Hochschule für Technik, Wirtschaft und Kultur Leipzig

Fakultät Informatik und Medien

Bachelorstudiengang Informatik

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Sturzvorhersage bei humanoiden Robotern mithilfe von Machine Learning

Eingereicht von: Max Liebing

Matrikelnummer: 73972

Leipzig 20. September 2024

Erstprüferin: Dr. Sibylle Schwarz

Zweitprüfer: Tobias Kalbitz

1 Abstrakt

Um Roboter erfolgreich Fußballspielen zu lassen, ist es essenziell, dass sie stabil laufen können ohne zu Stürzen. Beim Spielen ist es aber unvermeidlich, dass ein Roboter auch mal hinfällt. Eine maschinelle Vorhersage so eines Sturzes kann das Laufen somit deutlich verbessern. In dieser Arbeit werden die Machine Learning Modelle Multilayer Perceptron, Convolutional Netze und Rekurrente Netze mit Long Short-Term Memory verwendet um eine Sturzvorhersage zu implementieren. Jedoch gibt es die zusätzliche Herausforderung, die Modelle klein zu halten, um Rechenaufwand zu sparen. Dies stellt sich als besondere Hürde heraus, welche es schwer macht befriedigende Ergebnisse zu erzielen.

Inhaltsverzeichnis

2 Einleitung und Intension

2.1 Motivation

Eine der größten Herausforderungen bei der Entwicklung humanoider Roboter ist eine stabile Laufbewegung. Bei den Wettbewerben des Robocups (siehe Kapitel 3.1) lässt man solche Roboter in Fußballturnieren gegeneinander antreten. Das stabile Laufen ist eine spielentscheidende Grundfähigkeiten und am besten ist es, wenn der Roboter so wenig wie möglich fällt, denn wenn der Roboter stürzt, ist dies in vielerlei Hinsicht schlecht. Zum einen erleidet man erheblichen Zeitverlust beim Bewegen über das Spielfeld. Dadurch haben Gegner bessere Chancen den Ball oder Zweikämpfe zu gewinnen. Zum anderen sind jegliche Stürze schlecht für die Gelenke, da diese dadurch schneller erhitzen und gegebenenfalls im Laufe des Spiels dann weniger gut gesteuert werden können oder gar ganz ausfallen. Ebenfalls schonen weniger Stürze die Hardware vor zusätzlichem Verschleiß.

2.2 Ziel

Mit dieser Arbeit sollen verschiedene Machine Learning Modelle genutzt werden, um eine Echtzeit-Sturzvorhersage in die NAO-Software des Roboterfußballteams HTWK Robots zu implementieren. Dafür sollen als Eingangsdaten ausschließlich Informationen aus der *Inertial Measurement Unit* (Abk. **IMU**) genutzt werden.

Da der NAO leistungstechnisch eher schwach ausgestattet ist und bereits rechenintensive künstliche Netze (für z. B. Lokalisieren) auf dem Roboter laufen, soll

darauf geachtet werden, ein möglichst schlankes Modell zu entwickeln, um eine Echtzeitberechnung zu ermöglichen.

Am Ende muss dann evaluiert werden, wie gut die entwickelten Modelle einen Sturz vorhersagen und wie groß der benötigte Rechenaufwand ist, um sich dann für die beste Lösung zu entscheiden. Diese wird dann Grundlage für weitere Untersuchungen und für weiterführende Projekte, wie z. B. ein Präventivverhalten bei vorhergesagten Sturz in Form von Ausfallschritten oder der Vorbereitung eines gelenkschonenden Sturzes. In dieser Bachelorarbeit ist die Entwicklung eines reaktiven Verhaltens nicht vorgesehen.

2.3 Aufbau

Mit dieser Arbeit werden Machine Learning Modelle entwickelt, die eine zeitliche Echtzeit-Sturzvorhersage auf der NAO-Plattform ermöglichen. Dazu werden als Erstes die Hintergründe erläutert und danach die verwendeten Machine Learning Modelle vorgestellt und kurz erklärt. Daraufhin folgt die Darstellung des Entwurfs, in dem konzeptionell das geplante praktische Vorgehen aufgezeigt wird. Dabei werden die Grenzen dieser Arbeit sowie die Qualitätskriterien hervorgehoben. Ebenfalls wird gezeigt und begründet, welche Modelle Teil der Untersuchung sind und welche Kriterien für die Evaluation entscheidend sind. In der Evaluation werden dann die entwickelten Modelle nach vorher aufgestellten Kriterien verglichen und eine Entscheidung getroffen, welche Modelle als Lösung für das Problem geeignet sind. Zum Schluss folgt eine Zusammenfassung der Arbeit und ein Ausblick für mögliche darauf aufbauende Projekte.

3 Hintergrund

3.1 Ziele und Motivation des Robocups

Die Intention des RoboCups besteht darin, die Forschung von Robotik und Künstlicher Intelligenz voranzutreiben, indem eine ansprechende Herausforderung präsentiert wird, die das Interesse der Öffentlichkeit weckt [Rob]. Ähnlich wie mit Deep Blue, dem schachspielenden Roboter, der im Jahre 1997 den amtierenden Meister schlug, hat man den Traum, Mitte des 21. Jahrhunderts eine menschliche Fußballmannschaft mit einer reinen autonomen Robotermannschaft nach Fifa-Regeln zu besiegen.

Eine der Ligen, von den verschiedenen in welche der Robocup unterteilt ist, ist die Standard Platform League (Abk. **SPL**), in welcher auch das Team HTWK Robots spielt. Wie der Name es verrät, wird in dieser Liga auf einer Standartplattform gespielt, welche von den Teams nicht verändert werden darf. Das hat zur Folge, dass alle Teams mit den gleichen Robotern spielen und man lediglich die Software programmieren darf [SPL]. Das wiederum bedeutet auch, dass die SPL im Besonderen ausschließlich ein Programmierwettbewerb ist. Zum Stand dieser Arbeit wird der NAO als Plattform verwendet.

Abbildung 3.1: Der NAO

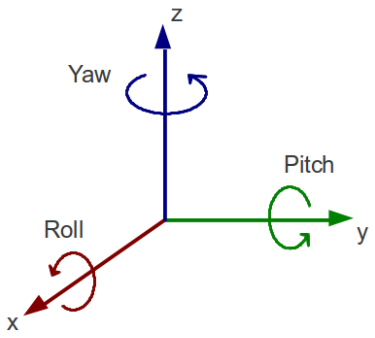
Quelle: http://doc.aldebaran.com/2-8/family/nao_technical/index_naov6.html

3.2 Die Plattform

3.2.1 Der NAO

Der NAO ist ein etwa halben Meter großer humanoider Roboter mit einem Gewicht von etwas mehr als fünf Kilogramm (siehe Abb. 3.1) von dem Unternehmen Softbanks, früher Aldebaran genannt. Er ist lediglich mit einer 1.6 GHz CPU und 1 GB RAM ausgestattet, wodurch er kaum mit modernen Smartphones mithalten kann. Des Weiteren verfügt er neben Kameras, Lautsprechern und Mikrophonen auch über eine, für diese Arbeit besonders wichtige, IMU [Naob] (z. dt. Inertiale Messeinheit/Trägheitssensoren).

