BABEŞ-BOLYAI UNIVERSITY CLUJ–NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

License Thesis

Reinforcement Learning methods applied in Mario

Abstract

Reinforcement learning is a subsection of machine learning. It's purpose is to teach some software agent to take adequate actions, so that he can maximize a given cumulative reward.

The aim of this dissertation is to analyze and compare some of the widely used reinforcement learning algorithms. In order to achieve our goal we needed to choose an environment which can be easily modeled to apply these methods. The model presented in this paper is based on the well-known game Mario. In addition to the comparison of these algorithms, the dissertation's other objective is to modify these methods in such a way, that we can maximize the results for the game of our choice.

In conclusion we can state that these algorithms depend heavily on the environment model and they exploit every inaccuracy of it. Therefore, we had to develop and specify our environment model continuously.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

JUNE 2017 SCHNEBLI ZOLTÁN

ADVISOR:
SZENKOVITS ANNAMÁRIA, ASSISTANT

BABEŞ-BOLYAI UNIVERSITY CLUJ–NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

License Thesis

Reinforcement Learning methods applied in Mario

SCIENTIFIC SUPERVISOR:
SZENKOVITS ANNAMÁRIA, ASSISTANT

STUDENT:
SCHNEBLI ZOLTÁN

JUNE 2017

UNIVERSITATEA BABEŞ-BOLYAI, CLUJ–NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licenţă

Metode de reinforcement learning aplicate in jocul Mario

CONDUCĂTOR ŞTIINŢIFIC:
SZENKOVITS ANNAMÁRIA, ASISTENT UNIVERSITAR

ABSOLVENT:
SCHNEBLI ZOLTÁN

IUNIE 2017

BABEŞ-BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Államvizsga-dolgozat

Visszacsatolásos tanulási módszerek a Mario játékra alkalmazva

TÉMAVEZETŐ:

SZENKOVITS ANNAMÁRIA,
EGYETEMI TANÁRSEGÉD

SZERZŐ:

SCHNEBLI ZOLTÁN

2017 JÚNIUS

Tartalomjegyzék

1. fejezet

Bevezető

Összefoglaló: *A gépi tanulás alapjai lesznek bemutatva, a visszacsatolásos tanulással a középpontban.*

1.1. A gépi tanulás

A gépi tanulás az a tudomány, amely célja, hogy a számítógép folyamatos iteráció során adatok és tapasztalatok általánosítása alapján képes legyen a folyamatos fejlődésre és végeredményként elérje egy konkrét feladat sikeres elvégzését [Domingos, 2012]. Ahogy egyre több adat áll rendelkezésünkre, annál nehezebb feladatokra vagyunk képesek megtanítani a számítógépünket.
Főbb kategóriái:

- Felügyelt tanulás (supervised learning) - megpróbáljuk modellezni a kapcsolatot a bemeneti és kimeneti adataink között. Rendelkezünk tanító-, teszt- és opcionálisan validációs halmazzal, amiket felcímkézünk a feladatnak megfelelően. A tanítóhalmaz alapján fogja a számítógép megtanulni a kapcsolatot a bemenetek és kimenetek között. A teszthalmaz segítségével tudjuk megmondani, hogy mennyire tanulta meg jól a feladatot a számítógép, míg a validációs halmazzal tovább tudjuk finomítani a modellezést. Főbb felügyelt tanulási módszerek: osztályozás (pl. emailek spam - nem spam osztályba való sorolása), regresszió (pl. egy ház árának becslése bizonyos paraméterek, illetve egy tanítóhalmaz alapján).

- Felügyeletlen tanulás - nem rendelkezünk felcímkézett tanítóhalmazzal. Az adathalmazunkban valamilyen struktúrát, szabályt keresünk. Felügyeletlen tanulási módszer például a klaszterezés (pl. ügyfelek csoportosítása vásárlási szokásaik alapján).

- Félig felügyelt tanítás - a tanítóhalmazunknak csak egy bizonyos része van felcímkézve. Ide tartozik a visszacsatolásos tanulás is, ami egy úgynevezett ügynök köré épül. Ennek az ügynöknek a célja az, hogy megfigyelései alapján, amiket egy környezettel való interakció során szerzett, cselekedeteket hajtson végre úgy, hogy közben maximalizálni tudja a jutalmait. A visszacsatolásos tanulást leginkább a robotikában használják. Például a japán Fanuc [Fan] cég olyan robotokat fejleszt, amelyek különböző feladatokat (például egyik dobozból tárgyakat rakni át egy másik dobozba) képes megtanulni egy éjszaka alatt [Rob].

1.2. Dolgozat áttekintése

A dolgozatunk célja egy olyan visszacsatolsásos ügynök létrehozása, amely képes átvinni a Mario játékot. E cél megvalósításához több módszerrel is próbálkoztunk.

A következő fejezetben bemutatásra kerül a visszacsatolásos tanulás elméleti háttere és megoldási módszerei 2. Majd az *Alkalmazás* című fejezetben (3) láthatjuk hogyan épül fel az alkalmazásunk és hogy miként alkalmaztuk ezeket a módszereket benne. Az eredményeinket a 4-es fejezetben mutatjuk be. Végül, a továbbfejlesztési lehetőségeket az 5-ös fejezetben soroltuk fel.

2. fejezet

Visszacsatolásos tanulás

Összefoglaló: *Visszacsatolásos tanulás alapjait taglaljuk.*

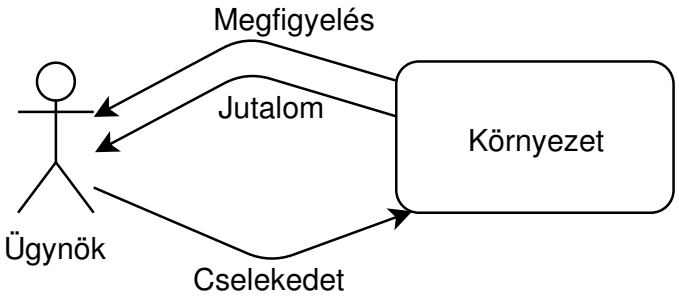
Ebben a fejezetben a visszacsatolásos tanulás lesz részletesen taglalva. Bemutatásra kerülnek ebben a doméniumban használatos alapfogalmak, általános megoldási módszerek és továbbá három konkrét módszer.

2.1. A visszacsatolásos tanulás alapjai

A visszacsatolásos tanulás az a gépi tanulási módszer, amely úgy próbál bizonyos szituációkhoz cselekvéseket társítani, hogy közben egy jutalmat maximalizál. A többi gépi tanulási módszerrel ellentétben, a tanulónak, akire a továbbiakban *ügynökként* fogunk hivatkozni, nincs megmondva, hogy milyen cselekedeteket hajtson végre. Fel kell fedezze a környezetét és meg kell tapasztalja, hogy melyek számára azok a lépések, amelyek a legnagyobb jutalmat reprezentálják [Sutton és Barto, 1998].

A felügyelt tanulással szemben, ahol a gép tesztadatok alapján próbál adott bemenetekre kimenetet származtatni [Russell et al., 2003], a visszacsatolásos tanulás központjában egy jutalom-orintált ügynök áll [Quah és Quek, 2006].

Minden ügynöknek van egy célja, amit a lehető leghatékonyabban szeretne elérni. Tudja érzékelni a környezét, tud cselekedni, viszont a cselekedetei megváltoztatják a körülötte lévő teret. Az ügynök nem tudja, hogy mennyire hatékony lépést választ magának hosszú távon, csupán egy pillanatnyi jutalmat kap cselekedetei után és érzékeli az új állapotot amibe került. Annak érdekében, hogy maximalizálni tudja a jutalmait, az ügynök minél több *tapasztalatot* kell szerezzen a környezetéről, cselekedetei következményeiről és a jutalmakról [Ribeiro, 2002]. Ezt a jelenséget megfigyelhetjük részletesebben a 2.1 illusztráción is, ahol az ügynök bizonyos ismeretekkel rendelkezve meglépi az általa választott lépést és ennek következtében jutalommal és további információval/tudással gazdagodik.



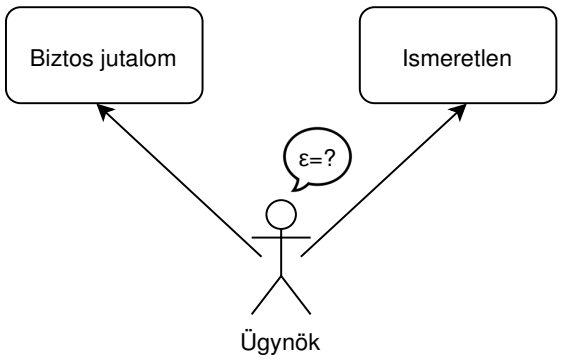
2.1. ábra. A képen megfigyelhető, hogy az ügynök a cselekedetei által kommunikál a környezetével. Rendelkezik a környezete állapotáról és ennek függvényében választja a lépését. Ennek következtében kap egy bizonyos számbeli jutalmat és érzékeli környezete változásait.

