Professionele Bachelor Toegepaste Informatica

Smart Pool Table:
Deep Reinforcement Learning Agent

Dary Schaeken

Promotoren:

Jeroen Benats
Arno Barzan

C4J
Hogeschool PXL Hasselt

Bachelorpaper Academiejaar 2018-2019

Professionele Bachelor Toegepaste Informatica

Smart Pool Table:
Deep Reinforcement Learning Agent

Dary Schaeken

Promotoren:

Jeroen Benats — C4J
Arno Barzan — Hogeschool PXL Hasselt

Bachelorpaper Academiejaar 2018-2019

Dankwoord

Een zoektocht naar een plaats waar ik me thuis voelde; leidde me uiteindelijk drie jaar geleden naar deze opleiding. Nu loopt dit driejarig traject ten einde met dit eindwerk en ik wil alle mensen bedanken die me gesteund hebben doorheen deze jaren, die niet altijd even vlot verliepen. Allereerst en voornaamst, mijn ouders en zeven broers en zussen. Ze begrepen niet altijd wat ik deed, maar steunden me elke stap die ik heb afgelegd. Naast mijn familie verdient mijn vriendin, Kathy Janssens, haar eigen bedankje voor het opvangen van mijn vele frustraties.

Natuurlijk bedank ik Bewire NV voor het mogelijk maken van de stage, en de steun die alle medewerkers boden om het project tot een goed einde te brengen. Er zijn altijd mensen die meer doen als nodig, en om deze personen in de verf te zetten, wil ik graag Jeroen Benats, die ondanks zijn drukke schema, tijd maakte om onze promotor te zijn, Jens Mortier, die het project vanaf het begin begeleidde, en Jonas Liekens en Anissa Schirock voor de technische begeleiding.

Naast de bedrijfspromotor, bedank ik Arno Barzan voor het ondersteunen van deze stage als de hogeschoolpromotor. Zijn ondersteuning heeft me vaak een hart onder de riem gestoken op de dagen waar de stage niet zo vlot verliep als ik wilde.

Ik heb dit echter niet alleen bereikt. Mijn medestudenten, met wie ik veel lijden heb kunnen delen, verdienen ook een bedankje. Specifiek wil ik hier mijn stagepartner Yente Martens aanhalen, die de stageopdracht samen met mij heeft volbracht. Ook wil ik Stephanie Govaers bedanken, voor de steun en het bijhouden van de deadlines voor alle stagiairs die zich in de kantoren van Bewire bevonden. Als laatste student, mag ik Timo Biesmans niet vergeten, die met zijn vaardigheden als software manager ons project in goede banen hield.

Als laatste bedank ik die lectoren, die door met hun tijd, vaardigheden, kennis maar vooral passie, mij hebben gevormd tot de professional die ik geworden ben.

Abstract

Artificiële intelligentie is een groeiend veld in de IT-sector. De verschillende takken van artificiële intelligentie worden over het algemeen apart bestudeerd. Binnen deze paper zal er zowel een *supervised learning* model, in de vorm van objectdetectie, als een *reinforcement learning agent* toegepast worden op het spel 'pool'.

Het onderzoek zelf focust zich op *reinforcement learning*. In het onderzoek wordt bekeken welk type algoritmes er beschikbaar zijn, en welk algoritme gepast zou zijn voor de probleemstelling. Verder wordt er kort bekeken hoe de diepte en breedte van een neuraal netwerk impact hebben op de werking van de algoritmes.

Uit het onderzoek is gebleken dat *deep deterministic policy gradient* of DDPG het algoritme is dat het best wordt toegepast binnen een omgeving met continue acties en *states*. Daarnaast is er bepaald dat het toevoegen van neuronen aan de lagen van een neuraal netwerk, tenminste als DDPG toegepast wordt, voor stabielere resultaten zorgt.

Deze bachelorproef vat de concepten die aan de grond van *reinforcement learning* liggen samen, en legt uit hoe een *deep reinforcement learning agent* ontwikkeld kan worden. Deze *agent* wordt uiteindelijk toegepast op een omgeving die het spel pool simuleert.

Inhoudsopgave

Lijst van gebruikte figuren

Lijst van gebruikte vergelijkingen

Lijst van gebruikte tabellen

Lijst van gebruikte afkortingen

A3C	Asynchronous Advantage Actor-Critic
AI	Artificiële Intelligentie
API	Application Programming Interface
CNN	Convolutional Neural Network
CNTK	Microsoft Cognitive Toolkit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
cuDNN	CUDA Deep Neural Network
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Network
GPU	Graphics Processing Unit
LeakyReLU	Leaky Rectified Linear Unit
LTS	Long Term Support
MAS	Multi Agent System
MDP	Markov Decision Process
MRP	Markov Reward Process
mse	Mean Squared Error
MVP	Minimal Viable Product
ONEIROS	Open-ended Neuro-Electronic Intelligent Robot Operating System
PoC	Proof of Concept
PyPI	Python Package Index
RAM	Random-access Memory
SARSA	State Action Reward State Action
VoTT	Visual Object Tagging Tool

Inleiding

Artificiële intelligentie is een onderzoeksthema waar al jaren aan gewerkt wordt. Artificiële intelligentie was als veld al een tweetal decennia oud toen de eerste pc's op de markt werden gebracht. Er kan echter nog steeds gezegd worden dat het veld in zijn kinderschoenen staat. In de jaren 50 toen het veld ontstond, waren de beschikbare computers simpelweg niet krachtig genoeg om de berekeningen te maken die nodig waren om een computer intelligent te maken.

De laatste jaren is dit echter sterk veranderd, en zetten ook meer en meer grote bedrijven in op deze tak van de informatica. Zo kocht Google Deepmind Technologies in 2014. Dit bedrijf schokte de wereld met zijn AlphaGo door de toenmalige Go kampioen Lee Sedol te verslaan.

Dit project focust zich op de mogelijkheden van *reinforcement learning*, een tak van artificiële intelligentie die mogelijk kan leiden tot generale AI, een AI die succesvol elke intellectuele taak kan uitvoeren die een mens kan uitvoeren, inclusief het leren van nieuwe activiteiten of kennis. Er wordt onderzocht of de huidig beschikbare algoritmes het mogelijk maken om een computer te leren poolen zoal een mens.

Om dit te bereiken wordt echter niet alleen de *reinforcement learning* tak van AI aangesproken, maar wordt ook *supervised learning* een belangrijk onderdeel. Om de computer toegang te geven tot het speelveld, wordt er objectdetectie toegepast. Dit werk bestaat hierdoor uit twee onderdelen, de praktische creatie van het systeem waar deze twee intelligenties leven en het onderzoek naar *reinforcement learning*.

I. Stageverslag

1 Bedrijfsvoorstelling

1.1 Introductie

C4J is één van de zes onderbedrijven van de koepel Bewire. Het is onder C4J dat de 'Smart Pool Table' stage plaatsvond. Echter omdat elk van deze onderbedrijven hun eigen expertise naar de tafel brengen, zijn er raakvlakken met al deze bedrijven geweest.

Bewire richt zich op een hoge kwaliteit bij elk aspect van het ontwikkelen van een applicatie. 'Goed genoeg' is niet iets dat past in hun vaandel. Zoals het hoort, testen ze elke feature van hun werk af, of het nu gaat om een volledig nieuw systeem, of een verbetering op wat een klant in gebruik heeft. Ook zullen ze nooit de verantwoordelijkheid over slechte functionaliteit afschuiven op wat er al bestaat, maar doen ze alles dat mogelijk is om de oude code te integreren als de klant dit vraagt.

Een tweede doel van Bewire is om een zo sterk mogelijk partnerschap tussen de verschillende aspecten van applicatieontwikkeling te creëren. Binnen het team zelf moet er een sterke (professionele) band zijn, maar tussen klant en team moet er net zo goed een goede relatie zijn. Enkel zo kan het bedrijf de perfecte applicatie voor de klant afleveren.

Als laatste wordt er binnen Bewire veel gedaan zodat hun specialisten hun kennis kunnen delen onder elkaar. Er zijn bijvoorbeeld de 'B-inspired' sessies, waar werknemers hun passies kunnen komen delen en het bedrijf of hun collega's warm kunnen maken voor deze nieuwe concepten. [1]

Deze drie pijlers worden de *Bewire way* genoemd en heeft ze tot driemaal toe het 'Great Place To Work' certificaat opgeleverd [2] [3]. Figuur 1 beeldt de structuur van Bewire uit, toen Appmind nog geen onderdeel vormde van het bedrijf.

Figuur 1 Bewire Ecosysteem [2]

1.2 C4J

C4J is waar je de backend specialisten van Bewire zal vinden. Ze specialiseren zich vooral in Java, maar je kan er ook enkele NodeJS-experten vinden. Naast het creëren van applicaties coachen ze verder nog externe partijen om hun kennis te vergroten.

Om hun klanten te overtuigen van hun expertise, werken ze met *Proof of Concepts*; kleinere applicaties die de bruikbare technologieën tonen. Hoewel ze gefocust zijn op backend, kunnen ze volledige oplossingen voorzien door gebruik te maken van de andere experten binnen Bewire [4].

1.3 Evance

Evance is de plaats waar frontend ontwikkelaars hun thuis kunnen vinden. Zij zijn er allemaal van overtuigd dat de frontend dan ook de belangrijkste laag van een applicatie is.

“Via de frontend willen wij iedereen zijn zintuigen digitaal prikkelen: de gebruiker van vanaf het eerste moment mee nemen naar een unieke ervaring. Natuurlijk is het opslaan en aanpassen van gegevens uiteindelijk belangrijk. De kunst echter, is het speciaal maken van deze ervaring voor iedereen [5].”

1.4 Codrigo

Voor ondersteuning bij het ontwikkelproces, en voor het verkorten van de tijd voordat de applicatie opgeleverd is, is Codrigo het aanspreekpunt. Hun agile manier van werken bestaat uit een workshop met de klant, waarbij ze de effectieve wensen van de klant te bepalen. Van hieruit wordt er met een high level offerte gewerkt. Pas als de klant hiermee tevreden is zal het project verder uitgeschreven en ontwikkeld worden. Hier stopt het nog niet voor Codrigo, want zelfs na de aflevering blijven ze ondersteuning bieden aan de klant. [6]

1.5 Dots & Arrows

De bedrijfsnaam is de beschrijving van hun werk; het verbinden van de punten in een project met pijlen. Hun taak bestaat uit het analyseren, ontwikkelen en testen van meerdere applicaties totdat deze vloeiend samenwerken. Om dit te bereiken gebruiken ze Mulesoft, een uniek integratieplatform om *Software as a Service* en *Enterprise* applicaties te verbinden. [7]

1.6 Trase

Blockchain was niet zo lang geleden de nieuwste hype, maar Bewire zag niet alleen hype, maar ook businesspotentieel in de technologie. Om hierop te in te spelen werd Trase opgericht, waar *blockchains* geïntegreerd worden in bedrijven met Hyperledger Fabric. Dit is een tool om private *blockchains* op te ontwikkelen.

Naast het ontwikkelen van blockchains, is er binnen Trase ook nog de Trase University, waar ze inhuis opleidingen verzorgen en de technologie verder uitdiepen. [8]

1.7 Appmind

Appmind is de laatste toevoeging aan de Bewire-koepel, en richt zich, zoals de naam verklapt, op het creëren van mobiele applicaties. Hun klantgerichte werkwijze heeft vier doelen. Allereerst bouwen ze zowel de iOS- als de Android-applicatie vanaf dezelfde codebase.

Ten tweede willen ze een snelle delivery en leggen ze zichzelf een deadline van maximum drie maanden op. Ze gebruiken hun expertise om het *UX design* zo te maken dat elke gebruiker zich speciaal

kan voelen omdat ze de applicatie gebruiken. En ten slotte, bieden ze doorlopend ondersteuning met nieuwe features waar nodig. [9]

2 Voorstelling stageopdracht

2.1 Probleemstelling

De opdracht start niet vanuit een probleemstelling, maar vanuit een experiment: Artificiële Intelligentie toepassen op het spel pool. De volledige opdracht bestaat uit drie onderdelen. Het eerste gedeelte is gefocust op een objectdetectie-model creëren dat de verschillende poolballen kan herkennen.

Het tweede onderdeel is het spel pool volgen, door de huidige status van het spel te kunnen opvragen via het objectdetectie-model, en de regels van pool toe te passen op deze observaties.

Het derde gedeelte is een *reinforcement learning* agent maken die het spel pool zal leren spelen via een simulatie. Uiteindelijk kunnen de voorspellingen van dit model de huidige speler hints geven wat zijn beste volgende zet kan zijn.

2.2 Technologieën

Om het eindresultaat van dit project te kunnen bereiken, zijn er een aantal technologieën en hardware-onderdelen nodig. Ter verduidelijking worden deze componenten toegelicht in deze sectie.

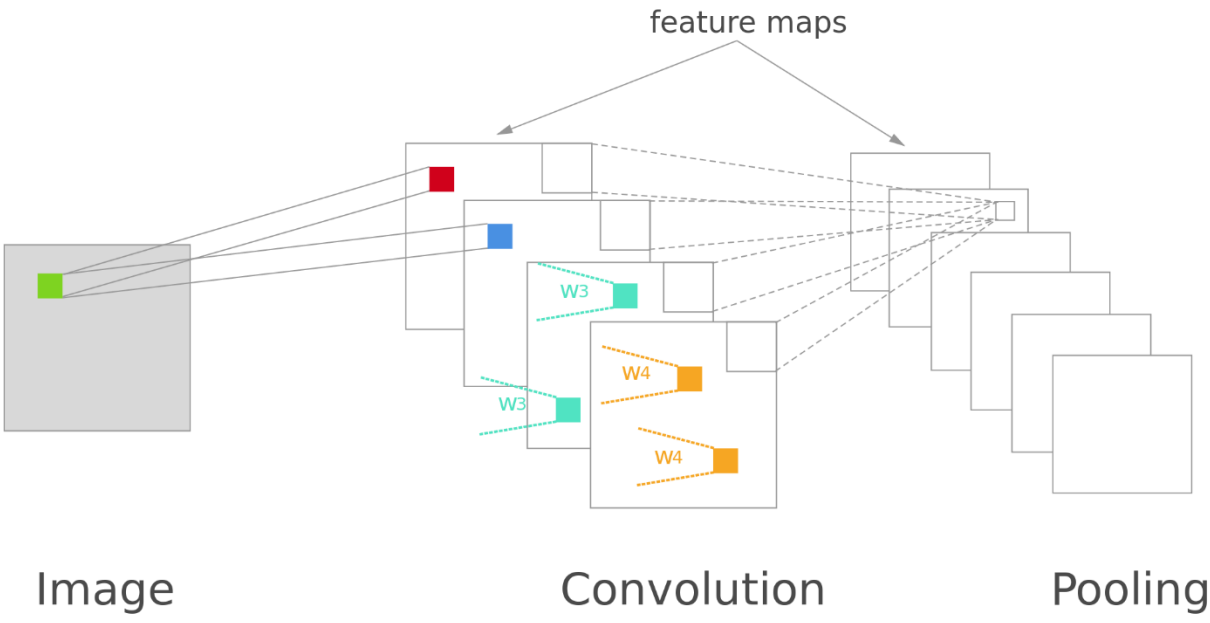
2.2.1 Artificiële intelligentie

Zoals voorheen benoemd, maakt AI een groot onderdeel uit van dit project. Er zijn verschillende takken binnen de AI, waarvan drie grotere, namelijk *supervised*, *unsupervised* en *reinforcement learning*. Belangrijk in dit project zijn *supervised* en *reinforcement learning*.

Naast deze takken is het ook nog belangrijk te benoemen dat beide toepassingen van AI voorbeelden zijn van *deep learning*. *Deep learning* is toepasbaar op alle takken van AI, en slaat op het gebruik van grote artificiële neurale netwerken. [10] Als we er een definitie op plaatsen, wordt er gesproken van *deep learning* vanaf dat er twee of meer *hidden layers* zijn. Een *hidden layer* (letterlijk: een verborgen laag) is een laag neuronen die zich tussen de invoerlaag en uitvoerlaag bevindt. De buitenwereld ziet deze laag niet, en kan hier ook niet mee interageren, vandaar de naam *"hidden"*.

Omdat AI een enorm breed domein is, limiteert dit werk zich hier tot één van de onderdelen die in het project gebruikt worden. *Reinforcement learning* wordt nauwer aangehaald binnen het tweede onderdeel van dit eindwerk.

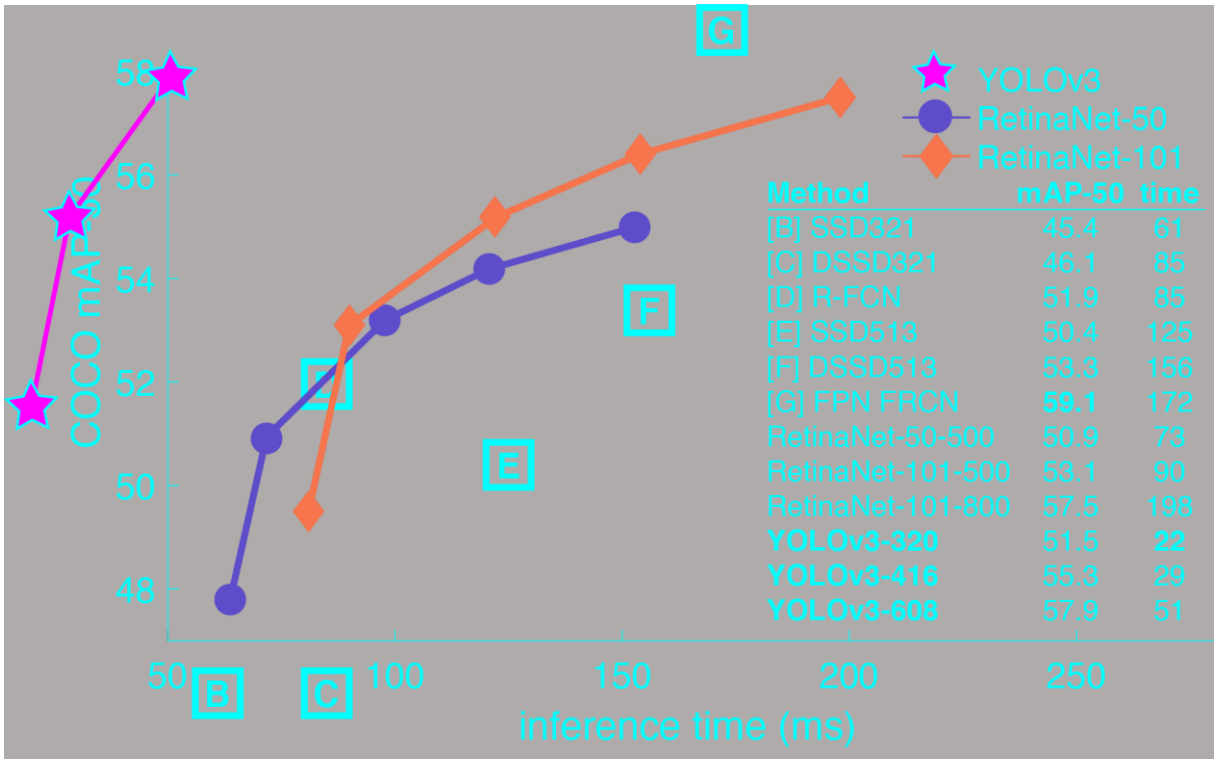
Het onderdeel dat hier nog besproken wordt, is het CNN dat gebruikt wordt om objectdetectie te bereiken. Een *convolutional neural network* is een type neuraal netwerk benoemd voor zijn typerende convolutie-lagen. Dit soort netwerk wordt voornamelijk gebruikt om afbeeldingen te verwerken en is hier tot nog toe de beste optie voor. [11] Naast de convolutie-lagen zijn ook de *pool*-lagen belangrijk. Figuur 2 toont deze lagen in hun typerende volgorde.



Figuur 2 Representatie van de belangrijkste lagen in een CNN [50]

Beide van dit type lagen maken gebruik van het *sliding window*-principe om hun data te doen krimpen, en de belangrijkste gegevens eruit te vissen. Dit zorgt voor een betere generalisatie van het te herkennen object, zodat er geen specifieke instantie herkend wordt, maar alle instanties van het object.

Het specifiekere netwerk waarvoor gekozen is binnen dit project is YOLOv3-tiny. Dit netwerk heeft een beperkte grootte, wat het leerproces versnelt, herkenning versnelt, en ook minder zwaar is voor de hardware. Aangezien er meer dan één systeem draait in de uiteindelijke configuratie is hier ook rekening mee gehouden. Een laatste reden dat de voorkeur naar de *tiny*-versie ging, is dat door zijn beperkte grootte, er meer ingezet kon worden op de resolutie van de foto's die verwerkt worden. Dit helpt met objecten van een afstand te herkennen. Figuur 3 toont YOLOv3 in vergelijking met andere CNN.



Figuur 3 YOLOv3 in vergelijking met andere CNN [12]

2.2.2 Keras

Keras is een *high-level application programming interface* die neurale netwerken bouwen versimpeld. Deze *library* is gemaakt in Python, en kan zijn functionaliteit uitvoeren op een aantal verschillende backends, namelijk Theano, CNTK en TensorFlow.

De naam Keras komt van het Grieks voor hoorn (κέρας). Dit verwijst naar het project waar Keras initieel ontwikkeld was, ONEIROS. Oneiros zijn de droom geesten vermeld in de Odyssee van Homeros, die verdeeld waren tussen zij die mensen voorliegen, zij die door een ivoren poort de Aarde bereikten, zij die de toekomst aankondigden en zij die door een poort van hoorn (κέρας) arriveerden.

Keras richt op een zo vlot mogelijke *developer experience* door een aantal principes nauw na te leven tijdens het ontwikkelen en uitbreiden van hun API. Deze principes zijn de volgende [13]:

- **End-to-end gebruikerwerkstromen bewust ontwerpen:** het doel is dat de kernstructuren en -methodes zo dicht mogelijk bij de concepten van de gebruiker liggen. Bij neurale netwerken zijn dit bijvoorbeeld lagen, netwerken, activatiefuncties,… en niet de specifieke details van de interne implementatie. Dit wordt doorgetrokken naar het *onboarding* proces, waar het niet uitmaakt hoe de API geïmplementeerd is, maar het aanleren hoe de gebruiker deze API kan gebruiken om zijn problemen op te lossen. Er worden dus geen stukken van een werkstroom

ontwikkeld, maar er wordt gezorgd dat per aanspreekbare klasse of functie volledige functionaliteit beschikbaar is.

- **Cognitief gewicht voor de gebruikers verminderen:** dit is te bereiken door consistent te zijn in naamgevingspatronen, maar ook door het aantal 'nieuwe' concepten zo laag mogelijk te houden. Hiermee wordt bedoeld dat er één centraal begrip is waarrond de API zich aftakt, zodat de gebruiker zich gemakkelijker een mentaal model kan vormen van de API. Verder moet er een balans zijn tussen de hoeveelheid klassen en functies, en de parameters van deze onderdelen. Dit kan bereikt worden door deze structuren modulair te maken. Het volgende element bij de ontwikkeling van de API is dat alle elementen die geautomatiseerd kunnen worden, ook geautomatiseerd zijn. Dit ligt in lijn met het ontwerpen van de werkstromen. Als die eerste stap correct verloopt, dan is er al een groot deel van de *low-level* stappen geautomatiseerd. Als laatste is goede documentatie met veel voorbeelden een noodzaak.
- **Nuttige feedback voor gebruikers:** gebruikersfouten vroeg opvangen, en de meest voorkomende fouten anticiperen is de gouden regel. Verder wordt er gezorgd dat de foutberichten zo duidelijk mogelijk zijn door ze aan deze vier vragen te laten voldoen:
    - Wat is er gebeurD?
    - In welke context is het gebeurd?
    - Wat verwachtte de software?
    - Hoe kan de gebruiker het probleem oplossen?

Door aan deze principes te voldoen is Keras met 250.000 individuele gebruikers uitgegroeid tot het meest gebruikte framework in zowel de industrie als de researchcommunity na TensorFlow, en Keras is de officiële frontend van TensorFlow. [14]

2.2.3 Visual Object Tagging Tool

Een *supervised learning* model trainen vraagt een hoeveelheid gelabelde data. In het geval van een objectdetectie-netwerk, zijn dit afbeeldingen van de objecten, waar labels over geplaatst worden.

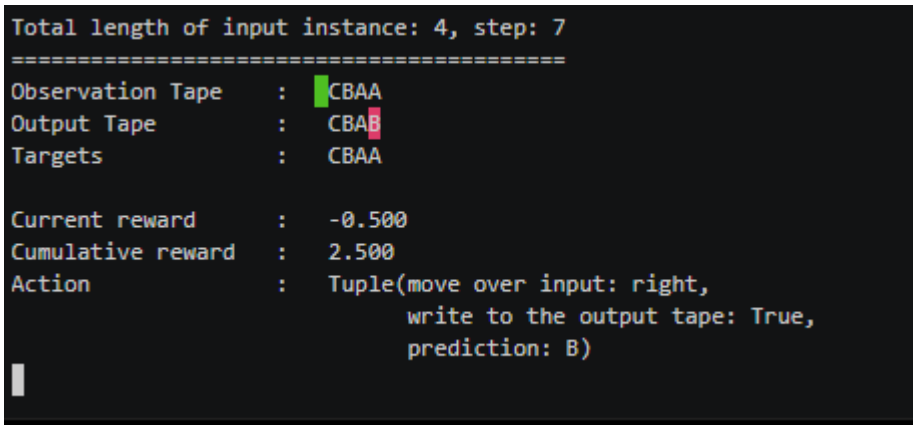
Om dit te doen, is er gekozen voor VoTT, of *Visual Object Tagging Tool*, een applicatie ontwikkeld door Microsoft om afbeeldingen te labelen. VoTT is platformonafhankelijk, wat zeer handig is aangezien de meeste AI-ontwikkeling plaatsvindt op Linux-systemen. Daarnaast laat VoTT toe om volledige video's te verwerken, waarbij de gebruiker aangeeft hoeveel frames of afbeeldingen per seconde video hij wil labelen. Dit zorgt ervoor dat het labelproces veel sneller verloopt dan wanneer individuele afbeeldingen gelabeld moeten worden.

