KU LEUVEN

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN

CAMPUS DE NAYER SINT-KATELIJNE-WAVER

Mobile Deep Visual Detection and Recognition

Thijs VERCAMMEN

Promotor(en): Prof. dr. ir. Toon Goedemé

Co-promotor(en): Ing. Floris De Feyter

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Elektronica-ICT
ICT

Academiejaar 2021 - 2022

Inhoudsopgave

Lijst van figuren

Lijst van afkortingen

AI Artificiële intelligentie
API Application programing interface
CNN Convolutional Neural Network
CPU Central Processing Unit
DSP Digital Signal Processor
GPU Graphics Processing Unit
NNAPI Neural Network API
NPU Neural Processing Unit
ONNX Open Neural Network Exange
Opset Operation Set
R-CNN Region Based Convolutional Network
ReLu Rectified Linear Unit
RoIs Region of Intrests
RPN Region Proposal Network
SVM Support Vector Machine
SSD Single Shot Detection
TFLite TensorFlow Lite
YOLO You Only Look Once

Hoofdstuk 1

Situering en doelstelling

1.1 Situering

Tegenwoordig wordt deep learning steeds meer en meer gebruikt om beeldverwerking problemen op te lossen. Via neurale netwerken kunnen we met meer en betere features werken om de afbeeldingen te analiseren. Maar veel van deze modellen hebben behoorlijk wat rekenkracht en geheugen nodig om te werken. Ook is er steeds meer interesse naar real-time toepassingen waarvan het resultaat zo snel mogelijk beschikbaar moet zijn. Dit wordt moeilijk bij veel hedendaagse systemen waarbij de foto eerst genomen moet worden en vervolgens door een computer geanalyseerd moet worden, omdat hedendaagse systemen veel rekenwerk en geheugen vragen.

Het automatisch detecteren en herkennen van producten in de schappen van een supermarkt heeft veel interessante real-time toepassingen. Het kan mensen helpen om snel de producten te vinden die ze nodig hebben. Ook kan de winkelmanagement zo een real-time status krijgen van de inventaris op de winkelrekken. Er kan ook nagekeken worden dat de producten staan waar ze horen te staan in de schappen. Voorgaande studies hebben reeds het potentieel van deep neurale netwerken laten zien voor deze taken. Maar er zit veel complex rekenwerk en geheugen vereisten achter het neurale netwerk voor deze toepassing. Deze beperking zorgen voor een groot struikelblok voor real-life toepassingen. Het zou namelijk handig zijn dat het neuraal netwerk kan worden uitgevoerd op een smartphone. In deze masterproef wordt er onderzocht hoe een bestaand neuraal netwerk kan worden aangepast zodat dit bruikbaar is voor een mobiele implementatie.

1.2 Probleemstelling

Mobiele apparaten zijn kleine toestellen met beperkt geheugen en beperkte rekenkracht. In deze masterproef wordt er onderzocht hoe het rekenwerk beperkt kan worden zodat het resultaat real-time geleverd kan worden. Er gaat ook onderzocht worden hoe alle data efficiënt kan worden opgeslagen op het toestel.

Bij deze masterproef willen we van een neuraal netwerk dat in een bepaald framework gemodel-

leerd en getraind is gaan naar een framework dat mobiele implementaties ondersteund. Zo wordt er voor een aantal verschillende frameworks onderzocht op welke manieren deze naar een mobiele implementatie kunnen gaan. Ook zal er per framework de omzetting naar mobiele implementatie voor verschillende CNN architecturen onderzocht worden. Vervolgens zal er ook gekeken worden naar verdere optimalisaties voor herkeningssystemen en detectiesystemen. Het uiteindelijke doel is een prototype applicatie ontwikkelen die het bestaande productherkenning netwerk implementeert op een mobiel apparaat. Waarbij het rekenwerk en de geheugen vereisten zijn geminimaliseerd zonder een groot effect te hebben op de accuraatheid van het model.

In deze masterproef zal het vooral gaan over het optimaliseren van een bestaand neuraal netwerk en het model omzetten naar verschillende frameworks. We gaan ervan uit dat in deze masterproef het model van het neuraal netwerk reeds is gemodelleerd en getraind.

1.3 Doelstellingen

Het uiteindelijke doel van deze masterproef is er voor zorgen dat een bestaand deep learning model aangepast kan worden zodat dit real-time resultaten kan geven op een mobiel apparaat. Dit gebeurt aan de hand van de volgende stappen:

- grondig begrijpen van een deep learning herkenningssysteem.
- grondig begrijpen van een deep learning detectiesysteem.
- Bespreken van frameworks/bibliotheken waarin het basis model ontwikkeld kan worden.
- Bespreken van frameworks die modellen optimaliseren en ondersteuning bieden voor mobiele implementaties.
- mogelijke manieren onderzoeken om een bestaand model te optimaliseren/converteren naar een model voor mobiele implementatie.
- onderzoeken voor optimalisaties voor herkenningssystemen en detectiesystemen.
- gevonden technieken testen en analiseren voor verschillende frameworks en neurale netwerken.
- prototype applicatie ontwerpen voor een mobiel apparaat.

Hoofdstuk 2

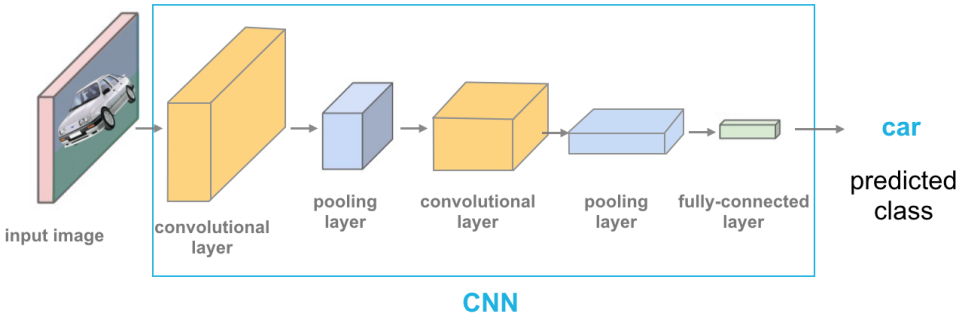
Herkenning en Detectie Algemeen

2.1 Deep learning-gebaseerde herkeningssystemen

Herkeningssystemen voorspellen wat de klasse van een object is in een afbeelding. Dit is het herkennen van objecten in digitale afbeeldingen zonder deze te lokaliseren of aan te duiden. Bij herkeningssystemen is er geen of weinig overlap tussen de trainingsafbeeldingen en de inputafbeeldingen. Bijvoorbeeld bij een gezichts herkenningssysteem wordt er een algemeen herkeningssystemen ontworpen dat gezichten herkent, en niet een systeem dat elk individueel gezicht herkent. Voor een herkenningssysteem is er een goed getraind netwerk nodig dat input afbeeldingen omzet in features. Er moet een database zijn met daarin de gegevens van de objecten die men wilt herkennen. Vervolgens hebben is er ook een methode nodig om features van het neuraal netwerk te vergelijken met de gegevens in de database om het juiste object te herkennen.

2.1.1 Herkenning

Wanneer er een getraind CNN is kan er een herkenningssysteem ontwikkeld worden. Als men bepaalde objecten in een afbeelding wil ontdekken gaat men met behulp van een CNN de afbeelding omzetten in een embedding. Embeddings Koehrsen (2018) zijn vector representaties die kunnen worden vergeleken in een embedding space, waar gelijkaardige objecten dichter bij elkaar liggen. De embedding van de input afbeelding wordt vergeleken met de embeddings die zich in een gallerij bevinden. Jiang et al. (2019) geeft aan dat met behulp van een query er gelijkaardige objecten uit de gallerij gehaald kunnen worden om deze vervolgens te gaan vergelijken in een embedding space. De gallerij is een database/verzameling met gekende embeddings/ID's van de objecten die men wilt herkennen. Een query is een embedding van de input waarvan het label niet gekend is. Gelijkaardige embeddings kunnen gezocht worden via de nearest neighbour techniek, waar we naar de klasse van de dichtsbijzijnde buur gaan kijken.

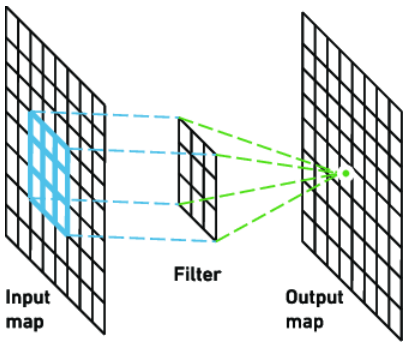


Figuur 2.1: CNN met 2 convolutie lagen en 2 pooling lagen en een fully-connected layer

2.1.2 convolutioneel neuraal netwerk (CNN)

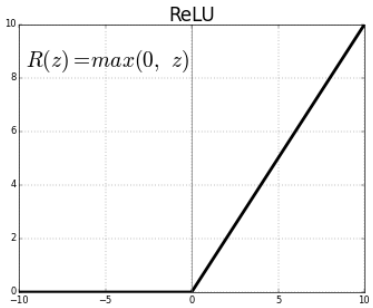
De belankrijkste bouwsteen van een herkenningssysteem is een goed getrainde CNN. In dit hoofdstuk wordt het CNN besproken dat beschreven wordt door Jiang et al. (2019). Het algemeen model van een CNN is weergegeven in figuur 2.1. In tegenstelling tot fully connected netwerken wordt bij een CNN de gewichten gedeeld over verschillende locaties om zo het aantal parameters te verminderen.

