# Självlärande Othello-spelare

Kan en dator lära sig att spela Othello?

K L A S  B J Ö R K Q V I S T
o c h  J O H A N  W E S T E R


**KTH Datavetenskap
och kommunikation**

Examensarbete
Stockholm, Sverige 2010

# Självlärande Othello-spelare

Kan en dator lära sig att spela Othello?

K L A S  B J Ö R K Q V I S T
o c h  J O H A N  W E S T E R

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Johan Boye
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
bjorkqvist_klas_OCH_wester_johan_K10026.pdf

Kungliga tekniska högskolan
*Skolan för datavetenskap och kommunikation*

**KTH** CSC
100 44 Stockholm

URL: www.kth.se/csc

# Referat

Kan en dator som bara har kunskap om ett spels regler lära sig att spela spelet enligt en framgångsrik strategi? Det är vad vi har undersökt i den här rapporten. Vi beskriver hur vi genom att använda Q-learning och artificiella neurala nätverk har skapat spelare som utan a priori-kunskap om strategi lär sig Othellostrategi genom att spela mot sig själva. Vi har experimenterat med ett antal olika parametrar för de neurala nätverken för ta fram en så bra spelare som möjligt.

Vi jämför sedan dessa spelare mot ett par hårdkodade strategier med goda resultat för våra självlärda spelare.

# Abstract

## Self-learning Othello player

Is it possible for a computer which only has knowledge of the rules of a game, to learn how to play the game successfully? In this report we attempt to do just that. We describe how we used Q-learning and artificial neural networks to create players who learns Othello strategy by playing against themselves. We try different neural network parameters in order to detemine how to create the best possible player.

The trained players are then evaluated against a couple of hardcoded strategies of different skill levels with very positive results.

# Förord

Den här rapporten skrivs som en del i kandidatexamensarbetet vid CSC på Kungliga Tekniska Högskolan. Arbetet har utförts av Klas Björkqvist och Johan Wester, med handledning av Johan Boye.

Arbetet har till stor del bestått av två delar: rapportskrivning och programmering av algoritmer för att träna och testa våra spelare. Vi har delat upp större delen av rapportskrivningen och programmeringen så att vi båda fått göra lika mycket. Vissa delar har vi skrivit tillsammans, t.ex. tolkningen av resultaten från våra undersökningar.

Ett stort tack till vår handledare Johan Boye för den utomordentliga hjälp han har givit oss under arbetets gång.

# Innehåll

# Del I

# Inledning

# Kapitel 1

# Introduktion

Datorer har sedan de först skapades fruktats av många människor. Otaliga böcker och filmer handlar om hur datorer med sitt kalla, logiska tänkande förslavar eller helt enkelt förintar mänskligheten. Datorer är idag trots stora framsteg inom AI inte i närheten smarta nog att göra en sådan form av revolution, då de bara är slavar under programmerarens fullständiga makt. Det datorer behöver är en möjlighet att lära sig att i någon mening tänka själva.

Det kanske bästa sättet att lära datorer att tänka är genom att lära dem att spela spel. Spel har väldefinerade regler och tydliga mål, vilket gör det relativt lätt att lära sig. Vi kommer därför att undersöka om en dator kan ta fram en egen taktik för att spela Othello. En liknande undersökning gjordes 2005 av van Eck och van Wezel vid Erasmus University Rotterdam [1].

## 1.1 Problemformulering

I den här rapporten avser vi att undersöka om man kan lära ett program att spela Othello med hjälp av reinforcement learning och artificiella neurala nätverk. För att avgöra hur bra vår självlärande spelare är kommer vi att jämföra den med två förprogrammerade strategier för Othello. Den ena strategin vi kommer jämföra med går ut på att i varje drag vända maximalt antal brickor. Den andra går ut på att i varje drag försöka få så många säkra brickor som möjligt, d.v.s. brickor som inte kan vändas tillbaka. Förutom dessa två strategier kommer vi även att undersöka hur vårt program står sig mot en spelare som i varje omgång slumpar fram sitt drag.

# Kapitel 2

# Bakgrund

## 2.1 Othello

Brädspelet Othello tros härstamma från England och uppfanns någon gång i slutet av 1800-talet. Då gick spelet under namnet Reversi. I spelets ungdom fanns inga officiella regler och därför har det tvistas om när och av vem som spelet egentligen uppfanns. Ungefär 20 år efter att spelet blivit känt dog det ut och glömdes så småning om bort. Först år 1971 väckte japanen Goro Hasegawa liv i spelet igen genom att patentera det under namnet Othello och införa de officiella reglerna som gäller än idag. [2]

### 2.1.1 Regler

Othello är ett spel mellan två spelare som spelas på ett 8x8-rutors bräde. Som spelpjäser används 64 stycken två-färgade brickor med en svart och en vit sida. Den ena spelaren har vita brickor och den andra svarta. Målet med spelet är att vända motståndarens brickor och vinnaren är den spelaren som har flest brickor när spelet är slut. Spelet är slut när ingen av spelarna längre kan vända någon av motståndarens brickor. Reglerna för att vända motståndarens brickor är enkla: det gäller att omringa dem. Det går till genom att placera en bricka så att motståndarens bricka/brickor hamnar mellan den just placerade brickan och en annan av spelarens brickor. Vändningar får ske horisontellt, vertikalt och diagonalt. Om en spelare under sin tur inte kan vända några brickor så går turen över till motståndaren. Varje parti startas med att fyra brickor placeras i mitten, två svarta på ena diagonalen och två vita på den andra. Svart gör det första draget. [2]

## 2.2 Reinforcement learning

För att en dator ska kunna spela ett spel räcker det med att den kan spelets regler. Men för att motståndaren ska få något utav en utmaning krävs dessutom att datorn utrustas med någon form av strategi för hur den ska vinna. Det finns flera sätt att

göra det på. En metod är att programmeraren hårdkodar en strategi i programmet som datorn då alltid kommer att spela efter. För just Othello skulle en sådan strategi kunna vara att i varje omgång vända så många brickor som möjligt. En annan metod är att låta programmet själv ta fram en strategi för hur man bäst spelar spelet. Reinforcement learning är en metod för att åstadkomma detta. [3]

Genom att använda reinforcement learning har Tesauro lyckats skapa ett program som spelar Backgammon på samma nivå som de bästa i världen[4].

### 2.2.1 Metoden

Grundtanken för reinforcement learning är ganska enkel: gör programmet något bra så får det någon form av belöning och gör det något dåligt så blir det bestraffat. Från den feedbacken försöker metoden sedan att välja de drag som maximerar den totala belöningen fram till dess att uppgiften är slutförd, i vårt fall när ett parti är färdigspelat. Viktigt att förstå är att metoden inte får någon som helst information om hur den ska förbättra sig. Istället blir den bara tillsagd om den har gjort något bra eller något dåligt och försöker i fortsättningen att upprepa respektive att undvika samma beteende. Man brukar prata om en agent och en omgivning. Agenten är den lärande delen och omgivningens uppgift är att beskriva de olika tillstånden och de tillhörande belöningarna. I varje tillstånd finns det ett antal drag som ger en viss belöning och leder till ett nytt tillstånd. Agentens uppgift är att välja drag.

Att välja drag är en viktig del av metoden och kan göras på många olika sätt. Regler för att välja ut vilket drag som ska spelas brukar i sammanhanget kallas för policy.

