KTH Computer Science
and Communication

Fyra-i-rad med förstärkt inlärning

En implementation av Q-learning

DD143X Examensarbete inom datalogi, grundnivå
CSC, Kungliga Tekniska Högskolan

Av:
Christoffer Janson
Neptunistigen 31
165 71 Hässelby
073 7536014
cjanson@kth.se

Carl Landefjord
Götgatan 78
118 30 Stockholm
073 7215275
carllan@kth.se

Handledare:
Johan Boye

June 13, 2011

Sammanfattning

Maskininlärning täcker de områden inom artificiell intelligens som handlar om en typ av inlärningsalgoritmer där program ska lära sig att utföra uppgifter utifrån de givna förutsättningarna. Förstärkt inlärning är ett område inom maskininlärningen som syftar på att programmet ska lära sig vad som är en bra handling genom att handlingarna belönas.

En algoritm som bygger på förstärkt inlärning är Q-learning. Den använder sig av en tabell med ett värde för varje tillstånd. Tabellen uppdaterar sina värden när nya belöningar blir utdelade på grund av beslut i miljön. Denna rapport beskriver hur Q-learning kan implementeras och hur implementationen beter sig mot tre olika artificiella motspelare.

Syftet med denna rapport är att undersöka hur Q-learnings inlärningsvariabel α påverkar inlärningshastigheten och för vilka värden en implementation kan tänkas fungera som bäst. Vårt bästa resultat fick vi då $\alpha = 0.9$.

Abstract

Machine learning is an umbrella term for certain types of learning algorithms in artificial intelligence. The purpose of machine learning is to make programs learn to performe tasks in accordance to certain given conditions. Reinforced learning is an area within machine learning which is all about making the program learn what is a good action by being rewarded.

One algorithm based on reinforcement learning is the Q-learning algorithm. It uses a table containing values for each state it has encountered. The table is updated with new values when new rewards are awarded by actions it has performed in the environment. This report describes how Q-learning can be implemented an how our implemention behaves against three different adversaries.

The purpose of this report is to investigate how the Q-learning learning vaiable α affects the rate of learning and for which values this implementation can performe optimaly. Our best result was $\alpha = 0.9$.

Förord

Detta är en kandidatuppsats som är skriven under vårterminen läsåret 2010/2011. Vi planerade aldrig en uppdelning av arbetet mellan oss men det föll sig naturligt så att Christoffer skrev större delen av implementationen, medan Carl har fokuserat mer på rapporten samt MATLAB. Vi skulle även vilja tacka Johan Boye för hans feedback på vårt arbete samt våra opponenter Carl Regårdh och Daniel Nyberg som har läst och satt sig in i vårt arbete.

Innehåll

Del I

Inledning

Kapitel 1

Inledning

1.1 Inledning

Innan barn kan kommunicera med sina föräldrar så lär de sig genom att interagera med omgivningen. Det är ett underliggande beteende hos oss att om någonting ger oss glädje så vill vi återuppleva det, och likaså om någonting gör oss illa eller ledsna så försöker vi undvika det. Orsak och verkan är en grundprincip som finns i naturen hos de flesta djur och som gör att vi lär oss viktiga saker, inte bara i tidig ålder, utan även genom hela livet.

Den här typen av inlärning går att rekonstruera i datorvärlden. I den här rapporten har vi undersökt hur datoriserad inlärning fungerar och hur man implementerar en artificiell intelligens som bygger på principen om förstärkt inlärning i brädspelet fyra i rad.

1.2 Problemformulering

Syftet med denna rapport är att analysera och beskriva hur vi går tillväga för att skapa en Q-learning-baserad artificiell intelligens som ska kunna agera motspelare i fyra i rad. Genom att låta vår AI spela mot olika motståndare ska den kunna träna upp sig och det vi vill veta hur snabbt den lär sig

Vi kommer studera hur effektiv Q-learning-algoritmen är när den spelar mot en annan AI som är inställd på att hindra den från att vinna, hur bra den spelar mot en motspelare som upprepar samma mönster varje match och mot en motspelare som lägger ut sina brickor helt slumpmässigt. Dessa tester kommer att utföras flera gånger med olika parametrar på en variabel i algoritmen som kallas α. Variabeln α styr inlärningshastigheten och är viktig att sätta till rätt värde för att göra Q-learning till en så effektiv lärande algoritm som möjligt.

1.3 Bakgrund

1.3.1 Maskininlärning

Maskininlärning handlar om att man inte behöver hårdkoda varje specifikt tillstånd utan man kan låta programmet resonera sig fram till bättre resultat. Det är ett stort område som använder sig av vetenskap från många håll, bland annat AI, datalogi, biologi och psykologi[4].

1.3.2 Förstärkt inlärning

Förstärkt inlärning är ett samlingsnamn för en typ av inlärningsalgoritmer, där tanken är att i stället för att med parametrar och övervakade tester lära en maskin hantera en uppgift, så får den genom trial and error orientera sig fram till ett så bra resultat som möjligt. En agent placeras i en miljö och varje gång den gör fel bedömning och förlorar så blir den bestraffad men får i stället en belöning om den når målet. Ju fler gånger agenten utför samma uppgift desto bättre blir den på att inte återupprepa tidigare misstag.

1.3.3 Förstärkt inlärning i praktiken

Förstärkt inlärning återfinns i många tv-spel. I bland annat Quake, Half-Life och Unreal serierna finns så kallade "dynamiska bottar" som agerar både motståndare och lagkamrat. Till skillnad från statiska bottar som redan har förprogrammerade rutter på olika kartor, så kan dynamiska bottar lära sig att hitta på vilken bana som helst utan att ha spelat den förut. Även svårighetsgraden ökar när man spelar mot bottarna för att de ska kunna ligga på ungefär samma nivå[5]. Förstärkt inlärning tillämpas även i logiska situationer så som pokerspel[6].

