Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Szilágyi Krisztián Gergely

AUTONÓM JÁRMŰ FEJLESZTÉSE

Önvezető szimulátor fejlesztése megerősítéses tanulással

KONZULENS

Szemenyei Márton

BUDAPEST, 2020

Tartalomjegyzék

Összefoglaló **3**
1 Irodalomkutatás **4**
1.1 Reinforcment Learning 4
1.2 PyBullet 8
2 Felhasznált technológia **10**
2.1 Colaboratory 10
2.2 PyTorch 10
3 Architektúra **12**
3.1 Felépítése 12
3.1.1 Multi-head Attention 12
3.1.2 A2C 15
3.1.3 RAdam 15
4 Szimulációs környezet **18**
4.1 PyBullet részletesebb bemutatása 18
4.2 Objektumok beolvasása 21
4.3 Jutalom függvény 23
4.3.1 Alfa 23
4.3.2 Béta 23
4.3.3 Gamma 25
4.3.4 Delta 25
4.3.5 Epszilon 26
5 Tanítás, eredmények **27**
6 Összefoglaló **30**
7 Irodalomjegyzék **31**

Összefoglaló

A feladatom egy az Irányítástechnika és Informatika Tanszéken megtalálható HPI Trophy Flux Buggy távirányítós versenyautó autonóm járművé alakítása az Önálló laboratórium 2 tárgy keretein belül. A projekt megvalósításán egy több fős csapat dolgozik. A hardvert fejlesztők feladata a szenzorok és a feldolgozó egység kiválasztása, integrálása, míg az én feladatom megalkotni azt a szoftvert, mely autonóm járművet varázsol a távirányítós autóból. Ehhez kell implementálni olyan funkciókat, mint például a sávkövetés/tartás, álló és mozgó akadályok detektálása, kikerülése, sőt akár reagálás a jelzőtáblákra, lámpákra.

A projekt egy kétéves cél, így tagoltam a cél eléréséhez vezető utat négy fázisra, négy félév szerint: lehetséges technológiák megismerése és kezdetleges architektúra megtervezése, tanulói környezet kialakítása, környezetben tanítás, tesztelés és architektúra finomítása, legvégül pedig integráció a célhardware-re és valós környezet béli teszt és finomítások. Ebben a beszámolóban az első kettő részfeladatról lesz szó.

Lényegében a feladat egy megerősítéses tanulással betanított neurális hálózat [1] alapú software létrehozása. A rendszer bemenete az autóra szerelt kamerából szerzett információk, a döntéshozatal utáni kimenete pedig az autó irányításához szükséges jel magasszintű reprezentációja. Tehát például mekkora szögben forduljon el, mekkora sebességgel menjen a kocsi, nem pedig, az, hogy a motornak mekkora kitöltési tényezőjű PWM jelet küldjön.

1 Irodalomkutatás

A projekt megvalósítása egy hosszútávú cél, az előző félévben a projekt szkópja főleg az irodalomkutatás és a tervezés volt, nem a megvalósítás és tesztelés. Megismerkedtem a megerősítéses tanulásban használatos fogalmakkal, algoritmusokkal és ezek alapján a célom volt összerakni egy kezdeti architektúrát, melyet a későbbiekben továbbfejlesztve elkészülhet egy teljesen autonóm módon működő jármű szoftvere. Ahhoz, hogy ezt a kezdeti architektúrát képesek legyünk eredményesen tanítani és tesztelni egy jól működő szimulációs környezetben kéne elhelyezni. A második félévben ez volt az elsődleges cél, hogy a számunkra fontos tulajdonságokkal rendelkező környezet létrehozzam és elkezdhessem tanítani az ágenst.

1.1 Reinforcment Learning

A gépi tanulás módszereit többféleképpen lehet csoportosítani, tanulási eljárás alapján három felé szokták osztani: van felügyelt (supervised), felügyelet nélküli (unsupervised) és megerősítéses (reinforcement) tanítás. Ezek más és más típusú problémákhoz nyújtanak hatékony segítséget. Osztályozáshoz, azaz adatok csoportosításához, valamint regresszióhoz felügyelt tanítást érdemes használni. A felügyelt jelző itt azt jelenti, hogy miután a gép kiszámolt egy csoportosítást, mi megmondjuk neki, hogy mi lenne a helyes eredmény, amiből tud tanulni. Tehát az adatok címkézettek, a gép adat-címke párokat kap tanuláskor, ellenben a felügyelet nélküli tanításnál. Ezt a módszert főleg klaszterezéshez [2], struktúraminták felismeréséhez használják. Az utolsó említett eljárás, a megerősítéses tanulás esetében a rendszer egy dinamikus környezettől kap valamilyen visszacsatolást a meghozott döntései után. Többnyire jutalom- vagy büntetőpontokat kap, miközben próbálja elérni a célját, például, hogy minél messzebbre jusson egy autóval a versenypályán. Ezt a koncepciót főleg játékok MI-jének fejlesztéséhez, robotok, autók navigációjához használják. Számunkra most ez lesz a fontos eljárás, ezt fogom a továbbiakban részletezni.

Mint említettem a megerősítés tanulás során egy ágens interaktál egy környezettel, amelyben különböző akciókat hajt végre, a környezet adott időpontbeli állapotától függően és ezért valamekkora jutalmat kap. Minél jobban megközelítette a célfeladatot az ágens, annál többet. Minden akció után a környezet egy új állapotba kerül. Egy epizódnak nevezünk egy olyan ciklust, melynek a végén a környezet visszaáll a

kezdő állapotára és kezdődik elölről a folyamat, akárcsak a felügyelt tanulásnál 1 epoch alatt végig megyünk az összes adaton.

Egy hasznos eljárást érdemes megemlíteni, mielőtt rátérnék az algoritmusokra: ez pedig az epsilon greedy stratégia [3]. Megerősítéses tanulás esetén kérdés, hogy milyen taktikát válasszon az ágens, inkább felfedezze a környezetet, vagy inkább a már felfedezett trajektóriát folytatva kiaknázza a lehetőségeket. Az epszilon változó alapján eldöntjük minden epizód elején, hogy felfedezünk (exploration) vagy a felfedezettet jobban kiaknázzuk (exploitation), ezzel nagyobb lehetséges jutalmat elérve. A probléma az, hogy ez a jutalom nem biztos, hogy a lehető legnagyobb (lokális maximumot találunk meg). Az epszilon az epizódok során folyamatosan csökken, mely a kiaknázás valószínűségét reprezentálja, tehát idővel egyre valószínűbb, hogy az ágens felfedez. A felfedezés azt jelenti, hogy nem a legvalószínűbb akciót választjuk, hanem véletlenszerűen mintavételezünk az akciók közül. Célja, hogy olyan állapotba is eljussunk, melyben még nem voltunk.

Az ágens legfőbb tulajdonsága a stratégia (policy), mely egy olyan függvény, ami minden állapothoz hozzárendel egy akciót. A sztochasztikus stratégiát, mely az állapotokhoz egy valószínűségi eloszlást rendel π-vel jelölünk, míg a determinisztikus stratégiát μ-vel szokás. A megerősítéses tanulás célja, hogy megtaláljuk az optimális stratégiát, vagyis azt a stratégiát, ami maximalizálja a teljes jutalom várható értékét.

Az π sztochasztikus stratégia szerinti érték (állapot-érték) függvény megmutatja, hogyha ezt a stratégiát követjük, mennyi az s állapot értéke, azaz mennyi a jövőbeli diszkontált jutalom várható értéke (1.2 egyenlet). A jövőbeli diszkontált jutalom (1.1 egyenlet) a jövőbeli jutalmak összege, exponenciálisan súlyozva a diszkont rátával ($0 < \gamma \leq 1$). Az állapot-érték függvényhez hasonlóan definiálhatunk akció-érték függvényt, mely egy állapot-akció párhoz rendeli a jövőbeli diszkontált jutalom várható értékét, adott π stratégia mellett:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t+1} \tag{1.1}$$

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[G_t | s_t = s] \tag{1.2}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{a\sim\pi}[G_t | s_t = s,\ a_t = a] \tag{1.3}$$

Q-tanulásnak [4] nevezzük azt az iterációs algoritmust, mely egy véletlenszerűen inicializált Q függvényből előállítja az optimális Q függvényt, azaz az összes stratégia közül a maximális akció-érték függvény. Ebből pedig már meghatározható az optimális stratégia, hiszen az optimális stratégiánk az optimális Q függvény szerinti legjobb akció meglépése.

A Q függvény megtanulása viszont rendkívül nehéz feladat lehet a nagy számú lehetséges állapottal rendelkező környezetek esetében, miközben a stratégia egy egyszerű függvény. Emiatt célszerűbbnek tűnik, ha közvetlenül a stratégiát próbálnánk meg megtanulni, az akció-érték függvény helyett. Ez a céljuk az ún. stratégia gradiens (policy gradient) módszereknek. A legegyszerűbb ilyen a REINFORCE algoritmus [5], más néven a Monte-Carlo policy gradient. A stratégia gradiens, azaz a költségfüggvény háló paraméterei szerinti deriváltja egy hosszadalmas levezetés után a következőképpen néz ki:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(s_t, a_t)\, G_t \tag{1.5}$$

A háló paramétereit Monte-Carlo módszerrel frissítjük, azaz, véletlen mintavételezéssel a várható értéket az átlaggal közelítjük. Egyszerűbben: Az algoritmus lényege, hogy ha egy akcióra nagy jutalmat kapott, akkor megerősítjük a döntésében, tehát úgy módosítjuk a háló paramétereit, hogy legközelebb nagyobb valószínűséggel hajtsa végre ezt az akciót. Ellenkező esetben ellenezzük a döntését, így csökkentjük az adott akció valószínűségét.

Ekkor jön elő az a probléma, hogy miként állítsuk be a jutalmazás mértékét. A legtöbb esetben nemnegatív értékek a jutalmak, így a hálót tulajdonképpen nem is büntetjük egy rossz döntésnél, inkább csak kevésbé erősítjük meg a döntésében. Ezért célszerű lenne kiszámolni egy alap értéket, melyet kivonunk az aktuális jutalomból. Ez a baseline fogalma, mint egy offset, eltoljuk a nulla átmenetet. Tehát ennél az alapértéknél jobb teljesítményt jutalmazzuk, a rosszabbat büntetjük. Módosul a stratégia gradiens a következő alakra:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(s_t, a_t)\,(G_t - b(s_t)) \tag{1.6}$$

Ennek a baseline-nak a meghatározására léteznek különböző, jól bevált módszerek. Például ahelyett, hogy konstansnak választanánk, válasszuk meg úgy, hogy akkor jó a jutalom, ha az nagyobb, mint az adott állapotból elérhető jutalom várható értéke, azaz az érték függvénynél. Ezt a különbséget hívjuk előny függvénynek (*advantage*), mely tulajdonképpen a Q és V függvények különbsége:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \tag{1.7}$$

A következő említendő stratégia gradiens módszer: az ún. Actor-Critic [6]. Az ilyen funkciót ellátó hálóknak két „fejük" van, azaz mondhatni két hálóból állnak. Van egy Actor fej, mely θ paraméterekkel rendelkezik és REINFORCE módszerrel tanulja az optimális stratégiát abba az irányba, amibe a Critic fej javasolja. A Critic fej viszont Q-tanulás segítségével az előny függvényt akarja előállítani *w* paraméterekkel. Pontosabban előtte algoritmustól függően az állapot-érték függvényt (V) vagy az akció-érték függvényt állítja elő. Az utóbbit szokták Q Actor-Critic-nek nevezni.

