MI ágensek Unity környezetben

Autóversenyző ágens fejlesztése Unity ML-Agents használatával

Készítette: Nagy Dániel

Konzulens: Dr. Vaitkus Márton

A projekt GitHub repositoryja: https://github.com/NDani23/Dr-Ai-ve

Bevezetés

A mesterséges intelligencia kutatása napjainkban (mint ahogy már párszor a történelem során) ismét a figyelem középpontjában van az informatika világában. A neurális hálókat használó gépi tanulás, vagy mélytanulás egyre több területén bukkan fel az iparnak és a tudománynak.

A mélytanulás egyik szélesebb körben elterjedt formája a megerősítéses tanulás, amely többek között a különböző autonóm rendszerek fejlesztésében tűnik ígéretes megoldásnak. Például az egyre nagyobb hírnévnek örvendő önvezető autók működésében is központi szerepet tölt be ez a technológia. Persze az sem véletlen, hogy bár ezen autók fejlesztése gőzerővel folyik, mégsem botlik beléjük az ember lépten-nyomon az utcán. Hiszen a mélytanulás, és talán különösen a megerősítéses tanulás még bizony eléggé gyerek cipőben jár.

A mesterséges intelligencia láz természetesen elérte a videójáték ipart is. Ez nem meglepő, hiszen ez a technológia kézenfekvő megoldásnak tűnhet a játékfejlesztés egyik talán legösszetetteb problémájára, mégpedig, hogy hogyan lehet élethű viselkedéssel felruházni egy gép által vezérelt karaktert/játékbéli objektumot. Egy algoritmusokkal leprogramozott NPC[1]-n sokszor azonnal feltűnik, hogy csak beégetett utasításokat követ, ami jelentős mértékben tudja rontani a játék által nyújtott élményt és atmoszférát. A túlságosan összetett, bonyolult viselkedést leíró algoritmusok pedig a program teljesítményére vannak negatív hatással, ami egy játék esetében szintén kritikus pont. Vajon a megerősítéses tanulás jelentheti a megoldást erre a problémára? Mennyire szűk még az a gyerekcipő és megéri-e egyáltalán ezzel érdemben foglalkozni? Többek között ezekre a kérdésekre is keresem a választ ebben a dolgozatban.

A projekthez a Unity által fejlesztett ML-Agents nevű keretrendszert használom, amely egy átlátható és viszonylag egyszerű felületet biztosít MI ágensek tanításához és fejlesztéséhez.

[1] non-player character

A projekt célja

Az önálló laboratórium 1 tantárgy keretein belül a célom ezzel a projekttel, hogy megismerjem a megerősítéses tanulás alapkoncepcióit, valamint, hogy fejlesszek és betanítsak egy autóversenyző ágenst.

Az ágens feladata, hogy adott pályaelemekből felépített, tetszőleges versenypályán jó eredményeket tudjon elérni, azaz minél pontosabban, megbízhatóan tudjon olyan köridőket produkálni, amelyek megközelítik, vagy akár túl is szárnyalják egy humán játékos eredményeit. Ezen tárgy keretein belül tehát még nem cél, hogy az ágensek egymás ellen, vagy a játékos ellen tudjanak versenyezni, ez a későbbiek kihívása lesz. Ugyanakkor az ágens képességeinek szemléltetése érdekében készítek egy játékprogramot, ahol a humán játékos adott pályákon összemérheti tudását a gép által vezérelt autókkal. A későbbiekben szeretném megvalósítani, hogy akár a felhasználói felületen keresztül, a játékos által, a megadott elemekből felépített tetszőleges pályán is lehessen versenyezni.

A fent említetteken túl célom még az is, hogy választ kapjak néhány bennem felmerülő, a megerősítés tanulással kapcsolatos kérdésre:

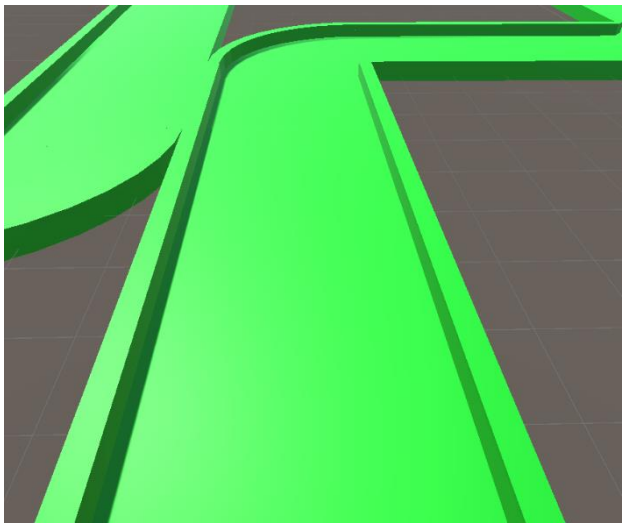
- Előzetes tapasztalat hiányában mekkora kihívás egy videójáték ágens betanítása?
- Milyen eredmények várhatóak egy algoritmusokkal leprogramozott NPC-hez képest?
- Fel tudja-e venni a versenyt egy betanított ágens egy humán játékos ellen?

Kiinduló állapot és az alkalmazás környezet

Ahogy a „projekt célja” részben említettem, az ágensek egy előre megadott pályán kell körbeérnie és jó köridőket futnia. Először is szeretnék arról írni, hogy hogyan épül fel maga a pálya és milyen szempontokat vettem figyelembe a tervezéskor.

