

# Parkoers navigatie door AI

Robbe Van de Keere

Wout Verdyck

## Abstract

In dit onderzoek gebruiken we reinforcement learning (RL) en genetische algoritmen (GA) voor het trainen van intelligente agents in een gesimuleerde tweedimensionale omgeving. We gebruiken Minecraft als game om agents te leren hoe ze een parkoers moeten voltooien door vooruit te lopen en te springen van blok tot blok, zonder te vallen. We vergelijken verschillende strategieën voor het ontwerpen van de policy en reward functies van de agents en evalueren hun prestaties en toepasbaarheid in verschillende scenario's. Ons doel is om niet alleen de efficiëntie van de agents te verbeteren, maar ook om inzicht te krijgen in de onderliggende mechanismen die hun gedrag beïnvloeden. De resultaten van dit onderzoek dragen bij aan de groeiende kennis over de toepassing van RL en GA in game-omgevingen. Door de optimale aanpak te identificeren, kunnen we beter begrijpen hoe agents met continue variabelen omgaan, wat waardevol is voor de verdere ontwikkelingen in kunstmatige intelligentie en spel technologie. We delen onze methodologie, presenteren experimentele resultaten en bespreken de implicaties van onze bevindingen.

## 1 Introductie

Reinforcement learning (RL) en genetische algoritmen (GA) bieden veelbelovende perspectieven voor toepassingen in de robotica. Het voorstellen van een robot als een agent, waarbij een reward functie wordt gedefinieerd, maakt het mogelijk om de robot diverse taken autonoom en zo efficiënt mogelijk te laten uitvoeren. In een gesimuleerde continue wereld kan de herhaaldelijke uitvoering van dezelfde actie tot verschillende resultaten leiden en moet de agent rekening houden met specifieke spel-mechanismen en vertragingen. Het ontwerpen van een dergelijke agent wordt aanzienlijk complex. Een goede policy en reward functie zoeken kan dan ook een uitdaging zijn, zeker als we naar meerdimensionale ruimtes kijken. Eerdere onderzoeken toonden aan dat implementaties zoals inverse reinforcement learning (IRL), waarbij de agent zelf op zoek gaat naar een geschikte reward functie in plaats van een zelf

gemaakte reward functie te gebruiken (Arora & Doshi, 2021; Boularias e.a., 2011), veelbelovend zijn.

Het vertalen van de staat waarin de agent zich bevindt naar een discrete omgeving wordt essentieel in dergelijke complexe scenario's. Als de agent te maken heeft met een beperkt aantal omgevingen, kan die worden getraind totdat hij in theorie in staat is elk parkoers zonder fouten af te leggen. De integratie van RL en genetische algoritmen biedt hierbij de mogelijkheid om complexe besluitvormingsprocessen te optimaliseren en aan te passen aan variabele omgevingsfactoren.

In dit onderzoek is Minecraft gekozen als game om gesimuleerde bots te trainen in het beheersen van een parkoers. Minecraft is een driedimensionaal spel, maar we beperken de training van de agents tot een tweedimensionaal parkoers. Het parkoers bestaat uit een startblok (startpositie) en een opeenvolging

van blokken. Tussen de blokken kunnen 0 tot 3 luchtblokken voorkomen en ook hoogteverschillen tussen de blokken zijn mogelijk. Het is aan de agent om een reeks van acties (springen en vooruit lopen) in het parkoers te voltooien zonder te vallen en in een zo kort mogelijke tijd.

Het hoofddoel van deze studie is gericht op de doeltreffende training van agents, met als specifieke focus het beheersen van een complex tweedimensionaal parkoers. Vervolgens worden verschillende strategieën geanalyseerd en geëvalueerd, die door deze agents tijdens de navigatie door het parkoers kunnen worden toegepast. Vaststellen welke strategieën het meest effectief zijn in diverse situaties is onze doelstelling. Dit onderzoek streeft niet alleen naar verbetering van de prestaties van de agents maar beoogt ook inzicht in de toepasbaarheid van de strategieën in verschillende omgevingscontexten.