Abbildung 3.2: Achsenbezeichnung beim Gyro

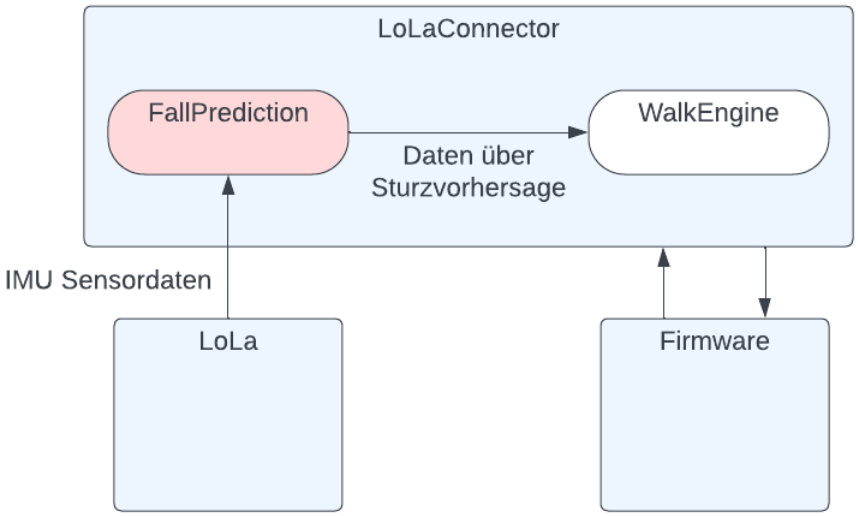


Quelle: http://doc.aldebaran.com/2-1/family/robots/joints_robot.html

3.2.2 Inertial Measurement Unit

Die meisten digitalen Geräte und intelligenten Fahrzeugen sind mit Trägheitssensoren ausgestattet. Damit erleichtert man Lokalisierung und Navigation. Eine IMU misst Winkelgeschwindigkeit und Beschleunigung. Es gibt auch IMUs, die mit einem Magnetometer, Magnetfeldvektoren messen [SKM20]. Die gemessenen Werte, werden genutzt, um geänderte Position, Richtung und Orientierung eines Systems, welches die IMU montiert hat, zu berechnen [ESY20].
Die IMU des NAO V5 ist mit einem Drei-Achsen-Gyroskop (Abk. **Gyro**) und einem Drei-Achsen-Beschleunigungsmesser, oder auch Accelerometer genannt, ausgestattet. Die Einheit befindet sich im Torso des Roboters. Die Ausgangsdaten sind jeweils die Winkelgeschwindigkeit vom Gyro und lineare Beschleunigung vom Accelerometer entlang der X, Y und Z Achsen, die beim Gyro auch als *Yaw*, *Pitch* und *Roll* bezeichnet werden [Naoa] (siehe Abb. 3.2). Mithilfe der IMU Werte lassen sich dann unter anderem auch die Torsoneigung berechnen, welche für die Firmware entscheidend darüber ist, ob der Roboter im aktuellen Moment als fallend erkannt wird.

Abbildung 3.3: Eingliederung in die Firmwarestruktur



3.3 Eingliederung in die Firmware von HTWK-Robots

Das mögliche Endprodukt der Sturzvorhersage soll als neuen Bestandteil in den LolaConnector integriert werden, der vom HTWK-Robots Team erstellt wurde und als Schnittstelle zwischen der Firmware und Lola fungiert. Innerhalb der Firmware wird Verhaltens- und Strategielogik implementiert sowie Sensordatenverarbeitung und Highlevel-Bewegungsgenerierung. Lola wiederum ist ein vom Hersteller bereitgestelltes Interface, das kontinuierlich Sensordaten alle 12 Millisekunden sendet. Innerhalb des LolaConnectors ist die WalkEngine ein wesentlicher Bestandteil, der das Laufverhalten steuert. Die WalkEngine wird als Hauptkonsument der Fallvorhersagedaten fungieren, was es ermöglicht, reaktives Verhalten basierend auf den Prognosen einzuleiten. Durch diese Integration wird die Sturzvorhersage nahtlos in die bestehende Firmwarestruktur eingegliedert und unterstützt so die Funktionalität des Roboters im Umgang mit potenziellen Sturzsituationen. Eine Skizze, die die betroffenen Bestandteile mit der neuen Ergänzung durch eine Sturzvorhersage zeigt, ist in Abbildung 3.3 zu sehen.

4 Machine Learning Methoden

4.1 Multilayer Perceptron

Für die Erläuterung von Multilayer Perceptron beziehe ich mich auf Kapitel 5 des Buches *Pattern Recognition and Machine Learning* von Christopher M. Bishop [Bis06, S.225ff].

Das klassische Multilayer Perceptron (Abk. **MLP**) ist ein sogenanntes Feed-Forward Netz. Das bedeutet, dass die Informationen nur in eine Richtung fließen. Ein standardmäßiges MLP besteht aus einer Reihe von funktionalen Transformationen, die auch als Schichten bezeichnet werden. Im einfachsten Fall gibt es jeweils eine Schicht für Ein- und Ausgabe, sowie eine versteckte Schicht. Für jeden Eingabewert gibt es zusätzlich eine Gewichtung (Weight) und einen Bias mit denen der Wert der Aktivierung berechnet wird. Die Summe aller Aktivierungen werden durch eine Aktivierungsfunktion der Knoten in den versteckten Schichten transformiert. Übliche Aktivierungsfunktionen sind die Sigmoid oder Hyperbeltangensfunktion, auch werden Rectifier vor allem in tiefen Netzen verwendet [GBB11]. Die daraus entstehenden Werte werden wieder linear mit Gewichtung und Bias kombiniert und ergeben damit die Eingangswerte (Aktivierungen) der nächsten Schicht. Die Parameter Gewichtungen und Bias werden durch Rückwärtspropagierung während eines Trainingsprozesses angepasst. Dabei wird die Güte eines Modells anhand einer Verlustfunktion ermittelt. Das Ziel des Trainingsprozesses ist es, den Ergebniswert der Verlustfunktion zu minimieren.

