Lösen eines Sudokus durch Bildverarbeitung und Klassifizierung

Cedric Stolze

Hochschule für Angewandte Wissenschaften Hamburg
cedric.stolze@haw-hamburg.de

28 Februar 2021

Zusammenfassung. Für diese Arbeit wurde eine Android Anwendung geschrieben, welche ein Bild von einem Sudoku lösen kann. Hierfür wurde Bildverarbeitung genutzt, um zum einen das Sudokugitter zu extrahieren und zum anderen die einzelnen Ziffern in den Zellen zu isolieren. Die Ziffern werden durch ein Convolutional Neural Network klassifiziert, welches auf Basis eines erweiterten MNIST Datensatzes trainiert wurde. Schlussendlich kann das übertragene Sudoku durch einen rekursiven Backtracking Algorithmus gelöst werden. Die Leistung der Anwendung wurde mit 100 Testbildern ausgewertet. Der Datensatz umfasst Bilder mit unterschiedlichen Qualitäten und potentiellen Störfaktoren. Die Anwendung konnte 94,6% aller Zellen in dem Testdatensatz korrekt klassifizieren. Limitierung traten vor allem in der Extraktion von zu stark gekrümmten Sudokugittern auf. Die Klassifizierung der Ziffern hatte einige Probleme mit der Erkennung einzelner Zahlen, wobei sich die Fehlerkennung auf unter 10% beschränkt.

Schlüsselwörter: Sudoku · Android · Bildklassifizierung · Convolutional Neural Network · Bildverarbeitung · Datenerweiterung

1 Einleitung

Um die Qualität von Machine Learning Anwendungen zu verbessern welche Bilddaten verarbeiten, ist eine umfangreiche Datenaufbereitung ein entscheidender Faktor. Hierdurch können Fehlerquellen in der Bildklassifikation minimiert werden, indem Störquellen im Vorfeld entfernt werden[6]. Dies soll am Beispiel von Sudoku in dieser Arbeit behandelt werden.
Sudoku ist ein Logikrätsel in einem 9x9 Gitter, welches zusätzlich in neun 3x3 Quadranten unterteilt ist. In allen 81 Zellen des Gitters darf jeweils eine Ziffer zwischen eins und neun stehen. Zu Beginn des Rätsels sind ein paar wenige Ziffern vorgegeben. Um es zu lösen, dürfen in jeder Spalte, Zeile und jedem Quadranten alle neun Ziffern nur ein mal vorkommen[10].
Klassischerweise werden Sudokus in Zeitschriften und Rätselheften gelöst[7], ähnlich wie Kreuzworträtsel mit Stift auf Papier. Für den Fall, dass man bei dem Rätsel nicht weiterkommt oder die Lösung überprüfen lassen möchte, muss das Gitter in einen Sudokurechner übernommen werden. Die 81 Zellen zu übertragen

kann jedoch einiges an Zeit kosten. Deshalb widmet sich diese Arbeit der korrekten Übertragung des Rätsels aus einem Bild in digitale Form. Dies wurde in Form einer Android Anwendung umgesetzt.
Die Herausforderung hierbei liegt darin das Gitter zu erkennen und die einzelnen Ziffern innerhalb der Zellen zu klassifizieren. Die Umsetzung wurde bereits in einem Blogeintrag[4] von Aakash Jhawar behandelt, welcher mithilfe von Python, Bildverarbeitung und Machine Learning eine Implementation geschrieben hat. Für das Projekt wurde auf Basis dessen eine Umsetzung für Android und Java geschrieben, sowie die Leistung anhand eines Testdatensatzes ausgewertet. Einige andere Arbeiten galten ebenfalls als Orientierung, implementierten jedoch keine Aspekte im Bereich Machine Learning[5][2].
Ziel des Projekts war es, über 90% der Zellen eines Sudokugitters korrekt aus einem Bild zu erkennen. Dabei soll die Anwendung mit sowohl gedruckten als auch handschriftlichen Ziffern umgehen können und leere Zellen erkennen. Zu dem soll die Klassifizierung sich stabil gegen Störungen verhalten. Das heißt, schlechte Lichtverhältnisse, leicht gebogene Rätsel und Perspektiven, welche das Sudoku verkrümmen. Diese Arbeit soll die verwendeten Techniken, Schritte und Prozesse beleuchtet, sowie das Projektergebnis evaluieren.

