Erkennung von Fehlern bei Halteübungen mittels Faltungsnetz

Dipl.-Kfm. Andreas Fabian Balck
Department Informatik, Hochschule für angewandte Wissenschaften Hamburg
andreas.balck@haw-hamburg.de

Zusammenfassung

Die richtige Durchführung von Halteübungen wird mittels Raspberry Pi Kamera und einem Faltungsnetz überwacht. Das Bild der Pi-Kamera wird in Echtzeit ausgewertet. Dadurch wird erkannt, ob eine Person im Bildausschnitt zu sehen ist und ob diese Person bestimmte Halteübungen durchführt. Das Netz unterscheidet dabei zwischen 6 Klassen (keine Person, Stehen, Plank, Plank falsch, Herabschauender Hund und Herabschauender Hund falsch). Der/die Sportler/in wird durch eine Bildschirmausgabe auf eventuelle Fehler bei der Durchführung der Halteübungen hingewiesen. Beispielsweise wird ein krummer Rücken oder eine falsche Positionierung der Arme erkannt.

Keywords: CNN; Maschinelles Lernen; Raspberry Pi; Faltungsnetze; inference

1 Einleitung

1.1 Motivation

Die inkorrekte Ausführung von Sportübungen kann zu Schädigungen führen. Dies gilt insbesondere auch für Halteübungen, wie z.B. bei einer Plank (Unterarmstütz) (siehe [3]). Daher ist besonders bei unerfahrenen Sportler:innen eine professionelle Überwachung durch einen Trainer:in ratsam. Eine derartige Überwachung kann allerdings nicht immer gewährleistet werden, sodass eine kostengünstige und simple Variante mittels künstlicher Intelligenz eine gute Alternative darstellen könnte.

In einer vorangegangenen Arbeit an der HAW wurde untersucht wie bei der Ausführung von bestimmten Sportübungen die Wiederholungen durch maschinelles Lernen gezählt werden können (siehe [2]). Im Gegensatz dazu befasst sich diese Arbeit mit Übungen, in denen wenig Bewegung stattfindet und die richtige Körperhaltung geprüft werden soll.

1.2 Ziel

Im Rahmen dieses Projektes wurde untersucht wie ein Neuronales Netz eingesetzt werden kann, um anhand von Bildern zu erkennen, ob bestimmte Halteübungen korrekt ausgeführt werden. Dabei soll erkannt werden, ob sich eine Person vor der Kamera befindet, ob diese steht, eine Plank (bzw. einen Unterarmstütz) oder den Herabschauenden Hund ausführt. Bei der Durchführung der Plank und des Herabschauenden Hundes soll außerdem festgestellt werden, ob diese korrekt ist. So sollen beispielsweise Fehlhaltungen wie ein krummer Rücken, ein zu hohes Gesäß oder eine falsche Positionierung der Arme detektiert werden.

2 Umsetzung

2.1 Erzeugen der Trainingsdaten

Die Trainingsbilder wurden mittels Raspberry Pi Kamera erzeugt. Jedes Bild hat dabei eine Auflösung von 640x480 Pixel. Es wurden pro Klasse ca. 450 Bilder aufgenommen, sodass sich insgesamt eine Summe von ca. 2.700 Originalbildern ergibt.

Bei dem Erstellen der Bilder wurde darauf geachtet, möglichst unterschiedliche Szenarien abzubilden. Es wurden zwei verschiedene Personen in unterschiedlicher Kleidung fotografiert. Die Halteübungen wurden von Vorne und von verschiedenen Seiten/ Winkeln fotografiert. Außerdem wurden die Bilder in unterschiedlichen Beleuchtungssituationen aufgenommen. In Abbildung 1 sind beispielhaft zwei Bilder zu sehen, die beide zur Klasse „Plank“ gehören und an denen gut veranschaulicht werden kann wie unterschiedliche Winkel, Klamotten und Personen in die Trainingsdaten eingearbeitet wurden. Der Hintergrund der Bilder war unruhig (diverse Gegenstände, Möbel etc.) und wurde auch von verschiedenen Positionen aus aufgenommen.

