FlexNet:
flexible neuronale Netzwerke für kontextbezogene Bildklassifizierung

Ein Projekt von Malik Pätzold

Kurzfassung

Oftmals sieht man sich beim Einsatz Künstlicher Intelligenz mit einigen Problemen konfrontiert. Wir wollten z.B. mithilfe von neuronalen Netzwerken Bilder von Plastikobjekten klassifizieren. Jedoch erkannte ich schnell ein Problem: Gerade bei einem umfangreicheren (bzw. „Tieferen“) Modell und einem großen Datensatz ist das Trainieren dieser Netzwerke recht Zeit- und Ressourcenaufwendig. Hinzu kommt, dass sich unser Datensatz häufig verändert hat. Dies ist natürlich ein großes Hindernis bei der Entwicklung der eigentlichen Anwendungen.

Aus diesem Grund haben wir uns entschieden, einen neuen Aufbau von neuronalen Netzwerken zu erforschen, welcher Änderungen am Datensatz zulässt, ohne die Notwendigkeit das gesamte Netzwerk nochmal zu trainieren. Dieses neue Modell mit den Namen FlexNet wird an einem eigenen Datensatz evaluiert und mit klassischen Architekturen verglichen.

Inhaltsverzeichnis

Einleitung

Ideenfindung

Seit mehreren Jahren beschäftige ich mich bereits mit Recyclingmöglichkeiten für Plastik. Ca. Anfang letzten Jahres (2020) habe ich versucht mithilfe Künstlicher Intelligenz einzelne Objekte und ihren Plastiktypen zu erkennen. Ein Bild des Gegenstandes wird einem Deep Learning Model gegeben, welches die festgelegte ID des Gegenstandes ermittelt. Mithilfe einer Datenbank kann dann der Plastiktyp zurückgegeben werden. Das Ganze hat technisch gesehen auch gut funktioniert. Jedoch erkannte ich schnell ein Problem: Gerade bei einem umfangreicheren (bzw. „Tieferen“) Modell und einem großen Datensatz ist das Trainieren dieser Netzwerke recht Zeit- und Ressourcenaufwendig. Hinzu kommt, dass sich unser Datensatz häufig verändert hat. So wurden oftmals neue Klassen hinzugefügt oder bereits existierende haben sich verändert, weil z.B. sich Aufkleber von Plastikflaschen verändert haben oder andere Produkte nun ein neues Design haben, usw. Dies führet dazu, dass auch bei kleinen Veränderungen in einem bestimmten Teil des Datensets das gesamte Modell neu trainiert werden musste, was oftmals ein stundenlanger Prozess ist. Da wir dadurch ab einen gewissen Punkt in unserer eigentlichen Arbeit, nämlich dem Entwickeln des Gesamtprozesses zum Plastikrecycling, erheblich gestört wurden, haben wir uns entschieden stattdessen erstmal eine Lösung für unser Problem zu finden.

Ansatz

Da (wie oben beschrieben) es sich in den meisten Fällen nur um kleine Teile des Datensatzes handelt die verändert werden, sollte man in der Theorie ebenfalls auch nur die Teile des neuronalen Netzes verändern, die direkt betroffen sind. Jedoch ist das in der Praxis nur schwer möglich, da das Netzwerk sehr verworren ist und man als außenstehender unmöglich genau sagen kann, welcher Teil des Netzwerkes wofür verantwortlich ist und inwieweit dieser verändert werden muss. Zwar gibt es immer wieder Untersuchungen in diesen Bereich, u.a. wurden Konvolutionale Neuronale Netzwerke (Convolutional Neural Networks, CNN), bzw. die Konvolutionalen Filter recht genau untersucht [1][2s], aber die Fully-Connected Layer (Vollständig verbundenen Schichten, also das eigentliche neuronale Netzwerk) sind recht schwierig zu entschlüsseln.

Glücklicherweise kommt uns ein Faktor entgegen: Bei genauerer Betrachtung unseres Datensatzes ist uns aufgefallen das sich dieser recht gut in Kategorien unterteilen lässt. So lässt sich jedes Bild einer generellen Kategorie, wie z.B. Flasche, Tüte, oder Verpackung zuordnen. Angenommen unser Input-Bild zeigt eine Colaflasche. Es würde dann erst als Teil der Kategorie Flasche erkannt werden, und dann der genauen Klasse Colaflasche zugeordnet werden.

Wenn man nun Änderungen in der Kategorie Flasche vornehmen möchte, muss nur der Teil des Netzwerkes welcher innerhalb der Kategorie Flaschen die Klassen bestimmt verändert werden. Bei hinzufügen einer neuen Kategorie muss nur der „obere“ Teil des Netzwerkes, also dort wo der Input einem Sub-Netzwerk zugeteilt wird, neu trainiert werden. Dieser Aufbau eines neuronalen Netzwerkes wird von uns als FlexNet (für flexibles neuronales Netzwerk) bezeichnet.