2 Projektüberblick

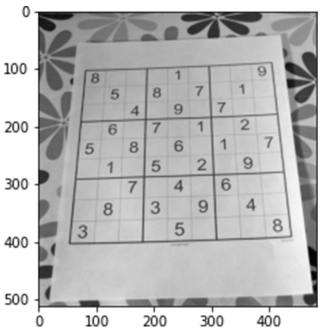
Das Projekt umfasst mehrere Teilgebiete. Zum einen die Umgebung, welche eine Android Anwendung ist. Die Anwendung wurde für Android 11 mit der Andrid SDK 30 in Java 1.8 geschrieben. Mit der App soll es möglich sein, ein Bild mit einem Sudokurätsel zu erkennen, die Zahlen zu erfassen und das Rätsel zu lösen. Innerhalb der Umgebung ist das Projekt in drei Funktionalitäten aufgeteilt.
Ein Bereich befasst sich mit Bildverarbeitung, welches dafür zuständig ist das Sudokugitter und die enthaltenen Zellen zu extrahieren und leere Zellen zu erkennen. Hierfür wird die Programmbibliothek OpenCV verwendet, welches bestimmte Algorithmen für die Bildverarbeitung bereitstellt. Verwendet wurde die OpenCV Version 3.4.11. Das Ziel in diesem Teilbereich ist eine Liste an Bildern aller Zellen aus dem Sudoku, welche für die Bildklassifikation aufbereitet sind. Die Basis der Umsetzung basiert auf dem in der Einleitung erwähnten Blog[4] und wurde für Java angepasst.
Der zweite Teilbereich umfasst das Klassifizieren der Bilder mit einem neuralen Netzwerk. Trainiert wurde das Modell in einer Python 3.7.9 Umgebung mithilfe von Keras und Tensorflow 2.0. Das Netzwerk ist in der Lage, handschriftliche Ziffern zu klassifizieren. In einer komprimierten Form kann das trainierte Modell für Embedded Machine Learning in Android Anwendungen genutzt werden. Es werden hier alle vorbereiteten 81 Bilder klassifiert und in eine Matrix gespeichert. Für das Lösen wird ein rekursiver Backtracking-Algorithmus[3] genutzt, auf welchen in dieser Ausarbeitung nicht tiefer eingegangen wird. Der Fokus der Arbeit liegt auf der Bildverarbeitung und Klassifizierung eines Sudokus.

3 Bildverarbeitung

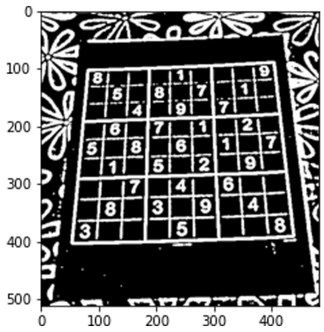
3.1 Gitterextraktion

Im ersten Schritt der Bildverarbeitung wird aus einem Eingabebild das dort enthaltene Sudokugitter extrahiert. Als Eingabe wird ein Bild erwartet, bei welchem das Sudoku vollständig sichtbar und nicht zu stark gekrümmt ist.
Das Eingabebild wird vorerst soweit aufbereitet, dass es weiterverarbeitet werden kann und nur die erforderlichen Informationen erhalten bleiben. Es werden alle Farben aus dem Eingabebild gezogen und diese in einen Graukanal transformiert. Für die weitere Bildverarbeitung werden RGB Farbinformation keine Rolle spielen und die Bildklassifikation erwartet nur einen Graukanal. Zusätzlich werden alle Pixel je nach Grauwert auf komplett Schwarz oder Weiß gesetzt und invertiert. Nach der Farbanpassung werden vorhandene Formen und Linien betont. Das Ziel ist es, Bildfehler und Linienunterbrechungen auszugleichen. Dies wird erreicht, indem zum einen ein Gaußscher Weichzeichnungsfilter verwendet wird um kleinere Pixelstörungen durch Unschärfe zu verwischen. Zum anderen durch die Linienverstärkung von OpenCV.