(a) Plank von schräg Vorne aufgenommen (b) Plank seitliche Aufnahme

Abbildung 1: Bespiele für verschiedene Personen, Klamotten und Winkel von Bildern einer Klasse (Plank)

Der Hintergrund wurde außerdem immer wieder verändert, da die Bilder über einen längeren Zeitraum hinweg entstanden sind und um das Netz nicht nur auf einen bestimmten Hintergrund zu trainieren. Aus diesem Grund wurden teilweise auch absichtlich größere Kartons und Yogamatten im Hintergrund platziert, um Möbel zu verdecken bzw. den Hintergrund zu verändern. In Abbildung 2 sind beispielhaft vier Aufnahmen des Hintergrunds zu sehen. An diesen Aufnahmen ist gut zu erkennen wie ein möglichst unterschiedlicher Hintergrund trotz eingeschränkter räumlicher Möglichkeiten erzeugt werden konnte. Als initiale Trainingsdaten wären Bilder vor einer einfarbigen Wand leichter handhabbar gewesen, derartige Fotos waren allerdings leider durch die derzeitige Corona-Situation nicht erzeugbar.

(a) Hintergrund ohne Boxen | Sonneneinstrahlung

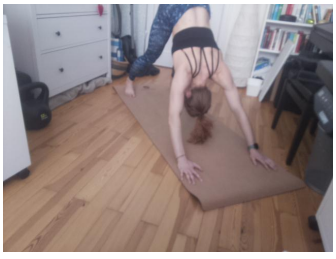
(b) Hintergrund mit Boxen und Yogamatte | Keine Sonneneinstrahlung

(c) Hintergrund mit großer Box | künstliches Licht

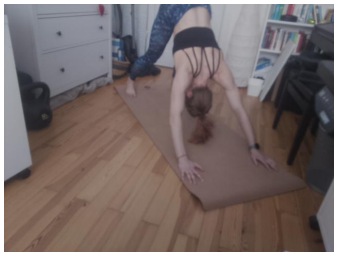
(d) Hintergrund mit roter Yogamatte und Verdeckung durch Schreibtisch | keine Sonneneinstrahlung

Abbildung 2: Bespiele für verschiedene Aufnahmen des Hintergrundes

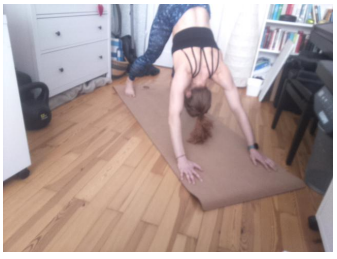
Pro Originalbild wurde jeweils zusätzlich eine Kopie mit um 20% erhöhter Helligkeit und eine Kopie mit um 20% reduzierter Helligkeit angelegt (siehe Abbildung 3). Dies verdreifacht den Trainingsbilddatensatz auf insgesamt ca. 8.100 Bilder. Während des Trainings wurden die Bilder darüber hinaus durch Spiegelungen und Rotation noch weiter verändert (siehe Abschnitt 2.2).



(a) Original



(b) Dunkler



(c) Heller

Abbildung 3: Bespiele für die Veränderung der Belichtung eines Bildes der Klasse Herabschauenderhund

2.2 Modellierung des Netzes

Die Aufgabe, anhand von Bildern zu erkennen, um was für eine Halteübung es sich handelt, oder ob diese Halteübung falsch bzw. richtig ausgeführt wird, kann durch eine Zuordnung der Bilder zu Klassen gelöst werden. Um einzelne Bilder jeweils einer bestimmten Klasse zuzuordnen, eignet sich ein Faltungsnetz (siehe [1, S. 40 f.]). Aus diesem Grund wurde ein Netz, wie in Code-Auszug 1 zu sehen ist, entwickelt.

Quellcode 1: Programmatische Beschreibung des neuronalen Netzes

```
1  model = Sequential([
2    data_augmentation,
3    layers.experimental.preprocessing.Rescaling(1./255),
4    layers.Conv2D(16, 5, padding='same', activation='elu'),
5    layers.MaxPooling2D(),
6    layers.Dropout(0.2),
7    layers.Conv2D(32, 3, padding='same', activation='elu'),
8    layers.MaxPooling2D(),
9    layers.Dropout(0.1),
10   layers.Conv2D(64, 3, padding='same', activation='elu'),
11   layers.MaxPooling2D(),
12   layers.Dropout(0.1),
13   layers.Conv2D(128, 3, padding='same', activation='elu'),
14   layers.MaxPooling2D(),
15   layers.Dropout(0.1),
16   layers.Conv2D(256, 3, padding='same', activation='elu'),
17   layers.MaxPooling2D(),
18   layers.Dropout(0.1),
19   layers.Conv2D(512, 3, padding='same', activation='elu'),
20   layers.MaxPooling2D(),
21   layers.Dropout(0.1),
22   layers.Flatten(),
23   layers.Dense(128, activation='elu'),
24   layers.Dense(num_classes),
25 ])
```

