# Agenda

| | |
|---|---|
| Day 3 | 1. Instance Based Learning: Introduction, K-nearest neighbor learning, locally weighted regression, radial basis function, cased based reasoning.<br>2. Reinforced Learning: Introduction, Learning Task, Q Learning |

- **Source Code**
- https://github.com/profthyagu

# 1.1 Instance Based Learning

- **Memory-based learning** (also called **instance-based learning**) is a type of learning algorithm that compares new test data with training data in order to solve the given machine learning problem. Such algorithms **search for the training data that are most similar to the test data** and make predictions based on these similarities.
- Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered a set of similar related instances is retrieved from memory and used to classify the new query instance
- **Examples :** *k-nearest neighbor learning , locally weighted regression,* ***Radial Basis function (RBF),*** *kernel machines and Case based Reasoning .*

# Instance based learning

- **Key idea**: In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning constructs the target function only when a new instance must be classified.

Only store all training examples $< x_i, f(x_i) >$ where *x* describes the attributes of each instance and *f(x)* denotes its class (or value).

**Use a Nearest Neighbor method:**

Given query instance $x_q$ ,

first locate nearest (most similar) training example $x_n$ ,

then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

Key idea: just store all training examples $\langle x_i, f(x_i)\rangle$

Nearest neighbor:

- Given query instance $x_q$, first locate nearest training example $x_n$, then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

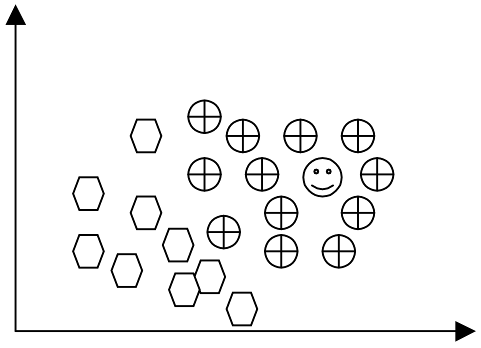
$k$-Nearest neighbor:

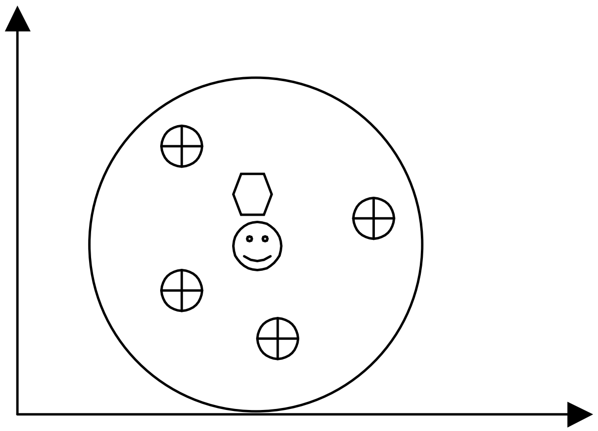
- Given $x_q$, take vote among its $k$ nearest nbrs (if discrete-valued target function)
- take mean of $f$ values of $k$ nearest nbrs (if real-valued)

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

## Example

Simple 2-D case, each instance described only by two values (x, y co-ordinates). The class is either ⊕ or ⬡





**Need to consider :**

1) Similarity (how to calculate distance)
2) Number (and weight) of similar (near) instances

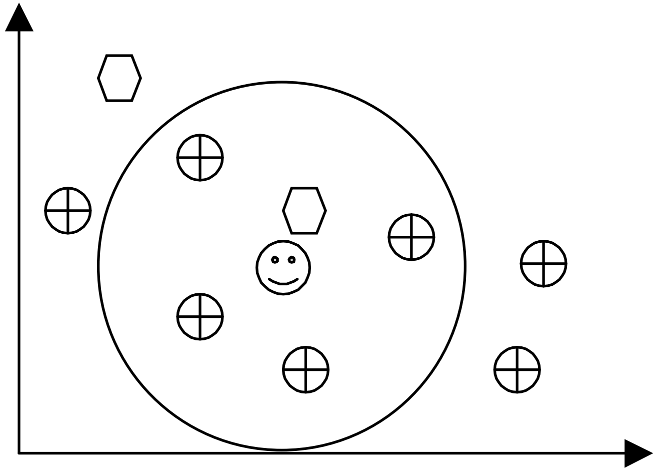
**Similarity:** Euclidean distance, more precisely let an arbitrary instance x be described by the feature vector (set of attributes) as follows: $< a_1(x), a_2(x), ... a_n(x) >$

where *ar(x)* denotes the value of the r^th^ attribute of instance *x*. Then the distance between two instances xi and xj is defined to be *d(x_i, x_j)* where

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

**k-Nearest Neighbor:**

Given query instance $x_q$ , take **vote** among its $k$ nearest neighbors to decide its class, (return most common value of $f$ among the $k$ nearest training elements to $x_q$ )



Simple-NN, class is ⬡ ,   5-NN, class is ⊗

Advantage: overcome class noise in training set.

**Key Advantages**

Instead of estimating the target function once for the entire data set (which can lead to complex and not necessarily accurate functions) IBL can estimate the required function locally and differently for each new instance to be classified.

**Algorithms for K-NN**

Consider, the case of learning a discrete-valued target function of the form $f : \Re^n \rightarrow V$, where $V$ is the finite set $\{v_1, ... v_s\}$

**Training Algorithm**

For each training example $< x\ ,\ f(x) >$,

add the example to the list of *training_examples*

# The K-NN algorithm for approximating a discrete valued function

Consider, the case of learning a discrete-valued target function of the form $f : \Re^n \rightarrow V$, where $V$ is the finite set $\{v_1, \ldots v_s\}$

**Training Algorithm :**

- For each training example *<x , f(x) >,* add the example to the list of *training_examples*

**Classification Algorithm**

- Given a query instance $x_q$ to be classified
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to $x_q$
  - Return $\hat{f}(x_q) \leftarrow \underset{\mathrm{v} \in \mathrm{V}}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$

$$\text{where } \delta(\mathrm{a}, \mathrm{b}) = 1 \quad \text{if } a = b$$
$$= 0 \quad \text{otherwise}$$

where argmax *f(x)* returns the value of *x* which maximises *f(x),*

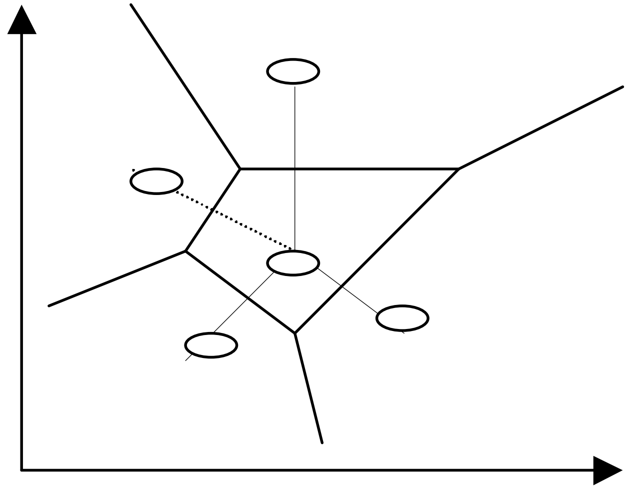
e.g. $\underset{x \in \{1,2,-3\}}{\operatorname{argmax}}(\mathrm{x}^2) = -3$

## Lazy vs Eager Learning

The K-NN method does not form an explicit hypothesis regarding the target classification function. It simply computes the classification for each new query instance as needed.

**Implied Hypothesis:** the following diagram (**Voronoi diagram**) shows the shape of the implied hypothesis about the decision surface that can be derived for a simple 1- NN case.

The decision surface is a combination of complex *polyhedra* surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example.



## Continuous vs Discrete valued functions (classes)

K-NN works well for discrete-valued target functions. Furthermore, the idea can be extended f or continuos (real) valued functions. In this case we can take mean of the $f$ values of $k$ nearest neighbors:

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

# When To Consider Nearest Neighbor ?

- Instances map to points in $\Re^n$
- Average number of attributes (e.g. Less than 20 attributes per instance)
- Lots of training data
- When target function is complex but can be approximated by separate local simple approximations

| **Advantages:** | **Disadvantages:** |
|---|---|
| Training is very fast | Slow at query time |
| Learn complex target functions | Easily fooled by irrelevant attributes |

Note that efficient methods do exist to allow fast querying (kd-trees)

# Distance-weighted k-NN

Might want to weigh nearer neighbors more heavily

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \qquad \text{where } w_i = \frac{1}{d(x_q, x_i)^2}$$

and $d(x_q, x_i)$ is distance between $x_q$ and $x_i$

**For continuous functions:**

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \qquad \text{where } w_i = \frac{1}{d(x_q, x_i)^2}$$

Note now it may make more sense to use all training examples instead of just *k*.

# Curse of Dimensionality

Imagine instances described by 20 attributes, but only 2 are relevant to target function: Instances that have identical values for the two relevant attributes may nevertheless be distant from one another
in the 20-dimensional space.

**Curse of dimensionalit**y: nearest neighbor is easily misled when high-dimensional X. (Compare to decision trees).

**One approach: Weight each attribute differently (Use training)**

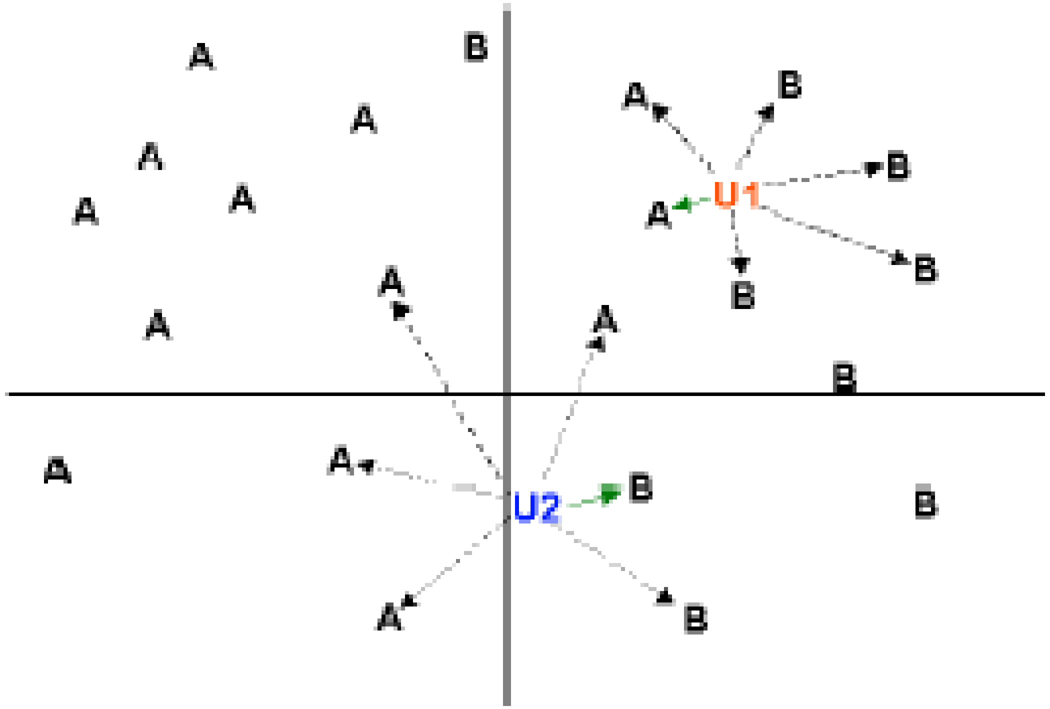
1) Stretch $j^{th}$ axis by weight $z_j$ , where $z_1, \ldots, z_n$ chosen to minimize prediction error
2) Use cross-validation to automatically choose weights $z_1, \ldots, z_n$
3) Note setting $z_j$ to zero eliminates dimension $i$ altogether

# Lab Program

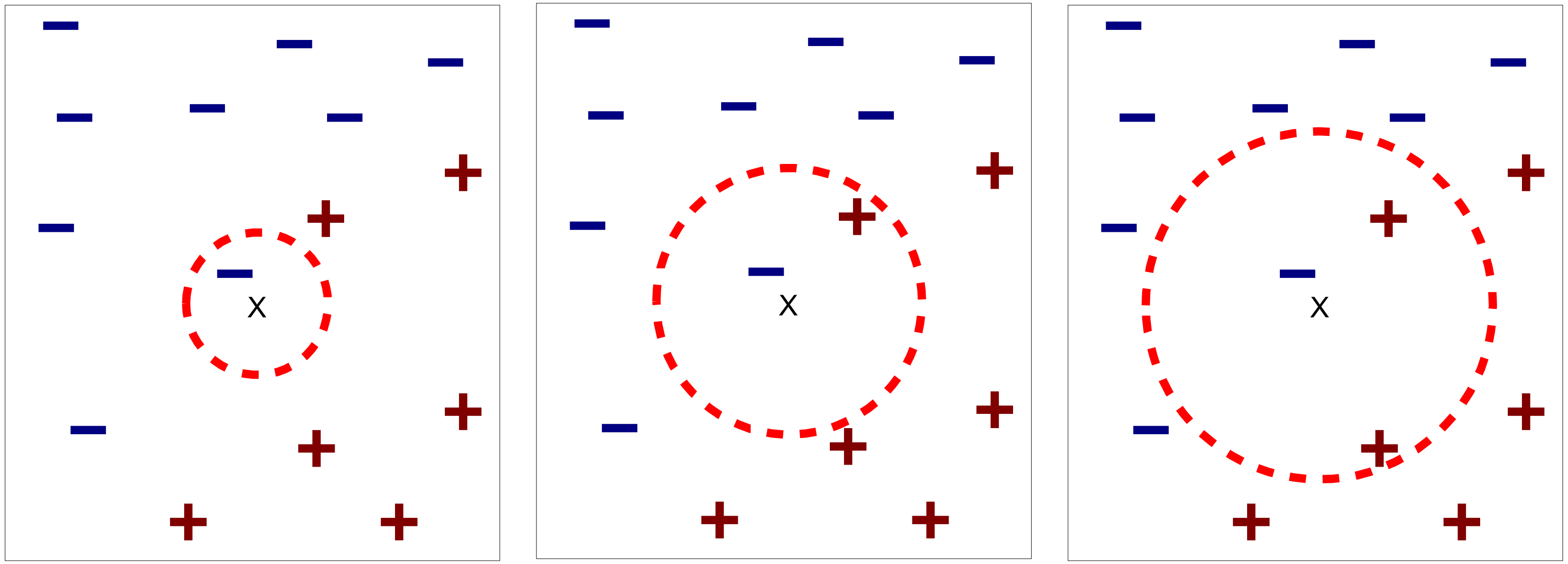
- Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Python ML library classes can be used for this problem.

# K-Nearest-Neighbor Algorithm

- **Principle:** points (documents) that are close in the space belong to the same class



# Definition of Nearest Neighbor



(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

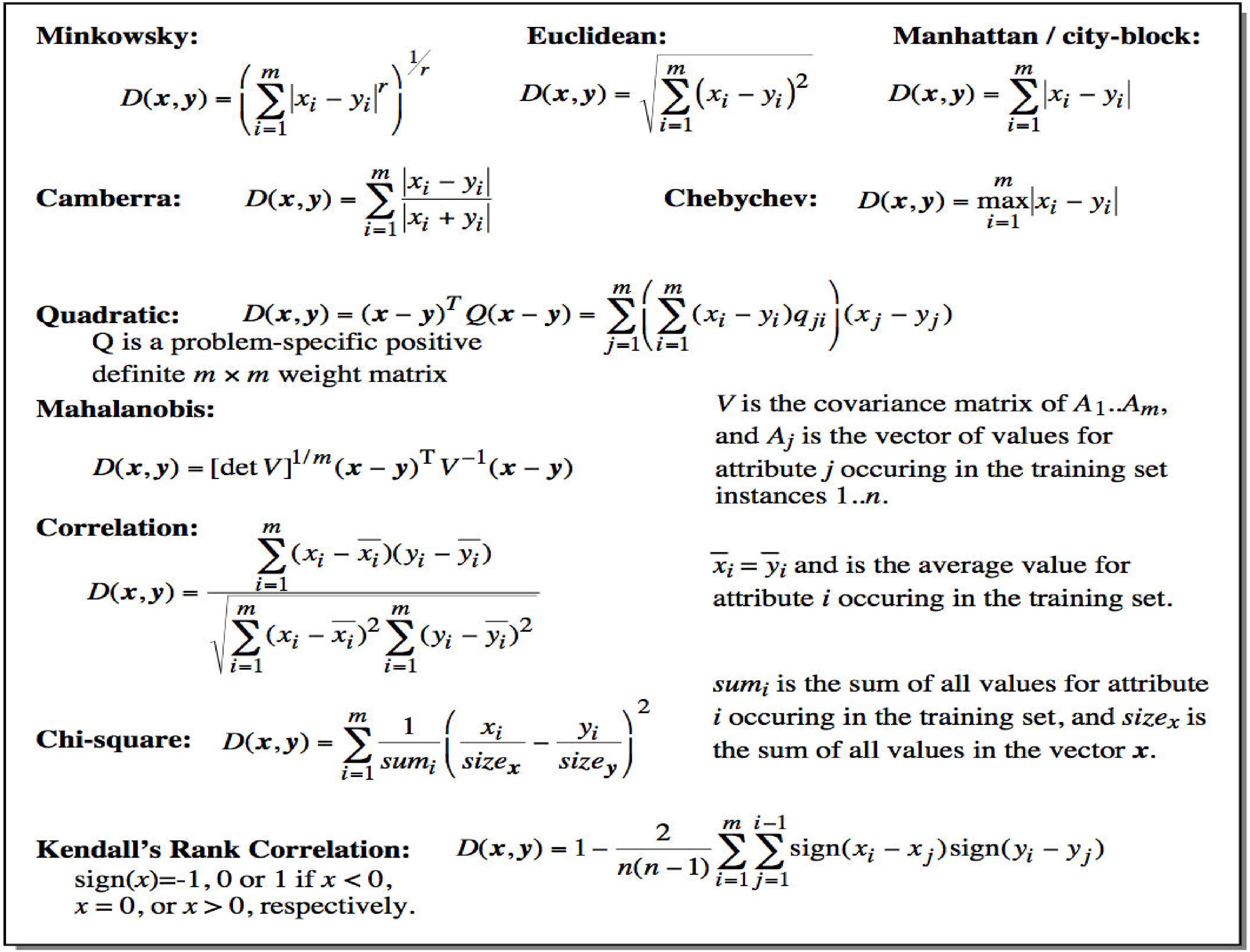
K-nearest neighbors of a record x are data points that have the k smallest distance to x

# Nearest Neighbor Classification

- Compute distance between two points:
  - Euclidean distance

$$d(p,q) = \sqrt{\sum_{i} (p_{i} - q_{i})^{2}}$$

# Distance Metrics



**Minkowsky:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \left( \sum_{i=1}^{m} |x_i - y_i|^r \right)^{1/r}$$

**Euclidean:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2}$$

**Manhattan / city-block:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{m} |x_i - y_i|$$

**Camberra:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{m} \frac{|x_i - y_i|}{|x_i + y_i|}$$

**Chebychev:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \max_{i=1}^{m} |x_i - y_i|$$

**Quadratic:**

$$D(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x} - \boldsymbol{y})^T Q (\boldsymbol{x} - \boldsymbol{y}) = \sum_{j=1}^{m} \left( \sum_{i=1}^{m} (x_i - y_i) q_{ji} \right) (x_j - y_j)$$

Q is a problem-specific positive definite $m \times m$ weight matrix

**Mahalanobis:**

$$D(\boldsymbol{x},\boldsymbol{y}) = [\det V]^{1/m} (\boldsymbol{x} - \boldsymbol{y})^{\mathrm{T}} V^{-1} (\boldsymbol{x} - \boldsymbol{y})$$

$V$ is the covariance matrix of $A_1..A_m$, and $A_j$ is the vector of values for attribute $j$ occuring in the training set instances $1..n$.

