Chapter 1 - Einführung in die KI

Einführung in die Künstliche Intelligenz
WiSe 2024/25

1 What is intelligence

1.1 Turing Test

Der Turing Test ist eine Methode mit der folgende Frage beantwortet werden kann: Wann verhält ein System sich intelligent? Es wird hierbei angenommen, dass etwas intelligent ist, wenn dessen Verhalten nicht von einer anderen intelligenten Entität unterschieden werden kann.

- Mensch interagiert blind mit zwei Spielern, wovon einer ein Computer ist.
- Wenn der Mensch nicht entscheiden kann, welcher der zwei Spieler der Computer ist, dann hat der Computer den Test bestanden.

1.2 The Chinese Room Argument

Das CRA behandelt die Frage ob Intelligenz das gleiche ist wie intelligentes Verhalten. Hierbei wir angenommen, dass wenn eine Maschine sich intelligent verhält, es nicht heißt, dass es intelligent ist.

- Eine Person kann kein Chinesisch und ist in einem Zimmer eingesperrt. Außerhalb des Zimmers ist eine weitere Person, die Notizen in Chinesisch in das Zimmer legen kann, aber nicht in einem anderen Weg mit der nicht Chinesisch sprechenden Person interagieren kann.
- Bei einem Frage-Antwort Szenario auf Chinesisch, hätte die Person innerhalb des Zimmers detaillierte Anleitungen die Fragen zu beantworten, ohne das Chinesisch zu übersetzen oder gar zu verstehen.

$\Rightarrow$ Ist die Person im Raum intelligent?

1.3 Müssen wir wissen was ein intelligentes System ist?

Laut McCarthy wird eine feste Definition nicht viel Einfluss auf die AI-Research haben.

1.4 General vs. Narrow AI

- General/Strong AI, kann beliebige intellektuelle Aufgaben erledigen
- Narrow/Weak AI, kann nur eine bestimmte Aufgabe (oder Menge von Aufgaben) erledigen.

1.5 Charakteristiken einer AI

- **Anpassbarkeit**: Die Fähigkeit, die Leistung durch Lernen aus Erfahrung zu verbessern
- **Autonom**: Die Fähigkeit, Aufgaben in Umgebungen ohne ständige Anleitung durch einen Benutzer/Experten auszuführen.
- **Rationalität**:
  - Law of Thoughts: Beginn der Argumentation und die Frage, was ,,richtige" Argumente und Denkprozesse sind.
  - Rationales Verhalten: Die Frage,,,das Richtige zu tun". In Systemen, die rational handeln, ist das ,,Richtige" das, von dem man erwartet, dass es die Leistung angesichts der verfügbaren Informationen maximiert.

1.6 Was ist eine AI?

Es gibt zwei Dimensionen aufgrund zwei Kriterien.

- Denkprozesse vs. Verhalten
- Erfolg gemäß Menschlichen Standards vs. Erfolg gemäß idealer Konzept der Intelligenz

Rationales Verhalten hat zwei Vorteile gegenüber dem Law of Thoughts

- Es ist allgemeiner (in vielen Situationen, gibt es beweisbare keine richtige Aktion)
- Es ist zugänglicher (Rationalität kann definiert und optimiert werden)

$\Rightarrow$ Rationalität ist selten ein sehr gutes Modell der Realität.

1.7 Fundamente der AI

- Philosophie: Logik, Reasoning, Gedächtnis als physisches System
- Mathematik: Formelle Repräsentation und Beweis Algorithmen, Berechnung, Wahrscheinlichkeit
- Psychologie: Anpassbarkeit, Wahrnehmungsphänomenen und Motorikkontrolle
- Wirtschaft: Spieltheorie, Rationale Entscheidungstheorie
- Linguistik: Wissensrepräsentation
- Neurowissenschaft: körperliches Substrat für geistige Aktivitäten
- Kontrolltheorie: Homöostatische Systeme, Stabilität, optimaler Entwurf von Agenten

1.8 Taxonomie von AI

AI > ML > DL

1.8.1 Sub-Disziplinen AI

ML, DL, Suche und Optimierung, Robotik, NLP, NLP, Computer Vision, ...

1.9 Was AI kann und nicht

- Aktuell ist AI oft in einzelne Probleme isoliert
- AI ist voreingenommen (Bsp.: Gesichtserkennung die nur auf weißen Menschen trainiert wurde, wird bei schwarzen Menschen Probleme haben)

Chapter 2 - KI-Systeme

Einführung in die Künstliche Intelligenz
WiSe 2024/25

1 Was ist ein KI-System?

⇒ die Untersuchung rationaler Agenten und ihrer Umgebung. Sie bestehen aus Umgebung und Agenten.
Umwelt : Umgebung oder die Bedingungen, unter denen ein Mensch, ein Tier oder eine Pflanze lebt oder arbeitet. Im Kontext KI: Umgebung eines KI-Agenten in dem es ist und operiert. Umwelt kann artificial sein.

1.1 Charakteristiken von Umgebungen

- Diskret vs. Kontinuierlich: Hat die Umgebung eine abzählbare Zahl von definierten States? Ja: Diskret; Nein: Kontinuierlich
- Beobachtbar vs. Teilweise beobachtbar oder nicht beobachtbar: Ist es möglich, den vollständigen Zustand der Umwelt zu jedem Zeitpunkt zu bestimmen, so wird sie als beobachtbar bezeichnet; andernfalls ist sie nur teilweise beobachtbar oder sogar unbeobachtbar.
- Statisch vs. Dynamisch: Wenn sich die Umgebung nicht ändert, während ein Agent handelt, dann ist sie statisch.
- Single Agent vs. Multiple Agent: Die Umgebung kann andere AGenten enthalten, die von der gleichen oder einer anderen Art sein können wie die des Agents.
- Zugänglichkeit vs. Unzugänglichkeit: Wenn ein Agent vollständige und genaue Informationen über die Umgebung des Zustands erhalten kann, dann wird eine solche Umgebung als zugängliche Umgebung bezeichnet, andernfalls als unzugänglich.
- Deterministisch vs. Nicht-deterministisch: Wenn der nächste Zustand der Umwelt vollständig durch den aktuellen Zustand und die Handlungen des Agenten bestimmt ist, dann ist die Umwelt deterministisch.
- Episodisch vs. Nicht-episodisch/sequenziell: In einer episodischen Umgebung besteht jede Episode darin, dass der Akteur wahrnimmt und dann handelt. Die Qualität seines Handelns hängt nur von der Episode selbst ab. In sequentiellen Umgebungen benötigt der Agent ein Gedächtnis für vergangene Handlungen.

⇒ Die Klassifizierung ist hilfreich, um nach bestimmten Algorithmen zu filtern. Nicht jeder Algorithmus funktioniert in jeder Umgebung.

2 Agenten

⇒ ein Agent kann fühlen, denken und handeln.

Regeln eines KI-Agenten

- Ein Agent muss in der Lage sein, die Umwelt wahrzunehmen
- Die Beobachtungen der Umwelt müssen genutzt werden, um Entscheidungen zu treffen
- Die Entscheidungen sollten zu einer Handlung führen
- (The action taken by the agent must be rational)

2.1 Problem von Rationalen Agenten

⇒ Eine Handlung ist rational, wenn sie die Leistung maximiert und das beste positive Ergebnis für den Agenten bringt.
→ Ein rationaler Agent, ist ein Agent der das Richtige macht. **Aber was ist das Richtige?**. Rationale Agenten sind nicht allwissend, und wissen nicht was die Ergebnisse der Aktion direkt sind.

Wie wird Leistung gemessen: Eine Funktion welche die Sequenz der Aktionen evaluiert. Diese Leistungskriterien sind abhängig von der Aufgabe.
→ Entwerfen Sie die Leistungsmessung auf der Grundlage des gewünschten Ergebnisses, nicht des gewünschten Verhaltens der Agenten.

