



FAKULTÄT FÜR ELEKTROTECHNIK UND  
INFORMATIONSTECHNIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Masterarbeit

**Vergleich von heuristischen und Deep  
Learning basierten Schedulability Analyse  
Verfahren für die Migration von  
Softwarekomponenten zur Laufzeit**

Tatjana Utz







FAKULTÄT FÜR ELEKTROTECHNIK UND  
INFORMATIONSTECHNIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Masterarbeit

**Vergleich von heuristischen und Deep  
Learning basierten Schedulability Analyse  
Verfahren für die Migration von  
Softwarekomponenten zur Laufzeit**

**Comparison of Heuristic and Deep  
Learning Based Schedulability Analysis  
Methods for Migration of Software  
Components at Runtime**

Autor:	Tatjana Utz
Aufgabensteller:	Prof. Dr. Uwe Baumgarten
Betreuer:	Sebastian Eckl, M.Sc.
Abgabedatum:	15.06.2019





Ich versichere hiermit, dass ich die von mir eingereichte Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

München, 15.06.2019

Tatjana Utz



# Zusammenfassung

Ziel dieser Masterarbeit ist der Vergleich von heuristischen und Deep Learning basierten Schedulability-Analyseverfahren. Hierfür werden mehrere etablierte Schedulability-Analysemethoden implementiert: Simulation, Prozessorauslastungs-Tests, Antwortzeit-Analyse und Arbeitsbelastungs-Tests. Zudem wird eine Schnittstelle zu der gegebenen Datenbank, die die Tasks und Task-Sets enthält, entwickelt. Es zeigt sich, dass selbst die exakten Verfahren nur eine Korrektklassifikationsrate von 93 % erreichen, was im Verhalten des Genode-Betriebssystems und daraus resultierenden ungenauen Task-Ausführungszeiten begründet liegt. Zum Vergleich wird ein rekurrentes neuronales Netz aus LSTM-Zellen für die Vorhersage der Lauffähigkeit von Task-Sets entwickelt. Es werden Experimente mit verschiedenen Hyperparametern durchgeführt, um die Leistung des Machine Learning Modells zu steigern. Dabei wird deutlich, dass die Leistung des rekurrenten neuronalen Netzes am stärksten von der Anzahl der Neuronen, der Anzahl der Zellen und der Dropout-Wahrscheinlichkeit abhängt. Ein Vergleich der beiden Schedulability-Analyseansätze bestätigt, dass die Leistung des rekurrenten neuronalen Netzes mit einer Genauigkeit von 98,58 % vergleichbar mit den exakten Verfahren ist, aber keine Verbesserung bei der Geschwindigkeit der Vorhersagen bietet. Dennoch ist der Einsatz in einem sicherheitskritischen System denkbar.

## Abstract

The aim of this master thesis is to compare heuristic and deep learning based schedulability analysis methods. For this purpose several traditional schedulability analysis methods are implemented: simulation, processor utilization tests, response time analysis and workload tests. In addition, an interface to the given database containing the tasks and task sets will be developed. It turns out that even the exact methods only achieve an accuracy of 93 %, which is due to the behavior of the Genode OS and the resulting inaccurate task execution times. For comparison, a recurrent neural network of LSTM cells is developed for predicting the executability of task sets. Experiments with different hyperparameters are performed to increase the performance of the machine learning model. It becomes clear that the performance of the recurrent neural network depends most on the number of neurons, the number of cells and the dropout probability. A comparison of the two schedulability analysis approaches confirms that the performance of the recurrent neural network with a precision of 98,58 % is comparable to the exact methods, but offers no improvement in the speed of predictions. Nevertheless, its use in a safety-critical system is conceivable.





# Inhaltsverzeichnis

<b>Zusammenfassung / Abstract</b>	<b>v</b>
<b>1 Einführung</b>	<b>1</b>
<b>2 Systembeschreibung</b>	<b>5</b>
2.1 Aufbau des MaLSAMi-Projekts . . . . .	5
2.2 Architektur des KIA4SM-Projekts . . . . .	5
2.2.1 Offline System . . . . .	8
2.2.2 Embedded Online System . . . . .	9
2.2.3 Dom0-Netzwerk . . . . .	10
<b>3 Grundlagen</b>	<b>11</b>
3.1 Schedulability-Analyse in Echtzeitsystemen . . . . .	11
3.1.1 Echtzeitsysteme . . . . .	11
3.1.2 Task-Modell . . . . .	11
3.1.3 Scheduling . . . . .	13
3.1.4 Schedulability-Analyse . . . . .	16
3.2 Machine Learning, neuronale Netze und Deep Learning . . . . .	17
3.2.1 Arten des Machine Learning . . . . .	17
3.2.2 Herausforderungen beim Machine Learning . . . . .	21
3.2.3 Grundlagen neuronaler Netze . . . . .	22
3.2.4 Vorwärts gerichtete Netze . . . . .	26
3.2.5 Deep Learning . . . . .	33
3.2.6 Rekurrente neuronale Netze . . . . .	35
<b>4 Stand der Technik</b>	<b>45</b>
4.1 Traditionelle und heuristische Schedulability-Analyse . . . . .	45
4.1.1 Fixed Priority . . . . .	45
4.1.2 Earliest Deadline First . . . . .	49
4.2 Schedulability-Analyse mit Machine Learning . . . . .	52
4.2.1 Veröffentlichungen . . . . .	52
4.2.2 Frühere Arbeiten am Lehrstuhl . . . . .	54
<b>5 Gegebene Datenbank mit den Task-Sets</b>	<b>57</b>
5.1 Aufbau der gegebenen Datenbank . . . . .	57
5.2 Datenanalyse . . . . .	59

<b>6</b>	<b>Traditionelle Schedulability-Analyse</b>	<b>63</b>
6.1	Hauptprogramm . . . . .	64
6.2	Kommandozeilenschnittstelle . . . . .	65
6.3	Datenbankschnittstelle . . . . .	68
6.3.1	Die Klasse Task . . . . .	69
6.3.2	Die Klasse Taskset . . . . .	69
6.3.3	Die Klasse Database . . . . .	69
6.4	Benchmark . . . . .	72
6.5	Logging . . . . .	74
6.6	Schedulability-Analyse Frameworks . . . . .	75
6.7	Simulation . . . . .	77
6.7.1	Scheduler . . . . .	77
6.7.2	Simulation eines Task-Sets . . . . .	79
6.7.3	Beispiel . . . . .	80
6.8	Prozessorauslastung . . . . .	80
6.8.1	Einfacher Prozessorauslastungs-Test . . . . .	83
6.8.2	RM-Prozessorauslastungs-Test . . . . .	83
6.8.3	HB-Prozessorauslastungs-Test . . . . .	84
6.9	Antwortzeit-Analyse . . . . .	85
6.9.1	Antwortzeit-Analyse eines Task-Sets . . . . .	85
6.9.2	Berechnung der Antwortzeit . . . . .	87
6.9.3	Beispiel-Rechnung . . . . .	87
6.10	Arbeitsbelastung . . . . .	89
6.10.1	RM-Arbeitsbelastungs-Test . . . . .	90
6.10.2	Hyperplanes $\delta$ -Exact Test . . . . .	95
6.11	Evaluation . . . . .	99
<b>7</b>	<b>Schedulability-Analyse mit einem rekurrenten neuronalen Netz</b>	<b>105</b>
7.1	Hauptprogramm . . . . .	107
7.1.1	Laden der Daten . . . . .	107
7.1.2	Datenvorverarbeitung . . . . .	110
7.2	Datenbankschnittstelle . . . . .	113
7.3	Benchmark . . . . .	114
7.4	Datenbank-Filter . . . . .	114
7.5	Logging . . . . .	116
7.6	Machine Learning Frameworks . . . . .	116
7.7	Machine Learning Modelle . . . . .	118
7.7.1	Erstellung eines Keras-Modells . . . . .	118
7.7.2	Training eines Keras-Modells . . . . .	120
7.7.3	Evaluation eines Keras-Modells . . . . .	122
7.8	Hyperparameter-Optimierung . . . . .	122
7.9	Visualisierung der Ergebnisse der Hyperparameter-Optimierung . . . . .	127

7.10 Evaluation . . . . .	129
7.10.1 Anpassung der Hyperparameter . . . . .	129
7.10.2 Auswertung der besten Hyperparameter-Kombination . . . . .	134
7.10.3 Korrelation zwischen der RNN-Leistung und den untersuchten Hyperparametern . . . . .	135
<b>8 Vergleich der Schedulability-Analyseansätze</b>	<b>139</b>
<b>9 Einschränkungen und Ausblick</b>	<b>143</b>
<b>10 Fazit</b>	<b>147</b>
<b>Abkürzungsverzeichnis</b>	<b>149</b>
<b>Abbildungsverzeichnis</b>	<b>151</b>
<b>Tabellenverzeichnis</b>	<b>153</b>
<b>Listingverzeichnis</b>	<b>155</b>
<b>Literatur</b>	<b>157</b>



# 1 Einführung

Der Traum vom selbstfahrenden Auto - früher noch Science-Fiction - rückt immer mehr in greifbare Nähe. Bereits heute entlasten computerbasierte Assistenzsysteme den Fahrer, zum Beispiel beim Einhalten der Fahrspur oder beim Einparken. Aber nicht nur die Intelligenz des Autos wird gesteigert, sondern auch die Intelligenz der Umgebung. Intelligente Verkehrs- und Transportsysteme (engl. *Intelligent Transport Systems, ITS*) machen den Straßenverkehr sicherer, effizienter und umweltfreundlicher [EKB15]. Die Informationen von verschiedenen Quellen, wie zum Beispiel Überwachungskameras und Wettersensoren, fließen zusammen und machen so das gesamte System intelligenter, zum Beispiel mit angepasster Routenplanung bei der Navigation. Dabei spielen auch Mobilgeräte wie Smartphones und Tablets eine immer größere Rolle. Mit „Smart Mobility“ oder „Kooperativen Intelligenten Transportsystemen“ (engl. *Cooperative Intelligent Transportation Systems, C-ITS*) wird der Trend der gegenseitigen Vernetzung und des Zusammenspiels von Fahrzeugen, Mobilgeräten und Verkehrs- und Transportinfrastruktur bezeichnet. Da die Entwicklung von C-ITS Clients bisher unabhängig voneinander stattfindet und in geschlossenen, heterogenen Hardware-Systemen resultiert, sind sie nicht ohne Weiteres miteinander kompatibel. Daher ist ein kooperatives Verhalten, das ad-hoc verfügbar ist, nur durch eine Neugestaltung der gesamten, system-übergreifenden Zusammenarbeit möglich. Ziel ist die Entwicklung einer gemeinsamen Plattform, um eine homogene und gemeinsame Laufzeitumgebung bereitzustellen, wie sie in Abbildung 1.1 dargestellt ist. Das Projekt KIA4SM (*Kooperative Integrationsarchitektur für zukünftige Smart-Mobility-Lösungen*) des Lehrstuhls für Betriebssysteme an der Technischen Universität München versucht dieses Problem durch die Entwicklung einer flexiblen Integrationsarchitektur basierend auf universellen Steuereinheiten (engl. *electronic control unit, ECU*) zu lösen [EKB15]. Eine solche Architektur erlaubt die Anpassung des Gesamtsystems an Hardware/Software Plug-and-Play Eigenschaften, womit eine geräte-unabhängige Bereitstellung und dynamische Rekonfiguration von Funktionalität zur Laufzeit ermöglicht wird.

Im Gegensatz zu dem sich schnell wandelnden ITS-Bereich ist die Automobilbranche geprägt durch eher lange und unflexible Entwicklungszyklen. Außerdem besteht die aktuelle Praktik darin, für jede neue, Software-basierte Funktionalität eine eigene Steuereinheit zu entwickeln, was eine starke Hardware-/Software-Bindung zur Folge hat. Gerade im Zusammenhang mit E-Autos muss deshalb ein Umdenken stattfinden, da Platz und Energie nur begrenzt zur Verfügung stehen.

Eine mögliche Lösung ist der gegenwärtige Trend zur Hardware-Konsolidierung gepaart mit einer Software-Virtualisierung. Statt vieler spezialisierter Hardware-Geräte

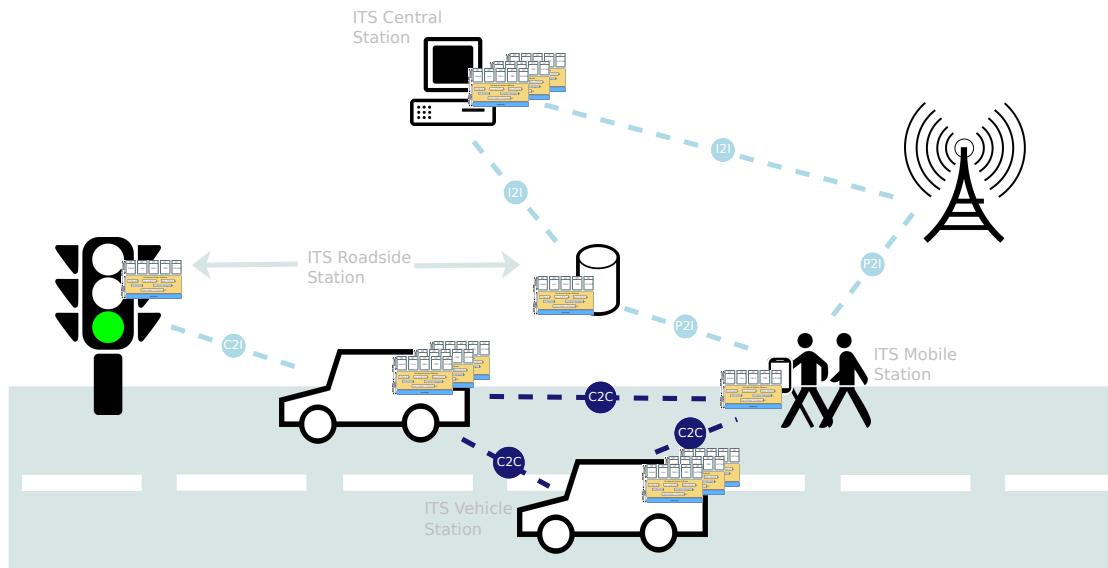


Abbildung 1.1: KIA4SM Vision - eine homogene Plattform für heterogene Geräte (Quelle: [EKB15, S. 2]).

werden nur noch einige wenige, universelle Geräte verwendet. Deshalb wird die notwendige Sicherheit nicht mehr wie bisher durch die Trennung der sicherheitskritischen Software-Komponenten auf unterschiedliche ECUs erreicht. Stattdessen basiert die vorgeschlagene Laufzeitumgebung auf einer Typ-1 Hypervisor-basierten Paravirtualisierung. Der Hypervisor ist eine Software-Abstraktionsschicht und verwaltet die Hardware-Ressourcen durch die Bereitstellung von mehreren Ausführungsumgebungen [Grö15]. Sogenannte virtuelle Maschinen oder Partitionen sind voneinander isoliert, obgleich sie über den Hypervisor die gleiche Hardware nutzen. Auf diese Weise ist bei gemeinsamer Hardware eine sichere Partitionierung von Funktionen anhand separierter Tasks möglich [EKB15].

Die Virtualisierung löst auch die feste Hardware-/Software-Bindung, wodurch eine Plug-and-Play Funktionalität und ein kontextsensitives Reaktionsverhalten ermöglicht werden, wie zum Beispiel die (De-)Aktivierung, Migration oder Update/Upgrade von Software-basierten Funktionen. Zusammengefasst ergibt sich ein Wechsel von der etablierten Hardware-gestützten zu einer Software-basierten Redundanz. Dieser Wechsel ist auch die Grundlage für neuartige Ausfallsicherheits-, Load-Balancing und Energiespar-Szenarien.

Diese Arbeit beschäftigt sich mit dem kontextsensitiven Reaktionsverhalten, genauer der Migration von Software-Komponenten. Dadurch ist Software nicht auf eine einzige ECU beschränkt und ein Hardwarefehler einer ECU bedeutet nicht notwendigerweise den Verlust der Funktionalität. Ein Systemausfall wird verhindert, indem die entsprechenden Tasks von der fehlerhaften Steuereinheit auf eine andere migriert werden.

---

Gerade bei sicherheitskritischen Komponenten, zum Beispiel dem Bremssystem, ist dies von höchster Relevanz.

Um jedoch ein funktionierendes System zu garantieren, muss die Migration von Software zwischen Steuereinheiten zur Laufzeit möglich sein. Das ist keine triviale Aufgabe, da der aktuelle Stand und die Eigenschaften der Tasks sowie die verfügbaren Ressourcen der Steuergeräte berücksichtigt werden müssen. Um die Migration von Software-Komponenten in einer gemeinsamen Echtzeit-Umgebung möglich zu machen, ist eine Strategie erforderlich, nach der die entsprechenden Tasks den ECUs zugeordnet werden. Außerdem muss sichergestellt werden, dass die Echtzeit-Bedingungen des resultierenden Task-Sets einer Steuereinheit eingehalten werden, und zwar vor der eigentlichen Migration. Dieses Problem ist allgemein bekannt als Schedulability-Analyse.

Es gibt eine Vielzahl verschiedener Ansätze für die Schedulability-Analyse. Exakte Analysen sind meist sehr komplex und rechenintensiv. Da gerade in Echtzeit-Systemen die Schnelligkeit und beschränkte Ressourcen eine große Rolle spielen, wird in [EKB15] die Idee einer Machine Learning basierten Schedulability-Analyse präsentiert. Das Postulat lautet, dass Methoden aus dem Bereich des maschinellen Lernens durch die Erkennung von bestimmten Mustern in Task-Sets schneller eine Entscheidung bezüglich der Lauffähigkeit liefern als traditionelle Methoden.

Bereits vorliegende Arbeiten am Lehrstuhl behandeln die Implementierung verschiedener Machine Learning Verfahren. Allerdings überprüfen diese Arbeiten nur, ob die verwendeten Verfahren überhaupt die Lauffähigkeit eines Task-Sets bzw. einzelner Tasks bestimmen können. Keine dieser Arbeiten betrachtet einen direkten Vergleich der Schedulability-Analyseansätze.

Ziel dieser Arbeit ist ein Vergleich zwischen einem Machine Learning Verfahren und etablierten Schedulability-Analysemethoden. Dabei wird insbesondere untersucht, ob die Analyse mit Machine Learning Methoden tatsächlich die postulierte Schnelligkeit gegenüber den gängigen Ansätzen bietet. Außerdem wird überprüft, ob es Unterschiede bei der Korrektheit zwischen den Verfahren gibt.

Die Arbeit gliedert sich wie folgt: Zunächst wird in Kapitel 2 ein Überblick über das zugrundeliegende System gegeben. In Kapitel 3 werden notwendige Grundlagen zur Schedulability-Analyse in Echtzeitsystemen und dem Bereich des maschinellen Lernens gegeben. Es folgt in Kapitel 4 eine Zusammenfassung von bereits bekannten Arbeiten zu den Themen klassische Schedulability-Analyse sowie im Zusammenhang mit Machine Learning. Außerdem werden die früheren Arbeiten am Lehrstuhl zusammengefasst. In Kapitel 5 gibt einen Überblick über die vorhandenen Daten, das bedeutet die gegebenen Tasks und Task-Sets. Kapitel 6 präsentiert das Konzept und die Implementierung der Komponente für traditionelle Schedulability Analyseverfahren. Der Aufbau eines neuronalen Netzes für die Schedulability-Analyse ist in Kapitel 7 gezeigt. Anschließend folgt in Kapitel 8 eine zusammenfassende Evaluierung der Ergebnisse und der Vergleich zwischen den Schedulability-Analyseansätzen. Als Abschluss werden in Kapitel 9 die aufgetretenden Probleme und Einschränkungen genannt sowie ein Ausblick auf zukünftige Arbeiten gegeben. Das Fazit in Kapitel 10 schließt die Arbeit ab.





## 2 Systembeschreibung

Zur Umsetzung der Ziele des KIA4SM-Projekts sind viele Komponenten notwendig, die bereits teilweise durch vorangegangenen Arbeiten am Lehrstuhl zur Verfügung stehen. Nachfolgend wird ein kurzer Überblick über die Architektur und die vorhandenen Komponenten des KIA4SM-Projekts gegeben.

### 2.1 Aufbau des MaLSAMi-Projekts

In Abbildung 2.1 ist der Aufbau des *MaLSAMi* (engl. *Machine-Learning supported Schedulability Analysis for Migration of software-based components during runtime*)<sup>1</sup> Projekts zu sehen. Dieses Projekt ist Teil des KIA4SM-Projekts und beinhaltet die Machine Learning gestützte Schedulability-Analyse.

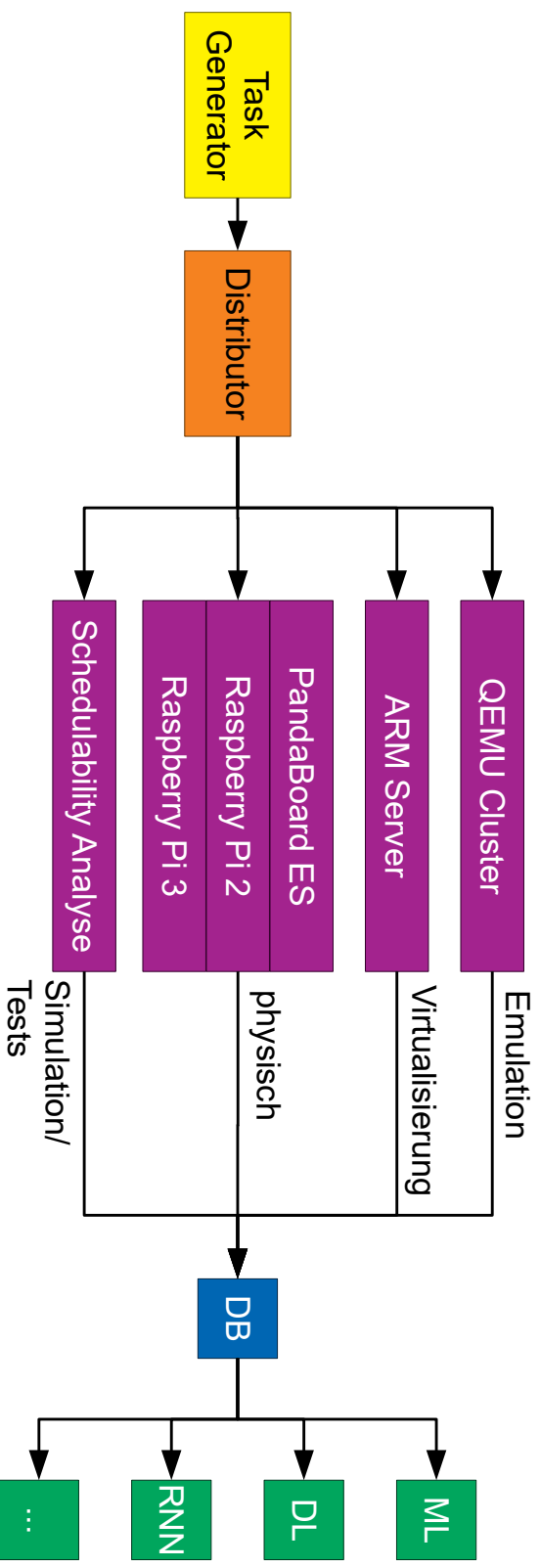
Der Task- bzw. Task-Set-Generator erstellt aus verschiedenen Beispieltasks unterschiedlich große Task-Sets. Diese Task-Sets werden an den Distributor übergeben, der entscheidet, auf welcher Plattform die Task-Sets ausgeführt werden. Zur Zeit stehen ein QEMU Cluster, ein ARM Server, PandaBoards ES sowie Raspberry Pis 2 und 3 zur Verfügung. Zusätzlich ist auch die Durchführung einer konventionellen Schedulability-Analyse, wie beispielsweise einer Simulation, möglich. Die Ergebnisse über die Lauffähigkeit der Task-Sets werden in einer SQL-Datenbank abgespeichert. Mit Hilfe der Daten aus der Datenbank werden schließlich verschiedene Machine Learning Modelle für die Klassifikation von Task-Sets bezüglich ihrer Lauffähigkeit trainiert. Bereits vorhanden ist eine Komponente mit Shallow Learning Verfahren (ML) und ein künstliches neuronales Netz (DL). Ziel dieser Arbeit ist die Entwicklung der Komponente für die traditionelle Schedulability-Analyse. Außerdem wird ein rekurrentes neuronales Netz implementiert.

### 2.2 Architektur des KIA4SM-Projekts

Abbildung 2.2 zeigt das KIA4SM-System und den Zusammenhang zwischen den enthaltenen Komponenten.

---

<sup>1</sup><https://malsami.github.io/>



**Daten Distribution      Daten Generierung      Daten Speicherung      Daten Analyse**

Abbildung 2.1: Aufbau der Toolchain des MaLSAMi-Projektes (Eigene Darstellung).

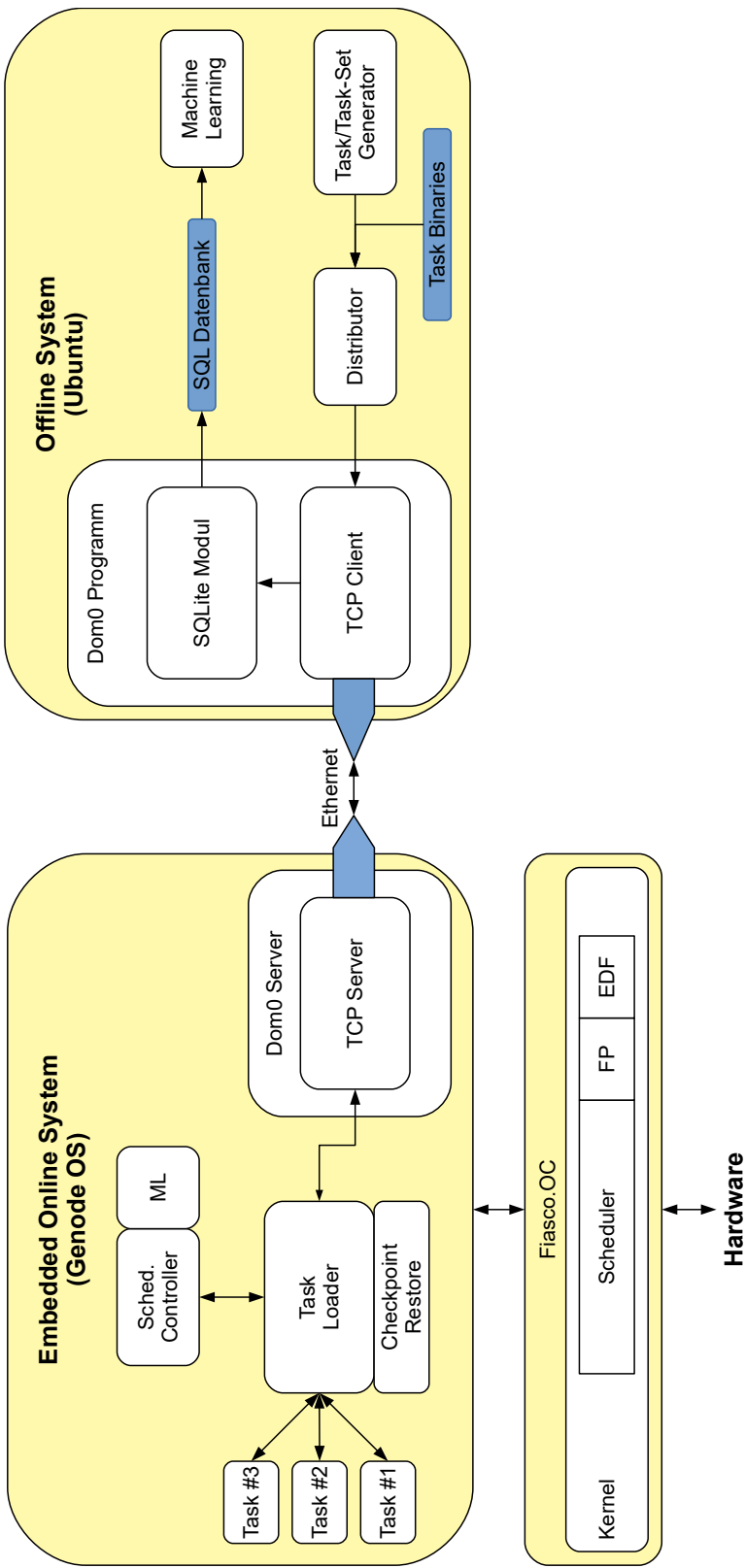


Abbildung 2.2: Vorhandene Architektur des KIA4SM-Projektes (In Anlehnung an [Gub16, S. 19], [Häc18, S. 3]).

### 2.2.1 Offline System

Die Bachelorarbeit von Johannes Fischer [Fis18] befasst sich mit der Entwicklung einer Taskgenerierungskomponente. Die entwickelte Komponente beinhaltet jeweils einen Task- und Task-Set-Generator sowie einen Distributor. Der Task-Generator erzeugt Tasks, die vom Task-Set-Generator verwendet werden, um Task-Sets zu generieren. Um eine einheitliche Basis zu garantieren, stehen einige Beispieltasks zur Verfügung, die Tabelle 2.1 beschreibt und auf unterschiedliche Weise miteinander kombiniert werden können. Auch die Größe eines Task-Sets ist variabel. In der aktuellen Version der verwendeten Datenbank werden Task-Sets bis zu einer maximalen Größe von vier Tasks unterstützt.

Tabelle 2.1: Beschreibung der verfügbaren Beispieltasks für die Erzeugung von Task-Sets (In Anlehnung an [Häc18, S. 8]).

Task-Name (PKG)	Beschreibung
cond_mod	Durchläuft bei ungeradem Eingabeparameter eine leere Schleife mit der dem Eingabeparameter entsprechenden Anzahl an Iterationen. Bei geradem Eingabeparameter unternimmt der Task nichts.
hey	Ausgabe von „hey“ auf der Konsole.
pi	Berechnet die Zahl Pi iterativ. Die Anzahl der Iterationen wird durch den Eingabeparameter bestimmt.
tumatmul	Berechnet alle Primzahlen bis zur per Eingabeparameter bestimmten Grenze. Gibt anschließend die letzten fünf Werte auf der Konsole aus.

Der Distributor erhält die erzeugten Task-Sets und entscheidet, wo diese ausgeführt werden. Zur Auswahl stehen sowohl virtuelle als auch reale Cluster aus Einplatinenrechnern. Wie aus Abbildung 2.1 ersichtlich, stehen momentan fünf verschiedene Cluster zur Verfügung:

- QEMU Cluster: Virtualisierungssoftware zur Emulation von Hardware,
- ARM Server: Virtualisierung von Hardware,
- PandaBoard ES: Einplatinenrechner,
- Raspberry Pi 2: Einplatinenrechner,
- Raspberry Pi 3: Einplatinenrechner.

Zusätzlich soll zukünftig eine klassische Schedulability-Analyse wie zum Beispiel die Simulation oder andere Tests möglich sein. Die Implementierung einer solchen Komponente ist Teil dieser Arbeit. Die Daten der Task-Sets werden über das Dom0-Netzwerk an die ausgewählte Plattform übertragen und dort ausgeführt. Die Ergebnisse über die Lauffähigkeit der Task-Sets werden wieder über die Dom0-Schnittstelle empfangen und in einer SQL-Datenbank abgespeichert. Diese wird zum Trainieren von verschiedenen Machine Learning Methoden verwendet.

### 2.2.2 Embedded Online System

Das Betriebssystem des eingebetteten Online-Systems basiert auf dem Genode OS Framework, einem kommerziellen Bausatz zur Entwicklung von sicheren und angepassten Betriebssystemen, erarbeitet an der Technischen Universität Dresden [Lab19]. Der Fokus liegt auf der Sicherheit, die durch die Abkapselung von Gerätetreibern, Systemdiensten, Protokollstacks und Anwendungen erreicht wird. Als Grundlage für das Betriebssystem dient ein x86 und ARM kompatibler Fiasco.OC L4 Mikrokern. Die Entwicklung eines geeigneten Betriebssystems für das KIA4SM-Projekt ist das hauptsächliche Ziel des ArgOS-Forschungsprojektes des Lehrstuhls für Betriebssysteme an der Technischen Universität München<sup>2</sup>.

Das KIA4SM-Projekt stellt verschiedene Konfigurationen von Genode für mehrere unterstützte Einplatinenrechner zur Verfügung [Ham17]. Alternativ zu den Hardware-Komponenten ist auch eine Emulation mit QEMU möglich. QEMU ist eine open-source Virtualisierungssoftware, die das Emulieren von speziellen CPUs oder Einplatinenrechnern ermöglicht [Ham17; Häc18].

Die Schnittstelle zwischen der Hardware und dem Genode-Betriebssystem ist der Fiasco.OC Mikrokern [Häc18]. Dieser Kernel enthält auch den Scheduler, der die auftretenden Tasks entweder nach dem Fixed Priority (FP) oder dem Earliest Deadline First (EDF) Algorithmus plant. Nähere Informationen zu den genannten Algorithmen befinden sich in Kapitel 3.1.3.

Der Taskloader ist für die Weitergabe der Tasks an den Scheduler verantwortlich [Häc18]. Dieser kann ausgewählte Tasks auch deaktivieren, wobei deaktivierte Tasks nicht an den Scheduler weitergeleitet werden. Der Scheduling Controller stellt sicher, dass die vorhandenen Tasks problemlos ablaufen. Damit diese Aufgabe bewältigt werden kann, hat der Scheduling Controller die Fähigkeit, dem Taskloader vorzuschreiben, welche Tasks deaktiviert werden. Die Entscheidungsfindung des Scheduling Controllers erfolgt dabei mit Hilfe von Antwortzeitanalyse für Fixed Priority Tasks. Bei Earliest Deadline First Tasks wird eine Analyse anhand von Fairness oder Auslastung dynamisch zur Laufzeit durchgeführt. Das Ziel des KIA4SM-Projektes sieht anstelle der genannten Verfahren an dieser Stelle eine Machine Learning Komponente für die Schedulability-Analyse vor.

---

<sup>2</sup><https://argos-research.github.io/>

### 2.2.3 Dom0-Netzwerk

Die Verbindung zwischen dem eingebetteten Online-System und dem Offline-System ist das Dom0-Netzwerk. Die Masterarbeit von Georg Guba beschäftigt sich mit der Entwicklung der Dom0-Toolchain, die für das Senden von Task-Sets an das Online-System und das Erstellen einer Datenbank für das Machine Learning zuständig ist [Gub16; Ham17]. Das Dom0-System besteht aus einer Online-Server-Komponente, die Teil des eingebetteten Systems ist, und einer Offline-Komponente, die als Client für den Server dient [Ham17]. Die Kommunikation findet über Ethernet statt.

Nach dem Start der Toolchain wartet die TCP-Komponente des Servers auf die Verbindungsanfrage eines Clients [Ham17]. Die Task- bzw. Task-Set-Generatoren des Offline-Systems generieren Task-Sets, für die der Distributor bestimmt, ob diese auf realen oder emulierten Einplatinenrechnern ausgeführt werden. Diese Daten werden an das Dom0-Netzwerk übergeben und an das Online-System übertragen. Nach der Ausführung der Task-Sets werden die resultierenden Log-Daten über die Lauffähigkeit durch das Dom0-Netzwerk an das Offline-System gesendet und mit Hilfe des SQLite-Moduls in eine Datenbank geschrieben. Diese Datenbank wird von einer Machine Learning Komponente genutzt, um die Lauffähigkeit der Task-Sets zu analysieren.

## 3 Grundlagen

### 3.1 Schedulability-Analyse in Echtzeitsystemen

#### 3.1.1 Echtzeitsysteme

Echtzeitsysteme sind Rechensysteme, die innerhalb eines bestimmten Zeitrahmens auf Ereignisse in ihrer Umgebung reagieren müssen [But11]. Folglich hängt das korrekte Verhalten dieser Systeme nicht nur vom Ergebnis der Berechnung ab, sondern auch von dem Zeitpunkt, zu dem ein Ergebnis erhalten wird. Eine Reaktion, die zu spät erfolgt, kann nutzlos oder sogar gefährlich sein. Beispiele für Echtzeitsysteme sind Flugsteuerungssysteme in der Avionik, Anwendungen für die Automobilindustrie und medizinische Systeme. In vielen Fällen sind Echtzeit-Computer, die eine Anwendung ausführen, in ein zu kontrollierendes System eingebettet. Solche eingebetteten Systeme sind zum Beispiel Mobilgeräte wie Smartphones oder größere Systeme wie Autos und Flugzeuge.

Es werden zwei Arten von Echtzeit- bzw. eingebetteten Systemen unterschieden: harte Echtzeitsysteme und weiche Echtzeitsysteme [BW96] [But11]. In harten Echtzeitsystemen ist es zwingend notwendig, dass die Antworten vor einer vorgegebenen Deadline erfolgen. Wird ein Ergebnis erst nach einer Deadline erhalten, kann das katastrophale Folgen für das zu kontrollierende System haben. In weichen Echtzeitsystemen hingegen ist die Reaktionszeit zwar wichtig, aber das System wird auch weiterhin richtig funktionieren, wenn Deadlines gelegentlich nicht eingehalten werden. Meist hat ein verspätetes Ergebnis trotzdem einen gewissen Nutzen für das System, auch wenn es einen Leistungsabfall nach sich zieht.

#### 3.1.2 Task-Modell

In vielen Echtzeit-Steuerungsanwendungen bilden periodische Aktivitäten den größten Rechenaufwand im System [But11]. Periodische Prozesse ergeben sich in der Regel aus der Erfassung von Sensordaten, Regelkreisen und der Systemüberwachung. Wenn eine Steuerungsanwendung aus mehreren parallelen, periodischen Tasks mit individuellen zeitlichen Einschränkungen besteht, muss das Betriebssystem garantieren, dass jede periodische Instanz regelmäßig mit der entsprechenden Rate aktiviert und innerhalb ihrer Frist abgeschlossen wird. Um solche Methoden und deren Tests zu beschreiben, wird im Folgenden das verwendete Task-Modell definiert.

Ein periodischer Prozess ist ein Task-Set bestehend aus mehreren Tasks  $\tau_1, \dots, \tau_n$ , die gemeinsam eine Systemfunktion bereitstellen. Jeder Task besteht wiederum aus mehreren Jobs  $J_1, \dots, J_k$ , wobei  $J_1$  die erste periodische Instanz und  $J_k$  die  $k$ -te periodische Instanz von  $\tau_i$  darstellt. Ein Task  $\tau_i$  wird durch das Tupel  $(T_i, D_i, C_i)$  dargestellt und wird durch die folgenden Eigenschaften bestimmt:

- Auslösezeit  $A_i$  (engl. *release time*): Zeitpunkt, zu dem ein Task zur Ausführung bereit ist, auch Aktivierung genannt.
- Periode  $T_i$  (engl. *period*): Zeitspanne, nach der der nächste Job eines periodischen Tasks ausgelöst wird.
- Frist  $D_i$  (engl. *deadline*): Zeitspanne, nach der die Ausführung des Tasks  $\tau_i$  beendet sein muss.
- Ausführungszeit  $C_i$  (engl. *execution time*): reine Rechenzeit, die zur vollständigen Ausführung eines Tasks gebraucht wird, entspricht häufig der Rechenzeit im ungünstigsten Fall (engl. *worst-case execution time, WCET*).
- Priorität  $P_i$ : Priorität von  $\tau_i$  im Falle eines prioritäts-basierten Systems.

Neben periodischen Tasks gibt es auch aperiodische bzw. sporadische Tasks. Beinahe jedes Echtzeitsystem muss auf externe Events reagieren, die zu zufälligen Zeitpunkten auftreten. Wenn so ein Event auftritt, führt das System ein Set von Jobs als Antwort aus. Die Auslösezeiten von solchen Jobs sind unbekannt bis das auslösende Event auftritt. Die beschriebenen Jobs werden sporadisch oder aperiodisch genannt, weil sie zufällige Auslösezeiten haben.

Zur Vereinfachung werden folgende Annahmen über das Task-Modell getroffen [LL73]:

- A1 Die Instanzen eines Tasks  $\tau_i$  sind periodisch und werden regelmäßig mit einer konstanten Rate aktiviert. Das Intervall  $T_i$  zwischen zwei nachfolgenden Aktivierungen ist die Periode des Tasks.
- A2 Alle Instanzen eines periodischen Tasks  $\tau_i$  haben die gleiche Deadline  $D_i$ , die der Periode  $T_i$  entspricht.
- A3 Alle Tasks in einem Task-Set sind unabhängig, das heißt es gibt keine Vorrangbeziehung und keine Ressourceneinschränkungen.
- A4 Alle Instanzen eines periodischen Tasks  $\tau_i$  haben die gleiche Ausführungszeit  $C_i$ .

Annahme A2 wird für einen einfachen Einstieg getroffen. Im späteren Verlauf werden aber auch Methoden und Algorithmen beschrieben, die nicht dieser Annahme entsprechen. Darauf wird an den entsprechenden Stellen hingewiesen und alternative



Annahmen genannt. Außerdem wird festgelegt, dass alle Tasks in einem Task-Set prä-emptiv sind und auf einem Prozessor geplant werden. Das heißt, dass ein Task jederzeit durch einen anderen Task unterbrochen werden kann und zu einem späteren Zeitpunkt auf diesem Prozessor fortgeführt bzw. vollendet wird.

### 3.1.3 Scheduling

Unter Ablaufplanung (engl. *scheduling*) versteht man das Festlegen einer zeitlichen Reihenfolge, nach welcher die auftretenden Tasks in einem System ausgeführt werden [HSW17]. Im Folgenden werden nur Systeme bestehend aus einem Prozessor (engl. *central processing unit, CPU*) mit einem Prozessorkern betrachtet, ein sogenannter Ein-Kern-Prozessor (engl. *single-core processor*). Die Ablaufreihenfolge kann aber auch für mehrere Ressourcen, das heißt für einen Mehr-Kern-Prozessor (engl. *multi-core processor*) mit entsprechenden Abhängigkeiten und Kommunikationen bestimmt werden. Das Ergebnis der Ablaufplanung ist eine Ablaufreihenfolge bzw. ein Ablaufplan (engl. *schedule*). Ein solcher Ablaufplan heißt brauchbar (engl. *feasible*), wenn alle Zeitbedingungen erfüllt werden, das bedeutet jeder Task vor seiner Deadline beendet wird [Cot+02]. Ein Task-Set ist planbar, wenn ein Scheduling Algorithmus einen brauchbaren Ablaufplan für dieses Task-Set erstellt.

Allgemein besteht eine Scheduling-Methode aus zwei Teilen [BW96]:

- ein Algorithmus, um die Nutzung der Prozessor-Ressourcen, insbesondere der Prozessorkerne, zu organisieren,
- ein Mittel, um das Verhalten des Systems im ungünstigsten Fall abzuschätzen, wenn der Scheduling-Algorithmus angewendet wird.

Die Vorhersage wird dazu benutzt, um die zeitlichen Anforderungen an das System zu bestätigen.

Die zahlreichen Algorithmen, welche für die Planung von Echtzeit-Tasks vorgeschlagen werden, werden nach mehreren Kriterien klassifiziert [But11]:

**offline vs. online** Beim offline Scheduling wird die Ausführungsreihenfolge zur Design- bzw. Compilezeit festgelegt [HSW17]. Alle Tasks, die Ausführungszeiten und Deadlines sind bekannt und werden zur Planung verwendet. Deshalb ergeben sich bei diesen Scheduling-Verfahren meist qualitativ bessere Lösungen als bei online Methoden, bei gleichzeitig geringer Komplexität. Beispiele für offline Scheduling sind ASAP (engl. *as soon as possible*), ALAP (engl. *as late as possible*) und List Scheduling.

Das online Scheduling bestimmt die Reihenfolge der Ausführung zur Laufzeit [HSW17] [But11]. Die Entscheidungen werden immer dann getroffen, wenn ein neuer Task in das System aufgenommen wird, oder ein laufender Task abgeschlossen wird. Zu einem Zeitpunkt  $t$  sind nur diejenigen Tasks bekannt, die zumindest teilweise ausgeführt wurden, aber nicht die zukünftig auftretenden Tasks. Deshalb werden auch

Informationen in die Planung miteinbezogen, die vor der Laufzeit nicht bekannt sind. Beispiele für online Scheduling sind FCFS (engl. *first come first served*), SJF (engl. *shortest job first*), SRTN (engl. *shortest remaining time next*), RR (engl. *round robin*) und EDF (engl. *earliest deadline first*).

**statisch vs. dynamisch** Statische Algorithmen treffen Entscheidungen basierend auf fixen Parametern, die einem Task vor dessen Aktivierung zugeordnet werden [But11]. Das Ablaufplanungsproblem ist vollständig bekannt, das heißt unter anderem die Berechnungszeiten, Anzahl der Tasks und deren Deadlines.

Dynamische Algorithmen treffen Entscheidungen basierend auf dynamischen Parametern, die sich möglicherweise während der Systementwicklung ändern. Zu einem Zeitpunkt sind nur die bis dahin existierenden Tasks bekannt, nicht aber die zukünftig auftretenden Tasks.

**präemptiv vs. nicht-präemptiv** Bei präemptiven Methoden kann ein Task zur Laufzeit jederzeit unterbrochen und zu einem späteren Zeitpunkt auf der gleichen Ressource fortgeführt werden, um zum Beispiel einem höher-priorisierten Task Vorrang zu gewähren [HSW17].

Dagegen untersagen nicht-präemptive Verfahren, dass die Ausführung eines Tasks unterbrochen wird. Die Ausführung des nächsten Tasks kann erst beginnen, wenn der vorherige Task vollständig ausgeführt ist.

**optimal vs. heuristisch** Ein Algorithmus gilt als optimal, wenn die für ein Task-Set gegebene Kostenfunktion minimiert wird. Ist keine Kostenfunktion definiert und das einzige Ziel eine ausführbare Ablaufreihenfolge, wird der Algorithmus optimal genannt, wenn er für jedes planbare Task-Set eine ausführbare Ablaufreihenfolge findet [But11].

Ein Algorithmus gilt als heuristisch, wenn er von einer heuristischen Funktion geleitet wird, um Entscheidungen bezüglich der Ablaufreihenfolge zu treffen. Ein heuristischer Algorithmus tendiert zur optimalen Ablaufreihenfolge, garantiert jedoch nicht, diese zu finden.

In dieser Arbeit werden ausschließlich online und präemptive Scheduling-Algorithmen betrachtet. Das heißt, die Entscheidung, welcher Task als nächstes ausgeführt wird, wird zur Laufzeit getroffen. Die Entscheidung basiert dabei auf Prioritäten, welche den Tasks statisch oder dynamisch zugeordnet werden. Sogenanntes prioritäts-basiertes Scheduling ist Event-gesteuert, das bedeutet die Entscheidungen werden getroffen, wenn bestimmte Events wie das Auslösen oder das Beenden von Jobs auftreten. Andere häufig gebräuchliche Bezeichnungen für diesen Ansatz sind Greed Scheduling und List Scheduling.

Beim Planen nach festen Prioritäten (engl. *fixed priority, FP*) wird jedem Task, genauer gesagt jedem Job, einmalig eine Priorität zugeordnet [Zöb08]. Zu jedem Zeitpunkt wird der Task mit der höchsten Priorität ausgeführt.

Zu dieser Klasse gehört das Planen nach Raten (engl. *rate monotonic*, RM). Die Prioritäten werden nach der Rate eines Tasks vergeben, das heißt je kürzer die Periodendauer eines Tasks, desto höher ist seine Priorität. Es ergibt sich demnach für ein Task-Set mit  $T_1 < \dots < T_n$  die Prioritäten-Reihenfolge  $P_1 > \dots > P_n$ .

Das Planen nach monotonen Fristen (engl. *deadline monotonic*, DM) ist eine Verallgemeinerung des Planens nach monotonen Raten. Die Prioritäten werden gemäß der Task-Deadlines vergeben, das bedeutet je kürzer die Deadline eines Tasks ist, desto höher ist dessen Priorität. Für ein Task-Set mit  $D_1 < \dots < D_n$  ergibt sich daher die Prioritäten-Reihenfolge  $P_1 > \dots > P_n$ .

Wenn  $D = T$  gilt, sind der RM- und DM-Algorithmus identisch [Liu00]. Für Task-Sets mit willkürlichen Deadlines funktioniert der DM-Algorithmus in dem Sinn besser, dass er eine brauchbare Ablaufreihenfolge findet, auch wenn der RM-Algorithmus keine findet. Das bedeutet aber auch, dass der RM-Algorithmus immer dann scheitert, wenn der DM-Algorithmus an einem Task-Set scheitert.

Planungsverfahren mit dynamischen Prioritäten ermöglichen es, dass ein Task  $\tau_i$  einmal von einem Task  $\tau_j$  verdrängt wird, und umgekehrt [Zöb08]. Um dieses Verhalten zu ermöglichen, müssen sich die Prioritäten von Tasks dynamisch zur Laufzeit ändern können.

Eine sehr bekannte Scheduling-Methode ist das Planen nach Fristen (engl. *earliest deadline first*, EDF). Die Taskprioritäten werden gemäß der absoluten Deadlines zugeordnet. Der Task mit der nächsten Deadline wird mit höchster Priorität ausgeführt. Wenn Präemption erlaubt ist und Tasks nicht um Ressourcen konkurrieren, dann erstellt der EDF-Algorithmus einen realisierbaren Ablaufplan für ein Task-Set mit willkürlichen Auslösezeiten und Deadlines auf einem Prozessor, wenn für dieses Task-Set eine realisierbare Ablaufreihenfolge existiert [Liu00]. Das heißt, dass der EDF-Algorithmus für solche Systeme optimal ist.

Eine andere Scheduling-Methode mit dynamischen Prioritäten basiert auf dem Spielraum eines Tasks [Zöb08]. Ein Spielraum  $L_i$  (engl. *laxity*) ist eine Zeitspanne, die zwischen Auslösezeit und Deadline abzüglich der Ausführungszeit ungenutzt bleibt:

$$L_i = D_i - A_i - C_i. \quad (3.1)$$

Beim Planen nach Spielräumen (engl. *least laxity first*, LLF) werden die Prioritäten gemäß des relativen Spielraums zugeordnet. Der Task mit dem kleinsten Spielraum wird mit höchster Priorität ausgeführt. Wenn Präemption erlaubt ist und Tasks nicht um Ressourcen konkurrieren, dann erstellt der LLF-Algorithmus einen realisierbaren Ablaufplan für ein Task-Set mit willkürlichen Auslösezeiten und Deadlines auf einem Prozessor, wenn für dieses Task-Set eine realisierbare Ablaufreihenfolge existiert [Liu00]. Das heißt, der LLF-Algorithmus ist für solche Systeme ebenfalls optimal.

### 3.1.4 Schedulability-Analyse

Ein Planbarkeits-Test (engl. *schedulability test*) oder auch die Planbarkeits-Analyse (engl. *schedulability analysis*) erlaubt die Überprüfung, ob für ein Task-Set mit einem gegebenen Scheduling-Algorithmus ein brauchbarer Ablaufplan erstellt wird. Mit Hilfe der Schedulability-Analyse wird demnach festgelegt, ob ein bestimmter Task oder ein Task-Set erfolgreich auf einem Prozessor abläuft, wenn auf diesem bereits andere Tasks laufen. Diese Analyse wird vor der Ausführung eines Tasks bzw. eines Task-Sets durchgeführt. Allgemein ist ein Task bzw. ein Task-Set planbar, wenn die zeitlichen Anforderungen in Form von Deadlines eingehalten werden.

Ein Schedulability-Test ist notwendig, hinreichend oder exakt. Ein notwendiger (engl. *neccessary*) Test garantiert, dass ein Task-Set nicht planbar ist, wenn der Test negativ ausfällt. Ein hinreichender (engl. *sufficient*) Test garantiert, dass ein Task-Set planbar ist, wenn der Test positiv ausfällt. Ein exakter Test ist notwendig und hinreichend. Wenn der Test negativ ausfällt, ist das Task-Set garantiert nicht planbar, und wenn der Test positiv ausfällt, ist das Task-Set garantiert planbar.

Für den FP- und den EDF-Algorithmus gibt es eine Vielzahl von Analyse- und Testverfahren, die auf unterschiedlichen Prinzipien und Eigenschaften eines Systems aufbauen. Nachfolgend wird ein Überblick über die verschiedenen Ansätze gegeben.

Eine exakte Analyse der Planbarkeit eines Task-Sets ist die Simulation. Dabei wird die Ablaufreihenfolge der Tasks mit Hilfe von Zeitplänen festgelegt und überprüft, ob alle Deadlines eingehalten werden. Die Darstellung der Ablaufreihenfolge ist zwar hilfreich, aber auch sehr rechenintensiv und deshalb nicht für Echtzeitsysteme praktikabel [BW96].

Eine andere Methode der Schedulability-Analyse basiert auf der Prozessorauslastung (engl. *utilization*). Die Prozessorauslastung  $U$  ist der Anteil der Prozessorzeit, die für die Ausführung eines Task-Sets verwendet wird [LL73]. Für  $n$  Tasks lautet die Formel:

$$U = \sum_{i=1}^n \frac{C_i}{T_i}. \quad (3.2)$$

Es gibt einen Maximalwert von  $U$ , unter dem ein Task-Set planbar ist und über dem dieses Task-Set nicht planbar ist. Solch eine Grenze ist vom Task-Set und dem verwendeten Algorithmus abhängig. Wenn die Prozessorauslastung für ein Task-Set dem Maximalwert entspricht, dann nutzt dieses Task-Set den Prozessor vollständig aus. In diesem Fall ist das Task-Set durch den verwendeten Algorithmus planbar, aber jeder Anstieg in der Berechnungszeit eines Tasks wird das Set unausführbar machen.

Für FP-Algorithmen liefert die Antwortzeit-Analyse exakte Ergebnisse über die Planbarkeit eines Task-Sets. Die Antwortzeit eines Tasks ist die maximale Zeit, die zwischen der Auslösezeit eines Tasks und dessen Vollendung vergeht. Damit ein Task planbar ist, darf die Antwortzeit selbst im ungünstigsten Fall niemals länger als die entsprechende Deadline sein. Die Forschung in diesem Bereich konzentriert sich vor allem auf die effektive, schnelle und genaue Berechnung der Antwortzeiten von Tasks.

Auch Tests basierend auf der Arbeitsbelastung stellen die Planbarkeit eines Task-Sets mit festen Prioritäten exakt fest. Die Arbeitsbelastung ist die für die Berechnung bestimmter Tasks notwendige Rechenzeit bezogen auf ein Zeitintervall. Ein Task-Set ist nur planbar, wenn die Arbeitsbelastung geringer als 1 ist, das heißt der Prozessor nicht überbelastet ist.

Für den EDF-Algorithmus gibt es neben der Simulation und dem Prozessorauslastungs-Test eine Analyse basierend auf dem Prozessor-Bedarf. Der Prozessor-Bedarf ist die Menge an Rechenzeit, die von bestimmten Tasks in einem Zeitintervall benötigt werden. Es leisten nur Tasks einen Beitrag, die in diesem Zeitintervall vollständig ausgeführt werden. Außerdem wird, anders als bei der Arbeitsbelastung für den FP-Algorithmus, die resultierende Rechenzeit nicht auf das Zeitintervall bezogen. Ein Task-Set ist planbar, wenn die benötigte Rechenzeit das betrachtete Zeitintervall nicht überschreitet. Das Finden einer oberen Grenze für das zu überprüfende Zeitintervall ist das Ziel der Forschung in diesem Bereich.

