Eötvös Loránd Tudományegyetem
Természettudományi Kar

Szakdolgozat

Az AlphaGo és változatai

Milovszky Bence
Matematika BSc
Matematika Elemző szakirány

Témavezető:
Csiszárik Adrián
Számítógéptudományi Tanszék

Budapest, 2020.

Köszönetnyílvánítás

Szeretnék köszönetet mondani témavezetőmnek, Csiszárik Adriánnak, aki szakértelmével és javaslataival segítette a szakdolgozatom létrejöttét. Köszönöm a konzultációkat, mely során betekintést nyerhettem a mesterséges intelligencia világába.

Köszönöm Gáspár Dávidnak a javaslatot a témaválasztást illetően, így egy számomra igen érdekes témán dolgozhattam. Köszönöm továbbá a sok-sok tanulással töltött órát is.

Végezetül köszönöm a családomnak és barátaimnak a támogatást, amit az egyetemi éveim alatt kaptam tőlük.

Tartalomjegyzék

1. Bevezetés **5**

2. Megerősítéses tanulás **6**

2.1. Markov döntési folyamat 7

2.2. Monte Carlo Tree Search 8

2.2.1. MCTS bemutatása az amőbán keresztül 11

3. Mesterséges neurális hálózatok **13**

3.1. A mesterséges neuronok és rétegbe szerveződésük 13

3.2. A mesterséges neurális hálózat tanítása 15

3.3. Konvolúciós neurális hálózatok 16

4. A go játék modellezése **18**

4.1. Az AlphaGo, az AlphaGo Zero és az AlphaZero összehasonlítása 18

4.1.1. A tanulás során a rendszerbe bevitt információk 18

4.1.2. A programok neurális hálózatai 22

4.1.3. A Monte Carlo Tree Search használata 24

4.1.4. Tanítás 28

4.1.5. Játékállások reprezentálása konvolúciós hálókkal 34

4.1.6. A programok kiértékelése 39

4.1.7. Az elért eredmények 41

4.2. Összefoglaló táblázatok 44

5. További eredmények **46**

6. Összegzés **47**

Hivatkozások **48**

A. A go játék szabályai **51**

Táblázatok jegyzéke

1. Bemeneti jellemzők az AlphaGo esetén. 19

2. Rendszerbe bevitt információk összefoglaló táblázata. 21

3. A különböző változatok MCTS felhasználásának összefoglaló táblázata. . . 27

4. A különböző változatok tanításának összefoglaló táblázata. 33

5. A változatok konvolúciós hálózatainak összefoglaló táblázata. 36

6. A különböző verziók főbb tulajdonságait összefoglaló táblázat. 44

7. A változatok eredményeit összefoglaló táblázat. 45

1. Bevezetés

A Google DeepMind fejlesztőcsapata által megalkotott AlphaGo program a mesterséges intelligencia fejlődésének egy fontos mérföldkövének tekinthető. A fejlesztők kezdetben csak olyan programot szerettek volna létrehozni, ami versenyez a többi go programmal, de a nagyobb sikerek után úgy döntöttek, hogy a megszerzett tudást később fontosabb, az emberiség számára hasznos kutatásoknál kamatoztatják.

A go társasjáték eredete körülbelül négyezer évvel ezelőttig nyúlik vissza. Bár a szabályrendszere egyszerű és könnyen meg lehet tanulni, a közel 10^{170}-féle lehetséges táblaállás miatt napjaink egyik legbonyolultabb táblajátékának tekinthető, éppen ezért kedvenc célpontja a mesterséges intelligencia kutatóknak. Különböző go programok születtek, köztük olyanok is, amik előnykövekkel legyőztek profi go játékosokat. Az áttörést 2015-ben az AlphaGo hozta, mely a világon először előnykő nélkül tudta legyőzni a játék Európa bajnokát.

A dolgozatom célja az AlphaGonak és a továbbfejlesztett változatainak a bemutatása és összehasonlítása.

A szakdolgozatom második és harmadik fejezetében bemutatom azokat a matematikai fogalmakat, amik szükségesek az AlphaGo és változatai felépítésének megértéséhez. Először a megerősítéses tanulást vezetem be röviden, majd egy gyors áttekintést adok a mesterséges neurális hálózatokról. A dolgozat negyedik fejezetében pedig összehasonlítom egymással az AlphaGot, az AlphaGo Zerot és az AlphaZerot, kiemelt figyelmet fordítva azokra a szempontokra, amikben a későbbi változatok eltértek az előzőektől. A szakdolgozatomat egy olyan fejezettel zárom, ahol röviden bemutatom a DeepMind további, főként a játékokkal kapcsolatos fontosabb eredményeit.

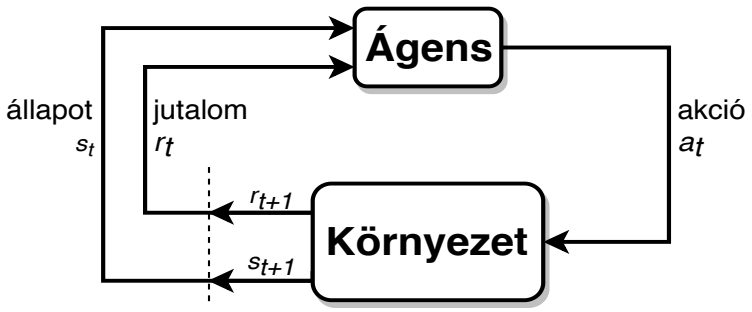
2. Megerősítéses tanulás

A fejezet megírásához a [1] könyvet, a [2] és a [3] cikkeket használtam fel.

A megerősítéses tanulás során egy ágens akciókat hajt végre egy környezetben. Két eleme van a tanulási folyamatnak:

- Az *ágens*: a tanuló és döntéshozó.
- A *környezet*: a dolog, amivel interakcióba lép az ágens.

A megerősítéses tanulás folyamata a következő: az ágens akciókat hajt végre (a go társasjátékban lépéseket), majd, miután a környezet erre az akcióra reagált, új állapotba kerül, amelyet megfigyel az ágens. A környezet továbbá jutalmakat ad. Az ágens célja, hogy maximalizálja a hosszú távon kapott jutalmakat.



1. ábra. Az ágens–környezet interakció a megerősítéses tanulásban.

Az ágens és a környezet diszkrét időközökben interaktálnak egymással ($t = 0, 1, 2, \dots$). Minden t időpontban az ágens megfigyeli az aktuális állapotot ($s_t \in S$, ahol S a lehetséges állapotok halmaza), majd kiválaszt egy akciót ($a_t \in A(S_t)$, ahol $A(s_t)$ az s_t állapotban elérhető akciók halmaza) és azt végrehajtja a környezetben. Egy időlépéssel később, az a_t akció következményeként, az ágens egy számszerű jutalmat kap ($r_{t+1} \in \mathbb{R}$) és egy új állapottal találja magát szemben: s_{t+1}. Ennek a folyamatnak a szemléltetésére szolgál az 1. ábra.

2.1. Definíció (Policy)**.** Az ágens modellezése során a modell feldolgozza az állapotot és minden akcióra meghatároz egy végrehajtási valószínűséget. Ezt a leképezést nevezzük az ágens *policyjának* és π_t-vel jelöljük. A $\pi_t(a|s)$ annak a valószínűsége, hogy a $t = (0, 1, 2, \dots)$ időpontban a $a_t = a$ műveletet hajtuk végre az $s_t = s$ állapotban.

Az ágens policyját, a tapasztalatait felhasználva, a tanítás során változtatjuk.

2.1. Markov döntési folyamat

A megerősítéses tanulás paradigma egy gyakran alkalmazott matematikai modellje a Markov döntési folyamat.

2.2. Definíció (Markov döntési folyamat)**.** A Markov döntési folyamat egy rendezett (S, A, P_a, R_a) négyes, ahol

- S egy véges halmaza az állapotoknak;
- A egy véges halmaza a lehetséges akcióknak;
- $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s,\ a_t = a)$ annak a valószínűsége, hogy t időpontban, s állapotból az a akciót végrehajtva $t + 1$ időpontban s' állapotba jutunk;
- $R_a(s, s')$ az azonnali jutalom (vagy a várható azonnali jutalom), amit akkor kapunk, ha s állapotból s' állapotba jutunk.

Megjegyzés. Mivel az állapotok halmaza és a lehetséges műveletek halmaza is véges, ezért véges Markov döntési folyamatokról beszélünk.

A megerősítéses tanulásos algoritmusok gyakran használnak különböző értékfüggvényeket, amik azt becsülik meg, hogy az ágensnek mennyire jó az, hogy az adott állapotban van (vagy mennyire jó, hogy az adott műveletet végrehajtva, az adott állapotba jut). Azt, hogy mennyire „jó”, azt a jövőbeni várható jutalmak mutatják meg.

Természetesen a várható jutalmakat azok a műveletek befolyásolják, amiket az ágens kiválaszt. Ezeket a jutalmakat szeretnénk maximalizálni és ennek modellezéséhez a Markov döntési folyamatot fogjuk felhasználni.

2.3. Definíció (Diszkontált jutalom)**.** A t időpontbeli diszkontált jutalmat úgy kapjuk meg, hogy a környezettől kapott jutalmat súlyozzuk γ^t-vel (ahol t=0,1,2,... időpontok) és $0 \leq \gamma \leq 1$ a diszkontálási tényező.

A diszkontált jutalom fogalmát azt motiválja, hogy egy későbbi időben megszerzett jutalom kevésbé értékes, mint egy korábban elért. Ez arra ösztönzi majd az ágenst, hogy minél kevesebb akcióval szerezze meg a jutalmat.

Vegyük az ágens policyját, amit továbbra is π-vel jelölünk. Jelöljük $v_\pi(s)$-sel az s állás értékét a π policyben. A $v_\pi(s)$ lesz a jutalom várható értéke, ha az s-ből indulva követjük a π-t.

2.4. Definíció (Értékfüggvény)**.** Az *értékfüggvényt* a következő képlettel definiáljuk:

$$v_\pi(s) = E_\pi\big[g_t|s_t = s\big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\right],$$

ahol a π az alkalmazott policy, $\mathbb{E}_\pi[\cdot]$ a várható érték, g_t a várható diszkontált jutalmak összege ($g_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$), t egy adott időpont és $0 \leq \gamma \leq 1$.

Például az AlphaGo esetében háromféle jutalomról beszélhetünk:

- a jutalom 1, ha nyer az ágens,
- a jutalom -1, ha kikap az ágens,
- a jutalom 0, ha még nem ér véget a parti.

A megerősítéses tanulás egyik fő feladata az, hogy megtalálja azt a π policyt, melyre maximális a várható diszkontált jutalmak értéke.

A Markov Döntési Folyamat alkalmazására egy példa a Monte Carlo Tree Search, melyet a következő alfejezetben fogok bemutatni.

2.2. Monte Carlo Tree Search

A Monte Carlo Tree Search (a továbbiakban gyakran MCTS) egy általános megközelítése a játékproblémák megoldásának, mely központi szerepet játszott az AlphaGo programban is. Ebben az alfejezetben bemutatom a Monte Carlo Tree Search felépítését és a könnyebb megértés érdekében bemutatom az algoritmus működését egy amőbajátékon keresztül.

Az MCTS-t szinte minden go program használja, sikerességet tekintve az áttörést az AlphaGo érte el: az algoritmust két mély neurális hálózattal kombinálták, melyek egyszerre felelnek az állások kiértékelésért és a lépések kiválasztásért.

Az algoritmust négy lépésre le tudjuk bontani. Ezeket a lépéseket ugyanúgy tudjuk értelmezni játékfákon[1] és döntési fákon egyaránt:

- „Kiválasztás” (Selection) - Játékfák esetében egy lépés kiválasztása, döntési fa esetében pedig egy akció végrehajtása. A kiválasztási lépésben különböző algoritmusokat alkalmazhatunk:
  - Az Upper Confidence Bound (UCB) [4] algoritmus azt a csúcsot fogja kiválasztani, ahol a jelenlegi becsült érték (pl. a győzelem valószínűségének az értéke) és egy véletlenszerű érték (úgynevezett „fudge” factor) összege a legnagyobb.
  - A UCT algoritmus egy rollout alapú algoritmus, mely a kezdeti állásból epizódokat mintavételez. Egy epizódnak egy állás-lépés-jutalom hármas tekinthető. A fát az epizódokból kinyert információ alapján építi fel.

    Lépés kiválasztása: Az UCT algoritmus az s állásban és a d mélységben azt a lépést fogja kiválasztani, amelyre az alábbi képlet a legnagyobb:

    $$Q_t(s, a, d) + c_{N_{s,d}(t), N_{s,a,d}(t)},$$

    ahol $Q_t(s, a, d)$ a becsült értéke az a lépésnek az s állásban és a d mélységben és a t időpontban, $N_{s,d}(t)$ azt jelöli, hányszor látogatta meg az s csúcsot a t időpontig a d mélységben, illetve $N_{s,a,d}(t)$ azt jelöli, hányszor választottuk az a lépést, amikor s állásban voltunk a t időpontban és a d mélységben [5].
  - A PUCT algoritmus a Markov Döntési Folyamat epizódjait egymás után megismétli és közben felhasználja az előző epizódok információit annak érdekében, hogy felfedezze és megtalálja az optimális lépéseket a későbbi epizódokban. A lépés kiválasztása hasonlít az UCT-éhez, tehát azt a lépést választjuk, aminél a lépés érték és egy változó a legnagyobb. Ennek a változónak az értéke függ egy úgynevezett felfedezési konstanstól, a valószínűségtől és a látogatás számától [6][7][8].

[1] Játékfa: olyan irányított gráf, melynek a csúcsai a játékállásokat, az élei pedig a végrehajtott lépéseket reprezentálják.

- „Kiterjesztés” (Expansion) - A kiválasztás során, ha egy olyan L csúcsig jutunk el, ami nem a játék végét jelenti, akkor az algoritmus egy vagy több csúccsal bővíti a fát.

- „Szimuláció/Kiértékelés” (Simulation, Evaluation) - A szimulációkat Monte Carlo rolloutok végzik. Ezek a rolloutok visszaadják egy adott csúcs becsült értékét, ami megfelelhet akár döntési fa vagy a játék tényleges végének, vagy egyszerűen egy olyan pontnak, ahol a jelenlegi becslés kellően pontosnak tekinthető, hogy ne legyen szükség további szimulációra. Több szimuláció végrehajtása közben a keresőfa folyamatosan növekszik és a releváns értékeink pontosabbak lesznek.