2.2 Typen von Agenten

- Reflex Agent: Aktion wird nur auf der Grundlage der aktuellen Wahrnehmung gewählt, ohne Geschichte.
  - Implementiert durch State-To-Action Mappings.
  - Sehr limitiert, und haben kein Wissen über Dinge die nicht direkt Wahrgenommen werden.
- Model-basierter Agent: Handlungen wie Reflexagenten, aber mit besserem Überblick über Umwelt. Zustand der Welt wird ebenfalls beobachtet.
  - Inputs werden neben der Interpretierung auch in interne Zustandsbeschreibung umgesetzt.
  - Wie wirken sich die Aktionen auf die Welt aus?
  - Welches Weltmodell wird gewünscht?
- Ziel-basierter Agent: Diese Agenten bauen auf den Informationen auf, die ein modellbasierter Agent speichert, wissen aber darüber hinaus, welche Zustände wünschenswert sind

  - Agenten mit Heuristik
  - Hauptunterschied zu vorherigen: Entscheidungsfindung wird mit bezogen
  - Schwierig wird es, wenn lange Abfolgen von Handlungen erforderlich sind, um ein Ziel zu finden/zu erreichen (tiefer Suchbaum)

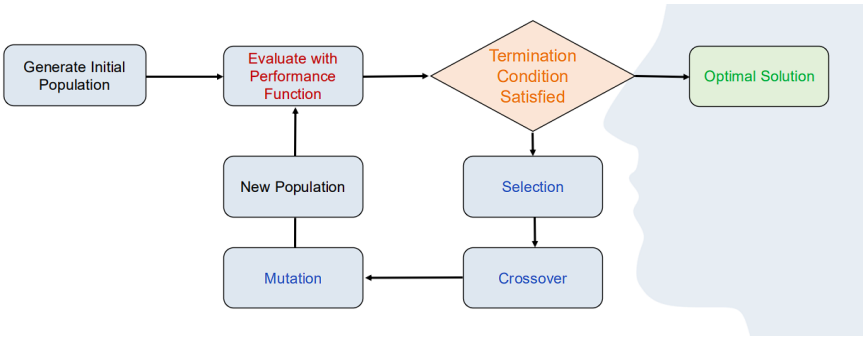
- Utility-basierter Agent: Anstelle von Zielen verwenden nutzungsbasierte Agenten eine Nutzenfunktion, mit der jede Aktion/jedes Szenario auf der Grundlage des gewünschten Ergebnisses bewertet werden kann
  - Ziele bieten eine binäre Unterscheidung, während eine Nutzenfunktion eine kontinuierliche Skala bietet
  - Kann bei der Auswahl zwischen widersprüchlichen Zielen helfen
  - Utility Funktion: Abbildung eines Zustands oder einer Folge von Zuständen auf eine reelle Zahl

- Lernender Agent: Diese Agenten verwenden ein zusätzliches Lernelement, um sich schrittweise zu verbessern und mit der Zeit mehr über eine Umgebung zu erfahren.
  - Lernt von ehemaligen Aktionen
  - Besteht aus vier Komponenten:
    * Lernendes Element: Zuständig Verbesserungen beim Lernen zu machen
    * Kritik: Gibt Feedback, und bewertet Agent Aktion
    * Leistungselement: Aktionsauswahl
    * Problem Generator: Zuständig Aktionen vorzuschlagen die zu neuen Erfahrungen führen

2.3 Wie macht man Agenten intelligent?

⇒ Intelligente Agenten machen intelligente Aktionen, aber wie entscheiden wir, was eine intelligente Aktion ist und wie wählen wir diese aus?

- Such Algorithmen: Problem wird auf ein Baum abgebildet von Aktionen, und die besten Ergebnisse werden gesucht. Beispiele: DFS, BFS, Brepth-FS, Deadth-FS
- Reinforcement Learning: Trial and Error. Ein RL-Loop herrscht zwischen Agent und Umgebung. Agent führt Aktion durch und Erhält Belohnung und neuen Zustand.
- Genetische Algorithmen: Survival of the fittest


Generate Initial Population
Evaluate with Performance Function
Termination Condition Satisfied
Optimal Solution
New Population
Selection
Mutation
Crossover

Chapter 3 - (Un-)informierte Suche

Einführung in die Künstliche Intelligenz
WiSe 2024/25

1 Wie löst man ein Problem?

- Problem-lösende Agenten sind Ergebnis-gesteuert und fokussieren sich darauf seine Ziele zu erfüllen
- Probleme müssen für Agenten anders formuliert werden. Zunächst muss ein Ziel definiert werden, und dann ein Problem bezogen auf das Ziel.

1.1 Der Zustandsraum / Zustände

Ein Zustand ist eine mögliche Situation in unserer Umgebung. Der Zustandsraum ist die Menge aller möglichen Zustände (die von einem Startzustand erreicht werden können). Oft definiert durch Startzustand und Nachfolgerfunktion (mapped State, auf Ihre Nachfolger -¿ kreiert State-Space-Graphen).

1.2 Übergang / Aktion

Übergänge sind mögliche Aktionen die durchgeführt werden, um in einen neuen Zustand zu gelangen. (In der VL werden nur direkte Übergänge (single actions) betrachtet.

1.3 Kosten

Kosten stellen den Aufwand dar um eine gewisse Aktion durchzuführen. Oft ist das Ziel die Kosten zu minimieren während man das Ziel erreicht.

1.4 Komponenten um ein Problem zu formulieren

- Zustandsraum und Startzustand
- Aktionsbeschreibung: Eine Funktion die ein Zustand zu einer Menge möglicher Aktionen auf dem Zustand mapped.
- Zieltest: Eine Funktion die untersucht, ob der aktuelle Zustand das Ziel erfüllt
- Kosten: Eine Funktion die eine Aktion zu deren Kosten mapped.

1.5 Pfad

eine Sequenz von Zuständen die durch eine Sequenz von Aktionen verbunden werden

1.6 Lösung

ein Pfad der den Agenten vom Startzustand, zum Zielzustand bringt

1.7 Optimale Lösung

Eine Lösung mit minimalen Pfadkosten

1.8 Planning Problem

ein Problem indem ein Startzustand gegeben ist und man diesen in ein gewünschtes Ziel umwandeln möchte, wobei zukünftige Aktionen und deren Ergebnisse berücksichtigt werden.

1.9 Suche

Der Prozess des Findens einer (optimalen) Lösung für ein solches Problem in der Form von Sequenzen und Aktionen.

1.9.1 Tree-Search Algorithms

⇒ Zustandsgraph wird als Baum verwendet, um simulierte iterative Erkundung des Zustandsraums zu ermöglichen. Es muss jedoch eine Strategie gegeben sein, um zu bestimmen welcher Knoten erweitert wird.

```
def TreeSearch(problem, strategy):
    initialize the search tree using the init.state of problem
    loop do
        if there are no candidates to expand then return false

        choose leaf node for expansion according to strategy
        if node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to search tree
    end
```

1.9.2 Fringe

Menge aller Knoten die am Ende aller Pfade besucht wurden

1.9.3 Depth

Zahl der Level im Suchbaum

1.10 Unterschied Zustand und Knoten

- ein Zustand ist eine Darstellung einer physikalischen Konfiguration, die eine bestimmte Situation in der Umgebung beschreibt.
- ein Knoten ist eine Datenstruktur der Teil eines Suchbaums ist, und einen Zustand enthält, sowie einen übergeordneten Knoten, die durchgeführte Aktion, die Pfadkosten und die aktuelle Tiefe des Baums.
- ein Knoten zu erweitern ist das Erstellen neuer Knoten und aktualisierung des Baums.

1.11 Such Strategien

Suchstrategie: wird definiert, indem die Reihenfolge der Knotenexpansion gewählt wird
Folgende Eigenschaften unterschieden sich bei verschiedenen Suchstrategien:

- Completeness: findet immer eine Lösung wenn es eine gibt
- Zeitkomplexität: Anzahl der Knotenerweiterungen
- Raumkomplexität: Maximale Anzahl der Knoten im Speicher
- Optimalität: Es wird immer die optimale Lösung gefunden

⇒ die Zeit- und Raumkomplexität wird gemessen durch den maximalen Verzweigungsfaktor des Suchbaums (b), der Tiefe der optimalen Lösung (d) und der maximalen Tiefe eines Zustands im Zustandsraum (m)(kann unendlich sein)

1.11.1 Hauptgruppen der Suchstrategien

- Uninformierte Suche: Die einzige bereitgestellte Information ist die Problembeschreibung
  - BFS
  - Uniform-Cost Search
  - DFS

  - Depth-Limited-Search
  - Iterative Deepening

- Informierte Suche: Zusätzliches Wissen ist vorhanden mit einer Idee wie man nach der Lösung sucht.
  - Greedy Best-first Search
  - A* Search
  - Memory-Bounded Heuristic Search

1.11.2 Uniform-Cost Search und BFS

- UCS: jeder Knoten hat einen festen Kostenwert (nicht unbedingt alle gleich), die sich im Laufe der Suche über den Pfad akkumulieren. Die niedrigsten kumulativen Kosten werden verwendet, um einen Pfad zu finden.
- BFS: Spezialfall von UCS, wenn alle Knoten gleiche Kosten haben. Baum wird von der Wurzel aus Ebene für Ebene surchsucht. BFS hört auf, sobald es ein Ziel erzeugt, während UCS alle KNoten in der Tiefe des Ziels untersucht.

⇒ wenn jeder Schritt positive Kosten hat, ist BFS complete. (Sonst evtl. infinite-loops)

$\Rightarrow BFS : O(b^d); UCS : O(b^{1+\text{floor(OptCost/eps)}}$ mit OptCost= Kosten der optimalen Lösung

⇒ BFS ist optimal aber zu Speicheraufwendig

1.11.3 DFS

⇒ erkundet den Baum so weit wie möglich entlang eines Zweiges, bevor er schrittweise zurückgeht und alternative Zweige erkundet.

