Általános információk

A diplomaterv szerkezete:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletesés pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő Diplomaterv sablon dokumentum tartalma. Ügyeljen a konzulens nevét és a beadás évét jelölő szövegdobozokra, mert azokra külön ki kell adni a frissítést. A mezők tartalma a sablonban a dokumentum adatlapja alapján automatikusan kerül kitöltésre.

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2.5cm, baloldalon 1cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel.

Minden oldalon - az első négy szerkezeti elem kivételével - szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2 táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le.

Az irodalomjegyzék szövegközi hivatkozása történhet a Harvard-rendszerben (a szerző és az évszám megadásával) vagy sorszámozva. A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben. Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóirat címeket csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internet hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

* a szakdolgozat készítő/diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás Karunkon, ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
* mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozat készítést ill. diplomatervezést kívánunk!

FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Fehér Gergő

Versenyautó pályaívének tervezése mesterséges intelligencia módszerekkel

Konzulens

BUDAPEST, 2017

Tartalomjegyzék

[Összefoglaló 7](#_Toc498604487)

[Abstract 8](#_Toc498604488)

[1 Bevezetés 9](#_Toc498604489)

[2 Elméleti háttér 10](#_Toc498604490)

[2.1 Megerősítéses tanulás 10](#_Toc498604491)

[2.1.1 Markovi döntési folyamat 10](#_Toc498604492)

[2.1.2 Algoritmusok kategorizálása 11](#_Toc498604493)

[2.1.3 Exploration exploitation probléma 13](#_Toc498604494)

[2.1.4 Gyakori felfedező függvények 14](#_Toc498604495)

[2.2 Neurális hálózatok 14](#_Toc498604496)

[2.2.1 Neurális hálózatok felépítése 15](#_Toc498604497)

[2.2.2 Aktivációs függvény szerepe és főbb típusai 16](#_Toc498604498)

[2.2.3 Neurális hálózatok alkalmazásai 18](#_Toc498604499)

[2.2.4 Költség függvények 18](#_Toc498604500)

[2.2.5 Hálózat tanítása 19](#_Toc498604501)

[2.2.6 Neurális hálózatok típusai 19](#_Toc498604502)

[2.3 Deep reinfocement learning 20](#_Toc498604503)

[2.3.1 Motiváció 20](#_Toc498604504)

[2.3.2 DQN 20](#_Toc498604505)

[2.3.3 DDQN 21](#_Toc498604506)

[2.3.4 DDPG 21](#_Toc498604507)

[2.3.5 Monte Carlo Tree Search 22](#_Toc498604508)

[3 Megerősítéses tanulás a gyakorlatban 24](#_Toc498604509)

[3.1 Kő-papír-olló megerősítéses tanulással 24](#_Toc498604510)

[3.2 Kő papír-olló neurális hálóval 24](#_Toc498604511)

[3.3 Értékelés 24](#_Toc498604512)

[4 Versenyautó irányításának megvalósítása 26](#_Toc498604513)

[4.1 Szimulátor 26](#_Toc498604514)

[4.1.1 Grafikus felület 26](#_Toc498604515)

[4.1.2 Szimulátor felépítése 28](#_Toc498604516)

[4.2 Kinematikai modell 29](#_Toc498604517)

[4.3 Probléma nehézségei 30](#_Toc498604518)

[4.4 Elméleti megfontolások 31](#_Toc498604519)

[4.4.1 Bemenetek 31](#_Toc498604520)

[4.4.2 A jutalom függvény 31](#_Toc498604521)

[4.4.3 Alkalmazott exploration function 32](#_Toc498604522)

[4.4.4 Hálózat szerkezete 32](#_Toc498604523)

[4.5 Implementált algoritmusok 32](#_Toc498604524)

[4.6 Felmerült akadályok 33](#_Toc498604525)

[4.6.1 Optimalizálás 33](#_Toc498604526)

[4.6.2 Teljesítmény 33](#_Toc498604527)

[4.6.3 Hyperparaméter optimalizáció 33](#_Toc498604528)

[4.6.4 Exploration function optimalizáció 33](#_Toc498604529)

[4.6.5 Experience replay mintavételezése 33](#_Toc498604530)

[4.7 Eredmények értékelése 33](#_Toc498604531)

[4.8 Lehetséges tovább fejlesztések 33](#_Toc498604532)

[4.9 Formázási tudnivalók 33](#_Toc498604533)

[4.9.1 Címsorok 33](#_Toc498604534)

[4.9.2 Képek 33](#_Toc498604535)

[4.9.3 Kódrészletek 34](#_Toc498604536)

[4.9.4 Irodalomjegyzék 34](#_Toc498604537)

[5 Összefoglaló 35](#_Toc498604538)

[6 Utolsó simítások 36](#_Toc498604539)

[Irodalomjegyzék 37](#_Toc498604540)

[Függelék 38](#_Toc498604541)

Hallgatói nyilatkozat

Alulírott **Fehér Gergő**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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, 2017. 11. 04.

...…………………………………………….

Fehér Gergő

Összefoglaló

Az utóbbi években a számítási kapacitás növekedésével egyre nagyobb teret nyernek gépi tanuló algoritmusok és az ún. mély tanulás, amivel a kutatók a műszaki problémák egyre több területén tudnak eddig még soha nem látott sikereket elérni. Ezek az algoritmusok ma már igen összetett problémákat is meg tudnak oldani emberi vagy sok esetben akár annál jobb szinten is, mint például a képfelismerés vagy akár olyan komplex játékokkal játszani, mint a Go.

Jelen szakdolgozat témája, ezen algoritmusoknak az irányítástechnikában való lehetséges alkalmazása, azon belül egy versenyautómodell irányításnak elméleti megvalósítása. Ennek megvalósításához a mély neurális hálózatokat, illetve megerősítéses tanuláson alapuló algoritmusokat használtam fel.

A szakdolgozat során először egy egyszerűbb példán összehasonlítottam Matlabban és Pythonban az algoritmus implementáláshoz szükséges eszköztárakat. Ezután implementálásra került a feladat megvalósításához használt keretrendszer, ahol grafikus felületen tudjuk beállítani a környezet, az autó és a vezérlő algoritmus paramétereit, illetve nyomon tudjuk követi az algoritmus tanulásának előre haladását is.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# Bevezetés

Manapság talán az egyik legfelkapottabb kutatási területnek a mesterséges intelligencia kutatása számít. Bár az alap ötletek és algoritmusok már régóta léteznek, sokáig hiányzott a tudósok mögül a számítási kapacitás, így a kezdeti lendület hamar alább hagyott. Viszont az informatika fejlődésével, a modern több magos CPU-k és a GPU-k számítási kapacitásának köszönhetően a kétezres évek elején új lökést kapott a terület. Bár amit ma mesterséges intelligenciának hívunk, az még nagyon messze áll attól, a sci-fi-kben látunk, viszont bizonyos területekre, problémákra fokuszáltan a modern algoritmusok már sokszor túlszárnyalják az ember teljesítményét is. Legyen az képeken objektumok esetleg emberek felismerése vagy olyan bonyolult játékokkal való játék, mint a Go. Mindenesetre a kifejlesztett algoritmusokat egyre több probléma területen lehet sikerrel alkalmazni. A dolgozatomban én egy irányítástechnikai probléma alternatív megoldását tűztem ki célul a később részletezett módszerekkel segítségével. A feladatom egy versenyautó modell irányítása a gépi tanulás egy válfajának a megerősítéses tanulás (angolul Reinforcement learning) segítségével olyan formában, hogy az az általa elképzelt szuboptimális íven haladjon végig a pályán, anélkül, hogy közben a pálya szélének vagy az esetlegesen a pályán elhelyezett akadályoknak ütközzön, mindezt minél rövidebb idő alatt. Természetesen erre léteznek más gépi tanulást nem alkalmazó algoritmusok is, viszont a feladatot egy kicsit megváltoztatva ezek már részletesebb tervezést és nagyon komplex algoritmusokat eredményeznének.