En enkel policy är att med en viss sannolikhet välja ett annat drag än det som förväntas vara det bästa. Detta för att undersöka om det kan finnas drag som ger större total belöning än vad som förväntas i det nuvarande tillståndet. Med facit i hand sparas resultatet för att agenten i framtiden ska kunna göra ett bättre beslut. Det handlar om att göra en avvägning mellan att utnyttja den kunskap som redan finns och att låta programmet utforska för att försöka hitta bättre lösningar. Vanligt är att i början låta sannolikheten för att välja andra drag vara ganska hög och att allt eftersom programmet lär sig sänka den. När programmet ska användas på riktigt är det lämpligt att alltid välja det drag som förväntas vara bäst. På så sätt kan programmet utnyttja sin kunskap till fullo. Den svåra delen av metoden är att ta fram en lämplig värdefunktion, d.v.s. funktionen för den förväntade totala belöningen i varje tillstånd.

Värdefunktionen kan skrivas som

$$V(s) = E(r|s) \tag{2.1}$$

där högerledet är väntevärdet för den förväntade totala belöningen $r$ givet att vi befinner oss i tillstånd $s$.

Värdefunktionen byggs upp successivt genom att låta programmet spela många partier mot sig själv och efter varje drag uppdatera värdena. Vi använder oss av Q-learning för att göra detta. [3]

### 2.2.2 Q-Learning

Det finns flertalet kända algoritmer som implementerar reinforcement learning, den vi kommer att använda går som sagt under namnet Q-Learning. I Q-learningalgoritmen skrivs värdefunktionen för $s$ som $V(s) = \max_{\forall a}(Q(s,a))$. $Q$-funktionen uppdateras enligt:

$$Q(s,a) = Q(s,a) + \mu\left(r + \gamma \max_{a' \in A}\left(Q(s',a') - Q(s,a)\right)\right) \tag{2.2}$$

och ger den förväntade totala belöningen i tillstånd $s$ om drag $a$ spelas. $r$ är belöningen man får för att göra draget $a$, $A$ är alla möjliga drag i tillstånd $s'$ som uppstår efter att drag $a$ har spelats, $\mu$ är inlärningshastigheten och $\gamma$ är en skalningsfaktor som tar hänsyn till osäkerheten kring framtida belöningar. En mer utförlig beskrivning av dessa parametrar ges i avsnitt 3.2.1. [3]

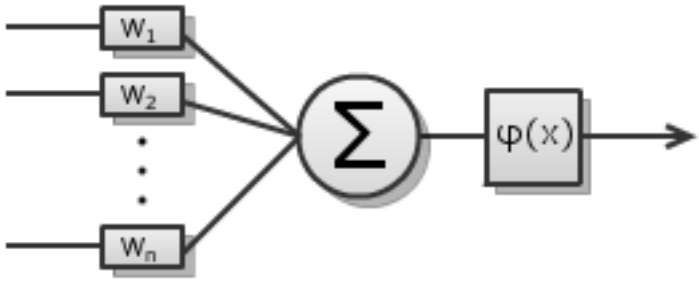
**Algorithm 1** Q-Learning algoritmen

Sätt $Q(s,a)$ till små slumpade värden för alla tillstånd $s$ och för alla möjliga drag $a$
**for** alla träningsomgångar **do**
  Initiera $s$
  **while** träningsomgången inte är slutförd **do**
    Välj drag $a$ med hjälp av någon policy
    Utför drag $a$ och mottag belöning $r$
    Drag $a$ leder till tillstånd $s'$
    $Q(s,a) \leftarrow Q(s,a) + \mu\left(r + \gamma \max_{a' \in A}\left(Q(s',a') - Q(s,a)\right)\right)$
    $s \leftarrow s'$
  **end while**
**end for**

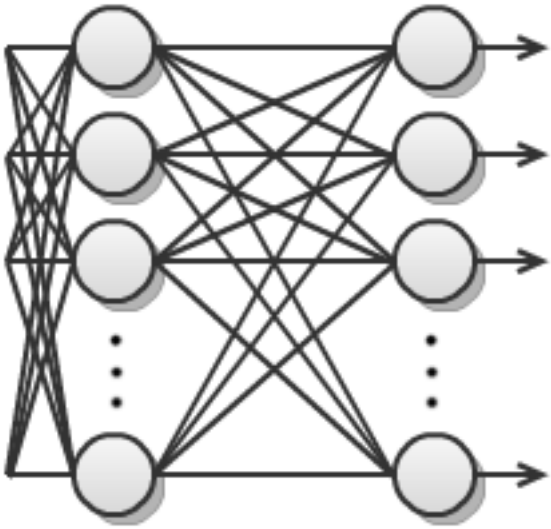
## 2.3 Artificiella Neurala Nätverk

Artificiella neurala nätverk (ANN) är en typ av beräkningsmaskiner. Inspirationen bakom ANN kommer från de biologiska system som finns i hjärnan hos människor och andra djur. Grundtanken är att ha flera enkla beräkningsenheter, neuroner, som helt enkelt summerar viktade ingångsvärden $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Resultatet av summationen går sedan genom en aktiveringsfunktion $\varphi(x)$, som bestämmer vilket värde neuronen skickar ut. Vanligt är att vilja ha 1 och 0, eller 1 och $-1$. En sån enhet brukar kallas perceptron. Se figur 2.1 för en grafisk illustration. Beräkningen perceptronen utför beskrivs av följande formel:

$$f(\mathbf{x}) = \varphi\left(\sum_{i=0}^{n} w_i \cdot x_i\right) \tag{2.3}$$




**Figur 2.1.** En perceptron.



**Figur 2.2.** Ett flerlagersnätverk.

Man bestämmer vad ett nätverk ska göra genom att bestämma värden på vikterna **w** som multipliceras med indatan. Man kan träna en ensam perceptron till att t.ex. göra klassificering, men endast om klasserna man försöker bestämma är linjärt separerbara. [3]

Det finns flera olika typer av neurala nätverk, den typen som är intressant för oss är flerlagersnät. Där består nätverket av flera lager med perceptroner, som här kallas noder, där utgångsvärden från det första lagret blir ingångsvärden till det andra lagret o.s.v. Se figur 2.2. Genom att lägga till fler lager slipper man undan begränsningen att vid klassificering behöva ha linjärt separerbara data. Man kan redan vid två lager klassificera klasser av godtycklig form, så länge man har tillräckligt många noder i det första lagret. Detta innebär möjlighet till andra typer av tillämpningar som t.ex. funktionsanpassning.

Att lägga på flera lager innebär dock inte bara fördelar. Det blir krångligare och tar längre tid att träna nätet. Den vanligaste metoden för att träna ett flerlagersnät kallas Error Backpropagation, eller bara *BackProp* [3]. Förenklat kan man säga att algoritmen tar skillnaden mellan det svar som nätet ger och det förväntade svaret och uppdaterar vikterna för att hamna närmare det rätta svaret. Eftersom man inte kan veta vilket lager det är som orsakar felet propagerar man felet bakåt genom alla lager. Genom att upprepa algoritmen flera gånger lär man nätverket att svara med vissa värden för viss indata.

