Food Classification mit einem Convolutional Neuronal Network

Benjamin Oechsle

27.02.2021

Das Ziel dieser Arbeit ist es, eine möglichst genaue Bildklassifizierung verschiedenster Obst- und Gemüsesorten mit einem Android Smartphone durch ein Convolutional Neural Network zu realisieren.
Dabei wird ein Neuronales Netz von Grund auf mit einem Datensatz aus 62.218 Bildern in 138 Klassen trainiert und das Ergebnis mit dem vortrainierten und im Transferlearningverfahren angepassten Resnet Mobilenet V2 verglichen. Desweiteren wird die Datenqualität und deren Auswirkungen auf das Ergebnis untersucht und abschließend bewertet. Durch Visualisierung der verschiedenen Featuremaps wird außerdem aufgezeigt, dass ein vortrainiertes Netz in der Lage ist, bei dem angewendeten Datenbestand ein besseres Ergebnis zu liefern als das selbst erstellte Modell.

1 Einleitung

Das Verfahren der Bildklassifizierung zielt darauf ab, ein Eingabebild in einer vorher definierte Menge von Klassen zu kategorisieren und gehört zum Bereich des maschinellen Lernens.
Diese Arbeit beschäftigt sich mit der Klassifizierung von Obst und Gemüse durch ein künstliches Neuronales Netzwerk (im folgenden KNN).
Durch die Globalisierung gelangen zunehmend mehr exotische Obst- und Gemüsesorten in die Regale der heimischen Supermarkt-Ketten. Dadurch wird es für den Verbraucher immer schwieriger, die angebotenen Lebensmittel zu kennen und sinnvoll in den eigenen Speiseplan zu integrieren. Diese Anwendung soll dabei helfen dem Nutzer mit der Kamera seines Android Smartphones in Echtzeit die ausgelegten Lebensmittel zu klassifizieren. Dabei ist die App in der Lage sowohl die passende Definition von Wikipedia als auch Rezepte von Chefkoch.de anzeigen zu lassen.

Die Arbeit lässt sich in sechs Kapitel aufteilen, die durch mehrere Unterkapitel gegliedert sind. Zunächst wird in Kapitel 2 in das Thema des maschinellen Lernens und der neu-

ronalen Netzte eingeführt. Kapitel 3 befasst sich mit dem zugrunde liegenden Datensatz und dessen Aufbereitung.
In Kapitel 4 wird auf den Aufbau des zugrunde liegenden neuronalen Netzes eingegangen. Dabei wird eine passende Architektur gewählt und die einzelnen Hyperparameter justiert, um eine möglichst hohe Genauigkeit zu erzielen. Des Weiteren wird das Ergebnis des eigens erstellten KNN mit einem Vortrainierten Modell, wie VGG16 oder MobilenetV2, verglichen. Damit soll geklärt werden, ob auf Basis der genannten Netze bessere Ergebnisse zu erzielen sind, indem lediglich die letzten Layer ersetzt und im Transfer Training angepasst werden. In Kapitel 5 wird die Konvertierung des Tensorflow Models in ein Tensorflow-Lite Modell erörtert, welches dann in einer passenden Android Implementierung genutzt werden kann. Kapitel 6 befasst sich mit der Auswertung der Ergebnisse, bevor abschließend ein Fazit gezogen wird.

2 Maschinelle Lernmethoden

Als Teilbereich der künstlichen Intelligenz gliedert sich maschinelles Lernen in die Bereiche unüberwachtes Lernen, verstärkendes Lernen und überwachtes Lernen.
Unüberwachtes Lernen ist dabei als eine Methode definiert, bei der dem Algorithmus keine Informationen über die zu lernenden Daten vorliegen und somit einzig aus Gemeinsamkeiten, die aus den Daten extrahiert werden, Gruppenzugehörigkeiten gebildet werden können. Ein Vorteil dieser Methode ist, dass die Daten nicht vorab korrekt von einem Menschen klassifiziert werden müssen. Bekannte Arten des unüberwachten Lernens sind Clustering Verfahren oder Autoencoder.
Beim verstärkenden Lernen wird dem Agenten durch Belohnung, welche auch negativ ausfallen kann, beigebracht, eine selbstständige Strategie zu entwickeln, um die konkrete Problemstellung zu lösen. Eine bekannte Variante des verstärkenden Lernens ist der Monte-Carlo-Tree-Search Algorithmus.
Beim überwachten Lernen werden die Daten zusätzlich mit Identifikatoren (im folgenden Labels) ausgestattet, die vorher korrekt mit den Daten verknüpft werden müssen. Anhand dieser Labels kann der Algorithmus mittels einer Loss-Funktion feststellen, ob die Daten korrekt klassifiziert wurden und mit dem mitgelieferten Label übereinstimmen. [2]

2.1 Neuronale Netze

Die künstlichen neuronalen Netze, die für die Durchführung dieser Arbeit genutzt werden, gehören dabei zum Bereich des überwachten Lernens. Als Vorbild für den Aufbau des neuronalen Netzes dient das Nervensystem eines Lebewesens, wobei der Fokus auf der Erkennung von komplexen Zusammenhängen ohne explizites Wissen über das zu lösende Problem liegt. Ihren Ursprung haben Neuronale Netze bereits in den 1940er Jahren

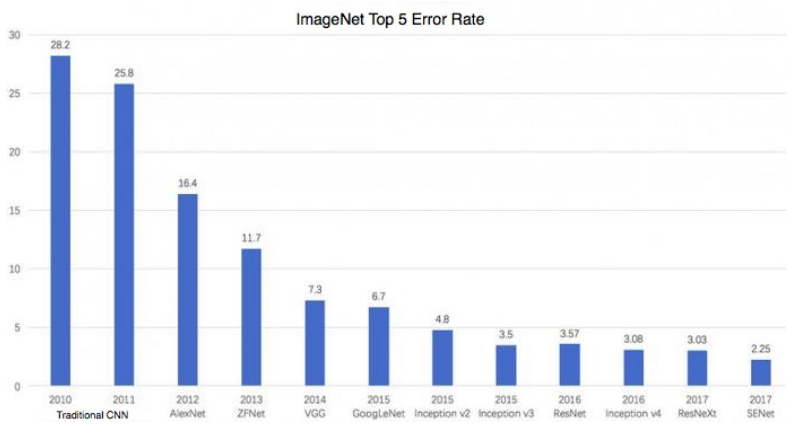


Abbildung 1: Fehlerraten von Neuronalen Netzen von 2010 bis 2017 [15]

des vergangen Jahrhunderts. Seit 2009 jedoch erleben die KNN dank verbesserter Algorithmen und steigender Rechenleistung eine Renaissance. Gerade im Bereich der Bild-, Video- und Audioklassifikation finden sie seitdem verstärkt Anwendung. Wie Abbildung 1 zeigt, sind die Fehlerraten in der Bilderkennung in den Jahren 2011 bis 2017 von 25,8% auf 2,25% signifikant gefallen. Dies hat die Anwendungsgebiete stark erweitert, so dass KNN mittlerweile ein wesentlicher Bestandteil des autonomen Fahrens, PredictedMaintenance, von Frühwarnsystemen, bei Sprach- und Klangsynthese sowie bei medizinischen oder biologischen Problemen geworden sind.[16]

3 Trainings- und Testdaten

Als Aufnahmegerät wird ein Samsung Galaxy S7 eingesetzt, das mittels der Serienbildfunktion unter Android 9 in der Lage ist, multiple Aufnahmen mit nur einer einzigen Betätigung durchzuführen. So werden pro Obst und Gemüse 100 Bilder aus verschiedenen Perspektiven und mit möglichst neutralem Untergrund erstellt. Abbildung 2 zeigt

eine Auswahl der erstellten Trainingsbilder. Diese werden dann in eine nach dem Lebensmittel klassifizierte Ordnerstruktur abgelegt.
Dabei wird bewusst auf einen Validierungsdatensatz verzichtet, da der später zu integrierende Fremddatensatz dies ebenfalls nicht vorsieht. Stattdessen wird im Code der vorhandene Trainingsdatensatz durch den, von der verwendeten Deep Learing Bibliothek Keras bereitgestellten, ImageDataGenerator in zwei disjunkte Datensätze aufgespalten. Hierbei werden 90% der Bilder für den Trainingsdatensatz und 10% der Bilder für den Validierungsdatensatz gewählt.

