# Leitfaden für die Spezifikation komplexer Systeme

des Kooperationsprojektes


# CyberProtect

| | |
|---|---|
| Veröffentlichung: 29.2.2020 | Version: 0.2 |
| Autoren: Jonas Klamroth | |
| Review: Max Scheerer | |

Baden-Württemberg
MINISTERIUM FÜR WIRTSCHAFT, ARBEIT UND WOHNUNGSBAU

Gefördert vom Ministerium für Wirtschaft, Arbeit und Wohnungsbau Baden-Württemberg

Aktenzeichen Zuwendungsbescheid:
3-4332.62-FZI/53

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Glossar

is makeindex, version 2.15 [TeX Live 2019] (kpathsea + Thai support). Scanning style file ./dokument.ist.....done (29 attributes redefined, 0 ignored). Scanning input file dokument.glo....done (22 entries accepted, 0 rejected). Sorting entries....done (96 comparisons). Generating output file dokument.gls....done (32 lines written, 0 warnings). Output written in dokument.gls. Transcript written in dokument.glg. mat4

**LTL** Linear Temporal Logic, Spezifikationssprache für temporale Eigenschaften.

**ML** Machine Learning (maschinelles Lernen), Methoden der Informatik die es erlauben das Lösen von Aufgaben zu lernen statt explizite Lösungsstrategien anzugeben, siehe 2.3 für eine ausführliche Erklärung.

**NN** Neural Network (neuronales Netz), eine ML-Methode, ursprünglich inspiriert durch biologische neuronale Netze deswegen auch manchmal künstliche neuronale Netze genannt (Artificial Neural Network).

**RL** Reinforcement Learning (bestärkendes Lernen), eine ML-Methode um Agenten in unbekannten Umgebungen Aufgaben beizubringen, siehe 2.6 für eine ausführliche Erklärung.

**SL** Supervised Learning (überwachtes Lernen), eine Kategorie von ML die auf Grund von gegebenen Eingabe-Ausgabe-Paaren lernt, siehe 2.4 für eine ausführliche Erklärung.

# 1 Motivation

Das Thema Künstliche Intelligenz (KI) hat in letzter Zeit einen Anstieg in Popularität erlebt. Das ist darauf zurückzuführen, dass es mehrere Durchbrüche in Bereichen wie Go, Schach oder selbstfahrende Autos erzielt wurden ([2], [3]), die für großes Aufsehen sorgten. Diese Entwicklungen wecken die Hoffnung, dass Aufgaben wie Autofahren, das Verstehen menschlicher Sprache und vieles mehr in (naher) Zukunft von Computern erledigt werden könnte. Während diese Errungenschaften Potenzial für einen enormen sozialen und wirtschaftlichen Nutzen bieten, werfen sie auch Fragen nach ihrer Zuverlässigkeit und Vertrauenswürdigkeit auf. Dies führt zu einer wachsenden Nachfrage nach Qualitätssicherungsmaßnahmen für KI-basierte Systeme. Insbesondere eine Gruppe renommierter Forscher der Region veröffentlichte einen offenen Brief, in dem sie um mehr Forschung/Finanzierung im Bereich zuverlässiger KI-Systeme bat [4].

Um sichere Systeme zu erreichen, muss der erste Schritt jedoch immer darin bestehen, irgendwie die Eigenschaften zu spezifizieren, die ein System haben muss, um als sicher/zuverlässig zu gelten. In diesem Sinne ist es das Ziel dieses Dokuments die derzeit verfügbaren/gebräuchlichen Techniken zur Spezifizierung von KI-basierten Systemen vorzustellen und diese bis zu einem gewissen Grad zu erweitern.

Das Dokument wird wie folgt gegliedert sein: Zunächst werden wir den Begriff “Künstliche Intelligenz” diskutieren und mehrere Teilbereiche aufzählen, die unter diesen Begriff fallen. Dann werden wir erläutern, welche aktuellen Probleme/Angriffe gegen diese Art von Systemen bekannt sind und welche Eigenschaften in der Regel untersucht werden, um diese Probleme zu lösen. Danach werden wir die gebräuchlichsten Methoden zur Spezifizierung von Anforderungen für diese Art von Systemen vorstellen. Darüber hinaus werden wir Erweiterungen der vorgestellten Ansätze vorstellen, um die Zugänglichkeit und die Verständlichkeit zu erhöhen. In einem letzten Kapitel werden wir Guidelines für die Spezifikation und Modellierung von Systemen vorstellen.

# 2 Grundlagen

In diesem Abschnitt stellen wir einige Grundlagen vor, auf die wir uns in den folgenden Kapiteln beziehen. Hauptsächlich führen wir ein, was wir unter den Begriffen “Künstliche Intelligenz” und “Maschinenlernen” verstehen und erklären, auf welche der Bereiche dieser Themen wir im Detail eingehen werden.

## 2.1 Künstliche Intelligenz

In diesem Abschnitt geben wir eine kurze Einführung in den Bereich der künstlichen Intelligenz (AI), um später interessante Eigenschaften für Spezifikationen aufführen zu können. Dieser Abschnitt ist keineswegs als vollständig zu verstehen. Wir möchten uns vielmehr speziell auf die Eigenschaften und Details konzentrieren, die für dieses Dokument von Interesse sein werden. Für eine vollständigere und tiefere Einführung in dieses Thema verweisen wir den interessierten Leser auf entsprechende Literatur [5, 6].

KI ist ein weit verbreiteter Begriff für sehr unterschiedliche Konzepte und Systeme. Definitionen für KI variieren extrem stark. Um zwei Beispiele zu nennen, betrachten Sie die Definition aus dem merriam-webster Onlinewörterbuch [1]:

> **artificial intelligence**: a branch of computer science dealing with the simulation of intelligent behavior in computers
>
> **künstliche Intelligenz (Übersetzung)**: ein Zweig der Informatik, der sich mit der Simulation von intelligentem Verhalten durch Computer beschäftigt

oder im Vergleich dazu der entsprechende Eintrag im Cambridgewörterbuch [2]:

> **artificial intelligence**: the study of how to produce machines that have some of the qualities that the human mind has, such as the ability to understand language, recognize pictures, solve problems, and learn
>
> **künstliche Intelligenz (Übersetzung)**: die Wissenschaft, wie man Maschinen produziert, die einige der Eigenschaften haben, die der menschliche Geist hat, wie z.B. die Fähigkeit Sprache zu verstehen, Bilder zu erkennen, Probleme zu lösen und zu lernen

Ein interessanter Unterschied zwischen diesen beiden Definitionen ist der Begriff der “Menschlichkeit”, der in der zweiten Definition eingeführt wird. Während sich die erste Definition ausschließlich ïntelligentes Verhaltenërwartet, erfordert die zweite Definition Ëigenschaften, die der menschliche Geist hat". Unabhängig von der genauen Definition ist der Bereich der Systeme, Algorithmen und Techniken, die als KI klassifiziert werden können, riesig. Um dieses Feld besser überblicken zu können werden wir versuchen, es in mehrere Unterbereiche zu unterteilen. Die erste Unterscheidung die gemacht werden kann, ist die zwischen regelbasierten Systemen und gelernten Systemen. Dabei besteht der Unterschied darin, dass erstere auf Grundlage vordefinierter Regeln handeln und diese auf gegebenen Eingaben anwenden, während der zweite Typ, gegeben eine Menge von Trainingsdaten, sein Verhalten selbstständig "lerntünd dann hoffentlich in der Lage ist, dies auf ungesehene Daten zu übertragen.

In Abb. 2.1 stellen wir den Zusammenhang der verschiedenen Begriffe, die wir im Folgenden einführen werden, graphisch dar. Wie Sie sehen, ist KI der Oberbegriff, der alle Unterbegriffe umfasst. KI kann dann wie erwähnt in regelbasierte KI und ML unterteilt werden. ML kann dann wiederum weiter unterteilt werden. Auf die Unterbegriffe von ML gehen wir genau im Abschnitt 2.3 ein.

[1] https://www.merriam-webster.com/dictionary/artificial%20Intelligenz, Stand 28.1.2020
[2] https://dictionary.cambridge.org/dictionary/english/artificial-intelligence, Stand 28.1.2020

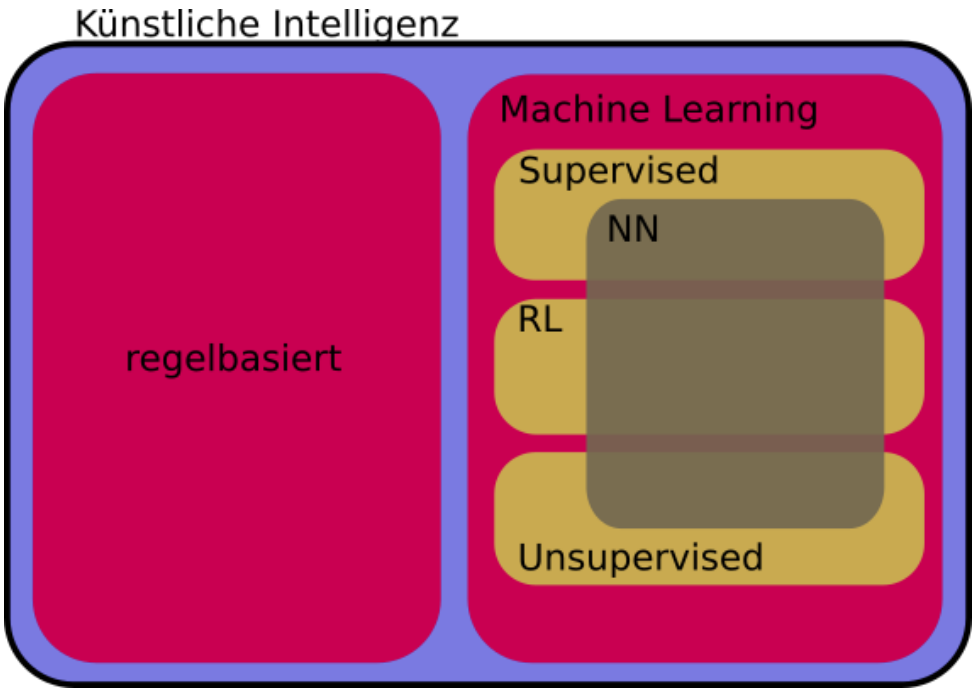


Abbildung 1: Eine Veranschaulichung der Beziehungen zwischen verschiedenen Begriffen im Bereich der KI

## 2.2 Regelbasierte KI

Dieser Teilbereich baut auf der Prämisse auf, dass Intelligenz als Menge von Regeln und Wissen und einer darüber liegende Logik erfasst werden kann. Es gibt viele verschiedene Ansätze dazu, wie diese Regeln und Wissensdatenbanken aufgebaut und genutzt werden, aber sie haben gemeinsam, dass vordefinierte Regeln verwendet werden, um Schlussfolgerungen zu ziehen und somit Intelligenz zu simulieren. Die vielleicht bekannteste Anwendung dieser Techniken sind Expertensysteme. Expertensysteme wurden für eine Vielzahl von unterschiedlichen Aufgaben eingesetzt, wie bspw. die Untersuchung von medizinischen Aufzeichnungen, Fehlersuche in komplexen Systemen und Erdbebenwarnsystemen.

## 2.3 Maschinelles Lernen

Maschinelles Lernen ( ML) hat mehrere Unterbereiche. Eine Vielzahl von Ansätze und Algorithmen können als maschinelles Lernen kategorisiert werden. In diesem Dokument werden wir einen Ansatz als ML-Ansatz bezeichnen, wenn er nicht auf explizite Anweisungen zur Erfüllung einer Aufgabe angewiesen ist, sondern aus den gegebenen Trainingsdaten sein Verhalten selbstständig lernt. Solche Ansätze umfassen Hidden-Markov-Modelle (HMMs), Dynamisch-bayesische Netzwerke (DBNs) und den wohl bekanntesten Ansatz: neuronalen Netzwerke (NNs).

Maschinelles Lernen wird oft in drei Teilbereiche unterteilt, “supervised Learning” (überwachtes Lernen), “unsupervised Learning” (unüberwachtes Lernen) und “Reinforement Learning” (bestärkendes Lernen). Jedes dieser Felder wird genutzt um verschiedene Probleme zu lösen. Dabei können die verwendeten Architekturen oder Methode sich allerdings überlappen. Beispielsweise können NNs in jeder dieser Kategorien verwendet werden (jedoch nicht genau auf die selbe Art).

