Tekoälymenetelmät go-lautapelissä

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
TkK-tutkielma
toukokuu 2018
Rami Ilo

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
RAMI ILO: Tekoälymenetelmät go-lautapelissä
Kandidaatintutkielma, 22 sivua
Tietotekniikka
toukokuu 2018

Tämä tutkielma tarkastelee, kuinka tietokonepelaajan on mahdollista pelata aasialaista go-lautapeliä. Tutkielmaa alustetaan esittelemällä perustietoa pelipuista, joita käytetään kuvaamaan täydellisen informaation pelien tiloja. Pelipuiden osalta keskitytään erityisesti Monte Carlo -hakumenetelmän käyttöön. Tämän jälkeen tarkastellaan syviä neuroverkkoja, joita voidaan käyttää erilaisten oppimista vaativien ongelmien ratkaisuun. Lopuksi tarkastellaan go-peliä sekä AlphaGo-sovellusta, joka käyttää Monte Carlo -puuhakua ja syviä neuroverkkoja go-pelipuun ratkaisemiseen.

Asiasanat
Monte Carlo, pelipuu, go, AlphaGo, neuroverkot

Sisällysluettelo

1 Johdanto 1
1.1 Täydellisen informaation pelit 1
2 Monte Carlo -puuhaku 3
2.1 Pelipuista 3
2.2 Minimax 3
2.3 Monte Carlo 6
2.3.1 UCT-Monte Carlo 8
2.4 Sovelluksia 9
3 Syvät neuroverkot 10
3.1 Neuroverkot 10
3.2 Syvät neuroverkot 12
4 Go 13
4.1 Gon säännöt 13
4.2 AlphaGo 15
5 Yhteenveto 20
6 Lähdeluettelo 21

1 Johdanto

Viimeaikainen tekoälyn kehitys on mahdollistanut tekoälyn käyttömahdollisuuksien laajenemisen ja sen käyttämisen esimerkiksi pelien pelaamiseen. Tietokone-pelaajan on mahdollista pelata tehokkaasti täydellisen informaation pelejä, kuten ristinollaa, käyttämällä erilaisia menetelmiä ja hakupuita pelitilojen seuraamiseen. Peleissä, joissa on suuri määrä mahdollisia pelitiloja (esim. go-lautapeli ja shakki), ei voida kuitenkaan käyttää triviaaleimpia algoritmeja liian suurten kustannusten vuoksi, vaan näitä varten täytyy kehittää erikoistuneita menetelmiä. [1]

Tämän tutkielman on tarkoitus tarkastella tietokonepelaajan gossa käyttämiä menetelmiä, sekä Google Deepmindin kehittämää go-peliä pelaavaa ohjelmistoa, AlphaGota.

Tässä luvussa alustetaan aihetta kertomalla täydellisen informaation peleistä. Luvussa 2 esittellään pelipuut sekä algoritmeja, joita voidaan käyttää tekoälypelaajan päätöksenteon sekä oppimisen perustana. Luvussa 3 esittellään lyhyesti neuroverkkojen toimintaa. AlphaGo käyttää useampia syviä neuroverkkoja. Luvussa 4 selitetään gon säännöt sekä kuinka AlphaGo käyttää lukujen 2 ja 3 metodeita gon pelaamiseen.

1.1 Täydellisen informaation pelit

Täydellisen informaation pelit ovat pelejä, joissa pelaaja voi aina muodostaa täydellisen kuvan pelitilanteesta ja seuraavista mahdollisista tiloista. Tämä deterministisyys tekee niistä sopivia pelejä tietokonepelaajan ratkaistavaksi. Esimerkiksi pokeri ja Yatzy ovat puolestaan epätäydellisen informaation pelejä, sillä niissä on joko piilotettu tietoa pelaajalta tai peli sisältää satunnaiselementin [2]. Kahden pelaajan täydellisen informaation peleissä pelaajat saavat vuorotellen vuoroja, ja heitä kutsutaan vakiintuneen tavan mukaisesti tässäkin tutkielmassa *mustaksi* ja *valkoiseksi*.

Pelin säännöt määrittelevät lopputilat, joissa jokaisessa peli päättyy. Jokaiseen lopputilaan sisältyy palkkio jonka *musta* pelaaja saa, mikäli peli päättyy kyseiseen tilaan. Peleissä ei ole välillisiä palkkioita, eli vain lopputilat sisältävät palkkion.

Mustan pelaajan tavoite on saada lopussa mahdollisimman korkea pistesaalis, kun taas *valkoinen* pelaaja pyrkii minimoimaan *mustan* saaman palkkion. [3]

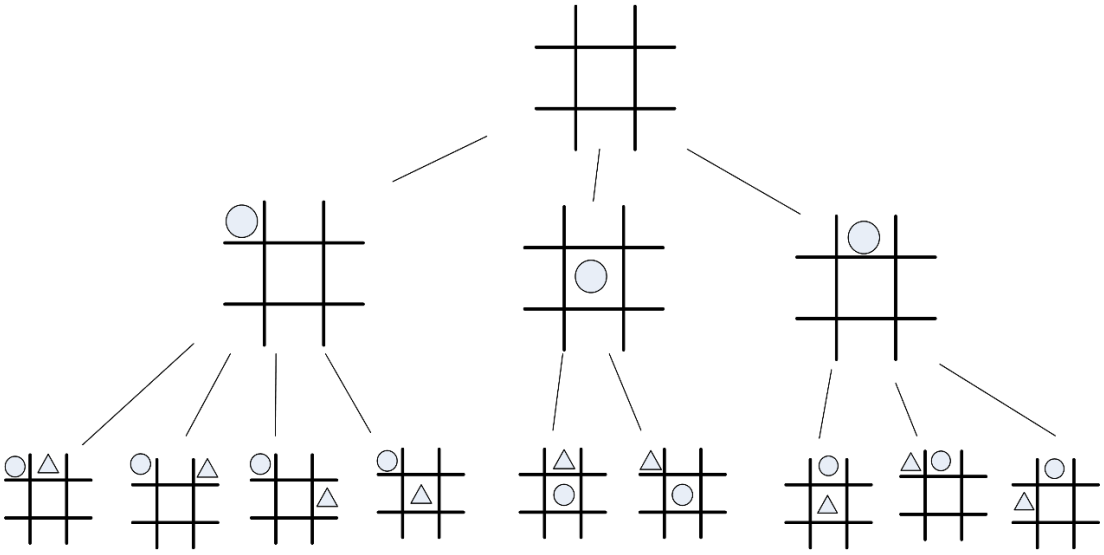
Tämän tutkielman tarkastelema go-lautapeli on kahden pelaajan täydellisen informaation peli [2].

