

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Fehér Gergő

Versenyautó pályaívének tervezése mesterséges intelligencia módszerekkel

Konzulens

BUDAPEST, 2017

Tartalomjegyzék

[Összefoglaló 5](#_Toc499330620)

[Abstract 6](#_Toc499330621)

[1 Bevezetés 7](#_Toc499330622)

[2 Elméleti háttér 8](#_Toc499330623)

[2.1 Megerősítéses tanulás 8](#_Toc499330624)

[2.1.1 Markovi döntési folyamat 8](#_Toc499330625)

[2.1.2 Algoritmusok kategorizálása 10](#_Toc499330626)

[2.1.3 Exploration, exploitation dilemma 12](#_Toc499330627)

[2.1.4 Elterjedt felderítő függvények 13](#_Toc499330628)

[2.2 Neurális hálózatok 13](#_Toc499330629)

[2.2.1 Neurális hálózatok felépítése 14](#_Toc499330630)

[2.2.2 Aktivációs függvény szerepe és főbb típusai 15](#_Toc499330631)

[2.2.3 Költség függvények 17](#_Toc499330632)

[2.2.4 Hálózat tanítása és optimalizálás 18](#_Toc499330633)

[2.2.5 Neurális hálózatok típusai 19](#_Toc499330634)

[2.3 Deep reinforcement learning 21](#_Toc499330635)

[2.3.1 Motiváció 21](#_Toc499330636)

[2.3.2 DQN 21](#_Toc499330637)

[2.3.3 DDQN 22](#_Toc499330638)

[2.3.4 DDPG 23](#_Toc499330639)

[2.3.5 Monte Carlo Tree Search 24](#_Toc499330640)

[3 Megerősítéses tanulás a gyakorlatban 26](#_Toc499330641)

[3.1 Kő-papír-olló megerősítéses tanulással 26](#_Toc499330642)

[3.2 Kő papír-olló neurális hálóval 27](#_Toc499330643)

[3.3 Értékelés 29](#_Toc499330644)

[4 Versenyautó irányításának tervezése és megvalósítása 30](#_Toc499330645)

[4.1 Szimulátor 30](#_Toc499330646)

[4.1.1 Grafikus felület 30](#_Toc499330647)

[4.1.2 Szimulátor felépítése 33](#_Toc499330648)

[4.2 Kinematikai modell 34](#_Toc499330649)

[4.3 Elméleti megfontolások 36](#_Toc499330650)

[4.3.1 Bemenetek 36](#_Toc499330651)

[4.3.2 A jutalom függvény 36](#_Toc499330652)

[4.3.3 Alkalmazott felderítő módszer 37](#_Toc499330653)

[4.3.4 Hálózat szerkezete 38](#_Toc499330654)

[4.4 Implementált algoritmusok 40](#_Toc499330655)

[4.5 Felmerült akadályok 43](#_Toc499330656)

[4.5.1 Optimalizálás 43](#_Toc499330657)

[4.5.2 Teljesítmény 43](#_Toc499330658)

[4.5.3 Megfelelő jutalomfüggvény megválasztása 44](#_Toc499330659)

[4.5.4 Hiperparaméter optimalizáció 44](#_Toc499330660)

[4.5.5 Komplexitás 44](#_Toc499330661)

[4.6 Eredmények értékelése 45](#_Toc499330662)

[4.7 Lehetséges továbbfejlesztések 48](#_Toc499330663)

[5 Összefoglaló 51](#_Toc499330664)

[Irodalomjegyzék 52](#_Toc499330665)

[Függelék 55](#_Toc499330666)

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. 24.

...…………………………………………….

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 azt meghaladó szinten is, mint például a képfelismerés vagy akár olyan komplex játékokkal játszani, mint a Go [1].

Jelen szakdolgozat témája, ezen algoritmusoknak az irányítástechnikában való lehetséges alkalmazása, azon belül egy versenyautómodell irányításnak elméleti megvalósítása. Ennek megalkotásához 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 level or even beyond, like image recognition or playing so difficult games as Go.

The topic of this bachelor thesis is the possible application of these algorithms in the field of control 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 control 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 alapötletek és egyszerűbb algoritmusok már régóta léteznek, sokáig hiányzott a tudósok és kutatók 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 [1]. 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 tanulásnak (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

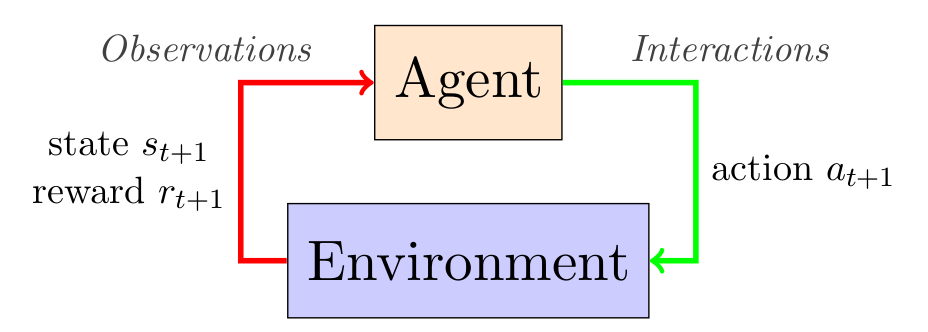
A megerősítéses tanulás a gépi tanulásnak egy olyan vállfaja, melyet a viselkedés pszichológia ihletett. Arra a problémára keresi a megoldást, hogy egy ágens az adott környezetben milyen döntéseket hozzon annak érdekében, hogy az azért kapott jövőbeli kumulált jutalmat maximalizálja.

### Markovi döntési folyamat

A megerősítéses tanulás témakörébe eső problémák gyakran modellezhetők az ún. markovi döntési folyamattal ( angolul *Markovian Decision Process*, továbbiakban MDP [2]). 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 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



2.1. ábra A Markovi döntési folyamat [3]

Egy ilyen MDP-ben mozgó 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 ágensnek 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.

|  |  |
| --- | --- |
|  | (2.1) |

Ahogy a való életben is, az ágensünk számára is kevésbé értékes a jövőbeni jutalom, így azok értékét a számításkor egy ***γ*** értékkel diszkontáljuk (2.1) szerint. 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.

|  |  |
| --- | --- |
|  | (2.2) |

vagy a Bellman egyenlet alapján:

|  |  |
| --- | --- |
|  | (2.3) |

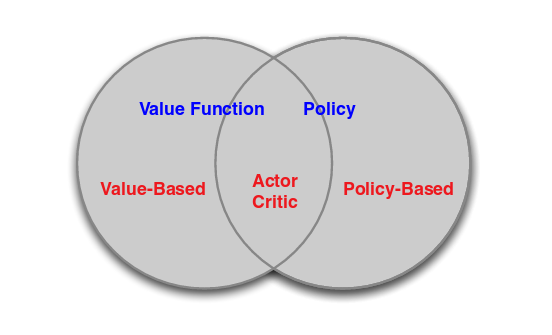
Azt a ***π(s)*** függvényt keressük, ahol a értékek maximálisak, ezt -al szoktuk jelölni. Ez az optimális stratégia. Elméletben, ha ismerjük ezt a függvényt minden állapotra és döntésre, akkor a kumulált jutalom maximalizálásához egyszerűen minden állapotban azt a döntést kell választani, ahol a függvény 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 [4].

|  |  |
| --- | --- |
|  | (2.4) |

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

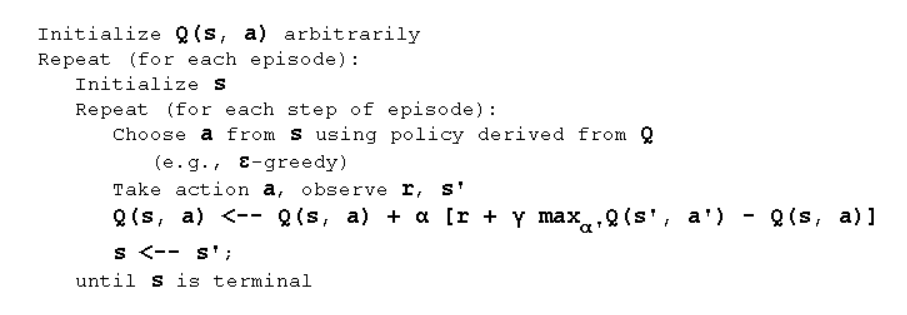
### Algoritmusok kategorizálása

A megerősítéses tanuláson alapuló algoritmusokat két nagyobb csoportra lehet osztani, modellfüggetlenre 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 maximalizálja [4]. Ez alapján az ágens előbb megpróbálja felfedezni a környezetet és modellt építeni, amely megmondja az egyes állapotátmeneteket és az azokért járó jutalmat, majd ezen modellre számolja ki az optimális stratégiát. A modellfüggetlen 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, amit a 2.2.-es ábra jól szemléltet.



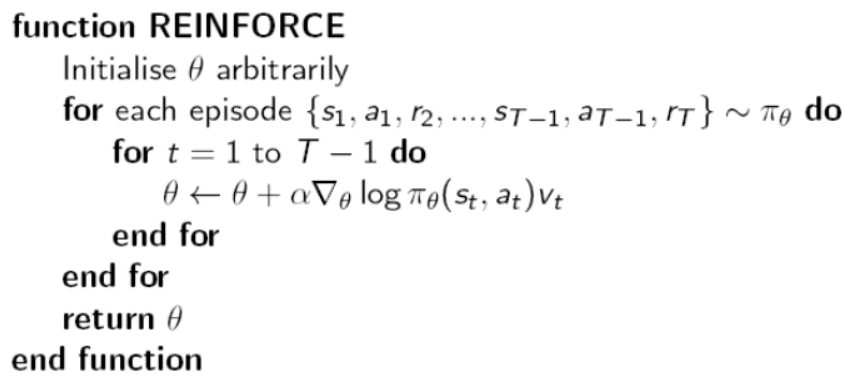
2.2. ábra Megerősítéses tanuláson alapuló algoritmusok csoportosítása [5]

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 [4]. Ennek a módszernek a hátránya viszont, hogy gyakran az algoritmus sok időt tölt szuboptimális stratégiák értékelésével, és olyan állapot átmenetek Q értékeinek becslésével, amit lehet, hogy az optimális stratégiát követve nem is érint. Ebbe a kategóriába tartozik például a már klasszikusnak számító Q-learning algoritmus, amit a 2.3.-as ábrán látunk.



2.2.3. ábra A Q-learning algoritmus [6]

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. Egy ilyen algoritmus a 2.4.-es ábrán található Reinforce elnevezésű algoritmus.