Det experimenteras även mycket med förstärkt inlärning. Ett av de mest kända experimenten är den inverterade pendeln som går ut på att en fjärrstyrd vagn ska kunna balansera en pinne på höjden. Genom att gunga fram och tillbaka så skall pinnen hållas kvar i sin position. Det som gör experimentet speciellt är att samtliga variabler är kontinuerliga istället för diskreta vilket komplicerar det hela när man skapar tillståndstabellen[1].

Att låta artificiell intelligens hantera mekanik med hjälp av ett belöningssystem är oftast ingen bra idé. Att träna ett förarlöst passagerartåg med hjälp av förstärkt inlärning är det ingen som gör, både för att utrustning kommer att gå sönder, men också för att det är väldigt farligt. Dock så finns det robotarmar som går att träna på detta sätt, där det inte är någon fara om armen kolliderar med andra objekt[6].

1.3.4 Historia

Det har forskats kring mänskligt beteende och system med konstgjord intelligens sedan 1950-talet, men i grund och botten bygger vetenskapen kring artificiell intelligens på psykologi och filosofi. Vid förra sekelskiftet började man studera hur djurs beteende förändrades när det vid upprepade tillfällen placerades i en viss miljö.

En av de första att studera trial and error var den amerikanska psykologen Edward Thorndike. Han blev känd för att ha lagt grunden till vetenskapen kring djurs beteende. Han myntade uttrycket "Law of Effect", vilket innebär att djur sannolikt, om tillfälle ges, kommer upprepa ett beteende som gav tillfredsställande resultat och undvika det som gav negativa konsekvenser.

"Law of Effect" delas i sin tur upp i två delar, valbar och associativ, där valbar innebär att man undersöker olika alternativ och väljer det alternativ som ger bäst utdelning. Den associativa delen syftar på hur man associera ett alternativ med miljön eller situationen man befinner sig i. Dessa två delar är fundamentala för förstärkt inlärning.

Den första att studera trial and error i ett datoriserad miljö var den amerikanska forskaren Marvin Minsky. I början på 50-talet byggde han det första neurala nätverket som hette SNARC (Stochastic Neural Analog Reinforcement Calculator). Under den här tiden hamnade förstärkt inlärning i skymundan för den övervakade inlärningen. Många forskare trodde att de studerade förstärkt inlärning när de i själva verket höll på med övervakad inlärning.

I början av 80-talet fick trial and error nytt liv. Bland annat Harry Klopf insåg att den förstärkta inlärningen förlorade sitt syfte när man inte använde miljön som utgångsläge för studierna. Förutom trial and error, är även dynamisk programmering för optimal kontroll och så kallad temporal inlärning, de tre viktiga beståndsdelarna för förstärkt inlärning.

1.4 Beskrivning av fyra i rad

Fyra i rad har ett spelbräde som består av 7 horisontella kolumner och 6 vertikala rader. Spelarna turas om att lägga en markör på närmast lediga plats från botten i någon av kolumnerna tills en spelare får fyra markörer i rad antingen vågrätt, lodrätt eller diagonalt eller tills dess att brädet är fullt (oavgjort).

Då det bara finns 7 kolumner för spelarna att välja så kommer den artificiella intelligensen bara behöva analysera brädet ur de 7 möjliga handlingar

som AI:n kan välja och att en spelare som bara placerar sina markörer slumpmässigt har $1 \div 7$ chans att välja den optimala kolumnen. Då Fyra i rad har 42 rutor och varje ruta har 3 olika möjliga tillstånd (spelare 1, spelare 2 eller tom) vilket betyder att antalet möjliga tillstånd som spelbrädet kan ha utan regler 3^{42} dvs $1.09 \cdot 10^{20}$. Dock så finns det regler i fyra i rad och alla tillstånd kan inte nås på grund av att en spelare har vunnit i ett tidigare tillstånd vilket minskar antalet möjliga tillstånd till $7.1 \cdot 10^{13}$ enligt Victor Allis[3].

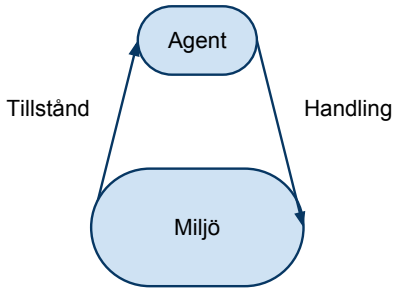
1.5 Q-learning

Q-learning är en belöningsbaserad AI algoritm. Algoritmen bygger på att agenter verkar på en miljö och att för varje handling som agenterna fattar beslut att utföra på miljön belönas eller bestraffas agenterna.

Belöningen som agenten använder för att fatta sitt beslut lagras i en så kallad Q-tabell som är indexerad på miljöns tillstånd. Efter varje handling så uppdateras Q-tabellens värde i det föregående tillståndet med hänsyn till den utförda handlingens resulterande tillstånd. Detta kan skrivas på formen:

$$Q(s, a) = Q(s, a) + \alpha(R + \gamma \cdot maxQ(s', a') - Q(s, a))$$

$Q(s, a)$ är det tillståndet som agenten befinner sig på i Q-tabellen. $Q(s', a')$ är det tillståndet som agenten kommer befinna sig i efter att den har utfört sin handling och R är en direkt belöning som kan vara en konstant, till exempel noll. α och γ är de två konstanter som påverkar inlärningshastigheten som algoritmen har. α är inlärningshastigheten som kan sättas till ett numeriskt värde som uppfyller $0 \leq \alpha \leq 1$. γ är hur mycket den framtida belöningen värderas för det nuvarande tillståndet. Denna konstant brukar vara ett eftersom programmeraren vanligtvis vill ta hänsyn till hela den framtida belöningen men kan i praktiken vara ett numeriskt tal som uppfyller $0 \leq \gamma \leq 1$. En visualisering av hur Q-learning fungerar finns i figur 1.1