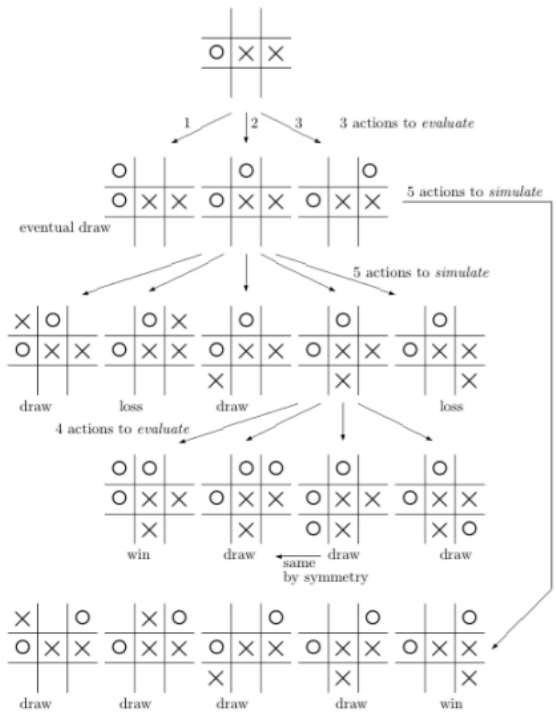
- „Visszaterjesztés” - A szimuláció során keletkezett eredményekkel frissíti a játékfát és a döntési fát.

Ez az eljárás különböző játékokban (például az ostáblában [9] és a Scrabbleben [10]) emberi teljesítményt meghaladó eredményt ért el, viszont a go játékban csak egy erősebb amatőr szintjét sikerült elérnie [11]. Ahhoz, hogy olyan hatékony legyen, mint az AlphaGoban, hatékonyan fel kell mérnie a következő lépések értékét.

Minden egyes helyzetet figyelembe kellene vennie, de néhány játéknál, mint például a go, ez lehetetlen, hisz túl sok lehetőséget kellene vizsgálnia. A minimális elvárásunk az, hogy legalább az első lépése jó legyen.

De hogy tudjuk eldönteni azt, hogy az a lépés jónak számít? Ha lenne egy jósunk, ő talán meg tudná mondani, melyik a legjobb lépés, de igazából ő sem lenne képes arra, hogy biztosan meghatározza, melyik az a kezdőlépés, amivel biztosan nyernénk. Ezért különböző valószínűségeket rendelünk minden lépéshez: ezek lesznek a rolloutok által meghatározott értékek. Nagyon fontos, hogy minél pontosabbak legyenek a valószínűségek, hiszen csak ekkor van esélyünk megnyerni a partit.

Ha megtalálnánk az ideális játékmenetet, egyhamar monotonná válna a játék, hisz a program ugyanazokat lépkedné mindig. Éppen ezért az algoritmus felváltva dönt aközött, hogy továbbhalad a játék megnyerése felé vagy kipróbál más lépéseket is.



2. ábra. Egy amőbaparti kimenetelei [2].

2.2.1. MCTS bemutatása az amőbán keresztül

A 2. ábrán egy adott állásból követhetünk végig egy amőbajátékot és közben megnézhetjük, hogyan működik a Monte Carlo Tree Search.

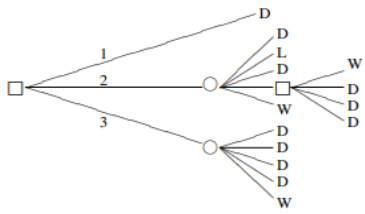
Először az MCTS három lépést értékel ki. Az elsőt nem választja, hiszen a játék jó eséllyel döntetlenbe torkollik.

Nézzük meg mi történik akkor, ha az MCTS a második lehetőséget választaná. Ekkor öt lépést szimulál le az algoritmus. Az ötből kettő vereséghez, kettő döntetlenhez vezet, egy lépést pedig tovább tudunk vizsgálni. Haladjunk tovább ezzel a lépéssel, legközelebb további négy akciót kell kielemezni: három után döntetlennel (két állás ráadásul szimmetrikus egymással), egy után pedig (a körrel játszó játékos számára) győzelemmel zárul a játék.

Vizsgáljuk meg a harmadik esetet is. Az algoritmus ekkor öt lépést szimulál le, ami után egyszer nyerne, négyszer pedig döntetlent játszana a program.

Ha nem a játékállásokkal szeretnénk bemutatni az algoritmus működését, akkor készíthetünk egy egyszerűbb ábrát is. A döntési fa modell viszont csak a lépések kimeneteleit

mutatja meg nekünk, így ezt csak akkor célszerű használni, ha nem vagyunk kiváncsiak arra, hogy milyen lépéseket értékel ki és szimulál a program, hanem csak az érdekel minket, hogy mi lesz a játék végkimenetele (3. ábra).



3. ábra. Az amőbapartink döntési fája [2].

Az ideális játékmenet tehát az, amikor a harmadik lépést választja ki az algoritmus. Hogy miért? Azért, mert egyrészt legrosszabb esetben csak döntetlent játszana a program, másrészt ha nyer, akkor korábbi időpontban tenné meg, mint ha a másik lépést tenné meg (2.3. definíció). Ehhez feltesszük azt, hogy a gépi ágens már akkor megkapja a jutalmát, mikor biztossá válik a győzelme.

Megjegyzés. Mivel a második esetnek is lehet győztes kimenetele, előfordulhat, hogy kihasználva a véletlen faktort, a program a második lépést fogja meglépni, bár ekkor egy tapasztaltabb amőba játékos tudná, hogy mit kellene lépnie a biztos győzelemért.

3. Mesterséges neurális hálózatok

A mesterséges neurális hálók áttekintéséhez a [12] könyvet, a [13] honlapot és a [14] jegyzetet használtam fel. A témában alapvető szerepet játszanak a neuronhálók, ugyanis a policy-t és az értékfüggvényt neuronhálóval fogjuk modellezni a következő fejezetben.

3.1. A mesterséges neuronok és rétegbe szerveződésük

A mesterséges neuronok a mesterséges neuronháló alapú gépi tanulás alapvető elemei. Felépítésüket az emberi agyban található neuronok szerkezete inspirálja.

A mesterséges neuronok olyan matematikai függvények, amelyek a mesterséges neuronhálók építőkövei.

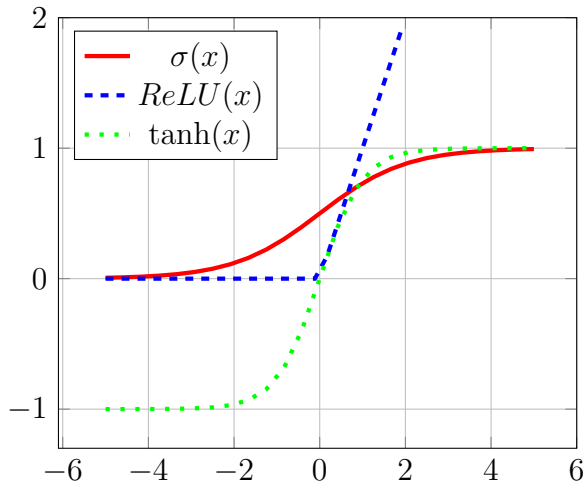
3.1. Definíció (Mesterséges neuron)**.** A *mesterséges neuronok* felépítését meg tudjuk határozni a következőképpen:

$$f_{w,b}(\underline{x}) = \sigma(\sum_{i=1}^{n} w_i x_i + b),$$

ahol $x_i \in \mathbb{R}$ a bemenet, $w_i \in \mathbb{R}$ a bemenethez tartozó súlyok, $i \in [n]$, $b \in \mathbb{R}$ az eltolás és a $\sigma : \mathbb{R} \to \mathbb{R}$ az alkalmazott aktivációs függvény.

Megjegyzés. Az *aktivációs függvények* olyan differenciálható függvények, amik a lineáris bemenetfüggvényből nemlineárisat készítenek. A leggyakrabban használt aktivációs függvények a következők:

- A *Szigmoid aktivációs függvény*:
  $\sigma(x) = \frac{1}{1+e^{-x}}$, ahol $\sigma : \mathbb{R} \to \mathbb{R}$ és x a bemenetünk.
- A *ReLU (*Rectified Linear Unit*) függvény*:
  $ReLU(x) = \max(0, x)$, ahol $ReLU : \mathbb{R} \to \mathbb{R}$ és x a bemenetünk.
- A *tanh(x) függvény*:
  $\tanh(x) = \frac{2}{1+e^{-2x}} - 1$, ahol $\tanh : \mathbb{R} \to \mathbb{R}$ és x a bemenetünk.



4. ábra. Az aktivációs függvények grafikonjai.

Mivel a szigmoid függvény értékkészlete 0 és 1 között van, így ezt akkor alkalmazzák, ha valószínűségeket kell vizsgnálni. Hátránya az, hogy ha az x abszolút értéke nagy, akkor a gradiense nagyon kicsi szám lesz, így csökken a tanulás hatékonysága.

Az utóbbi években a leggyakrabban használt aktivációs függvény a ReLU volt. Ennek oka az, hogy $x > 0$ értékekre a függvény lineáris, tehát a gradiense nem tűnik el, így kiküszöböli szigmoid függvény előző bekezdésben említett problémáját.

A mesterséges neuronokat rétegekbe is tudjuk szervezni, ekkor *mesterséges neuron rétegekről* beszélünk.

3.2. Definíció (Mesterséges neuron réteg)**.** *A mesterséges neuron réteget* fel tudjuk írni a következő alakban:

$$f_{W,\underline{b}}(\underline{x}) = \sigma(W\underline{x} + \underline{b}),$$

ahol $\underline{x} = (x_1, x_2, \ldots, x_n), ahol x_i \in \mathbb{R}$, tehát $\underline{x} \in \mathbb{R}^n$ a bemeneti vektor, $W \in \mathbb{R}^{m \times n}$ a súlymátrix, $\underline{b} \in \mathbb{R}^n$ az eltolásvektor és a $\sigma : \mathbb{R}^m \to \mathbb{R}^m$ az alkalmazott aktivációs függvény.

A mesterséges neuron rétegeket gráfként is tudjuk értelmezni. Ha a gráfunk aciklikus, azaz, ha az egyik rétegben található neuron nem vezet vissza egy korábbi rétegben található neuronra, akkor *előrecsatolt mesterséges neurális hálózatról* beszélünk.

3.3. Definíció (Előrecsatolt mesterséges neurális hálózat)**.** A *előrecsatolt mesterséges neurális hálózatot* az alábbi parametrizált függvénykompozícióval tudjuk megadni:

$$f_\theta(x) = f_L \circ \cdots \circ f_2 \circ f_1(x),$$

ahol θ_i az f_i függvény paraméterei, így a $\theta = (\theta_1, \theta_2, \ldots, \theta_L)$ a neurális háló paraméterei, az $f_1, f_2, \ldots, f_L$ függvények pedig a rétegek függvényei.

3.2. A mesterséges neurális hálózat tanítása

A mesterséges neurális hálózatunkat úgy szeretnénk megtanítani, hogy a neurális háló által számolt kimenet és az elvárt kimenet közötti eltérés minél kisebb legyen. Ezt úgy szeretnénk elérni, hogy veszünk egy függvényt, amely megfogja ezt az eltérést. Ezt a függvényt *veszteségfüggvénynek* nevezzük. A tanítás során ezt a veszteségfüggvényt szeretnénk minimalizálni.

3.4. Definíció (Veszteségfüggvény)**.** A *veszteségfüggvényt* az alábbi képlettel adhatjuk meg:

$$L(\theta, x, y) = \frac{1}{N} \sum_{i=1}^{N} d(f_\theta(x^{(i)}), (y^{(i)})),$$

ahol N az adatpontok száma, $x^{(i)}$ az i-edik adatpont, $y^{(i)}$ az i. adatponthoz elvárt kimenet, az $f_\theta(x)$ a neurális háló által számolt kimenet és a d egy skalár értékű függvény, amely méri a neuronháló kimenete és az elvárt kimenet közötti eltérést.

Megjegyzés. A különbséget megadhatjuk például 1-es ($d(\theta, x, y) = \|f_\theta(x) - y\|_1$) vagy 2-es normával ($d(\theta, x, y) = \|f_\theta(x) - y\|_2$) is.

Tanítás sztochasztikus gradiens ereszkedő módszerrel ([15]). A tanításhoz a gradiens ereszkedő módszert használjuk. Ha a teljes adathalmazra számolnánk ki a veszteségfüggvényünk gradiensét, annak a memóriaigénye nagy, ezért valamilyen sztochasztikus módszerrel közelítjük a gradienst. Egy lehetséges módszer, hogy egyesével vesszük az adathalmaz pontjait véletlenszerűen a tanítóhalmazból, de a jelenlegi chip design és az alkalmazott módszerek sajátossága miatt hatékonyabb, ha nem egyetlen adatpontot veszünk, hanem több adatpontot választunk véletlenszerűen a tanítóhalmazból, ezt a módszert nevezzük kötegelt sztochasztikus gradiens módszernek.

A sztochasztikus gradiens ereszkedő módszer lényege a következő: először veszi a hibafüggvény súlyok szerinti gradiensét, majd úgy módosítjuk a neurális hálózat súlyait, amerre a kiszámolt érték negatív. Bár a módszer nem garantálja, hogy globális minimumhoz érkezünk, mivel a hibafüggvény nem konvex, a tapasztalat azt mutatja, hogy egy lokális minimumba érkezve is már hatékonyan tud működni a tanult függvény.

3.3. Konvolúciós neurális hálózatok

A konvolúciós neurális hálózatok a gépi látás területén értek el áttörő eredményeket. Főbb használati területként ide lehet sorolni a kép- és hangfelismerést, a képklasszifikációs feladatokat és az orvosi képanalízist is.

