



Anwendung der TensorFlow Object Detection API zur Detektion gesuchter Objekte auf Bildern

Bachelorarbeit

Eingereicht von: Gardiner, Dennis
Studiengang: Bachelor Elektro- und Informationstechnik
Matrikelnummer: 2017507932
Betreuer: Prof. Dr. Hubert Welp
Korreferent: M.Eng. Christopher Dillmann
Bearbeitungszeit: vom 01.10.2021
bis 13.01.2022

Technische Hochschule Georg Agricola
Wissenschaftsbereich Elektro-/Informationstechnik und Wirtschaftsingenieurwesen
Herner Str. 45, 44787 Bochum

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	iv
Abkürzungsverzeichnis	v
1 Einleitung	1
2 Vorbereitung der Softwareumgebung	3
2.1 Installation der Basisprogramme	3
2.2 Installation der TensorFlow API und benötigter Bibliotheken	8
3 Kurzanleitung: Modell in zehn Schritten trainieren	12
4 Training eigener Modelle	13
4.1 Vorbereitung des Datenraums	13
4.2 Vorbereitung der Datensätze	14
4.3 Training eines Modells	21
4.4 Anwendung des trainierten Modells	26
5 Installation auf alternativen Systemen	30
6 Möglichkeiten zur Verbesserung der Objekterkennung	33
6.1 Die Nutzung verschiedener Modelle	33
6.2 Der Einfluss der Auflösung auf die Detektion	37
6.3 Die Anzahl an verwendeten Bildern variieren	40
6.4 Der Einfluss der Schrittzahl beim Training	42
6.5 Die Nutzung verschiedener Hintergründe	44
6.6 Die Ergebnisse eines optimierten Modells	46
6.7 Weitere denkbare Einflüsse ohne Überprüfung	48
7 Transfer auf eine alternative Problemstellung	51
8 Mögliche Probleme	53
8.1 <i>Loss</i> pendelt sich nicht ein, Möglichkeit 1	53
8.2 <i>Loss</i> pendelt sich nicht ein, Möglichkeit 2	53
8.3 <i>Loss</i> zeigt <i>nan</i> statt einem Wert	54
8.4 Die Kostenfunktion endet mit einem Ausreißer nach oben	55
8.5 Das Training oder der Export werden wegen einer inkompatiblen <i>Tensor shape</i> abgebrochen	55
8.6 Ein Terminalbefehl wird nicht korrekt ausgeführt	56
9 Schlussfolgerung und Ausblick	57
Literaturverzeichnis	59
Anlagen	60
A Aufnahmen für die Detektion auf dem Modell unbekannten Bildern	60
B Objekterkennung auf den Testbildern zum Vergleich der verschiedenen Modelle	61
B.1 TensorBoard-Ausgaben für den Vergleich verschiedener Modelle	61
B.2 Die Vorlagenbilder ohne Detektionen	61
B.3 Die Detektionen	61

C	Objekterkennung auf Testbildern für hochauflösende Süßigkeiten auf weißem Hintergrund	62
C.1	TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten auf weißem Hintergrund	62
C.2	Die Vorlagenbilder ohne Detektionen	62
C.3	Die Detektionen	62
D	Objekterkennung auf Testbildern für niedrigauflösender Süßigkeiten auf weißem Hintergrund	62
D.1	TensorBoard-Ausgaben für das Training niedrigauflösender Süßigkeiten auf weißem Hintergrund	62
D.2	Die Detektionen	62
E	Objekterkennung auf Testbildern für hochauflösende Süßigkeiten auf weniger Bildern	62
E.1	TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten mit weniger Bildern	62
E.2	Die Detektionen	62
F	Objekterkennung auf Testbildern für hochauflösende Süßigkeiten mit hoher Schrittzahl	63
F.1	TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten mit hoher Schrittzahl	63
F.2	Die Detektionen	65
G	Objekterkennung auf Testbildern für hochauflösende Süßigkeiten auf multiplen Hintergründen	65
G.1	TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten auf multiplen Hintergründen	65
G.2	Die Vorlagenbilder ohne Detektionen	65
G.3	Die Detektionen	65
H	Objekterkennung von Süßigkeiten nach Verbesserung	65
H.1	TensorBoard-Ausgaben für das Training von Süßigkeiten nach Verbesserung	65
H.2	Die Detektionen	65
I	Objekterkennung von Werkzeugen	65
I.1	TensorBoard-Ausgaben für das Training von Werkzeugen	65
I.2	Die Vorlagenbilder ohne Detektionen	65
I.3	Die Detektionen	65
Abstract		66
Erklärung		67

Abbildungsverzeichnis

Abbildung 1:	Aktivierte virtuelle Umgebung <i>ai-ve</i> vom Benutzer <i>Dennis</i> auf dem Computer <i>Renner</i>	5
Abbildung 2:	Erfolgreich installierte TensorFlow API	10
Abbildung 3:	Die verwendete Ordnerstruktur	13
Abbildung 4:	Das Programm <i>labelImg</i> mit geöffnetem Ordner der Trainingsbilder .	16
Abbildung 5:	Bild mit gelabelten Objekten	17
Abbildung 6:	Aktiviertes TensorBoard	25
Abbildung 7:	Ausgabe des trainierten Modells für fünf zufällig ausgewählte Bilder .	27
Abbildung 8:	Einstellungen der Lernrate für das <i>efficientdet_d1</i> -Modell	54
Abbildung 9:	TensorBoard Ausgabe für <i>Loss/regularization_loss nan</i>	54
Abbildung 10:	Anpassung der Regularisierung in <i>pipeline.config</i>	55
Abbildung 11:	Training-Demo: Bild nach Verkleinerung und Detektion auf neutralem Hintergrund (Papier)	60
Abbildung 12:	Training-Demo: Bild nach Verkleinerung und Detektion auf Holztisch	61
Abbildung 13:	Hohe Schrittzahl: TensorBoard-Ausgabe für das Modell <i>fas-ter_rcnn_resnet101_v1</i>	63
Abbildung 14:	Hohe Schrittzahl: TensorBoard-Ausgabe für das Modell <i>ssd_resnet101_v1_fpn</i>	64

Tabellenverzeichnis

Tabelle 1: TensorBoard-Ausgaben für den Vergleich der Modelle	34
Tabelle 2: Ergebnisse des Modellvergleichs auf den Testbildern	36
Tabelle 3: TensorBoard-Ausgaben für verschiedene Auflösungen der gleichen Bilder	38
Tabelle 4: Detektionsergebnisse für verschiedene Auflösungen der gleichen Bilder	39
Tabelle 5: TensorBoard-Ausgaben für weniger Trainingsdaten	41
Tabelle 6: Detektionsergebnisse für weniger Trainingsdaten	41
Tabelle 7: TensorBoard-Ausgaben für lange Laufzeiten	43
Tabelle 8: Detektionsergebnisse für lange Laufzeiten	43
Tabelle 9: TensorBoard-Ausgaben für multiple Hintergründe	45
Tabelle 10: Detektionsergebnisse für multiple Hintergründe	45
Tabelle 11: TensorBoard-Ausgabe für das optimierte Modell	47
Tabelle 12: Detektionsergebnis für das optimierte Modell	47
Tabelle 13: TensorBoard-Ausgabe für das Training mit Werkzeugen	52
Tabelle 14: Detektionsergebnis für das Training mit Werkzeugen	52

Abkürzungsverzeichnis

ai-ve	artificial intelligence virtual environment
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
CUDA	Compute Unified Device Architecture
cuDNN	NVIDIA CUDA ® Deep Neural Network library
CNN	Convolutional Neural Networks
GPU	Graphics Processing Unit
GUI	Graphical User Interface
KI	Künstliche Intelligenz
LTS	Long Term Support
OS	Operating System
PCI	Peripheral Component Interconnect
PIP	pip installs packages

1 Einleitung

Auch wenn die ersten theoretischen Hintergründe der in dieser Ausarbeitung eingesetzten künstlichen neuronalen Netzwerke, aus dem englischen kurz **ANN** genannt, bereits in den 1940'er Jahren erarbeitet wurden¹, konnte trotz erster ernsthafter Versuche der praktischen Umsetzung in den 1990'er Jahren lange Zeit keine nennenswerte Nutzung aus künstlichen neuronalen Netzwerken gezogen werden.

Dank der steten Fortschritte in der verfügbaren Rechenleistung, insbesondere bei den häufig für maschinelles Lernen genutzten Grafikkarten, ist der Einsatz neuronaler Netzwerke in den letzten 10 Jahren rasant angewachsen und ist, wenn auch meist unsichtbar, heutzutage aus dem Alltag kaum noch wegzudenken. So arbeitet beispielsweise die Spracherkennung in Smartphones und ähnlichen Geräten häufig über **ANNs**.

Eine wichtige Anwendung finden diese umgangssprachlich oft als künstliche Intelligenz oder kurz **KI** (bzw. **AI** im englischen) bezeichneten maschinellen Lernalgorithmen somit bei der Unterstützung des Menschen im Alltag. Hieran soll diese Ausarbeitung anknüpfen. Die Motivation besteht darin, die Möglichkeiten neuronaler Netzwerke zur Objektdetektion mittels der TensorFlow **API** zu eruieren, um später, beispielsweise in einem Roboter verwendet, angeforderte Objekte in der Umgebung zu finden und anzureichen.

Ein derart "intelligenter" Roboter wäre damit beispielsweise in der Lage, sehbehinderten Menschen ein selbstständigeres Leben zu ermöglichen. So könnte eine einfache mündliche Anweisung lauten: "Reiche mir einen Schokoriegel". Der Roboter müsste dann in der Lage sein, beispielsweise anhand eines Kamerabildes, aus einer Auswahl an Objekten den Schokoriegel zu erkennen und die Position zu bestimmen.

Diese Arbeit soll daher eine Anleitung zur effektiven Umsetzung maschinellen Lernens zur Objektdetektion mittels der Bibliothek TensorFlow bereitstellen, welche auch für Laien ohne Vorkenntnisse in der Programmierung maschineller Lernalgorithmen geeignet ist. Für mehr Details zum Thema maschinelles Lernen: Die Grundlagen zum maschinellen Lernen und insbesondere der hier verwendeten Sonderform von **ANNs**, den sogenannten Convolutional Neural Networks (**CNN**), wurden bereits in einer früheren Ausarbeitung erarbeitet und vorgestellt; diese Ausarbeitung findet sich im begleitenden github repository.

Die Anleitung besteht dabei aus mehreren Schritten und führt von der Basisinstallation der benötigten Programme wie Python oder der **GPU**-Bindung mittels **CUDA** und **cuDNN** über das einbinden der nötigen Bibliotheken und Hilfsmittel hin zu einer ersten Demo, wie die spätere Ausgabe eigener **CNN**-Modelle, auch als Netzwerke bezeichnet, aussehen könnte.

¹Rojas, 1996.

Anschließend werden die einzelnen Schritte zum Erstellen und Anlernen eines eigenen Modells erläutert. Das so trainierte Netzwerk wird dann für den Nutzer anwendbar und die Erkennung verwertbar gemacht. Anhand des gewählten Beispiels zur Erkennung von Süßigkeiten werden letztlich die Ergebnisse des Trainings an einigen Testbildern veranschaulicht und Möglichkeiten zur Einflussnahme und Verbesserung erläutert.

Zuletzt wird aus den Erkenntnissen ein optimiertes Modell erarbeitet, um die vorhergehenden Schlussfolgerungen zu überprüfen. Zusätzlich wird an einem anderen möglichen Anwendungsfall gezeigt, dass die genutzten TensorFlow-Modelle auch zur Erkennung anderer Objekte geeignet sind und sowohl die Anleitungen als auch die Optimierungsschritte allgemein gültig sind.

Um den Einstieg für unerfahrene Nutzer weiterhin möglichst einfach zu halten beinhaltet diese Ausarbeitung neben dem eigentlichen Tutorial außerdem eine Zusammenfassung erkannter möglicher Probleme oder aufgetretener Fehler, sowie deren Lösungen oder wenigstens Lösungsansätze.

2 Vorbereitung der Softwareumgebung

Im Folgenden werden soweit möglich Befehle für das Terminal verwendet, um ein einfaches Abarbeiten der Befehle per copy and paste zu ermöglichen. Da diese Befehle oft länger als eine Zeile sind, werden diese in dieser druckbaren Datei leider umgebrochen. Um die Befehle trotzdem einfach abarbeiten zu können findet sich im *others*-Ordner des begleitenden github repository eine Textdatei *tutorial_commands* mit allen Befehlen zum einfachen Kopieren². In dieser Ausarbeitung werden zur besseren Auffindbarkeit der Befehle die Zeilennummern der Textdatei referenziert.

Achtung: Die Befehle beinhalten zum Teil Versionsnummern verwendeter Software. Diese ist an die tatsächlich genutzte Version anzupassen.

2.1 Installation der Basisprogramme

Als Betriebssystem wird die Linux-Distribution Ubuntu 20.04.2 LTS genutzt. Die verwendeten Programme sollten theoretisch auch auf allen weiteren gängigen Linux-Distributionen sowie unter Windows funktionieren, allerdings wurde dies nicht getestet.

Sollte der Nutzer gerne eine Linux-Umgebung nutzen wollen, jedoch bereits über ein Endgerät mit Windows verfügen, bietet sich neben dem Dual Boot (eine gute Anleitung zur Installation findet sich im Ubuntu-Bereich von StackExchange³) eventuell eine Virtuelle Maschine an. Aufgrund der im Regelfall deutlich höheren Rechenleistung sollte die verwendete Virtualisierung jedoch in der Lage sein, auf die GPU des verwendeten Computers zuzugreifen, weshalb die Nutzung einer virtuellen Maschine regelmäßig nur bei möglichem PCI Passthrough sinnvoll ist.

Vor Beginn der einzelnen Installationen ist eine Überprüfung des Paket-Managers und der grundlegenden Treiber auf Aktualität sowie eine Installation der essenziellen Software sinnvoll, indem die Befehle (Z. 1-4)

```
$ sudo apt-get update  
$ sudo apt install build-essential  
$ sudo ubuntu-drivers autoinstall  
$ sudo reboot
```

ausgeführt werden. Anschließend kann die weitere Software installiert werden. Die nötigen Installationen orientieren sich an dem Tutorial von Lyudmil Vladimirov⁴.

²https://github.com/Katschka/Object-Detection-Tutorial/blob/main/others/tutorial_commands.txt

³<https://askubuntu.com/questions/726972/dual-boot-windows-10-and-linux-ubuntu-on-separate-hard-drives>

⁴Vladimirov, 2021a.

Neben dem [OS](#) wird Python in seiner aktuellsten Version, zur Zeit der Ausarbeitung 3.9.6, genutzt. Standardmäßig enthält Ubuntu 20.04.2 [LTS](#) bereits die Version Python 3.8.10. Ein Upgrade auf die letzte stabile Version ist nicht nötig, jedoch empfehlenswert, um auch bei allen anderen Installationen auf die aktuellste Version zurückgreifen zu können. Die aktuelle Python-Version kann über den Terminal-Befehl ([Z. 6](#))

```
$ python3.x -V
```

abgefragt werden. Ohne das Suffix `.x` sollte die Standardversion 3.8.10 angezeigt werden, mit Suffix `.9` sollte die aktuellste Version angezeigt werden. Die letzte Stabile Version kann auf der Phyton-Webseite überprüft werden⁵. Zum aktualisieren der Python-Version kann der Befehl ([Z. 8](#))

```
$ sudo apt-get install python3.x
```

verwendet werden, dabei muss der Patch, der Eintrag nach dem zweiten Punkt⁶, nicht angegeben werden.

Ein weiteres nützliches Tool ist Anaconda⁷, welches in diesem Fall in der Version 4.10.1 verwendet wird.

Die Installation wird mittels ([Z. 10-11](#))

```
$ wget /tmp https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh  
$ bash /tmp/Anaconda3-2021.05-Linux-x86_64.sh
```

durchgeführt. Die Installation wird geführt und die Aufforderungen sind selbsterklärend, zum Abschluss kann Anaconda direkt initialisiert werden. Nach erfolgreicher Installation sollte das aktuelle Terminal geschlossen werden.

Im nächsten Schritt wird mittels Anaconda eine virtuelle Umgebung erstellt. Dies gehört bei der Arbeit mit Python zum guten Ton, da ein Projekt komplett in einer virtuellen Umgebung umgesetzt werden kann.

Neben der klaren Struktur ermöglicht eine virtuelle Umgebung auch eine unabhängige Umsetzung sämtlicher Projekte mit der jeweils benötigten Software und vor allem Softwareversion, selbst wenn benötigte Pakete oder Abhängigkeiten in verschiedenen Umgebungen widersprüchlich wären. Eine virtuelle Umgebung ist dabei **keine** virtuelle Maschine sondern lediglich eine reine Projektumgebung.

⁵<https://www.python.org/downloads/source/>

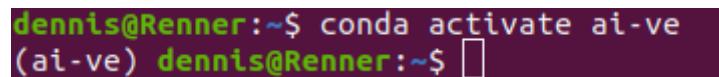
⁶Preston-Werner, o.D.

⁷<https://docs.anaconda.com/anaconda/install/linux/>

Im folgenden soll die virtuelle Umgebung [ai-ve](#) erstellt und genutzt werden: ([Z. 13-15](#))

```
$ conda create -n ai-ve pip python=3.9
$ conda activate ai-ve
$ conda update --all --yes
```

Diese virtuelle Umgebung wird dabei mit den grundlegenden Elementen der vorher installierten Python 3.x-Version ausgestattet. Nach Bestätigung durch den Nutzer kann die vorbereitete Umgebung aktiviert werden, im Terminal sollte nun der Name der virtuellen Umgebung vor dem Nutzer, hier *Dennis* auf dem Computer *Renner*, stehen:



```
dennis@Renner:~$ conda activate ai-ve
(ai-ve) dennis@Renner:~$ █
```

Abbildung 1: Aktivierte virtuelle Umgebung *ai-ve* vom Benutzer *Dennis* auf dem Computer *Renner*

Ab sofort werden alle weiteren Befehle in dieser virtuellen Umgebung durchgeführt, sie ist daher nach jedem öffnen eines neuen Terminals wieder zu aktivieren!

Für die Umsetzung der gewünschten Objekterkennung soll ein vorgefertigtes [CNN](#) genutzt werden, welches in Googles Framework TensorFlow enthalten ist. Zur Installation von TensorFlow kann der Python-interne Paketmanager [PIP](#) genutzt werden: ([Z. 17](#))

```
$ pip install --ignore-installed --upgrade tensorflow==2.5.0
```

Um TensorFlow noch effektiver nutzen zu können hat sich die Konvention durchgesetzt, TensorFlow nur als *tf* zu importieren⁸. Der folgende Befehl nutzt dies und gibt einige Informationen zum aktuellen Stand der Installationen: ([Z. 19](#))

```
$ python -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000,
1000])))"
```

Die Meldungen nach dem Import besagen, dass der gewünschte [GPU](#)-Support noch nicht gegeben ist. Um die eingangs bereits erwähnte deutlich höhere Leistung der [GPU](#) nutzen zu können werden weitere Installationen benötigt. Da in dem zur Projektierung genutztem Computer eine [GPU](#) vom Typ NVIDIA GTX 980 Ti genutzt wird sind zunächst das [CUDA](#) Toolkit, genutzt in Version 11.4, sowie [cuDNN](#) - letzteres benötigt eine Registrierung bei NVIDIA - zu installieren.

Für das vorgestellte System lauten die Befehle zur Installation von [CUDA](#) ([Z. 21-27](#))

⁸Vladimirov, 2021b.

```
$ sudo apt-get install linux-headers-$(uname -r)
$ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/ | 
  curl -O cuda-ubuntu2004.pin
$ sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600
$ sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/ | 
  curl -O repos/ubuntu2004/x86_64/7fa2af80.pub
$ sudo add-apt-repository "deb
  https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/ /"
$ sudo apt-get update
$ sudo apt-get -y install cuda
```

Einige Verknüpfungen müssen nun leider von Hand erstellt werden, dazu wird der integrierte Texteditor *nano* genutzt: ([Z. 29](#))

```
$ nano ~/.bashrc
```

Es öffnet sich die Systemdatei *.bashrc*, in welcher am Ende die folgenden Einträge ergänzt werden: ([Z. 31-34](#))

```
# CUDA related exports
export PATH=/usr/local/cuda-11.4/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-11.4/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export PATH=/home/$USER/Protobuf/bin${PATH:+:${PATH}}
```

Anschließend kann der Editor mit der Tastenkombination **Strg + x** beendet werden. Dabei wird abgefragt, ob die Änderung gespeichert werden soll, dies ist zu bestätigen. Dabei wird der Dateiname erneut bestätigt.

