RSTensorFlow: Analyse und Optimierung (vorläufiger Titel)

Michael Engel

I. Abstract

- Zusammenfassender Abstract

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.

Ab hier falls Optimierung von Conv2D

Dazu wird eine Analyse der RSTensorFlow-Implementierung (todo: github-link) durchgeführt, um ein Verständnis über die Architektur und die bisherige Implementierung der Conv2d-Operation zu gewinnen. Auf Basis dieser Analyse wird eine Optimierung der Conv2D-Operation durch sogenanntes Unfolding versucht zu erreichen. In Chellapilla et al. (2006) sowie in Rajbhandari et al. (2017) werden Convolutional Networks optimiert. Hierfür wägen beide Arbeiten das Potential des Unfoldings und erläutern die Funktionsweise.

III. Related Work

In Alzantot et al. (2017) wird die derzeit noch eher mangelhafte Unterstützung für tiefe neuronale Netze auf mobilen 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 diverse Experimente, welche die angepasste Version RSTensorFlow mit dem Original hinsichtlich der Performance vergleichen.

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 Karpathy 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. Daher 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.

A. Analyse des Quellcodes

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 Activities, welche auf das Inference Interface zugreifen, umgesetzt werden.
Für die Berechnungen der Matrixmultiplikation und der Convolution-Operation wird von TensorFlow die Eigen-Bibliothek **?** verwendet. Damit die 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 Authoren 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 in den Quelldateien

- conv_ops.cc und
- matmul_op.cc

nötig. Statt des Aufrufs zur Berechnung durch die Eigen-Bibliothek wird die jeweilige Funktion des RenderScript-Wrappers aufgerufen. An den einzelnen Android-Activities sind aufgrund der Kapselung keinerlei weitere Änderungen nötig. Somit nutzt beispielsweise die TF Classify App, welche in *ClassifierAcitivity.java* implementiert ist, automatisch die RenderScript-Unterstützung.

B. Der Build-Prozess

+ tools - bazel, android sdk, ndk + fällt in bazel script auf dass .so Ordner nur eingebunden wird, aber nicht die RenderScript OPS kompiliert werden - auf <website> wird die modifizierte variante von libtensorflow.rs sowie die normale .so zur verfügung gestellt - für eigene kompilierung (z.B. bei anpassungen in tf oder auch bei den renderscript operationen) fehlt die librs.mScriptConv.so (kompilierung durch eigenständiges projekt, obwohl quelldateien in repo eingebunden) + nachdem technische aspekte geklärt sind, wird versucht die experimente aus Alzantot et al. (2017) in VI-A zu rekonstruieren + RSRuntime so's werden nur eingebunden, nicht kompiliert - hier fehlt die librs.mScriptConv.so -> musste selbst kompiliert und eingebunden werden

+ das rstf projekt bietet auf der homepage die libtensorflowInference.so an - mit und ohne rs - für anpassungen muss allerdings selbst die libtensorflowInference.so kompiliert werden + RSRuntime so's durch extra projekt kompilierbar - librs.mScriptConv.so usw

VI. Experimente

+ Hardware-Tabelle (inkl. die Hardware aus Alzantot et al. (2017))

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 der gewählten Modelle für die Experimente.

+ was wird alles genutzt? - trepn - adb

A. Rekonstruktion

+ grafiken für matmul und conv2d - wi- Da keine Filteranzahl mit 1, 3 oder xxx gefunden wurde, und leider keine genaue beschreibung vorhanden ist, wie die Grafik zustande kommt, ist davon auszugehen, dass ein separates programm verwendet wurde (diese Hypothese wird gestützt durch fehlendes loggen der anzahl dimensionen im modifizierten tf quellcode) - Anzahl Filter ist Hyperparameter - K == Anzahl der Filter == Output-Dimension3 (Breite und Höhe durch stride definiert) e kommen die auf die jeweiligen werte der x-Achse? + conv2d: Anzahl Filter - wie in IV-B bei conv erläutert - Dout == anzahl filter - conv_ops.cc - Da keine Filteranzahl mit 1, 3 oder xxx gefunden wurde, und leider keine genaue beschreibung vorhanden ist, wie die Grafik zustande kommt, ist davon auszugehen, dass ein separates programm verwendet wurde (diese Hypothese wird gestützt durch fehlendes loggen der anzahl dimensionen im modifizierten tf quellcode) - Anzahl Filter ist Hyperparameter - K == Anzahl der Filter == Output-Dimension3 (Breite und Höhe durch stride definiert)

B. Metriken

Mögliche weitere Metriken wie in Kapitel 7 von Huynh et al. (2017) beschrieben ggf. verwenden

- Ist Metrik für dieses Projekt sinnvoll?
- Ist Metrik für dieses Projekt umsetzbar (Aufwand etc.)?