Den kanske trevligaste egenskapen hos artificiella neurala nätverk är deras egenskap att generalisera över indata. Genom att bara träna ett nätverk på en liten delmängd av t.ex. värden hos en funktion, kan man få nätverket att återskapa hela funktionen. Den här egenskapen är väldigt användbar inom reinforcement learning, där man ofta har väldigt stora tillståndsrum, och inte har möjlighet att bestämma en värdefunktion för alla möjliga tillstånd.

## 2.4 Termer

**Reinforcement learning** Sv. ”Belöningsbaserad inlärning”. En metod för att programmera självlärande system. Den engelska termen är betydligt mer vedertagen och kommer därför att uteslutande användas i den här rapporten. Beskrivs i avsnitt 2.2

**Q-learning** En typ av reinforcement learning.

**Policy** Regler för att välja drag.

**ANN** Artificiellt neuralt nätverk, en beräkningsmaskin beskriven i avsnitt 2.3.

**Neuron** En beräkningsenhet i ett artificiellt neuralt nätverk. Kallas även nod.

**Perceptron** En typ av neuron för artificiella neurala nätverk, mycket förenklad modell av en biologisk neuron.

**Gömt lager** Alla lager utom det sista i ett flerlagersnätverk.

**BackProp** En metod för att träna flerlagers ANN.

# Del II

# Metod

# Kapitel 3

# Nätverk och Träning

Den kanske vanligaste metoden att implementera Q-learning är med tabeller som sparar värdet av $Q$-funktionen för alla tillstånd och drag. När problemet metoden ska lösa har många tillstånd betyder det att den behöver stora mängder utrymme att spara tabellerna på. I Othello finns omkring $10^{28}$ olika tillstånd[5]. Det betyder att det i praktiken är omöjligt att använda tabeller för att lagra $Q$-funktionen i vårt fall. Vi använder därför ett ANN för att approximera $Q$-funktionen. Då behöver vi endast lagra vikterna till nätverket. Vi får även nätverkets egenskap att generalisera över indata på köpet, vilket bör hjälpa vår algoritm att lära sig en bra strategi utan att spela alla möjliga spel.

## 3.1 Nätverket

Ett flerlagersnätverk är en global funktionsapproximerare och kan alltså approximera vilken funktion som helst. Det enda som krävs är att det finns tillräckligt många noder i de gömda lagren. Det räcker med att ha ett tvålagersnät, d.v.s. ett gömt lager och ett ut-lager, om man har tillräckligt många noder i det gömda lagret. Att lägga på fler lager ökar nätverkskomplexiteten. Därför ävänds här ett tvålagernätverk för att approximera $Q$-funktionen.

Vi kommer att undersöka två representationer av indata till nätet.

**QL1** Indatan till nätverket är en vektor med 64 komponenter ur mängden $\{-1, 0, 1\}$. Varje komponent motsvarar en ruta på spelbrädet och har värdet 0 om ingen bricka ligger där, $-1$ om motspelarens bricka ligger där och 1 om spelarens egen bricka ligger där.

**QL2** Indatan är en vektor med 129 komponenter. De första 64 komponenterna motsvarar svart spelares brickor, 1 om spelaren har en bricka på motsvarande ruta på spelbrädet, $-1$ annars. De nästkommande 64 komponenterna motsvarar vit spelares brickor på samma sätt. Den sista komponenten är 1 om det är svart spelares tur och $-1$ om det är vit spelares tur.

Utdatan är i båda fallen en vektor med 64 komponenter som motsvarar dragen. Värdet av en komponent approximerar värdet av funktionen $Q(s, a)$ för tillståndet $s$ som beskrivs av indatan samt det drag $a$ komponenten motsvarar.

Vi undersöker två olika aktiveringsfunktioner $\varphi(x)$:

$$\varphi(x) = \frac{2}{1 + e^{-x}} - 1 \tag{3.1}$$

$$\varphi(x) = \arctan x \tag{3.2}$$

Tanken är att se om olika aktiveringsfunktioner påverkar hur bra och snabbt programmet lär sig spela.

Eftersom det inte finns någon metod för att fastställa det bästa antalet gömda noder i ett nätverk undersöks här flera möjligheter. De antal som undersöks är 40 och 64 gömda noder. I kommande kapitel hänvisas antalet gömda noder som H40 respektive H64.

## 3.2 Träning

### 3.2.1 Q-learning

Vår implementation av Q-learning beskrivs i algoritm 2.

**Algorithm 2** Q-learning med ANN

Initiera nätverket $QNET$ med slumpade vikter
**for** alla träningsomgångar **do**
  Initiera tillståndet $s$
  **repeat**
    $Q_s(a) \leftarrow QNET$.evaluate($s$) för alla drag $a$
    Välj drag $a_s$ med policy $\pi_{sm}$ {se 3.2.3}
    Utför drag $a_s$
    Observera nya tillståndet $s'$ {efter att motspelaren har gjort sitt drag}
    $Q_s(a) \leftarrow QNET$.evaluate($s$) för alla drag $a$
    $Q_{s'}(a') \leftarrow QNET$.evaluate($s'$) för alla drag $a'$
    $Q_s(a_s) \leftarrow Q_s(a_s) + \mu \cdot \max_{a'}(Q_{s'}(a') - Q_s(a))$
    $QNET$.train($s, Q_s$) {Träna nätverket med $Q_s$ som mål (se 3.2.2)}
  **until** träningsomgången är slut
  $Q_s(a) \leftarrow QNET$.evaluate($s$) för alla drag $a$
  Mottag en belöning $r$
  $Q_s(a_s) \leftarrow Q_s(a_s) + \mu \cdot r$
  $QNET$.train($s, Q_s$) {Träna nätverket med $Q_s$ som mål (se 3.2.2)}
**end for**

$Q_s(a)$ är en vektor med utdata från nätverket, d.v.s alla $Q(s, a)$ för ett tillstånd $s$.

I algoritmen finns det två parametrar: inlärningshastigheten $\mu$ och belöningen $r$.

Inlärningshastigheten skalar ner värdet som $Q$-funktionen uppdateras med. Ett för stort värde på $\mu$ innebär att algoritmen inte tar någon större hänsyn till tidigare värdet när den uppdaterar $Q$-funktionen som således blir väldigt beroende av resultatet från den senaste träningsrundan. Skulle $\mu$ istället sättas för lågt kommer det leda till att agenten lär sig väldigt långsamt, om $\mu = 0$ så lär den sig inget. Vi har satt ett fixt värde $\mu = 0.1$.

Belöningens syfte är som nämnts tidigare att tala om för algoritmen om den har gjort något bra eller dåligt, vunnit eller förlorat. Vi har valt att dela ut belöningen $r = 1$ vid vinst, bestraffa med $r = -1$ vid förlust och att vara neutrala vid ett oavgjort resultat, $r = 0$.

I algoritm 1 finns även skalningsfaktorn $\gamma$ som används för att handskas med osäkerheten kring framtida belöning, ju längre bort den framtida belöningen finns desto mer osäker är den. Detta beror naturligtvis på att vi inte vet hur motståndaren kommer att spela. Parametern kan anta värden mellan 0 och 1. En belöning som förväntas komma efter $n$ drag skalas med faktorn $\gamma^n$. Eftersom belöningen i vårt Othello-spel bara delas ut efter det att partiet avslutas är framtida belöningar det enda vi bryr oss om och därför har vi satt $\gamma = 1$. Vårt val av $\gamma$ innebär att vi kan bortse från parametern i algoritmen, vilket vi också har gjort för att göra den mer lättbegripligt.

