Optische Fingerpositionserkennung als Benutzerinterface für eingebettete Systeme

Kenneth Schmidtsdorff kenneth.schmidtsdorff@haw-hamburg.de
Department Informatik
Fakultät Technik und Informatik
Hochschule für Angewandte Wissenschaften Hamburg

Editor: Kenneth Schmidtsdorff

Kurzzusammenfassung

In dieser Arbeit wird eine optische Fingerpositionserkennung am Anwendungsbeispiel des Brettspiels Othello evaluiert. Dafür wurde ein eingebettetes System aus einem Raspberry Pi 4, Laserprojektor und Kamera konstruiert, welches die Fingerpositon in einem Kamerabild ermittelt und als Spielzug interpretiert. Die Extraktion des Spielbereichs findet durch manuell konfigurierte Filter statt, um anschließend einem neuronalen Netzwerk übergeben zu werden. Das Netzwerk besteht aus einem MobileNetV2 als Feature Extractor und hinzugefügten Fully Connected Layers, um per Regression die Fingerposition ermitteln zu können. Das Netz wurde mittels Transfer Learning trainiert, bei dem Dropout-Raten im Regressionsteil von 0%, 20% und 50% getestet wurden und ein Anteil von 0% das beste Ergebnis liefert. Das Netzwerk wird für die Coral USB Edge-TPU kompiliert und auf dieser ausgeführt, wodurch die Inferenzdauer des Systems auf 7ms reduziert wird. Der mittlere Positionsbestimmungsfehler liegt bei 1,9% des Eingabebildes, was sich in der praktischen Evaluation des Systems als präzise genug für die Bedienbarkeit herausstellt.

Stichworte: Coral-TPU, Raspberry Pi, MobilenetV2, Computer-Vision, Regression, Othello, Transfer Learning, Dropout, HCI, TFLite, Embedded Machine Learning, Deep Learning, Fingerposition

1. Einleitung

Die Fingerpositionserkennung ist eine Eingabemöglichkeit für Mensch-Maschine-Interaktionen. Dabei ist die Fingererkennung vor allem durch Touchscreens in Smartphones oder Tablets weit verbreitet.
Die rein optische Positionsbestimmung ist dagegen in der Praxis weniger verbreitet, bildet jedoch ein aktives Forschungsgebiet. Sie bietet dabei die Möglichkeit der Erkennung von Zeichen und Gesten, ohne eine definierte Oberfläche berühren zu müssen.

1.1 Verwandte Arbeiten

In [1] werden die Fingerspitzenpositionen einer Hand durch manuell konfigurierte Filter und Transformationen bestimmt, um Zeichensprache erkennen zu können. Mit vergleichbaren Verfahren werden auch in [2] die Fingerspitzenpositionen einer Hand bestimmt um eine Gestensteuerung eines Roboters zu implementieren. Beide erfordern eher statische, einfar-

bige Hintergründe um die Positionen sicher zu bestimmen.
Für die Erkennung in dynamischen Szenen werden aktuell Deep Learning Verfahren genutzt. So wird in [3] durch zwei konkatenierte Faltungsnetzwerke (CNN) eine Object Detection der Zeigefingerspitze erreicht. Die Berechnungen erfolgen jedoch auf einer Desktop GPU. Eine ebenfalls auf CNNs basierte Gestensteuerung durch Fingerpositionserkennung für augmented Reality, wurde von [4] entwickelt. Doch auch hier findet die Inferenz auf einem Server statt, dem ein Kamerabild gesendet wird, um die relativen Koordinaten der Fingerspitze zurück zu erhalten.
2020 entwickelte [5] eine Positionsbestimmung für alle Gelenke einer Hand, die auf modernen Smartphones in Echtzeit lauffähig ist. Dafür wurde eine zweistufige Object Detection implementiert, die zunächst die Hand findet und anschließend alle Gelenke.
Mit dem Sony Xperia Touch[6] gab es 2017 einen prototypischen Projektor, der ein Android-Benutzerinterface auf eine Oberfläche in fester Distanz projiziert und eine Bedienung mit mehreren Fingern gleichzeitig ermöglicht. Über die Methode zur Fingererkennung gibt es keine öffentlichen Informationen. Die Studie [7, c.4] setzte diesen Projektor als Gruppenmedium in Lernumgebungen ein. Dabei zeichnete der Projektor sich durch die große Bildfläche gegenüber Tablets für das gemeinsame Arbeiten aus.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, ein System zu konstruieren, bei dem sich eine projizierte Benutzeroberfläche durch Fingerplatzierung steuern lässt. Dabei soll im Gegensatz zum Xperia Touch [6] ein variabler Abstand zur Projektionsfläche und damit variable Bildgrößen möglich sein. Da die Benutzeroberfläche zwangsläufig auch auf die Finger projiziert wird, muss das System mit dynamischen Szenen und wechselnden Farbmustern auf der Hand umgehen können, weshalb wie in den verwandten Arbeiten auf eine Erkennung mit CNNs gesetzt wird. Dabei soll anders als bei [3] und [4] die Berechnung vollständig auf dem eingebetteten System erfolgen. Für die Fingerpositionsbestimmung wird, anders als in [5], nicht zwangsläufig die gesamte Hand sichtbar sein müssen.
Das Brettspiel Othello (auch bekannt als Reversi) wird als konkrete Testanwendung genutzt. Ziel ist es, das Brettspiel nur durch die Platzierung eines Fingers auf dem projiziertem Spielfeld, gegen das eingebettete System spielen zu können.

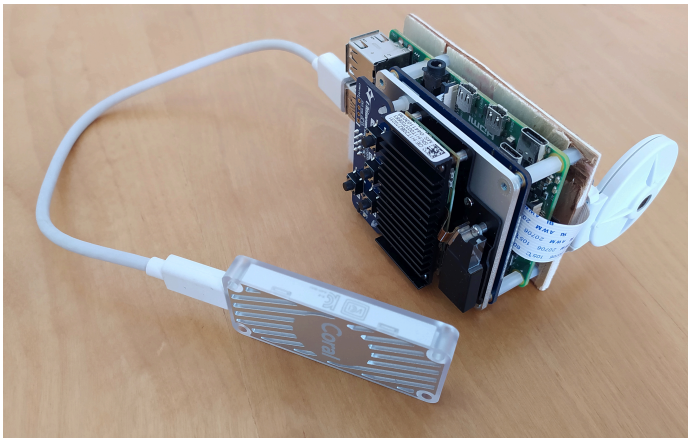
1.3 Othello

Othello ist ein Zwei-Personen Spiel auf einem schachbrettartiges Spielfeld mit 64 Feldern. Abwechselnd legen die Spieler einen Spielstein ihrer Farbe auf ein freies Feld, sodass mindestens 1 gegnerischer Stein zwischen dem neu gelegten und einem bereits platziertem, eigenen Spielstein eingeklemmt wird. Alle eingeklammerten Steine werden anschließend umgedreht, sodass diese die Farbe der umklammernden Steine annehmen.
Das Spiel ist vorbei, sobald das Spielbrett gefüllt ist, oder beide Spieler aussetzen müssen, weil keine gültige Platzierung möglich ist. Gewinner ist der Spieler mit den meisten Steinen seiner Farbe am Ende des Spiels.
Das Spiel eignet sich, durch sein einfaches Regelwerk und die diskrete Platzierung der Steine, als Anwendungsszenario für die Fingerpositionserkennung.