2 Monte Carlo -puuhaku

Monte Carlo -puuhaku esiteltiin ensimmäisen kerran vuonna 2006. Kaikkien nykyisten, tehokkaimpien tietokone-go-ohjelmien toteutus perustuu jonkinlaiseen Monte Carlo -puuhakuun. [3]

2.1 Pelipuista

Pelipuuhun voidaan koota kahden pelaajan täydellisen informaation pelin kaikki tilat. Puun solmut sisältävät jonkin pelin tilan, ja solmujen väliset viivat vastaavat siirtoja. Peli alkaa puun juurisolmusta ja kaikki muut solmut vastaavat jotakin äärellistä liikesarjaa. Jokainen lehtisolmu vastaa pelin päättymistä eri tilanteisiin, joten kulkemalla juurisolmusta lehtisolmuun saadaan aina käytyä läpi yksi peli. [3]



Kuva 1: Esimerkki osittaisesta ristinollan pelipuusta, johon on koottu osa mahdollisista pelitiloista (ei siis kaikkia). Risti on korvattu kolmiolla. [4]

2.2 Minimax

Minimax on yksinkertainen, mutta tietyissä rajoissa tehokas tapa käydä läpi pelipuuta. Siinä *musta* ja *valkoinen* pelaaja on nimetty paremmin kuvaavin termein MAX ja MIN. MAX pyrkii valitsemaan siirron, joka johtaa parhaaseen lopputulokseen omalta kannaltaan. MAX pyrkii aina voittoon mikäli mahdollista, ja tasapeliin jos voitto ei ole saavutettavissa. MIN:n tehtävänä on minimoida MAX:n saavuttama etu. Pelipuun solmut sisältävät numeerisia arvoja, joiden perusteella MAX pyrkii mahdollisimman suureen pistesaaliiseen ja MIN mahdollisimman pieneen.

Optimaalinen arvo pelipuussa on paras lopputulos, jonka tietyssä solmussa oleva pelaaja voi varmasti saavuttaa olettaen, että vastapelaaja pelaa parasta mahdollista vastastrategiaa. Näiden arvojen merkitsemistä puun tiloihin kutsutaan optimaaliseksi arvofunktioksi, ja solmusta olevan siirron optimaalinen toiminta-arvon on optimaalinen arvo kyseisen solmun lapsisolmussa.

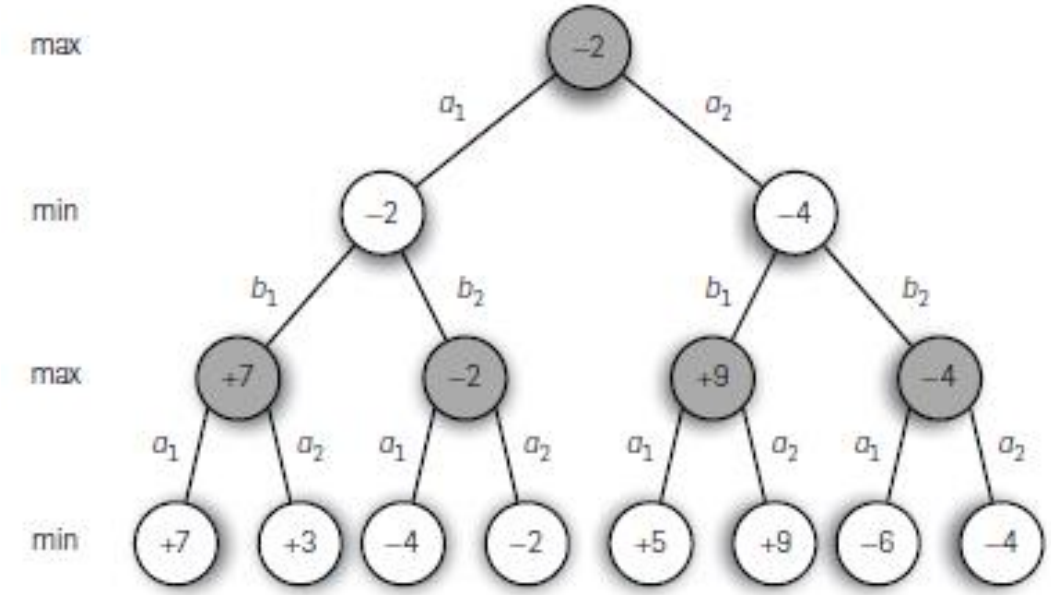
Jos kaikkien solmun lapsien arvot ovat tiedossa, on triviaalia valita optimaalinen siirto vanhemmasta solmusta: MAX valitsee siirron, jolla on suurin arvo, ja MIN pienimmän arvon. Jos puu on äärellinen, jokaisen solmun optimaalinen arvo voidaan laskea siirtymällä puussa alhaalta ylöspäin minimax-haulla. [2; 3]
Seuraavassa minimax-haun pseudokoodi [2]:

```
MINIMAX(S)
  syöte: solmu S
  ulostulo: solmun S optimaalinen arvo
 1  IF lapsi(s) = ∅ THEN
 2      RETURN arvo(S) /* S on lehtisolmu */
 3  ELSE IF nimi(S) = MIN THEN
 4      k ← +∞
 5      FOR ALL i ∈ lapsi(S) DO
 6          k ← min{k, MINIMAX(i)}
 7      END FOR
 8      RETURN k /* S on MIN-solmu */
 9  ELSE
10      k ← −∞
11      FOR ALL i ∈ lapsi(s) DO
12          k ← max{k, MINIMAX(i)}
13      END FOR
14      RETURN k /* S on MAX-solmu */
15  END IF
```

Vaikka minimax-haku johtaa optimaalisiin siirtoihin, se muuttuu erittäin kömpelöksi pelin monimutkaisuuden kasvaessa. Oletettaessa, että jokaisella ei-lehtisolmulla on sama määrä lapsia eli sama haarautumiskerroin, voidaan laskea minimax-haun aikakompleksisuus. Koska aikakustannus on verrannollinen läpikäytyihin solmuihin, saadaan [2]

$$1 + b + b^2 + \cdots + b^d = \frac{1 - b^{d+1}}{1 - b} = \frac{b^{d+1} - 1}{b - 1} = O(b^d),$$

jossa *b* on haarautumiskerroin ja *d* on hakupuun syvyys. Haun aikakustannus kasvaa siis eksponentiaalisesti puun syvyyden kasvaessa, mikä on erittäin epätoivottavaa varsinkin tarkastelemamme go-pelin kompleksisuus huomioon ottaen.



Kuva 2: Esimerkki minimax-hausta [3].

Negamax on minimax-hausta tehty variantti, joka poistaa tarpeen luokitella solmut MAX- ja MIN-arvoihin ottamalla käänteisarvon MIN-solmuista, jolloin niitä voidaan käsitellä samalla tavoin kuin MAX-solmuja. Tämä ei kuitenkaan muuta haun aikakompleksisuutta.

Minimax-hakua voidaan nopeuttaa pienentämällä pelipuun syvyyttä *d*, jolloin lehtisolmuille ei lasketakaan minimax-haulla optimaalista arvoa, vaan ne korvataan heuristisen evaluointifunktion antamilla arvioilla. Mikäli evaluointifunktio on tarpeeksi laadukas, arvion voi olettaa olevan lähes optimaalinen.

Toinen vaihtoehto nopeuttaa minimax-hakua on karsia pelipuuta käyttäen alfa-beta –karsintaa. Näitä metodeja ei kuitenkaan tarkastella tässä tutkielmassa tämän enempää, sillä nekin sopivat tarkoitukseen huonosti. [2; 3]

2.3 Monte Carlo

Useimmat perinteiset pelipuualgoritmit perustuvat johonkin evaluointifunktioon, jolla määritellään optimaalinen arvo pelipuun solmuille. Evaluointifunktiot ovat yleensä heuristisia, joten niiden tarkkuus on välttämätöntä.