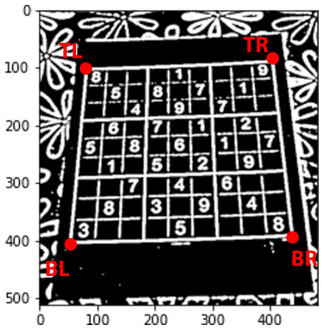
Um das Sudokugitter zu extrahieren werden die vier Eckpunkte des Gitters verwendet. Hierfür stellt OpenCV eine Funktion bereit, welche alle zusammenhängenden Konturen in einem Bild identifiziert. Diese werden nach ihrer einnehmenden Fläche im Bild absteigend sortiert. Das Sudokugitter wird dadurch erkannt, dass es die Rechteckkontur mit dem größten Flächeninhalt ist. Eine Kontur wird in OpenCV als ein Array an Punkten definiert. Sei ein Punkt P definiert durch $P = (x, y)$, wobei x ein Wert innerhalb des Breite w und y ein Wert innerhalb der Höhe h des Bildes ist. Die oberste linke Ecke des Bildes hat den Koordinatenpunkt $P_{min} = (0, 0)$ und die Ecke rechts unten wird durch den Punkt $P_{max} = (w - 1, h - 1)$ beschrieben. Die vier Eckpunkte der größten Rechteckkontur R werden wie folgt aus den Koordinaten $(x, y) \in R$ definiert. Der Eckpunkt links oben P_{tl} ist der kleinste Wert aus $(x + y)$ und der Punkt rechts unten P_{br} ist der größte Wert. Der Eckpunkt rechts oben P_{tr} ist wiederum der größte Wert aus $(x - y)$ und der links unten P_{bl} ist der Kleinste. Die resultierenden Eckpunkte werden in Abbildung 1c dargestellt.
Mithilfe aller Eckpunkte $C_R = \{P_{tl}, P_{tr}, P_{bl}, P_{br}\}$ von R wird das Sudokugitter ausgeschnitten und perspektivisch angepasst, sodass ein quadratisches Bild mit dem Sudoku übrig bleibt. Das neue Bild wird eine Breite und Höhe von der längsten Seite des Sudokus haben. Hierfür wird die größte Distanz D_{max} zwischen allen benachbarten Punkten in C_R berechnet. OpenCV stellt eine zweidimensionale Bildtransformation bereit, mit welcher alle Pixel innerhalb des aufgespannten Bereichs von C_R auf das neue Bild abgebildet wird. Im neuen Bild hat P_{tl} nun die Koordinaten $(0, 0)$ und P_{br} hat $(D_{max} - 1, D_{max} - 1)$.



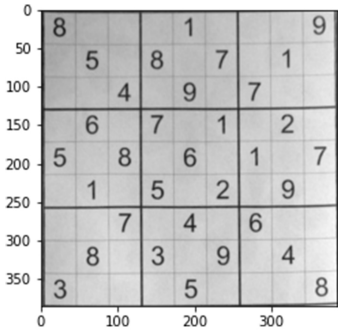
(a) originales Bild



(b) mit Farbanpassung



(c) Ecken vom Sudoku



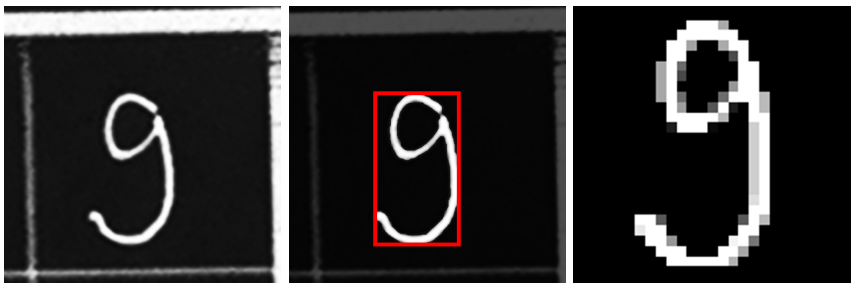
(d) extrahiertes Sudoku

Abb. 1: Die Zwischenergebnisse beim Extrahieren vom Sudokugitter. Vom Eingabebild(1a) werden die Farben entfernt und die Konturen betont(1b). Die Ecken vom größten Rechteck werden berechnet(1c) und mithilfe dieser das Sudoku auf die volle Größe transformiert(1d).

3.2 Zellenextraktion

Nachdem das Sudokugitter aus dem Eingabebild extrahiert ist, sollen die einzelnen Ziffern für die Bildklassifikation aufbereitet werden. Das Ziel ist eine Liste an 81 Einzelbildern der Ziffern im gewünschten Format 28x28 Pixel. Die Herausforderung besteht darin leere Zellen zu erkennen und ansonsten die Ziffer in der Zelle zu isolieren.
Sei die Höhe und Breite des Sudokugitters s Pixel. Es wird über jede Zelle iteriert indem der Iterationschritt k je Achse auf $k = s/9$ Pixel gesetzt wird, welches gleichzeitig die Zellengröße darstellt. Somit wird die relative Position einer Zelle im Gitter durch den Punkt $(i*k, j*k)$ mit $0 \leq i, j < 9$ definiert, welches die obere linke Ecke einer Zelle ist. Die gesamte Zelle kann dann von dem entsprechenden Punkt aus mit k berechnet und aus dem Gitter ausgeschnitten werden. Für die Bildklassifikation können diese jedoch noch nicht verwendet werden. Zuviele Störungen wie in Abbildung 2a gezeigt können sich noch in der Zelle befinden.