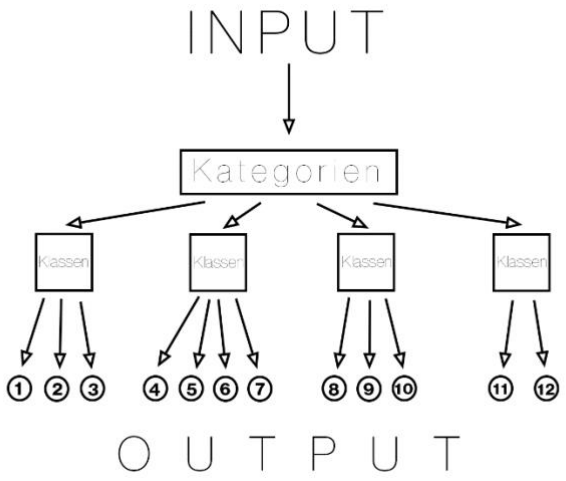


Abbildung 1: Beispielhafter Aufbau eines FlexNets mit vier Kategorien und 12 Klassen.

Vorgehensweise, Materialien und Methode

Überblick

Unser Hauptziel war es, ein funktionierendes FlexNet zu implementieren, testen und vergleichen. Dafür testeten wir zunächst ein paar Variationen im Aufbau des Netzwerks und des Datenflusses innerhalb des Netzwerkes, um die bestmöglichen Ergebnisse zu erzielen. Danach wird das FlexNet an einem kleinen Teil unseres Datensatzes (dazu gleich mehr) trainiert. Dann vergrößern wir den Datensatz und untersuchen, wie sich die Leistung des Models verändert.

Der Datensatz

Unserer Datensatz besteht aus insgesamt 42500 Farbbildern mit einer Abmessung von 512 Pixel x 512 Pixel. Es gibt 17 Klassen, für jede Klasse 2000 Trainings- und 500 Testbilder. Der Datensatz wurde mithilfe eines 3D-Modellierungsverfahren erstellt. Wir haben uns zu der Erstellung unseren eigenen Datensatzes entschieden, da wir keinen passenden, bestehenden Datensatz finden konnten.

Unser Datensatz bestehet aus vier Kategorien: Gummiente, Schweinekopf, Lego Bausteine und Chips. Jede Kategorie enthält mehrere Klassen, welche unterschiedliche Variationen der Kategorie enthalten. Diese Variationen können sich durch Farbe, Muster und Form unterscheiden.

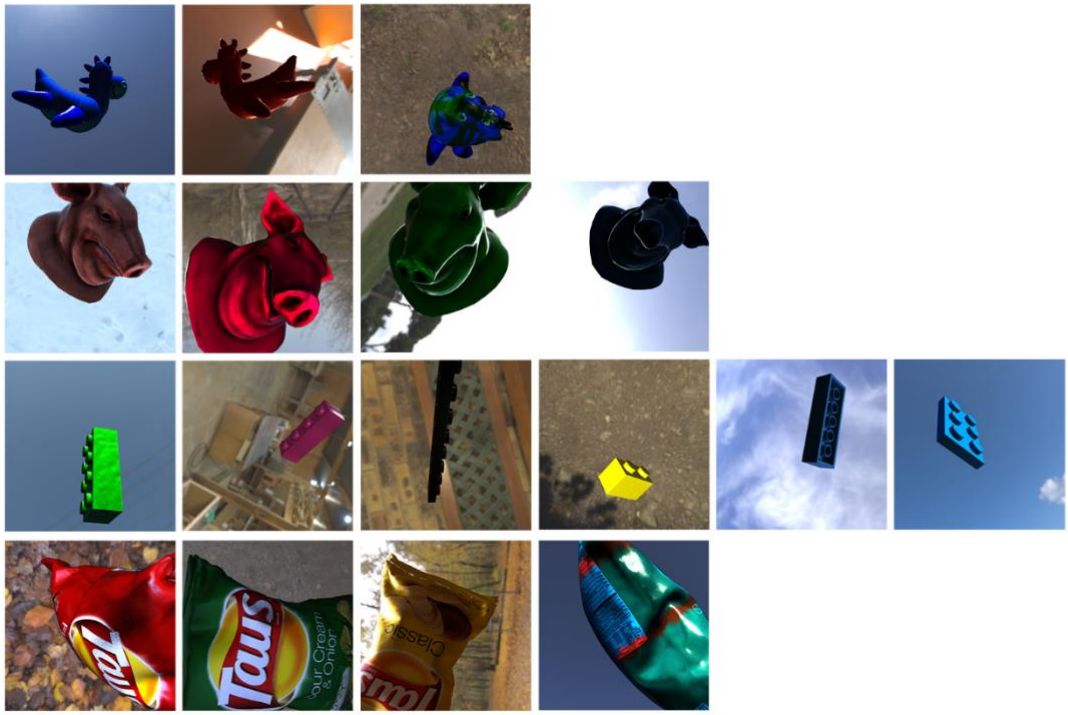


Abbildung 2: Beispielbilder für jeder Klasse der vier Kategorien.