Utöver dessa parametrar är en viktig del av Q-learning algoritmen antalet träningsomgångar. Vi har utgått ifrån att det behövs 10 miljoner spelade partier innan QL-spelaren slutar att lära sig. I de fall där vi sett att utvecklingskurvan fortfarande pekar tydligt uppåt vid träningens slut har vi valt att träna med ytterligare 10 miljoner partier. För träningar som gjorts mot MSB-spelaren har vi nöjt oss med att träna 2.5 miljoner partier som svart följt av 2.5 miljoner partier som vit.

### 3.2.2 ANN

För att träna nätverket används BackProp, som finns väl beskriven i [3]. Vår erfarenhet av att träna nätverk med BackProp visar att ett bra värde på inlärningshastigheten för nätverket är 0.1. Notera att det här är inlärningshastigheten för nätverket, vilket inte är samma parameter som den för Q-learningalgoritmen. Vi använder även en förbättring av BackProp som kallas momentum, vilket betyder att viktuppdateringarna behåller en stor del av deras senaste värde:

$$dW = \alpha \cdot dW + (1 - \alpha) \cdot nW$$
$$W = W + dW$$

där $W$ är en vikt, $dW$ är den senaste viktuppdateringen, $nW$ är den nya viktuppdateringen och $\alpha$ är en konstant. Vi använde $\alpha = 0.9$.

### 3.2.3 SoftMax

För att välja vilket drag som ska göras under träning använder vi en SoftMax-policy[3] $\pi_{sm}$ för sannolikheten att välja ett visst drag $a$:

$$P(a) = \frac{\exp(\beta Q_s(a))}{\sum_{a' \in A} \exp(\beta Q_s(a'))} \tag{3.3}$$

där $A$ är alla möjliga drag och $\beta$ är en parameter. För ett litet värde på $\beta$ väljs draget slumpmässigt, när $\beta$ växer ökar sannolikheten att det drag med högst $Q_s$-värde väljs. Vi låter $\beta$ växa linjärt under träning enligt

$$\beta = \frac{i}{C}$$

där $C$ är en konstant och $i$ börjar på 0 och inkrementeras efter varje parti. Vi har valt att använda $C = 3\ 000\ 000$ när vi tränar QL-spelarna mot sig själva och $C = 1\ 500\ 000$ när vi tränar mot MSB (se 4.1.2). Vårt val av $C$ innebär att vi får en policy som i början väljer drag slumpmässigt och som fram emot slutet av träningen med större och större sannolikhet väljer det hittills bästa draget.

När algoritmen sedan evalueras väljs alltid det drag med högst $Q_s$-värde.

# Kapitel 4

# Testning

## 4.1 Jämförande algoritmer

### 4.1.1 Maximera antalet vända brickor (MVB)

Den här strategin går ut på att i varje spelomgång välja det drag som resulterar i flest vända brickor. MVB tar inte någon hänsyn till om det valda draget ger motspelaren möjligheten att vända tillbaka ännu fler brickor.

### 4.1.2 Maximera antalet säkra brickor (MSB)

MSB-spelarens strategi är inte fullt lika naiv som MVB:s, här väljs det drag som resulterar i flest brickor som är omöjliga för motståndaren att vända tillbaka. Att avgöra vilka brickor som är säkra är för blotta ögat ett relativt enkelt problem, men att implementera en algoritm för det är mer komplext. Vi har tagit fram en heuristik för att lösa problemet och kan alltså inte garantera att vi alltid väljer det drag som ger flest säkra brickor.

För att en bricka ska vara säker gäller det att motståndaren i framtiden aldrig kan placera en bricka på samma rad, kolumn eller någon av de två diagonalerna för att vända den. För att se om så är fallet kontrollerar vi de fyra riktningarna var för sig. Först kollar vi om riktningen är full. Den andra kontrollen vi gör är att se om någon av de två närliggande positionerna är säkra och tillhör spelaren själv, skulle en av dessa vara utanför spelbrädet klassas den som säker och som om den tillhör spelaren. Till sist kollar vi om båda närliggande brickor är säkra och tillhör motståndaren. Uppfylls någon av dessa tre tester för samtliga riktningar har vi en säker bricka. Vår heuristik är implementerad efter denna princip och kommer alltså aldrig att returnera fler säkra brickor än vad som finns. Det finns dock möjlighet att säkra brickor inte upptäcks.

I de fall där det inte går att säkra några brickor används MVB-spelarens strategi.

### 4.1.3 Slump

Slumpspelaren är inte utrustad med någon som helst strategi. Vid varje spelomgång slumpas ett drag fram ur mängden av alla tillåtna.

### 4.1.4 Jämförelsealgoritmer mot otränade spelare

Tabell 4.1 visar vinstprocenten mot våra jämförelsealgoritmer för QL1 och QL2 med aktiveringsfunktion 3.2 och 64 gömda noder innan de tränats. Här syns det att MSB är bättre än MVB som i sin tur spelar bättre än slumpen.

**Tabell 4.1.** Vinstprocent innan träning

| | MSB | | MVB | | Slump | |
|---|---|---|---|---|---|---|
| | Vit | Svart | Vit | Svart | Vit | Svart |
| QL1 | 5.6 | 5.5 | 32.4 | 32.2 | 44.0 | 43.8 |
| QL2 | 13.5 | 6.1 | 51.8 | 30.6 | 56.4 | 54.1 |

## 4.2 Testmetod

Vi har testat vår implementerade Q-Learningspelare mot de tre ovanstående algoritmerna genom att spela 10 000 partier mot respektive algoritm. Under testningen har vi valt att för Q-Learningspelaren alltid välja det drag som ger högst förväntad belöning. Det betyder att den är deterministisk under testning. Eftersom både MVB och MSB också är deterministiska algoritmer kommer varje parti att bli identiskt.

För att kringgå det problemet har vi valt att slumpa fram de första 5 dragen i varje parti, detta ger oss tillräckligt många unika partier för att kunna dra slutsatser om vår spelare lärt sig något. För slumpspelaren är detta naturligtvis inte något problem.

Det finns ett antal parametrar vi kan variera för träningen som kan påverka hur bra spelaren blir:

- Representationen av indata. Vi har två olika sätt att representera indata som vi testar: QL1 och QL2. Se avsnitt 3.1 för en djupare beskrivning.
- Olika aktiveringsfunktioner i nätverket. Vi undersöker två olika, funktionerna 3.1 och 3.2. Se avsnitt 3.1.
- Antalet gömda noder i nätverket. Det påverkar nätverkets möjlighet att lära sig lösa uppgifter. Vi undersöker två antal, 40 och 64 gömda noder.
- Svart eller vit spelare. Vi testar våra QL-spelare som både svart och vit.
- Inlärningshastigheter för nätverket och Q-learningalgoritmen. Vi har valt att låta båda spelarna använda ett fixt värde på 0.1 i hela undersökningen.

- Policyn för att välja drag. Vi har valt att låta samtliga träningar använda samma SoftMax-policy, se 3.2.3.
- Träningsmotståndare. Vi testar att träna spelarna både mot sig själv och mot vår bästa jämförelsealgoritm, MSB.
- Skalningsfaktorn $\gamma$ för Q-learningalgoritmen. Vi använder alltid $\gamma = 1$, se 3.2.1.

