Szegedi Tudományegyetem
Természettudományi és Informatikai Kar
Informatikai Intézet

Számítástudomány Alapjai Tanszék

Szakdolgozat

Neurális hálózatokat és Monte Carlo fakeresést kombináló hibrid sakkrobot fejlesztése nyitókönyv integrációval

Készítette

Sere Gergő Márk
programtervező informatikus BSc

Témavezető

Dr. Békési József, főiskolai tanár
Számítástudomány Alapjai Tanszék

Szeged, 2025

Feladatkiírás

Téma megnevezése: Neurális hálózatokat és Monte Carlo fakeresést kombináló hibrid sakkrobot fejlesztése nyitókönyv integrációval
Feladat rövid leírása: Olyan AlphaZero-típusú [27] sakkprogram fejlesztése, amelyben a lépésgenerálás és a PUCT-alapú Monte Carlo fakeresés (MCTS) C++ nyelven valósul meg. A lépésirány- és értékbecslést reziduális konvolúciós neurális háló (CNN) végzi. A rendszer támogassa az önjátszásos tanítást, a kötegelt kiértékelést, a telemetriai méréseket, valamint az arénamérkőzéseken alapuló modellértékelést és monitorozást. A nyitókönyv biztosítsa a kezdőállások minél nagyobb sokféleségét.
Részfeladatok:

1. Bitboard-alapú sakkmag és szabályosságellenőrzés implementálása korszerű C++-ban (C++23).
2. PUCT-alapú MCTS megvalósítása adaptív paraméterezéssel, Dirichlet-zajjal és kötegelt kiértékelő interfésszel.
3. Reziduális CNN tervezése és betanítása önjátszásból származó adatra; aszinkron, automatikus kevert pontosságú (AMP) inferencia az önjátszás és az arénamérkőzések során.
4. Visszajátszási puffer, eredménymegállapítás (adjudikáció), feladás és arénamérkőzések implementálása, valamint telemetria.
5. Teljesítménymérés és összehasonlítás kiinduló megoldásokkal; reprodukálhatóság megvalósítása

Tartalmi összefoglaló

Dolgozatomban egy neurális hálózatokat és Monte Carlo fakeresést kombináló, AlphaZero-típusú hibrid sakkprogram fejlesztését mutatom be korlátozott erőforrású környezetben.

Az AlphaZero-típusú, megerősítéses tanuláson (*reinforcement learning*, RL) alapuló sakkprogramok gyakorlati reprodukálása jellemzően ipari léptékű számítási kapacitást (például TPU-klasztereket) igényel. Ez a forrásigény megnehezíti a módszertan elérhetőségét egyetemi kutatók és hobbifejlesztők számára. Azt vizsgálom, hogy egyetlen fogyasztói GPU-n (RTX 3070 Laptop) és körülbelül 15 órás tanítási időkeretben milyen mérnöki döntésekkel valósítható meg egy kísérleti célokra alkalmas, AlphaZero-típusú sakkprogram.

Az általam bemutatott rendszer C++23-alapú sakkmotort és PUCT-alapú Monte Carlo fakeresést (MCTS) kombinál egy PyTorch-ban implementált, reziduális konvolúciós neurális hálózattal (CNN). A megoldás kötegelt kiértékelést, önjátszást és arénamérkőzéseket alkalmaz. A korlátozott erőforrások és az adatdiverzitás közötti egyensúly megteremtése érdekében a rendszer tömör bemeneti reprezentációt és nyitókönyvet [18] használ.

Munkám főbb hozzájárulásai a következők: korlátos erőforrásokra optimalizált, C++/Python hibrid AlphaZero-típusú sakkmotor megvalósítása; a „Mindig világos” kanonikus nézet (*Canonical Always White*) bevezetése és indoklása a rendszerben; öt különböző konfiguráció szisztematikus kísérleti összehasonlítása; valamint egy teljesen automatizált benchmark- és reprodukciós feldolgozási lánc kialakítása.

Azonos hardverkörnyezetben öt különböző konfiguráció került tesztelésre. A **Nagy Entrópia** konfiguráció rövid futási idő alatt is 1249-es, saját skálán mért Elo-pontszámot (Elo-értéket, a sakkban használt relatív erősségmutatót) ért el, és felülmúlta a mohó heurisztikát, miközben a mélyebb, de kevésbé változatos keresést alkalmazó beállítások gyengébben teljesítettek. Méréseim azt jelzik, hogy rövid futási idő esetén a generált játszmák sokszínűségét biztosító stratégia fontosabb, mint a puszta keresési mélység.

Dolgozatomban öt kutatási kérdésre (RQ1-től RQ5-ig) keresem a választ, amelyek a rendszer mérnöki teljesítményét, skálázhatóságát, tanulási dinamikáját, az adatdiverzitást és a kísérletek reprodukálhatóságát vizsgálják.

Kulcsszavak: AlphaZero, Monte Carlo fakeresés (MCTS), megerősítéses tanulás (RL), hibrid architektúra, GPU-gyorsítás, kísérleti validáció

Tartalomjegyzék

Táblázatok jegyzéke

Ábrák jegyzéke

Rövidítések jegyzéke

- **AMP**: automatikus kevert pontosság (*Automatic Mixed Precision*, PyTorch-környezetben alkalmazott technika)
- **CNN**: konvolúciós neurális hálózat (*Convolutional Neural Network*)
- **CPU**: központi feldolgozóegység (*Central Processing Unit*)
- **EMA**: exponenciális mozgóátlag (*Exponential Moving Average*)
- **GIL**: globális interpreter zár (*Global Interpreter Lock*)
- **GPU**: grafikus feldolgozóegység (*Graphics Processing Unit*)
- **JSON**: strukturált adatcsere-formátum (*JavaScript Object Notation*)
- **LRU**: legrégebben használt (*Least Recently Used*)
- **MCTS**: Monte Carlo fakeresés (*Monte Carlo Tree Search*)
- **NPS**: csomópont másodpercenként (*Nodes Per Second*)
- **PUCT**: UCT-variáns döntési prior súlyozással (*Policy UCT / Prioritized UCT*)
- **RL**: megerősítéses tanulás (*Reinforcement Learning*)
- **RQ**: kutatási kérdés (*Research Question*)
- **SGD**: sztochasztikus gradiensmódszer (*Stochastic Gradient Descent*)
- **TDP**: tervezett hőteljesítmény (*Thermal Design Power*)
- **TPU**: tenzorprocesszor-egység (*Tensor Processing Unit*)
- **UCT**: fákra alkalmazott felső konfidenciahatár (*Upper Confidence bounds applied to Trees*)

1. fejezet

Bevezetés

Az AlphaZero [27] 2017-ben látványosan bemutatta, hogy a megerősítéses tanulás (RL) sikerrel alkalmazható komplex, teljes információjú játékokban is. Ugyanakkor a módszer gyakorlati reprodukálása jellemzően ipari léptékű erőforrásokat (például TPU-klasztereket) feltételez. Ez a magas belépési küszöb megnehezíti a technológia oktatását és kutatását, különösen a hazai felsőoktatási intézményekben. Dolgozatomban a „sakkrobot” kifejezés olyan szoftveres sakkprogramot jelöl, amely egy keresőmotort és egy neurális hálózatot kombinál.

Fő motivációm a technológia elérhetővé tétele hallgatók és kutatócsoportok számára. A cél egy olyan, korlátos erőforrásokra optimalizált mérnöki megközelítés kidolgozása, amely lehetővé teszi, hogy egyetlen fogyasztói GPU-n is érdemi kísérleteket lehessen végezni AlphaZero-típusú sakkágensekkel.

Munkámban azt vizsgálom, hogy egy NVIDIA RTX 3070 Laptop GPU-val, korlátozott, körülbelül 15 órás tanítási időkeretben milyen architekturális és hiperparaméterbeli kompromisszumokkal hozható létre működőképes, önjátszással tanuló sakkprogram. Az értékelés során öt eltérő konfiguráció kerül összehasonlításra a keresési mélység, az entrópia-alapú exploráció és a frissítési gyakoriság szempontjából.

Dolgozatom főbb eredményeit és hozzájárulásait az alábbiakban foglalom össze:

- **Architekturális egyszerűsítés:** A „Mindig világos” kanonikus nézet (*Canonical Always White*) bevezetésével a bemeneti állapottér komplexitása érdemben csökken, ami mérsékli az erőforrás-igényes adataugmentáció szükségességét.
- **Hibrid architektúra:** A rendszer a számításigényes feladatokat (lépésgenerálás, MCTS) nagy teljesítményű C++23 kóddal, míg a tanítást és a kötegelt kiértékelést rugalmas, PyTorch-alapú környezetben valósítja meg. A két réteg között egy vékony, minimális adatmásolást igénylő illesztőfelület biztosítja a hatékony adatcserét.
- **Eredmények röviden:** A **Nagy Entrópia** konfiguráció érte el a legjobb, saját skálán mért Elo-pontszámot, és a mérések alapján kis, de kimutatható előnyt mutatott

a többi beállítással szemben, ami a diverzitás fontosságára utal rövid tanítási horizonton.

- **Reprodukálhatóság:** Teljesen automatizált benchmark- és artefaktumgeneráló szkriptek, valamint verziózott YAML-konfigurációk rögzítik a hardver- és hiperparaméter-beállításokat, így a kísérleti futtatások azonos beállítások mellett más környezetben is megismételhetők.

A szakdolgozat készítése során két generatív, mesterséges intelligencián alapuló nyelvi modellcsaládot használtam kizárólag kiegészítő segédeszközként. Az OpenAI ChatGPT GPT-4o és GPT-5 modelljeit, valamint a Google Gemini 2.5-öt alkalmaztam a tanulási folyamat támogatására (szakirodalom gyors áttekintése, nyelvi-stiláris ellenőrzés), nem pedig a dolgozat érdemi tartalmának előállítására. Konkrétan vázlatos összefoglalók készítése, alapfogalmak tisztázása, valamint nyelvhelyességi javaslatok kérése során vettem igénybe ezeket az eszközöket. Nem használtam őket bekezdések vagy fejezetek automatikus létrehozására, és nem vettem igénybe az MI-t az algoritmusok, a forráskód vagy a szakmai következtetések megalkotásához; ezek a saját, önálló munkám eredményei.

Az MI által javasolt tartalmakat minden esetben kritikusan, segédeszközként kezeltem. A kapott információkat független, hivatkozott források alapján ellenőriztem, a nyelvi javaslatokat pedig dolgozatom érvrendszeréhez és saját megfogalmazásomhoz igazítva építettem be. A dolgozat tartalmáért és esetleges hibáiért teljes egészében én vállalom a felelősséget. Az MI használata az „A mesterséges intelligencia elfogadható és felelős használata a Szegedi Tudományegyetemen Hallgatói segédlet a mesterséges intelligencia tanulásban történő tudatos, hatékony és etikus alkalmazásához” című útmutatóban rögzített alapelvekkel összhangban, a hallgatói önálló munka elsődlegességét tiszteletben tartva történt.

A dolgozat felépítése a következő: az **Irodalmi áttekintés** fejezet a kapcsolódó szakirodalmat foglalja össze. Ezt követi a **Célkitűzések és módszertan** fejezet, amely meghatározza az RQ1-től RQ5-ig terjedő kutatási kérdéseket és a kísérleti elrendezést. A **Rendszer tervezése és implementáció** fejezet részletesen bemutatja a hibrid architektúrát és a teljesítménykritikus komponensek tervezési döntéseit. Az 5. fejezet az öt vizsgált forgatókönyv eredményeit ismerteti, végül pedig a **Megbeszélés és korlátok** fejezet a skálázhatósági tapasztalatokat és a kutatási tanulságokat összegzi.

2. fejezet

Irodalmi áttekintés

2.1 Klasszikus sakkprogramozás

A modern sakkmotorok gondolkodásmódját már Shannon korai elemzései keretezték [25]. Ő vezette be azt a ma is használt megkülönböztetést, amely szerint az *A-típusú* keresés kimerítő, állandó mélységig vizsgálja a legtöbb lehetséges folytatást, míg a *B-típusú* megközelítés csak néhány, előzetesen ígéretesnek ítélt jelöltet visz mélyre. A történeti fejlődés végül az A-típusú, kvázi „nyers erő” (*brute-force*) irányt részesítette előnyben. A megnövekedett számítási kapacitás és a kiforrott vágási heurisztikák jobban skáláztak, mint a korai, erősen kézi szabályokra támaszkodó szelektív keresések.

