Paper 10/2019 doi: 10.7795/320.201910

JUNGE wissenschaft

JungforscherInnen publizieren
online | peer reviewed | original

Verlag: Physikalisch-Technische Bundesanstalt
PTB

Mathematik & Informatik

Wettrennen gegen den Computer

ENTWICKLUNG EINES COMPUTERGEGNERS FÜR CARRERA-BAHNEN

Es wurde ein Programm entwickelt, das ein Rennauto auf einer Carrera-Bahn selbstständig steuert. Neben einem normalen Fahrmodus beinhaltet das System einen selbstlernenden Neural-Network-Fahrmodus, welcher es entweder ermöglicht, die Fahrkünste des Computers während eines Trainings durch einen menschlichen Spieler zu verbessern, oder den Computer mittels eines evolutionären Algorithmus' selbstständig trainieren zu lassen.

DER JUNGFORSCHER

Ferdinand Krämer (1998)
Johannes-Gymnasium, Lahnstein

Eingang der Arbeit:
8.6.2018
Arbeit angenommen:
28.11.2018

Wettrennen gegen den Computer

ENTWICKLUNG EINES COMPUTERGEGNERS FÜR CARRERA-BAHNEN

1. Einleitung

Spurgebundene Autorennbahnen, bei denen zwei oder mehr Fahrer mithilfe von Handcontrollern maßstabgetreue Modellautos über eine kleine Rennstrecke steuern, gehören schon seit über 60 Jahren zur Standardausstattung jedes Spielwarengeschäfts und der meisten Kinderzimmer. Gerade dort verstauben sie allerdings auch oft in den Regalen, wegen eines beim Kauf zu selten beachteten Nachteils: Zum sinnvollen Spielen benötigt man üblicherweise zwei oder mehr Fahrer.

Zwar gibt es bereits seit über 20 Jahren digitale Varianten der herkömmlichen Rennbahnanlagen, die oft auch die Möglichkeit bieten, ein Auto mit vorher eingestellter, konstanter Geschwindigkeit selbstständig über den Kurs fahren zu lassen, sonderlich spannend sind Rennen gegen solche Gegner jedoch eindeutig nicht.

Ziel meines Projektes war es also, diesen Nachteil von Autorennbahnen zu beheben: Mit einem intelligenten, lernfähigen Computergegner für Carrera-Bahnen (wie Autorennbahnen in Deutschland wegen des enormen Erfolgs dieser Marke oft auch synonym genannt werden).

Im Einzelnen sieht die Zielsetzung wie folgt aus:

- Es soll ein Programm zur automatischen, möglichst effektiven Steuerung eines Autos entwickelt werden sowie eine Hardwarekomponente, um die Befehle des Programms auf die Anlage zu übertragen.
- Um wirklich auf Dauer effektiv gegen jeden menschlichen Gegner anzukommen, soll das Programm eine gewisse lernfähige Komponente besitzen.
- Die Hardware und das Programm sollen ohne jegliche Komplikationen auf jeder beliebigen Carrera-Anlage bis zu einer bestimmten Größe funktionieren, die Anlage soll sich auch während der Laufzeit des Programms umbauen lassen.

2. Materialien

Als Grundlage für das Projekt dient eine Carrera-Digital-Anlage im Maßstab 1:32, deren Fahrzeuge ich im analogen Modus betreibe (siehe 3.). Die Größe der Fahrzeuge bietet dabei die optimale Voraussetzung für eine zuverlässige Erkennung in Webcam-Bildern. Besonders gut für das Kameratracking eignen sich drei Fahrzeuge, die relativ gleichförmige Oberseiten in auffälligen Farben (grün, gelb, blau) haben.

Zur Steuerung der Anlage verwende ich zwei Fischertechnik „ROBOTICS TXT“-Interfaces. Zwar wäre es sicher eine billigere Alternative gewesen, stattdessen einen Raspberry Pi oder ähnliche Systeme zu verwenden, aber da mir einer dieser Controller ohnehin zur Verfügung stand und durch das einheitliche Stecksystem von Fischertechnik die Verkabelung viel einfacher und übersichtlicher wird, entschied ich mich für diese Interfaces. Was die Fischertechnik-Interfaces für mein Projekt besonders geeignet macht, ist die Python-Library *ftrobopy*: Damit ist es möglich, das eigentliche Steuerungsprogramm in Python auf einem leistungsfähigen Computer laufen zu lassen, während die Interfaces nur als Ein- und Ausgänge, also als Schnittstelle zur Anlage dienen. Direkt auf der Hardware eines Raspberry Pi o. ä. wäre mein Programm ohne große Kompromisse auf Kosten der Effektivität nicht lauffähig gewesen, und mir ist keine Möglichkeit bekannt, einen Raspberry Pi derart einfach und effektiv mit dem Computer fernzusteuern wie diese Fischertechnik-Controller.

Ebenfalls aus Fischertechnik besteht die diverse Zusatzhardware: Ein Gehäuse

doi: 10.7795/320.201910

mit Bedienfeld zur Steuerung der Anlage, um nicht für sämtliche Einstellungen etc. immer wieder den Computer benutzen zu müssen, sowie eine Brücke mit Startampel und vier Lichtschranken, um die Rundenerfassung bei den Rennen möglichst exakt zu halten. Das Gehäuse enthält zudem noch zwei einfache Transistorverstärker, um überhaupt die Carrera-Fahrzeuge (14,8 V/3,5 A) mit den Fischertechnik-Controllern (9 V/1,5 A) ansteuern zu können, ein paar bistabile Relais (G6AK-274P-ST-US von Omron) zur bequemen automatischen Umschaltung der beiden Fahrstrecken zwischen Computer und menschlichem Spieler, sowie vier Leistungswiderstände. Die Versorgung der *Interfaces* etc. übernimmt ein kleines Labornetzgerät.

Die Erkennung des zu steuernden Autos geschieht mittels zweier Logitech-Webcams des Typs „C922 PRO STREAM“, die von einem ca. 1,80 m hohen Fischertechnik-Statik-Turm in der Mitte der Anlage senkrecht auf diese herunterschauen. Einige der für diesen Turm benötigten Teile wurden mir ebenfalls von Fischertechnik kostenlos zur Verfügung gestellt. Die Webcams decken ein Feld von ca. 1,50 m x 2,50 m ab, welches somit die maximale Größe der steuerbaren Anlage darstellt; Die Turmhöhe von 1,80 m steht meiner Einschätzung nach im optimalen Verhältnis zur Auflösung der Webcams.

Das Steuerungsprogramm ist vollständig in Python geschrieben und verwendet folgende Bibliotheken:

- *OpenCV* [1] für die Bildaufnahme und -verarbeitung sowie diverse *Interface-Features*
- *Imutils* [2], eine kleine Sammlung von Bequemlichkeitsfunktionen, die die Einbindung von *OpenCV* erleichtern
- *TensorFlow* [3] als einfache Bibliothek für maschinelles Lernen
- *ftrobopy* [4] für die Kommunikation mit den Fischertechnik-Controllern

3. Aufbau der Strecke und Ansteuerung

Als erstes galt es, grundsätzliche Voraussetzungen für die Ansteuerung der Anlage sowie das Tracking der Fahrzeuge zu entwickeln.

Ich entschied mich, die Anlage analog anzusteuern. Dafür gab es mehrere Gründe: Viele meiner Carrera-Fahrzeuge verfügen zwar über einen Digitaldecoder, aber die Nachahmung der proprietären Steuercodes mit *Python*, die zudem nicht einmal offiziell dokumentiert sind, hätte nur unnötigen weiteren Aufwand verursacht und wäre außerdem mit den Fischertechnik-Controllern gar nicht möglich gewesen. Die hohe Frequenz von mehreren Kilohertz, mit denen die digitalen Bahnsignale über die Versorgungsspannung an die Autos übermittelt werden, wird von diesen – zumindest im Online-Modus, also während der Fernsteuerung durch einen PC – nicht unterstützt, da die Controller in diesem Modus lediglich alle 0,01 s neue Befehle vom Computer entgegennehmen. Und da die zusätzlichen Features eines digitalen Betriebs (Weichen etc.) ohnehin auch aus anderen Gründen für mein Projekt nicht nutzbar gewesen wären, entschied ich mich für die analoge Variante.

Für die analoge Ansteuerung wird das (PWM-) Ausgangssignal des Fischertechnik-Controllers (9 V/1,5 A) mit einem Transistorverstärker auf 14,8 V und 3,5 A gebracht (siehe Abb. 1). Ursprünglich diente mir dafür ein alter Fischertechnik-Verstärker, der jedoch wegen Überlastung durchbrannte. Dessen Schaltung kopierte ich 1 : 1 mit neuen Bauteilen und einem etwas stärkeren Transistor auf eine Europlatine. Dadurch ging ich bezüglich der Funktionalität kein Risiko ein. Um später auch beide Strecken gleichzeitig ansteuern zu können, z. B. für Rennen nur zwischen computergesteuerten Fahrzeugen, baute ich gleich zwei dieser Verstärker ein.

Spätestens mit diesem Schritt war auch der Einbau einiger bistabiler Relais nötig (siehe Abb. 2): Ein Relais pro Strecke dient zum An- und Ausschalten der Stromversorgung für den normalen Betrieb über Handcontroller, da die dort anliegende Spannung bei gleichzeitiger direkter Ansteuerung der Strecke zu Problemen führte (das Auto er-

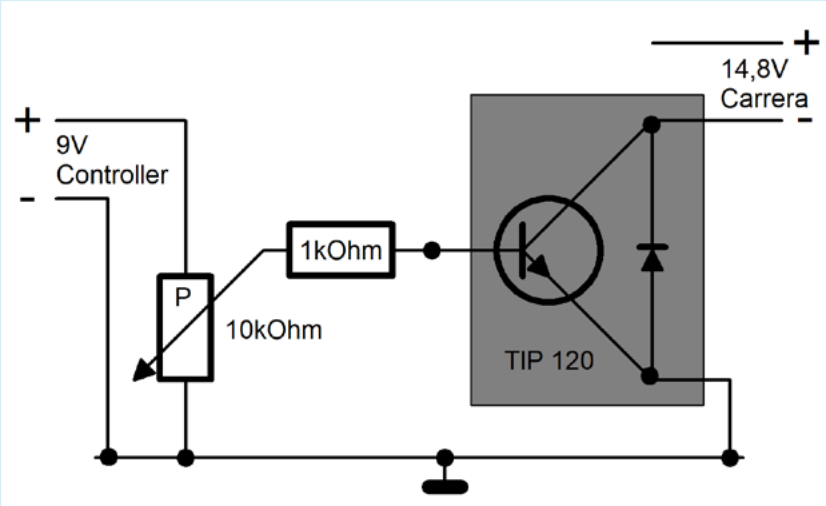

Abb. 1: Verstärkerschaltung

doi: 10.7795/320.201910

hielt dauerhaft die Maximalspannung und flog sofort mit Höchstgeschwindigkeit aus der nächstbesten Kurve). Ein zweites Relais für jede Strecke schaltet die direkt an den Gleisen angelöteten Anschlüsse zwischen Fahr- und Messmodus um: Im Fahrmodus ist dieser Anschluss mit der eben genannten Verstärkerschaltung verbunden, um die entsprechende Strecke mit dem Computer steuern zu können, im Messmodus mit einem Spannungsteiler aus einem 10 kΩ und einem 5 kΩ Leistungswiderstand. Dieser ist wichtig für den manuellen *Neural-Network*-Trainingsmodus, da der Computer dafür die Spannung an der Strecke messen können muss, die ein menschlicher Fahrer mit seinem Handcontroller vorgibt. Dies ist nur mit einem hochohmigen Spannungsteiler möglich, der die viel zu hohe Carrera-Streckenspannung wieder auf Fischertechnik verträgliche 9 V zurückübersetzt, ohne dabei dem Fahrzeug zu viel Leistung zu entziehen. Ein letztes Relais dient zur Abtrennung dieser Spannungsteiler von der Schaltung, wenn diese – außerhalb des *Neural-Network*-Trainingsmodus – nicht benötigt werden, um in diesem Fall auch diesen geringen Leistungsverlust zu vermeiden.