Het belangrijkste deel van een CNN zijn de convolutielagen (figuur 2.2). Waarbij men een kernel/filter over de input laat gaan wat als output een feature map genereerd. Een kernel bestaat uit een set van gewichten die met de input worden vermenigvuldigd. Deze kernel wordt over de input afbeelding geschoven om features te bepalen. Al de pixels binnen het veld van de kernel worden gereduceert tot een enkele waarde. CNN leren verschillende features met verschillende kernels in parallel. Waardoor de matrices met feature mappen steeds kleiner worden maar ook dieper worden. Een andere factor van een convolutie laag is de stride. Deze waarde geeft aan met hoeveel pixels de kernel telkens moet doorschuiven. Als de stride 1 is schuift de kernel steeds op met 1 pixel en als de stride 3 is schuift de kernel op met 3 pixels. Stride 1 zorgt voor meer features maar maakt het CNN trager omdat er dan meer bewerkingen moeten worden uitgevoerd. Een CNN bestaat uit een opeenvolging van een aantal convolutie lagen die steeds meer high-level features extraheren. Hoe meer convolutielagen een netwerk telt hoe meer features er uit de input worden gehaald, maar hoe trager het netwerk is.



Figuur 2.2: Convolutie laag waarbij een filter wordt herleid tot een output feature.

Elke convolutie laag wordt gevolgd door een niet-lineare activatie functie (Nwankpa et al. (2018)). De meest gebruikt functie hiervoor is de rectified linear unit (ReLu) (figuur 2.3). De ReLu wordt vaak gebruikt omdat deze eenvoudig is. De ReLu kan exact 0 weergeven en ziet er lineair uit. Max(0,x) is de ReLu bewerking, dus er wordt verdergegaan met 0 of de input waarde. Zonder een niet-lineare activatie functie kan het CNN herleid worden tot 1 convolultie laag die geen high-level features kan extraheren. Andere mogelijkheden voor Lineare activatie functies zijn: Sigmoid en Tangens hyperbolicus maar deze functies vragen meer rekenwerk.



Figuur 2.3: ReLu, waarbij het maximum wordt genomen van 0 en de input waarde.

Een volgende bouwsteen is de pooling laag waarbij het aantal samples in de feature map wordt verlaagt. De meest voorkomende methode is max-pooling waarbij er verder wordt gegaan met de maximum waarde in een bepaalde regio. Het doel van een pooling laag is om het aantal parameters te verminderen en zo ook het rekenwerk te verminderen. Er kan ook gebruik gemaakt worden van avarage pooling waarbij er verder wordt gegaan met de gemiddelde waarde van een regio. Er is ook minimal pooling waarbij er verder wordt gegaan met de minimum waarde.

Op het einde van elk CNN volgen er meestal 1 of meerdere fully connected lagen. Deze lagen connecteren elke input van één laag met elke activatie eenheid van de volgende laag. Dit zorgt voor meer parameters en meer rekenwerk maar wel meer features. Door het extra rekenwerk vormen deze lagen een vertragende factor. De fully connected lagen zorgen voor een classificatie op basis van de features van de convolutie lagen.

2.1.3 Trainen van een CNN

Het trainen van een CNN bestaat uit het leveren van veel voorbeelden aan het netwerk. Op basis van het resultaat van deze voorbeelden worden telkens de gewichten van de kernels aangepast, zodat er steeds een beter resultaat wordt geleverd.

De loss functie geeft de error van de voorspelling weer tijdens het trainen van een neuraal netwerk. Op basis van de loss functie gaat men via de stochastic gradiënt decent de gewichten van het netwerken bijstellen zodat bij een volgende trainingsinput de loss functie een beter resultaat geeft. Bij de stochastic gradiënt decent wordt er per batch/group trainingsvoorbeelden de gewichten bijgesteld. De gradienten worden berekend door de loss af te leiden naar de gewichten via de ketting regel, en de gewichten worden bijgesteld volgens de tegengestelde gradiënt.

De learning rate bij het trainen van een CNN beïnvloed de grootte van de stap waarmee de gewichten worden bijgesteld. Hoe kleiner de learning rate hoe langer het trainen van een CNN duurt. Maar als de learning rate te hoog is kan het resultaat een slecht getraind netwerk zijn, omdat de veranderingen op de gewichten te groot is om een beter resultaat te verkrijgen.

2.1.4 Transfer Learning

Bij transfer learning (Geiger et al. (2013))wordt er verder gebouwd op een model dat reeds getrained is. Hierbij wordt een model als basis gebruikt waarvan de toepassing gerelateerd is aan de gewilde toepassing. Bijvoorbeeld een model dat dieren kan herkennen wordt gebruikt als basis voor een model dat alleen honden herkent. Op dit basis model wordt er verdergetrained met een dataset specifiek voor de gewilde toepassing. Deze dataset kan veel kleiner zijn dan een dataset die gebruikt wordt om een nieuw model te trainen. Het trainen van een nieuw CNN kan soms weken duren. Via transfer learning kan de trainings periode met een grootte factor gereduceert worden. Deze methode wordt voornamelijk gebruikt om een 'nieuw' model te training vanwege de kleinere trainingsdataset en kortere trainingstijd.

2.2 Deep learning-gebaseerde detector

Object detectie is het lokaliseren en classificeren van objecten in een afbeelding, waarbij de objecten aangeduid worden met een Bounding box. Door gebruik te maken van CNN kunnen er vrij nauwkeurige object detectoren ontworpen worden. De classieke versie van object detectie is de sliding window benadering. Waarbij een venster met vaste grootte over de afbeelding schuift en telkens de gegevens binnen het venster analyseert. Vandaag de dag kan object detectie worden opgedeeld in twee methodes: de single-stage detector en de two-stage detector.

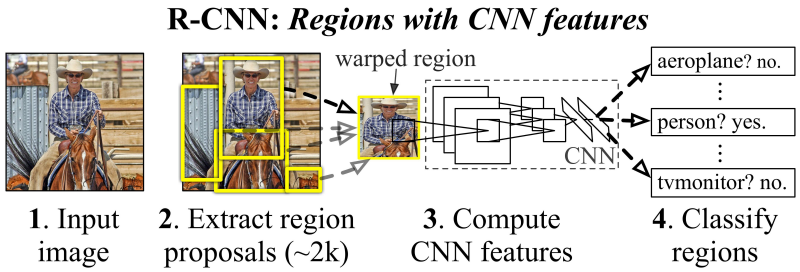
2.2.1 Two-stage detector

Two stage detectoren focussen op accuraatheid ten koste van de uitvoeringssnelheid. Zoals de naam zegt bestaat deze methode uit 2 niveaus. In het eerste niveau worden er Regions of Intrest (RoIs) gecreëerd. Dit is het filteren van regio's waarbij de kans groot is dat deze een object bevatten. Het tweede deel classificeert en verfijnt de lokalisatie van de RoIs die in het eerste deel gecreëerd werden. Dit gebeurt door elk van de RoIs door een CNN te voeren. Region-based Convolutional Neural Network (R-CNN) Girshick (2015) is het basis principe van de two-stage detectoren weergegeven in figuur 2.4. Hierbij wordt met een region proposal algoritme regio's uit de afbeelding gefilterd waar de kans groot is dat er objecten op staan. R-CNN bestaat uit 3 stappen:

1. Via een selective search algoritme Uijlings et al. (2013) worden er ongeveer 2000 mogelijke regio's met objecten geselecteerd.

2. Elke mogelijke regio wordt omgezet naar een feature vector via een CNN.

3. Elke regio wordt vervolgens geclassificeerd met een klas-specifieke Support Vector Machine (SVM) (Noble (2006)).

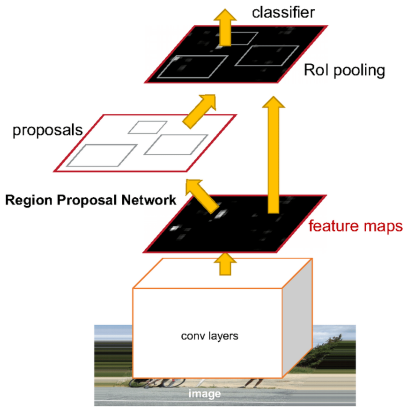


Figuur 2.4: R-CNN

R-CNN is een trage detector vermits elke RoIs door een CNN moet gaan. Deze methode is geëvolueerd tot de veel snellere methode Faster R-CNN Ren et al. (2016). Hierbij wordt de afbeelding door een CNN behandelt en vervolgens wordt er gebruik gemaakt van een Region Proposal Network (RPN) dit is weergegeven in figuur 2.5. Het RPN gaat zoals bij R-CNN regio's uit de afbeelding filteren waar de kans groot is dat er objecten opstaan. Maar het RPN werkt sneller en levert betere resultaten dan het region proposal algoritme.

Het RPN is een fully convolutional network dat per input een set van regio's als output geeft. Elk van deze regio's heeft een objectness score wat een maat is voor het object t.o.v. de achtergrond in de afbeelding. Om een region proposal te genereren wordt het RPN over de feature map geschoven die gegenereerd is door het voorgaande CNN. Op elke sliding window locatie worden er meerdere regio voorspellingen gedaan. Deze voorspelling wordt gedaan door verschillende anker boxen te evalueren.