Nach erfolgreicher Installation sollte in einem neuen Terminal der Befehl ([Z. 36](#))

```
$ nvidia-smi
```

die aktuelleste Version des Grafikkartentreibers, in diesem Fall 470.57.02, anzeigen. Weiterhin sollte ([Z. 38](#))

```
$ nvcc -V
```

die aktuelle **CUDA**-Version, hier also 11.4 anzeigen. Nach erfolgreicher Installation muss überprüft werden, ob der NVIDIA Persistance Daemon läuft: ([Z. 40](#))

```
$ systemctl status nvidia-persistenced
```

Wird hier kein aktiver (running) Dienst angezeigt, muss der Befehl (Z. 42)

```
$ sudo systemctl enable nvidia-persistenced
```

ausgeführt werden, um den Dienst zu aktivieren. Für die verwendete CUDA-Version wird im nächsten Schritt die cuDNN-Bibliothek manuell heruntergeladen⁹, da hierfür die Anmeldung bei NVIDIA notwendig ist. Im Terminal wird nun in das Verzeichnis gewechselt, in welchem die heruntergeladene Bibliothek liegt, standardmäßig ist dies *Downloads*, und das heruntergeladene Verzeichnis entpackt sowie installiert: (Z. 44-46)

```
$ cd ~/Downloads  
$ tar -xzvf cudnn-11.4-linux-x64-v8.2.2.26.tgz  
$ rm cudnn-11.4-linux-x64-v8.2.2.26.tgz
```

Anschließend müssen einige Dateien in den CUDA-Ordner kopiert werden: (Z. 48-50)

```
$ sudo cp cuda/include/cudnn*.h /usr/local/cuda/include  
$ sudo cp -P cuda/lib64/libcudnn* /usr/local/cuda/lib64  
$ sudo chmod a+r /usr/local/cuda/include/cudnn*.h /usr/local/cuda/lib64/libcudnn*
```

Weiterhin werden die Runtime Library¹⁰, die Developer Library¹¹, und die Beispiele¹² heruntergeladen. Diese können im Download-Ordner per Doppelklick direkt installiert werden.

Wird nun der Befehl (Z. 52)

```
$ python -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000,  
1000])))"
```

erneut ausgeführt, sollten alle erforderlichen Bibliotheken installiert sein. Nun kann die Installation überprüft werden: (Z. 54-56)

⁹https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.2.2/11.4_07062021/cudnn-11.4-linux-x64-v8.2.2.26.tgz

¹⁰https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.2.2/11.4_07062021/Ubuntu20_04-x64/libcudnn8_8.2.2.26-1+cuda11.4_amd64.deb

¹¹https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.2.2/11.4_07062021/Ubuntu20_04-x64/libcudnn8-dev_8.2.2.26-1+cuda11.4_amd64.deb

¹²https://developer.nvidia.com/compute/machine-learning/cudnn/secure/8.2.2/11.4_07062021/Ubuntu20_04-x64/libcudnn8-samples_8.2.2.26-1+cuda11.4_amd64.deb

```
$ cp -r /usr/src/cudnn_samples_v8/ $HOME  
$ cd $HOME/cudnn_samples_v8/mnistCUDNN  
$ make clean && make
```

Bei diesem Schritt tritt möglicherweise die Fehlermeldung *WARNING - FreeImage is not set up correctly. Please ensure FreeImage is set up correctly* auf. In diesem Fall kann die fehlende Bibliothek mit (Z. 58)

```
$ sudo apt-get install libfreeimage3 libfreeimage-dev
```

installiert und der Test fortgesetzt werden: (Z. 60)

```
$ ./mnistCUDNN
```

Nun sollte am Ende der Ausgaben im Terminal *Test passed!* stehen.

2.2 Installation der TensorFlow API und benötigter Bibliotheken

Nachdem alle erforderlichen Basisprogramme installiert sind werden vorgefertigte TensorFlow-Modelle über die TensorFlow Object Detection API nutzbar gemacht. Zunächst müssen dafür die Modelle heruntergeladen und entpackt werden: (Z. 62-64)

```
$ mkdir /home/$USER/TensorFlow && wget -P /home/$USER/TensorFlow  
https://github.com/tensorflow/models/archive/refs/heads/master.zip  
$ cd /home/$USER/TensorFlow  
$ unzip master.zip && mv /home/$USER/TensorFlow/models-master  
/home/$USER/TensorFlow/models && cd ~
```

Anschließend kann die aktuellste Version von Protocol Buffers¹³, zur Zeit v3.17.3, heruntergeladen werden: (Z. 66-67)

```
$ mkdir /home/$USER/Protobuf && wget -P /home/$USER/Protobuf https://github.com/_  
protocolbuffers/protobuf/releases/download/v3.17.3/protoc-3.17.3-linux-x86_64.zip  
$ cd /home/$USER/Protobuf && unzip protoc-3.17.3-linux-x86_64.zip
```

Um diesen Buffer nutzen zu können muss das aktuelle Terminal geschlossen und ein neues geöffnet werden, anschliend kann in das Verzeichnis (Z. 69)

¹³https://github.com/protocolbuffers/protobuf/releases/download/v3.17.3/protoc-3.17.3-linux-x86_64.zip

```
$ cd /home/$USER/TensorFlow/models/research
```

gewechselt und der folgende Befehl ausgeführt werden: ([Z. 71](#))

```
$ protoc object_detection/protos/*.proto --python_out=.
```

Als letzter Schritt vor der eigentlichen Installation der gewünschten TensorFlow-Schnittstelle ist die Installation der COCO [API](#) nötig, da diese von TensorFlow vorausgesetzt wird: ([Z. 73-75](#))

```
$ mkdir /home/$USER/cocoapi && wget -P /home/$USER/cocoapi https://github.com/cocodataset/cocoapi/archive/refs/heads/master.zip  
$ cd /home/$USER/cocoapi && unzip master.zip  
$ cp -r cocoapi-master/PythonAPI/pycocotools /home/$USER/TensorFlow/models/research
```

Zur Installation der TensorFlow Object Detection [API](#) werden im Verzeichnis ([Z. 77](#))

```
$ cd /home/$USER/TensorFlow/models/research
```

die Befehle ([Z. 79-80](#))

```
$ cp object_detection/packages/tf2/setup.py .  
$ python -m pip install --use-feature=2020-resolver .
```

ausgeführt. Die Installation ist nun abgeschlossen und kann mit dem Befehl ([Z. 82](#))

```
$ python object_detection/builders/model_builder_tf2_test.py
```

überprüft werden. Das Ende der längeren Terminal-Ausgaben sollte bei erfolgreicher Installation ähnlich zu [Abbildung 2](#) sein und bei allen Tests ein *OK* ausgeben:

```
[       OK ] ModelBuilderTF2Test.test_create_ssd_models_from_config
[ RUN      ] ModelBuilderTF2Test.test_invalid_faster_rcnn_batchnorm_update
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_invalid_faster_rcnn_batchnorm_update): 0.0s
I0824 20:42:55.031386 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_invalid_faster_rcnn_batchnorm_update): 0.0s
[       OK ] ModelBuilderTF2Test.test_invalid_faster_rcnn_batchnorm_update
[ RUN      ] ModelBuilderTF2Test.test_invalid_first_stage_nms_iou_threshold
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_invalid_first_stage_nms_iou_threshold): 0.0s
I0824 20:42:55.032325 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_invalid_first_stage_nms_iou_threshold): 0.0s
[       OK ] ModelBuilderTF2Test.test_invalid_first_stage_nms_iou_threshold
[ RUN      ] ModelBuilderTF2Test.test_invalid_model_config_proto
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_invalid_model_config_proto): 0.0s
I0824 20:42:55.032536 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_invalid_model_config_proto): 0.0s
[       OK ] ModelBuilderTF2Test.test_invalid_model_config_proto
[ RUN      ] ModelBuilderTF2Test.test_invalid_second_stage_batch_size
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_invalid_second_stage_batch_size): 0.0s
I0824 20:42:55.033344 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_invalid_second_stage_batch_size): 0.0s
[       OK ] ModelBuilderTF2Test.test_invalid_second_stage_batch_size
[ RUN      ] ModelBuilderTF2Test.test_session
[ SKIPPED ] ModelBuilderTF2Test.test_session
[ RUN      ] ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor): 0.0s
I0824 20:42:55.034083 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor): 0.0s
[       OK ] ModelBuilderTF2Test.test_unknown_faster_rcnn_feature_extractor
[ RUN      ] ModelBuilderTF2Test.test_unknown_meta_architecture
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_unknown_meta_architecture): 0.0s
I0824 20:42:55.034264 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_unknown_meta_architecture): 0.0s
[       OK ] ModelBuilderTF2Test.test_unknown_meta_architecture
[ RUN      ] ModelBuilderTF2Test.test_unknown_ssd_feature_extractor
INFO:tensorflow:time(__main__.ModelBuilderTF2Test.test_unknown_ssd_feature_extractor): 0.0s
I0824 20:42:55.034840 140288342872448 test_util.py:2102] time(__main__.ModelBuilderTF2Test.test_unknown_ssd_feature_extractor): 0.0s
[       OK ] ModelBuilderTF2Test.test_unknown_ssd_feature_extractor
-----
Ran 24 tests in 13.859s
OK (skipped=1)
```

Abbildung 2: Erfolgreich installierte TensorFlow API

Um bei späterer Verwendung auch direkt Bilder ausgeben zu können, wird außerdem noch ein aktuelles Framework für **GUIs** benötigt, hier bietet sich *PyQt5* an, welches mit ([Z. 84-87](#))

```
$ pip3 install --user PyQt5
$ sudo apt-get install python3-pyqt5
$ sudo apt-get install pyqt5-dev-tools
$ sudo apt-get install qttools5-dev-tools
```

installiert wird.

Die Installation ist nun abgeschlossen und es können eigene Modelle trainiert und damit Detektionen durchgeführt werden.

Um ein Beispiel für solche Detektionen zu erhalten kann das Python-Skript genutzt werden, welches im Tutorial bereitgestellt wird¹⁴. Dafür muss es jedoch zunächst auf *PyQt5* angepasst werden. Die angepasste Version findet sich im github repository und kann direkt mit folgendem Befehl heruntergeladen und ausgeführt werden: (Z. 89-90)

```
$ wget -O /home/$USER/TensorFlow/scripts/Demo_Objekterkennung.py  
https://raw.githubusercontent.com/Katschka/Object-Detection-Tutorial/main/scripts/_  
Demo_Objekterkennung.py?token=AV2PSU3AKY4VRNIBFKCL2LTBPWCZ4  
$ python /home/$USER/TensorFlow/scripts/Demo_Objekterkennung.py
```

Nach Durchlauf des Skripts öffnen sich zwei Fenster mit Beispielbildern und Detektionen verschiedener Objekte, außerdem wird im Terminal *Running inference for ...* angezeigt. Solche Ausgaben werden im nächsten Kapitel durch selbst trainierte Netzwerke auf dem Modell bisher unbekannten Bildern erzeugt.

¹⁴Originalskript:https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/_downloads/c3ac71663c072281de415e2f7a7f5670/plot_object_detection_saved_model.py

3 Kurzanleitung: Modell in zehn Schritten trainieren

Basierend auf einer vollständigen Installation der benötigten Software kann das Training eines Modells anhand der folgenden Schritte abgearbeitet werden. Eine detaillierte Erläuterung der einzelnen Schritte erfolgt anschließend in [Abschnitt 4](#).

1. Arbeitsraum vorbereiten: Vortrainiertes Modell herunterladen (Auflösung der Bilder und verfügbare Rechenleistung beachten), leeren *Workspace* anlegen oder von Vorlage erstellen
2. Bilder aufnehmen: Mögliche Variationen (Hintergrund, Beleuchtung, Winkel, Ausrichtung, Abstand etc.) abbilden - im Zweifelsfall weniger Bilder mit mehr Variationen
3. Bilder skalieren: Auf eine Größe anpassen, welche dem Menschen noch eine eindeutige manuelle Detektion erlaubt. Skript *resize_images* nutzen
4. Bilder labeln: Tool *labelImg* nutzen
5. Bilder aufteilen: Trainings- und Testdaten möglichst im Verhältnis 9:1. Skript *partition_dataset* nutzen
6. *TFRecords* erstellen: Skript *generate_tfrecord* nutzen
7. *pipeline.config*-Datei anpassen: Datei in *Workspace* kopieren und an Pfade und Anzahl der Klassen anpassen
8. Modell trainieren: Skript *model_main_tf2* in den *Workspace* kopieren und nutzen. Optional: Training mittels *TensorBoard* überwachen
9. Modell exportieren: Skript *exporter_main_v2* in den *Workspace* kopieren und nutzen
10. Modell anwenden: Skript *trained_object_detection* nutzen und nach Bedarf anpassen

4 Training eigener Modelle

Im Folgenden werden mehrere vortrainierte Modelle genutzt, welche auf die gegebene Problemstellung trainiert und anschließend exportiert werden. Da diese trainierten und exportierten Modelle mehrere Gigabyte Speicherplatz benötigen ist ein Upload aller Modelle auf github leider nicht möglich.

Bei Bedarf können jedoch zumindest die exportierten Modelle von der hochschuleigenen Cloud heruntergeladen werden.¹⁵

4.1 Vorbereitung des Datenraums

Prinzipiell ist es sinnvoll, für das Training eigener Modelle bzw. eigener Datensätze eine strukturierte Arbeitsumgebung zu schaffen. Im folgenden soll die nebenstehende Ordnerstruktur genutzt werden, welche im bereits angelegten TensorFlow-Ordner hinterlegt wird.

Diese Struktur besteht aus einigen allgemeinen Ordnern sowie einem dediziertem *workspace* für jede Aufgabenstellung.

Im übergeordneten TensorFlow-Ordner werden im Laufe der nächsten Schritte die vortrainierten Modelle heruntergeladen und im Ordner *pre-trained-models* abgelegt. Die heruntergeladenen Modelle können dann für jeden späteren Workspace genutzt werden.

Gleiches gilt für einige nützliche Scripte, welche ebenfalls hier abgelegt werden sollen.

Der eigentliche Ordner, in welchem die nötigen Trainingsdaten abgelegt werden, ist der *workspace*-Ordner *training_demo*. Für weitere Projekte kann also der Ordner *training_demo* als Vorlage für die tieferliegende Ordnerstruktur verwendet werden.

Die gesamte Struktur kann wahlweise über die grafische Benutzeroberfläche, von der DVD im Anhang oder direkt mit dem Befehl (Z. 92-93)

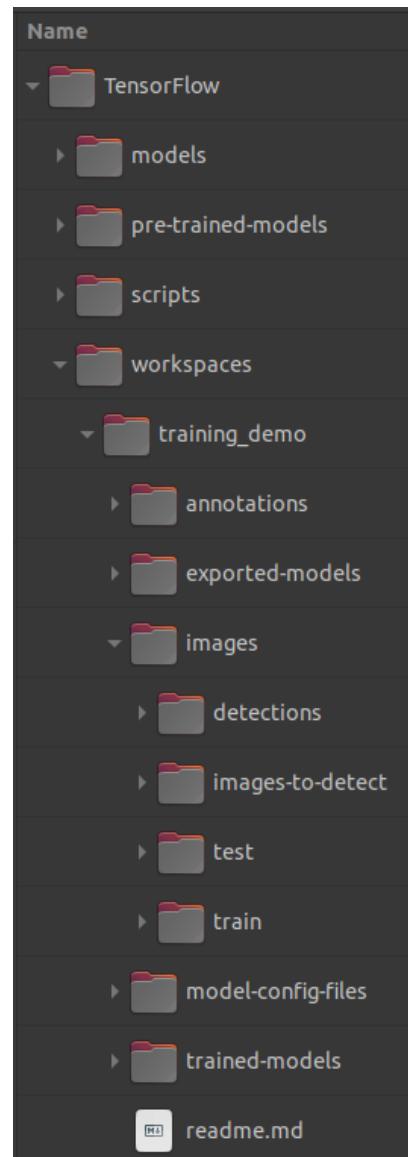


Abbildung 3: Die verwendete Ordnerstruktur

¹⁵<https://thga.sciebo.de/s/e3C1uJOVlSt0QpF>

```
$ cd /home/$USER/TensorFlow
$ mkdir pre-trained-models scripts workspaces workspaces/training_demo
workspaces/training_demo/annotations workspaces/training_demo/exported-models
workspaces/training_demo/images workspaces/training_demo/images/images-to-detect
workspaces/training_demo/images/detections workspaces/training_demo/images/test
workspaces/training_demo/images/train workspaces/training_demo/model-config-files
workspaces/training_demo/trained-models && cd workspaces/training_demo && touch
README.md && cd /home/$USER/TensorFlow
```

angelegt werden. Die ebenfalls angelegte Datei *README.md* ist noch leer und optional, gehört aber gerade bei öffentlich verfügbaren (github) Projekten zum guten Stil. Sie ist dafür gedacht, allgemeine Informationen über die genutzten Modelle, Einstellungen und zum Training genutzten Daten zu speichern und wird nach Bedarf manuell ergänzt.

Die Funktionen und Inhalte der übrigen Ordner werden in den nächsten Kapiteln deutlich.

4.2 Vorbereitung der Datensätze

Das Modell benötigt zum erlernen dessen, was erkannt werden soll, Beispieldaten, welche mit Annotationen bezüglich des abgebildeten Inhalts versehen sind. Dabei ist darauf zu achten, dass die verwendeten Bilder der späteren Eingabegröße der verwendeten Modelle entsprechen.

Leider ist der in den TensorFlow-Modellen integrierte Mechanismus zur Anpassung der Bildgrößen aktuell nicht in der Lage, die übergebenen Annotationen mit zu skalieren! Werden die Bilder in einer anderen Auflösung gelabelt als später im Algorithmus verwendet führt dies höchst wahrscheinlich dazu, dass das trainierte Modell schlechtere Ergebnisse erzielt, wenn nicht sogar komplett unbrauchbar ist.

Die dank stetig fortschreitender Entwicklung größer werdende Auflösung von Kamerabildern kann an dieser Stelle daher negativ ins Gewicht fallen, falls die Trainingsbilder eine deutlich höhere Auflösung besitzen, als von den genutzten Modellen bei der gegebenen Hardware verarbeitet werden kann. In diesem Fall können mit einem kleinen Tool ([Z. 95](#))

```
$ pip install scikit-image
```

und einem weiteren Skript alle Bilder in einem Ordner auf die gewünschte Größe neu skaliert werden, welches mit: ([Z. 97](#))

```
$ wget -O /home/$USER/TensorFlow/scripts/resize_images.py https://github.com/Katschka/_Object-Detection-Tutorial/raw/main/scripts/resize_images.py
```

heruntergeladen werden kann. Das Skript fragt dabei den Pfad zu den zu verkleinernden Bildern sowie die gewünschte Endgröße in Pixeln ab. Allgemein funktioniert das Skript nach dem Schema

```
python resize_images.py -i [PATH_TO_IMAGES_FOLDER] -x [HORIZONTAL SIZE IN PIXEL] -y  
[VERTICAL SIZE IN PIXEL]
```

Da im Skript **alle** Dateien im angegebenen Ordner ohne Prüfung auf das Dateiformat durchlaufen werden, sollte vorher sichergestellt werden, dass tatsächlich nur Bilder in dem Ordner liegen. Eventuelle Unterordner werden jedoch ignoriert.

Nach Durchlauf des Skripts werden die Bilder mit dem Präfix *resized_* gespeichert, die Originale bleiben unverändert erhalten. Zur Orientierung: bei der hier verwendeten Grafikkarte (*GTx 980 Ti*) mit ca. 6 GB Speicher konnten Bilder von bis zu 1280×1280 Pixeln gerade noch verarbeitet werden. Für einige der verfügbaren Modelle war dies jedoch bereits zu hochauflösend.

Für diese Anleitung sollen im folgenden Bilder verschiedener Süßigkeiten genutzt werden. Hierzu stehen insgesamt 159 Bilder¹⁶ zur Verfügung, welche mit folgendem Befehl in den *images*-Ordner heruntergeladen und entpackt werden können: ([Z. 99-100](#))

```
$ wget -O /home/$USER/TensorFlow/workspaces/training_demo/images/Trainingsdaten.zip  
https://github.com/Katschka/Object-Detection-Tutorial/blob/main/images/_  
sweets%20low%20resolution%20by%20Rami%20Alkholli.zip  
$ cd /home/$USER/TensorFlow/workspaces/training_demo/images && unzip Trainingsdaten.zip  
&& rm Trainingsdaten.zip
```

Um die Bilder und vor allem die Annotationen für das gewählte Modell nutzbar zu machen wird das Tool *labelImg* verwendet, welches mit ([Z. 102](#))

```
$ pip install labelImg
```

installiert werden kann. Das Programm wird durch die einfache Eingabe seines Namens in ein Terminal gestartet: ([Z. 104](#))

```
$ labelImg
```

Es öffnet sich die graphische Benutzeroberfläche des Programms, in welcher über den Button *Open Dir* der Bilder-Ordner ausgewählt werden kann, in welchem gerade die 159 Bilder

¹⁶Alkholli, 2020.

gespeichert wurden. Über den Button *Change Save Dir* muss der selbe Ordner nochmal ausgewählt werden, um den Speicherort der gleich erstellten Annotationen zu verifizieren. Alternativ kann auch direkt über das Terminal in den Ordner gestartet werden: ([Z. 106](#))

```
$ labelImg /home/$USER/TensorFlow/workspaces/training_demo/images
```

Hinweis: Durch den Aufruf von *labelImg* wird das hierzu verwendete Terminal quasi gesperrt, bis die graphische Oberfläche von *labelImg* geschlossen wird.