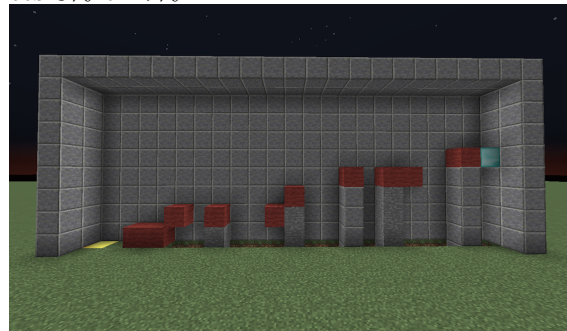
## 2 Methode

In deze sectie beschrijven we de gevolgde procedures en worden de verschillende geïmplementeerde keuzes toegelicht. Om onze agent als speler in een Minecraft wereld in te laden, maken we gebruik van „mineflayer”, g.d. Dit stelt ons in staat om één of meerdere ”bots” tegelijk in te laden in een lokale server. Voor de genetische implementatie gebruiken we 15 bots tegelijk, terwijl we ons bij de RL implementatie moeten beperken tot één enkele bot.

### 2.1 Genetisch algoritme

Het genetisch algoritme werkt zoals hierboven vermeld in generaties van 15 bots. Deze hebben vier simpele bewegingsmogelijkheden: voorwaarts sprinten, achterwaarts sprinten, al sprintend springen, en stoppen met bewegen. Elke 0.25 seconden wordt zo’n actie uitgevoerd. Met deze bewegingsopties en frequentie is het mogelijk om alle soor-

ten sprongen te halen. Zowel lange als korte sprongen kunnen gemaakt worden, aangezien sprinten kan afgewisseld worden met stoppen, zelfs tijdens de sprong. De agents starten met een willekeurige lijst van natuurlijke getallen van 0-3 die de verschillende acties voorstellen. De lijst is dus een reeks van acties die sequentieel worden uitgevoerd. De agents hebben een beperking op het aantal acties dat ze mogen uitvoeren. Bij ons parkoers van middelmatige lengte (Figuur 1) is dit maximaal 30 acties. Na elke uitgevoerde actie wordt de fitness berekend op basis van de huidige afstand tot het doel, en deze afstand wordt vervolgens toegevoegd aan een fitnesssteller. Het doel van het genetisch algoritme is om de fitnesssteller te minimaliseren. Als het doel is bereikt, stopt de bot met bewegen en blijft zijn fitnesssteller tot het einde van de generatie onveranderd. Als er minder acties zijn uitgevoerd, en de agent dus sneller zijn doel bereikt, resulteert dat in een lagere fitnesssteller op het einde. Tijdens selection wordt de hoogste helft van agents, volgens de waarde van hun fitnesssteller, verwijderd. Daarna worden er nieuwe agents aangemaakt via crossover; telkens worden twee agents met fitness  $fit_1$  en  $fit_2$  genomen, waarbij de acties willekeurig worden gecombineerd. De waarschijnlijkheid dat een actie wordt overgenomen van agent 2 is:  $\frac{fit_1}{fit_1 + fit_2}$ . Ten slotte in de mutatiestap heeft elke actie een kans om vervangen te worden door een nieuwe willekeurige actie. We proberen dit met een mutatie rate van 5% en daarna ook eens met rates 3% en 7%.



Figuur 1: Voorbeeld parkoers genetisch algoritme (lengte: 20 blokken)

Een andere eigenschap die we kunnen onderzoeken, is de prestatie bij langere parkoersen. We zullen kijken wat de gevolgen zijn bij een mutatie rate van 3% en 5% voor een parkoers van 44 blokken.

## 2.2 Reinforcement learning

Voor dit onderzoek maken we gebruik van het lineair Q-netwerkmodel, bekend als Q-learning (Watkins & Dayan, 1992). Dit model schat de waarde van verschillende acties in een bepaalde omgeving en selecteert zo de optimale actie (Tai & Liu, 2016). De agent heeft de mogelijkheid om te kiezen uit vijf verschillende acties: voorwaarts lopen, herpositioneren, een korte sprong maken, een middellange sprong maken en een lange sprong maken.

Een aantal parameters beïnvloeden het leerproces van de agent. De leersnelheid (LR) die de balans tussen snel leren en het vermijden van overfitting beschrijft, is vastgesteld op 0.001. De gammaparameter die de discount factor bepaalt, is ingesteld op 0.80, wat betekent dat de agent voornamelijk rekening houdt met onmiddellijke beloningen, maar ook aandacht besteedt aan toekomstige beloningen.

Om verkenning en exploitatie in evenwicht te houden, is een epsilon-greedy strategie geïmplementeerd. De epsilon waarde bepaalt de kans dat de agent een willekeurige actie kiest (verkenning) en dus niet de actie die volgens het huidige model de beste is (exploitatie). Deze waarde neemt bij elke stap af, met een initiële afname van 0.6% per stap, tot een minimum van 3% bereikt wordt. Er is dus minstens 3% kans dat de agent een willekeurige actie uitvoert.