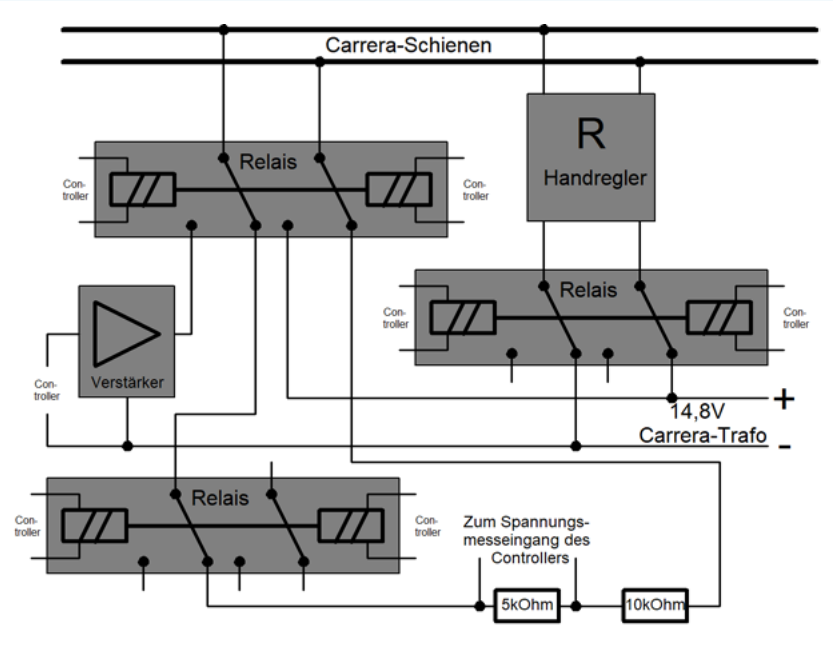

Abb. 2: Schaltplan für eine Spur der Carrera-Anlage

Abb. 3: Bedienpanel

All diese Technik findet Platz in einem kompakten und stabilen Fischertechnik-Kasten, an den sämtliche Stromversorgungen und Anlagenteile angeschlossen werden können (siehe Abb. 3). Die Oberseite dieses Kastens zieren dabei ein paar scheinbar wahllos verstreute Taster und Schalter. Diese dienen als Bedienfeld für beinahe alle Funktionen der Anlage, da eine Bedienung „direkt am Boden“ für den echten Spielbetrieb unerlässlich ist: Jedes Mal aufstehen, Maus und Tastatur benutzen und Einstellungen oder den Rennstart am Computer vornehmen ist auf Dauer sehr lästig. Der Computer dient nur noch zum eigentlichen Start des Programms sowie zur Anzeige von Kamerabildern, Streckenplan, Scandaten etc. und einiger unwichtiger Bedienelemente, könnte also für ein fertiges Produkt auch problemlos weggelassen bzw. in den „Steuerungskasten“ integriert werden. Gerne hätte ich zur Anzeige derartiger Daten oder generell zur einfacheren und moderneren Bedienung der Anlage auch die integrierten Touchscreens der beiden Fischertechnik-Controller genutzt, leider werden diese von *ftrobopy* aber bis jetzt in keiner Weise unterstützt, weder zur Anzeige noch zur Eingabe.

Zu einer spieltauglichen Carrera-Anlage gehören eine Startampel sowie eine Möglichkeit zur exakten Rundenzählung. Beides habe ich mit einer kleinen Fischertechnikbrücke verwirklicht. Die Startampel übernimmt gleichzeitig auch die Anzeige der siegreichen Strecke. Die zwei Lichtschrankenpaare dienen als Start- und Ziellinie für die beiden Strecken, wobei die zusätzliche

doi: 10.7795/320.201910

Startlinie Komfortfunktionen wie eine automatische Rückkehr zum Start oder eine exakte Geschwindigkeitsmessung ermöglicht.

Ein zentrales Element meiner Anlage ist der Kameraturm: dieser ist ca. 1,80 m hoch, aus Fischertechnik-Statik-Teilen gebaut und trägt an langen Armen in ca. 1,70 m Höhe zwei Logitech-Webcams („C922 PRO STREAM“). Diese sind von den Spitzen der Arme senkrecht nach unten auf die Anlage gerichtet, sodass am unteren Bildrand der Kamerabilder gerade so die Hälfte vom Fuß des Turms zu sehen ist, um die Bilder im Programm entsprechend lückenlos zusammenfügen zu können. Innerhalb dieses Sichtbereichs der Kameras darf die Carrera-Strecke beliebig aufgebaut werden. Die Webcams sind per freischwebendem USB-Kabel mit dem PC verbunden.

4. Entwicklung eines einfachen Computergegners

4.1 Erkennung der Fahrzeuge – Tracking

Damit der Computer ein Auto über die Strecke fahren lassen kann, muss er wissen, wo das Auto ist. Dies geschieht mithilfe der Kameras und der Klasse „*Tracking*“, die aufgrund ihrer Größe ausgelagert wurde und die einige Funktionen der *Python-Library* „*OpenCV*“ nutzt, um die Kamerabilder einzulesen und darin die Autos zu erkennen. Kleine Teile des Codes dieser Klasse sind aus einem Beitrag eines englischsprachigen Bilderkennungs-Blogs übernommen [1].

Die Klasse startet bei der Initialisierung einen neuen *Thread*, der die ganze Zeit läuft. In diesem *Thread* werden stets die aktuellsten *Frames* der beiden Kameras gelesen, von denen einer noch um 180° gedreht wird, damit sie lückenlos zusammengefügt werden können (siehe Abb. 4). Dieser kombinierte *Frame* wird jetzt in das HSV-Farbschema (**H***ue,* **S***aturation,* **V***alue*, also Helligkeit, Sättigung, Farbwert) konvertiert. Dieses eignet sich für den nächsten Schritt: die Farbbereichsfilterung. Dafür prüft *OpenCV* für jeden Bildpixel, ob sein Farbwert innerhalb eines vorgegebenen Bereichs liegt, und gibt das Ergebnis als Schwarz-weiß-Bild aus: Bildpunkte, die im Bereich liegen, werden weiß, alle anderen schwarz eingefärbt. Die so entstandenen Schwarz-Weiß-Bilder zeigen aufgrund des natürlichen Farbrauschens der Kameras noch viele kleine Pixelfehler: Einzelne Bildpunkte oder Streifen des Hintergrunds werden fälschlicherweise weiß eingefärbt und die Bereiche der Autos werden durch schwarze Artefakte unterbrochen. Um diese Fehler zu beheben, werden zunächst sämtliche weißen Flächen von ihren Rändern her um eine bestimmte Anzahl von Pixeln verkleinert (erodiert) (dadurch verschwinden die einzelnen weißen Pixel und Streifen), und dann werden die übriggebliebenen weißen Flächen an ihren Rändern um die doppelte Anzahl von Pixeln erweitert (dadurch wachsen die schwarzen Flecken zu). Nach mehrmaliger Erosion und anschließender Erweiterung stechen im entstehenden Schwarz-Weiß-Bild die Bereiche der Autos klar heraus (siehe Abb. 4). Nun wird eine Liste der in diesem Schwarz-Weiß-Bild gefundenen Konturen erstellt und überprüft: Wenn eine Struktur von relevanter Größe in der Farbe des zu erkennenden Autos (einstellbar über die erwähnten Farbbereichsgrenzen) erkannt wurde, werden die Koordinaten des Mittelpunkts dieser Struktur sowie der Eckpunkte eines Rechtecks, das die Struktur einschließt, gespeichert, welche so die Position und die ungefähre Ausdehnung eines Fahrzeugs wiedergeben. Diese Schritte werden für jedes zu erkennende Fahrzeug mit entsprechend anderen Farbbereichsgrenzen wiederholt, die Fahrzeuge werden also nur über ihre Farbe unterschieden.

Die Funktion `getPos()` gibt die gespeicherten Positionsdaten (also Mit-

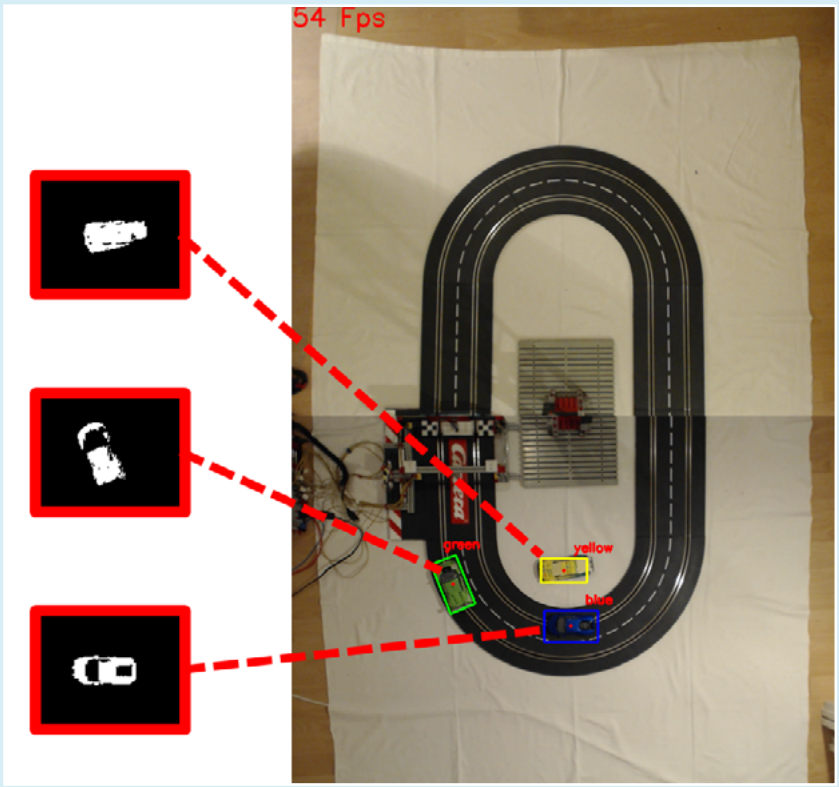

Abb. 4: Erkennung der Fahrzeuge in den Kamerabildern

doi: 10.7795/320.201910

tel- und Eckpunktkoordinaten) für alle erkannten Fahrzeuge aus. Außerdem stehen noch weitere kleinere Funktionen bereit, die z. B. die auf dem Bildschirm anzuzeigenden Kamerabilder um Rahmen um die Fahrzeuge oder den Streckenplan (siehe unten) um sich bewegende Punkte an der Position der Fahrzeuge ergänzen.