2.2.4 OpenAI Gym

OpenAI Gym is een toolkit ontworpen in 2016 als poging om twee vertragingsfactoren in onderzoek van *reinforcement learning* op te lossen. Als eerste was er een nood aan betere vergelijkende testen. Bij *supervised learning* werd voortgang gedreven door de grote gelabelde sets die beschikbaar gesteld werden door bijvoorbeeld ImageNet. Het equivalent in *reinforcement learning* zou een grote en diverse set omgevingen zijn. De toen bestaande open-source omgevingen hadden te weinig variatie en waren zeer vaak moeilijk op te zetten en te gebruiken.

Het tweede probleem was het gebrek aan een standaardisering van omgevingen in publicaties. Kleine en subtiele verschillen in de definitie van het probleem, de beloningsfunctie, of de set van acties, kan een taak drastisch versimpelen of juist moeilijker maken. Dit maakte het extreem moeilijk om gepubliceerde resultaten te reproduceren of resultaten te vergelijken tussen verschillende papers.

De omgevingen in de OpenAI Gym zijn variabel en bevatten verschillende problemen, van simpele algoritmes leren imiteren zoals de Copy-v0 omgeving zichtbaar in Figuur 4, tot het aansturen van robotische handen in simulaties zoals in Figuur 5. Op deze omgevingen kunnen dan *reinforcement learning* algoritmes toegepast worden, en kunnen de resultaten binnen één enkele omgeving vergeleken worden.



Figuur 4 Copy-v0 Omgeving [52]

Figuur 5 Handreach-v0 Omgeving [33]

Zo kunnen de betere algoritmes gevonden worden per probleemsituatie, maar ook algemene verbeteringen in *reinforcement learning* zijn beter op te volgen. Om dit nog meer in de hand te zetten, voorziet OpenAI Gym een publieke wiki waarop een scoreboard wordt bijgehouden. Hier kunnen onderzoekers en ontwikkelaars hun resultaten vergelijken, hun eigen resultaten toevoegen, en linken naar hun code en video's van hun oplossingen.

2.2.5 GoPro HERO 5 Black

De GoPro HERO 5 Black is de gekozen camera waar de beelden voor de training van het CNN en de live beelden van een spel gemaakt worden. Er is gekozen voor een action camera omwille van het geringe gewicht en formaat van de dit type camera. Figuur 6 toont de GoPro.

Figuur 6 GoPro HERO 5 Black [34]

Een groot voordeel dat de GoPro heeft over zijn competitie is de *linear mode* of lineaire modus. Alle action camera's hebben een groothoeklens, wat voor vervorming zorgt in de beelden, een zogenoemd *fisheye* effect. In Figuur 7 kan het verschil gezien worden tussen deze modi van de camera.

Figuur 7 Wide en Linear modus van GoPro [35]

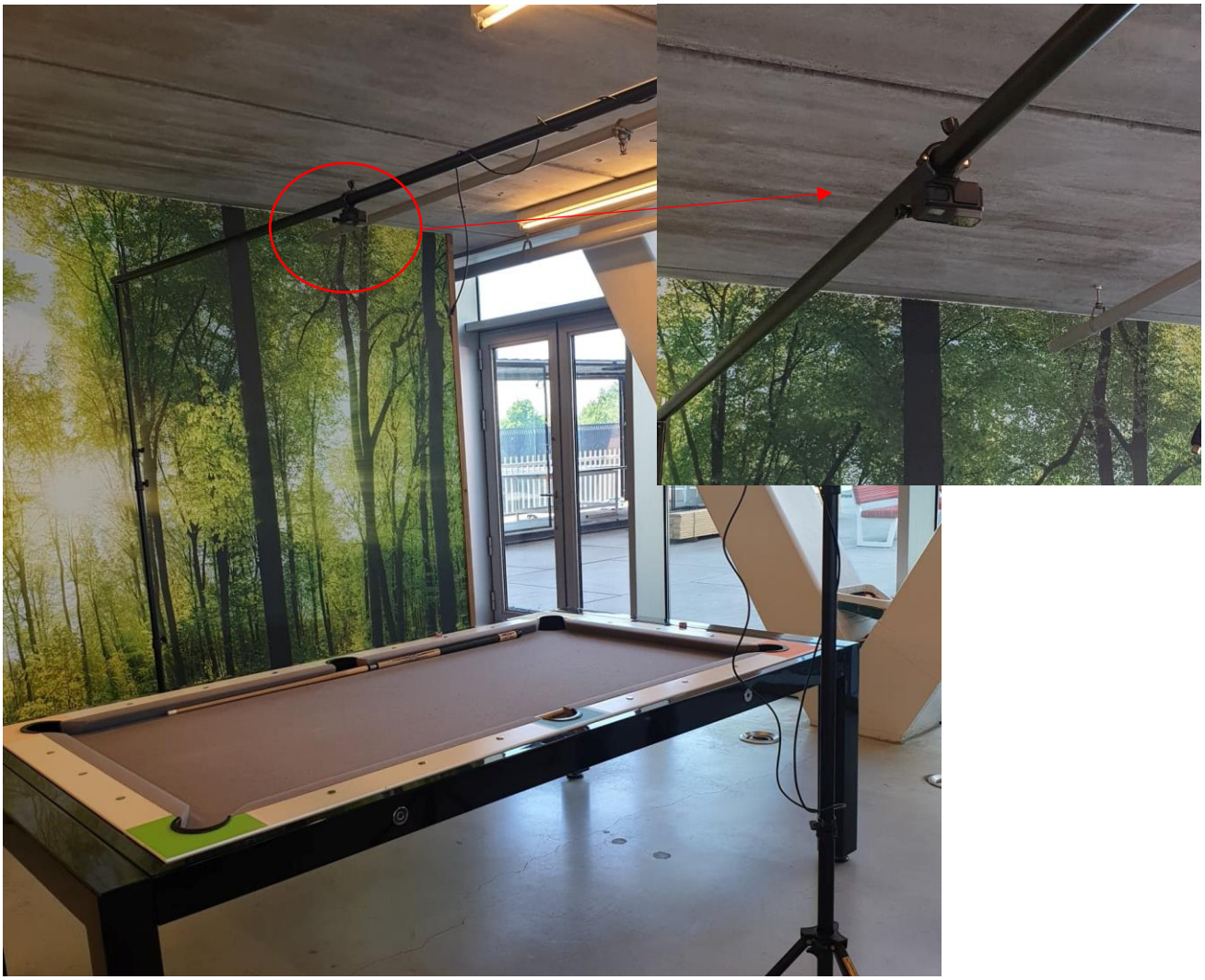
Het vlakke beeld dat de lineaire modus oplevert is handig voor object detectie, aangezien het de objecten niet vervormd. Daarnaast is het gemakkelijker om de posities van de ballen herkend door de object detectie om te zetten naar de *states* die nodig zullen zijn voor het *reinforcement learning* model.

Deze camera kunnen we rechtstreeks aan het pc-systeem verbinden via zijn micro-HDMI poort. Zo kunnen we de live beelden van de camera rechtstreeks voeden aan het object herkenningssysteem. Door zijn geringe formaat is het simpel om de camera te monteren boven een pooltafel. Om dit te bereiken is er een *handlebar mount* gebruikt, die bevestigd was aan een Linkstart BS-3031 achtergrondsysteem. In Tabel 1 zijn de specificaties van de HERO 5 Black, in zijn lineaire modus, te vinden. Figuur 8 toont de uiteindelijke opstelling van de camera.

Dimensies	62 x 45 x 33 mm

Gewicht	118 gram
Resolutie	2704 x 1520 @ 60 fps
Beeldratio	16:9
Data interface	USB 3.0
HDMI	Micro HDMI
Batterij levensduur	2 uur

Tabel 1 GoPro HERO 5 Black specificities



Figuur 8 Opstelling Pooltafel

2.2.6 NVIDIA GeForce RTX 2080 TI

Om de grote hoeveelheid afbeeldingen, en de videobeelden tijdens gebruik, te verwerken is er nood aan een sterke grafische kaart. De grafische kaart staat in voor de verwerking van alle grafische in- en output van een computer. Semantisch gezien is er een verschil tussen een grafische kaart en een *graphics processing unit*, maar deze distinctie wordt over het algemeen niet meer gemaakt. De GPU is een onderdeel van een grafische kaart, en het is in deze *processing unit* dat de berekeningen worden gemaakt.

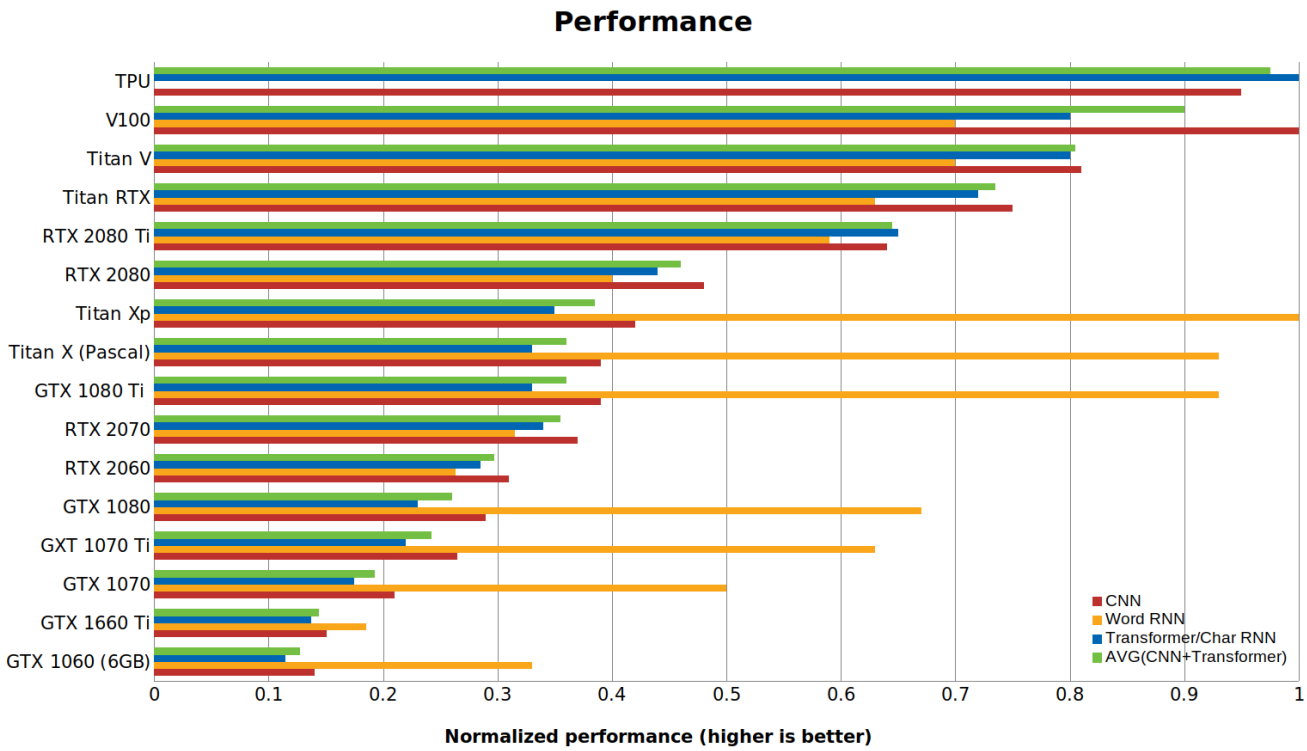
De voornaamste reden waarom de grafische kaart een belangrijke investering is, is dat deze zeer belangrijk is om een CNN, een neuraal netwerk dat afbeeldingen verwerkt, te kunnen trainen en gebruiken. Een GPU blinkt uit wanneer er grote hoeveelheden data verwerkt worden of als er veel berekeningen gemaakt moeten worden met grotere getallen, waar een CPU minder geschikt voor is.

Achter de schermen van *deep learning* en dus neurale netwerken, worden er vooral matrixvermenigvuldigingen gebruikt. De GPU is hier extreem voor geschikt, aangezien het ontwikkeld is om zo veel mogelijk parallelle berekeningen te maken. Om dit te bereiken maakt de GPU gebruik van zijn vele aanwezige kernen.

In het geval van een NVIDIA heten deze kernen CUDA *cores*. Het equivalent van deze kernen bij AMD heet *Stream processors*. Het verschil tussen deze twee ligt in de complexiteit van de kernen. CUDA *cores* zijn groter, complexer en gebruiken een veel hogere frequentie. Deze kernen hebben dus veel meer mogelijke doelen. Dit is ook de reden waarom er algemeen minder *cores* zitten op een NVIDIA GPU in vergelijking met een AMD GPU.

Stream processors zijn kleiner en simpeler, en hebben een lagere frequentie. Dit betekent dat de kaart meer optimalisatie uitvoert als hij de kernen aanstuurt, want elke kern heeft een specifiek doel, in tegenstelling tot een CUDA *core*. [15], [16]

NVIDIA staat veel verder dan AMD in de ondersteuning van AI-ontwikkelingen. CUDA is veel geschikter voor de training van AI en wordt gebruikt door de grootste namen in *deep learning* frameworks zoals TensorFlow en Pytorch.

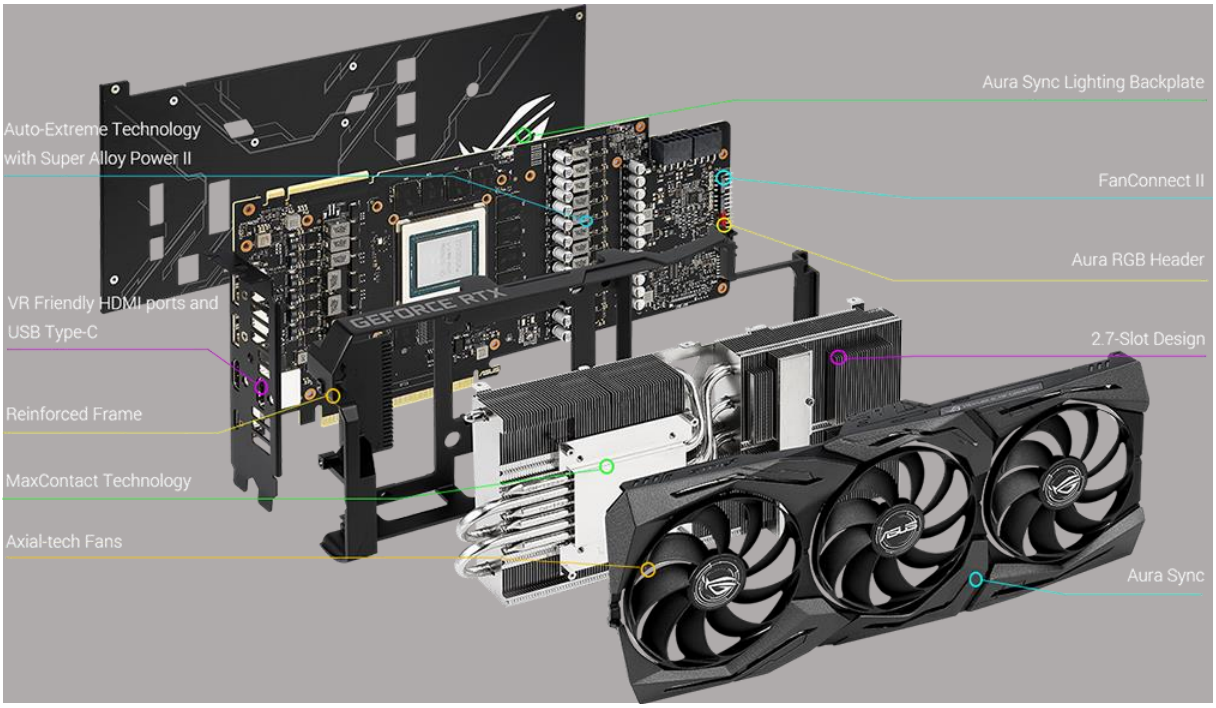


Figuur 9 Grafische kaart performantie in een aantal deep learning-toepassingen [17]

Figuur 9 toont een vergelijking van grafische kaarten in een aantal verschillende *deep learning*-toepassingen. Deze figuur toont aan dat de RTX 2080 Ti tot de top vijf behoort voor dit type toepassingen, en de hoogst gerangschikte *consumer* niveau-kaart is.

Titan RTX , Titan V en de V100 zijn zogenoemde *prosumer*-kaarten waar de winst aan processorkracht niet evenwichtig is ten opzichte van de prijs van de kaart. Indien prijs/kwaliteit de grootste factor is, raden de meeste bronnen de RTX 2070 aan. De RTX 2080 Ti levert echter de hoogste performantie in de huidige lijn grafische kaarten van NVIDIA. [17]

De RTX-lijn grafische kaarten hebben nog een extraatje op gebied van *machine learning*, namelijk de Tensor Cores. Dit type kernen zijn specifiek ontwikkeld voor de matrixberekeningen die zo belangrijk zijn binnen dit veld. Een GPU met dit type kernen is tot negen keer sneller in *mixed precision* matrixvermenigvuldigingen dan een vergelijkbare GPU van de vorige generatie. [18] Figuur 10 toont het gekozen model, de Asus GeForce RTX 2080 TI ROG-STRIX-RTX2080TI-O11G-GAMING.



Figuur 10 Asus GeForce RTX 2080 TI ROG-STRIX-RTX2080TI-O11G-GAMING [51]

3 Uitwerking stageopdracht

3.1 Inleiding

De stageopdracht bestaat uit drie grote fases. De eerste fase is de implementatie van objectdetectie, het herkennen en volgen van de poolballen tijdens het spel. Daarna wordt er programmatisch het spel gevolgd, en kunnen fouten of punten aangegeven worden via een interface. Bij de derde fase wordt een *reinforcement learning* algoritme gebruikt om een model te trainen, zodat deze hints kan leren geven.

Tijdens de uitwerking van deze opdracht is er gebruikt gemaakt van een Pop!_OS systeem. Pop!_OS is een aanpassing van een Ubuntu distributie beschikbaar in een LTS versie (18.04) of de nieuwere 19.04. Voor het project was er nood aan de 19.04 omdat er geen ondersteuning is voor RTX-kaarten in de 18.04. Alle commando's die aangegeven worden in dit onderzoek zijn gericht naar Debian gebaseerde Linux-systemen.

3.2 Voorbereidend werk

3.2.1 CUDA toolkit

Om van de volledige kracht van de GPU gebruik te kunnen maken moet de CUDA *toolkit* geïnstalleerd worden. Veel *machine learning* frameworks maken gebruik van de *toolkit* om hun functies versneld te doen verlopen. De eerste stap is het downloaden van de installatie van de website van NVIDIA. Op dit moment biedt NVIDIA geen versie aan voor Ubuntu 19.04 distributies maar de 18.10 versie werkt ook op een 19.04 distributie. Nadat er genavigeerd is naar de map waarin het bestand naar gedownload is, zijn de volgende commando's vereist om de CUDA *toolkit* te installeren:

```
$ sudo apt-get install build-essential dkms
$ sudo apt-get install freeglut3 freeglut3-dev libxi-dev libxmu-dev
$ sudo dpkg -i cuda-repo-<version>.deb
$ sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub
$ sudo apt-get update
$ sudo apt-get install cuda
```

'<version>' wordt vervangen met de specifieke versie die gedownload is. Voor het project is de laatste beschikbare versie gekozen namelijk 'ubuntu1810-10-1-local-10.1.105-418.39_1.0-1_amd64'. Na deze commando's en de computer opnieuw op te starten is de installatie van de CUDA *toolkit* voltooid.

3.2.2 cuDNN

cuDNN is een *GPU-accelerated library* van primitieven voor diepe neurale netwerken. Dit houdt in dat deze *library* geoptimaliseerde implementaties heeft van een aantal standaard routines die binnen neurale netwerken veel gebruikt worden. Voorbeelden hiervan zijn *convolution, pooling, normalization* en *activation* lagen.

Voordat cuDNN gedownload en geïnstalleerd kan worden, moet er een registratie gedaan worden op de *developer* website van NVIDIA. Als dit voltooid is, kan de correcte versie van cuDNN gedownload worden. Deze versie moet matchen met de geïnstalleerde versie van CUDA. In dit geval wordt de cuDNN v7.5.1 voor CUDA 10.1 gedownload. Er wordt uiteraard gekozen voor de Linux editie van de *library*. Er wordt niet gekozen voor de Debian versie, deze bestaat uit twee onderdelen en maakt de installatie moeilijker.

Als de installatie van CUDA correct verlopen is, zal deze zich bevinden in '/usr/local/cuda'. Dit kan gecontroleerd worden met dit commando:

```
$       whereis cuda
```

Als het bestand binnengehaald is, maakt het de installatie makkelijker door te navigeren naar de map waar dit bestand geplaatst is. Dan kunnen volgende commando's uitgevoerd worden om dit onderdeel correct te installeren:

```
$       tar -xzvf cudnn-<cuda_versie>-linux-x64-<cudnn_versie>.tgz
$       sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$       sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$       sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

Binnen dit project werd '<cuda_versie>' vervangen door '10.1' en 'cudnn_versie' met 'v7.5.1.10'. De installatie van cuDNN kan hierna geverifieerd worden met het volgende commando:

```
$       cat /usr/local/cuda/include/cudnn.h | grep CUDNN_MAJOR –A 2
```

3.2.3 OpenCV

OpenCV is een open source *library* die functioneel gericht is op het bereiken van real time *computer vision*. OpenCV is een noodzakelijke bibliotheek voor het verwerken van de live beelden die in het project gebruikt worden. Daarnaast wordt het gebruikt om de video's die gebruikt worden om de modellen te testen weg te schrijven, zodat het vergelijken van de verschillende modellen simpeler wordt.

Het installeren van OpenCV vraagt een lange reeks commando's. Hier wordt de lijst commando's opgebroken en wordt elk onderdeel toegelicht. De eerst stap is het zorgen dat de 'apt-get' pakket manager up-to-date is met de volgende twee commando's:

```
$       sudo apt-get update
$       sudo apt-get upgrade
```

Hierna moeten er enkele *developer tools* geïnstalleerd worden. Mogelijk zijn er al een aantal geïnstalleerd, maar alle noodzakelijke onderdelen worden hier bijgehaald zodat dit absoluut voor geen fouten meer kan zorgen.

```
$       sudo apt-get install build-essential cmake unzip pkg-config
```

De volgende stap is het installeren van enkele I/O-pakketten zodat OpenCV met de meeste standaard afbeeldingsformaten kan werken zoals JPEG en PNG. Hoewel het doel is om live video te gebruiken, zijn deze pakketten nodig voor het trainingsproces waar er wel afbeeldingen gebruikt worden.

```
$       sudo apt-get install libjpeg-dev libpng-dev libtiff-dev
```

Als dit gebeurd is, worden er ook enkele pakketten geïnstalleerd voor het verwerken van videobestanden, en video streams. Dit kan met de volgende commando's:

```
$       sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
```

```
$       sudo apt-get install libxvidcore-dev libx264-dev
```

Nu kunnen de ‘opencv’ en ‘opencv_contrib’ module afgehaald worden van hun officiële github releases. ‘3.4.4’ is de versie die binnen het project gebruikt is. Er kan een andere versie gekozen worden, maar de versie moet hetzelfde blijven over de twee commando’s. Deze twee commando’s kunnen wat tijd vragen, OpenCV is geen kleine *library*. De twee bestanden die er binnengehaald worden, worden meteen uitgepakt met de daaropvolgende twee commando’s:

```
$   wget -O opencv.zip https://github.com/opencv/opencv/archive/3.4.4.zip
$   wget -O opencv_contrib.zip https://github.com/opencv/opencv_contrib/archive/3.4.4.zip
$   unzip opencv.zip
$   unzip opencv_contrib.zip
```

De volgende stap is het opstellen van de OpenCV *build* via het ‘cmake’ commando. Er wordt eerst naar de ‘opencv’ map genavigeerd, waar een nieuwe map gemaakt wordt voor de *build*. Daarna komt het ‘cmake’ commando.