További két fontos változata létezik ennek a módszernek: az A2C (Advantage Actor-Critic [7]) és az A3C (Asynchronous A2C [8]). Ezeknél a Critic fej az állapot-érték függvényt állítja elő. Eddig nem említettem a Bellman-egyenletet, melyre alapszik a Q-tanulás, de tovább nem lehet megkerülni, mert egy egyszerűsítésre fel kell használnunk. A Bellman-egyenlet azt fejezi ki, hogy egy adott állapot-akció párból a lehető legnagyobb jutalom megegyezik a közvetlenül kapott jutalom és a következő állapotból elérhető legnagyobb jutalom összegével, az alábbi formulában felírható:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V^\pi(s_{t+1})] \tag{1.8}$$

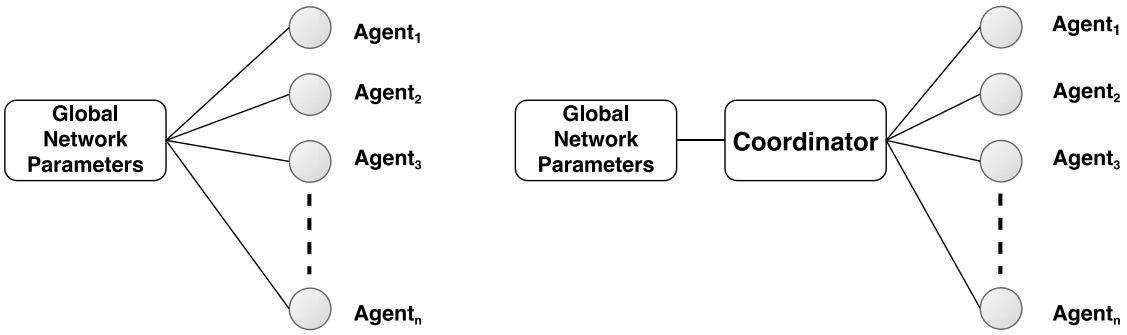
A jutalom 1-től indexelőik, tehát a nulladik akcióra a jutalom r_1. A Bellman-egyenletet felhasználva tudunk módosítani az előny függvény felírásán:

$$A^\pi(s_t, a_t) = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{1.9}$$

Valamint felírhatjuk az új stratégia gradienst, melyet az A2C és A3C esetén számolunk:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(s_t, a_t)\, A^\pi(s_t, a_t) \tag{1.10}$$

E két algoritmus lényege, hogy tanítás alatt több ágens hajt végre akciókat több párhuzamosan futó környezetben, függetlenül egymástól (lásd **1.1. ábra**). Egyik előnyük, hogy így könnyebb felfedezni a környezetet, így nincs szükség az epsilon greedy stratégiára. Az A3C nagy hátránya, hogy a globális háló paramétereket aszinkron módon használják, így előfordulhat az az inkonzisztenciát okozó eset, hogy az ágensek különböző stratégia verziót használnak éppen, ezért a paraméter frissítés nem lesz optimális. Ennek kiküszöbölésére az A2C bevezet egy koordinátort, mely szinkronizálja a szálakat. Megvárja, míg minden párhuzamosan futó ágens befejezte a feladatát (véget ért az epizódjuk, mivel vagy sikeres lett feladat, vagy mert például lejárt az idő). Csak ezután történik meg a frissítés, ezzel elérve a célt, hogy minden epizódot mindegyik ágens ugyanazzal a stratégia verzióval kezdi. Mérések alapján az A2C gyorsabb konvergenciához vezet.



1.1. ábra Bal oldalt az A3C, jobb oldalt az A2C működése látható

1.2 PyBullet

A PyBullet [9] egy ingyenes elérhető fizikai motor, melyet különböző szimulációs környezetekben végzett akciók és állapotok számolására használhatunk fel. Többnyire megerősítéses tanulásnál használják az effajta környezeteket, ebben szimulálják az ágens akciót, és az ebben szimulált állapotokra reagál az ágens.

A rendelkezésünkre áll néhány, a fejlesztők által elkészített környezet, melyeken viszonylag könnyedén tudunk változtatni, a saját feladatunkra szabni. A környezetek támogatnak folytonos és diszkrét akciókat. További előnyei még, hogy egy részletes

útmutató érhető el hozzá, valamint egyre többen használják, így folyamatosan fejlesztik is. Egy komolyabb hátránya van: Sok funkció nincs még implementálva, így jónéhány függvénnyel találkoztam, mely még nincs implementálva, ezért egy-két említett funkció még hibát dob, mert igazából nem létezik.

2 Felhasznált technológia

2.1 Colaboratory

A Colaboratory (a továbbiakban Colab) a Google ingyenes Jupyter jegyzetkezelő környezete [10] [11]. Egyszerűen használható Python kódok futtatására. A felhő alapú szolgáltatás mögött egy Linux rendszer áll, amelyre tölthetünk fel-le adatokat, futtathatjuk a kódunkat, akár GPU-n, sőt TPU-n (Tensor Processing Unit) is.

Legnagyobb előnye a Colab-nak, hogy a hardaware erőforrásai nagyságrendekkel erősebbek, nagyobb számítási kapacitással rendelkeznek, mint egy átlagos otthoni PC vagy laptop konfigurációja. Míg lokálisan egy NVIDIA GeForce GTX 1060 állna rendelkezésünkre, addig a Colab-nál ingyenes elérhető az NVIDIA Tesla K80 videókártyákat tartalmazó gépei (fizetős verziónál akár T4-et és P100-at is használhatnánk). Ez a GPU sokkal nagyobb teljesítményű és több memóriával rendelkezik, így ideálisabb tanításnál. Ez által a kódok CUDA futtatása is nagyságrendekkel gyorsabb Colab-ban [12].

Későbbiekben látni fogjuk, hogy egyelőre miért kell mellőznünk a használatát az első fázisban. Viszont a projekt későbbi fázisaiban már nem kell megkerülnünk, például a végleges tanításban szükség lesz rá, így valószínűleg sor kerül a használatára. Egyelőre viszont a JetBrains Python fejlesztőikörnyezetét, a PyCharm IDE-t használom [13].

2.2 PyTorch

A PyTorch egy Python [14] alapú nyílt-forráskódú tudományos könyvtár gépi tanulásos számításokhoz [15]. Vannak más hasonló könyvtárak, mint például a Keras vagy a TensorFlow, mindegyik másban jobb vagy rosszabb a másiknál. A Keras leginkább kezdőknek hasznos, mivel egyszerűen tanulható, cserébe nagyon korlátozott (specifikusan csak neurális hálózatok fejlesztésére találták ki) és lassú. Ezzel ellentétben a PyTorch alacsonyabb szintű, ezért nehezebb is kezelni, de sokkal gyorsabb, főleg nagy adathalmazokra, és több mindenre lehet felhasználni. A TensorFlow a PyTorch-nál is alacsonyabb szintű, így még nehezebb ügyesen kezelni.

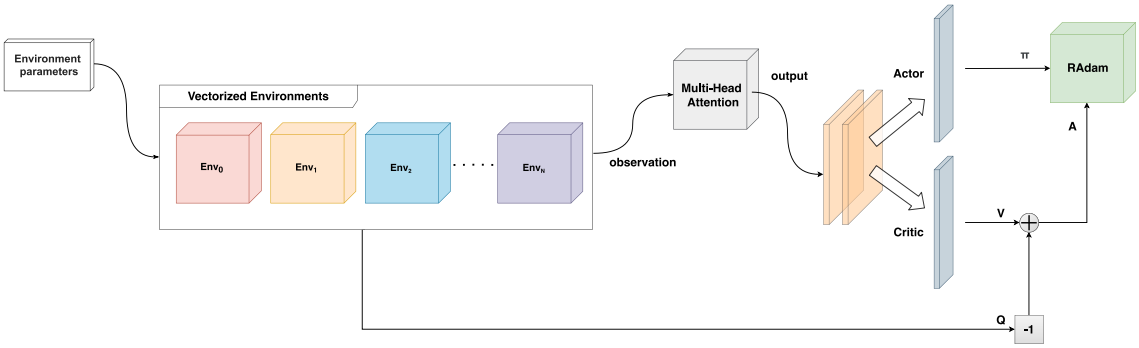
Én a PyTorch mellettem döntöttem, mivel a feladathoz hozzá tartozik, hogy bele kell tudni avatkozni a háló működésébe alacsony szinten is már. Ezenkívül rengeteg hasznos dokumentum található meg hozzá, leírások, példamunkák stb. Az sem elhanyagolandó szempont, hogy az algoritmusok, különböző komponensek melyeket felhasználunk a projektben többnyire szintén PyTorch felhasználásával készültek. Egy ilyen könyvtár leghasznosabb tulajdonsága, hogy könnyeddé teszi a többdimenziós tömbökön, vagyis a tenzorokon végzett műveletek számítását GPU segítségével, továbbá rengeteg olyan függvény és osztály van implementálva, melyeket fel szoktak használni gépi tanulás alkalmazások fejlesztésénél. A számunkra legfontosabb könyvtára a *torch.nn*, ebben speciálisan neurális hálók fejlesztését megkönnyítő osztályok és függvények állnak a rendelkezésünkre. Hatalmas mértékben gyorsítja a háló fejlesztését, ráadásul átláthatóbb és hordozhatóbb kódunk lesz, ha ezeket az alapfüggvényeket és osztályokat alkalmazzuk. Ebben találhatóak meg a konvolúciós, lineáris és más egyéb, például visszacsatolt rétegeket megvalósító osztályok, aktivációs függvények, költségfüggvények. Valamint a későbbiekben részletesen kifejtett Multi-head Attention függvény is.

3 Architektúra

A fejezet célja bemutatni a teljes architektúrát, végig vezetni az olvasót a főbb építőelemein, bemutatni a jelenleg felhasznált technológiákat.

