EÖTVÖS LORÁND TUDOMÁNYEGYETEM

TERMÉSZETTUDOMÁNYI KAR

TÁBLAJÁTÉKOK ÉS GÉPI TANULÁS

SZAKDOLGOZAT

Szlachányi Tádé

Matematika BSc

Matematikai elemző szakirány

Témavezető: Lukács András

Számítógéptudományi Tanszék

Budapest

2020

NYILATKOZAT

Név: Szlachányi Tádé

ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: XS1WAH

Szakdolgozat címe:
Táblajátékok és gépi tanulás

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2020.12.29

Szlachányi Tádé

a hallgató aláírása

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek, Lukács Andrásnak azért, hogy szakmai segítségére bármikor számíthattam, illetve a számtalan óráért, amit arra áldozott, hogy a szakdolgozatom létrejöhessen. Továbbá szeretnék családomnak megköszönni minden segítséget és támogatást amit kaptam tőlük.

Tartalomjegyzék

1. Bevezetés **1**

2. Módszerek **3**

2.1. Minimax 3

2.2. Felügyelt tanulás 5

2.3. Megerősítéses tanulás 6

3. Amőba **9**

3.1. Az amőba reprezentációja 11

3.2. Minimax algoritmus alkalmazása amőbára 12

3.3. Megerősítéses tanulás alkalmazása amőbára 15

4. Sakk **23**

4.1. A sakk reprezentációja 23

4.2. Minimax algoritmus alkalmazása sakkra 25

4.3. Megerősítéses tanulás alkalmazása sakkra 27

4.4. Felügyelt tanulás alkalmazása sakkra 29

4.5. Fejlesztési lehetőségek 32

5. Összefoglalás **34**

1. fejezet

Bevezetés

Mióta megalkották az első modern számítógépeket, az embereket folyamatosan foglalkoztatja a kérdés, hogy a gépek el tudják-e érni a humán intelligencia szintjét. Bár a számítógépek már másodpercenként több milliárd számítást tudnak elvégezni, rengeteg olyan feladat maradt, ami az emberek számára teljesen triviális, mint például egy kép, vagy egy hang felismerése, de a gép számára lényegében megoldhatatlan nehézségeket okoz. Számos ilyen jellegű probléma van, ami továbbra is megoldatlan, azonban létezik néhány terület, melyekben az elmúlt években hatalmas fejlődés történt. Az egyik ilyen területet a játékok jelentik. Számos játék létezik, amiben az elmúlt időben jutott el arra a szintre a technológia, hogy az ember által programozott számítógépek legyőzzenek minket, ilyen például a sakk. Garry Kasparov nemzetközi nagymester sakkozó, és korábbi világbajnok az 1980-as években azt állította, hogy sose fog olyan sakk program létezni, ami őt meg tudná verni. 1997-ben az IBM által fejlesztett Deep Blue nevű gép 3,5 : 2,5 -re legyőzte Kasparovot. Annak ellenére, hogy ez csak egy sakkmérkőzés volt, sokak szemében hatalmas szimbolikus jelentőséggel rendelkezett, hiszen azt jelezte, hogy a gépek képesek meghaladni az emberek intelligencia szintjét [1] .

A társasjátékokat hagyományosan az alábbi módon osztályozzák : táblajátékok (board games), kártyajátékok (card games), kockajátékok (dice games), papír és toll játékok (paper and pencil games), szerepjátékok (tabletop-role playing games), stratégiai játékok (strategy games), csempe alapú játékok (tile-based games). Létezik azonban két másik kategorizációs szempont is, amelyek a gépi tanulás szempontjából sokkal nagyobb jelentőséggel bírnak. Az egyik osztályozás determinisztikus (deterministic), illetve sztochasztikus (stochastic) játékokat különböztet meg. A determinisztikus játékokban előre ki lehet szá-

molni cselekvéseink eredményét, ilyen például a sakk, vagy a stratego, míg a stochasztikus játékokban szerepet kap a véletlen a játékállapot változásában, ilyen például a Monopoly, vagy a poker. A másik kategorizálás szerint létezik teljes információs (perfect information) és nem teljes információs (imperfect information) játék. Teljes információs játékban minden információ rendelkezésére áll az összes játékosnak, ilyen például a sakk, vagy a rulett, nem teljes információs játékban viszont a játékosok nem tudnak minden információt, ilyen például a poker, vagy a stratego. Szakdolgozatomban a teljes információs, determinisztikus játékokkal foglalkoztam, mivel ezek a legegyszerűbben modellezhetőek.

Diplomamunkámban azt vizsgálom, hogyan lehet a gépi tanulás módszereit felhasználni táblajátékokban mesterséges intelligencia létrehozására. A célom nem az volt, hogy olyan szintű programot készítsek, ami a mai legerősebb gépekkel versenybe tudna szállni, hanem, hogy minél több módszert, illetve különböző megközelítést bemutassak, majd kipróbáljam, hogy ezek közül melyik az, ami egy adott játékra jól működik, és melyik az, ami kevésbé. Próbáltam olyan módszereket tárgyalni, melyek általánosságban a táblajátékok nagy részére alkalmazhatóak, nem csak azokra, amiken teszteltem őket.

Szakdolgozatomban először bemutatom a felhasznált klasszikus mesterséges intelligencia, illetve gépi tanuláson alapuló módszereket, majd arról írok, ezeket hogyan alkalmaztam táblajátékokra. Ezután először az amőba játékban, később pedig a sakkban elért eredményeimről írok. Végül olyan módszereket is bemutatok, melyek nem fértek bele a szakdolgozat kereteibe, de úgy gondolom, hogy jól működhetnek. Szakdolgozatomban a neurális hálók és gépi tanulás témakörének alapfogalmait nem fogom bemutatni, ezek ismeretét feltételezem.

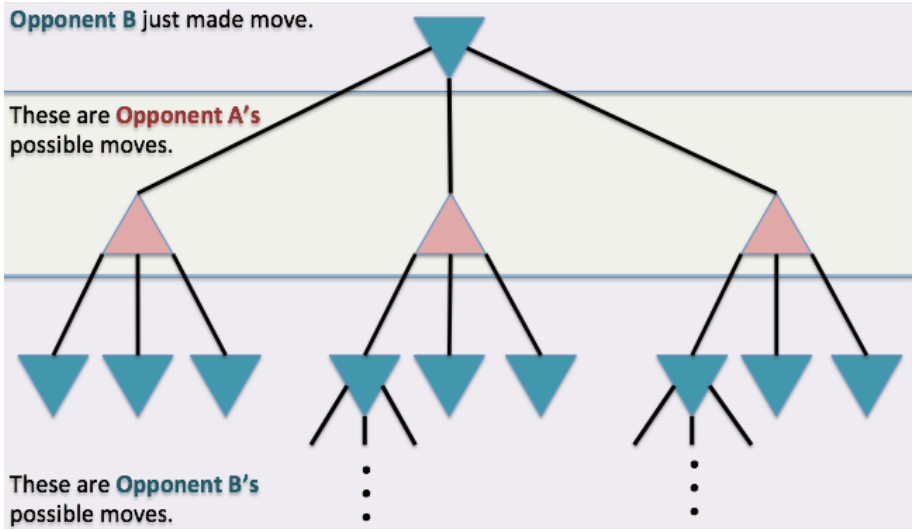
A módszerek tesztelésére, és a mérések megvalósítására programokat írtam Python nyelven. Ezeket az alábbi GitHub mappában lehet megtekinteni : `https://github.com/tszlachanyi/machine_learning_for_tabletop_games`.

2. fejezet

Módszerek

2.1. Minimax

Az egyik legszélesebb körben elterjedt döntéshozó módszer a minimax algoritmus. Használatának feltétele, hogy a játék kétszemélyes és zéró összegű játék legyen. Ez azt jelenti, hogy a játékosok érdekei olyan formában ellentétesek, hogy egy szituáció az egyik játékosnak pontosan annyira előnyös, amennyire a másiknak hátrányos. További feltétel, hogy determinisztikus, teljes információs (perfect information) játék legyen. Ez azért szükséges, hogy az algoritmus bármilyen állásból meg tudja alkotni a játék állapotterét, azt egy fa gráfként reprezentálva. A fa minden csúcsa a játék egy állásának felel meg. Gyökere a pillanatnyi állapot. A fában ezután minden csúcs gyerekei azoknak a játékállapotoknak felelnek meg, amelyek a szülőből egy lépéssel elérhetőek. Abban az állapotban, amiben az algoritmust meghívtuk, a soron lévő játékost nevezzük maximalizáló játékosnak, a másik játékost pedig minimalizáló játékosnak. A fa csúcsait szintekre lehet osztani, aszerint, hogy mekkora a távolságuk a gyökértől, vagyis hány lépéssel érhetőek el. Ekkor a páratlan szinteken azok az állapotok szerepelnek, melyekben a minimalizáló játékos következik, a páros szinteken pedig azok, melyekben a maximalizáló játékos (2.1.1. ábra). Az algoritmus működéséhez szükséges egy úgynevezett kiértékelő függvény, amely a játék minden végállapotához egy valós számot rendel. A legegyszerűbb ilyen függvény az, ha a maximalizáló játékos nyert az érték 1, ha a minimalizáló nyert -1, döntetlen esetén pedig 0. A kiértékelő függvény sokféle lehet, de fontos szempont, hogy minél kedvezőbb egy állapot a maximalizáló játékosnak, a hozzá rendelt érték annál nagyobb legyen, és minél kedvezőbb egy állapot a minimalizáló játékosnak, a hozzá rendelt érték annál kisebb legyen.



2.1.1. ábra. Állapottér reprezentációja fa gráffal [2]

Az algoritmus működése a következő: felrajzoljuk a játék állapotterének fáját a pillanatnyi állásból kiindulva. A fa leveleihez valamilyen kiértékelő függvény alapján értékeket rendelünk. Ezután az utolsó előtti szinttől visszafelé haladva értéket rendelünk a fa összes csúcsához. A maximalizáló játékos egy adott állapotban a lehetséges lépések közül azt akarja választani, ami számára a legkedvezőbb, vagyis ami a legnagyobb értékkel rendelkező állapothoz vezet, a minimalizáló játékos pedig azt, ami a legkisebb értékkel rendelkezőhöz. Ezért a maximalizáló játékos szintjein minden csúcshoz a gyerekei értékei közül a legnagyobbat rendeljük hozzá, a minimalizáló játékos szintjein pedig a gyerekei értékei közül a legkisebbet. Ha a fában minden állapot értéket kapott, akkor ezután a pillanatnyi állapotból kiválasztjuk azt a lépést, ami a legnagyobb értékhez vezet. Ez a lépés lesz az algoritmus döntése arra, hogy mit lépjen az adott állapotban.

Az algoritmusnak egyik hátránya a nagy számítási idő. Egy játék állapotterének nagyságára általában jó felső becslés n^m , ahol n az adott állapotban megtehető lépések száma, m pedig a játék hossza lépésekben mérve. Mivel az algoritmus műveletigénye exponenciálisan növekszik a lépések számával, ezért összetettebb játékokra való alkalmazása jelenlegi formájában nagyon lassú lenne. Ilyenkor csak M maximális mélységre futtatjuk, vagyis a fát nem a játék végéig rajzoljuk fel, csak annak az első M szintjéig. Ennek a fának azonban általában nem minden levele lesz a játék végállapota, éppen ezért az előbb említett kiértékelő függvény ebben az esetben nem működik. Ilyenkor olyan kiértékelő függvényt kell alkalmazni, amely a játék minden állapotán értelmezve van, nem pedig csak a végálla-

potokon. Az, hogy mi az ideális függvény játékonként eltér, és általában nem határozható meg egyértelműen.