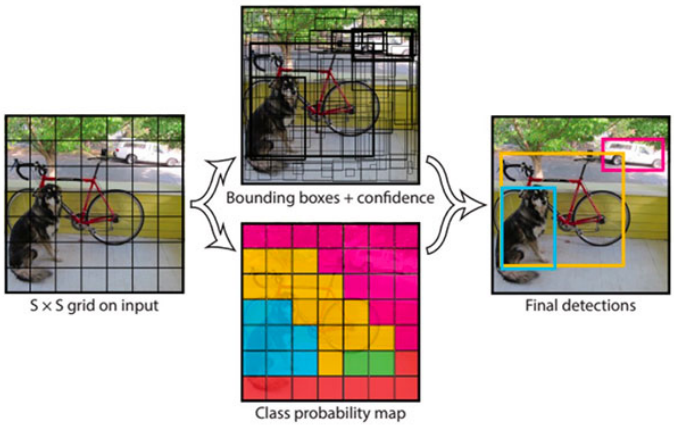
Vervolgens worden RoIs omgezet naar een feature vector met vaste lengte door RoI pooling. Elk van deze features gaat door een set van fully-connected lagen die 2 lagen als output heeft. een softmax laag die de klasse voorspelt, en een bounding box regressie laag die de bounding box voorspelt.



Figuur 2.5: Faster R-CNN

2.2.2 One-stage detector

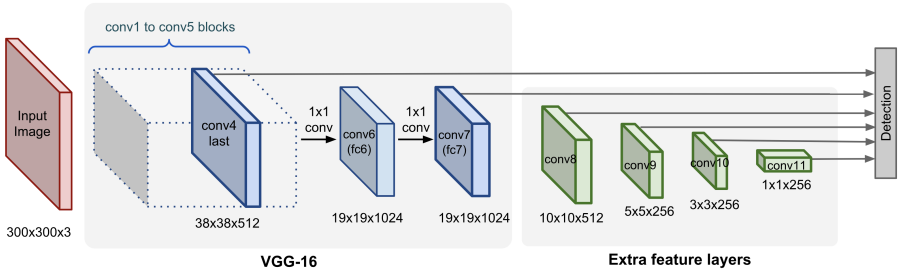
Bij one-stage detectoren gebeurt object detectie in één keer met één neuraal netwerk. Dus er is geen region proposal niveau meer zoals bij de two-stage detector. Deze detector gebruiken minder geheugen en rekenkracht t.o.v. two-stage detectoren. Dankzij deze verbeteringen zijn one-stage detectoren sneller dan two-stage detectoren omdat ze alles in één keer doen. Maar deze detectoren kunnen wat in nauwkeurigheid verliezen t.o.v. two-stage detectoren. Deze detectoren zijn zeer geschikt om gebruikt te worden op mobiele apparaten, omdat deze detectoren sneller zijn en minder geheugen nodig hebben. Twee veel gebruikte technieken van one-stage detectie zijn: You Only Look Once (YOLO) en Single Shot Detection (SSD).



Figuur 2.6: YOLO waarbij de input is opgedeeld in een S x S rooster. En waarbij bounding box voorspellingen zijn gedaan.

YOLO Redmon et al. (2016) verdeelt de afbeelding in een S x S rooster zoals in figuur 2.6 te zien is. De cel waarin het middelpunt van het object valt is verantwoordelijk voor de object detectie. Elke cel voorspelt K aantal bounding boxes en een score die aangeeft hoe zeker het model is dat een bepaalde bounding box een object bevat. Vervolgens is er nog een methode nodig om de overbodige bounding boxen te verwijderen. Een eerste mogelijkheid is door enkel bounding boxen te bepalen waarvan de voorspelling boven een bepaalde treshold ligt. Een andere methode is non-maxima supression deze methode zorgt ervoor dat elk object maar één bounding box heeft. Deze techniek houdt enkel de bounding box over met de beste voorspelling en onderdrukt de rest van de bounding boxen.

SSD Liu et al. (2016) is een one-stage detector (figuur 2.7) waarbij een afbeelding door verschillende convolutielagen gaat, wat als resultaat feature mappen op verschillende schalen oplevert. Op elke locatie van deze feature mappen van verschillende schalen wordt een vaste set van bounding boxen geëvalueerd. Voor elk van deze boxen wordt de zekerheid dat het een object bevat voorspelt. Op het einde wordt non maximum suppression gebruikt om de finale voorspelling te maken. Het netwerk van een SSD bestaat uit een basis netwerk dat gevormd wordt door een standaard classificatie netwerk zonder de fully-connected lagen. Vervolgens worden er extra convolutie lagen toegevoegd, wat het model toelaat om voorspellingen te doen op verschillende schalen.



Figuur 2.7: One-stage detector met VGG net backbone

Hoofdstuk 3

Herkenning en detectie implementatie op mobiel platform

Dit hoofdstuk zal gaan over het implementeren van deep learning herkeningssystemen en detectiesystemen op een mobiel platform. Er wordt besproken welke technologiën er gebruikt kunnen worden om een CNN te implementeren op een mobiel platform. Dit zullen een aantal frameworks en bibliotheken zijn die de programmeur in staat stelt om een bestaand model te implementeren op een mobiel apparaat. Deze frameworks gaan vergeleken worden om te kijken van welk framework het best gebruik wordt gemaakt voor een bepaalde toepassing. Er wordt ook onderzocht hoe een bestaande herkeningssystemen en detectiesystemen geöptimaliseerd kunnen worden zodat deze gebruikt kunnen worden op een mobiel platform. Bij het uitvoeren van een neuraal netwerk op een mobiel apparaat zal men rekening moeten houden met gelimiteerde rekenkracht en het beschikbaar geheugen. Ook moet er rekening gehouden worden met een beperkte batterij. Want CNN's voeren veel berekeningen uit wat meer energie verbruikt. Er zullen een aantal technieken besproken worden die men kan gebruiken om het geheugen van het model, het aantal bewerkingen en het energieverbruik te kunnen verbeteren. Er zal voornamelijk gefocust worden op Android implementaties.

3.1 Frameworks

Om machine learning modelen te ontwerpen en te trainen kan er gebruik gemaakt worden van frameworks. Deze frameworks geven de programmeur een set van tools die hun in staat stelt om op een overzichtelijke en flexibele manier machine learning modellen te ontwerpen en trainen. TensorFlow (Abadi et al. (2016)) en PyTorch (Li et al. (2020)) zijn de 2 voornaamste frameworks die gebruikt worden om neurale netwerken te ontwerpen. Veel van de tools en bibliotheken die gebruikt worden om herkeningssystemen en detectiesystemen te ontwerpen worden bovenop deze frameworks gebruikt. Er zijn veel discussies over welk van deze 2 frameworks het beste wordt gebruikt voor het ontwerpen van neurale netwerken. Om deze redenen worden de 2 frameworks besproken en wordt er gekeken welke mogelijkheden elk framework heeft voor mobiele implementatie.

3.1.1 TensorFlow

TensorFlow (Abadi et al. (2016)) is ontworpen door Google en is een open source framework voor machine learning implementaties dat focust op het ontwerpen, trainen en het deployen van neurale netwerken. Ook ondersteund TensorFlow meerdere programmeertalen zoals: Python, Java en C.

Door de introductie van de Keras API (Keras (2021)) is TensorFlow meer gebruiksvriendelijk geworden. Keras is een framework dat bovenop TensorFlow gebruikt kan worden, waarmee machine learning modellen kunnen ontworpen worden op een overzichtelijke manier. Het idee van Keras is om zo snel mogelijk van een idee naar een toepassing te gaan. Zo heeft TensorFlow een gebruiksvriendelijk API voor eenvoudige projecten en meer uitgebreide tools voor complexe projecten. Ondertussen is Keras geïntegreerd met het TensorFlow framework.

Programmeurs kunnen op een grafische manier de operaties voorstellen die moeten worden uitgevoerd. Hierbij bestaat de grafische voorstelling uit nodes, waarbij elke node een wiskundige operatie is. Zo krijgt de programmeur een duidelijk overzicht over de gehele toepassing. Door gebruik te maken van TensorBoard (Abadi et al. (2015)) kan data op een flexibele manier gevisualiseerd worden tijdens het ontwerpen van een machine learning model. TensorFlow biedt ook goede ondersteuning op gebied van deployment van een model.

De TensorFlow Object Detection API (Abadi et al. (2015)) is een open source framework dat bovenop TensorFlow werkt. Het maakt het voor de programmeur gemakkelijker om object detectie modellen te ontwerpen, trainen en deployen. Deze API voorziet een set van meer dan 40 modellen voorgetraind op de COCO dataset waarop de programmeur verder kan bouwen. Deze modellen maken gebruik van de volgende detectiemodellen: CenterNet (Duan et al. (2019)) RetinaNet (Lin et al. (2018)), EfficientDet (Tan et al. (2020)), SSD (Liu et al. (2016)) en Faster-RCNN (Ren et al. (2016)).

Voor mobiele implementatie biedt TensorFlow de mogelijkheid om een TensorFlow Lite (TFLite) (Abadi et al. (2015)) model te ontwerpen TFLite zorgt ervoor dat het model een lage latency heeft en een kleine binaire grootte. Het TFLite model kan geïmplenteerd worden op zowel Android als IOS applicaties. De meeste detectie modellen ontworpen met TensorFlow object detection API zijn niet bedoeld om op mobiele apparaten te werken. Deze modellen moeten eerst geconverteerd worden naar een TFLite model. De eerder besproken TensorFlow Object Detection API heeft ondersteuning om een model te exporteren naar een TFLite model. Vervolgens is er ook de optie om de TensorFlow Object Detection API te implementeren in Android Studio, zodat het TFLite object detectie model op eenvoudige manier geïmplenteerd kan worden. Het converteren kan makkelijk worden uitgevoerd via de volgende lijnen code in Python.

