hyväksymispäivä arvosana

arvostelija

Koneoppiminen shakkitekoälyissä

Verna Koskinen

Helsinki 8.5.2016

Kandidaatin tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Verna Koskinen

Työn nimi — Arbetets titel — Title

Koneoppiminen shakkitekoälyissä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatin tutkielma	8.5.2016	23 sivua + 0 liitesivua

Tiivistelmä — Referat — Abstract

Tässä tutkimuksessa tutustutaan koneoppimisen hyödyntämiseen shakkitekoälyn toteuttamisessa. Tutkielmassa tarkastellaan shakkitekoälyn kehitykseen liittyviä ongelmia tutustumalla ensin perinteisiin shakkimoottoreihin ja niiden tekniikoihin. Sen jälkeen tutkitaan koneoppivia shakkitekoälyjä ja kuinka niiden avulla perinteisten shakkimoottorien ongelmakohtia on pyritty ratkaisemaan.

Tutkielman tavoitteena on löytää niitä keinoja, joiden on havaittu toimivan tehokkaasti ratkaisuina yleisimpiin ongelmiin. Tutkielmassa tarkastellaan erityisesti arviointifunktion tuottamista vahvistusoppimisen keinoin, koska tällä tekniikalla on saatu erittäin lupaavia tuloksia koneoppivien shakkitekoälyjen saralla. Lisäksi tutustutaan aivan uuteen tapaan toteuttaa nollasummapeleissä yleisen syvyyshaun tarjoama toiminnallisuus koneoppimista hyödyntäen.

Tutkimuksessa osoitetaan, että vahvistusoppiminen yhdessä neuroverkkojen kanssa soveltuu hyvin koneoppivan shakkitekoälyn perustaksi. Yleisin sovelluskohde on arviointifunktio ja sen toteutukseen liittyvät yksityiskohdat käydään läpi tarkasti.

Vaikka koneoppivat shakkitekoälyt eivät yllä vielä täysin perinteisten shakkimoottorien tasolle, on kehitys ollut nopeaa ja uusimmat tulokset erittäin lupaavia. Tutkimuksen tuloksena esitetyt tekniikat soveltuvat hyvin myös muiden shakin kaltaisten nollasummapelien tekoälyjen kehitykseen.

ACM Computing Classification System (CCS):
I.2.6 [Artificial Intelligence]: Learning
I.2.6 [Artificial Intelligence]: Problem Solving, Control Methods, and Search

Avainsanat — Nyckelord — Keywords
tekoäly, vahvistusoppiminen, tilanne-ero-oppiminen, shakki

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Shakkitekoälyt ovat 1900-luvun puolivälistä lähtien olleet tutkijoiden kiinnostuksen kohteena ja useita shakkimoottoreita ja julkaisuja on syntynyt sen tuloksena [Hsu99, Hya77, MS10]. Shakkitekoälyissä on yritetty hyödyntää jo pitkään koneoppimista, mutta vasta aivan viime aikoina ollaan päästy tuloksiin, jotka yltävät lähelle parhaita perinteisiä shakkimoottoreita [BBT$^+$13, Lai15].

Koneoppiminen on tekoälytutkimuksen osa-alue, jossa koneoppimisalgoritmi luo datan perusteella mallin, jota voidaan käyttää ennustamaan tuloksia sille syötetyistä arvoista. Shakkimoottoreissa paljon käytetty koneoppimisen alalaji on vahvistusoppiminen, jota voidaan hyödyntää esimerkiksi pelitilan arviointifunktion luomiseksi.

Tässä tutkielmassa tarkastellaan millä keinoin koneoppimista on hyödynnetty shakkimoottoreissa, sekä tutustutaan joihinkin koneoppiviin shakkimottoreihin. Käytetty tutkimusmenetelmä on kirjallisuuskatsaus. Tutkimuksessa on keskitytty etenkin uusimpiin, nykyaikaisiin koneoppiviin shakkimoottoreihin ja niissä käytettyihin tekniikoihin.

Aluksi käsitellään perinteisiä shakkimoottoreita tutustuttaen niiden historiaan ja yleisiin algoritmeihin, jotka niiden taustalta löytyvät. Seuraavaksi tutustutaan todennäköisyysrajoitettuun hakuun joka on yksi uusimmista tavoista hyödyntää koneoppimista shakkimoottoreissa. Lopuksi tutustutaan tarkemmin vahvistusoppimiseen, joka on perusta suurimmalle osalle nykyaikaisista koneoppivista shakkitekoälyistä. Lopuksi esitellään muutamia käytännön toteutuksia, sekä niiden esiin tuomia parannuksia ja haasteita.

1.1 Shakkitekoälyjen historia

Shakkitekoälyt ovat kiinnostaneet ihmisiä jo pitkään ennen kuin niiden toteuttaminen on ollut mahdollista. Ensimmäinen tunnettu niinsanottu shakkitekoäly Turk oli unkarilaisen Wolfgang von Kempelenin rakentama 1769-luvulla [Gop13]. Laitteessa oli hieno mekaaninen sisus jota esiteltiin yleisölle ennen peliä, saaden heidät uskomaan laitteen aitouteen. Todellisuudessa esiteltävien rattaiden takana oli ihminen, joka teki siirrot salassa laitteen sisältä. Turk oli niin taitavasti rakennettu ja esiteltiin sopivasti teollisen vallankumouksen alussa, että se paljastui huijaukseksi vasta kun sitä oli ehditty kierrättää pelaamassa ympäri maailmaa 84 vuoden ajan.

Ehkäpä ensimmäinen todellinen shakkitekoäly oli Alan Turingin ja David Champernownen yhteistyössä 1984-luvulla kehittämä Turochamp [Cop04]. Tietokoneet eivät olleet vielä riittävän kehittyneitä ohjelman ajamiseen, mutta kaikki säännöt oli tarkkaan paperille kirjoitettuja. Turing itse simuloi koneen toimintaa. Turochampissa oli paljon nykyisissä shakkimoottoreissa nähtäviä piirteitä, kuten arviointifunktio, alkukantainen syvyyshaku ja erillinen loppupelin strategia. Turing ja Champernown kertovat jälkikäteen myös leikkineensä ajatuksella, että tärkeitä siirtoja tulisi tutkia syvemmälle ja huonoja siirtoja voisi jättää tutkimatta.

Todellinen läpimurto shakkimoottoreiden historiassa tapahtui 1997, kun Deep Blue voitti Garri Kasparovin turnauksessa [Hsu99]. Deep Bluen taustalla pyöri 30 kappaletta varta vasten shakkiin suunniteltua supernopeaa suoritinta ja sille arvioitiin Elo-luvuksi 2875. Elo-luku [Wor14] on shakissa yleisesti käytetty matemaattinen malli jolla voidaan selvittää pelaajan taso verrattuna muihin pelaajiin. Deep Blue jäi kuitenkin eläkkeelle voitettuaan Kasparovin ja pelasi liian vähän pelejä jotta vahvuuslukua voisi pitää vertailukelpoisena.

Nykyisiä shakkimoottoreita verrataan usein GNU Chess [MS10] moottoriin, joka on tieteellisiin tarkoituksiin kehitetty, yksi vanhimmista avoimen lähdekoodin shakkimoottoreista. Sen vahvuusluku on ollut noin 2800, jolla se yltäisi ihmispelaajana parhaan suurmestarin luokitukselle helposti. Sen toiminta perustuu pitkälti tehokkaaseen hakuun ja sen arviointifunktio on hyvin kevyt.

Yksi kaikkien aikojen vahvimmista shakkimoottoreista on myöskin avoimen lähdekoodin projekti, Stockfish [RCK16]. Sen kehitys alkoi vuonna 2008 ja sitä päivitetään yhä kuukausittain. Stockfishin taustalla on hyvin tarkkaan hiotut arviointifunktiot ja hakualgoritmit, jotka sisältävät laajan määrän manuaalisesti koodattua shakkitietoutta. Sen Elo-luku nousee jopa yli 3500:an ja se hallitsee kaikkia CCRL [BSB+16] (Computer chess rating list) luokituslistoja.

2 Perinteiset shakkitekoälyt

Perinteiset shakkitekoälyt perustuvat raakaan laskentavoimaan, mutta shakin pelipuun koko on niin suuri, ettei ole järkevää edes yrittää ratkaista sitä kokonaan. Pelipuu on puurakenne, jossa nykyinen pelitila on juurisolmu ja kaikki mahdolliset sitä seuraavat tilat sen lapsisolmuja. Lapsisolmuja seuraavat tilat ovat jälleen alemman tason solmut ja niin edespäin. Kun mahdollisia siirtoja on paljon, kasvaa pelipuun koko hyvin nopeasti niin suureksi, että sen käsittely vaatii valtavan määrän laskentatehoa ja muistia.

Lähes kaikki nykyaikaiset shakkimoottorit perustuvat pelipuun tutkimiseen syvyysrajoitteella, jonka aikana ja jälkeen lehtisolmuja arvioidaan erinäisillä pisteytysperiaatteilla. Sen lisäksi moottorit pyrkivät eri tavoin karsimaan tutkittavia haaroja, jotta ne eivät haaskaisi laskentatehoa parhaan siirron kannalta huonojen haarojen tutkimiseen. Karsintasääntöjen kehittäminen on kuitenkin hankalaa ja epäluotettavaa ja ne ovatkin tärkeimpiä tutkimuskohteita shakkitekoälyjen kannalta tänä päivänä.

2.1 Minimax

Minimax [CM83] algoritmi tutkii pelipuuta rekursiivisesti ja määrittelee pisteet jokaiselle pelitilalle sillä oletuksella, että molemmat pelaajat pelaavat optimaalisesti. Se pystyy tuottamaan optimaalisen ratkaisun yksinkertaisille peleille, joissa pelipuun koko ei kasva niin suureksi kuin shakissa.