A hálózat legfontosabb rétegei a *konvolúciós rétegek*. A konvolúciós rétegek emelik ki a képi jellemzőket. Ezeket a rétegeket több szűrővel vizsgáljuk. A szűrők olyan $F \times F$ méretű négyzetek, melyek egy bizonyos S lépésközök mentén letapogatják a bemenetként szolgáló képet és közben egy aktivációs függvényt használnak. Az eredmény egy aktivációs térkép lesz, mely a szűrő által kinyert eredményeket tárolja. Az aktivációs térképünket további szűrőkkel vizsgáljuk egészen addig, amíg meg nem kapjuk a kimenetünket. A kimenetünk lehet egy szám (például a go játékban a várható jutalom értéke), egy vektor (ami a go játék esetén tartalmazza a lehetséges lépések valószínűségét) vagy egy olyan vektor, melynek hossza megegyezik a kategóriák számával. Ez az úgynevezett „One-hot encoding”.

3.5. Definíció (One-hot encoding)**.** *One-hot encoding* esetén a kimenetünk a következő vektor:

$$\underline{y} = [y_1, y_2, \ldots, y_n],$$

ahol $\underline{y}$ a kimeneti vektor, n a kategóriák száma és $y_i = 1$, $y_1, y_2, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n = 0$, ha a bemenetünk az i. kategóriába tartozik.

Az aktivációs térképek térbeli mérete viszont egyre csak csökken. Ennek elkerülésére több módszer is létezik, például a kép szélét kipárnázhatjuk (padding). A szakdolgozatom csak ezt a lehetőséget vizsgálja, ugyanis az AlphaGo, az AlphaGo Zero és az AlphaZero konvolúciós hálózatának felépítéséhez nullákkal való párnázást (zero padding) használtak.

Általánosan elmondható, hogy a konvolúciós réteg egy $\mathbb{R}^{W_1 \times H_1 \times D_1} \to \mathbb{R}^{W_2 \times H_2 \times D_2}$ leképezés, ahol W_1, H_1, D_1 a bemenet szélessége, magassága és mélysége. Több paramétert is meg kell adnunk ahhoz, hogy pontosan meg tudjuk határozni a W_2, H_2, D_2 értékeket:

Szűrők száma: K

Szűrők mérete: F

Lépésköz: S

Padding (kipárnázás): P

A kimenetünk így tehát $W_2 \cdot H_2 \cdot D_2$ méretű lesz, ahol

$W_2 = (W_1 - F + 2P)/S + 1$

$H_2 = (W_1 - F + 2P)/S + 1$

$D_2 = K$

Megjegyzés. A szűrő paramétereinek száma: $F \cdot F \cdot D_1 + 1$ és a réteg összes paraméterének száma: $K \cdot F \cdot F \cdot D_1 + K$.

Megjegyzés. A szűrő mélysége megegyezik a bemenet mélységével, illetve az aktivációs térképünk mélysége megegyezik a szűrők számával. Továbbá a szűrők mérete praktikusan kisebb, mint a szűrők mérete, hogy a szűrők a különböző pozíciókban összeérjenek.

Nem feltétlenül csak konvolúciós rétegek találhatók a konvolúciós neurális hálózatban, hanem összevonó rétegek és teljesen összekötött rétegek is lehetnek a hálózatban.

Az *összevonó rétegek* csökkentik a bemenet dimenzióját, így könnyebben kezelhető lesz. Ezekre a rétegekre a szakdolgozatomban nem fogok külön kitérni, hiszen a három vizsgált programunk konvolúciós neurális hálózatának szerkezete nem tartalmaz ilyen rétegeket.

A *teljesen összekötött rétegek* esetében a bemenet minden eleme össze van kötve a réteg összes neuronjával. Általában a konvolúciós neurális hálózatok utolsó rétegeiként szokták alkalmazni őket. Ezekből a rétegekből lehet egy vagy több is és a kimenetelt készítik elő.

4. A go játék modellezése

Ebben a fejezetben több szempontot figyelembe véve összehasonlítom az AlphaGot a továbbfejlesztett változataival, megvizsgálva minden fontos szempontot, hogy miben fejlődtek a későbbi változatok. Mivel az AlphaZero nem csak a go, hanem a sakk és a sógi játékokkal is foglalkozik, így a teljesség kedvéért röviden néha ki fogok arra térni, hogy a sakkot és a sógit hogyan modellezték, illetve ezek a játékok hogyan befolyásolták az AlphaZero felépítését az előző két verzióval szemben.

Megjegyzés. Ebben a fejezetben gyakran szerepelni fog a *self-play*, mint fogalom. Ez alatt mindig az adott program egy előző iterációja elleni játékát fogom érteni.

Az AlphaGo, az AlphaGo Zero és az AlphaZero mellett említésre kerül még további két változat, amelyeket bár a DeepMind külön nem publikált, mégis fontos szerepük van.

AlphaGo Lee ([16]) Az első ilyen változat, az *AlphaGo Lee*, az a program, amely 2016 márciusában legyőzte a 9 danos Lee Sedolt 4:1-re.

AlphaGo Master ([16][17]) A második változat, az *AlphaGo Master*, online verziója az AlphaGonak, 2017 januárjában zsinórban 60 meccset nyert meg a legjobb játékosok ellen.

Megjegyzés. Az AlphaGora szokás *AlphaGo Fanként* is hivatkozni, ugyanis ez a program győzte le Fan Huit 2015 Októberében.

4.1. Az AlphaGo, az AlphaGo Zero és az AlphaZero összehasonlítása

Az alfejezet megírásához a [3], a [16] és a [18] cikkeket használtam fel.

4.1.1. A tanulás során a rendszerbe bevitt információk

Az első szempont, ami alapján az összehasonlítást végzem az az, hogy milyen bemenetet adtak meg a fejlesztők a tanulás során a rendszernek.

Profi go játékosok játék adatbázisa Az AlphaGo tanítása során felhasznált adathalmaz 29.4 millió állást tartalmazott 160 ezer játékból, amiket a KGS Go szerveren 6 és 9 dan közötti emberek játszottak le. A játékok 35.4%-a előnyköves játék volt. A passzolásokat kivették az adathalmazból. A későbbi változatok, tehát az AlphaGo Zero és az AlphaZero már nem használt fel semmilyen adathalmazt, csupán self-play közben ismerte meg a különböző lépések előnyeit és hátrányait.

Játékszabályok ismerete Mindhárom program tökéletesen ismerte a játék szabályait, ugyanis azok ismerete elengedhetetlen volt ahhoz, hogy tökéletesen tanítsák be a programokat. A játék szabályai megtalálhatóak az A. függelékben.

További bevitt jellemzők Az AlphaGo esetében az s állást 19×19-es síkok sorozatát tartalmazza. A méret onnan jön, hogy a go tábla 19×19 méretű. A felhasznált jellemzőket közvetlenül a játéktábla állásaiból számolták ki.

Jellemző	Db[2]	Leírás
Kő színe	3	Játékos köve / ellenfél köve / üres
Egyesek	1	Egy konstans sík, ami egyeseket tartalmaz
Körök a lépés óta	8	Hány kör telt el azóta, hogy megléptek egy lépést
Életek száma	8	A kövek/csoportok életeinek száma
Fogoly mérete	8	Az ellenfél hány kövét lehetne foglyul ejteni
Lehetséges fogoly mérete	8	Az ellenfél hány követ tudna foglyul ejteni
Életek a lépés után	8	A kövek/csoportok életeinek száma a lépés után
Foglyul ejtés (létránál)	1	Egy lépés az adott pillanatban egy sikeres elfogás
Szabadulás (létránál)	1	Egy lépés az adott pillanatban egy sikeres menekülés
Értelmesség	1	Egy lépés legális-e
Nullák	1	Egy konstans sík, ami nullákat tartalmaz
Játékos színe	1	Az aktuális játékos fekete-e

1. táblázat. Bemeneti jellemzők az AlphaGo esetén.

Az *AlphaGo* esetén a policy hálózat tanításához 48, míg az érték hálózat tanításához 49

[2]Síkok száma

bemeneti jellemzőt használtak, melyek a tábla egyes pontjainak állapotát jelzik: a kövek színeit (melyeket a nullákat és az egyeseket tartalmazó síkok reprezentálnak), a szabad szomszédos pontok számát, a foglyul ejtéseket, a lépések legalitását, a körök számát, amióta megjátszották az adott követ és (csak az érték hálózat esetében) a következő játékos színét. Továbbá, van két taktikai jellemző is, mely a létrázás kimenetelét számolja ki (hogy ki tud-e szabadulni a játékos a létrából, vagy esetleg az ellenfél foglyul ejti a kőcsoportját). Az összes jellemzőt az aktuális színhez viszonyítva számolták: a kövek színét nem feketével és fehérrel, hanem az aktuális játékossal és ellenfelével határozták meg. A bemeneti jellemzők teljes listáját az 1. táblázat tartalmazza.

Az *AlphaGo Zero* neurális hálózatának bemenete 17 darab, bináris értékeket tartalmazó, 19×19-es sík. 8 darab X_t sík megmutatja a táblán az adott játékos köveit: $X_t^i = 1$, ha az i metszésponton van a játékosnak köve a t időpontban és 0 az érték, ha a metszéspont üres, ellenfél köve van rajta vagy $t < 0$. 8 darab további Y_t sík az ellenfél köveit jelöli meg. A 17. sík a következő játékos színét jelzi: csupa 1 az értéke, ha a fekete következik és csupa 0, ha a fehér. Ezek a síkok együtt adják az AlphaGo Zero neurális hálózatának bemenetét: $s_t = [X_t, Y_t, X_{t-1}, Y_{t_1}, \ldots, X_{t-7}, Y_{t-7}, C]$. Az X_t és Y_t síkok tárolják a játék történetét (a neurális hálózat így tehát az előző 16 lépést bemenetként megkapja), ami azért is fontos, mert a go játékban nem csak a kövek állását kell figyelembe venni, hanem azt is, hogy nem ismétlődhetnek meg a játékállások (példa: ko szabály – A. függelék 12/b ábra). A C síkot sem hagyhatjuk el, mert a komi szabály szempontjából fontos, hogy épp melyik játékos lép és ejti foglyul az ellenfél köveit.

Az *AlphaZero* neurális hálózatának bemenete egy $N \times N \times (MT + L)$ kép, ami az állásokat ábrázolja az $N \times N$-es M síkok T halmazainak láncolásával. Minden halmaz az adott játékállást ábrázolja a $t - T + 1, \ldots, t$ időpontban. Ha az időpont kisebb mint 1, akkor nullákat fog ábrázolni. Szemben az AlphaGo Zeroval, a neurális hálózat mindig csak az aktuális játékosra összpontosít. Az M jellemző síkok tagjai azok a bináris jellemző síkok, melyek a játékos figuráinak helyeit mutatja meg. A sógi és a sakk miatt több síkot használtak a bábuk pozícióinak meghatározásához. A síkok második halmaza az ellenfél játékos figuráinak a helyeit tárolja. Vannak további konstans értékeket tároló síkok is, ezekből összesen L darab van. Ezek a síkok jelölik a játékos színét, a lépések számát és a speciális szabályokat: a sáncolás lehetőségét a sakkban (egy játékos legfeljebb egyszer

sáncolhat egy játékon belül), az aktuális játékállás ismétlődésének számát (3 esetén a sakkban, 4 esetén a sógiban automatikusan döntetlennel zárul a parti) és a lépések számát az előző ütés óta a sakkban (50 lépés előrehaladás nélkül automatikus döntetlent jelent).

Egy lépést a sakkban két részben lehet leírni: először kiválasztjuk azt a figurát, amivel lépni szeretnénk, majd utána kiválasztunk egy lépést a lehetséges lépések közül. A $\pi(a|s)$ policy ábrázolásához 73 darab 8×8-as síkot használtak fel, amely 4.672 lehetséges lépés valószínűségét tárolja. Az első 56 sík a lehetséges vezér lépéseket kódolja, melyet bármely figurára tudunk használni. Az 56 sík a következőképp áll össze: egy bábu legfeljebb 7 mezőt léphet 8 irányba. A következő 8 sík a huszár lehetséges lépéseit kódolja. Az utolsó 9 sík pedig a gyalog átváltozását kódolja úgy, hogy egyet előrelépve vagy az ellenfél figuráját leütve ér be illetve huszárrá, bástyává vagy futóvá változik-e át.

A sógi esetében a policy hálózathoz 139 darab 8×8-as síkot használtak fel, amely összesen 11.259 lehetséges lépés valószínűségét kódolja. Mivel a sógi szabályai nem annyira közismertek, mint például a sakké illetve a szakdolgozatom témája az, hogy a go játékot hogyan tanították be a különböző programoknak, így a jellemző síkok felépítésére külön nem térek ki.

A go játék esetében a policy hálózat ábrázolása megegyezik az AlphaGo Zeroéhoz. A valószínűségeket $19 \times 19 + 1$ méretű (361 lehetséges lépés és a passzolás) vektorban tárolja.

Szempontok	AlphaGo	AlphaGo Zero	AlphaZero
Profi játékosok játék adatbázisa	✓	✗	✗
Játékszabályok ismerete	✓	✓	✓
Bevitt jellemzők száma	48	17	17

2. táblázat. Rendszerbe bevitt információk összefoglaló táblázata.

4.1.2. A programok neurális hálózatai

A következő szempontunk a különböző változatok neurális hálózatainak meghatározása és összehasonlítása.

Az AlphaGo kétféle neurális hálózata ([3]) Az AlphaGo esetében két különböző neurális hálózatot tanítottak be. Az egyik hálózat – a policy hálózat (π) – a lépések kiválasztásáért, míg a másik hálózat – az érték hálózat (v) – a játékállások kiértékeléséért volt felelős. Felügyelt és megerősítéses tanulást használtak a tanítás közben.

A felügyelt tanulás során a fejlesztők egy olyan policy hálózatot használtak, mely megjósolja a profi játékosok lépését a go játékban. A ($p_\sigma(a|s)$) SL[3] policy hálózat egy olyan hálózat, melynek σ a súlya és az utolsó rétege egy softmax réteg, ami kimenetként egy valószínűségi eloszlást ad meg az összes lehetséges a lépésre. A policy hálózat bemenete az s, ami egy egyszerű reprezentációja a táblának.

Az AlphaGo a megerősítéses tanulás közben egy másik policy hálózatot használt. A p_ρ ρ súllyal rendelkező RL[4] policy hálózat szerkezetileg azonos az SL policy hálózattal. A súlyok kezdetben megegyeztek egymással, azaz $\rho = \sigma$.