Die Hyperparameter eines Modells sind jene Parameter, welche beim Trainingsprozess unverändert bleiben, also vorher fest definiert werden. Im Fall des MLP sind

diese u. a. Anzahl der Schichten, Anzahl der Knoten einer Schicht oder Wahl der Aktivierungsfunktion einer Schicht. Auf die Optimierung dieser Hyperparameter und auch die der anderen Modelle wird in Kapitel 5.4 eingegangen.

Für die Ausgabeschicht werden oft spezifische Aktivierungsfunktionen verwendet: Bei binären Klassifikationsproblemen wird oft die Sigmoidfunktion verwendet, während für Multiklassen-Probleme die Softmaxfunktion (4.1) genutzt wird, welche im wesentlichen eine angepasste Version der Sigmoidfunktion für mehrere Klassen ist.
Auch bei anderen Modellen wird oft eine extra vollständig verknüpfte Schicht mit Softmax-Aktivierungsfunktion am Ende eingefügt, um eine Klassifizierung in mehrere Klassen zu ermöglichen. Im späteren Kapitel 5.4 ist zu sehen, dass alle in dieser Arbeit entworfenen Modelle solch eine extra Schicht bekommen.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{4.1}$$

Die Biase können für jede Schicht in einer einzigen Gewichtung in einem zusätzlichen Knoten zusammengefasst werden, sodass ein einfaches MLP beispielsweise, wie in Abbildung 4.1 zu sehen, aufgebaut ist.

4.2 Convolutional Netze

Für die Erläuterung von Convolutional Netze (eng. Convolutional Neural Networks, folglich **CNN** abgekürzt), beziehe ich mich auf die Arbeit von Le Cun 1989 [Lec89] und Le Cun et al. 1998 [LBBH98].

Die hier beschriebenen CNN sind eine spezielle Art von Feed-Forward-Netzen, die sich besonders für die Verarbeitung von strukturierten Daten wie Bildern und Zeitreihen eignen. Sie vereinen drei grundlegende Ideen, die ihnen ihre Effektivität verleihen: lokale rezeptive Felder, geteilte Gewichtungen und räumliches oder zeitliches Subsampling.

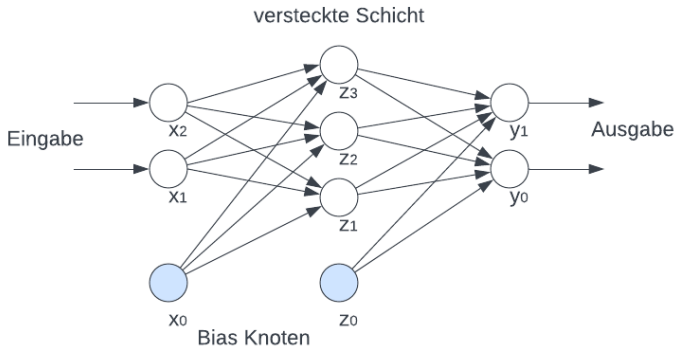


Abbildung 4.1: Grundsätzlicher Aufbau eines MLP, mit nur einer versteckten Schicht. Es sind mehr Schichten und pro Schicht auch mehr Knoten möglich. Jeder Knoten einer Schicht ist mit allen Knoten der nächsten Schicht verbunden. Jeder Knoten nimmt die Summe seiner Aktivierungen und berechnet mit seiner Aktivierungsfunktion seinen Ausgangswert für die nächste Schicht. Die Biase der linearen Transformationen einer Schicht (Pfeile zwischen Knoten) können als extra Knoten (hellblaue Knoten) in den jeweiligen Schichten zusammengefasst werden, um Rechenaufwand zu sparen.

Sie bestehen aus Faltungsschichten (Convolutional Schichten), welche wiederum sich aus mehreren sogenannten Feature Maps zusammensetzen. Jede dieser Feature Maps bezieht ihre Eingabewerte aus einer relativ schmalen Umgebung der vorherigen Schicht. Das Ziel ist es, verschiedene Merkmale oder Eigenschaften aus den Eingabedaten zu extrahieren. Alle Eingabewerte innerhalb dieser Umgebung haben die gleichen Gewichte und Bias, die als Kernel bezeichnet werden. Der Kernel wird über die Eingabedaten geschoben, um die Convolution durchzuführen.

Die genaue Position der extrahierten Eigenschaften ist oft nicht von großer Bedeutung. Aus diesem Grund wird die Präzision reduziert, indem die räumliche Auflösung durch eine Subsampling-Schicht verringert wird. Diese Schicht berechnet einen lokalen Mittelwert. Dadurch werden die Dimensionen der Feature Map reduziert und die Anfälligkeit für Verschiebungen und Verzerrungen verringert. Es gibt auch das Verfahren bei dem das lokale Maximum genommen wird. Diese Vorgehen werden auch Pooling genannt [RP99].

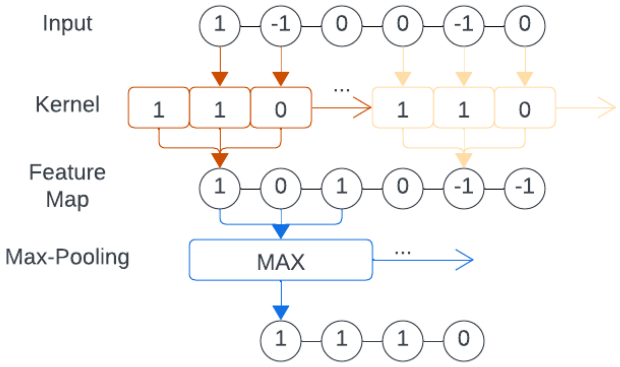


Abbildung 4.2: Beispiel wie ein CNN bei einer Zeitreihe funktioniert. Als Eingabe hat man die Zeitreihe. Durch das Schieben des Kernels und mithilfe seiner Gewichtungen wird eine Feature Maps des Convolutional Layers erstellt. Die Anzahl der Feature Maps kann beliebig sein. Im Subsampling Layer, hier ein Max-Pooling, wird die Länge der Reihe von sechs auf vier reduziert indem lokal immer das Maximum berechnet wird.

Die Hyperparameter eines CNN, welche in dieser Arbeit relevant bei der Optimierung sind (siehe Kapitel 5.4), sind die Anzahl der Convolutional-Schichten, die Anzahl der zu erstellenden Feature Maps in einer Schicht, die Größe des Kernels und die Wahl zwischen Maximum oder Mittelwert in den Subsampling-Schichten.

Ein Beispiel, wie ein CNN bei Zeitreihen funktioniert ist in Abbildung 4.2 zu sehen.

