# 1 Reinforcement Learning and AlphaZero

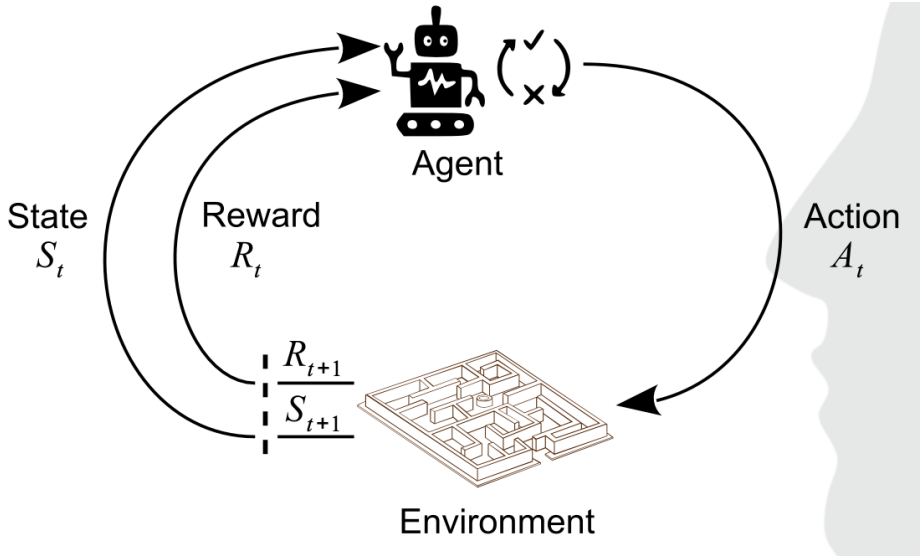
## 1.1 Grundlagen des Reinforcement Learning

Reinforcement Learning (RL) ist ein Ansatz des maschinellen Lernens, bei dem ein **Agent** durch Interaktion mit einer **Umgebung** lernt, eine Aufgabe zu lösen. Im Gegensatz zum Supervised Learning erhält der Agent keine korrekten Input-Output-Paare, sondern muss durch Ausprobieren (Trial-and-Error) und Feedback in Form von **Rewards** (Belohnungen) lernen.

**Der RL-Zyklus**

Der Agent und die Umgebung interagieren in diskreten Zeitschritten $t$:

1. Der Agent beobachtet den aktuellen Zustand (State) $S_t$.
2. Basierend auf $S_t$ wählt der Agent eine Aktion (Action) $A_t$.
3. Die Umgebung geht in einen neuen Zustand $S_{t+1}$ über und gibt einen Reward $R_{t+1}$ zurück.
4. Das Ziel des Agenten ist die Maximierung der kumulativen Belohnung über die Zeit (Return).




## 1.2 Markov Decision Processes (MDP)

Um RL-Probleme mathematisch zu formalisieren, werden **Markov Decision Processes** (MDPs) verwendet.

**Definition MDP**

Ein MDP ist ein Tupel $(S, A, T, R, \gamma)$:

- $S$: Menge aller möglichen Zustände.
- $A$: Menge aller möglichen Aktionen.
- $T$: Die **Transition Function** (oder das Modell) $T(s, a, s') = P(s'|s, a)$. Sie gibt die Wahrscheinlichkeit an, von Zustand $s$ mit Aktion $a$ in den Zustand $s'$ zu gelangen.
- $R$: Die **Reward Function** $R(s, a, s')$. Die Belohnung für den Übergang von $s$ nach $s'$ mittels $a$.
- $\gamma$: Der **Discount Factor** (Diskontierungsfaktor) mit $0 \leq \gamma \leq 1$.

### 1.2.1 Markov-Eigenschaft

Ein Prozess ist **Markovian**, wenn der zukünftige Zustand nur vom aktuellen Zustand und der gewählten Aktion abhängt, nicht aber von der Vergangenheit (Gedächtnislosigkeit):

$$P(S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, \dots) = P(S_{t+1}|S_t, A_t)$$

### 1.2.2 Credit Assignment Problem

Eine zentrale Herausforderung im RL ist das **Temporal Credit Assignment Problem**: Da Belohnungen oft verzögert auftreten (z.B. erst am Ende eines Spiels), muss der Algorithmus herausfinden, welche vergangenen Aktionen für den Erfolg (oder Misserfolg) verantwortlich waren.

### 1.2.3 Discounted Rewards

Um unendliche Horizonte zu handhaben und sofortige Belohnungen gegenüber späteren zu bevorzugen, werden zukünftige Rewards mit $\gamma$ diskontiert. Der **Return** $G_t$ ist die Summe der diskontierten Rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

## 1.3 Optimalität und Value Functions

Das Ziel ist es, eine **Policy** (Strategie) $\pi$ zu finden, die den erwarteten Return maximiert.