A dolgozatom három fő részből áll. Az első részben a megoldandó feladathoz felhasznált megerősítéses tanulás és a mély tanulás (angolul Deep learning) elméleti háttere kerül bemutatásra, kitérve az implementált algoritmusokra is. A második részben egy egyszerűbb gyakorlati példán kerülnek bemutatásra az alkalmazott módszerek, a megerősítéses tanulás és a mély tanulás, továbbá a szimulátor implementálásához használt programnyelv választása mögötti megfontolások. A harmadik részben ismertetem a megoldandó feladat megvalósítását. A szimulációk és a tanítás futtatásához implementált keretrendszert, a környezet és az autó fizikai modelljét, az alkalmazott algoritmusokat és az azokkal elért eredményeket.

Mivel a témakörben kevés a magyar szakirodalom, ezért a dolgozatomban gyakran fogom az angol szakkifejezéseket használni.

# Elméleti háttér

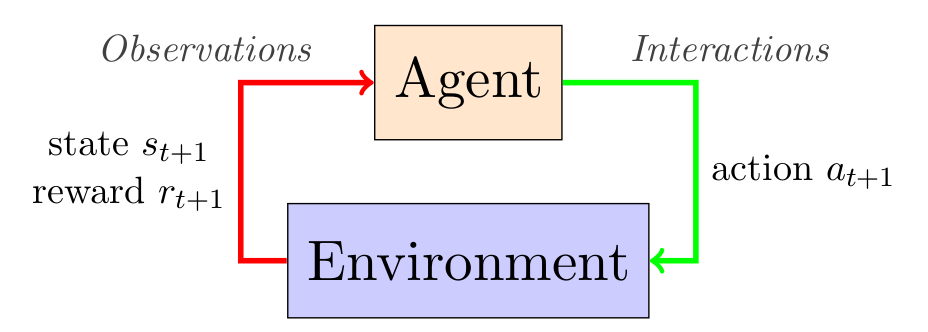
## Megerősítéses tanulás

A megerősítéses tanulás a gépi tanulásnak egy olyan válfaja, melyet a viselkedés pszichológia ihletett. Arra a problémára keresi a megoldást, hogy egy ágens az adott környezetben milyen döntéseket hozzon annak érdekében, hogy az azért kapott jövőbeli kumulált jutalmat maximalizálja.

### Markovi döntési folyamat

A megerősítéses tanulás témakörébe eső problémák gyakran modellezhetők az ún. markovi döntési folyamattal (Markovian Decision Process, továbbiakban MDP). Ez egy döntéshozatalok modellezésére létrehozott matematikai modell, melynek létezik determinisztikus és sztochasztikus változata is. Mi a továbbiakban csak a determinisztikus esettel foglalkozunk. Ez négy fő elemből áll. Az adott környezetben lévő állapotunk S, az S állapotban hozott döntésünk Az ezért kapott jutalom R és ennek hatására az új állapot S’.

Ez azt jelenti, hogy bármelyik állapot átmenetre, illetve a jutalom függvényre nem lehet hatással semmilyen múltbeli esemény, kizárólag a jelenlegi állapot és az abban hozott döntés.



1. Markovi döntési folyamat

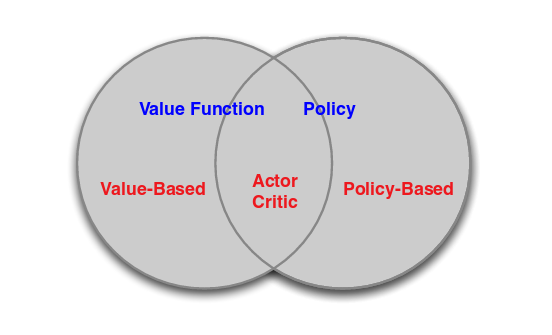
Az ágensnek van egy π(s) stratégiája ( továbbiakban policy) amelyet követve minden S állapothoz hozzárendel egy A döntést (továbbiakban action). Az A döntést meghozva kap a környezettől egy R jutalmat és az S’ következő állapotba jut. Az állapot átmenetért járó jutalom és a következő állapot általában előre nem ismert, ezeket az ágens valamilyen felfedező függvényt (továbbiakban exploration function) követve kell feltérképeznie. Ebben a kontextusban minden állapothoz definiálhatjuk a *V(s)* állapot érték függvényt (angolul state value), ami azt mondja meg, hogy az adott *S* állapotból az π(s) stratégiát követve mennyi a várható diszkontált kumulált jutalom, amit el fog érni.

Ahogy a való életben, az ágensünk számra is kevésbé értékes a jövőbeni jutalom, így azok értékét a számításkor egy γ értékkel diszkontáljuk. Definiálhatjuk, továbbá a Q(s, a) döntés érték függvényt (angolul action value function), ami azt adja meg, hogy az s állapotból a döntés után a π(s) stratégiát követve mennyi a várható diszkontált kumulált jutalom.

Azt a π(s) függvényt keressük, ahol a értékek maximálisak, ezt -al szoktuk jelölni. Ez az optimális stratégia. Elméletben, ha ismerjük ezt a függvényt minden állapotra és döntésre, akkor a kumulált jutalom maximalizálásához egyszerűen minden állapotban azt a döntést kell választani, ahol a függvény maximális. A probléma viszont, hogy a valóságban nem ismerjük az állapot átmeneteket és az jutalom függvényt, így a-t sem. Szerencsére viszont ezt az egyes lépések után a kapott jutalom és a következő állapot megfigyelése után a Bellman egyenlet alapján egy iteratív módszerrel tudjuk közelíteni. Ez a megfontolás adja az alapján a legtöbb megerősítéses tanulás alapú algoritmusnak.

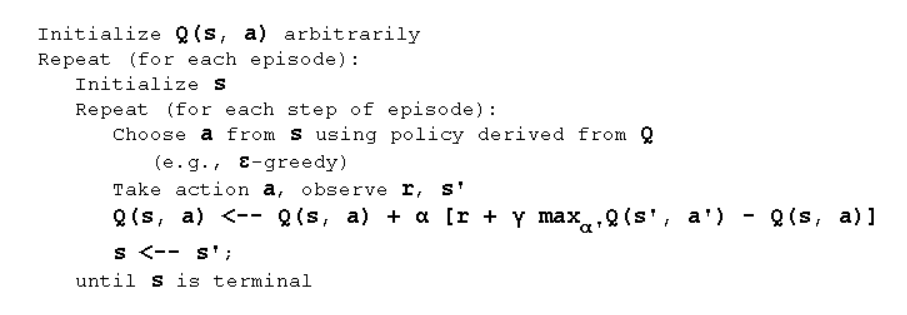
### Algoritmusok kategorizálása

A megerősítéses tanuláson alapuló algoritmusokat két nagyobb csoportra lehet osztani, modell független és a modell alapúra. A modell alapú algoritmusok mögötti elgondolás az, hogy ha egy MDP összes összetevőjét ismerjük, tehát a T(s,a,s’) állapotátmenet függvényt és az R(s,a,s’) jutalom függvényt is, akkor egyértelműen megtudjuk határozni azt az optimális policy-t, ami a jövőbeni jutalmat a maximalizálja. Ez alapján az ágens előbb megpróbálja felfedezni a környezetet és modellt építeni amely megmondja az egyes állapot átmeneteket és az azokért járó jutalmat, majd ezen modellre offline számolja ki az optimális stratégiát. A modell független megközelítés ezzel szemben egyből a action-value függvényt vagy közvetlenük a optimális stratégiát szándékozik meghatározni. A dolgozatomban ez utóbbi megközelítést alkalmazó algoritmusokat alkalmaztam, így ezek bemutatására szeretnék részletesebben kitérni. A modell független szemléletbe tartozó módszereket két további nagyobb csoportra lehet osztani.