In dieser Arbeit werden CNNs nur mit Convolutional- und Subsampling-Schichten verwendet. Daher werden diese Modelle später in Kapitel 5.4 Fully-Convolutional Netze (Abk. **FCN**) genannt.

4.3 Rekurrente Netze

Anders als bei Feed-Forward Netzen fließen die Informationen bei rekurrenten Netzen (engl. Recurrent Neural Networks, folglich **RNN** abgekürzt) nicht nur in eine Richtung, sondern werden auch innerhalb einer Schicht von Knoten zu Knoten weitergeleitet. Durch ihren Aufbau besitzen sie eine Art Kurzzeitgedächtnis und

sind daher besonders gut geeignet für unter anderem Spracherkennung [GMH13], Textgenerierung [SMH11] oder auch Zeitreihenvorhersagen [HS03]. Ein Beispiel wie RNNs grundlegend aufgebaut sind, ist unter Abb. 4.3 zu sehen.
Oft werden RNNs auch nach der Anzahl ihrer Ein- und Ausgabewerten unterteilt [AA]. So gibt es u. a. Netze, die mit einer Eingabe genau eine Ausgabe erzeugen (sog. One-To-One Netze) oder, wie in dieser Arbeit zum Einsatz kommen wird, mehrere Eingabeschritte erzeugen einen Output (sog. Many-To-One Netze, siehe Abb. 4.4).
Weiterentwicklungen von RNNs mit besonderen Speicherzellen, die ein längeres Kurzzeitgedächtnis durch Gatterarchitekturen ermöglichen, sind Long Short-Term Memory (Abk. **LSTM**) von Hochreiter und Schmidhuber 1997 [HS97] oder Gated Recurrent Units (Abk. **GRU**) von Chan et al. 2014 [CMG+14]. In dieser Arbeit werden LSTM als Many-To-One verwendet (siehe Kapitel 5.4).

Der einzige Hyperparameter eines RNN, welcher in dieser Arbeit relevant bei der Optimierung ist (siehe Kapitel 5.4), ist lediglich die Anzahl der Speicherzellen in der LSTM-Schicht.

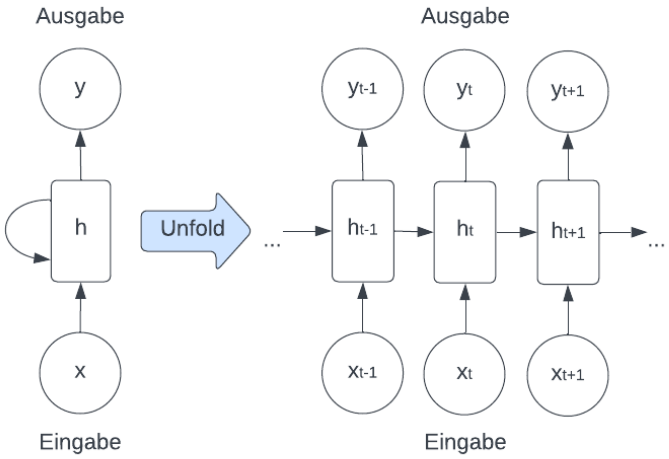


Abbildung 4.3: Grundlegender Aufbau eines RNN. Der versteckte Zustand h der Speicherzelle wird auch als zusätzlicher Eingabewert zur Berechnung im nächsten Schritt genutzt. Die Darstellung auf der rechten Seite zeigt den zeitlichen Verlauf der Eingabewerte, d. h. jedes Segment mit Eingabe, RNN-Schicht und Ausgabe (Kreis-Viereck-Kreis) stellt die selbe RNN Struktur wie auf der linken Seite dar.

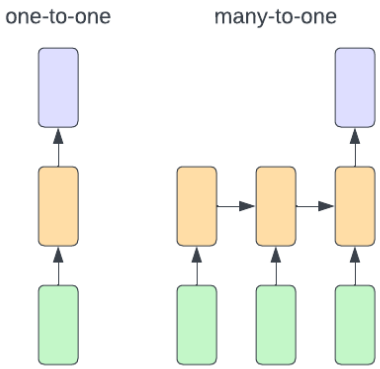


Abbildung 4.4: Bei einem One-To-One RNN Aufbau (links) wird nach einer Eingabe (grün) ein Ausgabewert (blau) berechnet. Bei einem Many-To-One RNN Aufbau (rechts) werden nach einer beliebigen Anzahl von Eingaben ein Ausgabewert berechnet. Auch hier sind die orangenen Vierecke nicht als verschiedene Schichten zu verstehen, sondern immer als die gleiche Schicht nur im zeitlichen Verlauf betrachtet. Weitere Formen sind One-To-Many und Many-To-May.

5 Entwurf

Der für die Erstellung dieser Arbeit entstandene Programmcode wird separat mitgeliefert bzw. ist unter dem Github-Repository https://github.com/mliebg/FallPrediction einsehbar.

5.1 Abgrenzung und Qualitätskriterien

Im Rahmen dieser Arbeit werden ausschließlich Machine Learning Modelle entwickelt, die eine Sturzvorhersage in Echteit für den NAO implementieren. Dem eingeschlossen sind die in diesem Kapitel weiteren Vorgehensweisen der folgenden Unterkapitel. Die Weiterverarbeitung der Ausgabewerte der Modelle für zum Beispiel einen Ausfallschritt ist nicht Teil dieser Arbeit.

Um zu bewerten wie erfolgreich die Arbeit ist, werden folgende Qualitätskriterien aufgestellt:

Die zu entwickelnden Modelle müssen die Vorhersage in Echtzeit erbringen, dafür wird festgelegt, dass die Modelle eine Obergrenze von 500.000 Floating Point Operations (Abk. **FLOPs**, zu dt. Gleitkommazahloperationen) nicht überschreiten dürfen. Umso weniger FLOPs benötigt werden, desto besser ist ein Modell zu bewerten. Ein Modell gilt als geeignet, wenn es zuverlässig einen Sturz vorhersagen kann. Eine korrekte Vorhersage für einen Sturz muss in mindestens 9 von 10 Fällen getroffen werden. Gleichzeitig darf es aber auch nicht zu viele Stürze fälschlicherweise vorhersagen, da eine Sturzvorhersage in irgendeiner Weise einen reaktiven Aufwand nach sich zieht und dies in nicht notwendigen Szenarien weitestgehend

vermieden werden soll. Das Modell soll nicht mehr als 10 % der Zeit, in der eigentlich stabiles Laufen antizipiert werden muss, eine falsche Vorhersage treffen.
Wird ein Modell entwickelt, welches den genannten Anforderungen entspricht, gilt die Arbeit als erfolgreich. Wenn mit dem verwendeten Entwurf, neue Erkenntnisse über mögliche Verbesserungen in weiterführenden Entwicklungen erlangt werden, ist dies als zusätzlicher Erfolg zu bewerten. Sollte sich herausstellen, dass der verwendete Entwurf ungeeignet ist und damit herausgefunden wird, dass andere Ansätze zu untersuchen sind, ist dies als Teilerfolg einzuschätzen.

