Reinforcement Leren

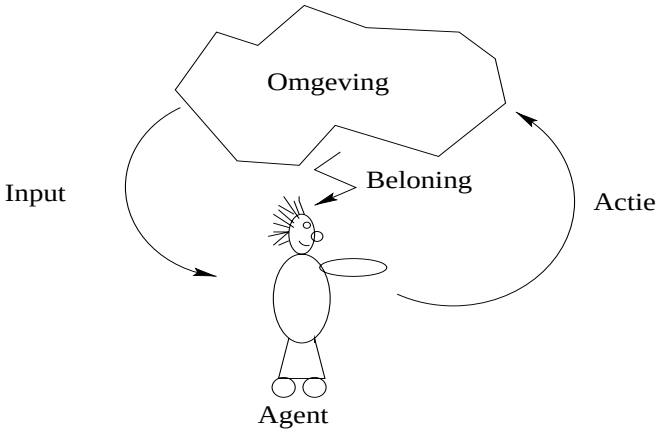
Marco A. Wiering (marco@cs.uu.nl)
Intelligent Systems Group
Institute of Computing and Computing Sciences
Universiteit Utrecht

Samenvatting

Dit korte overzichtsartikel beschrijft reinforcement leren als methode om agenten mee te leren controleren. Allereerst wordt de theoretische achtergrond van optimale controle theorie besproken. Vervolgens worden de principes van reinforcement−leer algoritmen en de belangrijkste 3 algoritmen beschreven. Tenslotte wordt kort ingegaan hoe RL algoritmen uitgebreid kunnen worden met exploratie technieken en functie approximatoren om het voor meer praktische problemen te gebruiken.

1 Introductie

Reinforcement leren (RL) stelt een agent in staat om te leren van zijn eigen ervaringen. De betekenis van reinforcement leren is "versterkings leren". Dat houdt in dat als de agent iets doet waarvoor die beloning krijgt, de agent dat gedrag daarna vaker zal uitproberen. Het doel van een agent is om zo veel mogelijk beloning over de lange termijn te vergaren. Hiervoor moet de agent verschillende gedragingen uitproberen en evalueren. Door het leren van het uitproberen van verschillende acties, kan de agent op een gegeven moment leren welk gedrag optimaal is. Hiervoor worden waarde−functies gebruikt welke weergeven hoe goed het is voor de agent om zich in een bepaalde toestand van de wereld te bevinden en hoe goed het is om dan een bepaalde actie te verrichten. Figuur 1 illustreert de interactie van een RL agent met zijn omgeving. De agent krijgt inputs van de omgeving binnen en selecteert daarmee zijn actie. Na het verrichten van zijn actie krijgt de agent een belonings signaal en maakt de agent een stap naar een nieuwe toestand in de wereld.



Figuur 1: Een RL agent die interacteert met zijn omgeving

Reinforcement leren is een groeiend onderzoeksveld. Zo is het al gebruikt om het spel Backgammon mee te leren (Tesauro, 92). In het begin werd hiervoor een neuraal netwerk random geinitialiseerd en dit netwerk werd gebruikt als evaluatie functie voor bordposities. Aanvankelijk speelde het netwerk dus willekeurige zetten. Na elke gespeelde partij werd de evaluatie functie bijgesteld door middel van een RL algoritme. Na het spelen van ruim 1 miljoen partijen tegen zichzelf, hetgeen in 1992 drie maanden kostte op een RS6000 computer, was het programma zo goed geworden dat deze zich tot de beste spelers van de wereld mocht rekenen. Reinforcement leren is ook gebruikt om robots te controlleren, om te leren schaken, om liften in een gesimuleerd gebouw te controlleren, voor verkeerslicht controle, etc.

De theorie van reinforcement leren gaat terug naar de theorie om systemen optimaal te controlleren in een bekende omgeving. Deze theorie van "optimale controle" hield zich bezig met dynamisch programmeren om gegeven een probleem specificatie een optimale oplossing te berekenen. Het interessante van reinforcement leren is dat de omgeving initieel geheel onbekend kan zijn, en de agent zelf de effecten van zijn acties moet leren en ook moet leren wat het doel is in deze omgeving. Een RL agent zal dus in het begin willekeurige acties uitvoeren om informatie mee te vergaren. Als de agent een keer beloning krijgt, kan deze natuurlijk het gedrag herhalen dat tot de beloning leidde, maar dat heeft weinig zin om bepaalde redenen: (1) De omgeving is meestal stochastisch dus een herhaling van gekozen acties hoeft niet altijd tot het bereiken van het doel te leiden; (2) De agent wil een optimale controller te leren en initiele "trials" die tot het doel leiden zijn meestal verreweg van optimaal. Dus moet de agent exploreren om ook andere acties uit te proberen.