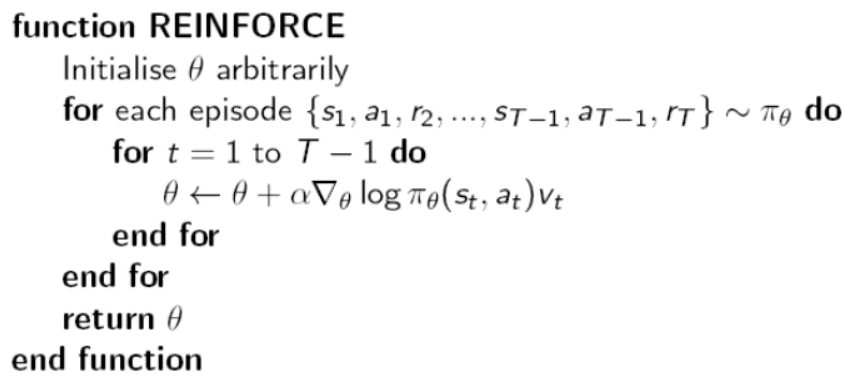
2. Algoritmusok csoportosítása

Az egyik az value based, amikor is az algoritmus különböző iteratív módszerekkel a Q(s, a) függvényt közelíti. A Q függvény ismeretében közvetve megtanulja az optimális stratégiát is, hiszen ha mindig a maximális Q értéket eredményező döntést választjuk, akkor determinisztikus környezetben maximalizáljuk a jövőbeli diszkontált jutalmat. Ennek a módszernek a hátránya viszont, hogy gyakran az algoritmus sok időt tölt suboptimális stratégiák értékelésével, és olyan állapot átmenetek Q értékeinek becslésével, amit lehet, hogy az optimális stratégiát követve nem is érintünk.



1. Q learning

A policy based algoritmusok ezzel ellentétben nem törekednek a pontos Q értékek feltérképezésére Mivel a megfelelő döntés meghozatalához nem szükséges a Q értékek pontos ismerete, elég, ha azokat egymáshoz viszonyítva helyesen tudjuk becsülni, így a policy based algoritmusok általában gyorsabban konvergálnak az ideális megoldáshoz. Ezek működésük során π policy-t követve megfigyelik a policy értékét az epizód végén, majd a policy paramétereit úgy hangolják, hogy ez az érték nőjön. Ehhez feltételezzük, hogy a policy-t valamilyen differenciálható függvény alkotja.



1. Reinforce

Folytonos állapotteret(angolul state space) és mozgásteret (angolul action space) tartalmazó problémák eset gyakran alkalmazzák ennek a két kategóriának a kombinációját. A mögötte megfontolás az, hogy mivel a bellman updatehez nem ismerjük a Q függvény maximumát ezért két függvény becslőt használunk. Egyet a policy értékeléséhez, ami megmondja, hogy adott S állapotban az A döntésnek mennyi az értéke. Ezt nevezik az angol szakirodalomban critic-nek, és egyet, ami megmondja, hogy adott S állapotból melyik A döntés fogja maximalizálni a jövőbeni jutalmat, ezt pedig actor-nak nevezik.

### Exploration exploitation probléma

Mivel a bemutatott algoritmusok az általuk megtapasztalt állapotátmenetekből és az azokért járó jutalomból tudnak tanulni, elengedhetetlen a hatékony működés érdekében ezeknek a megfelelő feltérképezése, egy ún. exploration function segítségével. Lehet bármilyen jó az algoritmus, ha egy az optimális policyhoz elengedhetetlen állapot átmenetet nem fedezett fel, akkor nem tudja elérni a globális optimumot. Erre több különböző módszert is kidolgoztak már, melyeket a későbbiekben bemutatok.

További problémát okoz a megerősítéses tanulás témakörében az ún. exploration-exploitation dilemma. Ez azt fogalmazza meg, hogy bár a célunk az, hogy maximalizáljuk a jövőbeli kumulált jutalmat, viszont ehhez az optimális stratégiát kell kövessünk, de ezt csak akkor tudjuk megtanulni, ha már megfelelő állapot átmenetet fedeztünk fel. Nehéz viszont megfogalmazni, hogy hol van az egyensúly a kettő között, különösen mivel ez minden problémakörnél mást jelent.

### Gyakori felfedező függvények

A következőkben a teljesség igénye nélküli szeretnék bemutatni, pár gyakran használt módszert az új átmenetek felfedezésére.

**ε-greedy**

ε valószínűséggel egyenletes eloszlással véletlenszerűen választunk a lehetséges döntések közül, 1- ε valószínűséggel pedig az π(s) policy szerinti döntést követjük. Az ε a tanulás előre haladásával folyamatosan csökkentjük, így a döntéshozás konvergál a π(s) policyhoz. A hátránya, hogy olyan problémák esetében, ahol egy epizódban a maximális jutalom eléréshez sok lépést kell tenni és az optimálishoz közeli stratégiák száma lényegesen kisebb az optimálistól távol esőkétől, a lépéssor végéhez kevesebbszer jut el, így ott kevesebb állapot átmenetet tud felfedezni.

**Softmax**

Mivel előfordulhat, hogy kettő vagy több döntéshez tartozó Q érték is közel van a maximumhoz, ezért ezeket gyakrabban szeretnénk megtapasztalni, mint amik nagyon távol helyezkednek el. Erre nyújt lehetőséget a softmax függvény ami az egyes döntésekhez az alábbi valószínűségi eloszlást rendeli hozzá.

A T az úgynevezett temperature paraméter, amivel állíthatjuk, hogy mennyire priorizálja a legnagyobb értékeket. esetén közelít az egyenletes eloszláshoz, esetén pedig a greedy policyhoz.

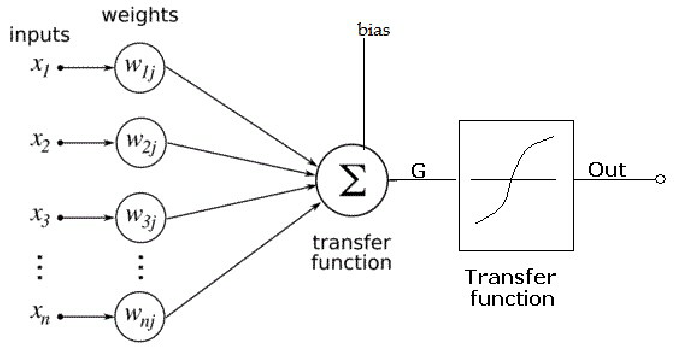
**A**

## Neurális hálózatok

A klasszikus megerősítéses tanuláson alapuló algoritmusok kiválóan működnek véges diszkrét állapot és akció terekben, azonban az érdekesebb problémák megoldására gyakran ez nem elég, mert vagy folytonosak, vagy pedig mert az állapottér annyira kiterjedt és több dimenziós, hogy nem lenne elég erőforrásunk, memória illetve számítási kapacitás hozzá. Szükségesnek látszik tehát valamilyen függvénybecslő használata, amely segítségével a Q(s,a) függvényt vagy a π(s,a) függvényt becsüljük. Erre manapság a legnépszerűbb megoldás a neurális hálók használata.

### Neurális hálózatok felépítése

A neurális hálózat olyan irányított gráf, amelynek a szerkezetét az emberi agyban található neuronokról mintázták. A bementei rétegek és a kimeneti rétegek között úgynevezett rejtett rétegek helyezkednek el. Ezeket a rétegeket a neuronok vagy más szóval perceptronok alkotják, amelyek a szomszédos rétegek neuronjaival kapcsolódnak valamilyen struktúra alapján, továbbá tartalmaznak egy ún. aktivációs függvényt. Ez egy olyan függvény, ami a neuron bemeneteinek súlyozott összegét valamilyen nemlineáris függvény alapján alakítja át.