2.4. ábra A Reinforce algoritmus [5]

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, melyek közül néhányat 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-t a tanulás előre haladásával folyamatosan csökkentjük, így a döntéshozás konvergál a **π(s)** *policy-*hoz. 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 használata, ami az egyes döntésekhez az alábbi valószínűségi eloszlást rendeli hozzá.

|  |  |
| --- | --- |
|  | (2.5) |

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 külön *ε*-t használnak az *ε-greedy* esetén vagy más *T temperature*-t a softmaxnál.

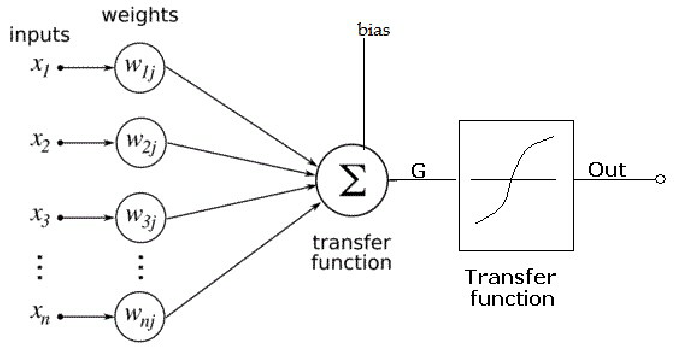
## Neurális hálózatok

A klasszikus megerősítéses tanuláson alapuló algoritmusok kiválóan működnek kisebb véges diszkrét állapot és akcióterekben, 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 mesterséges rendszer, amelynek a szerkezetét az emberi agyban található neuronokról mintázták. A bementi rétegek és kimeneti rétegek között ún. rejtett rétegek helyezkednek el. Ezeket a rétegeket a neuronok 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.



2.5. ábra Egy mesterséges neuron felépítése [7]

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.

. 

2.6. ábra Feedforward neurális háló felépítése [8]

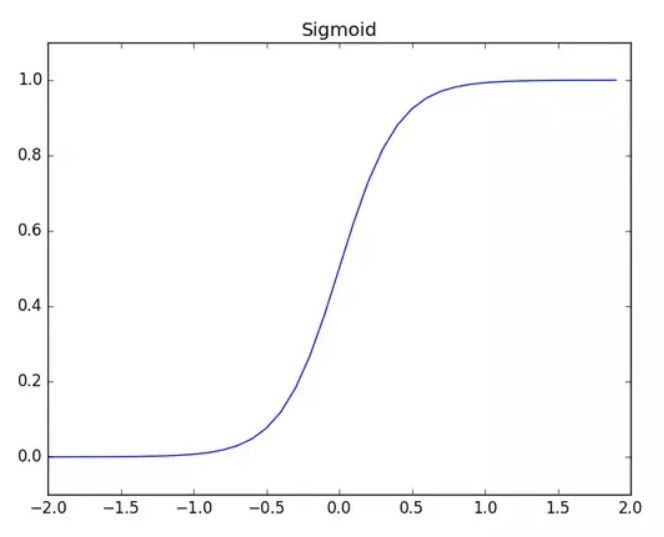
### Aktivációs függvény szerepe és főbb típusai

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.

|  |  |
| --- | --- |
|  | **(2.6)** |

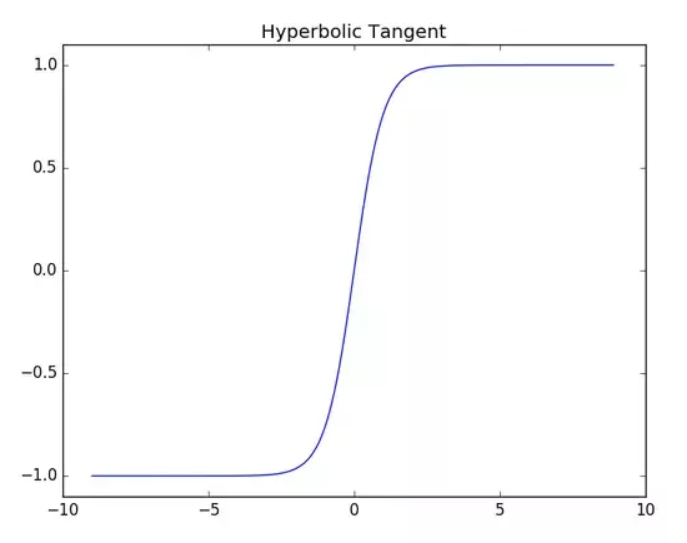


2.7. ábra A sigmoid függvény [9]

**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.

|  |  |
| --- | --- |
|  | **(2.7)** |



2.8. ábra A tanh függvény [9]

**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.

|  |  |
| --- | --- |
|  | **(2.8)** |



2.9. ábra A ReLu függvény [9]

### 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 AVERAGE ERROR**

Itt egyszerűen kiszámoljuk a becsült értékek és a valós értékek különbségét, majd ennek vesszük az átlagát.

|  |  |
| --- | --- |
|  | (2.9) |

**MEAN SQUARED ERROR**

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

|  |  |
| --- | --- |
|  | (2.10) |

**HUBERT LOSS**

Ha a tanító adataink között vannak hibás adatok, amik a valós adatoktól nagymértékben eltérnek (angolul *outlier*) az MSE hibafüggvény használata esetén nagyon eltorzíthatják a tanítást. Erre egy lehetséges megoldás a HUBERT LOSS használata, ami meghatározott intervallumon belül négyzetesen, azon kívül lineárisan veszi figyelembe a hibát, így kiküszöböli az esetlegesen előforduló *outlier*-ek által okozott nagymértékű torzítást.

|  |  |
| --- | --- |
|  | (2.11) |

### Hálózat tanítása és optimalizálás

A hálózat tanítása a rendelkezésre álló tanító adatok és az ún. *backpropagation* algoritmus segítségével történik. Az algoritmus teljes levezetését [10] mellőzve, most csak egy rövid koncepcionális összefoglalót adok az algoritmus működéséről, illetve a teljesség igénye nélkül bemutatok néhány a gyakorlatban gyakran használt optimalizációs módszert.

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 gradiens 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.

**STOCHASTIC GRDIENT DESCENT (SGD)**

Az egyik legrégebb óta alkalmazott optimalizációs eljárás. A *backpropagation* során kiszámolt parciális deriváltakat egy ***α*** tanulási rátával (továbbiakban *learning rate*) beszorozzuk, majd ezt kivonva az eredeti súlyokból frissítjük azokat. Ezt az eljárást iteráljuk addig, ameddig a hiba nem csökken az megszabott határérték alá.

|  |  |
| --- | --- |
|  | (2.12) |

Ahol a 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.

|  |  |
| --- | --- |
|  | (2.13) |
|  | (2.14) |

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

A**DAM**

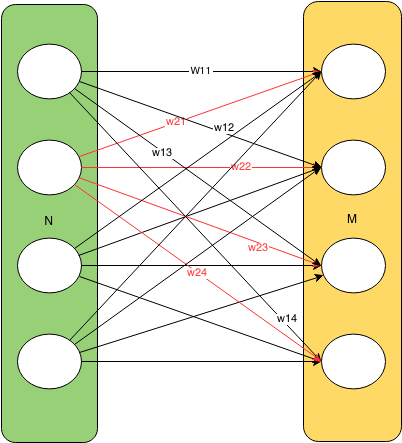
Az ADAM [11] is az SGD-nél megismert eljárásokon alapul, néhány kiegészítéssel. Fontos tulajdonsága, hogy minden élhez saját *learning rate*-et tart nyilván amelyeket, attól függően változtat adaptívan, hogy az adott él milyen gyakran volt frissítve az előző iterációk során. A gyakorlatban nagyon közkedvelt optimalizációs eljárás, mivel általában kevés finomhangolást igényel és gyakran ennek a használata jelenti a leggyorsabb és legpontosabb eredményt.

### 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**

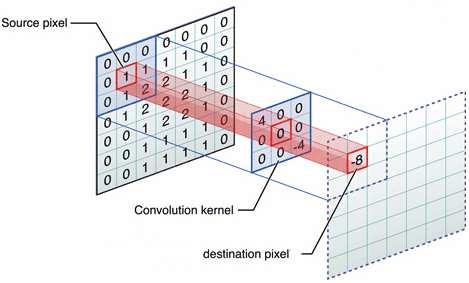
A *fully connected* vagy magyarul teljesen összekötött rétegek között minden neuron minden neuronnal össze van kötve. A legegyszerűbb neurális hálók ilyen rétegek egymás után pakolásával épülnek fel. A bemeneti adatok és a kimeneti adatok közötti kapcsolatot tanulja meg.



2.10. ábra Egy *fully connected* réteg felépítése [12]

**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.



2.11. ábra A konvolúciós réteg működése [13]

**Pooling rétegek**

Az ún *pooling* rétegek a bemeneti 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.



2.12. ábra A *maxpooling* réteg működése [14]

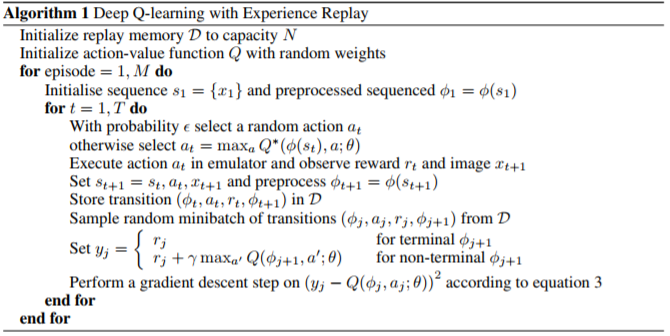
## Deep reinforcement learning

### 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 [4], 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 ez jellemzően 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 2.23.-as ábrán látható 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 [15]. Ez egy *Q learning*-hez hasonló algoritmus, azzal a különbséggel, hogy a függvényt egy neurális hálóval közelíti. Mivel a megtapasztalt állapotátmenet és jutalom eldobása rögtön a tanítás után nem lenne túl effektív, ezeket egy ún. *experience replay* memóriában tárolják. Ebből vesznek ki valamilyen eloszlással egy kötegnyi (angolul *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 csak a neurális hálóból segítségével tudjuk kinyerni, 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.