De theorie van reinforcement leren vertelt ons dat een agent uiteindelijk optimaal zal worden als aan een aantal condities voldaan is. Uiteindelijk betekent hier dat de agent alle acties in alle toestanden oneindig vaak moet hebben uitgeprobeerd. Hoewel dat voor praktische toepassingen dus onmogelijk is, kan er vaak na een eindig aantal "trials" al gestopt worden, en is de controller meestal al vrij goed. Langer uitproberen kan de agent wel verbeteren, maar een optimale oplossing kan nooit gegarandeerd worden in eindige tijd. Dit heeft te maken met de stochasticiteit in de omgeving. De agent zal namelijk nooit de kansen precies kennen van de echte transitie functie: wat is de kans dat de agent zich na het uitvoeren van een bepaalde actie in een bepaalde toestand in een nieuwe toestand bevindt? Als deze kans bijvoorbeeld 0.543 is, en de agent probeert deze actie 1000 keer uit, dan kan de voorspelde kans bijvoorbeeld 0.527 zijn, maar precies weet de agent het nooit. In sommige gevallen is het noodzakelijk om alle overgangskansen precies te kennen om het optimale gedrag te leren en dit kost nu eenmaal oneindig veel tijd. Toch laten allerlei onderzoeken zien dat het goed mogelijk is om een agent na eindige tijd te stoppen. In de meeste gevallen is dan een goede of bijna optimale oplossing gevonden.

In dit korte overzichtsartikel zullen we allereerst de modellering van het probleem aan de hand van Markov decision problems (MDPs) bespreken en ingaan hoe we met dynamisch programmeer algoritmen hiervoor optimale oplossingen kunnen berekenen. Vervolgens komt in Hoofdstuk 3 reinforcement leren aan de orde. Hier worden de principes van reinforcement−leer (RL) algoritmen besproken en worden drie verschillende algoritmen beschreven. In Hoofdstuk 4 wordt kort beschreven hoe de agent kan exploreren om de hele toestandsruimte te doorlopen en hoe de agent m.b.v. functie approximatoren waardefuncties voor zeer grote of continue toestandsruimtes kan leren. Hoofdstuk 5 concludeert dit overzicht.

2 Dynamisch Programmeren

Dynamisch programmeren wordt gebruikt voor verschillende toepassingen zoals in vision, bio−informatica, en optimale controle theorie. De toepassing is meestal om door herhaalde updates van een bepaalde functie over een toestandstuimte, een resulterende (optimale) functie te vinden. We zullen hier allereerst ingaan op de modellering van het controle probleem waarin wij geinteresseerd zijn.

2.1 Markov Decision Problems

Markov decision problems (MDPs) zijn problemen welke bestaan uit een aantal discrete toestanden, een aantal acties welke in de verschillende toestanden verricht kunnen worden, een transitiefunctie welke aangeeft naar welke toestand de wereld zal gaan als de agent een bepaalde actie in een toestand zal uitvoeren, en een beloningsfunctie welke aangeeft hoeveel beloning de agent krijgt als deze een actie uitvoert in een bepaalde toestand. Meer formeel bestaat een MDP uit:

1) Een verzameling mogelijke toestanden van de wereld: $S = \{s^1, s^2, ..., s^N\}$
2) Een verzameling acties welke uitgevoerd kunnen worden: $A = \{a^1, a^2, ... a^M\}$
3) Een transitiefunctie welke de kans aangeeft om naar een nieuwe toestand s(t+1) te gaan na het uitvoeren van actie a(t) in toestand s(t). Deze kans wordt genoteerd als: P(s(t), a(t), s(t+1)). Deze transitiefunctie moet voor alle acties in alle toestanden gespecificeerd worden en is gewoonlijk stationair (onafhankelijk van de tijd).
4) Een beloningsfunctie welke aangeeft hoe veel beloning r(t) de agent krijgt voor het uitvoeren van actie a(t) in toestand s(t) waarna de agent een stap maakt naar toestand s(t+1). Deze beloningsfunctie wordt genoteerd als R(s(t), a(t), s(t+1)).
5) Een discountfactor γ welke onmiddelijke beloningen afweegt tegen beloningen die later worden ontvangen. Dit zorgt er b.v. voor dat een agent liever zo snel mogelijk een schaakwedstrijd wint in plaats van te winnen na een groot aantal zetten.
6) Een tijdsindicator t = {1, 2, ..., T} waarin T ook oneindig kan zijn.

