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](#_Toc498783204)

[Abstract 8](#_Toc498783205)

[1 Bevezetés 9](#_Toc498783206)

[2 Elméleti háttér 10](#_Toc498783207)

[2.1 Megerősítéses tanulás 10](#_Toc498783208)

[2.1.1 Markovi döntési folyamat 10](#_Toc498783209)

[2.1.2 Algoritmusok kategorizálása 12](#_Toc498783210)

[2.1.3 Exploration, exploitation dilemma 14](#_Toc498783211)

[2.1.4 Elterjedt felderítő függvények 14](#_Toc498783212)

[2.2 Neurális hálózatok 15](#_Toc498783213)

[2.2.1 Neurális hálózatok felépítése 15](#_Toc498783214)

[2.2.2 Aktivációs függvény szerepe és főbb típusai 17](#_Toc498783215)

[2.2.3 Költség függvények 18](#_Toc498783216)

[2.2.4 Hálózat tanítása és optimalizálás 19](#_Toc498783217)

[2.2.5 Neurális hálózatok típusai 20](#_Toc498783218)

[2.3 Deep reinfocement learning 22](#_Toc498783219)

[2.3.1 Motiváció 22](#_Toc498783220)

[2.3.2 DQN 22](#_Toc498783221)

[2.3.3 DDQN 23](#_Toc498783222)

[2.3.4 DDPG 23](#_Toc498783223)

[2.3.5 Monte Carlo Tree Search 24](#_Toc498783224)

[3 Megerősítéses tanulás a gyakorlatban 26](#_Toc498783225)

[3.1 Kő-papír-olló megerősítéses tanulással 26](#_Toc498783226)

[3.2 Kő papír-olló neurális hálóval 26](#_Toc498783227)

[3.3 Értékelés 26](#_Toc498783228)

[4 Versenyautó irányításának megvalósítása 28](#_Toc498783229)

[4.1 Szimulátor 28](#_Toc498783230)

[4.1.1 Grafikus felület 28](#_Toc498783231)

[4.1.2 Szimulátor felépítése 30](#_Toc498783232)

[4.2 Kinematikai modell 31](#_Toc498783233)

[4.3 Probléma nehézségei 32](#_Toc498783234)

[4.4 Elméleti megfontolások 33](#_Toc498783235)

[4.4.1 Bemenetek 33](#_Toc498783236)

[4.4.2 A jutalom függvény 33](#_Toc498783237)

[4.4.3 Alkalmazott exploration function 34](#_Toc498783238)

[4.4.4 Hálózat szerkezete 34](#_Toc498783239)

[4.5 Implementált algoritmusok 34](#_Toc498783240)

[4.6 Felmerült akadályok 35](#_Toc498783241)

[4.6.1 Optimalizálás 35](#_Toc498783242)

[4.6.2 Teljesítmény 35](#_Toc498783243)

[4.6.3 Hyperparaméter optimalizáció 36](#_Toc498783244)

[4.6.4 Exploration function optimalizáció 36](#_Toc498783245)

[4.6.5 Experience replay mintavételezése 36](#_Toc498783246)

[4.7 Eredmények értékelése 36](#_Toc498783247)

[4.8 Lehetséges tovább fejlesztések 36](#_Toc498783248)

[4.9 Formázási tudnivalók 36](#_Toc498783249)

[4.9.1 Címsorok 36](#_Toc498783250)

[4.9.2 Képek 36](#_Toc498783251)

[4.9.3 Kódrészletek 36](#_Toc498783252)

[4.9.4 Irodalomjegyzék 37](#_Toc498783253)

[5 Összefoglaló 38](#_Toc498783254)

[6 Utolsó simítások 39](#_Toc498783255)

[Irodalomjegyzék 40](#_Toc498783256)

[Függelék 41](#_Toc498783257)

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 a 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 az meghaladó 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 megalkotásához a mély neurális hálózatokat, illetve megerősítéses tanuláson alapuló algoritmusokat használtam fel.

