Sávkövetés Duckietown környezetben/Duckietown lane following challenge (2021)

Szász Zsolt, Fazekas Lajos, Kozák Áron

***Kivonat*—Témának az „Önvezető autózás a Duckietown környezetben", mert mechatronikai mérnökként az autonóm drónok és okos gyártórobotok, és így ez a téma állnak hozzánk a legközelebb. Munkánk során a duckietown-gym környezetben végeztünk szimulációkat. Célunk az AI Driving olympics Lane following (LF) kihívásához egy deep learningen alapuló megoldás elkészítése volt. Az általunk választott első megvalósítás az imitation learning volt. Egy emberi sofőr vezeti az autót, a képkockákat és az éppen lenyomott billentyűket lementjük, majd ezekkel az adatokkal tanítjuk a hálót. Imitation learninggel CNN és LSTM hálókat tanítottunk. Majd megpróbálkoztunk reinforcement learninggel is, egész pontosan a Deep-Q-learning algoritmussal.**

I. Bevezetés

Napjainkban egyre több és több cég foglalkozik önvezető autókkal, vagy a vezetést megkönnyítő egyéb okos berendezésekkel. Az egyik legelterjedtebb ilyen rendszer a sávtartó asszisztens (LDW és LCA), mely először 2000-ben jelent meg a Mercedes Actros kamionban. A mi feladatunk is lényegében sávkövetés.

Imitation learning …

A Deep-Q-learninget a DeepMind vállalat fejlesztette ki, és viszonylag hamar lenyűgöző eredményeket ért el. Az apró vállalatot 2014-ben vásárolta fel a Google csillagászati 500 millió dolláros összegért. 2020-ban az Agent57 hálót 2600 különböző ATARI játékra tanították be, és valamennyiben hozta, esetenként meg is haladta az emberi szintet.

II. Tématerület ismertetése

Az általunk használt környezet a Duckietown [1][3] nevet viseli. A projekt 2016-ban indult az MIT-n, majd nyílt forráskódú lett, így ma bármelyik egyetem használhatja oktatási célokra. Későbbiek során rájöttünk, hogy ez azt is magával vonta, hogy kifejezetten kevesen foglalkoznak aktívan a témával, tehát a környezet implementálása a projektbe kifejezetten sok inkompatibilitási problémával küzdött.

A Duckietown két fő részből áll. Egy szabványosított elemekből felépülő városból, melyet utak alkotnak, rajtuk sávok vannak festve, mellettük közlekedési jelzések találhatóak. A városban lehetnek gyalogosok, akiket gumikacsák szimbolizálnak, és más autonóm járművek. A várost habszivacsból és szigszalagból a valóságban is meg lehet építeni. A másik rész a Duckiebot, egy olcsón megvásárolható robot, mely egy kamera segítségével érzékeli a környezetét, és PWM jelekkel irányítja saját szervóit. A Duckiebotot, vagy agentet a saját kódunkkal kell irányítani, hogy teljesítse az alábbi feladatok valamelyikét:

- **Lane following (LF):** Egy járművel kell követni a jobb oldali sávot egy zárt hurok alakú, elágazás nélküli, akadálymentes pályán.
- **Lane following with vehicles (LFV):** Ugyanaz a feladat, de a pályán más autonóm járművek is közlekednek, melyeket el kell kerülni, valamint az álló akadályokat is ki kell kerülni.
- **Lane following with dynamic vehicles and intersections (LFVI):** Szintén a jobb sávot kell követni, álló és mozgó akadályok elkerülése mellett. De a pálya nem csak egy zárt hurokból áll, útkereszteződések is vannak.

Mi a legegyszerűbb kihívást, a sima sávkövetést választottuk (LF), mert ez egy kiváló belépő feladat kezdő fejlesztőknek. Továbbá a Linux alapú környezet megismerése rengeteg időnket elvitte, ezek után nem akartunk belevágni egy komplikáltabb projektbe a határidők miatt.

