

# 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

©Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor(en) als de auteur(s) is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, kan u zich richten tot KU Leuven Technologiecampus De Nayer, Jan De Nayerlaan 5, B-2860 Sint-Katelijne-Waver, +32 15 31 69 44 of via e-mail [iiw.denayer@kuleuven.be](mailto:iiw.denayer@kuleuven.be).

Voorafgaande schriftelijke toestemming van de promotor(en) is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

# Voorwoord

Het voorwoord vul je persoonlijk in met een appreciatie of dankbetuiging aan de mensen die je hebben bijgestaan tijdens het verwezenlijken van je masterproef en je hebben gesteund tijdens je studie.

# Samenvatting

De (korte) samenvatting, toegankelijk voor een breed publiek, wordt in het Nederlands geschreven en bevat **maximum 3500 tekens**. Deze samenvatting moet ook verplicht opgeladen worden in KU Lokaal.

# Abstract

Het extended abstract of de wetenschappelijke samenvatting wordt in het Engels geschreven en bevat **500 tot 1.500 woorden**. Dit abstract moet **niet** in KU Loket opgeladen worden (vanwege de beperkte beschikbare ruimte daar).

**Keywords:** Voeg een vijftal keywords in (bv: Latex-template, thesis, ...)

# Inhoudsopgave

<b>Voorwoord</b>	<b>iii</b>
<b>Samenvatting</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>Inhoud</b>	<b>viii</b>
<b>Figurenlijst</b>	<b>ix</b>
<b>Tabellenlijst</b>	<b>x</b>
<b>Symbolenlijst</b>	<b>xi</b>
<b>Lijst met afkortingen</b>	<b>xii</b>
<b>1 Situering en doelstelling</b>	<b>1</b>
1.1 Situering . . . . .	1
1.2 Probleemstelling . . . . .	2
1.3 Doelstellingen . . . . .	2
<b>2 Herkenning en Detectie Algemeen</b>	<b>3</b>
2.1 Deep learning-gebaseerde herkenningssystemen . . . . .	3
2.1.1 Herkenning . . . . .	3
2.1.2 Convolutioneel neurale netwerk (CNN) . . . . .	4
2.1.3 Trainen van een CNN . . . . .	6
2.1.4 Transfer Learning . . . . .	7
2.1.5 ResNet50 . . . . .	7
2.2 Deep learning-gebaseerde detector . . . . .	7
2.2.1 Two-stage detector . . . . .	8
2.2.2 One-stage detector . . . . .	8

<b>3</b>	<b>Implementatie van Herkenning en detectie op mobiel platform</b>	<b>11</b>
3.1	Frameworks . . . . .	11
3.1.1	TensorFlow . . . . .	12
3.1.2	PyTorch . . . . .	12
3.2	Objectdetectie bibliotheken . . . . .	12
3.2.1	MMDetection . . . . .	13
3.2.2	Detectron2 . . . . .	13
3.2.3	GluonCV . . . . .	13
3.2.4	ImageAI . . . . .	14
3.3	Open Neural Network Exange (ONNX) . . . . .	14
3.3.1	TensorFlow naar ONNX conversie . . . . .	15
3.3.2	PyTorch naar ONNX conversie . . . . .	16
3.4	Frameworks voor mobiele implementatie . . . . .	16
3.4.1	TensorFlow Lite (TFLite) . . . . .	17
3.4.2	TensorFlow.js . . . . .	19
3.4.3	PyTorch Mobile . . . . .	19
3.4.4	Onnxruntime . . . . .	20
3.4.5	ONNX.js . . . . .	21
3.4.6	CoreML . . . . .	22
3.4.7	Mobile AI Compute Engine (MACE) . . . . .	22
3.4.8	Gerelateerd werk . . . . .	22
3.5	Optimalisaties van neurale netwerken voor snelheid en bestandgrootte . . . . .	23
3.5.1	Pruning . . . . .	23
3.5.2	Parameter kwantisatie . . . . .	24
3.5.3	Weight Clustering . . . . .	25
3.5.4	Convolutionele filter compressie en matrix factorisatie . . . . .	25
<b>4</b>	<b>Compatibiliteit van herkenningssystemen</b>	<b>26</b>
4.1	ResNet50 . . . . .	26
4.2	Van TensorFlow naar mobiel framework . . . . .	26
4.2.1	Van PyTorch naar mobiele implementatie . . . . .	27
<b>5</b>	<b>Compatibiliteit van detectie systemen</b>	<b>29</b>
5.1	faster rcnn . . . . .	29
5.2	Van TensorFlow naar mobiel mobiele implementatie . . . . .	29
5.2.1	TFLite implementatie . . . . .	29

---

5.2.2	ONNX implementatie . . . . .	30
5.3	Van PyTorch naar mobiele implementatie . . . . .	30
5.3.1	TorchScript implementatie . . . . .	32
5.3.2	ONNX implementatie . . . . .	32
<b>A</b>	<b>Uitleg over de appendices</b>	<b>36</b>



# Lijst van figuren

2.1	Voorbeeld van embedding space voor boek genres. . . . .	4
2.2	CNN met twee convolutie lagen en twee pooling lagen en één fully-connected laag .	4
2.3	Convolutielaag waarbij een filter wordt herleid tot een output feature. . . . .	5
2.4	ReLu, waarbij het maximum wordt genomen van 0 en de input waarde. . . . .	5
2.5	Max pooling waarbij er verder wordt gegaan met de maximum waarde in een 2x2 regio. . . . .	6
2.6	Faster R-CNN . . . . .	8
2.7	YOLO waarbij de input is opgedeeld in een S x S rooster. En waarbij bounding box voorspellingen zijn gedaan. . . . .	9
2.8	One-stage detector met VGG als backbone, elke feature map met een verschillende schaal wordt geëvalueerd. . . . .	10
3.1	Een weergave van de voornaamste frameworks die naar ONNX kunnen exporten en die een ONNX model kunnen importeren. . . . .	15
3.2	Implementatieflow van een TensorFlow model naar een mobiele implementatie. 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 geïmplementeerd worden op een client toestel waar er gebruik gemaakt kan worden van verschillende hardware componenten . . . . .	18
3.3	CNN voor en na pruning . . . . .	24
3.4	kwantisatie van twee floating point variabelen die worden omgezet naar twee fixed point variabelen met een gemeenschappelijke exponent. . . . .	24
3.5	Weight clustering: links zijn allemaal verschillende gewichten en na het uitvoeren van weight clustering zijn er in de matrix pointers terug te vinden die wijzen kunnen wijzen naar vier verschillende waarden. . . . .	25

# Lijst van tabellen

4.1	Alle operaties die terug te vinden zijn in het ResNet50 model en hun compatibiliteit met andere frameworks . . . . .	28
5.1	Alle operaties die terug te vinden zijn in het ResNet50 model en hun compatibiliteit met andere frameworks . . . . .	31

# Lijst van symbolen

Maak een lijst van de gebruikte symbolen. Geef het symbool, naam en eenheid. Gebruik steeds SI-eenheden en gebruik de symbolen en namen zoals deze voorkomen in de hedendaagse literatuur en normen. De symbolen worden alfabetisch gerangschikt in opeenvolgende lijsten: kleine letters, hoofdletters, Griekse kleine letters, Griekse hoofdletters. Onderstaande tabel geeft het format dat kan ingevuld en uitgebreid worden. Wanneer het symbool een eerste maal in de tekst of in een formule wordt gebruikt, moet het symbool verklaard worden. Verwijder deze tekst wanneer je je thesis maakt.

$b$	Breedte	$[mm]$
$A$	Oppervlakte van de dwarsdoorsnede	$[mm^2]$
$c$	Lichtsnelheid	$[m/s]$

# Lijst van afkortingen

AI Artificiële intelligentie  
API Application programming 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 Exchange  
Opset Operation Set  
R-CNN Region Based Convolutional Network  
ReLU Rectified Linear Unit  
RoI Region of Interest  
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 beeldverwerkingsproblemen op te lossen. Via neurale netwerken kunnen we met meer en betere features werken om de afbeeldingen te analyseren. Veel van deze modellen hebben echter behoorlijk wat rekenkracht en geheugen nodig om te werken. Bovendien is er steeds meer interesse naar real-time toepassingen waarvan het resultaat zo snel mogelijk beschikbaar moet zijn. Een voorbeeld hiervan zijn zelfrijdende auto's. Een zelfrijdende auto moet kunnen stoppen voor een persoon die de weg oversteeft. Hierbij moet zo snel en accuraat mogelijk een persoon herkend worden zodat de auto op tijd kan stoppen. Dit wordt moeilijk bij een aantal toepassingen waarbij de foto eerst genomen moet worden en vervolgens door een computer geanalyseerd moet worden. Dit is omdat huidige herkenningssystemen en detectiesystemen veel rekenwerk en geheugen vragen, het volgende voorbeeld is een mogelijke use-case.