A feladat megoldása során először egy egyszerűbb példán összehasonlítottam Matlabban és Pythonban az algoritmusok implementáláshoz szükséges eszköztárakat. Ezután megvalósításra került a feladathoz 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. Végül implementáltam és összehasonlítottam néhány a probléma sajátosságai szempontjából releváns mély megerősítéses tanuláson alapuló algoritmust.

Abstract

In the recent years, with the explosive growth of the computational power, the machine learning algorithms and the so called deep learning keep gathering more ground, whereby the reserachers are becoming able to succeed on more and more domain of technical problems. These algorithms nowadays can solve complex problems on a human levol or even beyond, like imagerecognition or playing such difficult games like Go.

The topic of this bachelor thesis is the possible application of these algorithms in the field of controll engineering, within it, the theoritical realisation of the control of a racing car. For the implementation of this, I used reinforcement learning based algorithms as well as deep neural networks.

While solving the task, first, using a simple example I compared the available task relevant toolboxes of Matlab and Python. Then, I designed and implemented the simulator framework later used for testing the algorithms. On this grafical user interface, the necessary parameters of the environment, the car and the contol algorithm can be set, as well as the progress of the learning can be observed. Finally, I implemented and compared the performance of some task relevant deep reinforcement learning based algorithms.

# Bevezetés

Manapság talán az egyik legfelkapottabb kutatási területnek a mesterséges intelligencia kutatása számít. Bár az alap ötletek és egyszerűbb 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-fikben látott emberi kinézettel és képességekkel rendelkező robot képétől, viszont bizonyos területekre, problémákra fokuszáltan a modern algoritmusok már sokszor túlszárnyalják az emberi teljesítményt 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 módszerek 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 a később 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 az angolnyelvű szakirodalom dominál, így gyakran fogom én is az angol szakkifejezéseket használni.

# Elméleti háttér

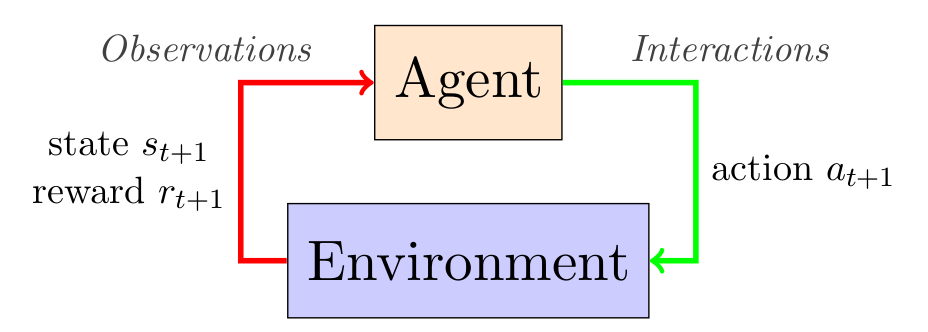
## Megerősítéses tanulás

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 ( angolul Markovian Decision Process, továbbiakban MDP). Ez egy döntéshozatalok modellezésére létrehozott matematikai modell, melynek többféle változata is létezik. Mi a továbbiakban csak a véges determinisztikus esettel foglalkozunk. Egy MDP-t a halmazok és függvények írnak le, ahol:

* ***S*** a környezet lehetséges állapotainak halmaza
* ***A*** a környezetben a lehetséges akciók halmaza
* ***P*** az állapot átmenetet leíró valószínűségi eloszlás függvény, ahol  ***S*** és  ***A****.* Determinisztikus esetben ez csak ez adott esetén csak egy értéknél 1, a többinél 0
* az állapotátmenetekért járó jutalom ahol,  ***S*** és  ***A***
* a diszkonttényező, ami a jelenlegi és a jövőbeni jutalom közti értékkülönbséget írja le