1. Egy mesterséges neuron felépítése

Mivel tetszőlegesen „mély” hálózatot létrehozhatunk a rejtett rétegek egymás után pakolásával, így ezért hívják az ilyen hálózatokat angolul Deep neural network-nek. Innen kapta a nevét a mély tanulás is, a Deep learning. Az ilyen hálózatokat felügyelt tanítás segítségével tudjuk tanítani. Ez azt jelenti, hogy adott bementre már rendelkezésünkre áll a megfelelő kimenet. Ezt hasonlítjuk össze a neurális háló kimenetével. A predikált kimenet és a valós kimenet különbségéből valamilyen hibafüggvényt számolunk. Ezt a hibafüggvényt felhasználva pedig az ún backpropagation segítségével úgy hangoljuk a hálózat súlyait, hogy a kimenete minél jobban illeszkedjen az elvárthoz anélkül, hogy túltanulna a hálózat. Az ilyen neurális hálókkal viszonylag jól lehet közelíteni tetszőleges függvényt, ezért alkalmas a megerősítéses tanulás esetében a kiterjedt állapot térben az állapotok absztrahálására, az információtartalmuk sűrítésére.

. 

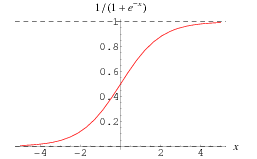
1. Feed forward neurális háló felépítése

### Aktivációs függvény szerepe és főbb típusai

Mint korábban említettem az egyes neuronok tartalmaznak egy aktivációs függvényt is. Ennek a jelentősége, hogy ha ez egyszerűen lineáris lenne, akkor bármilyen bemeneti vektor hatására a háló egy lineáris függvény szerinti kimenetet adna. A valóságban a modellezni kívánt összefüggések viszont a legritkább esetben alkotnak lineáris függvényt, így szükséges valamilyen nemlinearitást belevinni a hálózatba, hogy képes legyen tetszőleges függvény becslésére. Erre célra több függvény is megfelel, ezek közül szeretném ismertetni a legelterjettebbeket.

**Sigmoid**

A neurális hálók felfedezésekor ez volt az első használt aktivációs függvény. Nullában és egyben korlátos, így ha az elvárt kimenet is e két korlát között mozog, például egy klasszifikációs probléma esetén, akkor a kimeneti rétegben ez az aktivációs függvény hasznunkra válhat. Rejtett rétegekben manapság már ritkán használják, mert gyakaran okozhatja az ún. eltűnő gradiens problémát.



**Tanh**

Hasonló a sigmoidhoz, csak ez a függvény –1 és 1 közötti értékeket vehet fel. Elsősorban kimeneti rétegben szokás használni. Rejtett rétegekben ezt sem használják, mert ez is szenved az eltűnő gradiens problémától.



**Rectified Linear Unit (ReLu)**

A modern hálókban ma már szinte kizárólag ezt, vagy ennek változatait használják, mert nincs szükség a háló előtanítására, tovább ezzel kiküszöböli az eltűnő gradiens problémát.



### Neurális hálózatok alkalmazásai

A gépi tanulást, és így a neurális hálózatokat is általában két fajta problémakör megoldásra szokták használni. Az regressziós és a klasszifikációs problémákra.

A regresszió során valamilyen általában folytonos függvény értékét szeretnénk a bemenetek függvényében becsülni. A klasszifikáció során pedig diszkrét számú kategóriába, klaszterbe besorolni az egyes elemeket.

### Költség függvények

Attól függően, hogy a regressziós vagy klasszifikációs problémáról van szó eltérő költségfüggvényeket érdemes használni. Mivel a megoldandó feladat egy regressziós probléma a továbbiakban szeretnék pár regresszió esetén használt költségfüggvényt bemutatni azok tulajdonságaira is kitérve.

**MEAN SQUARED ERROR**

A nagyobb eltérés négyzetesen nagyobb hibát okoz, így azokat jobban bünteti a hibafüggvény. Elsősorban regressziós probléma megoldása esetén szokás használni.

**MEAN AVERAGE ERROR**

**HUBERT LOSS**

Ha az tanító adataink között vannak hibás adatok, amik a valós adatoktól agy mértékben eltérne (angolul outlier) az MSE hibafüggvény esetén nagyon eltorzíthatják az tanítást. Erre egy lehetséges megoldás a HUBERT LOSS, ami meghatározott intervallumon belül négyzetesen, azon kívül lineárisan veszi figyelembe a hibát, így kiküszöböli az outlierek által okozott nagymértékű torzítást.

### Hálózat tanítása

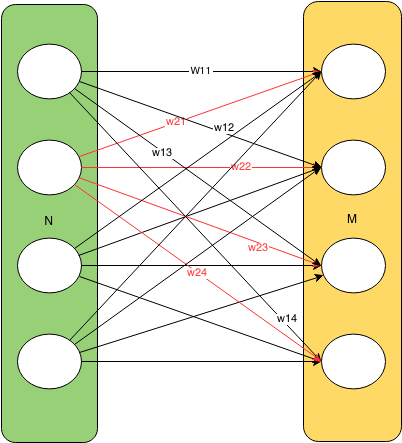
A hálózat tanítását a tanító adatok birtokban az ún Backpropagation algritmus alapján végezzük. …

### Neurális hálózatok típusai

A neurális hálózatok a rejtett rétegek szerkezetétől függően máshogy viselkedhetnek és más-más feladatok megoldására jók. Több fajta réteg és struktúra létezik már, ezek közül szeretném bemutatni az általam felhasználtakat.

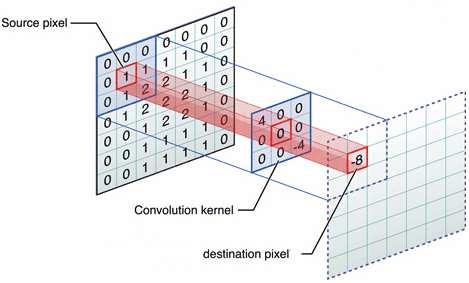
**FULLY CONNECTED**

A fully connected vagy magyarul teljesen összekötött rétegek között minden neuron minden neuronnal össze van kötve. A legegyszerűbb neurális hálók ilyen rétegek egymás után pakolásával épülnek fel. A bemeneti adatok és a kimeneti adatok közötti kapcsolatot tanulja meg.



**CONVOLUTIONAL**

A konvolúciós rétegek egy ún. kernellel pásztázzák végig a bementet és képzik le a kimenetet. Egy kernel súlyai egy kimeneti réteg leképzésére érvényesek, és egyszerre több kimenet is készülhet. Mivel az élek súlyai megosztottak, sokkal kevesebb paramétert tartalmaznak, mint a sima fully connected rétegek. Előnye, hogy az egymáshoz közelebb elhelyezkedő adatok kapcsolatát is figyelembe veszi, így gyakran alkalmazzák a feature extraction-re, az adatok közötti kapcsolatok kinyerésére. Létezik 1D, 2D, illetve 3D-s verziója is.



**Pooling rétegek**

Az ún pooling rétegek a bementi réteget megszűrve állítják elő a kimenetet. A szűrőt ami általában az általa lefedett tartomány maximumát adja vissza (MaxPooling), bizonyos lépésközzel pásztázza végig a bementét, így egyfajta dimenzió redukciót hajt végre. Gyakran használják együtt konvolúciós rétegekkel.