Az AlphaGo Zero neurális hálózata ([16]) Az AlphaGo Zero a f_θ mély neurális hálózatot használja. Ennek a neurális hálózatnak a bemenete az s nyers táblának a reprezentációja és annak a története és a kimenetele egy (valószínűség vektor, érték) pár, $(p, v) = f_\theta(s)$. A lépések valószínűségének p vektora a különböző lépések kiválasztásának valószínűségét (beleértve a passzolást) tartalmazza, azaz $p_a = Pr(a|s)$. A v érték egy skalár, mely az aktuális játékos győzelmi esélyét becsüli meg az s állásból. Ez a neurális hálózat az AlphaGonál használt policy és value hálózatok összetevésén alapul.

Az AlphaZero neurális hálózata ([18]) Ugyanúgy, ahogy az AlphaGo Zero, az AlphaZero is egy $(p, v) = f_\theta(s)$ mély neurális hálózatot használ. Ennek a $f_\theta(s)$ neurális hálózatnak a bemenete az s állás a táblán és a kimenetelei egy vektor, ami tartalmazza a lépések p valószínűségét, azaz minden a lépésre $p_a = Pr(a|s)$, és egy v skalár, mely meg-

[3]SL: Supervised Learning - felügyelt tanulás

[4]RL: Reinforcement Learning - megerősítéses tanulás

becsüli a z kimenetel várható értékét az s játékállásból, tehát $v \approx \mathbb{E}[z|s]$. Az AlphaZero, ugyanúgy, ahogy az előző változat, az AlphaGo Zero, self-playjel ismerte meg ezeket a p lépési valószínűségeket és a v értéket.

4.1.3. A Monte Carlo Tree Search használata

A harmadik szempont az, hogy a Monte Carlo Tree Search hogyan működik a különböző változatokban.

Keresés policy és érték hálózatokkal (AlphaGo [3]) Az AlphaGo az érték és a policy hálózatokat egy olyan MCTS algoritmussal kombinálja, ami előretekintő kereséssel lépéseket választ ki. A keresőfa minden s csúcsából minden $a \in \mathcal{A}$ legális lépés felé (s, a) élek futnak.

Minden (s, a) él különböző statisztikákat tárol: $P(s, a)$ a valószínűség, $W_v(s, a)$ és $W_r(s, a)$ a Monte Carlo becslések a teljes lépésértékre, $N_v(s, a)$ és $N_r(s, a)$ az érték hálózat és a rollout látogatásának száma és $Q(s, a)$ a kombinált átlagos lépésértéke az adott élnek.

Az APV-MCTS (asynchronous policy and value MCTS) algoritmus négy lépésből áll:

Kiválasztás. Minden egyes szimuláció a keresőfa gyökerétől kezdődik és addig tart, amíg az L időpontban nem éri el a levelet. Minden $t < L$ időlépésnél, a keresőfa statisztikája alapján, azt a lépést választjuk ki, melyre $a_t = \arg\max_a(Q(s_t, a) + u(s_t, a))$, ahol $u(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N_r(s,b)}}{1+N_r(s,a)}$ a PUCT algoritmus egy változata. Ebben a képletben c_{puct} egy konstans, amely a felfedezés szintjét határozza meg. Ez a keresés jobban preferálja azokat a lépéseket, amelyeknek nagyobb a valószínűsége és kisebb a látogatási száma, de aszimptotikusan azokat a lépéseket helyezi előtérbe, amelyeknek magas a lépés értéke.

Kiértékelés. Az s_L levél pozíciót sorba állítja kiértékelésre a $v_\theta(s_L)$ érték hálózat, ha nem volt korábban kiértékelve. Minden szimuláció második rollout szakasza az s_L levélcsúcsból indul és folytatódik egészen addig, amíg véget nem ér a játék. Minden $t \geq L$ időpontban a játékosok lépésének kiválasztása a rollout policy alapján történik: $a_t \sim p_\pi(\cdot|s_t)$. A játék végén a végeredményből kiszámolódik a kimenetel: $z_t = \pm r(s_T)$. Az r a jutalom: győztesnek a jutalma $+1$, a vesztesé pedig -1.

Visszaterjesztés. A szimuláció minden $t \leq L$ időpontjában a rollout statisztika frissül, ha elveszti a n_{vl} mérkőzéseket: $N_r(s_t, a_t)) \leftarrow N_r(s_t, a_t)) + n_{vl}$; $W_r(s_t, a_t) \leftarrow W_r(s_t, a_t) - n_{vl}$. A szimuláció végén a rollout statisztikák visszafelé frissülnek a $t \leq L$ időpontokban.

A virtuális vereségek kicserélődnek a kimenetellel: $N_r(s_t, a_t) \leftarrow N_r(s_t, a_t) - n_{v_l} + 1$; $W_r(s_t, a_t) \leftarrow W_r(s_t, a_t) + n_v l + z_t$. A $v_\theta(s_L)$ érték hálózat kimenetét felhasználták az érték statisztikák frissítéséhez. Ezt egy második visszafelé frissítéssel valósították meg a $t \leq L$ időpontokban: $N_v(s_t, a_t) \leftarrow N_v(s_t, a_t) + 1$, $W_v(s_t, a_t) \leftarrow W_v(s_t, a_t) + v_\theta(s_L)$. Minden él értékét a Monte Carlo becslések átlaga súlyozza: $Q(s, a) = (1 - \lambda)\frac{W_v(s,a)}{N_v(s,a)} + \lambda\frac{W_r(s,a)}{N_r(s,a)}$. Ezek a becslések felhasználják az érték hálózat és a rollout λ paraméterrel súlyozott kiértékeléseit.

Kiterjesztés. Amikor a látogatások száma meghalad egy küszöböt, vagyis $N_r(s, a) > n_{thr}$, egy $s' = f(s, a)$ utódcsúcs hozzáadódik a keresőfához. Az új csúcs $\{N(s', a) = N_r(s', a) = 0, W(s', a) = W_r(s', a) = 0, P(s', a) = p_\sigma(a|s')\}$ statisztikával rendelkezik és egy $p_\tau(a|s')$ policyt használ (ami hasonlót a rollout policyhez, csak több jellemzővel rendelkezik).

Az MCTS működése az AlphaGo Zeroban ([16]) Az AlphaGo Zero már egy jóval egyszerűbb változatát használja az APV-MCTS algoritmusnak, mint amit használt az *AlphaGo Fan* és az *AlphaGo Lee*.

A keresőfa minden s csúcsából minden $a \in \mathcal{A}$ legális lépés felé (s, a) élek futnak. Minden él statisztikákat tárol: $P(s, a)$ a valószínűség, $W(s, a)$ a teljes lépésérték, $N(s, a)$ a látogatások száma és $Q(s, a)$ az átlagos lépésértéke az adott élnek. Több szimuláció futott párhuzamosan a különböző keresési szálakon. Az algoritmus három lépésből áll.

Kiválasztás. A kiválasztási folyamat majdnem megegyezik az *AlphaGo Fanéhoz*. Minden szimuláció a keresőfa s_0 gyökér csúcsától indul és akkor fejeződik be, amikor elér egy s_L levélcsúcsot az L időpontban. Minden $t < L$ időpontban a keresőfa statisztikája alapján kiválasztásra kerül egy lépés: $a_t = \arg\max_a(Q(s_t, a) + U(s_t, a))$, felhasználva a PUCT algoritmus egy változatát:

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)},$$

ahol c_{puct} egy konstans, ami a felfedezés szintjét határozza meg. Összességében ugyanazt a következtetést le tudjuk vonni, mint az AlphaGo esetében: a keresés azokat a lépéseket preferálja, amelyeknek magas a valószínűsége és alacsony a látogatottsága, de aszimptotikusan preferálja azokat a lépéseket, amelyeknek magas a lépésértéke.

Kiterjesztés és kiértékelés. Az s_L levélcsúcs sorba lesz állítva, hogy a neurális hálózat kiértékelje: $(d_i(p), v) = f_\theta(di(s_L)$, ahol d_i egy véletlenszerű tükrözése vagy forgatása az s_L állásnak ($i \in [1 \dots 8]$).

A sorban található pozíciókat a neurális hálózat értékeli ki, 8-as méretű mini-batcheket használva. A keresési szál lezárul addig, amíg a kiértékelés be nem fejeződik. A levélcsúcs terjeszkedik, minden (s_L, a) él $\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_\alpha\}$ értékekkel fog rendelkezni.

Visszaterjesztés. Az él statisztikák visszafelé frissülnek minden $t \leq L$ időpontban. A látogatások számai megnövekednek: $N(s_t, a_t) = N(s_t, a_t) + 1$ és a lépésértéket az átlagértékre frissítik: $W(s_t, a_t) = W(s_t, a_t) + v$, $Q(s_t, a_t) = \frac{W(s_t,a_t)}{N(s_t,a_t)}$.

Alfa-Béta vágás ([12]) Néhány játéknál, köztük a sakknál és a gonál is gyakran használnak Alfa-Béta vágást. Ennek az a lényege, hogy a keresőfánkban levágjuk azokat a részfákat, amiket nem kell vizsgálnunk, hisz ha arra a részfára tévednénk, akkor a játék számunkra vereséggel zárulna. Ezzel tudjuk csökkenteni a kiértékelések számát. Az elnevezés onnan jött, hogy Alfával és Bétával a két játékost jelöljük: az Alfa mindig a legnagyobb értéket választja ki, míg a Béta mindig a legkisebbet. Az értékek, a játék előrehaladásával, folyamatosan frissülnek.

Hatékony keresés MCTS-sel az AlphaZeroban ([18]) Az Alfa-Béta keresés helyett az AlphaZero is MCTS algoritmust használ.

Minden (s, a) él különböző statisztikákat tárol: $N(s, a)$ a látogatások száma, a $W(s, a)$ a teljes lépésérték, a $Q(s, a)$ az átlagos lépésérték és a $P(s, a)$ az előzetes valószínűsége annak, hogy az a lépést választjuk az s állásban. Minden keresés az s_{root} gyökértől indul és addig tart, amíg el nem éri az s_L levélcsúcsot az L időpontban. A $t < L$ időpontban azt a lépést választjuk, amire $a_t = \arg\max_a(Q(s_t, a) + U(s_t, a)$, felhasználva a PUCT algoritmus egy változatát:

$$U(s, a) = C(s)P(s, a)\frac{\sqrt{N(s)}}{1+N(s,a)},$$

ahol $N(s)$ a szülő csúcsok látogatásának száma és $C(s)$ a felfedezési ráta, ami a keresési idővel arányosan lassan növekszik: $C(s) = \log((1 + N(s) + c_{base})/c_{base}) + c_{init}$. Ez

az érték lényegében konstans a gyors játékok tanítása közben. Az s_L levélcsúcsot sorba lesz állítva, hogy a neurális hálózat kiértékelje: $(p, v) = f_\theta(s_L)$. A levélcsúcs terjeszkedik, minden (s_L, a) él $\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_\alpha\}$ értékekkel fog rendelkezni. A keresés végén a látogatások számát és az értékeket frissíti az algoritmus a t időpontokban: $N(s_t, a_t) = N(s_t, a_t)+1, W(s_t, a_t) = W(s_t, a_t)+v, Q(s_t, a_t) = \frac{W(s_t,a_t)}{N(s_t,a_t)}$. A keresés eredménye egy π vektor, mely a lépések valószínűségét tartalmazza: $\pi_a = Pr(a|s_{root})$

Megjegyzés. Észrevehető, hogy az AlphaGo Zero egy olyan változót használ (d_i), ami az adott állás egy tükrözését vagy forgatását veszi figyelembe. Ennek az az oka, hogy a go játék invariáns a forgatásra és a tükrözésre (ezt a tulajdonságát az AlphaGo is kihasználja). Az AlphaZero ezzel szemben, mivel a sakk és a sógi egy aszimetrikus játék (például a gyalog csak előrefelé vagy átlósan léphet, ha leüt egy másik figurát), nem ismeri a szimmetriát, így nem is transzformálja a különböző játékállásokat az MCTS során.

Legalább négy évtizeden át a legerősebb sakk programok Alfa-Béta keresést használtak. Azok a sakk programok, amik hagyományos MCTS-t használnak, kevésbé voltak hatékonyak, mint az Alfa-Béta keresést használó programok. Az AlphaZero meghaladta ezeknek a programoknak a teljesítményét, mivel hatékonyan kombinálta az MCTS-t a neurális hálózatokkal. Az AlphaZero nagy teljesítményét és eredményeit látva (amit a 4.1.7 részben részletezek) megkérdőjeleződött az a széles körben elterjedt hit, hogy az Alfa-Béta keresés hatékonyabb ezekben a játékokban, mint az MCTS.

Szempontok	AlphaGo	AlphaGo Zero	AlphaZero
Legnagyobb $(Q(s,a) + U(s,a))$ értéket választja	✓	✓	✓
PUCT algoritmus használata	✓	✓	✓
Állás transzformálása keresés közben	✓	✓	✗

3. táblázat. A különböző változatok MCTS felhasználásának összefoglaló táblázata.

4.1.4. Tanítás

Miután megnéztük, hogy milyen információkat vittek be a fejlesztők a különböző verzióknak, megismertük a programok neurális hálózatait és megnéztük, hogyan működik az MCTS az egyes változatokban, a negyedik szempontunk az, hogyan tanították be az egyes változatokat a fejlesztők.