1. Markovi döntési folyamat

Az ágensnek van egy ***π(s)*** stratégiája (továbbiakban *policy*), amelyet követve minden  ***S*** állapothoz hozzárendel egy  ***A*** akciót (angolul *action*). Az döntést meghozva az következő állapotba jut és megkapja a környezettől az szerinti jutalmat. 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 állapot érték függvényt (angolul state value), ami azt mondja meg, hogy az adott 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 akció érték függvényt (angolul *action value function*), ami azt adja meg, hogy az állapotból , ***a*** döntés után a ***π(s)*** stratégiát követve mennyi a várható diszkontált kumulált jutalom.

vagy a Bellman egyenlet alapján:

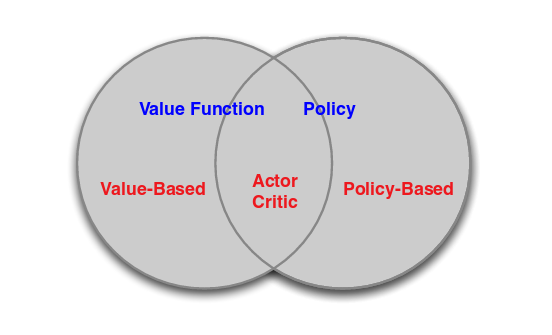
+

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, az egyes lépések után a kapott jutalom és a következő állapot megfigyelése után a *value iteration* (vagy *Bellman update*) segítségével garantáltan tudunk konvergálni felé.

Ez a megfontolás adja az alapját a legtöbb megerősítéses tanulás alapú algoritmusnak.

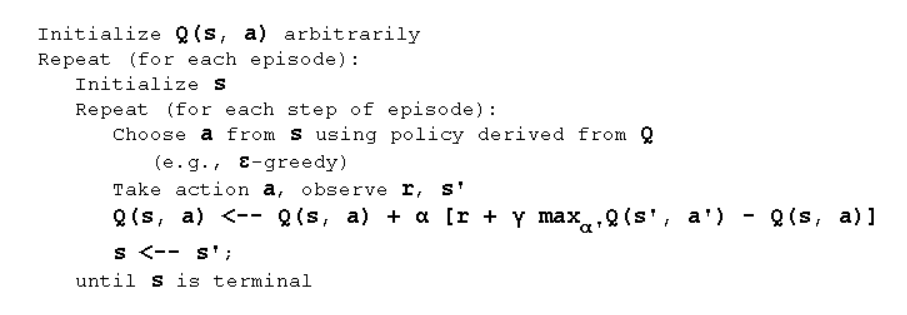
### Algoritmusok kategorizálása

A megerősítéses tanuláson alapuló algoritmusokat két nagyobb csoportra lehet osztani, modell független és a modell alapúra. A modell alapú algoritmusok mögötti elgondolás az, hogyha egy MDP összes összetevőjét ismerjük, tehát a állapotátmenet függvényt és az 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 valamelyik *valu*e függvényt vagy közvetlenük az 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.



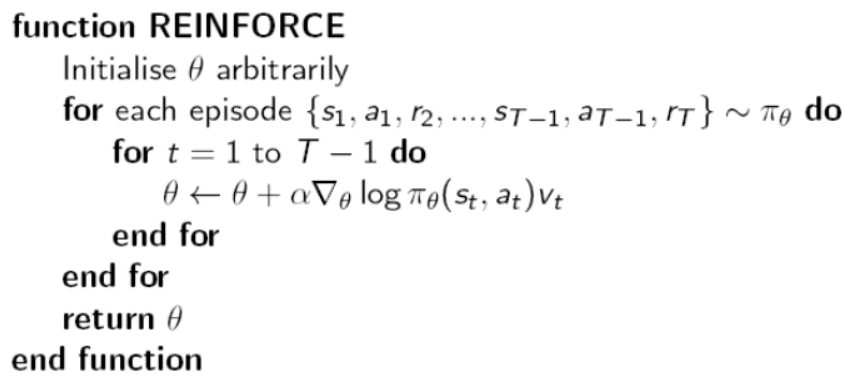
2. Megerősítéses tanuláson alapuló algoritmusok csoportosítása