2. Systemüberblick

2.1 Hardware

Das eingebettete System basiert auf einem Raspberry Pi 4B mit 2GB Arbeitsspeicher. Zur Beschleunigung der Inferenz des neuronalen Netzwerks ist zusätzlich ein Coral USB Beschleuniger über USB 3.0 angeschlossen. Dieser enthält eine Google Tensor Processing Unit(TPU) mit 4TOPS int8 Performance [8, S.1]. Als Projektor wird ein Nebra AnyBeam HAT an den Raspberry über dessen GPIOs angeschlossen. Der Laserprojektor wird als Standardbildausgabe genutzt und liefert eine Auflösung von 720p@60Hz [9].
Um die Fingerspitze zu erkennen, wird das Raspberry Pi Camera Module V2.1 genutzt, welches Bilder mit einer Auflösung von 8MP aufnehmen kann[10].
Das Foto 1a zeigt die verwendete Hardware. Von links nach Rechts ist dort zu sehen: Der Coral Stick, der Projektor mit blauem PCB und Aluminiumgrundplatte, der grüne Raspberry Pi, sowie die verwendete Kamera im weißen Gehäuse.
Das Foto 1b zeigt das System montiert auf einem Stativ. Die Energieversorgung ist über einen Akku gewährleistet, wodurch das System mobil einsetzbar ist. Auf dem Foto wird das Startspielfeld von Othello auf den Tisch projiziert.



(a) Verwendete Hardware auf Basis des Raspberry Pi 4

(b) Vollständiges System auf Stativ angebracht. Das Startspielfeld wird angezeigt

Abbildung 1: Fotos der Hardware

2.2 Ablauf der Zugbestimmung

Die Abbildung 2 zeigt in Ablaufdiagrammen die Ermittlung einer Benutzereingabe für einen Othello-Spielzug.
Sobald ein Spieler an der Reihe ist, wird dieser von der Othello-Implementation aufgefordert den Index des Spielfelds zu benennen, auf dem der nächste Stein platziert werden soll. Das

Ablaufdiagramm 2a zeigt die Hauptfunktion zur Bestimmung der Benutzereingabe durch Fingerpositionserkennung.
In Schritt A.2 wird durch die Funktion 'spielfeldindex_ermitteln()' die optische Positionserkennung genutzt, um den Spielfeldindex Feld1 des durch die Fingerspitze belegten Spielfelds zu bestimmen. Es folgt eine 500ms Wartezeit, bevor eine erneute Bestimmung durchgeführt wird. Das Ergebnis der zweiten Positionsbestimmung wird als Feld2 gespeichert. Die Wartezeit ist notwendig, um Fehleingaben durch Bildaufnahmen während der Bewegung zum gewünschten Feld vermeiden zu können. Deshalb wird in der Bedingung A.5 geprüft, dass in beiden Momenten das selbe Feld durch den Finger belegt wurde. Ist dies nicht der Fall, wird der letzte Spielfeldindex als Feld1 beibehalten und nach erneuter Wartezeit die aktuelle Fingerposition bestimmt.
Waren die erkannten Spielfeldindizes identisch, wird dem Spiel die Benutzereingabe gemeldet. Ungültige Züge werden vom Spiel erkannt und die Zugaufforderung erneut gestartet.

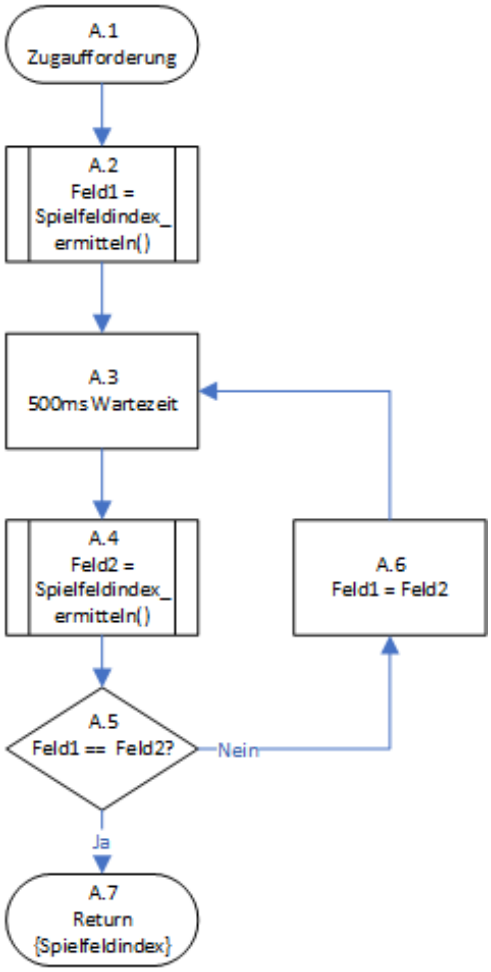
Der Ablauf 2b beschreibt die Funktion 'spielfeldindex_ermitteln()' detaillierter. In dieser Funktion wird in Aktion B.9 zunächst ein Foto über die PiCamera aufgenommen, indem das projizierte Spielfeld mit dem Finger zu sehen ist. Aus diesem Foto wird durch explizit definierte Filter der Ausschnitt des Spielbretts in konstanter Auflösung extrahiert. Die detaillierte Beschreibung dieses Extraktionsprozesses erfolgt im Abschnitt 2.4.
Aus dem extrahierten Foto des Spielfelds wird im Block B.11 durch ein neuronales Netzwerk die Fingerspitzenposition in relativen Spielfeldkoordinaten ermittelt. Dieses neuronale Netzwerk ist der Fokus dieser Arbeit und wird in den nachfolgenden Abschnitten explizit beschrieben. Wenn kein Finger auf dem Spielfeldfoto erkannt wurde, wiederholt sich dieser Ablauf. Anschließend wird dann aus der relativen Position der Fingerspitze der Spielfeldindex berechnet und als Ergebnis zurückgegeben.

2.3 Fingeranwesenheit

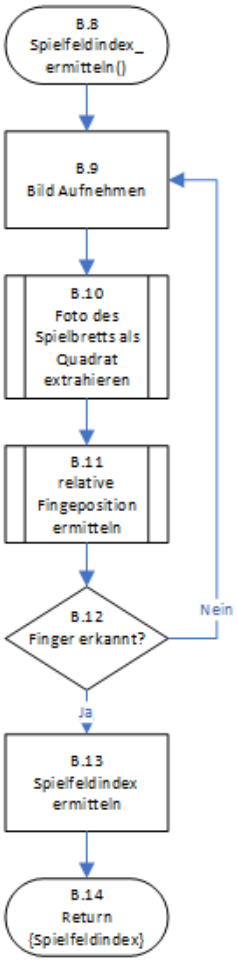
Das neuronale Netzwerk zur Positionsermittlung liefert zu jedem Eingabebild eine prognostizierte Fingerpositon. Es hat keine Ausgabemöglichkeit um anzuzeigen, ob überhaupt ein Finger erkannt wurde. Deshalb wird diese Ermittlung extern durchgeführt, indem das Eingabebild auch um 90°, 180° und 270° rotiert dem Netzwerk zur Positionsbestimmung gegeben wird. Nur wenn die Ausgabepositionen unter Berücksichtigung der Drehung in das selbe Spielfeld fallen, führt dies zu der Annahme, dass tatsächlich ein Finger auf dem Bild erkannt wurde.

