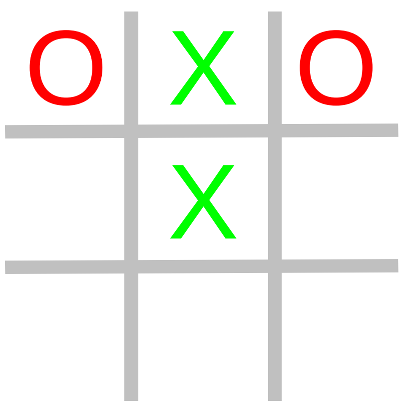
# Reinforcement Learning

- Ziel:
  - Lernen von Bewertungsfunktionen durch Feedback (Reinforcement) der Umwelt (z.B. Spiel gewonnen/verloren).
- Anwendungen:
  - **Spiele:**
    - Tic-Tac-Toe: MENACE (Michie 1963)
    - Backgammon: TD-Gammon (Tesauro 1995)
    - Schach: KnightCap (Baxter et al. 2000)
  - **Andere:**
    - Elevator Dispatching
    - Robot Control
    - Job-Shop Scheduling

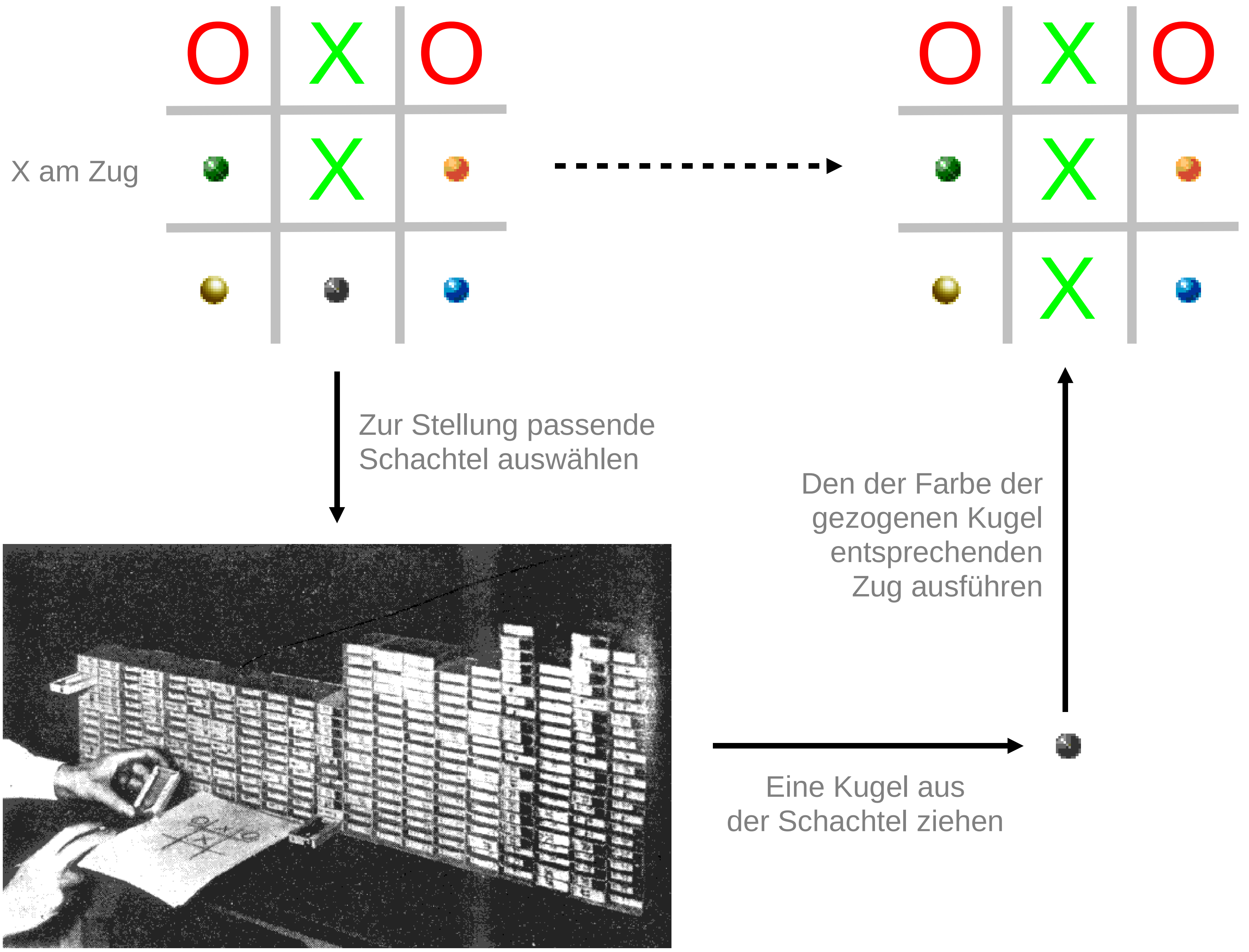
# MENACE (Michie, 1963)

