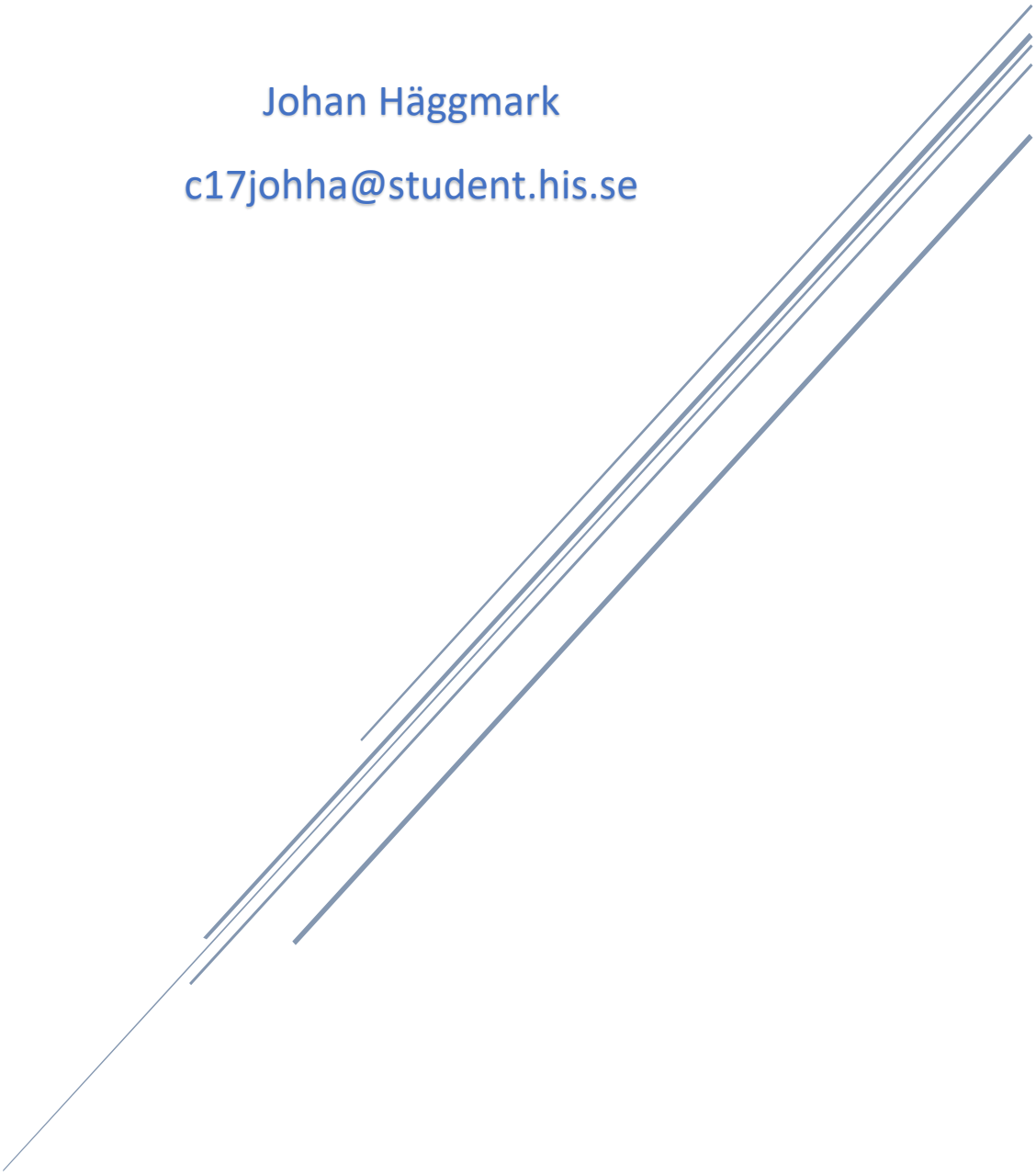
ARTIFICIELL INTELLIGENCE I TURORDNINGSBASERAT SPEL

Lättdriven generell Reinforcement Learning algoritm

Johan Häggmark

c17johha@student.his.se



Högskolan i Skövde
IT143G

Innehåll

Sammanfattning 2

Introduktion 3

Bakgrund 3

Historia 3

Nytänk i dagens algoritmer 4

Sammanhang 4

Spelet 4

Algoritmen 6

Neuralt nätverk 7

Bot 9

Metod 10

Resurser 10

Monte-Carlo tree search och Neuralt Nätverk 11

Träning 11

Datainsamling 11

Analys av träning 12

Resultat 13

Data 13

Instans 1 14

Instans 2 15

Instans 3 16

Instans 4 17

Instans 5 18

Instans 6 19

Instans 7 20

Instans 8 21

Analys 21

Diskussion 23

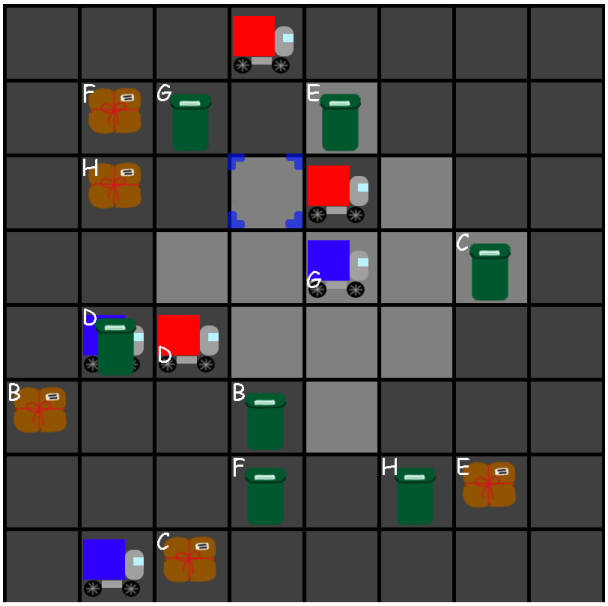
Referens 24

Sammanfattning

Denna rapport beskriver en Reinforcement Learning(RL) -algoritm i ett turordningsbaserat strategispel. Den implementerade algoritmen (även kallad Artificiell intelligens(AI)) använder sig av Monte-Carlo tree search(MCTS) och ett neuralt nätverk(NN). För att kunna ta till sig innehållet på bästa sätt, krävs det att man har introducerats för MCTS och NN.

Frågeställningen är huruvida algoritmen presterar och resultatet baseras på statistik, där datainsamling har utförts när AI:n har spelat flera tusentals matcher mot en bot.

Det har experimenterats med olika storlekar av det NN:et, dvs. antal ”hidden layers” och antal noder. Även inlärningsration och justeringar i MCTS har justerats. Algoritmen är skapad från grunden i Java och är tillämpad på ett egentillverkat spel som heter Truck Delivery, se Figur 1 Truck Delivery.



Figur 1 Truck Delivery

Introduktion

Bakgrund

AI breder ut sig i allt fler områden. Det forskas ständigt efter snabbare och effektivare algoritmer, samtidigt ökar datorers prestanda, vilket gör det lättare än någonsin att tillämpa AI.

Det finns flera subgenrer inom AI, bland annat RL och supervised learning(SL). Kaelbling, Littman & Moore. (1996) beskriver att inom genren RL är en av de svåra utmaningarna att effektivt hitta en optimal lösning (optimal lösning innebär att algoritmen kan producera perfekta resultat). RL tillämpas generellt när miljön är okänd.
Om det är möjligt att veta i vilken riktning en optimal lösning finns, kan det vara passande att tillämpa SL. Exempelvis bildigenkänning, och att det är känt vad träningsdatan(bilderna) innehåller. Det innebär också att SL har möjligheten att mäta hur nära algoritmen är en optimal lösning (åtminstone optimal för träningsdatan).
Att mäta hur nära en RL-algoritm är en optimal lösning kan vara svårt, då det saknas metoder som mäter om något är perfekt i en okänd miljö.

Reinforcement Learning

Antag att en spelare i ett spel står inför ett val. För att bestämma vilket val som ska göras, krävs en uppskattning av vilket val som maximerar chansen till att nå målet som eftersträvas. Det sanna värdet av val X beräknas genom att rekursivt utforska alla möjliga situationer efter val X.
Även om antal möjliga spelsituationer är ett ohanterligt stort tal, går det ändå använda strategin. Algoritmen väljer den mest lovande vägen, baserat på den begränsade utforskningen.

AlphaZero

AlphaZero är en världsledande algoritm när det kommer till komplexa turordningsbaserade strategispel. Algoritmen simulerar spel med hjälp av MCTS på ett generellt sätt och metoden kräver mycket datakraft. En annan del av AlphaZero, är att den använder sig av ett neuralt nätverk(NN). NN:et estimerar ett värde som används för att guida MCTS. Syftet att guida sökningen är att undvika simuleringar, där en eller flera spelare gör dåliga drag, dvs. ofarliga drag.

Ett tränat NN är en funktion där korrelationer har skapats med hjälp av backpropagation. Dessa korrelationer bidrar till att hitta abstraktioner i data, t.ex. bildigenkänning, då ett NN lärt sig att hitta specifika mönster i en bild. AlphaZeros NN hittar någon form av abstraktioner i en spelsituation och estimerar ett värde för den spelsituationen.

Historia

Datorer har under decennier varit ett tufft motstånd för människan, i strategiska spel. Schack är ett av dem.
1997 vann IBM:s Deep Blue (schackprogram) mot Garry Kasparov (dåvarande världsmästare i schack). Efter händelsen har intresset svalnat för ”människa möter dator i schack”, eftersom det är så tydligt att människan inte kan vinna. Idag är det mer intressant när ett datorprogram möter ett annat datorprogram.