2.4 Extraktion des Spielbretts

Abbildung 3 zeigt den Extraktionsprozess, um aus dem Kamerafoto den Ausschnitt des Spielfelds zu extrahieren. Dies wird durch eine Transformation des Eingabebilds erreicht, die das Spielbrett in konstanter Rotation und Größe darstellt.
Diese Transformation wird durch eine Homographie-Matrix beschrieben. In der ersten Ausführung des Programms muss diese Matrix zunächst bestimmt werden, weshalb die Bedingung in '2.' nicht erfüllt ist.
Basierend auf dem grünen Farbkanal des Eingabebilds wird das Bild in Graustufen dargestellt. Aus dem Graustufenbild wird durch einen Schwellwert für die Helligkeit ein 2-farbiges



(a) Zugaufforderung bearbeiten



(b) Einzelnen Spielfeldindex ermitteln

Abbildung 2: Ablaufdiagramme: Ermittlung des Spielzugs durch erkennen der Fingerposition

Bild erzeugt. Nach anschließender Rauschunterdrückung, um kleinere Flächen wie die Reflektion am Stativ herauszufiltern, entsteht Bild 3.
Für Bild 3 werden anschließend alle Umrisse um Formen bestimmt. Der Umriss mit einer Fläche von weniger als 90% des Gesamtfotos und mehr als 4 Pixeln pro Spielfeld (4*64 = 256 Pixel), wird dabei als Spielbrett interpretiert. Um diesen Umriss wird ein Rechteck mit minimaler Fläche approximiert. Dieses Rechteck zeigt Bild 4. Durch die Approximation eines Rechtecks, ist der Extraktionsprozess nicht in der Lage ein perspektivisch verzerrtes, trapezförmiges Spielbrett optimal zu umreißen. Der Projektor muss somit parallel zur Projektionsfläche ausgerichtet sein.
Die vier Eckpunkte des minimalen Rechtecks fließen dann in den Schritt 5 ein. In diesem

wird die Homographie-Matrix berechnet, welche die Transformation des Bildes beschreibt, um die vier Eckpunkte des minimalen Rechtecks auf die Eckpunkte des gewünschten Ausgabebildes zu projizieren.
Bild 6 entsteht somit durch Anwendung der Matrix auf das Eingabefoto. Es wird nur der gewünschte Ausschnitt des Zielrechtecks genutzt.
Die Matrixberechnung wird nur initial durchgeführt, da die Projektor- und Kameraposition während des Spielablaufs als konstant angenommen werden. Bei Platzierung der Spielerhand auf dem Spielbrett verkleinert sich der Abstand vom Projektor zur Projektionsfläche. Durch den verringerten Abstand wird das projizierte Bild kleiner. Es bleibt aber in jedem Fall vollständig innerhalb des initial bestimmten Rechtecks. Der Umgang mit diesen Verzerrungen ist Aufgabe des nachfolgenden neuronalen Netzes.

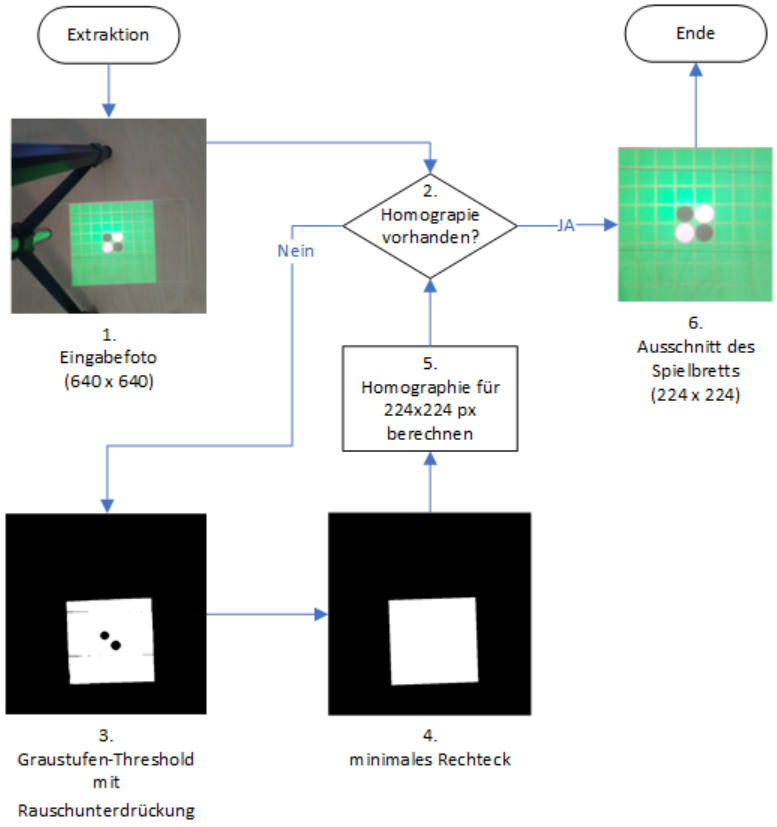


Abbildung 3: Extraktion des Spielbretts aus der Kameraaufnahme

2.5 Netzarchitektur

2.5.1 Modellierung der Aufgabe

Das neuronale Netz muss aus dem extrahierten Spielbrettfoto die Fingerspitzenposition eines Spielers ermitteln. Dabei muss es auf der Coral-TPU ausgeführt werden können.
Das Erkennen eines Objekt und seiner Position im Bild ist üblicherweise das Problem der Object Detection (vgl. [11, S.283]). Dabei wird jedes gefundene Objekt durch ein Rechteck eingerahmt und der jeweilige Bildausschnitt klassifiziert.
In diesem Anwendungsfall gibt es jedoch nur eine Klasse 'Fingerspitze', von der nur eine Instanz in jedem Bild auftreten kann. Durch diese Constraints reicht es aus, nur die Position als Ausgabegröße zu erhalten.
Die Position könnte durch Image Classification diskret in 64 Klassen, eine je Spielfeld bestimmt werden. Dies würde jedoch das Netz stark an den Anwendungsfall koppeln. Deshalb wird Object Localization mit kontinuierlichen Ausgabewerten realisiert.
Dabei soll das Netz zwei reelle Zahlen ausgeben, welche die relative X- und Y-Position der Fingerspitze im Bild beschreiben. Dadurch wird die Ausgabestruktur vom konkreten Anwendungsfall auf einem Spielbrett unabhängig.

