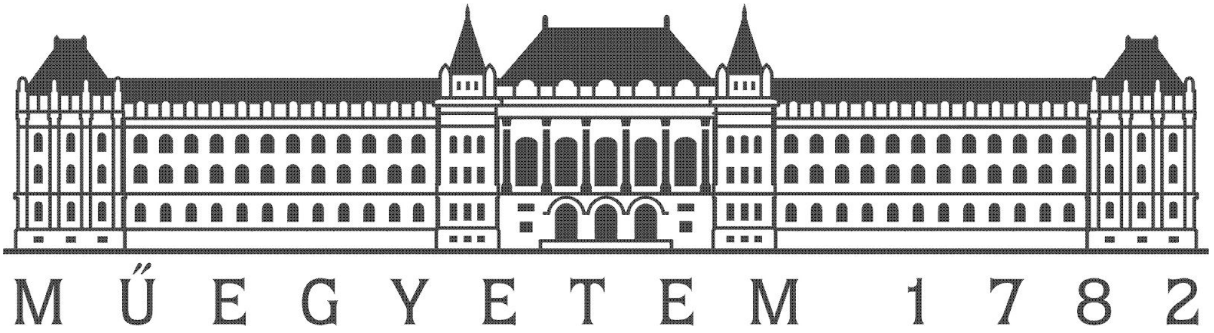


Kiskocsi tanítása reinforcement learninggel

Neurális hálózatok (BMEVIJV07) projekt dokumentáció

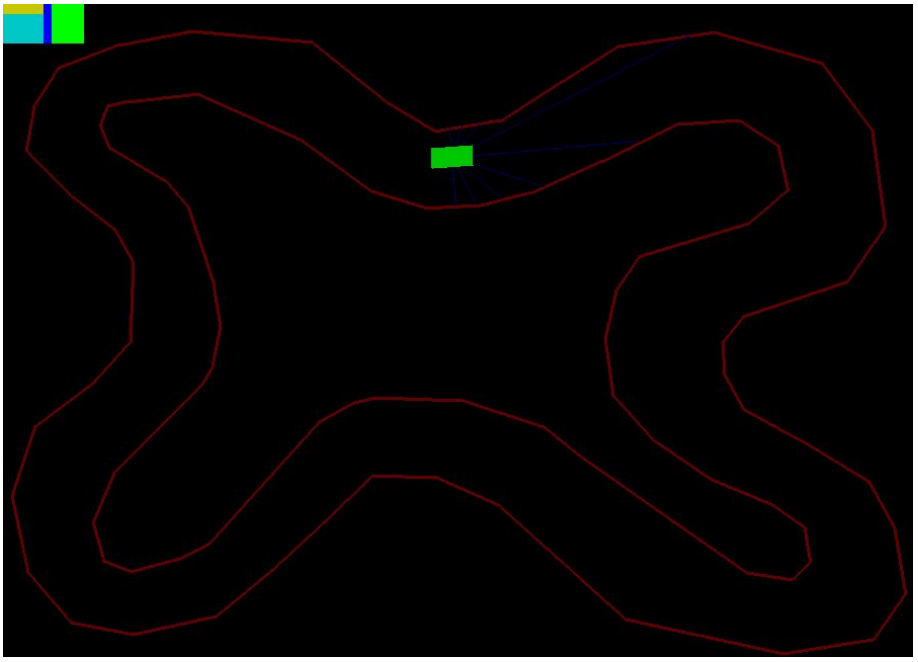
Készítette:
Bekes Nándor
GFUKR3
Csik Dávid
NVOUMV

A projekt célja és bemutatása

Motiváció

Mivel a csapatunkból mind a ketten a gépész karon tanulunk mechatronikát, amely robotikával is foglalkozik, úgy döntöttünk, hogy olyan projektfeladatot választunk, amely közel is áll ehhez a területhez. Pont ezért esett a választásunk egy olyan feladatra, amelyik Reinforcement Learning-gel tud tanulni.
A Reinforcement Learning közel áll ahhoz, ahogy az emberek is tanulnak, ez a trial and error megközelítéses tanítás. Ezen módszer segítségével a robotok ugyanúgy képesek lehetnek járni tanulni, esetleg a karjaikat mozgatni, ami jól jöhet különböző feladatok megoldásánál, ahol egy nem betanított robot valamilyen szabályozóval felszerelve sem lenne képes teljes értékű munkát végezni. Ilyen lehet nem egyenletes felületek polírozása, csiszolása, esetleg háztartásbeli feladatok ellátása.
Mindezek mellett ezen algoritmusok olyan stratégiákat tudnak kidolgozni, megtanulni, amelyek meghaladják az emberek képességeit is. Ilyen például az Alpha GO, vagy az OpenAI Five, amelyek különböző területeken (Go és Dota2) győzték le az adott bajnokot.
Viszont még egyikünk sem foglalkozott neurális hálózatokkal, ezért ez a projektfeladat egy kiváló lehetőség arra, hogy megismerjünk különböző algoritmusokat, használjuk őket, és hogy hozzá szoktassuk magunkat a különböző technikákhoz. Mindezen felül különböző bonyolultságú feladatok különböző méretű hálókat és másfajta technikákat igénylenek, amelyek közül ezen feladat megoldása során kipróbáltunk egy párat.

