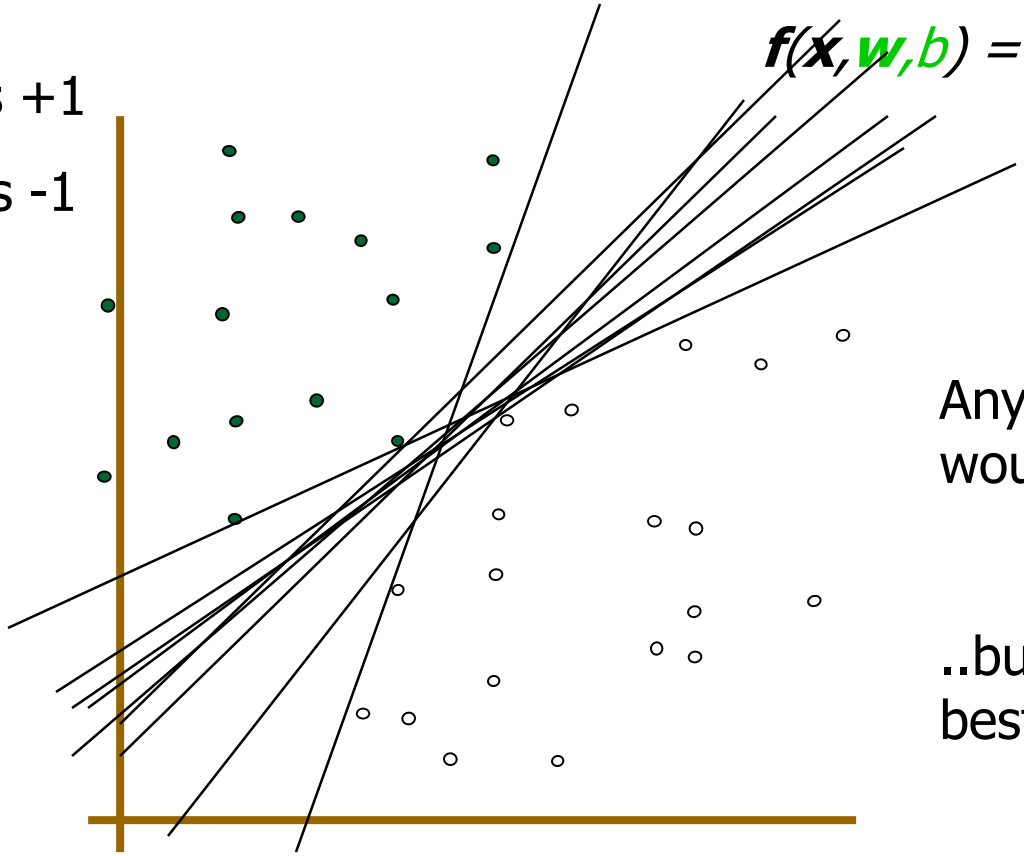
How would you classify this data?

Linear Classifiers



$f(\mathbf{x},\mathbf{w},b) = sign(\mathbf{w}\,\mathbf{x} + b)$

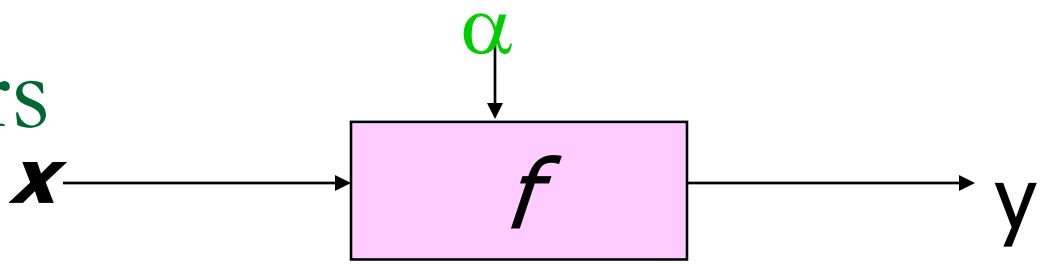
- denotes +1
- denotes -1



Any of these would be fine..

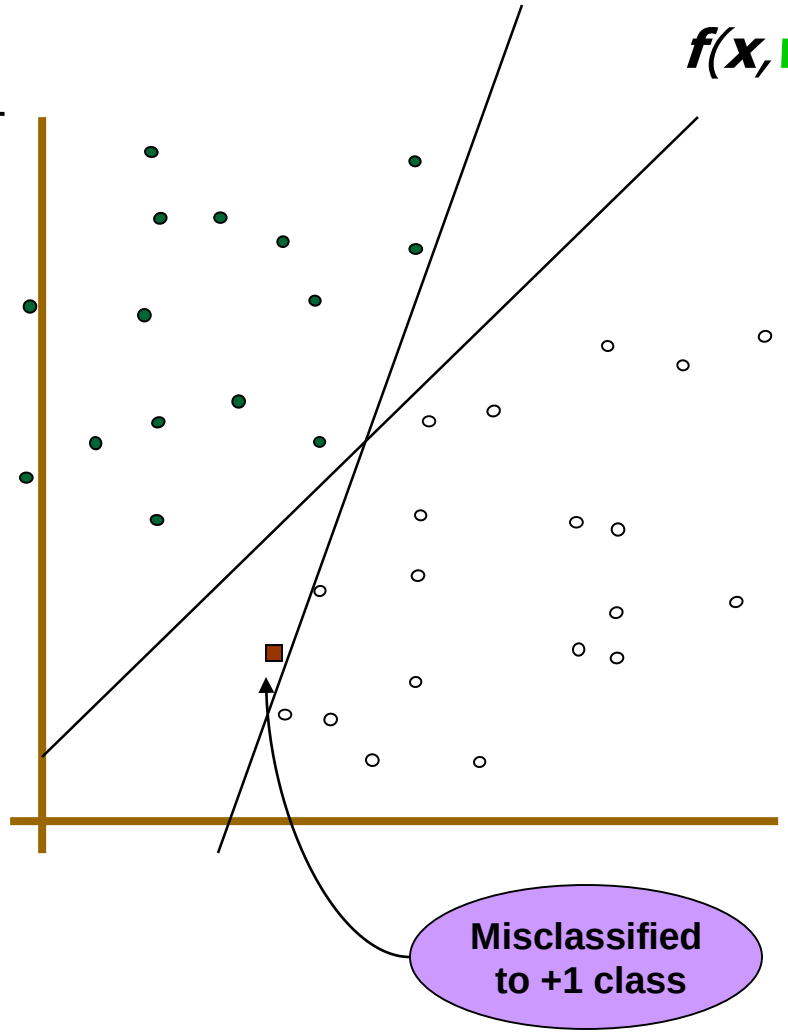
..but which is best?

Linear Classifiers



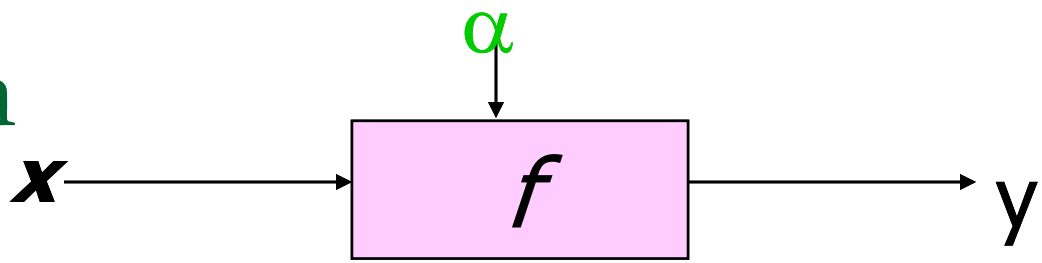
$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$

- denotes +1
- denotes -1



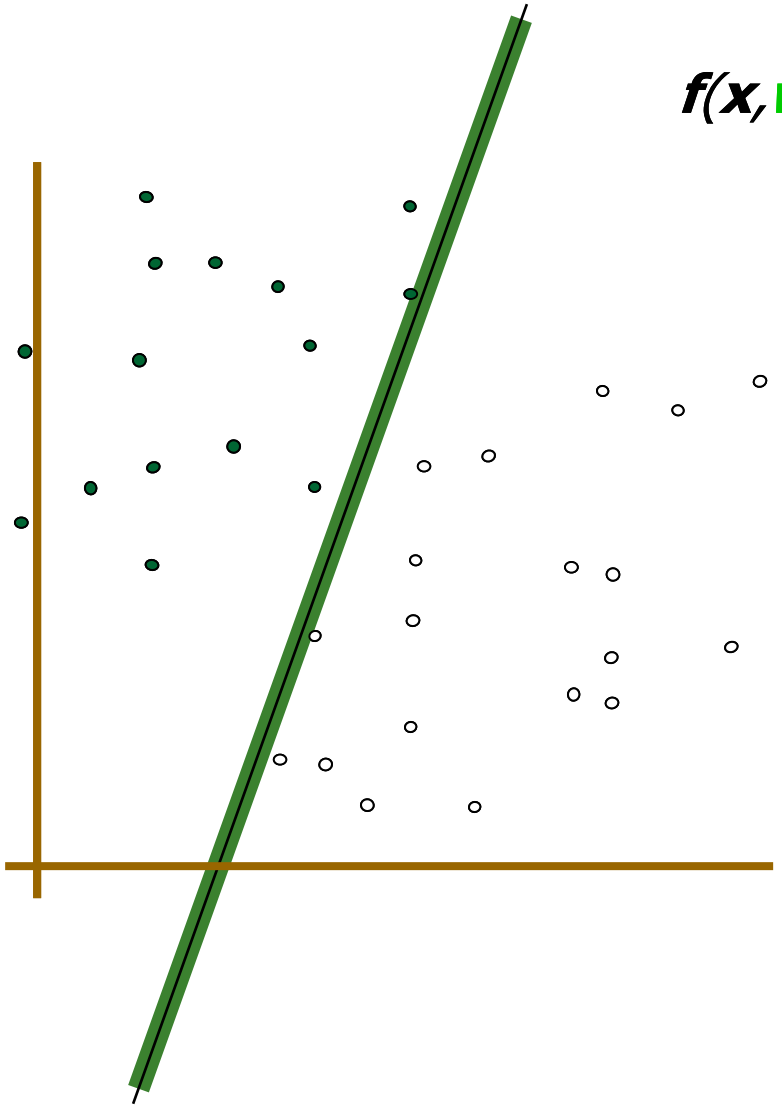
How would you classify this data?

Classifier Margin



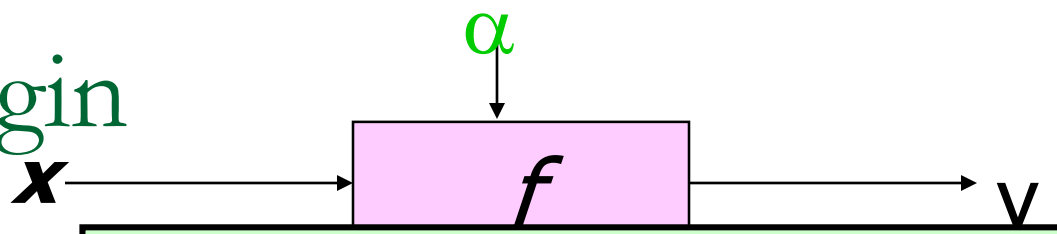
$f(\mathbf{x}, \mathbf{w}, b) = sign(\mathbf{w}\,\mathbf{x} + b)$

- denotes +1
- denotes -1



Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a datapoint.

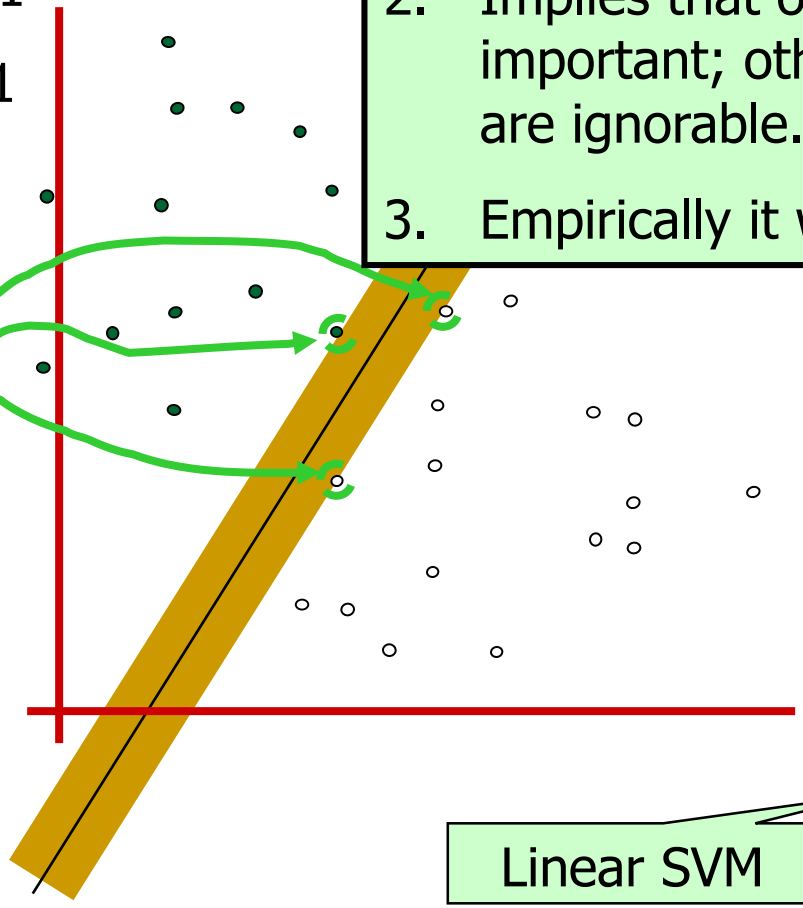
Maximum Margin



1. Maximizing the margin is good according to intuition and PAC theory
2. Implies that only support vectors are important; other training examples are ignorable.
3. Empirically it works very very well.

- denotes +1
- denotes -1

Support Vectors are those datapoints that the margin pushes up against

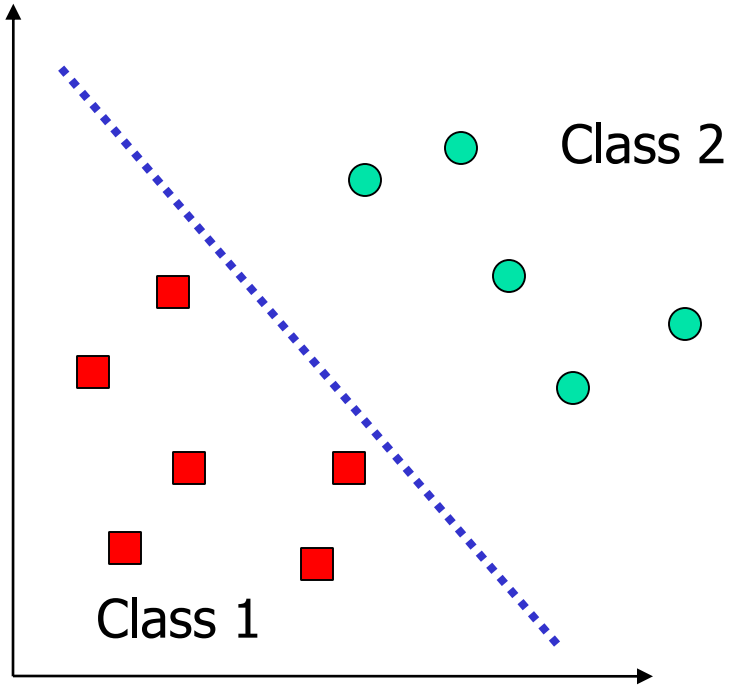


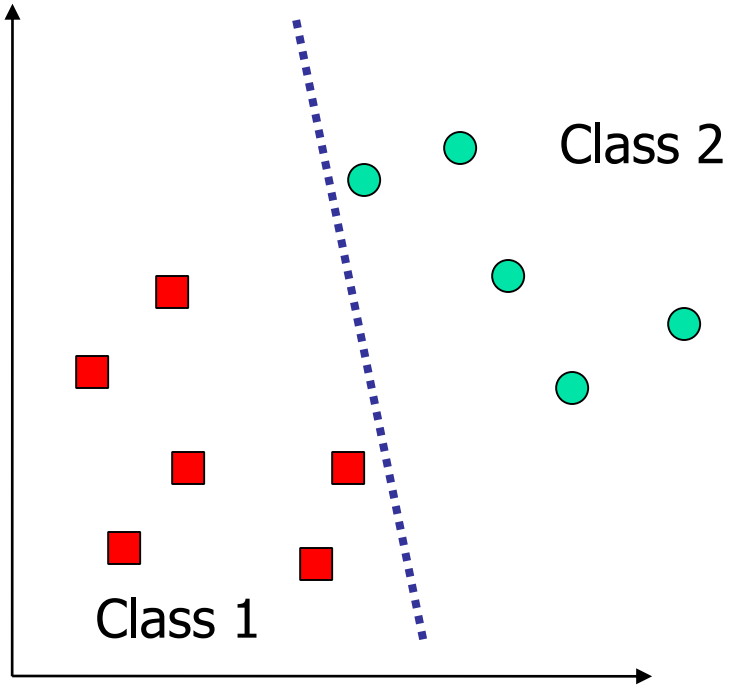
linear classifier with the, um, maximum margin.

This is the simplest kind of SVM (Called an LSVM)

Linear SVM

Example of Bad Decision Boundaries



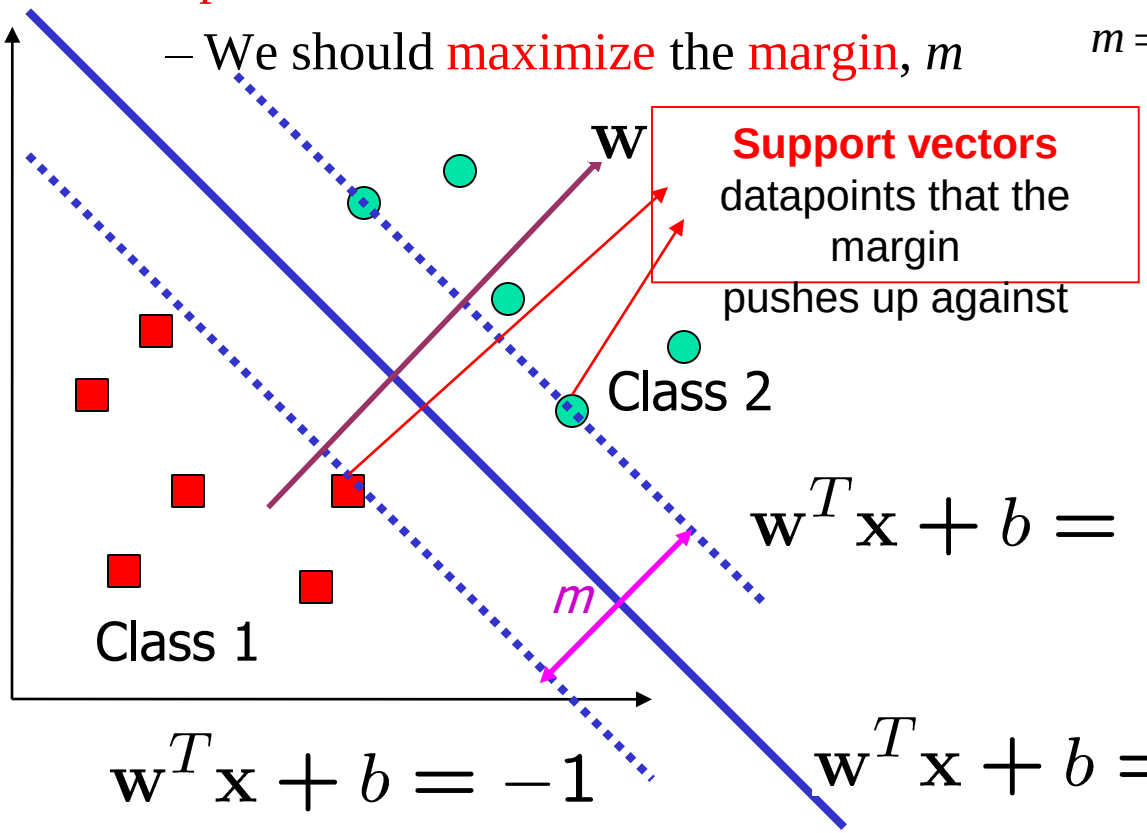


Good Decision Boundary: Margin Should Be Large

The decision boundary should be as far away from the data of both classes as possible

– We should maximize the margin, m

$$m = \frac{2}{\sqrt{w.w}} \qquad m = \frac{2}{||\mathbf{w}||}$$



$$\|\mathbf{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}.$$

The maximum margin linear classifier is the linear classifier with the maximum margin. This is the simplest kind of SVM (Called an Linear SVM)

The Optimization Problem

Let $\{x_1, ..., x_n\}$ be our data set and let $y_i \in \{1,-1\}$ be the class label of x_i

The decision boundary should classify all points correctly $\Rightarrow$

A constrained optimization problem

$$m = \frac{2}{||\mathbf{w}||}$$

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \qquad \forall i$$

$$\text{Minimize } \frac{1}{2}||\mathbf{w}||^2$$

- $||\mathbf{w}||^2 = \mathbf{w}^T\mathbf{w}$

$$\text{subject to } y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \qquad \forall i$$

Lagrangian of Original Problem

Minimize $\frac{1}{2}||\mathbf{w}||^2$

subject to $1-y_i(\mathbf{w}^T\mathbf{x}_i+b) \leq 0$ for $i = 1, \ldots, n$

The Lagrangian is

Lagrangian multipliers

$$\mathcal{L} = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{n} \alpha_i \left(1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right)$$

- Note that $||\mathbf{w}||^2 = \mathbf{w}^T\mathbf{w}$

Setting the gradient of $\mathcal{L}$ w.r.t. **w** and b to zero, we have

$$\mathbf{w} + \sum_{i=1}^{n} \alpha_i(-y_i)\mathbf{x}_i = \mathbf{0} \quad \Rightarrow \quad \boxed{\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i}$$

$$\boxed{\sum_{i=1}^{n} \alpha_i y_i = 0}$$

$\alpha_i \geq 0$

The Dual Optimization Problem

We can transform the problem to its dual

$$\text{max. } W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\text{subject to } \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

Dot product of X

α's → New variables
(Lagrangian multipliers)