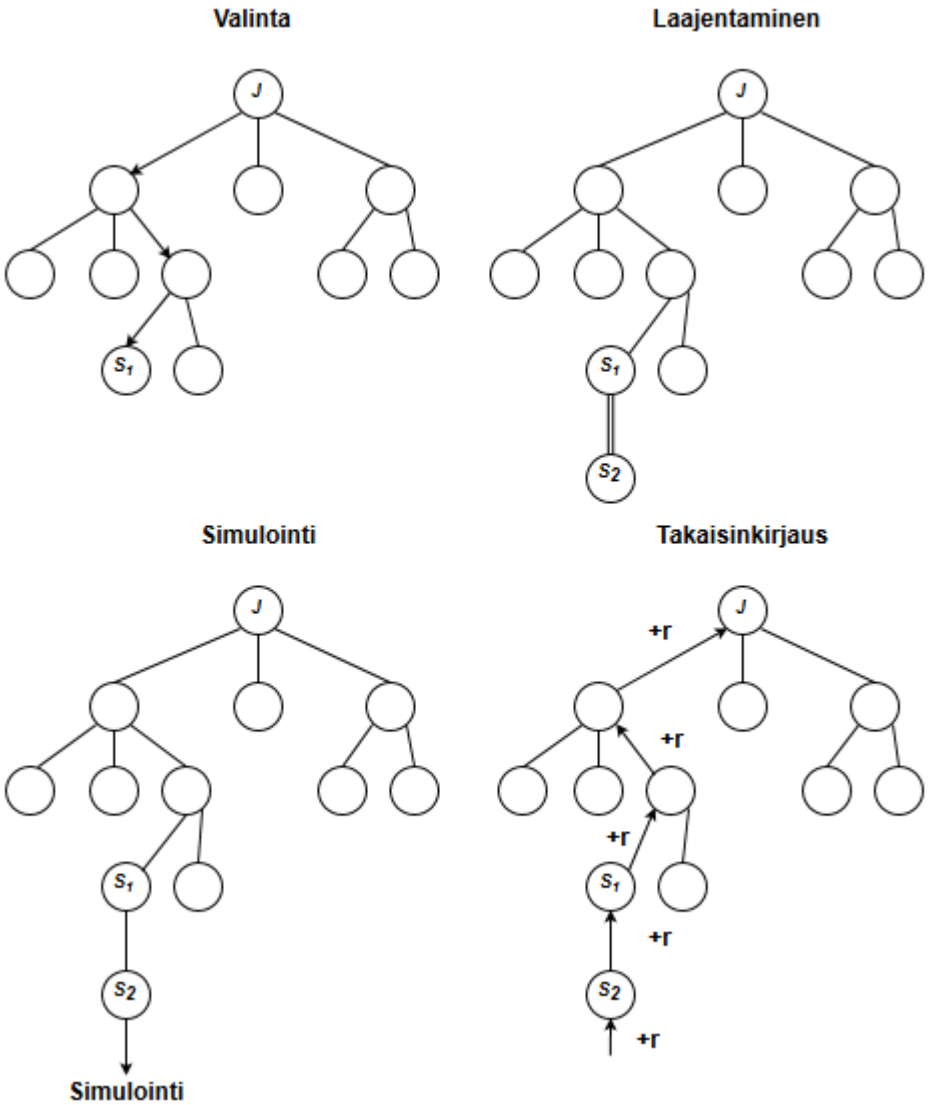
Satunnaistamisella ja tilastoimisella voidaan poistaa tarve tarkasti määritellylle evaluointifunktiolle. Evaluointifunktiosta voidaan saada arvio toistamalla esimerkkipelejä ja keräämällä niiden lopputuloksista tilastotietoa tietyn aikaa. Pelipuuta käytetään tiedon kirjaamiseen siitä, mitä siirtoja on jo koetettu sekä tallentamaan siirtojen lasketut arvot ja niiden reitit. Solmut, joista ei ole vielä koetettu yhtäkään siirtoa, asettautuvat puun etusijalle (osapuun lehdiksi), ja näin muodostuu osapuu, joka vastaa täydellistä, kaikki siirrot käsittävää pelipuuta. Toistamalla näytteenottoa arviot tulevat tarkemmiksi ja menneiden pelien perusteella joitakin, hyviin tuloksiin johtaneita solmuja aletaan painottaa, jolloin osittainen pelipuu muodostuu usein epäsymmetriseksi. Näytteenoton tärkein tarkoitus on erotella huonot siirrot hyödyllisistä.

Monte Carlo -puuhaku käsittää useamman eri algoritmin. Pääosin niiden toiminta voidaan jaotella seuraaviin vaiheisiin:

1. **Valinta.** Aloita juurisolmusta *J* ja laskeudu solmuun S_1, josta ei ole koetettu jokaista mahdollista siirtoa ja joka ei mielellään ole koko pelipuun lehtisolmu (jolloin seuraavia vaiheita ei voisi tehdä).
2. **Laajentaminen.** Valitse S_1:n vielä koettamatta oleva siirto, ja luo tästä siirrosta uusi lapsi, S_2, S_1:lle. S_2 on aina osapuun (ei koko pelipuun) lehtisolmu, sillä siitä ei ole vielä koetettu yhtäkään siirtoa.
   Siirron valinta voi olla satunnainen tai se voi noudattaa jotakin tiettyä ehtoa.
3. **Simulointi.** Suorita peliä alkaen S_2:sta kunnes peli päättyy. Määritä palkintoarvo *r* pelin loputtua. Simuloinnin pelaamisosuus käydään pelipuun ulkopuolella jollakin ohjelmistolla, joka esimerkiksi valitsee satunnaisia siirtoja tai käyttää jotakin tiettyä määriteltyä valintalogiikkaa.

Vaiheen tarkoitus on arvioida S_2:sta seuraavia pelin lopputuloksia, joten siitä voidaan tehdä useita pelisimulointeja ja niiden pohjalta muodostaa päätelmä kyseisen solmun optimaalisuudesta. Vaiheen 4. johdosta *J*:tä lähimpinä oleville solmuille tämä tapahtuu automaattisesti lehtisolmujen määrän kasvaessa.

4. **Takaisinkirjaus.** Päivitä kaikki (vaiheissa 1 ja 2 läpikäydyn) *J*:n ja S_2:n välisen reitin solmut sisältämään palkintoarvo *r*. Seuraavat iteroinnit (vaiheiden 1-4 läpikäynti) hyötyvät nyt tästä tiedosta, ja voivat ottaa huomioon valinnoissaan (elleivät ole satunnaistettuja) solmuista löytyvät palkintoarvot. Pelaajat ovat samoin kuin minimax-haussa MAX ja MIN, joten MAX pyrkii valitsemaan mahdollisimman suuret arvot ja MIN vastaavasti mahdollisimman pienet.
5. **Lopetus.** Jatka iterointia kunnes jokin tietty ehto täyttyy, esimerkiksi asetettu aika- tai muistiraja. Myös vaiheessa 1 iterointi voi loppua, mikäli valitaan jokin pelin päättävä solmu. [2; 3]



Kuva 3: Monte Carlo -puuhaun vaiheet 1-4.

Yleisesti Monte Carlo -puuhaun aikakompleksisuus on [5]:

$$O\left(\frac{mkI}{C}\right),$$

jossa *m* on yksittäisessä haussa laajennettavien lapsisolmujen määrä, *k* on rinnakkaisten hakujen määrä, *I* on suoritettavien iteraatioiden määrä ja *C* on laskentaan käytettävissä olevien prosessoriytimien määrä.