Listing 3.1: Converteren van TensorFlow naar een TensorFlow Lite model

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()
```

De TFLiteConverter neemt een TensorFlow model als input en op basis van dit model wordt een converter object gegenereerd. Het TFLite model is eigenlijk een geoptimaliseerde FlatBuffer (Google (2014)), deze kan herkend worden door de .tflite extensie. FlatBuffer is serialisatie bibliotheek waarbij het object niet verpakt wordt en dus rechtsreeks uitleesbaar is voor verdere toepassingen. Omdat het FlatBuffer object rechtsreeks uitleesbaar is zorgt dit voor een optimale snelheid en een efficiënt geheugen gebruik. Het FlatBuffer formaat is niet python afhankelijk en is bruikbaar over verschillende platformen zoals: C, Java en Python. Tijdens de TFLite conversie worden er ook een aantal verdere optimalisaties uitgevoerd om het model kleiner en sneller te maken via Grappler (Abadi et al. (2015)). Grappler is het standaard optimalisatie systeem van TensorFlow om automatisch het model te verbeteren. Grappler voert de volgende optimalisaties uit:

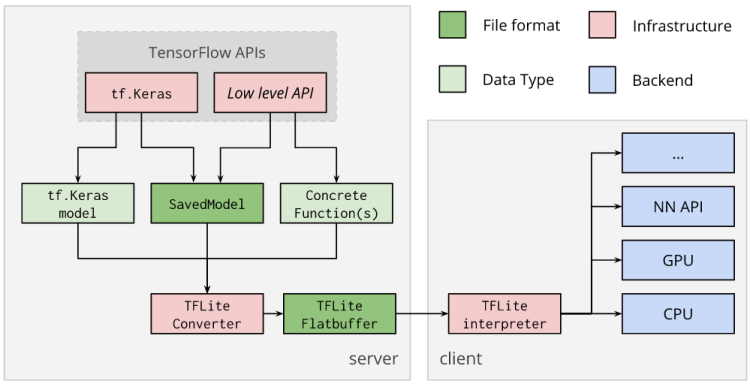
- Constant Folding optimizer: Is het vervangen van expressies/nodes waarvan de waardes niet veranderen tijdens de uitvoering door constanten.
- Arithmetic optimizer: Vermindert de complexiteit van operaties en verwijdert gemeenschappelijke subexpressies.
- Layout optimizer: Optimaliseert de data layout van tensoren zodat opperaties die afhankelijk zijn van het data formaat efficiënt gebeuren.
- Remapper optimizer: Deelnodes die standaard voorkomen in het model worden samengevoegd tot 1 operatie.
- Memory optimizer: Analyseert het geheugengebruik van operaties in het model en voegt bij zwaar geheugen gebruik swap operaties toe tussen CPU en GPU.
- Dependency optimizer: Het verwijderen of herschikken controle dependencies om het kritische pad van een model stap te verkorten.
- Pruning optimizer: het verwijderen van nodes in het model die geen invloed hebben op de output.
- Function optimizer: optimaliseert de functie bibliotheek van TensorFlow.
- Loop optimizer: Verbetert loops door redundante stack operaties te verwijderen en het verwijderen van 'dode' statische aftakingen.

De TFLite bibliotheek ondersteund een gelimiteerd aantal operaties van de standaard TensorFlow operaties. Hierdoor is het mogelijk dat bij complexere modellen de TFLite conversie niet altijd succesvol is. Omdat een bepaalde TensorFlow operatie geen TFLite tegenhanger heeft. Het is mogelijk om TensorFlow operaties te gebruiken die niet ondersteund worden door TFLite door dit mee te geven met de TFLiteConverter. Dit heeft als gevolg dat de TensorFlow Core bibliotheek mee geïmplementeerd moet worden waardoor de bestandsgrootte van het TFLite model zal toenemen.

Listing 3.2: Het gebruik van standaard TensorFlow operaties toelaten in het TFLite model

```
converter.target_spec.supported_ops = [
        tf.lite.OpsSet.TFLITE_BUILTINS, # laat TFLite operaties toe.
        tf.lite.OpsSet.SELECT_TF_OPS # laat standaard TenserFlow operaties toe.
  ]
```

Als er geen nieuw object detectie model ontworpen moet worden en het is voldoende om een model te hertrainen. Dan is het mogelijk om gebruik te maken van TFLite Model Maker (Abadi et al. (2015)). Dit geeft de mogelijkheid om een TFLite model te hertrainen met eigen dataset, waardoor de conversie stap overgeslagen kan worden. Hierbij kan er voor object detectie enkel gebruik gemaakt worden van een EfficientDet detector (Tan et al. (2020)).



Figuur 3.1: Aan de linker kant is te zien dat een TensorFlow model via de TFLite converter wordt omgezet in TFLite flatbuffer model. Het TFLite flatbuffer model kan vervolgens geïmplemnteerd worden op een client toestel waar er gebruik gemaakt kan worden van verschillende hardware componenten

Het framework geeft ook de mogelijkheid om TFLite model verder te verbeteren via hardware acceleraties en model optimalisaties. De model optimalisaties die TFLite ondersteund zijn: weight pruning, quantisatie en clustering. De quantisatie optimalisatie kan als optie worden meegegeven aan de TFLiteConverter. Weight pruning en clustering moeten uitgevoerd worden voordat het model geconverteerd wordt naar TFLite. Wat deze optimalisatie technieken exact inhouden wordt later besproken in paragraaf 3.5. De uitvoering van het TFLite model dat normaal op de CPU wordt uitgevoerd, kan versneld worden door gebruik te maken van verschillende hardware componenten die op het toestel aanwezig zijn. Elk van deze hardware componenten maakt gebruik van een eigen API waardoor er extra code moet worden geschreven specifiek voor elke hardware component. De TensorFlow Lite Delegate API (Abadi et al. (2015)) dient als een interface genaamd delegates voor deze componenten. Deze delegates ervoor dat er geen specifieke API geímplemnteerd moet worden voor een bepaalde hardware component. TFLite ondersteund meerdere delegates die kunnen geöptimaliseerd worden voor een specifiek platform. Voor Android kan er gebruik gemaakt worden van de GPU delegate en de Neural Network API (NNAPI) delegate. De NNAPI (Android (2021)) is ontworpen om intensieve berekeningen uit te voeren op een Android apparaat. Deze API kan wor-

den aangeroepen door deep learning applicaties om gebruik te maken van de GPU, DSP of NPU. In figuur 3.1 is een algemene workflow te zien om van een TensorFlow model naar een mobiele implementatie te gaan.

TensorFlow biedt ook de mogelijkheid om via TensorFlow.js een model te implementeren via javascript. Op deze manier kunnen modellen gebruikt worden in de browser en met Node.js. Via de TensorFlow.js API kan een TensorFlow model geconverteerd worden naar een TensorFlow.js model.

Listing 3.3: Converteer TensorFlow model naar TensorFlow.js

```
tensorflowjs_converter --input_format=<tf_saved_model> \
                       --output_node_names= <output_nodes>
```

3.1.2 PyTorch

PyTorch (Li et al. (2020)) is ontworpen door facebook en wordt zoals TensorFlow ook gebruikt om machine learning modellen te ontwerpen, trainen en deployen. Dit is een python gebaseerd framework dat focust op flexibiliteit. Door zijn flexibiliteit is het gemakkelijk om nieuwe functionaliteiten toe te voegen door bestaande code aan te passen of nieuwe code toe te voegen. Zoals TensorFlow kan het model worden voorgesteld op een grafische manier die bestaat uit nodes en waarbij elke node een operatie is. PyTorch maakt gebruik van externe tools zoals TensorBoard om data te visualiseren. PyTorch wordt voornamelijk gebruikt om te experimenteren met neurale netwerken. Bedrijven en onderzoekers gebruiken dit framework vooral om een CNN experimenteel op te bouwen en te trainen.

Via de Torchvision bibliotheek (Paszke et al. (2017)) die deel uitmaakt van het PyTorch project, heeft de programmeur toegang tot een set van hulpmiddelen om een deep learning model op te bouwen. Torchvision bevat populaire datasets, netwerk architecturen en technieken voor afbeelding transformaties. Voor object detectie biedt Torchvision ondersteuning voor de Faster-RCNN (Ren et al. (2016)), RetinaNet (Lin et al. (2018)), SSD (Liu et al. (2016)) en SSDlite (Sandler et al. (2019)) architecturen. Van deze architecturen zijn er een aantal voorgetrainde modellen terug te vinden in de Torchvision bibliotheek. Via transfer learning is de programmeur dan in staat om een eigen object detectie model te trainen.

PyTorch biedt zoals TensorFlow ook de mogelijk om het model te optimaliseren voor een mobiele implementatie. PyTorch mobile zit nog in zijn beta fase, dus er is een mogelijkheid dat er onverwachte complicaties optreden. Maar PyTorch Mobile wordt reeds gebruikt in een heel aantal toepassing waaruit we kunnen afleiden dat Pytorch mobile veilig gebruikt kan worden. De onderstaande lijnen code geven aan welke code er moeten worden uitgevoerd om een PyTorch model te converteren naar een PyTorch mobile model.

Listing 3.4: Converteren van PyTorch naar een PyTorch Mobile model