AlphaGo - A policy hálózat felügyelt tanulása ([3]) Az SL policy hálózatot véletlenszerű (s, a) állás-lépés pár mintavételezésével tanították. Ehhez sztohasztikus gradiens emelkedést használtak, hogy maximalizálják az ember által meglépett a lépés valószínűségét egy adott s állásban:

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

A tanításhoz a már korábban (4.1.1 részben) említett adathalmazt használták fel, amivel egy 13 rétegű policy hálózatot tanítottak be. Ezt az adathalmazt szétválasztották teszt halmazra (első 1 millió állás) és tanító halmazra (a maradék 28.4 millió állás). A tanításra a sztochasztikus gradiens ereszkedő módszert használták fel, tanulási rátának $\alpha = 0.003$-t adtak meg, illetve a minibatchek mérete m = 16 volt. A tanítás körülbelül 3 hétig tartott. A policy hálózat, mely az összes bemeneti jellemzőket használja, a teszthalmazon 57% pontossággal jósolta meg a profi játékosok lépéseit. Amikor bemenetként nem adták meg ezeket a jellemzőket és csak a korábbi lépések voltak megadva, az AlphaGo 55.7% pontossággal jósolt. Összehasonlításképp, a többi, korszerű technológiát használó kutatócsoport csak 44.4%-os pontosságot ért el a cikk benyújtásának időpontjában. Kisebb fejlődés a pontosságban nagyobb fejlődéshez vezetett a játék erősségét tekintve. Nagyobb hálózatok jobb pontosságot értek el, viszont lassabban értékeltek ki a keresés közben. A fejlesztők betanítottak egy gyorsabb, de pontatlanabb rollout policyt $p_\pi(a|s)$, mely 24.2% pontosságot ért el, $2\mu s$ alatt választott ki egy lépést, szemben a policy hálózattal, melynek 3 ms kellett egy lépés megtételéhez.

AlphaGo - A policy hálózat megerősítéses tanulása ([3]) A megerősítéses tanulás során a fejlesztők fő célja a policy hálózat javítása volt. Az aktuális p_ρ policy hálózat egy véletlenszerűen kiválasztott korábbi iterációval játszott. Az ellenfelek kiválasztásának

ezen módja elősegítette a túltanulás elkerülését. Bevezetésre került továbbá egy jutalom ($r(s)$), melynek értéke 0 volt, ha $t < T$ (t egy adott időpillanat és T időpontban ér véget a játék). Az eredmény az a $z_t = r(s_T)$ végső jutalom a jelenlegi játékos szemszögéből a t időpontban, melynek értéke $+1$, ha az adott játékos nyerne és -1, ha vesztene. A súlyokat minden t időpontban frissítik a sztochasztikus gradiens emelkedő módszerrel. A cél az volt, hogy maximalizálják a várható eredményt:

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial\rho} z_t$$

Egymás elleni mérkőzéseken az RL policy hálózata a játékok több mint 80%-át megnyerte az SL policy hálózat ellen. Letesztelték a legerősebb nyílt forráskódú go program, a Pachi ellen is, egy olyan szofisztikált Monte Carlo kereső program ellen, ami 100 ezer szimulációt hajt végre lépésenként. Keresés nélkül a megerősítéses tanulásos policy hálózat a játékok 85%-át nyerte meg a Pachi ellen. Összehasonlításképp, a korszerű, csak a felügyelt tanulással tanított policy hálózat a játékok 11%-át nyerte meg a Pachi ellen. Még az alig gyengébb Fuego ellen is csak a partik 12%-án nyert.

AlphaGo - Az érték hálózatok tanítása ([3]) Az érték hálózatokat megerősítéses tanulással tanították. Az érték hálózatok a $v^p(s)$ értékfüggvényt becsülik meg, ami a játék kimenetelét jósolja meg az s pozícióból úgy, hogy mindkét játékos a p policyt használja:

$$v^p(s) = \mathbb{E}[z_t|s_t = s, a_{t...T} \sim p]$$

Ideális esetben azt szeretnénk tudni, hogy a $v^*(s)$ tökéletes játéknak mi az optimális értékfüggvénye. Ennek ellenére a fejlesztők a legerősebb v^{p_ρ} policyjuk értékfüggvényét becsülik meg, felhasználva a p_ρ RL policy hálózatot. Az értékfüggvény közelítéséhez egy θ súlyokkal rendelkező $v_\theta(s)$ érték hálózatot használtak fel, melyre $v_\theta(s) \approx v^{p_\rho}(s) \approx v^*(s)$. Ennek a neurális hálózatnak hasonló a szerkezete, mint a policy hálózatnak, viszont a kimenete egy egyszerű jóslat és nem egy valószínűség. Az érték hálózat θ súlyainak tanításához (s, z) állás-kimenetel párokra illesztettek regressziót. Sztochasztikus gradiens ereszkedő módszert használtak, mellyel az átlagos négyzetes hibát minimalizálták a $v_\theta(s)$ jósolt érték és a z megfelelő kimenetel között:

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$$

AlphaGo Zero tanítása self-playjel ([16]) Az AlphaGo Zero esetében a 4.1.2 részben már említett f_θ mély neurális hálózatot használták. Az f_θ neurális hálózatot egy self-play megerősítéses tanulás algoritmus tanította, ami MCTS-t használt arra, hogy lépjen. Először is, a neurális hálózatnak adtak egy véletlenszerű súlyt: θ_0. Minden $i \geq 1$ iterációkor saját játékok jönnek létre. Minden t időpontban végrehajtódik az MCTS, $\pi_t = \alpha_{\theta_{i-1}}$, amely a neurális hálózat előző iterációját ($f_{\theta_{i-1}}$) használja és egy lépést a keresési valószínűségek (π_t) mintavételezésével tesz meg.

A játék a T lépésnél ér véget, amikor mindkét játékos passzol, amikor a keresési érték egy küszöb alá esik le, vagy amikor a játék meghalad egy maximális hosszúságot. Az AlphaGo Zero esetén $19 \cdot 19 \cdot 2 = 722$ lépésben maximálták egy játéknak a hosszát. Ha a program elérte ezt a lépésszámot, a kiértékelés Tromp–Taylor pontozással történt. A játék végén, a szabályoknak megfelelően, a jutalmak kiosztásra kerülnek. Ugyanúgy, ahogy az AlphaGonál, itt is $+1$ jár a győztesnek és -1 a vesztesnek.

Az adatokat minden t időpontban egy (s_t, π_t, z_t) rendezett hármasban tárolja, ahol a $z_t = \pm r_T$ a játék győztese az aktuális játékos szemszögéből a t időpontban. Ezzel párhuzamosan az új θ_i hálózat paraméterek az (s, π, z) halmazból tanulnak, mintavételezve az előző iterációk összes időpontjában tárolt adatait. A $(p, v) = f_{\theta_i}$ neurális hálózat úgy van beállítva, hogy minimalizája a hibát a v jósolt érték és a z tényleges érték között és maximalizálja a neurális hálózat p lépés valószínűségének hasonlóságát a π keresési valószínűséghez. A θ paramétereket gradiens ereszkedéssel állították be, melyhez egy olyan l veszteségfüggvényt használtak, mely az átlagos négyzetes hibát és a kereszt-entrópia veszteséget adja össze:

$$l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2, \tag{1}$$

ahol a c paraméter a túltanulás elkerüléséért felel.

A tanítás teljesen véletlenszerű viselkedésekkel kezdődött és emberi beavatkozás nélkül körülbelül három napig tartott. A tanítás során 4.9 millió játékot játszott le a program, minden MCTS-re 1600 szimulációt hajtott végre, ami megfelel körülbelül 0.4 másodperc gondolkodási időnek lépésenként. A paramétereket összesen 700 ezer lépésben tanították, mini-batchenként 2048 pozícióval. Meglepetésre az AlphaGo Zero 36 óra tanulás után túlteljesítette az AlphaGo Lee-t, amit korábban hónapokon át tanítottak. Az AlphaGo

Zero 4 TPU[5]-s gépet használt, szemben az AlphaGo Leevel, amit több gépen osztottak meg. Ezek a gépek összesen 48 TPU-t használtak.

Azért, hogy megmutassák a self-play megerősítéses tanulás előnyeit, betanítottak egy második neurális hálózatot, mely szerkezetileg azonos az előzővel és a KGS adathalmaz lépéseit jósolta meg. A felügyelt tanulás jobb kezdeti teljesítményt ért el és jobban meg tudta jósolni a profi játékosok által játszott játékok végkimenetelét. Hiába volt pontosabb a felügyelt tanulás, a „self-play" játékos összességében sokkal jobban teljesített, legyőzve az ember által tanított játékost a tanítás első 24 órájában.

Azzal, hogy az AlphaGo Zero fejlettebben gondolkodott, olyan lépéseket is látott, amiket ember nem. Számtalan, eddig ismeretlen variációt fedezett fel a program. Gyorsan megértette a különböző go fogalmakat, többek között a nyitást (fuseki), a taktikákat (tesuji), az ismétlődő játékállásokat (ko) és a végjátékot (yose). Érdekes módon a létrát - az egyik első ismert elemét a go-nak - csak tanítási folyamat egy sokkal későbbi szakaszában értette meg.

Történt egy második tanítása is az AlphaGo Zeronak, ahol már nagyobb neurális hálózatokat használtak és hosszabb ideig tartott. A tanítás közben a program megint teljesen véletlenszerűen viselkedett és a tanítása körülbelül 40 napig tartott. A tanítás során 29 millió játékot játszott le a program. A paramétereket 3.1 millió lépésen tanították, minibatchenként 2048 pozícióval. (Az AlphaGo Zero további elért eredményeit a 4.1.7 részben részletezem.)

AlphaZero betanítása három játékra ([18]) Az f_θ mély neurális háló θ paramétereit megerősítéses tanulással tanították, ehhez ugyanúgy saját játékokat használtak, mint a korábbi változatban. A θ paramétereket kezdetben véletlenszerűen adták meg. Minden játék t körében az MCTS futtat egy keresést a $s_{root} = s_t$ állásból és választ egy lépést, $a_t \sim \pi_t$, ami vagy arányosan (hogy felfedezze a keresőfát) vagy mohón (hogy a lehető legjobb lépést válassza) veszi figyelembe a látogatások számát.

A játéknak akkor van vége, ha mindkét játékos passzol (ez csak a go esetében lehetséges), vagy elértek egy előre meghatározott lépésszámot. A sógi és a sakk játékban 512 lépés

[5]Tensor Processing Unit - Google által kifejlesztett processzor, ami felgyorsítja a neurális hálózatok gépi tanulását

után a játék döntetlennel ér véget, míg a go esetében ugyanúgy 722 lépésre van lelimitálva a játék. Ez utóbbi esetben a játék kimenetelének értékelése egy bizonyos Tromp–Taylor szabállyal történik (ugyanúgy, mint az AlphaGo Zero esetében). A játék végén a s_T utolsó állásban az algoritmus, a játék szabályainak megfelelően, kiszámolja a játék kimenetelét (z): -1 jár a vereségért, 0 a döntetlenért és $+1$ a győzelemért.

Megjegyzés. Megfigyelhetjük, hogy az előző két változattal ellentétben az AlphaZero már ismeri a döntetlent, ennek 0 értéket ad: ez azért nagyon fontos, mert bár a goban továbbra sem lehetséges a döntetlen (bizonyos szabályok szerint), a sakkban és a sógiban lehetséges, hogy az adott játéknak nincs győztese. Sőt, úgy hiszik, hogy a sakknál az optimális kimenetel a döntetlen. Így tehát az AlphaZero, az AlphaGo Zeroval ellentétben, nem a győzelem valószínűségét becsüli meg és optimalizálja, hanem a várható kimenetelt.

A neurális hálózat θ paramétereit úgy állították be, hogy az minimalizálja a hibát a jósolt kimenetel (v_t) és a tényleges kimenetel (z) között, és naximalizálja a p_t policy vektor hasonlóságát a π_t keresési valószínűségeket tartalmazó vektorhoz. Ugyanúgy, mint az AlphaGo Zero esetében, az AlphaZeronál is a θ paramétereket gradiens ereszkedő módszerrel tanították, ahol az l veszteség függvény szintén az átlagos négyzetes hiba és a kereszt-entrópia veszteség összege: (1).

Az AlphaGo Zero esetében a saját játékokat az összes előző iteráció legjobb játékosa generálja. Minden iteráció végén összemérték az új játékos teljesítményét a legjobb játékossal. Az új játékos akkor cserélte le a legjobb játékost, ha legalább 55%-os előnnyel nyert az ellenfelével szemben. Az AlphaZero ezzel szemben egy olyan neurális hálózatot tart fent, melynek a paraméterei folyamatosan frissülnek, nem várja meg az iteráció végét. A saját játékokat a neurális hálózat legutolsó paramétereit felhasználva generálja.

Az AlphaZerot külön tanították be a sakkra, a sógira és a Gora. A tanítás 700 ezer lépésig tartott (minibatchenként 4096 pozícióval), véletlenszerűen meghatározott paramétekekkel kezdődve. Csak a tanítás során 5000 első generációs TPU-t (ezekkel generálták a játékokat) és 16 második generációs TPU-t (ezekkel tanították a neurális hálózatokat) használtak. A betanított AlphaZero összesen 4 első generációs TPU-t és 44 CPU-t használt. A tanítása sakkban 9 órán át, sógiban 12 órán át és Goban 13 napon át tartott.

Sakkban az AlphaZero először 4 óra (300 ezer lépés) után múlta felül a Stockfisht,

Sógiban az Elmot először 12 óra (110 ezer lépés) után teljesítette túl, míg az AlphaGo Leet először 30 óra (70 ezer lépés) után szárnyalta túl. A tanító algoritmus minden, egymástól független futás során hasonló teljesítményt nyújtott, utalva arra, hogy az AlphaZero tanító algoritmusának magas teljesítménye ismételhető. (Az AlphaZero további sikereit a 4.1.7 részben mutatom be.)

Szempontok	AlphaGo	AlphaGo Zero	AlphaZero
Betanított neurális hálózatok száma	2	1	1
Csak megerősítéses tanulás	✗	✓	✓
Self-play	✓	✓	✓
Limitált játék	✗	✓	✓
Legjobb játékos használata	✗	✓	✗
Döntetlen ismerete	✗	✗	✓

4. táblázat. A különböző változatok tanításának összefoglaló táblázata.

4.1.5. Játékállások reprezentálása konvolúciós hálókkal

Ebben a részben összehasonlítom a különböző változatok konvolúciós hálózatának felépítését.

AlphaGo ([3]) A policy hálózat bemenete egy $19 \times 19 \times 48$ kép, amely 48 síkot tartalmaz. Az első rejtett réteg nullákkal párnázza ki a kép szélét és így egy 23×23-as képet fog vizsgálni. Ezután k darab 5×5-ös szűrő egyesével lépkedve végigmegy a képen és a végén használja a ReLU aktivációs függvényt. A 2-12. rejtett rétegek mindegyike szintén nullákkal párnázza ki a kép szélét, viszont itt már csak 21×21-es képet vizsgál, melyen k darab 3×3-as szűrő megy keresztül, szintén egyet lépkedve, ReLU függvényt használva. Az utolsó rétegen egyesével halad végig 1 darab 1×1-es szűrő, különböző befolyással vizsgál minden pozíciót és egy softmax függvényt használ. Az AlphaGo végső verziója k = 192 filtert használt.