Jotta minimaxia voisi käyttää suuremmilla pelipuilla, on kehitetty syvyysrajoitteinen minimax algoritmi, joka toimii myös useimpien shakkimoottorien ytimessä. Siinä pelipuuta ei laskentatehon säilyttämiseksi tutkita ennalta määriteltyä syvyyttä pidemmälle. Toimiakseen syvyyshaku vaatii arviointifunktion, jolla tiloja voidaan pisteyttää vaikkei oltaisi vielä lehtisolmussa.

Syvyysrajoitetun minimaxin ongelmana on niin sanottu horisontti-ilmiö [Lai15]. Siinä minimax lopettaa tutkinnan tilaan, joka näyttää hetkellisesti hyvältä, esimerkiksi kuningattaren syötyä vastapuolen sotilas. Se ei kuitenkaan kykene huomaamaan, että heti seuraavassa tilassa vastapuolen sotilas pystyy nyt syömään kuningattaren.

Horisontti-ilmiön välttämiseksi on kehitetty erilaisia tekniikoita, kuten uinuva (quiescent) haku. Uinuva haku käynnistyy mikäli syvyyshaku pysähtyy tilaan, joka on jollakin tavalla vaarallinen, kuten vastustajan nappulan syömiseen. Se lisää solmuja hakuun ja arvioi alkuperäisen tilan vasta löytämänsä rauhallisen tilan perusteella, esimerkiksi kun yhtään pelinappulaa ei ole juuri syöty. Uinuvan haun varjopuolena on sen vaatima laskentatehon lisäys, jonka seurauksena syvyysrajoitetta voidaan joutua madaltamaan. On lisäksi vaikeaa rajata tiloja, joissa uinuva haku käynnistyy niin, että algoritmista saataisiin mahdollisimman suuri hyöty verrattuna horisonttiilmiön haittoihin.

2.2 Alpha-beta karsinta

Yleisesti käytetty karsintasääntö pelipuun haarojen tutkimiselle on alpha-beta karsinta [Bau78], jossa minimax algoritmiin lisätään ala- ja yläraja pisteytykselle. Jotta haaraa tutkittaisiin kokonaan, sen pisteytyksen tulee osua näiden rajojen määrittämään ikkunaan.

Olennaisinta alpha-beta karsinnan toiminnan kannalta on tutkittavien haarojen järjestys [Lai15]. Mikäli haarat ovat optimaalisessa järjestyksessä, nopeuttaa karsinta minimax algoritmia niin paljon, että pelipuuta voidaan samassa ajassa tutkia jopa kaksi kertaa syvemmälle. Epäoptimaalisessa järjestyksessä taas alpha-beta karsinta ei nopeuta minimaxia lainkaan.

Optimaalista järjestystä on mahdoton määrittää ennalta, koska jos paras haara tunnettaisiin jo ennalta, ei koko etsintävaihetta tarvittaisi. Siksi alpha-beta karsintaa varten shakkimoottoreissa käytetään paljon erilaisia heuristiikkoja, joilla järjestyksen voisi saada mahdollisimman lähelle optimaalista. Tämä on suurimpia ongelmia hyvien karsintasääntöjen kehittämisessä ja usein sääntöihin on päädytty hyvin manuaalisilla menetelmillä.

2.3 Arviointifunktio

Syvyysrajoitetun haun vaatima arviointifunktio on yksi shakkimoottorin olennaisimmista osista. Arviointifunktion tarkoitus on tuottaa arvio annetun pelitilan hyvyydestä ilman, että tilasta voidaan katsoa siirtoja eteenpäin.

Lähes kaikki parannukset perinteisiin shakkimoottoreihin juontavat juurensa arviointifunktioiden kehittämisestä [Lai15]. Arviointifunktiot sisältävät suuren määrän funktioiksi käännettyä shakkitietoutta ja voivat olla tuhansia koodirivejä pitkiä. Esimerkiksi StockFishin pelkkä sotilaiden arviointi on lähes 300 koodiriviä pitkä [RCK16].

Laajojen käsin koodattujen arviointifunktioiden ongelmaksi on osoittautunut arvojen hienosäädön vaikeus. Vaikka useiden perinteisten shakkimoottorien arviointifunktiot ovat StockFishin vastaavaa yksinkertaisempia, on StockFish hyvä esimerkki siitä, millaisia muuttujia funktioissa saatetaan huomioida. Tätä tietoa voidaan käyttää hyväksi mm. vastaavan koneoppimisalgoritmin piirteitä valittaessa.

StockFishin arviointifunktio koostuu yhdeksästä osasta, jotka on lueteltu alla hiukan yksinkertaistaen:

- Materiaaliarvo määrittelee jokaiselle laudalla olevalle pelinappulalle jonkin arvon ja osoittaa lisäksi etuja yhteistoiminnasta, esimerkiksi kun molemmat lähetit ovat yhä laudalla.
- Pelinappuloiden sijaintitaulut pisteyttävät pelinappuloiden sijainnin, riippumatta muista pelinappuloista.
- Sotilaiden rakenne pisteyttää tai rankaisee sotilaita riippuen niiden ja 2-3 lähimmän muun pelinappulan sijoittumisesta toisiinsa nähden.
- Pelinappulakohtaiset bonukset riippuvat pelinappulan tyypistä. Esimerkiksi torni saa bonuksen ollessaan samalla tasolla vastustajan sotilasrivin kanssa.
- Liikkuvuusbonusta varten lasketaan montako laillista siirtoa kullakin pelinappulalla on.
- Kuninkaan suojauksessa tarkastellaan onko kuningas linnoittautunut ja kuinka paljon sen lähellä on vihollisen nappuloita.
- Uhka jakaa pisteitä, tai rankaisee, riippuen siitä onko pelinappula suojattu.
- Tilabonus tulee vapaasta turvallisesta tilasta omalla puolella pelilautaa.
- Tasapelimäisyys tarkastelee kummankin puolen materiaaliarvoja ja joissain tilanteissa skaalaa koko arvion lähemmäs nollaa.

3 Todennäköisyysrajoitettu haku

Koneoppimista hyödyntävää Giraffe shakkimoottoria [Lai15] varten kehitettiin ainutlaatuinen todennäköisyysrajoitettu haku, joka pyrkii ratkaisemaan syvyysrajoitettuun hakuun ja pelipuun karsintaan liittyviä ongelmia. Siinä etsintää ei lopeteta

tietyssä syvyydessä, vaan silloin kun todennäköisyys, että solmu kuuluu optimaaliseen lopputulokseen on liian pieni. Seuraavaksi keskitymme todennäköisyysrajoitetun haun toimintaan Giraffen yhteydessä.

Shakin pelipuun hakuongelman tavoitteena on löytää tilat, joihin päädytään, jos molemmat pelaajat pelaavat teoreettisesti optimaalista peliä ilman, että koko pelipuuta käydään läpi. Eräs lähestymistapa ongelmaan on kuvata jokaiselle tilalle teoreettinen periaatevariaatio (theoretical principal variation), joka kertoo kuinka suurella todennäköisyydellä kyseinen tila on osa teoreettista tilasarjaa, joka johtaa optimaaliseen peliin. Käytännössä teoreettisia periaatevariaatioita voidaan laskea jollekin tilalle useita, sillä voi olla mahdollista pelata optimaalinen peli usealla eri tavalla. Laskennan yksinkertaistamiseksi oletetaan kuitenkin, että jokaisella tilalla on ainoastaan yksi periaatevariaatio. Nyt hakuongelma voidaan kuvata niin, että sen tavoitteena on tuottaa mahdollisimman varmasti periaatevariaatioon kuuluvia tiloja.

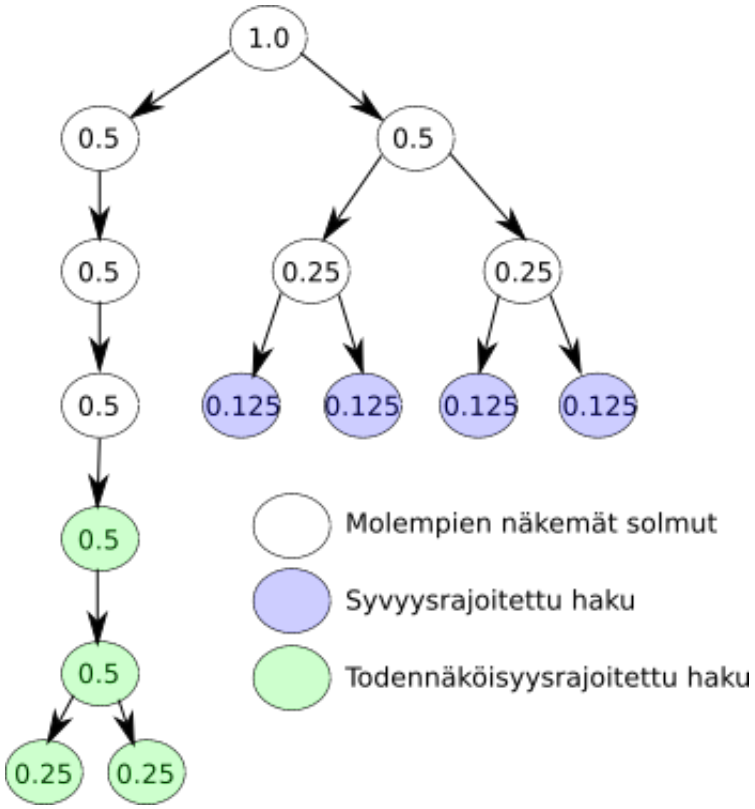
Etsityt solmut ovat suurimmassa osassa tapauksista samoja kuin syvyysrajoitetussa haussa. Todennäköisyysrajoitetun haun hyödyt tulevat esiin tilanteissa, joissa on vain vähän mahdollisia siirtoja, jolloin haaroja voidaan seurata syvemmälle. Tällainen tilanne on esimerkiksi silloin, kun toinen pelaajista on pelannut shakin, jolloin mahdollisia siirtoja on usein vain muutamia.