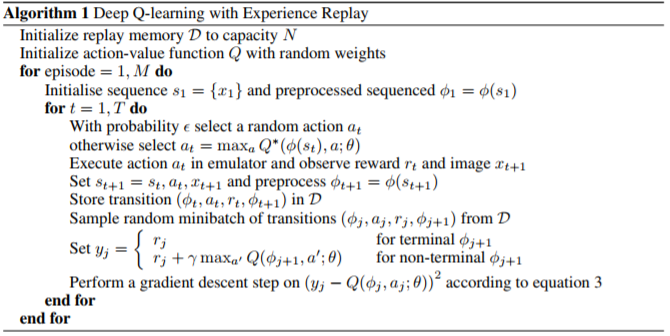
## Deep reinfocement learning

### Motiváció

Bár a klasszikus megerősítéses tanulás alapú algoritmusok kis állapotterü problémák esetén jól használhatók a állapottér és a mozgástér dimenzióinak növelésével hamar el lehet érni a módszer korlátait. A modern megerősítéses tanuláson alapuló algoritmusok, ma már szinte mind Deep Learninggel vannak kombinálva, a hatalmas állapottér kompenzálására. Az ezzel járó előnyökkel viszont sajnos hátrányok is járnak. Míg a klasszikus táblázatos módszert alkalmazó algoritmusok esetében megfelelő exploration function alkalmazása esetén garanciánk van arra, hogy az algoritmus a globális optimumhoz konvergál, addig a deep reinforcement learning algoritmusok gyakran megakadnak lokális optimumoknál. Továbbá míg a sima neurális hálók tanítása során álltalában a rendelkezésre álló tanító adatbázis statikus, addig a deep reinforcement learning esetében általában ez folyamatosan változik, mind méretében, mind eloszlását tekintve, így az eredményesebb működés elérése érdekében egyéb trükköket is alkalmaznunk kell.

### DQN

A mesterséges intelligencia témaköre a kezdeti lelkesedés után az új lökést 2013-ban kapta, amikor is a DeepMind nevű cég az általuk kifejlesztett Deep Q learning algoritmus segítségével, emberi teljesítményt túlszárnyaló eredményeket tudott elérni a klasszikus Atari számítógépes játékokban, csupán a képernyő pixeleit felhasználva, mint bemenet. Ez egy Q learninghez hasonló algoritmus, azzal a különbséggel, hogy a Q(s, a) függvényt egy neurális hálóval közelíti és a maximális Q értékez tartozó döntést hozza meg. Mivel a megtapasztalt állapotátmenet és jutalom eldobása a tanítás után, nem lenne túl effektív, ezeket egy ún. experience replay memóriában tárolják. Ebből vesznek ki valamilyen eloszlással egy mini batchnyi adatot és tanítják vele a hálót. Mivel a cél, hogy a háló az egyes állapotokhoz és akciókhoz tartozó Q értéket tudja becsülni, a kimenetére a R(s,a,s’)+maxaQ(s’,a) target értéket tesszük és erre tanítjuk rá. Sajnos mivel a maxaQ(s’,a)-t is a neurális hálóból vesszük ezért, a rendszer alapesetben nagyon instabil. Erre az ún. target network használatát találták ki, ami azt jelenti, hogy a maxaQ(s’,a) értéket egy másik háló segítségével nyerjük ki amit tau lépésenként szinkronizálunk az eredetivel.



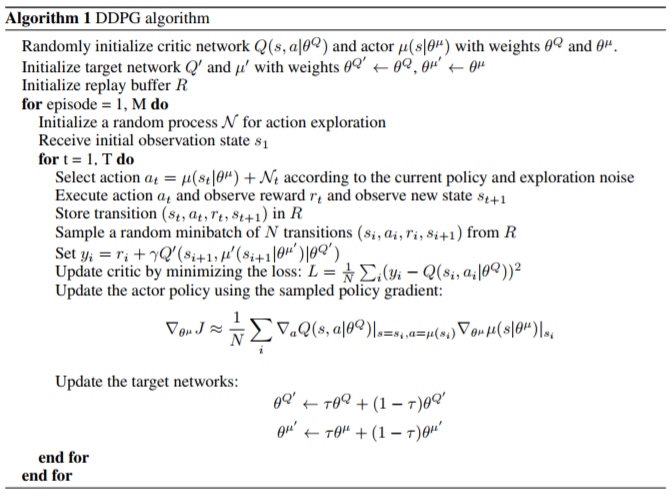
1. DQN experience replayel

### DDQN

A sima DQN algoritmus hajlamos túlbecsülni a Q értékeket ugyanis a tanítás elején ezek nagyon zajosak és ezeknek mindig a maximumát veszi. Erre a problémára a Double Deep Q Network nyújt segítséget. Itt két hálót tanítunk párhuzamosan. Az egyikkel azt határozzuk meg hogy a következő állapotból mi lenne az optimális akció, a másikkal pedig becsüljük, hogy az így kapott állapot akció párnak mennyi a Q értéke. Mivel a target network-höz már úgyis használunk egy hálót, fel tudjuk használni ehhez a változtatáshoz is. A döntéshozásnak és a döntés értékelésének ez a fajta szétválasztása bizonyítottan csökkenti a Q értékek biasát, ezáltal stabilabb működést kapunk.

### DDPG

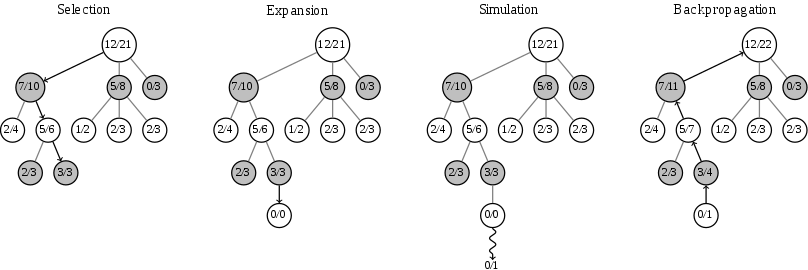
Míg a DQN és a DDQN algoritmusokat diszkrét kimenetek esetén tudjuk alkalmazni, a legtöbb irányítási algoritmusnál folytonos beavatkozó jelre van szükségünk, így más módszert kell keressünk. Erre nyújt megoldást a Deterministic Deep Policy Gradient módszer (DDPG). Ez egy actor-critic típusú algoritmus, tehát két fő részből áll. Egy actor hálózatból, ami az ideális policyt határozza meg, és egy critic hálózatból, ami a policy értékelését végzi. Ez a hálózat is szenved a neurális háló miatti instabilitástól, amit csökkenteni lehet a DQN-nél megismert target hálózatok használatával. Ezzel az algoritmussal folytonos kimenetet tudunk becsülni, így elméletben több irányítástechnikai probléma megoldáshoz is felhasználhatjuk.



### Monte Carlo Tree Search

A reinforcement learning alapját adó MDP-ben, az állapotokat ábrázolhatjuk egy fa gráf segítségével. A kiterjedt állapotterű szekvenciális problémák esetén ez a fa gráf gyakran nagyon mély, illetve széles, így a korlátozott erőforrásaink és a rendelkezésre álló idő miatt, az egész állapotteret nem tudjuk maradéktalanul felfedezni, így az eddig megismert módszerek nem alkalmazhatóak sikerrel. Erre egy alternatív megoldást jelent a Monte Carlo Tree Search nevű algoritmus. Ez egy iteratív algoritmus, melyet sikerrel alkalmaztak olyan bonyolult szekvenciális problémák megoldására, mint például a Go.

Az algoritmus négy egymás utáni lépésből áll. Döntés választása (selection), állapot kibontása (expansion), az állapotból szimulációk futtatása (simulation), majd az eredmények visszaterjesztése (backpropagation) és majd az első lépéssel újra kezdjük. Az algoritmus során addig követjük a π(s) stratégiánkat, ameddig biztosak vagyunk a lépésünkben. Amikor már nem tudjuk, mit lépjünk, akkor n darab szimulációt végzünk véletlenszerű döntések meghozásával, vagy valamilyen felfedező függvényt követve. Az így kinyert eredményeket visszaterjesztjük az szimulációk során meghozott döntés sorra, majd az így kapott eredmények segítségével az eddig bizonytalan helyzetből azt a döntést hozzuk meg ami a legjobb eredményre vezetett, így a döntési fában egy szinttel lejebb tudunk jutni. Lényegében az algoritmus a döntési fának megkeresi azon ágait, amik valószínűbben tartalmazzák a számunkra érdekesebb állapotokat és azt az ágat bontja tovább és fedezi fel.