Stockfish är ett schackprogram som vann TCE Computerchess championship 2016, Silver et al. (2017).

Datorprogram som spelar spel framgångrikt, är ofta ett mänskligt hantverk. Ett sådant program använder sig av datorkraft för att göra tunga rekursiva beräkningar med finjusterade

prioriteringsområden, d.v.s. programmeraren bestämmer vad som är viktigt i spelet. Den här metoden begränsar datorprogrammet till att aldrig bli smartare än sin skapare.

Nytänk i dagens algoritmer

AlphaZero är en AI som inte har några handgjorda prioriteringsområden. Därmed finns det möjlighet att hitta spelstrategier som kan vara utom människans kännedom.

Metoden AlphaZero använder sig av, har visat sig vara världsledande. Enligt Silver et al. (2017) har AlphaZero lärt sig att spela spel som Schack, Shogi (japanskt Schack) och Go på ett övermänskligt sätt. I samtliga fall har AlphaZero vunnit mot världsledande spelprogram, bland annat Stockfish.

Nyckeln till framgång:

- MCTS – Ett träd med noder och kanter som är simuleringar av möjliga utfall ifrån en viss situation. Trädet byggs upp av simuleringar. Detta är utgör grunden i att kunna föra statistik, som i sin tur ska sammanställas till strategiska beslut. Cameron et al. (2012) beskriver att det finns olika metoder av MCTS och att det finns flera sätt att tillämpa sökmetoden.
- Kampen mellan att utforska det okända i trädet eller att fördjupa sig i en väg som delvis redan är utforskad. Även om AlphaZero hade tillgång till enorma resurser, ska sökningar med dåliga val, till stor del undvikas. Att de inte ska undvikas helt, visar statistiskt ha en positiv påverkan på att lära sig nya strategier.
- NN:et estimerar värdet för en nod. Ju högre värde, desto högre prioritet att utforska grenarna efter den noden.

Källkoden till AlphaZero är inte tillgänglig för allmänheten och informationen i (Silver et al. 2017) beskriver inte arkitekturen av NN:et. AlphaZero tränades med hög parallellism på googles hårdvara Tensor Processing Unit (TPU) och efter 4 timmars träning slog den Stockfish. Ett antagande är att 4 timmars träning på specialanpassad hårdvara för machine learning, framtagen genom ett av världens främsta företag inom AI, skulle kunna skapa en oerhört komplex funktion (en funktion är vad ett NN är tänkt att skapa).

Sammanhang

Det skulle vara naturligt att tillämpa den bäst presterande algoritmen för en applikation. Tyvärr är tillgången av resurser en parameter som ofta har en stor påverkan på valet av algoritm. Resurser kan ses som processorkraft och minne. Processorkraft gör det möjligt att utföra någonting under en viss tid. Minne behövs för att hålla reda på information, och med information kan slutsatser produceras.

Det AlphaZero åstadkommit ligger till grund för den implementation som gjorts. AlphaZero visar att NN kan vara till nytta i MCTS. Resultatet i det här arbetet visar på huruvida en mer lättdriven algoritm kan prestera i ett turordningsbarserat strategispel.

Arkitekturen på AlpaZeros NN framgår inte av Silver et al. (2017). Därför är det intressant att utföra tester på olika storlekar av ett feedforward NN, där resultatet visar vikten av antal noder i ett NN. AlphaZeros MCTS är en krävande metod. En förenklad variant av MCTS har tillämpats, där NN fortfarande har en viktig roll.

Spelet

Algoritmen har testats på det ickekommersiella spelet Truck Delivery, Häggmark, Johan. 2018.

Truck Delivery är ett turodningsbaserat strategispel med fullständig information (all information i spelet är synligt för alla spelare), där två spelare möter varandra. Spelplanen är ett bräde med 8x8 rutor. På planen finns postpaket, postlådor och lastbilar. Spelarna har tre lastbilar var. Det går ut på att plocka upp postpaket med lastbilarna, för att sedan lämna paketen på rätt postlåda. Spelarna får flytta lastbilarna i rundor, då en lastbil har begränsat antal rutor den får flytta per runda. När en spelares lastbil har levererat ett postpaket till en postlåda, får spelaren poäng. Spelet har begränsat antal rundor och den som har mest poäng när rundorna är slut vinner.

Vid ett nytt spel placeras tio stycken postpaket och tio stycken postlådor slumpvist på olika positioner. En viktig detalj är att varje postpaket som utplacerats, måste ett till paket placeras på positionen som är spegelvänd till den ursprungliga positionen. Samma sak gäller för postlådorna. Detta bidrar till att de båda spelarnas lastbilar har samma distanser till postpaket och postlådor.

Lastbilarna har alltid samma startpositioner. Men postpaketen och postlådorna varierar. Antal möjliga spelsituationer vid start är:

$$64 * 62 * 60 * 58 * 56 * 54 * 52 * 50 * 48 * 46 = 4.610008941\text{E}15$$

Första postpaketet har tillgång till 64 möjliga positioner, i samband med att det läggs ut måste ett annat postpaket läggas ut på en spegelvänd position. För nästa postpaket återstår det 62 möjliga positioner. Sammanlagt ska 10 postpaket och 10 postlådor placeras ut. En lastbil kan ha 0 till 25 stycken möjliga positioner att förflytta sig till.

Vid start av spelet har en röd lastbil 14 möjliga positioner att förflytta sig till. Med lastbilens möjliga drag ökar antalet möjliga spelsituationer ökar till:

$$4.610008941\text{E}15 * 14$$

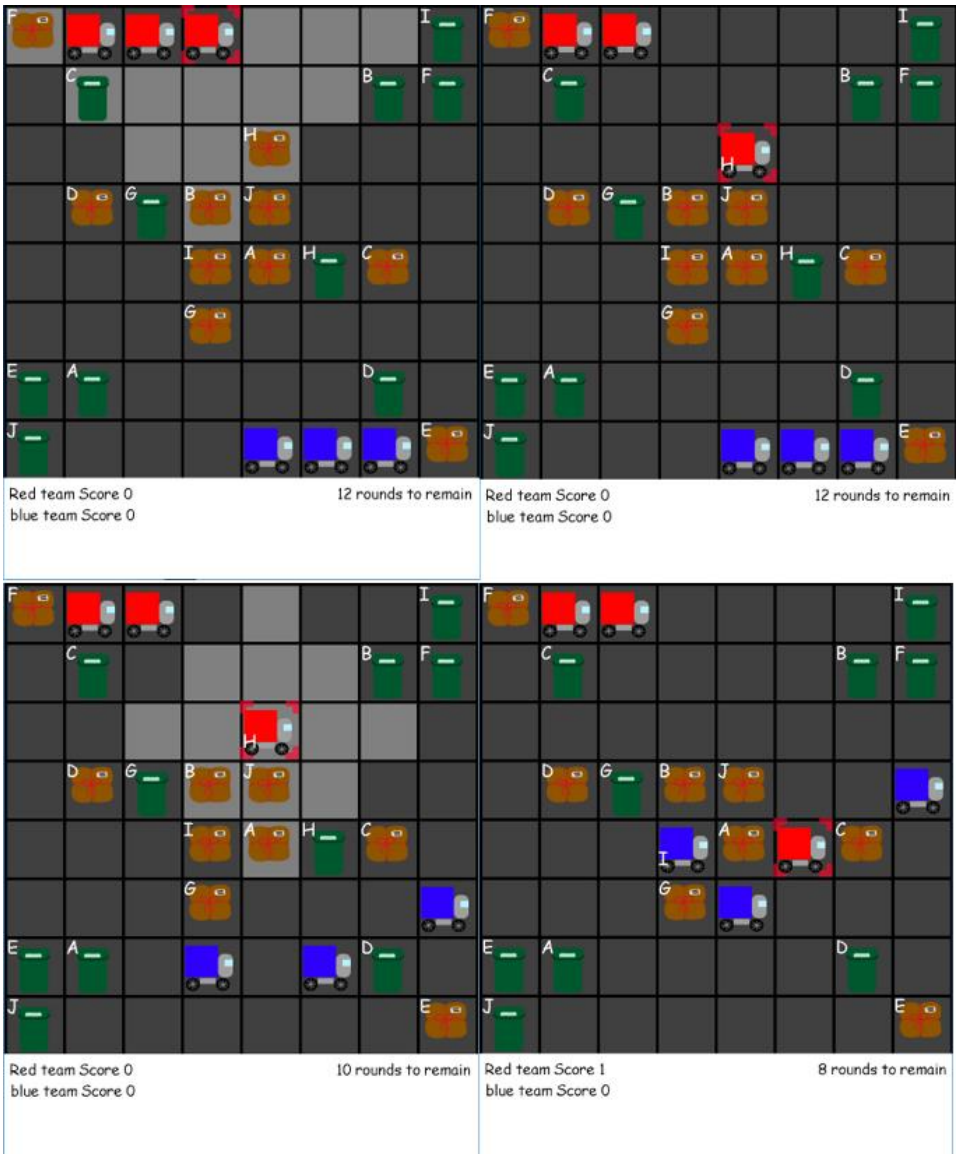
Medräknat nästa lastbil med 14 möjliga positioner att förflytta sig till, är antalet möjliga spelsituationer:

$$4.610008941\text{E}15 * 14^2$$

Antal spelsituationer växer exponentiellt ju fler drag som räknas, vilket gör det omöjligt att lagra värdet för alla spelsituationer.

Figur 2 Röd lastbil levererar ett paket, illustreras fyra spelsekvenser, hur spelaren med röda lastbilar tar ett poäng.