Verwendete Metrik in Alzantot et al. (2017): Ausführungszeit + im Code von tensorflow integriert + gibt auskunft über die Zeit einer einzelnen matmul-/conv2d operation und eines kompletten pfades + über adb logcat -s TF_ANDROID_LOG können diese in eine Datei gelenkt werden

Metrik CPU Load + über trepn profiler + gibt auskunft über die auslastung der ressource

Metrik GPU Load + über trepn profiler + gibt auskunft über die auslastung der ressource

Metrik RAM (Memory Space): + über trepn profiler + nicht straight forward, da von Trepn Profiler RAM des kompletten Systems gemessen wird + daher System im Ruhezustand profilen (inkl. laufender Trepn app) und (durchschnittlichen) RAM ermitteln und vom später gemessenen abziehen

C. Anpassungen von TensorFlow

+ Link zu meinem Repo + Die Trepn Profiler öut-of-the-boxäber die logausgaben müssen aufbereitet werden, da die vorhandenen logausgaben unzureichend sind + adb logging da nur ëin Stream"verfügbar - alles mit adb logcat TF_ANDROID_LOG in eine CSV-Datei Format: <Feld1>|<Feld n>|... Dabei ist Feld1 der Index, um was für eine Operation es sich handelt (da von classify sowohl conv2d als auch matmul ausgeführt wird) Für matmul: + die matrizengröße Für conv2d: + anzahl filter (== outdepth) + stride] + padding - Berechnung der Anzahl der matmul operationen -> hat einfluss auf ausführungszeit? ggf. mit unfolding doch verbesserbar? (für Ausblick dann) + filtergröße] + inputgröße]

D. Aufbau des Experiments

+ Software + Mit was werden die Daten erzeugt? - trepn profiler (app) -> Einstellungen definieren und speichern! - log ausgaben (direkt von tensorflow demo) -> siehe VI-C - über android system -> adb dumpsys xxx + Hardware + ggf. abbildung? Beschreibung des Experiment-Aufbaus

- Wahl der Android-Geräte (Samsung J5 und ggf. weitere), deren Android-Version und Hardware)
- Wie werden die gewählten Metriken gemessen?

E. Durchführung

+ trepn profiler: profile system - profiling erst im standard modus um die normale menge an memory zu erhalten - dann app starten

F. Auswertung

+ was kamen für ergebnisse heraus? + vergleich zu Alzantot et al. (2017) - deutet auf eine generelle verschlechterung mit rs hin - direkte vergleichbarkeit aber leider nicht gegeben, da die genaue erhebung der daten unbekannt

VII. Optional: Optimierung der Conv2D-Operation

+ unfolding ist mögliche optimierung - literatur: - statt viele kleine, elementweise Operationen -> nur noch 1 große matrix-multiplikation - theoretisch diskutieren (größe der matrizen) - 1 große matmul, könnte die bereits bestehende matmul impl nutzen -> output matrix muss als tensor aufbereitet werden - wie könnte es umgesetzt werden?

- Kurze Beschreibung der Convolution-Operation (wie in Li et al. (2017))
- Ist-Analyse des Quellcodes der Conv2D-Operation
- Unfolding (Chellapilla et al. (2006)) als mögliche Optimierung von Conv2D für RSTensorFlow
- Implementierung der (möglichen) Optimierung

VIII. Fazit und Ausblick

Einige Probleme: + erstmal zum laufen bekommn (von rs-tensorflow bereitgestellter mod. build funzt nicht, build prozess etc.) + rekonstruktion der experimente Hier kann aus dem referenzpaper bzw. dessen website zitiert oder sich darauf bezogen werden: ist schweres zeug, rs nativ in tensorflow zu integrieren

Schlussfolgerung: + nur weil es die gpu ist bzw. eine zusätzliche ressource, muss es nicht unbedingt schneller sein + ggf derzeit einfach nicht ausgereift -> standards! + oder renderscript einfach ungeeignet

- Fazit
  - Erzielt RSTensorFlow ähnliche Ergebnisse bei den hier getesteten Geräten im Vergleich zu den Nexus-Geräten, welche in Alzantot et al. (2017) verwendet wurden?
  - Konnte eine Verbesserung von Conv2D erzielt werden?
- Ausblick (oder Future Work)
  - TensorFlow Lite
  - OpenCL statt RenderScript

IX. Zukünftige Arbeiten

+ future work: unfolding umzusetzen (wie auswertung zeigt: sehr viele matmuls bei conv und damit ist nur noch 1 nötig -> performance-memory-tradeoff? eher unwahrscheinlich, da für conv2d und rs eh die tensoren kopiert werden und bei auswertung kaum veränderungen bei dem genutzten speicher zu sehen waren) und somit

Literatur

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.

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.stanford.edu. 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.