Az egyik legnagyobb kihívást az jelenti, hogy az ügynök cselekedetei a pillanatnyi jutalmon kívűl befolyásolják még a következő állapotokat és ennek következtében az *epizódban* szereplő összes, ez után következő jutalmat is. Így az ügynöknek nem csak a pillanatnyi, hanem a késleltett jutalmakból is tanulnia kell [Kaelbling et al., 1996]. Egy epizód olyan állapot-cselekedet-jutalom sorozatot jelképez, amely egy terminális állapotban végződik. Terminális állapot azt jelenti, hogy az ügynöknek nem áll rendelkezésére érvényes cselekedet az adott állapotban.

2.2. Felfedezés és kizsákmányolás

Az egyik kihívás ami csak a visszacsatolásos tanulásban van jelen, az a felfedezések és kizsákmányolások közötti megfelelő egyensúly megtalálása [Ishii et al., 2002]. Felfedezés alatt azt értjük, hogy az ügynök egy véletlenszerű cselekedet hajt végre egy bizonyos állapotban, míg a kizsákmányolás azt jelenti, hogy az ügynök az eddigi ismeretei alapján választja a számára legnagyobb jutalommal járó cselekedetet.

Annak érdekében, hogy az ügynök maximalizálni tudja a jutalmát, olyan cselekedeteket kell válasszon, amit a múltban már kipróbált és tudja róluk, hogy sok jutalmat generáltak számára. Ahhoz viszont, hogy felfedezze a legjobb cselekedeteket, ki kell próbáljon olyanokat is, amiket a múltban még nem próbált. Tehát az ügynök ki kell *zsákmányolja* az eddigi tudását annak érdekeben, hogy jutalomhoz jusson, viszont fel kell *fedezzen* új cselekedeteket is, hogy jobb döntéseket tudjon hozni a jövőben.
A dillemát az jelenti, hogy nem követhetjük csak a felfedezést, vagy csak a kizsákmányolás elvét, mivel biztos kudarchoz jutunk [Yogeswaran és Ponnambalam, 2012]. A legelterjedtebb megoldás e problémára az "ϵ mohó" ügynök (2.2), aki úgy próbálja megtalálni a felfedezések és kizsákmányolások közötti egyensúlyt, hogy az esetek 1-ϵ valószínűséggel követi az eddigi tudását és megpróbálja maximalizálni a jutalmát, míg a maradék ϵ valószínűséggel arra törekszik, hogy eddigi tudását bővítse, úgy hogy az adott állapotban egy véletlenszerű lépést visz véghez.



2.2. ábra. A kép az úgynevezett "ϵ mohó" ügynököt ábrázolja, aki az esetek többségében, pontosabban 1-ϵ valószínűséggel a biztos jutalmat választja, míg a maradék ϵ valószínűséggel, egy eddig számára ismeretlen, véletlen lépést hajt végre.

2.3. A visszacsatolásos tanulás elemei

Az ügynökön és környezeten túl, a visszacsatolásos tanulásnak további négy fő alkomponense van: *irányelv*, *jutalomfüggvény*, *értékfüggvény* és opcionálisan egy *modell* a környezetről.

Az *irányelv* meghatározza az ügynök viselkedését adott időben. Lényegében megmondja, hogy adott időben egy állapotban milyen valószínűséggel fogunk egy bizonyos cselekedetet végrehajtani [Sutton és Barto, 1998].

A *jutalomfüggvény* hátorzza meg az ügynök célját. Valójában minden állapot-cselekedet párhoz meghatároz egy értéket, egy úgynevezett *jutalmat*, ami a az állapot jóságát határozza meg. Az ügynök mindig megpróbálja maximalizálni a jutalmát és a jutalomfüggvény pedig megmondja, hogy adott pillanatban mely események gazdaságosak számára [Boyan és Moore, 1995].

A jutalomfüggvénnyel ellentétben az *értékfüggvény* meghatározza, hogy mi a célszerű hosszútávon. Tehát megmondja, hogy adott állapotból kiindulva, az ügynök mennyi jutalomhoz férhet hozzá. Függetlenül attól, hogy egy állapot azonnali jutalma kicsi, hosszútávon lehet hogy célszerűbb ezt választani, mivel értékes állapotok következnek utána [Sutton et al., 2000].

A környezet *modellje* utánozni próbálja a környezetet, vagyis egy adott állapotra és cselekedetre meg tudja jósolni a következő állapotot és a következő jutalmat. Tervezésre használják, ami alatt azt értjük, hogy úgy választunk cselekedetet, hogy számításba vesszük a lehetséges jövőbeli állapotokat, mielőtt még megtapasztalnánk azokat [Dayan és Niv, 2008].

2.4. Megoldási módszerek

A lentebb felsorolt módszerek a környezetre úgy tekintenek mint egy *Markov döntési folyamatra*. Ez egy olyan matematikai szerkezet, amely modellezi a döntéshozatalt, melyben a kimenet részben véletlenszerű és részben egy döntéshozó döntéseitől függ.

Egy Markov döntési folyamat egy $(S, A, P_{\cdot}(\cdot,\cdot), R_{\cdot}(\cdot,\cdot), \gamma)$ ötös által határozható meg [van Otterlo

és Wiering, 2012], ahol S és A a véges állapot- és cselekvéstér. A P függvény bármilyen állapotra (s) és cselekvésre (a) megadja a valszószínűségét a következő állapotnak:

$$\mathcal{P}_a(s, s') = Pr\left\{s_{t+1} = s' \mid s_t = s, a_t = a\right\}$$

Hasonlóan az R függvény, adott állapotra (s), cselekedetre (a) és bármilyen következő állapotra (s'), megadja a várható jutalmat:

$$\mathcal{R}_a(s, s') = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\}$$

$\gamma \in [0, 1]$ az engedmény faktor, amely meghatározza a pillanatyni és jövőbeli jutalmak között a fontosságot. Ha γ értéke a 0-hoz közeledik, akkor az ügynök hajlamos lesz arra, hogy csak az azonnali jutalmakat vegye figyelembe, míg, ha a faktor az 1-hez áll közelebb, akkor a jövőbeli jutalmakat próbálja maximalizálni [Watkins és Dayan, 1992].

2.4.1. Dinamikus programozás

A dinamikus programozás olyan algoritmusokat fed, amelyek adott környezeti modell alapján hatékony döntéseket tudnak hozni. Ezek az algoritmusok nagyon precíz modellt és nagyon sok számítást ígényelnek, mivel új döntéshozatal előtt az összes eddigi cselekedet esetén megvizsgálják, hogy mennyire volt hatékony. Mivel a környezeti modellt általában nem lehetséges előállítani, ezért a többi visszacsatolásos módszerek a dinamikus programozás ezen aspektusát próbálják kijavítani [Busoniu et al., 2010].

2.4.2. Monte Carlo módszerek

A Monte Carlo módszereknek *tapasztalatra* alapulnak, ami nem más, mint állapot, cselekedet és jutalom sorozat, amit vagy valós, vagy szimulált interakcióból kapunk. A valós tapasztalatból való tanulás azért fontos, mivel nem igényel modellt a környezetről, sem előzetes ismereteket a környezetről. Szimulált tapasztalaból való tanulás is hatásos lehet, viszont szükségünk van a környezet egy modelljére és időköltséges lehet szimulálni az átmeneteket [Doucet et al., 2001].

2.4.3. Időbeli-differencia tanulás (Temporal-Difference learning)

A visszacsatolásos tanulás legelterjedtebb gondolata az *időbeli-differencia* (TD) tanulás [Papadimitriou és Tsitsiklis, 1987]. Ez a módszer megpróbálja egyesíteni A Monte Carlo és dinamikus programozás módszerek elveit. Képes becsléseit valós időben firssíteni (nem szükséges megvárnia egy epizód végét) és tapasztalatból tanulni.