**Correlation:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \frac{\sum_{i=1}^{m} (x_i - \overline{x_i})(y_i - \overline{y_i})}{\sqrt{\sum_{i=1}^{m} (x_i - \overline{x_i})^2 \sum_{i=1}^{m} (y_i - \overline{y_i})^2}}$$

$\overline{x_i} = \overline{y_i}$ and is the average value for attribute $i$ occuring in the training set.

**Chi-square:**

$$D(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{m} \frac{1}{sum_i} \left( \frac{x_i}{size_{\boldsymbol{x}}} - \frac{y_i}{size_{\boldsymbol{y}}} \right)^2$$

$sum_i$ is the sum of all values for attribute $i$ occuring in the training set, and $size_{\boldsymbol{x}}$ is the sum of all values in the vector $\boldsymbol{x}$.

**Kendall's Rank Correlation:**

sign($x$)=-1, 0 or 1 if $x < 0$, $x = 0$, or $x > 0$, respectively.

$$D(\boldsymbol{x},\boldsymbol{y}) = 1 - \frac{2}{n(n-1)} \sum_{i=1}^{m} \sum_{j=1}^{i-1} \mathrm{sign}(x_i - x_j)\,\mathrm{sign}(y_i - y_j)$$

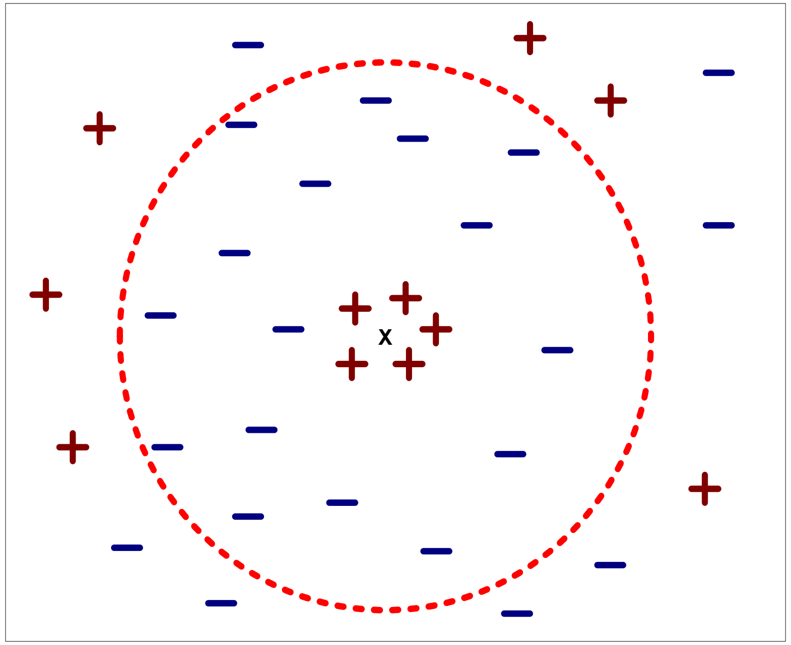
Figure 1. Equations of selected distance functions.
($\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of $m$ attribute values).

# Selection of Distance Metrics

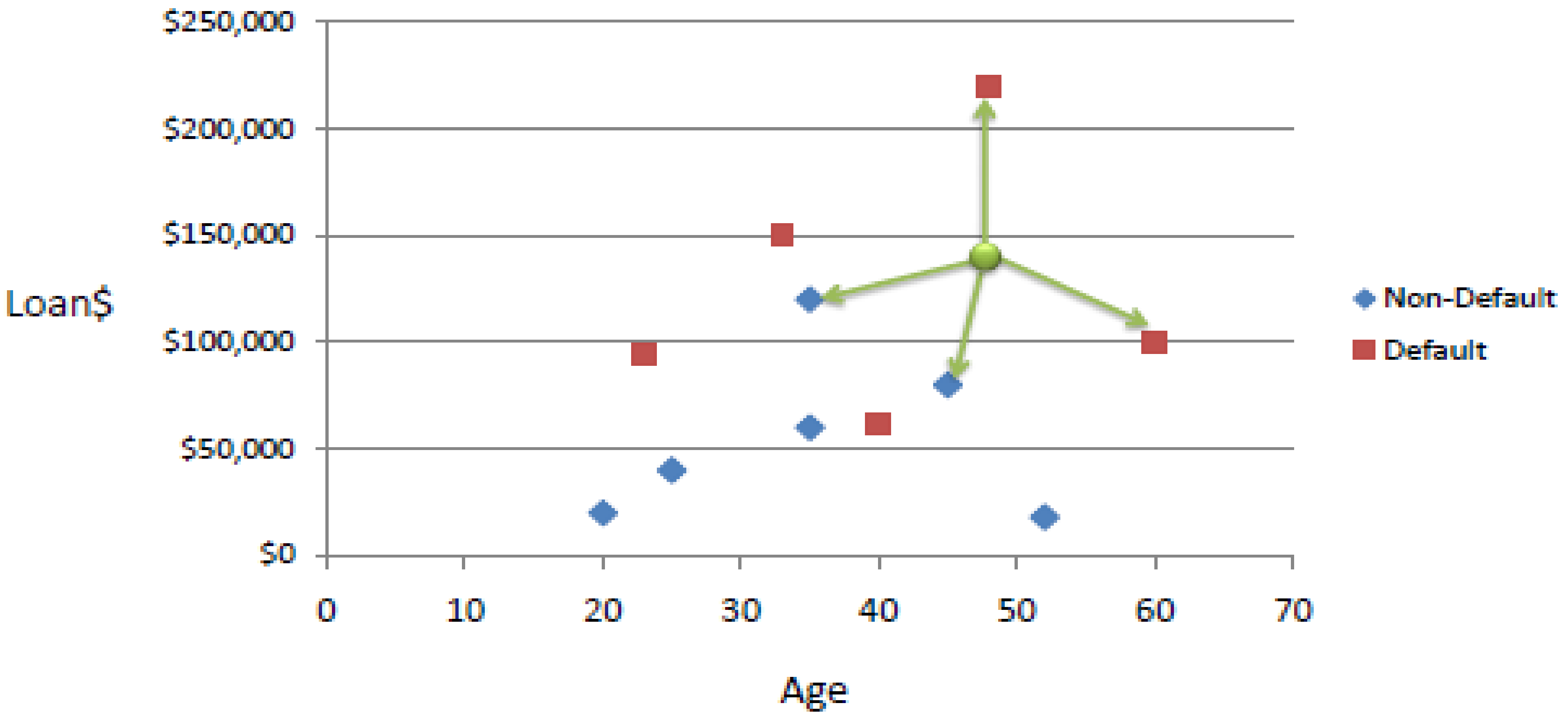
- You can choose the best distance metric based on the properties of your data. *If you are unsure, you can experiment with different distance metrics and different values of K together and see which mix results in the most accurate models*.
- Euclidean is a good distance measure to use if the input variables are similar in type (e.g. all measured widths and heights).
- Manhattan distance is a good measure to use if the input variables are not similar in type (such as age, gender, height, etc.).

# Nearest Neighbor Classification…

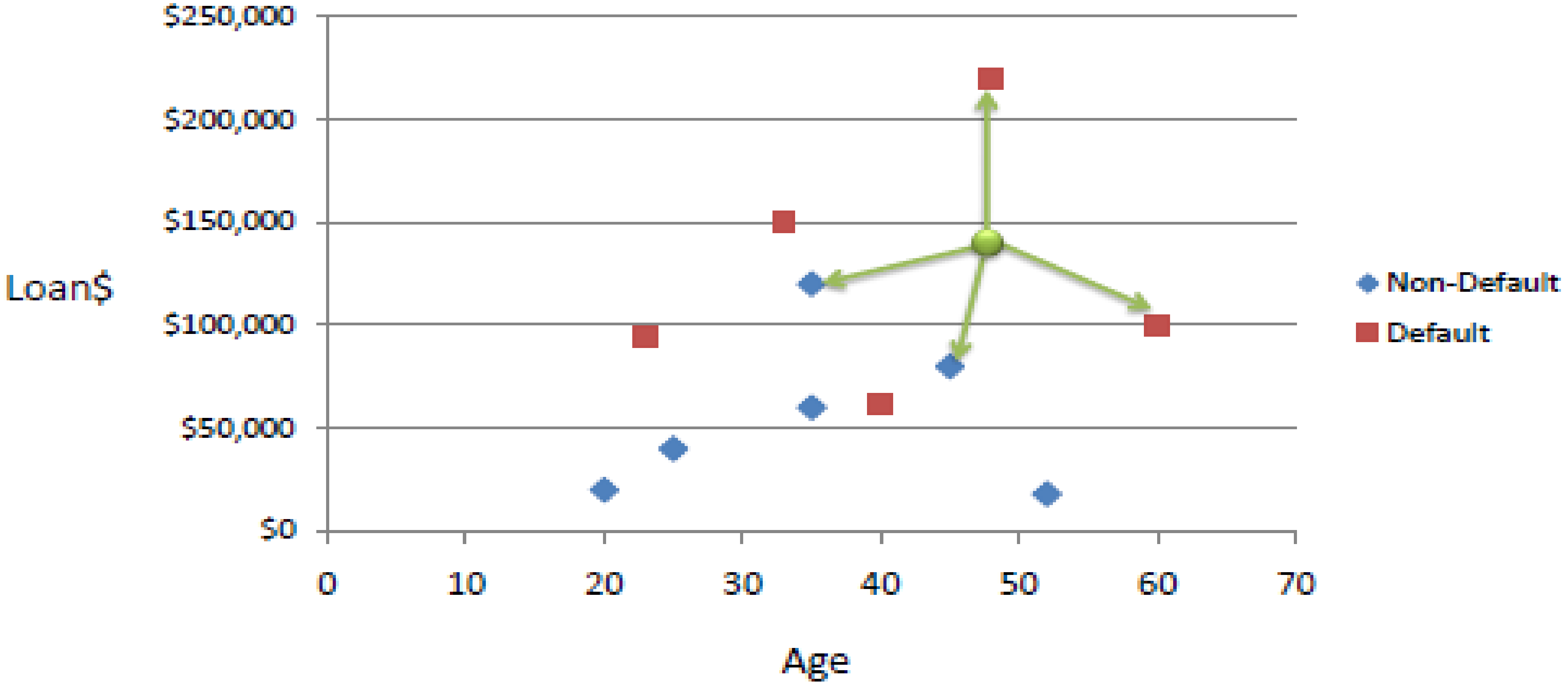
- **Choosing the value of k:**
  - If k is too small, sensitive to noise points
  - If k is too large, neighborhood may include points from other classes



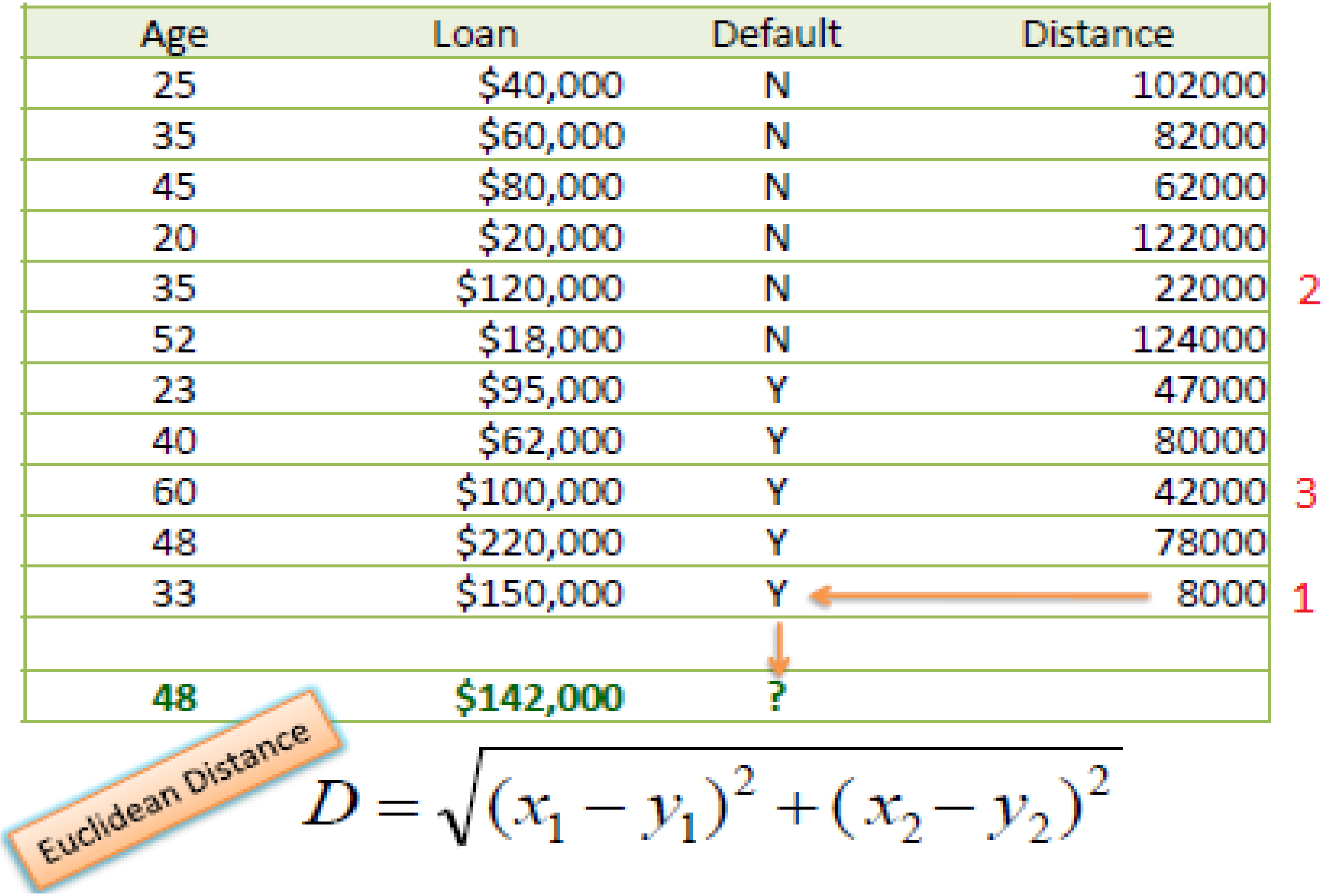
# Example: Consider the following data concerning credit default. Age and Loan are two numerical variables (predictors) and Default is the target.



Example: We can now use the training set to classify an unknown case (Age=48 and Loan=$142,000) using Euclidean distance. If K=1 then the nearest neighbor is the last case in the training set with Default=Y.



