Debreceni Egyetem
Informatikai Kar

A megerősítéses tanulás alkalmazása az Othello játékban

Témavezető:
Dr. Várterész Magda
Egyetemi docens

Készítette:
Andorkó Gábor
Programtervező informatikus (B.Sc.)

Debrecen
2011

Köszönetnyilvánítás

Ezúton szeretném kifejezni köszönetemet mindenkinek, aki a szakdolgozatom elkészítéséhez nagyban hozzájárult, elsősorban Dr. Várterész Magda tanárnőnek és Dr. Fazekas István tanár úrnak, akik felkeltették az érdeklődésemet a mesterséges intelligencia és a neurális hálózatok iránt.

Tartalomjegyzék

1. Bevezetés

Sok döntéshozási probléma, amivel a való életben találkozhatunk, természeténél fogva szekvenciális. Ezekben a problémákban az eredmény nem egyetlen döntéstől függ, hanem döntések sorozatától. A legjobb végeredmény eléréséhoz a döntéshozónak esetleg fel kell áldoznia közvetlen eredményeket. Jó döntési sorozatokat készítő stratégia megtalálása fontos probléma. Elméletileg egy ilyen stratégia minden egyes lehetséges szituációban vagy állapotban azt mutatná, hogy mi a legjobb döntés.

Jól ismert osztálya a szekvenciális döntéshozási problémáknak a Markov Döntési Folyamatok (Markov decision processes, MDP). A legfontosabb tulajdonságuk az, hogy az optimális döntés egy adott állapotban független azoktól az állapotoktól, amikkel a döntéshozó korábban szembesült. Az MDP-ket széles körben alkalmazzák az operációkutatásban és a gazdaságtudományban.

Számos olyan MDP algoritmus létezik, ami garantálja az optimális stratégia megtalálását. Ezek az algoritmusok együttesen dinamikus programozási módszerek néven ismertek. A dinamikus programozási módszerek képtelenek olyan feladatok megoldására, ahol a lehetséges állapotok száma nagy (amit úgy is hívnak, hogy a méretek átka). Egy másik probléma az, hogy a dinamikus programozás a feladat jellemzőinek pontos tudását igényli (amit úgy is hívnak, hogy a mintakészítés átka).

Egy viszonylag új osztálya ezeknek az algoritmusoknak a megerősítéses tanulási algoritmusok. Több tudományos területen értek el ezekkel eredményt. Számos sikeres gyakorlati alkalmazás született a robotikában, az ipari gyártásban és a kombinatorikus keresési problémák területén. Az egyik legmeggyőzőbb alkalmazás a TD-Gammon. Ez egy olyan rendszer, amely megtanul játszani a Backgammon játékkal úgy, hogy önmaga ellen játszik és az eredményekből tanul. A TD-Gammon elérte azt a szintet, hogy majdnem olyan jól játszik, mint a legjobb emberi játékosok.

Az operációkutatás és a gazdaságtudomány területén komoly érdeklődés mutatkozik a megerősítéses tanulási algoritmusok iránt. Például megerősítéses tanulást alkalmaznak légitársaságok teljesítménykezelésére és arra, hogy optimális stratégiát találjanak a helyfoglalások elutasítására vagy elfogadására különböző díjszabású osztályok esetén. Használnak megerősítéses tanulást arra is, hogy optimális irányítási stratégiát találjanak egy csoport liftnek.

Az Othellohoz hasonló játékok jól definiált szekvenciális döntéshozási problémák. Összetett problémamegoldást igényelnek, a teljesítményt könnyen lehet bennük mérni. Bebizonyosodott, hogy a játékok a különböző típusú problémamegoldó technikák tanulmányozásában és fejlesztésében nagy segítséget jelentenek. Ilyen például Moriarty és Miikkulainen munkája [1], akik

genetikus algoritmusokkal tanulmányozták a játszó neurális hálózatok mesterséges fejlődését. Néhány hálózatot arra használtak, hogy megtanulják az Othellot bármilyen szakértői tudás nélkül. Egy másik példa Chong munkája [2], aki fejlődési algoritmusokat használó neurális hálózatokat tanulmányozott, hogy megtanítsa az Othello játékot előre programozott emberi szaktudás nélkül.

Szakdolgozatomhoz felhasználtam Nees Jan van Eck és Michiel van Wezel munkáját [3]. Célom a megerősítéses tanulás leírása. Ennek bemutatására az óriási állapotterű Othello játékot használtam fel. Ehhez kapcsolódóan elvégeztem néhány kísérletet, amikben megerősítéses tanulást alkalmazok. A kísérletekben a megerősítéses tanuló ágensek megtanulják az Othello játékot emberi szakértők által biztosított tudás nélkül. Egy gyakran használt megerősítéses tanulási algoritmust is részletesen leírok, ez a Q-tanulás (Q-learning). Leírom az Othello játékot, az Othellot játszó ágenseket, amiket a kísérletekben használtam, valamint a kísérleteket is ismertetem.

2. Megerősítéses tanulás és szekvenciális döntéshozási problémák

Ebben a fejezetben rövid leírást adok a megerősítéses tanulásról, valamint a szekvenciális döntéshozási problémákról.

Az intelligens ágens szemszögéből írom le a megerősítéses tanulást. Egy intelligens ágens egy önálló entitás (rendszerint egy számítógépes program), ami ismételten érzékeli a bemeneteket a környezetéből, feldolgozza ezeket a bemeneteket és cselekszik a környezetében. Sok tanulási probléma könnyen leírható az ágens szemszögéből anélkül, hogy a problémát lényegesen megváltoztatnánk. A megerősítéses tanulásban az ágens és a környezet beállítása a következő módon történik. Minden egyes pillanatban a környezet egy bizonyos állapotban van. Az ágens figyeli ezt az állapotot és kizárólag az állapottól függően cselekszik. A környezet reagál egy sikeres állapottal és egy megerősítéssel, amit jutalomnak is neveznek. Az ágens feladata az, hogy megtanuljon optimálisan cselekedni úgy, hogy maximalizálja a közvetlen jutalmak és a jövőbeli (diszkontált) jutalmak összegét. Ez maga után vonhatja a közvetlen jutalmak feláldozását annak érdekében, hogy nagyobb összesítő jutalmat kapjunk hosszabb távon, vagy több információhoz jussunk a környezetről.

A megerősítéses tanulás formálisabb leírása a következő módon adható meg: a t-edik időpillanatban az ágens figyeli az $s_t \in S$ állapot környezetét és véghezvisz egy $a_t \in A$ cselekvést, ahol az S és az A a lehetséges állapotok és cselekvések halmazait jelentik. A környezet vissza-

csatolást nyújt egy r_t megerősítés vagy más néven jutalom formájában. Egy állapotátmenet is végbemegy a környezetben, ami az s_{t+1} állapothoz vezet. Így egy cselekvés a környezetet új állapotba viszi, amiben az ágensnek egy új cselekvést kell választania és ez a ciklus ismétlődik tovább. Az ágens feladata, hogy megtanulja a $\pi : S \rightarrow A$ leképezést (amit gyakran vezérelvnek vagy stratégiának neveznek), ami a legjobb cselekvést választja ki minden állapotban. Az összesítő jutalom várt értékét úgy kapjuk meg, hogy követünk egy tetszőleges π vezérelvet egy tetszőleges kezdeti s_t állapotból, amit a következő képlettel fejezhetünk ki:

$$V^{\pi}(s_t) = E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right], \tag{1}$$

ahol az r_{t+i} az a jutalom, amit az s_{t+i} állapotban végrehajtott cselekvésre kapunk a π vezérelvet alkalmazva és a $\gamma \in [0; 1)$ a diszkontálási ráta, ami meghatározza a késleltetett jutalmak viszonylagos értékét a közvetlen jutalmakhoz képest. Ez azért szükséges, mert a jutalmak lehetnek nem determinisztikusak. Az olyan jutalmakat, amiket i idő elteltével kapunk meg, diszkontáljuk γ^i-nel. Ha $\gamma = 0$, csak a közvetlen jutalmakat vesszük figyelembe. Ahogy γ közelít 1-hez, a jövőbeli jutalmakra nagyobb hangsúlyt fektetünk a közvetlen jutalmakhoz képest. Azonban a γ-nak 1 alatt kell lennie, hogy megakadályozzuk a V értékek végtelenné válását. A V^{π} függvényt a π vezérelv állapot-érték függvényének nevezzük.

A $V^{\pi}(s)$ állapot-érték függvényt használva a tanulási feladatot a következőképpen lehet definiálni. Az ágensnek egy optimális vezérelvet kell megtanulnia, azaz a π^*-ot, ami maximalizálja a $V^{\pi}(s)$-t minden s állapotra. Ez a vezérelv a következő:

$$\pi^* = arg\max_{\pi} V^{\pi}(s), \forall s \in S. \tag{2}$$

A jelmagyarázatot leegyszerűsítve a V^{π^*} állapot-érték függvényt a $V^*(s)$ optimális vezérelvvel fejezzük ki. A $V^*(s)$ az optimális érték függvény.

Az a környezet, amiben a megerősítéses tanuló ágens működik, rendszerint MDP (Markov döntési folyamatok). Egy MDP-ben az állapotátmenetek és a jutalmak is kizárólag az aktuális állapottól és az aktuális cselekvéstől függnek, de nem függnek a korábbi állapotoktól és cselekvésektől. Úgy hivatkoznak erre, mint Markov-tulajdonság vagy az út tulajdonság függetlensége. Tehát a jutalom és az állapot úgy határozható meg, hogy $r_t = r(s_t, a_t)$ és $s_{t+1} = \delta(s_t, a_t)$. Az $r(s_t, a_t)$ jutalom függvény és a $\delta(s_t, a_t)$ állapotátmenet függvény lehet nem determinisztikus is.

Megfigyelhető az (1)-es egyenletben az, hogy végtelen sok állapot felett összegeztünk. Ezt végtelen horizont problémának nevezzük. Néhány esetben a végtelen sok állapot feletti összegzést véges sok állapot feletti összegzésre cseréljük. Ebben az esetben egy véges horizont problémáról beszélünk. Ha egy problémának van véges horizontja, a γ diszkontálási ráta 1-re állítható, ami azt jelenti, hogy a közvetlen jutalmak és a jövőbeli jutalmak ugyanolyan súlyúak. Minden

véges horizont problémának létezik jutalom nélküli végállapota. Az ilyen állapot elérése elkerülhetetlen, és ha egyszer egy ágens elért egy ilyen állapotot, akkor további cselekvés (vagy lépés) nem lehetséges, és az ágens az adott állapotban marad anélkül, hogy további jutalmat kapna. A társasjátékokban – beleértve az Othellot is – a végállapot akkor következik be, amikor a játszma befejeződött.