Evaluierungen von Modellen & Submodellen

Alle Modelle und Submodelle werden am oben beschriebenen Datensatz getestet und ihre Leistung evaluiert. Dazu werden folgende Werte gemessen: Die Genauigkeit des Modells am Trainingsteil des Datensatzes, die Genauigkeit des Testteils des Datensatzes, gebrauchte Trainingszeit und benötigte Zeit, welche zum Ausführen des Modells an einem Bild benötigt wird.

Messung der Trainings- und Ausführzeit

Für Zeitmessungen wird die jeweilige Start- und Endzeit eines Prozesses vom Programm geloggt, aus der Differenz der beiden Werte wird die vom Programm benötigte Zeit berechnet. Alle unsere Programme zum Trainieren der Modelle sind gleich aufgebaut: Zuerst werden die benötigten

Trainingsdaten geladen und wenn noch nötig als Tensor umgewandelt. Als nächstes wird das Modell definiert & initialisiert. Nun startet der Trainingsloop, in welchem der eigentliche Optimierungsvorgang stattfindet. Während des Trainingsloop wird außerdem die aktuelle Version am Testdatensatz und einem Teil (in der Regel 10%) des Trainingsdatensatz evaluiert. Die Version mit der höchsten Testgenauigkeit wird gesichert. Zur gemessenen Trainingszeit gehört nur die Dauer, die es braucht, um den Trainingsloop auszuführen, das laden der Daten zählt nicht dazu.

Bei der Feststellung der Ausführzeit zählt ebenfalls nicht das laden von Daten, sondern ausschließlich der Dauer, die es braucht, um das Modell an einem Input-Bild auszuführen.

Messung der Trainings- und Testgenauigkeit

Alle Modelle werden während und nach dem Training auf ihre Genauigkeit geprüft. Hierfür gibt es zwei verschiedene Werte: Die Trainingsgenauigkeit und die Testgenauigkeit. Der Datensatz ist ebenfalls in zwei Teile unterteilt, nämlich den Test- und den Trainingsteil. Zur Bestimmung der Trainingsgenauigkeit durchlaufen die Bilder des Trainingsdatensatzes das zu testende Modell. Aus der Anzahl der richtig klassifizierten Bilder wird die Trainingsgenauigkeit (in Prozent) berechnet. Mithilfe des gleichen Prinzips und benutzten des Testdatensatzes wird die Testgenauigkeit berechnet. Doch warum ist die Unterteilung des Datensatzes in diese zwei Gruppen notwendig? Wie der Name nahelegt, wird der Trainingsteil des Datensatzes zum trainieren des neuronalen Netzwerkes benutzt. Das Netzwerk soll die generellen visuellen Eigenschaften der auf den Bildern zu sehenden Objekten erlernt. Jedoch kann es auch passieren, dass sich das Netzwerk stattdessen die spezifischen Bilder einfach „merkt“ und nicht generelle Muster erkennt. In diesem Fall kann eine hohe Trainingsgenauigkeit erreicht werden, welche aber nicht die tatsächliche Genauigkeit des Modells entspricht. Die Bilder aus dem Testdatensatz wurden vom Modell aber noch nie „gesehen“ und konnten daher auch nicht eingeprägt werden. Aus diesem Grund entspricht die Testgenauigkeit der tatsächlichen Genauigkeit des Modells.

Hardware

Alle Programme wurden auf einem uns zur Verfügung stehenden Rechner ausgeführt. Hier sind dessen Gerätespezifikationen:

CPU: AMD Ryzen 9 3900XT 12-Core

RAM: 64 GB

GPU: NVIDIA GeForce RTX 3090 (24 GB GPU-Speicher)

Software

Alle Programme wurden mit Python und verschiedenen Bibliotheken und Frameworks programmiert. Die genauen Spezifikationen sind:

Python 3.8.6

PyTorch 1.8.0 (CUDA 11.0)

OpenCV 4.4.0.44

NumPy 1.18.5

Matplotlib 3.3.2

Reproduzierbarkeit

Da an verschiedenen Stellen innerhalb des PyTorch Frameworks (und auch der NumPy Bibliothek) mit zufällig generierten Zahlen gearbeitet wird, sind die Ergebnisse der Programme standartmäßig nicht reproduzierbar. Aus diesem Grund werden alle Zufallsparameter am Anfang der Programme festgelegt. [3]