Es gilt das größte Feature zu isolieren, welches im Optimalfall die entsprechende Ziffer ist. Hierfür wird eine Funktion von OpenCV benutzt, welches mehrere zusammenhängende Pixel mit einem bestimmten Farbwert flutet. Dabei wird von einem Urspungspixel ausgehend nur die benachbarten Pixel eingefärbt, welche in einer definierten Toleranz dem Wert vom ursprünglichen Pixel entsprechen. Im ersten Durchlauf werden alle weißen Pixel mit dem Wert 64 geflutet, welcher ein Grauton ist. Die Funktion gibt zurück, wieviele Pixel dadurch geflutet wurden. Anhand dessen wird der Urspungspixel als Featurepixel gespeichert, welcher die meisten Pixel geflutet hat. Das Bild der Zelle sollte an diesem Prozessabschnitt nur schwarze und graue Pixel enthalten. Mithilfe des Featurepixels wird das größte Feature mit dem Wert 255, also Weiß geflutet. Das Feature ist nun das einzige Objekt im Bild, welches weiß ist. Daraus lässt sich nun eine Featurebox berechnen, welche die weißen Pixel im Bild genau einrahmen. Das Resultat wird in Abbildung 2b dargestellt.
Um die Extraktion der Zellen abzuschließen, wird das Feature mithilfe der Featurebox ausgeschnitten und auf einen schwarzen Hintergrund zentriert eingefügt. Wenn die Featurebox zu klein ist, dann wird die Zelle als leer gewertet. Zum Schluss wird die Zelle auf das passende Format von 28x28 Pixel skaliert, damit die Bildklassifikation die Zellen verarbeiten kann. Das Endergebnis der Zellenextraktion zeigt Abbildung 2c.



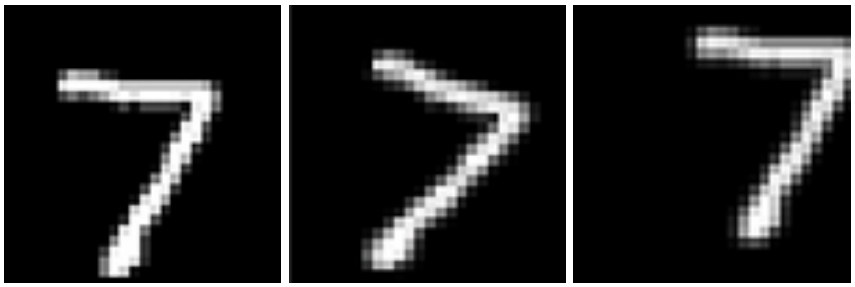
(a) unverarbeitet (b) Featurebox (c) skaliert

Abb. 2: Die Zwischenergebnisse beim Extrahieren der Zellen. In Abbildung 2a sind klare Linien zu sehen, welche für die Klassifikation störend sein können. Eine Featurebox wird berechnet, welche das größte Feature im Bild umschließt(2b). Das isolierte Feature wird ausgeschnitten und auf die passende Größe skaliert(2c).

4 Bildklassifizierung

4.1 Datensatz

Für die Bildklassifizierung von Zahlen bietet sich der bekannte MNIST Datensatz[9] an. Dieser umfasst 60000 Trainings- und 10000 Testbilder an handgeschriebenen Ziffern. Die Bilder haben einen Graukanal und eine Größe von 28x28 Pixeln. Da es beim Sudoku passieren kann, dass die Ziffern schräg oder leicht rotiert eingetragen werden, wurde der Datensatz mit einfacher Datenerweiterung auf insgesamt 300000 Trainings- und 50000 Testbilder vervielfacht. Die Datenerweiterung wurde speziell für den MNIST Datensatz geschrieben[1]. Dabei wird das originale Bild rotiert, vergrößert und versetzt.



(a) original (b) rotiert (c) versetzt

Abb. 3: Datenerweiterung des MNIST Datensatzes am Beispiel einer handgeschriebenen Sieben. Jedes Bild des originalen Datensatzes durchläuft mehrerer solcher Transformation um diesen künstlich zu vergrößern.