De bedoeling is nu dat de agent een reeks acties executeert welke zijn behaalde gedisconteerde som van beloningen maximaliseert. De agent gebruikt zijn policy−functie $\pi(.)$ om toestanden op acties te mappen:

$$a(t) = \pi(s(t))$$

In het begin is de optimale policy onbekend. Het doel is nu om de policy te vinden die de volgende functie maximaliseert:

$$\pi^*(.) = \arg\max_{\pi(.)} E(\Sigma_t \gamma^t R(s(t), \pi(s(t)), s(t+1)))$$

Hier wordt dus gekeken naar alle mogelijke policy−functies en wordt bekeken hoeveel "discounted" beloning de agent gemiddeld ontvangt door het volgen van een bepaalde policy. De E−operator is hier een verwachtings operator welke het gemiddelde van alle mogelijke trajecten in de toestandsruimte bepaalt. De hoeveelheid beloning is de (gedisconteerde) som van alle beloningen verkegen tijdens alle toekomstige stapjes (acties). Het moge duidelijk zijn dat we deze optimale policy

niet zo maar kunnen berekenen. We zouden de verwachte beloningssom van alle mogelijke policies wel kunnen uitrekenen, maar er zijn een exponentieel aantal policies. Het aantal policies is namelijk M^N waarbij M het aantal mogelijke acties van de agent is, en N het aantal mogelijke toestanden.

2.2 Dynamisch Programmeren

We kunnen de optimale policy berekenen met behulp van dynamisch programmeer algoritmen. Allereerst voeren we twee waardefuncties in: een toestand waardefunctie V(s) welke aangeeft hoeveel de som van de toekomstige beloningen wordt als de agent zich in toestand s bevindt en policy π gebruikt wordt om acties te selecteren. Ten tweede gebruiken we een actie−selectie waardefunctie Q(s,a) welke aangeeft hoe groot de verwachte som van de toekomstige beloningen wordt als de agent in toestand s zit en actie a selecteert, waarna de agent met policy π verder gaat. Deze functies dienen ervoor om verschillende policies met elkaar te vergelijken en kunnen recursief berekend worden. We willen allereerst dus dat V(s) de volgende waarde aanneemt:

$$V(s) = E(\Sigma_t \gamma^t R(s(t), \pi(s(t)), s(t+1)) \mid s(0) = s)$$

Of te wel de gedisconteerde som van toekomstige beloningen. Nu kunnen we gebruik maken van een lokale vergelijking om Q(s,a) te bepalen. We constateren dat de waarde van een actie in een toestand gelijk moet zijn aan de onmiddelijk ontvangen beloning plus de som van de beloningen die vanuit de nieuwe toestand wordt verkregen. Verder is er een kansverdeling om naar opvolgende toestanden te gaan. Met deze kennis kunnen we Q(s,a) dus als volgt bepalen:

$$Q(s,a) = \Sigma_{s'} P(s,a,s') (R(s,a,s') + \gamma V(s'))$$

Hiermee kunnen we dus Q bepalen als we V kennen. V kennen we nog niet, maar we kunnen V recursief uitdrukken in een vergelijking over de Q−waarden van de verschillende acties. De waarde van een toestand is namelijk gelijk aan de Q−waarde van de beste actie in die toestand. Dit houdt dus in dat we V als volgt kunnen bepalen:

$$V(s) = Max_a \ Q(s,a)$$

Tenslotte kunnen we de policy bepalen als we de Q−waarden van de verschillende acties weten. De beste actie in een toestand is namelijk de actie met de hoogste Q−waarde in die toestand:

$$\pi(s) = Arg\ Max_a\ Q(s,a)$$

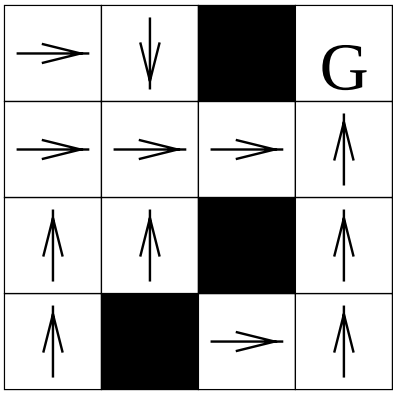
Nu hebben we dus drie formules om V, Q, en π te bepalen. Ze zijn echter allemaal van elkaar afhankelijk: als we de waarde van een toestand niet weten, kunnen we ook de waarde van een actie in een toestand, die een kans heeft om naar de nog niet geevalueerde toestand te gaan, nog niet bepalen. Dit lossen we op door recursief te itereren. De Bellman vergelijking (Bellman, 57) vertelt ons dat de optimale Q* en V* waarden aan de volgende vergelijking moeten voldoen:

$$Q^*(s,a) = \Sigma_{s'} P(s,a,s') (R(s,a,s') + \gamma V^*(s')),$$

waarbij V* opnieuw uitgedrukt kan worden in termen van Q:

$$V^*(s) = Max_a Q^*(s,a)$$

Dit betekent dus dat er een fixed point is in de waardefunctie ruimte. Als we eenmaal de optimale Q* en V* functie hebben gevonden, maakt dooritereren niets meer uit: alles blijft dan gelijk. Ter illustratie Figuren 2a en 2b: hier worden de optimale policy en waardefunctie afgebeeld voor een klein doolhof probleem waarbij elke stap 1 punt kost en het bereiken van het doel met 10 punten beloond wordt. Merk op dat er maar 1 optimale V−functie bestaat, maar meerdere optimale policies (sommige toestanden hebben meerdere mogelijke optimale acties).