Az egyik az ún. *value based*, amikor is az algoritmus különböző iteratív módszerekkel a függvényt közelíti. A függvény ismeretében közvetve megtanulja az optimális stratégiát is, hiszen ha mindig a maximális 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. Egy *value based algoritmus, a 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. Ezek működésük során ***π(s)*** *policy*-t követve megfigyelik a *policy* értékét az epizód végén, majd a *policy*-t leíró függvény paramétereit úgy hangolják, hogy ez az érték nőjön. Ehhez feltételezzük, hogy a *policy*-t valamilyen differenciálható függvény alkotja.



1. Egy *policy based* algoritmus, a Reinforce

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

### Exploration, exploitation dilemma

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. felderítő függvény (angolul *exploration function*) segítségével. Lehet bármilyen jó az algoritmus, ha egy az optimális *policy*-hoz elengedhetetlen állapotátmenetet 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, amihez az optimális stratégiát kell kövessünk, viszont ezt csak akkor tudjuk megtanulni, ha már megfelelő számú állapotátmenetet fedeztünk fel. Nehéz viszont megfogalmazni, hogy hol van az egyensúly a kettő között, különösen mivel ez minden problémakörnél mást jelent.

### Elterjedt felderítő függvények

A következőkben a teljesség igénye nélküli szeretnék bemutatni, két 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* *policy*-hoz.

Léteznek ezeken kívül más megközelítést alkalmazó módszerek is, illetve ezeknek különböző változatai, például szekvenciális problémák esetében, hogy minden diszkrét időpillanatra más *ε*-t használnak az *ε-greedy* esetén vagy más *T temperature*-t a softmaxnál.

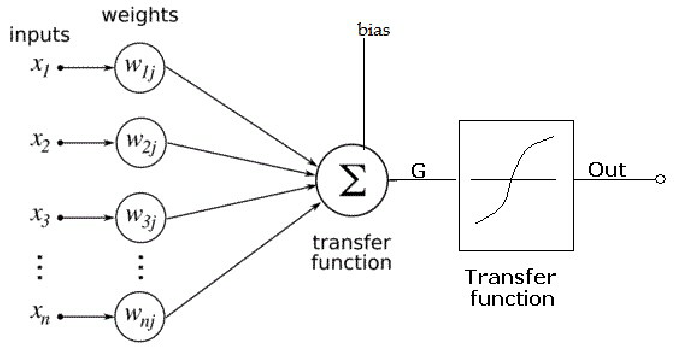
## Neurális hálózatok

A klasszikus megerősítéses tanuláson alapuló algoritmusok kiválóan működnek kisebb 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 függvényt vagy a stratégiát becsüljük. Erre manapság a legnépszerűbb megoldás a neurális hálók használata.

A gépi tanulást, és így a neurális hálózatokat is általában két fajta problémakör megoldásra szokták használni. A 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. Mivel a később használt algoritmusokkal egy regressziós problémát szeretnénk megoldani, a következő alfejezetekben, inkább az ezzel kapcsolatban releváns technikákra fogok kitérni.

### Neurális hálózatok felépítése

A neurális hálózat olyan irányított gráf, amelynek a szerkezetét az emberi agyban található neuronokról mintázták. A bementi rétegek és 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 szerkezet alapján, különböző súllyal számító élekkel. A neuronok továbbá tartalmaznak egy ún. aktivációs függvényt is. 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 mély neurális hálóknak (angolul *deep neural network*). 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 becsült kimenet és a valós kimenet különbségéből valamilyen hibafüggvényt számolunk és ezt a hibafüggvényt felhasználva a *backpropagation* nevű módszer 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.

. 

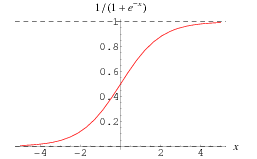
1. Feed forward neurális háló felépítése

### 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 a célra több függvény is megfelel, ezek közül szeretném ismertetni a legelterjedtebbeket.