3.1 Felépítése

Az architektúra több kisebb logikai komponensre bontható, ezeket fogom most részletesebben tárgyalni. A rendszer magja az A2C metódust megvalósító kétfejű neurális hálózat. A hálózat bemenetére helyeztünk egy Multi-Head Attention blokkot [16], melynek a bemenete a környezetekből érkező megfigyelésekből képzett tenzor. Mivel A2C-t használunk, így logikus több környezetet futtatnunk egyszerre, hardware erőforrásainktól függően akár például 16-ot is tudnánk egyszerre. A saját hardware konfigurációmmal 1-4-t futtattam egyszerre. A háló kimenetén egy RAdam (Rectified Adam) optimizer algoritmus található, mely a háló jelenlegi paramétereit és veszteségfüggvény gradienseit (stratégia gradiens) felhasználva számolja ki a háló új, frissített paramétereit. Az Actor fej kimenete a stratégia, vagyis az akciók eloszlása. A Critic fej kimenete az állapot-érték, melyből a Q diszkontált jutalmat kivonva (1.8 egyenlet) kapjuk meg az előnyt, pontosabban annak inverzét. Ezt amiatt érdemes így csinálni, mert nekünk nem gradiens csökkentés kell jelen esetben, hanem növelés az A2C miatt.



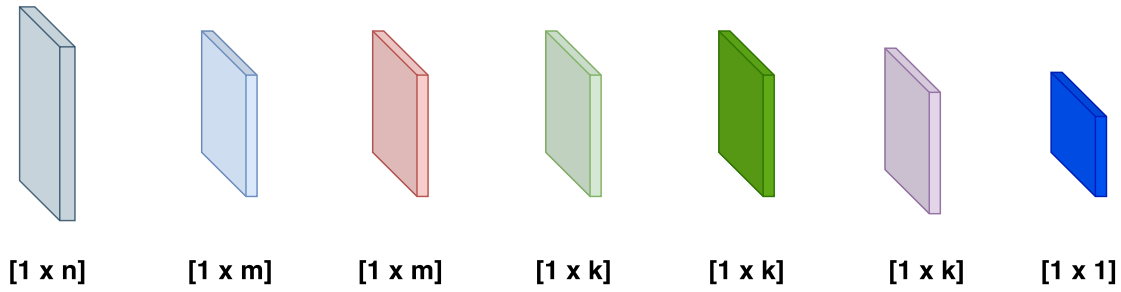
3.1. ábra Az architektúra jelenlegi állapota

3.1.1 Multi-head Attention

2019-ben nagy népszerűségnek örvendtek az ún. Transformer típusú neurális hálózatok [17]. Ezeknél a seq-to-seq, főleg nyelvi fordításra használt hálóknál

alkalmazott eljárások az Encoder-Decoder Attention és a Self-Attention. Az utóbbi esetében N darab bemenet interaktál egymással (self) és „kitalálják”, hogy melyikükre figyeljenek a legjobban (attention). Ezzel szemben az Encoder-Decoder Attention metódusban a bemenet a cél kimenettel interaktál. A Multi-head Attention megértéséhez előbb nézzük meg a Self-attention működését.

Minden bemeneti vektornak 3 reprezentációja van: egy *key* (**k**), *query* (**q**) és *value* (**v**) vektor. Ahhoz, hogy megkapjuk ezeket a vektorokat, a bemeneti **x** vektort egy az adott reprezentációhoz tartozó súlymátrixxal ($\mathbf{W}^K$, $\mathbf{W}^Q$, $\mathbf{W}^V$), például a $\mathbf{W}^K$ kulcs-mátrixxal szorzunk. Ezeknek a méretei a **3.2. ábra** alapján könnyedén megadhatóak: a $\mathbf{W}^K$ és $\mathbf{W}^Q$ mátrixnak azonos méretűnek kell lenniük, például $n \times m$-esnek, míg a $\mathbf{W}^V$ egy $n \times k$ méretű mátrix.



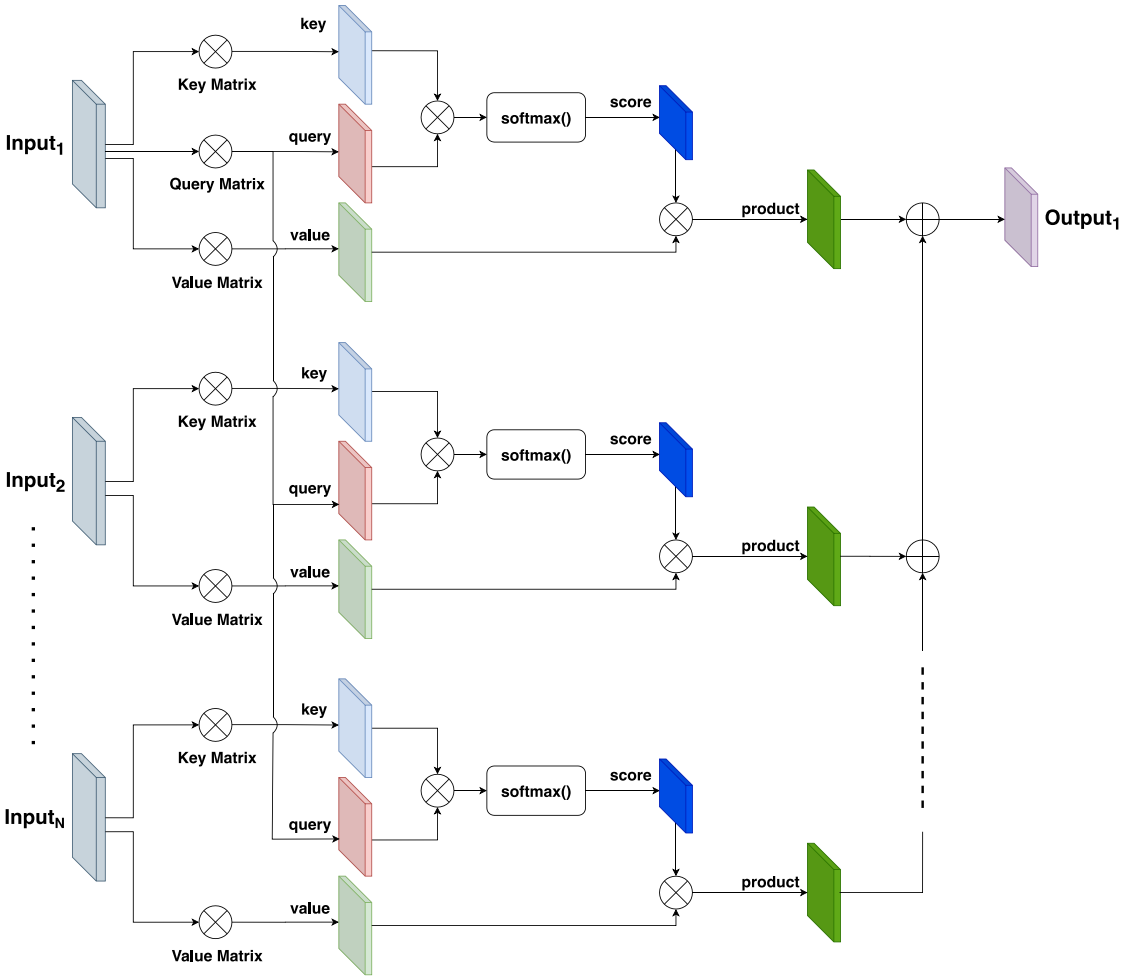
3.2. ábra A Self-Attention algoritmusban használt vektorok méretei.

Természetesen a bemeneti vektorokat egy mátrixba rendezve ($\mathbf{X}$) [illegible] módon 3 mátrix szorzással megkaphatjuk a reprezentációk **K**, **Q** és **V** mátrixát. Ezekután kiszámoljuk az első bemeneti $\mathbf{x}_1$ vektorhoz tartozó figyelem pontot. A **3.2 ábrán** látható sötétkék téglatest ezt a skalárt jelöli. Fontos, hogy az első bementhez tartozó kimeneti $\mathbf{y}_1$ vektor (a **3.3. ábrán** az Output_1) számításához csak az első bemenet $\mathbf{q}_1$ vektorát kell felhasználni. Ugyanígy a figyelem pontokhoz is csak a $\mathbf{q}_1$ kell. Ezek alapján, például az *i*-edik figyelem pontot a $\mathbf{q}_1$ és a $\mathbf{k}_i$ felhasználásival kapjuk meg. Tehát a $\mathbf{q}_1$ $1 \times m$ -es vektort szorozzuk az összes *key* reprezentációt tartalmazó **K** $m \times N$ -es mátrixával, ahol N a bemeneti vektorok száma. Így gyorsan megkaphatjuk az $1 \times N$ méretű figyelem pont vektort. Ezután a szorzatra számolunk egy *softmax*-ot, [illegible]ely a bemenetére adott vektor elemeit 0 és 1 közötti elemekre képezi, úgy, hogy az elemek összege 1 legyen.

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3.1}$$

A következő lépés, hogy a *value* reprezentációkat szorozzuk a kiszámolt figyelem pontokkal, majd ezeket az $1 \times k$ méretű súlyozott vektorokat összeadjuk elemenként. Az így megkapott vektor az első bemeneti vektorhoz tartozó kimeneti $\mathbf{y_1}$ reprezentációja. Láthatjuk, hogy a kimenet méretét a $\mathbf{W}^{V}$ mátrix oszlopainak számával határozhatjuk meg, a példa esetében k darab oszlopa van. Legvégül ezt a lépés sorozatot megismételjük minden bemeneti vektorra és megkapjuk a kimeneti vektorokból álló $\mathbf{Y}$ $N \times k$ méretű mátrixot.

Most már részletesebben is megérthetjük, mi a különbség a kétfajta Attention mechanizmus között. A Self-Attention esetében mind a 3 reprezentáció vektor a bemenettől származik, míg az Encoder-Decoder Attention esetében csak a *key* és *value* származik a bemenettől (enkóder), míg a *query* vektor a cél kimenettől (dekóder).



3.3. ábra A Self-Attention működése az első bemeneti Input_1 vektor esetére.