## 3.2 Machine Learning, neuronale Netze und Deep Learning

Maschinelles Lernen (engl. *Machine Learning*) ist eine Form der künstlichen Intelligenz (engl. *Artificial Intelligence*) und ein Begriff für das Fachgebiet, das Computern die Fähigkeit aus Daten zu Lernen gibt, anstatt explizit programmiert werden zu müssen [HK18]. Machine Learning umfasst eine Vielzahl von verschiedenen selbstlernenden Algorithmen, die kontinuierlich Informationen aus Daten extrahieren und damit Vorhersagen treffen [HK18; RM17]. Machine Learning wird in allen Wirtschaftsbranchen und Industriezweigen benutzt und ist auch häufig, teils unbewusst, Teil des Alltags: virtuelle Assistenten von Smartphones, Produktempfehlungen in Onlineshops, Spamfilter in E-Mail-Programmen, Text- und Spracherkennung oder Suchmaschinen [RM17].

Damit beispielsweise ein Spamfilter, das heißt ein maschinelles Lernprogramm, Spam-E-Mails von normalen E-Mails unterscheidet, ist ein Training erforderlich [Gér18]. Während diesem Training werden dem Lernalgorithmus verschiedene Beispiele für Spam-E-Mails und gewöhnliche E-Mails gezeigt. Diese Beispiele nennt man auch Trainingsdatensatz. Nach dem Training hat der Lernalgorithmus ein Muster entwickelt, nachdem auch unbekannte E-Mails eingeordnet werden.

### 3.2.1 Arten des Machine Learning

Wie in Abbildung 3.1 zu sehen ist, gibt es verschiedene Arten des Machine Learning, die im folgenden Abschnitt kurz vorgestellt werden.

#### Überwachtes Lernen

Beim überwachten Lernen (engl. *supervised learning*) wird ein Machine Learning Modell anhand gekennzeichneteter Daten trainiert, das heißt die Trainingsdaten bestehen aus

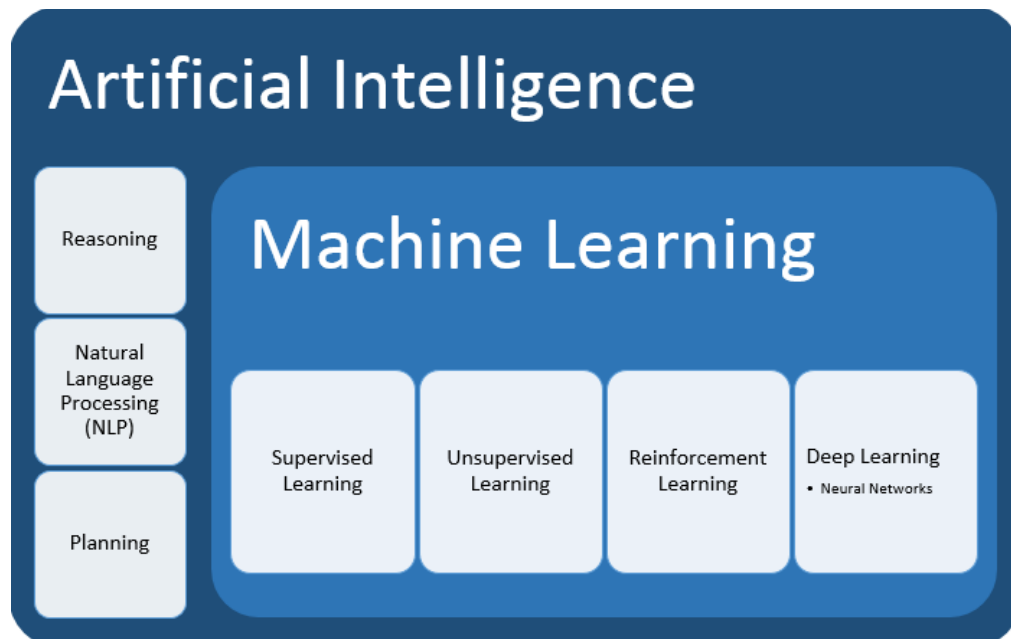


Abbildung 3.1: Überblick über die Teilbereiche der künstlichen Intelligenz und des maschinellen Lernens (Quelle: [HK18, S. 13]).

Eingaben, die auch Merkmale oder Features genannt werden, und der erwünschten Ausgabe, auch als Label bezeichnet [RM17; NZ18]. Ziel des überwachten Lernens ist es, Muster in diesen Trainingsdaten zu finden, um das Modell auch auf unbekannte Daten anzuwenden [HK18].

Eine Unterkategorie des überwachten Lernens ist die Klassifikation (engl. *classification*). Das Ziel der Klassifikation ist die Vorhersage bzw. Bestimmung einer Wahrscheinlichkeit, mit der ein Objekt zu einer Klasse gehört [NZ18]. Wird nur zwischen zwei Klassen unterschieden, wird von binärer Klassifikation gesprochen [RM17]. Ein Beispiel hierfür ist die Zuordnung einer E-Mail zu den Klassen „Spam“ oder „Nicht-Spam“. Die Einteilung kann aber auch in mehrere Klassen erfolgen, die sogenannte Mehrfachklassifikation. Ein Beispiel ist die Handschriften-Erkennung, bei der jedes alphanumerische Zeichen eine Klasse bildet. Die Ausgabe der Klassifikation sind diskrete Werte, genauer eine Wahrscheinlichkeit, mit der ein Objekt zu einer Klasse gehört, oder eine Klassenbezeichnung.

Im Gegensatz dazu steht die Regression, eine weitere Form des überwachten Lernens [NZ18]. Das Ziel der Regression ist das Erlernen einer kontinuierlichen mathematischen Funktion. Die Regression versucht einen Zusammenhang zwischen den Eingabevariablen und den entsprechenden Ausgabewerten, das bedeutet zwischen den Merkmalen und Labels, herzustellen [HK18; RM17]. Ein typisches Beispiel ist die Vorhersage des Preises eines Autos auf Grundlage der gegebenen Merkmale wie gefahrene Kilometer, Alter und Marke [Gér18]. Der einfachste Fall, die lineare Regression, ist ein linearer Zusammenhang, anschaulich gesehen eine Gerade, zwischen den Eingaben und Ausgaben.

Aber auch komplexere Beziehungen können erlernt werden. Die Lösung wird durch eine Hyperfläche beschrieben, eine  $n$ -dimensionale Ebene in einem  $(n+1)$ -dimensionalen Merkmals-Raum [NZ18]. Die logistische Regression kann auch zur Klassifikation eingesetzt werden, um die Wahrscheinlichkeit der Zugehörigkeit zu einer bestimmten Klasse zu bestimmen [Gér18].

Die bekanntesten und wichtigsten Lernalgorithmen des überwachten Lernens sind k-nächste-Nachbarn, lineare Regression, logistische Regression, Support Vector Machines, Entscheidungsbäume und Random Forests sowie neuronale Netze [Gér18].

### Unüberwachtes Lernen

Unüberwachtes Lernen (engl. *unsupervised learning*) wird bei sehr großen Datenmengen angewendet, die nicht gekennzeichnet sind und deren Struktur unbekannt ist [HK18; RM17]. Um die Bedeutung hinter den Daten zu verstehen, muss ein Algorithmus fähig sein, die Daten anhand von Mustern und Gruppen zu klassifizieren. Hierfür werden die Daten beim unüberwachten Lernen iterativ ohne menschliches Eingreifen analysiert.

Eine Disziplin des unüberwachten Lernens ist das Clustering. Dabei werden die Daten ohne vorherige Kenntnis über die Gruppenzugehörigkeit in zusammenhängende Gruppen, sogenannte Cluster, mit ähnlichen Eigenschaften sortiert [RM17; NZ18]. Beispiele für solche Lernalgorithmen sind k-Means, hierarchische Clusteranalyse und Expectation Maximization [Gér18].

Eine andere Form des unüberwachten Lernens ist die Dimensionsreduktion, auch Hauptkomponentenanalyse (engl. *Principal Component Analysis, PCA*) genannt [RM17; NZ18]. Umfangreiche Datensätze werden vereinfacht, indem die Dimensionen, das ist die Anzahl der Eingabevariablen, mit möglichst wenig Informationsverlust reduziert werden. Die redundanten Informationen werden auch „Rauschen“ genannt. Die Dimensionsreduktion wird als Komprimierungsverfahren angesehen. Wichtige Lernalgorithmen sind die bereits erwähnte Hauptkomponentenzerlegung (PCA), Kernel PCA, Locally-Linear Embedding und t-verteilter Stochastic Neighbor Embedding [Gér18].

Soll eine sehr große Datenmenge analysiert werden, ist dies von Hand meist nicht möglich [RM17; NZ18]. Die Algorithmen des unüberwachten Lernens helfen hier, die Informationen zu strukturieren und sinnvolle Beziehungen zwischen den Daten abzuleiten. Durch Clustering werden zum Beispiel auch Ausgabewerte bestimmt. Auf diese Weise können die gekennzeichneten Daten an einen Algorithmus des überwachten Lernens weitergegeben werden. Ebenso wird die Dimensionsreduktion zur Datenvorverarbeitung beim überwachten Lernen verwendet, um große Datensätze zu komprimieren.

### Verstärkendes Lernen

Beim verstärkenden oder bestärkenden Lernen (engl. *reinforcement learning*) wird ein System nicht mit Beispieldaten trainiert, sondern es erlernt anhand positiver oder negativer Rückmeldung auf eine ausgeführte Aktion ein optimales Verhalten innerhalb einer bestimmten Situation [NZ18; HK18]. Häufig verwendet um interaktive Aufgaben

zu lösen, findet das Lernen über Ausprobieren (Versuch und Irrtum, engl. *trial and error*) oder bewusste Planung statt [RM17]. Der Lernerfolg basiert auf einer Folge von richtigen Entscheidungen, die mit einer positiven Rückmeldung durch die Umgebung belohnt wird. Ebenso wird eine falsche Entscheidung mit einem negativen Belohnungssignal bestraft. Ein bekanntes Beispiel für verstärkendes Lernen sind Schachcomputer [RM17]. Ein richtiger Zug wird mit dem Schlagen einer Figur belohnt, oder eine ganze Folge an richtigen Zügen mit dem Sieg. Dagegen führt ein falscher Zug zum Verlieren einer Figur oder des gesamten Spiels.

#### Neuronale Netze und Deep Learning

Deep Learning ist eine Methode des maschinellen Lernens, bei der neuronale Netze mit mehreren Schichten verwendet werden, um iterativ Muster aus unstrukturierten, nicht-gekennzeichneten Daten zu lernen [HK18]. Die komplexen neuronalen Netze des Deep Learnings ahmen das menschliche Gehirn und dessen Funktionsweise nach. Häufige Anwendungsbereiche sind Bild- und Spracherkennung sowie Computer Vision (maschinelles Sehen, Bildverstehen).

Ein neuronales Netz besteht aus drei oder mehr Schichten: eine Eingabeschicht (engl. *input layer*), eine oder mehrere versteckte Schicht(en) (engl. *hidden layers*) und eine Ausgabeschicht (engl. *output layer*) [HK18]. Ein Beispiel für ein solches neuronales Netz zeigt Abbildung 3.2. Die Daten werden durch die Eingabeschicht aufgenommen. In den versteckten Schichten und der Ausgabeschicht werden die Daten basierend auf bestimmten Gewichtungen verändert. Ein typisches neuronales Netzwerk besteht aus Tausenden oder sogar Millionen von einfachen, dicht miteinander verbundenen Verarbeitungsknoten, den sogenannten Neuronen. Der Begriff „Deep Learning“ wird verwendet, wenn ein neuronales Netz aus vielen versteckten Schichten besteht. Je komplexer ein Problem, desto mehr versteckte Schichten werden benötigt. In einem iterativen Prozess wird das neuronale Netz kontinuierlich angepasst und Folgerungen gezogen, bis ein Haltepunkt erreicht ist.

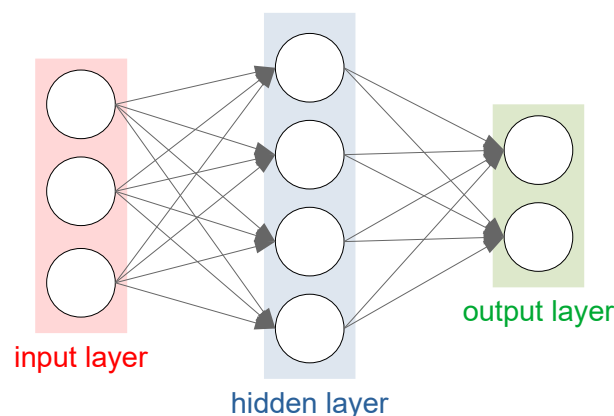


Abbildung 3.2: Allgemeine Architektur eines neuronalen Netzes (Quelle: [HK18, S. 31]).



### 3.2.2 Herausforderungen beim Machine Learning

Die Hauptaufgabe beim maschinellen Lernen ist die Auswahl eines Lernalgorithmus, um diesen mit entsprechenden Daten zu trainieren [Gér18]. Darin liegen aber auch die möglichen Fehlerquellen begründet, nämlich ein falscher Algorithmus oder schlechte Daten. Die Auswahl eines Lernalgorithmus ist vom jeweiligen Anwendungsfall abhängig und nicht verallgemeinert festgelegt. In jedem Fall ist das Ausprobieren verschiedener Verfahren und Einstellungen ratsam, wenn nicht sogar notwendig.

Damit ein maschinelles Lernverfahren etwas lernt, ist eine sehr große Datenmenge erforderlich [Gér18]. Selbst bei einfachen Aufgaben sind üblicherweise Tausende von Beispielen notwendig, bei komplexeren Aufgaben sogar Millionen Beispiele. Damit das erlernte Modell auch im Allgemeinen gültig ist, müssen die Beispieldaten die zu verallgemeinernde Situation repräsentieren. Zusätzlich werden bei der Datenvorverarbeitung die Daten in Form gebracht, informative Merkmale werden extrahiert und redundante oder irrelevante Merkmale entfernt [RM17]. Gut vorbereitete Daten helfen dabei eine Überanpassung (engl. *overfitting*) des Modells an die Trainingsdaten zu verhindern. Bei Overfitting funktioniert das Modell mit den Trainingsdaten sehr gut, es verallgemeinert allerdings nicht gut und versagt deshalb bei neuen, unbekannten Daten [Gér18; HK18; NZ18]. Das Gegenteil von Overfitting ist Unteranpassung (engl. *underfitting*) [Gér18]: das Modell ist zu einfach um die Struktur in den Trainingsdaten zu erlernen und versagt bereits bei diesen.

Wenn ein Modell mit einem Lernalgorithmus trainiert wird, ist es wichtig, eine Evaluation durchzuführen und die Verallgemeinerung zu überprüfen [Gér18]. Da das Modell mit den Beispieldaten lernt, sind diese nicht zum Testen verwendbar. Deshalb wird typischerweise die verfügbare Datenmenge in zwei Datensätze aufgeteilt: den Trainingsdatensatz und den Testdatensatz. Mit den Trainingsdaten erlernt das Modell ein Muster, mit den Testdaten wird das Modell bewertet [HK18; RM17]. Üblicherweise werden 80 % der verfügbaren Daten als Trainingsbeispiele und 20 % als Testdaten verwendet. Mit den Testbeispielen wird so festgestellt, ob das Modell die notwendige Allgemeingültigkeit erlernt hat, um auch mit neuen, unbekannten Daten umgehen zu können [HK18].

Häufig ist es notwendig, verschiedene Lernverfahren miteinander zu vergleichen, da für eine bestimmte Anwendung nicht vorhersagbar ist, welcher Lernalgorithmus anderen überlegen ist [RM17]. Deshalb wird noch ein weiterer Datensatz, der Validierungsdatensatz, benötigt, um die Ergebnisse der verschiedenen Lernalgorithmen miteinander zu vergleichen, wie Abbildung 3.3 darstellt. Die verschiedenen Verfahren werden alle mit dem gleichen Trainingsdatensatz trainiert und der beste Algorithmus wird mit Hilfe des Validierungsdatensatzes ausgewählt. Erst danach wird ein abschließender Test mit dem Testdatensatz durchgeführt, um die Verallgemeinerung des Modells zu überprüfen [Gér18].

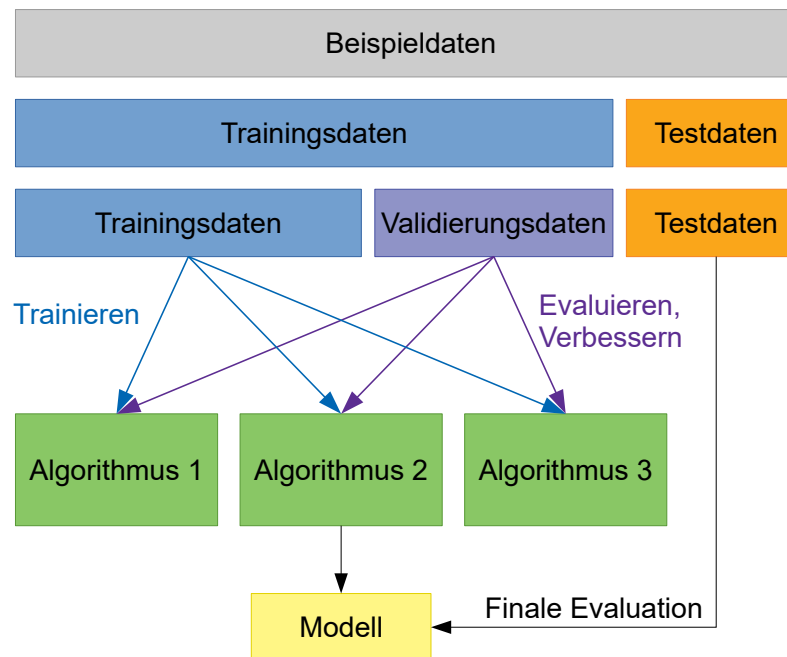


Abbildung 3.3: Aufteilung der Beispieldaten in verschiedene Datensätze und deren Verwendung (In Anlehnung an [Tan18]).

### 3.2.3 Grundlagen neuronaler Netze

Die Idee von künstlichen neuronalen Netzen (engl. *artificial neural networks*) ist es, das menschliche Gehirn nachzuahmen und auf diese Weise eine künstliche Intelligenz zu entwickeln. Biologisch betrachtet ist die Aufgabe des Gehirns die Aufnahme, Verarbeitung und Beantwortung von Reizen [Kin94]. Reize werden über Sensoren, die Rezeptoren, aufgenommen und als elektrochemische Signale über die Nervenbahnen an das Gehirn weitergeleitet. Dort werden diese Signale nach vorgegebenen Mustern verarbeitet und, falls notwendig, wird ein Reaktionsreiz an Gewebeteile wie Muskeln, die Effektoren, abgegeben. Das Gehirn, genauer die Hirnrinde (Neokortex), bildet in diesem System der Reizverarbeitung die wichtigste Schaltstelle. Die elementaren Verarbeitungseinheiten der Hirnrinde sind Neuronen, Zellen die auf elektrochemischem Weg Signale austauschen und sich gegenseitig erregen. Ein Mensch besitzt etwa  $10^{10}$  Neuronen mit  $10^{14}$  Verbindungen dazwischen, die zusammen ein neuronales Netz bilden. Das Reaktionsverhalten auf Reize wird nach und nach erlernt und ist in den Verbindungsmustern zwischen den Neuronen gespeichert. Ein Neuron wird auch als Prozessor des Gehirns bezeichnet und hat Ein- und Ausgabesignale. Die Eingabesignale sind elektrochemische Reize, die von anderen Neuronen über die Dendriten aufgenommen werden. Ist die Summe der Eingangssignale, das bedeutet das elektrische Potential, größer als ein Schwellenwert, „feuert“ das Neuron. Das bedeutet, das Neuron wird aktiv und sendet über das Axon einen kurzen Impuls an nachgeschaltete Neuronen. Der Aufbau eines biologischen Neurons ist in Abbildung 3.4 dargestellt. Zwischen der Eingabe (Dendriten) und dem

Prozessor (Neuron) befinden sich Synapsen, die ankommende Potentialwerte verstärken oder hemmen. Die Synapsen spielen eine wichtige Rolle im Lernverhalten, da sie ihre verstärkende oder hemmende Wirkung verändern.

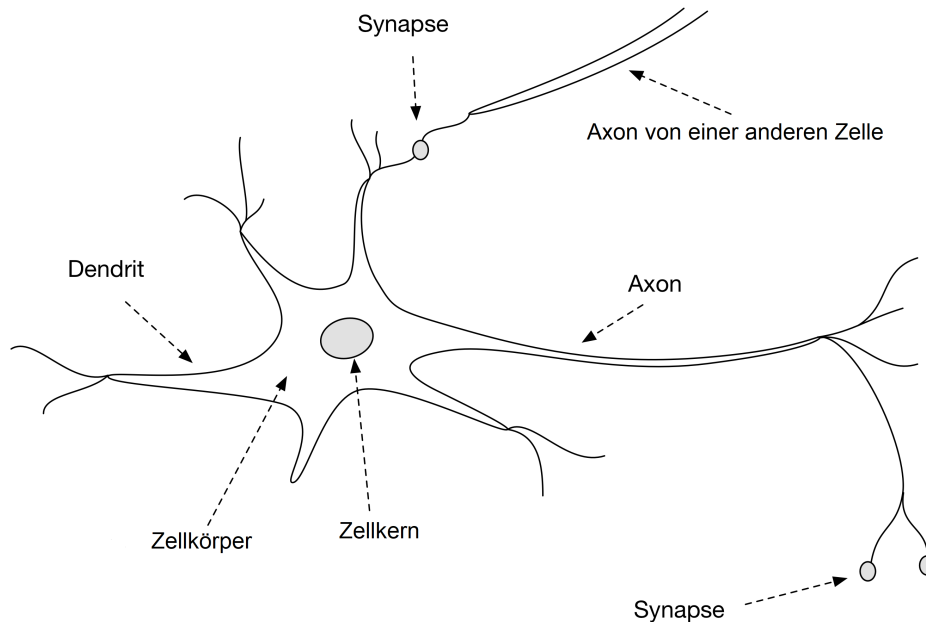


Abbildung 3.4: Aufbau eines biologischen Neurons: Dendriten (Eingabe), Neuron (Prozessor) und Axon (Ausgabe) (aus dem Englischen nach [GP17, Abb. 2-2]).

### Das künstliche Neuron

Ein künstliches Neuron wird als Prozessor mit zwei Zuständen betrachtet, die durch die binären Werte 0 für „Neuron feuert“ und 1 für „Neuron passiv“ repräsentiert werden [Kin94]. Abbildung 3.5 zeigt den Aufbau eines künstlichen Neurons. Die Eingabewerte  $e_1, \dots, e_n$  eines künstlichen Neurons sind ebenfalls entweder 0 oder 1, da sie Ausgaben vorgeschalteter Neuronen sind. Die Synapsen werden durch Multiplikationen mit positiven (Verstärkung) oder negativen (Hemmung) reellen Zahlen  $w_1, \dots, w_n$  (Gewichte) simuliert. Um den Ausgangswert  $a$  eines künstlichen Neurons zu bestimmen, wird die Summe aller mit den Synapsenwerten multiplizierter Eingänge gebildet (Nettoeingabe  $net$ ) und mit einem Schwellenwert  $\theta$  verglichen:

$$net = \sum_{i=1}^n w_i \cdot e_i \quad (3.3)$$

$$a = \begin{cases} 0 & \text{falls } net \leq \theta \\ 1 & \text{falls } net > \theta. \end{cases} \quad (3.4)$$

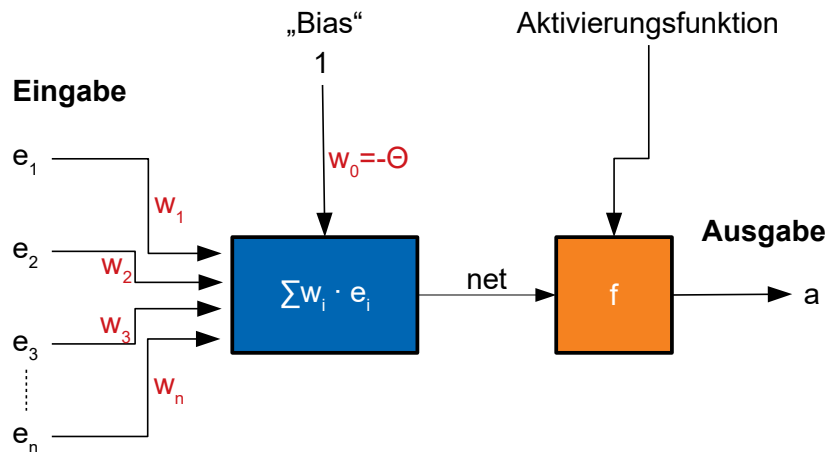


Abbildung 3.5: Mathematische Darstellung eines künstlichen Neurons (In Anlehnung an [NZ18, Abb. 7-2]).

Zur Vereinfachung wird der Schwellenwert  $\theta$  häufig als Gewicht  $w_0 = -\theta$  gezählt und eine zusätzliche Eingabe mit dem konstanten Wert 1 hinzugefügt. Diese Zusatzeingabe wird als Bias bezeichnet und ermöglicht die Einbringung des Schwellenwerts in die Summe und einen normierten Vergleich der Summe mit der Zahl 0:

$$net = \sum_{i=1}^n w_i \cdot e_i - \theta \quad (3.5)$$

$$a = f(net) = \begin{cases} 0 & \text{falls } net \leq 0 \\ 1 & \text{falls } net > 0. \end{cases} \quad (3.6)$$

Die Funktion  $f$  wird Aktivierungsfunktion (engl. *activation function*) genannt. Durch den Einsatz verschiedener Aktivierungsfunktionen lassen sich unterschiedliche Neuronen bilden. Häufig verwendete Aktivierungsfunktionen sind die Sprungfunktion und die sigmoide Funktion (siehe Abbildung 3.6):

$$f(x) = \begin{cases} 0 & \text{falls } x \leq 0 \\ 1 & \text{falls } x > 0 \end{cases} \quad \text{Sprungfunktion} \quad (3.7)$$

$$f(x) = \frac{1}{1 + e^{-c \cdot x}} \quad \text{sigmoide Funktion} \quad (3.8)$$

Bei der sigmoiden Funktion gibt die Konstante  $c$  die Steilheit des Übergangs von 0 nach 1 an. Im Gegensatz zur Sprungfunktion ist die sigmoide Funktion stetig differenzierbar.

### Das künstliche neuronale Netz

Ein neuronales Netz besteht aus mehreren, hinter- oder nebeneinander geschalteten künstlichen Neuronen [Kin94]. Abbildung 3.7 zeigt als Beispiel ein künstliches neuronales Netz bestehend aus drei Neuronen mit den Eingabewerten  $e_1$  und  $e_2$ , den

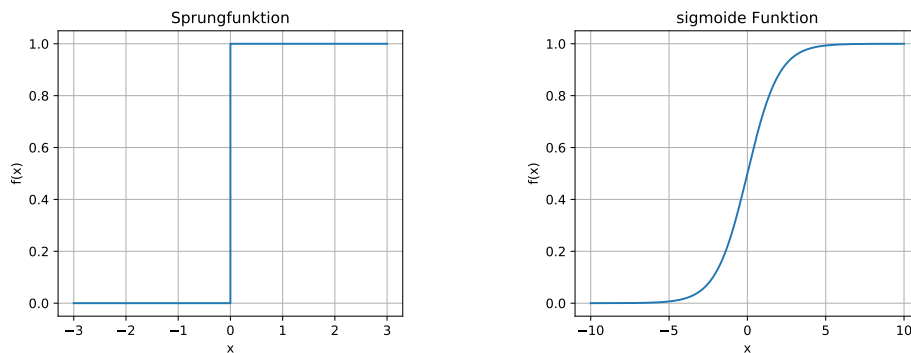


Abbildung 3.6: Aktivierungsfunktionen: die Sprungfunktion und die sigmoide Funktion ( $c = 1$ ) (Eigene Darstellung).

Ausgabewerten  $a_1, a_2$  und  $a_3$  sowie einem Bias. Die Gewichte sind in rot an die jeweilige Verbindung geschrieben. Zur vereinfachten Darstellung wird der Eingangsvektor  $\vec{e} = [1, e_1, e_2]$ , der auch den Bias enthält, und der Ausgangsvektor  $\vec{a} = [a_1, a_2, a_3]$  eingeführt. Die Gewichte bilden eine Gewichtsmatrix

$$W = \begin{pmatrix} w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \\ w_{03} & w_{13} & w_{23} \end{pmatrix} = \begin{pmatrix} -1 & -1 & 2 \\ -2 & 3 & -1 \\ 1 & 2 & 1 \end{pmatrix}$$

und die Berechnungsvorschrift lautet damit:

$$\vec{net} = W \cdot \vec{e} = \begin{pmatrix} -1 & -1 & 2 \\ -2 & 3 & -1 \\ 1 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ e_1 \\ e_2 \end{pmatrix}$$

$$\vec{a} = f(\vec{net}).$$

Da jedem Eingangsvektor  $\vec{e}$  ein Ausgangsvektor  $\vec{a}$  zugeordnet wird, gibt das Netz eine logische Funktion vor.

### Die Hebb'sche Lernregel

Wie gezeigt stellen neuronale Netze logische Funktionen dar [Kin94]. Um eine Funktion durch ein Netz abzubilden, ist eine Vorschrift zur Berechnung der Gewichte notwendig. Angelehnt an das biologische Vorbild werden die optimalen Gewichte mit Hilfe eines Lernverfahrens erlernt. Die Hebb'sche Lernregel ist benannt nach dem Psychologen Donald Hebb, der 1949 die folgende Hypothese aufstellte [Kin94, S. 23]:

„Die synaptische Eigenschaft (Verstärken oder Hemmen) ändert sich proportional zum Produkt von prä- und postsynaptischer Aktivität.“

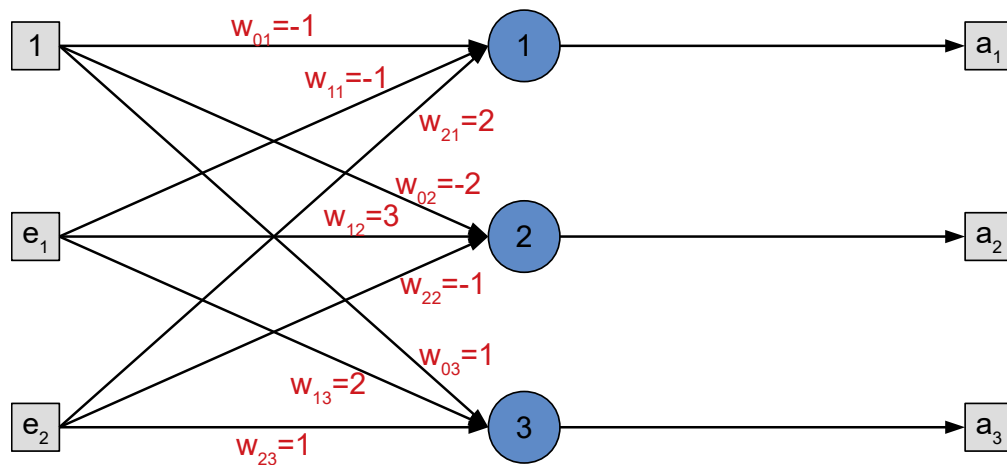


Abbildung 3.7: Beispiel für ein einfaches neuronales Netz aus drei Neuronen (In Anlehnung an [Kin94, S. 21]).

Laut Hebb erfolgt das Lernen durch eine iterative Änderung der Synapsenstärken. Dieser Vorgang ist auch auf das Lernverhalten eines künstlichen neuronalen Netzes anwendbar. Die Änderung des Gewichtes  $w_{ij}$  vom Eingabewert  $e_i$  zum Neuron  $j$  ändert sich um den Wert:

$$\Delta w_{ij} = \eta \cdot e_i \cdot a_j \quad \text{Hebb'sche Regel} \quad (3.9)$$

Der Faktor  $\eta$  ist eine Konstante und wird als Lernrate bezeichnet. Üblicherweise wird aber eine etwas abgewandelte Form der Hebb'schen Regel angewendet, die Delta-Regel:

$$\Delta w_{ij} = \eta \cdot e_i \cdot \Delta a_j \quad \text{Delta-Regel} \quad (3.10)$$

Statt des Ausgabewertes  $a_j$  wird die Differenz zwischen dem erwünschten Output und der erzielten Ausgabe verwendet, sodass sich das Gewicht  $w_{ij}$  nicht ändert, wenn die erzielte Ausgabe mit der gewünschten übereinstimmt.

### 3.2.4 Vorwärts gerichtete Netze

Bei den künstlichen neuronalen Netzen werden zwei Arten unterschieden: Netze mit Rückkopplung und vorwärts gerichtete Netze [Kin94]. Netze mit Rückkopplung führen Ausgabewerte auf die Eingabe zurück und werden im nachfolgenden Kapitel 3.2.6 erörtert. Vorwärts gerichtete Netze (engl. *feed forward networks*) bestimmen für jeden Eingabevektor einen Ausgabevektor ohne Rückkopplung. In diesem Sinne wird der Informationsfluss in solchen Netzen als vorwärts gerichtet betrachtet.

### Netzwerk-Topologien

Wie bereits im vorherigen Kapitel 3.2.3 gezeigt, lässt sich ein künstliches neuronales Netz mit einer Gewichtsmatrix  $W$  folgendermaßen beschreiben [Kin94]:

$$\vec{a} = f(\vec{net}) \quad \text{Ausgabewert} \quad (3.11)$$

$$\vec{net} = W \cdot \vec{e} \quad \text{Nettoeingabe} \quad (3.12)$$

wobei  $\vec{e}$  und  $\vec{a}$  die Eingabe- bzw. Ausgabevektoren sind und  $f$  die Aktivierungsfunktion ist. Mit einem solchen Netz lassen sich allerdings nicht alle logischen Funktionen beschreiben. So ist es zum Beispiel nicht möglich, die bool'sche XOR-Funktion darzustellen. Die Lösung für dieses Problem sind mehrstufige Netze, bei denen die Ausgabe als Eingabe für eine weitere Neuronenschicht dient. Mit mehrschichtigen Netzen sind alle Funktionen beschreibbar.

Ein mehrschichtiges neuronales Netz besteht aus mindestens drei Schichten: einer Eingabeschicht, versteckten Schichten und einer Ausgabeschicht [Kin94; GP17]. Die Eingabeschicht (engl. *input layer*) dient dazu, die Eingabewerte aufzunehmen. Die Anzahl der Neuronen in der Eingabeschicht entspricht aus diesem Grund der Anzahl der Eingangswerte. Die Neuronen der Eingangsschicht leiten die Eingangswerte unverändert an alle Neuronen der ersten versteckten Schicht weiter. Diese Form der Vernetzung wird auch vollständig verbunden (engl. *fully connected*) genannt. Auf die Eingabeschicht folgen eine oder mehrere versteckte Schichten (engl. *hidden layers*) [GP17]. Die Anzahl der versteckten Neuronen (engl. *hidden neurons*) ist unabhängig von der Anzahl der Eingangs- oder Ausgangswerte und basiert hauptsächlich auf der Komplexität der zu beschreibenden Funktion. Je komplizierter eine Funktion, desto mehr versteckte Schichten und desto mehr versteckte Neuronen sind notwendig. Die Ausgabeschicht (engl. *output layer*) gibt schließlich die berechneten Ausgabewerte des Netzes aus.

Abbildung 3.8 zeigt als Beispiel ein dreischichtiges neuronales Netz mit je zwei versteckten Neuronen und Bias-Werten. Es gibt zwei Eingabewerte  $e_1$  und  $e_2$ , sowie zwei Ausgabewerte  $a_1$  und  $a_2$ . Die versteckte Schicht speichert den Hilfsvektor  $\vec{h} = [1, h_1, h_2]$ . Dieser Hilfsvektor enthält einen Bias und die Ausgabewerte der ersten Stufe bzw. der ersten versteckten Schicht und wird bestimmt durch:

$$\vec{h} = f(\vec{net}_1) = f(W_1 \cdot \vec{e}), \quad (3.13)$$

$$\begin{pmatrix} 1 \\ h_1 \\ h_2 \end{pmatrix} = f \left[ \begin{pmatrix} 1 & 0 & 0 \\ -1 & 2 & -2 \\ -1 & -2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ e_1 \\ e_2 \end{pmatrix} \right].$$

Der Ausgabevektor ergibt sich folgendermaßen:

$$\vec{a} = f(\vec{net}_2) = f(W_2 \cdot \vec{h}), \quad (3.14)$$

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = f \left[ \begin{pmatrix} 2 & -3 & 0 \\ -2 & 3 & 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ h_1 \\ h_2 \end{pmatrix} \right].$$

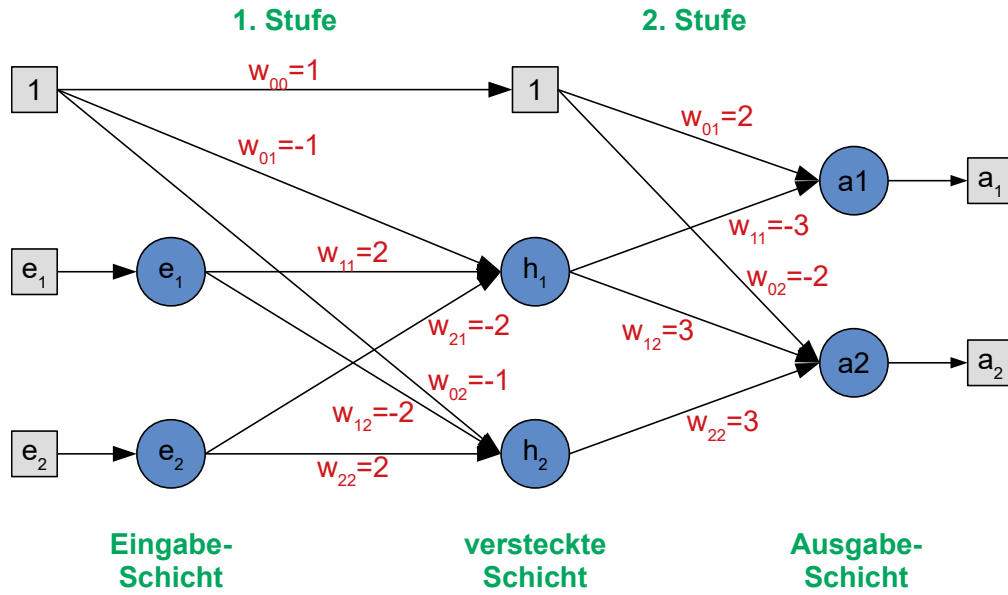


Abbildung 3.8: Beispiel für ein mehrschichtiges neuronales Netz mit drei Schichten und je zwei versteckten Neuronen (In Anlehnung an [Kin94, S. 27]).

Zusammengefasst lautet die Beschreibung des gesamten neuronalen Netzes:

$$\vec{a} = f(W_2 \cdot f(W_1 \cdot \vec{e})). \quad (3.15)$$

Da ein neuronales Netz aus beliebig vielen Schichten bestehen kann, wird das eben gezeigte Beispiel nachfolgend verallgemeinert. Gegeben seien ein Eingabevektor  $\vec{e}$ , ein Ausgabevektor  $\vec{a}$  und Hilfsvektoren  $\vec{h}_1, \vec{h}_2, \dots$ , die die verborgenen Schichten bilden. Außerdem seien  $f$  die Aktivierungsfunktion und  $W_1, W_2, W_3, \dots$  die Gewichtsmatrizen. Die Ausgabe des n-stufigen neuronalen Netzes berechnet sich durch:

$$\begin{aligned} \vec{h}_1 &= f(W_1 \cdot \vec{e}), \\ \vec{h}_2 &= f(W_2 \cdot \vec{h}_1), \\ \vec{h}_3 &= f(W_3 \cdot \vec{h}_2), \\ &\vdots \\ \vec{a} &= f(W_n \cdot \vec{h}_{n-1}) = f(W_n \cdot \dots \cdot f(W_3 \cdot f(W_2 \cdot f(W_1 \cdot \vec{e}))))). \end{aligned} \quad (3.16)$$

### Lernende Systeme

Bevor ein neuronales Netz in einer praktischen Anwendung einsetzbar ist, sind einige Arbeitsschritte notwendig [Kin94]. Allgemein hat sich folgender Arbeitsablauf etabliert:

1. Voraussetzung: Angabe einer logischen Funktion, die durch das Netz dargestellt werden soll, das heißt Eingabevektoren  $\vec{e}_1, \vec{e}_2, \vec{e}_3, \dots, \vec{e}_n$  mit den zugeordneten Ausgabevektoren  $\vec{a}_1, \vec{a}_2, \vec{a}_3, \dots, \vec{a}_n$ .



2. Wahl einer Topologie für das Netz.
3. Bestimmung der Gewichte  $w_1, w_2, w_3, \dots$ , sodass das Netz mit der gewählten Topologie die vorgegebene logische Funktion darstellt. Zur Bestimmung der Gewichte werden Lernverfahren angewendet.
4. Nach dem Lernen der Gewichte steht das Netz zur Verfügung und ist beliebig oft einsetzbar.

Damit ergeben sich zwei wichtige Phasen für die Bearbeitung und den Einsatz von neuronalen Netzen: die Lernphase (Lernen der Gewichte) und die Einsatzphase (Recall-Phase). Die Lernphase ist erfahrungsgemäß sehr rechenintensiv, muss aber auch nur einmal durchlaufen werden. Die Einsatzphase hat einen geringen Rechenaufwand und wird beliebig oft durchlaufen.

Das Lernen der Gewichte während der Lernphase erfolgt nach folgendem Algorithmus:

1. Initialisierung aller Gewichte mit Zufallszahlen.
2. Wahl eines zufälligen Inputvektors  $\vec{e}_j$ .
3. Berechnung des Outputvektors  $\vec{o}_j$  mit den momentanen Gewichten.
4. Vergleich von  $\vec{o}_j$  mit dem Zielvektor  $\vec{a}_j$ . Wenn  $\vec{o}_j = \vec{a}_j$ , dann weiter bei [2.]. Andernfalls Verbesserung der Gewichte nach einer geeigneten Korrekturformel und weiter bei [2].

Mögliche Korrekturformeln in [4.] sind zum Beispiel die Hebb'sche Lernregel aus Gleichung 3.9 oder die Delta-Regel aus Gleichung 3.10. Das gezeigte Lernverfahren benötigt zu jedem Inputvektor  $\vec{e}$  einen exakten Outputvektor  $\vec{a}$ , der mit dem berechneten Vektor  $\vec{o}$  verglichen wird. Die Gewichte werden nur nach einer Lernformel verändert, wenn  $\vec{o} \neq \vec{a}$ . Damit gehört das Verfahren zum überwachten Lernen. Es gibt auch neuronale Netze, die durch Lernmethoden des unüberwachten oder verstärkenden Lernens entwickelt werden. Dabei werden die Gewichte ohne Kenntnis der exakten Zielwerte zufällig verändert und eine Bewertungsfunktion beurteilt das Ergebnis. Nur wenn der berechnete Output besser ist, werden die veränderten Gewichte gespeichert. Eine dritte Lernmethode ist das Lernen durch Selbstorganisation. Die Gewichte ändern sich dabei in jedem Lernschritt, wobei die Änderung von den Nachbarschaften der Eingabemuster und der Wahrscheinlichkeitsverteilung der zulässigen Eingabemuster beim Lernen abhängig ist. Für die Anwendung von neuronalen Netzwerken in dieser Arbeit ist nur das überwachte Lernen von Bedeutung.

### Das Perzeptron

Das Perzeptron, vorgestellt von F. Rosenblatt im Jahr 1958, ist ein einstufiges lernfähiges Netz, wie das Beispiel in Abbildung 3.7 [Kin94]. Die Eingabe- und Ausgabewerte sind binäre Werte, im Allgemeinen die Zahlen 0 und 1. Der Lernalgorithmus lautet:

1. Initialisierung der Gewichte  $w_{ij}$  mit Zufallszahlen.
2. Auswahl eines zufälligen Eingabevektors  $\vec{e}_i$ , kurz  $\vec{e} = [e_1, e_2, \dots, e_n]$  wobei  $e_k \in \{0, 1\}$ .
3. Veränderung der Gewichte durch

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \quad (3.17)$$

mit

$$\Delta w_{ij} = \eta \cdot e_j \cdot \epsilon_i \quad (3.18)$$

und

$$\epsilon_i = a_i - f\left(\sum_k w_{ik} \cdot e_k\right) = a_i - f(net_i). \quad (3.19)$$

Dabei ist  $\epsilon_i$  der Unterschied zwischen Ziel-Output und Ist-Output an der Stelle  $i$ , und  $\eta > 0$  eine kleine Zahl, die auch als Lernrate bezeichnet wird.

4. Weiter mit [2].

Der Algorithmus läuft solange, bis das Netz für alle Eingangsvektoren die richtigen Zielvektoren ausgibt. Der Lernalgorithmus konvergiert in endlich vielen Schritten, falls es für die zu lernende Funktion eine Lösung in Form eines neuronalen Netzes gibt. Allerdings gibt es Funktionen, die nicht mit dem Perzeptron darstellbar sind, so zum Beispiel die bool'sche XOR-Funktion.

#### Das Multi-Layer Perzeptron

Einstufige neuronale Netze sind teils erheblich eingeschränkt, da mit ihnen bereits einfache Funktionen nicht darstellbar sind [Kin94]. Die Lösung hierfür sind mehrstufige Netze wie das Multi-Layer Perzeptron. Dieses Netzwerk, auch Mehrschichten-Netzwerk genannt, ist ein mehrstufiges neuronales Netz mit diversen versteckten Schichten. Jedes Neuron einer Schicht kann mit jedem Neuron der folgenden Schicht verbunden sein. Zudem ist in jeder Schicht ein Bias möglich. Als Aktivierungsfunktion wird häufig die sigmoide Funktion aus Gleichung 3.8 angewandt, da sie differenzierbar ist. Die Differenzierbarkeit der Aktivierungsfunktion ist eine wichtige Voraussetzung für ein geeignetes Lernverfahren, damit die im nachfolgenden Unterkapitel vorgestellte Fehlerrückführungs-Methode zur Anpassung der Gewichte möglich ist. Mit dem Multi-Layer Perzeptron ist jede logische Funktion darstellbar. Gleichzeitig ist diese Art von Netzwerk durch eine hohe Leistungsfähigkeit geprägt. Außerdem existiert ein vorteilhaft einsetzbares Lernverfahren, nämlich der Backpropagation-Algorithmus.

#### Die Backpropagation-Methode

Die Backpropagation-Methode oder Fehlerrückführungsmethode ist die wirksamste und meist verwendete Lernmethode beim Trainieren von mehrschichtigen neuronalen Netzen [Kin94]. Sie ist eine Gradienten-Abstiegsmethode und wurde erstmals von

Hinten, Rumelhart und Williams im Jahr 1986 beschrieben. Im nachfolgenden wird die Methode anhand eines zweistufigen Netzes erklärt, ist formal aber auf n-schichtige Netze mit  $n > 3$  übertragbar.

Gegeben sei ein neuronales Netz mit dem Eingabevektor  $\vec{e}$ , einer versteckten Schicht  $\vec{h}$  und der Ausgabe  $\vec{a}$ , wie in Abbildung 3.8. Der Zusammenhang zwischen den Vektoren aus den Gleichungen 3.13 und 3.11 ist:

$$\vec{a} = f(W_1 \cdot \vec{h}) \quad \text{Ausgabe} \quad (3.20)$$

$$\vec{h} = f(W_2 \cdot \vec{e}) \quad \text{versteckte Schicht} \quad (3.21)$$

Als Aktivierungsfunktion wird die sigmoide Funktion aus Gleichung 3.8 mit  $c > 0$  angewandt. Formal entspricht das Vorgehen dem beim Perzeptron: Die Gewichte werden so bestimmt, dass eine Fehlerfunktion minimiert wird. Die verwendete Fehlerfunktion (engl. *loss function*) lautet hier:

$$E = \frac{1}{2} \cdot \sum_i (z_i - a_i)^2. \quad (3.22)$$

Dabei ist  $z_i$  der vorgegebene Zielwert der zu lernenden Funktion und  $a_i$  der Ist-Wert, das heißt die Ausgabe des Netzes. Das bedeutet, die Fehlerfunktion bestimmt die Summe der quadratischen Fehler. Deshalb arbeitet das Netz umso besser, je kleiner  $E$  ist. Für  $E = 0$  bildet das neuronale Netz die gewünschte Funktion exakt ab. Das Lernverfahren sollte die Gewichte so verändern, dass  $E$  in jedem Schritt kleiner wird, das Verfahren folglich auf ein Minimum hinarbeitet. Wenn  $w_{ij}^1$  die Elemente der Matrix  $W_1$  aus Gleichung 3.20 und  $w_{ij}^2$  die Elemente der Matrix  $W_2$  aus Gleichung 3.21 sind, dann besteht ein Lernschritt aus der Änderung der Matricelemente, das heißt der Gewichte, nach folgender Formel:

$$\begin{aligned} \Delta w_{ij}^1 &= \eta \cdot \varepsilon_i \cdot h_j, \\ \Delta w_{ij}^2 &= \eta \cdot \sum_m e_m \cdot w_{mi}^1 \cdot e_j, \end{aligned} \quad (3.23)$$

mit

$e_i$  : Eingabewerte,  
 $h_j$  : versteckte Werte,  
 $\varepsilon := z_i - a_i$  = Zielwert - Netzwert = Fehler,  
 $\eta$  : Lernrate ( $\eta > 0$  frei wählbar).

Die Lernschritte verändern das neuronale Netz in Richtung eines (lokalen) Minimums und entsprechen der Korrekturformel eines einschichtigen Perzeptrons. Eine verbesserte Variante des gezeigten Algorithmus lautet:

$$\begin{aligned} \Delta w_{ij}^1 &= \eta \cdot \varepsilon_i \cdot a_i \cdot (1 - a_i) \cdot h_j, \\ \Delta w_{ij}^2 &= \eta \cdot \sum_m \varepsilon_m \cdot a_m \cdot (1 - a_m) \cdot w_{mi}^1 \cdot h_i \cdot (1 - h_i) \cdot e_j. \end{aligned} \quad (3.24)$$

Diese Korrekturformel ist präziser, da die Gewichte nicht mehr wesentlich verändert werden, je kleiner der Fehler wird. Nur wenn die berechneten Werte und die Zielwerte stark indifferent sind, werden die Gewichte auch stark verändert. Allgemein lautet damit der Lernalgorithmus mit Backpropagation, auch Fehlerrückführungsmethode genannt:

1. Initialisierung aller Gewichte mit Zufallszahlen.
2. Wahl eines zufälligen Ein-/Ausgabemusters der zu lernenden Funktion und Berechnung der Belegungen  $h_j$  der versteckten Schichten.
3. Korrektur der Gewichte entsprechend Gleichung 3.23 oder 3.24 für die vorgegebenen Eingabewerte  $e_i$  und Zielwerte  $z_i$ .
4. Weiter bei [2.].

Der große Vorteil der Backpropagation-Methode ist, dass ihr mathematischer Formalismus auf jedes Netz anwendbar ist und auch unabhängig von der zu lernenden Funktion ist. Allerdings ist die Anzahl an Lernschritten häufig sehr groß, sodass die Lernphase sehr rechenintensiv ist. Dies wird etwas verbessert, wenn die Gewichte mit kleinen Werten initialisiert werden. Bei großen Gewichten ergibt sich sonst keine wesentliche Verbesserung je Lernschritt. Auch wenn bereits mit zweischichtigen neuronalen Netzen alle Funktionen darstellbar sind, wird mit der Backpropagation-Methode nicht immer die richtige Lösung gefunden. Ursache hierfür ist die Minimierung der Fehlerfunktion. Wie bei jeder Minimierungs- oder Maximierungs-Aufgabe kann statt eines absoluten Minimums ein lokales Minimum gefunden werden. Um die zu lernende Funktion exakt abzubilden ist jedoch das absolute Minimum notwendig. Die Werte sind häufig ähnlich, aber eben nicht exakt. Auch die Anzahl der Neuronen in der versteckten Schicht spielt eine Rolle. Werden zu wenige Neuronen verwendet, ist die zu lernende Funktion möglicherweise nicht darstellbar, da die Kapazität des Netzes nicht ausreichend ist. Bei zu vielen versteckten Neuronen erhöht sich jedoch die Zahl der unabhängigen Variablen in der Fehlerfunktion, was dazu führt, dass die Wahrscheinlichkeit, nur ein lokales Minimum zu finden, erhöht ist. Zudem nimmt mit steigender Zahl an Neuronen die sowieso schon hohe Rechenzeit der Lernphase entsprechend zu. Für die genannten Schwierigkeiten gibt es keine theoretischen Lösungen. Eine Fehlentwicklung in der Lernphase lässt sich nur durch experimentelle Verifikation, das heißt Testläufe, vermeiden. Das bedeutet ein Programm so zu erstellen, dass während der Lernphase die zu lernenden Funktionswerte mit den Netz-Funktionswerten vergleichbar sind. Ist kein Erfolg bemerkbar, wird die Lernphase abgebrochen und nach entsprechenden Anpassungen, wie zum Beispiel einer Vergrößerung/Verkleinerung des Netzes oder einer Änderung der Initialwerte der Gewichte, wiederholt. Zur Beschleunigung der Konvergenz werden die Gewichte nach der Formel

$$w_{ij}^{neu} = \mu \cdot w_{ij}^{alt} + (1 - \mu) \cdot \Delta w_{ij} \quad (3.25)$$

berechnet, wobei  $\mu$  eine Variante der Lernrate ist, die zu Beginn klein bzw. nahe 0 ist und nach und nach größer wird bis zu einem Grenzwert 1.  $\Delta w_{ij}$  ist die Gewichtskorrektur aus den Gleichungen 3.23 und 3.24. Zu Beginn werden die Gewichte noch stark korrigiert, im Laufe des Lernverfahrens wird die Gewichtskorrektur immer schwächer.

### 3.2.5 Deep Learning

Als Deep Learning (dt. etwa *tiefgehendes Lernen*) werden neuronale Netze mit einer hohen Anzahl an versteckten Schichten und Parametern (Gewichten) bezeichnet [GP17]. Anders als bei „normalen“ vorwärts gerichteten Netzen ist die Anzahl der Neuronen deutlich größer, sodass komplexere Funktionen darstellbar sind. Auch die Vernetzung der Neuronen zwischen den Schichten ist daher deutlich komplexer. Werden bei den normalen neuronalen Netzen die Neuronenschichten noch vollständig miteinander verbunden, das heißt jedes Neuron der einen Schicht hatte eine Verbindung zu jedem Neuron der folgenden Schicht, so wandelt sich die Vernetzung bei Deep Learning Netzen zu lokal verbundenen Gruppen von Neuronen und Rückkopplungen vom Ausgang eines Neurons zu dessen Eingang. Da die Anzahl der Verbindungen größer ist, ergeben sich mehr Parameter (Gewichte), die während des Trainings optimiert werden müssen. Deshalb ist der Rechenaufwand, den Deep Learning Netze während des Trainings aufweisen, um ein Vielfaches höher.

Ein besonders bedeutender Vorteil von Deep Learning gegenüber normalen neuronalen Netzen ist die automatische Extrahierung von Merkmalen. Die Datenvorverarbeitung bei traditionellen Machine Learning Algorithmen ist sehr zeit- und arbeitsintensiv. Beim Deep Learning trifft das Netz selbst die Entscheidung, welche Eigenschaften bzw. Merkmale eines Datensatzes als Indikatoren für die Kennzeichnung von Daten relevant und nützlich sind.

Im Bereich des Deep Learning werden zwei Architekturen von neuronalen Netzen am häufigsten verwendet: das konvolutionale neuronale Netz (engl. *Convolutional Neural Network*, *CNN*) und das rekurrente neuronale Netz (engl. *Recurrent Neural Network*, *RNN*).