5.2 Datenerfassung

Die Quelldaten für die Entwicklung der Modelle stammen aus bereits erfassten IMU-Messwerten des NAOs. Es handelt sich um Logdaten aus dem Finalspiel des Robocups in Bordeaux 2023 gegen das Team B-Human. Diese Logs stammen von den sechs Feldspieler-Robotern, die in beiden Halbzeiten spielten, welche jeweils zehn Minuten dauern. Die Logs werden vom HTWK-Robots Team zur Verfügung gestellt und sollen als einzige Datengrundlage für die Entwicklung der Modelle genutzt werden. Die Logs des Torwarts werden nicht berücksichtigt, da im Laufe des Spiels dieser mehrfach nach dem Ball gesprungen ist, um ihn zu verteidigen und die Daten der Sprünge nicht sehr einfach von tatsächlichen unbeabsichtigten Stürzen zu unterscheiden sind.

Da die Logs sämtliche Informationen enthalten, die der Roboter im Laufe des Spiels mitschreibt, müssen die Daten zunächst gefiltert werden. Der Code zum Filtern der Logs ist in der Quelldatei `DataAquisition/LogFiltering/extractImuData.py` enthalten. Für das Filtern der Logs wird allerdings das Python-Backend von HTWK Robots benötigt.
Beim Filtern werden die Sensordaten ausgelesen, in denen die Messwerte der IMU zu finden sind. Zu den IMU-Werten gibt es jeweils einen Zeitstempel. Um später nachvollziehen zu können, ob der Roboter zum Zeitpunkt des Zeitstempels fallend oder stehend ist, werden zusätzlich noch die Torsoneigungswinkel ausgelesen. Zu den Werten der IMU gehören neben der Winkelbeschleunigung entlang der drei

Raumachsen des Gyroskops, auch die Werte der lineare Beschleunigung ebenfalls entlang der drei Raumachsen des Accelerometers. Der Zeitstempel dient neben der Funktion als Identifikator, auch zur zeitlichen Ordnung der Messwerte. Die Torsoneigungen werden genutzt, um zu bestimmen, ob der Roboter steht oder fällt. Dafür wird festgelegt, dass Roboter als fallend bezeichnet wird, wenn einer der Torsoneigungswinkel 25 Grad überschreitet.

Das Labeln der Daten ist in Quelldatei unter `DataAquisition/LogLabeling/FullyLabeled/labelLogdata.py` einsehbar. Wenn einer der Torsoneigungswinkel größer als 25 Grad ist, wird der Roboter als fallend eingeschätzt. Diese Einschätzung geht aus der Firmware hervor. Anschließend wird mithilfe dieser Einschätzung und dem Zeitstempel das Delta berechnet, um für jeden Zeitstempel die Zeit bis zum nächsten Mal wenn der Roboter fällt zu bestimmen. Basierend auf diesen Deltas erfolgt eine Klassifizierung in die fünf Kategorien:

- 0 : läuft stabil, $\Delta t > 2s$
- 1 : $1s < \Delta t \leq 2s$
- 2 : $0.5s < \Delta t \leq 1s$
- 3 : $0.2s < \Delta t \leq 0.5s$
- 4 : $0s < \Delta t \leq 0.2s$

5.3 Datenaufbereitung

Die Eingabewerte werden für die Modelle standardisiert. Diese Art der Normalisierung ist üblich für Zeitreihen [AB17]. Die Normalisierung der Werte ist vorteilhaft für das Training der Modelle [SS20]. Die Standardisierung der Eingabewerte ist in der Quelldatei `DataPreprocessing/Z-Norm/znorming.py` implementiert. Die normalisierten Z-Werte werden mithilfe Erwartungswert μ und Varianz σ aller Werte folgendermaßen berechnet:

$$z = \frac{x - \mu}{\sigma} \tag{5.1}$$

Die restliche Datenaufbereitung, welche Reduzierung der Eingangsdaten sowie eine Aufteilung in Test-, Validierungs- und Testdatensätze umfasst, sind in der Quelldatei unter `DataPreprocessing/createModelData_ts.py` einsehbar.

Es ist noch zu beachten, dass Zeitstempel, bei denen der Roboter als fallend gilt, bei späteren Trainings- und Testdurchläufen der Modelle ausgeschlossen werden, da es nicht eine Sturzvorhersage benötigt, wenn der Roboter bereits am Boden liegt. Außerdem werden die Daten so weit reduziert, bis alle Klassen gleichermaßen vertreten sind.

Des Weiteren werden die Daten in Form von Zeitfenstern strukturiert. Dazu werden Zeitreihen mit jeweils 100 und 200 zurückliegenden Einträgen untersucht. Der Roboter erhält alle 12 Millisekunden neue Messwerte. Das bedeutet, dass Werte beachtet werden, die 1,2 bzw. 2,4 Sekunden zurückliegen.

Zum Trainieren der Modelle werden die Daten in Training-, Validierungs- und Testdatensätze unterteilt mit folgender Verteilung:

- 70 % Training
- 15 % Validierung
- 15 % Test

5.4 Modellauswahl und Implementierung

Die Implementierung erfolgt mithilfe von Python Tensorflow, da später auf dem Roboter eine Tensorflow Light Version laufen soll.

Für alle Modelle wird als Verlustfunktion die Cross Entropy (5.2) verwendet. Es wird auf *One-Hot-Kodierung* der Klassen verzichtet, da unter Tensorflow die Standardimplementierung Names *sparse categorical crossentropy* verwendet werden kann, die es ermöglicht mit Ganzzahlen als Klassenbezeichnungen zu arbeiten.