```
$       cd ~/opencv-3.4.4
$       mkdir build
$       cd build
$       cmake -D CMAKE_BUILD_TYPE=RELEASE \
                -D CMAKE_INSTALL_PREFIX=/usr/local \
                -D INSTALL_PYTHON_EXAMPLES=ON \
                -D INSTALL_C_EXAMPLES=OFF \
                -D OPENCV_ENABLE_NONFREE=ON \
                -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
                -D PYTHON_EXECUTABLE/usr/bin/python \
                -D BUILD_EXAMPLES=ON ..
```

Nadat dit voorbereidend commando is uitgevoerd, kan OpenCV gecompileerd worden. Hiervoor wordt het ‘make’ commando gebruikt. Verder wordt er de ‘-j8’ vlag gebruikt. Deze vlag versnelt de compilatie van OpenCV door er meerdere *threads* aan toe te kennen. Het getal duidt het aantal *threads* aan, dit kan dus veranderd worden naargelang de capaciteiten van de hardware die beschikbaar is. Aangeraden is dat dit getal nooit het aantal kernen dat de processor ter beschikking heeft overschrijft. Het commando is:

```
$       make –j8
```

Het compileren van OpenCV kan even duren, afhangende van de gebruikte hardware. Als de compilatie volledig voltooid is, kan OpenCV geïnstalleerd worden met de volgende commando’s:

```
$       sudo make install
$       sudo ldconfig
```

De installatie kan geverifieerd worden met dit laatste commando:

```
$       pkg-config --modversion opencv
```

3.2.4 VoTT

Deze applicatie is niet moeilijk te installeren, maar omdat we ondersteuning nodig hebben voor het YOLO framework, wordt versie 1.7.1 gebruikt. Versie 2 ondersteunt YOLO niet meer als een output formaat.

```
$       wget https://github.com/Microsoft/VoTT/releases/download/v1.7.1/vott-linux.snap
$       snap install –dangerous vott-linux.snap
```

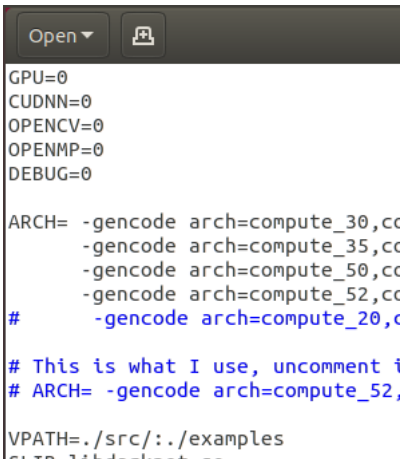
Met deze twee commando's is VoTT volledig geïnstalleerd.

3.2.5 YOLOv3 Tiny

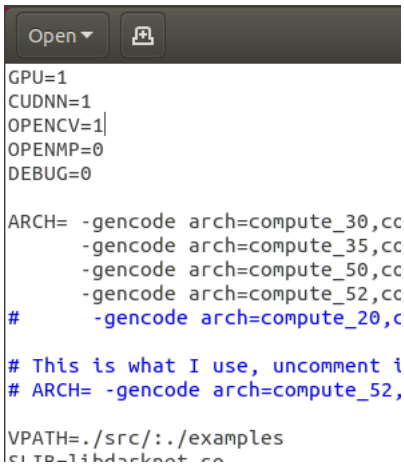
YOLOv3 Tiny is het CNN dat gebruikt wordt om de object detectie uit te voeren in het project. De eerste stap van de installatie is de git *repository* binnen te halen met het volgende commando:

```
$       git clone https://github.com/pjreddie/darknet
```

Hierna moeten er een aantal aanpassingen gemaakt worden in de 'Makefile' die zich bevindt in de 'darknet' folder. Hierin moeten drie lijnen aangepast worden. GPU mag op 1 gezet worden indien er een CUDA-installatie aanwezig is. CUDNN mag op 1 gezet worden als cuDNN geïnstalleerd is, en OPENCV mag op 1 gezet worden indien OpenCV aanwezig is op het systeem. Figuur 12 toont deze 'Makefile' voor de aanpassingen. Figuur 11 toont de aangepaste lijnen in de 'Makefile'.



Figuur 12 Makefile voor aanpassing



Figuur 11 Makefile na aanpassing

Als de 'Makefile' aangepast is, kan het 'make' commando uitgevoerd worden in de 'darknet' folder. Voor deze 'make' zijn er geen extra *threads* nodig door de kleiner grootte. Het commando is dus zeer simpel:

```
$       make
```

3.2.6 OpenAI Gym

OpenAI Gym, het framework voor het trainen van *reinforcement learning agents*, kan geïnstalleerd worden met één simpel commando.

```
$       pip install gym
```

Met dit commando wordt OpenAI Gym geïnstalleerd binnen een Python omgeving, of globaal afhangende van waar het commando is uitgevoerd. Binnen het project is er gewerkt met een omgeving die niet standaard in OpenAI Gym vervat zit. In onderdeel 3.4.1 kan gevonden worden hoe deze omgeving gebouwd is. De standaard OpenAI Gym is nodig tijdens de ontwikkeling van nieuwe omgevingen, en tijdens het gebruik van eender welke omgeving.

3.2.7 TensorFlow

TensorFlow is de backend die Keras zal aanspreken om de neurale netwerken in op te bouwen. Dit framework kan ook met een simpel pip-commando geïnstalleerd worden. Er is gekozen om gebruik te maken van de standaard TensorFlow installatie, en niet de GPU-editie. Die keuze is gemaakt om conflicten te voorkomen met de aanwezige CUDA en cuDNN installaties. Het commando is het volgende:

```
$ pip install tensorflow
```

3.2.8 Keras

Keras en de extensie ‘keras-rl’ zijn ook beschikbaar via PyPI, of een pip-commando. Om deze twee pakketten te installeren worden de volgende commando’s uitgevoerd:

```
$ pip install keras
$ pip install keras-rl
```

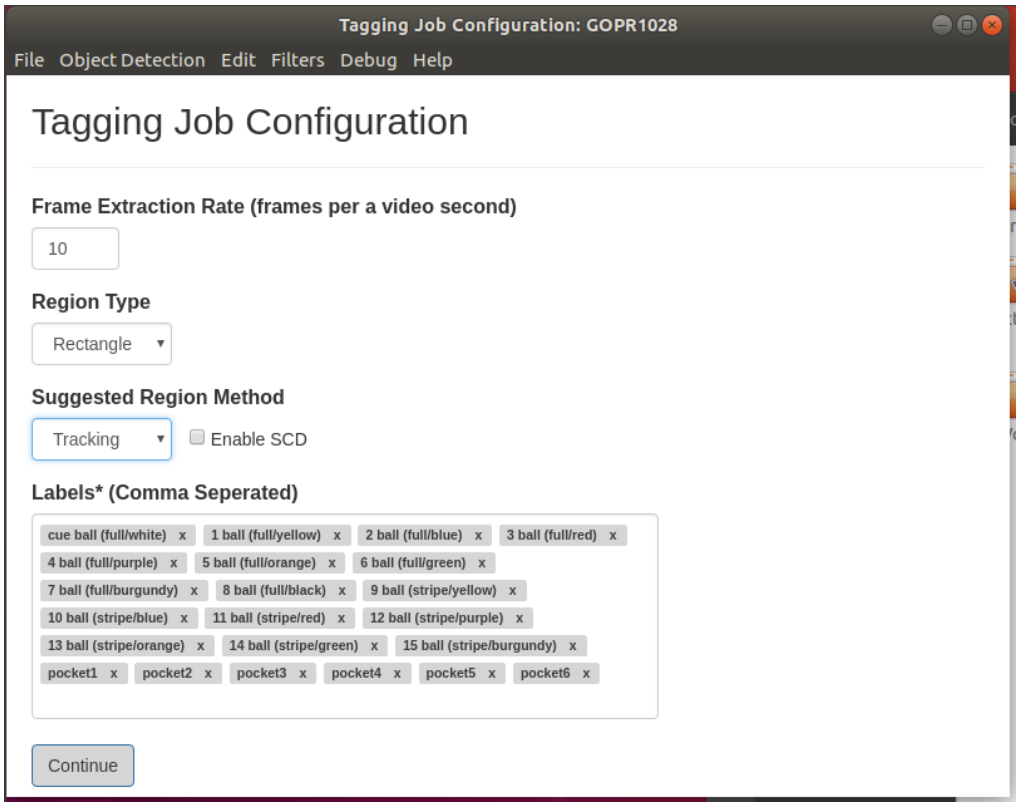
3.3 Object detectie

3.3.1 Dataset creëren

De eerste fase van het project was het herkennen en volgen van de poolballen. Objectdetectie is een onderdeel van *supervised learning.* Dit betekent dat de eerste stap het creëren van een dataset is. Voor het maken van deze dataset zijn er eerst een aantal video's gemaakt met de GoPro. Enkele video's waar de ballen één voor één over de tafel gerold werden, en video's van de poolspelletjes die door de medewerkers van Bewire gespeeld werden.

Om deze video's te labelen, wordt VoTT gebruikt. Bij het opstarten van het programma krijg je de optie om te kiezen tussen een map met afbeeldingen of een video. Nadat er een video geselecteerd is, zijn er twee instellingen die belangrijk zijn. Allereerst moet het aantal frames dat er per seconde video geselecteerd wordt ingevuld worden. Standaard staat dit op 10.

Daarnaast moeten ook de labels ingevuld worden, of met andere woorden, de namen van de objecten die je wilt herkennen. In Figuur 13 kan dit scherm gezien worden.



Figuur 13 VoTT video labeling configuratiescherm

Dan volgt het labelen van elk frame dat uit de video werd gehaald. Niet alleen de ballen worden gelabeld, maar elke pocket krijgt zijn eigen label. Weten in welke pocket een bal gespeeld wordt, is belangrijk voor sommige regelsets. Om de pockets te kunnen differentiëren zijn er gekleurde papieren rond de gaten geplakt. In Figuur 14 is het labelproces te zien.

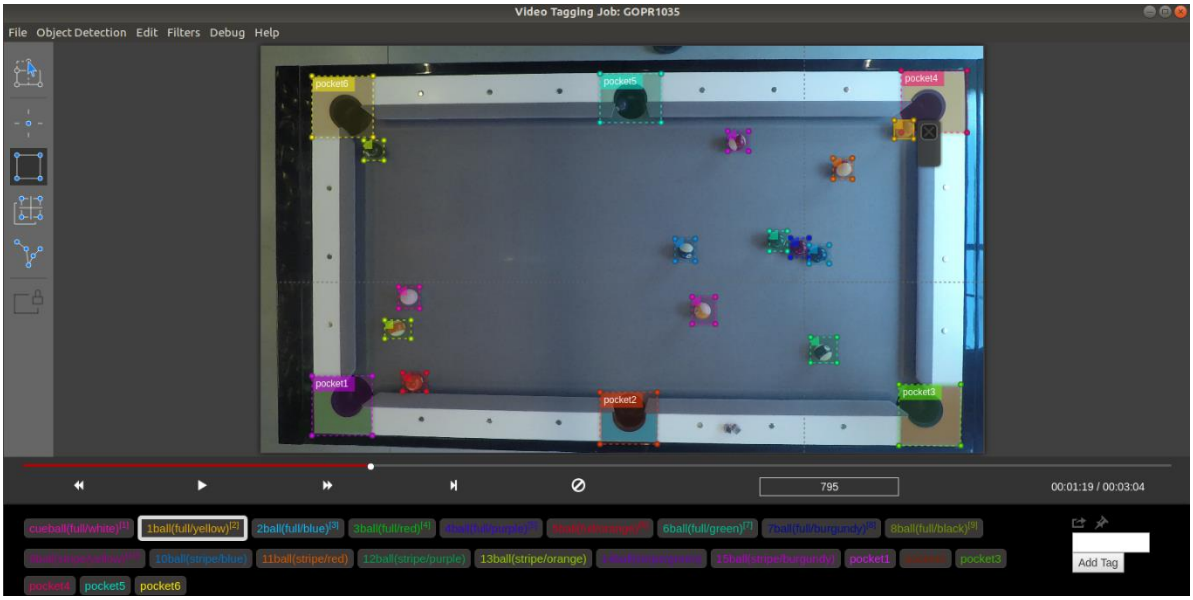
Als elk frame eenmaal gelabeld is, kunnen de labels geëxporteerd worden. De export bestaat uit een aantal bestanden. Elk frame wordt als een afbeelding opgeslagen. Bij elk van deze afbeeldingen hoort een tekstbestand. Dit tekstbestand bestaat uit evenveel lijnen als dat er objecten in dat frame gelabeld zijn. Deze lijnen bestaan uit vijf getallen. Het eerste getal is het klasse nummer. Dit verwijst naar de naam van het object in 'obj.names' een ander bestand dat zo dadelijk besproken wordt. De vier andere waardes geven de dimensies van het label weer ten opzichte van de afmetingen van de afbeelding waarin het label getekend staat. Een voorbeeld van deze lijnen:

```
18 0.924135 0.885194 0.083980 0.149310
```

Daarnaast genereert het programma een aantal configuratiebestanden die gebruikt kunnen worden om het YOLO-netwerk te trainen. Allereerst is er het 'yolo-obj.cfg' bestand. Dit configuratiebestand geeft weer hoe het YOLO-netwerk opgebouwd is en hoe deze omgaat met de afbeeldingen die ingevoerd worden. Dit bestand wordt niet gebruikt in het project, want het baseert zich op het YOLOv2-netwerk en niet het YOLOv3 Tiny netwerk dat hier gebruikt wordt.

Volgend zijn er het 'obj.data' en 'obj.names' bestanden. 'obj.data' duidt aan waar YOLO een aantal bestanden kan vinden dat het nodig heeft, alsook waar de gewichten opgeslagen moeten worden. Daarnaast bevat het ook het aantal klassen die gebruikt worden.

'obj.names' bevat de volledige lijst labels. Het is naar dit bestand dat het klassenummer verwijst om het juiste label aan het object te kunnen hangen.



Figuur 14 VoTT scherm gedurende label proces

De allerlaatste bestanden zijn de 'train.txt' en 'test.txt' bestanden, die een lijst van locaties naar afbeeldingen bevatten. Deze twee bestanden worden in het project niet gebruikt omdat er data van verschillende video's gebruikt wordt, dus deze splitsing in trainingsdata en testdata moet op de volledige dataset gebeuren.

Als alle video's geannoteerd zijn wordt een script, te vinden in Bijlage G: datasplitter.py, gebruikt om alle afbeeldingen die aanwezig zijn in de dataset te splitsen over de 'train.txt' en 'test.txt' bestanden. Tien procent van de data wordt gehouden als test data. In alle bestanden die er gebruikt worden in dit project, wordt 'obj' vervangen door 'pool'.

3.3.2 Trainen model

Met de gecreëerde dataset kan het model getraind worden. Er worden echter nog een paar aanpassingen aan 'darknet' gemaakt om het iets simpeler te maken om de beste resultaten te behalen. 'darknet' is een open-source neuraal netwerk framework geschreven in C en CUDA. Binnen dit framework is YOLO gecreëerd.

Het 'detector.c' bestand wordt aangepast zodat de resultaten van elke iteratie uitgeprint worden naar een tekstbestand. Een tweede aanpassing zorgt ervoor dat de gewichten van het netwerk elke 100 iteraties opgeslagen worden. Doordat deze gewichten bijgehouden worden kan *overfitting* en *underfitting* bestreden worden en kan het netwerk trainen zonder dat één persoon klaar moet staan om het trainen stop te zetten op het goede moment. In Bijlage H: detector.c train_detector aanpassingen kan de veranderde code gevonden worden.

Daarna werd er nog een aanpassing gemaakt aan 'image_opencv.cpp', dit laat toe dat er video's kunnen gebruikt worden om te testen en dat de resultaten opgeslagen worden in een nieuwe video. In 'bijlage I: image_opencv.cpp show_image_cv aanpassingen' kan de aangepaste code gevonden worden.

Na deze aanpassingen kan de training bijna gestart worden. Allereerst hebben we een set voorgetrainde gewichten voor de eerste paar lagen nodig. Om deze te verkrijgen worden de standaardgewichten voor YOLOv3 Tiny opgehaald, en halen hier deze gewichten uit. [19]

```
$       wget https://pjreddie.com/media/files/yolov3-tiny.weights
$       ./darknet partial cfg/yolov3-tiny.cfg yolov3-tiny.weights yolov3-tiny.conv.15 15
```

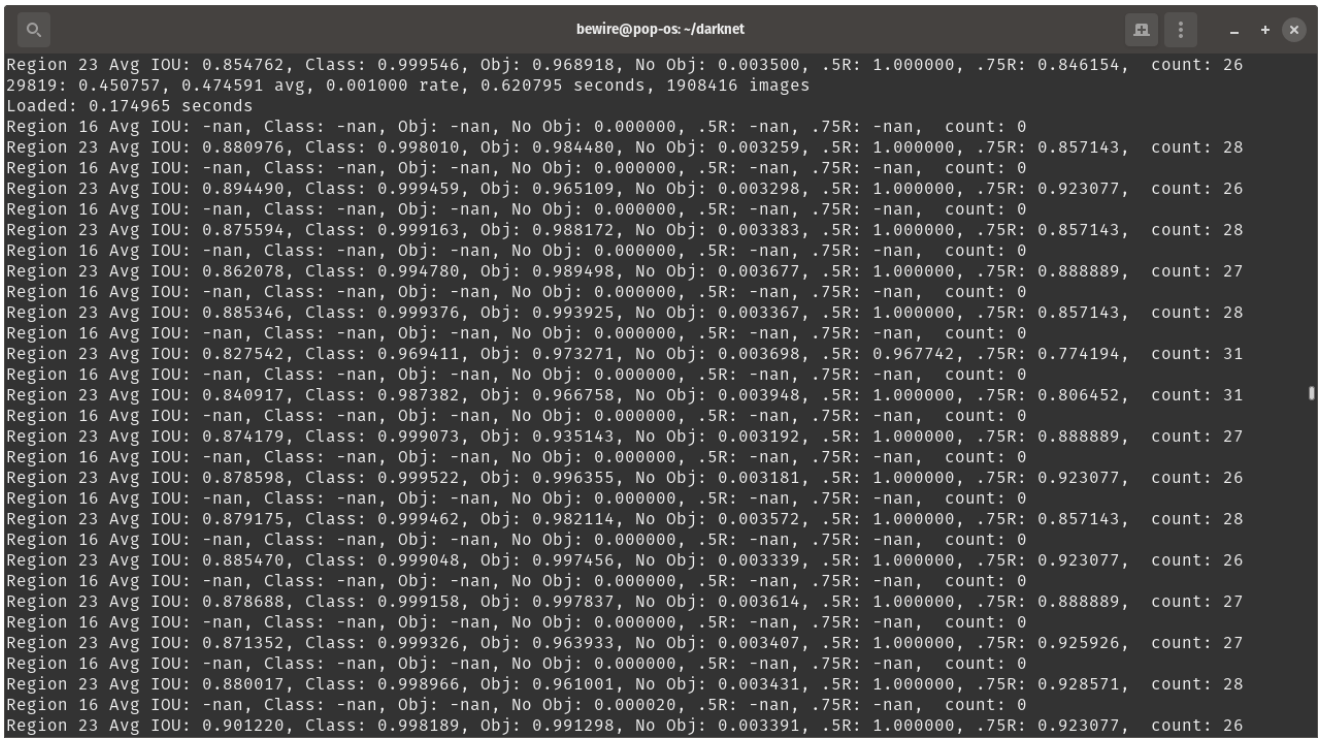
Dan moet er nog één configuratiebestand aangemaakt worden. Er wordt een kopie gemaakt van 'yolov3-tiny.cfg'. In dit project kiezen we om deze kopie 'yolov3-tiny-pool.cfg' te noemen. In dit kopie worden de volgende aanpassingen gemaakt:

- In elke '[yolo]' laag moet de 'classes' variabele veranderd worden naar het aantal klassen dat het model moet herkennen, in dit geval 22, 16 ballen en 6 gaten.
- In de '[convolutional]' laag voor elke '[yolo]' laag moet de 'filters' variabele naar een waarde gelijk aan (aantal klassen + 5) * 3 veranderd worden. In dit geval dus (22 + 5) * 3 = 81.

Meer informatie en exactere details over de lagen in het YOLOv3 model is te vinden in het eindwerk "Smart Pool Table: Deep learning en reinforcement learning vergelijken" van Yente Martens [20]. Met deze aanpassingen achter de rug kan het trainingsproces gestart worden met het volgende commando:

```
$       ./darknet detector train data/pool.data cfg/yolov3-tiny-pool.cfg yolov3-tiny.conv.15
```

Er verschijnt nu een heleboel output in de *terminal*. Voorbeeld output kan gevonden worden in Figuur 15.



Figuur 15 Terminal output tijdens het leerproces van het YOLO netwerk

De belangrijkste lijn in Figuur 15 is de tweede. Hieronder kort wat deze waardes betekenen:

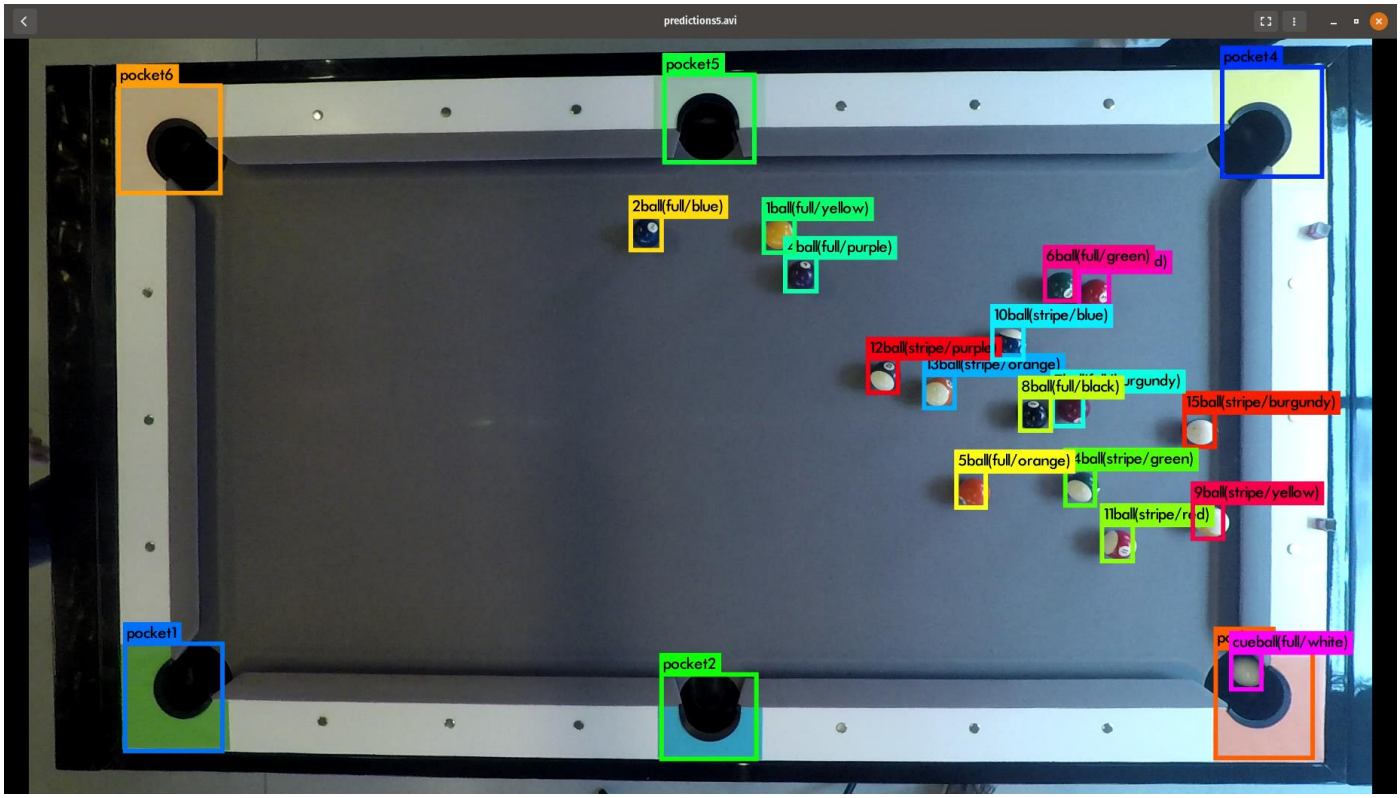
- **29819** is het getal dat aangeeft aan de hoeveelste iteratie het model zit.
- **0.450757** is het totale verlies van het leerproces.
- **0.474591** avg is het gemiddelde verlies, dit moet zo laag mogelijk worden.
- **0.001000** rate geeft de ingestelde leergraad aan.
- **0.620795** seconds geeft weer hoe lang het verwerken van de vorige batch in beslag nam.
- **1908416** images is het totaal van verwerkte afbeeldingen tijdens het leerproces

Aangeraden is dat er per klasse in de dataset 2000 trainingsiteraties worden gedaan. In het geval van dit project zijn dat er dus 44000. Om de gewichten te vinden met het laagste gemiddelde verlies, wordt er gebruik gemaakt van een script, te vinden in Bijlage J: ReadAndWrite.py, die de output gegenereerd tijdens het trainingsproces filtert op honderdtallen en in volgorde van goed naar slecht.

De gewichten kunnen dan op proef gesteld worden met het volgende commando, waar zowel een afbeelding als een video aan kan meegegeven worden.

