RSTensorFlow
-
Analyse einer Deep Learning Bibliothek mit GPU-Unterstützung für mobile Geräte

Hochschule München
Michael Engel

I. Abstract

Die Bedeutung mobiler Geräte ist in den vergangenen Jahren enorm gestiegen. So unterstützen diese kleinen Helfer den Anwender bei einer Vielzahl von Aufgaben im Alltag. Für viele dieser komplexen Anwendungen und Funktionen werden verschiedenste Deep Learning Modelle eingesetzt. Zur Spracherkennung sind beispielsweise Deep Recurrent Neural Networks die bevorzugte Wahl bei der Umsetzung. Solche Modelle führen allerdings sehr viele rechenintensive Berechnungen durch, weshalb diese bevorzugt als Client-Server-Anwendung umgesetzt werden. Das mobile Gerät schickt lediglich die Eingangsdaten an einen Server auf dem das Deep Learning Model ausgeführt wird. Zwar unterstützen einige Frameworks das Ausführen von Deep Learning Modellen auf mobilen Endgeräten, jedoch wird von diesen nur die CPU genutzt. Diese Arbeit untersucht RSTensorFlow, eine Modifikation des beliebten OpenSource-Frameworks TensorFlow, und versucht die hierfür durchgeführten Experimente zur Performance-Bewertung für andere mobile Geräte nachzustellen. Zunächst wird in diesem Paper sowohl RSTensorFlow und die in dem Projekt modifizierten Operationen als auch die von TensorFlow zur Verfügung gestellten Android-Apps näher vorgestellt. Bei eine technische Analyse von RSTensorFlow wird dann der Quellcode und der Build-Prozess von TensorFlow untersucht. Dabei werden auch die vorgenommen Änderungen von RSTensorFlow vorgestellt. Bevor die durchgeführten Experimente und dessen Resultate näher beschrieben werden, ist es nötig die von den Autoren durchgeführten Experimente zu rekonstruieren. Diese Rekonstruktion führt zur Definition von weiteren Metriken und Anpassungen am Quellcode von RSTensorFlow für die Experimente.

II. Einleitung

In den vergangenen Jahren haben sich mobile Geräte, wie beispielsweise Smartphones, zu Assistenten des Alltags entwickelt. Auf diesen kleinen Helfern werden viele komplexe Anwendungen und Funktionen genutzt, welche verschiedenste Deep Learning Modelle verwenden. So werden beispielsweise Deep Recurrent Networks Neural Networks zur Spracherkennung Graves et al. (2013) genutzt. Zur Erkennung von Objekten werden Deep Convolutional Networks eingesetzt Krizhevsky et al. (2012). Deep Learning Modelle führen viele sehr rechenintensive Berechnungen durch. Deshalb wird bevorzugt eine Art Client-Server-Anwendung umgesetzt, welche das Deep Learning Modell auf einem separaten Server ausführt. Die mobilen Clients schicken über eine entsprechende App die benötigten Eingangsdaten für das Netzwerk an den Server und dieser schickt das Ergebnis des Deep Learning Netzwerks zurück an den Client. Dies ist sowohl zeit- als auch kostenintensiv. Um Deep Learning Netzwerke direkt auf dem Mobilgerät zu verwenden, bieten Frameworks wie TensorFlow (https://www.tensorflow.org/) eine eigene Variante der Software an. Jedoch nutzen diese nicht alle Rechenressourcen des Endgeräts, sondern lediglich die CPU. Für größere Applikationen ist dies oft nicht ausreichend, um in angemessener Zeit ein adäquates Ergebnis zu erhalten. Projekte wie RSTensorFlow Alzantot et al. (2017) versuchen daher eine Unterstützung der GPU in das bestehende OpenSource-Framework TensorFlow zu integrieren und den Erfolg anhand geeigneter Metriken zu bewerten.
Die Veröffentlichung zum Projekt *RSTensorFlow* Alzantot et al. (2017) dient als zentrale Arbeit dieses Papers und stellt den Ausgangspunkt dar. Die in Alzantot et al. (2017) beschriebenen Experimente werden in diesem Paper nachgestellt und für andere Geräte, welche nicht direkt von Google hergestellt wurden, durchgeführt. Über die angepasste *TF Classify* App werden die Ausführungszeiten gemessen. Ebenso wird mit Hilfe der *Trepn Profiler* App die Auslastung der CPU und GPU überprüft. Diese Anwendung dient ebenfalls dem Aufzeichnen weiterer Metriken, wie beispielsweise der Auslastung des Speichers. Diese wurde bereits in Alzantot et al. (2017) als ein mögliches Bottleneck hinsichtlich der Performance vermutet. Für eine Optimierung der Conv2D-Operation wird das sogenannte Unfolding näher betrachtet. In Chellapilla et al. (2006) sowie in Rajbhandari et al. (2017) werden mit dieser Technik Convolutional Networks optimiert. Hierfür wägen beide Arbeiten das Potential des Unfoldings ab und erläutern die Funktionsweise.

III. Related Work

In Alzantot et al. (2017) wird die derzeit noch eher mangelhafte Unterstützung für tiefe neuronale Netze auf mobi-