PyTorch benutzt die cuDNN-Bibliothek, welche Deep Learning Modelle unterstützt, die auf CUDA fähigen Grafikkarten ausgeführt werden. Der cuDNN-Bibliothek stehen dafür verschiedene Algorithmen zur Verfügung, welche durch eine Benchmarking-Funktion ausgewählt werden. Jedoch ist diese Auswahl standartmäßig nichtdeterministisch. Durch deaktivieren dieser Funktion wird automatisch deterministisch ein Algorithmus ausgewählt. Dies bringt zwar Leistungseinbußen mit sich, welche aber durch benutzten von Grafikkarten für den Trainingsprozess wieder ausgeglichen werden. [3]

Durch die oben beschriebenen Maßnahmen sind die Leistungen der Modelle weitestgehend reproduzierbar, jedoch kann es bei Benutzung von unterschiedlicher Hardware oder Softwareversionen zu unterschieden kommen. Wir konnten während unserer Arbeiten nur selten Abweichungen beobachten, und wenn waren diese niemals größer als ein Prozent vom vorher gemessenen Wert.

Aufbau & Evaluierung der Modelle

Aufbau und Funktionsweise im Detail

Wie in Abbildung 1 gezeigt, ist der erste Schritt die Zuordnung des Inputs zu einer Kategorie. Hierzu hatten wir zwei Ansätze: Einmal ein klassisches CNN (Sub-) Modell, welches sich direkt für eine Kategorie entscheidet. Dies würde den Entscheidungsprozess zwar vereinfachen, jedoch muss das Modell beim Hinzufügen einer neuen Kategorie neu trainiert werden. Der zweite Ansatz ist etwas komplizierter: Für jede Kategorie gibt es ein Submodell, welches zwischen der spezifischen Kategorie und allen anderen Kategorien entscheiden soll. Für jede Kategorie gibt es ein solches Submodell, und die Ausgaben aller Submodelle werden in ein weiteres Zwischenmodell weitergeleitet. Dieses Zwischenmodell bestimmt die finale Kategorie-Zuweisung. Da unser Fokus auf Flexibilität und einer geringen Trainingszeit liegt, haben wir uns für die den zweiten Ansatz entschieden.

Die Submodelle für jede Kategorie bestehen in der Regel aus zwei Konvolutionalen Schichten mit Maxpool Layer, und drei Fully-Conntected Layer (oder Lineare Schichten nach PyTorch Terminologie). Bei Bedarf wurde auch noch eine dritte Konvolutionale Schicht verwendet. Die Ausgaben der Modelle bestehen aus einem Boolean Wert (als Zahl aber, spricht entweder 1 oder 0) und den beiden „rohen“ Zahlen, welche von dem Modell zurückgegeben werden. Für jedes Bild werden die Ausgaben der Modelle in eine Python Liste als Zwischenergebnis zusammengeführt, um sie an das Zwischenmodell weiter zu geben. Hier sind zwei Beispiele eines solchen Zwischenergebnisses bei einem Modell mit vier Kategorien:

[1, 0, 0, 0, -28.0425, 25.4559, 15.0320, -16.0260, 1.8950, -2.1078, 19.3685, -18.9028]

[1, 1, 0, 1, -0.4629, 0.2538, -2.7483, 2.7536, 4.4644, -4.7921, -7.8404, 6.9085]

Beim ersten Beispiel ist die korrekte Kategorie 0 (also die erste, nämlich Gummiente) und beim zweiten 3 (also die vierte, nämlich Chips). Auffällig ist das bei den neueren Kategorien mehr Submodelle eine positive Rückmeldung geben. Der Grund dafür ist, dass die älteren Modelle nicht an den neuen Teilen des Datensatzes trainiert wurden. Somit wird es immer schwieriger die Korrekte Kategorie korrekt hervorzusagen und die Genauigkeit sinkt.

Die Zwischenergebnisse werden von einem kleinen Modell Zwischenmodell verarbeitet, um eine Kategorie auszuwählen. Das Zwischenmodell besteht aus drei Linearen Schichten und gibt nur die vorhergesagte Kategorie zurück.

Sobald eine Kategorie vorhergesagt wurde, wird das entsprechende Klassen Submodell ausgeführt, welche die genaue Klasse bestimmt. Dieses benutzt wieder das Original Input-Bild und steht in keiner

Weise mit den vorherigen Modellen oder Ergebnissen in Verbindung. Das Klassen Sub-Modell besteht aus zwei Konvolutionalen Filterschichten mit jeweils einer Maxpooling Schicht. Danach folgen drei Fully-Connected Layer.