Ha a $\gamma = 1$ és a problémának van jutalom nélküli elkerülhetetlen végállapota, akkor az a tanulási feladat, amellyel az ágens szembenéz, az egy véletlenszerű legrövidebb út problémával egyezik meg. Egy ilyen problémában az ágens a legrövidebb utat keresi a kezdőállapottól a végállapotig egy gráfban, amelyben a csomópontok az állapotokat reprezentálják és az állapotátmenetek véletlenszerűek. A legrövidebb út az az út, ami minimalizálja a várt teljes veszteséget vagy ezzel ekvivalens módon maximalizálja a várt teljes jutalmat. A jutalom függvény egyszerűen átalakítható egy azzal ekvivalens veszteség függvénnyé.

Ha az ágens ismeri a V^* optimális érték függvényt, az állapotátmenet valószínűségeket és a várt jutalmakat, akkor az könnyen meghatározhatja az optimális cselekvést, alkalmazva a maximális várt érték alapelvet, azaz maximalizálja a várt közvetlen jutalom és az utód állapot diszkontált várt értékének összegét.

$$\pi^*(s) = arg\max_{a \in A} E[r(s,a) + \gamma V^*(\delta(s,a))] = \\ = arg\max_{a \in A} \left(E[r(s,a)] + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right) \quad (3)$$

ahol $T(s,a,s')$ az s állapotból az s' állapotba való átmenet valószínűségét jelenti az a cselekvés végrehajtása esetén.

Egy állapot és utód állapotainak értékei a következőképpen kapcsolódnak egymáshoz:

$$V^*(s) = E[r(s,\pi^*(s)) + \gamma V^*(\delta(s,\pi^*(s)))] = \\ = \max_{a \in A} \left(E[r(s,a)] + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right) \quad (4)$$

A (4)-es egyenletek a Bellman egyenletek [4]. Ezeket megoldva (minden állapotra egyet) egyedi értéket ad minden állapotra. Az egyenleteket nehéz megoldani, mert nemlineárisak a $\max$ operátor miatt. Az ilyen egyenletek megoldására a szokásos módok a dinamikus programozási technikák, úgymint értékiteráció és eljárásmód-iteráció.

A megerősítéses tanulás és a dinamikus programozás közeli kapcsolatban vannak, mivel mindkét megközelítést MDP megoldására használjuk. A megerősítéses tanulás és a dinamikus programozás közös ötlete az, hogy érték függvényeket tanulnak, amik így arra használhatók, hogy felismerjék az optimális eljárásmódot. Ennek a közeli kapcsolatnak az ellenére van egy

fontos különbség közöttük. A dinamikus programozás a jutalom függvény és az állapotátmenet függvény teljes tudását igényli. Emiatt a dinamikus programozásról azt mondják, hogy meg van fertőzve a mintakészítés átkával. Ez a szempont különbözteti meg a dinamikus programozást a megerősítéses tanulástól. A megerősítő tanulásban egy ágens nem szükségszerűen ismeri a jutalom függvényt és az állapotátmenet függvényt. A jutalmat és az új állapotot, ami a cselekvés eredménye, a környezet határozza meg, és egy cselekvés következményeit a környezettel kölcsönhatásban kell megfigyelni. Más szóval a megerősítéses tanuló ágensek nem szenvednek a mintakészítés átkától, mert nem szükséges ismerniük a környezetük egy modelljét.

Az eljárásmód-iteráció és az értékiteráció két népszerű dinamikus programozási algoritmus. Mindkét módszer használható arra, hogy megbízhatóan kiszámoljuk az optimális eljárásmódokat és érték függvényeket a véges MDP-kre az MDP teljes tudását megadva.

2.1. Q-tanulás

A Q-tanulás [5, 6] egy megerősítéses tanulási algoritmus, ami megtanulja egy $Q(s, a)$ függvény értékeit, hogy megtalálja az optimális eljárásmódot. A $Q(s, a)$ függvény értékei mutatják, hogy mennyire jó, hogy elvégezzen egy bizonyos cselekvést egy bizonyos állapotban. A $Q(s, a)$ függvény, amit Q-függvénynek is neveznek, úgy van definiálva, hogy az a cselekvést az s állapotban elvégezve a közvetlen jutalmat kapjuk, amihez a jövőben kapható jutalmak diszkontált értékét vesszük, ami azután egy optimális eljárásmódot követ:

$$Q(s, a) = E[r(s, a) + \gamma V^*(\delta(s, a))]. \tag{5}$$

Ha a Q-függvény ismert, egy π^* optimális eljárásmódot ad meg a következő képlet:

$$\pi^*(s) = arg \max_{a \in A} Q(s, a). \tag{6}$$

Ez azt mutatja, hogy ha egy ágens ismeri a Q-függvényt, nem kell ismernie az $r(s, a)$ jutalom függvényt és a $\delta(s, a)$ állapotátmenet függvényt ahhoz, hogy meghatározza a π^* optimális eljárásmódot.

A $V^*(s)$ és a $Q(s, a)$ a következőképpen állnak kapcsolatban:

$$V^*(s) = \max_{a \in A} Q(s, a). \tag{7}$$

A Q-függvény egy rekurzív definícióját megkapjuk a (7)-es egyenlet (5)-ösbe való behelyettesítésével:

$$Q(s, a) = E\left[r(s, a) + \gamma \max_{a' \in A} Q(\delta(s, a), a')\right]. \tag{8}$$

A Q-tanulási algoritmus a Q-függvény ezen definíciójából származik. Egy ágens, ami a Q-tanulási algoritmust használja, iteratív módon megközelíti a Q-függvényt, hogy megtanulja azt. Az algoritmus minden egyes iterációjában az ágens megfigyeli az aktuális s állapotot, kiválaszt egy bizonyos a cselekvést, végrehajtja ezt az a cselekvést és megfigyeli az eredményül kapott $r = r(s, a)$ jutalmat és az $s' = \delta(s, a)$ új állapotot. Ez azután frissíti a Q-függvény becslését, amit $\hat{Q}$-vel jelölök, a frissítő szabály szerint

$$\hat{Q}(s, a) \leftarrow (1 - \alpha)\hat{Q}(s, a) + \alpha \left(r + \gamma \max_{a' \in A} \hat{Q}(s', a') \right), \tag{9}$$

ahol az $\alpha \in [0, 1)$ a tanulási ráta paraméter. Watkins és Dayan [6] bebizonyította, hogy az ágens által becsült Q-értékek 1 valószínűséggel konvergálnak a valódi Q-értékekhez feltéve, hogy a környezet egy MDP korlátos jutalmakkal, a várt Q-értékek egy kereső táblában vannak tárolva és kezdőértékeik tetszőleges véges értékek, a $\gamma \in [0, 1)$, az $\alpha \in [0, 1)$, és az α-t 0-ig csökkentjük.

2.2. Q-függvény tanulása neurális hálózatokkal

A Q-tanulás segít elkerülni a mintakészítés átkát. Azonban egy másik probléma még mindig megmarad. A becsült Q-értékeket tárolni kell valahol a becslési folyamat alatt és után. A legegyszerűbb tárolási forma egy kereső tábla független bejegyzéssel minden állapot-cselekvés párra. Az a probléma a nagy állapot-cselekvés térrel, hogy lassú tanuláshoz vezet és nagy táblákat igényel olyan Q-értékekkel, amiket nem lehet számítógépes memóriában tárolni. Ezt a méretek átkának nevezzük. Ebben az esetben az Othello játék körülbelül 10^{29} bejegyzést igényel, ami világosan meghaladja még a legerősebb számítógép memória kapacitását is. Az Othello állapotterének mérete körülbelül 10^{28} és egy állapotban a szabályos lépések átlagos száma 10, ezért az állapot-cselekvés tér mérete körülbelül $10^{28} \times 10 = 10^{29}$.

A nagy állapot-cselekvés terek problémáját függvényközelítő módszer használatával közelíthetjük meg, azaz egy neurális hálózatot vagy egy regressziós technikát alkalmazunk. A függvényközelítő módszer szerint a Q-értékeket az állapotok és cselekvések függvényeként tároljuk, nem pedig minden állapot-cselekvés párra külön. Ez a tárolóhelyigény nagymértékű csökkenéséhez vezet, így lehetővé válik a Q-értékek tárolása nagy állapot-cselekvés terek esetén is. A szakdolgozatomban ismertetett kísérletekben előrecsatolt neurális hálózatokat használok egy rejtett réteggel, ami a függvényközelítést végzi. Ezért ezt a technikát sokkal részletesebben írom le.

A neurális hálózat egy olyan technika, ami képes reprezentálni komplex bemeneti és kimeneti kapcsolatokat. Az egyik leggyakoribb neurális hálózat modell az előrecsatolt neurális

hálózat (többrétegű perceptronnak is nevezik) egy rejtett réteggel. A neurális hálózatok ezen típusa sok csomópontból áll, amelyek három rétegben – egy bemeneti réteg, egy rejtett réteg és egy kimeneti réteg – vannak elrendezve. Egy adott réteg minden egyes csomópontja össze van kötve egy-egy súlyozott kapcsolattal a következő réteg összes csomópontjával. A bementi réteg az, ahol a bemeneti adatokat beadjuk a hálózatnak. A bemeneti réteg a rejtett rétegbe áramlik, a rejtett réteg pedig a kimeneti rétegbe.