A pálya

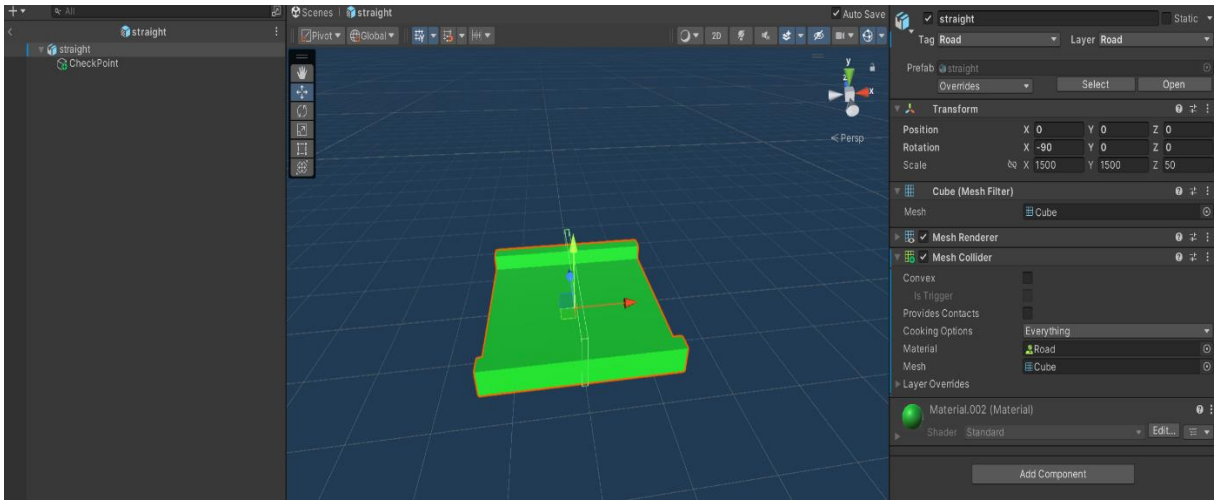
A pályák felépítéséhez jelenleg mindössze két különböző építőkockát modelleztem, amelyeknek tetszőleges kombinációja valid útvonalat eredményez. A versenypályák tehát egyenes szakaszokból és 90-fokos kanyarokból állhatnak az alkalmazás jelen állapotában. Egy lehetséges pályaszakaszt szemléltet az 1. ábra.



1. ábra: Egy pályarészlet

Az útelemek fallal vannak határolva, így természetes módon tudja érzékelni az ágens a pályahatárokat. Egy másik, gyakori megközelítés, hogy több Collider objektummal van körberakva a pálya, amelyek jelezni tudnak az ágensek, ha érintkezik velük. Ez egy kicsivel megengedőbb hozzáállás és sokkal gondosabb átgondolást igényel a jutalmak/büntetések definiálásakor.

A pályaelemekhez készítettem egy-egy prefab objektumot, amelyek a konkrét modellen kívül tartalmaznak minden alkotóelemet, amik szükségesek a helyes működéshez. A 2. ábrán egy ilyen prefab objektum látható.



2. ábra: Útszakasz prefab

Minden útszakaszhoz tartozik egy checkpoint objektum, ami lényegében egy trigger, ami jelzi az ágensek, hogy jó irányba halad. Ezt a későbbiekben részletesebben is tárgyalom. Fontos

beállítás minden útszakasznál a Tag és a Layer mezők, amik segítségével az ágens azonosítani tudja a pályahatárokat. A prefab természetesen tartalmaz még egy Collider objektumot is. Ennél azt emelném ki, hogy ezekhez rendelve van egy külön, „Road” nevű anyagleíró material, aminek a súrlódási tulajdonságai elég nagyok. Azt, hogy erre miért van szükség, a későbbiekben tárgyalom.

A Unity felhasználói felületén keresztül viszonylag egyszerűvé tettem az új pályák építését. Mindössze egy olyan szülőelem alá kell tenni őket, ami rendelkezik a „TrackScript” nevű logikával. Fontos azonban, hogy az útszakasz darabok menetirány szerint sorrendben legyenek. Ekkor a pálya logikáját leíró program összetudja gyűjteni a checkpointokat, és információt tud szolgáltatni az ágens számára arról, hogy a helyes irányba halad-e. Arra is oda kell figyelni, hogy a pályarészek a menetiránynak megfelelő irányba nézzenek, ugyanis a checkpointok előre mutató vektorját felhasználja az ágens a megfigyelések begyűjtése során. A helyes irány indikálására egy saját textúrát fogok használni. Egy így felépített pályakonstrukciót szemléltet a 3. ábra.