2.13. ábra 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 [16]. 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 csökkenti a ***Q*** értékek *offset*-ét, ezáltal stabilabb működést kapunk. Így összességében az algoritmus a 2.13.-as ábrán bemutatott DQN-től az alábbi egyenlet szerint tér el:

|  |  |
| --- | --- |
|  | (2.15) |

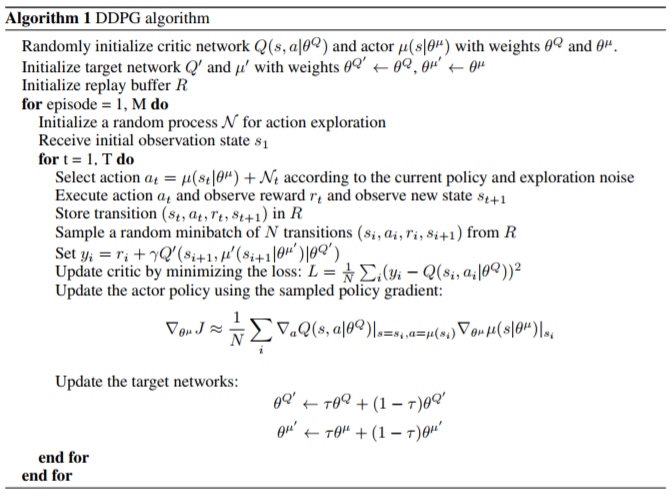
helyett

|  |  |
| --- | --- |
|  | (2.16) |

,ahol a hálózatunk paraméterei, pedig a *target* hálózatunk paraméterei, amit bizonyos időközönként szinkronizálunk a -val.

### 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 algoritmus a *Deep Deterministic Policy Gradient* módszer (továbbiakban DDPG [17]). 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. A teljes algoritmus a 2.14.-es ábrán láthatjuk.

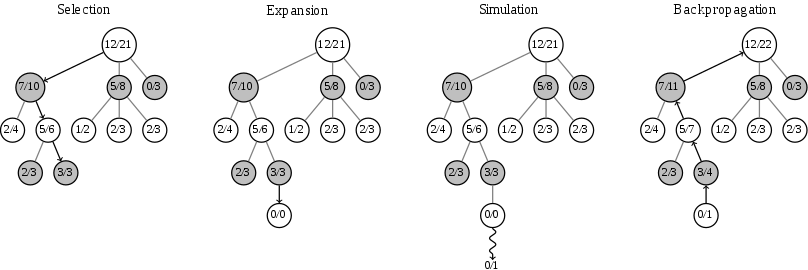


2.14. ábra A DDPG algoritmus [17]

### 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 [1].

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.



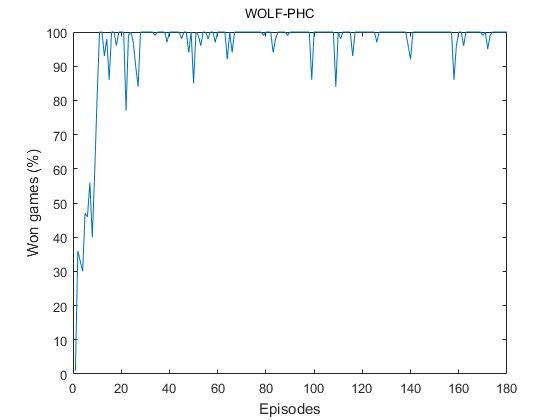
2.15. ábra A Monte Carlo Tree Search főbb lépései [18]

# Megerősítéses tanulás a gyakorlatban

A versenyautó irányításának megtervezése előtt egy egyszerűbb problémán teszteltem az algoritmusokat. Az tervezés 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

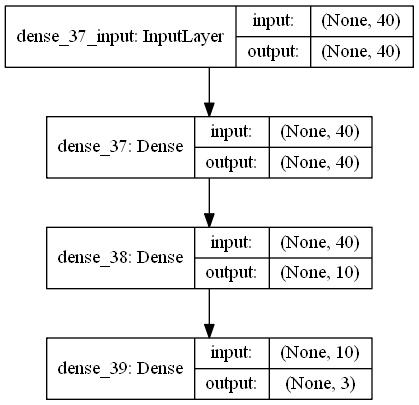
A közismert kő-papír-olló játékot két játékos játssza. A játékosok három különböző döntés közül választhat. A kő üti az ollót, a papír üti, a követ, és az olló üti a papírt. Egy ilyen környezetben egymás után több mecset is játszva a játékot modellezhetjük egy MDP-vel is, ha az ellenfelünk valamilyen statikus stratégiát követve játszik, például, hogy az utolsó játék kimenete alapján dönt. Egy ilyen MPD-ben pedig a klasszikus megerősítéses tanuláson alapuló algoritmusok garantáltan rátanulnak az optimális stratégiára. A teszthez egy a dolgozatban nem ismertettet algoritmust, a WOLF-PHC-t [19] implementáltam Matlab segítségével. ε-greedy felfedező stratégiát alkalmaztam. Továbbá a diszkont faktort, γ-át 0,9-re választottam. Az ε és a *learning rate* értékét az alábbi összefüggések szerint változtatom:



3.1. ábra A Matlabban implementált Wolf-PHC algoritmus által elért eredmények mozgóátlaga

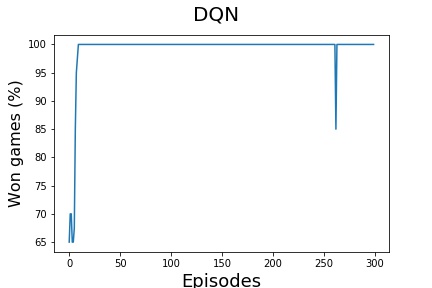
## Kő papír-olló neurális hálóval

Bár egy ilyen kis állapotterű probléma nem igényel feltétlenül egy neurális háló alapú algoritmust, a később használni kívánt keretrendszer kiválasztásának eldöntésére, implementáltam a 2.3.2-es és 2.3.3-as fejezetben bemutatott DQN és DDQN algoritmust. A teszt során 300 epizódig tanítottam a hálót 32 méretű *mini-batch*-ekkel, softmax felderítő függvényt használva, ahol a T *temperature* változó 2-ről indul és lépésenként 1/200-dal csökkent. Egy epizód 40 lejátszott meccset jelentett. Ahol egy nyerés +1 minden más -1 jutalmat jelentett. A *learning rate*-et 0.01-re választottam, illetve a γ-t 0.99-re. A *target network*-öt 10 lépésenként aktualizálom és növelem az 1,2-szeresére. A háló szerkezetére egy egyszerű előrecsatolt *fully connected* hálót választottam két rejtett réteggel egyenként 40 és 10 neuronnal, illetve relu aktivációs függvénnyel. A pontos felépítést a 3.2.-es ábrán láthatjuk.

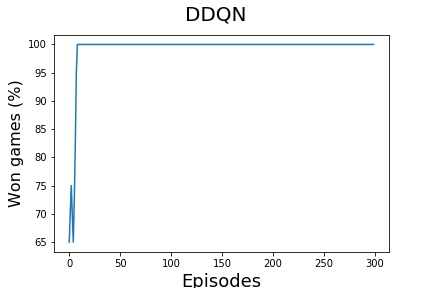


3.2. ábra Az alkalmazott neurális háló felépítése

Ebben a problémakörben az algoritmusok szinte azonnal rátanulnak az egyszerűbb stratégiákra ahogy azt a 3.3.-es és 3.4.-es ábrán is látjuk, illetve a bonyolultabbak se tartanak sokkal tovább. Az alábbi grafikonon a 40 meccsenkénti százalékos eredményt láthatjuk, egy olyan stratégiával szemben, ami mindig, azt lépi, ami a hárommal ezelőtti játékon lett volna a nyertes stratégia.

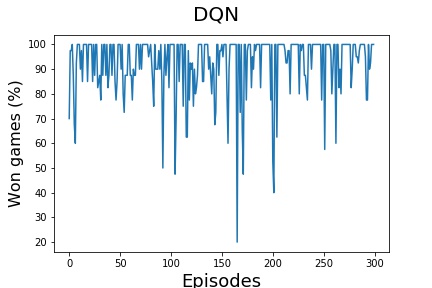


3.3. ábra A Pythonban implementált DQN algoritmus eredményeinek mozgóátlaga



3.4. ábra A Pythonban implementált DDQN algoritmus eredményeinek mozgóátlaga

Kipróbáltam továbbá az algoritmust *target network* használata nélkül is. Az alábbi grafikonon láthatjuk, hogy valóban jelentősen romlott a teljesítménye, és ha valamit megtanul, azt is nagyon bizonytalanul.



3.5. ábra A DQN algoritmus által elért eredmények mozgóátlaga *target network* használata nélkül

## 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 mivel a Python és így a hozzá írt modulok *open source* licence alá tartoznak, sokkal nagyobb online közösséggel rendelkeznek a mély tanulás területén, így a munka közben felmerülő kérdésekre, problémákra gyorsabban megoldást lehet találni a különböző fórumokon. Ezek miatt a megfontolások miatt a versenyautó irányításához implementált algoritmusokat és a tesztelésére létrehozott szimulátort Pythonban valósítottam meg.

# Versenyautó irányításának tervezése és megvalósítása

A feladat egy olyan algoritmus megvalósítása, amely a tanulási fázis után viszonylag tetszőleges alakú pályán a lehetőségekhez mérten valamilyen szuboptimális íven végig vezeti az autót a célig. Ehhez minden diszkrét időpillanatban a bemeneti adatai alapján megadja az autó irányításához szükséges kormányszöget és gyorsulást.

Az algoritmussal szemben elvárás, hogy a tanítás után képes legyen az autó végig haladni a pályán anélkül, hogy falnak vagy a pályán található akadályoknak ütközne. Lehetőség szerint a bemeneti szenzor adatokat és a kimenetet, úgy kell megválasztani, hogy miután az egy kellően általános pályán betanult, képes legyen egy másik hasonló pályán is végig menni.

## Szimulátor

Az algoritmus és a fizikai környezet számos paraméterrel rendelkezik, így szükséges volt egy grafikus felület megvalósítása, amelyen ezen paraméterek interaktívan változtathatók és nyomon követhetőek. Továbbá szükség van a környezet és az autó modell helyes működésének ellenőrzésére, a szimuláció aktuális állapotának megjelenítésére, mely futás időben ki és bekapcsolható igény szerint ezzel csökkentve a pálya kirajzolása okozta futásiidő növekedést. A tanítás közben, illetve után el kell tudjuk menteni a betanult modelleket, amiket később szükség esetén vissza kell tudjunk tölteni az algoritmusba. Ezen felül az eredmények későbbi kiértékelése céljából szükséges az adatok valamilyen nyomon követése, és egy logfájlban való kimentése. A következő alfejezetekben röviden bemutatom, hogy a megalkotott grafikus felület milyen funkciókkal rendelkezik, és hogy épül fel. A részletesebb használati útmutató és az indításhoz szükséges Python modulok verziói a Függelékek között találhatóak.