- Completeness: Nein, denn wenn die Tiefe unendlich ist, läuft der Algorithmus einfach weiter.
- Time Complexity: Jeder Zweig wird bis zum Maximum untersucht $O(b^m)$
- Space Complexity: Lineare Komplexität, weil nur ein Pfad und die nicht erweiterten Siblings gespeichert werden $O(b * m)$
- Optimal: Nein

Criteria	BFS	DFS
Concept	Traversing tree level by level	Traversing tree sub-tree by sub-tree
Data Structure (Queue)	First In First Out (FIFO)	Last In First Out (LIFO)
Time Complexity	O(Vertices + Edges)	O(Vertices + Edges)
Backtracking	No	Yes
Memory	Requires more memory	Less nodes are stored normally (less memory)
Optimality	Yes	Not without modification
Speed	In most cases slower compared to DFS	In most cases faster compared to BFS
When to use	If the target is relatively close to the root node	If the goal state is relatively deep in the tree

1.11.4 Depth-limited Search und Iterative Deepening

⇒ Tiefe wird beschränkt. Knoten die tiefer sind, werden nicht beachtet.

- Completeness: Nein
- Time Complexity: $O(b^l)$
- Space Complexity: $O(b \times l)$
- Optimal: Nein, siehe DFS

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

⇒ Die Tiefe kann auf verschiedene Arten ausgewählt werden. Beim Iterative Deepening, wird die Tiefe erhöht, sobald alle absehbaren Knoten besucht wurden.

- Completeness: Ja
- Time Completeness: $\sum_{i=1}^{d}(d-i+1) * b\hat{i}$
- Space Complexity: Linear $O(b * d)$
- Optimal: Ja

1.11.5 Bidirectional Search

$\Rightarrow$ zwei Suchen werden gleichzeitig durchgeführt. Eine Suche von der Wurzel und die andere vom Zielzustand. Wenn ein Knoten in beiden Suchen vorkommen, hält der Algorithmus. $\rightarrow$ Komplexität wird verringert, denn: $(b^{d/2} + b^{d/2} << b^d) \Rightarrow$ Jedoch ist dieser Algorithmus nur verwendbar, wenn Aktionen reversed werden können (Bsp.: Rubiks Cube)

1.12 Informierte Such Strategien

1.12.1 Heuristik

„Fausregel" die bei der Lösung eines Problems hilfreich sein kann. Bei der Baumsuche ist es eine Funktion h, die die verbleibenden Kosten zum Erreichen des Ziels schätzt. Heuristiken können jedoch auch Falsch liegen.

1.12.2 Greedy Best-first Search

Die Kosten von einem Knoten n zum Ziel werden geschätzt und die günstigsten Knoten werden expandiert.

- Completeness: Nein. Man kann in Schleifen feststecken. (aißer in endlichen Zustandsräumen und sicherstellen, dass Knoten nicht doppelt besucht werden)
- Time Complexity: Worst case $O(b^m)$
- Space Complexity: Alle Knoten müssen gespeichert sein. Worst case $O(b^m)$
- Optimality: Nein. Hängt von der Heuristik ab.

1.12.3 A*

$\Rightarrow$ Basiert auf Best-First-Search, aber versucht nicht nur die geschätzten Kosten $h(n)$ sondern auch tatsächlichen Kosten $g(n)$ zu minieren.
Ziel ist minimieren von $f(n) = g(n) + h(n)$, wobei $g(n)$ die aktuellen Kosten sind die geleistet wurden, $h(n)$ die geschätzten Kosten um von n zum Ziel zu kommen sind und $f(n)$ die Geschätzten Kosten vom Pfad zu Ziel sind.

- Completeness: Ja (außer es gibt unendlich Knoten)
- Time Complexity: $|h(n) - h*(n)| \leq O(\log h*(n))$
- Space Complexity: Alle Knoten müssen gespeichert sein
- Optimal: Kommt auf Heuristik an

1.12.4 Zulässige Heuristiken

⇒ Zulässig wenn sie die Kosten zur Erreichung eines Ziels niemals überschätzt.

1.12.5 Konsistente Heuristik

⇒ eine Heuristik ist konsistent, wenn für jeden Knoten n und jeden Nachfolger n' der durch eine beliebige Aktion a generiert wurde gilt: $h(n) \leq c(n, a, n') + h(n')$. → eine Heuristik ist konsistent, wenn eine Heuristik die Schrittkosten niemals überschätzt.

- Lemma 1: Jede konsistente Heuristik ist zulässig.
- Lemma 2: Wenn $h(n)$ konsistent ist, dann sind die Werte von $f(n)$ entlang des Pfades nicht-abnehmend

1.12.6 Entspanntes Problem

Ein Problem mit weniger Einschränkungen für die Aktionen.
→ Die Kosten einer optimalen Lösung für ein entspanntes Problem sind eine zulässige Heuristik für das ursprüngliche Problem

1.12.7 Dominanz

Wenn h_1 und h_2 beide zulässig sind, und gilt: $h_2(n) \geq h_1(n)$, dann wird h_1 von h_2 dominiert, also h_2 hat eine bessere Performance. → Dominante Heuristiken sorgen für weniger Expansion im Baum

1.12.8 Kombinieren von Heuristiken

Wenn h_1, h_2 zulässige Heuristiken sind, dann ist $h(n) = \max\{h_1(n), h_2(n), ..., h_m(n)\}$ ebenfalls zulässig und dominiert h_1 und h_2.

1.12.9 3 Alternativen zu A*

- Iterative-deepening A* (IDA*)
  - Abschlussinfo ist nicht die Tiefe sondern die f-cost$(g + h)$
- Rekursive Best-First Search (RBFS)
  - Probiert Best-first-search mit linearem Space nachzuahmen
  - verfolgt f-Value der bisher besten Alternative

  - Pfad, der von jedem Vorgänger des aktuellen Knotens aus erreichbar ist, undd ie heristischen Auswertungen werden mit Ergebnissen der Vorgänger aktualisiert

- (Simple) Speicher-bounded A* ((S)MA*)
  - schlechtester Blattknoten wird bei vollem Speicher discarded
  - Dessen Wert wird zum Parent aktualisiert
  - muss später nochmal gesucht werden später

1.12.10 Graphensuche

⇒ wenn man es nicht schafft, zu tracken welche Zustände mehrfach besucht werden, könnte das lineare Problem zu einem exponentiellen Problem werden

→ speichere die bereits besuchten Zustände (Bsp.: Dijkstra)

1.12.11 Baumsuche vs. Graphsuche

Bei der Baumsuche repräsentiert jeder Knoten einen möglichen Pfad zum dazugehörigen Domainzustand. Das Problem hierbei ist es, dass in manchen Fällen ein besserer Pfad später entdeckt wird. Die zuvor gefundenen Pfade müssen nochmals betrachtet werden mit dem neuen, günstigeren Pfad. Hierbei bieten sich 2 Lösungen

- Fähigkeit wiederholte Zustände zu detecten (Graph Search)
- Sicherstellen, dass der günstigere Pfad immer genommen wird (Konstistente Heuristik)

Chapter 3 - Lokale und gegnerische Suche

Einführung in die Künstliche Intelligenz
WiSe 2024/25

(Un-)informierte Suchen erkunden den Suchraum systematisch mit evtl. principled Pruning. Suchalgorithmen können Suchräume von bis zu 10^{100} Zuständen behandeln. Aber wenn die Suchräume viel größer sind oder der Pfad nicht das Ziel ist, sind diese Suchen ungeeignet.

1 Lokale Suchalgorhtmen

Optimierungsproblem: Alle Zustände können eine Lösung sein aber das Ziel ist es den Zustand zu finden, der eine Lösung minimiert/maximiert, je nach Zielfunktion. Es gibt kein Zielzustand und kein Pfad.

Konvergenz: eine Eigenschaft einer Sequenz die sich einer Grenze nähert aber dieses nie erreicht
Globales Optimum: Extremum einer Zielfunktion für den gesamten Inputsuchraum
Lokales Optimum: Extremum einer Zielfunktion für eine gegebene Region des Suchraums
Lokale Suche: Traversen nur einzelne Zustände und speichern den Pfad nicht. Zustand wird iterativ modifiziert um ein Kriterium zu verbessern. Vorteil hierbei ist dass es wenig Speicher benötigt und eine Lösung in großen Suchräumen gefunden werden kann. Aber ob eine (optimale) Lösung gefunden wird ist nicht sicher.
→ Verbessern durch Maximierung einer heuristischen Evaluierung in der nur lokale Verbesserungen verwendet werden

Algorithmusdesign Überlegungen

- Wie soll das Problem repräsentiert werden?
- Was ist die Zielfunktion?
- Was ist ein Nachbar eines Zustands?
- Gibt es ausnutzbare Beschränkungen?