4.2 Netzwerkarchitektur

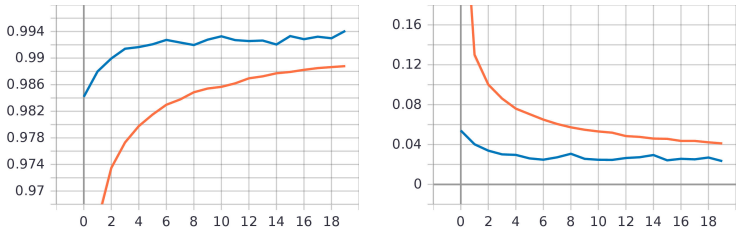
Für die Bildklassifikation wird ein Convolutional Neural Network genutzt. Dieses ist eine Unterspezifikation von neuralen Netzwerken und benutzt Convolutional Layer, welche mit zwei- und dreidimensionale Matrizen umgehen können. Benutzt werden in dem verwendeten Netzwerk zwei solcher Layer, wobei der erste Layer die Dimension 28x28x1 vom Eingabebild besitzt. Die Layer benutzen eine Filtergröße von 5x5 und die ReLU-Aktivierungsfunktion. Nach jedem Convolutional Layer werden zweidimensionale Max-Pooling-Layer eingesetzt. Diese komprimieren eine Eingabematrix, indem sie den maximalen Wert einer Untermengen der Eingabe auf die Ausgabe abbilden. Die Untermenge besteht im beschriebenen Netzwerk aus einem 2x2 Filter. Die beschriebenen Layer haben das Ziel, charakteristische Muster aus den Bilddaten zu ziehen und zu generalisieren. Für die eigentliche Klassifizierung dieser Eigenschaften werden Dense Layer genutzt. Hierfür wird vorerst mithilfe eines Flatten Layers die zweidimensionalen Tensor in eindimensionale Tensor transformiert. Zwei Dense Layer können dann die jeweilige Neuronenanzahl verkleinern, bis die Anzahl der Klassifikationskategorien erreicht wird. Dies liegt im Anwendungsfall bei 10 Ausgabeneuronen, welche jede mögliche Ziffer im Sudoku darstellt. Um Overfitting,

also das genaue Abbilden der Trainingsdaten zu verhindern, werden Dropout Layer genutzt. Dieses dienen als Regulierungsmethode bei neuronalen Netzen, welche einen gewissen Anteil der Neuron nicht einbeziehen. Der Verlust wird mit der Categorical-Crossentropy Funktion berechnet und als Optimierungsfunktion wird Adam genutzt.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_1 (Conv2D)	(None, 8, 8, 64)	51264
max_pooling2d_1 (MaxPooling2	(None, 4, 4, 64)	0
dropout (Dropout)	(None, 4, 4, 64)	0
flatten (Flatten)	(None, 1024)	0
features (Dense)	(None, 128)	131200
dropout_1 (Dropout)	(None, 128)	0
dense (Dense)	(None, 64)	8256
dropout_2 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

Total params: 192,202
Trainable params: 192,202
Non-trainable params: 0

Abb. 4: Die verwendete Netzwerkarchitektur für das MNIST Model. Es werden Convolutional Layer genutzt um aus zweidimensionalen Daten charakteristische Eigenschaften zu ziehen. Für die Klassifizierung werden zwei Dense Layer genutzt mit schlussendlich 10 Ausgabeneuronen. Für das vermeiden von Overfitting werden Dropout Layer verwendet, welche 50% der Neuronen beim Backtracking ignorieren.



(a) Episodengenauigkeit (b) Episodenverlust

Abb. 5: Verlauf der Genauigkeit und des Verlusts über die trainierten 20 Episoden. Der blaue Graph ist jeweils der Validierungsdurchlauf, der orange Graph stellt die Trainingsdurchläufe dar. Die Testgenauigkeit lag bei 0,994 und der Verlust bei 0,023.

4.3 Implementation in Android

Um das trainierte Modell in einer eingebetteten Umgebung verwenden zu können, wird dieses vorerst in ein Tensorflow Lite Model komprimiert. Tensorflow bietet für die Verwendung in Android Anwendungen entsprechende Schnittstellen und Bibliotheken an. Ein Problem welches bestehen bleibt sind die unterschiedlichen Datentypen für Bilder, welche Android und Tensorflow benutzen. Die 81 Bilder der Zellen nach der Bildverarbeitung(3.1) sind bereits im korrekten Farbschema und Größe. Jedoch nur als Android Bitmaps und nicht als ByteBuffer Objekte, welches der unterstütze Datentyp für Tensorflow ist. Hinzu kommt, dass aktuell jeder Pixel in den Zellenbildern die Farbwerte in 8-Bit Integern speichert, welche einen Wertebereich von 0 bis 255 besitzen. Das Modell erwartet jedoch Fließkommawerte zwischen 0 und 1. Diese Transformation wird mit der folgenden Java Methode[8] umgesetzt, dargestell in Abbildung 6.