In den nächsten Absätzen werden wir kurz auf jede der drei oben erwähnten Kategorien eingehen.

Wir werden Unterschiede und Ähnlichkeiten hervorheben und die gängisten Anwendungen skizzieren.

## 2.4 Supervised Learning (SL)

Die grundlegende Idee von SL ist die Ableitung einer Funktion aus einem Satz vorgegebener Trainingsdaten (Input-Output-Paare). "Betreut"beschreibt in diesem Kontext die Tatsache, dass die richtigen Antworten für die Trainingsinputs bekannt sein müssen, um diese Technik anwenden zu können. Wir werden Trainingsdaten, für die das korrekte Ergebnis bekannt ist in Zukunft als annotierte Trainingsdaten bezeichnen. SL wird oft für Aufgaben wie Bild-, Audio- oder Videoerkennung verwendet kann aber auch in vielen anderen Bereichen eingesetzt werden.

Um gewünschte Eigenschaften des betreuten Lernens spezifizieren zu können , formalisieren wir einen SL-Ansatz wie folgt (folgende Formalisierung ist angelehnt an [7]):

Wir definieren $X$ als Eingabebereich (Feature Space) und $Y$ als Ausgabebereich. Ein SL-Algorithmus ist eine Funktion $A$, die bei Eingabe einer Reihe von Trainingsdaten eine Funktion ausgibt: $A : (X \times Y)^m \rightarrow (X \rightarrow Y)$. Beachten Sie, wie wir weder spezifizieren, wie das Training an sich funktioniert, noch einen spezifischen Ansatz wie NNs oder HMMs annehmen. Hier wird ausschließlich formalisiert, dass es Trainingsdaten gibt (die annotiert sind) und daraus eine deterministische Funktion generiert wird, die Eingaben aus dem Eingabebereich in Ausgaben aus dem Ausgabebereich umwandelt.

Die Herausforderung des SL ist es eine Funktion $f$ zu finden, die ein beabsichtigtes Verhalten beschreibt. Um dies zu formalisieren, führen wir ein Mittel zur Messung von Ähnlichkeit zwischen Elementen von $Y$ ein (dies kann oft eine Norm sein): $D : (Y \times Y) \rightarrow \mathbb{R}$ und ein Orakel, das für jede gegebene Eingabe die "richtige" Antwort zurückgibt: $o : X \rightarrow Y$. Angesichts solcher Funktionen kann das SL-Problem als folgendes Optimierungsproblem formalisiert werden: $min \sum_{x \in X} D(f(x), o(x))$

Als Beispiel betrachten wir eine Anwendung, die Bilder mit Katzen erkennen soll. Bei einem Bild wird ein boolescher Wert ausgegeben, der angibt, ob das betrachtet Bild eine Katze enthält oder nicht. Für ein 8-Bit-Graustufen-Bild mit den Abmessungen 200 x 200 wäre der Eingabebereich $X = \mathbb{N}_{<256}^{40000}$ und der Ausgabebereich $Y = \{0, 1\}$. In dieser Situation könnte die Abstandsfunktion als $D(x) = 1 - o(x)$ definiert werden. Sie wird also einfach auf 1 gesetzt, wenn die Klassifizierung richtig ist, und auf 0, wenn sie nicht stimmt. Das Orakel ist nicht so einfach zu formalisieren. Das Problem gewünschte Eigenschaften nicht ohne Weiteres formalisieren zu können, wird in späteren Abschnitten noch weiter erörtert. Für den Moment wollen genügt es zu sagen, dass es eine Art Orakel geben muss (in diesem Fall wahrscheinlich ein Mensch), der die Trainingsdaten beschriftet hat.

## 2.5 Unbeaufsichtigtes Lernen

Unbeaufsichtigtes Lernen (UL) ist ähnlich wie SL, erfordert allerdings keine annotierten Trainingsdaten. Es wird oft für Aufgaben wie Clustering oder Schätzung von statischen Verteilungen verwendet. In diesem Dokument werden wir nicht auf diesen Teilbereich nicht weiter eingehen und uns im Weiteren auf SL und RL konzentrieren.

## 2.6 Bestärkendes Lernen

Bestärkendes Lernen (engl. Reinforcement Learning (RL)) ist die Disziplin einem Agent die Lösung einer Aufgabe in unbekannter Umgebung beizubringen. RL hat gegenüber SL den Vorteil, dass es keine explizit annotierten Trainingsdaten benötigt. In den letzten Jahre konnte man ein wachsendes Interesse an RL beobachten, da bemerkenswerte Durchbrüche dieses Gebiet erzielt wurden. Als Beispiel konnte ein Team von Google einen künstlichen Go-Spieler trainieren, der rein aus Spielen gegen sich

selbst lernte und den stärksten menschlichen Spieler der Welt (Go-Weltmeister) schlagen konnte (eine Errungenschaft, von der viele dachten, dass sie zu diesem Zeitpunkt noch lange nicht möglich sei) [2].

Um über Eigenschaften solcher Algorithmen sprechen zu können, formalisieren wir das RL-Setting. Die folgenden Definitionen sind angelehnt an [8]: Das RL-Setting kann als ein Vier-Tupel $(S, s_0, A, T, R)$ beschrieben werden, wobei $S$ ein Satz von Zuständen ist, $s_0$ ein Anfangszustand ist, $A$ eine Menge von Aktionen ist (die der Agent ausführen kann) T eine Übergangsrelation ist $T : S \times A \to S$, die beschreibt, wie der Zustand geändert wird, wenn eine bestimmte Aktion in einem bestimmten Zustand durchgeführt wird, und $R$ ist eine Belohnungsfunktion die Belohnungen/Strafen für jede Handlung des Agenten definiert $R : S \times A \to R$.

Diese Formalisierung beschreibt die deterministische Variante des RL-Settings. Es ist leicht möglich die Übergangsrelation durch eine Wahrscheinlichkeitsverteilung $P$ zu ersetzen, die probabilistische statt deterministische Übergänge beschreibt ($P : S \times A \times S \to [0, 1] \in Rs.t. \forall s_1 \in S : \forall a \in A : (\sum_{s_2 \in S} P(s_1, a, s_2)) = 1$).

Wir nennen $\Pi : S \to A$ eine Strategie. Damit können wir das RL-Problem als das Finden einer Strategie, die die erhaltene Belohnung maximiert: $\max_{\pi \in \Pi} \sum_{t=0}^{\infty} \gamma^t r_t$ (dabei ist $r_t$ ist die Belohnung zum Zeitpunkt t und $\gamma \in [0, 1]$ ist ein Abschwächungsfaktor).

Als Beispiel betrachten wir einen Agenten, der zu einem bestimmten Ziel in einer Rasterwelt laufen soll. Der Einfachheit halber betrachten wir eine feste Weltgröße von 5x5. Eine mögliche Modellierung dieses Szenarios ist wie folgt: Der Zustandsraum kann als zwei ganze Zahlen modelliert werden, wobei die erste die Position des Agenten und die zweite die Position der Zielposition angibt: $S = \mathbb{N}^2_{<25}$. Der Startzustand ist in diesem Fall einfach die Startposition des Agenten und die gewünschte Zielposition. Der Agent ist in der Lage, sich in 4 Richtungen zu bewegen (könnte auch anders definiert werden), so dass die Menge der Aktionen $A = \mathbb{N}_{<4}$ ist. Die Übergangsfunktion hat per Definition die Form $T : S \times A \to S$, also in diesem Beispiel $T : \mathbb{N}^2_{<25} \times \mathbb{N}_{<4} \to \mathbb{N}^2_{<25}$. Diese Relation würde kodieren, dass z.B. ein Schritt nach rechts die Agentenposition um eins erhöhen würde (wieder nur eine Möglichkeit, dies zu modellieren) und parallel einen Schritt nach links die Agentenposition um eins verringern würde (zusätzlich wäre es ist notwendig, SZeilenumbrüche"korrekt zu behandeln). Aufgrund von Platzbeschränkungen werden wir hier nicht auf weitere Einzelheiten eingehen. Zu guter Letzt müsste die Belohnungsfunktion definiert werden: Eine Möglichkeit wäre eine Belohnung von 1 zu verteilen, wenn Ziel tatsächlich erreicht ist (die Agentenposition ist also gleich der Zielposition) und ansonsten 0:

$$R(agent, target) = \begin{cases} 1 & agent = target \\ 0 & agent \neq target \end{cases}$$

Beachten Sie, dass es für den Erfolg einer RL-Anwendung entscheidend ist, wie das Problem modelliert wird. Insbesondere die Gestaltung der Belohnungsfunktion kann sehr herausfordern sein ist aber entscheidend, um die erwarteten Ergebnisse zu erzielen. Für eine ausführlichere Diskussion über die Herausforderungen und mögliche Fallstricke für Belohnungsfunktionen siehe Abschnitt 4.

# 3 Herausforderungen und Schwierigkeiten

In diesem Abschnitt werden wir einige Herausforderungen für Spezifikationen von KI diskutieren. Der Schwerpunkt wird hauptsächlich auf ML-Ansätzen liegen, da diese die interessantesten Fälle für Spezifikationen darstellen.

## 3.1 Paradox der fehlenden Spezifikationen

ML-Systeme werden meist in Situationen eingesetzt, in denen "traditionelleSSoftware nicht in der Lage ist, die zu bewältigende Aufgabe zu lösen. Dies führt in zweierlei Hinsicht zu einer unvorteilhaften Situation: Erstens, wenn die Situation nicht durch "traditionelle"Programmiertechniken lösbar ist, bedeutet das bereits, dass diese Aufgabe wahrscheinlich schwierig zu handhaben ist. Zweitens, da ein traditionelles Programm "klareßchrittweise Anweisungen hat, wie es eine gegebene Aufgabe löst, ist ein Programmierer, der in der Lage ist, ein Programm zu schreiben, oft auch in der Lage, es zu spezifizieren. Diese Eigenschaft ist bei ML-Ansätzen nicht vorhanden, da das System nicht programmiert, sondern gelernt wird. Eine weitere Perspektive darauf ist, dass ein Entwickler, wenn es eine vollfunktionale formale Spezifikation eines Programms gibt, diese höchstwahrscheinlich implementieren kann (oder es ist sogar möglich das Programm vollständig automatisch zu synthetisieren). Wir stellen also fest, dass die Spezifizierung von ML-Systemen insofern paradox ist, als dass sie genau in den Situationen angewendet wird, in denen es keine Spezifikation gibt oder es ist zumindest sehr schwer ist, sie zu finden.

## 3.2 Modularisierung von Systemen

Viele klassische Verifikationstechniken bauen auf der Tatsache auf, dass ein Programm aus Bausteinen unterschiedlicher Größe besteht und Verifikation auf jeder Detailstufe möglich ist. Es ist möglich Eigenschaften für ganze Programme, Module, Klassen, Funktionen/Methoden, sogar für einzelne Schleifen oder sogar Anweisungen zu beschreiben . Diese Modularität ist für die Verifikation auf zwei Arten unerlässlich: Erstens ist es möglich das System in unterschiedlicher Granularität zu verifizieren und sich auf bestimmte Teile (z.B. sicherheitskritische) zu konzentrieren. Zweitens ist es wichtig, da die Spezifikation hilfreich sein kann, wie Beckert et al. in [9]. In diesem Paper wird dargelegt, dass einige Teile der Spezifikation nicht notwendig sind, um das gewünschte Verhalten zu beschreiben, sondern um einem bestimmten Verifikationswerkzeug zu ermöglichen, den Beweis für die Korrektheit des Programms zu finden. Diese Modularität ist für ML-Ansätze nicht gegeben, da es nur sehr begrenzte Möglichkeiten gibt, ihre Zwischen- und Innenzustände zu untersuchen und Aussagen über diese zumachen. NN verhalten sich im Gegenteil eher wie Blackboxen, bei denen das einzige beobachtbare Verhalten das Ergebnis ist. Aus den genannten Gründen ist dieser Mangel an Modularität eine Herausforderung für die Spezifizierung dieser Art von Systemen.