Het automatisch detecteren en herkennen van producten in de rekken van een supermarkt heeft veel interessante real-time toepassingen. Het kan slechtzienden helpen om snel de producten te vinden die ze nodig hebben. Het winkelmanagement kan zo een real-time status krijgen van de inventaris op de winkelrekken. Er kan ook nagekeken worden of de producten staan waar ze horen te staan in de rekken. Eerdere studies hebben reeds het potentieel van diep neurale netwerken laten zien voor deze taken. Achter het neurale netwerk van deze toepassing zit echter veel complex rekenwerk en vereisten voor het geheugen. Deze beperkingen zorgen voor een groot struikelblok voor real-life toepassingen. Het zou namelijk handig zijn dat het neurale netwerk kan worden uitgevoerd op een smartphone. De voornaamste manier om zware neurale netwerken uit te voeren op een smartphone is door het neurale netwerk uit te voeren op een server zoals de cloud. Een internet connectie is nodig om data naar de cloud te sturen en van de cloud te ontvangen. Door het neurale netwerk op een smartphone uit te voeren is er geen nood meer aan een internet connectie. Dit zorgt ook voor een lagere response tijd omdat de applicatie niet meer moet wachten op het antwoord van het neurale netwerk dat in de cloud wordt uitgevoerd. De data blijft dan op de smartphone en wordt niet over het internet gecommuniceerd wat zorgt voor een betere privacy en

veiligheid. In deze masterproef onderzoeken we hoe een bestaand neurale 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. We gaan ook onderzoeken hoe alle data efficiënt kan worden opgeslagen op het toestel.

Bij deze masterproef willen we van een neurale netwerk dat in een bepaald framework gemodelleerd en getraind is gaan naar een mobiele implementatie. Zo wordt er voor een aantal verschillende frameworks onderzocht op welke manieren deze naar een mobiele implementatie kunnen gaan. Bovendien zullen we per framework de omzetting naar mobiele implementatie voor verschillende CNN-architecturen onderzoeken. Vervolgens zal er ook gekeken worden naar verdere optimalisaties voor herkenningssystemen en detectiesystemen. Het uiteindelijke doel is een prototype ontwikkelen dat een bestaand herkenningnetwerk en detectienetwerk implementeert op een mobiel apparaat. Bij dit prototype is dan het rekenwerk en de geheugen vereisten 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 neurale netwerk en het model omzetten naar een mobiele implementatie. We gaan ervan uit dat in deze masterproef het model van het neurale netwerk reeds gemodelleerd en getraind is.

## 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:

- Het bespreken van een deep learning herkenningssysteem en detectiesysteem.
- Bespreken van frameworks/bibliotheken waarin het basismodel ontwikkeld kan worden.
- Bespreken van frameworks en optimalisatietechnieken die ervoor kunnen zorgen dat bestaande modellen mogelijk kunnen uitgevoerd worden op een mobiel apparaat.
- gevonden technieken testen en analyseren voor verschillende frameworks en neurale netwerken met als uiteindelijk resultaat een werkend prototype te ontwikkelen.

## Hoofdstuk 2

# Herkenning en Detectie Algemeen

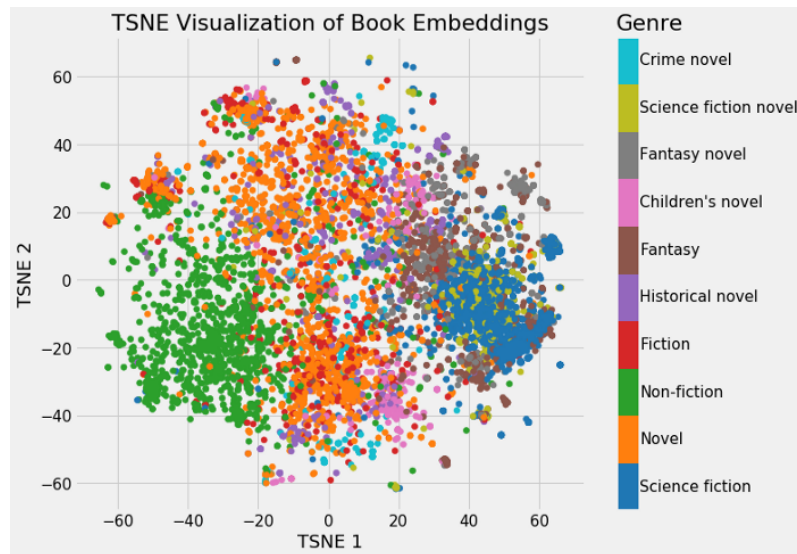
### 2.1 Deep learning-gebaseerde herkenningssystemen

Herkenningssystemen voorspellen wat de identiteit is van een afbeelding. Dit is het herkennen van objecten in digitale afbeeldingen zonder deze te lokaliseren of aan te duiden. Bij herkenningssystemen is er geen of weinig overlap tussen de trainingsafbeeldingen en de inputafbeeldingen. Bijvoorbeeld bij gezichtsherkenning wordt er een algemeen herkenningssysteem ontworpen dat gezichten herkent, en niet een systeem dat elk individueel gezicht herkent. Voor een herkenningssysteem is er een goed getraind neuraal netwerk nodig dat een input afbeeldingen omzet in features. Er moet een database zijn met daarin de gegevens van de objecten die men wil 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 neuraal netwerk is kan er een herkenningssysteem ontwikkeld worden. Als men een bepaald object in een afbeelding wil herkennen gaat men met behulp van een neuraal netwerk de afbeelding omzetten in een embedding. Volgens Koehrsen (2018) zijn embeddings vectoren die kunnen worden vergeleken in een embedding space, waar gelijkaardige objecten dicht bij elkaar liggen. De embedding van de input afbeelding wordt vergeleken met de embeddings die zich in een galerij bevinden. Jiang et al. (2019) vermeldt dat met behulp van een query er gelijkaardige objecten uit de galerij gehaald kunnen worden om deze vervolgens te gaan vergelijken in een embedding space. De galerij is een database/verzameling met gekende embeddings/ID's van de objecten die men wil 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. De nearest neighbour techniek vermeld in Ma et al. (2017) gaat in de embedding space kijken naar het K-aantal dichtste burens van een query. Het label dat het meest voorkomt tussen het K-aantal burens, zal dan ook het meest waarschijnlijke label zijn voor de query. Figuur 2.1 is een voorbeeld van een embedding space, een query is een punt in deze grafiek dat nog geen label heeft. Voor

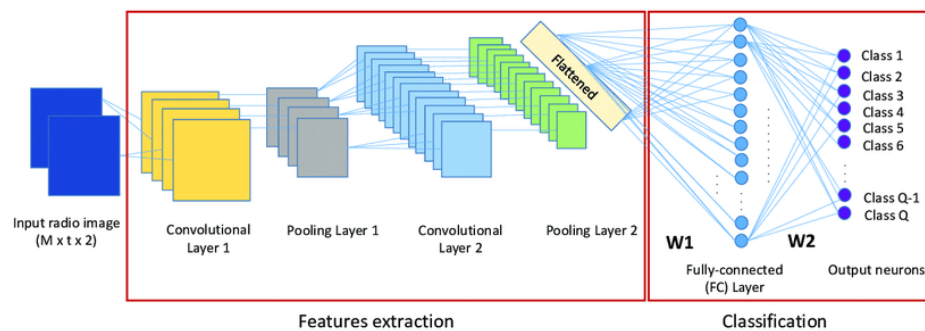
het herkennen van een afbeelding gaan we in de grafiek naar de K dichtsbijzjnde bureen kijken van de query.



**Figuur 2.1:** Voorbeeld van embedding space voor boek genres.  
Koehrsen (2018)

### 2.1.2 Convolutioneel neuraal netwerk (CNN)

De belangrijkste bouwsteen van een herkenningssysteem is een getraind CNN. In deze paragraaf bespreken we het CNN en zijn verschillende bouwstenen beschreven door Jiang et al. (2019). Figuur 2.2 is een voorbeeld van een CNN en zijn verschillende onderdelen.

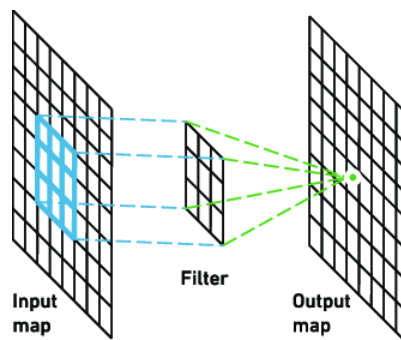


**Figuur 2.2:** CNN met twee convolutie lagen en twee pooling lagen en één fully-connected laag

Het belangrijkste deel van een CNN zijn de convolutielagen (figuur 2.3). Bij een convolutielaag wordt een kernel/filter over de input geschoven om features te bepalen. Een kernel bestaat uit een set van gewichten die met de input worden vermenigvuldigd. Deze actie wordt de convolutie genoemd. Al de pixels binnen het veld van de kernel worden gereduceerd tot een enkele waarde. De convoluties zijn zeer efficiënt om visuele informatie uit de input te halen. Convolutielagen leren

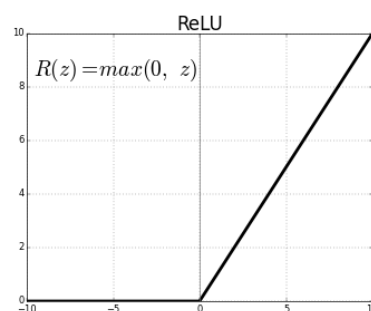


verschillende features door meerdere kernels in parallel uit te voeren per convolutielaag. Dit zorgt ervoor dat de matrices met featuremappen per kernel steeds kleiner worden maar ook dieper worden. Een andere factor van een convolutie laag is de stride. Deze waarde geeft aan met hoeveel pixels de kernel telkens moet doorschuiven. Als de stride één is dan schuift de kernel steeds op met één pixel en als de stride drie is dan schuift de kernel op met drie pixels. Stride één zorgt voor meer features per featuremap, maar maakt het CNN trager omdat er meer bewerkingen moeten worden uitgevoerd. Een CNN bestaat uit een opeenvolging van een aantal convolutielagen 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 2.3:** Convolutielaag waarbij een filter wordt herleid tot een output feature.