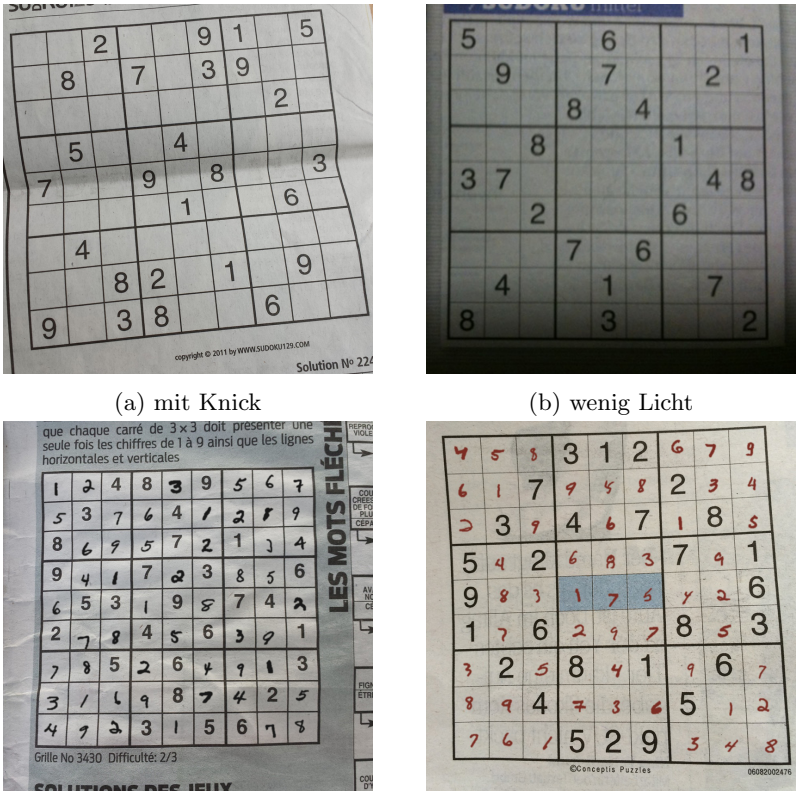
```
1  private ByteBuffer convertBitmapToByteBuffer(Bitmap bitmap) {
2      ByteBuffer byteBuffer = ByteBuffer.allocateDirect(inputSize);
3      byteBuffer.order(ByteOrder.nativeOrder());
4      int[] pixels = new int[imageSize * imageSize];
5      bitmap.getPixels(pixels, 0, bitmap.getWidth(), 0, 0,
           bitmap.getWidth(), bitmap.getHeight());
6      for (int pixel : pixels) {
7          float rChannel = (pixel >> 16) & 0xFF;
8          float gChannel = (pixel >> 8) & 0xFF;
9          float bChannel = (pixel) & 0xFF;
10         float pixelValue = (rChannel + gChannel + bChannel) / 3 / 255.f;
11         byteBuffer.putFloat(pixelValue);
12     }
13     return byteBuffer;
14 }
```

Abb. 6: Methode um Androids *Bitmap* Datentyp in ein $BytheBuffer$ umzuwandeln. Die *inputSize* in Zeile 2 beschreibt die zu schreibenden Bytes, welche sich berechnen lassen aus $inputSize = imageSize^2 * floatSize$, wobei $imageSize = 28$ und $floatSize = 4$. Somit ist das $ByteBuffer$ Objekt 3136 Vytes groß.

5 Auswertung

5.1 Testdaten

Der wesentliche Fokus des Projekts war es nicht, ein Sudoku Rätsel zu lösen. Die leistungstragenden Komponenten waren viel mehr die Bildverarbeitung und Bildklassifikation. Hier war das Ziel, dass die Anwendung möglichst stabil und zuverlässig arbeitet. Also auch mit schwierigen Eingaben umgehen kann.
Um dies zu messen, wurde ein Datensatz[11] an knapp über 100 Bildern herangezogen, welcher möglichst unterschiedliche Qualitäten abdeckt. Sie unterscheiden sich zum einen in der Kamera, mit welchem das Bild aufgenommen wurde. Zum anderen auch von den Lichtverhältnissen, Perspektiven und andere potentiellen Störungen. In einigen Rätseln sind Knicke vorhanden, sowie gedruckte und handschriftliche Zeichen. Der Datensatz soll die Eingaben abdecken, mit welchen die Anwendung umzugehen hat. Einige Beispiele der Testdaten sind in Abbildung 7 gezeigt.