## 3.3 Sich ändernde Systeme

Einige ML-Systeme lernen noch, während sie bereits im Einsatz sind (Online-Ansätze). Diese Systeme nehmen neuen Input auf und aktualisieren ihr Verhalten entsprechend. Ein Beispiel für ein solches System wäre ein Empfehlungssystem in einem Online-Shop. Dieses System aktualisiert seine Empfehlung für jeden Benutzer auf der Grundlage seiner zuvor gekauften Artikel. Dieses Verhalten führt dazu, dass sich das Ergebnis des Systems bei der gleichen Eingabe zu verschiedene Zeitpunkten unterscheiden kann. Ein sich so entwickelndes System kann eine weitere Herausforderung für die Spezifikation und


Verifikation darstellen, da es erfordert, dass die spezifizierten Eigenschaften noch generischer sind, um die Möglichkeit zu erfassen, dass das System sein Verhalten später noch ändert.

# 4 Annahmen

Wenn man versucht, ein System zu spezifizieren und später zu verifizieren, ist es entscheidend, die Annahmen zu verstehen, die während der Entwicklung gemacht wurden. Solche Annahmen können explizit oder implizit sein. Durch explizite Annahmen, meinen wir, dass sie Entwurfsentscheidung, bewusst von einem Architekten/Entwickler getroffen wurde (in diesem Fall ist diese Annahme evtl. sogar irgendwo dokumentiert). Wir bezeichnen als implizite Annahmen diejenigen, die indirekt durch die Entscheidung ein bestimmte Technik/Werkzeug/Ansatz zu nutzen getroffen werden. Implizite Annahmen können in verschiedenen Szenarien leicht übersehen werden. Um dieses Problem anzugehen, versuchen wir, einige der häufigsten Annahmen aufzulisten, die bei der Verwendung von ML-Ansätze gemacht werden, um das Bewusstsein für sie zu schärfen. Diese Liste ist keineswegs als vollständig zu verstehen, sondern dient vielmehr als Ausgangspunkt. Die hier vorgestellten Annahmen sind von der Arbeit in [10] inspiriert.

## 4.1 Annahmen über Daten

Die vielleicht am meisten unterschätzten Annahmen für ML-Ansätze sind die Annahmen, die über Trainingsdaten gemacht werden. Dies ist wohl nicht allzu überraschend, da es sich um Annahmen handelt, die in traditionelle Software nicht vorkommen. Herkömmliche Software ist unabhängig von ihren Daten, da die Logik programmiert und nicht gelernt wird. Bei ML-Ansätzen kann jedoch eine kleine Änderung im verwendeten Trainingsset signifikante Auswirkungen auf das Ergebnis haben. Daher sind Annahmen, die implizit für die Trainingsdaten getroffen wurden sehr wichtig und man sollte sie im Auge behalten.

Die grundlegendste Annahme bei den Daten ist, dass die Trainingsdaten unabhängig und identisch verteilt sind (IID, independently identically distributed). Die Trainingsdaten gelten als identisch verteilt, wenn sie aus der selben Verteilung erzeugt/gezogen. Unabhängigkeit ist in diesem Fall die statistische Unabhängigkeit, die besagt, dass die Wahrscheinlichkeit, einen bestimmten Trainingsinput zu ziehen keinen Einfluss auf die Wahrscheinlichkeit, dass eine andere Trainingsinput gezogen wird hat. Wie in [10] angegeben ist "dies ist jedoch immer bis zu einem gewissen Grad ungültig". Eine verwandte, vielleicht sogar noch grundlegendere Annahme ist, dass die Verteilung, aus der die Trainingsdaten stammen, die gleiche ist wie die, die man in der Realität vorfindet. Tatsächlich kann das Ignorieren der Kombination dieser beiden Annahmen als die indirekte Ursache für mehrere ML-Unfälle gelten, da die zur Verfügung gestellten Trainingsdaten nicht die angetroffene Situation in der Realität widerspiegelten (nicht identisch verteilt waren). Obwohl sie "bis zu einem gewissen Grad verletzt"werden, müssen sie im Allgemeinen angenommen werden. Trotzdem ist es wichtig, dass diese Annahmen bei der Konzeption, dem Training oder der Spezifikation eines ML-Systems nicht außer Acht gelassen werden.

## 4.2 Annahmen über Algorithmen

Die Anwendung eines ML-Ansatzes ist implizit eine Annahme an sich: Das erwartete Verhalten ist mit den richtigen Trainingsdaten erlernbar. Darüber hinaus werden bei konkreten Algorithmen oft noch zusätzliche Annahmen getroffen. In SL wird eine Metrik minimiert, so dass angenommen wird, dass diese Metrik die tatsächliche "Ähnlichkeitßwischen zwei Punkten widerspiegelt. Dementsprechend ist in der RL die Gestaltung einer angemessenen Belohnungsfunktion eine große Herausforderung. Durch die Wahl einer konkreten Funktion, wird die Annahme getroffen, dass diese Funktion eine passende ist. Diese Annahme ist eine Diskussion an sich wert. Es gibt eine Reihe von Beispielen, bei denen eine fehlerhafte Belohnungsfunktion zu einigen witzigen bist schlicht schlechten Ergebnissen führte [11]. Für eine ausführlichere Diskussion über dieses Thema/Annahme wird der interessierte Leser an [12]

verwiesen. Als ein Beispiel betrachten wir einen Reinigungsroboter: Die Belohnungsfunktion, die verwendet wird, ist die Menge des gesammelten Schmutzes. Eine Möglichkeit zur Maximierung wäre es Schmutz sammeln und dann wieder auf den Boden zu schütten, nur um ihn direkt wieder einzusammeln. Dabei ist die Menge des eingesammelten Schmutzes sehr hoch, aber die eigentliche Absicht des Entwicklers wird nicht erfasst. Dies ist ein sehr einfaches Beispiel, aber es ist nicht schwer, sich vorzustellen, dass eine angemessenen Belohnungsfunktion zu finden bei komplexeren Szenarien immer schwieriger wird.

## 4.3 Annahmen über die Umwelt

Dieser Punkt ist nicht spezifisch für ML/KI-basierte Systeme, aber dennoch wichtig. Wenn man versucht, ein System zu modellieren und in diesem Modell zu spezifizieren, dann sind die Annahmen, die in diesem Modell gemacht werden entscheidend für die Zusicherungen, die gemacht werden können (bzw. ihre Gültigkeit). Viele Modelle gehen z.B. davon aus, dass die zugrundeliegende Hardware fehlerfrei funktioniert, was offensichtlich nie mit 100%-tiger Sicherheit garantiert werden kann. Trotzdem ist das etablieren vergleichbarer Annahmen jedoch notwendig, um die Komplexität der Formalisierung beherrschbar zu halten.

## 4.4 Annahme allgemeiner Eigenschaften

Etwas anders sieht es aus, sobald man Annahmen über den Output eines trainierten Systems treffen will. Die meisten natürlich erscheinenden Annahmen, sind im Allgemeinen nicht ohne weiteres gültig. Tatsächlich werden wir sie als sehr allgemeine Eigenschaften von Systemen später in diesem Dokument auflisten (siehe 6). Typische Annahmen dieser Art sind Robustheit oder Monotonität. Annahmen dieser Art fälschlicher Weise zu machen, kann zu unerwartetem Verhalten führen.

# 5 Zuverlässigkeitsklassen

Bevor wir im nächsten Abschnitt auf Einzelheiten zu den Methoden der Spezifizierung für ML-basierte Systeme eingehen, ist es sinnvoll, sich einen Überblick auf hoher Ebene darüber zu verschaffen, welche Zusicherungen möglich sind. Dazu führen wir 4 Klassen der Zuverlässigkeit ein, in die sich Systeme einordnen lassen. Jede dieser Klassen erlaubt es, verschiedene Zuverlässigkeitsaussagen zu machen.

Die vier Zuverlässigkeitsklassen, die im Folgenden vorgestellt werden, sind "verifizierbar", "vollständig überwachbar", "teilweise überwachbaründ "nicht überwachbar". Beachten Sie, dass sich der Begriff der Überwachbarkeit (Monitorability) in unserem Kontext von dem unterscheidet, was in der Literatur der formalen Methoden meist unter diesem Begriff verstanden wird.

Bevor wir in die Definitionen unserer Klassen eintauchen, möchten wir betonen, dass diese Klassen subjektiv sind, dass verschiedene Personen/Organisationen dasselbe System unterschiedlich einstufen können. Das ist beabsichtigt, und wir werden darauf eingehen warum das so ist. Diese Klassen sollen für die Analyse zur Designzeit nützlich sein. Die Klassifikation eines Systems ist also als eine Entscheidung zur Designzeit zu sehen, die von einem Systemarchitekten getroffen wird. Wir werden veranschaulichen, wie eine solche Entscheidung zur Designzeit wiederum die Zusicherungen beeinflusst, die über das System gemacht werden können.

Wir definieren alle Klassen mit Bezug auf eine gegebenen Systemeigenschaft, die wir mit $\varphi$ bezeichnen. Ein System wird durch die Menge aller seiner Spuren beschrieben (eine Spur ist eine Folge von Systemzuständen). Eine Eigenschaft eines Systems kann somit als eine Menge von Spurenmengen dieses Systems beschrieben werden. In diesem Dokument gehen wir davon aus, dass $\varphi$ eine solche Eigenschaft ist. Beachten Sie zwei Dinge: Erstens ist es nicht unbedingt möglich, eine Formalisierung für eine solche Eigenschaft zu geben, und zweitens gibt es Eigenschaften, die nicht auf diese Weise beschrieben werden können, z.B. "Dies ist ein schönes System". Informell bedeutet dies, dass wir davon ausgehen, dass $\varphi$ objektiv beobachtbar ist.

**Definition 1** *Ein System ist verifizierbar, wenn mit vertretbarem Aufwand nachgewiesen werden kann, dass es $\varphi$ (immer) befriedigt.*

Wenn wir uns im Folgenden nicht auf eine bestimmte Eigenschaft $\varphi$ beziehen, nennen wir ein System verifizierbar, wenn es seine volle funktionale Spezifikation erfüllt. Zusätzlich spezifizieren wir absichtlich nicht weiter, wann der Verifikationsaufwand “vertretbar” ist, da dies stark vom Kontext abhängt, in dem ein System entwickelt wird. Betrachten Sie zu diesem Zweck die folgende Liste von Faktoren:

**1) Erfahrung** Ein in der Programmverifikation erfahrener Entwickler wird höchstwahrscheinlich in der Lage sein, die gewünschte Eigenschaft viel schneller nachzuweisen als jemand, der völlig neu auf dem Gebiet ist.

**2) Zeit** Da die Programmverifikation sehr zeitaufwendig sein kann, ist eine Verifikation aus Zeitgründen möglicherweise nicht möglich.

**3) Berechnungsleistung** Mehrere Methoden zur Programmverifikation beruhen auf einer Art von Solvern, die sehr rechenintensiv sein können.

**4) Typ des Systems** Einige Arten von Software, wie z.B. Treiber, sind leichter zu verifizieren als komplexe, wie z.B. neuronale Netze.

**5) Anwendungsbereich** Je nach dem Bereich, in dem das System eingesetzt wird, können unterschiedliche Sicherheitsanforderungen gelten (Normen, Standards, ...).

Die obige Liste ist keineswegs vollständig. Sie dient lediglich dazu zu zeigen, dass "vertretbar"von vielen Aspekten abhängt, die sich unter Umständen sogar gegenseitig beeinflussen können. So kann ein Treiber für ein kleines Gerät nach unserer Definition "verifizierbarßein, da es sich um ein eingebettetes System mit überschaubarer Komplexität handelt, das in einer Umgebung entwickelt wird, in der der Produktzyklus vergleichsweise langsam ist. Im Gegensatz dazu ist eine Gaming-App auf Grund der gegenteiligen Argumente höchstwahrscheinlich nicht verifizierbar. Man beachte jedoch, dass diese Faktoren trotz der schwer fassbaren “Vertretbarkeit” nur den Aufwand für die Verifizierung eines Systems beeinflussen. Das Ergebnis, nämlich dass das System tatsächlich verifiziert wird, bleibt dasselbe, sobald ein Beweis gefunden wird. Eine andere Perspektive ist, "verifizierbaräls eine parametrisierte Klasse zu betrachten, bei der die Höhe des Aufwands der Parameter ist, der in Abhängigkeit von den oben genannten Faktoren variieren kann.