- En lastbil flyttas till rutan med postpaket ”H” och plockar automatiskt upp det. När en lastbil har ett postpaket, står paketets bokstav nere till vänster om lastbilen.
- När lastbilen kan flyttas igen, flyttar spelaren lastbilen mot postlådan med bokstaven ”H”.
- Beroende på hur långt avståndet är och hur många paket lastbilen har, kan det ta flera rundor att flytta lastbilen till en position. Att bära ett paket, minskar avståndet för de möjliga positioner. I Bilden är de ljusgråa rutorna de möjliga positionerna för den markerade lastbilen.
- I sista spelsekvensen längst ner till höger, har lastbilen levererat paket ”H” till postlåda ”H”. Spelaren får ett poäng ”Red team Score 1” och postpaket med postlåda försvinner.



Figur 2 Röd lastbil levererar ett paket

Figur 2 Röd lastbil levererar ett paket visar att när lastbilen inte bär något postpaket, kan den flytta sig fyra rutor bort, men när den plockat upp paket ”H”, kan den som max flytta sig tre rutor bort. En lastbil kan max bära två paket, men då kan den bara flyttas en ruta bort i taget.

Algoritmen

Allt är skapat från grunden i Java med endast standardbibliotek. Huvudkomponenterna är MCTS och NN.

Varianten av MCTS går ut på att fördela ut fullständiga simuleringar (Simuleringar till spelets slut) på några få av de möjliga dragen. NN:et avgör vilka av de möjliga dragen som ska simuleras. Efter valet, är simuleringen helt slumpmässig i vilka vägar den tar.

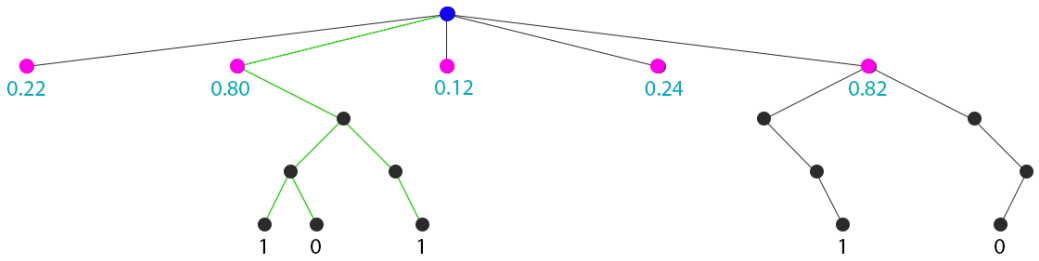
Fullständiga simuleringar är möjligt eftersom spelet alltid har fullständig information till båda spelarna. En simulator har utvecklats till AI:n så att simuleringar ska kunna utföras, utan att påverka det pågående spelet.

När AI:n ska göra ett drag, applicerar den spelsituationen i simulatorn. AI:n simulerar alla möjliga drag, där varje möjligt drag blir en ny spelsituation, dessa spelsituationer ska appliceras till NN:et som

inputdata (exakt vad detta är för inputdata beskrivs senare). NN:et producerar ett värde mellan 0 och 1. Värdet är en estimering på hur bra ett val är. Skulle värdet vara 1, är det estimerat till att garantera vinst oavsett vad som händer i spelet efter det valet.

I Figur 3 MCTS för AI:n illustreras det hur AI:n fattar ett beslut, på ett avskalat sätt. Varje nod är en unik spelsituation. AI:n har fem möjliga drag att välja bland från sin ursprungsposition. AI:n har fått resurser att utföra fem stycken fullständiga simuleringar. Processen går till så här:

1. Ta reda på vilka drag som är verkar vara mest lovande. AI:n simulerar varje drag (rosa noder) och använder varje spelsituation som inputdata till NN. NN producerar de värden som finns under de rosa noderna.
2. AI:n väljer att spendera de fullständiga simuleringarna på nod nr.2 och nod nr.5, eftersom de är estimerade till mest lovande resultat.
3. Trots att rosa nod nr.5 estimerades högst, väljer AI:n slutligen att göra draget för nod nr.2. Algoritmen väljer alltid den väg som ger det högsta medelvärdet från de fullständiga simuleringarna. Medelvärdet från nod nr.2 är 2/3. Nod nr.5:s medelvärde är 1/2
4. Alla medelvärden används som det sanna värdet till backpropagation till NN. NN tränas till att nod nr.2 estimeras till 2/3 istället för 0.8.



Figur 3 MCTS för AI:n

Istället för fem fullständiga simuleringar, som Figur 3 MCTS för AI:n visar, har AI:n i själva verket 50.000 fullständiga simuleringar att fördela ut.

Neuralt nätverk

NN:en som testats är av varianten ”feedforward” och består av 166 input-noder och en output-nod. ”feedforward” innebär att det inte finns några cykler i nätverket. Det finns input-layer, hidden-layers och output-layer, dessa är olika typer av lager. Det har testats olika antal hidden-layers och olika antal noder i dessa. Det minsta NN:et har haft ett hidden-layer med 16 noder. Det största NN:et har haft tre hidden-layer med 128 noder i varje lager. Varje lager har även en ”bias-nod”. Från ett lager, finns kopplingar från alla noder till alla noder i nästa lager. Kopplingarna utgörs av vikter, som är variabler. De startar med slumpmässiga värden mellan -1 och 1. Justering av vikter sker med backpropagation, för att skapa en önskvärd funktion. Tanken är att NN:et ska skapa en tillräcklig komplex funktion, där den som AlphaZero, känner igen abstraktioner av spelsituationer som är gynsamma. Det skulle innebära att AI:n fokuserar sökningen i värdefulla val.

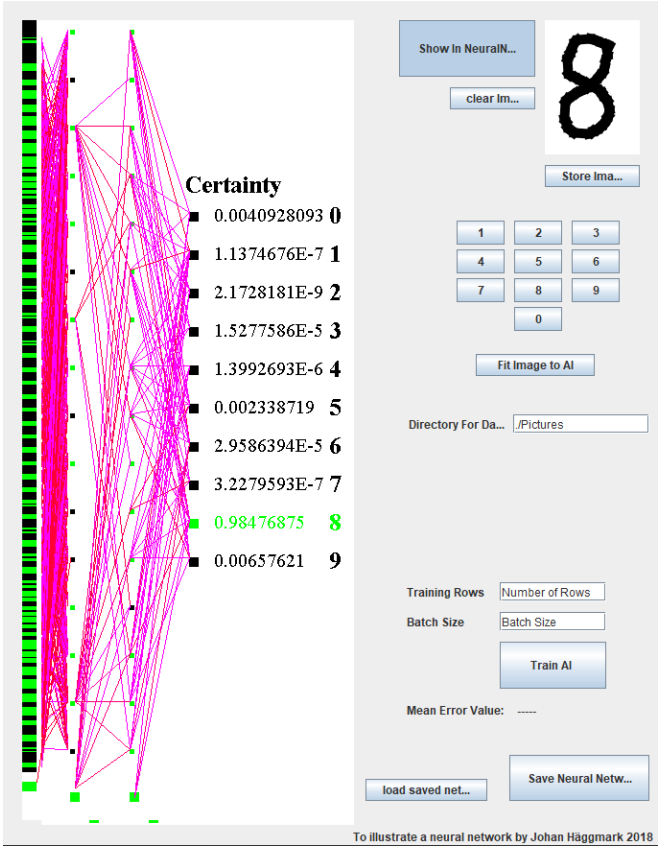
De 166 input-noderna är:

- För varje lastbild(6), distans (antal rutor) till varje postpaket(10), distans till varje postlåda(10) och distans till postlådor som hör till de paket som lastbilen bär(2). Det här är 132 variabler.

- För varje postpaket(10), distans till tillhörande postlåda(1), är postpaketet upplockat(1), är postpaketet levererat(1). Det här är 30 variabler.
- Antal rundor kvar(1), AI:ns poäng(1), motståndarens poäng(1), den som leder(1). Det här är 4 variabler.

Dessa variabler är AI:ns egenproducerade variabler. Många av dessa är beräkningar på objekts X- och Y-koordinater. Det är möjligt att använda koordinaterna direkt i NN, men det har visat sig vara effektivare med den processade datan. Tanken var att efterlikna strategitänket som en människa använder sig av. I schack är det viktigt hur pjäserna står i relation till varandra. Detta appliceras utan att påtvinga domänspecifika prioriteringar.
Det finns problem med att använda koordinater i NN, t.ex. postpaket med koordinater (8,8), har hög påverkar NN, medan postpaketet med koordinater(0,0) har ingen påverkan. Dessa koordinater kan vara lika viktiga i en spelsituation. Det är snarare relationen till olika objekt som avgör vad som bör prioriteras. Därför används avstånd mellan objekt istället.
Motiveringen att detta inte bidrar till domänspecifika prioriteringar är att backpropagation avgör om kopplingen i NN:et till dessa värden är relevant för estimeringen av en spelsituation. Är de inte det, kommer kopplingen att försvagas, så att de inte påverkar estimeringen.

Koden till NN har tidigare använts i en teckenigenkännings-applikation och visat sig att fungera, Häggmark, Johan. 2018. se Figur 4 NN känner igen siffran 8. NN:s kopplingar visas grafiskt till vänster, på vänster sida i bilden visas vilka kopplingar som blir aktiva i nätverket när pixlar för bilden på åttan skickas som inputdata till NN:et.