Az algoritmus feltételezi, hogy az ellenfél optimálisan játszik, és ez alapján hozza meg a lehető legjobb döntést. Ez a megközelítés nem szerez előnyöket ellenfele potenciális hibáiból, ezért csak abban az esetben optimális, ha az ellenfél is optimálisan játszik.

A kiértékelő függvény megválasztása nagy szerepet játszik a módszer erősségében. Ha nem jól határozzuk meg, hogy egy állapot melyik játékosnak mennyire kedvező, akkor az algoritmusunk rossz döntéseket fog hozni, mivel csupán a kiértékelő függvény értékeit használja fel, a játékállapotokról egyéb információkat nem kap. Éppen ezért fontos, hogy a módszer használatakor minden játékállapot megfelelő értéket kapjon [2, 3].

Alfa-béta vágás

Az alfa-béta vágás (alpha-beta pruning) egy olyan módosítása a minimax algoritmusnak, amely jelentősen csökkenti a számítási időt bizonyos esetekben. Ezáltal lehetővé teszi, hogy az algoritmust nagyobb mélységre alkalmazzuk hasonló számítási idő mellett. A módszer alapvető ötlete az, hogy a fában figyelmen kívül hagy néhány ágat, melyeknek az értékei nem módosítják a végső eredményt.

Az algoritmus két változót tárol el a fa bejárása közben, ezek az alfa és a béta. Alfa azt a minimális értéket jelenti, amit a maximalizáló játékos már biztosan el tud érni, béta pedig azt a maximális értéket, amit a minimalizáló játékos tud már biztosan elérni. Az alfa változó kezdeti értéke $-\infty$, a béta változóé ∞. Ha a fa bejárása közben a maximalizáló játékos döntései közben találunk egy értéket, ami nagyobb mint alfa, akkor alfát erre az értékre változtatjuk, mivel ezt az értéket már biztosan el tudja érni. Ugyanígy, ha a minimalizáló játékos talál egy értéket, ami kisebb mint béta, akkor bétát erre az értékre változtatjuk. Ha egy állapotban alfa értéke nagyobb vagy egyenlő bétával, akkor a jelenlegi ág kiértékelését átugorja az algoritmus, ezzel számítási időt megtakarítva [3].

2.2. Felügyelt tanulás

A gépi tanulási módszereket három fő kategóriába soroljuk, ezek a felügyelt tanulás (supervised learning), a felügyelet nélküli tanulás (unsupervised learning) és a megerősítéses

tanulás (reinforcement learning). Ezek közül a felügyelt tanulást és a megerősítéses tanulást fogom bemutatni, mivel ezeket használtam a megoldásaimban.

A felügyelt tanulás feladata az, hogy nagy mennyiségű cimkézett (labeled) adat alapján előállítson egy függvényt, melyet később arra használhatunk, hogy címkézetlen adatokhoz értéket rendeljünk. Ezt precízebben megfogalmazva, adott (x, y) értékpárok egy halmaza, és a feladat az, hogy meghatározzunk modellek valamilyen családjából egy f függvényt, úgy, hogy az $f(x)$ értékek minél nagyobb pontossággal közelítsék az y értékeket. Ilyen modellcsalád például a döntési fák, a neurális hálók, vagy a lineáris regresszió. A módszereim megvalósításához neurális hálókat használtam.

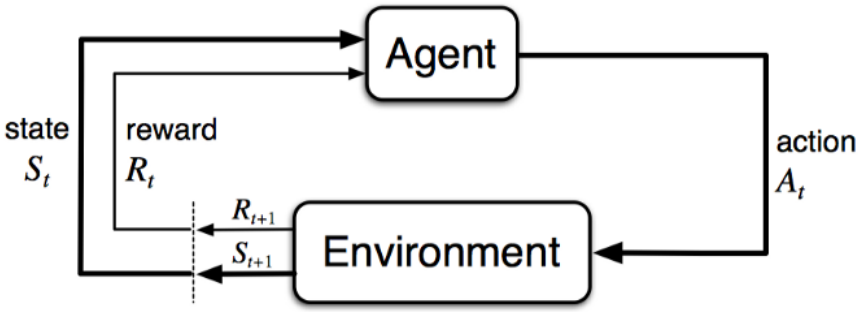
Az x értékeket leíró változónak, az y értékeket pedig célváltozónak szoktuk nevezeni. A felügyelt tanulást két kategóriára oszthatjuk aszerint, hogy a célváltozókat milyen halmazból vesszük, ezek a klasszifikáció és a regresszió. Klasszifikáció esetén a célváltozók kategóriák egy diszkrét halmazából származnak, feladatunk az, hogy a leíró változókról megállapítsuk, hogy a kategóriák közül melyikbe tartoznak. Regresszió esetén a célváltozók a valós számok egy folytonos részhalmazából származnak, feladatunk, hogy a leíró változókhoz olyan értékeket rendeljünk, amik a lehető legkisebb mértékben térnek el a célváltozók értékeitől.

A felügyelt tanulásban használt modellek részleteinek a [3, 4] irodalomban lehet utánanézni.

2.3. Megerősítéses tanulás

A megerősítéses tanulás alapvető elve, hogy egy úgynevezett ágens egy adott környezettel kommunikál, olyan módon, hogy diszkrét időközönként akciókat hajt végre ebben a környezetben, és ezért jutalmakat kap visszajelzésként (2.3.1. ábra). Ezek az akciók az ágenst egy új állapotba viszik át.

A megerősítéses tanulás nagy mértékben hasonló ahhoz, ahogy az emberek tanulnak. A cselekedeteinkre visszajelzést kapunk a környezetünktől, majd a visszajelzések alapján megváltoztatjuk, hogy a jövőben hogyan cselekszünk. Ehhez hasonlóan az ágens is a cselekedeteiért kapott jutalmak alapján tanul, vagyis folyamatosan változtatja magát olyan módon, hogy a lehető legtöbb jutalmat tudja összegyűjteni a cselekedeteivel. Ez a rendszer ideális játékok modellezésére, éppen ezért gyakran használják a megerősítéses tanulást ezen a területen.



2.3.1. ábra. Megerősítéses tanulás folyamata [5]

.

A felügyelt tanuláshoz hasonlóan a megerősítéses tanulás célja is egy függvény meghatározása, azonban a felügyelt tanulással ellentétben nem egy előre megadott adathalmazon tanul, hanem az adatokat a környezetével való interakcióból nyeri ki. Továbbá megerősítéses tanulásnál nincs egyértelműen meghatározva, hogy adott bemeneti értékhez milyen kimeneti értéket várunk, helyette a jutalmak határozzák meg, hogy egy hozzárendelés mennyire jó. A környezetben tett akciók megváltoztatják magát a környezetet és a benne tehető akciókat is, ezért az akciók nem értékelhetőek önmagukban, csak egy akciósorozat részeként. Éppen ezért egy adott akció nem mindig ugyanazt a jutalmat kapja.

A megerősítéses tanulás elméletének nagy része a jutalom hipotézisen (reward hypothesis) alapszik. A hipotézis azt mondja ki, hogy minden célra, amit az ágensünk el akar érni tekinhetünk úgy, mint a kapott jutalmak összege várható értékének maximalizálása.

A továbbiakban a Markov döntési folyamat nevű matematika modellt fogom használni környezetként.

2.1. Definíció. *Markov döntési folyamatnak, nevezzük a* (S, A, R, p, γ) *rendezett ötöst, ha* $p(s', r|s, a) = Pr[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a]$ *és* $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$ *a diszkontált jutalom, ahol:*

- S *az állapotok halmaza,* $S_t, S_{t+1} \in S$
- A *az akciók halmaza,* $A_t \in A$
- R *a jutalmak halmaza,* $R_t, R_{t+1} \in R$
- p *az úgynevezett átmeneti valószínűség*

- γ *a diszkontálás paramétere*

Tehát a Markov döntési folyamat meghatározza, hogy s állapotban a akciót végrehajtva mennyi az esélye annak, hogy s' állapotba kerülünk, illetve milyen r jutalmat kapunk. Ez egy elsőrendű Markov-modell (first order markov), vagyis a rendszer változása az aktuális állapot alapján kiszámítható, a régebbi állapotoknak nincs rá hatása. γ meghatározza, hogy a későbbi jutalmakat milyen mértékben vesszük figyelembe. Nagyobb γ érték nagyobb mértékben, $\gamma = 0$ esetén egyáltalán nem.

2.2. Definíció. *π politikának nevezzük egy adott s állapotban az akciók valószínűségének eloszlását.*

$$\pi(A_t = a|S_t = s)$$
$$\forall A_t \in A(s), S_t \in S$$

Amikor egy ágens egy politikát követ, akkor létrehoz állapotok, akciók és jutalmak egy sorozatát. Ezt a sorozatot nevezzük trajektóriának

Egy politikát meg lehet határozni paraméterek egy θ halmazával. Ilyen paraméterek például neurális hálók esetében a háló súlyai és eltolás értékei (bias). A megerősítéses tanulás feladata, hogy a θ paraméterek változtatásával megtalálja azt a π_θ politikát, aminek követése során az ágens által kapott jutalmak várható értéke a lehető legnagyobb. Ha a τ trajektóriáért kapott jutalmak összegét $r(\tau)$-val jelöljük, akkor ezt a feladatot a következő képpen írhatjuk le : maximalizáljuk a $J(\theta) = \mathbb{E}_{\pi_\theta}(r(\tau))$ várható értéket. A maximum megtalálására a gradiens módszert használhatjuk. A gradiens módszer egy függvény egy lokális maximumát úgy találja meg, hogy a függvény értékét a gradiense irányába tett lépésekkel változtatja. A jelenlegi esetben az okozza a nehézséget, hogy hogyan állapítsuk meg a J várható érték gradiensét a θ paraméterek függvényében. Erre a problémára ad megoldást a policy gradient módszer. Ennek részleteiről a [5] cikkben, illetve a [3] könyv 7. fejezetében lehet olvasni.

3. fejezet

Amőba

A továbbiakban két játékot vizsgálok, ezek az amőba és a sakk. A két játék kiválasztásának egyik szempontja az volt, hogy mindkettő determinisztikus, teljes információs és zéró összegű játék. Ezek reprezentációja kevésbé bonyolult, mint más típusú játékoké. További szempontom az volt, hogy szerettem volna egy összetettebb, és egy kevésbé összetett játékot választani. Az amőba a kettő közül a kevésbé összetett játék, a szabályai sokkal egyszerűbbek, mint a sakknak, illetve a játék 3×3-as variációjára a játékállapotok és a lépési lehetőségek száma is sokkal kisebb. Éppen ezért ezzel a játékkal kezdtem a módszerek tesztelését, mivel itt sokkal gyorsabban tudtam eredményeket elérni. Ezután szerettem volna egy összetettebb játékkal is megpróbálkozni, erre a sakkot választottam, mert széles körben ismert, számos bajnokságot rendeznek belőle, a játékban nem létezik egyértelműen ideális stratégia, ezért mindig benne rejlik a fejlődés lehetősége, illetve azért, mert szeretek sakkozni.

