Klassifizierung von Pflanzen auf Android-Smartphones mit Machine Learning

Jan-Niklas Jacobson
jan-niklas.jacobson@haw-hamburg.de

28. Februar 2021

Zusammenfassung

Diese Arbeit präsentiert eine Lösung für lokale Bildklassifizierung mithilfe des TensorFlow Frameworks am Beispiel von Pflanzen auf Android Smartphones. Es zeigt den kompletten Weg beginnend bei der Strukturierung der Datensätze über den Aufbau der Architektur bis hin zur Einbindung des fertigen Modells in nativem Java-Code auf einem Androidgerät. Dabei liegt der Schwerpunkt auf der Bedeutung von Datenaugmentation, Transferlernen und Fine-Tuning, um das Maximum aus einem Datensatz herauszuholen. Dafür wird das MobileNetV3-Large als Basismodell genutzt und mithilfe von Transferlernen auf einen neuen Datensatz, bestehend aus vier Pflanzenarten, trainiert. Das resultierende Modell hat eine Genauigkeit von über 95 % und kann auf einem beliebigen Android-Smartphone eingesetzt werden.

1 Einleitung

Durch die steigende Popularität von Smartphones und der Nachfrage nach smarten Applikationen gibt es ein zunehmendes Interesse an lokaler Inferenz

auf mobilen Endgeräten. Das Ziel dieser Arbeit ist es, dem Lesenden einen möglichen Ansatz zur mobilen Klassifizierung von Bildern zu zeigen, die verschiedenen Parameter, welche das Ergebnis beeinflussen, zu erläutern und so den Weg zu einem funktionierenden Modell zu vereinfachen.

Bildklassifizierung ist ein häufiges Problem, welches bei maschinellem Lernen auf mobilen Endgeräten zu lösen versucht wird. Bildklassifizierung ist der Prozess, ein Bild in seiner Gesamtheit zu erfassen, es einem bestimmtes Label zuzuordnen und es somit zu klassifizieren. Typischerweise bezieht sich die Bildklassifizierung auf Bilder, in denen nur ein Objekt vorkommt und analysiert wird. Das unterscheidet sie von der *Objekterkennung*, welche neben Klassifizierung auch Lokalisierung beinhaltet und daher zur Analyse von Fällen eingesetzt werden kann, in denen mehrere Objekte im Bild vorkommen können.

State of the Art bei der maschinellen Bildklassifizierung sind heutzutage sogenannte *Convolutional Neural Networks* (*CNN*). So sind beim jährlichen Wettbewerb der Benchmark-Datenbank ImageNet (*ILSVRC*) seit 2012 alle vorne platzierten Algorithmen CNN-Strukturen [1]. Eine solche Struktur wird auch das in dieser Arbeit erstellte Modell haben.

2 Methoden

Dieser Abschnitt behandelt alle Schritte und Methoden, die nötig sind, um die Ergebnisse in Abschnitt 3 zu verstehen und zu reproduzieren.

2.1 Tools

Alle in dieser Arbeit genutzten Tools sind kostenlos erhältlich. Im Folgenden wird eine kurze Übersicht gegeben.

2.1.1 TensorFlow (Lite)

Beschäftigt man sich mit Machine Learning auf Android-Geräten, stößt man früher oder später auf Googles *TensorFlow*. Tensorflow ist ein kostenfreies Open Source-Machine-Learning-Framework. 2019 veröffentlichte Google TensorFlow Lite, welches unter anderem für mobile Endgeräte optimiert ist und die Möglichkeit bietet, TensorFlow-Modelle in TensorFlow Lite-Modelle zu konvertieren. Damit eignet es sich am besten für Machine-Learning-Inferenz auf Smartphones. Für dieses Projekt wurde die zu diesem Zeitpunkt aktuelle Version 2.4.1 verwendet.