- **Deterministische Policy:** $a = \pi(s)$
- **Stochastische Policy:** $\pi(a|s) = P(a|s)$

**Value Functions**

- **State-Value Function** $V^\pi(s)$**:** Der erwartete Return, wenn man in Zustand $s$ startet und danach der Policy $\pi$ folgt.
- **Action-Value Function** $Q^\pi(s, a)$**:** Der erwartete Return, wenn man in Zustand $s$ startet, Aktion $a$ wählt und danach der Policy $\pi$ folgt.

### 1.3.1 Bellman-Gleichungen

Die Werte von Zuständen lassen sich rekursiv durch die Bellman-Gleichung beschreiben. Für die optimale Value Function $V^*$ gilt:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

Dies bedeutet: Der Wert eines Zustands ist die bestmögliche Aktion, die zum sofortigen Reward plus dem diskontierten Wert des Folgezustands führt.

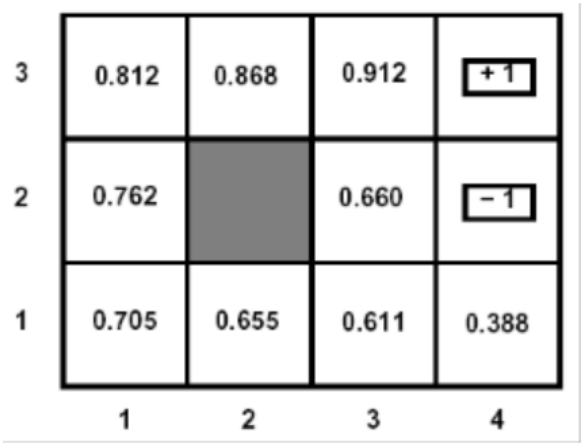
## 1.4 Lösen von MDPs: Value Iteration

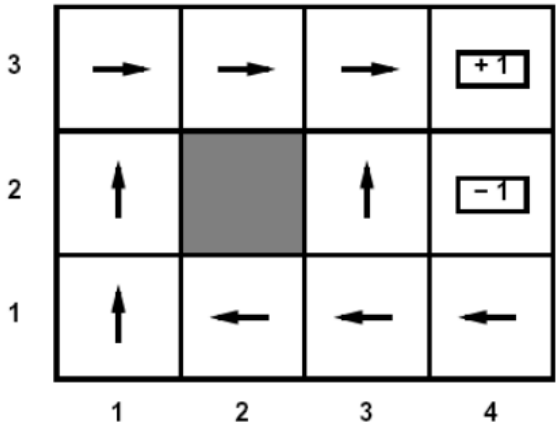
Wenn das Modell ($T$ und $R$) bekannt ist, spricht man von **Planning** (nicht Lernen). Ein klassischer Algorithmus hierfür ist die **Value Iteration** (Dynamische Programmierung).

**Value Iteration Algorithmus**

1. Initialisiere $V_0(s) = 0$ für alle Zustände.
2. Wiederhole bis zur Konvergenz (Update-Regel):

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R$$




## 1.5 Reinforcement Learning Arten

Wenn das Modell $(T, R)$ *nicht* bekannt ist, muss der Agent durch Interaktion lernen.

### 1.5.1 Model-Based vs. Model-Free

- **Model-Based:** Der Agent lernt zunächst ein Modell der Umgebung ($T$ und $R$ schätzen durch Zählen von Übergängen) und nutzt dieses dann zur Planung (z.B. mittels Value Iteration).
- **Model-Free:** Der Agent lernt direkt die Value Function oder die Policy, ohne die Übergangswahrscheinlichkeiten explizit zu modellieren.

### 1.5.2 Passive vs. Active Learning

- **Passive Learning:** Der Agent folgt einer festen Policy $\pi$ und lernt dabei die Nutzenwerte $V^{\pi}(s)$ (Policy Evaluation).
- **Active Learning:** Der Agent muss selbst entscheiden, welche Aktionen er wählt, um die optimale Policy zu finden. Hier entsteht das **Exploration vs. Exploitation** Dilemma.

## 1.6 Model-Free Learning Methoden

### 1.6.1 Temporal-Difference (TD) Learning

TD-Learning ist eine Methode für *Passive Learning*. Anstatt bis zum Ende einer Episode zu warten (wie bei Monte Carlo), wird die Schätzung $V(s)$ nach jedem Schritt basierend auf dem beobachteten Reward und der Schätzung des

nächsten Zustands $V(s')$ aktualisiert.Update-Regel (Running Average):

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha \underbrace{[R(s,a,s') + \gamma V(s')]}_{\text{Sample}}$$

Oder umgeformt:

$$V(s) \leftarrow V(s) + \alpha(\underbrace{R + \gamma V(s') - V(s)}_{\text{TD Error}})$$

Dabei ist $\alpha$ die Lernrate (Learning Rate).