2.3.1 UCT-Monte Carlo

UCT (eng. *Upper confidence bounds on trees*) on yksi yleisimmin käytetyistä Monte Carlo -puuhaun varianteista. On erittäin toivottavaa, että pelipuuhaku valitsisi parhaita siirtoja juurisolmusta eli olisi johdonmukainen. Tähän tarvitsemme UCT:tä. Jos kaikki pelipuun solmujen palkintoarvojen arviot todella olisivat tarkkoja, olisi parhaimpien siirtojen valinta helppoa valitsemalla suurin tai pienin arvo (MAX, MIN). Näin ei kuitenkaan ole, sillä satunnaistamisen johdosta arviot sisältävät virheitä minkä lisäksi ne eivät sisällä kaikkia siirtoja tai painottavat liikaa jotakin siirtoa. Tämän johdosta johdonmukainen valinta voi johtaa epäoptimaaliseen pelaamiseen, joten parhailta näyttävien siirtojen teko ei ole aina kannattavaa.

Ilmentämään ongelmaa, valitako optimaalinen tai epäoptimaalinen siirto, on esitetty nk. monikätinen pelikoneongelma (eng. *multi-armed bandit problem*, jossa one-armed bandit on pelikone). Jokainen tällainen peli päättyy yhden siirron jälkeen, ja pelaaja ansaitsee palkkion, joka riippuu vain valitusta siirrosta. Pelikoneet luonnollisesti vaativat jonkin panoksen, jotta niitä voi pelata. Tavoite on maksimoida pelaajan saama palkkio, ts. löytää paras mahdollinen siirto mahdollisimman nopeasti niin, että pelaajan palkkiota kuluu panoksina tämän löytämiseen mahdollisimman vähän.

Eräs ratkaisu monikätiseen pelikoneongelmaan on UCB1 [6], jonka keskeisiä ominaisuuksia on se, ettei se muuta solmujen muodostamien osajoukkojen koostumusta, vaan jatkaa pelipuun läpikäyntiä. Tämän johdosta se kykenee löytämään optimaaliset siirrot tarpeeksi monen iteraation jälkeen. UCT on tehty UCB1:n pohjalta, ja myös sillä on kyky löytää optimaaliset siirrot. Sen eräs

ominaisuus on vähän tarkasteltujen solmujen suosiminen, jolloin sen osapuu muodostuu suuremmaksi eikä suosi vain parhaalta näyttäviä siirtoja.

Monte Carlo -puuhaun ensimmäinen vaihe, valinta, etsii jonkin solmun S_1, jolloin saadaan lisää tietoa juurisolmun J siirroista. Solmusta siirtyminen alaspäin puussa johonkin lapsisolmuun S_L vaatii jonkinlaista päätöksentekoa. UCT-algoritmi käsittelee tätä ongelmaa monikätisen pelikoneongelman lailla, suurimman eron ollessa siirtokohtaisen todennäköisyysjakauman D_i indeksin i muuttuminen takaisinkirjaus-vaiheessa (Monte Carlo -puuhaun vaihe 4). [2; 3]

Kun simulaatioiden määrä kasvaa äärettömäksi, UCT:n optimaalinen arvofunktio lähestyy minimaxin vastaavaa (eli optimaalista eikä arvioitua). Lisäksi juurisolmuissa tehtävät, väärään tietoon perustuvat arviot ja todennäköisyys valita ei-optimaalinen siirto muodostuvat nollaksi. Tämän vuoksi UCT-Monte Carlo-algoritmi sopii hyvin go-peliin. [7]

2.4 Sovelluksia

Monte Carlo -puuhakua käytetään eri variantein tässä tutkielmassa tarkasteltavan go-lautapelin lisäksi monissa sovelluksissa, varsinkin peleissä. Esimerkkejä näistä ovat *Amazons*, *Breaktrough* ja *Havannah* [8], shakki ja shougi [9] sekä The Creative Assemblyn kehittämä tietokonepeli *Total War: Rome II* [10]. Google Deepmindin myös kehittämä AlphaZero, joka on AlphaGon yleistetty versio, sai esitiedoiksi pelkästään pelin säännöt eikä lainkaan esimerkkipelejä, mutta se suoriutui silti hyvin ja voitti aiemmat tietokoneohjelmat shakissa, shougissa ja gossa. Huomionarvoista on, että kyseisiin go-tietokonevastustajiin lukeutui AlphaGo Zero, AlphaGon paranneltu versio. [9]

3 Syvät neuroverkot

Tietokoneneuroverkot ovat nykyään laajasti käytettyjä järjestelmiä, jotka toimivat ihmisaivojen tavoin. AlphaGo käyttää kolmea syvää konvoluutista neuroverkkoa oppimisen ja päätöksentekonsa perustana. Syvät neuroverkot kykenevät monikerroksisuutensa ansiosta tekemään erittäin erikoistuneita ja tarkkoja laskutoimituksia.

3.1 Neuroverkot

Neuroverkko on kokoelma yhteen liitettyjä laskentayksiköitä, neuroneita, jotka yhdessä muodostavat hyvinkin erilaisiin käyttötarkoituksiin soveltuvan tiedonkäsittelyverkoston. Neuroverkkojen toiminta ja rakenne pohjautuu löyhästi ihmisaivoihin, vaikka ihmisen aivojen neuronien määrä onkin huomattavasti suurempi ja rakenne merkittävästi monimutkaisempi. [11] Neuroverkkoja voidaan käyttää monenlaisiin tarkoituksiin, kuten kasvojen ja ajoneuvojen tunnistukseen, luokitteluun tai pelien pelaamiseen.

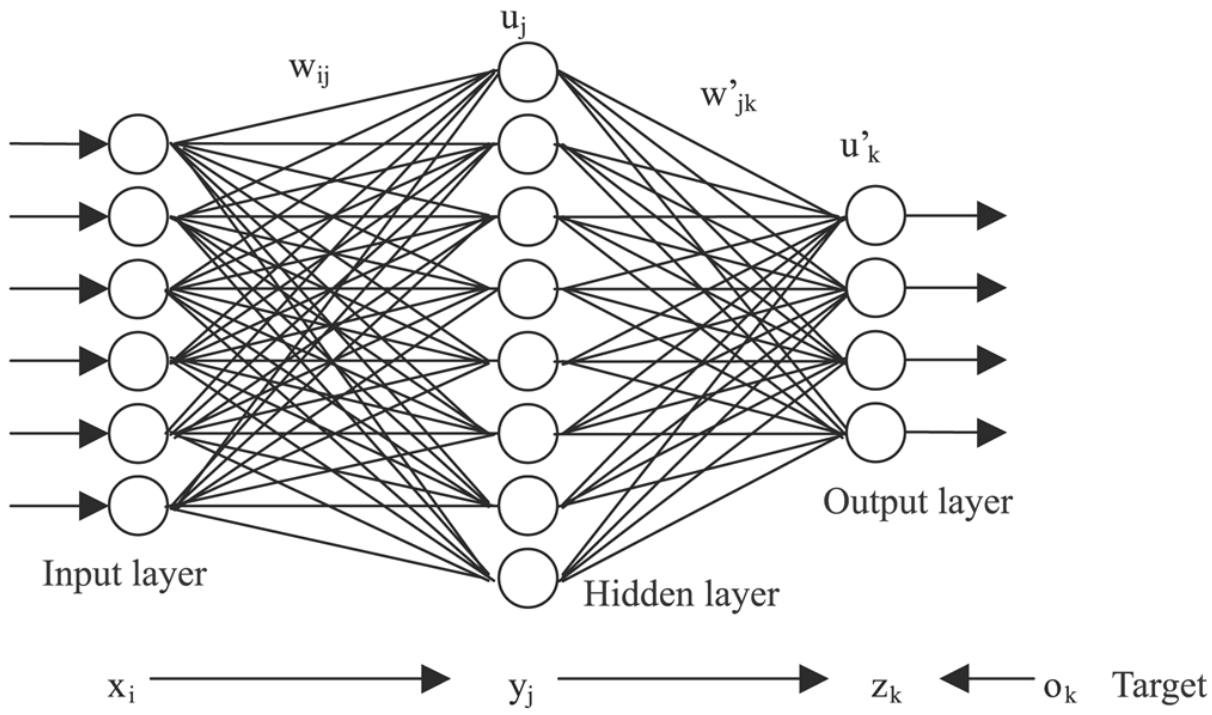
Neuronin signaalinkäsittelyn prosessi voidaan esittää seuraavassa muodossa [11]:

$$\mathbf{y}(\mathbf{x}) = \boldsymbol{\Phi}\left(\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{n}} \boldsymbol{w_i x_i}\right),$$

jossa y on ulostulosignaali, $\Phi()$ on aktivointifunktio, x_i:t ovat sisääntulomuuttujia ja w_i:t ovat sisääntulomuuttujia vastaavia painotusarvoja. Jokaisella neuronilla on joukko sisääntulomuuttujia ja niihin liitettyjä painotusarvoja. Painotusarvoja muuttamalla voidaan muokata neuroverkon toimintaa, ja neuroverkon oppiessa niitä voidaan säädellä eri tarkoituksiin pääsemiseksi.

Aktivointifunktio laskee summatuista sisääntuloista ja näihin yhdistetyistä painotuksista arvon, ja sen tarkoitus on saada eri arvoista keskenään vertailukelpoisia puristamalla ne jollekin tietylle välille. [11; 12] Ilman aktivointifunktiota jokainen neuroni voisi saada arvoja väliltä $[-\infty, \infty]$. Esimerkiksi mustavalkoisen kuvan tunnistuksessa aktivointifunktio voi antaa neuronille arvon väliltä [0, 1], jossa 0 vastaa täysin mustaa pikseliä ja 1 täysin valkoista pikseliä.

Neuroverkossa neuronit jaotellaan tasoihin, ja tason jokainen neuroni on yhteydessä vierekkäisten tasojen jokaiseen neuroniin (ks. kuva 4). Ensimmäinen taso koostuu sisääntuloja lukevista neuroneista ja viimeinen taso ulostuloneuroneista. Väliin jääviä tasoja kutsutaan piilotetuiksi tasoiksi, ja niiden määrä vaihtelee neuroverkon tarkoituksesta ja tehtävän monimutkaisuudesta riippuen. Neuroverkko voi olla asyklinen tai syklinen. Asyklisessä neuroverkossa on yksinkertaisempi rakenne, jossa signaalit kulkevat vain yhteen suuntaan eli sisääntuloneuroneista mahdollisten piilotettujen kerrosten kautta ulostuloneuroneille. Syklisessä rakenteessa signaalit voivat sen sijaan kulkea myös ”takaisinpäin”, joka mahdollistaa monia ominaisuuksia, kuten neuroverkon tasojen keskustelun toistensa kanssa ennen iteroinnin päättymistä, jolloin painotusarvoihin voidaan tehdä haluttuja muutoksia aiemmin. [12; 13]



Kuva 4: Neuroverkon rakenne. [12]

Neuroverkkoa voidaan opettaa; kun neuroverkko on saanut jonkin tuloksen, lopputulosta verrataan odotettuun lopputulokseen. Odotetun ja saadun tuloksen välisestä erotuksesta saadaan nyt virhe, jonka minimoimiseksi neuronien painotusarvoja *w* muutetaan odotettuun tulokseen pääsemiseksi. Jotta neuroverkolla olisi jotakin mihin verrata saamiaan tuloksia, täytyy sille syöttää jonkinlaista dataa halutunlaisista vertailukohdista oppimisen mahdollistamiseksi, kuten numeerisia arvoja tai kuvia. [11]

Konvoluutiset neuroverkot ovat neuroverkkoja, jotka ovat erikoistuneet kuvantunnistukseen. Ne tekevät oletuksen, että sisääntulona on visuaalinen kuva, jolloin käsittelyä voidaan tehostaa. [14] Niitä käytetään erityisesti kohteiden, kuten kasvojen tai autokameran näkemien objektien tunnistamiseen ja luokitteluun.

3.2 Syvät neuroverkot

Syvät neuroverkot eroavat tavallisista neuroverkoista niiden piilotettujen kerrosten määrän osalta. Sitä, montako piilotettua kerrosta neuroverkolla tulee olla, ei ole tarkkaan määritelty. Yleisesti neuroverkkoja, joilla on kaksi tai useampi piilotettu kerros, pidetään tällä hetkellä kuitenkin syvinä. Yhden piilotetun kerroksen neuroverkkoja taas sanotaan mataliksi (eng. shallow). [15] Käsitteen ollessa liukuva voi syvän neuroverkon vaatima piilotettujen kerrosten määrä kuitenkin kasvaa ajan myötä.

Monikerroksisuutensa ansiosta syvät neuroverkot voivat tehdä kompleksista luokittelua vaativia tehtäviä, sillä jokaisen tason kohdalla voidaan tunnistaa datasta enemmän tekijöitä ja näin luoda jälkimmäisille kerroksille paremmat edellytykset erikoistua johonkin tiettyyn osa-alueeseen. [15] Tietyt syvät neuroverkot kykenevät oppimaan itsenäisesti ja tehokkaammin kuin perinteiset neuroverkot [13].

Esimerkiksi käsinkirjoitettujen numeroiden tunnistamisessa syvät neuroverkot hyötyvät syvyydestään. Koska syötenumeroissa voi olla suurtakin variaatiota mallinumeroihin verrattuna, tarvitaan useita eri vaiheita. Numero voi olla eri kokoinen ja eri kulmassa kuin perinteinen numero, eikä se välttämättä vastaa juuri mitenkään mitään numeroita erottelevia luonteenpiirteitä kuten suljettua tilaa 4, 6, 8, 9 ja 0 tapauksessa. Tämän takia syvän neuroverkon täytyy lähteä tunnistuksissaan pienistä, toistuvista kuvioista ja niiden suhteesta toisiinsa. Opetusdatasta on tässä työssä valtava hyöty, sillä saamalla esimerkkejä käsinkirjoitetuista numeroista tietyt ominaisuudet kuitenkin yleensä tulevat esille.