Figur 4 NN känner igen siffran 8. NN:s kopplingar visas grafiskt till vänster

NN använder sig av aktiveringsfunktionen ”Sigmoid”:

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

I Backpropagation används ”Gradient Descent” där det går ut på att minska ”error” (hur mycket fel nätverket estimerar ett värde) och derivatan är uträknad av funktionen:

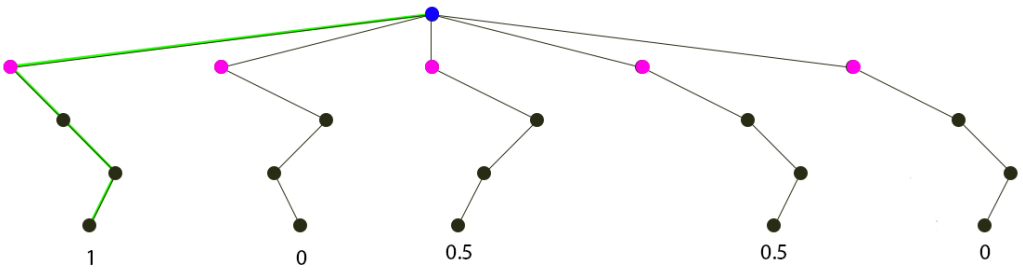
$$x(1 - x)$$

Att förklara matematiken bakom NN är en komplicerad resa och den kunskapen krävs inte för att ta till sig innehållet i denna rapport.

Bot

För att träna ett NN krävs stor mängd data. I det här fallet måste AI:n spela spelet många gånger. Det är orimligt att personligen träna AI:n. Därför är en ”bot” implementerad. Boten är ett program som kan spela spelet och är ett rimligt motstånd för denna AI:n. En skillnad på AI:n och boten, är att boten blir aldrig bättre eller sämre på att spela spelet.

Boten använder sig av en MCTS-metod, men den har inte tillgång till något NN. Låt säga att boten befinner sig i exakt samma spelsituation som AI:n i Figur 3 MCTS för AI:nFigur 3 MCTS för AI:n. Boten har på samma sätt som AI:n, tillgång till fem stycken fullständiga simuleringar. Eftersom boten inte har något NN som kan guida sökningen, får den helt enkelt sprida ut simuleringar på alla noder. Se Figur 5 MCTS för bot



Figur 5 MCTS för bot

Boten väljer att spela det drag med högst medelvärde från simuleringarna. I det verkliga testet har boten, precis som AI:n tillgång till 50.000 fullständiga simuleringar.

Metod

Det här avsnittet beskriver hur resultatet är producerat. Vissa aspekter från förra avsnittet beskrivs i en mer detaljerad nivå för att motivera vissa tillämpningar.

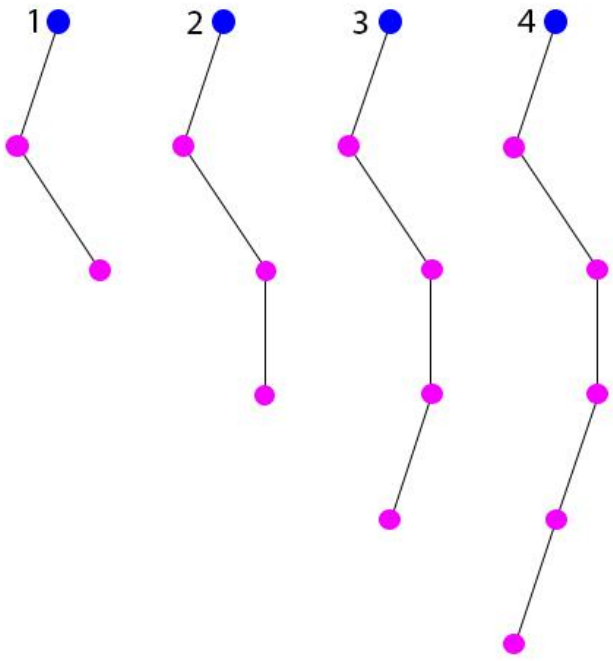
Resurser

AlphaZeros metod av MCTS kräver betydligt mer resurser än varianten som tillämpas i den implementerade AI:n. För varje ny nivå i sökningen, används NN:et för att estimera vilken gren som ska utforskas. Men för varje ny nod som tidigare inte besökts, börjar sökningen om från början.

Förutom att NN:et påverkar sökningen, finns en faktor som prioriterar grenar med lågt antal besök. Det här fungerar som en balansbräda där ena sidan vill utforska något obekant och den andra sidan vill fördjupa sig i något som den vet är bra.

Den här varianten implementerades i Truck Delivery. Det visade sig vara alldeles för tungrott för att producera ett resultat inom rimlig tid, då ingen parallell databehandling tillämpades. Allt beräknades på en processorkärna, till skillnad från AlphaZero som använde sig av hög parallellism med hjälp av Googles TPU.

AlphaZeros MCTS måste hålla koll på tidigare besökta grenar. Detta kräver hög frekvens av läsning till och från minnet och effekten blir att MCTS tar lång tid. Djupet i sökningen tar särskilt lång tid eftersom sökningen måste börja från start varje gång en ny nod/spelsituation upptäcks. I Figur 6 MCTS itererar för djup sökning, visas hur iterationen går till. MCTS har sökt sig till nivå två(1), för att komma till nivå tre itereras sökningen från början(2). I (3) och (4) itereras processen för att komma ännu djupare. Varje rosa nod innebär en läsning från minnet. (2)-(4) krävde tolv minnesläsningar och sökningen kom tre nivåer djupare. Minneshantering är ofta en flaskhals för datorprogram idag.



Figur 6 MCTS itererar för djup sökning

Denna iteration sker för att den tidigare nämnda balansbrädan ska fungera. Detta ger en chans till att utforska okända noder istället för att bara utforska djupet. Det är inte bara läsning i minnet vid varje

rosa nod, utan även skrivning. Nodens estimering ska uppdateras till att bli mindre värd, för varje besök.

Monte-Carlo tree search och Neuralt Nätverk

Den implementerade metoden saknar ”balansbrädan” i MCTS, istället använder den endast djupsökning och slumpmässiga vägar. Det är endast första nivån som NN bestämmer vilken väg som ska utforskas.

När AI:n tillämpar MCTS vid ett val. Simuleras alla möjliga drag i första nivån. Dessa blir spelsitutationer och appliceras till NN:et som i sin tur estimerar värden för samtliga drag. MCTS väljer ut olika antal drag i olika tester. Kombinationer med 4-6 av de högst estimerade dragen och 2 av de lägst estimerade dragen har testats.

De utvalda estimerade dragen är de enda som NN:et kommer tränas mot. Att endast välja de drag som estimerats högst, skulle kunna ses som effektivast eftersom AI:n har ett begränsat antal fullständiga simuleringar och då skulle det vara onödigt att använda dessa på vägar som NN:et estimerat lågt. En teori är att om de lågt estimerade vägar aldrig utforskas, kan det aldrig säkerställas att det sanna värdet också är lågt.

Träning

I sekvenser har åtta stycken instanser av AI:n tränats. Varje instans har haft ett unikt NN med slumpmässiga vikter från början. Det har testats två olika faktorer ”learning rate” (1 och 0.1) när backpropagation har ändrat vikterna. När ”error” inte haft en negativ trend, testas en annan ”learning rate”, för att komma ifrån eventuellt ”overfitting” eller andra problem som kan orsaka att backpropagation inte fungerar som förväntat.

Datainsamling

Data samlas in från och med första spelet för en ny instans av AI:n. För varje spel lagras två olika värden på en ny rad i ett excel-dokument som är kopplat till just den instansen av AI:n:

1. Vinst = 1, lika = 0, förlust = -1, skrivs i kolumn A
2. Medelvärdet av alla error-värden som AI:n använt i backpropagation under spelet, skrivs i kolumn B.

	A	B
1	-1	0,277058
2	1	0,239901
3	0	0,314202
4	1	0,242075
5	1	0,157472
6	1	0,18784
7	1	0,265498
8	1	0,232081
9	-1	0,207122
10	1	0,157948

Figur 7 Datainsamling, Kolumn A = vinst/lika/förlust, B = error. Rader = spel

Analys av träning

Under träningen kontrolleras datan i excel-dokumentet, och så länge antydan till förändring i resultatet finns, har träningen fortgått. Det har kontrollerats genom observation av medelvärden i grupper av 20 rader som applicerats i en graf.

Storleken på NN:et påverkar hur många spel AI:n hinner spela under en tid, men främst är det MCTS som tar tid.

Att utföra 1000 spel tar ungefär fem timmar.

Resultat

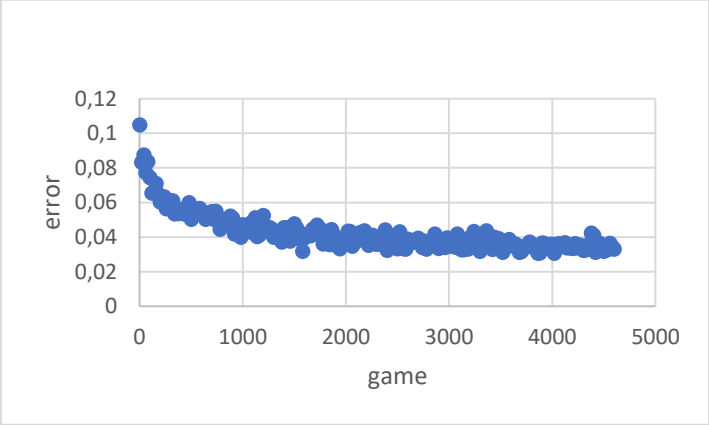
Det här avsnittet visar resultatet av den tillämpade metoden. Egenskaper för varje instans beskrivs och resultatet visas i form av grafer. Det tillkommer en kort analys om varje resultat. Fördjupning om resultatet beskrivs i slutet av avsnittet.

