Hallgatói nyilatkozat

Alulírott **Szilágyi Krisztián Gergely**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 14.

...…………………………………………….

Gergely

# Bevezetés

# Irodalmi áttekintés

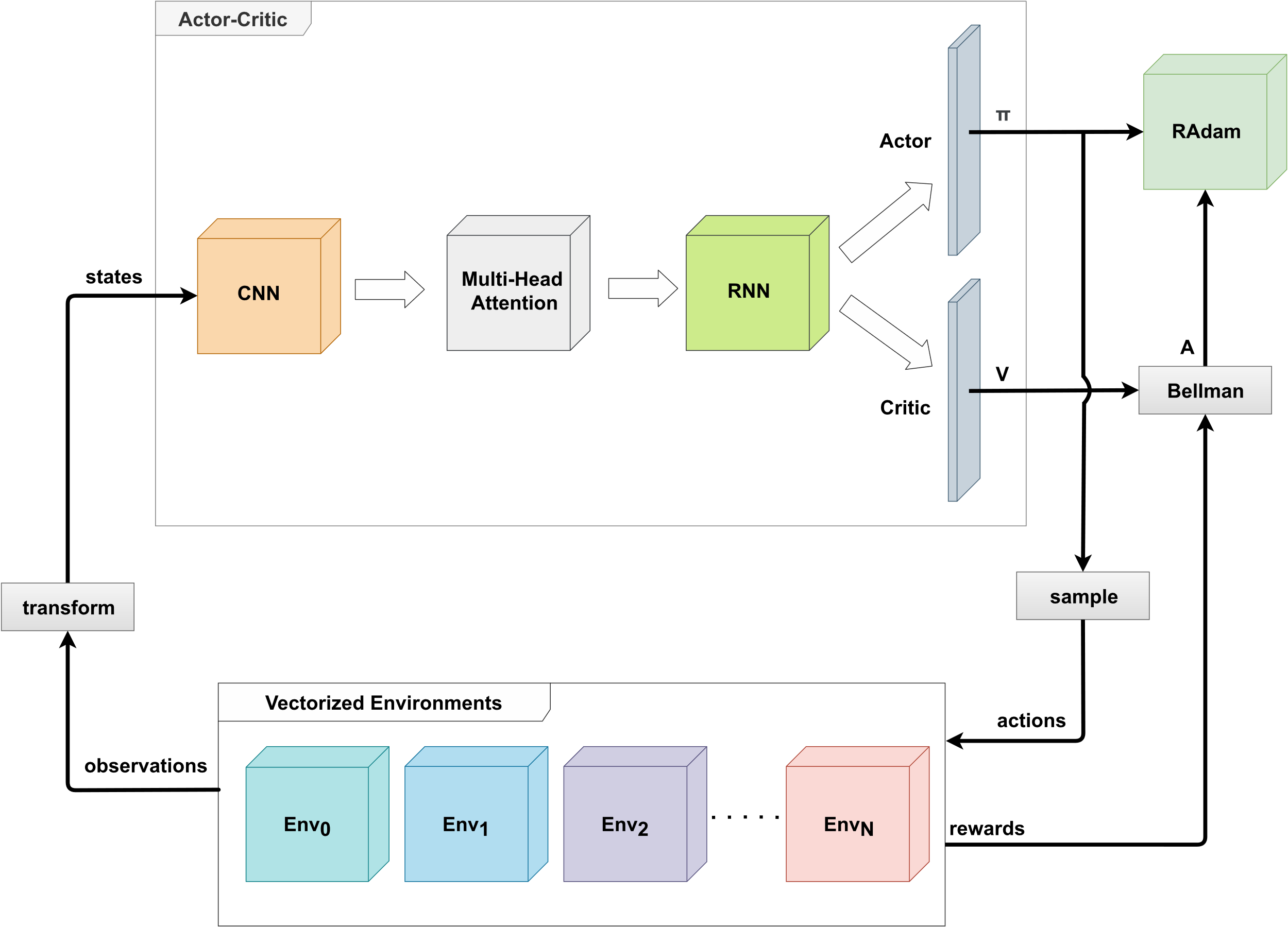
# Specifikáció, tervezés

# Megvalósítás

Ebben a fejezetben bemutatom, hogy a második fejezetben felvonultatott ismereteket, algoritmusokat miként alkalmaztam a gyakorlatban a feladatom elvégzéséhez. Előbb magát a megerősítéses tanulást megvalósító modell architektúráját részletezem, majd az általam használt szimulációs környezet felépítéséről és módosításairól írok. Végezetül az implementált jutalmazó függvényeket is bemutatom részletesen egyesével.

## Architektúra

Az alábbi képen látható a felépített Actor-Critic struktúra, melyet érdemes összevetni az MDP működésével (**2.8. ábra)**.



.. ábra A2C architektúra

### Felépítése

Az általam választott, megerősítéses tanulást megvalósító implementáció az A2C policy gradiens metódus lett. Az architektúra több kisebb logikai komponensre bontható: A rendszer magja az Actor-Critic ágens, melyet egy kétfejű mély neurális hálózattal valósítok meg. Az ágens tetszőleges algoritmussal megvalósítható, de érdemes olyat választani, melynek a paraméterei tanulhatóak és végig visszaterjeszthető a gradiens a modell bemenetéig. Az Actor fej kimenete a stratégia, vagyis az akciók eloszlása, melyből mintavételezünk akciókat, melyeket végrehajt az ágens az adott környezetekben. A Critic fej kimenete a becsült állapot-érték.

Az A2C alkalmazása miatt érdemes több környezetet futtatni egyszerre, amelynek csak a hardver erőforrások szabnak határt. Én párhuzamosan maximum négy környezetet futtatam a szimulációban. Az ágens bemenetéül a vektorizált környezetekből érkező megfigyelésekből képzett állapotokat kapja. Ezek RGB-D képekből álló csomagok, melyeken néhány transzformációt végre kell hajtani, mielőtt megkapja az Actor-Critic háló. Mivel a modell bemenetén megtalálható CNN rétegei NCHW () formátumú adatot várnak, ezért mint egyfajta batch-be össze kell csomagolni a környezetekből érkező képsorozatokat, így a batch mérete lesz.

A környezetektől visszakapjuk az akcióra számított jutalmat is, melyet a Critic fej kimenetén kapott állapot-értékkel együtt felhasználva a Bellman-egyenlettel előállítom az előny függvényt (2.15 egyenlet). Ez lényegében egy diszkontálást jelent, de ezenkívül itt egy kisebb trükköt is kell alkalmazni. Az előny mínusz egyszeresét kell venni (vagyis az állapot-értékből kell kivonni az akció-érték függvényt), mivel a jutalmat maximalizálni szeretnénk, ezért az optimalizáláskor nem minimumkeresést hajtunk végre, hanem a globális maximumot szeretnénk megtalálni.