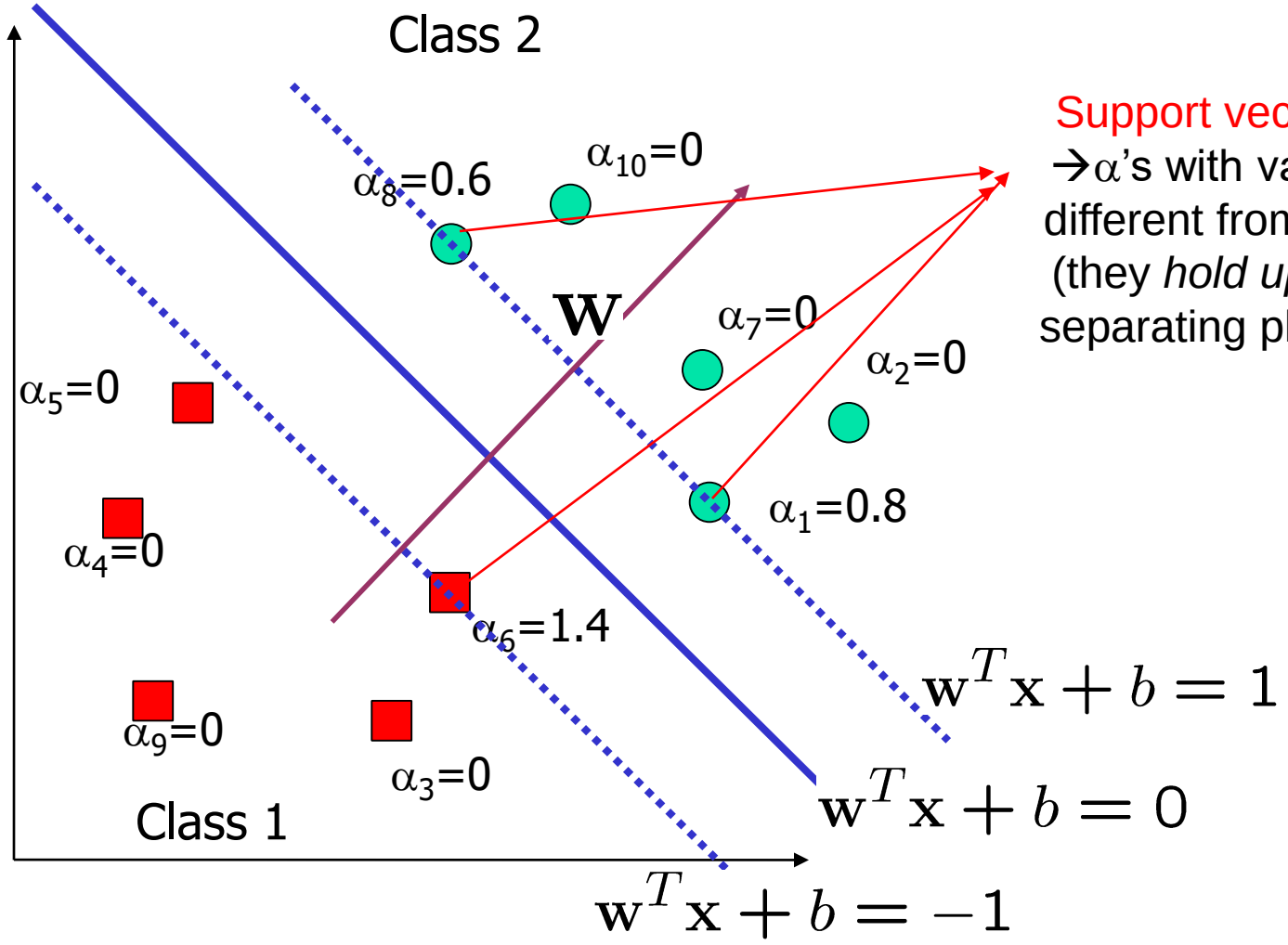
This is a convex quadratic programming (QP) problem

– Global maximum of α_i can always be found

→well established tools for solving this optimization problem (e.g. cplex)

Note: $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$

A Geometrical Interpretation

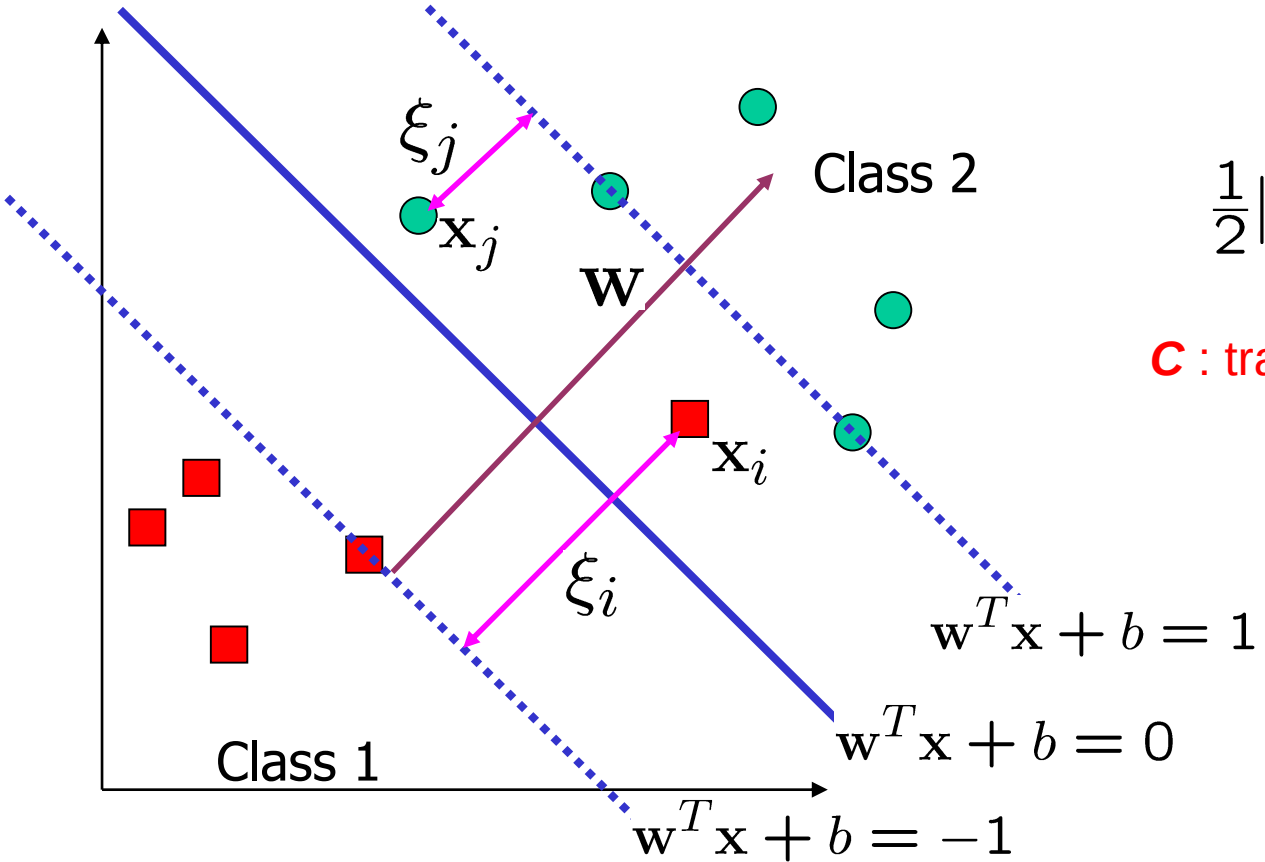


Support vectors
→α's with values different from zero (they *hold up* the separating plane)!

Non-linearly Separable Problems

We allow "error/slack variable" ξ_i in classification; it is based on the output of the discriminant function $\mathbf{w}^T\mathbf{x}+b$

ξ_i approximates the number of misclassified samples



New objective function:

$$\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

C : tradeoff parameter between error and margin; chosen by the user; large C means a higher penalty to errors

The Optimization Problem

$$\text{max. } W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\text{subject to } \boxed{C \geq \alpha_i \geq 0}, \sum_{i=1}^{n} \alpha_i y_i = 0$$

$$\mathbf{w} = \sum_{j=1}^{s} \alpha_{t_j} y_{t_j} \mathbf{x}_{t_j}$$

w is also recovered as

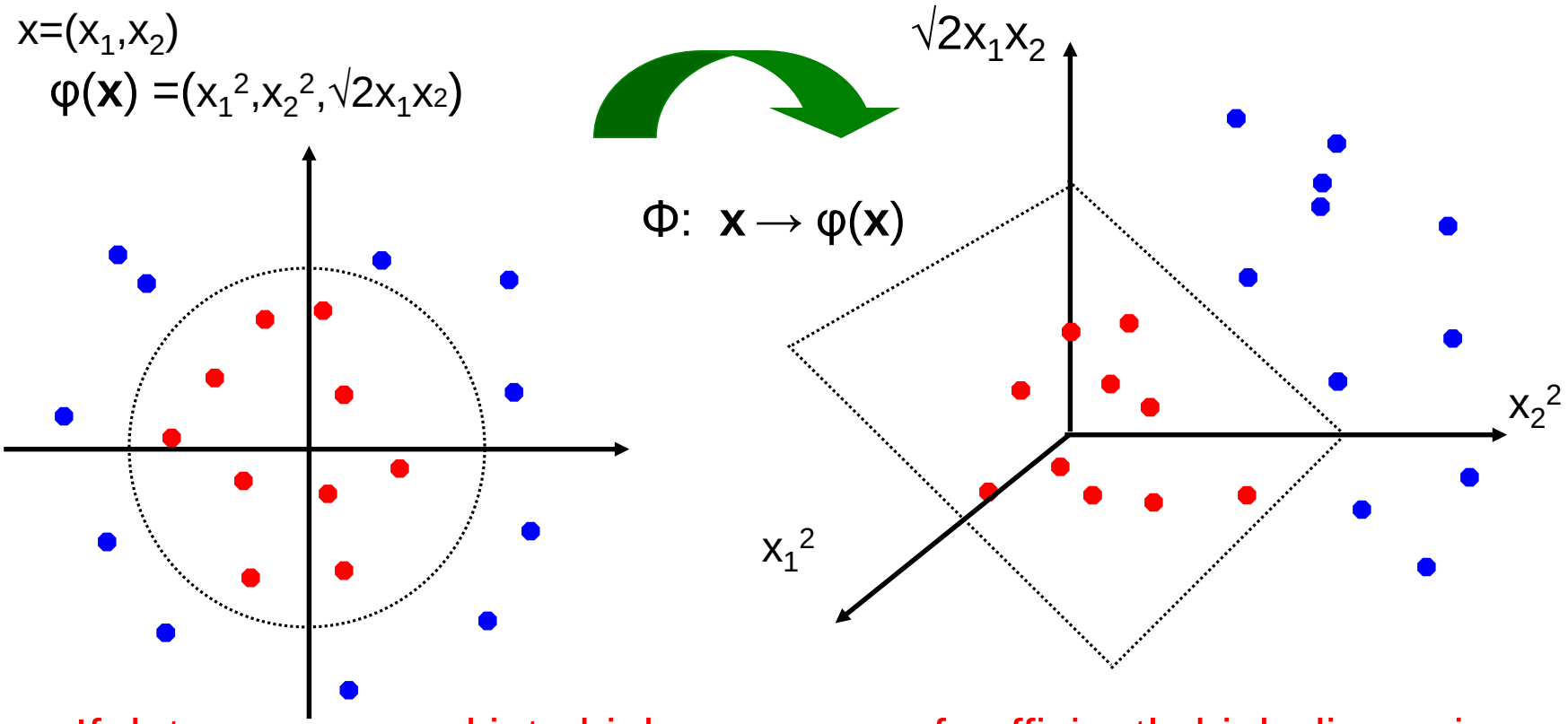
The only difference with the linear separable case is that there is an upper bound *C* on α_i

Once again, a QP solver can be used to find α_i efficiently!!!

Extension to Non-linear SVMs (Kernel Machines)

Non-linear SVMs: Feature Space

General idea: the original input space (x) can be mapped to some higher-dimensional feature space ($\varphi(\mathbf{x})$)where the training set is separable:



If data are mapped into higher a space of sufficiently high dimension, then they will in general be linearly separable;
N data points are in general separable in a space of N-1 dimensions or more!!!

This slide is courtesy of *www.iro.umontreal.ca/~pift6080/documents/papers/**svm**_tutorial.**ppt***

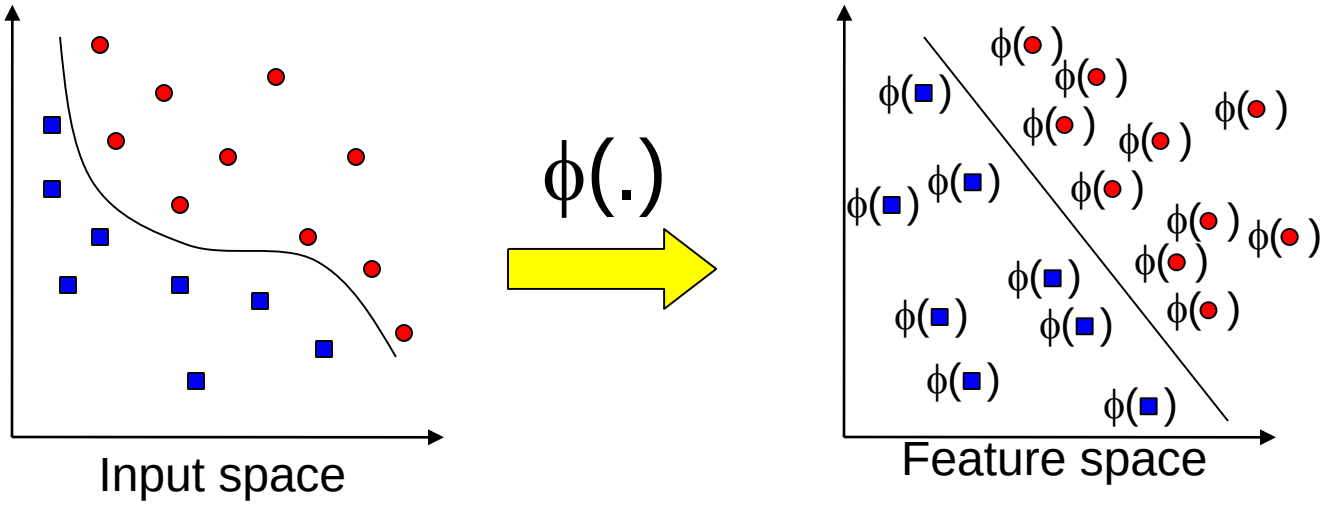
Transformation to Feature Space

Possible problem of the transformation

- High computation burden due to high-dimensionality and hard to get a good estimate

SVM solves these two issues simultaneously

- "Kernel tricks" for efficient computation
- Minimize $\|\mathbf{w}\|^2$ can lead to a "good" classifier



Kernel Trick ☺

Recall:

Note that data only appears as dot products

maximize
$$\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=j=1}^{N} \alpha_i \alpha_j y_i y_j x_i x_j$$

subject to
$$C \geq \alpha_i \geq 0, \sum_{i=1}^{N} \alpha_i y_i = 0$$

Since data is only represented as dot products, we need not do the mapping explicitly.

Introduce a Kernel Function (*) K such that:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

(*)Kernel function – a function that can be applied to pairs of input data to evaluate dot products in some corresponding feature space

Example Transformation

Consider the following transformation

$$\phi(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$$
$$\phi(\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}) = (1, \sqrt{2}y_1, \sqrt{2}y_2, y_1^2, y_2^2, \sqrt{2}y_1y_2)$$

Define the kernel function K (**x**,**y**) as

$$\begin{aligned} \langle \phi(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}), \phi(\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}) \rangle &= (1 + x_1y_1 + x_2y_2)^2 \\ &= K(\mathbf{x}, \mathbf{y}) \end{aligned}$$

$$K(\mathbf{x}, \mathbf{y}) = (1 + x_1y_1 + x_2y_2)^2$$

The inner product $\phi(.)\phi(.)$ can be computed by K without going through the map $\phi(.)$ explicitly!!!

Modification Due to Kernel Function

Change all inner products to kernel functions

For training,

Original

$$\text{max. } W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\text{subject to } C \geq \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

With kernel function

$$\text{max. } W(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1,j=1}^{n} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$\text{subject to } C \geq \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

Examples of Kernel Functions

Polynomial kernel with degree *d*

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + 1)^d$$

Radial basis function kernel with width σ

$$K(\mathbf{x}, \mathbf{y}) = \exp(-||\mathbf{x} - \mathbf{y}||^2/(2\sigma^2))$$

– Closely related to radial basis function neural networks

Sigmoid with parameter κ and θ

$$K(\mathbf{x}, \mathbf{y}) = \tanh(\kappa\mathbf{x}^T\mathbf{y} + \theta)$$

– It does not satisfy the Mercer condition on all κ and θ

Research on different kernel functions in different applications is very active

Example

Suppose we have 5 1D data points

- x_1=1, x_2=2, x_3=4, x_4=5, x_5=6, with 1, 2, 6 as class 1 and 4, 5 as class 2 $\Rightarrow$ y_1=1, y_2=1, y_3=-1, y_4=-1, y_5=-1, y_6=1

We use the polynomial kernel of degree 2

- $K(x,y) = (xy+1)^2$
- C is set to 100

We first find α_i (i=1, …, 5) by

$$\text{max.} \sum_{i=1}^{5} \alpha_i - \frac{1}{2} \sum_{i=1}^{5} \sum_{i=1}^{5} \alpha_i \alpha_j y_i y_j (x_i x_j + 1)^2$$

$$\text{subject to } 100 \geq \alpha_i \geq 0, \sum_{i=1}^{5} \alpha_i y_i = 0$$

Example

By using a QP solver, we get

α_1=0, α_2=2.5, α_3=0, α_4=7.333, α_5=4.833

– Verify (at home) that the constraints are indeed satisfied

– The support vectors are {x_2=2, x_4=5, x_5=6}

The discriminant function is

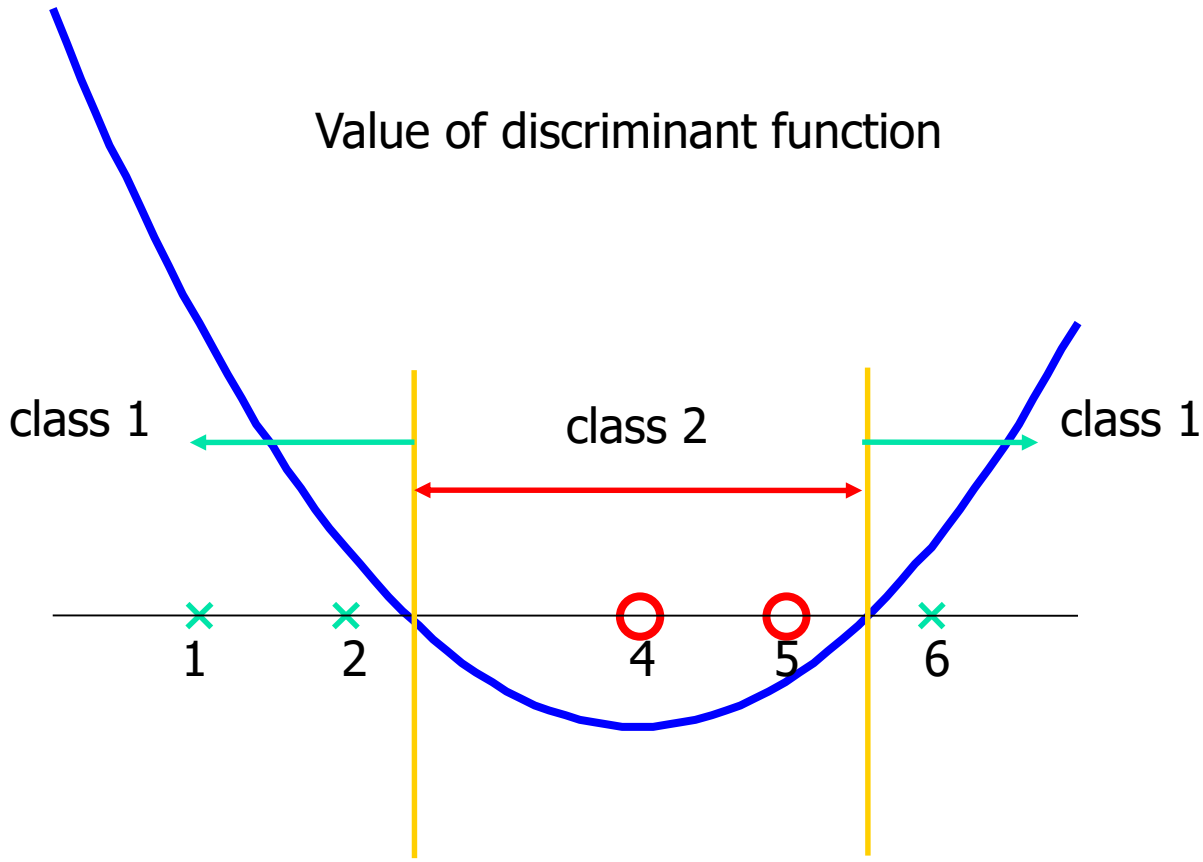
$$f(y) = 2.5(1)(2y+1)^2 + 7.333(-1)(5y+1)^2 + 4.833(1)(6y+1)^2 + b$$
$$= 0.6667x^2 - 5.333x + b$$

b is recovered by solving f(2)=1 or by f(5)=-1 or by f(6)=1, as x_2, x_4, x_5 lie on and all give b=9

$$y_i(\mathbf{w}^T\phi(z) + b) = 1$$

$$\rightarrow \quad f(y) = 0.6667x^2 - 5.333x + 9$$

Example


Value of discriminant function
class 1
class 2
class 1
1
2
4
5
6

Choosing the Kernel Function

Probably the most tricky part of using SVM.

The kernel function is important because it creates the kernel matrix, which summarizes all the data

Many principles have been proposed (diffusion kernel, Fisher kernel, string kernel, …)

There is even research to estimate the kernel matrix from available information

In practice, a low degree polynomial kernel or RBF kernel with a reasonable width is a good initial try

Note that SVM with RBF kernel is closely related to RBF neural networks, with the centers of the radial basis functions automatically chosen for SVM

Software

A list of SVM implementation can be found at
http://www.kernel-machines.org/software.html

Some implementation (such as LIBSVM) can handle multi-class classification

SVMLight is among one of the earliest implementation of SVM

Several Matlab toolboxes for SVM are also available

Recap of Steps in SVM

Prepare data matrix $\{(x_i,y_i)\}$

Select a Kernel function

Select the error parameter *C*

"Train" the system (to find all α_i)

New data can be classified using α_i and Support Vectors

Summary

Weaknesses

- Training (and Testing) is quite slow compared to ANN
  - Because of Constrained Quadratic Programming
- Essentially a binary classifier
  - However, there are some tricks to evade this.
- Very sensitive to noise
  - A few off data points can completely throw off the algorithm
- Biggest Drawback: The choice of Kernel function.
  - There is no “set-in-stone” theory for choosing a kernel function for any given problem (still in research...)
  - Once a kernel function is chosen, there is only ONE modifiable parameter, the error penalty *C*.

Summary

Strengths

- Training is relatively easy
  - We don't have to deal with local minimum like in ANN
  - SVM solution is always global and unique (check "Burges" paper for proof and justification).
- Unlike ANN, doesn't suffer from "curse of dimensionality".
  - How? Why? We have infinite dimensions?!
  - Maximum Margin Constraint: DOT-PRODUCTS!
- Less prone to overfitting
- Simple, easy to understand geometric interpretation.
  - No large networks to mess around with.