2.5.2 Auswahl der Netzarchitektur

Convolutional Neural Networks (CNN)[12] haben, seit dem im Jahr 2012 das 'Alexnet'[13] diese erfolgreich für den ImageNet Wettbewerb[14] nutzte um Bilder zu klassifizieren, eine wichtige Rolle in der Deep Learning Bildverarbeitung.
Bei CNNs wird, anders als bei den MultiLayer Perceptron (MLR) Netzwerken, die Nachbarschaftsbeziehung der mehrdimensionalen Inputvektoren beibehalten und nicht direkt in eine Dimension transformiert. Die Convolutional Layer bestehen dabei aus Filtern fester Größe, die auf den Inputvektor angewandt werden. Die Filterwerte sind zu lernende Parameter.
Um die Coral-TPU überhaupt nutzen zu können, muss das Netz als Tensorflow[15] Modell vorliegen und nur von Coral explizit genannte Operationen[16, table. 1] verwenden. Für die Konfiguration des Modells und das spätere Training wird Keras[17] verwendet. Keras ist ein Python-Wrapper um Tensorflow, der die Bedienbarkeit und damit Entwicklungsgeschwindigkeit erhöht.
Für die Auswahl der Netzarchitektur wird die Schnittmenge aus den direkt für die Edge-TPU verfügbaren Modellen [18] und den von Keras bereitgestellten Modellen [19] gebildet. Das beste Modell, sortiert nach Top-1 und Top-5 Genauigkeit, welches in den 8 MB großen Arbeitsspeicher eines Coral-Sticks passt, ist dabei das CNN MobilenetV2[20].

2.5.3 Transfer Learning

Transfer Learning ist ein Verfahren, um bereits trainierte Netzwerke für eine andere, spezielle Aufgabe weiter zu trainieren. Dahinter steckt die Annahme, dass beim ersten Training auf einem großen Datensatz allgemein hilfreiche Feature Maps gelernt wurden, die auch in anderen Domänen hilfreich sind, um wichtige Merkmale zu erkennen. Dadurch kann die neue Domäne mit einem deutlich kleineren Datensatz und Rechenaufwand gelernt werden. Dabei reduziert sich das Auftreten von overfitting (vgl. [21, S.2]).

Als Basismodell wird in Keras ein auf dem Imagenet[14] Datensatz trainiertes MobilenetV2[20] genutzt.

2.5.4 Regression Kopf

Nach dem eigentlichen CNN enthält das Mobilenet ursprünglich noch Layer die zur Klassifikation dienen. Diese werden entfernt und gegen ein Regressionsmodell ersetzt. Diese Layer haben die Aufgabe aus den Feature Maps des Mobilenets die relative Fingerposition zu bestimmen. Die Abbildung 4 zeigt die an das MobilenetV2 hinzugefügten Schichten zur Positionsbestimmung. Das MobilenetV2 wird somit als Feature Extractor für das Eingabebild genutzt. Nach dem Mobilenet wird ein global average pooling2d Layer eingesetzt, der jede 7x7 Featuremap des Mobilenets durch den Mittelwert zusammenfasst, sodass ein eindimensionaler Layer aus 1280 Neuronen entsteht. Es folgen Fully Connected (Dense) Layer, um die Anzahl Neuronen schichtweise zu reduzieren. Dabei wird die Anzahl zwischen jedem hidden Layer auf ein Viertel reduziert. Der Schritt von 1280 zu 320 Neuronen ist relativ groß gewählt, um die Anzahl der zu lernenden Parameter klein zu halten. Der letzte Layer besteht aus lediglich 2 Neuronen, die eine Fließkommazahl ausgeben, welche jeweils die relative X und Y-Position des erkannten Fingers repräsentiert. Zwischen den Hidden Dense Layern ist jeweils ein Dropout Layer positioniert, deren Konfiguration im Abschnitt 3 beschrieben wird.

Dropout Dropout ist ein Mechanismus um Overfitting zu reduzieren. Dabei wird beim Training für jeden Batch ein fest konfigurierter Anteil Neuronen zufällig ausgeschaltet und ihre Kanten ausgeblendet. Dadurch werden viele verschiedene Netze gleichzeitig trainiert und bei späterer Inferenz wird effektiv ein Mittelwert aus den vielen Einzelnetzen gebildet, der statistisch eine bessere Lösung liefert (*"... for regression with linear output units, the squared error of the mean network is always better than the average of the squared errors of the dropout networks."* [22, S.2]). Dies sei erfolgreich, da jedes Neuron somit eine nützliche Repräsentation darstellen muss und nicht erst in Kombination mit anderen Neuronen des selben Layers ein Feature repräsentiert oder sich neutralisiert (*"... it [dropout] encourages each individual hidden unit to learn a useful feature without relying on specific other hidden units to correct its mistakes."* [22, S.9]).

3. Training

3.1 Trainingsdatenerzeugung

Das Netzwerk wird durch Supervised Learning trainiert. Deshalb müssen Trainingsdaten mit korrekten Labels erzeugt werden.
Dafür wurde eine Applikation für das eingebettete System geschrieben, die zufällig Spielsteine auf dem Othello Brett anordnet und über den Projektor ausgibt. Der Bediener muss dann seine Fingerspitze auf ein beliebiges, freies Feld positionieren und die Enter Taste auf der angeschlossenen Tastatur drücken. Beim Drücken der Taste wird ein Foto aufgenommen, aus dem das extrahierte Spielbrett gespeichert wird. Anschließend wird ein neues, zufälliges Spielbrett generiert. Dabei sind die erzeugten Spielsteinbelegungen nicht zwangsläufig gültige Othello Spielsituationen.
Es werden zufällige Brettbelegungen generiert, um realistische Anwendungsbedingungen zu

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 224, 224, 3)]	0
mobilenetv2_1.00_224 (Functi	(None, 7, 7, 1280)	2257984
global_average_pooling2d (Gl	(None, 1280)	0
dense (Dense)	(None, 320)	409920
dropout (Dropout)	(None, 320)	0
dense_1 (Dense)	(None, 80)	25680
dropout_1 (Dropout)	(None, 80)	0
dense_2 (Dense)	(None, 20)	1620
dense_3 (Dense)	(None, 2)	42

Total params: 2,695,246
Trainable params: 437,262
Non-trainable params: 2,257,984

Abbildung 4: Architektur des verwendeten Netzwerks. Die Gewichte des Mobilenets sind eingefroren.

erzeugen und das Netzwerk lernen kann, die Steine nicht als Finger zu interpretieren.
Nach der Datensammlung wird für jedes Foto mit dem Tool LablelImg[23] die Fingerspitze mit einem Rechteck markiert. Anschließend wird der Mittelpunkt dieses Rechtecks als relative Fingerspitzenposition errechnet. Die Fingerspitzenposition ist dadurch der Mittelpunkt des Fingernagels. Die obere linke Ecke des Bildes entspricht dem Punkt (x: 0 ; y: 0) und die untere Rechte Ecke dem Punkt (x: 1 ; y: 1).
Abbildung 5 zeigt ein Trainingsbild in der Auflösung 224x224 Pixel, bei dem die zu lernende Position durch einen blauen Punkt visualisiert ist.
Der erzeugte Datensatz besteht aus 718 annotierten Fotos.