A bemutatásra kerülő módszerek számos különböző játékra alkalmazhatóak, azonban implementációjuk során rengeteg kérdésben kell döntéseket hozni. Ahhoz, hogy a számítógép a játékot bármilyen szinten is játszani tudja, először létre kell hoznunk egy környezetet, amelyből valamilyen módon információkat tud beolvasni, illetve az információk alapján lépéseket tud tenni a környezetben. Egy ilyen környezet lehet akár az is, melyben az emberek játszanak, vagyis egy játéktábla, melynek képét felvesszük egy kamerával, és ezt a felvételt adjuk bemenetként a programunknak. Azonban ebből a környezetből nagyon nehezen tudna információkat kinyerni a gép. Az egyik legjobb megközelítés a játék egy matematikai reprezentációja, melyben a játék minden állapotát, lépését és egyéb információit számoknak, vagy számok egy listájának feleltetjük meg. Attól függően, hogy

ezt a reprezentációt hogyan alkotjuk meg, az algoritmus bonyolultsága és hatékonysága sokat változhat. Továbbá a gépi tanulási módszerek alkalmazása során fontos, hogy a környezetünkben kis idő alatt nagy mennyiségű játékot le tudjunk futtatni, hogy megfelelő mennyiségű adatot tudjunk gyűjteni a tanuláshoz. Ezen kívül el kell dönteni, hogy a gépnek a játék szabályaiból mit adunk meg előre, és mi az, amit hagyunk, hogy a tanulási folyamat alatt sajátítson el. Ezek után az adott játékhoz és reprezentációhoz meg kell határoznunk az algoritmus paramétereit, mint például minimax esetén a fa mélysége, vagy a kiértékelő függvény, gépi tanulás esetén pedig a neurális háló méretei, illetve aktivációs függvényei, vagy a tanulás sebessége. Ezekre a problémákra nem létezik egyértelműen legjobb megoldás. A továbbiakban azt fogom bemutatni, hogy milyen megközelítésekkel próbálkoztam, és ezek közül melyik mennyire volt sikeres.

Az amőba egy népszerű kétszemélyes stratégiai társasjáték, melyet általában papíron tollal játszanak. A játék legegyszerűbb változatát egy 3×3-as négyzetrácson játszák. A játékosok felváltva illesztenek egy szimbólumot a négyzetrács egyik még nem foglalt mezőjére, egyik játékos egy x-et, a másik egy kört. A játékot az nyeri, aki előbb tud berajzolni 3-at a saját szimbólumából egy sorba, egy oszlopba, vagy egy átlóba (innentől amikor a sorokról írok, a sorok, oszlopok, és átlók összességére fogok hivatkozni). Ha a játékosok kifogynak a lehetséges lépésekből, akkor a játék döntetlennel ér véget. A játéknak ezen változatában elég gyorsan megtalálható az ideális stratégia, melyet ha mindkét játékos követ, a játék mindig döntetlennel végződik. Ennek ellenére ez egy megfelelő kezdőpont a módszerek kipróbálására. A játéknak léteznek bonyolultabb változatai is, melyekben a táblaméret nagyobb, vagy végtelen, illetve a győzelemhez 3-nál több szimbólumot kell elhelyezni egymás után megszakítás nélkül. Programjaimban a játék 3×3-as változata mellett nagyobb táblaméreteket is vizsgáltam. Nagyobb táblákra azt a változatott néztem, melyben egymás után 4 egyforma szimbólumot kell elhelyezni a győzelemhez. A játék végtelen nagy táblán játszott változatát nem vizsgáltam.

A programok mind a Python 3.7 verziójában készültek Jupyter notebook környezetben. A gépi tanulás megvalósítására a TensorFlow könyvtár 2.3.0 verzióját és a Keras könyvtár 2.4.0 verzióját használtam fel. A méréseket a saját számítógépemen futtattam, amely egy 6 magos 3,6 GHz órajelű AMD Ryzen 5 3600 processzort, egy AMD Radeon R9 380 videókártyát, és két darab 8 GB-os 3200 MHz sebességű Kingston HyperX Fury DDR4 memóriát tartalmaz.

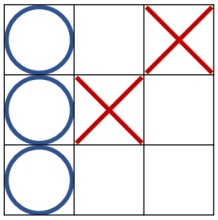
A játék reprezentációját, illetve az algoritmusokat a [3] 7. fejezete alapján készítettem

el, majd ezen módosításokat végeztem, hogy különböző eseteket is meg tudjak vizsgálni.

3.1. Az amőba reprezentációja

A játék egy állását egy $n \times n$ méretű négyzetrács írja le, melyet egy $n \times n$ méretű mátrixszal reprezentáltam. A mátrix minden eleme a négyzetrács egy mezőjének felel meg, értéke 1, ha a kezdő játékos szimbóluma van ott, -1, ha a második játékosé és 0, ha egyik játékos se rakott még a mezőre (3.1.1. ábra).

```
[[ 1  0 -1]
 [ 1 -1  0]
 [ 1  0  0]]
```



3.1.1. ábra. Amőba egy állásának reprezentációja. Az állás látható jobbra, a mátrix reprezentáció balra

Egy lépés a játékban abból áll, hogy a soron lévő játékos a saját szimbólumát lerakja egy még szabadon lévő mezőre. Ezt egy (x, y) számpárral ábrázoltam, ahol x azt jelenti, hogy a mező hanyadik sorban van, y pedig azt, hogy hanyadik oszlopban.

A környezetben egy meccs lejátszásához két függvényt kell implementálnunk, az egyik a kezdő játékos lépéseiért felelős, a másik a második játékos lépéseiért. Egy ilyen függvény bemenetként egy játékállapot reprezentációt kap, vagyis egy $n \times n$-es mátrixot, kimenetként pedig egy (x, y) párt ad vissza, amely azt mondja meg, hogy milyen lépést tesz az állapotban. A játék menete a következő: kiindulunk egy üres táblából, meghívjuk rá az első függvényt. A kapott lépés alapján változtatjuk a játék állapotát. Ezután a kialakult állapotra meghívjuk a második függvényt, majd ennek a lépése alapján változtatjuk az állapotot. Ugyanígy a függvényeket felváltva hívjuk addig, amíg a játék végállapotba nem kerül.

Innentől a feladat különböző játékos függvények létrehozása. Az alábbi három függvényt azért hoztam létre, hogy később az algoritmusokat tanítani, illetve tesztelni tudjam a segítségükkel.

- **Véletlen játékos** : Az adott játékállapotban kilistázza az összes szabályos lépést, majd ezek közül véletlenszerűen választ egyet.

- **Szabálytalan véletlen játékos** : A játék összes mezőjéből véletlenszerűen rak egyikre, attól függetlenül, hogy az már foglalt-e vagy sem. Megeshet, hogy egy szabálytalan lépést tesz, vagyis egy olyan mezőre próbál rakni, ami már foglalt, ilyenkor az általam készített környezetben azonnal veszít.

- **Emberi játékos** : Az adott játékállapotban kilistázza az összes szabályos lépést, majd megkéri a felhasználót, hogy válasszon közülük egyet, és ezt adja vissza kimenetként. Ez ad módot arra, hogy a környezetben ember is tudjon játszani.

3.2. Minimax algoritmus alkalmazása amőbára

Programjaimban a minimax algoritmust alfa-béta vágással használtam, mivel ez minden esetben jelentősen gyorsabbnak bizonyult.

Kiértékelő függvénynek 3×3-as táblán a következőt használtam. Ha egy állapotban a maximalizáló játékos nyert, akkor az érték 10000, ha a minimalizáló játékos nyert, az érték -10000, ha a végeredmény döntetlen, az érték 0. Ha az állapotban még nincs vége a játéknak, akkor azon sorok számából, melyekben két darab 1-es és egy 0 található, kivonja azon sorok számát, melyekben két darab -1-es és egy 0 van, és ezt az értéket adja vissza a függvény. Tehát ha még nincs vége a játéknak, a függvény értékeli azt is, hogyha egy játékos közel áll a győzelemhez, azáltal, hogy egy sorba már csak egy szimbólumot kell elhelyeznie. Nagyobb táblára ezen annyit változtattam, hogy abban az esetben, amikor nincs vége a játéknak, olyan sorokat keres, melyekben négy egymás melletti helyből háromban ugyanaz a szimbólum, és egy üres, mivel itt négy szimbólum kell a győzelemhez.

Az algoritmust először a maximális mélység megadása nélkül, vagyis az egész állapottér átvizsgálásával próbáltam ki. Az algoritmusnak ez a lehető legjobban játszó változata, azonban ehhez kell a legtöbb számítási idő is. Először a véletlen játékos ellen teszteltem. Kezdő játékosként 1000 játékból 994-ben nyert, 6-ban játszott döntetlent, és egyszer sem vesztett, másodikként kezdve pedig 1000 játékból 812-ben nyert, 188-ban játszott döntetlent, és szintén nem vesztett egyszer sem. A minimax algoritmusnak ezt a változatát nem meglepő, hogy nem lehet legyőzni, mivel az egész állapotteret át tudja vizsgálni, ezért sose választ olyan lépést, amely vereséghez vezetne. Megfigyelhető, hogy ha a minimax kezdő játékos, akkor sokkal kevesebb döntetlen lesz, mint akkor, amikor második játékos.

Ennek az a magyarázata, hogy ebben a játékban a kezdő játékosnak előnye van. Ez egyértelműen látszik abból, hogy ha két véletlen játékos játszik egymás ellen, akkor a kezdő játékos $58,60\%$ eséllyel nyer, a második játékos $28,74\%$ eséllyel nyer, a döntetlennek pedig $12,66\%$ az esélye 100000 lejátszott játék alapján.

A futási időket a véletlen játékos ellen játszott mérkőzéseken néztem, mivel ennek a számítási ideje elhanyagolható a minimax algoritmus számítási idejéhez képest. 3×3-as táblára átlagosan a játékidő 0,2 másodperc volt, 4×4-es táblára azonban már 1 perc alatt se tudott egy lépést se megtenni a minimax játékos, ezért ebben a formában az algoritmus nem használható 3×3-nál nagyobb táblákra. Ezekre az esetekre az algoritmusnak meg kell adni maximális mélységet. A 3.2.1. ábrán láthatóak a futási idők adott táblaméretre és maximális mélyégre. Az értékek minden esetben 10 játék átlagaként jöttek ki, melyekben a minimax játékos kezdett. Azokat az eseteket, melyekre nem tudott 1 perc alatt egy játékot se lejátszani, x-el jelöltem.

Játék hossza (másodperc)	*Maximális mélység*								
Tábla mérete	1	2	3	4	5	6	7	8	9
4x4	0,002	0,004	0,025	0,052	0,488	0,605	7,497	6,994	x
5x5	0,004	0,008	0,010	0,232	3,415	4,804	x	x	x
6x6	0,006	0,012	0,233	0,480	6,718	11,440	x	x	x
7x7	0,009	0,031	0,665	1,213	28,301	36,687	x	x	x
8x8	0,015	0,033	1,081	1,695	x	x	x	x	x
9x9	0,021	0,066	2,594	7,379	x	x	x	x	x

3.2.1. ábra. Egy játék átlagos hossza (másodpercben) véletlen játékos ellen, a tábla mérete és a maximális mélység függvényében

Az eredmények alapján megfigyelhető, hogy már a táblaméret kis méretű növelése is hatalmas emelkedést okoz a számítási időben. Ennek oka az, hogy egy $n \times n$ méretű tábla állapotterének fájában mindegyik szint csúcsainak száma maximum az előző szint csúcsai számának n^2-szerese lehet. Éppen ezért M maximális mélységre a legalsó átnézett szint csúcsai számának maximuma $(n^2)^M$.