A valódi feldolgozás egy előrecsatolt neurális hálózatban a rejtett rétegben és a kimeneti rétegben lévő csomópontokban történik. A csomópontok kimeneteit ezekben a rétegekben úgy kapjuk, hogy először kiszámítják a bemenetek súlyozott összegét és utána ezt az összeget átalakítják egy aktivációs függvény segítségével. A j-edik rejtett csomópont kimenetét a következőképpen kapjuk meg:

$$h_j = f\left(\sum_{i=0}^{n} w_{ji}^{(1)} x_i\right), \tag{10}$$

ahol n a bemeneti csomópontok száma, $w_{ji}^{(1)}$ az x_i bemeneti csomópont és a h_j rejtett csomópont közötti kapcsolat súlyát, és $f(.)$ az aktivációs függvényt jelenti. Egy w_{j0} eltolássúly is szerepel a képletben, ami egy külön bemeneti csomóponthoz kapcsolódik egy rögzített $x_0 = 1$ aktivációval. Következésképpen a k-adik kimeneti csomópont kimenetét adja meg a következő képlet:

$$y_k = \tilde{f}\left(\sum_{j=0}^{l} w_{kj}^{(2)} h_j\right), \tag{11}$$

ahol l a rejtett csomópontok száma, és $w_{kj}^{(2)}$ a h_j rejtett csomópont és az y_k kimeneti csomópont közötti kapcsolat súlyát jelenti. Ismét szerepel egy w_{k0} eltolássúly, ami egy külön rejtett csomóponthoz kapcsolódik rögzített $h_0 = 1$ aktivációval. Az $\tilde{f}(.)$ jelölést használom a kimeneti csomópontok aktivációs függvényére, így látható, hogy nem kell ugyanazt a függvényt használni, mint a rejtett csomópontoknál. A (10)-es egyenletet a (11)-es egyenletbe behelyettesítve megkapunk egy explicit kifejezést a teljes függvényre, amit a neurális hálózat reprezentál, ez a következő:

$$y_k = \tilde{f}\left(\sum_{j=1}^{l} w_{kj}^{(2)} f\left(\sum_{i=1}^{n} w_{ji}^{(1)} x_i\right)\right). \tag{12}$$

Sok különböző aktivációs függvény van. A leggyakrabban használt aktivációs függvények a lineáris, a szigmoid és tangens hiperbolikusz ($\tanh$-nak is nevezik) függvény. A kísérleteimben a tangens hiperbolikusz aktivációs függvényt használom a rejtett csomópontok és kimeneti csomópontok esetén is. A tangens hiperbolikusz aktivációs függvényt a következőképpen lehet megadni:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{13}$$

Az előrecsatolt neurális hálózatok súlyainak értékeit rendszerint úgy határozzák meg, hogy egy visszaterjesztésnek (backpropagation) nevezett algoritmust használnak. A visszaterjesztéses algoritmust használva a bemeneti példákat egyesével adagolják a neurális hálózatba. Minden bemeneti példa esetén a neurális hálózat kimeneti értékeit a (12)-es egyenlet alapján számítják. A kimeneti értékek és a kívánt kimeneti értékek közötti különbség hibaként jelentkezik. Ezt a hibát azután visszaterjesztjük a neurális hálózaton keresztül és a súlyok igazítására használjuk. A súlyokat a hiba gradiensének negatív irányába mozgatjuk. A súlyok igazításának nagysága függ az úgynevezett tanulási ráta (learning rate) tényezőtől. A visszaterjesztéses algoritmus minden egyes iterációja után a neurális hálózat jobb közelítést biztosít a kívánt kimeneti értékekre.

Amikor az előrecsatolt neurális hálózatot egy Q-függvény közelítésére használjuk, a neurális háló megtanul egy leképezést az állapot-cselekvés leírásból a Q-értékekre. Ezt a leképezést a visszaterjesztéses algoritmust használva tanulja. A visszaterjesztéses algoritmus optimalizálja a neurális hálózat súlyait olyan módon, hogy a (9)-es egyenlet alapján számolt „cél" Q-érték és a neurális hálózat által számolt „kimeneti" Q-érték között lévő hibát csökkenti. Az alábbi algoritmus a teljes Q-tanulási algoritmus, amiben egy neurális hálózatot a függvény közelítésére használunk:

A neurális hálózat összes súlyának véletlen kis kezdőértékeket állítunk be
Repeat(minden egyes kísérletre)
  Az aktuális s állapot meghatározása
  Repeat(a kísérlet minden egyes lépésére)
    Az aktuális s állapot megfigyelése
    Az összes a' cselekvésre s-ben a neurális hálózat kiszámítja a $\hat{Q}(s, a')$ értékét
    Egy π vezérelvet használva kiválasztunk egy a cselekvést
    $Q^{output} \leftarrow \hat{Q}(s, a')$
    Az a cselekvést végrehajtjuk
    Egy r közvetlen jutalmat kapunk
    Az s' új állapot megfigyelése
    Az összes a' cselekvésre s'-ben a neurális hálózat kiszámítja a $\hat{Q}(s', a')$ értékét
    A (9)-es egyenlet szerint kiszámítjuk a $Q^{target} \leftarrow \hat{Q}(s, a)$ értékét
    A neurális hálózatot a $Q^{target} - Q^{output}$ hiba visszaterjesztésével módosítjuk
    $s \leftarrow s'$
  Until s végállapot

Fontos megjegyezni, hogy a nagy állapot-cselekvés térrel rendelkező problémákban a neurális hálózat súlyai az állapot-cselekvés térnek csak egy apró része alapján vannak meghatározva. Neurális hálózat használatával általánosíthatunk állapotokra és cselekvésekre. A ko-

rábban meglátogatott állapot-cselekvés párok tapasztalata alapján a neurális hálózat képes tetszőleges állapot-cselekvés párok Q-értékére becslést adni. A neurális hálózat úgy éri ezt el, hogy megtanulja azokat a jellegzetességeket, amik hasznosak az állapot-cselekvés párok Q-értékeinek becslésében. Az Othello játékban egy ilyen jellegzetesség lehet például a következő szabály: amikor az ellenfél sarok pozíciókat birtokol, az mindig rossz az érték függvénynek. Az ilyen saját maga által felfedezett jellegzetességek alapján a neurális hálózat kimeneti rétege biztosítja a Q-értékek becslését. A neurális hálózat feladatának megkönnyítésére további jellegzetességek biztosíthatók a bemeneti rétegnek az alap állapot-cselekvés leírások mellett. Ez valószínűleg jobb eljárásmód tanulását teszi lehetővé.

Előrecsatolt neurális hálózattal a Q-függvény tanulása a következőképpen történhet:

- külön hálózatot alkalmazunk minden egyes cselekvésre.
- egy egyszerű hálózatot alkalmazunk külön kimeneti csomóponttal minden egyes cselekvésre.
- egy egyszerű hálózatot alkalmazunk, az állapotot és a cselekvést bemenetként, a Q-értéket kimenetként használjuk.

Az első kettő megközelítést használom a kísérletekben, ezért csak azokat írom le részletesebben.

Egy olyan egyszerű előrecsatolt neurális hálózatnak, ami külön kimeneti csomópontokkal rendelkezik minden egyes cselekvésre, a bemenete egy vagy több csomópontból áll, hogy egy állapotot reprezentáljon. A hálózat kimenete annyi csomópontból áll, amennyi cselekvés közül lehet választani. Egyszerű neurális hálózat használatakor az állapotokra és a cselekvésekre is lehet általánosítani.

Ha minden egyes cselekvésre külön előrecsatolt neurális hálózatot használunk, hogy egy állapotot reprezentáljon, akkor minden hálózat bemenete egy vagy több csomópontból áll. Minden hálózatnak csak egy kimeneti csomópontja van, ami azt a Q-értéket adja, ami kapcsolatban van a hálózatnak bemenetként adott állapottal és a neurális hálózat által reprezentált cselekvéssel. Egy bizonyos időpillanatban minden ilyen neurális hálózat megegyező állapotleírást kap bemenetként. Azonban minden hálózat egy Q-értékre van kódolva és minden egyes cselekvésnek megvan a saját neurális hálózata. Ebben az esetben csak az állapotok általánosítása lehetséges.

A Q-tanulást használó neurális hálózatok a Q-értékek tárolásával a Q-tanulásnál nagyobb problémákat is meg tudnak oldani egy kereső táblát használva, de a konvergencia nem garantált. Az a probléma, hogy ezek a hálózatok a Q-függvény nem lokális változtatásait végzik. Egy neurális hálózat egy súlyának megváltoztatásával egyszerre módosíthatja számos állapot

Q-értékét, mivel a Q-értékek az állapotok egy függvényeként vannak reprezentálva a hálózat súlyaival paraméterezve. Azonban a Q-tanulási konvergencia bizonyítása a Q-függvény lokális frissítésén alapul. Egy bizonyos állapot-cselekvés pár értékének frissítésekor a hálózat megsemmisítheti más állapot-cselekvés párok már megtanult értékeit. Ez okozza azt, hogy a neurális hálózat nem biztos, hogy a helyes Q-értékekhez konvergál.

2.3. Cselekvés választása

Az egyik kihívás, ami felmerül az a felderítés és a hasznosítás közötti kompromisszum. Egy ágens hasznosítja aktuális tudását a Q-függvényről, amikor kiválasztja a legnagyobb becsült Q-értékű cselekvést. Ha ehelyett az ágens egy másik cselekvést választ, akkor felderít, mert pontosítja az adott cselekvés Q-értékének becslését.

A cselekvés választására szolgáló algoritmusokban kompromisszumra kell jutni a felderítés és a hasznosítás között, mert egy ágens szeretné hasznosítani azt, amit már tud, hogy magas jutalmat kapjon, de szeretne felderíteni is, hogy jobban tudjon cselekvést kiválasztani a jövőben. Nem lehetséges egyszerre felderíteni és hasznosítani, ennélfogva konfliktus van a felderítés és a hasznosítás között.

Sok módszer létezik a felderítés és a hasznosítás kiegyensúlyozására. A kísérletekben az úgynevezett softmax cselekvés kiválasztó módszert használom. Eszerint az ágensek a cselekvéseket valószínűségek alapján választják, amik a becsült Q-értékeiken alapulnak Boltzmann eloszlás szerint. Adott s állapot esetén az ágens kipróbálja az a cselekvést a következő valószínűséggel:

$$\Pr(a|s) = \frac{exp(\hat{Q}(s,a)/T)}{\sum_{a' \in A} exp(\hat{Q}(s,a')/T)}, \tag{14}$$

ahol T egy hőmérsékletnek nevezett pozitív paraméter, ami a felderítés mértékét irányítja. Nagyon magas hőmérséklet majdnem véletlen cselekvés kiválasztást eredményez. Nagyon alacsony hőmérséklettel a cselekvés kiválasztás mohó lesz, azaz azt a cselekvést választjuk, amelyikre a becsült Q-érték a legnagyobb. A hőmérsékletet rendszerint fokozatosan csökkentjük, ami egy fokozatos átmenethez vezet a felderítésből a hasznosításba.

2.4. Multiágens környezetek

A Q-tanulás biztos, hogy optimális eljárásmódot nyújt egy állandó környezetben (azaz egy MDP-ben). Egy nem állandó környezetben a konvergencia nem biztos.

Egy többágensű beállításban a tanulási folyamat során minden ágensnek az a célja, hogy megtanuljon egy olyan stratégiát, ami adott ellenfelek viselkedése esetén optimális. Ez megfe-

lel egy Nash egyensúly tanulásnak. Azonban az egyes ágensek szemszögéből az ellenfelek a környezet részei. Mivel minden ágens alkalmazkodik az eljárásmódjához, egy egyszerű ágens által megfigyelve a környezet nem állandó és egy Nash egyensúlyhoz való konvergencia nem biztos. Ennek a ténynek az ellenére szabványos Q-tanulást alkalmazok egyszerre két Othellot játszó ágens esetén, amik egymás ellenfelei.