$$CE = -\sum_{i=1}^{C} y_i \log p_i \tag{5.2}$$

C : die Anzahl der Klassen in One-Hot-Kodierung

y_i : entspricht dem tatsächlichen Label als, $y_i \in 0, 1$

p_i : der entsprechende vorhergesagte Wert, $p_i \in [0; 1]$

Die wesentlichen Modelle, die untersucht werden sollen, wurden bereits unter Kapitel 4 vorgestellt. Es werden MLPs, FCNs und LSTMs für Zeitreihengrößen je 100 und 200 entwickelt. LSTMs werden als Many-To-One (vgl. Abb. 4.4) implementiert.
Die jeweiligen Hyperparamter werden mithilfe des Keras Tuner [OBL+19] ermittelt. Beim Optimieren wird in maximal einhundert Versuchen, zu je fünfzig Epochen ein Modell trainiert. Das Modell mit der niedrigsten CE über den Validierungsdaten wird als Modell mit den besten Hyperparametern bewertet. Die größtmögliche Auswahl an Hyperparametern ist so festgelegt, dass sie nicht 500.000 FLOPs überschreiten können. Dies wird experimentell durch Trial-and-Error mithilfe der Funktion `get_flops` aus `Models/utils.py` ermittelt. Diese Funktion entstammt dem Python-Backend von HTWK Robots. Für MLP und FCN wird sich zusätzlich auf maximal drei versteckte Schichten beschränkt.
Als letzte Schicht bekommt jedes Modell eine vollständig verknüpfte Schicht mit Softmaxaktivierungfunktion, für die Unterscheidung in die fünf Klassen. Die Architekturen der Gewinnermodelle sind unter Abb. 5.1, 5.2 und 5.3 zu sehen.

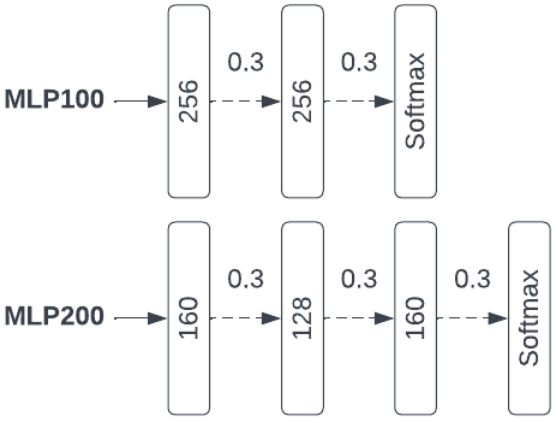


Abbildung 5.1: Aufbau der getesteten MLPs. Zahlen nach dem Modellname beziehen sich auf die Zeitreihenlänge der Eingabedaten. Jede Schicht ist vollständig mit der nächsten verbunden. Die Zahlen in den Schichten geben an, wie viele Knoten in der Schicht enthalten sind. Gestrichelte Linien markieren Dropouts mit der enstprechenden Droprate darüber. Dropouts helfen um gegen Overfitting vorzugehen [SWT18], indem während des Trainingsprozesses zufällig Knoten einer Schicht deaktiviert werden.

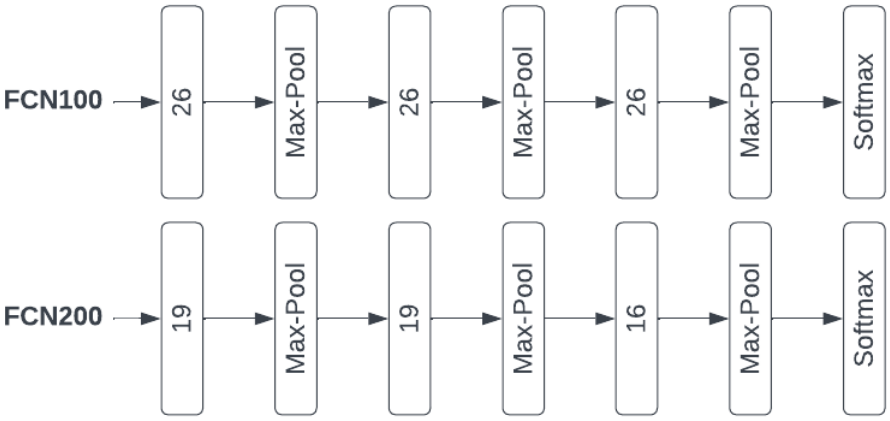


Abbildung 5.2: Zahlen nach dem Modellname beziehen sich auf die Zeitreihenlänge der Eingabedaten. In beiden getesteten FCN Modellen folgt auf jede Convolutional-Schicht eine Max-Pooling-Schicht. Die Zahlen der Schichten geben an, wie viele Feature-Maps pro Schicht erstellt werden.

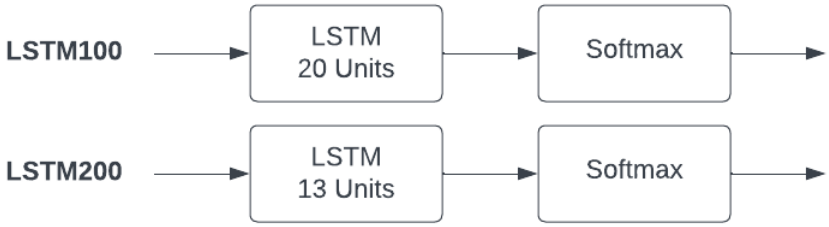


Abbildung 5.3: Die getesteten LSTM-Modelle bestehen lediglich aus einer Schicht mit verschiedener Anzahl an LSTM-Zellen. Zahlen nach Modellname beziehen sich auf die Zeitreihenlänge der Eingabedaten.

5.5 Evaluation

Die Implementierung der Evaluation ist in der Quelldatei `Models/eval.py` einsehbar.

Wesentliches Merkmal zum Vergleich aller Modelle soll die Genauigkeit auf dem Testdatensatz sein. Die Genauigkeit gibt an, wie viele korrekte Vorhersagen im Verhältnis zur Anzahl aller Vorhersagen eines Modells gemacht worden sind. Unter Tensorflow wird die *sparse categorical accuracy* (**SCA**) [tfS] hergenommen. Sie wird für ganzzahlig gelabelte Ausgabewerte genutzt, bei denen als Vorhersage die Klasse mit der höchsten Wahrscheinlichkeit gewählt wird. Der Wert der Genauigkeit ergibt sich dann einfach aus der Anzahl der richtigen Vorhersagen geteilt durch die Anzahl aller Vorhersagen.