4.2 Einscannen und Berechnen der Strecke

Als nächstes muss der Computer die Strecke, die beliebig aufgebaut werden darf, erkennen können. Dafür wird bei einem Druck auf den entsprechenden Taster am Bedienfeld die kleine Funktion `scannen()` gestartet, welche ein Auto mit moderater Geschwindigkeit eine Runde fahren lässt und dabei dessen mittels `getPos()` bestimmte Positionen in einer langen Liste sammelt. Die in dieser Liste gespeicherten Punkte ergeben also die Form der gescannten Strecke.

Die Ermittlung aller relevanten Streckendaten wie Position, Richtung und Länge der Geraden und Kurven aus dieser Liste wurde aufgrund der enormen Größe des benötigten Programms in die Klasse Streckenberechner ausgelagert. Dieser wird die Liste bei der Initialisierung übergeben, und mit dem Aufruf der Funktion `Streckenberechner.streckeberechnen()` werden alle benötigten Operationen automatisch ausgeführt, die der Übersichtlichkeit halber jeweils in einer eigenen Funktion liegen.

Die erste Unterfunktion, die die Funktion `streckeberechnen()` aufruft, ist `vorfiltern()` und streicht alle mehrfach vorhandenen Elemente aus der Streckenliste, die zum Beispiel entstehen, wenn die Kameras aufgrund besonders guter Lichtverhältnisse deutlich mehr Bilder pro Sekunde aufnehmen.

Die zweite Unterfunktion ist `strecke2dropout()`. Die eben bestimmten Punkte liegen zu dicht beieinander, um später anhand der Winkel aus den Vektoren zwischen diesen Punkten zuverlässig Kurve und Gerade unterscheiden zu können. Daher muss die Qualität hier künstlich vermindert werden. Dazu werden nur Punkte aus der Streckenliste an die Liste `dropout` angehängt, die einen gewissen, einstellbaren Abstand zueinander möglichst genau einhalten. Allerdings setzt diese verringerte Genauigkeit auch den Gestaltungsmöglichkeiten der Strecke Grenzen: Kann man in den ungefilterten Daten noch jede Einzelheit erkennen, so sinkt die Detailschärfe nun auf größer als zehn Streckenzentimeter. Dadurch kann der Computer zwar ohne Probleme die groben Konturen der Strecke erkennen und die Lage von Kurven und Geraden etc. berechnen, ist aber nicht mehr in der Lage, zuverlässig z. B. eine Kreuzung, Engstelle oder Spurwechselweiche zu unterscheiden. Diese Streckenelemente dürfen in Kombination mit meinem Programm daher nicht genutzt werden, da das computergesteuerte Fahrzeug nicht darauf reagieren könnte.

Die nächste Funktion, die beim anfänglichen Berechnen der Streckeneigenschaften aufgerufen wird, ist `dropout2vectors()`. Diese berechnet durch Koordinatensubtraktionen die Richtungsvektoren zwischen den Punkten in `dropout`, und hängt diese so an die neue Liste `vectors` an, dass ein Vektor in `vectors` immer denselben Index hat wie der Punkt in `dropout`, zu dem dieser Vektor hinführt. Sollten trotz der Reduktion der Scanliste in `strecke2dropout()` hier noch Nullvektoren übrigbleiben, d. h. Scanpunkte, die genau übereinanderliegen (was aufgrund der Arbeitsweise dieser Funktion im Bereich des Start/Zielpunktes durchaus vorkommen kann), so werden diese Punkte spätestens hier herausgefiltert und in allen Listen, also `dropout` und `vectors`, gelöscht.

Diese Wahrung der Reihenfolge ist im gesamten Streckenberechnungsprogramm sehr wichtig: Die Indizes sämtlicher hier erwähnter Listen müssen stets synchron laufen, da oft in einer späteren Funktion auf Daten aus Listen an anderer Stelle in der Kette zugegriffen werden muss. Generell muss man stets sämtliche, mit einem bestimmten gescannten Punkt verbundenen Daten erhalten können, indem man aus allen hier eingeführten Listen die Elemente mit demselben Index abruft.

Die nächste Funktion in der Kette, `vectors2winkel()`, ist eine der wichtigsten: Sie berechnet mithilfe der Formel:

$$\alpha = \cos^{-1}\left(\frac{|\vec{u} \circ \vec{v}|}{|\vec{u}| \cdot |\vec{v}|}\right)$$

die Winkel zwischen den Vektoren in `vectors` sowie über das Vorzeichen des Kreuzprodukts aus den Vektoren die Richtung dieser Winkel (links, rechts oder 0°, also geradeaus) und hängt diese Daten an die neue Liste `winkel` an.

In der nächsten Funktion, `winkel2winkelslices()`, wird diese Winkelliste vervierfacht: Die neue Liste `winkelslices` enthält in jedem Element nicht nur den Winkel an diesem Punkt, sondern auch noch die an den drei vorherigen Punkten, dafür wird aber die Richtungsinformation weggelassen. Diese Aufteilung dient nur als Vorbereitung für den nächsten Schritt: Die Unterscheidung zwischen Kurven und Geraden in der Funktion `winkelslices2structures`.

Die erste Idee, wie diese Unterscheidung vonstattengehen könnte, war, alle Winkel unter einem bestimmten Grenzwert als „Gerade“ zu definieren und alle darüber als Kurve. Leider brachte dieser Ansatz nicht die gewünschten Ergebnisse, da trotz der Vorfilterung in `strecke2dropout()` teilweise noch mitten in der Kurve die Winkel zwischen den Vektoren kleiner als 3° betrugen oder durch Ungenauigkeiten infolge schlechter Lichtverhältnisse beim Scan-

doi: 10.7795/320.201910

nen auf geraden Strecken größer als 5°. Nach einigen Experimenten mit Durchschnittsbildung oder erweiterter Vorfilterung entschied ich mich für folgende Lösung: Ein kleines *TensorFlow Neural Network*. Dieses ist definiert als einfaches *Deep Neural Network* mit 4 Eingabeneuronen, gefolgt von drei Neuronenschichten mit 10, 20 und 10 Neuronen sowie zwei Ausgangsklassen, nämlich „Kurve“ oder „Gerade“. Zum Trainieren dieses Netzwerks wurde händisch eine CSV-Liste von ca. 100 Zeilen mit jeweils vier aufeinanderfolgenden Ausgabewerten von `vectors2winkel()` und der Klassifikation, ob sich diese Abfolge in einer Kurve oder einer Geraden findet, angelegt und einem leicht abgewandelten Beispielprogramm von einer *TensorFlow*-Dokumentationsseite [3] übergeben, welches sich genau damit befasst. Nach dem recht kurzen Training kann das fertig trainierte Netzwerk als Modell gespeichert werden, sodass künftig beliebig darauf zugegriffen werden kann. Dies geschieht nun in `winkelslices2structures()`:

Durch das Definieren eines *TensorFlow*-`DNNClassifiers` mit dem Parameter `model_dir=[Pfad/zum/eben/trainierten/Modell]`, eine `input_function` für Vorhersagen mit diesem Netzwerk, die die eben erstellte Liste winkelslices Element für Element vorliest, und durch die Funktion `DNNClassifier.predict()` kann nun die Strecke in Kurven und Geraden aufgeteilt werden. Diese Aufteilung wird zusammen mit der aus winkel übernommenen Kurvenrichtung in der neuen Liste structures gespeichert. Dabei wird die Kurvenrichtung noch leicht bereinigt: wenn das Netzwerk auf Gerade entscheidet, wird unabhängig vom tatsächlichen Wert in winkel die Richtung 0 angehängt. Das Neural Network verwendet die 4er-Pakete von Winkelwerten aus der `winkelslices`-Liste als Eingabe und findet auf Basis seiner Trainingsdaten eine passende Ausgabe, also die Klassifikation „Kurve“ oder „Gerade“. Da diese Trainingsdaten dieselben oben beschriebenen Unschärfen enthielten wie auch die neuen Daten, hat das Neural Network gelernt, damit umzugehen und trotzdem zu über 98 Prozent zuverlässig zu entscheiden. Und da `winkelslices` für jeden einzelnen Scanpunkt ein Paket aus dem Winkel an diesem Punkt und an den letzten drei Punkten enthält, sind diese Pakete untereinander stets nur um einen Scanpunkt versetzt, was insbesondere an den Übergängen zwischen Kurven und Geraden wichtig ist: Das Netzwerk bekommt immer vier Eingangswerte, die es aber nur gemeinsam einem Ausgangswert zuordnen kann, und ändert dementsprechend auch alle vier letzten Werte in `structures`. Bekommt das Netzwerk nun an einem Übergang zwei Werte, die zu einer Kurve gehören, und zwei, die zu einer Geraden gehören, so ist es darauf trainiert, im Zweifel immer zugunsten der ersten Werte zu entscheiden (beim Übergang Kurve zu Gerade also Kurve), da die darauffolgenden Werte mit dem übernächsten Paket, welches dann in diesem Beispiel nur noch Winkelwerte einer Geraden enthält, korrigiert werden können. Zusätzlich zum Erstellen der `structures`-Liste zeichnet `winkelslices2structures()` auf Wunsch auch noch den Streckenplan in Form von farblich kodierten Vektoren (Kurven rot, Geraden blau) auf schwarzem Hintergrund in ein neues Fenster, welches ebenfalls genau die Größe der addierten Kamerabilder hat (siehe Abb. 5).