Das geöffnete Fenster sollte nun [Abbildung 4](#) entsprechen:

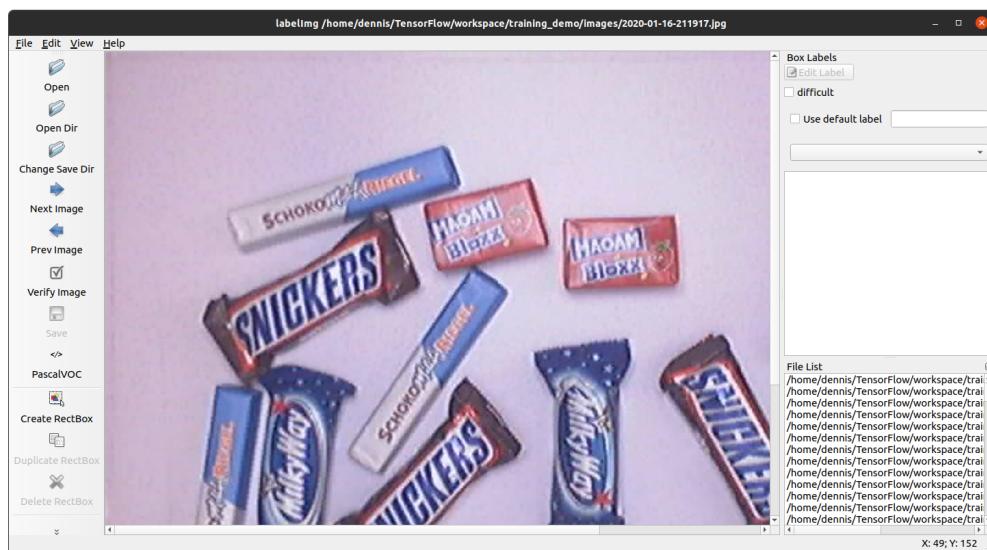


Abbildung 4: Das Programm *labelImg* mit geöffnetem Ordner der Trainingsbilder

Über den Button *Create RectBox* (oder das Tastaturkürzel **w**) kann in der Oberfläche eine Rechteckige Box um jedes einzelne Objekt gezogen werden. Eine Drehung der Box ist nicht möglich, in diesem Fall wird die Box entsprechend größer gezogen.

Nachdem das Objekt markiert wurde kann entweder ein neues Label eingegeben oder ein existierendes Label ausgewählt werden. Am rechten Bildschirmrand werden dabei alle bisher vergebenen Label aufgelistet. [Abbildung 5](#) zeigt ein Beispiel für ein vollständig gelabeltes Bild:



Abbildung 5: Bild mit gelabelten Objekten

Sobald alle Objekte im Bild wie gewünscht gelabelt wurden kann über den Speicher-Button oder **Strg+s** eine *.xml*-Datei gespeichert werden, welche ebenfalls im *images*-Ordner abgelegt wird. Gleichzeitig wird im Terminal angezeigt, welche Datei gerade gespeichert wurde.

Zwei weitere nützliche Tastaturkürzel sind **a** und **d**, um zwischen den Bildern vor- und zurückzuspringen. Für ein flüssigeres Arbeiten kann unter dem Menüpunkt *View* der Haken bei *Auto Save mode* gesetzt werden, was das manuelle Speichern obsolet macht. Außerdem kann im selben Menü mit der Option *Advanced Mode* festgelegt werden, dass nach Aktivierung permanent die Option *Create RectBox* ausgewählt bleibt.

Hinweis: 159 Bilder sind für maschinelles Lernen eher wenig Material, dennoch ist es eine zeitintensive Arbeit, alle Bilder korrekt zu labeln. Für größere Projekte bietet es sich daher an, diese Arbeit extern durchführen zu lassen. Hierzu finden sich zahlreiche Anbieter online, welche den Label-Prozess meist über Crowdsourcing an günstige Arbeiter verteilen.

Auf eine Auslagerung des Labels im Rahmen dieser Arbeit wurde aufgrund des verhältnismäßig geringen Umfangs verzichtet. Um der Anleitung ohne Zeitverzug weiter folgen zu können, stehen die fertigen *.xml*-Dateien ebenfalls zum Download bereit: ([Z. 108-109](#))

```
$ wget -O /home/$USER/TensorFlow/workspaces/training_demo/images/xmels.zip
https://github.com/Katschka/Object-Detection-Tutorial/raw/main/others/xmels.zip
$ cd /home/$USER/TensorFlow/workspaces/training_demo/images && unzip xmels.zip && rm
xmels.zip
```

Die fertig gelabelten Bilder müssen anschließend in Test- und Trainingsdaten unterteilt werden. Auf eine zusätzliche Unterteilung in einen Validierungsdatensatz wird bei den TensorFlow-

Modellen der Einfachheit halber verzichtet.

Zur Unterteilung kann das folgende Skript, welches ebenfalls an das Tutorial¹⁷ angelehnt ist, heruntergeladen werden: ([Z. 111](#))

```
$ wget -O /home/$USER/TensorFlow/scripts/partition_dataset.py https://github.com/J  
Katschka/Object-Detection-Tutorial/raw/main/scripts/partition_dataset.py
```

Das Skript basiert auf der oben angelegten Ordnerstruktur und nutzt die Ordner *test* und *train*, welche gerade angelegt wurden - sollte dies nicht der Fall sein, werden die Ordner nun angelegt.

Allgemein erfolgt ein Funktionsaufruf des Skripts nach dem Schema

```
partition_dataset.py -x -i [PATH_TO_IMAGES_FOLDER] -r 0.1
```

Dabei gibt das Argument **-x** an, dass die Bilder mit den zugehörigen *.xml*-Dateien verschoben werden sollen. Alternativ kann das Argument **-x** also auch weggelassen und die Bilder erst unterteilt und später gelabelt werden.

Das **-i** ist der Pfad, in welchem sich die Bilder befinden, und in dem auch schon mit *labelImg* gearbeitet wurde.

Zuletzt erwartet das Skript noch die Angabe eines Verhältnis, in welchem die Bilder in Trainings- und Testdaten unterteilt werden sollen. Die Angabe erfolgt in Dezimalbrüchen. Häufig wird hier der Wert 0.1 verwendet, sodass ein Verhältnis von 9:1 Trainings- zu Testbildern entsteht.

Für den Fall, dass ein unsauberer Verhältnis vorliegt, wie auch bei den hier verwendeten Bildern, wird die Anzahl der Testdaten immer aufgerundet. Für die 159 genutzten Bilder ergibt sich somit:

$$n_{test} \geq 0.1 \cdot 159 = 15.9 \quad \Rightarrow \quad n_{test} = 16$$

und

$$n_{train} = n_{total} - n_{test} = 159 - 16 = 143$$

Unter Berücksichtigung des oben angelegten Pfades kann das Skript daher mit dem Terminalbefehl ([Z. 113](#))

¹⁷Vladimirov, 2021c.

```
$ python /home/$USER/TensorFlow/scripts/partition_dataset.py -x -i  
/home/$USER/TensorFlow/workspaces/training_demo/images -r 0.1
```

ausgeführt werden. Damit sind die gelabelten Bilder in etwa im Verhältnis 9:1 in Training- und Testdaten aufgeteilt.

Eine kurze Anmerkung zu den unterstützten Bildtypen: In Zeile 35 des Skripts werden die Dateiendungen (**.jpg | .jpeg | .png**) gelistet. Sollten die Bilder mit einer anderen Dateiendung aufgenommen sein kann diese hier ergänzt werden.

Da TensorFlow die Label nicht als Text aus den *.xml*-Dateien verarbeiten kann wird als nächstes eine sogenannte *label map* erstellt, welche die verwendeten Label einem Integer, also einer ganzen Zahl zuordnet. Dazu wird einfach eine neue Datei mit einem Texteditor erstellt, welche mit der Dateiendung *.pbtxt* im Ordner *annotations* abgelegt wird.

Die Eingabe der Label folgt dem Schema

```
item {  
  id: 1  
  name: 'Schokoriegel'  
}  
  
item {  
  id: 2  
  name: 'MAOM'  
}
```

Die fertige Datei kann ebenfalls heruntergeladen werden: ([Z. 115](#))

```
$ wget -O /home/$USER/TensorFlow/workspaces/training_demo/annotations/label_map.pbtxt  
https://raw.githubusercontent.com/Katschka/Object-Detection-Tutorial/main/others/_  
label_map.pbtxt
```

Mit Hilfe dieser *label map* können nun die *.xml*-Dateien in von TensorFlow verwertbare Dateien, sogenannte *TFRecords*, umgewandelt werden. Dazu stellt das Tutorial¹⁸ ebenfalls ein Skript zur Verfügung, welches heruntergeladen werden kann: ([Z. 117](#))

```
$ wget -O /home/$USER/TensorFlow/scripts/generate_tfrecord.py  
https://raw.githubusercontent.com/Katschka/Object-Detection-Tutorial/main/scripts/_  
generate_tfrecord.py
```

Anschließend kann unter Zuhilfenahme des Pandas-Tools zur Datenverarbeitung ([Z. 119](#))

¹⁸Vladimirov, 2021d.

```
$ conda install pandas
```

für die Trainings- und die Testaden jeweils ein *TFRecord* erstellt werden: ([Z. 121-122](#))

```
$ python /home/$USER/TensorFlow/scripts/generate_tfrecord.py -x  
/home/$USER/TensorFlow/workspaces/training_demo/images/train -l  
/home/$USER/TensorFlow/workspaces/training_demo/annotations/label_map.pbtxt -o  
/home/$USER/TensorFlow/workspaces/training_demo/annotations/train.record  
$ python /home/$USER/TensorFlow/scripts/generate_tfrecord.py -x  
/home/$USER/TensorFlow/workspaces/training_demo/images/test -l  
/home/$USER/TensorFlow/workspaces/training_demo/annotations/label_map.pbtxt -o  
/home/$USER/TensorFlow/workspaces/training_demo/annotations/test.record
```

Nun stehen die *TFRecords* *test.record* und *train.record* im Ordner **annotations** zur Verfügung. Falls die Ordner oder die Pfade anders benannt sind kann das Skript allgemein nach dem Schema

```
python generate_tfrecord.py -x [PATH_TO_IMAGES_FOLDER]/[DATASET] -l  
[PATH_TO_ANNOTATIONS_FOLDER]/[LABEL_MAP.PBTXT] -o  
[PATH_TO_ANNOTATIONS_FOLDER]/[DATASET].record
```

aufgerufen werden.

Damit sind die Voraarbeiten für das Trainieren eines neuronalen Netzes zur Bilderkennung abgeschlossen.

4.3 Training eines Modells

Zunächst muss ein Modell aus dem *TensorFlow 2 Detection Model Zoo*¹⁹ ausgewählt werden. Da die vorliegenden Bilder ein Format von 640×480 Pixel aufweisen empfiehlt sich ein Modell mit der passenden Größe an Eingaben, beispielsweise das Modell *SSD ResNet50 V1 FPN* 640×640 (*RetinaNet50*), welches für das gewünschte Format das schnellste Modell sein soll: (Z. 124-127)

```
$ cd ~/Downloads
$ wget http://download.tensorflow.org/models/object_detection/tf2/20200711/
  ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz
$ tar -zxf ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz -C
  /home/$USER/TensorFlow/pre-trained-models
$ rm ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz
```

Ein Modell besteht aus den Ordnern *checkpoint* und *saved_model* sowie aus einer *pipeline.config*-Datei. Letztere muss in den Workspace kopiert und an die verwendete Ordnerstruktur angepasst werden, dies kann von Hand geschehen, oder die fertige Datei kann im Anschluss an die Erläuterung mit den folgenden Anpassungen heruntergeladen werden: (Z. 129-131)

```
$ cp /home/$USER/TensorFlow/pre-trained-models/
  ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/pipeline.config
  /home/$USER/TensorFlow/workspaces/training_demo/model-config-files/
  ssd_resnet50_v1_fpn_640x640_coco17_tpu-8
$ cd /home/$USER/TensorFlow/workspaces/training_demo/pre-trained-models/
  ssd_resnet50_v1_fpn_640x640_coco17_tpu-8
$ nano pipeline.config
```

- In Zeile 3 muss **num_classes** auf die Anzahl der verwendeten Label, hier also 4 (Schokoriegel, MAOAM, MilkyWay und Snickers) geändert werden
- In den Zeilen 6 und 7 kann die Bildgröße spezifiziert werden, hier also 480×640 Pixel
- In Zeile 131 kann die **batch_size** angepasst werden: Ein kleinerer Wert benötigt weniger Speicher, für die hier geschilderte Umgebung wurde ein Wert von 1 gewählt
- In Zeile 161 wird der Pfad zum Checkpoint (für den aktuellen User *Dennis*) angegeben:
/home/dennis/TensorFlow/pre-trained-models/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/checkpoint/ckpt-0

¹⁹https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

- In Zeile 167 wird der Typ von **classification** zu **detection** geändert
- In Zeile 168 wird der Wert auf **false** gesetzt (Eine TPU ist ein spezieller Prozessor für maschinelles Lernen)
- In Zeile 172 wird der Pfad zur *label map* angegeben: **/home/dennis/TensorFlow/workspaces/training_demo/annotations/label_map.pbtxt**
- In Zeile 174 wird der Pfad zum Trainings-record angegeben: **/home/dennis/TensorFlow/workspaces/training_demo/annotations/train.record**
- In den Zeilen 182 und 186 werden analog der *label map*-Pfad und der Pfad zum Test-record angegeben

Die Datei kann wieder mit **Strg + x** beendet werden, dabei ist das Speichern zu bestätigen. Achtung: Je nach gewähltem Modell können die Bezeichnungen und die Zeilenummern geringfügig von obigen angaben abweichen! Eine fertige *.config*-Datei kann nicht automatisch auch für andere Modelle genutzt werden.

Außerdem kann die Bearbeitung in einem etwas größerem Texteditor mit einer Syntaxhervorhebung, also einer Anpassung der Textfarbe, hilfreich sein: Die Änderungen nach Eingabe der **batch_size** sind als Strings einzugeben und werden daher in der Regel entsprechend farblich hervorgehoben.

Eine weitere Anmerkung: Es gibt eine Einstellung mit der Bezeichnung *max_number_of_boxes*, welche für die vorliegenden Bilder nicht verändert werden muss, jedoch standardmäßig auf 100 steht. Für das gewählte Modell ist diese Option in Zeile 165 hinterlegt.

Zuletzt sei auf die maximale Anzahl an Schritten hingewiesen, welche das Training durchläuft: **num_steps** findet sich in Zeile 162 und steht für das gewählte Modell standardmäßig auf 25,000. Sollte das TensorBoard am Ende des Trainings noch immer deutlich sinkende Kosten zeigen, welche sich jedoch noch keinem Grenzwert annähern, kann die Anzahl an Schritten vergrößert werden.

Die fertig konfigurierte Datei kann mit (Z. 133)

```
$ wget -O /home/$USER/TensorFlow/workspaces/training_demo/model-config-files/`  
ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/pipeline.config  
https://raw.githubusercontent.com/Katschka/Object-Detection-Tutorial/main/others/`  
pipeline.config
```

heruntergeladen werden. Zum trainieren des Modells wird nun ein fertiges Skript von TensorFlow verwendet, welches dazu in den Projektordner *training_demo* kopiert wird: (Z. 135)

```
$ cp /home/$USER/TensorFlow/models/research/object_detection/model_main_tf2.py  
/home/$USER/TensorFlow/workspaces/training_demo
```

Dies ist sinnvoll, da in der Konfiguration nur relative Pfade angegeben werden, welche sich auf den Speicherort des Skripts als Bezugspunkt beziehen. Würde das Skript an einer anderen Stelle gespeichert, müssten auch die Pfade in der Konfiguration und beim aufrufen des Skripts angepasst werden.

In einem neuen Terminal kann anschließend das Training des Modells gestartet werden: ([Z. 137-138](#))

```
$ cd /home/$USER/TensorFlow/workspaces/training_demo  
$ python model_main_tf2.py  
--model_dir=trained-models/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8  
--pipeline_config_path=model-config-files/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/]_  
pipeline.config
```

Zum Zeitpunkt der Arbeit tritt hierbei die Fehlermeldung *NotImplementedError: Cannot convert a symbolic Tensor (cond_2/strided_slice:0) to a numpy array. This error may indicate that you're trying to pass a Tensor to a NumPy call, which is not supported* auf, welche auf eine fehlerhafte Funktion innerhalb von TensorFlow zurückzuführen ist. Sollte das Training eines Modells durchgeführt werden, solange hierzu noch kein Update zur Verfügung steht, kann mit folgendem Befehl eine manuell gepatchte Version des Fehlerhaften Skripts eingespielt werden, die Originaldatei wird als Backup beibehalten: ([Z. 140-141](#))

```
$ cp /home/$USER/anaconda3/envs/ai-ve/lib/python3.9/site-packages/tensorflow/python/]_  
ops/array_ops.py  
/home/$USER/anaconda3/envs/ai-ve/lib/python3.9/site-packages/tensorflow/python/ops/]_  
backup_array_ops.py  
$ wget -O /home/$USER/anaconda3/envs/ai-ve/lib/python3.9/site-packages/tensorflow/]_  
python/ops/array_ops.py  
https://github.com/Katschka/Object-Detection-Tutorial/raw/main/others/array_ops.py
```

Ein weiteres mögliches Problem ist, dass das Training mit der Fehlermeldung *Resource exhausted* abgebrochen wird. Das bedeutet, dass nicht genügend **GPU**-Speicher zur Verfügung steht. In diesem Fall können mehrere Möglichkeiten genutzt werden:

- Die **batchsize** verringern
- Falls mit vertretbarem Aufwand möglich die Größe **fixed_shape_resizer** verringern

- Ein anderes Modell auswählen

Gerade, wenn auf dem Gerät bereits einige Zeit gearbeitet wurde, kann auch ein Neustart oder das Beenden von speicherintensiven Prozessen dazu beitragen, dass der Fehler vermieden wird.

Der Befehl

```
$ nvidia-smi
```

kann hierbei helfen, da er alle aktuellen Prozesse auf der **GPU** und deren Speichernutzung auflistet.

Wenn kein weiterer Fehler auftritt können einige Warnungen angezeigt werden, insbesondere *Deprecation Warnings*. Diese können jedoch im Regelfall ignoriert werden und zeigen an, dass in absehbarer Zukunft bestimmte Funktionen nicht mehr unterstützt werden. Solche Warnungen dienen daher primär den Entwicklern zur Anpassung ihres Quellcodes.

Nach einigen Lademeldungen und wahrscheinlich auch Warnungen fängt das eigentliche Training an. Im Terminal werden nun im Abstand einiger Sekunden bis Minuten (die genaue Dauer hängt wieder von Modell, Batchgröße, Hardware etc. ab) Informationen zum *Loss* und zur Lernrate angegeben. Prinzipiell gilt hier, je geringer der *Loss* ist, um so besser ist das Modell trainiert. Auf die Thematik wird in [Abschnitt 6](#) näher eingegangen.

Der Trainingsfortschritt kann auch grafisch überwacht werden. Dazu kann in einem zweiten Terminal mit dem Befehl ([Z. 143-144](#))

```
$ cd /home/$USER/TensorFlow/workspaces/training_demo
$ tensorboard --logdir=trained-models/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8
```

das sogenannte TensorBoard aktiviert werden. Nun kann ein beliebiger Browser geöffnet und die Adresse <http://localhost:6006/> eingegeben werden. Es öffnet sich das TensorBoard, welches unter anderem den Verlauf der Terminalausgaben aufarbeitet, wie [Abbildung 6](#) zeigt:



Abbildung 6: Aktiviertes TensorBoard

Leider ist dieses TensorBoard nicht Live. Um den aktuellen Trainingsverlauf anzuzeigen muss das TensorBoard über den entsprechenden Button rechts in der orangefarbenen Kopfzeile aktualisiert werden.

Alternativ kann unter den Einstellungen (das Zahnradsymbol oben Rechts) eine Aktualisierungsrate der Oberfläche ausgewählt werden.

Wie das Beispiel außerdem zeigt, ist der Verlauf des *Loss*, der übrigens auch häufig als *Kosten* bzw. im englischen entsprechend als *Cost* bezeichnet wird, nicht stetig fallend. In der Praxis kommt es im Gegenteil nahezu immer zu kürzeren Perioden des Anstiegs der Kosten.

Wichtig ist, dass der Trend eine fallende Funktion darstellt, weshalb das TensorBoard am linken Rand auch die Möglichkeit bietet, die Graphen zu glätten. Dies macht es unter Umständen einfacher, den Trend zu erkennen.

Insgesamt schließt dieses Modell das Training nach 01:04:48 Stunden ab und erreicht einen *total_loss* von 0.136. Zum Vergleich werden im [Abschnitt 6.1](#) noch weitere Modelle auf den hier verwendeten Demo-Daten trainiert und die Ergebnisse verglichen.