Az érték hálózat bemenete is egy $19 \times 19 \times 48$ kép, de a policy hálózattal ellentétben tartalmaz egy olyan bináris jellemző síkot, mely a jelenlegi játékos színét írja le. A 2-11. rejtett réteg megegyezik a policy hálózatéval, a 12. rejtett réteg egy további konvolúciós réteg, 13. rejtett rétegen 1 darab 1×1-es szűrő megy végig, egyesével lépkedve és a 14. rejtett réteg egy teljesen összekötött réteg 256 rectifier unittal. A kimeneti réteg egy teljesen összekötött réteg egy darab tanh unittal.

A policy és az érték hálózat szerkezete a 5. ábrán látható.

AlphaGo Zero ([16]) Az AlphaGo Zero olyan konvolúciós neurális hálózat szerkezetet használ, mely jól illeszkedik a Gora: a játék szabályai invariánsak és meg vannak határozva a kövek életei, azaz egy kőnek vagy egy csoportnak hány szabad szomszédos pontja van.

A bemeneti jellemzőket (s_t) egy reziduális torony dolgozza fel, amely egy konvolúciós blokkot és 19 vagy 39 reziduális blokkot tartalmaz.

A konvolúciós blokk a következő modulokat alkalmazza: 256 darab 3×3-as szűrő egyesével lépkedve végighalad a bemeneten, végrehajt egy batch normalisationt, majd alkalmazza a ReLU aktivációs függvényt.

Minden reziduális blokk a következő modulokat alkalmazza a bemenetére: 256 darab 3×3-as szűrő egyesével lépkedve végigtapogatja a bemenetet, végrehajt egy batch normalisationt, majd alkalmazza a ReLU aktivációs függvényt. A folyamatot még egyszer megismétli, csak a batch normalisation után a reziduális blokkhoz hozzáadja a bemenetet és utána alkalmazza a ReLU aktivációs függvényt.

A reziduális torony kimenete két különálló „head”-re válik szét, melyek a policyt (π) és az értéket (v) számolják ki. A policy head a következő modulokat alkalmazza: 2 darab 1×1-es szűrő egyet lépve letapogatja a bemenetet, végrehajt egy batch normalisationt és a végén használja a ReLU aktivációs függvényt. Ezután egy teljesen összekötött réteg egy $19^2 + 1 = 362$ méretű vektort ad kimenetként, amely a tábla összes pontjának és a passzolásnak ad egy valószínűséget.

A value head az alábbi modulokat alkalmazza: 1 darab 1×1-es filter egyesével lépkedve végigmegy a bemeneten, végrehajtja a batch normalisationt, majd alkalmazza a ReLU aktivációs függvényt. Egy teljesen összekötött réteg össze van kötve egy 256-os méretű rejtett réteggel, majd alkalmazza rá a ReLU aktivációs függvényt. A teljesen összekötött réteg össze van kötve egy skalárral, majd ezt a skalárt a tanh aktivációs függvény kimenetként adja vissza, melynek értéke -1 és 1 között van.

A hálózat összesen 44 vagy 84 rétegből áll, attól függően, hogy 20 vagy 40 blokkból áll a hálózat. A reziduális toronyban 39 vagy 79 paraméterezett réteg van illetve további két rétegből áll a policy head és háromból a value head.

Négy különböző neurális hálózatot hasonlítottak össze:

1. *dual-res:* A hálózat reziduális tornya 20 blokkot tartalmaz, melyet policy head és value head követ. Ezt a szerkezetet használják az *AlphaGo Zeronál.*

2. *sep-res:* A hálózat két, egyenként 20 blokkból álló reziduális tornyot tartalmaz. Az első tornyot a policy head, a másodikat a value head követi.

3. *dual-conv:* A hálózat egy 12 konvolúciós blokkból álló, nem reziduális tornyot tartalmaz, amit policy head és value head követ.

4. *sep-conv:* A hálózat két, egyenként 12 konvolúciós blokkból álló, nem reziduális

tornyot tartalmaz. Az első tornyot a policy head, a másodikat a value head követi. Ezt a szerkezetet használják az *AlphaGo Leenél*.

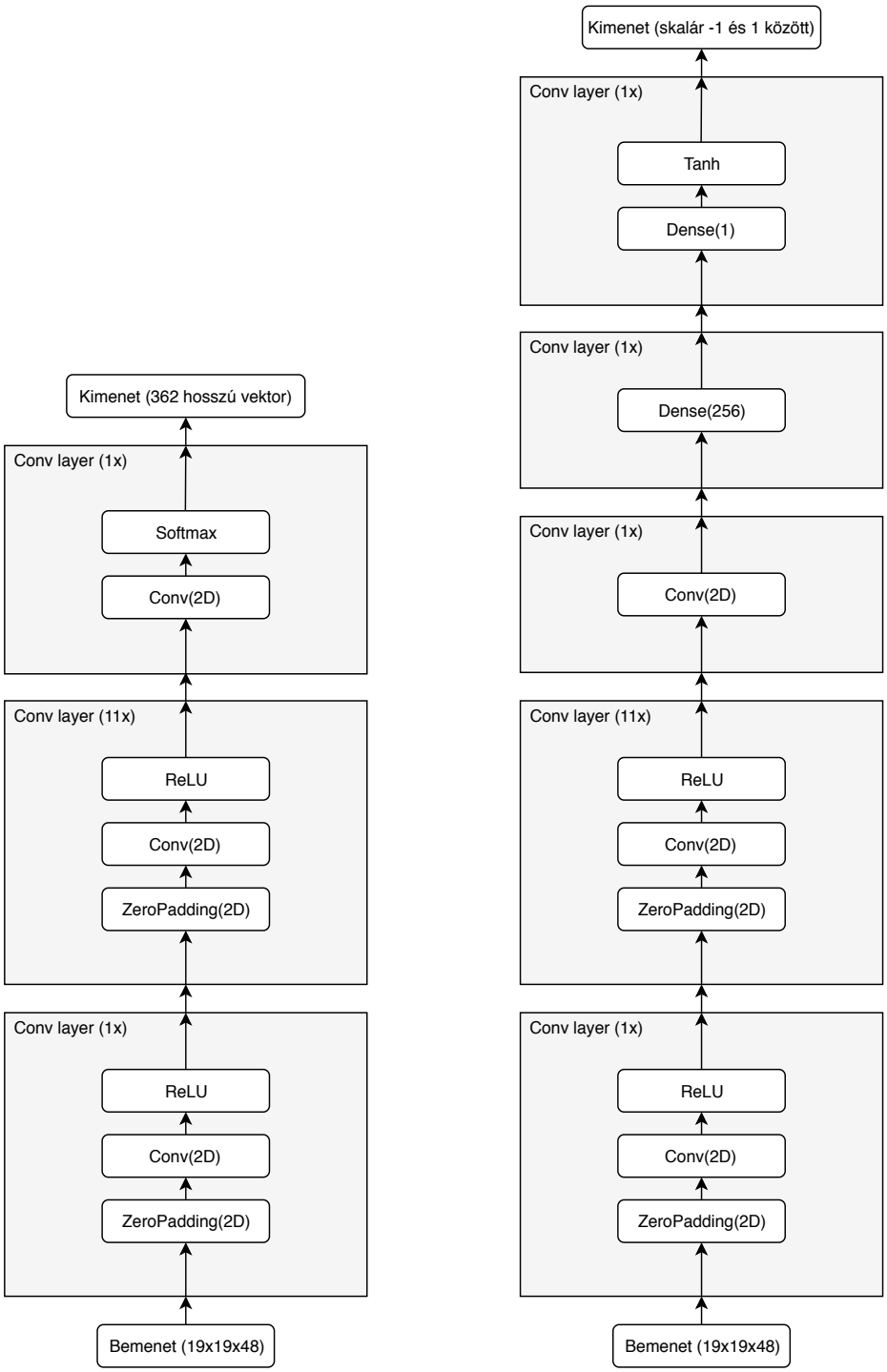
AlphaZero ([18]) A goval ellentétben a sakk és a sógi szabályai pozíció függőek (például a gyalogok a második sorból két lépést is előre tudnak lépni, illetve átváltozik a nyolcadik sorban) és vannak köztük hosszú távú interakciók is (például a futó, a bástya vagy a vezér egy lépéssel át tud haladni az egész táblán). Ezeknek a különbségeknek ellenére az AlphaZero ugyanazt a konvolúciós hálózat szerkezetet használja a három játéknál, mint az AlphaGo Zero.

Az AlphaGo Zero és az AlphaZero szerkezete a 6. ábrán látható.

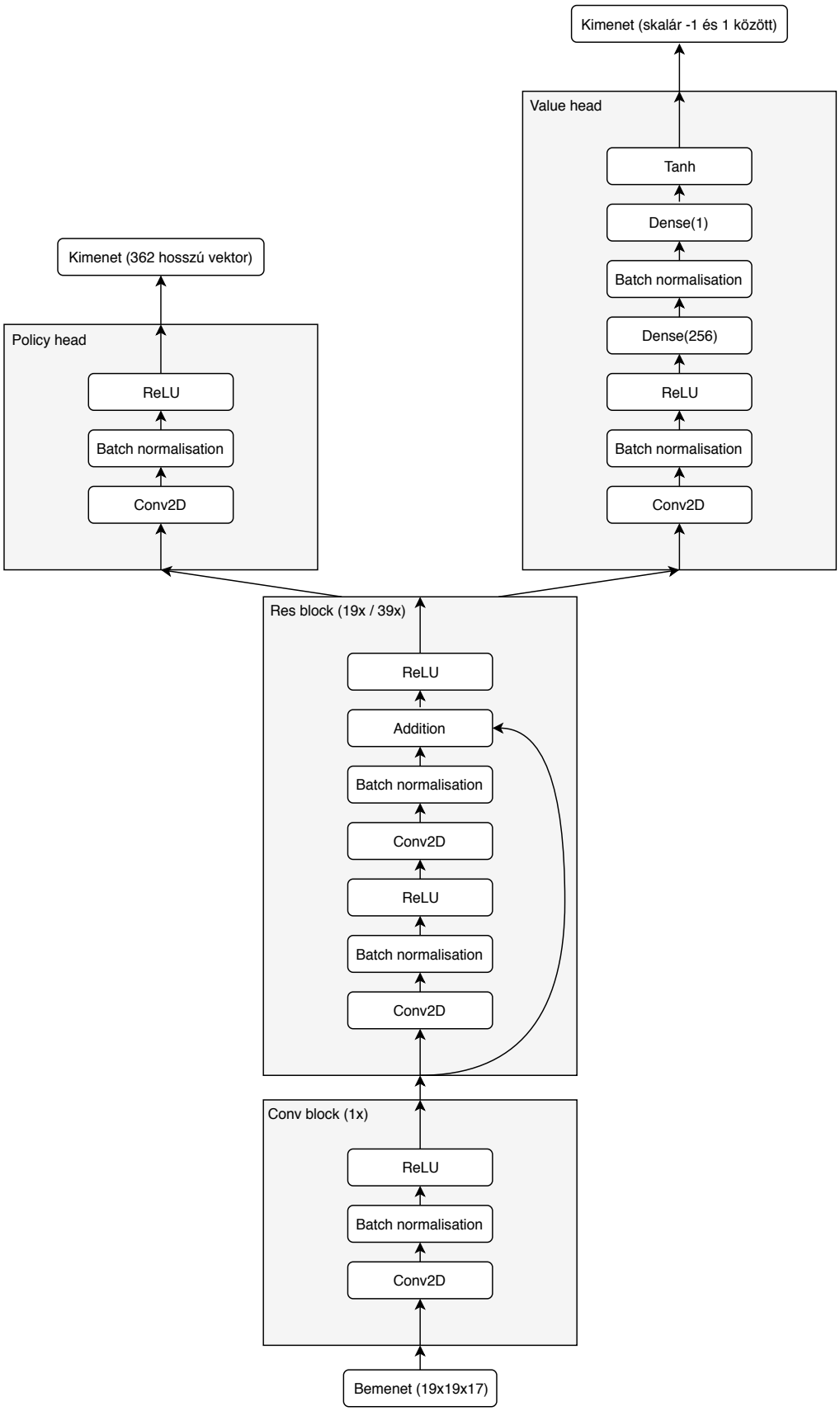
Szempontok	AlphaGo	AlphaGo Zero	AlphaZero
Tornyok száma	2	1	1
Reziduális blokkot tartalmaz	✗	✓	✓
Szétválik „head”-ekre	✗	✓	✓
Rétegek száma	28[6]	44	44

5. táblázat. A változatok konvolúciós hálózatainak összefoglaló táblázata.

[6] Összesen a két hálózatnak



5. ábra. Az AlphaGo policy (balra) és az érték hálózatának (jobbra) struktúrája.



6. ábra. Az AlphaGo Zero és az AlphaZero konvolúciós hálózatának struktúrája.

4.1.6. A programok kiértékelése

Ebben a részben azt szeretném bemutatni, hogyan értékelték ki a fejlesztők a programokat.

Az AlphaGo kiértékelése ([3]) Azoknak a go programoknak az erősségét, amelyekkel az AlphaGo játszott, úgy határozták meg, hogy egy belső versenyt indítottak a programoknak, majd lemérték a programok Élő pontszámát. Annak a valószínűségét, hogy az a program megveri a b-t, egy logisztikus függvénnyel határozták meg:

$$\text{p(a megveri b-t)} = \frac{1}{1+exp(c_{elo}(e(b)-e(a))}.$$

Az $e(\cdot)$ pontszámokat Bayes logisztikus regresszióval becsülték meg, amit a BayesElo program számolt ki. A konstansnak $c_{elo} = 1/400$ értéket adtak meg. A skálán, amin használtak az Élő pontszám meghatározásához, rögzítésre került Fan Hui profi játékos Élő pontszáma (2.908).

