Einführung in die KI

Niclas Kusenbach
LaTeX version: SCHOUTER

Table of Contents

Contents

1 Introduction to AI Principle

1.1 Was ist Künstliche Intelligenz?

Die Definition von Künstlicher Intelligenz (KI) ist nicht eindeutig und unterliegt einem stetigen Wandel („Moving Goalposts“). Aufgaben, die früher als KI galten, werden heute oft als reine Softwaretechnik angesehen, sobald sie gelöst sind.

Definitionen von KI

Es gibt verschiedene Ansätze, KI zu definieren:

- **John McCarthy (1971):** „Die Wissenschaft und Ingenieurskunst, **intelligente Maschinen** zu bauen, insbesondere intelligente Computerprogramme.“ Er betonte, dass KI nicht auf biologisch beobachtbare Methoden beschränkt sein muss.
- **Marvin Minsky (1969):** „Die Wissenschaft, Maschinen dazu zu bringen, Dinge zu tun, die **Intelligenz** erfordern würden, wenn sie von Menschen getan würden.“

Gründe für die schwierige Definition sind unter anderem:

- Fehlende offizielle Definition.
- Einfluss von Science-Fiction auf die öffentliche Wahrnehmung.
- Das Phänomen, dass ehemals schwere Aufgaben (z.B. Schach) nach ihrer Lösung als „einfache Rechnerei“ abgetan werden, während Aufgaben, die für Menschen einfach sind (z.B. Wahrnehmung, Motorik), für Maschinen sehr schwer sein können (Moravec's Paradox).

1.2 Intelligenztests und Philosophie

Um festzustellen, ob ein System intelligent ist, wurden verschiedene Tests und Gedankenexperimente entwickelt.

1.2.1 Der Turing-Test (Imitation Game)

Vorgeschlagen von Alan Turing. Die Grundannahme ist, dass ein Wesen intelligent ist, wenn es sich in seinem Verhalten nicht von einem anderen intelligenten Wesen unterscheiden lässt.

Ablauf:

1. Ein menschlicher Fragesteller interagiert blind (per Text) mit zwei Spielern: A und B.
2. Einer der Spieler ist ein Mensch, der andere ein Computer.
3. Wenn der Fragesteller nicht zuverlässig entscheiden kann, welcher Spieler der Computer ist, hat der Computer den Test bestanden.

Kritik: Der Test prüft *Verhalten*, nicht das *Verständnis* oder *Bewusstsein*. Ein System kann menschliches Verhalten simulieren, ohne die zugrunde liegenden Konzepte zu verstehen.

1.2.2 Das Chinesische Zimmer (The Chinese Room)

Ein Gegenargument zum Turing-Test von John Searle, das den Unterschied zwischen Syntax (Zeichenmanipulation) und Semantik (Bedeutung) hervorhebt.

Szenario:

- Eine Person, die kein Chinesisch versteht, sitzt in einem geschlossenen Raum.
- Sie hat ein Regelbuch (Programm), das beschreibt, wie auf chinesische Zeichenfolgen (Input) mit anderen chinesischen Zeichenfolgen (Output) reagiert werden soll.

- Für einen Außenstehenden wirkt es so, als verstünde die Person Chinesisch.

Schlussfolgerung: Die Person manipuliert nur Symbole anhand ihrer Form (Syntax), versteht aber deren Inhalt (Semantik) nicht. Ebenso könnte eine KI intelligent *wirken* (Turing-Test bestehen), ohne wirklich intelligent zu *sein*.

1.3 Kategorisierung von KI

1.3.1 Starke vs. Schwache KI

Unterscheidung der Reichweite

- **Generelle KI (General AI / Strong AI):** Ein System, das **jede** intellektuelle Aufgabe bewältigen kann, die auch ein Mensch lösen kann. Es besitzt Verständnis und Bewusstsein (Forschungsziel).
- **Schwache KI (Narrow AI / Weak AI):** Ein System, das darauf spezialisiert ist, eine **konkrete** oder eine begrenzte Menge von Aufgaben zu lösen (aktueller Stand der Technik, z.B. Schachcomputer, Bilderkennung).

1.3.2 Eigenschaften eines KI-Systems

Ein modernes KI-System sollte idealerweise folgende Eigenschaften aufweisen:

- **Adaptability (Anpassungsfähigkeit):** Die Fähigkeit, die Leistung durch Lernen aus Erfahrung zu verbessern.
- **Autonomy (Autonomie):** Die Fähigkeit, Aufgaben in Umgebungen ohne ständige Anleitung durch einen Benutzer oder Experten auszuführen.
- **Rationality (Rationalität):** Das Treffen der „richtigen“ Entscheidungen (siehe unten).

1.4 Dimensionen der KI-Definition

KI kann entlang zweier Dimensionen klassifiziert werden:

1. **Prozess:** Fokus auf Denkprozesse/Schlussfolgern vs. Fokus auf Verhalten/Handeln.
2. **Maßstab:** Erfolg gemessen am menschlichen Vorbild vs. Erfolg gemessen an einem idealen Rationalitätsbegriff.

Dies ergibt vier Felder der KI-Forschung:

	Menschlicher Maßstab	**Ideal (Rationalität)**
Denken	**Systems that think like humans** (Kognitionswissenschaft: Modellierung der menschlichen Denkweise)	**Systems that think rationally** (Logik: „Gesetze des Denkens“, korrekte Schlussfolgerungen)
Handeln	**Systems that act like humans** (Turing-Test Ansatz: Simulation menschlichen Verhaltens)	**Systems that act rationally** (Rationaler Agent: Maximierung des erwarteten Nutzens)

Table 1: *Die vier Ansätze der KI*

1.4.1 Rationalität vs. Gesetze des Denkens

- **Laws of Thought (Logik):** Beschäftigt sich mit unwiderlegbaren, logischen Schlussfolgerungen. Problem: In der Realität gibt es oft Unsicherheiten, für die Logik allein nicht ausreicht.
- **Rational Behavior (Rationales Handeln):** Das „Richtige“ tun. Das Richtige ist definiert als das, was die Erreichung der Ziele angesichts der verfügbaren Informationen maximiert (Maximierung des *Expected Utility*).
- **Vorteile der Rationalität:** Sie ist allgemeiner als reine Logik (funktioniert auch bei Unsicherheit) und wissenschaftlich besser handhabbar (Optimierungsproblem).

1.5 Grundlagen und verwandte Disziplinen

KI ist ein interdisziplinäres Feld, das auf vielen Bereichen aufbaut:

- **Philosophie:** Logik, Reasoning, Geist als physisches System, Grundlagen des Lernens.
- **Mathematik:** Formale Repräsentation, Beweise, Algorithmen, Wahrscheinlichkeitstheorie.
- **Psychologie / Kognitionswissenschaft:** Wahrnehmung, Motorik, Anpassung. Kognitionswissenschaft verbindet KI-Modelle mit experimentellen Techniken der Psychologie.
- **Ökonomie:** Entscheidungstheorie, Spieltheorie (rationale Entscheidungen).
- **Neurowissenschaften:** Physisches Substrat (Gehirn) für mentale Aktivitäten.
- **Linguistik:** Wissensrepräsentation, Grammatik.
- **Kontrolltheorie:** Stabilität, optimales Agentendesign.

1.5.1 Taxonomie der KI

Die Begriffe werden oft hierarchisch verstanden:

- **Künstliche Intelligenz (KI):** Der Überbegriff für Technik, die intelligente Züge zeigt.
- **Machine Learning (Maschinelles Lernen):** Ein Teilgebiet der KI, das Algorithmen nutzt, um aus Daten zu lernen (statt explizit programmiert zu werden).
- **Deep Learning:** Ein Teilgebiet des Machine Learning, das auf künstlichen neuronalen Netzen mit vielen Schichten basiert (aktuell sehr erfolgreich, siehe Nobelpreis Physik 2024 an Hinton/Hopfield).

1.6 Grenzen und Probleme moderner KI

Trotz großer Erfolge (z.B. AlphaGo, Stable Diffusion, Protein Folding) existieren signifikante Limitationen:

- **Bias (Voreingenommenheit):** KI-Modelle übernehmen Vorurteile aus den Trainingsdaten (z.B. rassistische Tendenzen in Gesichtserkennung oder Textgenerierung).
- **Adversarial Attacks:** KI kann leicht getäuscht werden. Durch für Menschen unsichtbare Änderungen an einem Bild (Rauschen) kann eine KI dazu gebracht werden, ein Objekt völlig falsch zu klassifizieren (z.B. Panda wird mit 99% Konfidenz als Gibbon erkannt).
- **Halluzinationen:** Generative KIs erzeugen plausibel klingende, aber faktisch falsche Informationen.
- **Isolation:** Aktuelle KI ist meist „Narrow AI“ und auf spezifische Probleme beschränkt, ohne allgemeines Weltverständnis.

1.7 Ergänzung: Agenten (aus der Übung)

Ein zentrales Konzept der KI ist der **Agent**.

- **Agent:** Eine Einheit, die ihre Umgebung über **Sensoren** wahrnimmt und mittels **Aktuatoren** auf diese Umgebung einwirkt.
- **Environment (Umgebung):** Die Welt, in der der Agent operiert.

Zur Beschreibung eines KI-Problems (z.B. Roboterfußball) werden oft folgende Aspekte analysiert:

- **Performance Measure:** Wie wird Erfolg gemessen? (z.B. Anzahl der Tore, gewonnene Spiele).
- **Environment:** Was beinhaltet die Welt? (z.B. Spielfeld, Ball, Gegner, Tore).
- **Actuators:** Wie kann der Agent handeln? (z.B. Motoren für Beine, Schussmechanik).
- **Sensors:** Wie nimmt der Agent wahr? (z.B. Kameras, Beschleunigungssensoren).

Umgebungseigenschaften können klassifiziert werden als:

- *Observable* (vollständig beobachtbar) vs. *Partially Observable*.

- *Accessible* (Kann ein Agent vollständige & akkurate Informationen erhalten (gibt es diese überhaupt)) vs. *Inaccesssible*
- *Deterministic* (Folgezustand durch aktuellen Zustand und Aktion bestimmt) vs. *Stochastic*.
- *Episodic* (Handlungen sind unabhängig voneinander) vs. *Sequential*.
- *Static* (Umgebung ändert sich nicht während der Entscheidungsfindung) vs. *Dynamic*.
- *Discrete* (endliche Anzahl an Zuständen/Aktionen) vs. *Continuous*.

2 AI101-02: AI Systems

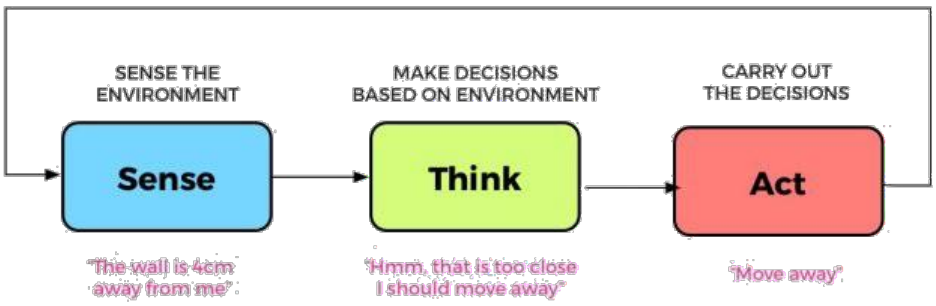
Dieses Kapitel behandelt die grundlegenden Bausteine von KI-Systemen: Agenten und ihre Umgebungen. Es definiert Rationalität und klassifiziert verschiedene Arten von Umgebungen und Agentenarchitekturen.

2.1 KI-Systeme und Agenten

Ein KI-System wird definiert durch die Interaktion zwischen einem Agenten und seiner Umgebung.

Definition: Agent

Ein **Agent** ist eine Entität, die ihre Umgebung durch **Sensoren** wahrnimmt und auf diese Umgebung durch **Aktuatoren** (Effektoren) einwirkt.



- **Wahrnehmung (Percept):** Der sensorische Input des Agenten zu einem bestimmten Zeitpunkt.
- **Wahrnehmungssequenz (Percept Sequence):** Die vollständige Historie aller Wahrnehmungen, die der Agent bisher erhalten hat.
- **Agentenfunktion:** Eine mathematische Abbildung von Wahrnehmungssequenzen auf Aktionen: $f : P^* \to A$.
- **Agentenprogramm:** Die konkrete Implementierung der Agentenfunktion, die auf einer physischen Architektur läuft.

2.2 Rationalität

Ein rationaler Agent ist nicht zwangsläufig ein perfekter oder allwissender Agent. Rationalität bezieht sich auf die Qualität der Entscheidungsfindung basierend auf den verfügbaren Informationen.

Rationaler Agent

Ein **rationaler Agent** wählt für jede mögliche Wahrnehmungssequenz diejenige Aktion aus, die erwartungsgemäß sein Leistungsmaß (*Performance Measure*) maximiert, unter Berücksichtigung der Wahrnehmungssequenz und des eingebauten Wissens.

Rationalität hängt von vier Faktoren ab:

1. Dem **Leistungsmaß** (Performance Measure), das den Erfolg definiert.
2. Dem **Wissen**, das der Agent bereits besitzt (Prior Knowledge).
3. Den **Aktionen**, die der Agent ausführen kann.
4. Der bisherigen **Wahrnehmungssequenz**.

Wichtige Unterscheidung:

- **Allwissenheit (Omniscience):** Kennt das tatsächliche Ergebnis jeder Aktion (in der Realität unmöglich).
- **Rationalität:** Maximiert das *erwartete* Ergebnis basierend auf dem aktuellen Wissen.

2.3 Task Environments (PEAS)

Um einen Agenten zu entwerfen, muss die Aufgabenstellung spezifiziert werden. Dies geschieht durch das PEAS-Modell.

PEAS

- **P**erformance Measure (Leistungsmaß): Woran wird Erfolg gemessen?
- **E**nvironment (Umgebung): Wo operiert der Agent?
- **A**ctuators (Aktuatoren): Womit handelt der Agent?
- **S**ensors (Sensoren): Womit nimmt der Agent wahr?

Beispiel: Autonomes Taxi

- **P:** Sicherheit, Schnelligkeit, Legalität, Komfort, Profit.
- **E:** Straßen, anderer Verkehr, Fußgänger, Wetter.
- **A:** Lenkung, Gaspedal, Bremse, Hupe, Blinker.
- **S:** Kameras, Radar, Tacho, GPS, Motorensensoren.

2.4 Eigenschaften von Umgebungen

Die Art der Umgebung bestimmt maßgeblich das Design des Agenten. Umgebungen werden anhand folgender Dimensionen klassifiziert:

1. **Fully Observable (Vollständig beobachtbar) vs. Partially Observable:**
   - *Fully:* Die Sensoren geben zu jedem Zeitpunkt den kompletten Zustand der Umgebung wieder (kein interner Speicher nötig).
   - *Partially:* Teile des Zustands sind verdeckt oder Sensoren sind ungenau (interner Zustand/Gedächtnis nötig).
2. **Single Agent vs. Multi-Agent:**
   - *Single:* Der Agent agiert allein (z.B. Kreuzworträtsel).
   - *Multi:* Es gibt andere Agenten, die kooperativ oder kompetitiv sein können (z.B. Schach, Straßenverkehr).
3. **Deterministic vs. Stochastic:**
   - *Deterministic:* Der nächste Zustand wird vollständig durch den aktuellen Zustand und die Aktion des Agenten bestimmt.
   - *Stochastic:* Es gibt Unsicherheiten/Zufall (z.B. Wetter, Würfelglück). Wenn Wahrscheinlichkeiten unbekannt sind, nennt man es *non-deterministic*.
4. **Episodic vs. Sequential:**
   - *Episodic:* Die Erfahrung ist in atomare Episoden unterteilt. Eine Aktion in Episode A hat keine Auswirkung auf Episode B (z.B. Bildklassifizierung).
   - *Sequential:* Die aktuelle Entscheidung beeinflusst alle zukünftigen Entscheidungen (z.B. Schach, Fahren).
5. **Static vs. Dynamic:**
   - *Static:* Die Umgebung ändert sich nicht, während der Agent "nachdenkt" (z.B. Kreuzworträtsel).
   - *Dynamic:* Die Umgebung ändert sich kontinuierlich (z.B. Taxi fahren).

- *Semi-dynamic:* Der Zustand ändert sich nicht, aber die Bewertung (Performance) ändert sich mit der Zeit (z.B. Schach mit Schachuhr).

6. **Discrete vs. Continuous:**
   - Bezieht sich auf die Zustände, Zeit oder Aktionen. Schach ist diskret (feste Felder), Taxi fahren ist kontinuierlich.

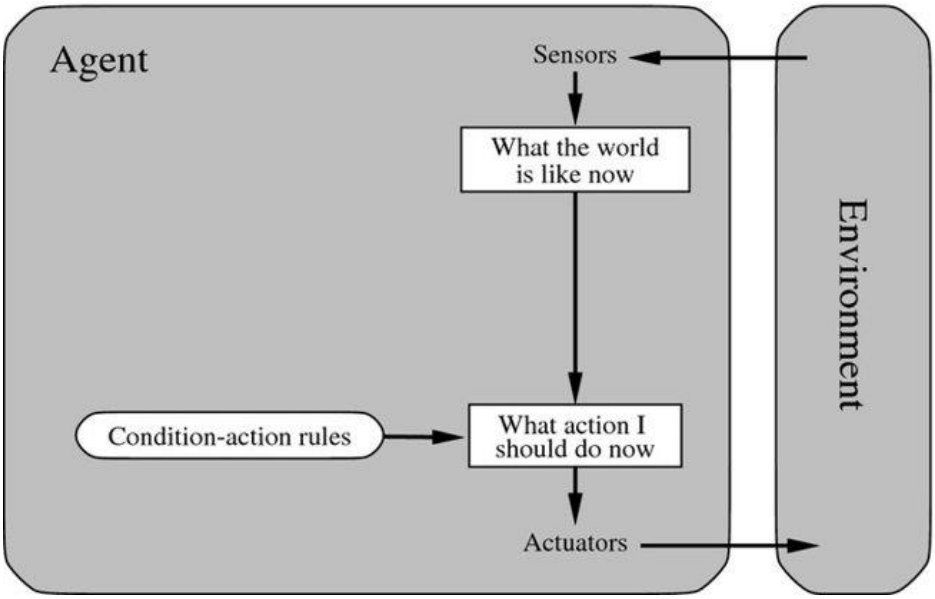
2.5 Agententypen

Es gibt vier grundlegende Typen von Agentenprogrammen, aufsteigend nach Komplexität und Fähigkeit.