**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 gyakran okozhatja az ún. eltűnő gradiens (angolul *vanishing gradient*) 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, mivel kiküszöböli az eltűnő gradiens problémát.



### 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 nagymértékben eltérnek (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 esetlegesen előforduló *outlier*-ek által okozott nagymértékű torzítást.

### Hálózat tanítása és optimalizálás

A hálózat tanítása a rendelkezésre álló tanító adatok és az ún. *backpropagation* algoritmus segítségével történik. Az algoritmus teljes levezetését mellőzve, most csak egy rövid koncepcionális összefoglalót adok az algoritmus működéséről, illetve a teljesség igénye nélkül bemutatok néhány a gyakorlatban gyakran használt optimalizációs algoritmust. A teljes levezetést ITT elérhető.

Az algoritmus során az adott bementre, amit általában kisebb kötegekben (angolul *mini-batch*) adunk a hálózatra, kiszámoljuk a becsült kimenetet -t majd ezzel és a valós kimenet különbségéből kiszámoljuk a hibafüggvény értékét. Ez után kiszámoljuk a hibafüggvény minden réteg minden súlyára vonatkozó parciális deriváltját és ezt felhasználva hangoljuk a hálózat súlyait. Ezekkel a grádiens alapú eljárásokkal, sajnos nem garantált, hogy a hibafüggvény globális minimumát érjük el, ellenben gyakran ennek a garanciának a hiánya nem okoz gondot. Továbbá több kiegészítő módszert is kidolgoztak már arra, hogy a lokális minimumokból kilendítsék a függvényt a tanítás során, mint például a momentum módszerv vagy a Nesterov momentum.

**STOCHASTIC GRDIENT DESCENT (SGD)**

Az egyik legrégebb óta alkalmazott optimalizációs eljárás. A *backpropagation* során kiszámolt parciális deriváltakat egy ***α*** tanulási rátával (továbbiakban *learning rate*),majd ezt kivonva az eredeti súlyokból frissítjük azokat, majd ezt az eljárást iteráljuk.

Ahol hálózat éleinek súlyai, a *learning rate,* n az alkalmazott *mini-batch* mérete, pedig az i.-edik tanítóadatot alkotó be és ki menet.

A fentebb említett lokális minimumba való beragadás elkerülésére találták ki az ún. momentum alkalmazását. Ezt annyit tesz, hogy a súly finomhangolása során az előző iterációban kiszámolt korrigáció a jelenlegi iterációban is jelen van valamekkora mértékben, ezzel bizonyos mértékben potenciálisan túllendítve az algoritmust az esetleges lokális minimumban való elakadástól.

Ahol az , ami azt mondja meg, hogy mennyire számítsanak bele a mostaniba a régebbi iterációk során kiszámol ***v*** értékek.

A**DAM**

Az ADAM is a SGD-nél megismert eljárásokon alapul, néhány kiegészítéssel. Az algoritmus teljes működésének leírása ITT érhető el. Fontos tulajdonsága, hogy minden élhez saját *learning rate*-et tart nyilván amelyeket, attól függően adaptívan változtat, hogy az adott él milyen gyakran volt frissítve az előző iterációk során. A gyakorlatban nagyon közkedvelt optimalizációs eljárás, mivel általában kevés finomhangolást igényel és gyakran ennek a használata jelenti a leggyorsabb és legpontosabb eredményt.

**NADAM**

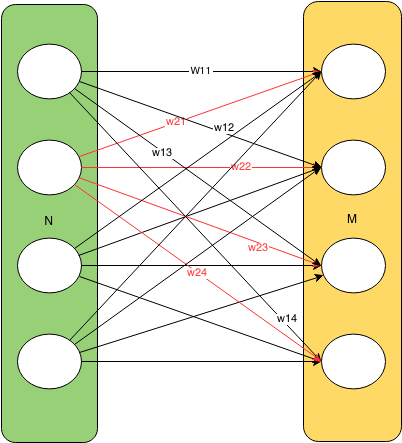
Az ADAM algoritmusnak a Nesterov momentummal kiegészített változata.