# Del III

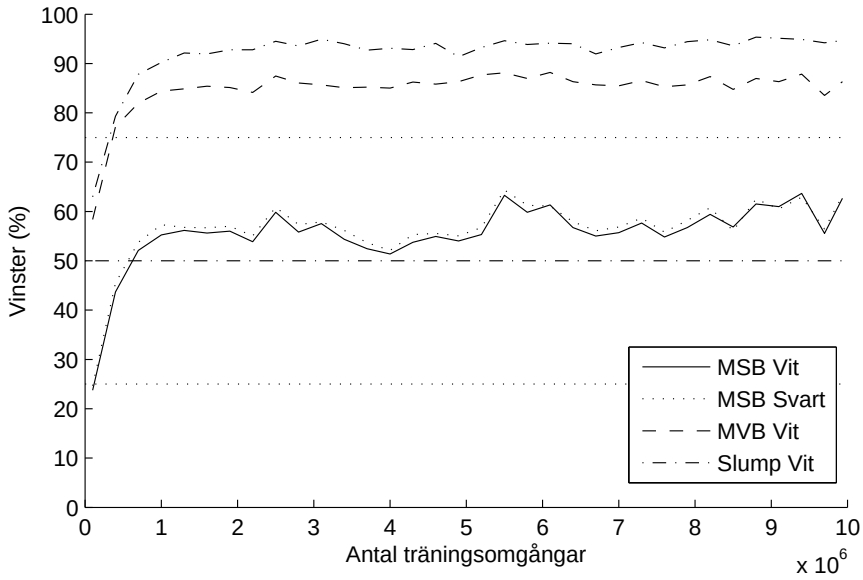
# Resultat

# Kapitel 5

# Resultat

Vid träning sparas nätverkets vikter var 100 000:e iteration. Vi testar sedan alla sparade vikter 10 000 gånger mot MSB, MVB och Slump. Vi redovisar QL-spelarnas vinstprocent mot dessa motståndare i grafer. För att öka läsbarheten har vi valt att använda medelvärdet av tre datapunkter från testningen för att skapa varje datapunkt som ritas. Det bibehåller de stora trender som finns och gör graferna mycket mer lättlästa.

## 5.1 QL1

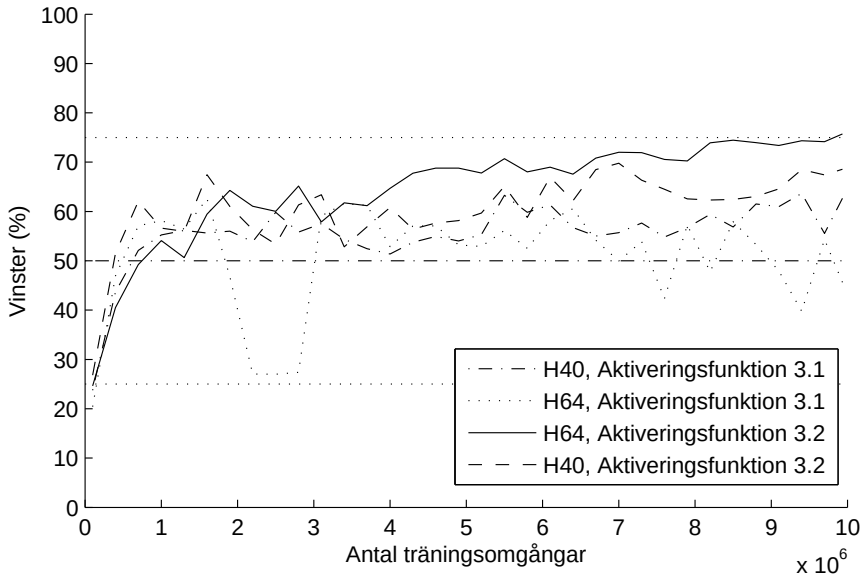



**Figur 5.1.** Vinstprocent för QL1 efter 10 miljoner träningar. ”MSB vit” betyder att MSB spelar som vit.

Figur 5.1 visar resultaten efter 10 miljoner träningsomgångar med QL1-representationen av indata, aktiveringsfunktionen 3.1 och 40 gömda noder, testad mot våra jämförelsealgoritmer.

Vi ser nivåerna jämförelsealgoritmerna spelar på: slumpspelaren är lättare att slå än MVB-spelaren, som är betydligt lättare att slå än MSB-spelaren. Vi ser också att resultaten blir nästan identiska för QL-spelaren som både svart och vit.

## 5.2 QL1 mot MSB



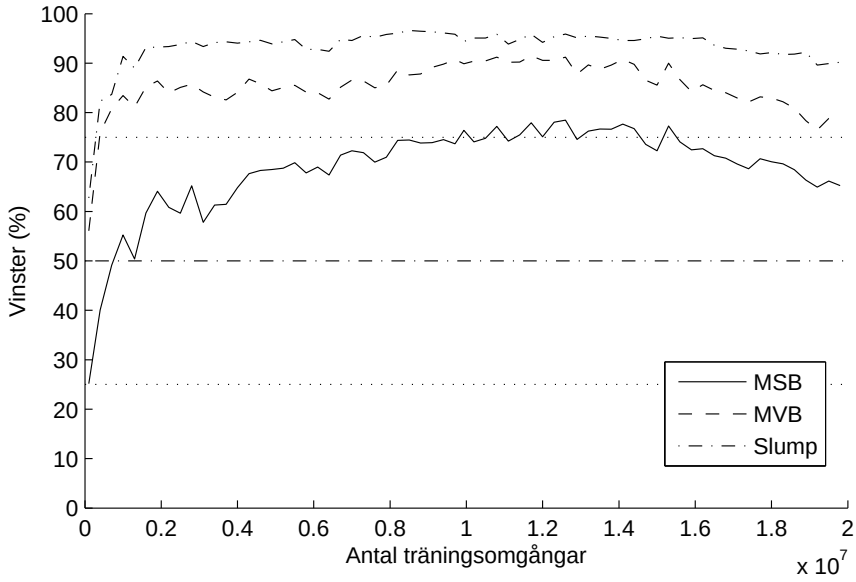

**Figur 5.2.** Vinstprocent för QL1 efter 10 miljoner träningar.

Figur 5.2 visar resultaten för QL1 för våra olika parametrar. QL1-spelaren spelar som svart i samtliga fall.

Vi ser tydligt att spelarna med aktiveringsfunktion 3.2 slutar med ett klart bättre resultat. Eftersom den fortfarande visade en uppgående trend valde vi att träna spelaren med 64 gömda noder och aktiveringsfunktion 3.2 ytterligare, se avsnitt 5.3.

På den prickade linjen ser vi en dal mellan två och tre miljoner. Det beror på att filerna som sparade nätverkets vikter av någon anledning var tomma och bör alltså bortses från.