2.5. Időbeli-differencia tanulási módszerek

A legelterjetebb TD módszerek a Q-tanulás, Sarsa, R-tanulás, Ügynök-kritikus, TD(0) és TD(λ). A legegyszerűbb, a TD(0), egy állapotának a t. időpillanatbeli (s_t) jóságának ($V(s_t)$) a frissítését a

következő képlet alapján végzi:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)]$$

ahol γ a Markov döntési folyamatban levő gamma, α egy úgynevezett tanulási ráta, ami megmondja hogy az új ismeretek milyen mennyiségben befolyásolják az eddigi tudást és r_t a cselekedetért kapott jutalom [Sutton és Barto, 1998]. Minimális eltérésekkel, az összes TD módszer az előbbi összefüggés alapján módosítja a feltételezéseit egy állapotról.

2.5.1. Q-tanulás

Chripstopher J. C. H. Watkins doktori disszertációjában vezette be a visszacsatolásos tanulás világába a Q-tanulást, amely környezeti modell nélkül képes optimális irányelvet találni [Watkins, 1989]. (Képes megtalálni a *Q-függvényt*)

A módszer arra alapszik, hogy egy adott állapot-cselekedet átmenet jóságát egyből firssíti, miután a cselekedetet végrehajtottuk [Watkins és Dayan, 1992]. Legegyszerűbb formájában a Q-tanulás a következő képpen írható fel [Rummery és Niranjan, 1994]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Ahol:

- $Q(s_t, a_t)$ egy mátrix, amely az állapot-cselekedet átmenetek Q értékét tartalmazza;
- r_{t+1} - jutalom amit a cselekedetért kaptunk;
- α - a tanulási ráta ($0 < \alpha \leq 1$), ami azt mutatja, hogy az új tapasztalat mennyire fontos az ügynök számára. Amikor a tanulási ráta 0, akkor az ügynök nem tanul semmit, ellenben, amikor 1, akkor mindig csak a legújabb tapasztalatot tartja számon;
- γ - az engedmény faktor ($0 < \gamma \leq 1$), megegyezik a Markön döntési folyamatban lévő γ - val;
- $\max_a Q(s_{t+1}, a)$ - a becslésünk a következő állapot jutalmára;
- $r_{t+1} + \max_a Q(s_{t+1}, a)$ - a tanult érték.

Annak függvényében, hogy egy feladatban hány állapot és cselekedet áll rendelkezésre, a Q-tanulást kétféle képpen lehet implementálni. Ha relatív kevés állapot-cselekedet átmenetünk van, akkor érdemes a táblázatos módszert alkalmazni, ahol ezeket egy mátrixban reprezentáljuk. Amikor túl költséges a memóriában, egy mátrixban eltárolni az átmeneteket, akkor ezeket *neurális hálókkal* szokták modellezni. Az utóbbi módszerről a 2.5.3 fejezetben lesz részletesebben szó.

A 2.3-es illusztráción látható pszeudokód leírja egy ügynök tanulását, aki a Q-tanulás elvét követi. Minden epizód elején kell válasszon egy kezdőállapotot. Környezeti modelltől függően, ez lehet vagy

kötött, vagy lehet véletlenszűren választani. Ezután kiválasztja és végrehajtja azt a cselekedetet, amely a legnagyobb jutalommal jár, tehát legnagyobb a Q értéke a mátrixban. Megfigyeli a jutalmat és az állapotot amibe került és frissíti az eddigi tudását a tapasztalt változások alapján a képlet segítségével. Átlép a következő állapotba és ismétli ezen lépéseket, míg el nem ért a kívánt epizódszám végéig. Egy epizód addig tart, amíg végállapotba nem kerül.

Inicializáljuk a Q mátrixot
Ismételd Minden epizódra
    Inicializáljuk az s állapotot
    Ismételd
        Válassz egy *a* cselekedetet a Q-mátrixból
        Lépd meg *a*-t és figyeld meg a jutalmat *r* és az új állapotot *s'*
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$
        $s \leftarrow s'$
    Ameddig az *s* állapot nem terminális

2.3. ábra. Q-tanulás pszeudokódja [Sutton és Barto, 1998] könyve alapján. A kódrészletben használt $Q(s_t, a_t)$ egy mátrix, amely az állapot-cselekedet átmenetek Q értékeit tartalmazza, r_{t+1} a jutalom, amit egy cselekedet elvégzéséért kapunk, α a tanulási ráta, ami az új tapasztalat fontosságát jelképezi és γ az engedmény faktor, amely meghatározza a pillanatnyi és jövőbeli jutalmak közötti fontosságot.

2.5.2. SARSA tanulás

A Sarsa algoritmust, 1994-ben Rummery és Niranjan [Rummery és Niranjan, 1994] fejlesztette ki a Q-tanulás algoritmusából kiindulva.

Maga a "*Sarsa*" név egy rövidítés, amit R. Sutton ajánlott Rummeryéknek. A név a *State-Action-Reward-State-Action* sorozatból (Állapot-Cselekvés-Jutalom-Állapot-Cselekvés) származik. Már a módszer neve jelzi, hogy az állapot-cselekedet átmenetek Q értékeit milyen elgondolás alapján frissítjük. Ahogy az alábbi összefüggésben is látszik, egy állapot-cselekedet átmenet függ az eddigi Q értéktől, a jutalomtól, amit a cselekedetért kaptunk és a következő állapot Q értékétől.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

A Q-tanulással ellentétben, ahol feltételezve van, hogy egy állapot után a legmagasabb Q értékű állpotba megyünk tovább ($\max_a Q(s_{t+1}, a)$), a Sarsa tanulásban követjük az eddigi irányelvünket, vagyis előrevetítjük, hogy milyen állapotba mennénk tovább ($Q(s_{t+1}, a_{t+1})$) [Precup et al., 2001]. (2.4-es ábra)

Inicializáljuk a Q mátrixot
Ismételd Minden epizódra
    Inicializáljuk az s állapotot
    Ismételd
        Válassz egy *a* cselekedetet a Q-mátrixból
        Lépd meg *a*-t és figyeld meg a jutalmat *r* és az új állapotot *s'*
        Válassz egy *a'* cselekedetet a Q-mátrixból *s'* állapotból kiindulva
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\ [\ r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\ ]$
        $s \leftarrow s'$
        $a \leftarrow a'$
    Ameddig az *s* állapot nem terminális

2.4. ábra. SARSA-tanulás pszeudokódja [Sutton és Barto, 1998] könyve alapján. A kódrészletben használt $Q(s_t, a_t)$ egy mátrix, amely az állapot-cselekedet átmenetek Q értékeit tartalmazza, r_{t+1} a jutalom, amit egy cselekedet elvégzéséért kapunk, α a tanulási ráta, ami az új tapasztalat fontosságát jelképezi és γ az engedmény faktor, amely meghatározza a pillanatnyi és jövőbeli jutalmak közötti fontosságot.

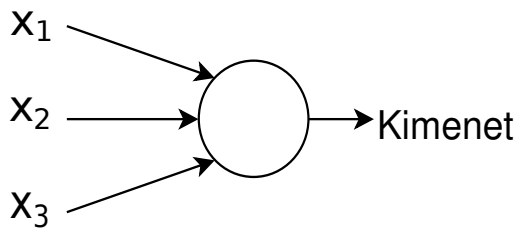
2.5.3. Deep Q-tanulás

Ahogy már említettük a 2.5.1 részben, ha az eseményterünk számossága meghaladja a memóriánk kapacitását, akkor le kell mondanunk a tömbös reprezentációról. Ilyen esetben kényelmes és hatékonyt megoldását nyújtanak a *mély neurális hálók* (angolul: *deep neural networks*).

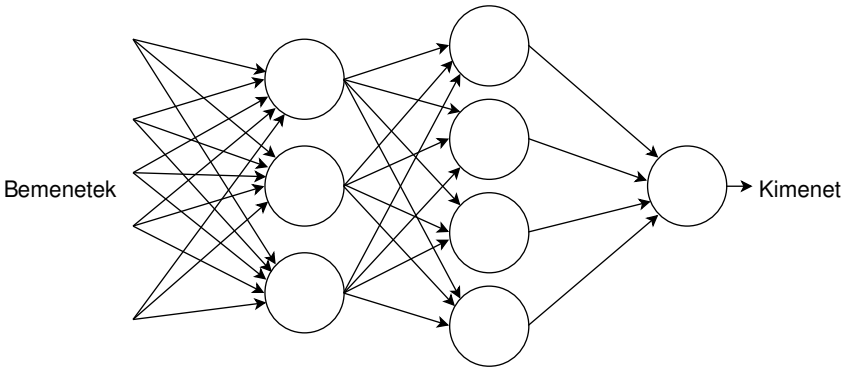
A standard neurális hálók egymáshoz kapcsolódó nódusokon, *mesterséges neuronokon* alapulnak [Schmidhuber, 2015]. (2.5) Ezek a neuronok három fő komponensből épülnek fel: bemenetből, súlyfüggvényből és aktiváló függvényből. Kapnak egy bemenetet, amit a súlyfüggvény felerősít vagy legyengít. Ezután az aktiváló függvény meghátorozza, hogy a neuron milyen mértékben aktiválódik. Mivel egy neurális hálóban több ezer neuron található, ezért a súlyfüggvényeinek megfelelő beállítása kulcsfontosságú szerepet tölt be a kívánt eredmény eléréséhez. Azt a folyamatot, amikor ezeket a súlyfüggvényeket szabályozzuk, tanításnak (trainingnek) nevezzük [Gershenson, 2003].