len Endgeräten erläutert. Die meisten Deep Learning Frameworks bieten zwar eine Variante für Mobilgeräte, können jedoch lediglich die CPU als Recheneinheit nutzen. Aktuelle Forschungen beschäftigen sich damit, diese Frameworks für Mobilgeräte zu erweitern, sodass weitere Komponenten für die komplexen Berechnungen tiefer neuronaler Netze genutzt werden können. Hier steht insbesondere die GPU als weitere Rechenressource im Fokus der Forschung für unterschiedlichste Anwendungen neuronaler Netze. Um die GPU nutzen zu können, wird von verschiedenen Forschungsprojekten entweder OpenCL ope oder RenderScript ren verwendet. Im Bereich der Continuous Vision Applikationen wird in Huynh et al. (2017) ein GPU-basiertes Deep Learning Framework vorgestellt. Hier wird zur Nutzung der GPU OpenCL ope in Kombination mit Vulkan vul eingesetzt. Ebenso greift wird in Huynh et al. (2016) auf OpenCL ope zurückgegriffen, um ein Framework für Deep Convolutional Neural Networks für mobile Geräte mit GPU-Unterstützung umzusetzen. Ein weiteres Forschungsprojekt ist CNNDroid Latifi Oskouei et al. (2016). Bei der Implementierung CNNDroid handelt es sich um eine OpenSource-Bibliothek für trainierte CNN auf Android-Geräten. Im Gegensatz zu den anderen Projekten integriert RSTensorFlow die GPU-Unterstützung in das beliebte, bereits bestehende Deep Learning Framework *TensorFlow*.
Für die Evaluierung des umgesetzten Frameworks werden von allen vorgestellten Forschungsprojekten Metriken wie die Ausführungszeit betrachtet. Diese Arbeit greift die allgemeine Vorgehensweise der in Alzantot et al. (2017) vorgestellten Experimente auf und wendet dies für andere Geräte an. Damit soll eine Validierung der in Alzantot et al. (2017) vorgestellten Ergebnisse erzielt werden. Die nicht performante Nutzung des Arbeitsspeichers von RenderScript wird von den Autoren von Alzantot et al. (2017) als ein möglicher Grund für die verschlechterte Ausführungszeit bei der Conv2D-Operation vermutet. Daher werden in diesem Paper weitere Metriken, wie beispielsweise die Speicherauslastung, betrachtet.

IV. RSTensorFlow

Bei *RSTensorFlow* handelt es sich um eine modifizierte Variante von *TensorFlow*. Es wurde der Kernel von TensorFlow angepasst, sodass dieses Tool über *RenderScript* die heterogenen Rechenressourcen von Android-Geräten nutzen kann. Das zu RSTensorFlow veröffentlichte Paper Alzantot et al. (2017) beschreibt unter anderem Experimente, welche die angepasste Version RSTensorFlow mit dem Original hinsichtlich der Performance vergleichen. Während für die Matrixmultiplikation eine Verbesserung festgestellt werden konnte, wird die Convolution-Operation bei der Verwendung von RenderScript langsamer durchgeführt.

A. Android-Demo

Für Android stellt TensorFlow bereits ein Android Package Kit (APK) zur Verfügung. Diese APK kann über die Homepage von TensorFlow ten (a) als Prebuild bezogen werden. Da sowohl in Alzantot et al. (2017) als auch im Rahmen dieser Arbeit Anpassungen an Tensorflow vorgenommen werden, wird die APK jedoch selbst kompiliert. Der Build-Prozess für die Demo-Apps wird in V-B näher erläutert. Die APK kann dann über das Android Debug Bridge (ADB) Tool auf dem Zielsystem installiert werden.
Die Demo-APK enthält in der verwendeten Version drei Apps, welche auf dem Zielgerät installiert werden. Alle drei Apps verwenden die Kamera des Smartphones und führen verschiedene Deep Learning Anwendungen auf Basis der aufgenommenen Bilder aus. Die App *TF Stylize* kennt mehrere Kunststile von verschiedenen Künstlern und passt das Kamerabild entsprechend des ausgewählten Stils an. *TF Detect* hingegen verwendet die TensorFlow Object Detection API zur Erkennung von Objekten im Bild aus 80 verschiedenen Kategorien in Echtzeit. Für diese Arbeit ist jedoch nur die *TF Classify* App relevant. Diese verwendet das Google Inception Model Szegedy et al. (2014), um die Gegenstände im Bild zu klassifizieren und die besten drei Ergebnisse anzuzeigen. Seit dem Release 1.4 von TensorFlow, welcher dem Branch *r1.4* des GitHub-Repositories ten (b) entspricht, enthält die Demo-APK eine vierte App zur Spracherkennung. *TF Speech* nutzt ein einfaches Speech Recognition Model und zeigt erkannte Wörter in der App an.

B. Modifizierte Operationen

Das Ziel in Alzantot et al. (2017) war es die zeitaufwendigsten Deep Learning Operationen zu optimieren. Hierfür wurden zunächst die prozentualen Anteile jedes Operationstyps während des Vorwärtspasses durch das verwendete Inception Model ermittelt. Mit ca. 75% haben die Convolution-Operationen (Conv2D) den größten Anteil an der Berechnungszeit im Vorwärtspass. Die Matrixmultiplikationen (matmul) sind mit ca. 7% auf dem zweiten Platz. Für diese beiden rechenintensiven Deep Learning Operationen wurde in Alzantot et al. (2017) versucht eine Performance-Steigerung durch die Verwendung von RenderScript zu erzielen.
Beide Operationen sind im Deep Learning Bereich von großer Bedeutung. Bei der Operation matmul wird eine einfache Multiplikation zweier Matrizen der Form

$$R^{l \times m} \times R^{m \times n} \rightarrow R^{l \times n}$$

durchgeführt. Die Convolution-Operation bei Deep Learning Modellen verwendet Tensoren. Ein Tensor wird in TensorFlow durch ein multidimensionales Array dargestellt. Bei Bildern werden häufig Tensoren T_E der Form $Width_E \times Height_E \times Depth_E$, und damit einer Dimension $D_E = 3$, als Eingabe für die Convolution-Operation verwendet. Ein weiterer Tensor T_F wird als Filter auf T_E angewendet, sodass ein Ausgabe-Tensor T_A mit reduzierten Dimensionen entsteht. Hierfür wird T_F in gleichmäßigen Intervallen, dem sogenannten Stride S, entlang der Breite und Höhe von T_E geschoben. In jedem Schritt wird das Skalarprodukt von T_F und dem Ausschnitt aus T_E gebildet. Die Produkte jeder Ebene in der Tiefe werden dann aufsummiert und das Ergebnis an die jeweilige Position in T_A gesetzt. Damit wird die Form von T_A durch die Form von T_E, T_F und diversen weiteren Hyperparametern, wie dem Stride S oder die Anzahl der Filter N, bestimmt. Die Unterlagen zu der Vorlesung *CS231n: Convolutional Neural Networks for Visual Recognition* der Stanford University Li et al. (2017)

gewähren einen tieferen Einblick in die Conv2D-Operation und Convolutional Neural Networks (CNN) im Allgemeinen. Die Convolution-Operation, welche bei Deep Learning Modellen verwendet wird, ist somit nicht mit der Faltung aus der Funktionalanalysis zu verwechseln.