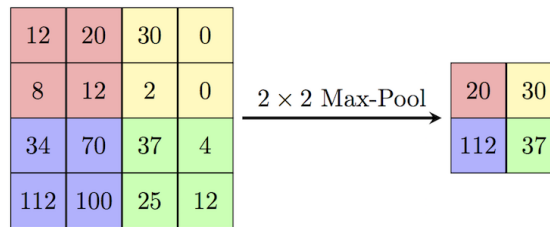
Na elke convolutielaag komt een niet lineaire activatie functie. De niet lineaire activatie functie zorgt ervoor dat het CNN niet herleid kan worden tot één convolutielaag die geen high-level features kan extraheren. De meest gebruikte functie hiervoor is de rectified linear unit (ReLU) (figuur 2.4). De ReLU wordt vaak gebruikt omdat deze veel sneller wordt uitgevoerd dan andere activatie functies zoals Sigmoid en Tangens hyperbolicus (Krizhevsky et al. (2017)). De ReLU kan exact 0 weergeven en ziet er lineair uit.  $\text{Max}(0, x)$  is de ReLU bewerking, dus er wordt verdergegaan met 0 of de input waarde.



**Figuur 2.4:** ReLU, waarbij het maximum wordt genomen van 0 en de input waarde.

Een volgende bouwsteen is de poolinglaag. Deze laag verminderd het aantal features per feature map. De meest voorkomende methode is max-pooling weergegeven in figuur 2.5 waarbij er verder wordt gegaan met de maximum waarde in een bepaalde regio. Het doel van een poolinglaag is om het aantal parameters te verminderen en zo ook het rekenwerk te verminderen. Er kan ook gebruik

gemaakt worden van average 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.



**Figuur 2.5:** Max pooling waarbij er verder wordt gegaan met de maximum waarde in een 2x2 regio.

Op het einde van elk CNN volgen er meestal één of meerdere fully connected lagen. Deze lagen connecteren elke input van één laag met elke activatie eenheid van de volgende laag, dit is weergegeven in het classificatie gedeelte van figuur 2.2. Dit zorgt voor meer parameters en meer rekenwerk, maar ook voor 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 afbeeldingen met labels aan het netwerk. Op basis van het resultaat van deze voorbeelden worden de gewichten van de kernels telkens aangepast. Zo levert het CNN steeds een beter resultaat.

Tijdens het trainen van een CNN nemen we een groep met trainingsafbeeldingen en geven we deze als input aan het CNN. Het CNN geeft een voorspelling van deze trainingsafbeelding. Vervolgens vergelijken we de voorspelling met de label van de trainingsafbeelding. Per groep trainingsafbeeldingen wordt er via een loss functie het verschil tussen de voorspelde waarde en de werkelijke waarde berekend. De loss functie geeft de error van de voorspelling weer tijdens het trainen van een neurale netwerk. Als al de groepen met trainingsafbeeldingen één keer zijn gebruikt als trainingsinput dan is er één epoch voltooid. Zo kan het CNN X-aantal epochs uitvoeren waarbij de trainingsafbeelding X-aantal keer door het CNN worden verwerkt.

De stochastic gradient descent is een techniek die men gebruikt om de loss van het CNN te minimaliseren. Hierbij wordt op basis van de loss functie de gradiënt voor elk gewicht berekend door de afgeleide te nemen van de loss naar dit gewicht.

$$gradient = d(loss)/d(gewicht) \quad (2.1)$$

Via de berekende gradiënten kunnen we nu de gewichten aanpassen zodat de loss geminimaliseerd wordt. We kunnen de factor waarmee we de gewichten aanpassen beïnvloeden met de learning rate. De learning rate is een factor die aangeeft hoe groot de stap moet zijn waarmee de gewichten worden aangepast.

$$gewicht = oud\_gewicht - (learningrate * gradient) \quad (2.2)$$

Hoe kleiner de learning rate hoe langer het trainen van een CNN duurt. Een te grote learning rate kan echter resulteren in een slecht getraind netwerk, omdat de veranderingen op de gewichten dan te groot zijn om een beter resultaat te krijgen.

### 2.1.4 Transfer Learning

Bij transfer learning (Geiger et al. (2013)) wordt er verder gebouwd op een model dat reeds getraind is. Hierbij maakt men gebruik van een basismodel waarvan de toepassing gerelateerd is aan de gewenste toepassing. Bijvoorbeeld een model waarmee we dieren kunnen herkennen gebruiken we als basis voor een model dat hondenrassen herkent. Op dit basismodel kan er verdergetraind worden met een dataset specifiek voor de gewenste toepassing. Deze dataset kan veel kleiner zijn dan een dataset die nodig is om een nieuw model te trainen. Het trainen van een nieuw CNN kan soms weken duren. Via transfer learning kan de trainingsperiode met een grote factor gereduceerd worden. Deze methode gebruiken we voornamelijk om een 'nieuw' model te trainen vanwege de kleinere trainingsdataset en kortere trainingstijd.

### 2.1.5 ResNet50

Het herkenningnetwerk dat we willen implementeren in deze masterproef is de ResNet50 architectuur. He et al. (2015) heeft vastgesteld dat als het aantal lagen van een CNN toeneemt dat op een bepaald moment de training accuraatheid daalt. Dit verschijnsel noemt men de vanishing gradient. In paragraaf 2.1.3 hebben we besproken hoe we de gradient kunnen berekenen tijdens het trainen van een CNN. Voor elke laag in het CNN moet de gradient opnieuw berekend worden door telkens opnieuw de afgeleide te berekenen. Hierdoor wordt de gradient steeds kleiner en kleiner tot deze een minimum bereikt. Waardoor de gewichten in de eerste lagen heel traag aanpassen of zelfs niet meer veranderen. He et al. (2015) die dit probleem hebben vastgesteld hebben dit opgelost door gebruik te maken van skip connections. Hierbij wordt de input van een laag rechtstreeks met een volgende laag geconnecteerd die x aantal lagen verder ligt. Op deze manier worden de gradienten per laag niet meer kleiner. ResNet50 bestaat uit 50 convolutie lagen waarbij er een skip connection plaatsvindt per 3 lagen. De resnet50 architectuur is opgebouwd uit ResNet blokken die bestaan uit 3 convolutie lagen en 1 skip connection.

## 2.2 Deep learning-gebaseerde detector

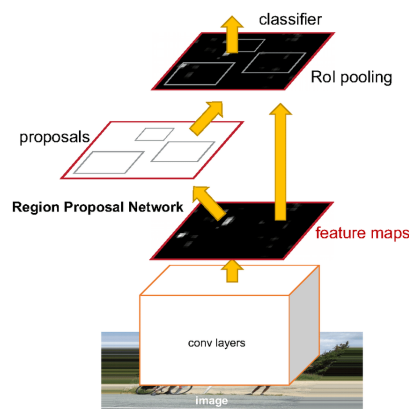
Objectdetectie is het lokaliseren en classificeren van objecten in een afbeelding, waarbij de objecten aangeduid worden met een Bounding box. Een bounding box is een kader die rond een object wordt getekend. De klassieke versie van objectdetectie is de sliding window benadering. Waarbij een venster met vaste grootte over de afbeelding schuift en telkens de gegevens binnen het venster analyseert. Momenteel kan objectdetectie 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 Interest (Rols) 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 Rols die in het eerste deel gecreëerd werden.

De voornaamste two-stage detector is de Faster R-CNN detector Ren et al. (2016). Hierbij verwerkt een CNN eerst de inputafbeeldingen en vervolgens wordt er gebruik gemaakt van een Region Proposal Network (RPN) weergegeven in figuur 2.6. Het RPN is een volledig convolutioneel netwerk dat regio's uit de afbeelding filtert waar de kans groot is dat er objecten ontstaan. Per input geeft het RPN een set van regio's als output. Elk van deze regio's heeft een objectness score wat aangeeft in welke mate de regio een object bevat. Om een region proposal te genereren wordt het RPN over de feature map geschoven dat gegenereerd is door het voorgaande CNN. Op elke sliding window locatie van het RPN worden er meerdere regio voorspellingen gedaan. Deze voorspelling doet het RPN door verschillende anchor boxes te evalueren per sliding window locatie. Anchor boxes zijn een voorgedefiniëerde set van bounding boxes met drie verschillende vormen in drie verschillende schalen. Via de non-maxima suppression methode zorgen we ervoor dat er maar één anchor box overblijft van de overlappende anchor boxes. Deze techniek houdt enkel de anchor box over met de beste voorspelling en onderdrukt de rest van de anchor boxes.

Na het RPN komt de RoI Pooling laag. Deze laag gebruikt max pooling om de feature map van elke RoI om te zetten naar een feature map met vaste grootte. Elk van deze features gaat door een set van fully connected lagen die twee lagen als output heeft. Een softmax laag die de klasse voorspelt, en een bounding box regressie laag die de bounding box voorspelt.