Ha a bemeneti és kimeneti réteg között több, mint egy rejtett réteg is található (2.6-os ábra), akkor *deep neurális hálóról* beszélünk [Schmidhuber, 2015]. Ezeknek a rejtett rétegeknek az a célja, hogy optimalizálja a tanulást. Segítségükkel képesek vagyunk komplexebb adatokat hatékonyabban modellezni [Bengio et al., 2009].

Azért érdemes a visszacsatolásos tanulás világában ezeket a modelleket használni, mivel a problémák sémáját könnyen rá tudjuk illeszteni a hálók szerkezetére. [Gu et al., 2016] A háló bemeneti rétegét a feladat állapottere, míg a kimeneti rétegét az ügynök mozgástere (cselekedet halmaza) fogja alkotni. Mivel jól tudnak függvényket közelíteni, ezért a Deep Q-tanulásban a Q függvényt így próbáljak előállítani.



2.5. ábra. Az ábrán egy egyszerű mesterséges neuron (perceptron) található, amely jelen esetben három bemenettel rendelkezik: x_1, x_2 és x_3, amelyekből majd egy kimenetet származtat.



2.6. ábra. Az ábrán egy deep neurális háló található, amelyen az utolsó két rejtett réteg látható, viszont a levegőből jövő nyilak sugallják, hogy még számos köztes réteg lehet a ezek és a bemenet között.

A Deep Q-tanulásban a legnagyobb kihívást az jelenti, hogy modellünk a Q függvényhez konvergáljon. Annak érdekében, hogy felgyorsítsuk a tanulási sebességet, fontos, hogy múltbéli átmenetekből, *tapasztalatból* is tanítsuk. Ezeket a tapasztalatokat túl költséges lenne eldobni első felhasználás után, ezért a rendszerüket folyamatosan ezekkel a régebbi emlékekkel tanítjuk [Lin, 1993]. Egy módszer ennek megvalósítására az lenne, hogy ezeket az információkat egy memória csomagban tároljuk és minden epizódus végén véletleszerű mintákkal tanítanánk a neurális hálónkat.

3. fejezet

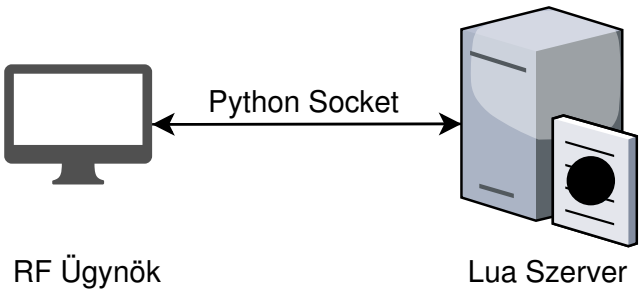
Alkalmazás

Összefoglaló: *Az alkalmazás lesz bemutatva.*

3.1. Alkalmazás áttekintése

Az alkalmazásunk célja különböző visszacsatolásos tanulási módszerek kipróbálása és ezek összehasonlítása. Ezért, nem lett volna célszerű túlbonyolítani az architektúránkat, mivel csak lassította volna a kommunikációt a komponenseink között. Ahogy a 3.1-es ábrán is látszik, projektünk két fontos részből áll: egy *Lua szerverből* (3.3) és egy *visszacsatolásos tanulás kliensből* (3.4). Ez a két modul *socketen* keresztül, *JSON* objektumokkal kommunikál egymással.

Socketnek nevezzük azt a kétoldalú kommunikációt két program között, amelyek egy halózaton belül futnak és ami segítségével ezek üzeneteket tudnak küldeni és fogadni egymás között. Ez a végpont az IP cím és portszám kombinációjából tevődik össze. Így a programok tudják, hogy melyik hálózaton és melyik másik programmal kommunikálnak. [SKT]



3.1. ábra. Az ábrán megtekinthetjük az alkalmazásunk architekturális felépítést. Két fő komponenssel rendelkezik: egy Lua szerverrel, amely környezetén belül fut az ügynökünk kö0rnyezete és egy RF ügynökből, ahol az RF a Reinforcement Learninget, vagyis a visszacsatolásos tanulást jelenti. Ez a két komponens Python Socketeken keresztül kommunikál egymással.

3.2. Üzenetek

Ahogy említettük, a két komponensünk JSON üzeneteken keresztül kommunikál egymással. Maga a JSON szó a JavaScript Object Notation kifejezés akronímája, amely egy szintaxist határoz meg arra vonatkozóan, hogy hogyan tároljuk és dolgozzunk az adatainkkal. Egy JSON objektum a következő képen néz ki: $\{"kulcs" : "érték"\}$, ahol az *érték* újabb ilyen párosokat foglalhat magába. A mi applikációnkba három fajta üzenetcsoportot különböztethetünk meg: *key*, *game* és *config*.

3.2.1. Key típusú üzenetek

Miután az ügynökünk meghatározta a számára legoptimálisabb lépést, a *key* típusú üzenetek segítségével tudja ezt elküldeni a szervernek (környezetnek). Miután elküldtük a szervernek a kívánt lépésünket, a szerver ezt meglépi és visszaküldi nekünk a játék állását a cselekedet utáni állapotban. Egy ilyen típusú kérésnek $\{"key" : \{"value" : "X"\}\}$ alakja van, ahol X a játékunk által felismert cselekedetek nevét veheti fel, vagyis a Fel, Le, Balra, Jobbra, Ugrás(A), Ok(B) és Start értékeket.

3.2.2. Config típusú üzenetek

A Config típusú üzenetcsoport a következő üzeneteket tartalmazza:

- $\{"config" : \{"divisor" : "X"\}\}$ - képesek vagyunk módosítani az elhagyni kívánt képkockáknak a számát; (frame divisor)
- $\{"config" : \{"image" : "X"\}\}$ - az emulátoron látott képnek a minőségét szabályozza;
- $\{"config" : \{"frame" : "X"\}\}$ - a szerver hánytól kezdje el számolni a képkockákat;
- $\{"config" : \{"speed" : "X"\}\}$ - milyen gyors legyen a játékmenet.

A legnagyobb fontossága az előbbiek közül a *frame divisor*-nak van, mivel ez fogja meghatározni, hogy a szerver hány képkockánként várja majd az ügynökünk cselekedeteit. Ha ezt túl nagyra állítjuk be, akkor az ügynök nem tudja érzékelni időben környezete változásait és így gyengén teljesítene. Mivel a szerver, miután megkapott egy cselekedetet, addig fogja "nyomva tartani a gombot" (ismétli az utoljára kapott cselekedetet) az emulátoron, ameddig egy újabbat nem kap. Ezért túl kicsi frame divisor esetén, bizonyos cselekedetek esetén, mint például az ugrásnál az ügynök nem tudja kihasználni a teljes mozgástartományát és így nem képes minden akadályon túljutni. Számos próbálkozás után arra jutottunk, hogy a legideálisabb az, mikor *két* képkockánként várja a szerver az üzeneteket. Így az ügynök időben észleli a közelgő veszélyeket és egyben van elég ideje ahhoz, hogy a cselekedeteket helyesen tudja végrehatjani.

A *speed* típusú config üzenet valójában három értékkel párosítható. Ezek a normal, maximum és turbo. Az utolsó kettő ugyanazt a hatást eredményezi: 10-szer gyorsabbra állítja a játék menetét, mint az alapértelmezett normál mód. Mivel az ügynök gyorsabban ki tudja számolni a számára optimális lépést, mint ahogy az emulátoron a játék előrehalad, ezért nagyon fontos és hasznos opció számunkra, hogy fel tudjuk gyorsítani a játékmenetet.

3.2.3. Game típusú üzenetek

A *game* típusú üzenetekkel az ügynök információt szerezhet környezetéről. Ebbe a kategóriába az üzenetek formátuma $\{"game" : \{"value" : "X"\}\}$, ahol X felveheti a következő opciók egyikét: Tiles, Image, Info és Reset.