V. Technische Analyse von RSTensorFlow

Bei TensorFlow handelt es sich um ein umfangreiches und komplexes Projekt, weshalb eigene Anpassungen ein gutes Verständnis dieses Deep Learning Framworks erfordern. In diesem Kapitel wird zunächst eine technische Analyse des Quellcodes im Hinblick auf die von RSTensorFlow vorgenommenen Anpassungen in V-A durchgeführt. Hier wird die Quellcode-Struktur von TensorFlow im Allgemeinen, sowie die Anpassungen von RSTensorFlow erläutert. Der Build-Prozess für die Android-Demo wird in V-B untersucht.
Die von TensorFlow bereits implementierten Demos für Android sind in Java geschrieben und verwenden das Android SDK. Das Deep Learning Framework TensorFlow hingegen ist mit C++ umgesetzt. Als Schnittstelle zwischen TensorFlow und Android wird das Android Native Development Kit (NDK) eingesetzt. Das NDK ermöglicht die Verwendung von C/C++ auf einer Android-Plattform und nutzt die Java Native Interface (JNI) API.
Das Framework TensorFlow wird für die Android-Demos in die Shared Object Datei *libtensorflow_inference.so* gepackt. In dieser .so-Datei befindet sich ebenfalls das *TensorFlowInferenceInterface*, welches den Zugriff der Android-App auf TensorFlow durch das NDK realisiert. Somit können Deep Learning Anwendungen für Android auf diesen Kern aufsetzen und bequem Android-Activities, welche auf das Inference Interface zugreifen, umgesetzt werden.

A. Analyse des Quellcodes

In TensorFlow ist für die Android-Demo Apps ein eigenes Teilprojekt definiert. Dieses beinhaltet die Activity-Klassen der in IV-A beschriebenen drei Apps. Die TF Classify App wird auf Basis der *ClassifierActivity.java* erstellt. Hier ist auch der Pfad zur verwendeten Model-Datei innerhalb der späteren APK deklariert. Diese enthält das von TF Classify verwendete Google Inception Model zur Klassifizierung von Objekten und liegt als Protocol Buffer (.pb) vor. Mit TensorFlow und TensorBoard kann dieses bei Bedarf visualisiert werden, um so ein tieferes Verständnis bei der Objektklassifizierung von TF Classify und der durchgeführten Operationen zu erlangen.
Für die meisten Deep Learning Operationen wird von TensorFlow die in C++ geschriebene Template-Bibliothek *Eigen* eig verwendet. Diese berechnet Operationen der linearen Algebra schnell und achtet dabei auf den verbrauchten Cache-Speicher. Eigen wird somit auch für die Matrixmultiplikation und die Convolution-Operation verwendet. Damit eine substituierende Verwendung von RenderScript möglich ist, müssen die Aufrufe der Eigen-Bibliothek im Kernel von TensorFlow ersetzt werden. RenderScript ist jedoch für die direkte Verwendung in Java konzipiert, weshalb die Integration in den TensorFlow-Kernel komplex ist. Die Autoren von Alzantot et al. (2017) haben zur Lösung des Problems für den Quellcode von RenderScript einen Wrapper in C++ umgesetzt, welcher das Skript aufruft. Die RenderScript-Dateien werden in separaten Shared Object Dateien gekapselt und stehen somit der Anwendung zur Verfügung. Für RSTensorFlow wurden so die Operationen matmul und conv2D in RenderScript umgesetzt. Damit TensorFlow statt der Eigen-Implementierung die RS-Variante nutzt, sind Anpassungen am Kernel von TensorFlow nötig. Hier müssen die Quelldateien

- conv_ops.cc und
- matmul_op.cc

im Kernel so geändert werden, dass statt des Aufrufs der Eigen-Bibliothek die jeweilige Funktion des RenderScript-Wrappers aufgerufen wird. Da es sich um eine Änderung am Kernel handelt, wirken sich diese direkt auf alle darauf aufsetzenden Funktionen aus. An den einzelnen Android-Activities ist daher keinerlei weitere Anpassung nötig. Die TF Classify App nutzt somit automatisch die RenderScript-Implementierung.