3. ábra: Egy pálya felépítése

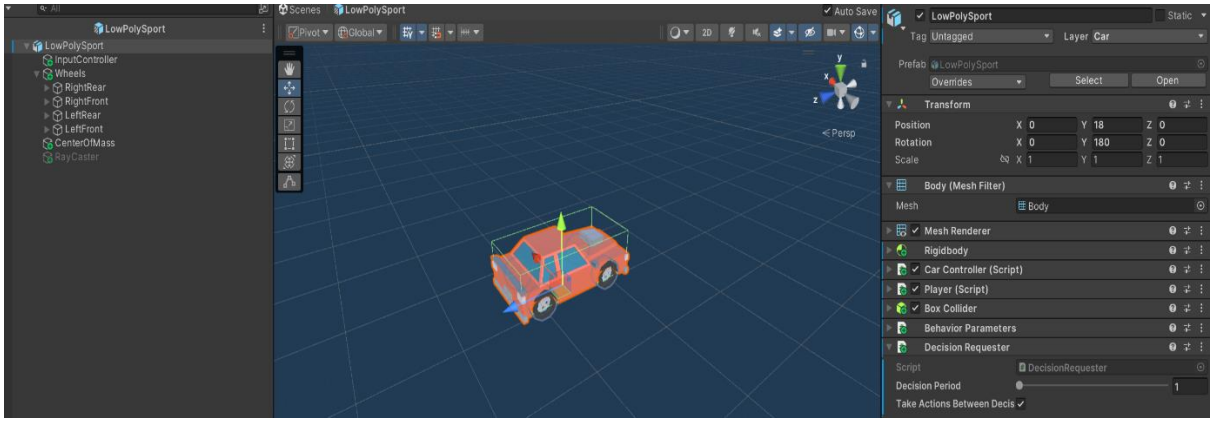
Az autó

Egy ilyen feladat esetében kiemelkedő fontosságú, hogy maga az autó, amit az ágens irányít, hogyan viselkedik. Ez az a szempont, ami a leginkább meghatározza a feladat nehézségét. Ha egy túlságosan idealizált, „arcade” stílusú viselkedéssel ruházzuk fel az autók, akkor egy ágens betanítása nem túl bonyolult feladat, hiszen sokszor egy ilyen játékban még fékeznie sem kell a játékosnak. Azonban a realizmus irányába lépve, akár egy nagyon bonyolult feladatot is kaphatunk (lásd önvezető autók). Ebben a projektben kezdetben megpróbáltam megtalálni az arany középutat e két megközelítés között. Az autó viselkedése azonban inkább még mindig egy arcade játékban megszokotthoz hasonlít. Ez lényegében csak annyit jelent, hogy az autónak túlzottan nagy tapadása van, így például nem tud kiforogni a kanyarokban a túl nagy gázadás vagy túl nagy bemeneti tempó miatt. Ez persze a másik oldalról azt is jelenti, hogy a versenyautó túlzottan alulkormányzott, így az ütközések elkerülése és a jó köridő teljesítése érdekében az ágensek a megfelelő helyeken fékeznie kell, és meg kell tanulnia az adott kanyarkombinációkra optimális versenyíveket. A későbbiekben szeretném a jármű viselkedését a realitás irányába módosítani.

Az autóhoz is készítettem egy prefab objektumot, ami mindent tartalmaz az ágens helyes működéséhez és a jármű irányításához, ez a 4. ábrán látható. Az autó mozgásáért a Unity által implementált WheelCollider objektumok a felelősek. Bár a jármű jelenlegi viselkedését egy egyszerűbb arcade stílusú logikával is könnyedén meg lehetett volna valósítani, azonban a WheelColliderek paramétereinek módosításával a későbbiekben egyszerűen lehet majd realisztikusabbá tenni az autó viselkedését.

Fontos szempontnak tartottam azt is, hogy könnyedén lehessen új, különböző viselkedéssel rendelkező autókat integrálni a játékba. Ezért a Unity felhasználói felületén keresztül

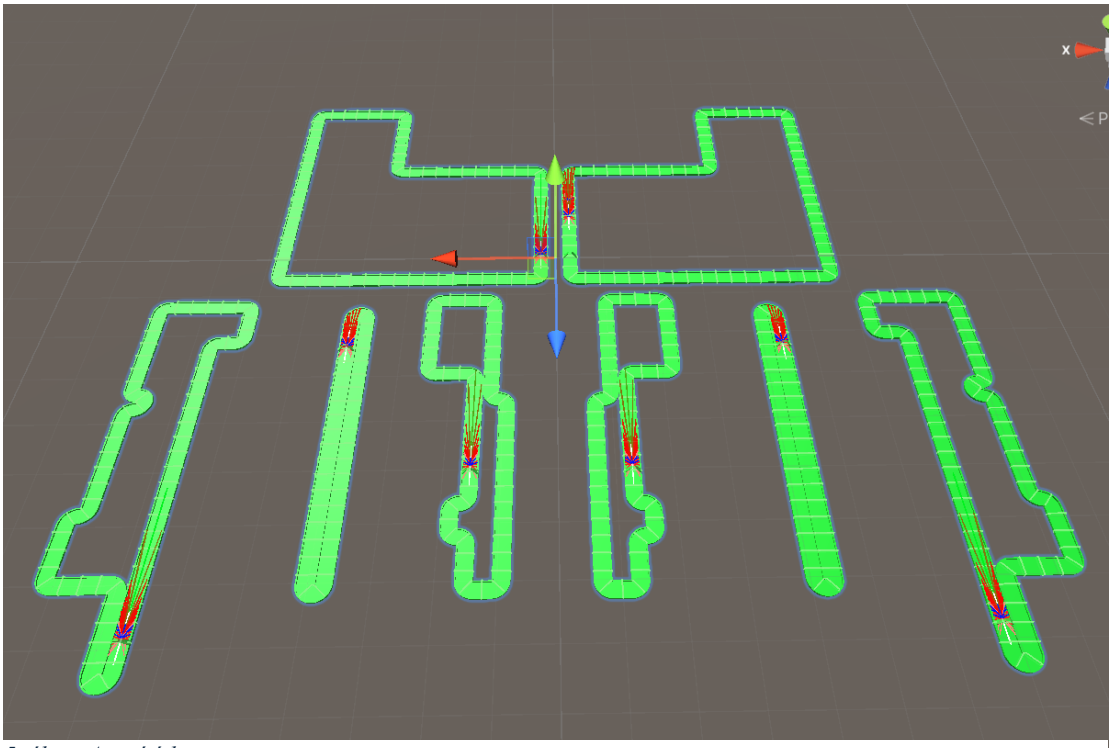
könnyedén lehet állítani többek között az autó súlyát, teljesítményét, maximális kormányszögét, súlypontját. Külön beállítható minden kerékre, hogy az kormányozható-e és kap-e hajtást. Ezekért a beállításokért a „CarController” és a „Wheel” nevű scriptek felelősek.



4. ábra: Az autó prefab

Tanítási környezet

Az ágenst egy időben 8 különböző pályán tanítottam, bár valójában ezek páronként szimmetrikusak. Minden pályán egyszerre 3 autó tanult, hogy minél sokszínűbb tapasztalatokat tudjon gyűjteni az ágens. A 4-4 páronként szimmetrikus pályát megpróbáltam olyan módon felépíteni, hogy mindegyiknek kicsit más karakterisztikája legyen és összeségében lefedjék a lehetséges kanyarkombinációk nagyrészét. A megadott útdarabokból felépíthető akár egy 180 fokos kanyar is, ami talán a legextrémebb kihívást jelentette az ágens számára, így az ilyen kanyaroknak külön készítettem 1-1 pályát. A tanító környezetről készített képernyőkép látható az 5. ábrán.