Konvolutionale neuronale Netze basieren auf der Arbeitsweise der Sehrinde (visueller Kortex) des menschlichen Gehirns und werden deshalb hauptsächlich in der Bildverarbeitung eingesetzt [RM17; GP17]. Sie liefern erstklassige Ergebnisse bei der Erkennung von Objekten und der Bildklassifizierung, was bedeutende Fortschritte in der Bildverarbeitung und des maschinellen Sehens ermöglicht. Durch sogenannte Faltungen lernen CNNs Funktionen höherer Ordnung in Daten und erkennen dadurch zum Beispiel Gesichter, Personen und Straßenschilder. Aber auch zur Analyse von Text- und Audio-Daten sind CNNs einsetzbar. Anwendung finden CNNs in selbstfahrenden Autos, in der Robotik, bei Drohnen und Hilfsmitteln für Sehbehinderte. Da in dieser Arbeit jedoch die rekurrenten neuronalen Netze, das sind Netze mit Rückkopplungen, von größerer Bedeutung sind, werden diese im nachfolgenden Kapitel 3.2.6 ausführlicher betrachtet.

### Aktivierungsfunktionen des Deep Learnings

Die vorgestellten Architekturen des Deep Learnings bringen auch einige Veränderungen gegenüber den normalen, vorwärts gerichteten Netzen mit sich, wie unter anderem neue Aktivierungsfunktionen, andere Neuronen-Typen und hybride Architekturen. Abgesehen von der bereits vorgestellten Sprungfunktion sowie der sigmoiden Funktion in Kapitel 3.2.3 sind weitere, häufig benutzte Aktivierungsfunktionen in Deep Learning Netzen in Abbildung 3.9 gezeigt. Die dazugehörigen Funktionsgleichungen lauten:

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad \text{Tangens Hyperbolicus (tanh)} \quad (3.26)$$

$$f(x) = \max(-1, \min(1, x)) \quad \text{hard Tangens Hyperbolicus} \quad (3.27)$$

$$f(x) = \max(0, x) = \begin{cases} 0 & \text{falls } x < 0 \\ x & \text{falls } x \geq 0 \end{cases} \quad \text{ReLU-Funktion (Rectified Linear Unit)} \quad (3.28)$$

$$f(x) = \max(0, 0.01x, x) \quad \text{Leaky ReLU-Funktion} \quad (3.29)$$

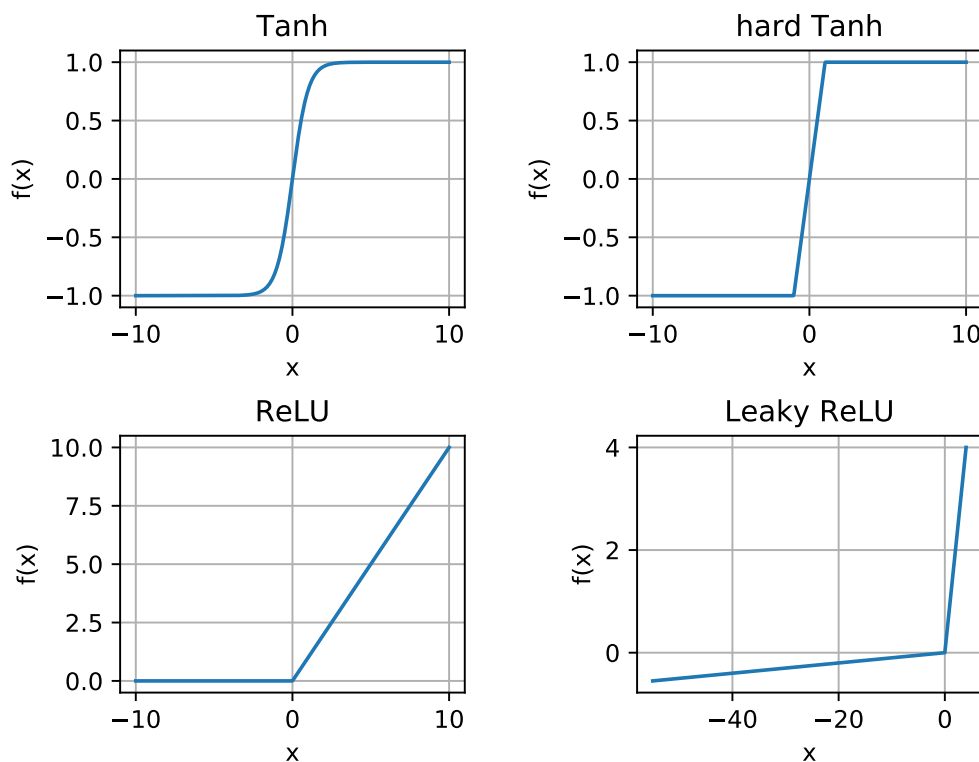


Abbildung 3.9: Aktivierungsfunktionen des Deep Learnings (Eigene Darstellung).

### 3.2.6 Rekurrente neuronale Netze

Bei den vorwärts gerichteten Netzen wird jedem Eingabevektor ein Ausgabevektor zugeordnet [Kin94]. Die Zuordnung wird durch einen vorwärts gerichteten Informationsfluss bestimmt. Außerdem haben die Vektoren eine feste Größe und sind unabhängig voneinander, das heißt die Reihenfolge der Eingabevektoren beim Trainieren spielt keine Rolle [RM17]. Im Gegensatz dazu wird bei Netzen mit Rückkopplung oder rekurrenten neuronalen Netzen (engl. *recurrent neural networks*, *RNN*) diese Beziehung aufgehoben und die Ausgabe wird wieder auf die Eingabe zurückgeführt, sodass ein Iterationsprozess (Schleife) entsteht [Kin94]. Es ergibt sich ein neuronales Netz mit Rückkopplung (Feedback). Deshalb spielt die Reihenfolge der einzelnen Eingaben eine Rolle, das bedeutet die Eingabevektoren sind voneinander abhängig [RM17].

Die Rückkopplungen werden auch als zeitliche Abhängigkeit angesehen, oder als Möglichkeit, Information über Zeitschritte hinweg zu senden [GP17]. Bei den klassischen neuronalen Netzen wird eine solche Zeit-Dynamik nur durch Schiebefenster bei der Eingabe ermöglicht, das heißt es wird zum Beispiel der vorherige, der aktuelle und der nächste Vektor als ein großer Eingabevektor an das neuronale Netz übergeben. Der Nachteil dieser Methode ist, dass ein solches Schiebefenster immer eine konstante, begrenzte Größe hat. Hier bieten rekurrente neuronale Netze einen bedeutenden Vorteil, da sie mit Hilfe der Feedback-Schleifen Sequenzen beliebiger Länge verarbeiten [Gér18]. Deshalb werden Netze mit Rückkopplung hauptsächlich bei der Analyse von Zeitreihen und Sequenzen eingesetzt, um damit die Zukunft vorherzusagen.

RNNs weisen verschiedene Eingabe-/Ausgabe-Muster auf, die in Abbildung 3.10 gezeigt sind [Gér18]:

- **1-zu-n**: Das RNN erhält eine einzelne Eingabe und soll eine Folge von Ausgaben berechnen. Damit wird zum Beispiel die Beschreibung eines Bildes automatisch erstellt.
- **n-zu-1**: Umgekehrt kann dem RNN auch eine Sequenz an Eingaben übergeben werden und es wird nur die letzte Ausgabe berücksichtigt. Ein solches Netz analysiert beispielsweise die Meinung einer Filmrezension, wie etwa miserabel oder fantastisch, anhand der verwendeten Wörter.
- **n-zu-n (synchron)**: Einem RNN wird eine Sequenz als Eingabe gegeben und es soll eine Sequenz von Ausgaben produzieren. Ein solches Netz wird zum Beispiel zur Vorhersage von Zeitreihen wie Aktienkurse verwendet. Bei Eingabe der Preise der letzten  $N$  Tage gibt das Netz die um einen Tag in die Zukunft verschobenen Preise aus.
- **n-zu-n (verzögert)**: Das RNN wandelt eine Eingabesequenz in einen Vektor um (Encoder) und verarbeitet diesen Vektor in eine Ausgabesequenz (Decoder). Damit ist beispielsweise das Übersetzen eines Satzes aus einer Sprache in eine andere möglich. Der Unterschied zum synchronen n-zu-n-Muster ist, dass der Satz erst komplett aufgenommen wird, bevor er übersetzt wird.

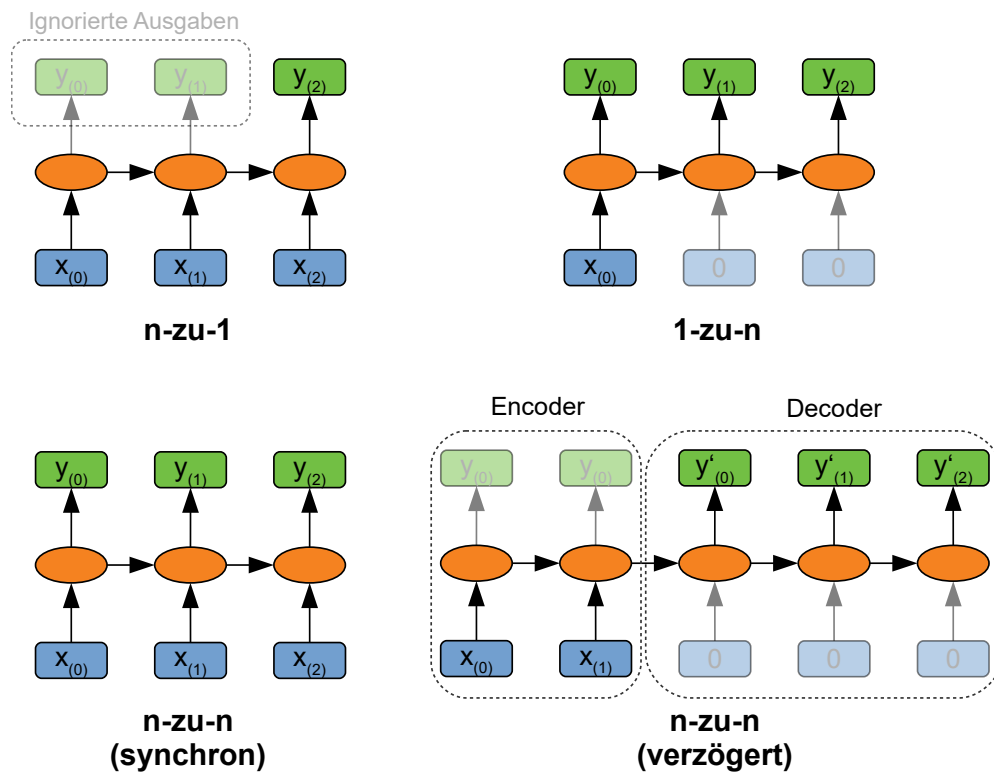
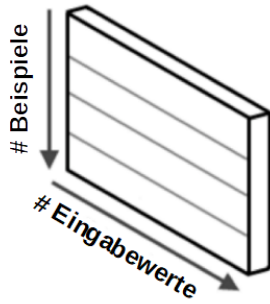


Abbildung 3.10: Mögliche Muster zwischen Ein- und Ausgabe bei rekurrenten neuronalen Netzen (In Anlehnung an [RM17, S. 531], [Gér18, S. 389]).



Im Unterschied zu den bisher betrachteten, vorwärts gerichteten Netzen hat die Eingabe eines rekurrenten neuronalen Netzes drei Dimensionen: Anzahl der Beispiele  $\times$  Anzahl der Eingabewerte (Eigenschaften/Merkmale)  $\times$  Anzahl der Zeitschritte (Sequenzlänge). Dies zeigt Abbildung 3.11.

Vorwärts gerichtetes Netz



Rekurrentes Netz

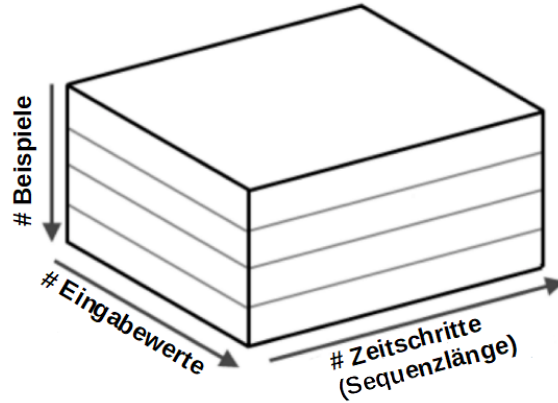


Abbildung 3.11: Eingabe-Dimensionen eines vorwärts gerichteten neuronalen Netzes im Vergleich zum RNN (aus dem Englischen nach [GP17, Abb. 4-17]).

### Rekurrente Neuronen

Ein rekurrentes Neuron ist einem vorwärts gerichteten Neuron sehr ähnlich, mit der Ausnahme, dass es auch eine rückwärts gerichtete Verbindung enthält [Gér18]. Ein rekurrentes Neuron hat eine Eingabe  $\vec{x}$  und eine Ausgabe  $\vec{y}$ , die es wieder an sich selbst schickt, wie in Abbildung 3.12 gezeigt. Zu jedem Ausführungszeitpunkt  $t$ , auch Frame genannt, erhält ein solches Neuron die Eingabe  $\vec{x}_{(t)}$  sowie die eigene Ausgabe aus dem vorherigen Schritt  $\vec{y}_{(t-1)}$ . Um diese Zeitabhängigkeit darzustellen, wird ein rekurrentes neuronales Netz entlang der Zeitachse aufgerollt, wie in Abbildung 3.12 für ein Neuron zu sehen ist.

Um ein Netzwerk aus mehreren rekurrenten Neuronen aufzubauen, wird jedem Neuron der Eingabevektor  $\vec{x}_{(t)}$  sowie der Ausgabevektor aus dem vorigen Schritt  $\vec{y}_{(t-1)}$  zugeführt. Abbildung 3.13 zeigt ein einfaches rekurrentes Netz bestehend aus einer Neuronenschicht und dessen aufgerollte Darstellung entlang der Zeitachse. Wie aus der Abbildung ersichtlich hat jedes rekurrente Neuron im Gegensatz zu einem klassischen Neuron zwei Gewichtsvektoren: einen Gewichtsvektor  $\vec{w}_x$  für die Eingaben und einen Gewichtsvektor  $\vec{w}_y$  für die Ausgaben des vorangegangenen Schritts, das heißt der Rückführung. Damit lässt sich die Ausgabe eines neuronalen Netzes mit einer rekurrenten Neuronenschicht folgendermaßen berechnen:

$$\vec{y}_{(t)} = f(\vec{x}_{(t)}^T \cdot \vec{w}_x + \vec{y}_{(t-1)}^T \cdot \vec{w}_y + b), \quad (3.30)$$

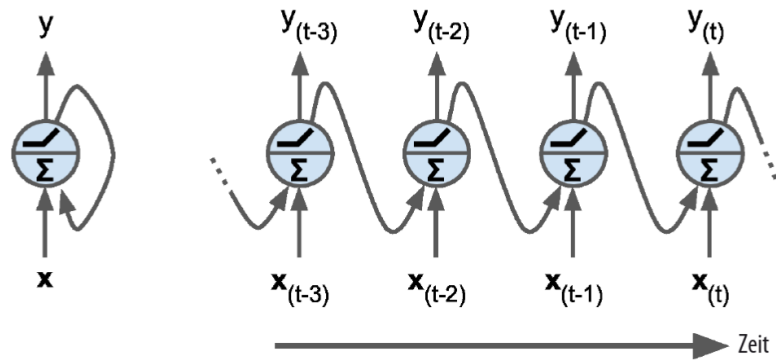


Abbildung 3.12: Darstellung eines rekurrenten Neurons entlang der Zeitachse aufgerollt (Quelle: [Gér18, S. 386]).

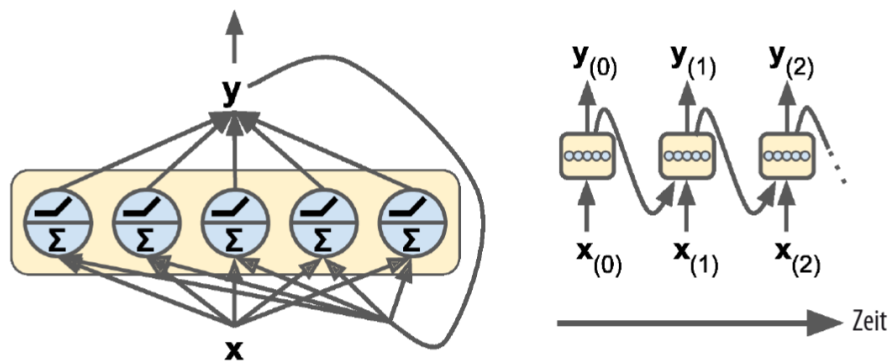


Abbildung 3.13: Darstellung einer rekurrenten Neuronenschicht entlang der Zeitachse aufgerollt (Quelle: [Gér18, S. 386]).

mit dem Bias  $b$  und der Aktivierungsfunktion  $f$ . Es zeigt sich, dass  $\vec{y}_{(t)}$  eine Funktion von  $\vec{x}_{(t)}$  und  $\vec{y}_{(t-1)}$  ist, was wiederum eine Funktion von  $\vec{x}_{(t-1)}$  und  $\vec{y}_{(t-2)}$  ist, und so weiter. Die Ausgabe  $\vec{y}_{(t)}$  ist deshalb eine Funktion aller Eingaben  $\vec{x}_{(0)}, \vec{x}_{(1)}, \dots, \vec{x}_{(t)}$  seit dem Zeitpunkt  $t = 0$ . Da beim ersten Schritt  $t = 0$  keine vorherigen Ausgaben zur Verfügung stehen, werden diese üblicherweise mit dem Wert 0 initialisiert.

Der eben geschilderte Zusammenhang zwischen der Ausgabe eines rekurrenten Neurons und aller Eingaben der vorherigen Schritte wird auch als eine Art Gedächtnis beschrieben [Gér18]. Der Teil eines rekurrenten Netzes, der seinen Zustand über mehrere Zeitschritte speichert, wird als Gedächtniszelle oder Speicherzelle bezeichnet. Das bisher betrachtete rekurrente Neuron bzw. die Neuronenschicht ist eine sehr einfache Gedächtniszelle. Der Zustand  $\vec{h}_{(t)}$  ( $h$  für engl. *hidden*) einer solchen Zelle ist eine Funktion der Eingaben zu diesem Zeitpunkt und des Zustandes zum vorherigen Zeitpunkt:

$$\vec{h}_{(t)} = f(\vec{h}_{(t-1)}, \vec{x}_{(t)}). \quad (3.31)$$

Bei den bisher in den Abbildungen 3.12 und 3.13 gezeigten einfachen Gedächtniszellen entspricht die Ausgabe  $\vec{y}_{(t)}$  dem Zustand  $\vec{h}_{(t)}$ . Im Allgemeinen ist dies aber nicht der Fall, wie Abbildung 3.14 zeigt.

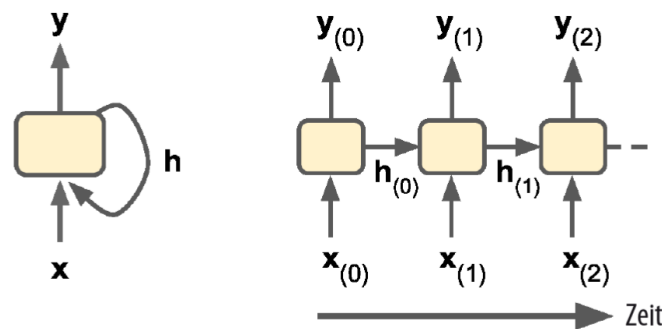


Abbildung 3.14: Darstellung einer Gedächtniszelle: der verborgene Zustand und die Ausgabe können unterschiedlich sein (Quelle: [Gér18, S. 388]).

### Trainieren von RNNs

Um ein rekurrentes neuronales Netz zu trainieren, wird das Netz entlang der Zeitachse aufgerollt, sodass anschließend das gewöhnliche Backpropagation-Verfahren anwendbar ist [Gér18]. Dieses Verfahren wird auch Backpropagation through Time (BPTT) genannt. Grundsätzlich basieren beide Algorithmen auf dem gleichen Prinzip, die Bezeichnung „über die Zeit“ (engl. *through Time*) weist jedoch darauf hin, dass einige der Gradienten zurück von einem zukünftigen zum aktuellen Zeitschritt fließen, und nicht von Schicht zu Schicht [GP17].

Als Beispiel wird nachfolgend ein einschichtiges RNN über eine Zeitsequenz von 12 Zeitschritten betrachtet [GP17]. Beim Backpropagation-Verfahren und der Variation

BPTT wird zunächst eine Vorwärtsrechnung über das aufgerollte Netz, das heißt über die 12 Zeitschritte, durchgeführt [GP17; Gér18]. Die dabei resultierende Ausgabesequenz wird mit einer Kostenfunktion  $C$  über eine bestimmte Anzahl an Zeitschritten evaluiert. Mit den so bestimmten Gradienten werden bei der Rückwärtsrechnung über das aufgerollte Netz die Gewichte der Verbindungen aktualisiert. Abbildung 3.15 gibt einen Überblick über das Verfahren.

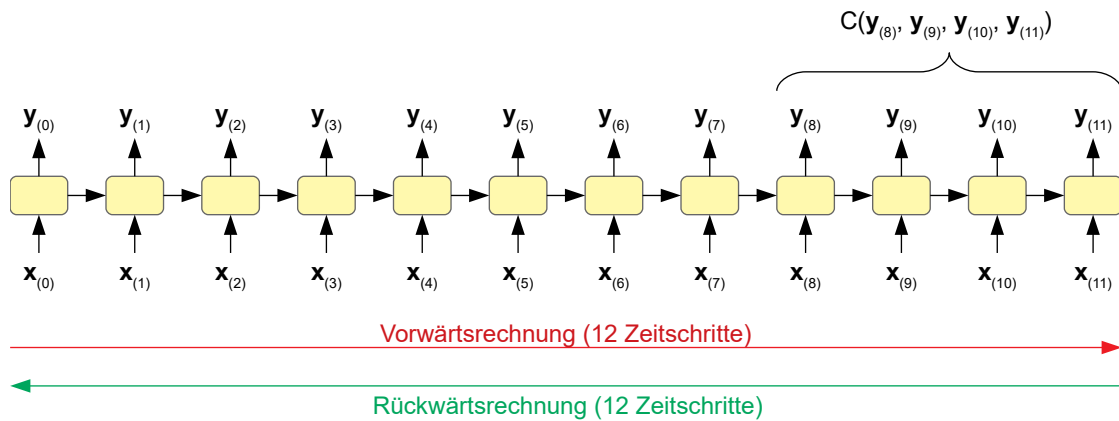


Abbildung 3.15: Standard Backpropagation through Time (BPTT) für 12 Zeitschritte (In Anlehnung an [GP17, Abb. 4-26], [Gér18, S. 395]).

Bei dem im vorherigen Abschnitt betrachteten Beispiel ergeben die 12 Zeitschritte kein aufwändiges Training [GP17]. Wird jedoch ein RNN für deutlich längere Sequenzen, wie zum Beispiel 1000, trainiert, dann enthält das aufgerollte RNN viele Zeitschritte, die in der Vorwärts- und Rückwärtsrechnung berücksichtigt werden müssen. Infolgedessen wird das Training mit zunehmender Sequenzlänge sehr schnell rechenintensiv. Um die Berechnungskomplexität zu reduzieren, wird zum Trainieren von solchen tiefen RNNs (engl. *Deep-RNNs*) der Truncated Backpropagation through Time Algorithmus angewendet, der in Abbildung 3.16 dargestellt ist. Dabei wird das RNN nur über eine begrenzte Anzahl an Zeitschritten, beispielsweise vier, aufgerollt und die Vorwärts- und Rückwärtsrechnung in kleinere Operationen aufgeteilt [GP17; Gér18].

Bei rekurrenten Netzen die lange Sequenzen verarbeiten, das heißt viele Zeitschritte berücksichtigen, tritt außerdem das Problem der schwindenden/explodierenden Gradienten auf [Gér18]. Der Backpropagation-Algorithmus arbeitet sich von der Ausgabe zur Eingabeschicht vor und berechnet währenddessen die entsprechenden Fehlergradienten. Bei Schichten nahe der Eingabeschicht werden diese Gradienten immer kleiner, sodass sich die Gewichte der ersten Schichten kaum noch verändern. Auch das Gegenteil ist möglich, nämlich dass die Gradienten immer größer werden und die Gewichte vieler Schichten eine extrem große Änderung erfahren. Beides verhindert ein effektives Training des neuronalen Netzes.

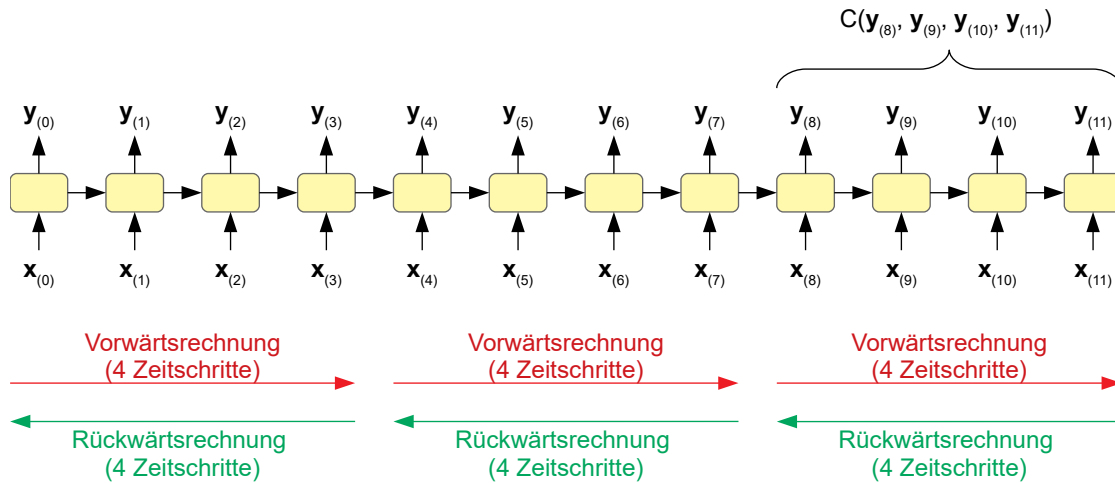


Abbildung 3.16: Truncated Backpropagation through Time für 12 Zeitschritte (In Anlehnung an [GP17, Abb. 4-27], [Gér18, S. 395]).

Ein weiteres Problem von lang laufenden RNNs besteht darin, dass die Erinnerung an die ersten Eingaben mit der Zeit verblasst [Gér18]. Die Daten werden beim Durchlaufen eines RNNs mehrfach transformiert, sodass bei jedem Schritt ein Teil der Information verloren geht. Nach einer Vielzahl an Schritten findet sich deshalb fast keine Information mehr der ersten Eingaben wieder. Diese Tatsache widerspricht dem Hauptgrund, weshalb RNNs für die Verarbeitung von Sequenzen eingesetzt werden.

Eine effektive Lösung für all die beschriebenen Probleme von rekurrenten neuronalen Netzen sind verschiedenartige Zellen mit Langzeitgedächtnis, die in den folgenden Abschnitten vorgestellt werden.

### LSTM-Zellen

Die am häufigsten verwendete Variante ist die Long Short-Term Memory (LSTM) Zelle, die 1997 von Sepp Hochreiter und Jürgen Schmidhuber eingeführt wurde [GP17; Gér18]. Wird eine LSTM-Zelle als Blackbox betrachtet, wird sie im Wesentlichen wie ein rekurrentes Neuron, die einfachste Form einer Gedächtniszelle, verwendet [Gér18]. Nur das Training konvergiert schneller und selbst weitreichende Abhängigkeiten in den Daten werden erkannt. Von außen betrachtet sieht eine LSTM-Zelle wie eine gewöhnliche rekurrente Zelle aus, außer dass ihr Zustand in zwei Vektoren  $\vec{h}_{(t)}$  und  $\vec{c}_{(t)}$  ( $c$  für engl. *cell*) aufgeteilt ist. Dabei wird  $\vec{h}_{(t)}$  als Kurzzeitgedächtnis und  $\vec{c}_{(t)}$  als Langzeitgedächtnis bezeichnet. Die Idee ist, dass das Netz lernt, was im Langzeitgedächtnis gespeichert werden soll, was es vergessen kann und wie das Gedächtnis zu interpretieren ist. Abbildung 3.17 zeigt den schematischen Aufbau einer LSTM-Zelle.

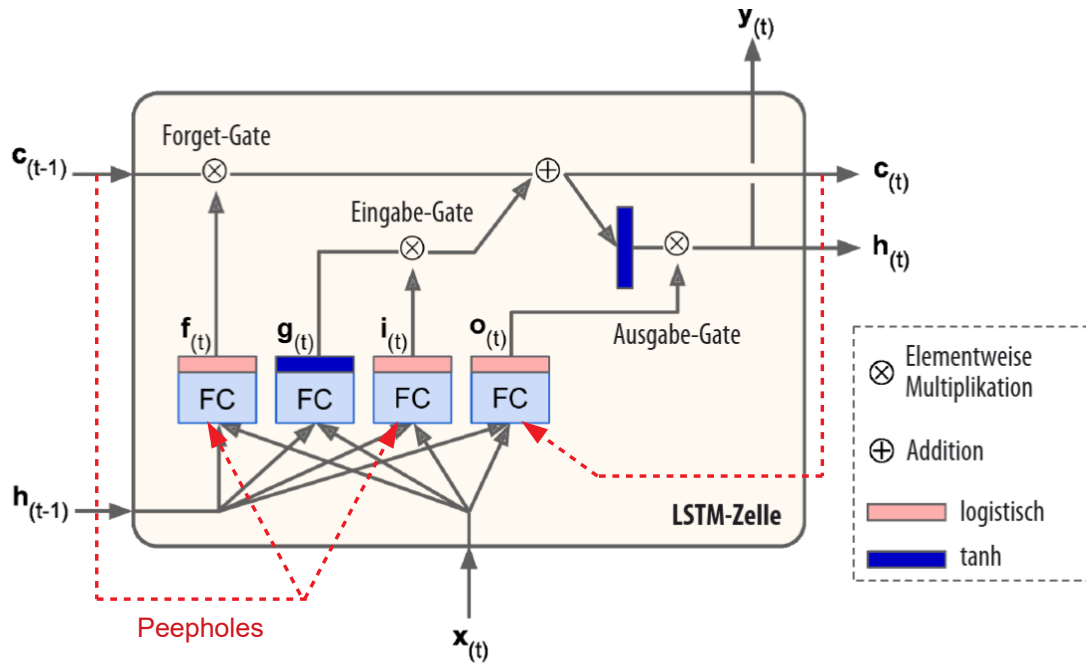


Abbildung 3.17: Schematischer Aufbau einer LSTM-Zelle mit Peephole-Verbindungen (In Anlehnung an [Gér18, S. 407]).

Der Langzeitzustand  $\vec{c}_{(t-1)}$  passiert zunächst ein Forget Gate, das einige Erinnerungen vergisst [Gér18]. Danach werden durch Addition neue Erinnerungen hinzugefügt, die von einem Input Gate ausgewählt werden. Das Ergebnis  $\vec{c}_{(t)}$  wird unverändert ausgegeben. Der Langzeitzustand wird außerdem nach der Addition kopiert und von einem Output-Gate gefiltert. Dies ergibt den Zustand des Kurzzeitgedächtnisses  $\vec{h}_{(t)}$ , was der Ausgabe der Zelle  $\vec{y}_{(t)}$  entspricht.

Zunächst wird die aktuelle Eingabe  $\vec{x}_{(t)}$  und der Zustand des Kurzzeitgedächtnisses  $\vec{h}_{(t-1)}$  an vier einzelne, vollständig verbundene (engl. *fully-connected*) Schichten übergeben [Gér18]. Die wichtigste Schicht analysiert die aktuellen Eingaben und den vorherigen Zustand und gibt  $\vec{g}_{(t)}$  aus. Diese Schicht entspricht einer einfachen Gedächtniszelle. Die anderen Schichten sind sogenannte Gate-Controller und haben sigmoide Aktivierungsfunktionen mit den Ausgabewerten  $[0, 1]$ . Anschaulich bedeutet der Wert 1 bzw. 0, dass das Gate geschlossen bzw. geöffnet ist. Das Forget-Gate bestimmt, welche Teile des Langzeitgedächtnisses gelöscht werden. Das Input-Gate steuert, welche Teile von  $\vec{g}_{(t)}$  ins Langzeitgedächtnis übertragen werden, und das Output-Gate beeinflusst das Auslesen bzw. Ausgeben von Teilen des Langzeitgedächtnisses. Zusammengefasst lernt eine LSTM-Zelle, wichtige Eingaben zu erkennen (Input-Gate), diese im Langzeitgedächtnis zu speichern und solange wie nötig aufzubewahren (Forget-Gate), sowie entsprechende Schlüsse daraus zu entnehmen (Output-Gate).

Die Zustände einer LSTM-Zelle lassen sich nach folgenden Formeln bestimmen:

$$\vec{i}_{(t)} = \sigma(W_{xi}^T \cdot \vec{x}_{(t)} + W_{hi}^T \cdot \vec{h}_{(t-1)} + \vec{b}_i), \quad (3.32)$$

$$\vec{f}_{(t)} = \sigma(W_{xf}^T \cdot \vec{x}_{(t)} + W_{hf}^T \cdot \vec{h}_{(t-1)} + \vec{b}_f), \quad (3.33)$$

$$\vec{o}_{(t)} = \sigma(W_{xo}^T \cdot \vec{x}_{(t)} + W_{ho}^T \cdot \vec{h}_{(t-1)} + \vec{b}_o), \quad (3.34)$$

$$\vec{g}_{(t)} = \tanh(W_{xg}^T \cdot \vec{x}_{(t)} + W_{hg}^T \cdot \vec{h}_{(t-1)} + \vec{b}_g), \quad (3.35)$$

$$\vec{c}_{(t)} = \vec{f}_{(t)} \otimes \vec{c}_{(t-1)} + \vec{i}_{(t)} \otimes \vec{g}_{(t)}, \quad (3.36)$$

$$\vec{y}_{(t)} = \vec{h}_{(t)} = \vec{o}_{(t)} \otimes \tanh(\vec{c}_{(t)}), \quad (3.37)$$

mit

$W_{xi}, W_{xf}, W_{xo}, W_{xg}$  : Gewichtsmatrizen der Verbindungen des Eingabevektors  $\vec{x}_{(t)}$  zu den vier Schichten,

$W_{hi}, W_{hf}, W_{ho}, W_{hg}$  : Gewichtsmatrizen der Verbindungen des Kurzzeitgedächtnisses  $\vec{h}_{(t-1)}$  zu den vier Schichten,

$b_i, b_f, b_o, b_g$  : Bias-Terme der vier Schichten.

Um den Gate-Controllern Kontextwissen zur Verfügung zu stellen, schlagen Felix Gers und Jürgen Schmidhuber in einer Veröffentlichung aus dem Jahr 2000 zusätzliche Verbindungen vor, sogenannte Peephole-Verbindungen [Gér18]. Der Zustand des Langzeitgedächtnisses aus dem vorigen Schritt  $\vec{c}_{(t-1)}$  wird jeweils eine zusätzliche Eingabe des Forget- und des Input-Gates. Das Output-Gate erhält als weitere Eingabe den aktuellen Zustand des Langzeitgedächtnisses  $\vec{c}_{(t)}$ . So haben die Gate-Controller die Möglichkeit, in das Langzeitgedächtnis zu schauen.

### GRU-Zellen

Die Gated Recurrent Unit (GRU) Zelle ist ein anderer Vorschlag für eine Zelle mit Langzeitgedächtnis von Kyunghyun Cho et al. datiert aus dem Jahr 2014 [Gér18]. Die GRU-Zelle ist eine vereinfachte Variante der LSTM-Zelle, die im Allgemeinen einfacher zu berechnen und zu implementieren ist als die LSTM-Zelle [GP17; Gér18]. Abbildung 3.18 stellt eine GRU-Zelle dar.

Die wichtigsten Vereinfachungen sind [Gér18]:

- Die beiden Zustandsvektoren  $\vec{h}_{(t)}$  und  $\vec{c}_{(t)}$  sind zu einem Zustandsvektor  $\vec{h}_{(t)}$  vereinigt.
- Es gibt nur einen Gate-Controller, der sowohl das Forget-Gate, als auch das Input-Gate steuert. Der Wertebereich des Gate-Controllers ist  $[0, 1]$ , wobei der Wert 1 für ein offenes Input-Gate und ein geschlossenes Forget-Gate steht, der Wert 0 für das Gegenteil. Anschaulich bedeutet das, dass beim Speichern einer Erinnerung zuerst der Speicherplatz dafür gelöscht wird.

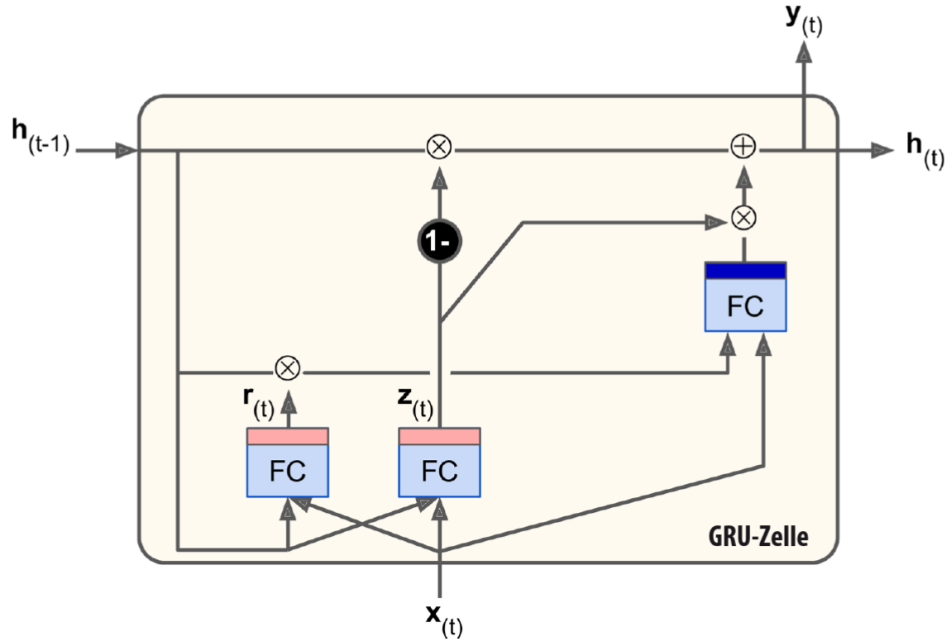


Abbildung 3.18: Schematischer Aufbau einer GRU-Zelle (Quelle: [Gér18, S. 407]).

- In jedem Schritt wird der vollständige Zustandsvektor ausgegeben, da kein Output-Gate vorhanden ist. Stattdessen steuert ein neuer Gate-Controller, welcher Teil des vorherigen Zustands der Hauptschicht zur Verfügung steht.

Mathematisch lässt sich eine GRU-Zelle folgendermaßen beschreiben:

$$\vec{z}_{(t)} = \sigma(W_{xz}^T \cdot \vec{x}_{(t)} + W_{hz}^T \cdot \vec{h}_{(t-1)}), \quad (3.38)$$

$$\vec{r}_{(t)} = \sigma(W_{xr}^T \cdot \vec{x}_{(t)} + W_{hr}^T \cdot \vec{h}_{(t-1)}), \quad (3.39)$$

$$\vec{g}_{(t)} = \tanh(W_{xg}^T \cdot \vec{x}_{(t)} + W_{hg}^T \cdot (\vec{r}_{(t)} \otimes \vec{h}_{(t-1)})), \quad (3.40)$$

$$\vec{h}_{(t)} = (1 - \vec{z}_{(t)}) \otimes \vec{h}_{(t-1)} + \vec{z}_{(t)} \otimes \vec{g}_{(t)}. \quad (3.41)$$

Eine der Hauptanwendungen von rekurrenten neuronalen Netzen mit LSTM- und GRU-Zellen ist die natürliche Sprachverarbeitung (engl. *natural language processing*, *NLP*), das bedeutet unter anderem maschinelle Übersetzung, automatisches Zusammenfassen oder Meinungsanalysen [Gér18].



## 4 Stand der Technik

### 4.1 Traditionelle und heuristische Schedulability-Analyse

#### 4.1.1 Fixed Priority

##### Simulation

Die Simulation ist ein exaktes Mittel, um die Planbarkeit eines Task-Sets mit fixen Prioritäten festzustellen [BW96]. Für Task-Sets, die eine gemeinsame Auslösezeit  $t = 0$  haben, ist es ausreichend, die Simulation bis zur Hyperperiode  $H$  durchzuführen. Die Hyperperiode ist das kleinste Intervall, nach dem sich der Ablaufplan wiederholt, und ist definiert durch das kleinste gemeinsame Vielfache (engl. *least common multiple*, LCM) der Perioden eines Task-Sets:

$$H = lcm(T_1, \dots, T_n). \quad (4.1)$$

Wenn alle Tasks in diesem Intervall ihre Deadlines einhalten, werden auch alle zukünftigen Deadlines eingehalten und das Task-Set ist mit dem FP-Algorithmus planbar.

##### Prozessorauslastung

Ein sehr einfacher, wenn auch nicht exakter Schedulability-Test für das Planen nach Raten (RM-Algorithmus) wird in [LL73] gezeigt. Der Test basiert auf der Berechnung der Prozessorauslastung. Ein Set aus  $n$  periodischen Tasks, deren Deadlines den Task-Perioden entsprechen, das heißt es gilt  $D_i = T_i$  für einen Task  $\tau_i$ , ist mit dem RM-Algorithmus planbar, wenn

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \quad (4.2)$$

Es ergibt sich eine Auslastungsgrenze, die für große Werte von  $n$  gegen  $U = \ln(2) = 0,693$  konvergiert. Das bedeutet, dass jedes Task-Set mit einer Prozessorauslastung von weniger als 69,3 % durch einen präemptiven FP-Algorithmus planbar ist, wenn die Prioritäten nach dem RM-Algorithmus vergeben werden. Gleichung 4.2 stellt dabei eine hinreichende Bedingung für die Planbarkeit eines Task-Sets dar, das heißt das Task-Set ist planbar wenn der Test positiv ausfällt [BW96].

Ein weniger pessimistischer, ebenfalls auf der Prozessorauslastung basierender Schedulability-Test wird in [BBB01] und [BBB03] vorgestellt. Der Ansatz basiert auf einem anderen Maximalwert der Prozessorauslastung, der sogenannten Hyperbolic Bound (HB).

Ein Set aus  $n$  periodischen Tasks und  $D_i = T_i$  ist demnach mit dem RM-Algorithmus planbar, wenn

$$\prod_{i=1}^n (U_i + 1) \leq 2, \quad (4.3)$$

wobei  $U_i$  die Prozessorauslastung eines Tasks  $\tau_i$  ist. Bei gleicher Komplexität ergibt sich ein Gewinn im Sinne der Schedulability gegenüber dem klassischen Test in [LL73], der für große  $n$  gegen  $\sqrt{2}$  tendiert.

### Antwortzeit-Analyse

Die exakte Bestimmung der Planbarkeit eines Task-Sets mit Deadlines kleiner oder gleich der Task-Perioden, entspricht  $D_i \leq T_i$  für einen Task  $\tau_i$ , bietet die Antwortzeit-Analyse (engl. *response time analysis*, RTA) [Aud+91] [Aud+93]. Der Schedulability-Test besteht aus der analytischen Bestimmung der Antwortzeit  $R_i$  eines Tasks  $\tau_i$  im ungünstigsten Fall (engl. *worst-case response time*, WCRT) und dem Vergleich dieses Wertes mit der Task-Deadline  $D_i$ . Ein Task-Set bestehend aus  $n$  Tasks ist mit dem DM-Algorithmus (Planen nach monotonen Fristen) planbar, wenn

$$\forall i : 1 \leq i \leq n \quad R_i \leq D_i. \quad (4.4)$$

Die WCRT des Tasks  $\tau_1$  mit der höchsten Priorität entspricht der Rechenzeit des Tasks, das heißt  $R_1 = C_1$ . Alle anderen Tasks werden Störungen von höher-priorisierten Tasks erfahren. Für einen Task  $\tau_i$  ist die Antwortzeit

$$R_i = C_i + I_i, \quad (4.5)$$

wobei  $I_i$  die maximale Störung (engl. *interference*) des Task  $\tau_i$  durch höher-priorisierte Tasks ist. Die maximale Störung tritt immer dann auf, wenn alle höher-priorisierten Tasks zeitgleich mit Task  $\tau_i$  ausgelöst werden. Dieser Moment wird kritischer Moment (engl. *critical instant*) genannt. Jeder Task  $\tau_j$  mit einer höheren Priorität als Tasks  $\tau_i$  wird in einem Intervall  $[0, D_i]$  folgendermaßen oft ausgelöst:

$$\text{Number\_of\_Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil. \quad (4.6)$$

Jede Auslösung von Task  $\tau_j$  bedeutet eine Störung der Größe  $C_j$ . Die gesamte Störung durch Task  $\tau_j$  ist

$$\text{Maximum\_Interference} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \quad (4.7)$$

Allgemein berechnet sich die gesamte Störung durch alle höher-priorisierten Tasks ausgehend von Task  $\tau_i$  zu

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \quad (4.8)$$

Damit ergibt sich die Antwortzeit eines Tasks  $\tau_i$  zu [JP86]:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \quad (4.9)$$

Der kleinste Wert von  $R_i$ , für den Gleichung 4.9 erfüllt ist, stellt die WCRT des Task  $\tau_i$  dar. Die Berechnung der WCRT ist nicht trivial, wird aber mit Hilfe einer Rekursion durchgeführt [Aud+93]:

$$R_i^{k+1} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j + C_i. \quad (4.10)$$

In [Aud+93] wird als Startwert  $R_i^0 = C_i$  angegeben, in [But11] hingegen  $R_i^0 = \sum_{j=1}^i C_j$ .

Die Rekursion wird in jedem Fall gestoppt, wenn  $R_i^{k+1} = R_i^k$ . Dieser Wert ist die Antwortzeit (WCRT)  $R_i$  eines Tasks  $\tau_i$ .

### Arbeitsbelastung

Ein anderer exakter Schedulability-Test für den RM-Algorithmus basiert auf dem Konzept der Level- $i$  Arbeitsbelastung (engl. *level- $i$  workload*) [LSD89]. Die Level- $i$  Arbeitsbelastung  $W_i(t)$  ist die Rechenzeit, die im Intervall  $[0, t]$  von einem Task  $\tau_i$  und allen Tasks mit höherer Priorität als  $P_i$  angefordert wird. Für ein Set aus periodischen Tasks mit einem kritischen Moment zum Zeitpunkt  $t = 0$  und  $D_i = T_i$  ist die Arbeitsbelastung:

$$W_i(t) = \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j. \quad (4.11)$$

Um die Planbarkeit eines Tasks bzw. eines ganzen Task-Sets zu bestimmen, werden folgende Hilfsfunktionen definiert:

$$L_i(t) = \frac{W_i(t)}{t}, \quad (4.12)$$

$$L_i = \min_{0 < t \leq T_i} L_i(t), \quad (4.13)$$

$$L = \max_{1 \leq i \leq n} L_i. \quad (4.14)$$

Laut [LSD89] ist ein Task  $\tau_i$  mit dem RM-Algorithmus planbar, wenn  $L_i \leq 1$ , und ein ganzes Task-Set ist planbar, wenn

$$L \leq 1. \quad (4.15)$$

Gleichung 4.15 charakterisiert die Planbarkeit eines Task-Sets als Minimierungsproblem über die Variable  $t$  im Intervall  $[0, T_i]$ . Dabei müssen nur Werte von  $t$  getestet werden, die ein Vielfaches der Task-Perioden von  $\tau_i$  und allen höher-priorisierten Tasks im Intervall  $[0, T_i]$  sind. Diese Werte werden Planungspunkte (engl. *scheduling points*) genannt und sind in Gleichung 4.16 für einen Task  $\tau_i$  mit  $T_i = D_i$  definiert. Die Planungspunkte

für einen Task sind dessen Deadline und alle Ankunftszeiten von Tasks mit höherer Priorität vor dieser Deadline.

$$S_i = (k \cdot T_j | j = 1, \dots, i; k = 1, \dots, \lfloor T_i / T_j \rfloor) \quad (4.16)$$

Ebenfalls auf der Arbeitsleistung basiert der Hyperplanes  $\delta$ -Exact Test ( $\delta$ -HET) aus [BB01] und [BB04]. Bei gleicher Komplexität läuft der  $\delta$ -HET im Mittel deutlich schneller ab als eine Antwortzeit-Analyse. Die notwendige und hinreichende Bedingung für die Planbarkeit eines Task-Sets mit fixen Prioritäten lautet

$$\forall i = 1, \dots, n \quad C_i + W_{i-1}(D_i) \leq D_i, \quad (4.17)$$

wobei die Arbeitsbelastung  $W_{i-1}(D_i)$  durch

$$W_{i-1}(D_i) = \min_{t \in P_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j + (D_i - t) \quad (4.18)$$

festgelegt ist und  $W_0(D_1) = 0$  ist. Die Menge der zu testenden Punkte ist

$$P_i(t) = \begin{cases} P_0(t) = \{t\} \\ P_i(t) = P_{i-1} \left( \left\lfloor \frac{t}{T_i} \right\rfloor T_i \right) \cup P_{i-1}(t). \end{cases} \quad (4.19)$$

Der Test ist über einen Parameter einstellbar, um das Verhältnis von Akzeptanzrate versus Komplexität abzustimmen. Der einstellbare  $\delta$ -HET wird von dem Test für den FP-Algorithmus in Gleichung 4.17 abgeleitet, indem  $P_i(b)$  um einen zusätzlichen Parameter  $\delta$  erweitert wird:

$$\begin{cases} P_0(b, \delta) = \{b\} & \forall \delta \\ P_i(b, \delta) = \begin{cases} P_{i-1} \left( \left\lfloor \frac{b}{T_i} \right\rfloor T_i, \delta \right) \cup P_{i-1}(b, \delta) & \text{wenn } b\delta \geq T_i \\ P_{i-1} \left( \left\lfloor \frac{b}{T_i} \right\rfloor T_i, \delta \right) & \text{andernfalls.} \end{cases} \end{cases} \quad (4.20)$$

[Leh90] betrachtet das Problem der Planung von periodischen Tasks mit fixen Prioritäten und beliebigen Deadlines ebenfalls auf Basis der Arbeitsbelastung. Um die Antwortzeit (WCRT) eines Tasks  $\tau_i$  zu ermitteln, wird das Konzept der Level- $i$  Busy Period verwendet. Eine Level- $i$  Busy Period ist ein Intervall, in welchem sich der Prozessor nicht im Leerlauf befindet, sondern alle Tasks mit Prioritäten gleich oder höher  $P_i$  vollständig ausgeführt werden. Die längste Antwortzeit eines Tasks  $\tau_i$  tritt während einer Level- $i$  Busy Period auf, die mit einem kritischen Moment begonnen wird, das bedeutet alle Tasks werden zum gleichen Zeitpunkt ausgelöst. Wenn  $D_i > T_i$  für einen Task  $\tau_i$  möglich ist, dann werden Instanzen einer folgenden Iteration ausgelöst, bevor die Ausführung der Instanzen der aktuellen Iteration beendet ist. Das bedeutet, dass

für die Planbarkeit die Anzahl der Instanzen eines Tasks  $\tau_i$  in der Level- $i$  Busy Period bestimmt werden muss:

$$N_i = \min\{k | W_i(k, kT_i) \leq 1\}. \quad (4.21)$$

Die notwendige und hinreichende Bedingung für die Planbarkeit eines Task-Sets mit fixen Prioritäten ist laut [Leh90] für einen Task  $\tau_i$ :

$$\max_{k \leq N_i} W_i(k, (k-1)T_i + D_i) \leq 1, \quad (4.22)$$

mit

$$W_i(k, x) = \min_{t \leq x} \left( \frac{\sum_{j=1}^{i-1} C_j \left\lceil \frac{t}{T_j} \right\rceil + kC_i}{t} \right). \quad (4.23)$$

Die Bedingung in Gleichung 4.22 muss für jeden Task eines Task-Sets bestehend aus  $n$  Tasks erfüllt sein, das heißt ein Task-Set mit fixen Prioritäten und beliebigen Deadlines ist planbar, wenn

$$\max_{1 \leq i \leq n} \left\{ \max_{k \leq N_i} W_i(k, (k-1)T_i + D_i) \right\} \leq 1. \quad (4.24)$$

#### 4.1.2 Earliest Deadline First

##### Simulation

Auch für den EDF-Algorithmus ist die Simulation über die Hyperperiode  $H$ , die in Gleichung 4.1 definiert ist, eine exakte Analyse der Planbarkeit eines Task-Sets. Werden alle Tasks zum gleichen Zeitpunkt  $t = 0$  ausgelöst und halten ihre Deadlines bis zur Hyperperiode ein, so werden auch alle zukünftigen Deadlines eingehalten und das Task-Set ist planbar.

##### Prozessorauslastung

In [LL73] wird eine auf der Prozessorauslastung basierende Analyse der Planbarkeit eines Task-Sets mit dem EDF-Algorithmus vorgestellt. Ein Set aus periodischen, unabhängigen und präemptiven Tasks mit relativen Deadlines gleich der Perioden, das heißt  $D_i = T_i$  für einen Task  $\tau_i$ , ist demnach mit dem EDF-Algorithmus auf einem Prozessor planbar, wenn

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1, \quad (4.25)$$

das bedeutet die gesamte Prozessorauslastung nicht mehr als 100 % beträgt. Gleichung 4.25 ist nur eine hinreichende Bedingung für die Planbarkeit eines Task-Sets mit dem EDF-Algorithmus für den Fall  $D_i = T_i$ .

Eine exakte Schedulability-Analyse für Task-Sets mit  $D_i > T_i$  für einen Task  $\tau_i$  ist in [Liu00] gegeben:

$$U = \sum_{i=1}^n \frac{C_i}{\min(D_i, T_i)} \leq 1. \quad (4.26)$$

Für Task-Sets mit Tasks, deren Deadlines kürzer als die Task-Perioden sind, das bedeutet  $D_i \leq T_i$  für einen Task  $\tau_i$ , ist Gleichung 4.26 nur eine hinreichende Bedingung zur Bestimmung der Planbarkeit.