Nachdem die Strukturen berechnet sind, wird vor dem letzten Schritt noch die Funktion `strukturenergänzen` aufgerufen, die die Verkürzung der Liste in `strecke2dropout` zugunsten einer höheren Auflösung rückgängig macht. Dafür werden zwischen die

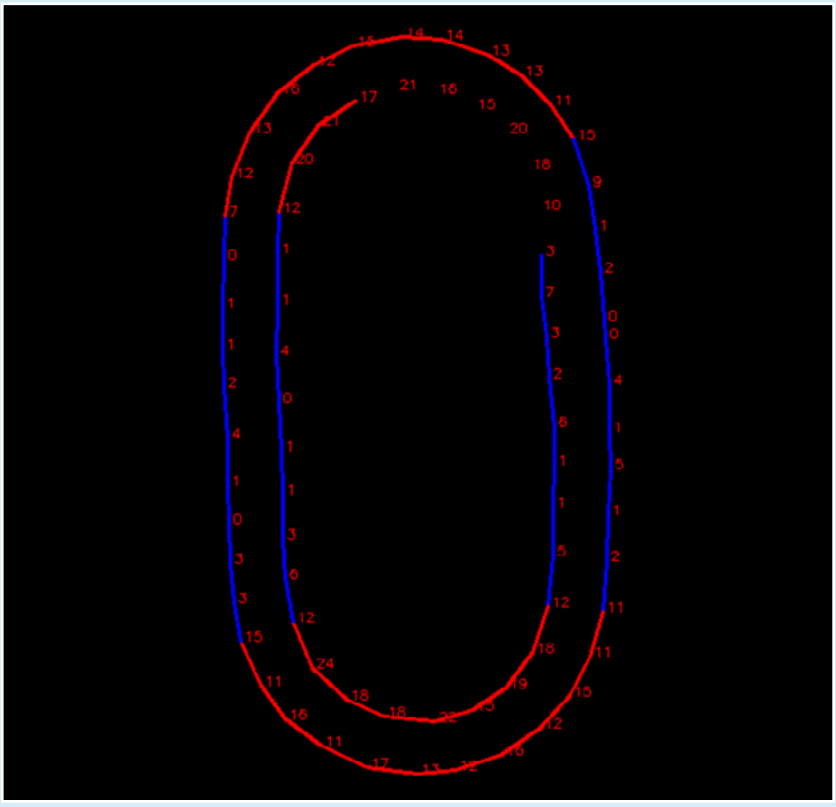

Abb. 5: Ein Streckenplan entsteht: aktuell berechnet `winkelslices2structures()` nach und nach die Lage der Kurven und Geraden

doi: 10.7795/320.201910

Werte der `structures`-Liste so viele Werte eingefügt, dass sie wieder exakt der `strecke`-Liste entspricht, die eingefügten Elemente übernehmen dabei die Struktureigenschaften ihrer Nachbarn.

Die letzte Funktion, die `Streckenberechner.streckeberechnen()` aufruft, ist `structures2strulän()`, wobei die Abkürzung strulän für „Strukturen und Längen" steht. Die von dieser Funktion zurückgegebene Liste `strulän` enthält Dreier-Tupel:

- die durch `winkelslices2structures()` korrigierten Winkelrichtungsangabe an diesem Punkt, die seit der Korrektur zuverlässig Auskunft über die Struktur der Strecke, also Kurve oder Gerade, gibt,
- der Gesamtlänge der Struktur (Kurve oder Gerade), zu der dieser Punkt gehört, und
- der Prozentsatz dieser Gesamtlänge der Struktur, der an diesem Punkt bereits zurückgelegt wurde.

Die Liste `strulän` ist daher die komplizierteste, die der `Streckenberechner` ausgibt, wird aber für die später beschriebenen *Neural-Network*-Fahren-Funktionen benötigt. Der beschriebene Scanvorgang und das Berechnen der Streckendaten können während der Laufzeit des Programms beliebig oft gestartet werden, die jeweiligen Streckendaten werden dann durch die neuen überschrieben. Es ist also möglich, jederzeit die Strecke umzubauen und nach einem Scan für den Computer befahrbar zu machen – was eine der Hauptzielsetzungen war.

Die Informationen, die der Computer nach dieser Scanfahrt und den dazugehörigen Berechnungen über die Strecke besitzt, reichen für den „einfachen Fahrmodus", bei dem der Computer das Auto automatisiert über die Strecke steuert.

4.3 Der „einfache Fahrmodus"

Der einfache Fahrmodus startet, wenn die Schalter am Bedienfeld entsprechend gestellt sind und der Start-Knopf gedrückt wird. Er beginnt mit einer Abfrage, ob die Liste `strulän` existiert. Ist das nicht der Fall, wird ein Warnton ausgegeben. Als Nächstes werden zwei Rundenzähler-Objekte zurückgesetzt, eines für jede Strecke. Die Rundenzähler-Klasse startet bei ihrer Initialisierung einen neuen Thread, der stetig den Zustand der zur jeweiligen Strecke gehörenden Zielichtschranke prüft und auf Basis dessen die Runden-Variable für diese Strecke inkrementiert. Zusätzlich wird per Relais der Strom für die vom menschlichen Fahrer kontrollierte Strecke abgeschaltet, um Frühstarts zu vermeiden. Nach dem Durchlaufen der Startampel – einer Abfolge von Befehlen zur Steuerung der entsprechenden Lampen und zur Erzeugung von Sinustönen verschiedener Frequenzen – wird die Strecke freigegeben, während parallel das eigentliche Fahrprogramm in der Funktion `fahren()` die Arbeit aufnimmt.

Dieses besteht aus einer Schleife, in der zunächst die aktuellen Trackingdaten abgefragt werden. Wurde ein Auto erkannt, wird dessen Position mithilfe der Funktionen aus der `Tracking`-Klasse als Punkt in der entsprechenden Farbe in den Streckenplan eingezeichnet – eine kleine Komfortfunktion, die so das Auto als beweglichen Punkt auf dem Bildschirm darstellt.

Dann folgt die eigentliche Fahrfunktion: In einer inneren Schleife werden die Koordinaten mit allen Koordinaten in der Liste `strecke` aus dem `Streckenberechner` verglichen, bis entweder die Liste zu Ende ist oder aber eine Übereinstimmung mit einer Toleranz von 4 (Kamerapixel) in x- und y-Richtung gefunden wurde. Mit dem Index des Listenelements, das mit den aufgenommenen Koordinaten so übereinstimmt, können nun aus den letzten zwei vom `Streckenberechner` bereitgehaltenen Listen alle für diesen Streckenpunkt relevanten Daten entnommen werden – also in diesem Fall vor allem der entsprechende Wert in der Liste `structures`, der Auskunft darüber gibt, ob sich das Auto an diesem Punkt auf einer Geraden oder einer Kurve befindet.

Befindet es sich auf einer Geraden, wird rekursiv eine vorher definierte Beschleunigung eingeleitet: Das heißt, dass bei jedem aufeinanderfolgenden Durchlauf der Hauptschleife, bei dem das Auto auf einer Geraden verortet wird, wird ein weiterer Befehl aus einer von mir erstellten und vorgegebenen Liste von Beschleunigungsbefehlen ausgeführt wird. Der Index, von dem der Befehl abzufragen ist, wird wieder auf 0 gesetzt, wenn das Auto in einem Durchlauf auf einer Kurve verortet wurde.

Befindet sich das Auto auf einer Kurve, folgt noch auf Basis der Nummer der Strecke, auf der sich das Auto befindet, und der Richtung der Kurve, die ebenfalls in `structures` gespeichert ist, eine Unterscheidung, ob es sich für dieses Auto um eine Innen- oder Außenkurve handelt. Hiervon hängt ab, welche der vordefinierten Bremsbefehlslisten das Programm zur Einleitung des in Kurven nötigen Abbremsmanövers nutzt (in einer Innenkurve muss stärker gebremst werden als in einer Außenkurve), das grundsätzliche Vorgehen ist aber analog zu dem auf geraden Strecken.

Außerdem findet sich in der Hauptschleife noch eine Abfrage, die das Auto des Gegners (sofern es erkannt wurde) trackt: So kann der Computer sein Auto anhalten, wenn der Gegner, z. B. nach einer zu schnell genommenen Kurve, aus der Bahn fliegt und auf der falschen Strecke landet.

Die Hauptschleife und damit die Fahrfunktion endet, wenn der Rundenwert eines der beiden Rundenzählerobjekte mit dem Zielrundenwert (standardmäßig fünf Runden) übereinstimmt. In diesem Fall wird noch eine kurze Tonfolge ausgegeben, während über die Lampen der Startampel angezeigt wird,

welches Fahrzeug gewonnen hat. Die Rundenzählerthreads werden gestoppt, und das computergesteuerte Fahrzeug kehrt automatisch zur Startlinie zurück.

4.4 Zwischenbilanz

Damit erfüllt das Programm alle Anforderungen: Es wurde ein Computergegner geschaffen, der auf beliebig umbaubaren Strecken gegen menschliche Spieler antreten kann – und zumindest ich habe große Schwierigkeiten gegen diesen Gegner zu gewinnen. Und selbst wenn andere dieses Programm schlagen, könnte es durch Anpassungen in den Beschleunigungs- und Bremsbefehlslisten verbessert werden. Natürlich wäre auch ein „Downgrade" möglich, um menschlichen Spielern bessere Chancen zu geben.

Aber genau hier liegt das größte Problem dieser einfachen Fahrroutine: Die Anpassung der Listen ist zwar einfach, durch diese Anpassungen aber spürbare Verbesserungen zu erzielen ist dagegen sehr schwierig und mit vielen Fallstricken gespickt. Zwar kann man argumentieren, das Auto würde effektiver fahren, indem man auf den geraden Strecken mehr beschleunigt. Probiert man dies tatsächlich aus, wird das Auto aufgrund der hohen Geschwindigkeit ziemlich sicher in der nächsten Kurve „den Abgang machen". Dem kann wiederum entgegengewirkt werden, indem man zu Beginn der Kurve stärker bremsen lässt. Aber hat sich die gesamte Anpassung dann überhaupt noch gelohnt, oder handelt es sich vielmehr um ein Nullsummenspiel? Erschwerend kommt hinzu, dass die Manöver nur rekursiv – vom Anfang der Streckenstruktur aus gedacht – programmiert werden können, das heißt: Man kann zwar zum Beispiel sehr leicht eine Beschleunigung am Anfang einer Geraden erwirken, aber kein Abbremsen am Ende, dies muss vielmehr zu Beginn einer Kurve erfolgen. Zum Zeitpunkt der Entwicklung dieser einfachen Fahrfunktion existierte die Funktion `structures2struklän()` im `Streckenberechner` noch nicht, sodass es tatsächlich nicht möglich war, Informationen über die restliche Länge der Streckenstruktur zu erhalten. Und selbst jetzt, da diese Möglichkeit besteht, wäre es schlicht viel zu aufwendig, solche Manöver im normalen Fahrmodus zu ergänzen, da dieser dafür quasi komplett neu entwickelt werden müsste.