A klasszikus sakkprogramok a Minimax-elvre épülő, korlátozott mélységű keresést alkalmaznak. A sakkjátékfa Shannon által becsült komplexitása nagyságrendileg 10^{120} lehetséges játék [25], miközben a pozíciók száma is óriási ($\approx 10^{43}$ és 10^{47} közötti nagyságrend), ezért a teljes játékfa bejárása gyakorlatilag lehetetlen. A fa levelein (például a nyugalmi keresés végén) heurisztikus kiértékelő függvény (*evaluation function*) ad pontszámot az állásnak. Ezek a függvények tipikusan több, egymástól elkülöníthető komponensre bontják a sakkpozíciót (anyag, gyalogstruktúra, tisztek aktivitása, királybiztonság stb.), majd *lineáris, kézzel hangolt súlyokkal* kombinálják ezeket. A nagymesterek és programozók évtizedeken át finomhangolták ezeket a súlyokat; ezzel szemben a neurális hálókon alapuló kiértékelésnél (amelyet dolgozatomban is alkalmazok) a nemlineáris, adatból tanult reprezentációk váltják fel a kézi súlytáblákat.

A keresési tér hatékony bejárását az alfa-béta vágás (Alpha-Beta pruning) [15] teszi lehetővé, amely matematikailag garantáltan ugyanazt az eredményt adja, mint a teljes Minimax. Az irreleváns ágak levágásával (ahol a lépés már biztosan rosszabb, mint egy korábban talált alternatíva) a módszer ideális rendezés esetén nagyjából a négyzetgyökére csökkenti a keresési fa effektív elágazási tényezőjét. A modern motorok ezt tovább javítják olyan heurisztikákkal, mint a null-move vágás [7] (ahol a lépés jogának átadása mellett keresés történik, feltételezve, hogy ha így is előny áll fenn, az eredeti állás még jobb) és a késői lépésredukció (Late Move Reductions, LMR) [6].

A Deep Blue 1997-es győzelme Kaszparov felett [5] rámutatott a számítógépes sakk nagy potenciáljára, de ez a győzelem specifikus hardverre és hatalmas kézi tudásbázisra épült. A mai legjobbak közé tartozó klasszikus motor, a Stockfish [30] is ezt a vonalat képviseli. Keresési algoritmusa továbbra is a klasszikus alfa-béta vágáson alapul, ugyanakkor a korábbi, kézzel hangolt kiértékelő függvényt ma már NNUE (Efficiently Updatable Neural Network) [21] technológiával megvalósított neurális értékelés váltotta fel. Ezen rendszerek közös korlátja a manuálisan hangolt tudás szükségessége és a domén-specifikusság.

2.2 Monte Carlo Tree Search és AlphaZero

2.2.1 MCTS alapok és fejlesztések

A Monte Carlo fakeresés módszereit Browne és mtsai [4] részletesen áttekintik; az alábbiakban dolgozatom szempontjából releváns elemek kerülnek röviden összefoglalásra. A Monte Carlo fakeresés (*Monte Carlo Tree Search*, MCTS) [8, 16] egy aszimmetrikus, legjobb-első (*best-first*) jellegű keresőalgoritmus véletlen szimulációkkal, amely a Go területén ért el először jelentős eredményt [10]. Az algoritmus elvileg nem igényel doménspecifikus heurisztikát (elegendőek a játékszabályok és a végállapotok kiértékelése), a gyakorlatban azonban a sikeres rendszerek a hatékonyság növelése érdekében számos heurisztikával egészítik ki az alapalgoritmust.

Az MCTS működése négy, egymásra épülő lépés ciklikus ismétlésén alapul:

1. **Szelekció:** A gyökértől indulva a fa mentén olyan útvonalat választunk, amely egy kiválasztási stratégia (pl. UCT vagy PUCT) szerint ígéretesnek tűnik. A bejárás addig folytatódik, amíg olyan csomóponthoz nem jutunk, ahol még van legalább egy, eddig nem vizsgált folytatás. A döntés célja minden lépésben az exploráció (új lehetőségek kipróbálása) és az exploitáció (jól bevált folytatások további vizsgálata) közötti kompromisszum.
2. **Expanzió:** A kiválasztott csomóponthoz *csak ekkor* adunk hozzá egy vagy több új gyermeket, amelyek a következő lépéseket reprezentálják. Így a fa kizárólag a ténylegesen vizsgált folytatások mentén nő tovább, nem pedig teljes szélességben.
3. **Szimuláció:** Az eredeti, „klasszikus” MCTS-variánsokban innen indul egy gyors, sokszor véletlenszerű (vagy egyszerű heurisztikákkal vezetett) játszma a végállapotig (rollout), és a végkimenetelt használják az állás értékének becslésére. A modern, AlphaZero-típusú rendszerek, így a jelen dolgozatban ismertetett megoldás is, a költséges rolloutokat jellemzően neurális értékbecsléssel váltják ki: a levélcsomópontnál a hálózat becsli meg a pozíció várható kimenetelét.
4. **Visszaterjesztés:** A kapott értéket visszavezetjük a fán felfelé a gyökérig, minden érintett csomópontban frissítve a látogatási számot (N) és az átlagos értéket (Q).

Így a következő iterációk egyre megbízhatóbb statisztikákra támaszkodhatnak, és a keresés fokozatosan a kedvezőbb ágak felé koncentrálódik.

Az UCT algoritmus [16] a többkaros bandit problémákra kidolgozott UCB1 stratégiát [2] alkalmazza a fában. Ennek képlete:

$$UCT = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N_i}{n_i}}$$

ahol w_i a nyerések száma, n_i a csomópont látogatottsága, N_i a szülő látogatottsága, C pedig az explorációs konstans. A RAVE-heurisztika [11] további gyors konvergenciát tesz lehetővé a szimulációk eredményének megosztásával a testvércsomópontok között.

Az AlphaGo [26] 2016-ban új alapokra helyezte ezt a megközelítést azzal, hogy a költséges és nagy varianciájú véletlen játszmabefejezéseket (rolloutokat) neurális hálózatokkal egészítette ki, és nagyrészt kiváltotta. Az AlphaGo Zero [28] tovább egyszerűsítette a rendszert. A korábbi különálló döntéshozó (policy) és értékbecslő (value) hálózatokat egyetlen, kétfejű reziduális hálózatba integrálta. Továbbá bevezette a PUCT kiválasztási formulát, amely a hálózat által jósolt a priori valószínűségeket ($P(s,a)$) használja az exploráció irányítására.

A PUCT (UCT variáns döntési prior (*policy prior*) súlyozással) alapformulájában [27, 28] a szakirodalomban gyakran egy konstans kiegyensúlyozási paraméterrel illusztrálják az exploráció szabályozását, ugyanakkor az AlphaGo Zero- és AlphaZero-tanulmányok kiegészítő anyagaiban valójában egy, a szülőcsomópont látogatottságától függő dinamikus c_{puct}-skálázást írnak le. A közösségi fejlesztésű, AlphaZero-típusú implementációkban (pl. KataGo [31], LC0 [17], CrazyAra [9]) ezt a dinamikus formulát veszik át és hangolják tovább; dolgozatomban is a (2.1) szerinti alakot alkalmazom:

$$c_{\text{puct}} = \log\left(\frac{N(s) + c_{\text{base}} + 1}{c_{\text{base}}}\right) + c_{\text{init}} \tag{2.1}$$

ahol tipikusan $c_{\text{base}} = 19\,652$, és c_{init} tipikusan 1 és 3 közötti érték (míg jelen rendszerben konkrétan $c_{\text{init}} = 1{,}55$). Ez biztosítja, hogy ritkábban látogatott állapotokban nagyobb exploráció történjen, tipikusan $c_{\text{puct}} \in [1, 5]$ tartományban (lásd (2.1)).

A PUCT akció-kiválasztási formula a következő pontszámot maximalizálja:

$$\text{Score}(s,a) = Q(s,a) + U(s,a),$$

ahol az explorációs tag definíciója:

$$U(s,a) = c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}.$$

Itt $N(s)$ a csomópont (állapot) látogatási száma, $N(s,a)$ az adott él látogatottsága, $Q(s,a)$ az átlagos érték, $P(s,a)$ pedig a hálózat által adott prior valószínűség. A rendszerben a konfigurációs $c_{\text{puct,scale}}$ érték (pl. 1,35 vagy 2,5) a (2.1) szerinti dinamikus faktort skálázza, ahogyan azt az 5.1. táblázat mutatja. Dirichlet-zaj biztosítja az explorációt a gyökérben. Az AlphaZero [27] ezt általánosította sakkra és shogira, emberfeletti szintet érve el néhány óra tanítással (sakkra 9 óra).

2.2.2 AlphaZero tanítási eljárás részletei

Az AlphaZero tanítási folyamata négy, egymást követő fázis iterálásán alapul [27]. Az **önjátszás** (self-play) szakaszban az aktuális modell játszik önmaga ellen változatos kezdőpozíciókból. Az eredeti AlphaZero-rendszerben minden lépéshez körülbelül 800 MCTS-szimuláció tartozik, a neurális háló érték- és policy-kimenetei pedig a levelekben irányítják a keresést [27]. A lépés kiválasztása a csomópontok látogatási számaiból képzett eloszlásból történik, amelyet a hőmérsékletparaméter torzít. Korai játékban magasabb τ érték mellett, később $\tau \to 0$ közelében, hogy a rendszer fokozatosan a legvalószínűbb lépések felé koncentráljon. A játék végén a kimenetel ($z \in \{-1, 0, +1\}$) minden, a játszma során előfordult pozícióhoz hozzárendelt értékcéllá válik; a saját implementációmban ugyanezt az elvet követem, de a szimulációk számát a rendelkezésre álló erőforrásokhoz igazítom (lásd 5.1. táblázat).

Az **adattárolás** fázisban a pozíció-lépésirány-kimenetel hármasokat visszajátszási pufferben gyűjtik. Az AlphaGo Zero [28] kiegészítő anyaga explicit módon a legutóbbi $\sim 500\,000$ játékot tartalmazó visszajátszási puffert ír elő. Az AlphaZero-tanulmányok [27, 29] ugyanennek a tanítási protokollnak az adaptációi, de a konkrét pufferméretet nem minden esetben részletezik. A közösségi reimplementációk ezért jellemzően 500k és 1M közötti puffert alkalmaznak. A minták (s_t, π_t, z_t) formában tárolódnak, ahol s_t a bemeneti reprezentáció (8 történeti lépés), π_t az MCTS látogatási eloszlás (policy target), és z_t a végső kimenetel a t-edik pozíció nézőpontjából.

A **tanítási fázis** mini-kötegeken optimalizálja a hálót. A veszteségfüggvény [27] három tagból áll:

$$\mathcal{L} = \underbrace{-\pi^T \log p}_{\text{policy-veszteség}} + \underbrace{\lambda_v (z - v)^2}_{\text{értékveszteség}} + \underbrace{\lambda_c \|\theta\|^2}_{\text{L2 regularizációs tag}} \tag{2.2}$$

ahol p és v a háló kimenetei, λ_v tipikusan 1, λ_c a súlycsökkenés koefficiense (10^{-4}). A gyakorlatban $\lambda_v = 1$, ezért dolgozatomban a $\mathcal{L}_{\text{val}}$ definíciójában elhagyom. Az AlphaZero az eredeti cikkben [27] momentumos SGD-t és lépcsőzetes (*stepwise*) ütemezést alkalmaz a tanulási rátára.

A modern AlphaZero-típusú reimplementációk gyakran bemelegítési szakasszal (*warmup*) és koszinuszos ütemezésű csökkenéssel kombinált stratégiát (SGDR-szerű [19] ütemezés újraindítás nélkül) használnak. Dolgozatomban is ezt a megközelítést követem.