### Prozessor-Bedarf

Ein anderer Ansatz für die Schedulability-Analyse des EDF ist bekannt als Prozessor-Bedarfs-Kriterium (engl. *processor demand criterion*). In [LM80] wird festgestellt, dass die Überprüfung aller absoluten Deadlines im Intervall  $[0, 2H]$  ausreichend ist, um die Schedulability eines Task-Sets festzustellen, wenn alle Tasks zum gleichen Zeitpunkt  $t = 0$  gestartet werden. Dabei bezeichnet  $H$  die Hyperperiode, wie sie in Gleichung 4.1 definiert ist. Basierend auf dieser Annahme wurde das Prozessor-Bedarfs-Kriterium entwickelt, das im Gegensatz zu Gleichung 4.26 eine notwendige und hinreichende Bedingung für Task-Sets mit  $D_i \leq T_i$  darstellt [BRH90]. Der Prozessor-Bedarf eines Tasks ist die Menge an Rechenzeit, welche von den Instanzen eines Tasks benötigt wird, die sowohl ihre Startzeit als auch ihre Deadline in einem Intervall der Länge  $t$  haben [BMR90]:

$$h(t) = \sum_{i=1}^n \max \left\{ 0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right\} C_i. \quad (4.27)$$

Für ein Task-Set mit Tasks, deren Deadlines unabhängig von den Task-Perioden sind, muss mindestens ein Intervall mit der Länge der größten relativen Deadline  $D_{\max}$  des Task-Sets betrachtet werden, um die Planbarkeit zu bestimmen. Damit wird Gleichung 4.27 folgendermaßen vereinfacht:

$$h(t) = \sum_{i=1}^n \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i. \quad (4.28)$$

Allgemein ist ein Set aus synchronen, periodischen Tasks und  $D_i \leq T_i$  mit dem EDF-Algorithmus planbar, wenn

$$U \leq 1, \quad \forall t > 0 : \quad h(t) \leq t. \quad (4.29)$$

Die Größe des zu überprüfenden Intervalls kann verkleinert werden, was die Komplexität des Schedulability-Test in Gleichung 4.29 deutlich reduziert. Laut [BMR90] ist es ausreichend, ein Intervall der Länge  $t_{ub}$  zu überprüfen:

$$1U \leq 1, \quad \forall t < t_{ub} : \quad h(t) \leq t \quad (4.30)$$

$$t_{ub} = \min \left\{ H + \max_{1 \leq i \leq n} \{D_i\}, \max_{1 \leq i \leq n} \{T_i - D_i\} \frac{U}{1 - U} \right\}.$$

In [RCM96] wird eine noch niedrigere obere Grenze für das zu testende Intervall angegeben:

$$t_{ub}^* = \frac{\sum_{i=1}^n (1 - \frac{D_i}{T_i}) C_i}{1 - U}. \quad (4.31)$$

Außerdem wird ein Schedulability-Test beschrieben, der die obere Grenze in Gleichung 4.31 mit der Hyperperiode  $H$  des Task-Sets kombiniert:

$$\forall t < \min\{t_{ub}^*, H\}. \quad (4.32)$$

Zur Bestimmung einer oberen Grenze des Testintervalls wird auch das Konzept der Busy Period verwendet [Spu96] [RCM96]. Für die Schedulability-Analyse eines periodischen Task-Sets, das synchron zum Zeitpunkt  $t = 0$  gestartet wird, muss nur die erste Busy Period  $[0, t_{busy}]$  der ersten Hyperperiode  $H$  betrachtet werden. Die Länge der Busy Period  $t_{busy}$  wird rekursiv berechnet:

$$\begin{aligned} 1t^0 &= \sum_{i=1}^n C_i, \\ t^{k+1} &= W(t^k) = \sum_{i=1}^n \left\lceil \frac{t^k}{T_i} \right\rceil C_i, \\ \text{Stop} : t^{k+1} &= t^k = t_{busy}. \end{aligned} \quad (4.33)$$

Damit ergibt sich, dass ein Set aus periodischen Tasks, die gleichzeitig zum Zeitpunkt  $t = 0$  gestartet werden, durch den EDF-Algorithmus planbar ist, wenn

$$U \leq 1, \quad \forall t \leq t_{busy} : h(t) \leq t. \quad (4.34)$$

Sowohl in [Hoa+06], als auch in [ZB09] wird eine kombinierte Schedulability-Analyse aus den Gleichungen 4.30 und 4.34 abgeleitet, die beide Ergebnisse zusammenfasst:

$$\begin{aligned} 1U &\leq 1, \quad \forall t \in P : h(t) \leq t \\ P &= \{d_k | d_k = kT_i + D_i \wedge d_k < \min\{t_{ub}, t_{busy}\}, k \in \mathbb{N}\}. \end{aligned} \quad (4.35)$$

Ein deutlich schnellerer Schedulability-Test basierend auf dem Prozessor-Bedarf wird in [ZB09] präsentiert. Die Quick convergence Processor-demand Analysis (QPA) bietet einen schnellen und einfachen Test für den EDF-Algorithmus und ist dabei notwendig und hinreichend. QPA funktioniert, indem mit einem Wert von  $t$  in der Nähe von  $\min\{t_{ub}, t_{busy}\}$  begonnen wird, und ein einfacher Ausdruck in Richtung 0 durchlaufen wird. Ein Task-Set ist planbar, wenn  $U \leq 1$  und das Ergebnis des iterativen Algorithmus, den Listing 4.1 präsentiert,  $h(t) \leq d_{\min}$  ist.

## Listing 4.1: Quick convergence Processor-demand Analysis (QPA)

```
t ← max{di | di < min{tub, tbusy}}  
while h(t) ≤ t ∧ h(t) > dmin do  
    if h(t) < t then  
        t ← h(t)  
    else  
        t ← max{di | di < t}  
if h(t) ≤ dmin then  
    Das Task-Set ist planbar.  
else  
    Das Task-Set ist nicht planbar.
```

---

Der Schedulability-Test nach [ZB09] besteht aus vier Schritten:

1. Berechnung der Prozessorauslastung  $U$  und Test nach Gleichung 4.25.
2. Berechnung der oberen Grenze  $t_{ub}$  nach Gleichung 4.30.
3. Berechnung der oberen Grenze  $t_{busy}$  nach Gleichung 4.33.
4. Ausführen des Algorithmus in Listing 4.1.

Auch hier wird eine verbesserte obere Grenze  $t_{ub}^{**}$  angegeben. Damit ergibt sich folgender Schedulability-Test:

$$\begin{aligned} 1U &\leq 1, \quad \forall t \in P: \quad h(t) \leq t \\ P &= \{d_k | d_k = kT_i + D_i \wedge d_k < t_{ub}^{**}, k \in N\} \\ t_{ub}^{**} &= \max \left\{ (D_1 - T_1), \dots, (D_n - T_n), \frac{\sum_{i=1}^n (T_i - D_i)U_i}{1 - U} \right\}. \end{aligned} \tag{4.36}$$

## 4.2 Schedulability-Analyse mit Machine Learning

### 4.2.1 Veröffentlichungen

Ein von Aytug et al. bereits im Jahr 1994 veröffentlichter Artikel hat den Nutzen und die Bedeutsamkeit von maschinellem Lernen bzw. der künstlichen Intelligenz allgemein im Bereich des Scheduling zum Thema [Ayt+94]. Zudem wird ein Überblick über zu dieser Zeit bekannte Arbeiten zum Thema Machine Learning im Scheduling-Umfeld gegeben.



In [Nem+17] wird festgestellt, dass heterogene Systeme<sup>1</sup> neue CPU-Scheduling-Techniken notwendig machen und durch exaktes Abschätzen der Performance von Anwendungen auf den verschiedenen Ressourcen das Scheduling deutlich verbessert wird. Deshalb stellen Nemirovsky et al. in ihrem Artikel ein Durchsatz-maximierendes, heterogenes CPU-Scheduling-Modell vor, das maschinelles Lernen nutzt. Genauer wird ein künstliches neuronales Netz verwendet, um die Ausführung mehrerer Threads auf verschiedenen Systemressourcen vorherzusagen. Entsprechend werden die Threads auf die Ressourcen verteilt und es wird für jede Ressource ein Ablaufplan erstellt. Damit wird eine Durchsatz-Verbesserung von 25 % bis 31 % gegenüber herkömmlichen Scheduling erreicht. Allerdings wird das neuronale Netz hier nur zur Unterstützung des Schedulers verwendet, um genauere Daten über die Threads zu erhalten, ersetzt diesen aber nicht.

In [Say+17] werden dynamische, heterogene Architekturen, sogenannte Composite Cores Architectures (CCAs), betrachtet. Dabei wird zur Laufzeit der am besten geeignete Kern für jede Anwendung erstellt, indem entweder Kerne zu einem größeren Kern zusammengefügt werden oder ein großer Kern in mehrere kleinere Kerne zerlegt wird. Der Artikel beleuchtet die dadurch auftretenden Herausforderungen für die Planung von Multithread-Anwendungen auf solchen CCA-Architekturen. Der zur Lösung dieser Herausforderung vorgeschlagene Ansatz basiert auf maschinellem Lernen und wird zur Vorhersage der richtigen Einstellungen für die Ausführung der Multithread-Anwendungen auf der CCA-Architektur, wie zum Beispiel Kern-Typ, Spannung oder Frequenz, verwendet. Die Autoren implementieren fünf bekannte Machine Learning Modelle und beobachten mit mehrschichtigen, neuronalen Netzen die höchste Genauigkeit. Allerdings ist die Leistung solcher Netze bezogen auf den Implementierungsaufwand deutlich schlechter als bei einfacheren Modellen wie Regression oder Entscheidungsbäume. Auch hier wird maschinelles Lernen nur zur Verbesserung des Scheduling in Hinblick auf die Energieeffizienz verwendet und ersetzt nicht das eigentliche Planungsverfahren.

Auch für harte Echtzeitsysteme wird in [Dav+98] ein Ansatz gezeigt, der ein neuronales Netz für die Verteilung von Tasks auf Prozessorkerne verwendet. Davoli et al. präsentieren ein mathematisches Modell, das die Planbarkeit eines Sets aus parallelen Programmen überprüft. Um ein neuronales Netz in diesem Zusammenhang anzuwenden, wird das Schedulability-Problem in ein Optimierungs-Problem umgewandelt. Simulationen zeigen vielversprechende Ergebnisse im Bezug auf die Genauigkeit und die Komplexität. Der Artikel bildet am ehesten einen Anhaltspunkt für diese Arbeit, da hier der Scheduler durch das neuronale Netz ersetzt wird. Allerdings wird nicht nur die Lauffähigkeit des Task-Sets bestimmt, sondern auch ein lauffähiger Ablaufplan erstellt. Außerdem basiert der Ansatz auf einem EDF-Scheduler, wohingegen in dieser Arbeit ein kombinierter FP-/EDF-Scheduler verwendet wird.

---

<sup>1</sup>Systeme mit unterschiedlichen Recheneinheiten wie zum Beispiel General Purpose Processors (GPPs), digitale Signalprozessoren (engl. *digital signal processors*, *DSPs*) oder Field Programmable Gate Arrays (FPGAs).

In [ZD95] werden Methoden des verstärkenden Lernens für die Maschinenbelegungsplanung (engl. *job shop scheduling*) angewendet. Ein neuronales Netz lernt mit Hilfe von Temporal Difference Learning, einer Methode des verstärkenden Lernens, eine Evaluierungsfunktion, mit der gute Lösungen für neue Scheduling Probleme gefunden werden. Die Ergebnisse zeigen, dass der Ansatz eine deutlich bessere Leistung als existierende Algorithmen in diesem Bereich aufweist. Deshalb wird verstärkendes Lernen als eine Möglichkeit angesehen, um schnell qualitativ hochwertige Lösungen für Scheduling-Probleme zu bestimmen. Der in diesem Artikel vorgestellte Ansatz ersetzt zwar den Scheduler, allerdings wird das neuronale Netz nur zum Finden des besten Ablaufplanes benutzt und nicht um die Planbarkeit eines Task-Sets zu überprüfen.

Ein rekurrentes neuronales Netz wird in [GB16] verwendet um Echtzeit-Scheduling-Probleme mit stückweise linearen Zielvorgaben zu lösen. Dabei hat jeder Job in einem Set eigene Optimierungsziele, zum Beispiel die Antwortzeit, den Energieverbrauch oder die Laufzeit. Ziel ist es, den besten Ablaufplan zu finden, der die Gesamtnutzenfunktion maximiert, das heißt die Optimierungsziele der einzelnen Jobs am besten erfüllt. Für zufällig generierte Sets ist der Algorithmus optimal, wenn das Set nicht überladen ist. Bei Überlastung übertrifft der Algorithmus existierende Scheduling-Strategien. Obwohl die Veröffentlichung sogar ein rekurrentes neuronales Netz nutzt, ist sie für diese Arbeit nicht von Nutzen, da lediglich der Scheduling-Algorithmus ersetzt wird. Eine Schedulability-Analyse wird mit dem neuronalen Netz jedoch nicht durchgeführt.

Die durchgeführte Literaturrecherche im Bereich maschinelles Lernen und Schedulability-Analyse ergab nur sehr wenige Veröffentlichungen. Gerade im Zusammenhang mit der Planbarkeitsanalyse wurde keine Anwendung von neuronalen Netzen oder Machine Learning Methoden gefunden. Im weiteren Feld des (Echtzeit-)Schedulings befassen sich die existierenden Arbeiten mit sehr spezifischen Problemen unter ganz bestimmten Annahmen und sind deshalb meist nicht allgemein anwendbar.

#### 4.2.2 Frühere Arbeiten am Lehrstuhl

Am Lehrstuhl für Betriebssysteme der Technischen Universität München sind im Rahmen des KIA4SM-Projektes bereits einige Arbeiten zum Thema maschinelles Lernen und Schedulability-Analyse vorhanden. Darin sind Untersuchungen von verschiedenen Verfahren und deren Nutzen für die Klassifizierung von Tasks und Task-Sets dokumentiert.

Robert Hamsch erforscht in seiner Bachelorarbeit die Möglichkeiten, die ein künstliches neuronales Netz im Bereich der Schedulability-Analyse bietet [Ham17]. Der Lernerfolg eines Deep Learning Netzes basiert auf der Größe der Datenmenge, weshalb Hamsch eine Komponente für die Generierung von Trainingsdaten entwickelt. Anschließend implementiert er eine Deep Learning Komponente für die Schedulability-Analyse, beginnend bei einem flachen Netz mit einer versteckten Schicht, hin zu einem Netz mit zwei versteckten Schichten. Experimente mit verschiedenen Aktivierungsfunktionen und Dimensionen der Neuronenschichten ergeben zum Schluss eine Genauigkeit von 75 % bis 80 %.

Das Ziel der Bachelorarbeit von Hans Rauer ist die Entwicklung eines vollständig trainierten, neuronalen Netzes, das die Ankunft von Softwarekomponenten, das heißt Tasks, an einer Steuereinheit (ECU) überwacht und die Wahrscheinlichkeit für die Lauffähigkeit des resultierenden Task-Sets bestimmt [Rau18]. Um ein solches Netz überhaupt trainieren zu können, ist eine große Menge an Daten notwendig, die Rauer mit Hilfe von Monte Carlo Simulation generiert. Anschließend implementiert Rauer ein sehr grundlegendes Modell eines rekurrenten neuronalen Netzes mit TensorFlow. Das Ergebnis seiner Arbeit ist ein Framework für die weitere Forschung an RNN-basierter Schedulability-Analyse. Die Arbeit von Rauer ist deshalb die Grundlage dieser Masterarbeit.

Das Thema der Masterarbeit von Robert Häcker ist die Vorhersage der Schedulability von Tasks mit Hilfe von Machine Learning Techniken aus dem Shallow Learning Bereich, das heißt neuronale Netze mit wenigen Schichten [Häc18]. Zum Trainieren der Shallow Learning Modelle ist ein möglichst realistischer und großer Datensatz notwendig, den Häcker durch eine eigene Komponente erzeugt. Anschließen gelingt es Häcker, fünf verschiedene Verfahren des maschinellen Lernens mit Hilfe des Scikit-learn Frameworks zu implementieren: Support Vector Machines, Entscheidungsbäume, k-nächste-Nachbarn, logistische Regression und Naive Bayes. Aufgrund der Definition dieser Verfahren ist es notwendig, für jede mögliche Größe der Task-Sets, der Einfachheit halber beschränkt auf ein bis fünf Tasks, ein eigenes Modell zu trainieren. Nach Anpassung der Modelle an die Daten erzielt Häcker mit einem Entscheidungsbaum die besten Vorhersagen mit 83,6 % Genauigkeit, was jedoch für sicherheitskritische Systeme nicht ausreichend ist.



## 5 Gegebene Datenbank mit den Task-Sets

Das Training von Machine Learning Algorithmen und neuronalen Netzen setzt eine große Menge an Daten voraus. Dabei gilt im Allgemeinen: je mehr Daten für das Training verwendet werden, desto besser sind die Ergebnisse, die das resultierende Modell liefert [Sca18]. Aber nicht nur die Menge ist entscheidend, sondern auch die Qualität der Trainingsdaten, zum Beispiel die Ähnlichkeit mit der realen Welt. Bei der Klassifikation muss zusätzlich darauf geachtet werden, dass genügend Beispiele für jede Kategorie vorhanden sind.

Die Generierung von Trainingsdaten für das Machine Learning ist ein wichtiger Teil des *MaLSAMi*-Projektes und ist zu einem großen Teil das Werk von Robert Häcker [Häc18]. Eine Beschreibung der Toolchain des *MaLSAMi*-Projektes findet sich bereits in Kapitel 2.1 und ist in Abbildung 2.1 gezeigt. Ein Task-Generator erstellt Tasks, die ein Task-Set-Generator zur Erstellung von Task-Sets verwendet. Die erzeugten Task-Sets werden von einem Distributor auf eine oder mehrere Plattformen verteilt und dort ausgeführt. Die Plattformen sind eingebettete Systeme, die momentan durch Einplatinencomputer, auf denen das Betriebssystem des Lehrstuhls bestehend aus L4 Fiasco.OC und Genode läuft, repräsentiert werden. Die Ergebnisse über die Lauffähigkeit der Task-Sets werden anschließend in einer SQL-Datenbank gespeichert. Die so erstellte Datenbank ist die Grundlage dieser Masterarbeit und wird deshalb im nachfolgenden Kapitel genauer betrachtet.

### 5.1 Aufbau der gegebenen Datenbank

In dieser Arbeit wird die Version „panda\_v3“ der Datenbank verwendet. Wie der Name bereits vermuten lässt, werden die Task-Sets auf einem Cluster aus PandaBoards ausgeführt. Die Datenbank besteht aus drei Tabellen: *TaskSet*, *Task* und *Job*. Abbildung 5.1 zeigt die Struktur der Tabellen und die relationalen Beziehungen zwischen diesen.

#### Die Tabelle *TaskSet*

Die Tabelle *TaskSet* enthält alle erstellten Task-Sets. Jedes Task-Set hat dabei eine eindeutige *Set\_ID*. Die Spalte *Successful* enthält die Lauffähigkeit des Task-Sets, wobei 0 für „nicht lauffähig“ und 1 für „lauffähig“ steht. Ein Task-Set besteht aus mehreren Tasks, die über ihre IDs referenziert werden. *TASK1\_ID* speichert die ID des ersten Tasks, *TASK2\_ID* die ID des zweiten Tasks und so weiter. Über die Task-ID werden die Attribute eines Tasks aus der Tabelle *Task* bestimmt. Die Task-ID  $-1$  steht dabei für einen ungültigen Task und zeigt an, dass keine weiteren Tasks folgen. In der aktuellen Version

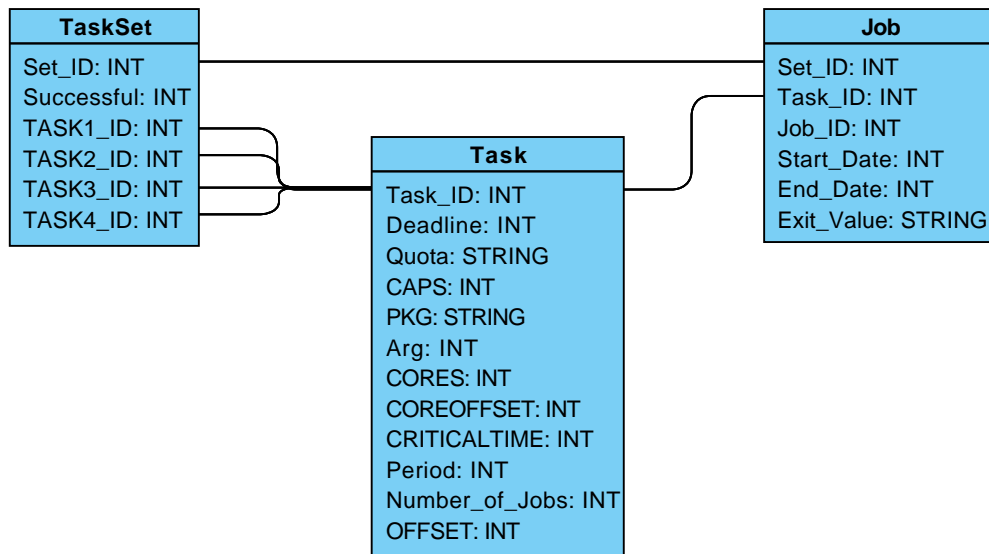


Abbildung 5.1: Struktur der gegebenen Datenbank und Zusammenhänge zwischen den Tabellen (Eigene Darstellung).

der Datenbank werden Task-Sets mit einer maximalen Größe von vier Tasks unterstützt. Tabelle 5.1 zeigt beispielhaft zwei Task-Sets wie sie in der Datenbank enthalten sind.

Tabelle 5.1: Zwei Beispiele für Task-Sets aus der Tabelle *TaskSet* (Eigene Darstellung).

Set_ID	Successful	TASK1_ID	TASK2_ID	TASK3_ID	TASK4_ID
46 429	1	104	49	227	−1
563 782	0	278	123	203	−1

### Die Tabelle Task

Die Tabelle *Task* enthält verschiedene Konfigurationen der Beispieltasks, die für die Erstellung von Task-Sets zur Verfügung stehen. Die Beispieltasks sind in Kapitel 2.2.1 und Tabelle 2.1 beschrieben. Jeder Task wird über die *Task\_ID* eindeutig identifiziert. Die restlichen Attribute eines Tasks sind in Tabelle 5.3 beschrieben. Tabelle 5.2 zeigt beispielhaft die Tasks der Task-Sets aus Tabelle 5.1.

### Die Tabelle Job

Ein Task wird nach einem bestimmten Intervall, der Periode, wiederholt ausgeführt. Eine Ausführung des Tasks bzw. der Task-Funktion wird als Job bezeichnet. Die Jobs

Tabelle 5.2: Beispiele für Tasks aus der Tabelle *Task* (Eigene Darstellung).

Task_ID	Priority	Deadline	Quota	CAPS	PKG	Arg
49	1	0	100 MB	235	pi	131 072
104	4	0	100 MB	235	cond_mod	22 876 792 454 961
123	4	0	100 MB	235	pi	2 097 152
203	1	0	100 MB	235	tumatmul	4096
227	2	0	100 MB	235	pi	32 768
278	4	0	100 MB	235	tumatmul	65 536

Task_ID	CORES	CORE-OFFSET	CRITICAL-TIME	Period	Number_-of_Jobs	OFFSET
49	2	1	5500	6000	7	0
104	2	1	6500	7000	3	0
123	2	1	7500	8000	3	0
203	2	1	500	1000	4	0
227	2	1	1500	2000	3	0
278	2	1	1500	2000	2	0

aller Tasks sind in der Tabelle *Job* gespeichert. Durch die Spalten *Set\_ID* und *Task\_ID* wird jeder Job eindeutig einem Task-Set und einem Task zugeordnet. Außerdem hat jeder Job eine eindeutige *Job\_ID*. Die Spalte *Start\_Date* enthält den Zeitpunkt, zu dem der Job gestartet wurde. Entsprechend speichert die Spalte *End\_Date* den Zeitpunkt, zu dem der Job beendet wurde. Dieser Zeitpunkt muss jedoch nicht mit dem Zeitpunkt übereinstimmen, an dem der Job vollständig ausgeführt wurde. Wird zum Beispiel die Criticaltime überschritten, wird der Job vor Vollendung abgebrochen und dieser Zeitpunkt als *End\_Date* abgespeichert. Ob ein Job erfolgreich ausgeführt wurde, wird der Spalte *Exit\_Value* entnommen, wobei *EXIT* für „erfolgreich“ und *EXIT\_CRITICAL* für „nicht erfolgreich“ steht.

## 5.2 Datenanalyse

Tabelle 5.4 gibt einen Überblick über die Verteilung der Task-Sets. Insgesamt stehen in der Datenbank 2 051 878 Task-Sets zur Verfügung, von denen ca. 78 % erfolgreich auf dem PandaBoard ES Cluster ausgeführt werden. Außerdem nimmt die Menge der Task-Sets mit der Anzahl an Tasks zu. Eine Ausnahme besteht für Task-Sets aus vier Tasks. Diese sind nicht in der Datenbank enthalten, da nur Task-Sets mit maximal drei Tasks generiert werden. Der Anteil der erfolgreichen Task-Sets bewegt sich bei allen Task-Sets zwischen 77 % und 87 %.

Tabelle 5.3: Beschreibung der Attribute eines Tasks (In Anlehnung an [Häc18, S. 9]).

Attribut	Beschreibung
Priority	Priorität des Tasks, notwendig für den FP-Scheduler.
Deadline	Deadline des Tasks, notwendig für den EDF-Scheduler.
Quota	Größe des Speichers in Byte der dem Task zur Verfügung steht.
CAPS	Wert für die Funktionalität des Tasks; die Funktionsweise dieses Wertes ist nicht eindeutig definiert.
PKG	Funktionsname des Tasks.
Arg	Argument, mit dem die Task-Funktion aufgerufen wird.
CORES	Anzahl der verfügbaren Kerne.
COREOFFSET	Offset des Kerns, auf dem der Task ausgeführt wird
CRITICALTIME	Zeitpunkt bis wohin der Task Ressourcen erhält, bei Überschreitung wird der Task beendet; entspricht der Deadline des Tasks.
Period	Periode des Tasks; Intervall in Millisekunden in dem der Task wiederholt wird.
Number_of_Jobs	Anzahl der Jobs, das heißt wie oft der Task wiederholt wird.
OFFSET	Wartezeit in Millisekunden vor dem Start des Tasks.

Tabelle 5.4: Verteilung der Task-Sets (Eigene Darstellung).

Größe des Task-Sets	erfolgreich	nicht erfolgreich	Summe	Anteil erfolgreich	Anteil nicht erfolgreich
1 Task	327	73	400	81,75 %	18,25 %
2 Tasks	46 102	7199	53 301	86,49 %	13,51 %
3 Tasks	1 549 477	448 700	1 998 177	77,54 %	22,46 %
4 Tasks	0	0	0	0 %	0 %
<b>Insgesamt</b>	1 595 906	455 972	2 051 878	77,78 %	22,22 %



Wie bereits im vorherigen Kapitel gezeigt, haben die Tasks verschiedene Attribute. Allerdings werden in der aktuellen Version der Datenbank nicht alle Attribute mit nutzbaren Werten belegt. So haben folgende Attribute bei allen generierten Tasks den gleichen Wert:

- Deadline = 0,
- Quota = 100M,
- CAPS = 235,
- CORES = 2,
- COREOFFSET = 1,
- OFFSET = 0.

Dies ist insbesondere dem Umstand geschuldet, dass die Datengenerierungskomponente laufend weiterentwickelt wird und noch lange nicht ihr volles Potential ausschöpft, oder die Bedeutung einiger Attribute noch nicht sicher festgelegt ist. Deshalb werden auch die genutzten Attribute nur mit wenigen, diskreten Werten belegt. Obwohl die Priorität eines Tasks theoretisch einen Wert aus dem Bereich  $[0, \dots, 127]$  annimmt, haben die 383U Tasks in der aktuellen Datenbank allesamt Prioritäten aus dem deutlich eingeschränkten Bereich  $[0, \dots, 5]$ . Der Vollständigkeit halber sei wiederholt, dass auch das *PKG*-Attribut der Tasks nur aus der Menge der Beispieltasks entnommen wird, demnach ['cond\_mod', 'hey', 'pi', 'tumatmul']. Die *Criticaltime* aller Tasks ist in der aktuellen Version ein Wert aus  $[500, 1500, 2500, 3500, 4500, 5500, 6500, 7500]$  und liegt immer genau 500 ms vor der entsprechenden Periode aus dem Wertebereich  $[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000]$ . Auch die Anzahl der Wiederholungen (*Number\_of\_Jobs*) bewegt sich momentan nur innerhalb eines relativ kleinen Wertebereiches von  $[0, \dots, 8]$ . Da das Ziel der Arbeit die Abschätzung des Nutzens eines neuronalen Netzes für die Schedulability-Analyse ist und außerdem keine andere Datenbank zur Verfügung steht, sind diese Einschränkungen hinnehmbar.



## 6 Traditionelle Schedulability-Analyse

Die klassischen Schedulability-Analyseverfahren sind in dem Projekt *traditional-SA*<sup>1</sup> verwirklicht. Es werden alle für das vorhandene System relevanten Methoden aus Kapitel 4 implementiert. Der Aufbau des Projektes ist in Abbildung 6.1 zu sehen. Obgleich das Zielsystem Multi-Prozessor-Plattformen sind, wird die nachfolgend gezeigte Komponente nur für ein Einprozessor-System entwickelt. Der Gedanke dahinter ist, dass aktuell nur ein Prozessor zur Ausführung genutzt wird und alle anderen Prozessoren eines Multi-Prozessor-Systems für Migrationskomponenten reserviert werden. Diese Einschränkung reduziert die Problemstellung auf ein Einprozessor-Problem.

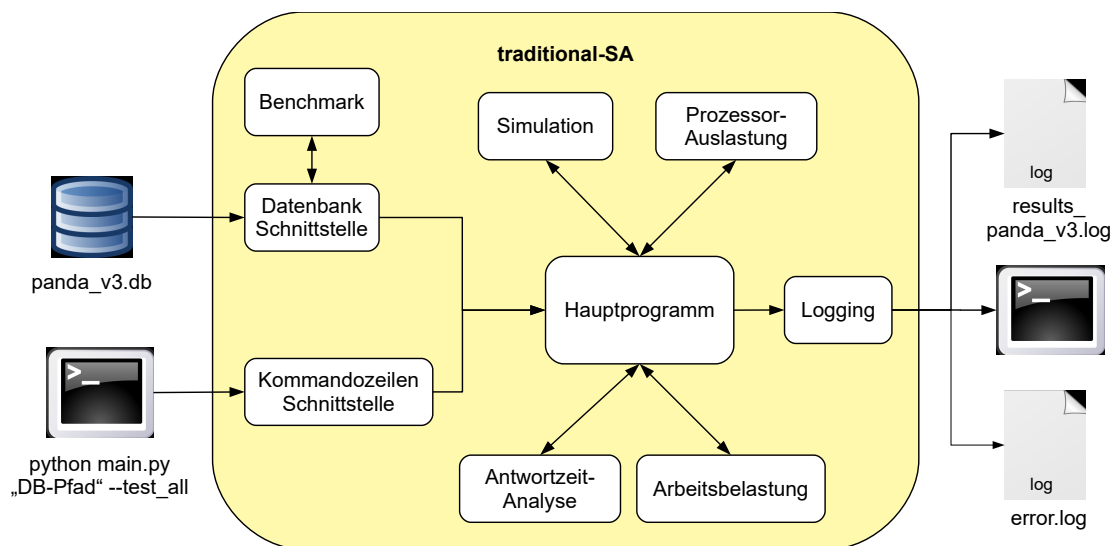


Abbildung 6.1: Blockdiagramm des *traditional-SA*-Projektes, der Komponente für die traditionelle Schedulability-Analyse (Eigene Darstellung).

Der Kern des Projektes ist das Hauptprogramm, das die verschiedenen Schedulability-Analyseverfahren für die Task-Sets aus der Datenbank durchführt und die Ergebnisse abspeichert. Das Hauptprogramm wird über die Kommandozeile gestartet, wobei angegeben werden muss, wo die Daten gespeichert und welche Analyseverfahren durchgeführt werden. Diese Informationen werden durch die Kommandozeilenschnittstelle eingelesen und an das Hauptprogramm weitergegeben.

<sup>1</sup><https://github.com/TatjanaUtz/traditional-SA>

An dem über die Kommandozeile festgelegten Speicherort befindet sich die Datenbank, wie sie in Kapitel 5 gezeigt ist. Die Datenbankschnittstelle beinhaltet Funktionen, um Daten aus der Datenbank zu lesen bzw. zu schreiben und ermöglicht dem Hauptprogramm so den Zugriff auf die Task-Sets. Da die Datenbank für die Tasks kein Attribut für die Ausführungszeit speichert, wird ein Benchmark-Modul entwickelt, das die durchschnittliche Laufzeit für jeden Task bestimmt und in der Datenbank speichert.

Alle implementierten Schedulability-Analyseverfahren sind in der konstanten Liste `VALID_SA` in der Datei `main.py` gespeichert und werden in vier Kategorien unterteilt:

- Simulation,
- Prozessorauslastungs-Tests,
- Antwortzeit-Analyse,
- Arbeitsbelastungs-Tests.

Für die Kommunikation des Hauptprogramms mit dem Benutzer ist das Logging-Modul zuständig. Mit Hilfe dieses Moduls werden unter anderem die Ergebnisse in eine Log-Datei geschrieben und über die Konsole ausgegeben, sowie eventuell aufgetretene Fehler in der Datei `error.log` gespeichert.

### 6.1 Hauptprogramm

Das Hauptprogramm ist durch die Methode `main()` in der gleichnamigen Datei implementiert. Der Python-Code des Programms ist in Listing 6.1 gezeigt.

Listing 6.1: Hauptprogramm des *traditional-SA*-Projektes

```
def main():
    # Kommandozeilenargumente lesen und verarbeiten
    db_dir, db_name, tests_todo = command_line_interface.read_input()

    # Logger erstellen und initialisieren
    logger = logging.init_logging(db_dir, db_name)

    if tests_todo is not None: # min. ein Test wird durchgeführt
        dataset = load_dataset(db_dir, db_name) # Daten laden

        for test in tests_todo: # Iteration über die To-Do-Liste
            results = test_dataset(dataset, test) # Test durchführen
            logging.log_results(test.__name__, results) # Ergebnis loggen
```

Zunächst werden durch die Kommandozeilenschnittstelle das Verzeichnis und der Name der Datenbank sowie die auszuführenden Tests eingelesen. Dann wird mit Hilfe des Logging-Moduls ein Logger erstellt und initialisiert. Wenn über die Kommandozeile mindestens ein gültiger Test eingegeben wird, werden die Daten aus der Datenbank geladen, das heißt alle dort verfügbaren Task-Sets. In einer Schleife über alle durchzuführenden Tests wird der Datensatz mit dem jeweiligen Verfahren getestet und die Ergebnisse gespeichert bzw. ausgegeben.

### Laden der Daten

Um den Datensatz aus der Datenbank zu laden, wird die Methode `load_dataset()` erstellt, die als Eingaben das Verzeichnis und den Namen der Datenbank erhält. Darin wird zunächst versucht, ein Objekt der Klasse `Database` zu erstellen, das den Zugriff auf die Datenbank ermöglicht, wie in Listing 6.2 zu sehen. Tritt dabei ein Ausnahmefehler auf, zum Beispiel weil eine notwendige Tabelle fehlt, wird dieser abgefangen und als Fehler geloggt. Wird das `Database`-Objekt erfolgreich erstellt, wird der Datensatz durch `dataset = my_database.read_table_taskset()` aus der Datenbank gelesen und anschließend zurückgegeben.

Listing 6.2: Erstellen eines Database-Objektes

```
try:
    my_database = Database(db_dir, db_name)
except ValueError as val_err:
    logger.error("Could not create Database-object:%s", format(val_err))
    return None
```

---

### Datensatz testen

Die Methode `test_dataset()` führt die übergebene Schedulability-Analysemethode für den übergebenen Datensatz durch und ist in Listing 6.3 gezeigt. Dafür wird in einer Schleife über alle Task-Sets die Analyse-Funktion ausgeführt und das Ergebnis mit dem echten Ergebnis, das in der Datenbank gespeichert ist, verglichen. Sind alle Task-Sets getestet, werden die Ergebnisse, das heißt die Anzahl der richtig bzw. falsch eingeschätzten Task-Sets, in ein Dictionary gespeichert und zurückgegeben.

## 6.2 Kommandozeilenschnittstelle

Die *traditional-SA*-Komponente wird über die Kommandozeile gestartet. Dabei muss der Pfad zu der verwendeten Datenbank übergeben werden, da die Komponente ohne

## Listing 6.3: Testen eines Datensatzes

```
def test_dataset(dataset, function):  
    # Variablen für die Ergebnisse der Schedulability-Analyse  
    true_positive, false_positive, true_negative, false_negative = 0, 0, 0, 0  
  
    # Datensatz mit der Schedulability-Analysemethode testen  
    start_time = time.time() # Startzeit  
    for taskset in dataset: # Iteration über alle Task-Sets  
        schedulability = function(taskset) # Planbarkeit überprüfen  
        real_result = taskset.result # reales Ergebnis des Task-Sets  
  
        # Testergebnis mit realem Ergebnis vergleichen  
        if schedulability is True and real_result == 1: # tp  
            true_positive += 1  
        elif schedulability is True and real_result == 0: # fp  
            false_positive += 1  
        elif schedulability is False and real_result == 1: # fn  
            false_negative += 1  
        elif schedulability is False and real_result == 0: # tn  
            true_negative += 1  
    end_time = time.time() # Endzeit  
  
    # Dictionary mit dem Ergebnis des Tests erstellen und zurückgeben  
    result_dict = {'tp': true_positive, 'fp': false_positive, 'tn':  
        true_negative, 'fn': false_negative, 'time': end_time-start_time}  
    return result_dict
```

---

diese Information nicht arbeitet. Mit Hilfe von weiteren Argumenten wird festgelegt, welche Schedulability-Tests durchgeführt werden.

Tabelle 6.1: Bedeutung der Kommandozeilen-Argumente des *traditional-SA*-Projekts (Eigene Darstellung).

Argument	Beschreibung
<code>-h, --help</code>	zeigt die Hilfe an
<code>--test_all</code>	alle Schedulability-Analyseverfahren
<code>-s, --simulation</code>	Simulation
<code>-u, --utilization</code>	Prozessorauslastungs-Tests
<code>-rta, --response_time_analysis</code>	Antwortzeit-Analyse
<code>-w, --workload</code>	Arbeitsbelastungs-Tests

Wie Tabelle 6.1 zeigt ist es möglich, alle Tests auf einmal durchzuführen, oder nur die Tests einer oder mehrerer Kategorien. Einzelne Verfahren sind jedoch nicht auswählbar, sondern immer nur eine Gruppe von Verfahren, die auf dem selben Prinzip, wie zum Beispiel der Prozessorauslastung, beruhen. Um alle Tests durchzuführen, lautet der Befehl unter Windows beispielsweise

```
python main.py "C:\...\panda_v3.db" --test_all
```

oder um nur die Simulation auszuführen:

```
python main.py "C:\...\panda_v3.db" --simulation
```

Die Kommandozeilenschnittstelle ist in der Datei `command_line_interface.py` implementiert. Die dort definierte Methode `read_input()` liest die Argumente von der Kommandozeile ein und verarbeitet diese.

Dafür wird zunächst durch die Methode `_create_argparser()` ein Objekt der Python-Klasse `ArgumentParser` erstellt, mit dem es möglich ist, Argumente von der Kommandozeile einzulesen. Dann werden die notwendigen und optionalen Argumente definiert. Ein notwendiges Argument wird durch die Methode

```
parser.add_argument("db_path", help="Pfad zur Datenbank")
```

hinzugefügt. In diesem Beispiel wird ein Argument namens `db_path` hinzugefügt. Mit `help` wird eine kurze Beschreibung des Arguments gespeichert, die dann angezeigt wird, wenn die `main()`-Methode mit der Option `-h` bzw. `--help` aufgerufen wird. Optionale Argumente werden durch

```
parser.add_argument("-s", "--simulation", help="Simulation durchführen",
                    action="store_true")
```

hinzugefügt. Bei diesem Beispiel wird eine Option mit dem Namen `simulation` hinzugefügt, die auch über die Abkürzung `-s` verfügbar ist. Über den Parameter `action` wird

eingestellt, dass kein zusätzlicher Wert eingegeben werden muss, sondern gespeichert wird, ob die Option gewählt ist oder nicht.

Die Methode `_create_argparser()` gibt den erstellten Parser an die Methode `read_input()` zurück, die dann mit

```
args = parser.parse_args()
```

die Argumente von der Kommandozeile einliest. Beispielsweise ist der übergebene Datenbank-Pfad dann durch `args.db_path` verfügbar und wird in einen Verzeichnispfad und den Namen der Datenbank aufgespalten.

Anschließend wird eine leere (To-Do-)Liste erstellt, in die alle durchzuführenden Tests gespeichert werden. Nach und nach werden alle Optionen überprüft und falls eine Option ausgewählt ist, werden die entsprechenden Schedulability-Analyseverfahren zur To-Do-Liste hinzugefügt. So werden zum Beispiel bei gesetztem `test_all`-Flag alle implementierten Tests in die To-Do-Liste aufgenommen. Am Ende wird der Verzeichnispfad und der Name der Datenbank sowie die To-Do-Liste zurückgegeben.

## 6.3 Datenbankschnittstelle

Die Datenbankschnittstelle (`database_interface.py`) beinhaltet alle notwendigen Klassen und Methoden, um mit der in Kapitel 5 vorgestellten Datenbank zu arbeiten. Hierfür werden drei Klassen erstellt: `Task`, `Taskset` und `Database`. Mit Hilfe der ersten beiden Klassen lassen sich Tasks und Task-Sets in einer einheitlichen Form darstellen. Die dritte Klasse ist die eigentliche Schnittstelle zur Datenbank, da hier die Verbindung zur Datenbank gespeichert wird und Methoden zum Lesen und Schreiben der Tabellen implementiert sind. Abbildung 6.2 stellt die Klassendiagramme der erstellten Klassen dar, die nachfolgend näher beschrieben werden.

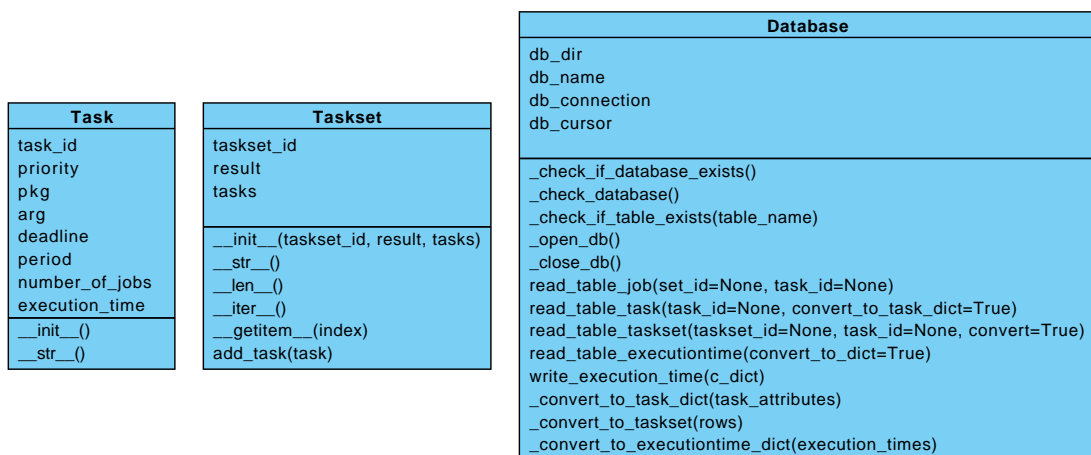


Abbildung 6.2: Klassendiagramme der Datenbankschnittstelle (Eigene Darstellung).



### 6.3.1 Die Klasse Task

Die Klasse Task repräsentiert einen Task. Momentan werden nur die Attribute ID, Priorität, PKG, Argument, Deadline, Periode, Anzahl der Jobs und Ausführungszeit unterstützt. Das Klassendiagramm ist in Abbildung 6.2 links zu sehen.

Der Konstruktor (`__init__()`) initialisiert die Attribute der Klasse mit den übergebenen Werten. Aus Gründen der Übersichtlichkeit werden die Eingabeparameter im Klassendiagramm weggelassen. Wenn dem Konstruktor keine Deadline übergeben wird, wird die Deadline mit der Periode des Tasks initialisiert. Die Methode `__str__()` gibt die String-Darstellung eines Tasks zurück.

### 6.3.2 Die Klasse Taskset

Die Klasse Taskset stellt ein Task-Set dar und hat deshalb die drei Attribute `taskset_id`, `result` und `tasks`. Die ersten beiden Attribute stehen für die entsprechenden Spalten *Set\_ID* und *Successful* der Tabelle *TaskSet*, wie sie in Kapitel 5.1 beschrieben ist. Das dritte Attribut ist eine Liste und enthält alle Tasks des Task-Sets als Objekte der Klasse Task. Abbildung 6.2 Mitte zeigt das Klassendiagramm.

Der Konstruktor (`__init__()`) initialisiert die Attribute mit den Eingabeparametern und sortiert die Tasks nach aufsteigenden Prioritäten. Die Methode `__str__()` gibt die String-Darstellung eines Taskset-Objekts zurück. Die Methoden `__len__()`, `__iter__()` und `__getitem__()` geben die Länge des Task-Sets, die der Anzahl der Tasks entspricht, einen Iterator für die Task-Liste und den Task mit dem Index `index` zurück. Um dem Task-Set einen Task hinzuzufügen wird die Methode `add_task()` erstellt. Diese hängt den übergebenen Task an die Task-Liste an und sortiert die Tasks dann nach Prioritäten.

### 6.3.3 Die Klasse Database

Die Klasse Database repräsentiert eine Datenbank und speichert alle notwendigen Eigenschaften, um mit der Datenbank zu arbeiten. Das Klassendiagramm ist in Abbildung 6.2 rechts dargestellt. Die Klasse hat folgende Attribute:

`db_dir` – Verzeichnis, in dem sich die Datenbank befindet,

`db_name` – Name der Datenbank bzw. der Datenbank-Datei,

`db_connection` – Verbindung zur Datenbank,

`db_cursor` – Cursor zum Arbeiten mit der Datenbank.

#### Konstruktor

Die Methode `__init__()` ist der Konstruktor der Klasse Database und initialisiert alle Attribute mit den übergebenen bzw. anderen sinnvollen Werten, wie sie in Listing 6.4 zu sehen sind. Anschließend wird überprüft, ob die durch `db_dir` und `db_name` definierte

Datenbank existiert. Zuletzt wird verifiziert, dass die notwendigen Tabellen *Job*, *Task*, *TaskSet* und *ExecutionTime* verfügbar sind.

Listing 6.4: Konstruktor der Klasse Database

```
def __init__(self, db_dir, db_name):
    self.db_dir = db_dir # Verzeichnis der Datenbank
    self.db_name = db_name # Name der Datenbank
    self.db_connection = None # Verbindung zur Datenbank
    self.db_cursor = None # Cursor für die Arbeit mit der Datenbank

    self._check_if_database_exists() # Existenz der Datenbank überprüfen

    # Existenz aller notwendigen Tabellen überprüfen
    self._check_database()
```

---

### Datenbank und Tabellen prüfen

Die Methode `_check_if_database_exists()` überprüft, ob die durch `db_dir` und `db_name` eindeutig bestimmte Datenbank existiert. Falls nicht, wird ein Ausnahmefehler generiert.

Die Methode `_check_database()` testet, ob die notwendigen Tabellen *Job*, *Task*, *TaskSet* und *ExecutionTime* verfügbar sind. Fehlt eine der ersten drei Tabellen, wird ein Ausnahmefehler erstellt. Wenn die Tabelle *ExecutionTime* nicht verfügbar ist, wird diese mit Hilfe des Benchmark-Moduls, das in Kapitel 6.4 beschrieben ist, erstellt.

Zum Testen ob eine Tabelle existiert, ist die Methode `_check_if_table_exists()` definiert. Darin wird eine SQL-Abfrage erstellt, die alle Tabellen mit dem übergebenen Namen sucht. Liefert die Abfrage kein Ergebnis, so existiert die zu testende Tabelle nicht.

### Datenbank öffnen und schließen

Die Methoden `_open_db()` und `_close_db()` sind zum Öffnen und Schließen der Datenbank vorgesehen. Um die Datenbank zu öffnen, wird eine Verbindung und ein Cursor erstellt und gespeichert. Zum Schließen der Datenbank werden eventuell vorgenommene Änderungen an der Datenbank gespeichert. Dann wird die Verbindung beendet und die Attribute `db_connection` und `db_cursor` gelöscht.

### Tabellen lesen und schreiben

Für jede Tabelle in der Datenbank gibt es eine Methode zum Lesen. So liest zum Beispiel die Methode `_read_table_job()` die Tabelle *Job*. Der Aufbau der Funktionen

ist im Großen und Ganzen gleich: die Datenbank wird geöffnet, die gewünschten Daten werden gelesen, die Datenbank wird geschlossen, die gelesenen Daten werden falls erwünscht in ein anderes Format umgewandelt und zurückgegeben. Der Unterschied liegt darin, ob alle oder nur bestimmte Daten gelesen werden und ob bzw. wie die Daten umgewandelt werden.

Die Methode `_read_table_job()` liest mit der SQL-Abfrage

```
SELECT * FROM Job
```

alle Jobs, oder mit

```
SELECT * FROM Job WHERE Set_ID = set_id AND Task_ID = task_id
```

nur die Jobs von einem bestimmten Task mit der ID `task_id` in einem bestimmten Task-Set mit der ID `set_id`. Zurückgegeben wird in jedem Fall eine Liste mit den gelesenen Job-Attributen.

Die Methode `_read_table_task()` liest ebenfalls alle Tasks oder nur den Task mit der ID `task_id`. Zusätzlich wird mit dem Parameter `convert_to_task_dict`, der standardmäßig auf `True` gesetzt ist, bestimmt, ob die Liste in ein Dictionary und die Tasks in Objekte der Klasse `Task` umgewandelt werden. Diese Umwandlung übernimmt die Methode `_convert_to_task_dict()`. Werden alle Tasks aus der Tabelle *Task* gelesen, wird die resultierende Liste nach aufsteigenden Task-IDs sortiert. So entspricht die Position eines Tasks innerhalb der Liste der Task-ID, sodass auch ohne Umwandlung in ein Dictionary auf die richtigen Tasks zugegriffen wird. Als Rückgabe hat die Methode entweder eine Liste mit den Task-Attributen als Tupeln oder ein Dictionary mit den Task-IDs als Schlüssel und den entsprechenden Task-Objekten als Werte.

Die Methode `read_table_taskset()` liest alle Task-Sets aus der Tabelle *TaskSet*, oder nur ein bestimmtes Task-Set mit der ID `taskset_id`, oder alle Task-Sets in denen der Task mit der ID `task_id` der einzige Task ist. Über den Eingabeparameter `convert`, dessen Standardwert `True` ist, wird festgelegt, ob die gelesenen Daten als Liste mit den Task-Set-Attributen als Tupeln oder als Liste bestehend aus Objekten der Klasse `Taskset` zurückgegeben werden. Die Umwandlung der Rohdaten in Objekte der Klasse `Taskset`, wobei die Task-IDs durch Objekte der Klasse `Task` ersetzt werden, übernimmt die Methode `_convert_to_taskset()`.

Die Methode `read_table_executiontime()` liest alle Laufzeiten aus der Tabelle *ExecutionTime*. Durch den Parameter `convert_to_dict`, der standardmäßig auf `True` gesetzt ist, wird bestimmt, ob die gelesenen Daten in ein Dictionary umgewandelt werden. Ein solches Dictionary wird durch die Methode `_convert_to_executiontime_dict()` erstellt, wobei die ID des Task der Schlüssel ist und der Wert die entsprechende durchschnittliche Ausführungszeit des Tasks.

Die Schreibmethoden haben den gleichen Aufbau wie die Lesemethoden, nur wird statt dem Lesen von bestimmten Daten und der Konvertierung in ein anderes Format eine neue Tabelle erstellt, falls diese noch nicht vorhanden ist, und die übergebenen Daten werden in diese Tabelle geschrieben.

Um Werte in die Tabelle *ExecutionTime* zu schreiben, ist die Methode `write_execution_time()` implementiert. Der Eingabeparameter ist ein Dictionary, das als Schlüssel eine Task-ID enthält und als Werte die entsprechenden durchschnittlichen Ausführungszeiten. Nach dem Öffnen der Datenbank wird zunächst eine Tabelle *ExecutionTime* erstellt, falls diese noch nicht existiert. Dann wird der Inhalt des Dictionary sequentiell in die Datenbank geschrieben. Sind dabei Einträge bereits vorhanden, werden diese überschrieben. Am Ende wird die Datenbank geschlossen, wobei die getätigten Änderungen gespeichert werden.

## 6.4 Benchmark

Eine wichtige Task-Eigenschaft ist die Ausführungszeit, die für viele der Schedulability-Analyseverfahren benötigt wird. Dieser Wert ist aber weder in der Tabelle *Task*, noch an anderer Stelle in der Datenbank verfügbar. Um realistische Laufzeiten zu bestimmen, wird die Datenbank selbst verwendet. In der Tabelle *TaskSet* existiert für jeden Task ein Task-Set, das nur diesen einen Task enthält. Außerdem speichert die Tabelle *Job* die ausgeführten Jobs aller Tasks und Task-Sets mit ihren Start- und Endzeitpunkten, das heißt indirekt die Laufzeit der Jobs. Die Idee hinter dem Benchmark-Modul ist deshalb, für jeden Task diejenigen Task-Sets zu finden, in denen der Task der einzige Task ist, und für diese Task-Sets die Attribute aller Jobs des Tasks aus der Tabelle *Job* zu lesen. Aus den Laufzeiten dieser Jobs wird die durchschnittliche Ausführungszeit des Tasks berechnet.

Die Benchmark-Komponente ist in der Datei `benchmark.py` implementiert. Die Methode, die den Benchmark durchführt, ist `benchmark_execution_times()` und in Listing 6.5 abgebildet. Als Eingabe erhält diese Funktion ein Objekt der Klasse *Database*, um auf die Datenbank zugreifen zu können.

Als erstes wird die komplette Tabelle *Task* gelesen und ein leeres Dictionary für die Laufzeiten in Abhängigkeit der Task-ID erstellt. In einer Schleife über die gelesenen Tasks werden alle Task-Sets aus der Datenbank gelesen, die den aktuellen Task als einzigen Task enthalten. Dann werden für diese Task-Sets alle ausgeführten Jobs des Tasks aus der Tabelle *Job* gelesen. Mit Hilfe der Methode `_calculate_executiontimes()` wird für jeden gelesenen Job dessen Laufzeit berechnet. Die durchschnittliche Ausführungszeit des Tasks wird als Mittelwert der Job-Ausführungszeiten bestimmt und gerundet im Dictionary unter der Task-ID gespeichert. Ist für jeden Task ein Durchschnittswert bestimmt und gespeichert, wird das erstellte Dictionary in die Tabelle *ExecutionTime* geschrieben.

Ein Ausschnitt der durch das Benchmark-Modul erstellten Tabelle *ExecutionTime* ist in Tabelle 6.2 zu sehen. Die Tabelle hat zwei Spalten: die Task-ID ist in der Spalte *TASK\_ID* enthalten und die durchschnittliche Ausführungszeit in Millisekunden in der Spalte *Average\_C*.

Listing 6.5: Bestimmung der durchschnittlichen Task-Ausführungszeiten

```

def benchmark_execution_times(database):
    task_list = database.read_table_task(dict=False) # Tabelle Task lesen
    c_dict = dict() # leeres Dictionary für Ausführungszeiten erstellen

    for task in task_list: # Iteration über alle Tasks
        # Task-Sets mit dem aktuellen Task als einzigen Task lesen
        taskset_list = database.read_table_taskset(task_id=task[0], convert=False)

        job_attributes = [] # leere Liste für die Jobs erstellen

        for taskset in taskset_list: # Iteration über alle Task-Sets
            # alle Jobs des aktuellen Tasks und Task-Sets hinzufügen
            job_attributes.extend(database.read_table_job(set_id=taskset[0],
                                                         task_id=task[0]))

        # Ausführungszeit jedes Jobs berechnen
        job_execution_times = _calculate_executiontimes(job_attributes)

        # durchschnittliche Ausführungszeit des aktuellen Tasks berechnen
        average_c = sum(job_execution_times) / len(job_execution_times)

        # Ausführungszeit runden und dem Dictionary hinzufügen
        c_dict[task[0]] = round(average_c)

    # Ausführungszeiten in Tabelle ExecutionTime schreiben
    database.write_execution_time(c_dict)

```

Tabelle 6.2: Beispiele aus der Tabelle *ExecutionTime* (Eigene Darstellung).

TASK_ID	Average_C
49	178
104	4878
123	1680
203	97
227	105
278	330

## 6.5 Logging

Das Logging-Modul ist für alle Programmmeldungen und -ausgaben verantwortlich. Hierfür wird das Logging-Modul von Python verwendet, mit dem Informationen mit unterschiedlichen Logging-Leveln ausgegeben werden, wie beispielsweise DEBUG, INFO und ERROR. Die Logging-Funktionalität ist in der Datei `logging_config.py` implementiert und enthält eine Methode zum Initialisieren des Logging (`init_logging()`) sowie eine Methode zum Ausgeben der Ergebnisse (`log_results()`).