```
$ ./darknet detector demo data/pool.data cfg/yolov3-tiny-pool.cfg backup/yolov3-tiny_40600.weights ~/Videos/ONE_MINUTE_GAME_FOOTAGE.MP4
```

Figuur 16 toont hoe de herkenning er in actie uit ziet.



Figuur 16 Object detectie model in actie

3.4 Reinforcement learning

3.4.1 Pool omgeving

Het is aangeraden om eerst het tweede onderdeel van dit eindwerk te lezen. Vanaf dit gedeelte zullen er verschillende termen gebruikt worden die hier niet worden uitgelegd om herhaling te voorkomen.

Om een *reinforcement learning agent* te kunnen trainen hebben we nood aan een omgeving waarin de agent acties kan uitvoeren. Standaard zit er geen pool omgeving in OpenAI Gym, dus we zullen er zelf een moeten maken. Om dit proces te versnellen is er de keuze gemaakt om een bestaande pygame om te vormen tot de omgeving. Het gekozen spel is ontwikkeld door Max Kovalovs, en is open-source beschikbaar op GitHub. [21]

Om een eigen Gym omgeving te maken moet deze een bepaalde structuur aanhouden. In het geval van dit project viel de structuur als volgt uit:

- gym-pool/
    - setup.py
    - gym_pool/
        - __init__.py
        - envs/
            - __init__.py
            - pool_env.py

Optioneel kan er in de 'gym-pool' map nog een *readme* en een *license* geplaatst worden. Het 'setup.py' bestand bevat de verschillende pakketten die de omgeving nodig zal hebben om te kunnen functioneren. In het geval van de 'gym-pool' omgeving zijn dat het standaard 'gym' pakket, de nieuwe omgeving gaat immers bepaalde kwaliteiten overerven van de algemene 'Env' superklasse waar alle omgevingen op gebaseerd zijn.

Verder is er het 'pygame' pakket, dat natuurlijk noodzakelijk is voor de functionaliteit van de pygame. Dan zijn er nog twee ondersteunende pakketten, namelijk het zeer bekende 'numpy' pakket, en het 'zope.event' pakket.

Dit wordt op de volgende manier in code beschreven:

```
from setuptools import setup

setup(name='gym_pool', version='0.1', install_requires=['gym','numpy','pygame',
'zope.event'])
```

Het '__init__.py' bestand dat zich in de 'gym_pool' map bevindt bevat de registratie van de omgeving. Pas als er een registratie van een omgeving is gebeurd, kan deze opgeroepen worden met de 'gym.make()' methode. Om de omgeving te registreren moet er een 'id' meegegeven worden en een 'entry-point'. De 'id' is de waarde die meegegeven wordt aan de 'make' methode om te bepalen welke omgeving opgestart moet worden. Het 'entry-point' duidt aan welke klasse de omgeving bevat die moet opgeroepen worden. In code is dit simpelweg:

```
from gym.envs.registration import register

register(id='Pool-v0', entry_point='gym_pool.envs:PoolEnv',)
```

Het is mogelijk meerdere omgevingen te registreren vanuit één enkel pakket door de register functie meerdermaal op te roepen, met verschillende 'id' waardes.

Het '__init__.py' bestand dat een laag dieper zit, in de 'envs' map, bevat enkel 'import' statements voor de omgeving. Dit ziet er als volgt uit:

```
from gym_pool.envs.pool_env import PoolEnv
```

Het belangrijkste bestand is het 'pool_env.py' bestand waar de omgeving zich in bevindt. Om de omgeving te maken moet de nieuwe omgeving allereerst overerven van de bestaande 'gym.Env' klasse.

Dan moeten er vier methodes geïmplementeerd worden, '__init__', 'reset', 'step', 'render'. De '__init__' methode wordt opgeroepen als de omgeving gecreëerd wordt met de 'gym.make()' methode. Deze methode zorgt ervoor dat er een aantal variabelen ingevuld zijn die de structuur van de omgeving duidelijk maken aan de gebruiker. Hieronder behoren de 'action_space' en 'observation_space' variabelen bijvoorbeeld. Deze twee *spaces* worden gebruikt om input en output van het *reinforcement learning* algoritme te bepalen. In het geval van dit project de input en output van het *actor* netwerk in het DDPG algoritme. Dit algoritme wordt uitgebreid uitgelegd in de het onderzoeksgedeelte, in onderdeel 3.2.4.

De 'reset' methode reset de omgeving als er een terminale *state* wordt bereikt, of als een aflevering afgelopen is. Hier worden er nieuwe startwaardes gegenereerd of de originele startwaardes terug geplaatst. Deze methode geeft ook meteen de eerste *state* van de volgende aflevering mee.

De 'step' methode heeft als parameter de actie die uitgevoerd moet worden. De vorm van deze actie wordt bepaald door de 'action_space' variabele. Deze methode voert de actie in op de omgeving, en observeert de volgende *state*. Deze geeft de methode terug, samen met de beloning voor de actie, of er een terminale *state* bereikt is of niet, en een lijst met extra info die de gebruiker mogelijk nodig heeft. Deze methode wordt het meest gebruikt.

De 'render' methode doet exact wat hij zegt, het maken van de visuele of tekstuele representatie van de omgeving. Deze methode wordt typisch opgeroepen binnen de 'step' methode.

Bij de implementatie van deze methodes werd rekening gehouden met de beschrijving van pool als een MDP. Dit MDP ziet er als volgt uit:

$$S = [(x_1, y_1), (x_n, y_n), \dots, (x_{16}, y_{16})] | x \epsilon [0{,}1000], y \epsilon [0{,}500]$$

$$A = \theta \epsilon [-\pi, \pi]$$

$$R = -1 + \big(n(s) - n(s')\big)$$

Vergelijking 1 MDP Pool Omgeving

Origineel was het plan om de ook rekening te houden met kracht bij het uitvoeren van acties. Ook de beloning werd eerst niet op deze manier uitgedrukt. Het idee was dat het *reinforcement learning* algoritme de regels van pool zou kunnen leren.

Hier is op geëxperimenteerd, maar het blijkt dat het algoritme zo'n ingewikkelde omgeving niet geconvergeerd krijgt. Langzaam werd de complexiteit van de omgeving afgebouwd. Allereerst werd de kracht uit de *action space* gehaald. Omdat de kracht en de hoek waarop gespeeld moest worden door hetzelfde netwerk gegenereerd werd, moest er normalisatie plaatsvinden. Dit gekoppeld met het

feit de *agent* niet kon weten welke waardes welke resultaten veroorzaakten, maakte convergentie in afzienbare tijd onmogelijk.

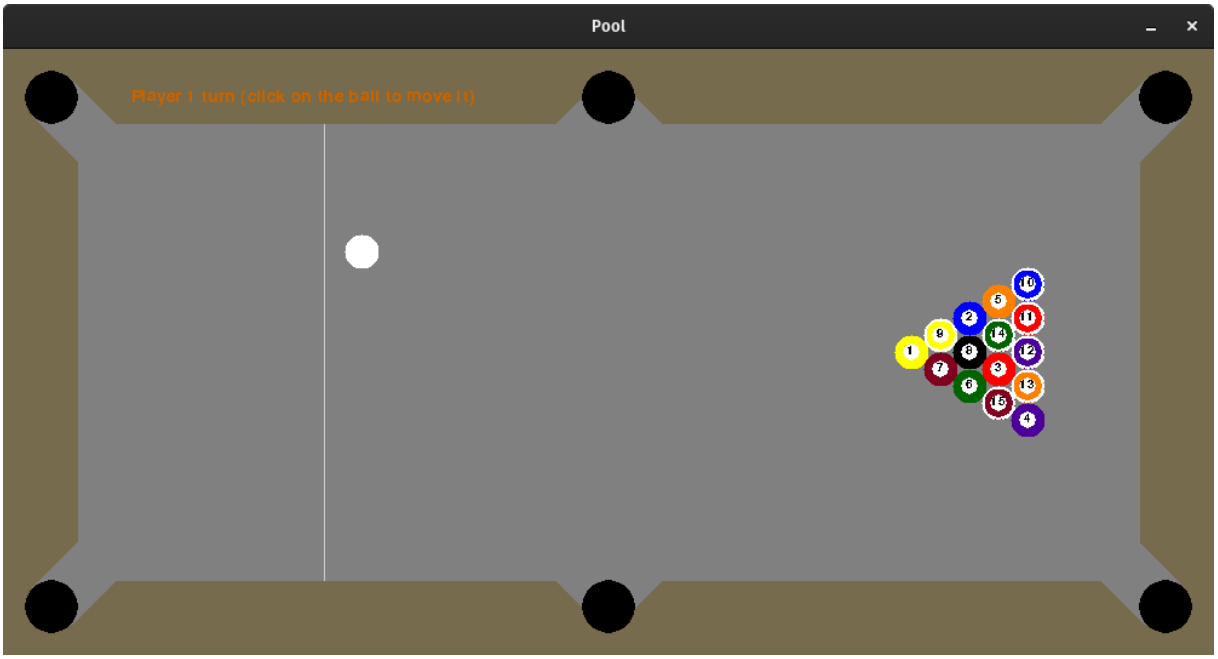
Hierna werd het beloningsysteem versimpel naar de formule voor R die te vinden is in Vergelijking 1. De originele manier waarop de beloning werd bepaald, volgens de regels van pool, bleek te complex voor de *agent*. Dezelfde actie, voor dezelfde speler, kon een totaal ander resultaat teweegbrengen gebaseerd op de vorige *states*. Er ging dus informatie verloren tussen de *states* in waar het model niet op kon reageren. Dit spreekt trouwens de Markov eigenschap van een MDP tegen.

Met deze aanpassingen was convergentie echter nog steeds niet gegarandeerd. Er werd een simpelere omgeving ontworpen, waar slechts twee ballen gebruikt werden om te kijken of convergentie überhaupt mogelijk was.

Als de omgevingen volledig klaar zijn, moet het pakket dat hier gecreëerd is nog geïnstalleerd worden. Dit wordt gedaan door in de map het volgende commando uit te voeren:

```
pip install –e .
```

In Figuur 17 kan de uiteindelijke omgeving gezien worden.



Figuur 17 Pool-v0 Omgeving

3.4.2 Agent trainen

De *agent* waar gebruik van gemaakt wordt binnen het project is de *wide* agent, te vinden in Bijlage C. De extra neuronen helpen met het convergeren van de gecompliceerde omgeving. Om de agent te trainen is er een python script gebruikt.

Allereerst wordt de agent geïmporteerd, samen met de OpenAI Gym

```
From DDPG_wide import DDPGAgentWide
import gym
```

Als het pakket gecreëerd in de vorige stap correct geïnstalleerd is kan daarna de omgeving gemaakt worden. Hierna worden de 'action_space' en 'observation_space' variabelen uit de omgeving getrokken om deze aan de constructeur van de *agent* te geven.

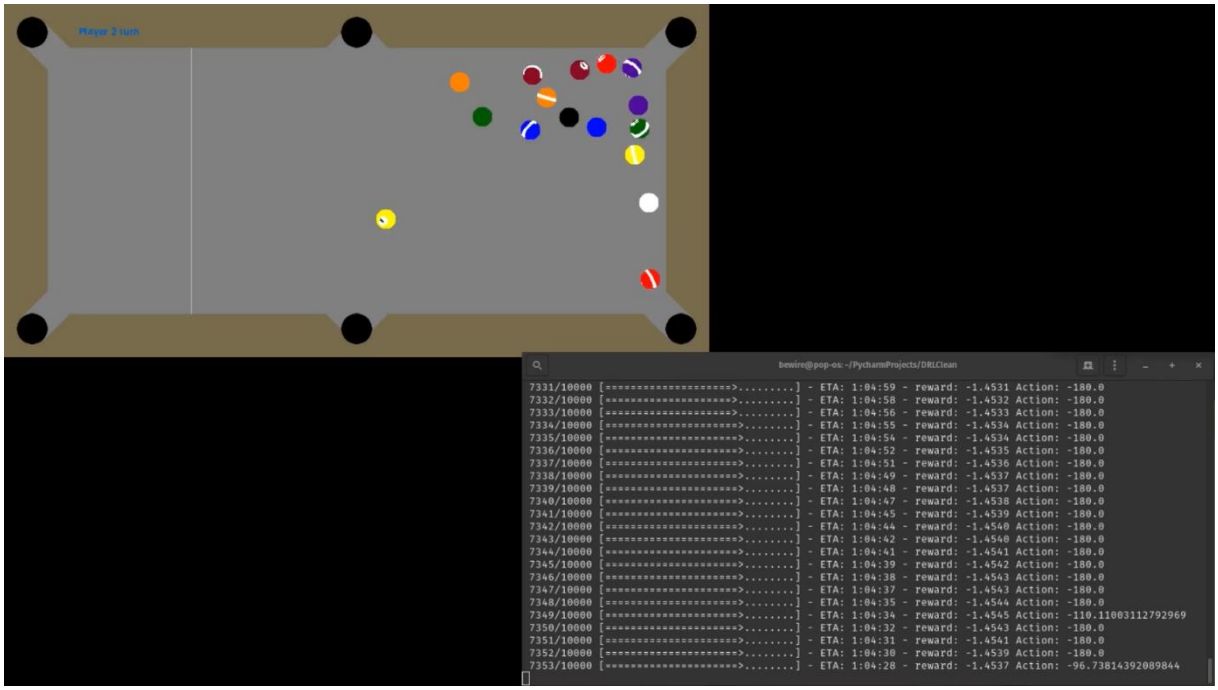
```
env = gym.make('pool-v0')

observation_space = env.observation_space
action_space = env.action_space

agent = DDPGAgentWide(observation_space=observation_space,
action_space=action_space)
```

Met dit compleet kan de 'fit' methode opgeroepen worden. Deze methode handelt het trainen van de agent af. Het resultaat van deze methode vangen we op als een object. Hier is er gekozen om dit object 'history' te noemen. Uit dit object kunnen allerhande statistieken over het trainingsproces getrokken worden. 'nb_steps' bepaal hoeveel keer de 'step' methode uitgevoerd zal worden, en de 'nb_max_episode_steps' bepaalt hoeveel stappen er maximum per aflevering genomen mogen worden.

```
history = agent.ddpg.fit(env, nb_steps=100000, nb_max_episode_steps=200, verbose=1)
```



Figuur 18 Agent in training. De omgeving is waarin de agent traint is zichtbaar, en in de terminal zijn een aantal gegevens over het trainingsproces zichtbaar.

In Figuur 18 is de agent te zien tijdens het trainingsproces. Als de agent getraind is worden de gewichten voor beide netwerken opgeslagen. Als de *agent* gebruikt wil worden, kunnen deze gewichten ingeladen worden.

```
agent.ddpg.save_weights('ddpg_agent_pool_weights.h5f', overwrite=True)
```

Een laatste stap is het bekijken van de resultaten van de training met de test methode. Ook deze methode genereert een object met statistieken in.