Az **aréna-értékelés** (*arena evaluation*) fogalma a szakirodalomban két, egymástól elkülönülő szerepben jelenik meg. Az AlphaGo Zero [28] tanítási protokolljában az úgynevezett aréna-kapuzás (*arena gating*) döntötte el, hogy egy jelölt háló átveheti-e az önjátszásban használt „best player" szerepét. Minden új háló 400 játszmát játszott az aktuális legjobb ellen, és csak akkor lépett elő, ha a győzelmi aránya meghaladta az 55%-ot. Ez a konzervatív küszöb csökkentette a zajból eredő fluktuációt, és segített elkerülni a tanulás instabil „visszacsúszását", ugyanakkor a tanulást nem tette monoton növekvővé, sok iteráció is eltelhetett új „best player" nélkül. Az AlphaZero-tanulmányok [27, 29] ezzel szemben kifejezetten elhagyják ezt a kapuzási lépést. Az önjátszásos adatok generálására mindig az éppen legutóbb tanított hálót használják, és az aréna jellegű tornákat csak utólagos erősségmérésre (pl. Elo-becslés Stockfish vagy AlphaGo-változatok ellen) alkalmazzák, nem pedig a további önjátszásban használt modell kiválasztására. Az MCTS-ben használt átlagolt Q-értékek részben „simítják" a neurális háló rövid távú hibáit, így a rendszer képes rosszabb köztes súlykonfigurációkból is „kimozogni" anélkül, hogy formális kapuzási döntésre lenne szükség.

2.2.3 Ismert tipikus hibamódok és kihívások

Az AlphaZero-típusú rendszerek számos tipikus hibamóddal rendelkeznek, amelyeket a szakirodalom [17, 27, 31] és a közösségi reimplementációk tapasztalatai részletesen dokumentálnak.

Értékbecslő ág kollapszusa: A neurális háló értékbecslő ága konstans értéket tanulhat meg (pl. minden pozícióra $v \approx 0$), különösen túl kis adatmennyiség vagy túl nagy hálókapacitás esetén. Ez azt eredményezi, hogy az MCTS nem kap hasznos értékbecsléseket, és véletlenszerű lépéseket választ. A lehetséges megoldások: kisebb háló, több adat, vagy az értékbecslő ág tanulási rátájának csökkentése.

Lépésirány-túlillesztés: A policy ág memorizálhatja a tanítóadatok specifikus pozícióit anélkül, hogy általános sakkstratégiát tanulna. Ez magas tanítási pontosságot és alacsony validációs teljesítményt eredményez. Ez különösen kis nyitókönyv esetén gyakori, amikor a modell ugyanazokat a pozíciókat látja újra és újra. A megoldás nagyobb nyitókönyv, adataugmentáció (tükrözés, forgatás), dropout.

Exploráció-kizsákmányolás egyensúly: Túl konzervatív exploráció (alacsony Dirichlet zaj, magas c_{puct}) korai konvergenciát okoz rossz lokális optimumba. Túl agresszív exploráció véletlenszerű játékot eredményez, lassítva a tanulást. Az optimális paraméterek játékonként változnak. Sakkban $\alpha = 0{,}3$, shogiban $\alpha = 0{,}15$, Go-ban $\alpha = 0{,}03$, amelyeket az AlphaZero cikk [27] és az AlphaGo Zero cikk [28] részletesen dokumentál (a kiegészítő anyagokban), és amelyeket a közösségi AlphaZero-típusú implementációk gyakorlatban is megerősítettek.

Aréna-értékelés zajérzékenysége: Kis meccsszám (< 100 játék) nagy varianciát eredményez a becslésben. Két azonos erejű modell mérkőzése esetén a kimenetel közel azonos, nagyjából 50%-os arány mindkét fél számára, így 100 játék esetén a 95%-os konfidenciaintervallum $\pm 10\%$. Az AlphaGo Zero által alkalmazott konzervatív küszöb (55%+ győzelmi arány) és nagy meccsszám (400+ arénaparti) csökkenti a hamis pozitív előléptetéseket; az AlphaZero esetében ehelyett folyamatos, kapuzás nélküli önjátszás zajlik, és az arénajellegű tornák csak külső erősségmérésre szolgálnak.

Számítási költség skálázódása: Az AlphaZero eredeti sakkfuttatása mintegy 9 órás volt, 5000 első generációs TPU-t használva az önjátszásos adatok generálására és 64 második generációs TPU-t a háló tanítására [29]. Közösségi becslések szerint ez több tízmilliónyi önjátszásos játszmát eredményezhetett, ami egyetlen fogyasztói GPU-n nagyságrendileg éveket venne igénybe. Kompromisszumok szükségesek: kevesebb iteráció, kisebb háló, kevesebb MCTS-szimuláció, kisebb puffer.

2.2.4 Leela Chess Zero tanulságok

A Leela Chess Zero [17] nyílt forráskódú közösségi projekt, amely az AlphaZero-módszertant valósítja meg elosztott önkéntes GPU-kon. Több ezer önkéntes GPU-idejét aggregálva a projekt 2018 óta több milliárd önjátszásos játszmát generált.

Hálóarchitektúra evolúció: Az LC0 kezdetben kisebb reziduális hálókat használt (pl. 6 reziduális blokk 64 és 128 csatornával), majd fokozatosan 15, 20 és később ∼40 blokkos hálókra nőtt több tízmillió paraméterrel. A 2020-as évek közepére a legmodernebb LC0/LZ hálók ∼40 vagy több blokkot és nagyobb csatornaszámokat használnak, gyakran tízmilliós paraméterszámmal, Squeeze-and-Excitation (SE, csatornaválasztó modul) [13] rétegekkel kiegészítve. Ez arra utal, hogy a sakkhoz szükséges reprezentációs kapacitás jelentős, és a kisebb hálók (5 és 10 blokk között) csak korlátozott játékerőt érnek el.

Tanítási trükkök: A KataGo [31] és az LC0 projekt számos fejlesztést vezetett be az AlphaZero eredeti receptjéhez képest. A KataGo összetettebb veszteségfüggvényt és adatpipeline-t alkalmaz (például többféle értékcélt, playout cap randomizációt és a policy-érték jel közötti arány finomhangolását), amelyek segítenek mérsékelni az értékfej túlillesztését és esetleges kollapszusát. Az LC0 projekt emellett *FPU reductiont* (First-Play Urgency csökkentés) és *smart pruningot* (intelligens levágás) vezetett be a gyakorlatban. Az FPU reduction a nem látogatott csomópontoknak konzervatívabb kezdeti értéket ad, elkerülve a túl agresszív, zajos explorációt, míg a smart pruning alacsony priorú lépéseket vág le korán, a keresés hatékonyságát javítva egy elfogadott pontosság-sebesség kompromisszum mellett.

Adatmennyiség vs. hálókapacitás: Az LC0 tapasztalata, hogy a hálóméret növelése csak akkor hasznos, ha az adatmennyiség arányosan nő. Korai fázisban (< 10M játék) a kis

hálók (6 és 10 blokk között) gyorsabban tanulnak. Nagyobb adatbázissal (100M+ játék) a nagy hálók (20 és 40 blokk között) előnybe kerülnek, de konvergenciájuk lassabb. Ez alátámasztja a tanterv alapú tanulás (*curriculum learning*) [3] megközelítést: kezdjük kis hálóval, majd növeljük a kapacitást konvergencia után.

Elosztott infrastruktúra: Az LC0 szerver-kliens architektúrát használ. Központi szerver osztja ki a tanítási súlyokat és gyűjti az önjátszás adatokat, kliensek (önkéntes GPU-k) generálják a játékokat és küldik vissza. Ez lehetővé teszi a skálázódást, de késleltetést és koordinációs költséget vezet be. Az általam vizsgált egyetlen GPU-s megközelítés elkerüli ezt a bonyolultságot, de korlátozott adatgenerálási sebességgel rendelkezik.

2.3 Közösségi implementációk és hardveroptimalizáció

A Leela Chess Zero (LC0, [17]) kezdetben az AlphaZero-nál kompaktabb, kisebb dimenziójú policy kimenetet alkalmazó lépéskódolási sémákat is vizsgált, szemben az AlphaZero 4672-es ($73 \times 8 \times 8$) kódolásával. Az újabb LC0 hálók azonban már jellemzően AlphaZero-stílusú, $73 \times 8 \times 8$ policy fejet alkalmaznak. A KataGo [31] és MuZero [24] kifinomult tanítási technikákat vezetett be. Kulcsfontosságú mérnöki technikák: kötegelt feldolgozás, `channels_last` (csatorna-utolsó) memória, vegyes precizió [20], aszinkron feldolgozási láncok, gyorsítótár. Hibrid C++/Python architektúrák [14] egyensúlyban tartják a sebességet és fejlesztési hatékonyságot.

2.4 Kutatási rés

Az AlphaZero-irányzat ipari léptékű hardverigénye [29] jelentős korlátot jelent a módszertan széles körű elterjedése szempontjából. A szakirodalomban számos, különböző minőségű és célú reimplementáció található (például tisztán Python-alapú, oktatási jellegű keretrendszerek vagy nagy, elosztott infrastruktúrára tervezett rendszerek), ugyanakkor nem találtam olyan, széles körben hivatkozott referenciaimplementációt, amely *egyidejűleg* teljesítené az alábbi követelményeket:

- magas szintű, gyakorlatban is értelmezhető sakktudást ér el;
- C++-alapú, teljesítményre optimalizált keresőmotort használ;
- egyetlen fogyasztói GPU-ra van optimalizálva, rövid (egynapos) futási idővel;
- oktatási és kutatási célokra jól dokumentált, könnyen újrafuttatható kísérleti pipeline-t biztosít.

Ez a hiány nehezíti az AlphaZero-módszertan oktatási alkalmazását és a reprodukálható kutatást. Korlátozott erőforrással rendelkező egyetemi kutatócsoportok és egyéni fejlesztők gyakorlatilag nem tudnak olyan, jól definiált, fogyasztói hardverre szabott referenciarendszerre támaszkodni, amely mind a játékerő, mind a mérnöki szempontok tekintetében iránymutató volna.

Dolgozatom ezért elsődlegesen a módszertan megvalósíthatóságára és mérhető, korlátos erőforrások mellett elérhető teljesítményére fókuszál; a pontos célkitűzéseket a következő fejezet részletezi.

3. fejezet

Célkitűzések és módszertan

Dolgozatom központi kérdése az, hogy az AlphaZero-típusú sakkprogram megvalósítható-e fogyasztói hardveren, korlátozott időkeret mellett, és ehhez milyen mérnöki döntések szükségesek. Emellett azt is vizsgálom, hogy ezekkel a korlátokkal milyen teljesítmény érhető el.

Ebben a fejezetben foglalom össze a kutatási célkitűzéseket, a hozzájuk tartozó kutatási kérdéseket és a kísérleti elrendezés főbb elemeit; a részletes rendszerleírást a következő fejezet tartalmazza.

A munka öt fő célkitűzése:

1. **Hibrid C++/Python architektúra megvalósítása** tiszta, jól dokumentált API-val.
2. **Komponens-szintű gyorsulások mérése** (sakkmag, MCTS, GPU-következtetés).
3. **Teljes tanítási ciklus demonstrálása** egynapos költségvetéssel.
4. **Arénamérkőzések metrikáinak rögzítése** a tanulási stabilitás követéséhez.
5. **A kísérletek reprodukálhatóságának biztosítása** benchmarkokkal és konfigurációkkal.

3.1 Kutatási kérdések (RQ1-től RQ5-ig)

A célkitűzésekből öt kutatási kérdés (RQ1-től RQ5-ig) adódik:

RQ1, Teljesítmény: Milyen komponens-szintű gyorsulások érhetők el a hibrid architektúrával?

RQ2, Skálázhatóság: Hogyan skálázódik a rendszer a párhuzamosítással, és melyek a szűk keresztmetszetek?

RQ3, Tanulási dinamika: Mutat-e a rendszer belső tanulási jeleket (a policy- és value-veszteség tartós csökkenése, valamint az arénamérkőzések győzelmi arányainak javulása) rövid (egynapos) tanítási horizonton?

RQ4, Adatdiverzitás: Milyen mértékben biztosítja a nyitókönyvvel [18] indított önjátszás és az explorációs beállítások együttese a pozíciók kellő sokszínűségét?

RQ5, Reprodukálhatóság: Mennyire támogatja a rendszer felépítése és az automatizált eszközlánc a futások reprodukálhatóságát (azonos beállítások mellett)?

Dolgozatom célja **nem** egy versenyképes sakkprogram létrehozása, hanem az AlphaZero-módszertan **megvalósíthatóságának** demonstrálása korlátozott erőforrásokkal, valamint a rendszer **mérnöki teljesítményének** jellemzése.