Applications of SVMs

- Bioinformatics
- Machine Vision
- Text Categorization
- Ranking (e.g., Google searches)
- Handwritten Character Recognition
- Time series analysis

→**Lots of very successful applications!!!**

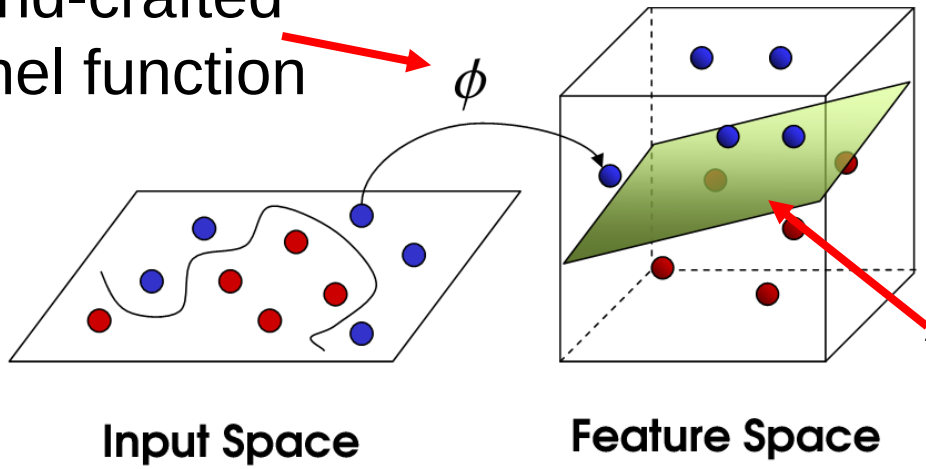
Reference

- **Support Vector Machine Classification of Microarray Gene Expression Data**, Michael P. S. Brown William Noble Grundy, David Lin, Nello Cristianini, Charles Sugnet, Manuel Ares, Jr., David Haussler
- www.cs.utexas.edu/users/mooney/cs391L/svm.**ppt**
- **Text categorization with Support Vector Machines:**
  learning with many relevant features

  T. Joachims, ECML - 98

SVM

Hand-crafted kernel function



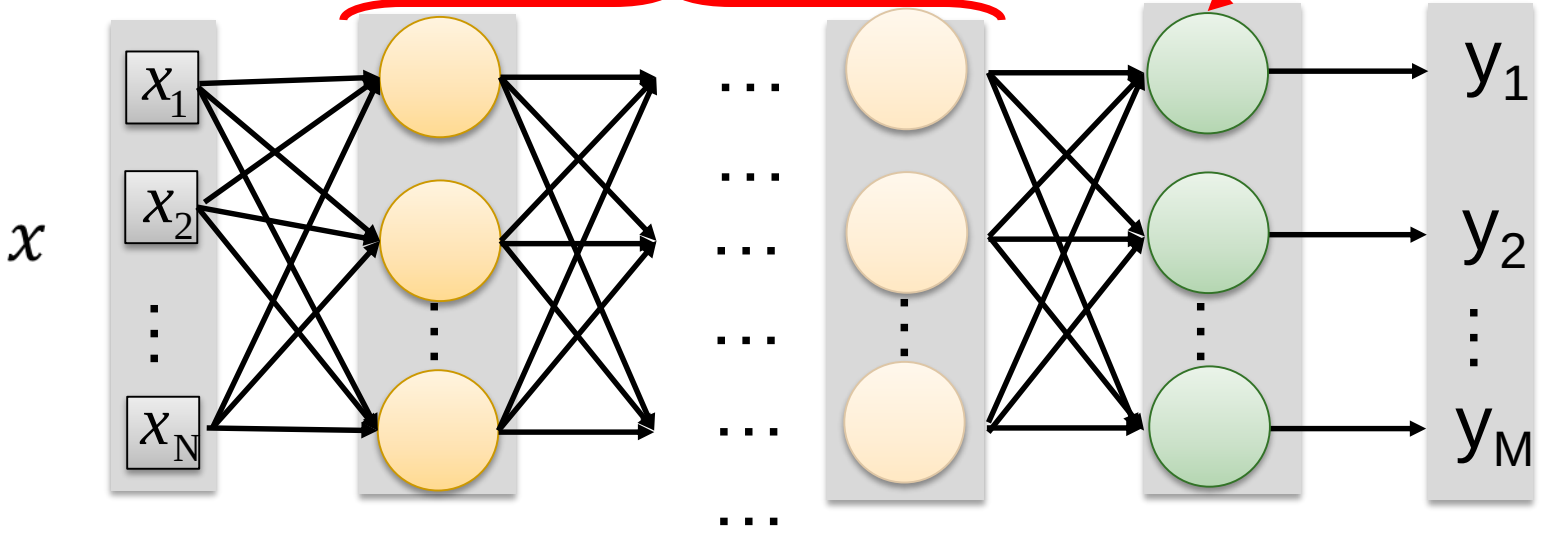
Apply simple classifier

Source of image: http://www.gipsa-lab.grenoble-inp.fr/transfert/seminaire/455_Kadri2013Gipsa-lab.pdf

Deep Learning

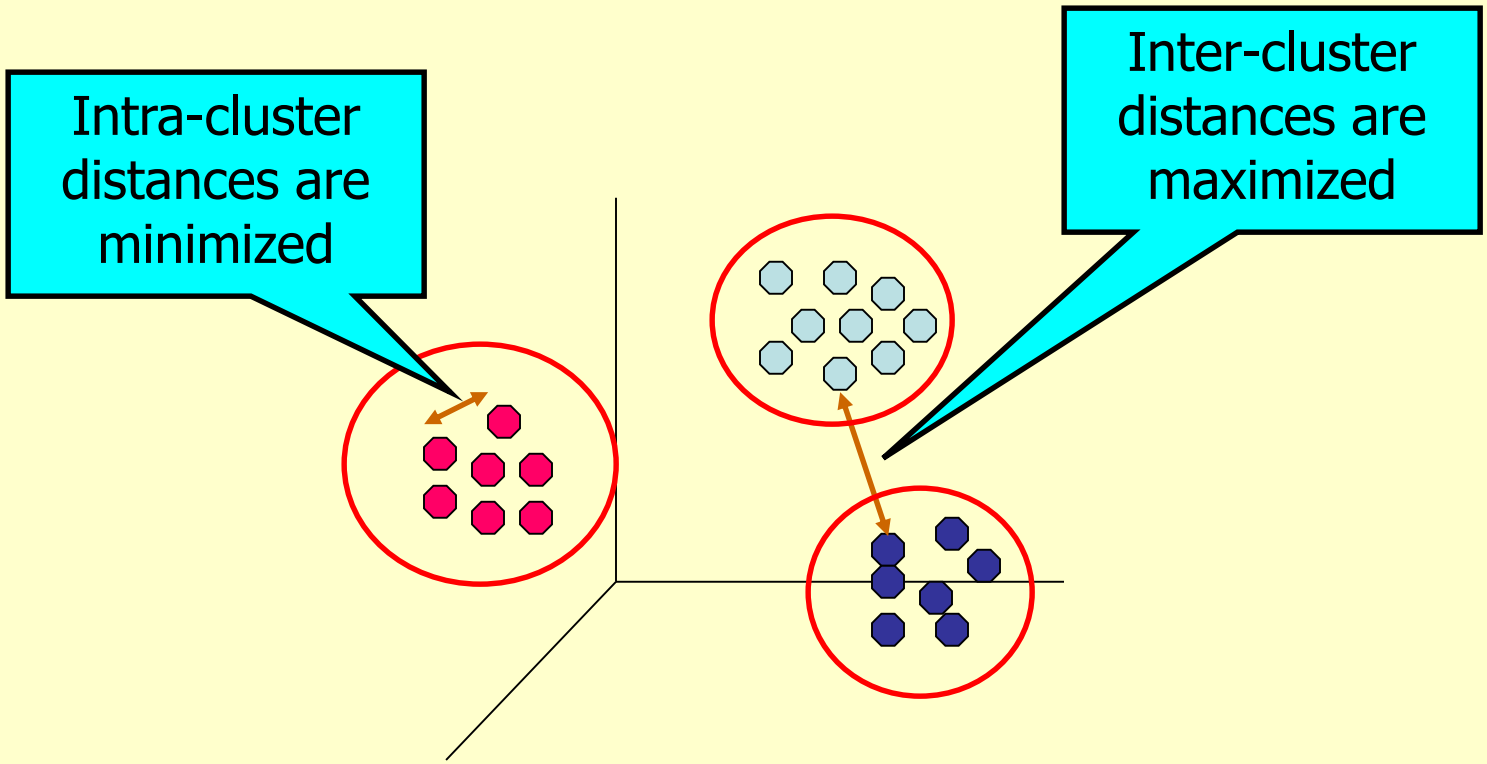
Learnable kernel $\phi(x$

simple classifier



What is Cluster Analysis?

- Finding groups of objects such that the objects in a group will be similar (or related) to one another and different from (or unrelated to) the objects in other groups



Clustering

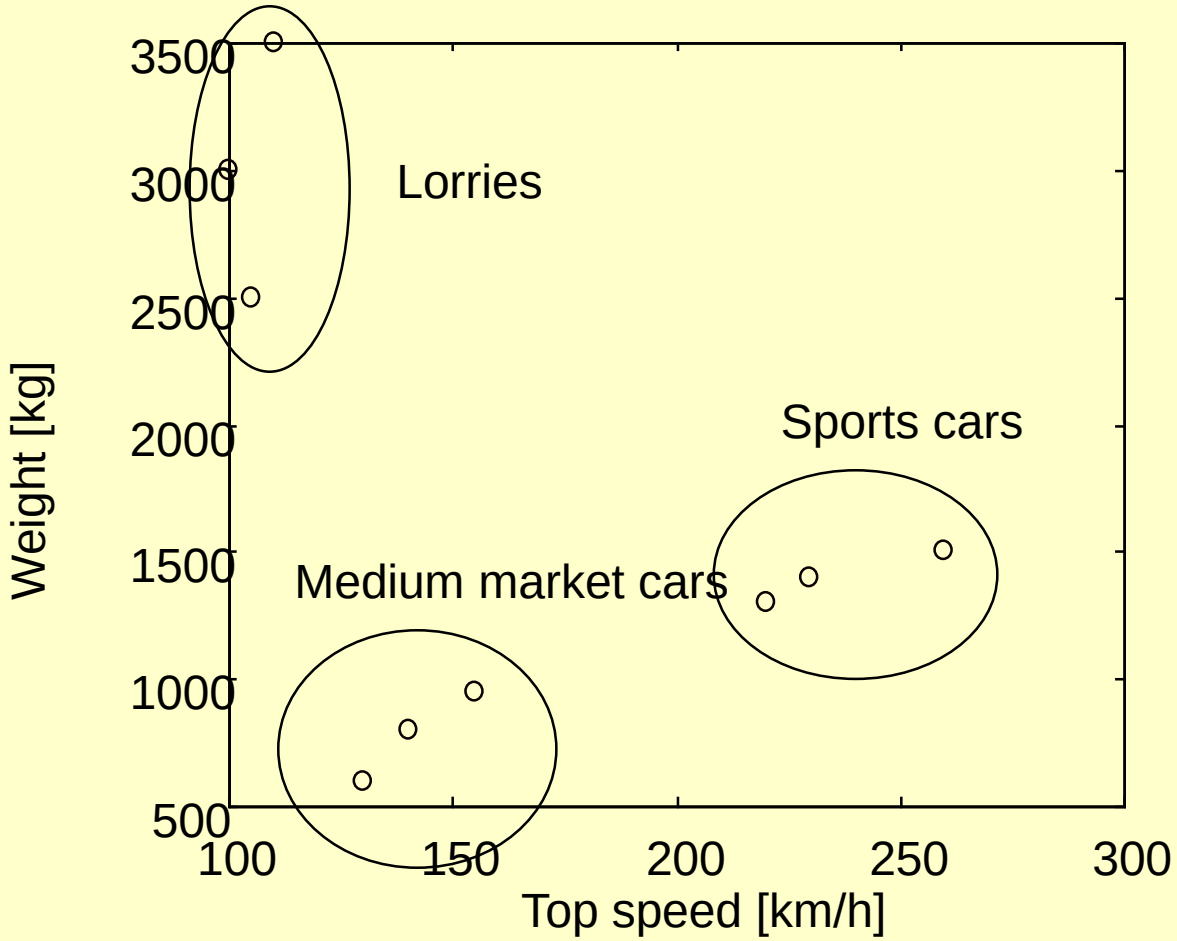
- However,
  - Similarity is hard to define, and …. “We know it when we see it”
  - The real meaning of similarity is a philosophical question. We will take a more pragmatic approach.

PC: http://www.funnyphotos.net.au/dog-owners/

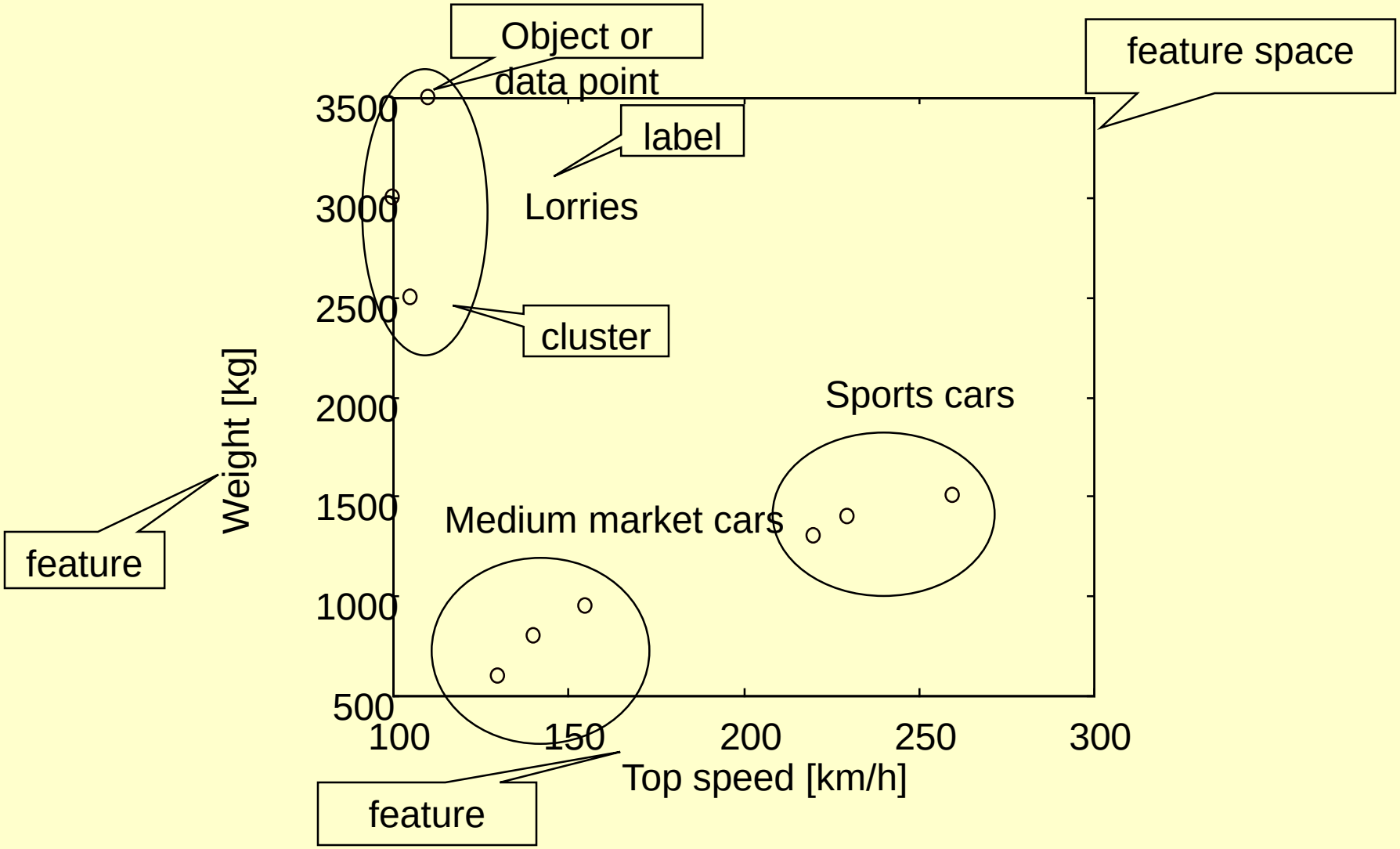
Vehicle Example

Vehicle	Top speed km/h	Colour	Air resistance	Weight Kg
V1	220	red	0.30	1300
V2	230	black	0.32	1400
V3	260	red	0.29	1500
V4	140	gray	0.35	800
V5	155	blue	0.33	950
V6	130	white	0.40	600
V7	100	black	0.50	3000
V8	105	red	0.60	2500
V9	110	gray	0.55	3500

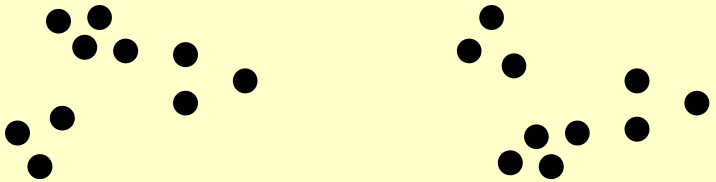
Vehicle Clusters



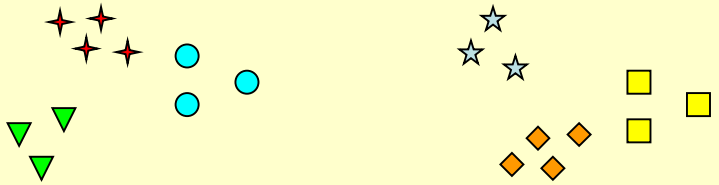
Terminology



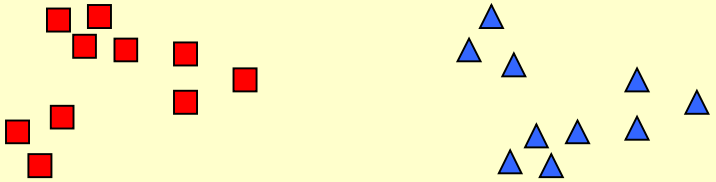
Clustering – is an ill-defined problem!!



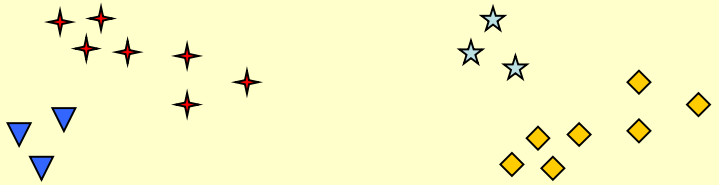
How many clusters?



Six Clusters



Two Clusters



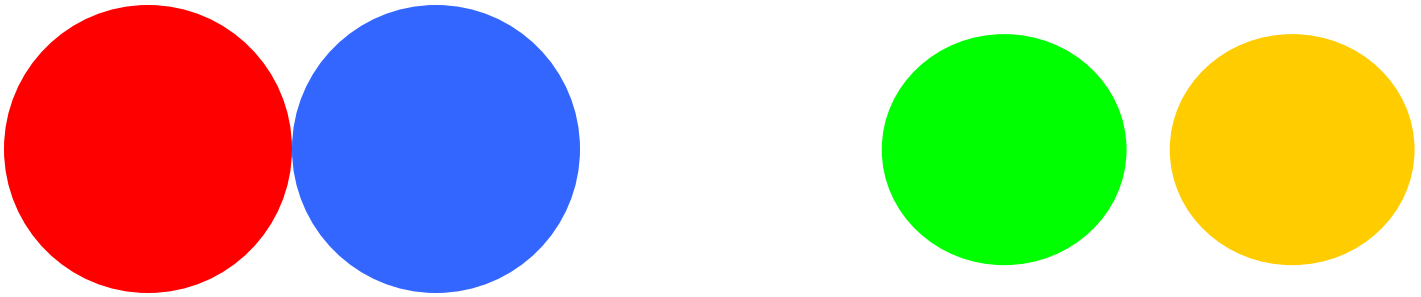
Four Clusters

Types of Clustering

- A clustering procedure outputs a set of clusters
- Important distinction between hierarchical and partitional sets of clusters
- Partitional Clustering
  - A division data objects into non-overlapping subsets (clusters) such that each data object is in exactly one subset
- Hierarchical clustering
  - A set of nested clusters organized as a hierarchical tree

Types of Clusters: Center-Based

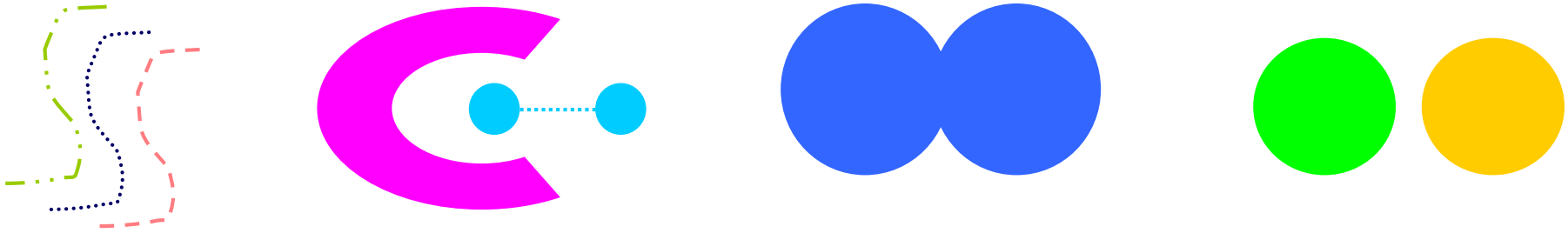
- Center-based
  - A cluster is a set of objects such that an object in a cluster is closer (more similar) to the “center” of a cluster, than to the center of any other cluster
  - The center of a cluster is often a centroid, the average of all the points in the cluster, or a medoid, the most “representative” point of a cluster



4 center-based clusters

Types of Clusters: Contiguity-Based

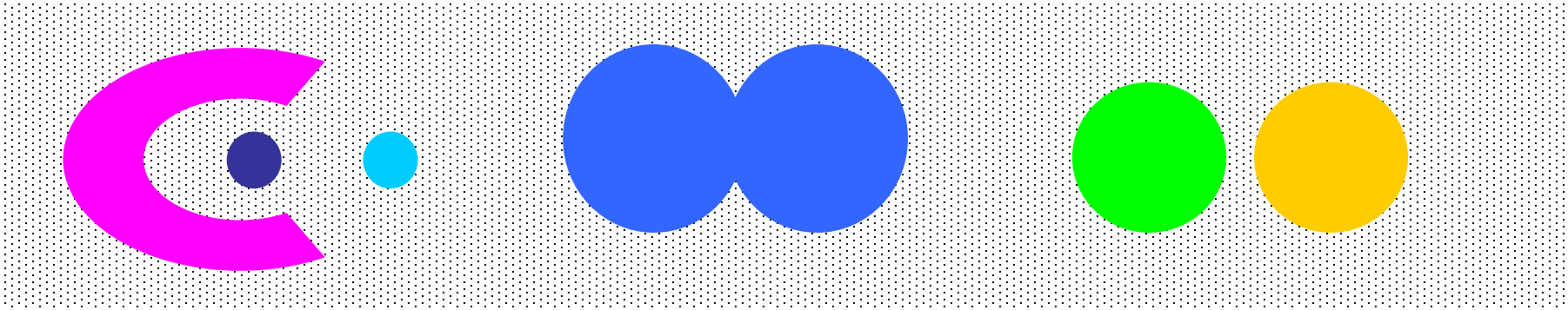
- Contiguous Cluster (Nearest neighbor or Transitive)
  - A cluster is a set of points such that a point in a cluster is closer (or more similar) to one or more other points in the cluster than to any point not in the cluster.



8 contiguous clusters

Types of Clusters: Density-Based

- Density-based
  - A cluster is a dense region of points, which is separated by low-density regions, from other regions of high density.
  - Used when the clusters are irregular or intertwined, and when noise and outliers are present.

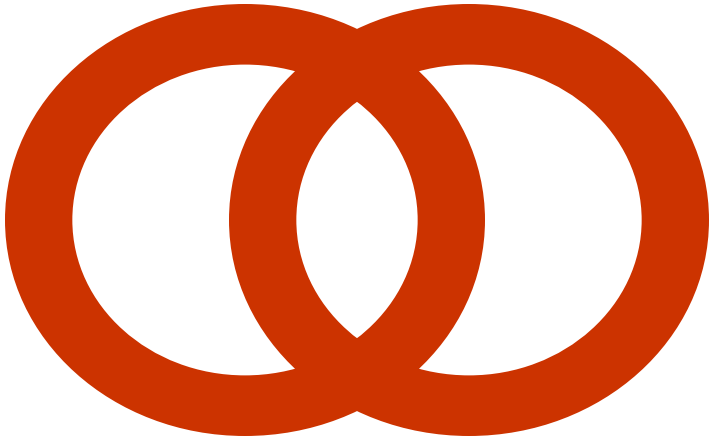


6 density-based clusters

Types of Clusters: Conceptual Clusters

- Shared Property or Conceptual Clusters
  - Finds clusters that share some common property or represent a particular concept.

.