Ymmärtääksemme miksi todennäköisyysrajoitettu haku toimii edellä kuvatussa tilanteessa paremmin, meidän tulee katsoa lähemmin miten se käytännössä toimii. Sen sijaan, että etsisimme esimerkiksi 10 siirron päähän tutkimastamme juurisolmusta, tutkitaan todennäköisyysrajoitetussa haussa ne solmut, joiden todennäköisyys kuulua teoreettiseen periaatevariaatioon on suurempi kuin esimerkiksi 1×10^{-7}. Aiemmasta oletuksesta, jossa teoreettisia periaatevariaatioita on vain yksi joka tilalle, voidaan johtaa hyvin naiivi implementaatio hausta, jossa juurisolmun todennäköisyys on 1 ja tämä jaetaan tasan kaikille solmun lapsille. Vastaavasti jokaisen lapsisolmun lapsen kesken jaetaan näiden vanhempien todennäköisyys, niin että $P(lapsi_i|vanhempi) = \frac{P(vanhempi)}{n}$ kun n on lapsisolmujen lukumäärä.

Kuvassa 1 esitetään tilanne, jossa syvyys- ja todennäköisyysrajoitetun haun tutkimat solmut eroavat edellä kuvatulla naiivilla implementaatiolla, kun syvyysrajoite on asetettu 4:ään ja todennäköisyysrajoite on vastaavasti 1/4. Todennäköisyysrajoitettu haku kykenee esimerkissä tutkimaan oikeanpuoleista haaraa, jossa mahdollisia siirtoja on viiden siirron ajan vain yksi, sinne saakka kunnes se haarautuu seitsemännellä tasolla. Mikäli kyseessä olisi ollut esimerkiksi kuninkaan uhkaus, olisi parantunut tutkintasyvyys voinut antaa selvän edun syvyysrajoitettuun hakuun verrattuna.

3.1 Todennäköisyyksien arviointi

Jotta solmun todennäköisyys kuulua teoreettiseen periaatevariaatioon voidaan laskea täsmällisemmin, voidaan käyttää neuroverkkoa todennäköisyyden $P(lapsi_i|vanhempi)$ oppimiseen tilan ominaisuuksien perusteella. Neuroverkkoja käsittelemme syvälli-



Kuva 1: Esimerkki syvyysrajoitetun ja todennäköisyysrajoitetun haun eroista tutkittavien solmujen osalta [Lai15].

semmin kappaleessa 4.4. Meidän tulee tässä vaiheessa tietää, että on tärkeää valita sopivasti algoritmille syötettävät piirteet, eli syötearvot joilla neuroverkkoa opetetaan. Piirteet tulee valita niin, että saadaan riittävästi tietoa täsmällisen arvion tuottamiseen ja jotta lopputuloksen kannalta turhilta ja ainoastaan laskeantakompleksisuutta lisääviltä piirteiltä vältyttäisiin.

Hyvä lähtökohta piirteiden valintaan on käyttää samoja piirteitä jotka ovat arviointifunktion kannalta hyviä [Lai15]. Piirteitä voidaan siis tunnistaa mm. tutkimalla perinteisiä arviointifunktioita. Vanhempisolmua hyvin kuvaavia piirteitä onkin esimerkiksi kaikkien pelinappuloiden sijainnit kartalla, linnoitusmahdollisuudet, pelinappuloiden kantamat ja hyökkäys- ja puolustuskartat. Niiden lisäksi on hyödyllistä lisätä piirteitä, jotka kuvaavat lapsisolmuun johtavaa siirtoa. Todennäköisyysrajoitteisen haun kehittäjä Matthew Lai käytti Giraffessa arviointifunktion piirteiden lisäksi viittä siirtymää kuvaavaa piirrettä. Piirteet olivat siirretyn pelinappulan tyyppi, pelinappulan alkuperäinen ja siirron jälkeinen sijainti, miksi nappulaksi pelinappula mahdollisesti ylennettiin, sekä siirron asema verrattuna muihin laillisiin siirtoihin.

Siirron asemaa lukuunottamatta piirteet on helppo laskea. Asemalla tarkoitetaan

sitä kuinka hyvä siirto oli verrattuna muihin laillisiin siirtoihin. Ajatellaan vaikka tilannetta jossa kaikista laillisista siirroista neljä vaikuttaa selvästi muita siirtoja paremmilta. Silloin muita siirtoja on turha tutkia enempää, koska vaikka jokin neljästä selvästi paremmaksi arvioiduista osoittautuisikin huonoksi, on todennäköistä, että ainakin jokin niistä on aina parempi kuin muut alunperin huonommiksi arvioidut siirrot. Jos sen sijaan selvästi parempia siirtoja on vain yksi, on kannattavaa tutkia myös muita vaihtoehtoja siltä varalta, että yksinäinen siirto osoittautuisikin huonoksi. Siirron asemalla pyritään siis välttämään tilanne, jossa arvioon luotetaan liian paljon ja hyväksymään ettei opittu arvio vastaa todellisuutta täydellisesti.

Koska ennen siirtojen arviointia ei voida mitenkään tietää miten ne asettuvat verrattuna muihin siirtoihin, on arvio tehtävä kahdesti. Ensin arvioidaan jokainen siirto aivan kuin se olisi paras mahdollinen siirto ja sen jälkeen arvioidaan siirrot uudestaan käyttäen aiemmalla kierroksella saatuja arvioita aseman selvittämiseksi.

Kun tämä toimenpide toistetaan jokaiselle arvioitavalle tilalle, saadaan aikaan tilan laillisille siirroille jokin hyvyysmitta. Sen jälkeen tulos normalisoidaan todennäköisyysjakauman muodostamiseksi. Lapsisolmut kerrotaan vanhempisolmun todennäköisyydellä, jotta saadaan jokaisen lapsisolmun todennäköisyys kuulua teoreettiseen periaatevariaatioon kaavan 1 mukaisesti,

$$P(lapsi_i|vanhempi) = \frac{H(lapsi_i)}{\sum_{j=1}^{n} H(lapsi_j)} \cdot P(vanhempi) \tag{1}$$

jossa n on vanhemman lasten lukumäärä ja H siirron hyvyysmitta. Tätä tietoa voidaan käyttää samoin kuin aiemmassa esimerkissä, jossa todennäköisyys muodostettiin naiivisti jakamalla vanhempisolmun todennäköisyys lapsisolmujen lukumäärällä.

Neuroverkon toteutus muilta osin seuraa samaa kaavaa kuin arviointifunktion neuroverkkototeutuksessa, joka esitellään kappaleessa 4.4.

Todennäköisyysrajoitetun haun opettamiseen voidaan käyttää ohjattua oppimista, sillä se keskittyy juurisolmuihin joiden hyvyys voidaan arvioida tilojen arviointifunktiolla, joka on toteutettu luvun 4 mukaisesti. Opetukseen voidaan käyttää esimerkiksi stokastista gradienttilaskeutuma-algoritmia (stochastic gradient descent) [Bot10].

Juurisolmuihin keskittymisen ansiosta opetusdatana voidaan käyttää hyvin shakissa usein vastaan tulevia tiloja kuvaavaa otosta. Todennäköisyysrajoitettu haku ei tutki haaroja, jotka johtaisivat esimerkiksi suuriin materiaalisiin epätasapainoihin, koska näiden tilojen todennäköisyys kuulua optimaaliseen peliin on hyvin pieni. Opetusdata voidaan siksi koostaa joko suoraan pelattujen pelien tietokannasta tai satunnaisesti valituista aikarajoitetulla haulla löydettyjen parhaiden pelihaarojen sisäisistä tiloista, kuten Giraffen tapauksessa oli tehty.

Opetusjoukon tilojen merkitsemiseen voidaan myös käyttää aikarajoitetun haun ja arviointifunktion yhdistelmää. Näin tilojen merkitseminen käytettäväksi ohjatussa oppimisessa ei vaadi ihmistyötä ja opetusjoukon kokoa voi helpommin kasvattaa.

4 Arviointifunktio koneoppimisella

Koneoppimiseen liittyvät tekniikat ovat kehittyneet viime aikoina osittain laskentatehon parantumisen ja osittain saatavilla olevien datamäärien kasvun myötä. Merkittävä edistys koneoppivien tekoälyjen suunnittelussa oli TD-Gammon, [Tes95], joka itseään vastaan pelaamalla oppi backgammonissa niin eteväksi, että se kykeni voittamaan hallitsevan maailmanmestarin useimmissa pelaamistaan peleistä. Sen perustana oli vahvistusoppiminen ja yksinkertainen neuroverkko pelilaudan käsittelyyn. Vahvistusoppimisessa kone kokeilee erilaisia siirtoja ja saa vasta lopuksi tietää toimintojen seurauksen. Neuroverkkoja käytetään epälineaaristen riippuvuussuhteiden oppimiseen ja siksi ne soveltuvat shakkimoottoreissa hyvin pelipuun tutkimiseen.

Todennäköisyysrajoitettua hakuakin useammin shakkitekoälyissä käytetään koneoppimisalgoritmia arviointifunktion tuottamiseen. Arviointifunktion tehtävä on arvioida kuinka hyvä annettu pelitilanne on esimerkiksi sen jälkeen kun pelipuuta ollaan kuljettu määriteltyyn maksimisyvyyteen. Koneoppimisella tuotetun arvointifunktion edut piilevät sen sovellettavuudessa erilaisiin tilanteisiin. Tämä on myös osaltaan heikkous, sillä yleistetty funktio on hitaampi, kuin tarkkaan rajattuihin tapauksiin vuosikausien aikana hiotut perinteiset arviointifunktiot.