A Multi-head Attention igazából nem más, mint több Self-Attention blokk párhuzamos működése. A $\mathbf{W}^{K}$, $\mathbf{W}^{Q}$ és $\mathbf{W}^{V}$ súlymátrixokat kis értékekkel szokás

inicializálni, például Gauss-eloszlással. Tanításkor ezeknek a mátrixoknak a súlyait frissíti a háló. A Multi-head előnye, hogy így az egyes Self-Attention blokkok különböző featureökre tanulhatnak rá, mivel a minden fejhez tartozó 3-3 súlymátrixot véletlenszerűen inicializálunk. A már említett mátrixokon kívül megtalálható benne még egy **Z** súlymátrix, melynek elemeit szintén tanításkor frissíti a háló. Ez a mátrix arra szolgál, hogy az első bementhez tartozó *h* fejű Multi-head Attention *h* darab kimeneti $\mathbf{y_1}$ vektorjaiból képzett $\mathbf{Y_1}$ mátrixot a **Z**-vel súlyozva kapunk egy kimeneti $\mathbf{z_1}$ vektort. Ezeket a $\mathbf{z_i}$ $(i = 1 \dots N)$ vektorokat adjuk át a neurális hálózatnak.

Mivel a függvényt nem is olyan régen implementálták PyTorch-ban, így ezzel szerencsére nem kellett külön foglalkozni.

3.1.2 A2C

Az architektúra lelke, vagyis maga az ágens az A2C policy gradient módszert megvalósító algoritmus. Ezt alkalmazom az optimális stratégia megtanulására.

A projekt első fázisának nem volt szkópja, hogy ezt az algoritmust tökéletesítsem, vagy fejlesszem le. Keresni kellett egy jól megírt és könnyedén használható implementációt. Első körben a Stable Baselines kódjait használtam fel, de itt két akadályba is ütköztem. A kisebb gond az volt, hogy TensorFlow felhasználásával íródott, mellyel továbbra is összeegyeztethetőségi problémák vannak. Nagyobb gondot okozott, hogy rengeteg absztrakt függvényt kellett volna implementálni, erre pedig jelenleg nem terveztünk időt szánni.

Így végül egy diáktársam kódját használtam fel. Kisebb módosításokat eszközöltem rajta, mely elég volt arra, hogy sikeresen fusson a kód és tanuljon az ágens. A felépítése elég egyszerű. A bemenetén egy konvolúciós blokk található, mely négy 2D konvolúciós rétegből áll [18], LeakyReLU aktivációsfüggvénnyel és a végén egy Average pooling réteg gondoskodik a dimenziócsökkentésről. Ezt követi opcionálisan egy LSTM réteg (Long Short-Term Memory [19]), melyből leágazik a két fej, azaz 1-1 lineáris réteg. Az Actor mérete az akciók száma, míg a Critic fej egy skalárt ad vissza.

3.1.3 RAdam

A Rectified Adam egy módosított Adam (Adaptive Moment Estimation [20]) algoritmus, ez a state-of-the-art optimalizáló eljárás. Azonban mielőtt rátérnék, hogy miért jobb a Rectified Adam, előbb nézzük meg, hogy működik az egyszerű Adam.

Az Adam két ismert algoritmus, az RMSProp és az AdaGrad jó tulajdonságait ötvözi. Célja a nevéből is adódóan az adaptív tanulási sebesség, akárcsak az RMSPropnál viszont itt a gradiens négyzetek összegzésén kívül a gradienseket is összegezzük, és kijavítja az AdaGrad nagy hátrányát: az időben csökkenő tanulási sebességet. A függvény paraméterei, az α, mely nem más, mint a tanulási ráta vagy lépéshossz (szokták η-val is jelölni), a β_1 és β_2, melyek a gradiensek első (átlag) és második momentumának (középnélküli varianciája) exponenciális felejtési rátája. Ezek 1 körüli értékek, míg az α egy kicsi szám. Valamint szükség van még az epszilonra (ε), mely a numerikus stabilitást biztosítja, azaz, hogy a nevező értéke sose lehessen nulla. Az Adam-et kiötlők ajánlásai alapján ezeket a következő módon szokás beállítani: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ és $\varepsilon = 10^{-8}$. Ha megnézzük az Adam PyTorch-os implementációját, default paraméterként ugyanezeket az értékeket fogjuk látni. Ezeket a paramétereket felhasználva tudjuk kiszámolni az első és második momentumot. Természetesen szükségünk van még a gradiens vektorra, melyet megkapunk a költségfüggvény θ szerinti deriváltjából, ahol a θ a háló paraméterei. A *t* alsó index az időlépést, azaz az időbeli iterációt jelöli.

$$g_t = \nabla_\theta J(\theta_t) \tag{3.2}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.3}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t \tag{3.4}$$

Egy további korrekciót kell még alkalmazni, ha netán a gradiensek átlaga és a gradiens négyzetek átlaga kezdetben nagyon kis értékűek lennének, akkor van rá esély, hogy beragadnak ilyen kis értéken. Ezért korrigálunk a bétákkal („bias-corrected” mozgó átlag és mozgó második momentum), így a kezdeti értékek (*t* = 0) a gradiensek és gradiens négyzetek lesznek (Hadamard/elemenkénti szorzatuk), így az egyenletek a következőképpen alakulnak:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.5}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.6}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.7}$$

A probléma az Adammal, hogy kezdetben nagy a variancia, melyet jó lenne csökkenteni. Erre az egyik módszer a *warmup* (AdamW), azaz, hogy a tanulási ráta nem egy konstans, vagy csökkenő érték (*decay*), hanem egy bizonyos T ideig kezdetben növeljük az alfát, ezzel csökkentve a varianciát. A *rectified* ezzel szemben úgy oldja meg ezt a problémát, hogy először kiszámoljuk az egyszerű mozgó átlag közelítésének (SMA) a maximum hosszát, melyet ρ_∞ -val jelölünk. Majd ezt felhasználva minden iterációban kiszámoljuk a ρ_t -t és ha ez átlép egy küszöböt, akkor változtatunk a tanulási rátán, azaz az alfán, mely egyébként jelen esetben egy konstans. Pontosabban beszorzunk egy ún. *variance rectification* (3.11 egyenlet) taggal. Egyéb esetben csak alfával súlyozzuk az első momentumot (3.13 egyenlet).

$$\rho_\infty = \frac{2}{1-\beta_2} - 1 \tag{3.8}$$

$$\rho_t = \rho_\infty - \frac{2t\beta_2^t}{1-\beta_2^t} \tag{3.9}$$

$$l_t = \frac{1}{\sqrt{\hat{v}_t}} \tag{3.10}$$

$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}} \tag{3.11}$$

$$\theta_{t+1} = \theta_t - \alpha r_t l_t \hat{m}_t \tag{3.12}$$

$$\theta_{t+1} = \theta_t - \alpha \hat{m}_t \tag{3.13}$$

4 Szimulációs környezet

A célhardveren tanítani, illetve folyton tesztelni a hálót lassú lenne, mert felesleges overheadet okozna az integráció, hiszen mindig jelen kéne lenni a laborban. Ezenkívül költséges is lehet, hiszen, egy komolyabb hiba vagy rossz döntés miatt tönkre mehet a versenyautó. Ezért célszerű a szoftver működését egy jó fizikai motorral szimulált környezetben tanítani és tesztelni, ahol ezek a hátrányok mind kiküszöbölhetők.

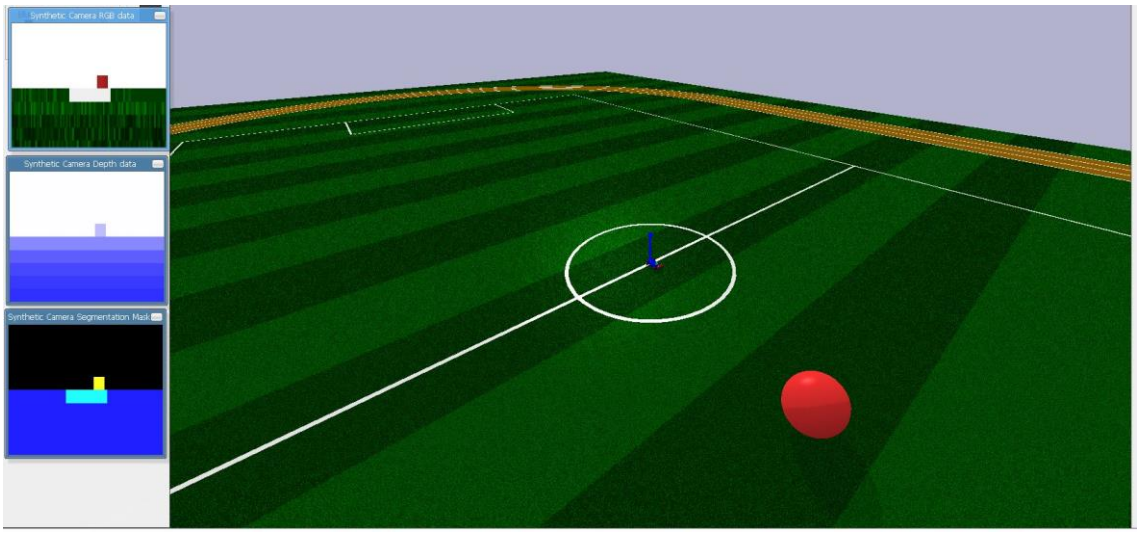
A félév során több környezettel is próbálkoztunk. Először a SafetyGym-mel, melyről kiderült, hogy a MuJoCo fizikai motort használja, melyre ingyen csak diákként lehet licenszt szerezni, viszont csak egy évre és egy adott gépre. Ez kizárja annak a lehetőségét, hogy a Colab-on igénybe vehessük, valamint a jövőbeli fejleszthetőséget kockáztatjuk meg azzal, ha nem kapunk később licenszt, vagy csak az eredeti árán, mely jelenleg 500€. Így másik motor után kellett nézni, míg végül a PyBullet-re esett a választás.

4.1 PyBullet részletesebb bemutatása

A PyBullet szimulációs környezete teljesen ingyenes mindenkinek, így Colabon is lehetne elvileg futtatni. Azonban itt sok problémába ütköztem. Mivel a „debug”-hoz, vagyis, hogy nyomon tudjuk követni az ágens akcióit a környezetben igen hasznos funkció lenne, ha meg tudnánk jeleníteni a környezetet. Így az egy fontos szempont volt, hogy a Colabon meg tudjuk ezt valósítani. Sajnálatos módon, mint kiderült ez egy nagyon nehezen megvalósítható feature. Egyszerűbb környezetekkel, mint például az Atari ezt sikerült elérni oly módon, hogy egy függvény videót készít a környezetben elvégzett akciókról és a futás végén ezt letölthetjük, visszanézhetjük. De a PyBullet egy SDK-ban (Software Development Kit) fut, melyet a Colab-bal nehéz megnyitni és sajnos nem lehet videóra rögzíteni sem. Így a Colab-ról egyelőre le kell mondanunk, amíg nem lesz tökéletes a szoftver működése és nem lesz kész hozzá a környezet, hogy mellőzni lehessen a szimuláció renderelést.