Figuur 2a: De optimale policy in een doolhof

5	6		10
6	7	8	9
5	6		8
4		6	7

Figuur 2b: De optimale waarde functie in deze doolhof

Hoe kunnen we deze optimale policy en waardefuncties nu berekenen? We beginnen met een initiele Q− en V−functie en itereren totdat er niets meer verandert. Dit wordt bijvoorbeeld gedaan door het Value Iteratie algoritme:

1) Initialiseer V(s) en Q(s,a) voor alle toestanden s en acties a. Meestal wordt hier V(s) = 0 en Q(s,a) = 0 voor alle toestanden en acties.
2) Herhaal stappen 3−5 totdat de waarde−functie niet of nauwelijks meer verandert.
3) Update de Q−functie voor alle toestanden en acties m.b.v. de volgende vergelijking:
   $Q(s,a) = \Sigma_{s'} P(s,a,s') (R(s,a,s') + \gamma V(s'))$
4) Pas de V−functie aan:
   $V(s) = Max_a Q(s,a)$
5) Pas de policy aan:
   $\pi(s) = Arg\ Max_a Q(s,a)$

Dit algoritme geeft de optimale Q*, V*, en policy terug, hoewel je hiervoor oneindig vaak moet dooritereren. Meestal wordt met de iteratie gestopt als de waarde−functie V nauwelijks meer verandert; dus als $|V^{t+1}(s) - V^t(s)| < \varepsilon$, voor alle toestanden s, waarbij ε een kleine waarde heeft. Als we dat doen, verkrijgen we een sub−optimale oplossing welke beter is naarmate ε kleiner is.

2.3 Dynamisch Programmeren als Tool voor Planning

Als het model met transitiekansen en beloningen a−priori gegeven is, kan met behulp van dynamisch programmeer algoritmen de optimale oplossing berekend worden. Voor Pad−plannings problemen is dit vaak veel efficienter dan het gebruik van conventionele padplanners zoals Breadth of Depth first search. In principe lijkt het algoritme wel wat op A* of Dijkstra's korste pad algoritme, maar deze standaard algoritmen kunnen niet met kansverdeling omgaan of met positieve beloningen (negatieve kosten), hetgeen DP algoritmen algemener bruikbaar maakt. Een voordeel van DP is dat alle informatie om een actie te selecteren in de policy zit. Dus als de Q− en V−functies nauwkeurig zijn hoeft er tijdens de executie van een agent niets meer geplanned te worden; de agent kan reactief in een toestand een actie selecteren en executeren. Er is dus een trade−off tussen de nauwkeurigheid van de waardefuncties en een mogelijke planning of lookahead strategie waarin gekeken wordt welke reeksen van acties ondernomen kunnen worden. Een goed voorbeeld om dit onderscheid duidelijk te maken is een spelprogramma zoals een schaakprogramma. Een schaakprogramma bestaat uit een evaluatie functie en een methode om zetten vooruit te rekenen. Als de evaluatie functie perfect is, is er geen noodzaak om dieper dan 1 zet te rekenen. Voor schaakprogramma's is het echter vrijwel onmogelijk om een perfecte evaluatie functie te hebben vanwege het enorme aantal toestanden, lookahead blijft hiervoor dus noodzakelijk. Voor bepaalde problemen als pad− planning of navigatie in stationaire werelden is lookahead echter niet noodzakelijk en kunnen we dus een snelle reactieve agent gebruiken. Helaas is DP alleen mogelijk als het model a−priori gegeven is. Als dit niet het geval is, moeten reinforcement−leer algoritmen toegepast worden. Dit wordt besproken in het volgende hoofdstuk.

3 Reinforcement Leren

Reinforcement leren (Kaelbling et al, 96; Sutton en Barto, 98) stelt een agent in staat te leren van zijn interactie met een omgeving. Initieel wordt de agent in een startpositie gezet en weet de agent niets over de overgangskansen en beloningen. Merk dus op dat de agent geen initieel doel kent; het enige dat de agent wil is zijn som van toekomstige beloningen maximaliseren, maar hoe hij dit moet doen moet hij zelf leren. Nadat een trial is begonnen, verkrijgt de agent na elke actie informatie om van te leren (we gaan allereerst weer uit van een discrete toestand/actie ruimte):

- Agent zit in toestand s(t)
- Agent selecteert actie $a(t) = \pi(s(t))$
- Agent executeert actie en gaat naar toestand s(t+1) en vergaart beloning r(t)
- Agent maakt een aanpassing van zijn waarde−functies m.b.v. de informatie (s(t), a(t), s(t+1), r(t))

Als nu een RL algoritme gebruikt wordt, leert de agent steeds beter om acties te selecteren welke hem een hoge lange termijn som aan beloningen geven. Voor de aanpassing van de waarde functies bestaan er 3 conventionele RL algoritmen: (1) Q− leren, (2) Monte Carlo Sampling en (3) Model−gebaseerde RL. We zullen ze alle 3 bespreken.