### Grafikus felület

A tanításhoz implementált szimulátor grafikus felhasználói felülete két részből áll. A programot elindítva rögtön a 4.1-es ábrán látható Settings oldalra kerülünk, ahol beállíthatjuk a pályára, az autóra illetve az irányítás mögötti algoritmusra vonatkozó paramétereket.

Az autóra vonatkozó főbb beállítások:

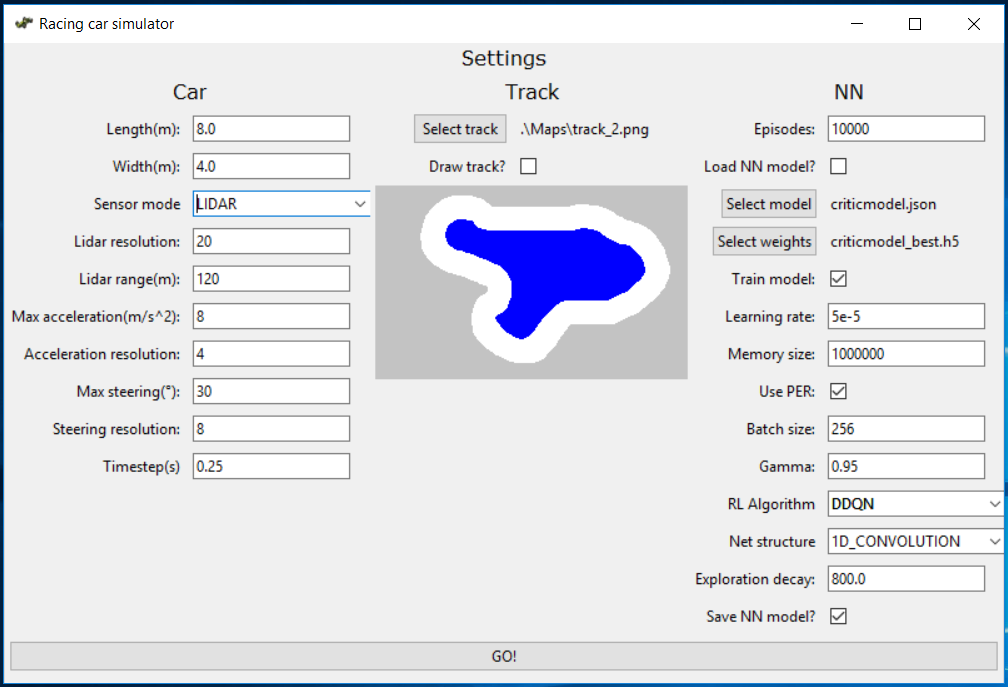
* Hosszúság
* Szélesség
* Szenzor rendszer típusa (LIDAR)
* LIDAR felbontása
* LIDAR maximális látótávolsága
* Maximális gyorsulás
* Maximális kormányszög
* Gyorsulás felbontása (a diszkretizáláshoz)
* Kormányszög felbontása (a diszkretizáláshoz)
* Alkalmazott időköz

A pályára vonatkozó beállítások:

* Pálya kiválasztása
* A tanítás során a pálya kirajzolása

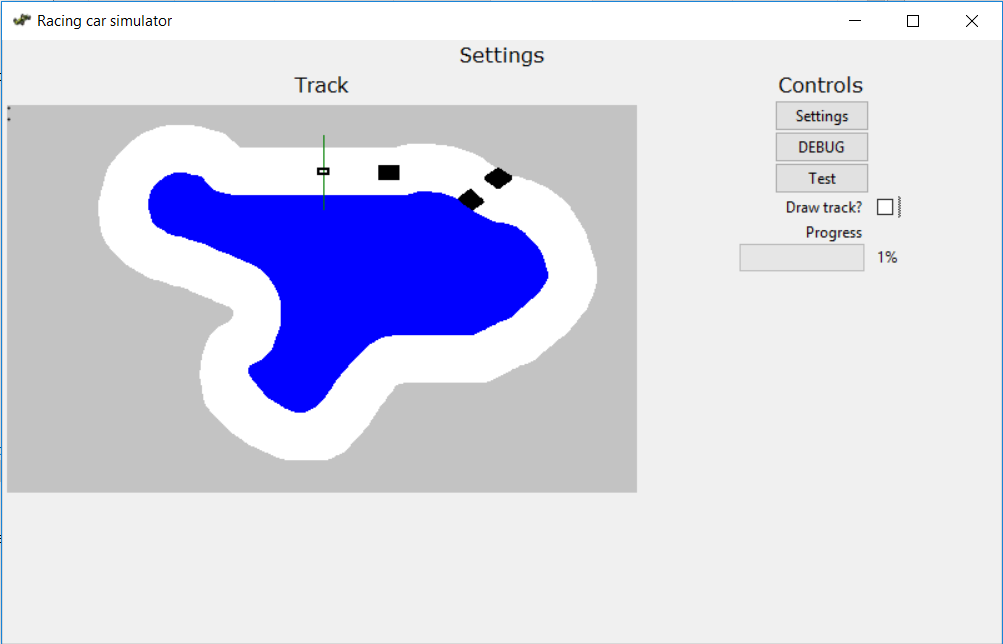
Az vezérlő algoritmus paraméterei:

* Epizódok száma
* Tanítás engedélyezése
* Learning rate
* Algoritmus kiválasztása
* Neurális háló modell betöltése
* Neurális háló súlyainak betöltése
* Neurális háló szerkezete
* Neurális háló mentése a tanítás végén
* *Prioritised experience replay* használata
* Exploration decay mértéke (1/n)
* Diszkont faktor
* Mini batch mérete
* *Experience replay* memória mérete



4.1. ábra A szimulátor beállítások menüje

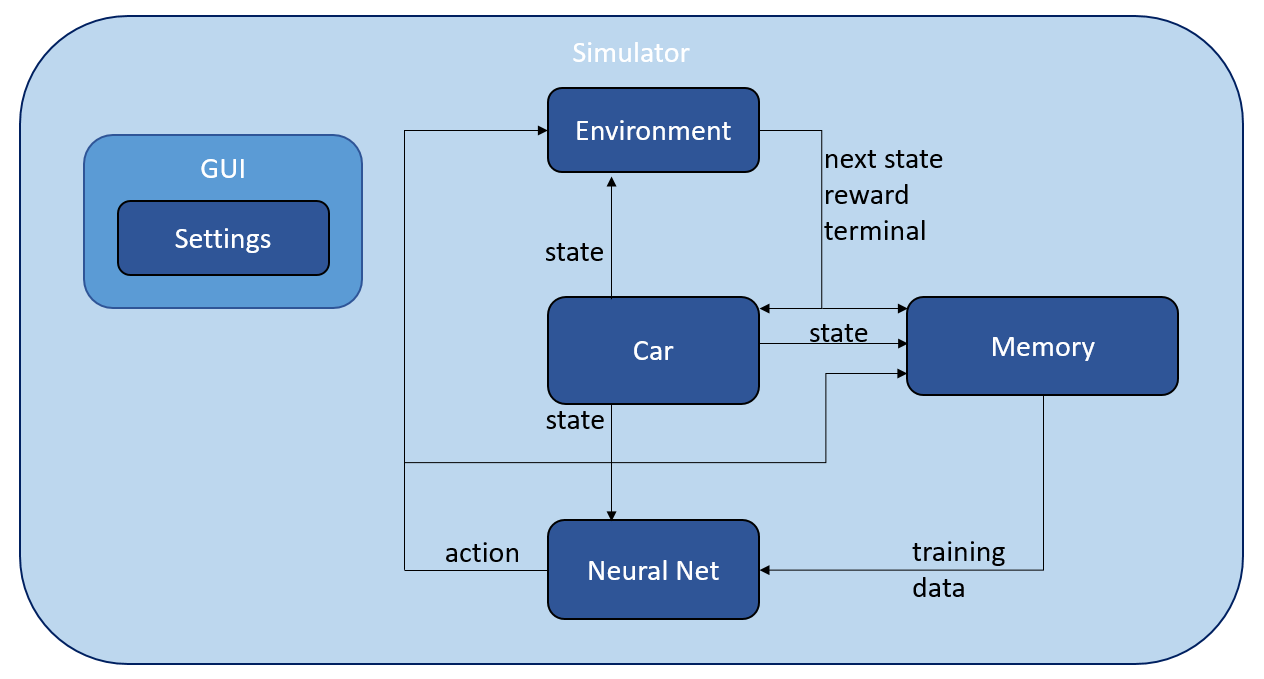
A szükséges beállítások után a *GO* gombra kattintás indítja a tanítást, és automatikusan átvált a szimulátor oldalra. Ez két részből áll. Az oldal bal felén található a pálya, ahol nyomon tudjuk követni, hogy az autó jelenleg milyen ívet követ. Az oldal jobb oldalán pedig egyéb kontrol gombok találhatóak. 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 automatikusan 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 leggyorsabban teljesített teszt kör eredményt, illetve, hogy ez melyik epizódban történt. A Draw track *checkbox* a pálya és az autó kirajzolását állítja, így annak beikszelése után nyomon tudjuk követni, hogy az autó éppen milyen íven halad. Bekapcsolt állapotban ez nagymértékben lassítja a program futását, így alapesetbe ez ki van kapcsolva.



4.2. ábra A szimulátor nézet

### Szimulátor felépítése

A szimulátor öt főbb részből áll. Az egyik a grafikus felhasználói felület, amely lehetővé teszi a beállítások módosítását, tárolását, illetve a tanulási folyamat nyomon követését. Az *environment* tartalmazza a pályára és környezetre vonatkozó információkat. Ez számolja ki a differenciál egyenletek megoldásával az autó aktuális pozícióját, az autó szenzor adatait, illetve az egyes döntések utáni jutalmat és ezt átadja az autónak. 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 szolgáltatja a beavatkozó jelet.