A Duckietown megépítése és a Duckiebotok megvásárlása azonban nem feltétel, a Duckietownhoz biztosítanak egy szimulációs környezetet, a Duckietown-gym -et, melyben egy teljesen virtuális szimulációban van lehetőségünk betanítani és futtatni a virtuális duckiebotunkat. Ez a környezet sajnos nem támogatja a Windowst, számunkra csak Linux alapú környezetben érhető el.

A projekt során a velünk szimultán dolgozó csoportok által felismert problémákat számításba vettük, és már elővigyázatos döntésekkel kezdtünk a szimuláció és tanítási folyamat felépítéséhez.

III. Rendszerterv

A célunk egy a kihívást teljesítő tanulási algoritmus létrehozása.A linux kernelű operációs rendszerek közül a Pop_os legfrisebb nem LST verzióját választottuk mivel az könnyen használható és nem futunk bele az Nvidia driverek hiányába az nvidia-docker környzet létrehozásakor.

A kitűzött maximális cél egy DeepQ-learning típusú CNN-LSTM modell létrhozása és tanítása. Ez a rendszer egy az értékelő függvény által adott pontot maximalizáló feladatot lát

el, miszerint kiszámolja az összes különféle kimenetéhez tartozó predikált pontszámot és a legnagyobbhoz, vagy egy bizonyos epszilon értéktől függően véletlenszerűhöz tartozót frissít a végrehajtott művelet után visszakapott érték alapján.

Ehhez először meghatároztuk basline-ként, hogy egy imitation learning alapú CNN hálót tudjunk tanítani. Ehhez az általunk megírt *imitation_learning.py* képes kimenteni a szimuláció képkockáit, a hozzá tartozó lenyomott billentyűt, valamint elkészít egy listát amiben feljegyezzük a lefutott „körök" és a hozzá tartozó képkockák számát. Ha legeneráltuk az adathalmazunkat a konvolúciós modell képes lesz az adott útfestéshez megtanulni a megfelelő bemeneti értékeket.

Második lépésként az egyszerű CNN hálóhoz képzett adatokkal egy CNN-LSTM hálót tanítsunk, ami főként a KERAS Framework ConvLSTM2D típusú rétegeivel dolgozik, ezt extra funkcionalitásként más képes lehet objektum kikerülésre is. Ennek a modellnek az előnye, hogy az aktuális útszakasz képe kontextusba van helyezve ez nagyban javítja a kimenet minőségét már kis epoch számnál is.

Végül a DeepQ-learning modell elkészítése és implementálása a GitHub: duckietown/gym-duckietown-agent címen futó repository által biztosítot Agent struktúra irányítási elemeként. Ezen struktúra legnagyobb különbsége, hogy egyszerre fut a szimulációval. A háló generál egy batch kimenetet az eddigi beállításai alapján, ezt a környezet reward function-je kiértékeli és a visszakapott értékek alapján módosítjuk a súlyainkat, majd ismételünk. (végül nem vettük igénybe ezt a repot, de működget a q learning)

IV. Megvalósítás, konvolúció

Kezdeti problémák: (probléma – megoldás/következmény)

- A Grub nem hajlandó a megfeleő adathordozóról bootolni – nem használunk utólag installált bootloader-t
- A leg kompatibilisebb operációs rendszer nem rendelkezik megfelelő driverekkel a videókártyán tanításhoz – Pop_os operációs rendszer használata
- A docker imagek generálása sorozatos hibákba fut több napi debugolás után is – docker környezet lehető legnagyobb mellőzése, a tanítás rendszeren futtatása
- Megszűnt a szükség a driverekre – képesek lettünk több feladat szimultán ellátására
- A cél háló (DeepQ) kezdő feladatnak túlságosan komplex – a feladat leegyszerűsítése és folyamattá alakítása
- Nincs adat generáló script – saját adatgenerátor létrehozása
- A virtuális gépek ineffektívek és gyengék rendszeren tanításhoz képest –rendszeren tanítunk

A hibák következtében csak CPU-n tudunk tanítani.

Először a 2D konvolúciós hálón alapuló Imitation learninget ismertetjük, mert ezt sikerült befejezni.