(a) mit Knick (b) wenig Licht

(c) störende Texte (d) verzehrtes Gitter

Abb. 7: Es wurden sowohl Sudokus mit gedruckten Ziffern (7a, 7b), als auch welche mit teilweise handschriftlich ausgefüllten Rätseln getestet (7c, 7d).

5.2 Ergebnisse

Die Ergebnisse der Auswertung des Datensatzes messen sich nicht daran, ob ein Sudoku komplett richtig übernommen wurde. Ein gutes Ergebnis ist erreicht, wenn möglichst viele Zellen korrekt klassifiziert wurden. Die Auswertung bezieht sich also mehr auf die einzelnen Zellen, als auf das Rätselgitter im Ganzen. Es wird demnach auf die korrekte Klassifizierung von mehr als 8100 Zellen bezug genommen.

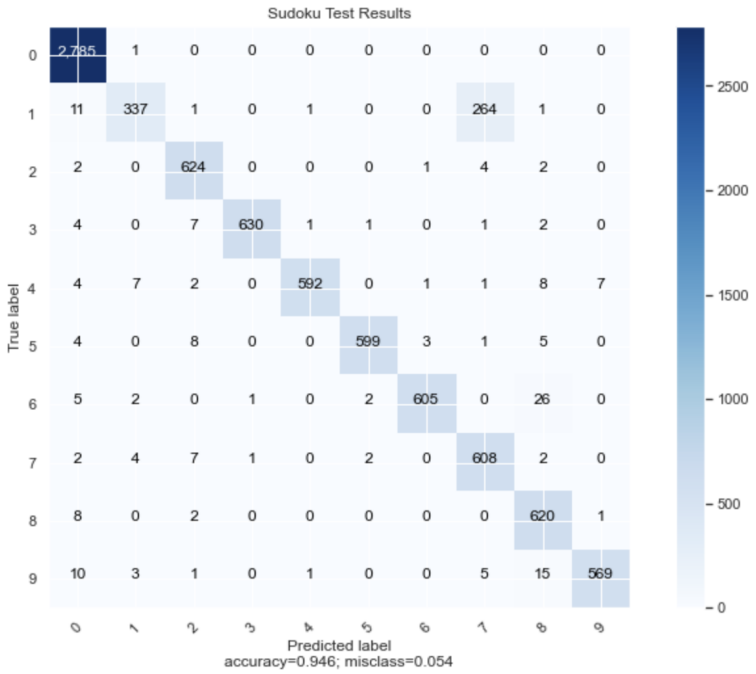


Abb. 8: Die Konfusionsmatrix macht die Genauigkeit von den Ergebnissen des Testdatensatzes deutlich. Die y-Achse beschreibt die tatsächlichen Ziffern einer Zelle, die x-Achse die klassifizierten Ziffern.

Insgesamt liegt die Genauigkeit der Klassifikationen der Zellen innerhalb der ausgewählten Testdaten bei 94,6%. Die Genauigkeit umfasst die Gesamtzahl aller klassifizierten Zellen, also das Zusammenspiel der Komponenten in der Anwendung. Möchte man die Genauigkeit auf die einzelnen Komponenten der Anwendung beziehen, muss der Aufgabenbereich dieser isoliert werden. Die Komponente der Bildverarbeitung sind unter anderem dafür zuständig, die leeren Felder zu erkennen. Betrachtet man also nur die Zeile der Konfusionsmatrix, welche die Ergebnisse für die Ziffer Null darstellen, dann erreicht die Erkennung der leeren Felder eine Genauigkeit von 99,9%. Für die Bildklassifikation stehen alle restlichen Elemente. Hier wurden 92,7% aller nicht leerer Zellen richtig erkannt.

264 mal wurde die Eins als eine Sieben erkannt, welches damit als größte Fehlklassifikation heraussticht. Grund hierfür sind die unterschiedlichen Schreibweisen und die Ähnlichkeit der beiden Ziffern. Sobald eine Sieben ohne horizontalen Strich dargestellt wird, fällt es der Bildklassifikation schwerer zwischen den Ziffern zu unterscheiden. Vor allem wenn die Sieben leicht schräg geschrieben wurde. Andersherum wurden insgesamt nur vier Siebenen als eine Eins klassifiziert.