Abbildung 2: Eigene Aufnahmen mit dem Samsung Galaxy S7

Die erstellten Bilddaten reichen jedoch quantitativ nicht aus und bieten nicht genug Varianz, um ein neuronales Netz von Grund auf zu trainieren. So müssen Alternativen gefunden werden, um den Datenbestand mit unterschiedlichen Obst- und Gemüsesorten zu erweitern.
Dabei muss darauf geachtet werden, dass möglichst viele unterschiedliche Bilder zu einem einzigen Lebensmittel vorhanden sind.
Als Online-Community Plattform für Datenwissenschaftler ist Kaggle.com eine gute Anlaufstelle für unterschiedlichste Datensätze. Auch im Bereich der Lebensmittel bietet die Webseite verschiedene Datensätze zum maschinellen Lernen an, wobei im folgenden zwei genauer betrachtet werden.

3.1 Datensatz Fruits-360

Der Fruits 360 Datensatz wurde von Mihai Oltean erstellt und wird stetig erweitert.
In der aktuellen Version (208.06.13.0) bietet er 46.937 Bilder, die in 35.133 Trainingsbilder und 11.804 Validationsbilder aufgegliedert sind.
Es werden insgesamt 70 verschieden Obstsorten abgebildet. Darunter befinden sich gängige Sorten, wie Apfel, Banane, Aprikose, Avocado, aber auch exotische wie Kaktusfrucht, Schlangenfrucht, Kaki oder die Melonenbirne, wie Abbildung 3 zeigt. Manche Obstsorten

werden dabei in verschiedene Unterkategorien aufgeteilt, so dass am Ende 134 distinkte Klassen zur Verfügung stehen.

(a) Kaktusfrucht (b) Schlangenfrucht (c) Kaki

Abbildung 3: Exotische Früchte in dem Fruits 360 Datensatz

Die Auflösung der Bilder ist mit 100x100 Pixeln angegeben.
Das Obst in diesem Datensatz wurde mit Hilfe einer rotierenden Scheibe vor einem weißen Hintergrund aufgenommen, so dass sich das Neuronale Netz auf die Merkmale der Obstsorte konzentrieren kann und nicht durch komplexe oder wechselnde Hintergründe abgelenkt wird.
Leider beinhaltet dieser Datensatz keine Gemüsesorten, so dass ein zweiter Datensatz als Quelle herangezogen werden muss. [13]

3.2 Datensatz Fruits and Vegetables

Der Datensatz Fruit and Vegetable von Kritik Seth beinhaltet sowohl Obst, als auch Gemüsebilder. Insgesamt umfasst er 4.291 Bilder, die in zehn Obst, und 27 Gemüsesorten unterteilt sind.
Dabei werden pro Klasse 100 Trainingsbilder, 10 Validierungsbilder und 10 Testbilder zur Verfügung gestellt.
Anders als im Datensatz Fruits 360 sind die Bilder nicht speziell für das Training von Neuronalen Netzen erstellt worden, sondern eine Zusammenstellung frei zugänglicher Bilder der Bing Bildersuche. Daher haben die Bilder keine genormte Auflösung und sind in unterschiedlichen, teils fehlerhaften Formaten abgelegt. Abbildung 4 zeigt außerdem, dass unter den Bildern einige mit Text überlagert sind oder nicht das gewünschte Motiv enthalten.

Um eine gute Datenbasis zu gewährleisten, muss der Datensatz händisch überprüft und bereinigt werden.
Erst wenn sich alle Bilder fehlerfrei in einer sinnvollen Ordnerstruktur befinden, kann mit

(a) Apple Logo (b) Clipart Zeichnung (c) Textüberlagertes Bild

Abbildung 4: Unbrauchbare Bilder aus dem Datensatz Fruit and Vegetable

der eigentlichen Bildvorverarbeitung begonnen werden. Leider stellt sich nach Durchsicht des Fruit and Vegetable Datensatzes heraus, dass etliche Bilder doppelt abgelegt sind oder nicht für eine erfolgreiche Erkennung verwendet werden können. Daher bleiben am Ende von 4.291 nur 1.528 Bilder in 20 Klassen übrig. Von den ursprünglich 37 Klassen sind die meisten im Fruits 360 Datensatz bereits vorhanden, so dass nur vier Klassen (Garlic, Cauliflower, Carot, Potato) neu in das Set aufgenommen werden können. [14]

Nach abschließender Bearbeitung und Sortierung der Bilder gliedert sich das Verhältnis wie folgt:

- Trainingsdaten: 62.218 Bilder in 138 Klassen.
- Validierungsdaten: 6.872 Bilder in 138 Klassen.
- Testdaten: 22.841 Bilder in 138 Klassen.

4 Neuronales Netz

Wie in Abbildung 5 dargestellt, besteht ein Neuronales Netz in der Regel aus Blöcken von Convolutional Layer, gefolgt von Pooling Layer. Die Convolutional Layer haben dabei die Aufgabe, anhand verschiedener Filter (Kernel) die Features oder Merkmale in einem Bild zu erkennen und in einer Featuremap zu hinterlegen.
Der Pooling Layer ist eine Art Merkmalkonzentration, indem er mit einem Kernel über die Pixel des Bildes iteriert und anhand der Größe des Kernels die so überprüften Pixel in der Ausgabe zusammenfasst, also die Featuremap verkleinert.
Das Pooling wird dabei angewendet, um die Anzahl der Parameter zu reduzieren und

sowohl ein Overfitting - das Auswendiglernen des Netzes - zu reduzieren, als auch die Rechenzeit des Lernprozesses zu minimieren.
Durch mehrmalige Wiederholung dieser Blöcke ist das Netzwerk in der Lage, ausreichend komplexe Features zu extrahieren, um eingegebene Bilder anhand der gespeicherten Merkmale klassifizieren zu können.
Dabei sind die Neuronen der einzelnen Schichten mit einer non-linearen Aktivierungsfunktion ausgestattet, die bestimmt, ob das Neuron bei einem auftretenden Muster "feuert" und die darauf folgende Schicht mit Werten versorgt. Die Kanten zwischen den Neuronen der verschiedenen Schichten verfügen dabei über Gewichte, deren Änderung in der Backpropagation im Laufe des Lernprozesses als das eigentliche Lernen beschrieben werden kann.