### 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.

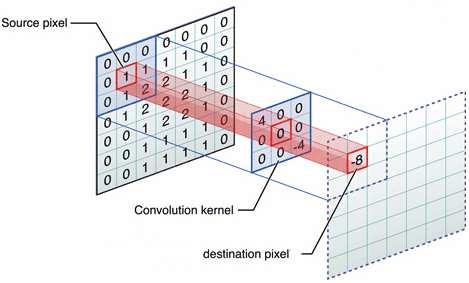
**FULLY CONNECTED**

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 *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.



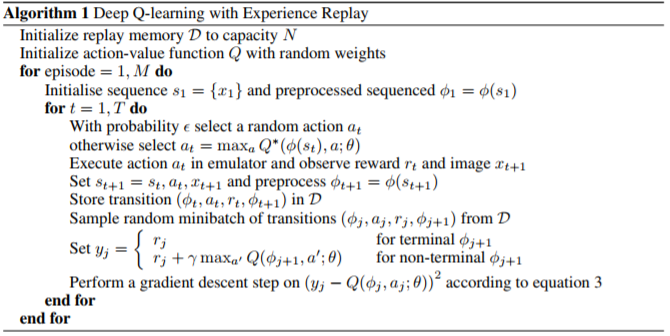
## Deep reinfocement learning

### Motiváció

Bár a klasszikus megerősítéses tanulás alapú algoritmusok kis állapotterü problémák esetén jól használhatók az állapottér és a akciótér dimenzióinak növelésével hamar el lehet érni a módszer korlátait. A modern megerősítéses tanuláson alapuló algoritmusok ma már szinte mind mély tanulással vannak kombinálva a hatalmas állapottér kompenzálására. Az ezzel járó előnyökkel viszont sajnos hátrányok is járnak. Míg a klasszikus táblázatos módszert alkalmazó algoritmusok esetében megfelelő felderítő függvény alkalmazása esetén garanciánk van arra, hogy az algoritmus az optimális *policy*-hoz 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 játékok egy részében csupán a képernyő pixeleit felhasználva, mint bemenet. Ez egy *Q learning*-hez hasonló algoritmus, azzal a különbséggel, hogy a függvényt egy neurális hálóval közelíti. 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 kötegnyi (*mini-batch*) 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 *target* értéket tesszük, és erre tanítjuk rá. Sajnos mivel a -t is a neurális hálóból segítségével nyerjük ki, ezért a rendszer alapesetben nagyon instabil. Erre az ún. *target network* használatát találták ki, ami azt jelenti, hogy a értéket egy másik háló segítségével nyerjük ki amit ***τ****-*lépésenként szinkronizálunk az eredetivel.



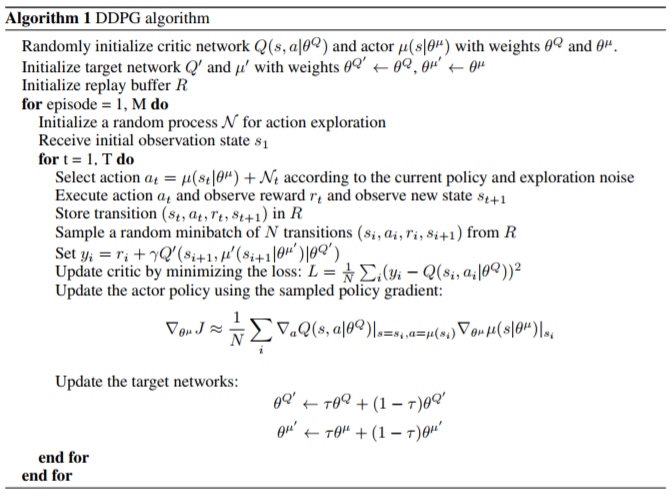
1. DQN algoritmus *experience replay-*el

### 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* algoritmus nyújt segítséget. Itt két hálót tanítunk párhuzamosan. Az egyikkel azt határozzuk meg, hogy a következő állapotból mi lenne az optimális akció, míg 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, ezt fel tudjuk használni ehhez a változtatáshoz is. A döntéshozásnak és a döntés értékelésének ez a fajta szétválasztása bizonyítottan csökkenti a ***Q*** értékek *offset*-ét, ezáltal stabilabb működést kapunk.