Um das Logging zu konfigurieren, muss zunächst ein eigener Logger erstellt werden:

```
logger = logging.getLogger('traditional-SA').
```

Durch die objektorientierte Schnittstelle mit Logger-Objekten wird für jedes Programmmodul ein eigener Logger erstellt, sodass sich eine Baumstruktur ergibt. An der Wurzel des Baums befindet sich der Root-Logger, der mit oben gezeigter Code-Zeile in der Methode `init_logging()` erstellt wird. Andere Logger werden vom Root-Logger abgeleitet, indem der oben übergebene String erweitert wird. Zum Beispiel wird der Logger für eine Funktion des Moduls `Simulation` durch folgenden Code erstellt:

```
logger = logging.getLogger('traditional-SA.simulation.Funktions-Name').
```

Die folgenden Einstellungsmöglichkeiten werden alle am Root-Logger vorgenommen und die abgeleiteten Logger übernehmen diese automatisch. Wie bereits erwähnt, gibt es verschiedene Log-Level, die bestimmen, welche Informationen ausgegeben werden. Dieses Level wird durch die Methode `setLevel()` des Logging-Modul von Python eingestellt. So stellt

```
logger.setLevel(logging.INFO)
```

das Log-Level INFO ein.

Des Weiteren ist es möglich die Ausgaben in bestimmte Kanäle zu lenken. Diese Kanäle werden durch sogenannte Handler definiert und konfiguriert. Der `FileHandler` schreibt die Logging-Daten in eine Datei, der `StreamHandler` gibt die Logging-Daten über die Standardfehleraussage (Konsole) aus. Im Logging-Modul des *traditional-SA*-Projektes werden, wie Listing 6.6 zeigt, zwei Handler definiert. Fehler werden in die Datei `error.log` im Verzeichnis der Datenbank geschrieben. Alle informativen Meldungen, das bedeutet alle bis auf Debug-Meldungen, werden auf der Konsole ausgegeben.

---

### Listing 6.6: Definition der Logging-Handler

```
log_file_handler = logging.FileHandler(os.path.join(db_dir, 'error.log'), mode='w+')
log_file_handler.setLevel(logging.ERROR)
log_console_handler = logging.StreamHandler()
log_console_handler.setLevel(logging.INFO)
```

---

Das Format der Log-Meldungen wird durch einen Formatter eingestellt:

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s').
```

Die Meldungen (message) werden mit einem Zeitstempel (asctime), dem Namen des Loggers (name) und dem Log-Level (levelname) ausgegeben. Das Format wird mit der Methode `setFormatter()` an die Handler übergeben. Die Handler werden dem Logger schließlich durch die Methode `addHandler()` zugewiesen.

Um die Ergebnisse der Schedulability-Analyseverfahren zu speichern, wird nicht das Logging-Modul von Python verwendet, sondern eine Log-Datei im Verzeichnis der Datenbank erstellt, in die die Ergebnisse durch die Methode `log_results()` geschrieben werden. Die Ergebnisse werden dabei sowohl in die Datei geschrieben, als auch auf der Konsole ausgegeben.

## 6.6 Schedulability-Analyse Frameworks

Damit die Schedulability-Analyseverfahren nicht selbst implementiert werden müssen und um etwaige Implementierungsfehler zu vermeiden, wird nach Frameworks für die Schedulability-Analyse gesucht. Bei der Recherche werden die Frameworks auf ihre integrierten Scheduler, die Möglichkeiten zur Anpassung an die eigenen Daten und die Einbindung in ein Python-Projekt hin untersucht. Eine Übersicht über die gefundenen Frameworks ist in Tabelle 6.3 gegeben. Dabei ist die Anzahl der am Ende verwendeten Frameworks möglichst gering zu halten, sodass Frameworks mit mehreren Methoden bevorzugt werden. Das optimale Ergebnis wäre ein Framework, das alle Schedulability-Analysemethoden implementiert.

Viele Frameworks haben den Nachteil, dass der notwendige FP-Scheduler nicht verfügbar ist, sondern häufig nur die Variationen RM und DM, wobei die Prioritäten nach den Perioden oder den Deadlines der Tasks vergeben werden. Auch das Task-Modell stimmt bei einigen Frameworks nicht mit den gegebenen Daten überein. Viele der Frameworks sind nur theoretisch in Papern veröffentlicht, aber nicht implementiert, oder die verfügbaren Downloads sind veraltet und werden nicht mehr unterstützt.

Letztlich fällt die Entscheidung auf eine eigene Implementierung der Schedulability-Analysefunktionen, da die entsprechenden Algorithmen als nicht sehr anspruchsvoll angesehen werden. Die Qualität der Implementierung wird für jede Methode überprüft und anhand zweier Beispiel-Task-Sets verifiziert. Lediglich für die Simulation wird das einfach zu integrierende Framework SimSo verwendet.

### SimSo

SimSo ist ein Simulator für den Vergleich von Echtzeit Scheduling Algorithmen [Che18]. Mit SimSo ist das Erstellen von Task-Sets, das Durchführen von Simulationen und das

Tabelle 6.3: Übersicht über die recherchierten Schedulability-Analyse Frameworks (Eigene Darstellung).

Framework	Simulation	Tests	Scheduler	Sprache	GUI	Probleme
Cheddar [Sin18]	✓	U, RTA	RM, DM, EDF	AAADL	✓	kein FP-Scheduler
MAST [Har18]	✓	RTA	EDF, FP	UML	✓	letzte Version vom 20.07.2008
RealTSS [DBC07]	✓	U, W	FP, RM, EDF, DM	TCL	✓	letzte Änderung <sup>d</sup> vom 11.01.2012
RTSSim [Kra09]	✓	×	FP	C	×	kein Download verfügbar
SimSo [Che18]	✓	×	EDF, RM, FP	Python	✓	
tms-sim [Klu16]	✓	×	FP, EDF	C++	×	letzte Änderung <sup>b</sup> am 23.08.2016
YARTISS [Cha+12]	✓	×	RM, DM, EDF	Java	✓	kein FP-Scheduler
TORSCHHE [Dvo18]	✓	RTA	FP	Matlab	✓	letzte Änderung <sup>c</sup> am 24.03.2017
YASA [Gol+02]	✓	×	DM, EDF, FP, RM, RR		✓	letzte Version <sup>d</sup> vom 26.01.2003
TIMES [Amn+04]	✓	RTA	RM, DM, FP, EDF	C	✓	letztes Update am 06.11.2008
RTSIM [BL11]	✓	×		C++	✓	letzte Version <sup>e</sup> vom 11.05.2017
pyCPA [Die+17]	✓	×	FP, RR	Python 2	×	unterstützt nur Python 2

<sup>a</sup><https://github.com/acassis/rtsim/tree/master/RealTSS><sup>b</sup><https://github.com/uni-a-sik/tms-sim><sup>c</sup><https://github.com/CTU-TIG/TORSCHHE><sup>d</sup><http://bar.comlab.uni-rostock.de/~bj/software/yasa/download.php><sup>e</sup><https://sourceforge.net/projects/rtsim/files/>



Sammeln von Daten aus Experimenten ohne weitere Kenntnis über das Framework möglich. Das Open-Source-Framework ist implementiert in Python und frei verfügbar unter der CeCILL-Lizenz. SimSo ist sowohl als Bibliothek, das heißt als Python-Modul, als auch als eigenständige Applikation verwendbar. SimSo stellt eine einfach zu bedienende graphische Oberfläche und eine Browser-Anwendung (SimSoWeb<sup>2</sup>) zur Verfügung. Aktuell werden unter anderem der EDF-, RM- und FP-Scheduler für Einprozessorsysteme unterstützt. Benutzerspezifische Scheduler werden als Python-Klasse implementiert und integriert. Die Inspektion der Dokumentation [Che18] zeigt, dass die Implementierung eines neuen Schedulers einfach zu realisieren ist. Auch das Übergeben des Task-Sets an das Framework ist unkompliziert, sodass für die Simulation das Framework SimSo ausgewählt wird.

## 6.7 Simulation

Die Simulation bis zur Hyperperiode eines Task-Sets ist eine exakte Methode zur Bestimmung der Lauffähigkeit, wenn alle Tasks zu einem gemeinsamen Zeitpunkt aktiviert werden [BW96]. Da das Attribut *OFFSET* bei allen Tasks den Wert 0 hat, wie in Kapitel 5 beschrieben, ist diese Bedingung erfüllt. Zur Klassifizierung eines Task-Sets wird das Task-Set zunächst mit Hilfe des Frameworks SimSo simuliert, und anschließend wird das Ergebnis auf Deadline-Verfehlungen hin untersucht. Halten alle Tasks ihre Deadlines ein, ist das Task-Set lauffähig.

### 6.7.1 Scheduler

Der Scheduler des vom Lehrstuhl modifizierten Fiasco.OC Mikrokern ist eine Kombination aus FP und EDF. Wie bereits in Kapitel 5 gezeigt, hat jeder Task ein Attribut *Priority*. Dieses Attribut enthält eine Priorität aus dem Wertebereich  $[1, \dots, 127]$ , wobei der Wert 1 für die höchste Priorität und der Wert 127 für die niedrigste Priorität steht. Für die Werte  $[1, \dots, 126]$  werden die Tasks nach dem FP-Algorithmus ausgeführt. Alle übrigen Tasks mit einer Priorität von 127 werden nach dem EDF-Algorithmus ausgeführt. Das bedeutet, dass immer derjenige Task mit der höchsten Priorität, dementsprechend dem niedrigsten Attribut-Wert, ausgeführt wird.

Da der Scheduler nicht standardmäßig in SimSo enthalten ist, muss eine eigene Scheduler-Klasse von der `simso.core.Scheduler`-Klasse abgeleitet werden [Che18]. Der kombinierte FP-/EDF-Scheduler ist in der Datei `fp_edf_scheduler.py` definiert und in Listing 6.7 zu sehen. Zunächst wird die Scheduler-Klasse importiert. Da die Klasse Scheduler kein Attribut für die Priorität eines Tasks bereitstellt, muss ein neues Task-Feld definiert werden. Die Klasse `fp_edf_scheduler` wird als Unterklasse der Scheduler-Klasse definiert. Jeder Scheduler muss die folgenden vier Funktionen implementieren:

---

<sup>2</sup><http://projects.laas.fr/simso/simso-web>

- Die `init()`-Methode wird aufgerufen wenn die Simulation gestartet wird.
- Die `on_activate()`-Methode wird bei der Aktivierung von Tasks aufgerufen.
- Die `on_terminated()`-Methode wird aufgerufen wenn ein Job seine Ausführung beendet.
- Die `schedule()`-Methode wird aufgerufen wenn der Scheduler ausgeführt werden muss.

Listing 6.7: Implementierung des FP-/EDF-Schedulers

```
from simso.core import Scheduler
@scheduler("fp_edf_scheduler.py", required_task_fields=[{'name': '
    priority', 'type': 'int', 'default': '0'}])

class fp_edf_scheduler(Scheduler):
    def init(self):
        self.ready_list = [ ] # eine Ready-Liste definieren

    def on_activate(self, job):
        self.ready_list.append(job) # Job an Ready-Liste anhängen
        job.cpu.resched()

    def on_terminated(self, job):
        self.ready_list.remove(job) # Job von Ready-Liste entfernen
        job.cpu.resched()

    def schedule(self, cpu):
        if self.ready_list: # wenn min. ein Job bereit ist
            prio_low = min(self.ready_list, key=lambda x: x.data['
                priority']).data['priority']
            if 0 <= prio_low < 127: # FP: Job mit der höchsten Priorität
                job = min(self.ready_list, key=lambda x: x.data['priority
                    '])
            elif prio_low == 127: # EDF: Job mit der nächsten Deadline
                job = min(self.ready_list, key=lambda x: x.
                    absolute_deadline)
            else:
                job = None
        return (job, cpu)
```

---

Grundsätzlich wird eine Liste mit zur Ausführung bereiten Jobs erstellt und diese mit den Methoden `on_activate()` und `on_terminated()` laufend aktualisiert. Aufgrund von SimSo-internen Strukturen wird dabei der Scheduler nicht direkt über die `schedule()`-Methode aufgerufen, sondern durch die `resched()`-Methode. Daraufhin wird derjenige Job aus der Liste ausgewählt, der als nächstes ausgeführt wird. Dafür wird zunächst der kleinste *Priority*-Wert bestimmt, folglich die höchste Priorität, um die Scheduling-Strategie zu bestimmen. Liegt der kleinste Wert unter 127, wird der Job mit der höchsten Priorität ausgewählt (Fixed Priority). Ist der kleinste Wert gleich 127, wird der Job mit der nächsten absoluten Deadline ausgewählt (Earliest Deadline First).

### 6.7.2 Simulation eines Task-Sets

Die Simulation eines Task-Sets ist in der Methode `simulate()` implementiert. Als Eingabe erhält die Methode ein Task-Set wie in 6.3.2 beschrieben. Damit SimSo ein Task-Set simuliert, muss ein Modell des Systems gestaltet werden. Hierfür wird ein Objekt der Klasse `simso.configuration.Configuration` erstellt, das alle Details über das System speichert. Da die Simulation über die Hyperperiode der Tasks ausgeführt wird, wird diese als kleinstes gemeinsames Vielfaches bestimmt. Mit

```
configuration.duration = hyper_period * configuration.cycles_per_ms
```

wird die Länge der Simulation in Takten eingestellt. Bevor die Tasks hinzugefügt werden, muss die Liste der Task-Felder um das Attribut `priority`, die Priorität, erweitert werden. Ein Task wird durch den Aufruf der Methode `add_task()` hinzugefügt, wie Listing 6.8 zeigt. Als worst-case Ausführungszeit (engl. *worst-case execution time*, WCET) wird die durchschnittliche Laufzeit, wie sie durch das Benchmark-Modul bestimmt wird, verwendet. Obwohl Tasks in Kapitel 3.1.2 als periodisch festgelegt werden, wird der Task-Typ als sporadisch definiert. Der Grund hierfür ist, dass die Tasks zwar periodisch ausgeführt werden, allerdings nur bis zu einer bestimmten Anzahl an Wiederholungen. Dieses Verhalten wird mit SimSo nur durch sporadische Tasks erreicht. Der Parameter `list_activation_dates` enthält eine Liste mit allen Zeitpunkten, an denen Jobs des Tasks aktiv werden. Die Aktivierungszeitpunkte eines Tasks sind der Zeitpunkt  $t = 0$  und alle Vielfachen der Perioden bis zur Anzahl der Jobs. Da SimSo-Tasks standardmäßig kein Feld für die Priorität enthalten, wird dieser Wert über den `data`-Parameter übergeben. Auf diese Weise werden alle Tasks des Task-Sets hinzugefügt, bevor schließlich eine Liste von Prozessoren erstellt wird. Da es sich um ein Einprozessor-System handelt, wird nur ein Prozessor generiert. Der selbst implementierte Scheduler wird durch

```
configuration.scheduler_info.filename='fp_edf_scheduler.py'
```

festgelegt. Vor der Initialisierung des SimSo-Modells wird die Konfiguration durch die Methode `configuration.check_all()` auf Fehler überprüft. Das Modell wird mit

```
model = Model(configuration)
model.run_model()
```

erstellt und ausgeführt. Die Ergebnisse der Simulation sind über `model.results` verfügbar. Damit besteht die Analyse der Lauffähigkeit darin, die Jobs aller Tasks auf eine Deadline-Verfehlung zu überprüfen. Speichert der Parameter `model.results.task[i].job[j].aborted` auch nur für einen Job den Wert `True`, so ist das Task-Set nicht lauffähig. Können die Jobs aller Tasks ohne Deadline-Verfehlung simuliert werden, so ist das Task-Set lauffähig.

---

**Listing 6.8: Hinzufügen eines Tasks zu einem SimSo-System**

---

```
configuration.add_task(  
    name=task_name, # Name des Tasks: T + task.id  
    identifier=i, # ID: Rang des Tasks im Task-Set  
    task_type="Sporadic", # Task-Typ: sporadisch  
    period=task.period, # Periode des Tasks  
    activation_date=0, # Zeitpunkt der ersten Aktivierung: t = 0  
    wcet=task.execution_time, # Ausführungszeit des Tasks  
    deadline=task.deadline, # Deadline des Tasks  
    list_activation_dates=activation_dates, # Liste der Aktivierungen  
    data={'priority': task.priority}) # Priorität des Tasks
```

---

### 6.7.3 Beispiel

Das Framework SimSo enthält eine graphische Oberfläche, mit der zum einen Systeme erstellt werden, und zum anderen das Ergebnis einer Simulation visualisiert wird. Am Beispiel des Task-Sets mit der ID 46 429 aus Tabelle 5.1 ist das Ergebnis der Simulation in Abbildung 6.3 zu sehen. Die Hyperperiode berechnet sich zu

$$H = \text{lcm}(T_{104}, T_{49}, T_{227}) = \text{lcm}(7000 \text{ ms}, 6000 \text{ ms}, 2000 \text{ ms}) = 42\,000 \text{ ms},$$

weshalb die Simulation bis zu diesem Zeitpunkt durchgeführt wird.

Die Sortierung des Task-Sets 46 429 nach Prioritäten ergibt die Task-ID-Reihenfolge 49 (Priorität 1), 227 (Priorität 2), 104 (Priorität 4). Deshalb wird Task 49 als erstes ausgeführt, gefolgt von Task 227. Die Ausführung von Task 104 wird zweimal von Wiederholungen des Tasks 227 unterbrochen. Task 104 hält dennoch seine Deadline zum Zeitpunkt  $t = 6500 \text{ ms}$  ein. Auch alle anderen Jobs der Tasks halten ihre Deadlines ein, sodass das Task-Set demzufolge lauffähig ist.

## 6.8 Prozessorauslastung

Die Prozessorauslastung ist der Anteil der Prozessorzeit, der für die Ausführung eines Task-Sets verwendet wird. Ist der Prozessor vollständig ausgelastet, das heißt es existieren keine arbeitsfreien Zeiten, so wird die Auslastung als 1 bzw. 100 % bezeichnet.

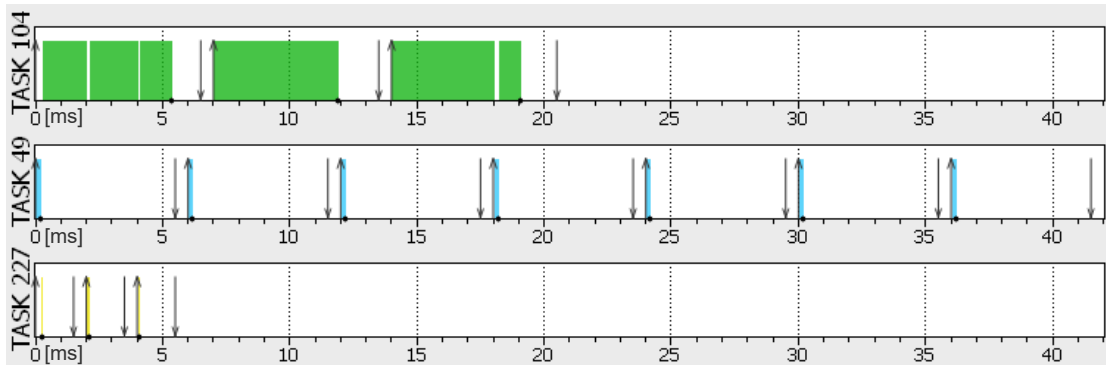


Abbildung 6.3: Beispiel für die Simulation: Ergebnis der Simulation des Task-Sets 46 429 (Eigene Darstellung).

Darauf basierend läuft ein Task-Set nur korrekt ab, wenn dieser Maximalwert nicht überschritten wird. Abhängig von den Eigenschaften des Task-Sets und der Tasks wird der Maximalwert eingeschränkt. Zum Beispiel ist es sinnvoll, den Maximalwert in Abhängigkeit von der Anzahl der Tasks zu bestimmen. Je mehr Tasks in einem Task-Set vorhanden sind, desto geringer sollte die maximale Prozessorauslastung sein, da durch Unterbrechungen und Task-Wechsel zusätzliche Arbeitsbelastungen dazukommen.

In Kapitel 4 werden bereits verschiedene Schedulability-Tests basierend auf der Prozessorauslastung für den FP- und EDF-Algorithmus gezeigt. Obwohl nicht alle genannten Bedingungen für diese Tests, wie zum Beispiel dass die Prioritäten den Perioden oder die Deadline der Periode entspricht, erfüllt sind, werden für experimentelle Zwecke trotzdem alle beschriebenen Tests implementiert und mit den vorhandenen Daten durchgeführt. Die Ergebnisse dieser Test sind in der Hinsicht relevant, dass ein Task-Set, das durch einen für den RM- oder DM-Scheduler konzipierten Test nicht als lauffähig eingestuft wird, auch mit dem FP-Scheduler definitiv nicht lauffähig ist. Zudem ist noch nicht endgültig festgelegt, ob die Prioritäten gänzlich unabhängig von den Perioden bzw. Deadlines der Tasks vergeben werden.

Alle Prozessorauslastungs-Tests sind in `utilization.py` implementiert und erhalten als Eingabe ein Task-Set. Der Programmablauf ist bei allen Tests gleich und in Abbildung 6.4 gezeigt.

Da alle Algorithmen selbst implementiert sind, ist es notwendig, die Korrektheit der Implementierung zu überprüfen. Anhand zweier zufällig ausgewählter Task-Sets aus der Tabelle *TaskSet* werden die Ergebnisse überprüft und verifiziert. Jeder Prozessorauslastungs-Test wird nachfolgend für das Task-Sets 46 429 exemplarisch durchgeführt, um den jeweiligen Algorithmus zu verdeutlichen und die Implementierungen besser nachvollziehen zu können. Dafür muss für jeden Task die Ausführungszeit (C), die Deadline (D) und die Periode (T) bekannt sein. Tabelle 6.4 fasst die wichtigsten Attribute der Beispiel-Task-Sets aus den Tabellen 5.1, 5.2 und 6.2 zusammen.

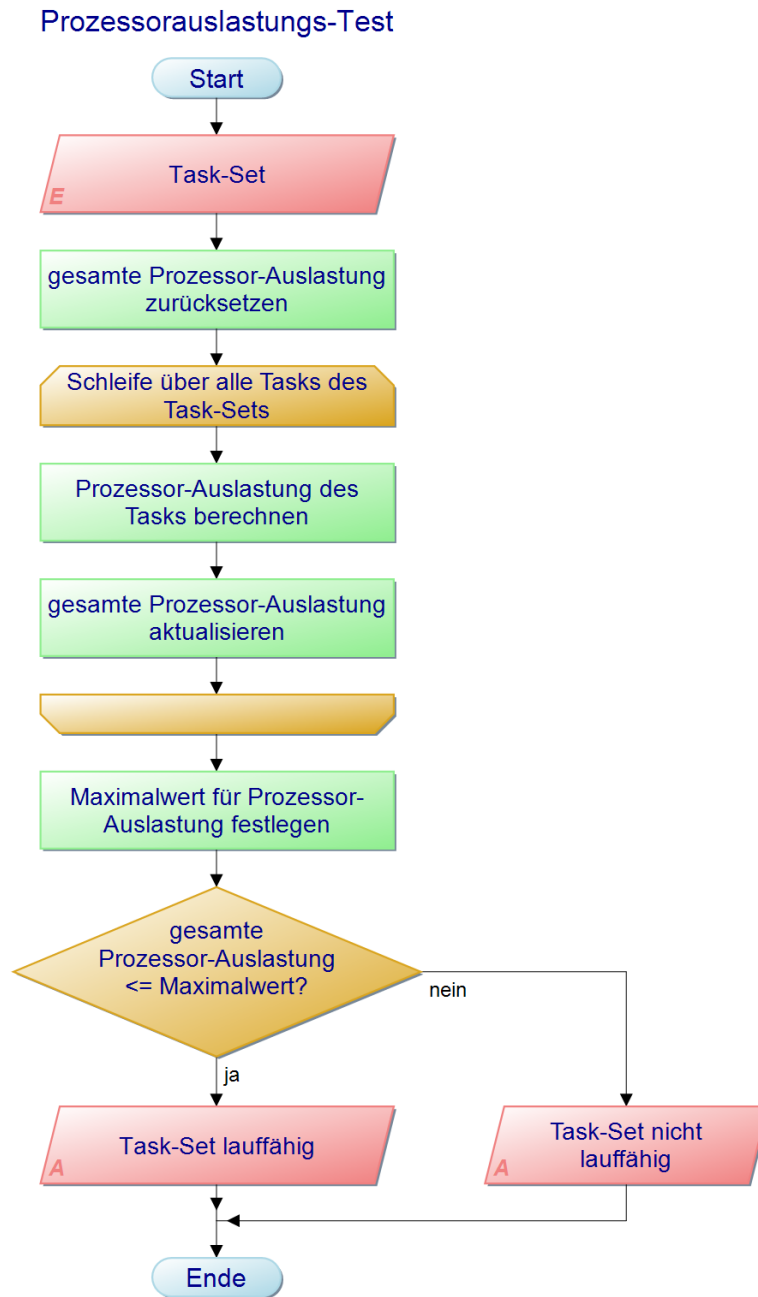


Abbildung 6.4: Allgemeiner Programmablaufplan der Prozessorauslastungs-Tests (Eigene Darstellung).

Tabelle 6.4: Zusammenfassung der wichtigsten Attribute des Beispiel-Task-Sets 46 429 (Eigene Darstellung).

Task-ID	Priorität (P)	Ausführungszeit (C)	Deadline (D)	Periode (T)
104	4	4878 ms	6500 ms	7000 ms
49	1	178 ms	5500 ms	6000 ms
227	2	105 ms	1500 ms	2000 ms

### 6.8.1 Einfacher Prozessorauslastungs-Test

Der einfache Prozessorauslastungs-Test basiert auf dem in [Liu00] vorgeschlagenen Test für den EDF-Algorithmus. Ein Task-Set ist demnach lauffähig, wenn

$$U = \sum_{i=1}^n \frac{C_i}{\min(D_i, T_i)} \leq 1 \quad (6.1)$$

erfüllt ist. Allerdings ist auch für jeden anderen Scheduler das Task-Set nicht lauffähig, wenn die Prozessorauslastung mehr als 100 % beträgt, weshalb der Test auch für den FP-Scheduler anwendbar ist. Implementiert ist der einfache Prozessorauslastungs-Test in der Methode `basic_utilization_test()`. Darin wird zunächst die gesamte Prozessorauslastung auf den Wert 0 gesetzt. Anschließend wird die Prozessorauslastung durch jeden Task nach Gleichung 6.1 berechnet und zur gesamten Prozessorauslastung addiert. Da der Maximalwert 1 beträgt, wird die berechnete gesamte Prozessorauslastung damit verglichen. Ist  $U \leq 1$ , so ist das Task-Set lauffähig und es wird der Wert `True` zurückgegeben. Andernfalls ist das Task-Set nicht lauffähig und es wird `False` zurückgegeben.

Als Beispiel wird das Task-Set 46 429 betrachtet. Die gesamte Prozessorauslastung ergibt sich zu:

$$\begin{aligned}
 U &= \sum_{i=1}^3 \frac{C_i}{\min(D_i, T_i)} = \\
 &= \frac{4878 \text{ ms}}{\min(6500 \text{ ms}, 7000 \text{ ms})} + \frac{178 \text{ ms}}{\min(5500 \text{ ms}, 6000 \text{ ms})} + \frac{105 \text{ ms}}{\min(1500 \text{ ms}, 2000 \text{ ms})} \approx \\
 &\approx 0,852825.
 \end{aligned}$$

Demnach ist das Task-Set lauffähig, da die gesamte Prozessorauslastung  $U$  kleiner als der Maximalwert 1 ist.

### 6.8.2 RM-Prozessorauslastungs-Test

Die Vorschrift des RM-Prozessorauslastungs-Tests für ein lauffähiges Task-Set lautet

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \quad (6.2)$$

und entspricht damit dem Schedulability-Test für den RM-Algorithmus in [LL73]. Die Implementierung des RM-Prozessorauslastungs-Test befindet sich in der Methode `rm_utilization_test()`. Wie auch beim einfachen Prozessorauslastungs-Test wird zunächst die gesamte Auslastung auf den Wert 0 gesetzt. Zur Berechnung der gesamten Prozessorauslastung wird die Ausführungszeit jedes Tasks durch dessen Periode geteilt und die Ergebnisse aufsummiert. Der Maximalwert der Prozessorauslastung wird nach Gleichung 6.2 berechnet, wobei  $n$  für die Anzahl der Tasks steht. Überschreitet die berechnete gesamte Prozessorauslastung diesen Maximalwert, so ist das Task-Set nicht lauffähig und es wird `False` zurückgegeben. Im anderen Fall ist das Task-Set lauffähig und es wird `True` zurückgegeben.

Für das Task-Set 46 429 ergibt sich mit dem RM-Prozessorauslastungs-Test, dass das Task-Set lauffähig ist, da der Maximalwert für drei Tasks eingehalten wird:

$$U = \frac{4878 \text{ ms}}{7000 \text{ ms}} + \frac{178 \text{ ms}}{6000 \text{ ms}} + \frac{105 \text{ ms}}{2000 \text{ ms}} \approx 0,779\,024 < 3(2^{1/3} - 1) \approx 0,779\,763.$$

### 6.8.3 HB-Prozessorauslastungs-Test

Ein weiterer auf der Prozessorauslastung basierender Test ist der HB-Prozessorauslastungs-Test, der die Hyperbolic Bound (HB) aus [BBB01] und [BBB03] verwendet. Der Test wird darin folgendermaßen definiert:

$$\prod_{i=1}^n (U_i + 1) \leq 2, \quad (6.3)$$

wobei die Prozessorauslastung durch einen Task durch  $U_i = \frac{C_i}{T_i}$  gegeben ist. Der HB-Prozessorauslastungs-Test ist in der Methode `hb_utilization_test()` implementiert. Als erstes wird darin die gesamte Auslastung auf den Wert 1 gesetzt. Dies ist notwendig, da die Task-Auslastungen nicht addiert, sondern multipliziert werden, um die gesamte Prozessorauslastung zu bestimmen. Zur Bestimmung der Lauffähigkeit wird anschließend die gesamte Prozessorauslastung mit dem Maximalwert 2 verglichen und der resultierende Bool'sche Wert `True` für „lauffähig“ bzw. `False` für „nicht lauffähig“ zurückgegeben.

Als Beispiel wird wieder das Task-Set 46 429 betrachtet:

$$\left(\frac{4878 \text{ ms}}{7000 \text{ ms}} + 1\right) \cdot \left(\frac{178 \text{ ms}}{6000 \text{ ms}} + 1\right) \cdot \left(\frac{105 \text{ ms}}{2000 \text{ ms}} + 1\right) \approx 1,838\,925 < 2.$$

Da das Ergebnis den Maximalwert von 2 nicht überschreitet, ist das Task-Set demnach lauffähig.



## 6.9 Antwortzeit-Analyse

Die Analyse der Antwortzeit ist wie die Simulation ein exaktes Mittel um die Lauffähigkeit eines Task-Sets zu bestimmen. Die Antwortzeit wird für den ungünstigsten Fall (WCRT) bestimmt, das heißt wenn alle Tasks zeitgleich zum Zeitpunkt  $t = 0$  gestartet werden. Ist für jeden Task  $\tau_i$  des Task-Sets die Bedingung  $R_i \leq D_i$  erfüllt, so ist das Task-Set lauffähig. Verletzt auch nur ein Task seine Deadline, wird das Task-Set als nicht lauffähig eingestuft. Die Antwortzeit eines Tasks  $\tau_i$  wird nach Gleichung 6.4 berechnet:

$$R_i^{k+1} = C_i + I_i = C_i + \sum_{\forall j \in \text{hp}(i)} \left\lceil \frac{R_j^k}{T_j} \right\rceil C_j. \quad (6.4)$$

Dabei ist  $I_i$  die Störung durch alle Tasks mit höherer oder gleicher Priorität als Task  $\tau_i$ . Diese Menge von Tasks wird im Nachfolgenden als HP-Set (engl. *higher priority set*) von Task  $\tau_i$  bezeichnet. Gleichung 6.4 ist eine rekursive Gleichung. Die Rekursion wird so lange durchgeführt, bis sich der Wert der Antwortzeit nicht mehr ändert, das bedeutet  $R_i^{k+1} = R_i^k$  ist. Da in der Literatur zwei verschiedene Startwerte angegeben werden, sind zwei Methoden für die Antwortzeit-Analyse (engl. *response time analysis*, RTA) implementiert, die sich nur in ihrem Startwert für die Rekursion unterscheiden und in der Datei `rta.py` definiert sind:

- `rta_audsley()`: Antwortzeit-Analyse mit Startwert  $R_i^0 = C_i$  aus [Aud+93],
- `rta_buttazzo()`: Antwortzeit-Analyse mit Startwert  $R_i^0 = \sum C_j$  aus [But11].

### 6.9.1 Antwortzeit-Analyse eines Task-Sets

Der Programmablaufplan der Antwortzeit-Analyse eines Task-Sets ist in Abbildung 6.5 dargestellt. Als Eingabe erhält die Methode ein Task-Set. Mit Hilfe einer Schleife wird über alle Tasks des Task-Sets iteriert und die Antwortzeit für jeden Task berechnet. Der Task, für den die Antwortzeit aktuell berechnet wird, wird im Folgenden als „Check-Task“ bezeichnet. Dafür wird zunächst ein Startwert festgelegt. Der Startwert nach [Aud+93] ist die Ausführungszeit des Check-Tasks. Der Startwert nach Buttazzo wird durch die Methode `_get_start_value_buttazzo()` berechnet und ist die Summe der Ausführungszeiten aller Tasks mit höherer oder gleicher Priorität sowie die Ausführungszeit des Check-Tasks. Die Berechnung der Antwortzeit des Check-Tasks übernimmt die Methode `_calculate_response_time()`, die ebenfalls in `rta.py` definiert ist und im folgenden Kapitel genauer betrachtet wird. Die von dieser Methode zurückgegebene Antwortzeit wird mit der Deadline des Check-Tasks verglichen. Ist die Antwortzeit größer als die Deadline, verfehlt der Task seine Deadline und das gesamte Task-Set wird als nicht lauffähig klassifiziert. Andernfalls wird der nächste Task im Task-Set überprüft und so weiter. Haben am Ende der Schleife alle Tasks ihre Deadlines eingehalten, so ist das Task-Set lauffähig.

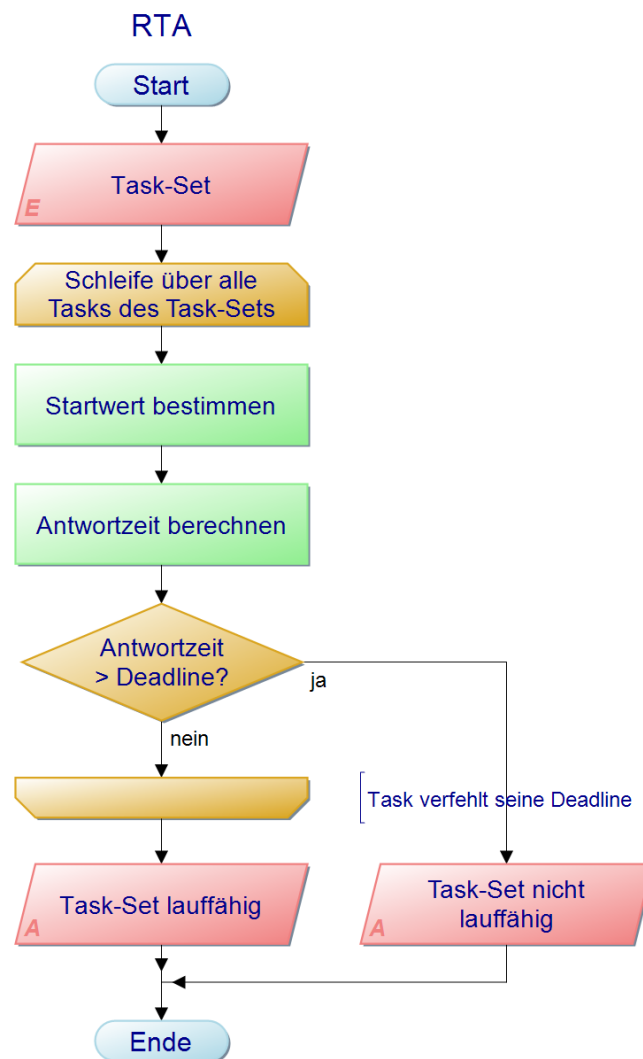


Abbildung 6.5: Programmablaufplan der Antwortzeit-Analyse (Eigene Darstellung).

### 6.9.2 Berechnung der Antwortzeit

Die Berechnung der Antwortzeit ist in der Methode `_calculate_response_time()` implementiert und in Listing 6.9 zu sehen. Zuerst wird das HP-Set für den Check-Task erstellt. Dies geschieht durch die Methode `_create_hp_set()`, indem über alle Tasks des Task-Sets iteriert wird und diejenigen Tasks mit höherer oder gleicher Priorität in das HP-Set kopiert werden. Wenn das HP-Set leer ist, wird keine Berechnung gestartet, da der Check-Task die höchste Priorität hat und deshalb ohne Unterbrechung ausgeführt wird. Die Antwortzeit entspricht in diesem Fall dem gegebenen Startwert. Andernfalls werden zwei Variablen erstellt, die die Antwortzeit der letzten Iteration (`r_old`) und der aktuellen Iteration (`r_new`) speichern. In einer Schleife wird solange eine neue Antwortzeit für den Check-Task berechnet, bis sich der Wert nicht mehr ändert. Innerhalb der Schleife wird zuerst die Antwortzeit der letzten Iteration in die Variable `r_old` verschoben. Dann wird die Störung durch alle Tasks des HP-Sets wie in Gleichung 6.4 gezeigt berechnet. Die neue Antwortzeit ergibt sich als Summe aus der Ausführungszeit des Check-Tasks und der Störung durch das HP-Set. Ist die neu berechnete Antwortzeit größer als die Deadline des Check-Tasks, verfehlt der Task seine Deadline. Aus diesem Grund wird die Berechnung hier abgebrochen und die aktuelle Antwortzeit zurückgegeben. Diese Schritte werden solange wiederholt, bis die Deadline des Check-Tasks überschritten wird, oder sich der Wert der Antwortzeit nicht mehr ändert. Im letzteren Fall wird die gefundene Antwortzeit des Check-Tasks von der Methode zurückgegeben.

### 6.9.3 Beispiel-Rechnung

Zum Überprüfen der Implementierung und um die Funktionsweise der Antwortzeit-Analyse zu veranschaulichen, wird nachfolgend der Rechenweg für das Task-Set 46 429 skizziert. Als Startwert wird die Ausführungszeit des jeweiligen Tasks verwendet, das heißt es wird die Antwortzeit-Analyse nach [Aud+93] durchgeführt. Die Berechnung nach [But11] ist identisch, nur wird ein anderer Ausgangswert für  $R_i^0$  eingesetzt. Die Reihenfolge der Tasks des Task-Sets 46 429 nach Prioritäten lautet: 49 (Priorität 1) - 227 (Priorität 2) - 104 (Priorität 4). Deshalb wird zuerst die Antwortzeit des Tasks 49 wie folgt berechnet:

$$\begin{aligned} hp(49) &= \emptyset \\ R_{49}^0 &= C_{49} = 178 \text{ ms} < D_{49} = 5500 \text{ ms} \end{aligned}$$

Das HP-Set von Task 49 ist leer, da der Task die höchste Priorität besitzt. Der Startwert ist die Ausführungszeit des Tasks. Da jedoch das HP-Set leer ist, wird kein weiterer Wert berechnet und die Antwortzeit des Tasks 49 entspricht der Ausführungszeit von 178 ms. Somit wird die Deadline bei 5500 ms auf jeden Fall eingehalten.

## Listing 6.9: Berechnung der Antwortzeit

```
def _caluclate_response_time(taskset, check_task, start_value):  
    # HP-Set erstellen  
    high_prio_set = _create_hp_set(taskset, check_task)  
  
    if not high_prio_set: # Check-Task hat die höchste Priorität  
        return start_value # Startwert zurückgeben  
  
    r_old = 0 # Antwortzeit der letzten Iteration  
    r_new = start_value # Antwortzeit der aktuellen Iteration  
  
    while r_old != r_new: # Antwortzeit ändert sich  
        r_old = r_new # Antwortzeit der letzten Iteration speichern  
  
        # Störung durch HP-Set berechnen  
        interference = 0  
        for task in high_prio_set: # Iteration über Tasks des HP-Sets  
            interference += math.ceil(r_old / task.period)  
                * task.execution_time  
  
        # Antwortzeit der aktuellen Iteration berechnen  
        r_new = check_task.execution_time + interference  
  
        if r_new > check_task.deadline: # Check-Task verfehlt Deadline  
            return r_new # aktuelle Antwortzeit zurückgeben  
  
    return r_new # berechnete Antwortzeit zurückgeben
```

---

Für den Task 227 ergibt sich:

$$\begin{aligned}
 hp(227) &= \{49\} \\
 R_{227}^0 &= C_{227} = 105 \text{ ms} \\
 R_{227}^1 &= \left\lceil \frac{R_{227}^0}{T_{49}} \right\rceil \cdot C_{49} + C_{227} = \left\lceil \frac{105 \text{ ms}}{6000 \text{ ms}} \right\rceil \cdot 178 \text{ ms} + 105 \text{ ms} = 283 \text{ ms} \\
 R_{227}^2 &= \left\lceil \frac{R_{227}^1}{T_{49}} \right\rceil \cdot C_{49} + C_{227} = \left\lceil \frac{283 \text{ ms}}{6000 \text{ ms}} \right\rceil \cdot 178 \text{ ms} + 105 \text{ ms} = 283 \text{ ms} \\
 R_{227}^2 &= R_{227}^1 = 283 \text{ ms} < D_{227} = 1500 \text{ ms}.
 \end{aligned}$$

Das HP-Set besteht aus dem Task mit der höchsten Priorität (Task 49). Der Startwert entspricht der Ausführungszeit von Task 227. Da das HP-Set nicht leer ist, wird der nächste Wert für die Antwortzeit berechnet, der die Deadline des Task 227 nicht überschreitet. Ein weiterer Wert für die Antwortzeit wird bestimmt, allerdings ändert sich dieser nicht zum vorherigen Wert. Die Antwortzeit von Task 227 ist 283 ms, was unterhalb der Deadline des Tasks bei 1500 ms liegt.

Die Rechnung für den letzten Task 104 lautet:

$$\begin{aligned}
 hp(104) &= \{49, 227\} \\
 R_{104}^0 &= C_{104} = 4878 \text{ ms} \\
 R_{104}^1 &= \left\lceil \frac{R_{104}^0}{T_{49}} \right\rceil \cdot C_{49} + \left\lceil \frac{R_{104}^0}{T_{227}} \right\rceil \cdot C_{227} + C_{104} = \\
 &= \left\lceil \frac{4878 \text{ ms}}{6000 \text{ ms}} \right\rceil \cdot 178 \text{ ms} + \left\lceil \frac{4878 \text{ ms}}{2000 \text{ ms}} \right\rceil \cdot 105 \text{ ms} + 4878 \text{ ms} = 5371 \text{ ms} \\
 R_{104}^2 &= \left\lceil \frac{R_{104}^1}{T_{49}} \right\rceil \cdot C_{49} + \left\lceil \frac{R_{104}^1}{T_{227}} \right\rceil \cdot C_{227} + C_{104} = \\
 &= \left\lceil \frac{5371 \text{ ms}}{6000 \text{ ms}} \right\rceil \cdot 178 \text{ ms} + \left\lceil \frac{5371 \text{ ms}}{2000 \text{ ms}} \right\rceil \cdot 105 \text{ ms} + 4878 \text{ ms} = 5371 \text{ ms} \\
 R_{104}^2 &= R_{104}^1 = 5371 \text{ ms} < D_{104} = 6500 \text{ ms}
 \end{aligned}$$

Das HP-Set besteht aus den Tasks 49 und 227, der Startwert ist die Ausführungszeit von Task 104. Der erste berechnete Wert für die Antwortzeit ergibt sich zu 5371 ms und ist identisch zum zweiten berechneten Wert. Die Antwortzeit ist deshalb 5371 ms, was unter der Deadline von Task 104 bei 6500 ms liegt. Das Task-Set 46 429 wird somit von der Antwortzeit-Analyse als lauffähig eingeordnet.

## 6.10 Arbeitsbelastung

Ein weiterer exakter Schedulability-Test basiert auf der Level-i Arbeitsbelastung, der Rechenzeit die in einem Intervall  $[0, t]$  von einem Task  $\tau_i$  und allen höher- oder gleich-priorisierten Tasks angefordert wird. Eine Schedulability-Analyse basierend auf diesem

Maß wird in [LSD89] für den RM-Scheduler vorgestellt. Ein weiterer Arbeitsbelastungs-Test ist der Hyperplanes  $\delta$ -Exact Test von [BB01; BB04]. Beide Ansätze werden in den nachfolgenden Kapiteln beschrieben.

### 6.10.1 RM-Arbeitsbelastungs-Test

Der RM-Arbeitsbelastungs-Test aus [LSD89] ist ursprünglich für den RM-Scheduler vorgesehen, wird hier aber der Vollständigkeit halber und aus experimentellen Zwecken dennoch auf den FP-Scheduler angewendet. So lassen sich zumindest alle nicht lauffähigen Task-Sets eindeutig bestimmen. Da in [LSD89] die Annahme gilt, dass die Deadlines der Tasks ihren Perioden entsprechen, werden zur Berechnung der Arbeitsbelastung ausschließlich die Task-Perioden verwendet.

Die Formel für die Arbeitsbelastung nach [LSD89] lautet:

$$W_i(t) = \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j. \quad (6.5)$$

Gemäß [LSD89] ist ein Task-Set nur lauffähig, wenn die Bedingung

$$L = \max_{1 \leq i \leq n} L_i \leq 1 \quad (6.6)$$

erfüllt ist, wobei  $n$  für die Anzahl der Tasks steht. Das bedeutet, dass die Arbeitsbelastung durch jeden Task maximal 1 betragen darf. Sobald ein Task diesen Wert überschreitet, ist das Task-Set nicht mehr lauffähig, da die Arbeitsbelastung größer als das verfügbare Zeitintervall ist. Um einen Task auf seine Planbarkeit hin zu untersuchen, wird die Gleichung

$$L_i = \min_{0 < t \leq T_i} L_i(t) \leq 1 \quad (6.7)$$

ausgewertet, wobei die Hilfsfunktion  $L_i(t)$  für die Arbeitsbelastung des Tasks bezogen auf den Zeitpunkt  $t$  steht:

$$L_i(t) = \frac{W_i(t)}{t}. \quad (6.8)$$

Zur Bestimmung der Arbeitsbelastung eines Tasks müssen nur diskrete Werte von  $t$  im Intervall  $[0, T_i]$  betrachtet werden, die sogenannten Planungspunkte (engl. *scheduling points*). Für einen Task  $\tau_i$  lauten die Planungspunkte:

$$S_i = \{k \cdot T_j \mid j = 1, \dots, i; k = 1, \dots, \lfloor T_i/T_j \rfloor\}. \quad (6.9)$$

Anschaulich betrachtet sind dies die Deadline des Tasks  $\tau_i$  und alle Vielfachen von Task-Perioden mit höherer oder gleicher Priorität als  $\tau_i$  vor dessen Deadline.

## RM-Arbeitsbelastungs-Test

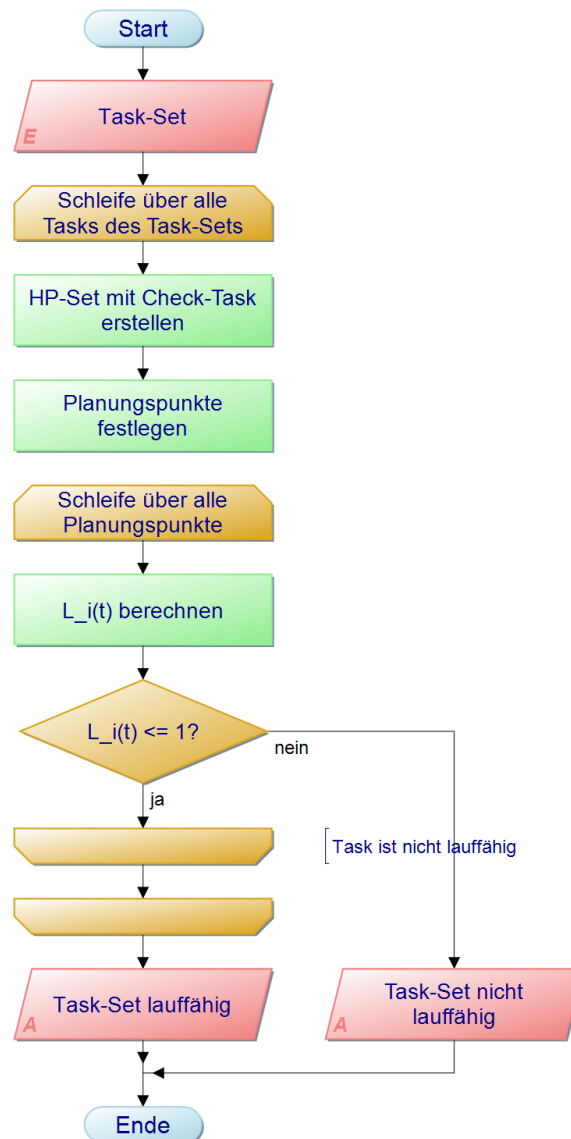


Abbildung 6.6: Programmablaufplan des Arbeitsbelastungs-Tests für den RM-Scheduler (Eigene Darstellung).

### Implementierung

Die Implementierung des RM-Arbeitsbelastungs-Tests befindet sich in der Methode `rm_workload_test()` (`workload.py`) und der Programmablaufplan ist in Abbildung 6.6 zu sehen.

Als Eingabeparameter erhält die Funktion ein Task-Set. In einer Schleife über alle Tasks des Task-Sets wird die Planbarkeit jedes Tasks überprüft. Der Task, der aktuell getestet wird, heißt im Folgenden wieder „Check-Task“. Dafür wird zunächst das HP-Set bestimmt, das bedeutet ein Task-Set mit allen Tasks die eine höhere oder gleiche Priorität haben als der Check-Task. Anders als bei der Antwortzeit-Analyse wird hier der Check-Task ebenfalls dem HP-Set hinzugefügt und fortan nur noch mit diesem Task-Set gearbeitet. Als nächstes werden die Planungspunkte durch die Methode `_get_scheduling_points()` festgelegt. In einer weiteren Schleife wird für jeden Planungspunkt die Arbeitsbelastung  $L_i(t)$  des Check-Tasks berechnet. Ist diese größer als 1, ist der Task nicht lauffähig und das gesamte Task-Set ebenfalls nicht. Andernfalls wird der nächste Task überprüft. Liegt die Arbeitsbelastung von jedem Task und jedem Planungspunkt unter 1, so sind alle Tasks planbar und das Task-Set ist lauffähig.

### Bestimmung der Planungspunkte

Die Planungspunkte für einen Check-Task und ein Task-Set werden durch die Methode `_get_scheduling_points()` festgelegt, wie Listing 6.10 zeigt. Als Eingabe erhält die Funktion das HP-Task-Set und den Check-Task. Als erstes wird eine leere Liste für die Planungspunkte angelegt. In einer Schleife wird über alle Tasks des übergebenen Task-Sets iteriert und der maximale Wert für  $k$  aus Gleichung 6.9 bestimmt, indem die Periode des Check-Task durch die Periode des aktuellen Tasks der Schleife geteilt und aufgerundet wird. In einer `while`-Schleife wird über alle ganzzahligen Werte von  $k$  im Intervall  $[1, \dots, k_{\max}]$  iteriert und die Planungspunkte berechnet. Mit jedem neuen Wert von  $k$  wird der entsprechende neue Planungspunkt durch Multiplikation von  $k$  mit der Periode des Tasks bestimmt. Nur wenn der Planungspunkt noch nicht in der Liste vorhanden ist, wird er hinzugefügt. Dies stellt sicher, dass kein Planungspunkt doppelt gespeichert wird. Am Ende der beiden Schleifen werden die Planungspunkte nach aufsteigender Reihenfolge sortiert und zurückgegeben.

### Berechnung der Arbeitsbelastung eines Tasks

Um die Arbeitsbelastung eines Tasks zu berechnen, sind die Methoden `_L_i()` und `_workload_i()` definiert. Die zweite Methode bestimmt die Arbeitsbelastung eines Tasks nach Gleichung 6.5, die von der ersten Methode in Relation zum Planungspunkt gesetzt wird (siehe Gleichung 6.8). In der Methode `rm_workload_test()` wird `_L_i()` mit dem aktuellen Planungspunkt  $t$  und dem HP-Task-Set aufgerufen und es wird der Code in Listing 6.11 ausgeführt. Darin wird zunächst die Methode `_workload_i()` mit dem übergebenen Planungspunkt  $t$  und dem Task-Set aufgerufen. Die darin berechnete Arbeitsbelastung wird durch den Wert von  $t$  geteilt und das Ergebnis zurückgegeben.



**Listing 6.10: Bestimmung der Planungspunkte**

```
def _get_scheduling_points(hp_taskset, check_task):
    scheduling_points = [ ] # leere Liste für Planungspunkte erstellen
    for task in hp_taskset: # Iteration über HP-Task-Set
        k_max = math.floor(check_task.period / task.period)
        k = 1
        while k <= k_max: # Iteration über alle k = 1, ..., k_max
            # Planungspunkt berechnen
            new_scheduling_point = k * task.period

            # Wenn Planungspunkt unbekannt: der Liste hinzufügen
            if new_scheduling_point not in scheduling_points:
                scheduling_points.append(new_scheduling_point)
            k += 1

        scheduling_points.sort() # Planungspunkte sortieren
    return scheduling_points
```

---

Die Berechnung der Arbeitsbelastung nach Gleichung 6.5 ist in Listing 6.12 zu sehen. Zu Beginn wird die Arbeitsbelastung des Task-Sets auf 0 gesetzt. Die Arbeitsbelastung für jeden Task wird berechnet und addiert. Zuletzt wird die Summe der Arbeitsbelastungen zurückgegeben.

**Listing 6.11: Berechnung der Arbeitsbelastung in Relation zum Planungspunkt**

```
def _L_i(t, taskset):
    l_i = _workload_i(t, taskset) / t
    return l_i
```

---

**Listing 6.12: Berechnung der Arbeitsbelastung**