Motiviert durch die Erkenntnis, dass nicht jedes System verifizierbar ist, definieren wir eine zweite Klasse der Zuverlässigkeit:

**Definition 2** *Ein System ist dann voll überwachbar, wenn es möglich ist, ein Entscheidungsverfahren zu implementieren, das in angemessener Zeit zur Laufzeit entscheidet, ob das aktuelle Verhalten der Komponente $\varphi$ erfüllt oder nicht.*

Intuitiv erfordert dies einen Monitor, der jederzeit in der Lage ist, zu erkennen, ob das System noch korrekt funktioniert oder nicht. Dies unterscheidet sich von der ersten Klasse in zwei wesentlichen Punkten: Erstens erlauben wir dem System die betrachtete Eigenschaft zu verletzen (aber wir müssen in der Lage sein, das zu erkennen), und zweitens spielt die Geschwindigkeit des Entscheidungsverfahrens eine entscheidende Rolle. Auch hier verzichten wir absichtlich auf eine weitere Präzisierung von “vernünftige Zeit”. Dies liegt daran, dass die Definition von “vernünftig”, wie die Definition von "vertretbar", stark vom Kontext der Anwendung abhängt. In einigen Fällen könnte es akzeptabel sein, dass folgende Aussage für ein System ausreichend ist; “Vor einer Stunde ist etwas schiefgegangen”. Oftmals ist es jedoch entscheidend, Fehler relativ schnell zu erkennen, um schnell auf das Problem reagieren zu können. Eine angemessene Zeit hängt also von der Art der Anwendung, aber auch von der Hardware ab, auf der der Monitor läuft. Bei vielen Systemen ist die Konstruktion eines Monitors eher möglich als eine vollständige Verifikation. Es gibt jedoch Systeme, bei denen selbst die Konstruktion eines Monitors für alle möglichen Systemzustände sehr schwierig ist. (z.B. neuronale Netze zur Bildklassifizierung). Unsere nächste Klasse der Zuverlässigkeit versucht, diese Idee zu erfassen.

**Definition 3** *Ein System ist teilweise überwachbar, wenn es möglich ist, ein Entscheidungsverfahren zu implementieren, das in angemessener Zeit zur Laufzeit entscheidet, ob das aktuelle Verhalten der Komponente $\varphi$ erfüllt oder nicht oder zu einem uneindeutigen Urteil führt.*

Hier lockern wir die Anforderungen an den Monitor, indem wir ihm erlauben, ein uneindeutige Urteile zu fällen. Auch wenn wir beschlossen haben keinen konkreten Prozentsatz zu verlangen, für den der Monitor zu einem eindeutigen Ergebnis kommen muss, kann es in der Praxis vernünftig sein, dies zu tun. Beachten Sie, dass das Nichtzustandekommen eines Urteils entweder auf die Art der Eigenschaft selbst oder auf den Monitor zurückzuführen sein kann. Auch wenn ein eindeutiges Urteil nicht gefunden wurde, kann dies durch ein Timeout verursacht worden sein oder auch nicht. Dies bedeutet auch, dass ein Monitor in der Lage sein kann, ein abschließendes Urteil für alle Systemzustände zu fällen, allerdings nicht rechtzeitig für alle Eingaben.

Der Vollständigkeit halber führen wir eine letzte Klasse von Zuverlässigkeit ein, die wir "nicht überwachbar"nennen:

**Definition 4** *Ein System ist dann nicht überwachbar, wenn es weder teilweise überwachbar noch verifizierbar ist.*

Wenn ein System nicht überwachbar ist, ist das schlimmste Fall. In diesem Fall können wir keine Zusicherungen für das System geben (weder zur Designzeit noch zur Laufzeit). Dennoch ist dies vielleicht kein verlorener Fall, da das Testen immer noch eine Option sein kann. Wenn es möglich ist, das System rigoros zu testen, um ein Qualitätskriterium zu gewährleisten, kann das je nach Kontext ausreichen.

Die Reihenfolge, in der wir die Klassen vorgestellt haben, stellt eine natürliche Ordnung dar, was die Stärke der erreichbaren Zusicherungen betrifft. Während eine überprüfbare Komponente “perfektes” Verhalten aufweist, ist ein vollständig überwachbares System zumindest in der Lage zu erkennen, ob es einen Fehler gemacht hat oder nicht. Diese Fähigkeit ist bei teilweise überwachbaren Systemen begrenzt und fehlt bei nicht überwachbaren Systemen völlig.

Ein System kann verifizierbar, aber nicht überwachbar sein. Stellen sich ein System vor, bei dem ein enormer Aufwand gerechtfertigt ist, um sicherzustellen, dass es wie erwartet läuft, das aber sehr anspruchsvolle Echtzeitanforderungen stellt. In diesem Fall wäre das System zwar verifizierbar, aber nicht überwachbar.

Man beachte außerdem, dass die tatsächliche Zuverlässigkeit eines Systems und die Zuverlässigkeitsgewährleistungen, die in der Praxis gemacht werden können, unterschiedlich sein können. Ein System kann vollkommen zuverlässig sein, aber es kann sehr schwer sein, dies zu beweisen. Unsere Klassen verhindern jedoch den umgekehrten Fall: Wenn ein System überprüfbar ist, können starke Zuverlässigkeitszusicherungen gemacht werden.

# 6 Eigenschaften von ML-Systemen

In diesem Abschnitt stellen wir einige Eigenschaften vor, die in der Literatur untersucht wurden, oder unserer Meinung nach interessant für den ML-Kontext sind. Wir werden konkrete Empfehlungen geben, wie diese Eigenschaften zu spezifizieren sind. Unsere Empfehlungen zielen darauf ab eine einfache Spezifizierung auch für Nicht-Experten-Benutzer zu ermöglichen und bleiben dennoch ausreichend ausdrucksstark. Dies ist eine bekannte Herausforderung, weshalb wir zusätzlich ein Kapitel für nutzerfreundliche Spezifikation zur Verfügung stellen (siehe 7).

Die Spezifizierung von ML-Systemen kann auf mehreren verschiedenen Ebenen und mit unterschiedlichen Granularität erfolgen. Wir werden verschiedene Ansätze vorstellen und versuchen, im Folgenden auf Unterschiede und Ähnlichkeiten zwischen ihnen hinzuweisen. Die Suche nach einem geeigneten Ansatz für ein System wird allerdings weiterhin eine Einzelfallentscheidung bleiben.

## 6.1 Allgemeine Eigenschaften

In diesem Abschnitt stellen wir einige Eigenschaften vor, die nicht spezifisch für ML-Systeme sind, aber möglicherweise eine besondere Bedeutung für sie haben. Wie in einem früheren Abschnitt erwähnt, werden diese Eigenschaften oft fälschlicherweise standardmäßig als gegeben angenommen. Wie wir im Folgenden erörtern werden, ist dies jedoch im Allgemeinen nicht zutreffend. Wir versuchen, eine normalisierte Art und Weise der Formalisierung dieser Eigenschaften vorzustellen, so dass sie für eine möglichst breite Menge an Systemen einsetzbar ist.

## 6.2 Robustheit

Robustheit ist derzeit wahrscheinlich die am besten erforschte und wohl auch realistischste Eigenschaft für NNs. Eine Formalisierung der Robustheit angelehnt an [10] ist wie folgt: Ein System implementiert die Funktion: $f : X \Rightarrow Y$. Bei Eingaben aus dem Eingaberaum $X$ gibt dieses System Werte aus dem Ausgaberaum $Y$ zurück. Bei zwei Prädikaten $p_1$ und $p_2$, die "Ähnlichkeitäuf $X$ bzw. $Y$ definieren ($p_2 : X \times X \Rightarrow \mathbb{B}$ und $p_2 : Y \times Y \rightarrow \mathbb{B}$), ist ein System robust gdw. $p_1(x_1, x_2) \Rightarrow p_2(f(x_1), f(x_2))$. Intuitiv heißt das, dass bei zwei ähnlichen Eingaben das System ähnliche Ausgaben ausgeben wird. Diese Eigenschaft kann für alle Systeme (unabhängig davon, ob sie auf ML basieren oder nicht) angegeben werden. Man beachte, dass ob ein System ist robust oder nicht von der konkreten Definition von Ähnlichkeit abhängt. Wie wir später sehen werden, kann sich dies als problematisch erweisen, da die Definition von Ähnlichkeit je nach Eingaberaum sehr herausfordernd sein kann (zumindest Ähnlichkeit nach menschlichen Standards). Für ML-Systeme hat diese Eigenschaft besondere Aufmerksamkeit erlangt da sich gezeigt hat, dass mehrere dem Stand der Technik entsprechende NNs diese Eigenschaft nicht haben [1]. In diesem Fall wird die Robustheit nur für Trainingsdaten untersucht wobei Ähnlichkeit als eine Norm auf dem Eingaberaum und als Gleichheit auf dem Ausgaberaum definiert wird. Intuitiv heißt das, dass alle Eingaben, die nah genug (bzgl. einer gewissen Norm) an einem Trainingsdatum liegen, genau so wie dieses Trainingsdatum klassifiziert werden. In der ML-Community, werden Gegenbeispiele zu dieser Eigenschaft als “kontradiktorische” Beispiele (adversarial Examples (AE)) bezeichnet. Szegedy et al. zeigten, dass es möglich ist, solche Fehlklassifikationen durch minimale Störungen im Eingabebereich zu provozieren. Dies wird vielleicht am besten klar am Beispiel von Bildklassifikationssoftware. Betrachten Sie hierzu das Beispiel aus Abb. 6.2. Links ist ein Bild eines Pandas dargestellt. Dann werden einige leichte Modifikationen an diesem Bild vorgenommen, die zu einem Bild führen, das für Menschen ununterscheidbar vom ersten ist. Die Klassifizierung des Netzwerkes ändert jedoch von “Panda” (mit 57,7% Konfidenz) zu “Gibbon” (mit 99,3% Konfidenz). Robustheit gegen solche AE ist wünschenswert. Wie erwähnt, hängt Robustheit in diesem Sinne jedoch stark von der Definition

der Ähnlichkeitsprädikate ab. Da AEs meist für NN betrachtet werden, ist eine vernünftige Wahl für $p_2$ (die Ausgabe-Ähnlichkeit) Gleichheit, da diese besagt, dass sich die Klassifikation bei ähnlichen Eingaben nicht ändert. Obwohl andere Ähnlichkeitsprädikate natürlich denkbar wären, werden wir diese aus Platzgründen in diesem Dokument nicht weiter besprechen. Wir konzentrieren uns vielmehr auf die Definition der Ähnlichkeit auf dem Eingabefeld. Die häufigst Wahl für diese Prädikate sind Normen wie L1, L2 oder L-inf. Ist der Eingabebereich der Form $\mathbb{I}^n$, wobei $(\mathbb{I}, +)$ eine abelsche Gruppe ist und Elemente von $\mathbb{I}^n$ als $i = (i_1, i_2, \ldots, i_n)$ geschrieben werden, sind die Definitionen für diese Normen nachstehend aufgeführt:

1. L1-Norm: $d(i1, i2) = \sum_{j=0}^{n} |i1_j - i2_j| < t$
2. L2-Norm: $d(i1, i2) = \sqrt{\sum_{j=0}^{n} |i1_j - i2_j|^2} < t$
3. Linf-Norm: $d(i1, i2) = max_j |i1_j - i2_j| < t$

Beachten Sie, dass jede Definition auf einer zusätzlichen Variablen $t$ beruht. Wenn Sie also eine Norm als Ähnlichkeitsprädikat genutzt wird, muss ein Schwellwert angegeben werden, um zwischen ähnlichen und nicht-ähnlichen Werten zu unterscheiden.

Ohne auf weitere Einzelheiten einzugehen, möchten wir betonen, dass wir mit unserer Definitionen von Robustheit eine große Bandbreite unterschiedlicher Eigenschaften adressieren. Wir haben AE als das prominenteste Beispiel vorgestellt, das ist allerdings nur eine Möglichkeit, die Ähnlichkeitsprädikate zu definieren. Tatsächlich kann die Robustheit in dieser Hinsicht als eine Kategorie von Eigenschaften angesehen werden. Beispiele für andere Eigenschaften solcher Art sind beispielsweise semantische AEs und semantische Invarianz [13].