5. ábra: A tanító környezet

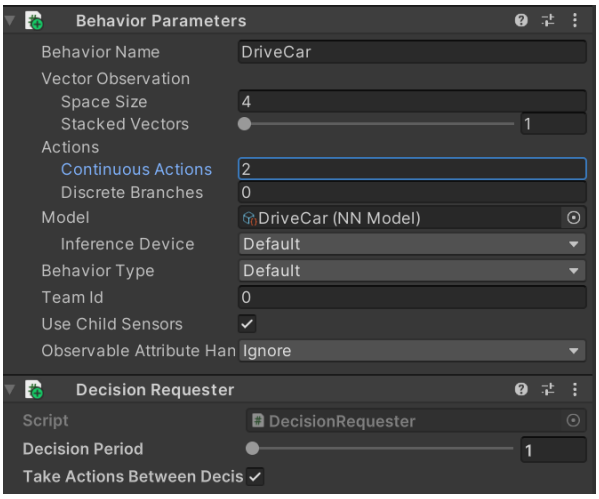
Az ágens működése

Egy megerősítéses tanulást alkalmazó ágens fejlesztésénél 4 főbb szempontot kell nagyon alaposan megtervezni és beállítani a kívánt működés eléréséhez. Az első és talán legegyszerűbb szempont, hogy milyen akciókat tehet az ágens működése során (action space). A második, sokkal kevésbé egyértelmű szempont, hogy hogyan érzékelje a környezetét az ágens, mik a megfigyelések, ami alapján a döntéseket hozza (observations). A harmadik, még a megerősítéses tanuláshoz kapcsolódó döntés, amit meg kell hoznia a fejlesztőnek, hogy hogyan értékelje az ágens döntéseit, milyen jutalmakat és büntetéseket hozzon. Fontos képesség egy ilyen neurális háló tanításánál, hogy a fejlesztő lényegében az ágens fejével tudjon gondolkodni, hiszen az csupán annyit lát a környezetéből, amit mi megadunk neki, és pusztán onnan tudja, hogy az adott helyzetben jól döntött-e, hogy milyen jutalmakat kap. A negyedik szempont természetesen a neurális háló és maga a tanítás paramétereinek beállítása, ami szintén nagy tapasztalatot és hozzáértést igényel.

Akciók és döntési periódus

A Unity ML-Agents keretein belül az ágens által megtehető akciókat két különböző féleképpen tudjuk megadni. Lehetőségünk van diszkrét és folytonos döntési ágak definiálására. Ennél az adott feladatnál kézenfekvő megoldás a folytonos akciókat választani, hiszen mind a kormányszög, mind a gázpedál állása egy folytonos tartományból veheti fel az értékeit a való életben. A projektben az ágens két folytonos ágon kell, hogy döntést hozzon. Az egyik természetesen maga a kormányszög, aminek a felhasználói felületen keresztül tudunk maximális értéket beállítani. Az autó sebességének manipulálására szolgál a másik ág, ami az autós játékokban megszokott módon egyszerre kezeli a gázadást és fékezést is (a negatív „gáz állás” fékezést jelent). Az ágensnek nem kell kuplungot és sebességváltót kezelnie.

Az akcióknál talán az egyetlen kevésbé egyértelmű rész a döntési periódus, amit a Unity ML-Agents keretein belül egy kötelező „Decision Requester” komponensel manipulálható. Mivel ebben az alkalmazásban, és az autóversenyzésben általában is egy jól meghozott döntést sokszor csak milliszekundumok választanak el egy rossz döntéstől, így a döntési periódust 1-re állítottam az alapértelmezett 3 helyett. Ezt tehát azt jelenti, hogy az ágensnek minden egyes lépésben két döntést kell meghozzon, mégpedig, hogy mekkora kormányszöggel és gáz állással menjen tovább. A viselkedési paramétereket és a Decision Requester beállítását a 6. ábra szemlélteti.