3.1 Q−leren

Q−leren (Watkins, 89) past 1 Q−waarde aan na elke stap. Nadat een stap gemaakt is en de informatie (s(t), a(t), s(t+1), r(t)) bekend is, maakt Q−leren de volgende leerstap:

$$Q(s(t), a(t)) = (1 - \alpha)\, Q(s(t), a(t)) + \alpha\, (r(t) + \gamma\, V(s(t+1)))$$

Hierin is α de leersnelheid welke een waarde tussen 0.0 en 1.0 heeft. In principe verschuift de Q−leerregel de Q−functie na elk stapje een beetje om met de laatste ervaring (stap) rekening te houden. Merk op hoe dicht de Q−leerregel bij DP algortimen staat: de kansverdeling is vervangen door een leersnelhied. Als α langzaamaan afneemt zodat aan bepaalde eisen voldaan is, convergeert de Q−functie met Q−leren naar de optimale Q*−functie als alle toestanden/actie paren oneindig vaak uitgeprobeerd worden. Q−leren is onder andere gebruikt om liften mee te controleren in een gesimuleerd gebouw. Hiervoor werd Q−leren gecombineerd met neurale netwerken (Crites and Barto, 96). Het resulterende algoritme was in staat om 4 liften beter te controlleren dan conventionele controllers en kan daarmee de totale wachttijd met 15% reduceren.

Q−leren leert steeds maar 1−stapje terug; de updates die van een doellokatie naar een beginlokatie gaan worden dus maar langzaam verricht. Stel bijvoorbeeld dat een agent moet leren dat er honderd passen rechts een doellokatie is. De eerste trial beweegt de agent zich random (hij heeft immers nog niets geleerd). Als de doellokatie nu gevonden wordt, en de agent daarvoor beloning krijgt, wordt enkel de laatste stap geupdated door de uiteindelijke positieve beloning. Dit zorgt soms dus voor een lang leerproces. Manieren om dit leren te versnellen zijn om gebruik te maken van Q(λ) leren (Peng and Williams, 96). We zullen daar nu niet verder op ingaan.

3.2 Monte Carlo Sampling

Monte Carlo sampling wordt gebruikt voor verschillende sampling methoden. In principe berust deze methode op het genereren van uitkomsten en het middelen van deze uitkomsten om een schatting te krijgen van de echte waarde (of kans op iets). Monte Carlo sampling werkt als volgt; de agent maakt een trial en stelt het leren uit (het is dus een off−line lerende agent). Tijdens de trial houdt de agent precies bij welke toestanden deze heeft gezien, welke actie hij daarin heeft verricht en hoeveel beloning hij heeft gekregen. Nadat de trial van de agent afgelopen is, wordt allereerst een cumulatieve reinforcement signaal berekend voor elk tijdstip tot het einde van de trial:

$$R(t) = \Sigma_{i=t}\ \gamma^{i-t}\ r(i)$$

Vervolgens worden alle voorgekomen (s(t), a(t)) paren geupdated m.b.v.:

$$Q(s(t), a(t)) = (1 - \alpha)\, Q(s(t), a(t)) + \alpha\, R(t)$$

Er wordt hierbij dus een verschuiving gemaakt van de Q−functie naar de bepaalde

som van beloningen in de trial. Er valt nog onderscheid te maken tussen first−visit en every−visit Monte Carlo methoden. De first−visit methode maakt 1 update (alleen voor de allereerst voorkomende (s, a) paar en de every−visit methode maakt een update voor alle keren dat (s, a) is voorgekomen in een trial. Als we een waar gemiddelde als uitkomst willen hebben (we berekenen dan het gemiddelde van alle trials waarin (s, a) voorkwam) dan kunnen we dan doen door de leersnelheid als volgt te laten afnemen:

$$\alpha = 1 / N(s,a)$$

Waarin N(s,a) het aantal keren is dat paar (s,a) is geupdated. Monte Carlo sampling heeft als voordeel dat de hele toekomst wordt gebruikt om te updaten. Het lijkt daarom wellicht dat Monte Carlo sampling sneller naar de optimale Q−functie zal convergeren dan Q−leren. Dit is echter vaak niet het geval, de variantie van de updates wordt namelijk veel groter. Hoewel Monte Carlo sampling geen bias heeft (de huidige Q−functie wordt helemaal niet gebruikt) is de variantie erg hoog. De variantie is erg hoog omdat de hele stochastische toekomst gebruikt wordt in de update, en deze toekomst kan veel verschillende uitkomsten genereren. Aangezien de fout van de Q−functie opgesplitst moet worden in de bias en variantie, kan het zo zijn dat Monte Carlo sampling een hoge fout in zijn huidige schatting heeft door de hoge variantie in de updates.