4.3. ábra A szimulátor blokkdiagram szerű felépítése

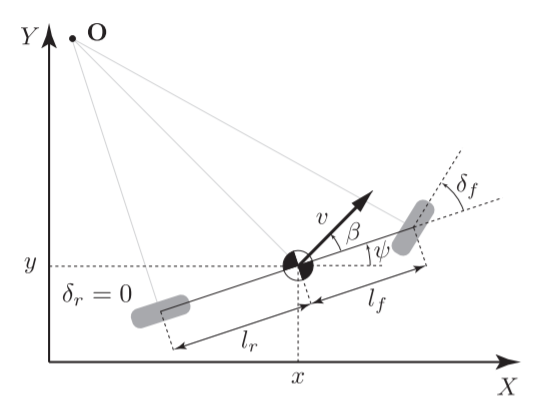
A megtapasztalt állapotátmenetek és az értük járó jutalmak tárolására szolgáló *experience replay memory*-t a Pythonhoz írt egy modul, a Pandas segítségével valósítottam meg. Ez egy adatok kezelésére írt *open source package*, amely kiválóan alkalmas az adatok tárolására és elemzésére. Az itt tárolt 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 md5 hash, amit szükség esetén a memóriában található állapot átmeneteket duplikáció mentesítésére lehet használni. A p pedig a *prioritised experience replay* (röviden PER) 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 [20]. 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, melynek csak az első kerekét tudjuk kormányozni. Az esetleges megcsúszásoktól jelen dolgozatban eltekintünk. Ekkor felírhatjuk a modell mozgásának differenciál egyenleteit, amelyek a következőek:



4.4. ábra A kinematikai bicikli modell szemléltetése

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 kerékelfordulást, amit az egyszerűség kedvéért hívjunk kormányszögnek, illetve számolom ki az autó új állapotát megadó pozíciót, sebességet és orientációt.

## 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. Ezt 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 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 *flag*-et, ami azt mondja meg, hogy a pálya elvárt haladási irányának megfelelően áll-e az autó, illetve az autó aktuális sebességét használom állapotvektorként. Ez elméletben gyorsabb tanulást eredményez, hiszen sokkal általánosabb, így ha például egyszer már megtanult, egyenes szakaszon végig haladni, akkor ezt a későbbi ilyen szakaszon is tudni fogja. Továbbá, ha az algoritmust egy kellőképpen általános pályán tanítjuk, akkor az egy másik pályán is használható, feltéve, hogy a pálya hasonló tulajdonságokkal rendelkezik (illetve megfelelően előfeldolgozott). Mindemellett életszerűbb is az, hogy egy önvezető autó a rajta található szenzoradatokra és esetleg a GPS-adatokra támaszkodjon, minthogy kizárólag a GPS által szolgáltatott adatokra. az így kapott állapotvektor -1 és +1 közé normalizálom, és ezt adom rá a háló bemenetére.

Próbálkoztam még az állapotvektor kiegészítésével, például a sebesség pályaívre merőleges és párhuzamos komponenseire való 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 [21].

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 jutalmazási rendszer (angolul *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 a pályaíven megtett úthosszal arányos jutalomfüggvényt használtam, így az algoritmus minden lépés után kap valamilyen visszacsatolást, hogy mennyire volt jó a lépése. Emellett, mivel a pálya adottságai miatt az autó hajlamos lesodródni az útról, ezért kiegészítettem még a pályaív tengelyére merőleges irányú elmozdulás büntetésével is. Először a tengelyre merőleges elmozdulást előjelesen büntettem, tehát ha távolodik, akkor negatív jutalom, ha közeledik, akkor pozitív, viszont ahogy tanult a háló, gyakran a tengely mentén szlalomozva haladt végig pályán, így később az tengelytől való elmozdulás abszolúlt értékével arányosan büntettem. Felmerülhet bennünk a kérdés, hogy ez nem okozza-e azt, hogy az autó csak a pályaív közepén halad, viszont a diszkontfaktor alkalmazása következtében az autó le fog térni a pálya közepéről, ha így nagyobb jövőbeni jutalmat tud elérni.

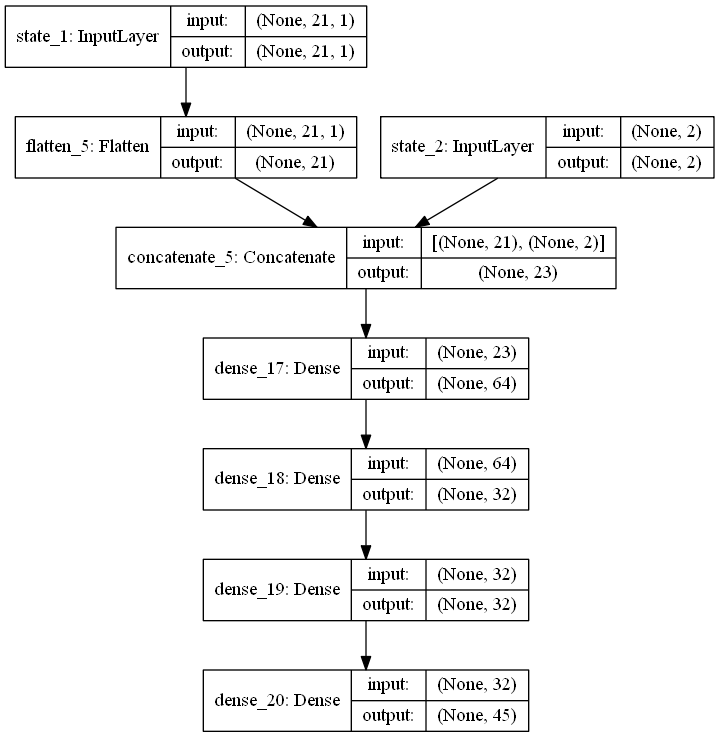
### Alkalmazott felderítő módszer

Sok alkalmazásban használják az ε-greedy felderítő algoritmust vagy valamelyik változatát az állapotátmenetek megfelelő felderítésére. Esetemben viszont nem tartottam előnyösnek a teljesen véletlenszerű lépések használatát, mivel a kellően általános állapotvektor miatt a háló viszonylag hamar megtanul egyszerűbb íveken, egyeneseken végig haladni, így egy ismeretlen helyzetben is valószínűleg megfelelő becslést tud adni arra vonatkozóan, hogy melyik akciók lehetnek az megfelelőbbek. Emiatt inkább a softmax algoritmust használtam. Tekintettel a probléma szekvenciális voltára, minden időpillanatra különböző T *temperature* változót választottam, amelyeket lineárisan csökkentek egy alsó korlátig, amennyibben az előző időpillanathoz tartozó T érték egy megadott alsó határértéket meghaladott.

### Hálózat szerkezete

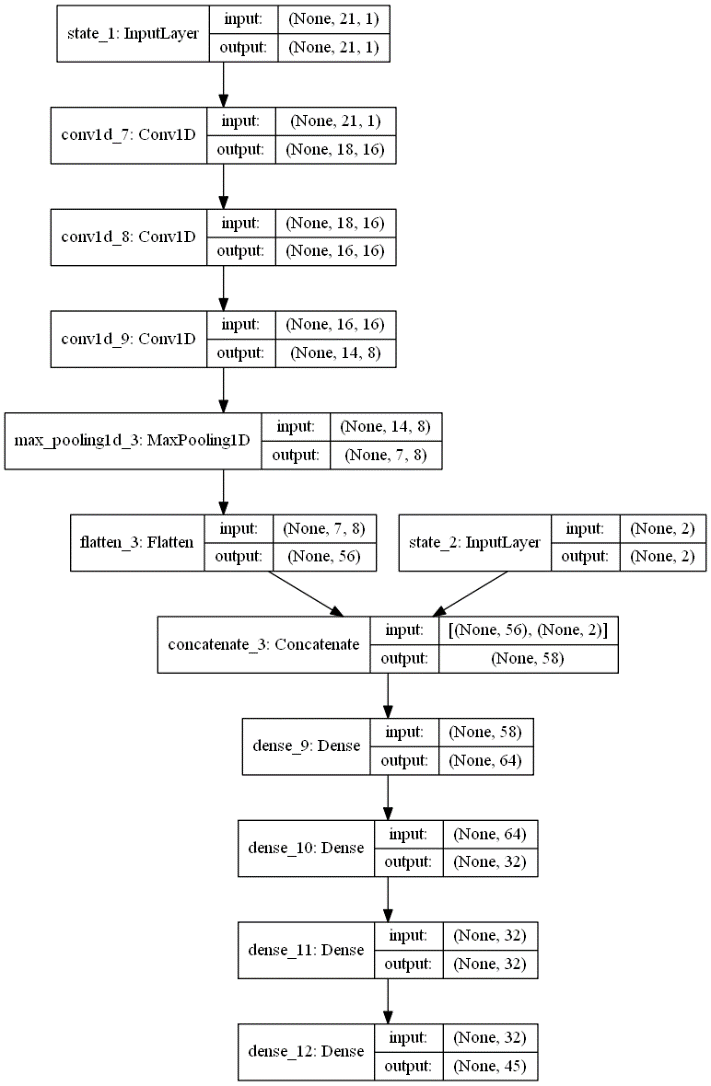
A neurális háló implementálásánál több hálózat típust is kipróbáltam. Mivel nem minden esetben a mély és nagy hálózatok adják a legjobb eredményt, ezért ezt is optimalizálni kell. Először egy bonyolultabb több bemeneti rétegből és több ágból álló hálózatot implementáltam, viszont ez a sok paraméter miatt nem érte el a kívánt eredményt ésszerű időn belül.

Ez után próbáltam a hálózat struktúráját a lehető legegyszerűbbre választani, hogy magát az algoritmus működését tudjam tesztelni. Erre egy három rejtett rétegből álló, egyszerű *fully connected* előrecsatolt hálózatot választottam 64,32 és 32 neuronnal, relu aktivációs függvényt használva. A kimeneten pedig a GUI-ban beállított Gyorsulás felbontás és Kormányszög felbontás felhasználásával, képlet szerinti lineáris kimeneti neuron található. A hálót Adam optimalizáló segítségével tanítom, az alapértelmezett beállításokkal és -es *learning rate*-tel. A hálózat pontos felépítését a 4.5-ös ábrán láthatjuk.