4.4 Anwendung des trainierten Modells

Nach abgeschlossenem Training kann eine weitere Funktion von TensorFlow genutzt werden, um das Modell zu exportieren. Dazu wird die Funktion, analog zum Trainingsaufruf, in den Arbeitsbereich kopiert: (Z. 146)

```
$ cp /home/$USER/TensorFlow/models/research/object_detection/exporter_main_v2.py  
/home/$USER/TensorFlow/workspaces/training_demo
```

Diese Funktion kann nun mit dem Befehl (Z. 148-149)

```
$ cd /home/$USER/TensorFlow/workspaces/training_demo  
$ python exporter_main_v2.py --input_type image_tensor --pipeline_config_path  
model-config-files/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/pipeline.config  
--trained_checkpoint_dir trained-models/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8  
--output_directory exported-models/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8
```

aufgerufen werden. Anschließend steht das Modell als *saved_model* im *output_directory*-Ordner zur Verfügung. Das fertige Modell kann nun getestet werden, wobei das folgende Skript genutzt werden kann: (Z. 151-152)

```
$ wget -O /home/$USER/TensorFlow/scripts/trained_object_detection.py  
https://github.com/Katschka/Object-Detection-Tutorial/raw/main/scripts/_  
trained_object_detection.py  
$ python /home/$USER/TensorFlow/scripts/trained_object_detection.py
```

Das Skript nutzt in der Standardeinstellung die in dieser Anleitung dargelegten Pfade und Dateinamen, kann jedoch auch für beliebige andere Pfade verwendet werden. Zur Veranschaulichung des Ergebnis wurden fünf zufällig ausgewählte Trainingsbilder in den *images-to-detect*-Ordner kopiert.

Nach der Verarbeitung öffnen sich die Bilder in einzelnen Fenster, wie Abbildung 7 zeigt:

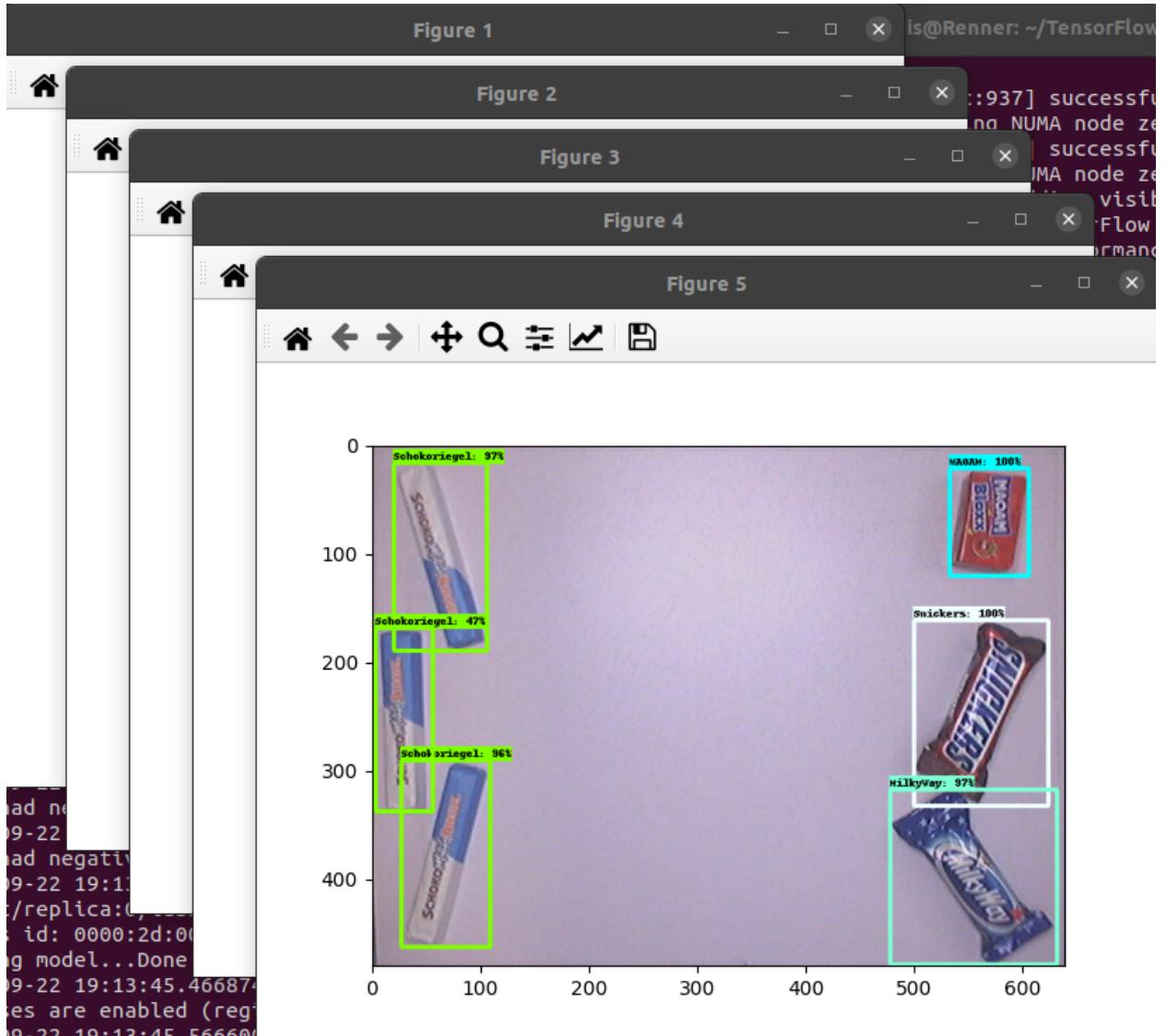


Abbildung 7: Ausgabe des trainierten Modells für fünf zufällig ausgewählte Bilder

Allgemein bietet das Skript folgende Optionen an:

```
trained_object_detection.py [-h] [-f FILETYPE] [-i IMAGE_PATH] [-l LABEL_PATH] [-m
    MODEL_PATH] [-o OUTPUT_PATH] [-s SUPPRESS_SHOWING]
```

- **-h** zeigt die Hilfstexte zu allen Optionen an
- **-f** fragt den Datentyp der Bilder ab, in denen Objekte erkannt werden sollen, also beispielsweise **jpg**, **png** etc.
- **-i** erwartet die Angabe eines Ordnerpfades, in welchem die zu verarbeitenden Bilder liegen, in dieser Anleitung **/home/\$USER/TensorFlow/workspaces/training_demo/images/images-to-detect**

- **-l** erwartet die Angabe zu Ort und Namen der Labelmap, also `/home/$USER/TensorFlow/workspaces/training_demo/annotations/label_map.pbtxt`
- **-m** fragt den Ordner ab, in welchem das trainierte und exportierte Modell mit dem Namen `saved_model.pb` liegt, in diesem Fall also `/home/$USER/TensorFlow/workspaces/training_demo/exported-models/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/saved_model`
- **-o** erwartet die Angabe eines Ordnerpfades, unter welchem die Detektionen gespeichert werden können, hier `/home/$USER/TensorFlow/workspaces/training_demo/images/detection`
- **-s** ist ein Flag, welches die Anzeige der Bilder in einzelnen Fenstern auf dem Desktop deaktiviert, beispielsweise wenn die Option **-o** genutzt wird

Der vollständige Funktionsaufruf für dieses Tutorial wäre demnach ([Z. 154](#))

```
$ python /home/$USER/TensorFlow/scripts/trained_object_detection.py -f jpg -i  
/home/$USER/TensorFlow/workspaces/training_demo/images/images-to-detect -l  
/home/$USER/TensorFlow/workspaces/training_demo/annotations/label_map.pbtxt -m  
/home/$USER/TensorFlow/workspaces/training_demo/exported-models/  
ssd_resnet50_v1_fpn_640x640_coco17_tpu-8/saved_model
```

Nach Eingabe des Befehls werden im Terminal einige Ausgaben zum Laden von TensorFlow, der **GPU**, des Modells etc. angezeigt, bevor abschließend die im Ordner *images-to-detect* abgelegten Bilder mit dem Vermerk *Running inference for* der Reihe nach bearbeitet werden.

Das Modell dabei auf bereits zum Training genutzten Bildern laufen zu lassen dient vor allem der Kontrolle, ob das Modell auf den Trainingsdaten korrekt angelernt wurde. In der tatsächlichen Anwendung bietet sich hier eine Kontrollgruppe von dem Modell noch unbekannten Bildern an.

Hierfür wurden zwei weitere Bilder aufgenommen, wobei das erste Bild *on_paper* stark an die bereits bekannten Trainingsbilder angelehnt ist. Das zweite Bild, *on_wood*, zeigt das gleiche Motiv, jedoch auf einem nicht mehr neutralen Hintergrund.

Beide Bilder wurden zunächst mit dem Skript *resize_images* verkleinert und anschließend vom vorher trainierten Modell verarbeitet. Die Detektion auf beiden Bildern zeigt jedoch deutliche Abweichungen voneinander, wie der Vergleich von [Abbildung 11](#) mit [Abbildung 12](#) in Anlage A offenbart.

Dies ist wahrscheinlich darauf zurückzuführen, dass in den Trainingsbildern der Hintergrund immer gleich ist, das Modell also lernt, dass der deutliche Kontrast zum Hintergrund ein

Identifikationsmerkmal darstellt, welches auf dem Holztisch durch die Maserung nicht mehr gegeben ist. Dieses Phänomen, dass zu viele Dinge als vermeintlich gesuchtes Objekt erkannt werden, wird auch als Unteranpassung bezeichnet.

Außerdem zeigt selbst das auf Papier aufgenommene Bild einige grobe Fehler, welche darauf hindeuten, dass das Modell noch nicht mit ausreichender Qualität trainiert ist.

Wie das Modell möglicherweise weiter verbessert werden kann wird daher im Abschnitt [6](#) erläutert.

5 Installation auf alternativen Systemen

Wie bereits in [Abschnitt 2](#) beschrieben ist diese Anleitung auch auf andere Hard- und Software übertragbar. Im Rahmen der Arbeit wurde ebenfalls eine Installation in den Laboren der Hochschule durchgeführt.

Dabei wird die beschriebene Installation unter Ubuntu 16.04 durchgeführt, als [GPU](#) kommt eine NVIDIA RTX2080 Super zum Einsatz. Durch das geänderte Betriebssystem sind einige wenige Anpassungen nötig, welche in folgendem erläutert werden sollen.

Zunächst ist festzustellen, dass in dieser älteren Version von Ubuntu standardmäßig die Python-Version 3.5 installiert ist, welche jedoch nicht mit der verwendeten [CUDA](#)- und TensorFlow-Version kompatibel ist. In diesem Fall ist also die Installation des aktuellen Pythons, zu diesem Zeitpunkt Python 3.9.7 obligatorisch.

Anschließend wird Anaconda in der gleichen Version installiert. Dabei ist aufgefallen, dass auf dem vorliegenden System kein Temp-Ordner vorhanden ist, weshalb der Download-Befehl leicht angepasst wird ([Z. 156-157](#)):

```
$ wget /Downloads https://repo.anaconda.com/archive/Anaconda3-2021.05-Linux-x86_64.sh  
$ bash /Downloads/Anaconda3-2021.05-Linux-x86_64.sh
```

Zur anschließenden Installation von TensorFlow in der bereits vorher genutzten Version 2.5 wird zunächst der [PIP](#) aktualisiert ([Z. 159](#)):

```
$ pip install --upgrade pip
```

Danach kann die Installation wie in [Abschnitt 2](#) beschrieben fortgesetzt werden ([Z. 13-19](#)). Wie oben auch wird die TensorFlow-Version 2.5 installiert.

Aufgrund der geänderten Ubuntu-Version muss beim Download von [CUDA](#) gemäß der Befehle in Zeile 21ff der Text *ubuntu2004* in *ubuntu1604* geändert werden. Die angepassten Befehle lauten damit ([Z. 161-167](#))

```
$ sudo apt-get install linux-headers-$(uname -r)^~I  
$ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/]  
  cuda-ubuntu1604.pin  
$ sudo mv cuda-ubuntu1604.pin /etc/apt/preferences.d/cuda-repository-pin-600  
$ sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/]  
  repos/ubuntu1604/x86_64/7fa2af80.pub  
$ sudo add-apt-repository "deb  
  https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/ /"  
$ sudo apt-get update
```

```
$ sudo apt-get -y install cuda
```

Die verwendete **CUDA**-Version ist in dieser Konfiguration 11.3, die vorher verwendete Version 11.4 ist mit Ubuntu 16.04 nicht kompatibel. Daher werden auch die Einträge in der Systemdatei ([Z. 29](#))

```
$ nano ~/.bashrc
```

auf diese Version angepasst: ([Z. 169-172](#))

```
# CUDA related exports
export PATH=/usr/local/cuda-11.3/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-11.3/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export PATH=/home/$USER/Protobuf/bin${PATH:+:${PATH}}
```

Die nächsten Schritte bis zum Download der **cuDNN**-Bibliothek von der NIDIA-Webseite bleiben unverändert ([Z. 36-50](#)). Analog zur älteren **CUDA**-Version wird auch eine etwas ältere Version von **cuDNN** installiert: Version 8.2.1 anstatt 8.2.2.

Beim anschließenden Test der Installation mittels ([Z. 52](#))

```
$ python -c "import tensorflow as tf;print(tf.reduce_sum(tf.random.normal([1000,
1000])))"
```

wurde festgestellt, dass die Bibliothek *libcudnn-so-8* fehlt. Diese kann jedoch mit zwei einfachen Befehlen nachinstalliert werden ([Z. 174-175](#)):

```
$ sudo apt-get install libcudnn8=8.2.1.*-1+cuda11.3
$ sudo apt-get install libcudnn8-dev=8.2.1.*-1+cuda11.3
```

Anschließend können die weiteren Schritte bis zur Ausführung des Demoskriptes ([Z. 54-90](#)) wie beschrieben durchlaufen werden. Bei der Ausführung des Skripts treten jedoch zwei Fehler auf.

Der erste Fehler enthält die Meldung *You are using ptxas 8.x, but TF requires ptxas 9.x*. Gleichzeitig zeigt der Befehl

```
$ nvcc -V
```

eine veraltete **CUDA**-Version an, trotz vorheriger Installation der korrekten Version. Hier kann das System einfach auf die korrekte Version hingewiesen werden ([Z. 177](#)):

```
$ export PATH=/usr/local/cuda-11.3/bin:$PATH
```

Der zweite Fehler bei der Ausführung des Demo-Skripts besagt *qt.qpa.plugin: Could not load the QT platform plugin "xcb" in " "* even though it was found. Hier hilft ebenfalls eine einfache Nachinstallation der zugrundeliegenden Bibliotheken ([Z. 179-180](#)):

```
$ sudo apt-get install libxkbcommon-x11-0  
$ sudo apt install libxcb-xinerama0
```

Nach diesen Installationen kann mit dem Training gemäß [Abschnitt 4 \(ab Z. 92\)](#) fortgefahrene werden.

6 Möglichkeiten zur Verbesserung der Objekterkennung

Im Folgenden werden verschiedene Möglichkeiten zur Verbesserung der Detektionen vorgestellt, erläutert und getestet. Die dabei erstellten und exportierten Modelle finden sich ebenfalls in der Hochschulcloud, die Trainingsdaten, Einstellungen und Ergebnisse im github repository.

Außerdem findet sich im github-Ordner *others* eine Datei mit allen Befehlen für die verwendeten Modelle. Aufgrund der oben definierten Ordnerstruktur können diese Befehle ausgeführt werden, sobald im Terminal in den korrekten *workspace* gewechselt wurde. Lediglich beim Detektieren mit den Modellen müssen die **[workspace]**-Einträge in der Datei angepasst werden. Zur Sicherheit enthält der Kopf der Datei nochmals eine Anleitung.

6.1 Die Nutzung verschiedener Modelle

Der *TensorFlow 2 Detection Model Zoo* bietet neben dem in [Abschnitt 4.3](#) genutzten Modell noch weitere bereits vortrainierte Modelle, welche über die passende Eingabegröße verfügen.

Da diese CNNs im Prinzip beliebig komplex gestaltet werden können, hat die Wahl des Modells einen großen Einfluss auf das Ergebnis des Trainings, aber natürlich auch auf die benötigte Dauer zum anlernen des Modells.

Die intuitive Annahme ist, dass ein komplexeres Modell auch bessere Ergebnisse erzielt, da es durch eine höhere Komplexität in der Lage ist, mehr Freiheitsgrade zu verarbeiten und damit aus den Trainingsdaten mehr Kriterien zur Identifikation der Objekte gewinnen kann.

Leider wird seitens der TensorFlow-Entwickler keine direkte Angabe zur Komplexität oder zum Aufbau der unterschiedlichen Modelle gemacht, daher wird das Training analog zum [Abschnitt 4.3](#) mit den passenden Modellen wiederholt und die Ergebnisse verglichen.

Um einen Vergleich der Modelle zu ermöglichen werden alle Modelle mit der gleichen Anzahl von 25,000 Schritten trainiert. Auf dieser Basis lassen sich anhand der TensorBoard-Ausgaben in [Anlage B.1](#) die benötigte Trainingszeit und die erreichten Kosten vergleichen, wobei beide Werte nur erste Indizien für die Qualität der Ergebnisse darstellen. Die Werte finden sich in [Tabelle 1](#):

Tabelle 1: TensorBoard-Ausgaben für den Vergleich der Modelle

Netzwerk	Zeit / hh:mm:ss	Kosten
efficientdet_d1	02:05:25	0.165
faster_rcnn_inception_resnet_v2	03:23:29	1.11e-4
faster_rcnn_resnet50_v1	00:57:05	1.34e-4
faster_rcnn_resnet101_v1	01:21:11	3.0e-5
ssd_mobilenet_v1_fpn	00:48:46	0.5549
ssd_mobilenet_v2_fpnlite	00:50:39	0.1259
ssd_resnet50_v1_fpn	00:52:40	0.0939
ssd_resnet101_v1_fpn	01:12:39	0.1349

Die Minima wurden jeweils in Grün, die Maxima in Rot hervorgehoben. Für das *ssd_mobilenet_v1_fpn* zeigt sich, dass die höchste Geschwindigkeit scheinbar auf Kosten der Qualität geht, da die verbleibenden Kosten des Modells mit deutlichem Abstand am höchsten sind. Das bedeutet vereinfacht, dass dieses Modell auf den Testdaten am meisten Fehler macht.

Positiv überraschend ist, dass das Netzwerk *faster_rcnn_resnet101_v1* mit den geringsten Kosten trainiert werden konnte, aber bei der Geschwindigkeit nicht das langsamste Netzwerk ist; das Netzwerk *faster_rcnn_inception_resnet_v2* benötigt ca. 2.5 mal so lange, um zu einem minimal schlechterem Ergebnis zu gelangen.

Im Vergleich mit den Angaben auf der github-Seite von TensorFlow zeigt sich außerdem, dass die von TensorFlow genannten Geschwindigkeiten nicht mit den hier ermittelten korrelieren und das zur Demonstration genutzte Modell entgegen der Erfahrungen von TensorFlow in der hier genutzten Umgebung nicht das schnellste Modell ist.

Sofern die benötigte Trainingszeit nicht relevant ist kann als nächstes überprüft werden, ob die Modelle sich überhaupt dem Grenzwert der Kostenfunktion angenähert haben. Die Abbildungen in Anlage B.1 zeigen, dass die Kostenfunktionen sich alle bereits deutlich der Konstanten unteren Grenze genähert haben, dennoch scheint bei den Modellen *efficientdet_d1*, *faster_rcnn_inception_resnet_v2* und *faster_rcnn_resnet101_v1* noch eine geringe Verbesserungsmöglichkeit zu bestehen.

Insbesondere das *efficientdet_d1*-Modell zeigt eine Kostenfunktion mit deutlichem Ausreißer vor dem Ende der Trainingsschritte, was auf Verbesserungspotential hindeutet. Leider sind solche Spitzen nicht zu vermeiden, weshalb eine Kontrolle über das TensorBoard dringend zu empfehlen ist, um einen potentiellen Ausreißer nach dem letzten Trainingsschritt zu erkennen.

Insgesamt ist das Verbesserungspotential jedoch sehr gering und ändert auch bei voller Aus-

schöpfung nichts daran, dass die geringsten Kosten beim Modell *faster_rcnn_res-net101_v1* auftreten.

In der Praxis zeigt sich jedoch, dass die geringsten Kosten alleine kein hinreichender Beweis für ein gut trainiertes und vor allem brauchbares Netzwerk sind. Aus diesem Grund werden die Ergebnisse der Detektion auf einigen Testbildern betrachtet.

Dazu wurden zunächst aus den Trainingsbildern drei zufällige Bilder entfernt. Die trainierten Modelle werden anschließend auf insgesamt 19 Bildern angewendet:

- drei Bilder, auf denen das Modell auch trainiert wurde
- drei Bilder, welche aus den Trainingsdaten entfernt wurden - die Bildbedingungen entsprechen damit Laborbedingungen
- 13 hochauflösenden Bildern (die Trainingsbilder waren im Format 640×480 Pixel) mit jeweils verschiedenen Hintergründen

Insgesamt sind auf den verwendeten Bildern 151 Objekte zu detektieren.