2.1.2 Google Colaboratory

Google *Colaboratory* (kurz Colab) ist eine Jupyter Notebook-Umgebung, welche von Google kostenlos bereitgestellt wird. Colab ermöglicht das Ausführen von Python-Notebooks im Browser. Dabei kann zwischen CPU-, GPU- und TPU-Umgebungen gewählt werden. Die Plattform bietet einige Vorteile: Notebooks können von überall ausgeführt werden, das fehleranfällige Setup von Anaconda auf dem eigenen Rechner fällt weg und die Rechenkraft der zur Verfügung gestellten Hardware übertrifft die meisten Heimrechner und beschleunigt so den Lernprozess. Colab kann mit Google Drive verbunden werden, um auf Lerndaten zuzugreifen. Das komplette Training in dieser Arbeit wurde in Colab ausgeführt.

2.1.3 Android Studio

Android Studio ist die offizielle Entwicklungsumgebung von Google für die Android-Softwareentwicklung. Seit Oktober 2020 unterstützt sie TensorFlow Lite-Modells und ist damit ideal, um diese in eine Android-Applikation einzubinden.

2.2 Datenakquisition und -vorbereitung

Datenakquisition und -vorbereitung gehören zu den ersten Schritte bei maschinellem Lernen. Für dieses Projekt wurde ein Datensatz bestehend aus ca. 450 Fotos von vier verschiedenen Pflanzenarten akquiriert. Die Fotos stammen aus dem Internet von verschiedenen Quellen und haben unterschiedliche Größen, Dateitypen und Seitenverhältnisse. Um die Bilder zu vereinheitlichen hat TensorFlow einige hilfreiche Methoden. Alles, worauf bei der Akquise geachtet werden muss, ist, dass die Bilder nach Label sortiert in dementsprechend benannte Ordner einsortiert werden. Die Ordnerstruktur der Daten in diesem Projekt ist in Abbildung 1 zu sehen.

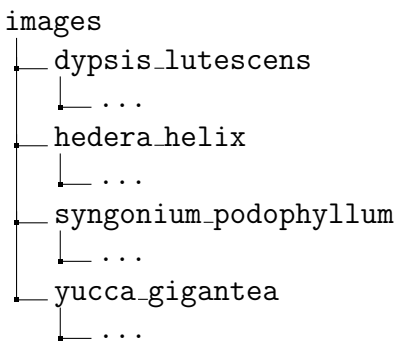

Abbildung 1: Die Rohdaten aufgeteilt in nach den Labeln benannte Ordner.

Im nächsten Schritt werden die Rohdaten mithilfe von TensorFlow in drei Datensätze aufgeteilt. 80 % der Daten werden in den Train-Datensatz eingefügt. Der Train-Datensatz beinhaltet die Bilder, mit denen das Modell später trainiert wird. Die restlichen 20 % der Daten werden erneut aufgeteilt. 80 % in Validation- und 20 % in Testdaten. Validation-Daten sind essenziell während des Trainings, um den Lernfortschritt des Modells einschätzen zu können und so Over- und Underfitting zu erkennen und vorzubeugen. Der Testdatensatz ermöglicht eine unverzerrte Schätzung der Fähigkeit des finalen, trainierten Modells und gibt so Aufschluss über den Trainingserfolg.

Bei der Aufteilung der Daten wird ebenfalls die Bild- und Batchgröße festgelegt. Die Bildgröße ist die Größe, in welche die Lerndaten zugeschnitten werden (falls deren Größe abweicht) und ist in diesem Projekt auf relativ große 224 x 224 Pixel festgelegt, da das Erkennen von Details wie Blattform

und Farbmuster unerlässlich für einen funktionierenden Pflanzenklassifikator ist. Die Batchgröße ist die Anzahl der Bilder, die dem Algorithmus während des Trainings auf einmal zugeführt werden. Größere Batchgrößen benötigen mehr Arbeitspeicher und zu große Batches können sogar die Qualität des Modells negativ beeinflussen [2]. Zu kleine Batchgrößen dagegen verlangsamen das Training, ohne bedeutende Genaugkeitsgewinne zu erbringen [3]. Es gilt also einen guten Mittelweg zu finden. In diesem Projekt bedeutet das eine Batchgröße von 16 Bildern.

2.3 Datenaugmentation

Abbildung 2: Datenaugmentation, angewendet auf ein Beispielbild aus den Trainingsdaten.