1.0.1 Hill Climbing / Greed Local Search

- Aktueller Zustand erweitern (Nachbarn generieren)
- Zum Nachbarzustand mit höchster Evaluierung gehen
- Solange machen bis Maximum erreicht ist.
  ⇒ bei lokalen Optima hört der Algorithmus auf

Ideen zur Problemlösung

1. Randomized Restart Hill Climbing: Unterschiedliche Startpunkte, resultieren in unterschiedlichen Optima
2. Stochastische Hill-Climbing: Successor Knoten wird zufällig ausgewählt, bessere Knoten bekommen Höhere Wahrscheinlichkeit ausgewählt zu werden

1.0.2 Ridge Problem

⇒ Jeder Nachbarzustand scheint downhill zu sein

1.0.3 Multi-Dimensionale State-Landscape

State-Landscape: Zustände sind als Orte repräsentiert mit Heuristischen Werten um evaluieren zu können **Problem**: Globales Optimum zu finden wird schlimmer mit mehreren Dimensionen

1.0.4 Gradient Descent

Gradient: Ableitung einer Funktion, die mehr als eine Eingangsvariable hat. Gradient misst Veränderung aller Gewichte in Bezug auf die Änderung des Fehlers.
Gradient Descent: Beliebte Optimierungsstrategie; Hill-Climb in einem kontinuierlichen Zustandsraum

- Ziel: Kostenfunktion minieren $J(\Theta)$; Ableitung: $\frac{\delta J(\Theta)}{\delta \Theta}$
- Gradient Vector: $\triangledown J(\underline{\Theta}) = \left[\frac{\delta J(\underline{\Theta})}{\delta \Theta_0} \ \frac{\delta J(\underline{\Theta})}{\delta \Theta_1} \ ...\right]$

Ablauf des Algorithmus:

1. Berechne Gradient: $\frac{\delta}{\delta x_i} J(x_1, ..., x_n)$
2. Schritt in Richtung Gradient: $x_i' = x_i - \lambda \frac{\delta}{\delta x_i} J(x_1, ..., x_n)$
3. Check if $J(x_1', ..., x_n') < J(x_1, ..., x_n)$
4. Falls Wahr dann Move annehmen
5. Sonst Reject und repeat

Learning Rate: Hyperparameter, der kontrolliert wir schnell sich das Modell zum Problem anpasst
⇒ kleinere Lernraten haben kleinere Veränderungen benötigen mehrere Trainingsepochen und größere Lernraten haben mehrere zügige Änderungen, und benötigen weniger Epochen. Große Lernraten können an ein lokales Optimum konvergieren (oder konvergieren gar nicht)

Wie bestimmte man den Gradient?: Formel mit multivariater Berechnung ableiten, Mathematiker fragen, automatische Differenziation.
⇒ Variationen von Gradient Descent können Performance verbessern. Funktioniert gut in smoothen Räumen, nicht in rough.

1.0.5 SAT

Cook's Theorem: SAT ist NP-complete

1.0.6 Beam Suche

Idee: k Zustände speichern, statt nur einem. Ablauf: Generiere zufällig Zustände, nach jeder Iteration werden alle Nachfolger generiert. Wähle k beste Nachfolger von der Liste und wiederhole

1.0.7 Simulated Annealing

⇒ Hill Climb aber ab und zu wird ein Schritt getätigt, der nicht verbessernd ist. Die Wahrscheinlichkeit dass ein Down-Hill Schritt getätigt wird sinnkt je mehr Zeit vergeht.
Temperatur: Hyperparameter, der kontrolliert wird oft ein "schlechter Schritt" erlaubt wird um ein lokales Optimum zu verlassen. Die Temperatur sinkt üblicherweise exponentiell im Prozess.
⇒ Wenn die Temperatur langsam genugt sinkt, konvergiert es zu einem globalen Optimum. (Kann jedoch extremst lang dauern)
Simmulated annealing ist eine allg. Sampling Strategie um von einem Raum zu samplen in Bezug auf eine definierte Wahrscheinlichkeitsverteilung.

2 Adversarial Search

⇒ Suche die betrachtet was passiert, wenn man vorausplanen möchte und die Umgebung gegen einen plant oder widersprüchliche Ziele im selben Suchraum hat.
Zero-Sum Games: Wenn eine Partei verliert, gewinnt die andere und die Nettoänderung in Reichtum ist 0.

2.1 Problemformulierung

Ein Spiel kann als Suchproblem beschrieben werden

1. Startzustand: Wie beginnt das Spiel
2. Spieler(s): Gibt an welcher Spieler dran ist

3. Aktion(s): Return eine zulässige Aktionsmenge in den Zustand s
4. Ergebnis(s, a): Übergangsmodell, gibt an in welchem Zustand man ist, wenn man Aktion a in Zustand s ausführt
5. Terminal(s): Testet ob der Zustand s die Zielbedingungen erfüllt
6. Utility(s, p): errechnet einen numerischen Wert für ein Terminalzustand s von der Perspektive des Spielers p

2.1.1 Spielbaum

Spielbäume sind in Levels aufgebaut die dem Spieler korrespondieren, Blattknoten sind Terminals. Jeder Terminal hat einen Utility-Value, welcher das Endergebnis angibt.

2.2 Wieso lernt AI wie man Spiele spielt?

1. Spiele sind nützliche Metrik: erlaubt Verbesserung nach der Zeit zu tracken; Quantifizierbar
2. Spiele ermöglichen Sichere Umgebung um zu Trainieren
3. Spiele wecken Interesse und Phantasie: Ein Weg um Verbesserungen zu zeigen

2.3 Spiele vs. Suchproblem

- Unvorhersehbarer Gegner:
- Spiele sind oft zeitlich begrenzt und brauchen Abschätzung
- Meisten Spiele sind deterministisch, turn-based, 2-Spieler, Zero-Sum
- Meisten "realen" Probleme sind Stochastisch, parallel, Multi-agent, Nutzenbasiert

2.4 MinMax Algorithmus

⇒ Ein Spielbaum wird gebaut, wobei jeder Knoten ein Spielzustand repräsentiert und die Kanten Aktionen. Der Gegner versucht das Spielergebnis zu minimieren, um die Wahrscheinlichkeit zu senken, dass der Max Spieler gewinnt. Der Spieler versucht den Wert zu erhöhen und so eine höhere Gewinnwahrscheinlichkeit zu haben.

- MinMax folgt DFS Konzept, wobei jedes Level entweder MIN (Gegner) oder MAX ist.
- Algorithmus ist Complete, wenn der Baum endlich ist
- Zeitkomplexität: $O(b^m)$
- Raumkomplexität: $O(b * m)$
- Optimal: Ja (mit Annahme dass der Gegner optimal ist)

2.4.1 Problem zu großer Spielbäume

In vielen Spielen ist der Spielbaum zu groß, um ihn vollständig zu durchsuchen. ⇒ mit heuristischen Evaluierungsfunktionen wird die Suchbaumtiefe limitiert. Für jeden Blattknoten wird der Heuristische Werte berechnet, welcher abschätzt wie das Spiel endet, wenn dieser Knoten weitergeführt wird.

Pruning: Baum schneiden, wobei unerwünschte Teile des Baums ignoriert werden, weil diese keinen Unterschied machen oder nicht Zielbringend sind.

2.4.2 Alpha-Beta-Pruning

⇒ Modifizierte optimisierte Version vom Minimax Algorithmus, indem Pruning verwendet wird um die Entdeckung zu reduzieren ohne die Korrektheit von Minimax zu verlieren.
AB-Pruning hängt von zwei Parametern ab:

- Alpha: Die beste (high-value) Wahl die bisher gefunden wurde im Pfad einer Maximierer Wurzel. Startwert des Pfades ist $-\infty$
- Beta: Die beste (low-value) Wahl die bisher gefunden wurde im Pfad einer Minimierer Wurzel. Startwert des Pfades ist $+\infty$

Idee: Entferne alle Knoten die das Endergebnis nicht wirklich beeinflussen aber den Algorithmus langsam machen.
Unterschied zu MiniMax: Max Spieler aktualisiert nur Alphawert und Min Spieler nur Betawert.
Beim Backtracking werden die Knotenwerte an den Parent weitergegeben. Alpha und Beta wird nur an Kinderknoten gegeben.
Optimal kann Komplexität von $O(b^{m/2})$ zu $O(b^m)$ reduziert werden.

Problem von Alpha-Beta-Suche: Benötigt schnelle Evaluierungsfunktion und in Spielen mit großem Branchingfaktor (bsp.: Go), ist der Algorithmus einfach nicht gut genug.

Chapter 4 - Constraint Satisfaction Problem

Einführung in die Künstliche Intelligenz
WiSe 2024/25

1 Constraint Satisfaction Problem

Constraint Satisfaction ist eine Technik indem ein Problem gelöst ist, wenn dessen Lösung bestimmten Constraints/Regeln erfüllt.

- Zustand: definiert durch Variablen X_i mit d Werten der Domain D_i
- Zieltest: eine Menge an Constraints C, die Kombinationen von Werten für Variablensubsets definieren

Um ein CSP zu lösen benötigt man ein Zustandsraum eine ungefähre Vorstellung der Lösung.
Echte Beispiele CSP

- Scheduling
- Sudoku

1.1 Zuweisung von Werten zu Variablen in CSPs

Ein Zustand wird durch die Zuweisung von Werten zu Variablen definiert. Es gibt drei Methoden um Werte zuzuweisen.

- Konsistente/Legale Zuweisung: Zuweisung verstößt keine Constraint
- Komplette Zuweisung: Zuweisung wobei jede Variable einen Wert hat und die Lösung des CSPs konsistent ist. (zB: richtig ausgefülltes Sudoku)
- Partielle Zuweisung: nur einige Variablen haben Werte.

1.2 Constaint Graphen

Ein Graph in dem jeder Knoten eine Variable ist und die Kanten ein constraint zwischen Variablen darstellen. $\Rightarrow$ hilft dabei das Problem zu abstrahieren und es zu verstehen.