B. Der Build-Prozess

Für den Build-Prozess von TensorFlow wird das von Google entwickelte Opensource Tool Bazel (https://bazel.build/) verwendet. Bazel automatisiert den Build und unterstützt standardmäßig mehrere Programmiersprachen wie Java und C++. Durch eine *WORKSPACE*-Datei wird ein Ordner und dessen Inhalt als Bazel Workspace definiert. Ebenso werden in der WORKSPACE-Datei externe Abhängigkeiten des Projekts, wie beispielsweise das Android SDK, referenziert. *BUILD*-Dateien innerhalb des Workspaces definieren Packages, welche der Organisation des Projekts dienen. Innerhalb der BUILD-Datei können Labels, Build-Regeln und mehr definiert werden. Das Projekt für die in V-A beschriebenen Shared Objects der RenderScript-Implementierung ist innerhalb der Projektstruktur von TensorFlow im Contribution-Ordner als neues Package angelegt. Im Contrib-Ordner werden Teilprojekte gesammelt, die irgendwann ein fester Bestandteil des Kerns von TensorFlow werden können.
Um die Android-Demo von TensorFlow auf Basis des Quellcodes zu bauen, wird *bazel build* mit dem Build-Target *//tensorflow/examples/android:tensorflow_demo* aufgerufen. Getrennt durch einen Doppelpunkt gibt der vordere Teil des Textes den Pfad der BUILD-Datei relativ zum Workspace an. Der hintere Teil spezifiziert die für den Build anzuwendende Regel. Bei *tensorflow_demo* wird eine Android-Regel angesprochen, welche eine APK als Ergebnis hervorbringt. Hier wird unter anderem das TensorFlow Inference Interface als Abhängigkeit eingebunden. Ebenso wird TensorFlow selbst als C/C++-Regel in den Prozess eingebunden. Für die Integration des RenderScript-Projekts im Contribution-Ordner wird dieses Package innerhalb der C/C++-Regel für die TensorFlow-Bibliotheken hinzugefügt. Jedoch wird hier lediglich der Ordner mit den bereits vorliegenden .so-Dateien eingebunden. Allerdings beinhaltet dieser Ordner nicht die bereits kompilierte *librs.mScriptConv.so*, welche die in RenderScript implementierten Operationen matmul und conv2D enthält. Um den Build-Prozess nicht zu verändern, wird das Projekt manuell

neu kompiliert und die resultierende *librs.mScriptConv.so* in den entsprechenden Ordner kopiert.

VI. Experimente

In diesem Paper werden die Experimente aus Alzantot et al. (2017) nachgestellt, um so eine Validierung der dortigen Ergebnisse zu ermöglichen. Damit die Resultate vergleichbar sind, werden die Experimente rekonstruiert. Diese Rekonstruktion wird in VI-A genauer beschrieben. Zusätzlich zu den in Alzantot et al. (2017) verwendeten Metriken, wie beispielsweise die Ausführungszeit, werden in dieser Arbeit weitere mögliche Metriken in Abschnitt VI-B für eine Anwendung beim Vergleich zwischen Eigen und RenderScript diskutiert. Ebenso wird in Abschnitt VI-C erläutert, welche Änderungen am Quellcode von RSTensorFlow durchgeführt wurden. Für diese Modifikationen wurde ein neuer Fork Engel vom RSTensorFlow-Repository auf GitHub erstellt. Hier wurden im Branch *RenderScript* nicht nur die Code-Änderungen veröffentlicht, sondern auch die Ergebnisse der in VI-E dargestellten Auswertungen.
Während in Alzantot et al. (2017) zwei Nexus-Geräte für die Durchführung der Experimente genutzt wurde, werden in dieser Arbeit zwei Samsung-Smartphones verwendet. Die Tabelle I stellt die genaueren Spezifikationen dieser Hardware dar.

Modell	Samsung J5	Samsung S7
Android OS	6.0	6.0
CPU	4x1,2 GHz	4x2,3 GHz und 4x1,6 GHz
GPU	Adreno 306	Mali-T880 MP12
RAM	1,5 GB	4 GB

Tabelle I
Ein Auszug aus der Hardware-Spezifikation nach https://www.gsmarena.com für die gewählten Smartphone-Modelle für die Experimente.

A. Rekonstruktion

Damit die in dieser Arbeit durchgeführten Experimente mit denen aus Alzantot et al. (2017) vergleichbar sind, wird versucht diese zu rekonstruieren. In Alzantot et al. (2017) liegt der Fokus für die Bewertung der RenderScript-Implementierung auf den beiden Operationen matmul und conv2D. Hierfür wurden die Metriken

- CPU-Last und
- Ausführungszeit

betrachtet.
Für die Messung der Auslastung der CPU wurde die App *Trepn Profiler* eingesetzt. Für die Messungen der Ausführungszeit wurde der Quellcode angepasst und ein Block zur Zeitmessung um die Anweisung für die jeweilige Operation gesetzt. Das Ergebnis wird dann auf einen Ausgabestrom gegeben, auf welchem mit dem *ADB*-Tool zugegriffen werden kann. Dies kann in den beiden Quellcode-Dateien

- conv_ops.cc und
- matmul_op.cc

überprüft werden. Jedoch sind bei den Auswertungen der Experimente in Alzantot et al. (2017) Multiplikationen mit quadratischen Matrizen dargestellt. Das Google Inception Model, welches von der TF Classify App verwendet wird, nutzt jedoch keine quadratischen Matrizen. In der Auswertung für die Convolution-Operation werden Filterzahlen aufgeführt, welche ebenfalls nicht im Google Inception Model für diese Operation definiert sind. Daher wird vermutet, dass für die Experimente in Alzantot et al. (2017) eine separate App genutzt wurde. In dieser Arbeit wird direkt auf der TF Classify App aufgesetzt. Somit ändern sich die Größen der Matrizen und die Anzahl der Filter. Die Vergleichbarkeit zu den Experimenten aus Alzantot et al. (2017) bleibt trotzdem bestehen, da es sich hier um variable Größen handelt und für den Vergleich zwischen Eigen und RenderScript weiterhin die gleiche Metrik verwendet wird.