3.1.1 Data Augmentation

Durch Data Augmentation können aus einem vorhandenen Datensatz weitere Trainingsdaten generiert werden. Dadurch hat das Netzwerk eine größere Datengrundlage um zu generalisieren und somit die eigentliche Aufgabe zu erkennen *"The task of Data Augmentation is to bake these translational invariances into the dataset such that the resulting models will perform well despite these challenges."*⌈⌊*24, S.4]*.
Bei Data Augmentation muss darauf geachtet werden, die Labels der generierten Daten gültig bleiben. Deshalb erfolgt eine Augmentation durch rotieren der Bilder in 90° Schritten. Durch den begrenzten Bereich der Fingerposition, können die Label ebenfalls um jeweils 90° transformiert werden. Durch den Extraktionsprozess 2.4 sind die Spielbretter immer parallel zu den Achsen ausgerichtet, wodurch eine Augmentation durch Rotation um andere Winkel unbrauchbar ist. Aus dem selbe Grund sind auch Verschiebungen oder Zuschnitte nicht möglich.
Die Anpassung der Helligkeit oder das Einfügen von Rauschen könnten gültige Möglichkeiten

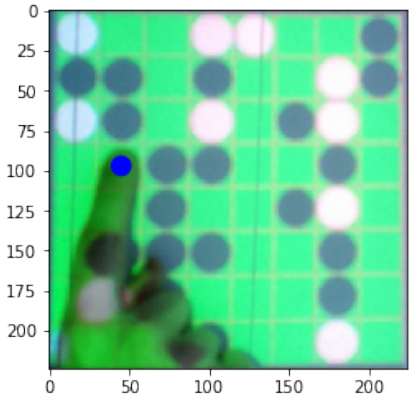


Abbildung 5: Beispielinput für das Netzwerk. Die Fingerspitze (x: 0.2 ; y: 0.433) ist durch den blauen Punkt für den Leser visualisiert. Das Koordinatensystem zeigt die Position in Pixeln.

sein, die jedoch in dieser Arbeit nicht untersucht wurden.
Durch die Augmentation stehen 4 * 718 = 2872 annotierte Bilder zur Verfügung.

3.2 Trainingsablauf

Für das Training wird der angereicherte Datensatz aufgeteilt in 80% Trainings- und 20% Validierungsdaten. Das Transfer Learning findet in 2 Phasen statt. In Phase 1 werden alle Gewichte des Mobilenets eingefroren, um ausschließlich den hinzugefügten Regressionsteil zu trainieren. Dadurch soll dieser Teil gezielt dazu gebracht werden aus den verfügbaren Featuremaps zwei Positionskoordinaten zu berechnen. Ansonsten könnte der Fehler, der größtenteils im zufällig initialisierten Regressionsteil entsteht, die Featuremaps des Mobilenets ungewollt verändern. Zudem beschleunigt dies das Training einer Epoche, da nur 16% der insgesamt 2.695.246 Parameter angepasst werden müssen. Sobald der Validation Loss nicht mehr signifikant steigt, wird die erste Phase beendet.
In der zweiten Phase beginnt das fine tuning des gesamten Netzwerks. Dabei werden alle Parameter zur Anpassung freigegeben. Der ImageNet[14] Datensatz, auf dem das Mobilenet vortrainiert wurde, besteht aus Fotos zu 1000 verschiedenen Klassen, vorrangig Tiere und Alltagsobjekte. Die Problemdomäne für die Fingerpositionsbestimmung ist jedoch deutlich spezieller. Es gibt immer einen vergleichbaren Hintergrund und es muss nur eine Klasse lokalisiert werden. Deshalb ist die Erwartung, durch das fine tuning des gesamten Netzwerks, das Mobilenet als Feature Extraktor spezialisieren zu können. Durch diese Spezialisierung soll ein geringerer Validation Loss erreicht werden.
Um eine geeignete Dropout Wahrscheinlichkeit zu ermitteln, wurde das Netzwerk mit drei verschiedenen Parametern trainiert. Es werden die in [22, S.2] genutzten Parameter 50% (Empfehlung für Hidden Layer) und 20% (Empfehlung für Input Layer), sowie 0% getestet, um eine Referenz ohne Dropout zu erhalten.
Die Trainingsdauer wurde durch Bewertung vorheriger Tests abgeleitet, sodass in Phase 1 über 30 Epochen und in in Phase 2 150 Epochen trainiert wird.

3.2.1 Hyperparameter

Die Tabelle 1 zeigt die festen Hyperparameter des Netzwerks für den Trainingsvorgang. Als Optimizer wird Adam[25] in der Standardkonfiguration von Keras genutzt, da dieser ein effizientes Lernen für ein breites Spektrum von Anwendungfsällen ermöglicht. Der Mean Squared Error bietet sich als Kostenfunktion an, da er ein positives Maß für die quadratische euklidische Distanz zwischen vorhergesagtem und tatsächlichem Label liefert. Die Batchgröße von 16 ist klein gewählt, wodurch grundsätzlich ein geringerer Loss-Value bei gleicher Anzahl Epochen erreicht werden soll (*"... the final loss values are lower as the batch sizes decrease ..."*[26, S.8]) und das Training auch auf einer Laptop-GPU mit 1GB VRAM ermöglicht. Trotz der potentiell besseren Loss-Werte ist es dennoch empfohlen Batchgrößen so zu wählen, dass der GPU Speicher maximal ausgenutzt wird, da dies die Ausführunggeschwindigkeit erhöhe und somit in Kombination mit höheren Lernraten in gleicher Zeit ein geringerer Loss erreiche (*"this report recommends you use a batch size that fits in your hardware's memory and enable using larger learning rates."*[26, S.8]).
Als Aktivierungsfunktion wird Rectified Linear Unit (ReLu) genutzt, da diese sich als Standard für Hidden Dense Layer in Keras etabliert hat.

Parameter	**Wert**
Optimizer	Adam (Keras Default Parameter)
Kostenfunktion	Mean Squared Error
Batchgröße	16
Aktivierungsfunktion	ReLu

Tabelle 1: Feste Hyperparameter des neuronalen Netzwerks

3.3 Ergebnisse

Phase 1 benötigt eine Zeit von etwa 1'50" und Phase 2 knapp über 33 Minuten in einer von der HAW Informatik Compute Cloud (ICC) bereitgestellten Umgebung mit GPU Beschleunigung. Die Abbildung 6 zeigt die Verläufe der Loss-Funktionswerte in Abhängigkeit der Trainingsepochen. Die blauen Linien zeigen dabei den jweiligen Loss für den Trainingsdatensatz, während die orangenen Linien den Fehler bei der Überprüfung mit dem Validierungsdatensatz anzeigen. Die erste Zeile der Darstellungen zeigt die Trainingsverläufe der ersten Phase, in der über 30 Epochen nur die neu hinzugefügten Schichten trainiert werden. Die zweite Zeile der Diagramme zeigt die Verläufe während der zweiten Phase, in der das gesamte Modell aus der ersten Phase für 150 Epochen weiter trainiert wird.
In jeder Spalte ist ein Netzwerk mit unterschiedlich gewähltem Dropout abgebildet. Das Modell ohne Dropout ist links, in der Mitte das Modell mit 20% und rechts das Modell mit 50% Dropout.
Die Kurven sind mit einem gleitenden Mittelwert der Fenstergröße 3 geglättet, um den Trend besser erkennen zu können.
Das Netz mit 0% Dropout beginnt in Phase 1 mit einem Training und Validation Loss von etwa 0,2 und verbessert sich bis Epoche 15 nicht merklich. Ab Epoche 16 beginnt eine logarithmisch abnehmende Verbesserung beider Werte. Ab diesem Zeitpunkt wächst die Differenz zwischen dem Trainings- und Validierungsdatensatz kontinuierlich. Bei Epoche 30

wird ein Training Loss von 0,04 und ein Validation Loss von 0,064 erreicht.
Das Netzwerk mit 20% Dropout beginnt direkt mit einer logarithmischen Verbesserung der Werte und nähert sich bei Epoche 30 einem Training Loss von 0,06 und einem Validationloss von 0,068 an. In den ersten 3 Epochen ist der Validation Loss niedriger als der Training Loss.
Das Netzwerk mit 50% Dropout beginnt mit einem Training Loss von 0,3289, welches es bis Epoche 30 auf 0,3249 verbessern kann. Der Validation Loss ist konstant bei 0,3281.