Data

Beskrivningarna för en instans är:

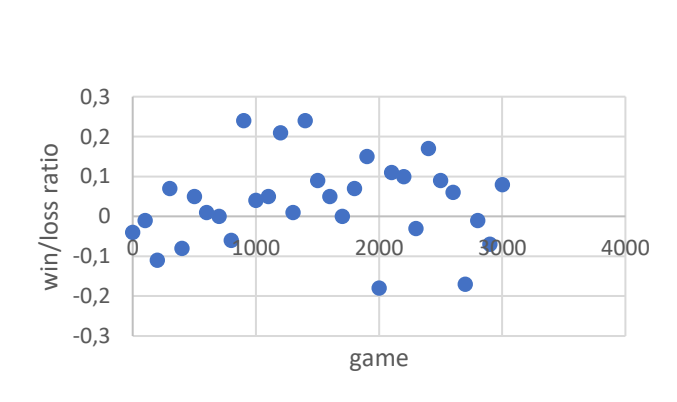
- Antal hiddenlayer i NN:et, samt antal noder i varje layer.
- Inlärningshastighet applicerad i backpropagation.
- Antal möjliga drag som NN:et estimerat högst, som MCTS ska utföra fullständiga simuleringar på
- Antal möjliga drag som NN:et estimerat lägst, som MCTS ska utföra fullständiga simuleringar på

Graf som visar genomsnitt av error. X-axeln är vilket spel errorvärdet tillhör, se Figur 8 graf för error



Figur 8 graf för error

Graf som visar vinst/förlust-ratio. 1 = vinst, 0 = lika och -1 = förlust. X-axeln är vilket spel, se Figur 9 graf för vinst ratio

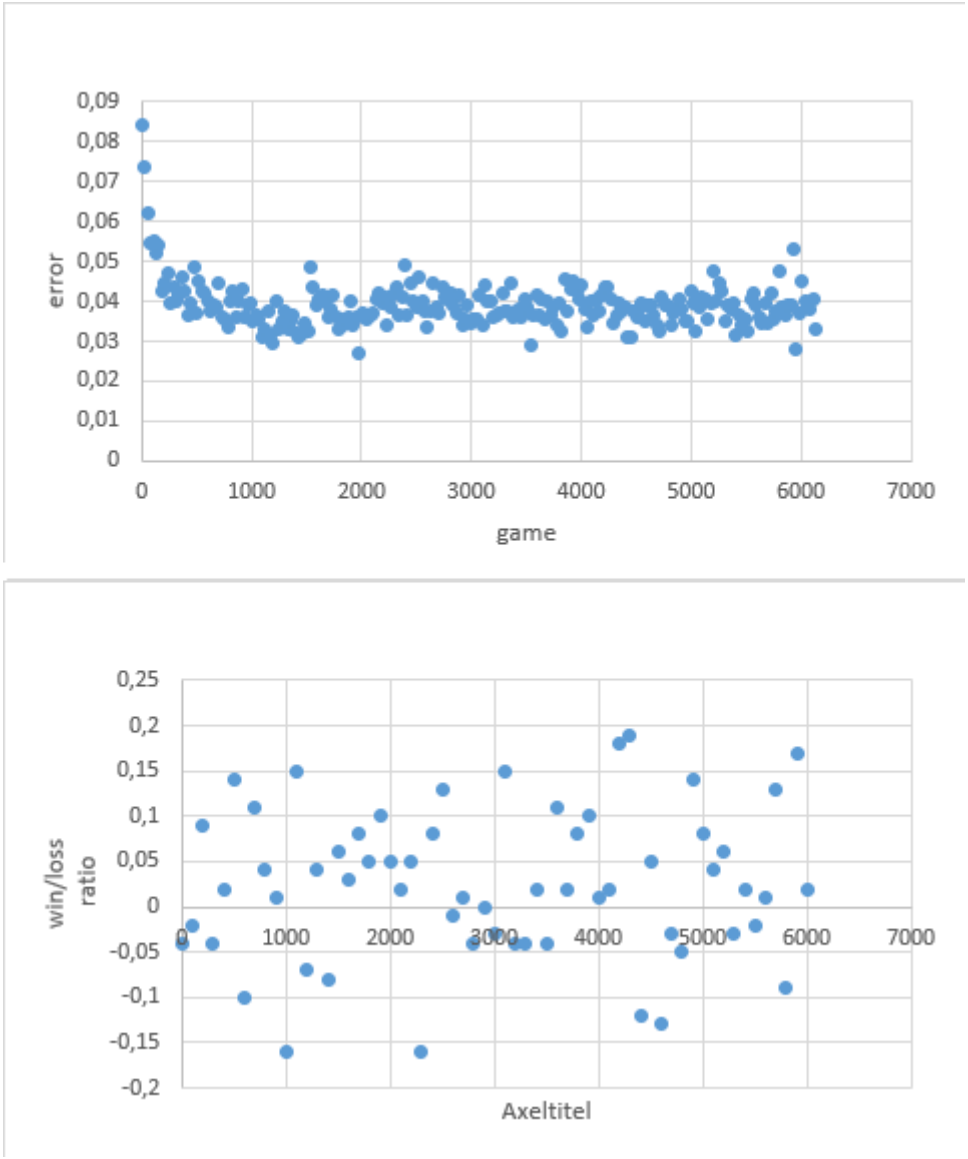


Figur 9 graf för vinst ratio

Prickarna i graferna är genomsnitt av data från flera spel. Det gör det tydligare att se trender

Instans 1

- 1 hiddenlayer med 16 noder
- Learning rate = 1
- Högt estimerade vägar = 4
- Lågt estimerade vägar = 2

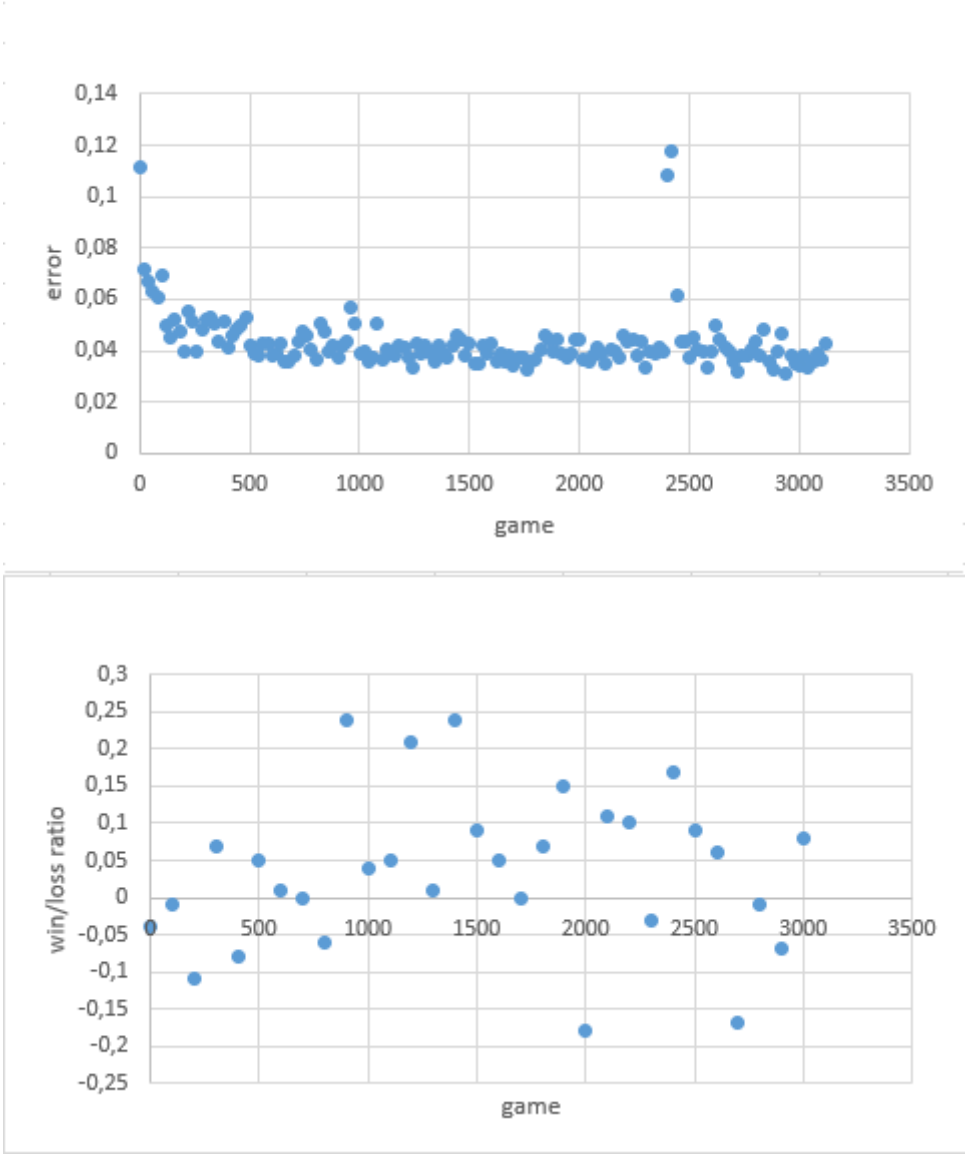


Figur 10 instans 1

Det här är instansen med det minsta NN. Efter 500 ca spel slutar error-kurvan att minska. Det är med en vag statistik som AI:n vinner oftare än boten.