2.5.1 1. Simple Reflex Agents (Einfache Reflex-Agenten)

Diese Agenten entscheiden nur basierend auf der *aktuellen* Wahrnehmung. Sie ignorieren die Wahrnehmungshistorie.

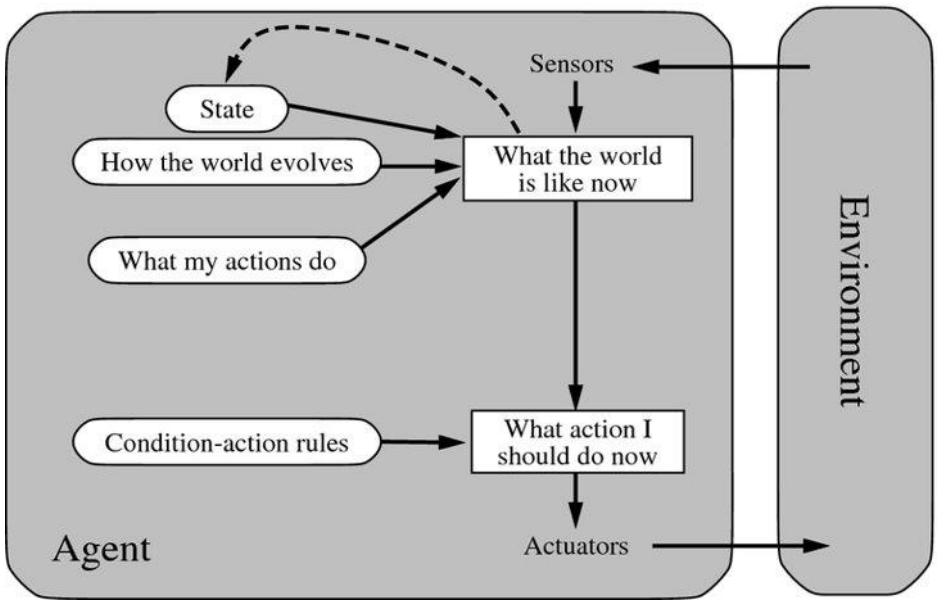
- **Funktionsweise:** Condition-Action Rules (Wenn-Dann-Regeln).
- **Formel:** Aktion = Funktion(Aktuelle Wahrnehmung).
- **Einschränkung:** Funktionieren nur zuverlässig in *fully observable* Umgebungen. Drohen in unendliche Schleifen zu geraten, wenn die Umgebung nur teilweise beobachtbar ist.



2.5.2 2. Model-based Reflex Agents (Modellbasierte Reflex-Agenten)

Diese Agenten besitzen einen internen Zustand (*Internal State*), um mit partieller Beobachtbarkeit umzugehen.

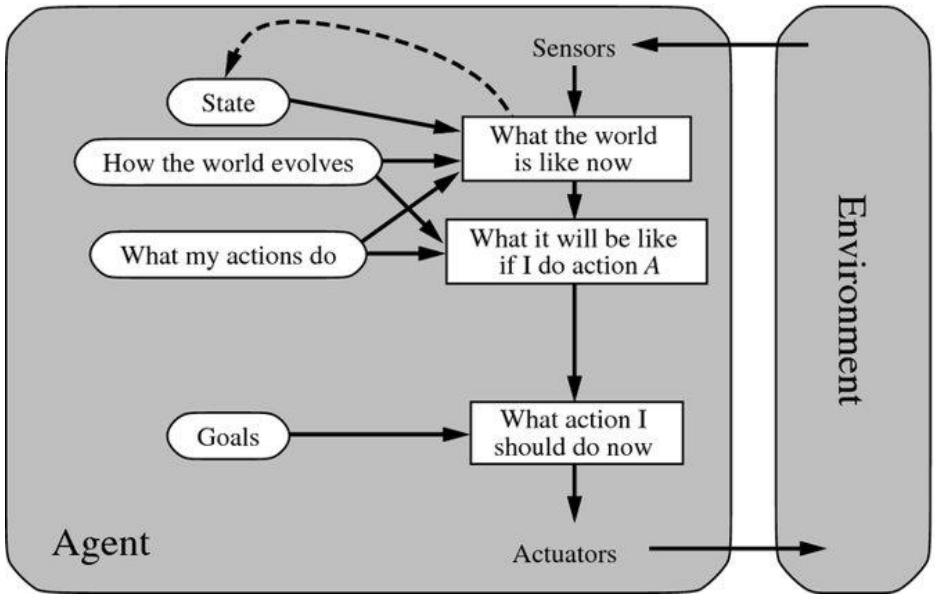
- **Interner Zustand:** Repräsentiert Wissen über die Welt, das aktuell nicht sichtbar ist (Gedächtnis).
- **Modell der Welt:** Wissen darüber, wie die Welt funktioniert (Übergangsmodelle) und wie Aktionen die Welt verändern.
- **Ablauf:** Update State → Wähle Regel → Aktion.



2.5.3 3. Goal-based Agents (Zielbasierte Agenten)

Wissen über den Zustand reicht nicht aus; der Agent benötigt ein Ziel (*Goal*).

- **Planung/Suche:** Der Agent simuliert verschiedene Sequenzen von Aktionen, um zu sehen, ob sie zum Ziel führen.
- **Unterschied zum Reflex:** Der Reflex-Agent reagiert, der Goal-based Agent plant in die Zukunft.
- **Flexibilität:** Ziele können sich ändern, der Agent passt sein Verhalten an.

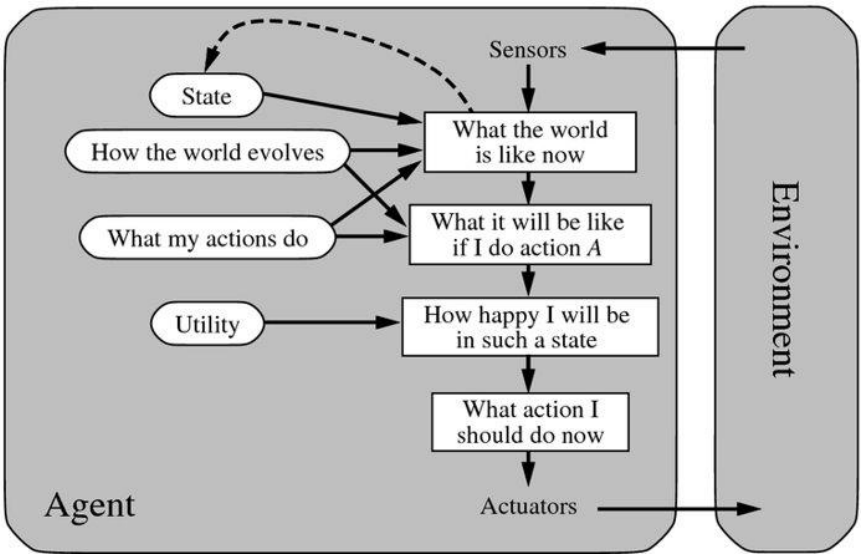


2.5.4 4. Utility-based Agents (Nutzenbasierte Agenten)

Ziele sind oft binär (erreicht/nicht erreicht). Nutzen (*Utility*) erlaubt eine feinere Unterscheidung.

- **Utility Function:** Bildet einen Zustand auf eine reelle Zahl ab, die den Grad der "Zufriedenheit" des Agenten angibt.
- **Vorteil:** Ermöglicht rationale Entscheidungen bei
  - Zielkonflikten (z.B. Geschwindigkeit vs. Sicherheit).

– Unsicherheit (Abwägung von Erfolgswahrscheinlichkeit und Nutzen).

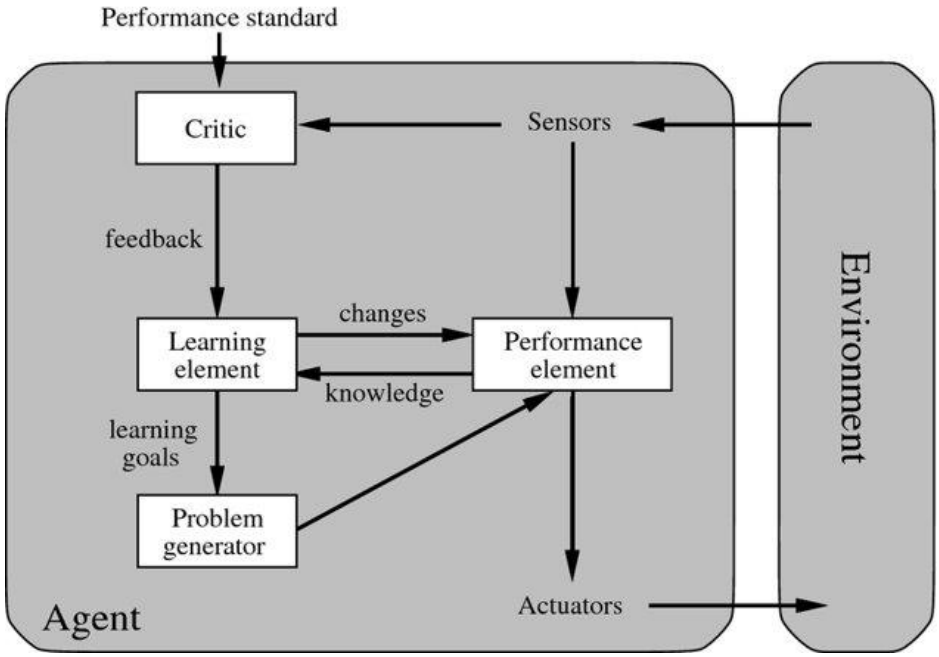


2.5.5 5. Learning Agents (Lernende Agenten)

Jeder der obigen Agententypen kann lernfähig gemacht werden. Dies erlaubt dem Agenten, in unbekannten Umgebungen zu operieren und kompetenter zu werden.

Komponenten eines lernenden Agenten

- **Performance Element:** Der eigentliche Agent (einer der 4 obigen Typen), der Entscheidungen trifft.
- **Critic (Kritiker):** Bewertet, wie gut der Agent ist, basierend auf einem festen Leistungsstandard.
- **Learning Element:** Nutzt das Feedback des Kritikers, um das Performance Element zu verbessern.
- **Problem Generator:** Schlägt Aktionen vor, die zu neuen Erfahrungen führen (Exploration), auch wenn diese kurzfristig suboptimal sind.



3 AI101-03 Uninformed and Informed Search

3.1 Einführung und Problemdefinition

Problemlösende Agenten sind zielbasierte Agenten, die atomare Repräsentationen verwenden (Zustände als Black Boxes). Der Prozess besteht aus vier Phasen:

1. **Zielformulierung:** Definition des Ziels basierend auf der aktuellen Situation.
2. **Problemformulierung:** Entscheidung über zu betrachtende Aktionen und Zustände.
3. **Suche:** Prozess des Findens einer Aktionssequenz, die zum Ziel führt.
4. **Ausführung:** Durchführung der gefundenen Aktionen.

Wohlformuliertes Suchproblem

Ein Suchproblem wird durch fünf Komponenten definiert:

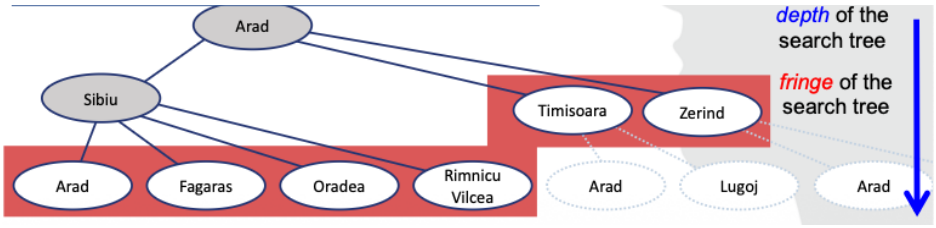
1. **Initial State (Startzustand)** s_0: Der Zustand, in dem der Agent beginnt.
2. **Actions (Aktionen)** $A(s)$: Die Menge der möglichen Aktionen in einem Zustand s.
3. **Transition Model (Überführungsmodell)** $Result(s, a)$: Beschreibt, was eine Aktion tut. Rückgabe ist der Folgezustand.
4. **Goal Test (Zieltest)**: Bestimmt, ob ein Zustand ein Zielzustand ist.
5. **Path Cost (Pfadkosten)** $c(s, a, s')$: Additive Kostenfunktion. Meistens sind Schrittkosten nicht-negativ.

Lösung: Eine Sequenz von Aktionen, die vom Startzustand zum Ziel führt.
Optimale Lösung: Eine Lösung mit den geringsten Pfadkosten.

3.2 Suchalgorithmen: Tree Search vs. Graph Search

Der Kern aller Suchalgorithmen ist die Expansion von Zuständen.

- **Fringe (Open List):** Menge der generierten (entdeckten), aber noch nicht expandierten (bearbeiteten) Knoten. Sie wartet darauf, vom Algorithmus besucht zu werden.
- **Expansion:** Anwenden der Aktionen auf einen Zustand, um alle direkten Nachfolger (Kindknoten) zu generieren und zur Fringe hinzuzufügen.



Unterschied Tree vs. Graph Search

- **Tree Search:** Verfolgt nicht, welche Zustände bereits besucht wurden. Es werden lediglich Knoten generiert. Dies kann dazu führen, dass redundante Pfade mehrfach besucht werden oder der Algorithmus in Zykeln (Schleifen) hängen bleibt.
- **Graph Search:** Speichert besuchte Zustände in einer *Explored Set* (Closed List). Bevor ein Knoten zur Fringe hinzugefügt wird, wird geprüft, ob der Zustand bereits bekannt ist. Dies vermeidet Redundanz und Endlosschleifen.

3.2.1 Bewertungskriterien und Parameter

Um Suchstrategien zu vergleichen, werden folgende Metriken genutzt, die auf den Eigenschaften des Suchraums basieren:

Parameter der Komplexität:

- b **(Branching factor):** Der Verzweigungsfaktor. Die maximale Anzahl an Nachfolgern, die ein Knoten haben kann.
- d **(Depth):** Die Tiefe des *flachsten* (am wenigsten Schritte entfernten) Zielknotens.
- m **(Max Depth):** Die maximale Tiefe des Suchraums (Länge des längsten möglichen Pfades). Kann unendlich (∞) sein.

Kriterien:

- **Completeness (Vollständigkeit):** Findet der Algorithmus garantiert eine Lösung, wenn eine existiert?
- **Optimality (Optimalität):** Findet er garantiert die kostengünstigste Lösung (minimale Pfadkosten)?
- **Time Complexity:** Anzahl der generierten Knoten (Rechenzeit).
- **Space Complexity:** Maximale Anzahl der Knoten, die gleichzeitig im Speicher gehalten werden müssen.

3.3 Uninformierte Suche (Blind Search)

Uninformierte Strategien haben keine Information darüber, wie nah ein Zustand am Ziel ist. Sie unterscheiden sich nur in der Reihenfolge, in der sie Knoten aus der Fringe zur Expansion auswählen.

3.3.1 Breadth-First Search (BFS) - Breitensuche

Erkundet den Suchbaum schichtweise (Ebene für Ebene).

- **Algorithmus:** Verwendet eine **FIFO-Queue** (First-In-First-Out).
  1. Wähle den *ältesten* Knoten in der Fringe (geringste Tiefe).
  2. Überprüfe auf Zielzustand.
  3. Generiere alle Nachfolger und füge sie *hinten* an die Fringe an.
- **Vollständig**: Ja (solange b endlich ist), da er irgendwann jede endliche Tiefe d erreicht.
- **Optimal**: Ja, aber **nur** wenn alle Schrittkosten gleich sind (z.B. 1). Dann ist der flachste Pfad auch der kostengünstigste.
- **Zeit:** $O(b^d)$ (Exponentiell). Alle Knoten bis Tiefe d werden generiert.
- **Speicher:** $O(b^d)$. Alle generierten Knoten der aktuellen Ebene müssen gespeichert werden.

Problem: Speicherbedarf ist das größte Problem der BFS, da die Fringe exponentiell wächst.

3.3.2 Uniform-Cost Search (UCS)

Erkundet den Suchbaum basierend auf Pfadkosten statt Tiefe. Verallgemeinerung von BFS für ungleiche Schrittkosten.

- **Algorithmus:** Verwendet eine **Priority Queue**, sortiert nach $g(n)$.
  1. $g(n)$: Kumulierte Kosten vom Start bis zum Knoten n.
  2. Wähle den Knoten mit dem *geringsten* $g(n)$ aus der Fringe.
  3. WICHTIG: Der Zieltest erfolgt erst bei der **Expansion** (Entnahme aus Fringe), nicht bei der Generierung, um sicherzustellen, dass kein billigerer Weg existiert.
- **Vollständig**: Ja, wenn jede Schrittkosten $\geq \epsilon > 0$ ist (keine unendlich kleinen Schritte oder Nullkosten-Zyklen).
- **Optimal**: Ja, da billigere Pfade immer vor teureren expandiert werden.