Een andere probleem is dat exploratie acties een grote verstorende rol kunnen hebben in Monte Carlo sampling. Exploratie is nodig (zie ook het volgende hoofdstuk) om de optimale Q−functie te kunnen leren, maar veel exploratie met Monte Carlo sampling zorgt voor een door de exploratie gebiasde feedback. Aangezien door exploratie soms slechte of sub−optimale acties geselecteerd worden, kan de verkregen som van beloningen soms veel lager uitvallen dan wat de huidige beste policy zou hebben verkregen. Q−leren heeft hier geen last van en wordt ook wel off−policy leren genoemd: bij Q−leren wordt altijd enkel de Q−functie aangepast op de direct gekozen actie en hebben exploratie acties geen verstorende invloed.

3.3 Model−gebaseerd Reinforcement Leren

Het derde en laatste reinforcement−leer algoritme is gebaseerd op het gebruik van een model. Dit model schat de transitie kansen en beloningen op de transities en gebruikt vervolgens dynamisch programmeer−achtige algoritmen om de waarde−functies te updaten. Dit zorgt vaak voor een leeralgortime welke veel minder ervaringen nodig heeft om een goede of de optimale Q−functie te leren. Het algoritme vereist echter wel dat de transitiefunctie wordt opgeslagen, dus is de ruimte−complexiteit van het leeralgoritme hoger dan bij Q−leren of Monte Carlo sampling.

In model gebaseerd reinforcement leren gebruiken we een aantal tellers om de transitie en beloningsfunctie te schatten uit de verkregen ervaringen van de agent. Hiervoor introduceren we de volgende tellers:

C(s, a) = aantal keren dat actie a in toestand s is verricht.
C(s, a, s') = aantal keren dat actie a in toestand s is verricht en de agent een stap heeft gemaakt naar toestand s'.
RT(s, a, s') = som van onmiddelijke beloningen nadat actie a in toestand s is verricht en er een stapje werd gemaakt naar toestand s'.

Uit deze tellers (welke real numbers kunnen zijn), kunnen we de transitie− en beloningsfunctie als volgt schatten:

$$P'(s, a, s') = C(s, a, s') / C(s, a)$$

en

$$R'(s, a, s') = RT(s, a, s') / C(s, a, s')$$

Na het aanpassen van de transitie− en beloningsfunctie na elk stapje kunnen we dynamisch programmeer algoritmen (value iteratie) gebruiken om direct een nieuwe waarde functie te berekenen. Dit is echter erg traag, omdat de hele Q−functie opnieuw berekend gaat worden op basis van 1 verandering van het model. Om hier versnelling in aan te brengen zijn er 2 algoritmen bedacht: real−time dynamisch programmeren en Prioritized Sweeping (Moore and Atkeson, 93). Real−time dynamisch programmeren past de Q−functie in een beperkt aantal iteraties (b.v. 1 of 5) aan zodat elke update niet te lang gaat duren. Prioritized Sweeping (PS) bekijkt welke Q−waarden aangetast worden door de laatste update en past enkel deze waarden aan. In principe werkt PS net als een nieuwsflits, als er ergens in een gebied een waarde geupdated wordt, worden enkel naburige toestanden opnieuw geupdated. Dit zorgt voor een heel snel algoritme om de noodzakelijke updates door te voeren voor de grootste aangetaste Q−waarden en andere Q−waarden niet te updaten. Experimenten hebben aangetoond dat PS veel sneller dan Q−leren of Monte Carlo sampling een goede schatting van de optimale Q−functie kan leren. Model−gebaseerde RL kan echter alleen gebruikt worden als de toestandsruimte goed gediscretiseerd kan worden en niet te groot is.

4 RL in de Praktijk

We hebben nu gezien hoe drie verschillende RL−algoritmen eruit zien. In de praktijk combineren we deze technieken met enkele andere methoden om een goed werkend syteem te verkijgen. De eerste methode is het gebruik van exploratie om de optimale Q−functie te leren, de tweede methode maakt gebruik van functie approximatoren voor het omgaan met grote of continue toestand/actie ruimtes.

4.1 Exploratie

Als we steeds acties selecteren met behulp van de huidige greedy policy $\pi(.)$ verkrijgen we meestal een lokaal optimum terug. Het kan namelijk zo zijn dat de Q−waarden van de huidige policy hoger zijn dan de Q−waarden van alternatieve acties waardoor deze nooit uitgeprobeerd worden. Het doel van exploratie is dus om van tijd tot tijd alternatieve acties te selecteren om zo op zoek te gaan naar de optimale policy.

De meest eenvoudige exploratie methode is de Max−random methode. Hierin wordt de huidige beste actie gekozen met kans $(1 - \varepsilon)$ en een alternatieve random actie met kans ε. Als de exploratie−rate ε langzaam naar 0 gaat, worden er steeds meer acties verricht volgens de greedy policy, terwijl er aan het begin veel geexploreerd wordt.