Om de waarnemingen te onderzoeken maken we gebruik van een willekeurig gegenereerd parkoers bestaande uit 10 blokken. De agent

moet dus maximaal 10 verschillende acties correct uitvoeren om het doel te bereiken. Indien de agent erin slaagt om tweemaal succesvol het doel te bereiken, wordt er een nieuw parkoers gegenereerd. Op deze manier zorgen we dat de agent na verloop van tijd vertrouwd is met alle mogelijke acties en omgevingen.

### 2.2.1 OMGEVING

De omgeving is de context waarin de agent opereert. Het omvat alle factoren die invloed hebben op de agent, op basis waarvan hij zijn volgende actie kiest. In dit onderzoek is de omgeving een lijst van 16 elementen. De eerste 15 elementen stellen de locaties van blokken voor zoals op Figuur 2. Het laatste element geeft aan of de agent in de richting van de eindpositie kijkt.

	0	1	2	
3	4	5	6	
7	8	9	10	
11	12	13	14	



Figuur 2: De omgeving van de agent is in dit geval  $\{0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1\}$

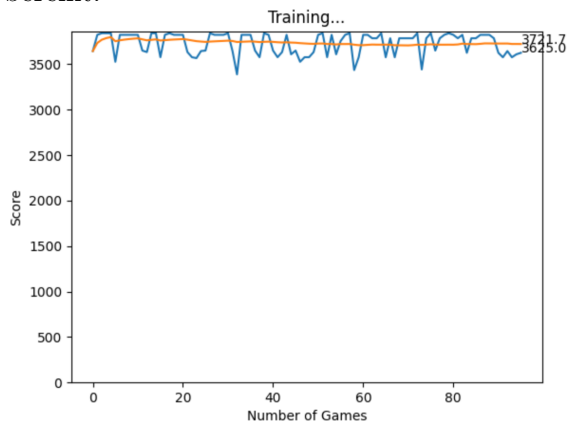
### 2.2.2 REWARD FUNCTIE

Om een reinforcement learning agent efficiënt te laten werken is een goede reward functie belangrijk. Een ongunstige reward functie zorgt ervoor dat de agent vast komt te zitten in een lokaal minimum. In deze sectie maken we enkel gebruik van een discrete implementatie voor beweging, wat betekent dat de agent na elke actie in het midden van het blok wordt geplaatst. We zullen itereren over enkele reward functies tot we een werkende implementatie bekommen. Een eerste reward-

functie zou de afstand tussen de positie van de agent en het doelpunt kunnen omvatten.

$$Reward = 100 \left( 1 - \frac{\text{distance\_to\_goal}}{\text{distance\_start\_to\_goal}} \right)$$

Hieruit volgt dat de agent een steeds hogere reward krijgt wanneer hij dichterbij het doel komt. Deze reward krijgt de agent bij elke stap. Verder kunnen we een grote reward definiëren als de agent het doel bereikt. Wanneer de agent van het parkoers valt of het parkoers niet kan voltooien binnen de  $n$  stappen, krijgt de agent een negatieve reward. Met deze reward functie krijgen we Figuur 3 als resultaat. De grafiek geeft  $-reward$  als score, wat betekent dat de agent alleen een negatieve score ontvangt wanneer het doel is bereikt.



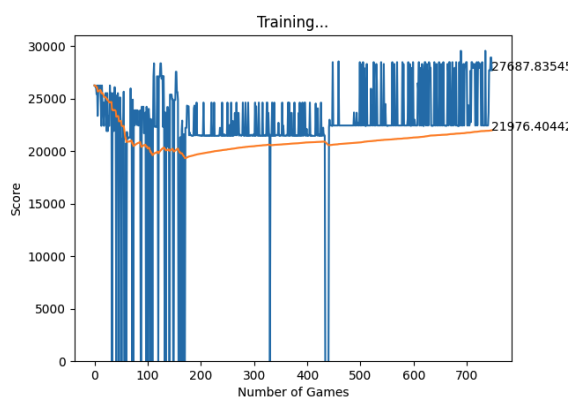
Figuur 3: Resultaat reward functie 1

De grafiek illustreert dat de agent niet bijleert. Na verloop van tijd vertoont de agent geen merkbare vooruitgang meer en lijkt hij stil te staan. Het is evident dat er zich problemen voordoen. In plaats van de reward te baseren op de globale positie van de agent kunnen we dit ook lokaal bekijken. We kunnen namelijk de reward baseren op de twee vorige posities.