Egy másik megközelítése a többágensű megerősítéses tanulásnak az, hogy megerősítéses tanulási algoritmust használunk, ami ahhoz alkalmazkodott, hogy optimális eljárásmódot találjon a többágenses problémák esetén. Az utóbbi időben Hu és Wellman [7] egy kibővítést ajánlott az eredeti Q-tanulási algoritmushoz, amit Nash Q-tanulásnak neveznek. A Nash Q-tanulásról bebizonyosodott – habár nagyon korlátozó körülmények esetén – többágensű beállításokban, teljes összegű véletlenszerű játékokban, hogy egy Nash egyensúlyhoz konvergáló stratégia.

3. Az Othello játék

3.1. A játék leírása

Az Othello vagy más néven Reversi egy kétszemélyes, determinisztikus, zéró összegű, teljes információjú táblajáték. A játék azért zéró összegű, mert a teljes jutalom rögzített és a játékosok jutalmai egymás ellentettjei. Az Othello azért teljes információjú játék, mert az állapot, azaz a játéktábla teljesen megfigyelhető.

Az Othellot ketten játsszák, általában egy 8×8 mezőből álló négyzetrácsos táblán, amit nem szoktak sakktáblához hasonlóan kiszínezni. A játékhoz 64 kétoldalú korong szükséges, amelyeknek az egyik oldala fekete, a másik pedig fehér. A tábla egyik oldalán a mezőket 1-től 8-ig számozzuk, míg az erre merőleges oldalon A-tól H-ig betűkkel jelöljük. Az egyik játékos a korongokat a fekete felükkel, a másik játékos pedig a fehér felükkel felfelé helyezi el a táblára. Kezdetben a tábla üres a középső négy mezőt kivéve, ahol kettő fekete és kettő fehér korong van, a feketék a főátlóban (a $D5$-ön és az $E4$-en) , a fehérek a másik átlóban (a $D4$-en és az $E5$-ön) .

A játszmát a fekete kezdi, majd ezután felváltva lépnek a játékosok. Szabályos lépésnek számít az, ha a játékos meg tudja fordítani az ellenfél legalább egy korongját, azaz ha a játékos egy korongot egy üres mezőre helyez úgy, hogy a mezőhöz képest legalább egy irányban (vízszintesen, függőlegesen vagy átlósan) folyamatosan az ellenfélnek van(nak) korongja(i), ami(ke)t a játékos egy saját korongja követ. Az ellenfél korongja(i) ebben az irányban átfordul(nak) és a játékos saját korongja(i) lesz(nek).

Egy játszma aktuális állapotán a játéktábla mezőinek állapotát értjük, azaz azt, hogy az

egyes mezőkön nincs semmi, fekete korong vagy fehér korong van. Egy szabályos lépés során a táblán a korongok száma 1-gyel nő. Ez alapján a teljes állapottér mérete körülbelül 10^{28}, és a játszma maximum 60 lépés hosszú lehet, de lehet ettől rövidebb is.

Ha egy játékos nem tud szabályosan lépni, akkor passzolnia kell. Azonban, ha tud szabályosan lépni, akkor passzolni nem lehet. A játszma akkor fejeződik be, amikor már egyik játékos se tud szabályosan lépni. Rendszerint ez akkor következik be, amikor mind a 64 mező ki van töltve, de néhány esetben olyan üres mezők maradhatnak, amiket egyik játékos se tud szabályosan kitölteni. Amikor a játszma befejeződött, a táblán lévő korongokat megszámoljuk. Az a játékos a győztes, amelyiknek több korongja van, mint az ellenfelének. Ha mindkét játékosnak ugyanannyi korongja van, akkor a játszma döntetlenre végződött.

3.2. Stratégiák

Az Othello egy játszmája három részre bontható fel: a nyitó játékra (ami 20-26 lépés után fejeződik be), a középső játékrészre és a végjátékra (ami a vége előtt 16-10 lépéssel kezdődik). A végjáték egyszerűen úgy játszható, hogy a saját korongok számát maximalizáljuk, az ellenfél korongjainak számát pedig minimalizáljuk. A nyitó játéknak és a középső játékrésznek az a célja, hogy a korongokat megtervezett módon helyezzük fel a táblára, így a végjátékban sok olyan koronggá lehet majd átváltani ezeket, amiket az ellenfél nem tud visszafordítani. Ezeket stabil korongoknak nevezzük.

Sok stratégia szerint játszható az Othello, ezek között vannak jók (ilyenek például a positional, a mobility vagy a parity stratégiák) és vannak nagyon rosszak is (ilyen például a maximum disc stratégia, amit a legtöbb kezdő játékos használ). Ezek közül kettő alapstratégiát írok le, a pozicionáló stratégiát (positional strategy) és a mozgékonysági stratégiát (mobility strategy), amelyeket felhasználtam a neurális hálózatok tesztelésére is. A stratégiák kombinálhatók és kiegészíthetők például minimax algoritmussal és alfa-béta vágással is, így növelhetjük a győzelem esélyét.

A pozicionáló stratégia a speciális korongpozíciók fontosságát hangsúlyozza. Az olyan pozíciók, mint a sarkok és az élek értékesek, míg bizonyos más pozíciókat érdemes elkerülni. A sarkok különösen azért értékesek, mert ha egyszer valaki elfoglalta, akkor az ellenfél soha nem tudja átfordítani őket. A játszma elején vagy közepén megszerzett sarki korongokat rendszerint fel lehet használni sokkal több stabil korong megszerzésére. A pozicionáló stratégiát használó játékos megpróbálja maximalizálni a saját értékes korongjainak számát, mialatt az ellenfél értékes korongjainak számát minimalizálja.

A mozgékonysági stratégia azon az ötleten alapul, hogy az ellenfelet olyan lépésre kénysze-

rítjük, amely lehetővé teszi sarki pozíció könnyű megszerzését. A legjobb mód arra, hogy az ellenfelet ilyen rossz lépésre kényszerítsük az, hogy minimalizáljuk az ellenfél mozgékonyságát, azaz az ellenfél szabályosan megtehető lépéseinek számát. Ezt a játékos a saját korongjainak minimalizálásával és azok csoportosításával érheti el.

4. Az Othellot játszó ágensek

Célom a megerősítéses tanuló ágensek betanítása az Othello játékra anélkül, hogy bármilyen emberi szakértő által nyújtott tudást használnék. Ahhoz, hogy ezt a célt elérjem, a kísérletekben kettő megerősítéses tanuló ágens játszik egymás ellen az Othello játékkal. Mindkét ágens a Q-tanulás (Q-learning) algoritmusát használja, hogy megtanulja melyik lépés a legjobb a játszma egy adott állapotában. A tábla aktuális állapotát (a fekete és fehér korongok elhelyezkedése a táblán) a játszma állapotaként használom. A tanulás alatt a Q-tanuló ágenseket periodikusan kiértékelem kettő egymástól különböző típusú viszonyítási alapként szolgáló ágenssel szemben, amelyek a pozicionáló vagy a mozgékonysági stratégiát játsszák.

A kísérletekben három különböző típusú játékost használok - a pozicionáló játékost, a mozgékonysági játékost és a Q-tanuló játékost. A különböző típusú játékosok különböző stratégiákkal játszanak, azaz különböző kiértékelő függvényeket használnak. Minden játékos kiértékelő függvénye egy numerikus értéket határoz meg a végállapotoknak: $+1$-et, -1-et és 0-t, amik ebben a sorrendben a győzelmet, a vereséget és a döntetlent jelentik. A játszma végéig a különböző típusú játékosok különböző kiértékelő függvényt használnak. A következő alfejezetekben részletesen leírom a különböző típusú játékosokat.

4.1. Pozicionáló játékos

A pozicionáló játékos nem tanul. A játszma végéig ez a játékos a pozicionáló stratégia szerint játszik. A játékos célja a nyitó játék és a középső játékrész alatt az, hogy maximalizálja a saját értékes pozícióit (úgymint sarkok és élek), mialatt minimalizálja az ellenfél értékes pozícióit. A pozicionáló játékos ezért a végjátékig a következő kiértékelő függvényt használja:

$$EVAL(s) = w_{A1}v_{A1} + w_{A2}v_{A2} + ... + w_{A8}v_{A8} + ... + w_{H8}v_{H8}, \tag{15}$$

ahol w_i egyenlő $+1$-gyel, ha az i mezőt a játékos saját korongja foglalja el, -1-gyel, ha az ellenfél egy korongja van rajta, és 0-val, ha üres. v_i az i mező értékével egyenlő. A mezők értékeit a következő táblázat mutatja:

	A	B	C	D	E	F	G	H
1	100	-20	10	5	5	10	-20	100
2	-20	-50	-2	-2	-2	-2	-50	-20
3	10	-2	-1	-1	-1	-1	-2	10
4	5	-2	-1	-1	-1	-1	-2	5
5	5	-2	-1	-1	-1	-1	-2	5
6	10	-2	-1	-1	-1	-1	-2	10
7	-20	-50	-2	-2	-2	-2	-50	-20
8	100	-20	10	5	5	10	-20	100

A táblázatban szereplő értékeket a Wipeout - Othello [8] program alapján határoztam meg. A sarki mezők a legjobb pozicionáló lépések a játékban, az úgynevezett X-mezők ($B2$, $B7$, $G2$ és $G7$) pedig a legrosszabbak. Ezért a sarki mezők értékei a legmagasabbak (100 pont) és az X-mezők értékei a legalacsonyabbak (-50 pont). Az X-mezők és az úgynevezett C-mezők ($A2$, $A7$, $B1$, $B8$, $G1$, $G8$, $H2$ és $H7$) esetén a -50 pont és a -20 pont használatának oka az, hogy ha a játékos ezek egyikére helyez el korongot, az megadja a lehetőséget az ellenfélnek arra, hogy sarkot szerezzen meg. A játékos számára így lehetetlenné válik az is, hogy megszerezze a sarkot ebből az irányból a játszma további részében. Az X-mezők és a C-mezők alacsony értékei a játékost arra kényszerítik, hogy amennyire csak tudja, elkerülje a játékot ezeken a mezőkön.

A végjáték akkor kezdődik el, ha a tábla mezőinek legalább 80%-át már elfoglalták, vagy amikor az összes sarki mezőt megszerezték, bármelyik is történjék először. A végjáték alatt a játékos célja, hogy maximalizálja a saját korongjainak a számát mialatt minimalizálja az ellenfél korongjainak a számát. Ahhoz, hogy ezt elérje, a végjáték alatt a pozicionáló játékos a következő kiértékelő függvényt használja:

$$EVAL(s) = n_{player} - n_{opponent}, \tag{16}$$

ahol az n_{player} a játékos által birtokolt mezők száma, az $n_{opponent}$ pedig az ellenfél által birtokolt mezők száma.