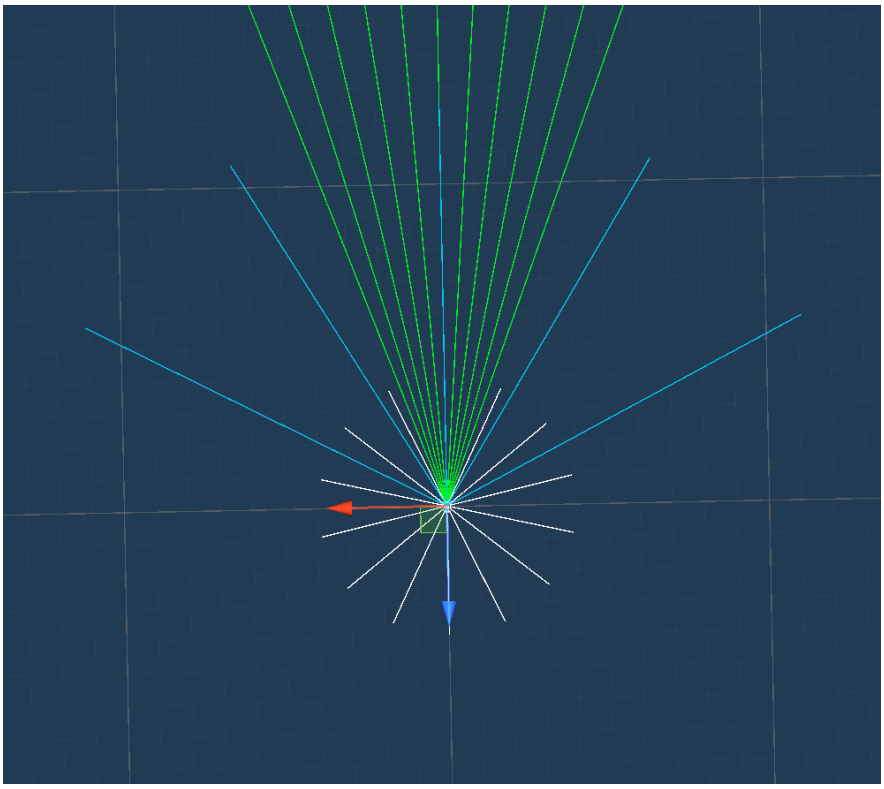


6. ábra: Viselkedési paraméterek

Megfigyelések

Az ágens leginkább az autót körülvevő sugár alapú szenzorok segítségével tud információkat gyűjteni a környezetéről. A Unity ML-Agents implementál egy nagyon jól használható és konfigurálható sugár alapú vizuális megfigyelés komponenst (RayPerceptionSensor3D). Minden ilyen komponensnél beállítható, hogy milyen irányba mennyi sugarat indítson, milyen messzire érjenek el ezek a sugarat és milyen más objektumokkal tudjanak metszést jelezni. Minden sugár 3 potenciális megfigyelést biztosít az ágens számára. Jelzik, hogy eltalált-e valamit az adott sugár, ha igen, akkor milyen irányban és milyen messze van az adott objektum. Ezen felül beállítható, hogy egy ilyen komponens milyen más játékbéli objektumokat tudjon beazonosítani Tagek alapján.

Az alkalmazásban 3 különböző RayPerceptionSensor3D komponens van hozzáadva az autóhoz. Ezek elhelyezkedése a 7. ábrán látható.



7. ábra: Sugarak elhelyezkedése

A legtöbb információt az előremutató, zöld színű sugarak szolgáltatják. Ezek csupán az autó előtt lévő pályát és falakat látják meglehetősen nagy felbontásban. Ezek a sugarak a leghosszabbak, hogy az ágens jó előre lássa például egy kanyar közeledtét, illetve a pálya nyomvonalát. A kék színű sugarak a checkpointok azonosítására szolgálnak, ez a komponens nem lenne feltétlenül szükséges a helyes működéshez, azonban segítségével az ágens tisztább képet kap a következő kanyar irányáról, illetve hasznos információkat szolgáltat abban az esetben, ha az ágens esetleg elakad, és ismét meg kell találnia a helyes irányt. A harmadik, fehér színű sugárvető komponens az autó közvetlen környezetéről szolgáltat információkat 360 fokos szögben. Ez a lassabb, kanyargósabb útszakaszokon lehet az ágens segítségére. Ez a komponens

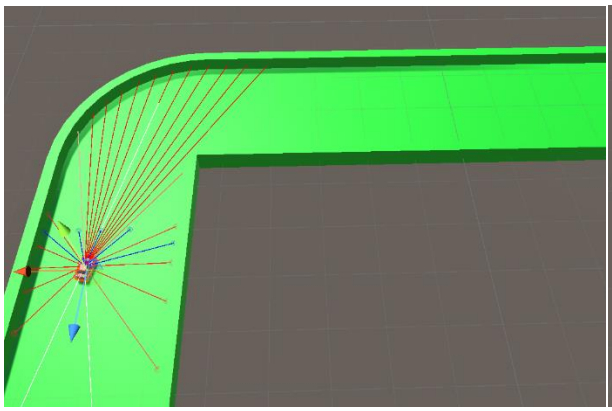
azonban abból a célból került az autóra, hogy a későbbiekben láthassa az ágens a körülötte/mögötte lévő autókat is, anélkül, hogy újra kéne tanítani az egész neurális hálót.

A Unity felhasználói felülete lehetőséget nyújt annak beállítására, hogy az egyes RayPerceptionSensor-ok egyszerre hány lépés megfigyeléseit továbbítsak a neurális háló felé. Ha egy körben több megelőző megfigyelés eredményeit is továbbítjuk, akkor az ágens egy nagyobb felbontású, tisztább képet kaphat a környezetéről. Mivel ennél a feladatnál nagyon fontos, hogy az ágens pontosan lássa a pályahatárokat, ezért a zöld és fehér sugarak esetén „Stackelt" sugarakat használok 3 paraméterrel.

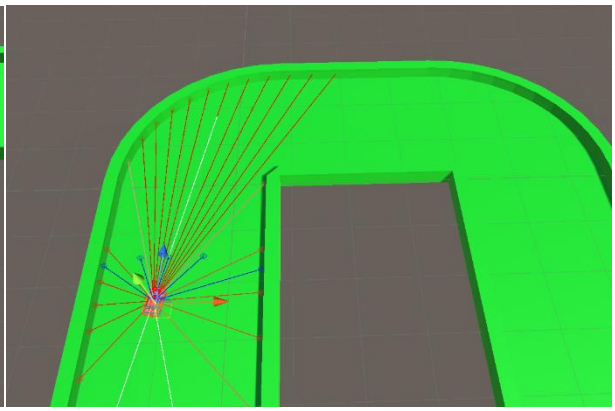
A sugarak segítségével az ágens egy elég tiszta képet kaphat a közvetlen környezetéről, és a pályán való elhelyezkedéséről, azonban a kívánt működés eléréséhez biztosítani kell még pár fontos információt, amik a sugarak alapján nem lennének egyértelműek. Ezért az alkalmazásban a sugarak mellet biztosítok az ágens számára még további 4 darab tradicionális vektor megfigyelést is. Ezek a következők:

- Az autó sebessége (Velocity vektor hossza)
- A következő 5 checkpoint összeadott előre mutató vektorjának hossza
- A következő 5 checkpoint összeadott előre mutató vektorjának és az autó előre mutató vektorjának skalár szorzata
- A következő 5 checkpoint összeadott előre mutató vektorjának és az autó keresztirányú vektorjának skalár szorzata

Felmerülhet a kérdés, hogy miért van szükség a checkpointok előre mutató vektorjaiból származó plusz információra, amikor a sugarakból már kap hasonló adatokat az ágens. Ehhez fontos kiemelni, hogy ez az alkalmazás nem egy utcai vezetés szimulálását végzi, inkább egy versenypályán történő időmérő edzéshez hasonlít. Ez azért lényeges, mert a valóságban egy autóversenyző nem csak azt tudja az adott pályáról, amit éppen lát, hanem pontosan tudja, hogy milyen kanyarok fognak következni. Sok hasonló alkalmazásban elterjedt megoldás, hogy a sugarakon túl még a következő checkpoint például előre mutató vektorját is megadják az ágensnek. Azonban ez nem tartalmaz kellő információt ahhoz, hogy egy adott kanyart ideálisan tudjon bevenni az ágens, ezt a 8. és 9. ábra szemlélteti.