In den beiden Zeilen 2 und 3 des Quellcodes 1 werden die ersten beiden Schichten des Models definiert. Diese ersten beiden Schichten werden nicht trainiert sondern dienen dazu, den Input, der in das Model gegeben wird, anzupassen. Die erste Schicht (Zeile 2 Quellcode 1) dient dazu, während des Trainings, jedes Trainingsbild innerhalb eines gewissen Rahmens zufällig zu verändern. Die Änderungen können dabei in Form einer horizontalen Spiegelung, einer Rotation um bis zu 10% und einer Vergrößerung um bis zu 5% angewandt werden (siehe Quellcode 2). Somit werden bei jeder Trainings-Epoche die Bilder in leicht veränderter Art in das Model gegeben und die Gefahr eines Overfittings wird reduziert (siehe [4]). In Abbildung 4 ist zu sehen, wie ein Bild auf neun verschiedene Arten durch die data-augmentation-Schicht verändert wurde.

Quellcode 2: Programmatische Beschreibung des neuronalen Netzes

```
1  data_augmentation = keras.Sequential(
2    [
3      layers.experimental.preprocessing.RandomFlip("horizontal",
4                                                   input_shape=(img_height,
5                                                                img_width,
6                                                                3)),
7      layers.experimental.preprocessing.RandomRotation(0.1),
8      layers.experimental.preprocessing.RandomZoom(0.05),
9    ]
10 )
```

Abbildung 4: Beispiele durch data-augmentation-Schicht verändertes Bild

Der Vorteil dieses Vorgehens gegenüber einer Veränderung/ Erweiterung der Bilder vor dem Training ist, dass ein Bild in verschiedenen Epochen unterschiedlich verändert wird. Ein weiterer Vorteil ist außerdem, dass die Veränderung der Bilder während des Trainings durch eine GPU (wenn vorhanden) berechnet werden kann. Zu beachten ist allerdings, dass dies die benötigte Dauer für das Training des Models erhöht. Nach dem Training, also bei der Anwendung des Netzes, entfällt diese Schicht.

Die zweite Schicht (Zeile 3 Quellcode 1) dient dazu, die Auflösung der eingehenden Bilder anzupassen. Die für das Netz gewählte Auflösung betrug 370x240 Pixel. Dies geschieht sowohl während des Trainings als auch zur Ausführungszeit. Dadurch erübrigt sich die Anpassung der Auflösung der eingehenden Bilder außerhalb des Netzes.

Auf die zweite Schicht folgen sechs Abschnitte von Schichten, die untereinander sehr ähnlich aufgebaut sind (Zeile 4-21 Quellcode 1). Ein Abschnitt besteht jeweils aus einem Faltungs-Layer, einem Max-Pooling-Layer und einem Dropout-Layer. Im ersten Abschnitt hat der Faltungs-Layer einen Faltungskern von 5x5. Die restlichen Faltungs-Layer haben alle einen Faltungskern von 3x3. Die Anzahl der Feature-Maps werden von Faltungs-Layer zu Faltungs-Layer immer weiter erhöht. Im Zusammenspiel mit den Max-Pooling-Layern hat dies den Effekt, dass die Größe der Feature-Maps immer kleiner wird, sich die Anzahl allerdings erhöht. Dies wird in der durch „visualkeras“ erzeugten Abbildung 5 angedeutet. Leider berücksichtigt „visualkeras“ die Anzahl der Feature-Maps nicht immer, sodass vor allem die Verkleinerung der Feature-Maps gut zu erkennen ist.

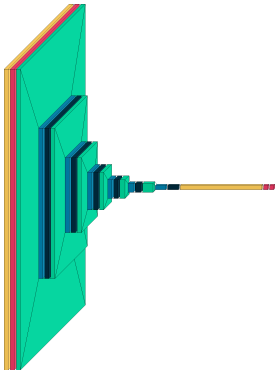


Abbildung 5: Darstellung des Modells durch „visualkeras“

Als Aktivierungsfunktion wurde bei allen Faltungs-Layern die „elu“-Funktion (Exponential Linear Unit) gewählt, da diese sowohl das „dead relu“, als auch das „vanishing gradient“-Problem beseitigt (siehe [8]). Dadurch ist es ohne Probleme möglich mehr als drei Faltungs-Layer hintereinander einzusetzen. Eigentlich führte die „selu“-Aktivierungsfunktion (Scaled Exponential Linear Unit) im Vergleich zu der „elu“-Aktivierungsfunktion zu besseren Trainingsergebnissen. Allerdings ist die „selu“-Aktivierungsfunktion noch nicht in TF-Lite implementiert, sodass eine Umwandlung zu einem TF-Lite-Modell fehlschlägt, wenn die „selu“-Aktivierungsfunktion im Netz eingesetzt wird.