Datenaugmentation ist besonders bei kleinen Datensätzen üblich. Es beschreibt die Veränderung der vorhandenen Bilder durch Drehen, Spiegeln oder Zoomen, Einfügen eines künstlichen Rauschens oder Ähnlichem. So kann die Trainingsmenge nachträglich vergrößert und damit Overfitting vorgebeugt werden [4]. In diesem Projekt besteht die Datenaugmentation aus zufälligen horizontalen Spiegelungen sowie Rotationen und Zooms von bis zu 20 %. In Abbildung 2 ist zu sehen, wie diese Veränderungen an einem Bild aussehen können.

2.4 MobileNetV3-Large als Basismodell

Ein CNN von Grund auf aufzubauen erfordert viel Rechenaufwand, Zeit und Daten. Eine Möglichkeit, um auch mit vergleichsweise kleinen Datensätzen ein gut angepasstes Modell zu erhalten, ist *Transferlernen*. Dabei wird ein

Modell nicht komplett neu trainiert, stattdessen nutzt man ein Basismodell, welches auf einem ausreichend großen und allgemeinem Datensatz trainiert wurde und dadurch effektiv als generisches Modell für jegliche Bildklassifizierung dient. Diesem Modell fügt man einen von Grund auf neu trainierten Klassifikator hinzu. Die zuvor gelernten Feature-Maps können so für den neuen Datensatz wiederverwendet werden. Während das Basismodell also Features enthält, welche allgemein für die Klassifizierung hilfreich sind, ist der finale, klassifizierende Teil des vorab trainierten Modells dann spezifisch für die neue Klassifizierungsaufgabe trainiert.

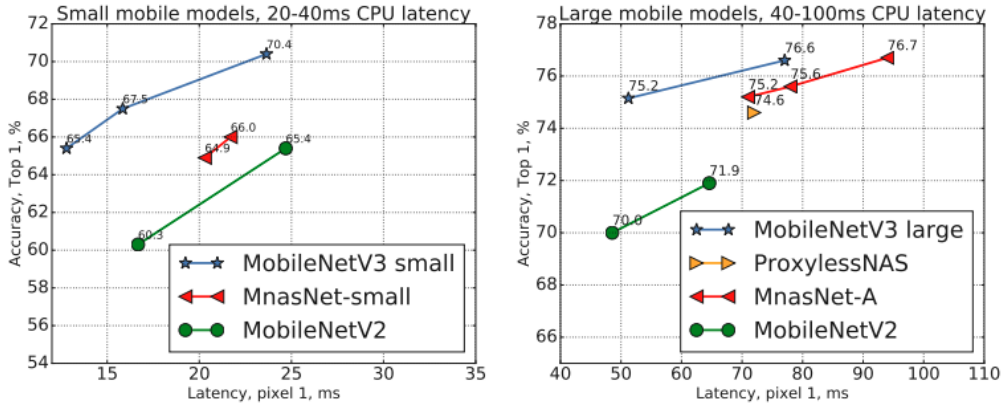

Abbildung 3: Vergleich von Genauigkeit und Latenz für mobile Modells bei der ImageNet-Klassifizierungsaufgabe unter Verwendung der Google Pixel 4 CPU [5].

Für die Zwecke dieses Projekts eignet sich Googles *MobileNetV3-Large* hervorragend als Basismodell. MobileNetV3-Large ist eine CNN-Architektur von Google, die eine gute Leistung auf mobilen Endgeräten anstrebt. Wie in Abbildung 3 zu sehen, zeichnet sich die dritte Version im Vergleich zu seinem Vorgänger MobileNetV2 unter anderem durch noch mal schnellere Inferenz bei Klassifizierungsaufgaben aus [5]. Das hier genutzte MobileNetV3-Large wurde auf dem ImageNet-Datensatz vortrainiert. ImageNet ist ein großer Forschungsdatensatz, bestehend aus 1,4 Millionen Bildern und 1000 teils sehr verschiedenen Klassen.

2.5 Modellarchitektur

```
inputs = Input(shape=IMG_SHAPE)
x = data_augmentation(inputs)
x = mobilenet_v3.preprocess_input(x)
x = base_model(x, training=False)
x = GlobalAveragePooling2D()(x)
x = Dropout(0.2)(x)
outputs = Dense(len(class_names), activation='softmax')(x)
model = Model(inputs, outputs)
```

Abbildung 4: Python-Code der Modellarchitektur.