B. Metriken

In Alzantot et al. (2017) werden als Metriken für den Vergleich zwischen Eigen und RenderScript die Ausführungszeit und Auslastung der CPU bei der Ausführung einer Operation betrachtet. Häufig werden weitere Metriken bei der Bewertung von Bibliothek für mobile Geräte wie Eigen oder RenderScript herangezogen. In Huynh et al. (2017) wird u.a. auch der Stromverbrauch gemessen. Diese Metrik ist gerade bei mobilen Geräten mit der limitierten Akkulaufzeit von großer Bedeutung. Diese Metrik wird in dieser Arbeit jedoch nicht genutzt. Der Fokus liegt auf Metriken, um die Performance beurteilen zu können.
Über die Trepn Profiler App lassen sich viele Kenngrößen der Performance auf mobilen Geräten messen. So kann nicht nur die Last der CPU im Allgemeinen, sondern sogar jedes einzelnes Kerns genau aufgezeichnet werden. Im Rahmen des Experimentes ist dieser Detailgrad nicht nötig, weshalb nur die gesamte CPU-Auslastung interessant ist. Die Autoren von RSTensorFlow vermuteten bei Ihrer Auswertung der Experimente einen Zusammenhang zwischen der schlechteren Performance von RenderScript und der Nutzung des Arbeitsspeichers. Daher wird diese Ressource über den Trepn Profiler aufgezeichnet. Eine weitere, interessante Metrik ist die Auslastung der GPU. Durch das Aufzeichnen dieser Kenngröße ist es möglich festzustellen, ob RenderScript tatsächlich in der Lage ist die Rechenressourcen mobiler Geräte zu nutzen. Diese Größe kann von der Trepn Profiler App nur für das Samsung J5 mitgeschnitten werden. Die GPU-Last ist für das Samsung S7 durch den Trepn Profiler leider nicht aufzuzeichnen, da diese App die in dem Gerät verbaute GPU nicht unterstützt. Die Verwendung einer alternativen Anwendung ist keine Option, da für das Experiment die gleiche Anwendung wie in der Referenzarbeit zu nutzen ist.
Über die Android Debug Bridge können die Ausführungszeiten von Operationen aufgezeichnet werden. Für die Matrixmultiplikation sind jedoch auch Informationen bezüglich der Größe der beteiligten Matrizen von Bedeutung. Das Gleiche gilt für die Convolution-Operation. Hier sind Informationen wie beispielsweise die Größe der Tensoren oder die Größe des Strides interessant. Die Anzahl der verwendeten Filter

für die Convolution-Operation entspricht genaue der Tiefe des Output-Tensors, weshalb für die Filterzahl keinerlei weitere Informationen ermittelt werden müssen. Die für das Aufzeichnen dieser Daten nötigen Anpassungen am Quellcode werden in VI-C näher erläutert.

C. Anpassungen von RSTensorFlow

Im Abschnitt VI-B wurde Notwendigkeit von Anpassungen des Quellcodes von RSTensorFlow dargestellt. Diese Änderungen werden in den C++-Dateien

- conv_ops.cc und
- matmul_op.cc

direkt im Kernel von TensorFlow vorgenommen. RSTensorFlow hat in beiden Klassen bereits einen einfachen Mechanismus zur Ermittlung der Ausführungszeit innerhalb der Methode

```
void Compute(OpKernelContext* ctx)
```

implementiert. Die Ausgabe der Zeit wird mit einem *stringstream* umgesetzt, welcher innerhalb der Android-App den Stream für das Logging verwendet. Somit lassen sich die Informationen aus dem Kernel wie einfache Log-Ausgaben mit dem ADB-Tool abgreifen. Für die Android-Apps von TensorFlow entspricht dies dem Befehl:

```
adb logcat -s TF\_ANDROID\_LOG
```

Die Ausgabe wird nun für die Matrixmultiplikation um Form der beiden Matrizen und der Ergebnis-Matrix erweitert. Bei der Convolution-Operation wird die Ausgabe des String-Streams um die Form des Eingabe-, Filter- und Ausgabe-Tensors sowie des zu verwendenden Strides und das sog. Zero-Padding erweitert. Für das einfache Wechseln zwischen der Eigen- und RenderScript-Implementierung wird außerdem die Präprozessor-Anweisung

```
#define USE_RENDERSCRIPT
```

festgelegt. Ist diese definiert, so wird beim bauen der Android-Apps die RenderScript-Implementierung verwendet. Damit stattdessen die Eigen-Bibliothek genutzt wird muss dieses Flag lediglich auskommentiert werden. Die Steuerung des Wechsels über die Oberfläche der fertigen Android-App wäre eine mögliche Erweiterung.
Die hier beschriebenen Änderungen sind im GitHub-Repository Engel unter dem Branch *RenderScript* einsehbar.

D. Aufbau und Durchführung

Für das Aufzeichnen der Daten per ADB wird das Smartphone mit einem USB-Kabel an einen Laptop angeschlossen. Auf diesem wird dann das ADB-Logging gestartet. Die Trepn Profiler App wiederum speichert die Daten in eine CSV-Datei auf den Speicher des Smartphones.
Zunächst werden in der Trepn Profiler App die nötigen Einstellungen, wie beispielsweise die aufzuzeichnenden Metriken, festgelegt. Vor dem Start des Experiments wird das Smartphone vollständig geladen, alle Programme geschlossen und das WLAN deaktiviert. Sobald das mobile Gerät am Laptop angeschlossen wurde, wird mit

```
adb logcat -c
```

der logcat-Buffer geleert und anschließend das ADB-Logging gestartet. Die Trepn Profiler App kann den Arbeitsspeicher nur für das gesamte System ermitteln. Um nun den durch die TF Classify App verbrauchten Speicher zu bestimmen, wird zunächst das Logging durch Trepn Profiler gestartet und mit dem Starten der TF Classify App wenige Sekunden gewartet. In diesen ersten Momenten wird der Arbeitsspeicher des Systems „im Ruhezustand“ ermittelt und kann zur Berechnung des durch die TF Classify App verbrauchten Arbeitsspeichers verwendet werden. Nach ca. 6 Minuten wird das sog. Profiling durch Trepn Profiler und das ADB Logging gestoppt.
Diese Routine wird für beide in der Tabelle I aufgeführten Geräte mit RSTensorFlow jeweils mit Eigen- und RenderScript-Unterstützung durchgeführt.

E. Auswertung

Für eine einfachere Auswertung werden die aufgezeichneten Daten zunächst aufbereitet. Während die CSV-Dateien der Trepn Profiler App manuell vorbereitet werden, wird für die durch das ADB-Logging aufgezeichneten Daten ein Python-Skript genutzt. Die aufbereiteten Daten werden mit einem R-Skript visualisiert.