A feladatunkhoz a PyBullet fejlesztői kettő hasonló környezetet építettek már ki. Mindkettőben az ágens egy kis távirányítós autó, ennek kell eljutnia egy nagy üres pályán (eredetileg egy focipályán) a pálya közepétől egy a pályán véletlenszerűen elhelyezett labdához. A két környezet abban különbözik leginkább, hogy míg az egyiknél a

megfigyelés csak a labda pozíciója a kamera képén (x,y), addig a másiknál a teljes RGB-D kamera kimenete. Az utóbbi környezetet fejlesztettem tovább, azonkívül, hogy kialakítottam egy kezdetleges tesztpályát a kocsinak, sok függvényt kellett kijavítani.

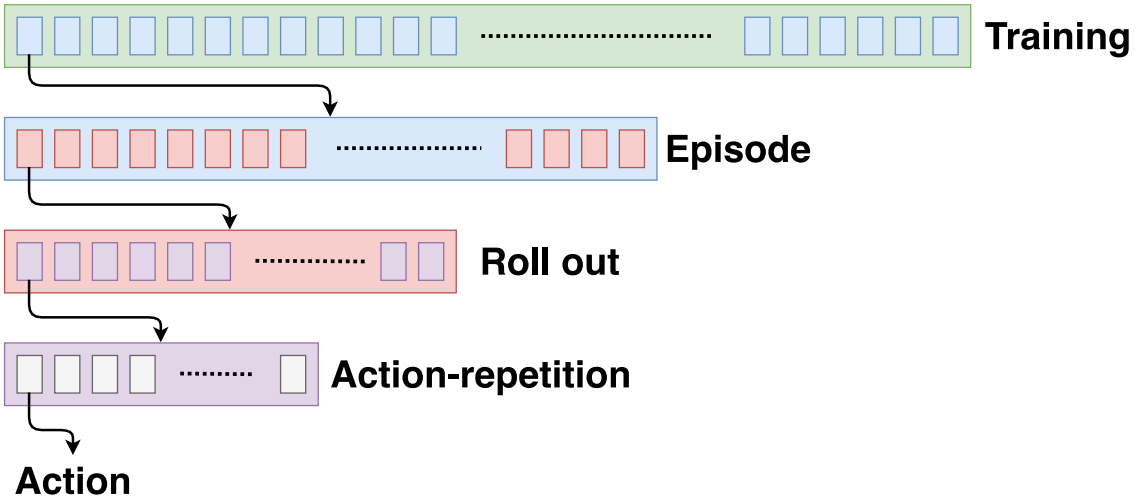


4.1. ábra Az eredeti PyBullet-es környezet, bal oldalt a kamera képe látható (felülről lefelé: RGB, mélység, szegmentált)

A Stable Baselines az OpenAI Baselines függvénykönyvtárán alapuló megerősítéses tanulást könnyítő függvények implementációit tartalmazza, a kezdőbb fejlesztők számára ajánlják, mivel a Baselines-nál stabilabb. Ennek a része is a SubprocVecEnv, mellyel könnyedén vektorizálhatjuk a környezeteinket. A Baselines viszont nem PyTorch, hanem TensorFlow segítségével íródott, mely elég sok gondolt okozott az összeegyeztetéseknél, de végül sikerült megoldani.

A tanítás folyamata igen lassú lenne, ha minden állapotnál egy akciót vennénk csak figyelembe, ezért érdemes több akciót megvárni, és csak azután frissíteni a modell paramétereit. Ennek a folyamatnak a megnevezése a roll-out, mellyel megadhatjuk, hogy hány lépést várjunk meg, mielőtt kiszámoljuk költségfüggvény eredményét. Ez felfogható úgy, mint a megfigyeléses tanulásnál alkalmazott batch fogalma, ahol szintén sokat lassítana a tanításon, ha az adatokat egyesével adnánk a hálónak és emiatt adatonként kellene frissítenie a paramétereit. Szokásosan kis számot szoktak meg adni roll-out-nak, például 5 vagy 6. Egy tanítást, mint már korábban említettem epizódokra bontunk (epoch), az epizód végén újraindítjuk az epizódot és ismét a nulladik állapotból indul az ágens. Egy epizód roll-out-okból áll (batch), viszont egy roll-out-ba is összefoghatunk több akciót, ez a **4.2. ábrán** látható. Tehát a jutalmat sem egyesével,

hanem például csak 10 akció után számoljuk ki, ugyanígy a megfigyelést is csak 10 akció után végezzük el.



4.2. ábra

A mi esetünkben egy tanítás például 100 epizódból áll, 1 epizód 80 roll-out-ból, 1 roll-out 5 akció-ismétlésből, melyben 5 akciót fogunk össze. Tehát egy tanítás során az ágens 200 ezer akciót végez, ha 4 környezetet párhuzamosítunk, akkor máris 800 ezer akcióról beszélünk.

A PyBullet egyik hasznos tulajdonsága, hogy a kezelőfelületén gombokat és csúszkákat helyezhetünk el, melyekkel bármilyen paramétert állíthatjuk. Sajnos a gomb implementálása kissé bug-osra sikerült, így a gomb helyén is csúszka jelenik meg. A **4.3. ábrán** látható felső kettő csúszka igazából gombok. A felső arra szolgál, hogy szeretnénk-e azt, hogy az epizódok kezdetén véletlenszerű pályát hozzon létre a környezet vagy sem. Ezt a funkciót jelenlég még nem implementáltam. A második gombot, ha bekapcsoljuk, akkor kocsi fölé állítja a kamerát, így tudjuk követni a kocsit könnyedén. Kikapcsolva szabadon nézelődhetünk a szimulációban. Az alatta lévő 6 csúszka a később említésre kerülő jutalmak súlyait állítja, melyet így szimuláció közben is hangolhatunk folyamatosan. A legalsó csúszkával a mozgó objektum sebességét változtathatjuk bármikor a szimuláció során.

4.3. ábra Gombok és csúszkák a GUI felületén

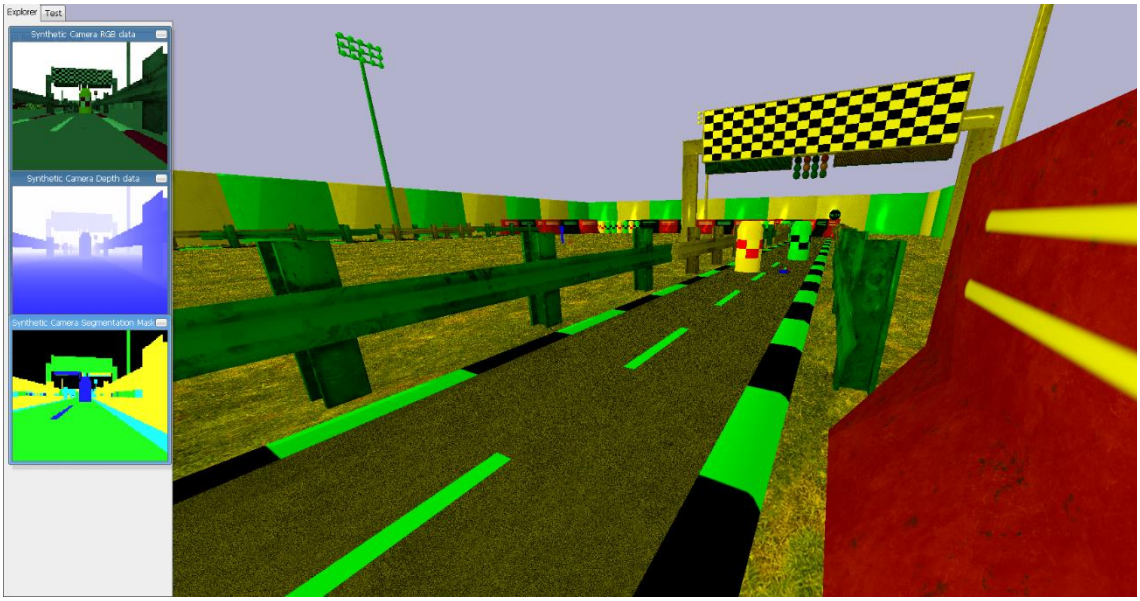
4.2 Objektumok beolvasása

A PyBullet egyik legnagyobb hátránya, hogy meglehetősen limitáltan tudja beolvasni az objektumokat és elhelyezni a környezetben. Alapvetően két fájl típust tud kezelni, az URDF és SDF fájlokat, ezek mind XML típusúak. Az előbbi a bonyolultabb objektumokat írja le, tipikusan a robotokat, alapvetően ezek az ágensek a szimulációban, míg az utóbbi az egyszerűbb tereptárgyak jellemzőit írja le. Ezek olyan tulajdonságokat írnak le, mint például tömeg, súrlódás, pozíció és orientáció, szín, anyagjellemzők, súrlódás stb. A PyBullet-nél csak kevés ilyen előre elkészített modell volt számunkra hasznos, így csak a versenyautó URDF modelljét használjuk fel. A többi objektumot kezdetben kézzel próbáltam elkészíteni, vagyis XML fájlokat írtam, melyekbe ingyenesen beszerzett OBJ fájlokat linkeltem. De ez rengeteg időnek tűnt, így alternatív megoldások után néztem. Végül megláttam azt, hogy néhány hónapja egy új commit részeként felkerült egy olyan segédfüggvény, mely WORLD típusú XML fájlokat tud beolvasni. Ez tömören egy tereptárgyakat (SDF) összefoglaló XML, melyben minden objektum megjelenik. Ezeket, valamint a korábban említett két fájl típust is a Gazebo 3D robot szimulátor szoftver készíti el. Így lehetőség nyílt arra, hogy vagy tervezzek

magamnak egy egész pályát, melyet egy az egyben be tud már olvasni a PyBullet, vagy egyszerűen keresek egy Gazebo-ban tervezett kész pályát. Végül találtam is egy szimpatikus pályát, mely sok tekintetben hasonlít az elképzeléseinkre, és egyelőre tökéletesen megfelel a szimulációhoz. Ha mégis kell, akkor könnyedén lehet változtatni rajta. Sok féle objektum található ezen a pályán, különböző falak és korlátok, lámpák, sávok, bóják stb. Egyetlen komolyabb hátránya, hogy maga a versenypálya egy fix objektum, és nem több részből áll, így jelenleg fixen futópálya alakú a versenypálya.