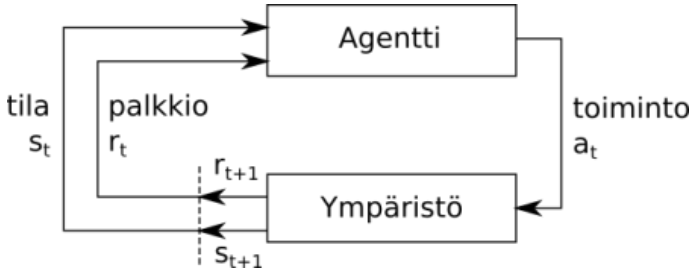
Vahvistusoppimisessa pyritään oppimaan mitä tekemällä tietyissä tilanteissa voidaan maksimoida tulevista tiloista saatava palkkio, kokeilemalla ja tutkimalla ympäristöä. Algoritmille ei kerrota mitä sen tulee tehdä, vaan sen tulee löytää kuhunkin tilanteeseen sopiva paras toiminto itse. Toiminnon tehtyään se ei saa tietoa lopullisesta palkinnosta heti ja valittu toiminto saattaa vaikuttaa myös tuleviin tiloihin. Vahvistusoppimisen keskeisimpiä piirteitä onkin yritys ja erehdys -tyyppinen ympäristön tutkiminen, sekä viivästynyt palkkio.

Keskeinen ongelma vahvistusoppimisessa on opitun tiedon hyödyntämisen ja tuntemattoman tutkimisen välinen suhde [Kun13]. Saadakseen maksimaalisen palkinnon, oppijan, eli agentin, tulee suosia toimintoja jotka johtavat sen kokemuksen perusteella parhaaseen palkkioon. Kuitenkin, löytääkseen parhaan palkinnon, sen tulee tutkia vaihtoehtoja joita se ei ole aiemmin kokeillut. Agentin tulee siis pystyä tutkimaan erilaisia vaihtoehtoja, mutta suosia niitä joista se saa korkeimman palkinnon. Koska ympäristö, jossa agentti toimii, on dynaaminen, sen tulee kokeilla vaihtoehtoja useita kertoja saadakseen luotettavan arvion niiden hyvyydestä.

Vahvistusoppiminen on hyvin lähellä sitä miten ihminen itsekin oppii [PDM11]. Esimerkiksi pelatessaan shakkia, ihmispelaaja arvioi siirtoja sen perusteella millaista tietoa se saa pelistä sillä hetkellä. Pelaaja voi arvioida omia ja vastustajan mahdollisia siirtoja ja hänellä on myös intuitiivinen käsitys siitä, millainen siirto on hyvä. Kuitenkin vasta pelin lähestyessä loppua, selviää voittaako pelaaja pelin vai ei. Pelin lopputuloksesta riippuen hän saattaa muuttaa intuitiotaan erinäisten siirtojen hyvyydestä seuraavaa peliä varten.

4.1 Markovin päätösprosessi

Vahvistusoppiminen voidaan kuvata kahden tekijän väliseksi sykliksi, jossa päätökset tekevä agentti ja kaikkea muuta hallitseva ympäristö ovat jatkuvassa vuorovaikutuksessa keskenään [Kun13, WPM05]. Agentti valitsee jokaisessa kohtaamassaan tilassa toiminnon ja ympäristö vastaa toimintoon antamalla agentille sitä seuraavan uuden tilan ja tilasta kertyvän palkinnon. On siis olemassa kaikkien mahdollisten tilojen joukko S jossa tämän hetkinen tila on $s_t \in S$ ja t kuvaa hetkeä ajassa. Jokaiseen tilaan s_t liittyy joukko mahdollisia toimintoja siinä tilassa $A(s_t)$, joista agentti valitsee toiminnon a_t. Tämän toiminnon seurauksena agentti saa ympäristöltä palkinnon toiminnostaan $r_{t+1} \in R$ ja uuden tilan s_{t+1}, jonka jälkeen sykli alkaa alusta, kuvan 2 mukaisesti.



Kuva 2: Agentin ja ympäristön vuorovaikutus vahvistusoppimisessa.

Voidakseen valita toiminnon, agentti tarvitsee tiedon siitä, mikä toiminto johtaa parhaaseen palkkioon. Sitä varten agentilla on toimintamalli (policy) π_t [SB98], joka sisältää todennäköisyyden jolla tietty toiminto valitaan kyseisessä tilassa, niin että $\pi_t(a|s)$ on todennäköisyys jolla $a_t = a$ jos $s_t = s$. Todennäköisyyteen perustuvan toimintamallin sijasta voidaan käyttää myös kartoitusta tilasta toiminnolle. Toimintamallia muokataan oppimisen edetessä niin, että agentti täyttäisi tavoitteensa, eli keräisi mahdollisimman suuren palkkion pitkällä aikavälillä.

Shakkitekoälyä varten voidaan ajatella että tila on pelilaudan sen hetkinen tilanne ja toiminnot kussakin tilassa lailliset siirrot siitä tilasta. Palkkio on pisteytys, joka kuvaa kuinka todennäköistä on voittaa peli kyseisestä tilanteesta. Yleensä niin, että mitä suurempi pistemäärä, sitä todennäköisempi voitto ja lähellä nollaa kummalla tahansa on yhtä suuri mahdollisuus voittoon. Vastaavasti negatiiviset luvut kuvaavat suurempaa todennäköisyyttä häviöön. Toimintamalli määrittelee minkä siirron agentti tekee kussakin tilassa.

Käyttääksemme vahvistusoppimista shakkitekoälyssä, meidän tulee voida mallintaa peli tiloina, toimintoina ja palkkioina. Tätä varten käytetään tyypillisesti Markovin päätösprosessia [Kun13]. Tilan tulee sisältää kaikki tarpeellinen tieto agentille uuden toiminnon valintaa varten. Yleisessä tapauksessa ympäristö luo tilan $s_t + 1$ tilasta t toiminnon r_t jälkeen säilyttäen kaikkien aiempien tilojen toiminnot kaavalla

$$P_G(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, s_0, a_0). \quad (2)$$

Jos nykyinen tila säilyttää kaiken oleellisen tiedon aiemmista siihen johtaneista tiloista, sanotaan että tilalla on Markovin ominaisuus. Tällöin kaava 2 sievenee muotoon

$$P_M(s_{t+1} = s', r_{t+1} = r | s_t, a_t). \quad (3)$$

Toisin sanoen tilalla on Markovin ominaisuus jos $P_G = P_M$ [SB98]. Tästä seuraa, että seuraavan toiminnon valinta voidaan tehdä aivan yhtä hyvin tunnettaessa vain nykyinen tila, kuin jos tunnettaisiin myös kaikkinykyistä tilaa edeltävät tilat. Shakissa pelilaudalla, jossa kaikkien nappuloiden sijainnit on esitetty, on Markovin ominaisuus. Siten, vaikka se ei sisällä tapahtumaketjua, joka johti pelitilanteeseen, kaikki seuraavaa siirtoa varten tarvittava tieto edellisistä tiloista on tallessa. Tämä on oleellista vahvistusoppimisen kannalta, jossa päätökset tehdään yleensä nykyisen tilan perusteella.

Kun shakkipeli mallinnetaan niin, että se täyttää Markovin ominaisuuden, sitä kutsutaan Markovin päätösprosessiksi. Lisäksi koska shakissa tiloja on äärellinen määrä (peli päättyy aina jomman kumman pelaajan voittoon tai pattitilanteeseen), sanotaan, että kyseessä on äärellinen Markovin päätösprosessi. Markovin päätösprosessi muodostuu tiloista $S = \{s_1, s_2, \ldots, s_n\}$ ja toiminnoista $A = \{a_1, a_2, \ldots, a_n\}$ niin, että tilassa s toiminnolla a, todennäköisyys jokaiseen seuraavaan tilaan s' määritellään kaavalla

$$P^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s | a_t = a\}. \quad (4)$$

Näitä todennäköisyyksiä kutsutaan siirtymätodennäköisyyksiksi. Samoin jokaiselle siirtymälle ($s \xrightarrow{a} s'$) voidaan laskea palkkio kaavalla

$$R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}, \quad (5)$$

jossa E on odotusarvo. Näin ollen jokaiselle tila-toiminto parille voidaan laskea odotettu palkkio

$$R(s, a) = \sum_{s' \in S} P^a_{ss'} R^a_{ss'}. \quad (6)$$

Odotettu palkkio kertoo agentille kuinka suuri välitön palkkio sille on odotettavissa valitessaan minkä tahansa mahdollisen toiminnon nykyisessä tilassaan. Koska vahvistusoppimisessa on tavoitteena maksimoida palkkion määrä pitkällä aikavälillä, on pelkästään tämän lähestymistavan käyttämisessä merkittävä heikkous. Mikäli agentti valitsee toimintonsa pelkästään välittömän palkkion perusteella, se saattaa jättää huomiotta tilan jossa on matala palkkio, mutta joka johtaisi tiloihin joissa palkkio on suuri.

4.2 Arvofunktio

Arvofunktio V^π kuvaa palkkiota, joka tilasta on odotettavissa pitkällä aikavälillä, jos agentti seuraa toimintamallia π [Kun13]. Tätä voidaan käyttää hyödyksi kun etsitään optimaalista toimintamallia, joka ottaa pitkän aikavälin palkkion huomioon.