Instans 2

- 1 hiddenlayer med 32 noder
- Learning rate = 1
- Högt estimerade vägar = 4
- Lågt estimerade vägar = 2



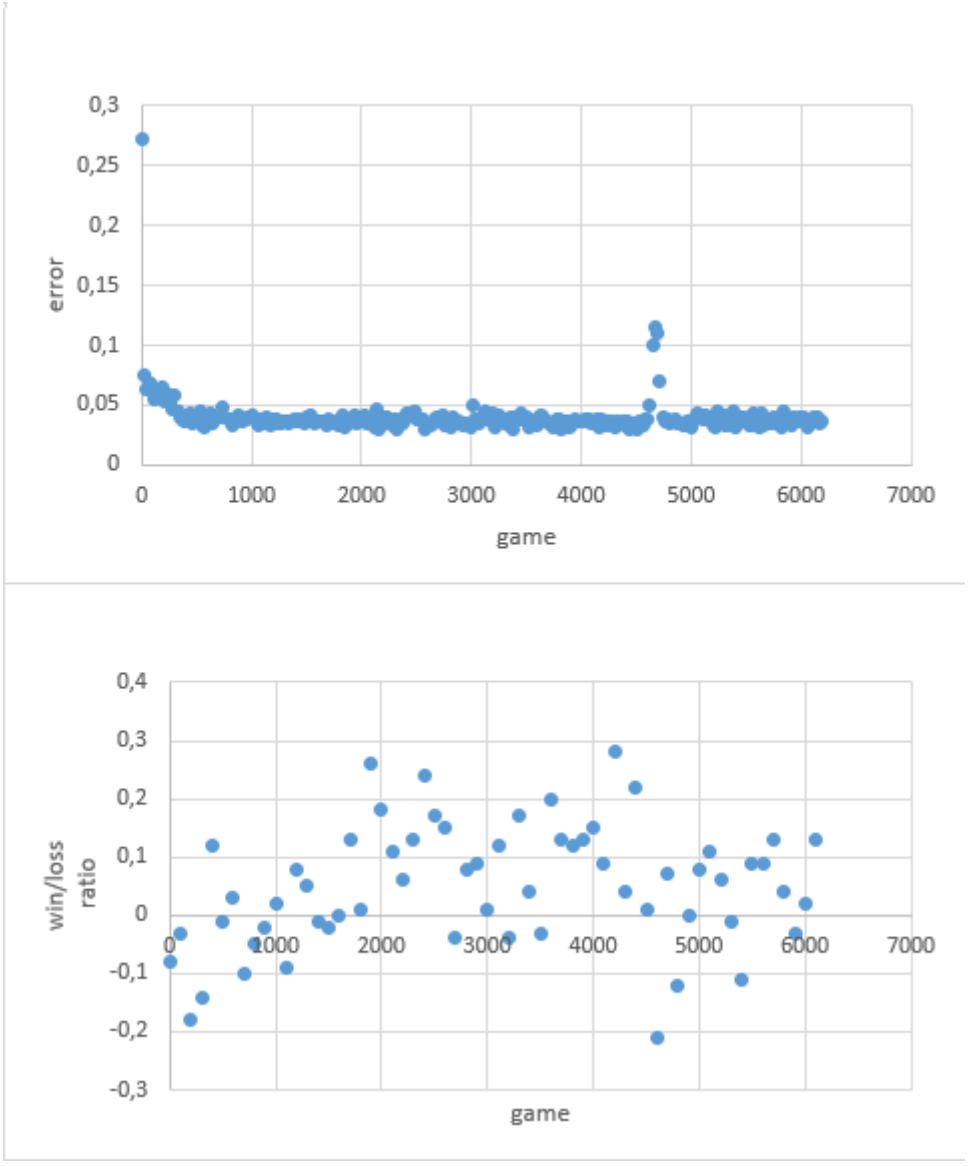
Figur 11 instans 2

Instansen har 32 noder istället för 16 som i föregående.

Notera att error-värdet är något ostadigt efter 2400 spel. Samtidigt börjar fler förluster dyka upp.

Instans 3

- 1 hiddenlayer med 64 noder
- Learning rate = 1
- Högt estimerade vägar = 4
- Lågt estimerade vägar = 2



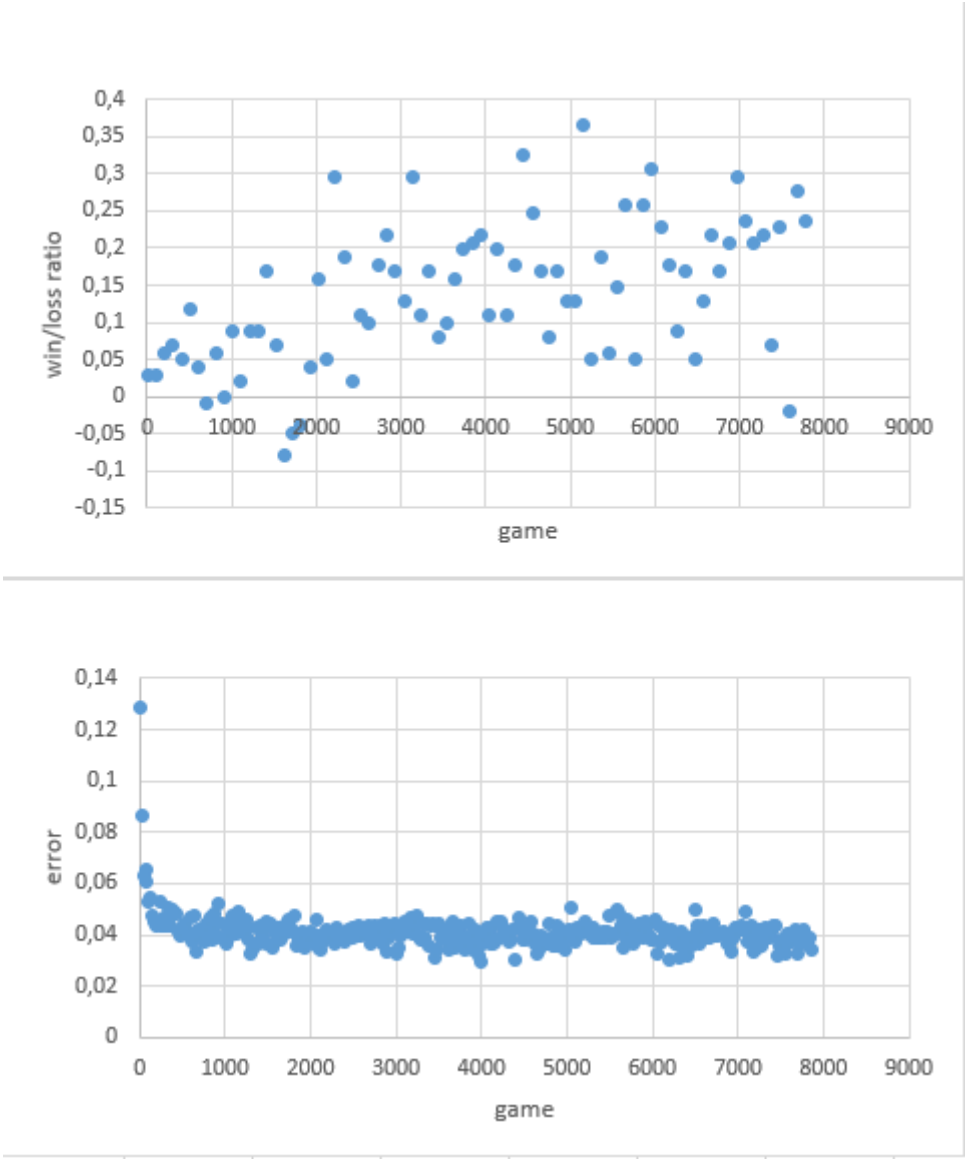
Figur 12 instans 3

Instansen har 64 noder istället för 32 som i föregående.

Det är en positiv trend i vinster fram till 4500 spel. Vid en kort period uppstår högre error och efter det minskar antal vinster.

Instans 4

- 2 hiddenlayer med 64 noder
- Learning rate = 1
- Högt estimerade vägar = 4
- Lågt estimerade vägar = 2