Die Architektur des Modells in diesem Projekt besteht aus mehreren Layern. Diese werden im Folgenden genauer erläutert. In Abbildung 4 ist der dazugehörige Code zu sehen. Die Reihenfolge im Code stimmt mit der Reihenfolge der Erläuterungen überein.

- Datenaugmentation, zusammengesetzt aus Flip-, Rotation- und Zoom-Preprocessing-Layers.
- *Rescaling*, ein weiteres Preprocessing-Layer, welches die Pixelwerte der Input-Bilder an das Basismodell angleicht. In diesem Fall bedeutet das das Skalieren jedes Pixelwertes von [0,255] zu [-1,1].
- MobileNetV3-Large als eingefrorenes Basismodell ohne Klassifikator.
- Klassifikator, bestehend aus *Pooling*-, *Dropout*- und *Prediction*-Layer:
  - Pooling-Layer: Ein Global-Average-Pooling-Layer, welches für jedes Bild alle Features in ein 1280-Elemente-Vektor reduziert.
  - Dropout-Layer: Deaktiviert 20 % der Neuronen für das Prediction-Layer, um Overfitting vorzubeugen.
  - Prediction-Layer: Dense-Layer, welches die Features in vier Werte pro Bild reduziert: Die Predictions für jedes Label. Als Aktivierungsfunktion wird die *Softmax*-Funktion benutzt, um die Wahrscheinlichkeitsverteilung der vier Labels darzustellen.

2.6 Transferlernen & Fine-Tuning

Um das Basismodell anzupassen, werden in diesem Projekt zwei verschiedene Verfahren angewendet:

- Transferlernen, wie bereits in Kapitel 2.4 erläutert.
- *Fine-Tuning*, d.h. die obersten Schichten des eingefrorenen Basismodells werden aufgetaut und gemeinsam mit den neu hinzugefügten Klassifikatorschichten trainiert. So ist es möglich, die Repräsentationen von Features höherer Ordnung im Basismodell zu fine-tunen, um sie für die jeweilige Aufgabe relevanter zu machen.

2.7 Hyperparameter

Während *Modellparameter* die Eigenschaften der Trainingsdaten beschreibt, die während des Trainings durch den Klassifikator eigenständig gelernt werden, sind *Hyperparameter* Eigenschaften, die den gesamten Trainingsprozess steuern. Dazu gehören etwa *Lernrate*, Anzahl der *Epochen* oder die *Aktivierungsfunktion*. Diese können einen erheblichen Einfluss auf das Ergebnis des Lernens haben. Hyperparameter müssen vor Beginn des Traings festgelegt werden. Leider ist aber es für die meisten Hyperparameter nicht möglich, sie a priori optimal zu bestimmen. Es existieren verschiedene Ansätze für die Hyperparameteroptimierung, aber dies ist nicht Teil dieser Arbeit. Hier wurde der Einfachheit halber auf Trial & Error zurückgegriffen.

```
1 base_learning_rate = 0.0001
2
3 model.compile(
4     optimizer=Adam(lr=base_learning_rate),
5     loss=SparseCategoricalCrossentropy(),
6     metrics=['acc']
7 )
```

Abbildung 5: Python-Code der Modellkompilierung.

In Abbildung 5 ist das Kompilieren des Modells zu sehen. Dort werden einige der Hyperparameter gesetzt. Im Folgenden werden diese nun genauer beschrieben und die hier getroffene Auswahl wird erläutert.

2.7.1 Lernrate

Die Lernrate definiert, wie schnell ein Netzwerk seine Parameter updatet. Eine niedrige Lernrate konvergiert gleichmäßiger, aber möglicherweise nur zu einem lokalen Optimum, anstelle des absoluten Optimums. Sie führt außerdem zu einem langsameren Lernprozess. Eine höhere Lernrate beschleunigt demnach das Lernen, aber konvergiert möglicherweise garnicht. Die optimale Lernrate ist stark abhängig von dem nächsten Hyperparameter, dem *Optimizer*.