Arvofunktio voidaan pilkkoa kahteen osaan. Toiminnon arvo $Q^\pi(s,a)$ kuvaa odotetun palkkion, kun agentti valitsee toiminnon a tilassa s, toimintamallin π mukaisesti kaavalla

$$Q^\pi(s,a) = E_\pi\{R_t|s_t = s, a_t = a\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s, a_t = a\} \tag{7}$$

jossa $\gamma \in [0,1]$ on alennuskerroin, jonka perusteella tulevia palkkioita alennetaan niin, että mitä kauempana palkkio on, sitä vähemmän se vaikuttaa toiminnon arvoon. Silloin tilan arvo $V^\pi(s)$ saadaan kaikkien tilan s toimintojen a odotusarvoista kerrottuna toimintamallin π mukaisella todennäköisyysjakaumalla $\pi(s,a)$

$$V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s,a). \tag{8}$$

Tarkasteltaessa arvofunktiota V^π lähemmin voidaan huomata, että kaavan 8 lisäksi tilan odotettu palkkio voidaan laskea rekursiivisesti suoraan sitä seuraavan tilan päivitetystä arvosta muuttamatta sen toimintaa. Kaavaa 9 kutsutaan Bellmanin yhtälöksi,

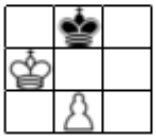
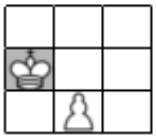
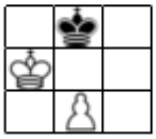
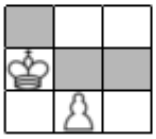
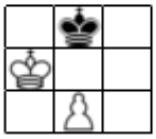
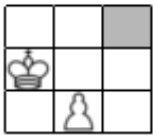
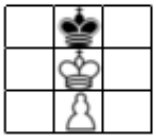
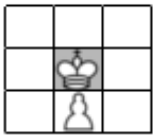
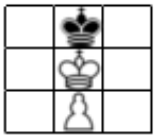
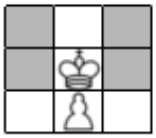
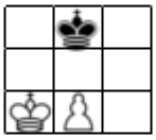
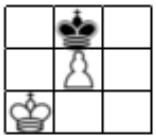
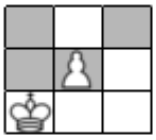
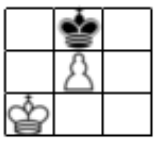
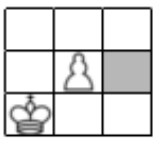
$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \tag{9}$$

jonka mukaan tilan arvo on yhtä kuin sitä seuraavan odotetun tilan palkkio ja sen alennettu arvo. Bellmanin yhtälö on johtanut useisiin tehokkaisiin dynaamisen ohjelmoinnin menetelmiin joilla Markovin päätösprosessit voidaan ratkaista [WPM05].

Optimaalinen toimintamalli π_* on se jossa kaikkien tilojen odotettu arvo on suurempi tai yhtä suuri kuin muissa toimintamalleissa. Jos optimaalinen toimintamalli on tiedossa, agentti voi yksinkertaisesti valita kaikista tilan toiminnoista ahneesti sen, jonka arvo on suurin. Tosimaailmassa optimaalista mallia kuitenkaan harvoin pystytään laskemaan, sillä tilojen määrän noustessa prosessointi- ja muistivaatimukset nousevat liian suuriksi. Näin on myös shakissa ja optimaalisen toimintamallin laskemisen sijaan on tyydyttävä likimalkaiseen arvioon siitä.

Kuvassa 3 on esimerkki Markovin päätösprosessista ja arvofunktiosta yksinkertaistetulla 3x3 shakkilaudalla. Valkoisen siirtovuorolla on mahdollista siirtää kuningasta kahteen eri ruutuun ja sotilasta ruudun ylöspäin, eli toiminnot tilasta s_t ovat $A = \{$*kuningas ylös, kuningas yläoikealle, sotilas ylös*$\}$. Vastapelaajan todennäköisyydet pelata toimintoa a_t seuraava tila s_{t+1} on esitetty taulukossa ja niissä α on jokin luku väliltä $[0,1]$. Välitön palkkio on esitetty sarakkeessa R ja arvofunktion

antama arvo toimintoa seuraaville tiloille sarakkeessa V. Taulukon avulla voidaan päätellä, että toiminto *sotilas ylös* antaa parhaan välittömän palkkion 0 ja parhaan pitkän aikavälin palkkion $\alpha \cdot 90 + (1 - \alpha) \cdot 70 > 0$, kun taas kumpi tahansa kuninkaan laillisista siirroista johtaa pelin häviöön ja siten välittömään palkkioon -100 ja pitkän aikavälin palkkioon 0.

s_t	a_t	s_{t+1}	$P(s_{t+1} \mid s_t, a_t)$	$R(s_t, a_t, s_{t+1})$	$V(s_{t+1})$
			1	−100	0
			0	0	90
			0	0	60
			1	−100	0
			0	0	90
			α	0	90
			1 - α	0	70

Kuva 3: Yksinkertaistettu esimerkki Markovin päätösprosessista arvofunktiolla, kun valkoisen vuoro kuvitteellisella 3x3 laudalla. Harmaat ruudut kuvaavat mustan pelaajan mahdollisia siirtoja.

4.3 Tilanne-ero-oppiminen

Tähän asti menestyksekkäimmät koneoppimista hyödyntävät shakkimoottorit ovat käyttäneet tilanne-ero-oppimista (temporal difference)(TD) [Lai15]. Tekniikalla pyritään ratkaisemaan vahvistusoppimisen ongelma, jossa peli pelataan loppuun saakka, jotta nähtäisiin pienten muutosten vaikutukset oppimiseen. Sen sijaan että pyrittäisiin tarkasti ennustamaan pelin lopputulos tietyillä siirroilla, pyrkii algoritmi parantamaan omaa ennustettaan pelin etenemisestä lähitulevaisuudessa. Jos tilanne eroaa ennusteesta, algoritmi korjaa arviotaan ja näin pystyy tehokkaasti oppimaan koko pelin ajalta. Tekniikka soveltuu erityisesti shakin kaltaiseen peliin, jossa peli on voitu pelata lähes täydellisesti aivan viime hetkille saakka ja kuitenkin hävitä loppumetreillä tehdyn huonon valinnan takia.

Ajatellaan tilannetta, jossa olemme arvioimassa jonkin tietyn shakin pelitilan hyvyyttä. Arvioimme voiko pelin voittaa tästä tilasta vai onko kenties todennäköisempää päätyä häviöön tai tasapeliin ja pisteytämme tilan sen mukaisesti. Oppiaksemme parantamaan arviota voimme pelata pelin loppuun, katsoa mikä pelin lopputulos oli, ja palata korjaamaan arviotamme sen perusteella. Tällä menetelmällä voimme oppia jotakin tilan hyvyydestä, mutta hävitämme kaiken tiedon siirroista, jotka tehtiin kyseisen tilan ja pelin lopputilan välillä. Tilanne-ero-oppimisessa tavoitteena on huomioida myös kaikki välissä tapahtuvat siirrot ja siinä vertaammekin arviota seuraavan tilan arvioon, jota taas verrataan sitä seuraavaan arvioon ja niin edelleen. Emme siis opi suoraan eroa pelin lopputulokseen, vaan opimme eron jokaisen peräkkäisen arvion välillä. Yhden päivityksen sijaan teemme $T - 1$ päivitystä, jos askelia tarkastelemastamme tilasta pelin loppuun on T kappaletta.

Jo aiemmin todettiin, että Bellmanin yhtälöllä (kaava 9) voidaan Markovin päätösprosessit, jollaiseksi myös shakki voidaan määritellä, ratkaista dynaamisen ohjelmoinnin keinoin. Tällöin tulee kuitenkin tuntea ympäristöstä muodostuva malli [Kun13], joka shakin tapauksessa ei pelipuun koon vuoksi ole mahdollista. Tilanne-ero-oppimisella voidaan dynaamisen ohjelmoinnin menetelmiä käyttää ilman, että malli tunnetaan etukäteen, kuten edellisessä esimerkissä huomattiin.

Tilanne-ero-oppimisessa pyrimme oppimaan arvofunktion arvion V_θ^π mahdollisimman lähelle todellista arvoa V^π. Virhe arvion ja todellisen välillä voidaan laskea esimerkiksi käyttäen jäännösvarianssia (MSE)

$$MSE(\theta) = \frac{1}{n}\sum_{i=1}^{n}(V_\theta^\pi(s_i) - V^\pi(s_i))^2, \tag{10}$$

jossa θ kuvaa arviota todellisesta arvosta ja lopputulos on summa todellisten ja arvioitujen arvojen erotuksista.

Pyrkimällä pienentämään arvion ja todellisen arvon välistä eroa, pääsemme mahdollisimman lähelle meille ennestään tuntematonta todellista arvofunktiota. Koska tuntemattomassa mallissa $V^\pi(s)$ on meille tuntematon, saamme arvion siitä käyttämällä kaavaa 9 tämän hetkiselle arviolle V_θ^π

$$V^{\pi}(s_t) \approx E\{r_t + \gamma V_{\theta}^{\pi}(s_{t+1})\}. \tag{11}$$

Shakin (ja monien muiden vastaavien pelien) tapauksessa on huomattu, että on kannattavaa päivittää tilanne-eroa ainoastaan kun ero on negatiivinen [BBT+13]. Tämä johtuu siitä, että siirryttäessä tilasta s_t tilaan s_{t+1}, molemmat pelaaja ja vastustaja ovat tehneet siirtonsa. Mikäli tilanne-ero on positiivinen, tarkoittaa tämä, että pelaajan tilanne on parantunut. Tilanteen parantuminen voi johtua vastustajan tekemästä virheestä ja missään tilanteessa ei ole järkevää opettaa agenttia luottamaan siihen, että vastapelaajan toistaisi virheensä. Ainoastaan jos ero on negatiivinen, voidaan varmasti sanoa sen johtuvan virheestä arviossa.

Käyttämällä tilanne-ero-oppimista onnistuttiin selkeästi osoittamaan vahvistusoppimisen vahvuus nollasummapeleissä, kun TD-Gammon hävisi vain yhden pelin 40:stä yhtä maailman sen hetken vahvimmista pelaajista Bill Robertieta vastaan vuonna 1995 [Tes95]. Vastaavat menetelmät eivät kuitenkaan osoittautuneet sellaisinaan riittäviksi siirryttäessä backgammonista monimutkaisempiin peleihin kuten shakkiin. Pelien välillä on merkittäviä eroja kuten se, että shakissa yksikin siirto voi vaikuttaa pelitilan arvioon huomattavasti, kun taas backgammonissa yksittäisen siirron vaikutus on parhaimmillaankin pieni. Shakissa ei pärjää suunnittelemalla hyvin yksittäisiä siirtoja vaan on kiinnitettävä enemmän huomiota pidemmän aikavälin taktiikkaan. Siksi shakkia varten kehitettiin versio tilanne-ero-oppimisesta, joka nimettiin TDLeaf(λ):ksi [BTW99].