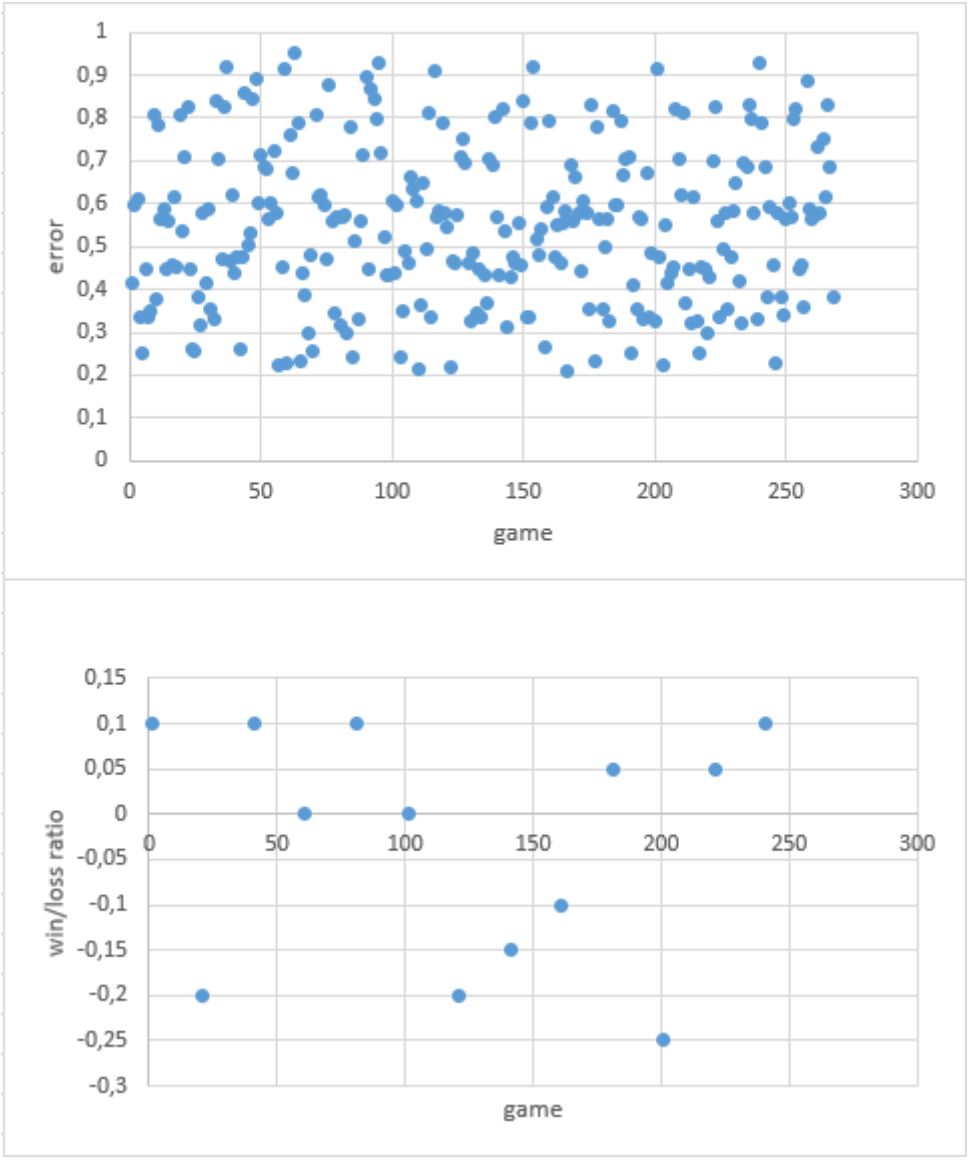
Figur 13 instans 4

Instansen har två hiddenlayers istället för ett som föregående.

Det är en positiv trend i antal vinster fram till 4000 spel. Det är oklart om det är förändring till något bättre efter det.

Instans 5

- 2 hiddenlayer med 128 noder
- Learning rate = 1
- Högt estimerade vägar = 4
- Lågt estimerade vägar = 2



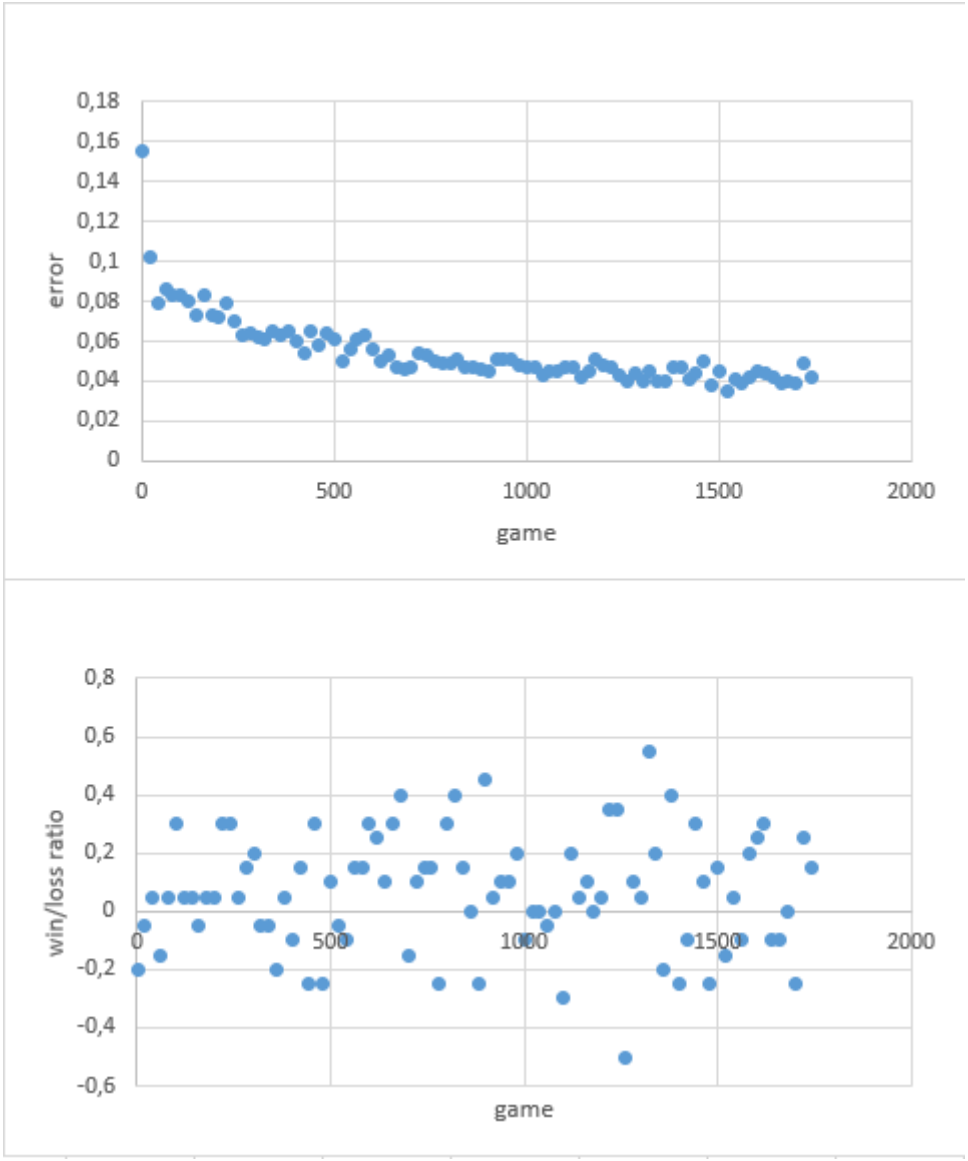
Figur 14 instans 5

Instansen har 128 noder istället för 64 som föregående.

Här är det oklart varför NN:et inte kan hantera 128 noder i hiddenlayers. Backpropagation ändrar vikterna för mycket. Problemet är frånvarande implementation av genomsnitt errorberäkning, dvs. error-värden blir för stora. Det bidrar till att derivatan i ”Gradient Descent” närmar sig noll och ingen ändring sker på vikterna i backpropagation.

Instans 6

- 2 hiddenlayer med 128 noder
- Learning rate = 0,1
- Högt estimerade vägar = 4
- Lågt estimerade vägar = 2



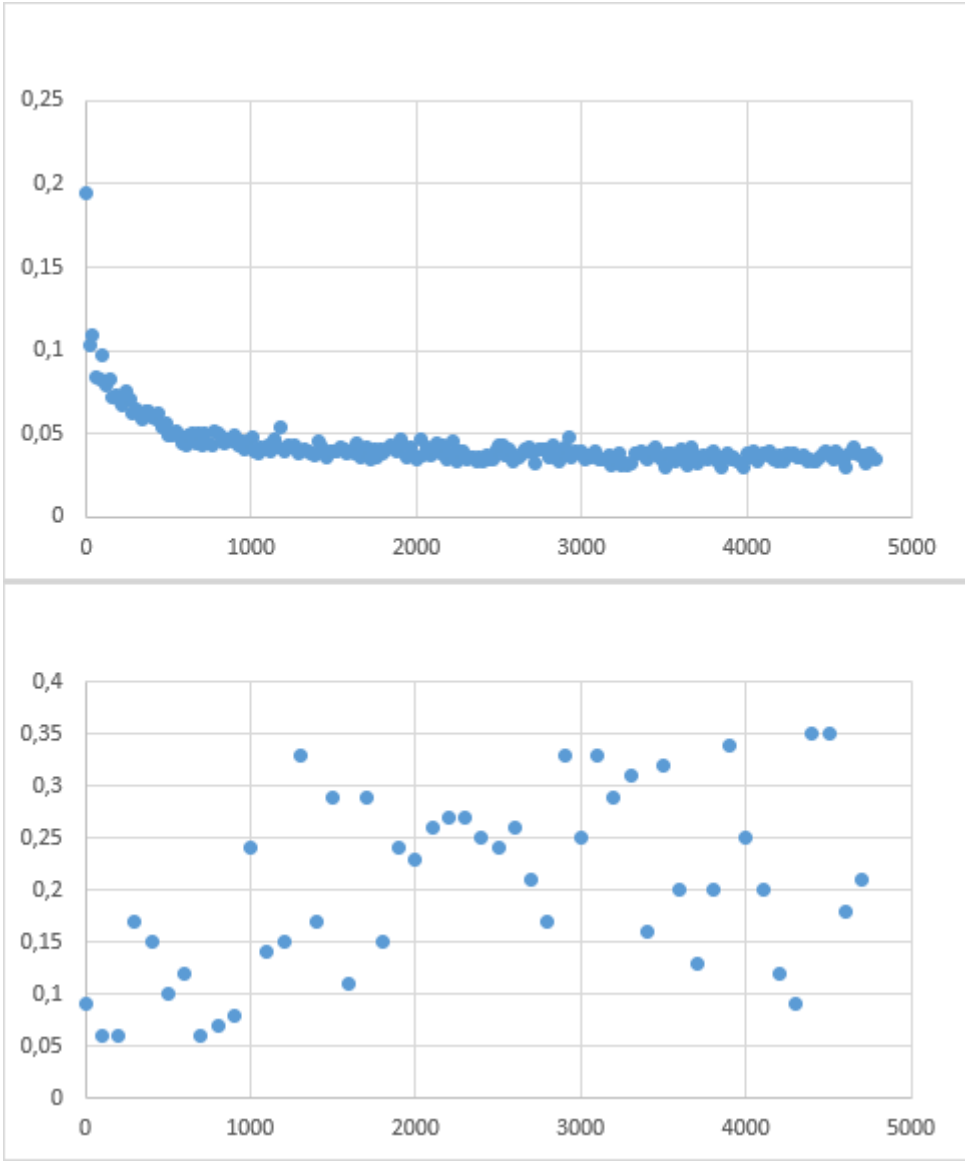
Figur 15 instans 6

Instansen har 0,1 i ”Learning rate” istället för 1, som i föregående.