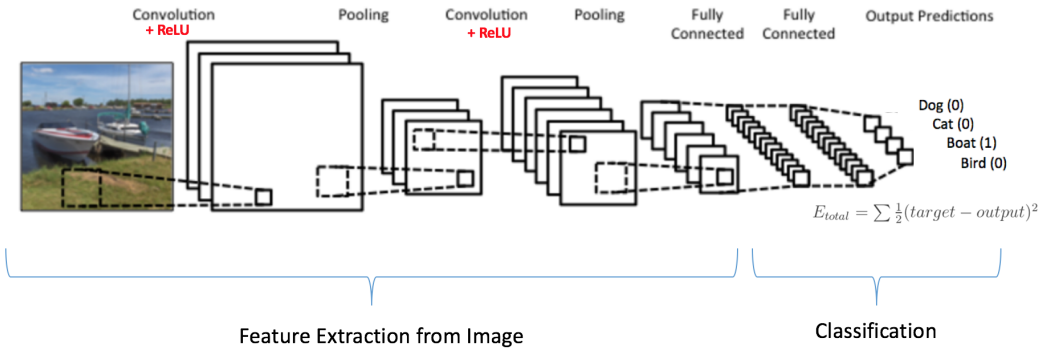


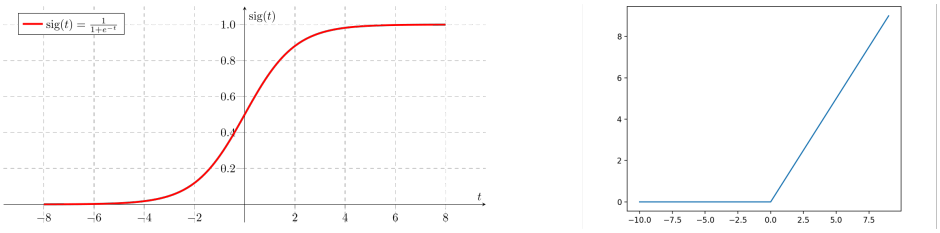
Abbildung 5: Aufbau eines Neuronalen Netzes
[11]

Während die ersten Convolutional Layer des Netzwerks einfache Strukturen, wie Kanten oder Linien in verschiedenen Ausrichtungen erkennen, kombinieren die Filter der tieferen Schichten diese Merkmale zu immer komplexeren Strukturen, die für den Menschen nicht mehr interpretierbar sind.
Nach dem letzten Convolutional Block wird die generierte Featuremap in einem Flatten Layer in einen eindimensionalen Vektor geschrieben, der mit ein oder mehreren vollständig verbundenen Layern (im folgenden Dense Layern) verknüft ist. Der letzte Dense Layer hat exakt so viele Neuronen, wie das Netzwerk an verschiedenen Klassen unterscheiden muss und gibt die Wahrscheinlichkeiten an den Output Neuronen mit einem Wert zwischen 0 und 1 aus.
Diese Wahrscheinlichkeiten werden mit den korreken Labels verglichen und die Abweichung durch eine Loss-Funktion bestimmt. Darauf folgend wird mittels Back Propagation das Netzwerk rückwärts durchlaufen, und anhand der gewählten Optimizer-Funktion die

Gewichte der einzelnen Kanten so angepasst, dass der Loss möglichst minimiert und die Genauigkeit maximiert wird.

4.1 Eigenes Modell

Die Überlegung ist, ein eigenes Netz zu entwickeln und durch Änderung der Hyperparameter eine möglichst hohe Genauigkeit, sowohl auf den Trainings-, als auch auf den Validierungsdaten zu erreichen.
Um später einen genaueren Vergleich ziehen zu können, wird dieselbe Eingabegröße der Bilder, wie sie vom vortrainierten MobilenetV2 akzeptiert wird (224x224x3), gewählt. Dabei steht 224 für die Anzahl der Pixel in horizontaler und vertikaler Ausrichtung. Der letzte Parameter gibt die Anzahl an Farbkanälen (Rot, Grün und Blau) an. Auf Grund dieser Eingabe ist der Inputvektor 224x224x3 = 150.528 groß.
Desweiteren wird eine Kernelgröße von 5,5 und als Aktivierungsfunktion ReLu (rectified linear activation function) gewählt, die sich in den letzten Jahren als sehr zuverlässig herausgestellt hat, [1] da sie das Problem des Vanishing Gradient gegenüber der Sigmoid Funktion $S(x) = \frac{1}{1+e^{-x}}$ verkleinert. Außerdem hat die ReLu Funktion, wie in Abbildung 6 dargestellt ist, durch die vereinfachte Berechnung $max(0, x)$ einen Performancevorteil gegenüber der Sigmoid Funktion.



(a) Sigmoid Funktion [18] (b) ReLu Funktion [4]

Abbildung 6: Sigmoid und ReLu Aktivierungsfunktionen

Gefolgt wird der Convolutional Layer von einem Max-Pooling mit einer Kernelgröße von 2x2 und einem Stride von 2, durch den die Featuremap um die Hälfte verkleinert wird und sich somit die Anzahl der Parameter halbiert. Dieser Aufbau wiederholt sich weitere drei mal, so dass insgesamt vier Convolutional Layer, jeweils gefolgt von Max Pooling Layern vorliegen, bevor das Ergebnis von 13x13x128 in einen eindimensionalen Vektor mit einer Ausgabegröße von 21.632 geschrieben wird. Dieser wird mit einem Dense Layer mit 128 Neuronen verbunden, der wiederum voll mit dem Output Layer mit seinen 138

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 224, 224, 64)	4864
max_pooling2d (MaxPooling2D)	(None, 112, 112, 64)	0
conv2d_1 (Conv2D)	(None, 110, 110, 64)	36928
max_pooling2d_1 (MaxPooling2	(None, 55, 55, 64)	0
conv2d_2 (Conv2D)	(None, 54, 54, 128)	32896
max_pooling2d_2 (MaxPooling2	(None, 27, 27, 128)	0
conv2d_3 (Conv2D)	(None, 27, 27, 128)	65664
max_pooling2d_3 (MaxPooling2	(None, 13, 13, 128)	0
dropout (Dropout)	(None, 13, 13, 128)	0
flatten (Flatten)	(None, 21632)	0
features (Dense)	(None, 128)	2769024
dense (Dense)	(None, 138)	17802

Total params: 2,927,178
Trainable params: 2,927,178
Non-trainable params: 0

Abbildung 7: Struktureller Aufbau des ersten KNN Modells

Neuronen verbunden ist. Wie in Abbildung 7 zu sehen ist erreicht das Modell somit eine Anzahl von 2.927.178 trainierbaren Parametern im gesamten Netz.
Mit dieser Anordnung und dem Adam Optimizer mit einer Learning Rate von 0.001 wird nach 8 Epochs eine Accuracy von 97,84% und eine validation accuracy von 95,58 erreicht, wie Abbildung 8 zeigt. Beim Testen des Modells, sowohl mit Testdaten als auch in der