D = sqrt[(48-33)^2 + (142000-150000)^2] = 8000.01  >> Default=Y



| Age | Loan | Default | Distance | |
|---|---|---|---|---|
| 25 | $40,000 | N | 102000 | |
| 35 | $60,000 | N | 82000 | |
| 45 | $80,000 | N | 62000 | |
| 20 | $20,000 | N | 122000 | |
| 35 | $120,000 | N | 22000 | 2 |
| 52 | $18,000 | N | 124000 | |
| 23 | $95,000 | Y | 47000 | |
| 40 | $62,000 | Y | 80000 | |
| 60 | $100,000 | Y | 42000 | 3 |
| 48 | $220,000 | Y | 78000 | |
| 33 | $150,000 | Y | 8000 | 1 |
| | | | | |
| 48 | $142,000 | ? | | |

Euclidean Distance

$$D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

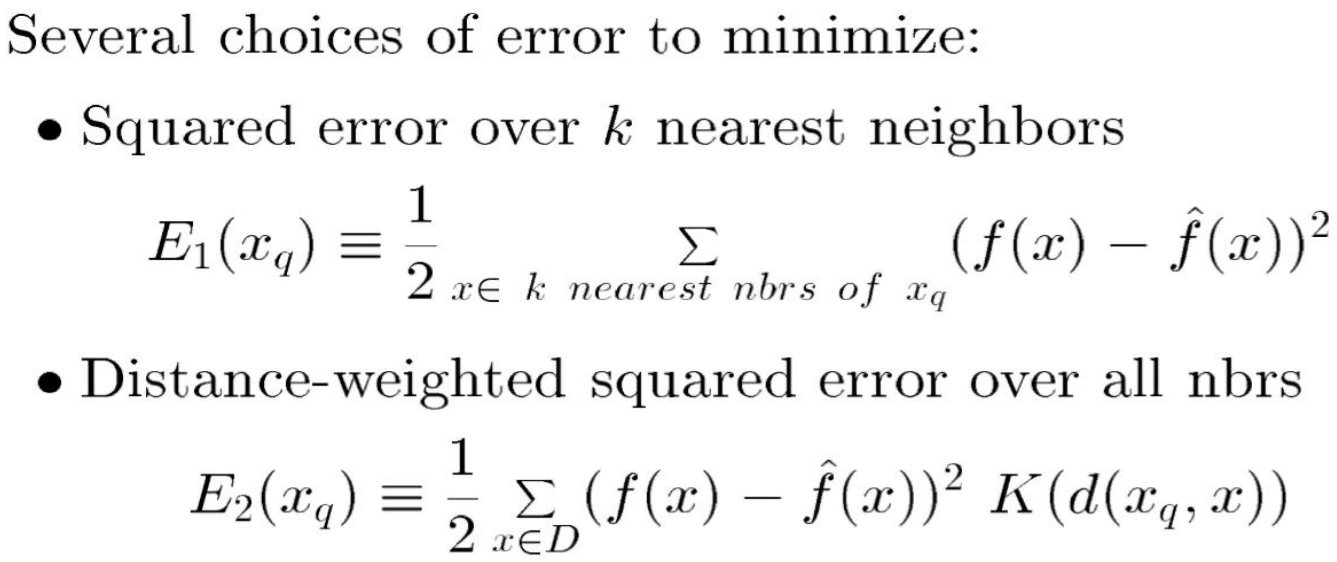
**With K=3, there are two Default=Y and one Default=N out of three closest neighbors. The prediction for the unknown case is again Default=Y.**

# Source Code

# 1.2 Locally-weighted Regression

**Basic idea:** k-NN forms local approximation to $f$ for each query point $x_q$
Why not form an explicit approximation f (x) for region surrounding $x_q$

- Fit linear function to k nearest neighbors
- Fit quadratic, ...
- Thus producing ``**piecewise approximation**"  to $f$



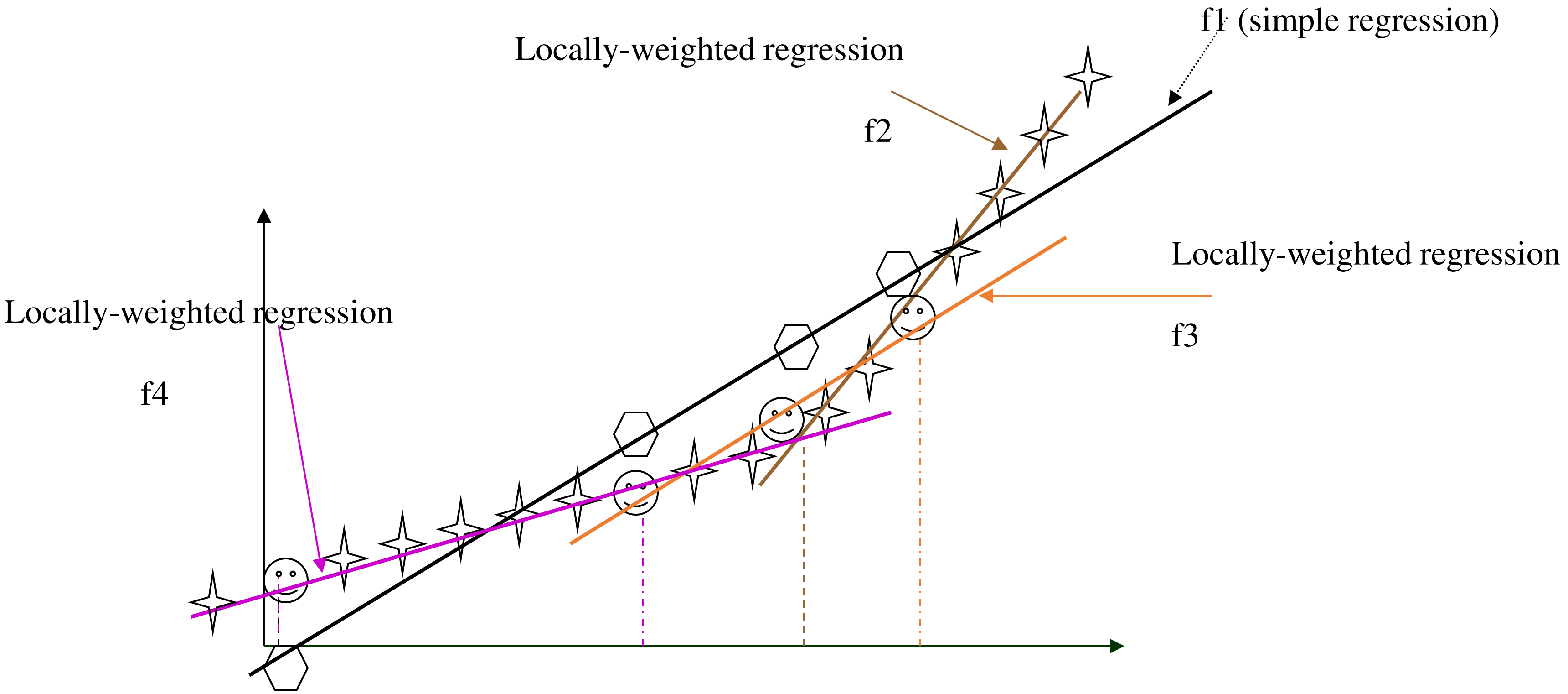
Several choices of error to minimize:

- Squared error over $k$ nearest neighbors

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in \; k \; nearest \; nbrs \; of \; x_q} (f(x) - \hat{f}(x))^2$$

- Distance-weighted squared error over all nbrs

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$


f1 (simple regression)
Locally-weighted regression
f2
Locally-weighted regression
f3
Locally-weighted regression
f4

✧ Training data

⬡ Predicted value using simple regression

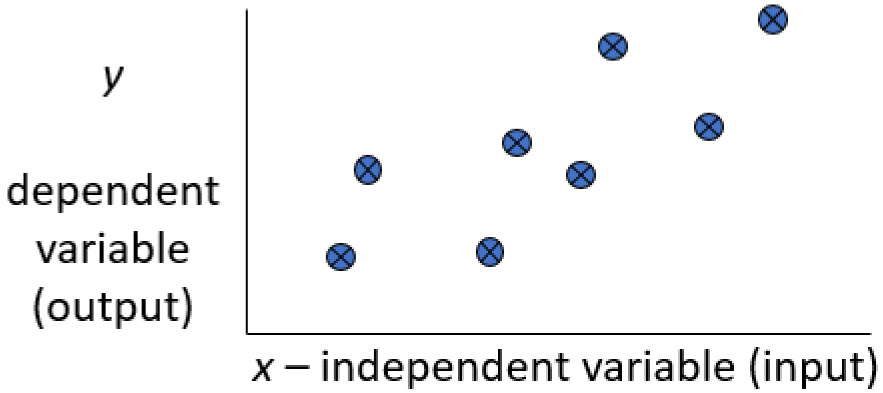
☺ Predicted value using locally weighted (piece-wise) regression

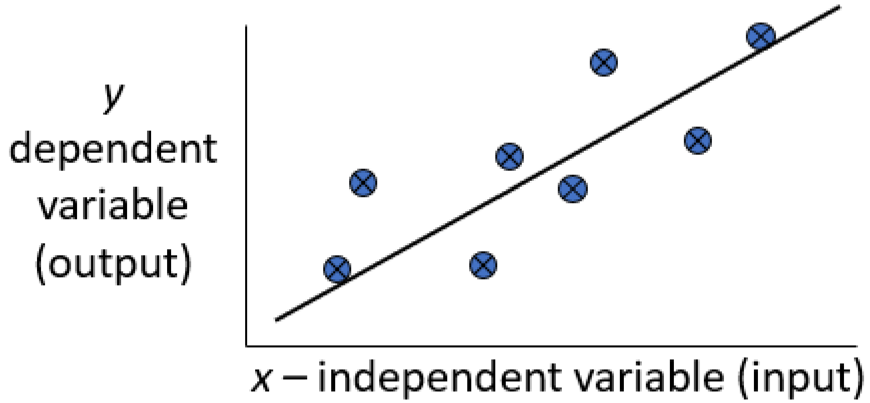
# Lab Program 8

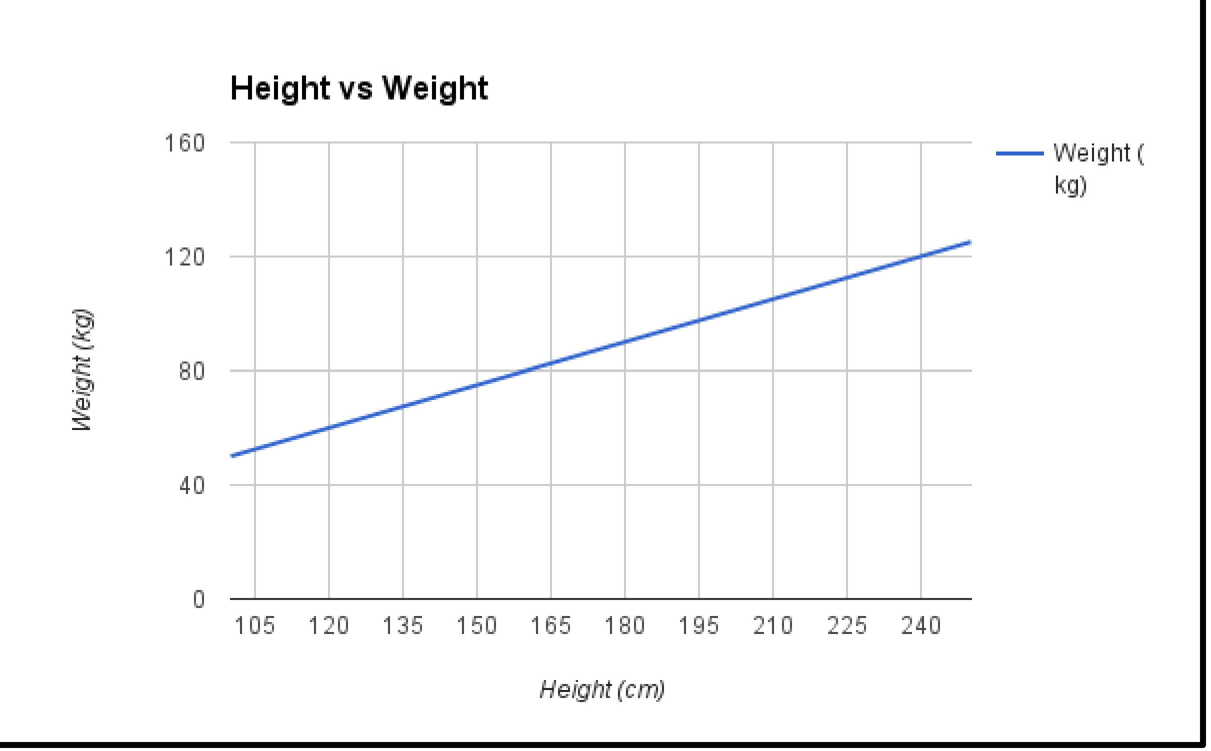
- **Implement the non-parametric Locally Weighted Regression (LOWESS) algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.**

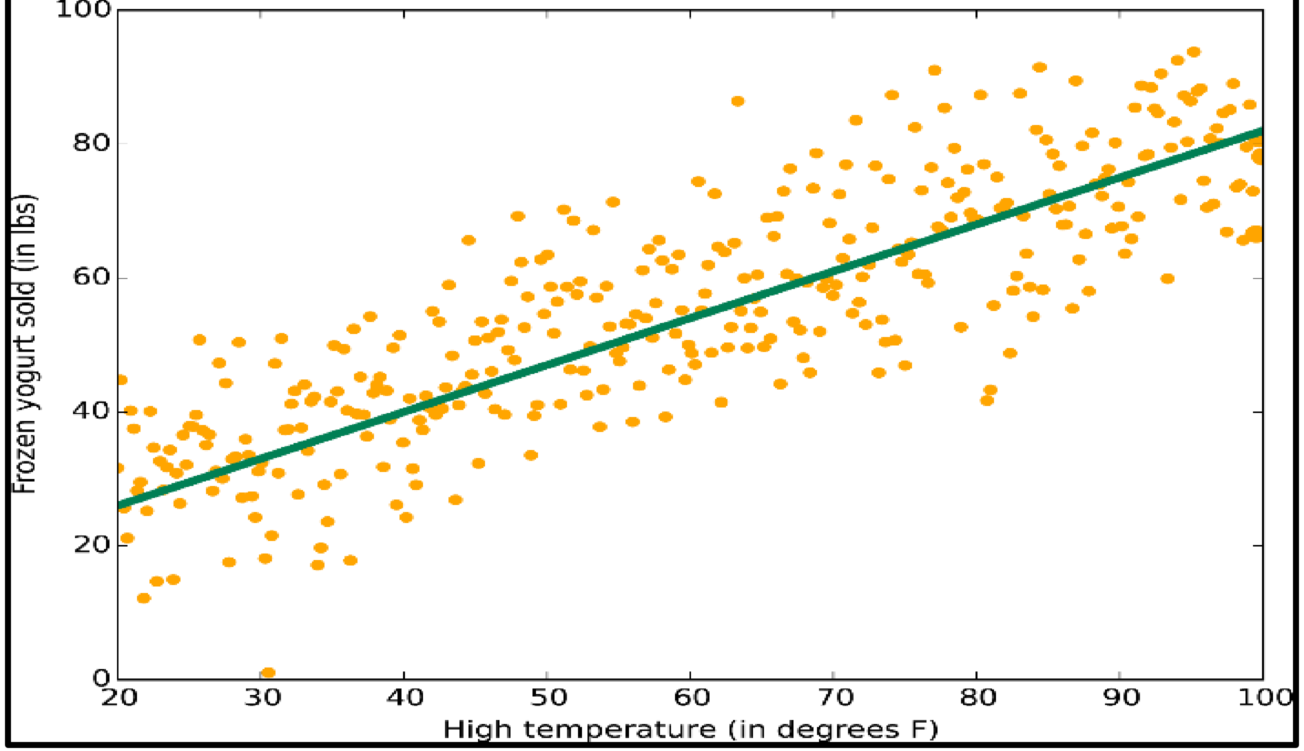
# Regression

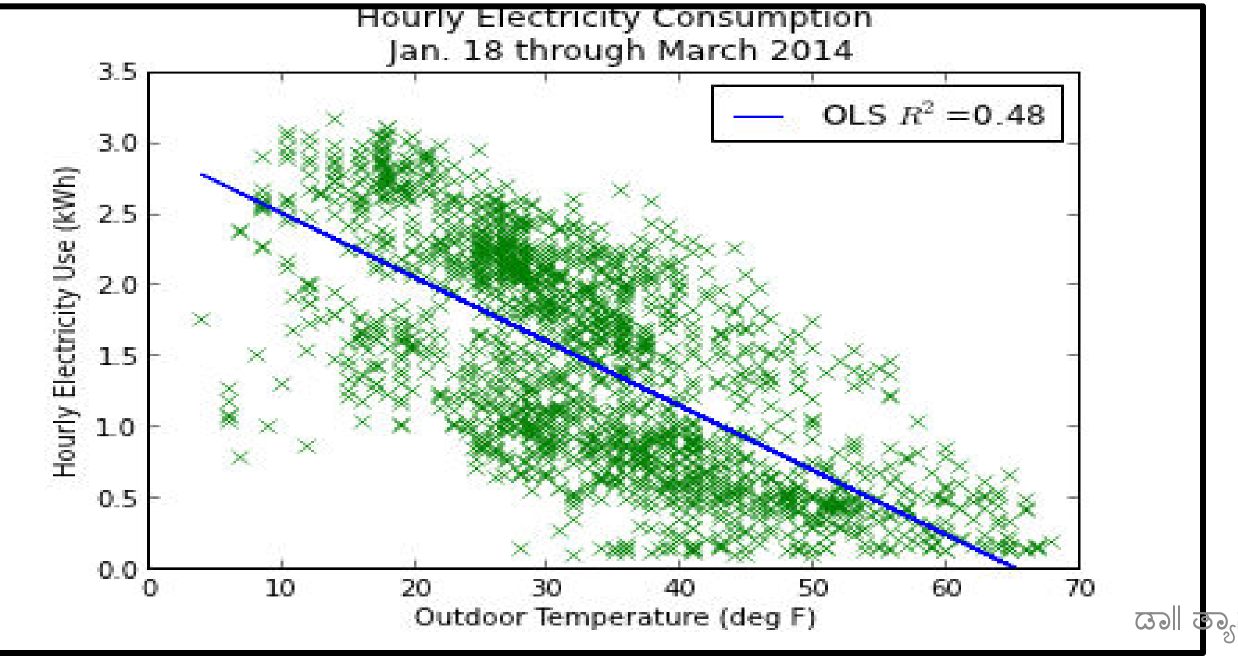
- Regression is a technique from statistics that is used to predict values of a desired target quantity when the target quantity is continuous .
- In regression, we seek to identify (or estimate) a continuous variable y associated with a given input vector x.
  - y is called the dependent variable.
  - x is called the independent variable.