4.5. ábra Az egyszerű előrecsatolt fully connected neurális hálózat felépítése

Miután ezzel az egyszerűbb architektúrával már sikerült a tanuló algoritmusnak eredményes stratégiát megtanulni, szerettem volna továbbfejleszteni, hogy még eredményesebb stratégiákat tudjon elsajátítani. Mint ahogy azt az elméleti részben kifejtettük, a sima *fully connected* rétegek nem veszik figyelembe az adatok egymáshoz viszonyított elhelyezkedésében található összefüggéseket, így viszont sok, a lidar szerű szenzoradatokban az egymás melletti szögekhez tartozó távolságadatokban lévő fontos összefüggést nem használ ki, amik potenciálisan gyorsíthatják, illetve javíthatják a tanulást. Ebből a megfontolásból inkább egy konvolúciós neurális háló struktúrát választottam, melynek két bemeneti rétege található. Az egyik ahol a távolságinformáció érkezik, a másik ahol az állapot vektor többi része. Az első ágon a távolságinformációkból három 1D konvolúciós réteg (sorrendben 16,16 és 8-as filter számmal és 4,3 és 3-as kernel mérettel, illetve 1-es *stride*-al *zero padding* nélkül) segítségével kinyerem a fontosabb összefüggéseket, jellemzőket, majd egy *maxpooling* réteggel megfelezem a dimenziók számát. Ezt követően a kapott adatokat és a második bemeneti rétegen lévő adatokat az előző három rejtett *fully connected* rétegből álló hálóval dolgozom fel. Itt is Adam optimalizálót használok az alapértelmezett beállításokkal és -es *learning rate*-tel. Az alkalmazott konvolúciós hálózat teljes felépítését a 4.6.-os ábrán láthatjuk.



4.6. ábra Az alkalmazott 1D konvolúciós hálózat felépítése

## Implementált algoritmusok

**DDPG**

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 kapott lépést. A folytonos kimeneti jelhez normális eloszlású zajt kevertem, így valósítottam meg az állapot átmenetek felfedezését. Sajnos mivel a környezet implementációja még nem volt megfelelően optimalizálva, illetve a tanításhoz használt számítógép számítási kapacitása is erősen korlátozott, ezt az irányt el kellett vessem és diszkrét kimenetekre kellett az algoritmust megtervezzem.

**DQN**

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. Ennek az eredménye az volt, hogy egy bizonyos időpillanatban az algoritmus a neurális háló által kiadott maximális Q értékhez tartozó akciót priorizálta, de a többi akciót is felfedezte. Ezzel az algoritmussal már sikerült úgy tanítani a hálót, hogy megtanuljon valamilyen eredményes stratégiát és körbe menjen. Viszont a sima DQN algoritmus hajlamos a Q értékeke túlbecslésére, ami ronthatja a tanítás teljesítményét.

**DDQN**

A Q értékek túlbecslését kiküszöbölendő átalakítottam az implementált algoritmus, hogy az a Double Deep Q Learning algoritmus valósítsa meg. Bár ez az algoritmus már csak kis mértékben becsülte túl az Q értékeket, viszont valamivel lassabban tanult a sima DQN-nél. Továbbá, mivel a probléma igen szekvenciális és az által meghatározott MDP meglehetősen mély állapot gráfot eredményez, így ha ennél jobban is javítani szeretnénk a tanítási időn és az eredményen egyéb technikákat is alkalmaznunk kell. Alap esetben az memóriában eltárolt állapotátmeneteket egyenletes eloszlással mintavételezzük. A tárolt állapotátmenetek között viszont vannak fontosabbak és kevésbé fontosak is, így ezeket tudjuk priorizálni, a tanítás során. Azok az átmenetek, amik Q értékét a háló már jól tudja becsülni, már nem sokat javítanak az algoritmus eredményén, így azokat az állapotátmeneteket kéne priorizáljuk a memóriában, amelyek predikált értéke nagymértékben eltér a valóstól, így gyorsíthatjuk a tanítást. Ez az ún. *prioritised experience replay* (röviden PER [22]). Ennek létezik rang alapú és arányos változata is. Én az arányos verziót implementáltam, ami az alábbi képlet alapján számolja ki egy-egy átmenet prioritását:

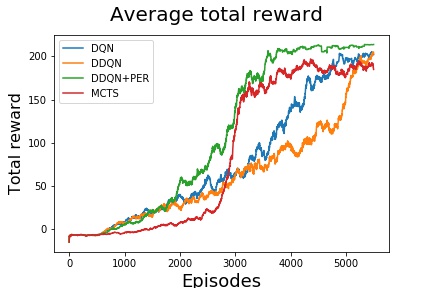
|  |  |
| --- | --- |
|  | (4.17) |

Ahol a nulla hiba esetén a nullás prioritást kizáró konstans, pedig a priorizálás súlyát adja meg. Majd ezeknek a prioritásoknak a segítségével mintavételezzük arányos súlyozással a memóriát. Vegyük észre, hogy esetén ez megegyezik az egyenletes eloszlással. Ezzel az epizódonkénti összes jutalom 100-as mozgóátlaga szembetűnően gyorsabban konvergál a maximum értékhez. Tovább javítottam a futásidőn a gyakran hívott függvények, mint például a szenzoradatokat kinyerő függvény optimalizálásával is.

**Monte Carlo Tree Search**

Az utolsó kipróbált algoritmus a Monte Carlo Tree Search volt. Bár ennek a teljesítménye kismértékben elmaradt a *prioritised experience replay*-el kombinált DDQN-étől, viszont ez is relatív hamar rátanul eredményes stratégiákra. Ennek az algoritmusnak részletesebb modellek esetén jön elő igazán az előnye a többi módszerrel szemben, mivel nem tölt sok időt a valószínűsíthetően nem optimális stratégiákhoz vezető lépések értékelésével, és lényegében közvetlenül a lépés aktuális stratégiához tartozó Q értékét „mintavételezzük”, nem pedig egy becslés segítségével becsüljük. A megfelelő döntések kiválasztásáért a softmax felfedező függvény a felelős, ahol a diszkrét időpillanatonkénti saját T érték használata eredményezi, hogy előbb egy szinten szimulációkat futtatt, majd azok értékét visszaterjeszti az előző szintek állapot átmeneteire, ez után dönt az adott szinten, hogy mi legyen a következő lépés. Az eredmények visszaterjesztését, úgy oldom meg, hogy egy epizódban az összes lépéshez tartozó állapotátmenetet egy ideiglenes bufferben tárolom, ahol a tárolt jutalmakat mindig az aktuális lépésért járó jutalmának diszkontált értékével változtatom, majd az epizód végén az egész ideiglenes buffert teszem bele a memóriába. Ennek felhasználásával az algoritmushoz szükséges fát, vagyis az abban tárolt Q értékeket és ez által a megtanult *policy*-t a neurális háló segítségével becsülöm.

Az egyes algoritmusok által elért eredményeket a 4.7.-es ábrán láthatjuk. A 190-es összesített jutalom jelenti, hogy az autó körbeért.



4.7. ábra Az egyes algoritmusokkal elért epizódonkénti kummulált jutalom 100-as mozgóátlaga

## 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 *profiler*-ét használtam fel. Ezzel nagy mértékben fel tudtam gyorsítani az szimulátor futását, főleg a gyakran meghívott szenzor adat generáló függvények optimalizálásával. Szükség volt továbbá az megtapasztalt állapotátmenetek hatékony, gyors hozzáférést lehetővé tevő tárolására. Ezt a Pythonhoz írt pandas modul segítségével oldottam meg, aminek a segítségével könnyen hozzáférhető és mintavételezhető adatbázist tudtam kialakítani a tanítóadatok számára.

### Teljesítmény

Kezdetben az algoritmus fejlesztését és tesztelését egy régebbi típusú kisebb számítási kapacitású laptopon végeztem, viszont ennek nagyon sok időbe telt az algoritmusok futtatása, illetve a kisebb teljesítményű NVIDIA gpu-ja nem tette lehetővé, hogy a tensorflow-nak a gpu optimalizált verzióját használjam. Később ezért egy Intel i7-6700-es processzorral, 16 gigabyte memóriával és egy NVIDIA Geforce GTX 950M-es 4 gigabyte memóriájú gpu-val rendelkező laptoppal folytattam a fejlesztést, illetve a tesztelést. Ez már jelentősen javította a fejlesztés sebességét, mivel sokkal hamarabb kaptam visszajelzést arról, hogy az algoritmus helyesen működik-e, köszönhetően a több memóriának és a gpu optimalizált tensorflow által okozott teljesítménynövekedésnek.

### Megfelelő jutalomfüggvény megválasztása

Fontos a jutalom függvény megfelelő megválasztása, és annak átgondolása, hogy a választott jutalmazási rendszer mire „ösztönzi” az ágenst. Kezdetben olyan jutalmazással próbálkoztam, ami -1 és 1 között van, az az alatti vagy feletti értékeket levágjuk, ebben az esetben viszont gyakran előfordult, hogy az algoritmus nem tudta egyértelműen megállapítani, hogy két vagy több lehetséges akció közül melyik az előnyösebb, mert a -1 alatti vagy 1 feletti részt, ami ezt eldöntötte volna, levágta. A végső algoritmus jutalmainak értékét ezért -10 és 40 közé választottam. Ez továbbá gyorsabb tanulást is eredményezett.

### Hiperparaméter optimalizáció

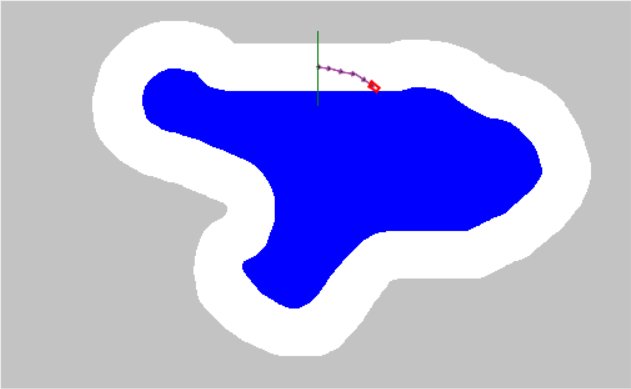
A neurális hálók és a megerősítéses tanulás egyik legnagyobb nehézségét a hiperparaméterek optimalizálása okozza. Lehet, hogy az algoritmust helyesen implementáltuk és a logika mögötte is jó, viszont könnyen múlhat a paraméterek beállításán a módszer eredményessége. Ezeknek az automatikus beállítására és finomhangolására már lépeznek modern módszerek, ezek viszont rendkívül számítás igényesek, így megfelelő hardware nélkül, hamarabb, illetve jobb eredményeket lehet elérni, ha a paramétereket empirikus, módon kézzel optimalizáljuk. A legnagyobb problémát a *batch* méret, a *learning rate*, az felderítő algoritmus és a T paraméterének megfelelő megválasztása jelentette.

### Komplexitás

A fejlesztés igen sokáig tartott, köszönhetően a feladat komplexitásának. Mivel az algoritmus több helyen is elcsúszhat, igen nehéz volt debugolni az elkészült megoldást, különösen, mert a megerősítéses tanulás esetén nem mindig egyértelmű, hogy mi és hol a hibát kiváltó tényező. Könnyen előfordulhat az is, hogy bár az implementációban elvi hiba található, az algoritmus mégis elkezd tanulni, és csak több ezer iteráció után látszik tisztán, hogy valami hibás, viszont ezt is nehéz verifikálni, mert néha mi sem ismerjük az optimális megoldást.