5.3 Limitierungen

Obwohl über 94% der Zellen korrekt klassifiziert wurden und die Erkennung von leeren Zellen stabil lief, hat die Anwendung seine Grenzen. Die Bildqualität der Kamera muss so scharf sein, dass zum einen alle Zahlen und deren Merkmale erkennbar sind. Zum anderen, dass auch ein zu starkes Pixelrauschen verhindert wird. Dies führte in einigen Fällen dazu, dass zu viele Pixelstörungen als Feature erkannt wurden und somit in unbrauchbare Ergebnisse resultierten. Diese Bilder aus dem verwendeten Datensatz[11] wurden vor dem Test herausgefiltert, da die Bildqualität nicht dem Kamerastandard in aktuellen Smartphones entsprachen. Aus diesem Grund wurden nur die Hälfte der 200 Testbilder für den Test verwendet.
Eine weitere Limitierung liegt in der Erkennung des Sudokugitters. Wenn das Gitter zu stark gekrümmt ist, kann die Bilderkennung dieses nicht mehr korrekt verarbeiten. Der Erkennungsprozess erwartet gerade Linien, wobei leichte Krümmungen ausgeglichen werden können. Stark verzehrte und kurvige Formen können jedoch ab einem gewissen Maß nicht umgewandelt werden.
Innerhalb der eigentlichen Zahlenklassifikation erhöht eine saubere Handschrift die Erfolgsquote. Zuviele Störungen innerhalb einer Zelle führen im schlimmsten Fall zu einer Fehlerkennung. Dies können z.B. kleine Notizen sein, die beim Sudoku häufig gemacht werden, um mögliche Ziffern in der Zelle zu notieren. Dies gleicht die Isolierung der Zellenfeatures in dem Großteil der Fälle jedoch aus, welches im Kapitel 3.2 erläutert wurde. Dies kann jedoch auch an die Grenzen kommen, wenn die Störfaktoren und das gesuchte Feature zusammenhängen.

Optimierungen sind also noch möglich, wobei die Anwendung bereits erfolgreich mit erwarteten Eingaben und einigen Störfaktoren umgehen kann. Die größten Optimierungspunkte liegen wie beschrieben in der Bildverarbeitung, insbesondere der Gittererkennung und in der Unterscheidung von einigen ähnlichen Ziffern bei der Klassifikation.

Literatur

1. Dhayalkar, S.R.: Augmentation techniques for mnist (2017), https://www.kaggle.com/dhayalkarsahilr/easy-image-augmentation-techniques-for-mnist
2. Diaz, J.L.: sudoku-solver (2019), https://github.com/joseluisdiaz/sudoku-solver
3. GeeksforGeeks: Sudoku — backtracking-7 (2021), https://www.geeksforgeeks.org/sudoku-backtracking-7/
4. Jhawar, A.: Sudoku solver using computer vision and deep learning (2019), https://aakashjhawar.medium.com/sudoku-solver-using-opencv-and-dl-part-1-490f08701179
5. Mishra, P.: Sudoku solver (2017), https://github.com/prashantmishra/sudoku-solver
6. Pal, K.K., Sudeep, K.S.: Preprocessing for image classification by convolutional neural networks. In: 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT) (2016). https://doi.org/10.1109/RTEICT.2016.7808140, https://ieeexplore.ieee.org/abstract/document/7808140
7. Smith, D.: So you thought sudoku came from the land of the rising sun ... (2005), https://www.theguardian.com/media/2005/may/15/pressandpublishing.usnews#:~:text=The%20Sudoku%20story%20began%20in,in%20each%20row%20or%20column.
8. Stanek, M.: Tensorflow lite classification on android (2019), https://thinkmobile.dev/mobile-intelligence-tensorflow-lite-classification-on-android/
9. Tensorflow: Mnist, https://www.tensorflow.org/datasets/catalog/mnist
10. Thom, D.: SUDOKU ist NP-vollständig. Ph.D. thesis (01 2007), https://www.researchgate.net/publication/260863995_SUDOKU_ist_NP-vollstandig
11. Wicht, B., Henneberty, J.: Mixed handwritten and printed digit recognition in sudoku with convolutional deep belief network. In: Document Analysis and Recognition (ICDAR), 2015 13th International Conference on. pp. 861–865. IEEE (2015)