$$Reward = d_0 - \frac{d_1 + d_2}{2}$$

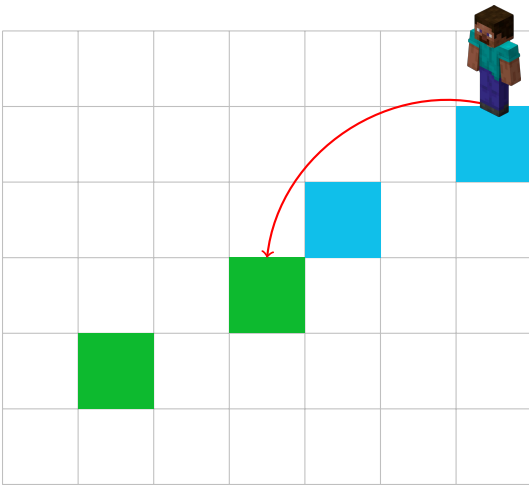
In deze formule is  $d_0$  de huidige afstand tussen de agent en het doel,  $d_1$  en  $d_2$  geven

de vorige twee afstanden tussen de agent en het doel weer. Met deze nieuwe reward functie zal de agent een grote positieve reward krijgen indien veel vooruitgang wordt gemaakt in 1 actie, terwijl negatieve beloningen worden toegekend wanneer de agent in de verkeerde richting beweegt. Bovendien kan een aanvullende controle worden toegevoegd om de agent te bestraffen bij stilstand. Als we deze nieuwe reward functie uitvoeren krijgen we als resultaat Figuur 4.



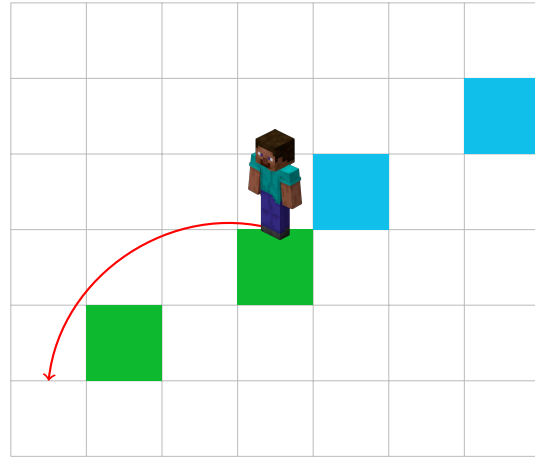
Figuur 4: Resultaat reward functie 2

Figuur 4 illustreert het leerproces van de agent. Na ongeveer 60 pogingen slaagt de agent er in een groot deel van de parkoersen succesvol af te leggen. Opmerkelijk is dat na ongeveer 180 pogingen de agent plotseling vastzit in een lokaal minimum. Wanneer we de agent observeren terwijl hij door het parkoers navigeert, merken we op dat een nieuw probleem zich voordoet.



Figuur 5: Agent kiest nieuwe actie op basis van blauwe blokken.

In Figuur 5 wordt waargenomen dat de agent een actie selecteert op basis van zijn omgeving, waarbij alleen de blauwe blokken zichtbaar zijn voor de agent. De afbeelding illustreert dat de agent een onjuiste actie kiest door over het blauwe blok te springen. We zien in dit geval geen probleem want er is toevallig een groene blok die hem opvangt waardoor de agent een reward krijgt. De situatie na de eerste sprong wordt voorgesteld in Figuur 5. Nadien zal de agent opnieuw, op basis van zijn omgeving, een volgende actie selecteren. De agent bevindt zich opnieuw in exact dezelfde omgeving als tijdens de vorige sprong. Aangezien de vorige actie een beloning heeft opgeleverd, zal de agent besluiten om dezelfde sprong te herhalen. In dit geval is er geen blok om de agent op te vangen, wat resulteert in een val.



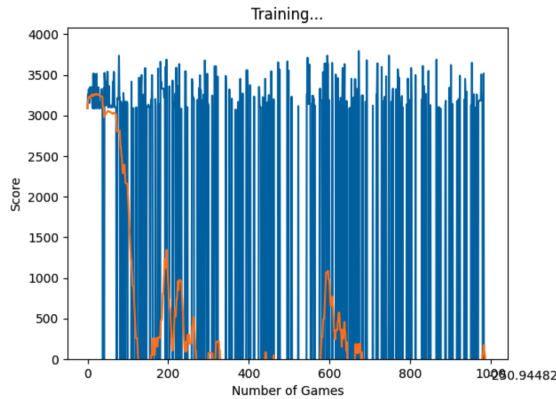
Figuur 6: Agent kiest nieuwe actie op basis van blauwe blokken.