1) Matrixmultiplikation: In Abbildung 1 sind die gemittelten Zeiten in Millisekunden für die von TF Classify durchgeführten beiden Arten der Matrizenmultiplikation zu sehen. Auf dem Samsung S7 benötigte RenderScript ungefähr doppelt so lange für beide Arten von Multiplikation. Für das Samsung J5 fällt diese Spanne wieder um einiges größer aus. Bei einer Matrixmultiplikation der Form

$$R^{1\times 1024} \times R^{1024\times 1008} \rightarrow R^{1\times 1008}$$

benötigt Eigen nur etwa ein Drittel der Zeit im Vergleich zu RenderScript. Für eine Matrixmultiplikation der Form

$$R^{1\times 2048} \times R^{2048\times 1024} \rightarrow R^{1\times 1024}$$

ist RenderScript schon fast 10 Mal langsamer als Eigen. Dieses Ergebnis widerspricht teilweise den Resultaten aus Alzantot et al. (2017) bezüglich der Zeiten für die Matrixmultiplikation. Hier wurde, zumindest für das Nexus 5X, eine Performance-Steigerung festgestellt. Allerdings war RenderScript auf diesem Smartphone nach Alzantot et al. (2017) in der Lage die GPU zu nutzen. Dies würde implizieren, dass RenderScript sowohl auf dem Samsung J5 als auch auf dem Samsung S7 die GPU nicht verwenden konnte. Bei Betrachtung der CPU-Last für das J5 ist festzustellen, dass die CPU unter der Verwendung von RenderScript durchgehend fast 100% ausgelastet war. Die Eigen-Variante hingegen lastete hier die CPU im Schnitt mit 80% aus. Auf dem Samsung S7 wiederum lastet Eigen die CPU stärker aus als RenderScript. Dies könnte trotz der längeren Laufzeit auf die Nutzung der GPU durch RenderScript hindeuten, wobei dies aufgrund der fehlenden Messwerte von der GPU nicht mit Sicherheit behauptet werden kann.

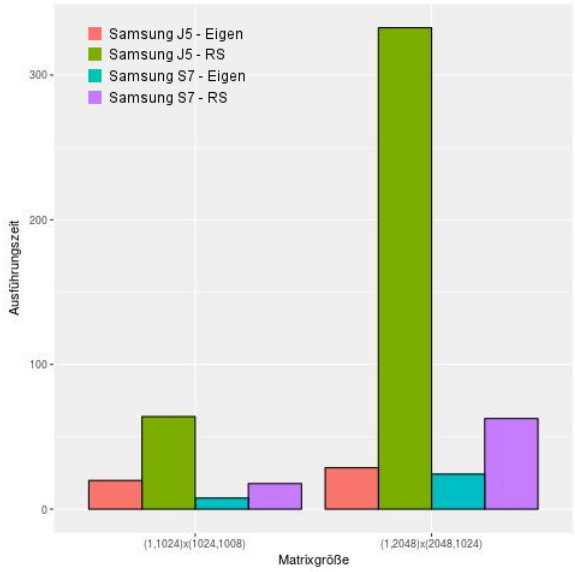

Abbildung 1. Die Ausführungszeiten in Millisekunden der von TF Classify durchgeführten Matrizenmultiplikationen.

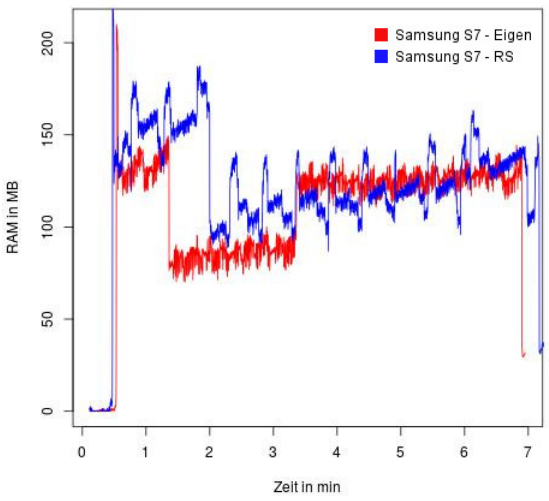

Abbildung 2. Ein Vergleich des benötigten Arbeitsspeichers von TF Classify zwischen der RenderScript- und Eigen-Implementierung.

2) Convolution: Die visualisierten Daten der Convolution-Operation unterstützen die in VI-E1 getroffene Schlussfolgerung, dass RenderScript keine GPU auf dem Samsung J5 nutzen kann. Für dieses Gerät wurden im Vergleich zu Eigen deutlich höhere Ausführungszeiten bei der Nutzung von RenderScript gemessen. Dabei steigt die Zeit allerdings nicht konstant mit der Anzahl verwendeter Filter. Dies liegt darin begründet, dass die Komplexität einer Convolution-Operation nicht nur von der Anzahl anzuwendender Filter abhängt. Die Größe des Input- und Filter-Tensors sowie die Größe des Strides und das Zero-Paddings sind hierfür ebenfalls relevante Parameter. Beim Samsung S7 hingegen hat RenderScript überraschenderweise und entgegen der Ergebnisse in Alzantot et al. (2017) die Convolution-Operationen oftmals schneller durchgeführt als Eigen. Ab 64 Filtern ist RenderScript ungefähr doppelt so schnell wie die Eigen-Bibliothek. In Alzantot et al. (2017) wurde die schlechtere Performance von RenderScript bei der Convolution-Operation mit einem Overhead bei der Reservierung von Arbeitsspeicher versucht zu begründen. Wie in Abbildung 2 zu sehen ist, wird bei RenderScript häufiger eine etwas größere Menge Speicherplatz reserviert. Da die Convolution-Operation bei RenderScript trotzdem schneller durchgeführt wurde, wird dies vermutlich keinen signifikanten Einfluss auf die Ausführungszeit haben. Eine mögliche Erklärung für den Performance-Unterschied kann auch die Nutzung der GPU nicht sein, da in Alzantot et al. (2017) RenderScript auf dem Nexus 5x diese Rechenressource nutzt und dennoch eine schlechtere Performance als Eigen erzielt. Ein Unterschied bei der Hardware zwischen dem Nexus 5X und dem Samsung S7 ist die Anzahl und Frequenz der CPU-Kerne. Bei der Convolution-Operation einzelnen Ebenen in der Tiefe des Filter-Tensors unabhängig voneinander auf den Eingabe-Tensor angewandt werden können, sind diese Operationen gut parallelisierbar. Daher liegt die Vermutung nahe, dass RenderScript diese Parallelisierung besser nutzen