Zum einen ist interessant, wie gut die Transferleistung der Modelle auf neue Umgebungen oder Hintergründe ist, zum anderen existieren verschiedene Fehlertypen, welche je nach Einsatzgebiet unterschiedlich stark gewichtet werden müssen, wie gleich erläutert wird.

Die Ergebnisse werden dazu in fünf Kategorien eingeteilt, auch wenn die Übergänge teilweise fließend sind:

- E1: Erfolgreich erkannt: Ein Objekt erkannt, richtig kategorisiert und richtig positioniert (den Rahmen also an der richtigen Position in der korrekten Größe eingezeichnet)
- E2: Fehlerhaft erkannt: Ein Objekt erkannt, richtig positioniert, aber falsch kategorisiert
- E3: Fehlerhaft positioniert: Ein Objekt erkannt, richtig kategorisiert, aber falsch positioniert
- E4: Nicht erkannt: Ein vorhandenes Objekt wurde nicht erkannt
- E5: Übererkannt: Ein nicht vorhandenes Objekt wird angezeigt

Welche der Fehler wie gewichtet werden müssen hängt von der gewünschten Anwendung ab. Die Grundintention hinter der Objekterkennung von Süßigkeiten liegt, wie bereits in der

Einleitung beschrieben, in der Unterstützung von sehbehinderten Menschen in einer autarken Lebensführung.

Lautet also beispielsweise die Anweisung eines blinden Menschen "Reiche mir ein MilkyWay", und das Modell findet lediglich kein MilkyWay, entsteht zumindest kein Schaden, die Fehlergewichtung ist also gering. Erkennt das Modell jedoch ein Snickers falsch als MilkyWay kann dies bei einer Erdnussallergie des Empfängers im schlimmsten Fall zum Tode führen, dieser Fehler muss also sehr stark gewichtet werden.

Die Ergebnisse auf den Testbildern in Anlage B werden in der folgenden Tabelle zusammengefasst:

Tabelle 2: Ergebnisse des Modellvergleichs auf den Testbildern

Netzwerk	E1	E2	E3	E4	E5
efficientdet_d1	133	3	6	11	7
faster_rcnn_inception_resnet_v2	135	0	3	13	6
faster_rcnn_resnet50_v1	126	5	9	15	5
faster_rcnn_resnet101_v1	139	4	6	5	3
ssd_mobilenet_v1_fpn	80	7	13	52	21
ssd_mobilenet_v2_fpnlite	65	3	2	81	18
ssd_resnet50_v1_fpn	68	19	26	42	999
ssd_resnet101_v1_fpn	68	22	35	26	999

Wie [Tabelle 2](#) zeigt, schneiden das *efficientdet_d1*-Modell sowie die *faster_rcnn*-Netze recht gut ab, die *ssd*-Modelle jedoch erzeugen zahlreiche Fehler.

Das Netzwerk *faster_rcnn_resnet_101* weiß nicht nur die geringsten Kosten auf, sondern zeigt auch die wenigsten Fehler. Die Tatsache, dass die beiden anderen *faster_rcnn*-Modelle in etwa die gleichen, vier mal höheren Kosten ausweisen, und dennoch das zweite *resnet*-Modell nur wenig mehr Objekte nicht korrekt erkannt hat untermauert erneut, dass die Kosten nur ein erstes Indiz sein dürfen, um unterschiedliche Modelle zu bewerten.

Gegenüber den sehr guten *rcnn*-Netzwerken stehen die deutlich schlechter abschneidenden *ssd*-Netzwerke. Die beiden *mobile*-Modelle sind in 34% bzw. 54% der Fälle nicht in der Lage, die Objekte überhaupt zu erkennen. Die *resnet*-Varianten erkennen im Gegensatz dazu sehr viel mehr Objekte, als tatsächlich vorhanden sind - dies stellt wieder eine Unteranpassung dar, ähnlich wie am Ende von [Abschnitt 4.4](#). Aufgrund der hohen Anzahl an fälschlicherweise angezeigten Objekte wurde in [Tabelle 2](#) der Wert 999 eingetragen, da ein tatsächliches Zählen nicht mehr sinnvoll möglich war.

Weiterhin ist auffällig, dass die *ssd*- und insbesondere *ssd_resnet*-Modelle häufig dazu tendieren, MilkyWays oder Schokoriegel als Snickers falsch zu klassifizieren. Unter der Eingangs erwähnten Intention der möglichen Nutzung durch sehbehinderte Menschen ist eine solche Fehlklassifikation ein absolutes Ausschusskriterium, da dies aufgrund möglicher Allergene eine potentiell tödliche Folge haben kann.

Auf Grundlage der in diesem Abschnitt ermittelten Daten sollen nun weitere Möglichkeiten zur Verbesserung der neuronalen Netzwerke erarbeitet, getestet und diskutiert werden.

6.2 Der Einfluss der Auflösung auf die Detektion

Die Annahme für diesen Abschnitt lautet, dass eine bessere Auflösung auch zu einem besseren Ergebnis bei der Detektion führt. Dies wird analog zum Menschen erwartet, welcher auf Bildern mit höherer Auflösung auch Objekte mit mehr Details erkennen kann. Im gleichen Maße ist davon auszugehen, dass ein neuronales Netzwerk auf Bildern mit höherer Auflösung mehr Möglichkeiten findet, ein Objekt zu identifizieren und zu kategorisieren.

Um diese These zu überprüfen wurde ein neuer Datensatz an Bildern auf neutralem, weißen Hintergrund erstellt, welche jedoch über eine hohe Auflösung von 4032×3024 Pixel verfügen. Die Bilder wurden mit Hilfe des Python Skriptes *partition_dataset* aufgeteilt in 130 Trainings- und 15 Testbilder.

Dieser und die im späteren Verlauf genutzten Datensätze finden sich ebenfalls im git repository.

Da die Leistung der vorhandenen Grafikkarte nicht ausreicht, um die Bilder in der ursprünglichen Größe zu verarbeiten, wurden die Bilder mittels des Skripts *resize_images* zunächst auf 1280×960 Pixel verkleinert.

Auf diesen Daten werden die drei Netze *faster_rcnn_resnet101_v1*, *ssd_mobilenet_v2_fpnlite* und *ssd_resnet101_v1_fpn* trainiert, die beiden *resnet*-Modelle in der Auflösungsvariante 1024×1024 , also mit einer höheren Komplexität des Modells.

Für das *mobile*-Netz existiert leider aktuell keine höhere Auflösung für die Eingabe, weshalb hier auch das Modell für 640×640 Pixel genutzt wird. In allen drei Modellen wird die *image_resizer*-Option auf die verwendeten 1280×960 Pixel gestellt.

Sämtliche genutzten Daten und Einstellungen finden sich ebenfalls im github-repository.

Um die Vergleichbarkeit zu ermöglichen wurde die Anzahl der Schritte für die beiden höher auflösenden Netze wieder auf 25,000 Stück reduziert, die Originaleinstellung im *TensorFlow Model Zoo* wäre 100,000.

Außerdem wurden die Bilder für den Vergleich zusätzlich auf die Größe von 640×480 Pixel verkleinert und mit den drei Netzwerken in der 640×640 Pixel Version verarbeitet. Die Bilder aus dem Tutorial und dem Modellvergleich werden **nicht** genutzt, da sonst die Vergleichbarkeit nicht mehr gegeben wäre.

Somit werden die selben Bilder durch die selben Modelle verarbeitet, lediglich die genutzte Auflösung und damit auch die Komplexität der Modelle unterscheidet sich.

Die TensorBoard-Ausgaben zu diesen Trainings finden sich in Anlage C.1 sowie in Anlage D.1 und werden in Tabelle 3 gegenübergestellt:

Tabelle 3: TensorBoard-Ausgaben für verschiedene Auflösungen der gleichen Bilder

Netzwerk	Bilder 1280×960		Bilder 640×480	
	Zeit / hh:mm:ss	Kosten	Zeit / hh:mm:ss	Kosten
faster_rcnn_resnet101	03:44:03	0.0876	01:22:31	1.93e-3
ssd_mobilenet	03:59:00	0.1420	00:48:51	0.2155
ssd_resnet101	06:31:21	0.1409	01:12:47	0.2299

Die Ausgaben bezogen auf die Modelle entsprechen in etwa dem, was bereits im Modellvergleich in Abschnitt 6.1 ausgegeben wurde. Der einzige Wert, der etwas überrascht, ist die hohe Laufzeit des *ssd_resnet* für die höher auflösenden Bilder. Davon abgesehen zeigt sich auch hier das gleiche Schema wie oben.

Hinsichtlich der Kostenverteilung zeigen die Ausgaben jedoch keinen eindeutigen Vorzug für eine der beiden Auflösungen. So sind die Kosten beim *faster_rcnn_resnet* auf den niedriger auflösenden Bildern zwar deutlich geringer, dafür aber bei den beiden *ssd*-Netzen deutlich höher.

Somit ist, genau wie auch beim Modellvergleich, eine Betrachtung der tatsächlich stattfindenden Detektionen das ausschlaggebende Mittel, um eine Bewertung vornehmen zu können. Hierzu werden insgesamt 20 Bilder verwendet:

- drei hochauflösende Bilder, auf denen das Modell auch trainiert wurde
- drei hochauflösende Bilder, welche aus den Trainingsdaten entfernt wurden
- zwei Bilder aus dem Demo-Datensatz mit niedriger Auflösung
- 13 hochauflösenden Bilder mit jeweils verschiedenen Hintergründen

Auf diesen Bildern sind insgesamt 163 Objekte zu erkennen.

Für die Detektionen gemäß der Anlagen **C** und **D** ergeben sich die Ergebnisse gemäß [Tabelle 4](#):

Tabelle 4: Detektionsergebnisse für verschiedene Auflösungen der gleichen Bilder

Netzwerk	E1	E2	E3	E4	E5
faster_rcnn_resnet101 HD	97	0	0	66	0
ssd_mobilenet_v2 HD	80	15	3	71	18
ssd_resnet101_v1 HD	67	9	15	74	999
faster_rcnn_resnet101 ld	125	10	5	28	1
ssd_mobilenet_v2 ld	77	16	4	73	19
ssd_resnet101_v1 ld	78	11	5	71	17

Für das *ssd_mobilenet* zeigt sich kein nennenswerter Unterschied zwischen den verwendeten Bildern. Da dieses Modell auf beiden Datensets mit der gleichen Eingabegröße trainiert wurde deutet dies darauf hin, dass die Auflösung der Bilder, auf denen die Detektionen stattfinden, keinen relevanten Einfluss hat, solange eine Untergrenze noch nicht unterschritten wurde. Dabei ist davon auszugehen, dass eine für den Menschen ausreichende Auflösung zur Erkennung durch das Modell ebenfalls noch ausreichend ist.

Für das *ssd_resnet*-Modell zeigt sich deutlich, dass die Verwendung eines Modells mit höherer Komplexität hier zu einer Verschlechterung führt. Dies entspricht nicht der zunächst erwarteten Verbesserung des Modells.

Bei Betrachtung der Detektionen in Anlage **C** zeigt sich deutlich, dass das trainierte Netzwerk nicht nur nicht in der Lage ist, eine Transferleistung auf unbekannte Hintergründe zu erbringen, sondern dass sehr viele Objekte angezeigt werden, die gar nicht existieren, weshalb eine Zählung der Fehler wieder abgebrochen und in [Tabelle 4](#) der Wert 999 eingetragen wurde.

Ein ähnliches Ergebnis zeigt auch das *faster_rcnn_resnet*, welches für die höher auflösenden Bilder ebenfalls eine etwas geringere Transferleistung zeigt. Analog zum Modellvergleich zeigt das Modell für beide Auflösungen insgesamt jedoch kaum falsche Detektionen bzw. Detektionen für gar nicht vorhandene Objekte.

Die fehlende Transferleistung insbesondere auf die Hintergründe mit wenig kontrastreichen Kanten und Linien wie bei der Fußmatte oder dem Granit deuten auf eine Überanpassung der Modelle hin. Die Tatsache, dass die komplexeren Modelle eine noch geringere Transferleistung zeigen spricht ebenfalls dafür, da die höhere Zahl an freien Parametern eine Überanpassung wahrscheinlicher macht.

Schwieriger zu deuten sind die vielen nicht existenten aber dennoch angezeigten Objekte des *ssd_resnet*. Das enorme Auftreten insbesondere auf linien gemusterten Hintergründen wie dem Holztisch deutet darauf hin, dass es sich um eine Unteranpassung handelt, bei der sich das Netzwerk vor allem auf die Erkennung durchgehender Kanten zwischen zwei Farben versteift hat. Einige der verwendeten Hintergründe weisen anscheinend genug kontrastreiche Linien auf, um das Modell aufgrund der geringen Transferleistung hier unbrauchbar zu machen.

Das Fazit ist daher, dass die Nutzung einer geringeren Auflösung in der Regel von Vorteil ist, da die Modelle einerseits deutlich schneller trainiert werden können, und andererseits zu einer besseren Transferleistung fähig sind - solange keine deutliche Unteranpassung auftritt.

Die Untergrenze der Auflösung wird dabei durch den Menschen bestimmt - solange der Mensch in der Lage ist, das Objekt zu erkennen, sollte auch der Computer in der Lage sein, das Objekt zu erkennen.

6.3 Die Anzahl an verwendeten Bildern variieren

Prinzipiell ist davon auszugehen, dass eine größere Datenbasis auch zu besseren Ergebnissen führt, da mehr Varianz in den Daten dazu beiträgt, die korrekten Merkmale der Objekte zu identifizieren und Merkmale des Hintergrundes auszuschließen.

Selbst wiederholt auftretende, aber prinzipiell unwichtige Parameter, wie der weiße Hintergrund der bisher verwendeten Trainingsdaten oder auch schlicht der Winkel oder die Position der Objekte im Bild werden in größeren Datensätzen einen geringeren Anteil haben, sodass die Modelle diese Dinge auszublenden lernen.

Da den Datenmengen nach oben hin praktisch keine Grenzen gesetzt sind, wird um Zeit zu sparen im folgenden die Datenmenge auf 50 Trainings- und 5 Testbilder reduziert, entsprechend ist von schlechteren Ergebnissen bei der Detektion auszugehen.

Auch dieser Datensatz findet sich wieder im begleitenden github repository. Es werden weiterhin die gleichen Netzwerke mit den gleichen Einstellungen wie bei den hochauflösenden Bildern im vorherigen Abschnitt verwendet, sodass die ersten drei Einträge in [Tabelle 3](#) bzw. in [Tabelle 4](#) als Referenz herangezogen werden können.

Die Tensorboardausgaben dieser Trainings finden sich in Anlage [E.1](#) und werden in der folgenden Tabelle zusammengefasst:

Tabelle 5: TensorBoard-Ausgaben für weniger Trainingsdaten

Netzwerk	Zeit / hh:mm:ss	Kosten
faster_rcnn_resnet101_v1	03:13:19	0.3074
ssd_mobilenet_v2	03:00:45	0.1067
ssd_resnet101_v1	04:57:37	0.0648

Zunächst ist festzustellen, dass sich die Trainingszeiten zwar verringert haben, jedoch mit ca. 5% Zeitersparnis für das *faster_rcnn_resnet*-Netz und ca. 25% für die *ssd*-Netze nicht mit der Verringerung der Datengröße um ca. 63% korreliert.

Dies ist darauf zurückzuführen, dass zwar die zu verarbeitenden *TFRecords* kleiner werden, der wesentliche Treiber für die Trainingszeit neben der Anzahl an zu trainierenden Parametern des Modells jedoch die in den *config*-Dateien festgesetzte Schrittzahl ist.

Überraschend ist auch, dass die Kosten für das *faster_rcnn_resnet*-Modell zwar gestiegen, für die beiden *ssd*-Modelle jedoch gesunken sind. Dies widerspricht zunächst der Erwartung, dass mit weniger Daten zum trainieren ein schlechteres Trainingsergebnis erzielt wird.

Zur besseren Beurteilung werden wieder die tatsächlichen Detektionen betrachtet. Hierfür werden die gleichen 20 Bilder wie in [Abschnitt 6.2](#) verwendet, auf denen 163 Objekte zu detektieren sind. Die Ergebnisse der Detektionen in Anlage E werden in [Tabelle 6](#) zusammengefasst:

Tabelle 6: Detektionsergebnisse für weniger Trainingsdaten

Netzwerk	E1	E2	E3	E4	E5
faster_rcnn_resnet101	95	1	4	65	0
ssd_mobilenet_v2	67	2	1	94	36
ssd_resnet101_v1	79	13	7	72	24

Die Ergebnisse zeigen für das *faster_rcnn_resnet* trotz der deutlichen Kostensteigerung nur eine leichte Verschlechterung. Die Transferleistung ist nach wie vor gering, und es wurden insgesamt fünf Fehler mehr gemacht, als beim Training mit dem vollen Datensatz. Bei insgesamt 95 korrekt erkannten Objekten auf den Testbildern zeigt sich das Modell somit überraschend robust hinsichtlich einer drastischen Reduktion der Trainingsdaten.

Dem gegenüber ist die Verringerung der Kosten für das *ssd_mobilenet* alleine mit den Ergebnissen nicht zu erklären, da diese eine deutliche Verschlechterung zeigen. So hat sich die Anzahl der nicht erkannten Objekte um gut ein Drittel erhöht, und die Anzahl der nicht existenten aber angezeigten Objekte verdoppelt. Eine Überprüfung der Detektionen zeigt jedoch, dass viele dieser Detektionen nicht vorhandener Objekte sehr großflächig sind und

eines oder mehrere Objekte der selben Kategorie einschließen.

Bei der Bewertung der Detektionen wurde diesen Fällen die Kategorie E5 zugewiesen und gleichzeitig das Objekt als nicht erkannt kategorisiert, sodass auch ein Fehler der Kategorie E4 gezählt wird. Wahrscheinlich werden ähnliche Fälle innerhalb der Trainingsdaten auftreten. Die Kostenfunktion erkennt in solchen Situationen höchst wahrscheinlich die Überlagerung der tatsächlich vorhanden Objekte in diesen Detektionen als wenigstens anteilig korrekt, wodurch sich insgesamt die Kosten reduzieren.

Ein anderes Phänomen stellt sich für das *ssd_resnet* ein, da aufgrund der geringeren Trainingsdaten anscheinend eine stärkere Anpassung an die Trainingsdaten stattfindet. Dies führt dazu, dass die vielen angezeigten aber nicht vorhandenen Objekte, welche in [Abschnitt 6.2](#) aufgetreten sind, wegfallen. Gleichzeitig ist das Modell in der Lage, eine etwas bessere Transferleistung zu erbringen und erkennt dadurch etwas mehr Objekte korrekt.

Damit widerspricht das Ergebnis den Erwartungen und das Modell konnte durch die geringere Zahl an Bildern eine deutlich geringere Unteranpassung erzielen, was tatsächlich zu einer Verbesserung geführt hat.

Unabhängig von diesem Phänomen sollte das Fazit jedoch sein, dass mehr Trainingsdaten prinzipiell zu besseren Ergebnissen führen. Das *ssd_resnet*-Modell stellt hier aufgrund der deutlichen Unteranpassung lediglich einen Ausreißer dar, welchem nicht durch eine Reduktion der Trainingsdaten, sondern durch ein verringern der Unteranpassung durch die bisherigen und noch folgenden Überlegungen entgegengewirkt werden sollte.

6.4 Der Einfluss der Schrittzahl beim Training

Wie bereits oben erläutert zeigen die TensorBoard-Ausgaben, ob die Kostenfunktion eines Modells bereits gegen ihren Minimalwert konvergiert ist. Prinzipiell ist durch ein Training über diesen Punkt hinaus keine relevante Verbesserung mehr zu erwarten. In ungünstigen Fällen kann ein zu langes Trainieren sogar zu einer stärkeren Überanpassung an die vorhandenen Trainingsdaten führen und damit die Transferleistung verringern.

Gleichzeitig ist es jedoch nicht einfach zu erkennen, ob die Kosten bereits gegen ihren Grenzwert konvergiert sind, ohne bereits in den Bereich der Überanpassung abzudriften. Obwohl die TensorBoards in Anlage [C.1](#) den Eindruck erwecken, dass die Kostenfunktionen bereits nahe am Minimum angekommen sind, sollen die Modelle *faster_rcnn_resnet* und *ssd_resnet* in den selben Einstellungen noch weiter trainiert werden. Auf ein Training des *mobilenet*-Modells soll aufgrund der langen Trainingszeiten verzichtet werden.

Die Schrittzahl der beiden Modelle wurde auf 100,000 erhöht, alle anderen Einstellungen,

die Trainingsdaten und die Testbilder für die Detektionen sind erneut die selben wie in [Abbildung 6.2](#). Es werden wieder die HD-Aufnahmen verwendet, sodass wieder die Tabellen 3 und 4 als Referenz dienen.