1.3 Constraint-Arten

- Unary Constraint: nur eine Variable (zB: Variable x_1 darf nicht den Wert 4 annehmen)
- Binary Constraint: Variablenpaar (zB: $x_1 \neq x_2$)
- Higher-Order constraint: 3 oder mehrere Variablen
- Preferences/Soft-Constraint: Nicht verpflichtend, aber Ziel ist es so viele wie möglich einzuhalten.

1.4 Wie löst man CSPs?

1.4.1 Suche

Werte werden nacheinander den Variablen zugewiesen. Es werden alle Constraints geprüft. Sobald eine Constraint verstoßen wird, wird gebacktracked. Dies passiert solange, bis alle Variablen geeignete Werte haben.

1.4.2 Constraint Propagation

Für jede Variable X_i wird eine Menge möglicher Werte D_i gespeichert. Die Größe von D_i wird versucht zu reduzieren, indem Werte gesucht werden die Constraints nicht erfüllen.

1.5 Backtracking Search

Tiefensuche mit Einzelvariablen Assignments pro Level. Backtracking ist die Basic Uninformierte Suche für CSPs.

1.6 Heuristiken für CSPs

- Domain-spezifische Heuristik: kommt auf Charakteristiken der Probleme an
- General-purpose Heuristik: Es gibt gute Allgemeine Heuristiken für CSPs.
  - Minimum Remaining Values Heuristik: Variable gewählt wird mit den wenigsten konsistenten Werten.
  - Degree Heuristik: Variable mit den meisten Constraints auf den verbliebenen Variablen
  - Least Constraining Value Heuristik: Sei eine Variable gegeben, wähle einen Wert welches die geringste Anzahl an Werte für verbliebene Variablen ausstreicht.

1.7 Knotenkonsistenz

Eine Variable ist konsistent, wenn die möglichen Werte alle Unären Constraints erfüllen.

1.7.1 Lokale Konsistenz

Definiert durch einen Fraph wobei jeder Knoten Konsistent mit deren Nachbarn ist.

1.8 Forward Checking

Idee: Die verbleibenden erlaubten Werte für noch nicht zugewiesenen Variablen wird gespeichert und die Suche terminiert wenn keine Variable legale Werte mehr hat.

1.9 Arc-Konsistenz

Arc: eine Constraint welche zwei Variablen beinhaltet = binary Constraint

Arc-Consistency: eine Arc ist konsistent, wenn für jeden Wert von X in der Domain von X ein Wert Y in der Domain von Y existiert, sodass die Constraint arc(X, Y) erfüllt ist.

1.9.1 Maintaining Arc Consistency

Nach jeder Zuweisung eines Wertes zu einer Variable, werden die möglichen Werte von Nachbarn geupdated.

1.10 Pfad-Konsistenz

Arc-Konsistenz ist oft ausreichend um das Problem zu lösen oder zu zeigen, dass das Problem unlösbar ist. Es kann jedoch Fälle geben, in denen immer ein konsistenter Wert in der Nachbarschaft existiert, aber dennoch die gesamte Lösung inkonsistent ist.
Pfad-Konsistenz: verschärft binäre Constraints, indem Tripel von Variablen betrachtet werden. Das Ziel ist es weitere Eingrenzen von Werten durch Konsistenz zwischen Variablenpaaren und deren Nachbarn.

$\Rightarrow$ Konzept kann generalisiert werden durch k-Consistency, wobei Menge von k Werten konsistent sein müssen. Aber das ist im Wort Case exponentiell. Arc Consistency ist meist genutzte Technik.

1.11 Lokale Suche für CSPs

- **Modifikationen für CSPs**:
  - Arbeitet mit kompletten Zuständen.
  - Erlaubt Zustände mit nicht erfüllten Constraints.
  - Operatoren ordnen Variablenwerte neu zu.
- **Min-Conflicts Heuristik**:
  - Wählt zufällig eine Konfliktvariable.
  - Ordnet den Wert zu, der die wenigsten Constraints verletzt.
  - Nutzt Hill-Climbing mit $h(n) = \#$ der verletzten Constraints.
- **Performance**:
  - Kann zufällig generierte CSPs mit hoher Wahrscheinlichkeit lösen.
  - Schwierig in einem schmalen Bereich R, wo viele Konflikte auftreten.

1.12 The Power of Problem Decomposition

- **Annahme**:
  - Suchraum für ein CSP mit n Variablen und d Werten ist $O(d^n)$.
- **Idee**:
  - Zerlege das Problem in Subprobleme mit jeweils c Variablen.
  - Jedes Subproblem hat eine Komplexität von $O(d^c)$.
  - Es gibt n/c Subprobleme, was eine Gesamtkomplexität von $O(n/c \cdot d^c)$ ergibt.
- **Effekt**:
  - Reduziert die Komplexität von exponentiell in n auf linear in n, falls c konstant ist.

1.13 Tree-structured CSPs

- Ein CSP ist baumstrukturiert, wenn im Constraint-Graph zwei beliebige Variablen durch genau einen Pfad verbunden sind.
- **Theorem**:
  - Jedes baumstrukturierte CSP kann in linearer Zeit in Bezug auf die Anzahl der Variablen gelöst werden: $O(n \cdot d^2)$.

1.13.1 Linearer Algorithmus für Baumstrukturierte CSPs

1. Wähle eine Variable als Wurzel, ordne Knoten so, dass ein Elternteil immer vor seinen Kindern kommt.
2. Für $j = n$ bis 2:
   - Mache die Kante (X_i, X_j) arc-konsistent, rufe `REMOVE-INCONSISTENT-VALUE`(X_i, X_j).
3. Für $i = 1$ bis n:
   - Weise X_i jeden Wert zu, der konsistent mit seinem Elternteil ist.

1.14 Nearly Tree-structured Problems

- Baumstrukturierte Probleme sind selten.
- Zwei Ansätze, um Probleme baumstrukturiert zu machen:
  1. **Cutset Conditioning**:
     - Entferne Knoten, sodass der verbleibende Graph ein Baum wird.
  2. **Knoten zusammenfassen (Collapsing Nodes)**:
     - Zerlege den Graphen in unabhängige baumförmige Subprobleme.

1.14.1 Cutset Conditioning

1. Wähle eine Teilmenge S der Variablen, sodass der Constraint-Graph nach Entfernung von S ein Baum ist (Cycle Cutset).
2. Weise S konsistente Werte zu (alle möglichen Kombinationen ausprobieren).
3. Entferne inkonsistente Werte aus den verbleibenden Variablen.
4. Löse das restliche CSP (baumstrukturiert).
5. Falls keine Lösung, wähle eine andere Wertezuweisung für S.

Chapter 6 - Aussagenlogik

Einführung in die Künstliche Intelligenz
WiSe 2024/25

1 Logik Basics

Syntax: eine Sequenz einer spezifischen Sprache welche gefolgt werden sollen um einen Satz zu erzeugen. Syntax ist die Repräsentation einer Sprache und hat einen Bezug auf Grammatik und Struktur.
Semantik: Semantik definiert die Bedeutung des Satzes.
Logik: Logik ist der Schlüssel hinter (formalem) Wissen. Es erlaubt uns wichtige Information zu filtern und eine Schlussfolgerung zu formulieren. In KI, wird Wissensrepräsentierung durch Logik geschaffen.
Wissensbasis: Repräsentiert die Fakten die in der echten Welt existieren. Es die Zentrale Komponente eines wissensbasierten Agenten. Es besteht aus einer Menge von Sätzen, die die Information beschreiben.
Inferenz-Engine: Die Engine eines wissensbasierten Systems, welche es erlaubt neues Wissen zu schließen.
Intelligenze Agenten: ein Zielorientierter Agent. Es nimmt die Umgebung durch Sensoren durch Observierungen und built-in Wissen wahr. Durch Aktuatoren agiert es in der Umgebung.
⇒ Ein Wissensbasierter Agent muss in der Lage sein Zustände, Aktionen und weiteres zu repräsentieren, neue Wahrnehmungen einzubauen, eigene Weltrepräsentation zu updaten, verstecke Eigenschaften der Umgebung und passende Aktionen zu erschließen.

1.1 Backus Naur Form

- Symbol → P, Q, R, ...
- Sentence → True — False — Symbol — Not(Sentence) — (Sentence AND Sentence) — (Sentence OR Sentence) — (Sentence ⇒ Sentence) — (Sentence ⇔ Sentence)

1.2 Semantik

Eine Interpretation spezifiziert welche Symbole Wahr und Falsch ergeben. Ein Modell ist eine Interpretation in der alle Sätze wahr sind.
Tautologie: eine Allgemeingültige Aussage.
Logische Äquivalenz: Zwei Sätze sind logisch Äquivalent, wenn Sie die selben Wahrheitswerte haben für jede Variabelsetzung.

