1 Reinforcement Learning und AlphaZero

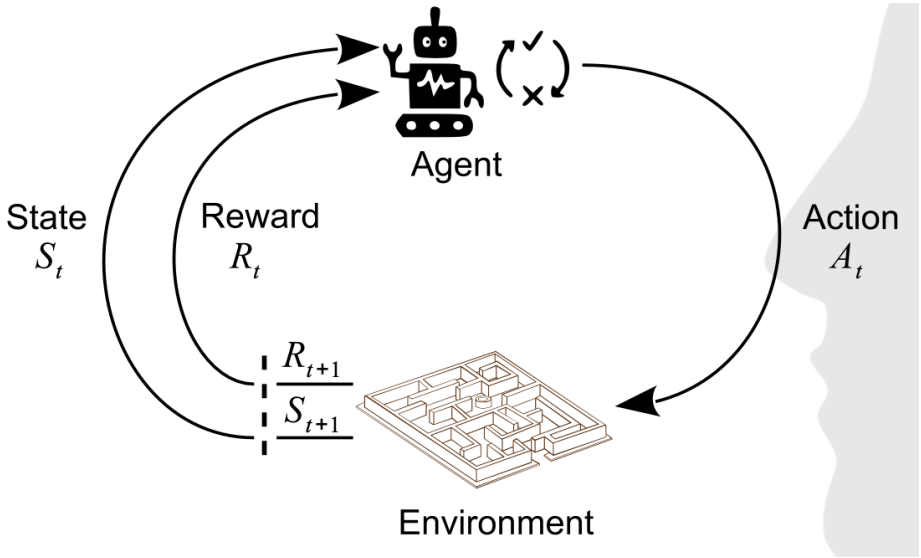
1.1 Grundlagen des Reinforcement Learning

Reinforcement Learning (RL) ist ein Teilgebiet des maschinellen Lernens, bei dem ein **Agent** lernt, wie er sich in einer **Umgebung** verhalten muss, um eine numerische Belohnung (**Reward**) zu maximieren. Im Gegensatz zum Supervised Learning erhält der Agent keine direkten Anweisungen (Labels), welche Aktion die beste ist, sondern muss diese durch Interaktion (Trial-and-Error) herausfinden.

Der RL-Zyklus

Der Prozess läuft in diskreten Zeitschritten t ab:

1. Der Agent beobachtet den aktuellen Zustand (State) S_t.
2. Basierend auf dieser Beobachtung wählt der Agent eine Aktion (Action) A_t.
3. Die Umgebung reagiert auf die Aktion, wechselt in einen neuen Zustand S_{t+1} und gibt ein Reward-Signal 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 formal zu beschreiben, werden **Markov Decision Processes** (MDPs) verwendet. Ein MDP ist definiert als ein Tupel (S, A, T, R, γ):

- S **(State Space):** Die Menge aller möglichen Zustände s.
- A **(Action Space):** Die Menge aller möglichen Aktionen a.
- T **(Transition Function):** Auch bekannt als das *Modell* der Umgebung. Es beschreibt die Wahrscheinlichkeitsverteilung der Zustandsübergänge:

  $$T(s, a, s') = P(S_{t+1} = s' \mid S_t = s, A_t = a)$$

  Dies ist die Wahrscheinlichkeit, im Zustand s' zu landen, wenn man im Zustand s die Aktion a ausführt.
- R **(Reward Function):** Die Belohnungsfunktion $R(s, a, s')$. Sie definiert den unmittelbaren Reward, den der Agent für den Übergang von s nach s' mittels Aktion a erhält.

- γ **(Discount Factor):** Der Diskontierungsfaktor mit $0 \leq \gamma \leq 1$. Er bestimmt, wie stark zukünftige Belohnungen im Vergleich zu sofortigen Belohnungen gewichtet werden (siehe unten).

1.2.1 Die Markov-Eigenschaft

Ein Prozess heißt **Markovian**, wenn die Wahrscheinlichkeit für den nächsten Zustand S_{t+1} nur vom aktuellen Zustand S_t und der Aktion A_t abhängt, und nicht von der gesamten Historie der vorherigen Zustände und Aktionen. Das "Gedächtnis" des Systems ist also im aktuellen Zustand vollständig enthalten.