- Lernt Tic-Tac-Toe zu spielen




- Hardware:
  - 287 Zündholzschachteln
    (1 für jede Stellung)
  - Perlen in 9 verschiedenen Farbe
    (1 Farbe für jedes Feld)

- Spiel-Algorithmus:
  - Wähle Zündholzschachtel, die der Stellung entspricht
  - Ziehe zufällig eine der Perlen
  - Ziehe auf das Feld, das der Farbe der Perle entspricht


X am Zug
Zur Stellung passende
Schachtel auswählen
Den der Farbe der
gezogenen Kugel
entsprechenden
Zug ausführen
Eine Kugel aus
der Schachtel ziehen

# Reinforcement Learning in MENACE

- Lern-Algorithmus:
  - Spiel verloren → gezogene Perlen werden einbehalten (*negative reinforcement)*
  - Spiel gewonnen → eine Perle der gezogenen Farbe wird in verwendeten Schachteln hinzugefügt (*positive reinforcement*)
  - Spiel remis → Perlen werden zurückgelegt (keine Änderung)
- führt zu
  - Erhöhung der Wahrscheinlichkeit, daß ein erfolgreicher Zug wiederholt wird
  - Senkung der Wahrscheinlichkeit, daß ein nicht erfolgreicher Zug wiederholt wird
- Credit Assignment Problem
  - Man weiß nicht, welcher Zug den Gewinn oder Verlust verursacht hat
  - Durch zahlreiche Spiele konvergiert obiges Verfahren jedoch

# Reinforcement Learning - Formalization

- Learning Scenario
  - a learning agent
  - $S$: a set of possible states
  - $A$: a set of possible actions
  - a state transition function $\delta: S \times A \rightarrow S$
  - a reward runction $r: S \times A \rightarrow \mathbb{R}$
- Enviroment:
  - the agent repeatedly chooses an action according to some *policy* $\pi: S \rightarrow A$
  - this will put the agent into a new state according to $\delta$
  - in some states the agent receives feedback from the environment (reinforcement)
- Markov property
  - rewards and state transitions only depend on last state

# MENACE - Formalization

- Framework
  - states = matchboxes
  - actions = moves/beads
  - policy = prefer actions with higher number of beads
  - reward = game won/ game lost
    - *delayed* reward: we don't know right away whether a move was good or bad

# Learning Task

find a policy that maximizes the cumulative reward

- delayed reward
  - reward for actions may not come immediately (e.g., game playing)
- therefore maximize cumulative reward $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$
  - reward from "now" until the end of time
  - immediate rewards are weighted higher (*discount factor* $\gamma$)
- training examples
  - generated by interacting with the environment (trial and error)
  - Note:
    - training examples are not supervised $(s, a_{max})$
    - training examples are of the form $(s, a, r)$