### Spezifikation

Die Formalisierung der Robustheit führt zu einem natürlich Spezifikationsansatz. Da die Robustheit eine ziemlich spezifische Eigenschaft ist, gibt es keine komplexe Spezifikationssprache, die für die erforderlich wäre. Wir schlagen folgende Schreibweise vor um Robustheit eines Systems auszudrücken:

$robust(s1, s2)$

Das ist die allgemeinste Form der Robustheit, wobei s1 und s2 die Prädikate für Ähnlichkeit auf Ein-/Ausgaberaum sind und Robustheit für jede mögliche Eingabe gefordert wird. Die Anforderung an die Robustheit für jede einzelnen Eingabe ist jedoch nicht immer sinnvoll oder sogar im Widerspruch zu bestimmten Definitionen von s1 und s2. Betrachten Sie das Beispiel der AEs wie oben beschrieben. Robustheit für jeden möglichen Input zu fordern wäre hier völlig sinnlos, da das äquivalent zur Begrenzung der Ausgabe auf eine einzige Möglichkeit wäre (wie durch eine einfaches induktives Argument eingesehen werden kann). Stattdessen ist es nur logisch, die Menge der Eingaben einzuschränken für die diese Robustheit gelten soll. Dies erfordert einen zusätzlichen Parameter, um die vorgestellte Robustheitsspezifikation einzuschränken:

$robust(s1, s2, x)$

Hier sind s1 und s2 gleich wie in der vorherigen Definition und x beschreibt eine Teilmenge von X. Es ist möglich, x als Prädikat auf X anzugeben, also eine implizite Definition einer Teilmenge von X oder explizit als Angabe einer Reihe von Eingaben, für die Robustheit muss gezeigt werden soll. Eine weitere mögliche Variante, die lediglich syntaktischer Zucker ist, wäre wie folgt:

$robust(s1, x)$

In diesem Fall wird angenommen, dass s2 die Gleichheit ist und daher ausgelassen werden kann.

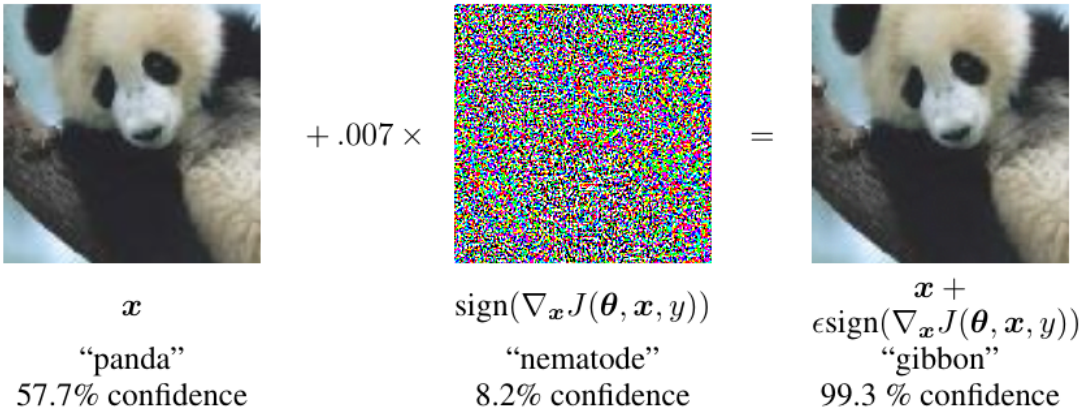


Abbildung 2: Ein kontradiktorisches Beispiel aus [1]

### Glattheit

Als Erweiterung des Begriffs der Robustheit ist eine mögliche Eigenschaft die Glattheit. Diese erfordert, dass die Funktion, die das System berechnet, unendlich oft differenzierbar ist. Diese Eigenschaft kann von großem Interesse sein, wenn die Interpolation zwischen Datenpunkten ein gewünschtes Merkmal ist. Genau wie bei Robustheit, kann diese Eigenschaft entweder für den gesamten Ausgaberaum oder für einen Teil davon gefordert werden. Wir halten die konvexe Hülle aller Trainingsinputs für eine interessante Wahl eines solchen Unterraums. Dies ist eine der Eigenschaften, die oft angenommen wird, aber nicht notwendigerweise gelten muss und somit explizit angegeben werden sollte. Ähnlich wie bei der Robustheit benötigt diese Eigenschaft keine komplexe Spezifikationssprache, sondern ist eher eine einfache Aussage.

### Monotonie

Monotonie kann wie folgt formalisiert werden: $\forall x_1, x_2 \in X : x_1 \geq x_2 \Rightarrow f(x_1) \geq f(x_2)$ wobei X der Eingabebereich des Systems ist und $\geq$ eine partielle Ordnung auf X bzw. Y ist. Beachten Sie hierbei, wie die Definition der Ordnungen entscheidend für die Semantik der Eigenschaft ist. Beachten Sie außerdem, dass mehrere Monotonieanforderungen für das gleiche System gestellt werden könnten. In diesem Fall könnte entweder der Schnitt von allen Anforderungen in Betracht gezogen werden, oder es wäre möglich, die verschiedene Anforderungen in eine einzige zu codieren, indem eine entsprechende Ordnung wählt.

Wie bei den letzten beiden Eigenschaften ist die Monotonie ein einfaches Prädikat, das für eine System gefordert werden kann. In diesem Fall wird es mit den beiden Ordnungen auf X und Y parametrisiert. Ähnliche Ansätze wie sie für die Robustheit präsentiert wurden, können für Monotonie gemacht werden.

### Andere allgemeine Eigenschaften

Da wir das Verhalten unseres Systems als Funktion beschreiben, sind alle Eigenschaften von Funktionen im mathematischen Sinne auch hier anwendbar. In den letzten drei Unterabschnitten haben wir die in unseren Augen relevantesten dieser Eigenschaften vorgestellt, diese Liste sollte allerdings nicht als

vollständig betrachtet werden. Je nach Anwendungskontext kann es ganz andere Eigenschaften dieser Kategorie geben, die von höherem Interesse sind. Mögliche andere Interessante Beispiele könnten Injektivität, Surjektivität oder Symmetrien sein, um nur einige zu nennen.

## 6.3 Eingangs-Ausgangs-Beziehung

Diese Eigenschaft kommt den traditionellen Spezifikationen des imperativen Programmierens wahrscheinlich am nächsten. Da ML-Systeme, genau wie traditionelle Programme, Ausgaben auf der Grundlage gegebenen Eingaben berechnet, ist es möglich traditionelle Ansätze mit Vor- und Nachbedingung auf sie anzuwenden. Formal kann dies wie folgt beschrieben werden: $\forall x \in X : \forall y \in Y : P(x) \Rightarrow Q(x, f(x))$ wobei X und Y der Ein- und Ausgaberaum sind und P und Q Prädikate auf $X$ bzw. $X \times Y$ ($P : X \Rightarrow \mathbb{B}$ und $Q : X \times Y \rightarrow \mathbb{B}$). Diese Art der Spezifikation erlaubt die Erfassung einer breiten Palette von Eigenschaften. Typische Beispiele für diese Art von Eigenschaften sind Intervallbedingungen der Form “Wenn die Eingabe im Bereich R1 liegt, dann ist die Ausgabe im Bereich R2”. Eine weitere mögliche Eigenschaft dieser Art ist eine Invariante auf der Ausgabe: “Die von diesem System generierte Ausgabe ist immer größer als Null und kleiner als 10”. Ein letztes Beispiel wäre die Reaktion auf boolesche Werte in der Eingabe: “Wenn der Eingang i1 auf True gesetzt ist, ist der Ausgang negativ und sonst positiv.” Alle diese Beispiele folgen der oben beschriebenen Form: wenn eine Vorbedingung gilt dann muss eine Nachbedingung gelten.

## 6.4 Trace-Eigenschaften

Die folgende Spezifizierungstechnik wird hauptsächlich für RL verwendet, kann aber auch für andere ML-Ansätze anwendbar sein. Wie in Abschnitt 2.6 beschrieben, ist Reinforcement Learning als ein System definiert das Zustände hat und Übergänge zwischen Zuständen durch Aktionen ausgelöst werden. Diese Art von System, passt perfekt für trace-basierte Spezifikationssprachen wie Linear Temporal Logic (LTL) oder Metric Temporal Logic (MTL). Wir erinnern uns, dass ein RL-System als ein Tupel $(\Sigma, s_{\text{Start}}, A, T, R)$ definiert war.

Um über Spezifikationen solcher Systeme zu sprechen, führen wir den Begriff der Spur ein. Eine Spur ist eine Folge von Zuständen: $s = (s_0, \ldots, s_n) \in \Sigma^n$. Gegeben ein RL-System P, können wir P nun durch alle seine Spuren definieren: $P = \{s = (s_0, \ldots, s_n) \mid \forall i \in (0, \ldots, n-1) : \exists\, a \in A : s_{i+1} = T(s_i, a) \wedge s_0 = s_{\text{Start}}\}$. Ein Programm ist also die Menge aller Spuren, die im Anfangszustand beginnen und dann durch Aktionen voranschreiten. Wir bezeichnen die Menge aller (möglicherweise unendlichen) Spuren über $\Sigma$ als $\Sigma^\infty$.

Bevor wir fortfahren, stellen wir einige Notationen vor, die wir verwenden werden: Wir nennen p ein Prädikat, wenn es eine Funktion ist, die jeden Zustand auf einen booleschen Wert abbildet: $p : \Sigma \rightarrow \mathbb{B}$. Wenn $s$ eine Spur ist ($s \in \Sigma^\infty$), bezeichnen wir das i-te Element von $s$ durch $s_i$ mit dem Anfangsindex 0 und das Suffix von $s$ beginnend bei Index i mit $[s]^i$. Außerdem bezeichnen wir die Konkatenation von zwei Spuren $s_1$ und $s_2$ (wobei $s_1$ endlich sein muss) durch $s_1 \circ s_2$.

Mit dieser Notation ist eine Spezifikation eine Menge von Spuren: $\varphi \subseteq \Sigma^\infty$. Ein Programm $P$ erfüllt eine Spezifikation $\varphi$ gdw. $P \subseteq \varphi$. Das heißt intuitiv, dass ein Programm eine Spezifikation erfüllt, wenn alle Läufe dieses Programms die angegebene Eigenschaft haben.

Es ist leicht zu erkennen, dass interessante Eigenschaften von Programmen kaum ohne weitere Hilfe in einer solchen Mengennotation beschreibbar sind. Deshalb wurden mehrere Sprachen entwickelt, die es erlauben, Eigenschaften als Spuren auszudrücken. Die wohl bekannteste ist Linear Temporal Logic (LTL) die in [14] eingeführt wurde. Wir stellen hier eine Schreibweise für LTL vor, wie sie in [15] beschrieben wurde, die sich leicht von der Schreibweise im Originalpapier unterscheidet. Gegeben eine Menge von Propositionen Prop die Syntax für LTL ist wie folgt definiert:

$$\psi \in \texttt{LTL} ::= tt \mid p \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid X\psi \mid \psi_1 U \psi_2$$

Wobei $tt$ die wahre Formel ist, $p \in$ Prop und $\psi, \psi_1, \psi_2 \in$ LTL. Die Semantik von LTL wird dann induktiv über der Struktur der LTL-Formeln wie folgt definiert:

$$\begin{aligned}
[\![tt]\!] &= \Sigma^\infty \\
[\![p]\!] &= \{s | p(s_0)\} \\
[\![\neg\varphi]\!] &= \Sigma^\infty \backslash [\![\varphi]\!] \\
[\![\varphi_1 \wedge \varphi_2]\!] &= [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![X\varphi]\!] &= \{s | [s]^1 \in [\![\varphi]\!]\} \\
[\![\varphi_1 U \varphi_2]\!] &= \{s \mid \exists j \; s.t. \; [s]^j \in [\![\varphi_2]\!] \text{ and } (i < j \text{ implies } [s]^i \in [\![\varphi_1]\!])\}
\end{aligned}$$

Wir möchten betonen, dass das traditionelle LTL für unendlich Spuren definiert ist. Es gibt jedoch mehrere Ansätze, die es erlauben, LTL auf endliche Spuren anzuwenden. Für eine ausführlichere Diskussion wird der interessierte Leser an [16, 17] verwiesen. Wie bereits erwähnt, ist die Semantik über Spurengruppen definiert. Die Menge von Spuren, auf der die wahre Formel ($tt$) gilt, ist die Menge aller Spuren $\Sigma^\infty$. Ähnlich ist die Menge von Spuren, für die ein Prädikat gilt, definiert als die Menge der Spuren, bei denen das Prädikat im Anfangszustand der Spur gilt. Das sind die beiden Basisfälle auf denen die anderen Definitionen aufbauen. Die eingeführten Operatoren können in logische Operatoren ($\wedge$ und $\neg$) und zeitliche Operatoren ($X$ und $U$) unterteilt werden. Die beiden erstgenannten sind standard boolesche Logikoperatoren. *not* invertiert der Wahrheitswert einer gegebenen Formel und *and* beschreibt die Konjunktion von zwei Formeln. Da diese beiden Operatoren eine logische Basis bilden, sind alle anderen standard Logikoperatoren (z.B. $\vee, \Rightarrow, \ldots$) mit diesen beiden Operatoren abbildbar. Die temporalen Operatoren werden *next* ($X$) und *until* ($U$) genannt. Intuitiv besagt *next*, dass die Formel im nächsten Zustand gilt, während *until* bedeutet, dass die erste Formel gilt, bis die zweite zum ersten Mal gilt. Parallel zu den logischen Operatoren sind diese beiden temporalen Operatoren eine Basis, durch die andere temporale Operatoren ausgedrückt werden können.