Vergleich mit anderen Modellen

Das FlexNet wird an folgenden Stufen gegen ResNet-50, spricht ein etablierte Modelle getestet: Einmal wenn das FlexNet an nur den 2 Grundkategorien trainiert ist und sobald eine Kategorie hinzugefügt wurde.

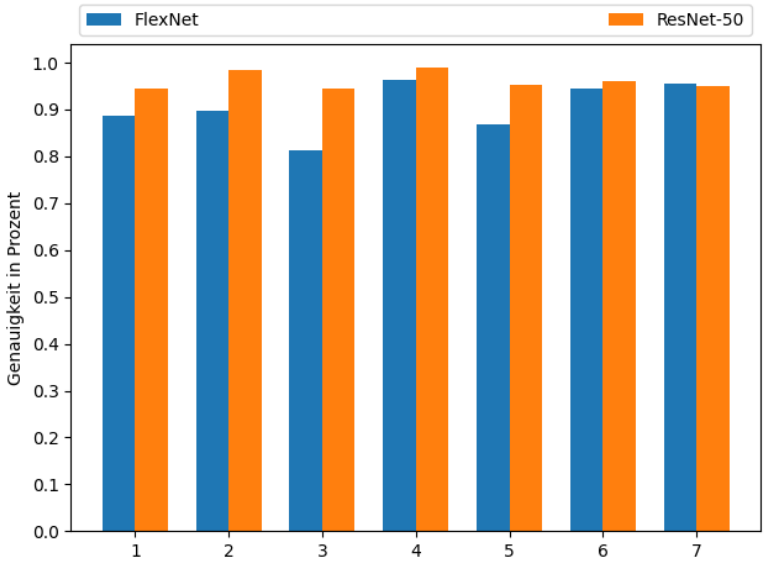
Wie bereits beschrieben werden die Modelle sowohl am Test- sowie Trainingsteil des Datensatzes getestet. Des Weiteren wird die Genauigkeit pro Kategorie und Klasse ermittelt. In manchen Fällen werden auch speziellere Analysen vorgenommen. Ein weiterer Faktor ist die gebrauchte Zeit, welche zum Ausführen und Trainieren der Modelle gebraucht wurde. Solange es jedoch keine gravierenden Zeitunterschiede (wie z.B. mehrere Sekunden) beim Ausführen der Modelle gibt, ist hauptsächlich die Trainingsdauer der Modelle ausschlaggebend. Die Feststellung der Trainingsdauer ist bei einem ResNet recht einfach zu ermitteln, jedoch gilt dies nicht für ein FlexNet, da dieses aus mehreren einzelnen Komponenten besteht. Daher werden die Trainingszeiten von allen Komponenten, welche entweder neu oder wieder trainiert werden mussten, zusammengerechnet.

Evaluierung des FlexNet

Hier die gemessenen Daten für die Leistung der Modelle bei mit nur den beiden Grundlagenkategorien Gummiente und Schweinekopf:

	Genauigkeit (Training)	Genauigkeit (Test)	Trainingszeit	Ausführzeit
FlexNet	99,2%	90,4%	1h 2m 43s	9ms
ResNet-50	100%	95,4%	1h 51m 28s	12ms

Anfänglich sind sich die Werte für beide Modelle noch recht ähnlich. Sie sind beim Trainingsteil des Datensatzes fast immer korrekt, beim Testteil erreichen sie eine Genauigkeit von über 90%. Das ResNet hat eine um fünf Prozentpunkte höhere Testgenauigkeit, welche zu kosten einer um 49 Minuten (oder 45%) längeren Trainingszeit kommt. Das FlexNet kann ein Bild durchschnittlich 3 Millisekunden schneller klassifizieren. Den kompletten Datensatz bestehend aus 42500 Bildern zu klassifizieren geht mit dem FlexNet also zwei Minuten und siebeneinhalb Sekunden schneller.