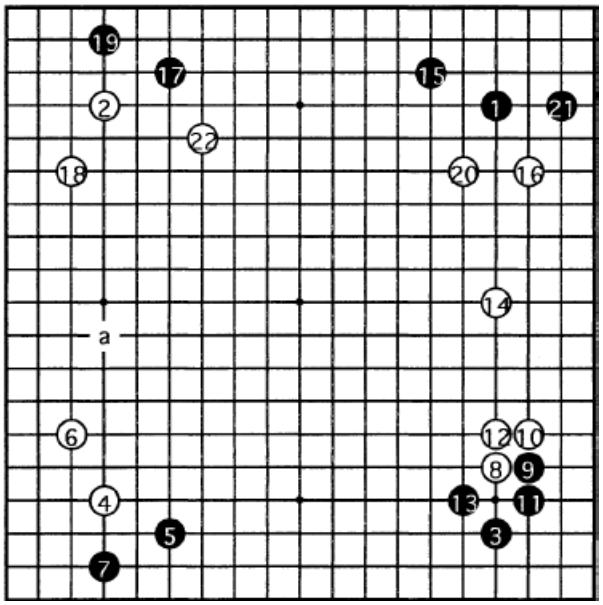
4 Go

Go on vanha aasialainen kahden pelaajan lautapeli. Sitä on pitkään pidetty suurena haasteena tekoälylle, eikä tietokonepelaajan ollut odotettu voittavan dan 9-tason ihmispelaajaa vielä pitkään aikaan. [7] Gon säännöt ovat yksinkertaiset, mutta pelin kompleksisuus sekä siirtojen kauaskantoisuus tekevät siitä loistavan pelin testata tekoälypelaajan kyvykkyyttä. Gon kombinatorinen monimutkaisuus on valtava. Jokaisella vuorolla on keskimäärin 200 mahdollista siirtoa, siinä missä shakissa niitä on vain 37. Lisäksi go-pelin normaali kesto on 300 siirtoa, shakin ollessa 57. Gossa on 10^{170} mahdollista pelitilaa ja shakissa 10^{47}.

Siirtojen määrän lisäksi ongelmaksi muodostuu myös niiden kauaskantoisuus; pelin alussa asetettu kivi voi vaikuttaa suuresti pelin loppuosaan, satoja siirtoja myöhemmin. Yksinkertaisten, heurististen evaluaatioiden laskeminen pelitiloista, jotka tuottavat haluttuja tuloksia esimerkiksi shakissa, eivät ole riittäviä gossa. [3]

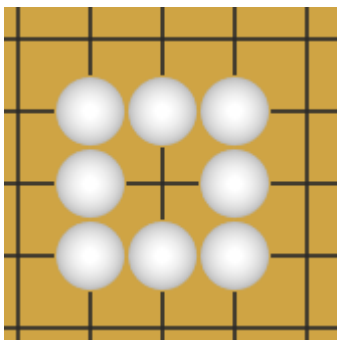
4.1 Gon säännöt

Gossa toinen pelaaja käyttää valkoisia ja toinen mustia kiviä, joita he sijoittavat vuorotellen pelilaudan viivojen tyhjiin risteymäkohtiin. Kiviä ei voi liikuttaa niiden asettamisen jälkeen, mutta ne voidaan vangita vastustajan toimesta. Pelin tarkoitus on hallita suurempaa määrää viivojen risteymäkohtia kuin vastustaja sekä vangita vastustajan kiviä omilla kivillä. Samanväriset kiviryhmät muodostavat *ryhmiä* jos ne ovat yhdistyneet toisiinsa risteymäkohtien viivoin. Jos kivellä tai kiviryhmällä ei ole yhtäkään *vapautta,* se poistuu pelistä. Vapaus määritetään kiveä tai kiviryhmää ympäröivien risteysten perusteella; kivellä tai ryhmällä on oltava jonkin osasensa vieressä ainakin yksi tyhjä risteymäkohta. [16]



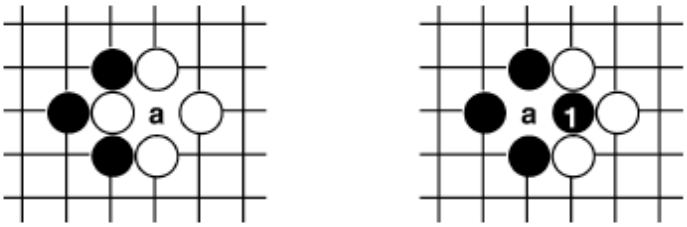
Kuva 5: Tyypillinen aloitusvaihe gossa, siirrot numeroitu aikajärjestyksen mukaan. Ryhmiä ovat {9, 11} ja {8, 12, 10}. [16]

Jos yhdenväriset kivet pitävät sisällään yhtä tyhjää risteystä, tätä muodostelmaa kutsutaan silmäksi (ks. kuva 6). Mikäli silmän muodostavalla ryhmällä on tämän lisäksi ainakin yksi vapaus, ei vastustaja voi asettaa kiveä silmän keskelle. Käyttämällä hyväksi tätä seikkaa voi pelaaja muodostaa ryhmiä, joilla on ainakin kaksi silmää, jolloin vastustaja ei voi milloinkaan saada kiveä näiden silmien sisälle ja vangita pelaajaa.



Kuva 6: Silmä [17].

Edellisen siirron toistaminen välittömästi on kiellettyä. Tätä sääntöä kutsutaan *ko*:ksi. Muuten peli voisi jumittua tilanteeseen, jossa pelaajat vangitsevat vuorotellen joitakin tiettyjä, samoja kiviä (ks. kuva 7).



Kuva 7: Esimerkki tilanteesta joka olisi mahdollinen ilman *ko*-sääntöä. Oikealta luettuna järjestyksessä musta ja valkoinen voisivat jatkaa ikuisesti kiven asettamista paikkaan a. [16]

Go-peli päättyy, kun kumpikin pelaaja ohittaa vuoronsa. Pelin loputtua pelaajien hallitsemat risteyskohdat ja heidän kiviryhmiensä vapaudet käydään lävitse, ja pelipisteitä jaetaan hallituista risteyksistä ja vangituista vastustajan kivistä. Musta pelaaja aloittaa aina pelin, ja valkoinen pelaaja saa tästä hyvityksen, *komin*, joka lisätään hänen lopullisiin pelipisteisiinsä. [7]

Pelilaudan vakiokoko on 19 x 19-ruutua, kokojen 13 x 13 ja 9 x 9 ollessa suositumpia variantteja lähinnä nopeisiin peleihin. Ihmispelaajien tasot käyvät aloittelijan 30-kyusta amatöörien 20-kyuhun ja 1-kyuhun, mestaritason pelaajien 1-danista 7-daniin sekä ammattilaispelaajien 1-danista 9-daniin, joista 9-danin ammattilaispelaaja on korkein taso. [7; 16]

4.2 AlphaGo

AlphaGo on Google Deepmindin kehittämä ohjelmisto, joka kykenee pelaamaan gota ja joka voitti 9-dan tasoisen ammattilaispelaajan Lee Sedolin maaliskuussa 2016 [18]. Tässä tutkielmassa keskitytään tammikuun 2016 tilanteeseen, sillä maaliskuun 2016 jälkeen Deepmind ei ole julkaissut yhtä kattavaa katsausta ohjelmastaan. Väliin jäävän kahden kuukauden aikana ei kuitenkaan tapahtunut merkittäviä muutoksia AlphaGon arkkitehtuurin rakenteeseen, ja vain syväoppimisen jatkuminen sekä laskentakapasiteetin kasvattaminen on kehittänyt AlphaGota [19].