Figur 1.1: Miljön ger Agenten sitt nuvarande tillstånd och Agenten returnerar en handling som ändrar miljöns tillstånd

1.5.1 Utforskande drag

Om miljön har ett begränsat antal tillstånd så kommer Q-learning-alortimens Q-tabell att konvergera mot en optimal tabell. När det finns många tillstånd så kan det inte finnas tillräckligt med minne för att lagra alla tillstånd eller så kan det finns för många tillstånd för att få en optimal tabell inom rimlig tid. Då kan det löna sig med slumpmässiga drag för att utforska andra tillstånd än de som finns i Q-tabellen eftersom det då finns en sannolikhet att algoritmen av en slump hittar ett tillstånd som är bättre än det tidigare bästa tillståndet.

Del II

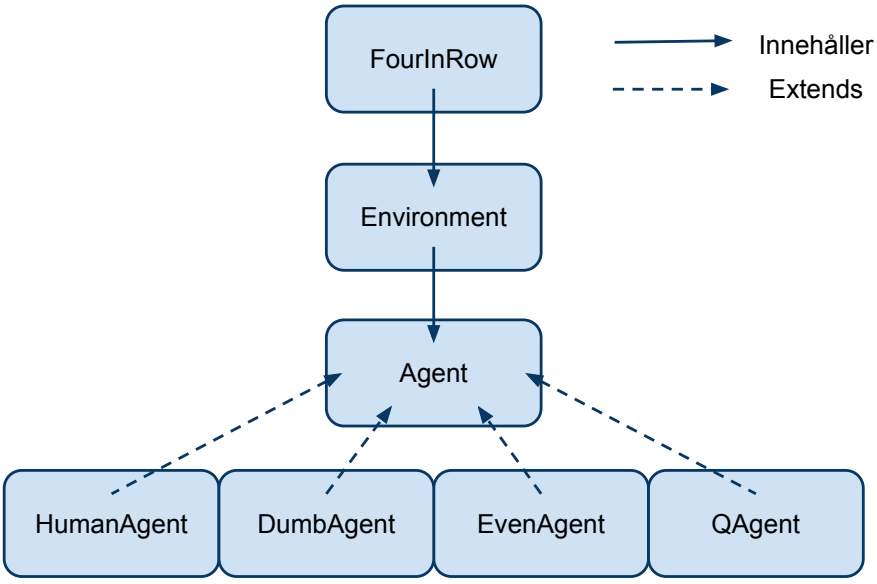
Metod

Kapitel 2

Metod

2.1 Implementationsbeskrivning

Implementationen av Q-learning är skriven i programspråket Java. Vid initieringen av implementationen så skapas en miljö som agenterna kan interagera mot. I miljön så skapas två objekt av typen Agent som kan sättas till antingen mänsklig spelare eller en AI-spelare 2.1.



Figur 2.1: Ett klassdiagram som visar hur klasserna hänger samman

2.1.1 Miljön

Miljön skapar en tom char array med 6 rader och 7 kolumner som representerar spelbrädet och skapar sedan två agenter som turas om att göra varsitt drag tills matchen är slut. Efter varje av miljön godkänt drag, kontrollerar miljön om den senast lagda markören ger den agent som placerade markören fyra i rad. Om så är fallet så meddelas agenten som vunnit respektive förlorat genom Agents metoder win() och lose() och spelet startas om.

2.1.2 Agent

Agent är ett objekt som innehåller metoder som kan ärvas av andra objekt för att ge miljön en mall att arbeta mot. Agent lagrar också information om antalet vinster, förluster samt oavgjorda matcher och en variabel som visar vilken spelmarkör som agenten har tilldelats av miljön.

2.1.3 Dum Agent

Den dumma agenten lägger vid sitt drag sin markör i kolumnen längs till vänster på spelbrädet. Om kolumnen blir full så lägger agenten i närmast lediga kolumn.

Syftet med denna agent är att studera hur snabbt QAgent kan lära sig att möta en motståndare som spelar med en upprepand strategi.

2.1.4 Slumpmässig Agent

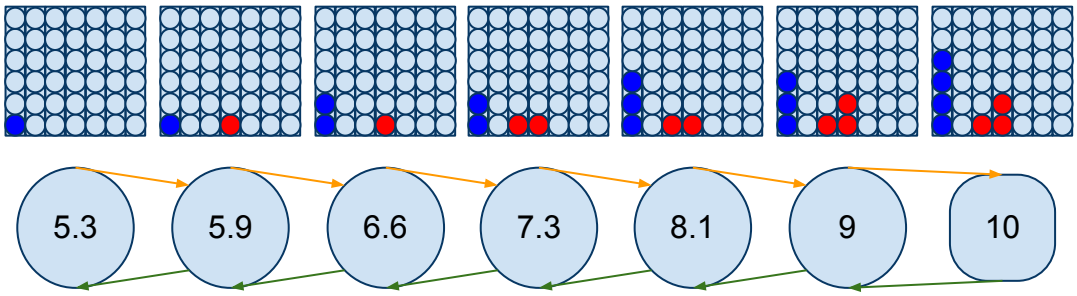
Den slumpmässiga agenten placerar vid sitt drag en markör i godtycklig ledig kolumn på spelbrädet.

Syftet med den här agenten är att studera hur snabbt QAgent kan lära sig att möta en motståndare som spelar utan någon strategi.