In Phase 2 reduziert das Netzwerk mit 0% Dropout seine Fehler deutlich innerhalb der ersten 40 Epochen auf einen Training Loss von $2,5 * 10^{-4}$ und Validation Loss von $4,5 * 10^{-4}$. Anschließend steigen beide innerhalb von einer Epoche wieder auf $4,6 * 10^{-2}$ und $1,3 * 10^{0}$ an. Die Werte reduzieren sich wieder langsam auf ein etwas geringeres Minimum, bevor Sie erneut sprunghaft ansteigen. Dieses schwingende Verhalten setzt sich bis zu Epoche 150 fort. Zu diesem Zeitpunkt ist ein Trainingsloss von $9,4 * 10^{-5}$ und ein Validationloss von $3,6 * 10^{-4}$ erreicht.
Das Netzwerk mit 20% Dropout verhält sich ähnlich, jedoch sind die lokalen Minima um etwa Faktor 10 größer, sodass am Ende ein Trainingsloss von $8,1 * 10^{-4}$ und ein Validationloss von $2,3 * 10^{-3}$ erreicht wird.
Im rechten Diagramm ist ein konstanter Verlauf für das Netzwerk mit 50% Dropout zu sehen. Der Training Loss liegt bei $3,249 * 10^{-1}$ und der Validation Loss beträgt $3,281 * 10^{-1}$. Das Training für das Netzwerk mit 50% Dropout wurde wiederholt, um das Ergebnis zu verifizieren und die Wahrscheinlichkeit einer seltenen, ungünstigen Gewichtsinitialisierung oder anderer, technischer Fehler zu reduzieren.

3.4 Auswertung

In Phase 1 zeigt das Netzwerk mit 0% Dropout nach der Verbesserung ab Epoche 15 eine Entwicklung von Overfitting. Dies ist durch die steigende Differenz zwischen Trainings- und Validierungskosten zu erkennen. Das Netzwerk mit 20% Dropout lernt zu Beginn schneller und nähert sich dann langsam einem vergleichbaren Validation Loss an. Dabei verbessert sich jedoch der Training Loss nicht signifikant stärker, was auf weniger Overfitting hinweist. Das Netzwerk mit 50% Dropout beginnt mit ähnlichen Werten nach der ersten Epoche, bleibt jedoch konstant. Hier konnte das Netzwerk nicht weiter lernen. Da auch der Training Loss konstant bleibt, ist hier von underfitting auszugehen. Dies könnte bedeuten, dass die durch den Dropout kleineren Hidden Layer nicht in der Lage sind die Positionsbestimmung zu erlernen oder die Lernrate in Kombination zu gering ist (*"Increasing the learning rate moves the training from underfitting towards overfitting."*[Kap.3.2][26]).
Ein Dropout von 0% und 20% produziert somit in Phase 1 einen vergleichbaren Validationloss, wobei 20% Dropout noch kein Overfitting erreicht hat und weiter trainiert werden könnte, weshalb es für Phase 1 die bessere Konfiguration darstellt.

In Phase 2 verbessert sich das Netzwerk mit 50% Dropout nicht. Es gilt die selbe Schlussfolgerung wie bereits in Phase 1.
Der Loss der Netze mit 0% und 20% Dropout schwingt deutlich. Nach den starken Anstiegen wird am Ende der Periode jedoch immer ein neues Minimum erreicht. Die Schwingung

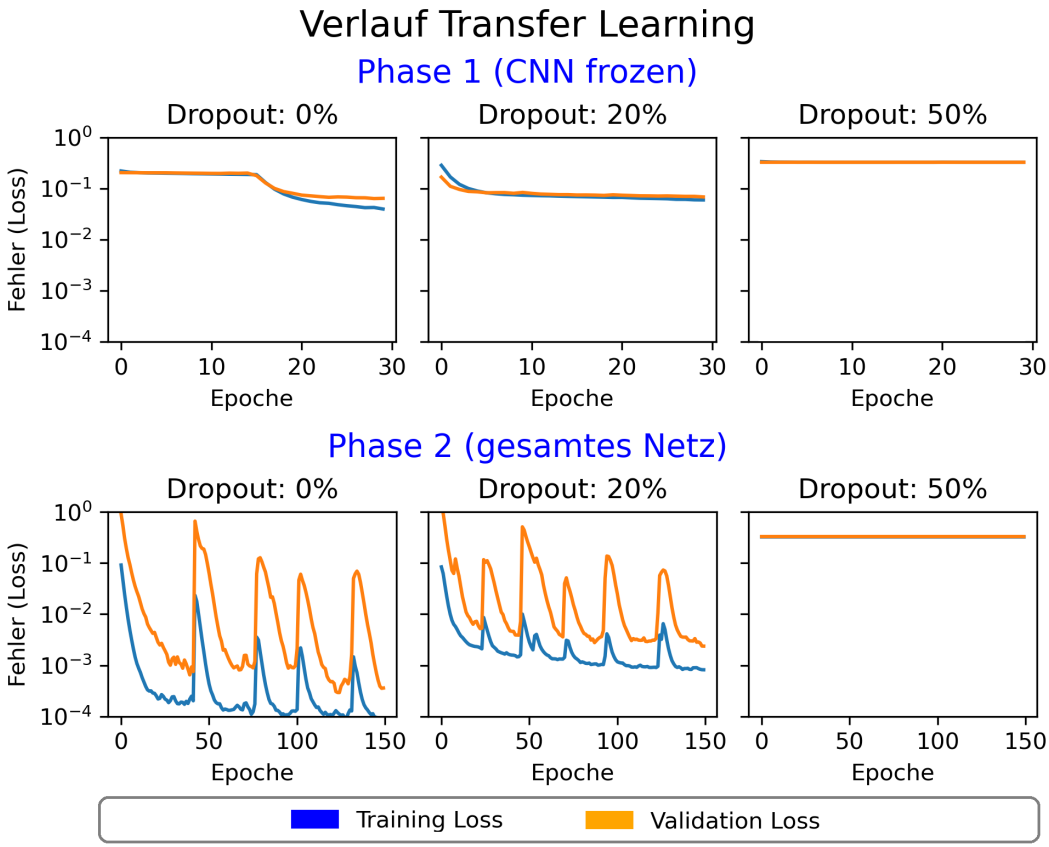


Abbildung 6: Trainingsverläufe für Phase 1 und 2 mit verschiedenen Dropout-Raten.

ist durch das Zusammenspiel der relativ kleinen Batchgröße 16 und dem Adam Optimizer zu erklären. Der schnelle Anstieg kann dann dadurch entstehen, dass für einen Batch, der nicht repräsentativ für den gesamten Trainingsdatensatz ist, eine Anpassung der Gewichte vorgenommen wurde, die für die restlichen Trainingsdaten erheblich schlechter ist. Da Adam die Lernrate an die Steigung des Losses anpasst, steigt diese durch die große Änderung an. Es wird somit ein schneller Anstieg des Fehlers, aber auch eine schnelle Annäherung an das vorherige niedrigere Niveau erreicht. Hier könnte die Erhöhung der Batchgröße hilfreich sein, um die Schwingungen zu vermeiden.
Bei beiden konnte durch das fine tuning des gesamten Netzwerks der Validation loss signifikant verkleinert werden. Bei dem Netzwerk mit 0% Dropout um den Faktor 177, bei 20% um Faktor 28 bei vergleichbaren Startwerten nach Phase 1. Das fine tuning hat somit dem Netzwerk erheblich geholfen die Fingerposition zu bestimmen.
Das Netzwerk mit 0% Dropout liefert nach 150 Epochen den geringeren Validation Loss, weshalb es der beste Kandidat für die weitere Evaluation ist. Jedoch ist der Abstand zwischen Trainings- und Validationloss doppelt so groß wie bei dem Netzwerk mit 20% Dropout.