Egy $n \times n$ méretű táblára az állapottér fájának mélysége n^2. A futási idő miatt már a vizsgált táblaméretekre is ennek csak egy kis részét tudjuk átvizsgálni egy körben (pél-

dául 9×9-es táblára a 81 szintből 4-et), nagyobb táblákra pedig ez az arány egyre kisebb. Azonban ez nem jelenti azt, hogy a minimax játékos erőssége jelentősen gyengébb lesz, mintha az egész állapotteret át tudná nézni, hiszen az emberek is általában csak pár lépéssel előre képesek gondolkodni, és mégis viszonylag magas szinten tudják játszani az amőbát. Annak vizsgálatára, hogy az algoritmus hogy teljesít kisebb maximális mélységekre, 4×4-es táblára vizsgáltam a győzelmi arányát 1000 játékból a véletlen játékos ellen. A kezdő játékosként elért győzelmi arányok a 3.2.2. ábrán láthatóak, a második játékosként elért győzelmi arányok pedig a 3.2.3. ábrán.

	Maximális mélység			
	1	2	3	4
Győzelem	90,2%	84,2%	85,1%	86,5%
Vereség	1,3%	0%	0,4%	0%
Döntetlen	8,5%	15,8%	14,5%	13,5%

3.2.2. ábra. Minimax játékos győzelmi arányai kezdő játékosként véletlen játékos ellen, 4×4-es táblára

	Maximális mélység			
	1	2	3	4
Győzelem	74,8%	69,7%	65,5%	73,4%
Vereség	3%	0,1%	1,5%	0%
Döntetlen	22,2%	30,2%	33,3%	26,6%

3.2.3. ábra. Minimax játékos győzelmi arányai második játékosként véletlen játékos ellen, 4×4-es táblára

A kis maximális mélység ellenére jól teljesít az algoritmus. Az 1 maximális mélység esetét leszámítva elenyésző a vereségek száma. Annak ellenére hogy 1 maximális mélység

esetén a legnagyobb a veszteségek száma, ilyenkor a legnagyobb a győzelmek száma is. Ennek oka az, hogy ekkor az algoritmus az ellenfél potenciális lépéseit egyáltalán nem veszi figyelembe, hanem minden esetben csak a saját következő lépése eredményét nézi. Éppen ezért nem próbálja megakadályozni az ellenfél győzelmét, kizárólag az a célja, hogy ő megnyerje a játékot, éppen ezért a döntetlenek száma jóval kisebb. Minden maximális mélységre a döntetlen meccsek száma jelentősen nagyobb mint, a 3×3-as táblán egész állapotteret átnéző minimax változat esetében, ez annak köszönhető, hogy 4×4-es táblán sokkal nagyobb az esélye a döntetlennek. Ha két véletlen játékos játszik egymás ellen, méréseim alapján 3×3-as táblán a döntetlen esélye $12,66\%$, míg 4×4-es táblán $41,52\%$.

Megvizsgálhatjuk továbbá a kisebb maximális mélységű változatok erősségét úgy is, hogy a teljes állapotteret átnéző változat ellen játszatjuk őket 3×3-as táblán. A minimax algoritmus determinisztikus, ezért két változatát akárhányszor játszatjuk egymás ellen, minden alkalommal ugyanaz lesz az eredmény. Éppen ezért itt nem győzelmi arányokat nézünk, hanem azt, hogy mi a végeredménye a meccsnek. Ha a megadott maximális mélységű játékos kezd, a végeredmény minden esetben döntetlen lesz. Ha másodikként játszik, akkor az 1 és a 2 maximális mélységű veszít, viszont minden más meccs döntetlen lesz. Ez is bizonyítja tehát, hogy a kis maximális mélységű minimax játékosok nem jelentősen gyengébbek, mint az egész állapotteret átnéző változat.

3.3. Megerősítéses tanulás alkalmazása amőbára

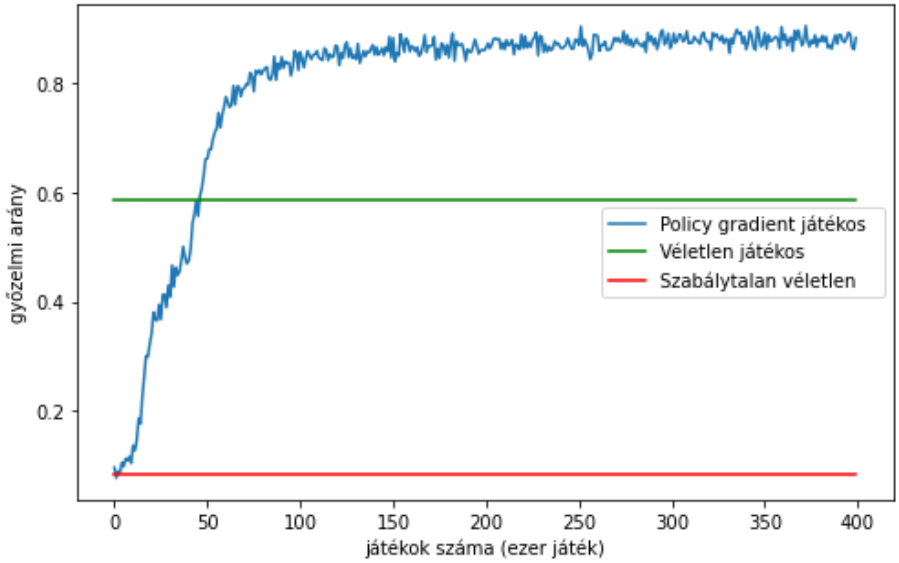
A megerősítéses tanulás megvalósításához előre kapcsolt neurális hálókat használtam. A különböző méretű táblákhoz mind különböző halókra van szükség, egy neurális háló nem tud több különböző méretű táblán játszani. Az $n \times n$ méretű táblához felhasznált neurális háló egy n^2 neuron nagyságú bemeneti réteget, három darab 100 neuron nagyságú rejtett réteget és egy n^2 neuron nagyságú kimeneti réteget tartalmaz. Az egymást követő rétegek mind teljesen kapcsoltak. A rejtett rétegekben ReLU, a kimeneti rétegben pedig softmax aktivációs függvényt használtam. A bemeneti réteg n^2 neuronja a tábla n^2 mezőjének felel meg, mindegyik 1, 0, vagy -1 értéket vesz fel aszerint, hogy az adott játékállapotban melyik játékos szimbóluma van az adott mezőn. A kimeneti réteg neuronjai szintén a tábla mezőinek felelnek meg, a softmax függvény miatt mindegyik értéke 0 és 1 közötti, összegük pedig 1. Éppen ezért tekinthetünk a kimeneti rétegre valószínűségi eloszlásként. Mindegyik neuron megmondja, hogy mekkora valószínűséggel választjuk a hozzá tartozó

mezőt következő lépésként. A neurális háló tehát a játék minden állapotához egy, az összes lépés halmazán értelmezett valószínűségi eloszlást rendel. Amikor ezt a neurális hálót egy meccs lejátszására használjuk, minden lépésnél kiszámoljuk az adott állapotra a valószínűségi eloszlást, majd ebből véletlen mintavétellel kiválasztunk egy lépést. Ennek a megközelítésnek a legnagyobb problémája az, hogy minden állapotban az összes lépés közül választ, nem pedig csak az adott állapotban megengedettek közül, ezért választhat szabálytalan lépéseket is. A játék szabályainak ezt a részét tehát nem adjuk meg előre, hanem a tanulási folyamat alatt kell rájönnie.

A neurális hálókat a policy gradient módszerrel tanítottam. Az ezen neurális hálók alapján játszó játékost innentől policy gradient játékosnak hívom. A tanulási folyamat abból áll, hogy a policy gradient játékos a véletlen játékos ellen játszik nagy mennyiségű meccset, és közben a kapott jutalmak alapján a neurális háló súlyai, illetve eltolás értékei (bias) változnak. A neurális háló kezdeti súlyait csonkolt standard normális eloszlásból vettem, az eltolási értékeket pedig kezdetben mind 0,1-ről indultak. A tanulás során a háló paramétereit 100 játékonként frissítettem az addig kapott jutalmak alapján. Egy lépéshez tartozó jutalom 1, ha az adott játékot megnyerte, -1 ha elvesztette, illetve 0 ha döntetlen lett a végeredmény.

A tanulást először 3×3-as táblára néztem. A 3.3.1. ábrán látható a tanulási folyamat 400000 lejátszott játékra. A kék görbe a policy gradient játékos győzelmi aránya a tanulási folyamat során, minden pontja az elmúlt 1000 játék győzelmi arányát jelenti. A zöld és piros egyenesek rendre a véletlen játékos és a szabálytalan véletlen játékos győzelmi arányát mutatják véletlen játékos ellen. Ezek a tanulási folyamat során nem változnak, az ábrán csak összehasonlítás céljából szerepelnek. A policy gradient játékos a tanulási folyamat elején a szabálytalan véletlen játékos erősségi szintjéről indul, mivel hozzá hasonlóan az összes lépés közül választ, nem csak az adott állapotban szabályos lépések listájából. A véletlen játékos szintjét nagyjából 40000 játék után éri el. A győzelmi arány növekedésének sebessége egy idő után jelentősen csökken. További tanulással már csak kis mértékben tudnánk innen fejleszteni a játékost.

A tanulási folyamat végére véletlen játékos ellen $88,43\%$-os győzelmi arányt ér el. Azonban vereségeinek nagy része annak köszönhető, hogy a valószínűségi eloszlásból egy nem optimális lépést választunk ki. A tanulás után módosíthatjuk úgy a játékost, hogy az eloszlásból nem mintavételezéssel választjuk ki a lépést, hanem mindig a legnagyobb valószínűséggel rendelkező lépést választjuk, mivel a neurális háló ezt tartja ideálisnak. Az



3.3.1. ábra. Győzelmi arányok a tanulási folyamat során véletlen játékos ellen.

így módosított játékos győzelmi aránya $97,12\%$, ami jelentős fejlődés. Ebben az álapotban a vereségek többsége abból származik, hogy a játékos az adott állapotban szabálytalan lépést tesz, vagyis olyan helyre próbál rakni, ami már foglalt. Ez könnyen elkerülhető, ha módosítjuk a játékost úgy, hogy ha egy szabálytalan lépést akarna tenni, akkor helyette az eloszlásból addig vesszük a következő legnagyobb elemet, amíg nem kapunk egy szabályos lépést. Ezzel a módosítással $97,92\%$-os győzelmi arányt tudunk elérni (3.3.2. ábra). 3×3-as táblára az ideális játékos a minimax algoritmus, ennek győzelmi aránya $99,94\%$ a véletlen játékos ellen. Ezt a szintet ugyan nem éri el a policy gradient játékos, de közel áll hozzá. Azonban ha egy ember néhány meccset játszik a policy gradient játékos ellen, elég hamar ki tudja ismerni a hibáit, és onnantól kezdve minden játékban le tudja győzni. Ennek oka, hogy az amőba 3×3-as táblára az ember számára egy túl egyszerű játék, ezért ha bármilyen hibát is vét a program, akkor egy ember könnyen le tudja győzni.