## 5.3 QL1 längre körning

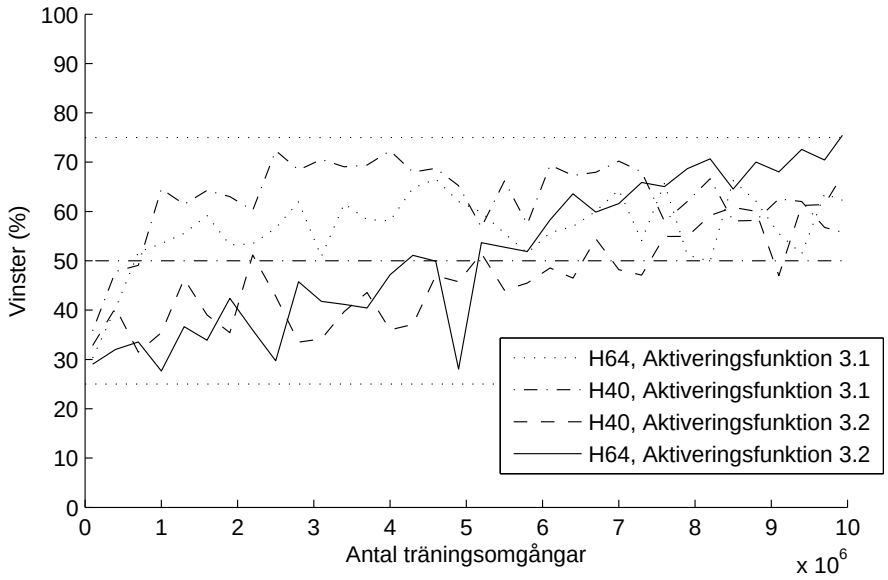



**Figur 5.3.** Vinstprocent för QL1 efter 20 miljoner träningar.

Figur 5.3 visar vinstprocenten för QL1-spelaren med 64 gömda noder och aktiveringsfunktion 3.2 under 20 miljoner träningsomgångar.

Det syns att resultaten når en topp efter 12 miljoner träningsomgångar, varefter den börjar spela sämre.

## 5.4 QL2

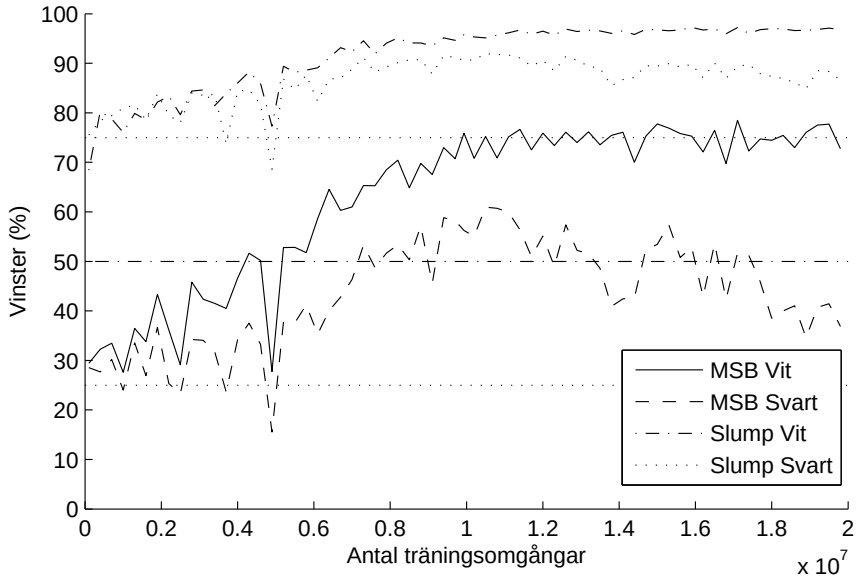



**Figur 5.4.** Vinstprocent för QL2 efter 10 miljoner träningar.

Figur 5.4 visar resultaten efter 10 miljoner träningsomgångar med QL2 som svart testad mot MSB.

Diagrammet visar att det finns en skillnad i hur inlärningen sker beroende på aktiveringsfunktion. De som använder aktiveringsfunktion 3.2 växer långsammare i början men stiger ovanför de med aktiveringsfunktion 3.1 när de närmar sig 10 miljoner träningar. Trenden för 64 gömda noder och aktiveringsfunktion 3.2 var fortfarande lika stark uppåt när den närmade sig 10 miljoner träningsomgångar. Därför valde vi att träna den ytterligare 10 miljoner omgångar, se avsnitt 5.5.

## 5.5 QL2 längre träning

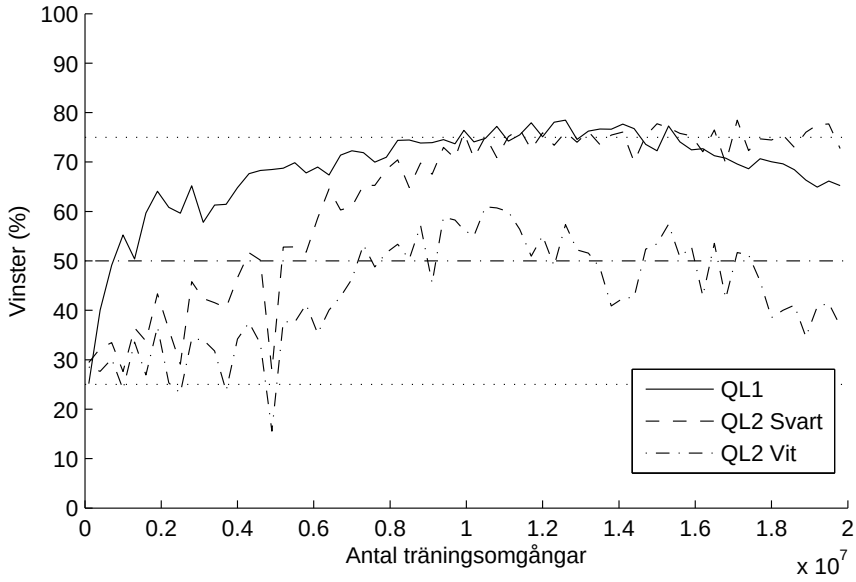



**Figur 5.5.** Vinstprocent för QL2 efter 20 miljoner träningar.

Figur 5.5 visar resultatet för QL2 med 64 gömda noder och aktiveringsfunktion 3.2 efter 20 miljoner träningsomgångar.

Vi ser hur inlärningen verkar avstanna strax efter 10 miljoner träningsomgångar. Efter det verkar den även tappa kunskap, men bara när QL-spelaren är vit.

## 5.6 QL1 jämfört med QL2



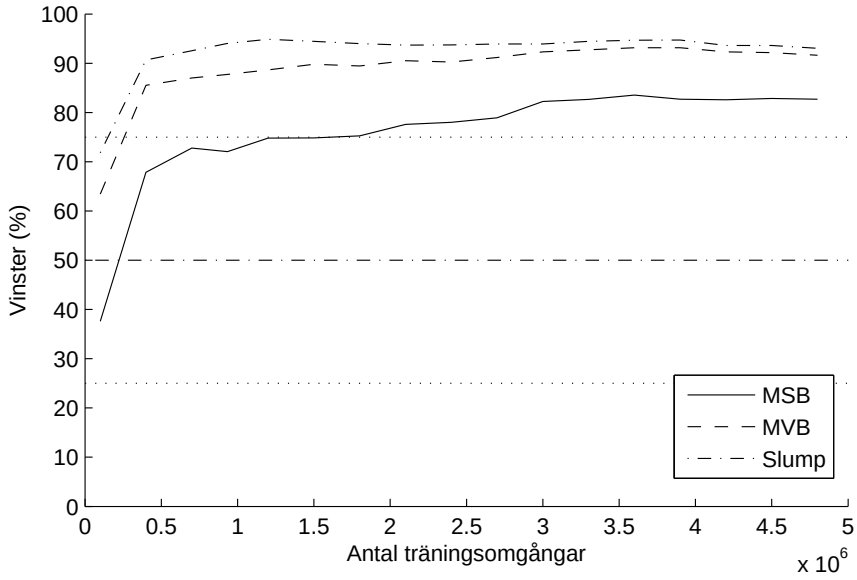

**Figur 5.6.** Vinstprocent efter 20 miljoner träningar.

Figur 5.6 visar resultaten efter 20 miljoner träningsomgångar med QL1 och QL2 jämförda mot MSB. QL1-spelaren visas enbart som svart, eftersom QL1-spelaren spelar ungefär lika bra som svart och vit. QL2-spelaren finns med som både svart och vit. Samtliga träningar är gjorda med 64 gömda noder och aktiveringsfunktion 3.2.