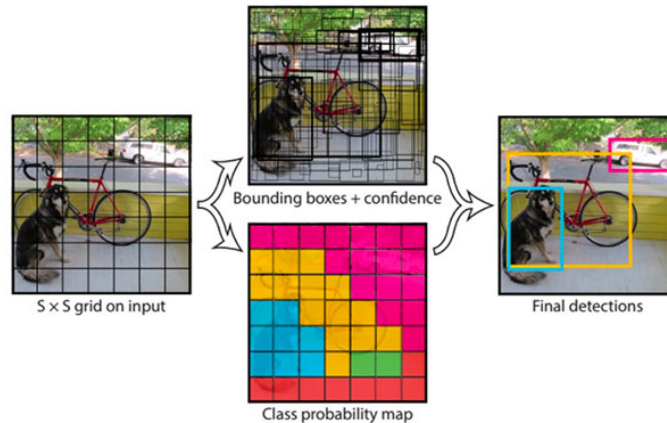


Figuur 2.6: Faster R-CNN

### 2.2.2 One-stage detector

Bij one-stage detectoren gebeurt objectdetectie in één keer. Dus er is geen region proposal niveau meer zoals bij de two-stage detector. Deze detectoren gebruiken minder geheugen en rekenkracht

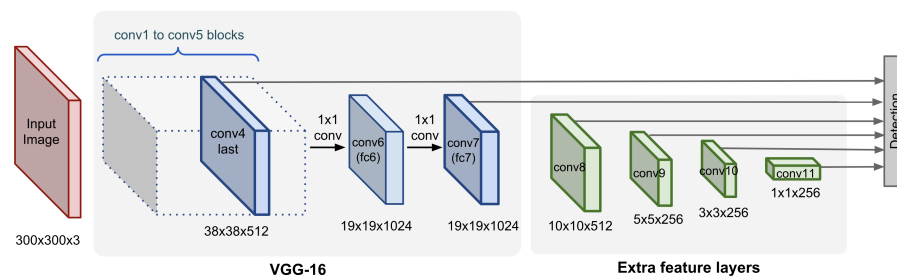
dan two-stage detectoren. Maar deze detectoren kunnen wat in nauwkeurigheid verliezen t.o.v. two-stage detectoren. Deze detectoren zijn zeer geschikt om gebruikt te worden op mobiele apparaten, omdat deze detectoren sneller zijn en minder geheugen nodig hebben. Twee veel gebruikte technieken van one-stage detectie zijn: You Only Look Once (YOLO) en Single Shot Detection (SSD).



**Figuur 2.7:** YOLO waarbij de input is opgedeeld in een  $S \times S$  rooster. En waarbij bounding box voorspellingen zijn gedaan.

YOLO Redmon et al. (2016) verdeelt de afbeelding in een  $S \times S$  rooster zoals in figuur 2.7 te zien is. Elke cel kan meerdere bounding boxes voorspellen. Per bounding box wordt er een score bepaald 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 boxes te verwijderen. Een eerste mogelijkheid is door enkel bounding boxes te bepalen waarvan de voorspelling boven een zekere threshold ligt. Een andere methode is non-maxima suppression, een methode die ervoor zorgt 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 boxes.

SSD Liu et al. (2016) is een one-stage detector waarbij een afbeelding door verschillende convolutielagen gaat. Dit resulteert in feature mappen met verschillende schalen. Op elke locatie van deze feature mappen wordt een vaste set van bounding boxes geëvalueerd. Voor elk van deze boxes wordt de zekerheid dat het een object bevat voorspeld. Op het einde wordt non maximum suppression gebruikt om de finale voorspelling te maken. In figuur 2.8 is te zien dat een SSD bestaat uit drie delen. De eerste twee delen bestaan uit een standaard classificatie netwerk zonder de fully-connected lagen en een set van extra convolutielagen. Het derde deel doet de effectieve detectie voor elke feature map met een verschillende schaal.



**Figuur 2.8:** One-stage detector met VGG als backbone, elke feature map met een verschillende schaal wordt geëvalueerd.

## Hoofdstuk 3

# Implementatie van Herkenning en detectie op mobiel platform

Dit hoofdstuk gaat over het implementeren van deep learning herkenningssystemen en detectiesystemen op een mobiel platform. Bij het uitvoeren van een neurale netwerk op een mobiel apparaat zal men rekening moeten houden met gelimiteerde rekenkracht en het beschikbaar geheugen. Het uitvoeren van de vele berekeningen en geheugen operaties zal ook invloed hebben op de batterij van het mobiel toestel. Eerst zullen we een aantal frameworks bespreken waarmee we neurale netwerken kunnen ontwerpen, trainen en deployen. Vervolgens zullen we een aantal bibliotheken bespreken die ons extra hulpmiddelen geven om detectiesystemen mee te ontwerpen. Dan gaan we een aantal frameworks bespreken die specifiek ontworpen zijn voor mobiele implementaties. Als laatste gaan we een aantal optimalisatietechnieken bespreken waarmee we het neurale netwerk verder kunnen optimaliseren. Er zal voornamelijk gefocust worden op Android implementaties.

### 3.1 Frameworks

Om machine learning modellen te ontwerpen en te trainen kan er gebruik gemaakt worden van frameworks. Deze frameworks geven de programmeur een set van tools die hem in staat stelt om op een overzichtelijke en flexibele manier deep learning modellen te ontwerpen en trainen. TensorFlow (Abadi et al. (2016)) en PyTorch (Li et al. (2020)) zijn de twee voornaamste frameworks om neurale netwerken te ontwerpen. Veel van de tools en bibliotheken die we gebruiken om complexere herkenningssystemen en detectiesystemen te ontwerpen worden bovenop deze frameworks gebruikt. Vanwege hun populariteit gaan we de twee frameworks bespreken en gaan we kijken welke mogelijkheden elk framework heeft.

### 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 deployen van neurale netwerken. Ook ondersteund TensorFlow meerdere programmeertalen zoals: Python, Java en C. Door de introductie van de Keras API (Chollet et al. (2015)) is TensorFlow meer gebruiksvriendelijk. Keras is een framework dat bovenop TensorFlow gebruikt kan worden en waarmee we machine learning modellen kunnen ontwerpen 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.

De TensorFlow Object Detection API (Abadi et al. (2015)) is een open source framework dat bovenop TensorFlow werkt. Het maakt het voor de programmeur eenvoudiger om detectiemodellen 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. De COCO dataset is een grote dataset voor objectdetectie en segmentatie gepubliceerd door Microsoft (Lin et al. (2015)). De API heeft voorgetrainde detectiemodellen van de volgende architecturen: 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)).

### 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 eenvoudig om nieuwe functionaliteiten toe te voegen door bestaande code aan te passen of nieuwe code toe te voegen.

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 objectdetectie 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 detectiemodel te trainen.

## 3.2 Objectdetectie 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 hulpmiddelen om de complexiteit van het proces 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 objectdetectoren en voorgetrainde modellen.

### 3.2.1 MMDetection

MMDetection voorgesteld door Chen et al. (2019) maakt deel uit van OpenMMLab en is een open source objectdetectie toolbox gebaseerd op PyTorch. Deze bibliotheek is specifiek ontworpen voor objectdetectie taken. MMDetection ondersteund ook 48 verschillende detectiemethodes zoals: YOLO (Redmon et al. (2016)), Faster R-CNN (Ren et al. (2016)), en SSD (Liu et al. (2016)). Van al deze architecturen samen heeft de programmeur toegang tot meer dan 200 voorgetrainde modellen. Een belangrijk aspect van MMDetection is dat het eenvoudige componenten bevat van verschillende detectie architecturen. Door deze componenten te combineren kan de programmeur een nieuw detectiemodel ontwerpen.

Doordat al de bounding box en masker operaties worden uitgevoerd op GPU's heeft MMDetection een grote trainingsnelheid. MMDetection biedt zelf geen ondersteuning om een bestaand model te optimaliseren naar een model voor mobiel gebruik.

### 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 een GPU. Via Modular, extensible design kan Detectron2 specifieke modules toevoegen aan bijna elk deel van een detectiesysteem. Detectron2 bevat meer dan 80 voorgetrainde modellen waarop de programmeur verder kan bouwen. Voor objectdetectie ondersteund Detectron2 zes verschillende standaardmodellen. Ondertussen heeft Detectron2 D2Go als uitbreiding gekregen dat ondersteuning biedt voor objectdetectie voor mobiele toepassingen.

### 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. Voor objectdetectie biedt GluonCV ondersteuning voor vier detectiemodellen: SSD (Liu et al. (2016)), Faster-RCNN (Ren et al. (2016)), YoloV3 (Redmon and Farhadi (2018)) en CenterNet (Duan et al. (2019)) Deze bibliotheek is ontworpen om snel prototypes en onderzoeksideeën te leveren voor onderzoekers. Dit doet GluonCV door de programmeur toegang te geven tot 170 voorgetrainde modellen, die hertraind kunnen worden voor een specifiek doel. Met behulp van een API die de complexiteit van het implementeren minimaliseert. GluonCV biedt ondersteuning voor PyTorch en MXNet om deep learning modellen te ontwerpen en trainen. De GluonCV documentatie bevat meer informatie voor MXNet dan voor PyTorch. MXNet is een framework zoals PyTorch en TensorFlow om neurale netwerken te ontwerpen, trainen en deployen (Chen et al. (2015)).