5. Der Computer fängt an zu lernen

Wie gut der Computergegner ist, hängt davon ab, wie er abhängig von der Strecke beschleunigt und bremst. Statt feste Vorgaben zu nutzen, wie im einfachen Fahrmodus, soll der Computer nun selbstständig Beschleunigungs- und Bremsbefehlslisten erarbeiten und optimieren.

5.1 Lernen durch Zuschauen

Den ersten Schritt in diese Richtung unternahm ich mit dem manuell trainierbaren *Neural-Network*-Modus. Wie der Name schon impliziert, kann sich der Computer in diesem Modus noch nicht komplett selbst verbessern, die Verbesserung geschieht aber deutlich einfacher und effektiver als beim normalen Fahrmodus, nämlich durch bloßes „Zuschauen" bei einem menschlichen Spieler: Das Programm soll also den Fahrstil seiner menschlichen Kontrahenten lernen und leicht optimiert gegen sie einsetzen können – den allgemeinen Fahrstil wohlgemerkt, nicht die konkrete Fahrweise auf einer bestimmten Strecke, weil das ein neues Training für jede Strecke erforderlich machen würde, was dem Sinn meines Systems widerspräche. Dazu stellt sich erst mal die Frage, woraus denn der Fahrstil eines Menschen beim Carrera-Fahren besteht, sprich: Auf welcher Grundlage trifft er seine Entscheidungen? Zur Beantwortung dieser Frage konnte ich meine eigenen Erfahrungen heranziehen; und ich persönlich beachte beim Fahren folgendes:

- Befindet sich das Auto auf einer Geraden oder in einer Kurve?
- Wie lang ist der Streckenabschnitt insgesamt? Davon hängt ab, wie stark beschleunigt oder gebremst wird.
- Wo genau befindet sich das Auto auf dem aktuellen Streckenabschnitt? Davon hängt ab, ob schon für den nächsten Streckenabschnitt beschleunigt oder gebremst wird.
- Die aktuelle Geschwindigkeit des Autos

Indem nun das Programm einem menschlichen Fahrer dabei „zuschaut", wie er so gut fährt, wie er kann, und dabei diese Daten zusammen mit der in diesem Moment vom Menschen mit dem Handregler getroffenen Geschwindigkeitsentscheidung in einer Liste aufzeichnet, um damit später ein *Neural Network* zu trainieren, lernt es einen universell auf jeder Strecke anwendbaren Fahrstil von diesem Menschen, wird also so gut wie er – und durch eine gewisse Vorfilterung der Daten vor dem Trainieren des Netzwerks (z.B. nur die Daten aus Runden, in denen die Rundenzeit unter dem Durchschnitt lag, werden benutzt) wird der Computer auf Dauer sogar zwangsläufig besser als der Mensch, da er diese „optimalen Runden" – im Gegensatz zum Menschen – stets perfekt reproduzieren kann.

5.1.1 Daten sammeln

Auf Basis dieser Vorüberlegungen wurde eine `zuschauen()`-Funktion entwickelt. Diese startet mit einer kleinen Tastenkombination auf dem Bedienfeld (damit sie nicht aus Versehen ausgelöst wird,), und beginnt wieder mit der Startampel, damit der Fahrer Zeit hat, sich vorzubereiten, und dem Initialisieren eines Rundenzählers. Nach Ablauf des Countdowns startet eine ähnliche Schleife wie beim einfachen Fahrprogramm: Permanent wird die Position des Autos getrackt, im Plan

doi: 10.7795/320.201910

eingezeichnet und mit den in `Streckenberechner.strecke` gespeicherten Scanpunkten verglichen. Bei einer Übereinstimmung wird allerdings nicht die Geschwindigkeit neu geregelt, sondern die Trainingsliste ergänzt: Nämlich um die Art der Streckenstruktur (Innenkurve/Gerade/Außenkurve), auf der sich das zu beobachtende, vom Menschen gesteuerte Auto gerade befindet, die Gesamtlänge dieser Struktur und den an diesem Punkt davon bereits zurückgelegten Prozentsatz (beides zu finden in `Streckenberechner.strulän`), die aktuelle Geschwindigkeit des Autos (berechenbar über Zeit zwischen zwei Messungen und die Länge der dazwischenliegenden Vektoren) und – als vom Netzwerk zu lernender Zielwert – um die Spannung an der Strecke. Diese gibt nämlich der trainierende Mensch mit seinem Handregler vor und ist also die einzige Größe, in der sich die vom Menschen getroffene Fahrentscheidung wiederspiegelt, die das *Neural Network* später reproduzieren und übertragen soll.

Gemessen wird diese Spannung über einen analogen Eingang eines der Fischertechnik-Interfaces, dem der Spannungsteiler aus einem 5 kΩ und einem 10 kΩ Widerstand vorgeschaltet ist, um den für 10 V ausgelegten Controller nicht mit 14,8 V Carrera-Spannung zu belasten. Der gemessene Spannungswert (in mV) wird vor dem Anfügen an die Trainingsliste allerdings noch durch die maximal messbare Spannung geteilt, mit 512 multipliziert und das Ergebnis auf ganze Zahlen gerundet; das liegt daran, dass die PWM-Ausgänge der Controller, mithilfe derer diese Spannung später reproduziert werden können soll, mit 512 Schritten arbeiten und das zu trainierende *Neural Network* dementsprechend auch genau 512 Ausgangsklassen kennen soll, in deren Raster der Spannungswert also erst umgerechnet werden muss.

Diese Daten werden als Tupel an eine Rundenliste angehängt. Diese wird jedes Mal, wenn eine neue Runde beginnt (abfragbar über den zurückgesetzten Rundenzähler), um die für diese Runde benötigte Zeit ergänzt. Die Rundenliste wird an die Trainingsliste angehängt. Die Hauptschleife endet erst, wenn der Nutzer entscheidet, dass er genug Trainingsdaten angesammelt hat: Er unterbricht das Programm durch erneutes Drücken des Start-Tasters.

Nach dem Stopp der Zuschauschleife folgt die Filterung der Trainingsliste, bevor das Programm in der Interfaceschleife auf neue Befehle wartet: Durch die Filterung der Daten soll die Trainingsqualität deutlich verbessert werden. Dazu wird zunächst der Durchschnitt der eben erfassten Rundenzeiten berechnet, dann werden alle Rundendatenlisten, deren Rundenzeit-Eintrag über der Durchschnittszeit liegt, aus der Trainingsliste gelöscht, und schließlich wird die endgültige Trainingsliste aus den restlichen Rundendatenlisten unter Auslassung der Rundenzeit-Einträge gebildet. Diese Liste wird auch sofort für den späteren Gebrauch mit *TensorFlow* vorbereitet, indem sie automatisch in einer .csv Datei mit Header gespeichert wird, um sie z.B. für Experimente mit anderen Netzwerkparametern aufheben zu können.

5.1.2 Training des Netzwerks

Nach dem Sammeln der Daten mit der `zuschauen()`-Funktion startet das Training des Fahrstilreproduktions-*Neural Networks*: Dieses geschieht in einer eigenen `Neural_Network`-Klasse, die mit einem Namensparameter initialisiert wird, was die Nutzung mehrerer unterschiedlich trainierter Fahrnetzwerke in einem Programm ermöglicht. In der Funktion `trainieren()` dieser Klasse werden zuerst die eben gespeicherten Daten wieder eingelesen und in *TensorFlow*-Datasets konvertiert, wozu das Framework praktischerweise direkt passende Funktionen mitbringt. Dann wird das eigentliche Netzwerk aufgesetzt: Diesmal um einiges größer als das Kurvenerkennungs-Netzwerk, mit zurzeit fünf Schichten mit 100, 200, 200, 200 und 300 Neuronen sowie 3 Eingangsparametern (Art der Streckenstruktur, Gesamtlänge der Streckenstruktur, davon schon zurückgelegt in Prozent) und 512 Ausgangsklassen (die die 512 Stufen der PWM-Regelung symbolisieren). Für diese Netzwerkparameter habe ich mich nach einer Reihe kurzer Tests entschieden. Ich habe das kleinste Netzwerk genommen, das gerade so in der Lage war, mit den gegebenen Daten vernünftig aussehende Geschwindigkeitswerte zu berechnen, da größere Netzwerke naturgemäß auch mehr Trainingsdaten erfordern. Der Code dieser Klasse ist teilweise übernommen aus Beispielcode auf der *TensorFlow*-Homepage [6].

Die Geschwindigkeit wird nicht als Eingangsparameter benutzt, weil die Einbeziehung der aktuellen Geschwindigkeit in die Berechnung der nächsten Aktion nur dann sinnvoll ist, wenn diese Berechnung in Echtzeit stattfindet, sprich: Auch im späteren Fahrbetrieb stets mit der aktuellen Geschwindigkeit gerechnet werden kann. Das ist nicht der Fall, da *TensorFlow* für die Auswertung eines *Neural Networks* mit neuen Daten bei meiner Implementierung stets ca. 0,5 Sekunden braucht, was für Echtzeitberechnungen während der Fahrt viel zu lang ist. Bis ich eine Lösung für dieses Problem finde, muss ich daher mit einer „abgespeckten“ Version des *Neural-Network*-Fahrmodus Vorlieb nehmen: Die Berechnungen werden dabei nicht mit aktuellen Werten während der Fahrt durchgeführt, sondern vorher, direkt nach dem Einscannen der Strecke. Dabei wird für jeden Scanpunkt ein Spannungswert berechnet, den der Computer nur noch mit dem Verstärker ausführen muss, wenn er das Auto später während der Fahrt an diesem Punkt verortet; da die aktuelle Geschwindigkeit zu dieser Zeit natürlich vor der Fahrt nicht genau bekannt ist, bleibt sie auch bei den Berechnungen außen vor.

Nach dem Definieren des Netzwerks wird es unter Zuhilfenahme der pas-

senden *TensorFlow*-Funktion mit dem Trainings-*Dataset* trainiert, und dann mithilfe der Evaluationsliste ausgewertet: Dafür werden die Berechnungen des Netzwerks auf Basis der Eingangsdaten aus der Liste mit den tatsächlichen Ausgangsdaten aus der Liste verglichen, um einen Genauigkeitswert zu errechnen, mit dessen Hilfe abgeschätzt werden kann, wie gut das Netzwerk trainiert ist oder generell arbeitet. Das fertige *Neural-Network*-Modell wird wieder in einem Ordner gespeichert, dessen Name vom Namensparameter der Klasse abhängig ist.