Antal vinst är relativt oförändrat från 500 spel. Dock en antydan till större spridning.

Instans 7

- 2 hiddenlayer med 128 noder
- Learning rate = 0,1
- Högt estimerade vägar = 6
- Lågt estimerade vägar = 2

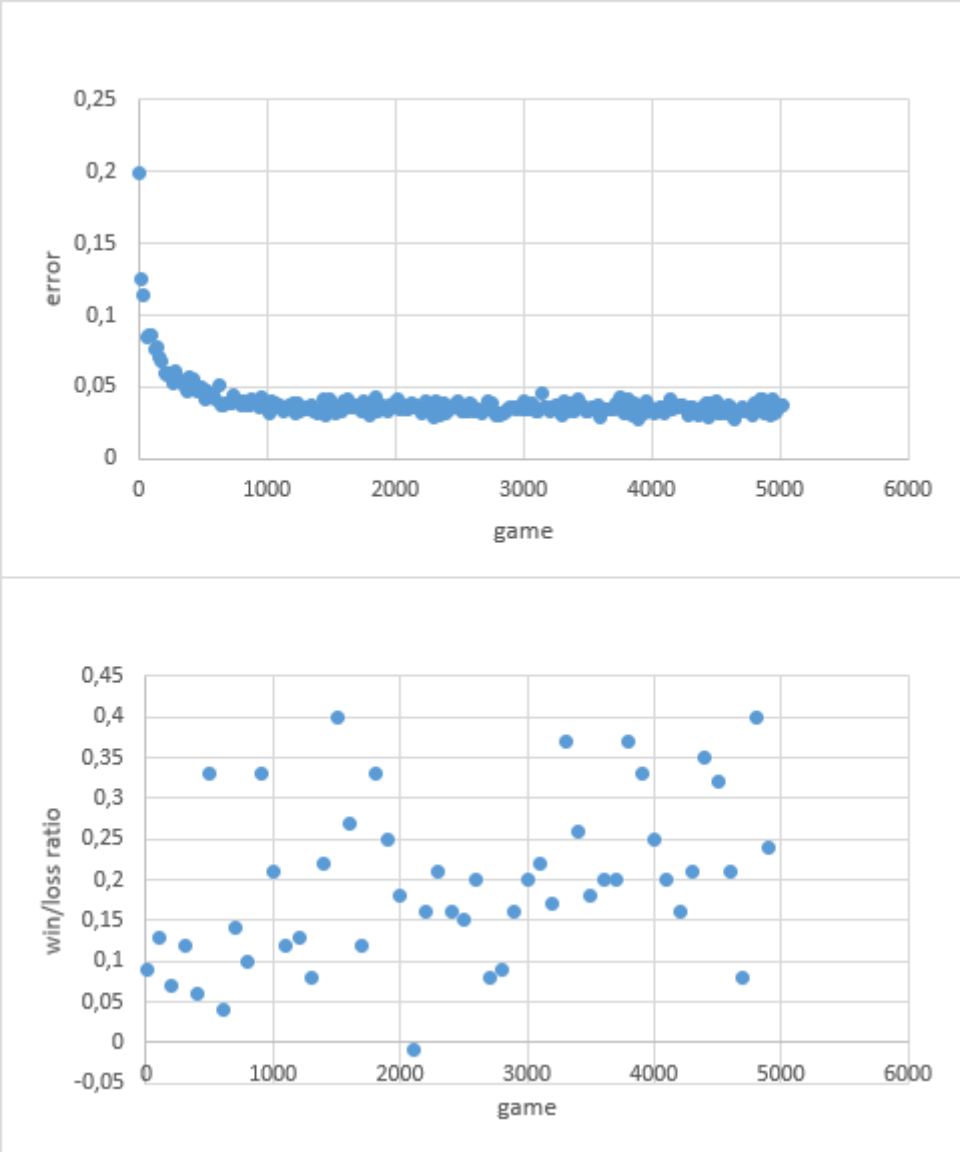


Figur 16 instans 7

Instansen har ”Högt estimerade vägar” satt till 6 istället för 4 som i föregående.
Det är en positiv trend i antal vinster fram till 2000 spel. Error-värden fortsätter minska fram till 4000 spel. Ingen tydlig data vilken påverkan det har för AI:ns prestation.

Instans 8

- 3 hiddenlayer med 128 noder
- Learning rate = 0,1
- Högt estimerade vägar = 6
- Lågt estimerade vägar = 2



Figur 17 instans 8

Instansen har 3 hiddenlayers istället för 2, som i föregående.

En positiv trend för antal vinster fram till 1500 spel. Datan där efter är något otydlig.

Analys

Ingen instans har en stabil ”win/loss ratio” > 0,4. Värt att förtydliga är att AI:n använder NN:et för att få ett säkrare medelvärde av ett val. Boten applicerar fullständiga simuleringar på alla möjliga drag. Boten får ett resultat från alla drag, medan AI:n får resultat endast från några få utvalda.

Ingen instans lyckas få ner ”error” < 0,03. En anledning till det kan vara att värdet på en nod som används som det faktiska värdet i backpropagation, dvs. det sanna värdet som det estimerade värdet ska tränas till, är inte ett äkta sant värde. Det är endast medelvärdet av de fullständiga simuleringar som utförts. Maximalt antal fullständiga simuleringar på en nod är 50.000. Som tidigare nämnt kan antal spelsituationer efter en nod vara ett ohanterligt stort tal. Därför kan 50.000 fullständiga simuleraringar ge olika medelvärden på samma nod/spelsituation.

Instans 1 och Instans 2 har inte någon speciellt hög ”win/loss ratio”. Dock är ”error” lika låg som många andra mer framgångsrika instanser. Ett NN skulle kunna ha låg ”error” utan att vara framgångsrik i matcher. För en hyfsat låg ”error”, skulle det räcka med att ett NN ser koppling mellan att motståndaren har fler poäng och att det är hög risk för förlust.

Instans 3 och Instans 4 har en tydlig skillnad i prestation. Instans 4 använder sig av 2 hiddenlayers istället för 1, och har en positiv ”win/loss ratio” väldigt tidigt i träningen. Ett antagande är att mer komplexa kopplingar har skapats med hjälp av ett till hiddenlayer.

Instans 5 visar ett resultat där det inte sker någon förbättring. Problemet låg i koden för NN:et, där summan för alla derivator till en nod användes, istället ska medelvärdet av alla derivator användas. Effekt blir värre ju fler noder som finns i ett lager. I denna algoritm har learning rate justerats för att undkomma problemet.

Instans 6, Instans 7 och Instans 8 har de största NN. Instans 6 avbröts pga. bristande resultat. Instansen kanske hade blivit bättre, men träningen kräver mycket resurser och den här instansen var inte lönsam för att fortsätta.

Instans 7 har fått tillgång till 6 högt estimerade vägar. Detta var gynnsamt jämfört med föregående instans, med 4 högt estimerade vägar. Notera att AI:n har tillgång till endast 50.000 fullständiga simuleringar oavsett hur många vägar som estimeras.

Instans 8 har det största NN och har producerat det absolut bästa resultatet. Träningen är mer än någon tidigare instans. Trots detta är AI:n inte speciellt mycket bättre än boten.

Diskussion

Även om fel hittades i koden för NN:et, anser jag att resultatet ändå är giltigt. Inlärningskurvan hade troligen sett annorlunda ut om implementation för backpropagation var korrekt. Men det hade ändå inte åtgärdat eventuell ”overfitting”.

Vid observation av spel, har det varit lätt att dra slutsatsen att AI:n inte är smart. I många fall är det uppenbart vad som borde göras för att öka sina chanser till vinst, men AI:n väljer ett annat drag som inte gynnar den på något sätt.

Det går inte dra slutsatsen att ”error” i NN:et är resultatet av ”win/loss ratio” och vice versa. Men det är tydligt att NN:et hjälper AI:n att göra bättre beslut än boten. Det betyder att när ”win/loss ratio” är över 0, finns det kopplingar som aktiveras i NN:et, av gynnsamma abstraktioner i spelet. Det är knappt märkbart i Instans 1 med endast 16 noder i ett hiddenlayer, men något existerar även där.

Så långt som dessa tester visar, är det inget som tyder på att AI:n kommer kunna bli en duktig spelare i Truck Delivery. Implementationen verkar inte kunna utveckla gynnsamma spelstrategier. Jag tror att bristen ligger i MCTS. Slumpmässig sökning verkar inte utveckla NN:et till att förstå komplexa strategier.

Baserat på resultatet finns det nya frågeställningar som är intressanta. För att få en tydligare bild av hur olika instanser presterar, skulle de kunna möta varandra. Det skulle även vara önskvärt att implementera algoritmen med parallellism, för att undersöka om algoritmen kan tränas fortare.

För att ta algoritmen till nästa steg, skulle det gå att utveckla den implementerade metoden av MCTS. Exempelvis att behålla metoden med fullständiga simuleringar, men istället för slumpmässiga vägar, tillämpas NN:et vid varje steg för guidning. Metoden skulle fortfarande vara mer lättdriven än AlphaZeros MCTS, eftersom metoden att lagra varje spelsituation i minnet, ej implementeras.

Referens

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis. (2017).

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

DeepMind, 6 Pancras Square, London N1C 4AG.

Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton. (2012)

A Survey of Monte Carlo Tree Search Methods

IEEE Transactions on Computational Intelligence and AI in Games

Häggmark, Johan. 2018. Teckenigenkänning

https://github.com/JohanHaggmark/Digit-Recognition-Java

Häggmark, Johan. 2018. Truck Delivery

https://github.com/JohanHaggmark/TruckDelivery