# Value Function

- Each policy can be assigned a value
  - the cumulative expected reward that the agent gets when s/he follows that policy

$$s_{t+1}=\delta(s_t,a_t)$$

$$V^{\pi}(s_t)=\sum_{i=0}^{\infty}\gamma^i r_{t+i}=r_t+\gamma r_{t+1}+\gamma^2 r_{t+2}\ldots=$$
$$=r_t+\gamma(r_{t+1}+\gamma r_{t+2}+\ldots)=r(s_t,a_t)+\gamma V^{\pi}(\delta(s_t,a_t))$$

- Optimal policy
  - the policy with the highest expected value for all states s

$$\pi^*=arg\,max_{\pi} V^{\pi}(s)$$

  - learning an optimal value function $V^*(s)$ yields an optimal policy

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

  - BUT: using the optimal value function for action selection requires knowledge of $r$ and $\delta$

# Q-function

- the Q-function does not evaluate states, but evaluates state-action pairs
  - the Q-function is the cumulative reward for starting in $s$, applying action $a$, and, in the resulting state $s'$, play optimally

$$Q(s,a) := r(s,a) + \gamma V^*(s') \qquad [s' = \delta(s,a)]$$

  → the optimal value function is the maximal Q-function over all possible actions in a state $V^*(s) = max_a Q(s,a)$

- *Bellman equation*: $\boxed{Q(s,a) = r(s,a) + \gamma\, max_{a'} Q(s',a')}$
  - the value of the Q-function for the current state $s$ and an action $a$ is the same as the sum of
    - the reward in the current state $s$ for the chosen action $a$
    - the (discounted) value of the Q-function for the best action that I can play in the successor state $s'$

# Learning the Q-function

- Basic strategy:
  - start with a some function $\hat{Q}$, and update it after each step
  - in MENACE: $\hat{Q}$ returns for each box $s$ and each action $a$ the number of beads in the box
- update rule:
  - the Bellman equation will in general not hold for $\hat{Q}$
    i.e., the left side and the right side will be different

  $\rightarrow$ new value of $\hat{Q}(s,a)$ is a weighted sum of both sides
  - weighted by a learning rate $\alpha$

$$\hat{Q}(s,a) \leftarrow (1-\alpha)\hat{Q}(s,a)+\alpha(r(s,a)+\gamma\, max_{a'}\hat{Q}(s',a'))$$
$$\leftarrow \hat{Q}(s,a)+\alpha[r(s,a)+\gamma\, max_{a'}\hat{Q}(s',a')-\hat{Q}(s,a)]$$

new Q-value for state $s$ and action $a$

old Q-value for this state/action pair

predicted Q-value for state $s'$ and action $a'$

# Q-learning (Watkins, 1989)

1. initialize all *Q(s,a)* with 0
2. observe current state *s*
3. loop
   1. select an action *a* and execute it
   2. receive the immediate reward and observe the new state *s'*
   3. update the table entry

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[\underline{\left(r(s,a) + \gamma \max_{a'} Q(s',a')\right)} - \underline{Q(s,a)}\right]$$

   4. $s = s'$

Temporal Difference:
Difference between the estimate of the value of an action before and after performing the action.
→ Temporal Difference Learning

# Miscellaneous

- Weight Decay:
  - $\alpha$ decreases over time, e.g. $\alpha = \frac{1}{1 + visits(s,a)}$
- Convergence:
  it can be shown that Q-learning converges
  - if every state/action pair is visited infinitely often
  - not very realistic for large state/action spaces
- Representation
  - in the simplest case, $\hat{Q}(s,a)$ is realized with a look-up table with one entry for each state/action pair
  - a better idea would be to have trainable function, so that experience in some part of the space can be generalized
  - special training algorithms for, e.g., neural networks exist

# SARSA

- performs *on-policy updates*
  - update rule assumes action a' is chosen according to current policy