TDLeaf(λ) on tilanne-ero-oppimisen ja minimax haun yhdistelmä ja se soveltuu shakkiin erinomaisesti. Sen ideana on laskea tilanne-ero juurisolmun sijasta minimax haun tuottamalle parhaalle lehtisolmulle. Näin voidaan huomioida myös pitkän aikavälin taktiikka aina etsintäsyvyyteen saakka.

4.4 Neuroverkot

Suuri osa koneoppimisessa käytettävistä malleista soveltuu hyvin yhden tai kahden asteen funktioiden oppimiseen. Tämä tarkoittaa sitä, että kovin hierarkkisen tiedon oppiminen on hankalaa. Hierarkkisen tiedon oppiminen on kuitenkin hyödyllistä shakin kaltaisissa peleissä, joissa pelinappulat vaikuttavat toisiinsa, ja mikäli hierarkiaa ei voitaisi hyödyntää, opittava malli kasvaisi hyvin suureksi [Lai15]. Jos esimerkiksi pyrkisimme tallentamaan arvofunktion yksinkertaisesti taulukkona, voidaan helposti kuvitella taulukon koon paisuvan hyvin nopeasti toteuttamiskelvottomiin mittasuhteisiin. Suuren mallin ongelmana on sen vaatiman laskentatehon määrän lisäksi riittävän laajan opetusjoukon hankkiminen mallin ylisovittamisen välttämiseksi, sekä aika, joka mallin oppimiseen kuluu.

Arvojen taulukoinnin sijaan haluammekin käyttää funktioiden likiarvoistamista. Likiarvoistamisessa on sekin hyvä puoli, että shakin kaltaisessa pelissä jossa erilaisia tiloja on valtava määrä, olisi epärealistista ajatella että voisimme oppia niistä jokaisen, etenkin kun jokaisen tilan luotettavaan oppimiseen tarvitaan useita toistoja. Funktion likiarvoistamisella voimme arvioida ennestään opitun perusteella kuinka

hyvä on jokin tila, jossa emme ole koskaan ennen käyneet [WPM05]. Vahvistusoppimisen yhteydessä on mielekkäintä käyttää funktion likiarvoistajana eteenpäin kytkettyä neuroverkkoa.

Eteenpäin kytketty neuroverkko koostuu kerroksiin jaetuista neuroneista, joista kaikki edeltävän kerroksen neuronit on yleensä kytketty kaikkiin seuraavan kerroksen neuroneihin. Yksittäisen neuronin aktivointi riippuu sitä edeltävän tason aktivoinneista ja vaikuttaa sitä seuraavien tasojen aktivointeihin.

Mitä neuroverkko pyrkii siis oppimaan, on likiarvoistus siitä, millaisella arvofunktiolla saadaan arvot joihin tilanne-ero-oppimisella ollaan päädytty, jotta arvofunktioa voitaisiin yleistää toimimaan myös tiloissa, joihin agentti ei koskaan oppiessaan törmännyt. Kyseessä on siis ohjattu oppiminen jossa opetusdata syöte on tilat, joissa agentti on vieraillut ja agentin oppiman arvofunktion arvot näille syötteille on luokittelu, jota algoritmilta toivotaan.

Monitasoisista neuroverkoista koostuvaa koneoppimista, joka kykenee hierarkkisten rakenteiden oppimiseen, kutsutaan syväoppimiseksi [Ben09]. Syväoppimista on käytetty onnistuneesti monilla tekoälyn osa-alueilla viime vuosina, kuten luonnollisen kielen käsittelyssä [MH09] ja tietokonepelien pelaamiseen [MKS+13]. Syväoppimisella on saavutettu jopa ihmisen suoritusta parempia tuloksia mm. konenäön saralla [CMS12].

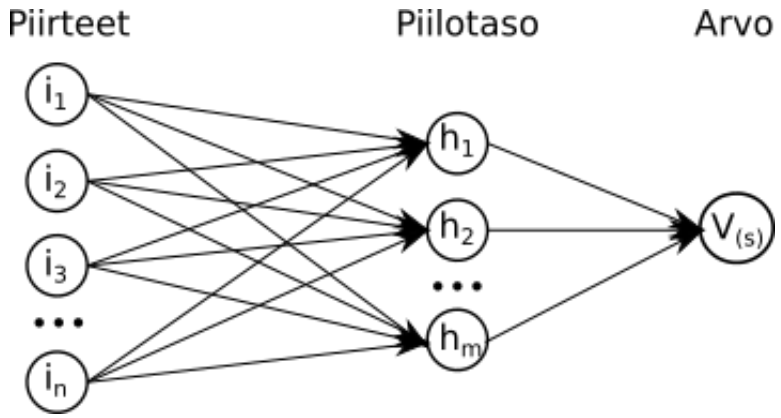
Jo aiemmin huomattiin, että shakissa pelinappuloiden sijainnit sisältävällä pelilaudalla on Markovin ominaisuus ja siten se voidaan kuvata vahvistusoppimisen tilana. Siten yksinkertaisin syötevektori neuroverkkoa hyödyntävälle shakkitekoälylle voisi olla vektori, jossa jokainen vektorin indeksi kuvaa yhtä pelinappulaa ja sen arvo pelinappulan sijaintia laudalla. On kuitenkin hyödyllistä antaa syötteenä myös muita esilaskettuja arvoja, jotta neuroverkko pystyy helpommin tunnistamaan samankaltaisia pelitiloja [Lai15]. Näitä voi olla esimerkiksi linnoitusmahdollisuudet, eri tyyppisten pelinappuloiden määrästä johdettu aineellinen arvo, liukuvien pelinappuloiden kantamat ja hyökkäys- ja puolustuskartat.

Kuvan 4 mukaisen kaksitasoisen neuroverkon neuroverkkoarkkitehtuuriin kuuluu syötevektori $I = \{i_1, i_2, \ldots, i_n\}$ jossa n on piirteiden lukumäärä, sekä piilotaso $H = \{h_1, h_2, \ldots, h_m\}$. $V(s)$ on ulostuloarvo, joka kuvaa arviota tilan hyvyydestä. Lisäksi on painovektorit w_{ih} syötevektorin ja piilotason välisille painoille ja w_{hv} piilotason ja ulostuloarvon välisille painoille.

Useimmiten piilotasojen aktivointifunktiona käytetään sigmoid-funktiota, kun taas ulostulo lasketaan lineaarisella aktivointifunktiolla [WPM05]. Näin ollen piilotason neuronien $H_h \in H$ aktivoinnit voidaan laskea kaavalla

$$H_h = \sigma(\sum_{i=1}^{n} w_{ih} I_i) \tag{12}$$

jossa $\sigma(x)$ on sigmoid-funktio $\sigma(x) = \frac{1}{1+e^x}$ ja ulostuloarvo saadaan laskemalla



Kuva 4: Yhdellä piilotasolla varustettu eteenpäin kytkevä neuroverkko joka tuottaa arvofunktion arvon tilalle.

$$V(s) = \sum_{h=1}^{m} w_{ho} H_h. \tag{13}$$

Myös muita aktivointifunktioita voidaan toki käyttää, mutta ainakin nämä ovat shakin tapauksessa hyväksi havaittuja.

Systeemin opettamiseen voidaan käyttää vastavirta (backpropagation) algoritmia, jonka tavoitteena on minimoida jäännösvarianssi tuotetun arvon ja halutun arvon (agentin oppiman) välillä. Vastavirta-algoritmi palaa eron havaitessaan päivittämään painovektoreita w_{ih} ja w_{hv} niin että jäännösvarianssi olisi mahdollisimman pieni jokaisella opetusdatan syötteellä.

Jäännösvarianssi voidaan minimoida laskemalla ensin delta-arvot lineaariselle aktivointifunktiolle

$$\delta_V(s) = V(s_\pi) - V(s_\theta) \tag{14}$$

jossa $V(s_\pi)$ on agentin oppima haluttu arvo ja $V(s_\theta)$ arvioitu arvo. Sen jälkeen lasketaan piilotason kaikkien neuronien delta-arvot

$$\delta_H(h) = \sum_{s} \delta_V(s) w_{hv} H_h (1 - H_h). \tag{15}$$

Nyt voidaan päivittää piilotason ja ulostuloarvon välinen painovektori $w_{hv} = w_{hv} + \alpha\delta_V(s)Hh$ ja syötevektorin ja piilotason välinen painovektori $w_{ih} = w_{ih} + \alpha\delta_H(h)I_i$, kun $\alpha \in [0, 1]$ on käytössä oleva oppimissuhde. Oppimissuhde vaikuttaa siihen paljonko yhdestä päivityksestä opitaan ja on hyödyllistä toteuttaa niin, että sen arvo laskee opittujen pelien lukumäärän noustessa [BBT+13]. Näin aluksi voidaan oppia paljon ja pelikertojen lisääntyessä oppiminen muuttuu hienovaraisemmaksi.

4.5 Opetusdatan muodostaminen

Tähän mennessä käsitellyillä tekniikoilla voidaan toteuttaa shakkimoottori, joka oppii vahvistusoppimista hyödyntäen tuottamaan tilan arviointifunktion. Näiden tekniikoiden lisäksi shakkimoottorin suorituksen kannalta on merkityksellistä millaista dataa moottorin opettamiseen käytetään.