2 Overlapping Circles

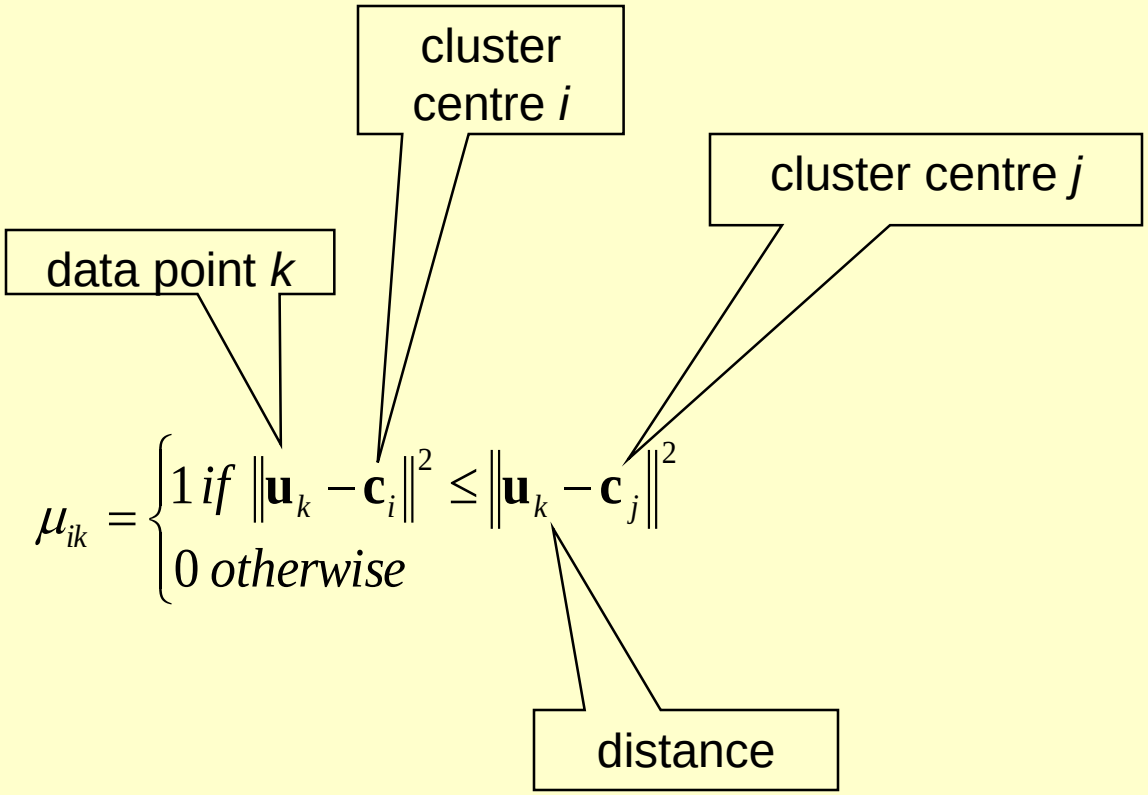
Types of Clusters: Objective Function

- Clusters Defined by an Objective Function
  - Finds clusters that minimize or maximize an objective function.
  - Enumerate all possible ways of dividing the points into clusters and evaluate the `goodness' of each potential set of clusters by using the given objective function. (No. of feasible partitions could be very large and the problem is NP Hard)
  - Can have global or local objectives.
    - Hierarchical clustering algorithms typically have local objectives
    - Partitional algorithms typically have global objectives
  - A variation of the global objective function approach is to fit the data to a parameterized model.
    - Parameters for the model are determined from the data.
    - Mixture models assume that the data is a ‘mixture' of a number of statistical distributions.

Types of Clusters: Objective Function …

- Map the clustering problem to a different domain and solve a related problem in that domain
  - Proximity matrix defines a weighted graph, where the nodes are the points being clustered, and the weighted edges represent the proximities between points
  - Clustering is equivalent to breaking the graph into connected components, one for each cluster.
  - Want to minimize the edge weight between clusters and maximize the edge weight within clusters

Membership matrix **M**



c-partition

All clusters *C* together fills the whole universe *U*

Clusters do not overlap

$$\bigcup_{i=1}^{c} C_i = U$$

$$C_i \cap C_j = \emptyset \quad for\ all\ i \neq j$$

$$\emptyset \subset C_i \subset U \quad for\ all\ i$$

$$2 \leq c \leq K$$

A cluster *C* is never empty and it is smaller than the whole universe *U*

There must be at least 2 clusters in a c-partition and at most as many as the number of data points *K*

Objective Functions: Hard and Fuzzy C Means

Minimise the total sum of all distances

$$J_{k-means} = \sum_{i=1}^{c} J_i = \sum_{i=1}^{c} \left(\sum_{k, \mathbf{u}_k \in C_i} \left\| \mathbf{u}_k - \mathbf{c}_i \right\|^2 \right)$$

$$J_{FCM} = \sum_{i=1}^{c} J_i = \sum_{i=1}^{c} \sum_{k=1}^{n} \mu_{ik}^{m} \left\| \mathbf{u}_k - \mathbf{c}_i \right\|^2$$

K-Means Type Clustering Algorithms

- Given *k*, the *k-means* algorithm consists of four steps:
  - Select initial representative points (means or centroids) at random.
  - Assign each object to the cluster with the nearest centroid.
  - Compute each centroid as the mean of the objects assigned to it.
  - Repeat previous 2 steps until no change.

The Objective Function of K-Means

- Most common measure is Sum of Squared Error (SSE) or summed Intra Cluster Spread (ICS)
  - For each point, the error is the distance to the nearest cluster center
  - To get SSE, we square these errors and sum them.

$$SSE = ICS(\vec{c}_1, \vec{c}_2, \ldots, \vec{c}_k) = \sum_{j=1}^{k} \sum_{\vec{X}_i \in C_j} d^2(\vec{X}_i, \vec{c}_j)$$

  - $\boldsymbol{X}_i$ is a data point in cluster C_j and $\boldsymbol{c}_j$ is the representative point (most often the mean) for cluster C_j .
  - Given two clusters, we can choose the one with the smallest error
  - One easy way to reduce SSE is to increase *k*, the number of clusters
    - A good clustering with smaller *k* can have a higher SSE than a poor clustering with higher *k*

Fuzzy *C* Means: Fuzzy membership matrix **M**

Point *k*'s membership of cluster *i*

Fuzziness exponent

$$\mu_{ik} = \frac{1}{\sum_{j=1}^{c} \left(\frac{d_{ik}}{d_{jk}} \right)^{2/(q-1)}}$$

Distance from point *k* to current cluster centre *i*

Distance from point *k* to other cluster centres *j*

$$d_{ik} = \left\| \mathbf{u}_k - \mathbf{c}_i \right\|$$

Fuzzy membership matrix **M**

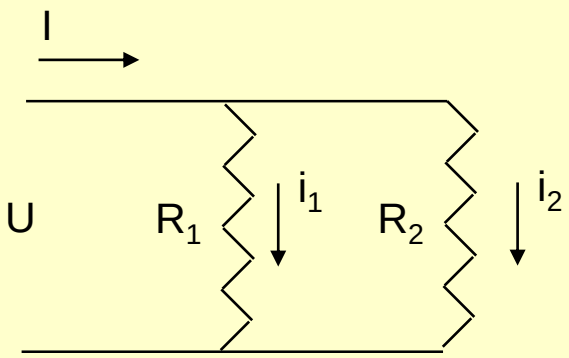
$$\mu_{ik} = \frac{1}{\sum_{j=1}^{c}\left(\frac{d_{ik}}{d_{jk}}\right)^{2/(q-1)}}$$

$$= \frac{1}{\left(\frac{d_{ik}}{d_{1k}}\right)^{2/(q-1)} + \left(\frac{d_{ik}}{d_{2k}}\right)^{2/(q-1)} + \cdots + \left(\frac{d_{ik}}{d_{ck}}\right)^{2/(q-1)}}$$

$$= \frac{\frac{1}{d_{ik}^{\,2/(q-1)}}}{\frac{1}{d_{1k}^{\,2/(q-1)}} + \frac{1}{d_{2k}^{\,2/(q-1)}} + \cdots + \frac{1}{d_{ck}^{\,2/(q-1)}}}$$

Gravitation to cluster *i* relative to total gravitation

Electrical Analogy



$$U = RI$$

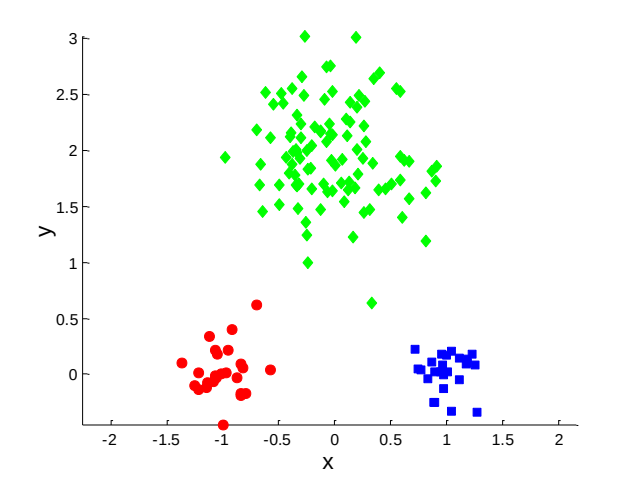
$$R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_c}}$$

$$R\frac{1}{R_i} = \frac{\frac{1}{R_i}}{\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_c}}$$

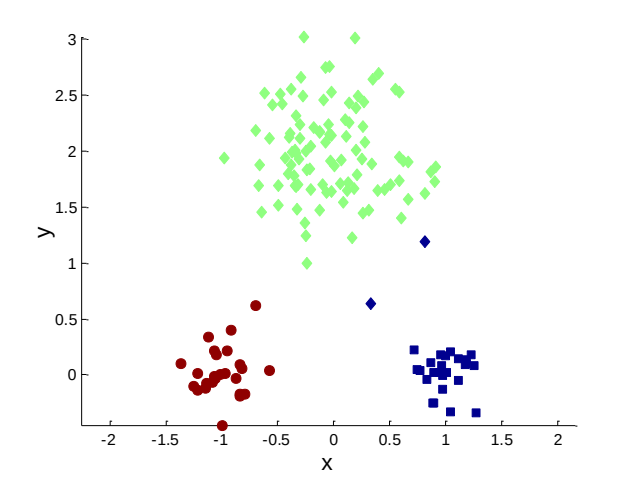
Same form as m_{ik}

$$R\frac{1}{R_i} = \frac{U}{I}\frac{1}{\frac{U}{i_i}} = \frac{i_i}{I}$$

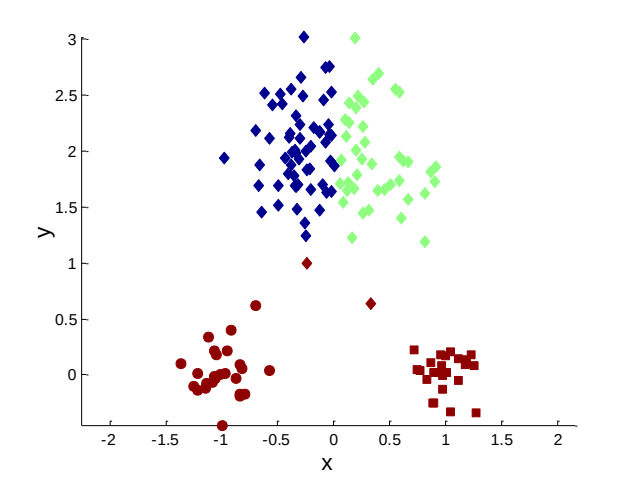
Two different K-means Clustering



Original Points

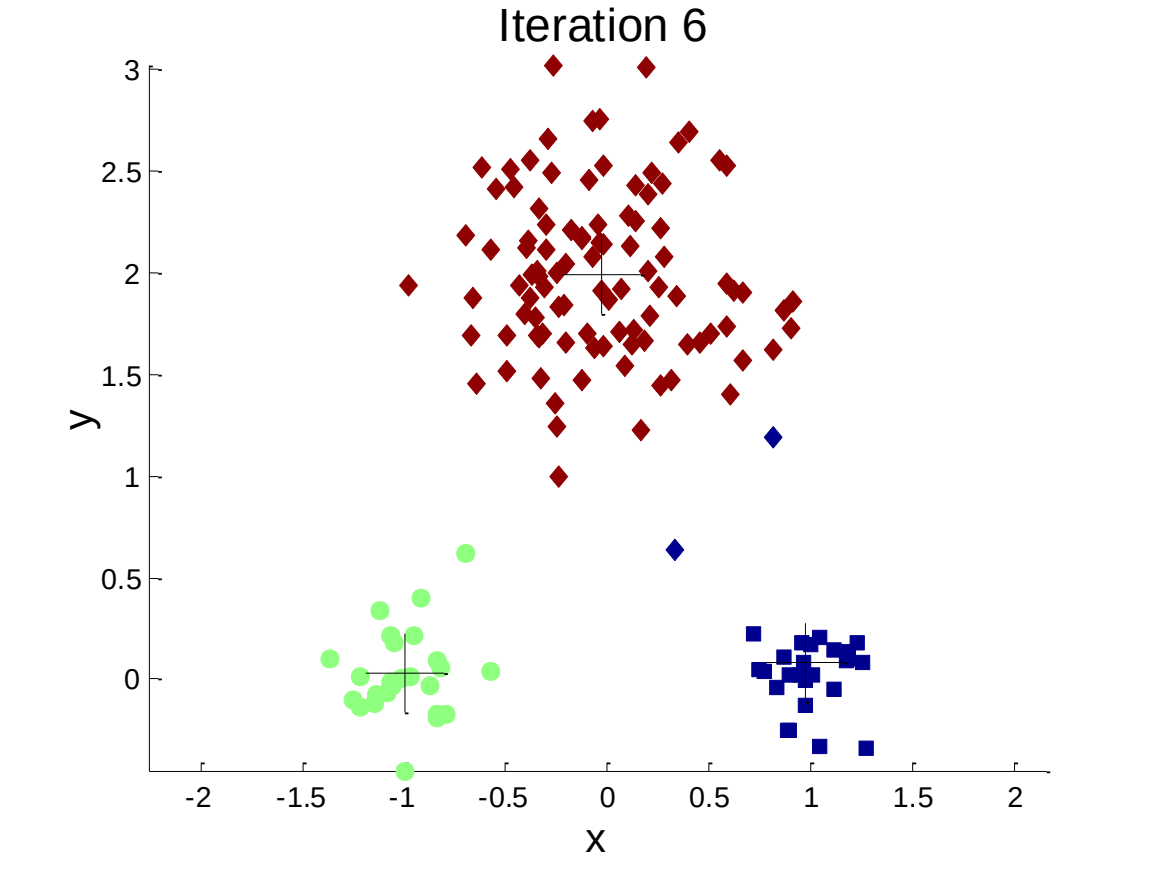


Optimal Clustering

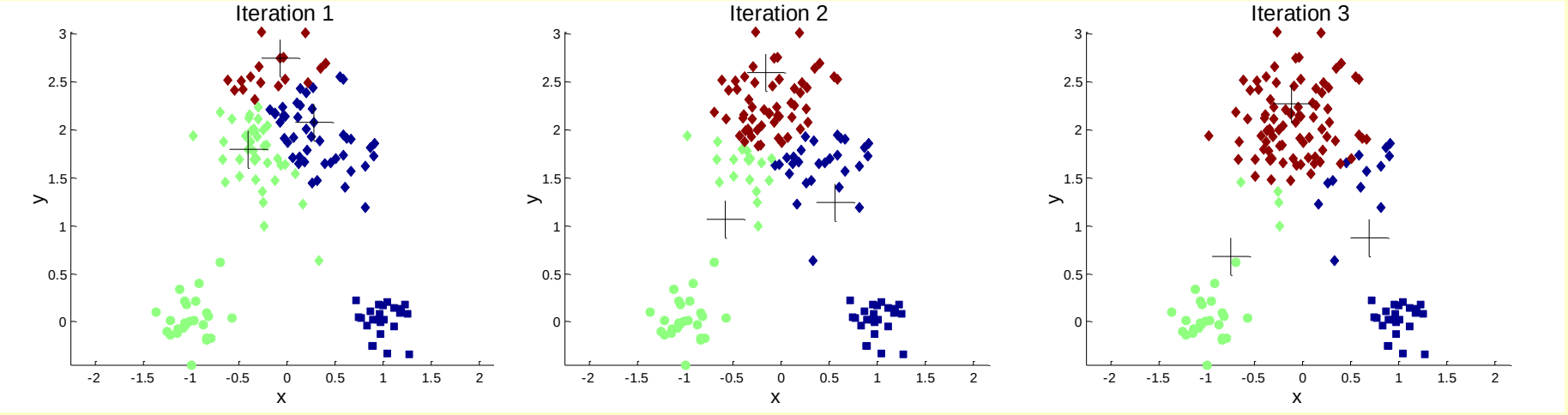


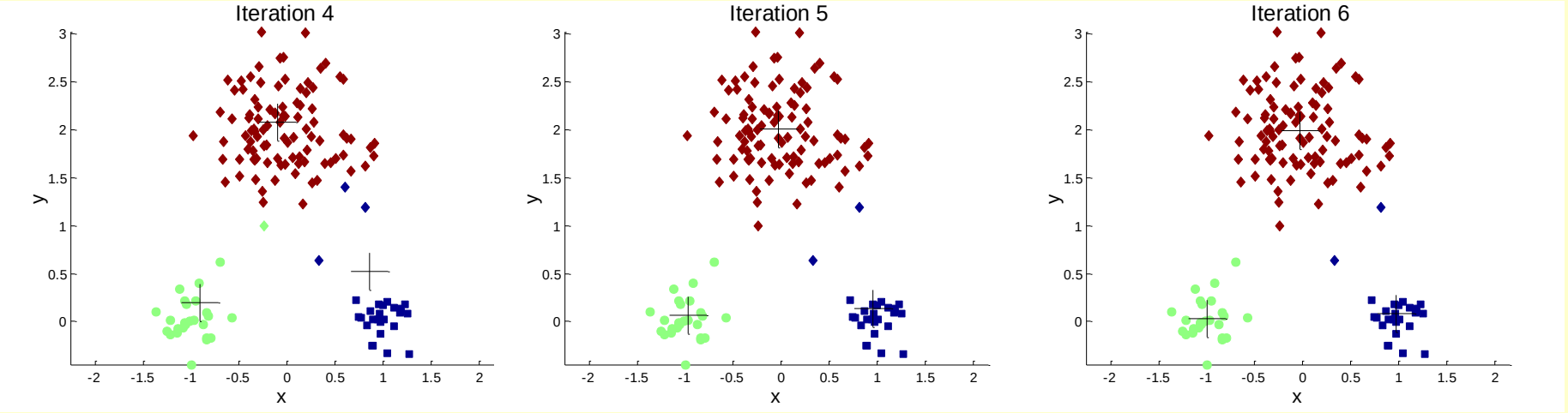
Sub-optimal Clustering

Importance of Choosing Initial Centroids

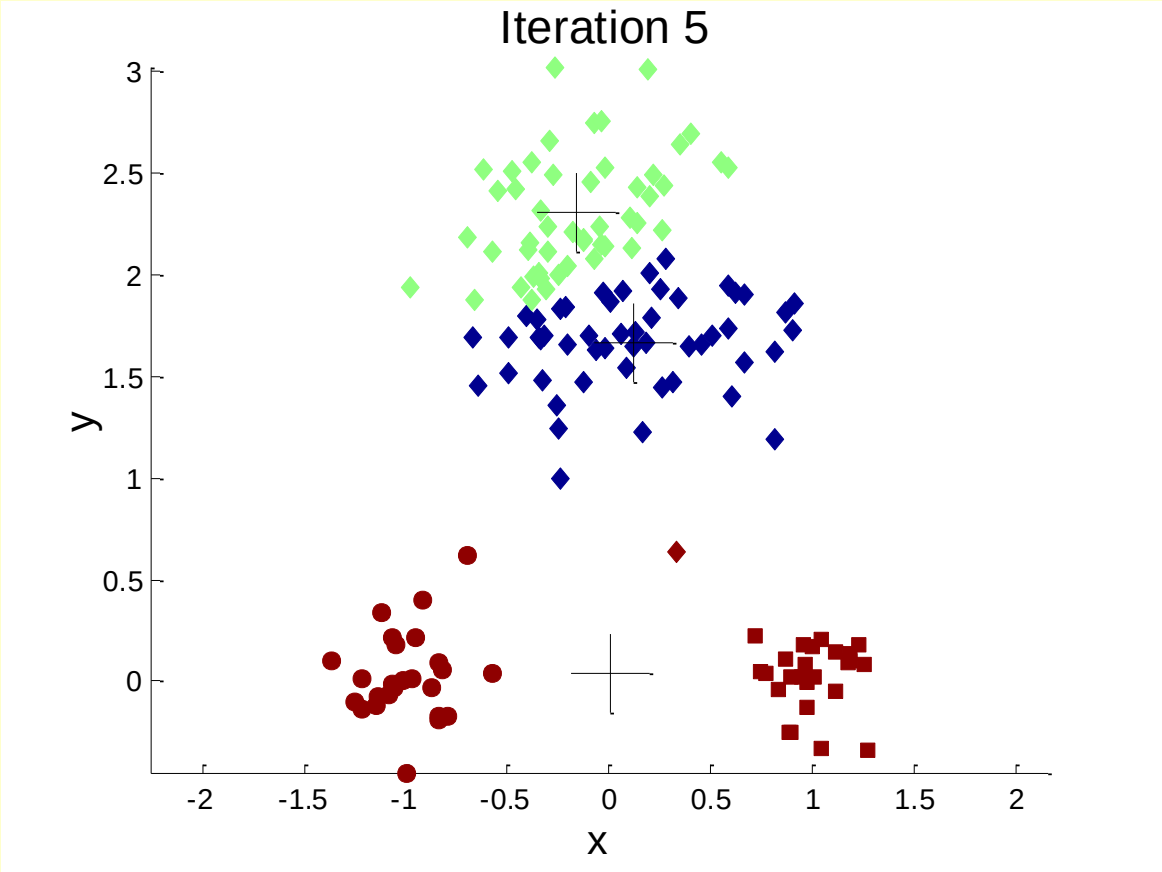


Importance of Choosing Initial Centroids

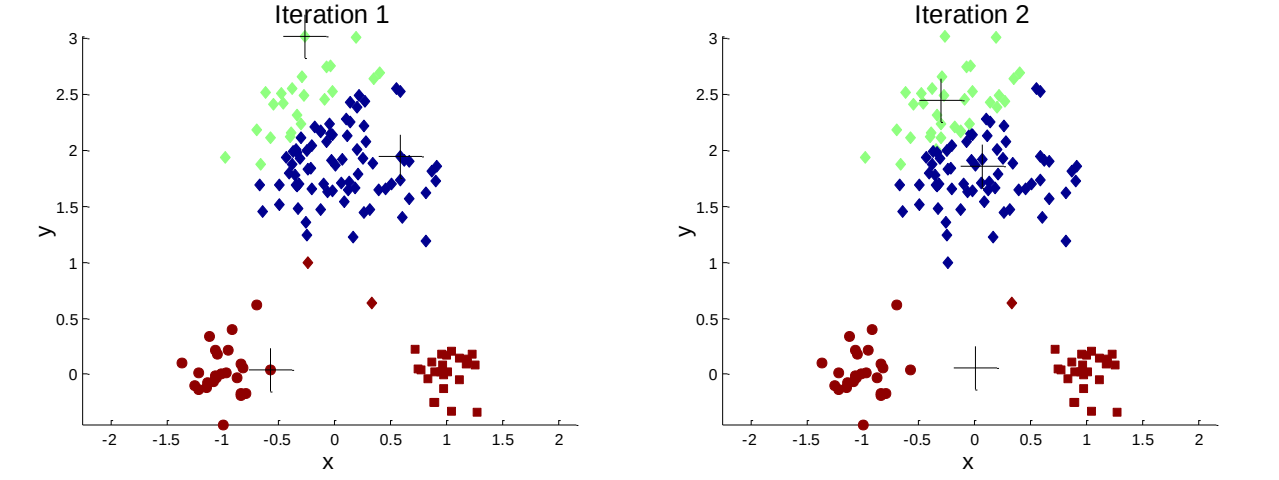


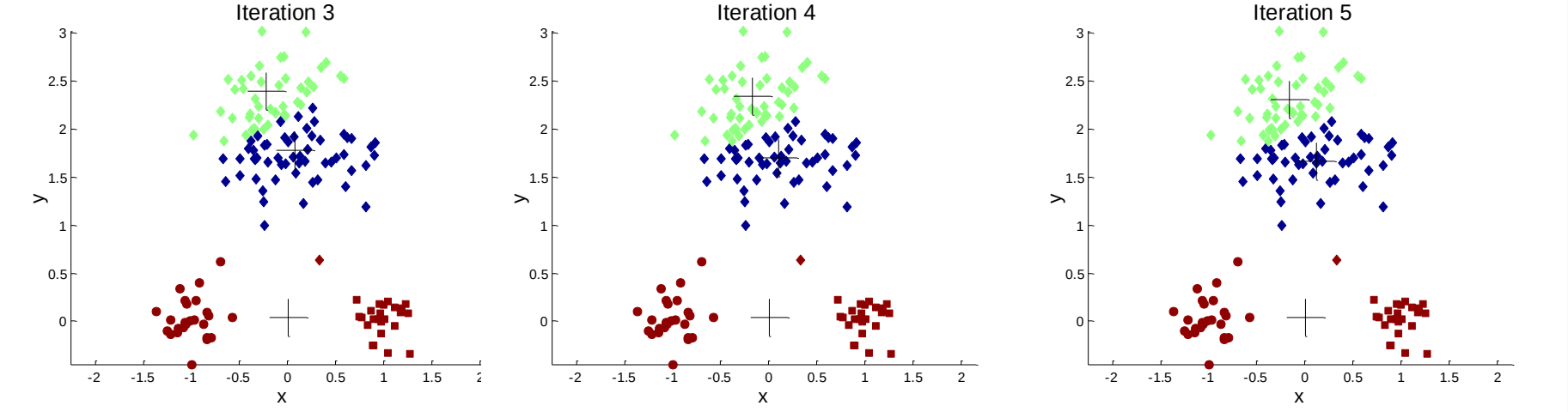


Importance of Choosing Initial Centroids …



Importance of Choosing Initial Centroids …





Problems with Selecting Initial Points

- If there are K 'real' clusters then the chance of selecting one centroid from each cluster is small.
  - Chance is relatively small when K is large
  - If clusters are the same size, n, then

$$P = \frac{\text{number of ways to select one centroid from each cluster}}{\text{number of ways to select } K \text{ centroids}} = \frac{K!n^K}{(Kn)^K} = \frac{K!}{K^K}$$