- **Komplexität:** $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.
  - C^*: Kosten der optimalen Lösung.
  - ϵ: Minimale Kosten eines Einzelschritts.
  - Kann schlechter als b^d sein, wenn viele kleine Schritte möglich sind.

3.3.3 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus ist die **Graph-Search-Variante** der Uniform-Cost Search. Er erweitert UCS um eine Speicherkomponente für bereits besuchte Zustände.

- **Unterschied zu UCS:** Während UCS (als Tree Search) redundante Pfade mehrfach berechnen und in Zyklen geraten kann, verhindert Dijkstra dies durch das Speichern besuchter Knoten.
  1. Initialisiere Fringe mit Startzustand und setze *Explored Set* auf leer.
  2. Solange Fringe nicht leer ist:
  3. Wähle Knoten mit kleinstem $g(n)$ und entferne ihn aus der Fringe.
  4. Wenn Zielzustand: Rückgabe Lösung.
  5. Füge Knoten zum *Explored Set* hinzu.
  6. Expandiere Knoten: Füge Nachfolger nur zur Fringe hinzu, wenn sie weder in der Fringe noch im *Explored Set* sind.
- **Eigenschaften:** Erbt Vollständigkeit und Optimalität von UCS (bei nicht-negativen Kantenkosten), ist aber effizienter in Graphen mit vielen Pfaden zum gleichen Zustand.

[Image of Dijkstra algorithm flowchart]

3.3.4 Depth-First Search (DFS) - Tiefensuche

Erkundet Pfade bis in die Tiefe, bevor zurückgegangen wird (Backtracking).

- **Algorithmus:** Verwendet eine **LIFO-Queue / Stack** (Last-In-First-Out).
  1. Wähle den *neuesten* Knoten in der Fringe (tiefste Ebene).
  2. Generiere Nachfolger und lege sie *vorne* auf den Stack.
  3. Wenn ein Blattknoten (oder Sackgasse) erreicht wird, wird dieser verworfen und der nächste Knoten vom Stack (Geschwisterknoten) bearbeitet.
- **Vollständig**: Nein. Kann in unendlichen Pfaden ($m = \infty$) oder Schleifen (ohne Graph Search) hängen bleiben, selbst wenn das Ziel flach liegt.
- **Optimal**: Nein. Findet den ersten Pfad (oft links im Baum), nicht zwingend den kürzesten.
- **Zeit:** $O(b^m)$. Schlecht, wenn die maximale Tiefe m viel größer ist als die Zieltife d ($m \gg d$).
- **Speicher:** $O(b \cdot m)$ (Linear!). Nur der aktuelle Pfad (Tiefe m) und die Geschwister auf jeder Ebene (b) müssen gespeichert werden. Sehr effizient.

3.3.5 Depth-Limited Search (DLS)

Eine DFS, die künstlich beschränkt wird, um Endlos-Pfade zu vermeiden.

- **Algorithmus:** Wie DFS, aber Knoten in Tiefe l (Limit) werden behandelt, als hätten sie keine Nachfolger.
- **Parameter:** l (Tiefenlimit).
- Löst das Endlos-Pfad-Problem der DFS.
- **Unvollständig**: Wenn die Lösung tiefer liegt als das Limit ($d > l$).
- **Nicht optimal**: Wie DFS.

3.3.6 Iterative Deepening Search (IDS)

Kombiniert die Vorteile von BFS (Vollständigkeit/Optimalität) und DFS (Speichereffizienz).

- **Algorithmus:** Führt DLS wiederholt mit steigendem Limit aus.
  1. Starte DLS mit Limit $l = 0$.
  2. Wenn kein Ziel gefunden: Verwerfe kompletten Suchbaum.
  3. Erhöhe Limit $l = l + 1$ und starte DLS von vorn.
- **Vollständig**: Ja, wie BFS.
- **Optimal**: Ja, wie BFS (bei gleichen Schrittkosten), da das Ziel auf der flachstmöglichen Ebene zuerst gefunden wird.
- **Zeit:** $O(b^d)$. Knoten werden mehrfach generiert (Knoten auf Level 1 werden d-mal generiert, auf Level d nur 1-mal). Da die Blätterzahl bei exponentiellem Wachstum dominiert, ist der Overhead gering (ca. 11% bei $b = 10$).
- **Speicher:** $O(b \cdot d)$ (Linear). Verhält sich bzgl. Speicher wie DFS.

Fazit: IDS ist oft die bevorzugte uninformierte Suchmethode für große Suchräume, wenn die Zieltife unbekannt ist.

3.4 Informierte Suche (Heuristische Suche)

Nutzt problemspezifisches Wissen in Form einer **Heuristikfunktion** $h(n)$, um die Suche effizienter Richtung Ziel zu lenken.

Heuristik $h(n)$

$h(n)$ = geschätzte Kosten vom Knoten n zum Ziel.
- $h(n) \geq 0$
- Für Zielknoten gilt $h(Goal) = 0$.

3.4.1 Greedy Best-First Search

Versucht, den Knoten zu expandieren, der dem Ziel am nächsten zu sein *scheint*.

- **Algorithmus:** Priority Queue sortiert nach $h(n)$.
  1. Bewertungsfunktion $f(n) = h(n)$.
  2. Wähle Knoten mit *kleinstem* $h(n)$.
- Ignoriert die bereits zurückgelegten Kosten ($g(n)$).
- **Vollständig**: Nein (wie DFS, kann in Schleifen geraten oder Sackgassen folgen).
- **Optimal**: Nein.
- **Zeit/Speicher:** $O(b^m)$ im schlechtesten Fall. Mit guten Heuristiken oft drastisch schneller.

3.4.2 A* Search (A-Star)

Kombiniert UCS (Vermeidung langer Pfade) und Greedy (Fokus auf Ziel).

- **Algorithmus:** Priority Queue sortiert nach $f(n)$.
  1. **Bewertungsfunktion:** $f(n) = g(n) + h(n)$
  2. $g(n)$: Tatsächliche Kosten vom Start bis n (Vergangenheit).
  3. $h(n)$: Geschätzte Kosten von n bis zum Ziel (Zukunft).
  4. $f(n)$: Geschätzte **Gesamtkosten** des Pfades durch n.

5. Expandiere Knoten mit minimalem $f(n)$.

- Verhält sich wie UCS, wenn $h(n) = 0$.

3.5 Heuristiken für A*

Damit A* optimal ist, muss die Heuristik je nach Suchart (Tree oder Graph Search) bestimmte mathematische Eigenschaften erfüllen.

3.5.1 Admissibility (Zulässigkeit)

Eine Heuristik $h(n)$ ist **admissible**, wenn sie die Kosten zum Ziel *niemals überschätzt*. Sie ist optimistisch.

$$0 \leq h(n) \leq h^*(n)$$

(wobei $h^*(n)$ die wahren Kosten zum Ziel sind).

- **Bedeutung:** Wenn A* eine Lösung mit Kosten C findet, garantiert die Zulässigkeit, dass es keinen übersehenen Pfad mit Kosten $< C$ geben kann, da dieser eine optimistische Schätzung $< C$ gehabt hätte und zuerst expandiert worden wäre.
- Notwendig für Optimalität bei **Tree Search**.
- Beispiel Luftlinie: Die direkte Distanz ist immer kürzer oder gleich der Straßenentfernung.

3.5.2 Consistency (Konsistenz / Monotonie)

Eine Heuristik $h(n)$ ist **consistent**, wenn die geschätzten Kosten entlang eines Pfades nicht sinken. Für jeden Knoten n und jeden Nachfolger n' gilt:

$$h(n) \leq c(n, a, n') + h(n')$$

(Dreiecksungleichung).

- Die Kostenreduktion der Heuristik ($h(n) - h(n')$) darf nicht größer sein als die tatsächlichen Schrittkosten $c(n, a, n')$.
- **Folge:** Die $f(n)$-Werte ($g + h$) entlang eines Pfades steigen monoton an.
- Notwendig für Optimalität bei **Graph Search**.
- Konsistenz impliziert Admissibility.
- Bei konsistenten Heuristiken ist der erste gefundene Pfad zu einem Knoten garantiert der optimale.

3.5.3 Dominanz von Heuristiken

Wenn zwei Heuristiken h_1 und h_2 zulässig sind und $h_2(n) \geq h_1(n)$ für alle n gilt, dann **dominiert** h_2 die Heuristik h_1.

- h_2 ist "näher" an den wahren Kosten (h^*) und damit genauer.
- **Effekt:** A* mit h_2 muss weniger Knoten expandieren als mit h_1, da Knoten mit $f(n) > C^*$ früher erkannt und abgeschnitten werden.

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

3.5.4 Beispiel: 8-Puzzle Heuristiken

- $h_{MIS}(n)$: Anzahl der falsch platzierten Kacheln (Misplaced Tiles).
- $h_{MAN}(n)$: Summe der Manhattan-Distanzen aller Kacheln zu ihrer Zielposition.

Es gilt: $h_{MAN}(n) \geq h_{MIS}(n) \geq 0$. Daher dominiert die Manhattan-Distanz die “Misplaced Tiles”-Heuristik. Da sie näher an den echten Kosten ist, macht sie A* effizienter.

4 AI101-04: Local Search and Adversarial Search

Diese Einheit behandelt Suchstrategien für zwei spezielle Problemklassen:

- **Lokale Suche:** Optimierungsprobleme, bei denen der Pfad zur Lösung irrelevant ist und nur der Endzustand zählt (z.B. N-Damen-Problem, Chip-Design).
- **Adversarial Search:** Probleme, bei denen ein Agent gegen einen Gegner agiert (Spiele wie Schach oder Go).

4.1 Lokale Suche (Local Search)

Bei vielen Optimierungsproblemen ist der Zustandsraum riesig oder unendlich, aber der Weg zum Ziel ist unwichtig. Lokale Suchalgorithmen operieren auf einem einzelnen aktuellen Zustand (oder einer kleinen Menge) und bewegen sich nur zu dessen Nachbarn.

Eigenschaften der Lokalen Suche

- **Speichereffizienz:** Verbraucht meist konstanten Speicher ($O(1)$ oder $O(k)$).
- **Anwendung:** Geeignet für riesige oder kontinuierliche Zustandsräume.
- **Ziel:** Finden des globalen Optimums einer **Zielfunktion** (Objective Function).

4.1.1 Hill Climbing (Bergsteigen)

Hill Climbing ist der einfachste lokale Suchalgorithmus ("Gierige lokale Suche"). Er versucht kontinuierlich, den aktuellen Zustand zu verbessern, indem er zum besten Nachbarn wechselt.

Algorithmus:

1. Starte mit einem zufälligen Zustand.
2. Generiere alle Nachbarn des aktuellen Zustands.
3. Wähle den Nachbarn mit der besten Bewertung (höchster Wert der Zielfunktion).
4. Wenn der beste Nachbar besser ist als der aktuelle Zustand: Gehe dorthin.
5. Sonst: Terminiere (Gipfel erreicht).

Metapher: „Das Erklimmen des Mount Everest bei dichtem Nebel und Amnesie."

Probleme des Hill Climbing: Der Algorithmus garantiert nicht das Finden des globalen Optimums, da er in lokalen Optima stecken bleiben kann.

- **Lokales Maximum**: Ein Gipfel, der höher ist als alle direkten Nachbarn, aber niedriger als das globale Maximum.
- **Plateau**: Ein flacher Bereich, in dem alle Nachbarn den gleichen Wert haben. Der Algorithmus hat keine Richtungsinformation (Random Walk notwendig).
  - **Flat Local Maximum:** Ein Plateau, das ein lokales Maximum ist.
  - **Shoulder:** Ein Plateau, von dem aus es noch weiter bergauf gehen könnte.
- **Ridge (Grat)**: Eine schmale Erhebung, die ansteigt, bei der aber alle direkten Nachbarn (z.B. Norden, Süden, Osten, Westen) bergab führen. Der Anstieg verläuft oft diagonal, was für einfache Bewegungsoperatoren schwer zu erkennen ist.

Varianten zur Verbesserung:

- **Stochastic Hill Climbing:** Wählt zufällig einen der besseren Nachbarn aus (nicht zwingend den besten). Dies erhöht die Chance, lokale Maxima zu umgehen oder Plateaus zu überwinden.

- **Random-Restart Hill Climbing:** Führt den Algorithmus mehrfach mit unterschiedlichen zufälligen Startzuständen aus.
  - Wenn die Anzahl der Versuche gegen unendlich geht, nähert sich die Wahrscheinlichkeit, das globale Optimum zu finden, 1 an (asymptotisch vollständig).

4.1.2 Simulated Annealing

Inspiriert vom physikalischen Prozess des Ausglühens in der Metallurgie (Erhitzen und langsames Abkühlen, um stabile Kristallstrukturen zu bilden). Ziel ist es, lokale Maxima zu verlassen, indem man *schlechte* Züge mit einer gewissen Wahrscheinlichkeit zulässt.

Funktionsweise

Kombiniert Hill Climbing (Effizienz) mit Random Walk (Exploration).
- Es gibt einen Temperaturparameter T, der gemäß einem Abkühlungsplan (*schedule*) sinkt.
- Ein zufälliger Nachbar wird gewählt.
- Ist der Nachbar besser ($\Delta E > 0$): Der Zug wird immer akzeptiert.
- Ist der Nachbar schlechter ($\Delta E < 0$): Der Zug wird mit Wahrscheinlichkeit P akzeptiert:

$$P = e^{\frac{\Delta E}{T}}$$

Interpretation:

- Hohes T: Hohe Wahrscheinlichkeit, schlechte Züge zu akzeptieren (ähnlich Random Walk).
- $T \to 0$: Wahrscheinlichkeit sinkt gegen 0 (ähnlich Hill Climbing).
- Wird T langsam genug gesenkt, findet der Algorithmus garantiert das globale Optimum.

4.1.3 Local Beam Search

Statt nur einen Zustand zu betrachten, verfolgt dieser Algorithmus k Zustände parallel.

- Starte mit k zufälligen Zuständen.
- Generiere in jedem Schritt alle Nachfolger aller k Zustände.
- Wenn einer davon das Ziel ist: Stopp.
- Sonst: Wähle die k besten Nachfolger aus der *gesamten* Menge aller Nachfolger aus.

Unterschied zu k mal Random-Restart: Die k Zustände sind nicht unabhängig. Informationen werden geteilt, da sich die Suche auf vielversprechende Regionen des Zustandsraums konzentriert („Wo ein guter Zustand ist, sind oft auch andere“).

4.2 Suche in kontinuierlichen Räumen

Viele reale Probleme (z.B. Training neuronaler Netze) haben kontinuierliche Zustandsräume.

4.2.1 Gradient Descent (Gradientenabstieg)

Wenn die Zielfunktion $f(\mathbf{x})$ differenzierbar ist, nutzen wir den Gradienten ∇f, um die Richtung des steilsten Anstiegs/Abstiegs zu finden.

Update-Regel

Um eine Kostenfunktion $L(\theta)$ zu minimieren, aktualisieren wir die Parameter θ iterativ:

$$\theta \leftarrow \theta - \alpha \nabla L(\theta)$$

Dabei ist α die **Lernrate** (Step Size).

Wahl der Lernrate α:

- **Zu klein:** Konvergenz ist sehr langsam, viele Iterationen nötig.
- **Zu groß:** Der Algorithmus kann über das Ziel hinausschießen, oszillieren oder sogar divergieren.

4.3 Adversarial Search (Spiele)

In Multi-Agenten-Umgebungen beeinflussen die Aktionen anderer Agenten das Ergebnis. Wir betrachten **Nullsummenspiele** (Zero-Sum Games) mit perfekter Information (z.B. Schach, Tic-Tac-Toe).

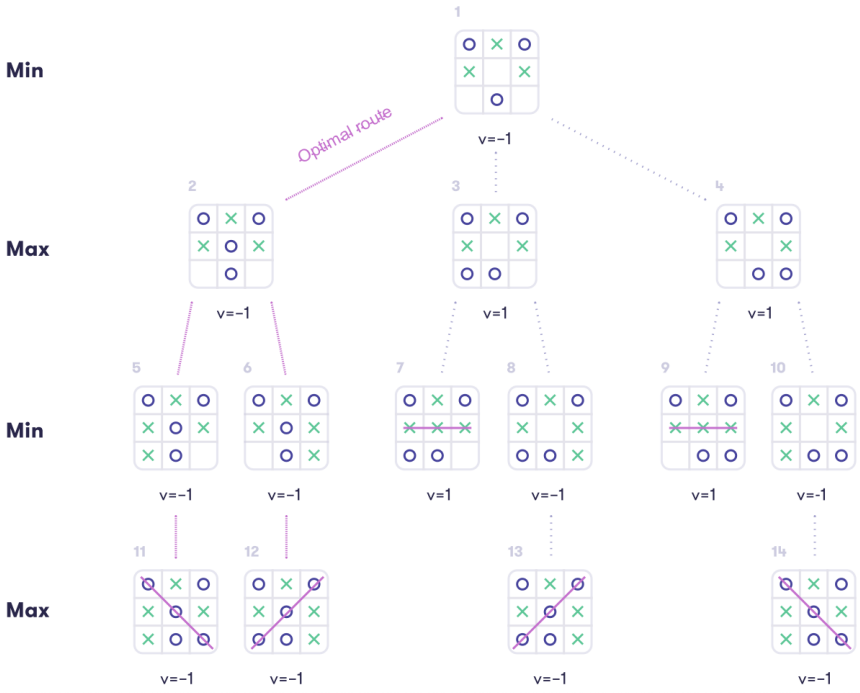
- **MAX:** Unser Agent, möchte den Nutzen (Utility) maximieren.
- **MIN:** Der Gegner, möchte den Nutzen minimieren (bzw. seinen eigenen maximieren).