# Megerősítéses tanulás a gyakorlatban

A versenyautó irányításának megtervezése előtt egy egyszerűbb problémán teszteltem az algoritmusokat. Az implementáció során két programnyelv jött szóba a Matlab és a Python. A Matlab azért, mert rendkívül kiterjedt eszköztára van és szimulációk futtatásához kézenfekvő megoldás, a Python pedig, mivel a tudományos életben szinte kizárólag ezt használják a mélytanulás alapú algoritmusok implementálásához a nagyon rugalmas és könnyen használható moduljainak köszönhetően. (Tensorflow, Keras)

## Kő-papír-olló megerősítéses tanulással

A közismert kő-papír-olló játékot két játékos játsza. A játékosok három különböző döntés közül választhat. A kő üti az ollót, a papír üti, a követ, és az olló üti a papírt. Egy ilyen környezetben egymás után több mecset is játszva a játékot modellezhetjük egy MDP-vel is, ha az ellenfelünk valamilyen statikus stratégiát követve játszik, például, hogy az utolsó játék kimenete alapján dönt. Egy ilyen MPD-ben pedig a klasszikus Q-learning garantáltan rátanul az optimális stratégiára.

A teszt során x,y és z paramétereket használtam

IDE EGY DIAGARMOT MATLAB

## Kő papír-olló neurális hálóval

Bár egy ilyen kis állapotterű probléma nem igényel feltétlenül egy neurális háló alapú algoritmust, a később használni kívánt keretrendszer kiválasztásának eldöntésére, implementáltam a fentebb bemutatott DDQN algoritmust. Ebbe a problémakörben az algoritmus szinte azonnal rátanul az egyszerűbb stratégiákra, illetve a bonyolultabbak se tartanak sokkal tovább.

A teszt során x,y és z paramétereket használtam

IDE EGY DIAGRAMOT MATLAB,PYTHON

## Értékelés

A problémát mind a klasszikus megerősítéses tanuláson alapuló algoritmus, mind a neurális hálóval ötvözött is maradéktalanul meg tudta oldani. A keretrendszerek összehasonlítását illetően, bár a Matlab neurális hálózatok implementálására rendelkezésre álló eszköztára sokat fejlődött az utóbbi időben, véleményem szerint a Python még mindig sokkal egyszerűbb és rugalmasabb keretrendszert kínál, továbbá a mivel a Python és így a hozzá írt modulok open source licence alá tartoznak, sokkal nagyobb online közösséggel rendelkezik a Deep Learning területén, így a munka közbe felmerülő kérdésekre, problémákra gyorsabban megoldást lehet találni a különböző fórumokon. Ezek miatt a megfontolások miatt az algoritmust és a tesztelésére létrehozott szimulátort Pythonban valósítottam meg.

IDE EGY STATISZTIKÁT, ohgy hányan hasznáknak PYTHONT

# Versenyautó irányításának megvalósítása

## Szimulátor

### Grafikus felület

A tanításhoz felhasznált szimulátor grafikus felhasználói felülete két részből áll. A programot elindítva a rögtön a Settings oldalra kerülünk, ahol beállíthatjuk a pályára , az autóra illetve az irányítás mögötti algoritmusra vonatkozó paramétereket.

Az autóra vonatkozó főbb beállítások:

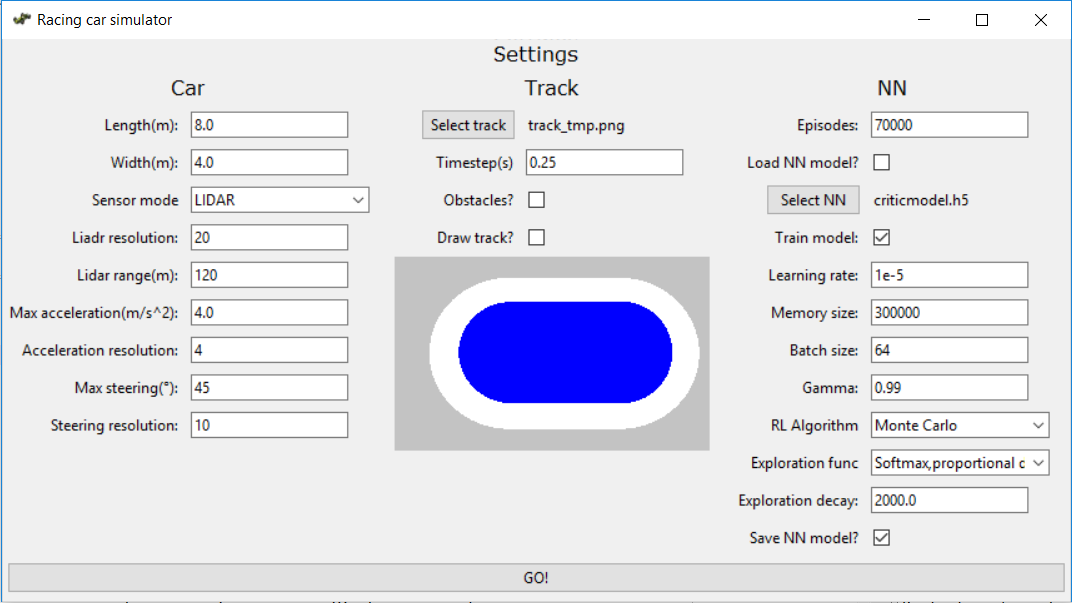
* Hosszúság
* Szélesség
* Szenzor rendszer típusa (lidar)
* Lidar felbontása
* Lidar maximális látótávolsága
* Maximális gyorsulás
* Maximális kormányszög
* Gyorsulás felbontása (a diszkretizáláshoz)
* Kormányszög felbontása (a diszkretizáláshoz)

A pályára vonatkozó beállítások:

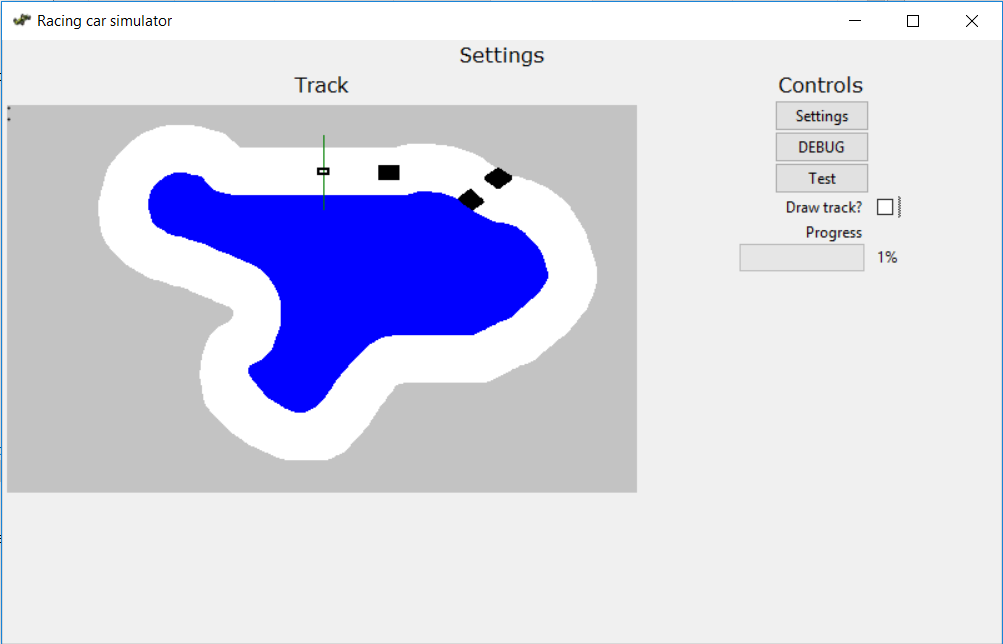
* Pálya kiválasztása
* Statikus akadályok használata
* A tanítás során a pálya kirajzolása
* Alkalmazott időköz

Az vezérlő algoritmus paraméterei:

* Epizódok száma
* Tanítás engedélyezése
* Learning rate
* Algoritmus kiválasztása
* Neurális háló betöltése
* Neurális háló mentése a tanítás végén
* Exploration function
* Diszkont faktor
* Mini batch mérete
* Experience replay memória mérete