  - For example, if K = 10, then probability = $10!/10^{10} = 0.00036$
  - Sometimes the initial centroids will readjust themselves in 'right' way, and sometimes they don't

Solutions to Initial Centroids Problem

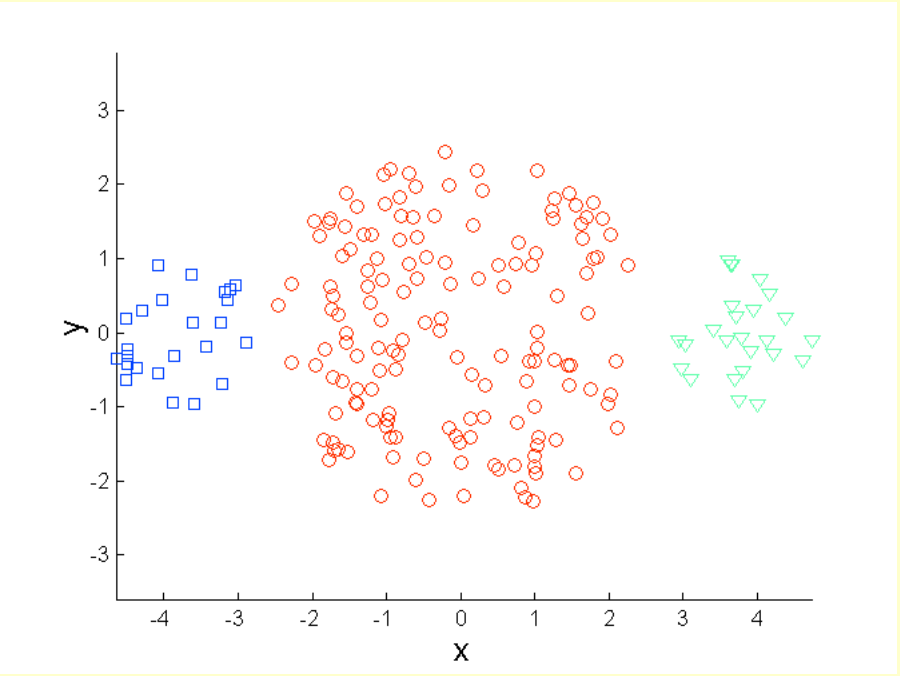
- Multiple runs
  - Helps, but probability is not on your side
- Sample and use hierarchical clustering to determine initial centroids
- Select more than k initial centroids and then select among these initial centroids
  - Select most widely separated
- Postprocessing
- Bisecting K-means
  - Not as susceptible to initialization issues

Limitations of K-means

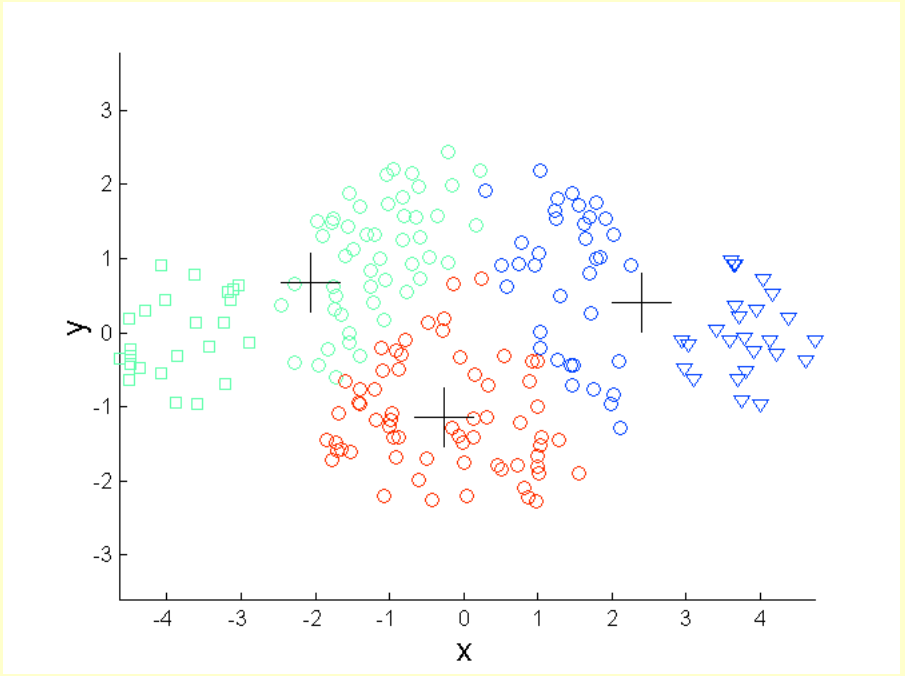
- K-means has problems when clusters are of differing
  - Sizes
  - Densities
  - Non-globular shapes

- K-means has problems when the data contains outliers.

Limitations of K-means: Differing Sizes

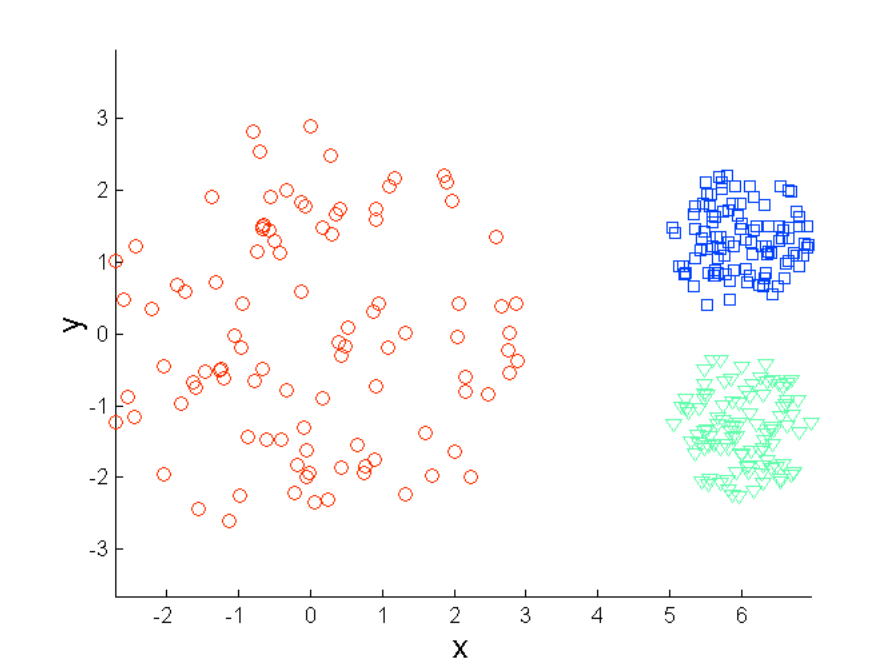


Original Points

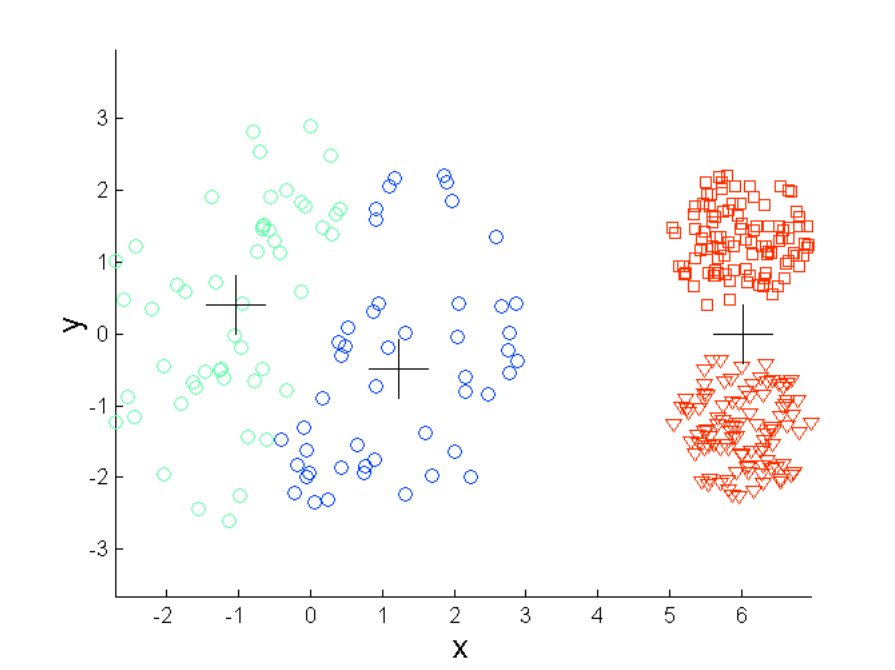


K-means (3 Clusters)

Limitations of K-means: Differing Density

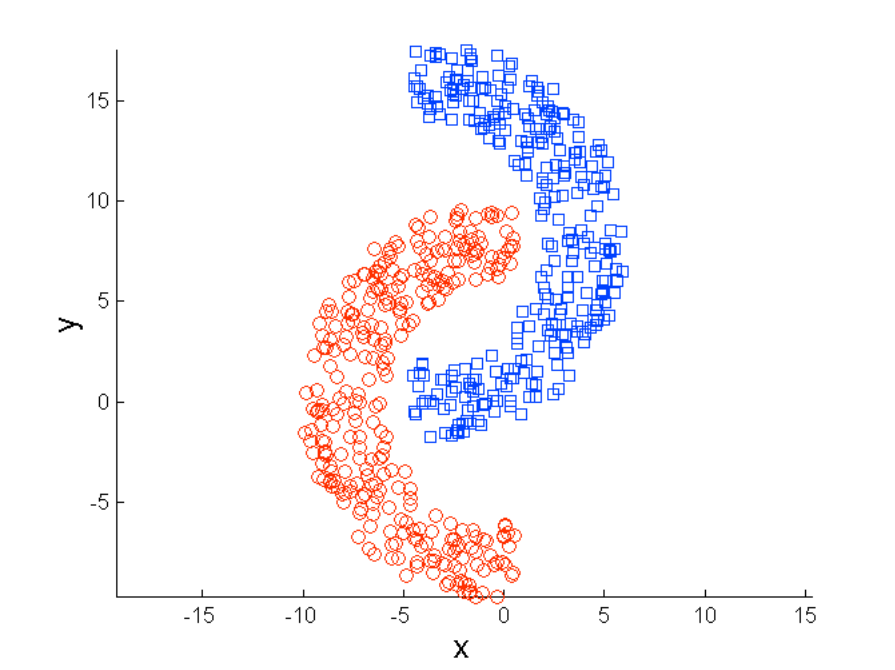


Original Points

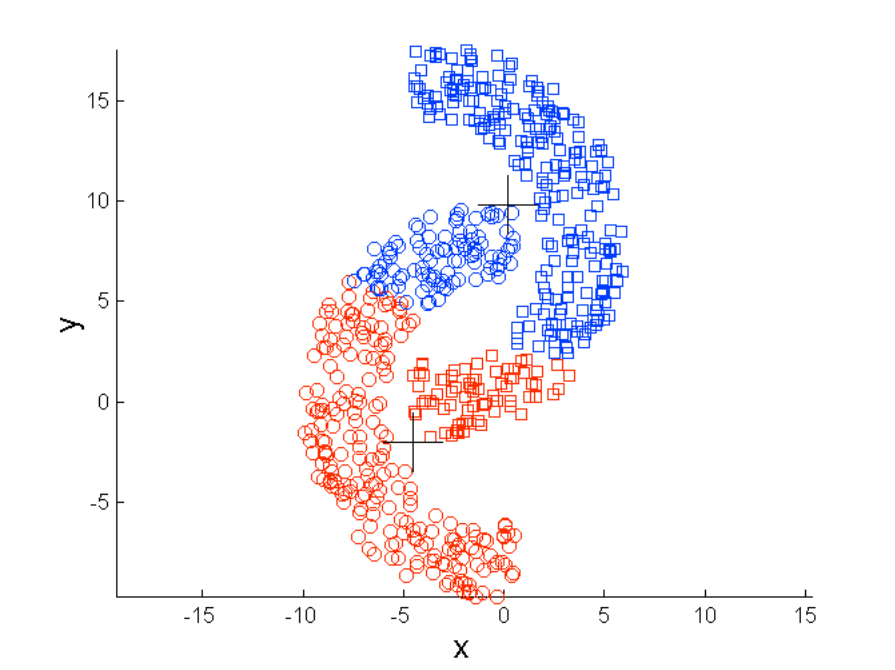


K-means (3 Clusters)

Limitations of K-means: Non-globular and linearly non-separable Shapes



Original Points



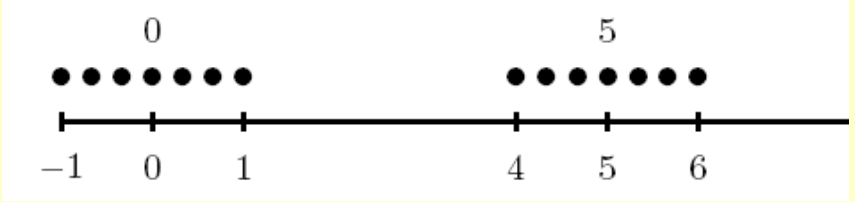
K-means (2 Clusters)

Landscape for Clustering Problem

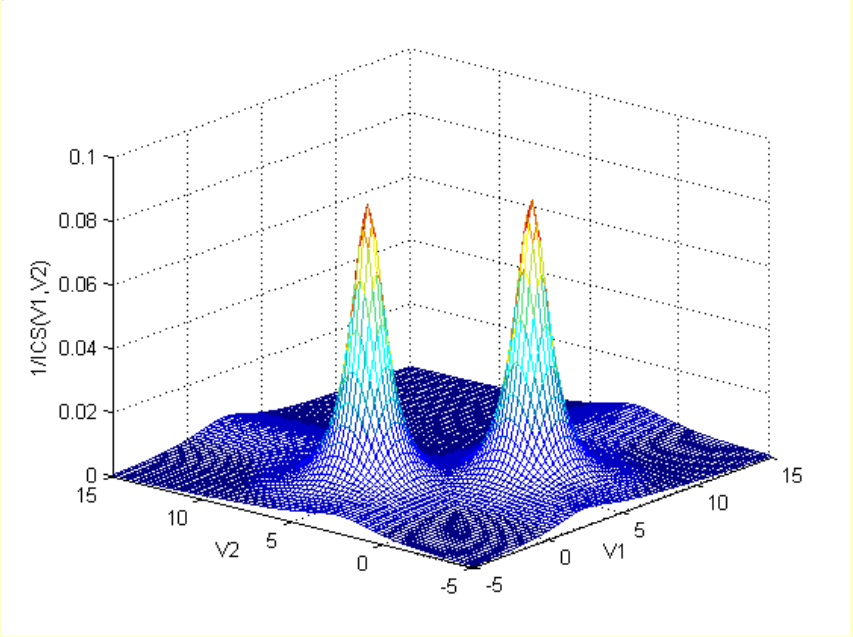
The k-means objective function

$$ICS(C_1, C_2, ..., C_k) = \sum_{j=1}^{k} \sum_{\vec{Z}_i \in C_i} d^2(\vec{Z}_i, \vec{V}_j)$$

Example of an extremely simple one-dimensional dataset

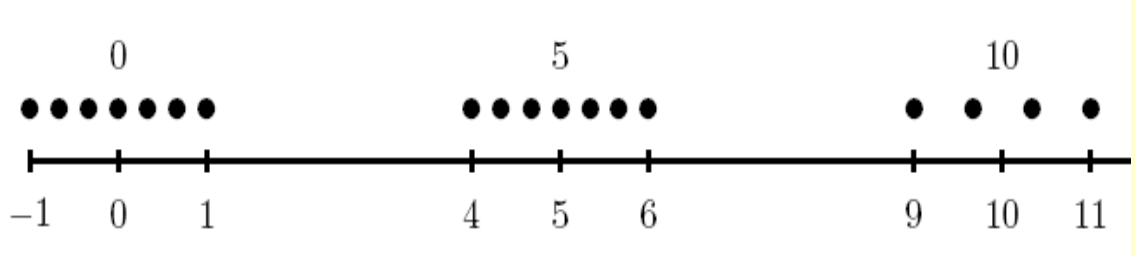


Fitness function (Reciprocal of ICS) plot of the hard c-means algorithm for above dataset

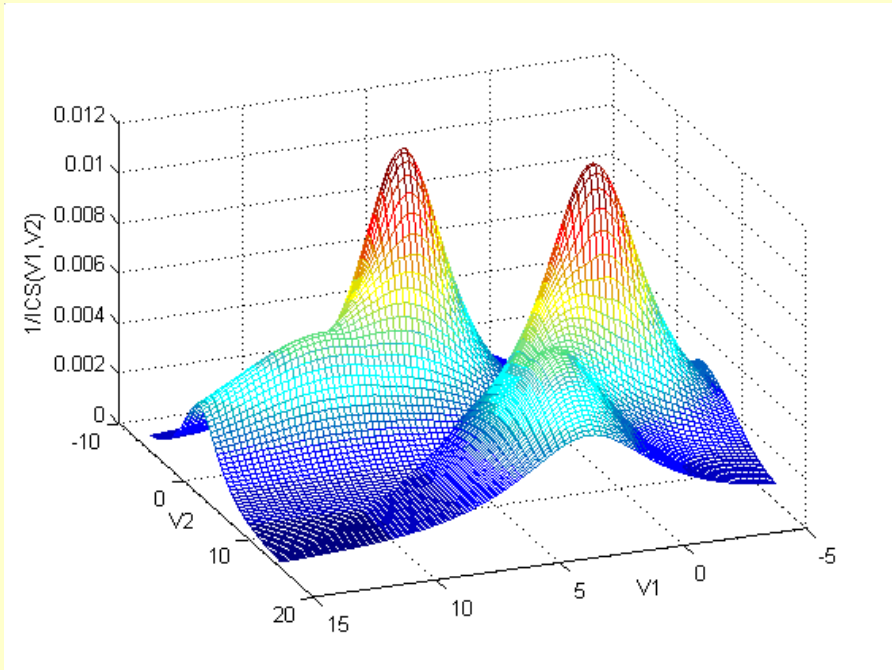


Landscape for Clustering Problem (Contd.)

The previous data but now
With some noise points



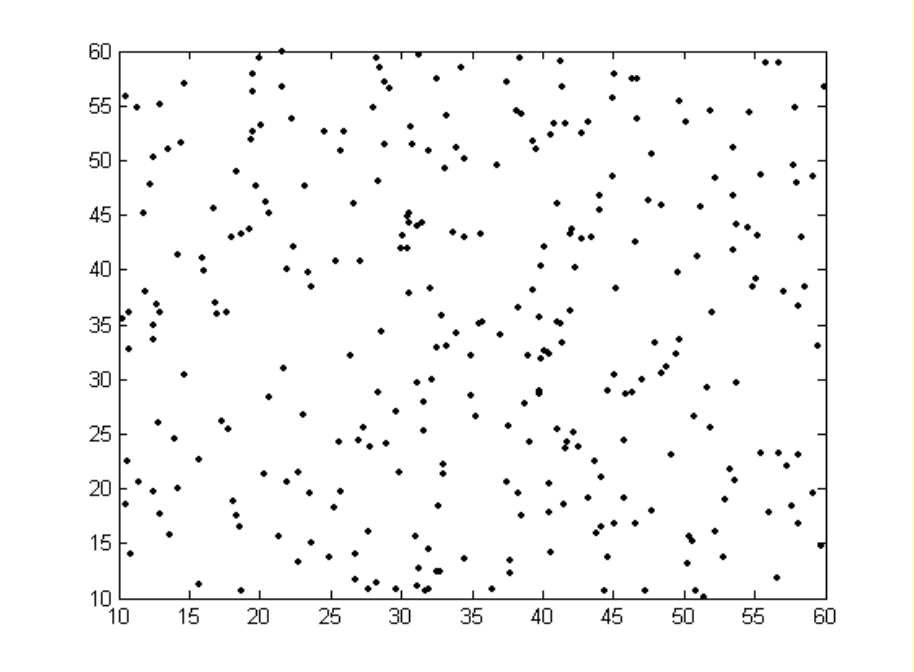
Fitness function (Reciprocal of ICS) plot of the hard c-means algorithm for above dataset



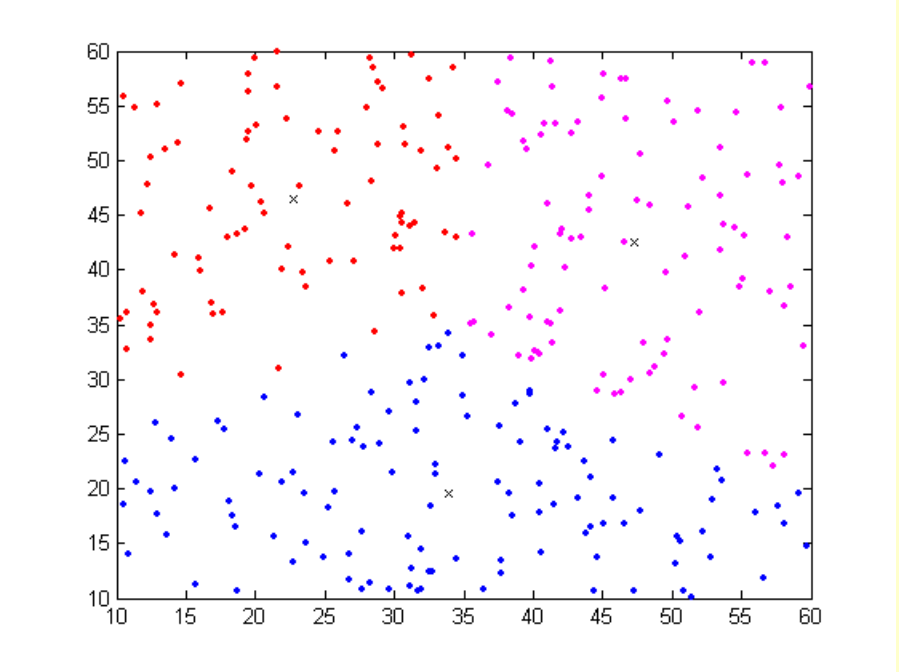
In addition to two global minima, we have local minima at: approximately (0, 10), (5, 10), (10, 0), (10, 5). For these local minima one prototype covers one of the data clusters, while the other prototype is mislead to the noise cluster.