2.1.5 QAgent

QAgent ärver egenskaper från Agent som bland annat innehåller statistik om antalet spelade matcher och de metoder som Environment kan anropa. Denna agent vill vinna och belönar därför vinnande tillstånd med 10 och bestraffar oavgjorda och förlorade tillstånd med -10. Om agenten vinner eller förlorar på ett tillstånd som inte tidigare finns i Q-tabellen så läggs det tillståndet till och alla tidigare besökta tillståndsvärden uppdateras för att ta hänsyn till det nya tillståndet. För att se hur tillståndsuppdateringen går till se figur 2.2.

Syftet med denna agent är att studera hur olika värden på agentens α-värde påverkar agentens inlärningshasitghet.



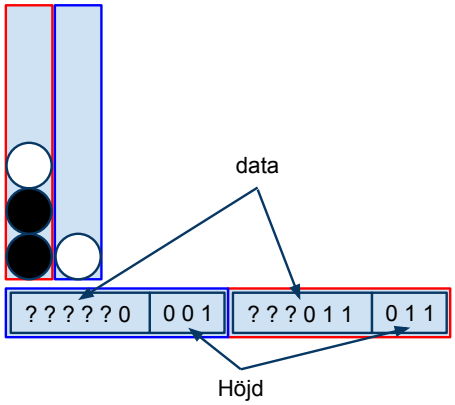
Figur 2.2: Oranga pilar visar handlingar och gröna pilar visar tillståndsuppdateringen när matchen är slut. $\alpha = 0.9$ i detta exempel

Tillståndsrepresentation

Det spelbräde som Environment-klassen lagrar består av en array med 42 char element. Eftersom en char i Java tar up två Byte så betyder det att hela arrayen kommer att ta upp $2 \cdot 42 = 84$ Byte[7]. Då fyra i rad har $7.3 \cdot 10^{13}$ möjliga tillstånd så kommer Q-tabellen att allokera $613.2 \cdot 10^{13}$ Byte om den använder samma metod som Environment använder för att lagra spelbrädet.

Detta optimerades dock genom att låta agenten formatera om miljöns bräde till ett mindre minneskrävande format. Detta gjordes genom en ny representation av spelbrädet bestående av 42 bitar. Varje bit kan antingen ha värdet 1 för att representera en AI-bricka eller 0 för att representera en motståndarbricka. Problemet som då uppstår är att en bit bara kan ha två tillstånd och en ruta på spelbrädet i fyra i rad kan ha tre tillstånd (spelare ett, spelare två och ledig ruta). För att representera de tre tillstånden reserverades 3 bitar för att ange höjden på antalet spelade brickor i varje kolumn, vilket innebär att inte mer än sammanlagt 21 bitar krävs för att representera höjden i alla kolumnerna.

Tillsammans med de 42 bitar som lagrar spelbrädet så tar spelbräde nu upp 63 bitar utan att förlora informationen kring någon av de tillstånd som kan uppstå på spelbrädet. Alltså kan hela spelbrädet lagras på en long eftersom en long i Java är 64 bitar [7]. Q-tabellen kommer alltså att ta upp $58.4 \cdot 10^{13}$ Byte vilket är 9.5% av vad den skulle ta upp om den använde samma spelbrädsrepresentation som Environment-klassen. Dock så kommer vi aldrig att uppnå en Q-tabell som innehåller alla tillstånd då tabellen med denna represenatation skulle ta 584 TB minne i anspråk. Figuren visar hur spelbrädet omvandlas från miljöns representation till den agentens representation 2.3.



Figur 2.3: Respektive färg på kolumnen omvandlat till 9 bitar i motsvarande färg. ? = spelar ingen roll

Agentens beslut

När miljön begär ett drag av en agent, har den som mest sju kolumner som den kan placera sin markör i. Beslutet om vilken kolumn som ska väljas fattas baserat på värdet som det tillståndet har i Q-tabellen. Om Q-tabellen inte innehåller ett tillstånd så värderas det tillståndet till 0. Q-learning-agenten kan fatta sitt beslut baserat på vilken kolumn som värderas högst men att enbart låta agenten arbeta på detta sätt skulle innebära en väldigt dålig tillväxt av tillstånd i Q-tabellen. Om agenten hittar en väg som den får mycket poäng för så kommer den att fortsätta utföra samma drag flera träningsmatcher i följd och det är inte förrän motståndaren har vunnit tillräckligt många gånger som agenten beslutar sig för att välja en annan kolumn.

Ett tillstånd kan ha maxvärdet 10 eller minvärdet -10 och för att få agenten att ta mer utforskande drag så sker ett slumpmässigt drag om den kolumn med det tillstånd som har störst värde och den kolumn med det tillstånd som har minst värde ligger för nära varandra, närmare bestämt inom 20% av $10 + \| - 10\|$. Detta gör att agenten kommer att utforska andra drag när det inte finns något självklart drag som är överlägset de andra och på så sätt lär den sig bättre. Agenten tar med andra ord bara hänsyn till sitt nästa drag och bryr sig inte om vad motståndaren gör för drag efter att agenten har gjort sitt drag.

Q-tabell

Den första tanken för att skapa en representation av Q-tabellen går till ett träd där varje nod har max sju löv eller noder. Denna lösning har dock stora nackdelar eftersom det tar lång tid att lägga till eller hämta ett element. Därför valdes en hashmap som lagrar alla tillstånd i tabellen. Då ett tillstånd bara tar upp 8 Byte så används en Long som nyckel till hashmappen. Då varje tillstånd behöver ett värde så använder vi en Double som värdet som är knuten till nyckeln. Detta innebär att en fullständig tabell kommer ha en ungefärlig storlek kring $(8 + 8) \cdot 7,3 \cdot 10^{13} \approx 1.1$ PB.