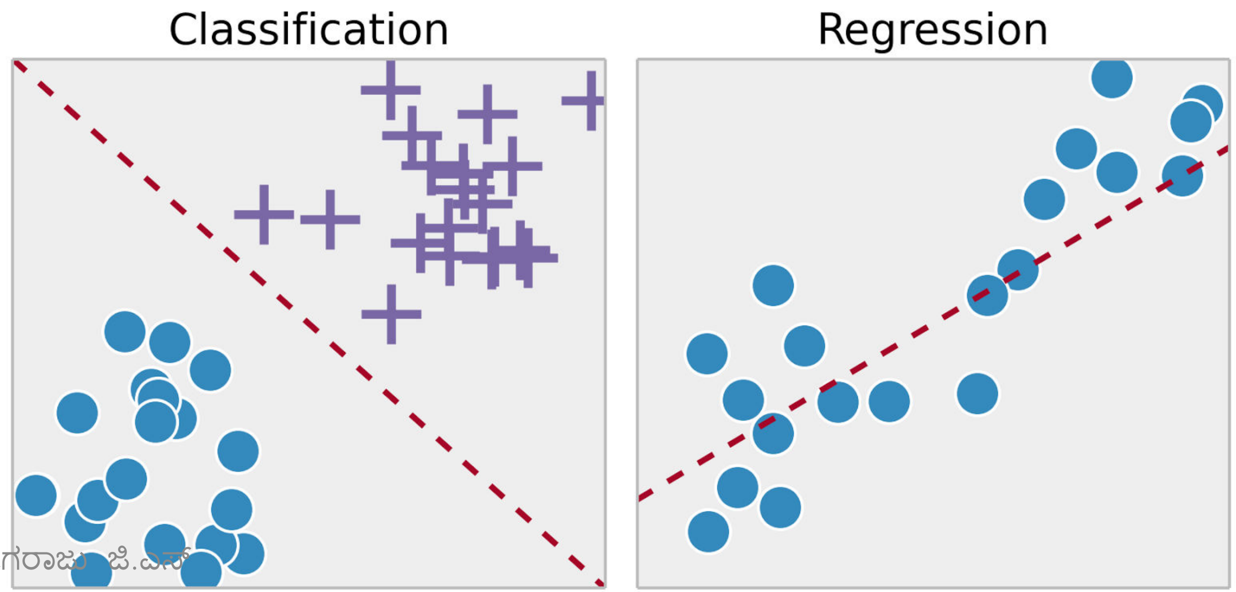




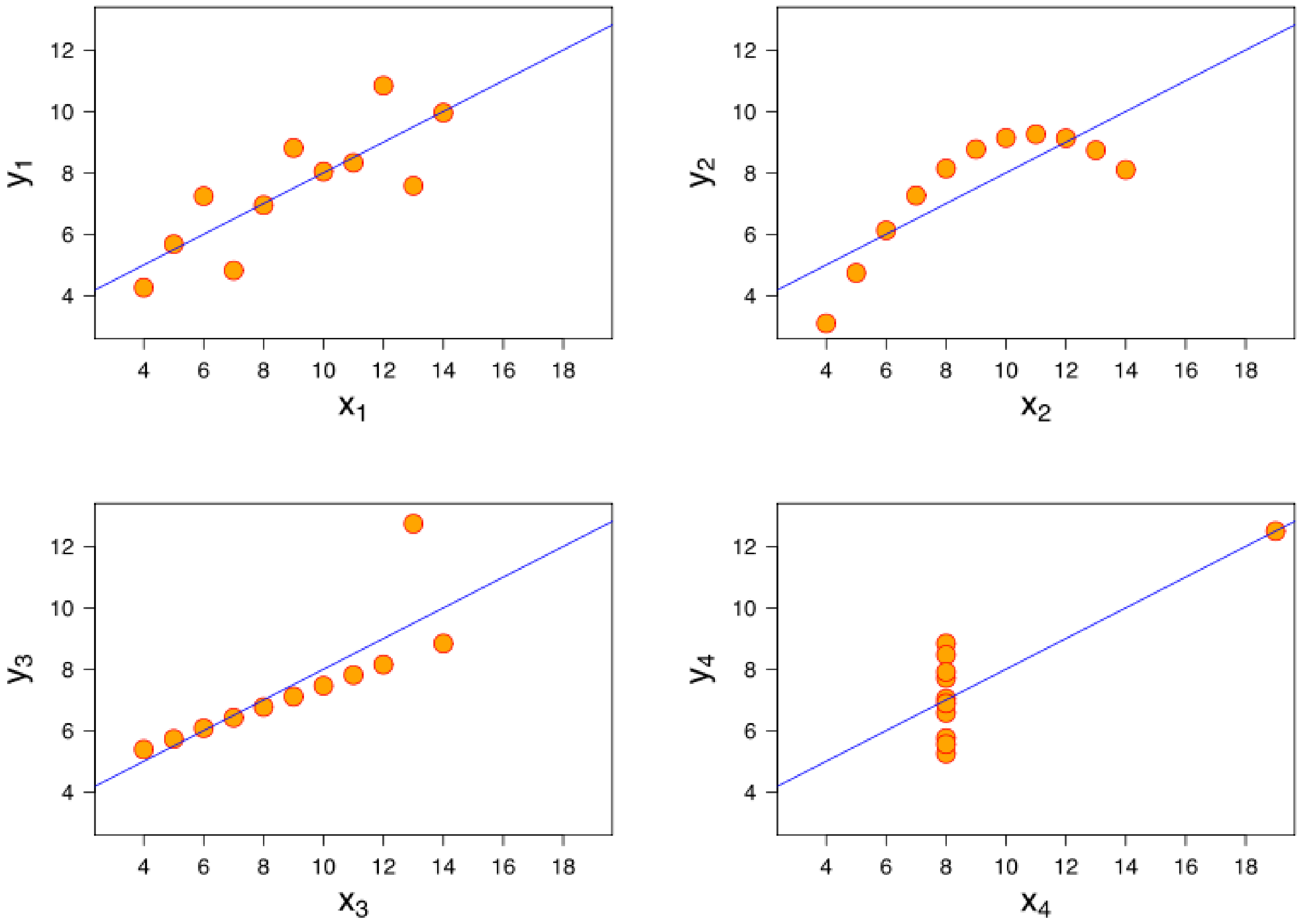

Height vs Weight
Weight (kg)
Height (cm)
Weight ( kg)
0
40
80
120
160
105
120
135
150
165
180
195
210
225
240


Frozen yogurt sold (in lbs)
High temperature (in degrees F)
0
20
40
60
80
100
20
30
40
50
60
70
80
90
100


Hourly Electricity Consumption
Jan. 18 through March 2014
OLS $R^2$ =0.48
Hourly Electricity Use (kWh)
Outdoor Temperature (deg F)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
10
20
30
40
50
60
70

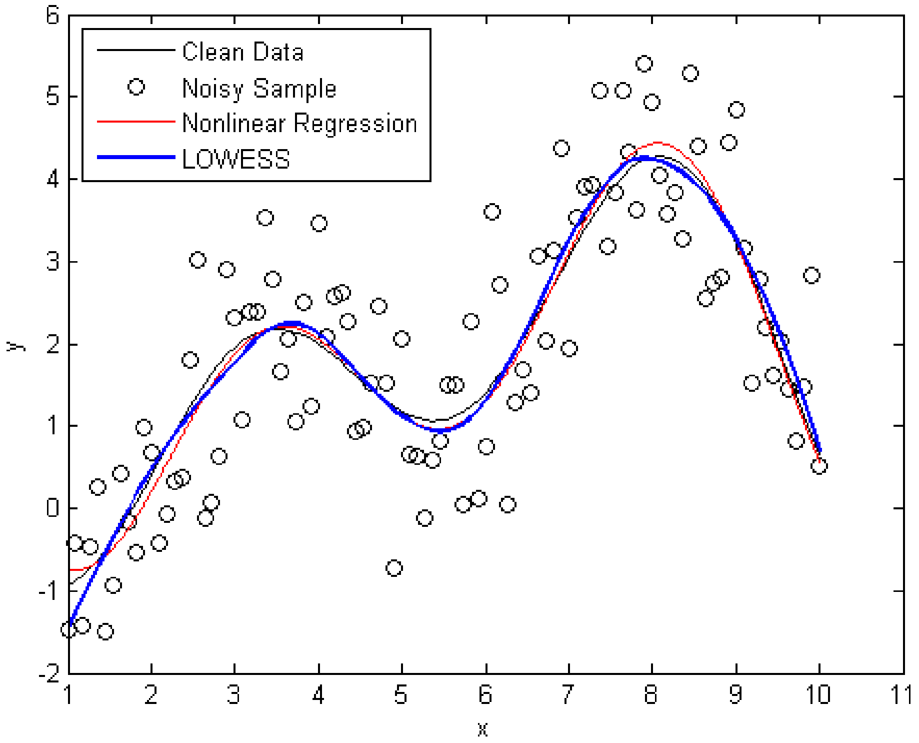

Classification
Regression

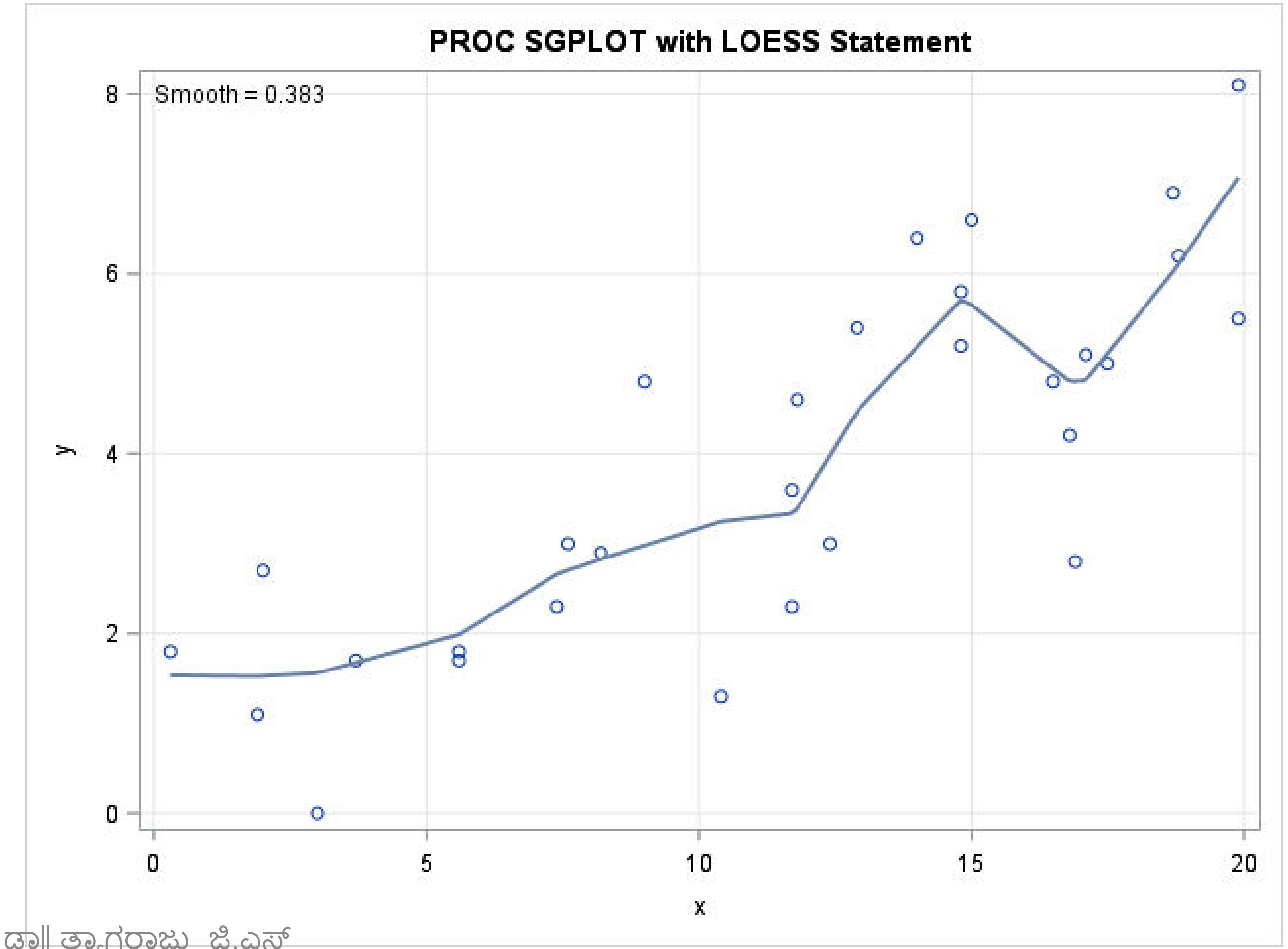
# What lines "***really***" best fit each case?



# Loess/Lowess Regression

- Loess regression is a nonparametric technique that uses ***local weighted*** regression to fit a **smooth curve** through points in a scatter plot.





# Lowess Algorithm

- Locally weighted regression is a very powerful non-parametric model used in statistical learning .Given a *dataset* **X**, **y**, we attempt to find a *model* parameter β**(x)** that minimizes *residual sum of* ***weighted*** *squared errors*. The weights are given by a *kernel function(k or w)* which can be chosen arbitrarily .

## Algorithm

1. Read the Given data Sample to **X** and the curve (linear or non linear) to **Y**
2. Set the value for Smoothening parameter or Free parameter say **τ**
3. Set the bias /Point of interest set **X0** which is a subset of **X**
4. Determine the weight matrix using :

$$w(x, x_o) = e^{-\frac{(x-x_o)^2}{2\tau^2}}$$

5. Determine the value of model term parameter β using :

$$\hat{\beta}(x_o) = (X^T W X)^{-1} X^T W y$$

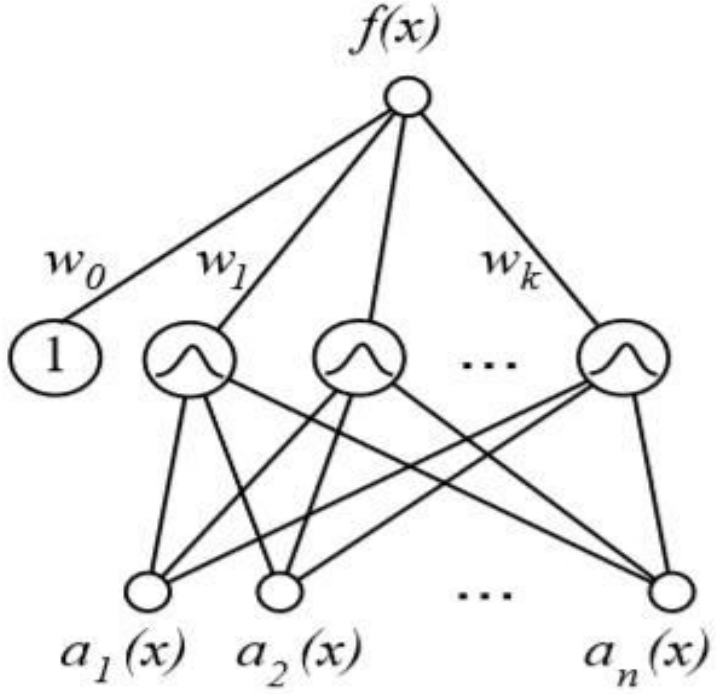
6. Prediction = $X_0 * \beta$

# Source Code

# 1.3 Radial basis Function Networks

- One approach to function approximation that is closely related to distance *weighted regression and also to artificial neural network its learning with radial basis functions .It is eager instead of lazy*
- *Global approximation to target function in terms of linear combination of local approximations*
- *Used e.g. for image classification*
- In this approach the learned hypothesis is a function of the form

$$f(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))$$



where $a_i(x)$ are the attributes describing instance $x$, and

$$f(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))$$

One common choice for $K_u(d(x_u, x))$ is

$$K_u(d(x_u, x)) = e^{-\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

# Training of RBF network

- Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process.
- First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of $x_u$ and $\sigma_u$^$_2$: that define its kernel function K,(d(x,, x)).
- Second, the weights w, are trained to maximize the fit of the network to the training data, using the global error criterion given by Equation

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2$$

- Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently.

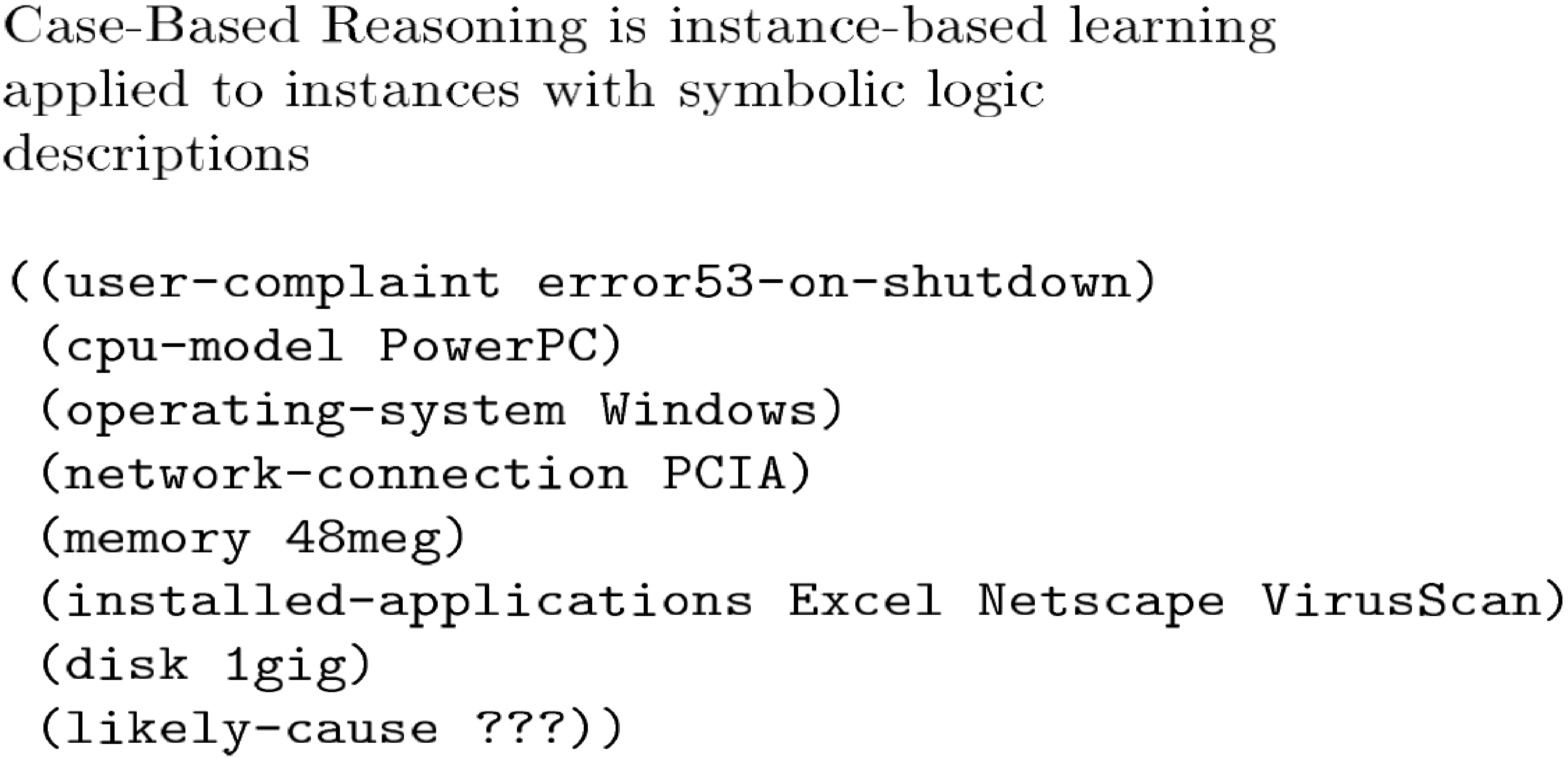
# 1.4 Case Based Reasoning

- Instance-based methods such as k-NEAREST NEIGHBOUR and locally weighted regression share three key properties.
    - First, they are lazy learning methods in that they defer the decision of how to generalize beyond the training data until a new query instance is observed.
    - Second, they classify new query instances by ana- lyzing similar instances while ignoring instances that are very different from the query.
    - Third, they represent instances as real-valued points in an n-dimensional Euclidean space
- Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third

# Case Based Reasoning

- In CBR, instances are typically represented using more rich symbolic descriptions, and the methods used to retrieve similar instances are correspondingly more elaborate.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a *stored library of previous designs* (Sycara et al. 1992), *reasoning about new legal cases based on previous rulings* (Ashley 1990), and *solving planning* and *scheduling problems by reusing and combining portions* of previous solutions to similar problems (Veloso 1992).