2.7.2 Optimizer

Der Optimizer nutzt die Ergebnisse der Loss-Function, um die Gewichte zu aktualisieren und das Modell damit in die richtige Richtung zu formen. Die Auswahl des richtigen Optimizers ist kompliziert und kann vermutlich eine eigene Arbeit füllen. In dieser Arbeit wird der Adam-Algorithmus genutzt. Adam ist ein weitverbreiteter Optimizer-Algorithmus, welcher unter anderem den Vorteil bietet, dass er die Lernrate im Verlauf des Trainings automatisch abnehmen lässt. Dies reduziert den Aufwand, der in Hyperparameteroptimierung gesteckt werden muss.

2.7.3 Anzahl der Epochen

Die Anzahl der Epochen meint die Anzahl der Male, die die gesamten Trainingsdaten dem Netzwerk während des Trainings gezeigt werden. Die Anzahl wird so lange erhöht, bis die Validierungsgenaugikeit beginnt zu sinken, auch wenn die Trainingsgenaugikeit steigt.

2.7.4 Aktivierungsfunktion

Die Aktivierungsfunktion normalisiert die *Logits*, also die rohen Predictions. Wie schon in Sektion 2.5 geschrieben, wird als Aktivierungsfunktion der Softmax-Algorithmus verwendet. Softmax wandelt die Logits in Wahrscheinlichkeiten um, welche summiert 100 % ergeben.

2.7.5 Verlustfunktion

Die Verlustfunktion ist die Funktion, die im Lernprozess versucht wird zu minimieren [6]. Ein perfekter Wert wäre also 0. Für Klassifizierungsprobleme wird typischerweise eine Kreuzentropie-Verlustfunktion gewählt. Diese berechnet einen Wert, der für alle Klassen die durchschnittliche Differenz zwischen der tatsächlichen und der vorhergesagten Wahrscheinlichkeitsverteilung zusammenfasst. In diesem Fall wurde die *Sparse*-Variante gewählt, da sich die Klassen in diesem Projekt gegenseitig ausschließen.

2.8 Post-Training-Quantisierung

Nach dem Training muss das erhaltene Keras-Modell für die Inferenz auf Android-Geräten noch in ein TFLite-Modell umgewandelt werden. Dafür kann der TFLiteConverter von TensorFlow genutzt werden. Bei der Umwandlung können außerdem Optimierungen festgelegt werden, z.B. *Post-Training-Quantisierung*. Es wird zwischen drei verschiedenen Techniken zur Quantisierung unterschieden, von denen hier aber nur auf die simpelste eingegangen wird: die Dynamikbereich-Quantisierung. Dabei werden die Gewichte statisch von Fließkomma zu 8-Bit-Integer quantisiert. So quantisierte Modelle sind etwa vier Mal kleiner und zwei bis drei Mal schneller, wobei die Genauigkeit um weniger als ein Prozent abnimmt [7]. Für diese Art der Quantisierung wurde sich auch in dieser Arbeit entschieden.

2.9 Hinzufügen von Metadaten

Um das neue Modell in einer Android-App zu nutzen, müssen vorher Metadaten hinzugefügt werden. Diese Metadaten enthalten von Menschen lesbare Teile, welche die besten Praktiken bei der Verwendung des Modells vermitteln, und maschinenlesbare Teile, welche von Codegeneratoren genutzt werden können, wie der TFLite Android-Codegenerator und die Android Studio ML-Bindungsfunktion. Um das Modell in der TFLite Beispielapp zu nutzen, werden Metadaten vorausgesetzt.

2.10 Nutzung in Android

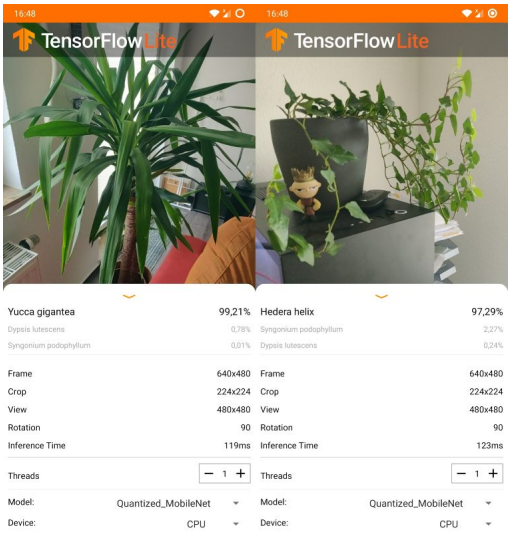

Abbildung 6: Screenshots aus der Beispielapp, welche den Einsatz des in diesem Projekt trainierten Modells auf einem OnePlus 6 Smartphone zeigen.