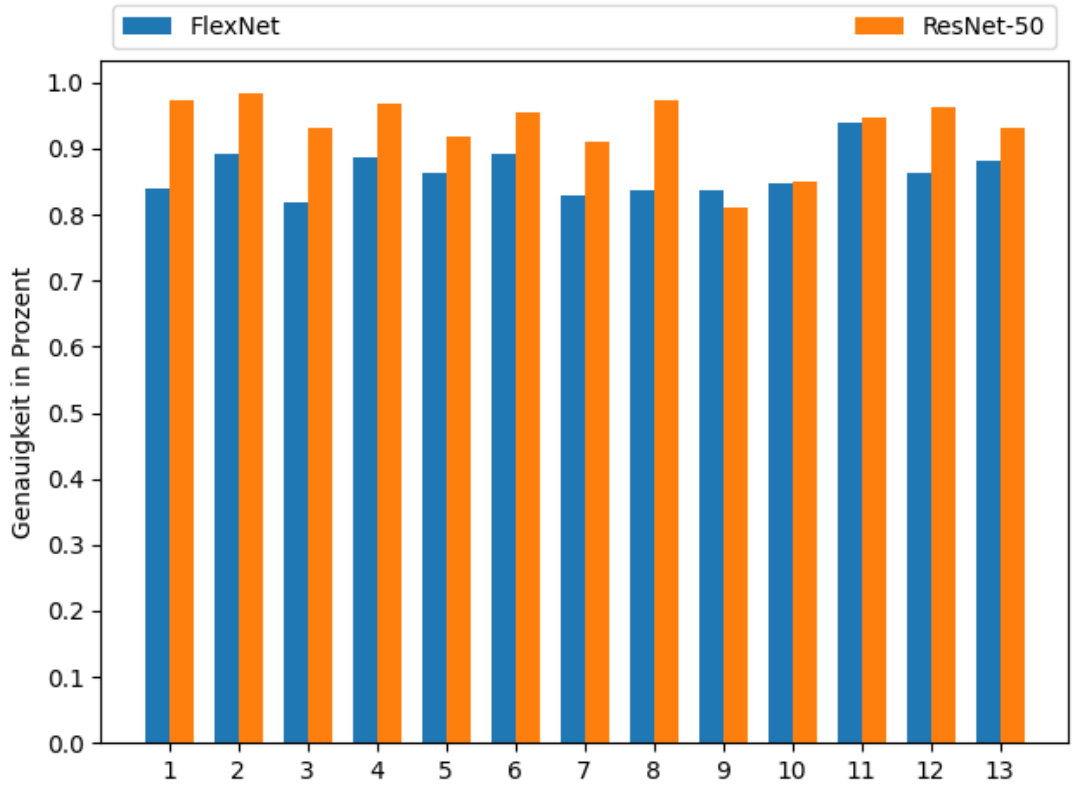


Die Abbildung links zeigt die Testgenauigkeit für jede Klasse. Für fast alle Klassen erreicht das ResNet eine höhere Genauigkeit, und der Unterschied zwischen dem FlexNet und ResNet ist max. ca. zehn Prozentpunkte.

Nun fügen wir zu dem bestehenden Modell die Kategorie Lego hinzu, das ResNet muss neu trainiert werden. Hier sind die gemessenen Daten für die Modelle mit den drei Kategorien:

	Genauigkeit (Training)	Genauigkeit (Test)	Trainingszeit	Ausführzeit
FlexNet	94,4%	86,2%	38m 19s	12ms
ResNet-50	99,9%	93,8 %	5h 12m 54s	13ms

Während die Trainingsgenauigkeit des ResNet konstant bleibt, singt sie beim FlexNet interessanterweise um fünf Prozentpunkte. Beide Modelle haben auch Verluste bei der Testgenauigkeit einbüßen müssen. Die Genauigkeit des ResNets sinkt um fünf Prozentpunkte, die des FlexNets um zehn. Somit ist die Testgenauigkeit des FlexNets durch Hinzufügen einer neuen Kategorie um 12% gesunken. Der signifikanteste unterschied liegt aber bei der Trainingszeit. Das ResNet benötigt fünf Stunden und zwölf Minuten, das FlexNet nur 38 Minuten. Somit benötigt das ResNet ca. achtmal länger um eine um 14% höhere Genauigkeit zu erreichen.