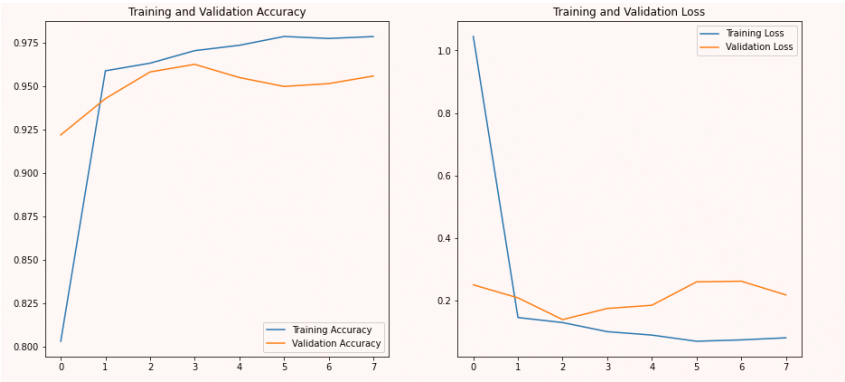


Abbildung 8: 97,84% Accuracy 95,58% Validation Accuracy

Android App, stellt sich jedoch eine signifikante Abweichung der Genauigkeit heraus, die wahrscheinlich auf eine schlechte Generalisierung des Netzes hindeutet. Um die Werte zu verbessern, wird das vorhandene Modell angepasst. Dazu wird, wie Abbildung 9 zeigt, nach dem flatten Layer ein weiterer Dense Layer mit 256 Neuronen hinzugefügt. Des Weiteren wird ein Dropout Layer hinzugefügt, der pro Epoch zufällige Neuronen stilllegt,

um dem Overfitting entgegenzuwirken. Außerdem wird die Learning Rate des Adam Optimizers auf 0,0001 korrigiert. Wie Abbildung 10 zeigt erreicht das Modell mit diesen

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 224, 224, 64)	4864
max_pooling2d_4 (MaxPooling2	(None, 112, 112, 64)	0
conv2d_5 (Conv2D)	(None, 110, 110, 64)	36928
max_pooling2d_5 (MaxPooling2	(None, 55, 55, 64)	0
conv2d_6 (Conv2D)	(None, 54, 54, 128)	32896
max_pooling2d_6 (MaxPooling2	(None, 27, 27, 128)	0
conv2d_7 (Conv2D)	(None, 27, 27, 128)	65664
max_pooling2d_7 (MaxPooling2	(None, 13, 13, 128)	0
dropout_1 (Dropout)	(None, 13, 13, 128)	0
flatten_1 (Flatten)	(None, 21632)	0
features (Dense)	(None, 256)	5538048
dense_2 (Dense)	(None, 128)	32896
dense_3 (Dense)	(None, 138)	17802

Total params: 5,729,098
Trainable params: 5,729,098
Non-trainable params: 0

Abbildung 9: Optimiertes Modell des KNN

Einstellungen eine Genauigkeit von 98,67% und eine validation Accuracy von 96,25%. Im Test wird nun eine Genauigkeit von über 90% erreicht, was eine Verbesserung darstellt, jedoch nicht optimal ist.

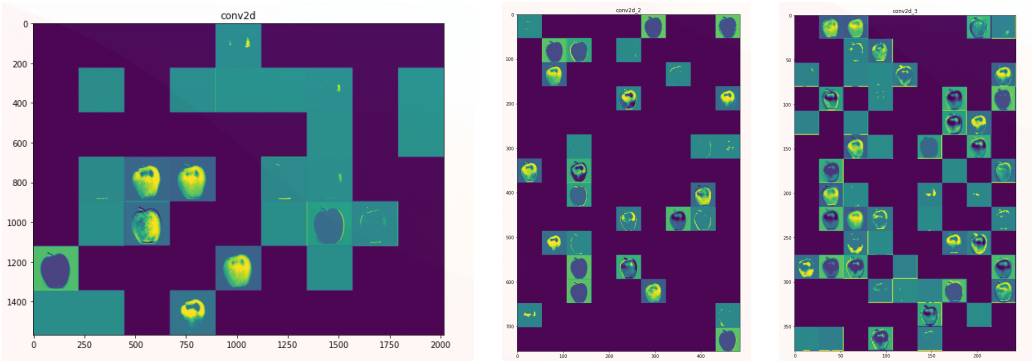
4.1.1 Visualisierung

Um besser zu verstehen, was das neuronale Netz "sieht", werden die einzelnen Schichten sichtbar gemacht und ausgegeben. Abbildung 11 zeigt die Featuremaps der ersten drei Schichten des KNN. Dazu wird ein Skript von towardsdatascience.com [5] genutzt, welches durch die Layer iteriert (Zeile 3) und jedes Feature in eine Matrix schreibt (Zeile 12 - 24), um diese dann zu rendern (Zeile 28 - 30). Es wird hierbei sichtbar, wie die Features, die von dem Netzwerk extrahiert werden, abstrakter werden, je weiter in das Netzwerk abgestiegen wird.

```
1 IMAGES_PER_ROW = 9
2 # Displays the feature maps
3 for layer_name, layer_activation in zip(layer_names, activations):
4     # Number of features in the feature map
5     n_features = layer_activation.shape[-1]
6     #The feature map has shape (1, size, size, n_features).
7     size = layer_activation.shape[1]
```

Abbildung 10: Abschließende Genauigkeit des eigenen KNN

```
    # Tiles the activation channels in this matrix
    n_cols = n_features // IMAGES_PER_ROW
    display_grid = np.zeros((size * n_cols, IMAGES_PER_ROW * size))
    # Tiles each filter into a big horizontal grid
    for col in range(n_cols):
        for row in range(IMAGES_PER_ROW):
            channel_image = layer_activation[0,
                                             :, :,
                                             col * IMAGES_PER_ROW + row]
            # Post-processes the feature to make it visually palatabl
            channel_image -= channel_image.mean() e
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            # Displays the grid
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                        scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')
```



(a) Erster Layer (b) Zweiter Layer (c) Dritter Layer

Abbildung 11: Visualisierung der Featuremaps des KNN

Ein anderer Weg, das KNN zu visualisieren, ist ein graues Bild mit statischem Rauschen in das trainierte Netzwerk zu geben. Statt jedoch mit stochastic gradient descent den Losswert zu minimieren, wird in diesem Fall versucht, mit stochastic gradient accent diesen zu maximieren, um bei der Back Propagation nicht die Gewichte an den Kanten zu verändern, sondern das Input Bild zu formen. Abbildung 12 zeigt dabei die ersten vier Schichten des so erstellten Bildes [6].

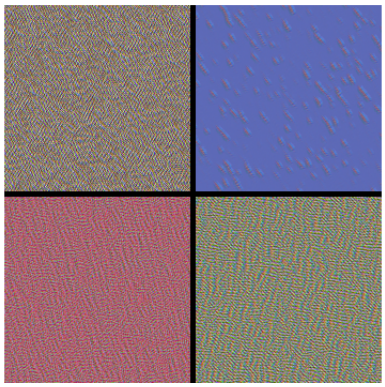
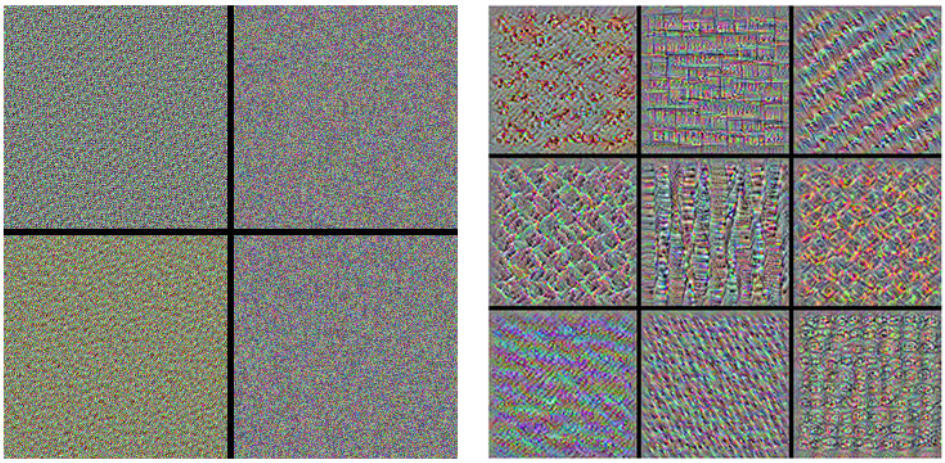