A felesleges tereptárgyak törlése és egyéb módosítások után még elhelyeztem két álló objektumot, valamint egy mozgó objektumot is magán a versenypályán. A versenyautót a pályán véletlenszerűen helyezzük el az epizódok elején. A többi pálya elemet még fix pozícióval generáljuk a környezetbe, a jövőben ezeket is elhelyezhetjük véletlenszerűen akár. Valamint kiemelendő objektum még a célvonal, valamint egy közlekedési lámpa is, melyeket felhasználunk a jutalom számításához. A pályán található még egy Stop tábla is, de ezt és egyéb táblákat jelenleg még nem használunk fel az autó tanításához.

Itt megemlíteném, a PyBullet második bug-ját, a WORLD és SDF fájlok beolvasása nem tökéletes, valamiért elrontja az anyagjellemzőket. Ez abban mutatkozik meg, hogy az objektumokat felváltva sárga vagy zöld színnel „maszkolja”. Ezt sajnos nem tudtam sehogy sem javítani. Az alábbi képen az elkészült versenypálya látható.



4.4. ábra Az elkészült pálya, a hibás színkezeléssel együtt

4.3 Jutalom függvény

A másik legfontosabb feladat a környezet elkészítése mellett a jutalmazó függvény megírása, mely eldönti, hogy az adott akcióra mekkora jutalmat ad. Ez kulcsfontosságú lépés, hiszen itt sok elvi hibát lehet ejteni, könnyű beleesni a kobra effektus esetébe, azaz, hogy azt hisszük egy megoldási javaslat tényleg megoldja a problémát/feladatot, de igazából csak még inkább rontunk rajta. Ezért sokat kell tesztelni, nehogy az ágensünk furcsa vagy haszontalan dolgot tanuljon meg. Például ne vágja le az utat egy kanyar helyett, vagy ne tolasson be a célba stb. Eddig öt féle jutalmazást implementáltam, melyeknél az ágens különböző objektum típusokra reagál. További jutalmazásokra azonban szükség lehet, például jelenleg az időt, a táblákat nem vesszük figyelembe.

Az alábbi öt jutalomból csak az első kettő folytonos, az utolsó három diszkrét. A diszkrét jutalmak, habár jó eredményre visznek minket, lassabb konvergenciát okoz, mivel az ágens ritkábban kap visszacsatolást a döntései után. A végső jutalmat az öt részeredmény súlyozott összegéből kapjuk meg:

$$R = w_R\left(w_\alpha\alpha + w_\beta\beta + w_\gamma\gamma + w_\delta\delta + w_\varepsilon\varepsilon\right) \tag{4.1}$$

4.3.1 Alfa

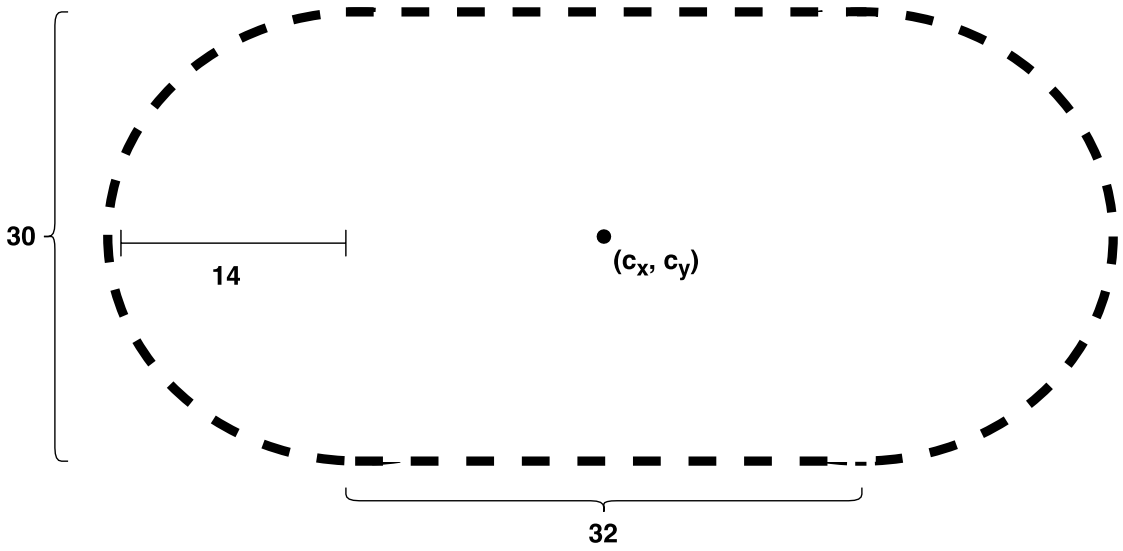
Ez a legegyszerűbb jutalmazó algoritmus az öt közül. Célja, hogy eljuttassa egy célba (időtől függetlenül). A jutalom annak a függvényében pozitív, hogy az előző állapothoz képest közelebb jutott-e a célhoz vagy sem. Ha nagyobb lett a távolság, azaz a céltól elfelé mozdult, akkor negatív a jutalom:

$$\alpha_t = dist_{t-1} - dist_t \tag{4.2}$$

4.3.2 Béta

A béta viszont a legbonyolultabb algoritmus ezek közül. Ez a sávtartásért felel, azaz az ágensünk, ne menjen át a szembe sávba és ne mehessen le a pályáról sem. Nehézsége abból adódott, hogy a sáv is egy fix objektum, melynek egyetlen pozíciója van (a középpont pozíciója), így ezzel máshogy kell távolságokat számolni. Nem találtam hasznos leírást, hogy mekkorák a méretei, így lemértem és függvényt illesztettem rá. Úgy kezelem, mintha lenne egy téglalap, melynek két oldalán egy-egy félkör található. A

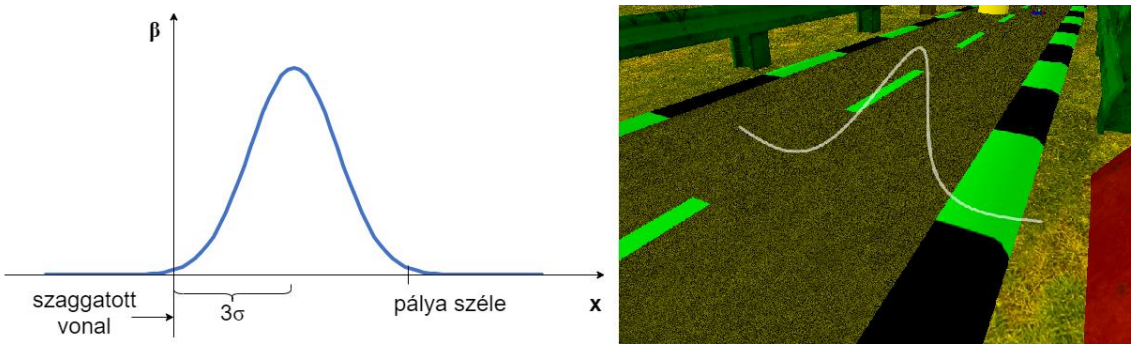
téglalap oldalai egész számokra jöttek ki, így pontosnak gondolom a mérést, de a félkörök nem pontosan félkörök, a tetejük kissé nyomott, így csak becsülöm a sugarat (**4.5. ábra**). Ezeket az adatokat felhasználva a szimuláció koordináta-rendszerében már egész pontosan lehet becsülni az autó helyzetét a sávokon belül.



4.5. ábra A szaggatott vonal méretei

Miután megbecsültük a pozíciót, már csak a pontozás mértékét kell beállítani, Ehhez Gauss-görbét használunk, melynek a maximuma a jobb oldali sáv közepére van állítva, a koordináta-rendszer origója a szaggatott sávon helyezkedik el (**4.6. ábra**). A görbe szélességét úgy állítottam be, hogy a 3σ távolság a sáv széleire essen. Ezen a távolságon kívül balra, vagyis a másik sávban nincs változtatás az úttest széléig, de a pozitív irányba, azaz az úttest szélétől kifelé súlyosan büntetünk (másik szélén is), innentől egy nagy abszolút értékű negatív számra (pl.: -100) állítjuk a bétát. Így az úttesten nem büntetünk, mivel a Gauss-görbe minimuma nulla, tehát venni kéne a természetes logaritmusát, így már negatív értékeket is kapunk a β-ra az úttesten, miközben a maximumát ugyanúgy a jobboldali sáv közepén veszi fel:

$$\beta = \begin{cases} \ln \dfrac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, & -6\sigma < x < 6\sigma \\ -100, & x \geq 6\sigma \\ -100, & x \leq -6\sigma \end{cases} \tag{4.3}$$



4.6. ábra Gauss-görbe elhelyezkedése a pályán

4.3.3 Gamma

A következő jutalmazás feladata, hogy a kocsi megálljon a piros lámpánál. Ez állapotgép-szerűen működik, bizonyos időközönként vált a lámpa piros és zöld között (sárgával egyelőre még nem szükséges foglalkozni). Ez grafikusan még nem jelenik meg a környezetben, az objektum változtatásokat megvalósítani a PyBullet-ben nem triviális.

Ha zöld a lámpa a gamma értéke nulla. Ha piros, akkor megvizsgáljuk az ágens távolságát a lámpától (2D-ben), és ha 2 egységen (méter) belül van, akkor a célunk, hogy álljon meg a kocsi, vagyis csökkentsük le a sebességét nullára. Tehát a béta legyen a sebesség mínusz egyszerese. Ha a kocsi már fél egységre megközelítette a lámpát, miközben piros, akkor a gammát is egy nagy abszolútértékű negatív konstansra állítjuk:

$$\gamma = \begin{cases} 0, & x > 2 \\ -\sqrt{{v_x}^2 + {v_y}^2}, & 2 \geq x \geq 0.5 \\ -100, & x < 0.5 \end{cases} \tag{4.4}$$

4.3.4 Delta

Biztonsági szempontból az egyik legfontosabb szempont, hogy az ágens kerülje ki az útjába kerülő objektumokat. A delta jutalom, annál nagyobb, minél kevésbé közelít meg egy álló objektumot. Egyelőre a legközelebbi objektum távolságát vetjük össze az ágens távolságával. Ezt a későbbiekben érdemes lesz még tovább fejleszteni, például, hogy csak arra az objektumra figyeljen, amerre halad, vagy akár több objektum távolságát is figyelembe vegye egyszerre. Jelenleg itt egy egyszerű Gauss-görbét használunk fel, mely a távolság csökkenésével egyre jobban büntet, de fontos, hogy korlátosan:

$$\delta = \begin{cases} -e^{-x^2}, & x < 1 \\ 0, & x \geq 1 \end{cases} \tag{4.5}$$

4.3.5 Epszilon

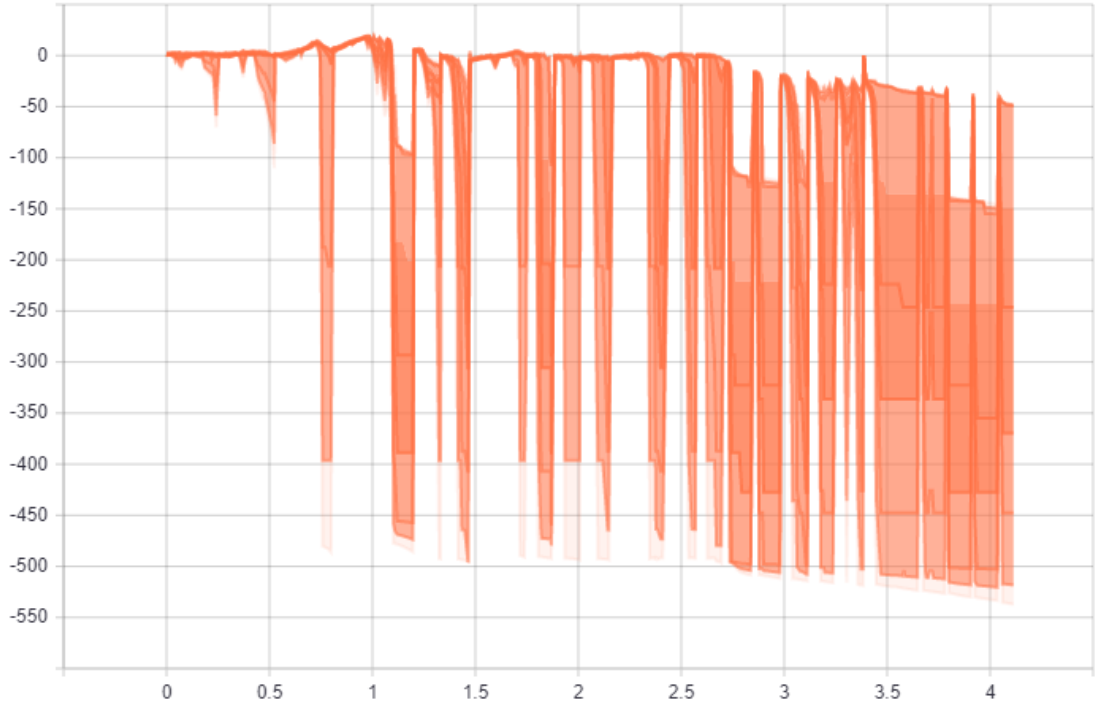
A mozgó járműveket is célszerű lenne elkerülnie az ágensnek, erre szolgál az epszilon algoritmus. A mozgó objektum a GUI-ban állítható konstans sebességgel mozog két fix pont között oda-vissza. Jelenleg egy ilyen objektum található a pályán, mely az úttest két széle között ingázik. Ha az ágens túlságosan megközelíti, például 1 méterre, akkor negatív jutalmat adunk. Ezt egy az abszcissza tükrözött Gauss-görbével érjük el. Annak érdekében, hogy ne legyen nagy ugrás $x = 1$-nél, főleg, amikor növeljük az ε súlyát az összjutalom számításakor, ezért a szórásnak a súly (w_ε) reciprokát választom:

$$\varepsilon = \begin{cases} \dfrac{1}{\ln \dfrac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(w_\varepsilon \cdot x)^2}}, & x < 1 \\ 0, & x \geq 1 \end{cases} \tag{4.6}$$

5 Tanítás, eredmények

Miután elkészült a környezet is több tanítást futtattam, mindegyik tanításnál 4 ágens tanult párhuzamosan. Az részeredményeket Tensorboard [21] segítségével ábrázoltam. Az alábbi képeken a jutalom alakulását lehet megfigyelni az időfüggvényében, a felbontás órákban van megadva.

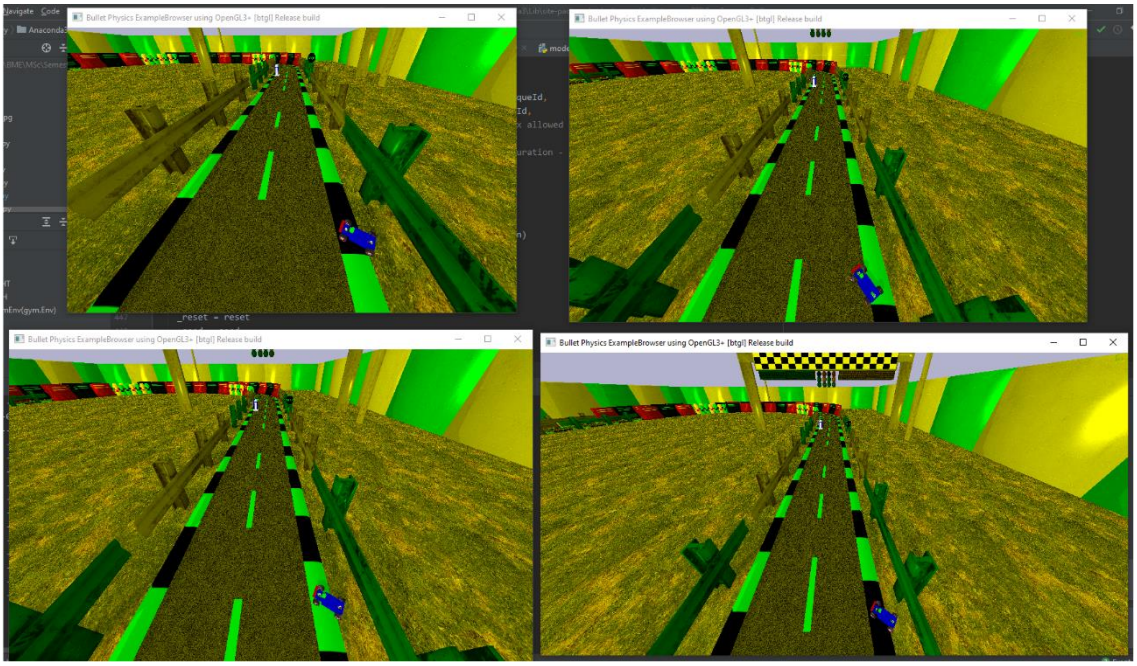
Az első méréseknél a jutalmak súlyai mind 1 értékűre voltak állítva, de látszódott, hogy 1 órán nagyon elszaporodtak a nagy negatív értékek (**5.1. ábra**), melyekből arra következtettem, hogy a kocsi folyton elhagyja a versenypályát. Mivel annak érdekében, hogy a tanítás ne legyen nagyon lassú, alap esetben a renderelés ki van kapcsolva. Bekapcsolva meg is bizonyosodtam, hogy az ágens rátanult egy olyan műveletsorozatra, melynél elindul hátrafelé, majd jobbra elhagyja a pályát és fennakad (**5.2. ábra**).



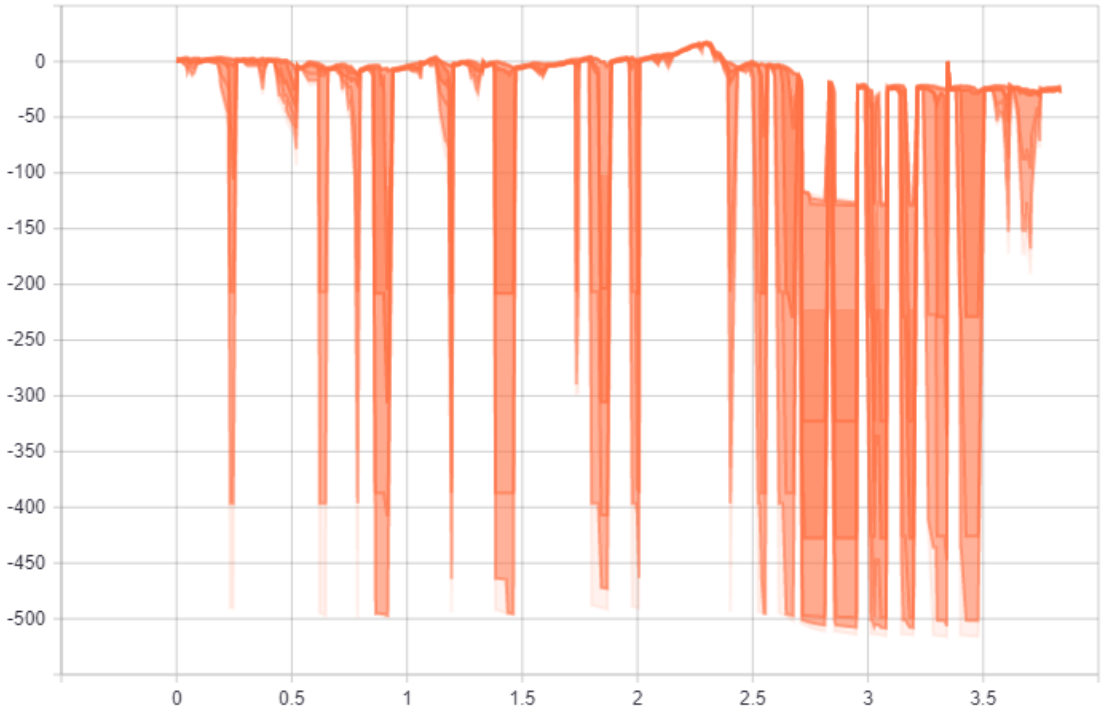
5.1. ábra A jutalom alakulása 1 súlyok esetén.

Először azt gondoltam, hogy a hiba abból ered, hogy a kocsi túl közel kerül az egyik álló objektumhoz, és rátanult, arra, hogy minél messzebb kerüljön attól, akár tolatással. De az objektumot eltávolítva is jelentkezett a hibás működés. Végül a jutalom algoritmusokat elemezve rájöttem, hogy közel sem esik egy nagyságrendbe az alfa a többivel, így esélye sincs megtanulni, hogy a cél felé kéne haladnia. Ezt javítva a w_α-át

30-ra állítva már elérjük azt is, hogyha a sávközepén tolat a kocsi, akkor is negatív a jutalom, eddig ez mindig pozitív jutalmat adott. Az **5.3. ábrán** láthatjuk a tanítás eredményét ezen paraméterek esetén. Látható, hogy kevesebbszer hagyta el a kocsi a pályát, de így is rátanult egy hasonló műveletre és beállt a jutalom értéke -25 körülire.