2 Logische Äquivalenzen Beispiele

- (A OR B) ≡ (B OR A) — Kommutativität (analog mit AND)
- ((A AND B) AND C) ≡ (A AND (B AND C)) — Kommutativität (analog mit OR)
- NOT(NOT(A)) ≡ A
- (A ⇒ B) ≡ (NOT(B) ⇒ NOT(A)) — Kontraposition
- (A ⇒ B) ≡ (NOT(A) OR B) — Implikationseliminierung
- NOT(A AND B) ≡ (NOT(A) OR NOT(B)) — DeMorgan analog mit OR zu AND
- (A AND (B OR C)) ≡ ((A AND B) OR (A AND C)) — Distributivität Analog mit OR (AND))

3 Inferenz

Die Logische Folgerung die man durch das betrachten des erlangten Wissens beschließt. Um zu prüfen ob eine Aussage A durch Wissensbasis inferiert wird (Wahr ist), könnte man alle Belegungen prüfen, aber das dauert zu lange.
⇒ Reasoning Patterns: Man kann aus gegebenem Wissen aus der Wissensbasis neues Wissen schließen durch Reasoning Patterns.

3.1 Modus Ponens

$A, A \Rightarrow B$ dann wissen wir, es gilt: B

4 Konjunktive Normal Form

Verundung von Oders. (Literal1 OR Literal2 OR Literal3) AND (Literal4 OR Literal5 OR Literal6) AND (Literal7 OR Literal8 OR Literal9).

4.0.1 Einheitsresolution

Wenn gegeben: I1 OR I2 OR ... OR Ik und NOT(Ii), dann können wir folgendes schließen:
I1 OR I2 OR ... OR Ii-1 OR Ii+1 OR ... OR Ik

⇒ Allg. Resolution: Wenn man in einer KNF ein NOT(m) gegeben hat, kann man dieses Streichen.

4.1 Satisfiability

Es existiert ein Modell welche die angepasste Wissensbasis erfüllt. Folgender Algorithmus wird angewendet, mit der Annahme es ist eine Formel in KNF gegeben:

- Finde zwei Klauseln mit Komplementären Literalen
- Wende Resolution an
- Füge resultierende Klausel hinzu, falls diese nicht da ist
- Teste. Wenn das Resultat die leere Klausel ist, dann ist die Formel nicht satisfiable.

5 Horn Klauseln

Horn Klauseln sind Implikationen, welche aus nur positiven Literalen bestehen. Oder auch: eine KNF mit höchstens einer Positiven Klausel.

6 Limitation von Aussagenlogik

Quantitative Dinge sind schwer zu beschreiben. Man kannt den Begriff der Objekte und Relation zwischen Objekten nicht in der Aussagenlogik.

Chapter 7 - Prädikatenlogik 1. Ordnung

Einführung in die Künstliche Intelligenz
WiSe 2024/25

1 First-Order Log

1.1 Elemente von FOL

- Objekte: Mitbewohner1, Mitbewohner2, ..
- Relationen (2 Parameter): Freunde(Mitbewohner1, Mitbewohner2)
- Funktionen (1 Parameter): Männlich(Mitbewohner2)
- Gleichheiten: Mitbewohner(Mitbewohner1) = Mitbewohner2

Funktionen können verwendet werden um Datenstrukturen zu definieren oder Integers. Eine Funktion kann auf jedes Objekt angewendet werden.

1.2 Forall and Exists

$\forall x \rightarrow$ für Alle Elemente x $\in X$
$\exists x \rightarrow$ es existiert ein Element x, für das gilt: ..
Axiom: ein basic Fact über eine Domain (initiale KB)
Theorem: Aussagen die logisch von Axiomen abgeleitet werden.

$\forall a : a \equiv \neg \exists a : \neg a$

1.3 Substitution

$\Rightarrow$ ersetzt eine oder mehrere Variablen mit etwas anderem in einem Satz.

Skolemkonstante: eine Konstante welche nicht woanders in der Knowledge Base vorkommt

1.4 Ablauf CNF First-Order

1. Convert to Negation Normal Form (Negation nur direkt vor Prädikaten)
2. Folie 20

3. Folie 20
4. Folie 20

2 Gödels Incompleteness Theorem

1. FOL ist nicht ausgeprägt genug um grundlegende Arithmetik zu modellieren
2. Für jedes konsistente System von Axiomen welches ausgeprägt genug ist um grundlegende Arithmetic.... (FOlie 32)
3. Folie 32

Chapter 8 - Ungewissheit

WiSe 2024/25

1 Ungewissheit

- Wahrscheinlichkeiten: Eine Art um mit Ungewissheit umzugehen.

Wahrscheinlichkeiten können auch mit einem subjektiven Glaube zusammenhängen. Eine Wahrscheinlichkeit p bedeutet, dass man glaubt, dass eine Aussage wahr ist in $p * 100\%$ der Fälle. Wahrscheinlichkeitstheorie ist über den Grad des Glaubens nicht des Grads der Wahrheit.

1.1 Grundlagen

Der Zustandsraum ist die Menge aller möglichen Ergebnisse. Ein Wahrscheinlichkeitsraum ist ein Zustandsraum mit einer Zuweisung von Wahrscheinlichkeiten für jedes Mögliche Ergebnis.

1.2 Kolmogorov's Axiome von Wahrscheinlichkeit

1. Alle Wahrscheinlichkeiten sind zwischen 0 und 1.
2. Zwangsweise wahre Aussagen haben die Wahrscheinlichkeit 1. Wenn sie zwangsweise falsch sind, ist die Wahrscheinlichkeit 0.
3. Die Wahrscheinlichkeit einer Disjunktion ist $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$
4. Diese Axiome schränken die Menge der probabilistischen Überzeugungen ein, die ein Akteur haben kann.

1.3 Dutch Book Theorem

ein Agent der nach Wahrscheinlichkeiten wettet, die gegen die Axiome der Wahrscheinlichkeit verstoßen, kann gezwungen werden, so zu wetten, dass er Geld verliert unabhängig vom Ergebnis.

1.4 Variablen

Eine zufällige Variable ist eine Funktion zwischen atomaren Ereignissen und einem Wertebereich. Die Wahrscheinlichkeitsfunktion P über atomaren Ereignissen, liefert eine Wahrscheinlichkeitsverteilung über alle Zufallsvariablen X.

$$P(X = x_i) = \sum_{w:X(w)=x_i} P(w)$$

- Propositionale oder boolesche Zufallszahlen können wahr oder falsch sein
- Diskrete Zufallsvariablen: (einen bestimmten Wert des Zustandsraums)
- Kontinuierliche Zufallsvariablen (begrenzt oder unbegrenzt): (Temp ¿ 23)

1.5 Gemeinsame Verteilungen

Eine gemeinsame Verteilung gibt die Wahrscheinlichkeit von kombinierten Ereignissen an.

Zum Beispiel: $P(x, y) \equiv P(X = x \wedge Y = y)$ ist die Wahrscheinlichkeit dass $X = x$ und $Y = y$ wahr sind.

1.6 Ausgrenzung

Für jede beliebige Menge von Variablen X und Y kann man die Wahrscheinlichkeit berechnen: $P(Y) = \sum_{i=1}^{n} P(x_i, Y)$.

Die resultierende Verteilung ist Randverteilung und die Wahrscheinlichkeiten sind Randwahrscheinlichkeiten.

1.7 Bedingte Wahrscheinlichkeit

Die Wahrscheinlichkeit von $X = x$, mit der Annahme, dass $Y = y$ wahr ist.

$$P(x|y) = \frac{P(x \wedge y)}{P(y)}$$

Die Produkt Regel ermöglicht eine alternative Formel: $P(x, y) = P(x|y) * P(y) = P(y|x) * P(x)$

Die Kettenregel folgt aus nachfolgender Anwendung von der Produktregel.

$$P(X_1, X_2, ..., X_n) = \Pi_{i=1}^{n} P(X_i|X_1, ..., X_{i-1})$$

1.8 Unabhängigkeiten

X und Y sind unabhängig voneinander, wenn eines der folgenden gilt:

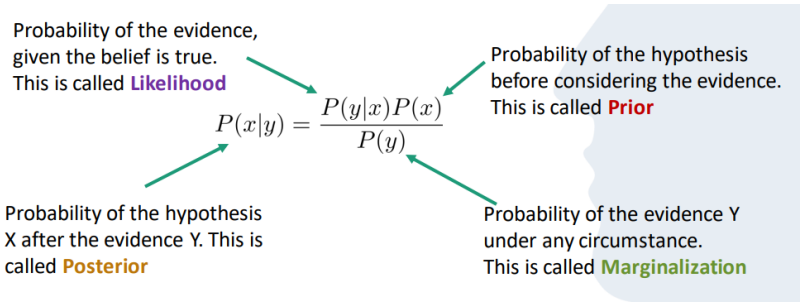
$$P(X|X) = P(X)$$

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

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

Eine unabhängige Variable ist nicht von anderen Variablen beeinflusst. Hierdurch wird die Menge an möglichen Werte verringert und die Komplexität sinkt. Aber absolut Unabhängige Variablen sind rar.

1.9 Bayes Regel



Chapter 9 - Bayes'sche Netzwerke

WiSe 2024/25

1 Naive Bayes Model

Ein naives Bayes-Modell geht davon aus, dass alle Auswirkungen unabhängig von der Ursache sind.

1.1 Bayes'sche Netzwerke

Ein Bayes-Netz sind einfache grafische Notationen für bedingte Unabhängigkeitsaussagen. Ein BN ist ein gerichteter azyklischer Graph mit Knoten (ein Knoten pro Variable) und Kanten. Eine Kante von Variable zu Variable induziert, dass ein Einfluss einer Variable auf eine weitere besteht.