2.1.6 Förhalande Agent

Denna Agent innehåller samma metoder och funktionalitet som QAgent. Skillnaden mellan den här agenten och QAgent är att den här agenten inte vill vinna utan spela oavgjort mot sin motspelare. Detta uppnås genom att agenten belönar tillstånd med oavgjort resultat med 10 och bestraffar tillstånd med vinnande eller förlorade resultat med -10.

Syftet med den här agenten är att studera hur QAgent inlärningshastighet är mot en agent som har som strategi att dra ut på matchen så långt som möjligt.

2.2 Metod

När vi testkörde QAgenten mot de andra agenterna började vi först med att bestämma antalet matcher den skulle spela. Den dumma agenten gick det fortare att träna upp sig mot än mot till exempel den slumpmässiga agenten, eftersom den dumma agenten följer samma mönster varje gång. Det betyder i sin tur att de slumpade matcherna kräver fler matcher per omgång för att påvisa att inlärning sker.

Vi lät även QAgenten spela 4 olika omgångar mot varje agent och ändrade sedan α-värdet för att se hur inlärningsprocessen ändrades. I varje omgång så spelade vi ett antal träningsmatcher, sedan stängdes inlärningen av och ett antal testmatcher genomförs där genomsnittet av dessa matcher sedan blev det värde som vi använder oss av i analysen.

Del III

Resultat

Kapitel 3

Resultat

3.1 Resultat

3.1.1 QAgent vs Dum Agent

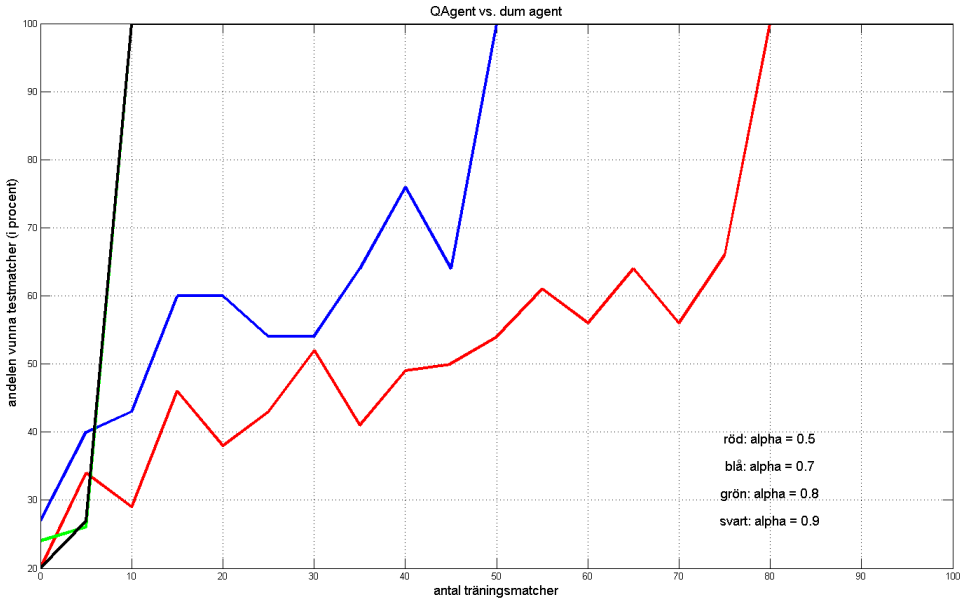
I det här testet så lät vi vår Q-learining algoritm spela 100 träningsmatcher mot en agent som lägger i den kolumn längst till vänster som är ledig. Var femte match kommer träningsmatcherna att avbrytas och 100 testmatcher kommer spelas utan att Q-tabellen uppdateras. Detta testades när α var satt till 0.9, 0.8, 0.7 och 0.5. Antalet vinster av dessa testmatcher sparades och resultatet kan ses i figur 3.1.

Vi undersökte även antalet inlärda tillstånd efter att träningsmatcherna var avslutade. Dessa visade att antalet inlärda tillstånd efter att Q-learning agent lär sig att slå den dumma agenten är noll och att agenten därefter inte lär sig någonting utan bara upprepar samma process för att slå sin motståndare och kan ses i tabell 3.1.

All data är baserad på att Q-learning agenten lägger först vilket betyder att agenten bara behöver lägga 4 gånger i samma kolumn för att vinna.

$\alpha =$	0.9	0.8	0.7	0.5
Antal tillstånd efter sista testet	57	66	439	537

Tabell 3.1: Q-tabellens antal tillstånd efter 100 träningsmatcher



Figur 3.1: En graf som visar genomsnittet av vunna testmatcher för QAgent under 100 träningmatcher mot en dum agent.