Anschließend kann es sofort erstmals ausgewertet werden: In der Funktion `Neural_Network.anwenden()` wird, wie schon erwähnt, auf Basis dieses Netzwerks für jeden Scanpunkt ein PWM-Wert von 0 bis 512 berechnet und in einer Liste gespeichert, und zwar für jede der beiden parallelen Strecken einzeln, da sich die Streckenstrukturarten, auf Basis derer das Netzwerk schließlich seine Entscheidungen trifft, zwischen diesen beiden unterscheiden: Wo eine Rechtskurve auf der einen Strecke eine Innenkurve bedeutet, bedeutet sie auf der anderen Strecke eine Außenkurve, was in Bezug auf die an dieser Stelle zu wählende Geschwindigkeit durchaus relevant ist. Die `Neural_Network.anwenden()`-Funktion startet jedes Mal nach dem Scannen der Strecke (falls zu dem Zeitpunkt ein entsprechendes Netzwerkmodell vorhanden ist), um die eventuell umgebaute Strecke auch direkt mit dem *Neural Network* nutzen zu können: Dieses ist schließlich nicht auf einen speziellen Streckenaufbau beschränkt, sondern hat einen universellen Fahrstil gelernt, den es auf jeder Strecke anwenden kann.

5.1.3 Der *Neural-Network*-Fahrmodus

Damit ist alles bereit für den eigentlichen *Neural-Network*-Fahrmodus: Wenn die Schalter am Bedienpanel entsprechend eingestellt sind, ein Streckenscan und ein Fahrnetzwerkmodell vorhanden sind und der Startknopf gedrückt wird, wird der Rundenzähler auf fünf Runden zurückgesetzt, die Startampel ausgelöst, nach Ablauf dieser die Strecken freigegeben und die `Neural_Network.fahren()`-Funktion gestartet, die die Kontrolle über das vom Computer zu steuernde Auto übernimmt. Diese ist der normalen `fahren()`- oder der `zuschauen()`-Funktion bis auf wenige Ausnahmen sehr ähnlich, weshalb ich hier nicht mehr allzu sehr in die Details gehen muss. Es gibt wieder eine Hauptschleife, in der immer wieder mithilfe von `Tracking.getPos()` die Position des vom Computer gesteuerten autos erfasst wird, die dann mit den gescannten Punkten verglichen wird. Wenn eine Übereinstimmung gefunden wurde, wird der Controllerausgang, der über den Verstärker das Auto steuert, auf den Wert gesetzt, der am Index der Übereinstimmung in der von `Neural_Network.anwenden()` berechneten Liste von *Neural-Network*-Entscheidungen für diese Strecke steht. So wird diese vorherberechnete Entscheidung in die Fahrpraxis umgesetzt. Neben einer Funktion, die den Computer bremsen lässt, wenn der Gegner auf der falschen Strecke landet, folgt hier auch noch eine Abfrage der Position des Gegnerfahrzeugs auf der *richtigen* Strecke, um mehr Gas geben zu können, je weiter der Gegner vorne liegt. So wird etwas mehr Dynamik in die Rennen gebracht. Da der Computer nur durch das manuelle Training noch längst nicht an die Grenze der maximal in den Kurven fahrbaren Geschwindigkeit stößt, führt diese Funktion auch quasi nie zu Problemen.

Die Positionsdaten des Gegnerfahrzeugs werden noch für eine weitere Funktion genutzt, die die Stärke des Konzepts, nämlich das Lernen vom Gegner, fördern soll: Diese Fahrdaten werden hier genau wie in der `zuschauen()`-Funktion gespeichert und, falls der Mensch gewinnt, sofort zum Training des verwendeten Fahrstil-*Neural-Networks* verwendet. Die Hauptschleife wird wieder unterbrochen, sobald einer der Rundenzähler sein Soll erreicht hat, also ein Fahrzeug am Ziel ist – und mit einem gut trainierten Netzwerk sollte das spätestens jetzt meistens das vom Computer gesteuerte sein.

Da der *Neural-Network*-Modus nur von seinem menschlichen Trainer lernt, ist es hier leicht möglich, ihn für die eigenen Bedürfnisse zu trainieren, um nicht permanent gegen einen viel zu gut fahrenden Computer zu verlieren. Gerade zur kontinuierlichen Verbesserung des eigenen Fahrstils ist dieser Modus sogar besonders gut geeignet: Der Computer „wächst“ schließlich automatisch mit seinem menschlichen Gegner (und Trainer) mit, wodurch jedes einzelne Rennen spannend bleibt, sofern man ihn eben selbst trainiert und nicht ein viel zu gutes, durch viele Fahrer trainiertes Netzwerk benutzt.

5.2 Lernen mit einem evolutionären Algorithmus

Der in 5.1 entwickelte manuelle *Neural-Network*-Modus erfüllt seinen Zweck recht gut und ist einfach zu trainieren. Dennoch ist er noch keine optimale Lösung, die dem Potenzial eines mit aktuellen Techniken der Informatik geschriebenen Programms gerecht wird. Vor allem hat er ein großes Manko: Ein mit ihm trainiertes *Neural Network* kann, trotz aller Vorfilterungen etc., niemals *viel* besser fahren als der Mensch, der es trainiert hat. Besser wäre eine Lösung, bei der sich der Computer nach einer Art *Trial-and-error*-Prinzip selbst verbessert, und zwar auch nicht auf eine bestimmte Strecke bezogen, sondern wieder auf einen allgemeinen Fahrstil.

Eine solche Lösung wurde mit dem evolutionären *Neural-Network*-Modus implementiert. Dieser nutzt Prinzipien aus der natürlichen Evolution, um eine ganze Population von Fahrstil-*Neural-Networks* (die alle so funktionieren wie in 5.1 beschrieben) über viele Generationen Schritt für Schritt weiterzuentwickeln. Irgendwann sollte ein Punkt

doi: 10.7795/320.201910

erreicht sein, ab dem kaum noch nennenswerte Verbesserungen spürbar sind, d. h. ein quasi optimales Fahrstil-*Neural-Network* gefunden wurde, welches künftig auf jeder Strecke eingesetzt werden kann. In meinem Programm nutze ich dafür einen genetischen Algorithmus mit einem *single-point Crossover*.

Das „evolutionäre Training" findet in der Klasse `Population` statt, die mit dem Parameter „Pfad" (Ordner, in dem die *Neural Networks* und deren Trainingslisten gespeichert werden sollen) initialisiert wird. Alle anderen benötigten Daten (Größe der Population, Nummer der Generation, höchste bisher erreichte Fitness) werden aus einer kleinen Datei in diesem Ordner gelesen. Das hat den Vorteil, dass das langwierige Training ohne Probleme unterbrochen und später an derselben Stelle wieder fortgesetzt werden kann, oder dass mehrere Populationen z. B. mit unterschiedlichen Größen parallel gepflegt werden können.

Außerdem liegt in diesem Ordner noch eine Datei namens training_x.csv: Diese entspricht einer Trainingsliste für den manuellen *Neural-Network*-Modus unter Auslassung der Spalte mit den PWM-Werten, enthält also nur eine Sammlung von Streckendaten (optimaler-, aber nicht zwingenderweise von der Strecke, auf der auch diese Population trainieren soll). Diese Liste ist nötig, weil die *Neural Networks* der Population mit „echten" Streckendaten trainieren sollen, diese sollten also nicht zufällig generiert sein.

Soll eine neue Population angelegt werden, geschieht das durch Bereitstellung eines solchen Ordners mit den nötigen Dateien und dem Aufruf der Funktion `Population.randomize()`. Diese liest die gewünschte Populationsgröße ein und erstellt entsprechend viele *Neural-Network*-Objekte, deren Trainingslisten sich aus den Daten aus training_x.csv und einer vollkommen zufällig generierten Spalte von PWM-Werten, die für jedes Netzwerk neu generiert wird, zusammensetzen. Außerdem werden die *Neural Networks* auch direkt mit diesen Trainingslisten erstmals trainiert.

Als nächstes werden die so generierten *Neural Networks* getestet. Dafür darf jedes Netzwerk der Population das Auto eine Runde lang fahren, wobei jeweils die Zeit gestoppt wird. Dabei kommt es anfangs oft vor, dass das Auto rausfliegt oder stehenbleibt: In diesem Fall muss ein Mensch eingreifen und zusätzlich einen Taster betätigen, der die von diesem *Neural Network* für die Testrunde benötigte Zeit um mehr als eine Stunde erhöht, damit das Netzwerk für diesen Fehler den verdienten evolutionären Nachteil bekommt. Durch ihre vollkommen zufällige Initialisierung erfüllen nun manche *Neural Networks* der Population ihre Aufgabe besser als andere, andere bleiben stehen oder fliegen aus der Kurve. Nach Abschluss der Testrunden werden alle diese Netzwerke nach ihrer benötigten Zeit bewertet.

Und hier setzt das erste evolutionäre Prinzip hinter diesem Training an, die Selektion: Je länger ein *Neural Network* für seine Testrunde gebraucht hat, desto seltener findet es sich im „Genpool" für die nächste Generation wieder. Konkret fliegen die zwei schlechtesten Kandidaten sogar direkt raus, das drittschlechteste landet einmal im Genpool, das viertschlechteste zweimal und so weiter. Die zwei besten werden aussortiert, allerdings nur zu dem Zweck, dass diese sich im nächsten Schritt auf jeden Fall untereinander „paaren" dürfen, weil es zu schade wäre, wenn ihre „Gene" durch unglückliche Zufälle verloren gingen, das könnte die gesamte Population wieder um einige Generationen zurückwerfen. Aus diesem Genpool werden nun zufällig Zweierpaare gezogen, bis die Zahl der gezogenen *Neural Networks* – zusammen mit den eben aussortierten besten *Neural Networks* – wieder der gewünschten Populationsgröße entspricht.

Nachdem der „Genpool" für die nächste Generation aufgestellt und die Elternkandidaten ausgewählt sind, müssen sich diese nur noch „kreuzen". Das geschieht nach einem sehr einfachen Prinzip: Es werden jeweils zwei *Neural-Network*-Objekte aus dem Elternpool zufällig ausgewählt und die PWM-Spalten ihrer Trainingslisten (also der einzige Punkt, in dem sie sich ursprünglich unterscheiden), an einer gemeinsamen zufälligen Stelle durchgeschnitten und gekreuzt wieder zusammengefügt, sprich: Eine der entstehenden Listen hat den Anfang von einem Elternteil und das Ende vom anderen, die andere umgekehrt. So entsteht eine Kinder-Generation, die in ihrer Größe identisch mit der Elterngeneration ist.