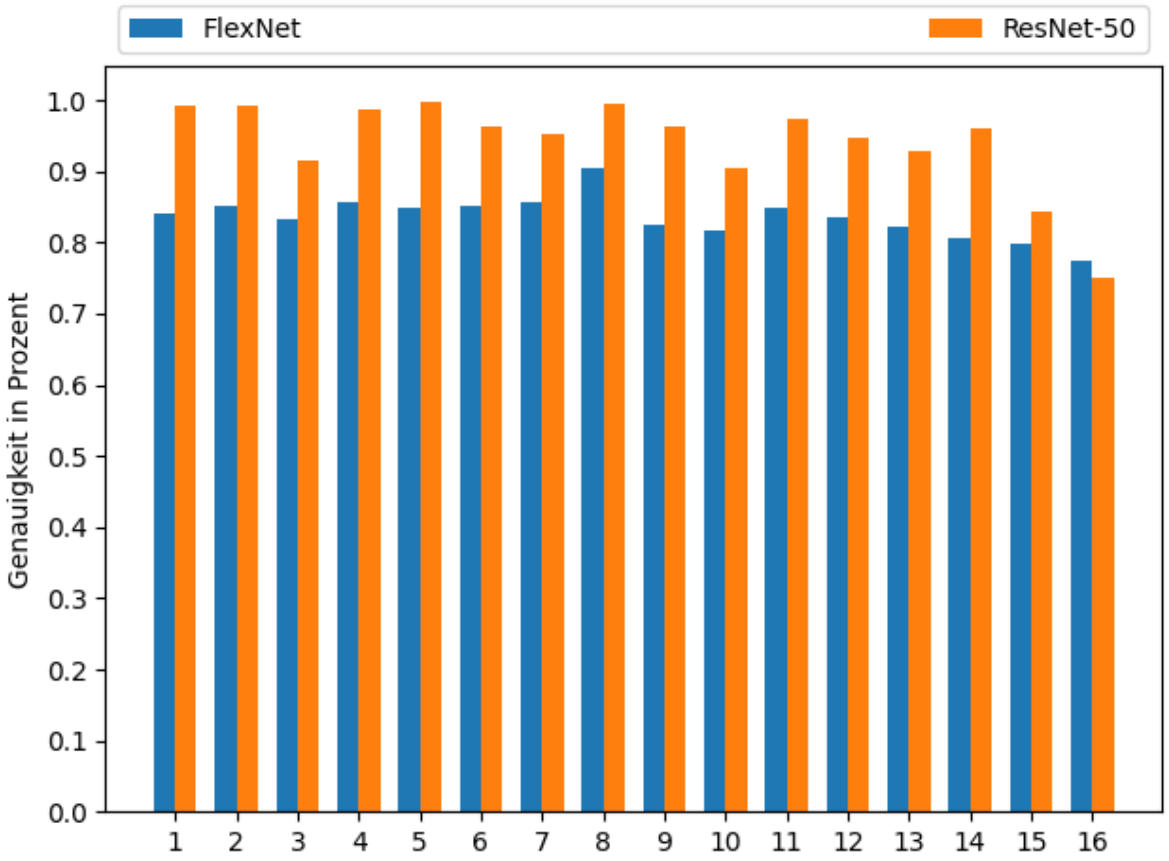


Bei Betrachtung der Grafik oben fällt auf, dass es bei der Genauigkeit des ResNets eine höhere Variation gibt. So ist zum Beispiel die niedrigste Genauigkeit beim ResNet und der Klasse 9 zu finden. Jedoch ist die übertrifft die Genauigkeit des FlexNets bei keiner Klasse die des ResNets.

Und hier die gemessenen Daten für alle vier Kategorien (Kategorie Chips hinzugefügt):

	Genauigkeit (Training)	Genauigkeit (Test)	Trainingszeit	Ausführzeit
FlexNet	90,2%	83,5%	1h 6m 32s	12ms
ResNet-50	99,9%	92,9 %	6h 45m 26s	13ms

Wie zu erwarten ist die Testgenauigkeit bei beiden Modellen gesunken. Das FlexNet benötigt nur 16% der Trainingszeit des ResNets und ist aber 10 Prozentpunkte weniger genau.