Rövid leírás



1. Ábra

Választott feladatunk egy matchbox betanítása több, különböző pályán a már fent említett Reinforcement Learning segítségével.

A projektfeladatunk elkészítéséhez PyCharm környezetben programoztunk. A neurális hálók megírásához pedig felhasználtuk a Keras-RL és Tensorflow importálható csomagjait. Végleges célunk az volt, hogy a kisautónk végig tudjon menni minél több tesztpályán a tanítás után.
A környezet viselkedésének (pl. kicsúszás, jutalom értéke) ellenőrzésére és debuggolásra írtunk egy kódot, hogy kézzel is irányíthassuk a kisautónkat.

Környezet

Úgy döntöttünk, hogy teljesen nulláról építünk fel saját magunknak egy környezetet, melyben a matchboxunk szabadon tud majd mozogni. Azért nem egy létező environment mellett döntöttünk, mert úgy éreztük, hogy sokkal nagyobb szabadságunk lesz a kód írása közben. S mindenek felett ezzel sokkal rugalmasabbak tudunk is lenni így, ami annak köszönhető, hogy nem kellett megtanulnunk, megértenünk egy számunkra ismeretlen kódot. Mindemellett az is fontos volt számunkra, hogy a kódban csak olyan dolgok szerepeljenek, amik tényleg szükségesek a neurális háló tanításához. Azt is szerettük volna, hogy könnyen tudjunk változtatni a problémás részeken, és ezeket összhangba hozni a kód többi részével.

2. Ábra: A környezet vizualizációja

Az 2. ábrán látható a bal felső sarokban 3 négyzet, melyek visszajelzésre szolgálnak. Az első négyzetben egy vízszintes szintvonal mutatja a gyorsítás/lassítás mértékét, a második négyzet hasonlóan mutatja a kormány jobbra-balra fordulását (itt szélsőhelyzetben van), a harmadik, piros négyzet a kicsúszást jelzi, intenzitása mutatja azt is, mennyire tér el a kanyarodási sugár a csúszás nélküli esethez képest.

A pálya

A környezetet Pygame-ben hoztuk létre, ahol egy limitált méretű ablakban jelenik meg a pálya és az autó. Maga az autó nincsen túlbonyolítva, mivel az csak egy téglalap. Ám a pályák generálása már valamivel nehezebb feladat volt. Külön-külön hozzuk létre a külső és belső íveket a pályához, amelyek között a matchboxunknak körbe kell mennie saját magától. A terep létrehozása után rögtön még a kiskocsink kezdő pozícióját és irányát is megadjuk. Mindezen adatot egy szövegfájlban tárolunk, amelyet a későbbiekben betöltünk, hogy le tudjuk generálni a pályát.

Az autó és érzékelés

Magának az autónak egyszerű felépítése van. Fontos tulajdonsága az autónak a sebesség, és kormányának a állásának a szöge. Ezek közül az autónak a sebessége input adat lesz. Ám ahhoz, hogy a pályán tudjon maradni, keresővonalakat is adtunk a kisautónkhoz. Ezen vonalaknak a mennyiségét és látószögét egyaránt be lehet állítani.
A keresővonalak segítségével ki tudjuk számolni a matchboxunk távolságát mind a külső, mind a belső ívtől, ám figyelnünk kellett arra, hogy egy vonal egyszerre több helyen metszi a belső és külső íveket, így meg kellett oldani, hogy csak az egy vonalra eső legközelebbi értékeket vegye figyelembe, mivel később csak ezen pontokat is be fogjuk adni a neurális hálónak, mint bemenet. Az autó a helyzetéről viszont más formában nem kap információt.
Ha viszont mégis hozzáér a falhoz, akkor azt úgy tekintjük, hogy nekiütközött, és ilyenkor az adott pályán való tanítás, illetve teszt befejeződik, és ugrik a következő pályára, vagy esetleg befejeződik a program futása, ha az utolsó pályán jár. Itt az volt a megfontolásunk, hogy a különböző pályákon szándékosan különböző helyzetben kezd az autó, (pl. jobb kanyar, bal kanyar, egyenes), így kevésbé valószínű, hogy a hálózat tanítása úgy konvergáljon be, hogy megtanulja az egyik helyzetet kezelni, de amikor már elér a pályának arra a részére, ahol más problémát kellene megoldania, beragadt egy lokális optimumba, és nem tudja megtanulni, milyen policyval tudná teljesíteni ezt a pályarészt.
Az ütközés detektálása is a keresővonalak segítségével történik, amint az egyik keresővonalon számított érték egy bizonyos küszöbérték alá csökken, akkor befejeződik az éppen soron lévő pálya.