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

1.2.2 Credit Assignment Problem

Eine der größten Herausforderungen im RL ist das **Temporal Credit Assignment Problem**. Belohnungen treten oft verzögert auf (z.B. wird ein Schachspiel erst nach vielen Zügen gewonnen). Der Algorithmus muss bestimmen, welche der vergangenen Aktionen für den späteren Erfolg (oder Misserfolg) verantwortlich waren und den "Credit" (die Anerkennung) entsprechend zuweisen.

1.2.3 Discounted Rewards (Return)

Der **Return** G_t ist die Summe der diskontierten Belohnungen ab Zeitpunkt t. Der Discount Factor γ sorgt dafür, dass die Summe bei unendlichen Horizonten konvergiert und modelliert die Unsicherheit über die Zukunft:

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

- $\gamma \to 0$: Der Agent ist "kurzsichtig" und kümmert sich nur um sofortige Belohnungen.
- $\gamma \to 1$: Der Agent ist "weitsichtig" und berücksichtigt zukünftige Belohnungen stark.

1.3 Optimalität und Value Functions

Das Ziel des Agenten ist es, eine **Policy** (Strategie) π zu finden, die den erwarteten Return maximiert.

- **Policy** $\pi(s)$**:** Eine Vorschrift, die jedem Zustand eine Aktion zuordnet (deterministisch $a = \pi(s)$ oder stochastisch $\pi(a|s)$).

Value Functions

- **State-Value Function** $V^\pi(s)$**:** Der Wert eines Zustands. Er gi
  man in Zustand s startet und danach immer der Policy π folgt.
- **Action-Value Function** $Q^\pi(s,a)$**:** Der Wert einer Handlung (Q-W
  an, wenn man in Zustand s startet, *einmalig* Aktion a wählt und

Um Policies zu bewerten, nutzen wir Value Functions:

1.3.1 Bellman-Gleichung der Optimalität

Die optimale Value Function $V^*(s)$ ist der maximal mögliche erwartete Return, der in einem Zustand erreicht werden kann. Sie lässt sich rekursiv durch die Bellman-Gleichung beschreiben:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

Erklärung der Terme:

- $\max_a$: Wir suchen die Aktion, die das beste Ergebnis liefert (Optimierung).
- $\sum_{s'} T(\dots)$: Da die Umgebung stochastisch sein kann, bilden wir den Erwartungswert über alle möglichen Folgezustände s', gewichtet mit ihrer Wahrscheinlichkeit T.
- $R + \gamma V^*(s')$: Der Wert setzt sich zusammen aus dem sofortigen Reward R und dem diskontierten Wert des nächsten Zustands $V^*(s')$.

1.4 Lösen von MDPs: Value Iteration

Wenn das Modell der Umgebung (T und R) *bekannt* ist, spricht man nicht von Lernen, sondern von **Planning**. Ein klassischer Algorithmus der Dynamischen Programmierung ist die **Value Iteration**.

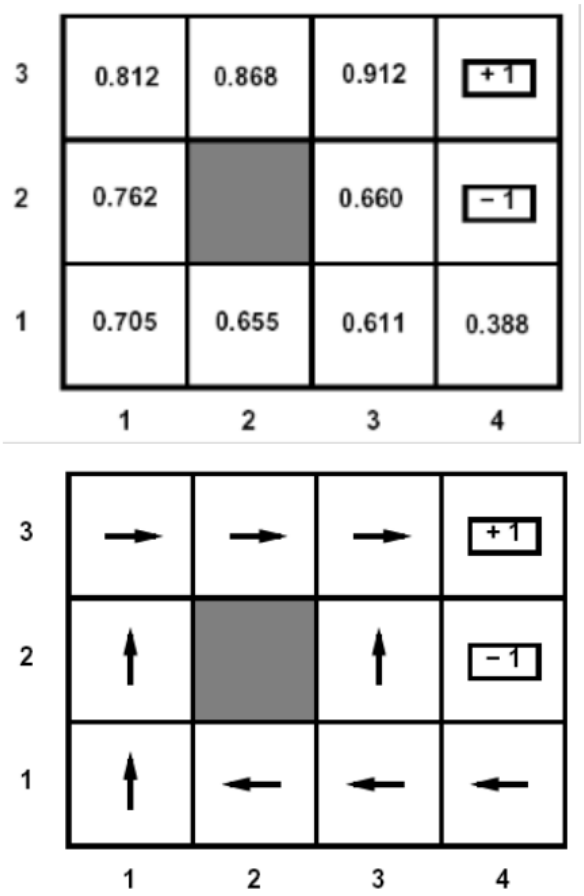
Value Iteration Algorithmus

1. Initialisiere $V_0(s) = 0$ für alle Zustände (oder zufällig).
2. Wiederhole in jeder Iteration k für alle Zustände s das Bellman-Update:

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

3. Stoppe, wenn die Änderung der Werte klein genug ist (Konvergenz).

Die resultierenden Werte konvergieren gegen V^*. Die optimale Policy kann dann einfach abgeleitet werden, indem man in jedem Zustand die Aktion wählt, die den Term maximiert (Greedy-Policy).



