Učenje umijetne neuronske mreže igranje šaha

Antun Maldini
Zlatan Sičanica
Bruno Rosan
Jure Perović
Damir Kovačević
Bojan Lovrović
Fakultet elektrotehnike i računarstva
Sveučilište u Zagrebu

I. Uvod

Već pri prvim primjenama računala za igranje igara – štoviše, čak i prije nego su te programe mogli upogoniti na računalima – ljudi su bavili problematikom igranja šaha.

Za igranje šaha, postoje dva glavna pristupa: - brute-force (glupo pretraživanje prostora stanja) uz minimax algoritam, i - korištenje heurističkih pravila. Niti jedan od dva pristupa ne radi dobro sam za sebe, tako da se u pravilu koristi kombinacija obadva. "Klasični" program za igranje šaha izgleda otprilike ovako: za početak se koristi knjiga poteza za otvaranje; nakon početka, koristi se što se već koristi; za kraj, koristi se unaprijed izračunat endgame tablebase. Na taj način se pri kraju i početku igre igraju sigurno optimalni potezi, dok je prostor za AI zapravo u središnjem dijelu igre.

Osnovni algoritam za šah (i druge turn-based igre) je minimax algoritam. Ideja iza algoritma je sljedeća: dva racionalna igrača igraju jedan protiv drugog. Uz nekakvu mjeru dobrote – koliko je stanje dobro za prvog igrača – prvi igrač tu mjeru želi maksimizirati, a drugi je minimizirati. Svaki igrač igra onaj potez koji ga najviše približava tom cilju, odnosno prvi igrač odabire potez najveće mjere dobrote, a drugi najmanje.

Prostor stanja možemo zamisliti kao stablo: početno stanje je korijen, dozvoljena sljedeća stanja su njegova djeca, a prijelaz iz jednog stanja u drugo odgovara odabiru jednog djeteta/podstabla.

Algoritam pretražuje prostor stanja tako da simulira sve moguće igre krenuvši iz početnog stanja, odnosno pretražuje sve grane stabla. Pritom svaki "sloj" (skup svih čvorova na nekoj dubini, engl. ply) odgovara stanju gdje je jedan od igrača na potezu. U korijenu, taj igrač je maksimizator (MAX, odnosno "mi") – on za najbolji potez odabire dijete s najvećom dobrotom; u sloju odmah ispod (na dubini 1), igrač je minimizator (MIN, odnosno "neprijatelj") – on za (svoj) najbolji potez odabire dijete s najmanjom dobrotom; u sloju ispod tog (na dubini 2), igrač je opet MAX, i t.d.

Svaki od igrača zapravo nastoji minimizirati svoj maksimalni mogući gubitak. Tako je najbolji potez maksimizatora maksimum od minumuma maksimumâ ... Hence the name, minimax.

Ako nema više potezâ, znači da je dostignut mat ili remi, i dobrota tog stanja se definira kao +1 (pobjeda), -1 (gubitak), ili 0 (izjednačeno). S obzirom da u pravilu ne možemo pretražiti *cijelo* stablo, nakon neke zadane dubine se za evaluaciju dobrote stanja koristi neka heuristika.

II. Pregled područja

Šah spada u veliku skupinu društvenih igara koje je relativno jednostavno naučiti ali zahtjevno za izvještiti. Drugim riječima, pravila igre su, matematički gledano, dobro definirana, pa je implementacija šaha na računalu predmet proučavanja dugi niz godina. Od 1950-tih razvijeno je stotine različitih programskih rješenja igranja šaha. Uobičajena formulacija problema jest za zadanu trenutačnu konfiguraciju vlastitih i protivničkih figura na ploči, odabrati jedan od dozvoljenih poteza kojim se maksimizira vjerojatnost pobjede. U svrhu spomenutog, prirodno je protivnika modelirati na isti način, pretpostavljajući da će i on vući poteze koristeći isti mehanizam kao i mi (teorija igara).