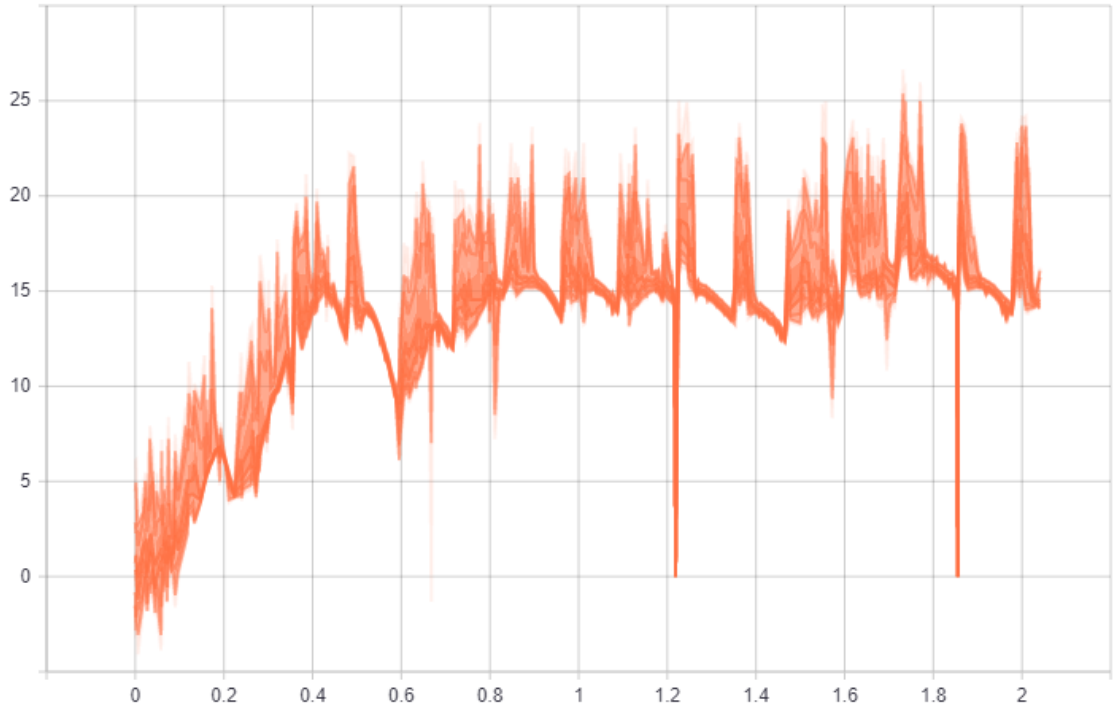


5.2. ábra A hibás betanulás



5.3. ábra A jutalom még a javítás után is negatív értékre konvergál.

A hiba javításához egyesével kéne végig nézni a különböző jutalmak alakulását az epizódok során. Egyelőre csak az alfa ellenőrzésére került sor, tehát ezesetben: $R = 30\alpha$. Az **5.4. ábrán** látható, hogy így már két óra után konvergált a jutalom értéke +15 környékére. Vagyis a kocsi határozottan a cél felé haladt és nem is lassan. Elvileg a célt el is érte, bár itt sem rendereltem, így ezt nem tudtam leellenőrizni. A lefelé tartó nagyobb csúcsok az epizódok kezdetét jelöli, itt a jutalom kis értékről nagyon gyorsan felugrik egy nagyobb értékre, majd szépen beáll a reális várható értékre az epizód vége felé.



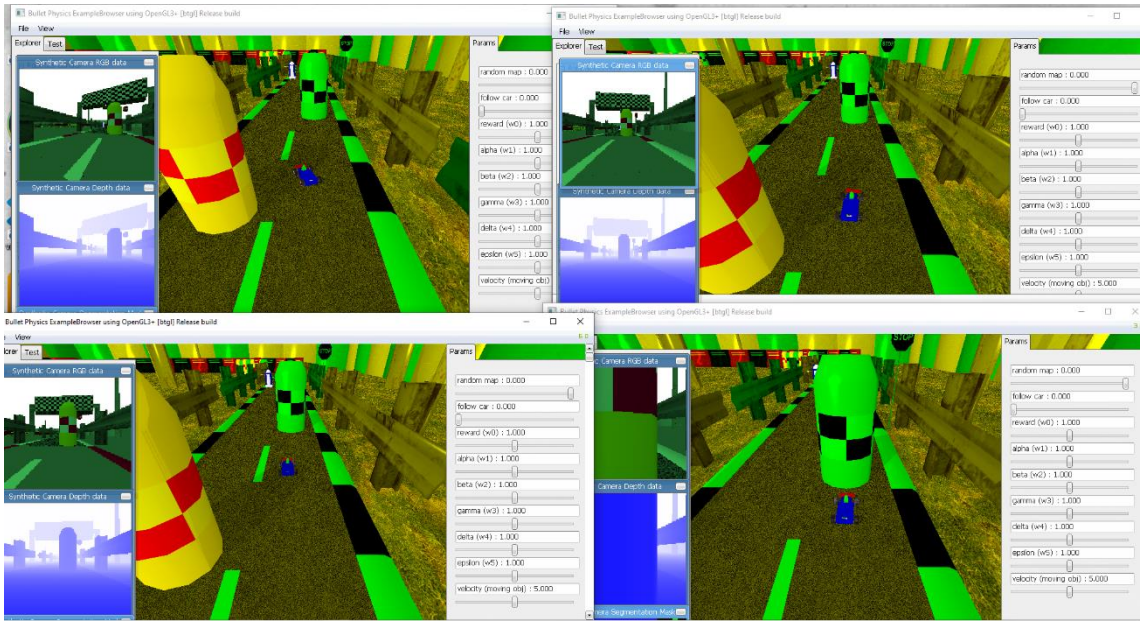
5.4. ábra Az alfa pozitív értékre konvergál.

6 Összefoglaló

A projekt ezen szakaszának a célja az volt, hogy miután elkészült a kezdeti architektúra, azt egy jól működő szimulációs környezetben tanítani és tesztelni legyünk képesek. A környezetet és a jutalom függvényt sikerült az általunk elképzelt feladatra szabni.

A következő fázisra sok cél van. Sajnos még sokat kell hangolni a jutalmak súlyain, akár magukon az algoritmusokon kell változtatásokat eszközölni. Érdemes lesz további jutalmazásokat és felhasználni a tanítás során, például az időt. Annál nagyobb a jutalom értéke, minél korábban ért el a célba a kocsi az epizód végezte előtt.

A pályával kapcsolatban is kellene még sok minden javítani. Az egyik ilyen, hogy nem teljesen véletlenszerűen generált a pálya. Ennél a legnagyobb akadályt az úttest fogja jelenteni, mely ugyebár egy objektumból áll. Későbbiekben a táblafelismerést is bele kéne a funkciók közé, így táblákat is el kell majd helyezni a környezetben. Ez a magas szintű randomizáció azt a célt szolgálja, hogy a kiskocsi vezetni tanuljon meg, és ne egy pályát magoljon be. Ezenkívül a magasságbeli változtatásokat sem tudjuk még megtanítani, érdemes lehet lejtőket és emelkedőket is betervezni a versenypálya bizonyos szakaszaiba. Ha készen van a környezet, akkor a tesztelése közben van értelme már fejleszteni, finomítani az architektúrán is.



6.1. ábra Négy ágens tanul egyszerre.

7 Irodalomjegyzék

[1] M. A. Nielsen, Neural Networks and Deep Learning, szerk., %1. kötet, , : Determination Press, 2015, p. .

[2] . . Siuly, Y. . Li és P. . Wen, „Clustering technique-based least square support vector machine for EEG signal classification,” *Computer Methods and Programs in Biomedicine,* %1. kötet104, %1. szám3, pp. 358-372, 2011.

[3] V. C. Raykar és P. . Agrawal, „Sequential crowdsourced labeling as an epsilon-greedy exploration in a Markov Decision Process,” , 2014. [Online]. Available: http://proceedings.mlr.press/v33/raykar14.pdf. [Hozzáférés dátuma: 31 5 2020].

[4] L. . Baird, „Residual algorithms: Reinforcement learning with function approximation,” *ICML,* %1. kötet, %1. szám, p. 30–37, 1995.

[5] „Reinforcement Learning / Successes of Reinforcement Learning,” , . [Online]. Available: http://umichrl.pbworks.com/Successes-of-Reinforcement-Learning/. [Hozzáférés dátuma: 31 5 2020].

[6] J. . Peters, S. . Vijayakumar és S. . Schaal, „Natural actor-critic,” *Lecture Notes in Computer Science,* %1. kötet, %1. szám, pp. 280-291, 2005.

[7] S. . Li, S. . Bing és S. . Yang, „Distributional Advantage Actor-Critic.,” *arXiv: Learning,* %1. kötet, %1. szám, p. , 2018.

[8] A. . Juliani, „Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C),” , . [Online]. Available: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2. [Hozzáférés dátuma: 31 5 2020].

[9] „GitHub: bulletphysics/bullet3 releases,” , . [Online]. Available: https://github.com/bulletphysics/bullet3/releases. [Hozzáférés dátuma: 31 5 2020].

[10] „Welcome to Colaboratory,” , . [Online]. Available: https://colab.research.google.com. [Hozzáférés dátuma: 22 5 2019].

[11] „Project Jupyter,” , . [Online]. Available: https://jupyter.org/. [Hozzáférés dátuma: 22 5 2019].

[12] „Parallel Programming and Computing Platform,” , . [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html. [Hozzáférés dátuma: 22 5 2019].

[13] „Download PyCharm,” , . [Online]. Available: https://www.jetbrains.com/pycharm/download/. [Hozzáférés dátuma: 22 5 2019].

[14] „About Python,” , . [Online]. Available: https://www.python.org/about. [Hozzáférés dátuma: 22 5 2019].

[15] N. . Ketkar, „Introduction to PyTorch,” , 2017. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-2766-4_12. [Hozzáférés dátuma: 22 5 2019].

[16] J. . Wang, X. . Peng és Y. . Qiao, „Cascade multi-head attention networks for action recognition,” *Computer Vision and Image Understanding,* %1. kötet, %1. szám, p. 102898, 2020.

[17] J. . Vig és Y. . Belinkov, „Analyzing the Structure of Attention in a Transformer Language Model,” *arXiv: Computation and Language,* %1. kötet, %1. szám, p. , 2019.

[18] M. D. Zeiler és R. Fergus, „Visualizing and Understanding Convolutional Networks,” 2014.. [Online]. Available: https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf.

[19] S. Hochreiter és J. Schmidhuber, „Long Short-Term Memory,” *Neural Computation,* %1. kötet9, %1. szám8, p. 1735–1780, 1997.

[20] S. . Bock, J. . Goppold és M. . Weiß, „An improvement of the convergence proof of the ADAM-Optimizer.,” *arXiv: Learning*, %1. kötet, %1. szám, p. , 2018.

[21] TensorFlow, „TensorFlow,” 2018.. [Online]. Available: https://www.tensorflow.org/tensorboard. [Hozzáférés dátuma: 2020.].