1.5 Reinforcement Learning (Model-Free)

In echten RL-Szenarien sind $T(s, a, s')$ und $R(s, a, s')$ *nicht* bekannt. Der Agent muss durch Interaktion lernen.

1.5.1 Unterscheidung der Lernarten

- **Model-Based RL:** Der Agent versucht zuerst, T und R zu schätzen (z.B. durch Zählen von Häufigkeiten) und plant dann (z.B. mit Value Iteration).
- **Model-Free RL:** Der Agent lernt direkt die Value-Function (V oder Q) oder die Policy, ohne die Dynamik der Welt explizit zu modellieren.

- **Passive Learning:** Der Agent folgt einer fixen Policy π und bewertet diese (Policy Evaluation).
- **Active Learning:** Der Agent wählt Aktionen selbst, um die optimale Policy zu finden (Exploration nötig).

1.5.2 Temporal-Difference (TD) Learning

TD-Learning ist eine Methode für *Passive Learning* (Model-Free). Anstatt bis zum Ende einer Episode zu warten (wie bei Monte-Carlo-Methoden), aktualisiert TD die Schätzung basierend auf dem nächsten Zeitschritt (*Bootstrapping*).

Update-Regel für $V(s)$:

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

- α **(Learning Rate):** Bestimmt, wie stark neue Informationen alte überschreiben ($0 < \alpha \leq 1$).
- **Target:** Der geschätzte "richtige" Wert basierend auf dem erhaltenen Reward R und der Schätzung des nächsten Zustands $V(s')$.

1.5.3 Q-Learning

Q-Learning ist der wichtigste **Off-Policy** Algorithmus für *Active Learning*. Da wir kein Modell haben, lernen wir $Q(s, a)$-Werte anstelle von $V(s)$-Werten, da uns Q-Werte direkt sagen, welche Aktion die beste ist.

Q-Learning Update Regel

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

Detaillierte Analyse der Parameter:

- **Alter Wert** $(1 - \alpha) Q(s, a)$: Wir behalten einen Teil des alten Wissens.
- **Sample / Target** $R + \gamma \max_{a'} Q(s', a')$: Dies ist die neue Schätzung. Wichtig ist der Term $\max_{a'} Q(s', a')$. Er bedeutet, dass wir für das Update davon ausgehen, dass wir im *nächsten* Zustand die bestmögliche Aktion wählen – selbst wenn der Agent dies in der Realität vielleicht gar nicht tut (deshalb "Off-Policy").
- **Konvergenz:** Q-Learning konvergiert gegen $Q^*(s, a)$, sofern alle Zustands-Aktions-Paare unendlich oft besucht werden und α passend verringert wird.

1.5.4 Exploration vs. Exploitation

Damit der Agent optimale Wege findet, muss er die Umgebung erkunden, anstatt immer nur das (bisher) Beste zu tun.

- **Exploitation (Ausbeutung):** Wähle Aktion mit höchstem Q-Wert: $a = \text{argmax}_a Q(s, a)$.
- **Exploration (Erkundung):** Wähle eine zufällige Aktion, um neue Zustände zu entdecken.
- **ϵ-Greedy Strategie:**
  - Mit Wahrscheinlichkeit ϵ: Wähle zufällige Aktion (Exploration).
  - Mit Wahrscheinlichkeit $1 - \epsilon$: Wähle beste Aktion (Exploitation).

  Oft wird ϵ über die Zeit verringert (Decay), um zu Beginn viel zu lernen und später optimal zu handeln.

1.6 Deep Q-Networks (DQN)

In komplexen Umgebungen (z.B. Atari-Spiele mit Pixel-Input) ist eine Tabelle für alle $Q(s, a)$ zu groß. DQN nutzt ein **Deep Neural Network** als Function Approximator: $Q(s, a; \theta) \approx Q^*(s, a)$, wobei θ die Gewichte des Netzes sind.