A. Adatok beszerzése, előkészítés

A futó szimulációból a képeket az `img = env.render()` paranccsal lehet kinyerni numpy array-ként. A kapott képeket a *prep.py* fáljnak a `preprocess(img)` függvénye végzi, ami az alábbi lépéseket hajtja végre a képen (mátrixon):

- **Konvertálás:** A numpy array-t PIL image formátummá alakítjuk a könnyebb kezelhetőség érdekében
- **Átméretezés:** A képet átméretezzük az eredeti 800x600-as felbontásról 80x60-as felbontásra, hogy kisebb hálóval is taníthassunk, kevesebb adatot kelljen tárolni.
- **Vágás:** A kép felső egyharmadát levágjuk, mert ezen csak a horizont fölötti ég látható. Illetve a pozíció adatok, de azok már az átméretezés után olvashatatlanok. A kapott kép 80x40 felbontású.
- **Küszöbölés:** A kép színét thresholding-gal módosítjuk, azaz a sötét pixelekből, az aszfaltból matt feketét csinálunk, a világos pixelekből pedig élénk sín lesz. A fehér oldalvonal teljesen fehér, a sárga sávelválasztó teljesen sárga lesz.

Az függvény ezek után egy PIL image-t ad vissza, amit majd később jpeg formátumban mentünk el, mert így lényegesen kisebb helyet foglal, mint egyből numpy array

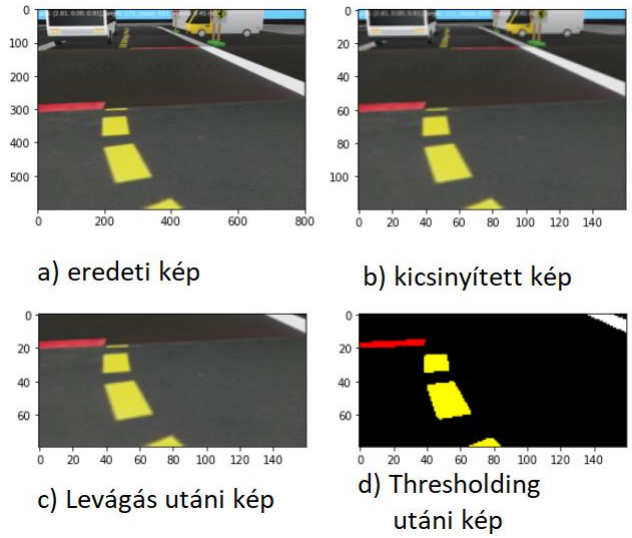

Fig. 1. A prepocessing vizualizálása. (ezen még nagyobb felbontást alkalmaztunk)

Maga a tényleges adatgyűjtés az *imitation_learning.py* -ban található. Ezt a Duckietown-gym *manual_control.py* módosításával hoztuk létre. Az importok után létrehozzuk a gym environmentet (`env`), azzal a külömbséggel, hogy alapértelmezetté tesszük a Duckietown-udem1-v0 környezetet, így rövidebb konzol paranccsal indítható, illetve a pályát (map) a zigzag_dist-re cseréltük, mert ebbe nincs elágazás és akadály, csak egy nagyobb hurok, amin az agent-ünk tud majd körbe-körbe menni. Továbbá átírtuk az

irányítást, hogy kényelmesebb legyen, ugyanis emberi sofőröknek kell adatokat termelnie a tanításhoz. A szimuláció gyorsabban fut, ezentúl normális ívben lehet kanyarodni, lehet menet közben kanyarodni, és kanyarodás közben nem megy hátrafele a robot.

Végül a fentebb leírt módon kinyerjük a `preprocess()` segítségével az előfeldolgozott képet a szimulációból. Ezen felül egy 5x1-es numpy tömbbe eltároljuk, éppen melyik billenytűt nyomja le a játékos (4 nyíl és a szóköz közül), így onehot enkódolású elvárt kimenetet kapunk, és kategorizálásként tanítható majd a háló. A kapott array-t és képet még el kell menteni. Ehhez szükség van egy egyedi névre, ami alapján majd be is tudjuk olvasni az adatokat. Ehhez felhasználjuk a menet indítása óta eltelt képkocka sorszámát, amit az `env.unwrapped.step_count` ad vissza. Ez azonban a program újraindításakor, illetve a játék végén (azaz amikor elhagyjuk az utat, vagy elérjük az 1500-as step_count-ot) resetelődik, ahogy meghívódik az `env.reset()` függvény. Ezért a menetek számát is beleírjuk a fájlnévbe. Ez a két szám már egyedi azonosítót alkot. A menetek számát egy külön *names.npy* fájlba tároljuk, hogy a program újraindítása esetén is lehessen folytatni a mentést, illetve majd tudjuk, hány adatot kell beolvasni.