Um Overfitting zu vermeiden, wurde hinter jedem Max-Pooling-Layer ein Dropout-Layer platziert. Eine Droprate von 20% nach dem ersten Max-Pooling-Layer und 10% bei den Anderen hat sich in diesem Projekt als sinnvoll erwiesen.

Die restlichen Schichten (Zeile 22-24 Quellcode 1) dienen schlussendlich dazu, die Informationen, die durch die Faltungs-Layer aus den Bildern extrahiert wurden, zu einzelnen Klassen zu verdichten. Dadurch ergibt sich pro eingegebenem Bild eine Angabe pro Klasse wie wahrscheinlich es ist, dass das Bild zu der jeweiligen Klasse gehört.

2.3 Trainieren des Netzes

Zum Trainieren des Netzes wurden die Bilder als Dataset geladen. Dies wurde durch die Tensorflow-Funktion $tf.keras.preprocessing.image\_dataset\_from\_directory$ realisiert. Durch dieses Vorgehen bekommen Bilder automatisch als Label den Namen, des Ordners, in dem sie liegen. Die Ordnerstruktur, in der die Bilder lagen, sah wie folgt aus:

- trainings_daten
  - HerabschauenderHund
  - HerabschauenderHund_falsch
  - KeinePerson
  - Plank
  - Plank_falsch
  - Stehen

Durch die Tensorflow-Funktion *tf.keras.preprocessing.image_dataset_from_directory* wird außerdem eine Batchsize festgelegt und es ist eine Teilung der Daten in Trainings- und Vallidierungsdaten möglich (siehe [7]). In diesem Projekt wurden 20% der Bilder als Validierungsdaten definiert.

Bei der Kompilierung des Models wurde als Fehlerfunktion „CategoricalCrossentropy“ gewählt (Zeile 2 Quellcode 3). Als Optimierer wurde „Adam“ gewählt (Zeile 1, Quellcode 3), da „Adam“ in der Regel bei dem Training von Faltungsnetzen anderen Optimierungsverfahren überlegen ist (siehe [9, S.6 f.]).

Quellcode 3: Kompilierung des Models

```
1  model.compile(optimizer='adam',
2                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
3                metrics=['accuracy'])
```

Das Netz wurde 20 Epochen lang trainiert. Eine Erhöhung der Epochen wäre aufgrund von Overfitting und kaum mehr vorhandener Leistungssteigerung nicht sinnvoll gewesen. Overfitting wurde außerdem durch das Einfügen von Dropout-Layern und durch die Data-Augmentation zur Laufzeit vermieden (siehe 2.2). Diese Schichten entfallen nach dem Abschluss des Trainings und sind somit im anwendbaren Model nicht mehr vorhanden.
Nach jedem Training wurden die Ergebnisse geprüft und entsprechende Anpassungen am Model und am Trainingsverfahren vorgenommen. In Abschnitt 3 folgt eine detaillierte Auswertung.

2.4 Anwendung des Netzes / inference

Das fertig trainierte Netz sollte auf einem Raspberry Pi die Bilder einer Raspberry Pi Kamera in Echtzeit auswerten. Um dies zu realisieren war es notwendig, das trainierte Model in ein Tensorflow-Lite-Model umzuwandeln (siehe [6]). Die Umwandlung eines abgespeicherten Models ist durch wenige Zeilen Python-Code möglich (siehe Quellcode 4).
Es kann allerdings zu Problemen kommen, wenn in dem Model bestimmte Funktionen/ Eigenschaften enthalten sind, die von TFLite nicht unterstützt werden. So wäre in diesem Projekt (wie in Abschnitt 2.2 schon erwähnt) eine „selu“-Aktivierungsfunktion effizienter gewesen. „selu“-Aktivierungsfunktionen werden allerdings zum jetzigen Zeitpunkt nicht von TFLite unterstützt und konnten deswegen nicht verwendet werden.