Wie die TensorBoard-Ausgaben in Anlage [F.1](#) zeigen, war nach 25,000 Schritten offensichtlich noch nicht der Grenzwert erreicht. Die Ergebnisse werden in der folgenden Tabelle zusammengefasst:

Tabelle 7: TensorBoard-Ausgaben für lange Laufzeiten

Netzwerk	Zeit / hh:mm:ss	Kosten
faster_rcnn_resnet101_v1	23:13:23	6.06e-3
ssd_resnet101_v1	28:42:24	0.0633

Insbesondere für das *faster_rcnn_resnet*-Modell ist der Anstieg der Trainingszeit nicht proportional zum Ansteig der Schrittzahl; statt einer erwarteten Vervierfachung benötigte das Modell mehr als Sechs mal so lange. Hier liegt das *ssd_resnet*-Modell deutlich näher an den Erwartungen.

Die Ursache hierfür dürfte vor allem in der Auslastung der Hardware liegen. Während des Trainings des *faster_rcnn*-Modells wurde zum Teil aktiv am Computer gearbeitet, was die verfügbaren Ressourcen für das Training reduziert hat. Das *ssd*-Netzwerk konnte die gesamte Kapazität der Hardware ausnutzen. Für beide Fälle gilt jedoch, dass eine zwangsläufige Erwärmung insbesondere der [GPU](#) dafür sorgt, dass die Rechenleistung nach einiger Zeit reduziert wird, um thermische Beschädigungen der Hardware zu vermeiden.

Die Kosten konnten für das *faster_rcnn*-Modell in etwa um den Faktor 14 und selbst für das *ssd*-Modell noch in etwa um den Faktor 2 reduziert werden. Zumindest für das *faster_rcnn*-Modell scheint sich damit die zusätzliche Trainingszeit auf den ersten Blick gelohnt zu haben.

Die Ergebnisse der Detektionen dieser Modelle werden in [Tabelle 8](#) zusammengefasst:

Tabelle 8: Detektionsergebnisse für lange Laufzeiten

Netzwerk	E1	E2	E3	E4	E5
faster_rcnn_resnet101	90	0	3	70	0
ssd_resnet101_v1	82	15	8	70	26

Wie befürchtet haben sich die Detektionen für das *faster_rcnn*-Netzwerk sogar leicht verschlechtert. Die Reduktion der Kostenfunktion scheint gemäß [Abbildung 14](#) vor allem auf die Verbesserung der Positionen der Detektionen zurückzuführen zu sein - da diese jedoch bereits in der kurzen Laufzeit hinreichend genau ermittelt werden konnten hat sich dadurch wie erwartet kein nennenswerter Unterschied für die Detektionen eingestellt.

Die nun größere Anzahl an nicht erkannten Objekten zeigt außerdem, dass die befürchtete stärkere Überanpassung eingetreten ist, die Transferleistung ist gesunken.

Das *ssd_resnet*-Modell konnte jedoch deutlich bessere Ergebnisse erzielen. Insbesondere die Fehler der nicht existenten aber dennoch angezeigten Objekte konnten im Vergleich zu [Tabelle 4](#) nahezu komplett eliminiert werden, außerdem wurden weniger Objekte falsch kategorisiert oder mit der falschen Position erkannt. Dies deutet darauf hin, dass die längere Trainingszeit damit auch geeignet ist, die vorherige Unteranpassung zu kompensieren.

Bestätigt wird die Annahme durch die TensorBoard-Ausgabe gemäß [Abbildung 14](#), bei welcher die Regularisierungskosten in etwa halbiert werden konnten.

Bei bereits guten Netzwerken konnte somit wie erwartet keine Verbesserung erzielt werden, insbesondere die Transferleistung wurde nicht erhöht. Bei Netzwerken mit starker Unteranpassung kann die Erhöhung der Schrittzahl jedoch dazu beitragen, die Ergebnisse zu verbessern.

6.5 Die Nutzung verschiedener Hintergründe

Die bisherigen Maßnahmen haben gezeigt, dass es für die verwendeten Modelle nur schwer möglich ist, eine Transferleistung auf neue Hintergründe zu erbringen. Das Training der gleichen Objekte auf verschiedenen Hintergründen soll es den Modellen erlauben, die diversen Hintergründe als Kriterium auszuschließen. Damit sollte zum einen das Trainingsergebnis allgemein verbessert werden, zum anderen sollte insbesondere die Transferleistung auf neue Hintergründe steigen.

In Anbetracht der erwarteten Verbesserung und nach den Ergebnissen in [Abschnitt 6.3](#) wird das Training auf insgesamt 5 Hintergründen mit jeweils 9 Bildern durchgeführt, sodass 40 Trainings- und 5 Testbilder genutzt werden, wobei darauf geachtet wurde, ein Testbild je Hintergrund zu nutzen. Diese Rohbilder finden sich ebenfalls im github repository.

Zur Nutzung in den Modellen werden diese Bilder wieder mit dem Skript *partition_dataset* auf 1280×960 Pixel skaliert. Anschließend werden die gleichen Modelle mit den gleichen Einstellungen auf den verkleinerten Bildern trainiert, wie auch schon in [Abschnitt 6.3](#) und bei den HD-Aufnahmen in [Abschnitt 6.2](#).

Als Referenz kann in dieser Konstellation jedoch nur noch [Tabelle 3](#) genutzt werden, da die Detektionen nun auf anderen Bildern stattfinden. Die TensorBoard-Ausgaben gemäß Anlage [G.1](#) werden in der folgenden Tabelle zusammengefasst:

Tabelle 9: TensorBoard-Ausgaben für multiple Hintergründe

Netzwerk	Zeit / hh:mm:ss	Kosten
faster_rcnn_resnet101_v1	03:17:34	0.0136
ssd_mobilenet_v2	03:00:55	0.1013
ssd_resnet101_v1	04:44:12	0.0594

Die Trainingszeiten liegen nach den Erfahrungen aus [Abschnitt 6.3](#) im erwarteten Bereich. Außerdem sind die Kosten trotz der geringen Anzahl an Bildern für die *ssd*-Modelle bisher am geringsten ausgefallen, lediglich für das *faster_rcnn_resnet* konnten durch die Erhöhung der Schrittzahl die Kosten noch weiter gesenkt werden.

Dies scheint somit die Annahme zu bestätigen, dass eine höhere Abwechslung nicht-relevanter Inhalte auf den Bildern den Lerneffekt und insbesondere die Transferleistung erhöht.

Die Betrachtung der Detektionsergebnisse in Anlage G bestätigt diese Annahmen zumindest für das *faster_rcnn*-Modell, jedoch nicht für die beiden *ssd*-Modelle, wie [Tabelle 10](#) zeigt:

Tabelle 10: Detektionsergebnisse für multiple Hintergründe

Netzwerk	E1	E2	E3	E4	E5
faster_rcnn_resnet101	172	2	0	15	0
ssd_mobilenet_v2	121	8	15	46	36
ssd_resnet101_v1	101	3	16	70	24

Für dieses Modell wurden dabei insgesamt 23 Bilder zur Detektion verwendet:

- fünf hochauflösende Bilder, eines je trainiertem Hintergrund, auf denen das Modell auch trainiert wurde
- fünf hochauflösende Bilder, eines je trainiertem Hintergrund, welche aus den Trainingsdaten entfernt wurden
- zwei Bilder aus dem Demo-Datensatz mit niedriger Auflösung
- drei hochauflösende Bilder aus dem Datensatz in [Abschnitt 6.2](#), welche nicht trainiert wurden
- acht hochauflösenden Bilder mit jeweils neuen, verschiedenen Hintergründen

Insgesamt konnten auf diesen Bildern 187 Objekte erkannt werden.

Dabei war das *faster_rcnn*-Modell nun in der Lage, auf fast allen Hintergründen auch auf

neuen Bildern die gesuchten Objekte zu erkennen. Interessant ist dabei jedoch einerseits, dass es bei der Erkennung von Schokoriegeln auf Bildern aus dem Demo-Datensatz zu Problemen kam. Andererseits überrascht die Tatsache, dass auf dem eigentlich "einfachem", schlicht pinken Hintergrund kein einziges Objekt detektiert werden konnte.

Das *ssd_mobilenet*-Modell konnte sich ebenfalls auf den bisher schwierigen Hintergründen wie Fußmatte und Granit verbessern, erzielt jedoch nach wie vor keine zufriedenstellenden Ergebnisse.

Wie in den bisherigen Versuchen erzielt auch hier das *ssd_resnet* wieder die schlechtesten Ergebnisse. So ist sowohl die Erkennung auf den schwierigen Hintergründen wie Fußmatte und Granit nach wie vor nicht erfolgreich, auch die Erkennung von Objekten auf Bildern aus dem Demo-Datensatz schlägt nun fehl und ist noch schlechter, als die Erkennung nach dem Training mit 50 weißen Bildern.

Somit bleibt die Feststellung, dass die Nutzung verschiedener Hintergründe definitiv zu einer Verbesserung beiträgt, wenn auch wieder das *ssd_resnet* als am schwierigsten zu trainierendes Netzwerk auffällt.

6.6 Die Ergebnisse eines optimierten Modells

Die bisherigen Erkenntnisse sollen nun genutzt werden, um ein optimiertes Modell zu trainieren. Um nicht erneut alle Bilder auf den neuen Hintergründen auf 640×480 Pixel verkleinern und neu labeln zu müssen, wurde auf diesen Punkt aus Zeitgründen verzichtet.

Davon ab wurde das *faster_rcnn_resnet101* ausgewählt, da es im Modellvergleich in [Abschnitt 6.1](#) am meisten Objekte korrekt erkannt hat und auch bei den vier Fehlerkategorien sehr gut und bei Zweien sogar am besten abgeschnitten hat.

Ergänzend wird das *faster_rcnn_inception*-Netzwerk trainiert. Dieses lieferte im Modellvergleich in [Abschnitt 6.1](#) ebenfalls sehr gute Ergebnisse, fiel jedoch dadurch auf, dass es keinerlei Objekte falsch kategorisiert hatte. Diese Eigenschaft kann ausschlaggebend für die Auswahl eines Modells sein, weshalb es hier erneut getestet wird.

Da die Trainingszeit des *faster_rcnn_inception*-Netzwerk im Modellvergleich jedoch mehr als zweieinhalf mal so hoch lag wie beim *faster_rcnn_resnet101* und die Ergebnisse sehr ähnlich waren wurde während der vorherigen Anpassungen auf die Investition der zusätzlichen Trainingszeit verzichtet.

Zum Training wurden die weniger komplexen Modelle in der 640×640 -Pixel Variante genutzt, um der Überanpassung aus [Abschnitt 6.2](#) vorzubeugen. Die Schrittzahl wurde nach

den Erfahrungen in [Abschnitt 6.4](#) bei 25,000 belassen.

Da sich in [Abschnitt 6.3](#) und [Abschnitt 6.5](#) insbesondere die Kombination vieler verschiedener Bilder mit verschiedenen Hintergründen als effektiv erwiesen hat, werden die Modelle sowohl mit dem gesamten Trainingssatz aus [Abschnitt 6.2](#) für hochauflösende Bilder mit weißen Hintergründen als auch mit den Bildern der vier später hinzugefügten Hintergründen aus [Abschnitt 6.5](#) trainiert.

Insgesamt ergeben sich somit 20 Test- und 171 Trainingsbilder.

Die Ergebnisse dieser Trainings werden in Anlage H dokumentiert. Für die TensorBoard-Ausgaben der Trainings ergeben sich die folgenden Werte:

Tabelle 11: TensorBoard-Ausgabe für das optimierte Modell

Netzwerk	Zeit / hh:mm:ss	Kosten
faster_rcnn_resnet101	05:23:07	6.41e-3
faster_rcnn_inception	05:28:03	0.071

Die Trainingszeit des *faster_rcnn_resnet101* ist demnach angestiegen und entspricht nun der des *faster_rcnn_inception*-Modells. Dies ist unerwartet, da nach dem Modellvergleich mit einer deutlich längeren Trainingsdauer des *faster_rcnn_inception*-Modells zu rechnen war.

Gleichzeitig sind die Kosten des *faster_rcnn_resnet101* sehr niedrig und entsprechen in etwa denen nach der langen Laufzeit von 100,000 Schritten. Dies entspricht somit den erwarteten Ergebnissen. Hier liegt der Unterschied zwischen den beiden Modellen jedoch im etwa gleichen Verhältnis wie in [Tabelle 1](#), sodass die höheren Kosten des *faster_rcnn_inception*-Netzwerks zu erwarten waren und erstmal kein Indiz für ein schlechteres Ergebnis sind.

Für die Detektionen ergeben sich die folgenden Ergebnisse:

Tabelle 12: Detektionsergebnis für das optimierte Modell

Netzwerk	E1	E2	E3	E4	E5
faster_rcnn_resnet101	168	3	7	4	0
faster_rcnn_inception	181	0	5	1	0

Das *faster_rcnn_resnet101* war damit also in der Lage, die Anzahl der erkannten Objekte zu steigern, es wurden lediglich vier Objekte nicht erkannt - im Vergleich zu [Tabelle 10](#) ist das eine Reduzierung um rund 73%.

Insbesondere auf den in [Abschnitt 6.5](#) schwierigen Hintergründen der pinken Decke und der Snoopy-Decke, also bei der Transferleistung, konnte sich das Modell verbessern. Hier liegen

jedoch nach wie vor alle vier nicht erkannten, sowie sechs der sieben erkannten, aber falsch positionierten Objekte.

Ein wenig enttäuschend ist lediglich die Anzahl von drei falsch kategorisierten Objekten - hier wäre je nach Aufgabenstellung noch Verbesserungsbedarf nötig. Es sei aber darauf hingewiesen, dass es sich hierbei in allen drei Fällen um doppelte Detektionen handelt: Das Modell konnte das korrekte Objekt erkennen, hat aber zusätzlich ein falsches Objekt angegeben.

Dem Gegenüber steht das *faster_rcnn_inception*-Netzwerk, welches mit den Optimierungen nun sogar in allen Kategorien bessere Ergebnisse erzielt. Insbesondere wurden keine Objekte falsch kategorisiert, der am stärksten gewichtete Fehler tritt also nicht auf.

Außerdem ist das Modell mit lediglich einem einzigen nicht erkannten Objekt nochmal deutlich effizienter, als das *faster_rcnn_resnet101*.

Zuletzt sei auch nochmals auf die bisher nicht erwähnte Sicherheit der Detektionen eingegangen. So wird zu jeder Detektion mitgeteilt, wie sicher sich das Modell bei der angegebenen Detektion ist. Diese liegt nach den Verbesserungen bei quasi allen Detektionen bei 100%, konnte also im Vergleich zu den Ergebnissen in [Abschnitt 6.5](#) ebenfalls leicht und im Vergleich zu allen anderen Versuchen in [Abschnitt 6](#) sogar zum Teil sehr deutlich gesteigert werden.

Insgesamt zeigen die vorgenommenen Anpassungen also, dass die Ergebnisse insbesondere hinsichtlich der Transferleistung verbessert werden konnten. Dies bestätigt die bisherigen Schlussfolgerungen, auch wenn nach wie vor kein absolut perfektes Ergebnis erzielt werden konnte.

6.7 Weitere denkbare Einflüsse ohne Überprüfung

Neben den oben getesteten und erläuterten Einflussmöglichkeiten existieren noch zahlreiche weitere Parameter, welche die Ergebnisse der Detektionen beeinflussen können. Auf Seiten der Datenbasis ist davon auszugehen, dass beispielsweise die Beleuchtung und damit einhergehende Faktoren wie Helligkeit, Schattenwurf und Kontraste, aber eventuell auch Reflexionen auf glänzenden Oberflächen die Ergebnisse verändern und das Training bzw. die Detektion im Regelfall erschweren.

Gleicher gilt für die Variablen wie Abstand oder Winkel der betrachteten Objekte. sofern diese in der Anwendung variieren können.

Auch bei den Objekten selbst kann eine gewisse Variation möglich sein. So sollte das Trai-

ning bei unsymmetrischen Objekten Bilder von allen Seiten eines Objekts beinhalten, um beispielsweise ein auf dem Kopf liegendes Objekt genauso zu erkennen wie ein richtig herum liegendes Objekt.

Eine andere Möglichkeit besteht in der Variation gleichartiger Objekte untereinander. So kann es beispielsweise vorkommen, dass die hier verwendeten Süßigkeiten deformiert werden, oder schlichtweg in der Verpackung verrutschen. Im Beispiel in [Abschnitt 7](#) wird der Fall aufgezeigt, dass es auch Variationen des selben Gegenstandes geben kann - so ist beispielsweise der Seitenschneider einer Marke blau und etwas länglicher, der einer anderen Marke eher breiter und rot.

Eine weitere Steigerung ist es, zum Teil verdeckte Objekte zu erkennen. In den hier vorgestellten Trainingsdaten wird der sehr anschauliche Fall von einigen auf einer flachen Oberfläche verteilten Süßigkeiten behandelt. In der Praxis ist es aber sehr wahrscheinlich, dass die Süßigkeiten auf einem Haufen oder in einer Schüssel liegen, sich zum Teil verdecken und überlappen.

All diese und weitere denkbare Abweichungen stellen potentielle Transfераufgaben dar, auf welche die gewünschten Modelle am effektivsten durch viele verschiedene Trainingsdaten vorbereitet werden. Zusätzlich kann es hilfreich sein, ähnliche Objekte zur Abgrenzung mitzutrainieren. Bei den verwendeten Süßigkeiten könnten dies beispielsweise Konkurrenzprodukte mit ähnlicher Optik sein, welche entweder als solche oder gar nicht erkannt werden sollen.

Seitens der Trainingseinstellungen gibt es ebenfalls noch weitere Parameter, die eingestellt werden können. So gibt es beispielsweise in den Modellen die Option einer Regularisierung, welche je nach Gewichtung eine Über- oder Unteranpassung verringern oder verstärken kann.

Auch andere Kostenfaktoren, wie beispielsweise die Lokalisierung oder die Kategorisierung, können in der *pipeline.config*-Datei unterschiedlich gewichtet werden. So könnte beispielsweise das Training derart angepasst werden, dass eher keine Detektionen angegeben werden als solche mit schlechter Kategorisierung.

Weitere Einflussgrößen finden sich rund um die Lernrate und die Aufteilung auf die Schrittzahl. So können Anfangswert und Maximalwert der Lernrate angepasst werden, sowie die Schrittzahl, in welcher dieses Maximum erreicht werden soll. Dies kann dem Modell helfen, schneller und erfolgreicher zu konvergieren, bei einer ungünstigen Wahl aber den Prozess deutlich verlangsamen oder sogar dafür sorgen, dass die Kostenfunktion divergiert.

In die gleiche Richtung zielt auch die Gewichtung des *momentum_optimizer* beim Training ab. Ein höherer Wert kann es dem Modell erlauben, Ausreißer in der Kostenfunktion

zu ignorieren und die Variablen in der selben Weise weiter anzupassen. Im Umkehrschluss besteht die Gefahr, dass eine Anpassung der Modellparameter aufgrund zu hoher Trägheit beim Lernen nicht in der nötigen Größe vorgenommen wird, mit der Folge eines schlechten Lernergebnis bis hin zur Divergenz der Kostenfunktion.

Wie die *pipeline.config*-Dateien der Modelle zeigen, existieren noch zahlreiche weitere Einstellungen, welche jedoch deutlich mehr Hintergrundwissen zur Funktion von neuronalen Netzwerken benötigen. Doch selbst mit vollem Verständnis für die zugrundeliegenden Theorien und Funktionen besteht immer die Gefahr, dass das Modell nach dem Training nicht nutzbar ist.

Aus diesem Grund wird, wie auch in dieser Arbeit, häufig auf das sogenannte Transferlearning zurückgegriffen: Aus den Erfahrungen vieler Anwender wurde der *TensorFlow Model Zoo* mit Einstellungen ausgeliefert, welche ein breites Spektrum an Anwendungsfällen zufriedenstellend verarbeiten können.

Zuletzt sei noch eine Möglichkeit zur Verbesserung der Detektionen nach dem Trainieren genannt. So greift das Skript *trained_object_detection* auf die Ausgabe der Modelle über die Sicherheit der Detektion zu und gibt nur solche Ausgaben weiter, die über 50% liegen.

Durch eine Erhöhung dieses *threshold* genannten Grenzwerts (in Zeile 148 des Skripts) könnten beispielsweise viele angezeigte aber nicht existente Objekte aussortiert werden.

7 Transfer auf eine alternative Problemstellung

Um zu zeigen, dass die verwendeten Modelle und Überlegungen auch auf andere Probleme übertragen werden können, soll eine weitere mögliche Problemstellung erarbeitet werden.

Neben der Unterstützung sehbehinderter Menschen im Alltag ist die Unterstürzung des Menschen während der Arbeit durch den Computer ein stetig wachsender Wirtschafts- und Wissenschaftszweig. So könnte beispielsweise eine alternative Aufgabe darin bestehen, in einer Werkstatt aus herumliegenden Werkzeugen das benötigte zu erkennen und anzugeben.