### 3.2.4 ImageAI

ImageAI (Martins et al. (2021)) is een eenvoudig te gebruiken Python bibliotheek die de programmeur in staat stelt om State-of-the-art AI features te implementeren. Het doel van deze bibliotheek is om met enkele lijnen code een objectdetector te trainen.

```
from imageai.Detection.Custom import DetectionModelTrainer

trainer = DetectionModelTrainer() # init
trainer.setModelTypeAsYOLOv3() # model type
trainer.setDataDirectory('path_to_dataset_dir') # dataset dir
#training configuratie(lijst met labels, batch size, aantal epochs,
#                           voorgetraind model)
trainer.setTrainConfig(object_names_array=["labels"], batch_size=4,
                        num_experiments=200,
                        train_from_pretrained_model="pretrained-
-yolov3.h5")

trainer.trainModel()
```

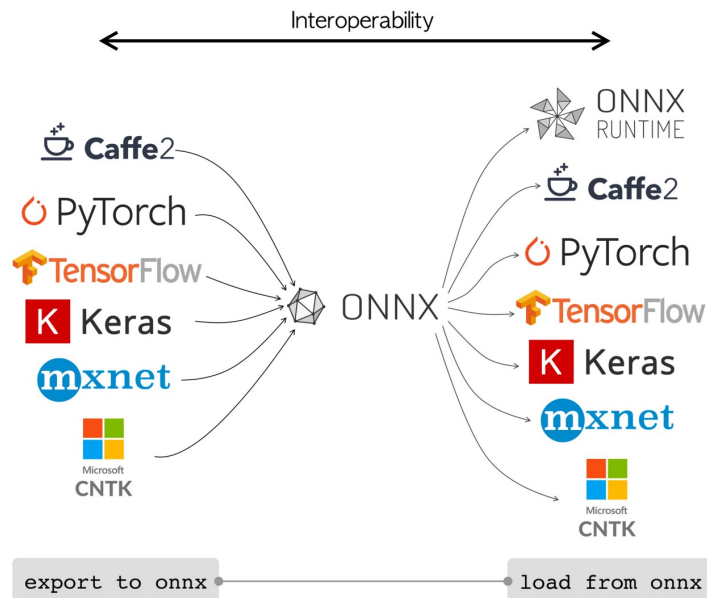
Deze bibliotheek levert tools voor herkenning, objectdetectie en videoanalyse. ImageAI geeft de programmeur toegang tot voorgetrainde detectiemodellen van RetinaNet (Lin et al. (2018)), YoloV3 (Redmon and Farhadi (2018)) en TinyYOLOv3 (Gai et al. (2021)) getraind met de COCO dataset. Deze bibliotheek werkt bovenop TensorFlow en maakt sinds juni 2021 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 deep learning model te exporteren naar een ander framework. Hierbij converteren we een bestaand model naar een ONNX model. Dit model kan op zijn beurt geconverteerd worden naar het gewenste framework (figuur: 3.1). Volgens de website van ONNX zijn er momenteel 23 frameworks die naar het ONNX framework geconverteerd kunnen worden, waaronder PyTorch en TensorFlow. Deze modellen kunnen we dan converteren naar een framework dat het model kan implementeren op een mobiel apparaat. Voor deze masterproef waarin een bestaand model ontworpen is in een bepaald framework, zal ONNX een belangrijke rol spelen. Op deze manier kunnen we een bestaand model implementeren met een framework dat mobiele implementatie ondersteunt.

ONNX definieert een aantal gemeenschappelijke sets van operaties en bouwblokken genaamd opsets. Deze opsets bevatten operaties die compatibel zijn met operaties van andere frameworks. Bij het converteren naar een ONNX model zullen de operaties die uitgevoerd worden in een bepaald framework vervangen worden door een ONNX equivalent gedefinieerd in een opsetversie. ONNX vernieuwt de opsets regelmatig, zo zal bij elke opsetversie de operaties worden aangepast waardoor er operaties bijkomen, wijzigen en wegvallen. Niet elk framework ondersteunt dezelfde opsetversie en dit zou tijdens het exporteren naar ONNX voor problemen kunnen zorgen. Dit is om-

dat een bepaalde operatie van een framework geen ONNX-equivalent zou kunnen hebben. Voor elke bibliotheek/framework is het aangeraden om dezelfde opsetversie te implementeren.



**Figuur 3.1:** Een weergave van de voornaamste frameworks die naar ONNX kunnen exporten en die een ONNX model kunnen importeren.

### 3.3.1 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 kan de convert functie worden opgeroepen uit de tf2onnx bibliotheek.

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

De conversie van TensorFlow naar ONNX wordt voor opsetversie 9 tot 15 ondersteund en getest. Opsetversie 6 tot 8 zou ook ondersteund moeten worden maar hier zijn geen testen voor. Bij de conversie wordt opsetversie 9 standaard gebruikt. Voor een meer recente opsetversie moet de opsetversie specifiek worden meegegeven via bovenstaande code met de optie '--opset'. In de Tf2onnx documentatie is een lijst terug te vinden met Tensorflow operaties die ONNX ondersteund. Als het TensorFlow model operaties bevat die niet herkend worden door onnxruntime, dan is het mogelijk om deze operaties mee te geven met de conversie functie. De TensorFlow naar ONNX conversie bestaat uit de volgende stappen:

- De converter verwacht een Protobuf bestand (Google (2021)) als input omdat de converter constante variabelen verwacht van het opgeslagen model. Protobuf is een serieel formaat dat platformonafhankelijk is.

- Het TensorFlow 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 geoptimaliseerd door bijvoorbeeld operaties samen te voegen of onnuttige operaties te verwijderen.

### 3.3.2 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.

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

De verplichte inputs nodig voor het uitvoeren van deze functie zijn: het PyTorch model, een set van inputargumenten en een output bestand. De set van inputargumenten bestaat uit een tuple van inputs of een dictionary die de input parameters van het model geeft. Bij het uitvoeren van `torch.onnx.export` zal de functie het model omzetten in een TorchScript model dat wordt besproken in paragraaf 3.4.3. Hierbij zal het model worden uitgevoerd en zullen al de uitgevoerde operaties bewaard worden. De inputargumenten 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. Dit doen we door de operator te registreren in de `native_functions.yaml` file die terug te vinden is in de volgende folder van de PyTorch bron code: `pytorch/aten/src/ATen/native` (PyTorch (2021)). Vervolgens moet in dezelfde folder de functie worden geschreven in een nieuwe C++ file. In deze folder is ook meer informatie beschikbaar voor het toevoegen van niet ondersteunde operaties.

## 3.4 Frameworks voor mobiele implementatie

Er zijn een aantal frameworks die de programmeur de mogelijkheid geven om een model te optimaliseren 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. We focussen op Android implementaties, maar de optie voor IOS zal ook vermeld worden. Niet elke framework voor mobiele implementatie voert de zelfde optimalisatietechnieken uit. Dus het

converteren van het standaardmodel naar het mobiele model zal voor elk framework een ander resultaat geven. Sommige modellen zullen na het optimaliseren in een bepaald framework een kleinere bestandsgrootte hebben dan bij andere frameworks. Deze frameworks zullen een model importeren van PyTorch, TensorFlow of ONNX.