Dies bedeutet, dass es ein höheres Overfitting aufweist und dadurch weniger Potential für Verbesserung bei noch längerem Training besitzt.

3.4.1 Bewertung des Fehlers

Durch die Nutzung des mittleren quadratischen Fehlers (MSE) als Kostenfunktion für das Netzwerk, beschreibt der Loss einer Epoche den mittleren quadratischen Abstand zwischen der tatsächlichen Fingerposition und der von dem Netz prognostizierten Position. Durch Berechnung der Quadratwurzel, lässt sich die mittlere Positionsabweichung (ME) errechnen. Für den Validationloss des besten Netzwerks mit 0% Dropout nach Phase 2 von $3,6 * 10^{-4}$ bedeutet dies: $ME = \sqrt{3,6 * 10^{-4}} \approx 0,019$. Diese ist so zu interpretieren, dass im Mittel über den gesamten Trainingsdatensatz die prognostizierte Position um 1,9% von der Korrekten abweicht. Bei dem konkreten Anwendungsfall Othello, wird das Bild in 8x8 Felder eingeteilt, sodass ein Spielfeld 12,5% des Bildes breit und hoch ist. Somit ist die mittlere Abweichung der Positionsbestimmung 15% einer Spielfeldlänge $(0,019/0,125 = 0,152)$ groß. Solange der Spieler seinen Finger in der Mitte eines Spielfeld positioniert, sollte somit das korrekte Spielfeld identifizierbar sein. Die Präzision des Netzwerks ist somit bereits für die weitere Evaluation ausreichend. Das Netzwerk mit 0% Dropout wird für nachfolgende Evaluationen genutzt.

4. Evaluation

4.1 Inferenz

Um das Modell effizient auf dem Raspberry Pi mit Unterstützung des Coral-Sticks ausführen zu können, muss es zunächst konvertiert werden. Im ersten Schritt wird das Tensorflow-Modell in ein quantisiertes Tensorflow-Lite-Modell (TFLite-Modell) konvertiert. Bei der 'Post-Training-Quantization' werden alle Gewichte von 32-Bit Fließkommazahlen in 8-Bit Integer konvertiert. Dafür wird der Wertebereich der vorhandenen Gewichte auf die verfügbaren Ganzzahlen skaliert, sodass jede Fließkommazahl dem nächstgelegenen Integer zugewiesen werden kann. Dadurch reduziert sich der benötigte Speicherplatz auf ein Viertel, wodurch sich auch die Geschwindigkeit der Inferenz erhöht. Der Outputlayer wird nicht quantisiert, sodass das Netzwerk weiterhin direkt die relativen Positionen zwischen 0 und 1 ausgibt.
Das TFLite Modell wird anschließend für die Edge-TPU des Coral-Sticks kompiliert. Die spätere Inferenz wird von einen TFLite Interpreter durchgeführt, der alle kompatiblen Operationen auf dem Coral-Stick ausführt.

4.1.1 Laufzeitmessungen

Der Boxplot 7 zeigt die Messergebnisse von Laufzeitmessungen für jeweils eine Inferenz des Netzwerks mit unterschiedlicher Hardwarekonfiguration. Dabei wurde das TFLite Modell auf der Raspberry Pi CPU selbst ausgeführt, das kompilierte Modell auf der über USB 2.0 angeschlossene Edge-TPU ausgeführt und die Anbindung der TPU über USB 3.0 getestet. Je Experiment wurden 11 Inferenzen durchgeführt. Die Y-Achse des Boxplots ist logarithmisch skaliert und gibt die Laufzeit in Millisekunden an.
Die Ausführung auf der CPU ist erwartungsgemäß am langsamsten und benötigt im Me-

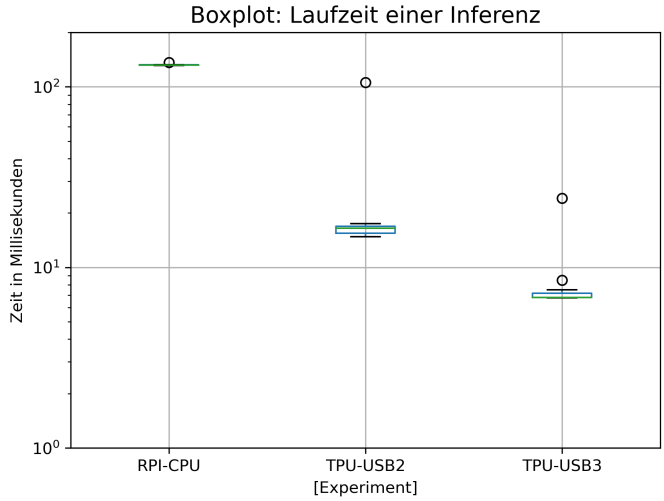


Abbildung 7: Boxplot: Laufzeitmessung für eine Inferenz mit unterschiedlicher Hardwarekonfiguration

dian 132 Millisekunden für die Ermittlung einer Fingerposition. Darauf folgt die Inferenz mit der über USB 2.0 angebundenen TPU, die im Median 16ms benötigt. Auffällig ist hier die erste Inferenz die noch 105ms benötigt hat. Auch die Anbindung über USB 3.0 zeigt einen Ausreißer bei der ersten Inferenz von 24ms und ist mit einen Median von 6,8ms die schnellste Konfiguration.
Die verhältnismäßig lange erste Inferenzdauer bei Verwendung der TPU ist durch den Kopiervorgang des Netzwerks auf diese zu erklären. Die Ergebnisse zeigen, dass die Verwendung des Coral-Sticks für diesen Anwendungsfall einen erheblichen Performancegewinn von Faktor 20 gegenüber der reinen CPU-Inferenz bringt. Beim Anschluss des Coral-Sticks ist auf die Verwendung von USB 3 zu achten, da dies einen merklichen Einfluss auf die Performance hat.
Mit einer mittleren Inferenzdauer von etwa 7 Millisekunden für die Bestimmung einer Fingerposition aus dem bereits ausgeschnittenen Foto ist das System schnell genug für die Echtzeitanwendung. Es ist jedoch zu beachten, dass der Extraktionsprozess zusätzliche Zeit in Anspruch nimmt.