Egy *Tiles* üzenettel a szerver visszaküldi az ügynöknek a környezete állapotát mátrix formában. Ahogy a 3.2-es ábrán is látjuk, az ügynök egy 13x13 as méretű tömböt kap, ahol minden tömbbeli elem értelmezési tartománya az $\{0, 1, 2, 3\}$.
Ezek jelentései:

- 0 - azt jelöli, hogy az adott pozíción nem található semmi;
- 1 - ábrázolja az összes semleges blokk és érmék helyzetét;
- 2 - jelöli az ellenségek helyzetét;
- 3 - jelképezi az ügynökünk helyzetét.

Az *info* típusú üzenettel az előbb leírt állapotleíráson kívül tudomást szerzünk arról, hogy ügynökünk hány élettel rendelkezik és hogy hány érmét vett fel a játék során.
Az *image* üzenet visszaküld egy képet a játék pillanatnyi állapotáról, míg a a *reset* típusú kérés törli a játék eddigi állását és elindítja újból a játékot.

0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	3	0	0	2	2	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

3.2. ábra. A jobb oldali ábrán láthatjuk, hogy az ügynökünk milyen formában "érzékeli" a környezetét, ami a bal oldali ábrán található. A 3-as mutatja az ügynökünk helyét, a 2-esek jelölik az ellenséges lényeket, az 1-esek reprezentálják a semleges blokkokat és érmékét, míg a 0-ok az üres helyet jelölik.

3.3. Lua szerver

Mivel a játékunkhoz (Mario), csak egy emulator segítségével tudunk könnyen kódot írni és platformfüggetlenné tenni, ezért szükségünk volt keresni egy ilyen környezetet. Választásunk az *FCEUX* [FCE] emulátorra esett, mivel ez egyben támogatja a Lua programozási nyelvet és platformfüggetlen is.

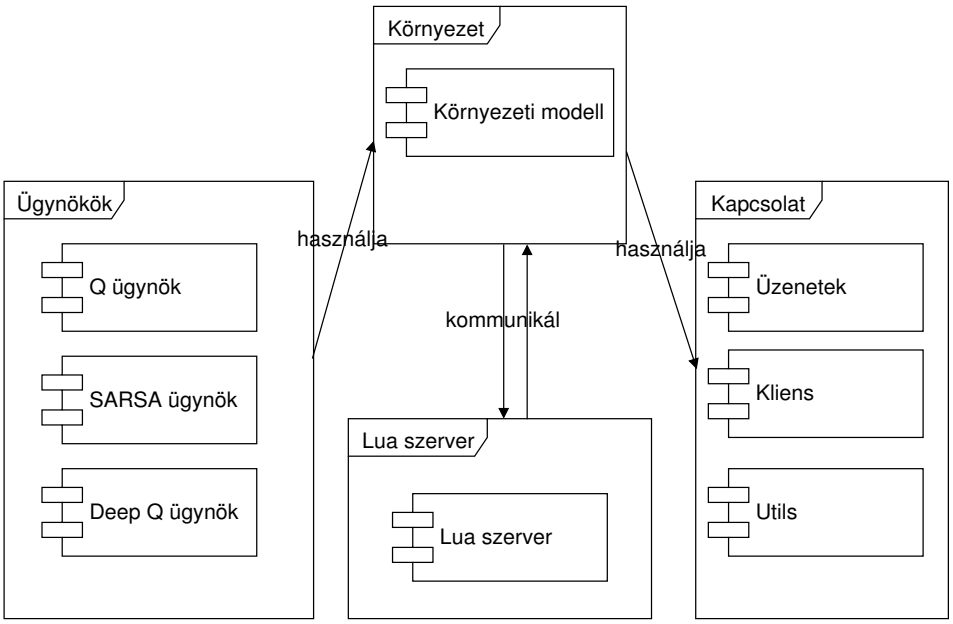
A Lua programozási nyelv a leggyorsabb interpretált szkriptnyelv a világon. Erőssége, hogy kis méretű, (1.1 MBájt forrásfájllal és dokumentációval együtt) minden olyan rendszeren használható amelyik rendelkezik egy standard C kompilátorral és könnyen beágyazható más nyelvben írt programokba. [Lua] A LuaRocks package manager segítségével létrehozhatunk és telepíthetünk Lua modulokat. A mi esetünkben, szükségünk van a *lua-cjson* és *lua-socket* modulokra, amelyek segítségével képesek vagyunk JSON üzeneteket feldolgozni és socketekre kapcsolódni.

Azért volt szükségünk a fentebb leírt modulokra és egyben a Lua környezetre, hogy képesek legyünk kommunikálni a játékkal, amit az emulátoron elindítottunk. Mivel az FCEUX támogatja a Lua szkripteket, keresnünk kellett egy modult, ami úgymond, szerverré alakítja az emulátoron futó játékot. Miután betöltünk egy szkriptet elvégezhetjük a feladatunkat miközben fut a játék, viszont a szkript felelőséggé lesz biztosítani a képkockák folyamatos betöltését és a memóracímek karbantartását. Dolgozatunk célja nem az volt, hogy egy emulátoron futó játékból szervert csináljunk, ezért kerestünk egy modult, ami ezt elvégezte.

Szerencsére, Marcus Edel biztosított egy ilyen csomagot azok számára, akik nem szerettek volna feltétlenül a kutatáson kívűl ezzel a problémával is foglalkozni és ezt nyílvánossá is tette az egyik github repozitoriján. [NES] Csomagja tartalmazza a szkirpteket, amelyek játékunkat szerverré alakítja. Kiolvassa a megfelelő memóriacímekről [RAM] a játék állását, ellenségek pozicióját és az elért pontszámot. Továbbá, megnyit socketen keresztül egy kommunikációs csatornát, ami egy kliens befogadására képes és bizonyos számú képkockánként üzeneteket vár tőle. Ezek az üzenetek be kell tartsák a 3.2 fejezetben leírt specifikációkat, ahhoz hogy a szerver képes legyen ezeket feldolgozni és válaszolni rájuk.

3.4. Python kliens

A Python kliens a lelke az alkalmazásunknak, mivel itt található az ügynökünk "agya". Ugyancsak itt vannak implementálva a visszacsatolásos tanulás különböző algoritmusai és ami a legfontosabb, hogy ugyancsak itt található a környezetünk ábrázolása is. Így, a kliensünket felbonthatjuk egy kommunikációs, egy visszacsatolásos tanulásos és egy környezet ábrázolásos részre. Ez megtekinthető a 3.3-as ábrán is.



3.3. ábra. Az ábrán megtekinthetjük az alkalmazásunk komponensdiagramját. A környezeti modellünk a Kapcsolat csomag osztályai segítségével képes kommunikálni a Lua szerverrel. Az ügynökeink pedig a környezeti modell felhasználásával képesek információt szerezni a játék állásáról és döntései minőségéről.

3.4.1. A Lua szerver és az RF ügynök közötti kommunikációt megvalósító komponens

A kommunikációs modul tartalmazza azokat az osztályokat és metódusokat, amelyek segítségével kapcsolatot tudunk teremteni a Lua szerverrel.

A *Client* osztály implementálja a socket kommunikációt a Python socket csomag segítségével. Addot host és port szám megléte mellett, képes rákapcsolódni az adott socketre. Ha sikeresen megtörtént a kapcsolatteremtés, akkor tudunk unikód üzeneteket küldeni és fogadni.

A *Message* osztály példánya foglalja magába az üzenetünket, a szervernek szeretnénk küldeni. A 3.1 részben leírt üzeneteket képes előállítani és egy végleges üzenetbe összefűzni ezeket. A legfontosabb metódusok ebben az osztályban a *clear*, amely törli az aktuális üzenetünket és a `finalize_message`, amely kiegészíti az elküldeni kívánt üzenetet a megfelelő zárójelekkel, annak érdekében, hogy megmaradjon a helyes JSON formátum.

A *Utils* osztály tartalmazza azokat a metódusokat amelyek értelmezik a fentebb említett üzeneteket. A fontosabb függvények ezek közül a `tiles_matrix_from_json`, amely felépíti a játék állását reprezentáló kétdimenziós tömböt, a `position_from_json`, amely megadja az ügynökünk aktuális pozicióját és a `player_state_from_json`, amely visszatéríti, hogy az ügynökünk végállapotba került-e vagy sem.

3.4.2. A környezet ábrázolását szolgáló modul

Ez a rész alkotja a környezeti modellünket, amely a 3.2-es alfejezetben leírt modul segítségével kommunikál az emulátorral és az attól kapott információkat feldolgozva továbbítja az ügynökünknek egy számára érthető formában. Két fontos dolgot valósít meg a környezeti modell: meghatározza a jutalom mátrixot és adott állapottérre (3.2-es ábrán látható két dimenziós tömb) meghatározza, hogy milyen állapotban vagyunk.