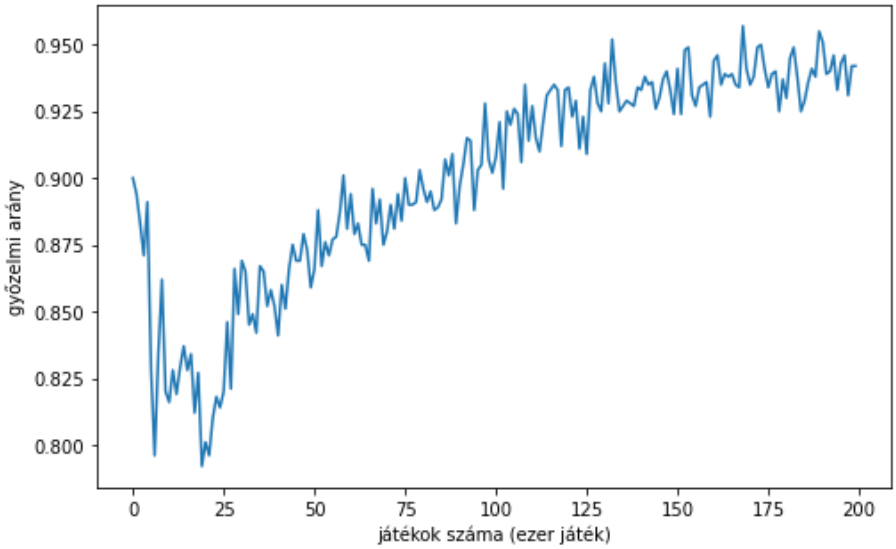
Az egyik oka, hogy a policy gradient játékos ennél a szintnél már nem tudott tovább tanulni az, hogy a véletlen játékos ellen játszott csak, ami egy nagyon gyenge ellenfél. A továbbiakban erősebb játékosok ellen tanítottam.

Az egyik lehetséges játékos, aki ellen taníthatnánk, az önmaga. Elmentjük a neurális háló jelenlegi állapotát, majd ez ellen a változat ellen tanítjuk tovább. Az elmentett változatot a folyamat közben nem változtatjuk. Ezzel azonban az a probléma, hogy mivel a policy gradient játékos kezdő játékosként tanult, ezért második játékosként egyáltalán

	Véletlen minta	Legvalószínűbb lépés	Legvalószínűbb lépés, csak legálisak
Győzelem	88,66%	96,89%	97,92%
Vereség	11,3%	3,11%	1,29%
Döntetlen	0,04%	0%	0,79%

3.3.2. ábra. Policy gradient játékos különböző változatainak győzelmi arányai kezdő játékosként véletlen játékos ellen

nem tud játszani. Éppen ezért egy teljesen új hálót tanítottam, aki második játékosként játszott 400000 játékot véletlen játékos ellen. Ezt a játékost innentől mentett játékosnak nevezem. A tanulás után a mentett játékos győzelmi aránya véletlen játékos ellen második játékosként $67,36\%$ akkor, ha hagyjuk neki, hogy szabálytalan lépéseket válasszon, illetve a lépéseket az eloszlásból vett mintával határozzuk meg. Ezután a policy gradient játékost a mentett játékos ellen tanítottam 200000 mérkőzésen keresztül. A tanulási folyamat a 3.3.3. ábrán látható.

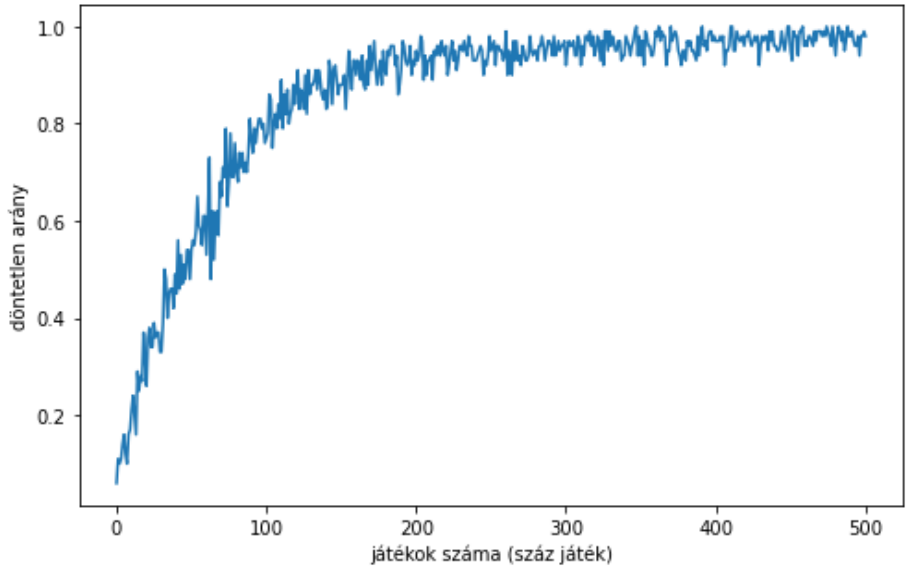


3.3.3. ábra. Policy gradient játékos győzelmi arányai a mentett játékos elleni továbbtanulás során

Meglepő módon a policy gradient játékos 90%-os győzelmi arányról indult, ami jobb, mint a véletlen játékos elleni győzelmi aránya. Ez azt mutatja, hogy ez az ellenfél se feltétlenül jobb a tanuláshoz, mint a véletlen játékos. A tanulás végére $94,2\%$-os győzelmi arányt ér el a policy gradient játékos, tehát teljesítménye egyértelműen javult. Azonban a továbbtanulás után a policy gradient játékos azon változatának, melynek nem engedünk meg szabálytalan lépéseket, illetve mindig a legjobb lépést választjuk, a győzelmi aránya kezdő játékosként véletlen játékos ellen $97,48\%$, amely $0,44\%$-al rosszabb a továbbtanulás előtti állapotnál. Ez azt mutatja, hogy a továbbtanulás során rátanult a mentett játékos hibáinak kihasználására, azonban más játékosok ellen nem feltétlenül lett erősebb. Két játékos erősségét nem tudjuk egyértelműen összehasonlítani, mivel különböző ellenfelek ellen különböző győzelmi arányokat érhetnek el. Éppen ezért nem lehet egyértelműen megmondani, hogy a policy gradient továbbtanulás előtti vagy utáni változata az erősebb.

A másik ellenfél, aki ellen végezhetjük a továbbtanítást, az a minimax játékos. A minimax játékosnak a 3 maximális mélységű változatát használtam, mert különben nagyon lassú lett volna a tanulási folyamat. Mivel ezt a játékost nem lehetséges legyőzni, ezért ebben az esetben a győzelmi arány helyett a döntetlenek arányát mértem. A jutalmat is ennek megfelelően módosítottam 0-ról 1-re döntetlen esetén. Ebben a formában a tanulás alatt a döntetlenek aránya végig 0% és $0,1\%$ között mozgott, vagyis nem történt fejlődés. Ezért változtattam a játékoson úgy, hogy már a tanulási folyamat során se engedtem meg neki a szabálytalan lépéseket, ezáltal fejlődést tudott elérni. A tanulási folyamat a 3.3.4. ábrán látható.

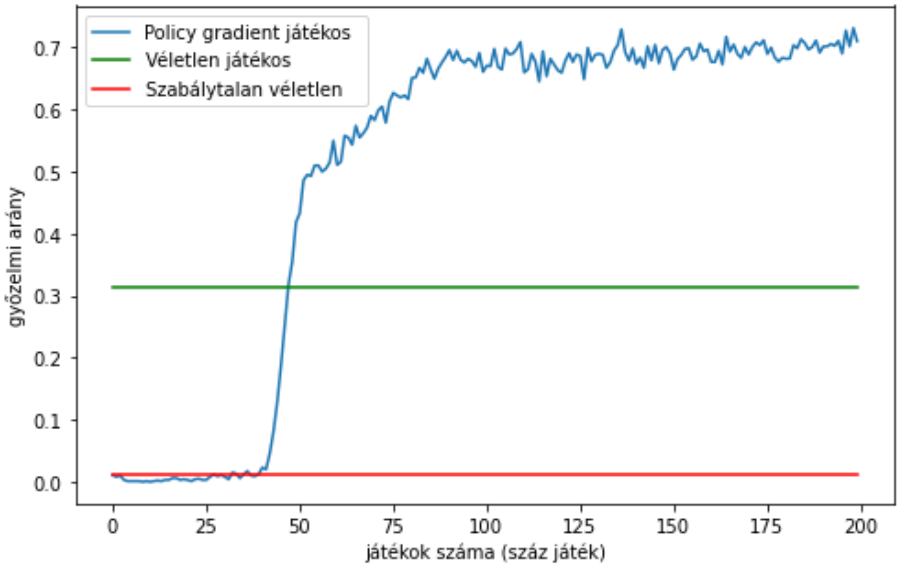
A tanulás végére a döntetlenek aránya $98,3\%$, ha pedig a policy gradient játékos az eloszlásból mindig a legjobb lépést választja, akkor minden játékban döntetlen lesz a végeredmény. Azonban ekkor a policy gradient játékos győzelmi aránya véletlen játékos ellen csak $90,7\%$, vagyis $7,22\%$-al rosszabb, mint a továbbtanulás előtt. Ennek oka az, hogy a minimax algoritmus determinisztikus, vagyis minden alkalommal azonos állásban ugyanazt a lépést teszi. Így a tanulás során rengeteg állapot nem fordult elő, ezeket az állapotokat a policy gradient játékos "elfelejtette" hogyan kell kezelni, vagyis a háló paraméterei úgy változtak, hogy ezekre az állapotokra már más döntéseket hozott. Ennek bemutatására nézzük a következő példát. A policy gradient játékos első lépésben a középső mezőre rak. A minimax játékos ekkor minden alkalommal a bal felső sarokba rak. Ebből az állapotból a policy gradient játékos tökéletesen játszik és sikerül a döntetlent elérnie. Azonban, ha a bal felső sarok helyett ellenfele a jobb felső sarokba rak, akkor már nem tudja, hogyan kell tökéletesen játszani, mivel ez nem fordult elő a továbbtanulás során.



3.3.4. ábra. Döntetlenek arányai a minimax elleni továbbtanulás során

Ezután a módszert nagyobb táblákra alkalmaztam. A neurális háló rejtett rétegeinek neuron számát 100-ról 300-ra emeltem, hogy jobban tudja kezelni a megnövekedett bemeneti és kimeneti neuronok mennyiségét. A tanulási folyamat során minden esetben a policy gradient kezdő játékosként 200000 mérkőzést játszott le a véletlen játékos ellen. A 3×3-as táblával ellentétben nagyobb táblákra a tanulási folyamat nem volt egyből eredményes. A 4×4-es, illetve 5×5-ös táblán a győzelmi arány egy ideig stagnált mielőtt hirtelen javulásba kezdett. Ez megfigyelhető a 3.3.5. ábrán, amely a 4×4-es táblára mutatja be a tanulási folyamatot.