QL2-spelaren lär sig långsammare i början, men når upp till samma nivå som QL1-spelaren efter 10 miljoner träningsomgångar. Vi ser även att QL2-spelaren spelar betydligt sämre som vit.

Både QL1 och QL2 når sin maximala förmåga strax efter 10 miljoner träningsomgångar, med väldigt lika maxresultat.

## 5.7 QL1 tränad mot MSB

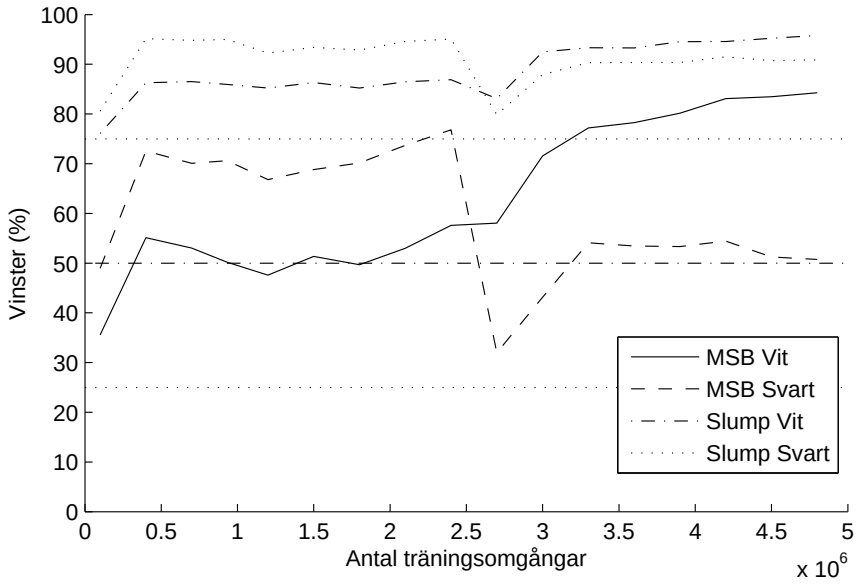



**Figur 5.7.** Vinstprocent efter 5 miljoner träningar.

Figure 5.7 visar resultaten efter 5 miljoner träningsomgångar mot MSB för QL1. Resultaten visas för 64 gömda noder, aktiveringsfunktion 3.2 och Q-learningspelaren som svart. Resultaten för andra parametrar var nästan identiska.

Det märks tydligt att QL1 snabbt lär sig att vinna över MSB-spelaren när den tränats emot den. Även slutresultatet efter 5 miljoner träningsomgångar är betydligt bättre än när QL1 tränats mot sig själv.

## 5.8 QL2 tränad mot MSB




**Figur 5.8.** Vinstprocent efter 5 miljoner träningar.

Figur 5.8 visar resultaten efter 5 miljoner träningsomgångar mot MSB för QL2. De visade resultaten är för 64 gömda noder och aktiveringsfunktion 3.2. Resultaten för andra parametrar gav liknande resultat.

Vi ser här en stor skillnad för QL2-spelaren när den spelar som svart jämfört med vit. Det beror på att de första 2.5 miljoner träningsomgångarna spelas med QL2-spelaren som vit och de sista 2.5 miljoner träningsomgångarna spelas med QL2-spelaren som svart.

# Del IV

# Diskussion

# Kapitel 6

# Diskussion

## 6.1 Val av jämförelsealgoritmer

Vi har som bekant använt oss av tre olika algoritmer för att avgöra om våra QL-spelare har lärt sig något: slump-spelaren, MVB-spelaren och MSB-spelaren. Förhållandet mellan dessa tre inbördes är att MSB-spelaren vinner i princip samtliga matcher mot de andra två. MVB-spelaren i sin tur vinner ungefär två tredjedelar av matcherna mot slump-spelaren.

### 6.1.1 Slump-spelaren

Vi hade två tankar med att implementera slump-spelaren. För det första ville vi visa att QL-spelarna, inte bara i teorin utan även i praktiken, spelar helt slumpmässigt i otränat tillstånd. Genom att testa den otränade spelaren mot slump-spelaren såg vi att de vinner lika mycket. Den andra tanken bakom slump-spelaren var att den borde ge oss en tidig indikation på om vår spelare lärt sig något, det krävs inte särkilt mycket till taktik för att vinna majoriteten av alla partier mot någon som spelar slumpmässigt.

### 6.1.2 MVB-spelaren

MVB-spelaren tog vi fram för att se om QL-spelarna skulle klara av att vinna mot någon taktik överhuvudtaget. Förutsatt att QL-spelarna verkligen lyckats att lära sig något räknade vi med att detta skulle bli en enkel match eftersom MVB-spelaren är väldigt naiv och förlorar en tredjedel av alla matcher mot slump-spelaren.

### 6.1.3 MSB-spelaren

MSB-spelaren är den bästa hårdkodade spelaren som vi tagit fram och syftet med den är att ge QL-spelarna en utmaning och utifrån resultatet av matcherna dra våra slutsatser.

## 6.2 Resultaten

### 6.2.1 Tillståndsrepresentation

Vi ser på figurerna 5.6 och 5.1 att det spelar stor roll hur vi representerar tillståndet till nätverket. För QL1-representationen spelar det ingen roll för resultatet om spelaren är svart eller vit. Samtidigt är det stor skillnad mellan svart och vit med QL2-representationen. Det beror med största sannolikhet på att QL1-representationen är ”symmetrisk” för båda spelarna. Indatan till nätverket blir precis samma om man inverterar spelläget, alltså vänder alla brickor och låter turen gå över till den andra spelaren. QL2-representationen däremot blir helt omvänt vid en sådan manöver, vilket betyder att den kan bete sig helt annorlunda beroende på om den är svart eller vit. En annan orsak till skillnaden kan vara att den enda komponenten i QL2 som anger vilken spelares tur det är får för lite vikt i nätverket.

Båda spelarna når sin topp strax efter 10 miljoner träningsomgångar, med en liknande vinstprocent. En skillnad mellan spelarna är att QL1 gör större framsteg med få träningsomgångar, medan QL2 verkar lära sig långsammare.

Slutsatserna vi kan dra från våra resultat är att QL1 är den bättre representationen, eftersom den ger lika bra resultat oavsett om den är svart eller vit spelare.