Ist das Modell mit Metadaten befüllt, kann es in Androidstudio importiert werden. Dafür besitzt die IDE einen eigenen Importer. Im Falle der Beispielapp reicht das Ersetzen eines der bestehenden Modelle. Danach kann die App auf einem beliebigen Androidsmartphone ausgeführt und das Modell im realen Betrieb getestet werden. Das Ergebnis sollte ähnlich aussehen wie in Abbildung 6 zu sehen.

3 Ergebnisse

Im Folgenden werden die Ergebnisse des Trainings und der Inferenz auf Android gezeigt und erläutert.

3.1 Training

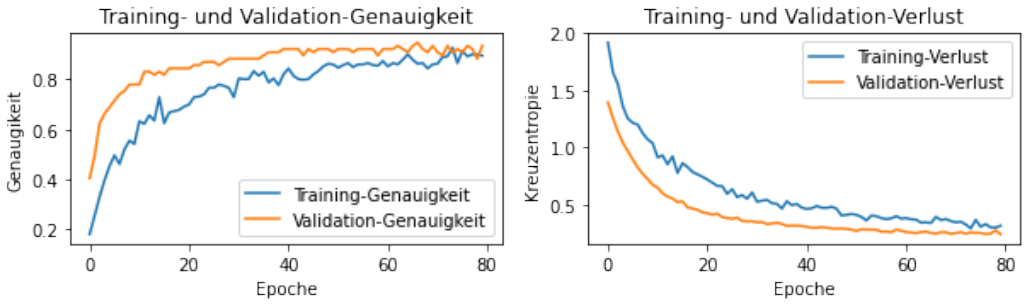

Abbildung 7: Graphen der Konvergenz von Genauigkeit und Verlust auf den Train- und Validation-Datensätzen bei dem initialen Lernvorgang des Modells.

Als erstes wurde das Modell mit eingefrorenem Basismodell trainiert. Das Modell wurde dabei mit einer Lernrate von 10^{-4} über 80 Epochen trainiert. In Abbildung 7 ist das Konvergieren von Genauigkeit und Verlust zu erkennen. Diese Anzahl von Epochen wurde gewählt, da auf dem Validation-Datensatz die Genauigkeit bei etwa 50 Epochen und der Verlust bei etwa 70 Epochen konvergiert. Danach verbessert sich das Modell nur noch auf den Trainingsdaten. Etwa bei 80 Epochen war das Modell auf beiden Datensätzen ähnlich genau.

Auf dem Testdatensatz hatte das Modell nach diesem Training eine Genauigkeit von 93,75 % bei einem Verlust von 20,93 % und auf den Validation-Daten eine Genauigkeit von 93,51 % und einen Verlust von 24,06 %.

3.2 Fine-Tuning

Im nächsten Schritt wurden die obersten Schichten des Basismodells aufgetaut, um ein Fine-Tuning zu ermöglichen. Nur die untersten 100 Schichten wurden eingefroren gelassen, da die unteren Schichten eher die allgemeinen Erkennungsmerkmale enthalten. Um Overfitting vorzubeugen, wurde die Lernrate hier auf 10^{-5} reduziert, da deutlich mehr Parameter trainiert werden. Das Fine-Tuning erstreckte sich über 20 Epochen.