Um eine einfachere Spezifizierung zu ermöglichen, präsentieren wir auch eine erweiterte Version von LTL. Wir nennen diese erweiterte Version von LTL eLTL inspiriert durch [15]. Wie aus der folgenden Definition der Semantik hervorgeht, sind die Erweiterungen rein syntaktischer Natur und erhöhen nicht die Ausdruckskraft von LTL.

$$\psi \in \texttt{eLTL} ::= tt \mid ff \mid p \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid X\psi \mid \psi_1 U \psi_2 \mid E\psi_2 \mid G\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \Rightarrow \psi 2 \mid \psi_1 \Leftrightarrow \psi_2$$

Wie zu erwarten ist, ist LTL eine Teilmenge von eLTL. Deshalb bleibt die Semantik der Operatoren, die bereits Teil von LTL sind gleich. Die Semantik der neu eingeführten Operatoren ist wie folgt definiert:

$$
\begin{aligned}
[\![ff]\!] &= \emptyset \\
[\![\varphi_1 \vee \varphi_2]\!] &= [\![\varphi_1]\!] \cup [\![\varphi_2]\!] \\
[\![\varphi_1 \Rightarrow \varphi_2]\!] &= [\![\varphi_2 \vee \neg\varphi_1]\!] \\
[\![\varphi_1 \Leftrightarrow \varphi_2]\!] &= [\![(\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)]\!] \\
[\![E\varphi]\!] &= [\![tt\ U\ \varphi]\!] \\
[\![G\varphi]\!] &= [\![\neg(E\neg\varphi)]\!]
\end{aligned}
$$

Die hier vorgestellten booleschen Operatoren sind standard boolesche Operatoren und werden deshalb hier nicht weiter erklärt werden. Die beiden neuen temporalen Operatoren ($E$ und $G$) erfassen intuitiv die Vorstellungen von “etwas passiert irgendwann” ($E$ für “eventually”) und “Etwas ist immer wahr” (G für global). Diese beiden Begriffe sind sehr verbreitet in Spezifikationen. Mit $G$ kann man Invarianten angeben, z.B. der Abstand, der zum nächsten Hindernis eingehalten werden soll, liegt immer über einer gegebenen Schwelle: $G$ safeDistance. Ähnlich zu ist es mit $E$ möglich, typische Liveness-Eigenschaften anzugeben, z.B. wann immer eine Anfrage eingeht, wird sie irgendwann in der Zukunft bearbeitet: Request $\Rightarrow E$ grantRequest.

`LTL` hat eine Reihe von Erweiterungen/Varianten, die unterschiedliche Unzulänglichkeiten von `LTL` adressieren. Die wohl bekannteste ist `MTL`[16, 18], die Ereignisse mit Zeitstempeln einführt und somit Intervalle für die zeitlichen Operatoren erlaubt. Dies ermöglicht Spezifikationen wie z.B. "wenn eine Anfrage erhalten wird, wird sie innerhalb von 3 Zeiteinheiten bearbeitet". Zeiteinheiten werden normalerweise als abstrakte Einheit behandelt. Eine weitere ähnliche Spezifikationssprache ist `STL` [19], die oft benutzt wird, um quantitative Spezifikation/Verifizierung zu ermöglichen. Dies ermöglicht es Aussagen darüber “wie sehr eine Spur eine Eigenschaft verletzt” [20]. Wir werden hier nicht im Detail auf diese Sprachen eingehen. Es sei allerdings gesagt, dass es weitere Dialekte/Erweiterungen/Varianten gibt, die wir hier aus Platzgründen nicht vorstellen werden, die für konkrete Anwendungen sehr interessant sein können.

### Verlorene Spuren

Die oben dargestellte Spezifikationsmethode beruht auf der Idee, dass es "guteünd ”schlechte” Spuren gibt: eine binäre Entscheidung, ob eine Spur eine Eigenschaft erfüllt oder nicht. Betrachten allerdings folgende Situation: Ein Auto fährt gerade auf eine Wand mit 100 km/h zu. Die Sicherheitseigenschaft, die wir zu überwachen versuchen ist: "das Auto kommt nie näher an das nächsten Objekt als 2m". Das Problem hier ist, dass, sobald wir erkennen, dass wir zu nah an der Wand sind es wahrscheinlich schon zu spät ist, um eine Kollision zu vermeiden. Um dieser Beobachtung zu formalisieren, führen wir drei verschiedene Arten von Spuren mit Hinblick auf eine gegebene Eigenschaft $\varphi$ ein:

1. fatale Spuren $\varphi_f$: $s \notin \varphi$
2. Verlorene Spuren $\varphi_d$:

$$
\begin{aligned}
s = (s_0, ..., s_n) \in \varphi \wedge \\
\exists k \in N : \forall (a_1, \ldots, a_k) \in A^k : \\
s \circ (T(s_n, a_1), T(T(s_n, a_1), a_2), \ldots, T(...T(s_n, a_1)..., a_k)) \in \varphi_f
\end{aligned}
$$

3. Sichere Spuren $\varphi_s = \varphi \setminus \varphi_d$

Der Begriff der verlorenen Spuren versucht, die Intuition zu erfassen, dass es Situationen gibt, in denen ein System noch nicht in einem fatalen Zustand ist (z.B. abgestürzt), aber es auch keine Möglichkeit mehr gibt, einen solchen Zustand zu vermeiden. Formal wird dies durch die Tatsache ausgedrückt, dass eine verlorene Spur eine Spur ist, die derzeit nicht die Sicherheitseigenschaft verletzt, aber keine Fortsetzung hat, die nicht zu einer fatalen Spur wird. Für viele Systeme, die in der physischen Welt agieren, gilt, dass die Spezifikationen den fatalen Fall beschreibt. Um jedoch nie in fatale Situation zu gelangen, muss man in der Lage sein, verlorene Situationen zu erkennen, bevor man sie erreicht. Die Grenze zwischen sicheren und verlorenen Zuständen zu finden ist in vielen Fällen sehr anspruchsvoll, da es teilweise das Lösen komplexer Aufgaben wie das Lösen von Differentialgleichungen erfordert. Man beachte nochmals, dass dies ein Problem spezifisch für die Laufzeitverifikation ist und bei apriori Verifikation nicht auftritt.

## 6.5 Eigenschaften von Trainingsdaten

Wie bereits in Kapitel 4.1 erwähnt hängt das Ergebnis einer Trainingsphase stark von den verwendeten Trainingsdaten ab. Zusätzlich werden mehrere Annahmen implizite dadurch getroffen, dass überhaupt ein ML-Ansatz verwendet wird. Daher ist es sinnvoll, auch über Spezifikationen für die Trainingsdaten nachzudenken. Die im Abschnitt 4.1 getroffenen Annahmen sind zum Beispiel mögliche Spezifikationen. Oftmals sind diese Annahmen sehr ideeller Natur weshalb es sinnvoll sein kann als Spezifikation abgeschwächte Versionen zu nutzen.

Obwohl wir der Meinung sind, dass Spezifikationen für Trainingsdaten ein wichtiges Thema sind und in zukünftigen Arbeit berücksichtigt werden sollte, werden wir hier nicht weiter ins Detail gehen, da dies den Rahmen dieses Dokuments sprengen würde. Der interessierte Leser sein an dieser Stelle an entsprechende Literatur verwiesen. Ein guter Ausgangspunkt dafür ist das Buch von Koller et al. “Probabilistic graphical Models” [21]. Dort werden Standardannahmen probabilistischer Modelle und unter welchen Bedingungen diese Annahmen getroffen werden können erörtert. Ein konkreteres Beispiel für einen Ansatz, der dem Ansatz von Spezifikationen für Trainingsdaten folgt, ist in [22] beschrieben.

## 6.6 Eigenschaften kombinieren

Obwohl viele der in diesem Kapitel vorgestellten Eigenschaften recht unterschiedlich sind, kann es nützlich sein, sie in einer Systemspezifikation zu kombinieren. Zum Beispiel könnte es interessant sein, relationale Eigenschaften wie Robubstheit oder Monotonität mit funktionellen Eigentschaften, z.B. in LTL, zu kombinieren. Es ist unmöglich, alle möglichen Kombinationen der hier vorgestellten Eigenschaften zu diskutieren. Wir werden allerdings ein konkretes Beispiel dafür geben, wie eine interessante Kombination aussehen könnte.

Betrachten wir ein sehr einfaches Szenario für ein selbstfahrendes Auto: Das Auto fährt gerade auf ein Hindernis zu. Wie bereits zuvor beschrieben, ist eine nützliche Eigenschaft hier, dass das Auto immer einen Mindestabstand zu allen Hindernisse einhält. Zusätzlich könnte es interessant sein, zu verlangen, dass das System robust ist, da eine Situation, in der eine minimale Manipulation der Eingabe die Entscheidung des Systems drastisch verändert, höchstwahrscheinlich ein Scheidepunkt in der Entscheidungsfindung ist. Solche Scheidepunkte sind in den meisten Fällen interessant zu beobachten um gegebenenfalls reagieren zu können.

# 7 Nutzerfreundliche Spezifikation

In diesem Kapitel geben wir einen Überblick über Techniken, die es ermöglichen, Spezifikationen auf eine weniger technische Art und Weise zu schreiben Alle hier vorgestellten Methoden sind Möglichkeiten, um Eigenschaften ausdrücken, wie sie im letzten Kapitel besprochen wurden. Einige der folgenden Methoden sind agnostisch bezüglich der Eigenschaften, die damit spezifiziert werden können, während andere nur für bestimmte Eigenschaften anwendbar sein.

## 7.1 Domänenspezifische Sprachen

Domänenspezifische Sprachen (Domain Specific Language (DSL)) sind Sprachen, die speziell für eine bestimmte Domäne entwickelt wurden. In den letzten Jahren haben DSLs eine breite Palette von Anwendungen gefunden [23, 24, 10]. Sie können als Schnittstelle zwischen komplexen und technischen Backends und Benutzern interagieren die möglicherweise nicht den technischen Hintergrund haben, um diese zu nutzen. In diesem Dokument werden wir DSL hauptsächlich im Sinne von Spezifikationssprachen verwenden. Oftmals werden die DSLs als Zwischensprache verwendet, die später übersetzt/kompiliert wird (in eine allgemeinere Sprache). Der Hauptvorteil von DSLs ist die Möglichkeit, die Gestaltung der Sprache speziell auf die Bedürfnisse einer bestimmten Domäne zu zuschneiden und damit eine vergleichsweise einfache und bequeme Nutzung zu ermöglichen.

Im Kontext der ML-Domäne wären in den Augen der Autoren z.B. autonomes Fahren, Bild-/Video-Klassifizierung und Gameplay-Agenten geeignete Anwendungsfälle für DSLs. Die genannten Szenarien sind einige der am häufigsten untersuchten Bereiche für ML und haben vergleichsweise klare Settings, die eine einfache, aber dennoch vollständige Spezifikation interessanter Eigenschaften ermöglichen sollten.

## 7.2 Grafische Editierwerkzeuge

Menschen neigen dazu, Informationen leichter zu verstehen, wenn sie in einer graphische Art und Weise dargestellt werden [25]. Es ist also nur natürlich, zu versuchen diese Idee auch für formale Spezifikationen zu verwenden. Oftmals geschieht dies über eine grafische Benutzeroberfläche (GUI), die es erlaubt, Spezifikationen in Point-and-Click-Manier zu erstellen. Die Verwendung einer solchen GUI kann mehrere Vorteile haben:

- Syntaxprüfung
- Plausibilitätsprüfungen
- Typprüfungen
- Autokomplettierung

Die Syntaxprüfung mag trivial klingen, kann aber eine große Erleichterung sein, besonders um Spezifikationen zu schreiben. Es reduziert außerdem das Risiko Fehler wie z.B. falsche Klammern oder Ähnliches zu machen, da das Codehighlighting diese Fehler deutlicher macht. Plausibilitätsprüfungen können es ermöglichen, nach in der Domäne bekannten Anti-Patterns zu suchen oder auf trivial widersprüchliche Aussagen zu prüfen. Wenn die Spezifikationssprache typisiert ist, können auf dieser Ebene auch Typprüfungen durchgeführt werden. Zusätzlich kann die Autokomplettierung dem Benutzer helfen, insbesondere wenn er nicht mit allen Schlüsselwörtern vertraut ist und erleichtert somit im Allgemeinen den Spezifikationsprozess.