```
import torch
import torchvision
from torch.utils.mobile_optimizer import optimize_for_mobile

model.eval()
example = torch.rand(1, 3, 224, 224)
traced_script_module = torch.jit.trace(model, example)
traced_script_module_optimized = optimize_for_mobile(traced_script_module)
traced_script_module_optimized._save_for_lite_interpreter("model.ptl")
```

Voordat het PyTorch model kan geoptimaliseerd worden voor mobiel gebruik moet het model eerst omgezet worden in een ScriptModule. De ScriptModule is een serieel model dat verder geoptimaliseerd kan worden en Python onafhankelijk is. Dit geeft ook de mogelijkheid om een model dat ontwikkeld is in Pytorch te gebruiken in toepassingen die worden uitgevoerd in een andere omgeving. Via TorchScript kan het model worden omgezet naar een ScriptModule. De torch.jit.trace functie verwacht als input het model en een voorbeeld input in de vorm van een torch.Tensor. De trace functie voert het model uit met de meegegeven voorbeeld input en bewaart al de operaties die op de input tensors worden uitgevoerd. De bewaarde operaties worden gebruikt om een ScriptModule te genereren. Bij de trace functie worden enkel de uitgevoerde operaties bewaard, andere elementen van het model maken geen deel uit van de ScriptModule. Op de PyTorch documentatie (Paszke et al. (2017)) is een lijst terug te vinden met al de functies die ondersteund worden bij het omzetten naar een ScriptModule.

Eenmaal dat er een scriptmodule is gegenereerd kan dit gebruikt worden om het model te optimaliseren voor mobiel gebruik. De optimize_for_mobile functie zal de volgende optimalisaties automatisch uitvoeren.

- De Conv2D en BatchNorm operaties worden samengevoegd tot een Conv2D operaties met aangepasten gewichten.
- Vervang 2D convoluties en lineaire operaties met hun PyTorch Mobile equivalent.
- Activatie functies RELU en tanh die volgen op Conv2D en lineaire operaties worden samengevoegd met de voorgaande Conv2D of lineaire operatie.
- Het verwijderen van dropout nodes (Paszke et al. (2017)).

PyTorch biedt ook nog de mogelijkheid om verdere optimalisaties toe te voegen in de vorm van quantisatie na het trainen van het model. Zoals TensorFlow is er ook de mogelijkheid om de uitvoering van het model te versnellen door gebruik te maken van de NNAPI (Android (2021)).

3.2 Object detectie bibliotheken

Er zijn heel wat bibliotheken die de programmeur kan importeren om een detectiesysteem te ontwerpen en te trainen. Deze bibliotheken geven de programmeur extra tools en hulpmiddelen om de complexiteit van het process te verminderen en de code overzichtelijk te maken. De bibliotheken werken bovenop TensorFlow of PyTorch en geven de programmeur toegang tot een groter aantal verschillende object detectoren en voorgetrainde modellen.

3.2.1 MMDetection

MMDetection voorgesteld door Chen et al. (2019) maakt deel uit van OpenMMLab en is een open source object detectie toolbox gebaseerd op PyTorch. De toolbox bevat gewichten van meer dan 200 voorgetrainde modellen. Via modulair ontwerp kan het detectie framework opgesplitst worden in verschillende componenten. Met de verschillende componenten kan een eigen detectie model worden gemaakt door de verschillende componenten te combineren. MMDetection ondersteund ook 48 verschillende detectie methodes zoals: YOLO (Redmon et al. (2016)), Faster R-CNN (Ren et al. (2016)), en SSD (Liu et al. (2016)). Doordat al de bounding box en masker operaties worden uitgevoerd op GPU's heeft MMDetectie een grootte trainingssnelheid. MMDetection biedt geen ondersteuning om een bestaand model te optimaliseren naar een model voor mobiel gebruik. Omdat een MMDetection model operaties kan bevatten die een Pytorch model niet ondersteund, kan het zijn de de optimalisatie naar PyTorch mobile niet lukt. Dus dit model zal geconverteerd moeten worden naar een framework waarbij het optimaliseren naar een mobiel model wel mogelijk is.

3.2.2 Detectron2

Detectron2 (Wu et al. (2019)) is een bibliotheek ontworpen door Facebook die segmentatie en detectie algoritmes ondersteunt. Zoals MMDetection werkt Detectron2 bovenop Pytorch en kan het netwerk getraind worden op 1 of meerdere GPU's. Volgens de Detectron2 documentatie heeft Detectron2 een betere trainingssnelheid dan MMDetection: 62 afbeeldingen t.o.v. 53 afbeeldingen. Via Modular, extensible design kan Detectron2 specifieke modules toevoegen aan bijna elk deel van een object detectiesysteem. Detectron2 bevat meer dan 80 voorgetrainde modellen waarop de programmeur verder kan bouwen. Voor object detectie ondersteund Detectron2 6 verschillende standaard modellen. Ondertussen heeft Detectron2 een uitbreiding gekregen D2Go dat ondersteuning biedt voor object detectie voor mobiel gebruik.

3.2.3 GluonCV

GluonCV (Guo et al. (2020)) is een bibliotheek die implementaties levert voor enkele van de State-of-the-art deep learning modellen. Deze bibliotheek is ontworpen om snel prototypes en onderzoek ideeën te leveren voor onderzoekers. Met behulp van een API die de complexiteit van het implementeren minimaliseert. GluonCV biedt ondersteuning voor PyTorch en MXNet (Chen et al. (2015))

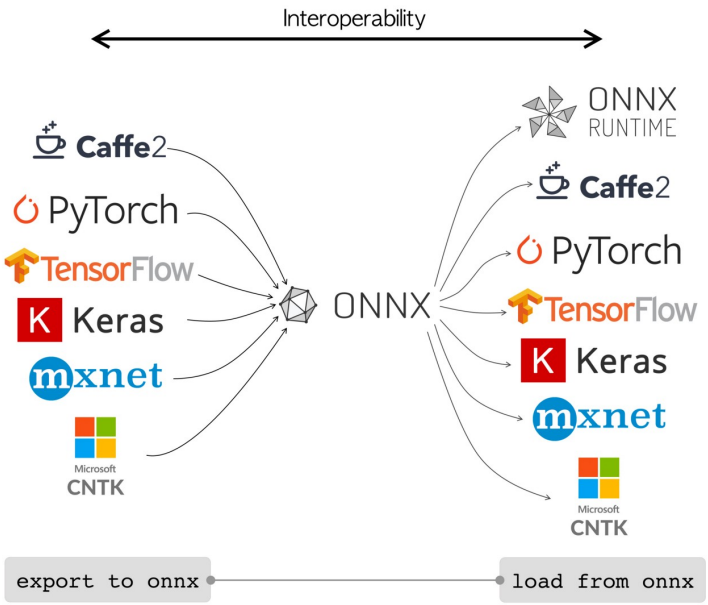
om deep learning modelen te ontwerpen en trainen, maar de GluonCV documentatie bevat meer informatie voor MXNet. Er kan ook gebruik gemaakt worden van meer dan 170 voorgetrainde modellen, die hertrained kunnen worden voor een specifiek doel. Voor object detectie biedt GluonCV ondersteuning voor 4 detectie modellen: SSD (Liu et al. (2016)), Faster-RCNN (Ren et al. (2016)), YoloV3 (Redmon and Farhadi (2018)) en CenterNet (Duan et al. (2019))

3.2.4 ImageAI

ImageAI (Martins et al. (2021)) is een makkelijk te gebruiken Python bibliotheek die de programmeur in staat stelt om State-of-the-art AI feautres te implementeren. Het doel van deze bibliotheek is om met enkele lijnen code een object detector te maken. Deze bibliotheek levert tools voor afbeelding herkening, object detectie en video analyse. ImageAI ondersteund object detectie door gebruik te maken van RetinaNet (Lin et al. (2018)) , YoloV3 (Redmon and Farhadi (2018)) en TinyYOLOv3 (Gai et al. (2021)) getraind met de COCO dataset. Deze bibliotheek werkt bovenop TensorFlow en sinds juni 2021 maakt ImageAI gebruik van een PyTorch backend. Ook deze bibliotheek geeft de programmeur de mogelijkheid om nieuwe modellen te trainen om specifieke objecten te detecteren.

3.3 Open Neural Network Exange (ONNX)

ONNX (2017) biedt de mogelijkheid om verschillende frameworks samen te laten werken. Hierbij wordt een bestaand model geconverteerd naar een ONNX model. Dit model kan op zijn beurt geconverteerd worden naar het gewilde framework (figuur: 3.2). ONNX ondersteund niet elk machine learning framework, maar toch de meest populaire. Het ONNX framework dient goed als een overkoepelend framework dat er voor zorgt dat verschillende frameworks compatibel zijn met elkaar. Volgens de website zijn er momenteel 23 frameworks die naar het ONNX framework geconverteerd kunnen worden. In deze frameworks kan bijvoorbeeld een CNN gemodelleerd en getrained worden. Om dan geconverteerd te worden naar een framework dat het model kan omzetten naar een mobile versie. Voor deze masterproef waarin een bestaand model ontworpen is in een bepaald framework. Zal ONNX een belangrijke rol spelen om het bestaand model te implementeren met een framework dat mobiele implementatie ondersteund. ONNX definieert een aantal gemeenschappelijke sets van opperaties en bouwblokken genaamd opsets. Deze opsets bevatten operaties die over verschillende frameworks gebruikt kunnen worden. Bij het converteren naar een ONNX modellen zullen de operaties die uitgevoerd worden in een bepaald framework vervangen worden door een ONNX equivalent gedefinieert in een opset versie. ONNX vernieuwt de opsets regelmatig, zo zal bij elke opset versie de operaties worden aangepast waardoor er operaties bijkomen, wijzigen en wegvallen. Niet elk framework ondersteund dezelfde opset versie en dit zou tijdens het exporteren naar ONNX voor problemen kunnen zorgen. Dit is omdat een bepaalde operatie in het orginele framework geen ONNX equivalent zou kunnen hebben. Voor elke bibliotheek/framework is het aangeraden om dezelfde opset-versie te implementeren. Anders zullen er dus operaties zijn die niet compatibel zijn met het gewilde framework voor mobiele implementatie.



Figuur 3.2: Een weergave van de voornaamste frameworks die naar ONNX kunnen exporten en die een ONNX model kunnen importern.

3.3.1 Onnxruntime

Het ONNX framework biedt via onnxruntime (ONNX (2019)) zelf ook de mogelijkheid om een model te deployen en te optimaliseren voor mobiel gebruik. Dit is mogelijk door het ONNX model te converteren naar een onnxruntime model via de volgende lijnen code.

Listing 3.5: Converteren van ONNX bestand naar onnxruntime bestand