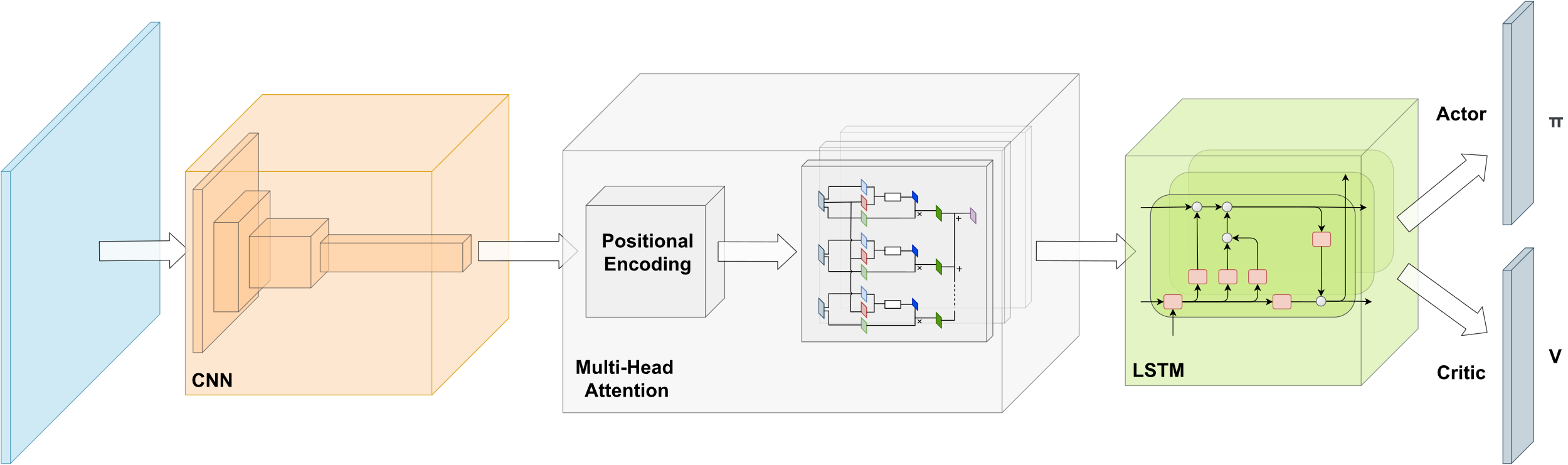
Az utolsó kiemelendő komponens a RAdam optimalizáló algoritmus, mely a háló jelenlegi paramétereit és a veszteségfüggvény gradienseit felhasználva számolja ki a háló új paramétereit a hiba-visszaterjesztés segítségével. A veszteségfüggvény három tag összegéből épül fel: Actor-tól kapott stratégiára, annak entrópiájára és a Critic fejre számolt költségből. Ezek minden esetben átlagértékek összegek helyett, mivel egy epizód idő előtt is befejeződhet, tehát nem biztos, hogy azonos hosszúságúak lesznek a batch-ek.

Az Actor költségét különbözőképpen kell számolni attól függően, hogy A2C-t vagy PPO-t alkalmazok, míg a másik két tag változatlan mindkét esetben. A Critic fej költsége az előnyre számított átlagos négyzetes hiba (*MSE* – *mean squared error*), mely arra ösztönzi a hálót, hogy minél pontosabban becsülje meg a jövőbeli diszkontált jutalmat. Az Actor kimenetén érkező nyers akció értékeket (*logits*) *softmax* segítségével konvertáljuk valószínűségi változókká. Az akciókat ebből a kategorikus eloszlásból mintavételezzük. Az eloszlás entrópiáját is hozzáadjuk a veszteségekhez regularizáció céljából: Az elfajuló (kis entrópiájú) a stratégiát elkerülendő ezt kivonjuk az összegből, viszont kis súllyal szerepel az egyenletben, hogy ne legyen egyenletes eloszlású (nagy entrópiájú) se a stratégia.

### Ágens

Az architektúra lelke, vagyis maga az ágens az Actor-Critic neurális hálózat. Ez a módszer alkalmas az optimális stratégia megtanulására.

A hálót alapvetően két egységre lehet bontani: Az első fele a bemenetén kapott állapot tenzort kódolja, vagyis a jellemzők kinyerésére szolgál (jellemző enkóder, *backbone*), míg a második fele lényegében a két fejet takarja, amelyek a kódolt jellemzőkből előállítják az optimális stratégiát és az állapot-érték függvényt. Az utóbbi komponens általában egyszerű, teljesen-összecsatolt osztályozó rétegeket tartalmaz. Diszkrét esetben az Actor fej egy (a lehetséges akciók száma) neuronokból álló osztályozó, míg a Critic fej egyetlen skalárt ad meg, az állapot-értéket. Ez a része a hálónak viszonylag egyértelműen adódik, azonban egyáltalán nem evidens, hogy a backbone része hogyan épül fel.



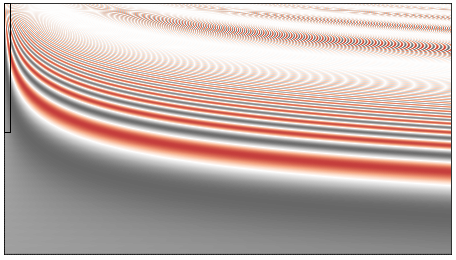
4.. ábra Actor-Critic modell

A backbone első komponense egy ortogonálisan inicializált CNN hálózat, mely a bemenetén kapott NCHW formátumú képek tenzorát egy (rejtett dimenzió) hosszú vektorba képezi le. A dimenziócsökkentést négy darab egymást követő leskálázó konvolúcióval [19] érem el, tehát pooling réteget nem alkalmazok. Az egyes konvolúciós rétegek aktivációs függvényeiként *Leaky ReLU* nemlinearitást alkalmazok, mely hasonló a *ReLU* függvényhez, de a negatív tartományban nem nullát rendel a bemenethez, hanem a kis meredekségű letörése miatt egy kicsi negatív számot.