Quellcode 4: Umwandlung in TFLite-Model

```
1  import os
2  # Convert the model to tflite
3  saved_model_dir = os.path.abspath(os.curdir) + '\\' + model_name
4  print(saved_model_dir)
5  converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
6  tflite_model = converter.convert()
7
8  # Save the model.
9  with open('model.tflite', 'wb') as f:
10   f.write(tflite_model)
```

Das TFLite-Model wurde auf dem Raspberry Pi innerhalb eines Python-Skripts benutzt, um die Bilder der angeschlossenen Raspberry Pi Kamera auszuwerten. In das TFLite-Model wird dafür ein einzelnes Bild als Eingabe gegeben. Die passende Skalierung des Bildes wird dabei

automatisch durch das Model übernommen (siehe Zeile 3 Quellcode 1), sodass dies nicht durch extra Code in der Anwendung geschehen muss. Dies hat außerdem den weiteren Vorteil, dass bei der Entwicklung der Anwendung nicht zwingend die genaue Auflösung der Bilder, die in das Model gegeben werden können, bekannt sein muss.
Das TFLite-Model gibt für jedes eingegebene Bild einen Wert pro Klasse zurück. Dieser gibt an mit welcher Wahrscheinlichkeit das eingegebene Bild zu der jeweiligen Klasse gehört. Bei dem verwendeten Model brauchte ein Raspberry Pi 4 Model B ohne zusätzliche Beschleunigungs-Hardware pro Bild durchschnittlich ca. 200 ms für die Auswertung eines Bildes.

Der Nutzer sieht bei der Anwendung des Programms das Live-Bild. Die Klasse mit der höchsten Wahrscheinlichkeit und die Zeit, die für die Berechnung notwendig war, werden am oberen Rand eingeblendet (siehe Abbildung 6).

Abbildung 6: Bild der Live-Anwendung

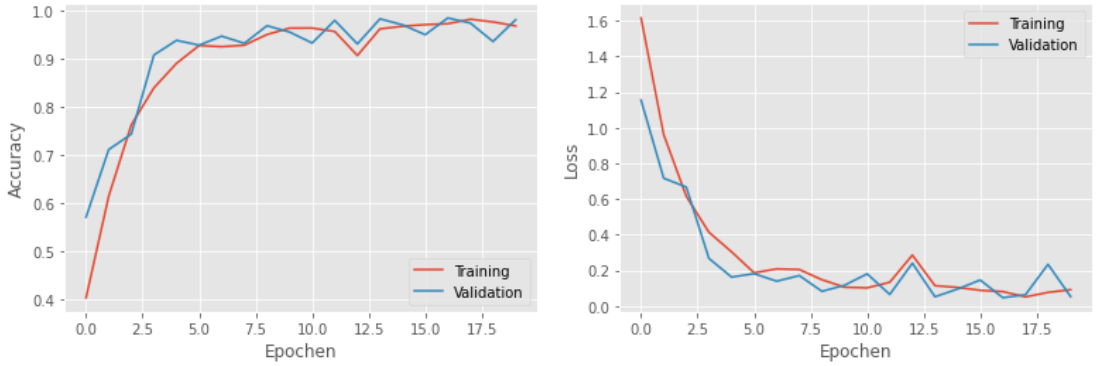
Bei einem Ausbau der Anwendung wäre es denkbar, z.B. einen Warnton auszugeben, wenn eine Fehlhaltung eingenommen wird. Außerdem wäre ggf. eine Einblendung eines Timers, der die Zeit pro Halteübung misst, denkbar. Es könnten auch spezifischere Korrekturhinweise gegeben werden wie z.B. „Achtung: Der Rücken ist krumm“. Dies würde allerdings eine noch spezifischere Unterteilung in weitere Klassen voraussetzen, da in dem angewandten Model nicht zwischen den verschiedenen Fehlern unterschieden wurde.

3 Analyse

3.1 Auswertung der Trainingsergebnisse

Bei einer Betrachtung der Genauigkeit und des Fehlers über den Trainingsverlauf ist festzustellen, dass in den ersten fünf Epochen eine hohe Steigerung der Genauigkeit erzielt wurde (siehe Abbildung 7). In den darauf folgenden Epochen nimmt der Zuwachs an Genauigkeit immer weiter ab bis er in den letzten Epochen nahezu gegen Null geht. Nach der letzten Epoche wurde bei den Validierungsdaten eine Genauigkeit von 98,14% erreicht.
Des Weiteren ist in Abbildung 7 gut zu erkennen, dass sich die Genauigkeit der Trainingsdaten und der Validierungsdaten relativ synchron zueinander entwickeln. Dies ein Zeichen dafür, dass das Netz die Trainingsdaten nicht einfach „auswendig lernt“ (Overfitting), sondern auch andere Daten, die nicht in den Trainingsdaten vorhanden sind, richtig auswertet. Das Netz verfügt

somit über eine Generalisierungsleistung.