1.7 AlphaZero

AlphaZero ist ein allgemeiner Algorithmus, der Spiele wie Schach, Shogi und Go meistert, ohne menschliches Vorwissen (außer den Spielregeln). Es kombiniert **Monte-Carlo Tree Search (MCTS)** mit tiefen neuronalen Netzen und Self-Play.

1.7.1 Netzwerk-Architektur

AlphaZero verwendet ein einziges tiefes neuronales Netz f_θ, das zwei Ausgabeköpfe (Heads) hat:

1. **Policy Head p:** Ein Vektor von Wahrscheinlichkeiten $p_a = P(a|s)$. Er schätzt ab, welche Züge gut sind (dient als Prior für die Suche).
2. **Value Head** v**:** Ein skalarer Wert $v \in [-1, 1]$. Er schätzt die Gewinnwahrscheinlichkeit des aktuellen Zustands ($+1$ Sieg, -1 Niederlage).

1.7.2 MCTS und der PUCT-Algorithmus

Anstatt Minimax mit Alpha-Beta-Pruning zu nutzen, führt AlphaZero eine MCTS-Suche durch. Um zu entscheiden, welcher Knoten im Suchbaum expandiert wird, nutzt es den **PUCT-Algorithmus** (Predictor Upper Confidence Bounds for Trees). Die Auswahlregel für eine Aktion a im Baum ist:

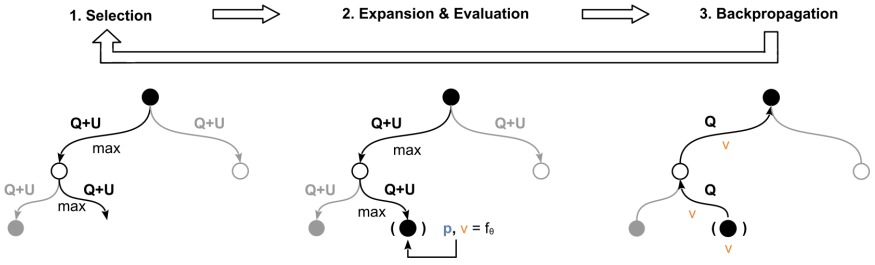
$$a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$$

Dabei ist U der Explorationsterm:

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

Erklärung der Parameter:

- $Q(s, a)$ **(Exploitation):** Der mittlere Wert (Value) der bisherigen Simulationen für diesen Zug.
- $P(s, a)$**:** Die "Prior"-Wahrscheinlichkeit für diesen Zug, die direkt vom *Policy Head* des neuronalen Netzes kommt. Gute Züge werden also bevorzugt behandelt.
- $N(s, a)$**:** Wie oft wurde dieser Zug im Baum bereits besucht?
- $\frac{\sqrt{\sum N}}{1+N}$**:** Dieser Term sorgt für Exploration. Züge, die selten besucht wurden (kleines $N(s, a)$), erhalten einen Bonus, besonders wenn der Elternknoten oft besucht wurde.
- c_{puct}**:** Eine Konstante, die die Balance zwischen Exploration (U) und Exploitation (Q) steuert.



1.7.3 Training und Loss-Funktion

Das neuronale Netz wird trainiert, um die Ergebnisse der MCTS-Suche vorherzusagen. Die MCTS-Suche liefert eine verbesserte Policy π_{MCTS} (basierend auf den Besuchszahlen N) und am Ende des Spiels steht der tatsächliche Gewinner z fest.

Die Loss-Funktion (Fehlerfunktion), die minimiert wird, ist:

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

Aufschlüsselung der Terme:

- $(z - v)^2$ **(Mean Squared Error):** Der Value Head v soll das tatsächliche Spielergebnis z (Sieg/Niederlage) so genau wie möglich vorhersagen.
- $-\pi_{\mathbf{MCTS}}^T \log \mathbf{p}$ **(Cross-Entropy):** Der Policy Head $\mathbf{p}$ (die Vorhersage des Netzes) soll der durch MCTS verbesserten Suchverteilung π_{MCTS} so ähnlich wie möglich sein. Das Netz lernt also, die langsame Baumsuche zu "imitieren".
- $c||\theta||^2$ **(L2-Regularisierung):** Verhindert Overfitting der Gewichte θ.

Das System verbessert sich iterativ ("Self-Play"): Das Netz spielt gegen sich selbst, generiert Daten, wird darauf trainiert und wird dadurch stärker, was wiederum bessere Trainingsdaten für die nächste Iteration liefert.