Zu diesem Zweck wurden auf den fünf bereits früher genutzten Hintergründen jeweils 14 Bilder aufgenommen, von denen je eines zur Detektion aus dem Lernprozess aussortiert wurde. Die verbleibenden 13 Bilder pro Hintergrund wurden nach den Erkenntnissen in [Abschnitt 6](#) auf das Format 640×480 Pixel herunter skaliert. Außerdem wurde aus jedem Datensatz pro Hintergrund ein Bild in den Testdatensatz verschoben.

Trainiert wurden die Bilder anschließend mit dem *faster_rcnn_resnet101* und dem *faster_rcnn_inception*-Netzwerk, jeweils in der Variante für 640×640 Pixel. Bis auf die Änderung in der Auflösung und der Komplexität der Modelle entspricht das Training in den Einstellungen dem in [Abschnitt 6.5](#).

Die sich ergebenden Datensätze und Einstellungen finden sich ebenfalls im github repository.

Für die Detektion wurden 18 Bilder verwendet:

- fünf bereits trainierte Bilder (eines pro Hintergrund)
- die fünf vorher aussortierten Bilder (eines pro Hintergrund)
- jeweils ein Bild auf den acht weiteren, dem Modell bis dahin komplett unbekannten Hintergründen

Die Bilder enthalten insgesamt 162 Objekte, die erkannt werden sollen.

Die TensorBoard-Ausgaben der Trainings in Anlage [I.1](#) werden in [Tabelle 13](#) zusammengefasst:

Tabelle 13: TensorBoard-Ausgabe für das Training mit Werkzeugen

Netzwerk	Zeit / hh:mm:ss	Kosten
faster_rcnn_resnet101	01:22:06	4.12e-3
faster_rcnn_inception	03:24:47	0.030

Die geringeren Trainingszeiten waren zu erwarten und ergeben sich durch die gewählte gerin gere Auflösung der Modelle im Vergleich zu [Abschnitt 6.6](#). Die Differenz in der Trainingszeit zwischen den beiden Modellen entspricht außerdem wieder dem Verhältnis, welches bereits in [Tabelle 1](#) festgestellt wurde.

Die Kosten des *faster_rcnn_resnet101* bewegen sich in der selben Größenordnung wie in [Abschnitt 6.5](#) und liegen somit im zu erwartenden Bereich. Außerdem ist der Unterschied zwischen beiden Modellen wieder in der selben Größe wie in [Tabelle 1](#), sodass davon ausgegangen werden kann, dass beide Modelle wieder ähnlich gute Ergebnisse erzielen.

Für die Detektionen ergeben sich die folgenden Ergebnisse:

Tabelle 14: Detektionsergebnis für das Training mit Werkzeugen

Netzwerk	E1	E2	E3	E4	E5
faster_rcnn_resnet101	147	2	8	7	1
faster_rcnn_inception	145	1	7	10	1

Die Ergebnisse sind wieder sehr gut, die Probleme treten ebenfalls vor allem auf dem pinken Hintergrund und der Snoopy-Decke auf. Dies entspricht ebenfalls den Erwartungen und bestätigt die Ergebnisse aus [Abschnitt 6](#).

Ähnlich wie in [Abschnitt 6.6](#) ist außerdem das *faster_rcnn_inception*-Netzwerk ein wenig besser darin, Objekte korrekt zu kategorisieren, auch wenn es diesmal ein Objekt falsch kategorisiert hat. Überraschenderweise ist jedoch gerade die Anzahl der gar nicht erkannten Objekte beim *faster_rcnn_inception*-Modell deutlich größer als beim *faster_rcnn_resnet101*. Dies zeigt, dass die korrekte Wahl des vortrainierten Modells auch von der jeweiligen Problemstellung abhängt und nicht immer abschließend im Vorhinein bestimmt werden kann.

Die weiteren Verbesserungsmöglichkeiten entsprechen somit denen, die bereits in [Abschnitt 6](#) besprochen wurden und liegen in erster Linie in der Verwendung weiterer Trainingsbilder mit mehr Variationen.

8 Mögliche Probleme

Dieses Kapitel soll einige wahrscheinliche Probleme, welche nicht direkt bei der Installation auftreten, erläutern und Lösungsmöglichkeiten aufzeigen. Sollte ein hier nicht genanntes Problem auftreten empfiehlt sich insbesondere das Forum stackoverflow²⁰, in welchem sowohl Fragen rund um Programmierung, Betriebssysteme als auch maschinelles Lernen und TensorFlow gestellt werden können. Häufig existiert hier sogar bereits eine Antwort auf die eigene Problemstellung.

8.1 *Loss pendelt sich nicht ein, Möglichkeit 1*

Idealerweise verlaufen die Kosten abfallend und flachen zum Ende hin zunehmend ab, nähern sich also einem unteren Grenzwert an. Sollte dies im TensorBoard nicht der Fall sein und die Kosten zwischen niedrigen und hohen Werten springen kann dies unterschiedliche Ursachen haben.

Ein möglicher Fehler liegt darin, dass die Annotationen nicht zu den Bildern passen. Die erste Ursache dafür könnte sein, dass beim Erstellen der *TFRecords* ein Fehler aufgetreten ist, beispielsweise ein falsch geschriebener Dateipfad.

Alternativ kann eine Diskrepanz zwischen der Bildgröße beim Labeln und der Bildgröße, welche dem Modell zum trainieren übergeben wird, dazu führen, dass die Label nicht mehr zu den Objekten passen. Dies kann passieren, wenn im gewählten Modell die Option *image_resizer* auf eine andere Größe gestellt ist, als die Bilder in *labelImg* aufgewiesen haben.

Je nach Situation bedeutet dies, dass entweder die Bilder skaliert und neu gelabelt werden müssen, oder dass ein anderes Modell bzw. eine andere Einstellung für die Option *image_resizer* gewählt werden muss.

8.2 *Loss pendelt sich nicht ein, Möglichkeit 2*

Eine weitere Ursache für ein ausbleibendes Lernen des Modell kann in der gewählten Lernrate liegen.

In diesem Fall sollte in der *pipeline.config*-Datei der Wert **learning_rate** verringert werden. Dieser wird unter Umständen in zwei Werte unterteilt, einen niedrigeren Startwert und einen angestrebten Basis- oder Maximalwert, verknüpft über eine Schrittzahl, über welche sich die Lernrate vom Start- an den Endwert annähert.

Häufig hilft hier, den Startwert zu verringern, eventuell auch in Kombination mit einer höhe-

²⁰<https://stackoverflow.com/questions>

ren Zahl an Schritten zum Erreichen des Endwerts, um die Lernkurve bewusst abzuflachen. Abbildung 8 zeigt die Einstellungen an einem Beispiel:

```
optimizer {
  momentum_optimizer {
    learning_rate {
      cosine_decay_learning_rate {
        learning_rate_base: 0.07999999821186066
        total_steps: 30000
        warmup_learning_rate: 0.001000000474974513
        warmup_steps: 2500
      }
    }
  }
}
```

Abbildung 8: Einstellungen der Lernrate für das *efficientdet_d1*-Modell

Eventuell führt ein starkes Pendeln auch zum *nan*-Fehler, wie im nächsten Abschnitt beschrieben:

8.3 *Loss* zeigt *nan* statt einem Wert

Der Wert **nan** steht für *not a number*. Im Normalfall bedeutet dies, dass ein Wert gegen unendlich (oder zumindest außerhalb des erlaubten Wertebereichs) strebt. In diesem Fall hilft ein Blick in das TensorBoard, welches anzeigt, welcher Wert hierfür verantwortlich ist:

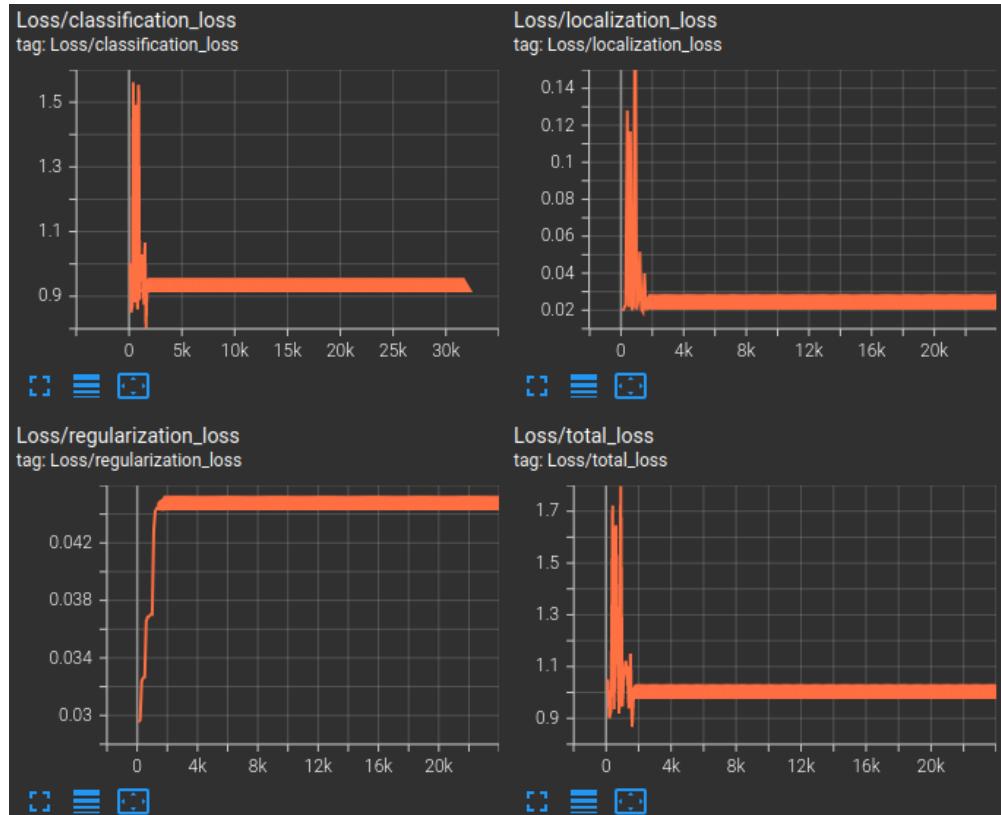


Abbildung 9: TensorBoard Ausgabe für *Loss/regularization_loss nan*

Wie Abbildung 9 zeigt, steigen die Regularisierungskosten an, während die anderen Kosten fallend sind. Dies lässt vermuten, dass die Gewichtung der Regularisierung zu gering ist.

Um dies auszugleichen kann in der *pipeline.config*-Datei der Wert **regularizer → weight** erhöht werden, wie [Abbildung 10](#) zeigt:

```
box_predictor {  
    weight_shared_convolutional_box_predictor {  
        conv_hyperparams {  
            regularizer {  
                l2_regularizer {  
                    weight: 3.999998989515007e-04  
                }  
            }  
        }  
    }  
}
```

Abbildung 10: Anpassung der Regularisierung in *pipeline.config*

Dieser Wert findet sich eventuell mehrfach in der Konfiguration!

8.4 Die Kostenfunktion endet mit einem Ausreißer nach oben

Aufgrund der zugrundeliegenden Mathematik kann es passieren, dass das TensorBoard für den Verlauf der Kostenfunktion aufzeigt, dass die letzten Trainingsschritte zu einer deutlichen Steigerung der Kosten geführt haben. Dies bedeutet im Normalfall eine deutliche Verschlechterung des Netzwerks.

Das Netzwerk ist jedoch auch in der Lage, aus diesem Fehler zu lernen. Dazu kann in der *.config*-Datei die Anzahl der Trainingsschritte um wenige Hundert erhöht werden. Anschließend wird der Trainingsbefehl für das Netzwerk unverändert erneut im Terminal ausgeführt. Das Training des Netzwerks wird dann mit den bestehenden Lernerfolgen fortgeführt, es entsteht also im Regelfall nur ein minimaler zeitlicher Mehraufwand, um das Problem zu beheben.

8.5 Das Training oder der Export werden wegen einer inkompatiblen *Tensor shape* abgebrochen

Dies kann der Fall sein, wenn in der *pipeline.config*-Datei nach Beginn des Trainings Werte wie die Bildgröße oder die Anzahl der Klassen verändert wurde.

Alternativ kann es beim genutzten Modell zu einer Inkompatibilität zwischen der eingestellten Bildgröße und der Modellarchitektur kommen, sodass die Bilder nicht verarbeitet werden können. Unter Umständen kann es hierbei helfen, das Bild künstlich auf ein Quadrat aufzufüllen, indem die Option *image_resizer* von *fixed_shape_resizer* auf *keep_aspect_ratio_resizer* geändert und ein sogenanntes *padding* ergänzt wird:

```
image_resizer {  
    keep_aspect_ratio_resizer {  
        min_dimension: 480  
        max_dimension: 640  
        pad_to_max_dimension: true  
    }  
}
```

Anschließend kann es erforderlich sein, den bisherigen Trainingsfortschritt zu löschen und das Training neu zu starten.

8.6 Ein Terminalbefehl wird nicht korrekt ausgeführt

Dies könnte einerseits an einer fehlenden Installation liegen. Hier sollte überprüft werden, ob die virtuelle Umgebung aktiviert ist, siehe auch [Abbildung 1](#).

Ist dies der Fall liegt der Fehler höchst wahrscheinlich an einem falschen Pfad. Dies bedeutet, dass entweder der aktuell ausgewählte Pfad im Terminal falsch ist, also beispielsweise versucht wird, ein Skript aus einem *workspace* heraus auszuführen. Oder es bedeutet, dass schlichtweg ein Tippfehler im Pfad vorliegt.

In der Praxis erweisen sich dabei das Kopieren von Pfaden (Tastenkombination **Strg + L**) und Dateinamen (Datei auswählen und **F2** drücken) aus der grafischen Oberfläche heraus als praktisches Mittel, um solche Fehler zu vermeiden. Außerdem kann das Terminal in vielen Fällen über die Tabulator-Taste mit einer automatischen Vervollständigung glänzen.

Ebenfalls denkbar sind vergessene Dateiendungen, beispielsweise das **.py** bei einem Skript.

Eine etwas schwieriger zu erkennende Fehlerquelle bei Pfaden können Sonder- oder Leerzeichen sein, da diese im Terminal in der Regel als Befehlszeichen verarbeitet werden. Unter Umständen kann dies mit einem sogenannten Escape-Symbol ausgeglichen werden, sicherer ist es jedoch, Sonder- und Leerzeichen auszulassen oder durch Unter- oder Bindestriche zu ersetzen.

Zuletzt besteht auch schlicht die Möglichkeit, dass ein Pfad noch gar nicht existiert. Dies könnte durch das Kopieren eines Befehls und Anpassen eines übergeordneten Ordners geschehen, wenn tiefere Ordnerebenen noch nicht angelegt wurden.

9 Schlussfolgerung und Ausblick

Wie diese Arbeit gezeigt hat, ist die Einstiegshürde in die Nutzung neuronaler Netzwerke sehr gering. Die nötigen Installationen können mit der Anleitung zur [Vorbereitung der Softwareumgebung](#) in wenigen Stunden von der Installation des Betriebssystems bis hin zur fertigen Umgebung vorgenommen werden.

Nach erfolgreicher Installation der benötigten Software können innerhalb der selben virtuellen Umgebung verschiedene Aufgabenstellungen nachverfolgt werden. Die hier vorgestellten Methoden, Skripte und Verbesserungsmöglichkeiten sind nicht an eine spezielle Problemstellung gebunden.

Zur Ausführung der nötigen Schritte sind keine Programmierkenntnisse notwendig, sodass auch "Laien" in der Lage sind, schnell erste Erfahrungen mit maschinellem Lernen zu machen. Eine erste Erfahrung im Umgang mit dem Terminal mag zwar die Eingewöhnungszeit verringern, ist aber ebenfalls nicht zwingend.

Der wesentliche Arbeitsaufwand für den Anwender besteht damit in der Erstellung und Aufarbeitung der Trainingsdaten, sprich dem Aufnehmen und Labeln der Fotos, eventuell mit einer vorherigen Skalierung auf ein kleineres Bildformat.

Diese Arbeiten sind monoton und können für viele Daten schnell mehrere Stunden bis Tage dauern. Da die Modelle mit Bildern, welche untereinander stark variieren, deutlich effektiver trainiert werden können, sollte hier statt auf Quantität der Fotos primär auf die Qualität hinsichtlich der Variation geachtet werden. So können in kürzerer Zeit bessere Ergebnisse erzielt werden.

Neben dieser Faustformel für die Datenbasis existieren noch zahlreiche weitere Parameter, mit welchen vor allem eine Feinjustage der Modelle während des Trainings möglich ist. Da auch das Training eines einzelnen Modells, gerade mit steigender Komplexität des Netzwerks, schnell mehrere Stunden beansprucht, ist dies jedoch eine zeitintensive Arbeit.

Somit muss hier für die Praxis abgewogen werden, ob mehrere Tage oder Wochen in die Feinabstimmung investiert werden können, oder ob die verbleibenden Fehler akzeptiert oder in nachgeschalteten Prozessen abgefangen werden können.

Abschließend lässt sich feststellen, dass die *faster_rcnn*-Modelle die besten Ergebnisse liefern, dennoch lohnt sich auch hier der Test mehrerer Varianten aus dieser Modellfamilie. Außerdem zeigten einige Annahmen zur Verbesserung der Ergebnisse, wie eine höhere Schrittzahl des Trainings oder die Verwendung von komplexeren Modellen, dass diese nicht zwingend zu einer Verbesserung der Ergebnisse führten, sondern je nach Ausgangslage sogar zu einer

Verschlechterung.

Damit bleibt dem Nutzer zwar die Aufgabe erhalten, verschiedene Modelle zu testen, er ist jedoch durch die Ergebnisse dieser Arbeit in der Lage, anhand der ersten Ausgaben der Modelle die jeweils am besten geeigneten Maßnahmen zu treffen.

Aufbauend auf der Motivation dieser Arbeit wäre der nächste Schritt, das trainierte Modell nicht nur zur bloßen Detektion aufzurufen, sondern das geladene Modell in eine größere Softwareanwendung einzubetten. In dieser Umgebung würde das Modell ebenfalls geladen, um dann über eine Schnittstelle ein Bild zur Detektion zu übergeben.

Die Anwendung müsste die Kategorie und die Koordinaten der erfolgten Detektionen, welche derzeit schlicht in das Terminal gedruckt werden, sinnvoll speichern oder an die aufrufende Stelle zurückgeben. Auf dieser Grundlage wäre es anschließend möglich, gegebenenfalls abhängig von der Objektgeometrie, die Koordinaten eines zentralen Punktes und den Winkel des Objekts zu bestimmen.

Diese derart bestimmte Lage und Ausrichtung könnten zuletzt in ein vorher definiertes Koordinatensystem transformiert werden, in welchem sich ein Greifroboter orientieren kann. Dieser Roboter müsste letztlich von der Anwendung mit den transformierten Daten in einer für ihn lesbaren Art angesprochen und so gesteuert werden, dass das nun bekannte Objekt auch tatsächlich gegriffen werden kann.

Literaturverzeichnis

Alkhooli, Rami (Jan. 2020): *TensorFlow 2 Object Detection API tutorial*. Version commit 93cedcc9ff61217f1ab0394ccddce6fca874f544. URL: <https://github.com/HubertWelp/SweetPicker3/tree/master/Trainingsdaten-bunt>. (aus dem THGA-Kurs Softwaretechnik von Prof. Dr. Hubert Welp, aufgerufen: 17.08.2021).

Preston-Werner, Tom (o.D.): *Semantic Versioning*. URL: <https://semver.org/lang/de/>. (aufgerufen: 18.08.2021).

Rojas, Raul (1996): *Neural Networks - A Systematic Introduction*. Berlin: Springer-Verlag.

Vladimirov, Lyudmil (Juni 2021a): *TensorFlow 2 Object Detection API tutorial*. Version commit 6770aff. URL: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/index.html>. (aufgerufen: 17.08.2021).

Vladimirov, Lyudmil (Juni 2021b): *TensorFlow 2 Object Detection API tutorial*. Version commit 6770aff. URL: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html#verify-your-installation>. (aufgerufen: 17.08.2021).

Vladimirov, Lyudmil (Juni 2021c): *TensorFlow 2 Object Detection API tutorial*. Version commit 6770aff. URL: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#partition-the-dataset>. (aufgerufen: 17.08.2021).

Vladimirov, Lyudmil (Juni 2021d): *TensorFlow 2 Object Detection API tutorial*. Version commit 6770aff. URL: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#convert-xml-to-record>. (aufgerufen: 17.08.2021).

Anlagen

In diesem Dokument werden nur wenige ausgewählte Bilder im Anhang geführt, alle weiteren Dateien finden sich neben github auf der beigefügten DVD und werden hier über den Ordner auf dieser DVD referenziert. Dabei erfolgt die Referenz nach dem Windows-Explorer-Format in der Form DVD : \Ordner\Unterordner....

A Aufnahmen für die Detektion auf dem Modell unbekannten Bildern

Die Vorlagen finden sich unter DVD : \misc.