A stabilabb és gyorsabb konvergencia érdekében ún. *LayerNorm*-ot alkalmazok a konvolúciók után. Az ismertebb *BatchNorm*-al szemben nem az egész batch-en keresztül normalizál csatornánkként, hanem minden egyes tenzort egyénileg kezelve az összes csatornáján keresztül normalizál. Mi esetünkben az egymáshoz nagyon közeli framek-en keresztül nem lenne értelme csatornánkként normalizálni.

Végül kicsomagolom a képeket, hogy ismét külön kezelhessük a különböző környezetektől kapott frameket. Ekkor a reprezentáció tenzorja méretű, ahol az első dimenzió a környezetek száma, a második a framek száma.

A következő két blokkot opcionálisan tartalmazza a modell, hogy különböző struktúrákat össze tudjak hasonlítani a tanítások során. Az első kikapcsolható blokk a Multi-Head Attention réteg [20], mely előtt szükségszerűen alkalmazom a bemutatott pozíció kódolási technikát. A bemeneti szekvencia hossza a képek száma (), hiszen a képek pozícióit szeretnénk kódolni, a környezetek mentén azonos lesz a kódolás (nyilvánvalóan nem szeretnénk különbséget tenni a független környezetek között). A rejtett dimenziók száma , mely egy-két nagyságrenddel nagyobb, mint a képek száma.



4.. ábra Pozíció kódolás, feltüntetve az általam használt kódok halmazát

A **4.3. ábrán** látható egy példa, melyen ábrázoltan, hogy hogyan alakulnak a kódok egy adott beállításnál, mely egy maximum 1920 hosszú és 1080 rejtett dimenziójú szekvenciát képes kódolni. Az így kapott Full HD képen már könyebben ki lehet venni az exponenciálisan nővő hullámhosszokat, a **2.12. ábrával** szemben. A bal felső sarokba rajzolt téglalap jelzi az általam választott dimenziók esetén a kódolási beállítást, ahol és a szekvencia hossza, azaz a képek száma: . Az arányok a jobb láthatóság érdekében enyhén módosításra kerültek.

A Multi-Head Attention megvalósításához a PyTorch implementációját használtam fel. A tanításokat egy kétfejű rétegen végeztem el, a paramétereket úgy állítottam be, hogy a bemenet dimenziói ne változzanak.

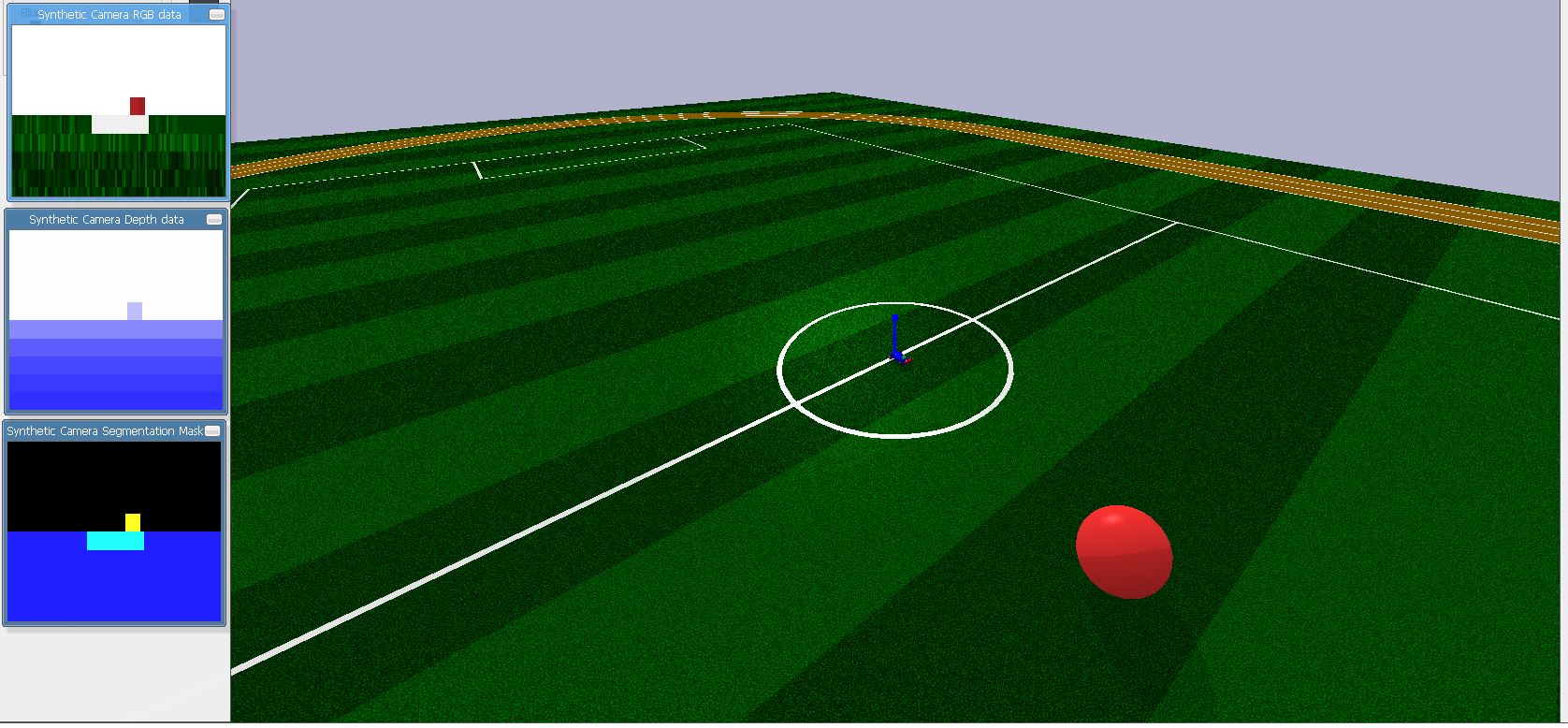
Az Attention blokkot követi opcionálisan egy LSTM cella (Long Short-Term Memory [21]). Az opcionalitás oly módon van korlátozva, hogy a Multi-Head Attention után mindenképp az LSTM réteg következzen. Összeségében tehát három módban futtatható az ágens: a jellemző enkóder csak a CNN blokkból, vagy CNN és LSTM blokkokból áll, vagy az összes blokk be van kapcsolva. Mivel az LSTM - és emiatt az enkóder - kimenetén lévő lineáris osztályozók is egy a környezetenként kódolt jellemző vektort vár (az LSTM felépítéséből adódóan nem változtat a dimenziókon), ezért még egy transzformációra szükség van. Mivel a képek menti változás információját is a vektorba kell tömöríteni ezért Attention használata esetén a képek mentén átlagolok, míg a többi esetben egyszerűen csak kiválasztom az utolsó kapott frame reprezentációját és azt adom bemenetként az LSTM cellának vagy már közvetlenül a lineáris rétegeknek. Így a backbone kimenete egy méretű mátrix lesz.