3. Ábra: Ütközés

Fizika

Az autó irányítása úgy történik, hogy a kisautónk minden egyes lépésben növeli vagy csökkenti egy bizonyos Δv értékkel a sebességet, és a kormány állásának a szögét pedig közvetlenül megadja a neurális háló kimenete.
Mind a sebesség, mind a kormány szögének állása szimmetrikusan limitálva van két érték közé. Ez a kormány állásánál egyértelmű, a sebességnél viszont nem. Nem akartuk, hogy csak előrefelé tudjon menni a járművünk, mivel a valós világban is tudnak tolatni az autók, így döntöttünk a szimmetrikus limitek mellett (Van olyan háló, ami ezt meg is tanulta kihasználni).
Megpróbáltuk modellezni egy nem éppen szokványosnak számító fizikai jelenség, a kicsúszásának a fizikáját. Ennek a megvalósítása csak részben fizikai alapon történik, de egy egész jó közelítést ad a valós viselkedésre. Ha túl nagy a kiskocsink sebessége, akkor az eddigi körpályáról egy nagyobb sugarú körpályára fog csúszni egy bizonyos csúszási faktorral skálázva a csúszás mértékét. Ezt a csúszást be és ki is lehet kapcsolni.

Kézi vezérlés

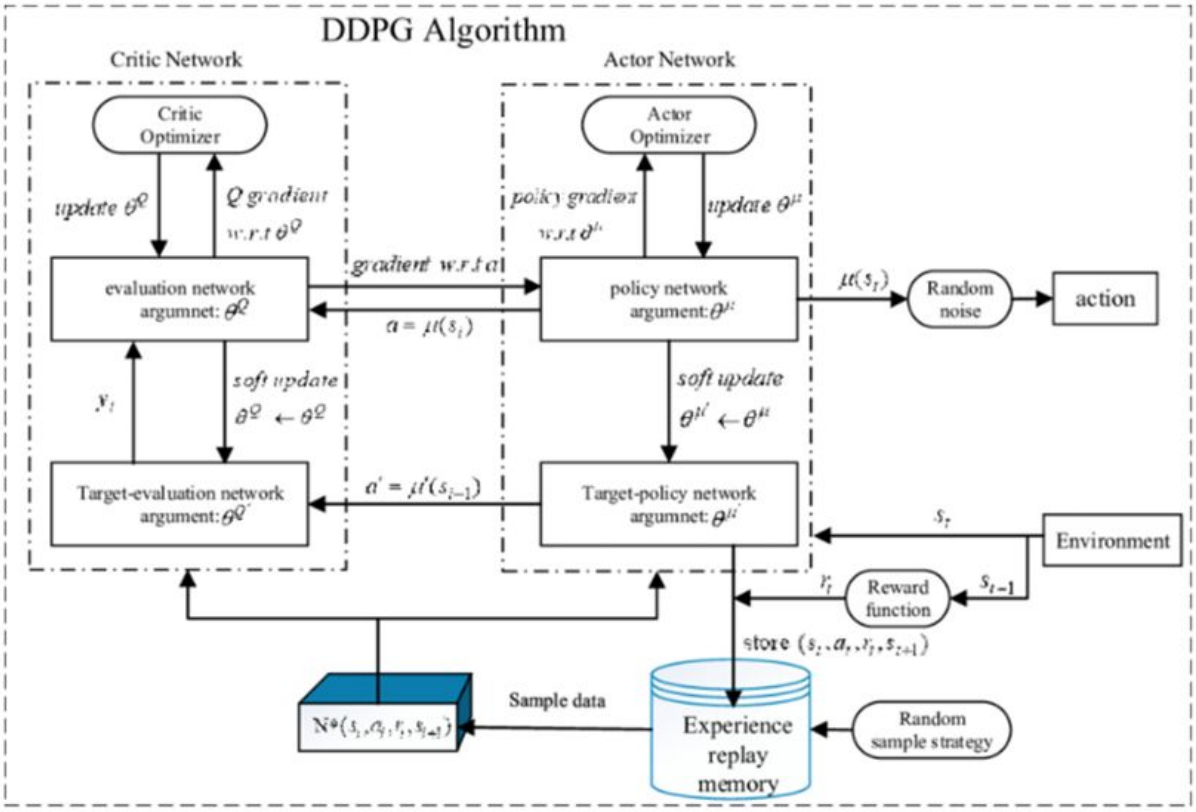
A neurális hálóval való tanítás mellett fontosnak éreztük, hogy mi magunk is ki tudjuk próbálni a kreálmányunkat, és ezért írtunk egy kis külön függvényt, amelynek a segítségével saját kezünkbe vehetjük az irányítást, és össze mérhetjük ügyességünket a betanított neurális háló tudásával szemben.
A kézi irányítás egyszerűen az iránygombok segítségével történik. A felfelé mutató iránygombbal gyorsítjuk, a lefelé mutatóval fékezzük az autót, a két oldalsó gombbal pedig kanyarodni tudunk. A sebességet irányító gombokkal is mindig bizonyos Δv sebességnövekedést/-csökkenést adunk a már meglévő sebességhez, ugyanez igaz a kormány szögére is.