De Max−random exploratie methode wordt het vaakst gebruikt. Men kan ook gebruik maken van de Boltzmann exploratie methode welke kansen om een actie te selecteren gegeven de Q−functie als volgt bepaald:

$$P(a(t) \mid s(t)) = \exp(Q(s(t), a(t)) / \tau) / \Sigma_a \exp(Q(s(t), a) / \tau)$$

Hierin is τ de (afnemende) temperatuur (als deze 0 is wordt de greedy actie geselecteerd, als deze oneindig is worden random acties geselecteerd).

De twee bovengenoemde exploratie methoden zijn indirecte exploratie methoden; ze houden geen rekening met wat de agent al bezocht heeft of in welke toestanden de agent al veel informatie verzameld heeft. Directe exploratie methoden maken hier wel gebruik van. Zo kan de agent bijvoorbeeld steeds de actie selecteren welke het minst vaak geselecteerd is en als alle acties in alle toestanden een N aantal keer geselecteerd zijn, kan de agent meer greedy gedrag gaan vertonen. Voor model−gebaseerd leren bestaan er enkele efficiente directe exploratie algoritmen om de hele toestandstuimte te doorlopen (Wiering and Schmidhuber, 1998). Deze algoritmen kunnen het leren van een (bijna) optimale policy aanzienlijk versnellen. Onderzoek naar exploratie heeft ook duidelijk gemaakt dat optimistische waardefuncties welke de waarde van toestanden optimistisch bekijken (volgens een statistische methode) zeer goede exploratie methoden kunnen opleveren.

4.2 Functie Approximatie

Als de toestandsruimte heel groot (b.v. door meerdere input dimensies) of continu is, dan is het vaak ondoenlijk om alle toestand/actie paren in een tabulaire representatie op te slaan. Daarom worden functie approximatoren in zulke gevallen gebruikt. Dit biedt de volgende voordelen: als een toestand nog nooit bezocht is, kan de functie approximator toch een Q−waarde voor die toestand hebben geleerd door gebruik te maken van generalisatie; daarom hoeft niet de hele toestandsruimte doorzocht te worden. Het leren kan dus veel sneller gaan met functie approximatoren, maar een nadeel is dat de functie approximator vaak problemen heeft om de exacte optimale waardefunctie op te slaan. Verder vergeet de functie approximator bepaalde toestandswaarden en kan de functie approximator soms overgeneraliseren zodat de preciese waarde van een toestand/actie paar niet goed geleerd wordt. Er zijn verschillende mogelijke functie approximatoren; (1) neurale netwerken worden veel gebruikt voor spelprogramma’s en ook wel voor robots; (2) CMACs worden gebruikt om eerst de toestandsruimte te discretiseren en op te splitsen in een verzameling discrete cellen, waarna de cellen de Q−waarde van een toestand bepalen. CMACs zijn onder andere gebruikt om gesimuleerd multi−agent voetbal mee te leren. (3) Verder worden self organizing netwerken, beslisbomen, locally weighted regression etc. gebruikt in combinatie met RL.

Het gebruik van de combinatie van functie approximatoren en RL biedt geen grote problemen indien Q−leren of Monte Carlo sampling gebruikt wordt. In dit geval kan de functie approximator direct gebruikt worden om de updates mee te maken. In het geval van model−gebaseerd RL kunnen neurale netwerken niet goed gebruikt worden, omdat het leren van de transitie− en beloningsfunctie met een neuraal netwerk grote problemen oplevert; namelijk hoeveel outputs moet het netwerk hebben als de

omgeving stochastisch is? Zelfs als dit een vaststaand gegeven aantal is, is de combinatie neurale−netwerken model−gebaseerde RL vaak erg traag en daarom vanwege efficientie redenen niet altijd goed toepasbaar.

In (Kaelbling et al., 96) staat een overzicht van onderzoek naar het gebruik van functie approximatoren in combinatie met RL.