### 3.4.1 TensorFlow Lite (TFLite)

Voor mobiele implementatie biedt TensorFlow de mogelijkheid om een TFLite model te ontwerpen (Abadi et al. (2015)). De meeste detectiemodellen ontworpen met TensorFlow Object Detection API zijn niet bedoeld om op mobiele apparaten te werken. Maar TensorFlow Object Detection API heeft ondersteuning om SSD (Liu et al. (2016)) en EfficientDet (Tan et al. (2020)) architecturen te exporteren naar een TFLite model. TFLite zorgt ervoor dat het model een lage latency heeft en een kleine binaire bestandsgrootte. Deze modellen kunnen geïmplementeerd worden op zowel Android als iOS applicaties. Via de TFLite ObjectDetector API kunnen we TFLite detectiemodellen eenvoudig implementeren in Android studio. Het converteren kan eenvoudig worden uitgevoerd via de volgende lijnen code in Python.

```
import tensorflow as tf

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

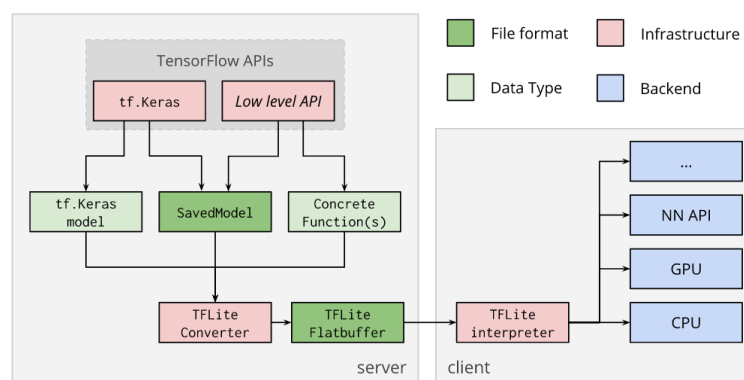
De TFLiteConverter neemt een TensorFlow model als input en op basis van dit model wordt een converterobject gegenereerd. Het TFLite model is eigenlijk een geoptimaliseerde FlatBuffer (Google (2014)), deze kan herkend worden door de .tflite extensie. FlatBuffer is serialisatie bibliotheek waarbij het 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 geheugengebruik. 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 via Grappler (Abadi et al. (2015)) om het model kleiner en sneller te maken. Grappler is het standaard optimalisatie systeem van TensorFlow om automatisch het model te verbeteren. Grappler voert een heel aantal optimalisatie uit, de volgende technieken zijn enkele van de voornaamste Grappler optimalisaties:

- Constant Folding optimizer: Is het vervangen van operaties door constanten, bij operaties waarvan de waarden niet veranderen tijdens de uitvoering.
- Remapper optimizer: operaties in het model worden samengevoegd tot één operatie.
- Memory optimizer: Analyseert het geheugengebruik van operaties in het model en voegt bij zwaar geheugengebruik swap operaties toe tussen CPU en GPU.
- Pruning optimizer: het verwijderen van operaties in het model die geen invloed hebben op de output.

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. De lijst met ondersteunde TFLite operaties is terug te vinden in de TFLite documentatie. 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.

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

TFLite geeft ook de mogelijkheid om modellen te hertrainen via TFLite Model Maker (Abadi et al. (2015)). Hierbij kan er voor objectdetectie enkel gebruik gemaakt worden van een EfficientDet detector (Tan et al. (2020)). Als EfficientDet voldoet aan de nodige specificaties en het is voldoende om dit netwerk te hertrainen, dan geeft TFLite model maker de mogelijkheid om een TFLite model te hertrainen met een eigen dataset. Op deze manier kan de conversiestap overgeslagen worden.



**Figuur 3.2:** Implementatieflow van een TensorFlow model naar een mobiele implementatie. 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 geïmplementeerd 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 modeloptimalisaties. De modeloptimalisaties die TFLite ondersteunt zijn: weight pruning, kwantisatie en clustering. Wat deze optimalisatietechnieken exact inhouden wordt later besproken in paragraaf 3.5. De kwantisatie optimalisatie kan als optie worden meegegeven aan de TFLiteConverter. Weight pruning en clustering moeten uitgevoerd worden voordat het model geconverteerd wordt naar TFLite.

De uitvoering van het TFLite model dat normaal op de CPU wordt uitgevoerd, kan versneld worden door gebruik te maken van verschillende hardwarecomponenten die op het toestel aanwezig zijn. Elk van deze hardwarecomponenten maakt gebruik van een eigen API waardoor er extra code moet

worden geschreven specifiek voor elke hardwarecomponent. De TensorFlow Lite Delegate API (Abadi et al. (2015)) dient als een interface in de vorm van delegates voor deze componenten. Deze delegates zorgen ervoor dat we geen specifieke API moeten implementeren voor een bepaalde hardwarecomponent. TFLite ondersteund meerdere delegates die kunnen geoptimaliseerd worden voor een specifiek platform. Voor Android kunnen we gebruik maken 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 wordt aangeroepen door deep learning applicaties om gebruik te maken van de GPU, DSP of NPU. In figuur 3.2 is een algemene workflow te zien om van een TensorFlow model naar een mobiele implementatie te gaan.

### 3.4.2 TensorFlow.js

TensorFlow biedt ook de mogelijkheid om via de TensorFlow.js API een model te implementeren in javascript. Op deze manier kunnen we gebruik maken van TensorFlow modellen in de browser en met Node.js. De TensorFlow.js API kan neurale netwerken modelleren, trainen en uitvoeren in javascript. Deze API heeft ook een set van voorgetrainde modellen, voor objectdetectie zijn er enkel voorgetrainde modellen van de SSD architectuur (Liu et al. (2016)). Ook kan via de TensorFlow.js API een TensorFlow model geconverteerd worden naar een TensorFlow.js model.

```
tensorflowjs_converter <dir_TensorFlow_model> <dir_TensorFlowjs_model>
```

TensorFlow.js ondersteund een gelimiteerd aantal TensorFlow operaties, in de TensorFlow.js documentatie is een lijst met ondersteunde operaties terug te vinden. Momenteel is er geen manier om standaard TensorFlow operaties te implementeren in TensorFlow.js.

### 3.4.3 PyTorch Mobile

PyTorch biedt zoals TensorFlow ook de mogelijkheid om het model te optimaliseren voor een mobiele implementatie. PyTorch mobile zit nog in zijn beta fase, dus er zijn componenten die gelimiteerde ondersteuning hebben. De onderstaande lijnen code geven aan welke code er moeten worden uitgevoerd om een PyTorch model te converteren 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.pt1")
```

Voordat we het PyTorch model kunnen optimaliseren voor mobiel gebruik moeten we het model eerst omzetten in een ScriptModule. De ScriptModule is een serieel model dat verder geoptimaliseerd kan worden en platformonafhankelijk is. Dit geeft ook de mogelijkheid om een model dat

ontwikkeld is in Pytorch te gebruiken 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 voorbeeldinput en bewaart al de operaties die zijn uitgevoerd. De bewaarde operaties worden gebruikt om een ScriptModule te genereren. Bij de trace functie zijn 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 operaties 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 aangepaste gewichten.
- 2D convoluties en lineaire operaties worden vervangen 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 kwantisatie na het trainen van het model. Wat deze techniek inhoudt wordt besproken in paragraaf 3.5. Zoals TensorFlow is er ook de mogelijkheid om de uitvoering van het model te versnellen door gebruik te maken van de NNAPI (Android (2021)). Op deze manier kan het PyTorch model operaties uitvoeren op andere hardwarecomponenten dan de CPU. De PyTorch NNAPI zit momenteel in de prototype fase en ondersteund maar een gelimiteerd aantal operaties die kunnen uitgevoerd worden met NNAPI. Via de volgende lijnen code kunnen we een TorchScript model optimaliseren zodat operaties worden uitgevoerd door de NNAPI.

```
import torch.backends._nnapi.prepare

nnapi_model = torch.backends._nnapi.prepare.convert_model_to_nnapi(traced,
                                                                    input_tensor)
```

#### 3.4.4 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.

```
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 optimalisatie uitgevoerd, de volgende lijst bevat 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 standaardoperaties 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 de NNAPI om het model sneller te maken. Het gegenereerde onnxruntime-model kan vervolgens geïmplementeerd worden in Android studio via de volgende lijnen code.

```
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);
```

### 3.4.5 ONNX.js

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.

```
const ort = require('onnxruntime-node');

const session = await ort.InferenceSession.create('./onnx_model.onnx');
const tensor = new ort.Tensor('float32', data, [3, 4]);
const feeds = { input: tensor};
const results = await session.run(feeds);
```

### 3.4.6 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 optimaliseert 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 Android implementaties.

Om een model te converteren naar CoreML zijn er 3 mogelijkheden. De eerste manier is rechtstreeks converteren vanuit PyTorch. Daarvoor moeten we eerst het 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 raadt Apple af omdat dit in nieuwe versies van CoreML niet meer ondersteund zal worden. CoreML biedt ook maar ondersteuning tot en met opsetversie 10 van ONNX. De volgende lijnen code tonen de implementatie voor de 3 manieren.