(a) Genauigkeit im Verlauf des Trainings (b) Fehler im Verlauf des Trainings

Abbildung 7: Fehler und Genauigkeit im Verlauf des Trainings

Eine noch differenziertere Methode das trainierte Netz zu untersuchen ist, Testdaten auswerten zu lassen und dabei eine Confusion Matrix aufzustellen. Dies wurde in Abbildung 8 gemacht. Bei den Zahlen innerhalb der Matrix handelt es sich um absolute Werte. Der Code zum Erstellen einer derartigen Matrix stammt im wesentlichen aus [10].

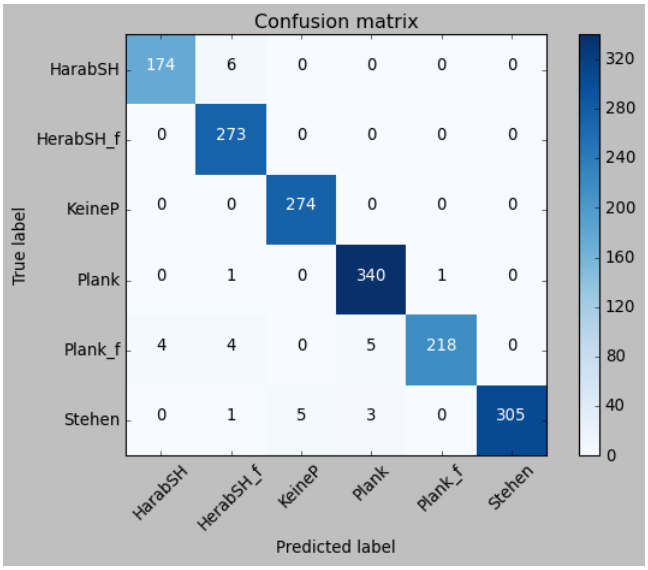


Abbildung 8: Confusion Matrix (mit abgekürzten Namen)

Durch die Confusion Matrix ist es möglich zu sehen, zu welcher Klasse die Bilder durch das Netz zugeordnet wurden, und zu welcher Klasse sie eigentlich gehören. In Abbildung 8 ist zu sehen, dass die meisten Bilder zu der Klasse zugeordnet wurden, zu der sie auch gehören (Diagonale von oben Links nach unten Rechts). Die Klasse von der die meisten Bilder (13 von insgesamt 231) falsch zugeordnet wurden, ist eine falsch ausgeführte Plank. Dies kann unter anderem daran liegen, dass eine falsche Ausführung relativ beliebig sein kann und dadurch auch die Trainingsdaten dieser Klasse teilweise recht unterschiedlich waren (siehe Abbildung 9). Die gleiche Begründung kann äquivalent bei der Klasse „Stehen“ angeführt werden.

(a) Plank_falsch 1 (b) Plank_falsch 2 (c) Plank_falsch 3

Abbildung 9: Bespiele für verschiedene Arten von Plank_falsch-Bildern

3.2 Praxistests

Bei der praktischen Anwendung des Models bestätigten sich die Erkenntnisse aus der Auswertung der Trainingsergebnisse. Die verschiedenen Klassen wurden also überwiegend richtig erkannt. Es offenbarte sich bei der Anwendung allerdings ein Problem, das durch die Testbilder vorher nicht aufgefallen war. Die Unterscheidung zwischen den Klassen „Plank_falsch“ und „HerabschauenderHund_falsch“ fiel dem Netz in manchen Haltungen schwer. Insbesondere dann, wenn man das Gesäß bei einer „richtigen“ Plank weit nach oben streckt, werden die beiden Klassen manchmal vertauscht.
Allerdings ist dabei auch zu berücksichtigen, dass diese beiden Klassen in der Realität ggf. einen gewissen Bereich haben, der sich überschneidet. So kann eine sehr falsch ausgeführte Plank mit einem deutlich zu hohem Gesäß einem falsch ausgeführten Herabschauenden Hund sehr ähnlich sein (siehe Abbildung 10).