4.3 Toepassingen

RL heeft veel verschillende toepassingen. Over het algemeen wordt het gebruikt voor controle of predictie, maar er zijn ook toepassingen van RL voor combinatorische optimalisatie problemen. Onder het laatste vallen onder andere de Ant Colony Systemen (Dorigo, 97). RL is al efficient toegepast om spelletjes te leren (schaken, dammen, backgammon) en biedt wellicht de meestbelovende methode om een spelletje te leren door een programma tegen zichzelf te laten spelen. Ook is RL gebruikt voor network routing (Littman and Boyan, 93), waarin een aantal pakketjes over een verbonden netwerk getransporteerd moeten worden (vergelijk internet routing systemen). De RL systemen kunnen vaak goed omgaan met de beschikbare resources of met veranderingen in de omgeving. RL is ook toegepast op lift controle (Crites and Barto, 96) en verkeerslicht controle (Wiering, 00). Voor verkeerslicht controle werd met RL geleerd wat de wachttijd van auto's is als een bepaald verkeerslicht op rood en op groen staat. Elk auto heeft een bepaald voordeel als zijn licht nu op groen wordt gezet en dit voordeel is gelijk aan de wachttijd voor een rood licht min de wachttijd op een groen licht. Vervolgens werden de verkeerslichten op een kruising gezet om de winst ta maximaliseren. Er is nu een verkeerslicht simulator (http://www.sourceforge.net/project/stoplicht) waarmee men met verschillende lerende en statische verkeerslicht controllers kan experimenteren. De voorlopige resultaten tonen aan dat RL de doorstroming van het verkeer aanzienlijk kan verbeteren ten opzichte van vaste controllers. RL wordt ook gebruikt in combinatie met speltheorie om rationele agenten te leren welke bepaalde matrix spelen tegen elkaar spelen en van de uitkomst kunnen leren. Zo onderzochten Sandholm and Crites (1995) of RL agenten de tit−for−tat strategie in de Prisoner's dilemma konden leren en dit bleek wel het geval te zijn. RL wordt ook gebruikt om robots mee te controleren, maar omdat RL veel trials nodig kan hebben, wordt vaak a−priori hand gegeven data gebruikt (joystick of programma) zodat het startpunt al vrij redelijk is en de agent niet heel veel moeite heeft om af en toe beloningen te vergaren.

5 Conclusie

We hebben in dit korte overzichtsartikel beschreven hoe reinforcement leren werkt. Allereerst hebben we de theoretische achtergrond van optimale controle beschreven en dynamisch programmeren besproken. Dynamisch programmeer algoritmen gebruiken waarde functies voor toestanden en toestand/actie paren en kunnen hiermee een optimale policy berekenen als het model bekend is. Als het model onbekend is, dan kunnen we reinforcement leren (RL) gebruiken om door middel van interactie met de omgeving een policy te leren. Als RL met een tabulaire representatie gebruikt wordt, leert een agent onder bepaalde condities de optimale policy als alle toestanden oneindig vaak bezocht worden. Hoewel dit geen sterk theoretisch uitgangspunt is

(oneindig lang exploreren is in realiteit onhaalbaar), leren de agenten vaak al na een relatief korte tijd een goede policy welke daarna langzaamaan beter wordt.

We hebben ook de combinatie van RL met exploratie methoden en functie approximatoren kort besproken. Huidig onderzoek in RL bekijkt hoe het opschalen van problemen gedaan kan worden door nog efficientere RL methoden, en welke functie approximatoren geschikt zijn voor het oplossen van bepaalde problemen. Ook wordt RL steeds vaker gebruikt in lerende multi−agent systemen. Dat RL een interessant gebied is met vele mogelijke nieuwe toepassingen, zal ertoe leiden dat steeds meer mensen RL als methode gaan gebruiken voor het oplossen van een bepaald probleem.

Referenties

Bellman, 57: R. Bellman, Dynamic Programming. Princeton University Press, 1957.

Crites and Barto, 96: R. Crites and A. Barto, Improving Elevator Performance using Reinforcement Learning. Advances in Neural Information Processing Systems 8, pp: 1017−1023, 1996.

Dorigo, 97: M. Dorigo and L. Gambardella, Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. Evolutionary Computation 1(1), pp: 53−66, 1997.

Kaelbling et al., 96: L. Kaelbling, M. Littman, and A. Moore, Reinforcement Learning: a Survey. Journal of Artificial Intelligence Research 4, pp: 257−285, 1996.

Littman and Boyan, 93: M. Littman and J. Boyan, A Distributed Reinforcement Learning Scheme for Network Routing. First International Workshop on Applications of Neural Networks to Telecommunication, pp: 45−51, 1993.

Moore and Atkeson, 93: A. Moore and C. Atkeson, Prioritized Sweeping: Reinforcement Learning with less Data and less Time. Machine Learning 13, pp: 103−130, 1993.

Peng and Williams, 96: J. Peng and R. Williams, Incremental Multi−step Q−learning. Machine Learning 22, pp: 283−290, 1996.

Sandholm, 95. T. Sandholm and R. Crites. On Multi−agent Q−learning in a Semi−Competitive Domain. IJCAI’95 workshop: Adaption and Learning in Multi−Agent Systems, pp: 164−176, 1995.

Sutton and Barto, 98: R. Sutton and A. Barto, Reinforcement Learning: an Introduction. MIT Press, 1998.

Tesauro, 92: G. Tesauro. Practical Issues in Temporal Difference Learning. Advances in Neural Information Processing Systems 4, pp: 259−266, 1992.

Watkins, 89: C. Watkins. Learning from Delayed Rewards, PhD thesis, King’s College, Cambridge, England, 1989.

Wiering and Schmidhuber, 98: M. Wiering and J. Schmidhuber, Efficient Model−based Exploration. Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats 6, pp: 223−228, 1998.

Wiering, 00: M. Wiering, Multi−agent Reinforcement Learning for Traffic Light Control. Seventeenth International Conference on Machine Learning, pp: 1151−1158, 2000.