Dit probleem kan op verschillende manieren worden aangepakt. Een initiële benadering is om de agent strenger te bestraffen wanneer die van het parkoers valt. Deze aanpak blijkt niet altijd effectief te zijn. Bijvoorbeeld, als de situatie zoals voorgesteld in Figuur 1 zich herhaaldelijk voordoet, zou dit leiden tot een aanzienlijke negatieve beloning, wat op zijn beurt andere problemen kan veroorzaken. Een alternatieve oplossing zou zijn om de omgeving te vergroten. Hoewel dit een mogelijke oplossing is, roept dit de vraag op: wanneer is de omgeving groot genoeg? Bovendien kan een grotere omgeving leiden tot een langzamer leerproces voor de agent. Een laatste overweging kan zijn om kortere sprongen te belonen met meer punten. We krijgen dus volgende reward functie:

$$Reward = w_i \left( d_0 - \frac{d_1 + d_2}{2} \right)$$

In deze aanpak wordt aan elke actie een gewicht  $w_i$  toegekend op basis van de afstand die door de actie wordt afgelegd. Het leerproces van de laatste oplossing wordt geïllustreerd in Figuur 7. De grafiek toont enkel de positieve waarden, omdat deze van primair belang zijn. Hierin is te zien dat

de oranje lijn convergeert naar een negatieve waarde, wat duidt op een leereffect.



Figuur 7: Resultaat reward functie 3

### 2.2.3 GAME RELATEERDE PROBLEMEN

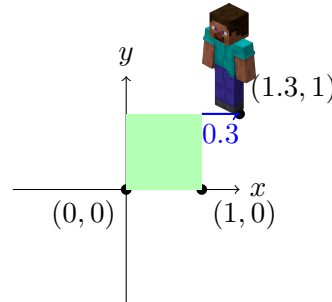
#### 1. vertraging

Minecraft is een multiplayer spel waarbij een client moet verbinden met een server. De server draait niet altijd lokaal en kan soms vertraging oplopen, wat problemen kan veroorzaken tijdens het trainen van een agent. Een agent kan een juiste actie kiezen, zoals een sprong van 2 blokken, waarbij het eerst vooruit moet lopen en dan enkele tientallen milliseconden later moet springen. Als er vertraging is, kan de agent te vroeg of te laat springen, waardoor de sprong niet succesvol wordt uitgevoerd, zelfs als de agent de juiste keuze heeft gemaakt. Om hier rekening mee te houden, mag de agent niet te veel negatieve beloningen ontvangen, aangezien dit het leervermogen kan beïnvloeden en het leren zeer langzaam maakt. Daarnaast kunnen we vertraging minimaliseren door in plaats van milliseconden, ticks te gebruiken. In normale omstandigheden duurt 1 tick 0.05 seconden en voert de server in elke tick berekeningen uit, zoals het bijwerken van de positie van de spelers. Als de server veel berekeningen moet uitvoeren en dus vertraging kan oplopen, zal één tick langer duren dan 0.05 seconden. De client kan op

basis van informatie die de server stuurt, berekenen hoeveel ticks per seconde worden verwerkt. De agent 10 milliseconden laten wachten voor het springen, kan vervangen worden door bijvoorbeeld 2 ticks te wachten.

#### 2. coördinaten

Het spelmodel heeft een breedte van 0.3 blokken. Een speler kan zich op de rand van een blok bevinden, zoals geïllustreerd in Figuur 8. De uitdaging hierbij is het vaststellen van de juiste positie waarop de omgeving moet worden weergegeven. De speler geeft de coördinaat  $(1,1)$  weer, al staat hij eigenlijk op blok  $(0,0)$ . Om de juiste positie te bepalen maken we gebruik van de richting waarin de speler kijkt en van blokken onder en naast de agent.



Figuur 8: Speler op rand van blok  $(0,0)$

### 2.2.4 CONTINUE WERELD

In een Minecraft wereld worden de coördinaten weergegeven als floating points en geen discrete getallen. In deze sectie zal de agent zelf bepalen wanneer het zich moet herpositioneren voordat het een sprong maakt. Om dit correct te laten werken moet de omgeving uitgebreid worden om meer informatie over de exacte positie te bevatten. Concreet geven we tien extra elementen mee in de state: als het deel na de komma van de z-coördinaat kleiner is dan 0.1 dan zal element 17 van de omgeving gelijk zijn aan 1 en alle volgende elementen zullen 0 zijn. Als dat niet het geval is, maar