8. ábra: jobb kanyart egyenes követ

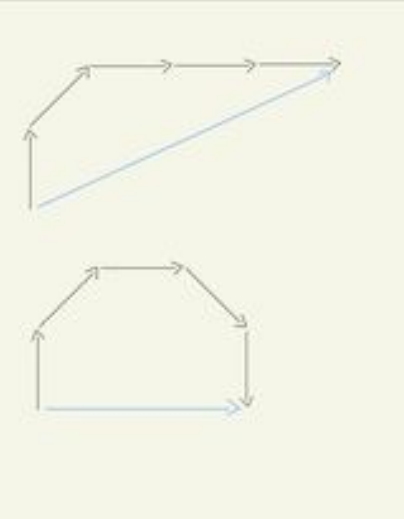


9. ábra: jobb kanyart kanyar követ

A két ábrán jól látszik, hogy csupán a következő checkpoint és a sugarak adatai alapján ez a két kanyar teljesen egyformának tűnik az ágens szemszögéből, mégis lényegesen más megközelítést igényel a kettő, a 9. ábrán látható pozícióban például az ágensnek már régen

fékeznie kéne, hogy el tudjon fordulni a kanyarban. Azzal viszont, hogy a következő 5 checkpointról szolgáltatok egy összegzett vektort, az ágens számára is megkülönböztethetővé válik a két eset. Összefoglalva tehát az összegzett vektor irányából az ágens információt kaphat a következő pályaszakasz általános haladási irányáról, a vektor hossza pedig a kanyar élességét jelezheti az ágens felé. A konkrét/precíz nyomvonalat pedig a sugarak alapján látja. A 10. ábrán látható, hogy a vektor megfigyelések alapján hogyan néz ki az ágens számára a fenti két kanyar.

Itt még egy dolgot szeretnék megmagyarázni, mégpedig, hogy miért az így keletkezett vektorokkal vett skaláris szorzatát adom oda az ágensnek, és miért nem magát a vektort. Ennek az az oka, hogy az ágens szempontjából csak az a fontos, hogy az autó orientációjához képest merre halad tovább az út. Hiszen, ha egy egyenes mentén halad, akkor teljesen mindegy, hogy az az egyenes a játéktérben épp például a +X irányba halad, vagy a -Z irányba. Viszont, ha magukat a vektorokat adnám oda az ágensnek, akkor az előbb említett két esetet bizony különbözőnek látná, és külön meg kéne tanulnia minden irányba egyenesen menni. Persze egy kellően mély neurális háló előbb-utóbb valószínűleg megtanulná az összefüggést, viszont az biztos, hogy emiatt tovább tartana a tanulás.



10. ábra: Vektor megfigyelés

Jutalmak / Büntetések

Az alkalmazásban használt büntetések és jutalmak:

- Megfelelő checkpointon való áthaladás: +2
- Fallal való ütközés: -(ütközési sebesség * 0,05 + │ütközés normálvektorja ∗ autó előre mutató vektorja│ * 5)
- Ütközésben maradás (collision stay): -0.01
- Sebesség < 10: -0.05
- Alap büntetés minden lépésben: -10/Maximális lépésszám

Itt talán a fallal való ütközés és az alap büntetés szorul némi magyarázatra. Amikor ütközés történik, akkor lényegében egy súlyozott összeget állítok elő a becsapódás sebességéből és a becsapódási szögből, amit aztán az ágens megkap büntetés formájában. Tehát minél nagyobb tempóval és minél nagyobb szögben találja el a falat, annál nagyobb büntetést kap. Eleinte egy konstans büntetést adtam ebben az esetben, viszont ez a megközelítés kicsivel több információt hordoz az ágens számára, minthogy ütközött és az rossz. Ez a megoldás jobban rávezeti tanítás közben az ágenst, hogy mit kéne máshogy csinálnia, például, hogy egy adott kanyarba kevesebb tempót kéne bevinnie.

Ha azt szeretnénk, hogy az ágens minél gyorsabban tegye a dolgát, akkor egy bevett módszer, hogy minden egyes lépésben adunk neki egy kis büntetést, így ösztökélve azt a haladásra. Mivel ebben az alkalmazásban a gyors cselekvés áll a középpontban, így természetes választás volt, hogy én is alkalmazni fogom ezt a megoldást. A büntetést még elszokás osztani az adott epizód maximális lépésszámával. Ez csak azt a célt szolgálja, hogy a Max Step paraméter változtatása ne legyen kihatással az epizód végére a felhalmozott büntetés mennyiségére.

Itt is kérdés lehet, hogy akkor miért van szükség büntetésre még akkor is, ha az autó sebessége kisebb mint 10? Ezzel azt a problémát szerettem volna megoldani, hogy amikor egy ágens falnak ütközött a tanítás során, egyszerűen megállt egyhelyben, és nem indult tovább még akkor sem, ha egyébként már visszagurult az ideális ívre. Ezzel a büntetéssel azonban sikerült valamelyest orvosolnom ezt a viselkedést.