# Case Based Reasoning



Case-Based Reasoning is instance-based learning applied to instances with symbolic logic descriptions

```
((user-complaint error53-on-shutdown)
 (cpu-model PowerPC)
 (operating-system Windows)
 (network-connection PCIA)
 (memory 48meg)
 (installed-applications Excel Netscape VirusScan)
 (disk 1gig)
 (likely-cause ???))
```

# Case Based Reasoning in CADET

- The CADET system (Sycara et al. 1992) employs case- based reasoning to assist in the conceptual design of ***simple mechanical devices such as water faucets.***

- It uses a library containing approximately ***75 previous designs and design fragments*** to suggest conceptual designs to meet the ***specifications of new design problems.***

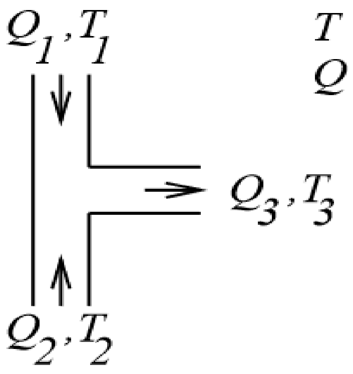
- Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.

- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

# Case Based Reasoning in CADET

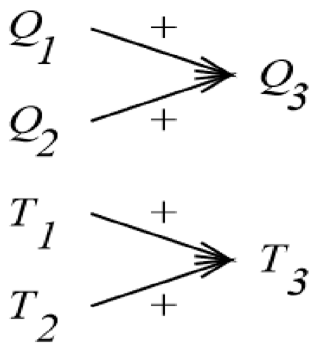
**A stored case:** T–junction pipe

Structure:



$T$ = temperature
$Q$ = waterflow

Function:

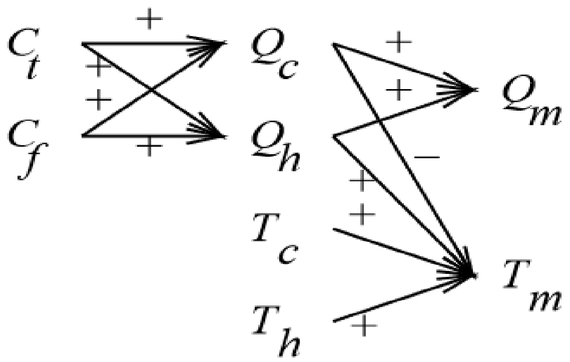


**A problem specification:** Water faucet

Structure:

?

Function:



- The top half of the figure shows the description of a typical stored case called a T-junction pipe. Its function is represented in terms of the qualitative relationships among the waterflow levels and temperatures at its inputs and outputs.
- In the functional description at its right, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail.
- For example, the output waterflow Q3 increases with increasing input waterflow Ql.
- Similarly a "-" label indicates that the variable at the head decreases with the variable at the tail.

- The bottom half of this figure depicts a new design problem described by its desired function. This particular function describes the required behavior of one type of water faucet.
- Here Qc, refers to the flow of cold water into the faucet, Qh to the input flow of hot water, and Qm, to the single mixed flow out of the faucet.
- Similarly, Tc, Th, and Tm , refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable Ct, denotes the control signal for temperature that is input to the faucet, and Cf denotes the control signal for waterflow. Note the description of the desired function specifies that these controls Ct, and Cf are to influence the water flows Qc, and Qh, thereby indirectly influencing the faucet output flow Qm, and temperature Tm.

- Given this functional specification for the new design problem, CADET searches its library for stored cases whose functional descriptions match the design problem.
- If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem.
- If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification.

# Generic properties of case-based reasoning system

- Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET. This may require a similarity metric different from Euclidean distance, such as the size of the largest shared subgraph between two function graphs.
-

# Generic properties of case-based reasoning system

- Multiple retrieved cases may be combined to form the solution to the new problem.
- This is similar to the k-NEAREST NEIGHBOR approach, in that multiple similar cases are used to construct a response for the new query.
- However, the process for combining these multiple retrieved cases can be very different, relying on knowledge-based reasoning rather than statistical methods.
-

# Generic properties of case-based reasoning system

- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving.
- One simple example of this is found in CADET, which uses generic knowledge about influences to rewrite function graphs during its attempt to find matching cases.
- Other systems have been developed that more fully integrate case-based reasoning into general search- based problem-solving systems. Two examples are ANAPRON (Golding and Rosenbloom 199 1) and PRODIGY/ANALOGY (Veloso 1992).

# Summary

- To summarize, case-based reasoning is an instance-based learning method in which instances (cases) may be rich relational descriptions and in which the retrieval and combination of cases to solve the current query may rely on knowledge- based reasoning and search-intensive problem-solving methods.
- One current re- search issue in case-based reasoning is to develop improved methods for indexing cases. The central issue here is that syntactic similarity measures (e.g., subgraph isomorphism between function graphs) provide only an approximate indication of the relevance of a particular case to a particular problem.

# 2. Reinforced Learning

- Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.
- This very generic problem covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state.
    - *For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward.*
- This chapter focuses on an algorithm called Q learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

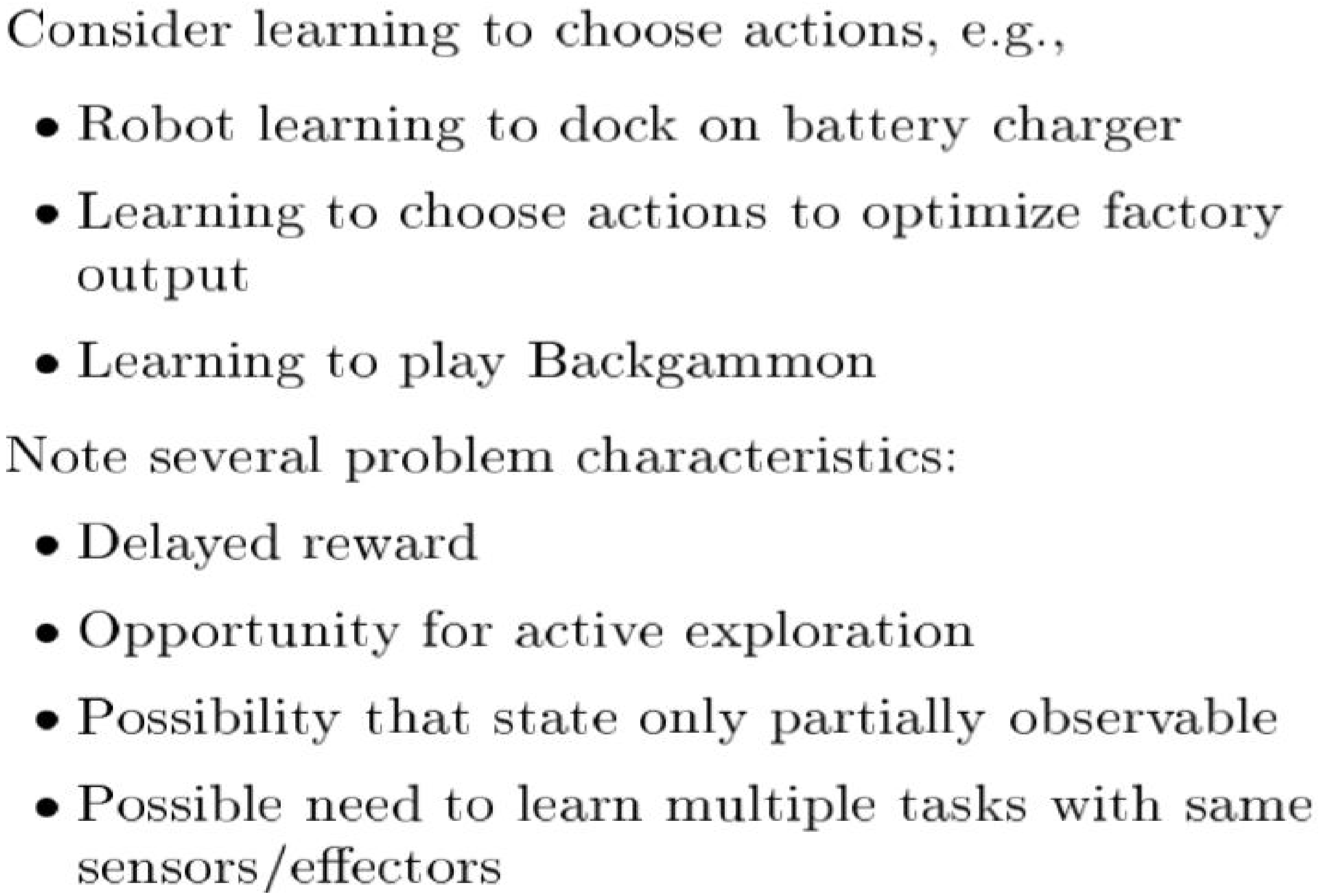
# Building a  Learning Robot

- Consider building a learning robot. The robot, or agent, has a *set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.*
  - For example, a mobile robot may have sensors such as a *camera and sonars*, and actions such as "move forward" and "turn.“
- Its task is to learn a control strategy, or policy, for choosing actions that achieve its goals.
  - For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

# 2. Reinforced Learning

- Control Learning
- Control policies that choose optimal actions
- Q learning
- Convergence

# Control Learning



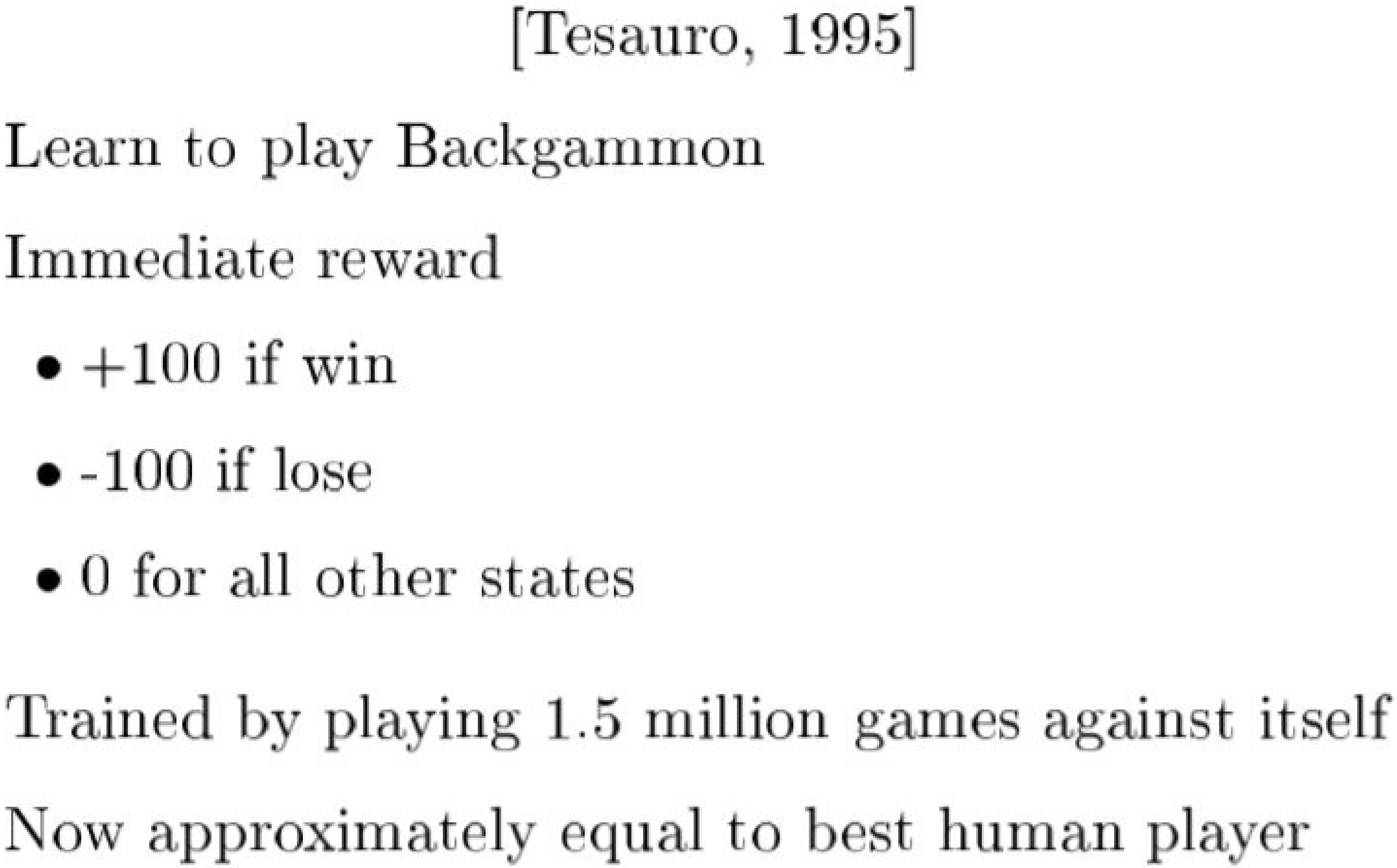
Consider learning to choose actions, e.g.,

- Robot learning to dock on battery charger
- Learning to choose actions to optimize factory output
- Learning to play Backgammon

Note several problem characteristics:

- Delayed reward
- Opportunity for active exploration
- Possibility that state only partially observable
- Possible need to learn multiple tasks with same sensors/effectors

# Example : TD Gammon



[Tesauro, 1995]

Learn to play Backgammon

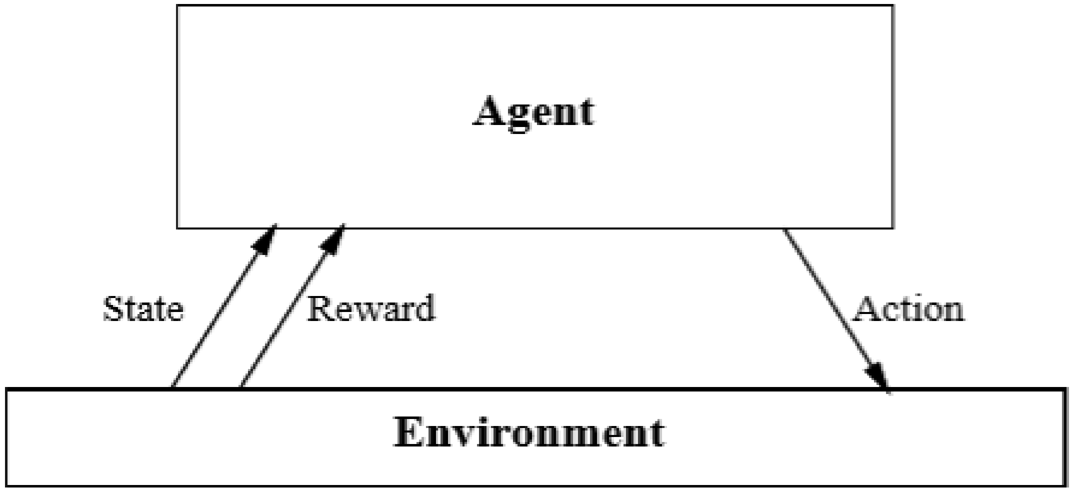
Immediate reward

- +100 if win
- -100 if lose
- 0 for all other states

Trained by playing 1.5 million games against itself

Now approximately equal to best human player

# Reinforcement Learning Problem



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

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \text{ , where } 0 \leq \gamma < 1$$

# Explanation

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.
- It can perform any of a set of possible actions **A**. Each time it performs an action **a**, in some state $\mathbf{s}_t$ the agent receives a real-valued reward **r**, that indicates the immediate value of this state-action transition. This produces a sequence of states $\mathbf{s}_i$, actions $\mathbf{a}_i$**,** and immediate rewards $\mathbf{r}_i$ as shown in the figure.
- The agent's task is to learn a control policy, **π : S -> A,** that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay

# The aspects which makes RL different from other

The  reinforcement learning problem differs from other function approximation tasks in several important respects

- **Delayed reward.**
- **Exploration**
- **Partially observable states**
- **Life-long learning**

# Delayed Reward

- The task of the agent is to learn a target function **π** that maps from the current state **s** to the optimal action **a = π**(s).
- Here training example is not of the form **(s, n(s)).**
- Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions.
- The agent, therefore, faces the problem of temporal credit assignment: ***determining which of the actions in its sequence are to be credited with producing the eventual rewards***.