A Few Open-ended Research Problems

- Detecting **Clustering Tendency** before doing any clustering operation: Hypothesis testing?



A random patch of data points
With no cluster



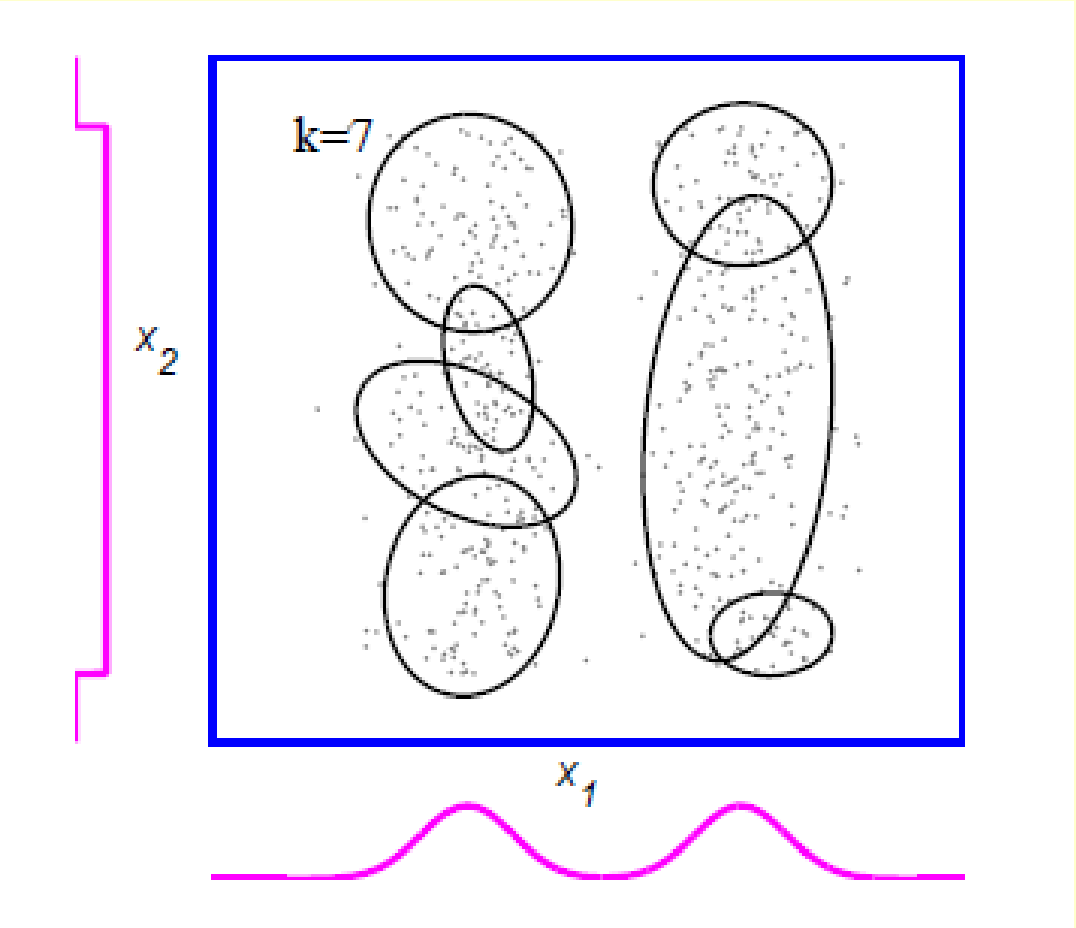
Clustering with K means when initialized
With c = 3

Simultaneous Feature Selection with Clustering

- Feature selection (i.e., attribute subset selection):
  - Eliminate noisy features
  - Discover the most important features
- Advantages
  - Reducing dimensionality
  - Improving learning efficiency
  - Increasing predictive accuracy
  - Reducing complexity of learned results

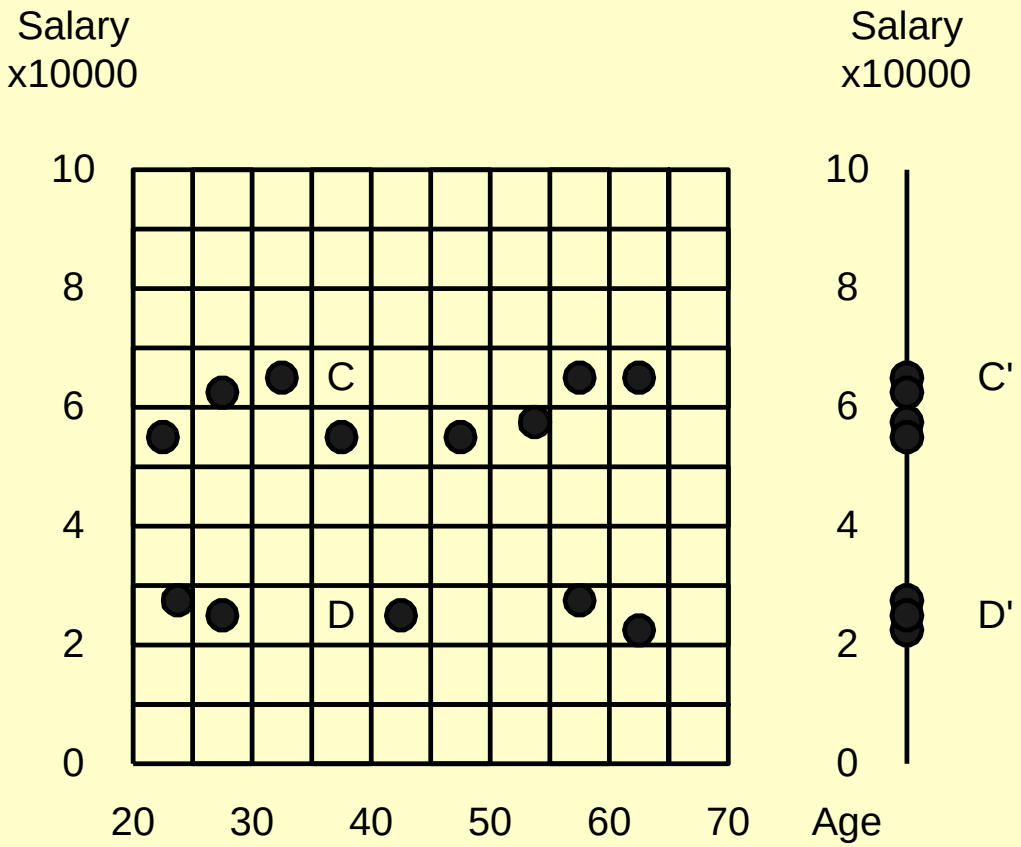
A must read for high-dimensional clustering:

One feature may be more discriminative than the other



Only feature x_1 can identify two clusters and not feature x_2

One feature may be more discriminative than the other (Contd.)



There is no cluster in the age subspace because there is no dense unit in that subspace. However there are two clusters in salary subspace

Open Problems

1) Can we use a proper representation of DE Vectors to make the clustering fully automatic?

2) Is it possible to select most important features of the dataset to reduce the computational overhead?

3) Can we choose a better way to evaluate the quality of the clusters using human user experience?

Sparse *k* Means Clustering

$$\underset{\Theta \in D, \mathbf{w}}{\text{maximize}} \left\{ \sum_{j=1}^{p} w_j f_j(\mathbf{X}_j, \Theta) \right\}$$

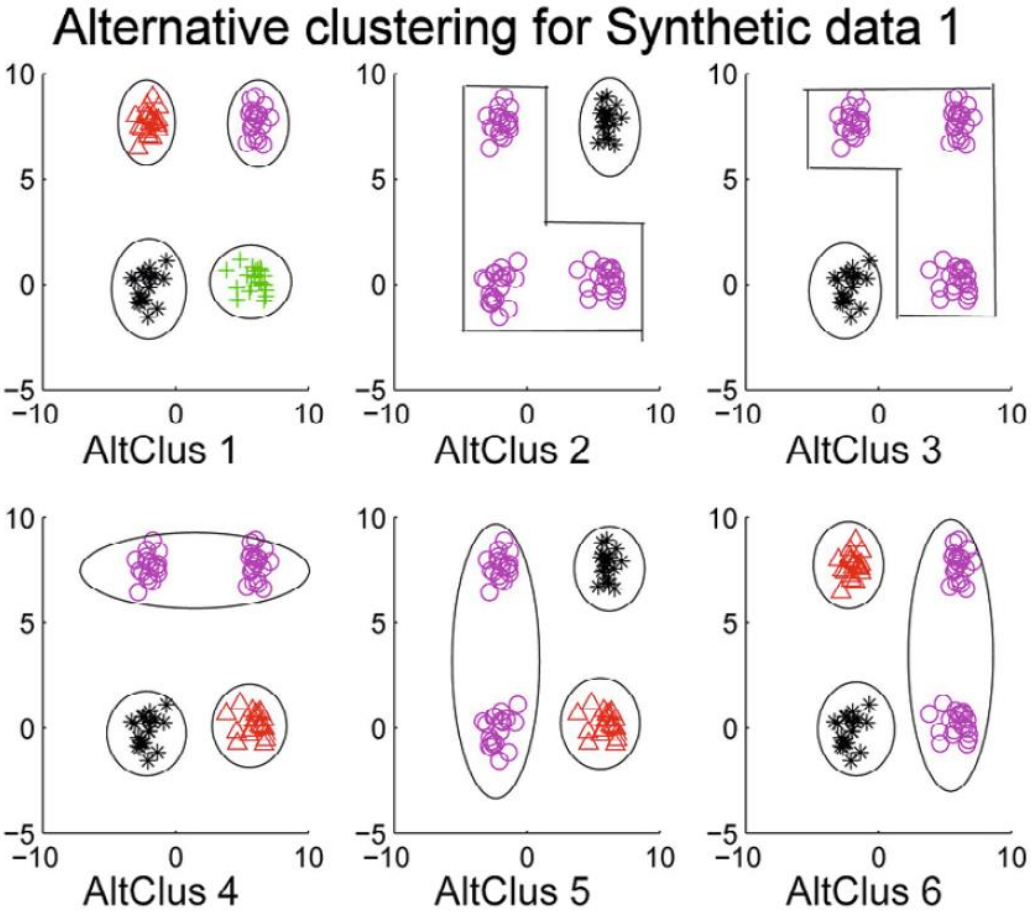
$$\text{subject to } ||\mathbf{w}||^2 \leq 1,\ ||\mathbf{w}||_1 \leq s,\ w_j \geq 0$$

Sparse Clustering: observations

- $w_1 = \cdots = w_p$ reduces to usual clustering
- L_1 penalty induces sparsity
- without L_2 penalty solution is trivial
- w_j is the contribution of feature j to the clustering
- in general, $f_j(\mathbf{X}_j, \Theta) > 0$ for some or all j

Daniela M. Witten and Robert Tibshirani, **A Framework for Feature Selection in Clustering,** *Journal of the American Statistical Association,* Vol. 105, No. 490 (June 2010), pp. 713-726

Towards Multi-context clustering – can we improve with evolutionary approaches?



Moumita Saha and Pabitra Mitra, VLGAAC: Variable Length Genetic Algorithm Based Alternative Clustering. ***ICONIP*** (2) 2014: 194-202

Towards Multi-context clustering – can we improve with evolutionary approaches?

Can we cluster a face database by using two contexts and present both partitions at one go?

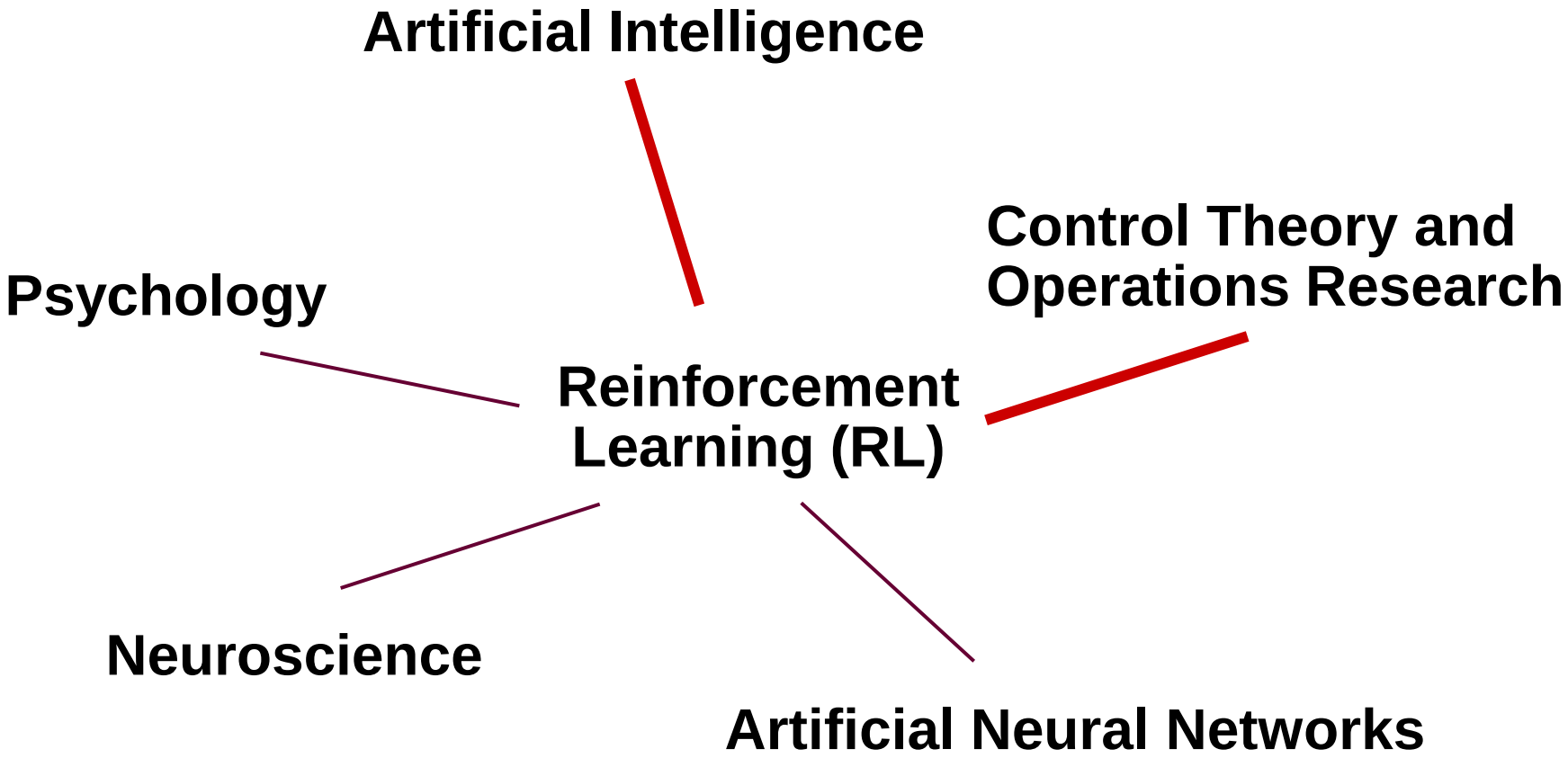
- Cluster the faces by who's face is that? – Tom, Dick, Harry….

- Cluster the faces as per the indicated emotion – sad, angry, happy…..

Reinforcement Learning

- Based on “Reinforcement Learning – An Introduction” by Richard Sutton and Andrew Barto

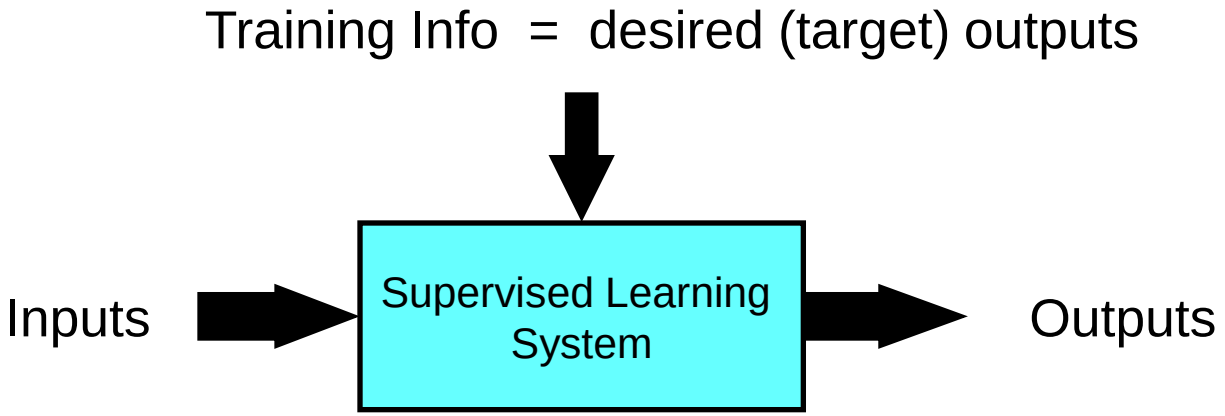
Learning from Experience Plays a Role in …



What is Reinforcement Learning?

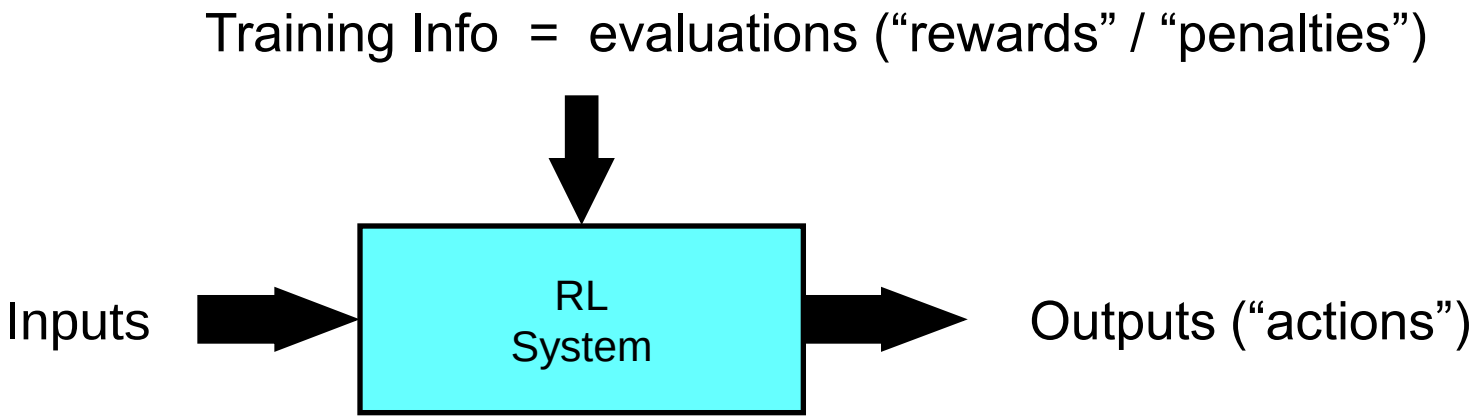
- Reinforcement learning is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.
- Learning from interaction
- Goal-oriented learning
- Learning about, from, and while interacting with an external environment
- Learning what to do—how to map situations to actions—so as to maximize a numerical reward signal

Supervised Learning



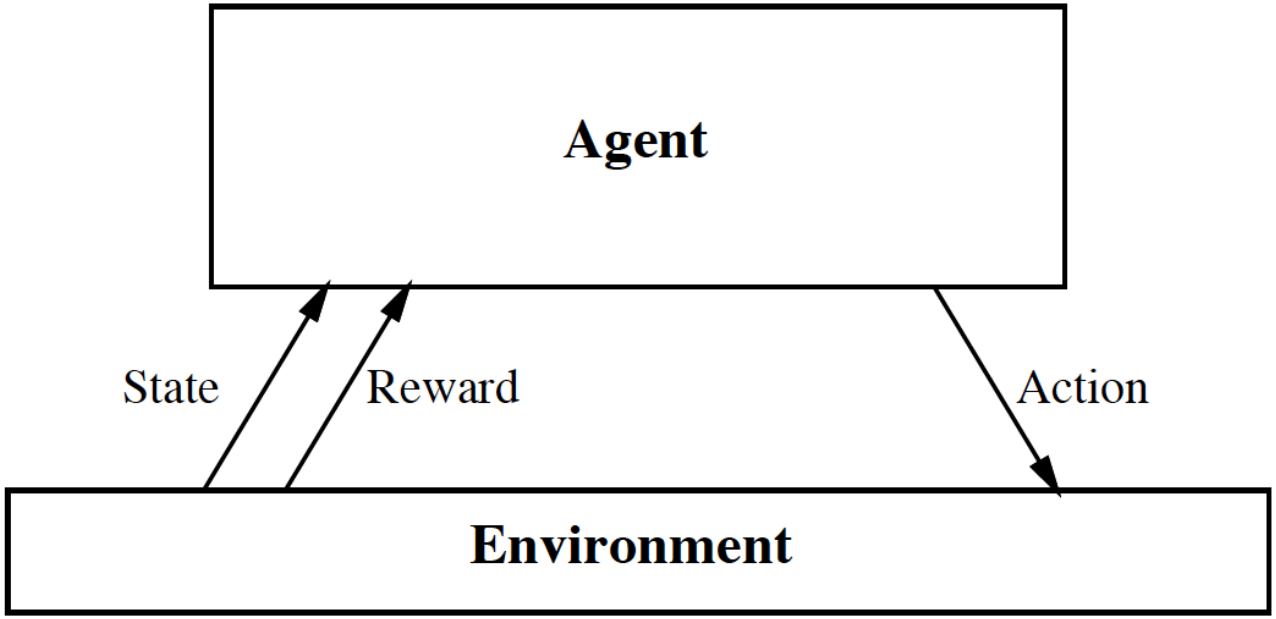
Error = (target output – actual output)

Reinforcement Learning



Objective: get as much reward as possible

The Big Picture



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

- Your action influences the state of the world which determines its reward

Key Features of RL

- In machine learning, the environment is typically formulated as a Markov decision process (MDP) as many reinforcement learning algorithms for this context utilize dynamic programming techniques.
- Learner is not told which actions to take
- Trial-and-Error search
- Possibility of delayed reward (sacrifice short-term gains for greater long-term gains)
- The need to ***explore*** and ***exploit***
- Considers the whole problem of a goal-directed agent interacting with an uncertain environment

The Markov Property

- By "the state" at step *t*, we mean whatever information is available to the agent at step *t* about its environment.
- The state can include immediate "sensations," highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations so as to retain all "essential" information, i.e., it should have the **Markov Property**:

$$Pr\{s_{t+1}=s', r_{t+1}=r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} =$$
$$Pr\{s_{t+1}=s', r_{t+1}=r \mid s_t, a_t\}$$

for all s', r, and histories s_t, a_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0.