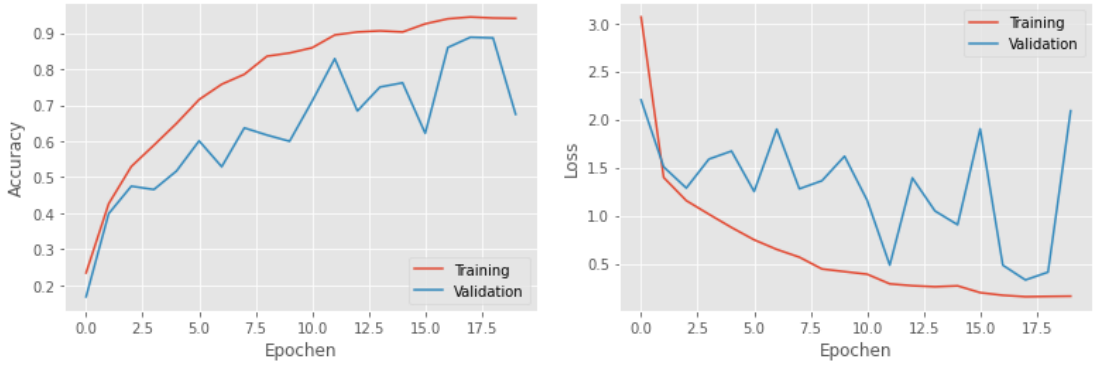
(a) Falscher Herabschauender Hund (b) Falsche Plank

Abbildung 10: Beispiel für Ähnlichkeit zwischen falscher Plank und falschem Herabschauenden Hund

Ein möglicher Ansatz, um dieses Problem zu lösen, wäre eine einzelne Klasse „Falsche_Ausführung“ im Zusammenspiel mit einer State Maschine, sodass bekannt ist, welche Position vorher eingenommen worden war. Beispielsweise könnte es einen State „Plank“ geben, und sobald das Netz erkennt, dass etwas falsch ausgeführt wird, wird in einen State „Plank_falsch“ übergegangen. Eine andere Möglichkeit wäre eine noch stärke Ausdifferenzierung der einzelnen Klassen, sodass Überschneidungen ausgeschlossen werden.

3.3 Vergleich mit anderen Modellierungsansätzen

Um die Berechnungsgeschwindigkeit während der Anwendung noch weiter zu erhöhen, wäre ein denkbarer Ansatz, ein Netz zu modellieren, das weniger Faltungslayer besitzt. Dies scheitert allerdings an der Komplexität der Bilder, da sich einzelne Klasse teilweise nur feingranular unterscheiden. So unterscheidet sich beispielsweise ein Bild auf dem eine Plank mit einem geraden Rücken zu sehen ist, oft nur wenig von einem Bild, auf dem eine Plank mit leicht gekrümmten Rücken zu sehen ist. Versucht man beispielsweise ein Netz mit drei Faltungs-Layern zu trainieren, kommt es zu einem Overfitting (siehe Abbildung 11).



(a) Genauigkeit im Verlauf des Trainings (3 Faltungs-Layer)

(b) Fehler im Verlauf des Trainings (3 Faltungs-Layer)

Abbildung 11: Fehler und Genauigkeit im Verlauf des Trainings (3 Faltungs-Layer)

Es findet somit keine Generalisierung statt und es wird nur eine Genauigkeit von 67,47% nach 20 Epochen bei den Validierungsbildern erreicht. Bei dem Betrachten der Confusion Matrix (Abbildung 12) wird deutlich, dass es dem Netz mit nur drei Faltungs-Layern schwer fällt richtige Ausführungen von falschen Ausführungen der Halteübungen zu unterscheiden. Außerdem ist hier das Problem der Ähnlichkeit zwischen manchen Bilder der Klasse „Herabschauender-Hund_falsch“ und „Plank_falsch“ schon in den Testdaten deutlich zu sehen. Des Weiteren sieht man, dass das Model etliche Bilder einfach als leeren Raum („keinePerson“) klassifiziert, obwohl eigentlich eine Person zu sehen ist.

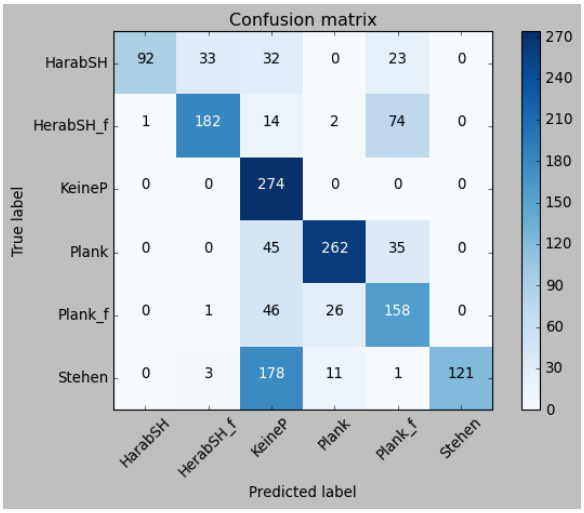


Abbildung 12: Confusion Matrix (mit abgekürzten Namen) von Model mit nur 3 Faltungs-Layern