# Exploration

- In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses.
- This raises the question of which experimentation strategy produces most effective learning.
- The learner faces a tradeoff in choosing whether to *favor exploration of unknown states and actions* (**to gather new information**), or *exploitation of states and actions that it has already learned will yield high reward* (**to maximize its cumulative reward**).

# Partially observable states

- Although it is convenient to assume that the agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information.
- For example, a robot with a forward-pointing camera cannot see what is behind it.
- In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

# Life-long learning

- Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors.
- For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers.
- This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

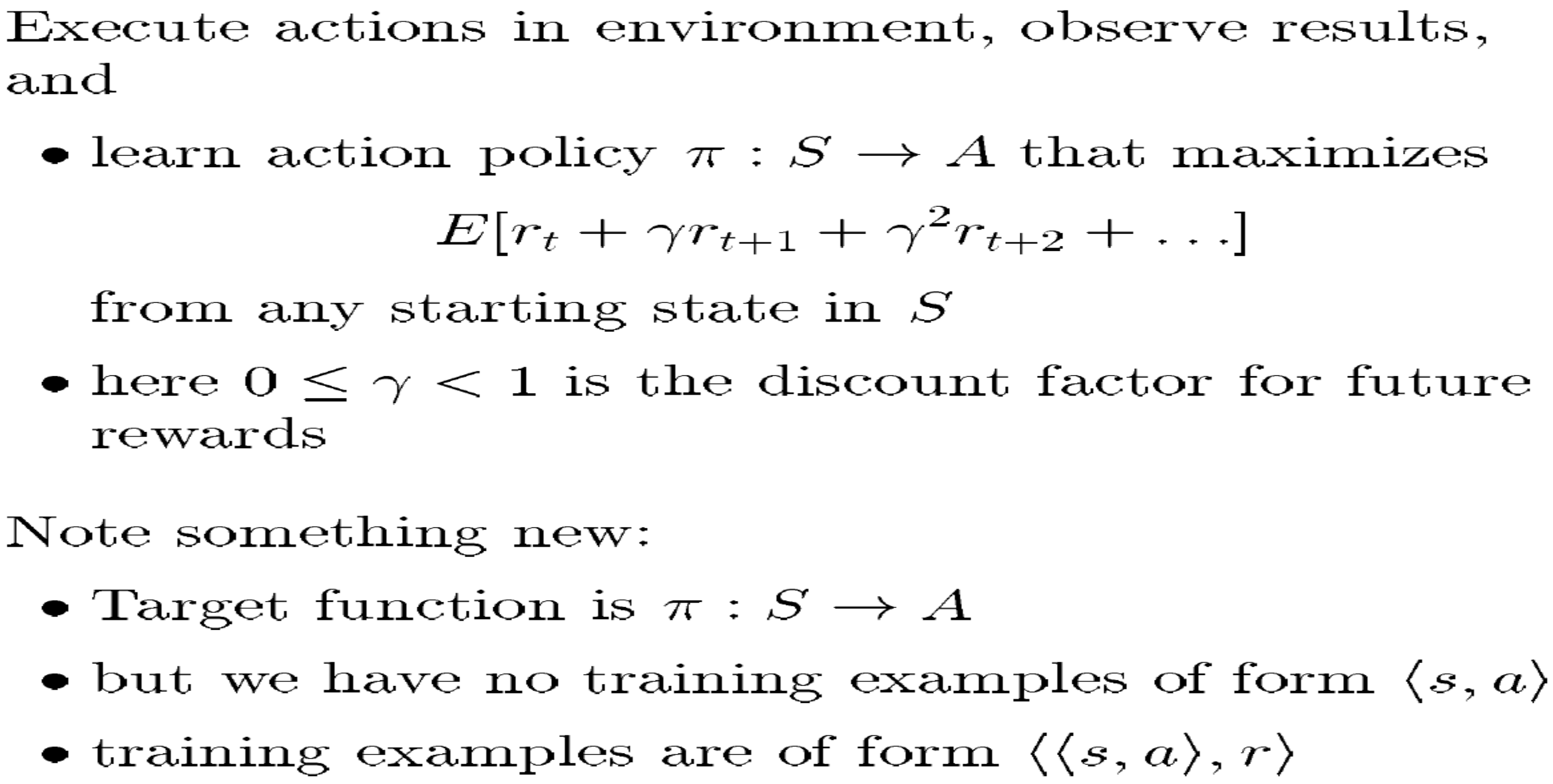
# THE LEARNING TASK

- In this section we formulate the problem of learning sequential control strategies more precisely. Note there are many ways to do so.
- For example, we might assume the agent's actions are deterministic or that they are nondeterministic.
- We might assume that the agent can predict the next state that will result from each action, or that it cannot.
- We might assume that the agent is trained by an expert who shows it examples of optimal action sequences, or that it must train itself by performing actions of its own choice.
- Here we define one quite general formulation of the problem, based on Markov decision processes.

# Markov Decision Process

- In a Markov decision process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state $s_t$, chooses a current action $a_t$, and performs it.
- The environment responds by giving the agent a reward $r_t = r(s_t, a)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a)$. Here the functions δ and r are part of the environment and are not necessarily known to the agent.
- In an MDP, the functions $\delta(s_t, a,)$ and $r(s_t, a)$ depend only on the current state and action, and not on earlier states or actions.
- Here we consider only the case in which S and A are finite. In general, δ and r may be nondeterministic functions, but we begin by considering only the *deterministic case*

# Agent's Learning Task



Execute actions in environment, observe results, and

- learn action policy $\pi : S \rightarrow A$ that maximizes

  $$E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$$

  from any starting state in $S$
- here $0 \leq \gamma < 1$ is the discount factor for future rewards

Note something new:

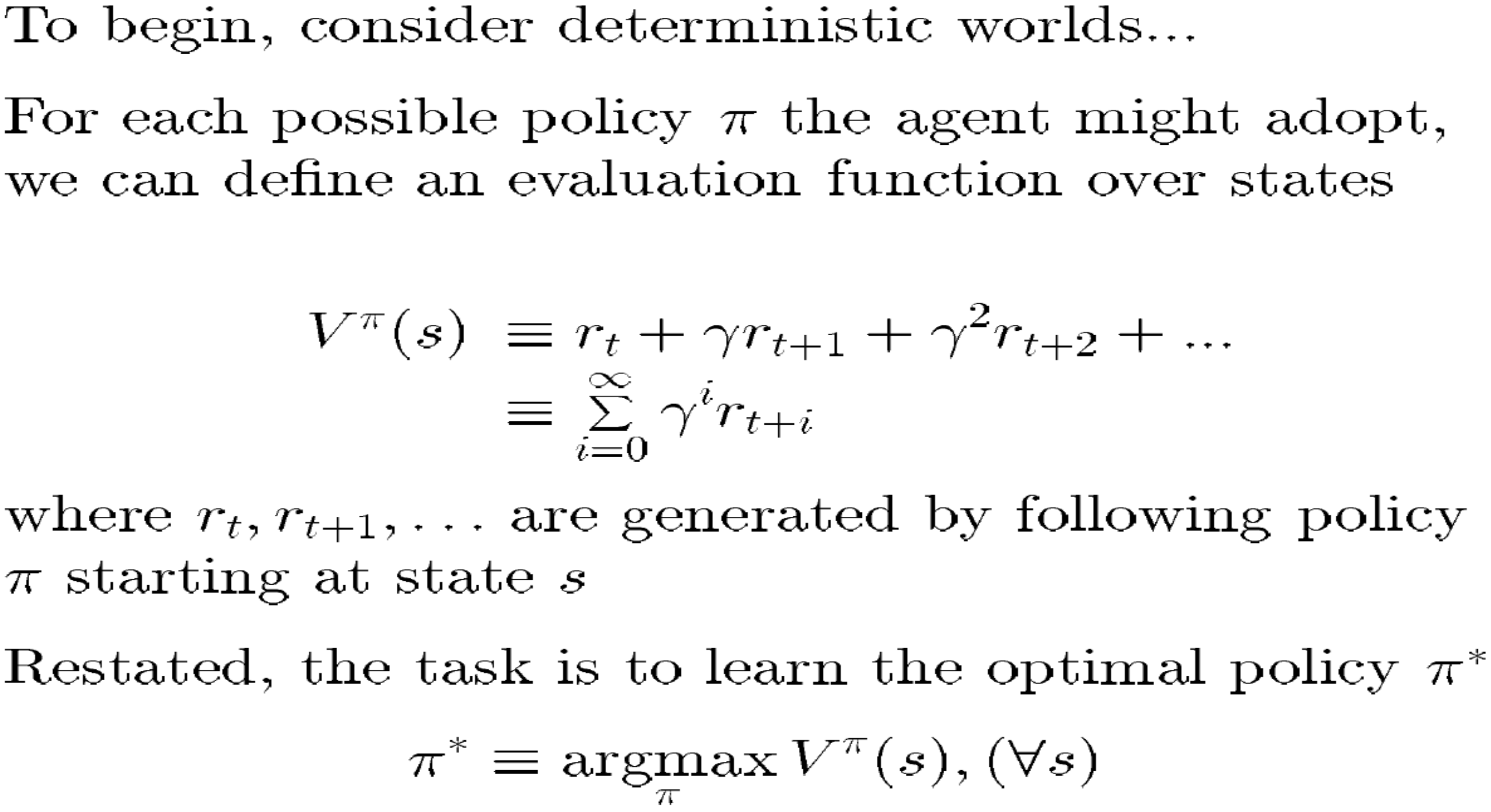
- Target function is $\pi : S \rightarrow A$
- but we have no training examples of form $\langle s, a \rangle$
- training examples are of form $\langle \langle s, a \rangle, r \rangle$

# Agents Learning Task (Value Function)

- The task of the agent is to learn a policy, **π : S -> A**, for selecting its next action **a**, based on the current observed state $s_t$; that is, **π(**$s_t$**) = a,**
- How shall we specify precisely which policy **π** we would like the agent to learn?
- One obvious approach is to require the policy that produces the greatest possible cumulative reward for the robot over time.
- To state this requirement more precisely, we define the cumulative value
- V$^{\pi}$ ($s_t$) achieved by following an arbitrary policy **n** from an arbitrary initial state $s_t$ as follows:

$$
\begin{aligned}
V^{\pi}(s_t) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\
&\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}
\end{aligned}
$$

# Value Function



To begin, consider deterministic worlds...

For each possible policy $\pi$ the agent might adopt, we can define an evaluation function over states

$$\begin{aligned} V^{\pi}(s) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

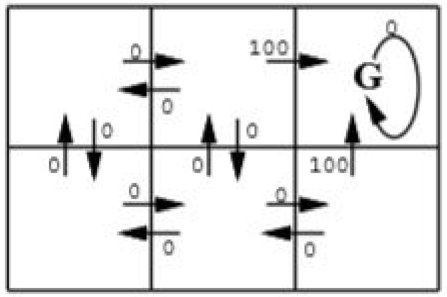
where $r_t, r_{t+1}, \ldots$ are generated by following policy $\pi$ starting at state $s$

Restated, the task is to learn the optimal policy $\pi^*$

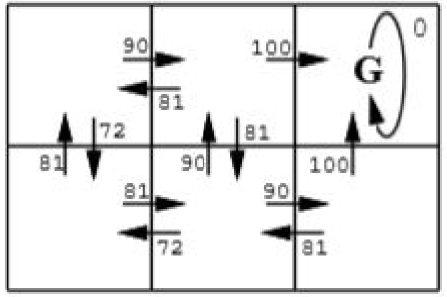
$$\pi^* \equiv \operatorname*{argmax}_{\pi} V^{\pi}(s), (\forall s)$$

The quantity V$^{\pi}$ $(s_t)$ is often called the *discounted cumulative reward achieved by policy* **π.**

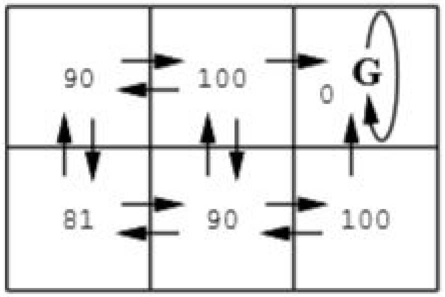
- The six grid squares in this diagram represent ***six possible states, or locations, for the agent***.
- Each arrow in the diagram represents a ***possible action the agent can take to move from one state to another***.
- The number associated *with each arrow represents the immediate reward* ***r(s,a)*** *the agent receives* if it executes the corresponding state-action transition.
- Note the immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into **the state labeled G.**
- It is convenient to think ***of the state G as the goal state,*** because the only way the agent can receive reward, in this case, ***is by entering this state.***
- Note in this particular environment, the only action available to the agent once it enters the ***state G is to remain in this state***.
- For this reason, we call **G** an absorbing state.



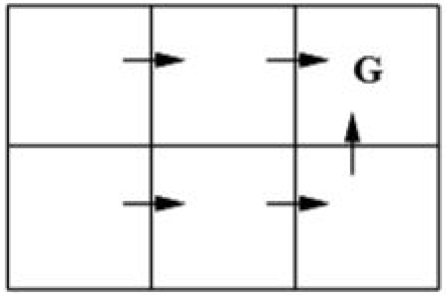
*r*(*s*, *a*) (immediate reward) values



*Q*(*s*, *a*) values

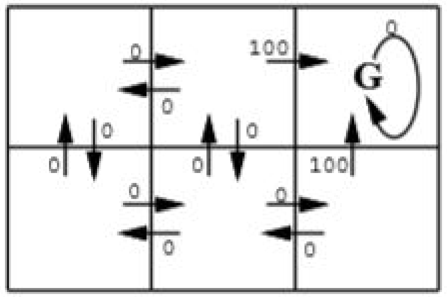


*V**(*s*) values

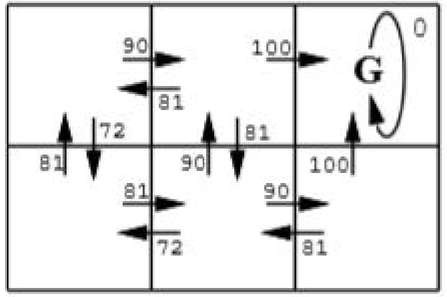


One optimal policy

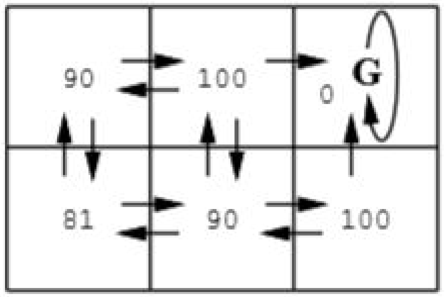
- A simple deterministic world to illustrate the basic concepts of ***Q-learning***. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, **r(s,a**) gives reward 100 for actions entering the goal state G, and zero otherwise.
- Values of V$^{\pi}$ $(s)$and Q(s, a) follow from r(s, a), and the discount factor Y=0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.



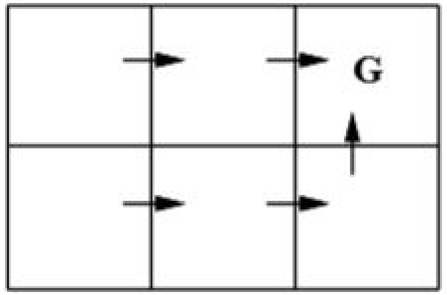
$r(s,a)$ (immediate reward) values



$Q(s,a)$ values

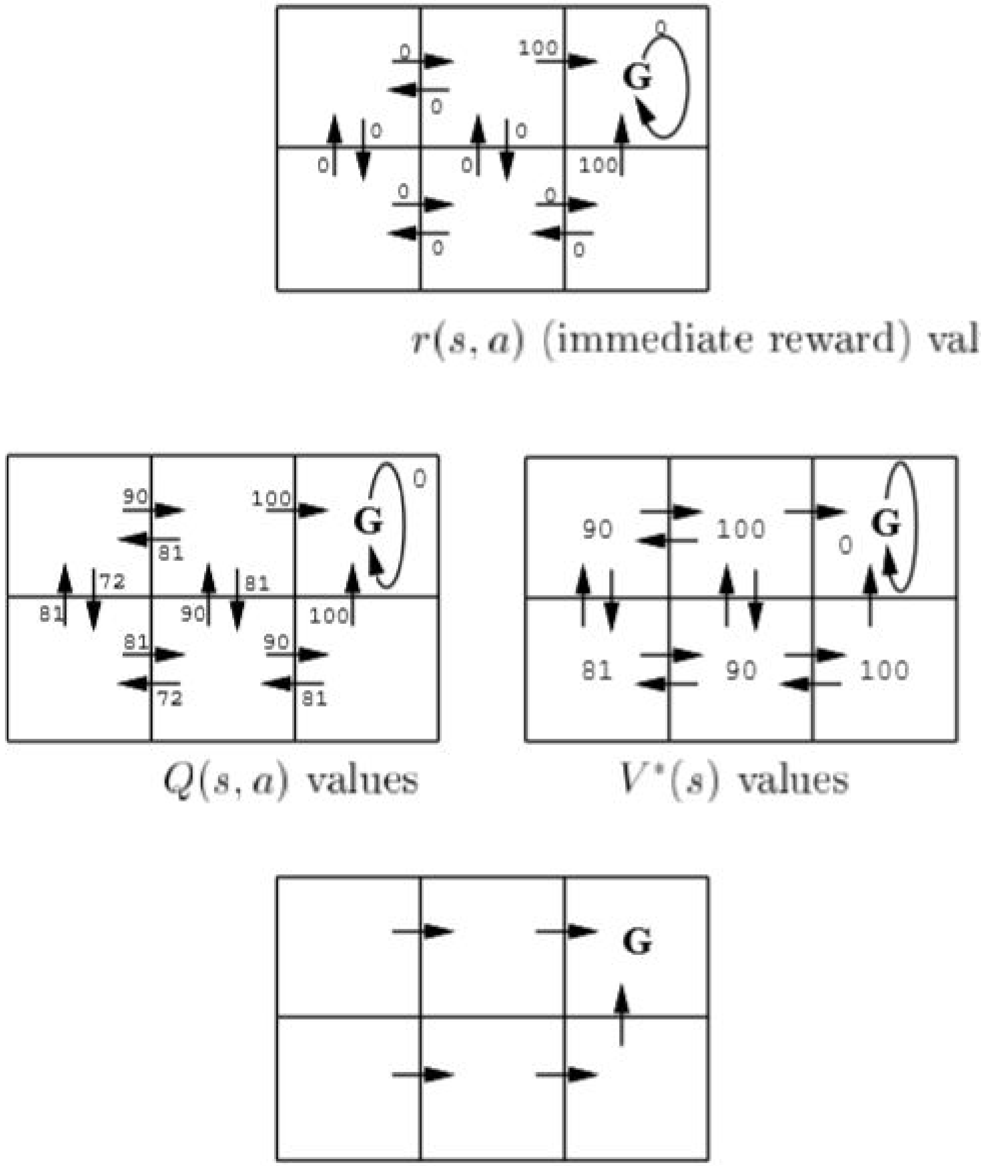


$V^*(s)$ values



One optimal policy

- The diagram at the right of Figure shows the values of V* for each state.
- For example, consider the bottom right state in this diagram. The value of V* for this state is 100 because the optimal policy in this state selects the "*move up*" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards. Similarly, the value of V* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is
- $0 + Y\ 100 + Y^{2}0 + Y^{3}\ 0 + ... = 90$