Um einschätzen zu können, wie gut die Modelle einen Sturz vorhersagen, soll eine Falsch-Positiv-Rate (FPR_{stabil}, siehe (5.3)) für das Ereignis 'Roboter läuft stabil' ermittelt werden. Dazu werden die falsch positiven Vorhersagen für Klasse 0 (stabil laufen) gezählt und durch die Anzahl, der dafür tatsächlichen negativen Ereignissen, geteilt. Diese sind die Summe der jeweiligen Anzahl n_c der im Testdatensatz auftretenden Klassen 1 bis 4, jene die einen Sturz innerhalb einer Zeit Δt vorhersagen.

$$FPR_{stabil} = \frac{FP_0}{\sum_{c=1}^{4} n_c} \tag{5.3}$$

Zusätzlich wird die Falsch-Negativ-Rate (FNR_{stabil}, siehe (5.4)) für das Ereignis 'Roboter läuft stabil' berechnet, um einzuschätzen wie häufig Stürze fälschlicher Weise vorhergesagt werden. Dazu werden die falsch negativen Vorhersagen für Klasse 0 (stabil laufen) gezählt und durch die Anzahl, der dafür tatsächlichen positiven Ereignissen, geteilt. Diese entsprechen der Anzahl n_0 der im Testdatensatz auftretenden Klasse 0.

$$FNR_{stabil} = \frac{FN_0}{n_0} \tag{5.4}$$

Wie bei SCA wird auch bei der Berechnung von FPR_{stabil} und FNR_{stabil} immer die Klasse mit der höchsten vorhergesagten Wahrscheinlichkeit als Vorhersage angenommen. Die Funktion `get_fp_and_fn` aus `Models/eval.py` ermittelt unter anderem FP_0 und FN_0, sowie n_c und n_0.

Um einen Vergleich für den Rechenaufwand zu bekommen, werden benötigte FLOPs der Modelle mithilfe der Tensorflow API ermittelt und verglichen. Dazu wird wieder die Funktion `get_flops` aus Kapitel 5.4 verwendet

6 Evaluation

6.1 Vergleich der Modelle

In Tabelle 6.1 sind die Ergebnisse der Evaluation der Modelle zu sehen. Für die Unterteilung in Modelle mit verschiedenen Zeitreihenlängen als Eingabe sind die Modelle extra mit 100 bzw. 200 zusätzlich markiert und werden folglich, wie in Tabelle 6.1 zu sehen, bezeichnet.
Alle entwickelten Modelle benötigen etwa die gleiche Anzahl an FLOPs. Das MLP100 hat die geringsten FLOPs, aber auch FCN200 ist diesem sehr nah. Da aber kein Modell die gesetzte Obergrenze von 500.000 FLOPs überschritten hat (was durch Limitierung der Hyperparameteroptimierung gewährleistet wurde), sind alle Modelle bezüglich Rechenaufwand geeignet.
Keines der Modelle erreicht einmal eine Genauigkeit von 0,8. LSTM100 hat mit 0,7712 die höchste Genauigkeit. Dem dicht gefolgt ist MLP100 mit nur etwa einem halben Prozentpunkt weniger.
Die FPR aller Modelle liegt unter 0,1. Nur bei MLP200 liegt sie knapp darüber. Das lässt schließen, dass die Modelle immerhin einen Sturz zuverlässig vorhersagen, in mindestens einem von zehn Fällen. LSTM schneidet hierbei besonders gut ab, mit der geringsten FPR bei LSTM100 mit 0,0352.
Die FNR ist am geringsten bei LSTM200. Dieses Modell hat allerdings die mit Abstand schlechteste Genauigkeit. Alle anderen Modelle sagen etwa in einem von drei Ereignissen in denen eigentlich stabil gelaufen wird, einen Sturz voraus. Die nächst besten FNR haben die FCNs mit etwa 0,30 bis 0,31.
Generell ist zu erkennen, dass mehr Information durch längere Eingabezeitreihen bei gleichzeitigen Anpassen der Modelle bezüglich FLOPs die Genauigkeit der

Modell	SCA	FPR_{stabil}	FNR_{stabil}	FLOPs
MLP100	0.7645	0.0679	0.3301	**441374**
MLP200	0.7252	0.1006	0.3435	467998
FCN100	0.6967	0.0844	0.3093	477832
FCN200	0.7262	0.0649	0.3018	445811
LSTM100	**0.7712**	**0.0352**	0.3526	463000
LSTM200	0.5604	0.0415	**0.2697**	458400

TABELLE 6.1: Vergleich Modelle. Bei den LSTM ist zu beachten, dass hier die FLOPs angegeben sind, wenn die gesamte Eingabereihe durch das Modell geht (bei LSTM100: 100 mal 4630 FLOPs, LSTM200: 200 mal 2292 FLOPs).

Modelle nicht wesentlich verbessert. Im Fall von MLP und LSTM ist sogar eine Verschlechterung zu sehen. Bei LSTM ist sie besonders deutlich mit fast 0,20, was schließen lässt, dass bei LSTM die Anzahl der LSTM-Zellen viel entscheidender als die Zeitreihenlänge ist. Nur bei FCN sind Verbesserungen sichtbar, diese sind bei doppelter Eingabelänge jedoch nur einige Prozentpunkte.

6.2 Empfehlung eines Modells und Bewertung der Arbeit

Keines der Modelle erreicht eine zufriedenstellende Genauigkeit bei gleichzeitig niedrigen FLOPs. Die FPR zeigen zunächst annehmbare Ergebnisse, allerdings ist die FNR aller Modelle noch viel zu hoch. Damit konnte mit dem gewählten Entwurf eine Sturzvorhersage in Echtzeit nicht erfolgreich entwickelt werden. Der gewählte Entwurf ist damit ungeeignet, jedoch hat sich herausgestellt, dass längere Zeitreihen als Eingabe keine wesentlichen Verbesserungen hervorbringen. Daher ist es für weitere Untersuchungen sinnvoll, kürzere Zeitreihen als Eingabe zu testen. Mit dieser Erkenntnis und da LSTM die höchste Genauigkeit erzielt, ist demnach auch ein RNN mit One-To-One Architektur (vgl. Abb. 4.4) für weitere Untersuchungen interessant. Demnach ergibt sich nach den in Kapitel 5.1 aufgestellten Qualitätskriterien für diese Arbeit ein Teilerfolg. Des Weiteren ergibt sich ein zusätzlicher

Erfolg, indem gezeigt wurde, dass die Verwendung von kürzeren Zeitreihen sinnvoll ist.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Es wurden MLP, FCN und LSTM mit Zeitreihenlängen als Eingaben zu je 100 und 200 entwickelt, um eine Sturzvorhersage in Echtzeit auf dem NAO zu implementieren. Die zu verwendenden Daten sind die Messwerte der IMU des Roboters. Durch die Beschränkung der Modellgrößen, die den Rechenaufwand niedrig halten sollen, stellt sich die Problematik als eine besondere Herausforderung heraus.
Die in dieser Arbeit entwickelten Modelle sind nicht in der Lage, zufriedenstellende Ergebnisse zu liefern. Allerdings haben die Testergebnisse gezeigt, dass die größere Eingabelänge der Zeitreihe von 200 Zeitstempeln nicht eindeutig bessere Ergebnisse als mit 100 Zeitstempeln erzielt. Das heißt, es ist möglich, dass noch kürzere Zeitreihen als Eingabe sinnvoll sein können. Dies würde auch Rechenaufwand reduzieren, was komplexere Netze erlauben würde.