Jo aiemmin puhuttiin hiukan siitä, minkälaista opetusdata on muodoltaan. Sen tulee täyttää Markovin ominaisuus, joka shakin tapauksessa yksinkertaisimmillaan tarkoittaa jokaisen pelinappulan sijaintia pelilaudalla. Lisäksi on hyödyllistä liittää dataan myös muita ennalta laskettuja piirteitä, kuten hyökkäys- ja puolustuskartat, jotta systeemin on helpompi oppia tunnistamaan samankaltaisia pelitiloja. Kuitenkin, mitä enemmän piirteitä on, sitä hitaampaa myös oppiminen on ja sitä suurempi opetusjoukko tarvitaan ylisovittamisen välttämiseksi.

Koneoppimisalgoritmille annettujen piirteiden lisäksi on tärkeää valita sopiva opetusjoukko, eli pelit joilla systeemi opetetaan. Tämä ei ole suoraviivainen tehtävä, sillä opetusjoukon pitäisi olla jakaumaltaan sellainen, joka kuvaa hyvin todellisia vastaan tulevia pelitiloja. Silloin moottori on parhaiten valmistautunut juuri niihin tiloihin, joita se todennäköisemmin pelissä kohtaa, sen sijaan että laskentatehoa olisi tuhlattu sellaisten pelien oppimiseen, joita ei lähes koskaan tule todellisuudessa vastaan, kuten vaikkapa pelejä joissa toisella osapuolella on lähes kaikki pelinappulat ja toisella vain muutamia. Käytännössä vain hyvin pieni osa kaikista mahdollisista shakin pelitiloista on opetuksen kannalta kiinnostavia [Thr95]. Toisin kuin todennäköisyysrajoitetun haun tapauksessa ja osittain ristiriitaisesti edellä kuvatun suhteen, on todellisten pelien jakaumaa hyvin kuvaavien pelitilojen lisäksi systeemin lisäksi opittava arvioimaan jonkin verran tiloja, joita todellisessa pelissä harvoin tulee vastaan. Tämä johtuu siitä, että se saattaa törmätä näihin tiloihin arvioidessaan syvyyshaun tuottamia lehtisolmuja, tai oppiessaan todennäköisyysrajoitetun haun todennäköisyyksien arviointia.

Lisäksi opetettavia pelitiloja on oltava suuri määrä. Mitä enemmän, sen parempi. Tämä tarkoittaa sitä, että tilojen tulee olla jollakin tavoin generoituja, eikä sopivaa jakaumaa voi valita käsin. Jonkinlaista valikointia pelitiloille on kuitenkin hyödyllistä toteuttaa, sillä sen lisäksi, että täysin vapaasti pelaava systeemi voisi tuhlata aikaa oppiakseen epäoleellisia tiloja, on vaikeaa estää tilanne jossa se päätyy valitsemaan siirtonsa aina noudattaen liian samanlaista kaavaa. Tällöin vaihtelevien pelitilanteiden oppiminen hidastuu.

Ongelmaa on pyritty ratkaisemaan monin keinoin. Esimerkiksi Falcon shakkimoottoria [DTKN08] varten kerättiin 10 000 suurmestaritason peliä ja jokaisesta pelistä valittiin satunnaisesti yksi pelitila. Puolet peleistä sijoitettiin opetusjoukkoon ja puolet testijoukkoon. Vaihtelevuuden lisäämiseksi jokaista pelitilaa muutettiin niin, että oli valkoisen siirtovuoro, joka johti suurempaan vaihteluun pelitilojen materiaalitasapainossa.

NeuroChessin [Thr95] opetusdata kerättiin myös korkealuokkaisten pelien tietokannasta satunnaisesti. Sen annettiin sitten pelata pelit loppuun alkaen näistä tiloista,

jotta se törmäisi oppiessaan vaihteleviin tiloihin. Myös KnightCap [BTW99] oppi pelaamalla, mutta ennalta valittujen tilojen sijaan se pelasi oikeita pelaajia vastaan internetissä.

Giraffe shakkimoottorin [Lai15] kanssa käytettiin edellä kuvattuja ratkaisuja yhdistelevää lähestymistapaa. Sen opetusdatan muodostamista varten kerättiin todellisten pelattujen pelien tietokannasta satunnaisesti 5 miljoonaa eri pelitilaa. Näille pelitiloille sovellettiin sen jälkeen satunnaisia laillisia siirtoja, luoden 175:n miljoonan pelin lopullinen opetusjoukko. Alkuperäiseen tilaan lisätyn satunnaisen siirron ansiosta pelitilat kuvasivat hyvin todellisia pelitiloja mutta sisälsivät normaalia enemmän vaihtelua. Lopuksi Giraffen annettiin pelata pelit loppuun aloittaen näistä pelitiloista.

Todellisten pelitilojen tietokantoja ja itse pelaamista voidaan käyttää hyödyksi, vaikka näin tuotettu data ei olekaan merkittyä, koska opetusmetodi hyödyntää vahvistusoppimista ohjatun oppimisen sijaan. Näin suuren opetusjoukon käyttäminen ei ole työlästä, koska systeemi oppii tilojen hyvyyden ilman ihmisen panosta.

5 Käytännön toteutuksia

Aikaisimmat koneoppimista hyödyntävät shakkitekoälyt ovat yleensä keskittyneet pelkän loppupelin ratkaisemiseen [Hau05]. Loppupelissä mahdollisia siirtoja on vähemmän, mutta toisaalta pelinappulat ovat yleensä vapaampia liikkumaan laudalla, joten vaikka ongelma onkin helpompi kuin koko pelin pelaaminen, on mahdollisten siirtojen määrä silti liian suuri koko pelipuun rakentamiseen. Yleisesti käytetty ja hyvin toimiva lähestymistapa on ollut geneettinen ohjelmointi, mutta on huomattu etteivät ohjelmat skaalaudu hyvin kun samoja menetelmiä on sovellettu kokonaisen pelin pelaamiseen.

NeuroChess Tilanne-ero-oppimiseen ja neuroverkkoon perustuva NeuroChess [Thr95] pystyi 2000-luvun alulla voittamaan joitakin satoja pelejä GNU Chess moottoria vastaan, jota pidettiin hyvänä esimerkkinä koneoppivien shakkitekoälyjen kehityksestä. Se oppi ainoastaan itse pelaamalla, eikä käyttänyt valmiita pelejä oppimiseen kuten useat myöhemmin tulleet moottorit. Sen jälkeen on havaittu, että pelkästään itseään vastaan pelaaminen on hidas tapa oppia ja harvoin johtaa hyviin tuloksiin [BBT+13]. NeuroChessin kehittänyt Sebastian Thrun epäröi kokeilunsa perusteella, ettei tilanne-ero-oppiminen soveltuisi niin monimutkaisiin peleihin kuin shakki, vaikka hyviä kokemuksia oltiinkin saatu esimerkiksi backgammonista.

KnightCap Läpimurto koneoppivien shakkimoottorien saralla oli KnightCap [BTW99], joka oppi internetissä pelaamiensa pelien avulla. Sen toiminta perustui neuroverkkoon ja TD oppimisen muunnelmaan TD-Leaf algoritmiin. Se aloitti esitäytetyillä arvoilla Elo-luvulla 1650 ja pelattuaan vain kolme päivää nousi yli 2150 pisteeseen. KnightCapin ongelma oli hitaus, joka johtui muun muassa sen käyttämästä suu-

resta lukumäärästä piirteitä. Se käytti lähes 6000:ta erilaista piirrettä arviointiin. Moottori on avoimen lähdekoodin projekti ja siihen on myöhemmin lisätty tietoutta aloitusliikkeistä, jonka ansiosta sen sen Elo-luku on noussut noin 2400:n paikkeille.

FUSc# Vuonna 2013 kehitetyn FUSc#:n [BBT+13] vahvuusluku nousi 2016:een pelattuaan vain 119 peliä. Se käytti myös TD-Leaf algoritmia ja vaikka se ei yltänyt parhaiden shakkimoottorien tasolle, sillä pystyttiin todentamaan että TD-Leaf arviointifunktion opettamiseen toimii shakkitekoälyillä hyvin. Se oli myös merkittävästi KnightCap:ia nopeampi tilojen arvioinnissa, sillä se käytti arviointiin paljon pienempää määrää piirteitä.

Giraffe Matthew Lain kehittämä Giraffe [Lai15] on uudempi vuonna 2015 kehitetty täysin geneeriseksi rakennettu shakkimoottori. Siinä on ratkaistu monia FUSc#:ssa esiintyneitä ongelmia. Shakkimoottorien ja ihmispelaajien tasoa arvioidaan yleisesti FIDE- luokituksella. Giraffe yltää normaalilla kotikoneella toimiessaan FIDE:n kansainvälisen mestarin tasolle, eli tasolle, jolla pelaavat parhaat 2.2% pelaajista. Sen Elo-luku on noin 2500. Se suoriutuu erityisen hyvin Strategic Test Suite (STS) -testissä, jonka tarkoituksena on arvioida shakkimoottorien tilannetajua. Erityisen kiinnostavia tuloksista tekee se, että niihin on päästy sen jälkeen, kun Giraffe on opetellut peliä vain 72:n tunnin ajan. Giraffe yltää STS testissä samalle tasolle parhaiden shakkimoottorien kanssa ja näiden arviointifunktioita on tyypillisesti hiottu vuosien ajan. Giraffe käyttää TD-Leaf tekniikkaa ja kolmitasoista syvää neuroverkkoa oppimiseen.