Markov Decision Processes

- If a reinforcement learning task has the Markov Property, it is basically a **Markov Decision Process (MDP)**.
- If state and action sets are finite, it is a **finite MDP**.
- To define a finite MDP, you need to give:
  - **state and action sets**
  - one-step "dynamics" defined by **transition probabilities**:

$$P^a_{ss'} = \Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\} \quad \text{for all } s, s' \in S,\ a \in A(s).$$

  - **reward probabilities**:

$$R^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\} \quad \text{for all } s, s' \in S,\ a \in A(s).$$

An Example Finite MDP

Recycling Robot

- At each step, robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge.
- Searching is better but runs down the battery; if runs out of power while searching, has to be rescued (which is bad).
- Decisions made on basis of current energy level: `high`, `low`.
- Reward = number of cans collected

Recycling Robot MDP

$S = \{\text{high}, \text{low}\}$

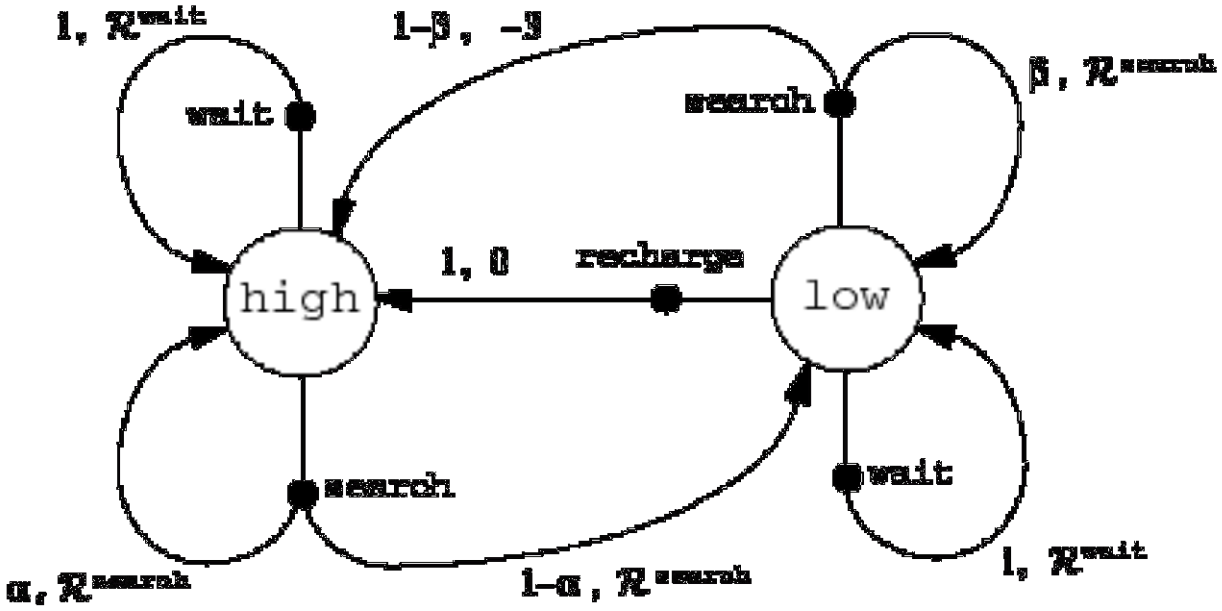
$A(\text{high}) = \{\text{search}, \text{wait}\}$

$A(\text{low}) = \{\text{search}, \text{wait}, \text{recharge}\}$

$R^{\text{search}} = \textit{expected no. of cans while searching}$

$R^{\text{wait}} = \textit{expected no. of cans while waiting}$

$R^{\text{search}} > R^{\text{wait}}$



Transition Table

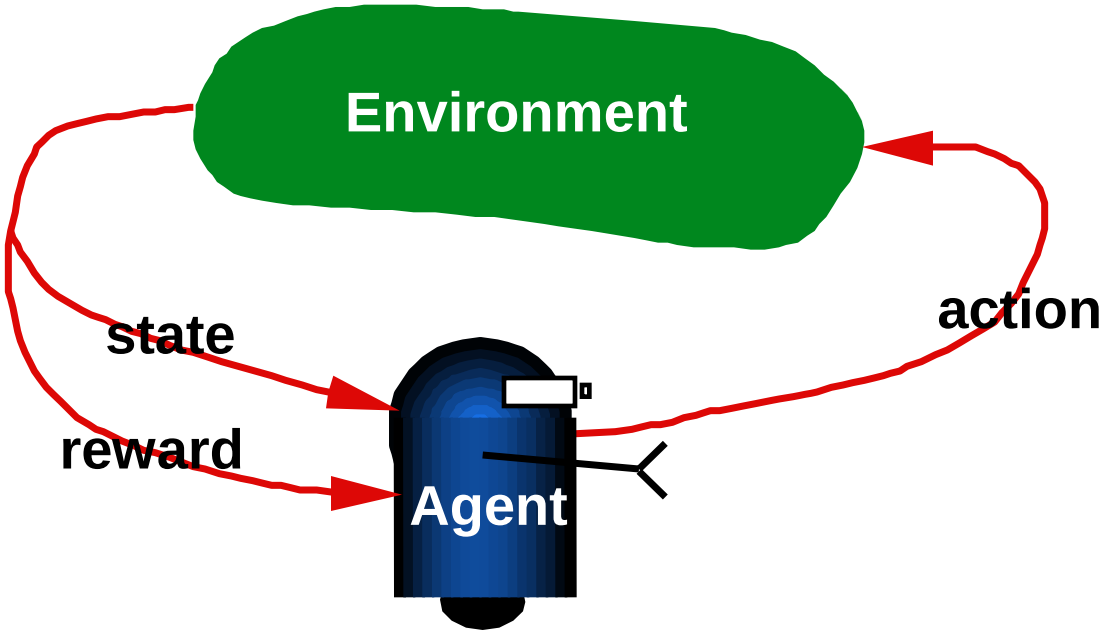
Table 3.1 Transition probabilities and expected rewards for the finite MDP of the recycling robot example.

s	s'	a	$\mathcal{P}^a_{ss'}$	$\mathcal{R}^a_{ss'}$
high	high	search	α	$\mathcal{R}^{\text{search}}$
high	low	search	$1-\alpha$	$\mathcal{R}^{\text{search}}$
low	high	search	$1-\beta$	-3
low	low	search	β	$\mathcal{R}^{\text{search}}$
high	high	wait	1	$\mathcal{R}^{\text{wait}}$
high	low	wait	0	$\mathcal{R}^{\text{wait}}$
low	high	wait	0	$\mathcal{R}^{\text{wait}}$
low	low	wait	1	$\mathcal{R}^{\text{wait}}$
low	high	recharge	1	0
low	low	recharge	0	0

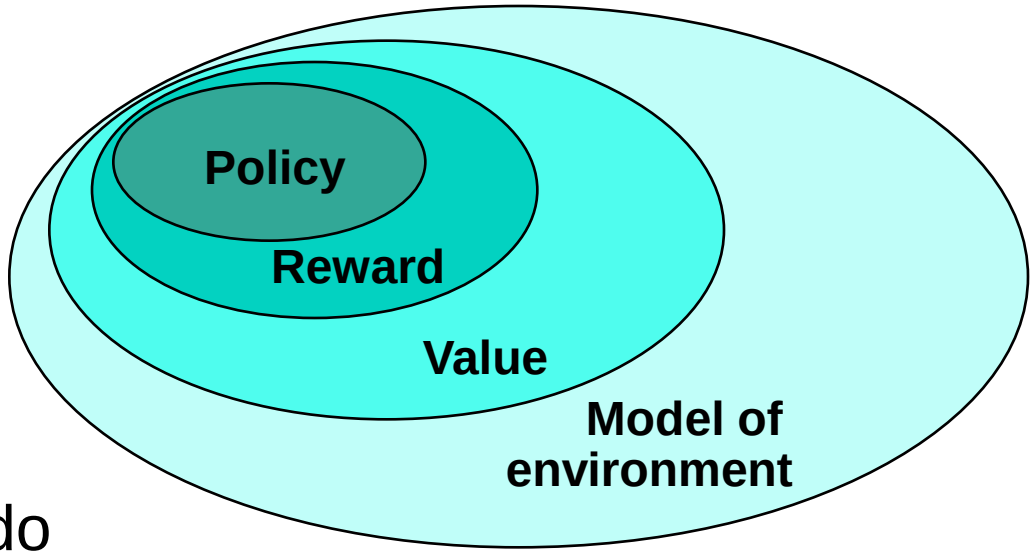
Note: There is a row for each possible combination of current state, s, next state, s', and action possible in the current state, $a \in \mathcal{A}(s)$.

Complete Agent

- Temporally situated
- Continual learning and planning
- Object is to ***affect*** the environment
- Environment is stochastic and uncertain



Elements of RL



- **Policy**: what to do
- **Reward**: what is good
- **Value**: what is good because it *predicts* reward
- **Model**: what follows what

Overview

- **Supervised Learning: Immediate feedback (labels provided for every input).**

- **Unsupervised Learning: No feedback (no labels provided).**

- **Reinforcement Learning: Delayed scalar feedback (a number called reward).**

- **RL deals with agents that must sense & act upon their environment.**
  - **This is combines classical AI and machine learning techniques.**
  - **It the most comprehensive problem setting.**

- **Examples:**
  - **A robot cleaning my room and recharging its battery**
  - **Robot-soccer**
  - **How to invest in shares**
  - **Modeling the economy through rational agents**
  - **Learning how to fly a helicopter**
  - **Scheduling planes to their destinations**
  - **and so on**

Complications

- **The outcome of your actions may be uncertain**
- **You may not be able to perfectly sense the state of the world**
- **The reward may be stochastic.**
- **Reward is delayed (i.e. finding food in a maze)**
- **You may have no clue (model) about how the world responds to your actions.**
- **You may have no clue (model) of how rewards are being paid off.**
- **The world may change while you try to learn it**
- **How much time do you need to explore uncharted territory before you**
  - **exploit what you have learned?**

The Task

- To learn an optimal *policy* that maps states of the world to actions of the agent.
  I.e., if this patch of room is dirty, I clean it. If my battery is empty, I recharge it.

$$\pi : S \rightarrow A$$

- What is it that the agent tries to optimize?
  Answer: the total future discounted reward:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad 0 \leq \gamma < 1$$

Note: immediate reward is worth more than future reward.

What would happen to mouse in a maze with gamma = 0 ?

Value Function

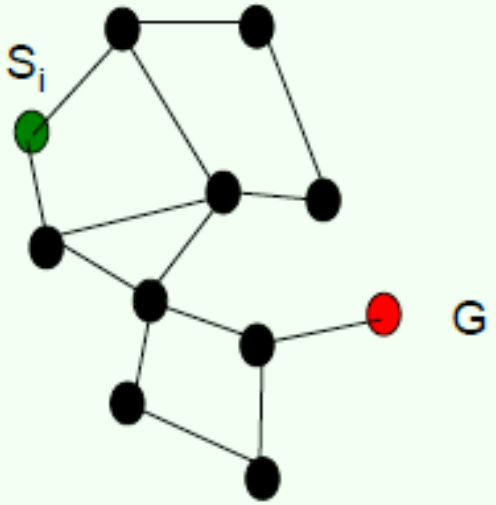
- Let's say we have access to the optimal value function that computes the total future discounted reward $V^*(s)$
- What would be the optimal policy $\pi^*(s)$?
- Answer: we choose the action that maximizes:

$$p^*(s) = \arg\max_a \left[r(s,a) + \gamma V^*(\delta(s,a)) \right]$$

- We assume that we know what the reward will be if we perform action "a" in state "s": $r(s,a)$
- We also assume we know what the next state of the world will be if we perform action "a" in state "s": $s_{t+1} = \delta(s_t, a)$

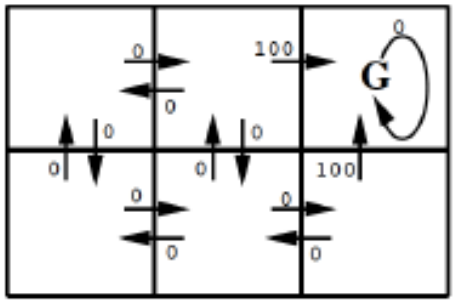
Example I

- Consider some complicated graph, and we would like to find the shortest path from a node Si to a goal node G.
- Traversing an edge will cost you "length edge" dollars.
- The value function encodes the total remaining distance to the goal node from any node s, i.e. V(s) = "*1 / distance*" to goal from s.
- If you know V(s), the problem is trivial. You simply choose the node that has highest V(s).

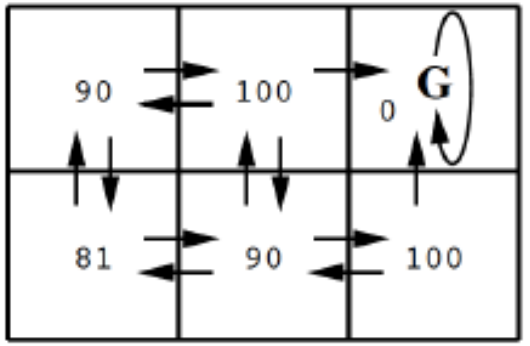


Example II

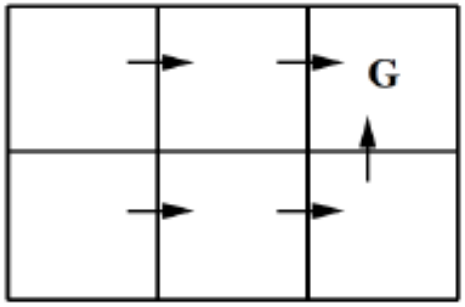
Find your way to the goal.



$r(s, a)$ (immediate reward) values



$V^*(s)$ values



One optimal policy

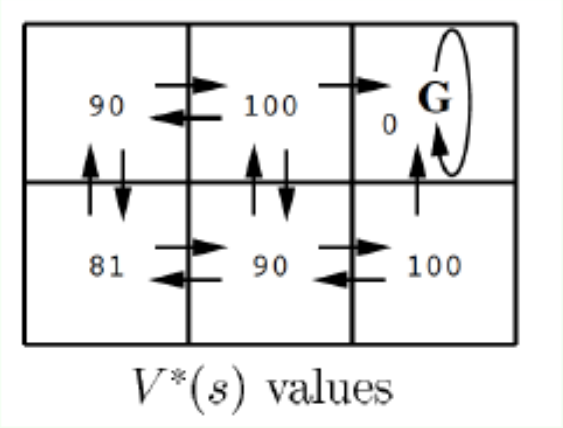
Q-Function

- One approach to RL is then to try to estimate V*(s). $V^*(s) \neg \max_a \left[r(s,a) + g V^*(d(s,a)) \right]$
- However, this approach requires you to know r(s,a) and delta(s,a).
- This is unrealistic in many real problems. What is the reward if a robot is exploring mars and decides to take a right turn?
- Fortunately we can circumvent this problem by exploring and experiencing how the world reacts to our actions. We need to *learn* r & delta.
- We want a function that directly learns good state-action pairs, i.e. what action should I take in this state. We call this Q(s,a).
- Given Q(s,a) it is now trivial to execute the optimal policy, *without knowing r(s,a) and delta(s,a).* We have:

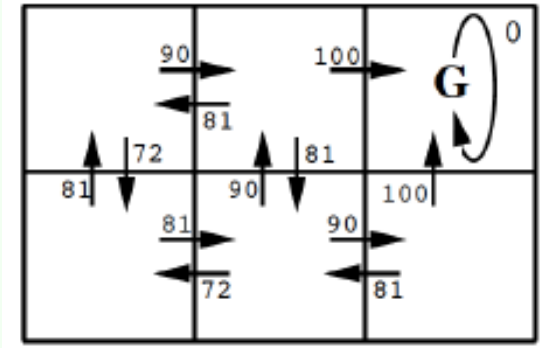
$$\pi^*(s) = \arg\max_a Q(s,a)$$

$$V^*(s) = \max_a Q(s,a)$$

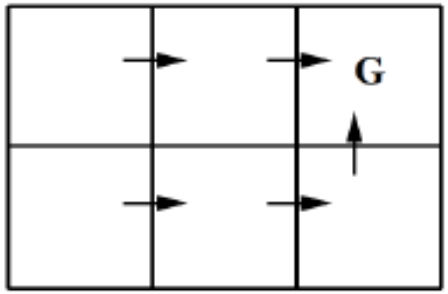
Example II



$V^*(s)$ values



$Q(s,a)$ values



One optimal policy

Check that

$$\pi^*(s) = \arg\max_a Q(s,a)$$

$$V^*(s) = \max_a Q(s,a)$$

Q-Learning

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$
$$= r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

- This still depends on r(s,a) and delta(s,a).
- However, imagine the robot is exploring its environment, trying new actions as it goes.
- At every step it receives some reward "r", and it observes the environment change into a new state s' for action a.
  How can we use these observations, (s,a,s',r) to learn a model?

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s' = s_{t+1}$

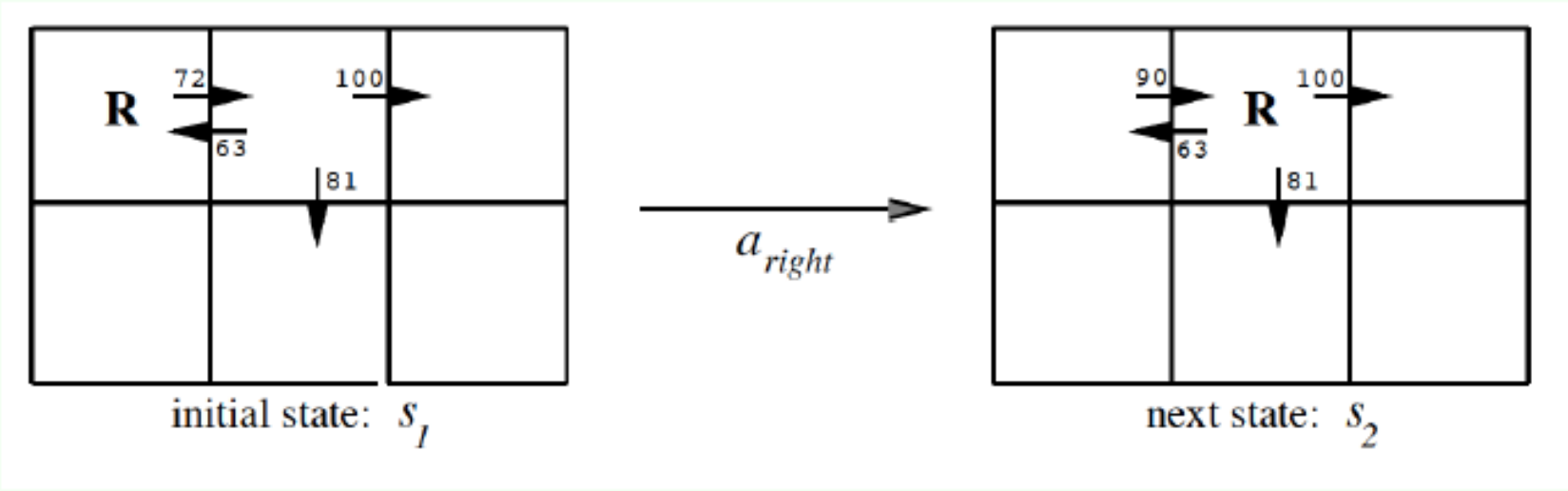
Q-Learning

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s' = s_{t+1}$

- This equation continually estimates Q at state s consistent with an estimate of Q at state s', one step in the future: temporal difference (TD) learning.
- Note that s' is closer to goal, and hence more "reliable", but still an estimate itself.
- Updating estimates based on other estimates is called bootstrapping.
- We do an update after each state-action pair. I.e., we are learning online!
- We are learning useful things about explored state-action pairs. These are typically most useful because they are likely to be encountered again.
- Under suitable conditions, these updates can actually be proved to converge to the real answer.

Example Q-Learning



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

$$\leftarrow 0 + 0.9 \max\{66, 81, 100\}$$

$$\leftarrow 90$$

Q-learning propagates Q-estimates 1-step backwards

Exploration / Exploitation

- It is very important that the agent does not simply follow the current policy when learning Q. (off-policy learning).The reason is that you may get stuck in a suboptimal solution. I.e. there may be other solutions out there that you have never seen.

- Hence it is good to try new things so now and then, e.g. $P(a \mid s) \propto e^{\hat{Q}(s,a)/T}$
  If T large lots of exploring, if T small follow current policy.
  One can decrease T over time.

Improvements

- One can trade-off memory and computation by cashing (s,s' ,r) for observed transitions. After a while, as Q(s' ,a') has changed, you can "replay" the update:

$$\hat{Q}(s,a) \neg \; r + g \max_{a'} \hat{Q}(s',a')$$

- One can actively search for state-action pairs for which Q(s,a) is expected to change a lot (prioritized sweeping).

- One can do updates along the sampled path much further back than just one step ($TD(\lambda)$ learning).

Extensions

- To deal with stochastic environments, we need to maximize *expected* future discounted reward:

$$Q(s,a) = E[r(s,a)] + g \sum_{s'} P(s' \mid s,a) \max_{a'} Q(s',a')$$

- Often the state space is too large to deal with all states. In this case we need to learn a function: $Q(s,a) \approx f_\theta(s,a)$

- Neural network with back-propagation have been quite successful.

- For instance, TD-Gammon is a back-gammon program that plays at expert level. state-space very large, trained by playing against itself, uses NN to approximate value function, uses TD(lambda) for learning.

