

**Teknikprogrammet**  
**Sven Eriksonsgymnasiet**



# **Reinforced Learning i virtuell miljö**

Reinforced Learning används i en virtuell miljö för att träna beteendet av en maskin

**Erik Holmström TE21IM**  
**Kacper Ertmański TE21IM**  
**Borås år 2023/2024**  
**Handledare: Christian Aronsson & Isak Altsäter**

## **Abstract**

The purpose of this project was to study reinforcement learning and its potential applications in real-world scenarios. The central focus revolved around addressing the problem statement: “How can reinforced machine learning be utilized to train a car to navigate within a virtual environment, using negative and positive reinforcement?”. To achieve this objective, various literary works on reinforced machine learning were reviewed, which provided valuable insights into the methodology adopted in this project. Throughout the project, five virtual environments were created and configured within Unity to facilitate the training of the agent in different scenarios aimed at developing specific behaviors. Subsequently, the behaviors exhibited by the agent were analyzed based on the collected data. The findings of this study suggest that reinforcement learning holds promise for real-world applications, albeit with potential challenges.

<b>1. Inledning</b>	<b>4</b>
<b>1.1 Bakgrund</b>	<b>4</b>
1.1.1 Begrepp	5
1.2 Syfte och frågeställningar	6
1.3 Avgränsningar	6
1.4 Metodbeskrivning	7
<b>2. Teori och tidigare forskning</b>	<b>9</b>
2.1 Teorin bakom och djupare förståelse inom RL	9
2.2 Tidigare forskning	11
<b>3. Arbetsgång och processredovisning</b>	<b>12</b>
3.1 Installation och konfiguration av programvara	12
3.1.1 Virtuellt Maskin	12
3.1.2 Mjukvara	13
3.1.3 Stegvis konfiguration	13
3.1.4 Felkällor	15
3.2 Grundläggande kodning av agenter	16
3.2.1 Script	16
3.2.2 Metoder	18
3.2.3 Parametrar	19
3.3 Utförande	20
3.3.1 Agenten	20
3.3.2 Träning	21
3.3.3 Miljöer	24
3.3.3.1 Träningsmiljö 1	24
3.3.3.2 Träningsmiljö 2	25
3.3.3.3 Träningsmiljö 3	26
3.3.3.4 Träningsmiljö 4	27
3.3.3.5 Tänka miljöer	28
<b>4. Resultat, utvärdering och diskussion</b>	<b>29</b>
4.1 Resultat av agentens träning	29
4.2 Svar på frågeställning	35
4.2.1 Hur kan reinforced machine learning träna en bil till att navigera i en virtuell miljö med hjälp av positiv och negativ reinforcement?	35
4.2.2 Vilka olika sorters hinder ger mest utmaning för bilen?, Varför?	36
4.2.3 Hur borde arbetet genomföras för att undvika dålig träningsdata för agenten, som skulle kunna vilseleda eller felaktigt utbilda den?	37
4.3 Diskussion	38
<b>5. Källförteckning</b>	<b>39</b>
<b>6. Bilagor</b>	<b>40</b>
6.1 C# script agenten baserar sitt beteende och inlärning utefter	40
6.2 .yaml parametrar för agent	45

# 1. Inledning

## 1.1 Bakgrund

Genom generationer av innovationer har människor sakta men stadigt utvecklats i flertal unika forskningsområden, från att spåra den förflutna historien bland döda lik genom kol-14 metoden, skapa virtuella världar nästintill identiska till vår vardagliga verklighet, tills dagens moderna maskiner med möjlighet av att kopiera eller förbättra mänskliga beteenden. Den utmärkta tekniken som utbildar robotar till att uppnå dessa mänskliga egenskaper kallas "reinforced machine learning".

Enligt en artikel från Bowyer M. Caleb (Bowyer 2022) har metoden ingen grund i datavetenskap utan väldigt långt ifrån, till och med i ett helt annat forskningsområde, vilket var inriktat i psykologi bland djur och dess beteende. Under 1930-talet har forskaren B.F Skinner utfört experiment på både råttor men också duvor för att utforska deras beteenden, djuren fick utföra progressivt mer komplexa utmaningar. Ifall djuren utförde de angivna instruktionerna korrekt fick de en belöning, genom denna process utvecklade båda djurarterna en aktiv reflektion utöver de förflutna upplevelser.

Samma princip har senare tillämpats inom området datateknik, genom att forskare eller programmerare skapar övningar som datorerna löser. Efter att en iteration utförts, belönas datorn för korrekta lösningar liknande till djurexperimenten. Däremot när datorn inte hanterar problemet med en korrekt metod utan att den inte tillämpar ny information får den ingen belöning. Det resulterar i att programmet utnyttjar gammal statistik i kombination av den nyligen insamlade datan för att nå lämpligare beteenden. Processen är en produkt av programmering och studier av djurbeteende, processen kallas reinforced machine learning.

Området för detta arbete är just 'reinforced machine learning', ofta kallat 'reinforced learning' och kommer bli förkortat till RL i resten av arbetet. Träningsmetoden används för utbildning av maskiner, vilket kan leda maskinen till att kunna utföra uppgifter självständigt men även utveckla sig själv. Det kan även med tillräcklig mängd av träningstillfällen användas för att uppfylla komplicerade mål med maskiner och därmed optimera, förenkla eller förbättra operationer inom flera olika områden.

Användning av RL har enligt Bowyer (2022) en huvudsaklig uppgift i forskning på artificiell intelligens, men även tillämpningar i andra avgränsade områden bland olika företags metoder för automatisering, utveckling av datorspel men även robotik.

Området som RL kommer tillämpas inom detta arbete är att utbilda en agent till att anpassa sig för en miljö inuti en virtuell miljö och dess utmaningar. Vilket kan sammanfattas som en form av automatisering. För att uppnå detta ska RL användas för att skapa grundläggande kunskaper hos maskinen för att vidareutvecklas till stadier i vilka maskinen självständigt anpassar sig och navigerar runt i världen med minimala felmarginal.

Anledningen till att forskning på RL utförs under detta arbete är på grund av dess omätbara potential som kan implementeras i vardaglig användning på allmän, kommersiell och privat nivå. Dess användning varierar på dessa nivåer från exempelvis långa arbetsuppgifter vilka kan enkelt hanteras genom automatiserade processer, vardagligt tillgängliga virtuella assistenter med optimerade konversationer och svar till frågor men också avancerad robotik.

Arbetet har även valts på grund av intresset över att kunna skapa en rimligt enkel och begriplig överblick för området RL, vilket privatpersoner kan utnyttja till sin egen fördel och implementera i sina intressen. Dessutom är arbetet intresseväckande på grund av den nya förståelsen över hur RL fungerar i våra vardagliga liv.

### 1.1.1 Begrepp

RL är ett mycket tekniskt område och använder därför många begrepp vilket förtydligar viktiga saker. Viktigast för detta arbete bland dessa begrepp är:

- **Agent** - Tinget som med hjälp av att följa en *policy* försöker maximera sin *return* vilket den får från *tillstånd* i sin *miljö*.
- **Policy** - Väljer vilken *handling* som ska tas utifrån *agentens tillstånd*.
- **Handling** - Metod för att övergå från ett *tillstånd* till ett annat med vägledning av en *policy*.
- **Return** - Ett värde som beskriver hur bra en viss *handling* var vid specifika *tillstånd*.
- **Tillstånd** - Värdet som beskriver konfigurationen av den *miljö agenten* befinner sig i.
- **Miljö** - Den virtuella miljön som innehåller en *agent*, olika *tillstånd* och andra ting.
- **Unity** - Är ett program med en spelmotor som i detta arbetet fyller syftet att underhålla *miljö* och *agenten* men även visualisera arbetet.
- **GitHub repository ML-agents** - En öppen källa för kod som tillåter *agenten* att utbildas i en *miljö*.
- **Target** - Ett objekt som förekommer inom *miljön* som fungerar som ett mål en agent ska uppnå.
- **Steps** - Lite vagt definierade inom RL, men kan beskrivas som antingen en fixerad uppdatering eller ett valintervall för en *agent*.

Kunskap kring vissa program och vad de bidrar till arbetet ger även en viktig grund till att förstå arbetet. Program som kommer bidra till en stor del av arbetet är bland annat unity, vilket är det program som lägger grunden för vår implementering av RL. Vanligtvis är Unity ett program som används för att enklare kunna implementera kod i grafiska moment inuti spel, men med hjälp av GitHub repository ML-agents kan det även användas för RL.

## 1.2 Syfte och frågeställningar

Syftet med arbetet är att utöka kunskapen inom RL om hur agenter tenderar att agera och hur de löser problem, för att senare kunna ge den lärda agenten en optimal kunskap om hur en riktig bil fungerar i verklig omgivning. För att uppnå detta syftet kommer arbetet att jobba utefter ett par frågeställningar.

Primära frågeställningarna som ska besvaras i detta arbete lyder som följande:

- “Hur kan Reinforced machine learning träna en bil till att navigera i en virtuell miljö med hjälp av positiv och negativ reinforcement?”

Sekundära frågeställningar för “Hur kan reinforced machine learning träna en bil...” lyder enligt följande:

- “Vilka olika sorters hinder ger mest utmaning för bilen?”, "Varför?"
- “Hur borde arbetet genomföras för att undvika dålig träningsdata för agenten, som skulle kunna vilseleda eller felaktigt utbilda den?”

Genom att svara på dessa frågor förväntas arbetet att ge insyn till RL och hur det kan tillämpas på ett mer effektivt sätt.

## 1.3 Avgränsningar

För att arbetet ska funka utifrån förutsättningen att alla resurser är tillgängliga kommer en del avgränsningar att läggas.

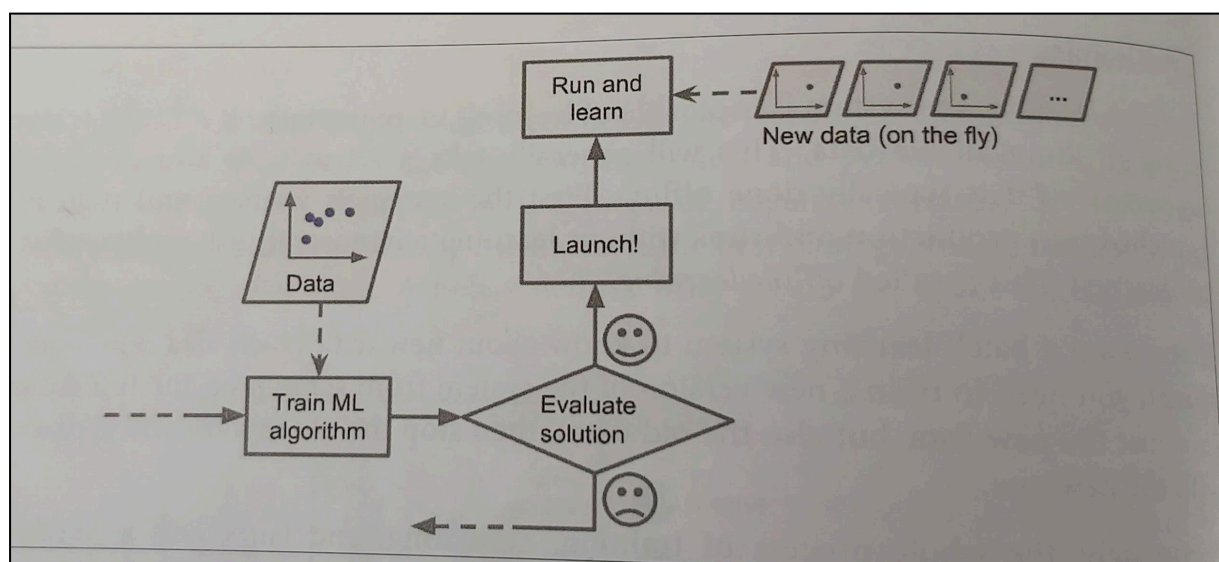
Den första avgränsningen blir att inte praktiskt testa den resulterande agenten i den verkliga världen. Med detta menas att utrusta en bil med sensorer och styrmoment för att bli kapabel att använda agenten till att navigera. Anledningen till denna avgränsning är att resurserna som krävs för att framställa detta kommer göra arbetet besynnerligen större och mer tidskrävande. Arbetet skulle även börja sträva mot andra ämnen än det som är i fokus för detta arbete.

Den andra avgränsningen är de andra två metoderna som finns under Machine learning, nämligen supervised- och unsupervised learning. Varför detta inte kommer användas är för att det inte finns tillräckligt med tid för att noggrant utföra tester med tolkbara resultat.

## 1.4 Metodbeskrivning

Resultatet som ska uppnås med detta arbete är att en agent som i eget designade virtuella miljöer kan optimera sin navigering för att nå ett mål på ett så bra sätt som möjligt.

För att nå detta resultat följer arbetet en process som beskrivs i ett kapitel om RL ur en bok (Géron 2019, s.16-17) skriven av Aurélien Géron. Metoden Géron beskrev, klassificerade han som online learning och följer ett arbetsflöde enligt följande schema vilket Géron valde för att representera processen.



(Géron 2019, s.16)

Så som processen tolkats kommer de olika stegen bli beskrivna och användas enligt följande.

1. “Data” innebär den information som agenten har tillgång till. Med detta menas agentens olika handlingar, den policy som agenten utvecklat, miljön den befinner sig i och liknande. Data blir både det som är givet till agenten i form av programmering, men även den data agenten själv har samlat in i föregående iterationer.
2. I steget “Train ML algorithm” kommer agenten att tränas i den miljö som var tillgiven.

3. "Evaluate solution" innebär att agentens prestanda döms och evalueras utifrån önskade egenskaper. Här delas processen i två delar. Blir det ett dåligt resultat repeteras processen från steg 1. "Data", där antingen olika handlingar omprogrammeras eller miljön ändras för att ge mer önskvärda resultat. Blir det åter ett dåligt resultat genomförs samma steg en gång till. När ett bra resultat eventuellt uppfyllts kommer processen fortsätta till steg 4.
4. "Launch! & Run and learn" Här får agenten träna i fred ett tag då den efter varje ny iteration tar åt sig ny data och förbättrar sig. Med förbättring menas det att agenten uppför sig mer säkert, effektivt och felfritt utefter vad som önskades.
5. När slutgiltiga resultaten är önskvärda kan processen börja om vid steg 1 "Data", med ny data för en ny situation. Exempel på sådan data är den miljö som agenten utsätts för eller olika handlingar den kan utföra. Det som önskas då en ny iteration påbörjas är att förbättra de svagheter som blev mest uppenbara under föregående iteration.

För att använda detta arbetsflödet behöver en miljö designas utifrån vad agenten ska lära sig under träningstillfället, men även en del handlingar som agenten kan utföra. Agenten tränar på att använda dessa handlingar under en lämplig tid för att den ska bli skicklig på det. Sedan provas den inom olika områden för att senare bli tränad på nytt utifrån svagheter som visade sig under provning. Denna process repeteras tills att agenten klarar av att navigera självständigt i miljön utan att åka av banan och uppnå positiv poängsättning. Kraven som kommer att ställas på agenten är enligt följande. Agenten ska kunna hålla sig till vägen och på egen hand och så effektivt som möjligt navigera till sitt mål.

Enligt ovanstående kommer arbetet att följa Gérons metod för att utveckla agenten. Dock för att testa agenten kommer miljöer att designas enligt de krav, avgränsningar och önskemål vi lagt på agentens beteende. För varje produkt som någon har skapat där hen vill att produkten ska förbättra sig krävs det en bra testbänk. Det mest relevanta att testa innan en produkt släpps är just det som den kommer att utsättas för, vilket även här i en virtuell miljö är relevant. Med tanke på detta kommer flera olika testbänkar att designas i form av specificerade virtuella miljöer för att prova olika områden och förmågor hos agenten som inte utvecklats i föregående miljö innan den kan ta sig till nästa steg av träning.

Olika testmoment som arbetet utvecklat är fartanpassning under svängning, komma till fullt stopp samt generell manövrering. När det gäller att kontrollera ett fordon's miljövänlighet och tålighet är det irrelevant för detta arbete då testet utförs virtuellt. Under arbetet kommer programmet Unity att användas för att underhålla och skapa den digitala miljön med de förbestämda hinder, bilen och terrängen.



## 2. Teori och tidigare forskning

### 2.1 Teorin bakom och djupare förståelse inom RL

Teorin bakom RL är ett djupt ämne beroende på vad ett arbete inom det väljer att undersöka. Grundläggande teorin bakom RL är dock som tidigare sagt så pass att en agent genom försök och misstag ska lösa problem utifrån givna möjliga handlingar den kan genomföra utan att behöva specificera exakt hur det problemet ska lösas.

När det gäller att ge en bild av den bakomliggande logiken avanceras ämnet en bit. Därför förklaras nu de mest grundläggande funktionerna inom RL och hur de fungerar.

Hur är det agenten tolkar den respons den får angående sina val under provning?

För att förklara detta används ett exempel från en artikel (Kaelbling, Littman & Moore 1996, s.239) som gett sig på att ge en så täckande förklaring av olika områden inom RL som möjligt.

<b>Environment:</b>	You are in state 65. You have 4 possible actions.
<b>Agent:</b>	I'll take action 2.
<b>Environment:</b>	You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.
<b>Agent:</b>	I'll take action 1.
<b>Environment:</b>	You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.
<b>Agent:</b>	I'll take action 2.
<b>Environment:</b>	You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions.
;	;

*(Kaelbling, Littman & Moore 1996, s.239)*

Det bilden ovanför visar är ett exempel av hur information är tillgiven till en agent som utifrån den informationen väljer en handling den kan ta. Efter detta val får agenten antingen en uppmuntran i form av ett positivt tal eller ett straff i form av ett negativt tal. Ett högre respektive lägre tal ger större påverkan då det är mer rätt eller fel jämfört med tal närmare värdet noll. Vad det är för värde som sätts på specifika händelser är i grunden utsatt av programmeraren. En sådan händelse är exempelvis att krocka in i en vägg, vilket ger ett negativt tal. Dock om agenten gör en handling som resulterar i flera olika straff och belöningar samtidigt kommer det bli ett helt nytt tal eftersom det kombinerar allt som den fick genom handlingen som genomfördes.

Informationen som agenten baserar sina val på kommer från miljön den befinner sig i. Exempel på sådan information är agentens tillstånd och alla dess möjliga handlingar. Till en början kommer agentens val att vara inget bättre än slumpmässigt. Men efter flera iterationer

börjar agenten dra kopplingar mellan vad den tidigare gjort vid ett visst tillstånd och hur det gick den gången.

Om bilden återigen används som ett exempel så ligger agenten två olika gånger i tillstånd 65. Första gången väljer den handling 2 och får en uppmuntran på 7 poäng. Nästa gång väljer agenten handling 2 men får istället en uppmuntran på 5 poäng. Anledningen till varför samma värde inte alltid tilldelas vid samma tillstånd och handling är för att en optimalt designad miljö är dynamisk. Med dynamisk syftas det på att det är små skillnader mellan varje träningstillfälle för att en och samma lösning inte alltid ska ge samma resultat. För att ge en förklaring grundad i verkligheten så är det inte alltid bra att lösa samma problem med samma metod. Åker en bilförare (agent) genom en korsning (miljö) utan att först leta efter annan trafik (handling) och klarar sig (belöning), betyder det inte att föraren varje gång framöver kan sluta söka efter trafik innan den åker genom korsningen. Anledningen till detta är eftersom andra bilars rutter och uppföranden inte är förutbestämda, med andra ord dynamiska. Däremot finns statiska hinder exempelvis stenbumlingar, väggar och byggnader som alltid är placerade på förutbestämda positioner. Detta kommer att vara information som agenten alltid har tillgång till, för att inte behöva överbelasta agenten och underlätta utbildningsprocessen.

Förutom hur belöningar och straff fungerar är det även bra att veta hur den policy som används fungerar och utvecklas. Logiken bakom en policy är grundad i möjligheter utifrån olika variabler som ändras beroende på en agents föregående handlingar. Från början kommer agenten följa en policy som tidigare sagts agerar slumpmässigt eftersom den ännu inte är erfaren och är utan den data som bestämmer variablerna. Målet för en agent är att optimera sin policy utifrån följande förhållande för att få så hög sammansatt belöning som möjligt.

$$\pi(a|s, \theta) = \Pr\{A_t = a \mid S_t = s, \theta_t = \theta\}$$

Detta förhållande beskriver är mellan en policy betecknat genom  $\pi$  och hur den är beroende av en handling betecknat  $a$ , tillstånd betecknat  $s$  och tid betecknat  $t$ . Under ett träningstillfälle kommer denna policy uppdateras efter varje val som agenten gör inuti miljön. Flera iterationer framåt kommer agenten kunna tolka vilka val som ger störst belöning utifrån vad policyn tidigare betecknat för att genomföra en mer informerad och lönsam lösning till problemet.

Det svåra med att uppnå en sådan policy är att balansera hur mycket belöning som agenten ska få, problemet i sig är att policyn försöker optimera en vinst, inte helt undvika en negativ belöning. Därför kan en agent lära sig att gå genom en negativ belöning för att nå en högre positiv belöning. Om situationen skulle beskrivas i ett verkligt scenario skulle det kunna likna följande.

Målet för en bil är att leverera gods på så snabb tid som den kan. Mellan platsen för avlastning och bilen finns det en stoppskylt och ett lite längre fram ett röd ljus. Eftersom bilen

även får belöning för att vara snabb kan den dra slutsatsen att straffen den får från att köra mot rödljus och stoppskylten är värt den vinst den gör på tid.

Problemet med denna lösning är just att köra mot ett rödljus och genom en stoppskylt är mot lagen. För att få agenten att inse detta behöver det negativa värden som den får från olagliga åtgärder vara så stor att den inte tänker på det som en möjlighet att köra mot ett rödljus.

Men då kommer ett nytt sammanhang då bilen måste välja mellan 2 dåliga alternativ. Här måste programmeraren vara den som bestämmer vad som anses mer eller mindre moraliskt. För en människa som måste välja ett alternativ skulle valet mellan att köra på en person och köra av vägen vara ett självklart val. Skulle en agent sättas inför samma situation skulle den bara se två olika värden och tolka sin lösning utifrån det. Därför är det programmerarens skyldighet att balansera hur stora olika negativa värden ska vara för att få fram det önskade beteendet ur agenten.

## 2.2 Tidigare forskning

Tidigare forskning och tillämpning inom området är brett eftersom RL kan användas till otroligt många olika syften.

Ett syfte som det har utnyttjats i en bred utsträckning är tv-spel, som Lillicrap Timothy (Aug 2017) skriver om hur de har tillämpat RL i sammanhanget av att skapa självstyrda motståndare i StarCraft II. I projektet involverades flertal agenter samtidigt på ett gemensamt plan där de fick samarbeta, medan riktiga spelare kunde påverka handlingen genom att göra egna val. När agenterna skulle lära sig under spelets gång utsattes de för flera problem vilka gjorde inlärningen svårare. Från att informationen som angavs till agenterna var begränsad på grund av att miljön syntes delvis då det utgick från placeringen på kartan och aktuell plats som agenten befann sig på, den stora storleken på kartan som skulle tolkas och även selektion och kontroll över trupper. Det krävde dessutom väldigt mycket tid för att utbilda agenter av den anledning att konsekvenserna av handlingar inte syntes direkt men skedde efter en längre tidsperiod. Utöver komplikationerna lyckades Lillicrap att uppnå agenter med en spelstil liknande till medelmåttiga nybörjare.

Ytterligare ett syfte är att försöka förutse en marknad, som Carapuço João, Neves Rui och Horta Nuno (Dec 2018) skriver i en artikel om hur RL implementeras för att skapa agenter vilka blir utsatta för gammal data på valutorna Euro och Dollar mellan åren 2010 och 2017 i en virtuell miljö med belönings signaler. Miljön som agenterna blev utbildade inom hade flera svårigheter, bland annat hade datan som förekom i miljön inget förutbestämt mönster. Detta försvårade förutsägbarheten och framtida handlingar som agenten skulle utföra. Förutom problemen som förekom lyckades agenterna att uppnå en vinstmarginal på  $114.0 \pm 19.6\%$  totalt och  $16.3 \pm 2.8\%$  årlig vinst.

## 3. Arbetsgång och processredovisning

### 3.1 Installation och konfiguration av programvara

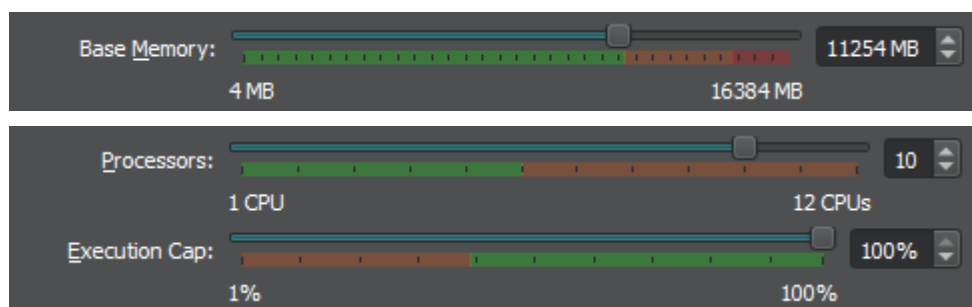
Detta kapitel går genom hur arbetet sätts upp och visar det fel som stöttes på under uppsättning samt hur det undveks och åtgärdades. Rubriker **3.1.1** - **3.1.4** visar denna process steg för steg.

Processen har tagits fram genom tre olika misslyckade försök att konfigurera arbetet och har nu resulterat i en fjärde fungerande metod som arbetet använde sig av för att inte stöta på fel som åtgärdas under **3.1.4 Felkällor**. Felkällorna är det som orsakade de tre första misslyckade försöken och hänvisas till genom **3.1.1** - **3.1.3** vid ställen som ofta resulterade i problem. Instruktionerna är skrivna i form av en guide för att enklare följa med processen, men allt skrivet nedan har genomförts under arbetet.

#### 3.1.1 Virtuellt Maskin

Processen nedan gjordes inuti en virtuell dator med hjälp av programmet **Oracle VM Virtualbox**. Detta är för att ha en ren arbetsbänk som minskar så många felkällor som möjligt. Om du får ett felmeddelande med att aktivera din VM bör du kontrollera att **Intel Virtualisation Technology ( IVT )**, eller motsvarande inställning för din hårdvara är på. Denna inställning ändras i dina **BIOS/UEFI** inställningar.

Med nedan givna konfiguration fungerade träning utan problem. Det mest krävande komponenter för denna sorts träning är **CPU** och **RAM** vilket är varför relativt mycket av den fysiska datorns kapacitet är tilldelad till den virtuella datorn.



*Inställningar för den virtuella datorn.*

Det **operativsystem** som användes under arbetet i den virtuella datorn är **Windows 10 Pro**. Arbetet provade inte olika system och kan därför inte garantera att följande instruktioner fungerar på andra operativsystem.

När det kommer till lagring behövs det för grundläggande applikationer ett ungefär av 60 GB tillgängligt på datorn. Dock stiger denna siffra snabbt, lagring i närheten av **80-100 GB** är därför rekommenderat för en bekväm träningsupplevelse.

### 3.1.2 Mjukvara

Inuti den virtuella datorn installerades följande program.

- Unity
- Python
- Anaconda
- Git
- Visual Studio Code

För Unity vill du använda dig av Unity Hub vilket enklare styr vilka versioner dina projekt använder sig av. Version **2022.3.18f1** användes för Unity projekten under arbetet.

För Python användes version **3.12.1**. En annan version kommer att specificeras i instruktionen under **3.1.3 Stegvis konfiguration**, dock eftersom den versionen inte kan laddas ner genom Pythons hemsida används 3.12.1 för tillfället.

Anaconda, specifikt **Anaconda Prompt**, är den programvara som kontrollerar den virtuella miljö som hanterar alla versioner av paket som **ML-Agents** använder sig av för att fungera.

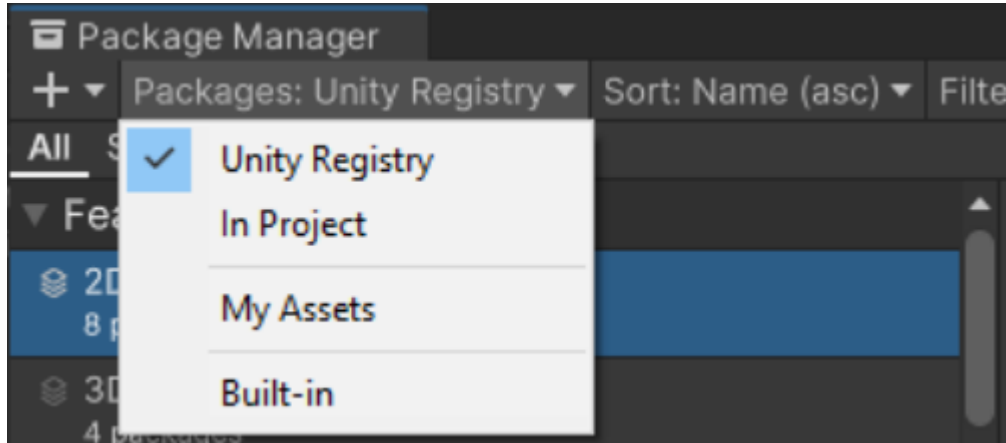
Git används för att kunna **kloa** det **GitHub repository** som håller på ML-Agents, vilket behövs för att träna en agent.

Visual Studio Code används för den **kodning** som behövs. Detta kan efter behov bytas ut till en annan mjukvara designad för kodning. Mjukvaran är dock **inte ett krav** till skillnad från de andra mjukvarorna ovan.

### 3.1.3 Stegvis konfiguration

Följande förklaring gäller efter att mjukvara i **3.1.2 Mjukvara** installerats. Om ett steg inte fungerar korrekt, hänvisar du till **3.1.4 Felkällor**.

1. Stäng av **brandväggen**
2. Börja ett Unity projekt med version **2022.3.18f1**
3. Tryck **Window → Package Manager → Unity Registry**



4. Sök efter ett paket vid namn ML-Agents och ladda ner paketet till ditt projekt.
5. Öppna **Anaconda Prompt** och navigera till projektet med syntaxen **cd [relativ filsökväg]**  
I detta fall gäller följande då namnet på Unity projektet är **GyarbRL**.

```
(base) C:\Users\Erik>cd GyarbRL
```

6. Klona in GitHub repository med följande syntax **git clone https://github.com/unity-Technologies/ml-agents**. Tänk på att hyperlänken kan ändras i framtiden. Hämta därför syntaxen direkt från deras repository.

```
(base) C:\Users\Erik\GyarbRL>git clone https://github.com/Unity-Technologies/ml-agents
Cloning into 'ml-agents'...
```

7. Skapa en virtuell miljö med syntaxen **conda create -n [namn på miljö] python=3.10.12**

Det är viktigt att versionen på python är 3.10.12 eftersom den specificeras i ML-Agents dokumentation men inte kan installeras från pythons hemsida.

```
(base) C:\Users\Erik\GyarbRL>conda create -n RL python=3.10.12
```

8. Aktivera den virtuella miljön genom syntaxen **conda activate [namn på miljö]**

```
(base) C:\Users\Erik\GyarbRL>conda activate RL
```

9. Uppgradera **pip** med syntaxen **python.exe -m pip install --upgrade pip**

10. Uppgradera **setuptools** med syntaxen **pip3 install --upgrade setuptools**

11. Installera **numpy 1.21.2** med syntaxen **conda install numpy=1.21.2**

Återigen är versionen viktig eftersom den specificeras i ML-Agents dokumentation.

12. Installera **mlagents** med syntaxen **conda install mlagents**

För att försäkra att miljön är konfigurerad kan du skriva syntaxen **mlagents-learn --help**. Om en lista med olika konfigurationer på syntaxen **mlagents-learn** framkommer är miljön redo för att påbörja träning. Det som återstår är att skapa en agent som ska tränas.

Om listan med konfigurationer inte framkommer. Radera miljön du nyss skapat för att sedan bygga upp den på nytt en gång till, eller hänvisa till **3.1.4 Felkällor** för att fixa problemet. En virtuell miljö och dess paket raderas genom syntaxen **conda remove --name [namn på miljö] --all**.

### 3.1.4 Felkällor

När rubriker **3.1.1 - 3.1.3** inte lyckats under arbetets gång var följande det som hjälpte bland lösningar för problemet.

- Stäng av **brandväggen**. Eftersom mjukvaran arbetar nära roten av dina filer kan brandväggen se det som ett skadligt program och därför blockera processen.
- Används ett **offentligt nätverk** vid installation av paket eller mjukvara kan det finnas blockage för vissa delar av programmen. Exempelvis kan Anaconda Prompt inte installeras då Anaconda installeras.
- Filer och **andra mjukvaror** som ligger på datorn kan störa programmet under installationen. Sätt därför upp en virtuell dator enligt **3.1.1 Virtuell Maskin**, vilket tillåter en ren arbetsbänk att arbeta med.
- Kontrollera att den virtuella miljön är baserad på **python** version **3.10.12**. Syntaxen **python --version** borde ge följande resultat.

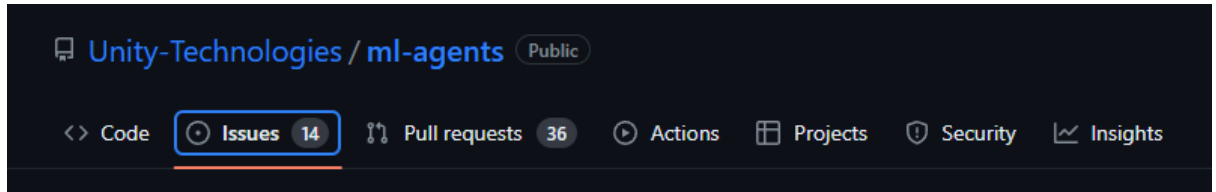
```
(RL) C:\Users\Erik\GyarbRL>python --version  
Python 3.10.12
```

Var säker på att syntaxen är skriven inuti en virtuell miljö eftersom att det är miljöns version som är relevant. För att öppna den virtuella miljön skriver du syntaxen **conda activate [namn på miljö]**.

- Kontrollera att versionen på **numpy** är **1.21.2**. Detta kan ha ändrats ifall paket utanför de ovan nämnda har installerats. Det som borde finnas i miljön är **pip, setuptools,**

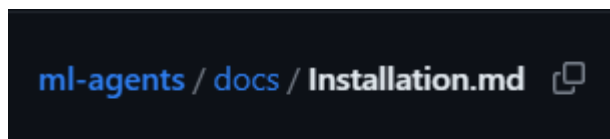
**mlagents och numpy.** Ifall avinstallation av onödiga paket eller återinstallation av krävda paket inte löser problemet följs processen för att ta bort och sedan installera miljön. Processen beskrivs mot slutet av rubrik **3.1.3 Stegvis konfiguration.**

- Möts problem utöver vad dessa tips kan lösa, hänvisades diskussioner hållna på GitHub repository ML-Agents **issues** tab för att hitta och möjligtvis lösa problem.



Användning av tjänsten ChatGPT kan även ge insyn till problem. Men eftersom den data ChatGPT baserar sina svar på är från år 2021 (Nuvarande datum 2024-01-28) kan svaren vara daterade eller vilseledande. ChatGPT är dock värt att ge en chans då ML-Agents släpptes för första gången år 2017.

Allt angivet under rubriken **3.1 Installation och konfiguration** är det som har utförts under arbetet. Dock om inget av ovanstående funkar för andra personer kan det experimenteras med att använda den installationsguide given på GitHub repository ML-Agents dokumentation under följande filväg.



Anledningen till varför guiden inte har följts under arbetets gång är eftersom det har försökts flera gånger utan framgång. Guiden given under rubrik **3.1 Installation och konfiguration av programvara** är baserad på egen provning, ML-Agents officiella dokumentation, samt en diskussion framtagen under GitHub repository ML-Agents issues tab med titel **“Couldn't connect to trainer on port 5004 using API version 1.0.0f, will perform inference instead.”** påbörjad av användare **egedursun**.

## 3.2 Grundläggande kodning av agenter

### 3.2.1 Script

I scripten byggs upp de grundläggande funktioner som agenten ska ha tillgång till. Detta kan innebära allt från att låta den navigera längs x- och z-axeln, tillåta den att hoppa längs y-axeln eller ge den möjligheten att driva en motor till en bil.

Även om agenten vill uppnå mer avancerade beteenden behövs inte den funktionen direkt i koden. Det är istället möjligt att uppnå det beteendet genom att ge agenten verktygen för att



skapa det beteendet själv. Nedanför beskrivs det kortfattat vad arbetets script behöver innehålla för att det ska funka på en grundläggande nivå.

Detta scriptet är grunden till det som användes under arbetsgången, eftersom nya funktioner har skapats efter behov och funktionalitet av agentens inläring och utveckling i beteende. Det slutgiltiga scriptet finns bifogat under **6.1. C# script agenten baserar sitt beteende och inläring utefter**. Vad delar av scripten betyder förklaras under nästa rubrik **3.2.2 Metoder**.

C/C++

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;

public class NamnPåAgent: Agent
{
    void Start()
    {

    }

    public override void OnEpisodeBegin()
    {

    }

    public override void CollectObservations(VectorSensor
sensor)
    {

    }

    public override void OnActionReceived(ActionBuffers
actionBuffers)
    {

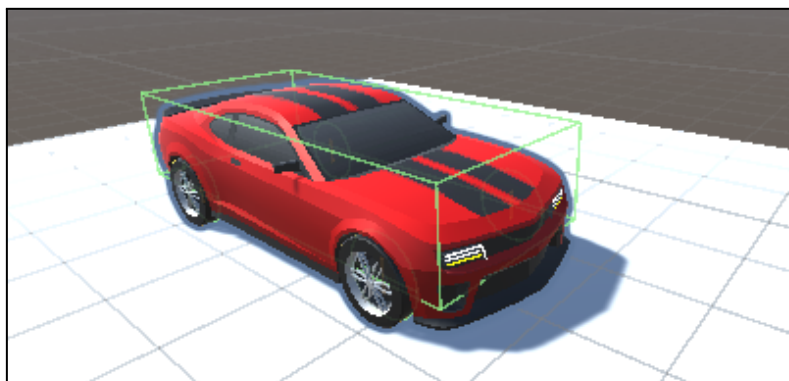
    }
}
```

```
}
```

### 3.2.2 Metoder

Ovan angivna script visar tre agent specifika metoder, men även **Start()** som är en standard unity metod. Vad metoderna **OnEpisodeBegin()**, **CollectObservations()** och **OnActionRecieved()** gör kommer att beskrivas kort. Dock kan dessa metoder konfigureras efter egna behov men används under arbetet för att hjälpa agenten på följande vis.

**Start()** används en gång per simulation när miljön initieras. Under arbetet användes denna metod för att initiera den komponent som tar hand om agentens fysiska barriär. Detta visualiseras nedan genom den gröna låda som omringar bilen och ansvarar för att upptäcka kollisioner mellan olika objekt.



*Det här är modellen för agenten i miljön. (gröna linjerna representerar kollision gränser för agenten)*

**OnEpisodeBegin()** används varje gång ett nytt träningstillfälle börjar. Följande är exempel på när ett träningstillfälle är över och programmet vill påbörja ett nytt. Då agenten har nått sitt mål, agenten har misslyckats med sitt mål eller om den begränsade tiden går ut. För att manuellt avsluta ett träningstillfälle används funktionen **EndEpisode()**. När den används kommer nästa träningstillfälle initieras, vilket är när **OnEpisodeBegin()** används.

Denna metod används främst för att nollställa agentens momentum och position så pass att den kan försöka en gång till istället för att spendera tid för att återvända tillbaka till början.

**CollectObservations()** används för att ge agenten information om sig själv och omgivningen. Denna metod aktiveras flera gånger per sekund. För arbetets syfte är observationer som hastighet, position och rotation av agenten något relativt grundläggande då dessa är till hjälp

för att ha kontroll över ett fordon. En observation kan läggas till med syntaxen **sensor.AddObservation(n)** (n = variabel som observeras).

**OnActionReceived()** används primärt för att tala om för agenten vilka handlingar den kan ta. Detta är möjligt genom att ange följande syntaxen till en variabel, **actionBuffers.ContinuousAction[n]** (n = positivt heltal).


Eftersom även denna metod är kallad flera gånger per sekund kan den användas för att kontrollera specifika tillstånd. Exempel på ett sådant tillstånd är ifall agenten har gjort något den borde eller inte borde ha gjort. När det tillståndet är bemött är det vanligt att ange agenten en belöning eller bestraffning genom syntaxen **SetReward(n)** eller **AddReward(n)** (n = rationellt tal) där ett negativt tal tolkas som en bestraffning medan ett positivt tolkas som en belöning. Skillnaden mellan de två syntaxen är att **SetReward(n)** sätter ett värde för hela träningstillfället. Medan **AddReward(n)** ackumulerar en belöning över ett träningstillfälle eftersom belöningar staplas med denna funktion.

### 3.2.3 Parametrar

Parametrar används för att konfigurera agenten till att träna enligt vissa specifika regler. En av dessa regler är **max\_steps** vilket anger en övre gräns till hur länge agenten kan träna i en virtuell miljö. Det finns även **summray\_freq** vilket anger hur ofta resultaten kring träningen ska skickas till Anaconda Prompt. Siffran i båda dessa tillfällen representeras av **steps**.

För referens tog det vid ett av arbetets träningstillfällen 25 minuter för agenten att genomföra 100.000 **steps**. Dock kan denna siffra påverkas kraftigt genom olika metoder, eftersom vid ett annat tillfälle nådde agenten 100.000 **steps** efter 10 minuter. Skillnaden mellan dessa två miljöer var endast antalet agenter som tränade samtidigt. Antalet agenter som tränade då 10 minuter gällde var fyra gånger större än vid 25 minuter.

Dessa parametrar konfigureras inuti en fil av typen .yaml och om det önskas finns mer information om specifika parametrar bland GitHub repository ML-Agents dokumentation under rubriken “Training Configuration File”. Nedan visas filsökvägen.

[ml-agents / docs / Training-Configuration-File.md](#) 

## 3.3 Utförande

### 3.3.1 Agenten



*Det här är modellen för agenten i miljön.*

Ovan visas en bild på den agenten som har tränats under arbetets utförande. Det önskade beteendet som arbetet söker till att uppnå genom RL har nämnts tidigare under rubrik **1.2 Syfte och frågeställningar** vilket är “hur en riktig bil fungerar i verklig omgivning”.

Arbetets definition på ett beteende som liknar en bil i en verklig omgivning innebär att acceleration, inbromsning, svängradie och manövrering fungerar som på en verklig bil tillåten på allmänna vägar. Vad denna bil ska uppnå, är att så effektivt som möjligt anpassa navigationen mot ett utsatt mål, vilket ska möjliggöras genom RL. Denna definition har dock möjlighet att variera mellan träningsmiljöer då olika beteenden ska åtgärdas i olika syften.

Eftersom att agentens mål är att lära sig köra bil kommer den att behöva en bil som fungerar enligt ovan lagda krav att navigera med. För att uppnå detta utan särskild begåvning inom Unity följdes en youtube video vid titeln “**Simple Car Controller in Unity (Copy Script)**” upplagd av **Prism** år 2022. Anledningen bakom valet är grundad i bilens enkla programmering samt att modellen är tillåten för öppen användning av skaparen. Ett enklare program är både lättare att förstå och effektivare att anpassa till agentens träning, eftersom det blir mindre komplicerat att implementera egna funktioner till koden. Detta gjorde videon relevant för arbetet på grund av att syftet inte angår programmering av bilar.

För att agenten ska nå beteendet av hur en verklig bil fungerar, förväntas agenten kontrollera acceleration, retardation, manövrering och automatisk navigation. Scriptet som tillåter detta beteende baseras på det script som är givet i videon nämnd ovan, men har blivit anpassad för arbetets användning inom RL. Hur det gjordes förklaras till största del under rubriken **3.2 Grundläggande kodning av agenter** på en generell nivå. Slutliga koden är den angiven

under **6.1 C# script agenten baserar sitt beteende och inlärning utefter**. Ta hänsyn till att delar av det script som hänvisas ändras mellan träningsmiljöer. Variabler som kommenteras ut kan exempelvis vara olika observationer eftersom att det varierar mycket utefter behov på information. Under rubriken **3.3.3 Miljöer** kommer det att finnas vilka funktioner och parametrar som var aktiva för träningstillfället.

Parametrar agenten använder är även givna under **6.2 .yaml parametrar för agent**. Om mer information kring arbetets parametrar önskas, hänvisas till rubrik **3.2.3 Parametrar**.

### 3.3.2 Träning

Under träning följdes metoden beskriven under rubrik **1.4 Metodbeskrivning**. Alla beslut vid ändring av data har belägg i resultat från föregående träning i syfte att åtgärda felaktigt beteende, fel med kod eller anpassning av miljö. Denna rubrik beskriver processen i så sann kronologisk ordning som möjligt för att tolka data och omforma miljöer och utveckla agentens beteende.

Flera olika konfigurationer provades för varje miljö där olika problem uppstod. Under rubrik **3.3.3 Miljöer** beskrivs det grundläggande upplägget av miljön samt en bild för att hjälpa till med visualisering av miljön.

Under början av testerna i **3.3.3.1 Träningsmiljö 1** hade agenten en tendens att avvika från initiala målet och välja att köra av banan, men dessutom utnyttja den vänstra halvan av miljöns yta för majoriteten av försöken utan någon anledning eller provokation. Däremot efter längre observation lyckades agenten att hitta ett mer optimalt mönster för att nå till punkten B med en vinglig men framgående kurs med färre slumpmässiga rörelser. Det som är viktigt att lägga märke till i denna träningsmiljö är att agentens target hade för lite variation för att anses vara dynamisk.

Detta orsakade att agenten inte följde sin target, istället lärde den sig ett mönster som hade stor chans att träffa alla möjliga positioner som target kunde befinna sig på.

Utvecklingen av agentens beteende uppnåddes genom funktionen av bestraffning för varje tillfälle agenten åkte av kartan, och belöning när agenten nådde sin target.

Tidiga statistiska värden på belöning vid 10.000 steg under agentens utbildning gav en median på -0.321 poäng. Vid 20.000 steg ökade medianen till -0.234 poäng, vilket tyder på att agenten utvecklar sitt beteende och mönster för att uppnå målet. Detta bekräftades då det visuellt observerades att agenten klart träffade target oftare efter varje träningstillfälle.

Dock observerades i denna miljö underliga beteenden från agenten. Eftersom att den vill träffa sin target så snabbt som möjligt väljer agenten att accelerera till maximal hastighet men har ingen avsikt för att bromsa efter att målet har träffats.

För att ytterligare visualisera detta fenomen ökades ytan i miljön enligt **3.3.3.2 Träningsmiljö 2** och funktionen som återställer bilen till sin plats kommenterades ur agentens script.

Under dessa omständigheter blev uppsättningen av **3.3.3.1 Träningsmiljö 1** ett problem eftersom allt inte löser sig i ett verkligt scenario, när ett fordon har nått sin destination försvinner inte momentum omedelbart. I den bredare miljön åker agenten först rakt fram för att senare efter den träffat sin target göra en våldsamt sväng rakt mot kanten av miljön. Detta beteende har uppmuntrats i den ursprungliga miljön eftersom det inte finns en efterföljd till dess beteende.

Ytterligare observationer under träningsmiljö 2 är att agenten har stor tendens att svänga åt vänster sida. Den har ingen strävan efter att navigera direkt till höger, istället väljer agenten att genomföra en lång vänstersväng för att tillslut orientera sig mot sin target.

För att åtgärda dessa problem designades **3.3.3.3 Träningsmiljö 3**. Metoden för att uppmuntra agenten till att använda sig av både vänster och höger sväng, var att positionera agenter enligt miljöns beskrivning. Därav resulterar i en mer dynamisk träning av agentens beteende under svängning.

Resultatet av denna miljö var att bilen blev för osäker på vad den skulle göra att den fortsatte fram och tillbaka i en liten area runt sin startposition. Tolkningen av detta beteende är att agentens observationer inte är tillräckligt dynamiska för att den ska känna igen skillnaden mellan rotationen även i denna miljö. Då den ena kolumnen lär sig att det är gynnsamt att köra vänster, lär sig den andra kolumnen att höger är gynnsamt. Samma gällde med sväng åt vänster och höger då rotationen orsakade förvirring och efter 500.000 **steps** med en median **return** på **0.00**. Med andra ord har agenten inte lärt sig att träffa sin target, samtidigt som agenten inte har kört av banan.

I ett försök att åtgärda detta designades **3.3.3.4 Träningsmiljö 4** som är designad för att positionera agentens target i alla möjliga vinklar runt agenten. Antalet agenter dras tillbaka till en singulär agent. Under denna miljö framgår den största dynamiska utvecklingen i beteenden.

Till en början hade agenten tillgång till följande observationer. Position av agent och target, agentens hastighet i x- och z-led, distans mellan target och agent. Det som var nytt för denna miljö var observationen rotation kring y-led. Målet med denna observation var att agenten skulle dra kopplingen mellan sin rotation och olika hastigheter i x- och z-led när den accelererar samt svänger.

Agenten var fortfarande endast belönad med en träff av sin target. Detta resulterade i att agenten blev vilse eftersom chansen att den råkar träffa sin target och blir belönad jämfört med föregående miljöer är mycket mindre. Största fokuset var dock fortfarande att lära agenten använda höger och vänster sväng.

Nästa åtgärd var att ändra i agentens parametrar. En parameter vid namn **learning\_rate** bestämmer över hur snabbt agenten känner igen ett mönster som den vill optimera i sin policy. Tolkningen är att agenten innan kände igen ena svängningen som mer gynnsam över den andra och sedan fortsatte med den riktningen. Resultatet av en lägre learning rate gav den viljan att använda sig av höger och vänster riktning. Dock efter en längre träningsperiod kommer agenten att börja optimera igen och möjligtvis väljer den en riktning eftersom den ena sidan slumpmässigt har varit mer gynnsam över den andra.

Under perioden som svängningen tränades blev dock agenten sämre på att träffa sin target. Lösningen för detta var att implementera den första dynamiska belöningen. Belöningen funkade genom att jämföra distansen mellan agent och target efter varje step. Om agenten var närmare än sitt förra step blir den belönad.

Resultatet av detta var att bilen ignorerar belöningen av att träffa sin target och började istället att försöka förlänga tiden den spenderade på väg mot sin target eftersom att den på så vis fick så mycket belöning som möjligt. Agenten visste vart sin target befann sig men undvek den medvetet. Detta resulterade i att agenten körde mot sin target långsamt, sedan missade agenten med mindre än en meters avstånd och förbereder för att vända om och långsamt köra mot sin target igen.

Detta beteendet uppkommer på grund av designen på belöningen. Agenten gynnar inte av att hålla sig nära target under längre tid. För att fixa problemet designades distansbelöningen på nytt. Förra designen belönar agenten när den åker mot sin target, står agenten stilla, får den varken belöning eller straff. Den nya designen bestraffar enbart beroende på distans. Är agenten nära sin target blir den belönad en viss summa, och om den är långt borta blir den bestraffad med en viss summa. Belöningen beror på en siffra som representerar den radie agenten befinner sig runt sin target. Siffran bestämmer gränsen mellan en bra och en dålig distans att befinna sig i. Ett lägre positivt heltal leder till att agenten strävar närmare mot sin target för att sedan stanna och plocka åt sig sin belöning.

Detta resultat är det slutliga som arbetet kommer sträva efter. Agenten kan navigera självständigt och stannar bredvid sin target i nästan varenda träningstillfälle den genomför. Men vad detta beteende innebär för agentens beteende i olika stadier av träning kommer tydligare att visualiseras och förklaras under **4.1 Resultat av agentens träning**.

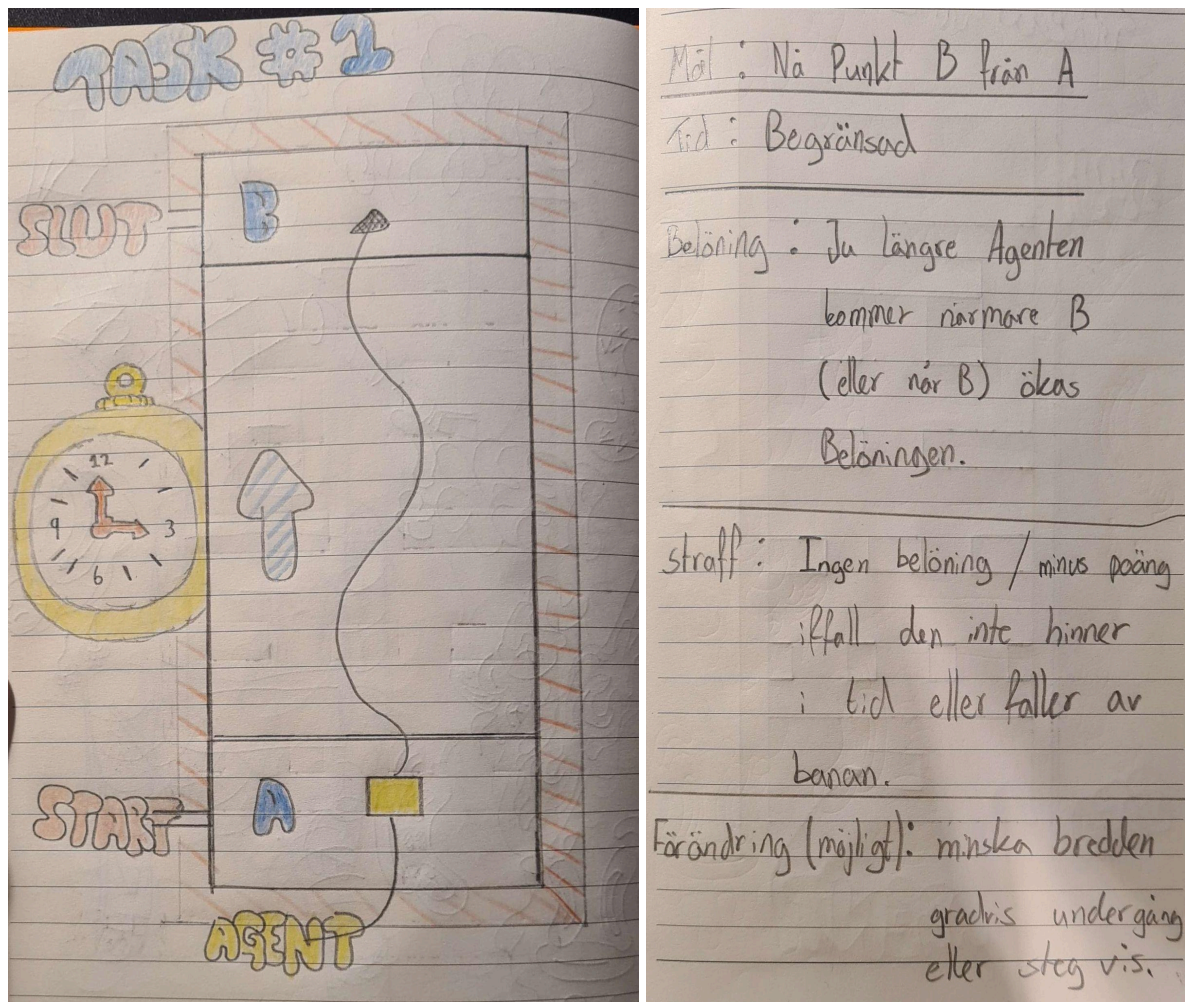


### 3.3.3 Miljöer

Miljöerna har en huvudsaklig uppgift, att utmana agenten till att utveckla nya kunskaper men även utnyttja förflutna lärdomar. Genom att träna agenten på miljöer med specifika egenskaper och utformningar, tränas agentens olika färdigheter, exempelvis svänga i kurvor, anpassa hastighet eller bromsa. Resultat från följande miljöer presenteras ovan under rubriken **3.3.2 Träning..**

#### 3.3.3.1 Träningsmiljö 1

Nedan anges den första skissen och designen för den första träningsmiljö agenten blev utsatt för.



Det här är skisserna för träningsmiljö 1, instruktioner för agenten och miljön.





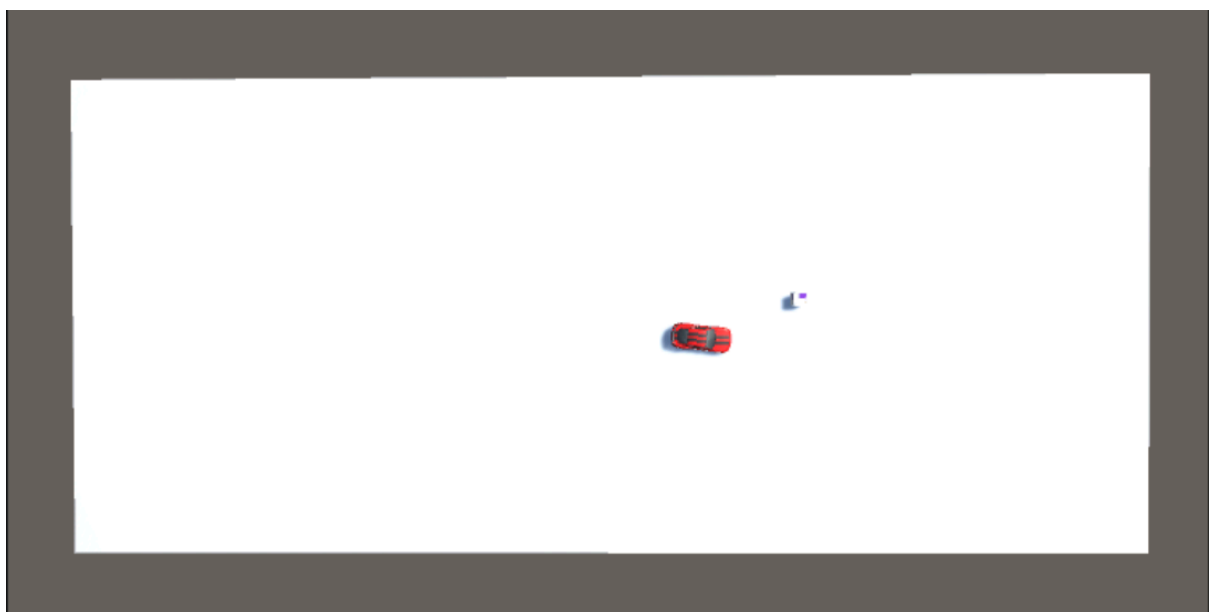
*Det här är slutliga designen för träningsmiljö 1, A representerar **Agent**, B representerar **Target**.*

Det grundläggande målet som agenten hade i denna miljö var att förflytta sig från punkt A till B. Detta målet tvingar agenten att utveckla manövreringsförmågor som tillåter den att inte ramla av kanten på miljön samt navigera mot dess target vilket är avgörande egenskaper för framtida testbänkar.

Det som underlättar agentens utveckling är att den inte har begränsningar på hur den får nå målet. Förutom ett begränsat antal steg den får utföra, under en inlärningsperiod.

Det som gör miljön dynamisk är att agentens target mellan varje inlärningsperiod förflyttar sig till en ny placering. Agentens target flyttas enbart inom det gröna området för att göra miljön relativt dynamisk.

### 3.3.3.2 Träningsmiljö 2

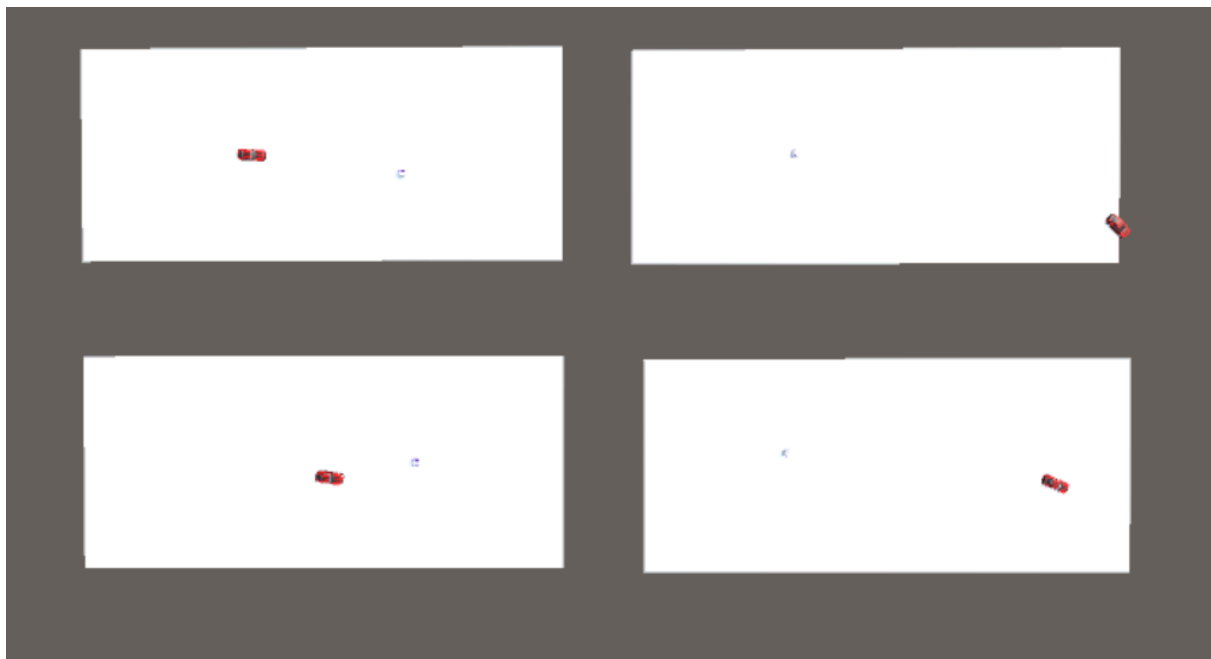


*Det här är design av miljö 2 men bredare yta.*

Syftet med denna miljö var att undersöka ett beteende som var observerat inom **träningsmiljö 1**. Beteendet under detta steg diskuteras senare under rubrik **4.1 Resultat av agentens träning**.

Fokus i miljön ligger inte på träning eftersom dess syfte är observation och tolkning av redan utvecklat beteende. För att säkerställa att beteendet inte var slumpmässigt utan regelbundet infördes en alternativ miljö bildad nedan.

### 3.3.3.3 Träningsmiljö 3

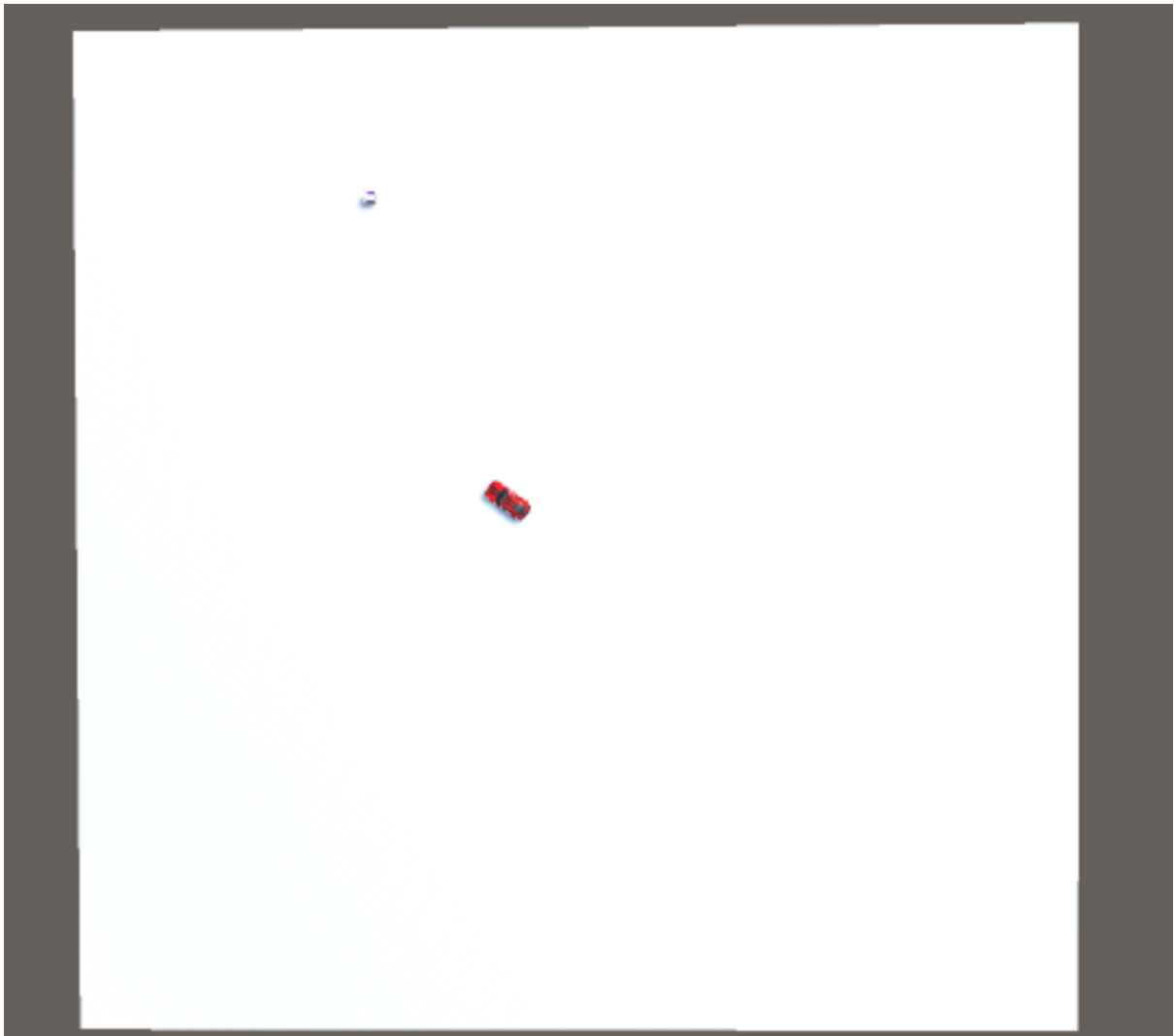


*Det här är fyra stycken kopior av miljö 2 med agenter som tränar samtidigt.*

Denna miljö har till skillnad från träningsmiljö 2 fyra agenter som tränar i samtid. Vänster kolumn är orienterad enligt träningsmiljö 2 medan höger kolumn har roterat 180°. Detta kan visualiseras genom att två personer står och kollar varandra i ansiktet.

Syftet med denna miljö var att undersöka beteendet när flera agenter tränar i samtid. En rotation på 180° stod till att undersöka om agenten hade tillräckligt belägg av observationer för att själv reda ut rotation av bilen och dra lämpliga kopplingar i sin policy, vilket är en stor del av manövrering.

#### 3.3.3.4 Träningsmiljö 4



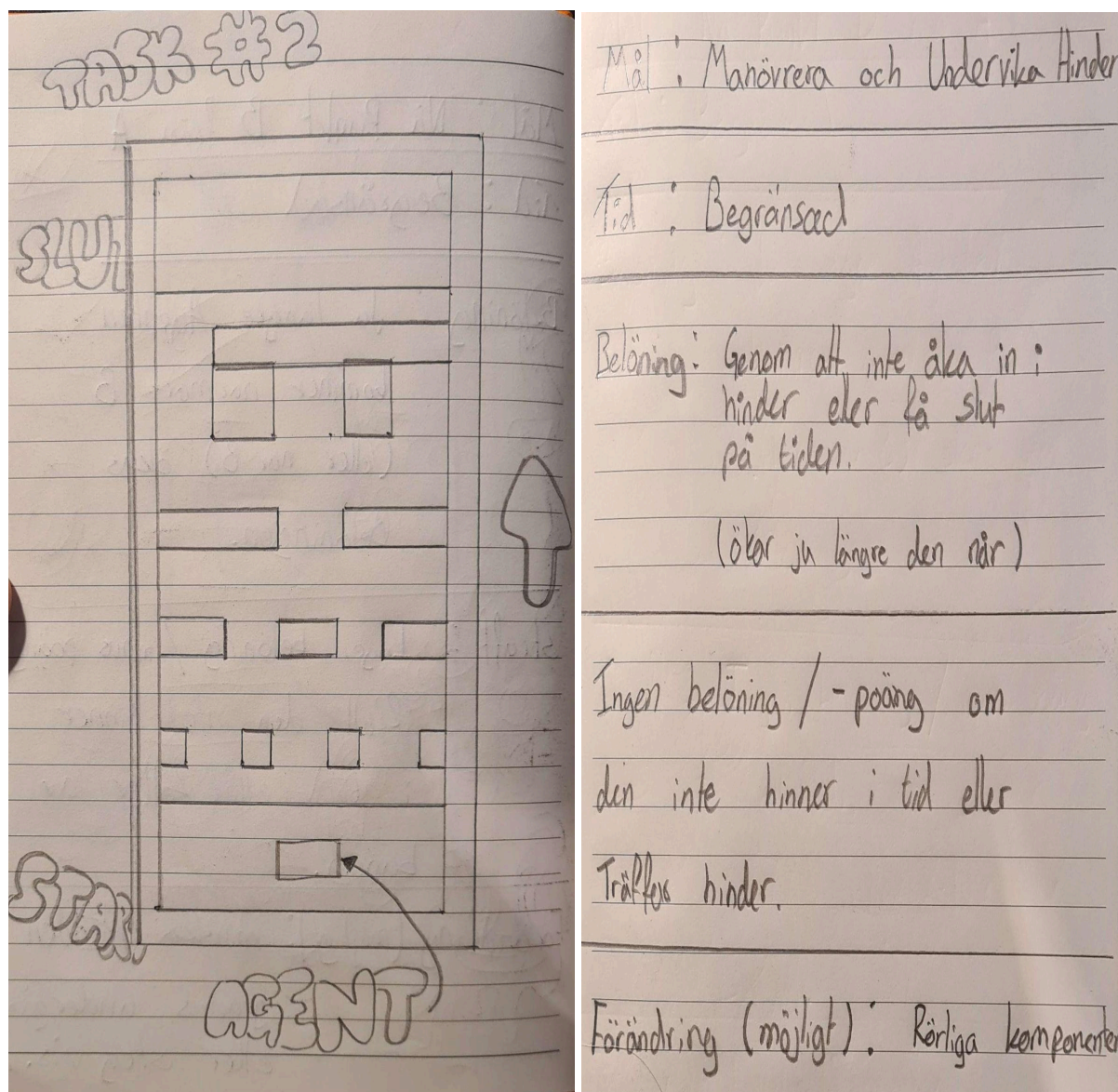
*Det här är design av miljö 4 med target som kan förekomma över hela ytan.*

Syftet med denna miljö är att ytterligare utforska bilens observationer kring rotation. Denna miljö är även den som står som standard inför majoriteten av träningstillfällen.

Upplägget av miljön är följande. På den vita plattan kan agentens target förekomma på vilken plats som helst. Efter att targeten har träffats slumpas den nya positionen och agenten hamnar åter i mitten av plattan med motorhuven riktad vänster.

Denna miljö är den mest dynamiska hittills eftersom agenten har större frihet inom manövrering men även utökade begränsningar i form av beteende, exempelvis fartbegränsning. Utöver de förflutna miljöers bestraffningar används (beroende på träningstillfället) även en hastighetsbegränsning och belöning beroende på distans till target.

### 3.3.3.5 Tänkta miljöer



Det här är skissen för en potentiell träningsmiljö 5, instruktioner för agenten och miljön

Huvudsakliga mål inom den potentiella träningsmiljö 5 är att utveckla agentens förmåga att kunna undvika väggar som hindrar en direkt väg för att nå målet och fortsatt utveckling av manövrerings kunskaperna i begränsade ytor.

Om det fanns tid skulle det kunna leda till potentiell utveckling hos agentens förmågor, men det skulle kräva att vid kollision med hinder bestraffa agenten med negativt antal poäng. Vilket skulle leda till att agenten undviker dessa beteenden där den bestraffas, och sedan leda till att den utvecklas utifrån dem.

För att implementera denna miljö skulle det kräva utökad tid för observationer, tolkningar av världen, uppbyggnad av miljön men dessutom att agenten har nått godtagbara kunskaper för att påbörja den komplexa utbildningen. Men för att tydliggöra har träningsmiljö 5 inte används under arbetsgången men har studerats för potentiell implementering.

## 4. Resultat, utvärdering och diskussion

### 4.1 Resultat av agentens träning

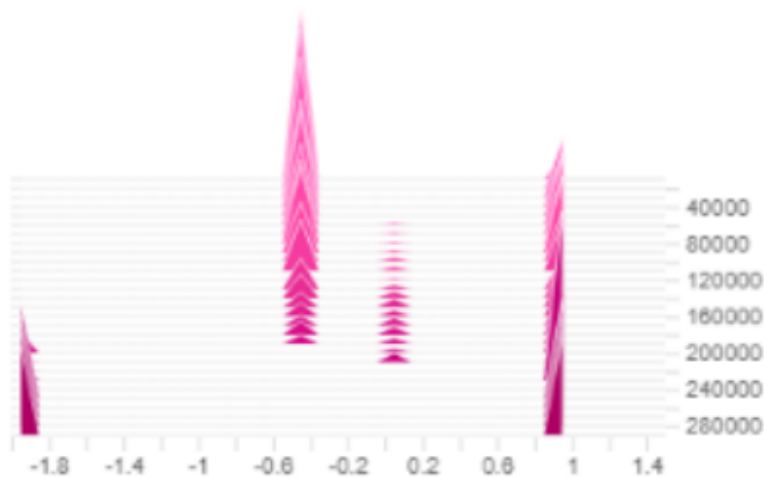
Resultatet för alla träningstillfällen skulle göra tolkningen ytterst svår att sammanställa på grund av att majoriteten av insamlad data användes för att utöka förståelsen av agentens försök att navigera och tolka sin miljö. Således valdes tre av de mest framgående och skildrande resultaten för visning som strävar efter att visa resultat från de viktigaste delarna av träningen. Grafer i färg rosa visar **3.3.3.1 Träningsmiljö 1**. Gul och svart representerar olika stadier av **3.3.3.4 Träningsmiljö 4**. När det gäller träningsmiljö 2 och 3 fanns inte tillräckligt med data för att framställa ett konkret resultat. Dessutom användes inte träningsmiljö 2 och 3 för träning, istället kommer träningsmiljö 1 och 4 användas för att motivera olika beteenden.

För var av de tre utvalda träningstillfällen kommer grafer för följande att presenteras.

**Sammanfattning av belöning och straff** genom träningens gång. **Policy entropy** beskriver hur slumpmässiga agentens handlingar är genom träningens gång. **Episode length** beskriver längden på ett träningstillfälle genom tid.

Först tolkas resultatet av **belöning och straff**. Nedan visas grafer över antalet **steps** i ett träningstillfälle på y-axel och på x-axel visas den belöning agenten tog emot.

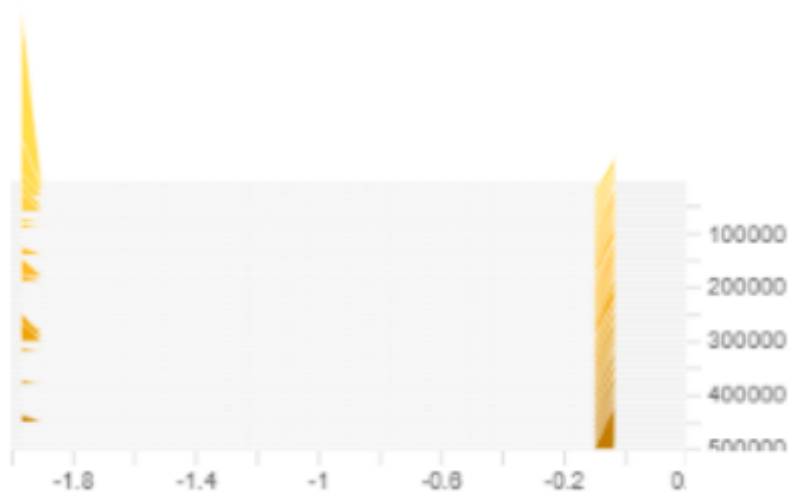
**Rosa grafen för belöning och straff** beskrivs från vänster till höger enligt följande: Straff för att kollidera med marken; Straff för att ramla av kanten av miljön; Belöning för att inte ramla av (tog bort mitt i träning); Belöning för att träffa target.



Tabell för Träningsmiljö 1, return/step.

Hur denna graf tolkas är genom att den efter ungefär **200.000 steps** avviker från dåligt beteende medan belöningen från sin target är regelbunden. Förklaringen för alla kollisioner som börjar mot slutet av träningen är att bilen lyckades hamna på sin kant medan träningen ej var observerad. Detta gav felaktig data mot slutet av träningen. Dock under tiden träningen observerades syntes jämn förbättring.

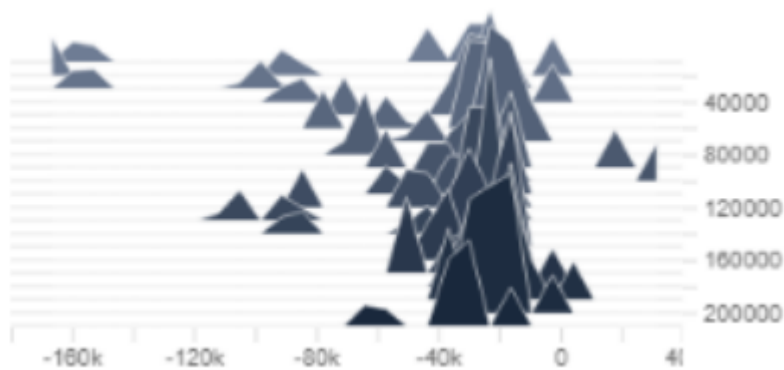
**Gula grafen för belöning och straff** beskrivs från vänster till höger enligt följande: Straff för att ramla av kanten av miljön; Straff på att ta längre tid på sig.



Tabell för Träningsmiljö 4, situation, return/step.

Hur denna graf tolkas är genom att lägga till belöningen den fick från att närma sig sin target medan den samtidigt blev bestraffad av att ta tid på sig. Agenten sökte inte efter att få en positiv belöning. Den sökte för att balansera den första vinsten den hittade. Dock slutade bilen att falla från kanten av miljön mot slutet av träningen med få undantag.

**Svarta grafen** för **belöning och straff** är inte lika enkel att beskriva som resten. Den dynamiska belöningen ger ett bredare spektrum att tolka. Men den enda funktionen som bestraffar eller belönar agenter är **distansen** från sin **target**.

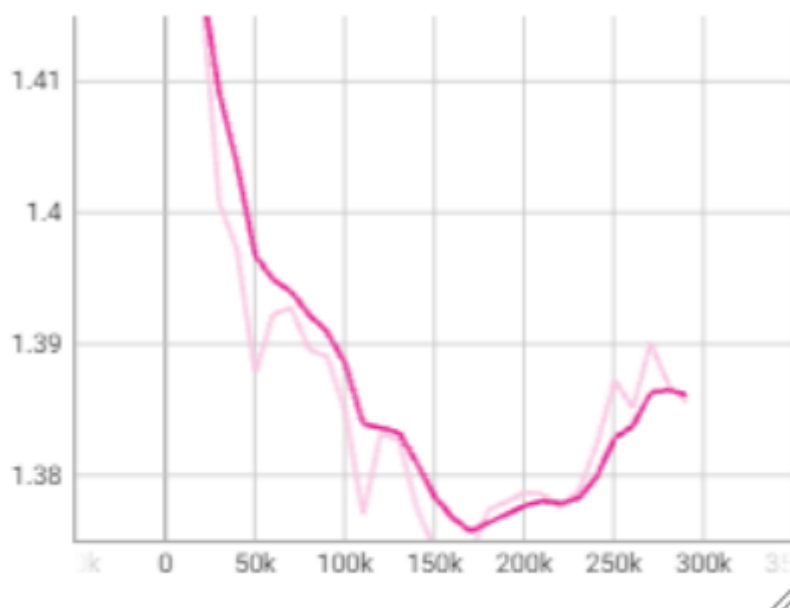


Tabell för Träningsmiljö 4, situation 2, return/step.

Hur denna graf tolkas är genom att den gradvis avsmalnar ju fler **steps** agenten tar. Vilket visar ett snabbare och bättre beteende.

Näst tolkas resultatet av **policy entropy**. Nedan visas grafer över **entropy**, eller slumpmässigt beteende på y-axel och på x-axel visas totala antalet **steps** träningen genomgått.

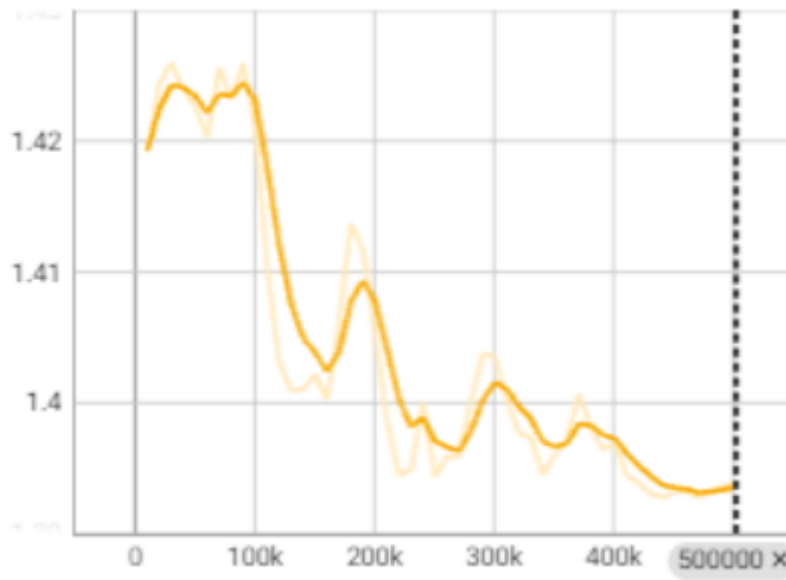
**Rosa grafen** för **policy entropy** visas nedanför och är den som är minst dynamisk mellan de olika träningstillfällena och kan förklaras enligt följande.



Eftersom miljön är så pass statisk finns det inte mycket variation i beteende som leder till bra resultat. Detta gör att agentens **policy** blir mindre slumpmässig efter en kort träningsperiod.

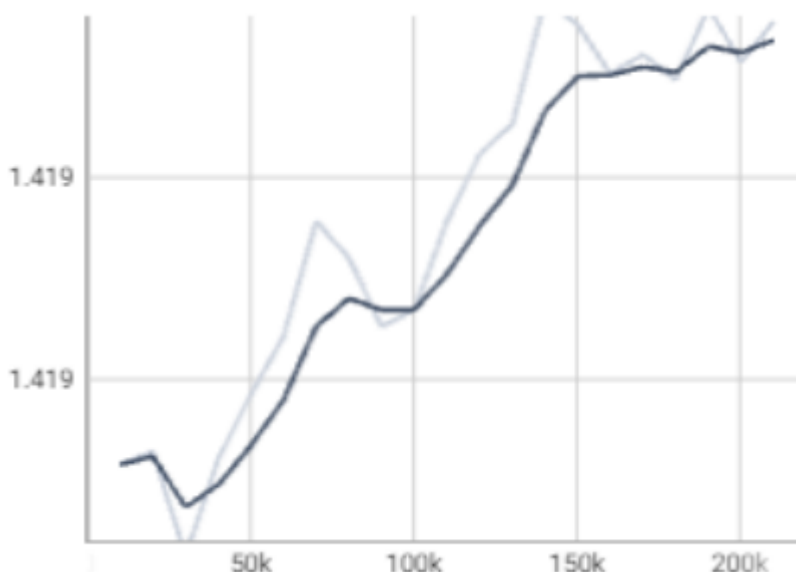
Varför **entropy** stiger mot slutet har samma förklaring som tidigare rosa graf och kommer att gälla för följande rosa grafer.

**Gula grafen** för **policy entropy** visas nedanför och kan förklaras enligt följande.



Grafen har ett stadigt avtagande under träningens gång då agentens **policy** blir mer anpassad. Varför det finns delar av grafen som drastiskt skiftar upp och ner är på grund av att miljön är mer dynamisk och kommer därför ibland att träffa på liknande förutsättningar som förra träningstillfället och ibland en helt ny situation. Trots detta ser vi en stadig minskning i slumpmässigt beteende över tid då den generellt blir bättre på att navigera.

**Svarta grafen** för **policy entropy** visas nedanför och kan förklaras enligt följande.

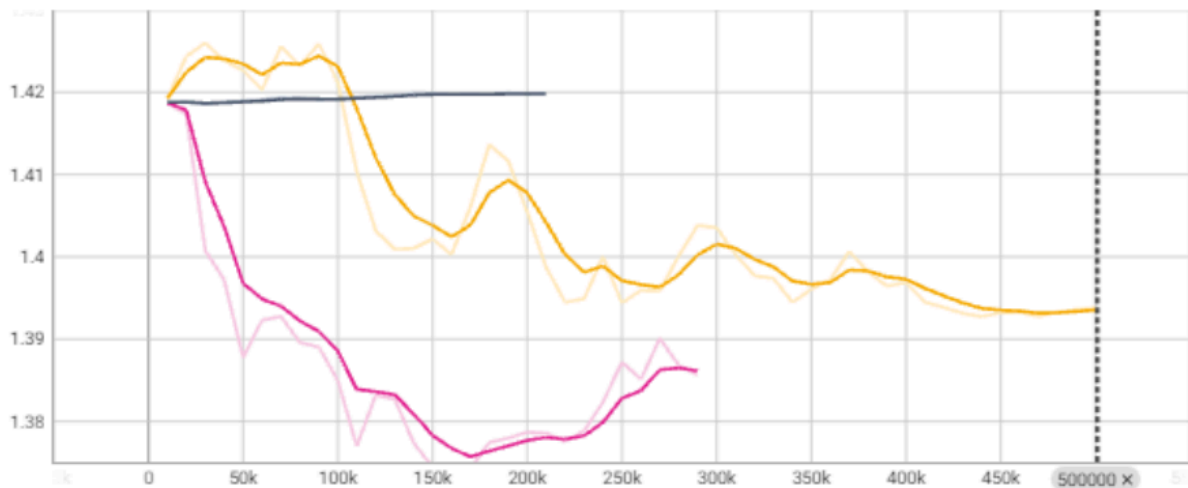


Varför denna graf till skillnad från föregående grafer stiger är att detta träningstillfälle snabbare fick resultat som upplevdes som stabila och regelbundna att bygga vidare på, då den



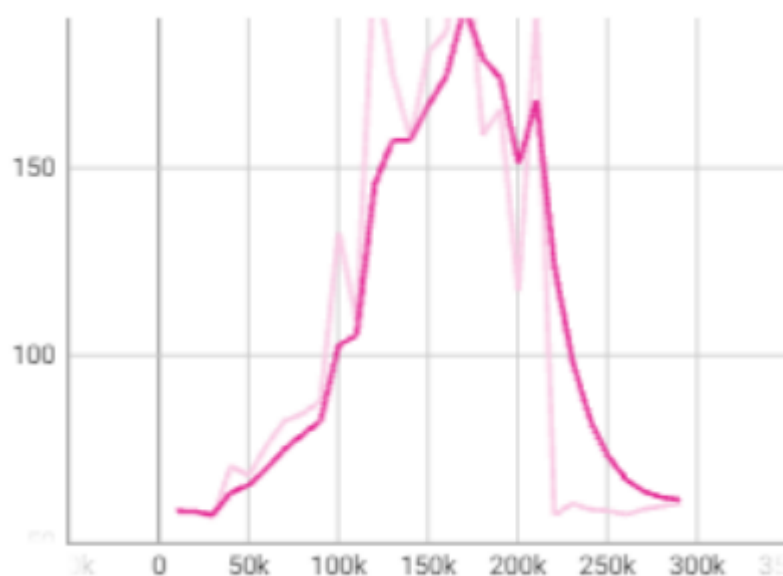
endast hade en belöning att anpassa. Dock var detta inte tillräckligt med tid för att grafem ska börja avta. Dock ser vi att agentens **policy** förbättras efter cirka **150.000 steps**, vilket vi även ser i den första **svarta grafen** då fler belöningar blir positiva.

Om värdena som representerar **entropy** jämförs mellan grafer visar det sig att skillnaden mellan den **svarta** och de andra två graferna har en stor skillnad i spridning. För att visa detta anges nedan alla data i samma graf.



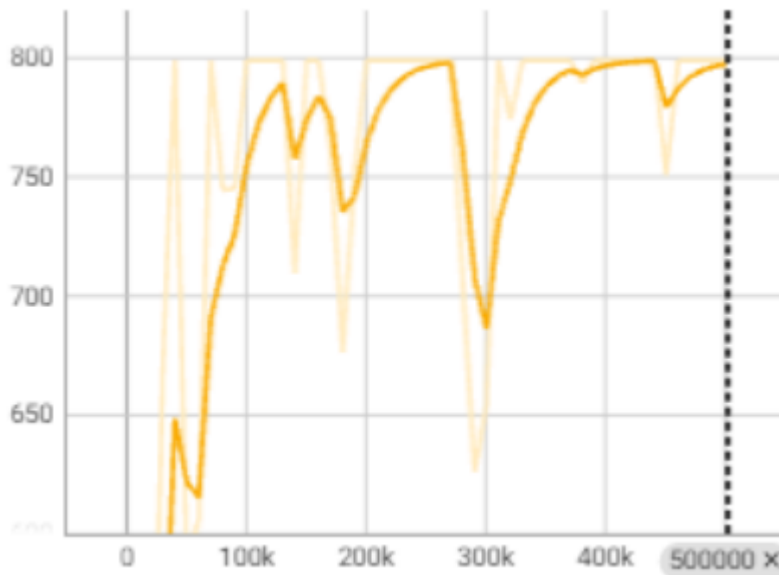
Sist tolkas resultatet av **Episode length**. Nedan visas grafer över **episode length** eller hur många **steps** agenten tar per träningstillfälle på y-axel och på x-axel visas totala antalet **steps** träningen genomgått.

**Rosa grafen** för **episode length** visas nedanför och är den som har kortast träningstillfällen. Den kan förklaras enligt följande.



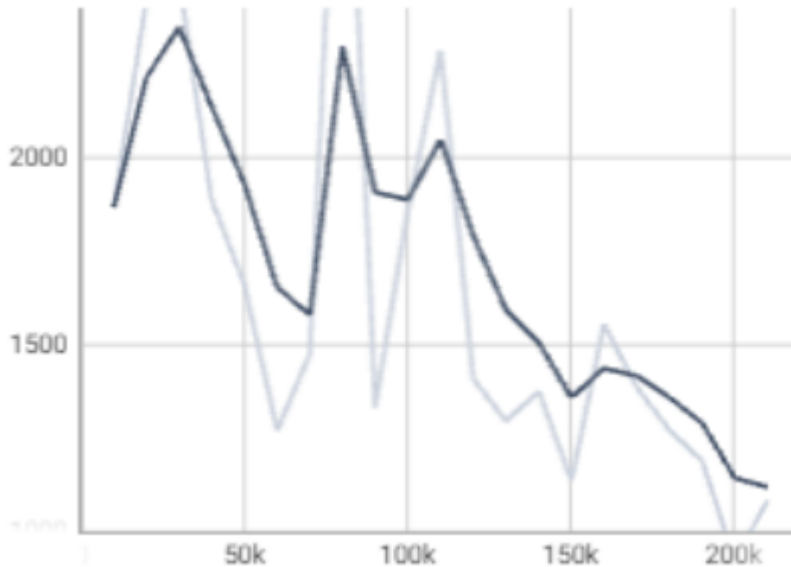
När träningen började hittade agenten inte sin **target** och ramlade konstant över kanten, vilket resulterade i korta träningstillfällen. Dock blev agenten med tiden säkrare på att hitta sin **target**, vilket resulterar till att agenten snabbt och relativt konstant träffar sin **target**. Detta säkra beteende började vid cirka 120.000 **steps**. Här visualiseras även då agenten hamnade på sin kant då längden på träningstillfällen störtade vid 220.000 **steps**.

**Gula grafen** för **episode length** visas nedanför och är den med teoretiskt längsta träningstillfällen. Den kan förklaras enligt följande.



Varför den är teoretiskt längst är på grund av att gränsen för antal steg i detta träningstillfälle var 800 **steps**. Varför agenten nådde gränsen i början var att den utvecklade sin manövrering utan att åka av kanten vare sig träffa sin **target**. Efter cirka 100.000 **steps** började agenten effektivt styra mot sin **target** vilket vi ser i övre grafer. Detta resultat visas dock inte här eftersom agenten inte ansåg det värdefullt att träffa sin **target** och fortsatte istället att köra mot den men inte på den. Detta bidrar även till det teoretiskt längst träningstillfället. Beteendet är beroende på hur funktionen för distans var programmerad och ändrades inför träningstillfället med **svarta grafer**.

**Svarta grafen** för **episode length** visas nedanför och är den med praktiskt längsta träningstillfällen. Den kan förklaras enligt följande.



Anledningen bakom långa träningstillfällen i början var att agenten lärde sig grundläggande manövrering. Ibland råkade den träffa sin **target** genom att spontant manövrera vilket orsakade drastiska skiftningar i grafen. Men enligt vad vi ser här och i grafen för **belöning och straff** finns det ett klart förbättrat beteende. Vid cirka **150.000 steps** ser vi här att beteendet nått ett bra resultat då **agenten** manövrerar både mot och på sin **target**.

När det gäller att besvara arbetets frågeställning ställd under **1.2 Syfte och frågeställningar** kommer ovan beskrivet beteende och observationer under rubrik **3.3.2 Träning** att användas för att ge ett svar så nära sanningen som möjligt.

## 4.2 Svar på frågeställning

### 4.2.1 Hur kan reinforced machine learning träna en bil till att navigera i en virtuell miljö med hjälp av positiv och negativ reinforcement?

Positiv och negativ reinforcement funkar väl för att träna en bil i en virtuell miljö. Det som tagits reda på angående frågan är flera olika saker, majoriteten av dem härstammar från att en bil är en avancerad sak att kontrollera. Inom offentligt tillgängliga demos för RL finns det mycket enkla agenter som kan med hjälp av grundläggande kodning resultera i praktiskt felfria agenter som uppträder precis som förväntat. Dock när det gäller funktionerna av en bil så blir situationen snabbt komplicerad. Det som märktes tidigt i arbetet var att statistiskt tillgiven negativ och positiv reinforcement inte är tillräckligt i alla situationer. När det gäller det syfte vi ville att agenten skulle uppnå var detta ett av fallen. Därför är det viktigt att anpassa reglerna till agentens belöningar så att de är anpassade för agentens beteende och syfte.

I detta arbete löstes detta genom en dynamisk regel som beroende på distansen från agentens target antingen belönade eller straffade agenten. Regeln var utformad enligt följande.

```
AddReward(X - Vector3.Distance(this.transform.localPosition, Target.localPosition));
```

X byts ut för en konstant som representerar hur långt avstånd agenten får ha till sin target innan den blir straffad.

Varför statiska regler inte fungerar lika väl som dynamiska i fallet av denna agenten är att agentens träningsstillfällen är relativt långa. Detta kan liknas till om en person springer ett maraton. En statisk regel skulle ge personen en belöning för att maratonet avklarades men ger inte några synpunkter på hur personen sprang. En dynamisk regel ger konstant respons i koppling med vad som händer, så om personen ramlar märker den att farten saktade ner och säger åt personen att inte göra det igen genom att straffa personen.

Med hjälp av den dynamiska regeln är det enklare för både personen och för agenten att dra kopplingar till vad som orsakar resultatet, eftersom personen blev tillsagd när något hen gjorde resulterade i en dålig sak.

På grund av ovanstående är det viktigt att reda ut en regel utefter syftet och önskade beteendet av agenten som tränas. Men att regeln ska vara dynamiskt anpassad visar arbetet starka belägg för.

#### 4.2.2 Vilka olika sorters hinder ger mest utmaning för bilen?, Varför?

Under träning av agenten observerades flera hinder som utmanade agentens beteenden. Exempel på hinder är skarpa kurvor, när agenten hamnar i sidled med target och en dynamisk miljö efter att agenten tränat statiskt.

Skarpa kurvor var problematiska för agenten då agenten ville utföra sitt syfte snabbt. Detta leder till att agenten accelererar in i kurvan eftersom agenten försöker svänga så snabbt som möjligt. Dock resulterar detta i att agenten voltar och kan därför inte fortsätta träna. Den första lösningen till detta under träning blev att straffa agenten då bilen roterar över en viss vinkel i y-led, med andra ord när den voltar. Dock funkade det inte att bestraffa självaste volten, istället bestraffas agenten slutligen då dess hastighet är över en viss gräns. Resultatet av detta blev att agenten saktade ner, vilket eliminerade agentens volter. Detta är samma princip som förklarades i frågan ovan, nämligen dynamiska straff och belöningar.

Då agenten är placerad bredvid sin target hamnar agenten i ett stadie då den inte vill backa ifrån, men att åka framåt resulterar även i en negativ belöning eftersom svängnings axeln på bilen som agenten styr har begränsningar. Detta hinder var främst problematiskt under tidigare delar av träningen då agenten strulade att svänga både vänster och höger. Lösningen

som användes under träning var ett samarbete mellan en långsammare inläring samt metoden för att dynamiskt belöna agenten beroende på distansen till agentens target.

En dynamisk miljö var ytterligare en utmaning för agenten då den tränat inom en statisk miljö. Anledningen är att agenter i en statisk miljö optimerar ett enda mönster då agentens target alltid befinner sig på samma ställe. Dock kräver en dynamisk miljö ett optimerat beteende, inte mönster. Skillnaden är att beteendet ändrar sig beroende på situation medan mönstret aldrig ändrar på sig beroende på situation.

Viktigt att lägga på minnet är att detta resultat inte gäller för alla projekt inom RL. Även om ett projekt följer samma syfte som ett annat är det inte garanterat att möta samma hinder då majoriteten av hinder är resultat av hur koden för agenten är formad.

#### 4.2.3 Hur borde arbetet genomföras för att undvika dålig träningsdata för agenten, som skulle kunna vilseleda eller felaktigt utbilda den?

Dålig träningsdata undviks i regel genom att kontrollera varje funktion, observation eller hinder som programmeras in. Om det är ett värde som agenten ska tolka behöver programmeraren först kontrollera att det stämmer överens med det förväntade värdet. Under arbetet uppnåddes detta genom den inbyggda metoden `Debug.Log()` som skickar ett meddelande till Unity konsolen under träning.

Detta var bland annat metoden för att undersöka vilken maximala hastighet agenten får navigera i, och vid vilken vinkel agenten började stöta på problem och voltade bilen.

Ytterligare är det viktigt att stegvis utbilda agenten så pass att den inte blir vilseledd av för mycket ny information och väljer att koppla fel data till fel uppförande. Ska en person lära sig att simma börjar en simskola med att låta den flyta runt i poolen, förstå att när personen andas in flyter hen bättre och att när hen trycker ner med armarna trycks hens kropp upp. Om agentens mål vore att lära sig simma innan den lär sig flyta, kan det hända att agenten drar en koppling mellan sin andning och hur snabbt den simmar, eller sin rotation och varför den flyter.

Under arbetet möttes detta problem tidigt då flertal observationer var givna till agenten. Som tur var märktes detta tidigt och rättfärdigades innan de data samlades in för grafer visade i **4.1 Resultat av agentens träning**.

## 4.3 Diskussion

Arbetsgången som arbetet följt förblev enligt det teoretiska uppställt enligt **1.4**

**Metodbeskrivning.** Det steg under processen som ytterligare kunde undersökas är främst data. Data agenten har tillgång till kan utvecklas i många olika led. Olika sätt att göra detta vore antingen genom att djupare undersöka olika metoder för olika dynamiska belöningar och straff för att tydligare vägleda agenten.

Utanför belöningar och straff finns det även mycket olika parametrar att prova för agenten. Parametrar har nämnts på olika ställen i arbetet, dock eftersom att parametrar är en sak som finjusterar agenten, hamnade arbetets fokus inte i detta område. Fokuset på arbetet var att ge övergripande syn på fenomenet för att enklare kunna ge in synvinklar till hur det kan användas i olika syften. Att perfekt utbilda en agent är obegripligt under det tillgängliga tidsfönstret då kunskaper om RL var avsevärt få vid början av arbetet. Det parametrar som slutligen användes finns bifogade under **6.2 .yaml parametrar för agent**.

När det gäller det önskade resultatet av arbetet behövde agentens beteende anpassas för att hinna inom det angivna tidsfönstret. Under **3.3.3.5 Tänka miljöer** visas exempelvis de miljöer som planerats men inte genomförts. Anledningen bakom detta val var att agentens utveckling krävde mer inlärning än vad som tidigare förväntats.

Grafer som valdes inför **4.1 Resultat av agentens träning** för att representera färdigheten av den slutliga agenten baserades på det som upplevs relevant för att motivera olika beteenden. Många träningstillfällen innan de som visas har resulterat i små fel som åtgärdades utan större redovisning i arbetet. Ett exempel på sådant fel är det som visas i slutet på rosa grafer under **4.1 Resultat av agentens träning** då agenten enligt tidigare beskrivet voltade bilen och inte kunde fortsätta träna. Dock eftersom att dessa fel inte påverkat agentens fortsatta prestanda har det inte redovisats i större del under arbetet. Det primära felet som möttes är som nämnda under **4.2 Svar på frågeställning : “Vilka olika sorters hinder ger mest utmaning för bilen?, Varför?”**.

Överlag har inriktningen på arbetet fortsatt enligt den ursprungliga tanken då många av de planerade miljöerna utfördes och ett gott resultat har nåtts och tolkats för läsare att bygga vidare på.

En tanke som funnits sedan tidigare delen av arbetet är “Hur ska arbetet kunna implementeras i verkligheten?”, nämligen att byta ut “virtuell” mot “verklig” i arbetets titel. Efter att arbetet utförts är svaret tydligare. Agenten som tagits fram är inte tillräckligt utbildad för att styra ett verkligt fordon i en främmande miljö. Fler beteenden som agenten utför skulle behöva tränas bort, och nya beteenden behöver införas för att göra den säkert att använda. Exempelvis har

agenten ännu ingen tolkning av vad som ska undvikas på y-led, exempelvis väggar. Detta är på grund av att agenten inte har tränat på specifikt detta.

Trots detta kan ett teoretiskt svar bildas på "Hur ska arbetet kunna implementeras i verkligheten?". Vi förmodar att agenten är fullständigt utbildad och anpassad till en verklig miljö. Hur denna agent skulle implementeras blir genom att ge den tillgång till all information som Unity ger den under virtuell träning. Detta inkluderar miljön den befinner sig i, igenkänning av olika föremål och fenomen. Observationer om sin egen hastighet, svängradie och plats i miljön. Detta kan möjliggöras genom olika sensorer och algoritmer för igenkänning, i syfte att försöka göra den verkliga miljön så pass virtuell som möjligt då det är enklare för en maskin att tolka det den kan integrera med. Det som krävs för att möjliggöra styrmoment är att ge agenten en länk till den fysiska världen, exempelvis genom hydraulik.

Om ovan implementation införts kommer produkten bidra till automation av transport, eftersom träningen genomfördes med en personbil blir godset i detta fall människor. Så länge som lagen tillåter kommer en korrekt utbildad agent kunna bli implementerad på allmänna vägar. Den nivå som arbetet genomförde implementering av RL på är dock inte tillräckligt för att släppa ut en färdig produkt på allmänna vägar. Det arbetets produkt bidrar till är en unik insyn på fenomenet RL för att förenkla utveckling för nybörjare som vill utforska projekt inom RL.

Om arbetet skulle återigen genomföras från en ny grund skulle olika saker förbättras. För det första skulle installation och konfiguration av mjukvara förenklats genom att följa den fungerande metoden som arbetet tog fram under dess utveckling. Den största mentala ansträngningen för att genomföra arbetet var nämligen att konfigurera den virtuella miljön eftersom att den officiella guiden inte fungerade som förväntat. En snabb konfiguration skulle leda till mer tid för design och träning av agent istället för miljö.

Det största bidraget till ett enklare utförande är förkunskaper inom området, därför skulle detta arbete vara en bra start för nya intresserade av RL för att påbörja sina egna projekt.

## 5. Källförteckning

Caleb M. Bowyer, Ph.D Candidate (Jul 24, 2022). "A Crude History of Machine Learning(RL)", Medium.com

Carapuço João, Neves Rui och Horta Nuno (Dec 2018). "Reinforcement learning applied to Forex trading" ScienceDirect.com

Du Sautoy, Marcus (2020). “De Skapande Maskinerna”, Stockholm: Natur & Kultur

Géron, Aurélien (2019). “Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow”, Kalifornien: O'REILLY.

Kaelbling Leslie Pack, Littman Michael .L, Moore Andrew .W (1996) “Reinforcement Learning: A Survey”, Los Angeles County: Journal of Artificial Intelligence Research

Lillicrap Timothy (Aug 2017). “StarCraft II: A New Challenge for Reinforcement Learning”, Arxiv.org

Unity Technologies (2024). “ml-agents”, github.com

## 6. Bilagor

### 6.1 C# script agenten baserar sitt beteende och inlärning utefter

Alla funktioner av denna script användes inte vid alla träningstillfällen.

```
C/C++
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;

public class CarController : Agent
{
    private float horizontalInput, verticalInput;
    private float currentSteerAngle, currentBreakForce;
    private bool isBreaking;
    private Rigidbody rBody;
    private float dis;
```



```

private float distan;

// Settings
[SerializeField] private float motorForce, breakForce, maxSteerAngle;

// Wheel Colliders
[SerializeField] private WheelCollider frontLeftWheelCollider,
frontRightWheelCollider;
[SerializeField] private WheelCollider rearLeftWheelCollider,
rearRightWheelCollider;

// Wheels
[SerializeField] private Transform frontLeftWheelTransform,
frontRightWheelTransform;
[SerializeField] private Transform rearLeftWheelTransform,
rearRightWheelTransform;

[SerializeField] private Transform Target;

private void Start()
{
    rBody = GetComponent<Rigidbody>();
    Debug.Log("Start");
}

private void GetInput()
{
    // Steering Input
    horizontalInput = Input.GetAxis("Horizontal");

    // Acceleration Input
    verticalInput = Input.GetAxis("Vertical");

    // Breaking Input
    isBreaking = Input.GetKey(KeyCode.Space);
}

//CAR FUNCTIONS
private void HandleMotor(float verticalInput)
{
    frontLeftWheelCollider.motorTorque = verticalInput * motorForce;
    frontRightWheelCollider.motorTorque = verticalInput * motorForce;
    currentBreakForce = isBreaking ? breakForce : 0f;
}

private void ApplyBreaking(float currentBreakForce)
{
    frontRightWheelCollider.brakeTorque = currentBreakForce;
    frontLeftWheelCollider.brakeTorque = currentBreakForce;
}

```

```

        rearLeftWheelCollider.brakeTorque = currentBreakForce;
        rearRightWheelCollider.brakeTorque = currentBreakForce;
    }

    private void HandleSteering(float horizontalInput)
    {
        currentSteerAngle = maxSteerAngle * horizontalInput;
        frontLeftWheelCollider.steerAngle = currentSteerAngle;
        frontRightWheelCollider.steerAngle = currentSteerAngle;
    }

    private void UpdateWheels()
    {
        UpdateSingleWheel(frontLeftWheelCollider, frontLeftWheelTransform);
        UpdateSingleWheel(frontRightWheelCollider, frontRightWheelTransform);
        UpdateSingleWheel(rearRightWheelCollider, rearRightWheelTransform);
        UpdateSingleWheel(rearLeftWheelCollider, rearLeftWheelTransform);
    }

    private void UpdateSingleWheel(WheelCollider wheelCollider, Transform wheelTransform)
    {
        Vector3 pos;
        Quaternion rot;
        wheelCollider.GetWorldPose(out pos, out rot);
        wheelTransform.rotation = rot;
        wheelTransform.position = pos;
    }

    //COLLISION DETECTION
    private bool isTouchingSomething = false;
    private bool isTouchingTarget = false;

    private void OnCollisionStay(Collision collision)
    {
        // Check if the car is touching inappropriate object
        if (collision.gameObject.CompareTag("DoNotHit"))
        {
            isTouchingSomething = true;
        }
        // Check if the car is touching appropriate object
        if (collision.gameObject.CompareTag("DoHit")){
            isTouchingTarget = true;
        }
    }

    private void OnCollisionExit(Collision collision)
    {

```

```

        // Reset the flag when the car is no longer colliding
        isTouchingSomething = false;
        isTouchingTarget = false;
    }

    //AGENT METHODS
    public override void OnEpisodeBegin()
    {
        // If the Agent fell or rolled over, zero its momentum
        /*if (this.transform.localPosition.y < 0 ||
this.transform.localRotation.eulerAngles.x <= -50f * Mathf.Deg2Rad ||
this.transform.localRotation.eulerAngles.x >= 50f * Mathf.Deg2Rad ||
this.transform.localRotation.eulerAngles.z <= -50f * Mathf.Deg2Rad ||
this.transform.localRotation.eulerAngles.z >= 50f * Mathf.Deg2Rad)
        {*/
            Debug.Log(Vector3.Distance(this.transform.localPosition,
Target.localPosition));
            //SetReward(-2f);
            this.rBody.angularVelocity = Vector3.zero;
            this.rBody.velocity = Vector3.zero;
            this.transform.localPosition = new Vector3( 0, 0.5f, 0);
            this.transform.rotation = new Quaternion( 0, 0, 0, 0);
        //}
        // Move the target to a new spot

        Target.localPosition = new Vector3(UnityEngine.Random.value * 80 - 40,
            0.5f,
            UnityEngine.Random.value * 80 - 40 );
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        // Target and Agent positions
        sensor.AddObservation(Target.localPosition);
        sensor.AddObservation(this.transform.localPosition);
        sensor.AddObservation(this.transform.rotation.y);
        sensor.AddObservation(Vector3.Distance(this.transform.localPosition,
Target.localPosition));

        // Agent velocity
        sensor.AddObservation(rBody.velocity.x);
        sensor.AddObservation(rBody.velocity.z);
    }

    public override void OnActionReceived(ActionBuffers actionBuffers)
    {

```

```

float verticalInput = actionBuffers.ContinuousActions[0]; // Acceleration
float horizontalInput = actionBuffers.ContinuousActions[1]; // Steering

// Apply the continuous inputs
HandleMotor(verticalInput);
HandleSteering(horizontalInput);

// Rewards

//AddReward(-1.0f);

/*
// Distansfunktion 1
dis = distan;
distan = Vector3.Distance(this.transform.localPosition,
Target.localPosition);

if (dis > distan) {
    Debug.Log("Getting closer");
    AddReward(0.01f);
}*/

Debug.Log(Vector3.Distance(this.transform.localPosition,
Target.localPosition));

// Distansfunktion 2 (användes aldrig samtidigt som funktion 1)
AddReward(13 - Vector3.Distance(this.transform.localPosition,
Target.localPosition));

AddReward(-1f / MaxStep);
// Reached target
if (isTouchingTarget)
{
    AddReward(1000.0f);
    /*Target.localPosition = new Vector3(UnityEngine.Random.value * 80 - 40,
    0.5f,
    UnityEngine.Random.value * 80 - 40 );*/
}

// Fell off platform
if (this.transform.localPosition.y < -2f)
{
    Debug.Log("hojd fel");
    AddReward(-2.0f);
    EndEpisode();
}

// Colliding with object which is not the target
if (isTouchingSomething)

```

```

{
    Debug.Log("Collision");
    AddReward(-2.0f);
    EndEpisode();
}
if (rBody.velocity.x >= 7 || rBody.velocity.z >= 7 || rBody.velocity.x <= -4 ||
rBody.velocity.z <= -4)
{
    Debug.Log("speeding");
    AddReward(-1f);
}
}
}

```

## 6.2 .yaml parametrar för agent

Vissa parametrar ändras emellanåt, men alla är relevanta.

```

Unset
behaviors:
  CarBehavior:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 2048
      learning_rate: 3.0e-4
      beta: 5.0e-3
      epsilon: 0.2
      lambda: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings: normalize: false

```

```
        hidden_units: 128
        num_layers: 2
    reward_signals:
        extrinsic:
            gamma: 0.99
            strength: 1.0
    max_steps: 500000
    time_horizon: 64
    summary_freq: 10000
```