```
!python -m onnxruntime.tools.convert_onnx_models_to_ort onnx_model.onnx
```

Zoals Pytorch en Tensorflow voert onnxruntime tijdens het converteren een aantal optimalisaties uit. Deze optimalisaties bestaan uit 3 niveaus: Basic, Extended en All. Standaard worden al de optimisatie uitgevoerd, de volgende lijst bevat al de optimalisaties die tijdens het converteren worden uitgevoerd.

- Basic: verwijderen van redundante nodes in het model en constant folding waarbij waarden door constanten worden vervangen zodat deze tijdens de uitvoering niet meer berekend moeten worden.
- Extended: één of meer ONNX standaard operaties samenvoegen tot één operatie
- All: optimaliseer afbeelding formaat door te converteren tussen NHWC gebruikt door ONNX en NCHW.

NHWC of NCHW formaat:

- N: Aantal afbeeldingen per groep.
- H: Hoogte van de afbeelding.
- W: Breedte van de afbeelding.
- C: Aantal kanalen van de afbeelding.

Onnxruntime geeft ook de mogelijkheid om het model verder te optimaliseren door gebruik te maken van NNAPI om het model sneller te maken. Het gegenereerde onnxruntime model kan vervolgens geïmplementeerd worden in android studio via de volgende lijnen code.

Listing 3.6: onnxruntime model implementeren in android studio

```
SessionOptions session_options = new SessionOptions();
session_options.addConfigEntry("session.load_model_format", "ORT");

OrtEnvironment env = OrtEnvironment.getEnvironment();
OrtSession session = env.createSession(<path to model>, session_options);
```

Het ONNX model kan ook worden geïmplementeerd via javascript, om vervolgens dit script te gebruiken in een webapplicatie. Via de volgende lijnen code kan het ONNX model worden uitgevoerd in javascript.

Listing 3.7: ONNX bestand implementeren in javascript

```
const ort = require('onnxruntime-node'); // bibliotheek inladen
const session = await ort.InferenceSession.create('./onnx_model.onnx');
const tensor = new ort.Tensor('float32', data, [3, 4]); // input tensor opstellen
const feeds = { input: tensor}; // input feeds maken, met input tensors
const results = await session.run(feeds); // model uitvoeren
```

3.3.2 TensorFlow naar ONNX conversie

Om een TensorFlow model naar ONNX formaat te exporteren wordt er gebruik gemaakt van de tf2onnx bibliotheek (ONNX (2021)). Via de volgende lijn code can de convert functie opgeroepen worden uit de tf2onnx bibliotheek.

Listing 3.8: Converteren van TensorFlow model naar ONNX model

```
!python -m tf2onnx.convert --saved-model <directory van opgeslagen model>
                    --output <onnx output file> --opset <opset versie>
```

De conversie van TensorFlow naar ONNX wordt voor opset versie 9 tot 15 ondersteund en getest. Opset versie 6 tot 8 zou ook ondersteund moeten worden maar hier zijn geen testen voor. Bij de

convertie wordt opset versie 9 standaard gebruikt. Voor een meer recente opset versie moet de opset versie specifiek worden meegegeven. In de Tf2onnx documentatie is een lijst terug te vinden met Tensorflow operaties die door ONNX worden ondersteund. Als het TensorFlow model operaties bevat die niet herkent worden door onnxruntime, dan is het mogelijk om deze operaties mee te geven met de conversie functie. De TensorFlow naar ONNX convertie bestaat uit de volgende stappen:

- De converter verwacht een Protobuf bestand (Google (2021)) als input, dit is een serieel formaat dat platform onafhankelijk is. Dis is omdat de converter constante variabelen verwacht van het opgeslagen model.
- Het TenserFlow protobuf formaat wordt geconverteerd naar het ONNX protobuf formaat zonder rekening te houden met specifieke operaties.
- De converter identificeert operaties en vervangt deze met het ONNX equivalent.
- Er wordt gekeken naar specifieke operaties die niet automatisch zijn vervangen door een ONNX equivalent. Bijvoorbeeld operatie relu6 wordt niet door ONNX ondersteund, maar kan vervangen worden door verschillende ONNX operaties te combineren.
- Het huidige ONNX model wordt verder geöptimaliseerd door bijvoorbeeld operaties samen te voegen of onnuttige operaties te verwijderen.

3.3.3 PyTorch naar ONNX conversie

De tools om een PyTorch model te converteren naar een ONNX model maken standaard deel uit van PyTorch. Zo kan via de volgende lijn code een PyTorch model geëxporteerd worden naar ONNX.

Listing 3.9: Converteren van PyTorch model naar ONNX model

```
torch.onnx.export(model, args, 'test.onnx')
```

De verplichte inputs nodig voor het uitvoeren van deze functie zijn: het PyTorch model, een set van input argumenten en een output bestand. De set van input argumenten bestaat uit een tuple van inputs of een dictionary die de input parameters van het model geeft. Bij het uitvoeren van de onnx.export functie zal het model worden omgezet in een TorchScript model via de eerder besproken trace functie. Hierbij zal het model worden uitgevoerd en worden al de uitgevoerde operaties bewaard. De input argumenten meegegeven met de export functie zullen gebruikt worden als input voor de trace functie. De operaties van het TorchScript model zullen dan vervangen worden door hun ONNX equivalent. De PyTorch documentatie bevat een lijst met al de operaties die ondersteund worden voor het uitvoeren van de export functie naar ONNX. Het is mogelijk om operaties die niet in deze lijst staan toe te voegen aan de PyTorch bron code als de operatie ondersteund wordt door ONNX. ook biedt PyTorch de mogelijkheid om zelf ontworpen operaties mee te exporteren naar ONNX. Als er gewerkt wordt met de voorgetrainde modellen uit de TorchVision bibliotheek dan is het model exporteerbaar naar ONNX tenzij het model geoptimaliseerd is via quantisatie.

3.4 Frameworks voor mobiele implementatie

Er zijn een aantal frameworks die de programmeur de mogelijkheid geven om een model te optilmaliseren voor een mobiel platform. Deze frameworks hebben bovendien een API ter beschikking die het mogelijk maakt om het geoptimaliseerd model te implementeren in een mobiele omgeving. Er zal voornamelijk gefocust worden op Android implementaties. Niet elke framework voor mobiele implementatie optimaliseert het model op dezelfde manier, zo zullen bepaalde frameworks een betere compatibiliteit hebben met het framework waarin het model getraind is. Dus het converteren van het standaard model naar het mobiele model zal voor elk framework een ander resultaat geven. Sommige modellen zullen na het optilmaliseren in een bepaald framework een kleinere bestandsgrootte hebben dan bij andere frameworks. Deze frameworks zullen een model importeren van PyTorch, TensorFlow of ONNX. De mobiele implementatie van PyTorch, TensorFlow en onnxruntime zijn al in een eerdere paragraaf besproken.

3.4.1 CoreML

Core ML (Apple (2018)) is het Apple framework om machine learning tools te integreren in een applicatie. Dit kan een model zijn van Create ML het machine learning framework van Apple zelf. Maar Core ML biedt ondersteuning om modellen te converteren van TensorFlow, PyTorch en ONNX naar Core ML. CoreML optimaliseerd de prestaties op het toestel door efficiënt gebruik te maken van de CPU en GPU. Uiteraard is dit framework enkel van toepassing voor Apple, en in deze masterproef wordt er vooral gefocust op een Android implementatie.

Om een model te converteren naar CoreML zijn er 3 mogelijkheden. Eerst is rechtstreeks converteren vanuit PyTorch, daarvoor moeten we eerst ons model omzetten in een torchscript. Een tweede manier is door het model rechtstreeks te converteren vanuit TensorFlow. De derde manier is via ONNX maar dat raad Apple af omdat dit in nieuwe versies van CoreML niet meer ondersteund zal wordten. CoreML biedt ook maar ondersteuning tot en met opset versie 10 van ONNX. De volgende lijnen code tonen de implementatie voor de 3 manieren.

Listing 3.10: Converteren van ONNX en TensorFlow bestand naar een CoreML model