Itt legyen egy 4.2-höz hasonló kép a tenzorok dimenzióval kiegészítve?

## Szimulációs környezet

Célhardveren tanítani, illetve ezzel párhuzamosan tesztelni a hálót egyrészt lassú lenne, mert felesleges overhead-et okozna az integráció, hiszen mindig jelen kéne lenni a laborban. Másrészt igencsak költséges is lehet, hiszen, egy komolyabb hiba vagy rossz döntés miatt kárt tehet magában és a környezetében is a versenyautó. Ezért célszerű a szoftver működését egy pontos fizikai motorral szimulált részletes környezetben tesztelni, ahol az említett a hátrányok természetesen nem jelentkeznek.

A kitűzött feladat elvégzéséhez a PyBullet fejlesztői kettő hasonló környezetet építettek már ki. Mindkettőben az ágens egy kis távirányítós versenyautó, amelynek el kell jutnia 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 (), addig a másiknál a teljes RGB-D kamera kimenete. Az utóbbi környezetet fejlesztettem tovább, jelenleg már teljesen új elemekből épül fel, kijavítottam és módosítottam a szükséges függvényeket, valamint új funkciókat hoztam létre és új jutalmazó függvényt implementáltam, melyet később részletesen bemutatok.



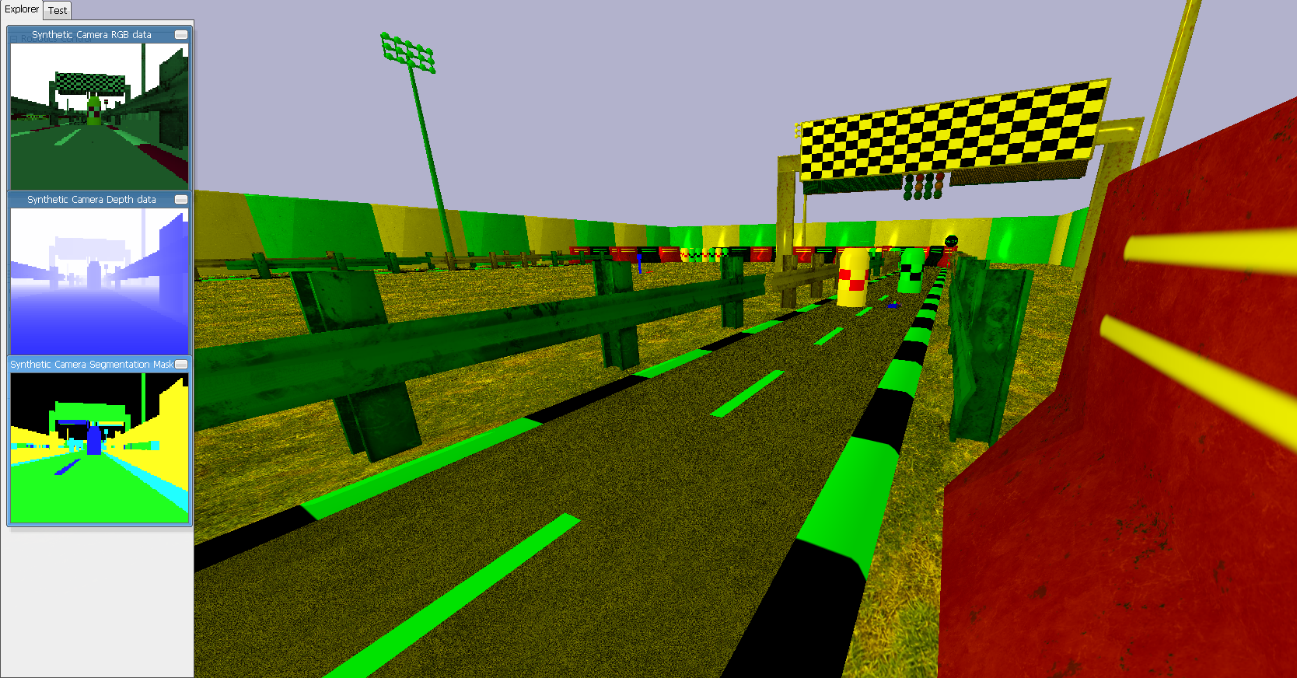
.. ábra Az eredeti PyBullet-es környezet, bal oldalt a kamera képe látható (felülről lefelé: RGB, mélység, szegmentált)

### Objektumok beolvasása

A PyBullet egyik legnagyobb hátránya, hogy meglehetősen limitáltan képes beolvasni az objektumokat és elhelyezni azokat a környezetben. Alapvetően két fájl típust képes kezelni, az ún. *URDF* és *SDF* fájlokat, ezek mind *XML* alapúak. Az előbbi a bonyolultabb objektumokat írja le, mozgatható egységekkel, csuklókkal stb. Tipikusan a robotok leírását szolgálja, alapvetően ezek lesznek az ágensek a szimulációban, míg az utóbbi az egyszerűbb tereptárgyak jellemzőit írja le. Ezek olyan tulajdonságokat tartalmaznak, mint például a tömeg, súrlódási együttható, pozíció és orientáció, szín, anyagjellemzők stb.

A PyBullet esetében csak kevés ilyen előre elkészített modell volt számomra hasznos, így csak a versenyautó *URDF* modelljét használtam fel. A többi objektumot kezdetben kézzel próbáltam elkészíteni az *XML* fájlok szerkesztésével, melyekbe ingyenesen beszerzett *OBJ* fájlokat linkeltem. Mivel rengeteg időt vett volna igénybe az összes szükséges tereptárgyat ily módon elkészíteni, ezért alternatív megoldások után néztem. Végül egy időközben implementált osztály lett a segítségemre, melyben található egy hasznos metódus, mely *WORLD* típusú *XML* fájlokat képes feldolgozni. Ez egy tereptárgyakat (*SDF*) egybefoglaló *XML*, melyben minden objektum megjelenik. A *WORLD*, valamint a korábban említett két fájl típus is a Gazebo 3D robot szimulátor szoftver segítségével generálható.

Ezáltal 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 számomra szimpatikus pályát, melyet kifejezetten a PyBullet-hez készítettek el. Sok tekintetben hasonlít az elképzeléseinkhez, és egyelőre tökéletesen megfelelt a szimulációhoz. Sokféle objektum található ezen a versenypályán: különböző falak és korlátok, lámpák, sávok, bóják stb. Egyetlen komolyabb hátránya, hogy maga a versenypálya egy fix objektum, és nem több részből áll, ezért jelenleg csakis futópálya alakú versenypálya érhető el.