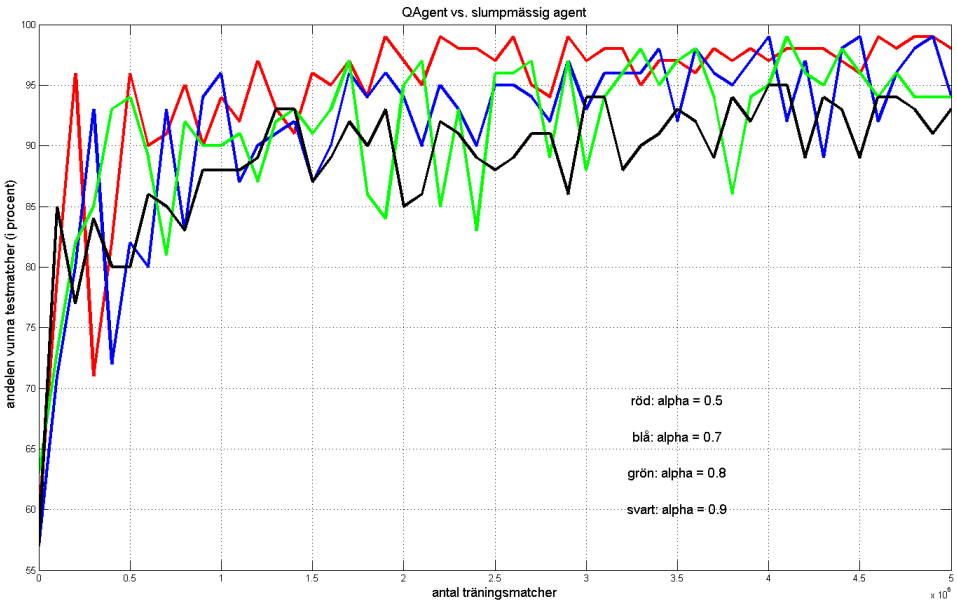
3.1.2 QAgent vs Slumpmässig Agent

I det här testet så lät vi en Q-learning agent möta en agent som spelar slumpmässigt. För att kunna visa skillnaden ordentligt mellan det olika α-värdena krävdes ett högt antal träningsmatcher. Vi valde 5 miljoner träningsmatcher så att händelseförloppet kan betraktas ordentligt. Antalet träningsmatcher mellan varje testomgång var 100000. Varje testomgång hade 100 testmatcher som användes för att logga antalet vunna matcher. Q-learning-agenten testades med α-värdena 0.9, 0.8, 0.7 och 0.5. Andelen vunna matcher kan ses i figur 3.2. Andelen vunna testmatcher ökar som du kan se logaritmiskt mot antalet spelade träningsmatcher.

Vi sparade också antalet inlärda tillstånd efter 500000 och 5000000 spelade träningsmatcher för att studera hur snabbt algoritmen lär sig. Även tiden loggades och kan ses med antalet tillstånd i tabell 3.2.

$\alpha =$	0.9	0.8	0.7	0.5
Antal tillstånd efter sista testet	15615376	12474847	13529262	6456351
Antal tillstånd efter 500000 träningsmatcher	3426525	3027836	3286428	2149914
Tid för körning (s)	827	214	333	129

Tabell 3.2: Antal tillstånd i Q-tabellen och tiden det tog att uppnå dem med olika värden för α



Figur 3.2: En graf som visar genomsnittet av vunna testmatcher för QAgent under fem miljoner träningmatcher mot en slumpmässig agent.

3.1.3 QAgent vs Förhalande Agent

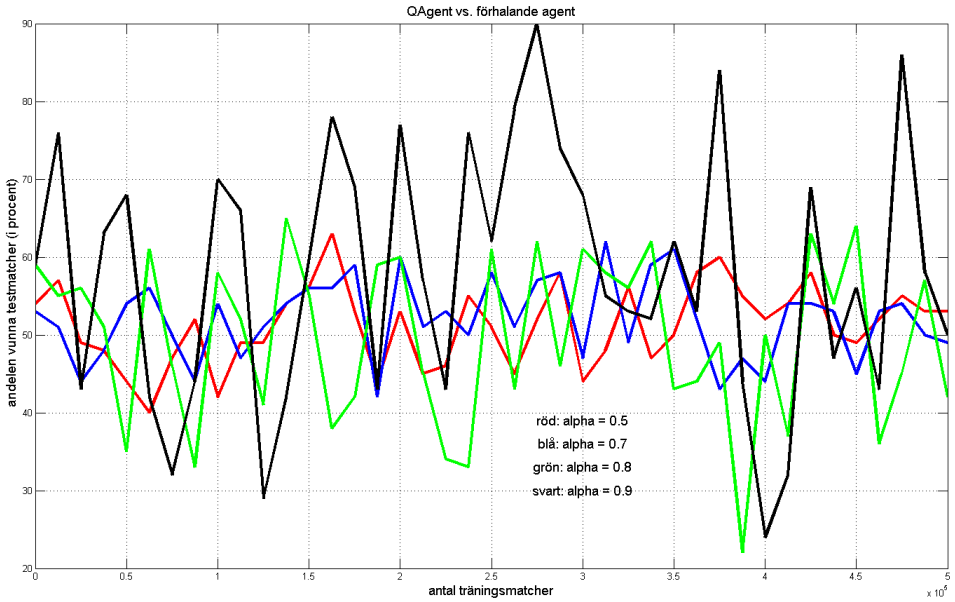
I det här testet så fick Q-learning agenten möta en motståndare som inte vill vinna eller förlora men spela lika. Vi lät QAgent spela 500000 träningsmatcher mot den förhalande agenten och för var 12500 träningsmatch så lät vi Q-learningen spela 100 testmatcher där andelen vinser sparades. Detta test utfördes med α för QAgent och EvenAgent satt till 0.9, 0.8, 0.7 och 0.5. Resultaten av dessa tester kan ses i figur 3.3.

Antalet inlärda tillstånd tillsammans med tiden det tog att spela en halv

miljon träningsmatcher visas i tabell 3.3

$\alpha =$	0.9	0.8	0.7	0.5
Antal tillstånd efter sista testet	6975921	6364281	6687699	6594651
Tid för körning (s)	95	77	102	95

Tabell 3.3: Antal tillstånd i Q-tabellen efter en halv miljon träningsmatcher och tiden det tog att uppnå dem med olika värden för α



Figur 3.3: En graf som visar genomsnittet av vunna testmatcher för QAgent under en halv miljon träningmatcher mot en förhalande agent.