AlphaGo käyttää konvoluuttisia neuroverkkoja, jotka saavat sisääntuloiksi 19 x 19 x 48- kokoisen kuvan pelilaudasta, josta ne kykenevät luomaan esityksen pelilaudan tilasta. Lukujen 19 muodostama neliö vastaa pelilaudan jokaista ruutua, ja lisänä olevat 48 tasoa ovat erityisiä erikoispiirre-arvoja. Näistä 48:sta kolme liittyy värien tunnistukseen (onko ruutu tyhjä vai valkoisen tai mustan kiven täyttämä) ja loput

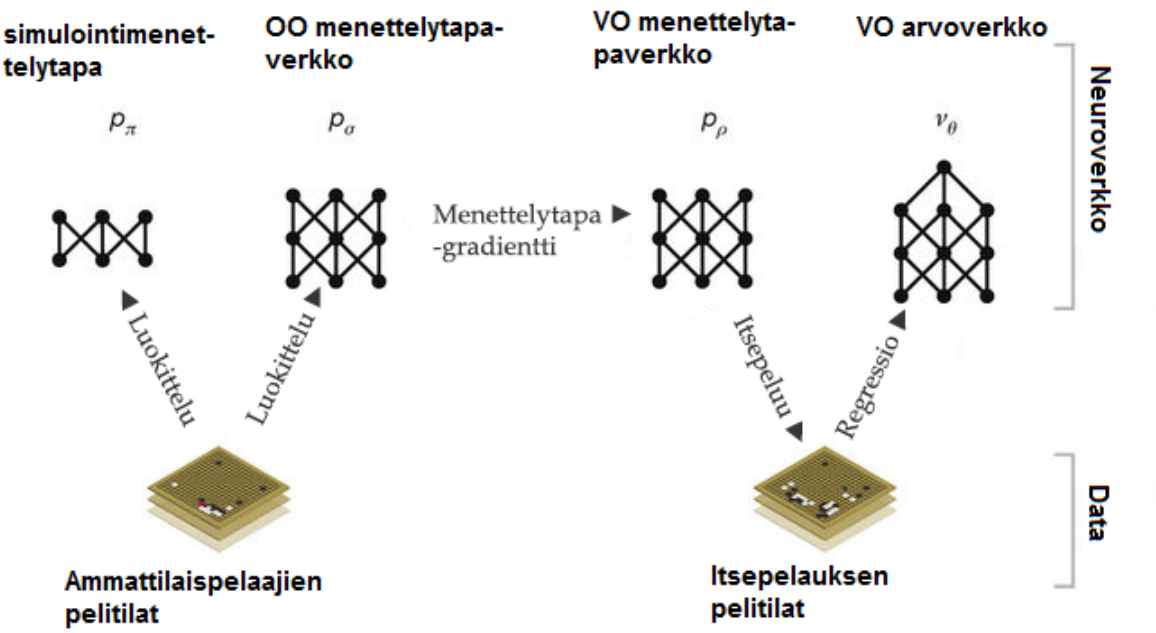
auttavat neuroverkkoa tunnistamaan ryhmien sidoksia ja vapauksia sekä muita seikkoja, kuten kuinka monta vuoroa on aikaa pelaamisesta. [1]

Lähtökohtana on neuroverkkojen opettaminen antamalla niille tietoa gon ammattilaispelaajien siirroista, ja luomalla näistä tiedoista menettelytapaverkko (eng. *policy network*). AlphaGon opettamisprosessi käsittää kolme neuroverkkoa. Neuroverkot yhdistetään Monte Carlo -puuhakuun.

Ensimmäinen neuroverkko – ohjatun oppimisen (eng. *supervised learning*) menettelytapaverkko (tästä eteenpäin p_σ) – käsittää 13 tasoa. Ulostulona sillä on todennäköisyysjakauma kaikkien laillisten siirtojen *a* osalta. Ohjatun oppimisen (OO) menettelytapaverkkoa opetetaan tekemällä satunnaisia näytteitä pelitilapareista (*S*, *a*), joiden halutaan muistuttavan mahdollisimman paljon ihmispelaajan siirtoa a tilassa *S.* AlphaGon OO-verkkolle annettiin dataksi 30 miljoonaa pelitilaa.

Ohjatun oppimisen menettelytapaverkko kykeni opettamisen jälkeen ennustamaan ammattilaisten siirtoja paremmin kuin muut go-ohjelmat (55,7 %:n tarkkuus vastaan 44,4 %:n tarkkuus) vuonna 2015 [20]. Menettelytapaverkon suoritusaika siirron löytämiseksi oli 3 millisekuntia. AlphaGota varten kehitettiin myös nopeasti toimiva simulointimenettelytapa p_π, joka oli epätarkempi (24,2 %), mutta kykeni valitsemaan siirron 2 mikrosekunnissa. [1]

Toisen neuroverkon – vahvistetun oppimisen (VO, eng. *reinforcement learning*) menettelytapaverkon (p_ρ) – tarkoitus on parantaa menettelytapaverkon toimintaa erityisellä gradientilla, joka muuttaa painotuskertoimia. Verkon rakenne on sama, ja sen painotuskertoimet ρ alustetaan samoin kuin OO-verkon painotuskertoimet σ. Siinä AlphaGo pelaa satunnaisesti valittua, aiempaa versiota itsestään vastaan ja oppii virheistään. Satunnaistamisella estetään verkon jämähtäminen johonkin tiettyyn pelityyliin, sillä mikäli se pelaisi jokaisella iteraatiolla vain edellistä versiotaan vastaan, riittäisi yhden tietyn pelityylin opettelu.



Kuva 8: Simulointimenettelytapaa ja ohjatun oppimisen (OO) menettelytapaverkkoa opetetaan ennakoimaan ammatilaispelaajan siirtoja pelitilojen joukosta. Vahvistetun oppimisen (VO) menettelytapaverkko alustetaan vastaavaksi kuin OO-verkko, ja tämän jälkeen siihen liitetään erityistä gradienttia itsepeluun yhteydessä, jotta se oppisi löytämään parhaita lopputuloksia. Lopuksi VO arvoverkko regressoi menettelytapaverkoista saadut todennäköisyysjakaumat ja saa niistä tulokseksi yksittäisen odotusarvon voittajasta. [1]**(Suom. tekijä.)**

Kolmas neuroverkko on vahvistetun oppimisen arvoverkko (eng. value network). Sen tarkoitus on käyttää arvofunktiota ennustamaan pelin lopputulos kummallekin pelaajalle tilasta S. Optimaalisen arvofunktion $v^*(S)$ saavuttaminen on sekä vaikeaa että epäkannattavaa, sillä se vaatisi pelaajien valitsevan aina parhaat siirrot. Tämän sijaan arvofunktio $v^p(S)$ estimoidaan arvoverkolle (v_θ) käyttäen aiemmin saatua vahvistetun oppimisen menettelytapaverkkoa p_ρ, jolloin $v_\theta(S) \approx v^p{}_\rho(S) \approx v^*(S)$. Arvoverkon arkkitehtuuri on sama kuin menettelytapaverkoilla, mutta sen ulostulona on yksittäinen arvio pelin lopputuloksesta (voittajasta) todennäköisyysjakauman sijaan.

AlphaGo yhdistää neuroverkot APV-Monte Carlo -puuhakuun (asynchronous policy and value). Pelipuun jokainen reunakohta sisältää siirtoarvon $Q(s, a)$, vierailumäärän $N(s, a)$ ja prioritodennäköisyyden $P(s, a)$. Pelipuuta simuloidaan Monte Carlo -algoritmilla, kuten aiemmin luvussa 2.3 kuvailtiin. Simulaatio käyttää UCT-Monte Carlo -puuhaullakin esiintynyttä solmukohtaista bonus-arvoa, joka vähenee

jokaisella vierailukerralla, jotta algoritmi kävisi puuta laajemmin läpi vierailemalla myös tarkastelemattomissa solmuissa.