Bedingte Wahrscheinlichkeitsverteilung: $P(X_i|Parent(X_i))$
Joint Distribution: $P(X_1, ..., X_n) = \Pi_{i=1}^{n} P(X_i|Parent(X_i))$

Lokale Markov Annahme: Jede Zufallsvariable X ist unabhängig von ihrer nicht-absteigenden Variante, da sie von ihren Eltern abstammt.

1.2 Bedingte Unabhängigkeit

Zwei Ereignisse oder Variablen sind unabhängig, wenn bestimmte Informationen gegeben sind. Wenn diese Information nicht gegeben ist, sind diese nicht unbedingt Unabhängig.

$$P(A|B, C, D) = P(A|B)$$

Wahrscheinlichkeit von A, mit Annahme das B stimmt und C und D gegeben ist. Wenn C und D unabhängig sind bzgl. A—B, dann kann man auch einfach die Wahrscheinlichkeit von A mit Annahme das B wahr ist berechnen.

1.3 Variable Eliminierung

Eine Möglichkeit um die Inferenz in BNs effizienter zu gestalten. Hierbei wird die Wahrscheinlichkeit eines Ereignisses berechnet, indem andere Variablen systematisch eliminiert werden.

Ziel: Wahrscheinlichkeit $P(X|e)$ zu berechnen, wobei:

- X die Zielvariable ist, deren Wahrscheinlichkeit man wissen möchte
- e Evidenzvariblen sind und andere Variablen im Netzwerk die keine Evidenz sind, eliminiert werden.

1.3.1 Schritte

1. Repräsentation als Faktoren: Wahrscheinlichkeiten im Bayes-Netz werden als Produkte von lokalen Faktoren dargestellt.
2. Evidenz einbeziehen: Integriere Evidenzvariablen in die Faktoren. Faktoren werden dadurch vereinfacht.
3. Variable eliminieren: Nicht Zielvariablen (die auch keine Evidenzvariablen sind), werden eliminiert. Dies geschieht durch Marginalisierung, also das Summieren über die möglichen Werte der Variablen: $\phi'(Y) = \sum_Z \phi(Y, Z)$. Hierbei wird Z eliminiert.
4. Faktoren multiplizieren: Alle relevanten Faktoren, die die zu eliminierende Variable enthalten, multipliziert. Danach wird die Variable durch Marginalisierung eliminiert.
5. Zielwahrscheinlichkeit berechnen: Es bleiben Faktoren übrig, die nur die Zielvariable und Evidenz betreffen.

1.3.2 Problem:

Inferenz in Bayes Netzen ist NP-hart

1.4 Sampling

Anstatt Wahrscheinlichkeiten exakt zu berechnen, generiert man große Stichprobenanzahlen aus dem Bayes Netz, die mit der zugrunde liegenden Wahrscheinlichkeitsverteilung übereinstimmen. Aus diesen Stichproben werden dann Wahrscheinlichkeiten durch Zählen geschätzt.

1.4.1 Prior-Sampling

Man generiert Stichproben, indem die Variablen in der topologischen Reihenfolge des Netzes durchlaufen werden. Jede Variable wird also erst betrachtet, nachdem ihre Eltern gesamplet wurden.

Schritte

1. beginne mit Variable ohne Eltern und ziehe einen Wert gemäß ihrer prioren Verteilung
2. Ziehe dann Werte für abhängige Variablen gemäß ihrer bedingten Verteilungen, basierend auf den gesampelten Werten der Eltern.

1.4.2 Markov Chain Monte Carlo Sampling

Man konstruiert eine Markov-Kette, deren Zustände den möglichen Werten der Variablen entsprechen und deren Übergangswahrscheinlichkeiten so gewählt werden, dass die Zustände der Kette asymptotisch die Zielwahrscheinlichkeit repräsentieren. Nach einer Aufwärm-Phase werden Zustände als Stichproben verwendet, um die Zielverteilung zu approximieren.

Markov-Kette: Sequenz von Zuständen, bei der der nächste Zustand nur vom aktuellen Zustand abhängt. (Markov-Eigenschaft)

1.4.3 Gibbs-Sampling

Eine MCMC-Technik, welches sich für Bayes Netze eignet.

1. Wähle Startkonfiguration aller Variablen
2. Wiederhole für jede Variable X_i: Ziehe einen neuen Wert für X_i, basierend auf der bedingten Verteilung $P(X_i|\text{alle anderen Variablen})$, während die anderen Variablen festgehalten werden.
3. Wiederhole den Prozess, bis die Markov-Kette konvergiert.

Was ist eine konvergierende Markov-Kette: Markov-Kette erreicht mit zunehmender Anzahl von Schritten eine stationäre Verteilung.

Was ist eine Markov-Blanket? Eine Markov-Blanket einer Variablen ist die minimal notwendige Menge von Variablen, die eine Zielvariable X von allen anderen Variablen unabhängig macht. Die Markov Decke besteht aus: Parents, Kindern, und den Eltern der Kinder.

Wann ist es garantiert, dass Gibbs Sampling konvergiert?: Es konvergiert nur, wenn für die Kette gilt:

- jeder Zustand ist erreichbar von jedem anderen Zustand
- aperiodisch
- es zu jedem Zustand zurückkehrt mit Wahrscheinlichkeit 1

Chapter 10 - Maschinelles Lernen und Neurale Netze

WiSe 2024/25

- Was ist Lernen?
- Was ist deklaratives und imperatives Wissen?
- Was ist induktives Lernen?
- Was ist der Ockham's Razor?
- Wie kann man Overfitting vermeiden?
- Was ist ML?
- Worin unterscheiden sich traditionelles Programmieren und ML?
- Worin unterscheiden sich Menschliches und maschinelles Lernen?
- Wo findet ML Anwendung?
- Wie baut man ein Lernendes System auf?
- Welche Arten von Lernen gibt es?
- Was ist Feature Engineering?
- Welche Regressionsansätze gibt es in der ML?
- Welche Klassifikationsansätze gibt es in der ML?
- Wie wird ML evaliuert?
- Wie werden Feature Vektoren erstellt?
- Mean Squared Error
- Was machen bei Overfitting?
- Was machen bei Underfitting?
- Wo ist die goldene Mitte? Wieso wird diese bevorzugt?
- **Watch the Youtubge Video on Slide 30**

Chapter 11 - Reinforcement Learning

WiSe 2024/25

1 Reinforcement Learning

Reinforcement Learning Algorithmen versuchen eine Policy zu finden, die die maximale kummulierte Reward für den Agenten liefern. Dies wird als Markov Decision Prozess repräsentiert.

1.1 Policy

Die Strategie, nach der ein Agent seine Aktionen auswählt, um optimale Belohnung zu erreichen.

Die Policy wird als Funktion dargestellt, die den Zustand einer Umgebung auf eine Aktion abbildet.

$$\pi(s) = a$$

1.1.1 Deterministische Policy

Policy wählt für jeden Zustand die selbe Aktion.

1.1.2 Stochastische Policy

Policy gibt eine Wahrscheinlichkeitsverteilung über mögliche Aktionen an.

1.2 Markov Decision Processes

Ein Mathematisches Modell zur BEschreibung von Entscheidungsproblemen in stochastischen Umgebungen. Es wird verwendet, um die Interaktion eines Agenten mit seiner Umgebung zu modellieren.

Ein Markov Decision Process wird als ein Tupel definiert:

$$\text{MDP} = (S, A, T_a, R_a)$$

S : eine Menge von Zuständen $s \in S$
A : eine Menge von Aktionen $a \in A$
T_a : eine Übergangsfunktion $T(s, a, s')$
R_a : eine Belohnungsfunktion $R(s, a, s')$

T_a ist die Wahrscheinlichkeit dass die Aktion a von s zu s' führt.
$\Rightarrow$ $P(s'|s,a)$ ist das Modell. Ein Model ist eine Darstellung der Umgebung, die es ermöglicht, vorherzusagen, wie sich die Umwelt auf Aktionen des Agenten ändert.

Außerdem wird ein Startzustand, ein Endzustand und ein Discountfaktor γ gegeben. Der Discountfaktor ist eine Zahl zwischen 0 und 1, die bestimmt, wie stark zukünftige Belohnungen im Vergleich zu unmittelbaren Belohnungen gewichtet werden.

1.2.1 Was bedeutet Markovian?

Es bedeutet, dass der gegenwärtige Zustand, die Zukunft und Vergangenheit unabhängig sind. Also wenn du die Gegenwart kennst, braucht du nicht wissen wie die Vergangenheit aussah, um eine gute Aktion zu wählen.

1.2.2 Wieso werden zukünftige Belohnungen verringert mit dem Discountfaktor?

Zukünftige Belohnungen sind weniger wert als sofortige, um sicherzustellen, dass der Agent konvergiert. Für jeden Zustand wird ein Nutzwert berechnet, der die Summe aller zukünftigen Discounted Werte ist.

1.2.3 Wieso möchte man Optimale Utility-Werte berechnen?

Weil Optimale Werte zu optimalen Policies führen.