Környezet együttműködése a tanító algoritmussal

Tanító algoritmusnak a Keras-RL csomagból használtuk fel a DDPG algoritmust. A reinforcement learning az ebben található algoritmusokkal úgy valósítható meg, hogy az ügynökhöz hozzá kell rendelni egy környezetet a gym csomagból vagy egy ahhoz hasonló felépítésű, megadott függvényekkel rendelkező egyéni környezetet. Mi egy ennek megfelelő osztályt hoztunk létre az autónak. Kipróbáltuk még a Stable Baselines csomag DDPG algoritmusát is, de ennek nem foglalkoztunk komolyabban a paraméterezésével.

A tanító algoritmus

Olyan reinforcement learning algoritmust kellett választanunk, amelynek több valós szám a kimenete. Erre a Deep Deterministic Policy Gradient (Lillicrap et al., 2015) módszer tűnt a legalkalmasabbnak.



4. Ábra: DDPG felépítése

A DDPG 4 neurális hálózatot használ a tanításhoz:

1. Critic hálózat (Q hálózat), mely a jövőbeli összes jutalmat próbálja megbecsülni
2. Actor hálózat (policy hálózat), melynek bemenetei a megfigyelések, kimenete kormányszög és gázpedál állás
3. Target critic
4. Target actor

A target hálózatok szerepe az, hogy mikor a hálók súlyait frissítjük, a cost függvény számolásánál ezekkel becsüljük a következő állapotban a hosszútávú jutalmat, ha az actor vagy critic hálózatokat használnánk fel erre a számolásra, könnyebben lenne divergens a rendszer, a target hálózatok tehát a stabilitást javítják.

A tanítás folyamata a következő:

- A környezettől megkapjuk a megfigyeléseket, melyeket majd a következő megfigyelésekkel, valamint a pillanatnyi jutalommal együtt eltárolunk egy bufferben
- A bufferből random mintavételezett batcheken tanítjuk a Critic hálózatot úgy, hogy a critic jóslata a pillanatnyi megfigyelésekre megegyezzen a tényleges pillanatnyi jutalomnak és a target criticnek a következő állapotra vonatkozó, hosszútávú jutalom becslésének összegével. Így a critic a hosszútávú jutalmat tanulja. A random

mintavételezés célja, hogy a minták eloszlása egyenletes legyen, ha a legutóbbi pillanatokat néznénk, akkor azok egymástól függenek.
- Frissítjük az actor hálózat súlyait. A számoláshoz a critic hálózattal becsült jutalom gradiensét használjuk fel.
- Frissítjük a target hálózatok súlyait úgy, hogy azok lassan követik a másik két hálózatot
- Az actor hálózat megadja a kimeneteket, de ehhez még random zajt teszünk, ami a felfedezést segíti elő, a háló kevésbé ragad meg egy lokális optimumban. A publikációban Ornstein-Uhlenbeck metódust használtak, ami korrelált, 0 középpontú zajt ad a kimenetre, így nem lehetetleníteni el a tanulást. Ők konstans σ paramétert használtak, ami a zaj nagyságát jellemzi, mi ezt nagy értékről indulva csökkentettük a tanulás során.

A hálók optimalizálását az eredeti publikációban ADAM-mal végezték, mi is ennél a módszernél maradtunk.
Az algoritmus a tanítás elején nem változtat a hálókon, ennek az az oka, hogy a bufferben elegendő mintát kell összegyűjteni ahhoz, hogy amikor ezekből mintavételezünk, azok függetlenek legyenek. Az actor és critic hálózatok esetén az ilyen kezdő lépések száma a *warmup steps* paraméterekkel külön állítható. Mi a critic hálózat esetén kis lépésszámot adtunk meg, mert úgy tapasztaltuk, hogy így is gyorsan tud tanulni, az actort csak 4000 lépés után kezdtük tanítani, így a critic addig tudott javulni, egyes esetekben már az actor tanítása előtt körbe tudott menni a pályán a kocsi a critic jóslatát felhasználva (ilyenkor nem az actor irányít, hanem a Q függvényt maximalizáló kimenetet adjuk a környezetnek).