4.2. Mozgékonysági játékos

A mozgékonysági játékos nem tanul. A játszma végéig ez a játékos a mozgékonysági stratégia szerint játszik. Ugyanúgy, mint a pozicionáló stratégiában, ebben a stratégiában is a sarki pozícióknak van nagy jelentősége. Továbbá ebben a stratégiában a mozgékonyság fogalma nagyon fontos. A mozgékonyság úgy definiálható, mint a játékos szabályosan megtehető lépéseinek száma egy bizonyos állapotban. A játékos célja a nyitó játékban és a középső játékrészben

az, hogy maximalizálja a saját maga által birtokolt sarki mezők számát és a saját mozgékonyságát, ugyanakkor minimalizálja az ellenfél által birtokolt sarki mezők számát és az ellenfél mozgékonyságát. Ahhoz, hogy ezt elérje, a végjátékig a mozgékonysági játékos a következő kiértékelési függvényt használja:

$$EVAL(s) = w_1(c_{player} - c_{opponent}) + w_2\frac{m_{player} - m_{opponent}}{m_{player} + m_{opponent}}, \tag{17}$$

ahol a w_1 és a w_2 a súly paraméterek, a c_{player} a játékos által birtokolt sarki mezők száma, a $c_{opponent}$ az ellenfél által birtokolt sarki mezők száma, az m_{player} a játékos mozgékonysága és az $m_{opponent}$ az ellenfél mozgékonysága. A w_1 súly értéke 10, a w_2 súly értéke 1.

A mozgékonysági játékos számára a végjáték ugyanakkor kezdődik, mint a pozicionáló játékos esetében, azaz amikor a tábla mezőinek legalább 80%-át már elfoglalták. A végjáték alatt a játékos célja megegyezik a pozicionáló játékoséval. Ezért a végjáték alatt a mozgékonysági játékos is ugyanazt a kiértékelési függvényt használja, mint a pozicionáló játékos, azaz a (16)-os függvényt.

4.3. Q-tanuló játékos

A Q-tanuló játékos az egyetlen, amely tanulási viselkedést mutat. Ez a játékos a Q-tanulási algoritmust használja, hogy megtanulja melyik a legjobb lépés a játszma egy adott állapotában. Egyetlen olyan speciális tábla jellemzőt se használok fel, amit emberi szakértő választott ki, mint például a játékosok korongjainak száma vagy a játékosok mozgékonysága. A játszma állapota alapján a játékos eldönti, hogy melyik cselekvést végezze el. Ez a cselekvés az a lépés, amit megjátszik. A játékos jutalma 0 a játszma végéig. A játszma befejezésétől függően a jutalma $+1$ a győzelemért, -1 a vereségért és 0 a döntetlenért, ami megfelel a zéróösszegű játék fogalmának. A játékos célja a maximális jutalmat jelentő optimális cselekvések kiválasztása.

A Q-tanulási algoritmus paramétereinek értékei próbaképpen lettek kiválasztva, következésképpen nem biztos, hogy optimálisak. A Q-tanulási algoritmus α tanulási rátáját 0.1-re, a γ diszkontálási rátáját 1-re állítottam. A tanulási ráta nem változik a tanulás alatt. Mint ahogy korábban leírtam, a γ 1-re állításával a közvetlen és a jövőbeli jutalmak esetén a súlyok megegyeznek. Ez elfogadható, mert mi csak nyerni akarunk, nem pedig nyerni amilyen gyorsan csak lehet. A Q-tanuló játékos a softmax aktivációs kiválasztó módszert használja, ahogy azt már leírtam a 2.3. alfejezetben. Ez a kiválasztó módszer a T hőmérsékletet használja, ami a felderítés mértékét irányítja. A hőmérséklet az n számú eddig lejátszott játszma csökkenő függvénye lesz a következőképpen:

$$T = ab^n \tag{18}$$

adott a és b konstansokkal. Egy c konstanst is használok. Amikor az $ab^n < c$ ahelyett, hogy softmax aktivációs kiválasztást használna a játékos, inkább csak egyszerűen kiválasztja azt a cselekvést, amelyiknek a legnagyobb a Q-értéke. Az a, b és c konstansokat 1-re, 0.9999995-re és 0.002-re állítom. A hőmérséklet ilyen változása miatt van egy fokozatos átmenet a felderítésből a hasznosításba. Körülbelül $12\,000\,000$ játszma után, amikor az $ab^n = c$, a felderítés megáll és a játékos mindig hasznosítja a tudását.

Két különböző típusú Q-tanuló játékossal kísérletezek. A Q-tanuló játékosok csak a Q-értékek tárolásában különböznek. Ezen a módon jól összehasonlíthatjuk a két játékost. A Q-tanuló játékos két típusa közötti különbséget részletesen leírom.

Az egyszerű-NN Q-tanuló (single-NN Q-learner) egy egyszerű előrecsatolt neurális hálózatot használ külön kimeneti csomóponttal minden egyes cselekvésre. A neurális hálózatnak 64 bemeneti csomópontja van. Ezeket a csomópontokat használva adjuk be a környezet állapotait a hálózatnak. A csomópontok megfelelnek a táblán lévő 64 mezőnek. Egy bemeneti csomópont aktiválása $+1$, ha a játékos saját korongja van az adott mezőn, -1, ha az ellenfél korongja van a mezőn és 0, ha a mező üres. A hálózatnak van egy rejtett rétege 44 $\tanh$ csomóponttal és egy kimeneti rétege 64 $\tanh$ csomóponttal. Minden kimeneti csomópont megfelel egy cselekvésnek (egy mező a táblán). Egy kimeneti csomópont értéke -1 és 1 között van és megfelel egy lépés Q-értékének. Természetesen amikor egy cselekvést kiválasztunk, csak a szabályos lépések Q-értékeit vesszük figyelembe. A neurális hálózat tanulási rátáját (amit nem szabad összetéveszteni a Q-tanulási algoritmus tanulási rátájával) 0.1-re állítom. A hálózat súlyainak véletlen kezdőértékeket adok a -0.1 és 0.1 közötti egyenletes eloszlásból.

Az összetett-NN Q-tanuló (multi-NN Q-learner) külön előrecsatolt neurális hálózatokat használ minden egyes cselekvésre. A neurális hálózatok száma 64. Minden hálózatnak 64 bemeneti csomópontja van. Ugyanúgy, mint az egyszerű-NN Q-tanulónál, ezeket a csomópontokat is a tábla mezőinek reprezentálására használom. Minden hálózatnak van egy rejtett rétege 36 $\tanh$ csomóponttal és egy kimeneti rétege 1 $\tanh$ csomóponttal. A kimeneti csomópont értéke -1 és 1 között van és megfelel a lépés Q-értékének. Minden neurális hálózat tanulási rátáját 0.1-re állítom. Minden hálózat súlyainak véletlen kezdőértékeket adok a -0.1 és 0.1 közötti egyenletes eloszlásból.

Az egyszerű-NN Q-tanuló által és az összetett-NN Q-tanuló által használt paraméterek teljes száma $5\,632$ és $149\,760$. Ez hatalmas tömörítést jelent összehasonlítva a 10^{29} paraméterrel, amikre szükség lenne ha kereső táblát használnék.

A kiértékelés alatt a viszonyítási alapként használt játékosokkal szemben a Q-tanuló játékos mindkét típusa a következő kiértékelési függvényt használja:

$$EVAL(s) = \max_{a \in A} \hat{Q}(s, a). \tag{19}$$

A Q-tanuló játékosok a kiértékelés alatt nem használnak felderítési eljárásmódot a cselekvés kiválasztására, de ahogy a (19)-es egyenlet mutatja, használnak olyan eljárásmódot, ami optimális a becsült Q-értékeknek megfelelően.

5. Az Othello táblajáték számítógépes megjelenítése

A szakdolgozatom fő témáját, a megerősítéses tanulás gyakorlati alkalmazását az Othello játékon keresztül mutatom be. A játék és a neurális hálózatok elkészítéséhez az Eclipse és az Encog rendszereket használtam fel. A programban a következőképpen lehet az Othelloval játszani:

- egy számítógépen ketten.
- a számítógép ellen, ami előre programozott stratégiákat használ.
- a számítógép ellen, ami neurális hálózatok segítségével tanult meg játszani.

Megfigyelhető az is, hogyan és milyen teljesítménnyel játszanak egymás ellen az egyes stratégiák és neurális hálózatok. A program lehetőséget ad arra is, hogy a számítógépet neurális hálózatok segítségével néhány paraméter megadásával megerősítéses tanulással megtanítsuk az Othello játékra, a betanított neurális hálózatokat el lehet menteni, majd ezek ellen is lehet játszani.

5.1. A grafikus felhasználói felület

Java nyelven készített programom grafikus felhasználói felületet is tartalmaz. Ezen keresztül lehet a program egyes funkcióit használni. A program kezelőfelülete angol nyelvű. A tanítási részben billentyűzeten keresztül lehet megadni a neurális hálózatok tanításához szükséges paraméterek értékeit. A játék minden további részében csak az egeret lehet használni.

A program kezdőablaka három választási lehetőséget kínál fel: a PLAY, a TRAIN és az EXIT gombokat.

Az EXIT gomb a program befejezését eredményezi.

A másik kettő választási lehetőséget a következő alfejezetekben írom le bővebben.