U duhu principa korištenih u umjetnoj inteligenciji, današnja *eng. state-of-art* programska rješenja ovog problema temelje se na pretraživanju prostora stanja modeliranih stablima pretraživanja. Svaka razina ovog stabla sadrži sve poteze koje igrač koji je na redu smije povući. Čvorovi u zadnjoj razini stabla nazivaju se listovi i njima je s obzirom na ishod partije pridijeljena odgovarajuća ocjena (1, -1 ili 0 u slučaju neriješenog ishoda). Iako bi teoretski mogli u svakom čvoru čuvati sumu ocjene listova do kojih taj čvor vodi i tako u svakom trenutku imati stvarnu ocjenu čvora, u praksi je takvo pridjeljivanje nemoguće izračunati u stvarnom vremenu. Razlog tome je broj mogućih poteza koji raste eksponencijalno nakon svakog odigranog poteza. Umjetna inteligencija ovom problemu doskače podrezivanjem stabala na unaprijed određenoj razini D, tako da se daljnje razine ni ne razmatraju, već se koristi ocjena čvorova te razine dodijeljena funkcijom evaluacije. Ispostavlja se da je implementacija ove funkcije jedan od najbitnijih segmenata u modeliranju sustava a naša zamisao bila je prepustiti aproksimiranje evaluacijske funkcije neuronskoj mreži.

Ideja je iz poznatih odigranih partija izvući veliku količinu konfiguracija, opisati ih značajkama i onda, korištenjem trenutno najpopularnijeg *eng. chess engine*-a Stockfish, svakoj mogućoj konfiguraciji dodijeliti ocjenu na način da bolja ocjena odgovara većoj vjerojatnosti konačne pobijede. Na tako

generiranom skupu podataka učimo neuronsku mrežu koju kasnije, tijekom igre, koristimo za procjenu ocjene određenog poteza.

Šah je između ostalog i *eng.zero-sum* igra što znači da suma svih ocjena teži prema nuli. U skladu s tim, pozitivna ocjena jednom igraču, negativna je za drugog, i obrnuto. Sad je lako za uočiti kako se prvobitni problem svodi na izbor poteza kojim se minimizira protivnikova pobjeda (i samim time maksimizira vlastita), odnosno bira se potez s najlošijom ocjenom za protivnika.

Spomenutim principom vode se poznatija suvremena rješenja zasnovana na strojnom učenju, pa i radovi Erika Bernhardssona te Matthewa Laija, na kojem se temelji naš rad.

Učenjem funkcije koja ocjenjuje svaku konfiguraciju ploče i kombiniranjem s algoritmom pretraživanja stanja možemo napraviti stroj koji igra šah. Erik Bernhardsson je ponudio rješenje u kojem je iz velikog skupa igara naučio funkciju evaluacije poteza preko trenutnog poteza p, sljedećeg poteza q i nasumično odabranog poteza r. Funkcija koju uči pretpostavlja da je sljedeći odabran potez q blizu optimalnog, a nasumično odabrani potez r gori od poteza q. Takva funkcija ne ovisi o pravilima šaha, što znači da njegova mreža mora iz skupa igara moći naučiti što su legalni potezi i kako se figure pomiču. Mreža se sastoji od 3 sloja i na ulaz prima 8*8*12 vrijednosti koje predstavljaju ploču i figure na ploči. Nakon naučene funkcije evaluacije za igranje šaha koristi negamax pretraživanje stanja s alfa-beta odsijecanjem.

Možda glavni problem kod prije spomenute metode je reprezentacija ulaza. Konfiguracije ploče sa sličnom evaluacijom bi trebale biti međusobno blizu u ulaznom prostoru. Rješenje Matthewa Laija iz svake konfiguracije izvlači više informacije od same pozicije figura. Rezultat je prostor značajki od oko 360 značajki za svaku konfiguraciju koje uzimaju u obzir koje su figure napadnute, koje obranjene, prava na rokadu te koliko daleko klizeće figure mogu ići, među ostalima. Navedene značajke čine prostor značajki gdje se slične situacije u igri nalaze međusobno blizu u ulaznom prostoru. Na temelju ovakvih značajki duboka neuronska mreža može naučiti složenije koncepte i skrivena znanja u igri šaha bez potrebe za implementiranjem složenijih pravila, te se smanjuje potreba za dubljim pretraživanjem prostora stanja prilikom odluke.