7.2 Ausblick

Auf diesen Ergebnissen aufbauend, liegt ein Schwerpunkt in zukünftigen Weiterentwicklungen auf der Untersuchung von kürzeren Zeitreihen. Darüber hinaus ist es ratsam, alternative Ansätze wie RNNs als One-To-One Lösungen zu evaluieren. Ein solcher Ansatz könnte neue Einblicke in die Modellierung von zeitlichen Abhängigkeiten bieten, bei möglicherweise ähnlichen Vorhersageleistungen.

Es ist jedoch wichtig anzumerken, dass die Implementierung eines Reaktionsverhaltens, wie Eingangs dieser Arbeit beschrieben, auf Basis dieser Modelle erst erfolgen sollte, wenn sie erfolgreich funktionieren. Ein funktionsfähiges Modell als Grundlage muss präzise Vorhersagen treffen, um effektiv zu sein. Daher ist es von entscheidender Bedeutung, dass die Modelle ausreichend validiert und optimiert werden, bevor sie in reaktive Prozesse integriert werden.

Literaturverzeichnis

[AA] Amidi, Afshine ; Amidi, Shervine: *Recurrent Neural Networks cheatsheet*. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks, . – Abgerufen am 23. März 2024

[AB17] Anthony Bagnall, Aaron Bostrom James Large Eamonn K. Jason Lines L. Jason Lines: *The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances*. 2017 https://doi.org/10.1007/s10618-016-0483-9

[Bis06] Bishop, Christopher: *Pattern Recognition and Machine Learning*. Springer, 2006 https://www.microsoft.com/en-us/research/publication/pattern-recognition-machine-learning/

[CMG+14] Cho, Kyunghyun ; Merrienboer, Bart van ; Gulcehre, Caglar ; Bahdanau, Dzmitry ; Bougares, Fethi ; Schwenk, Holger ; Bengio, Yoshua: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014

[ESY20] El-Sheimy, Naser ; Youssef, Ahmed: Inertial sensors technologies for navigation applications: state of the art and future trends. In: *Satellite navigation* 1 (2020), Nr. 1, S. 1–21. – ISSN 2662–1363

[GBB11] Glorot, Xavier ; Bordes, Antoine ; Bengio, Yoshua: Deep Sparse Rectifier Neural Networks. In: Gordon, Geoffrey (Hrsg.) ; Dunson, David (Hrsg.) ; Dudík, Miroslav (Hrsg.): *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*

Bd. 15. Fort Lauderdale, FL, USA : PMLR, 11–13 Apr 2011 (Proceedings of Machine Learning Research), 315–323

[GMH13] Graves, Alex ; Mohamed, Abdel-rahman ; Hinton, Geoffrey: Speech recognition with deep recurrent neural networks. In: *2013 IEEE international conference on acoustics, speech and signal processing* Ieee, 2013, S. 6645–6649

[HS97] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-term Memory. In: *Neural computation* 9 (1997), S. 1735–80. http://dx.doi.org/10.1162/neco.1997.9.8.1735. – DOI 10.1162/neco.1997.9.8.1735

[HS03] Hüsken, Michael ; Stagge, Peter: Recurrent neural networks for time series classification. In: *Neurocomputing* 50 (2003), S. 223–235

[LBBH98] Lecun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-based learning applied to document recognition. In: *Proceedings of the IEEE* 86 (1998), Nr. 11, S. 2278–2324. http://dx.doi.org/10.1109/5.726791. – DOI 10.1109/5.726791

[Lec89] In: Lecun, Yann: *Generalization and network design strategies*. Elsevier, 1989

[Naoa] *Inertial unit*. http://doc.aldebaran.com/2-1/family/robots/inertial_robot.html#robot-inertial, . – Abgerufen am 23. März 2024

[Naob] *NAO - Technical overview*. http://doc.aldebaran.com/2-1/family/robots/index_robots.html, . – Abgerufen am 23. März 2024

[OBL+19] O'Malley, Tom ; Bursztein, Elie ; Long, James ; Chollet, François ; Jin, Haifeng ; Invernizzi, Luca u. a.: *KerasTuner*. https://github.com/keras-team/keras-tuner, 2019

[Rob] *Objective [of RoboCup]*. https://www.robocup.org/objective, . – Abgerufen am 23. März 2024

[RP99] Riesenhuber, Maximilian ; Poggio, Tomaso: Hierarchical models of object recognition in cortex. In: *Nature neuroscience* 2 (1999), Nr. 11, S. 1019–1025

[SKM20] Seel, Thomas ; Kok, Manon ; McGinnis, Ryan S.: Inertial sensors—applications and challenges in a nutshell. In: *Sensors (Basel, Switzerland)* 20 (2020), Nr. 21, S. 1–5. – ISSN 1424–8220

[SMH11] Sutskever, Ilya ; Martens, James ; Hinton, Geoffrey E.: Generating text with recurrent neural networks. In: *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, S. 1017–1024

[SPL] *WELCOME TO ROBOCUP SPL!* https://spl.robocup.org/, . – Abgerufen am 23. März 2024

[SS20] Singh, Dalwinder ; Singh, Birmohan: Investigating the impact of data normalization on classification performance. In: *Applied Soft Computing* 97 (2020), S. 105524. http://dx.doi.org/https://doi.org/10.1016/j.asoc.2019.105524. – DOI https://doi.org/10.1016/j.asoc.2019.105524. – ISSN 1568–4946

[SWT18] Shirke, Vishal ; Walika, Ritesh ; Tambade, Lalita: Drop: a simple way to prevent neural network by overfitting. In: *Int. J. Res. Eng. Sci. Manag* 1 (2018), S. 2581–5782

[tfS] *Tensorflow Sparse Categorical Accuracy.* https://www.tensorflow.org/api_docs/python/tf/keras/metrics/SparseCategoricalAccuracy, . – Abgerufen am 23. März 2024

A Eidesstatliche Erklärung

Ich versichere, dass die Bachelorarbeit mit dem Titel „Sturzvorhersage bei humanoiden Robotern mithilfe von Machine Learning“ nicht anderweitig als Prüfungsleistung verwendet wurde und diese Bachelorarbeit noch nicht veröffentlicht worden ist. Die hier vorgelegte Bachelorarbeit habe ich selbstständig und ohne fremde Hilfe abgefasst. Ich habe keine anderen Quellen und Hilfsmittel als die angegebenen benutzt. Diesen Werken wörtlich oder sinngemäß entnommene Stellen habe ich als solche gekennzeichnet.

Leipzig, 20. September 2024 Unterschrift