```
test_history = agent.ddpg.test(env, nb_episodes=10, nb_max_episode_steps=100)
```

Geen van de omgevingen en modellen convergeerden. Dit kan de volgende redenen hebben:

- Trainingstijd: De hoeveelheid trainbare parameters die de *wide* agent bezit zouden meer trainingstijd nodig kunnen hebben om tot een degelijk resultaat te komen. Uit experimenten in het research gedeelte blijkt echter dat de *wide agent* niet zoveel trager goede resultaten bereikt als andere agents.
- Omgeving: De gebruikte omgeving is gebaseerd op een 'pygame' gecreëerd door iemand niet gerelateerd aan het project. Zowel de omgeving zelf, als de aanpassingen in de omgeving om er een gym omgeving van te maken kunnen fouten bevatten die de resultaten beïnvloeden.
- DDPG: Het *deep deterministic policy gradient* algoritme is niet geschikt voor deze probleemstelling, en een ander algoritme biedt mogelijk wel resultaten.
- Niet mogelijk met de huidige staat van AI: omgaan met continue *state* en *action spaces* met *reinforcement learning* algoritmes zijn relatief nieuwe technieken. Het is mogelijk dat deze algoritmes nog niet ver genoeg staan om een resultaat te bereiken voor dit type omgeving in het heden.

3.5 Poolspel volgen

Om het poolspel te kunnen volgen worden er enkele verdere aanpassingen in 'darknet' gemaakt. Er was immers de nood om te weten waar de ballen en gaten zich bevonden, en de labels op de video zijn niet genoeg om deze informatie op een degelijke manier te bepalen. In het 'image.c' bestand werden enkele lijnen code aangepast.

In de 'draw_detections' methode worden twee prints toegevoegd. De eerste print is simpelweg om te duiden wanneer de verwerking van een nieuw frame begint. Deze print geeft aan wanneer het object detectie model aan een nieuwe observatie begint.

De tweede print geeft per herkend object een aantal waardes van de box die rond het object getekend wordt. Deze waardes geven de locatie van elke box aan, en ook het label geassocieerd met die box. Verder worden ook de breedte en hoogte van het originele frame aan. Deze waardes kunnen gebruikt worden om de locaties om te zetten naar de waardes waar de *reinforcement learning* agent op getraind is.

Verder geeft het de laagste en hoogste x en y waardes. Deze waardes kunnen samengevoegd worden om het coördinaat van de hoek linksboven en rechtsonder bepalen. Met deze coördinaten kan het overlappend gebied tussen twee verschillende labels berekend worden, en dus bepaald worden of een bal een andere bal raakt, of dat de bal zich in een pocket bevindt..

Totdat de eerste print terug naar het script geduwd wordt, worden alle andere lijnen opgeslagen in een lijst. Deze lijst bevat de volledige *state* van het spel op dat moment en het is deze lijst waaruit de eigenlijke observatie wordt gehaald.

De eerste keer dat een *state* geanalyseerd wordt, worden er vaste locaties voor de pockets ingesteld. Dit wordt maar eenmaal gedaan, want object detectie werkt niet altijd perfect, en door deze locaties eenmaal in te stellen kan er herkend worden wanneer een bal zich in een pocket bevindt, zelfs als de pocket niet langer herkend wordt.

De volgende stap kijkt of er ballen in de pockets liggen tijdens deze huidige observatie. Per bal wordt er berekend hoe groot de overlap is tussen de bal en elke pocket. De oppervlakte van de rechthoek die de overlap voorstelt wordt gedeeld door de oppervlakte van de box rond de bal, met andere woorden, hoeveel procent de box rond de bal overlapt met de pocket. Er is in het team besloten om de bal als binnen te tellen als er een overlap van 85 procent wordt geregistreerd. Deze binnengespeelde ballen worden opgeslagen in een lijst.

Dan wordt er per bal in de lijst een aantal dingen gecontroleerd. In de eerste plaats wordt er gekeken of de bal de witte bal is. In dit geval wordt er geregistreerd dat er een fout is gemaakt.

De volgende controle kijkt of de bal de zwarte acht bal is. Indien dit de laatste bal is van de huidige speler wordt dit geregistreerd als een overwinning. Anders is het een fout en heeft de huidige speler verloren.

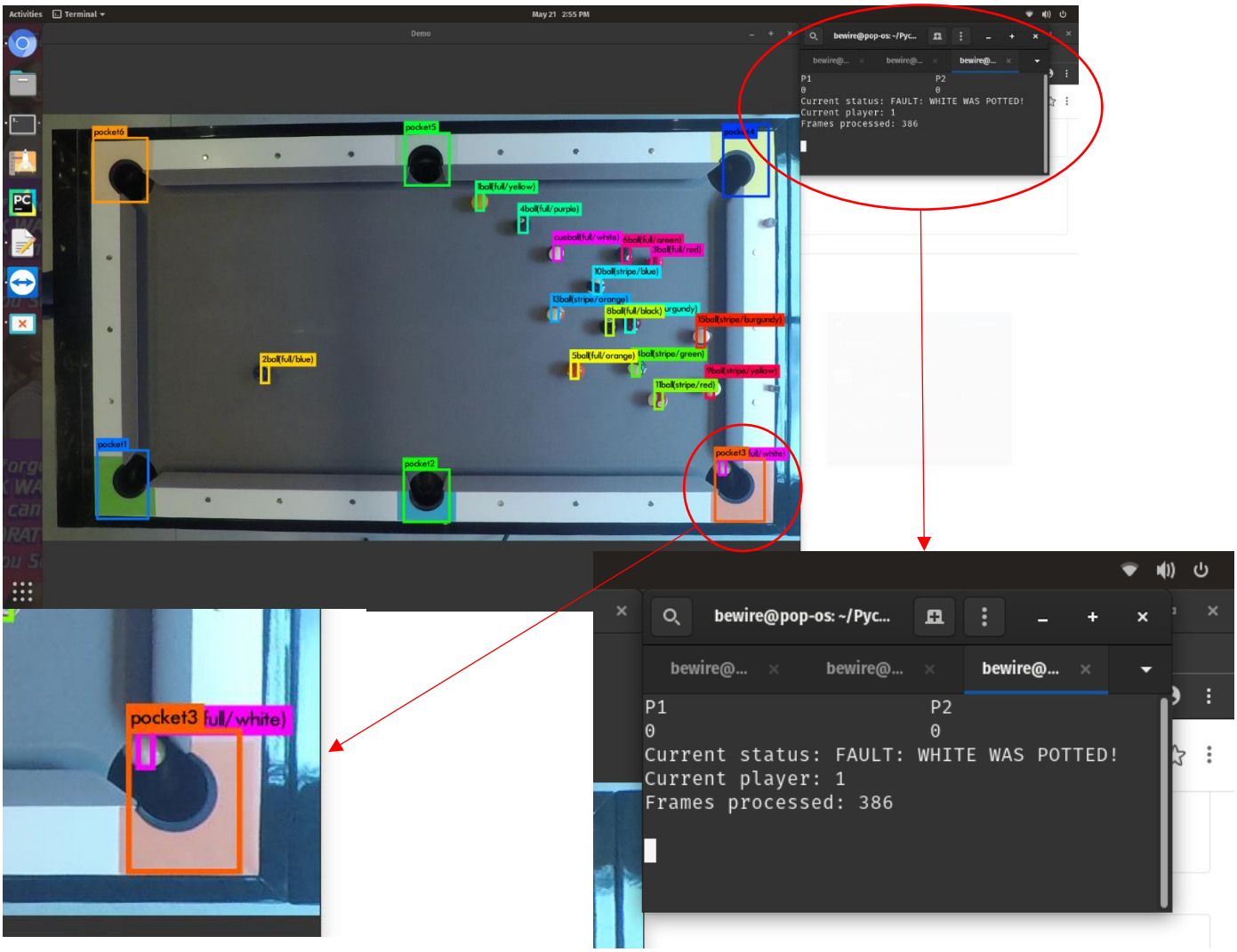
Als de bal één van de andere ballen is, wordt er eerst gekeken of er beslist is welke type ballen welke speler moet binnenspelen. Als dit nog niet gebeurd is wordt dit eenmalig vastgesteld. Indien het correct type bal is binnengespeeld, wordt dit getoond en wordt de score van de huidige speler verhoogd. Als het verkeerde type bal binnen gespeeld wordt, dan krijgt de andere speler een punt, en zal de fout geregistreerd worden.

De score van elke speler wordt berekend door het aantal ballen dat nog in spel zijn af te trekken van het aantal ballen dat er standaard in het begin van het spel aanwezig zijn. Dit gebeurt per type bal, zodat elke speler zijn eigen score heeft.

Verder wordt er geprobeerd om de eerste bal die door de witte bal geraakt wordt te bepalen. Dit lukt echter niet in de test omgeving.

3.6 Resultaten project

Als eindresultaat is er een consoleapplicatie die het spel pool kan volgen en aan kan geven wanneer een fout wordt gemaakt. Figuur 19 toont hoe deze console applicatie de output van de verwerking laat zien aan de spelers.



Figuur 19 Minimal Viable Product

Hoewel dit product bewijst dat dit concept kan werken, zijn er nog een aantal problemen die niet opgelost raakten binnen de stage.

Zoals vermeldt kon het *reinforcement learning* gedeelte, of fase 3, niet geïmplementeerd worden. Deze technologie bleek nog te jong om te implementeren op een complexe omgeving als het pool spel.

Verder is de object detectie niet altijd 100 procent accuraat, en kan het zeer moeilijk ballen die snel bewegen volgen. Hierdoor blijkt het in de meeste gevallen onmogelijk om te detecteren welke bal als eerste geraakt wordt.

Als laatste punt, omdat de object detectie niet perfect is, verschuiven de labels die geplaatst worden af en toe, of worden de verkeerde ballen herkend als ze zich in de pocket bevinden. Dit veroorzaakt dat de scores niet altijd kloppen, en ook dat de huidige speler te snel wisselt, omdat er beweging is gedetecteerd. Er is een foutmarge van vijf procent ingesteld, maar deze vangt het probleem niet in zijn geheel op.

3.7 Toekomstig werk

In dit project is gebleken dat objectdetectie, zelfs met een enorm snel netwerk, niet de mogelijkheid biedt om tracking uit te voeren op snelle objecten. Een degelijk objectdetectie-model, waar dan object tracking aan toegevoegd wordt, kan het probleem van de inaccurate detectie mogelijk verhelpen.

In het onderzoeksgedeelte worden de mogelijkheden om het *reinforcement learning* gedeelte te laten convergeren aangehaald.

3.8 Reflectie

Het project startte vreemd voor mij. Eén vrij brede uitlijning van de opdracht was alles dat we hadden toen we moesten beginnen. Het eerste moment waren we dus niet bezig met het verdelen van werk, hoe we bepaalde dingen gingen aanpakken en wat we exact gingen onderzoeken. In de opleiding zijn de opdrachten over het algemeen veel beter uitgelijnd, en ging het over toepassen van kennis die aangebracht was. Voor deze opdracht was er echter weinig voorkennis aanwezig.

Na een stroef verlopende eerste week, hebben we met de hulp van design sprints, en een aantal besprekingen met onze technische coach een beter beeld kunnen vormen van de opdracht en het aanverwante onderzoek.

In de eerste plaats was dit voornamelijk onderzoek verrichten over de technologieën die beschikbaar waren die in het kader van de opdracht pasten. Yente en ik zijn beiden eerder praktisch gerichte mensen, dus in plaats van de verschillende opties te onderzoeken, voerden we testen uit, en bekeken we onze eigen ervaring met de technologie boven de informatie die op het internet te vinden is. Dit kostte tijd, en mogelijk zelf te veel tijd. Het is echter een veel beter en sterker leerproces dan enkel informatie verwerken.

Omdat we zo nauw met deze technologieën werkten, is onze kennis op die gebied veel groter, en zal deze ook veel langer beschikbaar blijven. Daarnaast gaf het ons de mogelijkheid om ook in de interne werking dingen aan te passen zonder dat er problemen veroorzaakten werden in andere onderdelen van het project.

Als ik kritisch terugkijk op mijn eigen functioneren in het project zijn er een aantal dingen waar ik vindt dat ik mijneigen teleurgesteld heb. Er zijn een aantal momenten in het project geweest waar ik niet meteen een oplossing kon vinden voor een probleem en die tegenslag niet correct verwerkte. Kleine opbouwende frustraties die ik verkeerd uitte tegen mijn, met geluk, begripvolle projectpartners.

Natuurlijk heb ik niet alleen negatieve gevoelens bij mijn eigen werk, want ik ben trots op het uiteindelijke resultaat dat ik hier neerzet. Ik heb me tijdens de stage met veel ingewikkelde onderwerpen kunnen bezighouden en de enorme hoeveelheid die ik heb kunnen leren tijdens de periode verbaast me nog steeds.

Toen ik aan de stage begon, voelde ik me nog niet klaar voor het beroepsleven. Nu, met deze ervaring achter de rug, heb ik een veel duidelijker beeld van het werkveld en besef ik dat niemand ooit volledig klaar is om die stap te nemen. Het is nu dat ik begrijp dat een sprong in het wild is en ik ben klaar om die sprong te nemen.

II. Onderzoekstopic

1 Probleemstelling

Bewire heeft op zijn kantoor een mooie pooltafel staan, en hier mag tijdens de middagpauze op gespeeld worden. De regels van 'pool' worden echter niet altijd correct toegepast. Om dit te controleren worden er twee lagen artificiële intelligentie ontwikkeld in dit project. De eerste laag zorgt voor objectherkenning en de poolballen tracken. De verwerkte data van deze laag wordt als input gebruikt voor de tweede laag.

De tweede laag leert via *reinforcement learning* hoe het spel pool werkt, en kan dus een indicatie geven wanneer er fouten gemaakt worden. Hier eindigt het echter nog niet.

Het systeem, na de training, kan ook aan de huidige speler hints geven die het meeste kans op overwinning levert.

Om dit systeem te creëren moeten deze twee onderdelen uitgewerkt worden. Om de objectherkenning tot een goed einde te brengen, wordt dat onderdeel gebaseerd op de paper van Anissa Schirock [11].

Het tweede gedeelte, de AI gebaseerd op *reinforcement learning*, is de focus van dit onderzoek. *Reinforcement learning* werkt met een simulatie van de omgeving, die de werkelijke omgeving zo goed mogelijk kopieert, waarin de AI acties kan ondernemen. Elk van deze acties geven het systeem een score, met het doel een zo hoog mogelijke score te behalen.

Deze acties worden in het begin willekeurig gekozen, tot het systeem leert bepalen welke acties meer punten opleveren in welke situatie. Uiteindelijk leert de AI een perfect pad nemen om een zo hoog mogelijke score te halen.

Welke stappen er ondernomen moeten worden om een d*eep reinforcement learning*-model toe te passen op een situatie wordt hier onderzocht.

Om dit te bereiken wordt er een antwoord gezocht op de volgende vragen:

- Wat is reinforcement learning?
- Hoe maakt het model beslissingen?
- Hoe wordt een *deep reinforcement learning* model gecreëerd?

2 Methode van onderzoek

Om dit onderzoek tot een goed einde te brengen wordt er eerst een onderzoek gedaan naar de algemene theorieën achter *reinforcement learning*, via papers en andere ondersteunende literatuur.

Hierna volgt een diepere studie van de verschillende manieren om *reinforcement learning* toe te passen, zoals de *Temporal Difference*-methode, ook bekend als *Q-learning,* maar belangrijker de *deep reinforcement learning* algoritmes.

Toen dit eenmaal voltooid was, is er geëxperimenteerd binnen een OpenAI Gym-omgeving met een PoC om te bepalen welke methode er toegepast kan worden binnen de opdracht, met een versimpelde probleemstelling zodat de verwerking van de opties een stuk sneller verloopt.

Voor de creatie van de modellen wordt Keras gebruikt, een API die het simpeler maakt om een neuraal netwerk op te bouwen in de verschillende frameworks. Het gekozen framework is TensorFlow, om de tensorkernen beschikbaar op de GPU optimaal te benutten.

Het resultaat van dit onderzoek is een studie van *reinforcement learning* zijn, met een PoC waarvan de resultaten gebruikt worden om de conclusie te staven met de gewonnen statistieken.

3 Uitwerking onderzoek

3.1 Inleiding

Reinforcement learning, soms ook *approximate dynamic programming* of *neuro-dynamic programming* genoemd, is een type *machine learning* en wordt op hetzelfde niveau geplaatst als *supervised* en *unsupervised learning* [22].

Een korte samenvatting van deze twee andere principes: *supervised learning* is het meest gebruikte algoritme binnen *machine learning* [11]. *Supervised learning* werkt met gelabelde inputdata. Het doel is dat het systeem de kenmerken van de gelabelde data kan herkennen, en dan op nieuwe data een accurate voorspelling kan doen aan de hand van die kenmerken. Typisch wordt de gelabelde data gesplitst in een set trainingsdata en een test set. De test data wordt niet gebruikt gedurende het trainingsproces. Het resulterende model kan zichzelf testen door deze testdata te verwerken en de gemaakte voorspelling te vergelijken met het bestaande label. Uit de resultaten van deze stap kan de accuraatheid van een model afgeleid worden.

Unsupervised learning daarentegen werkt niet met gelabelde data, maar heeft wel nog steeds inputdata nodig. Doordat de data niet gelabeld is, is het veel simpeler om deze data te verkrijgen. Er wordt hier verwacht dat het systeem zelf verbanden gaat zoeken in de data. Het leerproces verloopt hierdoor trager, maar er kunnen door dit soort systemen wel verbanden gelegd worden tussen de data die niet door een mens gevonden worden. [23]

Reinforcement learning werkt niet met start data. De *agent* wordt in een simulatie van zijn omgeving geplaatst, en kan hier acties in uitvoeren. Elk van die acties heeft een bepaalde waarde berekend door een functie, die ook rekening houdt met de mogelijke toekomstige waardes die op deze actie volgen. Deze toekomstige waardes worden verminderd door een *discount factor* [24].

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot maxQ(s_{t+1}, a))$$

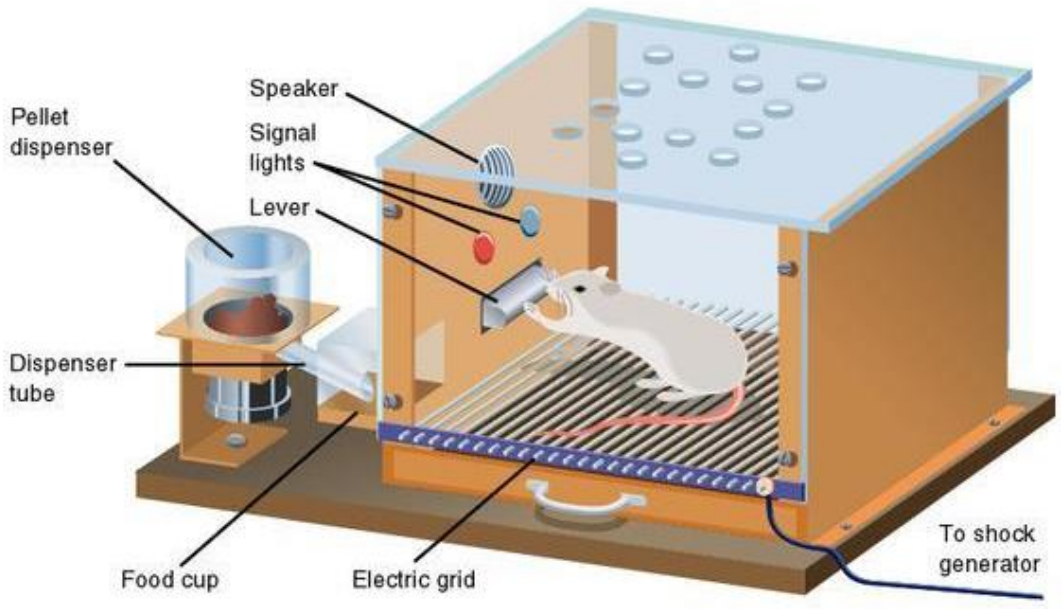
Vergelijking 2 Q-waarde formule

Een voorbeeld van deze formules is te vinden in Vergelijking 2, die verder in dit werk dieper toegelicht wordt. De simulatie van de omgeving waar een *reinforcement learning agent* in traint is een benadering van de reële omgeving waarin hij zijn functie moet uitvoeren. Sommige omgevingen zijn onmogelijk volledig te modelleren en moeten daarom versimpeld worden. [25] Een goed voorbeeld van een te complexe omgeving om volledig in een model te gieten is de realiteit.

Reinforcement learning zoekt dus naar de meest performante manier om een bepaald doel te bereiken, met een balans tussen exploitatie van de huidige kennis en verkenning van de nog onbekende routes. Een *reinforcement learning*-model biedt dus de mogelijk voor continu te leren, zelfs wanneer het al in gebruik is. Een spelomgeving is een perfecte omgeving om dit soort modellen te trainen, aangezien de meeste spellen al een puntensysteem hebben dat gemakkelijk vertaald wordt naar de beloningen voor een *reinforcement learning*-model [26].

3.2 Reinforcement Learning

Zoals hiervoor al vermeld, is *reinforcement learning* een tak in *Machine Learning*. *Reinforcement learning* heeft wortels in gedragspsychologie [22], waar dieren bepaalde gedragen krijgen aangeleerd door beloningen en straffen. Een voorstelling die je kan maken van het *reinforcement learning* proces is de Skinner box, een experiment in operante conditionering.



Figuur 20 Skinner Box [29]

In dit voorbeeld Figuur 20 van een Skinner Box wordt een rat getraind om bij een licht en geluidsindicatie, op een knop te duwen. Indien er op de knop geduwd wordt voordat het signaal gegeven wordt, krijgt het dier een schok toegediend via de vloer. Als er op het juiste moment op de knop geduwd wordt, zal de rat een hapje krijgen. Na een periode zal de rat dan het gewilde gedrag aanleren. Hetzelfde idee wordt toegepast in *reinforcement learning*.

Net zoals bij *supervised* en *unsupervised learning* zijn er variaties in *reinforcement learning*. *Reinforcement learning* op zich is meestal gericht op simpelere problemen, waar de *state space* en *action space* uitputbaar zijn, of worden deze beperkt tot discrete klassen om uitputbaarheid te bereiken. [27]

Ter verduidelijking, een *state space* is een verzameling van alle mogelijke *states* waarin de omgeving zich kan bevinden. De *state space* van een lichtknop is bijvoorbeeld {Uit(0), Aan(1)}. Dit is een voorbeeld van een discrete *state space*, een *state space* die bestaat uit een aantal vaste waardes. Continue *spaces*, zowel *action* als *state*, bevatten een oneindige hoeveelheid waardes binnen een interval. Een voorbeeld is een dimmer waar de lichtintensiteit zich tussen uit, of 0 en aan, 1, bevindt. De *state space* zou in dit geval gedefinieerd worden als het interval [0, 1]. Een *state space* wordt ook wel een *observation space* genoemd.

Action spaces zijn gelijkend met *state spaces*, met het verschil dat deze een verzameling is van alle mogelijke acties die genomen kunnen worden, in elke mogelijke *state*.

De reden waarom uitputbaarheid belangrijk is in dit type *reinforcement learning* is omdat er een beloningswaarde zal bijgehouden worden voor elke discrete *state* in de *state space* en elke discrete actie mogelijke in elke *state*.

Bij continue waardes is dit onmogelijk, aangezien dit een oneindig grote tabel zal worden. Bij grote hoeveelheden discrete waardes is dit ook moeilijker. In *reinforcement learning,* zonder neurale netwerken, wordt er voordat er voorspellingen gemaakt worden, het liefst de volledige tabel gevuld. Hoe grotere het aantal discrete waardes, hoe groter de tabel en hoe langer het vullen van deze tabel te vullen.

Om met dit probleem om te gaan, zijn er neurale netwerken geïntroduceerd in *reinforcement learning*, die een zo accuraat mogelijke schatting maken van de beste actie. Deze netwerken kunnen simpel zijn, maar bij *deep reinforcement learning* worden netwerken gebruikt met minimaal twee *hidden layers* of verborgen lagen.

Er zijn verschillende types netwerken die toegepast kunnen worden voor een *reinforcement learning* agent. De pionier studie in *deep reinforcement learning* paste een *Convolutional Neural Network* toe op Atari spellen [26]. Een CNN wordt meestal toegepast binnen *deep learning* toepassingen die afbeeldingen moeten verwerken. Door de schermen van het Atari spel, de huidige *state*, door het CNN te laten verwerken, werden de acties bepaald die er binnen de perken van het spel genomen werden.

Voordelen van *reinforcement learning* in vergelijking met andere vormen van ML zijn dat deze generische leertechnieken toegepast kunnen worden op een grote variatie van situaties, en er geen nood is aan grote hoeveelheden data. R*einforcement learning* is mogelijk de eerste stap in *General Artificial Intelligence* [22], of met andere woorden, een AI die taken kan afhandelen zoals een mens, en ook nieuwe taken kan leren zoals een mens, zonder dat hier nieuwe algoritmes of data voor moet voorzien worden.

Voordat er dieper op uitwerking ingegaan kan worden, moeten er eerst een aantal termen uitgediept worden.

3.2.1 Concepten

3.2.1.1 Markov Decision Process

Aan de basis van het Markov Decision Process ligt de Markov-eigenschap. Een proces heeft de Markov-eigenschap als de toekomst volledig bepaald kan worden door het heden, onafhankelijk van het verleden [28].

Binnen *reinforcement learning* betekent dit, dat om de voorspellingen te maken, het heden die zal bepalen, en het verleden hier geen invloed op heeft. Anders gesteld, de huidige *state* bevat alle relevante informatie. Vergelijking 3 toont hoe deze eigenschap wiskundig wordt uitgedrukt.

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \dots, S_t]$$

Vergelijking 3 Wiskunde weergave van Markov eigenschap

Vanuit *states* met de Markov-eigenschap kan een Markov-proces of een Markov-ketting samengesteld worden. Een Markov ketting is een set transities, bepaald door een kansverdeling. Om dit simpel voor te stellen, wordt de volgende situatie gebruikt:

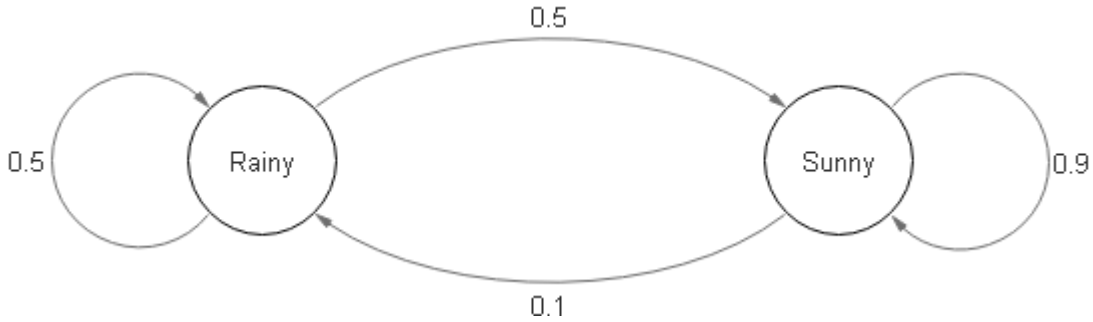
Het doel is om het weer te voorspellen. Er zijn twee mogelijke weersomstandigheden, of *states*, zonnig (Sunny) of regen (Rainy). Na een periode van observaties worden de transitiekansen in Tabel 2 vastgesteld.

$S_t \setminus S_{t+1}$	Sunny	Rainy

Sunny	0.9	0.1
Rainy	0.5	0.5

Tabel 2 Transitiekansen

Uit Tabel 2 kan de kans dat de huidige weeromstandigheid S_t zich de volgende dag S_{t+1} voortzet gevonden worden. De som van de waardes in elke rij is 1, of 100%. De Markov-ketting kan op verschillende manieren uitgedrukt worden zoals visueel, zichtbaar in Figuur 3Figuur 21, of met twee matrices, zoals in Vergelijking 4. De transitie matrix P, een matrixvoorstelling van de kansentabel, en een *initial state vector* I. Deze vector, een matrix van vorm (N,1) met N = aantal *states*, toont de kans dat de Markov-ketting zich in een bepaalde staat bevindt aan de start van het proces.



Figuur 21 Visuele representatie van een Markov-ketting

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix},\ I = \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix}$$

Vergelijking 4 Matrixrepresentatie van een Markov-ketting

Bij een standaard Markov-proces zijn de transities willekeurig verdeeld over de transitiematrix. Dit betekent dat er geen stimulansen aan de transities verbonden zijn, en ook geen acties ondernomen worden om een transitie te voltooien.

De volgende stap is het *Markov Reward Process*, dat een beloning verbindt aan elk transitie die er gemaakt kan worden. Een MRP is een veeltal (S, P, R, γ) met S als voorstelling van de eindige verzameling *states*, P als de transitiekans functie, R als de verzameling van alle mogelijke *rewards* of beloningen, en γ als kortingsfactor of *discountfactor*.

De kortingsfactor vermindert de waarde van alle toekomstige *rewards* die verbonden zijn aan een transitie. Deze factor ligt altijd tussen 0 en 1. Hoe dichter de waarde bij 0 ligt, hoe minder belangrijk toekomstige beloningen worden geschat. Indien de waarde 0 is, dan worden deze mogelijke *rewards* volledig genegeerd, en is enkel de beloning van de huidige transitie belangrijk. Daartegenover staat dat de grotere waardes meer belang hechten aan de toekomst, en met een waarde 1 is de toekomstige beloning even belangrijk als de huidige. [28], [29]

Deze factor helpt met het meest efficiënte pad naar de hoogste eindbeloning te gaan, doordat de mogelijke transities met hogere toekomstige beloning ook de transities zijn waar de hogere beloning te halen is.

Aangezien het niet voorspelbaar is hoelang een Markov proces zal zijn, kan je beloning voor eender welke transitie, in het geval dat γ ≠ 0, omschrijven als Vergelijking 5.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Vergelijking 5 Beloningsberekening [30]

De redenen dat toekomstige beloningen verkleind worden met de kortingsfactor γ is de onzekerheid dat de volgende stappen de beloningen vergaard kunnen worden, en dat dit vergelijkbaar is met het menselijke of dierlijk cognitie, dat een preferentie heeft voor onmiddellijke beloningen.

Indien γ = 1, dan kan de γ factor in Vergelijking 5 volledig genegeerd worden. Hoewel dit toepasbaar kan zijn in episodische gevallen, met een gelimiteerd aantal stappen T, wordt dit vermeden in continue problemen. In dit geval zal de vergelijking altijd uiteindelijk naar ∞ convergeren.

Met een factor kleiner als 1 kan oneindigheid vermeden worden. Als de *rewards* constant en niet nul zijn is het resultaat altijd eindig. Een voorbeeld is als de beloning een constante +1 is zal de waarde in een het ergste geval convergeren volgens Vergelijking 6.

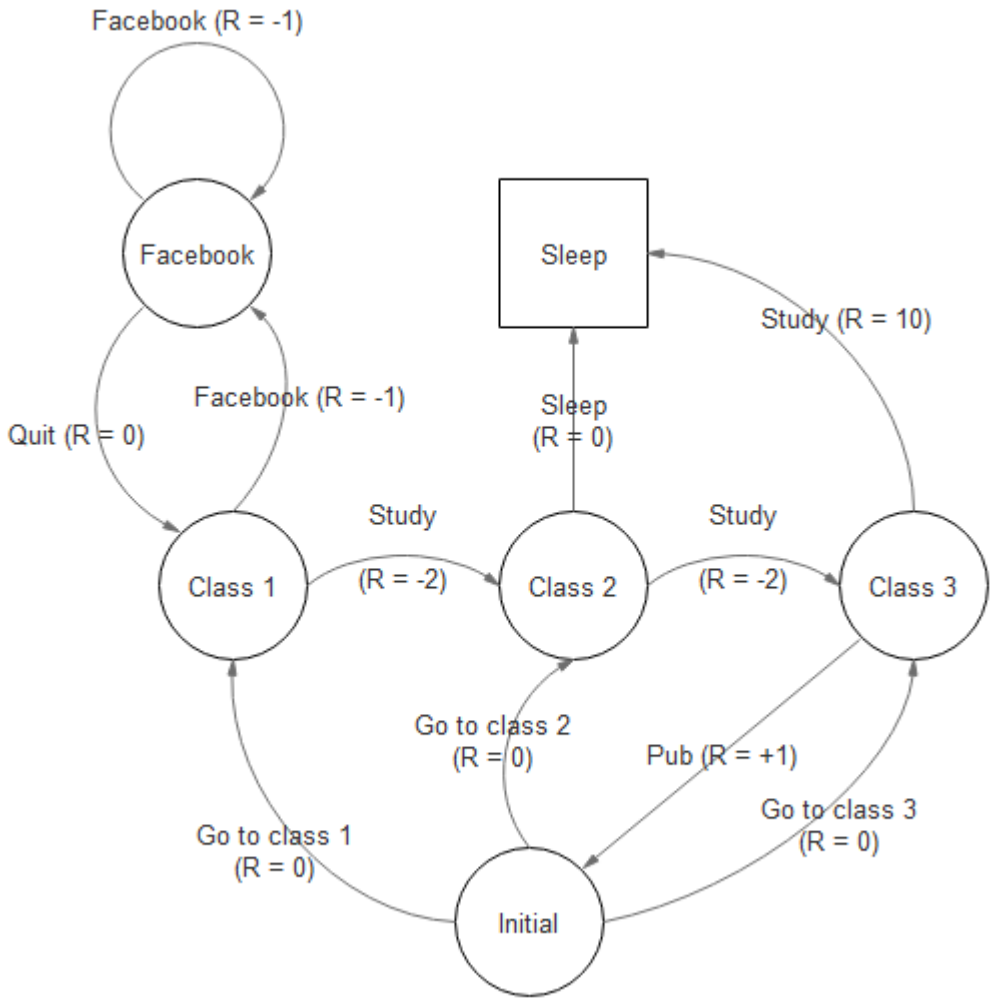
$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$$

Vergelijking 6 Convergentie van beloningen [28]

Een MRP kan dan weer uitgebreid worden met mogelijke acties, wat dan een MDP of een *Markov Decision Process* oplevert. MDP is een net zo goed een veeltal, (S, A, P, R, γ), waar A de nieuwe term is. A is de verzameling van alle mogelijke acties.

Om de actie te bepalen wordt er een *policy* π of een beleid gebruikt. Een beleid is een waarschijnlijkheidsdistributie over acties gegeven een *state* [31]. Tijdens het trainen van een netwerk binnen het kader van *reinforcement learning*, gaat het dus om of het bepalen van de beste *policy* voor de probleemstelling.

Ter voorbeeld nemen we Figuur 22, een volledig MDP model gebaseerd op een student. Initieel heeft hij keuze uit naar 1 van zijn drie klassen te gaan, maar alle drie deze acties leveren hem een neutrale beloning op.



Figuur 22 Complete voorstelling van een MDP in een studentsituatie [28]

Afhangende van welke policy er bepaald wordt door het systeem gaat hij andere acties ondernemen. Als Q-learning op dit model wordt toegepast zal de waarschijnlijkheidsdistributie die hieruit voorkomt een hoger kans op leveren voor “Go to class 3”. Vanuit de status “Class 3” zijn er immers twee paden die beloningen opleveren. Deze beloningen worden in rekening gebracht via de toekomstige *rewards* in de formule. De kans dat de andere *states* niet bereikt worden is echter nooit nul zolang het proces stochastisch blijft.

3.2.1.2 Model based en model free

Er zijn twee verschillende stromingen waarop een agent de optimale *policy* kan zoeken. *Model based* is het anticiperen van toekomstige mogelijkheden, zowel beloningen als *states* als transitiemogelijkheden. Er is een gelimiteerd aantal interacties met de omgeving tijdens het trainingsproces, waar de *agent* een model rond bouwt. Hierna zal het systeem simulaties doen tegen dit model.

Dit heeft het voordeel dat het leerproces sneller omdat er geen constante interacties zijn met de omgeving, en de omgeving niet gereset moet worden. Het systeem kan zijn ervaringen of *experience*

gebruiken om in te schatten hoe de actuele omgeving zal reageren. Een groot nadeel van dit type is dat, indien het model van de omgeving inaccuraat is, de training niet het verwachte resultaat zal opleveren. De oorzaak is hier dat het model van de omgeving verkeerde aanpassingen oplevert aan het beleid, en dus de geplande acties niet het resultaat opleveren dat er zou zijn volgens het model van de omgeving.

Voordat een *model based* algoritme zijn model kan bouwen, heeft het een aantal interacties met de omgeving nodig. Om deze interacties te doen is een *model based* algoritme gedwongen om *model free* technieken toe te passen.

Model free learning maakt geen gebruik van een model, elk van zijn interacties zijn met de realiteit of de simulatie die de realiteit zo dicht mogelijk representeert. Zij passen hun *value functions,* te vinden in 3.2.1.3, aan op beloningen die ze krijgen uit deze interacties. Het voordeel is dat deze interacties altijd een weerspiegeling zijn van hoe de actuele omgeving om zal gaan als diezelfde acties nogmaals uitgevoerd worden op de omgeving in dezelfde *state*.

Omdat de omgeving altijd de actie moet interpreteren duurt het leerproces wel langer in een *model free* algoritme. Een andere manier om *model free learning* te bekijken is als *trial-and-error learning*. De agent gaat acties ondernemen en kijken wat het resultaat is, en de volgende keer dat hij die *state* bereikt, houd hij rekening met zijn vorige ervaringen in die *state*. Dit kan inhouden dat hij dezelfde actie opnieuw neemt als deze goed was, of een willekeurige andere actie.

3.2.1.3 Value functions

Er zijn twee types *value functions* of waarde functies die gebruikt worden in *reinforcement learning* om de juiste *policy* te bepalen. De eerste is de *state value function* vindbaar als Vergelijking 7. Deze beschrijft de waarde van een staat wanneer een bepaald beleid π wordt gevolgd. Deze functie geeft aan hoe goed het is voor de *agent* om zich in een bepaalde staat te bevinden.

$$V^{\pi}(s) = E_{\pi}[R_t|s_t = s]$$

Vergelijking 7 State Value Function

De andere waarde functie is de *action value function*. Deze functie beschrijft de waarde van een actie *a*, in een bepaalde staat *s*, wanneer een bepaald beleid π gevolgd wordt. Vergelijkbaar met de *state value function*, geeft deze functie aan hoe goed het is voor een *agent* om een bepaalde actie te nemen in een bepaalde staat. Deze vergelijking staat omschreven in Vergelijking 8.

$$Q^{\pi}(s,a) = E_{\pi}[R_t|s_t = s, a_t = a]$$

Vergelijking 8 Action Value Function

Belangrijk te onthouden is dat in beide vergelijkingen de *policy* een bepalende factor is voor de uitkomst. De uitkomst van een *state* of *action* verandert immers als de manier waarop een keuze gemaakt wordt verandert.

Ook maken we gebruik van een verwachting, of *expectation,* E, om de willekeur op te vangen. Deze willekeur komt voor uit de stochastiek van een MDP zijn transitiefunctie.

Deze willekeur kan nog verder verhoogd worden als de *policy* ook stochastisch is. In dit geval moet er een combinatie gemaakt word uit de resultaten van alle mogelijke acties die we nemen.

De verwachting houdt dus rekening met de willekeur in toekomstige acties, volgens de huidige *policy* en de willekeur van de *state* die terug wordt gegeven door de omgeving. [32]

3.2.1.4 Bellman vergelijking

Bellman vergelijkingen worden overal gebruikt in *reinforcement learning*, en zijn noodzakelijk om de algoritmes die gebruikt worden binnen *reinforcement learning* te begrijpen. Beide *value* functies hebben een bijhorende Bellman vergelijking.

Kort samengevat is de Bellman verwachtingsvergelijking de som van verwachting, waarbij rekening gehouden wordt met de transistiewaarschijnlijkheid *p* van elk element van de som. Deze transitiewaarchijnlijkheid wordt gedefinieerd als Vergelijking 9.

$$P_{ss'}^{a} = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$$

Vergelijking 9 Transitiewaarschijnlijkheid

De uitkomst van Vergelijking 9 is dus de waarschijnlijkheid waarop, als de *agent* in staat *s* begint en actie *a* neemt dat hij in staat *s'* terecht komt.

Daarnaast is er de wiskundige definitie van de verwachte beloning $R_{ss'}^{a}$ als Vergelijking 10. De uitkomst is hier dus de beloning die verwacht wordt wanneer de *agent* start in staat *s* en actie *a* onderneemt en hij in staat *s*' terecht kwam.

$$R_{ss'}^{a} = E[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a]$$

Vergelijking 10 Verwachte beloning

Deze twee vergelijkingen vormen de basis voor de Bellman vergelijking. De Bellman vergelijkingen in Vergelijking 11 en Vergelijking 12 zijn recursief, de laatste term van de vergelijking, is de vergelijking, maar met een andere *state* als variabele. Deze vergelijkingen worden gebruikt om de waardes van *states* uit te drukken in functie van andere *states*. Dit geeft de mogelijkheid tot het iteratief oplossen van een MDP en ligt aan de basis van *reinforcement learning*. [29]

$$V^{\pi}(s) = \sum_{a} \pi(s, a) \sum_{s'} P_{ss'}^{a} [R_{ss'}^{a} + \gamma V^{\pi}(s')]$$

Vergelijking 11 Bellman vergelijking voor state waardes met een willekeurige policy

$$Q^{\pi}(s, a) = \sum_{s'} P_{ss'}^{a} [R_{ss'}^{a} + \gamma \sum_{a'} \pi(s', a') Q^{\pi}(s', a')]$$

Vergelijking 12 Bellman vergelijking voor action waardes met een willekeurige policy

Het doel is echter niet om de Bellman vergelijking op te lossen voor een willekeurige policy, maar voor de ideale policy. Dit kan met een simpele aanpassing bereikt worden. De optimale *policy* wordt voorgesteld door π*. In dit geval zijn Vergelijking 13 en Vergelijking 14 de uitgewerkte Bellman vergelijkingen. Er wordt hier dus geen gemiddelde berekent over de acties maar enkel gekeken naar die actie die de maximale *reward* oplevert.

$$V^{\pi*}(s) = max_{a} \sum_{s'} P_{ss'}^{a} [R_{ss'}^{a} + \gamma V^{\pi*}(s')]$$

Vergelijking 13 Bellman optimaliteitsvergelijking voor states

$$Q^{\pi}(s,a) = \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma \, max_a Q^{\pi*}(s',a')]$$

Vergelijking 14 Bellman optimaliteitsvergelijking voor actions

3.2.1.5 Gradient descent

Om *gradient descent* uit te leggen, wordt er een hypothetische situatie gebruikt. Stel, een persoon zit vast in de bergen en wil het dal bereiken. Omwille van de weersomstandigheden, is zicht enorm beperkt, en is het pad naar de bodem van de berg dus niet zichtbaar. De persoon moet dus steunen op een beperkte hoeveelheid lokale informatie om het minimum, de bodem van de berg, te vinden.

Om tot de bodem te geraken kan hij de *gradient descent* methode gebruiken. Hij voelt voor zich uit en zoekt de plaats waar de grond het steilst naar beneden afloopt. Na dit bij elke stap te doen, belandt hij uiteindelijk in het laagste punt. Hij kan weten dat hij dit punt heeft bereikt omdat, vanaf deze positie, elke richting naar boven loopt.

Veronderstel dat de persoon niet met een simpele observatie kan bepalen waar de grond het steilst afloopt, maar hier een apparaat voor moet gebruiken dat een bepaalde hoeveelheid tijd nodig heeft om de steilste locatie te vinden. Dit vermoeilijkt de opdracht omdat de keuze tussen tijd verliezen door het apparaat en tijd verliezen omdat het verkeerde pad tijdelijk gevolgd werd een nieuwe afweging is.

Deze anekdote bevat alle belangrijke onderdelen die gebruikt worden tijdens het optimalisatie proces van *reinforcement learning.* De helling zelf is de waarde-functie, het apparaat is differentiatie, met differentiatie kan de richtingscoëfficiënt van de functie berekend worden. De richting die hij opgaat, of hoe de hoeveelheid dat de gewichten worden aangepast, wordt ook bepaald door die richtingscoëfficiënt. [33]

3.2.2 Reinforcement learning zonder neurale netwerken

Naast *deep reinforcement learning* is er de variant zonder neurale netwerken, waaruit *deep reinforcement learning* gegroeid is. De meest bekende van deze is *Q-learning* of de specifiekere *Temporal Difference* methode.

Dit type *reinforcement learning* werkt met een zogenoemde *Q-table*. Deze Q-tabel houdt voor elke state, het resultaat van elke actie bij als een Q-waarde bij. Q-waardes leggen link tussen de beloning en *state action pairs*. Een andere manier om Q-waardes te bekijken is als de beloning met alle toekomstige beloningen in rekening gebracht.

Deze Q-waarde wordt in het geval van *Temporal Difference learning* berekend volgens Vergelijking 2. Deze formule wordt nu nauwer bekeken, en wordt hier dus herhaald als Vergelijking 15.

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot maxQ(s_{t+1}, a))$$

Vergelijking 15 Q-waarde formule als Temporal Difference learning wordt toegepast

De Q-waarde wordt dus bepaald door het gewogen gemiddelde van twee factoren. De eerste factor is de oude waarde. Dit kan de begin waarde zijn, geïnitialiseerd door de programmeur. Vaak is die beginwaarde nul, maar soms worden de tabellen gevuld met willekeurige waardes.

De tweede factor is gecompliceerder. α is de *learning rate* of leersnelheid van het systeem. Deze bepaalt hoe snel de Q-waardes zich aanpassen. Aan de start van het leren, zal dit een grote waarde hebben. Deze zal afnemen naarmate het leerproces verloopt. Hoe deze afname verloopt wordt

volledig bepaalt door de programmeur. De waarde wordt echter nooit nul tijdens het leerproces, want dan zou het systeem niet langer iets leren.

De volgende variabele, r_t, is de onmiddellijke beloning die het systeem krijgt voor het ondernemen van die bepaalde actie in die bepaalde *state*. Dan komt het belangrijkste deel van de formule, namelijk de berekening van de toekomstige beloning. γ, de *discount* factor die uitvoerig besproken is in onderdeel 3.2.1.1 zal hier niet verder worden aangehaald.

De andere helft van die laatste factor, is heel gelijkend met de term die voor het gelijkheidsteken staat. Hier is de recursiviteit ook weer aantoonbaar. *Q-learning* is een *greedy* algoritme, het stuurt aan op een *greedy* policy. Hiermee wordt bedoeld dat er altijd gekozen wordt voor de actie die de hoogste beloning oplevert. Dit is zichtbaar in deze term, waar de maximale Q-waarde geselecteerd wordt voor de volgende *state* bij het bepalen van mogelijke toekomstige beloningen. Dit reflecteer de optimaliteitsvergelijkingen van Bellman.

Het trainen van dit algoritme houdt het vullen van de tabel in. Afhangende van de grootte van de tabel kan er geopteerd worden om de volledige tabel te vullen voordat het model in productie gebracht worden.

De grote nadelen van dit type algoritmes is dat deze niet met continue waardes kunnen omgaan. Er moet dus gediscreteerd worden om het algoritme zijn taak te laten vervullen, of de omgeving moet met discrete waardes werken. Dit is een veel omgevingen geen optie, zeker niet als het de bedoeling is om de AI in de realiteit zijn taak te laten uitvoeren.

Eén tweede nadeel, zeker als we het vergelijken met *deep reinforcement learning*, is dat het model, met andere woorden, de Q-tabel, veel groter is dan de modellen die door *deep reinforcement learning* worden gebruikt. In Tabel 3 is dit verschil duidelijk zichtbaar.

Het derde grote nadeel is de tijd die nodig is om dezelfde hoeveelheid training te doorlopen. Ook dit nadeel kan aangetoond worden in Tabel 3.

Methods	Average Reward	Training Time	Model Size
Q-table	-6.4	136 min	1.12 GB
DQN	-21.3	27 min	162 KB
A3C (continuous action)	-19.44	13 min	8 KB
A3C (discrete action)	-18.46	17 min	149 KB
Random	-22.8	-	-

Tabel 3 Evaluatie resultaten van 100 afleveringen, trainingstijd, en model grootte info in een omgeving met twee ballen. [34]

Er zijn ook voordelen aan dit type algoritmes. Het eerste van deze is dat dit type algoritme in de situaties waarin het correct kan toegepast worden, voor betere resultaten zorgt dan de alternatieven.

Daarnaast leert dit type ook stabieler dan de *deep* variant, geleerde elementen worden niet verleerd. In Figuur 23 is te zien dat de Q-tabel een hoger gemiddelde beloning behaald dan de andere algoritmes. Ook zichtbaar op de figuur is dat de Q-tabel wel later is in het halen van die hoge beloningen, dit toont weer aan dat de trainingstijd van dit type veel hoger ligt.

Figuur 23 Gemiddelde beloning gespreid over 1000 afleveringen trainingstijd voor Q-tabel, DQN, A3C met continue acties, A3C met discrete acties, en een willekeurig beleid in een omgeving met twee ballen. [30]

3.2.3 Deep Q-network

Deep Q-network, beter gekend als DQN, is een uitbreiding op *Q-learning*. Het verschil met het voorheen besproken *Temporal Difference learning* ligt in het gebruik van een neuraal netwerk voor het inschatten van de optimale *action-value function*, in plaats van deze af te leiden van een tabel waar de hoogste waardes altijd beschikbaar zijn. Net als Q-learning is DQN *model-free*.

Er zijn een aantal problemen gerelateerd aan het gebruik van een neuraal netwerk om deze waardes in te schatten. *Reinforcement learning* wordt instabiel als er een niet-lineaire functie benadering zoals een neuraal netwerk gebruikt wordt om de Q-functie voor te stellen. Daarnaast is convergentie niet langer gegarandeerd. Het lijkt echter dat deze functie benaderaars nodig zijn om grote *state spaces* te kunnen leren en generaliseren. [35]

Deze instabiliteit wordt veroorzaakt door een correlatie tussen een sequentie van observaties, kleine veranderingen aan de Q-waardes die het beleid significant kunnen veranderen en daardoor de data distributie veranderen, en de correlatie tussen de actie-waardes en de doel waardes, of de toekomstige waardes. Deze doel waardes zijn de tweede factor besproken in onderdeel 3.2.2.

DQN vangt deze problemen met twee ideeën op. Het eerste idee is het biologisch geïnspireerde *experience replay*. Elke stap die genomen wordt door de agent wordt opgeslagen in een buffer. Deze buffer heeft een maximale grootte en zodra deze bereikt wordt, worden de oudste gegevens gewist en de nieuwe gegevens vervangen deze dan.

Tijdens het leren worden er uniform willekeurige ervaringen of datapunten geselecteerd uit deze buffer, en worden deze mini batches gebruikt om de Q-waarde functie aan te passen. Omdat datapunten meerdere keren geselecteerd kunnen worden in combinatie met steeds andere informatie, is dit algoritme zeer data efficiënt.

Het tweede idee dat de instabiliteit tegen gaat, is dat het bijwerken van de Q-waarde functie niet bij elke stap gebeurt. Een stap is een ondernomen actie. Deze twee ideeën zorgen dat er minder correlatie is tussen de observaties, en tussen actie en doel waardes. Het bijwerken van de Q-waardes periodiek doen zorgt ervoor dat het beleid niet bij elke stap verandert.

Deze methodes zijn veel efficiënter als de voormalige alternatieven, en kunnen toegepast worden op de grotere netwerken die nodig zijn om complexere probleemstellingen op te lossen.

In pseudo-code ziet het algoritme in de pionier-studie er als volgt uit:

```
Initialiseer replay geheugen D met capaciteit N
Initialiseer actie-waarde functie Q met willekeurige gewichten θ
Initialiseer doel actie-waarde functie Q' met gewichten θ' = θ
Voor aflevering = 1 tot M doe
        Initialiseer sequentie s1 = {x1} en voor verwerk sequentie ϕ1 = ϕ(s1)
        Voor t = 1 tot T doe
                Met waarschijnlijkheid ε selecteer willekeurige actie at
                Anders selecteer at = argmaxaQ(ϕ(st), a; θ)
                Executeer actie at in emulatie en observeer beloning rt en afbeelding xt+1
                Zet st+1 = st, at, xt+1 and voor verwerk ϕt+1 = ϕ(st+1)
                Sla transitie (ϕt, at, rt, ϕt+1) in D op
                Selecteer willekeurige mini batch van transities (ϕj, aj, rj, ϕj+1) uit D
                Zet yi = rj als de aflevering eindigt op stap j + 1
                Anders zet yi = rj + γ maxa'Q'(ϕj+1, a'; θ')
                Voer een gradient descent stap uit op (yi – Q(ϕj, aj; θ))2 met netwerk
        parameters θ
                Elke C stappen, reset Q' = Q
        Einde lus
Einde lus
```

Er wordt gebruik gemaakt van een ε-*greedy* algoritme. ε is de waarde die bepaalt of er gekozen wordt voor een willekeurige actie of voor de actie die de maximale waarde oplevert volgens het neurale netwerk.

Deze studie maakt gebruik van een CNN om observaties te maken. Er wordt gebruik gemaakt van een sequentie van afbeeldingen zodat de AI ook bewegingen kan observeren. Dit zijn de sequenties die benoemd worden in de pseudocode.

Een van de belangrijkste onderdelen is het uitvoeren van de *gradient descent*, uitgelegd in 3.2.1.5, na elke ondernomen stap. Deze stap verandert de gewichten van het neuraal netwerk, en daarmee het beleid dat het systeem gebruikt. Deze aanpassingen worden elke stap gedaan, maar het systeem neemt het nieuwe beleid, en dus de nieuwe gewichten en *Q-value function* pas aan na C aantal stappen.

Dit zorgt voor een vertraging tussen het bijwerken van Q en het moment dat de bijwerking een effect heeft op het doel y_j, wat op zich afwijkingen en trillingen minder waarschijnlijk maakt. [26], [36]

In Tabel 4 is er een vergelijking vindbaar tussen DQN, *State Action Reward State Action* of SARSA, een *agent* die met volledige willekeur acties onderneemt en een menselijke speler in een aantal Atari games. SARSA is een *reinforcement learning* algoritme dat niet steunt op neurale netwerken.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	-20.4	157	110	179
SARSA	996	5.2	129	-19	614	665	271
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	-3	18900	28010	3690

Tabel 4 Gemiddelde totale beloning voor een aantal leermethodes met een ε-greedy beleid met ε = 0.05 voor een vast aantal stappen. [26]

3.2.4 Deep Deterministic Policy Gradient

DQN heeft het mogelijk gemaakt om met continue *state spaces* om te gaan. De meeste probleemsituaties in de realiteit hebben echter naast continue *state spaces* ook continu *action spaces*. Om deze problemen aan te pakken zijn *actor-critic* algoritmes gecreëerd.

Dit type algoritmes maakt gebruik van twee neurale netwerken. Allereerst is er het *actor* netwerk. Dit netwerk geeft als uitkomst het beleid, of met andere woorden de waarschijnlijkheidsdistributie van acties over de *state*, en van daaruit wordt een actie gekozen. Deze actie in combinatie met de huidige *state* of observatie, zijn de invoer van het *critic* netwerk. Het *critic* netwerk bepaalt van deze data de *state* waarde, of met andere woorden de inschatting van beloningen die het systeem kan ontvangen vanaf die *state*. [37]

Eén van dit type algoritmes is *Deep Deterministic Policy Gradient*, of DDPG. Dit algoritme is een uitbreiding op DPG gebaseerd op het succes van DQN. Ook dit is een *model-free* algoritme en vereist dus constante interactie met een omgeving.

Het *actor* netwerk genereert een beleid door het deterministisch mappen van *states* aan een specifieke actie. Het idee is dat het *actor* netwerk het beleid voorstelt als een geparametiseerde waarschijnlijkheidsdistributie die stochastisch acties selecteert in een *state* volgens de parameters. [38]

Om dit netwerk bij te werken worden de parameters aangepast zodat het beleid leidt naar een grotere totaal beloning. Deze bijwerking wordt berekend door de ketting regel toe te passen op de verwachtte beloning vanaf een start distributie J met respect tot de parameters. Vergelijking 16 toont de zogenaamde *policy gradient* die hieruit voorkomt, die de prestatie van het beleid voorstelt.

$$V_{\theta\mu}J = E_{st\sim\rho^{\beta}}\left[\nabla_a Q(s,a|\theta^Q)\middle|s=s_t, a=\mu(s_t)\nabla\theta_\mu\mu(s|\theta^\mu)\middle|s=s_t\right]$$

Vergelijking 16 Policy gradient

Het *actor* netwerk vervult dezelfde functie als het netwerk van DQN, het inschatten van de *Q-values*. Ook de *replay buffer* en mini batches maken een terugkeer in DDPG. Omdat er met twee netwerken gewerkt wordt, is er wel nog een aanpassing nodig om het algoritme stabiel te houden.

Bij DQN wordt er gewerkt met harde updates van het netwerk, DDPG maakt gebruik van zachte updates. Er worden kopieën gemaakt van beide netwerken, die de doel waarden berekenen. Gebruik makend van die doel waarde kan het verlies berekent en geminimaliseerd worden, en deze waarden worden gebruikt om het originele *critic* netwerk bij te werken.

Met dit nieuwe *critic* netwerk wordt de *policy gradient* berekent, waar de *actor* mee wordt bijgewerkt. Na deze twee stappen zijn beide originele netwerken veranderd. Als er met harde updates gewerkt werd, zouden nu de gewichten van deze twee netwerken de nieuwe gewichten worden van de gekopieerde netwerken. Zachte updates vertragen dit proces, wat het leerproces stabiliseert, maar ook vertraagt. Vergelijking 17 en Vergelijking 18 toont de zachte updates voor de gewichten van beide doelnetwerken.

$$\theta^{Q\prime} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q\prime}$$

Vergelijking 17 Zachte update van critic netwerk

$$\theta^{\mu\prime} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu\prime}$$

Vergelijking 18 Zachte update van actor netwerk

De term die de leersnelheid bepaald is τ. Deze term is altijd kleiner als 1. Beide doel netwerken zijn nodig om stabiele doelen y_i te krijgen, die er voor zorgen dat de *critic* consistent getraind wordt zonder te divergeren.

Een ander probleem dat de kop opsteekt bij continue *action spaces* is verkenning. De oorzaak is de oneindig mogelijke acties die de *agent* tegen komt in elke bezochte *state*. [39] Om dit op te vangen wordt er gebruik gemaakt van een ruis proces dat tijdens het trainen de gekozen acties verandert.

Met dit allemaal samengevat ziet het algoritme er in pseudocode als volgt uit: [35]

Willekeurige initialisatie van *critic* netwerk $Q(s,a|\theta^Q)$ en *actor* $\mu(s|\theta^\mu)$ met gewichten θ^Q en θ^μ
Initialiseer doel netwerk Q' en μ' met gewichten $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialiseer *replay buffer* R
Voor aflevering = 1 tot M doe:
    Initialiseer een willekeurig proces Ɲ voor actie verkenning
    Ontvang initiële observatie s_1
    Voor t = 1 tot T doe:
        Selecteer actie $a_t = \mu(s_t|\theta^\mu) + Ɲ_t$ overeenkomstig met het huidig beleid en verkenningsruis
        Voer actie a_t uit en observeer beloning r_t en nieuw *state* s_{t+1}
        Sla transitie (s_t, a_t, r_t, s_{t+1}) op in R
        Bemonster een willekeurige mini batch van N transities (s_i, a_i, r_i, s_{i+1}) uit R
        Zet $y_i = r_i + \gamma Q'(s_{i+1},\mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update *critic* door verlies te minimaliseren: $L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$
        Update *actor* beleid met de bemonsterde *policy gradient*:
        $V_{\theta\mu}J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)\big|s = s_i, a = \mu(s_i)\nabla\theta_\mu\mu(s|\theta^\mu)\big|s = s_i$
        Voer zachte updates van doelnetwerken uit
    Einde lus
Einde lus

3.3 Het creëren van een model

3.3.1 Selecteren omgeving

Als prototype zal er een model getraind worden op één van de beschikbare omgevingen die vervat zitten in OpenAI Gym. Het uiteindelijke doel is een model creëren dat een één op één mapping is van een poolspel in de realiteit en een poolspel in een gesimuleerde omgeving.

Er moet een omgeving gekozen worden die aan een aantal variabelen voldoet om zodat het gekozen algoritme ook toegepast kan worden op de uiteindelijke pool omgeving van het project.

Allereerst moet er bepaald worden of er continue *state space* is. Het spel pool wordt gespeeld met zestien ballen. Elk van deze ballen kan op elk coördinaat van de tafel liggen, in combinatie met vijftien andere ballen die ook elk coördinaat kunnen innemen. De *state space* zou technisch gezien discreet kunnen zijn, maar deze *state space* is zo enorm, dat het nagenoeg onmogelijk wordt om deze op een discrete manier uit te drukken. De omgeving die gekozen wordt moet dus een continue *state space* hebben.

De tweede vraag is of er een continue *action space* is. Welke opties heeft de speler tijdens een spel pool? Hij kan de kracht F en ook de hoek α waaronder hij speelt bepalen. De hoek α waaronder gespeeld wordt bepaald door een evenwijdige lijn met de rand van de pooltafel die snijd met de witte bal. De hoek wordt in graden uitgedrukt dus $\alpha \in [0, 360[$. De kracht F kan uitgedrukt worden in een percentage in de simulatie dus $F \in [0,100]$. Ook hier kan er gesproken worden van een discrete *action space*, maar net zoals de *state space* is het niet mogelijk om dit uitputbaar te behandelen. Er wordt dus ook geopteerd voor een continue *action space*.

Er zijn een aantal omgevingen [40] die aan deze voorwaardes voldoen, maar er wordt hier gekozen voor de 'Pendulum-v0' omgeving. Deze omgeving is relatief simpel, waardoor er een model snel getraind op kan worden. Dit geeft de kans om een aantal verschillende modellen te testen, zelfs onder de tijdsdruk van de stage.

De 'Pendulum-v0' omgeving kan omschreven worden als het MDP in Vergelijking 19.

$$S = [\theta, v] | \theta \epsilon [-\pi, \pi], v \epsilon [-8,8]$$
$$A = F \, \epsilon \, [-2,2]$$
$$R = \, -(\theta^2 + 0.1\theta^2 + 0.001F^2)$$

Vergelijking 19 MDP Pendulum-v0 omgeving wiskundig omschreven

S is de *state space*, die twee variabelen bevat, de hoek waarop de pendel zich op dat moment bevindt, en de snelheid waarmee de pendel zich beweegt. A, de *action space* is een de kracht die het gewricht uitoefent op de pendel. R is de beloningsfunctie.

Het doel van de omgeving is het omhoog zwieren van de pendel, het bereiken van θ = 0, met zo weinig mogelijk kracht F en de pendel deze positie te laten behouden. De omgeving start met een willekeurige hoek en een willekeurige snelheid. Er is geen terminale *state*. [41]

3.3.2 Het creëren van de agent

Om de *agent* te creëren, zal er gebruik gemaakt worden van de 'keras-rl' *library*. Deze *library* maakt het bouwen van een *agent* simpeler, omdat de implementatie van het *reinforcement learning* algoritme niet zelf geschreven moet worden. Er moeten enkele componenten gecreëerd worden die dan aan een klasse van 'keras-rl' gegeven kan worden, die dan het updaten van de netwerken en het trainingsproces op zich neemt.

Allereerst moet het correcte algoritme dat toepasbaar is op de situatie gekozen worden. In onderdeel 3.3.1 is er bepaald dat het gaat over continue *action* en *state spaces*. Het algoritme dat we kiezen moet hier dus om mee kunnen gaan. DDPG heeft een implementatie in 'keras-rl', er wordt dus voor dit algoritme gekozen.

De eerste stap is het samenstellen van het *actor* model. Het model wordt gedefinieeërd via de 'Keras' API. Voor het *actor* model kan er gebruik gemaakt worden van de '*Sequential*' klasse. Dan moet de input gedefinieerd worden. De input is de huidige *state* en heeft de vorm van de *observation space*. Om de input laag te definiëren moet er ook rekening gehouden worden met de *window length*. Deze bepaalt hoeveel observaties het model verwacht. [42]

Hierop volgen twee volledige geconnecteerde '*Dense*' lagen, met elk een '*LeakyReLU*' activatie laag. De eerste van deze twee lagen bevat 400 neuronen, en de tweede 300. De output laag is ook een volledig geconnecteerde '*Dense*' laag. Het aantal neuronen in deze laag wordt bepaald door de hoeveelheid verschillende types actie die ondernomen kunnen worden.

Het uiteindelijke *actor* model bevat 122,201 trainbare parameters of gewichten.

In code ziet dit er als volgt uit:

```
observation_space = observation_space
action_space = action_space

nb_actions = action_space.shape[0]

actor_model = Sequential()
actor_model.add(Flatten(input_shape=(1,) + observation_space.shape))
actor_model.add(Dense(400))
actor_model.add(LeakyReLU())
actor_model.add(Dense(300))
actor_model.add(LeakyReLU())
actor_model.add(Dense(self.nb_actions))
actor_model.add(LeakyReLU())
```

Het *critic* model heeft ongeveer dezelfde opbouw, er zijn enkel verschillen in de input en output lagen. Omwille van deze verschillen kan er niet gebruik gemaakt worden van het '*Sequential*' model, maar moet er gebruik gemaakt van de '*Functional'* API.

Net zoals bij de *actor* is het begin van de *critic* de input laag, of in dit geval lagen. Eerst, wordt de actie input laag bepaald. Deze heeft dezelfde hoeveelheid neuronen als de output laag van de *actor*. Dan wordt de observatie input gedefinieerd. Deze is dezelfde als de input laag van het actor model en wordt dus ook geplet. De daarop volgende laag is de concatenatie laag, die de twee input lagen aan elkaar rijgt. Dan komen de verborgen lagen en de output laag. Dan wordt het model opgebouwd door in- en outputs aan de 'Model' klasse van 'Keras' te geven. Het *critic* netwerk heeft 122,601 trainbare parameters, een kleine hoeveelheid meer door de grotere inputlaag.

In code ziet het gebruik van de functionele API er als volgt uit:

```
action_input = Input(shape=(self.nb_actions,), name='action_input')
observation_input = Input(shape=(1,) + observation_space.shape, name='observation_input')
flattened_observation = Flatten()(observation_input)
x = Concatenate()([action_input, flattened_observation])
```

```
x = Dense(400)(x)
x = LeakyReLU()(x)
x = Dense(300)(x)
x = LeakyReLU()(x)
x = Dense(1)(x)
x = LeakyReLU()(x)
critic_model = Model(inputs=[action_input, observation_input], outputs=x)
```

Het volgende wat er moet gebeuren is de *replay buffer* en het willekeurig proces initialiseren. Als willekeurig proces wordt een Ornstein-Uhlenbeck-proces gebruikt. Hierna kan de *agent* worden samengesteld met al zijn parameters. De laatste stap is het compileren van de *agent* met de 'Adam' *optimizer*. Voor het *actor* netwerk is de *learning rate* ingesteld op 10^{-4} en voor de *critic* is de *learning rate* op 10^{-3}. De metriek is *mean squared error*, of 'mse'. Een andere belangrijke variabele zijn de *warmup steps*. Het netwerk leert door de *replay buffer* aan te spreken, maar er moet dus wel al data aanwezig zijn in die buffer. Tijdens de *warmup steps* worden er willekeurige waardes voor de acties gebruikt om een kleine hoeveelheid data in de buffer te plaatsen.

Deze laatste stappen zien er als volgt uit in code:

```
memory = SequentialMemory(limit=1000000, window_length=1)
random_process = OrnsteinUhlenbeckProcess(size=self.nb_actions, theta=.15, mu=0.,
sigma=.3)
ddpg = DDPGAgent(nb_actions=self.nb_actions, actor=self.actor_model,
critic=self.critic_model, critic_action_input=action_input, nb_steps_warmup_actor=100,
nb_steps_warmup_critic=100, memory=memory, random_process=random_process,
gamma=.99, target_model_update=0.001, batch_size=64)
ddpg.compile((Adam(lr=0.0001, clipnorm=1.), Adam(lr=0.001, clipnorm=1.)), metrics=['mse'])
```

Bijlage B: DDPG_agent.py toont de volledige code van de gecreëerde agent. Vanaf dit moment kan de agent getraind worden. Bijlage A: train.py toont hoe de trainingscode eruit ziet in het project. Hier wordt de belangrijkste lijn even toegelicht.

```
agent.ddpg.fit(env, nb_steps=1000000, visualize=True,nb_max_episode_steps=200, verbose
=1)
```

Deze lijn is degene die het trainingsproces start. Er wordt hier een omgeving aan meegegeven, hoeveel stappen de training mag duren, en hoelang een aflevering mag duren. De 'verbose' paramater stelt in hoeveel informatie over het trainingsproces wordt gegeven. De parameters voor de netwerken zijn over genomen uit het originele onderzoek. [35]

3.3.3 Breedte en diepte netwerk

Er zijn weinig bronnen die aanduiden hoeveel lagen, of hoe diep, en hoeveel neuronen, of hoe breed, een netwerk moet zijn om bepaalde doelen te kunnen bereiken. De bronnen die er wel zijn duiden vaak op de netwerken die gebruikt worden bij *supervised learning*.

Om dit te onderzoeken, zullen er twee extra *agents* gemaakt worden. De eerste is 'DDPGWideAgent' te vinden in Bijlage C. Deze agent vervangt het aantal neuronen in de verborgen lagen. De eerste laag bestaat uit 2000 neuronen en de tweede uit 1000 neuronen.

Ten tweede is er de ‘DDPGDeepAgent’ te vinden in Bijlage D. Deze agent verdubbelt het aantal lagen in het netwerk, maar brengt minder neuronen per laag naar de tafel. Elke laag bestaat uit 128 neuronen.

Alle andere instellingen blijven gelijk over de drie agents. De omgeving wordt bij elke start geseed, en zal ook dezelfde willekeurige getallen geven voor de startpositie en startsnelheid van de pendel.

3.4 Resultaten

Het trainingsproces is uitgevoerd met een standaard TensorFlow installatie, zonder GPU, op een systeem met een Intel i5 3230m met een 2.6 GHz kloksnelheid. Na het trainen van de drie modellen, kunnen de bereikte resultaten gevonden worden in Tabel 5.

	Paper	**Deep**	**Wide**
Gemiddelde beloning tijdens training	-371,68	-368,03	**-308,68**
Hoogste beloning tijdens training	-1,05	**-0,26**	-0,39
Gemiddelde beloning tijdens testen	**-132,39**	-182,42	-141,26
Hoogste beloning tijdens testen	**-0,76**	-116,66	-7,5
Trainingstijd in seconden	3494,573	3672,696	18431,364
Trainbare parameters *actor*	122201	50177	2010001
Trainbare parameters *critic*	122601	50305	2012001

Tabel 5 Resultaten van de drie agents na 100000 stappen training en 10 afleveringen van 100 stappen test met visualisatie actief.

Uit de resultaten kunnen we afleiden dat de *agent* zoals hij beschreven staat in de paper het beste resultaat oplevert tijdens het testen. Tijdens het trainen zijn er echter andere resultaten. Het is natuurlijk mogelijk dat de hoogste beloning gehaald door de *deep agent* een gelukkige samenloop van omstandigheden was, maar het interessantere resultaat is dat de gemiddelde beloning van de *wide agent* gespreid over 500 afleveringen een betere waarde heeft dan de andere twee.

In Bijlage E: Grafieken trainingsproces is te zien dat de *wide agent* stabielere resultaten oplevert, wat in sommige omgevingen zelfs belangrijker kan zijn dan een top resultaat neerzetten. Ook afleidbaar uit de grafieken is dat de *wide agent* een langere periode nodig heeft om te leren, en in het begin veel langer slechte resultaten toonde dan de andere *agents*. Dit betekent dat de positieve resultaten behaald door de *wide agent* ook gemiddeld hoger liggen.

De test resultaten Bijlage F: Grafieken testproces tonen aan dat de *deep agent* enorm instabiel is, en veel slechtere resultaten oplevert. Het aantal lagen uitbreiden lijkt geen positief effect te hebben, zeker niet als er in rekening gebracht dat het trainen van ongeveer 40 procent van het aantal parameters dezelfde tijd vraagt.

Wide agent toont hier ook weer een stabieler resultaat. Met deze resultaten lijkt het aan te raden om het aantal neuronen in de lagen te verhogen als er geen goed resultaat opgeleverd wordt door. Dit heeft echter wel een effect op de trainingstijd, omdat de netwerken vele malen zwaarder zijn, en ook meer parameters hebben om te trainen.

Als de omgeving oplosbaar is met de *agent* beschreven zoals in de paper is dit het aan te raden pad. *Wide agents* zijn de volgende opties, en zijn zeker interessant in gecompliceerde omgevingen. De afweging kan best gemaakt worden rond het aantal *inputs*. In deze omgeving waren er drie *inputs*, in de uiteindelijke pool omgeving, na pletten, 32.

3.5 Toekomstig werk

Limieten in tijd en kennis belette dat de volgende mogelijkheden onderzocht werden, maar zouden nog interessante opties zijn om te bekijken.

DDPG bleek niet snel genoeg te trainen om binnen de perken van de stage 'pool' te leren, zelfs als er maar een minima aan regels gesteld werden. Extra tijd, en een beter ontworpen omgeving, zouden mogelijk wel voor convergentie kunnen zorgen.

Een tweede optie die bekeken kan worden is een implementatie van het *Asynchronous Advantage Actor-Critic* of A3C algoritme. [43] Dit was niet beschikbaar in het 'keras-rl' pakket op het moment van dit schrijven, dus deze implementatie zou van de grond op geschreven moeten worden, wat een stap te ver was voor de python kennis die aanwezig was.

Verder kan de vergelijking uitgewerkt in 3.3.3 en 3.4 verder uitgewerkt worden door een vergelijking te doen waarbij de *wide agent* meer trainingstijd krijgt om al zijn parameters te kunnen finetunen. In omgevingen met grotere en ingewikkeldere *states* kan dit voor veel betere resultaten zorgen dan een klein netwerk.

3.6 Reflectie

Het onderzoek is vrij moeizaam op gang gekomen. Bepalen wat belangrijk genoeg was om te bespreken en wat geen meerwaarde vormde voor het onderzoek was niet makkelijk. Daarnaast was het onderzoek ook niet simpel, er kwam veel relatief zware wiskunde aan de pas, en ook hogere niveau concepten die in de opleiding niet aan bod kwamen.

Artificiële intelligentie is een enorm breed onderwerp, en zelfs het beperken tot *reinforcement learning* doet het niet krimpen. Aan de start van het onderzoek zijn er een grote hoeveelheid papers verwerkt met de bedoeling het onderzoek daarna te kunnen uitschrijven. Dit bleek geen goed systeem te zijn, omdat de kennis er wel was, maar het moeilijk was deze te verbinden met de juiste bronnen.

Wat door mij als het moeilijkst werd ervaren, was de essentie vinden in de onderwerpen die werden aangesneden. Uiteindelijk heb ik een aantal belangrijke concepten die in de meeste papers werden aangehaald dieper besproken. Hierna wou ik een deel van de geschiedenis van het algoritme dat gebruikt wordt door de *agents* bespreken, zodat de opbouw in *deep reinforcement learning* duidelijk wordt voor eenieder die leest.

Uiteindelijk is er nog geëxperimenteerd rond het aantal lagen en neuronen, waar ik geen recente bronnen over kon vinden, wat ik dus nog wel een waardevolle toevoeging vond aan het onderzoek.

Dit onderzoek probeert een balans te vinden tussen de technische uitleg, en de concepten uitleggen op een vereenvoudigde manier zodat mensen met een minder technische achtergrond ook waarde kunnen hechten aan deze paper. Ik had ook het doel met dit onderzoek om de manier om zelf de *agent* te bouwen duidelijk te maken voor ontwikkelaars. Hierachter zit ook de keuze om te steunen op de ‘Keras’ bibliotheek, die de zwaardere concepten afschermt achter simpelere API functies.

Er zijn een aantal dieptepunten geweest tijdens het uitwerken van dit onderzoek, maar aan de eindstreep staande, ben ik tevreden over hetgeen dat ik heb kunnen afleveren.

Conclusie

Deep reinforcement learning is een onderwerp dat zich naar de voorgrond heeft gewerkt in de afgelopen paar jaar. Na een aantal experimenten uitgevoerd te hebben, is er nog steeds een limiet aan de algoritmes die er op dit moment beschikbaar zijn om gecompliceerde omgevingen te verwerken.

Dit limiet is voelbaar op twee plaatsen: allereerst de enorme trainingstijd die vereist is zodat de algoritmes stabiel kunnen trainen, en het tweede is de kans dat zelf na een lange periode trainen, de algoritmes geen convergentie kunnen garanderen. Onderzoek wijst wel uit dat deze algoritmes kunnen convergeren, en goede resultaten opleveren in gelimiteerde omgevingen.

Zo bleek ook het spel 'pool' als omgeving te gecompliceerd om te convergeren met het gekozen algoritme DDPG. Het is echter wel mogelijk, in een beperkte mate, om via objectdetectie een computer als scheidsrechter te kunnen gebruiken. Dit concept kan, met genoeg trainingsdata en kracht toegepast worden op de meeste types sport.

Deep reinforcement learning is zeker toepasbaar om omgevingen met bepaalde limieten, en heeft een plaats in de huidige groeimarkt die artificiële intelligentie is, maar is mogelijk nog te jong om los te laten in de realiteit, wat de ingewikkeldste omgeving ooit is.

Bibliografie

[1] „B-Inspired,” Bewire, [Online]. Available: https://bewire.be/b-inspired/. [Geopend 13 Maart 2019].

[2] „Bewire,” Bewire, [Online]. Available: https://bewire.be/. [Geopend 13 Maart 2019].

[3] K. Van Bruystegem, „44 Belgian Companies can call themselves a ‘Great Place to Work®’,” Great Place To Work, 20 Maart 2019. [Online]. Available: https://www.greatplacetowork.be/en/blog/blog/44-belgian-companies-can-call-themselves-a-great-place-to-work. [Geopend 5 April 2019].

[4] „C4J,” C4J. [Online]. [Geopend 14 Maart 2019].

[5] „Evance,” Evance, [Online]. Available: https://evance.be/. [Geopend 14 Maart 2019].

[6] „Codrigo,” Codrigo, [Online]. Available: https://www.codrigo.be/. [Geopend 14 Maart 2019].

[7] „Dots & Arrows,” Dots & Arrows, [Online]. Available: https://dotsandarrows.be/. [Geopend 14 Maart 2019].

[8] „Trase,” Trase, [Online]. Available: https://www.trase.be/. [Geopend 14 Maart 2019].

[9] „Appmind,” Appmind, [Online]. Available: http://www.appmind.be. [Geopend 14 03 2019].

[10] J. Brownlee, „What is Deep Learning?,” Machine Learning Mastery, 16 Augustus 2016. [Online]. Available: https://machinelearningmastery.com/what-is-deep-learning/. [Geopend 01 April 2019].

[11] A. Schirock, LEGO Mindstorms EV3: Objectherkenning als een agent, PXL, 2018.

[12] J. Redman en A. Farhadi, *YOLOv3: An Incremental Improvement,* 2018.

[13] F. Chollet, „User experience design for APIs,” Keras, 21 November 2017. [Online]. Available: https://blog.keras.io/user-experience-design-for-apis.html. [Geopend 25 April 2019].

[14] „Keras Documentation,” Keras, [Online]. Available: https://keras.io/. [Geopend 25 April 2019].

[15] „Differentiate CUDA Cores(NVIDIA) and Stream processor(ATI/AMD),” Superuser, 7 Maart 2016. [Online]. Available: https://superuser.com/questions/649019/differentiate-cuda-coresnvidia-and-stream-processorati-amd. [Geopend 29 April 2019].

[16] A. Verma, „CUDA Cores vs Stream Processors Explained,” Graphics Card Hub, 27 Mei 2018. [Online]. Available: https://graphicscardhub.com/cuda-cores-vs-stream-processors/. [Geopend 29 April 2019].

[17] T. Dettmers, „Which GPU(s) to Get for Deep Learning: My Experience and Advice for Using GPUs in Deep Learning,” Tim Dettmers, 04 Maart 2019. [Online]. Available: https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/. [Geopend 29 April 2019].

[18] *NVIDIA TESLA V100 GPU,* 2017.

[19] AlexeyAB, „Yolo-v3 and Yolo-v2 for Windows and Linux," 23 April 2019. [Online]. Available: https://github.com/AlexeyAB/darknet#how-to-train-tiny-yolo-to-detect-your-custom-objects. [Geopend 10 Mei 2019].

[20] Y. Martens, Smart Pool Table: Deep learning en reinforcement learning vergelijken, Hasselt: PXL, 2019.

[21] M. Kovalovs, „Pool," 26 Maart 2018. [Online]. Available: https://github.com/max-kov/pool. [Geopend 20 Mei 2019].

[22] A. Gulli en S. Pal, Deep Learning with Keras, Packt Publishing, 2017.

[23] I. Salian, „SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?," Nvidia, 2 Augustus 2018. [Online]. Available: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/. [Geopend 8 Maart 2019].

[24] „A Beginner's Guide to Deep Reinforcement Learning," Skymind, [Online]. Available: https://skymind.ai/wiki/deep-reinforcement-learning. [Geopend 8 Maart 2019].

[25] J. Ravi, „Environment as a Markov Decision Process (MDP)," Pluralsight, 6 Juli 2018. [Online]. Available: https://app.pluralsight.com/player?course=understanding-algorithms-reinforcement-learning&author=janani-ravi&name=fcd9d1fb-47a1-4aac-ac9c-1a896e59a798&clip=5&mode=live. [Geopend 8 Maart 2019].

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonogluo, D. Wierstra en M. Riedmiller, „Playing Atari with Deep Reinforcement Learning," Deepmind Technologies, 19 December 2013. [Online]. Available: https://arxiv.org/pdf/1312.5602.pdf. [Geopend 8 Maart 2019].

[27] J. Ravi, „Demo: Q-Learning to Balance a Pole on a Cart," Pluralsight, 6 Juli 2018. [Online]. Available: https://app.pluralsight.com/player?course=understanding-algorithms-reinforcement-learning&author=janani-ravi&name=bef89f6b-37ce-412e-9485-d394808165a2&clip=4&mode=live. [Geopend 22 Maart 2019].

[28] M. Ashraf, „Reinforcement Learning Demystified: Markov Decision Processes (Part 1)," Towards Data Science, 11 April 2018. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690. [Geopend 22 Maart 2019].

[29] M. Ashraf, „Reinforcement Learning Demystified: Markov Decision Processes (Part 2)," Towards Data Science, 20 April 2018. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-2-b209e8617c5a. [Geopend 15 April 2019].

[30] S. Pitis, *Rethinking the Discount Factor in Reinforcement Learning:,* Toronto, 2019.

[31] M. Ashraf, „Reinforcement Learning Demystified: A Gentle Introduction," Towards Data Science, 7 April 2018. [Online]. Available: https://towardsdatascience.com/reinforcement-learning-demystified-36c39c11ec14. [Geopend 4 April 2019].

[32] J. Greaves, „Understanding RL: The Bellman Equations," [Online]. Available: https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/. [Geopend 15 April 2019].

[33] J. A. Snyman en D. N. Wilke, Practical Mathematical Optimization, Springer, 2018.

[34] P. Liao, N. Landy en N. Katz, „Deep Cue Learning: A Reinforcement Learning Agent for Playing Pool," Stanford Univerity, 2018. [Online]. Available: http://cs229.stanford.edu/proj2018/report/249.pdf. [Geopend 6 Mei 2019].

[35] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver en D. Wierstra, „Continuous Control With Deep Reinforcement Learning," Google Deepmind, 29 Februari 2016. [Online]. Available: https://arxiv.org/pdf/1509.02971.pdf. [Geopend 8 Mei 2019].

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg en D. Hassabis, „Human-level control through deep reinforcement," Deepmind, 26 Februari 2015. [Online]. Available: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf. [Geopend 7 Mei 2019].

[37] R. Gilman, „Intuitive RL: Intro to Advantage-Actor-Critic (A2C)," Hackernoon, 9 Januari 2018. [Online]. Available: https://hackernoon.com/intuitive-rl-intro-to-advantage-actor-critic-a2c-4ff545978752. [Geopend 8 Mei 2019].

[38] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra en M. Riedmiller, „Deterministic Policy Gradient Algorithms," Deepmind Technologies, 21 Juni 2014. [Online]. Available: http://proceedings.mlr.press/v32/silver14.pdf. [Geopend 8 Mei 2019].

[39] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru en Y. Tassa, „Safe Exploration in Continuous Action Spaces," 26 Januari 2018. [Online]. Available: https://arxiv.org/pdf/1801.08757.pdf. [Geopend 8 Mei 2019].

[40] „Classic Control," OpenAI Gym, [Online]. Available: https://gym.openai.com/envs/#classic_control. [Geopend 8 Mei 2019].

[41] C. Wen, „Pendulum v0," OpenAI Gym, 14 April 2018. [Online]. Available: https://github.com/openai/gym/wiki/Pendulum-v0. [Geopend 8 Mei 2019].

[42] B. Klebel, „Keras-RL adding extra dimension to RGB input," 3 Augustus 2018. [Online]. Available: https://github.com/keras-rl/keras-rl/issues/229#issuecomment-410226068. [Geopend 9 Mei 2019].

[43] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver en K. Kavukcuoglu, „Asynchronous Methods for Deep Reinforcement Learning," Google Deepmind, 16 Juni 2016. [Online]. Available: https://arxiv.org/pdf/1602.01783.pdf. [Geopend 10 Mei 2019].

[44] *Skinner Box.* [Art].

[45] R. S. Sutton en A. G. Barto, Reinforcement Learning: An Introduction, Cambridge, Massachusetts: The MIT Press, 2017.

[46] J. Redmon en A. Farhadi, *YOLOv3: An Incremental Improvement,* 2018.

[47] „Pendulum-v0," OpenAI, [Online]. Available: https://gym.openai.com/envs/Pendulum-v0/. [Geopend 25 April 2019].

[48] „Handreach-v0," OpenAI, [Online]. Available: https://gym.openai.com/envs/HandReach-v0/. [Geopend 26 April 2019].

[49] „GoPro HERO 5 Black," bol.com, [Online]. Available: https://www.bol.com/nl/p/gopro-hero-5-black/9200000065357395/. [Geopend 26 April 2019].

[50] D. Coleman, „GoPro HERO6 and HERO5 Built-In Fisheye Distortion Correction: Linear FOV," Have Camera Will Travel, 2016. [Online]. Available: https://havecamerawilltravel.com/gopro/gopro-fov-linear/. [Geopend 26 April 2019].

[51] W. Whitney, Artist, *Convolutional Network.* [Art]. Wikimedia Commons, 2014.

[52] ASUS, Artist, *ROG-STRIX-RTX2080TI-O11G-GAMING.* [Art]. ASUS, 2018.

[53] O. Gym, Artist, *Copy-v0.* [Art]. OpenAI Gym.

Bijlagen

A. Bijlage A: training.py