1.2.4 Value Funktion und Action-Value Funktion

- Value Funktion $V(s)$: Definiert den Wert eines Zustands, bei optimalen handeln.
- Action-Value Funktion $Q(s,a)$: Wert eines Zustands, wenn zuerst Aktion a ausgeführt wird und danach optimal gespielt wird.

$\Rightarrow$ alle optimalen Policies haben dieselbe optimale State-Value Funktion v^* und Action-Value Funktion q^*.

1.2.5 Bellman Equation of Optimality

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) * V^{\pi}(s')$$

V^{π} : Wert von Zustand s wenn nach der Policy π gehandelt wird
$R(s, \pi(s))$: Direkte Belohnung
γ : Discountwert
$T(s'|s, \pi(s))$: Übergangswahrscheinlichkeit dass der Agent in s' ankommt

1.2.6 Wie bestimmt man die Value-Function

Die Value Function $V^*(s)$ kann mit Value Iteration berechnet werden. Value Iteration ist eine Methode aus der dynamischen Programmierung, die die Bellman-Gleichung iterativ anwendet, bis die Werte konvergieren.

1. **Initialisierung**
   Starte mit $V_0^*(s) = 0$ für alle Zustände s.
2. **Iterative Berechnung**
   Berechne für jeden Zustand s den neuen Wert $V_{k+1}(s)$ mit der Bellman-Update-Gleichung:

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

$T(s, a, s')$: Übergangswahrscheinlichkeit zum nächsten Zustand
$R(s, a, s')$: Belohnung für Aktion a bei Zustand s
γ : Discountfaktor
$\max_a$: bedeutet, dass die beste Aktion gewählt wird

3. **Wiederholen bis zur Konvergenz**
   Bist sich die Werte kaum ändern

Theorem: Value Iteration konvergiert zu einzigartigen optimalen Werten.
Werte verbessern sich schrittweise und nähern sich der optimalen Lösung. Die optimale Strategie kann sich schon vor der vollständigen Konvergenz der Werte stabilisieren.

1.3 Definitionen

- **Model-basiert:** Lerne das Model und nutze es um die Optimale Policy zu finden

- **Model-frei:** Finde die optimale Policy ohne das Model zu lernen.
- **On-policy:** Agent lernt durch die Policy die es verwendet
- **Off-policy:** Agent lernt über Policy oder Policies die sich von der eigentlichen Policy unterscheiden
- **Passives Lernen:** Der Agent beobachtet einfach die Welt und versucht, die Vorteile der verschiedenen Zustände zu erlernen. Ziel ist es, eine festgelegte Strategie (Abfolge von Handlungen) auszuführen und diese zu bewerten.
- **Aktives Lernen:** Der Agent beobachtet nicht nur, sondern er handelt auch. Das Ziel ist es, zu handeln und eine optimale Politik zu lernen
- **Model-basiertes Lernen:** Lerne das Modell empirisch durch Erfahrung. Löse Werte als ob das gelernte Model korrekt ist.

1.4

Sample-Based Policy Evaluation

Methode um die Value Function für eine gegebene Policy zu schätzen, ohne die vollständigen Übergangswahrscheinlichkeiten der Umgebung zu kennen.
Anstatt die erwarteten Werte über alle möglichen zukünftigen Zustände s' exakt zu berechnen, werden Erfahrungswerte (Samples) genutzt.

1. Statt Bellman-Gleichung, Stichprobenbasierte Näherung
   Da $T(s, \pi(s), s')$ oft nicht bekannt ist, nimmt man Stichproben aus realer Interaktionen

$$\begin{aligned} \text{sample}_1 &= R(s, \pi(s), s'_1) + \gamma V_k^\pi(s'_1) \\ \text{sample}_1 &= R(s, \pi(s), s'_2) + \gamma V_k^\pi(s'_2) \\ &\vdots \\ \text{sample}_n &= R(s, \pi(s), s'_n) + \gamma V_k^\pi(s'_n) \end{aligned}$$

   Der neue Wert $V_{k+1}^\pi(s)$ wird dann als Mittelwert dieser Stichproben berechnet:

$$V_{k+1}^\pi(s) = \frac{1}{n} \sum_i \text{sample}_i$$

1.5 Temporal-Difference Learning

Ermöglicht Value Function $V^\pi(s)$ zu aktualisieren, ohne die Übergangswahrscheinlichkeiten der Umgebung explizit zu kennen. Sie kombiniert Ideen aus Monte Carlo Methoden (lernen aus realen Erfahrungen) und dynamischer Programmierung (Verwendung von Erfahrungen) und dynamischer Programmierung (Verwendung von Wertschätzungen während des Lernens).

Sample of V(s):	$sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$
Update to V(s):	$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$
Same update: (Explains the term TD)	$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Probleme mit TD-Value Learning: Es fehlen Q-Werte für Aktionsauswahl. Policy wird normalerweise durch Q-Values bestimmt. Aber diese werden nicht berechnet und so kann nicht die beste Aktion ausgewählt werden.
Lösung ⇒ Q-Werte.

1.6 Aktives Lernen

Bei aktivem Lernen kennt man die Übergänge nicht, die Belohnungen nicht und kann jede beliebige Aktion wählen. Basically Trial and Error.

1.7 Q-Learning

Modellfreie, wertbasierte Methode. Genutzt um eine optimale Policy zu lernen, indem es Q-Werte für jede Kombination aus Zustand und Aktion schätzt.
Ziel: Q-Funktion erlernen, die die kumulierten Reward man erhält, wenn man eine Aktion ausführt und danach optimal handelt.

$$\text{Optimale Strategie } \pi^*(s) = \arg\max_a Q^*(s,a)$$

⇒ wählt immer Aktion mit höchsten Q-Wert

Q-Learning Update-Regel

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

$Q_k(s,a)$: bisherige Schätzung für Q-Wert
α : Lernrate
$R(s,a,s')$: Belohnung für Aktion a in Zustand s
γ : Discountfaktor
$\max_{a'} Q_k(s',a')$: Wert der besten möglichen Aktion im nächsten Zustand s'

1.8 Exploration and Exploitation Trade-Off

- Exploration: neue Aktionen ausprobieren, um mehr über die Umgebung zu lernen
- Exploitation: beste bekannte Aktion ausführen, um sofort eine hohe Belohnunge zu erhalten

Dilemma

- Zu viel Exploration: Agent sammelt viel Wissen, bekommt aber möglicherweise lange Zeit keine hohen Belohnungen
- Zu viel Exploitation: Agent nutzt immer wieder die beste bisher bekannte Aktion, könnte aber eine noch bessere Möglichkeit übersehen

1.9 Deep Q-Networks

Erweiterung von Q-Learning, die anstelle einer Q-Tabelle ein tiefes NN verwenden. Dies ermöglicht das Lernen in großen und kontinuierlichen Zustandsräumen, wo klassische Q-Tabellen zu groß oder unpraktikabel wären.

1.10 Policy Search

Methode bei der die Policy direkt gelernt wird, anstatt eine Value Function.

1.10.1 Warum Policy Search?

Beste Policy ist nicht immer diejenige, die eine optimale Wertfunktion approximiert. In komplexen Umgebungen kann das Optimieren der Q-Funktion ineffizient sein, wenn es zu viele Zustände gibt.

Um dieses Problem zu lösen kann die optimale Policy direkt gelernt werden. Es optimiert direkt die Wahrscheinlichkeiten für Aktionen. Policy $\pi(s|s, \theta)$ wird durch Parameter θ beschrieben und mit Gradientenverfahren verbessert.

1.10.2 Probleme von Policy Search

- **Wie misst man ob eine Policy besser geworden ist?**
  Man muss viele Episoden ausführen
- **Hoher Rechenaufwand**
- **viele Features erschweren das Training**

2 AlphaZero

2.1 Komponenten von AlphaZero

- Deep Learning Network: Value und Policy Network
- Tree Search Algorithm: Predictor Upper Confidence Bounds for Trees Algorithm
- Neural Network Optimization: Supervised Training

2.2 Formulierung von Verlust

$$l = \alpha(z - v)^2 - \pi^T \log p + c||\theta||^2$$

$\alpha(z - v)^2$: Mean Squared Error
$\pi^T \log p$: Cross Entropy
$c||\theta||^2$: Regulizer

α : Value Loss Factor
z : Target Value
v : Predicted Value
π^T : MCTS simulation distribution
p : Policy Head output
c : L_2 Regularization constant

2.3 Such-Prinzipien von AlphaZero

AlphaZero limitiert die Tiefensuche mithilfe der Value Prediction. Und es reduziert die Breite der Suche durch Policy Prediction.

2.4 PUCT-Algorithm

Predictor Upper Confidence Bounds for Tree Algorithm

- Exploration by rollouts
- Move selection:
$$a_t = \text{argmax}_a(Q(s_t, a) + U(s_t, a))$$
- Update Q-Values by simple moving average
$$Q'(s_t, a) \leftarrow Q(s_t, a) + \frac{1}{n}[v - Q(s_t, a)$$

2.4.1 Monte-Carlo Tree Search Phases

- Selection
- Expansion Evaluation
- Backpropagation
- Back to Selection