A megvalósítás során a kritikus útvonalhoz tartozó komponenseket (bitboard-alapú sakkmag, PUCT-MCTS) C++23-ban, a tanulási folyamatokat pedig Pythonban, PyTorch [22]-ra épülő gépi tanulási feldolgozási lánccal valósítottam meg. A tervezés köteg-orientált, kötegelt GPU-feldolgozással. A rendszer többszintű LRU-gyorsítótárat (érték-, kimenet- és kódolási gyorsítótár) és ritka lépésirány-tárolást alkalmaz (körülbelül 75-szörös memóriamegtakarítással).

A sikerkritériumok három kategóriába sorolhatók:

- **Teljesítmény:** a pozíció/mp-ben és áteresztőképességben mért mutatók jelentős javítása a Python-alapú kiinduló megoldáshoz képest (python-chess sakkmag, tisztán Python MCTS, nem optimalizált neurális következtetés).
- **Tanulás:** a belső metrikák jelentős csökkenése (a policy-veszteség mintegy 40%-os, az érték-veszteség pedig nagyjából 50%-os relatív javulása).
- **Mérnöki:** automatikus metrika-naplózás (CSV/JSONL), YAML-konfigurációk és unit tesztek megléte.

A fenti célkitűzéseket a későbbi fejezetekben bemutatott kísérleti eredmények alapján értékelem, különösen az RQ1 (teljesítmény) és RQ3 (tanulási dinamika) kapcsán. A játékerő hosszú távú növekedésének és Elo-alapú külső validációnak a vizsgálata dolgozatom hatókörén kívül esik; a fókusz a technikai megvalósíthatóságon és a komponensteljesítményen van.

A kísérleti elrendezés három pillérre épül:

1. Számos izolált teljesítménymérés készült, amelyek lefedték a sakkmag, az MCTS-keresés, a neurális következtetés, az augmentáció és a visszajátszási puffer sebességét.
2. Ezt követően teljes rendszerfuttatások történtek egységes, 6 blokkos, 96 csatornás hálóval, eltérő szimulációs mélység és kötegméret beállításokkal.
3. Végül ablációs vizsgálatok készültek nyitókönyvvel és anélkül, erősen lecsökkentett gyorsítótár-kapacitással (gyakorlatilag gyorsítótár nélküli konfigurációkban), exponenciális mozgóátlag (*Exponential Moving Average*, EMA) és közvetlen súlyok összehasonlításával, valamint a játszma-kiértékelés progresszív és fix változataival.

A következő fejezet a fenti célok és módszertan megvalósítását ismerteti. Bemutatja a hibrid architektúra részleteit, a teljesítménykritikus komponensek tervezési döntéseit,

valamint a széles körben alkalmazott optimalizációs technikákat. A 3.1. szakaszban megfogalmazott, RQ1-től RQ5-ig terjedő kérdéseket az 5. fejezet értékeli ki, a 7. fejezet pedig ezeket egyenként összegzi és kontextusba helyezi.

4. fejezet

Rendszer tervezése és implementáció

4.1 Architektúra

A rendszer hibrid felépítésű. A számításigényes komponensek (bitboard sakkmag, virtuális veszteségre (*virtual loss*) épülő konkurencia-kezelésű MCTS-motor) C++23-ban íródtak, míg a neurális háló tanítása és következtetése Pythonra épülő PyTorch környezetben történik. A C++ és a Python közötti interfészt a `pybind11` könyvtár biztosítja, amely lapos tömbök használatával gyakorlatilag másolásmentes adatcserét tesz lehetővé. A numerikus adatok hatékony kezelését a `NumPy` [33] csomag biztosítja, míg a kísérleti eredmények vizualizációját a `matplotlib` [34] és `seaborn` [35] könyvtárak támogatják.

A rendszer alapvető tervezési elve a **kanonikus reprezentáció**, azaz a „Mindig világos” nézet. A hálózat kizárólag a világos szemszögéből látja a táblát. Sötét lépése esetén a táblát és a lépéseket a bemeneti kódolás során a rendszer tükrözi (vertikális tükrözés + színcsere), majd a hálózat kimenetét (policy) visszatükrözi. Hasonló „always white” nézetet alkalmaznak a modern neurális sakkmotorok is (pl. Stockfish NNUE [21, 30], LC0 [17]); dolgozatomban ezt a már bevált elvet igazítom a korlátozott erőforrású, egy-GPU-s AlphaZero-jellegű rendszerhez. Ez a megoldás felére csökkenti az állapottér komplexitását és feleslegessé teszi a heurisztikus adat-augmentációt. Az önjátszás során a kezdőállások egy részét véletlenszerűen a sötét játékos szemszögéből (FEN-tükrözéssel) veszem, de a kanonikus kódolás miatt ez a hálózat számára ekvivalens bemenetet eredményez. Dolgozatom keretei között ezt a kanonikus nézetet nem hasonlítottam közvetlen, nem-kanonikus baseline-hoz; a választást szakirodalmi és mérnöki szempontok (állapottér-csökkenés, egyszerűbb pipeline) indokolják.

A tanulási ciklus folyamatos, AlphaZero-stílusú tanulást valósít meg. Iteratív lépésekben ismétlődnek az önjátszás-tanítás-értékelés fázisai, a hálózat súlyai pedig a futás teljes időtartama alatt frissülnek, és az önjátszás mindig az éppen legfrissebb (EMA-val súlyozott) modellt használja. A klasszikus AlphaGo Zero-féle aréna-kapuzással szemben itt nincs olyan küszöbérték, amely az arénamérkőzések eredménye alapján engedélyezné vagy elutasítaná a jelölt modellt. Az arénamérkőzések a gyakorlatban kizárólag értékelő

metrikaként szolgálnak, rendszeres időközönként az aktuális modell egy korábbi pillanatfelvétel ellen játszik, az eredményekből pedig győzelmi arányt, döntetlenarányt és futásidőt számolok, és legfeljebb az összehasonlításhoz használt „legjobb” referencia-modellt frissítem; az önjátszásban használt modellválasztásba ez nem avatkozik be:

1. **Önjátszás:** Aszinkron munkaszálak generálják a játszmákat a legfrissebb hálózattal.
2. **Adatgyűjtés:** A generált játszmák egy körkörös visszajátszási pufferbe kerülnek (50 000-es kapacitás, nagy áteresztőképességű konfiguráció esetén 80 000).
3. **Tanítás:** Minden iterációban a GPU mini-kötegekben mintavételez a pufferből, és momentumos SGD optimalizálóval frissíti a hálózat súlyait.
4. **Szinkronizáció:** Az önjátszó szálak rendszeres időközönként átveszik az új súlyokat.

4.2 Módszerek

4.2.1 Sakkmag és bitboard reprezentáció

A rendszer alapját egy nagy teljesítményű C++23 sakkmag képezi. A sakktábla reprezentációjának egyik legelterjedtebb és hatékony módja a **bitboard** technika, amely a klasszikus motorok (pl. Stockfish [30]) és a szakirodalom de facto szabványává vált; itt saját implementációban alkalmazom ugyanezt az elvet. Egy 64 bites egész szám (pl. `uint64_t` C++-ban) pontosan megfeleltethető a sakktábla 64 mezejének. Minden báb típushoz (gyalog, huszár, futó, bástya, vezér, király) és színhez (világos, sötét) külön bittáblát tart a rendszer nyilván, ahol az 1-es bit jelzi a báb jelenlétét az adott mezőn.

A bitboardok előnye, hogy a lépésgenerálás és a táblaműveletek (pl. támadott mezők számítása) bitműveletekkel (AND, OR, XOR, bitshift) párhuzamosan végezhetők el az egész táblán. Például egy huszár összes lehetséges lépése egy adott mezőről előre kiszámítható és egy keresőtáblában (*lookup table*) tárolható. A lépésgenerálás során a

$$\texttt{targets} = \texttt{knight\_attacks[sq]} \;\&\; \sim \texttt{own\_pieces}$$

művelet néhány CPU ciklus alatt megadja az összes legális célmezőt, automatikusan kizárva a saját bábukat tartalmazó mezőket.

A csúszó bábuk (bástya, futó, vezér) esetében a rendszer a **PEXT-alapú bitboard-technikát** (Parallel Bit Extract) alkalmazza BMI2 utasításkészlettel (régebbi CPU-kon ennek szoftveres PEXT/bit-kompressziós megvalósításával). Ez egy hardveresen gyorsított, hash-alapú módszerrel kezeli a blokkoló bábukat, és gyakorlatilag konstans idejű ($O(1)$) lekérdezést biztosít a sugárirányú támadásokhoz.

A sakkmag felelős továbbá a szabályosság ellenőrzéséért (pl. király nem léphet sakkba), a háromszori lépésismétlés és az 50 lépéses szabály detektálásáért, valamint a Zobrist-hash [32] számításáért, amelyet a neurális kiértékelések gyorsítótárazásához (transzpozíciós gyorsítótár) használ a rendszer.

4.2.2 MCTS motor (C++)

A keresőmotor a Monte Carlo Tree Search (MCTS) algoritmust valósítja meg PUCT kiválasztási stratégiával. A C++ implementáció kritikus optimalizációkat tartalmaz a Python-alapú megoldásokhoz képest:

- **Konkurencia-kezelés (virtuális veszteség):** A `VIRTUAL_LOSS = 1.0` paraméterrel a kiválasztott csomópontok értékösszegét ideiglenesen negatív irányba tolja el a rendszer, így az éppen kiértékelés alatt álló útvonalak átmenetileg kevésbé vonzóak a további szimulációk számára. Ez a klasszikus MCTS *virtual loss* technika hatékonyan kezeli a függőben lévő, kötegelt neurális kiértékeléseket, és csökkenti annak esélyét, hogy a párhuzamos szimulációk ugyanarra az ágra koncentrálódjanak.
- **Kötegelt levélkiértékelés:** A levélcsomópontok kötegelve kerülnek kiértékelésre; a C++ oldal int32 tömbökbe gyűjti a pozíciókat, majd egyetlen tenzorként adja át a Python rétegnek.
- **Robusztus memóriakezelés:** A NodePool indexalapú, vektoralapú allokációja biztosítja, hogy az átméretezés a pointerek érvényességének megtartása mellett is skálázható legyen.
- **Nullösszegű visszaterjesztés:** A Backpropagate rutin minden szinten előjelet vált (value $= -$value), biztosítva az ellenfél szemszögének helyes kezelését.

4.2.3 Bemeneti reprezentáció és hálózat

A sakkállásokat az AlphaZero-ban is alkalmazott, síkokra bontott tenzorral kódolja a rendszer. Az aktuális és az azt megelőző 7 lépésállás mindegyikét 14 síkon kódolja a rendszer (összesen $8 \cdot 14 = 112$ sík), valamint további 7 meta-síkot ad hozzá, így a teljes bemenet $119 \times 8 \times 8$ méretű:

- **Előzmény (History):** Az aktuális és az azt megelőző 7 lépésállás (History=8).
- **Pozíció kódolás (14 sík):**
  - 12 sík: Saját/Ellenfél gyalog, huszár, futó, bástya, vezér, király (P, N, B, R, Q, K).
  - 1 sík: En Passant célmező (one-hot jelöléssel).
  - 1 sík: Ismétlődési számláló ≥ 2 (Döntetlen veszély jelzése).

- **Meta-síkok (7 db):** szín, lépésszám, sáncjogok és fél-lépés számláló, ezek több bináris síkon kódolva (pl. külön sík a világos/sötét jogoknak).

A neurális hálózat egy **ResNet v1 architektúrát** követi [12], amely a mély hálózatok tanítását teszi lehetővé az eltűnő gradiens probléma (*vanishing gradient*) enyhítésével. A hálózat alapépítőköve a reziduális blokk (Residual Block).

Egy reziduális blokk a következő rétegekből áll:

- Konvolúciós réteg (3 × 3 kernel, padding=1)
- Batch Normalization
- ReLU aktiváció (Rectified Linear Unit)
- Konvolúciós réteg (3 × 3 kernel, padding=1)
- Batch Normalization
- **Áthidaló kapcsolat (Skip Connection):** A blokk bemenetét hozzáadjuk a második konvolúció kimenetéhez.
- ReLU aktiváció

Az áthidaló kapcsolat (*skip connection*, *identity shortcut*) lehetővé teszi, hogy a gradiens akadálytalanul áramoljon vissza a hálózat elejére a backpropagation során, így sokkal mélyebb hálózatok is hatékonyan taníthatók.