B. Tanítás

A tanítást a *conv_training.py* végzi. A hálót a *conv_model.py conv_model()* függvénye generálja. Ebben a modellben 3 conv2D réteget használunk, majd egy Flatten réteg után még van 3 Dense réteg, és utána a kimeneti réteg. A végső hálóban használt részletes paramétereket lásd a képen. Sajnos rendes hiperparaméterezésre nem maradt időnk. Próbálkoztunk kicsit nagyobb hálóval is, de sajnos nem ért el jobb eredményt, ellenben jóval hosszabb lett a tanítás.

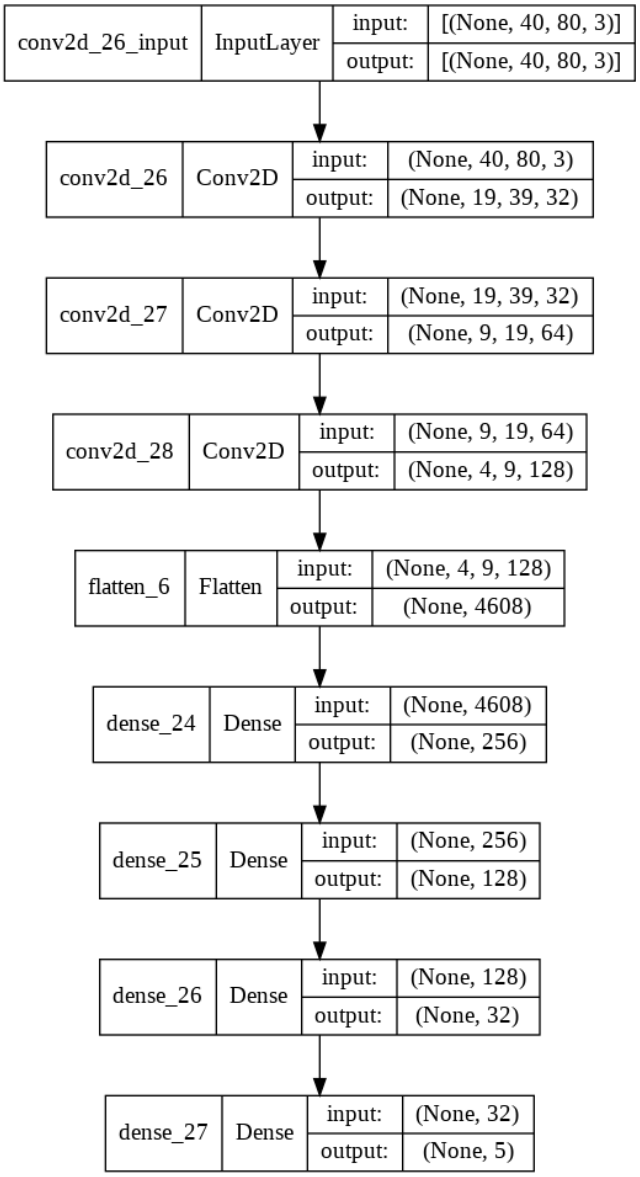

Fig. 2. A végső megoldásban használt háló felépítése. A konvolúciós rétegeket még egy-egy Batchnormalization és Dropout követi, amit nem ábrázoltunk