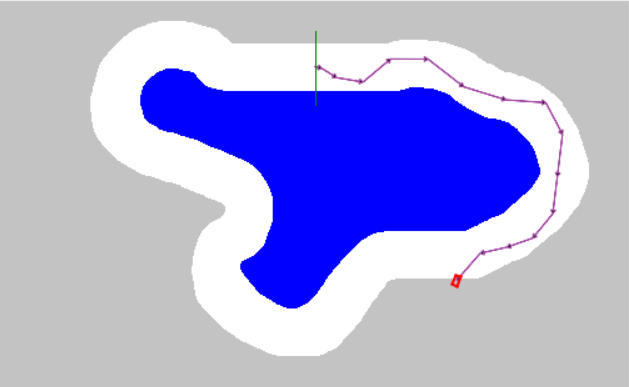
## Eredmények értékelése

A tanítás elején az első pár száz, ezer epizódig az autó láthatóan, gyakran hibázik, láthatóan még nem tanulta, meg, hogy milyen akciók okoznak ütközést, ahogy ezt az 4.8.-as ábrán is látjuk.



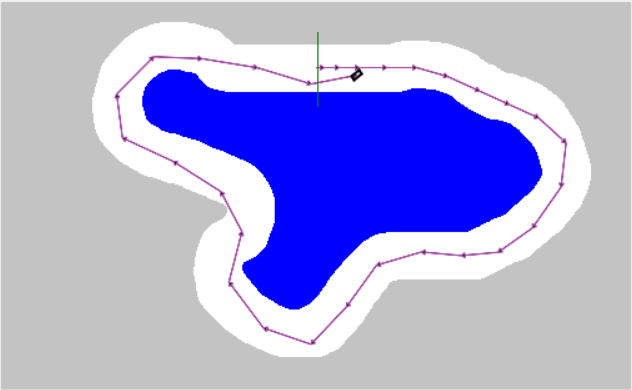
4.8. ábra Az autó útvonala a tanítás elején

A tanítás előrehaladásával már egyre tovább jut a pályán. Fokozatosan megtanulja bevenni a kanyarokat, de azért még gyakran találkozik olyan helyzetekkel, amikor rossz döntést hoz meg. Továbbá az eddigi haladási ív is távol áll az ideálistól, illetve nagyon töredezett is, gyakoriak a hirtelen irányváltások ahogy azt az 4.9.-es ábrán is látni.



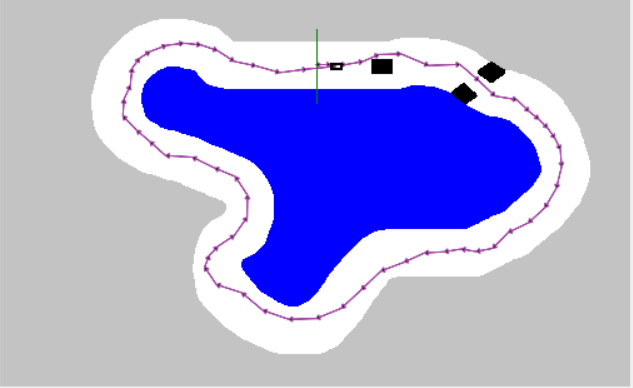
4.9. ábra Az autó útvonala kétezer epizód után

A tanítás végére olyan 6000 epizód után már szinte mindegyik algoritmus biztosan meg tudta tanulni racionális módon irányítani az autót a bemenő adatok függvényében. Ahogy az 4.10.-es képen is látjuk, ilyenkor már ideálisabb íveken halad végig, kevésbé sodródik a pályán és képes ütközés nélkül egészen a célvonalig eljutni egy több kanyart is tartalmazó pályán.



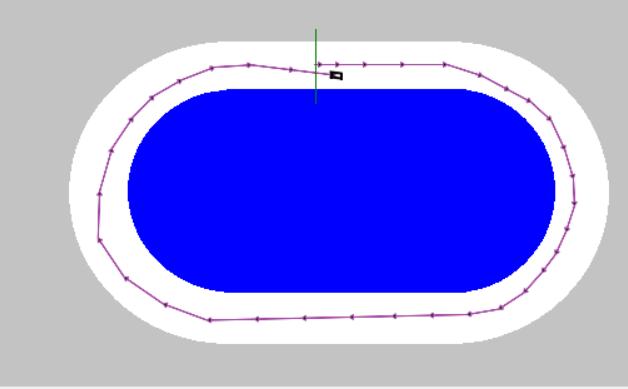
4.10. ábra Az autó útvonala a tanítás végén

Továbbá esetleges akadályok elhelyezése sem okozott számukra különösebb fennakadást, amennyiben hasonló akadályokkal nehezített pályán végeztük a háló tanítását. Bár a tanítás elején több időbe telik mire túljut a kezdeti akadályokon, mivel nagymértékben leszűkítik az utat, de miután már megtanulta kikerülni őket, a pálya hátralévő részén már hamarabb túljut.



4.11. ábra Az algoritmusok akadályokkal nehezített pályán is képesek megtanulni az autó megfelelő irányítását

Továbbá külön pozitívum, hogy a kellően általános pályán betanult algoritmusok képesek voltak más alakú pályákon is végig haladni gond nélkül, ahogy az a 4.12. képen látható. Ezekből az eredményekből látszik, milyen fontos a használt állapotok, illetve a algoritmusok megfelelő kiválasztása és tovább fejlesztése, például a *prioritised experience replay*-el.



4.12. ábra A betanított algoritmusok már más pályán is képesek eredményesen végighaladni

|  |  |  |
| --- | --- | --- |
| Algoritmus | Legjobb eredmény tanuló pályán (track\_2) (lépés a célig) | Legjobb eredmény teszt pályán (track\_oval)  (lépés a célig) |
| DDQN | 29 | 29 |
| MCTS | 38 | 32 |
| DQN | 27 | 28 |

4.13. ábra A tesztelt algortmusok legjobb ereményei a tanuló pályán és egy teszt pályán

## Lehetséges továbbfejlesztések

**ALGORITMUS**

A szimulátort és a megvalósított irányítást számos ponton tovább lehetne fejleszteni a jövőben. Az irányítás magját jelentő algoritmust illetően, véleményem szerint a probléma szekvenciális mivolta miatt a Monte Carlo Tree Search különböző változatainak implementálásával lehetne a legnagyobb teljesítményjavulást elérni. Ennek több változata is létezik, amelyeket sikerrel lehetne alkalmazni ezen a probléma területen, például az állapot tér felfedezésénél különböző biztonsági korlátok bevezetését alkalmazó változat szerintem kifejezetten alkalmas lenne, mivel még kevesebb időt töltene az algoritmus az ütközéshez vezető stratégiákkal.

**IMPLEMENTÁCIÓ**

Emellett a memória kezelését is lehetne optimalizálni, esetleg a felhasznált fának a tényleges építésével, vagy annak neurális hálóval való valamilyen kombinálásával. Ez a nagyobb kiterjedésű rendszereknél sokkal kritikusabb pontot jelent, ugyanis a nagy mennyiségű adatan nehéz keresni és mintavételezni.

**ÁLLAPOTÁTMENETEK FELDERÍTÉSE**

További fejlesztési irányt jelenthet más felderítő módszerek implementálása. A Monte Carlo Tree Search egy gyakori változata az UTC algoritmus, amely a felderítés során figyelembe veszi, hogy az adott állapotot, illetve állapotátmenetet hányszor látott az ágens. Illetve a jelenleg létező tanuló algoritmusok többségében, így az általam implementáltakban is igaz, hogy a tanítás során nem vesszük figyelembe a témában létező nem gépi tanulás alapú algoritmusokat és a szakértői tudást, így az algoritmusunk a nulláról kell megtanuljon mindent, ami igen hosszadalmas. Erre egy megoldást jelenthet az ún. Guided DQN [23] algoritmus, illetve a szemlélete, hogy a felderítés során, néha egy olyan akció sorozatot kövessen az algoritmus, amelyet valamilyen nem gépi tanuláson alapul verzió számít ki és ezt optimalizálja. Ez mind futás időben, mind eredményességben is jelentős javuláshoz vezethet.

**FELADAT RÉSZLETEZÉSE**

Ezen kívül tovább lehetne bonyolítani a feladatot, hogy részletesebb, a valósághoz minél jobban illeszkedő megoldást kapjunk. Ilyen változtatás lehetne például a dinamikus akadályok használata a tanítás során, esetleg más ágensek, akikkel a miénk versenyez. A tanítást végezhetnénk több pályára is egyszerre, így egyre általánosabb megoldást kapnánk, ami változatos alakú és tulajdonságú pályákon is boldogulna. Az ezen szakdolgozat során implementált algoritmusokat egyszerre csak egy pályán tanítottam, így volt, hogy arra a pályára jobban rátanult, viszont egy észrevehetően más tulajdonságokkal bíró pályán csak szlalomozva, az egyik faltól a másikig sodródva tudott végigmenni.

**BONYOLULTABB FIZIKAI MODELL**

További továbbfejlesztési irányt jelenthet a megfelelő állapot vektor megalkotása, más minőségű dimenziók bevitelével, a meglévők részletezésével. Illetve a valóság minél élethűbb megközelítése érdekében bonyolultabb, részletesebb járműmodelleket is lehetne alkalmazni, ami már figyelembe veszi az autó sajátosságait, mint a tömeget, a tehetetlenséget, a súlypontot, a tapadást és a megcsúszást.

**PÁRHUZAMOSÍTÁS**

Mivel egy nagyon részletes modell használata, illetve a bonyolultabb algoritmusok még időigényesebbek, ezért előnyös lenne a szimulációk párhuzamos elosztott rendszerekben való futtatása, illetve a szimulátor ehhez való adaptálása.

**FIZIKAI MEGVALÓSÍTÁS**

A megfelelő vezérlés implementálása és tanítása után a végső lépés ennek a valóéletbeni alkalmazása, például egy távvezérlésű játékautó, vagy önjáró robot segítségével. Ehhez szükség lehet az ún. *transfer learning* használatára, amikor is szimuláción betanítunk előre egy neurális hálót, majd az előtanított hálót vagy egy részét felhasználva, azt továbbtanítjuk, már a valós probléma és szenzoradatok segítségével.