Egy *állapot* meghatározza, hogy az ügynök adott időpillanatban milyen körülmények között található. A mi esetünkben az ügynök egy 13*13-as dimenziójú tömbként érzékeli a környezetét, ahol minden tömbbeli elem négy értéket vehet fel (lásd 3.2-es fejezet) ezért összesen 676 lehetséges állapotunk van. Mivel ez túl sok állapot ahhoz, hogy kézileg meg tudjuk adni minden állapot-cselekedet átmenetre a megfelelő jutalom értéket és ezeknek az összehangolása szinte lehetetlen feladat, ezért szükségünk volt általánosítani az állapotterünket. Olyan helyzetekben, ahol úgy éreztük, hogy az ügynökünk gyengén teljesít, megpróbáltunk új állapotokat megfogalmazni. Így, jelenleg 13 álapottal rendelkezik a környezeti modellünk. Ez a szám természetesen növelhető, annak érdekében, hogy jobb eredményeket érjünk el, viszont egy-egy új állapot behozatala a modellbe nagyon sok időbe és próbálkozásba telik.
A jelenleg definiált állapotok:

- nincs ellenség és blokk a közelünkben;
- blokk fölöttünk és nincs ellenség a közelben;
- ellenség az előttünk lévő mezőn;
- kis akadály előttünk;
- nagy akadály előttünk;
- blokk fölöttünk és ellenség több mint 2 blokk távolságban;
- levegőben vagyunk;
- nincs föld előttünk;
- beragadtunk nagy akadály előtt;
- beragadtunk kis akadály előtt;
- ellenség a hátunk mögötti mezőn;
- ellenség előttünk miközben levegőben vagyunk;
- beragadtunk akadály előtt miközben levegőben vagyunk.

A *jutalommátrix* kulcsfontosságú az ügynök tanításában. Rosszul konfigurált jutalommátrixszal az ügynökünk nem lenne képes megtanulni az optimális irányelvet. Ez a tömb meghatározza, hogy az ügynök egy adott állapotban bármilyen cselekedetre mennyi jutalmat kap. Ez a jutalom a [-1,1] intervallumból veszi fel az értékeit. Mivel az ügynöknek minden állapotban, minden cselekedetre kell ismernie a lehetséges jutalmakat, ezért egy 13*5-ös mátrixot kellett meghatároznunk. Bár az ügynökünk 7 cselekedetet képes elvégezni (3.2.1), mi mégis figyelmen kívűl hagytuk a Le, Fel, Start, illetve B akciókat és definiáltuk az átlós ugrásokat (bal- és jobb ugrás). Ezt azért tettük meg, mivel a célunk az volt, hogy minél gyorsabban és optimálisabban teljesítse a pályát, tehát nincs szüksége felvenni semmi segítséget. Segítség alatt gombákat, extra életet, pénzérméket és virágokat értünk. Ezeknek az a célja, hogy könnyítse a játékot a játékosok számára. (például minden 100. felvett érme után újabb életet kap a játékos) Így, ezeknek az akcióknak nincs jelentőségük amíg az ügynök normál formában van.

3.4.3. A visszacsatolásos tanulási modul

Ez a rész tartalmazza a különböző ügynökeink (Q, SARSA és Deep Q) megvalósításait, amelyek elméleti hátterét és működését a 2.4.3-as fejezetben mutattuk be.

Mindhárom ügynökünk másképp közelíti meg az optimális irányelv felfedezését, viszont vannak hasonlóságok is közöttük. Például mindhárom ugyanazzal a konfigurációs paraméterrel fut, de erről egy későbbi fejezetben fogunk részletesebben beszélni. Továbbá mindegyik ügynök az ϵ mohó stratégiával (2.2) határozza meg következő lépését.

A Q és SARSA ügynökkel ellentétben, ahol a Q függvényt egy tömbbel reprezentáljuk, a Deep Q ügynöknél ugyanezt egy *neurális hálóval* (2.5.3) próbáljuk elérni. A háló ábrázolásához egy Pythonban írt API-t, a *Keras*-t [Ker] használtuk, amely segítségével könnyen tudunk neurális hálókat modellezni. Különböző modulokat használhatunk amin a Keras futhat, mint például a CNTK [CNT], Theano [The] és TensorFlow [Ten]. Mi az utóbbit használtuk, mivel alapból ezt használja a Keras és minden tulajdonsággal rendelkezett, amire szükségünk volt. Egy szekvenciális modellt használtunk a háló reprezentálásához, ami annyit jelent, hogy a különböző rétegek egymás után következnek. Három réteget különítettünk el. Az első, ami az állapotokat fogja tőlünk megkapni lesz egyben a bemeneti rétegünk. Mivel tudjuk, hogy 13 állapotunk van összesen, ezért a bemenetünk dimenziója egy 13x1-es tömb lesz, ahol "1"-essel fogjuk jelölni, hogy pillanatanyilag milyen állapotban vagyunk. Ennek az eredményét továbbadjuk a rejtett rétegeknek. Ezek számával és milyenségével kapcsolatosan több fajta konfigurációval próbálkoztunk, aminek az eredményeit a 4.1-es fejezetben mutatjuk be részletesebben. Az utolsó rétegünk fogja visszaadni, hogy milyen cselekedetet érdemes adott körülményekben végrehatjatni.

Modellunk tanítására az ügynökünk eddigi tapasztalait használtuk fel. Egy listába eltároltuk mindig az utolsó 10000 (állapot, cselekedet, jutalom, következő állapot, végállapot-e) ötöst. Minden epizód végén választunk véletlenszerűen, rögzített számú minta ötöst az előbb említett listából. Majd a mintákból vett állapotokra megvizsgáljuk, hogy milyen cselekedeteket jósolt a hálónk. Végül súlyozzuk ezen állapotok jutalmait a Q függvény alapján 2.5.1 és frissítjük az élek súlyait a hálónkban az állapotok és jutalmak segítségével.

4. fejezet

Eredmények bemutatása és értékelése

4.1. Az ügynökök beállításai

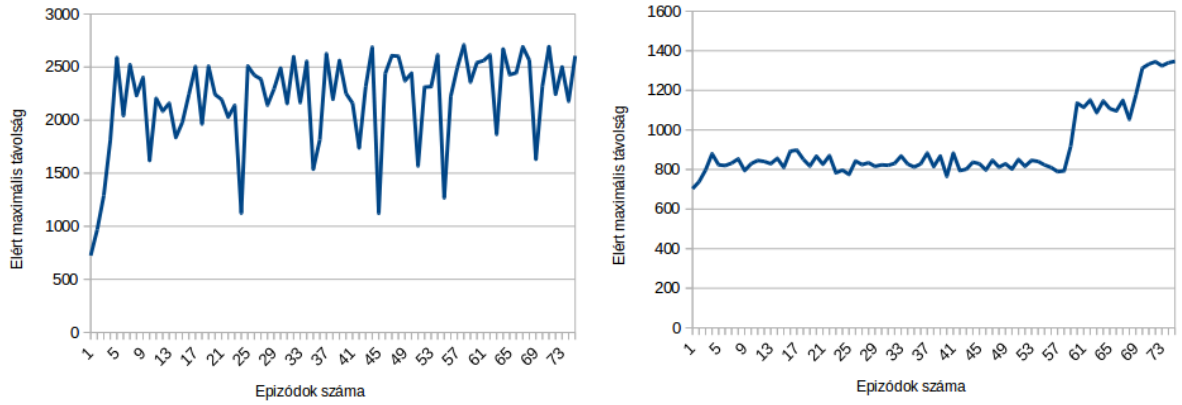
Az ügynökeinket, megfelelő erőforrás hiányában, 15 ezer epizód erejéig futtattuk, ami körülbelül 15 órának felelt meg. Ebből az első 14 ezerben az ügynökeink tanultak, míg a maradék ezerben teszteltük az eredményeket. Ahhoz, hogy ügynökeink bármiféle eredményeket tudjanak elérni, szükséges volt ezek paramétereinek megfelelő beállítására. Ahogy a 2.4.3-as fejezetben is láttuk, a Q, Deep Q és SARSA ügynöknek szüksége van egy tanulási rátára (α) és egy lecsengési faktorra (γ).

A tanulási ráta mondja meg, hogy az újonnan szerzett tapasztalat mennyire fontos az ügynök számára. Vagyis meghatározza annak a "lépésnek" a méretét, amivel az optimális stratégia felé haladunk. Ha az ügynök túl nagyokat "lép" a tanulásban, akkor megtörténhet, hogy túllépi és sohasem találná meg az optimális lépéssorozatot. Tehát divergálna.