Um ab und zu mal wieder ein paar „frische Gene" ins Spiel zu bringen, fehlt noch ein wichtiger Bestandteil der natürlichen Evolution: Die Mutation. Dafür werden die PWM-Spalten aller Kind-Trainingslisten Element für Element durchgegangen, und mit einer jeweils neu gewürfelten 2-prozentigen Wahrscheinlichkeit mutiert das jeweilige Element noch einmal zu einer vollkommen zufälligen Zahl. Diese Maßnahme ist nötig, weil sonst nach wenigen Generationen kaum noch Verbesserung möglich wäre, einfach weil das nötige „Genmaterial" fehlt: Erfolgreiche *Neural Networks* werden nach meinem System dermaßen übervorteilt, dass sie schon bald alle anderen verdrängen, obwohl sie noch längst nicht perfekt sind und dringend noch ein wenig „neuen Schwung" in Form von neuen Trainingsdaten brauchen, die nun die Mutation liefert.

Zum Schluss werden die *Neural-Network*-Objekte der Elterngeneration gelöscht und durch die der Kind-Generation ersetzt, und nachdem diese trainiert sind, können sie auch schon wieder getestet werden – eine neue Generation ist fertig.

Auf den entsprechenden Fahrmodus, der das jeweils beste *Neural Network* einer Population im Rennen einsetzt,

doi: 10.7795/320.201910

muss ich nicht eingehen, da dieser absolut identisch zu seinem Pendant im manuellen *Neural-Network*-Modus ist. Als „Bestes *Neural Network*“ wird nicht das Beste der aktuellen Generation ausgewählt, sondern es wird generationsübergreifend bestimmt, falls in einer Generation zufällig eine Verschlechterung stattgefunden hat, was aufgrund der Arbeitsweise von evolutionären Algorithmen wie diesem recht häufig vorkommt.

6. Weitere Funktionen der Anlage

6.1 Der Übungsmodus

Eine oft vermisste Funktion, bei der der menschliche Fahrer „einfach mal üben“ kann, startet, wenn der rechte Schalter in der rechten Position steht; in diesem Fall werden alle Relais in Richtung Handbetrieb geschaltet. Von diesem Modus aus kann mit einem Druck auf den Starttaster die Funktion `menschenrennen()` gestartet werden: Diese besteht aus dem üblichen Block aus Startampel und einer Schleife, die endet, wenn ein Rundenzähler sein Soll erreicht. Hier werden allerdings beide Autos mit verschiedenen Handreglern von menschlichen Fahrern gesteuert – falls man eben doch mal Mitspieler findet.

6.2 Hilfsfunktionen

Mit einer kleinen Tastenkombination lässt sich auf dem Bedienfeld eine Funktion starten, die die Aufgabe hat, das manuelle Fahrstil-*Neural-Network* neu zu trainieren, allerdings mit derselben Trainingsdatenliste. Das rührt daher, dass *TensorFlow* leider nicht deterministisch arbeitet, also aufgrund des Anfangszufalls in den Netzwerken aus den gleichen Trainingsdaten meist unterschiedliche Netzwerke generiert – auch an anderer im Projekt ein großes Problem, welches sich auch mit dem kleinen Trick nicht lösen ließ, dass nur einmal ein Netzwerk wirklich neu erstellt wird, dessen Kopien dann als Grundlage für alle späteren dienen, weil in der vorgefertigten `tf.estimator.DNNClassifier`-Klasse offenbar auch beim Training selbst noch der Zufall eine Rolle spielt. Deshalb stellt diese Funktion einen kleinen workaround dar: Wenn man einmal beim Training zufällig ein scheinbar schlechtes *Neural Network* erwischt, muss man nicht gleich alle Daten erneut sammeln, sondern kann hoffen, dass die Lage nach einem erneuten Training mit denselben Daten besser aussieht.

6.3 Eine Smartphone Anbindung

Und weil heutzutage kein Spielzeug, welches mit High-Tech punkten will, ohne zumindest rudimentäre Smartphone-Funktionen auskommt, habe ich auch meine Carrera-Anlage damit ausgerüstet: Allerdings nicht mit einer aufwendigen, eigenen App, sondern einem kleinen *Telegram-Bot*, welcher für meine Zwecke voll ausreicht.

Telegram ist ein Messenger-Dienst wie WhatsApp mit einer sehr umfangreichen Schnittstelle für *Bots*, also Programme, die mit dem Nutzer interagieren können. *Telegram* erlaubt es *Bots* unter anderem, dem Nutzer kleine Buttons in einer Konversation zu senden, die bei Betätigung jederzeit eine bestimmte Aktion auslösen. Meine „Smartphone-Anbindung“ besteht dementsprechend größtenteils daraus, dass die Scan-, Start- und Stopp-Buttons nicht nur physisch am Bedienfeld, sondern auch jederzeit virtuell in *Telegram* parat stehen, außerdem kann sich der Nutzer auf Wunsch den aktuell berechneten Streckenplan sowie Status-Updates zum aktuellen Fortschritt z. B. eines evolutionären Trainings zusenden lassen.

7. Diskussion und Ausblick

Mit dem manuellen *Neural-Network*-Modus erfüllt die vorgestellte Lösung alle eingangs gesteckten Ziele: Es wurde ein Computergegner entwickelt, der auf beliebigen Strecken, die sich umbauen lassen, ohne dass das Programm neu gestartet werden muss, sehr effektiv gegen menschliche Spieler antreten kann. Dazu wurde eine Hardwarekomponente entwickelt, die die Befehle des Programms auf die Anlage überträgt, zahlreiche Komfortfunktionen wie eine automatische Umschaltung der Strecken zwischen menschlicher- und Computersteuerung oder Verhinderung von Frühstarts unterstützt sowie nützliche Hilfen wie eine Startampel oder ein Bedienpanel am Gerät zur Verfügung stellt. Mit dem evolutionären *Neural-Network*-Modus wurde eine Lösung vorgestellt, die die Anforderungen noch übertrifft, indem das Programm nicht nur lernen kann, besser als der Mensch zu fahren, sondern – zumindest theoretisch – *perfekt*.

Gerade hier besteht aber auch noch einiges an Optimierungsbedarf: Bevor ein evolutionär trainiertes *Neural Network* nämlich so gut fährt wie ein manuell trainiertes, müssen einige Generationen vergehen: Meine „fortschrittlichste“ Population von 50 Netzwerken befindet sich in der 40. Generation. Außerdem werde ich mich noch mit dem Einfluss von Parametern wie Populationsgröße und Mutationswahrscheinlichkeit auf den Trainingserfolg auseinandersetzen. Da jedes Netzwerk einzeln auf der echten Strecke getestet werden muss (was auch bedeutet, dass stets ein Mensch anwesend sein muss, der das Auto wieder in die Bahn setzt, falls das Netzwerk versagt hat), ist der Aufwand für das evolutionäre Training sehr hoch. Daher habe ich in meinen bisherigen Experimenten nur mit sehr kleinen Populationsgrößen von maximal 60 Netzwerken gearbeitet. Die entsprechend geringe Größe des Genpools versuche ich durch die sehr hohe Mutationswahrscheinlichkeit von 60 Prozent auszugleichen.

Aber auch der verwendete evolutionäre Algorithmus ist sicher längst noch nicht perfekt: Hier wären sicher auch Experimente mit ganz anderen Strategien in der Selektion und der Mutation der

doi: 10.7795/320.201910

Netzwerke möglich, z. B. mit einer Plus-Selektion, die die besten Netzwerke einer Generation unverändert überleben lässt, oder einer Mutation, die das entsprechende Element nicht komplett zufällig neu wählt, sondern nur eine normalverteilte Zufallszahl addiert und so den Großteil der Mutationen in einem bestimmten Bereich hält. Mit derartigen Veränderungen habe ich bisher noch nicht experimentiert, da ich hierfür, um entsprechende Vergleiche mit anderen Strategien herstellen zu können, eine komplett neue Population trainieren müsste, mit dem entsprechenden Aufwand durch die Auswertung der einzelnen Netzwerke auf der Strecke.

Sowohl im evolutionären als auch im manuellen *Neural-Network*-Modus sind noch längst nicht die optimalen Abmessungen (Zahl der Schichten und Zahl der Neuronen pro Schicht) für ein Fahrstil-*Neural Network* gefunden, das nicht allzu schwerfällig lernt und trotzdem möglichst gut einen Fahrstil wiedergibt. Ein großes Problem, welches ich eben schon angesprochen habe, macht sich hier im evolutionären Modus außerdem besonders schwer bemerkbar: Die von mir verwendeten *TensorFlow*-Funktionen arbeiten – anders als anfangs angenommen – nur aufgrund der Trainingsdaten *nicht deterministisch*, was zu der Frage führt, ob mein evolutionäres Training überhaupt jemals zu sinnvollen Ergebnissen führen kann, da ich nur die Trainingsdaten kreuze und auf Basis dessen auf Verbesserung der Netzwerke hoffe – was eventuell vergebens, mindestens aber deutlich langwieriger ist, wenn bei der Erstellung des Netzwerks aus den Daten noch einmal massiv der Zufall mitmischt. Dieses Problem ließe sich vielleicht durch die Nutzung der *TensorFlow-LowLevel-API* beheben, was ich allerdings nicht mehr implementieren konnte, da ich den Fehler erst sehr spät bemerkt habe.

Der manuelle *Neural-Network*-Modus hat damit nur bedingt zu kämpfen und gilt deshalb als mein bisher bester Fahrmodus: Ihm käme es sehr zugute, wenn er einmal gründlich von jemandem trainiert werden würde, der deutlich besser fährt als ich, damit das Konzept seine Stärken voll ausspielen kann. Ein großer Schub für die Entwicklung wäre auch eine Möglichkeit, die *Neural-Network*-Berechnungen beim Fahren in Echtzeit ausführen und so z. B. die aktuelle Geschwindigkeit einbeziehen zu können, was ebenfalls die Ergebnisse deutlich verbessern sollte, da der Computer so erstmals tatsächlich dynamisch auf die Situation reagieren könnte. In diesem Fall könnten auch ganz andere Netzwerkarchitekturen in Betracht gezogen werden, z. B. ein rekursives *Neural Network*, das nicht nur die aktuelle Streckensituation, sondern auch zuvor getroffene Entscheidungen in die Berechnung mit einbezieht, was im Kontext eines Rennens durchaus sinnvoll erscheint. Eventuell hätte ich dafür aber auf ein anderes *Deep-Learning-Framework* umsatteln müssen.