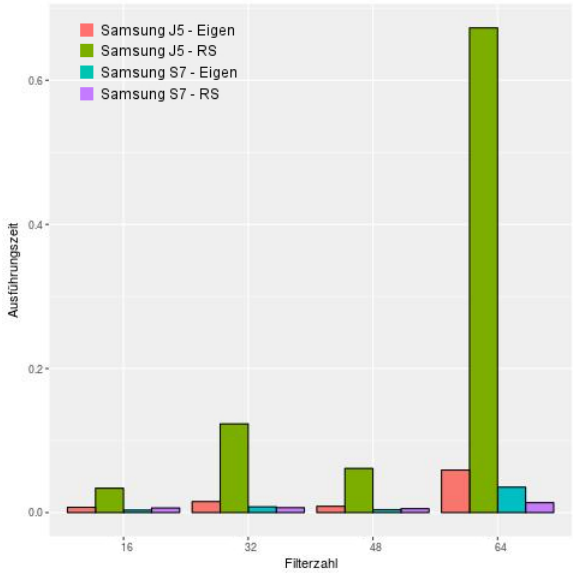

Abbildung 3. Die Ausführungszeiten in Millisekunden der von TF Classify durchgeführten Convolution-Operation für 16 bis 64 anzuwendende Filter.

kann als Eigen.

3) Inference-Pfad: Bei der Gesamtdauer eines einzelnen Inference-Pfades fallen für beide mobile Geräte deutliche Unterschiede zwischen der Eigen- und RenderScript-Variante auf. Beim Samsung S7 dauert ein Pfad mit der Eigen-Implementierung im Schnitt 0,4s. Die Variante mit RenderScript hingegen führt einen Pfad zwischen 0,8s und knapp 1,2s aus und benötigt damit mindestens zweimal so lange wie Eigen. Für das Samsung J5 fällt dieser Unterschied wesentlich ausgeprägter. Während Eigen einen Pfad im Durchschnitt mit 1s abarbeitet, ist die RenderScript-Variante mit ca. 15s um das 15-fache langsamer. Dies wirkte sich auch auf die Anzahl der durchlaufenen Pfade während des Experiments aus. Beim S7 durchlief die Variante mit Eigen knapp 800 Pfad-Iterationen

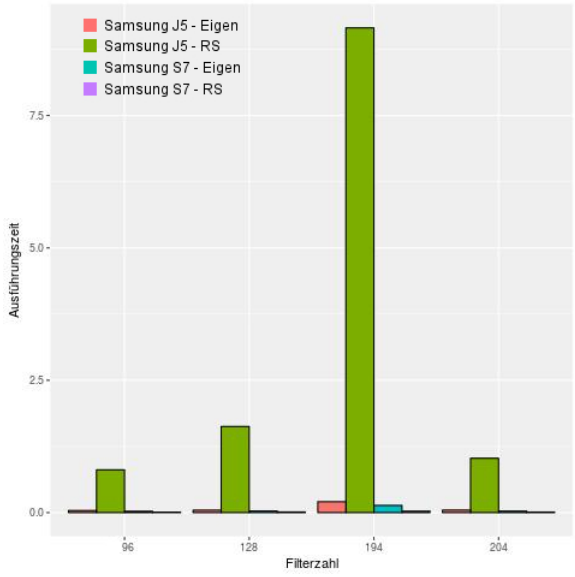

Abbildung 4. Die Ausführungszeiten in Millisekunden der von TF Classify durchgeführten Convolution-Operation für 96 bis 204 anzuwendende Filter.

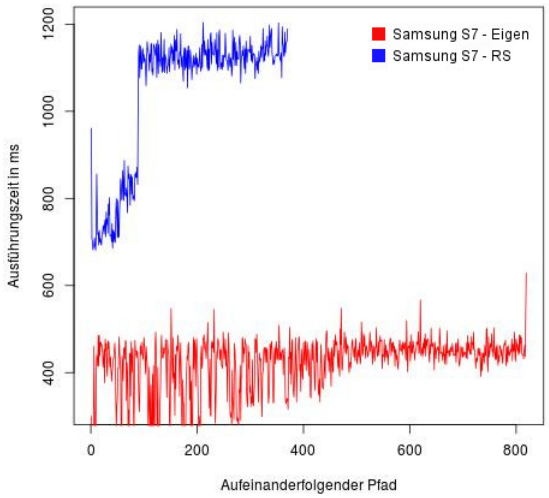

Abbildung 5. Die Anzahl der durchlaufenen Pfade ist bei der RenderScript-Variante um ca. die Hälfte geringer als bei der Berechnung mit Eigen.

und mit RenderScript nur knapp die Hälfte. Dieser Sachverhalt wird in Abbildung 5 besonders deutlich. Demzufolge muss die Verwendung von RenderScript auch indirekte, negative Auswirkungen auf den Inference-Pfad haben.