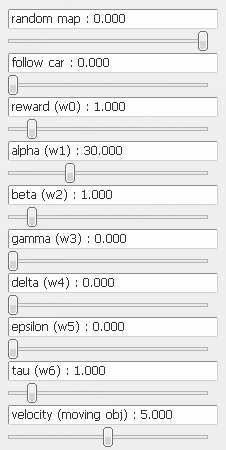
.. ábra Az elkészült pálya, a hibás színkezeléssel együtt

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 helyezem el az epizódok elején. A többi pálya elem jelenleg még fix pozícióval generálódik a környezetbe. Ezeken kívül kiemelendő objektum még a célvonal, valamint egy közlekedési lámpa is, melyeket felhasználok a jutalom számításához. A pályán található még egy Stop tábla is, de ezt és egyéb táblákat jelenleg még nem használok fel az autó tanításához.

Itt megemlíteném, a PyBullet legnehezebben javítható bug-ját: A *WORLD* és *SDF* fájlok beolvasása nem tökéletes, egyelőre ismeretlen okból az anyagjellemzők beolvasása hibás. 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.

### UI

A PyBullet egyik hasznos tulajdonsága, hogy a kezelőfelületén gombokat és csúszkákat helyezhetünk el, melyekkel bármilyen paramétert változtathatunk. Sajnos a gomb implementálása kissé hibásra sikerült, így a gomb helyett is csúszka jelenik meg (például a **4.6. ábra** látható felső kettő csúszka igazándiból gombok). A felső „csúszka” 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 a kocsi fölött fixálja a GUI ablakát, így könnyedén képesek lehetünk követni a kocsit a pályán. Kikapcsolva ezt a funkciót szabadon nézelődhetünk a szimulált környezetben. Az ez alatt megtalálható további 7 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.

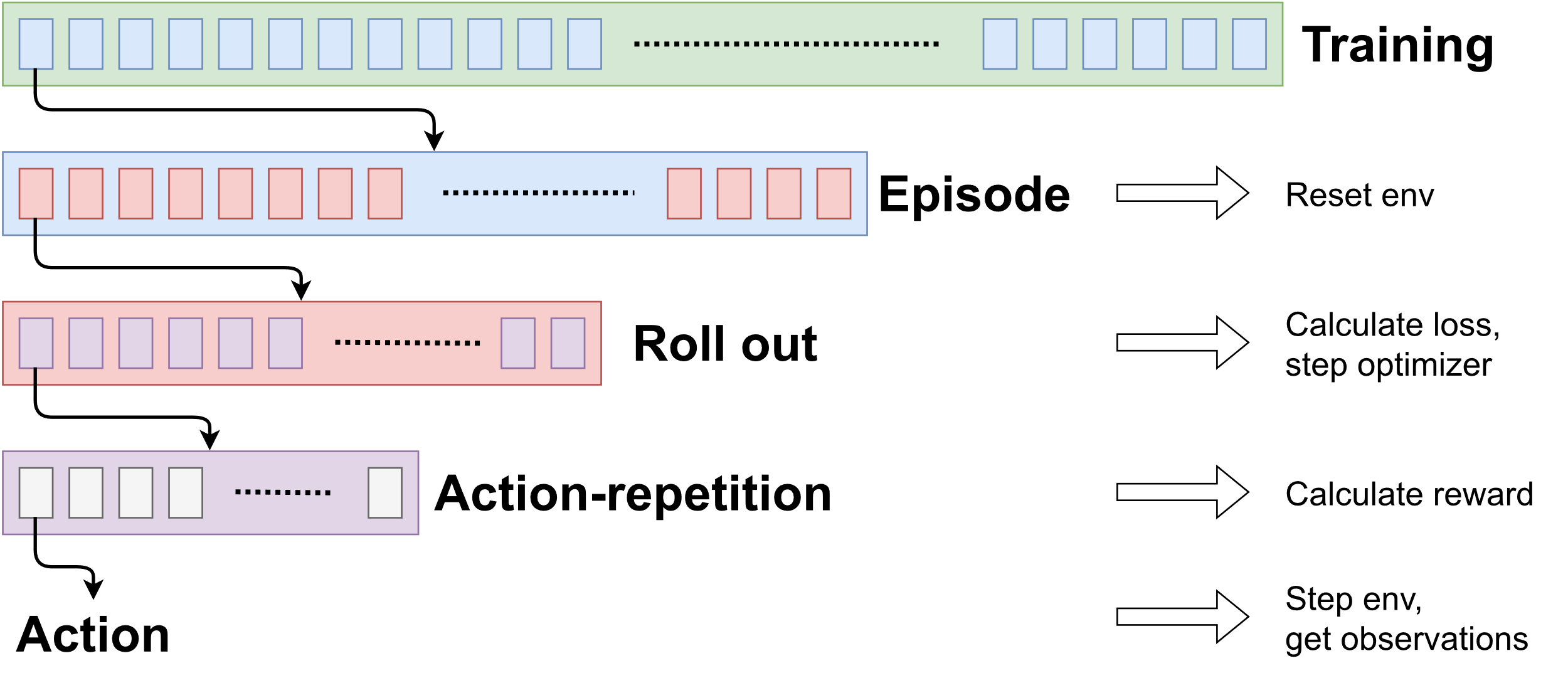


4.. ábra Gombok és csúszkák a GUI felületén

## Tanítás

A tanítás folyamata igencsak zajos és hosszadalmas lenne, ha minden állapotban egyetlen akciót vennénk csak figyelembe a költség számolásánál, ezért érdemes több akciót megvárni a modell paramétereinek frissítéséhez. Ennek a folyamatnak a megnevezése az ún. *roll-out*, mellyel megadhatjuk, hogy hány lépést (állapot-akció párt) várjunk meg, mielőtt kiszámoljuk költségfüggvény eredményét. Ez felfogható úgy, mint a felügyelt tanulásnál alkalmazott batch fogalma. A roll-out ennél fontosabb előnye, hogy így nem szükséges a teljes epizód trajektóriáját eltárolni a memóriában, ami az A2C esetében igen nagy probléma lenne, mivel több ágenst is futtatok egyszerre, ráadásul komplexebb feladatnál nagyon hosszadalmasak lehetnek az epizódok. Viszont ezesetben nincs meg még a végső jutalom, tehát nem tudjuk a jutalmakat diszkontálni, ezért egy trükköt kell alkalmazni: a roll-out utolsó állapotára a Critic fej által számolt érték, vagyis az utolsó állapotban predikált végső jutalmat diszkontáljuk. Szokásosan kis számú akciót szokás megvárni, például öt-hatot.