```
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')
```

### 3.4.7 Mobile AI Compute Engine (MACE)

Het MACE framework (Khan (2020)) is ontworpen door Xiaomi 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, zodat elk systeem evengoed presteert. Het MACE framework bespaart geheugen door de core library zo klein mogelijk te maken door het aantal externe dependencies te minimaliseren. Een YAML 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 geïmplementeerd kan worden in een applicatie via de MACE API.

### 3.4.8 Gerelateerd werk

In de paper geschreven door Luo et al. (2020) wordt PyTorch Mobile vergeleken met TensorFlow Lite voor verschillende classificatie architecturen. De architecturen die hier gebruikt worden zijn:

ResNet50, InceptionV3, DenseNet121, SqueezeNet, MobileNetV2 en MnasNet. Voor alle netwerk architecturen in deze paper geeft de optimalisatie naar TensorFlow Lite de kleinste bestandsgrootte 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 bestandsgrootte

Er zijn in voorgaande paragrafen al termen gevallen zoals pruning en kwantisatie. In deze paragraaf zullen we hier dieper op in gaan. we onderzoeken welke optimalisaties er kunnen worden toegepast om de snelheid en het gebruikt geheugen te verbeteren. Maar het optimaliseren van de snelheid en geheugen zal vaak negatieve gevolgen hebben voor de accuraatheid. Dus er zal een goede balans gevonden moeten worden tussen de optimalisaties en de accuraatheid van het neurale netwerk.

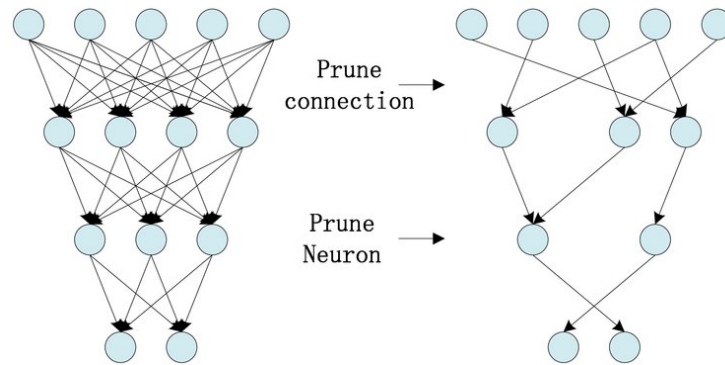
#### 3.5.1 Pruning

Pruning is de eerste stap van de Deep compression 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 bij het trainen van een neurale netwerk worden al de gewichten steeds berekend ongeacht hun invloed op het resultaat. Bij pruning gaan we de parameters met een kleine invloed op het resultaat verwijderd. Waardoor er geen berekeningen meer moeten uitgevoerd worden voor de verwijderde parameters, wat zorgt voor een snelheidswinst. Deze parameters hoeven we dan ook niet bij te houden waardoor het CNN model minder geheugen in beslag neemt.

Er zijn een aantal verschillende pruningtechnieken, we gaan er twee bespreken. Prune connection in figuur 3.3 of weight pruning wordt gebruikt door TensorFlow na het trainen van een model (zie paragraaf 3.4.1). Dit is het verwijderen van parameters die niet nuttig zijn in de gewichtstensor. Op deze manier wordt het aantal connecties tussen de lagen verminderd.

Een tweede techniek te zien op op figuur 3.3 is neuron pruning. Hierbij wordt een volledige kolom

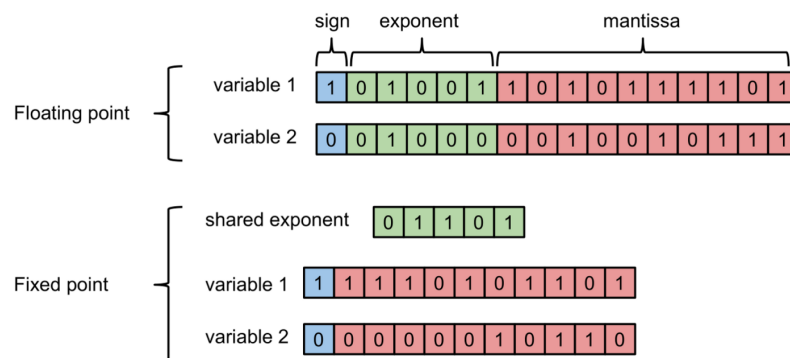
in de gewichten matrix verwijderd, daardoor verdwijnt een neuron in het neurale netwerk. Volgens Han et al. (2016) wordt voor VGG-16 (Simonyan and Zisserman (2015)) het aantal parameters met factor 13 verminderd, voor AlexNet (Krizhevsky et al. (2017)) met een factor 9. Zowel AlexNet als VGG-16 zijn hierbij getraind met de ImageNet dataset. Deze methode heeft zeer weinig tot geen effect op de accuraatheid.



**Figuur 3.3:** CNN voor en na pruning

### 3.5.2 Parameter kwantisatie

Het kwantiseren 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 moet op het systeem worden opgeslagen. De defaultrepresentatie van een waarde wordt opgeslagen als een floating point wat 4 bytes in beslag neemt. Dus voor miljoenen parameters hebben de gewichten veel schijfruimte nodig. Een mogelijke oplossing hiervoor is het kwantiseren van gewichten. Hierbij wordt de getalrepresentatie van de gewichten veranderd naar een representatie die minder geheugen in beslag neemt, een voorbeeld is te zien op figuur 3.4.



**Figuur 3.4:** kwantisatie van twee floating point variabelen die worden omgezet naar twee fixed point variabelen met een gemeenschappelijke exponent.

### 3.5.3 Weight Clustering

Hierbij worden de waarden van gewichten beperkt tot een set van beschikbare waardes (Han et al. (2016)) dit is te zien op figuur 3.5. Waarbij de waardes eenmalig 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 accurate. Han et al. (2016) past vervolgens Huffman encoding toe die een compressie uitvoert op de geclusterde parameters.



**Figuur 3.5:** Weight clustering: links zijn allemaal verschillende gewichten en na het uitvoeren van weight clustering zijn er in de matrix pointers terug te vinden die wijzen kunnen wijzen naar vier verschillende waarden.

### 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 accurateheid van het CNN. Bij matrix factorisatie (Goel et al. (2020)) worden grootte en complexe matrices opgesplitst in verschillende kleinere en eenvoudigere matrices. Op deze manier worden ook redundante operaties uit het model gefilterd.

## Hoofdstuk 4

# Compatibiliteit van herkenningssystemen

Voor het herkenningssysteem bestuderen we uitgebreid de ResNet50 architectuur. We gaan dit model trainen in het PyTorch en TensorFlow framework. Om vervolgens de mogelijke paden te bestuderen naar een mobiele implementatie.

### 4.1 ResNet50

He et al. (2015) heeft vastgesteld dat als het aantal lagen van een CNN toeneemt dat op een bepaald moment de training accuraatheid daalt. Dit verschijnsel noemt men de vanishing gradient. In paragraaf 2.1.3 hebben we besproken hoe we de gradient kunnen berekenen tijdens het trainen van een CNN. Voor elke laag in het CNN moet de gradient opnieuw berekend worden door telkens opnieuw de afgeleide te berekenen. Hierdoor wordt de gradient steeds kleiner en kleiner tot deze een minimum bereikt. Waardoor de gewichten in de eerste lagen heel traag aanpassen of zelfs niet meer veranderen. He et al. (2015) die dit probleem hebben vastgesteld hebben dit opgelost door gebruik te maken van skip connections. Hierbij wordt de input van een laag rechtstreeks met een volgende laag die x aantal lagen verder ligt. Op deze manier worden de gradienten per laag niet meer kleiner. ResNet50 bestaat uit 50 convolutie lagen waarbij er een skip connection plaatsvindt per 3 lagen. De resnet50 architectuur is opgebouwd uit ResNet blokken die bestaan uit 3 convolutie lagen en 1 skip connection.

### 4.2 Van TensorFlow naar mobiel framework

In figuur ... is het standaard ResNet50 netwerk dat in TensorFlow geïmplementeerd kan worden vanuit Keras te zien. In deze figuur kunnen we de voornaamste operaties terug vinden die in het netwerk gebruikt worden. Ook vinden we in deze architectuur 2 verschillende ResNet blokken terug: de ID-blok en de convolutie blok. In figuur ... worden de 2 verschillende blokken en hun operaties

weergegeven. Hierbij is te zien dat in het convolutieblok 2 extra operaties worden uitgevoerd tijdens de skip connection.

Voor het experiment van ResNet50 maken we gebruik van het standaard ResNet50 netwerk dat in TensorFlow geïmplementeerd kan worden vanuit Keras. Dit netwerk is vervolgens hertrained met een katten en honden dataset ... . Al de code is uitgevoerd in Google Colab met een GPU runtime.

In tabel 5.1 is te zien welke operaties er zijn terug te vinden in het ResNet50 model dat onet getrained is. Ook kunnen we in deze tabel terug vinden welke operaties TFLite ondersteund en met welke ONNX opset versie deze operaties compatibel zijn. Tijdens het converteren van TensorFlow naar TFLite worden optimalisaties uitgevoerd waarbij enkele operaties worden verwijderd, constant gemaakt of samengevoegd met een voorgaande laag. De tf2onnx converter maakt standaard gebruik van ONNX opset versie 9. Bij het converteren van TensorFlow naar ONNX onder standaard omstandigheden krijgen we de volgende fout **ValueError: StridedSlice: only strides=1 is supported**. Voor het converteren van TensorFlow naar ONNX is minstens opset versie 10 nodig.

#### 4.2.1 Van PyTorch naar mobiele implementatie

**Tabel 4.1** Alle operaties die terug te vinden zijn in het ResNet50 model en hun compatibiliteit met andere frameworks

Operaties	TensorFlow → TFLite	ONNX Opset	MACE
AddV2	fused/ond	1,6,7,13,14	/
AveragePooling	ondersteund	1	ond
BiasAdd	ond	1	ond
Cast	Ondersteund	1,6,9,13	/
Const	const	1,9,11,12,13	/
Conv2D	Ondersteund	1,11	ond
FusedBatchNormV3	fused/ond	6	/
GreaterEqual	Ondersteund	7	/
Identity	ver, fused, const	1,13,14,16	/
MatMul	Ondersteund	1,9,11	ond
MaxPool	Ondersteund	1,8,10,11,12	ond
Mean	Ondersteund	1,6,8,13	/
Mul	Ondersteund	1,6,7,13,14	/
NoOp	ver	/	/
Pack	Ondersteund	1,4,11,13	/
Pad	Ondersteund	1,2,11,13	ond
Placeholder	const/verw	1	/
RandomUniform	const/verw	1	/
Relu	fused	1,6,13,14	ond
Select	const/verw	7	/
Shape	ver/const	1,13,15	/
StatefulPartitionedCall	ond	/	/
StridedSlice	Ondersteund	1, 10	/
VarHandleOp	verw	/	/



## Hoofdstuk 5

# Compatibiliteit van detectie systemen

Voor detectiesysteem bestuderen we uitgebreid de Faster-RCNN architectuur met een ResNet50 backbone en de YOLO architectuur. We gaan voor deze modellen vertrekken vanuit het PyTorch en TensorFlow framework. Om vervolgens de mogelijke paden te bestuderen naar een mobiele implementatie.

### 5.1 faster rcnn

met een resnet50 backbone