Mikor a kutatást elkezdtük, milyen algoritmust lenne érdemes választanunk, először olyan algoritmussal terveztük a tanítást, ami falba csapódásig nem módosít a hálón, és a jutalmat a teljes kör alapján számolja. Ez feltehetőleg azon felül, hogy számításigényes, nem arra tanítaná az autót, az adott megfigyelésekre hogyan reagáljon, hanem arra, hogy a megadott pályákon tovább menjen, de feltehetően ez nehezen konvergálna és ismeretlen pályán rosszul teljesítene. A DDPG viszont azáltal, hogy a várható jutalmat próbálja megbecsülni a pillanatnyi állapotból, gyorsabban tanul, és rendszerint konvergál is egy olyan állapotba, ahol sok jutalmat szerez, csak a tanításhoz szükség van egy jó jutalmat számoló függvényre.

Nehézségek

Bemenetek normalizálása

Kezdetben nem sikerült eredményeket elérnünk, ehhez szükséges volt normalizálnunk a bemenetet. A sebességet ez nem érintette, csak a mért távolságokat transzformáltuk úgy, hogy átlaguk 0, szórásuk 1 legyen.

Jutalom megadása

Talán a legnagyobb kihívást a jutalom definiálása okozta. Csúszás nélküli esetben egyszerűen a sebesség volt a jutalom. Ez így nem működött jól, mert a jutalom értékei nagy tartományban mozogtak, 0 és 1 közé skálázva viszont jól működött a tanítás.
Nehezebb volt a jutalom függvény definiálása kicsúszás esetén. Ha csak a sebességet vettük figyelembe, akkor a kocsi a falba csapódva tudta maximalizálni a jutalmát. Ezért kezdetben a jutalmat egy kicsúszásra jellemző mennyiséggel büntettük. Ebben az esetben az autó olyan íven csapódott maximális sebességgel a falba, amin lényegesen kevesebbszer csúszott meg, mint amikor nem vettük figyelembe a csúszást. A csúszást ebben az esetben nem kapta meg a háló bemenetnek.
Ezután a jutalomnál azt vettük figyelembe, mennyire van a pálya közepén az autó, illetve mennyire áll párhuzamosan a fallal. Ezzel a módszerrel lényeges javulást tudtunk elérni.

Optimális méretű háló

A kísérlet célja

Megpróbáltuk kideríteni, hogy különböző mennyiségű keresővonalhoz mekkora háló illik, illetve ezeknek a felépítésére is kíváncsiak voltunk. Mindezek mellett, mivel nincsen meg a megfelelő összeállítás egyikünknél sem, ezért nem GPU-n, hanem processzoron tanítottuk az összes hálónkat, így elég lassú volt minden egyes tanítás, s ezen probléma kiküszöbölésére azt is meg akartuk nézni, hogy melyik háló tudja teljesíteni a pályákat a számunkra reális időkereteken belül.

Általános tudnivalók

Ahhoz, hogy releváns képet kapjunk az összehasonlításhoz, fixálnunk kellett egy pár paramétert.

- Warm up steps: 4000
- Training steps: 20000
- Step-ek maximális száma egy epochon belül tanítás közben: 500
- Step-ek maximális száma egy epochon belül teszt közben: 1000
- A memória mérete: 100000 step
- A critic és actor hálókban a minden egyes esetben megegyezik a rétegek típusa, száma és a neuronok mennyisége az adott layerekben.

- A *Reward* függvény mindenhol ugyanaz.
- A csúszást nem vettük figyelembe egyik esetben sem, mivel a tesztünk arra is vonatkozott, hogy egy alapot biztosítson arra az esetre, amikor már a kicsúszással is számolnunk kell.
- Keresővonalak száma is adott: 3, 5, 9, 15, 25 vagy 40

Összesen tizennyolc csoportra bontottunk fel ötven darab hálót a keresővonaljaik, rétegeik és neuronjaik számának a függvényében. Egy külön csoportosítást csináltunk csak a keresővonalak számának ismeretében is.
Minden egyes háló ezeknek megfelelően is lett elnevezve a következő formátumban:

k*x*_**l***y*_**n***z1*_*z2*_*z3*...

Ahol **k, l** és **n** a **k**eresővonalak, **l**ayerek és **n**euronok rövidítése, illetve *x*, *y* és *z1, z2, z3…* pedig mindezeknek a mennyiségét jelölik. Így könnyebben el tudunk igazodni a különböző modellek között anélkül, hogy be kellene töltenünk őket.
Megjegyzés: A mély háló elnevezés csak a feladathoz mért relatív mély hálót takar a későbbi felsorolásban. Igazán mély hálót nem terveztünk írni, mivel úgy gondoltuk, hogy a feladat bonyolultsága nem fogja igényelni azt.