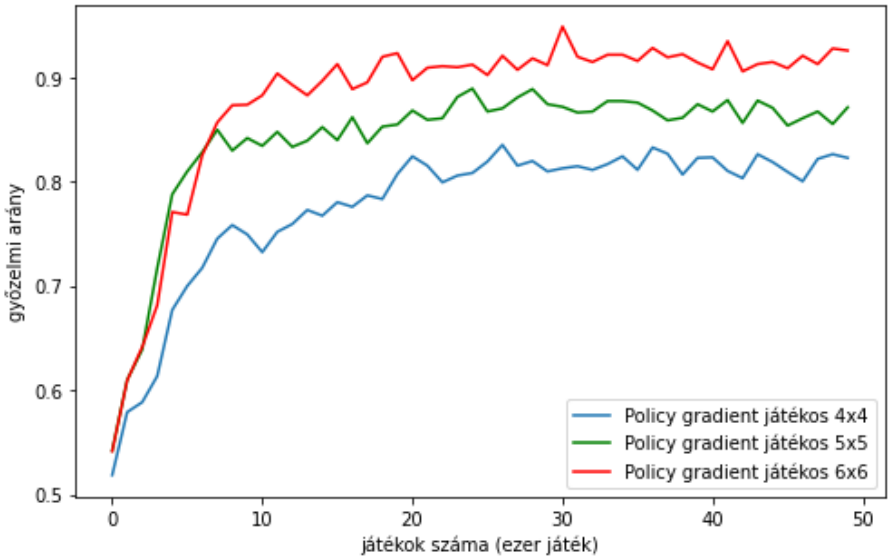
6×6-os táblára azonban ezt a stagnálási fázist nem tudja elhagyni, győzelmi aránya sose haladja meg az 1%-ot. Ennek oka, hogy egy olyan állapotból indul ki a neurális háló, ami túl gyenge, ezért a játékok elenyésző százalékát tudja csak megnyerni. Ebből az állapotból nem tud kijutni, mivel nem kap elég mennyiségű pozitív jutalmat, hogy tudja hogyan kell változnia a hálónak ahhoz, hogy játékokat tudjon nyerni. A 4×4-es, illetve 5×5-ös esetre szerencsés módon sikerült egymás után elég játékot megnyernie, hogy ebből az állapotból kilendüljön, azonban ahogy a tábla méretét növeljük, ennek a szerencsés kilendülésnek az esélye egyre csökken. Ennek a jelenségnek az elkerülése érdekében újra próbáltam a tanulást úgy, hogy közben nem engedtem meg a policy gradient játékosnak a szabálytalan lépések választását. Ez jelentősen növelte a számítási időt, ezért a tanulás hosszát is változtattam 50000 játékra. A 3.3.6. ábrán látszanak a tanulási folyamatok



3.3.5. ábra. Győzelmi arányok a tanulási folyamat során véletlen játékos ellen (4×4-es tábla)

az eddig említett három táblaméretre, a 3.3.7. ábrán pedig a három létrehozott játékos, valószínűségi eloszlásból legjobb lépést választó változatainak győzelmi arányai.

A módszer tehát kisebb módosításokkal használható 3×3-nál nagyobb táblákra is, azonban ezekre az esetekre a győzelmi arányok jelentősen kisebbek, mint a 3×3-as esetben, a tanulás pedig a táblamérettel arányosan egyre több számítási időt vesz igénybe.



3.3.6. ábra. Győzelmi arányok véletlen játékos ellen három különböző táblaméreten való tanulás során (szabálytalan lépések nincsenek engedélyezve)

	4x4	5x5	6x6
Győzelem	74,16%	85,76%	92,8%
Vereség	9,23%	11,22%	7,2%
Döntetlen	16,61%	3,02%	0%

3.3.7. ábra. Policy gradient játékos győzelmi arányai véletlen játékos ellen három különböző táblaméreten

4. fejezet

Sakk

A sakk gépi tanulással való megközelítésére az elmúlt években rengeteg próbálkozás történt, ezért számos különböző módszerről lehet információkat szerezni, amely alapján az érdeklődő el tudja kezdeni az implementációit. Először az amőbán már bevált módszereket próbáltam ki, hogyan működnek sakkra, majd ezeket próbáltam javítani részben a saját ötleteim alapján, részben pedig mások által már kipróbált módszerek segítségével. A játéknak egy néhány helyen leegyszerűsített változatát hoztam létre környezetként, mert így a reprezentáció, a gépi tanulás, illetve a környezet létrehozása is valamivel egyszerűbb volt, illetve rengeteg számítási időt megspóroltam. Az általam létrehozott környezetben nincsen sáncolás, en passant lépés, a pálya másik végére átvitt gyalogokat nem lehet lecserélni más bábura, a játék nem érhet véget döntetlennel, illetve nem matt adásával fejeződik be, hanem az ellenfél királyának leütésével.

4.1. A sakk reprezentációja

A játék egy állapotának reprezentálása számos módon lehetséges. Ezek közül az alábbi hármat alkalmaztam.

- A játék mind a 12 különböző figurájához, illetve az üres helyhez hozzárendelünk egy különböző számot. Egy állapot reprezentációja egy 8×8 méretű mátrix, melynek minden eleme a táblán megfelelő helyén lévő bábuhoz rendelt szám (4.1.1. ábra).
- A játék minden figurájához egy úgynevezett one-hot vektort rendelünk. Egy ilyen vektor elemei 11 darab nulla, és 1 darab egyes, a 12 különböző lehetséges vektor mind

egy különböző figurának felel meg. Az üres helyeket egy csupa 0-ból álló vektor jelzi. Egy állapot reprezentációja egy 8×8 méretű mátrix, melynek minden eleme egy one-hot vektor.

- Egy játékállapotban maximum 32 figura lehet egyszerre a táblán. Mind a 32 figurához hozzárendelünk egy (x, y, z) vektort, ahol x azt jelenti, hogy a figura hanyadik sorban van, y azt, hogy hanyadik oszlopban, z értéke pedig 1, ha a a figura fent van a táblán, és 0, ha már nincs. Egy állapot reprezentációja egy 32×3 méretű mátrix, melyben az első 8 sor a fehér gyalogoknak felel meg, 9-es és 10-es sor a két fehér bástyának, és ennek mintájára a többi sor a többi figurának.

$$\begin{pmatrix} 8 & 9 & 10 & 11 & 12 & 10 & 9 & 8 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 3 & 4 & 5 & 6 & 4 & 3 & 2 \end{pmatrix}$$

4.1.1. ábra. A kezdeti tábla (jobbra) 1. reprezentációja (balra)

Az első két reprezentáció a tábla mezői, a harmadik pedig a figurák elhelyezkedése alapján adja meg a játékállapotot. Ez két teljesen eltérő megközelítés, ezért hatékonyságukban jelentős eltérés van. Egy reprezentáció hatékonyságában az egyik legfontosabb tényező az, hogy a játék két hasonló állapota között kicsi, két eltérő állapota között pedig nagy legyen a különbség a reprezentációjukban. A tárgyalás egyszerűségének érdekében a továbbiakban csak a második reprezentációval végzett méréseimet fogom bemutatni.

A játék egy lépését egy (x_1, y_1, x_2, y_2) vektorral lehet reprezentálni, ahol x_1 annak a mezőnek a sorszáma, ahonnan a lépést tesszük, y_1 ennek az oszlopszáma, x_2 annak a mezőnek a sorszáma, ahova lépünk, y_2 pedig ennek az oszlopszáma. Másik lehetséges reprezentáció lenne egy figura szerinti megközelítés, például úgy, hogy a vektorban x_1 és y_1

helyett egy szám szerepel, amely annak a figurának felel meg, amelyik a lépést teszi. A két reprezentáció közül csak az előbbit használtam. Ennek a reprezentációnak a gyengesége az, hogy az algoritmusunknak $8^4 = 4096$ lehetséges lépés közül kell választania, minden egyes körben, amely jóval nagyobb az egy állapotban szabályos lépések számánál, ezért már a szabályos lépések megtanulása is komoly kihívást jelent a gépi tanulás során.

Az amőbához hasonlóan itt is a játékosokat olyan függvények reprezentálják, melyek egy játékállapothoz egy lépést rendelnek. A 3.1. fejezetben bemutatott 3 függvényt használom itt is a módszerek tesztelésére.

4.2. Minimax algoritmus alkalmazása sakkra

A legegyszerűbb kiértékelő függvény sakkban a figurák értékének összegzése. A legtöbb ember is ez alapján hoz döntéseket a játék során. Mindegyik figurához rendelünk egy értéket, a gyalog 1-et ér, a huszár és a futó 3-at, a bástya 5-öt, a vezér 9-et, a király pedig 100-at, így a király leütése mindig értékesebb lesz bármilyen más döntésnél. Az adott állapotban a táblán még fent lévő figuráink értékeit összeadjuk, az ellenfél figuráinak értékeit pedig ebből kivonjuk, az így kapott számot rendeli a függvény az állapothoz. Bonyolultabb kiértékelő függvények nem csak a táblán lévő figurák értékét veszik figyelembe, hanem azok elhelyezkedését is, így például hozzáadnak az értékhez, ha a figuráink védve vannak, vagy a tábla közepén helyezkednek el, és levonnak az értékből, ha a figuráink blokkolva vannak, vagy ha már nem tudunk sáncolni. A számítási idő minimalizálásának érdekében a figurák értékét adtam csak össze, a pozíciót nem pontoztam.

A sakk állapottere túl nagy ahhoz, hogy a minimax algoritmus át tudja nézni az egészet, ráadásul az általam létrehozott változatban a játék nem ér véget, ha 3-szor ugyanabba az állapotba kerülünk, éppen ezért az állapottér végtelen. A minimax algoritmust ezért csak maximális mélység megadása mellett használhatjuk. A játékok hossza a sakkban nagy mértékben eltérő lehet, és minden állapotban eltérő számú lehetséges lépés van, ezért a futási időt nem teljes játékok lejátszásával teszteltem, hanem a játék kezdőállapotában néztem meg, mennyi idő alatt tud döntést hozni az algoritmus (4.2.1. ábra).

A játék későbbi szakaszában általában több lehetséges lépés van, mint az elején, ezért később ezek az idők is nagyobbak. A maximális mélység növelésével a futási idő rendkívüli mértékben növekszik, ezért 5-nél nagyobb mélységre már nem igazán használható

	Maximális mélység					
	1	2	3	4	5	6
Futási idő (másodperc)	0,002	0,007	0,055	0,154	3,636	64,117

4.2.1. ábra. A minimax algoritmus döntési ideje a kezdeti állapotban, különböző maximális mélységek mellett

az algoritmus játékok lejátszására. A győzelmi arányok leméréséhez 4 maximális mélységig mindegyik változat játszott 100 játékot kezdő játékosként a véletlen játékos ellen, és mindegyik megnyerte mind a 100 játékot. Sakkban a véletlen játékos túl gyenge, ezért ellene nem tudjuk mérni a módszerek győzelmi arányát. Azonban meg lehet állapítani a segítségével, hogy a játékosok hány kör alatt tudják megnyerni a játékot. A 100 mérkőzésből az átlagos játékidő 1 maximális mélység mellett 30 kör volt, 2 maximális mélység mellett 42 kör, 3 maximális mélység mellett 34 kör, 4 maximális mélység mellett 32 kör. Látható, hogy az 1 maximális mélységű változat tud leggyorsabban nyerni, ennek oka az, hogy ez a leginkább agresszív játékos, mivel csak a következő lépésének eredményét nézi, az ellenfél válaszát nem veszi figyelembe. Ez a véletlen játékos ellen jól működő stratégia, azonban erősebb játékosok ellen már nem. Tehát nagyobb maximális mélység mellett nem feltétlenül nyer gyorsabban a minimax játékos.

A minimax algoritmus ellen személyesen játszott mérkőzéseim során az algoritmust viszonylag nehéz volt legyőzni, mivel általában a neki kedvezőtlen cserék elől kitért. Azonban két hibát meg lehet figyelni benne. Egyrészt a játék elején nagyon rosszul nyit, mivel nem lát elég lépéssel előre ahhoz, hogy tudja, melyik kezdés mennyire erős. Mivel az összes kezdést ugyanúgy értékeli a függvény, ezért a lehetséges lépések listájából az elsőt választja mindig, vagyis az algoritmusom esetében a bal oldali gyalogot egyesével lépteti előre, amíg nem lát ennél jobb lehetőséget. Másik hibája a maximális mélység miatti levágásból adódik. Bármilyen maximális mélységgel is használjuk az algoritmust, lesz a fának egy szintje, aminél nem lát tovább. Ha a fának ezen a szintjén lát egy nagyon jó lépést, akkor annak az irányába fog elindulni, annak ellenére, hogy lehetséges, hogy a következő lépésben az ellenfélnek emiatt egy még jobb lépése lesz. Például ha 3 maximális mélységre alkalmazzuk az algoritmust, és a 3. lépésben le tud ütni a vezérével egy bástyát, akkor ezt az irányt fogja választani annak ellenére, hogy a bástya védve van, és az ellenfél a

következő körben leüti a vezérét. Ennek a következménye, hogy az algoritmus páratlan maximális mélység esetén agresszívabban játszik, mint páros mélység esetén, mivel páratlan mélységnél a saját lépését látja utoljára, páros mélység esetén pedig az ellenfélét.