Abbildung 12: Visualisierung der ersten Schicht eines KNN

Wie gut zu sehen ist, werden in der ersten Schicht simple Muster, wie horizontale, vertikale oder diagonale Kanten erkannt. In den Schichten darunter sind jedoch, anders als erwartet, keine komplexeren Strukturen auszumachen, sondern das Rauschen überlagert das Bild. Dies liegt vermutlich daran, dass das Modell mit vergleichsweise wenigen Bildern angelernt wurde und sich deshalb die komplexen Features noch nicht so stark

herauskristallisiert haben. Vergleicht man die unteren Layer mit denen in einem vortrainierten Modell, wie dem MobilenetV2, lässt sich, wie in Abbildung 13 zu sehen ist, ein deutlicher Unterschied erkennen.



(a) Eigenes KNN (b) MobilenetV2

Abbildung 13: Vergleich der tiefen Layer des KNN

4.2 Vortrainierte Netze

Die Präzision von Faltungsnetzen ist in der Regel abhängig von der Anzahl der zu trainierenden Parameter.
Da komplexe Netze bei der Entwicklung jedoch ein hohes Maß an Erfahrung, einen großen Rechenaufwand und eine gewaltige Datenbasis benötigen, wird häufig Transfer Learning eingesetzt.
Transfer Learning ist eine Art des maschinellen Lernens, in der auf ein bereits vortrainiertes Netz aufgebaut wird und die vorhandene Struktur mit kleinen bis mittelgroßen Datensätzen auf die spezifischen Anforderungen eingestellt wird.
Dabei werden die unteren Schichten des bestehenden Netzes unverändert übernommen, was den Rechenaufwand erheblich verkürzt. Lediglich die Ausgabeschicht, und bei Bedarf nächst höhere Schichten, werden auf die individuellen Bedürfnisse angepasst. Das so optimierte KNN liefert dann in vielen Fällen eine höhere Präzision als individuell erstellte Netze.

4.2.1 VGG16

Nachdem das eigene Modell trainiert und optimiert wurde, werden die Ergebnisse nun mit einem etablierten Modell der Bilderkennung verglichen.
Zur Auswahl in Tensorflow stehen dabei eine Reihe an vortrainierten Netzen, die sich grob in zwei Kategorien einteilen lassen.
Die erste Kategorie beinhaltet große komplexe Netze, die im Hinblick auf Präzision entwickelt wurden.
Ein bekannter Vertreter ist das VGG16 Modell, welches von K. Simonyan und A. Zisserman an der University of Oxford in dem Paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" [3] veröffentlicht wurde. 2014 gewann das VGG16 den Imagenet Challenges [9] und war infolgedessen und aufgrund der hohen Generalisierung des Modells eine bevorzugte Wahl für viele Anwendungsgebiete.
Wie in Tabelle 1 dargestellt ist, besitzt das VGG 16 16 Convolutional Layer die in zwei Gruppen angeordnet sind. Die erste Gruppe besteht aus zwei mal zwei Convolutional Layer, jeweils gefolgt von einem Pooling Layer. Die zweite Gruppe besteht aus drei mal drei Convolutional Layer, gefolgt von jeweils einem Pooling Layer. Anschließend sind drei Dense Layer verbunden, die in eine Ausgabeschicht mit 1000 Ausgabeneuronen münden.

Nummer	**Convolution**	**Ausgabe Dimension**	**Pooling**	**Ausgabe Dimension**
Layer 1 & 2	convolution layer 64, kernel=3x3, padding=1, stride=1	224x224x64	Max Pooling, stride=2 size=2x2	112x112x64
Layer 3 & 4	convolution layer 128, kernel=3x3	112x112x128	Max Pooling, stride=2 size=2x2	56x56x128
Layer 5, 6, 7	convolution layer 256, kernel=3x3	56x56x256	Max Pooling, stride=2 size=2x2	28x28x256
Layer 8, 9, 10	convolution layer 512, kernel=3x3	28x28x512	Max Pooling, stride=2 size=2x2	14x14x512
Layer 11, 12, 13	convolution layer 512, kernel=3x3	14x14x512	Max Pooling, stride=2 size=2x2	7x7x512

Tabelle 1: Architektur des VGG16
[3]

4.2.2 MobilenetV2

Die zweite Kategorie der vortrainierten Netze sind die Resnets (Deep Residual Networks). Diese Netzwerke konzentrieren sich darauf, eine hohe Genauigkeit durch viele aufeinander folgende Schichten zu erzielen. Dabei muss bedacht werden, dass es nicht möglich ist, einfach immer mehr Convolutional Layers zu stacken, um immer bessere Genauigkeiten zu erhalten. Die Genauigkeit nimmt nach einer bestimmten Menge sogar wieder ab, wie in Abbildung 14 dargestellt ist, was an dem Problem des Vanishing- oder Exploding-Gradients liegt.[17]
Dies bedeutet, dass bei der Backpropagation die Gewichte in tiefen Netzen sehr lange sequentielle Ketten der Multiplikation durchlaufen müssen, was dazu führen kann, dass der Gradient so klein wird, dass sich die Änderung an den Gewichten relativiert und kein weiterer Lerneffekt auftritt.

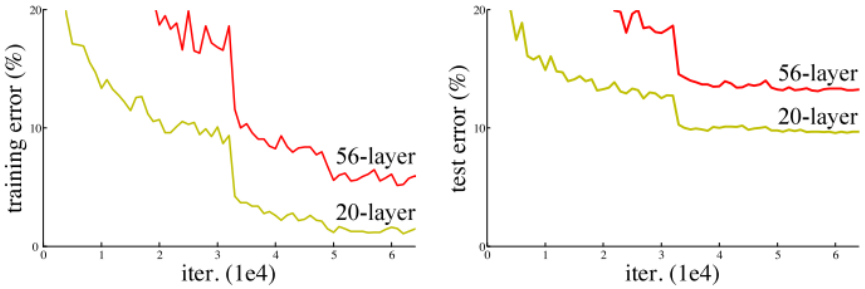


Abbildung 14: Abnahme der Genauigkeit trotz Erhöhung der Layertiefe [10]

Dieses Problem wurde 2015 von Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun in der Studie “Deep Residual Learning for Image Recognition” untersucht und mit so genannten Shortcuts gelöst.
Abbildung 15 zeigt einen Restblock, bei dem vereinfacht dargestellt, einzelne Layer im KNN übersprungen werden können, und die so direkt verbundenen Ebenen zusammengefasst werden [10].