### 5.2 Van TensorFlow naar mobiel mobiele implementatie

Om van een TensorFlow object detectie model te gaan naar een mobiele implementatie kunnen we een TFLite of een ONNX model implementeren in Android Studio. Het model dat we willen converteren is een voorgetrained Faster-RCNN object detectie model van de TensorFlow object detection API. Dit Faster-RCNN model gaan we vervolgens converteren naar een TFLite of ONNX formaat.

#### 5.2.1 TFLite implementatie

De TensorFlow object detection api stelt zelf een manier voor om een model te converteren naar het TFLite formaat. Via de volgende lijnen code kunnen we een model converteren. .... Tijdens het converteren krijgen we de volgende error: `ValueError: Only ssd or center_net models are supported in tf.lite. Found faster_rcnn in config.` Dit komt doordat de TFLite converter van de tensorflow object detection api enkel de conversie voor SSD en Centernet ondersteund.

Voor een object detectie model te converteren kan er ook gebruik gemaakt worden van de standaard TFLiteConverter. Bij het converteren naar TFLite kan de ConcatV2 operatie niet geconverteerd worden naar de TFLite concatenation operatie. Door het toevoegen van TensorFlow Flex

operaties kan het model zonder problemen geconverteerd worden naar een TFLite model. Omdat we gebruik maken van Flex operaties moet de TensorFlow core bibliotheek mee geïmplementeerd worden in Android studio.

Als we het TFLite model willen implementeren in Android studio via de TensorFlow Lite Task Library dan moeten we Metadata aan het model toevoegen. We moeten Metadata toevoegen aan het TFLite model zodat bij het uitvoeren van het model de TFLite API weet wat het input en output types zijn van het model. Maar bij het toevoegen van Metadata aan het TFLite model krijgen we de volgende fout: Keyerror 2708. Deze fout geeft weinig informatie, maar de oorzaak is dat de methode die de Metadata aan het model toevoegt maximaal 4 outputs verwacht. Het geconverteerd Faster-RCNN model heeft 8 outputs. Door het aantal outputs te reduceren tot 4 outputs kunnen we succesvol Metadata aan het model toevoegen. Bij het uitvoeren van het model met metadata krijgen we de volgende error: .... Deze fout ontstaat doordat tijdens het converteren naar TFLite de output vorm van [1, 300, 4] naar [1, 1, 1] verandert voor de bounding box coördinaten. Het model zal nog steeds meerdere bounding boxen voorspellen, dus het model verwacht een output buffer van [1, 300, 4]. Maar de outputbuffer die de TensorFlow Lite Task Library aanmaakt is van grootte [1, 1, 1].

We kunnen het TFLite model ook implementeren via de TensorFlow Lite Interpreter API, hierbij is er geen Metadata nodig. Door gebruik te maken van de TensorFlow Lite Interpreter API moeten we zelf de input en output buffers definiëren. De vereiste informatie om de correcte buffers te definiëren kan uit het TFLite model gehaald worden. Het TFLite model bevat de juiste informatie voor de inputbuffer, dus deze kan eenvoudig aangemaakt worden. Bij het aanmaken van de outputbuffers krijgen we terug een fout dat onze outputbuffer van de verkeerde grootte is. Dit komt doordat de output vorm volgens het TFLite model [1, 1, 1] is, maar het model levert meer resultaten. De TensorFlow Lite Interpreter API geeft ons de mogelijkheid om de outputbuffers aan te passen zodat deze de gewenste grootte hebben. Op deze manier kunnen we succesvol een Faster-RCNN model uitvoeren op een mobiel apparaat.

### 5.2.2 ONNX implementatie

In het Faster-RCNN model wordt er gebruik gemaakt van de Round operatie die pas beschikbaar is sinds opset versie 11. Al de andere operaties van het Faster-RCNN model worden ondersteund in eerdere opset versies. De round operatie zal ervoor zorgen dat bij het converteren naar onnx formaat minstens een opset versie 11 nodig is.

## 5.3 Van PyTorch naar mobiele implementatie

Om van een PyTorch object detectie model te gaan naar een mobiele implementatie kunnen we een Pytorch of een ONNX model implementeren in Android Studio. Het model dat we willen converteren is een voorgetraind Faster-RCNN uit de Torchvision bibliotheek. Dit model gaan we vervolgens converteren naar een TorchScript model een ONNX model

**Tabel 5.1** Alle operaties die terug te vinden zijn in het ResNet50 model en hun compatibiliteit met andere frameworks

Operaties	TensorFlow → TFLite	ONNX Opset	MACE
BroadcastTo	ond	8	/
ConcatV2	/	1	ond
Equal	ond	1	/
Exp	ond	1	/
ExpandDims	ond	1	/
Fill	ond	7	/
Floor	ond	1	ond
GatherV2	ond	1	/
Greater	ond	1	/
Less	ond	1	/
LogicalAnd	ond	1	/
Maximum	const,verw,fus	1	/
Minimum	const,verw,fus	1	/
NonMaxSuppressionV5	ond	10	/
Range	ond	7	/
RealDiv	/	1	/
Relu6	const,verw,fus	1	ond
Reshape	ond	1	ond
ResizeBilinear	ond	7	/
Round	ond	11	/
SelectV2	ond	7	/
Slice	ond	1	ond
Softmax	ond	1	ond
Split	ond	1	/
Sqrt	const,verw,fus	1	/
Square	const,verw,fus	1	/
Squeeze	ond	1	/
Sub	const,verw,fus	1	/
Sum	ond	1	/
Tile	const,verw,fus	1	/
TopKV2	ond	1	/
Transpose	ond	1	/
Unpack	ond	1	/
Where	ond	9	/
ZerosLike	ond	1	/

### 5.3.1 TorchScript implementatie

Bij het converteren naar een Torchscript module dat verder geïoptimaliseerd kan worden voor mobiel gebruik ondersteund PyTorch enkel `jit.script` voor Faster-RCNN. Na het converteren kunnen we het model verder optimaliseren voor mobiel gebruik. Bij het opslaan van deze geïoptimaliseerde scriptmodule voor mobiel gebruik crasht Google Colaborate zonder een boodschap. In plaats van het model op te slaan met de `_save_for_lite_interpreter` methode moeten we het model opslaan voor script gebruik. Hiervoor moet ook de `android_pytorch` bibliotheek worden geïmplementeerd in plaats van de `android_pytorch_lite` bibliotheek. bij het uitvoeren van het script model krijgen we een fout dat de `.nms` operatie niet wordt ondersteund. PyTorch geeft de mogelijkheid om de `torchvision_ops` bibliotheek te implementeren die ervoor zorgt dat alle faster rcnn operaties worden geïmplementeerd. Maar na het implementeren van deze bibliotheek krijgen we de volgende error .... We kunnen de Torchvision\_ops bibliotheek die terug te vinden is in de Github repositorie van Torchvision handmatig te implementeren in het Android studio project. Op deze manier kunnen de Torchvision operaties wel implementeren in Android studio. Wel moet het PyTorch model volledig onder CPU runtime worden geconverteerd naar een TorchScript model. anders krijgen we de volgende fout: .... Op deze manier kunnen we succesvol een PyTorch Faster-RCNN model uitvoeren op een mobiel toestel.

### 5.3.2 ONNX implementatie

Zoals bij TensorFlow is hier ook een minimale opset versie van 11 vereist. Al de andere operaties van het Faster-RCNN model worden ondersteund in eerdere opset versies.

# 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.

Chollet, F. et al. (2015). Keras. <https://github.com/fchollet/keras>.

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)*.
- 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.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.
- 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.
- 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. <https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526>.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60.
- 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 Dollár, P. (2018). Focal Loss for Dense Object Detection.
- Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., and Dollár, P. (2015). Microsoft coco: Common objects in context.

- 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.
- Ma, H., Gou, J., Wang, X., Ke, J., and Zeng, S. (2017). Sparse coefficient-based  $k$ -nearest neighbor classification. *IEEE Access*, 5.
- 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.
- 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](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>.
- PyTorch (2021). pytorch/aten/src/ATen/native at master · pytorch/pytorch. <https://github.com/pytorch/pytorch>.
- 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. <https://arxiv.org/abs/1804.02767>.
- Ren, S., He, K., Girshick, R., and Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. <http://arxiv.org/abs/1506.01497>.
- Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2019). MobileNetV2: Inverted Residuals and Linear Bottlenecks.
- Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. <https://arxiv.org/abs/1409.1556>.
- Tan, M., Pang, R., and Le, Q. V. (2020). EfficientDet: Scalable and Efficient Object Detection.
- Wu, Y., Kirillov, A., Massa, F., Lo, W.-Y., and Girshick, R. (2019). Detectron2. <https://github.com/facebookresearch/detectron2>.

## **Bijlage A**

# **Uitleg over de appendices**

Bijlagen worden bij voorkeur enkel elektronisch ter beschikking gesteld. Indien essentieel kunnen in overleg met de promotor bijlagen in de scriptie opgenomen worden of als apart boekdeel voorzien worden.

Er wordt wel steeds een lijst met vermelding van alle bijlagen opgenomen in de scriptie. Bijlagen worden genummerd met een drukletter A, B, C,...

Voorbeelden van bijlagen:

Bijlage A:     Detailtekeningen van de proefopstelling

Bijlage B:     Meetgegevens (op USB)



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.iw.kuleuven.be](http://www.iw.kuleuven.be)