4.3. Megerősítéses tanulás alkalmazása sakkra

Sakkra a megerősítéses tanulást a rétegek méretétől eltekintve ugyanazzal a neurális hálóval valósítottam meg, mint amit amőbára használtam. Ennek bemutatása a 3.3. fejezetben olvasható. A rejtett rétegek méretét 500-ra növeltem. A bemeneti réteg mérete $8 \cdot 8 \cdot 12 = 768$, amelynek a 64 darab 12 hosszú one-hot vektort adjuk bemenetként. A kimeneti réteg mérete 4096 neuron, vagyis a reprezentáció összes lehetséges lépésének száma (4.1. fejezet).

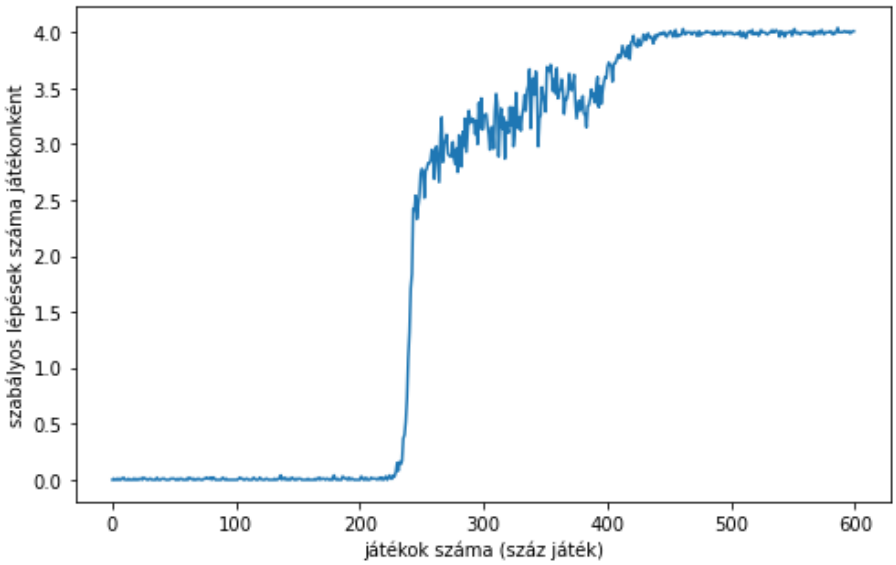
A tanuláshoz a policy gradient módszert használtam, a tanulás paraméterei az amőbán alkalmazottakkal megegyezőek (3.3. fejezet). A tanulás során egy játék maximális hosszát 100 lépésre állítottam be, hogy ne legyenek olyan mérkőzések, melyek aránytalanul sok időt vesznek igénybe. A játék 100 lépés után döntetlennel ér véget. Egy lépéshez tartozó jutalom 1, ha az adott játékot megnyerte, -1 ha elvesztette, és 0, ha döntetlen lett az eredmény.

A tanulási folyamat során a policy gradient játékos a véletlen játékos ellen 50000 játékból nem tudott egyet sem megnyerni. Ennek oka, hogy a reprezentációban az összes lehetséges lépésnek csak egy kis százaléka szabályos lépés, ezért nem tud végigjátszani egy játékot se úgy, hogy ne tenne szabálytalan lépést. Emiatt nem kap pozitív jutalmat, így nem tud tanulni sem. Ha a tanulás közben csak legális lépéseket engedünk meg a játékosnak, akkor egy játék átlagos hossza 15 másodpercre ugrik, ami mellett nem lehetséges érdemi tanítást végezni. A játékos azért nem tud fejlődni, mert nem szerez jutalmakat. Azonban ha kisebb eredményeket is jutalmaznánk, mint egy játék megnyerése, akkor motiválni tudnánk a tanulást. Ennek érdekében módosítottam a jutalmakat.

Először a minimax algoritmus által használt kiértékelő függvényt használtam fel a jutalom meghatározásához (4.2. fejezet). A függvénnyel a lépés előtti és a lépés utáni állapot értékét határoztam meg, majd a kettő különbségét rendeltem jutalomként a lépéshez. A szabálytalan lépések -100 jutalmat kaptak. Azonban ez se bizonyult elég kicsi célnak, mivel 50000 játék alatt egy pozitív jutalmat sem tudott szerezni, vagyis az ellenfél egy

figuráját se tudta leütni. Míg a tanulási folyamat elején a játékok nagy részében egy legális lépést se tudott megtenni, a végére megtanult minden játékban pontosan egy legális lépést tenni. Ez azonban nem elég arra, hogy le tudja ütni az ellenfél figuráit és jutalmat kapjon.

Ez azt mutatja, hogy először csak a szabályos lépéseket kell megtanítani a játékosnak. Ennek érdekében a jutalmakat úgy módosítottam, hogy a szabálytalan lépések 0 jutalmat kapnak, minden más lépés pedig az abban a játékban megtett szabályos lépések számát. A 4.3.1. ábrán látható a tanulási folyamat 60000 játékra.



4.3.1. ábra. Játékonként megtett szabályos lépések átlagos száma a tanulási folyamat során véletlen játékos ellen

Az amőbában a nagyobb táblákra tapasztaltakhoz hasonlóan itt is csak egy idő után indul be a fejlődés, mivel eleinte nem tud elég szabályos lépést megtenni a megfelelő mennyiségű jutalom szerzéséhez. Ezután a lépések száma hirtelen megugrik, majd rövid növekedés után megáll 4 lépésnél. Így, hogy már pár szabályos lépést meg tud tenni, megpróbáltam a minimax kiértékelő függvényének jutalmaival továbbtanítani. Ez azonban nem volt sikeres, mivel az első 4 lépésben az esetek túlnyomó többségében nem ütötte le az ellenfélnek egy figuráját se.

A probléma az, hogy a játékos a tanulás során csak a játék kezdeti állapotával találkozik, és erre meg tud tanulni pár lépést, azonban ahogy halad előre a játék és változik a

tábla, már nem tudja, hogy milyen szabályos lépéseket lehet tenni, mert ezekkel az állapotokkal nem találkozott eleget. Ennek megoldásához a játék minden szakaszából egyenletesen kell vennünk állapotokat, és ezeken kell tanítanunk a játékost. A megerősítéses tanulás azonban erre nem alkalmas, csak teljes játékok lejátszására, ezért erre felügyelt tanulást használtam.

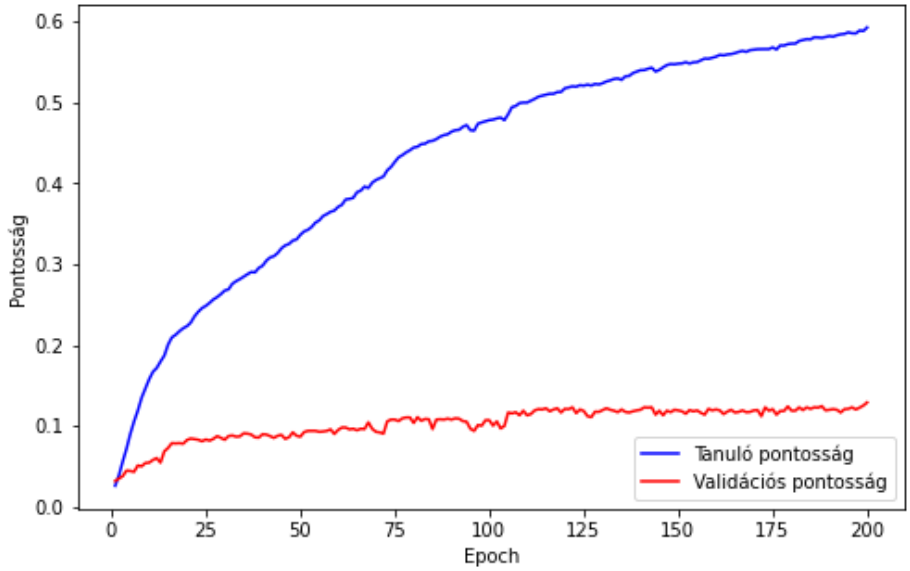
4.4. Felügyelt tanulás alkalmazása sakkra

Ahhoz, hogy a gépet megtanítsam szabályos lépéseket tenni, valódi meccseken vett állásokon klasszifikációs modellt tanítottam. Adathalmazomat a `https://www.ficsgames.org/` oldalról gyűjtöttem, ez 8816 állást, és az állásból megtett lépéseket tartalmazza. Mindegyik állás 2600 Élő-pontszám feletti játékosok mérkőzéseiből származik, ami a játékosok legmagasabb szintjét jelenti. Így a gép által tanult lépések nem csak szabályosak, hanem magas szintűek is. Az adatbázis csak fehér oldali lépéseket tartalmaz. Az adatokat a tanításhoz 7-3 arányban osztottam fel tanító és validációs halmazra. A tanítás során az adathalmaz állásai a leíró változók, lépései pedig a célváltozók. A cél az, hogy a tanulás végére a neurális háló minél nagyobb pontossággal tudja megmondani, hogy egy adott állapotban egy profi játékos mit lépne. Az ebben a fejezetben tárgyalt módszereket a [6] cikk ötletei alapján valósítottam meg.

A felhasznált neurális háló egy 768 neuron méretű bemeneti rétegből, egy 1000 neuron méretű rejtett rétegből és egy 4096 neuron méretű kimeneti rétegből áll, mindegyik teljesen kapcsolt a következővel. A rejtett rétegben ReLu aktivációs függvényt, a kimeneti rétegben softmax aktivációs függvényt használtam. A megerősítéses tanuláshoz hasonlóan a kimeneti réteg egy valószínűségi eloszlást ad meg az összes lehetséges lépésen. Mérkőzések lejátszásához az eloszlás alapján választottam egy véletlenszerű lépést minden állapotban. Az így létrehozott játékost innentől felügyelt tanuló játékosnak hívom.

A tanulási folyamathoz használt veszteség függvény (loss function) a háló kimeneti rétege és a helyes lépés közötti kereszt-entrópia (cross-entropy). A tanulás során ennek a függvénynek egy lokális minimumát keressük gradiens módszerrel, melyről részletesen a [4] könyvben lehet olvasni. A háló paramétereit 20 állásonként frissítettem. A tanulás hossza 200 epoch volt, vagyis az adathalmaz minden elemét 200-szor kapta meg bemenetként a háló a tanulás során.