4.2 Evaluation des Gesamtsystems

Für die praktische Evaluation wurden 5 Spiele gegen einen Computergegner gespielt. Alle Spiele gingen bis zur vollständigen Brettbelegung mit 64 Steinen, weshalb pro Spiel 30 menschliche Züge durch Fingerpositionserkennung gemacht wurden. Aus den somit 150 manuellen Zügen waren 7 gewünschte Positionen nicht oder erst nach anderer Fingerpositionierung möglich. 6 dieser nicht erkannten Positionen lagen auf Feldern am Spielbrettrand. Somit ergibt sich für den konkreten Anwendungsfall eine Accuracy von $143/150 = 95,3\%$

korrekt erkannter Spielfelder.
Während des Versuchs wurde von der Fingererkennung nie ein Stein gelegt, ohne dass ein Finger auf einem Feld platziert war. Die in Abschnitt 2.3 beschriebene Detektion der Fingeranwesenheit funktioniert somit in der Praxis gut.

4.2.1 Veränderte Umgebungsbedingungen

Ein Versuch in deutlich hellerer Umgebung als bei der Aufnahme der Trainingsdaten schlug fehl. Zum einen erkannte der Extraktionsprozess (2.4) das Spielbrett erst nach manueller Anpassung der Parameter, des Weiteren konnte das neuronale Netz die Fingerposition nicht zuverlässig aus den kontrastarmen Bildern ermitteln.
Auch der Versuch auf einem Holztisch mit stark sichtbarer Maserung zu spielen schlug fehl, da das Netz die Fingerposition zu unpräzise ermittelte.
Eine Vergrößerung des Abstands des Systems zur Projektionsfläche funktionierte jedoch, solange das projizierte Bild kontrastreich genug bleibt. So konnte auch auf einem Spielfeld mit verdoppelter Seitenlänge die Fingerposition problemfrei erkannt werden.
Um dem Problem mit Hintergrundmustern entgegenzuwirken sowie mit unterschiedlichen Kontrastleveln umgehen zu können, sollte der Trainingsdatensatz um diese Situationen erweitert werden.

5. Schluss

5.1 Zusammenfassung der Ergebnisse

Es wurde ein mobiles, eingebettetes System konstruiert, welches durch optische Fingerpositionserkennung über ein projiziertes Benutzerinterface bedient werden kann. Als Anwendungsbeispiel wurde das Brettspiel Othello realisiert. Dafür wird zunächst durch einen Extraktionsprozess das projizierte Spielfeld aus dem Kamerabild extrahiert und normiert. Auf diesem Bild wird durch ein angepasstes MobilenetV2 die relative Fingerposition ermittelt.
Durch Transfer Learning konnte das Netz schnell auf eine Abweichung von 1,9% der relativen Position trainiert werden, welche im Anwendungsfall nur 15% einer Spielfeldlänge entspricht. Beim Vergleich verschiedener Dropout-Raten zwischen den Hidden Layern wurde das Overfitting durch mehr Dropout wie erwartet reduziert. Jedoch waren die Validierungsergebnisse um Größenordnungen schlechter als ohne Dropout, sodass im weiteren Verlauf kein Dropout mehr genutzt wurde. Bei Trainingsphase 2 trat ein schwingendes Verhalten der Kostenfunktionswerte auf, welches auf ein ungünstiges Zusammenspiel von Batchgröße und Adam Optimizer zurückgeführt wird.
Die Nutzung des Coral-Sticks beschleunigt die Inferenz um das 20-fache gegenüber einer Ausführung auf der CPU und ist mit einer Inferenzdauer von 7 Millisekunden schnell genug für die praktische Anwendung.
Die praktische Evaluation zeigte, dass sich das Othello Spiel allein durch Platzierung des Fingers vollständig bedienen lässt, solange die Lichtverhältnisse nicht deutlich von den Trainingssituationen abweichen.

5.2 Ausblick

Weiterführend kann durch ausführliche Untersuchung der Hyperparameter die Effizienz und das Verständnis des neuronalen Netzwerks weiter gesteigert werden. Für die Positionsbestimmung kann evaluiert werden, wie gut die Abbildung auf ein Klassifizierungsproblem für die Spielfelder funktioniert und ob dadurch der vorangehende Extraktionsprozess entfallen könnte.
Auch ohne Anpassung der Netzarchitektur würde eine Erweiterung der Trainingsdaten durch geeignete echte Aufnahmen oder weitere Augmentationen dazu beitragen, das System auch bei schwächerem Kontrast oder bemusterten Projektionsflächen einsetzen zu können. Um einer Diskriminierung von Anwendern vorzubeugen, sollten Trainingsdaten mit Händen von unterschiedlichen Geschlechtern, Altersgruppen und Hautfarben aufgenommen werden.

Literatur

[1] J. Ravikiran, K. Mahesh, S. Mahishi, R. Dheeraj, S. Sudheender, and N. V. Pujari, "Finger detection for sign language recognition," in *Proceedings of the international MultiConference of Engineers and Computer Scientists*, vol. 1, no. 1, 2009, pp. 18–20.

[2] J. L. Raheja, K. Das, and A. Chaudhary, "An efficient real time method of fingertip detection," *arXiv preprint arXiv:1108.0502*, 2011.

[3] X. Liu, Y. Huang, X. Zhang, and L. Jin, "Fingertip in the eye: A cascaded cnn pipeline for the real-time fingertip detection in egocentric videos," *arXiv preprint arXiv:1511.02282*, 2015.

[4] M. Dani, G. Garg, R. Perla, and R. Hebbalaguppe, "Mid-air fingertip-based user interaction in mixed reality," in *2018 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. IEEE, 2018, pp. 174–178.

[5] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, C.-L. Chang, and M. Grundmann, "Mediapipe hands: On-device real-time hand tracking," *arXiv preprint arXiv:2006.10214*, 2020.

[6] Sony, "Xperia touch technische daten," 2017. [Online]. Available: https://www.sony.de/electronics/support/smart-sports-devices-xperia-smart-devices/xperia-touch/specifications

[7] M. Schmittchen and A. Avdullahu, "Evaluating portable touch projectors in the context of digital education," in *Learning and Collaboration Technologies. Designing, Developing and Deploying Learning Experiences*, P. Zaphiris and A. Ioannou, Eds. Cham: Springer International Publishing, 2020, pp. 169–178.

[8] Coral.ai, *USB Accelerator datasheet*, 1st ed., 2019. [Online]. Available: https://coral.ai/docs/accelerator/datasheet/

[9] N. Ltd., "Nebra anybeam hat - laser projector hat for raspberry pi," 2016. [Online]. Available: https://www.nebra.com/products/nebra-anybeam-hat-laser-projector-for-the-raspberry-pi

[10] R. P. Foundation, "Camera module 2," 2016. [Online]. Available: https://www.raspberrypi.org/products/camera-module-v2/

[11] M. Elgendy, *Deep Learning for Vision Systems*. Simon and Schuster, 2020.

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[16] Coral.ai, "Tensorflow models on the edge tpu," 2020. [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/#supported-operations

[17] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[18] Coral.ai, "Models - image classification — coral," 2020. [Online]. Available: https://coral.ai/models/image-classification/

[19] keras, "Keras applications." [Online]. Available: https://keras.io/api/applications/

[20] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[21] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *arXiv preprint arXiv:1411.1792*, 2014.

[22] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.

[23] tzutalin, "Labelimg," 2021. [Online]. Available: https://github.com/tzutalin/labelImg

[24] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay," *arXiv preprint arXiv:1803.09820*, 2018.