Giraffessa erityistä oli myös sitä varten kehitetty syvyysrajoitteisen haun ja alphabeta karsinnan korvaava todennäköisyysrajoitettu haku. Neuroverkkoon perustuva todennäköisyysrajoitettu haku paransi Giraffen vahvuusluokitusta 3000 pelatun pelin jälkeen 48 ± 12 Elo-pisteellä [Lai15], kun peliaikaa ei oltu rajoitettu. Ero on merkittävä, sillä 200 pisteen ero Elo-luvussa vastaa samaa kuin jos vahvempi pelaaja olisi kolme kertaa heikompaa pelaajaa parempi. Neuroverkon avulla toteutettu haku on kuitenkin perinteisiä hakualgoritmeja hitaampi, sillä lähellä pelipuun lehtisolmuja se saattaa tuhlata enemmän aikaa tilojen todennäköisyyksien määrittelyyn kuin varsinaiseen haarojen etsintään. Tästä syystä Giraffessa päätettiin käyttää todennäköisyysrajoitettua hakua vain kun etsittävien haarojen syvyys oli vähintään 100 solmua. Silloin parannus Elo-pisteissä oli noin 20 ± 12. Todennäköisyysrajoitetun haun vaikutus vahvuuslukuun kertoo siitä että syvyysrajoitetun haun ja siihen liittyvien karsintasääntöjen aiheuttamia ongelmia on pystytty sen avulla lieventämään.

6 Yhteenveto

Shakki on hyvin vanha ja arvostettu peli, jota on pidetty suosittuna erityisesti älykköjen keskuudessa. Ehkä siksi shakkitekoäly on kiinnostanut tutkijoita jo siitä läh-

tien kun sellaisen toteuttaminen on ollut edes teoriassa mahdollista. Tässä tutkielmassa on esitelty useita perinteisiä ja koneoppimista hyödyntäviä shakkimoottoreita. Lisäksi on tutustuttu komponentteihin, joista koneoppivat shakkitekoälyt usein muodostuvat. Samoja komponentteja voidaan hyödyntää myös muihin tekoälyn ja koneoppimisen aihepiireihin.

On osoitettu, että vahvistusoppiminen ja sen alatekniikka tilanne-ero-oppiminen, yhdessä neuroverkkojen kanssa soveltuu shakkimoottorien perustaksi hyvin. Näiden lisäksi käytetään usein samoja tekniikoita kuin perinteisissä shakkitekoälyissä; minimax hakua, tilojen arviointifunktioita ja pelipuun tutkittavien haarojen karsintaa. Uudempi lähestymistapa on todennäköisyysrajoitettu haku, joka on edellä mainittujen hakutekniikoiden koneoppimisen avulla tuotettu yleistys.

Yleisin koneoppimisen sovelluskohde on arviointifunktion toteuttaminen ja todennäköisyysrajoitettu haku on koneoppivien shakkitekoälyjen tutkimuksen uusimpia tuotoksia. Molemmissa tekniikoissa etuna on saatujen funktioiden yleispätevyys ja soveltuvuus erilaisiin tilanteisiin. Esimerkiksi niitä voisi vain pieniä muutoksia tekemällä käyttää minkä tahansa shakin kaltaisen nollasummapelin pelaamiseen. Ne ovat myös vähemmän alttiita inhimilliselle virheelle, kuin käsin koodatut arviointi- ja karsintasäännöt. Haittapuolena on funktioiden hitaus, joka pienentää mahdollisten etsittävien tilojen joukkoa.

Koneoppivat shakkitekoälyt eivät ole vielä yltäneet aivan parhaiden perinteisten shakkimoottorien tasolle. Siitä huolimatta, että esimerkiksi Giraffe tutkii saman määrän tiloja perinteisen shakkimoottorin kanssa noin 10 kertaa hitaammin, on se pelityyliltään perinteisiä shakkimoottorieita inhimillisempi tunnistaessaan pelitilanteita ja pelaa erityisen hyvin pelin aloitus- ja lopetusvaiheissa. Siten sen heikompi suorituskyky ei merkittävästi haittaa verrattaessa perinteisiin shakkimoottoreihin, sillä se pääsee lähes vastaaviin tuloksiin käyttäessään saman ajan arviointiin, vaikka arvioitujen tilojen lukumäärä on pienempi. On lisäksi huomattava, että kehitys koneoppimisen saralla on ollut hyvin nopeaa, varsinkin syvien neuroverkkojen käyttöönoton jälkeen. Vasta aivan viime päivinä on esimerkiksi nähty kuinka vahvistusoppimiseen ja syvään neuroverkkoon perustuva AlphaGo [SHM+16] on pystynyt voittamaan turnauksen hallitsevaa maailmanmestaria vastaan Go:ssa, jossa on vielä merkittävästi shakkiakin suurempi pelipuu.

Lähteet

Bau78 Baudet, G. M., An analysis of the full alpha-beta pruning algorithm. *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, New York, NY, USA, ACM, sivut 296–313, URL http://doi.acm.org/10.1145/800133.804359, 1978.

BBT+13 Block, M., Bader, M., Tapia, E., Ramírez, M. et al., Using reinforcement learning in chess engines, Bsc/MSc Seminar Artificial Intelligence and Machine Learning, URL http://www.ismll.uni-hildesheim.de/lehre/semKIML-13w/script/chess.pdf, 2013.

Ben09 Bengio, Y., Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, sivut 1–127. Julkaistu lisäksi kirjana. Now Publishers, 2009.

Bot10 Bottou, L. *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, luku Large-Scale Machine Learning with Stochastic Gradient Descent, sivut 177–186. Physica-Verlag HD, URL http://dx.doi.org/10.1007/978-3-7908-2604-3_16, 2010.

BSB+16 Banks, G., Smith, C., Banks, R., Szots, G. ja Russell, N., Computer chess rating lists, URL http://www.computerchess.org.uk/, 2016.

BTW99 Baxter, J., Tridgell, A. ja Weaver, L., Knightcap: A chess program that learns by combining td(lambda) with game-tree search. *CoRR*, url http://arxiv.org/abs/cs.LG/9901002, 1999.

CM83 Campbell, M. S. ja Marsland, T. A., A comparison of minimax tree searc algorithms. *Artificial Intelligence*, sivut 347–367, 1983.

CMS12 Cireşan, D., Meier, U. ja Schmidhuber, J., Multi-column deep neural networks for image classification. *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, sivut 3642–3649, 2012.

Cop04 Copeland, B., The essential turing. Clarendon Press, sivut 569–575, 2004.

DTKN08 David-Tabibi, O., Koppel, M. ja Netanyahu, N. S., Genetic algorithms for mentor-assisted evaluation function optimization. *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, New York, NY, USA, ACM, sivut 1469–1476, URL http://doi.acm.org/10.1145/1389095.1389382, 2008.

Gop13 Gopnik, A., A point of view: Chess and 18th century artificial intelligence. *BBCNewsMagazine*, URL http://www.bbc.com/news/magazine-21876120, 2013.

Hau05 Hauptman, A., Gp-endchess: Using genetic programming to evolve chess endgame players. *In: Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*. Springer, sivut 120–131, 2005.

Hsu99 Hsu, F.-H., Ibm's deep blue chess grandmaster chips. *Micro, IEEE*, sivut 70–81, 1999.

Hya77 Hyatt, R. M., Blitz v, a computer chess program. *Proceedings of the 15th Annual Southeast Regional Conference*, ACM-SE 15, New York, NY, USA, ACM, sivut 328–342, URL http://doi.acm.org/10.1145/1795396.1795442, 1977.

Kun13 Kunz, F., An introduction to temporal difference learning. URL http://www.ias.informatik.tu-darmstadt.de/uploads/Teaching/AutonomousLearningSystems/Kunz_ALS_2013.pdf, 2013.

Lai15 Lai, M., Giraffe: Using deep reinforcement learning to play chess. *CoRR*. url http://arxiv.org/abs/1509.01549, 2015.

MH09 Mnih, A. ja Hinton, G. E., A scalable hierarchical distributed language model. Teoksessa *Advances in Neural Information Processing Systems 21*, Koller, D., Schuurmans, D., Bengio, Y. ja Bottou, L., toimittajat, Curran Associates, Inc., sivut 1081–1088, URL http://papers.nips.cc/paper/3583--a--scalable--hierarchical--distributed--language--model.pdf, 2009.

MKS+13 Mnih, V., Kavukcuoglu, K., Silver, D. et al., Playing atari with deep reinforcement learning. *CoRR*. url http://arxiv.org/abs/1312.5602, 2013.

MS10 Mitsuta, T. ja Schmitt, L. M., Optimizing the performance of gnuchess with a genetic algorithm. *Proceedings of the 13th International Conference on Humans and Computers*, HC '10, Fukushima-ken, Japan, Japan, University of Aizu Press, sivut 124–131, URL http://dl.acm.org/citation.cfm?id=1994486.1994517, 2010.

PDM11 Potjans, W., Diesmann, M. ja Morrison, A., An imperfect dopaminergic error signal can drive temporal-difference learning. *PLoS Comput Biol*, sivut 1–20, URL http://dx.doi.org/10.1371%2Fjournal.pcbi.1001133, 2011.

RCK16 Romstad, T., Costalba, M. ja Kiiski, J., Stockfish, URL https://stockfishchess.org/, 2016.

SB98 Sutton, R. S. ja Barto, A. G., *Reinforcement Learning: An Introduction*. MIT Press. URL http://www.cs.ualberta.ca/\%7Esutton/book/ebook/the-book.html, 1998.

SHM+16 Silver, D., Huang, A., Maddison, C. J. et al., Mastering the game of go with deep neural networks and tree search. *Nature*, sivut 484–503, URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html, 2016.

Tes95 Tesauro, G., Temporal difference learning and td-gammon. *Commun. ACM*, sivut 58–68, URL http://doi.acm.org/10.1145/203330.203343, 1995.

Thr95 Thrun, S., Learning to play the game of chess. *Advances in Neural Information Processing Systems 7*. The MIT Press, sivut 1069–1076. 1995.

Wor14 World Chess Federation, Fide title regulations, URL http://www.fide.com/component/handbook/?id=174&view=article, 2014.

WPM05 Wiering, M., Patist, J. P. ja Mannen, H., Learning to play board games using temporal difference methods. Tekninen raportti UU-CS-2005-048, Department of Information and Computing Sciences, Utrecht University. 2005.