I. Kevés keresővonal, sekély háló, kevés neuron (4db)

Aktivációs függvény: *tanh*
Keresővonalak száma: 3-5. Layerek száma: 1. Neuronok száma: 10-30
A legjobban teljesítő háló a tanulás és a tesztek szempontjából az öt keresővonallal és harminc neuronnal rendelkező háló volt, ám ez is csak részben tudott végigmenni pályákon. Jól szerepelt még a három keresővonalas és tíz neuronos koncepció is.
Számítási időben mindegyik képes volt pár perc alatt megtenni mind az előírt húszezer lépést.

II. Kevés keresővonal, közepes háló, kevés neuron (4db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 3-5. Layerek száma: 2-3. Neuronok száma: 10-50
A tanulás és tesztek alapján sokkal több, mint húszezer lépés szükséges ezen neurális hálók betanításához, mivel a *k5_l3_n30_50_30* hálón kívül mindegyik csak kanyarodásra volt képes a kezdőpozícióból kiindulva. A megnevezett háló viszont három könnyű tesztpályát tudott teljesíteni.
A számítási időt tekintve még mindig azt lehet mondani, hogy gyorsak.

III. Kevés keresővonal, közepes háló, sok neuron (4db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 3-5. Layerek száma: 2-3. Neuronok száma: 100-300
A tesztek alapján ez egy nagyon sikeres csoport, mind a *k5_l2_n100*-as és a *k5_l3_n200_300_200*-as hálók több pályán is végig tudnak menni. A három keresővonallal rendelkező hálózatok viszont nem tudták tartani a lépést, valószínűleg túl nagy már a neurális háló szerkezete nekik.

Futási időben még mindig a gyors kategóriában tartózkodunk. Úgy tűnik, hogy ez inkább függ a keresővonalaink számától, mintsem a háló felépítésétől.

IV. Kevés keresővonal, mély háló, kevés neuron (2db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 3-5. Layerek száma: 6. Neuronok száma: 10-30
A tanítási fázis során a háló egy igazán meglepő stratégiával állt elő a háló. Nem tudott egyenesen menni, inkább egy hullámmozgás formájában közlekedett a falak között. A könnyebb pályákra ez a technika elég volt, ám a nehezebb pályákat már nem sikerült teljesíteniük.
A processzort még mindig nem terheli le ezen hálók kiszámítása.

V. Kevés keresővonal, mély háló, sok neuron (2db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 3-5. Layerek száma: 6. Neuronok száma: 100-300
Mind a tanítás, mind a tesztek sikertelenek voltak. A három keresővonalas hálózat egy pályán talán végig tud menni. Több step-re lenne szükség, hogy rendesen tudjanak ők tanulni.
Ezen hálók már lassabban teljesítették a feladatokat, így összességében nem a legmegfelelőbbek ehhez a feladathoz.

VI. Közepes mennyiségű keresővonal, sekély háló, kevés neuron (4db)

Aktivációs függvények: *tanh*
Keresővonalak száma: 9-15. Layerek száma: 1. Neuronok száma: 10-30
Egész jó eredményeket produkáltak a tanulás és a tesztek során a nagyobb hálók ebből a kategóriából, ám túlságosan ragaszkodnak a jobb oldalához a pályának, ezért gyakoriak az ütközések. Kiemelkedik a *k15_l1_n30* háló.
Viszonylag gyorsan tudják teljesíteni a feladatokat.

VII. Közepes mennyiségű keresővonal, közepes háló, kevés neuron (4db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 9-15. Layerek száma: 2-3. Neuronok száma: 10-50
A *k15_l3_n30_50_30*-as háló teljesített a legkiemelkedőbben ebből a csoportból, majdnem végig tudott menni a tesztpályákon, ami nagy haladás a többihez képest. Valószínűleg több step-ből kellene állnia a tanítási fázisnak.
A tanítás sebessége is megfelelő, bár már nem olyan gyors, mint a három-öt keresővonalas esetekben.

VIII. Közepes mennyiségű keresővonal, közepes háló, sok neuron (4db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 9-15. Layerek száma: 2-3. Neuronok száma: 100-300
Összességében csak a *k15_l2_n100*-as háló tudta abszolválni a tesztpályák némelyikét, köztük az egyik legnehezebbet.
Számítási időben is sokkal lassabbak, mint az ugyanilyen kategória három-öt keresővonallal. De a tizenöt keresővonalas megoldások eddig nagyon működőképesek.

IX. Közepes mennyiségű keresővonal, mély háló, kevés neuron (2db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 9-15. Layerek száma: 6. Neuronok száma: 10-30
Sem a tanulás és teszt fázis, sem a számítások elvégzésére igénybe vett idő nem megfelelő.

X. Közepes mennyiségű keresővonal, mély háló, sok neuron (2db)

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 9-15. Layerek száma: 6. Neuronok száma: 100-300
A tesztek során mindkét háló végig tudott menni legalább egy pályán. A tizenöt keresővonallal ellátott háló megint jobban szerepelt.
Számítási időben időben is elégséges teljesítménnyel dolgoztak ezek a hálók.