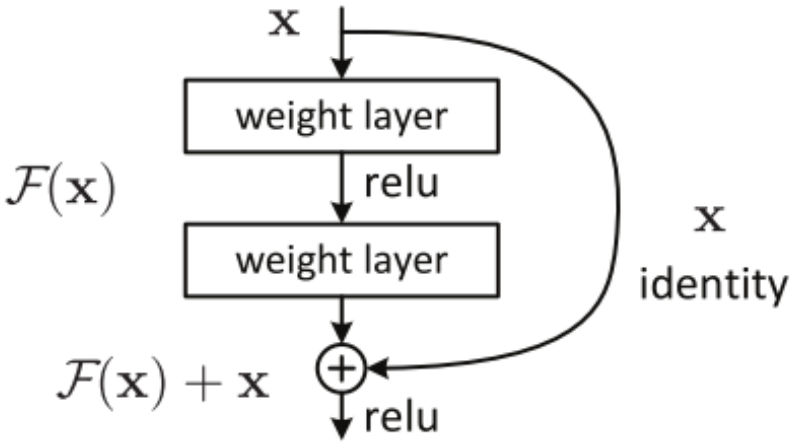


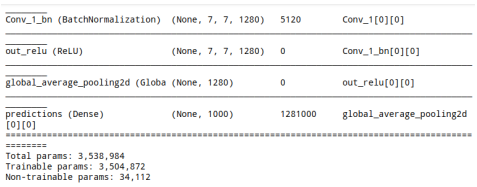
Abbildung 15: Ein Restblock
[10]

Durch diesen Shortcut ist es möglich, die Layertiefe signifikant zu erhöhen, ohne Gefahr zu laufen, im Training auf das Vanishing- oder Exploding-Gradiant Problem zu stoßen. Das MobilenetV2 basiert auf der Theorie der Resnets und beinhaltet 88 Layer mit 3.538.984 trainierbaren Parametern (Vergleich VGG16 23 Layer mit 138.357.544 Parametern). Es wurde mit dem Imagenet Competition Datensatz von 2012 trainiert, welcher 1.280.000 Trainings-, 50.000 Validierungs- und 100.000 Testbilder und 1.000 Klassen umfasst. Dabei kommt das MobilenetV2 auf eine Top-5 Accuracy von 90,1% (Vergleich VGG16 90,1%) und eine Top-1 Accuracy von 71,3% (Vergleich VGG16 71,3%). Der große Vorteil am MobilenetV2 gegenüber dem VGG16 ist jedoch seine Größe von 14 MB (Vergleich VGG16 528 MB) [12], welche es für den Einsatz auf Embedded Systems oder Mobilgeräten, wie sie in diesem Projekt eingesetzt werden, deutlich interessanter macht.

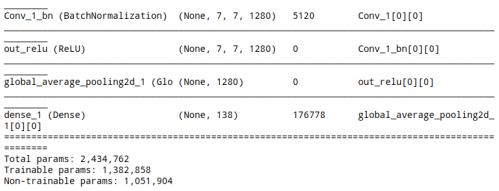
4.2.3 Transfer Learning

Nachdem die Wahl also auf das MobilenetV2 gefallen ist, muss in einem ersten Schritt die Modellstruktur geändert werden. Dazu wird der letzte Layer entfernt und ein neuer Output- Dense Layer erstellt, der mit 138 Neuronen die benötigte Anzahl an Klassen repräsentiert. Danach werden die ersten 23 Layer auf trainable=False gesetzt, um die Generalisierung, die durch die bereits trainierten Daten vorhanden ist, nicht zu verlieren.

Außerdem reduziert das Überspringen der ersten 23 Layer die zu trainierenden Parameter von 3.504.872 auf 1.382.858, wie in Abbildung 16 zu erkennen ist, was die Trainingszeit deutlich reduziert.



(a) MobilenetV2 im Ursprungszustand



(b) MobilenetV2 angepasst auf 138 Klassen

Abbildung 16: MobilenetV2 vor und nach der Anpassung

Auch bei diesem Modell findet der Adam Optimizer mit einer Learning Rate von 0,0001 Verwendung. Bereits nach zwei Epochen nähert sich der Wert 99% an und nach den gesamten 10 Epochen hat das Modell, wie in Abbildung 17 dargestellt wird, eine Accuracy von 99,99% und eine Validation Accuracy von 99,55% erreicht.

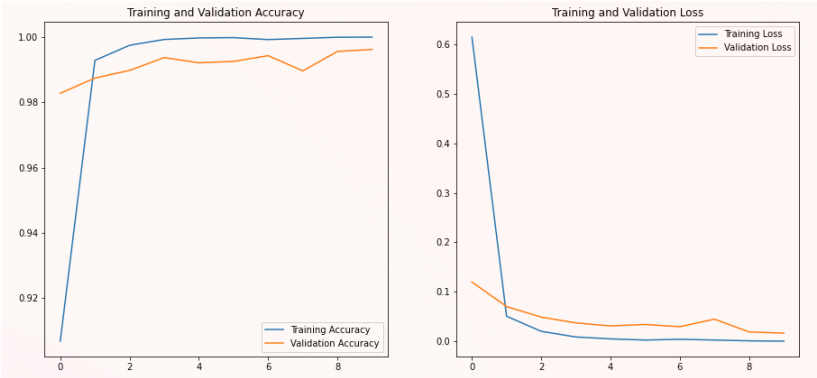
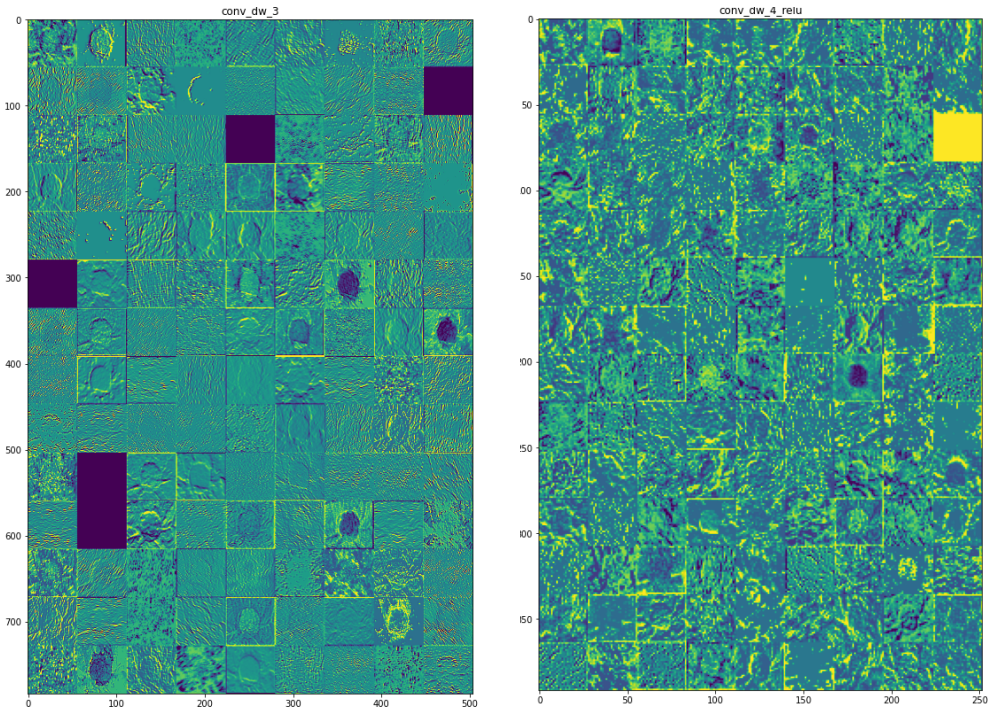


Abbildung 17: Accuracy 99,99% und Validation Accuracy von 99,55%

Auch nach einem Testdurchlauf bestätigt sich eine sehr hohe Genauigkeit des Netzwerks von 96,89%, so dass dieses Modell in der Android App eingesetzt werden kann. In Abbildung 18 kann man erkennen, dass die tieferen Featuremaps (Layer 3 und Layer 4) des MobilenetV2 deutlich detailreicher ausfallen, als sie das in unserem selbst erstellten Netz tun (siehe 5.1.1 Visualisierung).