Eine weitere zunächst denkbare Art von Netz könnte auf Posenerkennung (pose estimation) basieren. Das Problem an diesem Vorgehen wäre allerdings, dass in der Regel durch die Posenerkennung 17 Punkte auf dem menschlichen Körper markiert werden (siehe [5]). Dabei werden unter anderem zwei Punkte für die Schultern und zwei Punkte für die Hüften gesetzt. Diese vier Punkte werden miteinander verbunden. Durch dieses Vorgehen ist es allerdings nicht möglich eine Krümmung im Rücken festzustellen, wodurch ein Einsatz in diesem Projekt nicht zielführend gewesen wäre.
Dieses Problem hätte nur durch eine eigene Posenerkennung mit neuen Punkten, die auf dem Rücken verteilt werden, erreicht werden können. Ein derartiges Unterfangen hätte allerdings den Umfang dieses Projekts gesprengt.

4 Ausblick und Fazit

Eine Weiterführung des Projekts wäre denkbar. Beispielsweise könnten weitere Klassen eingeführt werden. Bei einer Erweiterung des Netzes um weitere Klassen, wäre zu prüfen, ob die jetzige Komplexität des Aufbaus ausreicht, oder ob das Netz entweder erweitert, oder sogar zu einem Dens-Netz umgebaut werden müsste. Nach dem Umbau müsste wiederum geprüft werden, ob die Hardware des Endgerät noch eine ausreichend schnelle Berechnung zur Laufzeit gewährleisten kann. Außerdem müssten natürliche neue Trainingsbilder erzeugt werden.

Zusammenfassend lässt sich feststellen, dass die Aufgabe insgesamt gut durch das in Abschnitt 2.2 beschriebene Netz gelöst werden konnte. Das einzige Problem während der Durchführung, dass in manchen Fällen die Klassen „HerabschauenderHund_falsch“ und „Plank_falsch“ vertauscht werden, ist zu verschmerzen bzw. könnte durch die in Abschnitt 3.2 angerissenen Lösungsansätze behoben werden. Auch die durchschnittliche Berechnungszeit von ca. 200 ms ist für die Anwendung ausreichend, da es keine harten Deadlines gibt und ein Wechsel zwischen den Klassen während der Anwendung nicht sekündlich zu erwarten ist. Als einer der wichtigsten Einflussfaktoren auf die Berechnungszeit hat sich die Auflösung, mit der die Bilder verarbeitet werden, herausgestellt.
Während der Arbeit an dem Projekt wurde deutlich, wie wesentlich der Einfluss sowohl der Quantität, als auch die Qualität der Trainingsbilder auf die Resultate sind. So erwirken kleine Änderungen an der Struktur des verwendeten Netzes oft einen geringeren Unterschied am Endergebnis als eine Erweiterung der Daten. Insbesondere das Aufhellen und Abdunkeln der Bilder und die damit einhergehende Verdreifachung der Trainingsbilder bewirkte einen großen Qualitätssprung in dem Projekt.

Literatur

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning.* Springer, Yorktown Heights, NY, USA, 2018.

[2] Leonard Bardtke. Workout klassifizierung. Web, July 2020.

[3] Tobias Kasprak Dr. Nicolas Gumpert. Unterarmstütz. Web, September 2020.

[4] Google. Tutorial_image classification. Web, Febuar 2020.

[5] Google. Pose estimation. Web, 2021. URL `https://www.tensorflow.org/lite/examples/pose_estimation/overview`; abgerufen am 25. Februar 2021.

[6] Google. Tensorflow lite converter. Web, 2021. URL `https://www.tensorflow.org/lite/convert/`; abgerufen am 24. Februar 2021.

[7] Google. Tensorflow/api/tf.keras.preprocessing.image_dataset_from_directory. Web, Januar 2021.

[8] Casper Hansen. Activation functions explained - gelu, selu, elu, relu and more. Web, August 2019.

[9] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. conference paper at ICLR 2015, 2014. Online erhältlich unter `https://arxiv.org/abs/1412.6980`; abgerufen am 24. Februar 2021.

[10] SalRite. Demystifying 'confusion matrix' confusion. Web, 2018. URL `https://towardsdatascience.com/demystifying-confusion-matrix-confusion-9e82201592fd`; abgerufen am 25. Februar 2021.