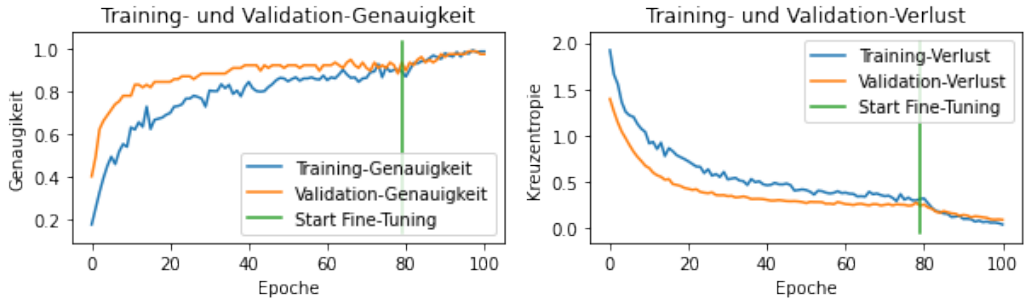

Abbildung 8: Graphen der Konvergenz von Genauigkeit und Verlust auf den Train- und Validation-Datensätzen des kompletten Lernvorgangs des Modells mit eingezeichnetem Beginn des Fine-Tunings.

Nach dem Fine-Tuning hat sich die Genauigkeit auf dem Testdatensatz auf 100 % verbessert und der Verlust ist auf 5,15 % gesunken. Leider ist der Testdatensatz auf Grund der geringen Menge an Daten sehr klein und besteht nur aus 16 Bildern. Auf den Validation-Daten hat das Modell in der letzten Epoche eine Genauigkeit von 97,4 % bei einem Verlust von 8,97 %.

3.3 Inferenz auf Android-Smartphones

Um die Inferenzzeit auf Android-Geräten zu testen, wurde als erstes die Benchmark-App von TensorFlow genutzt [8]. Die Inferenzzeit wurde auf zwei unterschiedlichen Geräten getestet: einem OnePlus 3T (OP3T) von 2016 und einem OnePlus 6 (OP6) von 2018. Das OP3T hat als CPU einen Qualcomm Snapdragon 821 und das OP6 einen Snapdragon 845. Der Benchmark wurde jeweils auf beiden Geräten auf der CPU mit verschiedenen Anzah-

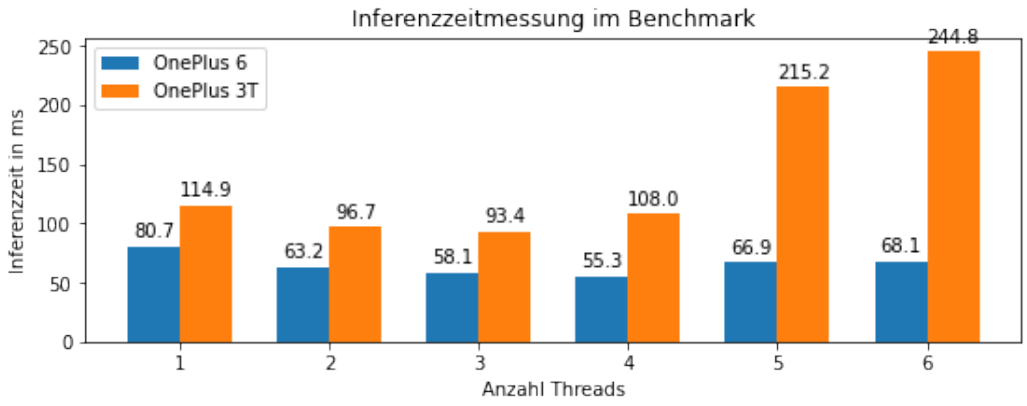

Abbildung 9: Graphen der Inferenzzeit in der Benchmark-App auf zwei verschiedenen Android-Geräten bei unterschiedlichen Anzahlen von Threads.

len an Threads ausgeführt. In Abbildung 9 ist die Inferenzzeit nach Gerät und Threadanzahl aufgeschlüsselt. Bei den beiden Geräten gibt es eine unterschiedliche Optimalanzahl an Threads, um die beste Performance zu erzielen: vier Threads auf dem OP6 und drei auf dem OP3T. Berücksichtigt man nur die Inferenzzeiten bei dieser optimalen Threadanzahl, ist das (zwei Jahre jüngere) OP6 40.8 % schneller.