Minden program legfeljebb öt másodperc gondolkodási időt kapott lépésenként. A pontozás kínai szabályok szerint történt, kominak 7.5 pontot határoztak meg. Fontos volt, hogy 7.5 pont legyen a fehér játékos előnye, ugyanis az érték hálózatot eszerint tanították be. (Azt, hogy az AlphaGo végül milyen eredményeket ért el, a 4.1.7. részben részletezem)

Az AlphaGo Zero kiértékelése ([16]) Az AlphaGo Zerot hasonló módon értékelték ki, mint az AlphaGot: ugyanazzal a logisztikus függvénnyel számoltak és a pontokat ugyanúgy Bayes logisztikus regresszióval becsülték meg, amit a BayesElo program számolt ki. Konstanstnak ugyanúgy $c_{elo} = 1/400$ értéket adtak meg.

Az Élő pontszámokat egy olyan torna eredményeiből számították ki, ami az AlphaGo Zero, az AlphaGo Master, az AlphaGo Lee és az AlphaGo Fan között zajlott. Az AlphaGo Zero nyers neurális hálózata is részt vett a tornán. Minden program öt másodperc lépési időt kapott. Az AlphaGo Fan, a CrazyStone, a Pachi és a GnuGo Élő pontszámát rögzítették és ezek is befolyásolták a torna értékét. Az *AlphaGo Fan* Fan Hui és az *AlphaGo Lee* Lee Sedol elleni meccseinek eredményei szintén rögzítésre kerültek, máskülönben az AlphaGo Élő pontszáma valótlanul magas lett volna a self-play eltolások miatt.

Az AlphaZero kiértékelése ([18]) Az AlphaZero relatív erősségét is Élő pontszámmal határozták meg. Annak a valószínűségét, hogy az a játékos legyőzi a b-t egy logisztikus függvénnyel becsülték meg, ám ez a függvény eltér a korábbi két verziónál alkalmazottnál:

$$\text{p(a megveri b-t)} = (1 + 10^{(c_{elo}(e(b)-e(a)))})^{-1}.$$

Az $e(\cdot)$ pontokat ugyanúgy Bayes lineáris regresszióval becsülték meg és ezeket ugyanúgy a BayesElo program számolta ki, mint az AlphaGonál és az AlphaGo Zeronál. Konstansnak szintén $c_{elo} = 1/400$ értéket állítottak be.

Az Élő pontszámokat egy olyan torna eredményeivel számították ki, ahol a programok egy másodperc időt kaptak lépésenként. A tornán az AlphaZero iterációi játszottak az egyik alapjátékossal: a Stockfishsel, az Elmoval vagy az AlphaGo Leevel. Az alapjátékosok Élő pontszámát nyílvánosan elérhető értékek alapján rögzítették.

4.1.7. Az elért eredmények

Ebben a részben azt szeretném bemutatni, milyen eredményeket értek el a programok.

AlphaGo nemzetközi sikerei ([3]) Ahogy a szakdolgozatomban már többször is utaltam rá, az AlphaGo volt az első olyan go program, amely előnykő nélkül győzött le profi játékost: a háromszoros Európa bajnok, 2 danos Fan Hui 5:0-ra kapott ki az AlphaGotól. A győzelem teljesen váratlan volt, hisz a szakemberek úgy gondolták, hogy ez az áttörést hozó győzelem még legalább egy évtizedre van.

Az AlphaGo megmérkőzött a többi go programmal is, köztük a CrazyStonennal és a Zennel, illetve a legerősebb nyílt forráskódú programokkal, a Pachival és a Fuegoval. A lejátszott 495 mérkőzésből csupán egyet veszített el az AlphaGo. Hogy az AlphaGo-t még nagyobb kihívás elé állítsák, olyan mérkőzéseket játszattak a fejlesztők fent említett programok ellen, ahol négy előnykövet adtak az ellenfélnek. A meccsek 77%, 86% és 99%-át megnyerte a Crazy Stone, a Zen és a Pachi ellen.

Az AlphaGo több gépen megosztott verziója egyértelműen erősebb volt. Az egygépes AlphaGo ellen a mérkőzések 77%-át, a többi go program ellen viszont az összes mérkőzését megnyerte.

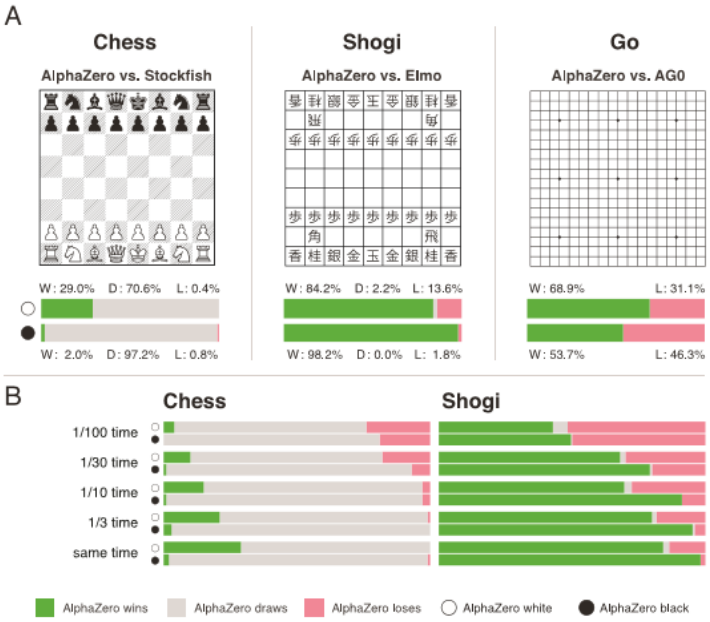
AlphaGo Zero fontosabb eredményei ([16]) Mindössze három nap self-play után az AlphaGo Zero 100:0 arányban legyőzte az AlphaGo Leet - azt a programot, ami legyőzte a 18-szoros világbajnok Lee Sedolt.

Az AlphaGo Zero, hogy a fejlesztők kiértékelhessék a teljesen betanított programot, mérkőzéseket játszott az AlphaGo Fan, az AlphaGo Lee és számos korábbi go program ellen. A fejlesztők minden programnak lépésenként 5 másodperces gondolkodási időt engedélyeztek. De nem csak ezek ellen a programok ellen játszott, hanem megmérkőzött az akkori legerősebb go program, az AlphaGo Master ellen is. A szerkezetükben és az algoritmusukban azonos, de tanításukban más programok (az AlphaGo Mastert felügyelt tanulással tanították) egy 100 mérkőzéses csata keretén belül csaptak össze, melyet az AlphaGo Zero 89:11 arányban nyert meg.

Már ennél a pontnál megfigyelhető, mennyivel erősebbek voltak a továbbfejlesztett

változatok az előzőeknél. Míg az AlphaGo Fan 3144, az AlphaGo Lee 3739, az AlphaGo Master 4858 Élő ponttal rendelkezett, addig az AlphaGo Zeronak 5185 Élő pontja volt.

Az AlphaZero három játékban elért eredménye ([18]) Az AlphaZerot három játékra tanították be: sakkra, sógira és gora. Sakkban a Stockfishsel, sógiban az Elmoval, míg goban az AlphaGo Zeroval játszott. Mindhárom program akkoriban a legerősebb programnak számított (a különböző játékokban).



7. ábra. Az AlphaZero által elért eredmények [18].

Goban az AlphaZero a játékok 61%-át nyerte meg az AlphaGo Zero ellen.

Sakkban az AlphaZero 1000 játékból 155-t nyert meg és csak 6-ot veszített el a Stockfish ellen. További játékokban, ahol gyakori nyitásokból kezdték a meccset, az AlphaZero minden nyitásból nyert partit (ez nem igaz a Stockfishre). Néhány játékban az AlphaZero feláldozott figurákat a hosszútávú stratégiai előnyért cserébe, ami arra utal, hogy a helyzeti kiértékelés sokkal folyékonyabb és kontextus függőbb, mint a szabályokon alapuló kiértékelés, amit a korábbi sakk programok használtak.

Sógiban az AlphaZero a mérkőzések 98.2%-át megnyerte az Elmoval szemben, amikor a feketével játszott (vagyis kezdett), összességében pedig a mérkőzések 91.2%-án nyert.

Az AlphaZero a sakkban és a sógiban másodpercenként csak 60 ezer lépést keres meg,

szemben a Stockfishsel, ami 60 milliót, míg az Elmo 25 milliót vizsgál meg. Az AlphaZero az alacsony számú kiértékelést a mély neurális hálózatának használatával kompenzálja, melynek segítségével jobban tud fókuszálni az ígéretesebb lehetőségekre. Az AlphaZero legyőzte úgy is a Stockfisht, hogy csak 1/10 gondolkodási időt kapott az ellenféltől és a mérkőzések 46%-át megnyerte Sógiban, amikor 1/100 gondolkodási ideje volt.

Ezek az eredmények leolvashatóak a 7. ábráról.

4.2. Összefoglaló táblázatok

Ebben az alfejezetben további két, a fejezethez kapcsolódó, összefoglaló táblázat kerül bemutatásra.

A 6. táblázat a verziók tanulási folyamatát és hatékonyságát hasonlítja össze. A 7. táblázat pedig a 4.1.7 részben megemlített fontosabb mérkőzéseket gyűjti össze, így könnyen átláthatjuk, hogy rövid időn belül milyen sikeres volt egy adott verzió.

Szempontok	AlphaGo([3])	AlphaGo Zero([16])	AlphaZero([18])
Felhasznált erőforrás	48 CPU, 8 GPU (egygépes)[7]	64 GPU, 19 CPU, 4 TPU	44 CPU, 4 TPU
Tanulás (idő)	Több hónap	kb 40 nap	Sakk: 9 óra Sógi: 12 óra Go: 13 nap
Tanulás (lépés)	340 millió	3.1 millió	700 ezer
Élő pont	3144[8]	5185	nincs adat (kb 5300)

6. táblázat. A különböző verziók főbb tulajdonságait összefoglaló táblázat.

[7] A megosztott változat 1202 CPU-t és 176 GPU-t használ.

[8] Amikor legyőzte Fan Huit.

Dátum	Verzió	Ellenfél	Eredmény
2015.	AlphaGo (egygépes)	Több go program	494:1
2015.	AlphaGo (megosztott)	AlphaGo (egygépes)	77%-23%
2015. október	AlphaGo Fan	Fan Hui	5:0
2016. március	AlphaGo Lee	Lee Sedol	4:1
2016. dec. vége - 2017 jan. eleje	AlphaGo Master	Online (profi) játékosok	60:0
2017.	AlphaGo Zero	AlphaGo Lee	100:0
2017.	AlphaGo Zero	AlphaGo Master	89:11
2017. december	AlphaZero	AlphaGo Zero	61%-39%
2017. december	AlphaZero	Stockfish	155:6 839 döntetlen
2017. december	AlphaZero	Elmo	912:77 11 döntetlen

7. táblázat. A változatok eredményeit összefoglaló táblázat.

5. További eredmények

A fejlesztési folyamatok közben a Google megalkotta a saját processzorát, a TPU-t, mely nemcsak felgyorsította a tanulási folyamatot, de lecsökkentette a programok energia felhasználását is: már az AlphaGo Zero is körülbelül 1/48 energiát fogyasztott, mint a több gépen megosztott AlphaGo.

Miután befejezték a versenyzést az AlphaGoval, a DeepMind fejlesztő csapat, felhasználva a fejlesztés során szerzett új tapasztalatokat, elkezdett fontosabb kutatásokban is részt venni. Az AlphaGo mögötti gépi tanulásos módszereket olyan célokra használták fel, mint például, hogyan lehetne csökkenteni az energia használatot egy épületen belül, és részt vettek néhány orvosi kutatásban is. Továbbá felfejlesztették a Google Fotók és a Google Fordító alkalmazásukat is [19][20].

2017. decemberében a fejlesztők megjelenítették az AlphaGo Teachet, mely a 6000 legnépszerűbb nyitást elemzi ki, felhasználva 231 ezer, emberek által, és 75, AlphaGo által lejátszott játék nyitását (körülbelül a parti első 30 lépését). Ezzel mi is meg tudjuk nézni, hogy a játékok elején az egyes állásoknál az AlphaGo melyik lépést választaná ki [21].

A szakdolgozatom írása közben, 2019. október 30-án a Nature publikálta a fejlesztők legújabb programját, az AlphaStart. Az AlphaStar a StarCraft II nevű játékban ért el olyan eredményt, amire kevés ember volt képes: feljutott a legmagasabb ligába, a Grandmaster Leaguebe. Bár ez a program nem verhetetlen, mégis jobb, mint a játékosok 99.8%-a [22].

Szintén a szakdolgozatom írása közben, 2019. november 19.-én publikálták a MuZero nevű programot, mely az AlphaZeroval ellentétben úgy sajátította el a go, a sógi és a sakk játékokat, hogy egyáltalán nem ismerte a szabályokat. A program 57 különböző Atari játékban is sikeres lett [23].

6. Összegzés

A szakdolgozatomban az AlphaGot és a továbbfejlesztett változatait mutattam be. A 2. fejezetben ismertettem, hogyan működik a megerősítéses tanulás folyamata. Külön kitértem a Markov Döntési Folyamatra és bemutattam a Monte Carlo Tree Search algoritmusának működési elvét. Ez utóbbira mutattam egy egyszerű 3×3-as amőbán keresztül egy példát: egy adott állásból végigkövethettem, mi mindent vizsgál meg az algoritmus és ezután meghatároztam a legjobb lépést.

A 3. fejezetben ismertettem, hogyan épülnek fel a mesterséges neurális hálózatok és hogyan lehet őket tanítani. Megvizsgáltam még a konvolúciós hálózatok felépítését, melyeknek fontos szerepük lesz az AlphaGonak és változatainak reprezentálásában.

Az A. függelékben ismertettem a go játék szabályait, melyek ismerete szükséges ahhoz, hogy a 4. fejezetben lássuk azt, mennyi különböző szabály betartására kell figyelniük a programoknak. Ezután a három programot hasonlítottam össze egymással. Megnéztem, mik azok a fontosabb szempontok, melyekben eltérnek egymástól és mik azok, melyekben megegyeznek egymással a különböző változatok. Az AlphaZero már bevitt heurisztikák nélkül ért el sikereket, a Monte Carlo Tree Search algoritmusának használata is fejlődött és kevesebb tanulási idő után értek el nagyobb eredményeket a programok. Az egyik legnagyobb eredménynek tekinthető, hogy az AlphaZero nem csak goban, hanem sakkban és sógiban is túlteljesítette a legerősebb programokat.