A „GO” gombra kattintás indítja a tanítást, és automatikusan átvált a szimulátor oldalra. Ez két részből áll. Az oldal bal felén található a pálya, ahol nyomon tudjuk követni, hogy az autó jelenleg milyen ívet követ. Az oldal jobb oldalán pedig egyéb kontrol gombok találhatóak. A settings gombbal visszajutunk a beállítások oldalra. A DEBUG gomb a debugoláshoz fontosabb információkat ír ki a konzolra. A Test gomb megnyomásával a következő körben az algoritmus figyelmen kívül hagyja a felfedező algoritmust és kizárólag a vezérlő algoritmustól kapott akciókra hagyatkozik, így futás közben is megnézhetjük, hogy addig mit tanult meg. Ez minden századik lépésben lefut és a legjobb eredményt elérő háló súlyait elmenti későbbi felhasználásra. Ezen kívül egy progressbar-on követhetjük nyomon, hogy az epizódok hány százalékát teljesítette már az algoritmus. Tanítás közben a konzolon megfigyelhetjük, a tanulás előrehaladását, a legutolsó száz epizód összes jutalmának átlagát, az eddig elért legjobb teszt kör eredményt, illetve, hogy ez melyik epizódban történt.



### Szimulátor felépítése

A szimulátor öt főbb részből áll. Az egyik a grafikus felhasználói felület, amely lehetővé teszi a beállítások módosítását, tárolását, illetve a tanulási folymat noymon követését. A environment tartalmazza a pályára vonatkozó információkat. Ez számolja ki a differenciál egyenletek megoldásával az autó aktuális pozícióját, az autó szenzor adatait, illetve az egyes döntések utáni jutalmat. Az autó tartalmazza az adatait. Az experience replay memória, ami az egyes megtapasztalt állapotátmenetekre vonatkozó információkat tárolja és mintavételezi azt a neurális háló számára. A neurális háló pedig a kapott tanító adatok segítségével tanulja meg a megfelelő vezérlést, és szolgáltata a bevatkozó jelet.

IDE EGY sablonos képet és egy kis leírást

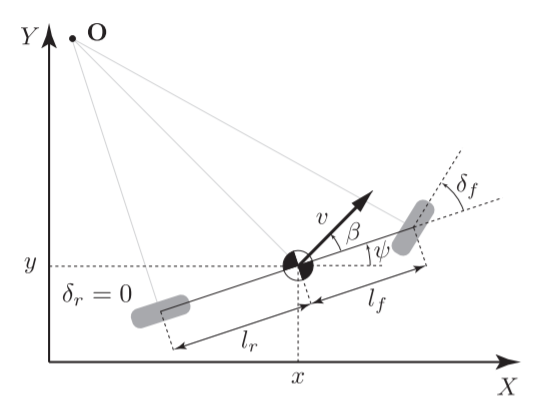
A megtapasztalt állapotátmenetek és az értük járó jutalmak tárolására szolgáló experience replay memory-t a Pythonhoz írt egy modul, a Pandas segítségével valósítottam meg. Ez egy adatok kezelésére írt open source package, amely kiválóan alkalmas az adatok tárolására és analizálására. Az itt tárolt adatok egy sorának struktúrája:

STATE,ACTION,REWARD,NEXT STATE,OVER,id, p

Itt az id a STATE,ACTION,REWARD, NEXT STATE és OVER felhasználásával készült hash, amit a memóriában található állapot átmeneteket duplikáció mentesíteni lehessen szükség esetén. A p pedig a prioritised experience replay implementálásához használt prioritás változó, amely megmondja, hogy a mini-batchek mintavételezésénél milyen prioritással számítson az adott állapotátmenet az eloszlásba.

## Kinematikai modell

Az autó kinematikai modelljének egy egyszerű kinematikai bicikli modellt alkalmaztam. Feltételezzük, hogy az autó súlypontja az alapterületének a közepén helyezkedik el, magassága elhanyagolható, és a tömege is, így nem foglalkozunk a tehetetlenségével. Az autó mozgását közelíthetjük egy a hossztengelyére illesztett bicikliével. Ennek csak az első kerekét tudjuk kormányozni. Az esetleges megcsúszásoktól jelen dolgozatban eltekintünk. Ekkor felírhatjuk a modell mozgásának differenciál egyenleteit, amelyek a következőek:



A differenciálegyenleteket 5. fokú runge-kutta módszerrel oldom meg, a Python Scipy tudományos számításokhoz használt modulja segítségével. A felhasználó felületen megadott időközönként változtatom meg a beavatkozó jelet, vagyis az algoritmus által kiszámolt gyorsulást és kormányszöget, illetve számolom ki az autó új állapotát megadó pozíciót, sebességet és orientációt.

## Probléma nehézségei

A probléma megoldása során több nehézség is felvetődött. Nem volt triviális, hogy mit lenne érdemes állapot vektorként használni, ami kellőképpen egyedien képezi le az állapotokat, hogy megfelelően tudjon rajta tanulni az algoritmus, viszont elég általános ahhoz, hogy ne a pályára tanuljon rá, hanem arra, hogy milyen helyzetben hogy kell irányítani az autót. Mivel a megválasztható beavatkozó jelet először folytonosra választottam a megfelelő algoritmus kiválasztása, is sok időt vett igénybe. Továbbá mivel ez a terület és így az algoritmusok nem teljesen kiforrottak, más problématerületen voltak letesztelve, így nem volt garancia arra, hogy esetemben is olyan eredményesen működnek. Az implementált algoritmusoknál azt is figyelembe kellett vegyem, hogy korlátozott számítási kapacitás, idő és más erőforrás ált a rendelkezésemre. Több algoritmusnak sem sikerült ésszerű időn belül, elfogadható megoldáshoz konvergálni. A hosszú futásidők, pedig az algoritmus fejlesztését lassúvá tették. Továbbá más más algoritmusok esetén más más exploration function használata volt indokolt, aminél annak paraméreit is megfelelően kellett finomhangolni.

## Elméleti megfontolások

### Bemenetek

A vezérlő algoritmus bemeneteinek megválasztásánál két lehetséges irányban indultam el. Az egyik, hogy a bemenetek az autó helyzete, sebessége és orientációja lenne. Ez a lehetőséget viszont hamar elvetettem, mert ez az állapot vektor nem elég általánosított, így az algoritmus tanítása nagyon sok időt venne igénybe, és akkor is a betanult háló csak az adott pályára, az adott konfigurációra lenne jó, mivel az algoritmus a pályát tanulná be.

A másik szerint pedig egy az autó elejétől a LIDAR működéséhez hasonlóan az orientációhoz viszonyítva különböző szögekben megmérem az autó és az akadály vagy fal távolságot. Ezeket a szenzoradatokat, egy flaget, ami azt mondja meg, hogy a pálya elvárt haladási irányának megfelelően áll-e az autó, illetve az autó aktuális sebességét használom állapot vektorként. Ez elméletben gyorsabb tanulást eredményez, hiszen sokkal általánosabb, így ha például egyszer már megtanult, egyenes szakaszon végig haladni, akkor ezt későbbi ilyen szakaszon is tudni fogja. Továbbá, ha az algoritmus egy kellőképpen általános pályán tanítjuk, akkor az egy másik pályán is használható, feltéve, hogy a pálya hasonló tulajdonságokkal rendelkezik (megfelelően előfeldolgozott). Mindemellett életszerűbb is az, hogy egy önvezető autó a rajta található szenzoradatokra és esetleg a GPS-adatokra támaszkodjon, minthogy csak a GPS által szolgáltatott adatokra.

Próbálkoztam még az állapotvektor a kiegészítésével, például a sebesség pályaívre merőleges és párhuzamos felbontásával, illetve a haladási irány pálya tengelyével bezárt szögével, viszont a tapasztalatok azt mutatták, hogy ezek nem javították jelentősen az eredményt, sőt inkább csak az algoritmus bonyolultságát és a tanítási időt növelték.