### DDPG

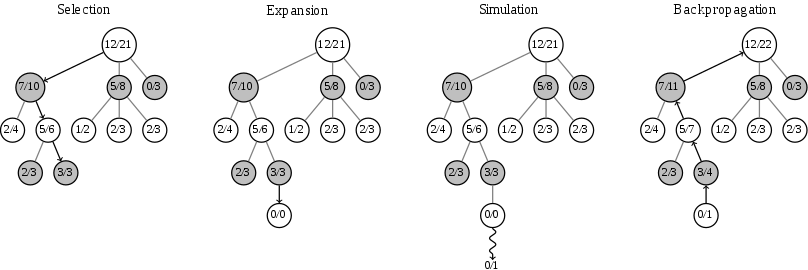
Míg a DQN és a DDQN algoritmusokat diszkrét kimenetek esetén tudjuk alkalmazni, a legtöbb irányítási algoritmusnál folytonos beavatkozó jelre van szükségünk, így más módszert kell keressünk. Egy ilyen,folytonos mozgástérre alkalmazható algorimtus a megoldást a *Deterministic Deep Policy Gradient* módszer (továbbiakban DDPG). Ez egy *actor-critic* típusú algoritmus, tehát két fő részből áll. Egy *actor* hálózatból, ami az ideális *policy*-t 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, ezért az eddig megismert módszerek nem mindig 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*). Az algoritmus során addig követjük a 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 a szimulációk során meghozott döntés sorra, majd az így kapott eredmények segítségével az eddig bizonytalan helyzetből azt a döntést hozzuk meg, ami átlagosan a legjobb eredményre vezetett, így a döntési fában egy szinttel lejjebb 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 *mély tanulás* 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:

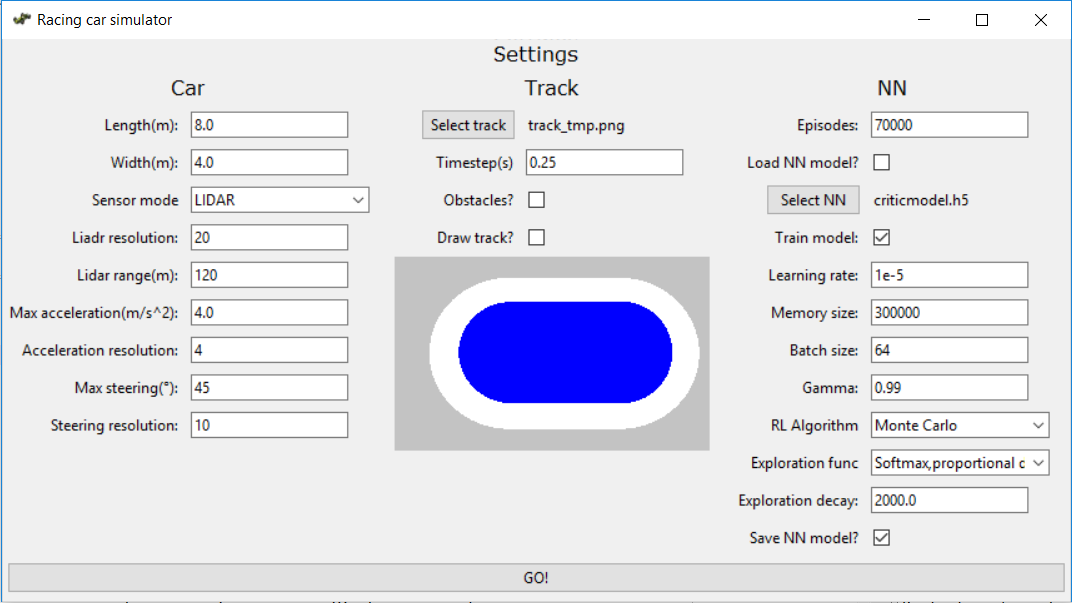
* Hosszúság
* Szélesség
* Szenzor rendszer típusa (lidar)
* Lidar felbontása
* Lidar maximális látótávolsága
* Maximális gyorsulás
* Maximális kormányszög
* Gyorsulás felbontása (a diszkretizáláshoz)
* Kormányszög felbontása (a diszkretizáláshoz)