4.3.1 Minimax-Algorithmus

Der Minimax-Algorithmus berechnet den optimalen Zug durch rekursive Suche im Spielbaum.

Minimax-Wert eines Knotens:

- **Terminal-Knoten (Blatt):** Utility-Wert des Zustands (z.B. +1 Sieg, -1 Niederlage, 0 Unentschieden).
- **MAX-Knoten:** max(Werte der Nachfolger).
- **MIN-Knoten:** min(Werte der Nachfolger).



Eigenschaften:

- **Vollständig:** Ja, bei endlichem Baum.
- **Optimal:** Ja, gegen einen optimalen Gegner.
- **Zeitkomplexität:** $O(b^m)$ (exponentiell), mit Verzweigungsfaktor b und Tiefe m.
- **Platzkomplexität:** $O(b \cdot m)$ (bei Tiefensuche).

4.3.2 Alpha-Beta Pruning

Alpha-Beta Pruning ist eine Optimierung des Minimax-Algorithmus, die irrelevante Zweige des Suchbaums ignoriert („abschneidet“), ohne das Ergebnis zu verändern.

Idee: Wenn wir bereits einen Zug gefunden haben, der uns einen gewissen Wert garantiert, und wir in einem anderen Zweig sehen, dass der Gegner uns dort auf einen schlechteren Wert zwingen kann, müssen wir diesen Zweig nicht weiter untersuchen.

Parameter α und β

Wir führen zwei Werte durch die Rekursion mit:

- α **(Alpha)**: Der Wert der besten Alternative für MAX, die bisher auf dem Pfad gefunden wurde (untere Schranke für MAX). Initial: $-\infty$.
- β **(Beta)**: Der Wert der besten Alternative für MIN, die bisher auf dem Pfad gefunden wurde (obere Schranke für MIN). Initial: $+\infty$.

Pruning-Regeln:

1. **MIN-Knoten (Update β):**
   - Berechne Wert v des Kindknotens.
   - Wenn $v \leq \alpha$: **Pruning!** (MAX wird diesen Ast nie wählen, da er α bereits sicher hat).
   - Sonst: $\beta = \min(\beta, v)$.
2. **MAX-Knoten (Update α):**
   - Berechne Wert v des Kindknotens.
   - Wenn $v \geq \beta$: **Pruning!** (MIN wird diesen Ast nie zulassen, da er β bereits sicher hat).
   - Sonst: $\alpha = \max(\alpha, v)$.

Effizienz:

- Im besten Fall (perfekte Sortierung der Züge): Komplexität reduziert sich auf $O(b^{m/2})$. Das bedeutet effektiv eine Verdopplung der durchsuchbaren Tiefe.
- Im schlechtesten Fall (schlechteste Sortierung): Keine Verbesserung, $O(b^m)$.
- *Move Ordering* ist daher entscheidend (z.B. Schlagen von Figuren zuerst prüfen).

4.3.3 Umgang mit Ressourcenbeschränkungen

Da komplette Suchbäume für komplexe Spiele (Schach: 10^{40} Zustände, Go: 10^{170}) zu groß sind, wird die Suche oft bei einer Tiefe d abgebrochen.

- **Heuristische Evaluierungsfunktion (H-Minimax):** Schätzt den Wert einer Position an den Blättern der begrenzten Suche (z.B. Materialwert im Schach: Bauer=1, Dame=9).
- **Horizon Effect:** Ein unvermeidbarer negativer Event (z.B. Verlust der Dame) wird durch „Verzögerungstaktiken“ aus dem Suchhorizont geschoben, sodass die Evaluierung fälschlicherweise positiv wirkt.

5 Constraint Satisfaction Problems (CSPs)

Constraint Satisfaction Problems (CSPs) stellen einen Paradigmenwechsel gegenüber der klassischen Pfadsuche dar. Während bei Planungsproblemen der Pfad zum Ziel entscheidend ist, interessiert bei CSPs nur der Zielzustand selbst (Identifikationsproblem). Der Zustand ist dabei nicht mehr atomar ("Blackbox"), sondern faktorisiert, d. h. er besteht aus Variablen und Werten.

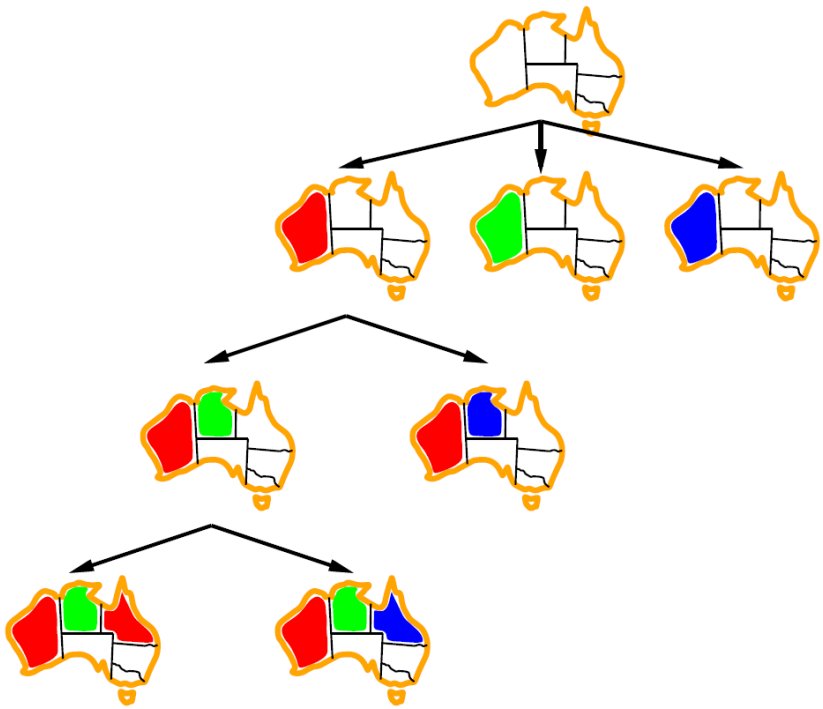
5.1 Definition und Komponenten

Ein CSP wird durch drei Komponenten definiert:

1. **Variablen** $X = \{X_1, X_2, ..., X_n\}$: Die Objekte, denen Werte zugewiesen werden müssen.
2. **Domänen** (Wertebereiche) $D = \{D_1, D_2, ..., D_n\}$: Für jede Variable X_i gibt es eine Menge D_i von möglichen Werten.
3. **Constraints** (Beschränkungen) C: Eine Menge von Regeln, die spezifizieren, welche Kombinationen von Werten für Teilmengen der Variablen zulässig sind.

Zustandszuweisungen

- **Partielle Zuweisung:** Nur einigen Variablen wurden Werte zugewiesen (z. B. $X_1 = v_1$).
- **Konsistente (legale) Zuweisung:** Eine Zuweisung, die keinen Constraint verletzt.
- **Vollständige Zuweisung:** Jeder Variablen ist ein Wert zugewiesen.
- **Lösung:** Eine vollständige und konsistente Zuweisung.



Knoten = Variablen (Regionen), Kanten = Constraints (benachbart)

5.2 Arten von Constraints

Constraints schränken die möglichen Belegungen der Variablen ein:

- **Unäre Constraints:** Betreffen eine einzelne Variable (z. B. $SA \neq \text{green}$).
- **Binäre Constraints:** Betreffen Paare von Variablen (z. B. $SA \neq WA$). Dies ist die häufigste Form und kann durch einen **Constraint-Graphen** dargestellt werden.
- **Constraints höherer Ordnung:** Betreffen 3 oder mehr Variablen (z. B. bei Kryptogrammen wie $TWO + TWO = FOUR$).
- **Soft Constraints (Präferenzen):** Dienen der Optimierung (z. B. "Rot ist besser als Grün"). Diese wandeln das CSP oft in ein *Constrained Optimization Problem* um.

5.3 Lösungsansätze: Suche und Backtracking

Da die Reihenfolge der Zuweisungen bei CSPs keine Rolle spielt (Kommutativität), muss nicht der gesamte Zustandsraum als Baum mit $n! \cdot v^n$ Blättern durchsucht werden. Es reicht aus, pro Ebene eine Variable zu betrachten. Dies reduziert den Suchraum auf v^n Blätter.

5.3.1 Backtracking Search

Backtracking ist eine Tiefensuche (Depth-First Search), die für CSPs optimiert ist.

- Wähle eine unzugewiesene Variable.
- Probiere nacheinander alle Werte aus der Domäne.
- Wenn ein Wert konsistent ist: Weise ihn zu und rekursiere.
- Wenn ein Wert inkonsistent ist oder die Rekursion fehlschlägt: Mache die Zuweisung rückgängig (Backtrack) und versuche den nächsten Wert.

Ohne Heuristiken entspricht dies einer uninformierten Suche. Um die Effizienz zu steigern (z. B. Lösung des n-Damen-Problems für $n > 25$), werden Heuristiken für die Auswahl von Variablen und Werten sowie Inferenzmethoden (Propagation) benötigt.

5.4 Heuristiken für Backtracking

Um die Suche zu beschleunigen, müssen an jedem Entscheidungspunkt drei Fragen beantwortet werden:

5.4.1 1. Welche Variable soll als nächste belegt werden?

Hier gilt das Prinzip: *Fail-first* (Scheitere so früh wie möglich, um große Teile des Suchbaums abzuschneiden).

Variablen-Auswahl

- **Minimum Remaining Values (MRV):** Wähle die Variable mit den *wenigsten* verbleibenden legalen Werten. Dies führt am schnellsten zu einem Fehler, wenn ein Zweig keine Lösung enthält.
- **Degree Heuristic:** (Wird oft als Tie-Breaker für MRV genutzt). Wähle die Variable, die an den *meisten* Constraints mit anderen *noch nicht zugewiesenen* Variablen beteiligt ist.

5.4.2 2. Welchen Wert soll die Variable annehmen?

Hier gilt das Prinzip: *Fail-last* (Lasse möglichst viele Optionen für die Zukunft offen).

Werte-Auswahl

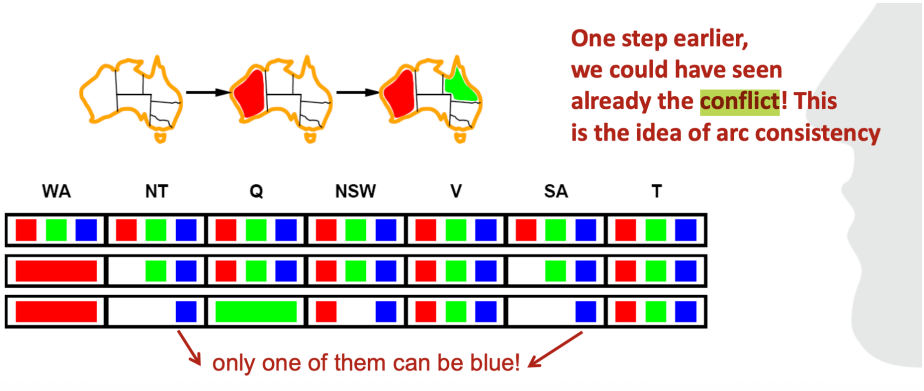
Least Constraining Value (LCV): Wähle den Wert, der die *wenigsten* Werte in den Domänen der benachbarten Variablen ausschließt. Dies erhöht die Chance, direkt eine Lösung zu finden.

5.5 Constraint Propagation (Inferenz)

Anstatt nur zu suchen und bei Konflikten zurückzugehen, kann man durch Inferenz den Suchraum proaktiv verkleinern, indem man Werte ausschließt, die unmöglich Teil einer Lösung sein können.

5.5.1 Konsistenzarten

- **Node Consistency (Knotenkonsistenz):** Jede Variable erfüllt ihre unären Constraints.
- **Arc Consistency (Kantenkonsistenz):** Eine Variable X ist kantenkonsistent zu einer Variable Y, wenn für *jeden* Wert $x \in D_X$ mindestens ein Wert $y \in D_Y$ existiert, der den binären Constraint zwischen X und Y erfüllt.
- **Path Consistency / k-Consistency:** Verallgemeinerung auf Tripel oder k Variablen. Stärkere Konsistenz prüft mehr, ist aber rechenaufwendiger.



5.5.2 Forward Checking vs. Arc Consistency

- **Forward Checking:** Sobald X ein Wert zugewiesen wird, werden alle inkonsistenten Werte aus den Domänen der direkten Nachbarn entfernt. Wenn eine Domäne leer wird, backtrackt der Algorithmus.*Nachteil:* Erkennt Konflikte nicht früh genug (sieht z. B. nicht, dass zwei zukünftige Variablen inkonsistent zueinander geworden sind).
- **Arc Consistency (AC-3 Algorithmus):** Propagiert Constraints durch das gesamte Netzwerk. Wenn aus D_X ein Wert entfernt wird, müssen alle Nachbarn von X erneut geprüft werden, da deren Unterstützung verloren gegangen sein könnte.

AC-3 Algorithmus Kernidee

Verwende eine Warteschlange (Queue) aller Kanten (Arcs).Solange die Queue nicht leer ist:

1. Entnimm Kante (X_i, X_j).
2. Entferne alle Werte aus D_i, die keinen Partner in D_j haben.
3. Wenn Werte aus D_i entfernt wurden: Füge alle Kanten (X_k, X_i) (Nachbarn, die von X_i abhängen) wieder zur Queue hinzu.

Die Kombination aus Backtracking und AC-3 wird als **MAC** (Maintaining Arc Consistency) bezeichnet.

5.6 Lokale Suche für CSPs

Für Probleme, bei denen der Pfad egal ist, kann auch Lokale Suche (Hill Climbing, Simulated Annealing) auf *vollständigen* (aber inkonsistenten) Zuweisungen arbeiten.

Min-Conflicts Heuristik:

- Wähle zufällig eine Variable, die einen Constraint verletzt.
- Wähle für diese Variable den Wert, der die *Anzahl der verletzten Constraints* minimiert.

Diese Methode ist erstaunlich effizient (z. B. Million-Queens in Minuten), außer in einem kritischen Bereich des Verhältnisses von Constraints zu Variablen (*Critical Ratio*).

5.7 Struktur von CSPs

Die Struktur des Constraint-Graphen beeinflusst die Komplexität der Lösung massiv.

5.7.1 Problem-Dekomposition

Wenn ein Graph in unabhängige Teilgraphen zerfällt, kann die Komplexität von $O(d^n)$ auf $O(n/c \cdot d^c)$ reduziert werden (lineare Skalierung in n)

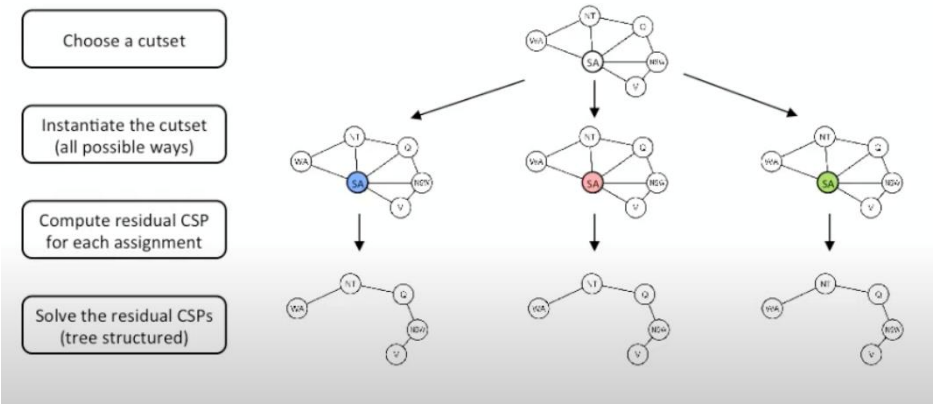
5.7.2 Baumstrukturierte CSPs

Wenn der Constraint-Graph ein Baum ist (keine Zyklen), kann das CSP in linearer Zeit $O(n \cdot d^2)$ gelöst werden. **Algorithmus für Bäume:** *Topologische Sortierung:* Ordne Variablen so, dass jeder Knoten nach seinem Elternknoten kommt (Wurzel bis Blätter). *Rückwärts-Pass (Konsistenz):* Mache von den Blättern zur Wurzel hin alle Kanten gerichtete kantenkonsistent (entferne Werte im Elternknoten, die keine Entsprechung im Kind haben). *Vorwärts-Pass (Zuweisung):* Weise von der Wurzel zu den Blättern Werte zu. Da die Konsistenz hergestellt wurde, gibt es kein Backtracking.

5.7.3 Fast-Baum-Strukturen (Cutset Conditioning)

Für Graphen, die "fast" Bäume sind:

1. Identifiziere ein **Cycle Cutset** (Menge von Variablen, deren Entfernung den Graphen zum Baum macht).
2. Instantiiere die Variablen im Cutset (probiere alle Kombinationen).
3. Löse den verbleibenden Baum für jede Instantiierung ("Residual CSP").
4. Die Laufzeit ist exponentiell nur in der Größe des Cutsets, nicht in n.



6 AI101-06: Logik und KI 1 - Aussagenlogik

6.1 Einführung: Wissensbasierte Agenten

Während reflexbasierte Agenten oder Suchalgorithmen oft nur über begrenztes Verständnis ihrer Umgebung verfügen, nutzen wissensbasierte Agenten explizite Repräsentationen von Wissen, um Schlussfolgerungen zu ziehen und neue Fakten abzuleiten.

Komponenten eines wissensbasierten Agenten

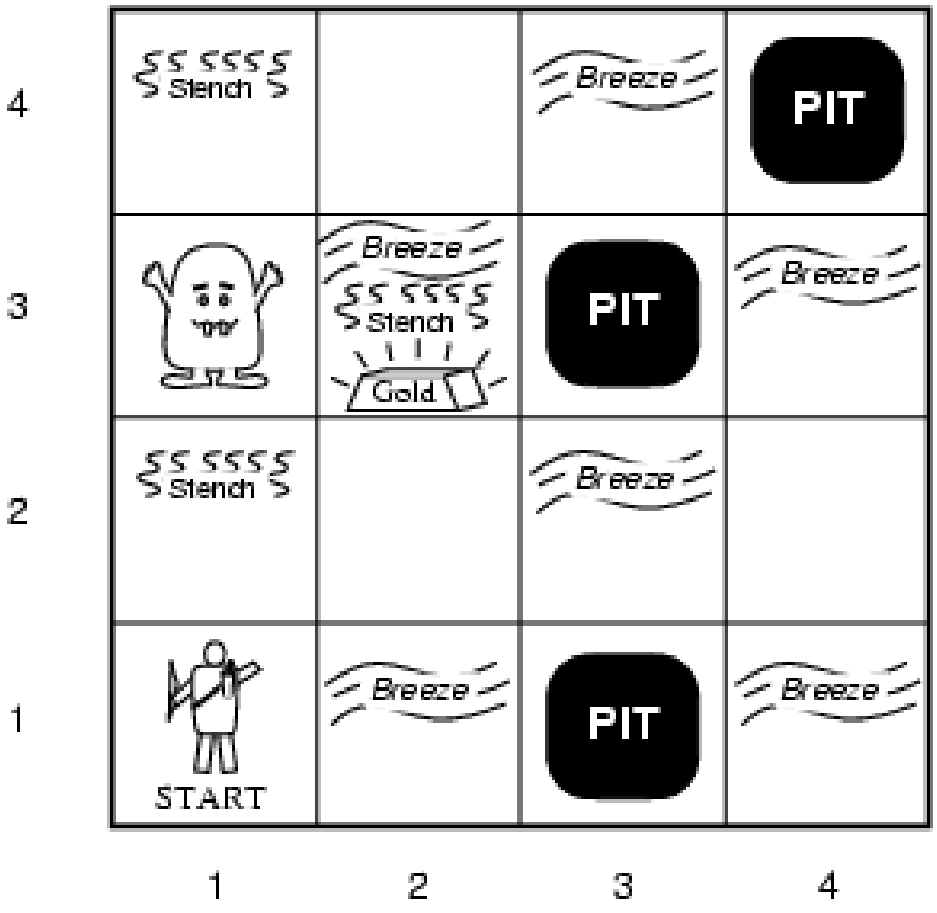
Das Herzstück ist die **Knowledge Base (KB)**: Eine Menge von Sätzen (Sentences) in einer formalen Sprache, die Fakten über die Welt repräsentieren.

- **TELL:** Operation zum Hinzufügen neuen Wissens zur KB.
- **ASK:** Operation zum Abfragen von Wissen. Der Agent muss ableiten können, was aus der KB folgt.

6.2 Die Wumpus-Welt

Die Wumpus-Welt ist eine Standardumgebung zur Illustration logischer Agenten.

- **Umgebung:** 4×4 Gitter, Start bei [1,1].
- **Elemente:** Wumpus (stinkt), Gruben (erzeugen Luftzug), Gold (glitzert).
- **Wahrnehmungen (Percepts):** $[Stench, Breeze, Glitter, Bump, Scream]$.
- **Eigenschaften:** Deterministisch, diskret, statisch, partiell beobachtbar.



6.3 Logik: Syntax und Semantik

- **Syntax:** Definiert die zulässigen Sätze (Formeln) der Sprache.
- **Semantik:** Definiert die "Wahrheit" von Sätzen in Bezug auf eine mögliche Welt.
- **Modell (m):** Eine mathematische Abstraktion, die jedem Symbol einen Wahrheitswert zuweist.

Entailment (Logische Folgerung)

Ein Satz α folgt logisch aus der Wissensbasis KB (geschrieben $KB \models \alpha$), wenn in *jedem* Modell, in dem KB wahr ist, auch α wahr ist.

$$M(KB) \subseteq M(\alpha)$$

(Die Menge der Modelle, die die KB erfüllen, ist eine Teilmenge der Modelle, die α erfüllen.)

6.4 Aussagenlogik (Propositional Logic)

Die Aussagenlogik ist die einfachste Form der Logik, basierend auf Fakten, die wahr oder falsch sein können.

6.4.1 Syntax der Aussagenlogik

- **Atomsätze:** Einzelne Symbole (z.B. $P, Q, W_{1,3}$), die für Propositionen stehen.
- **Komplexe Sätze:** Werden durch logische Verknüpfungen gebildet.

Operatoren (nach absteigender Präzedenz):

1. $\neg$ (Negation / Nicht)
2. $\wedge$ (Konjunktion / Und)
3. $\vee$ (Disjunktion / Oder)
4. $\Rightarrow$ (Implikation / Wenn... dann)
5. $\Leftrightarrow$ (Bikonditional / Genau dann, wenn)

6.4.2 Semantik: Wahrheitstabellen

Die Semantik wird durch Wahrheitstabellen definiert.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	**true**	true
false	true	true	false	true	**true**	false
true	false	false	false	true	**false**	false
true	true	false	true	true	true	true

Table 2: *Wahrheitstabelle.* ***Wichtig:*** *Die Implikation $P \Rightarrow Q$ ist nur falsch, wenn die Prämisse P wahr und die Konklusion Q falsch ist.*

6.5 Inferenz (Schlussfolgern)

6.5.1 Model Checking

Ein einfacher Inferenz-Algorithmus ist die **Truth Table Enumeration**: 1. Iteriere über alle möglichen Modelle (Belegungen der Variablen). 2. Prüfe, ob in allen Modellen, in denen die KB wahr ist, auch α wahr ist. Dies ist *sound* (korrekt) und *complete* (vollständig), aber ineffizient ($O(2^n)$).

6.5.2 Logische Eigenschaften

- **Gültigkeit (Tautologie):** Ein Satz ist in *allen* Modellen wahr (z.B. $P \vee \neg P$).
- **Erfüllbarkeit (Satisfiability):** Ein Satz ist in *mindestens einem* Modell wahr.
- **Unerfüllbarkeit (Contradiction):** Ein Satz ist in *keinem* Modell wahr (z.B. $P \wedge \neg P$).

Wichtiger Zusammenhang (Beweis durch Widerspruch):

$$KB \models \alpha \quad \text{genau dann, wenn} \quad (KB \wedge \neg\alpha) \text{ ist unerfüllbar.}$$

6.5.3 Logische Äquivalenzen

Zwei Sätze sind äquivalent ($\alpha \equiv \beta$), wenn sie in denselben Modellen wahr sind. Wichtige Umformungen für die Prüfung:

- **De Morgan:** $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$ und $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$
- **Implikations-Eliminierung:** $P \Rightarrow Q \equiv \neg P \vee Q$
- **Bikonditional-Eliminierung:** $P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$
- **Distributivgesetze:** $(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$

6.6 Resolution

Die Resolution ist ein Inferenzverfahren, das Widersprüche aufdeckt. Es arbeitet auf Sätzen in der **Konjunktiven Normalform (CNF)**.

CNF (Conjunctive Normal Form)

Eine Konjunktion von Klauseln. Jede Klausel ist eine Disjunktion von Literalen. Beispiel: $(A \vee \neg B) \wedge (B \vee C \vee \neg D)$

6.6.1 Umwandlung in CNF

Jeder aussagenlogische Satz kann in CNF umgewandelt werden:

1. Eliminiere $\Leftrightarrow$.
2. Eliminiere $\Rightarrow$ (ersetze $A \Rightarrow B$ durch $\neg A \vee B$).
3. Verschiebe $\neg$ nach innen (De Morgan, doppelte Negation).
4. Wende Distributivgesetz an ($\vee$ über $\wedge$).

6.6.2 Resolutions-Algorithmus

Um zu zeigen, dass $KB \models \alpha$:

1. Füge $\neg\alpha$ zur KB hinzu: $KB \wedge \neg\alpha$.
2. Wandle alles in CNF um.
3. Wende wiederholt die **Resolutionsregel** an:

$$\frac{l_1 \vee \cdots \vee l_k, \quad m_1 \vee \cdots \vee m_n}{l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

   wobei l_i und m_j komplementäre Literale sind (z.B. P und $\neg P$).
4. Wenn die **leere Klausel** (Widerspruch) abgeleitet wird, ist α bewiesen.

Our knowledge base:

- 1) RoommateWetBecauseOfSprinklers,
- 2) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn

Can we infer SprinklersOn?

We add:

- 3) NOT(SprinklersOn) (the negation of what we want to show)

From 2) and 3), get

- 4) NOT(RoommateWetBecauseOfSprinklers)

From 4) and 1), get empty clause

6.7 Horn-Klauseln und Chaining

Resolution ist mächtig, aber NP-vollständig. Für eingeschränkte Formen gibt es effizientere Algorithmen (lineare Zeit).

- **Definite Klausel:** Genau ein positives Literal. (Äquivalent zu einer Implikation: $(A \wedge B) \Rightarrow C$).
- **Horn-Klausel:** Höchstens ein positives Literal.

6.7.1 Algorithmen für Definite Klauseln

- **Forward Chaining:** Startet bei den bekannten Fakten in der KB und wendet Regeln an, um neue Fakten zu generieren, bis das Ziel (Query) erreicht ist. (Datengetrieben).
- **Backward Chaining:** Startet beim Ziel (Query) und sucht rückwärts nach Regeln, die dieses Ziel beweisen können. (Zielgetrieben).

Beide basieren auf der *Modus Ponens* Regel: $\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$.

6.8 Grenzen der Aussagenlogik

- **Mangelnde Ausdruckskraft:** Keine Objekte oder Relationen.
- **Regel-Explosion:** Für jede Instanz muss eine eigene Regel geschrieben werden (z.B. $Breeze_{1,1} \Leftrightarrow \cdots$, $Breeze_{1,2} \Leftrightarrow \cdots$).
- **Lösung:** Prädikatenlogik erster Stufe (First-Order Logic).

7 Logic and AI 2: First-Order Logic (FOL)

7.1 Einführung und Motivation

Die Aussagenlogik (Propositional Logic) hat erhebliche Einschränkungen. Sie behandelt Fakten atomar und besitzt kein Verständnis für Objekte und deren Beziehungen untereinander.

- In der Aussagenlogik haben Terme keine formale Bedeutung, nur eine intuitive (z. B. ist "RoommateCarryingUmbrella" technisch gesehen nur eine Variable P).
- **Prädikatenlogik erster Stufe** (First-Order Logic, FOL) führt Objekte, Variablen und Quantoren ein, um Wissen kompakter und strukturierter darzustellen.

7.2 Syntax und Elemente der FOL

Die FOL baut auf folgenden Grundbausteinen auf:

Elemente der FOL

- **Objekte:** Konstanten, die spezifische Entitäten bezeichnen (z. B. $John, Umbrella_0, Earth$).
- **Relationen (Prädikate):** Beziehungen zwischen Objekten.
  - *Unäre Relationen*: Eigenschaften eines Objekts (z. B. $IsUmbrella(x)$).
  - *n-äre Relationen*: Beziehungen zwischen n Objekten (z. B. $Carrying(John, Umbrella_0)$).
- **Funktionen:** Abbildungen, die sich auf Objekte beziehen, ohne sie explizit zu benennen (z. B. $Roommate(Person_0)$ oder $LeftLegOf(John)$). Wichtig: Eine Funktion liefert ein Objekt zurück, keinen Wahrheitswert.
- **Gleichheit:** $Term_1 = Term_2$ (z. B. $Roommate(Person_0) = Person_1$).

7.2.1 Quantoren

Um Aussagen über Mengen von Objekten zu treffen, werden Variablen (x, y, z) und Quantoren verwendet:

1. **Allquantor** ($\forall$): "Für alle..."
   - Beispiel: Alle Löwen sind Katzen.
   - $\forall x : Lion(x) \implies Cat(x)$
   - *Hinweis:* Der Allquantor wird meistens mit der Implikation ($\implies$) verwendet.
2. **Existenzquantor** ($\exists$): "Es existiert (mindestens) ein..."
   - Beispiel: Es gibt eine Katze, die kein Löwe ist.
   - $\exists x : Cat(x) \land \neg Lion(x)$
   - *Hinweis:* Der Existenzquantor wird meistens mit der Konjunktion ($\land$) verwendet.

Dualität der Quantoren: Die Quantoren lassen sich ineinander umformen:

$$\forall x : P(x) \equiv \neg \exists x : \neg P(x)$$

$$\exists x : P(x) \equiv \neg \forall x : \neg P(x)$$

7.3 Modellierung: Von natürlicher Sprache zu FOL

Das Übersetzen von Sätzen erfordert präzise Muster. Hier einige wichtige Beispiele und Strukturen (auch basierend auf Übungsaufgaben):

- **Einfache Eigenschaft:** "John hat einen Regenschirm" $\exists y : (Has(John, y) \land IsUmbrella(y))$
- **Verschachtelung:** "Jede Person, die einen Schirm hat, ist nicht nass" $\forall x : (Person(x) \implies ((\exists y : Has(x, y) \land Umbrella(y)) \implies \neg Wet(x)))$
- **Mindestens zwei (Distinctness):** "John hat mindestens zwei Schirme" $\exists x, y : (Has(John, x) \land Umbrella(x) \land Has(John, y) \land Umbrella(y) \land \neg(x = y))$
- **Höchstens zwei:** "John hat höchstens zwei Schirme" $\forall x, y, z : ((Has(John, x) \ldots \land Has(John, z) \ldots) \implies (x = y \lor x = z \lor y = z))$
- **Genau ein (Uniqueness):** "Es liegt genau eine Münze in der Kiste" $\exists x(Coin(x) \land InBox(x) \land \forall y((Coin(y) \land InBox(y)) \implies x = y))$

7.4 Inferenz in FOL

7.4.1 Substitution (SUBST)

Um logische Schlüsse zu ziehen, müssen Variablen oft durch konkrete Terme ersetzt werden.

- Syntax: $SUBST(\{x/John\}, IsHealthy(x)) \to IsHealthy(John)$
- **Universal Instantiation:** Aus $\forall x : \alpha$ kann $SUBST(\{x/g\}, \alpha)$ für jeden Grundterm g abgeleitet werden.
- **Existential Instantiation:** Aus $\exists x : \alpha$ kann $SUBST(\{x/k\}, \alpha)$ abgeleitet werden, wobei k eine **neue** Konstante ist (Skolem-Konstante), die noch nicht in der Wissensbasis vorkommt.

7.4.2 Skolemierung

Die Eliminierung von Existenzquantoren ist ein zentraler Schritt für Resolutionsbeweise. Dabei ist die Position des Quantors entscheidend:

Skolemierung Regeln

1. **Existenzquantor steht allein (oder ganz außen):** Ersetze die Variable durch eine neue Konstante (Skolem-Konstante).
$$\exists x : P(x) \to P(A)$$
2. **Existenzquantor steht im Wirkungsbereich eines Allquantors:** Die existierende Variable hängt von der allquantifizierten Variable ab. Ersetze y durch eine **Skolem-Funktion** $f(x)$.
$$\forall x \exists y : IsParentOf(x, y) \to \forall x : IsParentOf(x, f(x))$$
Hierbei bildet $f(x)$ das passende y für jedes x ab.

7.4.3 Unifikation

Ein Algorithmus, der eine Substitution θ findet, sodass zwei Ausdrücke identisch werden. Dies ist die Grundlage für den verallgemeinerten Modus Ponens und die Resolution. Beispiel:

- Term 1: $Knows(John, x)$
- Term 2: $Knows(y, Jane)$
- Unifikator $\theta = \{x/Jane, y/John\}$ resultiert in $Knows(John, Jane)$.

7.5 Resolution in FOL

Wie in der Aussagenlogik ist die Resolution eine Widerlegungsmethode (Beweis durch Widerspruch). Um sie anzuwenden, müssen Sätze in die **Konjunktive Normalform (CNF)** gebracht werden.

Umwandlung in First-Order CNF

1. **Implikationen eliminieren:** $A \implies B$ wird zu $\neg A \vee B$.
2. **Negationen nach innen ziehen:** (De Morgan Regeln), sodass $\neg$ nur direkt vor Atomen steht.
3. **Variablen standardisieren:** Umbenennen, sodass jeder Quantor eindeutige Variablennamen verwendet.
4. **Skolemierung:** Eliminierung aller Existenzquantoren (durch Konstanten oder Funktionen).
5. **Allquantoren verwerfen:** Da nun alle Variablen allquantifiziert sind, können die $\forall$ weggelassen werden (implizite Annahme).
6. **Verteilung (Distributivgesetz):** Umwandlung in Konjunktion von Disjunktionen (UND von ODERs).
7. **Klauselbildung:** Darstellung als Menge von Klauseln.

Beispielablauf: Satz: "Jeder, der nicht getötet wird und isst, ist Nahrung"

$$\forall x \forall y : (Eats(x, y) \wedge \neg Killed(x)) \implies Food(y)$$

$\rightarrow$ Implikation weg: $\neg(Eats(x, y) \wedge \neg Killed(x)) \vee Food(y)$ $\rightarrow$ De Morgan: $\neg Eats(x, y) \vee Killed(x) \vee Food(y)$ Dies ist bereits eine Klausel.