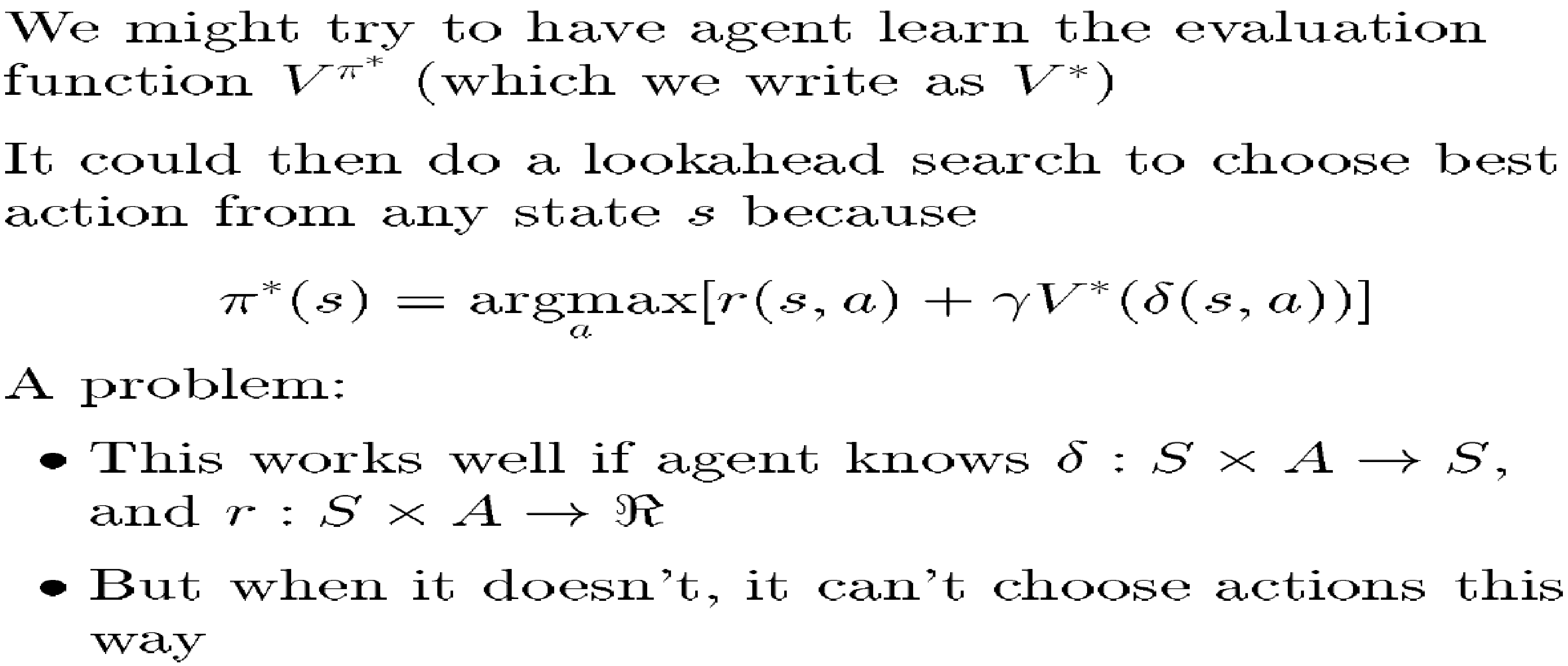
$r(s, a)$ (immediate reward) values

$Q(s, a)$ values

$V^{*}(s)$ values

One optimal policy

# What to Learn



We might try to have agent learn the evaluation function $V^{\pi^*}$ (which we write as $V^*$)

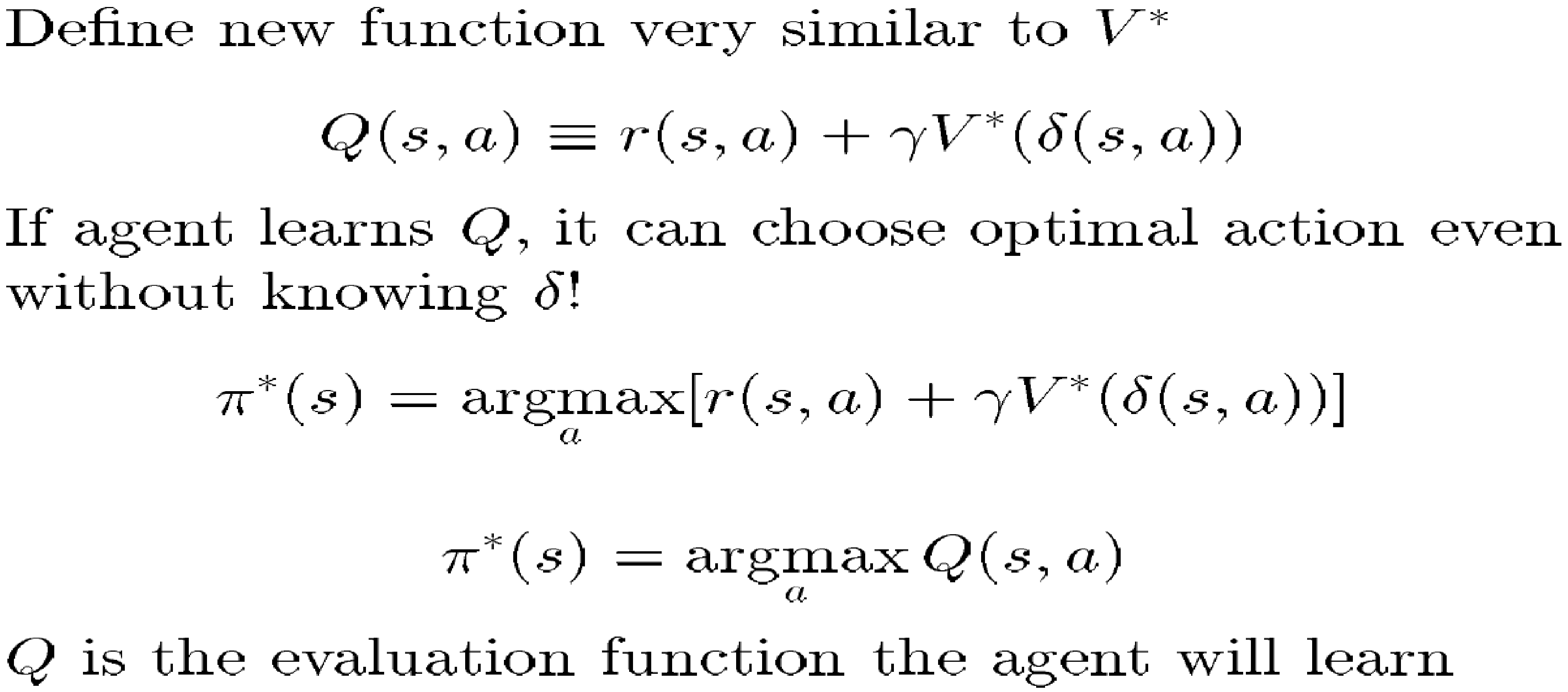
It could then do a lookahead search to choose best action from any state $s$ because

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))]$$

A problem:

- This works well if agent knows $\delta : S \times A \rightarrow S$, and $r : S \times A \rightarrow \Re$
- But when it doesn't, it can't choose actions this way

# *Q* Function



Define new function very similar to $V^*$

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

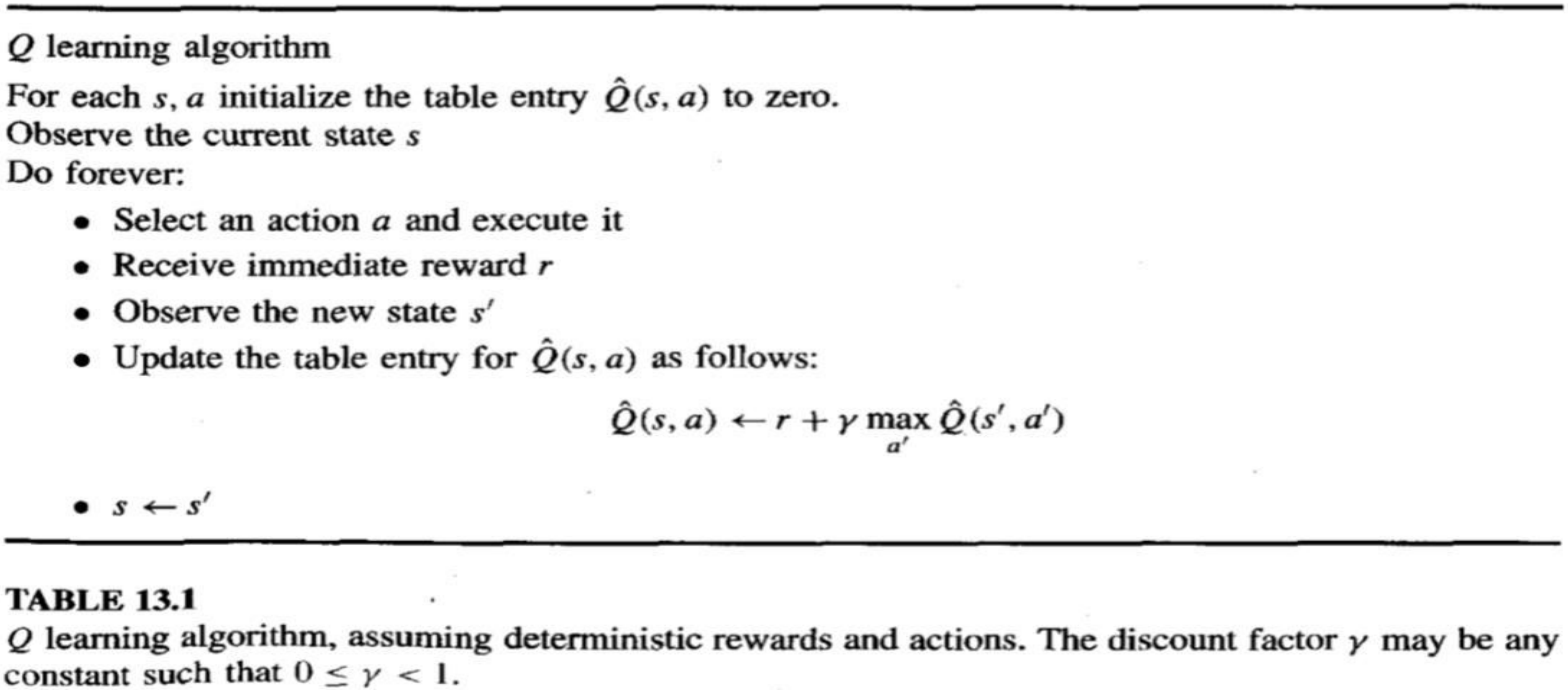
If agent learns $Q$, it can choose optimal action even without knowing $\delta$!

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a)$$

$Q$ is the evaluation function the agent will learn

# An Algorithm for Learning Q

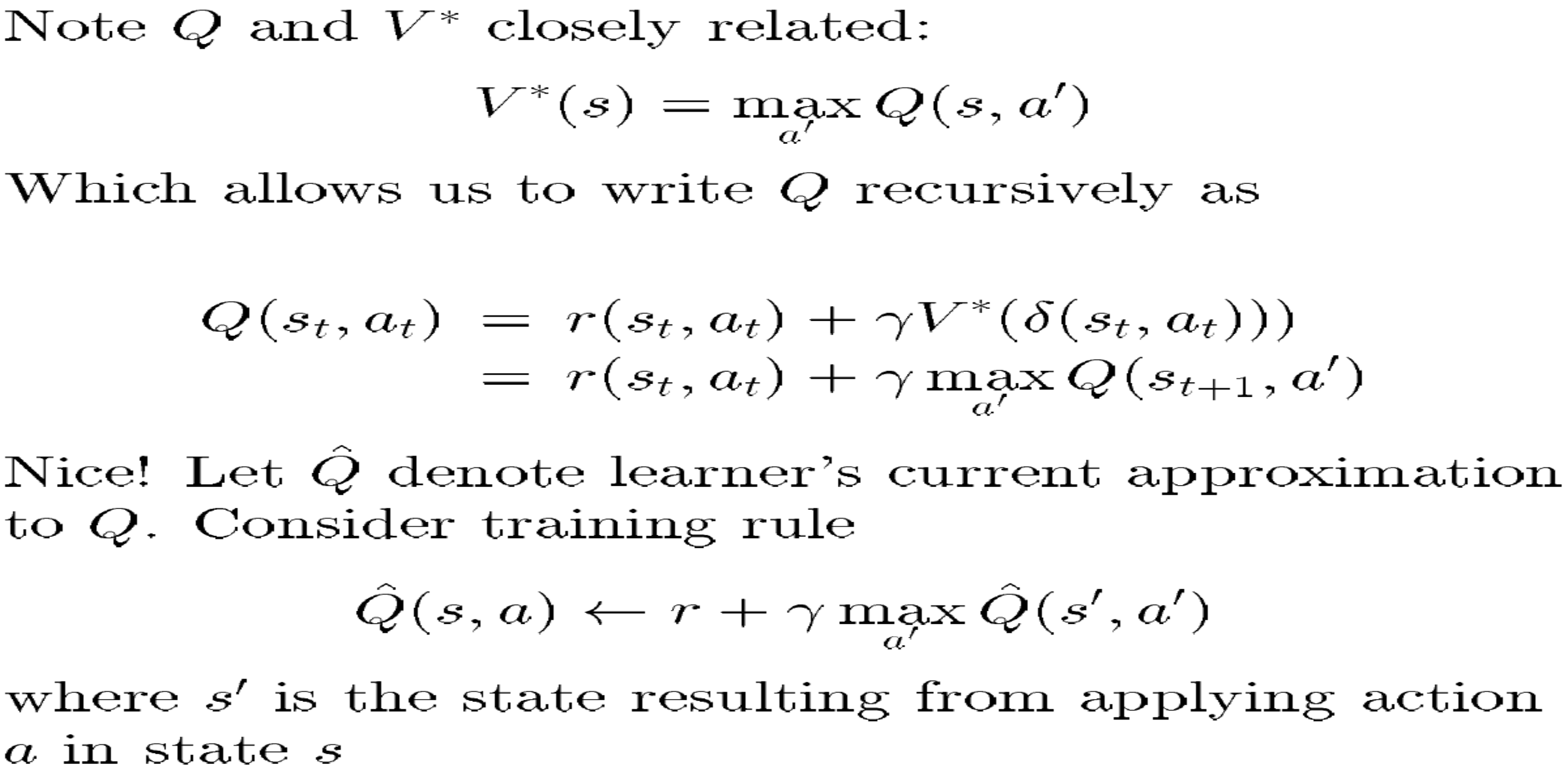


*Q* learning algorithm

```
For each $s, a$ initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state $s$
Do forever:
    • Select an action $a$ and execute it
    • Receive immediate reward $r$
    • Observe the new state $s'$
    • Update the table entry for $\hat{Q}(s, a)$ as follows:
          $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
    • $s \leftarrow s'$
```

**TABLE 13.1**
*Q* learning algorithm, assuming deterministic rewards and actions. The discount factor $\gamma$ may be any constant such that $0 \leq \gamma < 1$.