```
1.  from DDPG_paper import DDPGAgentPaper
2.
3.  import gym
4.
5.  ENV_NAME = 'Pendulum-v0'
6.
7.
8.  def main():
9.      env = gym.make(ENV_NAME)
10.     observation_space = env.observation_space
11.     action_space = env.action_space
12.
13.     load = False
14.
15.     agent = DDPGAgentPaper(observation_space=observation_space, action_space=action_
    space)
16.     if load:
17.         agent.ddpg.load_weights('ddpg_agent_{}_weights.h5f'.format(ENV_NAME))
18.     agent.ddpg.fit(env, nb_steps=1000000, visualize=True, nb_max_episode_steps=200,
    verbose=1)
19.     agent.ddpg.save_weights('ddpg_agent_{}_weights.h5f'.format(ENV_NAME), overwrite=
    True)
20.     agent.ddpg.test()
21.
22.
23. if __name__ == "__main__":
24.     main()
```

B. Bijlage B: DDPG_paper.py

```
1.  from keras import Input, Model
2.  from keras.layers import Dense, LeakyReLU, Flatten, Concatenate
3.  from keras.models import Sequential
4.  from keras.optimizers import Adam
5.  from rl.agents import DDPGAgent
6.  from rl.memory import SequentialMemory
7.  from rl.random import OrnsteinUhlenbeckProcess
8.
9.
10. class DDPGAgentPaper:
11.
12.     def __init__(self, observation_space, action_space):
13.         self.observation_space = observation_space
14.         self.action_space = action_space
15.
16.         self.nb_actions = action_space.shape[0]
17.
18.         self.actor_model = Sequential()
19.         self.actor_model.add(Flatten(input_shape=(1,) + observation_space.shape))
20.         self.actor_model.add(Dense(400))
21.         self.actor_model.add(LeakyReLU())
22.         self.actor_model.add(Dense(300))
23.         self.actor_model.add(LeakyReLU())
24.         self.actor_model.add(Dense(self.nb_actions))
25.         self.actor_model.add(LeakyReLU())
26.
27.         self.actor_model.summary()
28.
29.         action_input = Input(shape=(self.nb_actions,), name='action_input')
30.         observation_input = Input(shape=(1,) + observation_space.shape, name='observ
    ation_input')
31.         flattened_observation = Flatten()(observation_input)
32.         x = Concatenate()([action_input, flattened_observation])
33.         x = Dense(400)(x)
34.         x = LeakyReLU()(x)
35.         x = Dense(300)(x)
36.         x = LeakyReLU()(x)
37.         x = Dense(1)(x)
38.         x = LeakyReLU()(x)
39.         self.critic_model = Model(inputs=[action_input, observation_input], outputs=
    x)
40.
41.         self.critic_model.summary()
42.
43.         memory = SequentialMemory(limit=1000000, window_length=1)
44.         random_process = OrnsteinUhlenbeckProcess(size=self.nb_actions, theta=.15, m
    u=0., sigma=.3)
45.
46.         self.ddpg = DDPGAgent(nb_actions=self.nb_actions, actor=self.actor_model,
47.                               critic=self.critic_model, critic_action_input=action_i
    nput, nb_steps_warmup_actor=100,
48.                               nb_steps_warmup_critic=100, memory=memory, random_proc
    ess=random_process, gamma=.99,
49.                               target_model_update=0.001, batch_size=64)
50.         self.ddpg.compile((Adam(lr=0.0001, clipnorm=1.), Adam(lr=0.001, clipnorm=1.)
    ), metrics=['mse'])
```

C. Bijlage C: DDPG_wide.py

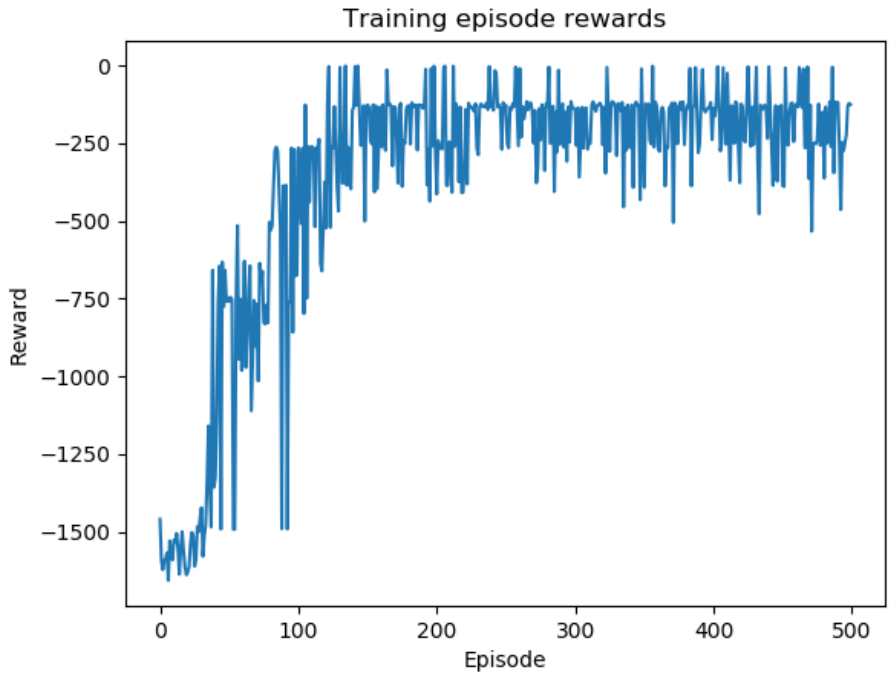
```
1.  from keras import Input, Model
2.  from keras.layers import Dense, LeakyReLU, Flatten, Concatenate
3.  from keras.models import Sequential
4.  from keras.optimizers import Adam
5.  from rl.agents import DDPGAgent
6.  from rl.memory import SequentialMemory
7.  from rl.random import OrnsteinUhlenbeckProcess
8.
9.
10. class DDPGAgentWide:
11.
12.     def __init__(self, observation_space, action_space):
13.         self.observation_space = observation_space
14.         self.action_space = action_space
15.
16.         self.nb_actions = action_space.shape[0]
17.
18.         self.actor_model = Sequential()
19.         self.actor_model.add(Flatten(input_shape=(1,) + observation_space.shape))
20.         self.actor_model.add(Dense(2000))
21.         self.actor_model.add(LeakyReLU())
22.         self.actor_model.add(Dense(1000))
23.         self.actor_model.add(LeakyReLU())
24.         self.actor_model.add(Dense(self.nb_actions))
25.         self.actor_model.add(LeakyReLU())
26.
27.         self.actor_model.summary()
28.
29.         action_input = Input(shape=(self.nb_actions,), name='action_input')
30.         observation_input = Input(shape=(1,) + observation_space.shape, name='observ
    ation_input')
31.         flattened_observation = Flatten()(observation_input)
32.         x = Concatenate()([action_input, flattened_observation])
33.         x = Dense(2000)(x)
34.         x = LeakyReLU()(x)
35.         x = Dense(1000)(x)
36.         x = LeakyReLU()(x)
37.         x = Dense(1)(x)
38.         x = LeakyReLU()(x)
39.         self.critic_model = Model(inputs=[action_input, observation_input], outputs=
    x)
40.
41.         self.critic_model.summary()
42.
43.         memory = SequentialMemory(limit=1000000, window_length=1)
44.         random_process = OrnsteinUhlenbeckProcess(size=self.nb_actions, theta=.15, m
    u=0., sigma=.3)
45.
46.         self.ddpg = DDPGAgent(nb_actions=self.nb_actions, actor=self.actor_model,
47.                               critic=self.critic_model, critic_action_input=action_i
    nput, nb_steps_warmup_actor=100,
48.                               nb_steps_warmup_critic=100, memory=memory, random_proc
    ess=random_process, gamma=.99,
49.                               target_model_update=0.001, batch_size=64)
50.         self.ddpg.compile((Adam(lr=0.0001, clipnorm=1.), Adam(lr=0.001, clipnorm=1.)
    ), metrics=['mse'])
```

D. Bijlage D: DDPG_deep.py

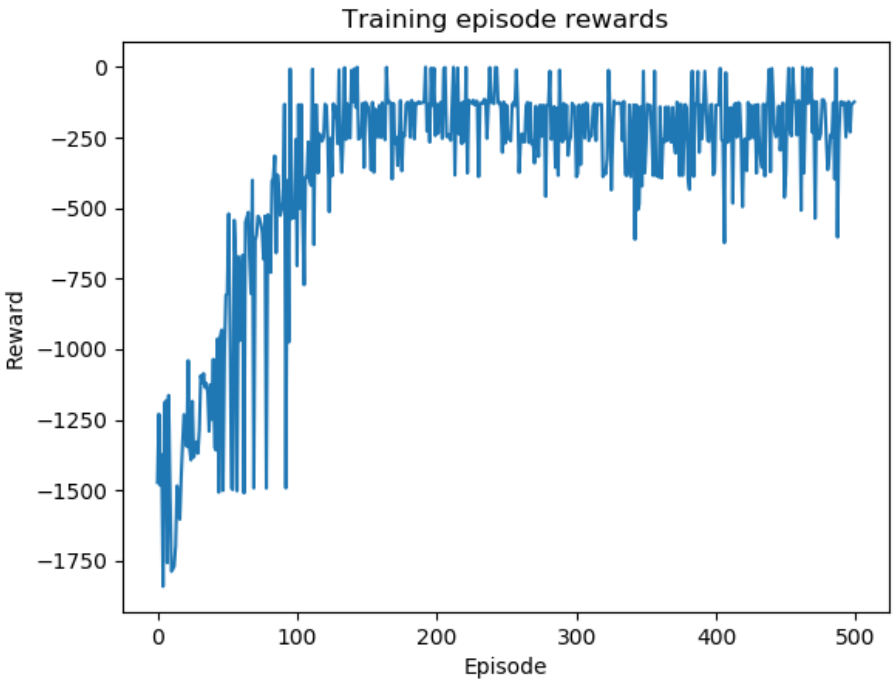
```
1.  from keras import Input, Model
2.  from keras.layers import Dense, LeakyReLU, Flatten, Concatenate
3.  from keras.models import Sequential
4.  from keras.optimizers import Adam
5.  from rl.agents import DDPGAgent
6.  from rl.memory import SequentialMemory
7.  from rl.random import OrnsteinUhlenbeckProcess
8.
9.
10. class DDPGAgentPaper:
11.
12.     def __init__(self, observation_space, action_space):
13.         self.observation_space = observation_space
14.         self.action_space = action_space
15.
16.         self.nb_actions = action_space.shape[0]
17.
18.         self.actor_model = Sequential()
19.         self.actor_model.add(Flatten(input_shape=(1,) + observation_space.shape))
20.         self.actor_model.add(Dense(128))
21.         self.actor_model.add(LeakyReLU())
22.         self.actor_model.add(Dense(128))
23.         self.actor_model.add(LeakyReLU())
24.         self.actor_model.add(Dense(128))
25.         self.actor_model.add(LeakyReLU())
26.         self.actor_model.add(Dense(128))
27.         self.actor_model.add(LeakyReLU())
28.         self.actor_model.add(Dense(self.nb_actions))
29.         self.actor_model.add(LeakyReLU())
30.
31.         self.actor_model.summary()
32.
33.         action_input = Input(shape=(self.nb_actions,), name='action_input')
34.         observation_input = Input(shape=(1,) + observation_space.shape, name='observ
    ation_input')
35.         flattened_observation = Flatten()(observation_input)
36.         x = Concatenate()([action_input, flattened_observation])
37.         x = Dense(128)(x)
38.         x = LeakyReLU()(x)
39.         x = Dense(128)(x)
40.         x = LeakyReLU()(x)
41.         x = Dense(128)(x)
42.         x = LeakyReLU()(x)
43.         x = Dense(128)(x)
44.         x = LeakyReLU()(x)
45.         x = Dense(1)(x)
46.         x = LeakyReLU()(x)
47.         self.critic_model = Model(inputs=[action_input, observation_input], outputs=
    x)
48.
49.         self.critic_model.summary()
50.
51.         memory = SequentialMemory(limit=1000000, window_length=1)
52.         random_process = OrnsteinUhlenbeckProcess(size=self.nb_actions, theta=.15, m
    u=0., sigma=.3)
53.
54.         self.ddpg = DDPGAgent(nb_actions=self.nb_actions, actor=self.actor_model,
55.                               critic=self.critic_model, critic_action_input=action_i
    nput, nb_steps_warmup_actor=100,
56.                               nb_steps_warmup_critic=100, memory=memory, random_proc
    ess=random_process, gamma=.99,
57.                               target_model_update=0.001, batch_size=64)
```

```
58.         self.ddpg.compile((Adam(lr=0.0001, clipnorm=1.), Adam(lr=0.001, clipnorm=1.)
    ), metrics=['mse'])
```

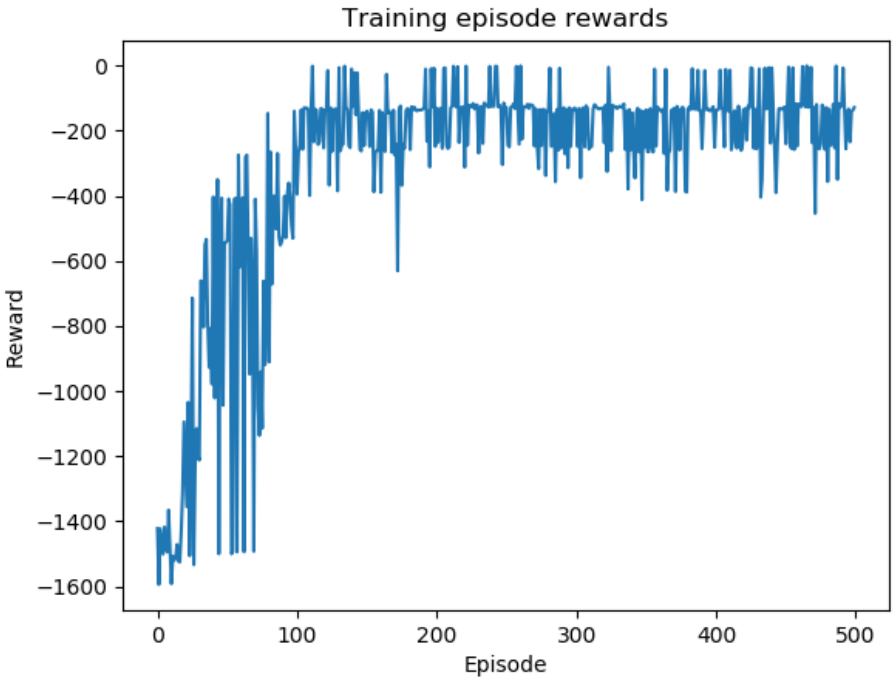
E. Bijlage E: Grafieken trainingsproces



Figuur 24 Grafiek leerproces van agent gebaseerd op de originele paper over een periode van 100000 stappen met visualisatie aan.

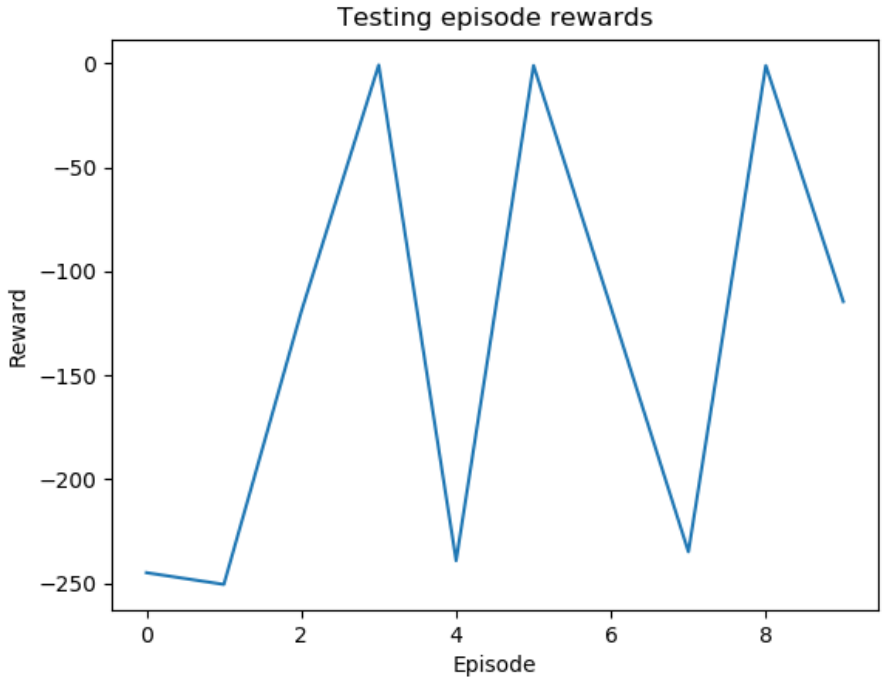


Figuur 25 Grafiek leerproces van deep agent over een periode van 100000 stappen met visualisatie aan.



Figuur 26 Grafiek leerproces van wide agent over een periode van 100000 stappen met visualisatie aan.

F. Bijlage F: Grafieken testproces



Figuur 28 Grafiek agent gebaseerd op paper over 10 test afleveringen van 200 stappen.

Figuur 27 Grafiek deep agent over 10 test afleveringen van 200 stappen.

Figuur 29 Grafiek wide agent over 10 test afleveringen van 200 stappen.

G. Bijlage G: datasplitter.py

```
1.  import glob, os, sys, getopt
2.
3.  def main(argv):
4.      output_location = ''
5.      input_dir = ''
6.
7.      try:
8.          opts, args = getopt.getopt(argv, "hi:o:", ["oloc=", "iloc="])
9.      except getopt.GetoptError:
10.         print('datasplitter -i <input_location> -o <path_from_darknet>')
11.         sys.exit(2)
12.     for opt, arg in opts:
13.         if opt == '-h':
14.             print('datasplitter -i <input_location> -o <path_from_darknet>')
15.             sys.exit()
16.         elif opt in ("-i", "--iloc"):
17.             input_dir = arg
18.         elif opt in ("-o", "--oloc"):
19.             output_location = arg
20.
21.     percentage_test = 10
22.
23.     file_train = open('train.txt', 'w')
24.     file_test = open('test.txt', 'w')
25.
26.     counter = 1
27.     index_test = round(100 / percentage_test)
28.
29.     for pathAndFilename in glob.iglob(os.path.join(input_dir, "*.jpg")):
30.
31.         title, ext = os.path.splitext(os.path.basename(pathAndFilename))
32.
33.         if counter == index_test:
34.             print('test: ' + pathAndFilename)
35.             counter = 1
36.             file_test.write(output_location + title + '.jpg' + '\n')
37.         else:
38.             print('train: ' + pathAndFilename)
39.             file_train.write(output_location + title + '.jpg' + '\n')
40.             counter = counter + 1
41.
42. if __name__ == "__main__":
43.     main(sys.argv[1:])
```

H. Bijlage H: detector.c train_detector aanpassingen

Veranderingen zijn gemarkeerd.

```
44. void train_detector(char *datacfg, char *cfgfile, char *weightfile, int *gpus, int n
    gpus, int clear)
45. {
46.     list *options = read_data_cfg(datacfg);
47.     char *train_images = option_find_str(options, "train", "data/train.list");
48.     char *backup_directory = option_find_str(options, "backup", "/backup/");
49.     FILE *file = fopen("output.txt", "w");
50.
51.     srand(time(0));
52.     char *base = basecfg(cfgfile);
53.     printf("%s\n", base);
54.     float avg_loss = -1;
55.     network **nets = calloc(ngpus, sizeof(network));
56.
57.     srand(time(0));
58.     int seed = rand();
59.     int i;
60.     for(i = 0; i < ngpus; ++i){
61.         srand(seed);
62. #ifdef GPU
63.         cuda_set_device(gpus[i]);
64. #endif
65.         nets[i] = load_network(cfgfile, weightfile, clear);
66.         nets[i]->learning_rate *= ngpus;
67.     }
68.     srand(time(0));
69.     network *net = nets[0];
70.
71.     int imgs = net->batch * net->subdivisions * ngpus;
72.     printf("Learning Rate: %g, Momentum: %g, Decay: %g\n", net->learning_rate, net-
    >momentum, net->decay);
73.     data train, buffer;
74.
75.     layer l = net->layers[net->n - 1];
76.
77.     int classes = l.classes;
78.     float jitter = l.jitter;
79.
80.     list *plist = get_paths(train_images);
81.     char **paths = (char **)list_to_array(plist);
82.
83.     load_args args = get_base_args(net);
84.     args.coords = l.coords;
85.     args.paths = paths;
86.     args.n = imgs;
87.     args.m = plist->size;
88.     args.classes = classes;
89.     args.jitter = jitter;
90.     args.num_boxes = l.max_boxes;
91.     args.d = &buffer;
92.     args.type = DETECTION_DATA;
93.     args.threads = 64;
94.
95.     pthread_t load_thread = load_data(args);
96.     double time;
97.     int count = 0;
98.     while(get_current_batch(net) < net->max_batches){
99.         if(l.random && count++%10 == 0){
100.                 printf("Resizing\n");
101.                 int dim = (rand() % 10 + 10) * 32;
102.                 if (get_current_batch(net)+200 > net->max_batches) dim = 608;
```

```
103.                    printf("%d\n", dim);
104.                    args.w = dim;
105.                    args.h = dim;
106.
107.                    pthread_join(load_thread, 0);
108.                    train = buffer;
109.                    free_data(train);
110.                    load_thread = load_data(args);
111.
112.                    #pragma omp parallel for
113.                    for(i = 0; i < ngpus; ++i){
114.                        resize_network(nets[i], dim, dim);
115.                    }
116.                    net = nets[0];
117.                }
118.                time=what_time_is_it_now();
119.                pthread_join(load_thread, 0);
120.                train = buffer;
121.                load_thread = load_data(args);
122.
123.                printf("Loaded: %lf seconds\n", what_time_is_it_now()-time);
124.
125.                time=what_time_is_it_now();
126.                float loss = 0;
127.        #ifdef GPU
128.                if(ngpus == 1){
129.                    loss = train_network(net, train);
130.                } else {
131.                    loss = train_networks(nets, ngpus, train, 4);
132.                }
133.        #else
134.                loss = train_network(net, train);
135.        #endif
136.                if (avg_loss < 0) avg_loss = loss;
137.                avg_loss = avg_loss*.9 + loss*.1;
138.
139.                i = get_current_batch(net);
140.
141.            fprintf(file, "%ld: %f, %f avg, %f rate, %lf seconds, %d images\n", get_c
    urrent_batch(net), loss, avg_loss, get_current_rate(net), what_time_is_it_now()-
    time, i*imgs);
142.            fflush(file);
143.
144.                printf("%ld: %f, %f avg, %f rate, %lf seconds, %d images\n", get_curr
    ent_batch(net), loss, avg_loss, get_current_rate(net), what_time_is_it_now()-
    time, i*imgs);
145.            if(i%100==0){
146.        #ifdef GPU
147.                    if(ngpus != 1) sync_nets(nets, ngpus, 0);
148.        #endif
149.                    char buff[256];
150.                    sprintf(buff, "%s/%s.backup", backup_directory, base);
151.                    save_weights(net, buff);
152.                }
153.            if(i%100 == 0){
154.        #ifdef GPU
155.                    if(ngpus != 1) sync_nets(nets, ngpus, 0);
156.        #endif
157.                    char buff[256];
158.                    sprintf(buff, "%s/%s_%d.weights", backup_directory, base, i);
159.                    save_weights(net, buff);
160.                }
161.                free_data(train);
162.            }
163.        #ifdef GPU
164.            if(ngpus != 1) sync_nets(nets, ngpus, 0);
```

```
165.        #endif
166.            char buff[256];
167.            sprintf(buff, "%s/%s_final.weights", backup_directory, base);
168.            save_weights(net, buff);
169.            fclose(file);
170.        }
```

I. Bijlage I: image_opencv.cpp show_image_cv aanpassingen

Veranderingen zijn gemarkeerd.

```
1.  VideoWriter* video;
2.
3.  int show_image_cv(image im, const char* name, int ms)
4.  {
5.     Mat m = image_to_mat(im);
6.     imshow(name, m);
7.     {
8.
9.         if(video == NULL){
10.            const char* output_name = "predictions.avi";
11.            video = new VideoWriter(output_name, VideoWriter::fourcc('M','J','P','G')
    ,30, Size(im.w,im.h));
12.            printf("\n DST output_video = %s  \n", output_name);
13.        }
14.
15.        video->write(m);
16.         printf("\n cvWriteFrame \n");
17.    }
18.    int c = waitKey(ms);
19.    if (c != -1) c = c%256;
20.    return c;
21. }
```