Egy tanítást epizódokra bontunk (hasonlóan az epoch-hoz), az epizód végén újraindítjuk a környezetet és ismét a nulladik állapotból indul az ágens. Egy epizód roll-out-okból áll, viszont egy roll-out-ba összefoghatunk több azonos akciót. Ennek a célja jelen esetben az, hogy összegyűjtsünk több megfigyelést is (framet) a környezetből és így a hálónak egyszerre több információt adunk át. Ezenkívül kevésbé lassítjuk így a szimulációt, mivel nem akciónkként értékeljük ki az állapotot.



.. ábra A tanítás felbontása a különböző részegységekre

Nézzünk egy példát: egy tanítást 100 epizódra végzünk el és 1 epizód fix 80 roll-out-ból áll (ha nem történhet olyan esemény a környezetben, mely esetén hamarabb is befejeződhet egy epizód). Egy roll-out 5 lépésből álljon, valamint 1 akciót ötször ismételjünk meg a szimulációban. Ekkor a tanítás során az ágens 200,000 akciót végez, ha 4 környezetet párhuzamosítunk, akkor máris 800,000 akcióról beszélünk. A modell paraméterei 8,000-szer frissültek.

A háló kimentését a jutalom alapján végzem: A legjobban jutalmazott modell paramétereit mentem ki. A modell viszonylag kicsi, néhány Mbyte tárhelyet foglal, így nem kellett a háló tömörítésével foglalkoznom (lásd *pruning* vagy *weight sharing*). A tanítás komoly számításigényén látszik, hogy még ez a nem túl komplex modell esetén is a fenti példát véve a 100 epizód 11 órát venne igénybe rendereléssel, melyet optimalizációkkal sikerült lecsökkenteni nagyjából 3 órára.

Imitation Learning használatát végül nem vettem igénybe. A közvetlen stratégiát tanuló módszerek (BC és DPL), valamint az IRL model-given megközelítése a tulajdonságaik és hátrányaik miatt a komplex környezetben nyilvánvalóan nem használhatóak. A modell nélküli eset tanítása viszont rendkívül sok időt vett volna valószínűleg igénybe, így végül ezzel sem próbálkoztam.

## Jutalom függvény

A másik esszenciális feladat a környezet elkészítése mellett a jutalmazó algoritmus megalkotása, mely eldönti, hogy az adott akcióra mekkora jutalmat ad. Ez kulcsfontosságú lépés, hiszen itt sok elvi hibát lehet ejteni, könnyedén előfordulhat a kobra effektus jelenség, azaz, hogy azt hisszük egy megoldási javaslat tényleg megoldja a problémát/feladatot, de igazából csak még inkább rontunk rajta. Emiatt sok időt kell a tesztelésre szánni, nehogy az ágensünk furcsa vagy haszontalan dolgot tanuljon meg: Például ne vágja le az utat egy kanyar helyett, vagy ne tolatva jusson el a célba stb. Összesen hat féle jutalmazást implementáltam, melyek segítségével az ágens az önvezetés különböző aspektusait sajátíthatja el. Ezek nagyon alapvető képességeket fednek le, így a későbbiekben további jutalmazásokra is szükség lehet, például a közlekedési táblákat nem veszem figyelembe.

Az alábbi hat jutalomból csak az első kettő ( és ) tekinthető folytonosnak, a többi diszkrétnek, vagyis ritkának. A ritka jutalmak, habár jó eredményre visznek minket, lassabb konvergenciát okoznak, mivel az ágens ritkábban kap visszacsatolást a döntései után. A végső jutalmat a hat részeredmény súlyozott összegéből kapjuk meg:

|  |  |  |
| --- | --- | --- |
|  |  | (4.1) |

### Alfa

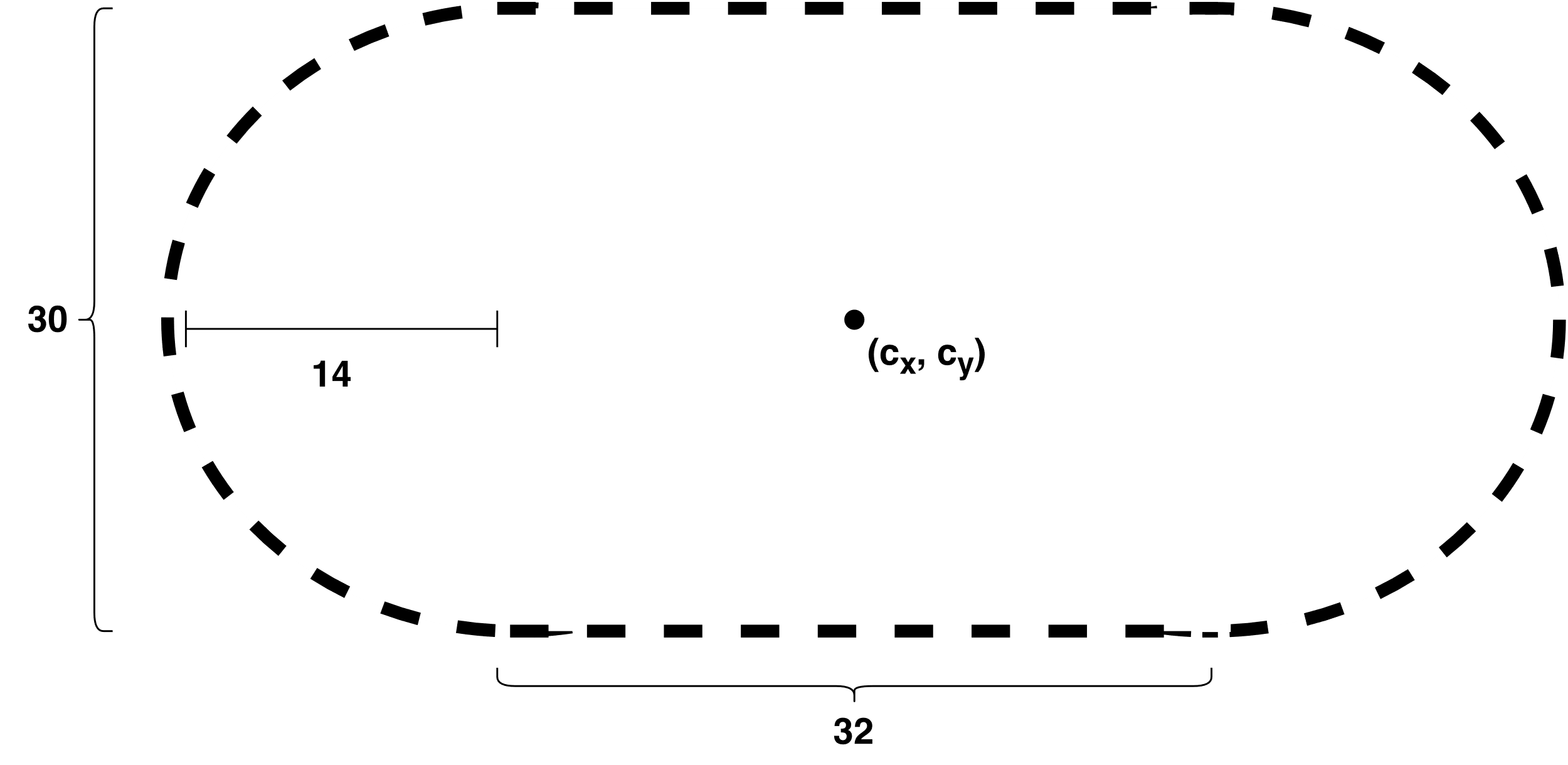
Ez az egyik legegyszerűbb jutalmazó algoritmus mind közül. Célja, hogy eljuttassa az ágenst a kijelölt célba (időtől függetlenül) egyenesvonalban. Folytonosnak tekinthető az alfa jutalmazás, tehát minden akció-ismétlés után kiértékeljük a függvényt. A jutalom annak a függvényében pozitív, hogy az előző állapothoz képest közelebb jutott-e a célhoz vagy sem. Ha nőtt a távolság, azaz a céltól elfelé mozdult, akkor negatív a jutalom:

|  |  |  |
| --- | --- | --- |
|  |  | (4.2) |

Ezenkívül még egy funkciót ellát az alfa jutalmazás. A tesztelések során kiderült, hogy nagyon könnyen elakad az ágens a pálya szélén, akár már az epizód elején. Mivel egyébként is hosszadalmas a tanítás, ezért bevezettem egy számlálót, mellyel nyomon követjük, hogy hány akción keresztül nem mozdult lényegesen arrébb az ágens az előző pozíciójához képest. Ha ez elért egy küszöbértéket, az azt jelenti, hogy a kocsi elakadt és ekkor újraindul a környezet és kezdődik a következő epizód. Mivel szeretnénk elkerülni, hogy a kocsi fennakadjon (és erre a később részletezett tau jutalom kevés), ezért egy nagy negatív értékre állítom az alfát ezesetben.

### Béta

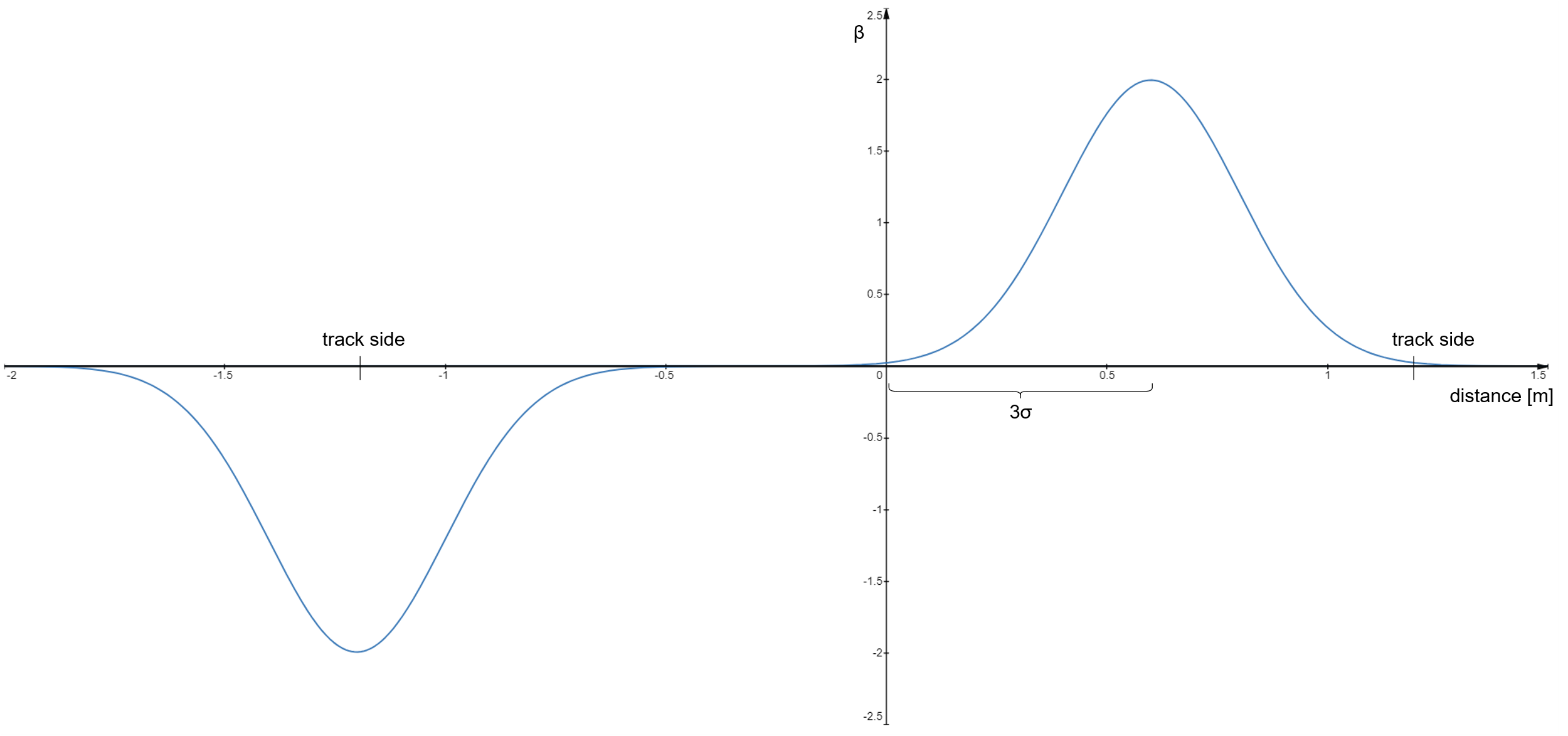
A béta viszont a legkomplexebb algoritmus az implementáltak közül. Ez az algoritmus a sávtartásért felel, azaz a célja, hogy az ágens tanulja meg, hogy ne menjen át a szembe sávba és ne menjen le az útról sem. Nehézsége abból adódott, hogy a sáv egy fix objektum a környezetben, melynek egyetlen attribútuma van, a középpontjának koordinátái ( és ). Ennyire kevés információval körülményesebb a távolságokat számolni. Mivel nem találtam leírást a pálya paramétereiről, így lemértem és függvényt illesztettem rá. Úgy kezelem a szaggatott vonalból képzett pályát, mintha egy téglalapból állna, 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 (**4.8. ábra**). Ezeket az adatokat felhasználva a szimuláció koordináta-rendszerében már viszonylag pontosan lehet becsülni az autó helyzetét a sávokon belül.