```
import coremltools as ct

# van Pytorch naar CoreML
model = ct.convert(
    traced_model,
    inputs=[ct.TensorType(shape=example_input.shape)]
)

# van TensorFlow naar CoreML
model = ct.convert('tf_model/saved_model.pb', source='tensorflow')

# van onnx naar CoreML
```

```
model = ct.converters.onnx.convert(model='my_model.onnx')
```

3.4.2 Mobile AI Compute Engine (MACE)

Het MACE framework (Khan (2020)) is ontworpen door Xaomi en dient specifiek voor mobiele toepassingen van neurale netwerken op Android, IOS, Linux en Windows. MACE biedt ondersteuning voor verschillende frameworks zoals: TensorFlow, Caffe en ONNX. MACE is ontworpen om accelerators van chips zo optimaal mogelijk te gebruiken voor AI taken. Bij het wisselen tussen verschillende systemen probeert MACE het verschil in uitvoering te minimaliseren. Het MACE framework bespaart geheugen door de core library zo klein mogelijk te maken door het aantal externe dependencies te minimaliseren. Een YML file wordt gebruikt om het model te beschrijven, op basis van deze informatie wordt een bibliotheek aangemaakt. Na het converteren naar een MACE model kan er een bibliotheek gemaakt worden die geïmplemnteerd kan worden in een applicatie via de MACE API.

3.4.3 Studies van het vergelijken van mobiele frameworks

In de paper geschreven door Luo et al. (2020) wordt PyTorch Mobile vergeleken met Tenserfolow Lite voor verschillende netwerk architecturen en een model getraind met dezelfde data. Voor alle netwerk architecturen in deze paper geeft de optimalisatie naar TensorFlow Lite de kleinste bestand grootte van het model. Uit deze paper is ook af te leiden dat de optimalisatie naar een mobiel model niet alleen afhankelijk is van het framework maar ook van de netwerk architectuur. Zo geeft TensorFlow Lite volgens Luo et al. (2020) betere latency resultaten voor de zwaardere netwerken (ResNet50, InceptionV3, DenseNet121) dan PyTorch Mobile. Maar PyTorch Mobile heeft op zijn beurt wel een betere latency voor SqueezeNet en MobileNetV2. Dus uit deze paper kunnen we afleiden dat TensorFlow Lite het beste de bestands grootte verkleint, maar dat de netwerk architectuur ook een rol speelt.

Febvay (2020) vergelijkt TensorFlow Lite met MACE voor verschillende neurale netwerken (SqueezeNet, MobileNetV1/V2). Hierbij geeft TensorFlow Lite het beste resultaat, TensorFlow Lite gaf een Top-1 resultaat van 69,19% en MACE gaf een top-1 resultaat van 66.84% bij MobileNetV1. Ook voor de latency gaf TensorFlow Lite in de meeste gevallen de beste resultaten buiten bij het gebruik van 4 of 6 CPU cores, dan gaf MACE betere resultaten.

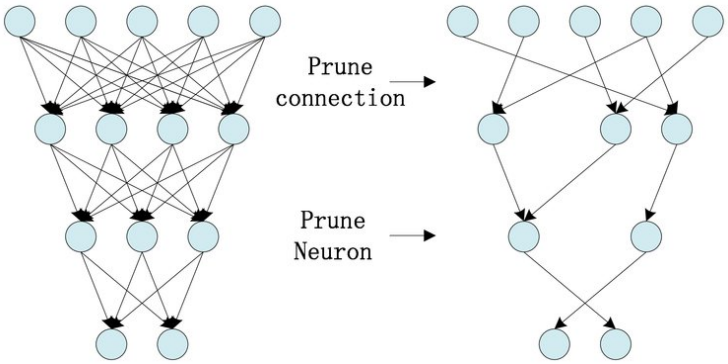
3.5 Optimalisaties van neurale netwerken voor snelheid en bestandgrootte

Er zijn in voorgaande paragrafen al technieken gevallen zoals prunning en quantisatie. In deze paragraaf zullen we hier dieper op in gaan. In deze paragraaf wordt er onderzocht welke optimalisaties er kunnen worden toegepast om de accuraatheid, snelheid en gebruikt geheugen te verbeteren. Maar het optimaliseren van een bepaalde factor zal vaak negatieve gevolgen hebben

voor een andere factor, dit zal meestal de accuraatheid zijn. Dus er zal een goede belans gevonden moeten worden tussen de optimalisatie en de negatieve gevolgen op de andere factoren.

3.5.1 Pruning

Pruning is de eerste stap van de Deep comression methode voorgesteld door Han et al. (2016). Bij het trainen van een CNN hebben bepaalde gewichten een grotere invloed op het resultaat. Andere gewichten hebben weinig tot geen invloed op het resultaat. Maar alle gewichten worden steeds berekend ongeacht hun invloed op het resultaat. Bij pruning worden de parameters met een kleine invloed op het resultaat verwijderd dit is weergegeven op figuur 3.3. Waardoor er geen berekeningen meer moeten uitgevoerd worden voor de verwijderde parameters. Deze parameters moeten dan ook niet meer worden bijgehouden waardoor het CNN model minder geheugen in beslag neemt. volgens Han et al. (2016) wordt voor VGG-16 het aantal parameters met factor 13 verminderd, voor AlexNet met een factor 9. Deze methode heeft zeer weinig tot geen effect op de accuraatheid.



Figuur 3.3: CNN voor en na pruning

Er zijn een aantal verschillende pruning technieken. TensorFlow maakt gebruik van weight pruning. Dit is het verwijderen van parameters in de gewicht tensor die niet nuttig zijn. Op deze manier wordt het aantal connecties tussen lagen verminderd, dit is te zien op figuur 3.3 als connection prune. Een tweede techniek te zien op op figuur 3.3 is neuron pruning. Hierbij wordt een voledige kolom in de gewichten matrix verwijderd, hierdoor verdwijnt.

3.5.2 Parameter quantisatie

Het quantiseren en delen van gewichten is een tweede methode voorgesteld door Han et al. (2016). Een CNN bestaat uit miljoenen gewichten, en de waarde van elke van deze gewichten moeten op het systeem worden opgeslagen. De default representatie van een waarde wordt opgeslagen als een floating point nummer wat 4 bytes in beslag neemt. Dus voor miljoenen parameters hebben de gewichten veel schijfruimte nodig. Een mogelijke oplossing hiervoor is quantiseren van gewichten. Hierbij wordt de getal representatie van de gewichten verandert naar een representatie die minder geheugen in beslag neemt.

3.5.3 Weight Clustering

Hierbij worden de waarden van gewichten beperkt tot een set van beschikbare waardes (Han et al. (2016)). Waarbij de waardes ëënmalig worden opgeslagen en al de gewichten refereren naar een waarde van de vaste set met waardes. Hoe kleiner de set met waardes is hoe minder geheugen er in beslag wordt genomen, maar een kleinere set van waardes zorgt ook voor een mindere accuratie. Dus de grootte van de set moet goed worden gekozen zodat er niet te veel geheugen wordt gebruikt met een accepteerbare daling in accuratie. Han et al. (2016) past vervolgens Huffman encoding toe die een compressie uitvoert op de gekwantiseerde parameters.

3.5.4 Convolutionele filter compressie en matrix factorisatie

Een andere methode voorgesteld is Compressed Convolutional Filters (Goel et al. (2020)). Hierbij wordt de grootte van de kernel verkleind om het aantal parameters en rekenwerk te verminderen. Maar door de kernels te verkleinen daalt de accuraatheid van het CNN. Bij matrix factorisatie (Goel et al. (2020)) worden grootte en complexe matrices opgesplitst in verschillende kleinere en simpelere matrices. Op deze manier worden ook redundante operaties uit het model gefilterd.

3.6 Converteren naar framework voor mobiele implementatie

Het doel van deze masterproef is om een bestaand netwerk op een mobiel platform te krijgen. Dus zullen de object detectoren die ontworpen zijn in TensorFlow en PyTorch geconverteerd moeten worden naar een framework voor mobiele implementatie. Voor het ontwerpen van object detectoren kan er gebruik gemaakt worden van bovenstaande bibliotheek (Detectron2, MMDetection, ImagAI en GluonCV). Maar dit kan het converteren meer complex maken. Omdat deze bibliotheken gebruik kunnen maken van niet operaties die niet compatibel zijn met het gewilde framework. Object detectoren zijn complexe systemen waarbij het converteren naar een ander framework complex of zelfs niet mogelijk zal worden. In deze paragraaf zal per objectdetector bibliotheek gekeken worden wat de mogelijkheden zijn om van een detector model naar een mobiele implementatie te gaan. De eerste stap zal zijn om te kijken welke mogelijkheden er zijn zonder te converteren naar een ander framework. Een tweede stap is via ONNX het huidige detectie model converteren naar een andere framework. Een derde stap is verder zoeken naar een alternatief als de eerste twee methodes niet lukken.

3.6.1 Van MMDetection naar detectie model voor mobile implementatie

Voor het testen van MMDetection nemen we de Kitty dataset Geiger et al. (2013) die bestaat uit auto's en voetgangers waarmee we een detector trainen via transfer learning. Voor de basis detector nemen we een voorgetrained Faster-RCNN detector.

Vermits MMDetection bovenop PyTorch werkt is de meest voor de hand liggende techniek om via PyTorch Mobile een model te genereren. Voordat het model kan geoptimaliseerd worden moet

het python afhankelijk model worden omgezet in TorchScript. Deze TorchScript module kan dan verder geoptimaliseerd worden voor mobiel gebruik. Het omzetten naar de scriptmodule geeft een TypeError fout waardoor het optimaliseren voor mobiel niet lukt. Deze fout is waarschijnlijk op te lossen door de 'example' input die nodig is voor de jit.trace functie uit te voeren aan te passen naar een gepaste input Tensor.

Voor gebruik te maken van andere frameworks voor mobiele implementatie zal het model eerst geconverteerd moeten worden naar ONNX formaat. MMDetection ondersteund de conversie naar ONNX, dit zit nog in zijn experimentele fase en MMDetection ondersteund momenteel enkel opset-versie 11 van ONNX. In de documentatie van MMDetection kan er een lijst gevonden worden met detectiemodellen die ondersteuning hebben voor het exporteren naar ONNX. MMDetection gebruikt een eigen script om een model te exporteren naar ONNX. Via de volgende lijn code is het mogelijk om het MMDetection model om te zetten naar een ONNX model.

Listing 3.11: Converteren van MMDetection naar een onnx bestand