A jelenlegi hálóarchitektúra főbb jellemzői a következők:

- **Törzs:** 6 reziduális blokk, 96 csatorna, 3 × 3 konvolúciók. Ez egyensúlyt teremt a kiértékelési sebesség és a játékerő között a korlátozott hardver környezetben.
- **Policy ág:** 1 × 1 konvolúció (2 csatorna) → Batch Norm → ReLU → Flatten → Fully Connected → 73×64 kimenet (4672 logit). A policy ág 73×64 logitet ad, amelyet a tanítási/inferencia réteg Softmax-szal valószínűségi eloszlássá alakít.
- **Értékbecslő ág (*Value Head*):** 1 × 1 konvolúció (12 csatorna) → Batch Norm → ReLU → Flatten → 256 rejtett neuron (ReLU) → teljesen összekötött réteg (*fully connected*) → 1 kimenet → Tanh. A kimenet $[-1, 1]$ intervallumban becsli a pozíció értékét (1: győzelem, 0: döntetlen, -1: vereség).
- **Inicializálás:** Kaiming Normal (*He*) inicializáció kerül alkalmazásra, amely a ReLU aktivációkhoz optimalizált szórású véletlen súlyokkal indítja a hálózatot.

4.2.4 Tanítási folyamat

A rendszer a „folyamatos tanulás” paradigmát követi. A self-play és a tanítás egy időben zajlik, a hálózat súlyai minden iterációban frissülnek, és az önjátszást mindig a legutóbbi modell végzi.

A veszteségfüggvény:

$$\mathcal{L} = \mathcal{L}_{\text{pol}} + \mathcal{L}_{\text{val}} - \lambda_{\text{entropy}} \cdot \mathcal{H}(p) + \lambda_c \cdot \|\theta\|^2$$

ahol $\mathcal{L}_{\text{pol}}$ a keresztentrópia, $\mathcal{L}_{\text{val}}$ a négyzetes hiba (MSE), $\lambda_{\text{entropy}} \approx 2 \cdot 10^{-4}$ az entrópia-regularizációs tag súlya, amely a negatív előjel miatt a hálózat kimeneti döntési stratégiájának entrópiájának növelésére ösztönöz, gátolva a túl korai konvergenciát. A veszteségfüggvény az AlphaZero-ban használt formulát követi, kiegészítve ezzel az entrópia-taggal. A korábbi AlphaZero-jellegű rendszerekben használt táblaszimmetriákra épülő tükrözéses adataugmentáció helyett a kanonikus nézet biztosítja a szimmetria-invarianciát.

A következő fejezet a fenti architektúra alapján lefuttatott kísérleteket és az azokból származó eredményeket mutatja be.

5. fejezet

Kísérletek és eredmények

A kiértékelés során öt kísérleti forgatókönyv került vizsgálatra, hogy a különböző hiperparaméter-beállítások hatása elkülöníthető legyen a tanulási teljesítményre. Minden mérés azonos hardverkonfiguráción (RTX 3070 Laptop GPU, Ryzen 5800H), kontrollált környezetben futott; a teljes tanítási idők 9 és 18,4 óra között változtak.

Öt célzott konfigurációt definiáltam a `configs/` könyvtárban található beállítások alapján.

5.1. táblázat. Az 5 kísérleti forgatókönyv paraméterei.

Név / Cél	Szimulációk száma	Köteg	PUCT	Zaj (α)
Referencia (kontroll)	128	256	1.35	0.3
Mély Keresés (pontosság)	**256**	256	1.35	0.3
Nagy Áteresztőképesség (sekély keresés)	**64**	256	1.35	0.3
Nagy Entrópia (exploráció)	128	256	**2.5**	**0.5**
Hatékonyság (gyakori frissítés)	128	**128**	1.35	0.3

5.1 Eredmények

5.1.1 RQ1: Rendszer teljesítmény

A C++23-ra történő áttérés és a virtuális veszteségre épülő konkurenciakezelés bevezetése számottevő sebességnövekedést eredményezett (lásd 5.2. táblázat).

5.2. táblázat. Sakkmag teljesítményének összehasonlítása.

Implementáció	Pozíció/mp	Relatív gyorsulás
Python (python-chess) [23]	10 699	1-szeres
C++ (Hybrid Core)	56 887	**5,3-szoros**

A neurális következtetés áteresztőképességét az 5.3. táblázat szemlélteti.

5.3. táblázat. Neurális következtetés teljesítménye (RTX 3070 Laptop GPU).

Eszköz	Precízió	Kötegméret	Pozíció/mp
CPU	Float32	1	570
CPU	Float32	64	2850
GPU (CUDA)	Float16	1	201
GPU (CUDA)	Float16	64	12 715
GPU (CUDA)	Float16	512	81 519

- **Sakkmotor (MoveGen):** A C++ implementáció 56 887 pozíció/mp sebességet ért el, szemben a Python (python-chess) 10 699 pozíció/mp teljesítményével. Ez 5,3-**szoros gyorsulást** jelent.
- **Neurális következtetés:** Az FP16 precízió és a `channels_last` memóriaelrendezés révén a rendszer 512-es kötegméret mellett 81 519 **pozíció/mp** áteresztőképességet ért el.
- **MCTS-skálázódás:** A keresési sebesség (NPS) a kötegméret növelésével mérsékelten nő 64-es méretig, ahol eléri a 25 000 **NPS** körüli csúcsot, majd csökken; a skálázódás nem lineáris.

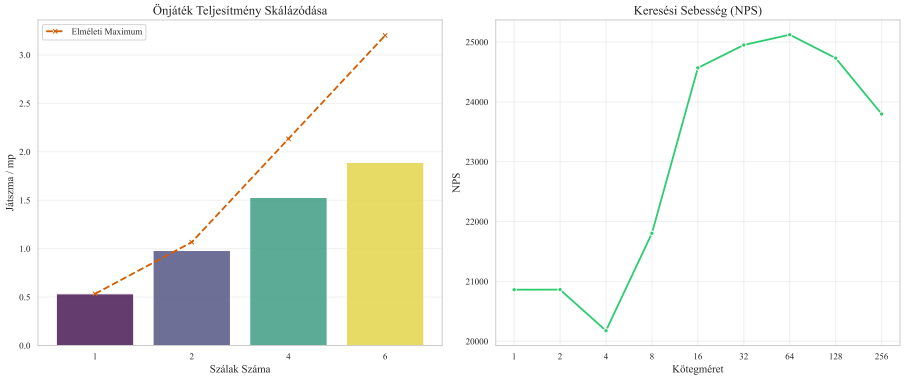
Erőforrás-hatékonyság: A rendszer energiahatékonysága a „Hatékonyság” forgatókönyv alapján került vizsgálatra. A 300 iterációs teljes tanítási ciklus futási ideje 9,1 **óra** volt az RTX 3070 Laptop GPU-n. A becsült energiafogyasztás (220 W TDP mellett) nagyságrendileg 2,0 kW h, ami azt mutatja, hogy a módszertan otthoni környezetben is alkalmazható, mérsékelt energiaigénnyel; a hosszabb futások arányosan több energiát igényeltek.

A mérések alapján az **RQ1**-re válaszolva kijelenthető, hogy a hibrid C++/Python architektúra a sakkmagban 5,3-szoros gyorsulást, a neurális következtetésben pedig nagykötegű futtatás mellett nagyságrendileg tízszeresnél nagyobb sebességnövekedést eredményezett a Python-alapú kiinduló megoldáshoz képest. Ez a teljesítmény elegendő az egynapos önjátszásos tanítási ciklus megvalósításához.

5.1.2 RQ2: Skálázhatóság

Az önjátszás párhuzamosítása során jól megfigyelhető az Amdahl-törvény hatása [1] (5.1. ábra). Az ábra bal oldali grafikonja az önjátszás teljesítményének skálázódását mutatja a szálak számának függvényében. A játszma/mp érték a szálak számával nő (1 szálon ≈0.5, 6 szálon ≈1.8 játszma/mp), azonban a mért gyorsulás jelentősen elmarad az elméleti lineáris maximumtól (szaggatott vonal). 6 szálnál a gyorsulás nagyjából 3.5×, a várható 6× helyett. Ez a Python GIL, a szálak közti szinkronizáció és az adatmozgatás többletköltségei miatt fellépő párhuzamosítási veszteségekre utal.

A jobb oldali grafikon a keresési sebességet (NPS) ábrázolja a kötegméret függvényében. A görbe egyértelmű maximumot mutat 64-es kötegméretnél (≈25 000 NPS), amely az adott hardveren az optimális munkapontot jelenti. Kisebb kötegeknél (1 és 4 közötti kötegméret esetén) a keretrendszer fix költségei dominálnak, ezért csak ∼20 000 NPS érhető el. 8 és 32 között meredek emelkedés figyelhető meg, ahogy a kötegelt feldolgozás jobban kihasználja az erőforrásokat, míg 128 és 256 esetén a várakozási idők, a memóriasávszélesség és a cache-hatások miatt ismét csökken a hatékonyság. A 4 körüli lokális törés, illetve az átmeneti visszaesés arra utal, hogy a kötegelt végrehajtás inicializációs költségei a kis-közepes tartományban még jelentősen terhelik a rendszert.



5.1. ábra. A keresési sebesség (NPS) és az önjátszás skálázódása.

5.4. táblázat. Önjátszás skálázódása munkaszálak függvényében.

Munkaszálak	Játszma/mp
1	0,53
2	0,98
4	1,53
6	**1,89**

A 5.4. táblázat alapján a 6 szálas konfiguráció 3,5-**szörös gyorsulást** nyújt az egyszálúhoz képest, ami alátámasztja a párhuzamos architektúra hatékonyságát, még ha a skálázódás nem is tökéletesen lineáris.

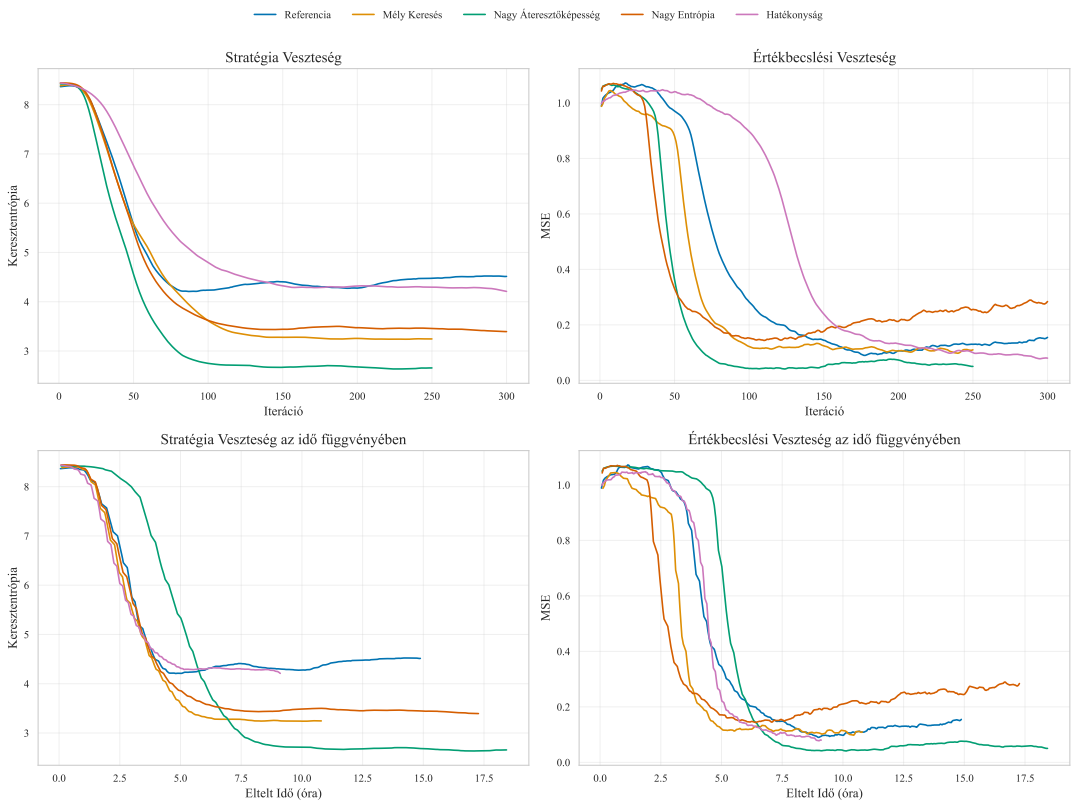
A skálázódást vizsgáló **RQ2** eredményei azt mutatják, hogy az önjátszás teljesítménye 6 munkaszálon kb. 3,5-szörösére nőtt az egyszálú futtatáshoz képest, a keresési sebesség (NPS) pedig 64-es kötegméretig skálázódik jól, ahol eléri a ∼25 000 NPS körüli csúcsot; a további skálázást elsősorban a Python GIL, illetve a host és a device közötti adatmozgatás korlátozza.