### 1.6.2 Q-Learning

Q-Learning ist eine *Active Learning* Methode. Da wir kein Modell $T$ haben, können wir aus $V(s)$ allein keine Policy ableiten. Daher lernen wir direkt $Q$-Werte (Q-Table).Q-Learning ist ein **Off-Policy** Algorithmus: Der Agent lernt den Wert der optimalen Policy, auch wenn er sich beim Sammeln der Daten anders (z.B. explorativ) verhält.

**Q-Learning Update**

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

Der Term $\max_{a'} Q(s',a')$ impliziert, dass wir vom Folgezustand aus die beste Aktion annehmen, unabhängig davon, welche Aktion der Agent tatsächlich als nächstes ausführt.

### 1.6.3 Exploration vs. Exploitation

Damit Q-Learning konvergiert, muss der Agent alle Zustands-Aktions-Paare ausreichend oft besuchen.

- **Exploitation:** Wähle die Aktion mit dem höchsten bekannten Q-Wert (Greedy).
- **Exploration:** Wähle eine zufällige Aktion, um neues Wissen zu sammeln.
- **$\epsilon$-Greedy:** Mit Wahrscheinlichkeit $\epsilon$ wähle eine zufällige Aktion, sonst die beste bekannte $(1-\epsilon)$. $\epsilon$ wird oft über die Zeit verringert.

## 1.7 Deep Reinforcement Learning

Klassisches Q-Learning speichert Werte in einer Tabelle. Bei komplexen Problemen (z.B. Bildschirminput) ist der Zustandsraum zu groß.

- **Deep Q-Networks (DQN):** Ein neuronales Netz approximiert die Q-Funktion: $Q(s,a;\theta) \approx Q^*(s,a)$.
- **Policy Search / Policy Gradients:** Anstatt Value-Funktionen zu lernen, wird die Policy $\pi_\theta(a|s)$ direkt parametrisiert und mittels Gradientenaufstieg auf den erwarteten Reward optimiert (z.B. REINFORCE).

## 1.8 AlphaZero

AlphaZero kombiniert **Monte-Carlo Tree Search (MCTS)** mit tiefen neuronalen Netzen und Self-Play. Es benötigt kein bereichsspezifisches Wissen (außer den Spielregeln).

### 1.8.1 Netzwerk-Architektur

Ein einzelnes tiefes neuronales Netz $f_\theta$ wird verwendet, das zwei Ausgaben liefert:

- **Policy Head p:** Eine Wahrscheinlichkeitsverteilung über mögliche Züge ($p_a \approx \pi(a|s)$).
- **Value Head $v$:** Eine skalare Bewertung des Zustands ($v \approx V(s)$) im Intervall $[-1,1]$ (Sieg/Niederlage).

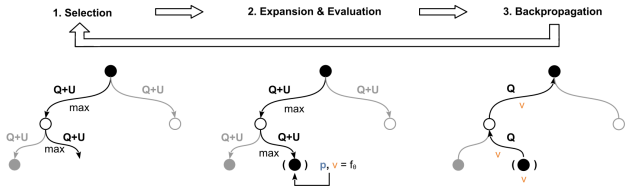
### 1.8.2 MCTS und PUCT

AlphaZero nutzt MCTS zur Planung. In jedem Schritt der Simulation wählt es Kanten basierend auf dem **PUCT-Algorithmus** (Predictor Upper Confidence Bounds for Trees):

$$a_t = \text{argmax}_a(Q(s,a) + U(s,a))$$

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

Hierbei dient $Q$ der Exploitation (Wertschätzung) und $U$ der Exploration (gesteuert durch die Prior-Wahrscheinlichkeit $P$ aus dem neuronalen Netz und der Besuchshäufigkeit $N$).




### 1.8.3 Training und Loss-Funktion

Das Netzwerk lernt aus den durch MCTS verbesserten Politiken ($\pi_{\text{MCTS}}$) und den tatsächlichen Spielergebnissen ($z$). Das Training minimiert folgenden Loss:

$$l = (z - v)^2 - \pi_{\text{MCTS}}^T \log \mathbf{p} + c||\theta||^2$$

- $(z - v)^2$: Mean Squared Error für den Value Head (soll das echte Endergebnis vorhersagen).
- $-\pi^T \log p$: Cross-Entropy für den Policy Head (das Netz soll die MCTS-Verteilung vorhersagen).
- $c||\theta||^2$: Regularisierungsterm.

Das System verbessert sich iterativ durch **Self-Play**: Das aktuelle beste Netzwerk generiert Trainingsdaten, die genutzt werden, um eine neue Version des Netzwerks zu trainieren.