A tanítás batchenként lett megvalósítva, mert nem sikerült egyszerre ennyi adatot beolvasni. Ez főleg az LSTM hálónál probléma, ahol több egymást követő framet kell összefűzni. Először beolvasunk egy batch-nyi (512 darab) tanító képet, és hozzá a már onehot encode-olt kimenetet (hogy milyen billentyűt nyomtunk). A képeket utána még tömbbé kell konvertálni. Így tehát végigiterálunk ez epoch-okon, majd epoch-onként az összes adaton batch_size méretű adatokban végigmegyünk. Előre nem tudjuk, hány adat és így hány batch lesz, de azt tudjuk, hogy 200 menet van összesen, és így a menetekből tudunk validation splitet csinálni. Mivel túl sok adatunk lett, ami rengeteg ideig tanul, és az egész estén át tanított háló rosszabb eredményt adott, mint az első kisebb adatbázison tanult, ezért végül csak az utolsó 50 meneten tanítunk, amiből 10-et validálásra használunk. Így egyszerre csak batch_size-nyi elem van betöltve a memóriába, amiket a keras könyvtár `model.train_on_batch()` függvényével ráküldünk a modelre. Majd a szintén batch-enként

`model.evaluate()` függvénnyel kiszámoljuk a validációs accuracy-kat, és átlagoljuk őket az összes validációs batchre. Ha az accuracy nő, elmentjük a modellt, ha pedig csökken patience (alapból 5) epoch-on át, akkor leállítjuk a tanulást breakkel. A tanítás a *conv_train_on_batch.py* meghívásával automatikusan elindul, amennyiben vannak adatok a megfelelő mappában. (Amiket az *imitation_learning.py* tud generálni)

C. Tesztelés

A tanítás némileg semmitmondó adatokat, loss-t és accuracyt szolgáltat a hálónkra, ráadásul teszt adatbázisunk sincs. Helyette vizualizáljuk, hogy mit csinál a modell a Duckietown környezetbe helyezve. Ehhez el kell indítani a *play.py*-t. Ekkor megjelenik ablakosan a környezet, benne a duckiebottal és a pályával. Az irányítást ezúttal a tanított háló végzi. Ha meghal, automatikusan újraindul, így lehet látni, mennyire jól vagy rosszul tudja követni a sávot. A legjobb tanításról feltöltöttük a videót, sajnos időhiány miatt csak telefonról. így látható, mit alkottunk.

V. Megvalósítás, LSTM

Majd ismertetem röviden az LSTM modellt. Ezeket a sorokat már a határidőhöz közeledve írom, az LSTM tanítása sajnos sokkal lassabb (mint kiderült, nincs is megírva többszálúva, így 16 helyett csak 1 magon tanul), és egy újabb gépen meghalt a Linux, az egyetlen megmaradt laptoppal pedig épp konvolúciós hálót tanítunk. Elméletben az LSTM hálónak memóriajellege is van, emlékszik az előző time_step-nyi (itt előző 5 frame) időpillanatra, így tudja, hogy éppen mennyivel megy, kanyarodik-e. Ezáltal pedig nagyobb potenciál van benne, de mivel egy tanítóadat nem egy, hanem 5 adatból áll, jóval lassabb is.

A. Adatok beszerzése, előkészítés

A háló ugyanazokkal az adatokkal tanítható, mint az előző is. Itt azonban fontos, hogy sorrendben legyenek az adatok, bár a konvolúciónál se keverjük őket, de ott megtehetnénk.

B. Tanítás

A modellt a *lstm_model.py* állítja elő. Hagyományosan Timedistributed Conv2D layereket követ 2 vagy több sima LSTM majd, majd azok kimeneteire Dense rétegeket kapcsolunk. Sajnos azonban ezt a megoldást nem bírtuk lefordítani. Így mi a kettő ötvözeteként 3 darab ConvLSTM2D réteget alkalmazunk, amiknek az 5 kétdimenziós kimenetét egy Conv3D rétegbe adjuk, amit már ki lehet teríteni Flatten réteggel. Ezután még egy rejtett Dense réteg és a kimeneti réteg következik.