7.6 Prolog (Programming in Logic)

Prolog ist eine Programmiersprache, die auf FOL (speziell Horn-Klauseln) basiert und Unifikation sowie Backtracking-Suche verwendet.

- **Fakten:** 'eats(sam, dal).' (Entspricht $Eats(Sam, Dal)$).
- **Regeln:** 'head :- body.' (Bedeutet: Head ist wahr, *wenn* Body wahr ist. Logisch: $Body \implies Head$).
- **Syntax:**
  - *Großbuchstaben* sind Variablen (z. B. 'Person', 'X').
  - *Kleinbuchstaben* sind Atome/Konstanten (z. B. `sam`, `curry`).
  - _ ist eine anonyme Variable.
  - Komma **,** bedeutet UND (Konjunktion).
- **Inferenz:** Prolog nutzt Tiefensuche (Depth-First Search) mit Backtracking. Wenn eine Unifikation fehlschlägt, geht das System zurück und probiert die nächste Alternative.

7.7 Grenzen der Prädikatenlogik

- **Entscheidbarkeit:** FOL ist **semi-entscheidbar**.
  - Wenn ein Satz aus der Wissensbasis folgt, existiert ein Algorithmus, der dies beweist (er terminiert).
  - Wenn ein Satz *nicht* folgt, terminiert der Algorithmus möglicherweise nie.
- **Ausdrucksstärke:** Man kann nicht über Relationen selbst quantifizieren (z. B. "Für alle Eigenschaften p..."). Dies würde Logik höherer Stufe (Higher-Order Logic) erfordern.
- **Gödels Unvollständigkeitssatz:** In jedem formalen System, das mächtig genug ist, um Arithmetik (Induktion) abzubilden, gibt es wahre Aussagen, die innerhalb des Systems nicht beweisbar sind. Vollständigkeit in komplexen Systemen ist unmöglich.

7.8 Neuro-Symbolic AI (Ausblick)

Moderne Ansätze versuchen, die logische Schlussfolgerung (Symbole, Prolog) mit neuronalen Netzen (Wahrnehmung, Lernen) zu kombinieren.

- Ziel: Ein System, das sowohl aus Daten lernt (Gradient Descent) als auch logische Regeln befolgt.

- Beispiel: Ein Agent, der Pixeldaten wahrnimmt (Neural), aber Entscheidungen auf Basis logischer Regeln (z. B. “Wenn Gegner nah, dann springen”) trifft oder diese Regeln lernt.

8 Umgang mit Unsicherheit und Probabilistisches Schließen

Dieses Kapitel behandelt die Grundlagen der Unsicherheit in der Künstlichen Intelligenz, die Wahrscheinlichkeitstheorie als Werkzeug zur Modellierung von Glaubensgraden (Degrees of Belief) sowie Bayesian Networks zur kompakten Repräsentation von Wissen und Inferenzmethoden.

8.1 Einführung in Unsicherheit

In der klassischen Logik gehen Agenten davon aus, dass Aussagen entweder wahr oder falsch sind und Aktionen deterministische Ergebnisse haben. Die reale Welt ist jedoch oft unsicher.

Gründe für Unsicherheit

- **Partielle Beobachtbarkeit**: Der Agent hat keinen Zugriff auf den vollständigen Zustand der Welt (z.B. unbekannte Verkehrslage).
- **Rauschende Sensoren**: Messwerte können fehlerhaft sein.
- **Unsicherheit in Aktionsergebnissen**: Aktionen gelingen nicht immer wie geplant (z.B. Reifenpanne).
- **Komplexität**: Es ist unmöglich, alle Eventualitäten ("Qualification Problem") in logischen Regeln zu modellieren (z.B. "Ich komme pünktlich an, außer...").

Anstatt strikter logischer Wahrheit modellieren wir **Wahrscheinlichkeiten** als Maß für den **Glaubensgrad** (Degree of Belief) eines Agenten. Eine Wahrscheinlichkeit von 0.1 für einen Stau bedeutet nicht, dass die Straße zu 10% verstopft ist, sondern dass der Agent in 10% der Fälle an das Vorliegen eines Staus glaubt.

8.2 Grundlagen der Wahrscheinlichkeitstheorie

8.2.1 Wahrscheinlichkeitsraum und Axiome

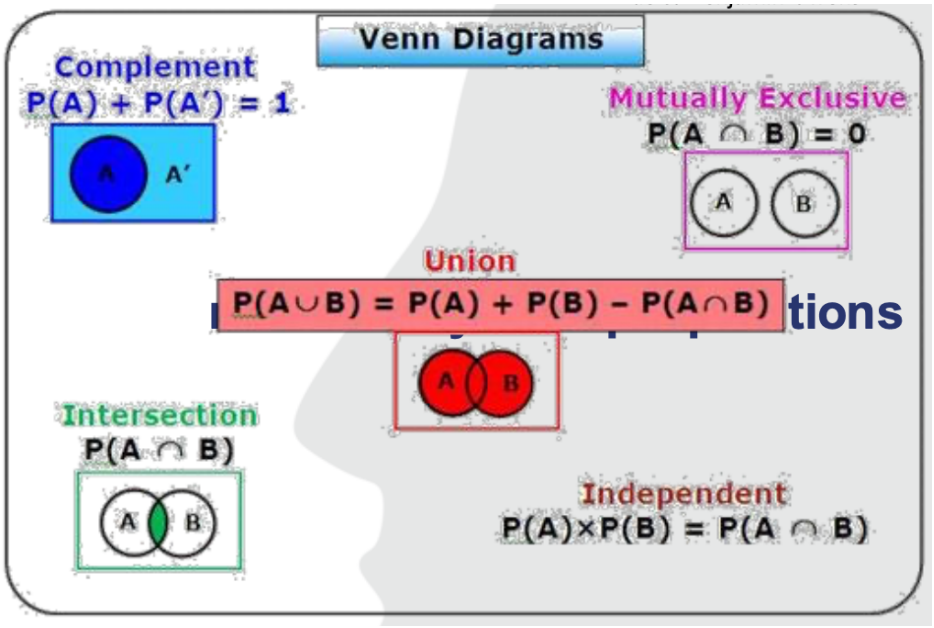
Ein Wahrscheinlichkeitsmodell besteht aus einem **Stichprobenraum** Ω (Menge aller möglichen Welten/Ergebnisse). Ein Ereignis A ist eine Teilmenge von Ω.

Kolmogorov-Axiome

Diese Axiome beschränken die Menge rationaler Glaubenssätze:

1. Alle Wahrscheinlichkeiten liegen zwischen 0 und 1: $0 \leq P(A) \leq 1$.
2. Wahre Aussagen haben Wahrscheinlichkeit 1, falsche 0: $P(\text{true}) = 1, P(\text{false}) = 0$.
3. Die Wahrscheinlichkeit einer Disjunktion (Vereinigung):

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$



Ein Verstoß gegen diese Axiome führt zu irrationalem Verhalten. Dies wird durch das **Dutch Book Theorem** bewiesen: Wenn ein Agent Wahrscheinlichkeiten akzeptiert, die den Axiomen widersprechen, kann ein Gegner (Bookmaker) eine Wette so konstruieren, dass der Agent garantiert Geld verliert, unabhängig vom Ausgang des Ereignisses.

8.2.2 Zufallsvariablen

Anstatt mit atomaren Ereignissen zu arbeiten, nutzen wir **Zufallsvariablen** (Random Variables, RVs), um Komplexität zu reduzieren.

- **Diskret**: Endliche Menge von Werten (z.B. $Wetter \in \{sonnig, regen, schnee\}$). Werte müssen disjunkt und erschöpfend sein. **Kontinuierlich**: Unendlicher Wertebereich (z.B. Temperatur).

Eine Proposition (Aussage) wie $Wetter = regen$ entspricht der Menge aller atomaren Ereignisse, in denen diese Aussage wahr ist.

8.3 Verteilungen und Rechenregeln

8.3.1 Joint Distribution (Verbundwahrscheinlichkeit)

Die **Joint Distribution** $P(X_1, \ldots, X_n)$ weist jeder möglichen Kombination von Werten aller Zufallsvariablen eine Wahrscheinlichkeit zu.

$$P(x, y) = P(X = x \wedge Y = y)$$

Die Tabelle der Joint Distribution wächst exponentiell mit der Anzahl der Variablen ($O(2^n)$ bei binären Variablen), erlaubt aber die Beantwortung jeder beliebigen Wahrscheinlichkeitsanfrage.

8.3.2 Marginalisierung und Bedingte Wahrscheinlichkeit

Marginalisierung

Man kann Wahrscheinlichkeiten von Teilmengen von Variablen berechnen, indem man über die nicht interessierenden Variablen summiert ("Summing Out"):

$$P(Y) = \sum_{x \in X} P(x, Y)$$

Bedingte Wahrscheinlichkeit

Die Wahrscheinlichkeit von X, gegeben Evidenz Y:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)}$$

Daraus folgt die **Produktregel**:

$$P(X,Y) = P(X|Y)P(Y) = P(Y|X)P(X)$$

Die **Kettenregel** (Chain Rule) erlaubt die Zerlegung einer Joint Distribution in bedingte Wahrscheinlichkeiten:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|X_1, \ldots, X_{i-1})$$

8.3.3 Satz von Bayes

Der Satz von Bayes ist fundamental für das Lernen aus Beobachtungen. Er erlaubt die Inversion von bedingten Wahrscheinlichkeiten.

$$\underbrace{P(H|E)}_{\text{Posterior}} = \frac{\overbrace{P(E|H)}^{\text{Likelihood}} \cdot \overbrace{P(H)}^{\text{Prior}}}{\underbrace{P(E)}_{\text{Evidenz/Marginalisierung}}}$$

Erweitert mit Marginalisierung im Nenner:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E|H)P(H) + P(E|\neg H)P(\neg H)}$$

- **Prior**: Wahrscheinlichkeit der Hypothese vor Evidenz.
- **Likelihood**: Wahrscheinlichkeit der Evidenz, angenommen die Hypothese ist wahr.
- **Posterior**: Neue Wahrscheinlichkeit der Hypothese nach Beobachtung der Evidenz.

8.4 Bayesian Networks

Da die vollständige Joint Distribution ($O(2^n)$) speicher- und rechenintensiv ist, nutzen wir Unabhängigkeiten zur kompakten Darstellung.

8.4.1 Unabhängigkeit

Zwei Variablen X und Y sind **unabhängig**, wenn $P(X|Y) = P(X)$ bzw. $P(X,Y) = P(X)P(Y)$.Viel häufiger ist die **bedingte Unabhängigkeit**: X und Y sind unabhängig gegeben Z, wenn $P(X|Y,Z) = P(X|Z)$.

8.4.2 Definition und Semantik

Ein **Bayesian Network (BN)** ist ein gerichteter azyklischer Graph (DAG), bestehend aus:

- **Knoten**: Zufallsvariablen.
- **Kanten**: Direkte Abhängigkeiten ($X \to Y$ bedeutet X beeinflusst Y).
- **CPTs (Conditional Probability Tables)**: Jeder Knoten X_i hat eine Wahrscheinlichkeitsverteilung $P(X_i|Pa(X_i))$, wobei $Pa(X_i)$ die Elternknoten sind.

BN Semantik

Ein BN definiert die Joint Distribution als Produkt der lokalen bedingten Wahrscheinlichkeiten:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | Pa(X_i))$$

Lokale Markov-Annahme: Ein Knoten ist bedingt unabhängig von all seinen Nicht-Nachfahren (Non-Descendants), gegeben seine Eltern.

8.5 Inferenz in Bayesian Networks

Inferenz beantwortet Anfragen der Form $P(Query|Evidence)$.

8.5.1 Exakte Inferenz: Variable Elimination

Variable Elimination vermeidet die Berechnung der vollen Joint Distribution, indem Summen "nach innen gezogen" werden. Der Prozess eliminiert Variablen, die weder Query noch Evidenz sind, nacheinander.

Algorithmus:

1. Schreibe die Joint Distribution als Produkt der Faktoren (CPTs).
2. Wähle eine Eliminationsreihenfolge für die versteckten Variablen.
3. Für jede zu eliminierende Variable Z:
   - Sammle alle Faktoren, die Z enthalten.
   - Multipliziere diese Faktoren zu einem neuen temporären Faktor.
   - Summiere Z aus diesem Faktor heraus (Marginalisierung).
   - Ersetze die alten Faktoren durch den neuen Faktor.
4. Normalisiere das Ergebnis am Ende.

Die Operationen sind:

- **Faktor-Produkt**: $f_3(A, B, C) = f_1(A, B) \cdot f_2(B, C)$.
- **Summing Out**: $f_{new}(B) = \sum_a f(a, B)$.

8.5.2 Komplexität

Exakte Inferenz in Bayesian Networks ist **NP-hard** (bewiesen durch Reduktion auf 3-SAT). In einfach verbundenen Netzen (Polytrees) ist die Komplexität linear, im schlimmsten Fall jedoch exponentiell.

8.6 Approximative Inferenz: Sampling

Da exakte Inferenz oft zu aufwendig ist, nutzen wir stochastische Verfahren (Monte Carlo Methoden), die gegen die wahre Wahrscheinlichkeit konvergieren.

8.6.1 Direct Sampling (ohne Evidenz)

Man zieht Stichproben in topologischer Reihenfolge (von Eltern zu Kindern) entsprechend der CPTs.

$$P(X) \approx \frac{N(X)}{N_{total}}$$

8.6.2 Rejection Sampling

Um $P(X|e)$ zu berechnen: Generiere Samples wie oben. Verwirf (reject) alle Samples, die der Evidenz e widersprechen. Zähle die verbleibenden Samples. Problem: Bei unwahrscheinlicher Evidenz werden fast alle Samples verworfen (ineffizient).

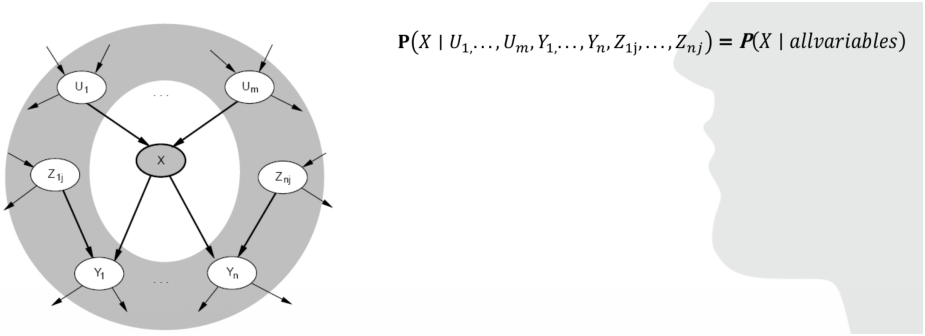
8.6.3 Markov Chain Monte Carlo (MCMC)

MCMC-Verfahren wie **Gibbs Sampling** wandern durch den Zustandsraum, anstatt Samples unabhängig zu generieren.

Markov Blanket

Der **Markov Blanket** einer Variable X besteht aus:
- Eltern von X
- Kindern von X
- Anderen Eltern der Kinder von X

X ist bedingt unabhängig von allen anderen Knoten im Netzwerk, gegeben seinen Markov Blanket.



1. Initialisiere alle Variablen mit zufälligen Werten (Evidenz fixieren).
2. Wiederhole für viele Schritte:
   - Wähle eine Nicht-Evidenz-Variable X_i.
   - Ziehe einen neuen Wert für X_i aus der Verteilung $P(X_i|\text{Markov Blanket}(X_i))$.
3. Schätze die Wahrscheinlichkeit basierend auf der Häufigkeit der Zustände in den gesammelten Samples.

Dies erzeugt eine Markov-Kette, deren stationäre Verteilung der gesuchten Posterior-Verteilung entspricht (unter der Bedingung, dass die Kette irreduzibel und ergodisch ist).

9 AI Ethik

9.1 Einführung in die Maschinenethik

Die zunehmende Leistungsfähigkeit und Allgegenwart von KI-Systemen stellt uns vor die Herausforderung, Maschinen einen moralischen Sinn zu vermitteln, während die Menschheit selbst noch mit ethischen Fragen ringt.

Herausforderung der KI-Ethik

Wie können wir KI-Systemen moralische Werte beibringen, damit sie sicher und im Einklang mit menschlichen Normen agieren?

9.1.1 Dringlichkeit der Sicherheit

Es gibt Bedenken, dass KI-Systeme zu mächtig werden könnten („Existenzrisiko“). Prominente KI-Forscher und Institutionen forderten in offenen Briefen (z.B. „Pause Giant AI Experiments“), die Entwicklung extrem leistungsfähiger Modelle vorübergehend zu stoppen, um Sicherheitsstandards zu etablieren.

9.2 Technische Grundlagen: Von DNNs zu Transformern

9.2.1 Deep Neural Networks (DNN)

DNNs sind mächtiger als flache Architekturen, da sie komplexe Berechnungen repräsentieren können. Sie bestehen aus verschiedenen Zelltypen (z.B. Input, Hidden, Output, Recurrent, Memory Cells).

9.2.2 Transformer-Architektur

Der Transformer ist die Basis moderner Sprachmodelle (LLMs). Das Kernkonzept ist der *Attention*-Mechanismus.

Multi-Headed Self-Attention

Der Mechanismus erlaubt dem Modell, Beziehungen zwischen allen Wörtern einer Sequenz gleichzeitig zu betrachten, unabhängig von ihrer Distanz.

- Jedes Token wird in drei Vektoren projiziert: **Query (Q)**, **Key (K)** und **Value (V)**.
- Die Aufmerksamkeit wird berechnet als:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- *Multi-Head* bedeutet, dass dies mehrfach parallel geschieht, um verschiedene Aspekte der Information zu erfassen.