5.1.1. A tanítás

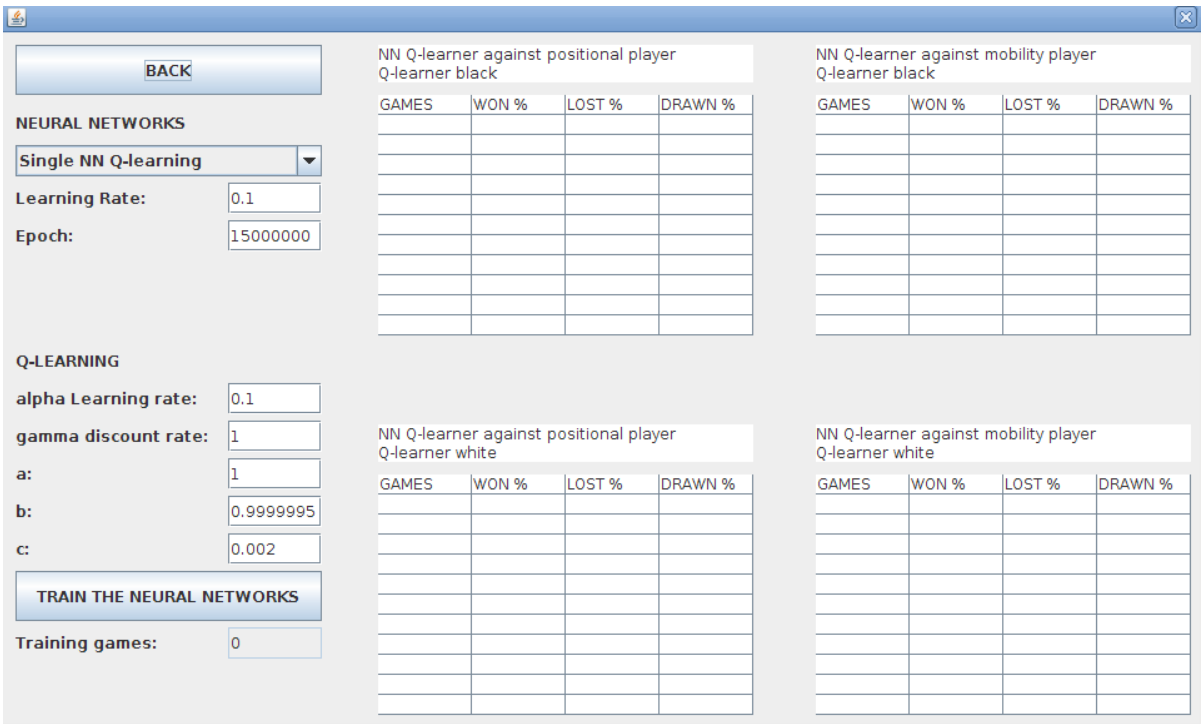
Ha a TRAIN gombot választottuk, akkor a megjelenő ablakban a neurális hálózatok tanítását végezhetjük el, valamint a betanított neurális hálózatok teszteléseinek eredményét láthatjuk táblázatokban. Neurális hálózatnak Single-NN Q-learning vagy Multi-NN Q-learning állítható be. Megadhatjuk a neurális hálózat és a Q-tanulás paramétereit is.

Az ablak bal felső sarkában található BACK gomb megnyomásával az aktuális ablak eltűnik és a kezdőablakba kerülünk vissza. Ha ismét tanítani szeretnénk a neurális hálózatokat, akkor újra a TRAIN gombot kell megnyomni és be kell állítani az általunk kívánt paraméterek értékeit, ugyanis a paraméterek és a táblázatok az eredeti kezdőértékeiket veszik fel.

A TRAIN THE NEURAL NETWORKS gomb megnyomásakor a neurális hálózatok tanítását lehet elindítani, ami egy előre megadott számú játszma után megáll és a neurális hálózatokat automatikusan leteszteli a program. A tesztek eredményei bekerülnek a táblázatokba.

A tanítás bármikor megállítható a STOP gomb megnyomásával. A TRAIN THE NEURAL NETWORKS gomb ismételt megnyomásával új tanítás kezdődik a már beállított paraméterekkel.

A SAVE gombbal elmenthetjük a betanított neurális hálózatokat, amik ellen akár majd játszhatunk is. Ehhez vissza kell lépni a kezdőablakba és a PLAY gombot kell megnyomni.



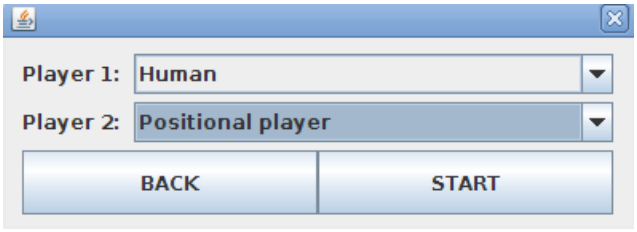
5.1.2. A játék

Ha a PLAY gombot választottuk, akkor a megjelenő ablakban kiválaszthatjuk az egyes játékosokat (Player 1, Player 2). Alapbeállításként a Human játékos van beállítva mindkettőhöz. Ezt azonban szabadon átállíthatjuk. A program első indításakor csak három játékos található meg a választhatók között, ezek a Human, a Positional player és a Mobility player. Az utóbbi kettő stratégiát a 4.1. és a 4.2. alfejezetekben írtam le. Ha ezeket állítjuk be bármelyik játékoshoz, akkor azt a számítógép fogja irányítani. Ha neurális hálózatokat tanítottunk be, akkor azok is megjelennek a választható játékosok listáján.

A játékosok alatt kettő gomb található: a BACK és a START.

A BACK gomb megnyomásával az aktuális ablak eltűnik és a kezdőablakba kerülünk vissza. A játékosokon végzett módosítások érvényüket vesztik és a PLAY gomb ismételt megnyomásával mindegyik alapbeállítás szerinti értéket vesz fel.

A START gomb megnyomásával elindul az Othello játék egy új ablakban.



A játék elindulásakor az ablakban egy PASS és egy END GAME gombot, egy Othello táblát, egy Game History-t valamint a játszma állását és az aktuális játékost láthatjuk kiírva.

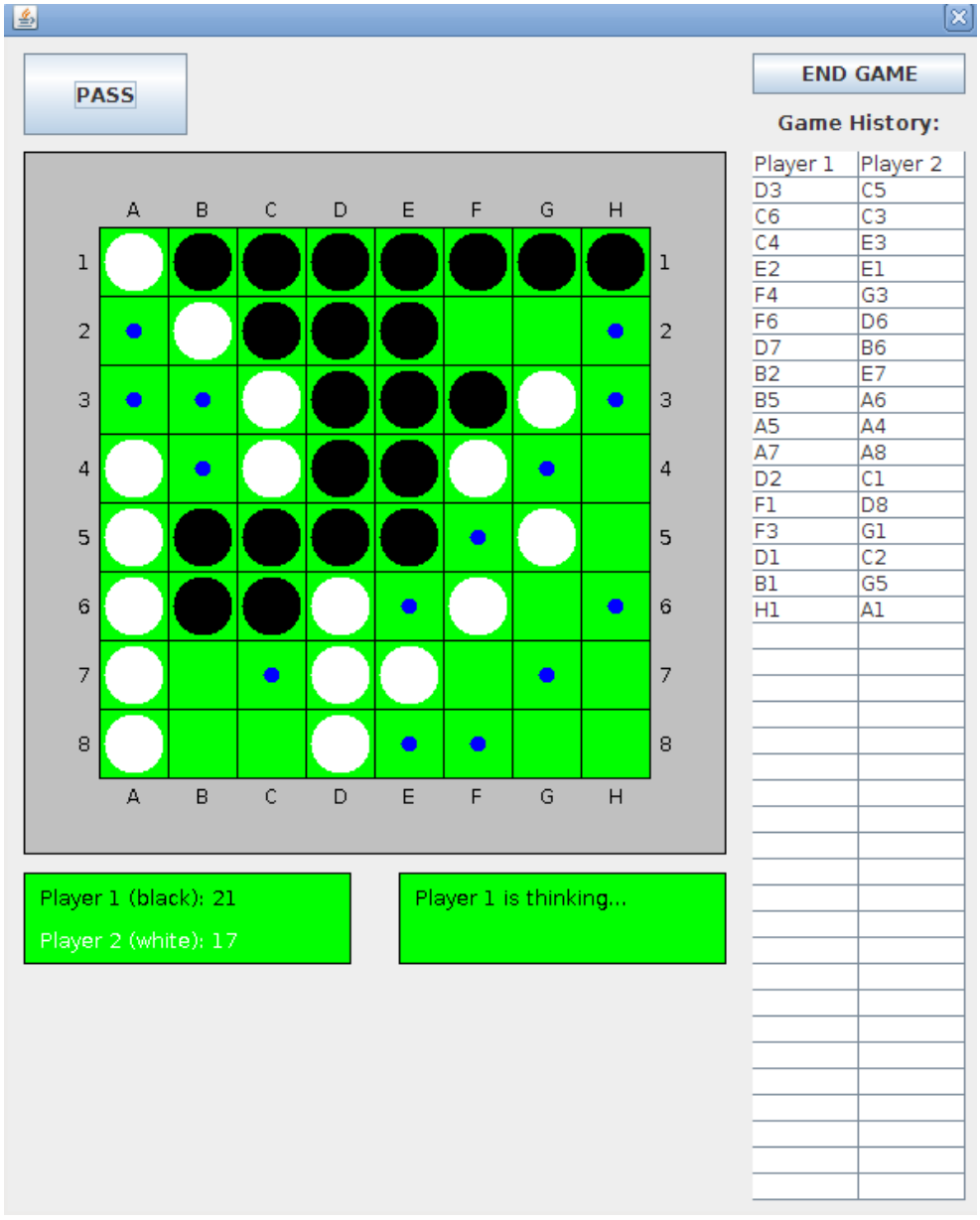
A PASS gomb csak egy meghatározott esetben reagál a megnyomására. Az Othello szabálya szerint csak akkor, ha az aktuális játékos nem tud érvényesen lépni. Ha kétszer egymás után PASS került a Game History-ba, akkor a PASS gombot már nem lehet megnyomni, a játszma ilyenkor befejeződik.

Az END GAME gomb megnyomásával az aktuális ablak eltűnik és az előzőbe kerülünk vissza. A START gomb ismételt megnyomásával a játék tábláját tartalmazó ablakban minden alapbeállítás szerinti értéket vesz fel.

A Game History táblázatába a játékosok egyes lépéseinek koordinátái kerülnek egy betű és egy szám karakter formájában,valamint ha nem tud szabályosan lépni a játékos, akkor a PASS karaktersorozat. A táblázat első oszlopa a fekete játékosé (kezdő játékos), a második oszlopa a fehér játékosé. A táblázat nem tartalmazza a játszma elején az Othello táblán lévő 4 korong koordinátáit.

A játékosok korongjainak száma a játszma aktuális állása, ami a játéktábla alatt bal oldalon van kiírva. Jobb oldalon az éppen lépő játékos, valamint ha a játszma befejeződik, a győztes, vagy a döntetlen eredmény lesz kiírva.

A játéktábla alapbeállításként tartalmazza a kezdő négy korongot, valamint a kezdő játékos lehetséges lépéseit, amelyek kis kék körökkel vannak ábrázolva. Az aktuális játékos csak ezek egyikére helyezheti el a saját korongját.