A tanuló halmazon mért pontosság a tanulás végéig emelkedett, az idő múlásával



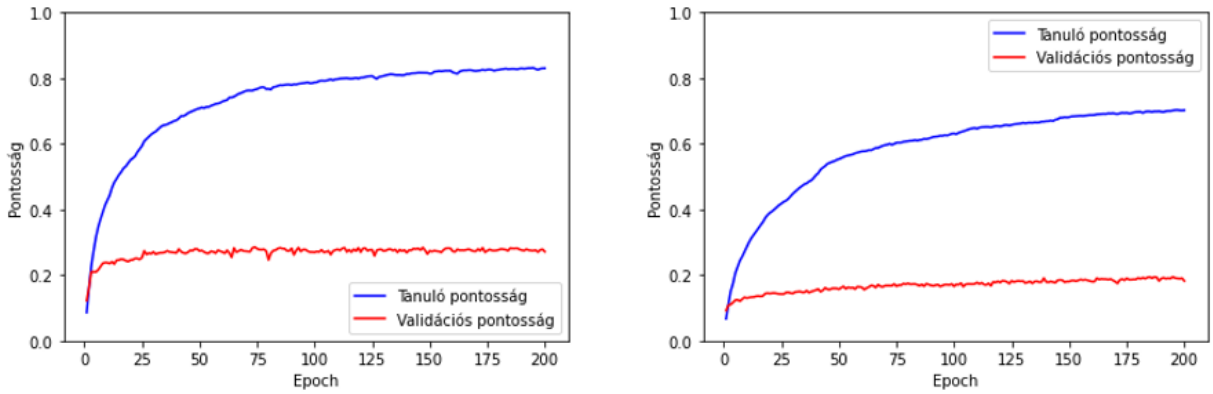
4.4.1. ábra. Felügyelt tanulás során elért pontosságok

azonban lelassult a növekedés. A validációs halmazon mért pontosság egy ideig jelentősen növekedett, azonban a tanulás végére szinte teljesen megállt. A tanulás végére a tanuló halmazon mért pontosság $0,5929$, a validációs halmazon mért pontosság $0,1293$. Ez azt jelenti, hogy ha egy olyan állást adunk a hálónak, ami a tanító halmazban benne volt, akkor $59,29\%$ eséllyel találja el a hozzá tartozó lépést, viszont ha bármilyen állást kaphat, akkor az ahhoz tartozó lépést csak $12,93\%$ eséllyel találja el. Azonban ez nem jelenti azt, hogy amikor nem találja el a profik által tett lépést, akkor szabálytalan lépést választ.

A győzelmi arány mérésére a felügyelt tanuló játékos 10 mérkőzést játszott a véletlen játékos ellen. A mérkőzések hosszát 100 maximális lépésre limitáltam. A 10 játékból a felügyelt tanuló játékos mindet elvesztette, mert mindegyikben szabálytalan lépést tett. Átlagosan játékonként 4,7 szabályos lépést tudott tenni. A mérést megismételtem úgy, hogy a szabálytalan lépéseket nem engedtem meg neki. A 10 játékból a felügyelt tanuló játékos 21-ben nyert, 25-ben vesztett, 54-ben pedig döntetlen lett az eredmény, vagyis nem sikerült befejezni 100 lépés alatt.

A tanulás sikerességét a legnagyobb mértékben az hátráltatja, hogy a kimeneti réteg mérete túl nagy a szabályos lépések mennyiségéhez képest. Ennek a problémának a megoldására a játékos feladatát két külön részre bontottam, melyekre két külön neurális hálót tanítottam. Az első háló határozza meg, hogy a játékos melyik mezőről lépjen el, a második háló pedig arról dönt, hova lépjen az első háló által megadott mezőről. Így

mindkét neurális háló kimeneti rétegének mérete az eddigi 4096 neuronról 64 neuronra csökkent. A második háló bemeneti rétegében az eddigi 768 neuronon kívül szükséges egy új neuron, amely annak a mezőnek a sorszámát tartalmazza, ahonnan el kell lépnie. Ezt az információt tanulásnál az adatbázisban az adott állapothoz rendelt lépésből nyeri ki, mérkőzések lejátszása esetén pedig az első neurális háló kimenetéből kapja. A két hálón két külön tanulást végeztem. A kimeneti és bemeneti rétegeken kívül a neurális háló szerkezetét, illetve a tanulás paramétereit az eddigiekhez képest nem változtattam. A két tanulási folyamat során elért pontosságok a 4.4.2. ábrán látszanak. Bal oldalt az első háló tanulási folyamata, jobb oldalt a második háló tanulási folyamata szerepel.



4.4.2. ábra. Felügyelt tanulás során elért pontosságok, Bal oldalt az első háló tanulási folyamata, jobb oldalt a második háló tanulási folyamata látható

A tanulás végére az első háló tanuló halmazon mért pontossága $0,8302$, a validációs halmazon mért pontossága $0,2715$. A második háló tanuló halmazon mért pontossága 0.7020, a validációs halmazon mért pontossága pedig $0,1822$. Ezek az eredmények jelentősen jobbak, mint 1 háló tanítása esetén, azonban ez nem feltétlenül jelenti azt, hogy jobb lesz az így létrehozott játékos.

A módszer legnagyobb hibája az, hogy a profi játékosok általában nem játszák végig a játékot, hanem amikor már látják a biztos vereséget, akkor feladják azt, így nem tanulja meg a felügyelt tanuló játékos, hogyan kell egy játékot befejezni egy előnyös szituációból. Ráadásul az általam létrehozott környezetben a játék megnyeréséhez nem elég mattot adni, hanem le kell ütni az ellenfél királyát, ez pedig egészen biztosan nem történt meg az adatbázis egyik lépésében sem.

4.5. Fejlesztési lehetőségek

A mérések készítése közben számos új ötlet felmerült, melyek implementációja már nem fért bele a szakdolgozat kereteibe. Ebben a fejezetben ezen módszerek közül néhányat fogok röviden megemlíteni. Ezek azok a fejlesztési irányok, amelyeket az eddig tárgyalt módszerek után elsőként valósítanék meg.

A 4.4. fejezetben említett két hálóból a második helyett használhatunk 6 különböző hálót, melyek a játék 6 különböző figurájának lépéseiért felelősek. Mind a 6 hálót csak az egyik figura lépésein tanítjuk, majd a játék során mindig azt a hálót használjuk, amelyik figurájával lépni akarunk az első háló döntése alapján. Így egy hálónak egyrészt sokkal kevesebb lehetséges lépést kell megtanulnia, másrészt pedig nem kell megtanulnia megállapítani, hogy épp milyen figurával kell lépnie [7].

A módszereimben a neurális hálónak bemenetként mindig csak a tábla egy egyszerű reprezentációját adtam meg. Ebből a döntések meghozásához a hálónak minden információt meg kellett tanulnia kinyerni. Azonban ha bemenetként összetettebb információkat is megadunk a hálónak, azzal megkönnyíthetjük a tanulási folyamatot. Ilyen információk például az egyes figuratípusok száma a táblán, a mezők maximális száma, amit az egyes figurák a lehetséges irányokba lépni tudnak, az egyes mezőket támadó és védő figurák listája vagy az, hogy az egyes játékosok tudnak-e sáncolni. Ezek mind olyan információk melyek egy ember számára könnyen megállapíthatóak, azonban a gép nehezen tudja kinyerni őket a tábla reprezentációjából [8].

Sakk tanulására a legtöbben konvolúciós hálót használnak az általam alkalmazott előrecsatolt hálók helyett. A konvolúciós háló megkönnyítheti a tábla reprezentációjából az információk kinyerését.

A 4.4 fejezetben létrehozott neurális hálókat tovább taníthatjuk megerősítéses tanulással. A megerősítéses tanulás legnagyobb nehézsége az volt, hogy a neurális hálónak nem tudtuk megtanítani a szabályos lépéseket. Azonban a felügyelt tanulással létrehozott hálók már viszonylag jól tudnak szabályos lépéseket tenni, ezért ez egy megfelelő kiindulópont lehet a megerősítéses tanuláshoz.

Számos magas szintű mesterséges intelligencia sakkban a minimax algoritmust alkalmazza alapként, a gépi tanulást pedig a kiértékelő függvény meghatározására használja. Ennek sok lehetséges módja van, ebből én kettőt emelek ki. Egyik ilyen megközelítés az, hogy a 4.4 fejezetben létrehozott neurális hálók kimenetét építjük be a kiértékelő függ-

vénybe. A minimax algoritmus a döntés meghozásakor figyelembe veszi a figurák értékét, illetve azt is, hogy egy profi játékos az adott állapotban mit lépne a háló szerint. Így jó döntéseket tud hozni azokban az állásokban is, melyeket a tanulás során nem látott. Ennek a megvalósítását részletesebben a [6] cikk tárgyalja. A másik ilyen megközelítés szerint egy összetettebb kiértékelő függvény, mint például a Stockfish értékeinek meghatározására használunk felügyelt tanulást. Ezáltal a minimax algoritmus használata során a kiértékelés sebességét fel tudjuk gyorsítani. Így összetettebb, magasabb szintű kiértékelő függvényeket is tudunk használni az algoritmus futási idejének növelése nélkül. Ennek megvalósításáról [8] cikkben lehet olvasni.

5. fejezet

Összefoglalás

Összességében a megvalósított módszerek az amőbára jó eredményeket értek el. A minimax algoritmus működött a legjobban, ez 3×3-as táblán tökéletesen tud játszani. Nagyobb táblákra a keresési mélység csökkenése miatt nem mindig tud tökéletes döntéseket hozni az algoritmus, azonban még így is szinte legyőzhetetlennek bizonyult a legtöbb változata. A megerősítéses tanulás által létrehozott játékos ugyan nem érte el a minimax algoritmus erősségét, de kisebb számítási idő mellett alig gyengébb szinten tudott játszani. Ha találnánk továbbtanításához egy erősebb játékost, mint például a minimax algoritmusnak egy valamilyen valószínűségi alapon működő módosítását, akkor véleményem szerint el tudná érni a tökéletes szintet ez a módszer is.

A sakk az amőbánál egy jóval összetettebb játék, ezért nem meglepő, hogy esetében ugyanazok a módszerek kevésbé voltak eredményesek. A minimax algoritmus teljesített itt is a legjobban, de ez is legfeljebb kezdő szintű embereket tudna legyőzni. A megerősítéses tanulás problémába ütközött a szabályos lépések megtanulásánál, ezért ez a megközelítés önmagában nem bizonyult használhatónak. Felügyelt tanulással ugyan a szabályos lépések elsajátítása eredményesebb volt, de ez a módszer önmagában nem bizonyult elégnek ahhoz, hogy a gépi játékos magas szinten tudjon játszani. A mai legerősebb sakk mesterséges intelligenciák, mint például a Stockfish, egyrészt szuperszámítógépeket használnak, melyek nagyságrendekkel gyorsabban tudják végezni a számításokat a hagyományos asztali gépeknél. Másrészt pedig ezek nem csak egy-egy módszert, hanem azok kombinációját használják. Ezért a dolgozatban bemutatott módszerek továbbfejlesztését a különböző megközelítések kombinációjával, illetve a 4.5. fejezetben említett ötletek segítségével tartom lehetségesnek.

Irodalomjegyzék

[1] Patrick Gebhardt. The history of chess AI, Feb 2019. https://blog.paessler.com/the-history-of-chess-ai.

[2] Marissa Eppes. Game Theory — The Minimax Algorithm Explained, Aug 2019. https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1.

[3] Valentino Zocca, Gianmario Spacagna, Daniel Slater, and Peter Roelants. *Python deep learning*. Packt Publishing Ltd, 2017. https://drive.google.com/file/d/1vnBT8D_9G5XAOHhX9QpdIPmqyaA2nJ_L/view.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Sanyam Kapoor. Policy gradients in a nutshell. *Medium-Towards Data Science*, 1(1), 2018. https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d.

[6] Sayon Bhattacharjee. Predicting professional players' chess moves with deep learning, Nov 2018. https://towardsdatascience.com/predicting-professional-players-chess-moves-with-deep-learning-9de6e305109e.

[7] Barak Oshri and Nishith Khandwala. Predicting moves in chess using convolutional neural networks, 2016. http://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf.

[8] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015. https://arxiv.org/pdf/1509.01549.pdf.