Kísérleteink kezdete során a tanulási rátát α = 0.1 -re állítottuk, mivel úgy véltük, hogy ez elég kicsi lesz. Viszont ilyen rátával a SARSA ügynökünknek csupán 2.19 százalékos sikere volt, vagyis tanítás után 1000 játékból csak 22-őt sikerült neki végigvinnie. Miután az α értékét lecsökkentettük 10^{-6}-ra, ügynökünk sikerességi rátája megnőtt 50.7 százalékra.

Ugyanez a jelenség megfigyelhető volt a Deep Q ügynöknél is, viszont a standard Q ügynök pont fordítottan reagált. Az α = 0.1 paraméterrel sokkal gyorsabban megtanulta az optimális lépéssorozatot és jobban teljesített (56.8%-os sikerességi ráta) a 10^{-6}-os beállításhoz képest, (0%-os sikerességi ráta) ahol az ügynök fejlődése ugyan látszott, viszont még sok ezer epizódra lett volna szüksége, hogy eredményeket tudjon felmutatni. Ezenkívül megfigyelhetjük még a 4.1-es ábrán, hogy bár nagyobb α értékre az ügynök sokkal hamarabb végig tudja vinni a játékot, viszont, ahogy fentebb említettük, "túllépi" az optimalitást és többször hibáz.

A lecsengési faktor megmondja, hogy az ügynökünk mennyire tervezi hosszú távon maximalizálni a jutalmait, ezért nem volt célszerű ezzel a paraméterrel kísérleteznünk. Mivel a célunk az volt, hogy az ügynök a lehető legjobban teljesítsen, ezért szükség volt rá, hogy a faktort egy 1-hez közeli értékre állítsuk, mivel így megpróbálja majd a jutalmait miaximalizálni. Az előbbi meggondolás alapján a γ-t 0.95-re állítottuk.



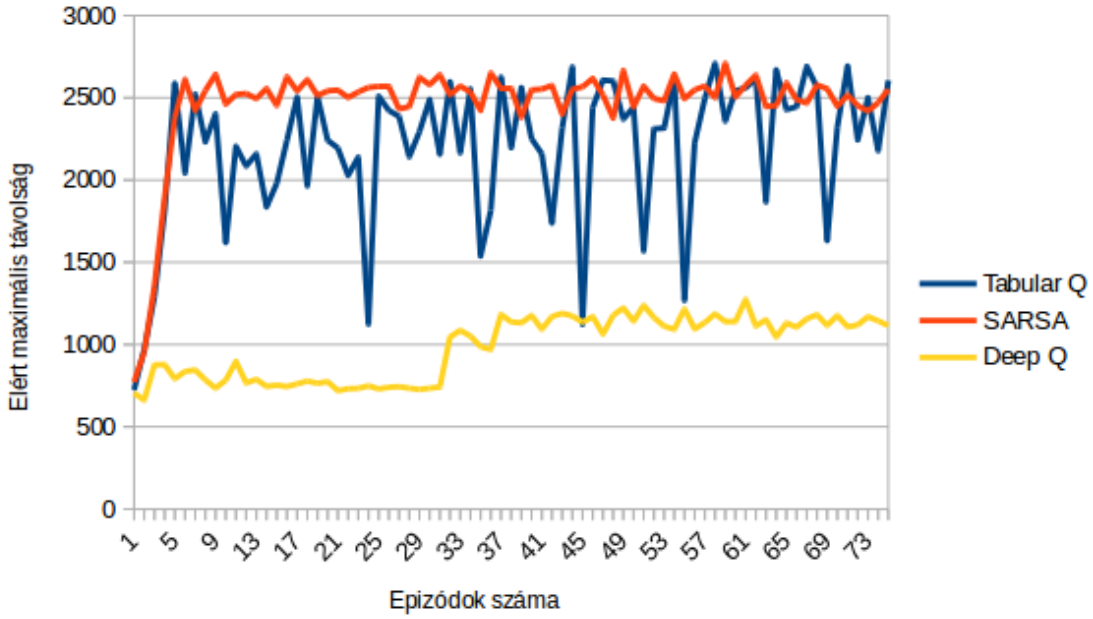
4.1. ábra. A fenti képeken a standard Q ügynök teljesítményei látszanak különböző tanulási ráta értékekre. Ezek a teljesítmények 15 ezer epizód eredményei, ahol az ügynök által elért maximális távolságot rajzoltuk ki. Ahhoz, hogy láthatóbb legyen, 200-anként átlagolva vannak az értékek. A bal oldali képen az $\alpha = 0.1$, míg a jobb oldalt az $\alpha = 10^{-6}$-os konfiguráció látszik.

A Deep Q ügynök esetén, a fenti két paraméteren kívűl még meg kellett határozzuk, hogy a neurális hálójában hány rejtett rétege legyen, milyen nagy legyen az emlékezete és hogy egyszerre mennyi adattal legyünk képesek tanítani a rendszert.
A hálónk összes rétege teljesen összekapcsoltak, amelyek 128 neuronnal rendelkeznek. A bemeneti és kimeneti rétegen kívűl egy rejtett rétegünk van. Próbálkoztunk még 2 rejtett réteggel is, viszont, így megduplázódtak a neuronjaink száma, amiket be kell állítson magának. Az utóbbi konfiguráció mellett 15 ezer epizód elteltével sem volt érzékelhető az ügynök tanulása. Az emlékezet nagyságát kezdetben 5000 egységre hagytuk, viszont úgy vettük észre, hogy nem igazán hajt végre új cselekedeteket bizonyos idő elteltével tanulás közben. Mivel a memóriánk megengedte, megnöveltük ennek a méretét 15000 egységre. A hálónk azon tulajdonságát, hogy egyszerre mennyi adattal tudjuk tanítani, másnéven, hogy mekkora legyen a *batch* mérete. Kísérleteink kezdete során a batch méretét 32-re állítottuk, viszont így nem láttunk fejlődést az ügynök fejlődésében. Ezért addig dupláztuk ezt a számot, amíg nem láttunk fejlődést. Ezt az elgondolást követve jutottunk el az 1024-es batch mérethez.

4.2. Az ügynökök rangsorolása

Célunk az volt, hogy egy olyan programot alkossunk, amelyik a lehető legtöbbször viszi át a játékunkat. Ezért, ügynökeinket, ezen aspektus alapján rangosoroltuk. (4.2-es ábra) Ahogy a 4.1-es fejezet elején is említettük, ügynökeinket 15 ezer hosszúságú epizódokra futtattuk, viszont a lentebb említett statisztikákat csak az utolsó 1000 epizód teljesítményeiből származtattuk.

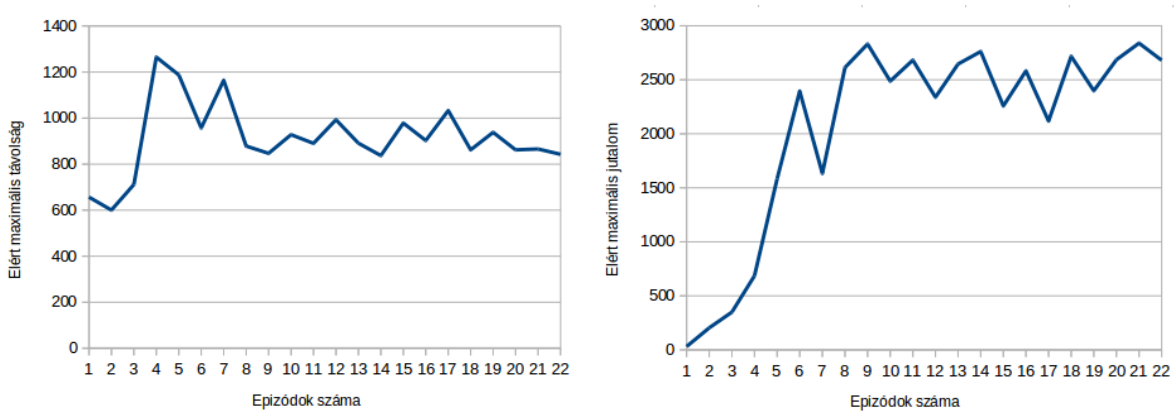
A legjobb eredményt a standard Q ügynökkel értük el, amely 56.8 % -os sikerességi rátával rendelkezett. Ezt azután értük el, hogy a környezeti modellünkben módosítottuk a jutalomfüggvény értékeit.