Az 5. fejezetben pedig további eredményeket mutattam, amik a DeepMind fejlesztő csapat nevéhez fűződnek.

A szakdolgozatom végén megállapíthatjuk, hogy ezek az eredmények egy lépéssel közelebb visznek minket a mesterséges intelligencia régen kitűzött céljának megvalósításához: egy olyan általános játékrendszer megalkotásához, amely képes elsajátítani bármilyen játékot.

Hivatkozások

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[2] Michael C. Fu. Monte carlo tree search: A tutorial. In *WSC '18: Proceedings of the 2018 Winter Simulation Conference*, pages 222–236, Piscataway, NJ, USA, 2018. IEEE Press. 578181.

[3] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[5] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Chenjun Xiao and Martin Müller. Integrating factorization ranked features in mcts: An experimental study. pages 34–43, 04 2017.

[7] Adrien Couëtoux. Monte carlo tree search for continuous and stochastic sequential decision making problems. pages 74–75, 09 2013.

[8] Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, Mar 2011.

[9] Gerald Tesauro and Gregory R. Galperin. On-line policy improvement using monte-carlo search. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 1068–1074. MIT Press, 1997.

[10] Brian Sheppard. World-championship-caliber scrabble. *Artif. Intell.*, 134(1-2):241–275, January 2002.

[11] Dennis Reidsma, Haruhiro Katayose, and Anton Nijholt. *Advances in Computer Entertainment, 10th International Conference, ACE 2013*, volume 8253. 11 2013.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[13] Stanford University. *Neural Network 1.* http://cs231n.github.io/neural-networks-1/ (Hozzáférés ideje: 2019. december 31.).

[14] Engedy István. *Konvolúciós neurális hálózatok (CNN)*. http://home.mit.bme.hu/~engedy/NN/NN-CNN.pdf, (Hozzáférés ideje: 2019. december 31.).

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[17] DeepMind. *AlphaGo Master series: 60 online games.* https://deepmind.com/alphago-master-series (Hozzáférés ideje: 2019. december 31.).

[18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[19] Demis Hassabis and Fan Hui. *Exploring the mysteries of Go with AlphaGo and China's top players.* https://deepmind.com/blog/article/exploring-mysteries-alphago 2017. április 10. (Hozzáférés ideje: 2019. december 31.).

[20] Dr. Meskó Bertalan. *Az AlphaGo alakíthatja a jövő egészségügyét.* https://mediq.blog.hu/2016/05/19/az_alphago_alakithatja_a_jovo_egeszsegugyet, 2016. május 19. (Hozzáférés ideje: 2019. december 31.).

[21] DeepMind. *AlphaGo Teach: Discover new and creative ways of playing Go*. https://alphagoteach.deepmind.com/ 2017. december (Hozzáférés ideje: 2019. december 31.).

[22] The AlphaStar Team. *AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning*. https://bit.ly/2PRlroJ 2019. október 30. (Hozzáférés ideje: 2019. december 31.).

[23] DeepMind. *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. https://bit.ly/2qYawiI, (Hozzáférés ideje: 2019. december 31.).

[24] Balogh Pál. *A gó*. Terebess Kiadó, 2001.

[25] Balogh Pál. *A gó (Példajátszma 19x19-es táblára)*. https://mek.oszk.hu/00100/00136/html/jatszma.htm (Hozzáférés ideje: 2019. december 31.).

[26] Balogh Pál. *A gó (Gó szabályok)*. https://mek.oszk.hu/00100/00136/html/szabaly.htm (Hozzáférés ideje: 2019. december 31.).

[27] Balogh Pál. *A gó (Gó szótár)*. https://mek.oszk.hu/00100/00136/html/szotar.htm (Hozzáférés ideje: 2019. december 31.).

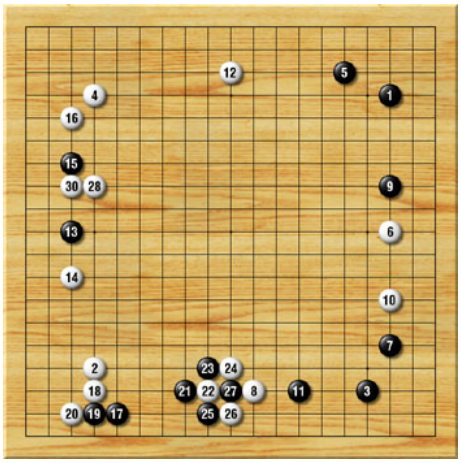
A. A go játék szabályai

A szabályok megírásához és a képekhez a [24] könyvet használtam fel.

A go egy olyan ősi kínai kétszemélyes társasáték, melynek szabályait nagyon könnyű elsajátítani.

A Got a leggyakrabban 19x19-es táblán játszák, bár vannak olyan változatok, ahol 9x9-es vagy 13x13-as táblát használnak. A játékosok nem a mezőkre, hanem azok sarkaira teszik a köveiket. A legtöbb hasonló játékkal ellentétben a fekete kezd, utána felváltva követik egymást a játékosok. Egyik játékos sincs lépésknyszerben, megengedett tehát a *passzolás*. Mindig csak egy üres helyre lehet lépni: ez számít egy kő *lerakásának*.

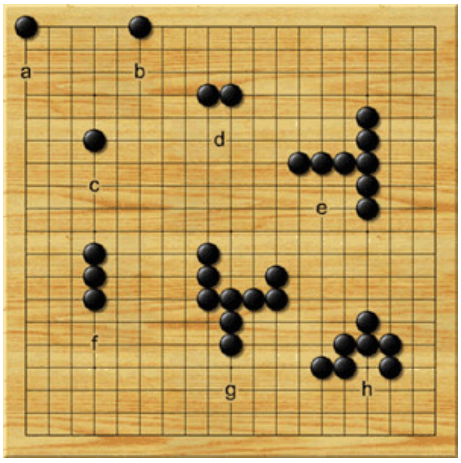
Az időkorlát a modern goban 30 perc és 10 óra között van. Ha a játékosnak elfogy az ideje, nem biztos, hogy veszít, mert ekkor életbe lép a *byo-yomi szabály*, ami lehetővé teszi, hogy fél perc alatt lépjen egy játékos. Ha nem lép, akkor elveszti a játszmát, függetlenül attól, hogy éppen nyerésre áll-e.



8. ábra. Egy go játszma első 30 lépése. Tekinthetjük ezt egyfajta nyitásnak is [25].

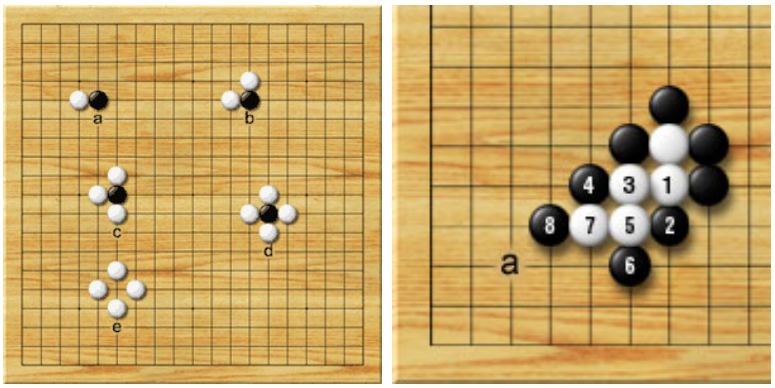
Több olyan követ, amik szomszédosan összeérnek, *csoportoknak* nevezünk. Egy kőnek vagy egy csoportnak pontosan annyi *élete* van, ahány szomszédos mező szabad. Egy kőnek legfeljebb 4 élete van, de a csoportoknak több életük is lehet. Ha egy játékos úgy tesz le egy követ, hogy az ellenfél kövének vagy csoportjának elfogy az élete, akkor *foglyul ejti* azt/azokat.

A 9. ábrán több különböző alakzat látható. Az „a”-nak 2, a „b”-nek 3, a „c”-nek 4, a „d”-nek 6, az „e”-nek 16, az „f”-nek 8, a „g”-nek 16 és a „h” alakzatnak 11 élete van.



9. ábra. Alakzatok [26].

A játék egyik stratégiai alapeleme, hogy mindig legyen legalább egy szabad szomszédos mező, ahová le lehet még tenni követ, így meg lehet akadályozni a bekerítést. Az egyik ilyen nehezen kivédekezhető alakzat a *létra*, ahol a támadó játékos a tábla széle felé próbálja kiszorítani a védekező játékost (10./b ábra).

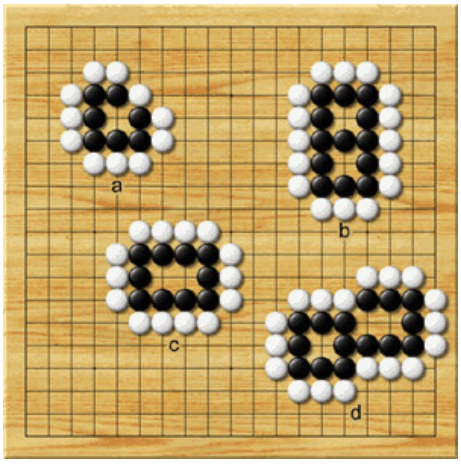


10. ábra. (a) A fekete kő bekerítése [26] és (b) példa egy létrára [27].

A 10./a ábrán az látható, ahogy a fehér kő folyamatosan *keríti be* a fekete követ. Ekkor a fekete kőnek folyamatosan fogy az élete, az „a”-ban 3, a „b”-ben 2, a „c”-ben már csak 1, a „d” állásban pedig már nincs is élete, így le kell venni a tábláról („e” állás). Ezeket a köveket foglyoknak nevezzük.

A bekerített szabad területet nevezzük *szemnek*. Ezek olyan területek, ahová az ellenfél nem tud követ letenni, vagy értelmetlen, mert a következő körben fogoly lesz. *Élő alakzatról* akkor beszélünk, ha legalább két olyan pontja van a területnek, ahová az ellenfél nem tehet. Ezt a két pontot nevezzük az alakzat szemének. Az egy szemű alakzatok belsejét a játékos be tudja fogni, mivel ilyenkor az ellenfél játékos kőcsoportjának elfogy az élete. Ezeket

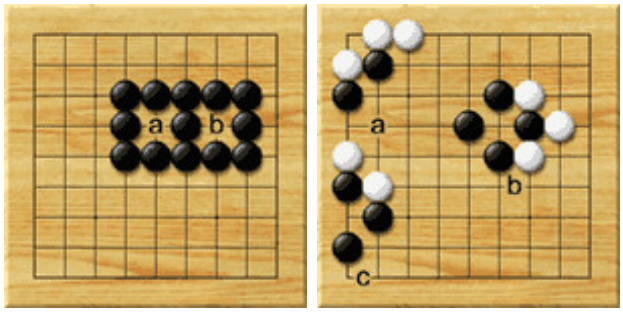
nevezzük *halott alakzatoknak*.



11. ábra. Példák szemekre [26].

A 11. ábrán az „a” és a „c” egy szemű, vagyis halott alakzatok, míg a „b” és „d” élő alakzatok.

A játék szabályzata tiltja az *„öngyilkos” lépéseket*, azaz, hogy olyan helyre tegyen le követ, ahol nem lenne élete. A 12./a ábrán is ilyen „öngyilkos” lépéseket láthatunk. Ellenben megengedett lépésnek számít, ha ezzel az ellenfél egyik kövét vagy kőcsoportját foglyul tudja ejteni.



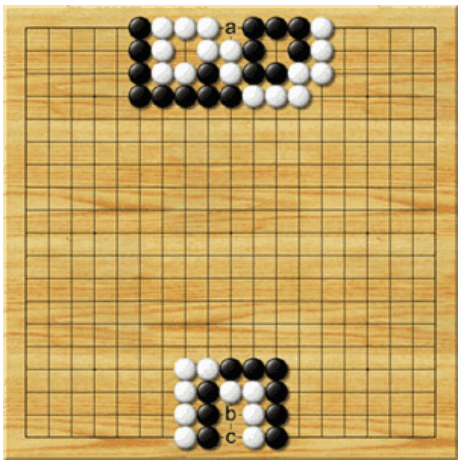
12. ábra. (a) Öngyilkos lépések és (b) „ko” állások [26].

A játék egyik legfontosabb szabálya a *ko szabály*, ami azt mondja ki, hogy nem alakulhat ki két lépésen belül ugyanaz a játékállás. Ugyanakkor előfordulhat az, hogy a játékos egy lépéssel később visszaüt: ezt nevezzük ko-harcnak. A 12./b ábrán három „ko” állást is láthatunk.

Megjegyzés. A 8. ábrán (a tábla alsó részén) is láthatunk egy „ko” állást.

A 13. ábrán az „a”, a „b” és a „c” pontra egyik játékosnak sem éri meg lépni, mert a

másik játékos leütné a kövét. Ezeket a helyzeteket nevezzük *sekinek*. A seki helyzetből adódó pontszámok nem számítanak bele a játék végeredményébe.



13. ábra. Példa seki helyzetekre [26].

Az erősebb játékosok általában előnyköveket szoktak adni a gyengébbeknek. Minimum 2, maximum 9, de előfordulhat, hogy akár 13 előnykövet kap a gyengébbik játékos, melyeknek előre meghatározott helyük van. Az AlphaGo-t megelőzően csak előnykövekkel tudott nyerni egy program profi játékos ellen.

A játék győztese az, aki a több pontot szerezte. Egy-egy pontot érnek a körbekerített metszéspontok, illetve a foglyok. Bár a szabályok nem tiltják meg a döntetlen végkimenetelt, mégis igyekeznek ezt a lehetőséget elkerülni. A döntetlen kiküszöbölésére *komit* alkalmaznak - ez a fekete fizetsége a fehérnek a játék kezdési jogáért. Általában 5.5 pontot ér, így a parti nem tud döntetlennel végződni.

A játéknak akkor van vége, ha mindkét játékos úgy érzi, nem tud több pontot érő lépést végrehajtani (vagyis mindkét játékos passzol). A játék végén a semleges pontokat betömik. A semleges pontok azok a pontok, ahová bármely játékos lép, nem tud pontot szerezni.