het is wel kleiner dan 0.2, zal alleen element 18 gelijk zijn aan 1, en dit patroon herhaalt zich voor alle tien nieuwe elementen van de omgeving. Aangezien een speler tot net voorbij de rand van een blok kan staan en dus technisch gezien een andere z-coördinaat kan krijgen voor hetzelfde blok, zal de state ook een element bevatten dat aanduidt of het blok recht onder de agent vast of lucht is. Als het blok onder de agent lucht is en element 18 gelijk is aan 1, kan de agent waarnemen dat hij net voorbij de rand van het vorige blok staat. In dit geval kan de agent ervoor kiezen zich te herpositioneren in plaats van vooruit te lopen en te vallen. Na elke actie wordt gewacht tot de agent stilstaat op vaste grond voordat een nieuwe state wordt opgehaald en een nieuwe actie gekozen wordt. Die wachttijd is noodzakelijk omdat de agent geen informatie heeft over de snelheid, waardoor hij geen onderscheid kan maken tussen een omgeving waarin hij in beweging is en een waarin hij stilstaat. Daarom zorgen we ervoor dat hij stilstaat voorafgaand aan elke actie. Zelfs met deze beperking besparen we tijd tegenover de discrete versie waar de agent na elke actie automatisch naar het centrum van het huidige blok moet wandelen. In deze continue versie kan de agent landen op een blok en meteen verdergaan wanneer herpositioneren niet nodig is. Dat vraagt echter wel een stuk meer complexiteit aangezien elke omgeving nu meerdere states kan opleveren afhankelijk van de exacte positie op het blok.

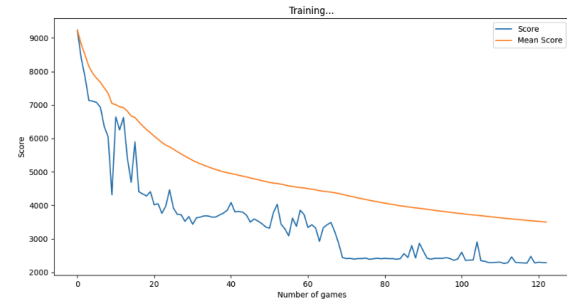
### 3 Resultaten

#### 3.1 Genetisch algoritme

Het resultaat van 120 generaties trainen op een zelfgemaakt parkoers van 20 blokken (Figuur 1), is weergegeven in Figuur 9, waarbij een mutatie rate van 5% is toegepast.

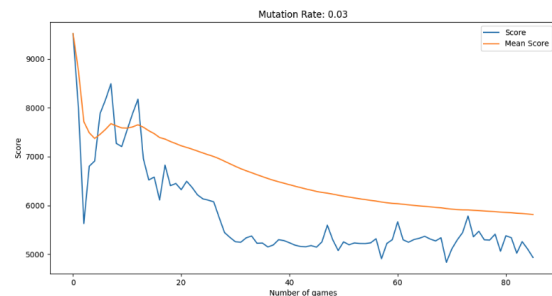
De score in deze grafiek stelt de gemiddelde fitness van de 15 bots voor op het einde van

elke generatie. Een lagere score is beter, en we zien dat de score convergeert rond 2400, waar de meeste bots het einde bereiken. Er zijn enkele lokale pieken en dalen op de grafiek, die kunnen veroorzaakt zijn door willekeurigheid in de crossover- en mutatie-stap. Maar ook game gerelateerde problemen zoals vertraging kunnen ervoor zorgen dat sommige goede acties falen of slechte acties slagen, wat ook de fitness zal veranderen op onvoorspelbare manieren.



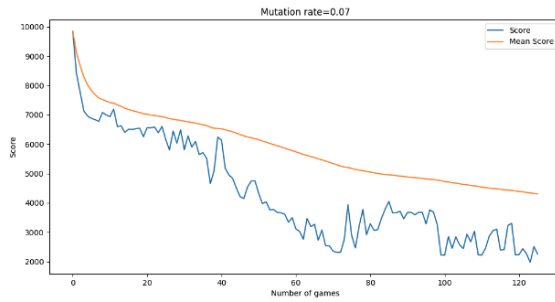
Figuur 9: Resultaat genetisch algoritme, mutatie rate van 5%

Het resultaat voor hetzelfde parkoers met een mutatie rate van 3% staat in Figuur 10. Hier convergeert het algoritme al vanaf 40 generaties naar een score van 5200, waarna het in nog 40 extra generaties geen vooruitgang meer maakt. Verder hebben we dit ook uitgevoerd bij een mutatie rate van 7% (Figuur 11), waarbij we een vergelijkbaar resultaat zien in vergelijking met de 5% mutatie rate, maar er is een grotere variantie tussen opeenvolgende generaties.