Die graphische Darstellung kann mehrere sehr unterschiedliche Formen haben, oft sind es allerdings diagrammähnliche Formen. Es ist nicht unüblich, vorhandene grafische Modelle wiederzuverwenden insbesondere die in der Domäne bereits verwendet werden und diese um zusätzliche Funktionen zu erweitern, um den formalen Anforderungen zu entsprechen (z.B. UML erweitert mit B-Spezifikationen

[26]). Für eine detaillierte Diskussion grafischer Spezifikationsansätze und entsprechender Werkzeuge wird der interessierte Leser auf [26] verwiesen.

## 7.3 Natürliche Sprachen als Spezifikationen

Im Idealfall möchte man Spezifikationen in natürlicher Sprache verfassen können. Das ist umso wichtiger, als dass Anforderungen normalerweise in natürlicher Sprache für Systeme vorliegen. Es gibt umfangreiche Forschung darüber, wie man Spezifikationen aus natürliche Sprache in eine formale Sprache übersetzten kann. Ein Ansatz ist die Einschränkung der natürlichen Sprache auf bestimmte Muster oder Satzstrukturen. Eine solche feste Struktur ist vorteilhaft bei der Übersetzung in formale Sprachen. Es gibt mehrere komplexere Ansätze, die verschiedene Arten von natürlicher Sprache ermöglichen in Anforderungen übersetzt zu werden [27, 11, 28]. Alle Ansätze dieser Art haben eine Form von Einschränkungen der natürlichen Sprache, die verwendet werden kann, erlauben aber dennoch eine benutzerfreundlichere Art und Weise der Spezifizierung.

# 8 Leitlinien

In diesem Kapitel stellen wir Richtlinien vor, die helfen sollen ein ML-basiertes System zu spezifizieren. Wir werden versuchen, allgemeine Ratschläge, die für alle ML-basierten Systeme anwendbar sind, und weisen darauf hin falls Einschränkungen gelten.

Die hier vorgestellten Leitlinien werden die wichtigsten Erkenntnisse der letzten Kapitel sein, die wir hier versuchen in konkrete Ratschläge zu verdichten. Der erste Leitfaden, den wir dem Leser bei der Spezifikation von ML-basierten Systemen empfehlen wollen ist der folgende:

**Guideline 1: Alternative Methoden prüfen**

Prüfen Sie, ob der Einsatz von ML in einem bestimmten Kontext unerlässlich ist.

Es gibt mehrere Beispiele für Anwendungen, bei denen Forscher versuchten, ML anzuwenden es sich allerdings herausstellte, dass “klassische” Techniken immer besser für den konkreten Anwendungsfall waren. Manchmal wird dies erst während (oder sogar erst nach) der Entwicklung bemerkt. Da die Spezifizierung eines ML-basierten Systems besonders herausfordernd sein kann, ist die einfachste Form diese Herausforderung zu umgehen, sie vollständig zu vermeiden. Dies ist nicht immer möglich oder machbar, sollte aber bei der Entwicklung sicherer Systeme berücksichtigt werden. Zusätzlich ist es teilweise dank Erkenntnissen, die man mit Hilfe der ML-Techniken gewonnen hat, möglich traditionelle Software für die gleiche Aufgabe zu entwickeln, die bisher nicht möglich war. Es lohnt sich also sich vor während aber auch nach Entwicklung eines Systems die Frage zu stellen, ob ML-Techniken an dieser Stelle (immer noch) nötig sind und sie gegebenenfalls zu vermeiden.

**Guideline 2: Annahmen berücksichtigen**

Beachten Sie sowohl explizit als auch implizit gemachte Annahmen.

Im Kapitel 4 haben wir diskutiert, was für Annahmen oftmals bei der Verwendung von ML-Techniken getroffen werden . Annahmen fälschlicherweise zu machen kann zu willkürlichen Schlussfolgerungen führen. Dies ist in unseren Augen besonders relevant für ML-Ansätze, da es hier mehrere Annahmen gibt, die häufig gemacht werden (teilweise da sie für traditionelle Software gelten), die im ML-Kontext nicht gelten. Für Spezifikationen ist dies jedoch von größter Bedeutung. Einige der Eigenschaften, die man leicht einfach erwartet, müssen bei ML-basierten Systemen explizit angegeben werden (z.B. Robustheit).

**Guideline 3: Kategorisieren Sie Ihr System**

Machen Sie sich klar, zu welcher Zuverlässigkeitsklasse Ihr System gehört und ergreifen Sie geeignete Maßnahmen.

In Kapitel 5 haben wir 4 Klassen von Zuverlässigkeitsklassen vorgestellt. Wir ermutigen den Leser, ein System in eine dieser Klassen einzustufen, bevor man konkrete Spezifikationen schreibt. Dies hat zwei Hauptvorteile: Erstens erlaubt es bereits in einem frühen Stadium zu reflektieren, welche Zusicherungen realistisch sind und zweitens entwickelt man ein besseres Verständnis dafür, welche der Spezifikations- und Verifikationstechniken geeignet sind, um die gewünschten Zusicherungen zu erreichen.

**Guideline 4: Verifikationsmethode im Auge behalten**

Behalten Sie die Verifikationsmethode, die Sie planen zu verwenden, während der Spezifikation im Hinterkopf.

Jede Verifikationstechnik kann ihre Stärken und Schwächen haben. Dies bereits bei der Spezifikation zu berücksichtigen, ist hilfreich, wenn später versucht wird diese Eigenschaft zu zeigen. Betrachten Sie als Beispiel “verlorene” Spuren, wie sie in 6.4 vorgestellt wurden. Die Herausforderung, mit “verlorenen” Spuren umzugehen, tritt nur bei Laufzeitüberwachung auf und ist somit bereits zur Spezifikationszeit relevant, da es die Art und Weise beeinflussen könnte, wie Eigenschaften angegeben werden. Ähnliche Beispiele existieren auch für andere Verifikationstechniken, weshalb wir den Leser dazu ermutigen, die beabsichtigte Verifikationsmethode während der Spezifikation im Hinterkopf zu behalten.

**Guideline 5: Gebrauch von geeigneten Spezifikationssprachen**

Denken Sie an die Vielfalt der anwendbaren Spezifikationssprachen und nutzen Sie eine Ihrem Kontext entsprechende.

Wie im Abschnitt 6 dargestellt, gibt es eine breite Palette von Eigenschaften von verschiedenen Typen, die auf ML-basierte Systeme anwendbar sein können. Aufgrund der unterschiedlichen Beschaffenheit dieser Eigenschaften ist die Auswahl der richtigen Spezifikationssprache sehr wichtig. Wir haben versucht, die wichtigsten Sprachen für verschiedene Arten von Eigenschaften vorzustellen und lieferten zusätzliche Referenzen als Ausgangspunkt für weitere Literaturrecherchen. Es ist also entscheidend welche Eigenschaften für jedes System/Kontext am relevantesten sind und den dazu passenden Spezifikationsmechanismus entsprechend zu wählen.

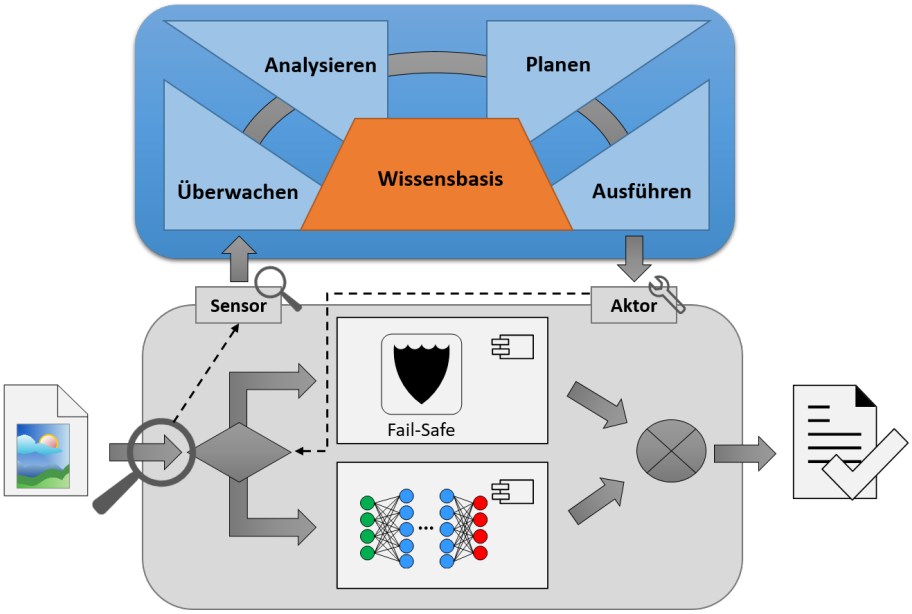


Abbildung 3: Aufbau des Demonstrators

# 9 Fallstudie

In diesem Kapitel stellen wir einen Demonstrator vor, der entwickelt wurde, um ein prototypisches Beispiel dafür zu geben, wie die Eigenschaften und Richtlinien, die in den vorhergehenden Kapiteln vorgestellt wurden, verwendet werden können, um ein sichereres System zu erreichen.

## 9.1 Szenario

Das Szenario ist ein Förderband, auf dem Werkstücke zusammengefügt werden. Genauer gesagt, müssen die Schrauben eines Werkstücks befestigt werden. Die Herausforderung dabei ist, dass es Schrauben unterschiedlicher Farbe (nämlich grün und rot) gibt und Schrauben fehlen. Die Aufgabe des Roboters in dieser Phase des Förderbandes besteht darin die grünen Schrauben anzuziehen, ohne die roten zu berühren. Für unseren Demonstrator werden wir uns nur auf die Implementierung der Software die für die Einteilung der Schrauben in 3 verschiedene Kategorien verantwortlich ist konzentrieren.

## 9.2 Spezifikation

Die ideale Eigenschaft für dieses System wäre, dass die Klassifizierung immer korrekt ist. Dies ist jedoch, wie in den vorangegangenen Kapiteln erwähnt, eine Eigenschaft die nicht formalisierbar und daher nicht realistisch ist. Um dennoch eine sinnvolle Spezifikation zu beschreiben, beschlossen wir, uns auf eine bekannte Schwäche von Bildklassifikationssoftware zu konzentrieren: Die Empfindlichkeit gegenüber Änderungen in der Beleuchtung. Um dies zu erreichen, haben wir zwei Eigenschaften kombiniert:

- Robustheit: Wir haben spezifiziert, dass das NN auf dem Trainingsset robust sein sollte. Dafür wurde die L-inf-Norm mit einem Epsilon von 20 genutzt (so dass jedes Pixel bis zu 20 Punkte verändert werden kann ohne die Klassifizierung zu beeinflussen)
- Eine Invariante über die Farbkanäle des Bildes: Wir überwachen den durchschnittlichen Pixelwert jedes Farbkanals und melden eine Verletzung, wenn der Wert um mehr als 10% vom Durch-

schnittswert des Trainingssets abweicht

Beachten Sie, dass die Robustheit in diesem Fall eine Eigenschaft ist, die “gutes” Verhalten beschreibt: wenn ein Bild in einen Epsilonbereich eines bekannten Bildes fällt, ist es nachgewiesen, dass die Klassifizierung korrekt sein ist. Im Vergleich dazu beschreibt die Invariante ein “schlechtes” Verhalten, da sie Fälle beschreibt, in denen die Farbwerte der untersuchten Bilder “verdächtig” sind. Wir haben diese beiden Eigenschaften auf die natürliche Art und Weise kombiniert: Es wird zunächst geprüft, ob die Eingabe durch die Robustheit als korrekt klassifiziert eingestuft werden kann. Ist dies nicht der Fall werden die Farbkanäle auf die Invariante geprüft. Was wir als formell betrachtet überprüfen ist die Eigenschaft: $\neg robust(x) \wedge \neg farbig\_invariant(x)$ für eine gegebene Eingabe $x$ und geeignet Definitionen der Prädikate.