### 6.2.2 Aktiveringsfunktioner

I figurerna 5.2 och 5.4 märks tydligt att aktiveringsfunktionen spelar en stor roll. För QL1 sker inlärningen i ungefär samma takt för båda aktiveringsfunktionerna fram till ungefär 4 miljoner träningsomgångar, varefter en tydlig skillnad kan märkas till fördel för aktiveringsfunktion 3.2.

För QL2 märks att funktion 3.1 ger en snabbare inlärning med en topp en bit över 70% vinst mot MSB efter ungefär 3 miljoner träningsomgångar. Detta att jämföra med funktion 3.2 där inlärningen går långsammare, men toppen nås efter drygt 10 miljoner träningsomgångar runt 75% (se figur 5.5). Allt verkar tyda på att funktion 3.2 är bättre lämpad för uppgiften.

### 6.2.3 Antal gömda noder

Varken för QL1 eller QL2 märks någon väsentlig skillnad mellan 40 och 64 gömda noder, det syns tydligt i figurerna 5.2 och 5.4. Det tyder på att 40 gömda noder räcker för att approximera $Q$-funktionen. Med 64 gömda noder har resultatet en tendens att svänga mer, så det gäller att vara uppmärksam så att man inte väljer vikter som resulterar i ett lokalt minimum.

### 6.2.4 Träningsmotståndare

I våra tester var spelarna som tränades mot MSB-spelaren de som gav klart bäst resultat mot våra jämförelsealgoritmer (se figur 5.7 och 5.8). Det var väntat eftersom Q-learning lär spelarna att slå den strategi de tränas mot. Det betyder dock inte

att spelaren kan spela bra mot andra strategier. Våra andra jämförelsealgoritmer är emellertid så pass dåliga att vi inte kan uttala oss om hur bra QL-spelaren som tränats mot MSB kan spela mot andra strategier.

Som beskrivs i avsnitt 6.2.1 spelar det stor roll för QL2 om den spelar som svart eller vit. Det syns tydligt i figur 5.8, där man kan se att det sker en stor förändring för spelarna vid 2.5 miljoner när träningen går över från att träna QL-spelaren som svart till vit.

## 6.3 Sammanfattning

Som tydligt framgår i kapitel 5 har våra QL-spelare lyckats lära sig en taktik där de spelar bättre än från början. I de flesta fall har taktikerna dessutom blivit så bra att de slår våra jämförelsealgoritmer med marginal. Samtidigt har vi märkt att den enda av parametrarna vi testade som spelade någon större roll för slutresultatet var aktiveringsfunktionen, där arctangens-funktionen visade sig ge betydligt bättre resultat än den sigmoida funktionen vi jämförde med.

Även träningsmotståndaren spelade stor roll, men vid träning mot samma strategi som vi jämför mot blir inte resultaten lika rättvisande, p.g.a. sättet Q-learning fungerar på.

Som nämndes i introduktionen så har en liknande undersökning gjorts av van Eck och van Wezels vid Erasmus University Rotterdam[1]. I deras rapport finns det dock ett par väsentliga skillnader jämfört med vår.

Först och främst har man använt sig av helt andra typer av jämförelsealgoritmer. De har förutom sina QL-spelare implementerat två spelare, Rörlighets-spelaren (Mobility player) och positions-spelaren (Positional Player). Rörlighets-spelarens taktik är att placera sina brickor kompakt för att på så sätt minska antalet möjliga drag för motståndaren. Tanken med det är att till slut tvinga motståndaren att ge bort hörnen. Positions-spelarens taktik är att placera brickor på strategiskt viktiga positioner som t.ex kanter eller hörn och samtidigt undvika att ge motståndare möjlighet att göra detsamma.

Den andra stora skillnaden som finns mellan rapporterna är att de även testat en implementation av Q-learning där totalt 64 stycken ANN används, ett för varje drag.

Precis som i vår rapport lyckades deras QL-spelare vinna majoriteten av alla partier mot jämförelsealgoritmerna. Deras träningsgrafer visar att även deras QL-spelare gör en del baksteg under träningens gång och att det bästa resultatet inte nödvändigtvis behöver var det från den sista träningen.

Ur deras slutsatser framkommer det att implementationen med 64 nätverk varken blir bättre eller sämre än den med ett. Dock tar den längre tid att träna.

# Kapitel 7

# Avslutningsvis

## 7.1 Felkällor

Vi har identifierat flera möjliga felkällor:

- Genom att endast ha testat varje permutation av aktiveringsfunktion och antal gömda noder en gång, har vi lämnat oss öppna för en risk att våra träningar har fastnat i ett suboptimalt läge. Det kan vara möjligt att träna en spelare som är bättre än de vi har fått fram. Det handlar om hur vikterna i nätverken initieras, vilket i vårt fall sker med slump. Anledningen till att vi inte har testat mer är att en träning tar minst sex timmar på den bästa hårdvaran vi har haft tillgänglig.
- Våra resultat visar att våra QL-spelare lär sig spela bra mot de jämförelsealgoritmer vi testar mot. Värt att påpeka är att det egentligen inte säger någonting om hur bra våra spelare är på att spela mot andra strategier. Det bör även påpekas att de QL-spelare som är tränade mot MSB-spelaren antagligen är sämre än de QL-spelare som tränats mot sig själva om de ställs mot andra taktiker. Detta eftersom QL-spelarna som är tränade mot sig själva bör ha tagit fram en mer generell taktik, medan den som är tränad mot MSB-spelaren lär sig att specifikt slå den taktiken.
- Vid testning slumpas de första fem dragen för varje spelare. Det betyder att den ena spelaren t.ex. kan ha fått ett hörn eller någon annan strategisk fördel innan spelarna ens börjar spela.

## 7.2 Slutsatser

Vi ställde i början frågan: går det att lära en dator att spela Othello? Nu kan vi med säkerhet svara att det är fullt möjligt.

Våra QL-spelare har lyckats lära sig att besegra samtliga jämförelsealgoritmer. Den kanske bästa förbättringen gjordes mot MSB-spelaren, där QL-spelarna gick från att innan träning förlora 95% av partierna till att vinna upp mot 80% av dem.

När vi själva har spelat mot QL-spelarna märker vi en tydlig förbättring efter träningen. Innan träningen kunde vi lätt vinna, oftast efter bara ett tiotal drag. Efter träningen besegrar vi den fortfarande utan större problem, men nästan aldrig innan spelbrädet är fullt med brickor. Vi kan inte att avgöra hur QL-spelarens taktik fungerar, men det märks tydligt att den insett hur viktigt det är att inte ge hörnen till motspelaren.

Det hela betyder att mänskligheten för tillfället inte behöver oroa sig över att datorerna ska ta över.

# Litteraturförteckning


[1] N. Jan van Eck och Michiel van Wezel. Reinforcement learning and its application to othello. Technical report, Erasmus University Rotterdam, Nederländerna, `http://publishing.eur.nl/ir/repub/asset/7142/ei2005-47.pdf`, 2005.

[2] Svenska Othelloförbundet. Så spelar man othello. `http://othello.nu/OthelloRegler/SaSpelarManOthello.html`, 2010-01-27.

[3] Stephen Marsland. *Machine Learning, An algorithmic perspective*. Chapman & Hall/CRC, 2009.

[4] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[5] L. Victor Allis. *Searching for Solutions in Games and Aritficial Intelligence*. PhD thesis, University of Limburg, Maastricht, Nederländerna, 1994.