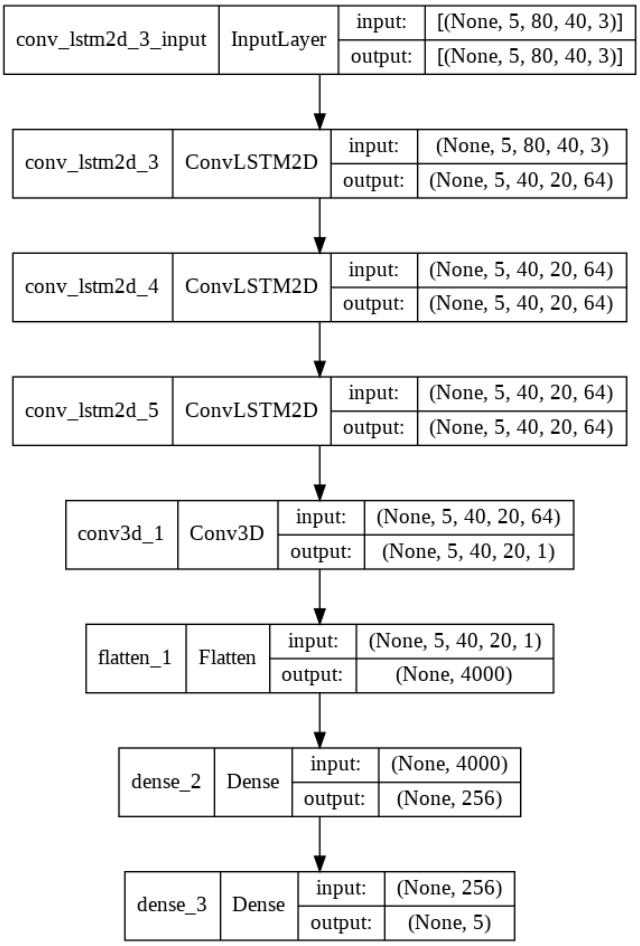

Fig. 3. Az LSTM hálónk felépítése. A rétegeket egy-egy Batchnormalization réteg követi, amit nem rajzoltunk oda.

Az egyetlen fontos különbség, hogy a háló bemenete nem csak az aktuális képkocka, hanem az előző 4-is. Ezeket a tanításnál a *lstm_train_on_batch.py*-ban további előfeldolgozásként össze kell tapasztani.

C. Tesztelés

A tesztelés is ugyanúgy működik, de itt is az előző 5 bemenetet várja a háló. Az indulásokkor ezért néhány frameig előre megyünk. Tesztelni az *lstm_play.py* meghívsával lehet. Végül a hálónkat csak 3 epochon át tudtuk trainelni, és a kapott háló csak egyenesen megy a szimulációban.

VI. Megvalósítás, Q-learning

A hálót viszonylag későn kezdtük el tanítani, e sor írásakor még javában dolgozik, de már látható néha, hogy sikerülget kanyarodnia. Majd leadáskor levideózzuk a szimulációt. Valószínűleg 200 epochon lesz trainelve.

A. Tanítás

A Q-learningnek egy sokkal egyszerűbb hálója van, amit a *q_model.py* generál. Ebben mindössze 3 Conv2D és egy Flatten után 2 Dense réteg található. A hálóhoz ugyanolyan előkészítés kell, mint a korábbiakhoz.