## 9.3 Umsetzung

Unser Demonstrator ist mit Hilfe eines neuronalen Netzes in Python implementiert. Das Netzwerk erhält ein Bild des Werkstücks als Eingabe und gibt aus, wo Schrauben in welcher Farbe erkannt wurden. Es basiert auf Keras mit Tensorflow-Backend und wurde auf etwa 6000 Bilder, die zu diesem Zweck aufgenommen wurden trainiert. Der Einfachheit halber wurde das Netzwerk sehr klein (nur 4 Schichten: 2 Faltungsschichten und 2 vollständig zusammenhängende Schichten) gehalten. Das Netzwerk erreicht bei einem separaten Testset von Bildern eine Genauigkeit von etwa 98%. Zur Überprüfung der beschriebenen Eigenschaften haben wir dieses neuronale Netz in eine MAPE-K-Schleife eingebettet. A Die MAPE-K-Schleife ist ein architektonisches Entwurfsmuster für adaptive Systeme, das es ermöglicht ein System zu Überwachen und entsprechende Änderung seiner zukünftigen Handlungen einzubringen. Siehe Abbildung 3 für einen Überblick über unser System.

Die Verifizierung der Robustheit wurde mit einem Werkzeug namens *Neurify* durchgeführt [29]. *Neurify* wurde für jedes einzelne Bild des Trainingssets aufgerufen und wir konnten zeigen, dass das Netzwerk die gewünschte Robustheit aufweist.

## 9.4 Ergebnis

Durch die Anwendung des vorgestellten Ansatzes konnten wir für einen sehr kleines Testset zwei Dinge zeigen. Erstens konnte Bilder erkannt werden, da bei leichter Auffälligkeit der Farbkanäle eine automatische Korrektur angewendet werden konnte. Zweitens wurden für Bilder die zu sehr gestört waren von unserem System erkannt und ein Fail-Safe-Zustand wurde aktiviert. Unser System war also in der Lage die Qualität des ursprünglichen Ansatzes zu verbessern und die Anzahl der Fehlklassifikationen zu reduzieren. Da dieser Ansatz nur für eine sehr kleine Stichprobe getestet wurde, bleibt die Herausforderung dies noch ausführlicher zu testen.

## 9.5 Anwendung der Richtlinien

Bei der Konstruktion des Systems folgten wir den vorgeschlagenen Richtlinien aus dem vorherigen Kapitel. Wir werden nun darstellen, wie wir sie in diesem Beispiel verwendet haben:

Alternative Methoden prüfen

Im vorgestellten Szenario haben wir ML zur Klassifizierung von Bildern verwendet. Dies ist eines der prominentesten Beispiele, wo ML-basierte Ansätze sich im Vergleich zu traditionelle Ansätze als de facto unschlagbar etabliert haben. Den Autoren ist derzeit keine Möglichkeit bekannt, die es erlaubt für die Bildklassifikation ohne ML-basierte Techniken vergleichbare Ergebnisse zu erzielen.


Annahmen im Auge behalten

Wir haben nicht nur Robustheit angenommen, sondern sie mit einem Werkzeug bewiesen. Wir haben also eine Eigenschaft explizit spezifiziert, die wir während der Laufzeit nutzen konnten. Zusätzlich ist die IID-Annahme in diesem Fall nach Meinung der Autoren gerechtfertigt, da die Umgebung, in der der Roboter eingesetzt wird, stark eingeschränkt und somit eine repräsentativer Trainingssatz (verhältnismäßig) plausibel.

Kategorisieren Sie Ihr System

Wir haben die in Kapitel 5 vorgestellte Klassifizierung verwendet und dann eine Kombination aus Methoden aus den Klassen eins und zwei angewandt. Wir haben statisch verifiziert, dass das Netz auf den Trainingsdaten robust ist und dieses Wissen dann während der Laufzeit wiederverwendet, um das gewünschte Ergebnis zu erhalten.

Verifikationsmethode im Auge behalten

Der kritische Punkt für unseren Ansatz war, ob es realistisch ist schnell genug zu testen, ob ein bestimmtes Bild im Robustheitsbereich eines Trainingsdatums liegt. Falls dies zu langsam wäre würde die Überwachung wie wir sind umgesetzt haben an mangelnder Echtzeitfähigkeit scheitern. Um sicher zu sein, dass die Spezifikation realistisch waren, haben wir die erwartete Laufzeit eines solchen Verfahrens berechnet und erkannte, dass wir in der Lage sind die entsprechenden Berechnungen schnell genug auszuführen. Somit war die Verifikationsmethode bereits während der Spezifikation präsent und wurde berücksichtigt.

Verwendung geeigneter Sprachen

Wir verwendeten in diesem Fall keine spezifische Sprache, sondern implementierte die Monitore manuell direkt aus den gewünschten Eigenschaften. Diese vorgehen ist für sehr einfache Eigenschaften möglich wird allerdings für komplexere Eigenschaften immer schwieriger.

## Literatur


[1] SZEGEDY, Christian ; ZAREMBA, Wojciech ; SUTSKEVER, Ilya ; BRUNA, Joan ; ERHAN, Dumitru ; GOODFELLOW, Ian ; FERGUS, Rob: Intriguing Properties of Neural Networks. (2014), Februar

[2] SILVER, David ; SCHRITTWIESER, Julian ; SIMONYAN, Karen ; ANTONOGLOU, Ioannis ; HUANG, Aja ; GUEZ, Arthur ; HUBERT, Thomas ; BAKER, Lucas ; LAI, Matthew ; BOLTON, Adrian ; CHEN, Yutian ; LILLICRAP, Timothy ; HUI, Fan ; SIFRE, Laurent ; DRIESSCHE, George van d. ; GRAEPEL, Thore ; HASSABIS, Demis: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), Oktober, S. 354–359

[3] LEVINSON, Jesse ; ASKELAND, Jake ; BECKER, Jan ; DOLSON, Jennifer ; HELD, David ; KAMMEL, Soeren ; KOLTER, J Z. ; LANGER, Dirk ; PINK, Oliver ; PRATT, Vaughan u. a.: Towards fully autonomous driving: Systems and algorithms. In: *2011 IEEE Intelligent Vehicles Symposium (IV)* IEEE, 2011, S. 163–168

[4] DIETTERICH, Thomas G. ; HORVITZ, Eric J.: Rise of concerns about AI: reflections and directions. In: *Communications of the ACM* 58 (2015), September, S. 38–40

[5] SCHMIDHUBER, Juergen: Deep Learning in Neural Networks: An Overview. In: *Neural Networks* 61 (2015), Januar, S. 85–117

[6] KOTSIANTIS, S. B.: Supervised Machine Learning: A Review of Classification Techniques. In: *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Amsterdam, The Netherlands, The Netherlands : IOS Press, 2007, S. 3–24

[7] HERBRICH, Ralf ; WILLIAMSON, Robert C.: Learning and generalization: theoretical bounds. (2002)

[8] KÖNIGHOFER, Scott N. ; TOPCU, Ufuk: Safe Reinforcement Learning via Shielding. (2019)

[9] BECKERT, Bernhard ; BORMER, Thorsten ; KLEBANOV, Vladimir: Improving the Usability of Specification Languages and Methods for Annotation-Based Verification. In: AICHERNIG, Bernhard K. (Hrsg.) ; BOER, Frank S. (Hrsg.) ; BONSANGUE, Marcello M. (Hrsg.): *Formal Methods for Components and Objects*. Berlin, Heidelberg : Springer, 2012 (Lecture Notes in Computer Science), S. 61–79

[10] VAN WESEL, Perry ; GOODLOE, Alwyn E.: Challenges in the verification of reinforcement learning algorithms. (2017)

[11] LEHMAN, Joel et a.: The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities. In: *arXiv:1803.03453 [cs]* (2019), November

[12] AMODEI, Dario ; OLAH, Chris ; STEINHARDT, Jacob ; CHRISTIANO, Paul ; SCHULMAN, John ; MANÉ, Dan: Concrete Problems in AI Safety. (2016), Juli

[13] HOSSEINI, Hossein ; POOVENDRAN, Radha: Semantic adversarial examples. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, S. 1614–1619

[14] PNUELI, A.: The Temporal Logic of Programs. In: *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*, 1977, S. 46–57

[15] CINI, Clare ; FRANCALANZA, Adrian: An LTL Proof System for Runtime Verification. In: BAIER, Christel (Hrsg.) ; TINELLI, Cesare (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems* Bd. 9035. Springer Berlin Heidelberg, 2015. – ISBN 978–3–662–46680–3 978–3–662–46681–0, S. 581–595

[16] HO, Hsi-Ming ; OUAKNINE, Joël ; WORRELL, James: Online Monitoring of Metric Temporal Logic. In: BONAKDARPOUR, Borzoo (Hrsg.) ; SMOLKA, Scott A. (Hrsg.): *Runtime Verification*, Springer International Publishing, 2014 (Lecture Notes in Computer Science). – ISBN 978–3–319–11164–3, S. 178–192

[17] LEUCKER, Martin ; SCHALLHART, Christian: A Brief Account of Runtime Verification. In: *The Journal of Logic and Algebraic Programming* 78 (2009), Mai, Nr. 5, S. 293–303

[18] KOYMANS, Ron: Specifying Real-Time Properties with Metric Temporal Logic. In: *Real-Time Systems* 2 (1990), November, Nr. 4, S. 255–299

[19] MALER, Oded ; NICKOVIC, Dejan: Monitoring Temporal Properties of Continuous Signals. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* Bd. 3253. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–23167–7 978–3–540–30206–3, S. 152–166

[20] DONZÉ, Alexandre ; FERRÈRE, Thomas ; MALER, Oded: Efficient Robust Monitoring for STL. In: *Computer Aided Verification* Bd. 8044. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–39798–1 978–3–642–39799–8, S. 264–279

[21] KOLLER, Daphne ; FRIEDMAN, Nir: *Probabilistic graphical models: principles and techniques*. Cambridge, Mass. : MIT Press, 2009 (Adaptive computation and machine learning)

[22] GU, Xiaozhe ; EASWARAN, Arvind: Towards safe machine learning for CPS: infer uncertainty from training data. In: *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. Montreal, Quebec, Canada : Association for Computing Machinery, April 2019 (ICCPS '19), S. 249–258

[23] SUJEETH, Arvind K. ; LEE, HyoukJoong ; BROWN, Kevin J. ; ROMPF, Tiark ; CHAFI, Hassan ; WU, Michael ; ATREYA, Anand R. ; ODERSKY, Martin ; OLUKOTUN, Kunle: OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning, 2011

[24] FALL, Andrew ; FALL, Joseph: A domain-specific language for models of landscape dynamics. In: *Ecological Modelling* 141 (2001), Juli, S. 1–18

[25] MYERS, B.A. ; CHANDHOK, R. ; SAREEN, A.: Automatic data visualization for novice Pascal programmers. In: *[Proceedings] 1988 IEEE Workshop on Visual Languages*. Pittsburgh, PA, USA : IEEE Comput. Soc. Press, 1988, S. 192–198

[26] PANG, Cheng ; PAKONEN, Antti ; BUZHINSKY, Igor ; VYATKIN, Valeriy: A Study on User-Friendly Formal Specification Languages for Requirements Formalization, 2016

[27] MIRIYALA, Kanth ; HARANDI, Mehdi T.: Automatic derivation of formal software specifications from informal descriptions. In: *IEEE transactions on Software Engineering* (1991), Nr. 10, S. 1126–1142


[28] TOMMILA, Teemu ; PAKONEN, Antti ; VALKONEN, Janne: Ontology-Driven Natural Language Requirement Templates for Model Checking I&C Functions. In: *InEnlarged Halden programme group meeting, EHPG-2013, Storefjell, Norway, 10th–15th March*, 2013

[29] WANG, Shiqi ; PEI, Kexin ; WHITEHOUSE, Justin ; YANG, Junfeng ; JANA, Suman: Efficient Formal Safety Analysis of Neural Networks. In: BENGIO, S. (Hrsg.) ; WALLACH, H. (Hrsg.) ; LAROCHELLE, H. (Hrsg.) ; GRAUMAN, K. (Hrsg.) ; CESA-BIANCHI, N. (Hrsg.) ; GARNETT, R. (Hrsg.): *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, S. 6367–6377