XI. Sok keresővonal, sekély háló, kevés neuron (4db)

Aktivációs függvények: *tanh*
Keresővonalak száma: 25-40. Layerek száma: 1. Neuronok száma: 10-30
Tanulás szempontjából elégséges szinten teljesítettek, ezekhez a hálókhoz már sokkal bonyolultabb hálókra lenne szükség, vagy sokkal több step-re.
Itt már kiütközik a keresővonalak befolyása a tanulás és tesztek sebességére. A huszonöt keresővonalas hálózatok általában háromszor annyi idő alatt teljesítik ugyanazt a feladatot, mint a három keresővonalas hálók. Legalább jobban szerepelnek a teszteken. A negyven keresővonallal rendelkező neurális hálók számítási igénye hatalmas. Nagyon lelassítják a CPU teljesítményét. Átlagosan négyszer több időre van szüksége egy tanítási folyamatnak negyven keresővonallal, mint hárommal.

XII-XV. Sekély háló, sok neuron

Ezek a hálók érdemben nem tudtak mit nyújtani. Tanulni ezekkel a paraméterekkel és step számmal nem tudtak, és lassúak is voltak.

XVI-XVIII. Sok keresővonal - összes

Aktivációs függvények: *linear/ReLU*, *tanh*
Keresővonalak száma: 25-40.

Nagyobb teljesítményre lenne szükségük, esetleg jobb beállításokra ezeknek a hálóknak, mivel nagyon lassúak, és a tanulni is elég későn kezdenek el.

Tapasztalatok a kísérlettel kapcsolatban

Összességében teljesen változatos eredményekkel szolgált a kísérlet, mivel nem csak egy típusú háló vált be, hanem több fajta is. És a keresővonalaktól való függés is egy meglepő eredményt adott. Például a sekély és kevés neuront tartalmazó hálóknál a három és a huszonöt keresővonalas variációk teljesítettek jól, az öt, kilenc, tizenöt illetve a negyven vonalasok viszont nem. Ez arra enged következtetni minket, hogy nem feltétlenül első vagy másodfokú összefüggés áll fenn a háló komplexitása illetve a keresővonalak száma között, hanem valami sokkal összetettebb.
Itt meg is kell említenünk a legjobban teljesítő koncepciónkat, amely egy sekély hálót, illetve kilenc keresővonalat tartalmaz. (*k9_best_nn).* Ez nem csak a tanító pályákat teljesíti hibátlanul, de a teszt pályákon is 10/13 eredményt ér el (egy pálya nem teljesíthető az autóval egy éles kanyar miatt).

Kicsúszás

Fentebb említettük, hogy itt a legnagyobb nehézség egy olyan jutalom függvény megadása volt, ami tényleg a pályán körbeérést segítette elő. Végül olyan megoldással álltunk elő, amiben a sebesség jutalma és a pályán középen, illetve irányban maradást jellemző mennyiség jutalma is exponenciálisan tartott 1-hez, a kettő mennyiséget súlyoztuk. A függvény leegyszerűsítve:

$$reward = a\,(1 - e^{-b \cdot v}) + (1 - a)\,e^{c \cdot (p-1)}$$

ahol *v* a sebesség, *p* a középen maradást és pályával párhuzamosságot jellemző mutató. A függvényeket azért így határoztuk meg, mert kis sebességeknél gyorsan növelhető a jutalom, viszont nagy sebességeknél már nincs nagy különbség a jutalom sebességfüggő részében, így jobban számít, hogy az autó ne rossz helyzetben álljon a pályán. Hasonló megfontolásokból lett a *p*-től függő rész is exponenciális. A paramétereket több iterációban határoztuk meg, tanítás közben figyelve, hogy milyen viselkedéssel maximalizálja a modell a jutalmat. Ilyen megfigyelésekből egy idő után már jól tudtunk következtetni arra, mely paraméterek milyen módosításával közelíthetünk a kívánt viselkedéshez. Végül a fenti paramétereket így határoztuk meg: $a = 0,04$, $b = 20$, $c = 12$.

Eredmények

Itt már nem sikerült elérnünk olyan eredményeket, mint csúszás nélküli esetben, a 7 tanító pálya közül a legjobban betanított hálók is csak 4-5 pályát tudtak ütközés nélkül teljesíteni. A teszt pályákon pedig az eredmények igen rosszak, a hálók nem lettek robosztusak

Érdekes volt az a jellemző különbség, hogy a kisebb hálók azt tanulták meg, hogy sebességüket szabályozva, lassan mentek végig mindenhol. A nagyobb hálók ezzel szemben már azzal is próbálkoztak, hogy az egyenes részeken gyorsítsanak, kanyarban pedig lassan haladjanak. Érdekes példa erre az egyik pálya, mely kis méretű és közelítőleg

kör alakú. Itt azon kisebb hálók, melyek sikerrel megtanulták ezt a pályát, olyan sebességgel mentek végig, mellyel vagy nem, vagy csak kicsit tudtak csúszni. Ezzel szemben a pályát megtanuló mélyebb, nagyobb rétegenkénti neuronszámú hálók azt a stratégiát választották jellemzően, hogy a kormányt szélsőhelyzetben tartva, a sebességet (és ezzel a kanyarodás ívét is) megfelelően szabályozva mentek körbe.