5.2. A játék egyes elemeinek reprezentációja

A játék tábláját a programban egy 2 dimenziós tömbbel ábrázolom, ami a játszma aktuális állapotát mutatja, nullákkal az üres mezőket, egyesekkel az első (fekete) játékos által birtokolt

mezőket, kettesekkel a második (fehér) játékos által birtokolt mezőket jelöltem. A hármasok az aktuális játékos által megtehető szabályos lépéseket jelölik. A kezdőállapotot a következő tömb reprezentálja a programban:

```
int[][] board ={
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 3, 0, 0, 0, 0},
    {0, 0, 3, 2, 1, 0, 0, 0},
    {0, 0, 0, 1, 2, 3, 0, 0},
    {0, 0, 0, 0, 3, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0}
};
```

A pozicionáló játékos által felhasznált (a 4.1. alfejezetben ismertetett) táblázatot is egy 2 dimenziós tömbben tárolom a programban, amely a következő:

```
int[][] positional ={
    {100, -20, 10, 5, 5, 10, -20, 100},
    {-20, -50, -2, -2, -2, -2, -50, -20},
    {10, -2, -1, -1, -1, -1, -2, 10},
    {5, -2, -1, -1, -1, -1, -2, 5},
    {5, -2, -1, -1, -1, -1, -2, 5},
    {10, -2, -1, -1, -1, -1, -2, 10},
    {-20, -50, -2, -2, -2, -2, -50, -20},
    {100, -20, 10, 5, 5, 10, -20, 100}
};
```

6. Kísérletek és eredmények

Két olyan kísérletet hajtottam végre, amelyekben az ágensek Q-tanulást használtak, hogy megtanulják az Othello játékot. Az első kísérletben két egyszerű-NN Q-tanuló játszott egymás ellen, míg a második kísérletben két összetett-NN Q-tanulót használtam. Mindkét kísérletben egy előre meghatározott számú gyakorló játszma után a tanítást kikapcsoltam és mindkét Q-tanuló játékost kiértékeltem. A kiértékelés során 244 játszmát játszottak le a két viszonyítási alapként szolgáló játékos ellen, amelyek a pozicionáló és a mozgékonysági játékosok voltak.

Mivel a kiértékelés alatt mindkét Q-tanuló játékos és a viszonyítási alapként szolgáló játékosok is determinisztikus eljárásmódot használnak (kivéve a pozicionáló és a mozgékonysági játékosok esetén akkor, amikor több azonos értékű lépés közül kell választaniuk, ilyenkor véletlenszerű kiválasztást használnak), ezért a kiértékelt játszmáknak különböző pozíciókból kell

indulniuk. Ezeket a különböző pozíciókat úgy határoztam meg, hogy négy véletlen lépést tettem a kiindulási pozícióból. Négy ilyen véletlen lépés összesen 244 állapotba vihet. A teljesítmény mérésére a Q-tanulók által el nem veszített kiértékelt játszmák százalékos arányát használtam.

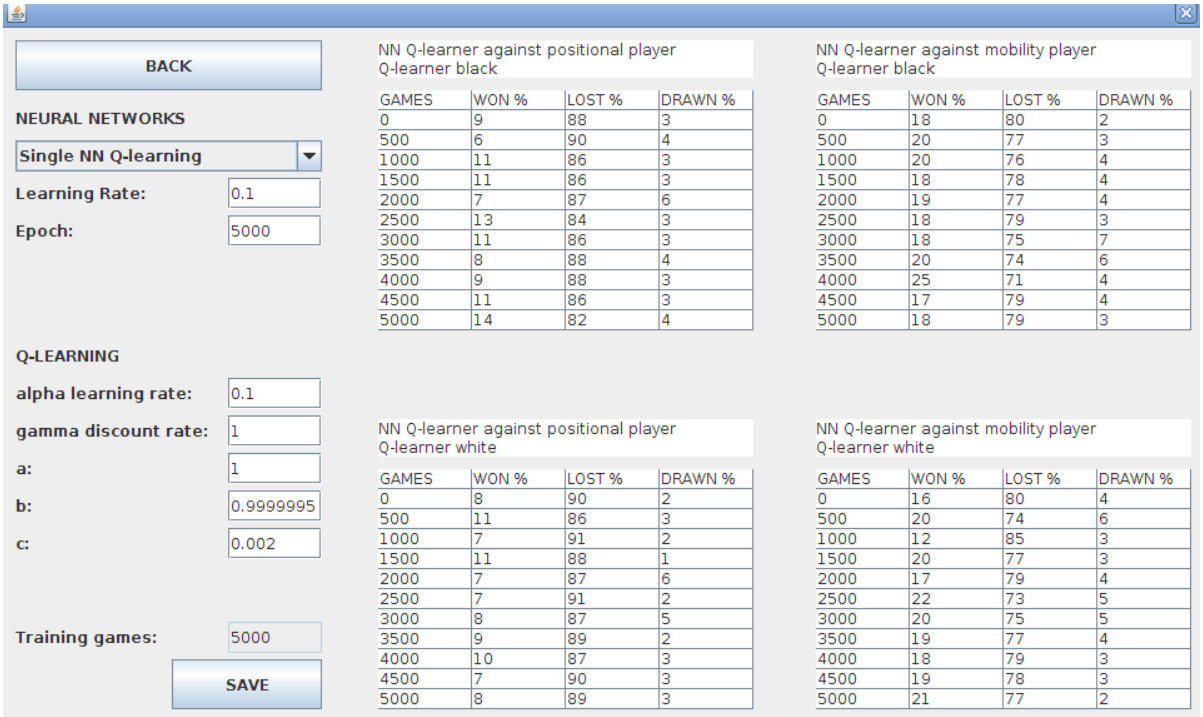
A tanítási fázis alatt a beállított tanítás hosszát (a tanításhoz felhasznált játszmák száma) 10 egyenlő részre osztottam, így a tanítás alatt minden egyes ilyen rész után a neurális hálózatokat leteszteltem. Ezzel a tanítási fázis alatt képet kaphatunk arról, hogy a neurális hálózatok adott pillanatban mennyire tudnak jól játszani az Othelloval.

A kiértékelési fázis alatt a kísérletekben minden ágens a saját kiértékelési függvényét használta, amelyeket a 4. fejezetben ismertettem, azaz a különböző típusú ágensek különböző kiértékelési függvényeket használnak.

A kísérletek eredményei táblázatokba kerülnek. A táblázatokban a megnyert, az elveszített és a döntetlen játszmák százalékos megoszlását láthatjuk a Q-tanuló szemszögéből. A táblázatok külön mutatják a pozicionáló és a mozgékonysági játékosok elleni eredményeket aszerint, hogy a Q-tanuló fekete vagy fehér korongokkal játszott.

6.1. 1. kísérlet: Othello játék tanulása egyszerű-NN Q-tanulókkal

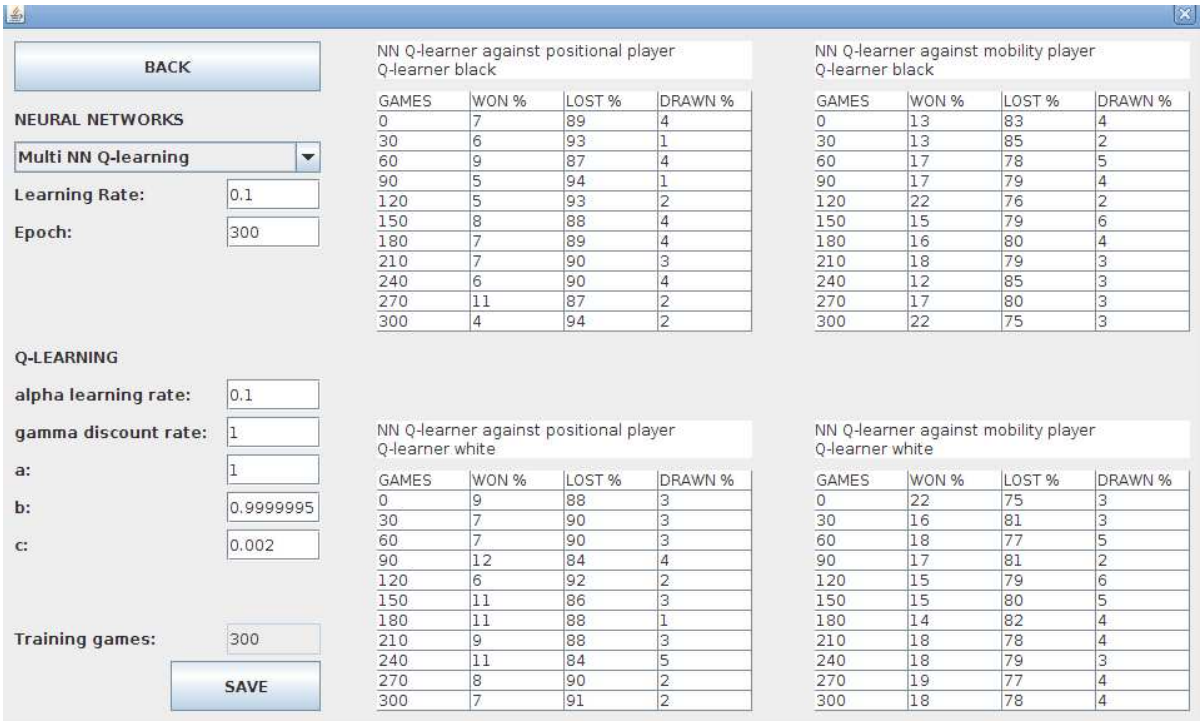
Az első kísérletben két egyszerű-NN Q-tanuló tanulta az Othello játékot egymás ellen. A következő kép mutatja a kísérlet eredményeit:



Az első kísérletben a szakdolgozatomban már korábban leírt értékeket adtam meg a tanítás paramétereinek, de néhány algoritmus csak hosszabb ideig tartó tanítás során fut le. Ezt a tanítás képernyőjén megadott paraméterekkel módosíthatjuk.

6.2. 2. kísérlet: Othello játék tanulása összetett-NN Q-tanulókkal

A második kísérletben két összetett-NN Q-tanuló tanult meg játszani az Othelloval úgy, hogy egymás ellen játszottak. A következő kép mutatja a kísérlet eredményeit:



A második kísérlet során a tanítási paraméterek értékei megegyeznek az első kísérlet során használtakkal, kivéve a tanítás hosszát, ugyanis az összetett neurális hálózatok lassabban tanulnak.

6.3. Tapasztalatok és következtetések

A tanítás erőforrásigénye nagy, de ez az egyes algoritmusok optimalizálásával csökkenthető. A két különböző kísérletben az alkalmazott neurális hálózatok összetettsége, bonyolultsága is különbözik. Azt tapasztaltam, hogy az első kísérletben szereplő Q-tanulók gyorsabban tanulnak, mint a második kísérletben lévő Q-tanulók. A neurális hálózatok által használt algoritmusok módosításával, a paraméterek értékeinek megváltoztatásával vagy más algoritmusok kombinálásával jobb és gyorsabb tanulást érhetünk el. A neurális hálózatok szerkezetének megváltoztatásával is befolyásolhatjuk a kísérletek eredményeit.