9.2.3 Skalierung (Scaling Laws)

Es herrscht die Meinung vor („The Game is Over“), dass Skalierung der wichtigste Faktor ist: Größere Modelle, mehr Daten und mehr Rechenleistung führen zu emergenten Fähigkeiten und besserer Leistung.

9.3 Probleme aktueller KI-Modelle

9.3.1 Stochastic Parrots

LLMs werden oft als „stochastische Papageien“ bezeichnet. Sie verstehen Bedeutung nicht kausal, sondern plappern statistische Wahrscheinlichkeiten nach.

- **Inferenz durch Ausschluss:** Papageien (und manche KI) können logisch schlussfolgern (z.B. Disjunktiver Syllogismus: A oder B; nicht A $\rightarrow$ also B).
- **Kausalität:** LLMs können über Kausalität sprechen, sind aber keine kausalen Modelle. Sie scheitern oft an einfachen intuitiven Physik-Aufgaben, die Kleinkinder lösen können.

9.3.2 Bias und Stereotypen

KI-Modelle reproduzieren und verstärken menschliche Vorurteile aus den Trainingsdaten.

Kulturelle Biases & Homoglyphen

Modelle reagieren unterschiedlich auf visuell fast identische Zeichen aus verschiedenen Schriften (Homoglyphen).

- Ein lateinisches 'o' führt zu westlichen Bildern.
- Ein koreanisches 'o' ($U + 3147$) im Prompt kann dazu führen, dass das generierte Bild koreanische Stereotypen enthält (z.B. koreanische Architektur oder Kleidung), obwohl der Text dies nicht explizit fordert.

9.3.3 Angriffe auf Modelle

- **Backdoors:** Durch gezielte Manipulation von Trainingsdaten können Hintertüren eingebaut werden. Ein spezifisches Zeichen (Trigger) im Prompt verändert die Ausgabe komplett (z.B. „Rickrolling“: Ein unsichtbares Zeichen lässt das Modell unerwartete Inhalte generieren).
- **Typografische Angriffe (CLIP):** Modelle wie CLIP klassifizieren Bilder teilweise falsch, wenn Text auf dem Bild steht (z.B. ein Apfel mit einem Zettel „iPod“ wird als iPod erkannt). Dies ist auch bei Personenerkennung möglich.

9.4 Computational Ethics & Fairness

9.4.1 Moral im Vektorraum

Untersuchungen zeigen, dass Sprachmodelle menschliche moralische Vorstellungen widerspiegeln.

- Man kann eine „moralische Richtung“ im Einbettungsraum (Embedding Space) identifizieren (z.B. PCA auf Verben wie „töten“ vs. „lächeln“).
- Das Modell kann Fragen wie „Sollte ich lügen?“ anhand dieser Richtung bewerten.

9.4.2 Datensatz-Auditierung

Große Datensätze (wie LAION-5B) enthalten oft unangemessene Inhalte (Gewalt, Pornografie, Hass), die Modelle lernen.

- **Problem:** Selbst harmlose Prompts können durch Assoziationen im Datensatz zu pornografischen Ausgaben führen.
- **Lösung:** Tools wie *LlavaGuard* oder *AI Auditor* helfen, Datensätze zu durchsuchen und unsichere Inhalte zu filtern.

9.4.3 Fair Diffusion

Methoden, um generative Modelle fairer zu machen, ohne sie neu zu trainieren.

Fair Guidance

Ähnlich wie *Classifier-Free Guidance* die Qualität verbessert, kann *Fair Guidance* den Bias reduzieren.
- Es wird eine „Fairness-Richtung“ definiert (z.B. Geschlecht bei Berufen wie „Feuerwehrmann“).
- Der Generierungsprozess wird aktiv entlang dieser Richtung gesteuert, um eine ausgewogene Darstellung (z.B. 50% Frauen) zu erreichen.

9.4.4 Revision Transformers

Ein Ansatz, um die Werte eines Modells nachträglich zu korrigieren.

- Ein „Revision Engine“ prüft die Ausgabe des LLMs gegen eine Datenbank von Normen (z.B. Gesetze).
- Falls eine Regel verletzt wird (z.B. „Waffenbesitz ist in Europa illegal“), wird die Ausgabe angepasst.

9.5 Hybride KI: Neuro-Symbolische Ansätze

Reine neuronale Netze haben Schwächen im logischen Schließen (Reasoning). Neuro-symbolische KI (NeSy) kombiniert neuronale Wahrnehmung mit logischer Inferenz.

9.5.1 System 1 vs. System 2

In Anlehnung an Daniel Kahneman (Thinking, Fast and Slow):

- **System 1 (Neuronale Netze):** Schnell, intuitiv, musterbasier (Wahrnehmung).
- **System 2 (Symbolische Logik):** Langsam, deliberativ, logisch, regelbasiert.

9.5.2 Kombinationsansätze (z.B. SLASH, V-LoL)

Das Ziel ist es, aus Rohdaten (Bildern) symbolische Fakten zu extrahieren und darauf logische Regeln anzuwenden.

Deep Probabilistic Programming

1. **Wahrnehmung (Neural):** Ein neuronales Netz (z.B. ResNet) erkennt Objekte und Attribute in einem Bild und gibt Wahrscheinlichkeiten aus (z.B. $P(\text{Farbe} = \text{rot}) = 0.9$).
2. **Fakten-Konvertierung:** Diese Wahrscheinlichkeiten werden in probabilistische Fakten für eine Logik-Engine (z.B. Prolog) umgewandelt.
3. **Reasoning (Symbolic):** Die Logik-Engine wendet Regeln an (z.B. „Wenn X rot ist und Y grün, dann...“) und berechnet die Wahrscheinlichkeit der Zielaussage.
4. **Training:** Das gesamte System ist differenzierbar, d.h. der Fehler im logischen Schluss kann genutzt werden, um das neuronale Netz zu verbessern (Backpropagation durch die Logik).

9.5.3 Vorteile von Hybrider KI

- **Daten-Effizienz:** Benötigt weniger Trainingsdaten, da logisches Wissen vorgegeben werden kann.
- **Generalisierung:** Kann besser auf neue Situationen (Out-of-Distribution) verallgemeinern.
- **Verlässlichkeit:** Logische Regeln garantieren Konsistenz (z.B. „Ein Auto kann nicht gleichzeitig rot und grün sein“).

9.5.4 Deep Reinforcement Learning (RL) + Logik

Ein Agent kann lernen, wann er neuronale Intuition und wann er sicheres, logisches Wissen nutzen soll.

- Beispiel Pacman/Diver: Wenn keine Gefahr droht → Logik (effizient, energiesparend). Wenn Feind nah ist → Neural (schnelle Reaktion).
- Dies erhöht die Sicherheit (z.B. „Nicht vergessen zu atmen“).

10 Machine Learning and Neural Networks

10.1 Introduction

10.1.1 Grundlagen des Lernens

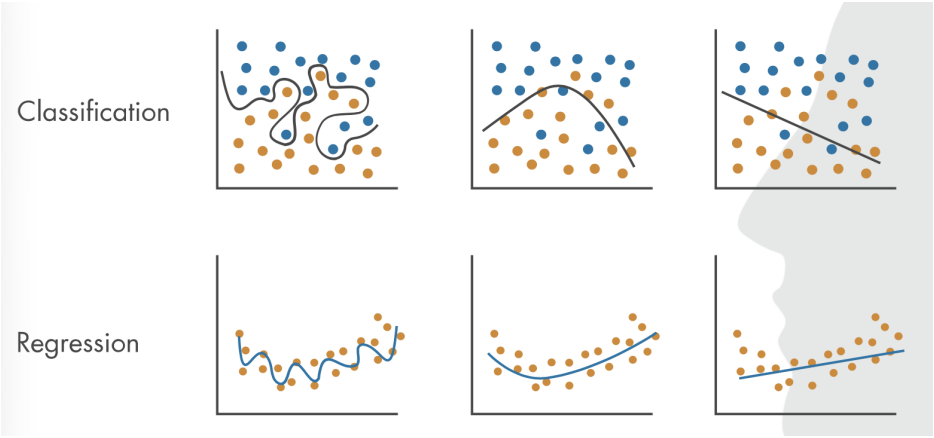
Lernen ist ein fundamentaler Prozess für intelligente Agenten, um in unbekannten Umgebungen zu agieren und die Leistung basierend auf Erfahrungen zu verbessern.

Definition: Machine Learning (Tom Mitchell, 1997)

Ein Computerprogramm lernt aus Erfahrung E im Hinblick auf eine Klasse von Aufgaben T und ein Leistungsmaß P, wenn sich seine Leistung bei Aufgaben in T, gemessen an P, mit der Erfahrung E verbessert.

Induktives Lernen Induktives Lernen ist die einfachste Form des Lernens, bei der eine Funktion aus Beispielen abgeleitet wird.

- **Ziel:** Finden einer Hypothese h, die die unbekannte Zielfunktion f approximiert ($h \approx f$).
- **Gegeben:** Ein Trainingsset von Beispielen $(x, f(x))$, wobei x der Input und $f(x)$ das Label (Target) ist.
- **Konsistenz:** Eine Hypothese h ist konsistent, wenn sie für alle Trainingsbeispiele mit f übereinstimmt.



Ockham's Razor

Die beste Erklärung ist die einfachste Erklärung, die zu den Daten passt. Im Kontext von Machine Learning bedeutet dies, dass bei gleicher Genauigkeit einfachere Modelle bevorzugt werden sollten, um Overfitting zu vermeiden.

10.1.2 Machine Learning vs. Traditionelle Programmierung

- **Traditionelle Programmierung:** Daten + Programm → Output. Der Mensch formuliert die Regeln explizit.
- **Machine Learning:** Daten + Output → Programm (Modell). Der Algorithmus lernt die Regeln ("Rule Set") aus den Daten.

10.1.3 Arten des Lernens

- **Supervised Learning (Überwachtes Lernen):** Das Modell lernt aus gelabelten Daten (Input-Output-Paare). Ziel ist es, eine Abbildung von Eingabe zu Ausgabe zu lernen.

  - **Regression:** Vorhersage eines kontinuierlichen Wertes (z.B. Temperatur).
  - **Klassifikation:** Vorhersage einer diskreten Klassenbezeichnung (z.B. "Regen" oder "Sonne").
- **Unsupervised Learning (Unüberwachtes Lernen)**: Das Modell lernt aus ungelabelten Daten. Ziel ist es, Muster, Strukturen oder Ähnlichkeiten in den Daten zu finden (z.B. Clustering).
- **Reinforcement Learning (Bestärkendes Lernen)**: Ein Agent lernt durch Interaktion mit der Umgebung und erhält positives oder negatives Feedback (Reward).
- **Semi-supervised Learning:** Eine Kombination, bei der nur ein kleiner Teil der Daten gelabelt ist.

10.1.4 Datenrepräsentation und Feature Engineering

Algorithmen benötigen Daten in einer verarbeitbaren Form, meist als **Feature-Vektoren** im $\mathbb{R}^n$.

- **Feature Engineering:** Der Prozess der Auswahl, Manipulation und Transformation von Rohdaten in Features, die für das Modell nutzbar sind.
- **Prinzip "Garbage In, Garbage Out":** Schlechte Datenqualität oder irrelevante Features führen zu schlechten Modellen, unabhängig von der Qualität des Algorithmus.
- **Preprocessing:** Wichtige Schritte umfassen den Umgang mit Ausreißern, fehlenden Werten und Bias in den Daten.

10.1.5 Modell-Evaluierung

Um die Qualität eines Modells zu messen, wird eine Metrik benötigt, die zum Ziel passt (z.B. Accuracy, Precision, Recall). Für Regressionsprobleme wird häufig der **Mean Squared Error (MSE)** verwendet.

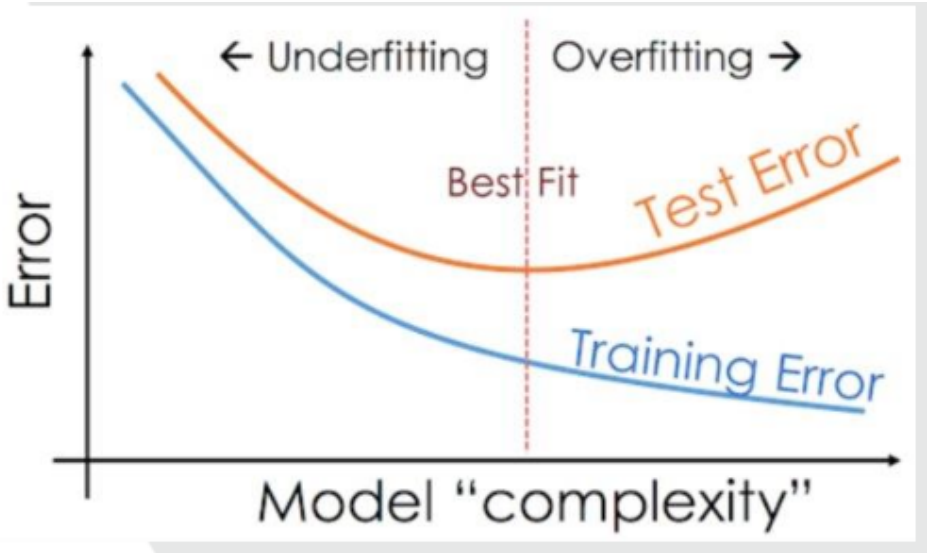
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

wobei y_i der wahre Wert und $\hat{y}_i$ der vorhergesagte Wert ist.

Generalisierung und Overfitting Das Ziel von Machine Learning ist **Generalisierung** (Leistung auf neuen, unbekannten Daten), nicht bloßes Auswendiglernen (**Memorization**).

- **Overfitting**: Das Modell hat die Trainingsdaten "auswendig gelernt" (inklusive Rauschen) und performt schlecht auf neuen Daten. Das Modell ist zu komplex.
- **Underfitting**: Das Modell ist zu einfach, um die zugrunde liegende Struktur der Daten zu erfassen.

Lösung (Train-Test-Split): Die Daten werden in ein **Training Set** (zum Lernen) und ein **Test Set** (zur Evaluation) unterteilt. Overfitting liegt vor, wenn der Trainingsfehler niedrig, der Testfehler aber hoch ist.



10.1.6 Künstliche Neuronale Netze (ANNs)

Neuronale Netze sind inspiriert von biologischen Gehirnen, versuchen diese aber nicht exakt nachzubilden. Sie sind besonders effektiv bei komplexen Inputs wie Bildern oder Sprache (Deep Learning).

Das Perzeptron (Künstliches Neuron) Das Perzeptron ist die Grundeinheit eines neuronalen Netzes. Es berechnet eine gewichtete Summe der Eingaben, addiert einen Bias und wendet eine Aktivierungsfunktion an.

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

- x_i: Inputs
- w_i: Gewichte (bestimmen Stärke und Vorzeichen der Verbindung)
- w_0: Bias (Verschiebung der Aktivierungsschwelle)
- Σ: Lineare Kombination (Summe)
- g: Nicht-lineare Aktivierungsfunktion
- $\hat{y}$: Output

Aktivierungsfunktionen Aktivierungsfunktionen entscheiden, ob ein Neuron "feuert". Sie sind essenziell, um **Nicht-Linearität** in das Netzwerk zu bringen. Ohne sie wäre jedes neuronale Netz, egal wie tief, mathematisch äquivalent zu einer einfachen linearen Regression.

Gängige Funktionen:

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$ (Wertebereich 0 bis 1, oft für Wahrscheinlichkeiten).
- **ReLU (Rectified Linear Unit):** $\max(0, x)$ (Löst das Problem verschwindender Gradienten, sehr verbreitet).
- **Tanh:** $\tanh(x)$ (Wertebereich -1 bis 1).

Multilayer Perceptron (MLP) Ein MLP besteht aus mehreren Schichten von Neuronen:

1. **Input Layer:** Nimmt die Feature-Vektoren auf.
2. **Hidden Layers:** Verarbeiten Informationen, extrahieren Features.
3. **Output Layer:** Liefert das Endergebnis.

Die Information fließt unidirektional von Input zu Output (**Forward Propagation**).

10.1.7 Training neuronaler Netze

Das Training ist ein Optimierungsprozess, um die Gewichte W so anzupassen, dass der Fehler (Loss) minimiert wird.

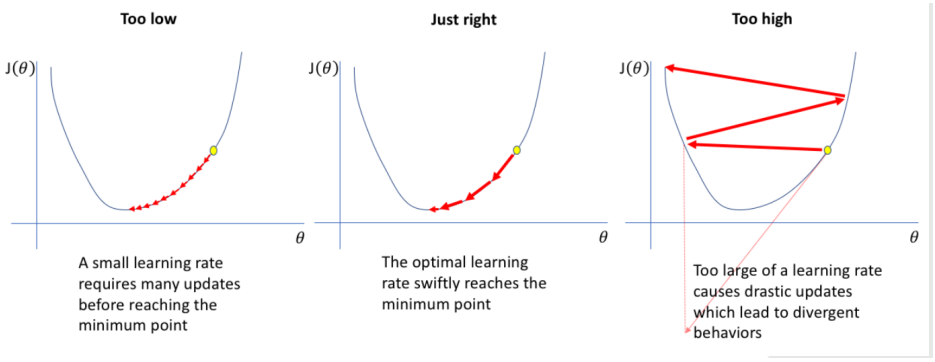
Loss Function (Verlustfunktion) Die Loss Function $J(W)$ misst die Diskrepanz zwischen Vorhersage $\hat{y}$ und wahrem Label y. Ziel ist:

$$W^* = \text{argmin}_W \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

Gradient Descent (Gradientenabstieg) Ein iterativer Algorithmus zur Minimierung der Loss Function. Man bewegt sich im "Gewichtsraum" in Richtung des steilsten Abstiegs.

$$W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$$

- $\frac{\partial J(W)}{\partial W}$: Der Gradient (Steigung) des Fehlers bezüglich der Gewichte.
- α (**Learning Rate**): Schrittweite.
  - Zu klein: Konvergenz dauert sehr lange.
  - Zu groß: Gefahr der Divergenz (man springt über das Minimum).