5.1.3 RQ3: Tanulási dinamika és bajnokság

A tanulási dinamikát az 5.2. ábra foglalja össze. A bal oldali ábrák a stratégiai (policy) veszteséget, a jobb oldaliak az értékbecslési (value) MSE-t mutatják, felül az iterációk, alul az eltelt futási idő függvényében. A stratégiai veszteség szempontjából a **Nagy Áteresztőképesség** (zöld) konfiguráció adja a legjobb eredményt, a keresztentrópia kb. 2,7 körüli értéken stabilizálódik. Ezt követi a **Mély Keresés** (sárga) és a **Nagy Entrópia** (narancs) 3,3 és 3,5 közötti végértékkel. A **Hatékonyság** (lila) és különösen a **Referencia** (kék) konfigurációk ugyan az első ∼50 iterációban gyorsan csökkennek, utána azonban 4,2 és 4,6 közötti platóra állnak be, ami arra utal, hogy a gyakori frissítések miatt a döntési stratégia viszonylag korán „megfagy”. A görbék lokális fluktuációi természetesek a kis adatmennyiség és a rövid tanítási horizont miatt; az iterációról iterációra történő ingadozás jelentős része tanítási zajnak tekinthető, ezért a konfigurációk közötti különbségek értelmezésekor elsősorban a tartós trendekre (tartósan alacsonyabb, stabil szintekre) támaszkodom.

Az értékbecslési veszteség eltérő mintázatot mutat. A **Nagy Áteresztőképesség** MSE-je nagyon alacsony, ∼0,05 és 0,07 közötti tartományban marad. A **Hatékonyság** konfiguráció lassabban, de hasonlóan alacsony szintre (kb. 0,08 és 0,1 közötti tartományba) konvergál. A **Referencia** és a **Mély Keresés** görbéje valamivel magasabban, nagyjából 0,1 és 0,15 közötti tartományban stabilizálódik, míg a **Nagy Entrópia** esetén a veszteség a 100. iteráció után ismét növekedni kezd, és a tanítás végére ∼0,25 és 0,3 közötti értékre áll be. Ez arra utal, hogy a tartósan magas exploráció ugyan javítja a döntési stratégia sokféleségét, de a value-fej tanulását a zajos, időben változó célok megnehezítik.

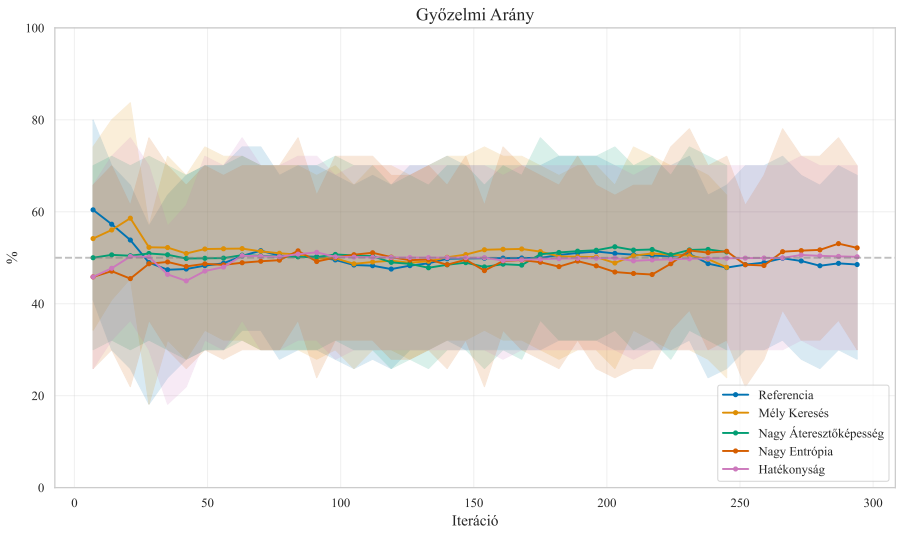
Az időalapú összehasonlítás (alsó sor) a fenti trendeket futásidővel súlyozza. A **Hatékonyság** konfiguráció fejezi be leggyorsabban a tanulást (kb. 9 óra), de közben magas stratégiai veszteségen marad, és a későbbi Elo-mérések alapján ez a konfiguráció teljesít a leggyengébben. A többi konfiguráció 14 és 18 óra közötti futási időt igényel. A **Nagy Áteresztőképesség** és a **Mély Keresés** a leghosszabb, de ezek adják a legalacsonyabb végső stratégiai veszteséget. A **Nagy Áteresztőképesség** így a futásidő és a végső teljesítmény között kedvező kompromisszumot jelent, míg a **Referencia** konfiguráció hosszabb idő alatt is lényegesen rosszabb döntési stratégián marad.



5.2. ábra. Tanulási dinamika: policy- és value-veszteségek az iterációk függvényében.

Az arénamérkőzések eredményeit az 5.3. ábra szemlélteti. Az átlagos győzelmi arány az első 20 és 30 közötti iterációban gyorsan konvergál az 50%-os, kiegyenlített szint közelébe, és ezt követően mind az öt konfiguráció nagyon hasonló tartományban marad. A **Referencia** (kék) kb. 60% körüli kezdő értékről fokozatosan esik vissza ∼50% alá. A **Mély Keresés** (sárga) és a **Nagy Áteresztőképesség** (zöld) végig enyhén 50% fölött, míg a **Nagy Entrópia** (narancs) és a **Hatékonyság** (lila) közvetlenül az 50%-os vonal körül stabilizálódik. A tanítás végére a görbék mindössze néhány százalékpontos különbséggel, nagyjából 48 és 52% között helyezkednek el, ezért a konfigurációk közötti eltérések a mérési bizonytalanságon belül maradnak.

Az árnyékolt sávok az iterációnkénti 24 arénapartiból számolt 95%-os konfidenciaintervallumokat jelölik. A sávok nagy szélessége (tipikusan kb. $\pm$20 és 25 százalékpont közötti érték) azt mutatja, hogy az egyes iterációk közti ingadozások jelentős része statisztikai zaj, nem pedig tartós teljesítménykülönbség. Ez alátámasztja a folyamatos, AlphaZero-stílusú tanulás választását az AlphaGo Zero-féle szigorú aréna-kapuzással szemben. Ilyen zajos becslések mellett az előreléptetési döntések megbízhatatlanok lettek volna, ezért a jelen kísérletekben az arénamérkőzéseket kizárólag mérőszámként használtam, és nem engedtem, hogy a self-play modell kiválasztásáról döntsenek.

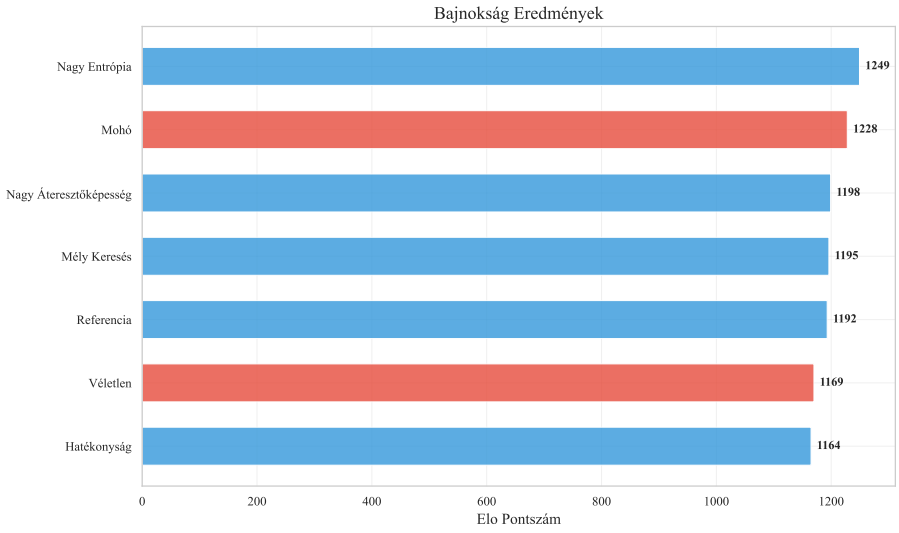


5.3. ábra. Győzelmi arányok alakulása a tanítás során (95% konfidenciaintervallummal).

A modellek relatív játékerejét körmérkőzéses (round-robin) bajnokságban mértem, két kézzel definiált referenciaügynökkel kiegészítve:

- **Véletlen (*Random*):** tisztán véletlenszerű lépések.
- **Mohó (*Greedy*):** egyszerű, anyagelőnyt maximalizáló heurisztika.

A bajnokság végeredményét az 5.4. ábra mutatja Elo-pontszámok formájában.



5.4. ábra. A bajnokság végeredménye (Elo-pontszámok).

A körmérkőzéses bajnokság eredményei szerint a **Nagy Entrópia** konfiguráció (1249 Elo) teljesített a legjobban, egyedüliként megelőzve a **Mohó** (*Greedy*) ágenst (1228 Elo). A többi tanult konfiguráció (**Referencia** 1192, **Mély Keresés** 1195, **Nagy Áteresztőképesség** 1198 Elo) szűk sávban, közvetlenül a Mohó alatt csoportosul; ez jelzi, hogy bár mindegyik modell elsajátított egy használható sakkstratégiát, a rendelkezésre álló időben egyik sem tudott a kézzel tervezett heurisztikánál érdemben erősebb játékot kialakítani.

A **Hatékonyság** modell 1164 Elo-val a mezőny végén végzett, és még a **Véletlen** ágensnél is gyengébbnek bizonyult (1169 Elo; a becsült Elo-pontszám alapján, a mérési zajt figyelembe véve). Ez arra utal, hogy a túl agresszív frissítési stratégia a hálózat instabil, irányt tévesztett tanulásához vezethet (katasztrofális felejtés vagy rossz lokális optimum), ezért az ilyen beállítások gyakorlatban kerülendők.

Fontos ugyanakkor kiemelni, hogy a bajnokság korlátozott játszmaszáma miatt az Elo-becslések bizonytalansága várhatóan több tíz pont nagyságrendű. A kisebb különbségek (például 1164 versus 1169 Elo) ezért inkább jelzésértékűek, mintsem statisztikailag egyértelműen szignifikáns eltérések; a konfigurációk relatív sorrendje megbízhatóbb információt ad, mint az abszolút értékek.

A tanulási dinamikára vonatkozó **RQ3** kapcsán az látszik, hogy a policy- és value-veszteséggörbék mind az öt konfiguráció esetén jól látható csökkenést mutatnak, különösen a **Nagy Áteresztőképesség** és **Mély Keresés** beállításoknál, ami azt jelzi, hogy rövid, 9 és 18 óra közötti tanítási horizonton kimutatható, stabil tanulás zajlik. Az 5.2. ábra görbéi alapján a policy-veszteség a tanítás elején mért értékéhez képest a legtöbb konfigurációnál nagyságrendileg 40%-os, míg az értékveszteség 50%-os relatív javulást mutat; ez vizuálisan összhangban van a 3. fejezetben megfogalmazott sikerkritériumok nagyságrendjével.

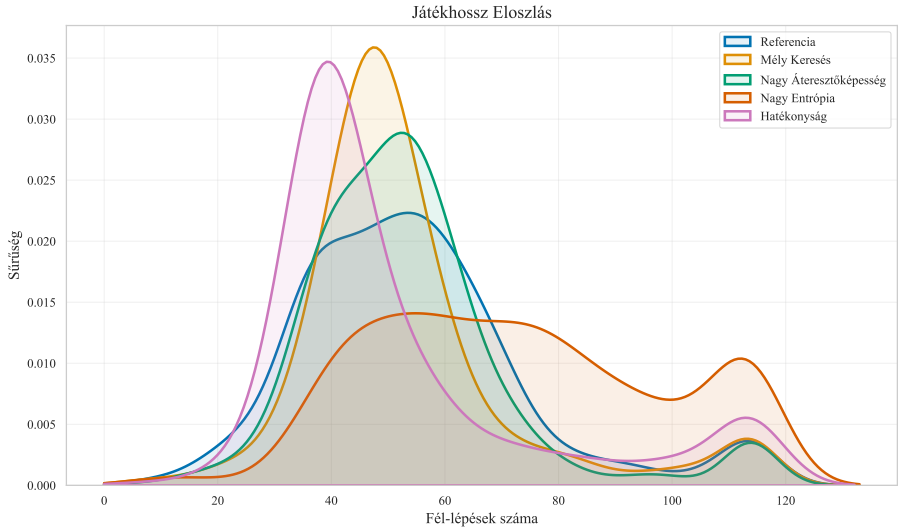
5.1.4 RQ4: Adatdiverzitás

Az 5.5. ábra a lejátszott játszmák hosszának eloszlását mutatja fél-lépésekben. A **Mély Keresés** (sárga) konfiguráció sűrűségfüggvénye adja a legmagasabb csúcsot. A maximum kb. 0,036 körül alakul, nagyjából 45 és 50 fél-lépés között. A **Hatékonyság** (lila) eloszlása a legkeskenyebb és legkoncentráltabb, csúcsa kb. 40 fél-lépés körül jelenik meg, ami arra utal, hogy a gyakori gradiens-frissítések (`samples_per_new_game` = 8) és a kisebb kötegméret gyors, de viszonylag egyoldalú konvergenciához vezettek, így a játszmák többsége hasonló hosszúságú, kevés változatos végjátékkal.

A legmarkánsabban eltérő viselkedést a **Nagy Entrópia** (narancs) konfiguráció mutatja, amelynek eloszlása egyértelműen *bimodális*. Egy első csúcs látható kb. 50 fél-lépésnél, míg egy második, alacsonyabb csúcs kb. 110 fél-lépés körül jelenik meg. Az első csúcs a középjátékban lezáruló partikat, a második a hosszú, végjátékig elnyúló játszmákat reprezentálja. A magas exploráció ($c_{\text{puct}} = 2.5$, Dirichlet $\alpha = 0.5$) egyrészt lehetővé teszi a döntő taktikai ütközéseket, másrészt gyakran megakadályozza a korai lépésismétléses döntetleneket, így széles skálán, sokféle pozíciót lefedve tud tanulni a modell. Ez a diverzitás összhangban van a konfiguráció kiemelkedő Elo-eredményével.

A **Mély Keresés** (sárga) és a **Nagy Áteresztőképesség** (zöld) eloszlásai hasonló, egymóduszú görbéket adnak. A tömegük zöme 40 és 60 fél-lépés között helyezkedik el, a **Nagy Áteresztőképesség** csúcsa azonban kissé jobbra tolódik, ami arra utal, hogy a sekélyebb keresés (64 szimuláció) ritkábban dönt el partikat már a középjátékban. A **Referen-**

cia (kék) konfiguráció eloszlása a két szélsőség között helyezkedik el, mind csúcsmagasságban, mind pozícióban, ami jól illeszkedik a referencia jellegéhez.



5.5. ábra. A lejátszott játszmák hosszának eloszlása (fél-lépésekben).

Az adatdiverzitást firtató **RQ4**-re a válasz, hogy a magas explorációs paraméterekkel futó **Nagy Entrópia** konfiguráció sokkal változatosabb, bimodális játszmahossz-eloszlást produkál, ami szélesebb pozíciókészletet eredményez. A Nagy Entrópia konfiguráció bimodális játszmahossz-eloszlása jól rezonál az RQ3 alatt mért Elo-előnnyel. A sokszínűbb tanítópozíciók rövid tanítási horizonton fontosabbnak bizonyultak, mint a puszta keresési mélység növelése.

5.1.5 RQ5: Reprodukálhatóság

A reprodukálhatóságot három szinten támogattam: (1) verziózott konfigurációs állományokkal, (2) expliciten kezelt véletlenszám-generátorokkal és determinisztikus beállításokkal, valamint (3) teljesen automatizált benchmark- és ábrageneráló eszközlánccal.

Konfigurációk és futtatási környezet. Minden kísérleti futás `YAML`-alapú konfigurációs fájlokból indul, amelyeket a `main.py` belépési pont `--config` és `--override` kapcsolókkal rétegez egymásra. A konfigurációkezelő komponens minden futás elején egy *végleges*, már felülbírált konfigurációt állít elő, amelyet a rendszer a `runs/<azonosító>/config/merged.{yaml,json}` állományokba ment. Így a dolgozatban szereplő öt forgatókönyv (Referencia, Mély Keresés, Nagy Áteresztőképesség, Nagy Entrópia, Hatékonyság) minden hiperparaméter-beállítása (MCTS-szimulációk száma, kötegméret, explorációs paraméterek stb.) visszakereshető és más gépen is pontosan újrahasználható. A Python-környezet főbb csomagjai a `requirements.txt`-ben rögzítettek; a kritikus függőség, a PyTorch, fix verzióval (`torch==2.8.0`) szerepel.

Véletlenszám-generátorok és determinisztika. A rendszer globális `SEED` paramétere (`config.SEED`) határozza meg, hogy a futás determinisztikus módban történjen-e. Amennyiben a `SEED` értéke nem nulla, a `main.py` a tanítás indításakor inicializálja a Python beépített `random` modulját, a `NumPy` véletlenszám-generátorát, valamint a `torch` CPU- és GPU-RNG-it a megadott maggal. Ezzel párhuzamosan beállítja a `torch.use_deterministic_algorithms(True)` opciót, a `CUBLAS_WORKSPACE_CONFIG` környezeti változót, letiltja a `cuDNN` adaptív `benchmark` módját, illetve a logikai processzorszámhoz igazítja az intra- és inter-op szálak számát. Ez a kombináció gyakorlatban közel determinisztikus viselkedést eredményez ugyanazon hardveren, viszont egyes, GPU-n futó műveletek és a párhuzamos önjátszás miatt bitpontosan azonos futások nem minden esetben garantálhatók. A teljesítménymérések és Elo-becslések emiatt várhatóan néhány tíz pont nagyságrendű szórást mutatnak.

Az elemző és benchmark szkriptek külön dedikált magokat használnak. A `tools/benchmarks.py` modulban az inferencia-benchmark rögzített `2025`-ös Torch-maggal fut, az önjátszás áteresztőképességét mérő komponens a `SelfPlayEngine` belső RNG-jét `123`-as maggal inicializálja, míg a C++ sakkmag- és MCTS-benchmarkok véletlen FEN-állásait szintén paraméterezhető, alapértelmezésben `2025`-ös mag vezérli. Az arénamérkőzésekre épülő bajnokság `NumPy default_rng(2025)` generátorral készül, így a párosítások sorrendje és a lépésválasztások sztochasztikus komponense is kontrollált.

Artefaktum- és ábragenerálás. Minden, a dolgozatban szereplő tanítási metrika, eloszlás és Elo-összehasonlítás automatikusan generált. A tanítás során a rendszer iterációnként CSV- és JSONL-formátumban naplózza a legfontosabb mutatókat (veszteségek, önjátszás- és tanítási idők, puffer- és resign-statisztikák), az arénamérkőzések PGN-fájljai pedig iterációnként elkülönített könyvtárakba kerülnek. A `tools/generate_artifacts.py` szkript ezeket az állományokat olvassa be, a konfigurációs metaadatokkal együtt, majd egységes betűtípus- és stílusbeállítások mellett automatikusan készíti el az ábrákat (tanulási dinamika, skálázódási görbék, játszmahossz-eloszlás, bajnoki Elo-eredmények).

A fenti mechanizmusok eredményeként a kísérletek reprodukálása a gyakorlatban a következő, jól definiált lépésekre bontható:

1. A kód és a `requirements.txt` alapján a futtatási környezet felépítése, azonos vagy hasonló GPU-val.
2. A kívánt konfiguráció kiválasztása (pl. `configs/run4_high_exploration.yaml`) és a teljes tanítás lefuttatása a `main.py` szkripttel, opcionálisan rögzített `SEED` értékkel.
3. A benchmark-szkriptek (`tools/benchmarks.py`) futtatása opcionális teljesítménymérésekhez.

4. Az ábrák teljes automatikus újragenerálása a `tools/generate_artifacts.py` eszközzel, amely a `runs/` könyvtárból olvassa ki a metrikákat.

A rendszer felépítése, a verziózott konfigurációk, az explicit RNG-kezelés és az automatizált artefaktum-pipeline együttesen biztosítja, hogy az itt bemutatott kísérleti eredmények azonos vagy hasonló hardverkörnyezetben számottevően kis szórással, gyakorlatban reprodukálhatók legyenek.

5.2 Összefoglalás

A mérések szerint a rendszer technikai teljesítménye (56 887 pozíció/mp, 25 000 NPS) a kitűzött kutatási célok szempontjából elegendőnek bizonyult. A kísérleti eredmények alapján korlátozott erőforrások mellett az **exploráció maximalizálása** a kulcs a játékerő növeléséhez, nem pedig a puszta számítási mélység vagy a frissítési sebesség.

6. fejezet

Megbeszélés és korlátok

Ebben a fejezetben az RQ1-től RQ5-ig terjedő kérdésekre kapott eredményeket értelmezem, majd kitérek a módszertan korlátaira.

6.1 Eredmények értelmezése

6.1.1 Az exploráció dominanciája

Az exploráció dominanciája elsősorban az RQ3 (tanulási dinamika) és RQ4 (adatdiverzitás) szempontjából releváns. Korlátozott erőforrás mellett a változatos adatgenerálás fontosabbnak bizonyult, mint a puszta keresési mélység. A magas explorációs beállítások ($\alpha = 0.5$, $c_{\text{puct}} = 2.5$) megakadályozták, hogy a rendszer túl korán passzív stratégiákhoz tapadjon. Míg a Hatékonyság forgatókönyv gyorsan konvergált egy szűk, de gyenge stratégiára (és ezzel elveszítette rugalmasságát), a Nagy Entrópia ágens szélesebb körű keresést végzett. Ez stabilabb játékot eredményezett, miközben a Hatékonyság konfiguráció a Random referencia szintjén vagy az alatt végzett, érdemi előny nélkül.

6.1.2 A rendszer skálázhatósága

A rendszer skálázhatósága (RQ1, RQ2) tekintetében a kötegelt levélértékelés és a virtuális veszteségre épülő konkurenciakezelés együtt 3,5**-szörös** önjátszás-gyorsulást adott 6 szálon, miközben megőrizte a fa diverzitását. A NodePool indexalapú, vektoralapú allokációja biztosítja, hogy az átméretezés a pointerek érvényessége mellett is skálázható legyen. A fő szűk keresztmetszetek a Python GIL és a host és a device közötti adatmozgatás jelentik, ami jelzi, hogy több GPU vagy teljesen natív önjátszás további gyorsulást hozhat.

6.2 Korlátok

6.2.1 Abszolút játékerő

Bár a rendszer a saját, zárt skálán mért 1249 Elo-pontszámmal legyőzte a mohó (Greedy) ágenst, ez a szint nagyjából egy erősebb amatőr játékos szintjének felel meg, és nem hasonlítható közvetlenül sem a FIDE-, sem az online szerverek Elo-értékeihez. Az 1249-es Elo-érték így kizárólag a dolgozatomban definiált, zárt bajnokságon belül értelmezhető. A modell képes alapvető pozíciós elvek követésére (például a gyalogstruktúra védelmére), de hiányzik belőle a mély taktikai számolás képessége, amelyhez nagyságrendileg több (milliós nagyságrendű) önjátszásos játszmára lenne szükség. A szakirodalomban tipikusan több tízezer, sőt akár milliós nagyságrendű játszmára épülő, standardizált ellenfelek elleni méréseket használnak FIDE-közeli skálákhoz; dolgozatomban ilyen típusú, külső, Stockfish-alapú Elo-kalibrációt tudatosan nem végeztem.

Ez összhangban áll a Célkitűzések fejezetben megfogalmazott korlátozott hatókörrel. A cél a módszertan megvalósíthatóságának demonstrálása volt, nem pedig versenyképes motor fejlesztése.

6.2.2 Számítási horizont

A 300 iteráció és a 9 és 18 óra közötti futási idő a „megvalósíthatósági bizonyítás” kategóriájába esik. Az AlphaZero eredeti tanulmánya több ezer TPU-t használt, nagyságrendekkel nagyobb önjátszásos adatmennyiség mellett. Bár kísérleti validációval sikerült igazolni a módszertant, a versenyképes szint elérése ezen a hardveren hónapokat venne igénybe.

7. fejezet

Összefoglalás

Dolgozatomban egy fogyasztói hardverre optimalizált, AlphaZero-típusú sakkprogramot mutattam be. A rendszer C++23-alapú keresőt és PyTorch-alapú tanítási modult egyesít, kifejezetten egy RTX 3070 Laptop GPU környezetére tervezve.

A fejlesztés három fő pillére az architekturális egyszerűsítés (a „Mindig világos” kanonikus nézet), a virtuális veszteségre épülő konkurenciakezelés és a kötegelt levélértékelés volt. A kanonikus nézet a bemeneti állapottér felére csökkentésével egyszerű, mégis hatékony mérnöki kompromisszumot jelent, amely jól illeszkedik a rendelkezésre álló hardverkorlátokhoz. Munkámban a hatását elsősorban kvalitatív módon vizsgálom, ugyanakkor nem törekszem a többi architekturális döntéstől való szigorú, kvantitatív elválasztására. A három pillér együttesen teszi lehetővé, hogy a rendszer korlátozott erőforrások mellett is elegendő keresési sebességet és áteresztőképességet érjen el. Az öt konfigurációt lefedő kísérleti validáció alapján a magas entrópiájú keresés bizonyult a leghatékonyabbnak a rendelkezésre álló adat- és időkeret mellett.

Az RQ1-től RQ5-ig terjedő kutatási kérdésekre adott válaszokat az alábbiakban foglalom össze:

1. **RQ1, Teljesítmény:** a C++23-alapú sakkmag 56 887 pozíció/mp sebességet ért el, míg a GPU-alapú neurális következtetés 81 519 pozíció/mp áteresztőképességet biztosított nagy kötegméreteknél. Ez legalább 5,3-szoros gyorsulást jelent a sakkmag esetén a Python-alapú megoldáshoz képest, és nagyságrendileg tízszeresnél nagyobb gyorsulást a neurális következtetésben, ami elegendő az önjátszásos tanításhoz egynapos időkeretben.
2. **RQ2, Skálázhatóság:** az önjátszás teljesítménye 6 munkaszálon kb. 3,5-szörösére nőtt az egyszálú futtatáshoz képest, a keresési sebesség (NPS) pedig 64-es kötegméretig skálázódik jól, ahol eléri a $\sim$25 000 NPS körüli csúcsot; a további skálázást a Python GIL és a host és a device közötti adatmozgatás korlátozza.
3. **RQ3, Tanulási dinamika:** a policy- és value-veszteség mind az öt vizsgált konfiguráció esetén jelentősen csökkent, különösen a **Nagy Áteresztőképesség** és **Mély**

Keresés beállításoknál, ami azt mutatja, hogy rövid tanítási horizonton is érdemi, mérhető tanulás zajlik.

4. **RQ4, Adatdiverzitás:** a magas explorációs beállításokkal futó **Nagy Entrópia** konfiguráció bimodális játszmahossz-eloszlást és sokkal változatosabb tanítóadatot eredményezett, és a mérések alapján kis előnnyel felülmúlta a mohó heurisztikát a saját Elo-skálán, ami alátámasztja az adatdiverzitás szerepét korlátozott erőforrások mellett. Az itt közölt Elo-értékek a korlátozott játszmaszám miatt becslések, várhatóan néhány tíz pontos bizonytalansággal.
5. **RQ5, Reprodukálhatóság:** a teljesen automatizált benchmark- és artefaktumgeneráló szkriptek, valamint a YAML-konfigurációk rögzítik a hardver- és hiperparaméter-beállításokat, így a kísérleti futtatások más környezetben is megismételhetők, legfeljebb kisebb, a sztochasztikusságból adódó Elo-szórással.

Dolgozatom elsődleges hozzájárulása egy keretrendszer, amely referenciapontot nyújt az AlphaZero-módszertan korlátozott erőforrású megvalósításához. A jövőbeli fejlesztési irányok közé tartozik a jelenlegi 6 blokkos háló fokozatos skálázása 10 és 20 blokk közötti méretig, és annak vizsgálata, hogy az itt bemutatott paraméterezés mellett hogyan változik a tanulási dinamika. További lehetőség a tanult környezetmodellek (pl. MuZero-típusú megközelítések) bevezetése, ahol a környezetmodell is tanulható lenne. Emellett fontos jövőbeli lépés lehet a rendszer adaptálása elosztott (kliens-szerver) környezetre. Ezek a fejlesztések közelebb vihetnek egy olyan AlphaZero-típusú sakkmotorhoz, amely már nemcsak kísérleti, hanem gyakorlati szempontból is versenyképes játékerőt képvisel.

Irodalomjegyzék

[1] Amdahl, G. M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, 483–485 (1967). doi:10.1145/1465482.1465560

[2] Auer, P., Cesa-Bianchi, N., Fischer, P. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47(2–3):235–256 (2002). doi:10.1023/A:1013689704352

[3] Bengio, Y., Louradour, J., Collobert, R., Weston, J. Curriculum Learning. In: *Proceedings of the 26th International Conference on Machine Learning (ICML 2009)*, 41–48 (2009). doi:10.1145/1553374.1553380

[4] Browne, C. B., Powley, E., Whitehouse, D., *et al.* A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43 (2012). doi:10.1109/TCIAIG.2012.2186810

[5] Campbell, M., Hoane, A. J., Hsu, F.-H. Deep Blue. *Artificial Intelligence* 134(1–2):57–83 (2002). doi:10.1016/S0004-3702(01)00129-1

[6] Chess Programming Wiki. Late Move Reductions. https://www.chessprogramming.org/Late_Move_Reductions. Utolsó megtekintés: 2025. 12. 07.

[7] Chess Programming Wiki. Null Move Pruning. https://www.chessprogramming.org/Null_Move_Pruning. Utolsó megtekintés: 2025. 12. 07.

[8] Coulom, R. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: *Proceedings of the 5th International Conference on Computers and Games (CG 2006)*, LNCS 4630, 72–83 (2007). doi:10.1007/978-3-540-75538-8_7

[9] Czech, J., Willig, M., Beyer, A., Kersting, K., Fürnkranz, J. Learning to Play the Chess Variant Crazyhouse Above World Champion Level with Deep Neural Networks and Human Data. *Frontiers in Artificial Intelligence* 3:24 (2020). doi:10.3389/frai.2020.00024

[10] Gelly, S., Wang, Y. Exploration Exploitation in Go: UCT for Monte-Carlo Go. In: *NIPS 2006 Workshop on Online Trading of Exploration and Exploitation*, Whistler, Canada (2006).

[11] Gelly, S., Silver, D. Combining Online and Offline Knowledge in UCT. In: *Proceedings of the 24th International Conference on Machine Learning (ICML 2007)*, 273–280 (2007). doi:10.1145/1273496.1273531

[12] He, K., Zhang, X., Ren, S., Sun, J. Deep Residual Learning for Image Recognition. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*, 770–778 (2016). doi:10.1109/CVPR.2016.90

[13] Hu, J., Shen, L., Sun, G. Squeeze-and-Excitation Networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2018)*, 7132–7141 (2018). doi:10.1109/CVPR.2018.00745

[14] Jakob, W., Rhinelander, J., Moldovan, D. pybind11: Seamless operability between C++11 and Python. https://github.com/pybind/pybind11. Utolsó megtekintés: 2025. 12. 07.

[15] Knuth, D. E., Moore, R. W. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence* 6(4):293–326 (1975). doi:10.1016/0004-3702(75)90019-3

[16] Kocsis, L., Szepesvári, C. Bandit Based Monte-Carlo Planning. In: *Machine Learning: ECML 2006*, LNCS 4212, 282–293 (2006). doi:10.1007/11871842_29

[17] Leela Chess Zero (Lc0). Open-source neural network chess engine. Project: https://lczero.org/. GitHub: https://github.com/LeelaChessZero. Utolsó megtekintés: 2025. 12. 07.

[18] Lichess. *chess-openings dataset*. https://github.com/lichess-org/chess-openings. Utolsó megtekintés: 2025. 12. 07.

[19] Loshchilov, I., Hutter, F. SGDR: Stochastic Gradient Descent with Warm Restarts. In: *Proceedings of the 5th International Conference on Learning Representations (ICLR 2017)*. https://openreview.net/forum?id=Skq89Scxx. Utolsó megtekintés: 2025. 12. 07.

[20] Micikevicius, P., Narang, S., Alben, J., *et al.* Mixed Precision Training. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*. https://openreview.net/forum?id=r1gs9JgRZ. Utolsó megtekintés: 2025. 12. 07.

[21] Nasu, Y. Efficiently Updatable Neural-Network-based Evaluation Function for computer Shogi. Technical report, 28th World Computer Shogi Championship, 2018. https://github.com/asdfjkl/nnue. Utolsó megtekintés: 2025. 12. 07.

[22] Paszke, A., Gross, S., Massa, F., *et al.* PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32* (NeurIPS 2019), 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html. Utolsó megtekintés: 2025. 12. 07.

[23] Fiekas, N. python-chess: a chess library for Python. https://python-chess.readthedocs.io. Utolsó megtekintés: 2025. 12. 07.

[24] Schrittwieser, J., Antonoglou, I., Hubert, T., *et al.* Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature* 588(7839):604–609 (2020). doi:10.1038/s41586-020-03051-4

[25] Shannon, C. E. Programming a Computer for Playing Chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41(314):256–275 (1950). doi:10.1080/14786445008521796

[26] Silver, D., Huang, A., Maddison, C. J., *et al.* Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529(7587):484–489 (2016). doi:10.1038/nature16961

[27] Silver, D., Schrittwieser, J., Simonyan, K., *et al.* Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint* arXiv:1712.01815 (2017).
https://arxiv.org/abs/1712.01815.
Utolsó megtekintés: 2025. 12. 07.

[28] Silver, D., Schrittwieser, J., Simonyan, K., *et al.* Mastering the Game of Go without Human Knowledge. *Nature* 550(7676):354–359 (2017).
doi:10.1038/nature24270

[29] Silver, D., Hubert, T., Schrittwieser, J., *et al.* A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play. *Science* 362(6419):1140–1144 (2018).
doi:10.1126/science.aar6404

[30] Stockfish developers. Stockfish chess engine.
https://stockfishchess.org.
Utolsó megtekintés: 2025. 12. 07.

[31] Wu, D. J. Accelerating Self-Play Learning in Go. *arXiv preprint* arXiv:1902.10565 (2019).
https://arxiv.org/abs/1902.10565.
Utolsó megtekintés: 2025. 12. 07.

[32] Zobrist, A. L. A New Hashing Method with Application for Game Playing. *Technical Report 88*, Computer Sciences Department, University of Wisconsin, Madison, WI (1970).
https://research.cs.wisc.edu/techreports/1970/TR88.pdf.
Utolsó megtekintés: 2025. 12. 07.

[33] Harris, C. R., *et al.* Array Programming with NumPy. *Nature* 585(7825):357–362 (2020).
doi:10.1038/s41586-020-2649-2

[34] Hunter, J. D. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* 9(3):90–95 (2007).
doi:10.1109/MCSE.2007.55

[35] Waskom, M. L. Seaborn: Statistical Data Visualization. *Journal of Open Source Software* 6(60):3021 (2021).
doi:10.21105/joss.03021

Nyilatkozat

Alulírott Sere Gergő Márk programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Számítástudomány Alapjai Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Szeged, 2025. december 7.

_____

aláírás

Köszönetnyilvánítás

Köszönettel tartozom témavezetőmnek a szakmai iránymutatásért és az értékes visszajelzésekért. Hálás vagyok családomnak és barátaimnak a támogatásért és bátorításért. Köszönet illeti a fejlesztői közösséget is a nyílt forráskódú eszközök és könyvtárak elérhetővé tételéért, amelyek e munka szakmai alapját adták.