# Összefoglaló

A dolgozatom során egy irányítástechnikai problémára kerestem lehetséges alternatív megoldásokat a mély megerősítéses tanulás felhasználásával. A munka során elkészült keretrendszer alkalmas különböző algoritmusok tanítására, a tanulási folyamat nyomonkövetésére és tesztelésére, emellett az implementált módszerek eredményesen tudják vezérelni a versenyautó modellt több más-más tulajdonságokkal rendelkező pályán is.

Ahogy a dolgozatomból és a szimulációk eredményéből is látszik, a megerősítéses tanulás és a neurális hálók egy igen hatásos eszközt jelenthetnek az irányítástechnikai problémák megoldása során. Segítségükkel olyan irányítási feladatok is könnyen elvégezhetőek, amiket korábban csak rendkívül bonyolult analitikus módon lehetett megoldani. Az alkalmazásukhoz szükséges hardware pedig a technika fejlődésével egyre kevésbé lesz költséges, így több területen is alternatívát jelenthetnek az eddig létező hagyományos megoldásokkal szemben, vagy azokkal kombinálva még hatásosabb módszereket alkothatnak. Mindenesetre tisztán látszik, hogy a bennük rejlő potenciál igen magas és a következő években évtizedekben egyre gyakrabban fognak megjelenni a tanuló algoritmusokra épülő gyakorlati megoldások is a való életben.

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | D. Silver és A. Huang, „Mastering the game of go with deep neural networks  and tree searc” *Nature,* 529., p. 484–489., 2016.. |
| [2] | D. Silver, Reinforcement Learning lecture slides: MDP  Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching\_files/MDP.pdf. [Hozzáférés dátuma: 23. 11. 2017.]. |
| [3] | B. Haanstra, *Reinforcement Learning 101,* 2016.  Available: https://kofzor.github.io/Reinforcement\_Learning\_101/  [Hozzáférés dátuma: 23. 11. 2017.] |
| [4] | C. J. C. H. Watkins és P. Dayan, „Q-learning,”  *Machine Learning,* 8., p. 279–292., 1992.. |
| [5] | D. Silver, *Reinforcement Learning lecture slides: Policy Gradient Methods.*  Available: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching\_files/pg.pdf  [Hozzáférés dátuma: 23. 11. 2017.] |
| [6] | T. Eden, A. Knittel és R. v. Uffelen, University of Sydney, CS9417 course notes: Reinforcement Learning  Available: http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html  [Hozzáférés dátuma: 24. 11. 2017.]. |
| [7] | T. Sau, *Breath acetone-based non-invasive detection of blood glucose levels,* International Journal on Smart Sensing and Intelligent Systems,8. 2015. |
| [8] | A. Karpathy, CS231n Convolutional Neural Networks for Visual Recognition lecture notes  Available: http://cs231n.github.io/neural-networks-1/  [Hozzáférés dátuma: 24. 11. 2017.]. |
| [9] | „Quora,” [Online]. Available: https://algorithmsdatascience.quora.com/ReLu-compared-against-Sigmoid-Softmax-Tanh.  [Hozzáférés dátuma: 24. 11. 2017.]. |
| [10] | J. Makin, Cornell University: CS5740 course lecture notes.  Available: http://www.cs.cornell.edu/courses/cs5740/2016sp/resources/backprop.pdf. [Hozzáférés dátuma: 23. 11. 2017.]. |
| [11] | D. P. Kingma és J. L. Ba, „Adam: A method for stochastic optimization,” in *International Conference on Learning Representations*, 2015. |
| [12] | [Online].  Available: https://pennlio.wordpress.com/2014/04/11/fully-connected-locally-connected-and-shared-weights-layer-in-neural-networks/.  [Hozzáférés dátuma: 24. 11. 2017]. |
| [13] | [Online]. Available: https://i.stack.imgur.com/GvsBA.jpg. [Hozzáférés dátuma: 24. 11. 2017.]. |
| [14] | [Online]. Available: http://cs231n.github.io/convolutional-networks/#pool. |
| [15] | V. Mnih, K. Kavukcuoglu és D. Silver, „Human-level control through deep reinforcement learning”518., p. 529–533., 2015.. |
| [16] | H. v. Hasselt, A. Guez és D. Silver, „Deep Reinforcement Learning with Double Q-learning,” in *Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016.. |
| [17] | T. P. Lillicrap és J. J. Hunt, „Continous control with deep reinforcement learning,” in ICLR, 2016.. |
| [18] | [Online]. Available:  https://www.revolvy.com/main/index.php?s=Heuristic%20(computer%20science). |
| [19] | M. Bowling, PhD dissertation: Multiagent Learning in the Presence of Agents with Limitations*,* Pittsburgh, 2003. |
| [20] | J. Kong, M. Pfeiffer, G. Schildbach és F. Borrelli, Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design  [Online]. Available:  http://www.me.berkeley.edu/~frborrel/pdfpub/IV\_KinematicMPC\_jason.pdf. [Hozzáférés dátuma: 23 11. 2017]. |
| [21] | J. Clark és D. Amodei, *Faulty Reward Functions in the Wild*  [Online]. Available: https://blog.openai.com/faulty-reward-functions/  [Hozzáférés dátuma: 23 11. 2017] |
| [22] | T. Schaul, J. Quan, I. Antonoglou és D. Silver, „Proritised experience replay,” in International Conference on Learning Representations, 2016.. |
| [23] | A. Taitler és N. Shimkin, „Learning to Strike Accurately with DQN-based Algorithms” International Journal of Mathematical Models and Methods in Applied Sciences *,* 11., pp. 158-169., 2017.. |
| [24] | Google, „Tensorflow,” Google, [Online].  Available: https://www.tensorflow.org/api\_docs/python/.  [Hozzáférés dátuma: 23. 11. 2017.]. |
| [25] | „Wolfram Mathworld,” [Online]. Available:  http://mathworld.wolfram.com/SigmoidFunction.html. |
| [26] | „Wolfram Mathworld,” [Online]. Available: http://mathworld.wolfram.com/HyperbolicTangent.html. |

Függelék

**F.1. Használati útmutató**

A szimulátort a *main.py* script futtatásával lehet elindítani, ami után a szükséges modulok betöltését követően a kezdőképernyő jelenik meg a beállításokkal. Itt a 4.1.1-es fejezetben felsorolt beállításokat találjuk. Az autó beállításainál a numerikus értékekhez nullánál nagyobb pozitív számot szükséges írni a szimulátor helyes működéséhez. Továbbá a *Lidar* *resolution*, *Acceleration* *resolution* és a *Steering* *resolution* mezőhöz páros egész számot.

A *Select track* megnyomása után a felugró dialógus panelben kiválaszthatjuk a pályát. Ez lehet .png, .jpg esetleg .bng formátumú kép. Néhány előre elkészített pálya található a *Maps* mappában is, de új pályák elkészítésére is van lehetőség a később ismertetett megkötések betartásával. A kiválasztást követően a pálya előnézeti képén ellenőrizhetjük azt. A *Draw track* checkbox beikszelésével az autó helyzete és útvonala dinamikusan megjelenítésre kerül futás közben is. Ezt az opciót futásidő közben is interaktívan tudjuk változtatni majd. Az algoritmus beállításainál a *Load model?* beikszelésével kívülről betöltött neurális háló modellt használhatunk. Ehhez továbbá szükséges a *Select model* és a *Select weights* gombokra kattintás után felugró dialogós ablakok segítségével a betölteni kívánt neurális háló modell és a hozzá tartozó súlyok betöltése. Néhány betanított háló megtalálható a *Models* mappában lévő almappákban. A neurális hálót tartalmazó fájlnak .json kiterjesztésűnek, a súlyokat tartalmazó fájlnak .h5 kiterjesztésűnek kell lennie. Ez után beállíthatjuk, hogy a hálót szeretnénk-e tanítani vagy nem. Ezt követően be tudjuk állítani a szükséges hiperparamétereket, az alkalmazott háló típusát és az algoritmust. A *Use PER* checkbox bepipálásával állíthatjuk be, hogy alkalmazzon-e *Prioritised Experience Replay*-t az algoritmus a tanítás során. Ez után az ablak alján található *GO* gombra kattintva kis várakozás után a szimulátor oldalra kerülünk. Itt látjuk a pályát és a rajta haladó autó útvonalát. A nyilak az autó elmozdulásvektorait adják meg az egyes diszkrét időlépések után. A *Settings* gombra kattintva megszakítjuk a tanítást és visszajutunk a beállításokhoz. A *DEBUG* gombra kattintva néhány debugoláshoz szükséges információ kerül kiírásra a konzolra. A *Test* gomb megnyomása után a következő epizódban, az autó kizárólag az algoritmus által számolt vezérlőjelekre hagyatkozik, így tesztelhetjük, hogy az addig megtanult stratégia mennyire eredményes. Minden századik epizódban automatikusan így tesztelődik az algoritmus és a legjobb eredményt, tehát a legkevesebb lépés alatt célba érő modellt és a hozzá tartozó súlyokat kimenti a *Models*  mappába. Ezt később visszatöltve tesztelhetjük az algoritmust más pályákon is. Ha a megfelelő opciót beállítottuk, a modell és a súlyok a tanítás végén is kimentődnek. Végül a *progress bar*-on nyomonkövethetjük, hogy a megadott epizódszámból százalékosan addig hányat teljesített.

**ÚJ PÁLYA KÉSZÍTÉSE**

Új pályák készítésére is van lehetőség, bár mivel nem ezen volt a fókusz a munka során, így nem teljesen rugalmas a szimulátor. A legegyszerűbb megoldás egy meglévő pályát átalakítani a megfelelő megkötésekkel. Ezek a következők:

* A startvonal a (x:251,y:30),(x:251,y:80) szakaszon van. Ennek a közepéről indul az autó a kép jobboldala felé az oldalra merőlegesen.
* A pálya RGB kódja: (255,255,255)
* A pálya külső részének RGB kódja: (195,195,195)
* A pálya és a kép széle el kell legyen határolva a külső pálya réz által
* A pálya belső pályarész RGB kódja: (0,0,255)
* A pályán található esetleges akadályok RGB kódja: (0,0,0)
* A pálya egy startvonaltól induló, a startvonalba záródó hurok, ami nem tartalmazhat elágazásokat.

**F.2. A szükséges Python modulok**

A szimulátor futtatásához az alábbi fontosabb Python modulok, illetve Python 3.5.2 szükségesek:

Keras 2.0.8

matplotlib 2.0.2

numpy 1.13.3

pandas 0.20.3

python 3.5.2

scikit-image 0.13.0

scikit-learn 0.19.0

scipy 0.19.1

tensorflow-gpu 1.3.0

tensorflow-tensorboard 0.1.8