### A jutalom függvény

A jutalomfüggvény megfelelő megválasztása létfontosságú, az ideális működés elérése érdekében. Az algoritmus lényegében ennek segítségével értelmezi, hogy mi a cél, és ha nem elég konkrét, könnyen előfordulhat, hogy bár sikerül maximalizálnia a kumulált jutalmát, viszont az eredményezett működés messze áll attól, amit a tervezés során elképzeltünk.

Esetünkben kézenfekvő lenne az a jutalmazási rendszer, amely szerint valamekkora pozitív jutalmat kap, ha beér a célba és negatívat, ha az autó valamivel ütközik. Ez a fajta ritka jutalmazás (sparse rewarding) az ilyen szekvenciális, egymásra épülő lépésekből álló problémák esetén nem célravezető, ugyanis az algoritmus csak nagyon későn kap visszacsatolást arról, hogy az általa követett stratégia mennyire jó. Ez nagyon lassú tanuláshoz vezethet. Ehelyett én az pályaíven megtett úthosszal arányos jutalomfüggvényt használtam, így az algortimus minden lépés után kap valamilyen visszacsatolást, hogy mennyire volt jó a lépése. Emellett a pálya adottságai miatt az autó hajlamos lesodródni az útról, ezért kiegészítettem még a pályaív tengelyére merőleges irányú elmozdulás büntetésével is. Felmerülhet bennünk a kérdés, hogy ez nem okozza-e azt, hogy az autó csak a pályaív közepén halad, viszont a diszkontfaktor miatt ez nem így van.

### Alkalmazott exploration function

Exploration function-nak az ε-greedy és a softmax-ot is kipróbáltam, amikből az utóbbi bizonyult eredényesebbnek. Ezt még annyival egészítettem ki, hogy szekvenicában minden diszkrét időpillanathoz külön temperature változót rendeltem és mindig az csökkentettem lineárisan amelyik időpillanatban járt az autó.

### Hálózat szerkezete

Dense

## Implementált algoritmusok

Először folytonos gyorsulás és kormányszög értékek mellett szerettem volna megoldani a feladatot, ehhez az elméleti részben bemutatott Deep Deterministic Policy Gradient (DDPG) módszert implementáltam. Egy actor hálózattal becsültem meg az aktuális állapothoz tartozó optimális akciót, egy másik critic hálózattal pedig értékeltem az így a kapott lépést. A folytonos kimeneti jelhez normális eloszlású zajt kevertem, így valósítottam meg az állapot átmenetek felfedezését. A neurális hálók szerkezetét illetően kipróbáltam egy 1D konvolúciós és egy sima fully connected architechtúrát is. Sajnos mivel a környezet implementációja még nem volt megfelelően optimalizálva, illetve a tanításhoz használt számítógép számítási kapcitása is erősen korlátozott, ezt az irányt el kellett vessem és diszkrét kimenetekre kellett az algoritmust megtervezzem.

A következő algoritmus a Deep Q learning volt. Itt diszkretizáltam a gyorsulás és a kormányszög skáláját és a neurális hálóval az aktuális állapotban minden lehetséges akcióhoz tartozó Q értékeket becsültem vele. Végül a kimenetek közül egy diszkrét eloszlás alapján választottam ki a végleges akciót, amelyhez az eloszlást a kapott Q értékek softmax függvényértékei. Minden diszkrét időpillanathoz külön temperature változót definiáltam, amit lineárisan csökkentettem egy minimum korlátig. Ennek az eredménye az volt, hogy egy bizonyos időpillanban az algoritmus a neurális háló által kiadott maximális Q értékhez tartozó akciót priorizálta. Mivel a DQN algoritmus gyakran túlbecsli a Q értékeket ezért ez egy idő után nem konvergált elfogadható megoldás felé.

A Q értékek túlbecslését kiküszöbölendő átalakítottam az implementált algoritmus, hogy az a Double Deep Q Learning algoritmus valósítsa meg. Bár ez az algoritmus már nem becsülte túl az Q értékeket, viszont mivel a probléma igen szekvenciális és az által meghatározott MDP meglehetősen mély állapot gráfot eredményez, így az algoritmus tanítása nagyon sok időt vesz igénybe. A hosszabb futásigényű, illetve gyakran hívott függvények optimalizálásával, illetve az ún prioritised experience replay használatával ugyan ezen tudtam javítani, de még ezzel együtt is hosszú tanítás után is csak a pálya egy kis részén tudott túljutni.

Végül a Monte Carlo Tree Search algoritmus bizonyult a legeredményesebbnek. Ez az előzőkéhez képest sokkal kevesebb idő alatt rátanult eredményes stratégiákra.

## Felmerült akadályok

### Optimalizálás

A tanítási algoritmusnak önmagában sok a futásideje, ezért fontos volt az implementált program optimalizálása. Ehhez a spyder fejlesztő környezet egyik beépülő moduljaként működő profilerét használtam fel. Ezzel nagy mértékben fel tudtam gyorsítani az szimulátor futását, főleg a gyakran meghívott szenzor adat generáló függvények optimalisálásával. Szükség volt továbbá az megtapasztalt állapotátmenetek hatékony, gyors hozzáférést lehetővé tevő tárolására.

### Teljesítmény

Mivel a fejlesztéshez és a futtatáshoz használt számítógép

### Hyperparaméter optimalizáció

### Exploration function optimalizáció

### Experience replay mintavételezése

## Eredmények értékelése

## Lehetséges tovább fejlesztések

A következő fejezet pár példán keresztül bemutatja a diplomatervekben és szakdolgozatokban szokásosan előkerülő formázások megvalósítását.

## Formázási tudnivalók

A dokumentum folyószövegéhez használjuk a **Normál** (angol Word esetén Normal) stílust.

### Címsorok

A fejezetcímek esetén a **Címsor 1-4** (Heading 1-4) stílusokat használjuk.

### Képek

A képhez használjuk a **Kép** stílust.

Képaláírást a képen jobb gombbal kattintva a Képaláírás beszúrása… opcióval adhatunk hozzá, így az automatikusan **Képaláírás** (Caption) stílusú lesz.



1.1. ábra: Példa képaláírásra

### Kódrészletek

Kódrészletek beillesztése esetén használjuk a **Kód** stílust.

using System;

namespace MyApp

{

class Program

{

static void Main( string[] args )

{

Console.WriteLine( "Szia Világ!" );

}

}

}

### Irodalomjegyzék

Az Irodalomjegyzékben szereplő hivatkozásokat **Irodalomjegyzék sor** stílussal formázzuk, a címüket pedig **Irodalomjegyzék forrás** stílussal emeljük ki.

A szövegbe a hivatkozásokat a Kereszthivatkozás beszúrása (Insert cross-reference) funkcióval helyezzük el (példa egy így beszúrt hivatkozásra: [1]), így azok automatikusan frissülnek a hivatkozások átrendezésekor.

# Összefoglaló

# Utolsó simítások

Miután elkészültünk a dokumentációval, ne felejtsük el a következő lépéseket:

* Kereszthivatkozások frissítése: miután kijelöltük a teljes szöveget (Ctrl+A), nyomjuk meg az F9 billentyűt, és a Word frissíti az összes kereszthivatkozást. Ilyenkor ellenőrizzük, hogy nem jelent-e meg valahol a "Hiba! A könyvjelző nem létezik." szöveg.
* Dokumentum tulajdonságok megadása: a dokumentumhoz tartozó meta adatok kitöltése (szerző, cím, kulcsszavak stb.). Erre való a Dokumentum tulajdonságai panel, mely a Fájl / Információ / Tulajdonságok / Dokumentumpanel megjelenítése úton érhető el.
* Kinézet ellenőrzése PDF-ben: a legjobb teszt a végén, ha PDF-et készítünk a dokumentumból, és azt leellenőrizzük.

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék