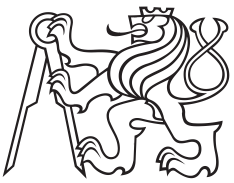
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

Šachové algoritmy využívající hluboké neuronové sítě

Lukáš Hejl
Otevřená informatika
Informatika a počítačové vědy

Květen 2017
Vedoucí práce: Mgr. Branislav Bošanský, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Lukáš H e j l

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Šachové algoritmy využívající hluboké neuronové sítě

Pokyny pro vypracování:

Nově vyvinutý algoritmus pro hraní deskové hry Go nedávno porazil světového mistra v této hře. Tento výsledek byl dosažen díky kombinaci heuristické funkce naučené pomocí hlubokých neuronových sítí (DNN) a herně-teoretického algoritmu, Monte Carlo Tree Search (MCTS). Podobný přístup byl následně aplikován i pro jiné hry, včetně šachu. V šachových algoritmech se, na rozdíl od hry Go, nepoužívá MCTS algoritmus a je tak nutné prozkoumat možnosti integrace naučených heuristik do standardních algoritmů založených na alfa-beta prořezávání. Cílem studenta je proto (1) prozkoumat a implementovat metody pro naučení evaluačních heuristik v šachu s využitím hlubokých neuronových sítí, (2) implementovat alespoň 2 algoritmy (jeden založený na alfa-beta prořezávání, jeden založený na MCTS), (3) experimentálně porovnat jejich výkonnost s existujícími programy.

Seznam odborné literatury:

[1] Lai, Matthew. "Giraffe: Using Deep Reinforcement Learning to Play Chess", arXiv:1509.01549, 2015
[2] David, Omid E. and Netanyahu, Nathan S. and Wolf, Lior. "DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess" ICANN 2016

Vedoucí bakalářské práce: Mgr. Branislav Bošanský, Ph.D.

Platnost zadání: do konce letního semestru 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 12. 1. 2017

Poděkování / Prohlášení

Chtěl bych především poděkovat mému vedoucímu práce Mgr. Branislavovi Bošanskému, Ph.D. za ochotu a cenné rady. Dále bych chtěl poděkovat Ing. Janu Drchalovi, Ph.D. za cenné rady ohledně neuronových sítích. Také bych chtěl poděkovat svojí rodině za podporu během mého studia a psaní této práce.

Dále bych rád poděkoval za přístup k výpočetním a úložným zařízením ve vlastnictví stran a projektům přispívající do the National Grid Infrastructure MetaCentrum poskytované v rámci programu „Projects of Large Research, Development, and Innovations Infrastructures“ (CESNET LM2015042).

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 26. 5. 2017

.....

Abstrakt / Abstract

Nedávno vyvinutý algoritmus AlphaGo pro hraní hry Go, porazil světového mistra v této hře. Tohoto úspěchu bylo dosaženo pomocí heuristických funkcích založených na hlubokých konvolučních neuronových sítí s využitím algoritmu Monte Carlo Tree Search. V šachách bylo dosaženo výborných výsledků s využitím hluboké neuronové sítě, která porovnávala šachové pozice s přesností 98 % a byla integrována do algoritmu Alpha-Beta prořezávání.

Tato práce se zaměřila na replikaci těchto výsledků a na experimentální porovnání algoritmů Alpha-Beta prořezávání a Monte Carlo Tree Search ve hře šachy.

Zmíněné výsledky se zcela nepodařilo zreplikovat. Naučená sít dosáhla při porovnání jen na přesnost 93,9 %. Proto, aby porovnání algoritmů mělo vypovídající hodnotu, byla také použita neuronová sít z šachového programu Giraffe, která ohodnocuje šachové pozice.

Na základě provedených experimentů nedosáhl algoritmus Monte Carlo tree search i přes použité heuristiky na úroveň algoritmu Alpha-Beta prořezávání. Algoritmus Alpha-Beta prořezávání dosáhl s využitím heuristik na úroveň přibližně kolem 1900 Elo podle žebříčku CCRL 40/40.

Klíčová slova: neuronové sítě, alphabeta prořezávání, monte carlo tree search, šachy

AlphaGo, a recently developed algorithm to play the game Go, has defeated a human world champion in the game. This success has been achieved with heuristic functions based on deep convolutional neural networks used with Monte Carlo Tree Search algorithm. In chess, there were excellent results with deep neural networks, that compared chess positions with 98% accuracy and were integrated into Alpha-Beta prunning algorithm.

This thesis focused on replication of these results, and on experimental comparison of algorithms Alpha-Beta prunning and Monte Carlo Tree Search in chess.

The mentioned results were not fully replicated. The trained network achieved only 93.9% accuracy of comparison. In order to reasonably compare the algorithms, a neural network from chess engine Girraffe, that evaluated chess positions, was used.

Based on the performed experiments, the Monte Carlo tree search algorithm could not achieve the level of the Alpha-Beta prunning algorithm despite the heuristics used. The Alpha-Beta pruning algorithm with use of heurisitcs achieved level around 1900 Elo according to the CCRL 40/40 ladder.

Keywords: neural networks, alphabeta pruning, monte carlo tree search, chess

Title translation: Chess-Playing Algorithms with Deep Neural Network Heuristics

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

V roce 2016 AlphaGo [1] porazilo mistra světa ve hře Go. Úspěchu bylo dosaženo pomocí heuristických funkcích založených na hlubokých konvolučních neuronových sítí integrovaných do algoritmu Monte Carlo Tree Search.

V šachách byl tehdejší mistr světa Garri Kasparov poražen již v roce 1997 počítačem Deep Blue vyvinutého firmou IBM, který neuronové sítě nevyužil. Použitím neuronových sítí pro šachy se zabývají články [2] a [3], ve kterých bylo dosaženo výborných výsledků, což poukazuje na možnost využití neuronových sítích pro šachy podobně jako ve hře Go. V šachách jsou ale na rozdíl od hry Go používané algoritmy založené na Alpha-Beta prořezávání.

1.1 Cíle práce

Práce si klade za cíl:

- Naučit heuristiky založené na hlubokých neuronových sítích na základě článku[3]
- Integrovat naučené heuristiky do algoritmů založených na Alpha-Beta prořezávání a Monte Carlo Tree Search
- Experimentálně porovnat implementované algoritmy s několika existujícími šachovými programy

Kapitola 2
Teorie her

Tato kapitola se zabývá teoretickým úvodem do problematiky teorie her. V této kapitole jsou také popsané algoritmy pro prohledávání stavového prostoru, které jsou v této práci implementovány. Jimiž jsou Alpha-Beta prořezávání a Monte Carlo tree search.

2.1 Hry s úplnou informací

Hry s úplnou informací [4] jsou takové hry, kde všichni hráči mají k dispozici informace o celém stavu hry (např. rozestavění figurek, předchozí tahy atd.) a které nejsou založené na pravděpodobnosti a jsou tudíž deterministické.

Mezi hry s úplnou informací patří např. Šachy, Go, Dáma a mnoho dalších.

2.2 Hry s nulovým součtem

Hry s nulovým součtem [4] jsou takové dvouhráčové hry, kde zisk jednoho hráče odpovídá ztrátě hráče druhého. V takovýchto hrách se hráčům nevyplatí spolupracovat, tudíž se jedná o ryze kompetitivní hry.

Příkladem hry s nulovým součtem jsou šachy.

2.3 MiniMax

MiniMax [4] je rekurzivní algoritmus založený na prohledávání do hloubky. Hledající nejlepší tah pro zadaný stav hry. Během prohledávání stavového prostoru se střídají MAX a MIN úrovně. V každé MAX úrovni, algoritmus vybírá maximální hodnotu z nižší úrovně (MIN). Tato úroveň odpovídá hráči, který se snaží zahrát tah, který maximalizuje jeho zisk. V každé MIN úrovni, algoritmus vybírá minimální hodnotu z nižší úrovně (MAX). Tato úroveň odpovídá protihráči, který se naopak snaží zahrát tah, který minimalizuje zisk protihráče.

V hrách jako jsou šachy, není možné prohledat celý stavový prostor hry. Protože šachy mají jednak velký faktor větvení, tak se i běžná šachová partie skládá z přibližně 80 tahů[5]. Proto je nutné zavést omezení na prohledávanou hloubku stavového prostoru.

Při dosažené této hloubky algoritmus už dál neprohledává a provede ohodnocení prohledávaného stavu pomocí heuristické ohodnocovaní funkce. Cílem je, aby tato heuristická funkce byla co nejpřesnější, protože má zásadní vliv na sílu šachového programu. Časová složitost algoritmu je $O(b^d)$ kde b je faktor větvení odpovídající validním tahům a d je prohledávaná hloubka (resp. délka hry). Pro šachy je faktor větvení roven přibližně 35 a délka hry je přibližně 80 tahů[5].

```
function MiniMax(state, depth, maximizingPlayer)
    if isTerminal(state) then
        return GameResult(state)
    if depth == 0 then
        return Evaluate(state)

    if maximizingPlayer then
        bestMoveValue := -inf
        for each legal move of state do
            nextState = DoMove(state, move)
            moveValue = MiniMax(nextState, depth - 1, False)
            UndoMove(nextState, move)
            if moveValue > bestMoveValue then
                bestMoveValue := moveValue
    else
        bestMoveValue := inf
        for each legal move of state do
            nextState = DoMove(state, move)
            moveValue = MiniMax(nextState, depth - 1, True)
            UndoMove(nextState, move)
            if moveValue < bestMoveValue then
                bestMoveValue := moveValue

    return bestMoveValue
```

Výpis 2.1. Pseudokód algoritmu MiniMax

2.4 NegaMax

NegaMax [6] je úprava algoritmu MiniMax pro hry s nulovým součtem. U her s nulovým součtem platí že $max(a, b) = -min(-a, -b)$, toho je právě využívá algoritmus NegaMax. Oproti MiniMaxu vyžaduje NegaMax, aby ohodnocení stavu hry bylo z pohledu hráče, který je v tomto stavu na tahu.

Algoritmus je zde uveden z důvodu, že následující algoritmus Alpha-Beta prořezávání a jeho rozšíření s transpoziční tabulkou je uveden v podobě NegaMaxu z důvodu zjednodušení pseudokódu.

```
function NegaMax(state, depth)
    if isTerminal(state) then
        return GameResult(state)
    if depth == 0 then
        return Evaluate(state)

    bestMoveValue := -inf
    for each legal move of state do
        nextState = DoMove(state, move)
        moveValue = -NegaMax(nextState, depth - 1)
        UndoMove(nextState, move)
        bestMoveValue := max(bestMoveValue, moveValue)
    return bestMoveValue
```

Výpis 2.2. Pseudokód algoritmu NegaMax

2.5 Alpha-Beta prořezávání

Alpha-beta prořezávání je algoritmus vylepšující algoritmus MiniMax. Vylepšení spočívá v tom, že neprohledává podstromy herního stromu, které by neměli vliv na nalezení optimálního tahu.

Algoritmus proto zavádí navíc dvě proměnné alpha a beta. Alpha reprezentuje nejlepší nalezenou hodnotu pro MAX úroveň a beta reprezentuje nejlepší nalezenou hodnotu pro MIN úroveň. Pokud při prohledávání nastane situace, kdy je alpha $\geq$ beta, tak se v aktuálně prohledávaném uzlu herního stromu prohledávání ukončí, protože dále prohledávané tahy by nepřispěli k nalezení lepšího tahu.

Na začátku se hodnota alpha inicializuje na $-\infty$ a hodnota beta se inicializuje na ∞.

Algoritmus nalezne stejné řešení jako algoritmus MiniMax. Časová složitost algoritmu je v nejhorším případě stejná jako algoritmu MiniMax.

Pokud jsou tahy ideálně uspořádané, tak je časová složitost $O(b^{\frac{d}{2}})$ kde b je větvící faktor odpovídající validním tahům a d je prohledávaná hloubka. Díky tomu je možné prohledat za stejný čas až 2x větší hloubku, než by zvládl prohledat algoritmus MiniMax.

Proto je dobré zavést také heuristickou funkci, která bude určovat pořadí prohledávání tahů, s cílem se nejvíce přiblížit ideálnímu uspořádání.

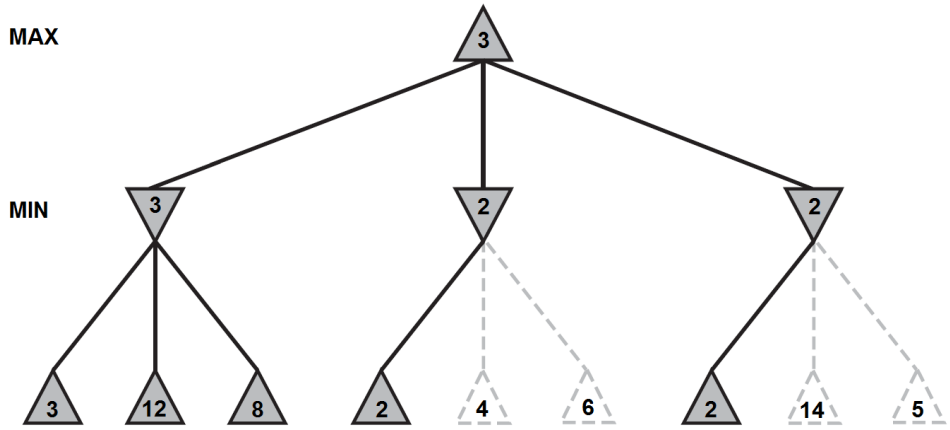
```
1  function AlphaBeta(state, depth, alpha, beta)
2      if isTerminal(state) then
3          return GameResult(state)
4      if depth == 0 then
5          return Evaluate(state)
6
7      bestMoveValue := -inf
8      for each legal move of state do
9          nextState := DoMove(state, move)
10         moveValue := -AlphaBeta(nextState, depth-1, -beta, -alpha)
11         UndoMove(nextState, move)
12         if moveValue > bestMoveValue then
13             bestMoveValue := moveValue
14         if moveValue > alpha then
15             alpha := moveValue
16         if alpha >= beta
17             break;
18     return bestMoveValue
```

Výpis 2.3. Pseudokód algoritmu Alpha-Beta prořezávání

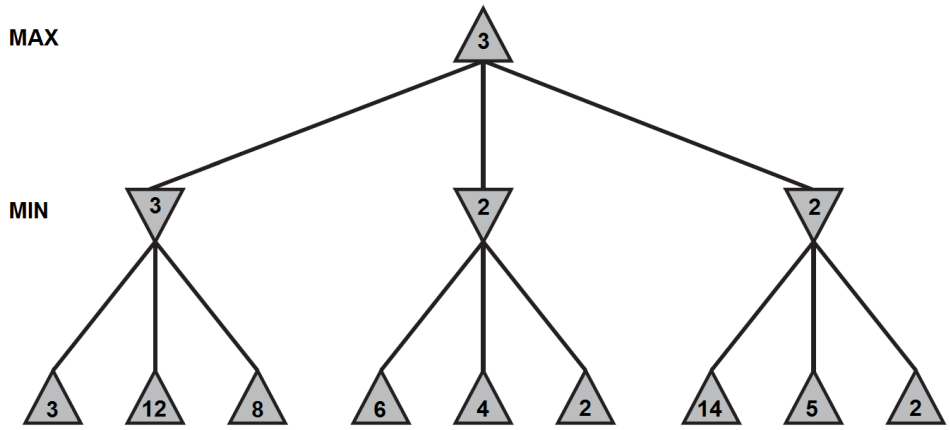
Při alpha-beta prořezávání je množství prohledaných uzlů velice závislé na jejich uspořádání. Což demonstrují následující obrázky. Obrázek 2.1 znázorňuje situaci, kdy jsou listy daného prohledávacího stromu ideálně uspořádané a je tak prohledáno nejméně uzlů. Naopak obrázek 2.2 znázorňuje situaci opačnou. Listy prohledávacího stromu jsou špatně uspořádané a není tak možné během prohledávání odříznou žádné větve a algoritmus musel prohledat úplně celý herní strom.

2.6 Transpoziční tabulka

Transpoziční tabulka [7] je tabulka, která mapuje hash šachové pozice na informace uložené do transpoziční tabulky při přechozím prohledáním pozice. Při prohledávání



Obrázek 2.1. Příklad ideálního uspořádání listů prohledávacího stromu. Čárkovaně znázorněné uzly stromu, jsou uzly, které během prohledávání nebyli navštíveny. Obrázek byl inspirován[4]



Obrázek 2.2. Příklad špatného uspořádání listů prohledávacího stromu, kde nedošlo k odříznutí žádných větvích stromu. Obrázek byl inspirován[4]

herního stromu jsou některé pozice navštíveny vícekrát různými posloupnostmi tahů. Proto se algoritmus během prohledávání vždy nejdřív podívá do transpoziční tabulky, jestli pro danou pozici neobsahuje informace. Tím dochází ke zrychlení prohledávání, protože stejné pozice nejsou prohledávané vícekrát a jsou použitá data získaná z transpoziční tabulky. Transpoziční tabulka je velmi užitečná při použití iterativního prohlubování v kombinaci s algoritmem Alpha-Beta prořezávání, kde je možné transpoziční tabulku použít pro lepší uspořádání tahů, čímž se zabývá kapitola 2.7.

Transpoziční tabulka se implementuje jako hash tabulka, takže získání informací z tabulky má konstantní časovou složitost.

V šachách se pro výpočet hashe šachové pozice používá Zobrist hashování[8], což je způsob jak transformovat šachovou pozici na 64 bitovou hodnotu.

2.7 Alpha-Beta prořezávání s transpoziční tabulkou

Alpha-Beta prořezávání s transpoziční tabulkou [9] je algoritmus Alpha-Beta rozšířený o transpoziční tabulku. Do transpoziční tabulky jsou po každý prohledávaný stav hry uloženy následujíc informace:

- Nejlepší nalezený tah
- Ohodnocení stavu do kterého do které vede uložený nejlepší tah
- Typ uložené hodnoty, který nabývá následujících hodnot:
  - LowerBound - Používá se pro inicializaci hodnoty alpha
  - UpperBound - Používá se pro inicializaci hodnoty beta
  - ExactValue - Používá se pro uložení hodnot v ostatních případech
- Hloubka ve které byl stav hry prohledán a uložen do transpoziční tabulky

Uložený nejlepší tah a ohodnocení stavu do kterého vede uložený nejlepší tah, je využíváno pro lepší uspořádání tahů, které by mělo přispět odříznutí více větví herního stromu. Uspořádání se provádí tak, že nejdříve je prohledávaný tah získaný z transpoziční tabulky a následně jsou prohledávány ostatní tahy bez změny jejich uspořádání.

```
function AlphaBeta(state, depth, alpha, beta)
    prevAlpha := alpha
    if isTerminal(state) then
        return GameResult(state)
    if depth == 0 then
        return Evaluate(state)
    Retrieve(state, ttMove, ttValue, ttType, ttDepth)
    if ttDepth >= depth then
        if ttType == ExactValue then
            return ttValue
        elseif ttType == LowerBound then
            if ttValue > alpha then
                alpha := ttValue
        elseif ttType == UpperBound then
            if ttValue < beta then
                beta := ttValue
        if alpha >= beta then
            return ttValue
    if ttDepth >= 0 then
        nextState := DoMove(state, ttMove)
        bestMoveValue := AlphaBeta(nextState, depth-1, alpha, beta)
        UndoMove(nextState, ttMove)
        bestMove := ttMove
        if bestMoveValue >= beta then
            goto DONE
    else
        bestMoveValue := -inf
    for each legal move of state do
        if move == ttMove then
            continue
        if bestMoveValue > alpha
            alpha := moveValue
        nextState := DoMove(state, move)
        moveValue := -AlphaBeta(nextState, depth-1, -beta, -alpha)
        UndoMove(nextState, move)
        if moveValue > bestMoveValue then
            bestMoveValue := moveValue
            bestMove := move
            if bestMoveValue >= beta then
                goto DONE
DONE:
    if bestMoveValue <= prevAlpha then
        ttType := UpperBound
    elseif bestMoveValue >= beta then
        ttType := LowerBound
    else
        ttType := ExacValue
    Store(state, bestMove, bestMoveValue, ttType, depth)
    return bestMoveValue
```

Výpis 2.4. Pseudokód algoritmu Alpha-Beta prořezávání s transpoziční tabulkou

2.8 Ohodnocovací(Heuristická) funkce

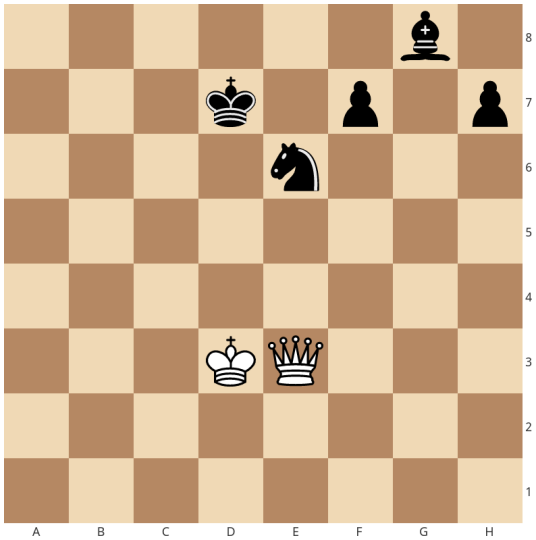
Ohodnocovací funkce [10] také známá jako heuristická funkce je funkce, která pro zadaný stav hry vrací přibližnou číselnou reprezentaci. Tato reprezentace udává jak moc je daný stav hry dobrý pro aktuálního hráče. Ohodnocovací funkce se používá v případech, kdy není možné nebo je velice výpočetně náročné zjistit skutečný výsledek hry. V takových situací ohodnocovací funkce vrací přibližnou hodnotu reprezentující jak dobrý je aktuální stav hry. Proto při navrhování ohodnocovací funkce je snaha, aby tato funkce vracela hodnoty, co nejblíže skutečné hodně. Protože čím je přesnější, tím lepší tahy budou nalezeny během prohledávání.

U dvou hráčových her s nulovým součtem se často používá funkce, která ohodnocuje stav hry z pohledu jednoho z hráčů (např. u šachů z pohledu bílého hráče), protože ohodnocení z pohledu druhého hráče, je opačnou hodnotou.

2.9 Efekt horizontu

Efekt horizontu [11] je docela závažný problém způsobeny tím, že je prohledávání ukončeno vždy ve stejné hloubce, ve které heuristická funkce provede ohodnocení. Tím se může stát, že ohodnocená pozice se může sice jevit jako velmi dobrá, avšak o jeden tah dále může protihráč sebrat cenou figurku jakou je např. dáma. A právě ukončením prohledávání před tímto tahem šachový program o tomto riziku nevěděl.

Obrázek 2.3 znázorňuje situaci, kdy je kůň bráněn dalším pěšcem. Pokud se prohledávání zastaví po dosažení hloubky 1, může se jako nejlepší tah jevit tah Qxe6+, neboli sebrat protihráči koně, protože soupeř přijde o koně. Avšak z důvodu ukončení prohledávání šachový program již nevidí, že kůň je bráněn pěšcem a hráč po zahrání tohoto tahu přijde o svoji nejcennější figurku, kterou je dáma.



Obrázek 2.3. Příklad šachové pozice, kde díky efektu horizontu bílý hráč považuje za nejlepší tah sebrat protihráči koně, ale neuvědomí si, že kůň je bráněn pěšcem.

Zvětšování hloubky prohledávání obecně vůbec neřeší tento problém, protože pokaždé, kdy ukončíme porovnávání nějaké větve v zádané hloubce a provedeme ohodnocení, nastane tento problém. Jelikož obecně není možné prohlédat všechny větve až do

konečných pozic. Používá se speciální druh prohledávání nazývaný Quiesence prohledávání. Místo aby se při dosažení maximální prohledávané hloubky provedlo ohodnocení pozice pomocí heuristické funkce, použije se právě tento speciální druh prohledávání.

Tím je zajištěno, že se ohodnocení jen klidné pozice.

2.10 Quiesence prohledávání

Quiesence prohledávání [11] je speciální druh prohledávání. Který prohledává pouze omezené množství tahů, které jsou považovány za nebezpečné, s cílem aby byli ohodnocované pouze pozice, které jsou považované za klidné. Protože ohodnocení pozic, které nejsou klidné, způsobuje efekt horizontu, který má zásadní vliv na nalezené nejlepší pozice. Čím se zabývá kapitola 2.9.

Na rozdíl od dalších uvedených algoritmů, není omezen hloubkou prohledávání, tudíž všechny listy prohledávaného stromu jsou buď koncové pozice, nebo klidné pozice. Teprve u těchto pozic se provádí ohodnocování.

Tento algoritmus může také využívat algoritmus Alpha-Beta prořezávání pro snížení počtu prohledaných pozic.

```
1  function Quiescence(state, alpha, beta)
2      value = Evaluate()
3      if value >= beta then
4          return beta
5      if alpha < value then
6          alpha := value
7      for each legal capture move of state do
8          nextState := DoMove(state, move)
9          moveValue := -Quiescence(nextState, -beta, -alpha)
10         UndoMove(nextState, move)
11         if score >= beta then
12             return beta
13         if score > alpha then
14             alpha = score
15     return alpha
```

Výpis 2.5. Pseudokód algoritmu Quiesence prohledávání

2.11 Iterativní prohlubování

Iterativní prohlubování[4] je algoritmus který na rozdíl od předešlých algoritmů sám neprovdání prohledávání ale snaží se řešit problém, že až na výjimky není možné dopředu odhadnout čas potřebný k prohledání herního stromu o zadané hloubce. Iterativní prohlubování obstarává postupné zvětšování prohledávané hloubky. Proto potřebuje využívat jiný algoritmu prohledávající do zadané hloubky. Tento algoritmus postupně volá se zvětšující se hloubkou, která začíná na hloubce 1 a zvětšuje se do nekonečna nebo vypršení času k dispozici pro prohledávání. Díky tomu je možné kdykoliv prohledávání ukončit a algoritmus poskytne nejlepší tah nalezený z hloubky, kterou stihnul celou prohledat.

Pokud by byl např. algoritmus Alpha-Beta prořezávání prohledávající určitou hloubku ukončen dřív, než by danou hloubku stihl celou prohledat, můžou nastat následující problémy:

- Algoritmus nestihl prohledat ani jeden z možných hráčových tahů do požadované hloubky. V takovém případě nemá k dispozici žádný tah, který by mohl zahrát. Jelikož už vypršel čas k dispozici pro prohledávání tak může zahrát např. náhodný tak, který může být však klidně nejhorší možný.
- Algoritmus prohledá několik validních hráčových tahů, ale nestihl prohledat všechny. Vlivem špatného uspořádání tahů, se algoritmus nemusel ještě dostat k optimálnímu tahu a nemusí tak vrátit optimální tah. Což je na rozdíl od předchozí situace o trochu lepší situace než zahrání náhodného tahu.

Tyto uvedené problémy je algoritmus iterativního prohlubování schopen zcela eliminovat.

Ačkoliv iterativní prohlubování stojí více výpočetního času než prohledání pouze konkrétně zadané hloubky, tak může být použit v kombinaci s transpoziční tabulkou a dříve prohledané hloubky mohou být použity pro uspořádání tahů, které může např. v případě algoritmu Alpha-Beta prořezávání výrazně zrychlit prohledávání, jak je uvedeno v kapitole 2.7 zabývající se právě algoritmem Alpha-Beta prořezávání s využitím transpoziční tabulky.

2.12 Monte Carlo tree serach

Monte Carlo tree search[12] dále uvedený jako MCTS je na rozdíl dříve uvedených algoritmů heuristický algoritmus. Herní strom buduje na základě statistiky navštívených uzlů a jejich zisku.

Velkého úspěchu dosáhl algoritmus MCTS ve hře Go[1], kde AlphaGo porazil mistra světa v této hře. Hra Go je považována za těžší hru, než jsou šachy, kvůli většímu větvícímu faktoru a většímu množstvý tahů v jednotlivých hrách.

Jedna iterace MCTS se skládá z následujících čtyř kroků

2.12.1 Selekce

Během selekce se prochází herní strom od kořene směrem k jeho listům, dokud se nenarazí na uzel, který není zcela expandovaný. Pokud není tento uzel koncovým stavem dojde k jeho expanzi.

Pro výběr následovníků během procházení se často používá metoda UCB1, kde je vybrán následovník $\tilde{v}$, který maximalizuje výraz

$$\frac{Q(\tilde{v})}{N(\tilde{v})} + c\sqrt{\frac{2lnN(v)}{N(\tilde{v})}}$$

Kde $N(v)$ odpovídá počtu návštěv daného uzlu v, $N(\tilde{v})$ odpovídá počtu návštěv uzlu $\tilde{v}$ který je potomek uzlu v, $Q(\tilde{v})$ odpovídá součtu zisku uzlu $\tilde{v}$, konstanta c ovlivňuje poměr mezi prozkoumáváním dalších následovníků a zaměřením se následovníky s vysokou cenou. Menší hodnota c způsobuje, že jsou více prohledávány následovníci s vysokým ziskem. Naopak větší hodnota c způsobuje, že jsou více prozkoumávány další následovníci .

2.12.2 Expanze

Během expanze dochází k přidání jednoho či více následovníků uzlu vybrané během kroku selekce. Postupným expandováním jednotlivých uzlů je vytvářen herní strom. Expandují se pouze uzly, které v herním stromu nemají přidané všechny následovníky.

2.12.3 Simulace

Během simulace jsou z vybraného uzlu hrány tahy všech hráčů, dokud není dosaženo koncového stavu. Pro dohrání hry dokonce může být použito rovnoměrné rozdělení pravděpodobnosti výběru tahů. Mnohdy je však lepší přizpůsobit výběr tahů řešenému problému a použít heuristickou funkci, která bude lepším tahům přiřazovat větší pravděpodobnost výběru.

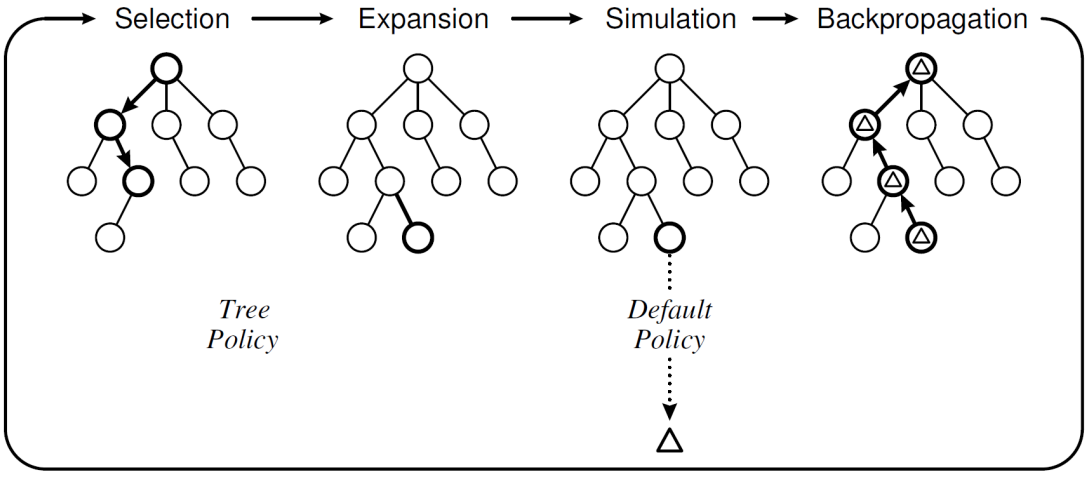
2.12.4 Zpětná propagace

Během zpětné propagace dochází k aktualizaci hodnot jednotlivých uzlů na základě výsledku simulace. Aktualizují se hodnoty všech uzlů, které jsou mezi kořenem a uzlem, ze kterého byla provedená simulace.

Hodnoty jednotlivých uzlů je potřeba aktualizovat z pohledu toho, jaký z hráčů v konkrétním uzlu byl na tahu. Pro dvouhráčové hry s nulový součtem je možné využít následující algoritmus popsaný pseudokódem:

```
1  function Backpropagation(node, value)
2      while node != null do
3          node.visitCount := node.visitCount + 1
4          node.value := node.value + value
5          value := -value
6          node := node.parent
```

Výpis 2.6. Pseudokód algoritmu zpětného šíření.



Obrázek 2.4. Obrázek znázorňující jednotlivé kroky jedné iterace algoritmu. Převzato z [12]

2.12.5 Výběr nejlepšího tahu

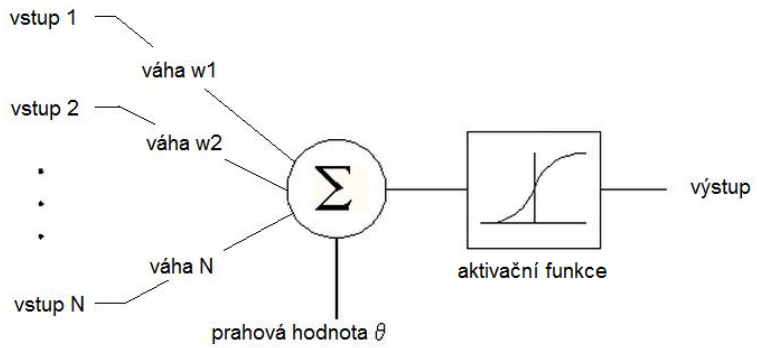
Existuje několik možností jak po ukončení běhu algoritmu MCTS vybrat nejlepší tah, který bude zahrán. Zde jsou uvedeny tři nejznámější metody výběru nejlepšího tahu.

- Výběr tahu s největší hodnotou
- Výběr tahů, který byl nejvíc krát navštíven
- Výběr tahu s největší hodnotou a který byl zároveň nejvíce krát navštíven.

Kapitola 3
Neuronové sítě

Neuronové sítě se inspirují principem fungování neuronů v lidském mozku. Skládají se z neuronů, které jsou uskupené do několika vrstev. Dopředné neuronové sítě jsou schopné řešit jak klasifikační, tak i regresivní úlohy.

Příklad neuronu je uveden na obrázku 3.1. Neuron se skládá z několika svých vstupů, ke kterým má příděly váhy. Všechny vstupy jsou vynásobeny svými vahami a sečteny. K této hodnotě se přičítá tzv. práh (anglicky bias). Na tento výsledek se použije aktivační funkce, která je zpravidla nelineární viz kapiola 3.3. Výsledek po aplikaci aktivační funkce je výstup jednoho neuronu.



Obrázek 3.1. Znázornění neuronu. Obrázek převzat z[13]

Toto je možné zapsat matematicky následovně:

$$z = \sigma(\sum_{i=1}^{N} w_i * x_i + b)$$

- Kde z je výstup neuronu
- Kde x_i je i-tá složka vstupního vektoru
- Kde w_i je váha i-té složky vstupního vektoru
- Kde b je tzv. práh (anglicky bias)
- Kde $\sigma(x)$ je libovolná aktivační funkce

Každá neuronová síť se skládá minimálně ze dvou vrstev, jimiž jsou vstupní vrstva a výstupní vrstva. Mezi těmito vrstvami může být několik dalších vrstev, které se nazývají skryté. Typicky neuronové sítě obsahují těchto skrytých vrstev několik, protože neuronové sítě složené pouze ze vstupní a výstupní vrstvy, dokáží řešit jen velmi omezené množství problémů.

3.1 Plně propojená vrstva

Plně propojená vrstva[14] je vrstva neuronových sítí, která obsahuje N vstupů a M neuronů, které jsou zároveň jejím výstupem. Do každého z těchto M neuronů vstupuje

všech N vstupů. Každý vstup do neuronu je násoben váhou, příslušného vstupu. M neuronů o N vstupech má dohromady $M * N$ proměnných, které bude potřeba naučit. Pokud je použit i posun, tak celkový počet proměnných je $M * N + M$

3.2 Softmax vrstva

Softmax[14] je vrstva používaná převážně v úlohách, kdy je neuronová síť použita ke klasifikaci do několika kategorií (např. rozpoznání číslice z obrázku). Výstup Softmax vrstvy transformuje její vstup do intervalu $\langle 0, 1\rangle$ Hodnoty na vstupu Softmax vrstva transformuje do intervalu $\langle 0, 1\rangle$ a zároveň součet všech transformovaných hodnot je roven 1. Tento výstup je možné také interpretovat jako pravděpodobnost správného zařazení do konkrétní kategorie. (např. při rozpoznání číslic, pravděpodobnost, že rozpoznání číslice je číslice 6)

Před touto vrstvou se již aktivační funkce nepoužívají.

Rovnice pro transformaci vstupu pomocí Softmax vrstvy:

$$f(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{k=0}^{N} e^{x_k}}$$

Kde x je vektor o k hodnotách, např. k kategorií.

3.3 Aktivační funkce

Aktivační funkce jsou z pravidla nelineární funkce. Pokud by byla použita lineární aktivační funkce, bylo by možné celou více vrstvou neuronovou sít, předělat do jednovrstvé neuronové sítě, která by fungovalo úplně stejně jako více vrstvá neuronová sít. Další podmínkou kterou aktivační funkce musí splňovat, je diferencovatelnost, aby bylo možné neuronové sítě učit pomocí metod založených na gradientním sestupu. V neuronových sítích se využívá mnoho různých aktivačních funkcí. V této podkapitolo jsou uvedeny aktivační funkce používané v této práci.

3.3.1 Sigmoida

Sigmoida [14] je definována následovně:

$$f(x) = \frac{1}{1 + e^x}$$

Sigmoida patří mezi často používané aktivační funkce.

Její nevýhoda je, že pro výstupní hodnoty blízké 0 nebo 1, je hodnota derivace velice malá a tím se učení zpomaluje.

3.3.2 ReLU

ReLU [15] je definována následovně:

$$f(x) = \begin{cases} x & \text{pro } x > 0, \\ 0 & \text{pro } x \leq 0 \end{cases}$$

ReLU je aktuálně velice často používá aktivační funkce. Převážně kvůli často rychlejšímu učení neuronové sítě, než u aktivační funkce Sigmoida.

Její nevýhoda je, že pro záporné hodnoty má nulovou derivaci. Kvůli nulové derivace není možné neuron učit. Takový neuron se mnohdy označuje jako „mrtvý “

3.3.3 Leaky ReLU

Leaky ReLU [16] je definována následovně:

$$f(x) = \begin{cases} x & \text{pro } x > 0, \\ 0.01x & \text{pro } x \leq 0 \end{cases}$$

Leaky ReLU řeší problém s nulovou derivací pro záporné hodnoty.

3.3.4 Softplus

Softplus [17] je definována následovně:

$$f(x) = ln(1 + e^x)$$

Jedná se o hladkou verzi aktivační funkce ReLU

3.4 Učení neuronových sítí

Pro učení dopředných neuronových sítí se používají algoritmy založené na gradientním sestupu. Nejznámějším algoritmem pro učení neuronových sítí je stochastický gradientní sestup. Algoritmus garantuje nalezení lokálního minima funkce, ale může k tomuto lokálnímu minimu konvergovat velice pomalu.

Pro zlepšení učení neuronových sítí, bylo vytvořeno několik dalších algoritmů, jako je např. Adam, u kterého se ukázalo, že k lokálnímu minimu konverguje rychleji, než ostatní algoritmy[18].

3.5 Autoenkóder

Autoenkóder[19] je druh neuronové sítě, která je učena bez učitele. To znamená, že data na vstupu nejsou nijak ohodnocená. Výstupem autoenkódéru je vektor o stejném počtu hodnot, jak jsou na vstupem. Během učení autoenkódéru je snaha aby byl např. po redukci dimenze vstupních dat, tyto data opět schopen zrekonstruovat.

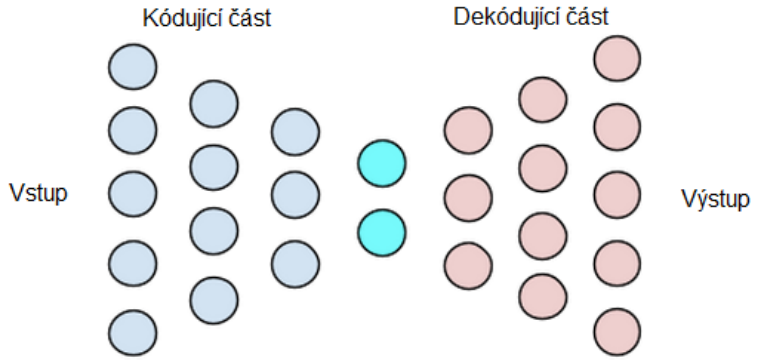
Autoenkóder se skládá ze dvou částí. První část je část kódující, která vektor vstupních dat transformuje na vektor např. o menším počtu hodnot např. z vektoru obsahujícího 773 hodnot, na vektor obsahující 600 hodnot.

Druhou částí autoenkódéru je část dekódující, která naopak výstup z kódující částí transformuje na vektor o stejném počtu hodnot jako na vstupu kódující části. Např. vektor obsahující 600 hodnot transformuje na vektor obsahující 773 hodnot.

Kódující a dekódující částí se mohou skládat z několika skrytých vrstev. Takovým auoenkódérum se říká hluboké. Příklad takového autoenkódéru můžete vidět na obrázku 3.2. Kde je znázorněn autoenkódér, který vstupní data o pěti hodnotách, redukuje na vektor o dvou hodnotách. Který se snaží zpětně transformovat na vektor se stejnými hodnotami, jako byl na vstupu kódující části.

Autoencoder má několik možných způsobů využití:

- Před trénování neuronové sítě. Předtím než je celá neuronová sít učena se naučí autoenkóder. Autoenkóder tak zastává funkci inicializace proměnných neuronové sítě. To může vést, k dosažení lepší přesnosti a zkrácení doby učení celé neuronové sítě.
- Redukce dimenze vstupních dat.
- Zvětšení dimenze vstupních dat.



Obrázek 3.2. Znázornění autoenkodéru. Obrázek převzat z[20]

Redukce a zvětšení dimenze vstupních dat, je závislá na datech, na kterých je autoenkódér naučení. Není např. možné použít autoenocder naučený na redukci dimenze šachový pozic pro redukci dimenze pozic ve hře dáma (pokud by pozice v obou hrách byli kódované do vektoru stejné délky).

V mnoha případech je využita pouze kódující část autoenkodéru a dekódující část autoenkódéru sloužila pouze pro naučení právě kódující části autoenkodéru.

3.5.1 Učení

Existuje několik způsobů jak autoencoder učit. Základní způsob učení spočívá v učení všech vrstev autoencoderu najednou. Tuto metodu je možné použít u autoencoderů s malým počtem vrstev v kódující a dekódující části. Pro hluboké autoencódéry je účinnější učit autoenkóder postupně. Například pokud máme hluboký autoencoder složený z vrstev 773-400-100-400-773, tak nejdříve naučíme autoencoder 773-400-773. Z takto naučeného autoencóderů následně použijeme kódující část 773-400 a učíme autoencoder 400-100-400, kde jako vstup, který má autoenkoder zrekonstruovat použijeme výstup částí 773-400. Následně všechny naučené vrstvy dáme dohromady a získáme naučených hluboký autoencoder.

Pro dosažení lepší schopnosti rekonstrukce vstupních dat auteonkóderem je možné takto před učený autoencoder následně učit jako celý. Tím, že se takto autoencoder před učí, zkonvertuje mnohdy rychleji k lokálnímu minimu, než kdybychom autoencoder učili bez před učení.

Kapitola 4
Učení neuronových sítí

Tato kapitola se zabývá naučením dvou neuronových sítí, které jsou využívány jako heuristiky v implementovaných algoritmech. Pro učení obou neuronových sítí je využíván algoritmus Adam [18] se kterým stejně jako je popsáno v článku [18] bylo dosahováno nejlepších výsledků.

4.1 Databáze her

Pro učení neuronových sítí byla zvolená databáze CCRL 40/40 [21]. Tato databáze byla vybrána z důvodu, protože obsahuje hry odehrané nejlepšími šachovými programy. A proto je pravděpodobné, že databáze bude obsahovat kvalitnější hry, než databáze obsahující hry odehrané hráči online hraním.

Existuje mnoho dalších databází her odehraných i lidmi různé úrovně. Problémem u databází obsahující hry odehrané hráči online je, že většina obsahuje hry z rychlých variant šachů, kde každý hráč má 10 a méně minut na celou hru. Důsledkem toho může být zahrání horších tahů. Které by negativně ovlivňovali přesnost, učených neuronových sítí.

4.1.1 Reprezentace šachového pole

Reprezentace šachových pozic byla převzatá z článku [3].

Cílem bylo, aby se neuronová síť naučili porovnávat šachové pozice, s co nejmenším počtem přidaných informacích. Také bylo cílem, aby neuronová síť mohla využít výhod pramenících z vlastnění více dam (i dalších figurek) získaných povýšením pěšce a pozic těchto figurek.

Například reprezentace použití v Giraffe[2] umožňuje zakódovat pozici pouze standartního počtu figurek. Pokud hráč vlastní například dvě dámy, neuronová síť má k dispozici pouze informaci, že hráč vlastní dvě dámy, ale zná pozici jen jedné z nich. Proto byla zvolená reprezentace, která se skládá z uspořádaného vektoru 768 hodnot. V této reprezentaci jsou od sebe oddělené figurky obou hráčů. Vektor se skládá z 12 částí obsahující 64 hodnot, které kódují umístění figurky určitého typu a barvy na šachovnici. Pokud na zvoleném políčku je figurka umístěna ve vektoru je na příslušné pozici hodnota 1, jinak 0.

Zakódované postavení figurek na šachovnici je doplněno o dalších 5 hodnot doplňují informace, které není možné vyčíst z pouhého postavení figurek na šachovnici. To jsou následující:

- Barva hráče který je na tahu. 0 pro bílého hráče, 1 pro černého hráče.
- Možnost bílého hráče provést krátkou rošádu
- Možnost bílého hráče provést dlouhou rošádu
- Možnost černého hráče provést krátkou rošádu
- Možnost černého hráče provést dlouhou rošádu

Tato reprezentace neumožňuje zakódovat, jestli hráč může provést braní mimochodem (en passant). Tento speciální tah se způsobem jakým jsou vybírány šachové pozice do datové sady, vyskytuje v datové sadě velmi zřídka.

Tato reprezentace také umožňuje efektivně pracovat i s nestandartními počty figurek jako je více než 32 figurek na šachovnici. Tomu se ale v této práci nevěnujeme.

Další výhoda této reprezentace spočívá v tom, že pokud uvažujeme šachové partie se standartním počtem figurek, tak v nejhorším případě je pouze 37 hodnot nenulových (32 figurek + 5 hodnot doplňující stav hry) a zbylých 736 hodnot je nulových. Tohoto je možné využít pro markantním zrychlení násobení matic při vypočítávání výstupu neuronové sítě [3].

4.2 Neuronová síť na porovnávání pozic

Na základě článku [3] je cílem naučit neuronovou sít porovnávat dvojici šachových pozic mezi sebou.

4.2.1 Příprava datové sady

Pro učení této neuronové sítě byla použita již zmíněná databáze CCRL 40/40[21]. Databáze obsahuje dohromady 700132 šachových partií. Z nichž 241332 skončilo výhrou bílého hráče a 179363 skončilo výhrou černého hráče. Zbytek partií skončil remízou a nebyl pro učení této sítě použit. Protože rozšíření datové sady o partie které skončili remízou nepřispívá podle článku [3] k dosažení lepším výsledkům.

Z těchto partií byli vybrány všechny pozice, které nepatřili mezi prvních 5 tahů, protože prvních několik zahajovacích tahů je obsaženo v mnoha hrách s různými výsledky. Navíc většina šachových programů odehraje prvních několik tahů podle knih zahájení, bez dalšího prohledávání. Vybrány byli také pouze pozice, ve kterých nebyl zahrán tah, který by bral figurku, protože podle článku [3] tahy, při kterých je sebrána figurka vedou jen ke krátkodobému zisku, protože často protihráč na tento tah zareaguje opět sebráním figurky. Navíc jelikož integrována neuronová síť bude po integraci do prohledávacích algoritmů porovnávat pouze klidné tahy, tak s pozicemi, kde by mohla být sebraná figurka, vůbec nebude pracovat.

Takto bylo vybráno dohromady 23240004 šachových pozic z partií, kde vyhrál bílý hráč a 18085883 šachových pozic z partií, kde vyhrál černý hráč. Tyto pozice byli rozděleny na dvě sady. První sada bude použita pro učení neuronové sítě a obsahuje 90 % šachových pozic ve kterých vyhrál bílý hráč, i černý hráč. Zbylých 10 % pozic bude použito jako testovací sada, která bude sloužit k odhadu přesnosti naučené neuronové sítě. Tato sada nebude použita pro naučení neuronové sítě.

4.2.2 Architektura

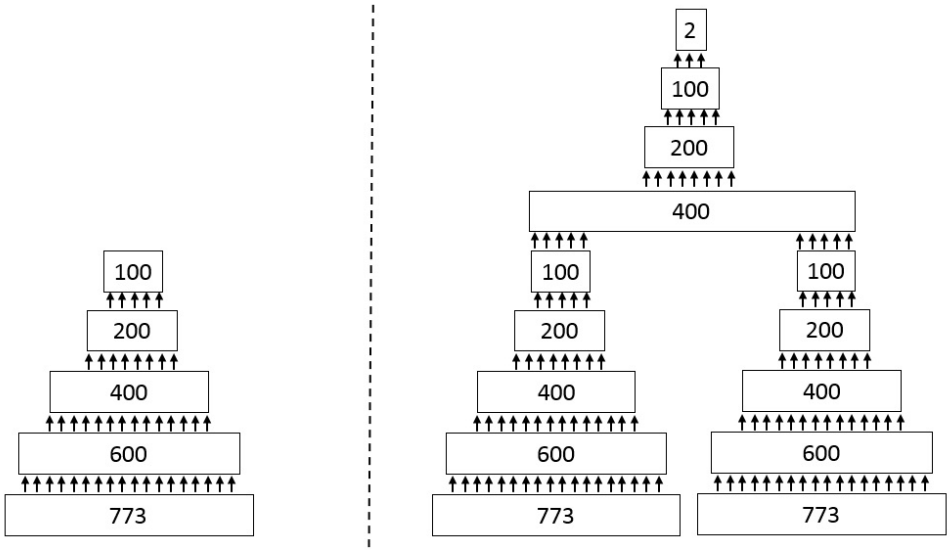
Architektura neuronové sítě byla převzata z článku [3]. Skládající se ze dvou částí. První částí je autoenkodér, který redukuje vstupní vektor o 773 hodnotách na vektor o pouhých 100 hodnotách. Je složen ze vstupní vrstvy, třech skrytých vrstev a výstupní vrstvy. Které popořadě mají následující počty neuronů: 773-600-400-200-100.

Do celé neuronové sítě je tento autoenkodér použit dvakrát, pro každou z porovnávaných pozic. Mezi těmito dvěma autoenkodéry jsou všechny proměnné sdílené. Tudíž po naučení neuronové sítě nezávisí výstup z autoenkodéru, na tom jestli pozice byla v pravé nebo levé části neuronové sítě.

Druhá část neuronové sítě je část, která obstarává porovnávání výstupů z autoenkodérů mezi sebou a rozhoduje, která z pozic je lepší. Tato část se skládá se vstupní

vrstvy, třech skrytých vrstev a výstupní vrstvy. Které popořadě mají následující počty neuronů: 200-400-200-100-2. Výstup z neuronové sítě udává, která z porovnávaných pozic je lepší. Pokud je lepší pozice v pravé části neuronové sítě, bude první hodnota neuronu na výstupu větší než hodnota druhého neuronu na výstupu. Pokud bude lepší pozice v levé části, bude to naopak.

Architekturu obou částí neuronové sítě, znázorňuje obrázek 4.1



Obrázek 4.1. Architektura neuronové sítě porovnávající dvě šachové pozice. Obrázek převzat z článku[3]

Všechny vrstvy, kromě výstupní vrstvy obsahující pouze dva neurony používají jako aktivační funkci, funkci ReLU. Poslední výstupní vrstva je pouze transformovaná pomocí vrstvy Softmax.

4.2.3 Učení autoenkodéru

Učení autoenkoédu kódující a dekódující šachové pozice reprezentované jako vektor 773 spadá do kategorie učení bez učitele. Neuronová síť nemá k dispozici, žádné ohodnocení vstupních šachových pozic. K dispozici má pouze šachové pozice, bez dalších informací.

Autoenkodér se tak učí nalézt, jak reprezentaci šachové pozice tvořenou vektorem 773 hodnotami zmenšit na vektor o 100 hodnotách. Jak bylo popsáno v kapitole 3.5, tato metoda je následně použita jako inicializace proměnných celé neuronové sítě, který podle článku[22] urychluje a zvyšuje přesnost naučené neuronové sítě oproti náhodné inicializaci proměnných neuronové sítě.

Pro učení autoenkodéru bylo použit jak pozic, kde vyhrál bílý hráč, tak pozic kde vyhrál hráč černý. Dohromady to je 41325887 šachových pozic. Z nichž bylo použito 90 % použito pro učení autoenkódéru a 10 % pro odhad jeho chyby.

Na základě článku[3] byla zvolena pro kódující část funkce ReLU jako aktivační funkce. Pro část dekódující byly zvoleny následující funkce jako aktivační funkce:

- Lineární
- ReLU
- Sigmoida
- Softplus

S těmito funkcemi byli naučené jednotlivé autoenkodéry, jejich přehled je uveden v tabulce 4.1 včetně použitých parametrů a dosažené střední kvadratické chyby. Učení probíhalo jak je popsáno v kapitole 3.5, po jednotlivých částech autoenkodéru.

Všechny autoencodéry byli v první fázi učeny po dobu 20 epoch s velikostí dávky 1000 a učícím algoritmem Adam[18] s parametrem 0,0002.

Aktivační funkce dekódovací části	Střední kvadratická chyba
Lineární	0,00977
ReLU	0,01111
Sigmoida	0,02607
Softplus	0,01818

Tabulka 4.1. Dosažené střední kvadratické chyby autoenkodérů po první fázích učení.

Po naučení autoenkodérů byli jednotlivé části autoencodérů poskládané do celého hlubokého autoenkodérů (773-600-400-200-100-200-400-600-773), který byl po několik epoch učet jako celek. Tím byla dosažená ještě o něco menší střední kvadratická chyba. Dosažené výsledky jsou uvedeny v tabulce 4.2 včetně použitých parametrů pro učení.

Aktivační funkce dekódovací části	Střední kvadratická chyba	Epoch	Adam
Lineární	0,00781	20	0.00001
ReLU	0,00407	20	0.00001
Sigmoida	0,00036	100	0.0001
Softplus	0,00329	20	0.00001

Tabulka 4.2. Dosažené střední kvadratické chyby autoenkodérů po pokračování učení celých autoenkodérů.

Bylo také experimentováno s regulačními metodami, kde v případě použití Dropoutu[23], byla schopnost autoenkodéru zrekonstruovat vstup, výrazně horší.

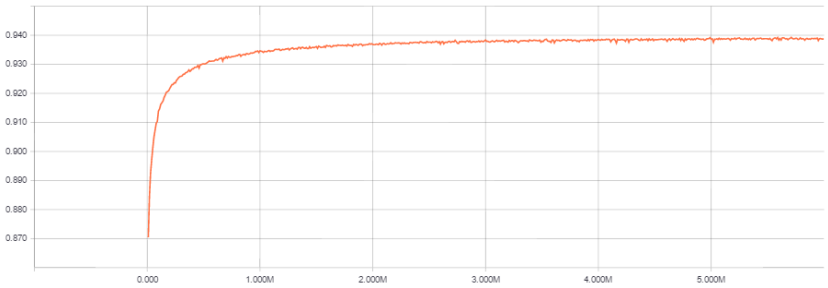
4.2.4 Učení celé sítě

Potom, co bylo naučeno několik autoenkodérů s různými aktivačními funkcemi v dekódující části, byl vybrán, ten se kterým se neuronová sít nejrychleji učila.

Jednalo se o autoenkodér s lineární aktivační funkcí v dekódující části. S tímto autoenkodérem bylo dokončeno celé učení neuronové sítě.

Metoda učení neuronové sítě je založen na článku[3]. Na začátku jsou všechny pozice oddělené do dvou částí. Kde 90 % dat bylo použito pro učení neuronové sítě a zbylých 10 % bylo použito pro testování přesnosti neuronové sítě. Na začátku každé epochy byly šachové pozice kde vyhrál bílý hráč i kde vyhrál černý hráč zamíchané. Z nich byla vybrána podmnožina 10000000 dvojic pozic. Kde ve dvojici byla vždy pozice z množiny, kde bílý hráč vyhrál a druhá z množiny, kde černý hráč vyhrál. Jejich pořadí bylo v rámci dvojice také náhodně zamícháno.

Pro naučení neuronové sítě byl použit algoritmus Adam[24] s parametrem 0,0003, velikostí dávky 1000 a učení probíhalo po dobu 1000 epoch. Na konci každé epochy se provedlo určení přesnosti neuronové sítě na testovací množině a proměnné neuronové sítě byli uloženy. Po skončení učení byl vybrán uložený stav neuronové sítě takový, který dosahoval největší přesnosti na testovací množině.



Obrázek 4.2. Graf znázorňující vývoj přesnosti na testovací množině během učení neuronové sítě.

4.2.5 Dosažené výsledky

Největší přesnosti při porovnávání dvou šachových pozic, které se povedlo dosáhnout je 93,9 % což je značně méně, než bylo dosaženo článku[3], kde bylo dosaženo přesnosti 98,0 %

Při integraci neuronové sítě se ukázalo, že s menší dosaženou přesností při porovnání pozic, hrál šachový program velice špatně.

Zkusili jsme kontaktovat autory článku[3], ale autoři článku na zaslané e-maily neodpověděli.

Jelikož by vlivem špatných výsledků dosahovaných s naučenou neuronovou sítí bylo ovlivněno porovnání algoritmů Alpha-Beta prořezávání a MCTS, tak byla pro porovnání také použita neuronová sít z šachového programu Giraffe[2], tím by porovnání algoritmů mělo mít vyšší vypovídací hodnotu.

4.3 Neuronové sítě na predikci nejlepšího tahu

Rozhodli jsme se naučit také několik neuronových sítí, které by byli schopné s rozumnou přesností, predikovat následující tah. Tuto skupinu několika neuronových sítí by bylo následně možné využít v algoritmu Alpha-Beta prořezávání pro uspořádání tahů s cílem zvýšit prohledávanou hloubku. Další možnost použití by byla v rámci MCTS pro lepší než zcela náhodnou simulaci her až do koncových stavů.

Jelikož je potřeba aby tyto neuronové sítě bylo možné vyhodnotit velice rychle, tak cílem nebylo dosáhnout nejlepší přesnosti predikce tahu, ale najít co nejmenší neuronovou síť s přijatelnou přesností.

4.3.1 Příprava datové sady

Pro vytvoření datové sady pro učení této neuronové sítě byla opět použita databáze CCRL 40/40[21]. Z této databáze byli odstraněny šachové partie, ve kterých alespoň jeden z hráčů měl Elo nižší než 3000. Tímto bylo zajištěno, že pro učení neuronových sítí budou použity partie, odehrané nejlepšími šachovými programy. U kterých je předpokládáno, že jimi zahrané tahy budou jen ve velmi málo případech špatné. Z každé šachové partii byly do datové sady vybrány všechny pozice se zahranými tahy. Pozice a v nich zahrané tahy byli vybírány ze všech partií včetně her, které skončili remízou.

Tabulka 4.3 udává přehled o počtu pozic pro každý typ figurky, ze kterých bylo touto figurkou taženo.

Pro učení výše uvedených neuronových sítí bude vždy použito 90 % tahů a zbylých 10 % bude použito jako testovací množina pro odhad přesnosti naučených neuronových sítí.

Typ figurky	Celkový velikost datové sady
Pěšák	3578644
Kůň	2430612
Střelec	2752946
Věž	3829732
Dáma	2360936
Král	2878145

Tabulka 4.3. Velikosti datových sad pro taky každého typu figurky.

Pro učení predikce typu figurky bylo použito všech pozic s tahy všech typů figurek. Tudíž celková velikost datové sady je 17831015 pozic. Z nichž opět bylo pro učení použito 90 % a zbylých 10 % bylo použito pro testování přesnosti.

4.3.2 Architektury

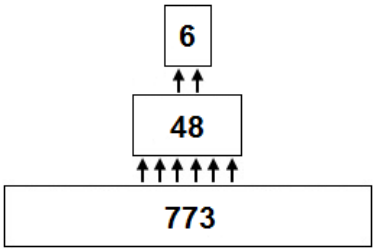
V šachách na rozdíl např. od hry Go je zapotřebí zakódovat nejen políčko na které byla přesunuta figurka, tak i políčko odkud byla tato figurka přesunuta, protože je možné na stejné políčko přesunout několik figurek a nebylo by možné rozlišit, která z figurek tam opravdu má být přesunuta.

Pokud bychom chtěli přesně zakódovat přesunutí figurky, bylo by dohromady potřeba 4096 hodnot. Navíc v případě povýšení pěšce, je také zapotřebí zakódovat na jaký typ figurky byl pěšec povýšen, což vyžaduje ještě další hodnoty.

Naproti tomu ve hře Go pro reprezentaci tahu stačí pouze 361 hodnot.

Jak již bylo zmíněno, cílem je najít několik malých neuronových sítích, které bude možné velice rychle vyhodnocovat. Na základě toho byla vytvořena skupina 7-mi neuronových sítí. Kde jedna neuronová síť bude predikovat typ figurky, kterou bude taženo a zbylých 6 neuronových sítí se bude zabývat tím, na jaké políčko bude určitá figurka přesunuta.

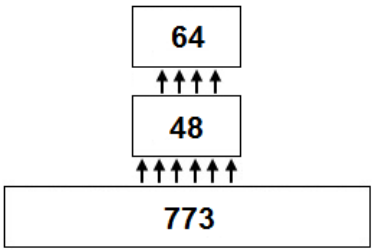
Obrázek 4.3 znázorňuje neuronovou síť na predikci, typu figurky, kterou bude v zadané šachové pozice taženo. Byla zvolená neuronová síť, která má na vstupu stejný vektor 773 hodnot jako neuronová síť, na porovnávání pozic. Obsahuje jednu skrytou vrstvu, ve které je 48 neuronů a ve výstupní vrstvě má 6 neuronů, kde každý reprezentuje jeden z typů figurek.



Obrázek 4.3. Architektura neuronové sítě pro predikci typu figurky, kterou bude taženo.

Obrázek 4.4 znázorňuje neuronovou síť na predikci, políčka na šachovnici, na které bude figurka konkrétního typu přesunuta. Opět byli zvolené neuronové sítě, které mají na vstupu vektor 773 hodnot jako neuronová síť, na porovnávání pozic. Obsahuje jednu skrytou vrstvu, ve které je 48 neuronů a ve výstupní vrstvě má 64 neuronů, kde každý reprezentuje jedno konkrétní políčko na šachovnici.

Pro obě neuronové sítě byla použita aktivační funkce Leaky ReLU. Na výstup obou neurononových sítí je aplikovaná vrstva Softmax.



Obrázek 4.4. Architektura neuronové sítě pro predikci pole na které bude taženo konkrétním typem figurky.

4.3.3 Učení neuronových sítí

Vzhledem k dříve zmiňovanému malému počtu proměnných v obou typech neuronových sítích, nebyli během učení použité žádné metody regularizace. Protože u takto malých neuronových sítí s malým počtem proměnných a naopak velkým množství šachových pozic, je velice nepravděpodobné, že by se neuronové sítě přeučili. Což také potvrzují dosažené výsledky.

Pro naučení neuronové sítě na predikci typu figurky, byl použít učící algoritmus Adam[18] s parametrem 0,0003 a byla učena po dobu 1000 epoch s velikostí dávky 1000.

Pro naučení neuronové sítě na predikci pole na které bude konkrétní figurka přesunuta, byl použít učící algoritmus Adam[18] s parametrem 0,0003 a byla učena po dobu 1000 epoch s velikostí dávky 1000.

Po ukončení učení byla u všech neuronových sítí vybraná vždy ta, které měla nejmenší chybu na testovací množině.

4.3.4 Dosažené výsledky

Během učení bylo po každé uplynuté epoše provedeno vyhodnocení na testovací sadě a také provedeno uložení naučené neuronové sítě. Po dokončení učení byla vybraná a dále používaná neuronová síť taková, která dosahovala největší přesnosti při predikci typu figurky nebo pozice figurky na testovací datové sadě. Pro všechny naučené neuronové sítě jsou dosažené přesnosti uvedeny v tabulce 4.4 a tabulce 4.5 . Z tabulky 4.4 je pozorovatelný jev, že na čím méně polí může být daným typem figurek taženo, tím je větší přesnost predikce pole na které bude taženo.

V tabulkách jsou uvedeny hodnoty Top-1, Top-3 a Top-5. Kde je možné obecně pojmou jako Top-k, což znamená, že skutečně zahraný pro danou šachovou pozici byl mezi prvními k tahy s největší pravděpodobností.

Např. Top-3 88,30 % pro figurku krále znamená, že skutečný tah, byl v 88,30 % případech mezi třemi tahy s největší pravděpodobností.

Typ	Top-1	Top-3	Top-5
Pěšec	49,75	81,11	92,04
Kůň	56,58	84,47	93,45
Střelec	41,58	71,18	84,87
Věž	27,34	51,89	66,20
Dáma	29,14	54,87	69,30
Král	55,37	88,30	97,34

Tabulka 4.4. Přesnosti predikce pole na které bude figurkou táhnuto, pro všechny figurky.

Top-1	Top-3	Top-5
46,63	87,26	99,26

Tabulka 4.5. Přesnost predikce typu figurky, kterou bude táhnuto.

Pro tabulku 4.5 Top-k znamená, že predikovaný typ figurky byl mezi k figurkami s největší pravděpodobností.

Kapitola 5
Integrace neuronových sítí

Pro implementaci enginu byl zvolen programovací jazyk C++. Jednotlivé vrstvy neuronových sítí byly implementovány s pomocí knihovny pro lineární algebru Eigen[25].

Při implementaci byli také použity části zdrojových kódům z šachového enginu Giraffe, jejichž autorem je Matthew Lai[2]. Konkrétně se jedná o tyto části:

- Generování validních tahů včetně operacích se šachovým polem
- Převod herního pole do reprezentace využívanou neuronovou sítí v enginu Giraffe
- Výpočet výstupu neuronové sítě použité v enginu Giraffe

Všechny převzaté zdrojové kódy z enginu Giraffe jsou oddělené od vlastního zdrojového kódu, umístěním do složky giraffe.

Několik následujících souborů byli převzato a upraveno pro použití v této práci:

- board.h
- board.cpp

Ve snaze využít maximální výkon použitých CPU (Intel) jsou zdrojové kódy slinkované s proprietární knihovnou Intel® Math Kernel Library[26], umožňující zrychlit používané maticové a vektorové operace využívané pro výpočty výstupu neuronových sítí.

5.1 Komunikace

Pro komunikaci mezi uživatelem nebo jiným šachovým programem je využíván bezstavový protol Universal Chess Interface[27].

5.2 Implementace Quiescence prohledávání

Při implementaci Quiescence prohledávání byli do prohledávání zařízeny následující tahy:

- Sebrání libovolné figurky
- Povýšení pěšce na královnu

5.3 Pravděpodobnostní hráč

Tento hráč na rozdíl od ostatních nevyužívá prohledávání k nalezení nejlépe ohodnoceného tahu, ale vždy zahraje tah s největší pravděpodobností. Pro výpočet pravděpodobnosti bylo naučeno 7 neuronových sítí, které tento hráč využívá. Viz kapitola 4.3.

Tyto neuronové sítě je možné využít těmito způsoby:

- Výpočet úplné pravděpodobnosti
- Výběr figurky s největší pravděpodobností pro kterou je vybrán tah s největší pravděpodobností

5.3.1 Výpočet úplné pravděpodobnosti

Engine vezme aktuální stav hry a převede si ho do potřebné reprezentace pro neuronové sítě. Pomocí neuronové sítě predikující typ tažené figurky, je vypočtena apriorní pravděpodobnost $P(F)$. Následně se odstraní typy figurek, které v dané pozici nemohou táhnout. Pro zbylé figurky se zajistí aby $\sum_{f=0}^{N} P(f) = 1$. Následně pomocí každé z 6-ti neuronových sítí je vypočtena podmíněná pravděpodobnost tažení na konkrétní pozice danou figurkou $P(U|F)$. Opět jsou odstraněny nevalidní tahy a zajištěno aby $\sum_{u=0}^{N} P(u|F) = 1$. Následně je zahrán tah s maximální sdruženou pravděpodobností $P(U, F)$

5.3.2 Výběr figurky s největší pravděpodobností pro kterou je vybrán tah s největší pravděpodobností

Engine vezme aktuální stav hry a převede si ho do potřebné reprezentace pro neuronové sítě. Pomocí neuronové sítě predikující typ tažené figurky, je vypočtena apriorní pravděpodobnost P(F). Následně je vybrána figurka s největší pravděpodobností, která v dané pozici může provést validní tah. Pomocí neuronové sítě je vypočtená podmíněná pravděpodobnost tažené na konkrétní pozice pro dříve vybranou figurku. Na základě toho je vybraný validní tah dříve vybranou figurkou na políčko s největší pravděpodobností.

5.3.3 Porovnání

Experimentálně byli porovnané tyto dvě možnsoti využití, pomocí sady na testování strategie. Bylo ukázáno, že výpočet úplné pravděpodobnosti a následné zahrání figurky s největší sdruženou pravděpodobností dosahuje značně lepších výsledků. Srovnání je uvedeno v tabulce 5.1.

Tah	Celkový počet bodů
Úplná pravděpodobnost	3267
Figurka s největší pravděpodobností	2674

Tabulka 5.1. Srovnání možností využití neuronových sítí s pravděpodobností.

Obě možnosti využití 7-mi neuronových sítí se potýkají s dvěma velmi podobnými problémy:

- Zvolená reprezentace neumožňuje odlišit povýšení pěšce na jiné figurky. Při implementaci proto bylo rozhodnuto, že pěšák bude vždy povýšen na dámu, jelikož dáma patří mezi nejsilnější figurku ve hře.
- Zvolená reprezentace také neumožňuje v situacích kde dvě figurky stejné barvy a stejného typu mohou provést tah na stajné políčko, která z figurek má tah provést. Tato situace nastává velmi zřídka a může se týkat všech Při implementaci je tohle vyřešeno, tím, že se vybere tah figurky, která je dříve vložena do seznamu validních tahů. Příklad této situace je znázorněn na obrázku 5.1

5.4 Alpha-Beta prořezávání

Do algoritmu Alpha-Beta prořezávání je možné integrovat neuronové sítě několika způsoby.

První možnost je integrovat neuronovou sít, která ohodnocuje pozice v tomto případě neuronovou síť převzatou z Giraffe [2]. Tuto síť následně používat pro ohodnocování klidných pozic. Výhodou této metody je, že pro každou pozici stačí vypočítat

Obrázek 5.1. Příklad šachové pozice, kde bílý hráč může oběmi koni táhnout na políčko D5. Zvolená reprezentace výstupů ze 7-mi neuronových sítí v této situaci neodovolí kterým z konů se má táhnout.

její ohodnocení jen jednou a dále už algoritmus Alpha-Beta prořezávání pracuje jen s hodnotou.

Další možností je integrovat neuronovou síť, která porovnává šachové pozice, tudíž se šachové pozice neohodnocují, ale jen porovnávají mez sebou. Tuto metodu představili v článku [3]. V algoritmu Alpha-Beta prořezávání jsou tak všechny místa, kde dochází k porovnávání hodnot, nahrazení porovnávání šachových pozic.

Naučená neuronová síť porovnává šachové pozice z pohledu bílého hráče, proto je potřeba při porovnávání z pohledu černého hráče, výstupy z neuronové sítě prohodit.

Tato metoda může přinášet nevýhodu v tom, že úplně každé porovnávání dvou pozic je potřeba provádět, skrz neuronovou síť. Což způsobuje zpomalení.

5.5 Vylepšení pomocí uspořádání tahů

Jak bylo zmíněno v kapitole 2.5 uspořádání tahů má velký vliv na množství odříznutých šachových pozic a může tak zvýšit hloubku, kterou se podaří během prohledávání prohledat. Proto je vhodné se uspořádáváním tahů zabývat.

Pro uspořádávání tahů je možné využít neuronovou síť na porovnávání šachových pozic. Toto použití má hlavní nevýhodu spočívající v tom, že porovnání pozic pomocí neuronové sítě je velice časově náročné a pro úplně seřazení i nelezení nejlepšího tahů je těchto porovnání potřeba mnoho. Kvůli velkému zpomalení uspořádáváním pozic, by výhoda z dobrého uspořádání pozic a možnost tak odříznou více šachových pozic během prohledávání, byla potlačena časovou náročností uspořádávání.

Lepší variantou pro uspořádání tahů je využít neuronové sítě, které přiřazují možným tahům pravděpodobnost. Na rozdíl od neuronové sítě na porovnávání pozic je vyhodnocování těchto neuronových sítí velice rychlé. Ačkoliv je nutno pro každý tah vyhodnotit 7 neuronových sítí, tak jelikož tyto neuronové sítě jsou malé, je jejich vyhodnocování velice rychlé.

5.6 Monte Carlo stromové prohledávání

Při integrování neuronových sítí do algoritmu MCTS je možné neuronové sítě použít jednak pro ohodnocování pozic, tak pro simulování her až do konce.

5.6.1 Simulace her

Pro simulování her až do koncového stavu je možné využít zcela náhodného rozhodování, kde všechny tahy v dané pozici mají stejnou pravděpodobnost, že budou zahrány. Tato metoda se ukázala jako poměrně špatná, protože při simulaci vznikaly hry, které se skládaly z více než 400 tahů. Takovéto hry jsou v šachách spíše ojedinělé. Simulování takto dlouhých partií je také časově náročnější a tudíž není možné provést takové množství iterací algoritmu.

O něco lepší možností pro simulování her až do koncového stavu je použít neuronové sítě pro přiřazení pravděpodobnosti, že konkrétní tah bude zahrán. Následně podle přiřazené pravděpodobnosti náhodně vybrat tah, který bude zahrán. Tato metoda je výrazně lepší než zcela náhodné vybírání tahů, ale vzhledem k tomu, že pro od simulování her až do konce je potřeba mnohokrát použít neuronové sítě pro přidělení pravděpodobnosti tahů v každé šachové pozici byli dosažené výsledky z důvodu malého množství iterací algoritmu MCTS horší.

5.6.2 Ohodnocování pozic

Další možností využití neuronových sítí v algoritmu MCTS je pro nahrazení simulovaní her až do koncového stavu, za použití neuronové sítě pro ohodnocení pozice. Tato metoda může mít nevýhodu v tom, že jsou ohodnocovány i pozice, které nejsou označené jako klidné, tudíž může docházet k efektu horizontu.

S cílem eliminovat možný efekt horizontu byla navrhnuta metoda umožňující ohodnocovat pouze klidné pozice. Místo ohodnocení šachové pozice, se provede simulace hry, při které jsou voleny pouze takové tahy, aby vedli do klidné pozice. Klidná pozice je následně ohodnocena neuronovou sítí.

5.7 Implementované enginy

Na základě výše popsaných možností integrace neuronových sítí do algoritmů Alpha-Beta prořezávání a MCTS bylo implementováno několik verzí obou algoritmů využívající neuronové sítě. Které jsou následně porovnané mezi sebou i s existujícími enginy v kapitole 6.

5.7.1 Alpha-Beta prořezávání s porovnáním pozic

Pro implementaci algoritmu Alpha-Beta prořezávání využívajícího neuronovou síť pro porovnávání šachových pozic, byla použita naučená neuronová síť viz kapitola 4.2. Implementovaný algoritmus využívá Quiesence prohledávání aby bylo zajištěno, že jsou ohodnocovány pouze klidné pozice.

Byli implementovány dohromady čtyři varianty algoritmu

- Základní algoritmus Alpha-Beta prořezávání
- Alpha-Beta prořezávání rozšířený o transpoziční tabulku
- Alpha-Beta prořezávání s řazením tahů při prohledáváním
- Alpha-Beta prořezávání rozšířený o transpoziční tabulku a s řazením tahů při prohledáváním

Při implementování verze Alpha-Beta prořezávání vyžívající transpoziční tabulku bylo oproti pseudokódu 2.4 odstraněno používání záznamu typu ExactValue, protože u naučené neuronové sítě nelze říct, že pokud platí $a \leq b$ a zároveň $a \geq b$ tak že by platilo $a = b$, z tohoto důvodu by ponechání záznamu typu ExactValue způsobovalo problémy.

Pro uspořádání pozic bylo využito 7-mi neuronových sítí, které přiřadí každému tahu pravděpodobnost, že bude v dané pozici zahrán. Tahy jsou následné prohledávány v pořadí od největší pravděpodobnosti po nejmenší.

5.7.2 Alpha-Beta prořezávání s ohodnocování pozic

Pro implementaci algoritmu Alpha-Beta prořezávání využívajícího neuronovou sít pro ohodnocování pozic byla použita neuronová sít s enginu Giraffe [2]. Implementovaný algoritmus využívá Quiesence prohledávání aby bylo zajištěno, že jsou ohodnocovány pouze klidné pozice.

Stejně jako v předchozí podkapitole 5.7.1 byli implementovány čtyři varianty algoritmu Alpha-Beta prořezávání.

Přidání transpoziční tabulky bylo na rozdíl od sítě na porovnávání šachových pozic implementováno přesně podle pseudokódu 2.4.

Pro uspořádání pozic bylo stejně jako v přechozí podkapitole 5.7.1 využito 7-mi neuronových sítí.

5.7.3 MCTS

První varianta místo simulování her až do koncového stavu provádí přímo ohodnocení pozice pomocí neuronové sítě převzaté z enginu Giraffe [2].

Další varianta ohodnocuje pouze klidné pozice. Tudíž místo simulace hry až do koncového stavu jsou během simulace voleny náhodně tahy tak, aby vedli do klidné pozice, která je následně stejně jako v předchozí variantě, ohodnocena pomocí neuronové sítě. Během simulace vedoucí do klidné pozice, jsou tahy vybírány náhodně na základě přidělené pravděpodobnosti jejich zahrání, vypočtené pomocí 7-mi neuronových sítí přiřazující pravděpodobnost zahrání jednotlivým tahům.

Kapitola 6
Experimentální porovnání výsledků

Tato kapitola se zabývá porovnáním jednotlivých implantovaných algoritmů využívajících neuronové sítě pro hraní šachů. Jejich porovnání je provedeno dvěma způsoby.

- Porovnání šachových programů za pomocí sady na testování strategie
- Porovnání šachových programů mezi sebou i s existujícími šachovými programy při odehrání 100 šachových partií. Kde vždy půlku her šachový program odehrál za bílého hráče a druhou půlku za černého hráče.

6.1 Sada na testování strategie

Sada na testování strategie[28] je sada obsahující dohromady 1500 šachových pozic, jejíž autoři jsou Dann Corbit a Swaminathan Natarajan. Sada je rozdělená do 15-ti částí, kde každá část se skládá ze 100 šachových pozic s několika bodově ohodnocenými tahy. Každá z těchto částí se zaměřuje na testování jedné konkrétní strategie. Názvy jednotlivých částí sady:

- Undermining
- Open Files and Diagonals
- Knight Outposts
- Square Vacancy
- Bishop vs Knight
- Re-Capturing
- Offer of Simplification
- Advancement of f/g/h pawns
- Advancement of a/b/c Pawns
- Simplification
- Activity of the King
- Center Control
- Pawn Play in the Center
- Queens and Rooks to the 7th Rank
- Avoid Pointless Exchange

Pro každou šachovou pozice z této sady je vždy uveden nejlepší tah, případně několik dalších tahů. Šachový engine za nalezení nejlepšího tahů získá 10 bodů. Pokud nenajde nejlepší tah, může za takový tah získat 0-9 bodů. Dohromady může engine získat až 15000 bodů, kde množství získaných bodů souvisí s jeho silou a Elo.

Sada na testování strategie přináší oproti odehrání výhodu v tom že, není nutné, aby enginy hráli každý proti každému, stačí ohodnotit každý enginy zvlášť a porovnat je podle jejich dosaženého score.

Nevýhodou této metody může být, nutnost všechny enginy otestovat na stejném hardwaru. Tato nevýhoda při hraní enginů mezi sebou není, protože u té je nutné

Obrázek 6.1. Příklad šachové pozice z části sady zaměřené na změnu hodnoty střelce a koně v závislosti na šachové pozici.

zajistit stejné podmínky, pouze pro oba hráče při dané partii. Tudíž všechny partie nemusí být nutně odehrané na stejném hardwaru.

Na obrázku 6.1 je vybraná šachová pozice z části sady zaměřené na změnu hodnoty střelce a koně v závislosti na šachové pozici. V tabulce 6.1 je pro tuto šachovou pozici uvedeno bodové ohodnocení několika tahů. Tahy neuvedené v této tabulce mají ohodnocení 0 bodů.

Tah	Body
Bxd7	10
Ne3	3
Rcd1	3
Be2	2

Tabulka 6.1. Bodově ohodnocené tahy pro šachovou pozici z obrázku 6.1.

6.1.1 Způsob testování

Testování bylo provedeno s využitím zdrojů Metacentra, konkrétně se jedná o cluster Lex, který obsahuje dva procesory Intel Xeon E5-2630V3.

Pro testovaní bylo použito všech 1500 pozic ze sady na testování strategie. Na nalezení nejlepšího tahu měl každý engine k dispozici jednu minutu, během které musel vrátit jím nejlepší nalezený tah. Výjimkou byl hráč vracející tah s největší pravděpodobností, protože nevyužívá prohledávání. Tudíž využil jen zlomek času, který měl k dispozici.

Testu bylo také podrobeno několik existujících enginů.

6.1.2 Porovnání výsledků

V tabulce 6.2 je možné vidět výsledky dosažené pomocí naučené neuronové sítě, která porovnává šachové pozice. Tabulka obsahuje celkem čtyři varianty algoritmu Alpha-Beta prořezávání

- Základní algoritmus Alpha-Beta prořezávání
- Alpha-Beta prořezávání rozšířený o transpoziční tabulku
- Alpha-Beta prořezávání s řazením tahů při prohledáváním

Transpoziční tabulka	Řazení tahů	Získáno bodů
Ne	Ne	3144/15000
Ne	Ano	3287/15000
Ano	Ne	2928/15000
Ano	Ano	3028/15000

Tabulka 6.2. Body získané s algoritmem Alpha-Beta prořezávání a naučenou neuronovou sítí.

- Alpha-Beta prořezávání rozšířený o transpoziční tabulku a s řazením tahů při prohledáváním

Z tabulky je možné pozorovat, že po přidání transpoziční tabulky, je dosažený počet bodů menší než v případě nepoužití transpoziční tabulky. Důvodem je, že přidání transpoziční tabulky, vyžaduje provádět více porovnávání pozic pomocí neuronové sítě, než v případě kdy není použitá transpoziční tabulka. Tudíž zrychlení, které přináší použití transpoziční tabulky není dostatečné, aby dorovnalo zpomalené způsobení více porovnáváním pozic neuronovou sítí.

Naopak v případě použití neuronových sítí pro uspořádání prohledávaných tahů, tak s jejich použití bylo dosaženo více bodů. Uspořádání tahů pomocí neonových sítích je méně časově náročné než porovnávání pozic pomocí neuronové sítě.

Transpoziční tabulka	Řazení tahů	Získáno bodů
Ne	Ne	8665/15000
Ne	Ano	9090/15000
Ano	Ne	9107/15000
Ano	Ano	9232/15000

Tabulka 6.3. Body získané s algoritmem Alpha-Beta prořezávání a neuronovou sítí převzatou z Giraffe[2]

V tabulce 6.3 je možné vidět výsledky dosažené pomocí neuronové sítě převzaté z enginu Giraffe[2], která ohodnocuje šachové pozice a použití algoritmu Alpha-Beta prořezávání.

Jako v přechozím případě s neuronovou sítí na porovnávání pozic, jsou v tabulce opět uvedené čtyři varianty algoritmu Alpha-Beta prořezávání stejné jako v předchozím případě.

Z tabulky je možné pozorovat, že nejlepších výsledků bylo dosaženo s použitím transpoziční tabulky a uspořádání tahů. Na rozdíl od neuronové sítě na porovnávání šachových pozic stačí pro každou navštívenou pozici během prohledávání vypočítat ohodnocení pozice pomocí neuronové sítě a s touto hodnotou již dále pracovat, tudíž přidání transpoziční tabulky má minimální dopad na zpomalení prohledávání.

Uspořádání tahů pomocí neuronových sítí lehce zpomaluje prohledávání, ale zároveň zvyšuje množství pozic, které je možné odříznout a prohledat tak do větší hloubky. Tudíž stejně jako v případě s neuronovou sítí na porovnávání pozic, uspořádání tahů přispělo k získání více bodů.

Další tabulka 6.4 obsahuje dosažené výsledky za pomocí algoritmu MCTS a použití neuronové sítě převzaté z enginu Giraffe[2]. V tabulce jsou uvedené dvě varianty algoritmu MCTS

- Varianta kde jsou pozice rovnou ohodnoceny

Popis	Získáno bodů
MCTS s ohodnocováním všech	7489/15000
MCTS s ohodnocováním klidných pozic	3781/15000

Tabulka 6.4. Body získané s použitím algoritmu MCTS

- Varianta kde jsou ohodnocovány pouze klidné pozice. Do klidné pozice se dostává náhodnou simulací založené na neuronových sítí na predikci nejlepšího tahu

Z tabulky je možné pozorovat, že první varianta MCTS bez ohodnocování klidných pozic je výrazně lepší než druhá.

V následující tabulce 6.5 je pro srovnání uvedeno několik exitujících šachový enginů pro které je známé Elo.

Název enginu	Elo	Získáno bodů
Smash	2053	9132/15000
Sayuri	1725	8992/15000

Tabulka 6.5. Body získané běžnými šachovými programy.

Při celkovém srovnání lze pozorovat, že nejlepších výsledků bylo dosaženo s enginem využívajícího Alpha-Beta prořezávání s transpoziční tabulkou, řazením tahů a neuronové sítě převzaté z Giraffe, který se podle získaných bodů pohybuje okolo 2000 Elo.

Z celkových výsledků je vidět, že algoritmus MCTS nedosáhl ani na úroveň základní verze Alpha-Beta prořezávání, ale dosáhl značně lepších výsledků, než naučená neuronová sít, pro porovnávání šachových pozic.

Nejhorších výsledků dosahuje neuronová síť na porovnávání tahů, pravděpodobně to je způsobeno nízkou hloubkou prohledávání, která je způsobena pomalým vyhodnocováním neuronové sítě a také nedosažením přesnosti 98.0% prezentované [3], ale pouze 93.9%

6.2 Hraní mezi sebou

Dalším kritériem použitým pro srovnání implementovaných algoritmů využívající neuronové sítě je odehrání několika her mezi sebou i mezi existujícími šachovými enginy.

6.2.1 Způsob testování

Při experimentu kdy proti sobě hráli jednotlivé enginy bylo opět využito zdrojů Metacentra. Na rozdíl od experimentu s využitím datové sady na testování strategie hraní probíhalo na několika různých hardwarových konfiguracích s podobným výkonem. Avšak vždy vybraná dvojice enginů hrála mezi sebou na stejném stroji, tudíž měli k dispozici vždy stejné prostředky a nebyl tak žádný z dvojice enginů zvýhodněn.

Jelikož se tato práce nezabývá způsobem řízení času, který enginy běžně implementují a je jich nedílnou součástí, která ovlivňuje jejich herní výkon. Tak při experimentech měl vždy každý engine na nalezení tahu minutu a nebyl omezen celkovým časem pro šachovou partii.

Použité ostatní engniny pro srovnání s programy implementovanými v této práci byli omezeny:

- Enginy nemohli používat knihu zahájení.

- Enginy nemohli používat databázi koncových pozic.
- Každý engine prohledával herní prostor, pouze po dobu jemu vymezenou. Herní prostor nemohl prohledávat dopředu, v době kdy právě byl na tahu jeho soupeř.

6.2.2 Porovnání výsledků

Tabulka 6.6 prezentuje výsledky nejlepších programů z každé skupiny, jež byli dříve porovnány na základě sady na testování strategie. V tabulce šachový program uvedený v levém sloupci hraje za bílého hráče, proti šachovému programu z horního řádku, který hraje za černého hráče. Každý šachový program odehrál 100 her proti každému. Vždy půlku her odehrál za bílého hráče a druhou půlku za černého hráče. Za každou hru mu byli přiřazeny body, podle výsledku šachové partie a jejich součet byl zapsán do tabulky.

- 1 bod za výhru
- 0 bodů za remízu
- -1 bod za prohru

- AB — Označuje program s integrovanou neuronovou sítí na porovnávání pozic integrovanou do algoritmu Alpha-Beta prořezávání a uspořádáváním tahů pomocí neuronových sítí
- ABG — Označuje program využívající neuronovou sít z Giraffe[2] integrovanou do algoritmu Alpha-Beta prořezávání s transpoziční tabulkou a uspořádáváním tahů pomocí neuronových sítí
- MCTS — Označuje program využívající neuronovou sít z Giraffe[2] integrovanou do algoritmu MCTS s ohodnocováním všech pozic (i pozic neoznačených jako klidné)

V tabulce jsou také uvedeny šachové programy Smash a Sayuri pro srovnání s existujícími šachovými programy. Z tabulky je patrné, že Alpha-Beta prořezávání s transpoziční tabulkou využívající neuronovou sít z Giraffe[2] a neuronové sítě pro uspořádání tahů. Je schopen se vyrovnat existujícím šachovým programům Smash a Sayuri.

-	AB	ABG	MCTS	Sayuri	Smash
AB	-	-39	8	-50	-21
ABG	48	-	49	15	-7
MCTS	-4	-34	-	-42	-35
Sayuri	49	-10	39	-	-4
Smash	50	-5	38	7	-

Tabulka 6.6. Body získané hraním mezi sebou

Kapitola 7
Závěr

V rámci bakalářské práce byla naučena hluboká neuronová sít porovnávající šachové pozice. Na rozdíl od článku[3], kde bylo dosaženo přesnosti 98.0%, bylo dosaženo přesnosti porovnání jen 93.9%. Pokusy o zmenšení naučenou neuronovou sít na velikost uvedou v článku[3] se nepodařily, protože po zmenšení neuronové sítě její přesnost výrazně klesla.

Naučeno bylo také sedm malých neuronových sítí, které dohromady predikují následující tah. U těchto neuronových sítí bylo požadováno jejich rychlé vyhodnocování, proto výsledné dosažené přesnosti jsou menší. S jinou architekturou neuronové sítě, obsahující více neuronů, je možné dosáhnout větších přesností. Ale z důvodu požadavku na rychlé vyhodnocení těchto neuronových sítí byly zvoleny menší neuronové sítě s menší přesností, ale rychlejším vyhodnocením.

Pro prohledávání herního stromu byly implementovány algoritmy Alpha-Beta prořezávání a MCTS. Algoritmus Alpha-Beta prořezávání byl implementovaný v několika variantách využívající jak naučenou neuronovou sít porovnávající šachové pozice, tak i převzatou neuronovou sít z šachového programu Giraffe[2]. Algoritmus MCTS byl implementován ve dvou variantách s využitím neuronové sítě z šachového programu Giraffe[2].

Na základě provedených experimentů vyplývá, že nejlepších výsledků bylo dosaženo s použitím Alpha-Beta prořezávání využívajícího transpoziční tabulku a heuristiku založenou na neuronových sítích pro uspořádání tahů. Při srovnání s existujícími šachovými programy se implementovaný šachový program pohybuje kolem hodnoty 1900 Elo podle žebříčku CCRL 40/40[21]. Aktuálně nejlepší šachový program Stockfish v tomto žebříčku dosahuje 3391 ELO.

Literatura

[1] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot a others. Mastering the game of Go with deep neural networks and tree search. *Nature*. 2016, 529 (7587), 484–489.

[2] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*. 2015,

[3] Omid E David, Nathan S Netanyahu a Lior Wolf. *DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess*. In: *International Conference on Artificial Neural Networks*. 2016. 88–96.

[4] Stuart Russell a Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3 vydání. Prentice Hall, 2010 .

[5] Louis Victor Allis a others. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen, 1994.

[6] *Negamax*.
`https://chessprogramming.wikispaces.com/Negamax`. Navštíveno: 23. 5. 2017.

[7] *Transposition Table*.
`https://chessprogramming.wikispaces.com/Transposition+Table`. Navštíveno: 23. 5. 2017.

[8] *Zobrist Hashing*.
`https://chessprogramming.wikispaces.com/Zobrist+Hashing`. Navštíveno: 23. 5. 2017.

[9] Dennis Michel Breuker. Memory versus search in games. 1998,

[10] *Evaluation*.
`https://chessprogramming.wikispaces.com/Evaluation`. Navštíveno: 23. 5. 2017.

[11] *Quiescence Search*.
`https://chessprogramming.wikispaces.com/Quiescence+Search`. Navštíveno: 23. 5. 2017.

[12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis a Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*. 2012, 4 (1), 1–43.

[13] *Neuronové sítě*.
`https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=21471`. Navštíveno: 23. 5. 2017.

[14] Michael A Nielsen. *Neural networks and deep learning*.
`https://neuralnetworksanddeeplearning.com`. 2015. Navštíveno: 23. 5. 2017.

[15] Vinod Nair a Geoffrey E Hinton. *Rectified linear units improve restricted boltzmann machines*. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010. 807–814.

[16] Andrew L Maas, Awni Y Hannun a Andrew Y Ng. *Rectifier nonlinearities improve neural network acoustic models.* In: *Proc. ICML*. 2013.

[17] Xavier Glorot, Antoine Bordes a Yoshua Bengio. *Deep Sparse Rectifier Neural Networks.*. In: *Aistats*. 2011. 275.

[18] Diederik Kingma a Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 2014,

[19] Francois Chollet. *Building Autoencoders in Keras*. `https://blog.keras.io/building-autoencoders-in-keras.html`. Navštíveno: 23. 5. 2017.

[20] *Deep Autoencoders*. `https://deeplearning4j.org/deepautoencoder`. Navštíveno: 23. 5. 2017.

[21] *CCRL 40/40*. http://www.computerchess.org.uk/ccrl/4040/. Navštíveno: 23. 5. 2017.

[22] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent a Samy Bengio. Why does unsupervised pre-training help deep learning?. *Journal of Machine Learning Research*. 2010, 11 (Feb), 625–660.

[23] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever a Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting.. *Journal of Machine Learning Research*. 2014, 15 (1), 1929–1958.

[24]

[25] Gaël Guennebaud, Benoît Jacob a others. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[26] *Intel® Math Kernel Library (Intel® MKL) — Intel® Software*. https://software.intel.com/en-us/intel-mkl. 2010.

[27] *UCI protocol*. `http://wbec-ridderkerk.nl/html/UCIProtocol.html`. Navštíveno: 23. 5. 2017.

[28] Dann Corbit a Swaminathan Natarajan. *Strategic Test Suite*. `https://sites.google.com/site/strategictestsuite/about-1`. Přistoupeno: 2017-04-25.

[29] *Pravidla šachu FIDE*. `http://chess.cz/www/assets/files/informace/legislativa/PravidlaSachuFIDE2014.pdf`. Navštíveno: 23. 5. 2017.

[30] *Quiet Moves*. `https://chessprogramming.wikispaces.com/Quiet+Moves`. Navštíveno: 23. 5. 2017.

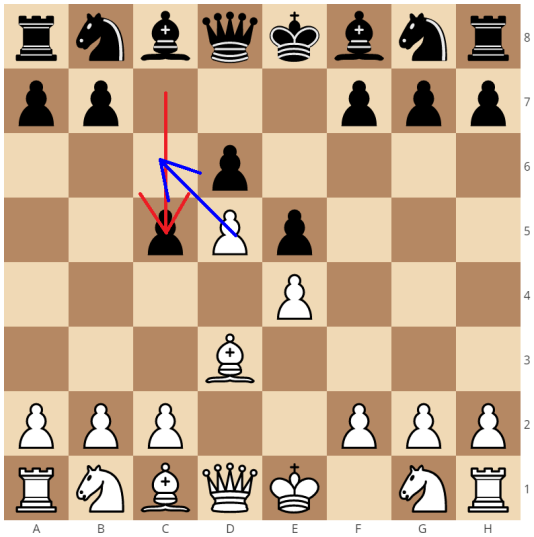
[31] Daniel Ross. *Arpad Elo and the Elo Rating System*. `http://en.chessbase.com/post/arpad-elo-and-the-elo-rating-system`. Navštíveno: 23. 5. 2017.

Příloha A
Používané pojmy

A.1 Braní mimochodem (en passant)

Braní mimochodem (en passant)[29] je označení pro speciální tah pěšce, kdy je možné soupeři sebrat pěšce, který se ze své výchozí pozice posunul o dvě políčka v jednom tahu a přitom přešel přes políčko, které ohrožuje náš pěšec. V takovém případě je možné soupeřova pěšce, sebrat stejným způsobem, jako kdyby provedl tah, při kterém by se posunul jen o jedno políčko. Soupeřova pěšce je možné takto sebrat pouze v bezprostředně následujícím tahu po tom, co takto soupeř táhl pěšcem. Po dokončení tahu stojí náš pěšec na políčku, na kterém by stál soupeřův pěšec v případě, že by se posunul jen o jedno políčko.

Pokud uvažujeme šachovou partii se standartním počátečním rozestavením figurek, pro provedení tahu je nutné, aby měl bílý hráč pěšce v páté řadě a černý hráč provedl tah pěšcem ze sedmé řady do páté řady, vedle pěšce bílého hráče. Pro provedení tahu černým hráčem, je situace obdobná.



Obrázek A.1. Příklad šachové pozice, kde černý hráč posunul pěšce z výchozí pozice o dvě políčka na políčko C5. Následně bílý hráč provádí braní mimochodem, při kterém pěšec bílého hráče z políčka D5 se přesune na políčko C6 a zároveň sebere černému hráči pěšce z políčka C5.

A.2 Klidný tah

Klidný tah [30] je označení pro tah, při kterém nedochází k sebrání figurky nebo povýšení figurky na dámu, věž, koně či střelce. Je možné označovat jako klidný tah, tahy které navíc nezpůsobí šach.

A.3 Klidná pozice

Klidná pozice [11] je označení pro pozici, ve které jsou všechny validní tahy pouze klidné tahy.

A.4 Elo rating

Elo rating [31] je metoda umožňující relativně porovnat úroveň hráčů na základě série her které mezi sebou odehrají. Metodu původně pro použití v šachách vytvořit Arpad Elo, je však možné tuto metodu použít i pro další hry.

Na základě Elo ratingu dvou hráčů je možné vypočítat jejich očekávaný zisk bodů. Čím má jeden z hráčů vyšší Elo rating než druhý hráč, tím je větší očekávaný zisk bodů tohoto hráče.

Očekávaný zisk bodů pro hráče A se vypočte jako

$$E_A = \frac{1}{1 + 10^{\frac{(R_B - R_A)}{400}}}$$

Kde R_A odpovídá Elo ratingu hráče A a R_B odpovídá Elo ratingu hráče B.

Například pokud bude rozdíl v Elo ratingu mezi dvěma hráči 100, tak to odpovídá tomu, že hráč s vyšším Elo ratingem získá 0,64% bodů a hráč s nižším Elo ratingem získá 0,36% bodů.

Příloha B
Obsah CD

- `thesis.pdf` — Elektronická verze textu bakalářské práce.
- `tools.zip` — Pomocné nástroje pro předzpracování dat, testování šachových programů a nástroj umožňující utkání dvou šachových programů.
- `training.zip` — Skripty pro učení neuronových sítí.
- `text.zip` — Zdrojové soubory textu bakalářské práce.
- `source.zip` — Zdrojové soubory implementovaných algoritmů.
- `example.zip` — Spustitelná ukázka.

Příloha C
Popis programu

C.1 Kompilace

Pro kompilaci je vyžadováno gcc alespoň verze 5 a výše a cmake verze 3.2 a výše. Ve výchozím nastavení se kompiluje šachový program s algoritmem Alpha-Beta prořezávání, transpoziční tabulkou, řazením tahů a neuronovou sítí z Giraffe[2]

Pro zkompilování je potřeba zadat následující příkazy:

```
cmake .
make
```

Pro kompilování jiného šachového programu je potřeba v souboru `CMakeLists.txt` odkomentovat příslušné řádky. Např. pro kompilaci jedné z implementací MCTS odkomentovat tento řádek:

```
#add_executable(mcts_eval main_mcts_eval.cpp ${SOURCE_FILES_GIRAFFE}...
```

Pro rychleší vyhodnocování neuronových sítí je potřeba mít nainstalovanou knihovnu Intel® Math Kernel Library[26] a odkomentovat a případně upravit následující řáky:

```
#add_definitions(-DEIGEN_USE_MKL_ALL)
#include_directories("/opt/intel/mkl/include")
#link_directories("/opt/intel/mkl/linux/mkl/lib/intel64/")
#link_libraries(mkl_intel_lp64 mkl_sequential mkl_core gomp pthread m dl)
```

C.2 Ukázka

Ukázka která spustí dva šachové programy proti sobě je na CD v souboru example.zip. Pro její spuštění je vyžadován python verze 3 s nainstalovanou knihovnou python-chess.

Skript v ukázce využívá konfigurace uložené v souboru config.json, kterou při spuštění načte.

Je možné nastavovat následující:

- `WHITE_ENGINE_NAME` — Pojmenování šachového programu
- `WHITE_ENGINE` — Cesta ke spustitelnému soboru šachového programu
- `WHITE_ENGINE_SKIP_LINES` — Kolik řádek výstupu šachového programu se má přeskočit. Některé šachové programy vypisují po startu informace, tímto jsou přeskočeny, a skript je nezpracuje.
- `WHITE_ENGINE_PROTOCOL` — Protolkol pro komunikaci s šachovým programem. Možné volit mezi UCI a XBOARD. Šachový program tento protokol musí podporovat.
- `WHITE_ENGINE_TIME_PER_MOVE` — Jak dlouho může šachový program prohledávat jeden tah.

- `WHITE_ENGINE_CMD` — Možnost zadat vlastní příkaz, který se bude posílat šachovému programu aby započal s prohledáváním.

Stejné možnosti nastavení jsou i pro černého hráče, kde je místo prefixu `WHITE` prefix `BLACK`.

Oba šachové programy z ukázky je možné spustit i samostatně a komunikovat s nimi pomocí základních příkazů protokolu UCI.