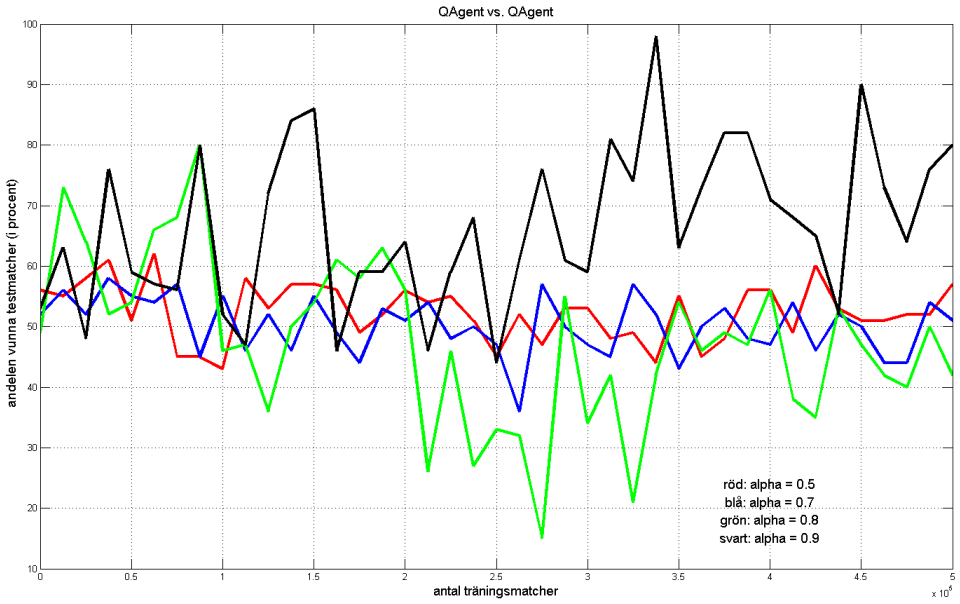
3.1.4 QAgent vs QAgent

Den sista testomgången som vi vill analysera är hur en Q-learining algoritm spelar mot sig själv. Vi valde därför att skapa två lika QAgenter med varsin Q-tabell och lät dem spela 500000 träningsmatcher mot varandra. Precis som med den förhalande agenten lät vi dom köra 12500 träningsmatcher mellan varje testomgång. Resultatet mätt i antal vinster för den QAgent som lägger först kan ses i figur 3.4.

Antalet tillstånd och tiden det tog att spela en halv miljon träningsmatcher finns i tabell 3.4.

$\alpha =$	0.9	0.8	0.7	0.5
Antal tillstånd efter sista testet	6282995	5772590	6639937	6566913
Tid för körning (s)	75	70	100	88

Tabell 3.4: Antal tillstånd i Q-tabellen efter en halv miljon träningsmatcher och tiden det tog att uppnå dem med olika värden för α



Figur 3.4: En graf som visar genomsnittet av vunna testmatcher under en halv miljon träningmatcher mellan två QAgenter. Grafen är baserad på den AI som lägger första markören varje match.

3.2 Analys

3.2.1 Minneshantering

Det största problemet med att köra testerna var inte tiden det tog att träna med Q-learing-algoritmen utan minnesåtgången som Q-tabellen tog upp. I alla våra testfall så allokerades 1 GB RAM-minne för heapen vilket var den största möjliga minnesmängd som vår testdator kunde hantera. Som man

kan se i tabell 3.2 så är skillnaden i tid mellan $\alpha = 0.9$ och $\alpha = 0.8$ är 613 sekunder vilket beror på att det allokerade minnet för heapen som Q-tabellen sparas på tog slut och att operativsystemet då börjar använda en sparmetod som kallas "växelminne" för att spara data. Växelminne används när operativsystemet har slut på RAM-minne och börjar spara undan data temporärt på hårddisken som är betydligt långsammare. Detta inträffar för oss någon gång när Q-tabellen har omkring 14 miljoner tillstånd lagrade vilket betyder ett tillstånd tar upp ca $14/1024 \approx 0,013$ MB ≈ 13 kB. Som mest kom vi upp i 15615376 tillstånd i Q-tabellen vilket motsvarar 0.2 ppm av alla möjliga tillstånd.

3.2.2 Andelen vunna matcher

Den motståndare som Q-learning-algoritmen snabbast lär sig att besegra är den dumma agenten som lägger sin markör i närmast lediga kolumn från vänster. Det som är intressant att se i tabell 3.1 på sidan 16 är att antalet tillstånd när träningsmatcherna är spelade är större när $\alpha = 0.5$ än när $\alpha = 0.9$. Detta påvisar att Q-learning-algoritmen lär sig snabbare att vinna mot motståndaragenten när α är högre. Detta antagande står sig väl om man studerar tabell 3.5 där $\alpha = 0.9$ ger de bästa resultaten mot alla motspelare utom den slumpmässiga agenten där högre α-värde ger sämre resultat. Detta påvisar att det är högre chans för vinst med ett lägre α-värde när Q-learning-algoritmen får möta en mer oförutsägbar motspelare.

$\alpha =$	0.9	0.8	0.7	0.5
Slumpmässig	0.89	0.91	0.92	0.94
Förhalande	0.58	0.50	0.52	0.51
Q-learning	0.67	0.48	0.50	0.52

Tabell 3.5: Andelen vunna testmatcher i genomsnitt

3.2.3 Antal inlärda tillstånd

Andelen inlärda tillstånd är ett annat område som är intressant att analysera för att påvisa inlärningshastigheten. I tabell 3.6 så har vi valt att sammafatta antalet tillstånd som finns i Q-tabellen efter 500000 träningsmatcher. I tabellen så kan man se att för $\alpha = 0.9$ så är antalet inlärda tillstånd störst medan $\alpha = 0.8$ generellt har de sämsta antalet inlärda tillstånd.