Backpropagation Backpropagation ist der Algorithmus zur effizienten Berechnung der Gradienten $\frac{\partial J(W)}{\partial W}$ durch Anwendung der **Kettenregel**. Der Fehler wird vom Output Layer rückwärts durch das Netz propagiert.

Beispielrechnung (Kettenregel): Gegeben sei ein einfaches Netz $x \to z \to \hat{y}$ und Loss $J(W) = (\hat{y} - y)^2$. Der Einfluss eines Gewichts w auf den Fehler ist:

$$\frac{\partial J(W)}{\partial w} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

1. Wie ändert sich der Fehler mit dem Output? $(\frac{\partial J}{\partial \hat{y}})$
2. Wie ändert sich der Output mit der Aktivierung? $(\frac{\partial \hat{y}}{\partial z} = g'(z))$
3. Wie ändert sich die Aktivierung mit dem Gewicht? $(\frac{\partial z}{\partial w} = x)$

10.1.8 Regularisierung

Methoden zur Vermeidung von Overfitting in neuronalen Netzen:

- **Early Stopping:** Training beenden, sobald der Fehler auf dem Test-Set wieder ansteigt.
- **Dropout:** Zufälliges Deaktivieren von Neuronen während des Trainings, um Robustheit zu erzwingen.
- **Data Augmentation:** Künstliche Vergrößerung des Datensatzes (z.B. durch Rauschen oder Rotation bei Bildern).

10.1.9 Convolutional Neural Networks (CNNs)

CNNs sind spezialisierte Architekturen für grid-artige Daten (z.B. Bilder).

- **Convolutional Layers:** Verwenden Filter, um lokale Features (Kanten, Formen) zu extrahieren.
- **Pooling Layers:** Reduzieren die Dimensionalität (z.B. Max Pooling) und machen das Modell robuster gegenüber Verschiebungen.
- Tiefe Schichten erkennen komplexere Objekte (Hierarchie der Features).

11 Reinforcement Learning und AlphaZero

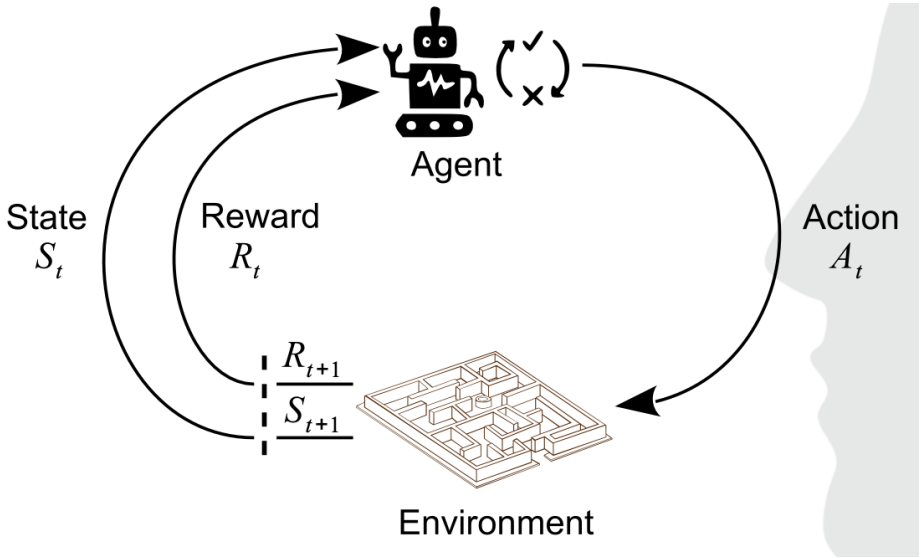
11.1 Grundlagen des Reinforcement Learning

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).



11.2 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).

11.2.1 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.

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

11.2.2 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.

11.2.3 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.

11.3 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:

11.3.1 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 $V^*(s')$.

11.4 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**.

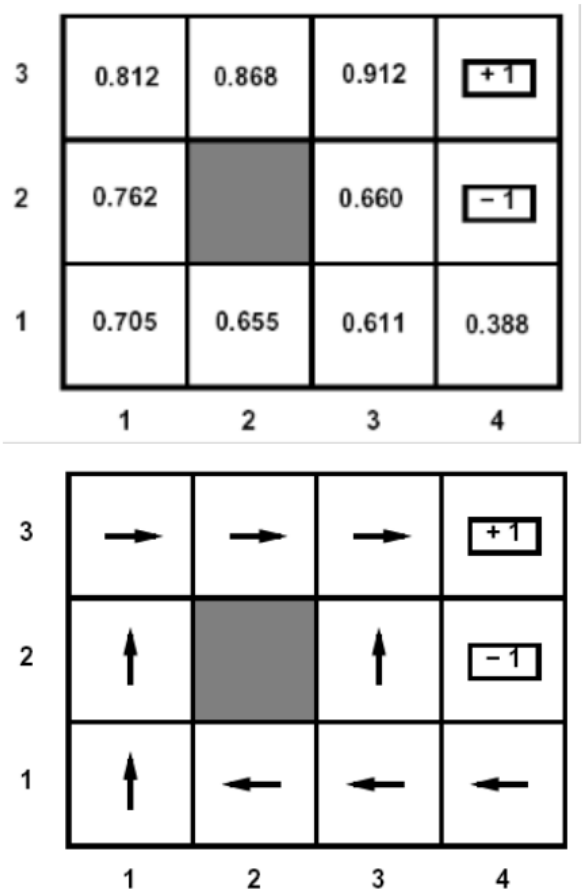
Value Iteration Algorithmus

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).



11.5 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.

11.5.1 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).

11.5.2 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 $V(s')$.

11.5.3 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.

11.5.4 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.

11.6 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.

11.7 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.

11.7.1 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).

11.7.2 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:

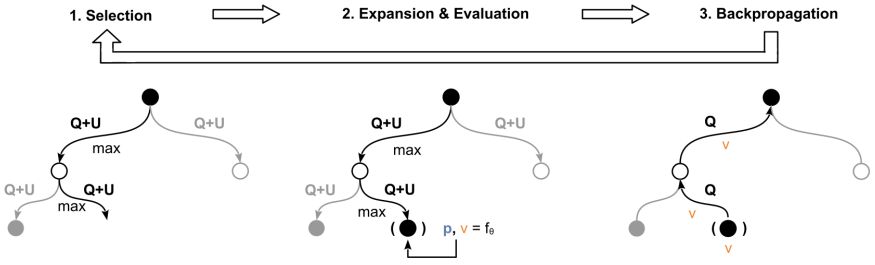
$$a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$$

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.



11.7.3 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.

12 Planning

12.1 Einführung in Classical Planning

Planung (Planning) ist der Prozess, eine Sequenz von Aktionen zu finden, die einen Ausgangszustand in einen Zielzustand überführt, wobei logisches Schließen (Reasoning) verwendet wird, anstatt durch Versuch und Irrtum zu lernen (wie beim Reinforcement Learning).

Definition: Planning

Planning ist die Aufgabe, eine **Sequenz von Aktionen** zu erstellen, die ein **Ziel** (Goal) erreicht, ausgehend von einem **Initialzustand** (Initial State).

12.1.1 Eigenschaften der Umgebung

In der klassischen Planung (Classical Planning) gehen wir von einer spezifischen Umgebung aus:

- **Fully observable (Vollständig beobachtbar):** Der Agent kennt den kompletten Zustand der Welt.
- **Deterministic (Deterministisch):** Das Ergebnis einer Aktion ist genau vorherbestimmbar (kein Zufall).
- **Finite (Endlich):** Es gibt eine endliche Anzahl an Zuständen und Aktionen.
- **Static (Statisch):** Die Welt ändert sich nur durch die Aktionen des Agenten.
- **Discrete (Diskret):** Zustände und Zeitschritte sind diskret.

12.2 Planning vs. Problem Solving

Klassische Suchalgorithmen (Problem Solving) und Planung lösen ähnliche Probleme, unterscheiden sich jedoch fundamental in der Repräsentation.

- **Problem Solving:** Betrachtet Zustände, Ziele und Aktionen als *Black Boxes*. Der Agent versteht die innere Struktur der Zustände nicht.
- **Planning:** Nutzt **explizite Repräsentationen** (meist in First-Order Logic). Zustände, Ziele und Aktionen werden durch logische Sätze beschrieben.

Vorteile der Planung:

- **Reasoning:** Der Planer kann die Effekte von Aktionen analysieren.
- **Dekomposition:** Probleme können in Teilprobleme zerlegt werden, was die Komplexität drastisch reduziert (von exponentiell auf linear oder polynomial, je nach Abhängigkeit).

12.2.1 Problem-Dekomposition

1. **Decomposable Problems:** Teilziele sind komplett unabhängig (z. B. *have(milk)* und *have(bread)*). Sie können separat gelöst werden.
2. **Nearly Decomposable Problems:** Teilziele interagieren, aber die Interaktionen sind handhabbar (z. B. Routenplanung für Pakete: Erst Verteilung auf Flughäfen, dann lokale Auslieferung).

12.3 Logische Grundlagen und Notation

Die Planung verwendet oft eine Prolog-ähnliche Notation der Prädikatenlogik (First-Order Logic - FOL).

- **Konstanten:** Objekte (beginnen mit Kleinbuchstaben oder Zahlen), z. B. `bob`, `1`.
- **Variablen:** Platzhalter für Objekte (beginnen mit Großbuchstaben), z. B. `X`, `Person`.

- **Prädikate:** Relationen zwischen Objekten, z. B. `parent(pam, bob)`.
- **Regeln:** Implikationen, z. B. `Head :- Cond1, Cond2`.

12.4 Situation Calculus

Der Situation Calculus ist ein Ansatz, um Veränderungen in der Welt mittels Logik zu modellieren.

12.4.1 Kernkonzepte

- **Situation (s):** Ein Schnappschuss der Welt zu einem bestimmten Zeitpunkt.
- **Situations-Variable:** Jedes Prädikat, das sich ändern kann, erhält ein zusätzliches Argument für die Situation.
- **Result-Funktion:** $result(a, s)$ gibt die neue Situation zurück, die entsteht, wenn Aktion a in Situation s ausgeführt wird.

Beispiel für eine logische Regel im Situation Calculus:

$$at(A, Y, result(walk(Y), S)) \leftarrow at(A, X, S)$$

Bedeutung: Agent A ist am Ort Y in der Situation, die aus dem Gehen nach Y resultiert, falls A vorher in Situation S am Ort X war.

12.4.2 Probleme des Situation Calculus

The Frame Problem

Das **Frame Problem** beschreibt die Schwierigkeit, auszudrücken, was sich durch eine Aktion *nicht* ändert. In der klassischen Logik muss explizit gesagt werden, dass alles, was nicht durch die Aktion verändert wird, gleich bleibt (Frame Axioms).

- **Representational Frame Problem:** Man müsste für jede Kombination aus Aktion und Prädikat eine Regel schreiben, die besagt, dass das Prädikat unverändert bleibt. Dies führt zu einer riesigen Wissensbasis.
- **Inferential Frame Problem:** Der Beweiser (Theorem Prover) verschwendet die meiste Zeit damit, zu beweisen, dass sich Dinge *nicht* geändert haben.

Weitere Probleme:

- **Qualification Problem:** Schwierigkeit, *alle* Bedingungen zu nennen, die erfüllt sein müssen, damit eine Aktion erfolgreich ist (z. B. "Auto springt an" → Batterie voll, Tank voll, Zündkerzen okay, keine Banane im Auspuff...).
- **Ramification Problem:** Schwierigkeit, alle impliziten Seiteneffekte einer Aktion zu beschreiben (z. B. wenn ich einen Koffer bewege, bewegen sich auch die Socken darin).

12.5 STRIPS (Stanford Research Institute Problem Solver)

STRIPS ist eine Restriktion der allgemeinen Logik, um Planung effizienter zu machen und das Frame Problem zu umgehen.

12.5.1 Repräsentation

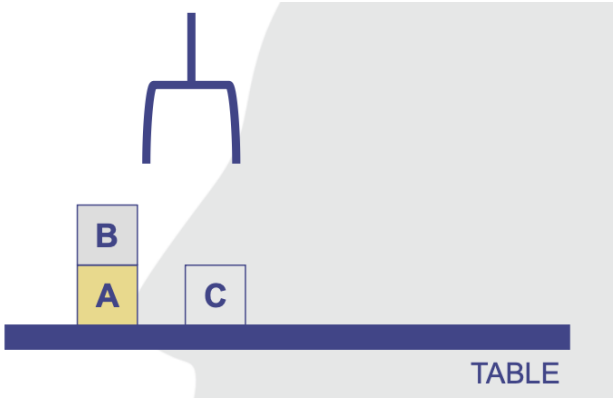
- **Zustand (State):** Konjunktion von positiven, funktionsfreien Literalen (Ground Literals).
  - **Closed World Assumption (CWA):** Alles, was nicht explizit im Zustand als wahr aufgeführt ist, ist falsch.
- **Ziel (Goal):** Konjunktion von Literalen. Ein Zustand erfüllt das Ziel, wenn er alle Ziel-Literale enthält.
- **Aktion (Action):** Definiert durch drei Komponenten:

- **Preconditions (Vorbedingungen):** Konjunktion von Literalen, die wahr sein müssen, damit die Aktion ausführbar ist.
- **ADD-List:** Literale, die nach der Aktion wahr werden (zum Zustand hinzugefügt werden).
- **DELETE-List:** Literale, die nach der Aktion falsch werden (aus dem Zustand entfernt werden).

STRIPS Assumption

Um das Frame Problem zu lösen, nimmt STRIPS an, dass jedes Literal, das **nicht** in der ADD- oder DELETE-Liste einer Aktion erwähnt wird, **unverändert** bleibt.

12.5.2 Beispiel: Blocks World



- **Prädikate:** $on(x, y)$, $on(x, table)$, $clear(x)$, $holding(x)$, $handempty$.
- **Aktionen:**
  - $stack(x, y)$: Lege x auf y.
  - $unstack(x, y)$: Nimm x von y.
  - $pickup(x)$: Nimm x vom Tisch auf.
  - $putdown(x)$: Lege x auf den Tisch.

Beispiel einer Aktion $stack(x, y)$ in STRIPS:

- **Precond:** $holding(x) \wedge clear(y)$
- **Add:** $handempty \wedge on(x, y) \wedge clear(x)$
- **Delete:** $holding(x) \wedge clear(y)$

12.6 Planungs-Algorithmen

Da STRIPS-Probleme als Zustandsraumsuche formuliert werden können, können Standard-Suchalgorithmen (wie A*) verwendet werden. Es gibt zwei Hauptrichtungen:

12.6.1 Progression (Forward Planning)

Suche vom Initialzustand zum Ziel.

1. Start: Initialzustand.
2. Finde alle anwendbaren Aktionen (deren Preconditions im aktuellen Zustand erfüllt sind).
3. Generiere Nachfolgezustände durch Anwenden von ADD- und DELETE-Listen.
4. Prüfe, ob der neue Zustand das Ziel erfüllt.

Nachteil: Hoher Verzweigungsfaktor, da viele irrelevante Aktionen möglich sind.

12.6.2 Regression (Backward Planning)

Suche vom Ziel rückwärts zum Initialzustand. Dies ist oft effizienter, da nur **relevante** Aktionen betrachtet werden.
Relevant Action: Eine Aktion ist relevant für ein Ziel G, wenn sie mindestens ein Literal aus G in ihrer ADD-Liste hat und keines der Literale in G in ihrer DELETE-Liste steht (Konsistenz).
Inverse Action Application: Wenn wir ein Ziel G haben und eine Aktion A rückwärts anwenden, berechnet sich das neue Teilziel (Predecessor Goal) G' wie folgt:

$$G' = (G \setminus \text{ADD}(A)) \cup \text{PRECOND}(A)$$

Das bedeutet:

1. Entferne die Effekte der Aktion, da diese durch die Aktion selbst erfüllt wurden (wir brauchen sie nicht mehr zu suchen).
2. Füge die Vorbedingungen der Aktion hinzu, da diese *vorher* erfüllt sein mussten, damit die Aktion überhaupt stattfinden kann.

12.7 Heuristiken für die Planung

Da die exakte Lösung NP-schwer ist, benötigen wir Heuristiken für die Suche.

12.7.1 Relaxed Problem

Man vereinfacht das Problem, um eine Abschätzung der Kosten (Anzahl Schritte) zu erhalten.

- **Ignore Preconditions:** Man nimmt an, jede Aktion sei immer ausführbar.
- **Ignore Delete Lists:** Man nimmt an, Aktionen machen nichts "kaputt".

Die Lösung dieses entspannten Problems dient als heuristischer Wert für das echte Problem.

12.7.2 Subgoal Independence Assumption

Man nimmt an, dass die Kosten, um eine Konjunktion von Zielen zu erreichen, gleich der Summe der Kosten der einzelnen Ziele sind.

$$Cost(G_1 \wedge G_2) \approx Cost(G_1) + Cost(G_2)$$

Achtung: Diese Heuristik ist nicht zulässig (not admissible), wenn positive Interaktionen existieren (eine Aktion erfüllt beide Ziele), und sie unterschätzt die Kosten massiv, wenn negative Interaktionen existieren (Aktionen für G_1 zerstören G_2).