7. Összefoglalás

Szakdolgozatomban egy gépi tanulási algoritmust, a megerősítéses tanulást írtam le a szekvenciális döntéshozási problémák megoldására. A megerősítéses tanulás képes közelítő megoldást adni nagy szekvenciális döntéshozási problémákra úgy, hogy függvényközelítő módszert használ. Leírtam a Q-tanulást, egy gyakran használt megerősítéses tanulási algoritmust, amit neurális hálózatokkal kombináltam.

Az Othello játékot használtam fel a Q-tanulás alkalmazásának bemutatására. Az volt a célom, hogy tanulmányozzam a különböző Q-tanulási ágensek képességét az Othello játék megtanulására anélkül, hogy bármilyen emberi szakértő által biztosított tudást használnának. Ez nem oldható meg a hagyományos dinamikus programozási technikákkal, mert az Othellonak hatalmas az állapottere.

Az Othello kísérletekben megvizsgáltam két különböző típusú Q-tanulási ágenst. Tanulmányoztam olyan Q-tanulókat, amelyek egy egyszerű neurális hálózatot használnak külön kimeneti csomóponttal minden egyes cselekvésre és olyan Q-tanulókat, amelyek külön neurális hálózatot használnak minden egyes cselekvésre. Kereső táblát használó Q-tanulókat nem tanulmányoztam, mert az túl sok memóriát igényelt volna. Úgy tűnik, hogy azok a Q-tanulók, amelyek egyszerű neurális hálózatot használnak az Othello megtanulására gyorsabbak, mint azok a Q-tanulók, amelyek külön neurális hálózatot használnak minden egyes cselekvésre. Az egyes neurális hálózatok implementációjának optimalizálásával pontosabb eredményt kaphatunk erről.

Tanulmányozható a speciális tábla jellegzetességek felhasználásának hatása a Q-tanulási ágensekre és a tanulás egyszerűsítésére, bár ez megsérti az Othello játék bármilyen emberi szakértő által biztosított tudás nélküli tanulását. Ilyen érdekes tábla jellegzetesség lehet az Othello játékban például a mezők beszámítása a sarkok, az átlók és a sorok figyelembevételével. Ezek a tábla jellegzetességek fontos Othello fogalmakat foglalhatnak magukba, mint például a stabilitás. Ezek a jellegzetességek könnyen felismerhetők emberi játékosok számára a vizuális rendszerük miatt, de egy számítógépnek ezek nehezen felismerhetők, akárcsak a tábla minták.

A megerősítéses tanulásnak sok lehetséges alkalmazása lehet az operációkutatás és a gazdaságtudomány területén. Néhány alkalmazást már megemlítettem a bevezetésben. White [9] általános áttekintést nyújt az MDP alkalmazásokról az operációkutatásban. Lehetséges, hogy az ilyen problémák közül néhány újraformalizálható úgy, hogy a megerősítéses tanulás azon képességét használjuk fel, hogy képes függvényközelítés során általánosítani.

A probléma szabályokba foglalásának pontatlansága és a megoldás pontatlansága közötti kompromisszum is egy érdekes téma. A pontatlanság első formája akkor következik be, amikor

a döntéshozási probléma szándékosan egyszerű, hogy a dinamikus programozású alkalmazás lehetséges legyen. A második formája pedig akkor következik be, amikor a becsült értékű függvények pontatlanok a függvényközelítés használata miatt. Néhány problémára jobb megoldást kaphatunk, ha a függvényközelítésből származó pontatlanságot részesítjük előnyben a probléma egyszerű leírásából származó pontatlansággal szemben.

8. Irodalomjegyzék

[1] Moriarty DE, Miikkulainen R. Discovering complex Othello strategies through evolutionary neural networks. Connection Science (1995);7(3):195–210.

[2] Chong SY, Tan MK, White JD. Observing the evolution of neural networks learning to play the game of othello. IEEE Transactions on Evolutionary Computation (2005);9(3):240–51.

[3] Nees Jan van Eck , Michiel van Wezel. Application of reinforcement learning to the game of Othello. Computers & Operations Research 35 (2008);1999–2017.

`http://www.computerscience.nl/docs/vakken/ici/RL_NN_Othello.pdf`

[4] Bellman RE. Dynamic programming. Princeton, NJ, USA: Princeton University Press; (1957).

[5] Watkins CJCH. Learning from delayed rewards. PhD thesis, Cambridge University, Cambridge, England; (1989).

[6] Watkins CJCH, Dayan P. Q-learning. Machine Learning (1992);8(3):279–92.

[7] Hu J, Wellman MP. Nash Q-learning for general-sum stochastic games. Journal of Machine Learning Research (2003);4:1039–69.

[8] Doucette MJ. Wipeout: the engineering of an Othello program. Project report, Acadia University, Wolfville, NS, Canada; (1998).

[9] White DJ. A survey of applications of Markov decision processes. The Journal of the Operational Research Society (1993);44(11):1073–96.

[10] Stuart J.Russell, Peter Norvig: Mesterséges intelligencia - Modern megközelítésben, Panem, (2005).

[11] Altrichter Márta, Horváth Gábor, Pataki Béla, Strausz György, Takács Gábor, Valyon József: Neurális hálózatok, Panem, (2007).

[12] Wettl Ferenc, Mayer Gyula, Szabó Péter: Latex kézikönyv, Panem, (2004).

9. Mellékletek

Ebben a fejezetben a szakdolgozatomhoz készített program néhány forráskód részlete található.

A neurális hálózatok teszteléséhez az első négy véletlen kezdőlépést az alábbi programrészlet segítségével lehet megadni:

```
1   public void randomActions(){
2     randomsteps = new int[244][8];
3         int counter1 = 0;
4         for(int a = 0; a < 8; a++){
5           for(int b = 0; b < 8; b++){
6             if(board[a][b] == 3){
7               for (int x = 0; x < 8; x++){
8               for (int y = 0; y < 8; y++){
9                 board2[x][y] = board[x][y];
10              }
11            }
12                executeAction(a, b, board2);
13                for(int c = 0; c < 8; c++){
14                  for(int d = 0; d < 8; d++){
15                    if(board2[c][d] == 3){
16                          for (int x = 0; x < 8; x++){
17                          for (int y = 0; y < 8; y++){
18                            board3[x][y] = board2[x][y];
19                          }
20                      }
21                          executeAction(c, d, board3);
22                          for(int e = 0; e < 8; e++){
23                            for(int f = 0; f < 8; f++){
24                              if(board3[e][f] == 3){
25                                    for (int x = 0; x < 8; x++){
26                                    for (int y = 0; y < 8; y++){
27                                      board4[x][y] = board3[x][y];
28                                    }
29                                }
30                                    executeAction(e, f, board4);
31                                    for(int g = 0; g < 8; g++){
32                                      for(int h = 0; h < 8; h++){
33                                        if(board4[g][h] == 3){
34                                          randomsteps[counter1][0] = a;
35                                          randomsteps[counter1][1] = b;
36                                          randomsteps[counter1][2] = c;
```

```
37                                          randomsteps[counter1][3] = d;
38                                          randomsteps[counter1][4] = e;
39                                          randomsteps[counter1][5] = f;
40                                          randomsteps[counter1][6] = g;
41                                          randomsteps[counter1][7] = h;
42                                          counter1++;
43                                  }
44                                          }
45                                      }
46                                      player = 3 - player;
47                                  nextplayer = 3 - nextplayer;
48                                    }
49                                  }
50                              }
51                              player = 3 - player;
52                          nextplayer = 3 - nextplayer;
53                        }
54                      }
55                    }
56                    player = 3 - player;
57                nextplayer = 3 - nextplayer;
58                  }
59                }
60              }
61    }
```

Az alábbi programrészlet a játszma során az aktuális lépésnek megfelelően írja át (színezi) a táblán lévő korongokat:

```
1   public int colorizeBoard(int x, int y, int irX, int irY, int player
        , int[][] xboard){
2       int ret = -1;
3     if(x >= 8 || x < 0 || y >= 8 || y < 0)
4         return -1;
5     if(irX == 0 && irY == 0){
6       xboard[x][y] = player;
7       return 1;
8     }
9       if(xboard[x][y] == 0 || xboard[x][y] == 3)
10        return -1;
11      if(xboard[x][y] == player)
12        return 1;
13        ret = colorizeBoard(x + irX, y + irY, irX, irY, player,
              xboard);
```

```
14        if(ret == 1)
15          xboard[x][y] = player;
16        return ret;
17      }
```

A következő programrészlet a következő játékos által megtehető lépéseket adja meg:

```
1   public void nextPlayerPossibleSteps(int[][] xboard){
2     for(int i = 0; i < 8; i++){
3         for(int j = 0; j < 8; j++){
4           if(xboard[i][j] == player){
5             for(int x = -1; x <= 1; x++){
6               for(int y = -1; y <= 1; y++){
7                 if((i + x) > 7 || (i + x) < 0 || (j + y) > 7 || (j +
                      y) < 0)
8                   continue;
9                 if(xboard[i + x][j + y] == nextplayer)
10                  findPossible(i + x, j + y, x, y, nextplayer, xboard
                        );
11              }
12            }
13          }
14        }
15    }
16  }
17
18  public void findPossible(int x, int y, int irX, int irY, int color,
        int[][] xboard){
19      if (x < 0 || x > 7 || y < 0 || y > 7)
20        return;
21      if (xboard[x][y] == 3)
22        return;
23      if (xboard[x][y] == 0) {
24        xboard[x][y] = 3;
25        return;
26      }
27      if (xboard[x][y] == color)
28        findPossible(x + irX, y + irY, irX, irY, color, xboard);
29  }
```

A következő programrészlet egy egyszerű-NN Q-tanuló neurális hálózatának létrehozása:

```
1     BasicNetworkAG network1 = new BasicNetworkAG();
2     network1.addLayer(new BasicLayer(new ActivationTANH(), false, 64)
          );
```

```
3     network1.addLayer(new BasicLayer(new ActivationTANH(), false, 44)
          );
4     network1.addLayer(new BasicLayer(new ActivationTANH(), false, 64)
          );
5     network1.setLogic(new FeedforwardLogic());
6     network1.getStructure().finalizeStructure();
7     network1.reset();
```

Az alábbi programrészlet egy összetett-NN Q-tanuló neurális hálózatainak létrehozása:

```
1     BasicNetworkAG[] network1 = new BasicNetworkAG[64];
2     for(int i = 0; i < 64; i++){
3       network1[i] = new BasicNetworkAG();
4       network1[i].addLayer(new BasicLayer(new ActivationTANH(), false
            , 64));
5       network1[i].addLayer(new BasicLayer(new ActivationTANH(), false
            , 36));
6       network1[i].addLayer(new BasicLayer(new ActivationTANH(), false
            , 1));
7       network1[i].setLogic(new FeedforwardLogic());
8       network1[i].getStructure().finalizeStructure();
9       network1[i].reset();
10    }
```