Njegova evaluacijska funkcija koristi prilagođenu verziju TD-Leaf algoritma koja procjenu ne temelji na ishodu partije već na odlukama koje bi ista funkcija donijela u nekom potezu u bliskoj budućnosti. Također, dio stabla koji se uzima u obzir kod pretraživanja nije određen dubinom kao kod većine današnjih sličnih sustava, već vjerojatnošću da odabrana staza u stablu bude dio teoretski optimalne staze koja vodi do kraja partije uzimajući u obzir da se oba igrača vode istom logikom igranja.

III. Implementacija

Implementacija se temelji na diplomskom projektu studenta Matthewa Laija [2]. Ovo poglavlje će proći kroz postupak generiranja skupa za učenje, izvlačenje vektora značajki za pojedini ulazni primjer, izradu te na kraju treniranje mreže.

Tablica I: Vrijednosti figura.

Figura	Vrijednost
Pijun	1
Konj	3
Lovac	3
Kula	5
Kraljica	9
Kralj	15

A. Generiranje podataka za učenje

Prije svega potrebno je bilo generirati skup podataka za učenje. Pri tome se je koristila baza igara gdje je za svaku igru bila poznata lista poteza. Iz ove liste je dobivene lista stanja kroz cijelu igru te je svakom stanju pridružena ocijena dobivena evaluacijom. Iako je [2] rađen uz pomoć vlastite evaluacijske funkcije, mi smo koristili Stockfish. Lista parova stanja i ocjene u konačnici čini naš skup primjera za učenje.

B. Vektor značajki

Izvlačenje vektora značajki se dijeli na tri grupe: globalne značajke, značajke vezane uz figure te značajke vezane uz kvadrate. Valja napomenuti da iako pojedini elementi ovoga vektora poprimaju vrijednosti iz različitih domena, na kraju izvlačenja ove su vrijednosti transformirane u domenu $[0, 1]$.

Globalne značajke započinju sa binarnom vrijednosti koja predstavlja informaciju jeli bijeli na redu. Slijede četiri skalara od kojih je svaki zastavica sa vrijednosti postavljenom na jedan ukoliko je trenutačno omogućena ta rošada. Posljednji u listi globalnih značajki je podvektor kod kojeg je svaki element broj trenutno prisutnih figura na ploči za pojedini tip figure.

Značajke vezane uz figure su sastavljene od liste figura sa lokacijama i ostalim podatcima potrebnim ovisno o tipu figure. Za reprezentaciju ploče bilo je potrebno koristiti *python chess* koji u sebi nema mehanizam diskriminacije različitih figura istog tipa, dok je u [2] takvo svojstvo navedeno kao obavezno. To je bio glavni razlog za izradu našeg sučelja za kontrolu stanja ploče, izgrađenog povrh već postojećeg *python chessa*. Naše sučelje sprema poziciju svake figure te ju ažurira prilikom svakog pokreta jednog od igrača, tako da se one mogu međusobno razlikovati. Ovo u praksi znači da će informacije o ljevom lovcu uvijek stizati na istom mijestu u vektoru značajki bez obzira gdje se on na ploči nalazio (čak i ako si lijevi i desni lovac zamijene lokacije). Uz lokaciju, za neke je figure potrebno poslati još i dozvoljeno klizanje u bilo kojem dozvoljenom smjeru. Ovdje se konkretno radi o četiri broja za kulu i lovca, te osam za kraljicu, budući da ona može klizati u osam smjerova.

Značajke vezane uz kvadrate imaju ocjenu najslabijeg napadača i najslabijeg branitelja za pojedino polje i te su dvije vrijednosti prisutne za svaki kvadrat (polje) na ploči. Vrijednosti pojedinog tipa figure dane su u tablici I.