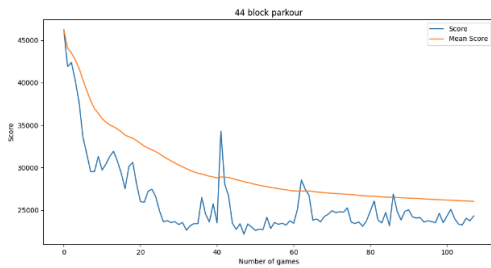
Figuur 10: Resultaat genetisch algoritme, mutatie rate van 3%





Figuur 11: Resultaat genetisch algoritme, mutatie rate van 7%

Bij het parkoers van 44 blokken slaagt de agent er niet in het doel te bereiken bij een mutatie rate van 5%. Het convergeert uiteindelijk naar een score van 24000, zoals je kunt zien in Figuur 12.



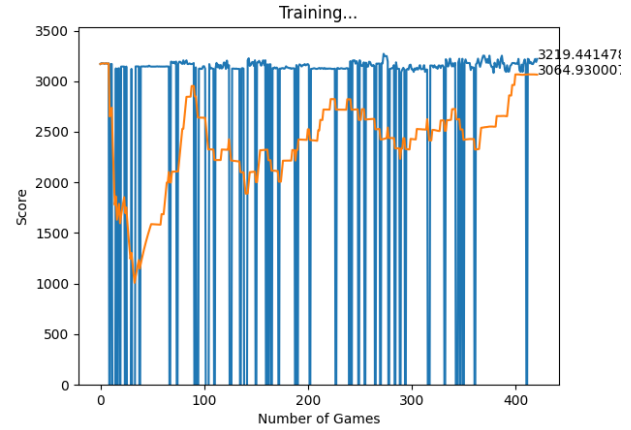
Figuur 12: Resultaat genetisch algoritme, mutatie rate van 3%

### 3.2 Reinforcement learning

Bij het observeren van de grafiek van de reward functie in Figuur 7 zien we dat er vele pieken optreden. Deze pieken kunnen om verschillende redenen ontstaan. Er bestaat steeds een kans van 3% dat de agent een willekeurige actie uitvoert. Daarnaast kan de agent ook falen als gevolg van vertragingen in het spel. De oranje lijn geeft aan dat de grafiek convergeert naar een negatieve waarde, wat betekent dat de agent vaker het doel bereikt dan dat hij faalt.

De continue implementatie blijkt niet te werken. Dit is duidelijk zichtbaar in Figuur 13. Hoewel de agent in staat is om de parkoersen te voltooien, vertoont deze geen aantoonbare

verbetering en lijkt zelfs achteruit te gaan, aangezien de gemiddelde score langzaam toeneemt.



Figuur 13: Resultaat continu RL algoritme

## 4 Discussie

De resultaten in sectie 3.1 geven aan dat bij het parkoers van 20 blokken een mutatie rate van 5% zorgt voor de snelste convergentie naar een optimale oplossing. Bij een lengte van 44 blokken is de rate van 5% waarschijnlijk te hoog, omdat de kans om onderweg een fout te maken bij een langer parkoers groter wordt.

In sectie 3.2 zien we dat voor RL enkel de discrete implementatie van onze agent werkt. In de continue implementatie wordt de complexiteit verhoogd, wat gepaard gaat met game-gerelateerde problemen zoals vertraging. Deze problemen leiden tot onvoorspelbaarheid in de continue implementatie. Vanwege de onvoorspelbare uitkomsten van acties lijken de rewardfuncties daar niet effectief te functioneren. Dit leidt tot het aanleren van foute acties en na enige tijd van eenzelfde parkoers te falen kan het ook leiden tot overtraining. De agent faalt niet alleen vaak in de continue omgeving, maar vertoont zelfs een afname in prestaties. In de discrete implementatie zijn er ook enkele fouten waar te nemen, maar de betere betrouwbaarheid van



sprongen zorgt ervoor dat de reward functies leiden tot een langzame verbetering in performantie. In meer dan de helft van de gevallen leert de agent zijn omgeving correct te interpreteren en is hij in staat om op deze manier ook nieuwe parkoersen op te lossen. Reinforcement learning gebruiken in games zoals Minecraft is niet eenvoudig om verschillende redenen die onder andere gegeven zijn in 2.2.3. Het trainen van het model ondervindt ook vertraging, mede doordat er geen directe mogelijkheid bestaat om het tempo van het spel te verhogen. Deze beperking is hoofdzakelijk te wijten aan het ontbreken van ingebouwde versnellings mogelijkheden in Minecraft, waardoor de agent gebonden is aan de natuurlijke snelheid van het spel. De beperking heeft invloed op het leertempo, wat de uitdagingen benadrukt bij het trainen van reinforcement learning-modellen. Hoewel de huidige implementatie convergeert naar een oplossing, zal het in sommige gevallen niet de meest optimale reeks van acties vinden. Er zijn diverse methoden die effectiever zijn dan de huidige aanpak. Jiang (2023) bevat een aantal van deze optimalisaties zoals double DQN waarbij men 2 neurale netwerken gebruikt voor het berekenen van Q waarden en dan onafhankelijk acties kiest.