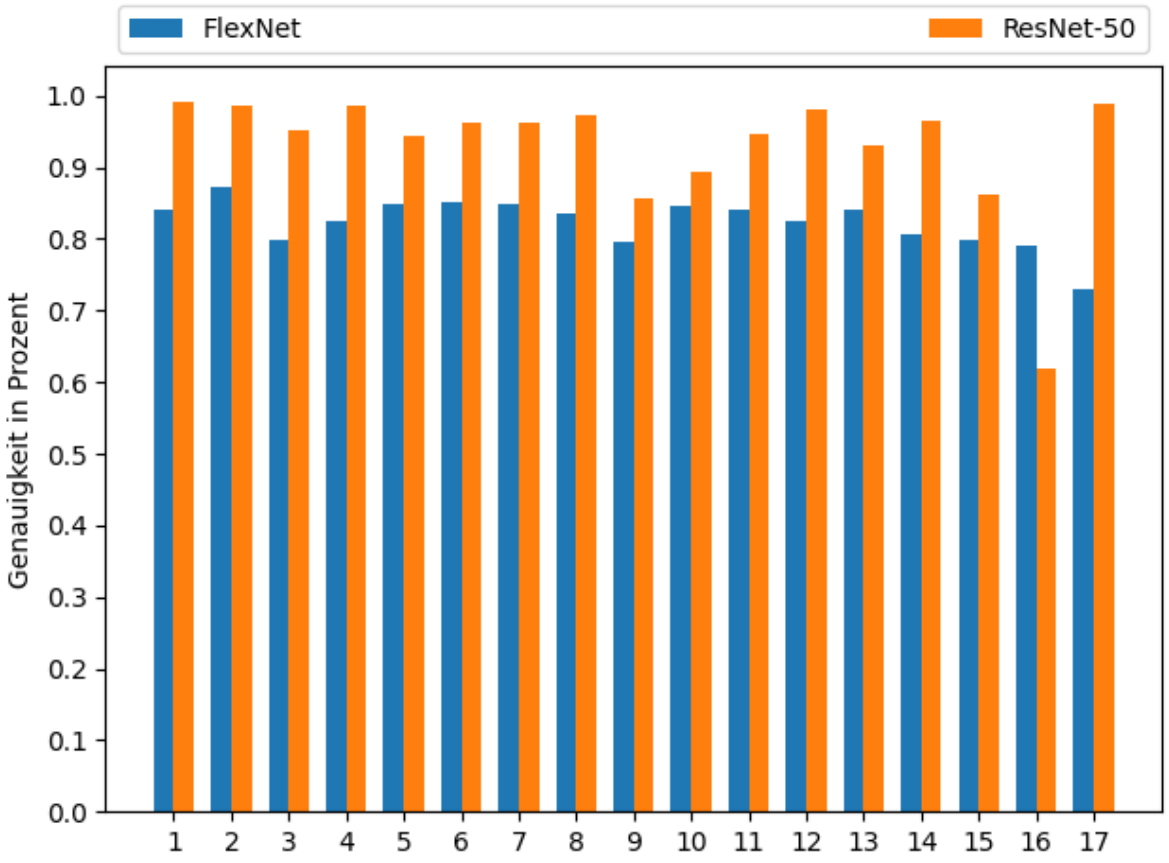


Hier ist ebenfalls auffällig das es mehr Variation in der Genauigkeit des ResNets gibt. Die höchste und niedrigste Genauigkeit wurde beide vom ResNet erreicht.

Zum Schluss fügen wir in der Kategorie Getränkedose eine weitere Klasse hinzu:

	Genauigkeit (Training)	Genauigkeit (Test)	Trainingszeit	Ausführzeit
FlexNet	89,7%	82,2%	9m 32s	12ms
ResNet-50	99,8%	93,1 %	7h 3m 47s	13ms

Der größte Unterschied bei den Trainingszeiten ist hier zu finden: Während nach nur neun ein halb Minuten sich das FlexNet fertig angepasst hat, benötigt das ResNet 42-mal mehr Zeit. Interessanterweise steigert das ResNet sogar seine Testgenauigkeit.



Das liegt vermutlich an der hohen Genauigkeit für die hinzugefügte Klasse (17). Auch wenn für Klasse 16 die Genauigkeit abrutscht, erreicht das ResNet für einige Klassen fast 100-prozentige Genauigkeit.

Zusammenfassung & Ausblick

Unser Ziel war es ein FlexNet zu implementieren, testen und vergleichen. Es lässt sich zunächst erstmal festhalten, dass alle Ziele erreicht wurden. Die Implementierung war dabei am schwierigsten (und bekommt witzigerweise am wenigsten Aufmerksamkeit in einer wissenschaftlichen Arbeit), da wir mit vielen bugs und anderen Problemen zu kämpfen hatten. Jedoch sind wir froh, eine funktionierende Version eines FlexNets fertiggestellt zu haben.

Die Testergebnisse zeigen das das FlexNet im Verleich zu einem ResNet eine schlechtere Leistung hat. Aber auch wenn wir von der Genauigkeit des FlexNets an manchen Stellen etwas enttäuscht waren, sind wir mehr als glücklich über die niedrigsten Trainingszeiten. Schließlich waren die der Hauptgrund zum Start dieses Projektes.

Wir planen das FlexNet weiter zu entwickeln und für unser in der Einleitung erwähntes Problem zur Plastikerkennung einzusetzen. Unserer Meinung nach ist es möglich die Genaunigkeit weiter zu steigern, indem man bessere Architekturen (z.B. ein kleines ResNet oder ein VGG-16 ähnliches Modell) als Submodell benutzt. Die unter der Verwendung von sehr einfach aufgebauten Submodellen erzielten Ergebnisse bestärken uns in dieser Annahme nur.

Gleichzeitig wollen wir in Zukunft auch noch andere Anwendungsfelder für das FlexNet, außerhalb der Bildklassifizierung, finden und erforschen.

Quellen- und Literaturverzeichnis

[1]: https://github.com/udacity/deep-learning-v2-pytorch/blob/master/convolutional-neural-networks/conv-visualization/conv_visualization.ipynb, Aufgerufen am 17.01.2021

[2] Zijie J. Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng (Polo) Chau: CNN EXPLAINER: Learning Convolutional Neural Networks with Interactive Visualization.

https://arxiv.org/pdf/2004.15004.pdf, Aufgerufen am 17.01.2021

[3]: https://pytorch.org/docs/stable/notes/randomness.html, Aufgerufen am 17.01.2021

Unterstützungsleistung

Michael Mertens, Immanuel-Kant-Gymnasium Dortmund, Projektbetreuer

Aladdin Persson, Technische Beratung bzgl. Implementierung der Modelle