(a) Layer 3 (b) Layer 4

Abbildung 18: Visualisierung der tiefen Layer vom MobilenetV2

5 Implementierung

5.1 Konvertierung in Tensorflow lite

Bevor das Modell in einer Android Umgebung genutzt werden kann, muss das Tensorflow Modell in ein Tensorflow-Lite Modell konvertiert werden. Dazu wird der Code in Abbildung 19 verwendet.

```
GpuDelegate delegate = new GpuDelegate();
Interpreter.Options options = (new Interpreter.Options()).addDelegate(delegate);
Interpreter interpreter = new Interpreter(tensorflow_lite_model_file, options);
try {
  interpreter.run(input, output);
}
```

Abbildung 19: Code für das Convertieren in ein tf-lite Modell

Das Tensorflow-Lite Format ist speziell für Embeded Systems oder Mobilanwendungen unter IOS und Android konzipiert, welchen keine leistungsstarken CPUs oder GPUs zur Verfügung stehen. Bei dem Konvertierungsprozess wird das Tensorflow Modell dabei Optimierungen in Bezug auf die Inference Geschwindigkeit und den Speicherverbrauch unterzogen, ohne dabei an Genauigkeit zu verlieren.

5.2 Applikation

Die Application setzt auf das Tensorflow Beispielprojekt [7] für Image Classification von Google auf. Dabei muss der vorhandene Code angepasst werden, um ein passendes Inputformat zu generieren.

```
1    int imageTensorIndex = 0;
2    DataType imageDataType = tflite.getInputTensor(imageTensorIndex).dataType
     ();
3    int probabilityTensorIndex = 0;
4    int[] probabilityShape =
5        tflite.getOutputTensor(probabilityTensorIndex).shape();
6    DataType probabilityDataType = tflite.getOutputTensor(
     probabilityTensorIndex).dataType();
7
8    // Creates the input tensor.
9    inputImageBuffer = new TensorImage(imageDataType);
10
```

```
11      // Creates the output tensor and its processor.
12      outputProbabilityBuffer = TensorBuffer.createFixedSize(probabilityShape,
        probabilityDataType);
```

Das Tensorflow-Lite Modell, welches dem Beispiel mitgeliefert wird, ist mit dem Tensorflow-Lite Modelmaker[8] von Google erstellt worden, einem Framework, dass das Erstellen von Neuronalen Netzen stark vereinfacht und ihnen automatisch weitere Input Layer und Image Pre-Processing Schritte voran stellt.
Des weiteren werden von dem Tool Metadaten erzeugt, welche der Android Entwicklungsumgebung dabei helfen, das Input- und Ouputformat zu vereinfachen. Da das selbst erstellte KNN nicht durch den Modelmaker erzeugt wurde, muss das Input- und Outputformat im Code modifiziert werden. Außerdem wird die Funktionalität der Definitionsanzeige und der Rezeptanzeige implementiert.
Dazu werden zwei neue Klassen erstellt, die das Handling der Buttons verwalten. Bei Betätigung wird der Name des erkannten Objekts an die Wikipedia API übergeben (Zeile 5) und geprüft, ob ein passender Artikel vorhanden ist. (Zeile 21)
Wenn die Prüfung positiv ausfällt, wird der Benutzer zum angefragten Artikel weitergeleitet. (Zeile 22)

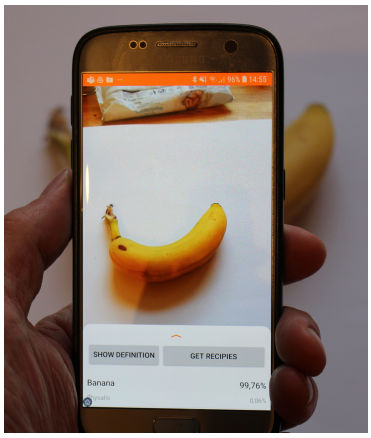
```
1       @Override
2       protected String  doInBackground(String... params)
3       {
4
5           String searchURL = "https://en.wikipedia.org/w/api.php?action=
        opensearch&format=json&search=" + params[0];
6           URLConnection urlConn = null;
7           BufferedReader bufferedReader = null;
8           try
9           {
10              URL url = new URL(searchURL);
11              urlConn = url.openConnection();
12              bufferedReader = new BufferedReader(new InputStreamReader(urlConn
        .getInputStream()));
13
14              StringBuffer stringBuffer = new StringBuffer();
15              String line;
16              while ((line = bufferedReader.readLine()) != null)
17              {
18                  stringBuffer.append(line);
19              }
```

```
20              JSONArray object =  new JSONArray(stringBuffer.toString());
21               if (object.getJSONArray(3).length() >= 1 ) {
22                   return object.getJSONArray(3).getString(0);
23               } else {
24                   return "Object not found";
25               }
26          }
```

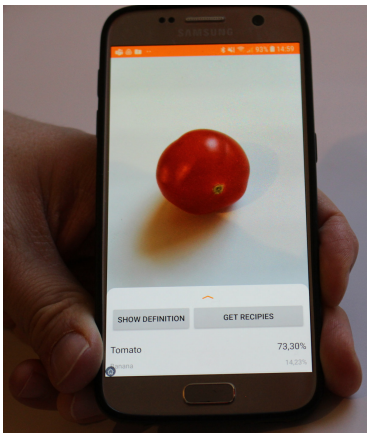
Da Chefkoch.de über keine öffentlich einsehbare API verfügt, wird hier einfach der Name des Objekts gesucht und der Nutzer auf die Ergebnisseite weitergeleitet.

6 Auswertung

Nach der erfolgreichen Implementierung des Models in die Android App und ausgiebigem Testen kristallisierte sich ein essenzielles Problem heraus. Die Bildklassifizierung der App fällt im Praxistest deutlich schlechter (schätzungsweise 20% richtig klassifizierte Bilder) aus, als es die Testdaten mit über 96% Genauigkeit vermuten lassen. Nach einiger Recherche konnte festgestellt werden, dass lediglich Obst und Gemüse, welches vor einem neutralen weißen Hintergrund, wie in Abbildung 20 dargestellt, gefilmt wird, richtig klassifiziert werden kann. Sobald der Hintergrund, wie in Abbildung 21, Variationen aufweist, wird das Obst und Gemüse, ohne erkennbare Anhaltspunkte, als Birne klassifiziert. Dadurch fällt der Fokus auf den für das Training verwendeten Datensatz Fruits-360.
Da die Trainingsbilder im Datensatz alle vor einem neutralen weißen Hintergrund mit gleichmäßiger Ausleuchtung aufgenommen wurden, und der Testdatensatz in dem gleichen Setting erstellt wurde, erklären sich die guten Ergebnisse des Testdurchlaufs und die schlechte Generalisierung des Modells auf Alltagssituationen. Um ein besseres Ergebnis und eine höhere Generalisierung zu erreichen, werden zwingend Bilder aus Alltagssituationen benötigt, die variable Blickwinkel, Hintergründe und Beleuchtungssituationen beinhalten. Dann erst ist das KNN in der Lage, die relevanten, spezifischen Merkmale der einzelnen Obst- und Gemüsesorten richtig zu lernen, zuzuordnen und nicht relevante Informationen, wie den Hintergrund, zu ignorieren. Leider ist es trotz ausgiebiger Recherche nicht möglich, einen weiteren brauchbaren Datensatz zu beschaffen, der die benötigten Bilddaten enthält, um die Klassifizierung des Modells signifikant zu verbessern.