More on Function Approximation

- For instance: linear function:

$$Q(s,a) \approx f_q(s,a) = \sum_{k=1}^{K} q_k^a \Phi_k(s)$$

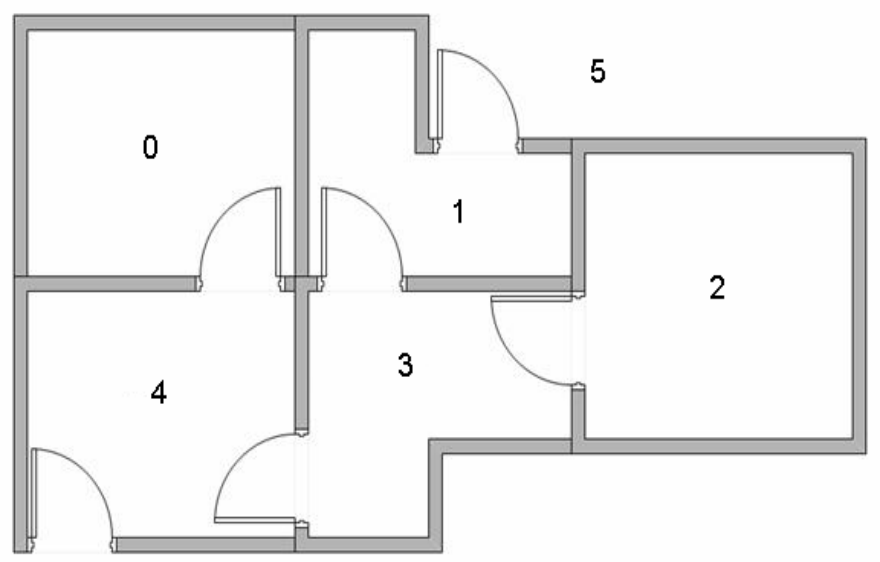
- The features Phi are fixed measurements of the state (e.g. # stones on the board).
- We only learn the parameters theta.
- Update rule: (start in state s, take action a, observe reward r and end up in state s')

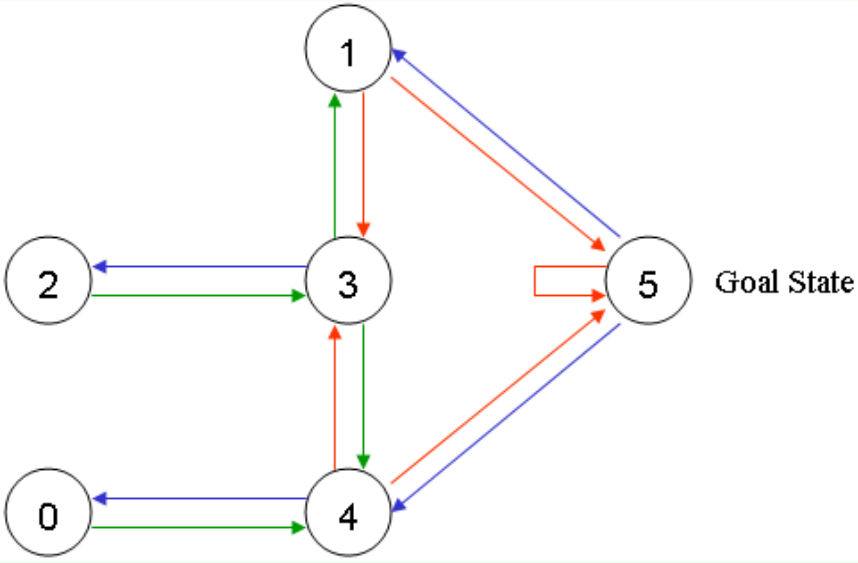
$$q_k^a \leftarrow q_k^a + \alpha \left(r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a) \right) \Phi_k(s)$$

- change in Q

Q Learning Example

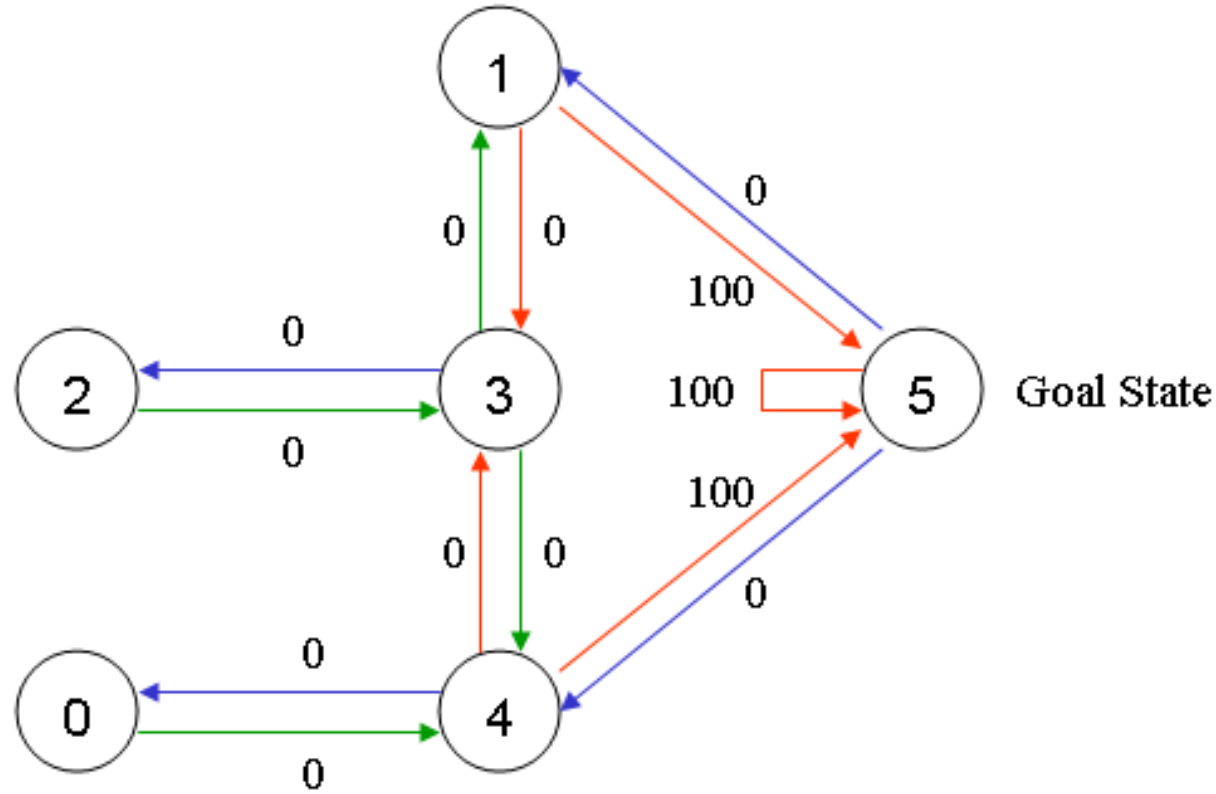
Suppose we have 5 rooms in a building connected by doors as shown in the figure below. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside).



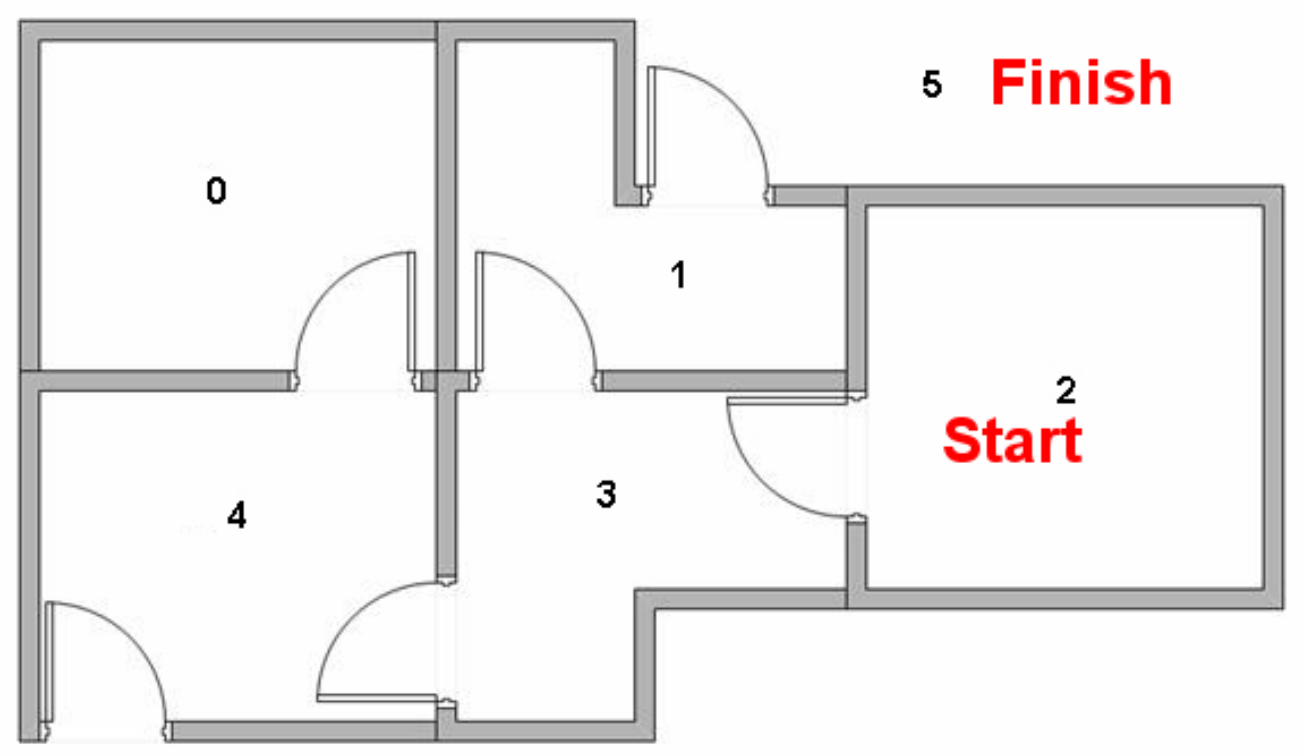


- We can represent the rooms on a graph, each room as a node, and each door as a link.

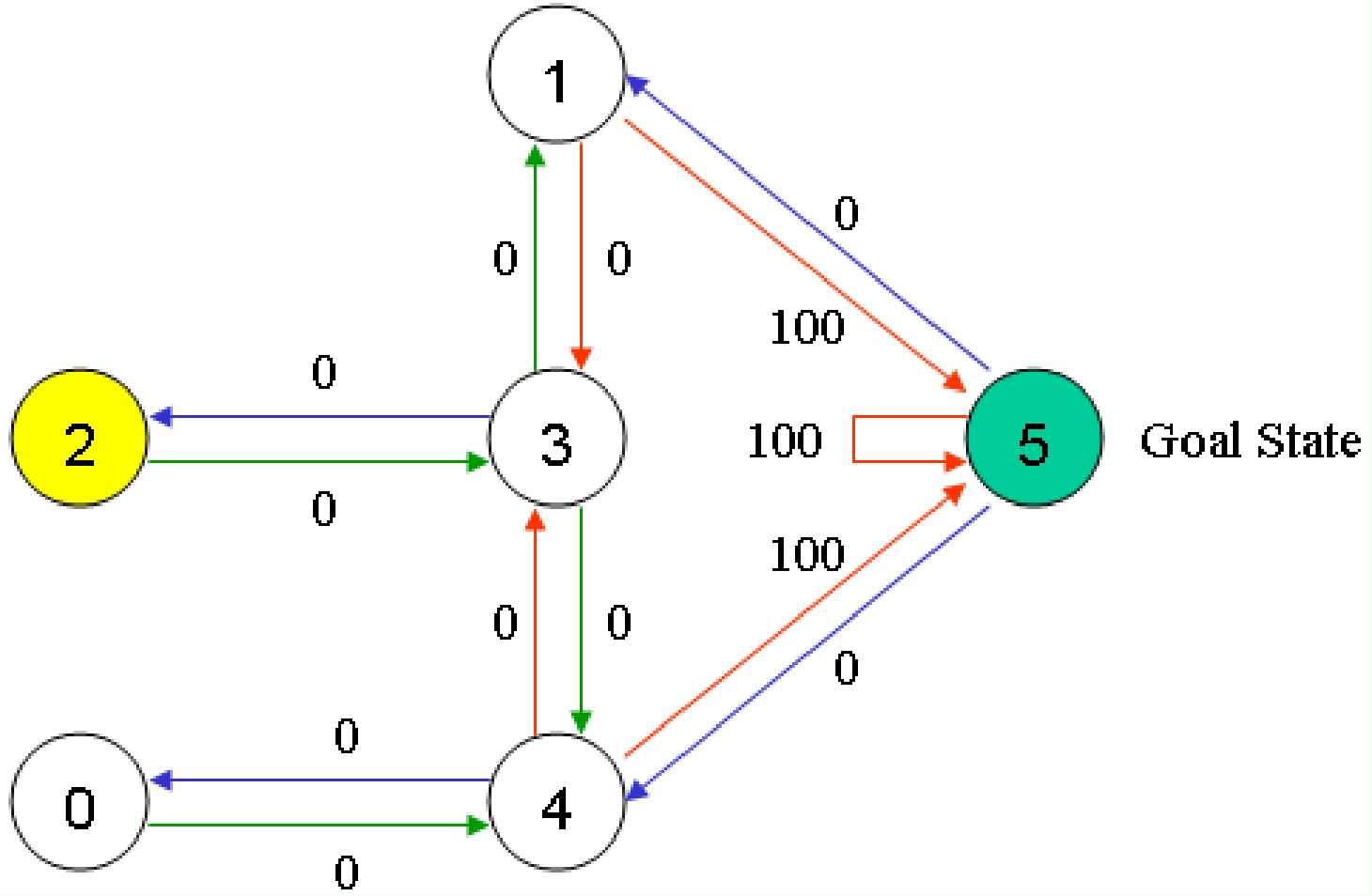
- For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5.
- To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The doors that lead immediately to the goal have an instant reward of 100.
- Other doors not directly connected to the target room have zero reward.
- Because doors are two-way (0 leads to 4, and 4 leads back to 0), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown below:



- Of course, Room 5 loops back to itself with a reward of 100, and all other direct connections to the goal room carry a reward of 100. In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".
- Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.
- Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).



- The terminology in Q-Learning includes the terms "state" and "action".
- We'll call each room, including outside, a "state", and the agent's movement from one room to another will be an "action". In our diagram, a "state" is depicted as a node, while "action" is represented by the arrows.



▪**Suppose the agent is in state 2. From state 2, it can go to state 3 because state 2 is connected to 3. From state 2, however, the agent cannot directly go to state 1 because there is no direct door connecting room 1 and 2 (thus, no arrows). From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3). If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3. If the agent is in state 1, it can go either to state 5 or 3. From state 0, it can only go back to state 4.**

▪**We can put the state diagram and the instant reward values into the following reward table, "matrix R".**

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

▪The -1's in the table represent null values (i.e.; where there isn't a link between nodes). For example, State 0 cannot go to State 1.

- Now we'll add a similar matrix, "Q", to the brain of our agent, representing the memory of what the agent has learned through experience. The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes).
- The agent starts out knowing nothing, the matrix Q is initialized to zero. In this example, for the simplicity of explanation, we assume the number of states is known (to be six). If we didn't know how many states were involved, the matrix Q could start out with only one element. It is a simple task to add more columns and rows in matrix Q if a new state is found.
- The transition rule of Q learning is a very simple formula:

$$Q(\text{state}, \text{action}) = R(\text{state}, \text{action}) + \text{Gamma} * \text{Max}[Q(\text{next state}, \text{all actions})]$$

According to this formula, a value assigned to a specific element of matrix Q, is equal to the sum of the corresponding value in matrix R and the learning parameter Gamma, multiplied by the maximum value of Q for all possible actions in the next state.

- **Our virtual agent will learn through experience, without a teacher (this is called unsupervised learning). The agent will explore from state to state until it reaches the goal. We'll call each exploration an episode. Each episode consists of the agent moving from the initial state to the goal state. Each time the agent arrives at the goal state, the program goes to the next episode.**

- **Each episode is equivalent to one training session. In each training session, the agent explores the environment (represented by matrix R), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix Q. More training results in a more optimized matrix Q. In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.**

- **The Gamma parameter has a range of 0 to 1 (0 <= Gamma > 1). If Gamma is closer to zero, the agent will tend to consider only immediate rewards. If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.**

- To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state. The algorithm finds the actions with the highest reward values recorded in matrix Q for current state.

- The Q-Learning algorithm goes as follows:

1. Set the gamma parameter, and environment rewards in matrix R.

2. Initialize matrix Q to zero.

3. For each episode:

   - Select a random initial state.
   - Do While the goal state hasn't been reached.

     - Select one among all possible actions for the current state.
     - Using this possible action, consider going to the next state.
     - Get maximum Q value for this next state based on all possible actions.
     - Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
     - Set the next state as the current state.
     - End Do

End For

- To understand how the Q-learning algorithm works, we'll go through a few episodes step by step.
- We'll start by setting the value of the learning parameter Gamma = 0.8, and the initial state as Room 1.
- Initialize matrix Q as a zero matrix:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

- **Look at the second row (state 1) of matrix R. There are two possible actions for the current state 1: go to state 3, or go to state 5. By random selection, we select to go to 5 as our action.**

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

- Now let's imagine what would happen if our agent were in state 5. Look at the sixth row of the reward matrix R (i.e. state 5). It has 3 possible actions: go to state 1, 4 or 5.

- **Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]**
- **Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100**

- **Since matrix Q is still initialized to zero, Q(5, 1), Q(5, 4), Q(5, 5), are all zero. The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1).**
- **The next state, 5, now becomes the current state. Because 5 is the goal state, we've finished one episode. Our agent's brain now contains an updated matrix Q as:**

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

This entry will be 100

This entry will be 0

- **For the next episode, we start with a randomly chosen initial state. This time, we have state 3 as our initial state.**
- **Look at the fourth row of matrix R; it has 3 possible actions: go to state 1, 2 or 4. By random selection, we select to go to state 1 as our action.**
- **Now we imagine that we are in state 1. Look at the second row of reward matrix R (i.e. state 1). It has 2 possible actions: go to state 3 or state 5. Then, we compute the Q value:**

- Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]

- Q(3, 1) = R(3, 1) + 0.8 * Max[Q(1, 3), Q(1, 5)] = 0 + 0.8 * Max(0, 100) = 80

- **We use the updated matrix Q from the last episode. Q(1, 3) = 0 and Q(1, 5) = 100. The result of the computation is Q(3, 1) = 80 because the reward is zero. The matrix Q becomes:**

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

- The next state, 1, now becomes the current state. We repeat the inner loop of the Q learning algorithm because state 1 is not the goal state

- So, starting the new loop with the current state 1, there are two possible actions: go to state 3, or go to state 5. By lucky draw, our action selected is 5.

- **Now, imaging we're in state 5, there are three possible actions: go to state 1, 4 or 5. We compute the Q value using the maximum value of these possible actions.**

Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]

Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100

- The updated entries of matrix Q, Q(5, 1), Q(5, 4), Q(5, 5), are all zero. The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1). This result does not change the Q matrix.

- Because 5 is the goal state, we finish this episode. Our agent's brain now contain updated matrix Q as:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

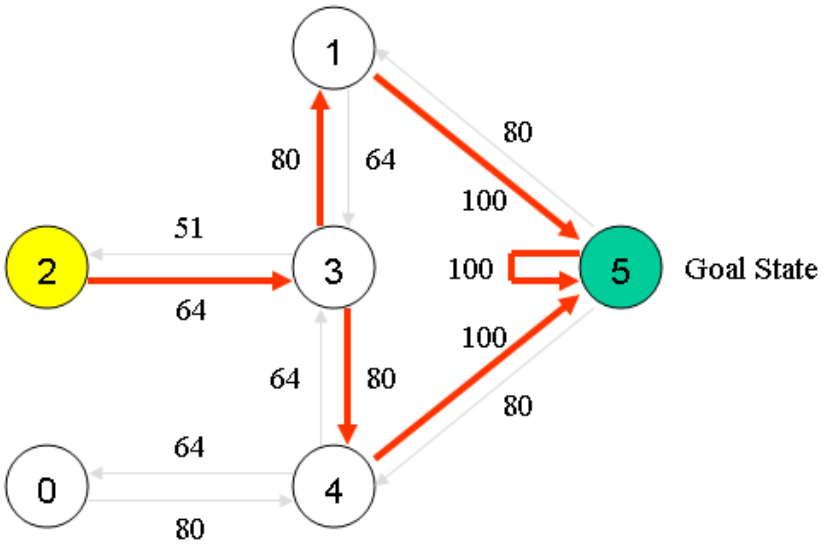
- If our agent learns more through further episodes, it will finally reach convergence values in matrix Q like:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

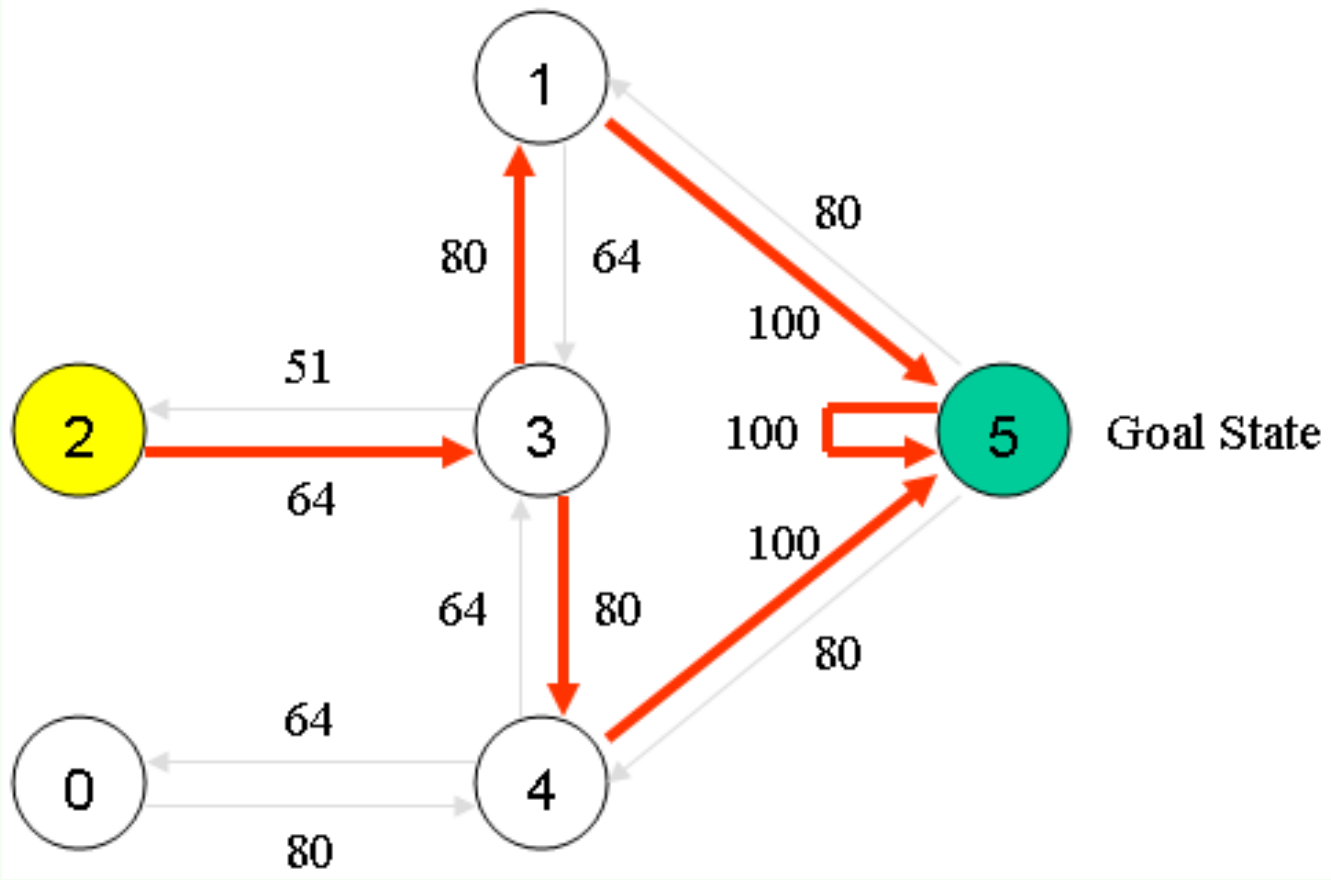
- **This matrix Q, can then be normalized (i.e.; converted to percentage) by dividing all non-zero entries by the highest number (500 in this case):**

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

- Once the matrix Q gets close enough to a state of convergence, we know our agent has learned the most optimal paths to the goal state. Tracing the best sequences of states is as simple as following the links with the highest values at each state.



For example, from initial State 2, the agent can use the matrix Q as a guide:
From State 2 the maximum Q values suggests the action to go to state 3.
From State 3 the maximum Q values suggest two alternatives: go to state 1 or 4. Suppose we arbitrarily choose to go to 1.
From State 1 the maximum Q values suggests the action to go to state 5.
Thus the sequence is 2 - 3 - 1 - 5.



The *n*-Armed Bandit Problem

- Choose repeatedly from one of *n* actions; each choice is called a **play**
- After each play a_t, you get a reward r_t, where

$$E\left\langle r_t \mid a_t \right\rangle = Q^*(a_t)$$

These are unknown **action values**
Distribution of r_t depends only on a_t

- Objective is to maximize the reward in the long term, e.g., over 1000 plays

To solve the *n*-armed bandit problem, you must **explore** a variety of actions and then **exploit** the best of them.

The Exploration/Exploitation Dilemma

- Suppose you form estimates

$$Q_t(a) \approx Q^*(a)$$ **action value estimates**

- The **greedy** action at *t* is

$$a_t^* = \arg\max_a Q_t(a)$$

$$a_t = a_t^* \Rightarrow \text{exploitation}$$

$$a_t \neq a_t^* \Rightarrow \text{exploration}$$

- You can't exploit all the time; you can't explore all the time
- You can never stop exploring; but you should always reduce exploring

Action-Value Methods

- Methods that adapt action-value estimates and nothing else, e.g.: suppose by the *t*-th play, action a had been chosen k_a times, producing rewards $r_1, r_2, \ldots, r_{k_a}$, then

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a}$$ "sample average"

$$\lim_{k_a \to \infty} Q_t(a) = Q^*(a)$$

ε-Greedy Action Selection

- Greedy action selection:

$$a_t = a_t^* = \arg\max_a Q_t(a)$$

- ε-Greedy:

$$a_t = \begin{cases} a_t^* & \text{with probability } 1-\varepsilon \\ \text{random action with probability } \varepsilon \end{cases}$$

… the simplest way to try to balance exploration and exploitation