```
!python ./tools/deployment/pytorch2onnx.py <config_file> <checkpiont_file>
        --output-file <output file>
```

pytorch2onnx.py is het MMDetection script om een model te converteren naar ONNX formaat. De config file is het bestand dat het neuraal netwerk beschrijft. En de checkpoint file is een file die tijdens het trainen wordt aangemaakt als checkpoint. Het finale model is normaalgezien terug te vinden als latest.pth, dit is het laatste checkpoint dat tijdens het trainen wordt aangemaakt. Op het einde van deze lijn code is het mogelijk om nog extra opties toe te voegen die in de MMDetection documentatie terug te vinden zijn. Dit script gebruikt geen nieuwe methode om naar ONNX te converteren, maar maakt eigenlijk gebruik van de ONNX export functie van pytorch. Bovenstaande lijn code converteert het Faster-RCNN model succesvol naar een ONNX model. Wel moet er bij vermeld worden dat dit MMDetection model een éénvoudig model is waarbij geen speciale aanpassingen zijn gedaan. Dus bij complexere modellen zou het resultaat anders kunnen zijn. Doordat we nu een ONNX model hebben zijn er een aantal nieuwe mogelijkheden om het model te implemnteren op een mobiel apparaat.

We kunnen het ONNX model kunnen we xuccesvol omzetten naar een onnxruntime model, maar bij het implementeren van onnxrunime model in android studio krijgen we de volgende error: 'java.lang.UnsatisfiedLinkError: No implementation found for long ai.onnxruntime.
.createOptions(long) tried Java_ai_onnxruntime_OrtSession_00024SessionOptions
_createOptions and Java_ai_onnxruntime_OrtSession_00024SessionOptions_createOptions_J'
Deze error onststaat wanneer de applicatie een bibliotheek probeerts in te laden, maar deze bibliotheek bestaat niet. Zowel de standaard bibliotheek en de onxxruntime-mobile bibliotheek geven deze fout. Het ONNX model implementeren in javascript werd daarintegen wel uitgevoerd zonder fouten.

Met het gegenereerde ONNX model is het mogelijk om het model om te zetten naar een TensorFlow model en vervolgens naar een TensorFlow Lite model om te zetten. Hierbij komen al 3 conversies aan te pas en met elke convertie dus is de kans groter dat het uiteindelijke model een error geeft. Het converteren van een .onnx bestand naar een TensorFlow Lite model kan met behulp van de

volgende lijnen code.

Listing 3.12: Converteren van ONNX bestand naar een TensorFlow Lite model

```
import tensorflow as tf
import onnx
from onnx_tf.backend import prepare

#ONNX model inladen
onnx_model = onnx.load("model.onnx")  # inladen onnx model
output = prepare(onnx_model)
output.export_graph('tf_model.pb') # model exporteren naar TensorFlow model

#Ingeladen model omzetten naar TensorFlow Lite model
converter = tf.lite.TFLiteConverter.from_saved_model('tf_model.pb')
converter.target_spec.supported_ops = [
        tf.lite.OpsSet.TFLITE_BUILTINS, # enable TensorFlow Lite ops.
        tf.lite.OpsSet.SELECT_TF_OPS # enable TensorFlow ops.
]
tflite_model = converter.convert() # converteer model
```

Eerst moet het ONNX model ingeladen worden als een standaard TensorFlow model. Vervolgens moest bij het converteren naar een TensorFlow Lite model eerst vermeld worden welke TensorFlow operaties er ondersteund moeten worden. Het effectief converteren naar een TensorFlow Lite model duurde ongeveer 30 minuten voor dit testmodel. bij het uitvoeren van dit model krijgen we de volgende error:
'java.lang.AssertionError: Error occurred when initializing ObjectDetector: Didn't find op for builtin opcode 'MUL' version '5'. An older version of this builtin might be supported. are you using an old TFLite binary with a newer model.'
Deze fout geeft aan dat de 'MUL' operatie niet ondersteund wordt door TensorFlow.

Bij het converteren naar CoreML vanuit PyTorch stoten we op hetzelfde probleem als bij het converteren naar PyTorch Mobile. Er wordt namelijk ook op dezelfde TypeError gestoten bij het omzetten naar TorchScript model. De tweede manier om naar CoreML te gaan is vanuit TensorFlow, maar dit is een vrij omslachtige manier omdat we dan de volgende converties moeten maken MMDetection $\rightarrow$ONNX $\rightarrow$TensorFlow $\rightarrow$CoreML. Maar om van TensorFlow naar CoreML te gaan verwacht de converter een Keras model, maar dit is er niet vermits het model in MMDetection is ontworpen.

Om van een MMDetection model naar een mobiele implementatie te gaan is een complex process dat op een heel aantal problemen stoot. Geen enkel model werd na het converteren succesvol uitgevoerd. De enige methode waarbij het model werdt uitgevoerd zonder fouten is de onnxruntime implementatie in javascript.

Bibliografie

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). TensorFlow: A system for large-scale machine learning.

Android (2021). Neural Networks API | Android NDK. `https://developer.android.com/ndk/guides/neuralnetworks?hl=nl`.

Apple (2018). Core ML | Apple Developer Documentation. `https://developer.apple.com/documentation/coreml`.

Chen, K., Wang, J., Pang, J., Cao, Y., Xiong, Y., Li, X., Sun, S., Feng, W., Liu, Z., Xu, J., Zhang, Z., Cheng, D., Zhu, C., Cheng, T., Zhao, Q., Li, B., Lu, X., Zhu, R., Wu, Y., Dai, J., Wang, J., Shi, J., Ouyang, W., Loy, C. C., and Lin, D. (2019). MMDetection: Open MMLab Detection Toolbox and Benchmark.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems.

Duan, K., Bai, S., Xie, L., Qi, H., Huang, Q., and Tian, Q. (2019). CenterNet: Keypoint Triplets for Object Detection.

Febvay, M. (2020). Low-level Optimizations for Faster Mobile Deep Learning Inference Frameworks. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 4738–4742, Seattle WA USA. ACM.

Gai, W., Liu, Y., Zhang, J., and Jing, G. (2021). An improved tiny yolov3 for real-time object detection. *Systems Science & Control Engineering*, 9(1):314–321.

Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*.

Girshick, R. (2015). Fast R-CNN.

Goel, A., Tung, C., Lu, Y.-H., and Thiruvathukal, G. K. (2020). A survey of methods for low-power deep learning and computer vision.

Google (2014). FlatBuffers: Memory efficient Serialization Library. `https://google.github.io/flatbuffers/`.

Google (2021). Protocol Buffers - Google's data interchange format. `https://github.com/protocolbuffers/protobuf`.

Guo, J., He, H., He, T., Lausen, L., Li, M., Lin, H., Shi, X., Wang, C., Xie, J., Zha, S., Zhang, A., Zhang, H., Zhang, Z., Zhang, Z., Zheng, S., and Zhu, Y. (2020). GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing.

Han, S., Mao, H., and Dally, W. J. (2016). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.

Jiang, X., Hadid, A., Pang, Y., Granger, E., and Feng, X. (2019). *Deep Learning in Object Detection and Recognition, edited by Xiaoyue Jiang, Abdenour Hadid, Yanwei Pang, Eric Granger, Xiaoyi Feng.* Springer Singapore : Imprint: Springer, Singapore, 1st ed. 2019. edition.

Keras, T. (2021). Keras documentation: About Keras. `https://keras.io/about/`.

Khan, J. (2020). MACE: Deep learning optimized for mobile and edge devices. `https://heartbeat.comet.ml/mace-deep-learning-optimized-for-mobile-and-edge-devices-5e6941cc0533`.

Koehrsen, W. (2018). Neural Network Embeddings Explained.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. (2020). PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and DollÃ¡r, P. (2018). Focal Loss for Dense Object Detection.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. (2016). SSD: Single Shot MultiBox Detector. volume 9905, pages 21–37.

Luo, C., He, X., Zhan, J., Wang, L., Gao, W., and Dai, J. (2020). Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices.

Martins, M., Mota, D., Morgado, F., Wanzeller, C., Martins, P., and Abbasi, M. (2021). ImageAI: Comparison Study on Different Custom Image Recognition Algorithms. In Rocha, I., Adeli, H., Dzemyda, G., Moreira, F., and Ramalho Correia, A. M., editors, *Trends and Applications in Information Systems and Technologies*, Advances in Intelligent Systems and Computing, pages 602–610, Cham. Springer International Publishing.

Noble, W. S. (2006). What is a support vector machine? *Nat Biotechnol*, 24(12):1565–1567. Bandiera_abtest: a Cg_type: Nature Research Journals Number: 12 Primary_atype: Reviews Publisher: Nature Publishing Group.

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning.

ONNX (2017). ONNX Tutorials. https://github.com/onnx/tutorials.

ONNX (2019). ONNX Runtime (ORT). https://onnxruntime.ai/docs/.

ONNX (2021). tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX. https://github.com/onnx/tensorflow-onnx/blob/42e800dc2945e5cadb9df4f09670f2e20eb6d222/support_status.md.

Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2017). PyTorch. https://www.PyTorch.org.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788. ISSN: 1063-6919.

Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement.

Ren, S., He, K., Girshick, R., and Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2019). MobileNetV2: Inverted Residuals and Linear Bottlenecks.

Tan, M., Pang, R., and Le, Q. V. (2020). EfficientDet: Scalable and Efficient Object Detection.

Uijlings, J. R., R, van de Sande, K. E., A, Gevers, T., Smeulders, A. W., and M (2013). Selective Search for Object Recognition. *International Journal of Computer Vision*, 104(2):154–171. Num Pages: 154-171 Place: New York, Netherlands Publisher: Springer Nature B.V.

Wu, Y., Kirillov, A., Massa, F., Lo, W.-Y., and Girshick, R. (2019). Detectron2. https://github.com/facebookresearch/detectron2.

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS DE NAYER SINT-KATELIJNE-WAVER
J. De Nayerlaan 5
2860 SINT-KATELIJNE-WAVER, België
tel. + 32 15 31 69 44
iiw.denayer@kuleuven.be
www.iiw.kuleuven.be