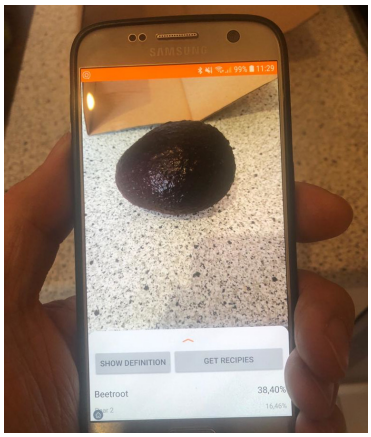


(a) Banane (99,76%)

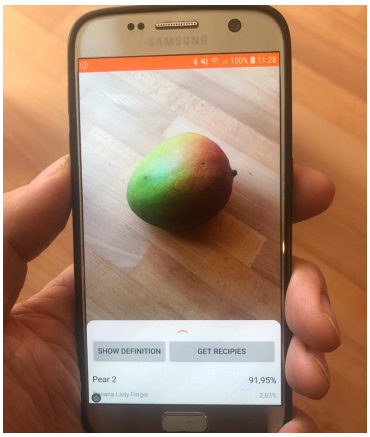


(b) Tomate (73,30%)

Abbildung 20: Richtig klassifiziertes Obst



(a) Rote Beete statt Avocado (38,40%)



(b) Birne statt Mango (91,95%)

Abbildung 21: Falsch klassifiziertes Obst

7 Fazit

Diese Arbeit hat sich mit dem Erkennen und Klassifizieren von Obst und Gemüsesorten durch ein künstliches neuronales Netz beschäftigt, sowie der anschließenden Implementierung des angelernten Modells in eine Android Applikation.
Dabei hat sich gezeigt, dass die zugrunde liegende Datenbasis das Fundament des gesamten Projektes ist. Trotz intensiver Vorverarbeitung und Begutachtung, sowie Sortierung der Daten, war es am Ende der Wahl eines falschen Datensatzes geschuldet, dass die Genauigkeit in der Anwendung massiv gemindert wurde. Dabei spielte die falsche Annahme, dass für ein erfolgreiches Training eines KNN Bilder mit neutralem, einfarbigem Hintergrund besser geeignet sind, als solche, bei denen der Hintergrund wechselt, eine zentrale Rolle. Um eine Klassifizierung der unterschiedlichen Obst- und Gemüsesorten zuverlässig in Alltagssituationen durchführen zu können, bedarf es deutlich mehr und vor allem diverserer Bilddaten, als es die vorhandenen Datensätze, die frei im Netz verfügbar sind, bieten. Des Weiteren hat die schnelle Entwicklung der Tensorflow Umgebung und die vielen Abhängigkeiten zu anderen Frameworks, gerade in Verbindung mit Android, zu erheblichen Problemen geführt. Durch die Verwendung von vorkonfigurierten und getesteten Docker Images kann diese potentielle Fehlerquelle in zukünftigen Projekten minimiert und die Fehlersuche auf den eigenen Code eingeschränkt werden.

Literatur

[1] Alex Krizhevsky, Geoffrey E. H.: ImageNet Classification with Deep Convolutional Neural Networks. (2016). – URL http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf. – Zugriffsdatum: 2020-12-28

[2] Alpaydin, Ethem: *Maschinelles Lernen*. De Gruyter Oldenbourg, 2019. – URL https://doi.org/10.1515/9783110617894. – Zugriffsdatum: 2020-12-28. – ISBN 9783110617894

[3] A.Zisserman, K. S. und: VGG16 Netz. (2014). – URL https://arxiv.org/pdf/1409.1556.pdf. – Zugriffsdatum: 2021-01-06

[4] Brownlee, Jason: *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. 2020. – URL https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks. – Zugriffsdatum: 2021-01-06

[5] Chakravarthy, Arnav: *Visualizing Intermediate Activations of a CNN trained on the MNIST Dataset*. 2019. – URL https://towardsdatascience.com/visualizing-intermediate-activations-of-a-cnn-trained-on-the-mnist-dataset-2c34426416c8. – Zugriffsdatum: 2020-12-28

[6] Chollet, Francois: *How convolutional neural networks see the world*. 2016. – URL https://blog.keras.io/how-convolutional-neural-networks-see-the-world.html. – Zugriffsdatum: 2020-12-28

[7] Google: *Tensorflow Example*. 2020. – URL https://www.tensorflow.org/lite/examples. – Zugriffsdatum: 2021-01-03

[8] Google: *Tensorflow Modelmaker*. 2020. – URL https://github.com/tensorflow/examples/tree/master/tensorflow_examples/lite/model_maker. – Zugriffsdatum: 2021-01-03

[9] imagenet: *Imagenet Challenge*. 2020. – URL http://image-net.org/challenges/LSVRC/2016/index. – Zugriffsdatum: 2021-01-03

[10] Kaiming He, Shaoqing R.: Deep Residual Learning for Image Recognition. (2016). – URL https://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf. – Zugriffsdatum: 2020-12-28

[11] Karn, Ujjwal: *An Intuitive Explanation of Convolutional Neural Networks*. 2016. – URL https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. – Zugriffsdatum: 2021-01-06

[12] Keras: *Keras Applications*. 2020. – URL https://keras.io/api/applications/. – Zugriffsdatum: 2020-12-28

[13] Oltean, Mihai: *Fruits 360 Dataset*. 2020. – URL https://www.kaggle.com/moltean/fruits/version/22. – Zugriffsdatum: 2021-01-03

[14] Seth, Kritik: *Fruit and Vegetable*. 2020. – URL https://www.kaggle.com/kritikseth/fruit-and-vegetable-image-recognition. – Zugriffsdatum: 2021-01-03

[15] Vision, Computer ; Intelligence: *An introduction to Convolutional Neural Networks*. 2018. – URL https://iitmcvg.github.io/summer_school/DLSession3/. – Zugriffsdatum: 2021-01-06

[16] Wikipedia: *künstliches neuronales Netz*. 2020. – URL https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz. – Zugriffsdatum: 2021-01-03

[17] Wikipedia: *Residual neural network*. 2021. – URL https://en.wikipedia.org/wiki/Residual_neural_network. – Zugriffsdatum: 2020-12-28

[18] Wikipedia: *Sigmoidfunktion*. 2021. – URL https://de.wikipedia.org/wiki/Sigmoidfunktion. – Zugriffsdatum: 2021-01-06