A pályára vonatkozó beállítások:

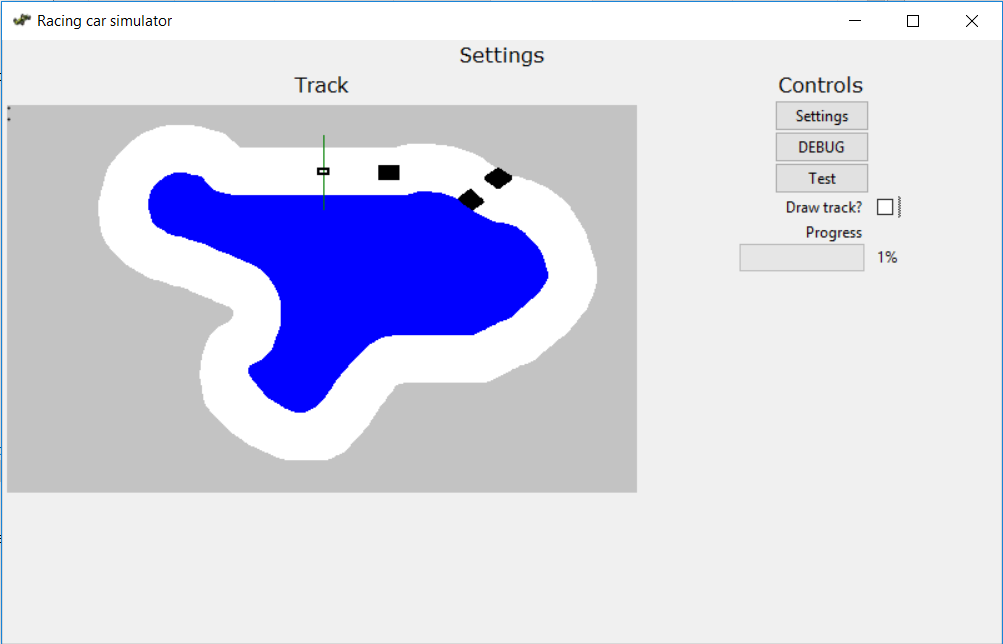
* Pálya kiválasztása
* Statikus akadályok használata
* A tanítás során a pálya kirajzolása
* Alkalmazott időköz

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

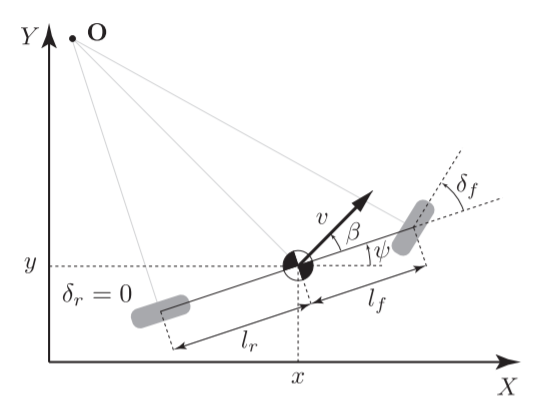
A megtapasztalt állapotátmenetek és az értük járó jutalmak tárolására szolgáló experience replay memory-t a Pythonhoz írt egy modul, a Pandas segítségével valósítottam meg. Ez egy adatok kezelésére írt open source package, amely kiválóan alkalmas az adatok tárolására és analizálására. Az itt tárolt adatok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