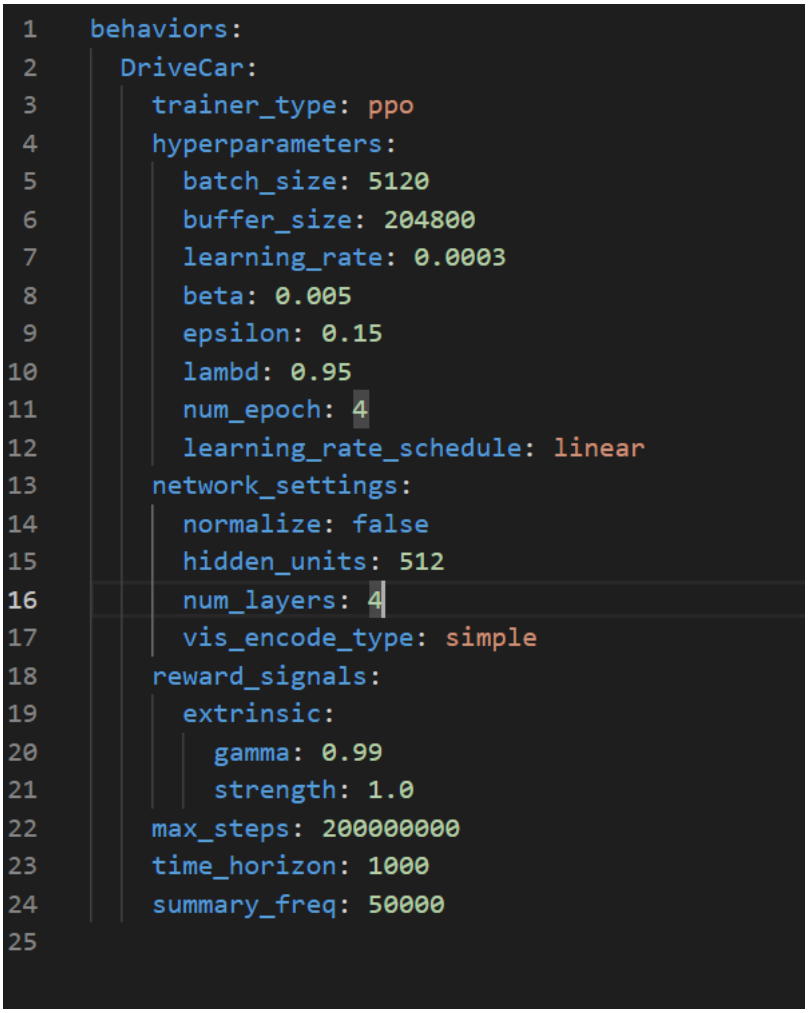
A jutalmak kiosztásánál talán az volt a legnagyobb kihívás, hogy megtaláljam az egyensúlyt a gyors haladás és a stabil, megbízható vezetés között. Mivel az alkalmazásban az autó nem tud megsérülni, így eleinte előfordult, hogy az ágens bevállalta a fallal való ütközésért járó büntetést, mert így nagyobb tempót tudott átvinni egyes kanyarokon, és így összességében jobb pontszámmal zárta az epizódot. Az ellenkező eset az volt, amikor túl nagy büntetést kaptak az ágensek ütközés esetén, és emiatt sokkal óvatosabban vezettek, nem mertek annyira közel kerülni a falhoz és ezzel persze rengeteg időt veszítettek. Így egy köztes megoldást kellet kitalálnom. Kicsit kisebbre vettem az ütközésért járó büntetést, viszont átalakítottam a környezetet olyan módon, hogy biztosan ne nyerjen időd az ágens azzal, hogy esetleg súrolja a falat. Itt utalnék vissza a pálya leírása részben említett anyagtulajdonság leíró material használatára. Ahogy már említettem, ebben az anyagleíróban nagyon nagyra vettem a falak súrlódását, ami azt eredményezte, hogy még kis ütközés esetén is nagyon visszalassul az autó. Érdekes volt látni, hogy ezzel a megoldással az ágensek mennyire közel mernek menni a falakhoz anélkül, hogy hozzáérnének, rengeteg időt nyerve ezzel.

Látható, hogy a jutalmazás logikája elég egyszerű. Még számos esetben lehetne jutalmat vagy éppen büntetést adni az ágensnek (például rossz checkpointon való áthaladás), viszont a számtalan kísérletből azt a következtetést vontam le, hogy jobb eredményeket produkál egy ágens, ha a jutalmak egyszerűek, és nagyobb szabadságot engedünk neki az optimális stratégia megtalálására. Azt is megfigyeltem, hogy az ágensek mintha nagyobb mértékben reagálnának a pozitív jutalmakra, mint a negatív büntetésekre. Ezek alapján megéri inkább a jó döntéseket jutalmazni, mint a rosszakat büntetni.

Konfiguráció

A tanításhoz a Unity ML-Agents-ben alapértelmezetten beállított, a **Proximal policy optimization** (ezután **PPO**) algoritmust használtam. A PPO egy általánosan viszonylag jó eredményeket produkáló megerősítéses tanulás algoritmus. A másik elterjedt algoritmus a **Soft Actor Critic (SAC),** ami egy valamivel stabilabb, kevésbe bizonytalan eredményt nyújtó, viszont jóval költségesebb algoritmus. A SAC ezen felül sokkal érzékenyebben reagál az esetleg rosszul beállított konfigurációkra, ezért is választottam kezdésnek a PPO-t amit ilyen szempontból nehezebb „elrontani” és így ideálisabb döntés egy kevés tapasztalattal rendelkező fejlesztőnek. Bár ahogy említettem, a PPO kevésbé érzékeny a konfigurációs apróságokra, mégis jó pár dolgot kellet jelentősen megváltoztatnom az alapértelmezett beállításokhoz képest. A különböző paraméterekről és alapértelmezett értékeiről a következő linken található bővebb információ:https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md

A projekt végső konfigurációs fájlja a 11. ábrán látható. Fontos, hogy ebben a fájlban csak a projekt szempontjából fontosabb, és az alapértelmezettől eltérő konfigurációs beállítások szerepelnek.



11. ábra: Konfiguráció

Az első dolog, amit szeretnék kiemelni, az magának a tanított neurális hálónak a mérete. Összehasonlításképpen, alapértelmezetten a Unity ML-Agents egy 2 rétegű, rétegenként128 neuront tartalmazó hálót használ (num_layers és hidden units). Ebből látszik, hogy az általam használt háló eléggé nagy. Ez elég érdekesnek tűnhet, hiszen „ránézésre" maga a feladat elsőre nem tűnik túlságosan bonyolultnak. Mégis, az elvégzett kísérletekből kiindulva azt állapítottam meg, hogy ennél kisebb háló esetén sokkal nehezebben, vagy egyáltalán nem tudja az ágens megoldani a feladatot. Ennél nagyobb hálók viszont már nem rövidítik tovább a tanulási időt.