Kun saavutetaan osapuun lehtisolmu S_L askeleella L, laajennetaan pelipuuta tästä alkaen. Lopputulosten todennäköisyydet tallennetaan prioritodennäköisyyksiksi jokaiselle lailliselle siirrolle a valvotun oppimisen menettelytapaverkkoon p_σ. S_L evaluoidaan kahdella tavalla: arvoverkon v_θ osalta, sekä kunnes saavutetaan lopettava tila T nopean, satunnaistetun simulointimenettelytavan osalta ja saadaan tuloksena Z_L. Nämä evaluaatiot yhdistetään sekoitusparametriä λ käyttäen lehtisolmun evaluaatioksi $V(S_L)$ [1]:

$$V(S_L) = (1-\lambda)v_\theta(S_L)+\lambda Z_L.$$

Simulaation lopuksi kaikkien vierailtujen reunojen siirtoarvot ja vierailumäärät päivitetään. Jokainen reuna kokoaa sen kautta suoritettujen simulaatioiden vierailujen määrän $N(a, s)$ ja keskimääräisen evaluaation $Q(a, s)$. Kun haku on valmis, valitaan juurisolmusta J eniten vierailtu siirto. Ilman arvoverkon v_θ käyttöä ohjatun oppimisen menettelytapaverkko kykeni parempaan kokonaisvaltaiseen peliin kuin vahvistetun oppimisen menettelytapaverkko. Tämän on oletettiin johtuvan siitä, että VO etsii vain optimaalista siirtoa eikä laajempaa joukkoa lupaavilta vaikuttavia siirtoja. Kuitenkin kun näitä käytettiin yhdessä v_θ kanssa, VO suoriutui paremmin. [1]

Evaluointimenettely ja arvoverkot vaativat merkittävästi enemmän laskentatehoa kuin perinteiset, heuristiset hakumenetelmät. AlphaGo suorittaa asynkronista, monisäikeistä Monte Carlo -puuhakua keskusprosessoreilla (eng. *CPU*) ja menettelytapa- ja arvoneuroverkkoja graafisilla prosessoriyksiköillä (eng. *GPU)*. Lopullinen AlphaGo käytti 40 hakusäiettä 48 keskusprosessoriyksiköllä ja 8 graafista prosessoriyksikköä. Useammalle laitteelle jaoteltu AlphaGo käytti sen sijaan 40 hakusäiettä 1202 keskusprosessorilla ja 176 graafista prosessoriyksikköä. AlphaGo voitti 494 495 ottelusta muita go-ohjelmia vastaan, ja myös Euroopan go-mestarin, dan 2-tasoisen pelaajan Fan Huin lokakuussa 2015. [1]

Google Deepmindin myöhemmin, vuonna 2017 kehittämä AlphaGo Zero voitti AlphaGon. AlphaGo Zeron ominaisuus on, että se kykenee oppimaan pelaamalla itseään vastaan eli kokonaan ilman, että sille syötetään dataa ammattilaispelaajien

esimerkkisiirroista. AlphaGo Zero koostuu vain yhdestä neuroverkosta, ja sen käyttämä variantti Monte Carlo -puuhausta on huomattavasti yksinkertaisempi eikä sisällä simulointivaihetta. [19]

5 Yhteenveto

Tässä tutkielmassa esiteltiin pelipuita käyttäviä algoritmeja sekä niiden toimintaa, tärkeimpänä Monte Carlo -puuhaku sekä sen variantti UCT. Lisäksi käytiin lyhyesti läpi neuroverkkojen toimintamalli ja rakenne. Neuroverkkoja kohtaan on tällä hetkellä kovia odotuksia, ja niiden kehittäminen tarjoaa yhä uudempia ja tehokkaampia sovelluskohteita aivan uudenlaiselle tietojenkäsittelylle ja laskennalle.

Go-lautapeli on vain yksittäinen sovelluskohde, mutta sen kompleksisuus huomioon ottaen se tarjoaa hyvän pohjan tutkia neuroverkkojen ja joidenkin ratkaisua etsivien algoritmien yhteistoimintaa.

AlphaGon ja siitä kehitettyjen versioiden AlphaGo Zeron ja AlphaZeron nopea kehitys ja itseoppiminen avaavat valtavia mahdollisuuksia tulevaisuudessa. Versioiden väliin jäävän kahden vuoden aikanakin tapahtui jo suurta kehitystä, joten tämä lupaa saman kehityksen jatkumista ja luultavasti sen kiihtymistäkin.

6 Lähdeluettelo

[1] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M *ja muut.* 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, sivut 484-489.

[2] Jouni Smed & Harri Hakonen. (2017). Algorithms and Networking for Computer Games, 2nd Edition. sivut 129-158.

[3] Gelly S, Kocsis L, Schoenauer M, Sebag M, Silver D, Szepesvari C & Teytaud O. 2012. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Commun. ACM* 55, sivut 106-113.

[4] Liang L, Hong L, Hao W, Taoying L & Wei L. 2014. A Parallel Algorithm for Game Tree Search Using GPGPU.

[5] Yifan J & Shaun B. 2015. Monte Carlo tree search report; Stanfordin yliopisto "CME 323: Distributed Algorithms and Optimization".

[6] Auer P, Cesa-Bianchi N & Fischer P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, sivut 235-256.

[7] Gelly S & Silver D. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artif. Intell.* 175, sivut 1856-1875.

[8] Lorentz R. 2016. Using evaluation functions in Monte-Carlo Tree Search. *Theor. Comput. Sci.* 644, sivut 106-113.

[9] Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, Lanctot M, Sifre L, Kumaran D, Graepel T *ja muut.* 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.

[10] Champandard A. 2014. Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI http://aigamedev.com/open/coverage/mcts-rome-ii/ sivun tila 9.5.2018

[11] Zhang Z. 2016. A gentle introduction to artificial neural networks. *Ann. Transl. Med.* 4,

[12] Huang W. 2018. Moving below the surface: Artificial Neural Network https://independentseminarblog.com/2017/10/03/moving-below-the-surface-artificial-neural-networks-william/ sivun tila 20.4.2018

[13] Schmidhuber J. 2015. Deep learning in neural networks: an overview. *Neural Networks : The Official Journal of the International Neural Network Society* 61, sivut 85-117.

[14] Sutskever, I. & Nair, V. (2008). Mimicking Go experts with convolutional neural networks. In Artificial Neural Networks - Icann 2008, Pt Ii, Kurkova, V., Neruda, R. and Koutnik, J. eds. (Berlin: Springer-Verlag Berlin), sivut 101-110.

[15] Introduction to Deep Neural Networks (Deep Learning) https://deeplearning4j.org/neuralnet-overview sivun tila 9.5.2018

[16] Muller M. 2002. Computer Go. *Artif. Intell.* 134, sivut 145-179.

[17] Muodostettu internet-go-pelissä "Cosumi" https://www.cosumi.net/en/ sivun tila 13.5.2018

[18] Borowiec S. 2016. AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol sivun tila 11.5.2018. *The Guardian*

[19] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A *ja muut.* 2017. Mastering the game of Go without human knowledge. *Nature* 550, sivut 354-359.

[20] Clark C & Storkey A. 2015. Training Deep Convolutional Neural Networks to Play Go. *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37* sivut 1766–1774.