Ein weiterer bei Vorführungen meines Projekts oft geäußerter, aber nicht sinnvoll behebbarer Kritikpunkt sind die deutlichen Beschränkungen in Bezug auf den Aufbau der Strecke: Steilkurven, Engstellen, Kreuzungen und Loopings gehören für viele offenbar untrennbar zu einer Carrera-Anlage dazu, können mit meiner Tracking-Methode aber nicht erfasst und damit auch nicht im Programm implementiert werden.

Danksagung

Das Netzgerät wurde mir dankenswerter Weise von meiner Schule, dem Johannes-Gymnasium in Lahnstein, zur Verfügung gestellt. Weitere Kleinteile ,wie Widerstände, Transistorverstärker, usw. wurden von der Schule finanziert.

Ebenfalls danken möchte ich der Firma Fischertechnik GmbH, die mir einen zweiten Controller sponserte, dessen zusätzliche Ein- und Ausgänge für einen praxistauglichen Ausbau der Hardware unerlässlich waren. Einige der für den Kameraturm benötigten Teile wurden mir ebenfalls von Fischertechnik kostenlos zur Verfügung gestellt.

Zu erwähnen ist noch die freundliche Unterstützung der Firma Stadlbauer (Hersteller der Carrera-Produkte), die mir drei Fahrzeuge zukommen ließ.

Ein großer Dank geht an Prof. Dr. Helmut Bollenbacher von der Hochschule Koblenz, der mir bei der Verbesserung meiner Verstärker mit neuen Transistoren half.

Quellen:

[1] Verwendete Python-Bibliotheken:
[2] OpenCV (https://www.opencv.org/)
[3] Imutils (https://www.github.com/jrosebr1/imutils)
[4] TensorFlow (https://www.tensorflow.org/)
[5] ftrobopy (https://www.github.com/ftrobopy)
[6] Teile des Programmcodes bauen auf folgenden Programmbeispielen auf:
[7] Tracking-Funktion: https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/#
[8] Neural Network-Trainings-Funktion: https://www.tensorflow.org/get_started/estimator
(ältere Version, aufgerufen August 2017)

doi: 10.7795/320.201910

Publiziere auch Du hier!

FORSCHUNGSARBEITEN VON SCHÜLER/INNE/N UND STUDENT/INN/EN

In der Jungen Wissenschaft werden Forschungsarbeiten von SchülerInnen, die selbstständig, z. B. in einer Schule oder einem Schülerforschungszentrum, durchgeführt wurden, veröffentlicht. Die Arbeiten können auf Deutsch oder Englisch geschrieben sein.

Wer kann einreichen?

SchülerInnen, AbiturientInnen und Studierende ohne Abschluss, die nicht älter als 23 Jahre sind.

Was musst Du beim Einreichen beachten?

Lies die Richtlinien für Beiträge. Sie enthalten Hinweise, wie Deine Arbeit aufgebaut sein soll, wie lang sie sein darf, wie die Bilder einzureichen sind und welche weiteren Informationen wir benötigen. Solltest Du Fragen haben, dann wende Dich gern schon vor dem Einreichen an die Chefredakteurin Sabine Walter.

Lade die Erstveröffentlichungserklärung herunter, drucke und fülle sie aus und unterschreibe sie.

Dann sende Deine Arbeit und die Erstveröffentlichungserklärung per Post an:

Chefredaktion Junge Wissenschaft
Dr.-Ing. Sabine Walter
Paul-Ducros-Straße 7
30952 Ronnenberg
Tel: 05109 / 561508
Mail: sabine.walter@verlag-jungewissenschaft.de

Wie geht es nach dem Einreichen weiter?

Die Chefredakteurin sucht einen geeigneten Fachgutachter, der die inhaltliche Richtigkeit der eingereichten Arbeit überprüft und eine Empfehlung ausspricht, ob sie veröffentlicht werden kann (Peer-Review-Verfahren). Das Gutachten wird den Euch, den AutorInnen zugeschickt und Du erhältst gegebenenfalls die Möglichkeit, Hinweise des Fachgutachters einzuarbeiten.

Die Erfahrung zeigt, dass Arbeiten, die z. B. im Rahmen eines Wettbewerbs wie **Jugend forscht** die Endrunde erreicht haben, die besten Chancen haben, dieses Peer-Review-Verfahren zu bestehen.

Schließlich kommt die Arbeit in die Redaktion, wird für das Layout vorbereitet und als Open-Access-Beitrag veröffentlicht.

Was ist Dein Benefit?

Deine Forschungsarbeit ist nun in einer Gutachterzeitschrift (Peer-Review-Journal) veröffentlicht worden, d. h. Du kannst die Veröffentlichung in Deine wissenschaftliche Literaturliste aufnehmen. Deine Arbeit erhält als Open-Access-Veröffentlichung einen DOI (Data Object Identifier) und kann von entsprechenden Suchmaschinen (z. B. BASE) gefunden werden.

Die Junge Wissenschaft wird zusätzlich in wissenschaftlichen Datenbanken gelistet, d. h. Deine Arbeit kann von Experten gefunden und sogar zitiert werden. Die Junge Wissenschaft wird Dich durch den Gesamtprozess des Erstellens einer wissenschaftlichen Arbeit begleiten – als gute Vorbereitung auf das, was Du im Studium benötigst.

doi: 10.7795/320.201910

Richtlinien für Beiträge

FÜR DIE MEISTEN AUTOR/INN/EN IST DIES DIE ERSTE WISSENSCHAFTLICHE VERÖFFENTLICHUNG. DIE EINHALTUNG DER FOLGENDEN RICHTLINIEN HILFT ALLEN – DEN AUTOR/INNEN/EN UND DEM REDAKTIONSTEAM

Die Junge Wissenschaft veröffentlicht Originalbeiträge junger AutorInnen bis zum Alter von 23 Jahren.

- Die Beiträge können auf Deutsch oder Englisch verfasst sein und sollten nicht länger als 15 Seiten mit je 35 Zeilen sein. Hierbei sind Bilder, Grafiken und Tabellen mitgezählt. Anhänge werden nicht veröffentlicht. Deckblatt und Inhaltsverzeichnis zählen nicht mit.

- Formulieren Sie eine eingängige Überschrift, um bei der Leserschaft Interesse für Ihre Arbeit zu wecken, sowie eine wissenschaftliche Überschrift.

- Formulieren Sie eine kurze, leicht verständliche Zusammenfassung (maximal 400 Zeichen).

- Die Beiträge sollen in der üblichen Form gegliedert sein, d. h. Einleitung, Erläuterungen zur Durchführung der Arbeit sowie evtl. Überwindung von Schwierigkeiten, Ergebnisse, Schlussfolgerungen, Diskussion, Liste der zitierten Literatur. In der Einleitung sollte die Idee zu der Arbeit beschrieben und die Aufgabenstellung definiert werden. Außerdem sollte sie eine kurze Darstellung schon bekannter, ähnlicher Lösungsversuche enthalten (Stand der Literatur). Am Schluss des Beitrages kann ein Dank an Förderer der Arbeit, z. B. Lehrer und Sponsoren, mit vollständigem Namen angefügt werden. Für die Leser kann ein Glossar mit den wichtigsten Fachausdrücken hilfreich sein.

- Bitte reichen Sie alle Bilder, Grafiken und Tabellen nummeriert und zusätzlich als eigene Dateien ein. Bitte geben Sie bei nicht selbst erstellten Bildern, Tabellen, Zeichnungen, Grafiken etc. die genauen und korrekten Quellenangaben an (siehe auch Erstveröffentlichungserklärung). Senden Sie Ihre Bilder als Originaldateien oder mit einer Auflösung von mindestens 300 dpi bei einer Größe von 10 · 15 cm! Bei Grafiken, die mit Excel erstellt wurden, reichen Sie bitte ebenfalls die Originaldatei mit ein.

- Vermeiden Sie aufwendige und lange Zahlentabellen.

- Formelzeichen nach DIN, ggf. IUPAC oder IUPAP verwenden. Gleichungen sind stets als Größengleichungen zu schreiben.

- Die Literaturliste steht am Ende der Arbeit. Alle Stellen erhalten eine Nummer und werden in eckigen Klammern zitiert (Beispiel: Wie in [12] dargestellt …). Fußnoten sieht das Layout nicht vor.

- Reichen Sie Ihren Beitrag sowohl in ausgedruckter Form als auch als PDF ein. Für die weitere Bearbeitung und die Umsetzung in das Layout der Jungen Wissenschaft ist ein Word-Dokument mit möglichst wenig Formatierung erforderlich. (Sollte dies Schwierigkeiten bereiten, setzen Sie sich bitte mit uns in Verbindung, damit wir gemeinsam eine Lösung finden können.)

- Senden Sie mit dem Beitrag die Erstveröffentlichungserklärung ein. Diese beinhaltet im Wesentlichen, dass der Beitrag von dem/der angegebenen AutorIn stammt, keine Rechte Dritter verletzt werden und noch nicht an anderer Stelle veröffentlicht wurde (außer im Zusammenhang mit **Jugend forscht** oder einem vergleichbaren Wettbewerb). Ebenfalls ist zu versichern, dass alle von Ihnen verwendeten Bilder, Tabellen, Zeichnungen, Grafiken etc. von Ihnen veröffentlicht werden dürfen, also keine Rechte Dritter durch die Verwendung und Veröffentlichung verletzt werden. Entsprechendes Formular ist von der Homepage www.junge-wissenschaft.ptb.de herunterzuladen, auszudrucken, auszufüllen und dem gedruckten Beitrag unterschrieben beizulegen.

- Schließlich sind die genauen Anschriften der AutorInnen mit Telefonnummer und E-Mail-Adresse sowie Geburtsdaten und Fotografien (Auflösung 300 dpi bei einer Bildgröße von mindestens 10 · 15 cm) erforderlich.

- Neulingen im Publizieren werden als Vorbilder andere Publikationen, z. B. hier in der Jungen Wissenschaft, empfohlen.

doi: 10.7795/320.201910

Impressum

Junge Wissenschaft
c/o Physikalisch-Technische
Bundesanstalt (PTB)
www.junge-wissenschaft.ptb.de

Redaktion
Dr. Sabine Walter, Chefredaktion
Junge Wissenschaft
Paul-Ducros-Str. 7
30952 Ronnenberg
E-Mail: sabine.walter@verlag-jungewissenschaft.de
Tel.: 05109 / 561 508

Verlag
Dr. Dr. Jens Simon,
Pressesprecher der PTB
Bundesallee 100
38116 Braunschweig
E-Mail: jens.simon@ptb.de
Tel.: 0531 / 592 3006
(Sekretariat der PTB-Pressestelle)

Design & Satz
Sabine Siems
Agentur „proviele werbung“
E-Mail: info@proviele-werbung.de
Tel.: 05307 / 939 3350