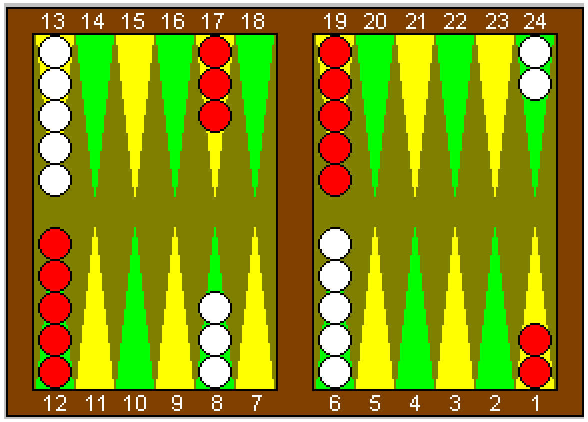
$$Q(s,a) \leftarrow Q(s,a) + \alpha [r(s,a) + \gamma Q(s',a') - Q(s,a)]$$

  - convergence if the policy gradually moves towards a policy that is greedy with respect to the current Q-function

- $\varepsilon$-greedy policies
  - choose random action with probability $\varepsilon$, otherwise greedy
  - trade off exploration vs. exploitation
    - exploration is necessary to get a wide variety of state action pairs
    - exploitation is necessary for convergence

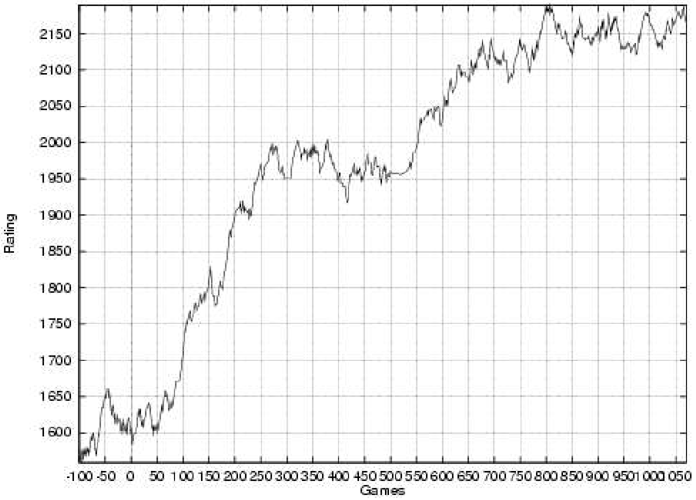
# TD-Gammon (Tesauro, 1995)

- weltmeisterliches Backgammon-Programm
  - Entwicklung von Anfänger zu einem weltmeisterlichen Spieler nach 1,500,000 Trainings-Spiele gegen sich selbst (!)
  - Verlor 1998 WM-Kampf über 100 Spiele knapp mit 8 Punkten
  - Führte zu Veränderungen in der Backgammon-Theorie und ist ein beliebter Trainings- und Analyse-Partner der Spitzenspieler




- Verbesserungen gegenüber MENACE:
  - Schnellere Konvergenz durch Temporal-Difference Learning
  - Neurales Netz statt Schachteln und Perlen erlaubt Generalisierung
  - Verwendung von Stellungsmerkmalen als Features

# KnightCap (Baxter et al. 2000)




- Lernt meisterlich Schach zu spielen
  - Verbesserung von 1650 Elo (Anfänger) auf 2150 Elo (guter Club-Spieler) in nur ca. 1000 Spielen am Internet
- Verbesserungen gegenüber TD-Gammon:
  - Integration von TD-learning mit den tiefen Suchen, die für Schach erforderlich sind
  - Training durch Spielen gegen sich selbst -> Training durch Spielen am Internet

# Reinforcement Learning Resources

- Book
  - On-line Textbook on Reinforcement learning
    - http://www.cs.ualberta.ca/~sutton/book/the-book.html
- Demos
  - Grid world
    - http://thierry.masson.free.fr/IA/en/qlearning_applet.htm
  - Robot learns to crawl
    - http://iridia.ulb.ac.be/~fvandenb/qlearning/qlearning.html
  - Pole Balancing Problem
    - http://www.bovine.net/~jlawson/hmc/pole/sane.html
- Reinforcement Learning Repository
  - tutorial articles, applications, more demos, etc.
    - http://www-anw.cs.umass.edu/rlr/