Összefoglalás

Eredmények

A neurális háló felépítése nem okozott túl nagy problémát Keras-szal. A Keras-RL package segítségével érthetően és gyorsan paraméterezhető volt a tanító algoritmus.
A legnagyobb problémákat, mint ahogy fentebb már kifejtettük, a jutalmak, illetve a normalizálás jelentette. Amint ezeket kijavítottuk, és alkalmaztuk, sokkal jobb eredményeket kaptunk.
Végeredményként a kisautónk önmagában szépen végig tud menni pályákon megfelelő beállításokkal, valós fizikai környezet szimulálása nélkül. A kicsúszást szimulálva viszont nem tudtunk igazán jó eredményeket elérni.

Személyes tapasztalatok

Bekes Nándor

Számomra a legérdekesebb az volt a projektben, hogyan befolyásolható jutalomfüggvényekkel a tanítás eredménye, illetve az, hogy kipróbálhattam sok órán ismertetett módszer hatását (pl. normalizáció a bemeneten, regularizáció a háló rétegeinek tanításánál), és ezekkel tényleg jelentős javulást értünk el.

Csik Dávid

Eddig még nem foglalkoztam neurális hálózatokkal, s ezáltal eleinte elég nehéz volt megérteni a működését mind a Deep Deterministic Policy Gradient algoritmusnak, mind maguknak a neurális hálózatoknak a felépítését. Ezen felül nehéz volt átlátni számomra azt is, hogy a különböző környezetek hogyan kommunikálnak egymással.
Összességében már nem követnék el olyan hibákat, mint amilyeneket a projektünknek elején csináltam.

Ötletek

Röviden összefoglaljuk, milyen ötleteink voltak még, amelyeket nem valósítottunk meg, de a későbbiekben megvalósításra, kipróbálásra érdemesnek gondoltunk:

- A csúszás kezelése úgy, hogy a kormányzást egy jól betanított egy kimeneti neuronnal rendelkező háló végzi, és egy másikat tanítunk gázpedállal sebességet szabályozni. Ezzel a megoldással el tudnánk kerülni olyan trade-off problémákat,

amit a jutalom megírásának nehézségeinél mutattunk be, valamint egyszerre egy kimeneti neuront egyszerűbb lenne tanítani. Esetleg ezeket a modelleket egyesíthetjük úgy, hogy egy harmadik, két kimenetű hálót tanítunk meg arra, hogy ugyanazt a policyt kövesse, mint a különálló modellek. Itt természetesen ügyelni kell arra, hogy a tanításnál a ténylegesen előforduló bemenetek legyenek egyenletes eloszlásban mintaként felhasználva, a háló ebben a tartományban tudjon jó regressziót készíteni.

- A modell tanításába nem avatkoztunk be előre meghatározható ismeretekkel, például melyik kanyarban mekkora sebességgel lehet menni, mikor milyen szögben kellene álljon a kormány stb. Valamint megtehettük volna, hogy az autót kézi vezérléssel előre tanítjuk, a critic hálózatot feltehetően így egészen hatékony lenne tanítani, és rövidülne a szükséges futásidő. Mi ezeket nem valósítottuk meg, mert ki akartuk próbálni, hogy magára hagyva mennyire képes a DDPG használható eredményeket produkálni. A kicsúszás figyelembevételével ez már nem is működött igazán jól. Feltehetően minél több ismeretet használnánk fel, annál gyorsabb lenne a tanítás és robosztusabb is lenne a modell viselkedése.
- A kocsi fizikája igen egyszerű, mert úgy gondoltuk, hogy elegendő egy pontatlanabb, kisebb számítási igényű modellt használnunk, és ha ezzel jó eredményeket sikerül elérni, akkor érdemesebb a szimuláció részletességén javítani. A kicsúszást viszont a hálózatok nem tanulták meg megbízhatóan kezelni, ezért nem is készítettünk pontosabb dinamikai modellt, de a továbbiakban érdemes lenne ezen változtatni.

Források

Deep Deterministic Policy Gradient publikáció (Lillicrap et al. 2015)
arXiv article about DDPG

Deep Deterministic Policy Gradient
https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

Deep Deterministic Policy Gradient Picture (4.ábra)
https://www.researchgate.net/figure/Deep-Deterministic-Policy-Gradient-DDPG-algorithm-structure_fig3_338552761

ShipAI
https://github.com/jmpf2018/ShipAI