Az is látszik, hogy a max_steps értéke szintén hatalmas. Egy átlagos nehézségű feladatra való betanításához általában kb. 10-50 millió lépésre van szükség. Ehhez hozzátartozik, hogy a végleges, betanított ágens kb. 90 millió lépés után érte el az optimálishoz már eléggé közeli teljesítményt. A lassú tanulás részben tudatos döntés eredménye. Mivel a feladat tökéletes elvégzéséhez nagyon precíznek kell lennie az ágensnek (például minél közelebb engedi kanyarokban a falhoz az autót, annál jobb lesz a körid), így a tanítási konfigurációs beállításokat sokkal inkább a stabilitás irányába mozdítottam el. A túl gyorsan változó viselkedés ugyanis

hatalmas visszaesést jelenthet, gondoljunk csak arra, hogy egy kicsivel korábban történő kanyarodás (az időnyerés érdekében) következtében egyből a falban köthet ki az autó. A stabilabb tanítás érdekében a következő konfigurációs beállításokat módosítottam:

- batch és buffer méret jelentős mértékű növelése
- epsilon csökkentése

A time_horizon egy másik nagyon fontos paraméter, amiről véleményem szerint kevés szó esik, ezért a Unity ML-Agents dokumentációjából kiemelném az ehhez kapcsolódó magyarázat talán leglényegibb részét: „*This number should be large enough to capture all the important behavior within a sequence of an agent's actions.”*

Ha jól értelmezen, ez lényegében azt jelenti, hogy egy adott lépésben hány megelőző lépést képes figyelembe venni az ágens. Az autóversenyzésben egy adott döntés meghozatala nagyban befolyásolja, hogy később milyen pozícióba kerül a versenyző. Gondolhatunk például arra, hogy egy kanyar közeledtével már jó előre ki kell húzódni a pálya külső ívére annak érdekében, hogy optimálisan bevehető legyen az adott kanyar. Így ezt a paramétert is jelentősen megnöveltem az alapértelmezetthez képest. Túl kicsi time_horizon esetén érdemes belegondolni abba a helyzetbe, amikor egy sikeres kanyarbevétel után az ágens már csak arra tud „visszaemlékezni”, hogy közvetlen a kanyar előtti pillanatban milyen helyzetben volt. Ez az én meglátásom szerint azért lehet probléma, mert így az ágens csak úgy fogja tudni bevenni a kanyarokat, ha már valahogy a kanyar előtt az ideális ívre keveredett, viszont arra már nem fog emlékezni, hogy hogyan került oda. Ezáltal nem tudja megtanulni az ideális íveket, amivel rengeteg időt veszíthet.

Demonstráló applikáció

A betanított ágens képességeinek demonstrálása érdekében készítettem egy egyszerű játék alkalmazást, aminek a keretein belül a felhasználó összemérheti tudását a géppel. A játékos előre összeállított pályákon versenyezhet az ágenssel. Ahogy már említettem, a későbbiekben célom az is, hogy a felhasználó maga tudjon összeállítani egy pályát a felhasználói felületen keresztül. Az applikáció egy béta verziójáról láthatunk egy képernyőképet a 12. ábrán.

12. ábra: Applikáció

A képen látott időeredményt nagyjából 10. próbálkozásra sikerült elérnem, és még így is több, mint fél másodperces hátrányban voltam az ágenshez képest 1 kör után egy nagyon rövid pályán. Az applikációt tesztelve megállapítható, hogy az ágens képes felvenni a versenyt egy humán játékossal, sőt megbízhatóbb és jobb eredményeket is tud nyújtani általában. Természetesen biztosan lehetne olyan nyakatekert kanyarkombinációkat összeállítani, ami még kifogna az ágensen, de véleményem szerint egy ilyen elemekből felépített klasszikus pályán mindig meg tudja állni a helyét.

Összefoglalás és fejlesztési lehetőségek

Összességében nagyon élveztem a projekttel járó munkát, tanulást és kísérletezést. Azt gondolom sikerült elérnem a kezdeti kitűzött céljaimat. Viszont azt is fontos megjegyezni, hogy a kísérletezések kezdetén sokkal kevésbé jöttek az eredmények, mint ahogyan azt vártam. A kezdetben kigondolt koncepcióim szinte mindegyike jelentős mértékben megváltozott mostanra, ahogy a fejlesztés során begyűjtöttem a tapasztalatokat. Azt gondolom ebben a témában nagyon könnyű a kezdetekben elbíznia magát az embernek, hiszen a megerősítéses tanulás alapkoncepciója nagyvonalakban meglehetősen könnyen érthetőnek tűnhet. Csak akkor jöttem rá, hogy mennyi minden apróság áll a háttérben, és milyen apró dolgokon bukhat meg egy-egy tanítás, amikor már napok óta nemsikerült elérnem, hogy legalább egy egyenes mentén végig tudjanak menni az ágensek. Azt gondolom tehát, hogy ha tapasztalat nélkül állunk neki ennek a feladatnak, akkor a fejlesztési időből valószínűleg nem fogunk tudni sokat megspórolni. Úgy érzem azonban, hogy a félév végére sikerült „ráéreznem az ízére” a megerősítéses tanulásnak, és a begyűjtött tapasztalatok hatalmas segítséget fognak nyújtani a későbbiekben.

Természetesen az alkalmazás jelen állapotában csak az első felvonásában van. Még rengeteg téren szeretném továbbfejleszteni ezt a projektet. Ezeket a lehetőségeket sorolom fel a teljesség igénye nélkül:

- Többféle útszakasz darab beépítése
- Egymás/játékos ellen versenyző ágensek
- Tetszőleges pálya építésének a lehetősége a felhasználói felületen keresztül
- Sokkal realisztikusabb autó viselkedés
- Több különböző, egyedi viselkedésű jármű