4.. ábra A szaggatott vonal méretei

Miután megbecsültem a pozíciót, már csak a pontozás mértékét kell beállítani. Ehhez Gauss-görbét használok, melynek a maximuma a jobb oldali sáv közepére van állítva, míg a koordináta-rendszer origója a szaggatott sávon helyezkedik el (**4.9. ábra**). A görbe szélességét úgy állítottam be, hogy a távolság a sáv széleire essen. Így lényegében ezen az intervallumon kívül balra, vagyis a szembe sávban nincs büntetés az úttest széléig, mivel a Gauss-görbe közel nulla értéket vesz fel. Azonban az úttest széleitől kifelé súlyosan büntetünk, innentől egy nagy abszolút értékű negatív számra (pl.: -10) állítom a bétát. Viszont a szembe sávban való haladást is büntetni kell, tehát arra a sávra egy az abszcisszára tükrözött, ugyanakkora szórású Gauss-görbét illesztettem. A középpontját, azaz minimumát a pálya szélén éri el. Önkényesen választottam meg az áttérés helyét is az egyik görbéről a másikra, úgy, hogy az átmenetben minimális ugrás legyen:

|  |  |  |
| --- | --- | --- |
|  |  | (4.3) |



4.. ábra Gauss-görbe elhelyezkedése a pályán

### Gamma

A következő jutalmazás célja, hogy a kocsi megálljon a piros lámpánál. Az algoritmus állapotgép-szerűen működik, bizonyos időközönként vált a lámpa piros és zöld között (sárgával nem tartottam szükségesnek foglalkozni). A váltás grafikusan nem jelenik meg a környezetben, az objektum változtatásokat megvalósítani a PyBullet-ben sajnos nem triviális.

Ha zöld a lámpa a gamma értéke nulla. Ha piros, akkor megvizsgáljuk az ágens távolságát a lámpától (csak a síkban), és ha 2 egységen (méter) belül van, akkor a cél, hogy álljon meg a kocsi, vagyis csökkentse le a sebességét nullára. Ezért a gamma értékét a sebesség függvényében választom meg, jelen eseteben egyszerűen a sebesség mínusz egyszerese. Ha a kocsi már fél méterre is megközelítette a lámpát, miközben az piros, akkor a gamma szintén egy nagy abszolútértékű negatív konstans értéket vesz fel:

|  |  |  |
| --- | --- | --- |
|  |  | (4.4) |

### Delta

Biztonsági szempontból az egyik legfontosabb képesség, hogy az ágens ki tudja kerülni az útjába kerülő objektumokat. A delta jutalom, annál nagyobb, minél kevésbé közelít meg az ágens egy álló objektumot. A legközelebbi objektum távolságát vetjük össze az ágens távolságával. Ha ez a távolság kisebb, mint egy küszöbérték (pl.: ), akkor közelinek is tekintem ezt az objektumot. Ezenkívül ellenőrzöm, hogy a tárgy benne van-e az ágens 30°-os látóterében, és hogy felé halad-e a kocsi.

Ha mindhárom feltétel teljesül, akkor a távolság függvényében büntetjük az ágens akcióit. A függvény alakja ezesetben egy x-tengelyre tükrözött Gauss-görbe. A görbe a kocsi és a legközelebbi álló objektum közti egyenesre illeszkedik, mégpedig oly módon, hogy az origó egybe esik az objektum középpontjával. A cél, hogy az ágens általunk választott sugarú körön kívül kerülje el az objektumot, így a körön belül büntetünk, azon kívül pedig nincs jutalmazás.

A görbe szélességét azért választottam 7-nek, mert így a értékét 0 és 100 között változtatva sem lesz mérhető ugrás méternél:

|  |  |  |
| --- | --- | --- |
|  |  | (4.5) |

### Epszilon

Nemcsak az álló, hanem a mozgó járműveket is célszerű lenne elkerülnie az ágensnek, erre szolgál az epszilon algoritmus. A mozgó objektum a GUI-ban változtatható sebeséggel mozog két fix pont között oda és vissza. Jelenleg egy ilyen objektum található a pályán, mely az úttest két széle között ingázik a célvonalnál. Hasonlóan a deltához, ha az ágens túlságosan megközelíti, akkor negatív jutalmat adunk:

|  |  |  |
| --- | --- | --- |
|  |  | (4.6) |

### Tau

Még egy fontos tényezőt nem vesz eddig figyelembe az ágens: az időt. Épp ezért szükséges bevezetni egy olyan jutalmazó függvényt is, mely arra sarkallja az ágenst, hogy ne csak egyszerűen eljusson a célba, hanem ezt minél gyorsabban tegye meg. Logikailag a függvényt úgy érdemes felépíteni, hogy legyen pozitív a jutalom, ha az epizód befejezte előtt eljutott a kocsi a célba és legyen nagy negatív a jutalom, ha nem. Minél hamarabb fejezi be az epizódot, annál nagyobb a jutalom. Látható, hogy nehézkes tesztelni a függvény hatását, hiszen epizódonként csak egyszer fut le az algoritmus.

A függvény alakját hiperbolikusnak választottam meg (lásd az alábbi képletet, ahol *T* egy epizód maximális lehetséges ideje). Fontos megjegyezni, hogy ezek igazából nem idő dimenziójú mértékek, hanem a pontosság kedvéért akciók számában vannak mérve. Ezért nem is fordulhat elő, hogy egy nagyon kicsi számmal való osztás miatt elszállna a jutalom, hiszen a legelső jutalom számolása egy roll-out mennyiségű akció után történik, ezért a numerikus stabilitásra nem kell figyelni (valamint nem is reális, hogy az ágens elér a célba az első akciók alatt).

|  |  |  |
| --- | --- | --- |
|  |  | (4.7) |