# Training Rule to Learn *Q*



Note $Q$ and $V^*$ closely related:

$$V^*(s) = \max_{a'} Q(s, a')$$

Which allows us to write $Q$ recursively as

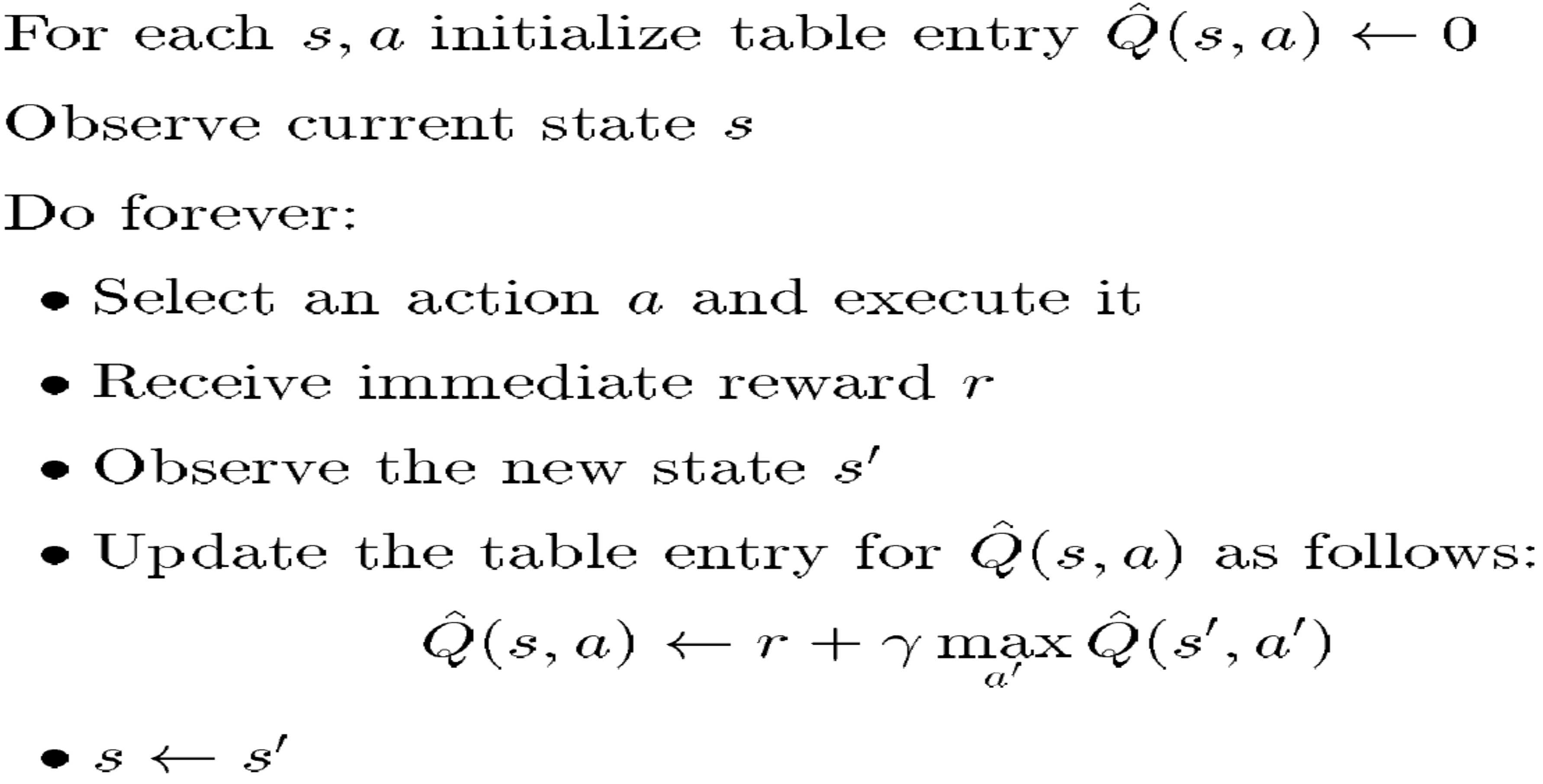
$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t))) \\ &= r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

Nice! Let $\hat{Q}$ denote learner's current approximation to $Q$. Consider training rule

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

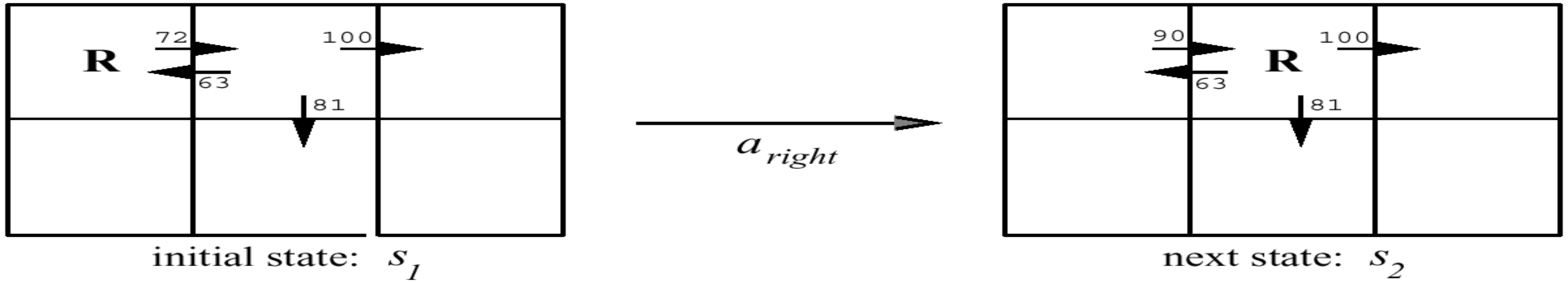
where $s'$ is the state resulting from applying action $a$ in state $s$

# Q Learning for Deterministic Worlds



```
For each $s, a$ initialize table entry $\hat{Q}(s,a) \leftarrow 0$
Observe current state $s$
Do forever:
  • Select an action $a$ and execute it
  • Receive immediate reward $r$
  • Observe the new state $s'$
  • Update the table entry for $\hat{Q}(s,a)$ as follows:
        $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$
  • $s \leftarrow s'$
```

# Updating $\hat{Q}$



$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{63, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

notice if rewards non-negative, then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s,a) \geq \hat{Q}_n(s,a)$$

and

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$$

$\hat{Q}$ converges to $Q$. Consider case of deterministic world where see each $\langle s, a \rangle$ visited infinitely often.

*Proof*: Define a full interval to be an interval during which each $\langle s, a \rangle$ is visited. During each full interval the largest error in $\hat{Q}$ table is reduced by factor of $\gamma$

Let $\hat{Q}_n$ be table after $n$ updates, and $\Delta_n$ be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n = \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)|$$

For any table entry $\hat{Q}_n(s,a)$ updated on iteration $n+1$, the error in the revised estimate $\hat{Q}_{n+1}(s,a)$ is

$$\begin{aligned}
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) \\
&\qquad -(r + \gamma \max_{a'} Q(s',a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s',a') - Q(s',a')| \\
&\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - Q(s'',a')| \\
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &\leq \gamma \Delta_n
\end{aligned}$$

# Nondeterministic Case(Cont')

$Q$ learning generalizes to nondeterministic worlds

Alter training rule to

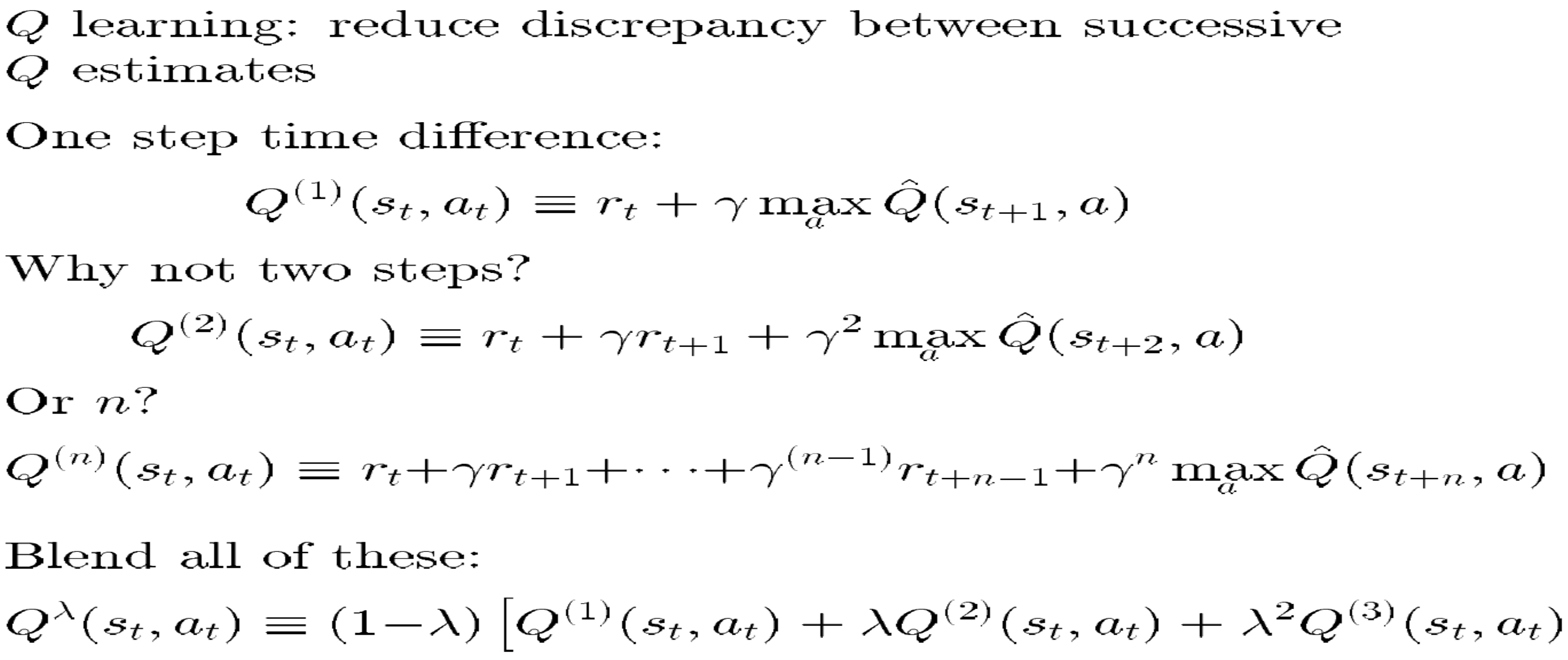
$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a)+\alpha_n[r+\max_{a'}\hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1+visits_n(s,a)}$$

Can still prove convergence of $\hat{Q}$ to $Q$ [Watkins and Dayan, 1992]

# Temporal Difference Learning



$Q$ learning: reduce discrepancy between successive $Q$ estimates

One step time difference:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

Why not two steps?

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

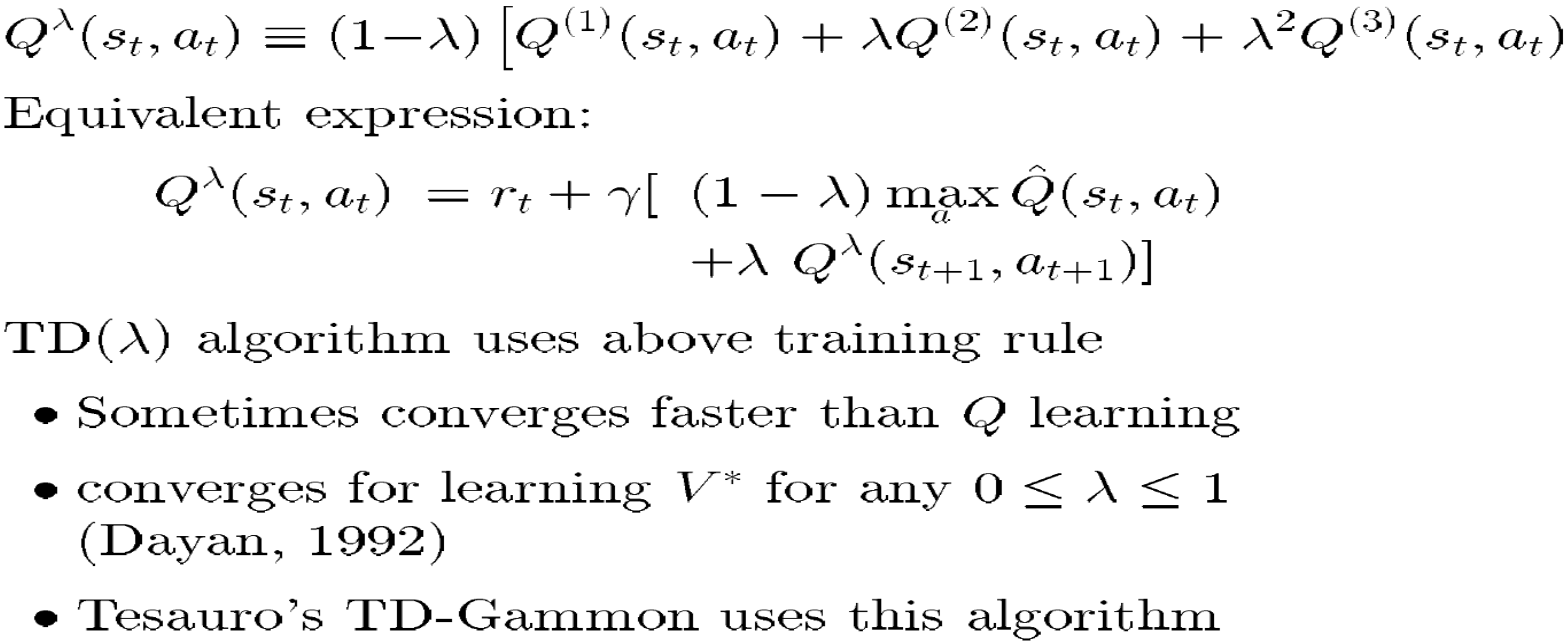
Or $n$?

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Blend all of these:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda) \Big[ Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t)$$

# Temporal Difference Learning(Cont’)



$$Q^{\lambda}(s_t, a_t) \equiv (1{-}\lambda) \Big[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t)$$

Equivalent expression:

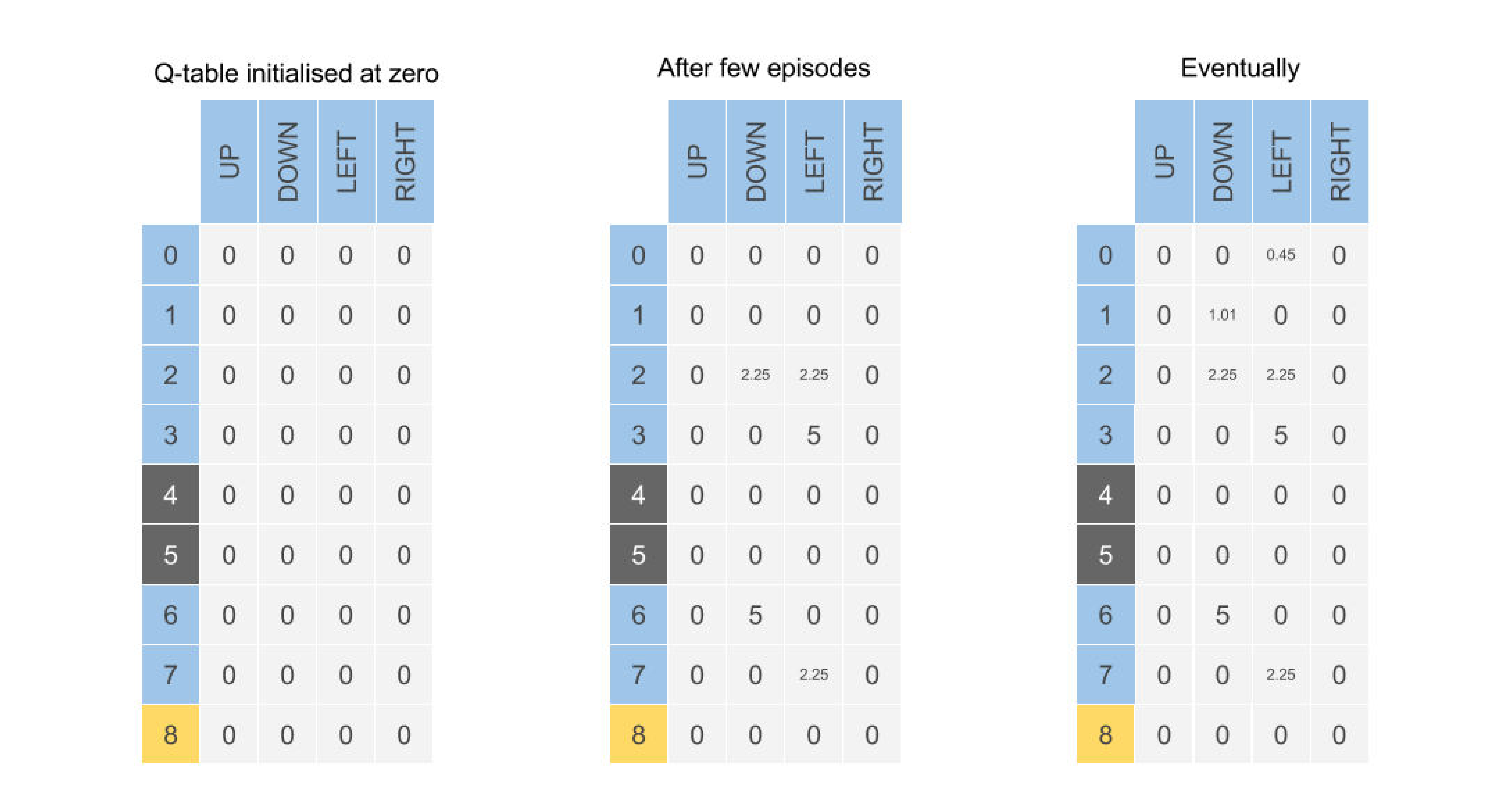
$$\begin{aligned} Q^{\lambda}(s_t, a_t) \ &= r_t + \gamma[\ \ (1 - \lambda) \max_a \hat{Q}(s_t, a_t) \\ &\qquad\qquad +\lambda \ Q^{\lambda}(s_{t+1}, a_{t+1})] \end{aligned}$$

TD($\lambda$) algorithm uses above training rule

- Sometimes converges faster than $Q$ learning
- converges for learning $V^*$ for any $0 \leq \lambda \leq 1$ (Dayan, 1992)
- Tesauro’s TD-Gammon uses this algorithm

# Subtleties and Ongoing Research

- Replace $\hat{Q}$ table with neural net or other generalizer
- Handle case where state only partially observable
- Design optimal exploration strategies
- Extend to continuous action, state
- Learn and use $\hat{\delta} : S \times A \rightarrow S$
- Relationship to dynamic programming



### Q-table initialised at zero

| | UP | DOWN | LEFT | RIGHT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 |

### After few episodes

| | UP | DOWN | LEFT | RIGHT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 2.25 | 2.25 | 0 |
| 3 | 0 | 0 | 5 | 0 |
| 4 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0 | 5 | 0 | 0 |
| 7 | 0 | 0 | 2.25 | 0 |
| 8 | 0 | 0 | 0 | 0 |

### Eventually

| | UP | DOWN | LEFT | RIGHT |
|---|---|---|---|---|
| 0 | 0 | 0 | 0.45 | 0 |
| 1 | 0 | 1.01 | 0 | 0 |
| 2 | 0 | 2.25 | 2.25 | 0 |
| 3 | 0 | 0 | 5 | 0 |
| 4 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0 | 5 | 0 | 0 |
| 7 | 0 | 0 | 2.25 | 0 |
| 8 | 0 | 0 | 0 | 0 |