## 5 Conclusie

Genetische algoritmen convergeren snel maar zijn alleen geschikt voor korte parkoersen, onder andere vanwege de mutatie rate. Het genetisch algoritme zal altijd convergeren naar de beste oplossing, maar is beperkt tot het functioneren voor één parkoers. Aan de andere kant zien we dat reinforcement learning trager convergeert, maar het in staat is elk parkoers op te lossen. RL wordt echter beperkt in efficiëntie door de beperkte hoeveelheid uitvoerbare sprongen en de vereiste om na elke actie te herpositioneren. Onze pogingen om een meer flexibele con-

tinue implementatie te realiseren, waren niet succesvol. Dit komt voort uit de noodzaak om het model voortdurend een discrete omgeving te geven en bij meer gedetailleerde representaties van de continue game-omgeving wordt dit al snel complex.

Voor verdere studie kunnen we overwegen om andere methoden te gebruiken voor het implementeren van de reward functie, of onderzoek te doen naar een efficiëntere representatie van de omgeving, waardoor de agent meer informatie kan verkrijgen. Een mogelijk experiment zou zijn om inverse reinforcement learning (IRL) te gebruiken om een betere reward functie te krijgen. Of in plaats van een beperkt aantal statische acties zou een experimentele toevoeging gebruikmaken van een continue actie ruimte (Rummery & Niranjana, 1994; „Safety-Aware Task Composition for Discrete and Continuous Reinforcement Learning”, 2023; van Hasselt & Wiering, 2007). De agent kan dan zelf bepalen hoe ver vooruit of achteruit te springen of stappen. Het resultaat zou meer flexibiliteit opleveren, waarbij de agent in staat is diverse sprongen uit te voeren.

Een aanvullend experiment met betrekking tot genetische algoritmen zou zich richten op de invloed van de mutatie rate in relatie tot de lengte van het parkoers. Het doel zou zijn om te bepalen wat de optimale mutatie rate is voor een specifieke lengte van het parkoers.

## 6 Disclaimers

Het onderzoek heeft gebruikt gemaakt van verschillende AI tools: ChatGPT, Jetbrains AI assistent, Github Copilot, Copilot chat en Bing voor zowel het schrijven van code als tekst voor het onderzoek. Vooral prompts zoals "Herschrijf deze zin in academische stijl" bleken zeer handig te zijn. AI code assistenten hielpen vooral bij implementatie vragen zoals "Geef een voorbeeld programma dat gebruik maakt van functie x".

## Referenties

- Arora, S., & Doshi, P. (2021). A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297, 103500. <https://doi.org/https://doi.org/10.1016/j.artint.2021.103500>
- Boularias, A., Kober, J., & Peters, J. (2011, november). Relative Entropy Inverse Reinforcement Learning. In G. Gordon, D. Dunson & M. Dudík (Red.), *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 182–189, Deel 15). PMLR. <https://proceedings.mlr.press/v15/boularias11a.html>
- Jiang, M. (2023). Learning Curricula in Open-Ended Worlds. *mineflayer*. (g.d.) (4.14.0). <https://github.com/PrismarineJS/mineflayer>
- Rummery, G. A., & Niranjan, M. (1994, september). *On-line Q-learning using connectionist systems* (CUED/F-INFENG/TR Nr. 166). Cambridge University Engineering Department. [ftp://svr-ftp.eng.cam.ac.uk/reports/rummery\\_tr166.ps.Z](ftp://svr-ftp.eng.cam.ac.uk/reports/rummery_tr166.ps.Z)
- Safety-Aware Task Composition for Discrete and Continuous Reinforcement Learning*. (2023). <https://doi.org/10.48550/arXiv.2306.17033>
- Tai, L., & Liu, M. (2016). A robot exploration strategy based on Q-learning network. *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, 57–62. <https://doi.org/10.1109/RCAR.2016.7784001>
- van Hasselt, H., & Wiering, M. A. (2007). Reinforcement Learning in Continuous Action Spaces. *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 272–279. <https://doi.org/10.1109/ADPRL.2007.368199>
- Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3), 279–292. <https://doi.org/10.1007/BF00992698>