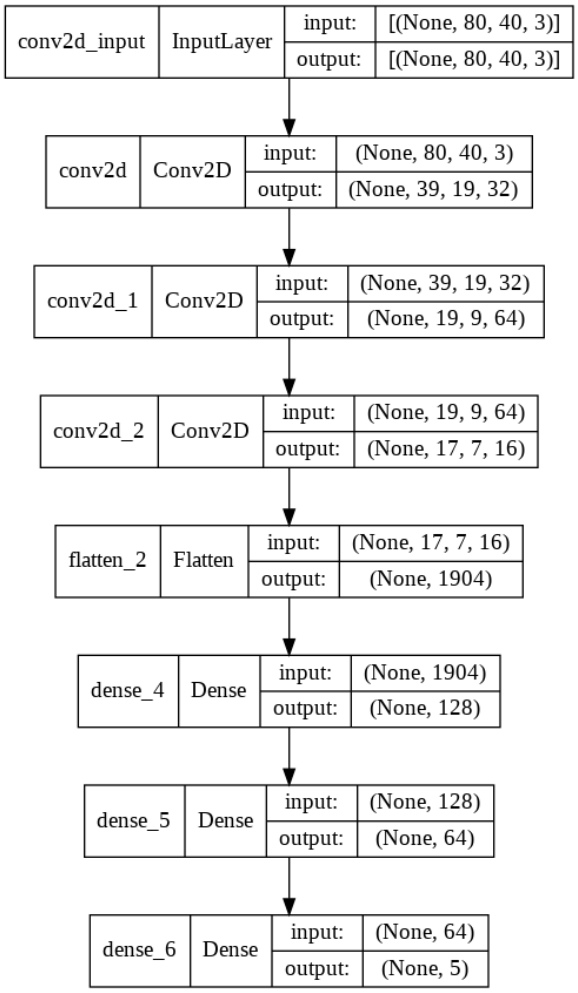

Fig. 3. A Deep-Q-learning hálója.

A tanítás a *q_train.py* meghívásával indul. Először létrehozunk egy environmentet. Ez és a step ki van szervezve a *create_env.py* fáljba. Ekkor indul az epoch. Először observationtime (alapból 500) darab megfigyelést futtatunk. Azaz epsilon valószínűséggel véletlenszerű akciót hajtunk végre, 1-epsilon valószínűséggel a modellünk predikált lépését hajtjuk végre. A bemenetet, végrehajtott akciót, a következő bemenetet, és a jutalmat (reward) elmentjük egy listába. Az epsilon idővel csökken, minden epoch végén megszorozzuk decayyel. Másodszor betöltjük 50-es batch-ekbe a listáról a megfigyeléseket. A tanítóadatok a bemenetek lesznek. Az elvárt kimeneteket a Bellmen egyenlettel számítjuk ki, amihez szükség van a rewardra, a következő bemenetre és a végrehajtott akcióra is. A kapott bemenettel és elvárt kimenettel batchenként tanítom a hálót. Ha minden batch lefutott, csökken az epsilon és indul a következő epoch.

B. Tesztelés

Hasonlóan tudja a kapott háló irányítani a szimulációt a *q_play.py* meghívásával.

VII. Jövőbeli tervek, összefoglalás

Mindhárom háló működőképes, de a tanítások rövidek, kevés adatot használnak. Nagyságrendileg gyorsítani lehetne, ha sikerülne rábírni a gépet, hogy GPU-n tanítson. A lenti lehetőségek megvalósításának feltétele, hogy sokkal gyorsabban lehessen tanítani.

Fejlesztési lehetőségek:

- Sokkal jobb eredményeket lehetne elérni több adattal. (Jelenleg 200 menetben 200 000 frameünk van, de egyszerre legfeljebb a negyedén tanítunk, hogy belátható idő alatt végezzen. Újratanításnál más részét vesszük az adatbázisnak. Akár 5-szörennyi adattal is lehetne tanítani)
- Nagyobb hálókat, több és nagyobb Conv2D és Dense rétegekkel is be lehetne tanítani, azok job eredményt adnának
- Lehetne tovább tanítani. Early stoppingnál nagyobb patiencevel, reinforcement learningnél (q learning) pedig sokkal több epochig.

VIII. Köszönet

Köszönjük a segítséget:

- Facskó Vincének, aki rengeteg adatot generált imitation learninghez. Ami végül túl sok is volt, és egyszerre csak egy részén tanítottunk.
- Nagy Viktornak, aki segített setupolni egy virtuális gépen Ubuntu 20.04-et. Ami végül meghalt.
- Kozák Balázsnak, aki segített setupolni Ubuntu 20.04-et egy laptopra. Amit végül nem használtunk, mert több 10 éves, és iszonyat lassú.

References

Examples:

[1] Duckietown Book The AI Driving Olympics: https://docs.duckietown.org/daffy/AIDO/out/index.html
[2] Francisco S. Melo, „Convergence of Q-learning: a simple proof”
[3] http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf
[4] Duckietown Git-hub https://github.com/duckietown/gym-duckietown
[5] Barto A, „Reinforcement learning” ISBN 987-0-08-053739-9