VII. Optimierung der Conv2D-Operation

In Chellapilla et al. (2006) sowie in Rajbhandari et al. (2017) wird die Technik des Unfoldings für die Optimierung von Convolutional Networks beschrieben. Beim Unfolding wird sowohl der Eingabe-Tensor T_E als auch der Filter-Tensor T_F in eine Matrix M umgewandelt, sodass nur noch eine einfache Matrixmultiplikation der Form

$$M_F \cdot M_E^T$$

durchgeführt werden muss. Das Ergebnis ist wiederum eine Matrix, deren Zeilen die Tiefen-Ebene des Output-Tensors entspricht.
Wie in Chellapilla et al. (2006) fällt dementsprechend als Overhead die Umwandlung der beiden Tensoren in die Matrizen und die Umformatierung der Ergebnis-Matrix in einen Tensor an. Ebenso wird für die separaten Matrizen zusätzlicher Arbeitsspeicher benötigt. Die Reservierung des RAM ist, wie bei den Experimenten VI deutlich wurde, hinsichtlich einer möglichen Performance-Einbuße eher gering einzuschätzen. Allerdings würde die doppelte Menge an Arbeitsspeicher benötigt, was bei sehr großen Tensoren durchaus problematisch auf mobilen Geräten werden kann. Da die Größe der Tensoren von TF Classify allerdings eher klein sind, ist die Speichermenge für diesen speziellen Fall kein Problem.
Die Umsetzung des Unfoldings könnte sowohl für die RenderScript- als auch die Eigen-Implementierung eine Performance-Steigerung bewirken. Hierfür muss der Eingabe- und Filter-Tensor zunächst in die benötigte Form gebracht werden. Im nächsten Schritt wird ein Aufruf der Matrixmultiplikation mit den zu Matrizen geformten Tensoren ausgeführt. Das Ergebnis muss nun nur noch in die geforderte Tensor-Form gebracht werden.

VIII. Fazit und Ausblick

In diesem Paper wurde RSTensorFlow, eine angepasste Variante der Machine Learning Bibliothek TensorFlow, vorgestellt und näher untersucht. Ebenso wurden die in Alzantot et al. (2017) durchgeführten Experimente für die Matrixmultiplikation und die Convolution-Operation nachgestellt und weitere Metriken für die eigenen Experimente definiert. Hierfür wurden Quellcode-Anpassungen im Kernel von RSTensorFlow umgesetzt.
Die von den durchgeführten Experimenten erhobenen Daten widersprechen teilweise den in Alzantot et al. (2017) präsentierten Ergebnissen. Während im durchgeführten Experiment für RenderScript bei der Matrixmultiplikation eine schlechtere Performance gemessen wurde, wurde in Alzantot et al. (2017) eine Steigerung der Performance angegeben. Bei Convolution-Operation steigen die Ausführungszeiten auf dem Samsung J5, ähnlich zu denen Werten in Alzantot et al. (2017), bei erhöhen der Filterzahl. Das Samsung S7 hingegen erzielt mit RenderScript eine fast doppelt so schnelle Bearbeitung der Convolution-Operation. Der Grund für diese Performance-Steigerung kann nicht an der Nutzung der GPU durch RenderScript liegen, da in Alzantot et al. (2017) vom Nexus 5X die GPU von RenderScript genutzt wurde und dennoch eine Verschlechterung der Performance zu messen war. Auch unter Verwendung der anderen Metriken, wie beispielsweise die CPU-Last, lassen sich hierzu keine sicheren Aussagen treffen. Die erzielten Ergebnisse decken sich somit nicht komplett mit den Resultaten aus Alzantot et al. (2017). Dabei kann der exakte Grund für die Unterschiede nicht festgestellt werden, sodass lediglich Vermutungen geäußert werden können. Diese Arbeit hat jedoch gezeigt, dass sich RenderScript auf verschiedenen mobilen Geräten zu sehr unterschiedlich auswirkt. Daher ist RenderScript für den breiten Einsatz noch nicht geeignet, hat

aber auch durchaus Potential für den Einsatz im Bereich Deep Learning auf mobilen Endgeräten.

LITERATUR

Eigen. URL http://eigen.tuxfamily.org/index.php?title=Main_Page. Zuletzt besucht: 06.12.2017.

OpenCL. URL https://www.khronos.org/opencl/. Zuletzt besucht: 06.12.2017.

RenderScript. URL https://developer.android.com/guide/topics/renderscript/compute.html. Zuletzt besucht: 06.12.2017.

TensorFlow, a. URL https://tensorflow.org. Zuletzt besucht: 06.12.2017.

TensorFlow - GitHub Repository, b. URL https://github.com/tensorflow/tensorflow. Zuletzt besucht: 06.12.2017.

Vulkan. URL https://www.khronos.org/vulkan/. Zuletzt besucht: 06.12.2017.

M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava. RSTensorFlow: Gpu enabled tensorflow for deep learning on commodity android devices. 1st International Workshop on Deep Learning for Mobile Systems and Applications, 2017. URL https://nesl.github.io/RSTensorFlow. Software available from nesl.github.io.

K. Chellapilla, S. Puri, and P. Simard. High Performance convolutional neural networks for document processing. Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule (France), 2006. URL http://www.suvisoft.com.

M. Engel. RSTensorFlow - GitHub Repository. URL https://github.com/engelmi/tensorflow. Zuletzt besucht: 06.12.2017.

A. Graves, A. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013. URL http://arxiv.org/abs/1303.5778.

L. N. Huynh, R. K. Balan, and Y. Lee. DeepSense: A gpu-based deep convolutional neural network framework on commodity mobile devices. 2016 Workshop on Wearable Systems and Applications, 2016.

L. N. Huynh, Y. Lee, and R. K. Balan. DeepMon: Mobile gpu-based deep learning framework for continuous vision applications. 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017.

A. Karpathy. Convolutional neural networks (cnns / convnets). URL http://cs231n.github.io/convolutional-networks/.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 1201–1205, 2016.

F.-F. Li, J. Johnson, and S. Yeung. , 2017. URL http://cs231n.github.io/convolutional-networks/. Zuletzt besucht: 16.11.2017.

S. Rajbhandari, Y. He, O. Ruwase, M. Carbin, and T. Chilimbi. Optimizing cnns on multicores for scalability, performance and goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 267–280, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037745. URL http://doi.acm.org/10.1145/3037697.3037745.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.