```
def _workload_i(t, taskset):
    w_i = 0
    for task in taskset:
        w_i += math.ceil(t / task.period) * task.execution_time
    return w_i
```

---

**Beispiel**

Als Beispiel wird wieder das Task-Set 46 429 mit den Tasks 49 (Priorität 1), 227 (Priorität 2) und 104 (Priorität 4) betrachtet. Zur Überprüfung der Lauffähigkeit des Task-Sets werden alle Tasks auf ihre Lauffähigkeit getestet, beginnend mit dem Task höchster Priorität (Task 49):

$$hp(49) = \{49\} \quad \Rightarrow S_{49} = \{T_{49}\} = \{6000 \text{ ms}\}$$

$$\begin{aligned} t = 6000 \text{ ms} : \quad W_{49}(6000 \text{ ms}) &= \left\lceil \frac{6000 \text{ ms}}{T_{49}} \right\rceil \cdot C_{49} = \left\lceil \frac{6000 \text{ ms}}{6000 \text{ ms}} \right\rceil \cdot 178 \text{ ms} = 178 \text{ ms} \\ L_{49}(6000 \text{ ms}) &= \frac{W_{49}(6000 \text{ ms})}{6000 \text{ ms}} \approx 0,029667 \leq 1. \end{aligned}$$

Da Task 49 der Task mit höchster Priorität ist, enthält das HP-Set nur den zu testenden Task selbst. Die Menge der Planungspunkte besteht deshalb auch nur aus der Periode des Task 49. Somit muss nur die Arbeitsbelastung für den Planungspunkt  $t = 6000 \text{ ms}$  bestimmt werden. Da diese bezogen auf den Planungspunkt kleiner als 1 ist, ist der Task lauffähig.

Für Task 227 ergibt sich:

$$hp(227) = \{49, 227\} \quad \Rightarrow S_{227} = \{T_{227}\} = \{2000 \text{ ms}\}$$

$$\begin{aligned} t = 2000 \text{ ms} : \quad W_{227}(2000 \text{ ms}) &= \left\lceil \frac{2000 \text{ ms}}{6000 \text{ ms}} \right\rceil \cdot 178 \text{ ms} + \left\lceil \frac{2000 \text{ ms}}{2000 \text{ ms}} \right\rceil \cdot 105 \text{ ms} = \\ &= 283 \text{ ms} \\ L_{227}(2000 \text{ ms}) &= \frac{W_{227}}{2000 \text{ ms}} = 0,1415 \leq 1. \end{aligned}$$

Das HP-Set besteht aus den Tasks 49 und 227. Die Planungspunkte sind aber lediglich die Periode von Task 227, da die Periode von Task 49 mit 6000 ms deutlich größer ist als die Periode von Task 227 und deshalb für die Planbarkeit von Task 227 nicht berücksichtigt werden muss. Für den Zeitpunkt  $t = 2000 \text{ ms}$  ergibt sich für Task 227 eine Arbeitsbelastung von 283 ms bzw. 0,1415, was weniger als 1 ist. Der Task ist deshalb lauffähig.

Als letztes wird Task 104 überprüft:

$$\begin{aligned} hp(104) &= \{49, 227, 104\} \\ \Rightarrow S_{104} &= \{T_{49}, 2 \cdot T_{49}, T_{227}, T_{104}\} = \{2000 \text{ ms}, 4000 \text{ ms}, 6000 \text{ ms}, 7000 \text{ ms}\} \end{aligned}$$

$$\begin{array}{lll} t = 2000 \text{ ms} : & W_{104}(2000 \text{ ms}) = 5156 \text{ ms} & L_{104}(2000 \text{ ms}) = 2,5805 > 1 \\ t = 4000 \text{ ms} : & W_{104}(4000 \text{ ms}) = 5266 \text{ ms} & L_{104}(4000 \text{ ms}) = 1,3165 > 1 \\ t = 6000 \text{ ms} : & W_{104}(6000 \text{ ms}) = 5371 \text{ ms} & L_{104}(6000 \text{ ms}) = 0,895167 \leq 1. \end{array}$$

Das HP-Set besteht aus allen drei Tasks. Als Planungspunkte ergeben sich alle Vielfachen der Task-Perioden bis zur Deadline, entspricht der Periode, von Task 104 bei 7000 ms. Nacheinander werden die Arbeitsbelastungen für jeden Planungspunkt bestimmt und mit dem Wert 1 verglichen. Da zum Zeitpunkt  $t = 6000$  ms das Verhältnis aus Arbeitsbelastung zu Planungspunkt unter 1 liegt, ist Task 104 planbar und das gesamte Task-Set damit lauffähig.

### 6.10.2 Hyperplanes $\delta$ -Exact Test

Der Hyperplanes  $\delta$ -Exact Test ( $\delta$ -HET) aus [BB01; BB04] ist ein exaktes Schedulability-Analyseverfahren und überprüft die Lauffähigkeit eines Task-Sets, indem die folgende Gleichung für jeden Task ausgewertet wird:

$$C_i + W_{i-1}(D_i) \leq D_i. \quad (6.10)$$

Der Index  $i$  steht dabei für die Position des Check-Tasks innerhalb des nach aufsteigenden Prioritäten sortierten Task-Sets. Nur wenn die Bedingung für jeden Task erfüllt ist, wird das Task-Set als lauffähig eingeordnet. Die Arbeitsbelastung wird dabei folgendermaßen berechnet:

$$W_{i-1}(D_i) = \min_{t \in P_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j + (D_i - t), \quad (6.11)$$

wobei  $W_0(D_1) = 0$  gilt. Außerdem ist es ausreichend nur eine bestimmte Menge an Zeitpunkten zu testen, die durch die Menge  $P$  gegeben sind:

$$P_i(t) = \begin{cases} P_0(t) = \{t\} \\ P_i(t) = P_{i-1} \left( \left\lfloor \frac{t}{T_i} \right\rfloor T_i \right) \cup P_{i-1}(t). \end{cases} \quad (6.12)$$

### Implementierung

Der  $\delta$ -HET ist in der Methode `het_workload_test()` (`workload.py`) in Anlehnung an den Pseudo-C-Code aus [BB01; BB04] implementiert. Abbildung 6.7 zeigt den Programmablaufplan der Methode. Wie alle bisherigen Tests erhält die Funktion ein Task-Set als Eingabe. Zunächst werden zwei Listen für die Arbeitsbelastungen der Tasks mit 0 initialisiert. Diese Listen sind notwendig, um die Arbeitsbelastungen von Tasks in einem bestimmten Intervall nicht doppelt berechnen zu müssen. In einer Schleife über alle Tasks des Task-Sets wird für jeden Task die Arbeitsbelastung durch höher-priorisierte Tasks mit Hilfe der Methode `_w_i_het()` berechnet. Anschließend wird die Ausführungszeit des Check-Task addiert. Ist die Summe größer als die Deadline des Tasks, verfehlt der Task seine Deadline und das Task-Set ist nicht lauffähig. Andernfalls wird der nächste Task überprüft. Halten alle Tasks ihre Deadlines ein, so ist das Task-Set lauffähig.

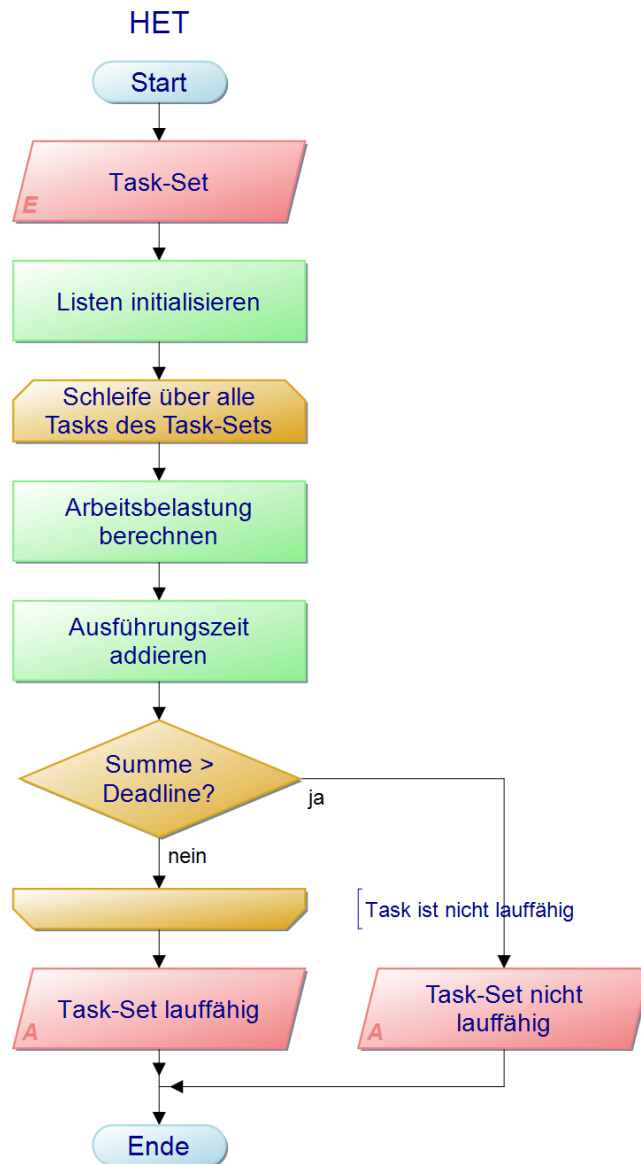


Abbildung 6.7: Programmablaufplan des Hyperplanes  $\delta$ -Exact Tests ( $\delta$ -HET) (Eigene Darstellung).

### Berechnung der Arbeitsbelastung

Die Methode `_W_i_het()` berechnet die Arbeitsbelastung durch alle Tasks mit höherer Priorität als Task  $\tau_i$  im Intervall  $[0, b]$ . Diese Arbeitsbelastung wird nachfolgend als  $W_i(b)$  bezeichnet und folgt durch Umformung aus Gleichung 6.11:

$$W_i(b) = \min_{t \in P_i(b)} \sum_{j=1}^i \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t). \quad (6.13)$$

Gemäß [BB01; BB04] wird die Menge  $P$  der Planungspunkte, wie sie Gleichung 6.12 definiert, in zwei Untermengen aufgespalten, und durch weitere Umformungen ergibt sich für die Arbeitsbelastung die folgende rekurrente Form mit  $f = \lfloor b/T_i \rfloor$  und  $c = \lceil b/T_i \rceil$ :

$$W_i(b) = \min\{b - f(T_i - C_i) + W_{i-1}(fT_i), cC_i + W_{i-1}(b)\}. \quad (6.14)$$

Die Methode `_W_i_het()` implementiert Gleichung 6.14 und erhält deshalb als Eingabeparameter den Index  $i$  des Check-Tasks, die Deadline  $b$  des Check-Task und das Task-Set selbst. Das Task-Set muss dabei nach aufsteigenden Prioritäten sortiert sein. Der erstellte Python-Code ist in Listing 6.13 gezeigt.

Listing 6.13: Bestimmung der Arbeitsbelastung beim  $\delta$ -HET

```
_last_psi = [ ]
_last_workload = [ ]

def _W_i_het(i, b, taskset):
    if i <= 0: # W_0(T_1) = 0
        return 0

    global _last_psi, _last_workload
    if b <= _last_psi[i]: # wenn W(i, b) bereits berechnet ist
        return _last_workload[i] # nicht weiter gehen

    f = math.floor(b / taskset[i-1].period)
    c = math.ceil(b / taskset[i-1].period)
    branch0 = b - f*(taskset[i-1].period - taskset[i-1].execution_time)
        + _W_i_het(i-1, f*taskset[i-1].period, taskset)
    branch1 = c*taskset[i-1].execution_time + _W_i_het(i-1, b, taskset)
    _last_psi[i] = b
    _last_workload[i] = min(branch0, branch1)
    return _last_workload[i]
```

Das  $i$ -te Element der Liste `_last_workload` enthält die Arbeitsbelastung, die durch den Aufruf von `_W_i_het(i, _last_psi[i])` berechnet wird. Wird dieser Wert erneut für die Berechnung einer anderen Arbeitsbelastung benötigt, muss er nicht noch einmal berechnet werden. Wie bereits beschrieben ist die Arbeitsbelastung des Tasks mit der höchsten Priorität per Definition  $W_0 = 0$ , weshalb dieser Fall zuerst geprüft wird. Es wird getestet, ob die zu berechnende Arbeitsbelastung bereits berechnet ist und in den Listen `_last_psi` und `_last_workload` gespeichert ist. Andernfalls wird die Arbeitsbelastung wie in Gleichung 6.14 berechnet und zurückgegeben.

### Beispiel

Nachfolgend wird wieder das Task-Set 46 429 verwendet, um beispielhaft den  $\delta$ -HET durchzuführen. Nach aufsteigenden Prioritäten sortiert besteht das Task-Set aus den Tasks 49, 227 und 104. Task 49 hat die höchste Priorität und deshalb den Index  $i = 1$ . Es ergibt sich für Task 49 nach Gleichung 6.10:

$$\begin{aligned} C_1 + W_0(D_1) &= 178 \text{ ms} + W_0(5500 \text{ ms}) = 178 \text{ ms} \leq D_{49} = 5500 \text{ ms} \\ W_0(5500 \text{ ms}) &= 0 \text{ ms.} \end{aligned}$$

Da Task 49 keinen höher-priorisierten Task hat, ergibt sich die Arbeitsbelastung  $W_0(D_1)$  zu 0 und der Task hält seine Deadline ein.

Der zweite Task mit Index  $i = 2$  ist Task 227 und die Berechnung für diesen Task lautet:

$$\begin{aligned} C_2 + W_1(D_2) &= 105 \text{ ms} + W_1(1500 \text{ ms}) = 283 \text{ ms} \leq D_2 = 1500 \text{ ms} \\ \left[ \begin{array}{l} W_1(b = 1500 \text{ ms}) = \min\{\text{branch0}, \text{branch1}\} = 178 \text{ ms} \\ f = \left\lfloor \frac{b}{T_1} \right\rfloor = \left\lfloor \frac{1500 \text{ ms}}{6000 \text{ ms}} \right\rfloor = 0 \quad c = \left\lceil \frac{b}{T_1} \right\rceil = \left\lceil \frac{1500 \text{ ms}}{6000 \text{ ms}} \right\rceil = 1 \\ \text{branch0} = b - f(T_1 - C_1) + W_0(fT_1) = \\ \quad = 15000 \text{ ms} - 0 \cdot (6000 \text{ ms} - 178 \text{ ms}) + W_0(0 \cdot 6000 \text{ ms}) = 1500 \text{ ms} \\ \text{branch1} = cC_1 + W_0(b) = 1 \cdot 178 \text{ ms} + W_0(1500 \text{ ms}) = 178 \text{ ms} \end{array} \right. \end{aligned}$$

Die Arbeitsbelastung  $W_1(D_2)$  berechnet sich zu 178 ms, sodass sich in Summe 283 ms ergeben, was unter der Deadline von Task 227 liegt. Damit ist der Task hiernach lauffähig.

Der letzte Task 104 hat die niedrigste Priorität und den Index  $i = 3$ . Die Berechnung erfolgt auf folgende Weise:

$$C_3 + W_2(D_3) = 4878 \text{ ms} + W_2(6500 \text{ ms}) = 5654 \text{ ms} < D_3 = 6500 \text{ ms}$$

$$\left[ \begin{array}{l} W_2(b = 6500 \text{ ms}) = \min\{\text{branch0}, \text{branch1}\} = 776 \text{ ms} \\ f = \left\lfloor \frac{b}{T_2} \right\rfloor = \left\lfloor \frac{6500 \text{ ms}}{2000 \text{ ms}} \right\rfloor = 3 \quad c = \left\lceil \frac{b}{T_2} \right\rceil = \left\lceil \frac{6500 \text{ ms}}{2000 \text{ ms}} \right\rceil = 4 \\ \\ \text{branch0} = b - f(T_2 - C_2) + W_1(fT_2) = \\ \quad = 6500 \text{ ms} - 3 \cdot (2000 \text{ ms} - 105 \text{ ms}) + W_1(3 \cdot 2000 \text{ ms}) = 993 \text{ ms} \\ \left[ \begin{array}{l} W_1(b = 6000 \text{ ms}) = \min\{\text{branch00}, \text{branch01}\} = 178 \text{ ms} \\ f = \left\lfloor \frac{b}{T_1} \right\rfloor = \left\lfloor \frac{6000 \text{ ms}}{6000 \text{ ms}} \right\rfloor = 1 \quad c = \left\lceil \frac{b}{T_1} \right\rceil = \left\lceil \frac{6000 \text{ ms}}{6000 \text{ ms}} \right\rceil = 1 \\ \\ \text{branch0} = b - f(T_1 - C_1) + W_0(fT_1) = \\ \quad = 6000 \text{ ms} - 1 \cdot (6000 \text{ ms} - 178 \text{ ms}) + W_0(1 \cdot 6000 \text{ ms}) = 178 \text{ ms} \\ \text{branch1} = cC_1 + W_0(b) = 1 \cdot 178 \text{ ms} + W_0(6000 \text{ ms}) = 178 \text{ ms} \end{array} \right. \\ \\ \text{branch1} = cC_2 + W_1(b) = 4 \cdot 105 \text{ ms} + W_1(6500 \text{ ms}) = 776 \text{ ms} \\ \left[ \begin{array}{l} W_1(b = 6500 \text{ ms}) = \min\{\text{branch10}, \text{branch11}\} = 356 \text{ ms} \\ f = \left\lfloor \frac{b}{T_1} \right\rfloor = \left\lfloor \frac{6500 \text{ ms}}{6000} \right\rfloor = 1 \quad c = \left\lceil \frac{b}{T_1} \right\rceil = \left\lceil \frac{6500 \text{ ms}}{6000 \text{ ms}} \right\rceil = 2 \\ \\ \text{branch0} = b - f(T_1 - C_1) + W_0(fT_1) = \\ \quad = 6500 \text{ ms} - 1 \cdot (6000 \text{ ms} - 178 \text{ ms}) + W_0(1 \cdot 6000 \text{ ms}) = 678 \\ \text{branch1} = cC_1 + W_0(b) = 2 \cdot 178 \text{ ms} + W_0(6500 \text{ ms}) = 356 \text{ ms} \end{array} \right. \end{array} \right.$$

Die Arbeitsbelastung durch die Tasks 227 und 49 berechnet sich zu 776 ms, was in Summe mit der Ausführungszeit von Task 104 den Wert 5654 ms ergibt. Dieses Ergebnis liegt deutlich unter der Deadline von Task 104, was bedeutet, dass der Task seine Deadline nicht verfehlt. Somit ist das Task-Set 46 429 nach dem  $\delta$ -HET lauffähig.

Abschließend sei angemerkt, dass sich der Rechenweg und die resultierenden Werte des  $\delta$ -HET und der Antwortzeit-Analyse sehr ähneln. Allerdings ist laut [BB01; BB04] der  $\delta$ -HET im Allgemeinen deutlich effizienter und schneller als die Antwortzeit-Analyse.

## 6.11 Evaluation

Zur Evaluation werden die Schedulability-Analyseverfahren mit den Daten aus Kapitel 5 getestet. Um die Ergebnisse auszuwerten und miteinander zu vergleichen, sind zusätzliche Begriffe notwendig, die im Folgenden kurz erklärt werden.

Jeder vorgestellte Schedulability-Test ist im Grunde ein Klassifikator, der ein Task-Set in die Klassen „lauffähig“ und „nicht lauffähig“ einordnet. Beantwortet der Klassifikator

eine Ja/Nein-Frage, wie hier die Frage „Ist das Task-Set lauffähig?“, wird von einem positiven (Einordnung als „lauffähig“) oder negativen (Einordnung als „nicht lauffähig“) Test gesprochen [Wik18]. Zur Beurteilung eines Klassifikators wird das Testergebnis mit dem wirklichen Ergebnis verglichen, wobei sich vier mögliche Fälle ergeben, die auch als „Wahrheitsmatrix“ bezeichnet werden:

- Richtig positiv (engl. *true positive*, *tp*): das Task-Set ist lauffähig und der Test hat dies richtig angezeigt.
- Falsch positiv (engl. *false positive*, *fp*): das Task-Set ist nicht lauffähig, aber der Test hat es fälschlicherweise als lauffähig eingestuft.
- Richtig negativ (engl. *true negative*, *tn*): das Task-Set ist nicht lauffähig und der Test hat dies richtig angezeigt.
- Falsch negativ (engl. *false negative*, *fn*): das Task-Set ist lauffähig, aber der Test hat es fälschlicherweise als nicht lauffähig eingestuft.

Aus diesen Werten lassen sich Kenngrößen zur Beurteilung eines Klassifikators ableiten. Die Korrektklassifikationsrate oder auch Treffergenauigkeit (engl. *score* oder *accuracy*) ist der Anteil aller Objekte die korrekt klassifiziert werden:

$$P(\text{richtig klassifiziert}) = \frac{\text{richtig klassifiziert}}{\text{alle Fälle}} = \frac{tp + tn}{tp + fp + tn + fn}. \quad (6.15)$$

Die Genauigkeit (engl. *precision*) ist der Anteil der korrekt als positiv klassifizierten Ergebnissen an der Gesamtheit der als positiv klassifizierten Ergebnissen:

$$P(\text{tatsächlich lauffähig}|\text{positives Testergebnis}) = \frac{tp}{tp + fp}. \quad (6.16)$$

Die Trefferquote oder auch Richtig-positiv-Rate (engl. *recall*) ist der Anteil der korrekt als positiv klassifizierten Objekten an der Gesamtheit der tatsächlich positiven Objekten:

$$P(\text{positives Testergebnis}|\text{tatsächlich lauffähig}) = \frac{tp}{tp + fn}. \quad (6.17)$$

Die Ergebnisse der verschiedenen traditionellen Schedulability-Analyseverfahren sind in Tabelle 6.5 zusammengefasst.

Die Korrektklassifikationsrate (engl. *accuracy*) bewegt sich bei allen Verfahren zwischen 87,89 % und 93,00 %, wobei die Antwortzeit-Analyse das beste Ergebnis zeigt, und der auf der Hyperbolic Bound basierende Prozessorauslastungs-Test das schlechteste. Die Simulation zeigt mit 91,66 % nur das zweitbeste Ergebnis, obgleich eigentlich die Erwartung ist, dass die Simulation das beste Ergebnis zeigt. Dennoch ist das Ergebnis plausibel, da die Antwortzeit-Analyse ebenfalls ein exakter Schedulability-Test ist und die Ergebnisse beider Verfahren in derselben Größenordnung liegen. Genauso ist auch



Tabelle 6.5: Ergebnisse der traditionellen Schedulability-Analyseverfahren (Eigene Darstellung).

	Simulation	Prozessorauslastung			Antwortzeit-Analyse		Arbeitsbelastung	
		einfach	RM	HB	Audsley	Buttazzo	RM	$\delta$ -HET
tp	1 539 991	1 577 634	1 543 295	1 582 226	1 533 263	1 533 259	1 552 528	1 506 046
fp	115 206	290 025	223 402	296 262	81 022	80 904	160 544	109 803
tn	340 766	165 947	232 570	159 710	374 950	375 068	295 428	346 169
fn	55 915	18 272	52 611	13 680	62 643	62 647	43 378	89 860
Gesamt	2 051 878	2 051 878	2 051 878	2 051 878	2 051 878	2 051 878	2 051 878	2 051 878
tp	75,05 %	76,89 %	75,21 %	77,11 %	74,72 %	74,72 %	75,66 %	73,40 %
fp	5,61 %	14,13 %	10,89 %	14,44 %	3,95 %	3,94 %	7,82 %	5,35 %
tn	16,61 %	8,09 %	11,33 %	7,78 %	18,27 %	18,28 %	14,40 %	16,87 %
fn	2,73 %	0,89 %	2,56 %	0,67 %	3,05 %	3,05 %	2,11 %	4,38 %
accuracy	91,66 %	84,97 %	86,55 %	84,89 %	93,00 %	93,00 %	90,06 %	90,27 %
precision	93,04 %	84,47 %	87,35 %	84,23 %	94,98 %	94,99 %	90,63 %	93,20 %
recall	96,50 %	98,86 %	96,70 %	99,14 %	96,07 %	96,07 %	97,28 %	94,37 %
Laufzeit	2 h 8 min 36 s	7 s	9 s	6 s	1 min 24 s	1 min 23 s	1 min 27 s	1 min 22 s

das im Vergleich schlechte Abschneiden der Prozessorauslastungs-Tests nicht weiter verwunderlich, da sie zum einen nicht exakt sind, und zum anderen auch nicht für den in dieser Arbeit angenommenen Scheduler entworfen sind. Trotzdem ist der prozentuale Unterschied von gerade einmal ca. 5 % überraschend.

Die Genauigkeit (engl. *precision*), das bedeutet der Anteil aller als lauffähig klassifizierten Task-Sets die auch tatsächlich lauffähig sind, liegt bei allen Verfahren zwischen 84,23 % und 94,99 %. Auch hier sind die Werte der exakten Analysen sehr ähnlich mit Ergebnissen von 93,04 % bis 94,99 %, und die Prozessorauslastungs-Tests liegen mit ca. 7 % Unterschied dahinter. Die Genauigkeit bestätigt deshalb die Feststellungen, die bereits in Abhängigkeit von der Korrektklassifikationsrate getroffen sind.

Die Trefferquote (engl. *recall*) ist für den Hyperplanes  $\delta$ -Exakt Test mit 94,37 % am geringsten, das bedeutet 94,37 % aller lauffähigen Task-Sets werden auch als solche eingestuft. Den höchsten Wert von 99,14 % erreichte der Prozessorauslastungs-Test mit der Hyperbolic Bound. Hier zeigt sich eine im Vergleich zur Korrektklassifikationsrate und der Genauigkeit gegenteilige Werteverteilung: die Prozessorauslastungs-Tests zeigen mit Werten zwischen 96,70 % und 99,14 % deutlich bessere Ergebnisse als die exakten Tests mit Werten von 94,37 % bis 96,50 %. Zu erklären ist dies damit, dass die Prozessorauslastungs-Tests offensichtlich deutlich häufiger ein Task-Set als lauffähig einstufen und damit optimistischer sind als die exakten Tests.

Dieses Verhalten zeigt sich auch, wenn die falsch positiv (fp) eingeordneten Task-Sets betrachtet werden. Die exakten Analysen haben mit Werten von 3,94 % bis 5,61 % eine geringe nicht exakten Tests zwischen 10,89 % und 14,44 % der Task-Sets falsch positiv klassifiziert haben. Aber gerade dieser Faktor ist in sicherheitskritischen Systemen extrem wichtig, da hiervon die korrekte Funktion des Systems abhängt. Kann ein Task-Set, das fälschlicherweise als lauffähig eingeordnet wird, nicht korrekt ablaufen, hat das negative Folgen für das gesamte System. Deshalb zeichnet eine gute Schedulability-Analysemethode ein niedriger Wert bei den falsch positiven Task-Sets aus. Das beste Ergebnis liefert die Antwortzeit-Analyse mit einem Wert von 3,94 %.

Wie bereits unter 6.10.2 genannt, ist laut [BB01; BB04] der  $\delta$ -HET im Allgemeinen deutlich effizienter und schneller als die Antwortzeit-Analyse. Die Ergebnisse dieser Arbeit bestätigen dies nur bedingt. Die Ausführungszeit des  $\delta$ -HET ist mit 1 min 22 s bei Ausführung auf einem Rechnercluster aus 40 CPUs<sup>3</sup> zwar etwas schneller als die Antwortzeit-Analyse mit 1 min 24 s bzw. 23 s, allerdings ist das Ergebnis auch etwa 3 % schlechter. Der  $\delta$ -HET zeigt in den Untersuchungen zwar schneller ein Ergebnis für die Klassifikation als die Antwortzeit-Analyse, aber kein besseres, und ist folglich nicht effizienter.

---

<sup>3</sup>Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz

Wie aus Tabelle 6.5 ersichtlich ist, spielt der Startwert bei der Antwortzeit-Analyse keine signifikante Rolle. Beide Verfahren liefern beinahe identische Ergebnisse und unterscheiden sich in der Ausführungszeit nur um 1,5 s. Möglicherweise sind die in dieser Arbeit verwendeten Task-Sets mit einer maximalen Größe von vier Tasks aber auch zu klein, um einen deutlichen Unterschied zwischen den Startwerten zu erkennen, da die Berechnungsdauer mit steigender Task-Anzahl je Task-Set proportional zunimmt.

Wird die Rechenzeit betrachtet, die die Verfahren für die  $\sim 2$  Mio. Task-Sets aus der Datenbank auf einem Rechnercluster aus 40 CPUs<sup>3</sup> benötigen, so lässt sich feststellen, dass die exakten Verfahren eindeutig komplexer sind als die Prozessorauslastungs-Tests, die sich im Bereich von bis zu 10 s bewegen, was  $5 \mu\text{s}$  je Task-Set entspricht. Die Simulation benötigt die längste Rechenzeit mit über zwei Stunden bzw. ungefähr 4 ms pro Task-Set. Die beiden auf der Arbeitsbelastung basierenden Tests und die Antwortzeit-Analysen liegen mit Werten im Bereich von 1 min 23 s, was  $42 \mu\text{s}$  je Task-Set entspricht, nahe an den nicht exakten Prozessorauslastungs-Tests.

Wie bereits erwähnt, ist es auffällig, dass die Simulation als exaktes Schedulability-Analyseverfahren nur eine Korrekturklassifikationsrate von 91,66 % erreicht und nicht die erwartungsgemäßen 100 %. Das deutet darauf hin, dass die Datenbank auch Task-Sets enthält, die auf dem PandaBoard ES Cluster nicht erfolgreich ablaufen, obwohl dies theoretisch möglich ist.<sup>4</sup> Eine weitere Erklärung ist auch die zur Simulation und für die anderen Berechnungen verwendete Ausführungszeit im schlimmsten Fall (WCET). In dieser Arbeit wird hierfür ein Durchschnittswert bestimmt, da keine anderen Werte für die WCET vorhanden sind. Dies führt zwangsläufig zu unterschiedlichen Rechenergebnissen, was ebenfalls die Abweichung erklärt.

Zusammenfassend lässt sich konstatieren, dass alle Ergebnisse sehr ähnlich sind und es keine signifikanten Unterschiede gibt. Die exakten Tests, das heißt die Simulation, Antwortzeit-Analyse und der  $\delta$ -HET, liefern insgesamt bessere Ergebnisse als die nicht exakten Tests. Überraschenderweise liefern aber auch die Analysemethoden, die nicht für den FP-Scheduler gedacht sind, Ergebnisse in der gleichen Größenordnung. Das beste Ergebnis liefert die Antwortzeit-Analyse mit dem Startwert aus [But11] mit einer Korrekturklassifikationsrate von 93 %, nur 3,94 % falsch positiven Task-Sets und einer Ausführungszeit von ungefähr  $40 \mu\text{s}$  je Task-Set.

---

<sup>4</sup>**Anmerkung:** Erst am Ende dieser Arbeit wurde die Ursache hierfür im Verhalten des Genode-Betriebssystems gefunden (siehe Kapitel 9, Abschnitt „Gegebene Daten“).



## 7 Schedulability-Analyse mit einem rekurrenten neuronalen Netz

Als Vergleich zu den gängigen Schedulability-Analyseverfahren aus Kapitel 6 wird in dem Projekt *RNN-SA*<sup>1</sup> ein neuronales Netz, genauer ein rekurrentes neuronales Netz (RNN), implementiert und trainiert. Die Idee hinter der Schedulability-Analyse mit einem neuronalen Netz ist, dass Machine Learning Modelle Muster in den Task-Sets erlernen und damit schnellere und bessere Entscheidungen bezüglich der Lauffähigkeit von unbekannten Task-Sets treffen als die üblichen Ansätze. Außerdem wird angenommen, dass Task-Sets dynamisch als Sequenz von nacheinander auftretenden Tasks entstehen. Am naheliegendsten ist deshalb die Anwendung eines rekurrenten neuronalen Netzes, das speziell für die Verarbeitung von sequentiellen Daten konstruiert ist, da die rekurrenten Verbindungen als eine Art Gedächtnis für vergangene Zeitschritte dienen.

Die Ergebnisse der traditionellen Schedulability-Analyseverfahren zeigen, dass die verwendete Datenbank auch Task-Sets enthält, die auf dem PandaBoard ES Cluster nicht erfolgreich ablaufen, obwohl dies theoretisch möglich ist, da selbst die exakten Methoden wie Simulation, Antwortzeit-Analyse und  $\delta$ -HET keine 100 %-ige Genauigkeit erreichen. Deshalb wird ein Filter erstellt, der die durch ein exaktes Verfahren verifizierten Task-Sets aus der Datenbank filtert. Diese Task-Sets werden für das Training eines RNNs verwendet. Auf diese Weise wird sichergestellt, dass das neuronale Netz keine falschen Muster erlernt, da die Ursache für den Unterschied zwischen der realen Ausführung und der theoretischen Analyse noch nicht geklärt ist.<sup>2</sup> Die angepasste Toolchain des *MaLSAMi*-Projektes ist in Abbildung 7.1 zu sehen.

Abbildung 7.2 stellt den Aufbau des *RNN-SA*-Projektes dar. Als Eingabe dient wieder eine Datenbank, auf die mit Hilfe der Datenbankschnittstelle, wie in Kapitel 7.2 beschrieben, zugegriffen wird. Wie schon im *traditional-SA*-Projekt ist ein Benchmark-Modul enthalten, das die durchschnittliche Laufzeit der Tasks bestimmt. Die Filterung der Datenbank mit Hilfe der Antwortzeit-Analyse aus dem *traditional-SA*-Projekt ist in Kapitel 7.4 dokumentiert. Das Hauptprogramm erstellt und trainiert ein rekurrentes neuronales Netz. Die verschiedenen Arten von RNNs sind im Modul *ML-Modelle* gespeichert. Die Einstellungen für das RNN und dessen Training sind durch die Datei *params.py* vorgegeben und werden dort durch den Benutzer angepasst. Das Hauptprogramm enthält auch die Hyperparameter-Optimierung, die für verschiedene Parameter neuronale

---

<sup>1</sup><https://github.com/TatjanaUtz/RNN-SA>

<sup>2</sup>**Anmerkung:** Erst am Ende dieser Arbeit wurde die Ursache im Verhalten des Genode-Betriebssystems gefunden (siehe Kapitel 9, Abschnitt „Gegebene Daten“)

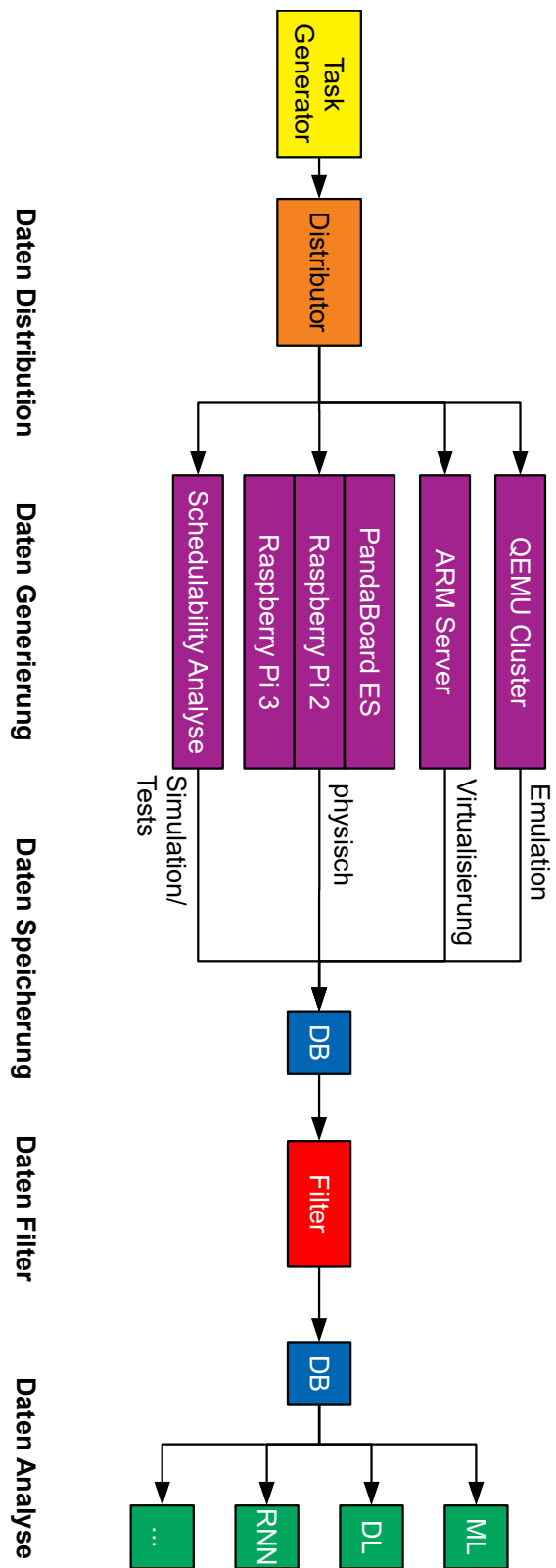


Abbildung 7.1: Aufbau der Toolchain des *ML5AMi*-Projektes mit Datenbank-Filter (Eigene Darstellung).

Netze trainiert und deren Leistungen miteinander vergleicht. Die Ergebnisse dieser Experimente werden in eine CSV-Datei gespeichert und mit dem Plotting-Modul visualisiert. Alle anderen Informationen werden über das Logging-Modul auf der Konsole und in eine Fehler-Datei ausgegeben.

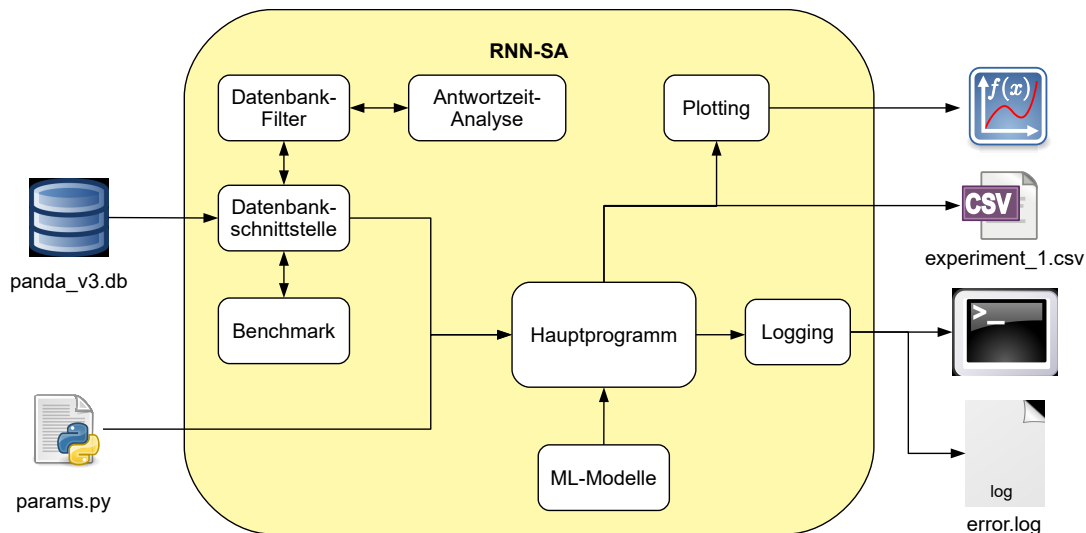


Abbildung 7.2: Blockdiagramm des *RNN-SA*-Projekts, der Komponente für die Schedulability-Analyse mit einem RNN (Eigene Darstellung).

## 7.1 Hauptprogramm

Das Hauptprogramm ist in der Datei `main.py` in der `main()`-Funktion enthalten. Der Python-Code ist in Listing 7.1 gezeigt.

Zuerst wird das Verzeichnis und der Name der Datenbank festgelegt. In der aktuellen Version befindet sich die Datenbank im Arbeitsverzeichnis des *RNN-SA*-Projektes, das durch `os.getcwd()` erhalten wird. Dann wird das Logging-Modul initialisiert. Anschließend werden die Daten aus der Datenbank gelesen und für das Machine Learning vorbereitet. Die Hyperparameter-Optimierung findet mit Hilfe des Frameworks Talos statt und ist in der Funktion `hyperparameter_exploration()` definiert. Soll nur ein rekurrentes neuronales Netz mit bestimmten Parametern trainiert und evaluiert werden, ist dies mit der Funktion `train_and_evaluate()` möglich.

### 7.1.1 Laden der Daten

Die Daten zum Trainieren, Evaluieren und Testen des neuronalen Netzes werden durch die Methode `load_data()` aus der Datenbank gelesen und vorverarbeitet. Als Eingabeparameter erhält die Funktion das Verzeichnis und den Namen der Datenbank. Mit

Listing 7.1: Hauptprogramm des RNN-SA-Projektes

```
def main():
    # Verzeichnis und Name der Datenbank festlegen
    db_dir, db_name = os.getcwd(), "panda_v3.db"

    # Logger erstellen und initialisieren
    logger = logging_config.init_logging(db_dir, db_name)

    data = load_data(db_dir, db_name) # Daten laden

    #####
    ### HYPERPARAMETER-OPTIMIERUNG MIT TALOS ###
    #####
    hyperparameter_exploration(data=data, name='experiment', num='1')

    #####
    ### SINGLE KERAS-MODELL ###
    #####
    # ein Keras-Modell trainieren und evaluieren
    train_and_evaluate(data)
```

---



Hilfe dieser Informationen wird eine Verbindung zur Datenbank hergestellt und die komplette Tabelle *CorrectTaskSet* gelesen. Außerdem werden die gelesenen Reihen mit `random.shuffle(rows)`

gemischt, um Bias oder andere Muster im Datensatz zu vermeiden. Das verbessert die Qualität und Vorhersagegenauigkeit des Machine Learning Modells. Um das Mischen reproduzierbar zu machen, was notwendig für die Hyperparameter-Optimierung ist, wird das Python-Modul `random` folgendermaßen initialisiert:

```
import random
random.seed(4).
```

Dadurch werden die Reihen aus der Tabelle immer auf die gleiche Art und Weise durcheinandergewürfelt.

Durch die Methode `_split_tasksets()` werden die Reihen in zwei Listen aufgeteilt:

- Task-Sets in Form von Tupel aus Task-IDs,
- Labels, wobei 0 für „nicht lauffähig“ und 1 für „lauffähig“ steht.

Dann wird die vollständige Tabelle *Task* aus der Datenbank gelesen und die Task-Attribute ebenfalls als Reihen, nicht als Task-Objekte, gespeichert. Die Vorverarbeitung der Task-Attribute geschieht durch die Funktion `_preprocess_task_attributes()` und wird in Kapitel 7.1.2 genauer geschildert. In einer Schleife über alle Task-Sets werden alle ungültigen Task-IDs, das heißt alle Tasks mit  $ID = -1$ , gelöscht und die gültigen Task-IDs durch die entsprechenden Task-Attribute-Tupel ersetzt. Das resultierende dreidimensionale Array sieht wie in Abbildung 7.3 aus, wobei beispielhaft die beiden Task-Sets 46 429 und 563 782 aus Tabelle 5.1 dargestellt sind.

Die erste Dimension, die x-Richtung, speichert die Task-Attribute, zum Beispiel die Priorität oder die Deadline des Task. Die zweite Dimension, die y-Richtung, repräsentiert die Tasks eines Task-Sets oder anders ausgedrückt die Zeitschritte, wenn das Task-Set als Sequenz aus Tasks angesehen wird. Die dritte Dimension, die z-Richtung, stellt die Task-Sets dar, das heißt die Menge an Trainings-Beispielen.

Anschließend werden die Task-Sets durch die Methode `_pad_sequences()` auf eine einheitliche Länge gebracht. Hierfür wird die Funktion `keras.preprocessing.sequence.pad_sequences()` aus dem Keras-Framework aufgerufen, die alle Task-Sets mit weniger Tasks als das größte Task-Set mit Nullen auffüllt und ein NumPy-Array der Form

$$\text{Anzahl Task-Sets} \times \text{maximale Anzahl Tasks je Task-Set} \times \text{Anzahl Attribute je Task}$$

zurückgibt. Diese Dimensionen werden in den Konfigurationsparametern `time_steps`, für die maximale Anzahl der Tasks je Task-Set, und `element_size`, für die Anzahl der Attribute je Task, gespeichert. Zur konfliktfreien und einheitlichen Weiterverarbeitung wird auch die Liste der Labels in ein NumPy-Array umgewandelt:

```
labels_np = np.asarray(labels, np.int32).
```

Task 1	Prio	PKG	Arg	D	Periode	#Jobs
Task 2						
⋮	⋮	⋮	⋮	⋮	⋮	⋮

4	tumatmul	65536	1500	2000	2
4	pi	2097152	7500	8000	3
1	tumatmul	4096	500	1000	4

4	cond_mod	22876792454961	6500	7000	3
1	pi	131072	5500	6000	7
2	pi	32768	1500	2000	3

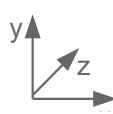


Abbildung 7.3: Form der Eingabe-Daten des RNNs anhand der beiden Beispiel-Task-Sets 46 429 und 563 782 (Eigene Darstellung).

Um die Daten in einen Trainings-, Evaluations- und Testdatensatz aufzuteilen, wird die Funktion `sklearn.model_selection.train_test_split()` aus dem Sklearn-Paket verwendet. Hierfür wird zunächst ein leeres Dictionary `data` erstellt, das am Ende die Task-Sets und Labels für jeden Datensatz enthält. Die Daten werden mit

```
data['train_X'], test_val_x, data['train_y'], test_val_y =
    sklearn.model_selection.train_test_split(tasksets_np, labels_np,
        test_size=0.2, random_state=42)
```

aufgeteilt: 80 % der Daten werden als Trainingsdatensatz gespeichert, der Rest wird durch einen weiteren Aufruf der `train_test_split()`-Methode in einen Evaluations- und einen Testdatensatz mit jeweils 10 % der Daten aufgeteilt. Durch den Parameter `random_state` wird auch hier die Aufteilung der Daten reproduzierbar gemacht.

Am Ende der Methode `load_data()` wird das erstellte Daten-Dictionary `data` zurückgegeben.

### 7.1.2 Datenvorverarbeitung

Die Datenvorverarbeitung bildet die Grundlage für den Lernerfolg eines Machine Learning Modells. Da die Daten meist nicht in genau dem richtigen Format und in perfekter Qualität zur Verfügung stehen, müssen die Daten entsprechend vorbereitet werden. Dabei gilt es zum Beispiel fehlende Werte, doppelt vorhandene Beispiele, Zeichendreher, Tippfehler, uneinheitliche Einheiten und unplausible Daten zu korrigieren [NZ18].

Die in dieser Arbeit verwendete Datenbank enthält keine der genannten Fehler. Es müssen lediglich die Task-Attribute in ein für das neuronale Netz nutzbares Format gebracht werden und unnütze bzw. redundante Informationen entfernt werden. Diese

Aufgabe wird während des Ladens der Daten durch die Methode `_preprocess_task_attributes()` übernommen, die als Eingabe eine Liste mit den Tasks erhält.

Die in der Tabelle *Task* verfügbaren Attribute sind in der konstanten Liste `DEFAULT_FEATURES` gespeichert, wobei die Position innerhalb der Liste der Spalten-Nummer in der Tabelle *Task* entspricht:

```
DEFAULT_FEATURES = ['Task_ID', 'Priority', 'Deadline', 'Quota', 'CAPS',
                    'PKG', 'Arg', 'CORES', 'COREOFFSET', 'CRITICALTIME', 'Period',
                    'Number_of_Jobs', 'OFFSET'].
```

Die nachfolgend beschriebenen Schritte der Datenvorverarbeitung werden in Tabelle 7.1 anhand des Tasks 49 beispielhaft gezeigt.

Tabelle 7.1: Schritte der Datenvorverarbeitung am Beispiel von Task 49 (Eigene Darstellung).

Aktion	Features	Task-Attribute
Ausgangspunkt	Task_ID, Priority, Deadline, Quota, CAPS, PKG, Arg, CORES, COREOFFSET, CRITICALTIME, Period, Number_of_Jobs, OFFSET	49, 1, 0, 100M, 235, pi, 131072, 2, 1, 5500, 6000, 7, 0
<code>_delete_unused_features()</code>	Priority, PKG, Arg, CRITICALTIME, Period, Number_of_Jobs	1, pi, 131072, 5500, 6000, 7
<code>_one_hot_encoding()</code>	Priority, PKG_cond_mod, PKG_hey, PKG_pi, PKG_tumatmul, Arg, CRITICALTIME, Period, Number_of_Jobs	1, 0, 0, 1, 0, 131072, 5500, 6000, 7
<code>_standardize()</code>	Priority, PKG_cond_mod, PKG_hey, PKG_pi, PKG_tumatmul, Arg, CRITICALTIME, Period, Number_of_Jobs	0.0, 0.0, 0.0, 1.0, 0.0, 6.366034256382969e-10, 0.7142857142857144, 0.7142857142857143, 0.8571428571428572

### Unnötige Features löschen

Als Erstes werden alle unnötigen Features, das heißt Attribute, die für alle Tasks den gleichen Wert haben (z.B. *Quota*) oder keine relevante Information beinhalten (z.B. *Task\_ID*),

durch die Methode `_delete_unused_features()` gelöscht. Welche Features genutzt und damit nicht gelöscht werden, ist in der konstanten Liste

```
USE_FEATURES = ['Priority', 'PKG', 'Arg', 'CRITICALTIME', 'Period', 'Number_of_Jobs']
```

festgehalten. Diese sind die Priorität, der Name des Tasks (*PKG*), das Argument, die kritische Zeit (= Deadline), die Periode und die Anzahl der Jobs. Als Eingabe erhält die Methode eine Liste mit den Tasks und eine Liste mit den momentan gespeicherten Features. In einer Schleife über diese Features, beginnend beim letzten Attribut *OFFSET*, wird überprüft, ob das Feature gelöscht werden soll. Falls ja, wird die entsprechende Spalte aus der Task-Liste entfernt. Nach der Schleife wird die Feature-Liste aktualisiert, das bedeutet alle gelöschten Attribute werden daraus entfernt. Die Rückgabe der Methode ist die bearbeitete Task-Liste und die aktualisierte Feature-Liste.

### One-Hot-Kodierung

Künstliche neuronale Netze können keine Zeichenketten, sondern nur numerische Werte verarbeiten. Deshalb müssen alle Task-Attribute, die nicht-numerisch sind, zwingend in numerische Werte umgewandelt werden. Da das *PKG*-Attribut den Namen des Tasks bzw. der Funktion als String enthält, wird für dieses Feature eine One-Hot-Kodierung durchgeführt. Allgemein können nicht-numerische Merkmale auch auf andere Art und Weise in numerische Werte umgewandelt werden. Gibt es allerdings nur wenige Möglichkeiten für den Wert eines Attributs, ist die One-Hot-Kodierung ein gängiges und effizientes Mittel. Die Idee der One-Hot-Kodierung ist, für jeden der verschiedenen möglichen Werte ein neues Dummy-Merkmal zu erstellen [RM17]. Für das *PKG*-Attribut sind das die neuen Attribute *PKG\_cond\_mod*, *PKG\_hey*, *PKG\_pi* und *PKG\_tumatmul*. Damit wird der Task-Name binär beschrieben, indem das entsprechende Merkmal auf den Wert 1 und alle andere Merkmale auf den Wert 0 gesetzt werden. Die Kodierung des *PKG*-Attributs ist in dem konstanten Dictionary

```
PKG_ENCODING = {
    'cond_mod': [1, 0, 0, 0],
    'hey': [0, 1, 0, 0],
    'pi': [0, 0, 1, 0],
    'tumatmul': [0, 0, 0, 1]}
```

gespeichert und wird von der Methode `_one_hot_encoding()` durchgeführt. Als Eingabe erhält die Funktion eine Liste mit den Tasks und eine Liste mit den aktuellen Features. Das *PKG*-Attribut wird in jedem Task durch die One-Hot-Kodierung ersetzt und die neuen Merkmalsspalten der Feature-Liste hinzugefügt. Die Rückgabe der Methode ist die veränderte Task-Liste und die erweiterte Feature-Liste.

### Standardisierung

Die meisten Lern- und Optimierungsalgorithmen zeigen erheblich verbessertes Verhalten, wenn die Features von gleicher Größenordnung sind [RM17]. Um die Merkmale einer Datenmenge auf die gleiche Größenordnung zu bringen, sind zwei Ansätze gebräuchlich: die Normierung und die Standardisierung. Beide Verfahren sind in der Methode `_standardize()` implementiert, die als Eingabe eine Task-Liste erhält.

Bei der Normierung wird ein Feature auf das Intervall  $[0, 1]$  abgebildet. Sie ist somit ein Spezialfall der Min-Max-Skalierung. Die Formel für die Normierung eines Wertes  $x^{(i)}$  lautet:

$$x_{\text{norm}}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}, \quad (7.1)$$

wobei  $x_{\min}$  bzw.  $x_{\max}$  der minimale bzw. maximale Wert der gesamten Feature-Spalte  $x$  ist. Die Normierung der Task-Attribute ist wie folgt implementiert:

```
normalized = sklearn.preprocessing.MinMaxScaler(feature_range=(0, 1)).  
    fit_transform(task_attributes).
```

Bei der Standardisierung werden die Werte einer Merkmalsspalte um den Mittelwert 0 mit einer Standardabweichung von 1 zentriert, sodass sich eine Normalverteilung ergibt. Die Formel lautet:

$$x_{\text{std}}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}, \quad (7.2)$$

wobei  $\mu_x$  der Mittelwert und  $\sigma_x$  die Standardabweichung der Feature-Spalte  $x$  ist. Die Standardisierung der Task-Attribute ist folgendermaßen implementiert:

```
standardized = sklearn.preprocessing.StandardScaler().fit_transform(  
    task_attributes).
```

Welches Verfahren auf die Task-Attribute angewendet wird, muss durch Auskommen-tieren festgelegt werden. Im aktuellen Stand des *RNN-SA*-Projektes werden die Daten normiert und nicht standardisiert. Der Vollständigkeit halber und für spätere Versuche ist aber auch die Standardisierung implementiert.

## 7.2 Datenbankschnittstelle

Die Datenbankschnittstelle ist identisch zu der Schnittstelle aus dem Projekt *traditional-SA*. Es werden lediglich einige Methoden angepasst bzw. neu hinzugefügt, um die Datenbankschnittstelle auf die leicht veränderte Datenbank anzupassen. Im Einzelnen unterscheidet sich die Datenbank des *RNN-SA*-Projektes nur durch die zusätzliche Tabelle *CorrectTaskSet* von der im Projekt *traditional-SA*. Die Erstellung und Bedeutung dieser Tabelle wird in Kapitel 7.4 genauer erläutert.

### Tabellen prüfen

Die Methode `_check_database()` des *RNN-SA*-Projektes überprüft zwar ebenfalls, ob alle notwendigen Tabellen verfügbar sind, allerdings sind diese die Tabellen *Job*, *Task*, *TaskSet* und *CorrectTaskSet*. Bei den ersten drei Tabellen wird wie im *traditional-SA*-Projekt ein Ausnahme-Fehler erstellt, falls eine der Tabellen fehlt. Die letzte Tabelle wird mit Hilfe des Datenbank-Filters aus Kapitel 7.4 erstellt, wofür die Tabelle *ExecutionTime* unbedingt vorhanden sein muss. Aus diesem Grund wird die Existenz der *ExecutionTime*-Tabelle nur überprüft und bei Fehlen durch das Benchmark-Modul erstellt, wenn die Filterung der Datenbank erforderlich ist, das bedeutet wenn die Tabelle *CorrectTaskSet* fehlt.

### Tabellen lesen und schreiben

Das *RNN-SA*-Projekt benötigt die Tabelle *CorrectTaskSet*. Daher sind in der Datenbankschnittstelle des *RNN-SA*-Projektes jeweils zusätzlich eine Methode zum Lesen und Schreiben der Tabelle *CorrectTaskSet* implementiert.

Mit der Methode `read_table_correcttaskset()` wird die komplette Tabelle *CorrectTaskSet* gelesen und die Task-Set-Attribute werden als Liste aus Tupeln zurückgegeben.

Die Methode `write_correct_taskset()` schreibt ein übergebenes Task-Set, das ein Objekt der Klasse *Taskset* sein muss, in die Tabelle *CorrectTaskSet*. Dafür wird die entsprechende Tabelle zuerst erstellt, falls diese in der Datenbank noch nicht vorhanden ist. Abhängig von der Anzahl der Tasks werden die Attribute des Task-Sets, das bedeutet die Set-ID, das Label und die Task-IDs, in die Datenbank geschrieben. Die Unterscheidung zwischen den Task-Anzahlen ist notwendig, da ein *Taskset*-Objekt nur gültige Tasks enthält. In die Tabelle müssen aber auch die IDs von nicht vorhandenen Tasks mit dem Wert  $-1$  belegt werden.

## 7.3 Benchmark

Das Benchmark-Modul des *RNN-SA*-Projektes ist identisch zu dem im *traditional-SA*-Projekt, das bereits im Kapitel 6.4 beschrieben ist.

## 7.4 Datenbank-Filter

Der Datenbank-Filter sortiert alle Task-Sets aus der Datenbank aus, deren Ergebnis nicht mit dem einer exakten Schedulability-Analyse methode übereinstimmt. Dafür liest die Komponente alle Task-Sets aus der Tabelle *TaskSet*, testet sie mit einem exakten Verfahren und speichert die Task-Sets mit übereinstimmenden Ergebnissen in der Tabelle *CorrectTaskSet*. Diese Tabelle wird schließlich zum Trainieren eines neuronalen Netzes verwendet.

Der Datenbank-Filter ist in der Datei `database_filter.py` durch die Methode `filter_database()` implementiert. Als Eingabe erhält die Funktion ein Objekt der Klasse `Database`, um Zugriff auf die Datenbank zu haben. Der Quellcode der Methode ist in Listing 7.2 dargestellt.

Listing 7.2: Implementierung des Datenbank-Filters