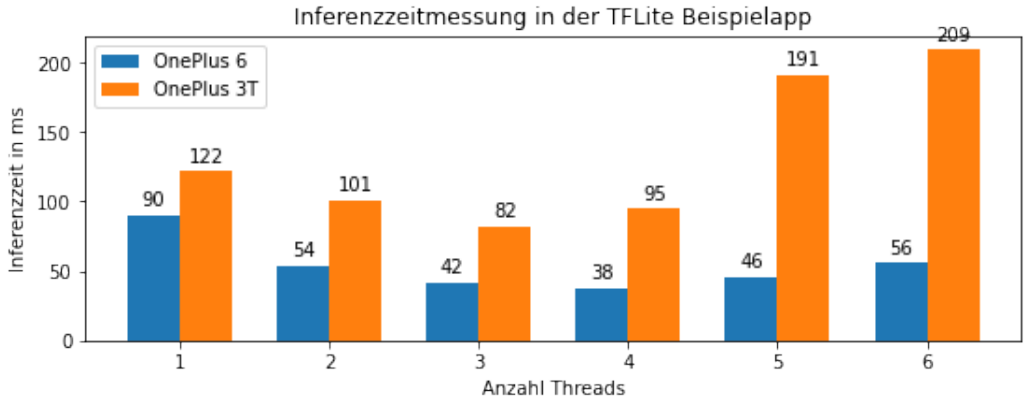

Abbildung 10: Graphen der Inferenzzeit in der TFLite Beispiel-App auf zwei verschiedenen Android-Geräten bei unterschiedlichen Anzahlen von Threads.

Das TensorFlow-Team schreibt, dass die in der Benchmark-App gemessenen Inferenzzeiten leicht von denen im realen Betrieb in einer App abweichen

können [8]. Während man im Internet eher von höheren Zeiten im Realbetrieb liest, waren hier die beobachteten Inferenzzeiten des Modells in der TFLite Bildklassifizierungsbeispielapp [9] tatsächlich etwas niedriger, wie in Abbildung 10 zu sehen ist. Die optimalen Threadanzahlen sind hingegen gleichgeblieben.

4 Fazit

Das Ziel dieser Arbeit war es, ein Modell für mobile Inferenz zu erstellen, welches eine kleine Auswahl an Pflanzen klassifizieren kann und den Weg dorthin verständlich zu machen. Unterschiedliche State of the Art Tools wurden dem Lesenden näher gebracht. Die verschiedenen Parameter, die vor und nach dem Training eingestellt werden können, wurden erläutert. Das Endprodukt ist ein Modell, welches in einer beliebigen Android-App genutzt werden kann und eine Genauigkeit von über 95 % hat.

Leider war die Anzahl der Daten begrenzt, weshalb nicht genug Testbilder vorhanden waren, um Genauigkeit und Verlust des Modells zweifelsfrei zu bestimmen. Auch wäre eine größere Anzahl von Klassen interessant, um zu sehen, inwiefern sich die Genauigkeit mit steigender Anzahl verändert.

Literatur

[1] ImageNet Large Scale Visual Recognition Challenge. `http://image-net.org/challenges/LSVRC`.

[2] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836*, 2016.

[3] Dmytro Mishkin, Nikolay Sergievskiy and Jiri Matas. Systematic evaluation of CNN advances on the ImageNet. *arXiv:1606.02228*, 2016.

[4] Connor Shorten and Taghi M. Khoshgoftaar. A survey on Image Data Augmentation for Deep Learning. J Big Data 6, 60. https://doi.org/10.1186/s40537-019-0197-0, 2019.

[5] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le and Hartwig Adam. Searching for MobileNetV3. *arXiv:1905.02244*, 2019.

[6] Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning, 82. The MIT Press; Illustrated edition, 2016.

[7] TensorFlow Model Optimization Toolkit - Post-Training Integer Quantization. https://medium.com/tensorflow/tensorflow-model-optimization-toolkit-post-training-integer-quantization-b4964a1ea9ba, 2019.

[8] TensorFlow Lite guide. Performance measurement. https://www.tensorflow.org/lite/performance/measurement.

[9] TensorFlow Lite image classification Android example application. https://github.com/tensorflow/examples/tree/master/lite/examples/image_classification/android.