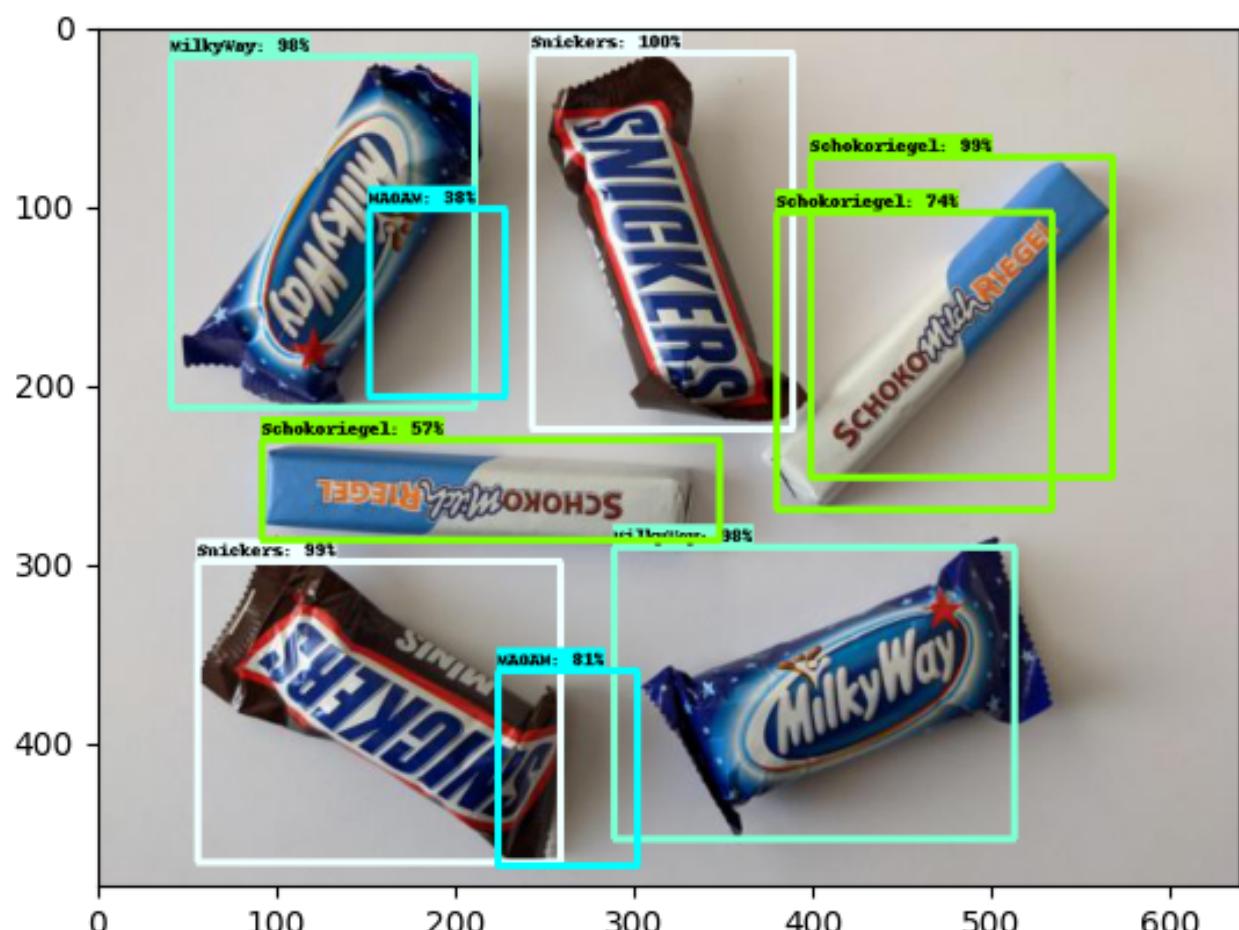


Abbildung 11: Training-Demo: Bild nach Verkleinerung und Detektion auf neutralem Hintergrund (Papier)

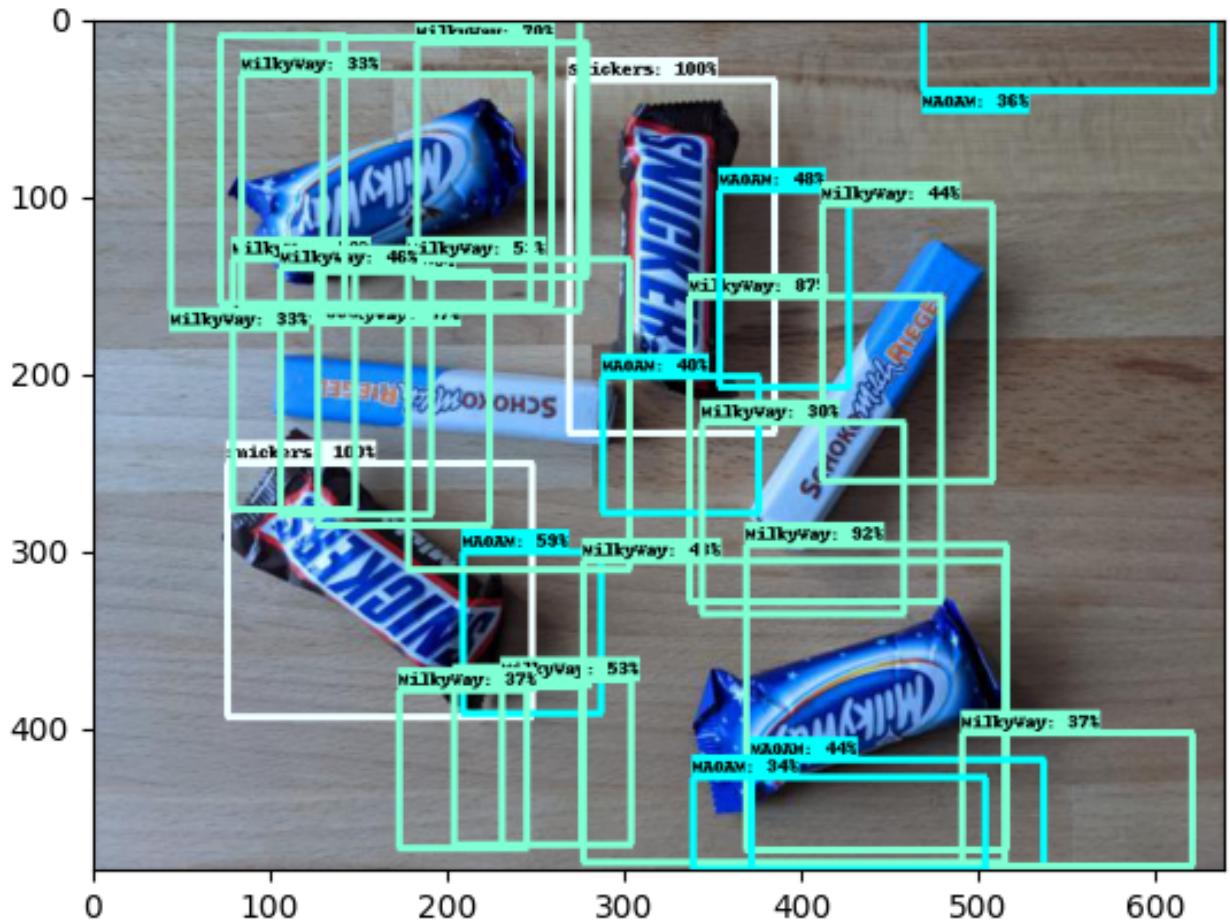


Abbildung 12: Training-Demo: Bild nach Verkleinerung und Detektion auf Holztisch

B Objekterkennung auf den Testbildern zum Vergleich der verschiedenen Modelle

B.1 TensorBoard-Ausgaben für den Vergleich verschiedener Modelle

DVD : \tensorboards\model_comparison

B.2 Die Vorlagenbilder ohne Detektionen

DVD : \workspaces\model_comparison\images\images-to-detect

B.3 Die Detektionen

DVD : \workspaces\model_comparison\images\detections\

C Objekterkennung auf Testbildern für hochauflösende Süßigkeiten auf weißem Hintergrund

C.1 TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten auf weißem Hintergrund

DVD : \tensorboards\new_sweets_HD

C.2 Die Vorlagenbilder ohne Detektionen

DVD : \workspaces\new_sweets_HD\images\images-to-detect

Diese Vorlagen werden für die Abschnitte 6.2, 6.3 und 6.4 genutzt und daher in den Anlagen D, E und F nicht nochmals aufgeführt

C.3 Die Detektionen

DVD : \workspaces\new_sweets_HD\images\detections\

D Objekterkennung auf Testbildern für niedrigauflösender Süßigkeiten auf weißem Hintergrund

D.1 TensorBoard-Ausgaben für das Training niedrigauflösender Süßigkeiten auf weißem Hintergrund

DVD : \tensorboards\new_sweets_ld

D.2 Die Detektionen

DVD : \workspaces\new_sweets_ld\images\detections\

E Objekterkennung auf Testbildern für hochauflösende Süßigkeiten auf weniger Bildern

E.1 TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten mit weniger Bildern

DVD : \tensorboards\new_sweets_fewer_pics

E.2 Die Detektionen

DVD : \workspaces\new_sweets_fewer_pics\images\detections\

F Objekterkennung auf Testbildern für hochauflösende Süßigkeiten mit hoher Schrittzahl

F.1 TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten mit hoher Schrittzahl

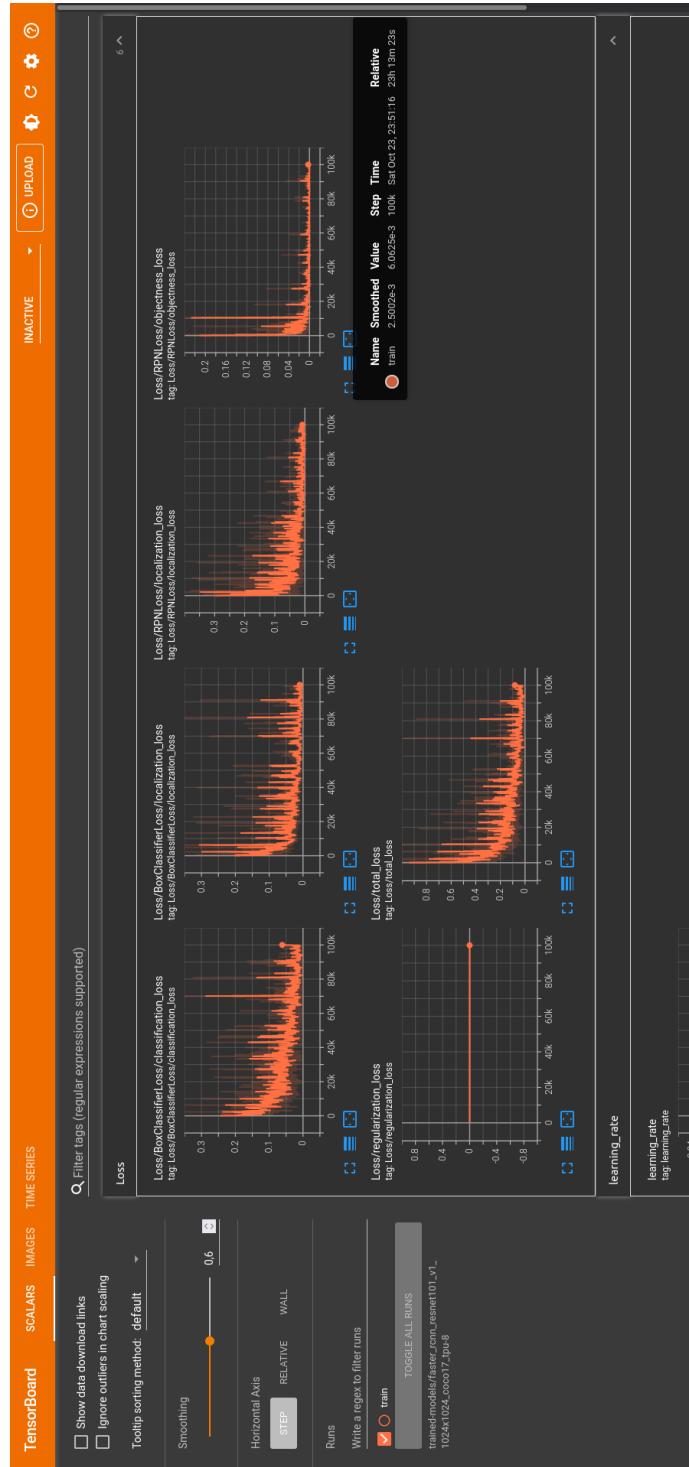


Abbildung 13: Hohe Schrittzahl: TensorBoard-Ausgabe für das Modell *faster_rcnn_resnet101_v1*

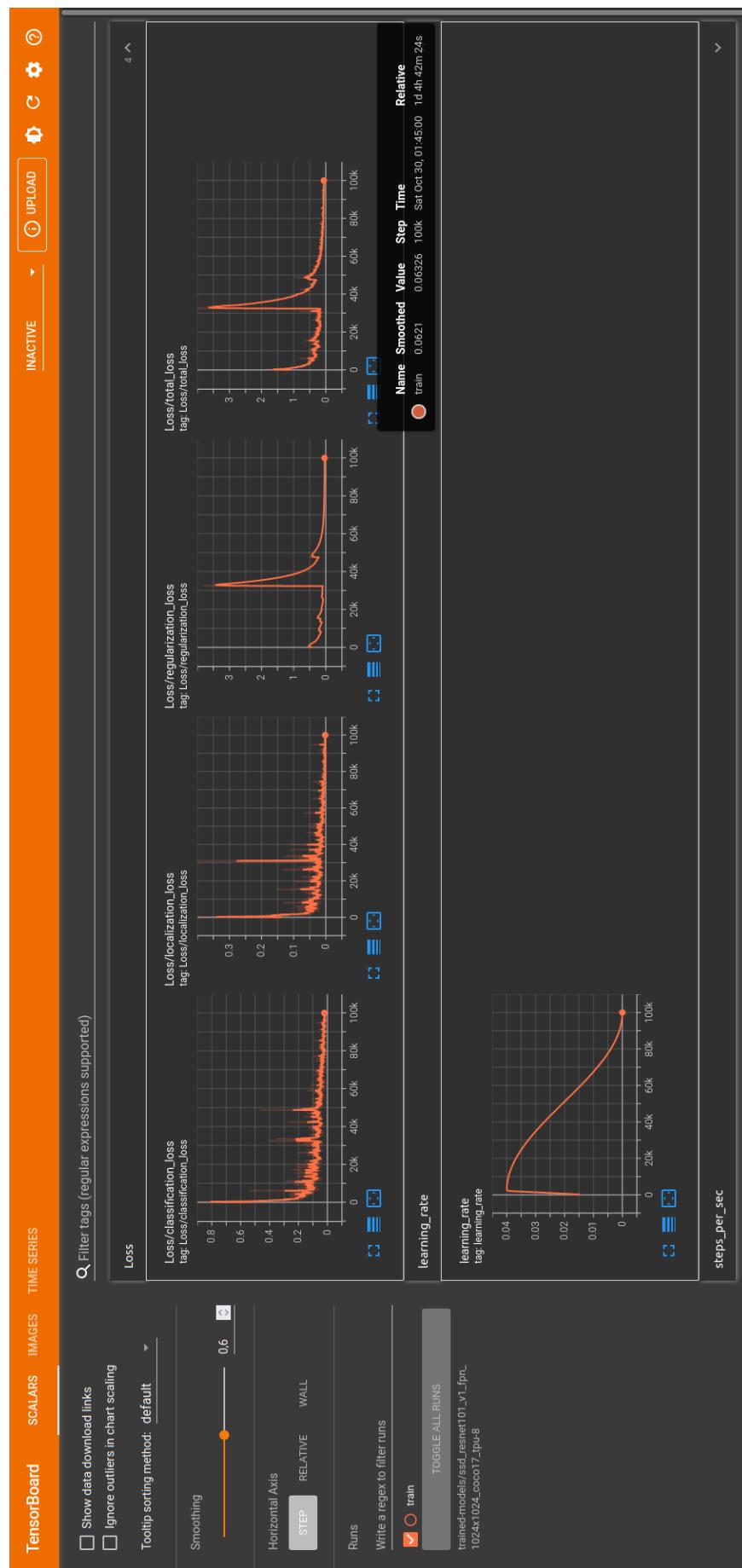


Abbildung 14: Hohe Schrittzahl: TensorBoard-Ausgabe für das Modell *ssd_resnet101_v1_fpn*

F.2 Die Detektionen

DVD : \workspaces\runtime_comparison\images\detections\

G Objekterkennung auf Testbildern für hochauflösende Süßigkeiten auf multiplen Hintergründen

G.1 TensorBoard-Ausgaben für das Training hochauflösender Süßigkeiten auf multiplen Hintergründen

DVD : \tensorboards\new_sweets_backgrounds

Diese Vorlagen werden für die Abschnitte 6.5 und 6.6 genutzt und daher in der Anlage H nicht nochmals aufgeführt.

G.2 Die Vorlagenbilder ohne Detektionen

DVD : \workspaces\new_sweets_backgrounds\images\images-to-detect

G.3 Die Detektionen

DVD : \workspaces\new_sweets_backgrounds\images\detections\

H Objekterkennung von Süßigkeiten nach Verbesserung

H.1 TensorBoard-Ausgaben für das Training von Süßigkeiten nach Verbesserung

DVD : \tensorboards\model_comparison

H.2 Die Detektionen

DVD : \workspaces\model_comparison\images\detections\

I Objekterkennung von Werkzeugen

I.1 TensorBoard-Ausgaben für das Training von Werkzeugen

DVD : \tensorboards\tools

I.2 Die Vorlagenbilder ohne Detektionen

DVD : \workspaces\tools\images\images-to-detect

I.3 Die Detektionen

DVD : \workspaces\tools\images\detections\

Anwendung der TensorFlow Object Detection API zur Detektion gesuchter Objekte auf Bildern

Using the TensorFlow Object Detection API to detect searched objects on images



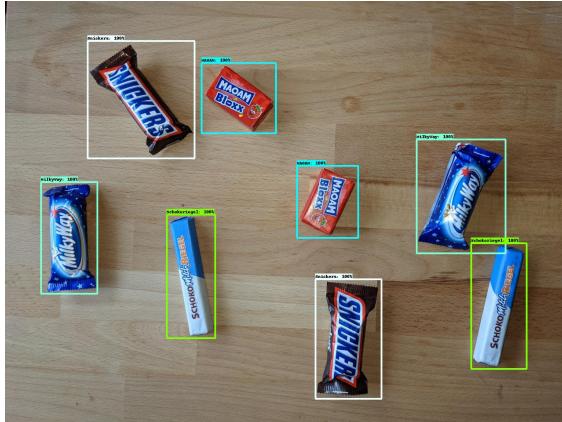
Technische
Hochschule
Georg Agricola

Von	Dennis Gardiner	By
Betreuer	Prof. Dr. Hubert Welp	Supervisor
Korreferent	M.Eng. Christopher Dillmann	Second reviewer
Bearbeitungszeit	01.10.2021 - 13.01.2022	Time frame

Diese Bachelorarbeit führt einen Anwender maschinellen Lernens zur Objektdetektion durch die nötigen Schritte zur Installation und Anwendung maschineller Lernalgorithmen mittels TensorFlow, auch ohne Vorkenntnisse über die programmiertechnischen Hintergründe.

Die Ergebnisse dieser Anwendung werden anschließend erläutert und mögliche Verbesserungen erarbeitet, getestet und die neuen, mit diesen Änderungen erzielten Ergebnisse diskutiert. Dabei werden sowohl Einflüsse durch die Einstellungen beim Training der verwendeten Modelle, als auch durch die Auswahl und Aufbereitung der Trainingsdaten berücksichtigt.

Abschließend werden die Erkenntnisse genutzt, um die ursprüngliche Objektdetektion zu optimieren und die Technik auf eine andere Aufgabenstellung zu transferieren. Gleichzeitig wird für zukünftige Anwender eine Zusammenfassung aus aufgetretenen Fehlern und Problemen sowie deren Lösungen bereitgestellt.



This bachelor thesis guides a user of machine learning for object detection through the necessary steps to install and apply machine learning algorithms using TensorFlow, even without prior knowledge of the used programming techniques.

Afterwards the results of this application are explained and possible improvements worked out, tested and the new results obtained with these changes are discussed. Influences due to the settings during the training of the used models as well as due to the selection and preparation of the training data are taken into account.

Finally, the lessons learned are used to optimize the original object detection and to transfer the technique to another task. At the same time, a summary of errors, problems and their solutions is provided for future users.

Erklärung

Ich versichere, die Arbeit selbständig angefertigt zu haben. Soweit ich von Firmen oder von anderer Seite erhaltene Unterlagen oder Veröffentlichungen, Untersuchungs - und Planungsarbeiten, Berechnungen, Risse, Zeichnungen, Pläne, graphische Darstellungen oder dergleichen - ganz oder teilweise - in die Arbeit übernommen habe, ist dies im Text oder auf den Anlagen vermerkt. Alle aus anderen Quellen sinngemäß oder wörtlich übernommenen Gedanken habe ich eindeutig als solche gekennzeichnet.

A handwritten signature in blue ink, appearing to read "Gardiner".

Dennis Gardiner

Bochum, den 13.01.2022