4.2. ábra. A fenti képeken a standard Q, SARSA és Deep Q ügynök teljesítményei (epizódonként elért maximális távolságai) látszanak. Ezek a teljesítmények 15 ezer epizód eredményei, ahol az ügynök által elért maximális távolságokat rajzoltuk ki. Ahhoz, hogy a változási tendecia világosabb legyen, 200-anként átlagolva vannak az értékek. Megfigyelhető, hogy a legjobban a standard Q és SARSA ügynök teljesített, míg a Deep Q ügynök még csak tanulási fázisban van.

Ezután következik a SARSA ügynökünk, aki 50.7 %-ban teljesítette sikeresen az 1000 epizódot. Ő akkor érte el ezt a teljesítményt, amikor a tanulási ráta értékét lecsökkentettük 10^{-6}-ra és kivettük az algoritmusból azt az optimalizációt, amivel az ügynök felfedezési rátáját szerettük volna növelni. Ez annyiban állt, hogy amikor az ügynök kiválasztja a legnagyobb Q értékkel rendelkező cselekdetet és ennek negatív az értéke, akkor e helyett egy véletlenszerű cselekdetet hajt végre, mivel mindegy kéne legyen, hogy egy nem célszerű lépést hajt végre vagy felfedez egy új átmenetet. A leggyengébben a Deep Q ügynök teljesített, aki csupán 3.59 %-ban tudta befejezni a játékot. Ahogy a 4.2 ábrán is látszik az ügynök sokkal lassabban tanul, mint a vetélytársai.

Egy másik rangsorolási elv lehetett volna még az elért maximális jutalom alapján, viszont arra kellett rájöjjünk, hogy a szerzett jutalom nagysága nem mindig arányos az ügynök által maximálisan elért távolsággal. (4.3-as ábra) Miután észrevettük, hogy az ügynök sokkal több jutalmat összegyűjtött, mint amilyen távolra elért, kiszámoltuk a jutalom és távolság statisztikák *Pearson korrelációs együtthatóját*. Ez a szám megmondja, hogy két változó mennyire korrelált, vagyis hogy mennyire vannak összeköttetésben. Azt vettük észre, hogy a standard Q és SARSA ügynökök statisztikái között a korrelációs együttható +/- 0.65 volt, míg a Deep Q ügyknél 0.35. Az utóbbinál azért volt kevesebb, mivel olyan lépéssorozatot tanult meg az ügynök, ahol maximalizálni tudta a jutalmát, anélkül, hogy végig kellett volna vigye a játékot. Ez a környezet modelljének a pontatlanságának róható fel. Ez is kihangsúlyozza, hogy a körenyezeti modell megtervezése a legkritikusabb lépés egy visszacsatolásos ügynök létrehozásában.



4.3. ábra. A fenti képeken a Deep Q ügynök távolság és jutalom teljesítményei látszanak. Ezek a teljesítmények 15 ezer epizód eredményei, ahol az ügynök által elért maximális távolságot és jutalmat rajzoltuk ki. Ahhoz, hogy könnyebben leolvasható legyen a változási tendencia, 200-anként átlagoltuk az értékeket. A bal oldali képen látható, hogy az ügynök nagyon gyengén szerepelt. A pálya távolságának csupán egyharmadát teljesítette, miközben a jutalma, ami a jobb oldalon látható, azt jelöli, hogy nagyon sok jutalmat gyűjtött össze. Ez azt jelenti, hogy a jutalom mátrixban rosszul vannak definiálva az értékek.

5. fejezet

Összefoglaló

5.1. Következtetések

Célunk az volt, hogy visszacsatolásos tanulási módszerekkel egy olyan ügynököt hozzunk létre, amely képes átvinni a Mario játékot. Ahol jelenleg tartunk, azt valamilyen szinten sikernek tekinthetjük, megpróbáltuk modellezni a Mario játékot és eredményekre jutottunk. Bár nem sikerült maximalizálni a Mario ügynökünk sikerességét, viszont számítási képességek hiányában úgy érezzük, hogy kihoztuk a legtöbbet az adott erőforrásokból.

5.2. Továbbfejlesztési lehetőségek

A későbbiekben szeretnénk jobb erőforrásokat szerezni. Továbbá, jó lenne megvalósítani, hogy az ügynökünk ne csak az első pályát legyen képes átvinni, hanem az egész játékot. E cél eléréséhez új állapotokat kell definiálnunk (például vegye fel az érméket, ha van a közelében; próbáljon meg lemenni a csövekben, ha letséges) és finomítanunk kell a környezeti modellünket. A meglévő ügynökeinket ki szeretnénk bővíteni Monte Carlo 2.4.2 módszerekkel, hogy tapasztalatból is tanulhassanak. Szívesen megpróbálkoznánk a dinamikus programozás módszereivel 2.4.1 is, hogy az ügynökünk "szimulálja" a játék kimenetét minden cselekedet végrehajtása előtt és ezáltal eldönthesse, melyik lenne a következő legoptimálisabb művelet számára. Ehhez viszont, ahogy már említettük, sok memóriára és számítási képességre van szükségünk, mivel nagyon sok köztes lépés lehet, míg végállapothoz jutunk. Érdemes lenne az eddig felhasznált módszerek elveit egyesíteni, amely által saját módszert is létrehozhatnánk a feladat jobb eredményeinek elérése érdekében.

Irodalomjegyzék

Microsoft cognitive toolkit home page. https://www.microsoft.com/en-us/cognitive-toolkit/. Accessed: 2017-04-20.

Fceux. http://www.fceux.com/web/home.html. Accessed: 2017-04-15.

Fanuc home page. https://www.fanuc.eu/. Accessed: 2017-04-21.

Keras documentation. https://keras.io/. Accessed: 2017-02-22.

Lua official website. https://www.lua.org/. Accessed: 2017-12-29.

Nes. https://github.com/zoq/nes. Accessed: 2017-12-20.

Super mario bros ram map. https://datacrystal.romhacking.net/wiki/Super_Mario_Bros.:RAM_map. Accessed: 2018-01-09.

Fanuc robots learns a new job overnight. https://www.technologyreview.com/s/601045/this-factory-robot-learns-a-new-job-overnight/. Accessed: 2017-04-21.

Oracle documentation. https://docs.oracle.com. Accessed: 2018-03-26.

Tensorflow home page. https://www.tensorflow.org/. Accessed: 2017-04-20.

Theano home page. https://github.com/Theano. Accessed: 2017-04-20.

Bengio, Y. és others, . Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Boyan, J. A. és Moore, A. W. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, pages 369–376, 1995.

Busoniu, L., Babuska, R., De Schutter, B., és Ernst, D. *Reinforcement learning and dynamic programming using function approximators*, volume 39. CRC press, 2010.

Dayan, P. és Niv, Y. Reinforcement learning: the good, the bad and the ugly. *Current opinion in neurobiology*, 18(2):185–196, 2008.

Domingos, P. A few useful things to know about machine learning. *Communications of the ACM*, 55 (10):78–87, 2012.

Doucet, A., De Freitas, N., és Gordon, N. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

Gershenson, C. Artificial neural networks for beginners. *arXiv preprint cs/0308031*, 2003.

Gu, S., Lillicrap, T., Sutskever, I., és Levine, S. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

Ishii, S., Yoshida, W., és Yoshimoto, J. Control of exploitation–exploration meta-parameter in reinforcement learning. *Neural networks*, 15(4-6):665–687, 2002.

Kaelbling, L. P., Littman, M. L., és Moore, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

Lin, L.-J. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

Papadimitriou, C. H. és Tsitsiklis, J. N. The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987.

Precup, D., Sutton, R. S., és Dasgupta, S. Off-policy temporal-difference learning with function approximation. In *ICML*, pages 417–424, 2001.

Quah, K. H. és Quek, C. Maximum reward reinforcement learning: A non-cumulative reward criterion. *Expert Systems with Applications*, 31(2):351–359, 2006.

Ribeiro, C. Reinforcement learning agents. *Artificial intelligence review*, 17(3):223–250, 2002.

Rummery, G. A. és Niranjan, M. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.

Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., és Edwards, D. D. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

Sutton, R. S. és Barto, A. G. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Sutton, R. S., McAllester, D. A., Singh, S. P., és Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

Otterlo, M.van és Wiering, M. Reinforcement learning and markov decision processes. In *Reinforcement Learning*, pages 3–42. Springer, 2012.

Watkins, C. J. és Dayan, P. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Watkins, C. J. C. H. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

Yogeswaran, M. és Ponnambalam, S. Reinforcement learning: exploration–exploitation dilemma in multi-agent foraging task. *Opsearch*, 49(3):223–236, 2012.