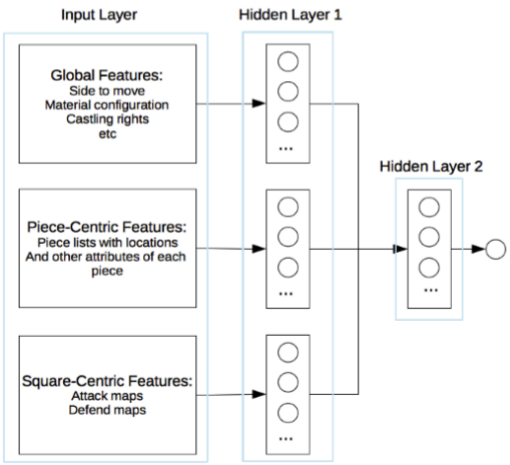
C. Izrada mreže

Topologija mreže se sastoji od dva skrivena sloja te jednog izlaznog kao na slici 1. Ulazni sloj je sa prvim skrivenim

Tablica II: Broj neurona po slojevima.

Sloj	Broj neurona
Ulazni (prvi dio)	15
Ulazni (drugi dio)	144
Ulazni (treći dio)	128
Prvi skriveni (prvi dio)	10
Prvi skriveni (drugi dio)	110
Prvi skriveni (treći dio)	100
Drugi skriveni	100
Izlazni	1

povezan lokalno, dok su prvi i drugi skriveni povezani potpuno. Drugi skriveni sloj je sa izlaznim slojem odnosno neuronom također povezan potpuno. Brojevi neurona po slojevima mreže mogu se vidjeti u tablici II.



Slika 1: Topologija mreže.

D. Treniranje mreže

Za treniranje mreže korišten je skup primjera ocjenjenih konfiguracija ploče. Preuzet je niz šahovskih partija koje su naknadno označene alatom Stockfish. Ocjene su normalizirane na interval od -1 do 1, kako bi odgovarale izlaznoj funkciji, tangensu hiperbolnom. Funkcija gubitka definirana je kao srednje kvadratno odstupanje izlaza mreže od očekivane vrijednosti. Za minimizaciju gubitka korišten je backpropagation algoritam s Adam optimizatorom. Rezultati nažalost nisu ispunili očekivanja jer se srednja kvadratna greška nije uspjela spustiti ispod 1.05, što je loš rezultat za interval $[0, 1]$. Uz to, kad bi se mreža koristila za igranje, čini se kao da povlači prvi potez koji joj se ponudi, umjesto obavljanja detaljnije analize. Razlog za ovo je vjerojatno sam algoritam optimizacije koji nije dovoljno jak da bi rješio ovako složen problem.

E. Program

Izvorni kod za izvođenje programa je pohranjen u datotekama: *AlternativeTrainer.py*, *ChessANNBoardInterface.py*, *Evaluator.py*, *FeatureExtractor.py*, *GameLoader.py*, *Network.py* i *NetworkTrainer.py*. Dok je *Network.py* samo prisutan za xor testiranje mreze, *NetworkTrainer.py* i *AlternativeTrainer.py* služe za treniranje. *Evaluator.py* se je koristio za izradu skupa za učenje.

IV. Zaključak

Iako se je implementacija pokazala neuspiješnom dala je uvid u najbolje tehnike ostvarenja ovakvog sustava ([2] i [1]) kao i dostupne alate za brže dolaženje do rezultata. Činjenica je da je ovaj projek bio preambiciozni pothvat. Glavni razlog je što ne postoji "točno" riješenje već je bilo potrebno do dovoljno dobrog vektora značajki dolaziti empirijskim metodama, što nam je osobno potvrdio i autor [2]. U njegovom radu, na kojem je ovaj projekt temeljen, za optimizaciju se koristio TD-Leaf(λ) algoritam, koji je vjerojatno efikasniji u rješavanju ovakvih problema. S pozitivne strane, barem je pokazano kako ovakva verzija algoritma za učenje nije efikasna za problem igranja šaha.

References

[1] Erik Bernhardsson. Deep learning for... chess, 2014. https://erikbern.com/2014/11/29/deep-learning-for-chess.

[2] Matthew Lai. Using deep reinforcement learning to play chess. 2015. https://chessprogramming.wikispaces.com/Giraffe.