```
def filter_database(database):  
    dataset = database.read_table_taskset() # Tabelle TaskSet lesen  
  
    # Task-Sets mit Hilfe der Antwortzeit-Analyse überprüfen  
    for taskset in dataset: # Iteration über alle Task-Sets  
        schedulability = rta.rta_buttazzo(taskset) # Planbarkeit überprüfen  
        real_result = taskset.result # reales Ergebnis des Task-Sets  
  
        # Testergebnis mit realem Ergebnis vergleichen  
        if schedulability is True and real_result == 1: # tp  
            # Task-Set in Tabelle CorrectTaskSet schreiben  
            database.write_correct_taskset(taskset)  
        elif schedulability is True and real_result == 0: # fp  
            pass  
        elif schedulability is False and real_result == 1: # fn  
            pass  
        elif schedulability is False and real_result == 0: # tn  
            # Task-Set in Tabelle CorrectTaskSet schreiben  
            database.write_correct_taskset(taskset)
```

---

Zunächst liest die Methode alle Task-Sets aus der Tabelle *TaskSet*. Dann wird in einer Schleife für jedes Task-Set die Planbarkeit durch ein exaktes Schedulability-Analyseverfahren bestimmt.

Aus denen im *traditional-SA*-Projekt implementierten Verfahren sind die Simulation, die Antwortzeit-Analyse und der  $\delta$ -HET exakt. Da die Simulation deutlich mehr Zeit in Anspruch nimmt als die Antwortzeit-Analyse oder der  $\delta$ -HET und dabei keine besseren Ergebnisse liefert, wird die Simulation ausgeschlossen. Die Ergebnisse der Antwortzeit-Analyse haben eine höhere Genauigkeit als die des  $\delta$ -HET, weshalb die Antwortzeit-Analyse als exakte Methode für den Datenbank-Filter ausgewählt wird. Ob als Startwert der Wert nach [Aud+93] oder nach [But11] genommen wird, macht sich nicht in den Ergebnissen bemerkbar. Lediglich die Dauer zur Berechnung unterscheidet sich minimal, weshalb die Antwortzeit-Analyse nach [But11] zur Bestimmung der Laufbarkeit eines Task-Sets verwendet wird.

Das durch die exakte Methode erhaltene Ergebnis über die Lauffähigkeit wird mit dem in der Datenbank gespeichertem Ergebnis bei der Ausführung auf dem PandaBoard ES Cluster verglichen. Nur wenn die Werte übereinstimmen, wird das Task-Set in die Tabelle *CorrectTaskSet* geschrieben. Der Aufbau der Tabelle ist identisch zur Tabelle *TaskSet* (siehe Tabelle 5.1).

### 7.5 Logging

Das Logging-Modul ist beinahe identisch zu dem des *traditional-SA*-Projektes. Allerdings werden nur Fehlermeldungen in der Datei `error.log` im Verzeichnis der Datenbank gespeichert und allgemeine Informationen auf der Kommandozeile ausgegeben. Anders als im *traditional-SA*-Projekt werden aktuell keine Ergebnisse gespeichert, sondern nur über die Kommandozeile ausgegeben.

### 7.6 Machine Learning Frameworks

Für die Implementierung von Machine Learning Modellen und neuronalen Netzen stehen eine Vielzahl von Frameworks zur Verfügung, die die Entwicklung abstrahieren und vereinfachen. Für den Aufbau und das Training des rekurrenten neuronalen Netzes in dieser Arbeit sind verschiedene Frameworks integriert, die nachfolgend vorgestellt werden.

#### TensorFlow

TensorFlow wird erstmalig im Jahr 2015 vom Projektteam „Google Brain“ vorgestellt und ist ein Framework für numerische Berechnungen [Sca18; HRL18]. TensorFlow unterliegt der Apache-2.0-Lizenz und findet Anwendung bei der Entwicklung und Implementierung von maschinellen Lernalgorithmen, insbesondere im Bereich des Deep Learnings und der künstlichen neuronalen Netze. Die Berechnungen werden als Datenflussgraphen gespeichert und sind in unterschiedlichen Umgebungen und auf vielerlei Hardwareplattformen ausführbar. Die Kernfunktionen von TensorFlow sind in C++ geschrieben, aber die primäre Schnittstelle ist Python. TensorFlow unterstützt verteiltes Lernen, das heißt der Lernprozess ist auf Rechnerclustern ausführbar, die aus CPUs oder GPUs bestehen. Zusätzlich sind die Berechnungsgraphen auch in der Cloud, zum Beispiel der Google Cloud Platform, einsetzbar. TensorFlow integriert auch ein eigenes Visualisierungswerkzeug: TensorBoard arbeitet mit Webbrowsern und ist zum Beobachten des Anlernprozesses, Finden von Fehlern, Veranschaulichen von Abläufen und Vereinheitlichen von Experimenten verwendbar.



## Keras

Das Framework Keras ist eine Deep Learning Abstraktionsbibliothek mit dem Ziel, schnelles Experimentieren mit neuronalen Netzen zu ermöglichen [Cho+15; HRL18]. Keras ist eine Schnittstelle zu anderen Frameworks für maschinelles Lernen wie TensorFlow, CNTK oder Theano [Cho+15; Sca18]. Statt auf Leistungsfähigkeit und Funktionsumfang baut Keras auf die Benutzerfreundlichkeit, Modularität, einfache Erweiterbarkeit und die Kompatibilität mit Python. Keras unterstützt konvolutionale und rekurrente neuronale Netze sowie Kombinationen von beiden und läuft nahtlos auf CPU und GPU. Veröffentlicht ist das Framework unter der MIT-Lizenz und ist kompatibel mit den Python-Versionen 2.7 bis 3.6. Auf der Homepage von Keras<sup>3</sup> sind viele Beispiele und Tutorials für verschiedene Anwendungsfälle vorhanden, die den Einstieg in den Umgang mit der Bibliothek erleichtern. Um Keras zu installieren, muss zuvor ein Backend installiert werden, kurzum TensorFlow, Theano oder CNTK [Cho+15]. TensorFlow wird dabei von Keras empfohlen und ist deshalb als Standardeinstellung in Keras hinterlegt.

## Talos

Talos ist ein unter der MIT-Lizenz verfügbares Framework zur vollautomatischen Hyperparameter-Anpassung und Modell-Evaluation [Aut19b]. Das Framework ist innerhalb von Minuten, ohne neue Syntax lernen zu müssen, in ein bestehendes Projekt integrierbar, da es mit jedem Keras-Modell funktioniert. Bei voller Kontrolle über die Keras-Modelle erlaubt Talos es, Experimente für die Hyperparameter-Anpassung zu konfigurieren, durchzuführen und zu evaluieren. Talos beinhaltet Zufallssuche, Raster-suche und wahrscheinlichkeitstheoretische Strategien. Es werden die Betriebssysteme Linux, Mac OS X und Windows unterstützt sowie die Python-Versionen 2 und 3. Außerdem ist die Ausführung auf CPU-, GPU- oder Multi-GPU-Systemen möglich. Talos folgt dem sogenannten POD-Workflow:

- P – Vorbereiten (engl. *prepare*): Festlegen des Hyperparameter-Raums für das Experiment und Einstellen der Experiment-Optionen, wie zum Beispiel der Optimierungsstrategie.
- O – Optimieren (engl. *optimize*): automatisches Finden der optimalen Hyperparameter-Kombination für ein gut verallgemeinerndes Modell.
- D – Einsetzen (engl. *deploy*): das Modell und alle erforderlichen Einstellungen werden gespeichert, um das Vorhersagemodell in der vorgesehenen Umgebung einzusetzen.

Zusätzlich sind Funktionen zur Evaluation und Auswertung mit graphischer Visualisierung verfügbar.

---

<sup>3</sup><https://keras.io/>

## 7.7 Machine Learning Modelle

Das Modell des rekurrenten neuronalen Netzes wird mit dem Framework Keras erstellt und ist in der Datei `ml_models.py` gespeichert. Der prinzipielle Aufbau des rekurrenten neuronalen Netzes ist in Abbildung 7.4 dargestellt.

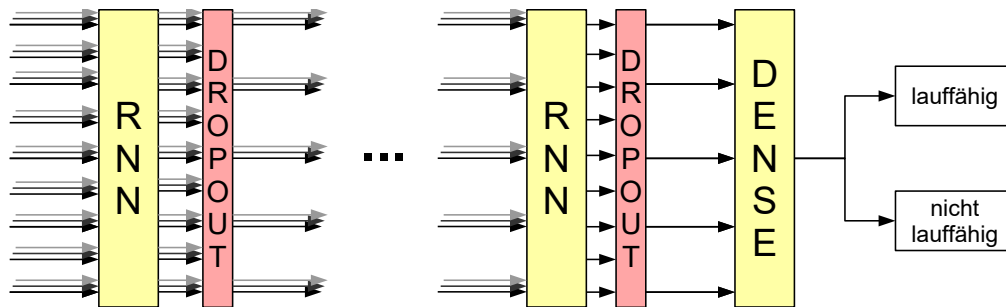


Abbildung 7.4: Aufbau des Keras-RNN-Modells (Eigene Darstellung).

Das Netz besteht aus mehreren rekurrenten Neuronenschichten. Diese Schichten können auch LSTM- oder GRU-Zellen sein. Die Eingaben sind Sequenzen von Features, das heißt die Attribute der Tasks, die nacheinander dem Task-Set hinzugefügt werden. Die Sequenzen werden von den rekurrenten Schichten verarbeitet und als Sequenzen an die nächste Schicht weitergegeben. Die letzte rekurrente Schicht gibt nicht die gesamte Sequenz zurück, sondern nur den letzten Zeitschritt, da die Lauffähigkeit des gesamten Task-Sets bestimmt wird und nicht nach jedem Task. Die Werte des letzten Zeitschritts werden von einer vollständig verbundenen Schicht (engl. *dense layer*) in eine einzelne Ausgabe umgewandelt, die das Task-Set der Klasse „lauffähig“ (Wert 1) oder „nicht lauffähig“ (Wert 0) zuordnet.

Um das Modell mit dem Framework Talos kompatibel zu machen, wird keine Klasse definiert, sondern eine Funktion, die das Keras-Modell erstellt, trainiert und das Modell sowie die Ergebnisse des Trainings zurückgibt. Als Eingabe erhält die Funktion den Trainings- und Evaluationsdatensatz sowie ein Hyperparameter-Dictionary. Der Quellcode der Funktion für das Modell aus LSTM-Zellen ist in Listing 7.3 gezeigt.

Aus Zeitgründen wird nur ein Modell, nämlich ein RNN aus LSTM-Zellen, erstellt. Deshalb wird die Entwicklung eines Keras-Modells nachfolgend exemplarisch anhand dieses RNNs erläutert. Mit den gleichen Arbeitsschritten wird auch ein RNN aus rekurrenten Neuronenschichten oder GRU-Zellen erstellt, indem an den entsprechenden Stellen die LSTM-Zellen ersetzt werden.

### 7.7.1 Erstellung eines Keras-Modells

Die Kern-Datenstruktur von Keras ist das Modell, ein Weg um Neuronenschichten zu organisieren [Cho+15]. Das LSTM-Modell ist in der Methode `_build_lstm_model()` definiert. Die einfachste Form eines Modells ist das `Sequential`-Modell, das die Schichten

Listing 7.3: Funktion zur Beschreibung des LSTM-Modells

```
def LSTM_model(x_train, y_train, x_val, y_val, hparams):
    from params import config # Konfigurationsparameter importieren

    model = _build_LSTM_model(hparams, config) # Keras-Modell erstellen
    model.compile(...) # Modell für Training konfigurieren
    out = model.fit(...) # Modell trainieren
    return out, model
```

linear aufeinander stapelt. Ein solches Modell wird durch

```
model = keras.models.Sequential()
```

erstellt. Mit der `add()`-Methode werden dem Modell verschiedene Schichten hinzugefügt. Eine LSTM-Schicht wird durch

```
model.add(keras.layers.LSTM(
    units=hparams['hidden_layer_size'],
    activation=hparams['hidden_activation'],
    return_sequences=False,
    input_shape=(config['time_steps'], config['element_size'])))
```

eingefügt, wobei mit `unit` die Anzahl der Neuronen und mit `activation` die verwendete Aktivierungsfunktion bestimmt wird. Werden mehrere LSTM-Schichten aufeinander gestapelt, werden zwischen den Schichten die kompletten Sequenzen übertragen, das heißt `return_sequences=True` ist gesetzt. Die letzte LSTM-Schicht in einem neuronalen Netz gibt nur das letzte Ergebnis, kurzum den letzten Zeitschritt, zurück (`return_sequences=False`). Mit `input_shape` wird für die erste LSTM-Schicht in einem Netzwerk die Gestalt der Eingabedaten festgelegt. Alle weiteren LSTM-Schichten müssen diesen Parameter aber nicht spezifizieren.

Das GRU-Modell sowie das RNN-Modell werden auf die gleiche Art und Weise erstellt, indem anstatt der LSTM-Schichten GRU- bzw. SimpleRNN-Schichten verwendet werden.

Mit Dropout-Schichten wird die Wahrscheinlichkeit einer Überanpassung reduziert, da sich die Abhängigkeiten zwischen den Eingaben der Dropout-Schichten verringern [Sca18]. Eine Dropout-Schicht wendet Dropout auf die Eingabe an, das bedeutet es werden zufällig Signale ausgeschaltet und nicht mehr weitergegeben. Wie viele der Eingangssignale nicht weitergegeben werden, bestimmt der Parameter `rate`:

```
model.add(keras.layers.Dropout(rate=1-hparams['keep_prob'])).
```

Durch mehrfaches Hinzufügen der beschriebenen Schichten zu dem Sequential-Modell wird ein rekurrentes neuronales Netz wie in Abbildung 7.4 aufgebaut.

Die letzte Schicht eines rekurrenten neuronalen Netzes ist die Ausgabeschicht, eine vollständig verbundene Neuronenschicht, die folgendermaßen hinzugefügt wird:

```
model.add(keras.layers.Dense(  
    units=config['num_classes'],  
    activation='sigmoid')).
```

Der Parameter `units` bestimmt die Anzahl der Ausgabeeinheiten. Da für jedes Task-Set nur ein Wert als Label gespeichert wird (0 für „lauffähig“, 1 für „nicht lauffähig“), ist nur eine einzelne Ausgabeeinheit notwendig. Ist das Ergebnis eines Task-Sets wie das PKG-Attribut durch eine One-Hot-Kodierung kodiert (z.B. 10 für „lauffähig“, 01 für „nicht lauffähig“), sind zwei Ausgabeeinheiten notwendig. Die Aktivierungsfunktion muss für die binäre Klassifizierung zwingend die sigmoide Funktion sein, wie durch Gleichung 3.8 definiert, um die Einteilung in die zwei Klassen zu ermöglichen.

### 7.7.2 Training eines Keras-Modells

Wenn das Modell erstellt ist, muss der Lernprozess durch die `compile()`-Methode konfiguriert werden:

```
model.compile(  
    optimizer=hparams['optimizer'],  
    loss='binary_crossentropy',  
    metrics=['accuracy']).
```

Das Attribut `optimizer` bestimmt den Optimierungsalgorithmus, das heißt wie die Gewichte des neuronalen Netzes in jedem Schritt angepasst werden. Häufig verwendete Funktionen sind der SGD (engl. *stochastic gradient descent*), RMSProp, Adagrad, Adadelta, Adam, Adamax und Nesterov Adam Optimierer, die alle in Keras enthalten sind. Optional sind zusätzlich noch die Eigenschaften des Optimierers, wie zum Beispiel die Lernrate, einstellbar. Da in der aktuellen Version des Projekts die Lernrate automatisch angepasst wird, ist eine weitere Konfiguration des Optimierungsalgorithmus nicht notwendig.

Die Verlustfunktion wird durch `loss` bestimmt. Für die binäre Klassifizierung muss hier unbedingt die binäre Kreuzentropie verwendet werden:

$$loss = -(y \log(p) + (1 - y) \log(1 - p)). \quad (7.3)$$

Die Variable `y` steht dabei für die Klassenbezeichnung (Label) und `p` für den vom Modell vorhergesagten Wert.

Durch `metrics` werden Maßfunktionen festgelegt, die die Leistung des Modells bewerten. Sehr häufig wird die Korrektklassifikationsrate (engl. *accuracy*) verwendet, die bereits in Kapitel 6.11 durch Gleichung 6.15 definiert ist.

Das Training eines Keras-Modells wird von der Methode `fit()` durchgeführt:

```
out = model.fit(  
    x=x_train,  
    y=y_train,  
    batch_size=params['batch_size'],  
    epochs=params['num_epochs'],  
    verbose=config['verbose_training'],  
    callbacks=callbacks,  
    validation_data=[x_val, y_val],  
    shuffle=True).
```

Die Trainingsdaten, welche aus Task-Attributen und Labels bestehen, werden durch die Parameter `x` und `y` übergeben. Während des Trainings werden die Trainingsdaten in Batches, das bedeutet in Teildatensätze, aufgeteilt und die Gewichte der Neuronen-Verbindungen dafür angepasst. Die Größe dieser Teildatensätze wird mit `batch_size` festgelegt. Durch `epochs` wird die Anzahl der Epochen, das heißt wie oft der gesamte Datensatz analysiert wird, bestimmt. Nach jeder Epoche wird mit Hilfe der Evaluationsdaten (`validation_data`) die aktuelle Leistung des Modells bewertet. So kann schon während des Trainings die Leistung des Modells beobachtet werden. Falls keine Verbesserung erkennbar ist, kann das Training abgebrochen werden. Wie viel Information über die Kommandozeile ausgegeben wird, bestimmt der Parameter `verbose`. Es werden entweder gar keine Informationen angezeigt (`verbose=0`), eine detaillierte Fortschrittsanzeige (`verbose=1`) oder jeweils eine Zeile mit den Evaluationsergebnissen pro Epoche (`verbose=2`). Mit `callbacks` werden dem Training Callbacks hinzugefügt. Das sind Funktionen die beim Anlernen aufgerufen werden, um einen Einblick in die Statistiken zu bekommen und Entscheidungen zum Anlernprozess dynamisch zu treffen. Aktuell werden durch die `_init_callbacks()`-Methode folgende Callbacks erstellt und hinzugefügt:

- `ModelCheckpoint`: speichert das Modell nach jeder Epoche.
- `EarlyStopping`: stoppt das Training, wenn die überwachte Maßfunktion keine Verbesserung anzeigt.
- `TensorBoard`: Visualisierung mit TensorBoard.
- `ReduceLROnPlateau`: reduziert die Lernrate, wenn sich die Maßfunktion nicht mehr verbessert.

Der Parameter `shuffle` bestimmt, ob die Trainingsdaten nach jeder Epoche durcheinander gemischt werden. Die Rückgabe der `fit()`-Methode ist ein `History`-Objekt, das eine Aufzeichnung der Verlustwerte und Korrektklassifikationsraten für das Training und die Evaluation von jeder Epoche speichert.

### 7.7.3 Evaluation eines Keras-Modells

Die Evaluation eines Keras-Modells ist im Hauptprogramm in der Methode `train_and_evaluate()` definiert. Das Modell wird durch die `LSTM_model()`-Funktion erstellt und trainiert und folgendermaßen evaluiert:

```
loss, accuracy = model.evaluate(data['text_X'], data['text_y'], batch_size
                                =params.hparams['batch_size'], verbose=params.config['verbose_eval']).
```

Es wird der Testdatensatz und die Größe der Teildatensätze übergeben. Mit `verbose` wird wie schon beim Training eingestellt, wie detailreich die Ausgabe auf der Konsole ist. Die Rückgabe der Evaluations-Funktion sind zwei Werte: `loss` ist der Wert der Verlustfunktion und `accuracy` die Korrektklassifikationsrate des Testdatensatzes.

## 7.8 Hyperparameter-Optimierung

Die Standardparameter der Machine Learning Frameworks sind selten für eine spezielle Aufgabenstellung optimiert [RM17]. So ist zum Beispiel die Größe der neuronalen Schichten, das heißt die Anzahl der Neuronen, unter anderem von der Dimension der Eingabedaten abhängig und muss für jeden Anwendungsfall individuell angepasst werden. Die Einstellungsmöglichkeiten eines neuronalen Netzes und dessen Trainings werden Hyperparameter genannt und bestimmen maßgeblich die Leistung des Netzes. Insofern ist das Finden der optimalen Hyperparameter, auch Hyperparameter-Optimierung genannt, eine wichtige, aber auch schwierige Aufgabe in der Entwicklung eines Machine Learning Modells.

Oft enthält ein Machine Learning Projekt mehrere Hyperparameter, die häufig auch noch voneinander abhängig sind [NZ18]. Zudem gibt es keine allgemein gültigen Regeln, wie die einzelnen Hyperparameter zu bestimmen sind. Daher ist es nicht trivial, einen optimalen Satz von Hyperparametern zu finden. Der simpelste Weg ist das Ausprobieren verschiedener Konfigurationen mit anschließender Evaluation der Leistung des Modells. Dabei sind zwei häufig angewendete Methoden die Grid- und die Zufallssuche. Bei der Grid-Suche werden mehrere mögliche Werte für verschiedene Hyperparameter festgelegt und für jede Kombination wird ein Modell trainiert. Bei der Zufallssuche werden die Parameter vollkommen zufällig ausgewählt. Als Maß für die Leistung eines Modells wird häufig die Korrektklassifikationsrate (engl. *accuracy*) des Evaluationsdatensatzes verwendet.

Im RNN-SA-Projekt wird für die Hyperparameter-Optimierung das Framework Talos integriert. Mit diesem ist es ohne großen Aufwand möglich, Experimente mit verschiedenen Hyperparametern durchzuführen und die Ergebnisse zu evaluieren. Talos trainiert dabei automatisch mehrere Keras-Modelle mit den unterschiedlichen Hyperparameter-Kombinationen. Um Talos in ein bestehendes Projekt zu integrieren bzw. ein Hyperparameter-Experiment zu starten, sind drei Dinge notwendig [Aut19a]: ein Hyperparameter-Dictionary, ein funktionierendes Keras-Modell und ein Talos Experiment.

## Hyperparameter-Dictionary

Der erste Schritt in jedem Experiment ist das Festlegen der Hyperparameter die optimiert bzw. untersucht werden, den sogenannten Hyperparameter-Raum. Hierfür wird von Talos ein Hyperparameter-Dictionary, ein reguläres Python-Dictionary, benötigt, das die Hyperparameter und deren Grenzen für das Experiment definiert. Das Hyperparameter-Dictionary des RNN-SA-Projektes für Talos ist in der Datei `params.py` gespeichert und sieht beispielsweise wie in Listing 7.4 aus.

Listing 7.4: Hyperparameter-Dictionary für die Verwendung mit Talos

```
hparams_talos = {
    'batch_size': [32, 64, 128, 256, 512, 1024],
    'num_epochs': [150],
    'keep_prob': [1.0],
    'num_cells': [1],
    'hidden_layer_size': (27, 1000, 50),
    'hidden_activation': [keras.activations.tanh],
    'optimizer': [keras.optimizers.Adam],
    'learning_rate': [0.0001]
}
```

Die Parameter können auf drei verschiedene Arten definiert werden:

- als Liste mit diskreten Werten, wie zum Beispiel die Größe der Teildatensätze (`batch_size`),
- als Wertebereich in Form eines Tupels (Minimum, Maximum, Schritte), wie zum Beispiel die Größe der Neuronenschichten (`hidden_layer_size`) oder
- als Liste mit einem einzelnen diskreten Wert, wie zum Beispiel die Anzahl der Epochen (`num_epochs`).

Grundsätzlich werden in einem Talos-Experiment alle Parameter, die zur Einstellung eines Keras-Modells zur Verfügung stehen, berücksichtigt, sofern sie in dem Hyperparameter-Dictionary vorhanden sind.

Um ein Keras-Modell auch ohne Talos zu trainieren, wird ein weiteres Dictionary `hparams` angelegt. Die darin enthaltenen Hyperparameter sind identisch zu denen aus `hparams_talos`, speichern aber jeweils nur einen diskreten Wert. Dies ist notwendig, da Keras keine Listen als Eingabe akzeptiert.

Um auch andere Einstellungen, die nicht die Leistung des neuronalen Netzes beeinflussen, an einem zentralen Ort zu speichern, wird ein drittes Dictionary erstellt:

das Konfigurationsparameter-Dictionary (config) wie in Listing 7.5 gezeigt. Diese Konfigurationsparameter speichern zum Beispiel, welche Callbacks verwendet werden, wie viele Informationen auf der Konsole ausgegeben werden und die Dimension der Eingabe-Daten.

Listing 7.5: Konfigurationsparameter-Dictionary für allgemeine Einstellungen

```
config = {
    'use_checkpoint': False,
    'checkpoint_dir':
        os.path.join(os.getcwd(), "experiments", "LSTM", "checkpoints"),
    'checkpoint_verbose': 1,
    'use_earlystopping': True,
    'use_tensorboard': False,
    'tensorboard_log_dir':
        os.path.join(os.getcwd(), "experiments", "LSTM", "logs"),
    'use_reduceLR': True,
    'verbose_training': 2,
    'verbose_eval': 0,
    'time_steps': 4,
    'element_size': 12,
    'num_classes': 1
}
```

---

Tabelle 7.2 gibt einen Überblick über die aktuell integrierten Konfigurations- und Hyperparameter sowie deren Bedeutung.

### Keras-Modell

Das Ziel von Talos ist eine möglichst einfache Integration des Frameworks ohne großen Aufwand. Deshalb ist jedes Keras-Modell in einem Talos-Experiment verwendbar. Es müssen lediglich die Hyperparameter durch entsprechende Referenzen auf die Dictionary-Einträge ersetzt werden.

Das Keras-Modell für das rekurrente neuronale Netz aus LSTM-Zellen wird bereits im vorherigen Kapitel 7.7 ausführlich beschrieben. Das Modell steht über die Funktion `LSTM_model()` zur Verfügung und die Hyperparameter werden darin durch die entsprechenden Referenzen ersetzt.



Tabelle 7.2: Beschreibung der aktuell integrierten Konfigurations- und Hyperparameter (Eigene Darstellung).

Parameter	Bedeutung
batch_size	Größe der Teildatensätze, das heißt die Anzahl an Task-Sets, für die ein Update der Gewichte durchgeführt wird.
num_epochs	Anzahl der Trainingsepochen, in denen der gesamte Datensatz verarbeitet wird.
keep_prob	Anteil der Eingaben, die weitergeleitet werden; $(1 - \text{keep\_prob})$ ist entsprechend der Anteil an Eingaben, der durch eine Dropout-Schicht ausgeschaltet wird.
num_cells	Anzahl der Zellen (LSTM, GRU) bzw. der rekurrenten Schichten.
hidden_layer_size	Größe der Neuronenschichten, entspricht der Anzahl der Neuronen je Schicht.
hidden_activation	Aktivierungsfunktion der Neuronenschichten.
optimizer	Optimierungsalgorithmus, mit dem die Gewichte angepasst werden.
use_checkpoint	Ob der ModelCheckpoint-Callback verwendet wird (True) oder nicht (False).
checkpoint_dir	Verzeichnis, in dem das Modell gespeichert wird.
checkpoint_verbose	Ausführlichkeit der ausgegebenen Informationen beim Speichern des Modells.
use_earlystopping	Ob der EarlyStopping-Callback verwendet wird (True) oder nicht (False).
use_tensorboard	Ob der TensorBoard-Callback verwendet wird (True) oder nicht (False).
tensorboard_log_dir	Verzeichnis, in dem die Log-Dateien für TensorBoard gespeichert werden.
use_reduceLR	Ob der ReduceLRonPlateau-Callback verwendet wird (True) oder nicht (False).

Fortsetzung auf der nächsten Seite

Tabelle 7.2. (Fortsetzung).

Parameter	Bedeutung
verbose_training	Ausführlichkeit der ausgegebenen Informationen während des Trainings.
verbose_eval	Ausführlichkeit der ausgegebenen Informationen während des Tests.
time_steps	Anzahl der Zeitschritte = Länge der Sequenz = maximale Anzahl an Tasks je Task-Set.
element_size	Länge jedes Vektors der Sequenz = Anzahl der Attribute je Task.
num_classes	Anzahl der Klassen; Anzahl der Bits mit denen die Klassenbezeichnung kodiert ist.

### Talos-Experiment

Stehen ein funktionsfähiges Keras-Modell und ein Hyperparameter-Dictionary zur Verfügung, wird ein Talos-Experiment mit der Methode `Scan()` wie in Listing 7.6 gezeigt konfiguriert und gestartet.

#### Listing 7.6: Durchführen eines Talos-Experiments

```

talos.Scan(
    x=data['train_X'], # Trainingsfeatures
    y=data['train_y'], # Trainingslabels
    params=params.hparams_talos, # Hyperparameter
    model=ml_models.LSTM_model, # Keras-Modell
    dataset_name=name, # Name des Experiments
    experiment_no=num, # Nummer des Experiments
    x_val=data['val_X'], # Evaluationsfeatures
    y_val=data['val_y'], # Evaluationslabels
    grid_downsample=0.1, # Downsampling der Kombinationsmöglichkeiten
    print_params=True # Ausgabe der Hyperparameter
)

```

Die Trainings- und Evaluationsdatensätze werden durch die Parameter `x`, `y`, `x_val` und `y_val` übergeben. Das Hyperparameter-Dictionary wird durch den Parameter `params` und das Keras-Modell durch `model` festgelegt. Die Ergebnisse des Experiments werden in einer CSV-Datei gespeichert, deren Namen mit den Übergabewerten `dataset_name`

für den Namen und `experiment_no` für die Nummer des Experiments festgelegt werden. Das Attribut `grid_downsample` bestimmt, wie viele aller Kombinationsmöglichkeiten aus dem Hyperparameter-Raum tatsächlich untersucht werden. Ist dieser Wert zum Beispiel 0,1, werden zufällig 10 % aller Kombinationen ausgewählt und für diese Möglichkeiten jeweils ein Modell trainiert. Die Hyperparameter der jeweiligen Iteration werden auf der Konsole ausgegeben, wenn `print_params=True` gesetzt ist.

## 7.9 Visualisierung der Ergebnisse der Hyperparameter-Optimierung

Um die Ergebnisse der Experimente mit den Hyperparametern zu visualisieren, sind in der Datei `plotting.py` verschiedene Darstellungsmethoden implementiert. Aktuell werden folgende Grafiken erstellt:

- Darstellung der Korrektklassifikationsrate der Evaluation in Abhängigkeit eines einzelnen Hyperparameters („Single Line Plot“).
- Darstellung der Wahrheitsmatrix für den Test-Datensatz („Confusion Matrix“).
- Darstellung der Korrelationsmatrix für die Korrektklassifikationsrate der Evaluation in Abhängigkeit von allen Hyperparametern („Correlation Matrix“).

Die Ergebnisse der Talos-Experimente werden in einer CSV-Datei gespeichert, die mit dem Python-Code in Listing 7.7 geöffnet und gelesen wird. Als Beispiel werden in diesem Codeausschnitt die Spalten 2, das heißt die Korrektklassifikationsrate der Evaluation (*val\_acc*), und 6, das bedeutet die Größe der Teildatensätze (*batch\_size*), ausgelesen und in den beiden Listen `x` und `y` gespeichert.

Listing 7.7: Öffnen und Lesen einer CSV-Datei

```
x = [] # Daten der x-Achse (ein Hyperparameter)
y = [] # Daten der y-Achse (Korrektklassifikationsrate der Evaluation)

with open('experiment_1.csv', 'r') as csvfile: # CSV-Datei öffnen
    plots = csv.reader(csvfile, delimiter=',')
    header = next(plots) # Kopfzeile lesen
    for row in plots: # Iteration über alle Reihen
        x.append(int(row[6]))
        y.append(float(row[2]))
```

Um die Korrektklassifikationsrate in Abhängigkeit von einem Hyperparameter darzustellen, müssen diese beiden Spalten aus der CSV-Datei ausgelesen und dann durch den Quellcode in Listing 7.8 visualisiert werden.

### Listing 7.8: Darstellung einer Funktion $y = f(x)$

```
import matplotlib.pyplot as plt
plt.plot(x, y, 'o') # Liniendiagramm von  $y = f(x)$ 
plt.show() # alle Diagramme anzeigen
```

---

Die Darstellung der Wahrheitsmatrix übernimmt die Funktion `get_confusion_matrix()`, die in Listing 7.9 aufgelistet ist. Zunächst wird ein Modell und die Daten geladen. Anschließend werden für den Testdatensatz Vorhersagen durch das Modell getroffen. Mit Hilfe der Sklearn-Bibliothek werden die getroffenen Vorhersagen des Modells mit denen in der Datenbank verglichen. Die Rückgabewerte sind die Werte der Wahrheitsmatrix, demnach die Anzahl der richtig positiven (tp), falsch positiven (fp), richtig negativen (tn) und falsch negativen (fn) Vorhersagen.

### Listing 7.9: Bestimmung der Wahrheitsmatrix

```
def get_confusion_matrix():
    model = keras.models.load_model(os.path.join(config['checkpoint_dir',
        ], "weights.best.hdf5")) # Gewichte laden

    data = main.load_data(os.getcwd(), "panda_v3.db") # Daten laden

    # Vorhersagen generieren
    y_pred = model.predict_classes(data['test_X'])

    # Werte der Wahrheitsmatrix bestimmen
    tn, fp, fn, tp = sklearn.metrics.confusion_matrix(data['test_y'],
        y_pred).ravel()
```

---

Die Visualisierung der Korrelationsmatrix geschieht mit Hilfe des Frameworks Talos und ist in Listing 7.10 dargestellt. Durch die Reporting-Klasse stellt Talos verschiedene Befehle zur Verfügung, um die Ergebnisse eines Experiments zu analysieren. Als Eingabe muss lediglich eine CSV-Datei mit den Ergebnissen übergeben werden. Um die Korrelationsmatrix zu erzeugen, steht die Funktion `plot_corr()` zur Verfügung.

Weitere Ausführungen über die Visualisierung von Daten gehen über den Rahmen dieser Arbeit hinaus. An dieser Stelle sei für detailliertere Informationen sowie Beispiele und Tutorials auf die Dokumentationen von Talos<sup>4</sup> und Matplotlib<sup>5</sup> verwiesen.

---

<sup>4</sup>[https://autonomio.github.io/docs\\_talos/#reporting](https://autonomio.github.io/docs_talos/#reporting)

<sup>5</sup><https://matplotlib.org/index.html#>

## Listing 7.10: Visualisierung der Korrelationsmatrix