$\alpha =$	0.9	0.8	0.7	0.5
Slumpmässig	3426525	3027836	3286428	2149914
Förhalande	6975921	6364281	6687699	6594651
Q-learning	6282995	5772590	6639937	6566913

Tabell 3.6: Antal tillstånd i Q-tabellen efter 500000 träningsmatcher

3.2.4 Diskussion

När vi tränade upp Q-learning-algoritmen mot sig själv och mot den förhalande algoritmen så hoppar den lite i början men går mot en 50% vinst/förlust. Eftersom den förhalande algoritmen är en variant på QAgent så är det inte så konstigt att den lyckas lika bra som den vanliga Q-learning-algoritmen. När värdet på α är 0.8 eller 0.9 så blir det väldigt stora variationer i antalet vunna testmatcher mellan träningsmatcherna. Det beror antagligen på att när den vinner så blir värdeändringarna i Q-tabellen för stora och de tillstånd som besökts under matchen får för stor belöning vilket påverkar nästa täningsmatch.

Enligt grafen 3.4 så förlorar Q-algoritmen mer om den har ett α-värde på 0.8 och vinner mest om värdet är 0.9. Det kan vara slumpen som avgör att det blir så. Om man kör 100 gånger med α 0.8 respektive 0.9 så finns nog chansen att de skiftar plats där α=0.8 ligger bättre till hälften av gångerna och tvärt om.

Det behövdes bara 100 matcher för att kunna visa hur fort det går att slå den dumma agenten. Eftersom den följer samma mönster hela tiden så räcker det troligtvis att Q-learning-algortimen slår den en enda gång för att den ska vinna 100% av matcherna, om nu α är 0.8 eller högre. QAgent klassen har som tidigare nämnt en spärr inbyggd som gör att den slumpar nästa steg om inget drag är överlägset bäst. Om nu α är tillräckligt hög så kommer den slumpmekanismen aldrig att slå in och den kommer att göra om samma drag i alla nästkommande matcher och vinna. Om man studerar de omgångar där α är 0.5 och 0.7 så syns det att den lär sig hela tiden och till slut hittar den en perfekt väg.

3.3 Slutsats

Den första slutsats som kan dras av detta projekt är att en AI som använder en Q-learning-algoritm aldrig kommer att bli optimal eftersom att en fullständig Q-tabell blir för stor för en vanlig dator att hantera. Vi lyckades endast få in 0.2 ppm av den möjliga tillståndsmängden på 1024 MB så Q-

learning är knappast en effektiv metod för att lära en artificiell intelligens spela fyra i rad.

För stora värden på α så blir variationerna i antalet vunna matcher större mellan träningsmatcherna vilket kan ses i figur 3.3 och 3.4. I samma figurer har de lägre α-värdena långsammare inlärning men variationerna i antalet vunna matcher blir mindre. I figur 3.2 så syns det att de lägre α-värdena gör bättre ifrån sig än de större vilket betyder att sannolikheten för vinst blir större med lägre α på grund av att belöningarna i Q-tabellen varierar mindre.

Om man nu vill utveckla ett fyra i rad-spel med en Q-learning-algoritm som motståndare så borde den tränas upp mot sig själv med ett α-värde på $0.5 - 0.7$ eftersom det enligt grafen 3.4 verkar vara den som har minst variation mellan maximalt och minimalt antal vunna matcher mot slutet. Om man vill ha en mångsidig AI med hög vinstchans så passar $\alpha = 0.9$ bäst eftersom den lärde sig fler tillstånd än de andra α-värdena. Dock så kan man inte dra slutsatsen att högre α-värde alltid är bättre eftersom $\alpha = 0.8$ gjorde sämre ifrån sig än $\alpha = 0.5$ och $\alpha = 0.7$ i alla tester utom ett.

Litteraturförteckning

[1] Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*, Tredje Upplagan, Prentice Hall, ISBN-13: 978-0-13-207148-2

[2] Richard S. Sutton, Andrew G. Barto, *A Reinforcement Learning: An Introduction*, `http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html`, Publicerat (2005-01-04), hämtat (2011-03-03)

[3] Victor Allis, *A Knowledge-based Approach of Connect-Four*, `http://www.connectfour.net/Files/connect4.pdf`, Publicerat (1988), hämtat (2011-03-03)

[4] Tomas Olsson, *Maskininlärning och adaption*, `http://www.student.nada.kth.se/~d93-tol/ml/uppsats`, Publicerat (1997-08-07), hämtat (2011-04-10)

[5] Randar Cox, *The Bot FAQ*, `http://members.cox.net/randar/botfaq.html`, Publicerat (2006), hämtat (2011-04-10)

[6] Alex J. Champandard, *Reinforcement Learning*, `http://reinforcementlearning.ai-depot.com/Intro.html`, Publicerat (2002), hämtat (2011-04-10)

[7] *Primitive Data Types*, `http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`, Publicerat (2011), hämtat (2011-04-10)

.1 Bilagor

.1.1 Terminologi

- **Agent**: En instans av den artificiella intelligensen som arbetar i en programmiljö.
- **Artificiell intelligens (AI)**: Vetenskapen kring konstgjord intelligens.
- **Dynamisk programmering**: Algoritmer som löser optimeringsproblem genom att att dela upp problem i mindre delproblem och sedan lösa dessa för sig och arbeta sig uppåt.
- **Neurala Nätverk**: Artificiell Intelligens som bygger på algoritmer som efterliknar hjärnans närvceller.
- **Q-learning**: En inlärningsalgoritm som bygger på reinforced learning.
- **Reinforced learning**: Ett samlingsnamn för inlärningsmetoder som i grunden bygger på trial and error.
- **Temporal inlärning**: En sannolikhetsmetod som bygger på dynamisk programmering.
- **Trial and error**: En problemlösningsmetod som analyserar fel och gör förbättringar till nästa körning.
- **Heap**:reserverat minne för programvaribler.

.1.2 Kod

Källkod finns att ladda ner på `http://www.nada.kth.se/~cjanson/FourInRow.zip`