```
import talos
r = talos.Reporting('experiment_1.csv') # CSV-Datei öffnen und lesen
r.plot_corr() # Korrelationsmatrix visualisieren
plt.show() # alle Diagramme anzeigen
```

## 7.10 Evaluation

### 7.10.1 Anpassung der Hyperparameter

Da die Standardeinstellungen von Keras nicht für die hier gedachte Anwendung als Schedulability-Analyseverfahren optimiert sind, werden nacheinander verschiedene Werte für die Hyperparameter untersucht. Als Maß für die Leistung des RNNs dient dabei die Korrektklassifikationsrate des Evaluationsdatensatzes. Dabei gilt: je größer dieser Wert, desto besser die Leistung des neuronalen Netzes. Der Evaluationsdatensatz wird verwendet, da nur so die Verallgemeinerung des Modells abschätzbar ist und eine Über- bzw. Unteranpassung erkennbar wird, im Gegensatz zum Trainingsdatensatz.

#### Anzahl der Epochen

Die Anzahl der Epochen gibt an, wie oft während des Trainings über den gesamten Trainingsdatensatz iteriert wird. Es werden Werte zwischen 0 und 200 getestet.

Wird die Korrektklassifikationsrate in Abhängigkeit von der Anzahl der Epochen wie in Abbildung 7.5 dargestellt, so ergibt sich eine Sättigungskurve. Ab ungefähr 125 Epochen ändert sich die Korrektklassifikationsrate nur noch geringfügig und pendelt sich bei einem Wert von 93,2 % ein. Da das implementierte Keras-Modell den EarlyStopping-Callback integriert, der das Training bei ausbleibender Verbesserung der Korrektklassifikationsrate beendet, werden die folgenden Experimente mit einer maximalen Anzahl von 150 Epochen durchgeführt.

#### Größe der Teildatensätze

Die Größe der Teildatensätze (engl. *batch size*) bestimmt, wie viele Beispiele, hier Task-Sets, für die Anpassung der Gewichte des RNNs berücksichtigt werden. Übliche Werte liegen zwischen 32 und 512, wobei häufig Zweier-Potenzen gewählt werden [Kes+16]. Es werden deshalb mehrere Experimente mit Zweier-Potenzen sowie anderen zufälligen Werten zwischen 32 und 1024 durchgeführt. Das Ergebnis ist in Abbildung 7.6 zu sehen.

Zwischen der Korrektklassifikationsrate der Evaluation und der Größe der Teildatensätze besteht ein linearer Zusammenhang: je kleiner die Teildatensätze, desto besser die Leistung des neuronalen Netzes. Das beste Ergebnis von 95,75 % wird für Teildatensätze mit 128 Task-Sets erreicht. Der Vorteil von größeren Teildatensätzen ist ein deutlich

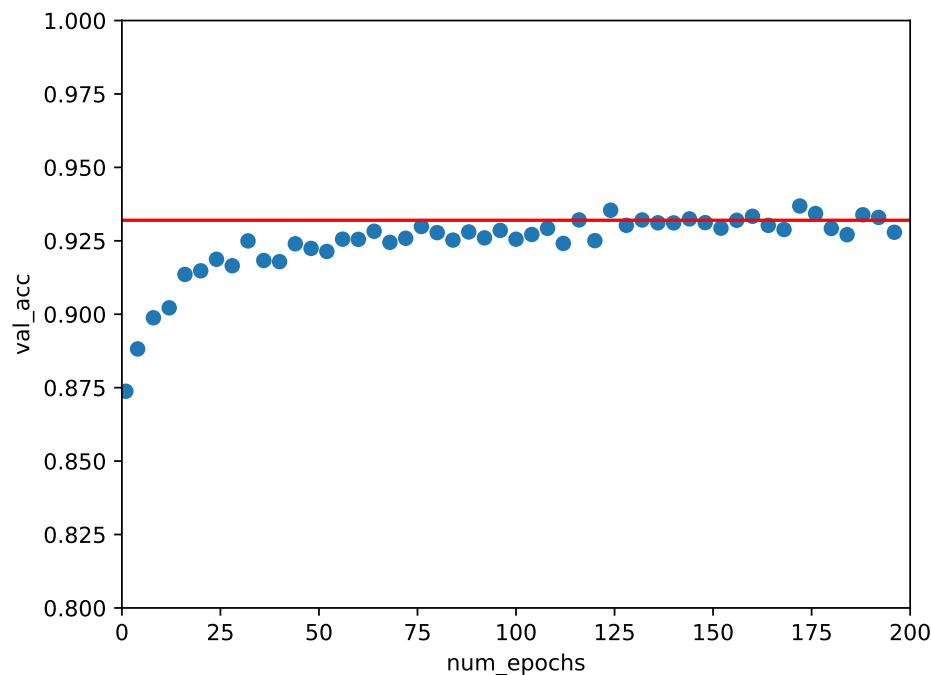


Abbildung 7.5: Hyperparameter-Optimierung: die Korrektklassifikationsrate in Abhängigkeit von der Anzahl der Epochen (Eigene Darstellung).

schnelleres Training. Allerdings ist die Verallgemeinerung von solchen Machine Learning Modellen im Allgemeinen signifikant schlechter, als bei Verwendung von kleineren Teildatensätzen. Als Kompromiss zwischen der Leistung und der Trainingsdauer wird für die folgenden Experimente die Größe der Teildatensätze auf 128 festgelegt.

Erst nach Abschluss der Experimente mit der Batch-Size wurde festgestellt, dass sich der Trainingsdatensatz bei jeder Ausführung des Hauptprogramms ändert. Daraufhin wurden alle zufälligen Programmschritte, wie zum Beispiel das Mischen der Task-Sets und die Aufteilung in verschiedenen Datensätze, reproduzierbar gemacht, um die Ergebnisse der Telexperimente vergleichbar zu machen. Das ist für die Experimente mit der Größe der Teildatensätze allerdings nicht gegeben, weshalb der Zusammenhang in Abbildung 7.6 nur als grober Überblick betrachtet werden darf, nicht aber als exaktes Ergebnis, da die Punkte aus unterschiedlichen Telexperimenten mit verschiedenen Trainingsdatensätzen stammen.

### Größe der versteckten Schichten

Die Größe der versteckten Schichten gibt an, aus wie vielen Neuronen die versteckten Schichten bzw. LSTM-Zellen bestehen. Die Anzahl der Neuronen ist dabei unter anderem von der Komplexität der Eingabedaten, das heißt der Anzahl an Features, abhängig.

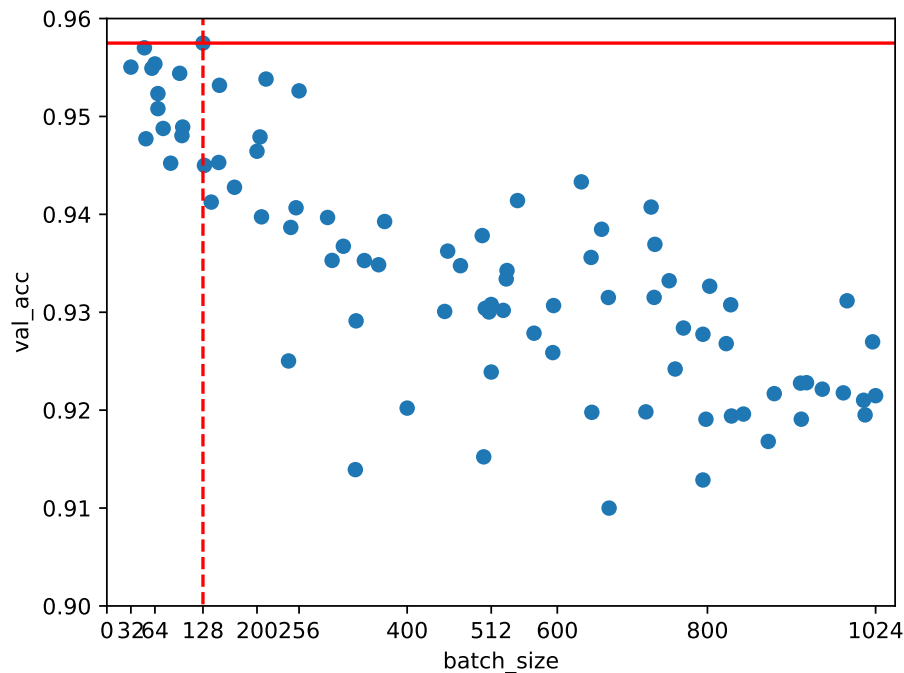


Abbildung 7.6: Hyperparameter-Optimierung: die Korrektklassifikationsrate in Abhängigkeit von der Größe der Teildatensätze (Eigene Darstellung).

Deshalb werden zunächst Vielfache der Anzahl an Task-Attributen getestet, wie 3, 9 und 27. Anschließend werden zufällige Werte bis zu 1000 untersucht. Um die so entstandene Sättigungskurve zu bestätigen, wird zuletzt ein Experiment mit den Werten 2000, 3000 und 4560 durchgeführt. Die Ergebnisse sind in Abbildung 7.7 dargestellt.

Wie bereits bei der Anzahl der Epochen resultiert auch die Korrektklassifikationsrate in Abhängigkeit von der Größe der versteckten Schichten in einer Sättigungskurve. Die Kurve steigt bis zu einem Wert von 100 Neuronen steil an und pendelt sich ab ungefähr 300 Neuronen bei einer Korrektklassifikationsrate von 97,70 % ein. Bei 319 Neuronen wird das erste Mal der Maximalwert von 97,77 % erreicht, weshalb dieser Wert für alle nachfolgenden Experimente verwendet wird.

### Anzahl der Zellen

Die Anzahl der Zellen bestimmt, wie viele rekurrente Schichten bzw. Zellen hintereinander gestapelt werden, möglicherweise mit Dropout-Schichten dazwischen. Versuchsweise werden Modelle mit einer und bis zu zehn LSTM-Zellen trainiert und die Korrektklassifikationsrate der Evaluation miteinander verglichen. Abbildung 7.8 zeigt das Ergebnis dieses Experiments.

Die maximale Korrektklassifikationsrate beträgt 98,47 % bei zehn LSTM-Zellen. Aber bereits bei drei Zellen wird ein annähernd gleiches Ergebnis von 98,29 % erreicht.

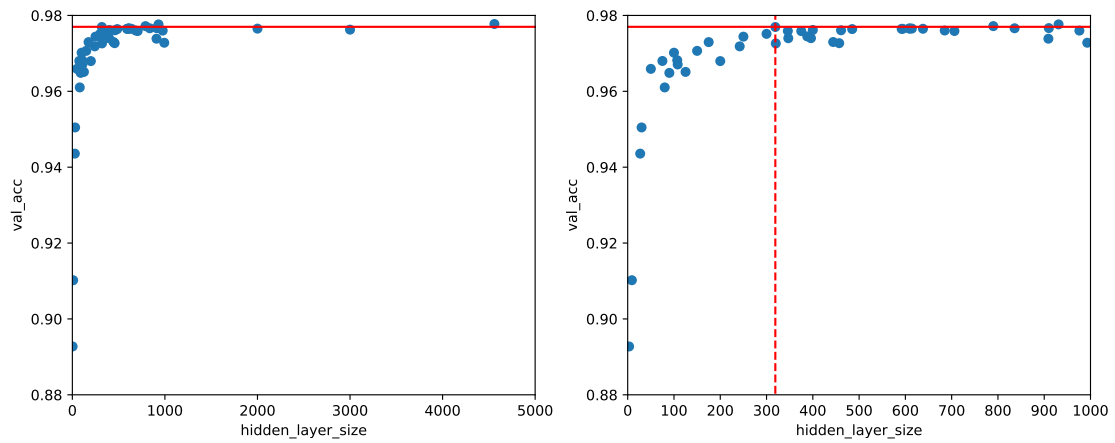


Abbildung 7.7: Hyperparameter-Optimierung: die Korrektklassifikationsrate in Abhängigkeit von der Größe der versteckten Schichten (Eigene Darstellung).

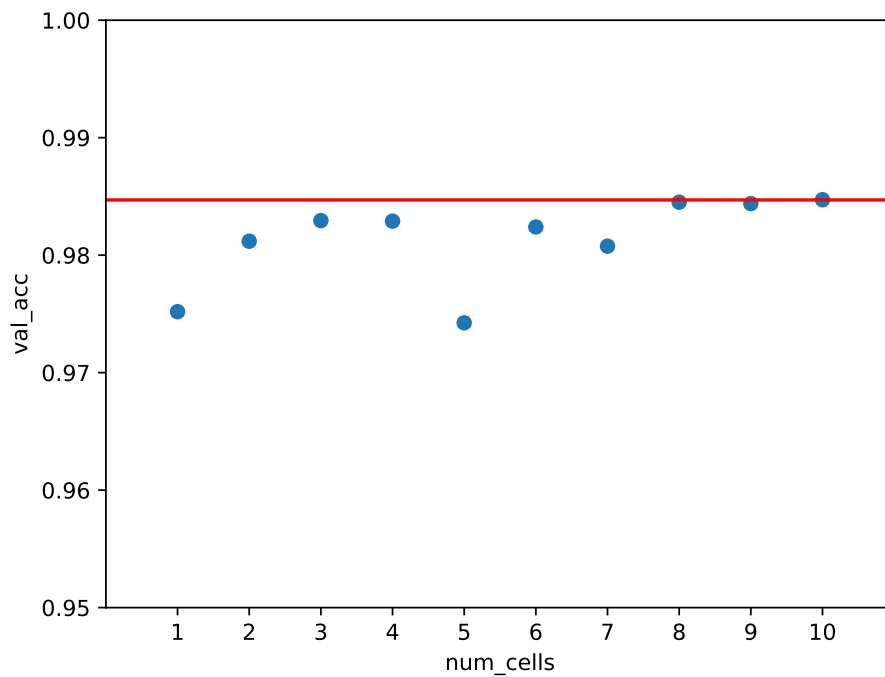


Abbildung 7.8: Hyperparameter-Optimierung: die Korrektklassifikationsrate in Abhängigkeit von der Anzahl der Zellen (Eigene Darstellung).



Überraschend ist das im Vergleich schlechte Abschneiden von fünf gestapelten LSTM-Zellen mit nur 97,29 %, das sogar noch unter dem Ergebnis von einer Zelle bei 97,52 % liegt. Dafür gibt es keine Erklärung. Die weiteren Experimente werden mit drei Zellen ausgeführt, da mit zunehmender Anzahl an Zellen die Trainingsdauer deutlich ansteigt.

## Dropout

Mit den Dropout-Schichten werden zufällig Signale zwischen den LSTM-Zellen ausgeschaltet, womit die Wahrscheinlichkeit für eine Überanpassung reduziert wird. Der Hyperparameter `keep_prob` bestimmt, wie viele Signale erhalten bleiben. Die prozentuale Anzahl der abgestellten Signale bestimmt sich zu  $(1 - \text{keep\_prob})$ . In zwei Telexperimenten werden alle Möglichkeiten mit einer Nachkommastelle im Intervall  $[0, 1]$  untersucht. Das Ergebnis ist in Abbildung 7.9 festgehalten.

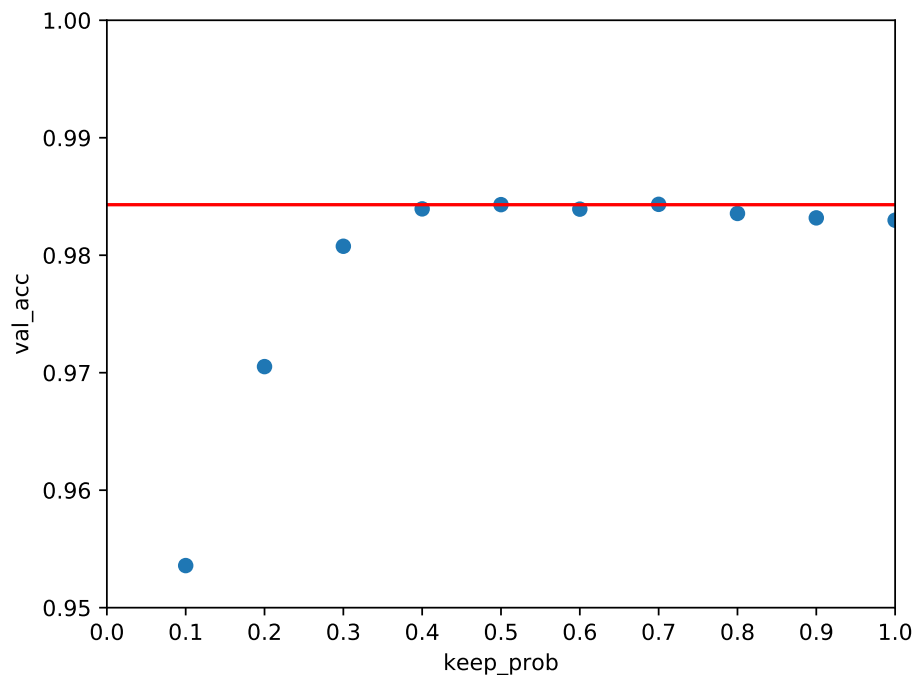


Abbildung 7.9: Hyperparameter-Optimierung: die Korrektklassifikationsrate in Abhängigkeit von der Dropout-Wahrscheinlichkeit (Eigene Darstellung).

Die höchste Korrektklassifikationsrate von 98,43 % wird bei zwei Werten erreicht: wenn 70 % der Signale erhalten bleiben und wenn die Hälfte der Signale abgeschaltet wird. Ähnliche Werte mit einer maximalen Abweichung von 0,08 % ergeben sich für alle Wahrscheinlichkeiten im Intervall  $[0,4, \dots, 0,8]$ . Für die weiteren Experimente wird für den `keep_prob`-Parameter ein Wert von 0,5 verwendet.

### 7.10.2 Auswertung der besten Hyperparameter-Kombination

Nach der Untersuchung der einzelnen Hyperparameter wird für jeden Hyperparameter der beste Wert festgelegt. Tabelle 7.3 zeigt die beste Hyperparameter-Kombination. Mit diesen Werten wird ein RNN trainiert und im Verzeichnis `RNN-SA/experiments/LSTM/checkpoints` in der Datei `weights.best.hdf5` gespeichert. Das Training dauert insgesamt drei Stunden und 42 Minuten auf einem Rechnercluster aus 40 CPUs<sup>6</sup> und wird in 22 Epochen durchgeführt, das heißt der gesamte Trainingsdatensatz wird 22 Mal analysiert, bis keine Verbesserung der Korrektklassifikationsrate mehr feststellbar ist. Abbildung 7.10 zeigt den Verlauf der Verlustfunktion sowie der Korrektklassifikationsrate des Evaluationsdatensatzes. Beide Kurven zeigen eine beinahe ideale Lernkurve. Die Verlustfunktion reduziert sich von 0,1982 auf 0,0318, was einer Verbesserung von über 80 % entspricht. Die Korrektklassifikationsrate steigt von anfangs 91,61 % auf 98,41 % an, was sehr dicht am idealen Ergebnis von 100 % liegt.

Tabelle 7.3: Beste Hyperparameter-Kombination (Eigene Darstellung).

Hyperparameter	Wert
num_epochs	150
batch_size	128
hidden_layer_size	319
num_cells	3
keep_prob	0,5

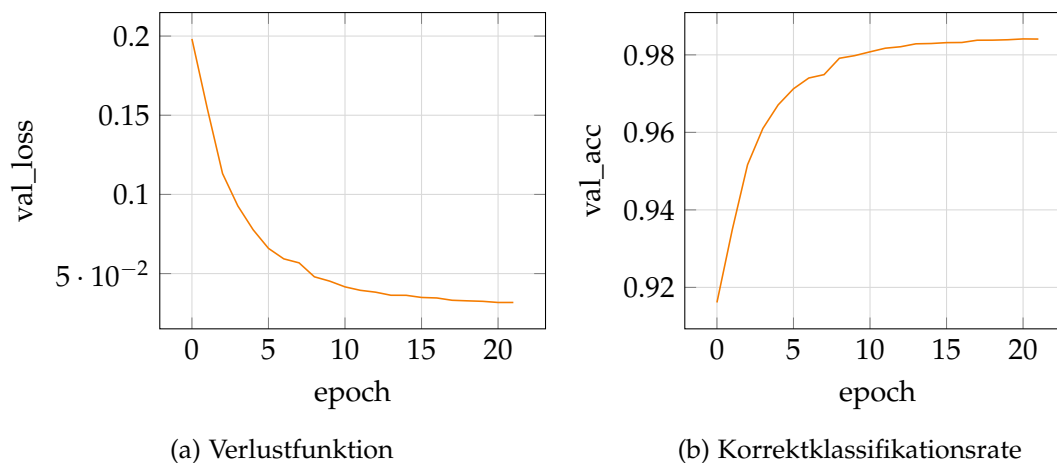


Abbildung 7.10: Verlustfunktion und Korrektklassifikationsrate des Evaluationsdatensatzes während des Trainings (Eigene Darstellung).

<sup>6</sup>Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz

Tabelle 7.4: Ergebnis des LSTM-RNNs mit der besten Hyperparameter-Kombination (Eigene Darstellung).

	<b>LSTM</b>
<b>tp</b>	152 714
<b>fp</b>	2319
<b>tn</b>	35 120
<b>fn</b>	680
<b>Gesamt</b>	190 833
<b>tp</b>	80,02 %
<b>fp</b>	1,22 %
<b>tn</b>	18,40 %
<b>fn</b>	18,76 %
<b>accuracy</b>	98,43 %
<b>precision</b>	98,50 %
<b>recall</b>	99,56 %
<b>Laufzeit</b>	30 s

Tabelle 7.4 zeigt das Ergebnis des RNN-Modells aus LSTM-Zellen mit der besten Hyperparameter-Kombination. Der Testdatensatz besteht aus 190 833 Task-Sets, von denen 98,43 % korrekt klassifiziert werden, was ein sehr gutes Ergebnis ist. Die Genauigkeit, das heißt der Anteil der korrekt als lauffähig klassifizierten Task-Sets an der Gesamtheit der als lauffähig klassifizierten Task-Sets beträgt sogar 98,50 %. Die Trefferquote besagt, dass 99,56 % der tatsächlich lauffähigen Task-Sets auch als solche klassifiziert werden. Auch die Anzahl der fälschlicherweise als lauffähig klassifizierten Task-Sets (fp) ist mit 1,22 % sehr gering. Die Berechnung der Vorhersagen für alle 190 833 Task-Sets benötigt ungefähr 30 s auf einem Rechnercluster aus 40 CPUs<sup>7</sup>, was einem Wert von 157  $\mu$ s je Task-Set entspricht. Zusammenfassend wird mit dem angewendeten Modell in kurzer Rechenzeit ein sehr gutes Ergebnis erzielt. Mit den nahezu optimalen Werten für die Korrektklassifikationsrate und der Genauigkeit ist der produktive Einsatz des RNNs im automotiven Kontext möglich.

### 7.10.3 Korrelation zwischen der RNN-Leistung und den untersuchten Hyperparametern

Zusätzlich zur Betrachtung der einzelnen Hyperparameter wird im Anschluss eine Zufallssuche mit den vielversprechendsten Werten, wie in Tabelle 7.5 festgehalten, durchgeführt. Dabei ergeben sich insgesamt 3900 verschiedene Kombinationsmöglichkeiten, von denen zufällig 390 ausgewählt werden. Da das Training der verschiedenen

<sup>7</sup>Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz

neuronalen Netze sehr viel Zeit in Anspruch nimmt und von Talos nach 44 getesteten Kombinationen immer noch eine Rechenzeit von 646 Stunden, entspricht ungefähr 27 Tagen, vorausgesagt wird, wird das Experiment deshalb an dieser Stelle aus Zeitgründen unterbrochen.

Tabelle 7.5: Hyperparameter-Raum der Zufallssuche (Eigene Darstellung).

Hyperparameter	Werte
num_epochs	150
batch_size	10 zufällig Werte aus dem Intervall [64, ..., 256] sowie die Werte [64, 128, 256]
hidden_layer_size	10 zufällige Werte aus dem Intervall [100, ..., 400]
num_cells	[3, 4, 5, 6, 7, 8]
keep_prob	[0,4, 0,5, 0,6, 0,7, 0,8]

Um dennoch einen Überblick über den Zusammenhang der einzelnen Hyperparameter mit der Leistung des RNNs zu bekommen, werden die Ergebnisse aller Experimente und der Zufallssuche zusammengefasst und eine Korrelationsmatrix erstellt, die in Abbildung 7.11 gezeigt ist.

Die Korrelation ist ein Maß für den Zusammenhang zwischen zwei oder mehreren Merkmalen [Wik19]. Dabei bedeutet ein Wert von 0, dass kein Zusammenhang zwischen den beiden Variablen besteht, und bei einem Wert von 1 ein sehr starker Zusammenhang. Auch die Richtung des Zusammenhangs ist von Bedeutung: bei einer positiven Korrelation bedeutet ein hoher Wert des eines Merkmals tendenziell auch einen hohen Wert des zweiten Merkmals, wohingegen bei einer negativen Korrelation, auch Antikorrelation, zu einem hohen Wert des einen Merkmals tendenziell ein niedriger Wert des anderen Merkmals gehört.

Wie aus Abbildung 7.11 ersichtlich, ist die Leistung des neuronalen Netzes in Form der Korrektklassifikationsrate sehr stark von der Größe der versteckten Schichten abhängig, mit einer Korrelation von 0,9. Auch die Anzahl der Zellen bzw. versteckten Schichten hat einen großen Einfluss auf die Korrektklassifikationsrate. Das lässt sich damit erklären, dass die Funktion zwischen den Task-Attributen und der Lauffähigkeit eines Task-Sets komplex ist, weshalb eine größere Anzahl an Neuronen bessere Ergebnisse liefert. Die Anzahl der Epochen sowie die Größe der Teildatensätze spielen jedoch keine wesentliche Rolle. Das liegt möglicherweise auch daran, dass nur sehr wenige Experimente mit diesen beiden Merkmalen durchgeführt wurden und die Korrelation deshalb im Vergleich geringer ausfällt. Eine Antikorrelation liegt beim Parameter `keep_prob` vor, die bedeutet: je mehr Signale nach jeder LSTM-Zelle ausgeschaltet werden, desto besser ist die Leistung des Modells.

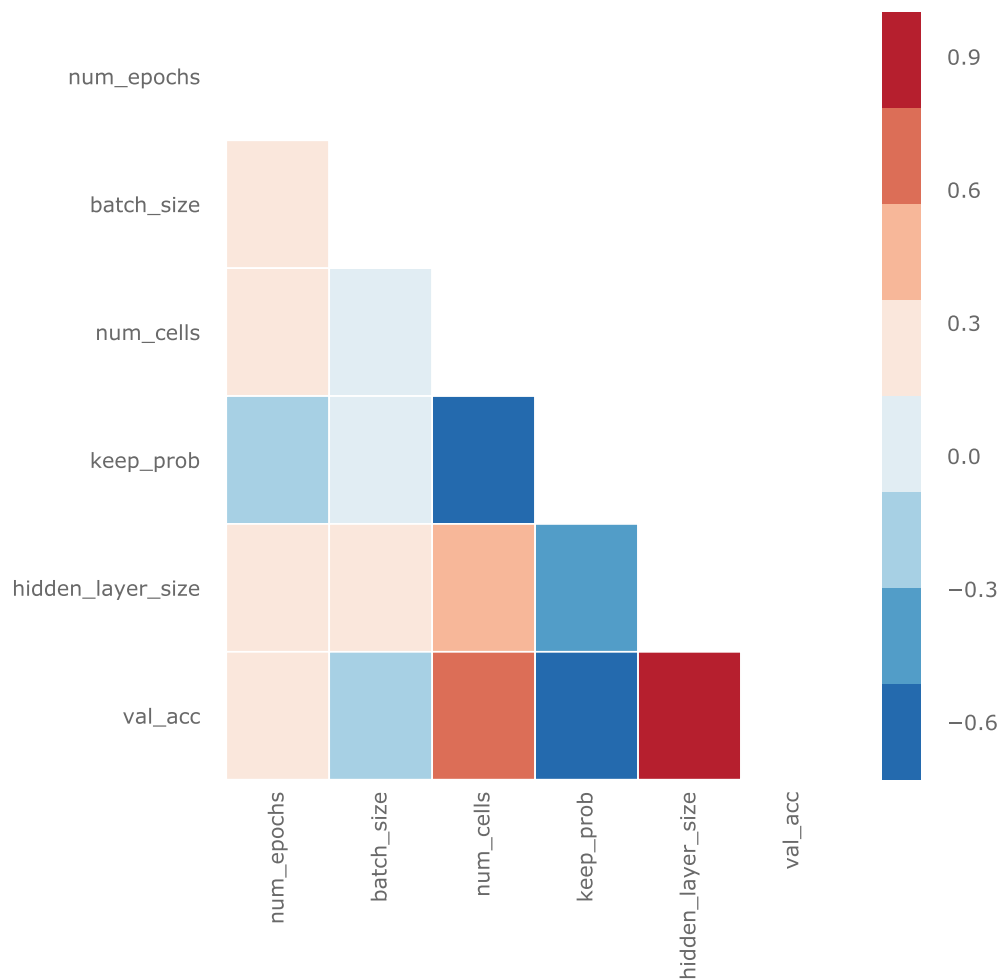


Abbildung 7.11: Korrelationsmatrix: Zusammenhang zwischen den Hyperparametern und der Leistung des RNNs (Eigene Darstellung).

Zusammengefasst kann anhand der Korrelationsmatrix festgehalten werden, dass die Leistung eines rekurrenten neuronalen Netzes im Wesentlichen von der Anzahl der Zellen, der Größe der versteckten Schichten und der Dropout-Wahrscheinlichkeit abhängt.



## 8 Vergleich der Schedulability-Analyseansätze

Ein direkter Vergleich zwischen den klassischen Schedulability-Analyseverfahren und dem rekurrenten neuronalen Netz ist aufgrund dessen, dass die Datenbank für das Machine Learning mit der Antwortzeit-Analyse gefiltert wird, nicht möglich. Deswegen werden die konventionellen Methoden noch einmal für die gefilterte Datenbank ausgeführt. Zudem werden für alle Task-Sets der Tabelle *CorrectTaskSet* Vorhersagen mit dem rekurrenten neuronalen Netz getroffen, da die Evaluation des RNNs nur anhand des Testdatensatzes durchgeführt wird, das bedeutet nur mit einem Teil der Tabelle. Auf diese Art und Weise ist die Leistung der verschiedenen Schedulability-Analyseansätze für die gefilterte Datenbank vergleichbar. Das Ergebnis ist in Tabelle 8.1 zusammengefasst.

Das beste Ergebnis zeigt die Antwortzeit-Analyse, da dieses Verfahren zur Filterung der Datenbank angewendet wird und folglich die 100 %-Marke widerspiegelt. Da der Startwert nur minimale Auswirkungen auf das Ergebnis der Antwortzeit-Analyse hat, werden beide Verfahren als Bezugspunkt für die anderen Methoden betrachtet.

Wird die Leistung eines Schedulability-Analyseverfahrens anhand der Korrekturklassifikationsrate beurteilt, so liefern das LSTM-Netz und die Simulation die besten Ergebnisse mit 98,55 % und 98,16 %. Aber auch der Hyperplanes  $\delta$ -Exact Test ist mit 96,34 % sehr gut. Wie bereits bei der Evaluation der traditionellen Verfahren kommen auch hier die Prozessorauslastungs-Tests nicht an die Werte der anderen Methoden heran, da diese Tests nicht exakt sind. Überraschend ist jedoch, dass der Arbeitsbelastungs-Test für den RM-Scheduler mit 95,79 % in der gleichen Größenordnung wie der  $\delta$ -HET liegt. Das deutet darauf hin, dass der Unterschied zwischen dem FP- und dem RM-Scheduler bei dem verwendeten Task-Modell und den gegebenen Task-Sets keine große Rolle spielt.

In Echtzeitsystemen spielt die Genauigkeit eine sehr wichtige Rolle. Diese muss dabei möglichst hoch sein, da sie den Anteil der lauffähigen Task-Sets an allen als lauffähig klassifizierten Task-Sets wiedergibt. Anders ausgedrückt bedeutet eine geringe Genauigkeit, dass viele Task-Sets als lauffähig eingestuft werden, obwohl sie es in Wirklichkeit nicht sind. Eine solche Falsch-Klassifizierung ist gerade in sicherheitskritischen Systemen fatal und kann zum Systemversagen führen. Bei den untersuchten Verfahren erreicht das RNN das beste Ergebnis mit einer Genauigkeit von 98,58 %, dicht gefolgt von der Simulation mit 97,76 % und dem  $\delta$ -HET mit 97,29 %. Auch hier sind die Prozessorauslastungs-Tests wieder deutlich schlechter und insofern für den gedachten Einsatz nicht empfehlenswert.

Tabelle 8.1: Ergebnisse der Schedulability-Analyseverfahren im Vergleich (Eigene Darstellung).

	LSTM		Simulation		Prozessorauslastung			Antwortzeit-Analyse		Arbeitsbelastung	
					einfach	RM	HB	Audsley	Buttazzo	RM	$\delta$ -HET
tp	1 527 687	1 533 251	1 524 892	1 495 837	1 528 318	1 533 259	1 533 259	1 533 255	1 505 280		
fp	22 061	35 088	215 873	171 349	222 269	118	0	80 278	41 878		
tn	353 007	339 980	159 195	203 719	152 799	374 950	375 068	294 790	333 190		
fn	5572	8	8367	37 422	4941	0	0	4	27 979		
Gesamt	1 908 327	1 908 327	1 908 327	1 908 327	1 908 327	1 908 327	1 908 327	1 908 327	1 908 327		
tp	80,05 %	80,35 %	79,91 %	78,38 %	80,09 %	80,35 %	80,35 %	80,35 %	78,88 %		
fp	1,16 %	1,84 %	11,31 %	8,98 %	11,65 %	0,01 %	0,00 %	4,21 %	2,19 %		
tn	18,50 %	17,82 %	8,34 %	10,68 %	8,01 %	19,65 %	19,65 %	15,45 %	17,46 %		
fn	0,29 %	0,00 %	0,44 %	1,96 %	0,26 %	0,00 %	0,00 %	0,00 %	1,47 %		
accuracy	98,55 %	98,16 %	88,25 %	89,06 %	88,09 %	99,99 %	100,00 %	95,79 %	96,34 %		
precision	98,58 %	97,76 %	87,60 %	89,72 %	87,30 %	99,99 %	100,00 %	95,02 %	97,29 %		
recall	99,64 %	100,00 %	99,45 %	97,56 %	99,68 %	100,00 %	100,00 %	100,00 %	98,18 %		
Laufzeit	5 min 3 s	1 h 48 min 1 s	6 s	7 s	5 s	1 min 10 s	1 min 9 s	1 min 10 s	1 min 8 s		



---

Obwohl die Trefferquote in sicherheitskritischen Systemen im Vergleich zur Genauigkeit keine so große Rolle spielt, wird sie dennoch der Vollständigkeit halber kurz bewertet. Die Trefferquote gibt wieder, wie viele der tatsächlich lauffähigen Task-Sets auch als solche erkannt werden. Das schlechteste Ergebnis von 97,56 % liegt beim Prozessorauslastungs-Test für den RM-Scheduler vor. Die Simulation und der RM-Arbeitsbelastungs-Test erreichen sogar die 100 %. Alle Verfahren schneiden sehr gut ab und es gibt keinen nennenswerten Unterschied, das heißt alle Verfahren erkennen tatsächlich lauffähige Task-Sets sehr gut.

Wie bereits erwähnt sind falsch als lauffähig klassifizierte Task-Sets extrem problematisch, weshalb der Anteil der falsch-positiven Task-Sets verglichen wird. Hier schneiden das LSTM-RNN mit nur 1,16 % sowie die Simulation mit 1,84 % am besten ab. Anders als bei der Korrektklassifikationsrate oder der Genauigkeit liefern die auf der Arbeitsbelastung basierenden Methoden mit 4,21 % und 2,19 % deutlich schlechtere Ergebnisse. Gleichbleibend sind die ungefähr um 10 % schlechteren Werte der Prozessorauslastungs-Tests.

Ziel dieser Arbeit ist es, neben der Leistung eines RNNs auch die Geschwindigkeit der Vorhersage mit den gängigen Schedulability-Analyseverfahren zu vergleichen. Hier werden die Erwartungen nicht erfüllt, da das RNN für die 1 908 327 getesteten Task-Sets fünf Minuten und drei Sekunden, entspricht 159  $\mu$ s je Task-Set, für die Vorhersagen auf einem Rechnercluster aus 40 CPUs<sup>1</sup> benötigt, was mehr als vier Mal so lange ist wie die Rechenzeit der Antwortzeit-Analyse oder des  $\delta$ -HET mit ungefähr 36  $\mu$ s pro Task-Set. Lediglich die Simulation liefert mit 3 ms je Task-Set eine noch langsamere Vorhersage über die Lauffähigkeit eines Task-Sets.

Zusammengefasst ist festzustellen, dass die Leistung des RNNs für die Schedulability-Analyse sehr gut bzw. nahezu optimal ist und auf jeden Fall mit der Simulation und der Antwortzeit-Analyse, das heißt exakten Methoden der Schedulability-Analyse, vergleichbar ist. Allerdings wird die erwartete Geschwindigkeit beim Treffen der Lauffähigkeitsvorhersagen nicht erreicht. Dies ist damit erklärbar, dass das verwendete RNN-Modell eine bedeutende Komplexität beinhaltet und die Berechnungen deshalb aufwändiger als bei der Antwortzeit-Analyse sind. Zudem werden keine GPUs für die Berechnungen des neuronalen Netzes verwendet, was ebenfalls eine Erklärung für die lange Ausführungszeit ist.

---

<sup>1</sup>Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz



## 9 Einschränkungen und Ausblick

In dieser Arbeit wurden verschiedene traditionelle Schedulability-Analyseverfahren implementiert. Außerdem wurde ein rekurrentes neuronales Netz aus LSTM-Zellen für die Vorhersage der Lauffähigkeit von Task-Sets entwickelt. Es wurden mehrere Experimente mit verschiedenen Hyperparametern durchgeführt, um die für diesen Anwendungsfall optimale Einstellung des neuronalen Netzes abzuschätzen. Beide Ansätze für die Schedulability-Analyse wurden evaluiert und anschließend miteinander verglichen.

### Stand der Technik

Die erste Einschränkung trat bereits während der Literaturrecherche auf, da für den FP-Scheduler nur exakte, aber keine heuristischen Schedulability-Analyseverfahren gefunden wurden. Außerdem bezogen sich viele der recherchierten Methoden nur auf spezielle FP-Scheduler, bei denen die Prioritäten zum Beispiel nach den Perioden (RM-Scheduler) vergeben werden. Literatur über die Anwendung von Methoden des maschinellen Lernens für die Schedulability-Analyse war ebenfalls sehr wenig zu finden, wobei diese zudem nur für spezifische Anwendungsfälle konzipiert sind.

### Gegebene Daten

Zu Beginn der Implementierung wurde die Datenbank betrachtet, wobei festgestellt wurde, dass eine für die Schedulability-Analyse enorm wichtige Task-Eigenschaft darin nicht enthalten ist: Die Ausführungszeit im schlimmsten Fall (engl. *Worst-Case Execution Time*, WCET). Dieses Problem wurde vorerst durch ein Benchmark-Modul gelöst, das die durchschnittliche Laufzeit für jeden Beispieltask bestimmt. In Zukunft sollte hier aber auf jeden Fall nachgebessert werden, indem eine exakte WCET für jeden Task in der Datenbank gespeichert wird. Des Weiteren sind die in dieser Arbeit berücksichtigten Beispieltasks eher synthetisch und haben bisher keinen Bezug zur Realität. Deshalb sollten die Tasks ausgetauscht werden, zum Beispiel mit Hilfe von Frameworks wie COBRA (*COde Behaviour fRAMework*)<sup>1</sup> oder TASKers [Eic+18], mit denen realistische Tasks mit exakter WCET und Task-Sets generiert werden.

Die Aufteilung der Task-Sets in die beiden Klassen „lauffähig“ und „nicht lauffähig“ ist in der verwendeten Version der Datenbank nicht ideal: 78 % der Task-Sets sind lauffähig, nur 22 % sind dies nicht. Für maschinelles Lernen wäre eine Verteilung von 50:50 ideal, was in zukünftigen Versionen der Datenbank berücksichtigt werden sollte.

---

<sup>1</sup><http://cobra.idlab.uantwerpen.be/index.php>

Gegen Ende dieser Arbeit wurde die Ursache für das nicht exakte Ergebnis der Simulation geklärt: Das Verhalten des Genode-Betriebssystems. Die in der Tabelle *Job* gespeicherten Start- und Endzeitpunkte sind die Zeiten, zu denen ein Job gestartet bzw. vollständig ausgeführt wurde, Unterbrechungen durch höher-priorisierte Tasks mit einbezogen. Was in diesen Zeiten allerdings nicht enthalten ist, sind die Zeiträume, die Genode für das Starten und Beenden jedes Tasks benötigt. In diesen Zeiträumen vor und nach der Ausführung jedes Tasks wird zum Beispiel Speicherplatz allokiert und wieder freigegeben. Wie lange das Starten bzw. Beenden eines Tasks dauert, ist aber nicht nachvollziehbar und kann auch nicht aufgezeichnet werden. Deshalb sind die verwendeten Ausführungszeiten der Tasks stellenweise sehr weit von der Realität entfernt, was das Ergebnis der Simulation und der anderen Verfahren verzerrt. Auch wird mit diesem Wissen der Einsatz des Datenbank-Filters unnötig, da die Daten nicht fehlerhaft sind, sondern lediglich für die traditionellen Schedulability-Analyseverfahren nicht sinnvoll nutzbar. Aus diesem Grund sollte der Filter entfernt werden und das rekurrente neuronale Netz erneut mit der kompletten Datenbank trainiert werden.

### Datenbankschnittstelle

Aktuell werden die Task-Sets aus der Datenbank gelesen, was für das *RNN-SA*-Projekt auch so vorgesehen ist. Im *traditional-SA*-Projekt sollten hingegen die Task-Sets nicht aus der Datenbank, sondern direkt vom Distributor kommen. Außerdem wurden der Übersichtlichkeit halber für die Tasks und Task-Sets eigene Klassen ähnlich denen im Taskloader implementiert. Beim Einlesen der Daten nimmt die Erstellung der Task- bzw. Taskset-Objekte jedoch sehr viel Rechenzeit in Anspruch, deshalb sollten die Klassen weggelassen werden. Im *RNN-SA*-Projekt wurde dies bereits teilweise umgesetzt, da neuronale Netze nur numerische Felder als Eingabe akzeptieren. Das Format der Daten sollte in beiden Projekten angeglichen werden und eventuell an das Format des Distributors angepasst werden, sodass die Daten im gesamten *MaLSAMi*-Projekt einheitlich sind.

In dieser Arbeit werden alle Task-Sets auf einmal aus der Datenbank gelesen und im Arbeitsspeicher abgelegt, was allerdings sehr viel Speicherkapazität besetzt. Deshalb sollten die Daten nach und nach durch eine Pipeline geladen werden, um den Speicherbedarf zu reduzieren und das Lesen der Daten zu beschleunigen.

### Schedulability-Analyse Frameworks

Bei der Suche nach Frameworks der Schedulability-Analyse trat eine weitere Einschränkung auf. Zwar wurden einige Frameworks untersucht, dennoch passte am Ende nur ein einziges Framework zu dem verwendeten Scheduler und dem Task-Modell. Zudem war bei einigen Frameworks kein gültiger Download verfügbar oder gar keine Implementierung vorhanden. Deshalb mussten schließlich die traditionellen Schedulability-Analyseverfahren, bis auf die Simulation, selbständig implementiert werden.

---

## Datenvorverarbeitung

Der Lernerfolg eines neuronalen Netzes hängt sehr stark von den Eingabedaten ab. Deshalb ist die Datenvorbereitung ein wichtiger Schritt in der Entwicklung eines Machine Learning Modells. Aktuell ist sowohl die Normierung als auch die Standardisierung der Task-Attribute im *RNN-SA*-Projekt enthalten. Durchgeführt wird aber nur die Datennormierung, um die Eingabewerte auf das Intervall  $[0, 1]$  zu normieren. Allerdings wurde aus Zeitgründen nicht genauer recherchiert, welchen Einfluss die Normierung bzw. Standardisierung auf die Leistung eines neuronalen Netzes hat, sowie welcher Unterschied bezüglich der Leistung zwischen den beiden Verfahren besteht. Überdies ist nicht klar, wie die Auswirkungen der Normierung/Standardisierung auf voneinander abhängigen Daten, wie zum Beispiel Deadline und Periode oder Argument und PKG, aussehen und ob solche Daten auf besondere Weise behandelt werden müssen. Es sollte folglich eine gründliche Recherche zu dem Thema Standardisierung und Normierung im Machine Learning durchgeführt werden und die Datenvorverarbeitung entsprechend angepasst werden.

## Machine Learning Frameworks

Aus der Idee heraus, ein gründliches Verständnis für das Framework TensorFlow wäre notwendig, um ein leistungsfähiges RNN zu entwickeln, wurde einige Zeit in die Einarbeitung in TensorFlow investiert. Die für diese Arbeit benötigten Einstellmöglichkeiten sind aber auch in der Abstraktionsbibliothek Keras verfügbar, mit der die Erstellung des RNNs recht einfach und auch ohne tiefergehendes Verständnis für TensorFlow möglich ist.

Um das Training der neuronalen Netze vor allem während der Hyperparameter-Optimierung zu beschleunigen, wurde versucht, die GPU-Unterstützung von Keras bzw. TensorFlow zu nutzen. Jedoch gestaltete sich die Installation der benötigten Softwarekomponenten und Treiber extrem schwierig, da abhängig vom installierten Betriebssystem und der/n verfügbaren Graphikkarte/n spezifische Versionen notwendig sind. Zusätzlich muss auch noch das Keras-Modell angepasst werden, um es auf mehreren GPUs ausführen zu können. Aus Zeitgründen wurden die neuronalen Netze schließlich ohne GPU-Support auf einem Rechnercluster mit 40 CPUs<sup>2</sup> trainiert. Um bei zukünftigen Experimenten die Trainingsdauer zu reduzieren, sollte das Modell so erweitert werden, dass die Ausführung auf GPUs möglich ist.

## Machine Learning Modelle

Während dieser Arbeit wurde ein rekurrentes neuronales Netz aus LSTM-Zellen implementiert, da diese Form in der Literatur am häufigsten zitiert wird. Die ursprünglich vorgesehene Entwicklung eines klassischen RNNs aus rekurrenten Neuronenschichten sowie eines RNNs bestehend aus GRU-Zellen war im zeitlichen Rahmen der Arbeit

---

<sup>2</sup>Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz

nicht realisierbar. Der Vollständigkeit halber sollten diese neuronalen Netze aber auf jeden Fall implementiert und einer Hyperparameter-Optimierung unterzogen werden, da sich möglicherweise bei einer anderen Art von RNN bessere Ergebnisse ergeben.

### Hyperparameter-Optimierung

Um den Zeitrahmen der Arbeit trotz einiger neuer Ideen, Herausforderungen und Probleme einzuhalten, wurde nur mit den ausgewählten Hyperparametern experimentiert. Experimente mit anderen Hyperparametern, wie zum Beispiel dem Optimierungsalgorithmus oder der Aktivierungsfunktion, wären aber definitiv sinnvoll und zu empfehlen. Auch können noch andere Hyperparameter, die in dieser Arbeit überhaupt nicht berücksichtigt wurden (z.B. Lernrate), an verschiedenen Stellen des Machine Learning Modells untersucht werden, um so die Leistung des RNNs noch mehr zu steigern.

### Vergleich

Ziel dieser Arbeit war der Vergleich von heuristischen Schedulability-Analyseverfahren mit Deep Learning basierten Ansätzen. Da die Recherche zu heuristischen Verfahren für den FP-Scheduler keine Ergebnisse lieferte, wurden nur exakte und hinreichende Methoden implementiert. Infolge dessen, dass die Entwicklung dieser Komponente und die Hyperparameter-Optimierung des neuronalen Netzes sich deutlich zeitintensiver gestaltete als geplant, wurde der Vergleich nur zu einem rekurrenten neuronalen Netz aus LSTM-Zellen durchgeführt. Um am Ende eindeutige Aussagen zu treffen, welcher Schedulability-Analyseansatz in diesem Anwendungsfall die besten Ergebnisse liefert, sollten aber zum einen die beiden anderen RNN-Arten implementiert werden, und zum anderen ein umfassender Vergleich zwischen den traditionellen Methoden und allen bereits in früheren Arbeiten entwickelten Machine Learning Modellen durchgeführt werden. Einschließlich dieser Arbeit stehen für einen solchen Vergleich verschiedene traditionelle Verfahren, mehrere Shallow Learning Modelle, ein neuronales Netz aus dem Deep Learning sowie ein rekurrentes neuronales Netz (RNN) zur Verfügung. Interessant wäre ferner noch die Anwendung eines konvolutionalen neuronalen Netzes (CNN) oder einer Kombination aus RNN und CNN.

### Sonstiges

Die in dieser Arbeit entwickelte Komponente für die traditionellen Schedulability-Analyseverfahren ist Teil des *MaLSAMi*-Projektes, aber momentan noch nicht in dessen Toolchain integriert. Das *RNN-SA*-Projekt ist bereits für den Einsatz in der Toolchain konstruiert. Die Machine Learning Modelle aus den früheren Arbeiten sind das dagegen nicht, da die Toolchain damals noch nicht in ihrer jetzigen Form vorlag. Deshalb sollten zum einen die Schnittstellen zwischen den einzelnen Komponenten festgelegt werden, und zum anderen die Schedulability-Analyse-Module entsprechend angepasst werden, um den umfassenden Vergleich zwischen den Ansätzen zu ermöglichen.

## 10 Fazit

Nach einer Einarbeitung in die Themen der Schedulability-Analyse in Echtzeitsystemen und maschinelles Lernen wurden zunächst verschiedene Schedulability-Analyseverfahren für den FP- und den EDF-Algorithmus recherchiert. Anschließend wurden die Simulation, drei Prozessorauslastungs-Tests, die Antwortzeit-Analyse mit zwei verschiedenen Startwerten und zwei auf der Arbeitsbelastung basierende Tests implementiert. Für die Simulation wurde das Framework SimSo verwendet. Außerdem wurde eine Schnittstelle zu der gegebene Datenbank, die die Tasks und Task-Sets enthält, entwickelt, sodass eine Schedulability-Analyse mit den implementierten Verfahren möglich war. Es zeigte sich, dass selbst die exakten Verfahren nur eine Korrektklassifikationsrate von 93 % erreichten, was durch das Verhalten des Genode-Betriebssystems und daraus resultierenden ungenauen Task-Ausführungszeiten erklärt werden konnte.

Zum Vergleich wurde ein rekurrentes neuronales Netz aus LSTM-Zellen mit Keras und TensorFlow entwickelt. Um das Machine Learning Modell nicht mit zu diesem Zeitpunkt noch als fehlerhaft angesehenen Daten zu trainieren, wurde die Datenbank durch ein exaktes Schedulability-Analyseverfahren gefiltert. Zur Maximierung der Leistung des neuronalen Netzes wurden verschiedenen Hyperparameter festgelegt, mit denen eine Vielzahl von Experimenten durchgeführt wurde. So wurde zum einen der Zusammenhang zwischen den Hyperparametern und der Leistung des RNNs identifiziert und zum anderen die beste Hyperparameter-Einstellung für die Schedulability-Analyse bestimmt. Das RNN bestehend aus drei LSTM-Zellen mit je 319 versteckten Neuronen zeigte das beste Ergebnis mit einer Korrektklassifikationsrate von 98,43 %. Durch die Experimente wurde deutlich, dass die Leistung des RNNs am stärksten von der Anzahl der Neuronen abhängt, gefolgt von der Anzahl der Zellen und der Dropout-Wahrscheinlichkeit.

Das Ergebnis der Antwortzeit-Analyse wurde beim Vergleich der beiden Schedulability-Analyseansätzen als Bezugspunkt angewendet. Das Ergebnis des RNNs mit 98,58 % Genauigkeit kommt bereits sehr nahe an den Maximalwert von 100 % heran und übertrifft alle anderen traditionellen Schedulability-Analyseverfahren. Die erwartete Geschwindigkeit gegenüber den herkömmlichen Methoden wurde jedoch nicht bestätigt.

Abschließend wird festgehalten, dass der Einsatz eines rekurrenten neuronalen Netzes aus LSTM-Zellen für die Schedulability-Analyse in einem Echtzeitsystem anhand der verglichenen Merkmale definitiv möglich ist.





# Abkürzungsverzeichnis

<b>ARM</b>	Mikroprozessor-Architektur (engl. <i>Advanced RISC Machine</i> )
<b>BPTT</b>	Trainingsverfahren (engl. <i>Backpropagation Through Time</i> )
<b>C-ITS</b>	kooperatives intelligentes Transportsystem (engl. <i>Cooperative Intelligent Transportation System</i> )
<b>CNN</b>	konvolutionales neuronales Netz (engl. <i>Convolutional Neural Network</i> )
<b>CPU</b>	zentrale Recheneinheit (engl. <i>Central Processing Unit</i> )
<b>CSV</b>	Text-/Dateiformat (engl. <i>Comma-Separated Values</i> )
<b><math>\delta</math>-HET</b>	Hyperplanes $\delta$ -Exact Test
<b>DM</b>	Planen nach monotonen Fristen (engl. <i>Deadline Monotonic</i> )
<b>ECU</b>	elektronisches Steuergerät (engl. <i>Electronic Control Unit</i> )
<b>EDF</b>	Planen nach Fristen (engl. <i>Earliest Deadline First</i> )
<b>fn</b>	falsch negativ (engl. <i>false negative</i> )
<b>FP</b>	Planen nach festen Prioritäten (engl. <i>Fixed Priority</i> )
<b>fp</b>	falsch positiv (engl. <i>false positive</i> )
<b>GPU</b>	Grafikprozessor (engl. <i>Graphics Processing Unit</i> )
<b>GRU</b>	Gedächtniszelle (engl. <i>Gated Recurrent Unit</i> )
<b>HB</b>	Hyperbolic Bound
<b>ITS</b>	intelligentes Verkehrs- und Transportsystem (engl. <i>Intelligent Transport System</i> )
<b>KIA4SM</b>	Kooperative Integrationsarchitektur für zukünftige Smart-Mobility-Lösungen
<b>LCM</b>	kleinstes gemeinsames Vielfaches (engl. <i>Least Common Multiple</i> )
<b>LLF</b>	Planen nach Spielräumen (engl. <i>Least Laxity First</i> )
<b>LSTM</b>	Gedächtniszelle (engl. <i>Long Short-Term Memory</i> )

<b>MaLSAMi</b>	Machine Learning gestützte Schedulability Analyse für die Migration von Softwarekomponenten zur Laufzeit (engl. <i>Machine-Learning supported Schedulability Analysis for Migration of software-based components during runtime</i> )
<b>OS</b>	Betriebssystem (engl. <i>Operating System</i> )
<b>PCA</b>	Hauptkomponentenzerlegung (engl. <i>Principal Component Analysis</i> )
<b>QEMU</b>	Virtualisierungssoftware (engl. <i>Quick Emulator</i> )
<b>RM</b>	Planen nach Raten (engl. <i>Rate Monotonic</i> )
<b>RNN</b>	rekurrentes neuronales Netz (engl. <i>Recurrent Neural Network</i> )
<b>RR</b>	Rundlauf-Verfahren (engl. <i>Round Robin</i> )
<b>RTA</b>	Antwortzeit-Analyse (engl. <i>Response Time Analysis</i> )
<b>SQL</b>	Programmiersprache für Datenbanksysteme (engl. <i>Structured Query Language</i> )
<b>TCP</b>	Übertragungssteuerungsprotokoll (engl. <i>Transmission Control Protocol</i> )
<b>tn</b>	richtig negativ (engl. <i>true negative</i> )
<b>tp</b>	richtig positiv (engl. <i>true positive</i> )
<b>WCET</b>	maximale Ausführungszeit (engl. <i>Worst-Case Execution Time</i> )
<b>WCRT</b>	Antwortzeit im ungünstigsten Fall (engl. <i>Worst-Case Response Time</i> )
<b>x86</b>	Mikroprozessor-Architektur

# Abbildungsverzeichnis

1.1	KIA4SM Vision - eine homogene Plattform für heterogene Geräte . . . . .	2
2.1	Aufbau des <i>MaLSAMi</i> -Projektes . . . . .	6
2.2	Architektur des KIA4SM-Systems . . . . .	7
3.1	Teilbereiche der künstlichen Intelligenz . . . . .	18
3.2	Allgemeine Architektur eines neuronalen Netzes . . . . .	20
3.3	Aufteilung der Beispieldaten in verschiedene Datensätze . . . . .	22
3.4	Aufbau eines biologischen Neurons . . . . .	23
3.5	Mathematische Darstellung eines künstlichen Neurons . . . . .	24
3.6	Aktivierungsfunktionen: Sprungfunktion und sigmoide Funktion . . . . .	25
3.7	Beispiel für ein einfaches neuronales Netz . . . . .	26
3.8	Beispiel für ein mehrschichtiges neuronales Netz . . . . .	28
3.9	Aktivierungsfunktionen des Deep Learnings . . . . .	34
3.10	Ein- und Ausgabemuster bei RNNs . . . . .	36
3.11	Eingabe-Dimensionen eines vorwärts gerichteten Netzes und eines RNNs	37
3.12	Darstellung eines rekurrenten Neurons . . . . .	38
3.13	Darstellung einer rekurrenten Neuronenschicht . . . . .	38
3.14	Darstellung einer Gedächtniszelle . . . . .	39
3.15	Standard Backpropagation through Time (BPTT) . . . . .	40
3.16	Truncated Backpropagation through Time . . . . .	41
3.17	Aufbau einer LSTM-Zelle . . . . .	42
3.18	Aufbau einer GRU-Zelle . . . . .	44
5.1	Struktur der gegebenen Datenbank . . . . .	58
6.1	Blockdiagramm des <i>traditional-SA</i> -Projekts . . . . .	63
6.2	Klassen der Datenbankschnittstelle . . . . .	68
6.3	Beispiel für die Simulation . . . . .	81
6.4	Programmablaufplan der Prozessorauslastungs-Tests . . . . .	82
6.5	Programmablaufplan der Antwortzeit-Analyse . . . . .	86
6.6	Programmablaufplan des RM-Arbeitsbelastungs-Tests . . . . .	91
6.7	Programmablaufplan des Hyperplanes $\delta$ -Exact Tests ( $\delta$ -HET) . . . . .	96
7.1	Aufbau des <i>MaLSAMi</i> -Projektes mit Filter . . . . .	106
7.2	Blockdiagramm des <i>RNN-SA</i> -Projekts . . . . .	107
7.3	Form der Eingabe-Daten des RNNs . . . . .	110

7.4	Aufbau des Keras-RNN-Modells . . . . .	118
7.5	Hyperparameter-Optimierung: Anzahl der Epochen . . . . .	130
7.6	Hyperparameter-Optimierung: Größe der Teildatensätze . . . . .	131
7.7	Hyperparameter-Optimierung: Größe der versteckten Schichten . . . . .	132
7.8	Hyperparameter-Optimierung: Anzahl der Zellen . . . . .	132
7.9	Hyperparameter-Optimierung: Dropout-Wahrscheinlichkeit . . . . .	133
7.10	Verlustfunktion und Korrektklassifikationsrate während des Trainings . .	134
7.11	Korrelationsmatrix . . . . .	137

# Tabellenverzeichnis

2.1	Verfügbare Beispieltasks für die Erzeugung von Task-Sets . . . . .	8
5.1	Beispiele aus der Tabelle <i>TaskSet</i> . . . . .	58
5.2	Beispiele aus der Tabelle <i>Task</i> . . . . .	59
5.3	Beschreibung der Task-Attribute . . . . .	60
5.4	Verteilung der Task-Sets . . . . .	60
6.1	Kommandozeilen-Argumente des <i>traditional-SA</i> -Projekts . . . . .	67
6.2	Beispiele aus der Tabelle <i>ExecutionTime</i> . . . . .	73
6.3	Schedulability-Analyse Frameworks . . . . .	76
6.4	Zusammenfassung der wichtigsten Attribute des Beispiel-Task-Sets 46 429	83
6.5	Ergebnisse der traditionellen Schedulability-Analyseverfahren . . . . .	101
7.1	Schritte der Datenvorverarbeitung anhand eines Beispieltasks . . . . .	111
7.2	Beschreibung der Konfigurations- und Hyperparameter . . . . .	125
7.3	Beste Hyperparameter-Kombination . . . . .	134
7.4	Ergebnis des LSTM-RNNs . . . . .	135
7.5	Hyperparameter-Raum der Zufallssuche . . . . .	136
8.1	Vergleich der Schedulability-Analyseverfahren . . . . .	140



# Listingverzeichnis

4.1	Quick convergence Processor-demand Analysis (QPA) . . . . .	52
6.1	Hauptprogramm des <i>traditional-SA</i> -Projektes . . . . .	64
6.2	Erstellen eines Database-Objektes . . . . .	65
6.3	Testen eines Datensatzes . . . . .	66
6.4	Konstruktor der Klasse Database . . . . .	70
6.5	Bestimmung der durchschnittlichen Task-Ausführungszeiten . . . . .	73
6.6	Definition der Logging-Handler . . . . .	74
6.7	Implementierung des FP-/EDF-Schedulers . . . . .	78
6.8	Hinzufügen eines Tasks zu einem SimSo-System . . . . .	80
6.9	Berechnung der Antwortzeit . . . . .	88
6.10	Bestimmung der Planungspunkte . . . . .	93
6.11	Berechnung der Arbeitsbelastung in Relation zum Planungspunkt . . . . .	93
6.12	Berechnung der Arbeitsbelastung . . . . .	93
6.13	Bestimmung der Arbeitsbelastung beim $\delta$ -HET . . . . .	97
7.1	Hauptprogramm des <i>RNN-SA</i> -Projektes . . . . .	108
7.2	Implementierung des Datenbank-Filters . . . . .	115
7.3	Funktion zur Beschreibung des LSTM-Modells . . . . .	119
7.4	Hyperparameter-Dictionary für die Verwendung mit Talos . . . . .	123
7.5	Konfigurationsparameter-Dictionary für allgemeine Einstellungen . . . . .	124
7.6	Durchführen eines Talos-Experiments . . . . .	126
7.7	Öffnen und Lesen einer CSV-Datei . . . . .	127
7.8	Darstellung einer Funktion $y = f(x)$ . . . . .	128
7.9	Bestimmung der Wahrheitsmatrix . . . . .	128
7.10	Visualisierung der Korrelationsmatrix . . . . .	129





# Literatur

- [Amn+04] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson und W. Yi. „TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems“. In: *Formal Modeling and Analysis of Timed Systems*. Hrsg. von K. G. Larsen und P. Niebert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, S. 60–72. ISBN: 978-3-540-40903-8.
- [Aud+91] N. Audsley, A. Burns, M. Richardson und A. Wellings. „Hard Real-Time Scheduling: The Deadline-Monotonic Approach“. In: *IFAC Proceedings Volumes* 24.2 (Mai 1991), S. 127–132. DOI: 10.1016/s1474-6670(17)51283-5.
- [Aud+93] N. Audsley, A. Burns, M. Richardson, K. Tindell und A. Wellings. „Applying new scheduling theory to static priority pre-emptive scheduling“. In: *Software Engineering Journal* 8.5 (1993), S. 284. DOI: 10.1049/sej.1993.0034.
- [Aut19a] Autonomio. TALOS. [Online; Zugriff am 16.04.2019]. 2019. URL: [https://autonomio.github.io/docs\\_talos/#introduction](https://autonomio.github.io/docs_talos/#introduction) (besucht am 16.04.2019).
- [Aut19b] Autonomio. talos - Hyperparameter Optimization for Keras. <https://github.com/autonomio/talos>. [Online; Zugriff am 03.04.2019]. 2019.
- [Ayt+94] H. Aytug, S. Bhattacharyya, G. Koehler und J. Snowdon. „A review of machine learning in scheduling“. In: *IEEE Transactions on Engineering Management* 41.2 (Mai 1994), S. 165–171. DOI: 10.1109/17.293383.
- [BB01] E. Bini und G. Buttazzo. „The space of rate monotonic schedulability“. In: *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002*. IEEE Comput. Soc, 2001. DOI: 10.1109/real.2002.1181572.
- [BB04] E. Bini und G. Buttazzo. „Schedulability analysis of periodic fixed priority systems“. In: *IEEE Transactions on Computers* 53.11 (Nov. 2004), S. 1462–1473. DOI: 10.1109/tc.2004.103.
- [BBB01] E. Bini, G. Buttazzo und G. Buttazzo. „A hyperbolic bound for the rate monotonic algorithm“. In: *Proceedings 13th Euromicro Conference on Real-Time Systems*. IEEE, 2001. DOI: 10.1109/emrts.2001.934000.
- [BBB03] E. Bini, G. Buttazzo und G. Buttazzo. „Rate monotonic analysis: the hyperbolic bound“. In: *IEEE Transactions on Computers* 52.7 (Juli 2003), S. 933–942. DOI: 10.1109/tc.2003.1214341.
- [BL11] C. Bartolini und G. Lipari. RTSIM. <http://rtsim.sssup.it/>. [Online; Zugriff am 29.11.2018]. 2011.

- [BMR90] S. Baruah, A. Mok und L. Rosier. „Preemptively scheduling hard-real-time sporadic tasks on one processor“. In: *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE, 1990. DOI: 10.1109/real.1990.128746.
- [BRH90] S. K. Baruah, L. E. Rosier und R. R. Howell. „Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor“. In: *Real-Time Systems 2.4* (Nov. 1990), S. 301–324. DOI: 10.1007/bf01995675.
- [But11] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Springer US, 2011. DOI: 10.1007/978-1-4614-0676-1.
- [BW96] A. Burns und A. Welling. *Real-Time Systems and Their Programming Languages (International Computer Science Series)*. Addison-Wesley, 1996. ISBN: 020140365X.
- [Cha+12] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet und M. Qamhi. „YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms“. In: *WATERS 2012*. Juli 2012.
- [Che18] M. Cheramy. *SimSo - Simulation of Multiprocessor Scheduling with Overheads*. <http://projects.laas.fr/simso/>. [Online; Zugriff am 27.11.2018]. 2018.
- [Cho+15] F. Chollet u. a. *Keras*. <https://keras.io>. [Online; Zugriff am 26.03.2019]. 2015.
- [Cot+02] F. Cottet, J. Delacroix, C. Kaiser und Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, 2002. ISBN: 0-470-84766-2.
- [Dav+98] R. Davoli, F. Tamburini, L.-A. Giachiniand und F. Fiumana. „Schedulability Checking in Real-Time Systems using Neural Networks“. In: Apr. 1998.
- [DBC07] A. Diaz, R. Batista und O. Castro. „Realtss: a real-time scheduling simulator“. In: *2007 4th International Conference on Electrical and Electronics Engineering*. IEEE, Sep. 2007. DOI: 10.1109/iceee.2007.4344998.
- [Die+17] J. Diemer, P. Axer, D. Thiele und J. Schlatow. *pyCPA*. <https://pycpa.readthedocs.io/en/latest/>. [Online; Zugriff am 29.11.2018]. 2017.
- [Dvo18] J. Dvořák. *TORSCHÉ Scheduling Toolbox for Matlab*. <http://rttime.felk.cvut.cz/scheduling-toolbox/>. [Online; Zugriff am 28.11.2018]. 2018.
- [Eic+18] C. Eichler, T. Distler, P. Ulbrich, P. Wägemann und W. Schröder-Preikschat. „TASKers: A Whole-System Generator for Benchmarking Real-Time-System Analyses“. In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Hrsg. von F. Brandner. Bd. 63. OpenAccess Series in Informatics (OASISs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 6:1–6:12. ISBN: 978-3-95977-073-6. DOI: 10.4230/OASISs.WCET.2018.6.

- 
- [EKB15] S. Eckl, D. Krefft und U. Baumgarten. „COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions“. In: *Conference on Future Automotive Technology*. 2015.
- [Fis18] J. Fischer. „Extension of a Toolchain for Schedulability Analysis by a Task Generator Component for Evaluation of Different Real-Time Task Models“. Bachelorarbeit. Technische Universität München, 2018.
- [GB16] Z. Guo und S. K. Baruah. „A Neurodynamic Approach for Real-Time Scheduling via Maximizing Piecewise Linear Utility“. In: *IEEE Transactions on Neural Networks and Learning Systems* 27.2 (Feb. 2016), S. 238–248. doi: 10.1109/tnnls.2015.2466612.
- [Gér18] A. Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow*. O'Reilly, 5. Jan. 2018. ISBN: 3960090617.
- [Gol+02] F. Golasowski, J. Hildebrandt, J. Blumenthal und D. Timmermann. „Framework for validation, test and analysis of real-time scheduling algorithms and scheduler implementations“. In: *Proceedings 13th IEEE International Workshop on Rapid System Prototyping*. IEEE Comput. Soc, 2002. doi: 10.1109/iwrsp.2002.1029750.
- [GP17] A. Gibson und J. Patterson. *Deep Learning*. O'Reilly UK Ltd., 11. Aug. 2017. ISBN: 1491914254.
- [Grö15] S. Grösbrink. „Adaptive virtual machine scheduling and migration for embedded real-time systems“. Diss. Universität Paderborn, 2015.
- [Gub16] G. Guba. „Port and Extension of a Toolchain Regarding Machine Learning Supported Schedulability Analysis in Distributed Embedded Real-Time Systems“. Masterarbeit. Technische Universität München, 2016.
- [Häc18] R. Häcker. „Shallow Learning basierte Schedulability Analyse für die Migration von Softwarekomponenten zur Laufzeit“. Masterarbeit. Technische Universität München, 2018.
- [Ham17] R. Hamsch. „Extension of a Machine Learning Based Toolchain for Migration Planning by a Deep Learning Component“. Bachelorarbeit. Technische Universität München, 2017.
- [Har18] M. G. Harbour. *MAST - Modeling and Analysis Suite for Real-Time Applications*. <https://mast.unican.es/>. [Online; Zugriff am 26.11.2018]. 2018.
- [HK18] J. Hurwitz und D. Kirsch. *Machine Learning for dummies*. IBM Limited Edition. <https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=IMM14209USEN>. [Online; Zugriff am 05.10.2018]. John Wiley & Sons, Inc., 2018. ISBN: 978-1-119-45495-3.

- [Hoa+06] H. Hoang, G. Buttazzo, M. Jonsson und S. Karlsson. „Computing the Minimum EDF Feasible Deadline in Periodic Systems“. In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. IEEE, 2006. DOI: 10.1109/rtcsa.2006.22.
- [HRL18] T. Hope, Y. S. Resheff und I. Lieder. *Einführung in TensorFlow*. Dpunkt.Verlag GmbH, 1. Mai 2018. ISBN: 3960090749.
- [HSW17] A. Herkersdorf, W. Stechele und T. Wild. „HW/SW-Codesign“. Vorlesungsskript (SS 2017). Technische Universität München. 2017.
- [JP86] M. Joseph und P. Pandya. „Finding Response Times in a Real-Time System“. In: *The Computer Journal* 29.5 (Mai 1986), S. 390–395. DOI: 10.1093/comjnl/29.5.390.
- [Kes+16] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy und P. T. P. Tang. „On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima“. In: (15. Sep. 2016). arXiv: <http://arxiv.org/abs/1609.04836v2> [cs.LG].
- [Kin94] W. Kinnebrock. *Neuronale Netze. Grundlagen, Anwendungen, Beispiele*. Oldenbourg, 1994. ISBN: 3486229478.
- [Klu16] F. Kluge. *tms-sim – Timing Models Scheduling Simulation Framework - Release 2016-07*. Techn. Ber. Universität Augsburg, Juli 2016. DOI: 10.13140/RG.2.1.2544.0246.
- [Kra09] J. Kraft. *RTSSim - A Simulation Framework for Complex Embedded Systems*. Techn. Ber. Mälardalen University Sweden, März 2009.
- [Lab19] G. Labs. *About Genode*. <https://genode.org/about/index>. [Online; Zugriff am 22.02.2019]. 2019.
- [Leh90] J. Lehoczky. „Fixed priority scheduling of periodic task sets with arbitrary deadlines“. In: *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE, 1990. DOI: 10.1109/real.1990.128748.
- [Liu00] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN: 0130996513.
- [LL73] C. L. Liu und J. W. Layland. „Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment“. In: *Journal of the ACM* 20.1 (Jan. 1973), S. 46–61.
- [LM80] J. Y.-T. Leung und M. Merrill. „A note on preemptive scheduling of periodic, real-time tasks“. In: *Information Processing Letters* 11.3 (Nov. 1980), S. 115–118. DOI: 10.1016/0020-0190(80)90123-4.
- [LSD89] J. Lehoczky, L. Sha und Y. Ding. „The rate monotonic scheduling algorithm: exact characterization and average case behavior“. In: *[1989] Proceedings. Real-Time Systems Symposium*. IEEE Comput. Soc. Press, 1989. DOI: 10.1109/real.1989.63567.

- 
- [Nem+17] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal und A. Cristal. „A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs“. In: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, Okt. 2017. DOI: 10.1109/sbac-pad.2017.23.
- [NZ18] C. N. Nguyen und O. Zeigermann. *Machine Learning - kurz & gut*. Dpunkt.Verlag GmbH, 25. Apr. 2018. 183 S. ISBN: 3960090528.
- [Rau18] H. Rauer. „Recurrent Neural Network Based Schedulability Analysis for Migration of Software Components at Runtime“. Bachelorarbeit. Technische Universität München, 2018.
- [RCM96] I. Ripoll, A. Crespo und A. K. Mok. „Improvement in feasibility testing for real-time tasks“. In: *Real-Time Systems* 11.1 (Juli 1996), S. 19–39. DOI: 10.1007/bf00365519.
- [RM17] S. Raschka und V. Mirjalili. *Machine Learning mit Python und Scikit-Learn und TensorFlow*. MITP Verlags GmbH, 22. Dez. 2017. 584 S. ISBN: 3958457339.
- [Say+17] H. Sayadi, N. Patel, A. Sasan und H. Homayoun. „Machine Learning-Based Approaches for Energy-Efficiency Prediction and Scheduling in Composite Cores Architectures“. In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, Nov. 2017. DOI: 10.1109/iccd.2017.28.
- [Sca18] M. Scarpino. *TensorFlow für Dummies*. Wiley VCH Verlag GmbH, 12. Sep. 2018. 320 S. ISBN: 3527715479.
- [Sin18] F. Singhoff. *Cheddar: an open-source real-time scheduling tool/simulator*. <http://beru.univ-brest.fr/~singhoff/cheddar/>. [Online; Zugriff am 26.11.2018]. 2018.
- [Spu96] M. Spuri. *Analysis of Deadline Scheduled Real-Time Systems*. Techn. Ber. 1996.
- [Tan18] S. Tanwar. *How to get a grip on Cross Validations*. <https://medium.freecodecamp.org/how-to-get-a-grip-on-cross-validations-bb0ba779e21c>. [Online; Zugriff am 05.02.2019]. Dez. 2018.
- [Wik18] Wikipedia. *Beurteilung eines binären Klassifikators* — *Wikipedia, Die freie Enzyklopädie*. [https://de.wikipedia.org/wiki/Beurteilung\\_eines\\_binären\\_Klassifikators](https://de.wikipedia.org/wiki/Beurteilung_eines_binären_Klassifikators). [Online; Zugriff am 26.03.2019]. 2018.
- [Wik19] Wikipedia. *Korrelation* — *Wikipedia, Die freie Enzyklopädie*. <https://de.wikipedia.org/w/index.php?title=Korrelation&oldid=185998179>. [Online; Zugriff am 29. April 2019]. 2019.
- [ZB09] F. Zhang und A. Burns. „Schedulability Analysis for Real-Time Systems with EDF Scheduling“. In: *IEEE Transactions on Computers* 58.9 (Sep. 2009), S. 1250–1258. DOI: 10.1109/tc.2009.58.
-

- [ZD95] W. Zhang und T. G. Dietterich. „A Reinforcement Learning Approach to job-shop Scheduling“. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. 1995.
- [Zöb08] D. Zöbel. *Echtzeitsysteme*. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-76396-3.